Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Кафедра комплексной информационной безопасности
электронно-вычислительных систем (КИБЭВС)

Р.В. Мещеряков

Методы программирования

Методические указания
для студентов специальности «210202»

Томск
2007

УДК 681.3

Мещеряков Р.В. Методы программирования: Методические указания к лабораторным работам по курсу «Методы программирования» для студентов специальности «210202 - Конструирование и технология электронно-вычислительных средств».

ТУСУР. – Томск: Изд-во ТУСУР, 2007. – 237 с.

Методические указания содержат изложение структур данных, которые рассматриваются как исполнители с заранее известным набором команд. Для выполнения лабораторных работ используется информация по структурам данных и использование формального понятия «исполнитель» («вычислитель»). Приводится информация по структурам данных, реализованных в различных языках программирования.

Рассмотрены наиболее применимые прикладные алгоритмы: сортировка, поиск, работы с деревьями, хеширование.

Предназначено для студентов специальности «210202 - Конструирование и технология электронно-вычислительных средств» по курсу «Методы программирования»

© Мещеряков Р.В., 2007
© ТУСУР, 2007

ВВЕДЕНИЕ

Развитие электронно-вычислительной техники привело широкому применению языков программирования в области разработки программного обеспечения. Каждый язык программирования обладает своими особенностями, отличающими его от других. Однако есть среди них нечто общее: они оперируют данными посредством знаний, макросов, алгоритмов, процедур, функций и других команд.

Это привело к тому, что каждый программист разрабатывает при написании программ на основе представления заданных пользователем входных и выходных данных, генерирует свои собственные внутренние структуры данных, над которыми производит действия программа. Программа обрабатывает данные, используя заложенные разработчиком функции и процедуры обработки информации, ориентированные на использование определенных структур данных.

Для последующего использования терминов «информация» и «данные» введем определения:

Информация – любые сведения о каком-либо событии, сущности, процессе и т.д., являющемся объектом операций: восприятия, передачи, преобразования или использования.

Данные – информация, фиксированная в определенной форме, пригодной для последующей обработки, хранения и передачи.

Базы данных – описание конкретных фактов некоторой модели предметной области.

В дальнейшем будем рассматривать, во-первых, теоретические основы представления структур данных, которой посвящен раздел 1 и, во-вторых, наиболее часто используемые представления на известных языках программирования: Assembler, Basic, C/C++, Pascal, изложенные в разделе 2. В разделах прикладных алгоритмов рассмотрены основные алгоритмы, используемые при проектированиии информационных систем и являющихся базовыми.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ. СТРУКТУРЫ ДАННЫХ

Как можно заметить данные не могут существовать «сами по себе», т.к. при отсутствии возможности обрабатывать, накапливать и передавать они ничего не значат. Представим себе устройство, способное запоминать и хранить бесконечное количество информации, но при этом нет возможности получить ее из этого устройства.

Для этого при рассмотрении структур данных будем использовать понятие «*исполнитель*» («*вычислитель*») — человек, организация, механическое или электронное устройство, робот и т.п., умеющий выполнять некоторый вполне определенный набор действий.

Каждое действие, которое способен выполнить исполнитель, называется *предписанием*, а вся совокупность таких действий — *системой предписаний* исполнителя. Приказ на выполнение действия, выраженный формальным, заранее фиксированным способом называется вызовом предписания. Невозможность выполнения какого-либо действия приводит к аварийной ситуации, которое будем называть «отказом». Отказ исполнителя приводит к остановке исполнителя и генерирует в выходное устройство сигнал.

Таким образом можно определить структуры данных через исполнителей, работающих на любых, заранее определенных, совокупностях некоторых элементов одного и того же типа Е (например: целое, вещественное, символ, строковое и др. с ограничением — могут принимать только дискретные значения). Перечислим структуры данных: стек, очередь, дек, последовательность, множество, нагруженное множество, список, дерево, вектор, графы. Данное перечисление не ограничивается, на их основе можно строить более сложные и разветвленные структуры.

Содержание параграфа каждого исполнителя структуры данных содержит: систему предписаний, графическое представление, краткое описание работы.

1.1 Способы задания элементов определенного типа

Первым, наиболее простым и очевидным способом задания и конструирования типов элементов является **перечисление**. Для конкретного объекта задается область определения, перечисленная во множестве.

Например, множество дней недели {пн, вт, ср, чт, пт, сб, вс} или

месяцев в году {январь, февраль, март, апрель, май, июнь, июль, август, сентябрь, октябрь, ноябрь, декабрь}.

Кроме того, у объекта может быть значение *неопр.*, если он не равен ни одному из значений заданного множества. На перечисления могут быть наложены свойства упорядоченности по определенному критерию. Так, на множествах дней недели можно задать несколько критериев:

- {пн, вт, ср, чт, пт, сб, вс} – обычный ("русский") порядок дней недели;
- {вс, пн, вт, ср, чт, пт, сб} – порядок дней недели, принятый в некоторых странах;
- {вс, сб, пт, чт, ср, вт, пн} – обратный порядок дней недели;
- {вс, вт, сб, ср, пн, пт, чт} – по алфавиту.

Данный неполный список можно продолжать, выбирая требуемый критерий упорядоченности. Дополнительно, при задании упорядоченного перечисления, объекту могут быть назначены значения +∞ или -∞. Значения объекта будем считать упорядоченными в порядке их перечисления:

-∞ < пн < вт < ср < чт < пт < сб < вс < +∞.

Вторым способом задания и конструирования типов элементов является **отрезок**. Отрезком задается границами в упорядоченном перечислении и, следовательно, область определения. Например, номера дней в месяце могут принимать значения от 1 до 31: 1..31, или рабочие дни недели: пн..пт. Аналогично задаются значения *неопр.*, +∞ и -∞.

Способ конструирования **запись** позволяет объединить несколько объектов в один, более сложный. Например, можно определить запись типа "дата" нашей эры заданием следующих объектов:

Наименование	Диапазон	Примечание
год	{0..+∞}	Реально можно ограничиться определенным значением, например, 2100 г.
месяц	{январь, февраль, март, апрель, май, июнь, июль, август, сентябрь, октябрь, ноябрь, декабрь}	Возможно однозначное соответствие номеру месяца в году {1..12}
день	1..31	Необходимы дополнительные ограничения в зависимости от месяца
день недели	{пн, вт, ср, чт, пт, сб, вс}	Возможно однозначное соответствие номеру {1..7}

Например задание даты: "2001.05.01.2" — 1 мая 2001 года, вторник.

Далее, при рассмотрении исполнителей, будем использовать выражение "структура данных элементов типа E", подразумевая этим, что каждый элемент заданного исполнителя оперирует произвольным типом, сконструированным по изложенным правилам.

1.2 Стек элементов типа Е

Система предписаний, записанная формально:

1.	начать работу	
2.	сделать *стек* пустым	
3.	*стек* пуст/*стек* - не пуст	:да/нет
4.	добавить элемент <вх : Е> *в стек*	
5.	взять *элемент из стека в* <вых : Е>	
6.	вершина *стека*	::Е
7.	удалить вершину *стека*	
8.	кончить работу	

Курсивом показаны части имени предписания, которые можно опускать. Таким образом, «Стек.добавить <Е>», «Стек.добавить элемент <Е> в стек», «Стек.добавить элемент <Е>» и «Стек.добавить <Е> в стек» означают одно и то же. На рисунке 1 приводится структура стека.

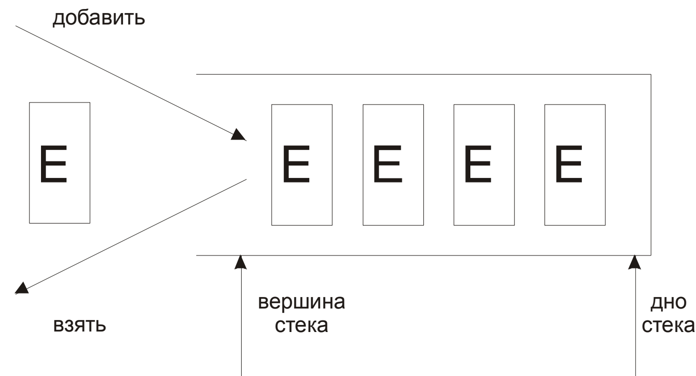


Рисунок 1– Структура «Стек»

По предписанию «начать работу» стек начинает работу. После выполнения этого предписания, как и после предписания «сделать пустым», стек пуст, т.е. не содержит ни одного элемента. Предписания «пуст»/«не пуст» анализируют, есть ли в стеке элементы, и вырабатывают в качестве ответа соответствующее значение — либо «да», либо «нет». Предписания «добавить» и «взять» выполняются в соответствии со структурой стека: по предписанию «взять» из стека изымается элемент, который был добавлен последним, и его состояние присваивается фактическому выходному параметру. Предписание «удалить вершину» отличается от предписания «взять» только тем, что состояние изымаемого элемента (вершины стека) никуда не попадает, а безвозвратно теряется.

Наконец, запись ::Е в предписании «вершина» означает, что это предписание в зависимости от вида использования *либо принимает, либо*

вырабатывает значение типа Е. Как и всякое предписание, вырабатывающее значение, «вершину» можно использовать в выражениях и в качестве фактического входного параметра. Кроме того, это предписание можно использовать и в левой части оператора присваивания, и в качестве выходных и входно-выходных фактических параметров. Например, можно написать «Стек.вершина := 0» или «Стек.вершина := Стек.вершина + 1».

Таким образом, запись «Стек.вершина» можно рассматривать как обозначение для объекта-вершины стека. Заметим, что при этом используется или меняется только состояние вершины стека, а число элементов в стеке не изменяется. Естественно, что если в стеке нет элементов, то предписания «взять», «удалить» и «вершина» приводят к отказу.

1.3 Очередь элементов типа Е

Система предписаний:

1. начать работу	
2. сделать *очередь* пустой	
3. *очередь пуста/очередь* не пуста	: да/нет
4. добавить *элемент* <вх : Е> в *конец очереди*	
5. взять *элемент из начала очереди в* <вых : Е>	
6. начало *очереди*	::Е
7. удалить начало *очереди*	
8. кончить работу	

На рисунке 2 приводится структура очереди.

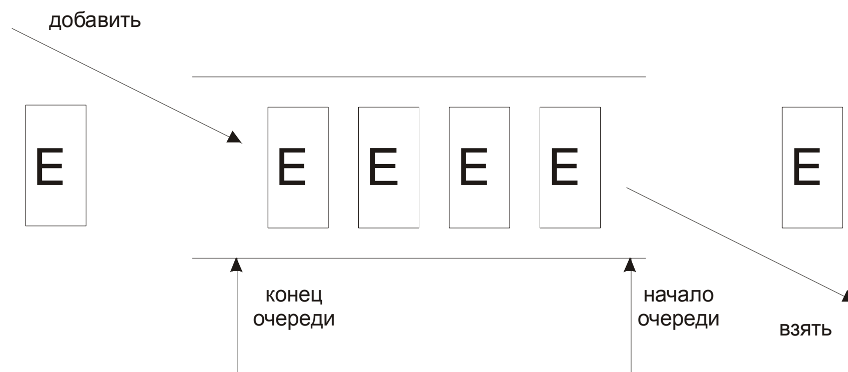


Рисунок 2 – Структура «Очередь»

Очередь является аналогом стека, с той лишь разницей, что добавление элементов производится в конец очереди (на рисунке 2 слева), а операции: взять, посмотреть, изменить или удалить можно лишь начало (на рисунке 2 справа). Одним из распространенных названий стека является «стек типа LIFO» (LIFO — Last Input First Output — последним пришел, первым ушел), а названием очереди – «стек типа FIFO» (LIFO — First Input First Output — первым пришел, первым ушел).

1.4 Дек элементов типа E

Система предписаний:

1. начать работу
2. сделать *дек* пустым
3. *дек* пуст/*дек* не пуст : да/нет
4. добавить *элемент* <вх : E> в начало/конец *дека*
5. взять *элемент* из начала/конца *дека в* <вых : E>
6. качало/конец *дека* ::E
7. удалить начало/конец *дека*
8. кончить работу

Таким образом, если работать с деком только с левого края («добавить в начало», «взять из начала», «начало», «удалять начало»), то фактически получается структура стека. Аналогично мы получим стек, если будем работать только с правого края (в другой нотации «конца дека» или «вершины дека»). Пользуясь предписаниями «добавить в конец» и «взять из начала», мы можем получить очередь, элементы которой будут продвигаться справа налево. С помощью противоположной пары предписаний можно получить очередь, элементы которой движутся слева направо. В целом же эта структура называется деком (сокращение от английского double ended queue — очередь с двумя концами) и представлена на рисунке 3.

Рисунок 3 – Структура «Дек»

Как видно из рисунка 3 дек напоминает два стека, соединенных своими днами. С другой стороны дек напоминает две очереди, которые работают в обратных друг другу направлениях над одним и тем же объемом данных.

1.5 Множество элементов типа Е

Система предписаний:

1. начать работу	
2. сделать *множество* пустым	
3. *множество пусто/множество* не пусто	: да/нет
4. добавить *элемент* <вх : Е> *в множество*	
5. удалить *элемент* <вх : Е> *из множества*	
6. *элемент* <вх : Е> принадлежит *множеству*	: да/нет
7. взять *какой-нибудь элемент множества в* <вых : Е>	
8. кончить работу	

В отличие от всех предыдущих структур множество является структурой, элементы которой не упорядочены. Предписание «взять», например, выдает какой-то элемент множества. Был ли он добавлен первым, вторым, n-м или последним, сказать нельзя. Как обычно, по предписанию «взять» элемент из множества забирается «материально», т.е. после этого предписания взятого элемента в множестве не будет. Предписания «добавить» и «удалить» следует понимать в теоретико-множественном смысле: если добавляемый элемент уже был в множестве или если удаляемого элемента в множестве нет, то отказа не возникает, а множество не изменяется.

Предписание «элемент <вх : Е> принадлежит множеству» берет состояние фактического параметра и проверяет, есть ли такой элемент в множестве. Множество при этом никак не меняется.

1.6 Нагруженное множество

Система предписаний:

1. начать работу	
2. сделать *множество* пустым	
3. *множество* пусто/*множество* не пусто	: да/нет
4. добавить *элемент* <вх : E> в *множество*	
5. удалить *элемент* <вх : E> *из множества*	
6. *элемент* <вх : B> принадлежит *множеству*	: да/нет
7. взять *какой-нибудь элемент множества в* <вых : E>	
8. нагрузка **<**вх : E>	:: Y
9. кончить работу	

Нагруженное множество отличается от обычного тем, что каждый элемент можно «нагрузить» — сопоставить ему некоторое значение определенного типа **Y**. Это делается с помощью предписания «нагрузка», принимающего и вырабатывающего значения типа **Y**. Если этим предписанием не пользоваться, то внешне нагруженное множество неотличимо от обычного. Запись

М.нагрузка **<вх** : е>: = у

устанавливает нагрузку элемента **е** множества **М** в состояние **у** независимо от того, какова была нагрузка этого элемента раньше. Если в предписании «нагрузка» указан элемент, не принадлежащий множеству, то возникает ситуация **отказ.** После добавления элемента к множеству независимо от того, был уже такой элемент в множестве или нет, была ли у него нагрузка определена или нет, состояние одинаково — новый элемент принадлежит множеству и его нагрузка имеет состояние **неопр.**

1.7 Последовательность элементов типа Е

Система предписаний:

1. начать работу	
2. сделать *последовательность* пустой	
3. *последовательность* пуста/не пуста	: да/нет
4. добавить *элемент* <вх : E> *в конец последовательности*	
5. встать в начало *последовательности*	
6. есть/нет непрочитанные *элементы*	: да/нет
7. прочесть *очередной элемент последовательности в* **<**вых : E>	
8. очередной *элемент последовательности*	::E
9. пропустить *очередной элемент последовательности*	
10. кончить работу	

Совокупность элементов, с которыми работает этот исполнитель, является линейно упорядоченной. Элементы последовательности в каждый момент времени разделены на две части — прочитанную и непрочитанную (рисунок 4).

После предписаний «начать работу» и «сделать пустой» прочитанная и непрочитанная части последовательности пусты (не содержат ни одного элемента). По предписанию «добавить <вх : Е> в конец» элемент добавляется в конец последовательности (непрочитанная часть при этом увеличивается, а прочитанная не изменяется).

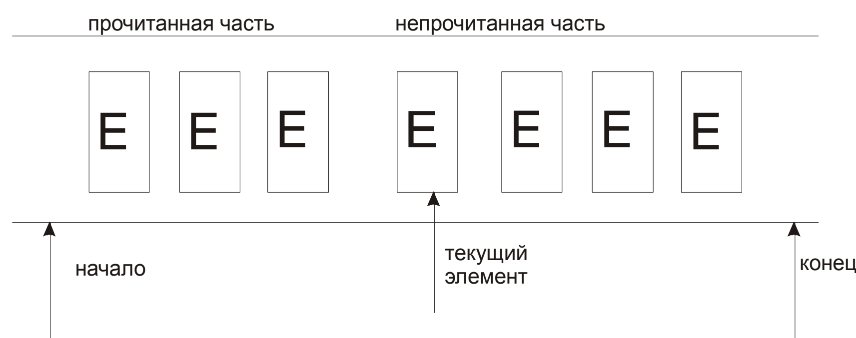


Рисунок 4 – Структура «Последовательность»

По предписанию «встать в начало» прочитанная часть делается пустой, а непрочитанная совпадает со всей последовательностью. Предписание «есть непрочитанные элементы» отвечает “да”, если в непрочитанной части есть элементы, и “нет”, если эта часть пуста.

Непрочитанная часть последовательности является аналогом очереди. Очередным элементом (аналог начала очереди) называется первый элемент непрочитанной части. По предписаниям «прочесть», «пропустить» (аналоги «взять» и «удалить» для начала очереди) очередной элемент перемещается из непрочитанной части в прочитанную (т.е. прочитанная часть увеличивается на этот элемент, непрочитанная уменьшается и очередным становится следующий элемент последовательности).

1.8 Л2-список элементов типа Е

Система предписаний:

1. начать работу 2. сделать *список* пустым 3. *список* пуст/список не пуст : да/нет 4. установить *указатель* в начало/конец *списка*

5. *указатель* в начале/конце *списка* : да/нет
6. *передвинуть указатель списка* вперед/назад
7. добавить *элемент* <вх : E> до *указателя*/за *указателем списка*
8. взять *элемент списка* до *указателя*/за *указателем в* <вых : E>
9. *элемент списка* до *указателя*/за *указателем* ::E
10. удалить *элемент списка* до *указателя*/за *указателем*
11. кончить работу

Л2-список — линейный двунаправленный список — можно представлять себе в виде перекидного календаря или своего рода бус из элементов типа E (рисунок 5).

Элементы списка линейно упорядочены. Положение перед первым элементом списка называется *началом* списка, положение за последним элементом называется *концом* списка. Кроме элементов в списке имеется еще *указатель,* который можно представлять себе стрелочкой, расположенной в начале, в конце или между элементами списка (в перекидном календаре указатель — это место, на котором раскрыт календарь).

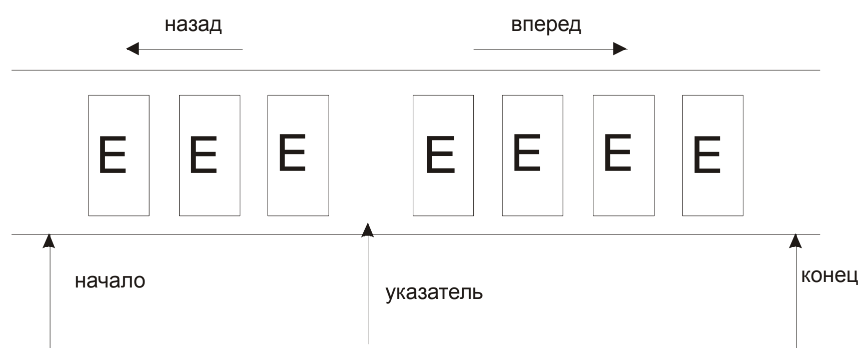


Рисунок 5 – Структура «Л2-список»

После предписаний «начать работу» и «сделать список пустым» список становится пустым, т.е. не содержит элементов. Для пустого списка понятия «начало» и «конец» существуют и совпадают, а указатель расположен одновременно и в начале, и в конце списка (рисунок 5).

По предписанию «передвинуть указатель вперед/назад» указатель смещает один элемент соответственно вперед (от начала к концу списка), либо назад (от конца к началу).

Если пользоваться только предписаниями «добавить» / «взять» / «элемент» / «удалить», а сам указатель не двигать, то части списка до указателя и за указателем представляют собой полные аналоги стеков.

Название списка «линейный двунаправленный» означает, что элементы списка упорядочены линейно, а указатель можно двигать и вперед, и назад.

1.9 Л1-список элементов типа Е

Система предписаний:

№	Предписание	
1.	начать работу	
2.	сделать *список* пустым	
3.	*список* пуст/список не пуст	: да/нет
4.	установить *указатель* в начало *списка*	
5.	*указатель* в конце *списка*	: да/нет
6.	*передвинуть указатель списка* вперед	
7.	добавить *элемент* <вх : Е> за *указателем списка*	
8.	взять *элемент списка* за *указателем* в <вых : Е>	
9.	*элемент списка* за *указателем*	:: Е
10.	удалить *элемент списка* за *указателем*	
11.	кончить работу	

Л1-список — линейный однонаправленный список — отличается от Л2-списка тем, что он обеспечивает последовательный просмотр элементов списка только от начала к концу (встать в начало, пока не в конце, передвинуть вперед). Передвинуть указатель назад нельзя. Кроме того, в Л1-списке работать (добавлять / брать / получать / менять / удалять элементы) можно только за указателем, т.е. впереди по ходу движения. Образно говоря, в Л1-списке только одна «сторона» указателя является активной.

1.10 Вектор элементов типа Е с произвольным доступом

Система предписаний:

№	Предписание	
1.	начать работу	
2.	*элемент вектора* с индексом <вх : И>	:: Е
3.	кончить работу	

Вектор представляет собой совокупность элементов типа Е, "пронумерованную" значениями типа И (номер элемента называется его *индексом*). Особенностью данной структуры является произвольный

доступ к элементам по индексу И. После предписания "начать работу" все элементы вектора существуют и все имеют состояние **неопр.**

Вектор элементов типа Е с произвольным доступом обеспечивает наиболее мощное, по сравнению с предыдущими структурами данных. На данной структуре можно реализовать все предыдущие.

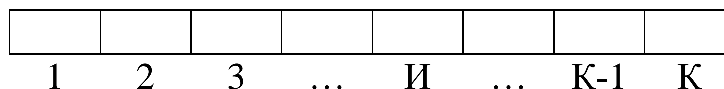


Рисунок 6 – Структура «Вектор элементов»

1.11 Матрица элементов типа Е с индексами типа И1, И2.

Система предписаний:

1. начать работу
2. *элемент матрицы с индексами* <вх : И1,И2> :: Е
3. кончить работу

Матрица является полным аналогом вектора с той лишь разницей, что индексов у нее два. Соответственно представлять матрицу надо не в виде линейной последовательности элементов, а в виде прямоугольной таблицы.

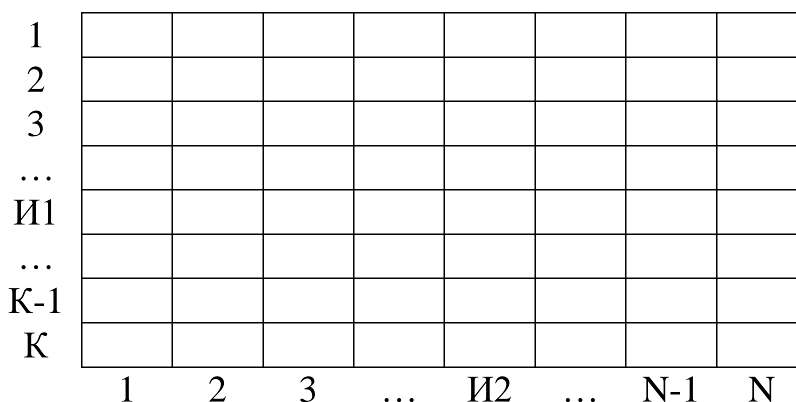


Рисунок 7 – Структура «Матрица элементов»

Аналогично можно построить матрицу с тремя и более индексами. С другой стороны, при установлении однозначного соответствия между

индексами и натуральными целыми числами, начинающихся с нуля, можно реализовать матрицу большей размерности на векторе. В этом случае индексы будут пересчитываться по определенным правилам. Так для матрицы с двумя индексами правила пересчета индексов могут быть следующими:

И1=целая часть (И/МАКС(И1))

И2=остаток (И/МАКС(И1))

Подобным образом можно увеличивать размерность матрицы.

1.12 Динамический вектор элементов типа Е

Система предписаний:

1.	начать работу	
2.	сделать *вектор* пустым	
3.	*вектор* пуст / *вектор* не пуст	: да/нет
4.	число *элементов вектора*	: Z+
5.	добавить *элемент* <вх : Е> *в конец вектора*	
6.	удалить *элемент из конца вектора*	
7.	*элемент вектора с индексом* <вх : И>	:: Е
8.	кончить работу	

Динамический вектор отличается от обычного только тем, что:

- индекс у него меняется всегда от 1 (0) до числа элементов в векторе;
- число элементов в векторе может меняться динамически (т.е. в процессе работы) изменяться.

После "начать работу", как и после "сделать вектор пустым", динамический вектор пуст, т.е. не содержит ни одного элемента. Предписания "добавить" и "удалить" добавляют или удаляют элемент в конце вектора (т.е. справа, если рисовать вектор в порядке возрастания индексов).

Следующие две структуры данных отличаются от предыдущих, т.к. имеют не только более сложную структуру, но и их исполнители могут иметь различную конфигурацию, следовательно, систему предписаний.

1.13 Графы

Граф G=(V, E) состоит из конечного непустого множества *узлов* V и множества *ребер* E. Если ребра представлены в виде упорядоченных пар (v, w) узлов, то граф называется *ориентированным*; v называется *началом*,

а w — *концом* ребра (v, w). Если ребра — неупорядоченные пары (множества) различных вершин, также обозначаемые (v, w), то граф называют *неориентированным*.

Путем в ориентированном и неориентированном графе называют последовательность ребер вида (v_1, v_2), (v_2, v_3), …, (v_{n-1}, v_n). Говорят, что этот путь *идет из* v_1 в v_n и имеет длину n-1. Часто такой путь представляют последовательностью v_1, v_2, v_3, …, v_{n-1}, v_n узлов, лежащих на нем. В вырожденном случае один узел обозначает путь длины 0, идущий из этого узла в него же. Путь называется *простым*, если все ребра и все узлы на нем, кроме, быть может, первого и последнего, различны. *Цикл* — это простой путь длины не менее 1, который начинается и кончается в одном и том же узле. Заметим, что в неориентированном графе длина цикла должна быть не менее.

Известно несколько представлений графа G=(V, E).

Один из них — *матрица смежностей*, т.е. матрица А размера ||V||x||V||, состоящая из 0 и 1, в которой A[i, j]=1 тогда и только тогда, когда есть ребро из узла i в узел j, либо их количество. Представление в виде матрицы смежностей удобно для тех алгоритмов на графах, которым часто нужно знать, есть ли в графе данное ребро, ибо время, необходимое для определения наличия ребра, фиксировано и не зависит от ||V|| и ||E||. Основной недостаток применения матрицы смежностей заключается в том, что она занимает память объема $||V^2||$ даже тогда, когда граф содержит 0(||V||) ребер.

Второй способ представления графа — в виде двоичных векторов строк и столбцов матрицы смежности. Таким образом перечисляются все ребра графа. Такое представление может способствовать значительной эффективности алгоритмов на графах.

Третьим способом представления графа является возможность составления списков. *Списком смежностей* для узла v называется список всех узлов w, смежных с v. Граф можно представить с помощью ||V|| списков смежностей, по одному для каждого узла.

Заметим, что представление графа в виде списков смежностей требует памяти порядка ||V||+||E||. Представлением с помощью списков смежностей часто пользуются, когда $||E|| << ||V||$.

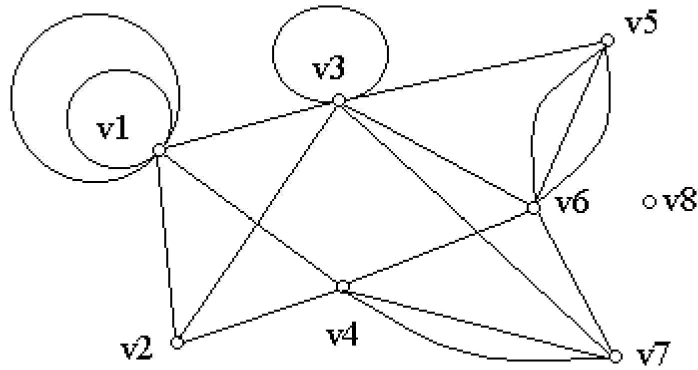


Рисунок 8 – Структура «Граф»

	v1	v2	v3	v4	v5	v6	v7	v8
v1	2	1	1	1	0	0	0	0
v2	1	0	1	1	0	0	0	0
v3	1	1	1	0	1	1	1	0
v4	1	1	0	0	0	1	2	0
v5	0	0	1	0	0	3	0	0
v6	0	0	1	1	3	0	1	0
v7	0	0	1	2	0	1	0	0
v8	0	0	0	0	0	0	0	0

а) матрица смежностей

вершины {v1, v2, v3, v4, v5, v6, v7, v8};
ребра {(v1,v1), (v1,v1), (v1,v2), (v1,v3), (v1,v4), (v2,v3), (v2,v4), (v3,v3), (v3,v5), (v3,v6), (v3,v7), (v4,v6), (v4,v7), (v5,v6), (v5,v6), (v5,v6), (v6,v7)}

б) список вершин и ребер графа

v1	v1, v1, v2, v3, v4
v2	v1, v3, v4
v3	v1, v2, v3, v5, v6, v7
v4	v1, v2, v6, v7, v7
v5	v3, v6, v6, v6
v6	v2, v4, v5, v5, v5, v7
v7	v3, v4, v4, v6
v8	

в) список смежностей графа

Рисунок 9 – Способы представления структуры данных «Граф»

1.14 Деревья

Ориентированный граф без циклов называется *ориентированным ациклическим графом*. (*Ориентированное*) *дерево* (иногда его называют *корневым деревом*) — это ориентированный ациклический граф, удовлетворяющий следующим условиям:

1) имеется в точности один узел, называемый *корнем*, в который не входит не одно ребро;
2) в каждый узел, кроме корня входит одно ребро;
3) из каждого корня к каждому узлу идет путь (который, как легко показать, единствен).

Глубина узла v в дереве — это длина пути из корня в v. *Высота узла* v в дереве – это длина самого длинного пути из v в какой-нибудь лист. *Высотой дерева* называется высота его корня. *Уровень* узла v в дереве равен разности высоты дерева и глубины узла v.

Упорядоченным деревом называется дерево, в котором множество сыновей каждого узла упорядочено. При изображении упорядоченного дерева мы будем считать, что множество сыновей каждого узла упорядочено слева направо. *Двоичным (бинарным) деревом* называется такое упорядоченное дерево, что

1) каждый сын произвольного узла идентифицируется либо как *левый сын*, либо как *правый сын*;
2) каждый узел имеет не более одного левого сына и не более одного правого сына.

Двоичное дерево называется *полным*, если для некоторого целого числа k каждый узел глубины меньшей k имеет как левого, так и правого сына и каждый узел глубины k является листом. Полное двоичное дерево высоты k имеет ровно 2^{k+1}-1 узлов.

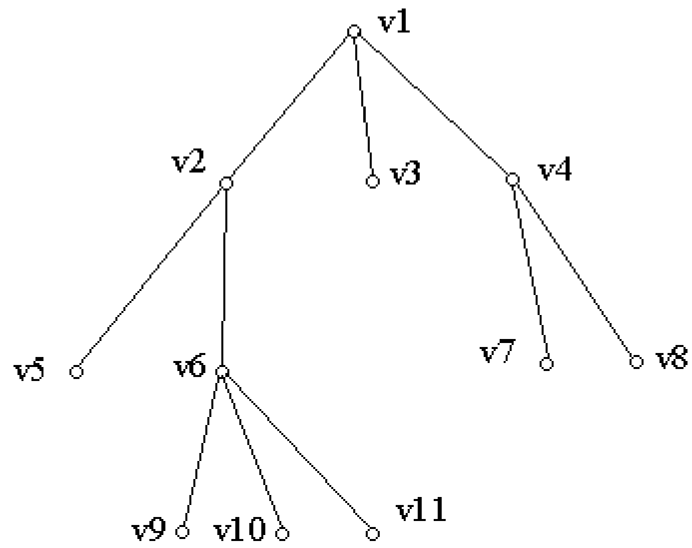
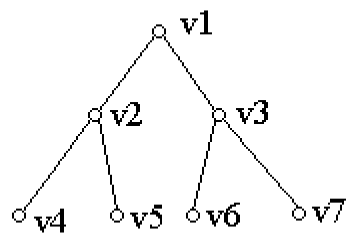


Рисунок 10 – Структура «Дерево»



v1(v2(v4(), v5()), v3(v6(), v7()))

Рисунок 11 – Способ представления «Двоичного дерева»

2 ПРАКТИЧЕСКАЯ ЧАСТЬ. СТРУКТУРЫ ДАННЫХ

2.1 Общие сведения.

Современные языки программирования используют вектор конечного размера. На векторе с произвольным доступом к элементам могут быть реализованы все структуры данных. Следовательно, можно в явном виде создать программу, моделирующую работу исполнителя (вычислителя). Некоторые реализации имеют особенности, поэтому будем придерживаться наиболее общих положений. Отметим также, что все приводимые сведения используют в основе своей архитектуру компьютера IBM PC.

Для дальнейшего изложения опишем некоторые общие принципы (приведенное краткое изложение не претендует на полноту, а введено для напоминания сведений, пройденных в других курсах.):

Минимальной единицей информации является бит (bit) — который может принимать значение 0 или 1.

Минимальной адресуемой ячейкой памяти примем байт (byte) — состоит из набора 8 бит, в двоичном представлении минимально число — 00000000, максимальное — 11111111, в десятичной системе счисления — 0 и 255 соответственно, часто используют шестнадцатеричную запись — 00h и FFh. Иногда используют восьмеричную запись, но в настоящее время такой тип записи встречается редко, поэтому использовать его не будем.

X	X	X	X	X	X	X	X
7	6	5	4	3	2	1	0

где X – 0 или 1

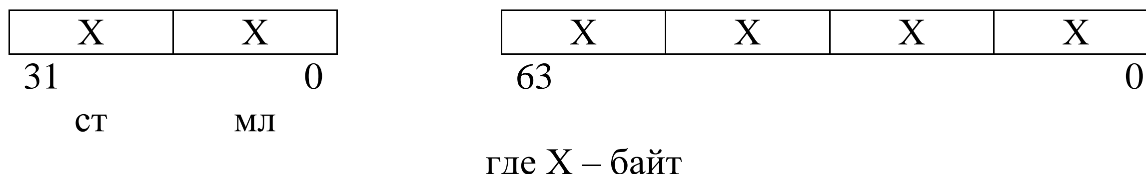
Рисунок 12 – Хранение байта

Слово (word) — это два соседних байта. Размер слова — 16 разрядов. Оно может содержать от 0000h до FFFFh, что соответствует 0 и 65535 десятичному, либо, если один разряд используется как знаковый от –32768 до +32767.

Двойное слово (double word) — это четыре соседних байта или, что то же самое, два соседних слова. Размер двойного слова — 32 бита. Таким образом двойным словом можно представить диапазон чисел от 00000000h до FFFFFFFFh, то есть от 0 до 4294967295, либо от –2147483648 до +2147483647

Как правило, для хранения числа используется обратный порядок хранения — сначала идет младший байт, затем старший. Данная особенность необходима была для первых моделей компьютеров, когда процессор был 8-разрядным, т.к. для сложения необходимо было сначала

выбрать младший байт, затем старший и произвести вычисление. Однако в регистрах числа размером в слово и более хранятся в "нормальном" виде. Так, например число 12345678h будет храниться в последовательности 78 56 34 12.



где X – байт

Рисунок 13 – Хранение слова и двойного слова

Отрицательные числа хранятся в дополнительном коде: $2^k-|x|$.

Специальным способом хранятся двоично-десятичные числа (binary coded decimal, BCD-числа), строящиеся по следующему принципу: берется десятичная запись числа и каждая его цифра заменяется на четыре двоичные цифра (от 0000 до 1001), обозначающие эту цифру в двоичной системе. Например, число 128 будет представлено так: 0001 0010 1000. Достоинством двоично-десятичных чисел является то, что они практически не требуют перевода из десятичной системы. Чаще всего данное представление используется для решения класса характерных задач, чаще всего коммерческих, в которых требуется производить применение к большому массиву числовых данных одно-двух арифметических операций и выводом также большого количества результатов, таким образом, большое количество машинного времени тратится на перевод из одной системы счисления в другую. Недостаток же этого представления заключается в том, что имеющиеся в ПК команды позволяют выполнять арифметические операции только над однозначными двоично-десятичными числами, операции же над многозначными числами приходится реализовывать программным способом, что долго. В то же время операции над многозначными двоичными числами реализуются аппаратно, а поэтому выполняются быстрее.

Вещественные числа хранятся, как правило, в экспоненциальном представлении: мантисса и порядок. В зависимости от точности представления отводится определенное число байт.

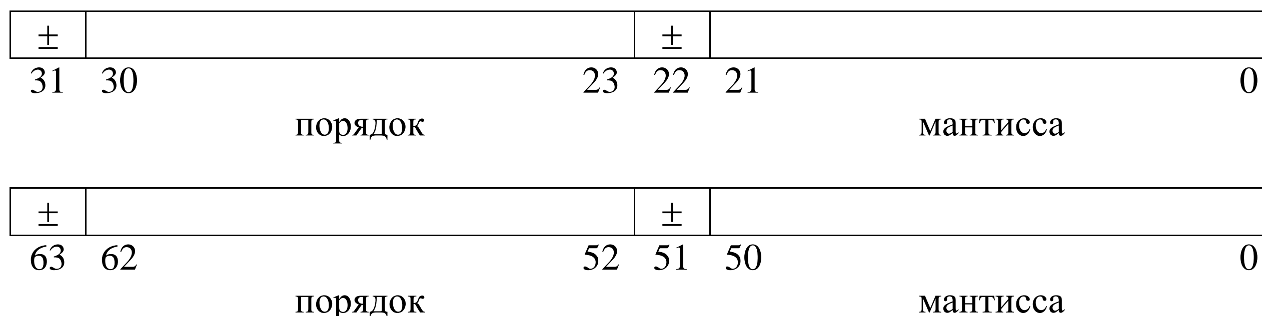


Рисунок 14 – Хранение вещественных чисел.

Представление символьных данных. Как и любая другая информация символьные данные должны храниться двоичном виде. Для этого каждому символу ставится в соответствие некоторое неотрицательное число, называемое кодом символа, и это число хранится в ПК в двоичном виде. Конкретное соответствие между символами и их кодами называется системой кодировки. В ЭВМ, как правило, используются 8-разрядные коды символов. Это позволяет закодировать 256 различных символов, чего вполне достаточно для представления многих символов, используемых на практике. Поэтому для кода символа достаточно выделить один байт.

В ПК обычно используется система кодировки ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией). В ней не предусмотрены коды для букв русского алфавита. Поэтому используются модифицированные системы кодировок, включающие буквы русского алфавита. Наиболее известные системы кодировок: "Альтернативная кодировка ГОСТ", "КОИ-8" (код обмена информацией – 8 бит), CP-1251 (Windows-кодировка), ISO 8859-5 (Unix-кодировка). Данный неполный перечень можно продолжать, но это заняло бы слишком много места. Отметим, что существует универсальная кодировка Unicode, под каждый символ которой отводится два байта, следовательно, можно закодировать 65536 символов. В данной универсальной кодировке хранятся большинство алфавитов мира, а также основные японские и китайские иероглифы.

Для хранения строковых данных можно выделить три способа хранения:

1. Строка со счетчиком — хранится собственно последовательность символов и текущая длина строки (счетчик), а также адрес начала строки. Длина ограничена размером счетчика.
2. Строка с признаком конца строки – храниться последовательность символов, заканчивающаяся определенным символом — "концом строки", чаще всего ноль. Другое название ASIIZ-строка. Кроме того, также хранится адрес начала строки.

3. Строка с хранением адреса начала и конца строки. Является модифицированным способом хранения первого типа. Вместо текущей длины хранится адрес конца строки.

Важной особенностью всех языков программирования является разделение данных на константы и переменные. В зависимости от этого и типа транслятора (компиляция или интерпретация) по-разному производятся преобразования над различными типами данных.

2.2 Ассемблер (Assembler)

Директивы определения данных

Место для хранения переменных отводится директивами определения данных. Одна из них предназначена для описания данных размером в байт, вторая — для описания данных размером в слово, а третья — описания данных размером в двойное слово. В остальном директивы практически не отличаются друг от друга.

Директива DB (db)

По директиве DB (define byte, определить байт) определяются данные размером в байт. Ее синтаксис такой

[<имя>] db <операнд> {,<операнд>}

Встречая такую директиву, ассемблер вычисляет операнды и записывает их значения в последовательные байты памяти. Первому из этих байтов дается указанное имя, по которому на этот байт можно ссылаться из других мест программы.

Существует два основных способа задания операндов директивы db:

- ? (знак неопределенного значения)
- константное выражение со значением от –128 до 255.

Остальные способы задания операндов производны от них.

Операнд ?

Возможный пример:

X DB ?

По этой директиве описывается переменная X. Для нее отводится один байт памяти, в который ничего не записывается (в этом байте будет то, что осталось от предыдущей программы, и предугадать, что там находится нельзя). В этом случае говорят, что переменная не получает начального значения.

Операнд – константное выражение со значением от –128 до 255

Язык ассемблер позволяет описывать переменные с начальными значениями. Для этого в качестве операнда директивы DB указывается выражение, которое ассемблер вычислит и значение которого запишет в ячейку, отведенную под переменную. Это и есть начальное значение переменной. Позже при выполнении программы, его можно будет и изменить, можно будет что-то записать в эту ячейку, но к началу выполнения программы в этой ячейке уже будет находиться значение.

В простейшем и наиболее распространенном случае начальное значение байтовой переменной задается в виде числа с величиной от –128 до 255. Например:

A DB 254 ; 0FEh

B DB –2 ; 0FEh (=256-2=254)

C DB 17h ; 17h

По каждой из этих директив ассемблер отводит один байт под переменную и записывает в этот байт указанное число. Таким образом, к началу выполнения программы переменная A будет иметь значение 254, B — значение –2, а переменная C — значение 17h.

В другом распространенном случае в качестве начального значения переменной указывается символ. Такое значение можно задать двояко: либо указать числовой код символа, либо указать сам символ в кавычках. Например, в системе кодировки ASCII код символа "*" равен 2Ah, поэтому следующие две директивы эквивалентны.

S DB 2Ah

S DB "*"

Ясно, что в ячейке памяти будет храниться одно и то же число – 2Ah, но второй способ записи нагляднее и не требует знания кода символа.

Директива с несколькими операндами

Запись директивы DB с одним операндом удобна, когда надо описать скалярную переменную, но неудобно, когда надо описать переменную-массив. Если надо описать массив из 5 байтов с некоторыми начальными значениями, то это можно сделать так:

```
M	DB 5
	DB –5
	DB ?
	DB "*"
	DB 34h
```

Отметим, что имя дается только первому элементу, а остальные оставляются безымянными. В данном случае имя указано в первой директиве. Ясно, что, если в массиве много элементов, то такой способ описания массива слишком громоздок. Поэтому допускается упрощенная форма описания массивов, когда он описывается одной директивой, но с

несколькими операндами — со столькими, сколько элементов в массиве. Например, для написанного выше примера можно написать только одну директиву:

M	DB 5, -5, ?, "*", 34h

Операнд-строка

Возможно еще одно сокращение в директиве DB: если в ней несколько соседних операндов – символы, то их можно объединить в одну строку. Например, следующие три директивы эквивалентны:

S	DB 'a', 'b', 'c'

S	DB 'abc'

S	DB 'a'

	DB 'b'

	DB 'c'

Вопрос о том, объединять ли соседние символы в одну строку или нет, а если объединять, то какие именно, решает сам автор программы. Например, записанную ранее директиву можно записать и так:

S	DB 'ab', 'c'

S	DB 'a', 'bc'

Операнд – конструкция повторения DUP

Рассмотрим еще одно сокращение в записи директивы DB. Довольно часто в директиве приходится указывать одинаковые операнды. Например, если мы хотим описать байтовый массив R из 8 элементов с начальными значениями 0 для каждого из них, то это можно сделать так:

R	DB 0, 0, 0, 0, 0, 0, 0, 0

Эту директиву можно записать так:

R	DB 8 DUP (0)

Здесь в качестве операнда использована конструкция повторения, в которой сначала указывается коэффициент повторения, затем служебное слово DUP (dublicate, копировать), а за ним в круглых скобках — повторяемая величина.

В общем случае эта конструкция имеет следующий вид:

k DUP (p_1, p_2, …, p_n)

Где k – константное выражение с положительным значением, n>=2, p_i — любой допустимый операнд директивы DB (в частности, это может быть снова конструкция повторения). Данная запись является сокращением для k раз повторений последовательности, указанных в скобках операндов.

Например, директивы слева эквивалентны директивам справа:

X DB 2 DUP ('ab', ?, 1) X DB 'ab', ?, 1, 'ab', ?, 1

Y DB –7, 3 DUP (0, 2 DUP (?)) Y DB –7, 0, ?, ?, 0, ?, ?, 0, ?, ?

Отметим, что вложенность конструкций DUP можно использовать для наглядного описания многомерных массивов. Например, директиву

A DB 20 DUP (30 DUP (?))

можно рассматривать как описание байтовой матрицы A размера 20x30, в которой элементы расположены в памяти следующим образом: первые 30 байтов — это элементы первой строки матрицы, следующие 30 байтов — элементы второй строки и т.д.

Директива DW (dw)

Директивой DW (define word, определить слово) описываются переменные размером в слово. Она аналогична директиве DB, поэтому лишь вкратце рассмотрим допустимые виды ее операндов.

Операнд ?

Возможный пример:

X DW ?

По этой директиве ассемблер отводит под переменную X слово памяти, в которое ничего не записывает, т.е. эта переменная не получает начального значения.

Константное выражение со значением от –32768 до 65535

Возможные примеры:

A DW 1234h

B DW –2

По этим директивам под переменные A и B отводится по слову памяти и в эти ячейки записываются указанные числа, которые становятся начальными значениями этих переменных.

Как и в случае директивы DB, неотрицательные числа записываются в память как числа без знака, а отрицательные числа — в дополнительном коде. Но здесь имеется отличие от директивы DB. На языке ассемблер числа записываются в обычном виде, а в памяти компьютера они хранятся

в обратном порядке, поэтому по этим директивам в памяти будут храниться 4 байта в следующем порядке: 34h 12h FEh FFh.

Частным случаем рассматриваемого вида операнда директивы DW может быть строка из одного или двух символов, например:

S1 DW '01'

S2 DW '2'

Если указана строка из двух символов, тогда ассемблер берет коды указанных символов (в нашем случае — 30h (код '0') и 31h (код '1')) и образует слово 3031h, которое и считается начальным значением описываемой переменной S1. Но как и любое число размером в слово, данное значение будет записано в память в обратном порядке. Если же в правой части директивы DW указан один символ, тогда к нему слева приписывается символ с кодом 0 и дальнейшие действия ассемблера будут такими же, как и в случае двухсимвольной строки. Поэтому по этим двум директивам память будет заполнена следующим образом: 31h 30h 32h 00h.

В связи с тем, что операнды-строки записываются в память в обратном порядке, что не характерно для строк, то подобные операнды редко указываются в директиве DW.

Адресное выражение

В качестве операнда директивы DW может быть указано адресное выражение, т.е. выражение, значением которого является адрес. Чаще всего применяется случай адресного выражения, указываемого через имя переменной или метки. Например:

C DB ?

D DW C

В этом случае ассемблер записывает в слово, выделенное под переменную D, адрес переменной C, который становится начальным значением переменной C.

Несколько операндов, конструкция повторения

В правой части директивы DW можно указать любое число операндов, а также конструкцию повторения. Например:

E DW 40000, 30 DUP (?)

Результат будет аналогичен выполнению директивы DB, с то лишь разницей, что будут отводиться слова.

Директива DD (dd)

По директиве DD (define double word, определить двойное слово) описываются переменные, под которые отводятся двойные слова. В общем эта директива похожа на предыдущие.

Операнд ?

Пример:

A DD ?

Под переменную A выделяется двойное слово, в которое ассемблер ничего не записывает, т.е. переменная A не получает начального значения.

Целое число со значением от -2^{31} *до* $2^{32}-1$

Пример:

B DD 123456h

В данном случае переменная B получает начальное значение, причем это значение ассемблер записывает в память в обратном порядке: 56h 34h 12h 00h.

Константное выражение (со значением от -2^{15} *до* $2^{16}-1$*)*

Обратите внимание на диапазон возможных значений выражения — он в два раза меньше диапазона чисел, которые можно записать в двойном слове. Единственное исключение — это явно задать в директиве DD "большое" число. Если же мы укажем хотя бы одну операцию, то ответ тут же будет взят по модулю 2^{16}. Например, по директиве

C DD 8000h+8002h

Начальным значением переменной C станет число 2, а не число 10002h.

Адресное выражение

Такой операнд задает абсолютный адрес. При выполнении данной директивы имя заменяется на адресную пару, состоящее из номера сегмента, где описано это имя, и из смещения имени внутри данного сегмента. Например:

A DD B

Что эквивалентно:

A DD seg B : offset B

Отметим, что операнд можно указать адресную пару следующего вида:

<имя сегмента> : <адресное выражение>

В этом случае ассемблер заменяет имя сегмента на его номер, а адресное выражение (чаще всего имя переменной) заменяет на его смещение, но отсчитанное не от начала того сегмента, где оно описано, а от начала указанного сегмента.

Директивы эквивалентности и присваивания

Рассмотрим как описываются константы. Это делается с помощью директивы эквивалентности — директивы EQU (equal, равно), имеющей следующий синтаксис:

<имя> EQU <операнд>

Здесь обязательно должно быть имя, должен быть и операнд, причем только один. Директивой EQU автор программы заявляет, что указанному операнду он дает указанное имя, и требует, чтобы все вхождения этого имени в текст программы ассемблер заменял на этот операнд. Отметим, что директива носит чисто информативный характер, по ней ассемблер ничего не записывает в машинную программу. Поэтому директиву EQU можно ставить в любом месте программы.

Операнд – имя

Если в правой части директивы указано имя регистра, переменной, константы и т.п., тогда имя слева объявляется синонимом данного имени и все последующие вхождения в текст программы этого имени-синонима ассемблер будет заменяться на имя, указанное справа. Например:

```
A DW ?

B EQU A

C DW B              ; эквивалентно: C DW A
```

Операнд – константное выражение

Примеры:

```
N EQU 100

K EQU N*N-1

STAR EQU '*'
```

Если в правой части директивы EQU стоит константное выражение, тогда указанное слева имя принято называть именем константы. Значением этой константы объявляется значение выражения. Все последующие вхождения в текст программы имени константы ассемблер будет заменять на значение этой константы. Если в константном выражении используются имена других констант, то они должны быть

описаны раньше данной директивы EQU, иначе ассемблер, просматривающий текст программы сверху вниз, не сможет вычислить значение этого выражения.

Операнд – любой другой текст

Примеры:

S EQU 'Вы ошиблись'

LX EQU X+(N-1)

WP EQU WORD PTR

В данном случае считается, что указанное имя обозначает операнд в том виде, как записан (операнд не вычисляется). Именно на этот текст и будет заменяться каждое вхождение данного имени в программу. Например, следующие предложения слева эквивалентны предложениям справа:

ANS DB S, '!'	ANS DB 'Вы ошиблись', '!'
NEG LX	NEG X+(N-1)
INC WP [BX]	INC WORD PTR [BX]

Такой вариант директивы EQU обычно используется для того, чтобы более короткие обозначения для часто встречающихся длинных текстов.

Рассмотрим еще одну директиву языка ассемблера, похожую на директиву EQU, — директиву присваивания:

<имя> = <константное выражение>

Эта директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличии от констант, определенных по директиве EQU, данная константа может менять свое значение, обозначая в разных частях текста программы разные числа. Например:

```
K=10
A DW K     ; эквивалентно: A DW 10
K=K+4
B DB K     ; эквивалентно: B DB 14
```

Если с помощью директивы EQU можно определять имя, обозначающее не только число, но и другие конструкции, то по директиве присваивания можно определить только числовую константу. Кроме того, если имя указано в левой части директивы EQU, то оно не может

появляться в левой части других директив (его нельзя переопределять). Я вот имя, появившееся в левой части директивы присваивания, может снова появиться в начале другой такой директивы (но только такой).

2.3 Си/Си++ (C/C++)

Языке "C" имеется только несколько основных типов данных:

CHAR один байт, в котором может находиться один символ из внутреннего набора символов.

INT Целое, обычно соответствующее естественному размеру целых в используемой машине.

FLOAT С плавающей точкой одинарной точности.

DOUBLE С плавающей точкой двойной точности.

Кроме того имеется ряд квалификаторов, которые можно использовать с типом INT: SHORT (короткое), LONG (длинное) и UNSIGNED (без знака). Квалификаторы SHORT и LONG указывают на различные размеры целых. Числа без знака подчиняются законам арифметики по модулю 2 в степени N, где N — число битов в INT; числа без знаков всегда положительны. Описания с квалификаторами имеют вид:

SHORT INT X;

LONG INT Y;

UNSIGNED INT Z;

Слово INT в таких ситуациях может быть опущено, что обычно и делается. Количество битов, отводимых под эти объекты зависит от имеющейся машины.

Цель состоит в том, чтобы SHORT и LONG давали возможность в зависимости от практических нужд использовать различные длины целых; тип INT отражает наиболее "естественный" размер конкретной машины. Каждый компилятор свободно интерпретирует SHORT и LONG в соответствии со своими аппаратными средствами. Все, на что вы можете твердо полагаться, это то, что SHORT не длиннее, чем LONG.

Константы

Рассмотрим константы типа INT и FLOAT мы. Отметим еще только, что как обычная

123.456e-7,

так и "научная" запись

0.12e3

для FLOAT является законной.

Каждая константа с плавающей точкой считается имеющей тип DOUBLE, так что обозначение "E" служит как для FLOAT, так и для DOUBLE.

Длинные константы записываются в виде 123L. Обычная целая константа, которая слишком длинна для типа INT, рассматривается как LONG.

Существует система обозначений для восьмеричных и шестнадцатеричных констант: лидирующий 0(нуль) в константе типа INT указывает на восьмеричную константу, а стоящие впереди 0X соответствуют шестнадцатеричной константе. Например, десятичное число 31 можно записать как 037 в восьмеричной форме и как 0X1F в шестнадцатеричной. Шестнадцатеричные и восьмеричные константы могут также заканчиваться буквой L, что делает их относящимися к типу LONG.

Символьная константа

Символьная константа — это один символ, заключенный в одинарные кавычки, как, например, 'x'. Значением символьной константы является численное значение этого символа во внутреннем машинном наборе символов. Например, в наборе символов ASCII символьный нуль, или '0', имеет значение 48, а в коде EBCDIC — 240, и оба эти значения совершенно отличны от числа 0. Написание '0' вместо численного значения, такого как 48 или 240, делает программу не зависящей от конкретного численного представления этого символа в данной машине. Символьные константы точно так же участвуют в численных операциях, как и любые другие числа, хотя наиболее часто они используются в сравнении с другими символами. Правила преобразования будут изложены позднее.

Некоторые неграфические символы могут быть представлены как символьные константы с помощью условных последовательностей, как, например, \N (новая строка), \T (табуляция), \0 (нулевой символ), \\ (обратная косая черта), \' (одинарная кавычка) и т.д. Хотя они выглядят как два символа, на самом деле являются одним. Кроме того, можно сгенерировать произвольную последовательность двоичных знаков размером в байт, если написать

'\DDD'

где DDD — от одной до трех восьмеричных цифр, как в

#DEFINE FORMFEED '\014' /* FORM FEED */

Символьная константа '\0', изображающая символ со значением 0, часто записывается вместо целой константы 0, чтобы подчеркнуть символьную природу некоторого выражения.

Константное выражение

Константное выражение это выражение, состоящее из одних констант. Такие выражения обрабатываются во время компиляции, а не при прогоне программы, и соответственно могут быть использованы в любом месте, где можно использовать константу, как, например в

```
#DEFINE MAXLINE 1000
CHAR LINE[MAXLINE+1];
```

или

```
SECONDS = 60 * 60 * HOURS;
```

Строчная константа

Строчная константа — это последовательность, состоящая из нуля или более символов, заключенных в двойные кавычки, как, например,

```
"I AM A STRING"            /* я - строка */
```

или

```
""          /* NULL STRING */          /* нуль-строка */
```

Кавычки не являются частью строки, а служат только для ее ограничения. те же самые условные последовательности, которые использовались в символьных константах, применяются и в строках; символ двойной кавычки изображается как \".

С технической точки зрения строка представляет собой массив, элементами которого являются отдельные символы. Чтобы программам было удобно определять конец строки, компилятор автоматически помещает в конец каждой строки нуль-символ \0. Такое представление означает, что не накладывается конкретного ограничения на то, какую длину может иметь строка, и чтобы определить эту длину, программы должны просматривать строку полностью. При этом для физического хранения строки требуется на одну ячейку памяти больше, чем число заключенных в кавычки символов. Следующая функция STRLEN(S) вычисляет длину символьной строки S не считая конечный символ \0.

```
STRLEN(S)              /* RETURN LENGTH OF S */
CHAR  S[];
 {
   INT I;

   I = 0;
```

```
  WHILE (S[I] != '\0')
        ++I;
  RETURN(I);
}
```

Будьте внимательны и не путайте символьную константу со строкой, содержащей один символ: 'X' — это не то же самое, что "X". Первое — это отдельный символ, использованный с целью получения численного значения, соответствующего букве х в машинном наборе символов. Второе — символьная строка, состоящая из одного символа (буква х) и \0.

Описания

Все переменные должны быть описаны до их использования, хотя некоторые описания делаются неявно, по контексту. Описание состоит из спецификатора типа и следующего за ним списка переменных, имеющих этот тип, как, например,

```
INT LOWER, UPPER, STEP;
CHAR C, LINE[1000];
```

Переменные можно распределять по описаниям любым образом; приведенные выше списки можно с тем же успехом записать в виде

```
INT LOWER;
INT UPPER;
INT STEP;
CHAR C;
CHAR LINE[1000];
```

Такая форма занимает больше места, но она удобна для добавления комментария к каждому описанию и для последующих модификаций.

Переменным могут быть присвоены начальные значения внутри их описания, хотя здесь имеются некоторые ограничения. Если за именем переменной следуют знак равенства и константа, то эта константа служит в качестве инициализатора, как, например, в

```
CHAR BACKSLASH = '\\';

INT I = 0;

FLOAT EPS = 1.0E-5;
```

Если рассматриваемая переменная является внешней или статической, то инициализация проводится только один раз, согласно концепции до начала выполнения программы. Инициализируемым явно автоматическим переменным начальные значения присваиваются при каждом обращении к функции, в которой они описаны. Автоматические переменные, не инициализируемые явно, имеют неопределенные значения, (т.е. мусор). Внешние и статические переменные по умолчанию инициализируются нулем, но, тем не менее, их явная инициализация является признаком хорошего стиля.

Мы продолжим обсуждение вопросов инициализации, когда будем описывать новые типы данных.

Преобразование типов

Если в выражениях встречаются операнды различных типов, то они преобразуются к общему типу в соответствии с небольшим набором правил. В общем, автоматически производятся только преобразования, имеющие смысл, такие как, например, преобразование целого в плавающее в выражениях типа F+I. Выражения же, лишенные смысла, такие как использование переменной типа FLOAT в качестве индекса, запрещены.

Во-первых, типы CHAR и INT могут свободно смешиваться в арифметических выражениях: каждая переменная типа CHAR автоматически преобразуется в INT. Это обеспечивает значительную гибкость при проведении определенных преобразований символов. Примером может служить функция ATOI, которая ставит в соответствие строке цифр ее численный эквивалент.

```
 ATOI(S)                    /* CONVERT S TO INTEGER */
 CHAR S[];
 {
INT I, N;

   N = 0;
   FOR ( I = 0; S[I]>='0' && S[I]<='9'; ++I)
        N = 10 * N + S[I] - '0';
```

```
  RETURN(N);
}
```

Как уже обсуждалось ранее, выражение

```
S[I] - '0'
```

имеет численное значение находящегося в S[I] символа, потому что значение символов '0', '1' и т.д. образуют возрастающую последовательность расположенных подряд целых положительных чисел.

Другой пример преобразования CHAR в INT дает функция LOWER, преобразующая данную прописную букву в строчную. Если выступающий в качестве аргумента символ не является прописной буквой, то LOWER возвращает его неизменным. Приводимая ниже программа справедлива только для набора символов ASCII.

```
LOWER(C) /* CONVERT C TO LOWER CASE; ASCII ONLY */
INT C;
{
  IF ( C >= 'A' && C <= 'Z' )
        RETURN( C + '@' - 'A');
  ELSE   /*@ Записано вместо 'A' строчного*/
        RETURN(C);
}
```

Эта функция правильно работает при коде ASCII, потому что численные значения, соответствующие в этом коде прописным и строчным буквам, отличаются на постоянную величину, а каждый алфавит является сплошным — между а и Z нет ничего, кроме букв. Это последнее замечание для набора символов EBCDIC систем IBM 360/370 оказывается несправедливым, в силу чего эта программа на таких системах работает неправильно — она преобразует не только буквы.

При преобразовании символьных переменных в целые возникает один тонкий момент. Дело в том, что сам язык не указывает, должны ли переменным типа CHAR соответствовать численные значения со знаком или без знака. Может ли при преобразовании CHAR в INT получиться отрицательное целое? К сожалению, ответ на этот вопрос меняется от машины к машине, отражая расхождения в их архитектуре. На некоторых машинах (PDP-11, например) переменная типа CHAR, крайний левый бит которой содержит 1, преобразуется в отрицательное целое ("знаковое расширение"). На других машинах такое преобразование сопровождается

добавлением нулей с левого края, в результате чего всегда получается положительное число.

Определение языка "C" гарантирует, что любой символ из стандартного набора символов машины никогда не даст отрицательного числа, так что эти символы можно свободно использовать в выражениях как положительные величины. Но произвольные комбинации двоичных знаков, хранящиеся как символьные переменные на некоторых машинах, могут дать отрицательные значения, а на других положительные.

Неявные арифметические преобразования работают в основном, как и ожидается. В общих чертах, если операция типа + или *, которая связывает два операнда (бинарная операция), имеет операнды разных типов, то перед выполнением операции "низший" тип преобразуется к "высшему" и получается результат "высшего" типа. Более точно, к каждой арифметической операции применяется следующая последовательность правил преобразования.

- Типы CHAR и SHORT преобразуются в INT, а FLOAT в DOUBLE.
- Затем, если один из операндов имеет тип DOUBLE, то другой преобразуется в DOUBLE, и результат имеет тип DOUBLE.
- В противном случае, если один из операндов имеет тип LONG, то другой преобразуется в LONG, и результат имеет тип LONG.
- В противном случае, если один из операндов имеет тип UNSIGNED, то другой преобразуется в UNSIGNED и результат имеет тип UNSIGNED.
- В противном случае операнды должны быть типа INT, и результат имеет тип INT.

Подчеркнем, что все переменные типа FLOAT в выражениях преобразуются в DOUBLE; в "C" вся плавающая арифметика выполняется с двойной точностью.

Преобразования возникают и при присваиваниях; значение правой части преобразуется к типу левой, который и является типом результата. Символьные переменные преобразуются в целые либо со знаковым расширением, либо без него, как описано выше. Обратное преобразование INT в CHAR ведет себя хорошо — лишние биты высокого порядка просто отбрасываются. Таким образом

```
INT I;
CHAR C;
I = C;
C = I;
```

значение 'c' не изменяется. Это верно независимо от того, вовлекается ли знаковое расширение или нет.

Если x типа FLOAT, а I типа INT, то как

x = I;

так и

I = x;

приводят к преобразованиям; при этом FLOAT преобразуется в INT отбрасыванием дробной части. Тип DOUBLE преобразуется во FLOAT округлением. Длинные целые преобразуются в более короткие целые и в переменные типа CHAR посредством отбрасывания лишних битов высокого порядка.

Так как аргумент функции является выражением, то при передаче функциям аргументов также происходит преобразование типов: в частности, CHAR и SHORT становятся INT, а FLOAT становится DOUBLE. Именно поэтому мы описывали аргументы функций как INT и DOUBLE даже тогда, когда обращались к ним с переменными типа CHAR и FLOAT.

Наконец, в любом выражении может быть осуществлено ("принуждено") явное преобразование типа с помощью конструкции, называемой перевод (CAST). В этой конструкции, имеющей вид

(имя типа) выражение

Выражение преобразуется к указанному типу по правилам преобразования, изложенным выше. Фактически точный смысл операции перевода можно описать следующим образом: выражение как бы присваивается некоторой переменной указанного типа, которая затем используется вместо всей конструкции. Например, библиотечная процедура SQRT ожидает аргумента типа DOUBLE и выдаст бессмысленный ответ, если к ней по небрежности обратятся с чем-нибудь иным. таким образом, если N — целое, то выражение

SQRT((DOUBLE) N)

до передачи аргумента функции SQRT преобразует N к типу DOUBLE. (Отметим, что операция перевод преобразует значение N в надлежащий тип; фактическое содержание переменной N при этом не изменяется). Операция перевода имеет тот же уровень старшинства, что и другие унарные операции.

Инициализация

Если явная инициализация отсутствует, то внешним и статическим переменным присваивается значение нуль; автоматические и регистровые переменные имеют в этом случае неопределенные значения (мусор).

Простые переменные (не массивы или структуры) можно инициализировать при их описании, добавляя вслед за именем знак равенства и константное выражение:

```
INT X = 1;
CHAR SQUOTE = '\'';
LONG DAY = 60 * 24;    /* MINUTES IN A DAY */
```

Для внешних и статических переменных инициализация выполняется только один раз, на этапе компиляции. Автоматические и регистровые переменные инициализируются каждый раз при входе в функцию или блок.

В случае автоматических и регистровых переменных инициализатор не обязан быть константой: на самом деле он может быть любым значимым выражением, которое может включать определенные ранее величины и даже обращения к функциям.

Автоматические массивы не могут быть инициализированы. Внешние и статические массивы можно инициализировать, помещая вслед за описанием заключенный в фигурные скобки список начальных значений, разделенных запятыми. Если количество начальных значений меньше, чем указанный размер массива, то остальные элементы заполняются нулями. Перечисление слишком большого числа начальных значений является ошибкой. К сожалению, не предусмотрена возможность указания, что некоторое начальное значение повторяется, и нельзя инициализировать элемент в середине массива без перечисления всех предыдущих.

Для символьных массивов существует специальный способ инициализации; вместо фигурных скобок и запятых можно использовать строку:

```
CHAR PATTERN[] = "THE";
```

Это сокращение более длинной, но эквивалентной записи:

```
CHAR PATTERN[] = { 'T', 'H', 'E', '\0' };
```

Если размер массива любого типа опущен, то компилятор определяет его длину, подсчитывая число начальных значений. В этом конкретном случае размер равен четырем (три символа плюс конечное \0).

Макроподстановка

Определение вида

```
#DEFINE TES     1
```

приводит к макроподстановке самого простого вида — замене имени на строку символов. Имена в #DEFINE имеют ту же самую форму, что и

идентификаторы в "c"; заменяющий текст совершенно произволен. Нормально заменяющим текстом является остальная часть строки; длинное определение можно продолжить, поместив \ в конец продолжаемой строки. "Область действия" имени, определенного в #DEFINE, простирается от точки определения до конца исходного файла. Имена могут быть переопределены, и определения могут использовать определения, сделанные ранее. Внутри заключенных в кавычки строк подстановки не производятся, так что если, например, YES — определенное имя, то в PRINTF("YES") не будет сделано никакой подстановки.

Так как реализация #DEFINE является частью работы макропредпроцессора, а не собственно компилятора, имеется очень мало грамматических ограничений на то, что может быть определено. Так, например, любители Алгола могут объявить

```
#DEFINE THEN
#DEFINE BEGIN {
#DEFINE END   ;}
```

и затем написать

```
IF (I > 0) THEN
  BEGIN
        A = 1;
        B = 2
  END
```

Имеется также возможность определения макроса с аргументами, так что заменяющий текст будет зависеть от вида обращения к макросу. Определим, например, макрос с именем MAX следующим образом:

```
#DEFINE MAX(A, B)  ((A) > (B) ? (A) : (B))
```

когда строка

```
X = MAX(P+Q, R+S);
```

будет заменена строкой

```
X = ((P+Q) > (R+S) ? (P+Q) : (R+S));
```

Такая возможность обеспечивает "функцию максимума", которая расширяется в последовательный код, а не в обращение к функции. При правильном обращении с аргументами такой макрос будет работать с любыми типами данных; здесь нет необходимости в различных видах MAX для данных разных типов, как это было бы с функциями.

Указатели и массивы

Указатель — это переменная, содержащая адрес другой переменной. указатели очень широко используются в языке "C". Это происходит отчасти потому, что иногда они дают единственную возможность выразить нужное действие, а отчасти потому, что они обычно ведут к более компактным и эффективным программам, чем те, которые могут быть получены другими способами.

Указатели обычно смешивают в одну кучу с операторами GOTO, характеризуя их как чудесный способ написания программ, которые невозможно понять. Это безусловно справедливо, если указатели используются беззаботно; очень просто ввести указатели, которые указывают на что-то совершенно неожиданное. Однако, при определенной дисциплине, использование указателей помогает достичь ясности и простоты. Именно этот аспект мы попытаемся здесь проиллюстрировать.

Указатели и адреса

Так как указатель содержит адрес объекта, это дает возможность "косвенного" доступа к этому объекту через указатель. Предположим, что x — переменная, например, типа INT, а px — указатель, созданный неким еще не указанным способом. Унарная операция & выдает адрес объекта, так что оператор

```
px = &x;
```

присваивает адрес x переменной px; говорят, что px "указывает" на x. Операция & применима только к переменным и элементам массива, конструкции вида &(x-1) и &3 являются незаконными. Нельзя также получить адрес регистровой переменной.

Унарная операция * рассматривает свой операнд как адрес конечной цели и обращается по этому адресу, чтобы извлечь содержимое. Следовательно, если Y тоже имеет тип INT, то

```
Y = *px;
```

присваивает Y содержимое того, на что указывает px. Так последовательность

```
px = &x;
Y = *px;
```

присваивает Y то же самое значение, что и оператор

```
Y = X;
```

Переменные, участвующие во всем этом необходимо описать:

INT X, Y;

INT *PX;

с описанием для X и Y мы уже неодонократно встречались. Описание указателя

INT *PX;

является новым и должно рассматриваться как мнемоническое; оно говорит, что комбинация *PX имеет тип INT. Это означает, что если PX появляется в контексте *PX, то это эквивалентно переменной типа INT. Фактически синтаксис описания переменной имитирует синтаксис выражений, в которых эта переменная может появляться. Это замечание полезно во всех случаях, связанных со сложными описаниями. Например,

DOUBLE ATOF(), *DP;

говорит, что ATOF() и *DP имеют в выражениях значения типа DOUBLE.

Вы должны также заметить, что из этого описания следует, что указатель может указывать только на определенный вид объектов.

Указатели могут входить в выражения. Например, если PX указывает на целое X, то *PX может появляться в любом контексте, где может встретиться X. Так оператор

Y = *PX + 1

присваивает Y значение, на 1 большее значения X;

PRINTF("%D\N", *PX)

печатает текущее значение X;

D = SQRT((DOUBLE) *PX)

получает в D квадратный корень из X, причем до передачи функции SQRT значение X преобразуется к типу DOUBLE.

В выражениях вида

Y = *PX + 1

унарные операции * и & связаны со своим операндом более крепко, чем арифметические операции, так что такое выражение берет то значение, на которое указывает PX, прибавляет 1 и присваивает результат переменной Y. Мы вскоре вернемся к тому, что может означать выражение

Y = *(PX + 1)

Ссылки на указатели могут появляться и в левой части присваиваний. Если PX указывает на X, то

*PX = 0

полагает X равным нулю, а

*PX += 1

увеличивает его на единицу, как и выражение

(*PX)++

Круглые скобки в последнем примере необходимы; если их опустить, то поскольку унарные операции, подобные * и ++, выполняются справа налево, это выражение увеличит PX, а не ту переменную, на которую он указывает.

И, наконец, так как указатели являются переменными, то с ними можно обращаться, как и с остальными переменными. Если PY - другой указатель на переменную типа INT, то

PY = PX

копирует содержимое PX в PY, в результате чего PY указывает на то же, что и PX.

Указатели и массивы

В языке "C" существует сильная взаимосвязь между указателями и массивами, настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. Вариант с указателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере для начинающего. Описание

INT A[10]

определяет массив размера 10, т.е. набор из 10 последовательных объектов, называемых A[0], A[1], ..., A[9]. Запись A[I] соответствует элементу массива через I позиций от начала. Если PA — указатель целого, описанный как

INT *PA

то присваивание

PA = &A[0]

приводит к тому, что PA указывает на нулевой элемент массива A; это означает, что PA содержит адрес элемента A[0]. Теперь присваивание

X = *PA

будет копировать содержимое A[0] в X.

Если PA указывает на некоторый определенный элемент массива A, то по определению PA+1 указывает на следующий элемент, и вообще PA-I указывает на элемент, стоящий на I позиций до элемента, указываемого PA, а PA+I на элемент, стоящий на I позиций после. Таким образом, если PA указывает на A[0], то

*(PA+1)

ссылается на содержимое A[1], PA+I - адрес A[I], а *(PA+I) — содержимое A[I].

Эти замечания справедливы независимо от типа переменных в массиве A. Суть определения "добавления 1 к указателю", а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, I в PA+I перед прибавлением умножается на размер объектов, на которые указывает PA.

Очевидно существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий. Так как имя массива является синонимом местоположения его нулевого элемента, то присваивание PA=&A[0] можно записать как

PA = A

Еще более удивительным, по крайней мере, на первый взгляд, кажется тот факт, что ссылку на A[I] можно записать в виде *(A+I). При анализировании выражения A[I] в языке "C" оно немедленно преобразуется к виду *(A+I); эти две формы совершенно эквивалентны. Если применить операцию & к обеим частям такого соотношения эквивалентности, то мы получим, что &A[I] и A+I тоже идентичны: A+I — адрес I-го элемента от начала A. С другой стороны, если PA является указателем, то в выражениях его можно использовать с индексом: PA[I] идентично *(PA+I). Короче, любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении.

Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду. Указатель является переменной, так что операции PA=A и PA++ имеют смысл. Но имя массива является константой, а не переменной: конструкции типа A=PA или A++, или P=&A будут незаконными.

Когда имя массива передается функции, то на самом деле ей передается местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем.

Многомерные массивы

В языке "C" предусмотрены прямоугольные многомерные массивы, хотя на практике существует тенденция к их значительно более редкому использованию по сравнению с массивами указателей. В этом разделе мы рассмотрим некоторые их свойства.

Рассмотрим задачу преобразования дня месяца в день года и наоборот. Например, 1-ое марта является 60-м днем невисокосного года и 61-м днем високосного года. Давайте введем две функции для выполнения этих преобразований: DAY_OF_YEAR преобразует месяц и день в день года, а MONTH_DAY преобразует день года в месяц и день. Так как эта последняя функция возвращает два значения, то аргументы месяца и дня должны быть указателями:

```
 MONTH_DAY(1977, 60, &M, &D)
```

Полагает M равным 3 и D равным 1 (1-ое марта).

Обе эти функции нуждаются в одной и той же информационной таблице, указывающей число дней в каждом месяце. Так как число дней в месяце в високосном и в невисокосном году отличается, то проще представить их в виде двух строк двумерного массива, чем пытаться прослеживать во время вычислений, что именно происходит в феврале. Вот этот массив и выполняющие эти преобразования функции:

```
STATIC INT DAY_TAB[2][13] = {
    (0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),
    (0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
};


DAY_OF_YEAR(YEAR, MONTH, DAY)      /* SET DAY OF YEAR */
INT YEAR, MONTH, DAY;        /* FROM MONTH & DAY */
{
   INT I, LEAP;
  LEAP = YEAR%4 == 0 && YEAR%100 != 0 \!\! YEAR%400 == 0;
   FOR (I = 1; I < MONTH; I++)
 DAY += DAY_TAB[LEAP][I];
```

```
   RETURN(DAY);
{

MONTH_DAY(YEAR, YEARDAY, PMONTH, PDAY) /*SET MONTH,DAY
*/
INT YEAR, YEARDAY, *PMONTH, *PDAY; /* FROM DAY OF YEAR */
{
 LEAP = YEAR%4 == 0 && YEAR%100 != 0 \!\! YEAR%400 == 0;
  FOR (I = 1; YEARDAY > DAY_TAB[LEAP][I]; I++)
YEARDAY -= DAY_TAB[LEAP][I];
  *PMONTH = I;
  *PDAY = YEARDAY;
}
```

Массив DAY_TAB должен быть внешним как для DAY_OF_YEAR, так и для MONTH_DAY, поскольку он используется обеими этими функциями.

Массив DAY_TAB является первым двумерным массивом, с которым мы имеем дело. По определению в "C" двумерный массив по существу является одномерным массивом, каждый элемент которого является массивом. Поэтому индексы записываются как

DAY_TAB[I][J]

а не

DAY_TAB [I, J]

как в большинстве языков. В остальном с двумерными массивами можно в основном обращаться таким же образом, как в других языках. Элементы хранятся по строкам, т.е. при обращении к элементам в порядке их размещения в памяти быстрее всего изменяется самый правый индекс.

Массив инициализируется с помощью списка начальных значений, заключенных в фигурные скобки; каждая строка двумерного массива инициализируется соответствующим подсписком. Мы поместили в начало массива DAY_TAB столбец из нулей для того, чтобы номера месяцев изменялись естественным образом от 1 до 12, а не от 0 до 11. Так как за экономию памяти у нас пока не награждают, такой способ проще, чем подгонка индексов.

Если двумерный массив передается функции, то описание соответствующего аргумента функции должно содержать количество

столбцов; количество строк несущественно, поскольку, как и прежде, фактически передается указатель.

Массивы указателей; указатели указателей

Так как указатели сами являются переменными, то вы вполне могли бы ожидать использования массива указателей.

Указатели и многомерные массивы

Начинающие изучать язык "C" иногда становятся в тупик перед вопросом о различии между двумерным массивом и массивом указателей.

Если имеются описания

```
INT A[10][10];
INT *B[10];
```

то A и B можно использовать сходным образом в том смысле, что как A[5][5], так и B[5][5] являются законными ссылками на отдельное число типа INT. Но A — настоящий массив: под него отводится 100 ячеек памяти и для нахождения любого указанного элемента проводятся обычные вычисления с прямоугольными индексами. Для B, однако, описание выделяет только 10 указателей; каждый указатель должен быть установлен так, чтобы он указывал на массив целых. Если предположить, что каждый из них указывает на массив из 10 элементов, то тогда где-то будет отведено 100 ячеек памяти плюс еще десять ячеек для указателей. Таким образом, массив указателей использует несколько больший объем памяти и может требовать наличие явного шага инициализации. Но при этом возникают два преимущества: доступ к элементу осуществляется косвенно через указатель, а не посредством умножения и сложения, и строки массива могут иметь различные длины. Это означает, что каждый элемент B не должен обязательно указывать на вектор из 10 элементов; некоторые могут указывать на вектор из двух элементов, другие — из двадцати, а третьи могут вообще ни на что не указывать.

Хотя мы вели это обсуждение в терминах целых, несомненно, чаще всего массивы указателей используются для хранения символьных строк различной длины.

Структуры

Структура — это набор из одной или более переменных, возможно различных типов, сгруппированных под одним именем для удобства обработки. (В некоторых языках структуры называются "записями").

Традиционным примером структуры является учетная карточка работающего: "служащий" описывается набором атрибутов таких, как

фамилия, имя, отчество (Ф.И.О.), адрес, код социального обеспечения, зарплата и т.д. Некоторые из этих атрибутов сами могут оказаться структурами: Ф.И.О. Имеет несколько компонент, как и адрес, и даже зарплата.

Структуры оказываются полезными при организации сложных данных особенно в больших программах, поскольку во многих ситуациях они позволяют сгруппировать связанные данные таким образом, что с ними можно обращаться, как с одним целым, а не как с отдельными объектами.

Давайте снова обратимся к процедурам преобразования даты. Дата состоит из нескольких частей таких, как день, месяц, и год, и, возможно, день года и имя месяца. Эти пять переменных можно объединить в одну структуру вида:

```
STRUCT DATE \(
INT  DAY;
INT  MONTH;
INT  YEAR;
INT  YEARDAY;
CHAR MON_NAME[4];
\);
```

Описание структуры, состоящее из заключенного в фигурные скобки списка описаний, начинается с ключевого слова STRUCT. За словом STRUCT может следовать необязательное имя, называемое ярлыком структуры (здесь это DATE). Такой ярлык именует структуры этого вида и может использоваться в дальнейшем как сокращенная запись подробного описания. Элементы или переменные, упомянутые в структуре, называются членами. Ярлыки и члены структур могут иметь такие же имена, что и обычные переменные (т.е. Не являющиеся членами структур), поскольку их имена всегда можно различить по контексту. Конечно, обычно одинаковые имена присваивают только тесно связанным объектам.

Точно так же, как в случае любого другого базисного типа, за правой фигурной скобкой, закрывающей список членов, может следовать список переменных.

Оператор

```
STRUCT   \( ...\) X,Y,Z;
```

синтаксически аналогичен

```
INT X,Y,Z;
```

в том смысле, что каждый из операторов описывает X , Y и Z в качестве переменных соответствующих типов и приводит к выделению для них памяти.

Описание структуры, за которым не следует списка переменных, не приводит к выделению какой-либо памяти; оно только определяет шаблон или форму структуры. Однако, если такое описание снабжено ярлыком, то этот ярлык может быть использован позднее при определении фактических экземпляров структур. Например, если дано приведенное выше описание DATE, то

```
STRUCT  DATE D;
```

определяет переменную D в качестве структуры типа DATE. Внешнюю или статическую структуру можно инициализировать, поместив вслед за ее определением список инициализаторов для ее компонент:

```
STRUCT DATE D=\( 4, 7, 1776, 186, "JUL"\);
```

Член определенной структуры может быть указан в выражении с помощью конструкции вида

имя структуры . Член

Операция указания члена структуры "." связывает имя структуры и имя члена.

Структуры могут быть вложенными; учетная карточка служащего может фактически выглядеть так:

```
 STRUCT  PERSON  \(
   CHAR NAME[NAMESIZE];
   CHAR ADDRESS[ADRSIZE];
   LONG ZIPCODE;   /* почтовый индекс */
   LONG SS_NUMBER; /* код соц. Обеспечения */
   DOUBLE SALARY;  /* зарплата */
   STRUCT DATE BIRTHDATE; /* дата рождения */
   STRUCT DATE HIREDATE; /* дата поступления на работу */
 \);
```

Структура PERSON содержит две структуры типа DATE.

В языке "C" существует ряд ограничений на использование структур. Обязательные правила заключаются в том, что единственные операции, которые вы можете проводить со структурами, состоят в определении ее адреса с помощью операции & и доступе к одному из ее членов. Это влечет за собой то, что структуры нельзя присваивать или копировать как целое, и что они не могут быть переданы функциям или возвращены ими.

На указатели структур эти ограничения, однако, не накладываются, так что структуры и функции все же могут с удобством работать совместно. И, наконец, автоматические структуры, как и автоматические массивы, не могут быть инициализированы; инициализация возможна только в случае внешних или статических структур.

Описание

```
STRUCT DATE *PD;
```

говорит, что PD является указателем структуры типа DATE. Запись, показанная на примере

```
PD->YEAR
```

является новой. Если P - указатель на структуру, то

```
P-> член структуры
```

обращается к конкретному члену. (Операция -> - это знак минус, за которым следует знак ">".)

Так как PD указывает на структуру, то к члену YEAR можно обратиться и следующим образом

```
(*PD).YEAR
```

но указатели структур используются настолько часто, что запись -> оказывается удобным сокращением. Круглые скобки в (*PD).YEAR необходимы, потому что операция указания члена структуры старше, чем * . Обе операции, "->" и ".", ассоциируются слева направо, так что конструкции слева и справа эквивалентны

```
P->Q->MEMB                   (P->Q)->MEMB

EMP.BIRTHDATE.MONTH          (EMP.BIRTHDATE).MONTH
```

Операции работы со структурами "->" и "." наряду со () для списка аргументов и [] для индексов находятся на самом верху иерархии старшинства операций и, следовательно, связываются очень крепко.

Поля

Когда вопрос экономии памяти становится очень существенным, то может оказаться необходимым помещать в одно машинное слово несколько различных объектов; одно из особенно распространенных употреблений — набор однобитовых признаков в применениях, подобных символьным таблицам компилятора внешне обусловленные форматы данных, такие как интерфейсы аппаратных средств также зачастую предполагают возможность получения слова по частям.

Представьте себе фрагмент компилятора, который работает с символьной таблицей. С каждым идентификатором программы связана

определенная информация, например, является он или нет ключевым словом, является ли он или нет внешним и/или статическим и т.д. Самый компактный способ закодировать такую информацию — поместить набор однобитовых признаков в отдельную переменную типа CHAR или INT.

Обычный способ, которым это делается, состоит в определении набора "масок", отвечающих соответствующим битовым позициям, как в

```
#DEFINE  KEYWORD    01
#DEFINE  EXTERNAL   02
#DEFINE  STATIC     04
```

(числа должны быть степенями двойки). Тогда обработка битов сведется к "жонглированию битами" с помощью операций сдвига, маскирования и дополнения.

Некоторые часто встречающиеся идиомы:

```
FLAGS \!= EXTERNAL  \! STATIC;
```

включает биты EXTERNAL и STATIC в FLAGS, в то время как

```
FLAGS &= \^(eXTERNAL \! STATIC);
```

их выключает, а

```
IF ((FLAGS & (EXTERNAL \! STATIC)) == 0) ...
```

истинно, если оба бита выключены.

Хотя этими идиомами легко овладеть, язык "C" в качестве альтернативы предлагает возможность определения и обработки полей внутри слова непосредственно, а не посредством побитовых логических операций. Поле — это набор смежных битов внутри одной переменной типа INT. Синтаксис определения и обработки полей основывается на структурах. Например, символьную таблицу конструкций #DEFINE, приведенную выше, можно бы было заменить определением трех полей:

```
 STRUCT  \(
UNSIGNED IS_KEYWORD : 1;
UNSIGNED IS_EXTERN  : 1;
UNSIGNED IS_STATIC  : 1;
 \)   FLAGS;
```

Здесь определяется переменная с именем FLAGS, которая содержит три 1-битовых поля. Следующее за двоеточием число задает ширину поля в битах. Поля описаны как UNSIGNED, чтобы подчеркнуть, что они действительно будут величинами без знака.

На отдельные поля можно ссылаться, как FLAGS.IS_STATIE, FLAGS. IS_EXTERN, FLAGS.IS_KEYWORD и т.д., то есть точно так же,

как на другие члены структуры. Поля ведут себя подобно небольшим целым без знака и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры более естественно переписать так:

FLAGS.IS_EXTERN = FLAGS.IS_STATIC = 1;

для включения битов;

FLAGS.IS_EXTERN = FLAGS.IS_STATIC = 0;

для выключения битов;

IF (FLAGS.IS_EXTERN == 0 &&FLAGS.IS_STATIC == 0)...

для их проверки.

Поле не может перекрывать границу INT; если указанная ширина такова, что это должно случиться, то поле выравнивается по границе следующего INT. Полям можно не присваивать имена; неименованные поля (только двоеточие и ширина) используются для заполнения свободного места. Чтобы вынудить выравнивание на границу следующего INT, можно использовать специальную ширину 0.

При работе с полями имеется ряд моментов, на которые следует обратить внимание. По-видимому наиболее существенным является то, что отражая природу различных аппаратных средств, распределение полей на некоторых машинах осуществляется слева направо, а на некоторых справа налево. Это означает, что хотя поля очень полезны для работы с внутренне определенными структурами данных, при разделении внешне определяемых данных следует тщательно рассматривать вопрос о том, какой конец поступает первым.

Другие ограничения, которые следует иметь в виду: поля не имеют знака; они могут храниться только в переменных типа INT (или, что эквивалентно, типа UNSIGNED); они не являются массивами; они не имеют адресов, так что к ним не применима операция &.

Объединения

Объединения — это переменная, которая в различные моменты времени может содержать объекты разных типов и размеров, причем компилятор берет на себя отслеживание размера и требований выравнивания. Объединения представляют возможность работать с различными видами данных в одной области памяти, не вводя в программу никакой машинно-зависимой информации.

В качестве примера, снова из символьной таблицы компилятора, предположим, что константы могут быть типа INT , FLOAT или быть указателями на символы. Значение каждой конкретной константы должно храниться в переменной соответствующего типа, но все же для управления

таблицей самым удобным было бы, если это значение занимало бы один и тот же объем памяти и хранилось в том же самом месте независимо от его типа. это и является назначением объединения — выделить отдельную переменную, в которой можно законно хранить любую одну из переменных нескольких типов. Как и в случае полей, синтаксис основывается на структурах.

```
UNION U_TAG \(
INT IVAL;
FLOAT FVAL;
CHAR *PVAL;
\) UVAL;
```

Переменная UVAL будет иметь достаточно большой размер, чтобы хранить наибольший из трех типов, независимо от машины, на которой осуществляется компиляция, — программа не будет зависеть от характеристик аппаратных средств. Любой из этих трех типов может быть присвоен UVAR и затем использован в выражениях, пока такое использование совместимо: извлекаемый тип должен совпадать с последним помещенным типом. Дело программиста — следить за тем, какой тип хранится в объединении в данный момент; если что-либо хранится как один тип, а извлекается как другой, то результаты будут зависеть от используемой машины.

Синтаксически доступ к членам объединения осуществляется следующим образом:

имя объединения.член

или

указатель объединения ->член

то есть точно так же, как и в случае структур. если для отслеживания типа, хранимого в данный момент в UVAL, используется переменная UTYPE, то можно встретить такой участок программы:

```
IF (UTYPE == INT)
PRINTF("%D\N", UVAL.IVAL);
ELSE IF (UTYPE == FLOAT)
PRINTF("%F\N", UVAL.FVAL);
ELSE IF (UTYPE == STRING)
PRINTF("%S\N", UVAL.PVAL);
ELSE
```

```
PRINTF("BAD TYPE %D IN UTYPE\N", UTYPE);
```

Объединения могут появляться внутри структур и массивов и наоборот. Запись для обращения к члену объединения в структуре (или наоборот) совершенно идентична той, которая используется во вложенных структурах.

В сущности, объединение является структурой, в которой все члены имеют нулевое смещение. Сама структура достаточно велика, чтобы хранить "самый широкий" член, и выравнивание пригодно для всех типов, входящих в объединение. Как и в случае структур, единственными операциями, которые в настоящее время можно проводить с объединениями, являются доступ к члену и извлечение адреса; объединения не могут быть присвоены, переданы функциям или возвращены ими. Указатели объединений можно использовать в точно такой же манере, как и указатели структур.

Тип void задает пустое множество значений. Он используется для обозначения типа функций, которые не возвращают результат. Нельзя описывать объекты с типом void. Любое выражение можно явно преобразовать к типу void, получившееся выражение можно использовать только как выражение-оператор, как левый операнд операции запятая или в качестве второго или третьего операнда в операции ?:

2.4 Паскаль (Pascal)

В качестве примера будем использовать версию Паскаля: Borland Turbo Pascal 7.0.

Идентификаторы

Идентификаторы обозначают следующее:

- Константы
- Поля записей
- Функции
- Метки
- Процедуры
- Программы
- Типы
- Модули
- Переменные

Идентификаторы могут иметь любую длину, но только первые 63 символа являются значимыми для компилятора.

- Первый символ идентификатора должен быть буквой.
- Последующие символы, должны быть буквами, цифрами или знаком подчеркивания (не пробелом!).

Подобно зарезервированным словам, идентификаторы можно записывать в любом регистре, компилятор не чувствителен к регистру.

Квалифицированные идентификаторы:

Если существуют несколько образцов одного и того же идентификатора, то вы можете квалифицировать идентификатор с помощью идентификатора модуля, для того, чтобы выбрать нужный образец идентификатора. Объединенный идентификатор называется квалифицированным идентификатором.

Примеры:

(* Идентификаторы *)

```
WriteLn

Exit

Real2String
```

(* Квалифицированные идентификаторы *)

```
System.MemAvail (* модуль = System, идентификатор = MemAvail *)

Dos.Exec (* модуль = Dos, идентификатор = Exec *)

Crt.Window (* модуль = Crt, идентификатор = Window *)
```

Типы

Type (зарезервированное слово)

Объявление типа определяет идентификатор, который обозначает тип.

Синтаксис:

Type идентификатор = ЛюбойТип;
идентификатор = ЛюбойТип;

ЛюбойТип — идентификатор любого из следующих типов:

- Массив
- Файл
- Объект
- Перечислимый
- Указатель
- Вещественный
- Запись
- Множество
- Строка

Записи

Record (зарезервированное слово)

Запись содержит несколько компонентов, или полей, которые могут иметь различные типы.

Синтаксис:

```
Record
Поля;
Поля;
...
Поля
End;
```

или

```
Record
Поля;
...
Case переключатель : тип Of
Вариант : (поля);
...
...
Вариант : (поля)
End;
```

Замечания:

Каждый список полей - список идентификаторов, разделенных запятыми сопровождаемых двоеточием и указанием типа.

Пример:

```
{ Определения записей }
Type Class = (Num, Dat, Str);
Date = Record
D, M, Y : Integer;
End;
Facts = Record
Name : String[10];
Case Kind : Class Of
Num : (N : Real);
Dat : (D : Date);
Str : (S : String);
End;
```

Порядковые типы

Turbo Pascal имеет десять предопределенных порядковых типов:

Пять из них (целочисленные типы) обозначают подмножество целых чисел. Другие пять предопределенных порядковых типов — булевы (Boolean, WordBool, LongBool, ByteBool) и символьный тип Char.

Два других класса определяемых пользователем порядковых типов — перечислимые типы и типы поддиапазонов. Со всеми перечислимыми типами могут использоваться следующие стандартные функции.

Тип поддиапазона - это диапазон значений порядкового типа называемый главным типом.

Синтаксис:

константа1 .. константа2

Замечания:

При определении типа поддиапазона задается наименьшее и наибольшее значение в поддиапазоне. Обе константы должны иметь один и тот же перечислимый тип, и значение первой константы должно быть меньше или равно значению второй.

Директива компилятора $R управляет проверкой диапазона типов поддиапазона.

Примеры:

{ Поддиапазоны }

0..99

-128..127

Вещественные типы

Вещественный тип имеет набор значений, который является подмножеством вещественных чисел, которые могут быть представлены в виде числа с плавающей точкой и с фиксированным количеством разрядов.

Число с плавающей точкой обычно состоит из трех значений - M, B и E – так что M x B E = N, где B - всегда 2, а M и E - интегральные значения, находящиеся внутри диапазона вещественных чисел.

Turbo Pascal обеспечивает пять предопределенным вещественных типов. Каждый тип имеет свой диапазон и точность:

Тип	Диапазон	Точность	Байт
Real	2.9e-39..1.7e38	11-12	6
Single	1.5e-45..3.4e38	7-8	4
Double	5.0e-324..1.7e308	15-16	8
Extended	3.4e-4932..1.1e493	2 19-20	10
Comp	-9.2e18..9.2e18	19-20	8

Обратите внимание: тип Comp — 64-разрядное целое число. В нем можно хранить только интегральные значения в диапазоне (-2 63 + 1)..(2 63 - 1).

Turbo Pascal поддерживает две модели генерации объектного кода с числами с плавающей запятой:

- Программная поддержка чисел с плавающей запятой {$N-}
- Аппаратная (80x87) поддержка чисел с плавающей запятой {$N+}

Вы можете использовать директиву компилятора $N для переключения между двумя моделями. В режиме $N+, директива компилятора $E управляет, включать ли библиотеки эмуляций 80x87 в готовую программу.

Множества

Set (зарезервированное слово)

Объявление множества (набора).

Синтаксис:

Set Of тип

Замечания:

Исходный тип набора должен быть порядковым с не более, чем 256 возможными значениями. Порядковые значения верхнего и нижнего пределов исходного типа должны быть между в диапазоне от 0 до 255. Значение множества можно задать с помощью конструктора множества,

записав выражения в скобках. Каждое выражение обозначает значение множества.

Запись [] обозначает пустое множество, которое является совместимым со всеми типами множеств.

Пример:

```
{ Типы наборов }
Type  Day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
CharSet = Set Of Char;
Digits = Set Of 0..9;
Days = Set Of Day;

        { Конструкторы множеств }
['0'..'9', 'A'..'Z', 'a'..'z', '_']
[1, 5, I + 1 .. J - 1]
[Mon..Fri]
```

Строковые типы

(String: зарезервированное слово)

Строковая переменная — это последовательность символов с динамической длиной, и постоянным максимальным размером в диапазоне от 1 до 255.

Синтаксис:

String [константа]

или

String

Замечания:

Строковый тип, объявленный без максимального размера имеет размер 255. Строковые константы записываются в одинарных кавычках, например:

- 'Turbo Pascal'
- 'That''s all'

Две последовательных одиночных кавычки используются для задания в строке одиночной кавычки.

Со строковыми типами могут использоваться следующие операторы:

```
+   =   <>   <   >   <=   >=
```

Стандартная функция Length возвращает динамическую длину строки.

Пример:

```
{ Определения строковых типов }
```

Const LineLen = 79;

Type Name = String[25];

Line = String[LineLen];

Символьный тип

Переменные порядкового типа Char используются для хранения символов ASCII. Символьные константы записываются в апострофах, например:

'A', '3' или '*'

Один символ апострофа записывается как два апострофа в апострофах, например:

''''

Булевы типы

Имеются четыре предопределенных булевых типа: Boolean, WordBool, LongBool и ByteBool.

Синтаксис:

Type Boolean = (False, True);

WordBool = (False, True);

LongBool = (False, True);

ByteBool = (False, True);

Замечания:

Эти типы имеют следующие размеры:

Boolean	Byte	8 бит
WordBool	Word	16 бит
LongBool	LongInt	32 бита
ByteBool	Byte	8 бит

Поскольку данные типы являются перечислимыми порядковыми типами, то существуют следующие связи:

False < True

Ord(False) = 0

Ord(True) = 1

Succ(False) = True

Pred(True) = False

Тип Boolean является предпочтительным, так как он использует наименьшее количество памяти. ByteBool, WordBool и LongBool существуют только для совместимости с Windows.

В выражениях, следующие операторы сравнения производят результат типа Boolean:

= <> > < >= <= IN

Для совместимости с Windows, булевы типы могут принимать порядковые значения не только 0 и 1.

Булево выражение является ложью (False), если его порядковое значение равно нулю и истиной (True), если его порядковое значение не равно нулю. Булевы операторы Not, And, Or и Xor работают проверяя значения на 0 (False) или не 0 (True), но всегда возвращают результат с порядковым значением 0 или 1.

Массив

Array (зарезервированное слово). Определяет массив.

Синтаксис:

Array [тип_индексов] Of тип_элементов

Замечания:

Позволяются несколько типов индексов, если они разделены запятыми.

Элементы массива могут иметь любой тип, а тип индексов должен быть порядковым.

Пример:

```
Type IntList  = Array[1..100] Of Integer;
CharData = Array['A'..'Z'] Of Byte;
Matrix = Array[0..9, 0..9] Of Real;
```

Указательные типы

Переменная указательного типа содержит адрес в памяти динамической переменной определенного типа.

Вы можете присвоить значение указательной переменной с помощью:

- процедур New или GetMem
- оператора @ или функции Addr
- функции Ptr

Зарезервированное слово NIL обозначает указатель, который не указывает никуда.

Указатель:

Предопределенный тип Pointer обозначает нетипизированный указатель (указатель который не указывает на какой-либо определенный тип).

PChar:

Предопределенный тип PChar обозначает указатель на строку с завершающим нулем.

```
Type PChar = ^Char;
```

Borland Pascal для Windows поддерживает набор расширенных правил синтаксиса (управляемых с помощью директивы компилятора $X) для облегчения обработки строк типа PChar.

Пример:

```
{ Объявление указательных типов }
Type BytePtr = ^Byte;
WordPtr = ^Word;
IdentPtr = ^IdentRec;
IdentRec = Record
Ident : String[15];
RefCount : Word;
Next : IdentPtr;
End;
```

Объектные типы (Object: зарезервированное слово)

Объект — структура данных, которая содержит фиксированное число компонентов.

Синтаксис:

```
Object
Поле;
Поле;
...
Метод;
Метод;
End;
```

Замечания:

Описание поля объекта состоит из идентификатора, двоеточия и типа данных. Кроме того, объект содержит заголовки методов. Каждый

компонент является или полем (которое содержит данные указанного типа) или методом, который выполняет операцию с полями объекта.

Объявление поля содержит идентификатор, который обозначает поле и его тип данных. Объявление метода содержит заголовки процедур, функций, конструктора или деструктора.

Поле = ИмяПоля(ей) : тип;

Метод = Procedure ИмяМетода(<параметр(ы)> : тип);

или = Function ИмяМетода(<параметр(ы)> : тип) : тип;

или = Constructor ИмяМетода(<параметр(ы)> : тип

[;<параметр(ы)> : тип]); [virtual];

или = Destructor ИмяМетода[(<параметр(ы)> : тип)]; [virtual];

Объектный тип может наследовать компоненты другого объектного типа. Объект наследования - потомок, а объект, от которого произошло наследование — предок. Область видимости объектного типа состоит из него самого и всех его потомков.

Файлы

File (зарезервированное слово)

Файловый тип состоит из линейной последовательности компонентов любого типа, кроме файлового.

Синтаксис:

file of тип

или

file

Замечания:

Если слово of и тип компонента опущены, то такое описание обозначает нетипизированный файл. Предопределенный файл типа Text определяет файл содержащие символы, организованные в строки.

Пример:

(* Объявления файловых типов *)

Type Person = Record

FirstName : String[15];

LastName : String[25];

Address : String[35];

End;

PersonFile = File Of Person;

NumberFile = File Of Integer;

SwapFile = File;

Целочисленные типы

В Turbo Pascal предопределено пять целочисленных типов. Каждый тип обозначает подмножество целых чисел:

Тип	Диапазон	Формат
Shortint	-128..127	Знаковый 8 бит
Integer	-32768..32767	Знаковый 16 бит
Longint	-2147483648..2147483647	Знаковый 32 бита
Byte	0..255	Беззнаковый 8 бит
Word	0..65535	Беззнаковый 16 бит

Все целочисленные типы являются порядковыми.

Переменные

Var (зарезервированное слово)

Объявление переменной связывает идентификатор и его тип с областью памяти, могут храниться значения этого типа.

Синтаксис:

Var идентификатор, … идентификатор : тип;
идентификатор, … идентификатор : тип;

Замечания:

Для указания абсолютного адреса в памяти может использоваться зарезервированное слово Absolute.

Зарезервированное слово Var также используется для объявления переменного параметра.

Примеры:

{ Объявления переменных }

Var X, Y, Z : Real;

I, J, K : Integer;

Done, Error : Boolean;

Vector : Array[1..10] Of Real;

Name : String[15];

InFile, OutFile : Text;

Letters : Set of 'A'..'Z';

Постоянные выражения

Постоянное выражение — это выражение, которое может быть оценено компилятором без непосредственного запуска программы.

Поскольку компилятор должен быть способен оценить постоянное выражение во времени компиляции, то следующие конструкции не позволяются в постоянных выражениях:

- Ссылки на переменные и типизированные константы (за исключением выражений базового адреса)
- Оператор @ (за исключением выражений базового адреса)
- Обращения к функциям (за исключением следующих)

Константы

Описания констант (Const: зарезервированное слово)

Описание константы содержит идентификатор, который обозначает константу внутри блока, в котором происходит объявление. Идентификатор константы не может быть включен в свое собственное объявление.

Синтаксис:

```
Const идентификатор = значение;
      идентификатор = значение;
```

Замечания:

Выражение "значение", используемое в описании константы должно быть написано так, чтобы компилятор смог оценить его значение во время компиляции. В Turbo Pascal позволяются постоянные выражения в качестве расширения стандартного Pascal.

Для объявления инициализированной переменной можно использовать типизированные константы. В отличие от нетипизированных констант, при объявлении типизированной константы можно задать и тип, и значение константы. Типизированные константы могут изменяться при работе программы точно так же как и переменные.

Примеры:

```
(* Описания констант *)
Const MaxData  = 1024 * 64 - 16;
NumChars = Ord('Z') - Ord('A') + 1;
Message = 'Hello world !';
```

Типизированные константы

Типизированные константы подобны инициализированным переменным (переменным с заданными начальными значениями).

В отличие от нетипизированных констант, объявление типизированной константы определяет сразу и тип, и значение константы. Типизированные константы могут использоваться подобно переменным того же самого типа, и могут находиться в левой части оператора присваивания.

Обратите внимание: типизированные константы инициализируются только один раз — при их создании в начале программы. При каждом входе в процедуру или функцию, локальные типизированные константы не инициализируются повторно.

В дополнение к стандартным константам, значение типизированной константы может быть задано выражением базового адреса.

Примеры:

(* Объявления типизированных констант *)

```
Type Point = Record
X, Y : Real
End;


Const Minimum : Integer = 0;
Maximum : Integer = 9999;
Factorial : Array [1..7] Of Integer = (1, 2, 6, 24, 120, 720, 5040);
HexDigits : Set Of Char = ['0'..'9', 'A'..'Z', 'a'..'z'];
Origin : Point = (X : 0.0; Y : 0.0);
```

Константы с простым типом

Объявление константы с простым типом содержит значение константы. Значение типизированной константы может быть определено, с использованием выражения базового адреса. Поскольку типизированная константа фактически является переменной с заданным начальным значением, то она не может быть заменена на обычную константу.

Примеры:

```
Const Maximum : Integer = 9999;
Factor : Real = -0.1;
BreakChar : Char = #3;
```

Константы типа записей

Объявление константы типа записи содержит идентификатор и значение каждого поля записи. Поля должны быть определены в том же порядке, в котором они были объявлены при определении типа записи.

- Если запись содержит поля файлового типа, то константы такого типа записи не могут быть объявлены.
- Если запись содержит вариант, то определены могут быть только поля указанного варианта.
- Если вариант содержит поле метки, то ее значение должно быть задано.

Примеры:

```
Type Point = Record
```

```
X, Y : Real;
End;
Vector = Array [0..1] Of Point;
Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
Date = Record
D : 1..31;
M : Month;
Y : 1900..1999;
End;
Const Origin : Point = (X : 0.0; Y : 0.0);
Line : Vector = ((X : -3.1; Y : 1.5), (X : 5.8; Y : 3.0));
SomeDay : Date = (D : 2; M : Dec; Y : 1960);
```

Константы типа наборов

Объявление константы типа набора содержит значение набора, заданное с использованием постоянного выражения.

Примеры:

```
Type Digits = Set Of 0..9;
Letters = Set Of 'A'..'Z';
Const EvenDigits : Digits = [0, 2, 4, 6, 8];
Vowels : Letters = ['A', 'E', 'I', 'O', 'U', 'Y'];
HexDigits : Set Of '0'..'z' = ['0'..'9', 'A'..'F', 'a'...f'];
```

Константы типа массивов

Объявление константы типа массива определяет значения компонентов массива. Тип компонентов массива может быть любым, кроме файлового.

Пример:

```
Type Status = (Active, Passive, Waiting);
StatusMap = Array [Status] Of String[7];
Const StatStr : StatusMap = ('Active', 'Passive', 'Waiting');
{ компоненты StatStr:
StatStr[Active] = 'Active'
StatStr[Passive] = 'Passive'
StatStr[Waiting] = 'Waiting' }
```

Символьные массивы:

Упакованные константы со строковым типом (символьные массивы) могут быть определены и как одиночные символы, и как строки. Например, такое определение:

Const Digits : Array [0..9] Of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

может быть выражено более коротко:

Const Digits : Array [0..9] Of Char = '0123456789';

Нуль-основанные символьные массивы:

Нуль-основанный символьный массив - это такой массив, в котором индекс первого элемента равен нулю, а последнего - положительному ненулевому целому числу. Например:

Array [0..X] Of Char;

Если вы включаете расширенный синтаксис (с помощью директивы компилятора {$X+}), то нуль-основанный символьный массив может быть инициализирован строкой, длина которой меньше, чем объявленная длина массива. Например:

Const FileName = Array [0..79] Of Char = 'TEST.PAS';

Если строка короче, чем длина массива, то оставшиеся символы устанавливаются равными NULL (0), и массив будет содержать строку с нулевым окончанием.

Константы — многомерные массивы:

Такие константы определяются, заключением констант каждой размерности в отдельные наборы круглых скобок, разделенные запятыми.

Например, такое объявление:

Type Cube = Array[0..1, 0..1, 0..1] Of Integer;

Const Maze : Cube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));

создает инициализированный массив Maze:

Maze[0, 0, 0] = 0

Maze[0, 0, 1] = 1

Maze[0, 1, 0] = 2

Maze[0, 1, 1] = 3

Maze[1, 0, 0] = 4

Maze[1, 0, 1] = 5

Maze[1, 1, 0] = 6

Maze[1, 1, 1] = 7

Константы со строковым типом

Объявление строковой константы содержит максимальную длину строки и ее начальное значение:

Примеры:

```
Const Heading : String[7] = 'Section';
NewLine : String[2] = #13#10;
TrueStr : String[5] = 'Yes';
FalseStr : String[9] = 'No';
```

Константы со структурным типом

Объявление константы со структурным типом содержит значение каждого из компонентов структуры. Borland Pascal поддерживает объявления констант типов Array, Record, Set и Pointer. Константы типа типизированных файлов и константы массивов и записей, содержащих файловые компоненты, не допускаются.

Константы объектных типов

Объявление константы объектного типа использует тот же самый синтаксис как и объявление константы типа записи. Для компонентов метода нельзя задать начального значения.

Примеры:

```
Const ZeroPoint : Point = (X : 0; Y : 0);
ScreenRect : Rect = (A : (X : 0; Y : 0); B : (X : 80; Y : 25));
CountField : NumField = (X : 5; Y : 20; Len : 4; Name : Nil;
Value : 0; Min : -999; Max : 999);
```

Константы объектных типов, содержащих виртуальные методы, не должны инициализироваться через обращение к конструктору — такая инициализация производится компилятором автоматически.

Константы указательных типов

При объявлении константы указательного типа обычно используется выражение базового адреса для определения значения указателя. Если вы включаете расширенный синтаксис (с помощью директивы компилятора {$X+}), то типизированная константа типа PChar может быть инициализирована значением строковой константы.

Примеры:

```
Type Direction = (Left, Right, Up, Down);
StringPtr = ^String;
```

```
NodePtr = ^Node;
Node = Record
Next : NodePtr;
Symbol : StringPtr;
Value : Direction;
End;
Const S1 : String[4] = 'DOWN';
S2 : String[2] = 'UP';
S3 : String[5] = 'RIGHT';
S4 : String[4] = 'LEFT';
N1 : Node = (Next : NIL; Symbol : @S1; Value : Down);
N2 : Node = (Next : @N1; Symbol : @S2; Value : Up);
N3 : Node = (Next : @N2; Symbol : @S3; Value : Right);
N4 : Node = (Next : @N3; Symbol : @S4; Value : Left);
DirectTable : NodePtr = @N4;
```

Константы процедурного типа

Процедурная константа должна содержать идентификатор процедуры или функции, который совместим с типом константы.

Пример:

```
Type ErrorProc = Procedure(ErrorCode : Integer);
Procedure DefaultError(ErrorCode : Integer); Far;
Begin
WriteLn('Ошибка ', ErrorCode, '.');
End;


Const ErrorHandler : ErrorProc = DefaultError;
```

Стандартная директива Absolute

Используйте зарезервированное слово Absolute, чтобы объявить абсолютную переменную (которая находится по заданному или абсолютному адресу в памяти).

Синтаксис:

```
Var идентификатор : тип Absolute сегмент:смещение;
```

или

Var идентификатор : тип Absolute переменная;

Замечания:

Первая форма непосредственно задает адрес (сегмент и смещение) переменной. Оба значения должны быть внутри диапазона $0000..$FFFF (от 0 до 65,535).

Вторая форма объявления размещает новую переменную поверх существующей переменной (по тому же самому адресу).

Если в списке переменных присутствует слово Absolute, то в данной строке может находиться только одна переменная.

Предостережение:

Используйте первую форму объявления осторожно. Во время работы Windows в защищенном режиме, ваше приложение не может иметь прав доступа к областям памяти вне вашей программы. При попытке обращения к этим областям, выполнение вашей программы, вероятно, будет приостановлено.

Вторая форма объявления абсолютных переменных безопасна для приложений Windows. Память, к которой вы обращаетесь находится внутри области памяти вашей программы.

Пример:

Type VectorTable = Array [0..255] Of Pointer;

Var IntVectors : VectorTable Absolute 0:0;

CrtMode : Byte Absolute $0040:$0049;

Str : String;

StrLen : Byte Absolute Str;

2.5 Бейсик (BASIC)

Имя переменной начинается с латинской буквы, затем одна или несколько букв или цифр в произвольном сочетании.

Вещественные числа занимают 4 байта — двойное машинное слово в экспоненциальном представлении (отдельно кодируется мантисса и порядок), что позволяет хранить числа в диапазоне 10^{-30} до 10^{+30}, с точностью более 6 десятичных знаков.

Целые переменные и константы отмечаются в наименовании записью справа символа %. Целое число занимает 2 байта — машинное слово, таким образом целое число задается в диапазоне от –32768 до +32767.

Последний тип — строковые данные, задаваемые строками переменной длины и состоящие из собственно последовательности символов и текущей длины. Длина может меняться от 0 (пустая строка) до некоторого максимального значения (обычно 255). Для указания интерпретатору признак символьной переменной права от имени добавляют знак $ (знак доллар). Значения строковых переменных заключаются в кавычки ("), в некоторых реализациях в апострофы ('). Чаще всего знак апострофа используется как комментарий.

Бейсик не требует предварительного описания типа переменной, а при первом присвоении отводит место в памяти для нее. По умолчанию все числовые переменные равны нулю, а строковые — пустой строке ("").

Массив значений задается оператором DIM. Особенностью языка считается, что индексы массива начинают задаваться с нуля. Таким образом DIM M(10) задает 11 индексов 0..10.

2.6 Пролог (Prolog)

Объекты данных

На рис. Приведена классификация объектов данных Пролога. Пролог-система распознает тип объекта по его синтаксической формы. Это возможно благодаря тому, что синтаксис Пролога предписывает различные формызаписи для различных типов объектов данных. В гл. 1 мы уже видели способ, с помощью которого можно отличить атомы от переменных: переменные начинаются с прописной буквы, тогда как атомы – со строчной. Для того, чтобы Пролог-система распознала тип объекта, ей не требуется сообщать больше никакой дополнительной информации (такой, например, как объявление типа данных). Синтаксически все объекты данных в Прологе представляют собой термы.

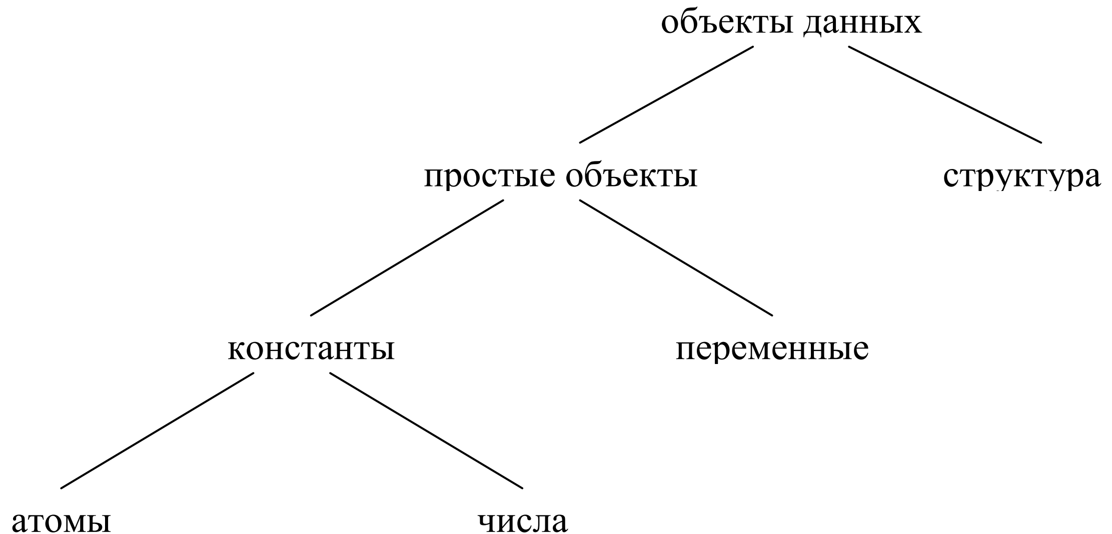


Рисунок 15 – Объекты данных Пролога.

Атомы и числа.

Вообще же атомы могут представлять собой цепочки следующих символов:

- прописные буквы A, B, …, Z
- строчные буквы a, в, …, z
- цифры 0, 1, 2, …, 9
- специальные символы, такие как + - * / < > = : . & _ ~

Атомы можно создавать тремя способами:

(1) из цепочки букв, цифр и символа подчеркивания _, начиная такую цепочку со строчной буквы:

анна

nil

x 25

x_25

x__y

альфа_бета_процедура

мисс_Джонс

сара_джонс

(2) из специальных символов:

<--->

=========>

…

.:.

::=

Пользуясь атомами такой формы, следует соблюдать некоторую осторожность, поскольку часть цепочек специальных символов имеют в Прологе заранее определенный смысл. Примером может служить :-.

(3) из цепочки символов, заключенной в одинарные кавычки. Это удобно, если мы хотим, например, иметь атом, начинающийся с прописной буквы. Заключая его в кавычки, мы делаем его отличным от переменной:

'Том'

'Южная_Америка'

'Сара Джонс'

Числа в Прологе бывают целыми и вещественными. Синтаксис целых чисел прост, как это видно из следующих примеров: 1, 1313, 0, -97. Не все целые числа могут быть представлены в машине, поэтому диапазон целых чисел ограничен интервалом между некоторыми минимальным и максимальным числами, определяемыми конкретной реализацией Пролога. Обычно реализация допускает диапазон хотя бы от –16383 до 16383, а часто, и значительно более широкий.

Синтаксис вещественных чисел зависит от реализации. При обычном программировании на прологе вещественные числа используются редко.

Причина этого кроется в том, что Пролог — это язык, предназначенный в первую очередь для обработки символьной информации. При символьной обработке части используются целые числа, например, для подсчета количества элементов списка; нужда же в вещественных числах невелика.

Переменные

Переменные — это цепочки, состоящие из букв, цифр и символов подчеркивания. Они начинаются с прописной буквы или символа подчеркивания:

X

Результат

Объект2

Список_участников

_х23

_23

Всякий раз, когда в предложении появляется одиночный символ подчеркивания, он обозначает новую анонимную переменную. Если анонимная переменная встречается в вопросе, то ее значение не выводится при ответе системы на этот вопрос.

Лексический диапазон имени — одно предложение. Это значит, что если, например, имя X15 встречается в двух предложениях, то оно обозначает две разные переменные. Однако внутри одного предложения каждое его появление обозначает одну и ту же переменную. Для констант ситуация другая: один и тот же атом обозначает один и тот же объект в любом предложении, иначе говоря, — во всей программе.

Структуры

Структурные объекты (или просто структуры) — это объекты, которые стоят из нескольких компонент. Эти компоненты, в свою очередь, могут быть структурами. Например, дату можно рассматривать как структуру, состоящую из трех компонент: день, месяц, год. Хотя они и составлены из нескольких компонент, структуры в программе ведут себя как единые объекты. Для того, чтобы объединить компоненты в структуру, требуется выбрать функтор. Например, для даты подойдет функтор **дата**. Тогда дату 1-е мая 2001 года можно записать так:

дата (1, май, 2001)

Рисунок 16 – Дата — пример структурного объекта.

Все структурные объекты можно изображать в виде деревьев (рисунок). Корнем дерева служит функтор, ветвями, исходящими из него, — компоненты. Если некоторая компонента тоже является структурой, тогда ей соответствует поддерево в дереве, изображающей весь структурный объект.

Списки

Список — это последовательность, составленная из произвольного числа элементов, например, **энн, теннис, том, лыжи**. На Прологе это записывается так:

[**энн, теннис, том, лыжи**]

Однако таково лишь внешнее представление списков. Все структурные объекты Пролога — это деревья. Списки не являются исключением из этого правила. Каким образом можно представить список в виде стандартного прологовского объекта? Мы должны рассмотреть два случая: пустой список и не пустой список. В первом случае список записывается как атом []. Во втором случае список следует рассматривать как структуру, состоящую из двух частей:

(1) первый элемент, называемой *головой* списка;

(2) остальная часть списка, называемая *хвостом*.

Например, для списка

[**энн, теннис, том, лыжи**]

энн — это голова, а хвостом является список

[теннис, том, лыжи]

В общем случае, головой может быть что угодно (любой прологовоский объект, например, дерево или переменная); хвост же должен быть списком. Голова соединяется с хвостом при помощи специального функтора. Выбор этого функтора зависит от конкретной реализации Пролога; будем считать, что это точка:

.(Голова, Хвост)

Поскольку **Хвост** — это список, он либо пуст, либо имеет свои собственную голову и хвост. Таким образом, выбранного способа представления списков достаточно для представления списков любой длины. Наш список представляется следующим способом:

.(**энн, .(теннис, .(том, .(лыжи, []**))))

Видно, что такой способ представления в случае большой глубины вложенности подэлементов в хвостовой части списка может привести к довольно запутанным выражениям. Поэтому в Прологе предусматривается более лаконичный способ изображения списков, при котором они записываются как последовательности элементов, заключенных в квадратные скобки. Программист может использовать оба способа, но представление с квадратными скобками, конечно, в большинстве случаев пользуется предпочтением. Необходимо помнить, что это всего лишь косметическое улучшение и что во внутреннем представлении наши списки выглядят как деревья. При выводе же они автоматически преобразуются в более лаконичную форму.

На практике часто бывает удобным трактовать хвост списка как самостоятельный объект. Например, пусть

L = [a, b, c]

Тогда можно написать:

Хвост = [b, c] и L = .(a, Хвост)

Для того, чтобы выразить это при помощи квадратных скобок, в Прологе предусмотрено еще одно расширение нотации для представления списка, а именно вертикальная черта, отделяющая голову от хвоста:

L = [a | Хвост]

На самом деле вертикальная черта имеет более общий смысл: мы можем перечислить любое количество элементов списка, затем поставить символ "|", а после этого — список остальных элементов. Так, только что рассмотренный пример можно представить различными способами:

[a, b, c] = [a| [b, c]] = [a, b | [c]] = [a, b, c | []]

Подытожим:

- Список — это структура данных, которая либо пуста, либо состоит из двух частей: *головы* и *хвоста*. Хвост, в свою очередь сам является списком.
- Список рассматривается в Прологе как специальный частный случай двоичного дерева. Для повышения наглядности программ в Прологе предусматриваются специальные средства для списковой нотации, позволяющие представлять списки в виде

[Элемент1, Элемент2, …]

или

[Голова | Хвост]

или

[Элемент1, Элемент2, … | Остальные]

3. ДЕРЕВЬЯ. ПРИКЛАДНЫЕ АЛГОРИТМЫ

Деревья представляют собой иерархическую структуру некой совокупности элементов. Знакомым вам примерами деревьев могут служить генеалогические и организационные диаграммы. Деревья используются при анализе электрических цепей, при представлении структур математических формул. Они также естественным путем возникают во многих областях компьютерных наук. Например, деревья используются для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ.

3.1. Основная терминология

Дерево — это совокупность элементов, называемых узлами (один из которых определен как *корень*), и отношений ("родительских"), образующих иерархическую структуру узлов. Узлы, так же, как и элементы списков, могут быть элементами любого типа. Будем изображать узлы буквами, строками или числами. Формально дерево можно рекуррентно определить следующим образом.

1. Один узел является деревом. Этот же узел также является корнем этого дерева.

2. Пусть n — это узел, а T_1, T_2, ..., T_k — деревья с корнями n_1, n_2, …, n_k соответственно. Можно построить новое дерево, сделав n родителем узлов n_1, n_2, ..., n_k. В этом дереве n будет корнем, а T_1, T_2, ..., T_k — поддеревьями этого корня. Узлы n_1, n_2, ..., n_k называются сыновьями узла n.

Часто в это определение включают понятие *нулевого дерева*, т.е. "дерева" без узлов, такое дерево будем обозначать символом Λ.

Рассмотрим оглавление книги, схематически представленное на рис.17. Это оглавление является деревом, которое в другой форме показано на рис.17. Отношение родитель-сын отображается в виде линии. Деревья обычно рисуются сверху вниз, как на рис. 17, так, что родители располагаются выше "детей".

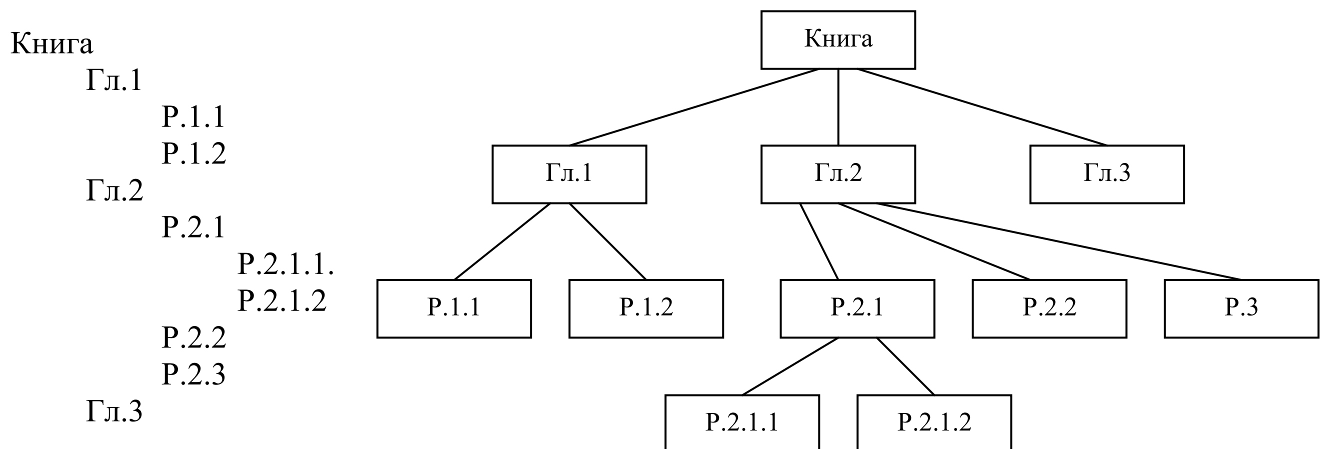


Рисунок 17 – Оглавление книги и его представление в виде дерева.

Корнем этого дерева является узел *Книга*, который имеет три поддерева соответственно с корнями *Гл.1*, *Гл.2*, *Гл.3*. Эти отношения показаны линиями, идущими из корня *Книга* к узлам *Гл.1*, *Гл.2*, *Гл.3*. Узел *Книга* является родителем узлов *Гл.1*, *Гл.2*, *Гл.3*, а эти три узла — сыновьями узла *Книга*.

Третье поддерево, с корнем *Гл.3*, состоит из одного узла, остальные два поддерева имеют нетривиальную структуру. Например, поддерево с корнем *Гл.2* в свою очередь имеет три поддерева, которые соответствуют разделам книга *Р.2.1*, *Р.2.2.*, *Р.2.3*; последние два поддерева, соответствующие подразделам книги *Р.2.1.1* и *Р.2.1.2*.

Пример показывает типичные данные, для которых наилучшим представлением будут деревья. В этом примере родительские отношения устанавливают подчиненность глав и разделов: родительский узел связан с узлами сыновей (и указывает на них), как узел Книга подчиняет себе узлы *Гл.1*, *Гл.2* и *Гл.3*.

Путем из узла n_1 в узел n_k называется последовательность узлов n_1, n_2, …, n_k, где для всех i, $1 \le i \le k$, узел n_i является родителем узла n_{i+1}. Длиной пути называется число, на единицу меньшее числа узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе. На рис. 17 путем длины 2 будет, например, путь от узла *Гл.2* к узлу *Р.2.1.2*.

Если существует путь из узла *a* в *b*, то в этом случае узел a называется предком узла b, а узел b — потомком узла a. Например, на рис.17 предками узла *Р.2.1* будут следующие узлы: сам узел *Р.2.1* и узлы *Гл.2* и *Книга*, тогда как потомками этого узла являются опять сам узел *Р.2.1* и узлы *Р.2.1.1* и *Р.2.1.2*. Отметим, что любой узел одновременно является и предком, и потомком самого себя.

Предок или потомок узла, не являющийся таковым самого себя, называется *истинным предком* или *истинным потомком* соответственно. В дереве только корень не имеет истинного предка. Узел, не имеющий истинных потомков, называется *листом*. Теперь поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа. На рис. 17 высота узла *Гл.1* равна 1, узла *Гл.2* — 2, а узла *Гл.3* — 0. *Высота дерева* совпадает с высотой корня. *Глубина узла* определяется как длина пути (он единственный) от корня до этого узла.

Порядок узлов

Сыновья узла обычно упорядочиваются слева направо. Поэтому два дерева на рис. 18 различны, так как порядок сыновей узла **a** различен. Если порядок сыновей игнорируется, то такое дерево называется *неупорядоченным*, в противном случае дерево называется *упорядоченным*.

Упорядочивание слева направо сыновей ("родных детей" одного узла) можно использовать для сопоставления узлов, которые не связаны отношениями предков потомки. Соответствующее правило звучит следующим образом: если узлы a и b являются сыновьями одного родителя и узел a лежит слева от узла b, то все потомки узла a будут находиться слева от любых потомков узла b.

Существует простое правило, позволяющее определить, какие узлы расположены слева от данного узла n, а какие — справа. Для этого надо прочертить путь от корня дерева до узла n. Тогда все узлы и их потомки, расположенные слева от этого пути, будут находиться слева от узла n, и, аналогично, все узлы и их потомки, расположенные справа от этого пути, будут находиться справа от узла n.

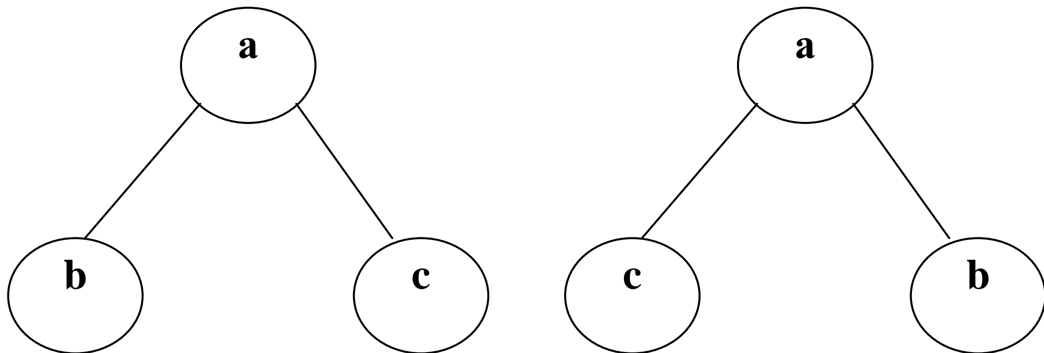


Рисунок 18 – Два различных упорядоченных дерева.

Прямой, обратный и симметричный обходы дерева

Существует несколько способов обхода (прохождения) всех узлов дерева. Три наиболее часто используемых способа обхода дерева называются обход в прямом порядке, обход в обратном порядке и обход во внутреннем порядке (последний вид обхода также часто называют симметричным обходом, будем использовать названия как синонимы). Все три способа обхода рекурсивно можно определить следующим образом:

- Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.
- Если дерево T состоит из одного узла, то в список обхода записывается этот узел.
- Далее, пусть T — дерево с корнем n и поддеревьями T_1, T_2, ..., T_k, как показано на рис. 18. Тогда для различных способов обхода имеем следующее.

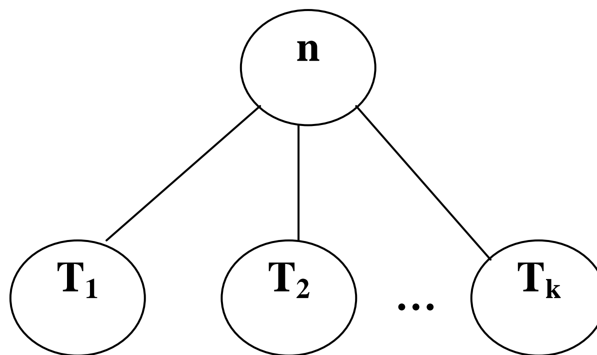


Рисунок 19 – Дерево T.

Обход узлов дерева равнозначен упорядочиванию по какому-либо правилу этих узлов. Поэтому в данном разделе будем использовать слова "обход узлов" и "упорядочивание узлов" как синонимы.

1. При *прохождении в прямом порядке* (т.е. при *прямом упорядочивании*) узлов дерева T сначала посещается корень n, затем узлы поддерева T_1, далее все узлы поддерева T_2 и т.д. Последним посещаются узлы поддерева T_k.

2. При *симметричном обходе* узлов дерева T сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень n, затем последовательно в симметричном порядке все узлы T_2, …, T_k.

3.Во время *обхода в обратном порядке* сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев T_2, …, T_k, также в обратном порядке,

последним посещается корень n.

В листинге показан набросок процедуры PREORDER (Прямое упорядочивание), составляющей список узлов дерева при обходе его в прямом порядке. Чтобы из этой процедуры сделать процедуру, выполняющую обход дерева в обратном порядке, надо просто поменять местами строки (1) и (2). В листинге представлен набросок процедуры INORDER (Внутреннее упорядочивание). В этих процедурах производится соответствующее упорядочивание деревьев путем вызова соответствующих процедур к корню дерева.

Листинг. рекурсивные процедуры обхода деревьев

```
procedure PREORDER (n: узел );
begin
(1)       занести в список обхода узел n;
(2)        for для каждого сына с узла n в порядке слева направо do
PREORDER(c)
end; { PREORDER }

процедура INORDER
procedure INORDER ( n: узел );
begin
if n — лист then
занести в список обхода узел n;
else begin
INORDER(самый левый сын узла n) ;
занести в список обхода увел n;
for для каждого сына с узла n, исключая самый левый,
в порядке слева направо do
INORDER(c)
End
end; { INORDER }
```

Пример. Сделаем обход дерева, показанного на рис 19, в прямом порядке. Сначала запишем (посетим) узел 1, затем вызовем процедуру PREORDER для обхода первого поддерева с корнем 2. Это поддерево состоит из одного узла, которое и записываем. Далее обходим второе поддерево, выходящее из корня 1, это поддерево имеет корень 3. Записываем узел 3-й вызываем процедуру PREORDER для обхода первого поддерева, выходящего из узла 3. В результате получим список узлов 5, 8 и 9 (именно в таком порядке). Продолжая этот процесс, в конце получим полный список узлов, посещаемых при прохождении в прямом порядке исходного дерева: 1, 2, 3, 5. 8, 9, 6, 10, 4 и 7.

Подобным образом, предварительно преобразовав процедуру PREORDER в процедуру, выполняющую обход в обратном порядке (как указано выше), можно получить обратное упорядочивание узлов дерева из рис. 19 в следующем виде: 2, 8, 9, 5, 10, 6, 3, 7, 4 и 1. Применяя процедуру INORDER, получим список симметрично упорядоченных узлов этого же дерева: 2, 1, 8, 5, 9, 3, 10, 6, 7 и 4.

При обходе деревьев можно применить следующий полезный прием. Надо нарисовать непрерывный контур вокруг дерева, начиная с корня дерева, рисуя конур против часовой стрелки и поочередно обходя все наружные части дерева.

При прямом упорядочивании узлов надо просто записать их в соответствии с нарисованным контуром. При обратном упорядочивании после записи всех сыновей переходим к их родителю. При симметричном (внутреннем) упорядочивании после записи самого правого листа переходим не по ветви в направлении корню дерева, а к следующему "внутреннему" узлу, который еще не записан. Например, если на рис. 19 узлы 2 и 1 уже записаны, то как бы перескакиваем "залив" между узлами 2 и 3 и переходим к узлу 8. Отметим, что при любом упорядочивании листья всегда записываются в порядке слева направо, при этом в случае симметричного упорядочивания между сыновьями может быть записан родитель.

Помеченные деревья и деревья выражений

Часто бывает полезным сопоставить каждому узлу дерева метку (label) или значение. Дерево, у которого узлам сопоставлены метки, называется *помеченным деревом*. Метка узла — это не имя узла, а значение, которое "хранится" в узле. В некоторых приложениях будем изменять значение метки, поскольку имя узла сохраняется постоянным. Полезна следующая аналогия: дерево-список, узел-позиция, метка-элемент.

Пример. На рисунке 20 показано дерево с метками, представляющее арифметическое выражение (a + b) * (a + c), где n1, …, n7 — имена узлов (метки на рисунках поставлены рядом с соответствующими узлами). Правила соответствия меток деревьев элементам выражений следующие.

1. Метка каждого листа соответствует операнду и содержит его значение, например, узел на n4 представляет операнд a.

2. Метка каждого внутреннего (родительского) узла соответствует оператору. Предположим, что узел n помечен бинарным оператором θ (например, + или *) и левый сын этого узла соответствует выражению E1, а правый — выражению E2. Тогда узел n и его сыновья представляют выражение (E1)θ(E2). Можно удалять родителей, если это необходимо.

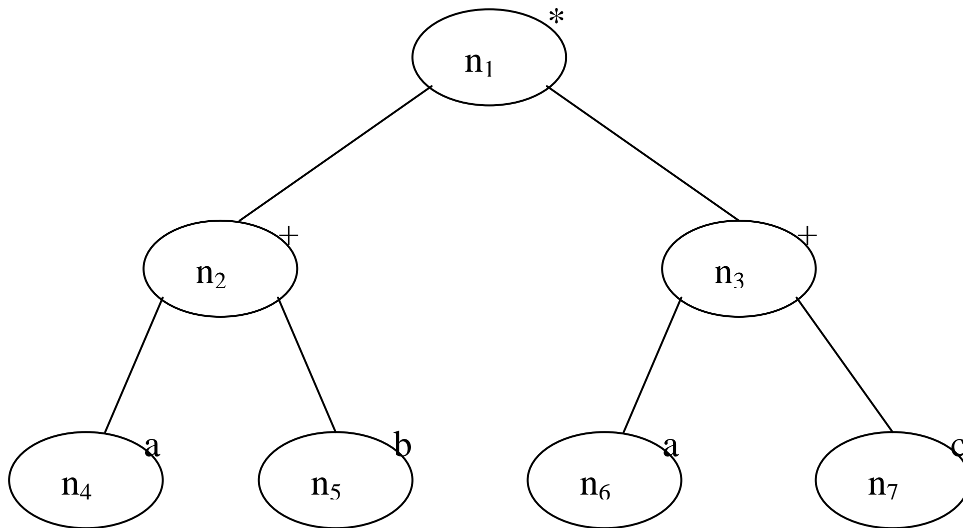


Рисунок 20 – Дерево выражения с метками.

Например, узел n2 имеет оператор +, а левый и правый сыновья представляют выражения (операнды) а и и соответственно. Поэтому узел n2 представляет (a) + (b),,т.е. a+b. Узел n1 представляет выражение (a+b) * (a+c), поскольку оператор * является меткой узла n1, выражения a+b и a+c представляются узлами n2 и n3 со ответственно.

Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом упорядочивании получаем известную префиксную форму выражений, где оператор предшествует и левому, и правому операндам. Для точного описания префиксной формы выражений сначала положим, что префиксным выражением одиночного операнда а является сам этот операнд. Далее, префиксная форма для выражения (E1)θ(E2), где θ — бинарный оператор, имеет вид θP1P2, здесь P1 и P2 — префиксные формы для выражений E1 и E2. Отметим, что в префиксных формах нет необходимости отделять или выделять отдельные префиксные выражения скобками, так как всегда можно просмотреть префиксное выражение θP1P2 и определить единственным образом P1 как самый короткий префикс выражения P1P2.

Например, при прямом упорядочивании узлов (точнее, меток) дерева, показанного на рис. 20, получаем префиксное выражение *+ab+ac. Самым коротким корректным префиксом для выражения +ab+ac будет префиксное выражение узла n2: +ab.

Обратное упорядочивание меток дерева выражений дает так называемое постфиксное (или польское) представление выражений. Выражение θP1P2 в постфиксной форме имеет вид P1P2θ, где P1 и P2 — постфиксные формы для выражений E1 и E2 соответственно. При использовании постфиксной формы также нет необходимости в применении скобок, поскольку для любого постфиксного выражения P1P2, легко проследить самый короткий суффикс P2 что и будет корректным

составляющим постфиксным выражением. Например, постфиксная форма выражения для дерева на рис.20 имеет вид ab+ac+*. Если записать это выражение как P1P2*, то P2 (т.е. выражение ac+) будет самым коротким суффиксом для ab+ac+* и, следовательно, корректным составляющим постфиксным выражением.

При симметричном обходе дерева выражений получим так называемую инфиксную форму выражения, которая совпадает с привычной "стандартной" формой записи выражений, но также не использует скобок. Для дерева на рис. 20 инфиксное выражение запишется как a + b * a + c. Читателю предлагается разработать алгоритм обхода дерева выражений, который бы выдавал инфиксную форму выражения со всеми необходимыми парами скобок.

Вычисление "наследственных" данных

Обход дерева в прямом или обратном порядке позволяет получить данные об отношениях предок-потомок узлов дерева. Пусть функция postorder(n) вычисляет позицию узла n в списке узлов, упорядоченных в обратном порядке. Например, для узлов n2, n4 и n5 дерева, представленного на рис. 20, значения этой функции будут 3, 1 и 2 соответственно. Определим также функцию desc(n), значение которой равно числу истинных потомков узла n.

Эти функции позволяют выполнить ряд полезных вычислений. Например, все узлы поддерева с корнем n будут последовательно занимать позиции от postorder(n) - desc(n) до postorder(n) в списке узлов, упорядоченных в обратном порядке. Для того, чтобы узел x был потомком узла y, надо, чтобы выполнялись следующие неравенства:

$$\text{postorder}(y) - \text{desc}(y) \le \text{postorder}(x) \le \text{postorder}(y).$$

Подобные функции можно определить и для списков узлов, упорядоченных в прямом порядке.

3.2. Абстрактный тип данных TREE

В предыдущих главах списки, стеки, очереди и отображения получили трактовку как абстрактные типы данных (АТД). В этом разделе рассмотрим деревья как АТД и как структуры данных. Одно из наиболее важных применений деревьев — это использование их при разработке реализации различных АТД. Например, далее покажем, как можно использовать "дерево двоичного поиска" при реализации АТД, основанных на математической модели множеств. В следующих двух главах будут представлены многочисленные примеры применения деревьев при реализации различных абстрактных типов данных.

В этом разделе представим несколько полезных операторов, выполняемых над деревьями, и покажем, как использовать эти операторы в различных алгоритмах. Так же, как и в случае списков, можно предложить

большой набор операторов, выполняемых над деревьями. Здесь рассмотрим следующие операторы.

1. PARENT(n, T). Эта функция возвращает родителя (parent) узла n в дереве T. Если n является корнем, который не имеет родителя, то в этом случае возвращается Λ. Здесь Λ обозначает "нулевой узел" и указывает на то, что выходим за пределы дерева.

2. LEFTMOST_CHILD(n, T). Данная функция возвращает самого левого сына узла n в дереве T. Если л является листом (и поэтому не имеет сына), то возвращается Λ.

3. RIGHT_SIBLING(n, T). Эта функция возвращает правого брата узла n в дереве T и значение Λ, если такового не существует. Для этого находится родитель p узла n и все сыновья узла p, затем среди этих сыновей находится узел, расположенный непосредственно справа от узла n. Например, для дерева LEFTMOST_CHILD(n2) = n4, RIGHT_SIBLING (n4)= n5 и RIGHT_SIBLING(n5)]= Λ.

4. LABEL(n, T). Возвращает метку узла n дерева T. Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.

5. CREATE(v, T1, T2, …, Ti) — это обширное семейство "созидающих" функций, которые для каждого i = 0, 1, 2, ... создают новый корень r с меткой v и далее для этого корня создает i сыновей, которые становятся корнями поддеревьев T1, T2, …, Ti. Эти функции возвращают дерево с корнем r. Отметим, что если i == 0, то возвращается один узел r, который одновременно является и корнем, и листом.

6. ROOT(T) возвращает узел, являющимся корнем дерева T. Если T — пустое дерево, то возвращается Λ.

7. MAKENULL(T). Этот оператор делает дерево T пустым деревом.

Пример. Напишем рекурсивную PREORDER и нерекурсивную NPREORDER процедуры обхода дерева в прямом порядке и составления соответствующего списка его меток. Предположим, что для узлов определен тип данных node (узел), так же, как и для типа данных TREE (Дерево), причем АТД TREE определен для деревьев с метками, которые имеют тип данных labeltype (тип метки). В листинге приведена рекурсивная процедура, которая по заданному узлу n создает список в прямом порядке меток поддерева, корнем которого является узел n. Для составления списка всех узлов дерева надо выполнить вызов PREORDER(ROOT(T)).

Листинг: Рекурсивная процедура обхода дерева в прямом порядке

```
Procedure PREORDER (n : node);
var
c: node;
begin
```

```
print(LABEL(n, T));
c:= LEFTMOST_CHILD(n, T) ;
while c <> Λ do begin
PREORDER(c) ;
c:= RIGHT_SIBLING(c, T)
end
end; { PREORDER }
```

Теперь напишем нерекурсивную процедуру для печати узлов дерева в прямом порядке. Чтобы совершить обход дерева, используем стек S, чей тип данных STACK уже объявлен как "стек для узлов". Основная идея разрабатываемого алгоритма заключается в том, что, когда дошли до узла n, стек хранит путь от корня до этого узла, причем корень находится на "дне" стека, а узел n — в вершине стека.

Один из подходов к реализации обхода дерева в прямом порядке показан на примере программы NPREORDER в листинге эта программа выполняет два вида операций, т.е. может находиться как бы в одном из двух режимов. Операции первого вида (первый режим) осуществляют обход по направлению к потомкам самого левого еще не проверенного пути дерева до тех пор, пока не встретится лист, при этом выполняется печать узлов этого пути и занесение их в стек.

Во втором режиме выполнения программы осуществляется возврат по пройденному пути с поочередным извлечением узлов из стека до тех пор, пока не встретится узел, имеющий еще "не описанного" правого брата. Тогда программа опять переходит в первый режим и исследует новый путь, начиная с этого правого брата.

Программа начинается в первом режиме с нахождения корня дерева и определения, является ли стек пустым. В листинге показан полный код этой программы.

Листинг: нерекурсивная процедура обхода дерева в прямом порядке

```
procedure NPREORDER ( T: TREE );
var
m : node; { переменная для временного хранения узлов }
S: STACK;{стек узлов, хранящий путь от корня до родителя TOP(5)
текущего узла m}
begin { инициализация }
MAKENULL(S);
m:= ROOT(T) ;
while true do
if т <> Λ then begin
print( LABEL (m, T) ) ;
```

```
PUSH(m, S) ;
{исследование самого левого сына узла m}
m:=LEFTMOST_CHILD(m, T)
end
else begin
{завершена проверка пути, содержащегося в стеке}
if EMPTY (S) then
return;
{исследование правого брата узла, находящегося в вершине стека}
m:= RIGTH_SIBLING (TOP (S), T);
POP (S)
end
end; {NPREJRDER}
```

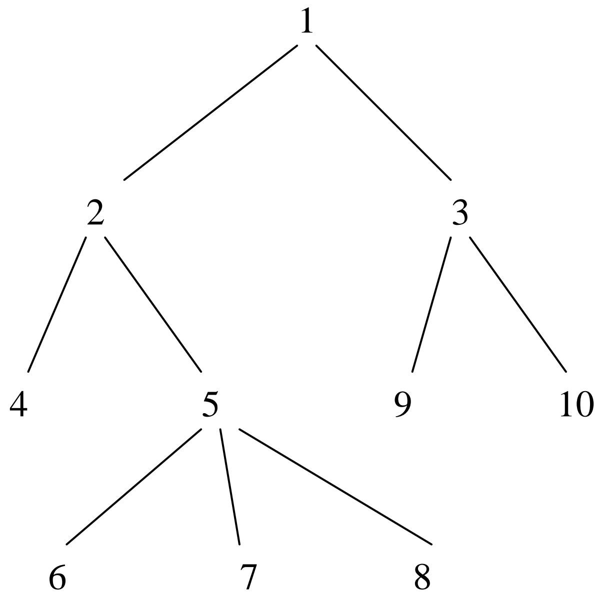
При рассмотрении листинга нетрудно заметить, что активационную запись можно заносить в стек при каждом вызове процедуры PREORDER(n) и, это главное, стек будет содержать активационные записи для всех предков узла n.

3.3. Реализация деревьев

В этом разделе представим несколько основных реализаций деревьев и обсудим их возможности для поддержки операторов.

Представление деревьев с помощью массивов

Пусть T — дерево с узлами 1, 2, …, n. Возможно, самым простым представлением дерева T, поддерживающим оператор PARENT (Родитель), будет линейный массив A, где каждый элемент A[i] является указателем или курсором на родителя i, корень узла дерева T отличается от других узлов тем, что имеет нулевой указатель или указатель на самого себя как на родителя. В языке Pascal указатели на элементы массива недопустимы, поэтому будем использовать схему с курсорами, тогда A[i] =j , если узел j является родителем узла i, и A[i] = 0, если узел i является корнем.



а. Дерево

	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	6	6	6	3	3

б. Курсоры на родителей

Рисунок 21 – Дерево и курсоры на родителей.

Данное представление использует то свойство деревьев, что каждый узел, отличный от корня, имеет только одного родителя. Используя это представление, родителя любого узла можно найти за фиксированное время. Прохождение по любому пути, т.е. переход по узлам от родителя к родителю, можно выполнить за время, пропорциональное количеству узлов пути. Для реализации оператора LABEL можно использовать другой массив L, в котором элемент L[i] будет хранить метку узла i, либо объявить элементы массива A записями, состоящими из целых чисел (курсоров) и меток.

Пример. на рис.21 показаны дерево и массив A курсоров на родителей этого дерева.

Использование указателей или курсоров на родителей не помогает в реализации операторов, требующих информацию о сыновьях. Используя описанное представление, крайне тяжело для данного узла n найти его сыновей или определить его высоту. Кроме того, в этом случае невозможно определить порядок сыновей узла (т.е. какой сын находится правее или левее другого сына). Поэтому нельзя реализовать операторы, подобные LEFTMOST_CHILD и RIGHT_SIBLING. Можно ввести искусственный порядок нумерации узлов, например нумерацию сыновей в

возрастающем порядке слева направо. Используя такую нумерацию, можно реализовать оператор RIGHT_SIBLING, код для этого оператора приведен в листинге. Для задания типов данных node (узел) и TREE (Дерево) используется следующее объявление:

```
type
node = integer;
TREE = array [1..maxnodes] of node;
```

В этой реализации предполагаем, что нулевой узел Λ представлен 0.

Листинг: оператор определения правого брата

```
procedure RIGHT_SIBLING ( п: node; T: TREE ) : node;
var
parent: node;
begin
parent:= T[п] ;
for i:= n + 1 to maxnodes do
if T[i] = parent then
return (i) ;
return(0) { правый брат не найден }
end; { RIGHT_SIBLING }
```

Представление деревьев с использованием списков сыновей

Важный и полезный способ представления деревьев состоит в формировании для каждого узла списка его сыновей. Эти списки можно представить любым методом, описанным ранее, но, так как число сыновей у разных узлов может быть разное, чаще всего для этих целей применяются связанные списки.

На рис. 22 показано, как таким способом представить дерево, изображенное на рис. 21, а. Здесь есть массив ячеек заголовков, индексированный номерами (они же имена) узлов. Каждый заголовок (header) указывает на связанный список, состоящий из "элементов"-узлов. Элементы списка header[i] являются сыновьями узла i, например узлы 9 и 10 — сыновья узла 3.

Прежде чем разрабатывать необходимую структуру данных, нам надо в терминах абстрактного типа данных LIST (список узлов) сделать отдельную реализацию списков сыновей и посмотреть, как эти абстракции согласуются между собой. Позднее увидим, какие упрощения можно сделать в этих реализациях. Начнем со следующих объявлений типов:

```
type
node = integer;
LIST = {соответствующее определение для списка узлов};
position = {соответствующее определение позиций в списках };
TREE = record
header : array [1..maxnodes] of LIST;
labels : array [1..maxnodes] of labeltype;
root : node
end;
header
```

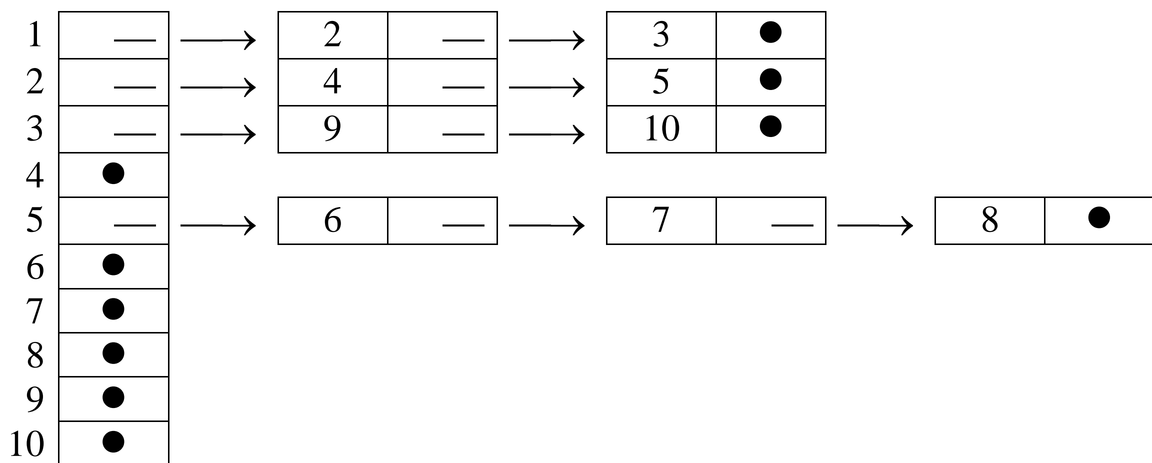


Рисунок 22 – Представление дерева с помощью связанных списков.

Предполагаем, что корень каждого дерева хранится отдельно в поле root (корень). Для обозначения нулевого узла используется 0.

В листинге представлен код функции LEFTMOST_CHILD. В качестве упражнения можете написать коды других операторов.

Листинг: функция нахождения самого левого сына

```
function LEFTMOST_CHILD ( n: node; T: TREE ): node;
{ возвращает самого левого сына узла п дерева Т }
var
L: LIST; { скоропись для списка сыновей узла n }
begin
L:= T.header[n] ;
if EMPTY(L) then { n является листом }
return(0)
else
return(RETRIEVE(FIRST(L), L})
end; { LEFTMOST_CHILD }
```

Теперь представим конкретную реализацию списков, где типы LIST и position имеют тип целых чисел, последние используются как курсоры в массиве записей cellspace (область ячеек):

```
var
cellspace: array [ 1. ..maxnodes] of record
node: integer
next: integer
end;
```

Для упрощения реализации можно положить, что списки сыновей не имеют ячеек заголовков. Точнее, поместим T.header[n] непосредственно в первую ячейку списка, как показано на рис 22. В листинге представлен переписанный (с учетом этого упрощения) код функции LEFTMOST_CHILD, а также показан код функции PARENT, использующий данное представление списков. Эта функция более трудна для реализации, так как определение списка, в котором находится заданный узел, требует просмотра всех списков сыновей.

Листинг: функции, использующие представление деревьев посредством связанных списков

```
function LEFTMOST_CHILD ( л: node; T: TREE ): node;
{ возвращает самого левого сына узла n дерева T }
var
L: integer; { курсор на начало списка сыновей узла n}
begin
L:= T.header [n] ;
if L = 0 then { n является листом }
return(0)
else
return(cellspace[L].node)
end; { LEFTMOST_CHILD }
function PARENT ( л: node; T: TREE ): node;
{ возвращает родителя узла n дерева Г }
var
p: node; { пробегает возможных родителей узла n }
i: position; { пробегает список сыновей p }
begin
for p:== 1 to maxnodes do begin
i:= T.header [p] ;
while 1 <> 0 do { проверка на наличие сыновей узла p }
if cellspace[i].node = n then
```

```
return(p)
else
i:== cellspace[i] .next
end;
return(0) { родитель не найден }
end; { PARENT }
```

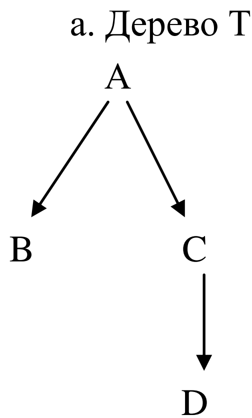
Представление левых сыновей и правых братьев

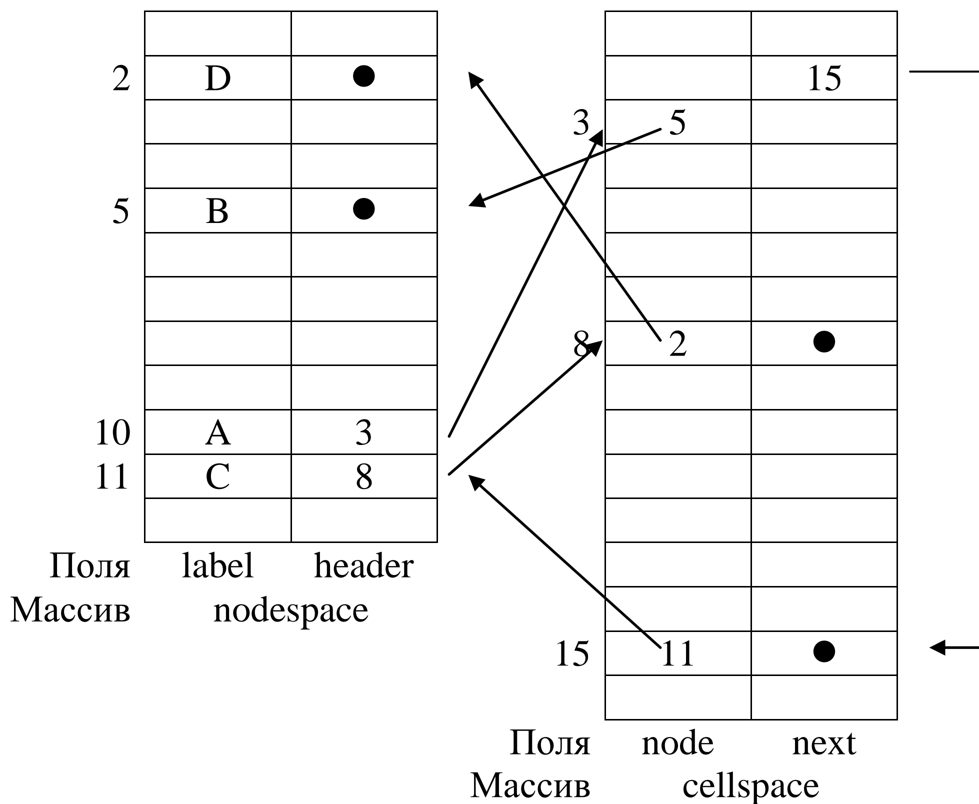
Среди прочих недостатков описанная выше структура данных не позволяет также с помощью операторов CREATED создавать большие деревья из малых. Это является следствием того, что все деревья совместно используют массив cellspace для представления связанных списков сыновей; по сути, каждое дерево имеет собственный массив заголовков для своих узлов. А при реализации, например, оператора CREATE2(v, T1, T2) надо скопировать деревья T1 и T2 в третье дерево и добавить новый узел с меткой v и двух его сыновей — корни деревьев T1 и T2.

Если хотим построить большое дерево на основе нескольких малых, то желательно, чтобы все узлы всех деревьев располагались в одной общей области. Логическим продолжением представления дерева, показанного на рис. 22, будет замена массива заголовков на отдельный массив nodespace (область узлов), содержащий записи с произвольным местоположением в этом массиве. Содержимое поля header этих записей соответствует "номеру" узла, т.е. номеру записи в массиве cellspace, в свою очередь поле node массива cellspace теперь является курсором для массива nodespece, указывающим позицию узла. Тип TREE в этой ситуации просто курсор в массиве nodespace, указывающий позицию корня.

Пример. На рис. 23, а показано дерево, а на рис. 23,б - структура данных, где узлы этого дерева, помеченные как A, B, C, D, размещены в произвольном порядке размещены списки сыновей.

а. Дерево T





б. Структуры данных

Рисунок 23 – Структура данных для дерева, использующая связанные списки.

Структура данных, показанная на рис. 23, б, уже подходит для того, чтобы организовать слияние деревьев с помощью операторов CREATEi. Но и эту структуру можно значительно упростить. Для этого заметим, что

цепочка указателей поля next массива cellspace перечисляет всех правых братьев.

Используя эти указатели, можно найти самого левого сына следующим образом. Предположим, что cellspace[i].node =n. (Повторим, что "имя" узла, в отличие от его метки, является индексом в массиве nodespace и этот индекс записан в поле ceellspace[i].node.) Тогда указатель nodespace[n]. header указывает на ячейку самого левого сына узла n в массиве cellspace, поскольку поле node этой ячейки является именем этого узла в массиве nodespace.

Можно упростить структуру, если идентифицировать узел не с помощью индекса в массиве nodespace, а с помощью индекса ячейки в массиве cellspace, который соответствует данному узлу как сыну. Тогда указатель next (переименуем это поле в right_sibling — правый брат) массива cellspace будет точно указывать на правого брата, а информацию, содержащуюся в массиве nodespace, можно перенести в новое поле leftmost_child (самый левый сын) массива cellspace. Здесь тип TREE является целолочисленным типом и используется как курсор в массиве cellspace, указывающий на корень дерева. Массив cellspace можно описать как следующую структуру:

```
var
cellspace: array[I..maxnodes] of record
label: labeltype;
leftmost_chlld: integer ;
rlght__sibling: integer
end;
```

Пример. Новое представление для дерева, показанного на рис.24, а, схематически изображено на рис.24. узлы дерева расположены в тех же ячейках массива, что и на рис.24, б.

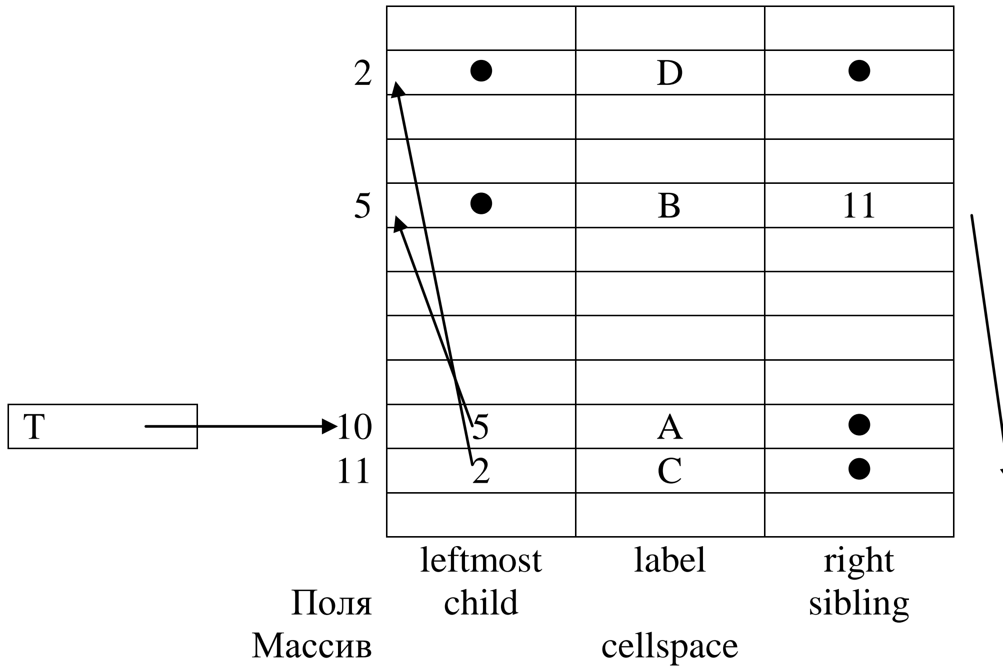


Рисунок 24 – Представление дерева посредством левых сыновей и правых братьев.

Используя описанное представление, все операторы, за исключением PARENT, можно реализовать путем прямых вычислений. Оператор PARENT требует просмотра всего массива cellspace. Если необходимо эффективное выполнение оператора PARENT, то можно добавить четвертое поле в массив cellspace для непосредственного указания на родителей.

В качестве примера операторов, использующих структуры данных рис.25, напишем код функции CREATE2, показанный в листинге. Здесь предполагаем, что неиспользуемые ячейки массива cellspace связаны в один свободный список, который в листинге назван как avail, и ячейки этого списка связаны посредством поля right_sibling. На рис. 3.11 показаны старые (сплошные линии) и новые (пунктирные линии) указатели в процессе создания нового дерева.

Листинг: Функция CREATE2

```
function CREATE2 ( v: labeltype; T1, T2: integer ): integer;
{ возвращает новое дерево с корнем v и поддеревьями T1 и T2 }
var
temp: integer; { хранит индекс первой свободной ячейки для корня
нового дерева }
begin
```

```
temp:= avail;
avall:= cellspace[avall] .right_sibling;
cell space [temp] .leftmost_child:= T1;
cell space [temp] .label:= v;
cellspace[temp].right_sibling:= 0;
cellspace[Tl].right_slbling:= T2;
cellspace[T2].right_sibling:= 0;{необязательный оператор}
return(temp)
end; { CREATE2 }
```

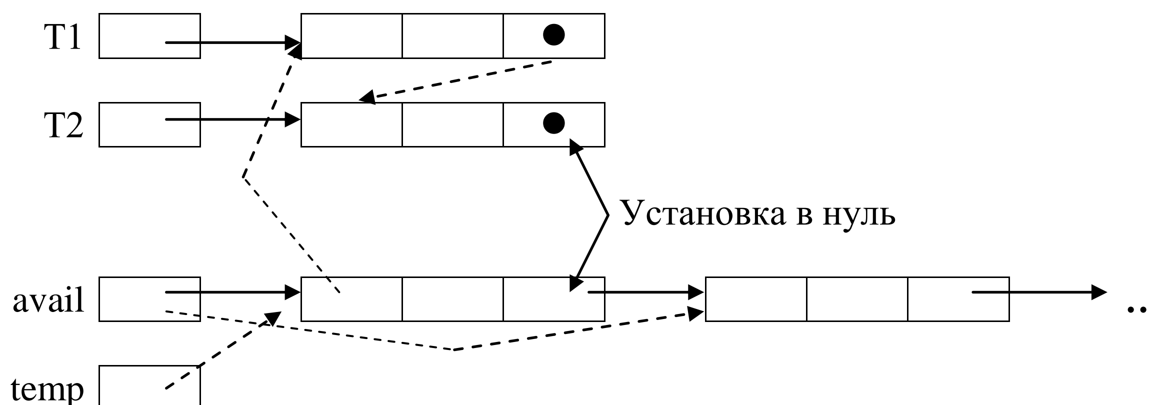


Рисунок 25 – Изменение указателей при выполнении функции CREATE2.

Можно уменьшить область памяти, занимаемую узлами дерева (но при этом увеличится время выполнения операторов), если в поле right_sibling самого правого сына вместо нулевого указателя поместить указатель на родителя. Но в этом случае, чтобы избежать двусмысленности, необходимо в каждую ячейку поместит еще двоичную (логическую) переменную, которая будет показывать, что содержится в поле right_sibling: указатель на правого брата или указатель на родителя.

При такой реализации можно найти для заданного узла его родителя, следуя за указателями поля right_sibling, пока не встретится указатель на родителя. В этом случае время, необходимое для поиска родителя, пропорционально количеству сыновей у родителя.

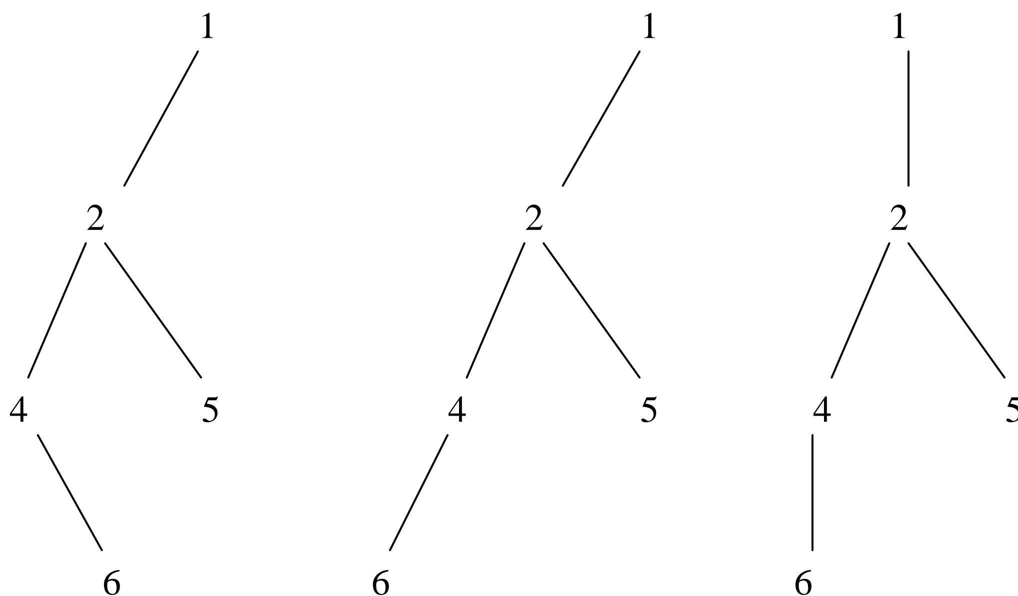
3.4. Двоичные деревья

Деревья, которые определены в начале главы, называются упорядоченными ориентированными деревьями, поскольку сыновья любого узла упорядочены слева направо, а пути по дереву ориентированы от начального узла пути к его потомкам. Двоичное (или бинарное) дерево — совершенно другой вид. Двоичное дерево может быть или пустым

деревом, или деревом, у которого любой узел или не имеет сыновей, или имеет либо левого сына, либо правого сына, либо обоих. Тот факт, что каждый сын любого узла определен как левый или как правый сын, существенно отличает двоичное дерево от упорядоченного ориентированного дерева.

Пример. Если примем соглашение, что на схемах двоичных деревьев левый сын всегда соединяется с родителем линией, направленной влево и вниз от родителя, а правый сын — линией, направленной вправо и вниз, тогда на рис.26, а,б представлены два различных дерева, хотя они оба похожи на (упорядоченное ориентированное) дерево, показанное на рис.26. Пусть не смущает тот факт, что деревья на рис.26,а, б различны и не эквивалентны дереву на рис. 26. Дело в том, что двоичные деревья нельзя непосредственно сопоставить обычному дереву. Например, на рис. 26,а узел 2 является левым сыном узла 1 и узел 1 не имеет правого сына, тогда как на рис. 26, б узел 1 не имеет левого сына, а имеет правого (узел 2). В тоже время в обоих двоичных деревьях узел 3 является левым сыном узла 2, а узел 4 — правым сыном того же узла 2.

Обход двоичных деревьев в прямом и обратном порядке в точности соответствует таким же обходам обычных деревьев. При симметричном обходе двоичного дерева с корнем n левым поддеревом T1 и правым поддеревом T2 сначала проходится поддерево T1, затем корень n и далее поддерево T2. Например, симметричный обход дерева на рис. 3.12, а даст последовательность узлов 3,5,2,4,1.



а б

Рисунок 26 – Два двоичных, дерева (а), "обычное" дерево (б).

Представление двоичных деревьев

Если именами узлов двоичного дерева являются их номера 1, 2, …, n, то подходящей структурой для представления этого дерева может служить массив cellspace записей с полями leftchild (левый сын) и rightchild (правый сын), объявленный следующим образом:

```
var
cellspace: array[1..maxnodes] of record
leftchild.: integer;
rightchild: integer
end;
```

В этом представлении cellspace[i]. leftchild является левым сыном узла i, а cellspace[i].rightchild — правым сыном. Значение 0 в обоих полях указывает на то, что узел i не имеет сыновей.

Пример. Двоичное дерево на рис.26,а можно представить в виде табл.

Представление двоичного дерева

	Значение поля leftchild	Значение поля rightchild
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Пример: коды Хаффмана

Приведем пример применения двоичных деревьев в качестве структур данных Для этого рассмотрим задачу конструирования кодов Хаффмана. Предположим, что имеем сообщения, состоящие из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении. Например, имеем сообщения, состоящие из пяти символов a, b, c, d, e, которые появляются в сообщениях с вероятностями 0.12, 0.4, 0.15, 0.08 и 0.25 соответственно.

Для того, чтобы закодировать каждый символ последовательностью из нулей и единиц так, чтобы код любого символа являлся префиксом кода сообщения, состоящего из нулей и единиц последовательным удалением префиксов (т.е. кодов символов) из этой строки.

Пример. В таблице показаны две возможные кодировки для наших пяти символов. Ясно, что первый код обладает префиксным свойством, поскольку любая последовательность из трех битов будет префиксом для

другой последовательности из трех битов; другими словами, любая префиксная последовательность однозначно идентифицируется символом. Алгоритм декодирования для этого кода очень прост: надо поочередно брать по три бита и преобразовать каждую группу битов в соответствующие символы. Например, последовательность 001010011 соответствует исходному сообщению bcd.

Таблица. Два двоичных кода

Символ	Вероятность	Код 1	Код 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

Легко проверить, что второй код также обладает префиксным свойством. Процесс декодирования здесь не отличается от аналогичного процесса для первого кода. единственная сложность для второго кода заключается в том, что нельзя сразу всю последовательность битов разбить на отдельные сегменты, соответствующие символам, так как символы могут кодироваться и двумя и тремя битами. Для примера рассмотрим двоичную последовательность 1101001, которая опять представляет символы bcd. два бита 11 однозначно соответствуют символу b, поэтому их можно удалить, тогда получится 01001. Здесь 01 также однозначно определяет символ c и т.д.

Задача конструирования кодов Хаффмана заключается в следующем: имея множество символов и значения вероятностей их появления в сообщениях, построить такой код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальной. Предположим, что хотим минимизировать среднюю длину кода для того, чтобы уменьшить длину вероятного сообщения (т.е. чтобы сжать сообщение). Чем короче среднее значение длины кода символов, тем короче закодированное сообщение. В частности, первый код из примера 3.11 имеет среднюю длину кода 3. Это число получается в результате умножения длины кода каждого символа на вероятность появления этого символа. Второй код имеет среднюю длину 2.2, поскольку символы a и d имеют суммарную вероятность появления 0.20 и длина их кода составляет три бита, тогда как другие символы имеют код длиной 2. Отсюда следует очевидный вывод, что символы с большими вероятностями появления должны иметь самые короткие коды

Можно ли придумать код, который был бы лучше второго кода? Ответ положительный: существует код с префиксным свойством, средняя длина которого равна 2.15. Это наилучший возможный код с теми же

вероятностями появления символов. Способ нахождения оптимального префиксного кода называется алгоритмом Хаффмана. В этом алгоритме находятся два символа a и b с наименьшими вероятностями появления и заменяются одним фиктивным символом, например x, который имеет вероятность появления, равную сумме вероятностей появления символов a и b. Затем, используя эту процедуру рекурсивно, находим оптимальный префиксный код для меньшего множества символов (где символы a и b заменены одним символом x). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа a будет соответствовать коду символа x с добавленным нулем перед этим кодом, а для кода символа b перед кодом символа x будет прибавлена единица.

Можно рассматривать префиксные коды как пути на двоичном дереве: прохождение от узла к его левому сыну соответствует 0 в коде, а к правому сыну - 1. Если пометим листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева. Префиксное свойство гарантирует, что нет символов, которые были бы метками внутренних узлов дерева (не листьев), и наоборот, помечая кодируемыми символами только листья дерева, то обеспечивается префиксное свойство кода этих символов.

Для реализации алгоритма Хаффмана будем используем лес, т.е. совокупность деревьев, чьи листья будут помечены символами, для которых разрабатывается кодировка, а корни помечены суммой вероятностей всех символов, соответствующих листьям дерева. Будем называть эти суммарные вероятности весом дерева. Вначале каждому символу соответствует дерево, состоящее из одного узла, в конце работы алгоритма получим одно дерево, все листья которого будут помечены кодируемыми) символами. В этом дереве путь от корня к любому листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует 0, а правый — 1.

Важным этапом в работе алгоритма является выбор из леса двух деревьев с наименьшими весами. Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих деревьев. При слиянии деревьев создается новый узел, который становится корнем объединенного дерева и который имеет в качестве левого и правого сыновей корни старых деревьев. Этот процесс продолжается до тех пор, пока не получится только одно дерево. Это дерево соответствует коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

Теперь опишем необходимые структуры данных. Во-первых, для представления двоичных деревьев будем использовать массив TREE (Дерево), состоящий из записей следующего типа:

```
record
left child: integer;
rightchild: integer
parent: integer
end
```

Указатели в поле parent (родитель) облегчают поиск путей от листа к корню при записи кода символов. Во-вторых, используется массив ALPHABE T (Алфавит), также состоящий из записей, которые имеют следующий тип:

```
record
symbol: char;
probability: real;
leaf: integer { курсор }
end
```

В этом массиве каждому символу (поле symbol), подлежащему кодированию, ставится в соответствие вероятность его появления (поле probability) и лист, меткой которого он является (поле leaf). В-третьих, для представления непосредственно деревьев необходим массив FOREST (Лес). Этот массив будет состоять из записей с полями height (вес) и root (корень) следующего типа:

```
record
weight : real;
root : integer;
end
```

Листинг программа построения дерева Хаффмана

```
while существует более одного дерева в лесу do
begin
i:= индекс дерева в FOREST с наименьшим весом;
j:= индекс дерева в FOREST со вторым наименьшим весом;
Создание нового узла с левым сыном FOREST[i].root и правым
сыном FORES T[j].root;
Замена в FOREST дерева i деревом, чьим корнем является ,
новый узел и чей вес равен
FOREST[i].weight + FORES T[j].weight;
Удаление дерева j из массива FOREST
end;
```

Для реализации строки (4) листинга, где увеличивается количество

используемых ячеек массива TREE, и строк (5) и (6), где уменьшается количество ячеек массива FOREST, будем использовать курсоры lasttree (последнее дерево) и lastnode (последний узел), указывающие соответственно на массив FOREST и массив TREE. Предполагается, что эти курсоры располагаются в первых ячейках соответствующих массивов. Предположим, что все массивы имеют определенную объявленную длину, но здесь не будем проводить сравнение этих ограничивающих значений со значениями курсоров.

В листинге приведены коды двух полезных процедур. Первая из них, lightones, выполняет реализацию строк (2) и (3) листинга по выбору индексов двух деревьев с наименьшими весами. Вторая процедура, функция create(n1, n2), создает новый узел и делает заданные узлы n1 и n2 левым и правым сыновьями этого узла.

Листинг. Две процедуры

```
procedure lightones (var least, second : integer);
{присваивает переменным least и second индексы массива FOREST,
соответствующие деревьям с наименьшими весами. Предполагается, что
lasttree ≥ 2. }

var
i : integer;
begin (инициализация least и second,
рассматриваются первые два дерева }
if FOREST[1].weight <= FOREST[2].weight then
begin least:= 1; second:= 2 end
else
begin least:= 2; second:= 1 end
for i:= 3 to lasttree do
if FOREST[i].weight < FOREST[least].weight then
begin second:= least; least:= i end
else if FOREST[i] .weight < FOREST[second].weight then
second:= i end; { lightones }

function create ( lefttree, righttree: integer ): integer;
{ возвращает новый узел, у которого левым и правым сыновьями
становятся FOREST[lefttree] .root и FOREST[righttree].root}
begin
lastnode:= lastnode + 1;
{ ячейка TREE[lastnode] для нового узла }
TREE[lastnode] .leftchild:= FOREST[lefttree] .root;
TREE[lastnode].rightchild:= FOREST[righttree] .root;
```

```
{ теперь введем указатели для нового узла и его сыновей}
TREE[lastnode].parent:= 0;
TREE[FOREST[lefttree].root].parent:= lastnode;
TREE[FOREST'[righttree] .root].parent:= lastnode;
return(lastnode)
end; { create }
```

Теперь все неформальные операторы листинга можно описать подробнее. В листинге приведен код процедуры Huffman, которая не осуществляет ввод и вывод, а работает со структурами, показанными на рис. 26, которые объявлены как глобальные.

```
procedure Huffman;
var
i, j:integer; (два дерева с наименьшими весами из FOREST}
newroot: integer;
begin
while lasttree > 1 do begin
lightones(i, j);
newroot := create(i, j);
{ далее дерево i. заменяется деревом с корнем newroot}
FOREST[i].weight:=FOREST[i].weight + FOREST[j].weight;
FOREST[i].root:= newroot;
{ далее дерево j заменяется на дерево lasttree, массив FOREST
уменьшается на одну запись }
FOREST[j]:= FOREST[lasttree];
Lasttree:= lasttree -;
end
end; {Huffman}
```

После завершения работы алгоритма код каждого символа можно определить следующим образом. Найдем в массиве ALPHABET запись с нужным символ в поле symbol. Затем по значению поля leaf этой же записи определим местоположение записи в массиве TREE, которая соответствует листу/помеченному рассматриваемым символом. Далее последовательно переходим по указателю parent от текущей записи, например соответствующей узлу л, к записи в массиве TREE, соответствующей его родителю р. По родителю р определяем, в каком его поле, leftchild или rightchild, находится указатель на узел л, т.е. является ли узел п левым или правым сыном, и в соответствии с этим печатаем 0 (для левого сына) или 1 (для правого сына). Затем переходим к родителю узла р и определяем, является ли его сын р правым или левым, и в соответствии с эти печатаем следующую 1 или 0, и т. д. до самого корня дерева. Таким образом, код

символа будет напечатан в виде последовательности битов, но в обратном порядке. Чтобы распечатать эту последовательность в прямом порядке, надо каждый очередной бит помещать в стек, а затем распечатать содержимое стека в обычном порядке.

3.5. Реализация двоичных деревьев с помощью указателей

Для указания на правых и левых сыновей (и родителей, если необходимо) вместо курсоров можно использовать настоящие указатели языка Pascal. Например, можно сделать объявление

```
type
node = record
leftchild: ↑ node;
rightchlld: ↑ node;
parent: ↑ node
end
```

Используя этот тип данных узлов двоичного дерева, функцию create (листинг) можно переписать так, как показано в следующем листинге

Листинг. Код функции create при реализации с помощью указателей.

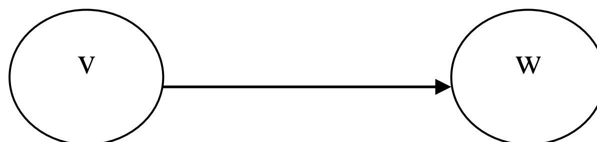
```
Function create (lefttree, righttree : ↑ node) : ↑ node;
var
root : ↑ node;
begin
new(root);
root↑.leftchild:=lefttree;
root↑.rightchild:= righttree;
root↑.parent:= 0;
lefttree↑.parent: = root;
righttree↑.parent:= root;
return(root)
end; { create }
```

4. ОРИЕНТИРОВАННЫЕ ГРАФЫ. ПРИКЛАДНЫЕ АЛГОРИТМЫ

Во многих задачах, встречающихся в компьютерных науках, математических дисциплинах часто возникает необходимость наглядного представления между какими-либо объектами. Ориентированные и неориентированные — естественная модель для таких отношений. В этой главе рассмотрены основные структуры данных, которые применяются для представление ориентированных графов, а также описаны некоторые основные алгоритмы определения связности ориентированных графов и нахождения кратчайших путей.

4.1. Основные определения

Ориентированный граф (или сокращенно орграф) G = (V, E) состоит из множества вершин V и множества дуг E. Вершины также называют узлами, а дуги — ориентированными ребрами. Дуга представима в виде упорядоченной пары вершин (v, w) где вершина v называется началом, а w



— концом дуги. Дугу (v, w) часто записывают как v→w и изображают в виде

Говорят также, что дуга v→w ведет от вершины v к вершине w, а вершина w смежная с вершиной v.

Пример. На рис. 27 показан орграф с четырьмя вершинами и пятью дугами.

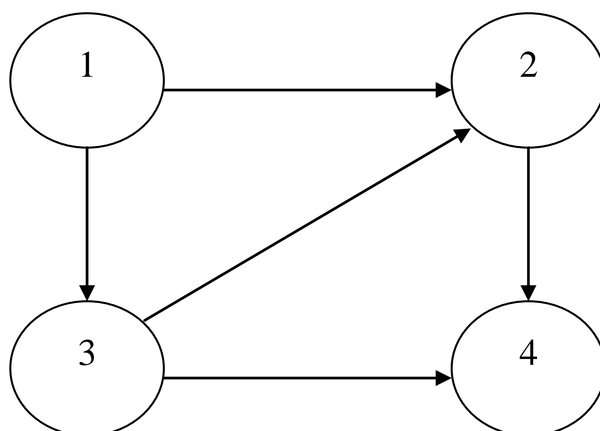


Рисунок 27 – Ориентированный граф.

Вершины орграфа можно использовать для представления объектов,

а дуги — для отношений между объектами. Например, вершины орграфа могут представлять города, а дуги — маршруты рейсовых полетов самолетов из одного города в другой. В виде орграфа может быть представлена блок-схема потока данных в компьютерной программе. В последнем примере вершины соответствуют блокам операторов программы, а дугам — направленное перемещение потоков данных.

Путем в орграфе называется последовательность вершин v1, v2, …, vn, для котороц существуют дуги v1→ v2, v2→v3, …, vn-1→vn. Этот путь начинается в вершине, проходя через вершины v2, v3, vn-1, заканчивается в вершине vn. Длина пути — количество дуг, составляющих путь, в данном случае длина пути равна n-1. Как особый случай пути рассмотрим одну вершину v как путь длины 0 от вершины v к этой же вершине v.

Путь называется простым, если все вершины в нем, за исключением, может быть, первой и последней, различны. Цикл — это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине.

Во многих приложениях удобно к вершинам и дугам графа присоединить какую-либо информацию. Для этих целей используются помеченных орграф, т.е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

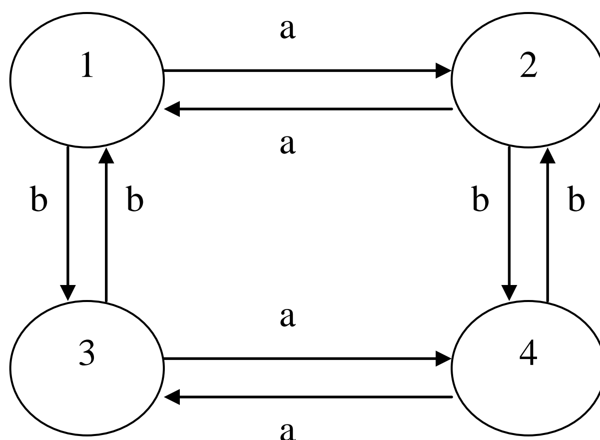


Рисунок 28 – Помеченный граф.

В помеченном орграфе вершина может иметь как имя, так и метку. Часто метки вершин будем использовать в качестве имен вершин. Так, на рис. 28 числа, помещенные в вершины, можно интерпретировать как имена вершин и как метки вершин.

4.2. Представления ориентированных графов

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Одним из наиболее общих представлений орграфа G = (V, E) является матрица смежности. Предположим, что множество вершин V = {1, 2, …, n}. Матрица смежности для орграфа G — это матрица A размера n×n со значениями булевого типа, где A[i, j] == true тогда и только тогда, когда существует дуга из вершины i в вершину j. Часто в матрицах смежности значение true заменяется на 1, а значение false — на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в котерых надо часто проверять существование данной дуги.

Обобщением описанного представления орграфа, с помощью матрицы смежности можно считать представление помеченного орграфа также посредством матрицы смежности, но у которой элемент A[i, j] равен метке дуги i→j. Если дуги от вершины i к вершине j не существует, то значение A[i, j] не может быть значением какой-либо допустимой метки, а может рассматриваться как "пустая" ячейка.

Пример. На рис. 29 показана матрица смежности для помеченного орграфе из рис. 29. Здесь дуги представлены меткой символьного типа, а пробел используется при отсутствии дуг.

	1	2	3	4
1		a		b
2	a		b	
3		b		a
4	b		a	

Рисунок 29 – Матрица смежности для помеченного графа из рис.30.

Основной недостаток матриц смежности заключается в том, что она требует Ω (n^2) объема памяти, даже если дуг значительно меньше, чем n^2. Поэтому для чтения матрицы или нахождения в ней необходимого элемента требуется время порядка O(n^2), что не позволяет создавать алгоритмы с временем O(n) для работы с орграфами, имеющими порядка O(n) дуг.

Поэтому вместо матриц смежности часто используется другое представление для орграфа G == (V, E), называемое представлением посредством списков смежности. Списком смежности для вершины i называется список всех вершин, смежных с вершиной i, причем определенным образом упорядоченный. Таким образом, орграф G можно

представить посредством массива HEAD (Заголовок), чей элемент HEAD [i] является указателем на список смежности вершины i. Представление орграфа с помощью списков смежности требует для хранения объем памяти, пропорциональны сумме количества вершин и количества дуг. Если количество дуг имеет порядок O(n), то и общий объем необходимой памяти имеет такой же порядок. Но и для списков смежности время поиска определенной дуги может иметь порядок O(n), так как такой же порядок может иметь количество дуг у определенной вершины.

Пример. На рис.30 показана структура данных, представляющая орграф из рис.30 посредством связанных списков смежности. Если дуги имеют метки, то их можно хранить в ячейках связанных списков.

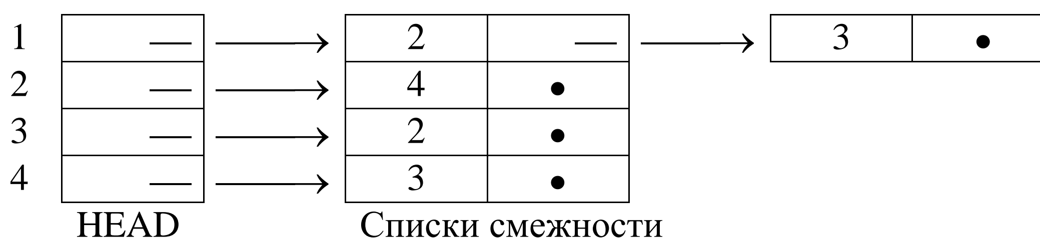


Рисунок 30 – Структура списков смежности для оргарафа графа из рис.30.

Для вставки и удаления элементов в списках смежности необходимо иметь массив HEAD, содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины. Но если известно, что граф не будет подвергаться изменениям (или они будут незначительны), то предпочтительно сделать массив HEAD массивом курсоров на массив ADJ (от adjacency — смежность), где ячейки ADJ[HEAD[i]], ADJ[HEAD[i] + 1] и т.д. содержат вершины, смежные с вершиной i, и эта последовательность смежных вершин заканчивается первым встреченным нулем в массиве ADJ. Пример такого представления для графа из рис 30 показан на рис. 31.

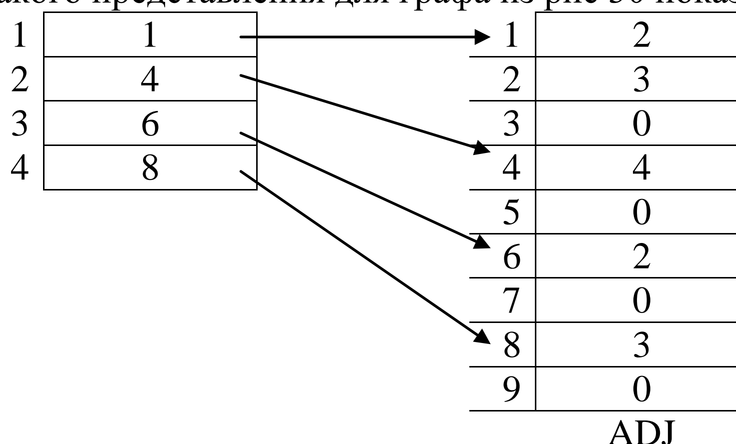


Рисунок 31 – Другое представление списков смежности для оргарафа графа из рис.30.

АТД для ориентированных графов

По привычной схеме сначала надо формально определить абстрактные типы данных, соответствующие ориентированным графам, а затем рассмотреть операторы, выполняемые над этими АТД. Не будем неукоснительно следовать этой схеме, так как на этом пути нам не встретятся большие неожиданности, а принципиальные структуры данных, используемые для представления орграфов, уже были описаны ранее. Наиболее общие операторы, выполняемые над ориентированными графами, включают операторы чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператор перемещения по последовательностям дуг.

Последний оператор требует небольших пояснений. Часто в программах встречаются операторы, которые неформально можно описать подобно следующему оператору:

```
for каждая вершина w, смежная с вершиной v do
{ некоторые действия с вершиной w }
```

Для реализации такого оператора необходим индексный тип данных для работы с множеством вершин, смежных с заданной вершиной и. Например, если для представления орграфа используются списки смежности, то индекс — это позиция в списке смежности вершины и. Если применяется матрица смежности, тогда индекс — целое число, соответствующее смежной вершине. Для просмотра множества смежных вершин необходимы следующие три оператора.

1. FIRST(v) возвращает индекс первой вершины, смежной с вершиной и. Если вершина v не имеет смежных вершин, то возвращается "нулевая" вершина Λ.

2. NEXT(v, i) возвращает индекс вершины, смежной с вершиной и, следующий за индексом i. Если i — это индекс последней вершины, смежной с вершиной v возвращается Λ.

3. VERTEX(v, i) возвращает вершину с индексом i из множества вершин, смежных с v.

Пример 6.5. Если для представления орграфа используется матрица смежности, то функция VERTEX(v, i) просто возвращает индекс i. Реализация функций FIRST(v) и NEXT(v, i) представлена в листинге. В этом листинге матрица смежности А размера $n \times n$ определена вне этих функций с помощью следующего объявления:

```
array [1..n, l..n] of boolean
```

В этой матрице 0 обозначает отсутствие дуги. Эти функции позволяют реализовать оператор () так, как показано в листинге. Отметим,

что функцию FIRST(v) можно реализовать как NEXT (v, 0).

Листинг. Операторы просмотра множества смежных вершин

```
function FIRST (v : integer) : integer;
var
i : integer;
begin
for i: = 1 to n do
if A[v, i] then
return(i);
return (0) { вершина v не имеет смежных вершин)
end; { FIRST }
function NEXT ( v: integer; i: integer): integer;
var
j : integer;
begin
for j := i + 1 to n do
if A[v, j] then
return(j);
return(0)
end; { NEXT }
```

Листинг: Последовательный просмотр вершин/ смежных с вершиной v

```
i:= FIRST(v) ;
while i <> 0 do begin
w := VERTEX(v, i);
{ действия, с вершиной w }
i:= NEXT(v, i)
end
```

4.3. Задача нахождения кратчайшего пути

В этом разделе рассмотрим задачи нахождения путей в ориентированном графе. Пусть есть ориентированный граф G = (V, E), у которого все дуги имеют неотрицательные метки (стоимости дуг), а одна вершина определена как источник. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам графа G (здесь длина пути определяется как сумма стоимостей дуг, составляющих путь). Эта задача часто называется задачей нахождения кратчайшего пути с одним источником. Может показаться, что более

естественной задачей будет нахождение кратчайшего пути от Уточника к определенной вершине назначения. Но эта задача в общем случае имеет такой же уровень сложности, что и задача нахождения кратчайшего пути для всех вершин графа (за исключением того "счастливого" случая, когда путь к вершине назначения будет найден ранее, чем просмотрены пути ко всем вершинам графа).

Отметим, что будем говорить о длине пути даже тогда, когда она измеряется в других, не линейных, единицах измерения, например во временных единицах.

Можно представить орграф G в виде карты маршрутов рейсовых полетов из одного города в другой, где каждая вершина соответствует городу, а дуга v→w — рейсовому маршруту из города и в город w. Метка дуги v→w — это время полета из города v в город w. В этом случае решение задачи нахождения кратчайшего пути с одним источником для ориентированного графа трактуется как минимальное время перелетов между различными городами.

Для решения поставленной задачи будем использовать "жадный" алгоритм, который часто называют алгоритмом Дейкстры (Dijkstra). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S. Назовем такой путь особым. На каждом шаге алгоритма используется также массив D, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, т.е. для всех вершин будут найдены "особые" пути, тогда массив D будет содержать длины кратчайших путей от источника к каждой вершине.

Алгоритм Дейкстры представлен в листинге. Здесь предполагается, что в орграфе G вершины поименованы целыми числами, т.е. множество вершин V= {1, 2, …,n}, причем вершина 1 является источником. Массив C — это двумерный массив стоимостей, где элемент C[i, j] равен стоимости дуги i→j. Если дуги i→j не существует, C[i, j] ложится равным ∞, т.е. большим любой фактической стоимости дуг. На каждом шаге D[i] содержит длину текущего кратчайшего особого пути к вершине i.

Листинг. Алгоритм Дейкстры

```
procedure Dijkstra;
begin
(1)     s:= {i};
(2)     for 2:= 2 to n do
```

```
(3)       D[i]:= C[l, i]; { инициализация D }
(4)    for i:= 1 to n - 1 do begin
(5)       выбор из множества V\S такой вершины w,
что значение D[w] минимально;
(6)       добавить w к множеству S;
(7)       for каждая вершина v из множества V\S do
(8)         D[v] := min(D[v] , D[w] + C[w, v])
end
end; { Dijkstra }
```

Пример. Применим алгоритм Дейкстры для ориентированного графа, показанного на рис. 25. Вначале S = {1}, D[2] = 10, D[3] == ∞, D[4] == 30 и D[6] = 100. На первом шаге цикла (строки (4) - (8) листинга) w = 2, т.е. вершина 2 имеет минимальное значение в массиве 2). Затем вычисляем D[3] == min(∞, 10 + 50) = 60. D[4] и D[5] не изменяются, так как не существует дуг, исходящих из вершины 2 и ведущих к вершинам 4 и 5. Последовательность значений элементов массива D после каждой итерации цикла показаны в таблице.

Несложно внести изменения в алгоритм так, чтобы можно было определить сам кратчайший путь (т.е. последовательность вершин) для любой вершины. Для этого надо ввести еще один массив P вершин, где P[v] содержит вершину, непосредственно предшествующую вершине и в кратчайшем пути. Вначале положим P[v] =1 для всех v≠1. В листинге после строки (8) надо записать условный оператор с условием D[w]+C[w,v]<D[v], при выполнении которого элементу P[v] присваивается значение w. После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива P.

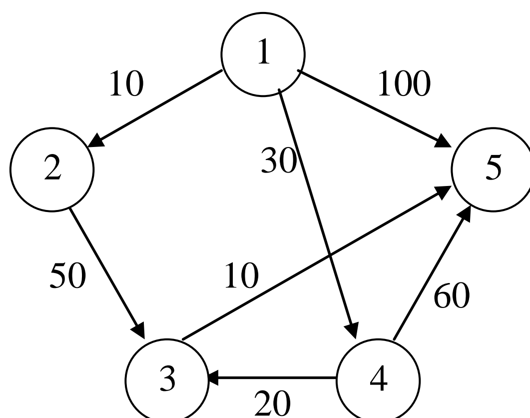


Рисунок 32 – Орграф с помеченными дугами.

Таблица Вычисления по алгоритму Дейкстры для орграфа из рис. 25.

Итерация	S	w	D[2]	D[3]	D[4]	D[5]
Начало	{1}	-	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Пример. Для орграфа из предыдущего примера массив Р имеет следующие значения Р[2] = 1, Р[3] = 4, Р[4] = 1, Р[5] = 3. Для определения кратчайшего пути, например, от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие вершины, начиная с вершины 5. На основании значений массива Р определяем, что вершине 5 предшествует вершина 3, вершине 3 — вершина 4, а ей, в свою очередь, — вершина 1. Таким образом, кратчайший путь из вершины 1 в вершину 5 составляет следующая последовательность вершин: 1, 4, 3, 5.

Обоснование алгоритма Дейкстры

Алгоритм Дейкстры — пример алгоритма, где "жадность" окупается в том смысле, что если что-то "хорошо" локально, то оно будет "хорошо" и глобально. В данном случае что-то локально "хорошее" — это вычисленное расстояние от источника к вершине w, которая пока не входит в множество S, но имеет кратчайший особый путь. (Напомним, что особым назвали путь, который проходит только через вершины множества S.) Чтобы понять, почему не может быть другого кратчайшего, но особого, пути, рассмотрим рис. 33. Здесь показан гипотетический кратчайший путь к вершине w, который сначала проходит до вершины x через вершины множества S, затем после вершины x путь, возможно, несколько раз входит в множество S и выходит из него, пока не достигнет вершины w.

Но если этот путь короче кратчайшего особого пути к вершине w, то и начальный сегмент пути от источника к вершине x (который тоже является особым путем) также короче, чем особый путь к w. Но в таком случае в строке (5) листинга при выборе вершины w должны выбрать не эту вершину, а вершину x, поскольку D[x] меньше D[w]. Таким образом, пришли к противоречию, следовательно не может быть другого кратчайшего пути к вершине w, кроме особого. (Отметим здесь определяющую роль того факта, что все стоимости дуг неотрицательны, без этого свойства помеченного орграфа алгоритм Дейкстры не будет работать правильно.)

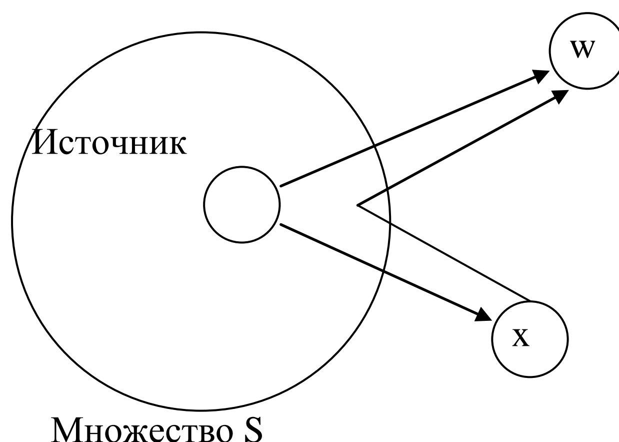


Рисунок 33 – Гипотетический кратчайший путь к вершине w.

Для завершения обоснования алгоритма Дейкстры надо еще доказать, что D[v] действительно показывает кратчайшее расстояние до вершины v. Рассмотрим до ление новой вершины w к множеству S (строка (6) листинга), после чего происходит пересчет элементов массива D в строках (7), (8) этого листинга; при этом может получиться более короткий особый путь к некоторой вершине v, проходящий через вершину w. Если часть этого пути до вершины w проходит через вершины предыдущего множества S и затем непосредственно к вершине v, то стоимость этого пути, D[w] + C[w, v], в строке (8) листинга сравнивается с D[v] и, если новый путь короче, изменяется значение D[v].

Существует еще одна гипотетическая возможность кратчайшего особого пути к вершине v, которая показана на рис. 33. Здесь этот путь сначала идет к вершине w, затем возвращается к вершине x, принадлежащей предыдущему множеству S, затем следует к вершине v. Но реально такого кратчайшего пути не может быть. Поскольку вершина x помещена в множество S раньше вершины w, то все кратчайшие пути от источника к вершине x проходят исключительно через вершины предыдущего множества S. Поэтому показанный на рис. 6.8 путь к вершине x, проходящий через вершину w, не короче, чем путь к вершине x, проходящий через вершины множества S. В результате и весь путь к вершине и, проходящий через вершины x и w, не короче, чем путь от источника к вершине x, проходящий через вершины множества S, и далее непосредственно к вершине v. Таким образом, доказано, что оператор в строке (8) листинга действительно вычисляет длину кратчайшего пути.

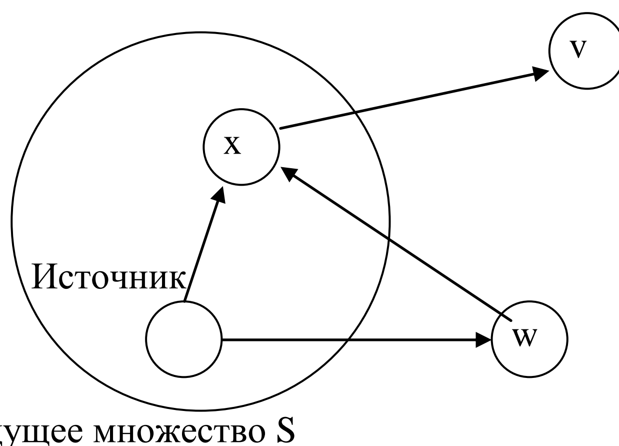


Рисунок 34 – Реально невозможный кратчайший особый путь.

Время выполнения алгоритма Дейкстры

Предположим, что процедура листинга оперирует с орграфом, имеющим n вершин и e дуг. Если для представления графа используется матрица смежности, то для выполнения внутреннего цикла строк (7) и (8) потребуется время O(n), а для всех n-1 итераций цикла строки (4) потребуется время порядка$O(n^2)$. Время, необходимое для оставшейся части алгоритма, как легко видеть, не превышает этот же порядок.

Если количество дуг e значительно меньше, n^2, то лучшим выбором для представленного орграфа будут списки смежности, а для множества вершин V \ S — очередь с приоритетами, реализованная в виде частично упорядоченного дерева. Тогда время выбора очередной вершины из множества V \ S и пересчет стоимости путей для одной дуги составит O(log n), а общее время выполнения цикла строк (7) и (8) — O(e log n), а не $O(n^2)$.

Строки (1) - (3) выполняются за время порядка O(n). При использовании очередей с приоритетом для представления множества V \ S строка (5) реализуется посредством оператора DELETEMIN, а каждая из n - 1 итераций цикла (4) - (6) требует времени порядка O(logn).

В результате получаем, что общее время выполнения алгоритма Дейкстрф ограничено величиной порядка O(e logn). Это время выполнения значительно меньше, чем $O(n^2)$, когда e существенно меньше n^2.

4.4. Нахождение кратчайших путей между парами вершин

Предположим, что имеем помеченный орграф, который содержит время полета по маршрутам, связывающим определенные города, и хотим построить таблицу, где приводилось бы минимальное время перелета из одного (произвольного) города в любой другой. В этом случае сталкиваемся с общей задачей нахождения кратчайших путей, т.е.

нахождения кратчайших путей между всеми парами вершин орграфа. Более строгая формулировка этой задачи следующая: есть ориентированный граф G = (V, E), каждой дуге $v \rightarrow w$ этого графа сопоставлена неотрицательная стоимость C[v, w]. Общая задача нахождения кратчайших путей заключается в вхождении для каждой упорядоченной пары вершин (v, w) любого пути от вершины v в вершины w, длина которого минимальна среди всех возможных путей от v к w.

Можно решить эту задачу, последовательно применяя алгоритм Дейкстры к каждой вершины, объявляемой в качестве источника. Но существует прямой способ решения данной задачи, использующий алгоритм Флойда (R. W. Floyd). Для определенности положим, что вершины графа последовательно пронумерованы от 1 до n. Алгоритм Флойда использует матрицу А размера $n \times n$, в которой вычисляются длины кратчайших путей. Вначале A[i, j] = C[i, j] для всех $i \neq j$. Если дуга $i \rightarrow j$ отсутствует, то C[i, j] == ∞. Каждый диагональный элемент матрицы А равен 0.

Над матрицей А выполняется n итераций. После k-й итерации A[i, j] содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k.

На k-й итерации для вычисления матрицы А применяется следующая формула:

Ak[i, j] = min(Ak-1[i, j], Ak-1[i, k] + Ak-1[k,j]).

Нижний индекс k обозначает значение матрицы А после k-й итерации, но не означает, что существует n различных матриц, этот индекс используется для сокращения записи. Графическая интерпретация этой формулы показана на рис. 35

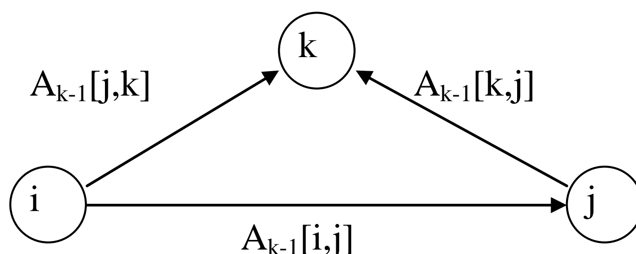


Рисунок 35 – Помеченный ориентированный граф в путь от вершины i к вершине j.

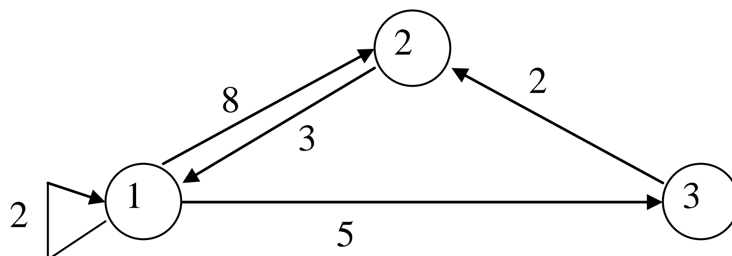


Рисунок 36 – Помеченный ориентированный граф.

Для вычисления Ak[i, j] проводится сравнение величины Ал-1[i, j] (т.е. стоимость пути от вершины i к вершине j без участия вершины k или другой вершины с более высоким номером) с величиной Ak-1[i,k]+Ak-1[k, j] (стоимость пути от вершины i до вершины k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k дешевле, чем Ak-1[i,j] величина Ak[i,j] изменяется.

Пример. На рис.36 показан помеченный орграф, а на рис. 37 - значения матрицы А после трех итераций.

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i,j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i,j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i,j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

$A_3[i,j]$

Рисунок 37 – Последовательные значения матрицы А.

Равенства Ak[i, k] = Ak-1[i, k] и Ak[k, j] = Ak-1[k,j] означают, что на k-й итерации элементы матрицы А, стоящие в k-й строке и k-м столбце, не изменяются. Более того, все вычисления можно выполнить с применением только одной копии матрицы А. Процедура, реализующая алгоритм Флойда, представлена в следующем листинге.

Листинг. Реализация алгоритма Флойда

```
procedure Floyd ( var A: array[l..n, 1..n] of real;
C: array[l..n, 1..n] of real);
var
i, j, k: integer;
begin
for i:= 1 to n do
for j:= 1 to n do
A[i, j]:= C[i, j];
for i: = 1 to n do
A[i, i]:= 0;
for k:= 1 to n do
for i:= 1 to n do
for j: = 1 to n do
if A[i, k] + A[k, j] < A[i, j] then
A[i, j]:= A[i, k] + A[k, j]
end; { Floyd }
```

Время выполнения этой программы, очевидно, имеет порядок $O(n^3)$, поскольку в ней практически нет ничего, кроме вложенных друг в друга трех циклов. Доказательство "правильности" работы этого алгоритма также очевидно и выполняется с помощью математической индукции по k, показывая, что на k-й итерации вершина k включается в путь только тогда, когда новый путь короче старого.

Сравнение алгоритмов Флойда и Дейкстры.

Поскольку версия алгоритма Дейкстры с использованием матрицы смежности находит кратчайшие пути от одной вершины за время порядка $O(n^2)$, то в этом случае применение алгоритма Дейкстры для нахождения всех кратчайших путей потребует времени порядка $O(n^3)$, т.е. получим такой же временной порядок, как и в алгоритме Флойда. Константы пропорциональности в порядках времени выполнения для обоих алгоритмов зависят от применяемых компилятора и вычислительной машины, а также от особенностей реализации алгоритмов. Вычислительный эксперимент и измерение времени выполнения — самый простой путь подобрать лучший алгоритм для конкретного приложения.

Если e, количество дуг в орграфе, значительно меньше, чем n^2, тогда, несмотря на относительно малую константу в выражении порядка $O(n^3)$ для алгоритма Флойда, рекомендуется применять версию алгоритма Дейкстры со списками смежности. В этом случае время решения общей задачи нахождения кратчайших путей имеет порядок $O(ne \log n)$, что значительно лучше алгоритма Флойда, по крайней для больших

разреженных графов.

Вывод на печать кратчайших путей

Во многих ситуациях требуется распечатать самый дешевый путь от одной вершины к другой. Чтобы восстановить при необходимости кратчайшие пути, можно в алгоритме Флойда ввести еще одну матрицу P, в которой элемент P[i, j] содержит вершину k, полученную при нахождении наименьшего значения A[i, j]. Если P[i, j] = 0, то кратчайший путь из вершины i в вершину j состоит из одной дуги i→j. Модифицированная версия алгоритма Флойда, позволяющая восстанавливать кратчайшие пути, представлена в листинге.

Листинг. Программа нахождения кратчайших путей

```
procedure Floyd. ( var A: array[l..n, l..n] of real;
C:array[l..n, l..n] of real; P:array[l..n, 1..n] of integer;
var
i, j, k: integer;
begin
for i:= 1 to n do
for j:= 1 to n do begin
A[i, j]:= C[l, j] ;
P[2, j] := 0 end ;
for i:= 1 to n do
A[i, 1]:= 0;
for k:= 1 to n do
for i:= 1 to n do
for j:= 1 to n do
if A[i, k] + A[k, j] < A[i, j] then begin
A[i, j]:= A[i, k] + A[k, j];
P[i, j] := k
end
end; { Floyd }
```

Для вывода на печать последовательности вершин, составляющих кратчайший путь от вершины i до вершины j, вызывается процедура path(i,j), код которой приведен в листинге.

Листинг. Процедура печати кратчайшего пути

```
procedure path (i,j : integer);
var
```

```
k : integer;
begin
k:= P[i, j];
if k= 0 then
return
path(i, k);
writeln(k);
path(k,j)
end; { path }
```

Пример. На рис. 38 показана результирующая матрица P для орграфа из рис. 36.

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

P

Рисунок 38 – Матрица P для орграфа из рис. 37.

Транзитивное замыкание

Во многих задачах интерес представляет только сам факт существования пути длиной не меньше единицы, от вершины i до вершины j. Алгоритм Флойда можно приспособить для решения таких задач. Но полученный в результате алгоритм еще до Флойда разработал Уоршелл (S. Warshall), поэтому будем называть его алгоритмом Уоршелла.

Предположим, что матрица стоимостей C совпадает с матрицей смежности для данного орграфа G, т.е. C[i, j] == 1 только в том случае, если есть дуга $i \rightarrow j$, и C[i, j] = 0, если такой дуги не существует. Предположим, что хотим вычислить матрицу A такую, что A[i, j] = 1 тогда и только тогда, когда существует путь от вершины i до вершины j длиной не менее 1 и A[i, j] == 0 — в противном случае. Такую матрицу A частот называют транзитивным замыканием матрицы смежности.

Пример. На рис. 39 показано транзитивное замыкание матрицы смежности орграфа из рис. 38.

	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1

P

Рисунок 39 – Транзитивное замыкание матрицы смежности.

Транзитивное замыкание можно вычислить с помощью процедуры, подобной Floyd, применяя на k-м шаге следующую формулу к булевой матрице A:

Ak[i,j] = Ak-1[i, j] or (Ak-1[i, k] and Ak-1[k, j]).

Эта формула устанавливает, что существует путь от вершины i до вершины j, проходящий через вершины с номерами, не превышающими k, только в следующих случаях.

1. Уже существует путь от вершины i до вершины j, который проходит через вершины с номерами, превышающими k-1.
2. Существует путь от вершины i до вершины k, проходящий черех вершины м номерами, не превышающими k-1, и путь от вершины k до вершины j, который также проходит через вершины с номерами, не превышающими k-1/

Здесь, как и в алгоритме Флойда, Ak[i, k] = Ak-1[i, k] и Ak[k, j] = Ak-i[k,j], и вычисления можно выполнять в одной копии матрицы А. Программа Warshall транзитивного замыкания показана в листинге.

Листинг. Программа Warshall для вычисления транзитивного замыкания

```
procedure Warshall ( var A: array[l..n, l..n] of boolean;
C: array[l..n, l..n] of boolean );
var
i, j, k: integer;
begin
for i: = 1 to n do
for j:= 1 to n do
A[i, j]:= C[i, j];
for k:= 1 to n do
for i.:= 1 to n do
for j:= 1 to n do
if A[i, j] = false then
A[i,j]:= A[i, k] and A[k, j]
end; { Warshall }
```

5. СОРТИРОВКА. ПРИКЛАДНЫЕ АЛГОРИТМЫ

Сортировкой, или упорядочиванием списка объектов называется расположение этих объектов по возрастанию или убыванию согласно определенному линейному отношению порядка, такому как отношение "≤" для чисел. Это очень большая тема, поэтов она разбита на две части: внутреннюю и внешнюю сортировку. При внутренней сортировке все сортируемые данные помещаются в оперативную память компьютера, где можно получить доступ к данным в любом порядке (т.е. используется модель памяти с произвольным доступом). Внешняя сортировка применяется тогда, когда объем упорядочиваемых данных слишком большой, чтобы все данные можно было поместить в оперативную память. Здесь узким местом является механизм перемещения больших блоков данных от устройств Внешнего хранения данных к оперативной памяти компьютера и обратно. Тот факт, что физически непрерывные данные надо для удобного перемещения организовывать в блочную структуру, заставляет нас применять разное разные методы внешней сортировки.

Сортировку можно проводить только в случае, если введено отношение порядка "<" на множестве ключей вводится таким образом, чтобы для любых значений a, b, c выполнялись следующие условия:

1. справедливо одно из соотношений: $a < b$, $a = b$, $a > b$ (закон трихотомии);
2. если $a < b$ и $b < c$, то $a < c$ (закон транзитивности).

Эти два свойства определяют математическое понятие линейной упорядоченности. Каждое множество с отношением < , удовлетворяющее свойствам 1, 2, поддаются сортировке. Задача сортировки – найти такую перестановку записей p(1), p(2), … , p(n) после которой ключи расположились бы в неубывающем порядке $K_{p(1)} \le K_{p(2)} \le \ldots \le K_{p(n)}$.

Сортировка называется устойчивой, если записи с одинаковыми ключами остаются в прежнем порядке

$$p(i) < p(j), \quad \text{если } K_{p(i)} = K_{p(j)},\ i < j$$

5.1. Модель внутренней сортировки

Представим основные принципиальные алгоритмы внутренней| сортировки. Простейшие из этих алгоритмов затрачивают время порядка $O(n^2)$ для упорядочивания n объектов и потому применимы только к небольшим множествам объектов. Один из наиболее популярных алгоритмов сортировки, так называемая быстрая сортировка, выполняется в среднем за время O(n logn). Быстрая сортировка хорошо работает в большинстве приложений, хотя в самом худшем случае она также имеет время выполнения $O(n^2)$. Существуют другие методы сортировки, такие как пирамидальная сортировка или сортировка слиянием, которые в самом

худшем случае дают время порядка O(n logn), но в среднем (в статистическом смысле) работают не лучше, чем быстрая сортировка. Отметим, что метод сортировки слиянием хорошо подходит для построения алгоритмов внешней сортировки. Также рассмотрим алгоритмы другого типа, имеющие общее название "карманной" сортировки. Эти алгоритмы работают только с данными определенного типа, например с целыми числами из ограниченного интервала, но когда их можно применить, то они работают очень быстро, затрачивая время порядка O(n) в самом худшем случае.

Всюду в этой главе будем предполагать, что сортируемые объекты являются записями, содержащими одно или несколько полей. Одно из полей, называемое ключом, имеет такой тип данных, что на нем определено отношение линейного порядок "≤". Целые и действительные числа, символьные массивы — вот общие примеры таких типов данных, но, конечно, можно использовать ключи других типов данных, лишь бы на них можно было определить отношение "меньше чем" или "меньше чем или равно".

Задача сортировки состоит в упорядочивании последовательности записей таким образом, чтобы значения ключевого поля составляли неубывающую последовательность. Другими словами, записи r_1, r_2, ..., r_n со значениями ключей k_1, k_2, ..., k_n надо расположить в порядке r_{i1}, r_{i2}, ..., r_{in}, таком, что k_{i1}, k_{i2}, ..., k_{in}. Не требуется, чтобы все записи были различными, и если есть записи с одинаковыми значениями ключей, то в упорядоченной последовательности они располагаются рядом друг с другом в любом порядке.

Будем использовать различные критерии оценки времени выполнения алгоритмов внутренней сортировки. Первой и наиболее общей мерой времени выполнения является количество шагов алгоритма, необходимых для упорядочивания n записей. Эта мера особенно информативна, когда ключи являются строками символов, и поэтому самым "трудоемким" оператором будет оператор сравнения ключей. Если размер записей большой, то следует также учитывать время, необходимое для перемещения записей. При создании конкретных приложений обычно ясно, по каким критериям нужно оценивать применяемый алгоритм сортировки.

5.2. Простые схемы сортировки

По-видимому, самым простым методом сортировки является так называемый метод "пузырька". Чтобы описать основную идею этого метода, представим, что записи, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Записи с малыми значениями ключевого поля более "легкие" и "всплывают" вверх наподобие пузырька. При первом проходе вдоль массива, начиная проход снизу, берется первая

запись массива и ее ключ поочередно сравнивается с ключами последующих записей. Если встречается запись с более "тяжелым" ключом, то эти записи меняются местами. При встрече с записью с более "легким" ключом эта запись становится "эталоном" для сравнения, и все последующие записи сравниваются с этим новым, более "легким" ключом. В результате запись с наименьшим значением ключа окажется в самом верху массива. Во время второго прохода вдоль массива находится запись со вторым по величине ключом, которая помещается под записью, найденной при первом проходе массива, т.е. на вторую сверху позицию, и т.д. Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать записи, найденные за предыдущие проходы, так как они имеют ключи, меньшие, чем у оставшихся записей. Другими словами, во время i-го прохода не проверяются записи, стоящие на позициях выше i. В листинге приведен описываемый алгоритм, в котором через A обозначен массив из n записей (тип данных cordtype). Здесь и далее в этой главе предполагаем, что одно из полей записей называется key (ключ) и содержит значения ключей.

Листинг. Алгоритм "пузырька"

```
for i:=l to n-l do
for j:= 1 downto i + 1 do
if A[j].key < A[j - 1].key then
swap(A[j], A[j - 1])
```

Процедура swap (перестановка) используется во многих алгоритмах сортировки для перестановки записей местами, ее код показан в следующем листинге.

Листинг процедура Swap

```
procedure swap { var x, y: recordtype )
{ swap меняет местами записи x и y }
var
temp: recordtype;
begin
temp:= x;
x:=y;
y:= temp;
end; { swap }
```

Сортировка вставками

Второй метод, который рассмотрим, называется сортировка

вставками, так как на i-м этапе "вставляем" i-й элемент A[i] в нужную позицию среди элементов A[1], A[2], …, A[i-1], которые уже упорядочены. Сказанное можно записать в виде следующей подпрограммы:

```
for i:=2 to n do
переместить A[1] на позицию j < i. такую, что
A[1] < A[k] для j < k < и
либо A[l] > A[j - 1], либо j = 1
```

Чтобы сделать процесс перемещения элемента A[i] более простым, полезно ввести элемент A[0], чье значение ключа будет меньше значения ключа любого элемента A[1], ..., A[n]. Можно постулировать существование константы - ∞ типа keytype, которая будет меньше значения ключа любой записи, встречающейся на практике. Если такую константу нельзя применить, то при вставке A[i] в позицию j - 1 надо проверить, не будет ли j = 1, если нет, тогда сравнивать элемент A[i] (который сейчас находится в позиции j) с элементом A[j - 1]. Описанный Показан в листинге.

Листинг. Сортировка вставками

```
A[0] .key:= - ∞;
for i:= 2 to n do begin
j:= i;
while A[j] < A[j - 1] do begin
swap(A[j], Alj - 1]) ;
j:= j - 1
end
end
```

Сортировка посредством выбора

Идея сортировки посредством выбора также элементарна, как и те два метода сортировки, которые уже рассмотрены. На i-м этапе сортировки выбирается запись с наименьшим ключом среди записей A[i], …, A[n] и меняется местами запись с записью A[i]. В результате после i-го этапа все записи A[1], …, A[i] будут упорядочены. Сортировку посредством выбора можно описать следующим образом:

for i:=1 to n-1 do

выбрать среди A[i], ..., A[n] элемент с наименьшим ключом и поменять его местами с A[i];

Более полный код, реализующий этот метод сортировки, приведен в листинге.

Листинг Сортировка посредством выбора.

```
var
lowkey: keytype; { текущий наименьший ключ, найденный при проходе по элементам A [i], ..., A[n] }
lowindex: integer; { позиция элемента с ключом lowkey }
begin
for i:= 1 to n - 1 do begin
lowindex:= 1;
lowkey:= A[i].key;
for j:= 2 + 1 to n do
{ сравнение ключей с текущим ключом lowkey }
if A[j].key < lowkey then begin
lowkey:= A[j] .key;
lowindex:= j
end;
swap(A[i], A[lowindex])
end
end;
```

Временная сложность методов сортировки

Методы "пузырька", вставками и посредством выбора имеют временную сложность $O(n^2)$ и $\Omega(n^2)$ на последовательностях из n элементов. Рассмотрим метод "пузырька". Независимо от того, что подразумевается под типом recordtype, выполнение процедуры swap требует фиксированного времени. Поэтому строки (3), (4) затрачивают c1 единиц времени; c2 — некоторая константа. Следовательно, для фиксированного значения i цикл строк (2)-(4) требует не больше c2(n-i) шагов: c2 — константа. Последняя константа несколько больше константы c1, если учитывать операции с индексом k в строке (2). Поэтому вся программа требует

$$c_3 n + \sum_{i=1}^{n-1} c_2 (n-1) = \frac{1}{2} c_2 n^2 + (c_3 - \frac{1}{2} c_2) n$$

шагов, где слагаемое $c_3 n$ учитывает операции с индексом i в строке (1). Последнее выражение не превосходит $(c_2/2 + c_3)n^2$ для $n \geq 1$, поэтому алгоритм "пузырька" имеет временною сложность $O(n^2)$. Нижняя временная граница для алгоритма равна $\Omega(n^2)$, поскольку если даже не выполнять процедуру swap (например, если список уже отсортирован), то все равно $n(n - 1)/2$ раз выполняется проверка в строке (3).

Далее рассмотрим сортировку вставками (листинг 8.3). Цикл while в строках (4)-(6) выполняется не более O(i) раз, поскольку начальное значение j равно i, а затем j уменьшается на 1 при каждом выполнении

этого цикла. Следовательно, цикл for строк (2) - (6) потребует не более $c\sum_{i=2}^{n} i$ шагов для некоторой константы c. Эта сумма имеет порядок O(n^2).

Можно проверить, что если список записей первоначально был отсортирован в обратном порядке, то цикл while в строках (4) - (6) выполняется ровно i-1 раз, поэтому строка (4) выполняется $\sum_{i=2}^{n}(i-1) = n(n-1)/2$ раз. Следовательно, сортировка вставками в самом худшем случае требует времени не менее Ω(n^2). Можно показать, что нижняя граница в среднем будет такой же.

Наконец, рассмотрим сортировку посредством выбора, показанную в листинге 8.4. Легко проверить, что внутренний цикл в строках (4) - (7) требует времени порядка O(n - i), поскольку j здесь изменяется от i + 1 до n. Поэтому общее время выполнения алгоритма составляет $c\sum_{i=1}^{n-1}(n-1)$ для некоторой константы c. Эта сумма, равная cn(n - 1)/2, имеет порядок роста O(n^2). С другой стороны, нетрудно показать, что строка (4) выполняется не менее $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1) = n(n-1)/2$ раз независимо от начального списка сортируемых элементов. Поэтому сортировка посредством выбора требует времени не менее Ω(n^2) в худшем случае и в среднем.

Подсчет перестановок

Если размер записей большой, тогда процедура swap для перестановки записей, которая присутствует во всех трех вышеописанных алгоритмах, занимает больше времени, чем все другие операции, выполняемые в программе (например, такие как сравнение ключей или вычисление индексов массива). Поэтому, хотя уже показано, что время работы всех трех алгоритмов пропорционально n^2, необходимо более детально сравнить их с учетом использования процедуры swap.

Сначала рассмотрим алгоритм "пузырька". В этом алгоритме процедура swap выполняется (см. листинг 8.1, строка (4)) не менее $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1) = n(n-1)/2$ раз, т.е. почти $n^2/2$ раз. Но поскольку строка (4) выполняется после условного оператора в строке (3), то можно ожидать, что число перестановок значительно меньше, чем $n^2/2$ В самом деле, в среднем перестановки выполняются только в половине из всех возможных случаев. Следовательно, ожидаемое число перестановок, если все возможные исходные последовательности ключей равновероятны, составляет примерно $n^2/4$. Для доказательства этого утверждения рассмотрим два списка ключей, которые обратны друг другу: L1=k1,k2,…,kn и L2=kn,kn-1,…,k1. Переставляются ключи ki и kj из одного списка, если они расположены в "неправильном" порядке, т.е. нарушают отношение линейного порядка, заданного на множестве значений ключей. Такая перестановка для ключей ki и kj выполняется только один раз, так как они расположены неправильно или только в

списке L1, или только в списке L2, но не в обоих сразу. Поэтому общее число перестановок в алгоритме "пузырька", применимом к спискам L1 и L2, равно числу пар элементов, т.е. числу сочетаний из n по 2 $C_n^2 = n(n-1)/2$. Следовательно, среднее число перестановок для списков L1 и L2 равно n(n - 1)/4 или примерно $n^2/4$. Поскольку для любого упорядочивания ключей существует обратное к нему, как в случае списков L1 и L2, то отсюда вытекает, что среднее число перестановок для любого списка также равно примерно $n^2/4$.

Число перестановок для сортировки вставками в среднем точно такое же, как и для алгоритма "пузырька". Для доказательства этого достаточно применить тот же самый аргумент: каждая пара элементов подвергается перестановке или в самом упорядочиваемом списке L и в обратном к нему, но никогда в обоих.

Если на выполнение процедуры swap необходимы большие вычислительные или временное ресурсы, то легко увидеть, что в этом случае сортировка посредством выбора более предпочтительна, нежели другие рассмотренные методы сортировки. В самом деле, в алгоритме этого метода (листинг 8.4) процедура swap выполняется вне внутреннего цикла, поэтому она выполняется точно n - 1 раз для массива длиной n. Другими словами, в этом алгоритме осуществляется O(n) перестановок в отличие от двух других алгоритмов, где количество таких перестановок имеет порядок $O(n^2)$. Причина этого понятна: в методе сортировки посредством выбора элементы "перескакивают" через большой ряд других элементов, вместо того чтобы последовательно меняться с ними местами, как это делается в алгоритме "пузырька" и алгоритме сортировки вставками.

В общем случае, если необходимо упорядочить длинные записи (поэтому их перестановки занимают много времени), целесообразно работать не с массивом записей, а с массивом указателей на записи, используя для сортировки любой подходящий алгоритм. В этом случае переставляются не сами записи, а только указатели на них. После упорядочивания указателей сами записи можно расположить в нужном порядке за время порядка O(n).

Ограниченность простых схем сортировки

Еще раз напомним, что каждый из рассмотренных в этом разделе алгоритмов выполняется за время порядка $O(n^2)$ как в среднем, так и в самом худшем случае. Поэтому для больших л эти алгоритмы заведомо проигрывают алгоритмам с временем выполнения O(n logn), которые будут описаны в следующем разделе. Значение n, начиная с которого быстрые алгоритмы сортировки становятся предпочтительнее простых методов сортировки, зависит от различных факторов, таких как качество объектного кода программы, генерируемого компилятором, компьютера,

на котором выполняется программа сортировки, или от размера записей. Определить такое критическое значение n для конкретных задач и вычислительного окружения (компилятор, компьютер и т.п.) можно после экспериментов с профайлером (профайлер — это подпрограмма протоколирования, позволяющая оценить время выполнения отдельных функций и операторов программы) и на основании выдаваемых им результатов. Практический опыт учит, что при n, не превышающем 100, на время выполнения программы влияет множество факторов, которые с трудом поддаются регистрации и поэтому не учтены в анализе, проведенном в этом разделе. Для небольших значений n рекомендуем применять простой в реализации алгоритм сортировки Шелла (Shell), который имеет временную сложность $O(n^{1,5})$.

5.3. Быстрая сортировка

Первый алгоритм с временем выполнения O(n logn), который рассмотрим далее, является, по-видимому, самым эффективным методом внутренней сортировки и поэтому имеет наименование "быстрая сортировка" или сортировка методом Хоара (Hoare C.A.R.). в этом алгоритме для сортировки элементов массива A[1],…,A[n] из этих элементов выбирается некоторое значение ключа v в качестве опорного элемента, относительно которого переупорядочиваются элементы массива. Желательно выбрать опорный элемент близким к значению медианы распределения значений ключей так, чтобы опорный элемент разбивал множество значений ключей на две примерно равные части. Далее элементы массива переставляются так, чтобы для некоторого индекса j все переставленные элементы A[1],…,A[j] имели значения ключей, меньшие чем v, а все элементы A[j+1],…,A[n] — значения ключей, большие или равные v. Затем процедура быстрой сортировки рекурсивно применяется к множествам элементов A[1], …, A[j] и . A[j+1],…,A[n] для упорядочивания этих множеств по отдельности. Поскольку все значения ключей в первом множестве меньше, чем значения ключей во втором множестве, то исходный массив будет отсортирован правильно.

Теперь начнем разрабатывать рекурсивную процедуру quicksort(i, j), которая будет работать с элементами массива A, определенным вне этой процедуры. Процедура quicksort(i, j) должна упорядочить элементы A[i], ..., A[j]. Предварительный набросок процедуры показан в листинге 8.5. Отметим, что если все элементы A[i], ..., A[j] имеют одинаковые ключи, над ними не производятся никакие действия.

Листинг. Процедура быстрой сортировки

if A[i], ..., A[j] имеют не менее двух различных ключей then begin

```
пусть v — наибольший из первых двух найденных различных
ключей;
переставляются элементы A[i], …, A[j] так, чтобы
для некоторого k, i+l≤k≤j, A[i], ..., A[k-l] имели ключи
меньшие, чем v, а A[k], ..., A[j] — большие или равные v;
quicksort(i, k-1);
quicksort{k, j);
end
```

Напишем функцию findpivot (нахождение опорного элемента), реализующую проверку в строке (1) листинга, т.е. проверяющую, все ли элементы A[i], ..., A[j] одинаковы. Если функция findpivot не находит различных ключей, то возвращает значение 0. В противном случае она возвращает индекс наибольшего из первых двух различных ключей. Этот наибольший ключ становится опорным элементом. Код функции findpivot приведен в листинге.

Листинг 8.6. Функция findpivot

```
function findpivot ( i, j: integer ): integer;
var
firstkey: keytype;
{ примет значение первого найденного ключа, т.е. A[i].key }
k: integer; { текущий индекс при поиске различных ключей}
begin
firstkey:= A[i].key;
for k:= i + 1 to J do { просмотр ключей }
if A[k] .key > firstkey then { выбор наибольшего ключа)
return (k)
else if A [k] . key  firstkey then
return (i) ;
return (0)  {различные ключи не найдены}
end;  {findpivot}
```

Теперь реализуем строку (3) из листинга, где необходимо переставить элементы A[i], …,A[j] так, чтобы все элементы с ключами, меньшими опорного значения, располагались слева от остальных элементов. Чтобы выполнить эти перестановки введем два курсора l и r, указывающие соответственно на левый и правый концы той части массива A, где в настоящий момент переставляем (упорядочиваем) элементы. При этом считаем, что уже все элементы A[i], …, A[l - 1], расположенные слева от l, имеют значения ключей, меньшие опорного значения. Соответственно элементы A[r + 1], …, A[j], расположенные справа от r, имеют значения

ключей, большие или равные опорному значению (рис.40). Нам необходимо рассортировать элементы A[l], …, A[r].
Ключи опорное значение
Ключи опорное значение

Рисунок 40 – Ситуация, возникающая в процессе перемещения элементов.

Сначала положим l=i и r=j. Затем будем повторять следующие действия, которые перемещают курсор l вправо, а курсор r влево до тех пор, пока курсоры не пересекутся.

1. Курсор перемещается l вправо, пока не встретится запись с ключом, не меньшим опорного значения. Курсор r перемещается влево, также до тех пор, пока не встретится запись с ключом, меньшим опорного значения. Отметим, что выбор опорного значения функцией findpivot гарантирует, что есть по крайней мере одна запись с ключом, значение которого меньше опорного значения, и есть хотя бы одна запись со значением опорного ключа. Поэтому обязательно существует промежуток между элементами A[i] и A[j], по которому могут перемещаться курсоры l и r.
2. Выполняется проверка : если l r (на практике возможна только ситуация, когда l =r + 1), то перемещение элементов A[i], …, A[j] заканчивается.
3. В случае l r (очевидно, что случай l = l невозможен) переставляем местами элементы A[l] и A[r]. После этого запись A[l] будет иметь значение ключа меньшее, чем опорное значение, а A[r] - большее или равное опорному значению. Курсор l перемещается на одну позицию от предыдущего положения вправо, а курсор r - на одну позицию влево. Далее процесс продолжается с пункта 1.

Описанный циклический процесс несколько неуклюжий, поскольку проверка, приводящая к окончанию процесса, расположена посредине. Если этот процесс оформить в виде цикла repeat, то этап 3 надо переместить в начало. Но в этом случае при l =i и r =j сразу меняются местами элементы A[i] и A[y], независимо от того, надо это делать или нет.

Но с другой стороны, это несущественно, так как не предполагаем первоначальное упорядочивание элементов A[i], …, A[j]. В любом случае, читатель должен знать о таких шалостях алгоритма, поэтому они не должны его озадачивать. Функция partition (разделение), которая выполняет перестановки элементов и возвращает индекс l, указывающий на точку разделения данного фрагмента массива А на основе заданного опорного значения pivot, приведена в листинге.

Листинг. Функция partition

```
function purtition (i, j: integer; pivot: keytype ) : integer;
var
l, r: integer; {курсоры}
begin

l:= i;
r:= j;
repeat
swap(A[l], A[r]);
while A[l] .key < pivot do
l:=l+1;
while A[r].key >= pivot do
r:= r - 1
until
l > r;
return(l)
end; { partition }
```

Теперь можно преобразовать эскиз алгоритма быстрой сортировки (листинг) в настоящую программу quicksort (быстрая сортировка). Код этой программы приведен в листинге. Для сортировки элементов массива А типа array[1..n] of recordtype надо просто вызвать процедуру quicksort(1,n).

Листинг. Процедура быстрой сортировки

```
procedure quicksort ( i, j: integer );
{ сортирует элементы A[1], .... A[j] внешнего массива A }
var
pivot: keytype; { опорное значение }
pivotindex: integer;
{ индекс элемента массива A, чей ключ равен pivot }
k: integer;
```

```
{начальный индекс группы элементов,, чьи ключи > pivot}
begin
pivotindex:= findpivotd(i, j) ;
if pivotindex <> 0 then begin
{ если все ключи равны, то ничего делать не надо }
pivot := A[pivotindex] .key;
k:= portition (i, j, pivot);
Quicksort(i, k - 1);
Quicksort (k, j)
end
end; {quicksort}
```

Временная сложность быстрой сортировки

Покажем, что время выполнения быстрой сортировки n элементов составляет в среднем O(n long) и O(n2) в худшем случае. В качестве первого шага в доказательстве обоих утверждений надо показать, что время выполнения процедуры partition(i, j, v) имеет порядок O(j - i + 1), т.е. пропорционально числу элементов, над которыми она выполняется.

Чтобы "разобраться" с временем выполнения процедуры partition, воспользуемся приемом, который часто применяется при анализе алгоритмов. Надо найти определенные элементы, для которых в принципе можно вычислить время выполнения, а затем необходимо показать, что каждый шаг алгоритма для обработки одного элемента требует времени, не превышающего константы. Тогда общее время выполнения алгоритма можно вычислить как произведение этой константы на количество элементов.

В данном случае в качестве элементов можно просто взять элементы A[i], …,A[j] и для каждого из них оценить время, необходимое для достижения в процедуре partition курсорами l и r этого элемента, начиная от исходного положения этих курсоров. Сначала отметим, что эта процедура всегда возвращает значение курсора, разбивающее множество элементов A[i], …, A[j] на две группы, причем каждая из этих групп содержит не менее одного элемента. Далее, так как процедура заканчивается только тогда, когда курсор l перейдет через курсор r, то отсюда следует, что каждый элемент просматривается процедурой хотя бы один раз.

Перемещение курсоров осуществляется в циклах строк (4) и (6) листинга путем увеличения на 1 значения l или уменьшения на 1 значения r. Сколько шагов может сделать алгоритм между двумя выполнениями оператора l: = l + 1 или оператора r:= r + 1? Для того чтобы смоделировать самый худший случай, надо обратиться к началу процедуры. В строках (1) и (2) происходит инициализация курсоров l и r. Затем обязательно выполняется перестановка элементов в строке(3). Далее предположим, что

циклы строк (4) и (6) перескакивают без изменения значений курсоров l и r. На втором и последующих итерациях цикла repeat перестановка в строке (3) гарантирует, что хотя бы один из циклов while строк (4) и (6) , будет выполнен не менее одного раза. Следовательно, между выполнениями оператора l:== l + 1 или r:=r-1 в самом худшем случае выполняются строки (1), (2), дважды строка (3) и проверка логических условий в строках (4), (6), (8) и снова в строке (4). Все эти операции требуют фиксированного времени, независящего от значений i и j.

В конце исполнения процедуры выполняются проверки в строках (4), (6) и (8), не связанные ни с какими "элементами", но время этих операций также фиксировано поэтому его можно "присоединить" к времени обработки какого-либо элемента. Из всего вышесказанного следует вывод, что существует такая временная константа с, чго на обработку любого элемента затрачивается время, не превышающее с. Поскольку процедура partition (i, j, v) обрабатывает j - i + 1 элементов, то, следовательно, общее время выполнения этой процедуры имеет порядок O(j - i + 1).

Вернемся к оценке времени выполнения процедуры quicksort (i, j). Легко проверить, что вызов процедуры findpivot в строке (1) листинга занимает время порядка O(j - i + 1), а в большинстве случаев значительно меньше. Проверка логического условия в строке (2) выполняется за фиксированное время, так же, как и оператор в строке (3), если, конечно, он выполняется. Вызов процедуры partition (i, j, pivot), как показано, требует времени порядка O(o - i + 1). Таким образом, без учета рекурсивных вызовов quicksort каждый отдельный вызов этой процедуры требует времени, по крайней мере пропорционального количеству элементов, упорядочиваемых этим вызовом quicksort.

Если посмотреть с другой стороны, то общее время выполнения процедуры quicksort является суммой по всем элементам количества времени, в течение которого элементы находятся в части массива А, обрабатываемой данным вызовом процедуры quicksort.

Время выполнения быстрой сортировки в среднем

Как всегда, выражение "время выполнения в среднем" понимается как усреднение времени выполнения сортировки по всем возможным упорядочениям исходных наборов элементов в предположении, что все упорядочения равновероятны. Для простоты также предположим, что не существует записей с одинаковыми ключами. В общем случае при возможном равенстве значений некоторых ключей анализ времени выполнения также осуществляется сравнительно просто, но несколько по-иному.

Сделаем еще одно предположение, которое тоже упрощает анализ. Будем считать, что при вызове процедуры quicksort (i, j) все упорядочивания элементов A[i], …, A[j] равновероятны. Это

предположение можно попытаться обосновать исходя из того очевидного факта, что опорное значение и, примененное в предыдущем вызове quicksort, нельзя использовать для разбиения данного подмножества элементов, так как здесь все элементы имеют значения ключей либо меньшие и, либо большие, либо равные v. Более тщательный анализ программы быстрой сортировки показывает, что предыдущий опорный элемент, скорее всего будет находиться возле правого конца подмножества элементов, равных или больших этого опорного значения (т.е. не будет находиться с равной вероятностью в любом месте этого подмножества), но для больших множеств этот факт не имеет определяющего значения.

Обозначим через T(n) среднее время, затрачиваемое алгоритмом быстрой сортировки на упорядочивание последовательности из n элементов. Очевидно, что время T(1) равно некоторой константе c1, поскольку имеется только один элемент и не используется рекурсивный вызов процедуры quicksort самой себя. При n>1 требуется время c2т (c2 - некоторая константа), чтобы определить опорное значение и разбить исходное множество элементов на два подмножества; после этого вызывается процедура quicksort для каждого из этих подмножеств. Было бы хорошо (с точки зрения анализа алгоритма), если бы можно утверждать, что для подмножества, которое упорядочивается данным вызовом процедуры quicksort, опорное значение с одинаковой вероятностью может быть равным любому из n элементов исходного множества. Но в этом случае нельзя гарантировать, что такое опорное значение сможет разбить это подмножества на два других непустых подмножества, одно из которых содержало бы элементы, меньшие этого опорного значения, а другое — равные или большие его. Другими словами, в рассматриваемом подмножестве с ненулевой вероятностью могут находиться только элементы, меньшие этого опорного значения, либо, наоборот, большие или равные опорному значению. Именно по этой причине в алгоритме быстрой сортировки опорное значение выбирается как наибольшее значение ключей первых двух различных элементов. Отметим, что такое определение опорного значения не приводит к эффективному (т.е. примерно равному) размеру множеств, на которые разобьется рассматриваемое подмножество относительно этого опорного значения. При таком выборе опорного значения можно заметить тенденцию, что "левое" подмножество, состоящее из элементов, чьи ключи меньше опорного значения, будет больше "правого" подмножества, состоящего из элементов, ключи которых больше или равны опорному значению.

Предполагая, что все сортируемые элементы различны, найдем вероятность того, что "левое" подмножество будет содержать i из n элементов. Для того чтобы "левое" подмножество содержало i элементов, необходимо, чтобы опорное значение совпадало с (i + 1)-м порядковым

значением в последовательности упорядоченных в возрастающем порядке элементов исходного множества. При нашем методе выбора опорного значения данное значение можно получить в двух случаях: (1) если элемент со значением ключа, равного (i + 1)-му порядковому значению, стоит в первой позиции, во второй позиции стоит любой из i элементов с ключами, меньшими, чем у первого элемента; (2) элемент со значением ключа, равного новому опорному значению, стоит во второй позиции, а в первой — любой из i элементов с ключами, меньшими, чем у второго элемента. Вероятность того, что отдельный элемент, такой как (i+1)-е порядковое значение, появится в первой позиции, равна 1/n. Вероятность того, что во второй позиции будет стоять любой из i элементов со значением ключа, меньшим, чем у первого элемента, равна i/(n - 1). Поэтому вероятность того, что новое опорное значение находится в первой позиции, равна i/n(n - 1). Такое же значение имеет вероятность события, что новое опорное значение появится во второй позиции. Отсюда вытекает, что "левое" множество будет иметь размер i с вероятностью 2i/n(n - 1) для $1 \le i \le n$.

Теперь можно записать рекуррентное соотношение для T(n):

$$T(n) \le \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \left[T(i) + T(n-i) + c_2 n \right] \qquad (1)$$

Неравенство (1) показывает, что среднее время выполнения алгоритма быстрой сортировки не превышает времени c_2n, прошедшего от начала алгоритма до рекурсивных вызовов quicksort, плюс среднее время этих рекурсивных вызовов. Последнее время является суммой по всем возможным i произведений вероятностей того, что "левое" множество состоит из i элементов (или, что то же самое, вероятностей того, что "правое" множество состоит из n - i элементов) и времени рекурсивных вызовов T(i) и T(n - i).

Теперь надо упростить сумму в выражении (1). Сначала заметим, что для любой функции f(i) путем замены i на n-i можно доказать следующее равенство:

$$\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i) \qquad (2)$$

из равенства (2) следует, что

$$\sum_{i=1}^{n-1} f(i) = \frac{1}{2} \sum_{i=1}^{n-1} \left(f(i) + f(n-i) \right) \qquad (3)$$

Положив f(i) равными слагаемым в выражении (1) и применяя

равенство (3), из неравенства (1) получим

$$T(n) \le \frac{1}{2}\sum_{i=1}^{n-1}\left\{\frac{2i}{n(n-1)}\left[T(i)+T(n-i)\right]+\frac{2(n-i)}{n(n-1)}\left[T(n-i)+T(i)\right]\right\}+c_2 n \le \frac{1}{n-1}\sum_{i=1}^{n-1}\left[T(i)+T(n-i)\right]+c_c n$$

(4)

Далее снова применяем формулу (3) к последней сумме (4), положив f(i) = T(i) :

$$T(n) \le \frac{2}{n-1}\sum_{i=1}^{n-1}T(i)+c_2 n \qquad (5)$$

Отметим, что последнее неравенство в (4) соответствует выражению, которое получили бы, если бы предполагали равные вероятности для всех размеров (от 1 до n) левых множеств. Покажем, как можно оценить сверху решение рекуррентного неравенства (5). Докажем, что для всех $n \ge 2 T(n) \le cn \log n$ для некоторой константы c.

Доказательство проведем методом индукции по n. Для n = 2 непосредственно из неравенства (5) для некоторой константы c получаем $T(2) \le 2c = 2c \log 2$. Далее, в соответствии с методом математической индукции, предположим, что для $i \le n$ выполняются неравенства $T(i) \le ci \log i$. Подставив эти неравенства в формулу (5), для T(n) получим

$$T(n) \le \frac{2c}{n-1}\sum_{i=1}^{n-1} i\log i + c_2 n \qquad (6)$$

Разобьем сумму в (6) на две суммы: в первой $i \le n/2$ во второй $i > n/2$. В первой сумме $\log i \le \log(n/2) = \log n - 1$, во второй сумме log i не превышают log n. Таким образом, из (8.6) получаем

$$\begin{aligned} T(n) &\le \frac{2c}{n-1}\left[\sum_{i=1}^{n/2} i\log i + \sum_{i=n/2+1}^{n-1} i\log i\right]+c_2 n \le \\ &\le \frac{2c}{n-1}\left[\sum_{i=1}^{n/2} i(\log n - 1) + \sum_{i=n/2+1}^{n-1} i\log n\right]+c_2 n \le \\ &\le \frac{2c}{n-1}\left[\frac{n}{4}\left(\frac{n}{2}+1\right)\log n - \frac{n}{4}\left(\frac{n}{2}+1\right)+\frac{3}{4}n\left(\frac{n}{2}-1\right)\log n\right]+c_2 n = \qquad (7) \\ &= \frac{2c}{n-1}\left[\left(\frac{n^2}{2}-\frac{n}{2}\right)\log n - \left(\frac{n^2}{8}+\frac{n}{4}\right)\right]+c_2 n \le \\ &\le cn\log n - \frac{cn}{4} - \frac{cn}{2(n-1)} + c_2 n \end{aligned}$$

Если положить $c \ge 4c_2$, тогда в последнем выражении сумма второго и четвертого слагаемых не превысит нуля. Третье слагаемое в последней сумме (7) отрицательное, поэтому можно утверждать, что $T(n) \le cn \log n$ для константы $c = 4c_2$. Этим заканчивается доказательство того, среднее время выполнения алгоритма быстрой сортировки составляет O(n logn).

Реализация алгоритма быстрой сортировки

Алгоритм быстрой сортировки можно реализовать не только посредством процедура quiksort так, чтобы среднее время выполнения равнялось O(n logn). Можно создать другие процедуры, реализующие этот алгоритм, которые будут иметь тот же порядок O(n logn) времени выполнения в среднем, но за счет меньшей константы пропорциональности в этой оценке буду работать несколько быстрее (напомним, что порядок времени выполнения, такой как O(n logn), определяется с точностью до константы пропорциональности). Эту константу можно уменьшить, если выбирать опорное значение таким, чтобы оно разбивало рассматриваемое в данный моент подмножество элементов на две примерно равные части. Если всегда такие подмножества разбиваются точно пополам, тогда каждый сортируемый элемент имеет глубину точно logn в дереве. Для сравнения: средняя глубина элементов в процедуре quicksort (листинг) составляет 1.4 logn. Так что можно надеяться, что при "правильном" выборе опорного значения выполнение алгоритма ускорится.

Например, можно с помощью датчика случайных чисел выбрать три элемента из подмножества и средний (по величине) из них назначить опорным значением. Можно также, задав некоторое число k, выбрать случайным образом (например, опять с помощью датчика случайных чисел) k элементов из подмножества, упорядочить их процедурой quicksort или посредством какой-либо простой сортировки. В качестве опорного значения взять медиану (т.е. (k + 1)/2-й элемент) этих k элементов. Заметим в скобках, что интересной задачей является определение наилучшего значения k как функции от количества элементов в подмножестве, подлежащем сортировке. Если k очень мало, то среднее время будет близко к тому, как если бы выбирался только один элемент. Если же k очень велико, то уже необходимо учитывать время нахождения медианы среди k элементов. Другие реализации алгоритма быстрой сортировки можно получить в зависимости от того, что делаем в случае, когда получаются малые подмножества.

Существует еще один метод "ускорения" быстрой сортировки за счет увеличения используемого пространства памяти компьютера. Отметим, что этот метод применим к любым алгоритмам сортировки. Если есть достаточный объем свободной памяти, то можно создать массив указателей на записи массива A. Затем следует организовать процедуру сравнения ключей записей посредством этих указателей. В этом случае нет необходимости в физическом перемещении записей, достаточно перемещать указатели, что значительно быстрее перемещения непосредственно записей, особенно когда они большие (состоят из многих полей). В конце выполнения процедуры сортировки указатели будут

отсортированы слева направо, указывая на записи в нужном порядке. Затем сравнительно просто можно переставить сами записи в правильном порядке.

Таким образом, можно сделать только n перестановок записей вместо O(n logn) перестановок, и эта разница особенно значительна для больших записей. Отрицательными аспектами такого подхода являются использование дополнительного объема памяти для массива указателей и более медленное выполнение операции сравнения ключей, поскольку здесь сначала, следуя за указателями, надо найти нужные записи, затем необходимо войти в записи и только потом взять значения поля ключа.

5.4. Пирамидальная сортировка

В этом разделе рассмотрим алгоритм сортировки, называемой пирамидальной, его время выполнения в худшем случае такое, как и в среднем, и имеет порядок O(n logn). Этот алгоритм можно записать в абстрактной (обобщенной) форме, используя операторы множеств INSERT, DELETE, EMPTY и MIN. Обозначим через L список элементов, подлежащих сортировке, а S - множество элементов типа recordtype (тип записи), которое будет использоваться для хранения сортируемых элементов. Оператор MIN возвращает запись из множества S с минимальным значением ключа. В листинге показан абстрактный алгоритм сортировки, который далее преобразуем в алгоритм пирамидальной сортировки.

Листинг. Абстрактный алгоритм сортировки

```
For x∈L do
INSERT (x,S);
While not EMPTY(S) do begin
y:=MIN(S);
writeln(y);
DELETE(y,S);
end
```

Если список L содержит n элементов, то наш абстрактный алгоритм требует выполнения n операторов INSERT, n операторов MIN, n операторов DELETE и n + 1 операторов EMPTY. Таким образом, общее время выполнения алгоритма имеет порядок O(n logn), если, конечно, используется подходящая структура данных.

Частично упорядоченное дерево поддерживает операторы INSERT и DELETEMIN с временем выполнения O(logn). Но на частично упорядоченном дереве нельзя реализовать общий оператор DELETE с

временем выполнения O(logn) (всегда найдется отдельный элемент, требующий линейного времени удаления в самом худшем случае). В связи с этим отметим, что в листинге удаляются только элементы, имеющие минимальные значения ключей. Поэтому строки (4) и (6) можно объединить одним оператором DELETEMIN, который будет возвращать элемент у. Таким образом, используя структуру данных частично упорядоченного дерева, можно выполнить абстрактный алгоритм сортировки за время O(n logn).

Сделаем еще одно изменение в алгоритме листинга, чтобы избежать печати удаляемых элементов. Заметим, что множество S всегда хранится в виде кучи в верхней части массива А, даже когда содержит только i элементов ($i < n$). Для частично упорядоченного дерева наименьший элемент всегда хранится в ячейке A[1]. Элементы, уже удаленные из множества S, можно было бы хранить в ячейках A[i+1], …, A[n], отсортированным в обратном порядке, т.е. так, чтобы выполнялись неравенства: $A[i+1] \geq A[i+2] \geq \ldots \geq A[n]$. Поскольку элемент A[1] является наименьшим среди элементов A[1],…A[i], то для повышения эффективности оператора DELETEMIN можно просто поменять местами элементы A[1] и A[i]. Поскольку новый элемент A[i] (т.е. старый A[1]) не меньше, чем A[i+1] (элемент A[i+1] был удален из множества S на предыдущем шаге как наименьший), то получим последовательность A[i], …, A[n], отсортированным в убывающем порядке. Теперь надо заняться размещением элементов A[1], …, A[i-1].

Поскольку новый элемент A[1] (старый элемент A[i]) нарушает структуру частично упорядоченного дерева, его надо "протолкнуть" вниз по ветвям дерева, как это делается в процедуре DELETEMIN листинга. Здесь для этих целей используем процедуру pushdown (протолкнуть вниз), оперирующую с массивом А, определенным вне этой процедуры. Код процедуры pushdown приведен в листинге. С помощью последовательности перестановок данная процедура "проталкивает" элемент A[first] вниз по дереву до нужной позиции. Чтобы восстановить частично упорядоченное дерево в случае нашего алгоритма сортировки, надо вызвать процедуру pushdown для first=1.

Листинг. Процедура pushdown

```
procedure pushdown { first, last: integer );
{ Элементы A[ first] ,…, A[last] составляют частично
упорядоченное дерево за исключением, возможно, элемента
A[first] и его сыновей. Процедура pushdown
восстанавливает частично упорядоченное дерево }
var
г: integer; { указывает текущую позицию A[first] }
```

```
begin
r:= first; { инициализация }
while г <= last div 2 do
if last = 2*r then begin
{ элемент в позиции г имеет одного сына t, в позиции 2*r }
if A[r].key > A[2*r].key then
swap(A[r], A[2*r]);
r:= last { досрочный выход из цикла while}
end
else { элемент в позиции r имеет двух сыновей
в позициях 2*r и 2*r + 1 }
if A[r].key > A[2*r].key and
A[2*r].key <= A[2*r + 1].key then begin
{ перестановка элемента в позиции
с левым сыном }
swap(A[r], A[2*r]);
г:= 2*r
end
else if A[r].key > A[2*r + 1].key and
A[2*r + 1].key < A[2*r].key then begin
{ перестановка элемента в позиции с правым сыном }
swap(A[r], A[2*r+1]);
end
else {элемент в позиции r не нарушает порядок в частично упорядоченном дереве}
r:=last {выход из цикла while}
end; {pushdown}
```

Вернемся к листингу и займемся строками (4)-(6). Выбор минимального элемента в строке (4) прост — это всегда элемент A[1]. Чтобы исключить оператор печати в строке (5), поменяем местами элементы A[1] и A[i] в текущей куче. Удалить минимальный элемент из текущей кучи также легко: надо просто уменьшить на единицу курсор i, указывающий на конец текущей кучи. Затем надо вызвать процедуру pushdown(l, i - 1) для восстановления порядка в частично упорядоченном дереве кучи A[1], …, A[i - 1].

Вместо проверки в строке (3), не является ли множество S пустым, можно проверить значение курсора i, указывающего на конец текущей кучи. Теперь осталось рассмотреть способ выполнения операторов в строках (1), (2). Можно с самого начала поместить элементы списка L в массив A в произвольном порядке. Для создания первоначального частично упорядоченного дерева надо последовательно вызывать процедуру pushdown(j, n) для всех j = n/2, n/2 - 1, ..., 1. Легко видеть, что

после вызова процедуры pushdown(j, n) порядок в ранее упорядоченной части строящегося дерева не нарушается, так как новый элемент, добавляемый в дерево, не вносит нового нарушения порядка, поскольку он только меняется местами со своим "меньшим" сыном. Полны код процедуры heapsort (пирамидальная сортировка) показан в листинге

Листинг. Процедура пирамидальной сортировки

```
procedure heapsort;
{ Сортирует элементы массива A[1], ..., A[n]
в убывающем порядке }
var
i: integer; { курсор в массиве A }
begin
{ создание частично упорядоченного дерева }
for i: = n div 2 downto 1 do
pushdown (i, n);
for i:= n downto 2 do begin
swap(A[l], A[i]);
{ удаление минимального элемента из кучи }
pushdown (1, i-1)
{ восстановление частично упорядоченного дерева )
end
end; { heapsort }
```

Анализ пирамидальной сортировки

Сначала оценим время выполнения процедуры pushdown. Из листинга видно, что тело цикла while (т.е. отдельная итерация этого цикла) выполняется за фиксированное время. После каждой итерации переменная r по крайней мере вдвое увеличивает свое значение. Поэтому, учитывая, что начальное значение r равно first, после i итераций будем иметь $r \geq first*2^i$. Цикл while выполняется до тех пор, пока $r > last/2$. Это условие будет выполняться, если выполняется неравенство $first * 2^i > last/2$. Последнее неравенство можно переписать как

$$i > \log(last/first) - 1 \qquad (*)$$

Следовательно, число итераций цикл while в процедуре pushdown не превышает log(last/first).

Поскольку $first \geq 1$ и $last \leq n$, то из (*) следует, что на каждый вызов процедуры в строках (2) и (5) листинга затрачивалось время, по порядку не большее, чем O(log n). Очевидно, что цикл for строк (1), (2) выполняется n/2 раз. Поэтому общее время выполнения этого цикла имеет порядок O(n

log n). Цикл в строках (3) - (5) выполняется n-1 раз. Следовательно, на выполнение всех перестановок в строке (4) тратится время порядка O(n) а на восстановление частично упорядоченного дерева (строка (5)) - O(n log n). Отсюда вытекает, что общее время выполнения цикла в строках (3) - (5) имеет порядок O(n log n) и такой же порядок времени выполнения всей процедуры heapsort.

Несмотря на то, что процедура heapsort имеет время выполнения порядка O (n log n) в самом худшем случае, в среднем ее время несколько хуже, чем в быстрой сортировке, хотя и имеет тот же порядок. Пирамидальная сортировка интересна в теоретическом плане, поскольку это первый рассмотренный алгоритм, имеющий в самом худшем случае время выполнения O(n log n). В практических ситуациях этот алгоритм полезен тогда, когда надо не сортировать все n элементов списка, а только отобрать k наименьших элементов, и при этом k значительно меньше n. Если надо сделать только итераций цикла (3) - (5), то на это потратится только время порядка O (k log n). Поэтому отбор k минимальных элементов процедура heapsort выполнит за время порядка O (n+k log n) Отсюда следует, что при $k \leq n/\log n$ на выполнение этой операции потребуется время порядка O(n).

5.5. Карманная сортировка

Правомерен вопрос: всегда ли при сортировке n элементов нижняя граница времени выполнения имеет порядок Ω (n log n)? В следующем разделе рассмотрим алгоритмы сортировки и их нижнюю границу времени выполнения, если о типе данных ключей не предполагается ничего, кроме того, что их можно упорядочить посредством некой функции, показывающей, когда один ключ меньше, чем другой. Часто можно получить время сортировки меньшее, чем Ω (n log n) но необходима дополнительная информация о сортируемых ключах.

Пример. Предположим, что значения ключей являются целыми числами из интервала от 1 до n, они не повторяются и число сортируемых элементов также равно n. Если обозначить через A и B массивы типа array [1..n] of recordtype, n элементов, подлежащих сортировке, первоначально находятся в массиве A, тогда можно организовать поочередное помещение в массив B записей в порядке возрастания значений ключей следующим образом:

```
for i:=1 to n do
B[A[i].key]:=A[i]
```

Этот код вычисляет, где в массиве B должен находиться элемент A[i] и помещает его туда. Весь этот цикл требует времени порядка O(n) и работает корректно только тогда, когда значения всех ключей различны и являются целыми числами из интервала от 1 до n.

Существует другой способ сортировки элементов массива A с временем O(n), но без использования второго массива B. Поочередно посетим элементы A[1], …, A[n]. Если запись в ячейке A[i] имеет ключ j и j≠i, то меняются местами записи в ячейках A[i] и A[j]. Если после этой перестановки новая запись в ячейке A[i] имеет ключ k и k≠1 то осуществляется перестановка между A[i] и A[k] и т.д. Каждая перестановка помещает хотя бы одну запись в нужном порядке. Поэтому данный алгоритм сортировки элементов массива A на месте имеет время выполнения порядка O(n).

```
for i:=1 to n do
while A[i].key <> i do
swap (A[i], A[A[i].key]);
```

Приведенная программа - это простой пример "карманной" сортировки, где создаются "карманы" для хранения записей с определенным значением ключа. Далее проверяем, имеет ли данная запись ключ со значением r, и если это так, то помещаем эту запись в "карман" для записей, чьи значения ключей равны r. В программе "карманами" являются элементы массива B, где B[i] - "карман" для записей с ключевым значением i. Массив в качестве карманов можно использовать только в простейшем случае, когда известно, что не может быть более одной записи в одном "кармане". Более того, при использовании массива в качестве "карманов" не возникает необходимости в упорядочивании элементов внутри "кармана", так как в одном "кармане" содержится не более одной записи, а алгоритм построен так, чтобы элементы в массиве располагались в правильном порядке.

Но в общем случае нужно быть готовым к тому, что в одном кармане может храниться несколько записей, а также должны уметь объединять содержимое нескольких карманов в один, располагая элементы в объединенном "кармане" в правильном порядке. Для определенности далее будем считать, что элементы массива A имеют тип данных recordtype, а ключи записей - тип keytype. Кроме того, примем, но только в этом разделе, что тип данных keytype является перечисленным типом, то есть таким, как последовательность целых чисел 1, 2, …, n, или как символьные строки. Обозначим через listtype (тип списка) тип данных, который представляет списки элементов типа recordtype. Тип listtype может быть любым типом списков, описанным ранее, но в данном случае нам наиболее подходят связанные списки, поскольку будем их наращивать в "карманах" до размеров, которые нельзя предусмотреть заранее. Можно только предвидеть, что общая длина всех списков фиксирована и равна n, поэтому при необходимости массив из n ячеек может обеспечить реализацию списков для всех карманов.

Необходим также массив B типа array[keytype] of listtype. Это будет "массив" карманов, хранящих списки (или заголовки списков, если используются связанные списки). Индексы массива B имеют тип данных keytype так что каждая ячейка этого массива соответствует одному значению ключа. Таким образом, уже можно обобщить программу, поскольку теперь "карманы" имеют достаточную емкость.

Рассмотрим, как можно выполнить объединение "карманов". Формально над списками a_1, a_2, …, a_i и b_1, b_2, …, b_j надо выполнить операцию *конкатенации* списков, в результате которой получим список a_1, a_2, …, a_i, b_1, b_2, …, b_j. Для реализации оператора конкатенации CONCATENATE (L_1, L_2), который заменяет список L_1 объединенным списком L_1L_2, можно использовать любые представления списков.

Для более эффективного выполнения оператора конкатенации в добавление к заголовкам списков можно использовать указатели на последние элементы списков (или на заголовок списка, если список пустой). Такое нововведение позволит избежать просмотра всех элементов списка для нахождения последнего. На рис. 41 пунктирными линиями показаны измененные указатели при конкатенации списков L_1 и L_1 в один общий список L_1. Список L_2 после объединения списков предполагается "уничтоженным", поэтому указатели на его заголовок и конец "обнуляются".

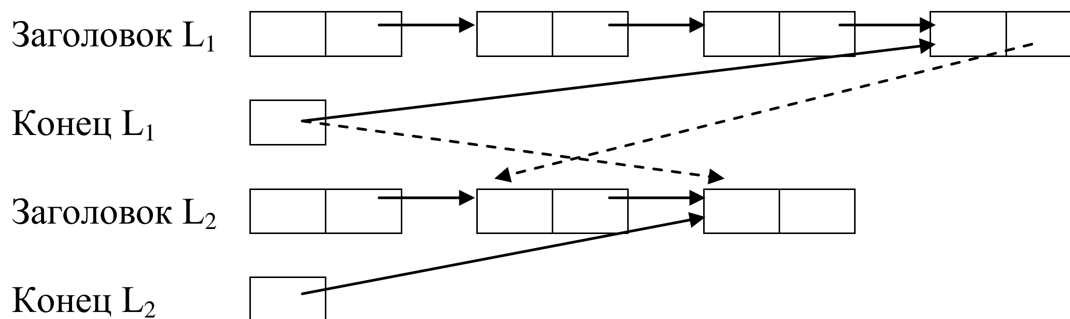


Рисунок 41 – Конкатенация связанных списков.

Теперь можно написать программу "карманной" сортировки записей, когда поле ключей является перечислимым типом. Эта программа, показанная в листинге, написана с использованием базовых операторов, выполняемых над списками. Как говорилось, связанные списки являются предпочтительной реализацией "карманов", но, конечно, можно использовать и другие реализации. Напомним также, что в соответствии с нашими предположениями, массив A типа array [1..n] of recordtype содержит элементы, подлежащие сортировке, а массив B типа array[keytype] of listtype предназначен для "карманов". Будем также считать, что переменные перечислительного типа keytype могут

изменяться в пределах от lowkey (нижний ключ) до highkey (верхний ключ).

Листинг. Программа binsort ("карманная" сортировка)

```
procedure binsort;
{ Сортирует элементы массива A, помещая отсортирована
список в "карман" B[lowkey] }
var
i: integer;
v: keytype;
begin
{ начинается занесение записей в "карманы" }
for 2:= 1 to n do
{ помещение элемента A[l] в начало "кармана",
соответствующего ключу этого элемента }
INSERT(A[i], FIRST(B[A[1].key]). B[A[1].key]);
for v:= succ(lowkey) to highkey do
{конкатенация всех "карманов" в конец "карман" B[lowkey]}
CONCATENATE(B[lowkey], B[v])
end; { binsort }
```

Анализ "карманной" сортировки

Если сортируется n элементов, которые имеют m различных значений ключей (следовательно, m различных "карманов"), тогда программа из листинга выполняется за время порядка O(n+m), если, конечно, используется подходящая структура данных. В частности, если m≤n, то сортировка выполняется за время O(n). В качестве "подходящей" структуры данных можно принять связанные списки. Указатели на концы списков, показанные на рис. 41, полезны, но не обязательны.

Цикл в строках (1), (2) листинга помещают записи в "карманы", выполняется за время порядка O(n), поскольку оператор INSERT в строке (2) требует фиксированного времени, так как вставка записей всегда осуществляется в начало списков. Разберемся с циклом в строках (3), (4), осуществляющих конкатенацию списков. Сначала временно предположим, что указатели на концы списков существуют. Тогда строка (4) выполняется за фиксированное время, а весь цикл требует времени порядка O(m). Следовательно, в этом случае вся программа выполняется за время O(n+m).

Если указатели на концы списков не используются, тогда в строке (4) до начала непосредственного процесса слияния списка B[v] со списком B[lowkey] необходимо просмотреть все элементы списка B[v], чтобы найти его конец. Заметим, что конец списка B[lowkey] искать не надо, он

известен. Дополнительное время, необходимое для поиска концов всех "карманов", не превышает O(n) поскольку длина всех "карманов" равна n. Это дополнительное время не влияет на порядок времени выполнения всей процедуры binsort, поскольку O(n) не больше, чем O(n+m).

5.6. Сортировка множеств с большими значениями ключей

Если количество различных ключей m не превосходит количества элементов n, то время выполнения O(n+m) программы из листинга в действительности будет иметь порядок O(n). Но каково будет это время, если $m=n^2$, например? По-видимому, если время выполнения программы O(n+m), то должно быть $O(n+n^2)$, т.е. $O(n^2)$. Но можно ли учесть тот факт, что множество значений ключей ограничено, и улучшить время выполнения программы? Ответ положительный: приведенный ниже алгоритм, являющийся обобщением "карманной" сортировки, даже на множестве значений ключей вида 1, 2, ..., n^k для некоторого фиксированного k выполняет сортировку элементов за время O(n).

Пример. Рассмотрим задачу сортировки n целых чисел из интервала от 0 до n^2-1. Выполним сортировку этих чисел в два этапа. На первом этапе используем n "карманов", индексированных целыми числами от 0 до n-1. Поместим каждое сортируемое число i в список "кармана" с номером i mod n. Но в отличие от процедуры листинга каждый новый элемент помещается не в начало списка, а в его конец, и это очень важное отличие. Заметим, что если хотим эффективно выполнить вставку элементов в конец списков, то надо для представления списков использовать связанные списки, а также указатели на концы списков.

Например, пусть n = 10, а список сортируемых чисел состоит из точных квадратов от 0^2 до 9^2, расположенных в случайном порядке 36, 9, 0, 25, 1, 49, 64, 16, 81, 4. В этом случае номер "кармана" для числа i равен самой правой цифре в десятичной записи числа i. В таблице а показано размещение сортируемых чисел по спискам "карманов". Отметим, что числа появляются в списках в том же порядке, в каком они расположены в исходном списке, например в списке "кармана" 6 будут располагаться числа 36, 16, а не 16, 36, поскольку в исходном списке число 36 предшествует числу 16.

Теперь содержимое "карманов" поочередно объединим в один список:

0, 1, 81, 64, 4, 25, 36, 16, 9, 49 (*)

Если в качестве структуры данных используются связанные списки, а также указатели на концы списков, то операции конкатенации n элементов, содержащихся в карманах, займет время порядка O(n).

Целые числа из списка, созданного путем конкатенации на предыдущем этапе, снова распределяются по карманам, но уже по другому

принципу. Теперь число i помещается в карман с номером i div n, т.е. номер кармана равен наибольшему целому числу, равному или меньшему i/n. На этом этапе добавляемые в "карманы" числа также вставляются в концы списков. После распределения списков по "карманам" снова выполняется операция конкатенации всех списков в один список. В результате получим отсортированный список элементов.

В таблице б показан список (*), распределенный по "карманам" в соответствии с правилом, когда число i вставляется в "карман" с номером i div n.

Таблица. Двухэтапная "карманная" сортировка

"Карман"	Содержимое	"Карман"	Содержимое
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	64, 4	4	49
5		5	
6	25	6	64
7	36, 16	7	
8		8	81
9	9, 49	9	
а		б	

Чтобы увидеть, почему этот алгоритм работает правильно, достаточно заметить, что когда числа помещаются в один "карман", например числа 0, 1, 4 и 9 - в карман 0, то они будут располагаться в возрастающем порядке, поскольку в списке (*) они упорядочены по самой правой цифре. Следовательно, в любом "кармане" числа также будут упорядочены по самой правой цифре. И, конечно, распределение чисел на втором этапе по "карманам" в соответствии с первой цифрой гарантирует, что в конечном объединенном списке все числа будут расставлены в возрастающем порядке.

Покажем правильность работы описанного алгоритма в общем случае, предполагая, что сортируемые числа являются произвольными двузначными целыми числами из интервала от 0 до $п^2$-1. Рассмотрим целые числа i = an + b и j == cn +d, где все числа a, b, c и d из интервала от 1 до n - 1. Пусть $i < j$. Тогда неравенство $a>c$ невозможно, поэтому $a \le c$. Если $a < c$, тогда на втором этапе сортировки число i будет находиться в "кармане" с меньшим номером, чем номер "кармана", в котором находится число j, и, следовательно, в конечном списке число i будет предшествовать числу j. Если a = c, то число b должно быть меньше, чем d. Тогда в объединенном списке после первого этапа сортировки число i будет

предшествовать числу j, поскольку число i находилось в "кармане" с номером b, а число j — в кармане с номером d. На втором этапе числа i и j будут находиться в одном "кармане" (так как a=c), но число i будет вставлено в список этого "кармана" раньше, чем число j. Поэтому и в конечном списке число i будет предшествовать числу j.

5.7. Общая поразрядная сортировка

Предположим, что тип данных ключей keytype является записями со следующими полями:

```
type
keytype = record
day: 1..31;
month: (jan, ..., dec);
year: 2000..2099 end;                                    (*)
```

либо массивом элементов какого-нибудь типа, например

```
type
keytype = array[1..10] of char;                          (**)
```

Далее будем предполагать, что данные типа keytype состоят из k компонент f_1, f_2, …, f_k типа t_1, t_2, ..., t_k. Например, в (*) t_1 = 1..31, t_2 = (jan,…,dec), а t_3 = 2000..2099. В (**) k = 10, а $t_1 = t_2 = \ldots = t_k ==$ char.

Предположим также, что хотим сортировать записи в лексикографическом порядке их ключей. В соответствии с этим порядком ключевое значение (a_1, a_2, …, a_k) меньше ключевого значения (b_1, b_2, …, b_k), где a_i, b_{2i}, — значения из поля f_i (i = 1, 2, …, k), если выполняется одно из следующих условий:

1. $a_1 < b_1$, или
2. $a_1 = b_1$ и $a_2 < b_2$ или

k. $a_1 = b_1$, $a_2 = b_2$, …, $a_{k-1} = b_{k-1}$ и $a_k < b_k$

Другими словами, ключевое значение (a_1, a_2, …, a_k) меньше ключевого значения (b_1, b_2, …, b_k), если существует такой номер j ($1 \le j \le k$-1), что $a_1 = b_1$, $a_2 = b_2$, …, $a_j = b_j$ и $a_j < b_j$.

Если принять сделанные выше определения ключей, то в этом случае значения ключей можно рассматривать как целочисленные выражения в некоторой системе счисления. Например, определение (**), где каждое поле ключа содержит какой-либо символ, позволяет считать эти символы целочисленными выражениями по основанию 128 (если использовать только латинские буквы) или, в зависимости от используемого набора

символов, по какому-то другому основанию. В определении типов (*) значения поля year (год) можно считать целыми числами по основанию 100 (так как здесь значения поля изменяются от 2000 до 2099, т.е. могут изменяться только последние две цифры), значения поля month (месяц) очевидно можно интерпретировать как целочисленные выражения по основанию 12, а значения третьего поля day (день) — как целочисленные выражения по основанию 31. Предельный случай такого подхода: любые целые числа (из конечного фиксированного множества) рассматриваются как массивы цифр по основанию 2 или другому подходящему основанию. Обобщение "карманной" сортировки, использующее такое видение значений ключевых полей, называется поразрядной сортировкой.

Основная идея поразрядной сортировки заключается в следующем: сначала проводится "карманная" сортировка всех записей по полю f_k (как бы по "наименьшей значащей цифре"), затем отсортированные по "карманам" записи объединяются (наименьшие значения идут первыми), затем к полученному объединенному списку применяется "карманная" сортировка по полю f_{k-1}, снова проводится конкатенация содержимого "карманов", к объединенному списку применяется "карманная" сортировка по полю f_{k-2}, и т.д. Так же, как в примере с "карманной" сортировки с большим распределением ключей, при сортировке записей по "карманам" новая запись, вставляемая в карман, присоединяется в конец списка элементов этого "кармана", а не в начало, как это было в "чистой карманной" сортировке. После выполнения "карманной" сортировки по ключам f_k, f_{k-1}, f_{k-2}, …, f_1 и последнего объединения записей получим их результирующий список, где все записи будут упорядочены в лексикографическом порядке в соответствии с этими ключами. Эскиз описанного алгоритма в виде процедуры radixsort (поразрядная сортировка) приведен в листинге. Для обоснования этого алгоритма можно применить методику, показанную в примере м большим распределением ключей.

Листинг. Поразрядная сортировка

```
procedure radixsort;
{ radixsort сортирует список A из n записей по ключевым полям
f1, f2, …, fk типов t1, t2, …, tk соответственно.
В качестве "карманов" используются массивы Bi типа
array [ti] of listtype, 1 ≤ i ≤ k, где
listtype —тип данных связанных списков записей }
begin
for 2:= k downto 1 do begin
for для каждого значения v типа ti do
{ очистка "карманов" }
сделать Bj.[v] пустым;
```

```
for для каждой записи r из списка A do
переместить запись r в конец списка "кармана" Bj.[v],
где v — значение ключевого поля fi записи r;
for для каждого значения v типа ti
в порядке возрастания v do
конкатенация Bi [v] в конец списка A
end
end; { radixsort }
```

Анализ поразрядной сортировки

Предполагаем, что для эффективного выполнения поразрядной сортировка применяются подходящие структуры данных. В частности, предполагаем, список сортируемых элементов представлен в виде связанного списка, а не массива. Но на практике можно использовать и массив, если добавить еще поле связи типа recordtype для связывания элементов A[i] и A[i + 1] для i=1, 2, …,n-1. Таким способом можно создать связанный список в массиве A за время порядка O(n). Отметим также, что при таком представлении элементов нет необходимости в их копировании в процессе выполнения алгоритма, так как для перемещения записей из одного списка в другой достаточно изменить значения поля связи.

Как и ранее, для быстрого выполнения операции конкатенации будем спользовать указатели на конец каждого списка. Тогда цикл в строках (2), (3) листинга выполняется за время $O(s_i)$, где s_i— число различных значений типа t_i. Цикл строк (4), (5) требует времени порядка O(n), цикл строк (6), (7) - $O(s_i)$. Таким образом, общее время выполнения поразрядной сортировки составит

$\sum_{i=1}^{k} O(s_i + n)$, т.е. $O\left(kn + \sum_{i=1}^{k} s_i\right)$, или, если k константа, $O\left(n + \sum_{i=1}^{k} s_i\right)$.

Пример. Если ключи являются целыми числами из интервала от 0 до n^k-1 для некоторой константы k, то можно обобщить пример с большим распределением ключей и рассматривать ключи как целые числа по основанию n, состоящие из не более чем k "цифр". Тогда для всех i, $1 \le i \le k$, t_i — целые числа из интервала от 0 до n-1 и $s_i = n$. В этом случае выражение $O\left(n + \sum_{i=1}^{k} s_i\right)$ примет вид O(n+kn), которое имеет порядок O(n), поскольку k — константа.

Другой пример: если ключи являются символьными строками фиксированной длины k, тогда для всех i $s_i = 128$ (например) и $\sum_{i=1}^{k} s_i$ также будет константой. Таким образом, алгоритм поразрядной сортировки по символьным строкам фиксированной длины выполняется за время O(n). Фактически, если k — константа и все s_i являются константами (или если даже имеют порядок роста O(n)), то в этом случае поразрядная сортировка выполняется за время порядка O(n). Но если k растет вместе с ростом n, то время выполнения поразрядной сортировки может отличаться от O(n).

Например, если ключи являются двоичными числами длины log n, тогда k == log n и s_i = 2 для всех i. В таком случае время выполнения алгоритма составит O(n log n).

5.8. Время выполнения сортировок сравнениями

Существует "общеизвестная теорема" (из математического фольклора), что для сортировки n элементов "требуется времени n log n". В предыдущем разделе показано, это утверждение не всегда верно: если тип ключей такой, что значения ключей выбираются из конечного (по количеству элементов) множества (это позволяет с "пользой" применить метод "карманной" или поразрядной сортировки), тогда для сортировки достаточно времени порядка O(n). Однако в общем случае нельзя утверждать, что ключи принимают значения из конечного множества. Поэтому при проведении сортировки можно опираться только на операцию сравнения двух значений ключей, в результате которой можно сказать, что одно ключевое значение меньше другого.

Во всех алгоритмах сортировки, которые рассмотренных ранее, истинный порядок элементов определялся с помощью последовательных сравнений значений двух ключей, затем, в зависимости от результата сравнения, выполнение алгоритма шло по одному из двух возможных путей. В противоположность этому, алгоритм, подобный показанному в примере предыдущего раздела, за один шаг распределяет n элементов с целочисленными значениями ключей по n "карманам", причем номер "кармана" определяется значением ключа. Все программы предыдущего раздела используют мощное средство (по сравнению с простым сравнением значений) языков программирования, позволяющее за один шаг находить в массиве по индексу местоположение нужной ячейки. Но это мощное средство невозможно применить, если тип данных ключей соответствует, например, действительным числам. В языке Pascal и в большинстве других языков программирования нельзя объявить индексы массива как действительные числа. И если даже сможем это сделать, то операцию конкатенации "карманов", номера который являются машинно-представимыми действительными числами, невозможно будет выполнить за приемлемое время.

Деревья решений

Сосредоточим свое внимание на алгоритмах сортировки, которые в процессе сортировки используют только сравнения значений двух ключей. Можно нарисовать двоичное дерево, в котором узлы будут соответствовать "состоянию" программы после определенного количества сделанных сравнений ключей. Узлы дерева можно также рассматривать как соотвествие некоторым начальным упорядочиваниям данных, которые "привели" программу сортировки к такому "состоянию". Можно также

сказать, что данное программное "состояние" соответствует нашим знаниям о первоначальном упорядочивании исходных данных, которые можно получить на данном этапе выполнения программы.

Если какой-либо узел соответствует нескольким упорядочиваниям исходных данных, то программа на этом этапе не может точно определить истинный порядок данных и поэтому необходимо выполнить следующее сравнение ключей, например, "является ли $k_1<k_2$?". В зависимости от возможных ответов на этот вопрос создаются два сына этого узла. Левый сын соответствует случаю, когда значения ключей удовлетворяют неравенству $k_1<k_2$; правый сын соответствует упорядочению при выполнении неравенства $k_1>k_2$ (при предположении, что нет повторяющихся ключей). Таким образом, каждый сын "содержит" информацию, доступную родителю, плюс сведения, вытекающие из ответа на вопрос, "является ли $k_1<k_2$?"

Размер дерева решений

Если сортируется список из n элементов, тогда существует n! == $1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$ возможных исходов, которые соответствуют различным упорядочениям исходного списка элементов a_1, a_2, …, a_n. В самом деле, на первую позицию можно поставить любой из n элементов, на вторую — любой из оставшихся n - 1 элементов, на третью — любой из n - 2 элементов и т.д., произведение количества всех этих частных исходов дает n!. Таким образом, любое дерево решений, описывающее алгоритм сортировки списка из n элементов, должно иметь не менее n! листьев. Фактически, если удалить узлы и листья, соответствующие невозможным сочетаниям элементов, получим в точности n! листьев.

Двоичные деревья, имеющие много листьев, должны иметь и длинные пути. Длина пути от корня к листу равна количеству сравнений, которые необходимо сделать, чтобы получить то упорядочение, которое соответствует данному листу, исходя из определенного начального списка элементов L. Поэтому длина самого длинного пути от корня к листу — это нижняя граница количества шагов (количество сравнений ключей), выполняемых алгоритмом сортировки в самом худшем случае. На также не надо забывать, что, кроме сравнения ключей, алгоритм выполняет и другие операции.

Рассмотрим, какова же может быть длина путей в двоичном дереве с k листьями. Двоичное дерево, в котором все пути имеют длину p или меньше, может иметь 1 корень, 2 узла на первом уровне, 4 узла на втором уровне и далее 2^i узлов на уровне i. Поэтому наибольшее число листьев на 2^p уровне p, если нет узлов выше этого уровня. Отсюда следует, что двоичное дерево с k листьями должно иметь путь с длиной не менее log k. Если положить k=n!, то получим, что любой алгоритм сортировки, использующий для упорядочивания n элементов только сравнения

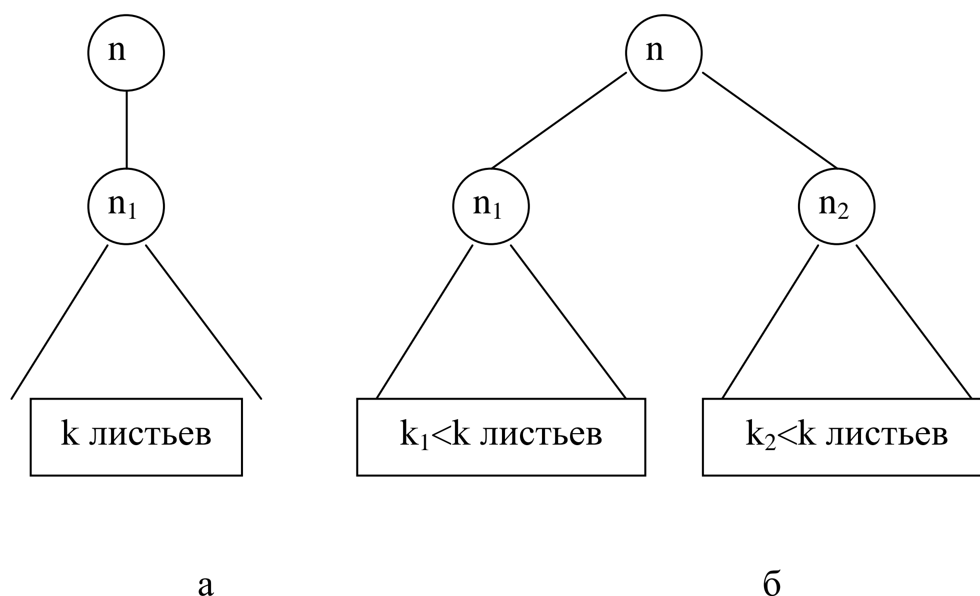
значений ключей, в самом худшем случае должен выполняться за время, не меньшее $\Omega(\log(n!))$

Какова степень роста величины $\log(n!)$? По формуле Стирлинга $n!$ можно достаточно точно аппроксимировать функцией $(n/e)^n$, где $e = 2.7183\ldots$ — основание натурального логарифма. Поскольку $\log((n/e)^n) = n \log n - n \log e = n \log n - 1.44n$, то отсюда получаем, что $\log(n!)$ имеет порядок $n \log n$. Можно получить более простую нижнюю границу, если заметить, что $n!$ является произведением не менее $n/2$ сомножителей, каждое из которых не превышает $n/2$. Поэтому $n! \geq (n/2)^{n/2}$. Следовательно, $\log(n!) \geq (n/2)\log(n/2) = (n/2)\log n - n/2$. Из любой приведенной оценки вытекает, что сортировка посредством сравнений требует в самом худшем случае времени порядка $\Omega(n \log n)$.

Анализ времени выполнения в среднем

Можно ожидать, что алгоритм, который в самом худшем случае выполняет за время $O(n \log n)$, в среднем будет иметь время выполнения порядка $O(n)$ или, по крайней мере, меньшее, чем $O(n \log n)$. Но это не так, покажем, оставляя детали доказательства читателю в качестве упражнения.

Надо доказать, что в произвольном двоичном дереве с k листьями средняя глубина листьев не менее $\log k$. Предположим, что это не так, и попробуем построить контрпример двоичного дерева T с k листьями, у которого средняя глубина листьев была бы меньше $\log k$. Поскольку дерево T не может состоять только из одного узла (в этом случае невозможен контрпример, так как утверждение выполняется — средняя глубина равна 0), пусть $k \geq 2$. Возможны две формы двоичного дерева T, которые показаны на рис.42.



а б

Рисунок 42 – Возможные формы двоичного дерева.

Дерево на рис. 42а не может быть контрпримером с минимальной средней глубиной листьев, поскольку дерево с корнем в узле n_1 имеет столько же листьев, что и дерево T, но его средняя глубина меньше. Дерево на рис.42, б не нарушает требования к контрпримеру. Пусть средняя глубина листьев поддерева T_1, составляет $\log k_1$, а средняя глубина листьев дерева T_2-$\log k_2$. Тогда средняя глубина листьев дерева T равна

$$\left(\frac{k_1}{k_1+k_2}\right)\log(k_1)+\left(\frac{k_2}{k_1+k_2}\right)\log(k_2)+1.$$

Поскольку $k_1+k_2=k$, то последнее выражение можно переписать так:

$$\frac{1}{k}\left(k_1\log(2k_1)+k_2\log(2k_2)\right). \qquad (*)$$

Легко проверить, что при $k_1 = k_2 = k/2$ выражение (*) принимает значение logk. Теперь читатель должен показать, что выражение (*) при выполнении условия $k_1 + k_2 = k$ имеет минимум, когда $k_1 = k_2$. Так как минимум выражения (*) равен logk, то можно считать доказанным, что контрпример дерева T со средней глубиной листьев, меньшей чем logk, не существует.

5.9. Порядковые статистики

Задача вычисления *порядковых статистик* заключается в следующем: дан список из n записей и целое число k, необходимо найти ключ записи, которая стоит в списке, отсортированном в возрастающем порядке, в k-й позиции. Для краткости эту задачу будем называть "задачей нахождения k-й порядковой статистики". Специальные случаи этой задачи возникают при k = 1 (нахождение минимального элемента), k = n (нахождение максимального элемента) и, если n нечетно, k = (n+1)/2 (нахождение медианы).

В некоторых случаях решение задачи вычисления порядковых статистик находится за линейное время. Например, нахождение минимального элемента (как и максимального) требует времени порядка O(n). Как упоминалось при изучений пирамидальной сортировки, если $k \le n/\log n$, тогда для нахождения k-й порядковой статистики можно построить кучу (за время порядка O(n)) и затем выбрать из нее k наименьших элементов за время $O(n + k \log n) = O(n)$. Подобным образом при $k \ge n - n/\log n$ также можно найти k-ю порядковую статистику за время O(n).

5.10. Вариант быстрой сортировки

По-видимому, самым быстрым в среднем способом нахождения k-й порядковой статистики является рекурсивное применение процедуры, основанной на методе быстрой сортировки. Назовем эту процедуру select

(выбор). Процедура select(i, j, k) находит k-й элемент среди элементов A[i], …, A[j], которые взяты из некоторого большого массива A[1], ..., A[n]. Процедура select выполняет следующие действия.

1. Назначает опорный элемент, скажем v.

2. Используя процедуру partition из листинга быстрой сортировки, разделяет элементы A[i], …, A[j] на две группы. В первой группе A[i], ..., A[m - 1] значения ключей всех записей меньше v, во второй группе A[m], …, A[j] — равны или больше u.

3. Если $k \leq m - i$, тогда k-й элемент находится в первой группе и к ней снова применяется процедура select(i, m - 1, k). Если $k > m - i$, тогда вызывается процедура select(m, j, k - m + i).

Рано или поздно вызов процедуры select(i, j, k) будет сделан для элементов A[i], ..., A[j], имеющих одинаковые значения ключей (и поэтому скорее всего будет i = j). Значение ключа этих элементов и будет искомым значением k-й порядковой статистики.

Так же, как и метод быстрой сортировки, процедура select (так как она была описана ранее) в самом худшем случае потребует времени не менее $\Omega(n^2)$. Например, если при поиске минимального элемента в качестве опорного элемента всегда брать наибольшее из возможных значений ключей, то получим для этой процедуры время выполнения порядка $O(n^2)$. Но в среднем процедура select работает значительно быстрее, чем алгоритм быстрой сортировки, в среднем процедура select в аналогичной ситуации вызывается только один раз. Можно провести анализ процедуры select теми же методами, которые применялись при анализе алгоритма быстрой сортировки, но чтобы избежать сложных математических выкладок, здесь ограничиваемся только интуитивными соображениями, показывающими среднее время выполнения процедуры select. Эта процедура повторно вызывает себя на подмножество, которое является только частью того множества, для которого она вызывалась на предыдущем шаге. Сделаем умеренное предположение, что каждый вызов этой процедуры осуществляется на множестве элементов, которое составляет 9/10 того множества, для которого был произведен предыдущий вызов процедуры. Тогда, если обозначить через T(n) время, затрачиваемое процедурой select на множестве из n элементов, для некоторой константы c будем иметь

$$T(n) \leq T\left(\frac{9}{10}n\right) + cn$$

Используя технику из следующей главы, можно показать, что решением неравенства будет T(n) = O(n).

5.11. Линейный метод нахождения порядковых статистик

Чтобы гарантировать для процедуры, подобной select, в самом

худшем случае время выполнения порядка O(n), необходимо показать, что за линейное время можно выбрать такой опорный элемент, который разбивает множество элементов на два подмножества, размер которых не больше некоторой фиксированной доли исходного множества. Например, решение неравенства показывает, если опорный элемент не меньше (n/10)-го порядкового элемента и не больше (9n/10)-го порядкового элемента, тогда множество, для которого вызывается процедура select, разбивается на подмножества, не превышающие 9/10 исходного множества, и это гарантирует линейное время в самом худшем случае.

Нахождение "хорошего" опорного элемента можно осуществить посредством следующих двух шагов.

1. Выполняется разделение n элементов на группы по 5 элементов, оставляя в стороне группу из 1 - 4 элементов, не вошедших в другие группы. Каждая группа из 5 элементов сортируется любым алгоритмом в порядке возрастания и берутся средние элементы из каждой группы. Всего таких средних элементов будет n div 5.
2. Используя процедуру select, находится медиана этих средних элементов. Если n div 5 — четно, то берется элемент, наиболее близкий к середине. В любом случае будет найден элемент, стоящий в позиции (n + 5) div 10 отсортированного списка средних элементов.

Если среди записей не слишком много таких, значения ключей которых совпадают со значением опорного элемента, тогда значение опорного элемента достаточно далеко от крайних значений ключей. Для простоты временно положим, что все значения ключей различны (случай одинаковых ключевых значений будет рассмотрен ниже). Покажем, что выбранный опорный элемент (являющийся (n + 5) div 10 порядковым элементом среди n div5 средних элементов) больше не менее 3*((n - 5) div 10) элементов из n исходных элементов. В самом деле, опорный элемент больше (n - 5) div 20 средних элементов. В свою очередь каждый из этих средних элементов больше двух элементов из той пятерки элементов, которой он принадлежит. Отсюда получаем величину 3* ((n-5) div 10). Если n≥75, тогда 3*((n - 5) div 10) не меньше n/4. Подобным образом доказывается, что выбранный опорный элемент меньше или равен 3*((n - 5) div 10) элементов. Отсюда следует, что при n≥75 выбранный опорный элемент находится между первой и последней четвертями отсортированного списка исходных элементов.

Алгоритм нахождения k-й порядковой статистики в виде функции select показан в листинге. Как и в алгоритмах сортировки, предполагаем, что список исходных элементов представлен в виде массива A записей типа recordtype и что записи имеют поле key типа keytype. Нахождение k-й порядковой статистики выполняется, естественно, посредством вызова функции select (1, n, k).

Листинг. Линейный алгоритм нахождения k-й порядковой статистики

```
function select {i, j, k: integer ) : keytype;
{ Функция возвращает значение ключа k-го элемента
из A[i], …, A[j] }
var
m: integer; { используется в качестве индекса }
begin
if j - 1 < 74 then begin
{ элементов мало, select рекурсивно не применяется }
сортировка A[i] , …, A[j] простым алгоритмом;
return(A[i + k - 1] .key)
end
else begin { select применяется рекурсивно }
for m:= 0 to (j - i - 4) div 5 do
{ нахождение средних элементов в группах из 5-ти элементов }
нахождение 3-го по величине элемента в группе
A[i + 5*m], …, A[i + 5*m + 4] и
перестановка его с A[i + m];
pivot:= select(i, i+(j-i-4) div 5, (j-i-4) div 10);
{ нахождение медианы средних элементов }
m:= partition(i, j, pivot);
if k <= m - i then
return(select(i, m - 1, k))
else
return(select(m, j, k - (m - i)))
end
end; { select }
```

Для анализа времени выполнения процедуры select из листинга положим здесь, что j - i + 1 = n. Строки (2), (3) выполняются, если n не превосходит 74. этому, если даже выполнение строки (2) требует времени порядка $O(n^2)$ в общем случае, здесь потребуется конечное время, не превосходящее некоторой константы с. Таким образом, при $n \leq 74$ строки (1) - (3) требуют времени, не превосходящего некоторой константы c_1.

Теперь рассмотрим строки (4) - (10). Строка (7), разбиение множества элементов по опорному элементу, имеет время выполнения O(n), как было показано при анализе алгоритма быстрой сортировки. Тело цикла (4), (5) выполняется n/5 раз, упорядочивая за каждую итерацию 5 элементов, на что требуется фиксированное время. Поэтому данный цикл выполняется за время порядка O(n).

Обозначим через T(n) время выполнения процедуры select на n элементах. Тогда строка (6) требует времени, не превышающего T(n/5). Поскольку строку (10) можно достигнуть только для n≥75, а, как было показано ранее, при n≥75 количество элементов, меньших опорного элемента, не превышает 3n/4. Аналогично, количество элементов, больших или равных опорному элементу, не превышает 3n/4. Следовательно, время выполнения строки (10) не превышает T(3n/4). Отсюда вытекает, что для некоторых констант c_1 и c_2 справедливы неравенства:

$$T(n) \le \begin{cases} c_1, & \textit{если} \quad n \le 74, \\ c_2 + T(n/5) + T(3n/4), & \textit{если} \quad n \ge 74. \end{cases} \qquad (*)$$

В неравенстве выражение c_2n представляет время выполнения строк (1), (4), (5) и (7), выражение T(n/5) соотвествует времени выполнения строки (6), а T(3n/4) — строк (9), (10).

Далее покажем, что T(n) из (*) имеет порядок O(n). Но сначала слов о "магическом числе" 5, размере групп в строке (5) и условии n ≤ 75, которое определяет способ нахождения порядковой статистики (посредством рекурсивного повторения функции select или прямым способом). Конечно, эти числа могут быть и другими, но в данном случае они выбраны так, чтобы в формуле (*) выполнялось неравенство 1/5 + 3/4 < 1, необходимое для доказательства линейного порядка величины T(n).

Неравенства (*) можно решить методом индукции по n. Предполагаем, что решение имеет вид cn для некоторой константы c. Если принять $c \ge c_1$, тогда T(n)≤cn для всех n от 1 до 74, поэтому рассмотрим случай, когда n ≥ 75. Пусть, в соответствии с методом математической индукции, T(m) ≤ cm для всех m, m < n. Тогда из (*) следует, что

$$T(n) \le c_2n + cn/5 + 3cn/4 \le c_2n + 19cn/20. \qquad (**)$$

Если положить $c = \max(c_1, 20c_2)$, тогда из (**) получаем T(n) ≤ cn/20 + cn/5 + 3cn/4 = cn, что и требовалось доказать. Таким образом, T(n) имеет порядок O(n).

Случай равенства некоторых значений ключей

Напомним, что при создании процедуры листинга нахождения k-й порядковой статистики было предположение, что значения ключей различны. Это сделано из-за того, что в противном случае нельзя доказать, что множество элементов при разбиении распадется на подмножества, не превышающие 3n/4. Для возможного случая равенства значений ключей необходимо только немного изменить функцию select из листинга. После строки (7), выполняющей разбиение множества, надо добавить операторы, собирающие вместе записи, имеющие значения ключей, совпадающих с опорным элементом. Пусть таких ключей будет p ≥ 1. Если m-j≤k≤m-i+p,

тогда в последующем рекурсивном вызове функции select нет необходимости — достаточно вернуть значение A[m].key. В противном случае строка (8) остается без изменений, а в строке (10) осуществляется вызов select(m + p. j, k - (m - i) - p).

5.12. Краткое описание наиболее распространных сортировок

Метод выбора

Упорядоченный массив располагается на том же самом месте в памяти, что и исходный. Во время первого прохода осуществляется поиск наименьшего элемента. После того как элемент найден, его меняют местами с первым элементом исходной последовательности, в результате чего наименьший элемент занимает первую позицию. Затем ищется следующий наименьший элемент среди оставшихся. Ставится на второе место. Поиск элементов со следующими наименьшими значениями ключа и помещение их в соответствующие позиции продолжается до тех пор, пока все элементы не будут отсортированы в восходящем порядке. Если N элементов, то N-1 проходов, т.к. в ∀ проходе в соответствующую позицию упорядоченной последовательности заносится только один элемент. Для i прохода требуется N-i сравнений. Общее число сравнений равно: $C = \sum_{i=1}^{N-1} (N-i) = \frac{N}{2}(N-1)$. Максимальное число перестановок равно N-1, в лучшем случае ни одной, среднее число перестановок пропорционально N/2.

Метод обмена (метод «пузырька»)

Упорядоченная последовательность находится на том же самом месте, что и исходная. В процессе сортировки производится попарное сравнение соседних элементов. Если порядок следования элементов нарушен, они меняются местами. В течение первого прохода сравниваются элементы A_1 и A_2, если $A_2 < A_1$, то они меняются местами. Этот процесс повторяется для пар элементов A_2 и A_3; A_3 и A_4 и т.д.

После первого прохода наибольший элемент займет N-ю позицию, а минимальный всплывает. В каждом следующем проходе очередные наибольшие элементы занимают позиции N-1, N-2... После ∀ прохода может быть сделана проверка, были ли совершены перестановки в течение данного прохода. Если перестановок не было, то последовательность упорядочена и дальнейших проходов не требуется.

Число сравнений зависит от числа проходов, необходимых для сортировки. В худшем случае, когда последовательность имеет обратную упорядоченность, в течение ∀i прохода выполняются перестановки, а число проходов равно N-1. При этом для сортировки требуется максимальное число сравнений (N-1) + (N-2) + (N-3) + ... + 1, т.е.

$C=\sum_{i=1}^{N-1}(N-i)=\frac{N}{2}(N-1)$. В лучшем случае, когда последовательность уже упорядочена, потребуется всего один проход и N-1 сравнение. Число обменов зависит от упорядоченности исходной последовательности. В max – 0,5 N(N-1).

Метод вставок

Упорядоченная последовательность создается на свободном участке памяти. Под сортировку выделяется объем памяти, равный длине отсортированного массива записей. А – исходная; В – упорядоченная.

1. A_1 становится первым элементом В.
2. A_2 сравнивается с B_1, если $A_2 < B_1$, то B_1 сдвигается на одну позицию, а A_2 занимает его место.
3. В каждом i проходе процесса сортировки элемент Ai сравнивается поочередно со всеми элементами В, начиная с B_1. При обнаружении $B_j>A_j$ элементы B_j, B_{j+1}, B_{j+2},..., B_{j-1} передвигаются на одну позицию, освобождая место для элемента A_j, который занимает j позицию.

В худшем случае i-1 сравнение для ∀i прохода. Худшим оказывается случай, когда последовательность отсортирована в нужном порядке. Максимальное число сравнений ⇒Σ арифметической прогрессии.

1 + 2 + 3 +...+(N-1)

$$C_{\max}=\sum_{i=1}^{N-1}(N-1)=0{,}5N(N-1)$$

$C_{\min}$ = N-1, если последовательность имеет обратную упорядоченность.

Среднее число сравнений ≈0,25 N^2

Среднее число перестановок ≈ 0,25 N^2

Метод подсчета

1. В свободной памяти
2. К+1 элемент упорядоченной последовательности превышает ровно К элементов и занимает К+1 позицию.

В процессе сортировки на каждом i-м проходе i элемент исходной последовательности попарно сравнивается со всеми остальными элементами, если $A_i>A_j$ то значение К увеличивается на единицу. По окончании прохода значение К равно числу элементов, меньших, чем j. Номер позиции i-го элемента в последовательности В будет равен К+1.

Для сортировки последовательности из N элементов требуется N проходов, в любом проходе выполняется N сравнений., т.е. в ∀ случае N^2 сравнений. Используется, если в исходной последовательности нет одинаковых элементов. В противном случае алгоритм необходимо

модифицировать.

Метод сортировки Шелла

Метод сортировки Шелла использует сравнения и перестановки элементов. Для проведения сортировки последовательность из N элементов делится на N/2 групп или на (n-1)/2, если N нечетно.

Каждая группа содержит по два элемента. Если число элементов нечетно, то одна часть будет содержать три элемента. Элементы, принадлежащие одной группе, отстают на N/2 позиций. Это расстояние называется шагом.

В течение первого прохода осуществляется упорядочивание элементов любой группы методом вставок. Для осуществления ∀ следующего прохода устанавливается шаг, равный половине предыдущего шага. (у дробной части берется целая часть). Для сортировки последовательности из N элементов требуется $\log_2 N$ проходов. Число сравнений зависит от шага. Можно брать следующую последовательность шагов: N/2, N/4, N/8 и т.д.

Приблизительное число сравнений равно $N \log_2 N$.

6. АЛГОРИТМЫ ДЛЯ ВНЕШНЕЙ ПАМЯТИ. ПРИКЛАДНЫЕ АЛГОРИТМЫ

Начнем главу с обсуждения различий в характеристиках доступа к устройствам основной (оперативной) и внешней памяти (например, дисководов). Затем представим несколько алгоритмов сортировки файлов данных, хранящихся на устройствах внешней памяти. Завершается глава обсуждением структур данных и алгоритмов, таких как индексированные файлы и В-деревья, которые хорошо подходят для хранения и поиска информации на вторичных устройствах памяти.

6.1. Модель внешних вычислений

В алгоритмах, которые обсуждали до сих пор, предполагалось, что объем входных данных позволяет обходиться исключительно основной (оперативной) памятью. Но как быть, если нам нужно, например, отсортировать всех государственных служащих по продолжительности их рабочего стажа или хранить информацию из налоговых деклараций всех граждан страны? Когда возникает необходимость решать подобные задачи, объем обрабатываемых данных намного превышает возможности основной памяти. В большинстве компьютерных систем предусмотрены устройства внешней памяти, такие как жесткие диски, или запоминающие устройства большой емкости, на которых можно хранить огромные объемы данных. Однако характеристики доступа к таким устройствам внешней памяти существенно отличаются от характеристик доступа к основной памяти. Чтобы повысить эффективность использования этих устройств, был разработан ряд структур данных и алгоритмов. В этой главе обсудим структуры данных и алгоритмы для сортировки и поиска информации, хранящейся на вторичных устройствах памяти.

В Pascal и некоторых других языках программирования предусмотрен файловый тип данных, предназначенный для представления данных, хранящихся во вторичной памяти. Даже если в языке, которым вы пользуетесь, файловый тип данных не предусмотрен, в операционной системе понятие "внешних" файлов, несомненно, поддерживается. О каких бы файлах ни говорили (файлах, предусмотренных в Pascal, или файлах, поддерживаемых непосредственно операционной системой), в любом случае нам придется действовать в рамках ограничений, касающихся способов доступа к файлам. Операционная система делит вторичную память на блоки одинакового размера. Размер блока зависит от конкретного типа операционной системы и обычно находится в пределах от 512 до 4096 байт.

Файл можно рассматривать как связанный список блоков, хотя чаще всего операционная система использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а

каждый внутренний узел содержит указатели на множество блоков файла. Если, например, 4 байт достаточно, чтобы хранить адрес блока, а длина блока составляет 4096 байт, тогда корневой блок может содержать указатели максимум на 1024 блока. Таким образом, файлы, состоящие максимум из 1024 блоков (т.е. примерно четырех миллионов байт), можно представить одним корневым блоком и блоками, содержащими сам файл. Файлы, состоящие из максимум 2^{20} блоков, или 2^{32} байт, можно представить одним корневым блоком, указывающим на 1024 блока промежуточного уровня, каждый из которых указывает на 1024 блока-листа, содержащих определенную часть файла, и т.д.

Базовой операцией, выполняемой по отношению к файлам, является перенос одного блока в буфер, находящийся в основной памяти. Буфер представляет собой зарезервированную область в основной памяти, размер которой соответствует размеру блока. Типичная операционная система обеспечивает чтение блоков в том порядке, в каком они появляются в списке блоков, который содержит соответствующий файл, т.е. сначала читаем в буфер первый блок файла, затем заменяем его на второй блок, который записывается в тот же буфер, и т.д.

Теперь нетрудно понять концепцию, которая лежит в основе правил чтения файлов в языке Pascal. Каждый файл хранится в виде определенной последовательности блоков; каждый такой блок содержит целое число записей. (Память будет использоваться нерационально, если хранить части одной и той же записи в разных блоках). Указатель считывания всегда указывает на одну из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, настало время прочитать следующий блок файла.

Аналогично, процесс записи файла в языке Pascal можно рассматривать как процесс создания файла в буфере. Когда записи "записываются" в файл, фактический они помещаются в буфер для этого файла — непосредственно вслед за записями, которые уже находятся там. Если очередная запись не помещается в буфер целиком, содержимое буфера копируется в свободный блок вторичной памяти, который присоединяется к концу списка блоков для данного файла. После этого можно считать, что буфер свободен для помещения в него очередной порции записей.

Стоимость операций со вторичной памятью

Природа устройств вторичной памяти (например, дисководов) такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико в сравнении со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке. Допустим, например, что у нас имеется блок из 1000 целых чисел на диске,

вращающемся со скоростью 1000 об/мин. Время, которое требуется для позиционирования считывающей головки над дорожкой, содержащей этот блок (так называемое время установки головок), плюс время, затрачиваемое на ожидание, пока требуемый блок сделает оборот и окажется под головкой (время ожидания), может в среднем составлять 100 миллисекунд. Процесс записи блока в определенное место во вторичной памяти занимает примерно столько же времени. Однако за те же 100 миллисекунд машина, как правило, успевает выполнить 100 000 команд. Этого времени более чем достаточно, чтобы выполнить простую обработку тысячи целых чисел, когда они находятся в основной памяти (например, их суммирование или нахождение среди них наибольшего числа). Этого времени может даже хватить для выполнения быстрой сортировки целых чисел.

Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, нам придется, таким образом, в первую очередь учитывать количество обращений к блокам, т.е. сколько раз считываем в основную память или записываем блок во вторичную память. Такая операция называется доступом (или обращением) к блоку. Предполагается, что размер блока фиксирован в операционной системе, поэтому у нас нет возможности ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам. Таким образом, мерой качества алгоритма, работающего с внешней памятью, является количество обращений к блокам. Изучение алгоритмов, работающих с внешней памятью, начнем с рассмотрения способов внешней сортировки.

6.2. Внешняя сортировка

Сортировка данных, организованных в виде файлов, или — в более общем случае — сортировка данных, хранящихся во вторичной памяти, называется внешней сортировкой. Приступая к изучению внешней сортировки, сделаем предположение, что данные хранятся в Pascal-файле. Покажем, как алгоритм сортировки слиянием позволяет отсортировать файл с n записями всего лишь за O(log n) проходов через файл; этот показатель намного лучше, чем O(n) проходов, которые требовались алгоритмам, изучавшимся ранее. Затем рассмотрим, как использовать определенные возможности операционной системы по управлению чтением и записью блоков, что может ускорить сортировку за счет сокращения времени "бездействия" компьютера (периоды ожидания, пока блок будет прочитан в основную память или вписан из основной памяти во внешнюю).

Сортировка слиянием

Главная идея, которая лежит в основе сортировки слиянием,

заключается в том, что организуется файл в виде постепенно увеличивающихся серий, т.е. последовательностей записей $r_1, \dots, r_k$, где ключ r_i не больше, чем ключ r_{i+1}, $1 \le i \le k$. Будем говорить, что файл, состоящий из $r_1, \dots, r_m$ записей, делится на серии длиной k, если для всех $i \ge 0$, таких, что $ki \le m$ и $r_{k(i-1)+1}, r_{k(i-1)+2}, \dots, r_{ki}$, является последовательностью длиной k. Если m не делится нацело на k, т.е. $m = pk + q$, где $q<k$, тогда последовательность записей $r_{m-q+1}, r_{m-q+2}, \dots, r_m$, называемая хвостом, представляет собой серию длиной q. Например, последовательность целых чисел, показанная на рисунке 43, организована сериями длиной 3. Обратите внимание, что хвост имеет длину, меньшую 3, однако и его записи тоже отсортированы.

7 15 29	8 11 13	16 22 31	5 12

Рисунок 43 – Файл с сериями длиной 3.

Главное в сортировке файлов слиянием — начать с двух файлов, например f_1 и f_2, организованных в виде серий длиной k. Допустим, что (1) количества серий (включая хвосты) в f_1 и f_2 отличаются не больше, чем на единицу; (2) по крайней мере один из файлов f_1 или f_2 имеет хвост; (3) файл с хвостом имеет не меньше серий, чем другой файл.

В этом случае можно использовать достаточно простой процесс чтения по одной серии из файлов f_1 и f_2 , слияние этих серий и присоединения результирующей серии длиной 2k к одному из двух файлов g_1 и g_2 , организованных в виде серий длиной 2k. Переключаясь между g_1 и g_2, можно добиться того, что эти файлы будут не только организованы в виде серий длиной 2k, но будут также удовлетворять перечисленным выше условиям (1) - (3). Чтобы выяснить, выполняются ли условия (2) и (3) достаточно убедиться в том, что хвост серий f_1 и f_2 слился с последней из созданных серий (или, возможно, уже был ею).

Итак, начинаем с разделения всех n записей на два файла f_1 и f_2 (желательно, чтобы записей в этих файлах было поровну). Можно считать, что любой файл состоит из серий длины 1. Затем можно объединить серии длины 1 и распределить их по файлам g_1 и g_2, организованным в виде серий длины 2. Сделаем f_1 и f_2 пустыми и объединяем g_1 и g_2 в f_1 и f_2 , которые затем можно организовать в виде серий длины 4. Затем объединяем f_1 и f_2, создавая g_1 и g_2, организованные в виде серий длиной 8, и т.д.

После выполнения i подобного рода проходов получатся два файла, состоящие из серий длины 2^i. Если $2^i \ge n$, тогда один из этих двух файлов будет пустым, а другой будет содержать единственную серию длиной n, т.е. юудет отсортирован. Так как $2^i \ge n$ при $i \ge \log n$, то нетрудно заметить, что в этом случае будет достаточно [log n]+1 проходов. Каждый проход требует чтения и записи двух файлов, длина каждого из них равна

примерно n/2. Общее число блоков, прочитанных или записанных во время одного из проходов, составляет, таким образом, около 2n/b, где b — количество записей, умещающихся в одном блоке. Следовательно, количество операций чтения и записи блоков для всего процесса сортировки равняется O((n logn)/b), или, говоря по-другому, количество операций чтения и записи примерно такое же, какое требуется при выполнении O(log n) проходов по данным, хранящимся в единственном файле. Этот показатель является существенным улучшением в сравнении с O(n) проходами, которые требуются многим из алгоритмов сортировки, изучавшихся ранее.

В листинге показан код программы сортировки слиянием на языке Pascal. Считываем два файла, организованных в виде серий длины k, и записываем два файла, организованных в виде серий длины 2k. Предлагаем читателям, воспользовавшись изложенными выше идеями, самостоятельно разработать алгоритм сортировки файла, состоящего из n записей. В этом алгоритме должна logn раз использоваться процедура merge (слияние), представленная в листинге.

Листинг. Сортировка слиянием.

```
procedure merge ( k: integer; { длина входной серии }
fl, f2. gl, д2: file of recordtype);
var
outswitch: boolean;
{ равна true, если идет запись в g1 и false, если в g2 }
winner: integer;
{ номер файла с меньшим ключом в текущей записи }
used: array[1..2] of integer;
{ used[j] сообщает, сколько записей прочитано
к настоящему времени из текущей серии файла fj }
fin: array[1..2] of boolean;
{ fin [j]==true, если уже закончена серия из файла fj:
либо прочитано k записей, либо достигнут конец файла fj}
current: array[1..2] of recordtype;
{ текущие записи из двух файлов }

procedure getrecord ( i: integer);
{ Перемещение по файлу fi, не выходя за конец файла или
конец серии. Устанавливается fin[2]=true, если достигнут
конец серии или файла }
begin
used[i]:= used[i] + 1;
if (used[i]= k) or (i = 1) and eof(fl) or (i=2) and eof(f2) then fin [i] := true
else if i = 1 then read(f1, .current[1])
```

```
else read(f2, current[2])
end; { getrecord }

begin { merge }
outswitch:= true; {первая объединенная серия записывается в gl}
rewrite(gl); rewrite(g2);
reset(.fl); reset(f2);
while not eof(f1) or not eof(f2) do begin {слияние двух файлов}
{инициализация}
used[1]:=0; used[2]:=0;
fin[1]:=false; fin[2]:=false;
getrecord(1); getrecord(2);
while not fin[1] or not fin[2] do begin {слияние серий}
{вычисление переменной winner (победитель)}
if fin[1] then winner:=2
{ f2 "побеждает" по умолчанию: серия из f1 исчерпалась}
else if fin[2] then winner:= 1
(f1 "побеждает" по умолчанию}
else (не исчерпалась ни одна из серий}
if current[1].key < current[2].key then winner:=1
else winner:= 2;
if outswitch then write(gl, current[winner])
else write(g2, current[winner]);
getrecord(winner)
end;
{ закончено слияние двух серий, далее надо "переключить"
выходной файл и повторить процедуру }
outswitch:= not outswitch
end
end; { merge }
```

Обратите внимание, что процедура merge, показанная в листинге, вовсе не требует, чтобы отдельная серия полностью находилась в памяти: она считывает и записывает последовательно запись за записью. Именно нежелание хранить целые серии в основной памяти заставляет нас использовать два входных файла. В противном случае можно было бы читать по две серии из одного файла одновременно. В таблице приведен пример последовательности действий сортировки слиянием.

Таблица. Сортировка слиянием

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

а) исходные файлы

28	31	93	96	54	85	30	39	8	10	8	10
3	5	10	40	9	65	13	90	69	77	22	

б) серии длиной 2

3	5	28	31	9	54	65	85	8	10	69	77
10	40	93	96	13	30	39	90	8	10	22	

в) серии длиной 4

3	5	10	28	31	40	93	96	8	10	22	77
9	13	30	39	54	65	85	90	8	10	69	

г) серии длиной 8

3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96

8 8 10 10 22 69 77

д) серии длиной 16

3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

е) серии длиной 32

Ускорение сортировки слиянием

В таблице продемонстрирован пример процедуры сортировки слиянием, которая начинается с серий длины 1. Можно сэкономить немало времени, если начали эту процедуру с прохода, который считывает в основную память группы из k записей (при соответствующем k), сортирует их (например, с помощью процедуры быстрой сортировки) и записывает во внешнюю память в виде серии длиной k.

Если, например, есть миллион записей, то потребуется 20 проходов по этим данным, чтобы выполнить сортировку, начиная с серий длиной 1. Если, однако, у нас есть возможность одновременно поместить в основную память 10 000 записей, то сможем за один проход прочитать 100 групп из 10 000 записей, отсортировать каждую группу и получить таким образом 100 серий длиной 10 000, поделенных поровну между двумя файлами. Таким образом, всего семь проходов и слияний потребовалось бы для сортировки файла, содержащего не более 10 000 x 2^7 = 1 280 000 записей.

Минимизация полного времени выполнения

В современных компьютерных системах с разделением времени пользователю обычно не приходится платить за время, в течение которого его программа ожидает считывания блоков данных из файла (операция, характерная для процесса сортировки слиянием). Между тем, полное время выполнения сортировки превышает (зачастую значительно) время обработки данных, находящихся в основной памяти. Если же нам приходится сортировать действительно большие файлы, время обработки которых измеряется часами, полное время становится критической величиной, даже если не платим за него из собственного кармана, и проблема минимизации полного времени процесса сортировки слиянием выходит на первый план.

Как уже указывалось, время, необходимое для считывания данных с магнитного диска или других носителей, как правило, существенно превышает время, затрачиваемое на выполнение простых вычислений с этими данными (например, слияния списков). Таким образом, можно предположить, что при наличии лишь одного канала, по которому происходит обмен данными с основной памятью, именно этот канал и станет тем "узким местом", которое будет тормозить работу системы в целом. Этот канал обмена данными все время будет занят, и полное время работы системы будет практически равно времени, затрачиваемому на обмен данными с основной памятью, т.е. все вычисления будут выполняться практически мгновенно после того, как появятся соответствующие данные, и одновременно с тем, пока будет считываться или записываться следующая порция данных.

Даже в условиях такой относительно простой вычислительной среды следует позаботиться о минимизации затрат времени. Чтобы увидеть, что может произойти, если выполняем попеременное поблочное считывание двух входных файлов f_1 и f_2. Файлы организованы в виде серий определенной длины, намного превышающей размер блока, поэтому, чтобы объединить две такие серии, нужно прочитать несколько блоков из каждого файла. Предположим, однако, что все записи в серии из файла f_1 предшествуют всем записям из файла f_2. В этом случае при попеременном считывании блоков все блоки из файла f_2 должны оставаться в основной памяти. Основной памяти может не хватить для всех этих блоков, но даже если и хватит, придется (после считывания всех блоков серии) подождать, пока не будет скопирована и записана вся серия из файла f_2.

Чтобы избежать подобных проблем, рассматриваем ключи последних записей в последних блоках, считанных из f_1 и f_2, например ключи k_1 и k_2 соответственно. Если какая-либо из серий исчерпалась, естественно, считываем следующую серию из другого файла. Если серия не исчерпалась, считываем блок из файла f_1, если, конечно, $k_1 < k_2$ (в противном случае считываем блок из f_2). То есть определяем, у какой из двух серий будут первой выбраны все ее записи, находящиеся в данный момент в основной памяти, и в первую очередь пополняем запас записей именно для этой серии. Если выбор записей происходит быстрее, чем считывание, то известно, что когда будет считан последний блок этих двух серий, для и последующего слияния не может остаться больше двух полных блоков записей; возможно, эти записи будут распределены по трем (максимум!) блокам.

Многоканальное слияние

Если "узким местом" является обмен данными между основной и вторичной памятью, возможно, удалось бы сэкономить время за счет увеличения числа каналов обмена данными. Допустим, что в нашей

системе имеется 2m дисководов, каждый из которых имеет собственный канал доступа к основной памяти. Можно разместить на m дисководах m файлов (f_1, f_2, …, f_m), организованных в виде серий длины k. Тогда можно прочитать m серий, по одной из каждого файла, и объединит серию длиной mk. Эта серия помещается в один из m выходных файлов (g_1, g_2, … g_m), каждый из которых получает по очереди ту или иную серию.

Процесс слияния в основной памяти можно выполнить за O(log m) шагов на одну запись, если организуем m записей кандидатов, т.е. наименьших на данный момент невыбранных записей из каждого файла, в виде частично упорядоченного дерева или другой структуры данных, которая поддерживает операторы INSERT или DELETEMIN, выполняемые над очередями с приоритетами за время порядка O(log n). Чтобы выбрать из очереди с приоритетами запись с наименьшим ключом, надо выполнить оператор DELETEMIN, а затем вставить (оператор INSERT) в очередь с приоритетами следующую запись из файла-победителя в качестве замены выбранной записи.

Если у нас имеется n записей, а длина серий после каждого прохода умножается на m, тогда после i проходов серии будут иметь длину m^i. Если m^i>n, т.е. после i = $\log_m n$ проходов, весь список будет отсортирован. Так как $\log_m n = \log_2 n / \log_2 m$, то "коэффициент экономии" (по количеству считываний каждой записи) составляет $\log_2 m$. Более того, если m — количество дисководов, используемых для входных файлов, и m дисководов используются для вывода, можно обрабатывать данные в m раз быстрее, чем при наличии лишь одного дисковода для ввода и одного дисковода для вывода, и в 2m раз быстрее, чем при наличии лишь одного дисковода для ввода и вывода (входные и выходные файлы хранятся на одном диске). К сожалению, бесконечное увеличение m не приводит к ускорению обработки по "закону коэффициента log m". Причина заключается в том, что при достаточно больших значениях m время, необходимое для слияния в основной памяти (которое растет фактически пропорционально logm), превосходит время, требующееся для считывания или записи данных. Начиная с этого момента дальнейшее увеличение m ведет, по сути, к увеличению полного времени обработки данных, поскольку "узким местом" системы становятся вычисления в основной памяти.

Многофазная сортировка

Многоканальную (m-канальную) сортировку слиянием можно выполнить с помощью лишь m+1 файлов (в отличие от описанной выше 2m-файловой стратегии). При этом выполняется ряд проходов с объединением серий из m файлов в более длинные серии в (m+1)-м файле. Вот последовательные шаги такой процедуры.

1. В течение одного прохода, когда серии от каждого из m файлов

объединяются в серии (m+1)-го файла, нет нужды использовать все серии от каждого из m входных файлов. Когда какой-либо из файлов становится выходным, он заполняется сериями определенной длины, причем количество этих серий равно минимальному количеству серий, находящихся в сливаемых файлах.

2. В результате каждого прохода получаются файлы разной длины. Поскольку каждый из файлов, загруженных сериями в результате предшествующих m проходов, вносит свой вклад в серии текущего прохода, длина всех серий на определенном проходе представляет собой сумму длин серий, созданных за предшествующие m проходов. (Если выполнено менее m проходов, можно считать, что гипотетические проходы, выполненные до первого прохода, создавали серии длины 1).

Подобный процесс сортировки слиянием называется многофазной сортировкой. Точный подсчет требуемого количества проходов как функции от m (количества файлов) и n (количество записей), а также нахождения оптимального начального распределения серий между m файлами оставлен для упражнений. Однако приведем здесь один пример общего характера.

Пример. Если m = 2, то начинаем с двух файлов f_1 и f_2, организованных в виде серий длины 1. Записи из f_1 и f_2 объединяются, образуя серии длины 2 в третьем файле f_3. Выполняется слияние серий до полного опустошения файла f_1 (здесь для определенности предполагаем, что в файле f_1 меньше записей, чем в файле f_2). Затем объединяем оставшиеся серии длины 1 из f_2 с таким же количеством серий длины 2 из f_3. В результате получаются серии длины 3, которые помещаются в файл f_1. Затем объединяем серии длины 2 из f_3 с сериями длины 3 из f_1. Эти серии длиной 5 помещаются в файл f_2, который был исчерпан во время предыдущего прохода.

Последовательность длин серий 1, 1, 2, 3, 5, 8, 13, 21, … представляет собой последовательность чисел Фибоначчи. Эта последовательность удовлетворят рекуррентному соотношению $F_i = F_{i-1} + F_{i-2}$ для $i \geq 2$ с начальными значениями $F_0 = F_1 = 1$. Обратите внимание, что отношение последовательных чисел Фиббоначи F_{i-1}/F_i приближается к "золотому соотношению" $(\sqrt{5}+1)/2 = 1/618...$(->/5+1)/2 по мере увеличения i.

Оказывается, чтобы сохранить нужный ход процесса сортировки (пока не будет отсортирован весь список), начальные количества записей в f_1 и f_2 должны представлять собой два последовательных числа Фибоначчи. Например, в таблице показано, что произойдет, если начнем наш процесс с n=34 записями (34 — число Фибоначчи F_8), распределенными следующим образом: 13 записей в файле f_1 и 21 запись в файле f_2 (13 и 21 представляют собой числа Фибоначчи F_6 и F_7, поэтому

отношение F_7/F_6 равно 1.615, очень близко к 1.618). Состояние файлов в таблице показано как a(b), что означает a серий длиной b.

Таблица. Пример многофазной сортировки

После прохода	f_1	f_2	f_3
Вначале	13(1)	21(1)	Пустой
1	Пустой	8(1)	13(2)
2	8(3)	Пустой	5(2)
3	3(3)	5(5)	Пустой
4	Пустой	2(5)	3(8)
5	2(13)	Пустой	1(8)
6	1(13)	1(21)	Пустой
7	Пустой	Пустой	1(34)

Когда скорость ввода-вывода не является „узким местом"

Когда "узким местом" является считывание файлов, необходимо очень тщательно выбирать блок, который должен считываться следующим. Как уже было указано нужно избегать ситуаций, когда требуется запоминать много блоков одной серии, поскольку в этой серии наверняка имеются записи с большими значениям ключей, которые будут выбраны только после большинства (или всех) записей другой серии. Чтобы избежать этой ситуации, нужно быстро определить, какая серия первой исчерпает те свои записи, которые в данный момент находятся в основной памяти (эту оценку можно сделать, сравнив последние считанные записи из каждого файла).

Если время, необходимое для считывания данных в основную память, сопоставимо со временем, которое занимает обработка этих данных (или даже меньше его), тщательный выбор входного файла, из которого будет считываться блок, становится еще более важной задачей, поскольку иначе трудно сформировать резерв записей в основной памяти.

Рассмотрим случай, когда "узким местом" является слияние, а не считывание или запись данных. Это может произойти по следующим причинам.

1. Если в нашем распоряжении есть много дисководов или других накопителей, ввод-вывод можно ускорить настолько, что время для выполнения слияния превысит время ввода-вывода.

2. Может стать экономически выгодным применение более быстродействующих каналов обмена данными.

Поэтому имеет смысл подробнее рассмотреть проблему, с которой можно столкнуться в случае, когда "узким местом" в процессе сортировки слиянияем данных, хранящихся во вторичной памяти, становится их объединение. Сделаем следующие положения.

1. Объединяем серии, размеры которых намного превышают

размеры блоков.

2. Существуют два входных и два выходных файла. Входные файлы хранятся на одном внешнем диске (или каком-то другом устройстве, подключенном к основной памяти одним каналом), а выходные файлы — на другом подобном устройстве с одним каналом.

3. Время считывания, записи и выбора для заполнения блока записей с наименьшими ключами среди двух серий, находящихся в данный момент в основной памяти, одинаково.

С учетом этих предположений рассмотрим класс стратегий слияния, которые предусматривают выделение в основной памяти нескольких входных буферов (место для хранения блока). В каждый момент времени какой-то из этих буферов содержать невыделенные для слияния записи из двух входных серий, причем одна из них будет находится в состоянии считывания из входного файла. Два других буфера будут содержать выходные записи, т.е. выделенные записи в надлежащим образом объединенной последовательности. В каждый момент времени один из этих буферов находится в состоянии записи в один из выходных файлов, а другой заполняется записями, выбранными из входных буферов.

Выполняются (возможно, одновременно) следующие действия.

1. Считывание входного блока во входной буфер.

2. Заполнение одного из выходных буферов выбранными записями, т.е. записями с наименьшими ключами среди тех, которые в настоящий момент находятся во входном буфере.

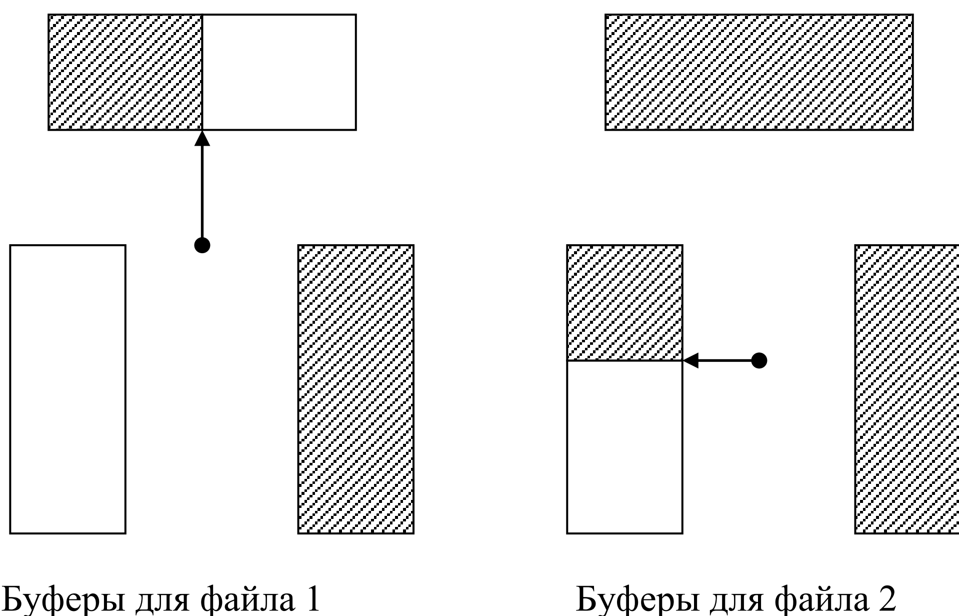
3. Запись данных другого выходного буфера в один из двух формируемых выходных файлов.

В соответствии с предположениями, эти действия занимают одинаковое время. Для обеспечения максимальной эффективности их следует выполнять параллельно. Это можно делать, если выбор записей с наименьшими ключами не включает записи, считываемые в данный момент. Следовательно, должны разработать такую стратегию выбора буферов для считывания, чтобы в начале каждого этапа (состоящего из описанных действий) b невыбранных записей с наименьшими ключами уже находились во входных буферах (b —количество записей, которые заполняют блок или буфер).

Условия, при которых слияние можно выполнять параллельно со считыванием, достаточно просты. Допустим, k_1 и k_2 — наибольшие ключи среди невыбранных записей в основной памяти из первой и второй серий соответственно. В таком случае в основной памяти должно быть по крайней мере b невыбранных записей, ключи которых не превосходят $\min(k_1, k_2)$. Сначала покажем, как можно выполнить слияние с шестью буферами (по три на каждый файл), а затем покажем, что будет достаточно четырех буферов, если они будут использоваться совместно для двух файлов.

Схема с шестью входными буферами

Схема работы с шестью входными буферами представлена на рис.44 (два выходных буфера здесь не показаны). Для каждого файла предусмотрены три буфера. Каждый буфер рассчитан на b записей. Заштрихованная область представляет имеющиеся записи, ключи расположены по окружности (по часовой стрелке) в возрастающем порядке. В любой момент времени общее количество невыбранных записей равняется 4b (если только не рассматриваются записи, оставшиеся от объединяемых серий). Поначалу считываем в буферы первые два блока из каждой серии. Поскольку всегда имеется 4b записей, а из одного файла может быть не более 3b записей, известно, что имеется по крайней мере b записей из каждого файла. Если k_1 и k_2 — наибольшие имеющиеся ключи в двух данных сериях, должно быть b записей с ключами, не большими, чем k_1, и b записей с ключами, не большими, чем k_2. Таким образом, имеются b записей с ключами, не большими, чем $\min(k_1, k_2)$.



Буферы для файла 1 Буферы для файла 2

Рисунок 44 – Схема слияния с шестью входными буферами.

Вопрос о том, какой файл считывать следующим, тривиален. Как правило, поскольку два буфера будут заполнены частично (как показано на рисунке 44), в нашем распоряжении будет только один пустой буфер, который и следует заполнять. Если получается, что каждая серия имеет два полностью заполненных и один пустой буфер, используйте для заполнения любой из двух пустых буферов. Обратите внимание: утверждение о том, что невозможно исчерпать серию (имеются b записей с ключами, не большими, чем $\min(k_1, k_2)$), опирается исключительно на факт наличия 4b записей.

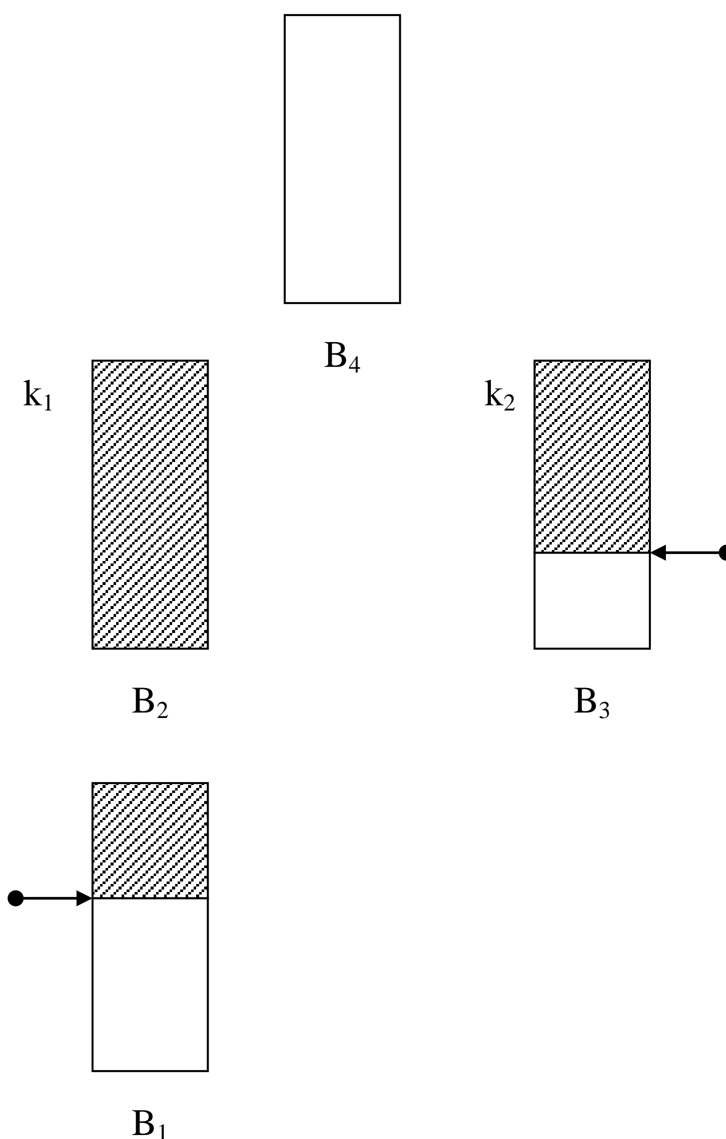
Стрелки на рис.44 изображают указатели на первые (с наименьшими ключами) имеющиеся записи из двух данных серий. В языке Pascal можно представить такой указатель в виде двух целых чисел. Первый, в диапазоне 1-3 представляет "указываемый" буфер, а второй, в диапазоне 1-b, — запись в этом буфере. Как альтернативный вариант можно реализовать эти буферы в виде первой, средней и последней трети одного массива и использовать одно целое число в диапазоне 1-3b. При использовании других языков, в которых указатели могут указывать на элементы массивов, следует предпочесть указатель типа ↑recordtype.

Схема с четырьмя буферами

На рисунке 45 представлена схема с четырьмя буферами. В начале каждого этапа у нас имеется 2b записей. Два входных буфера назначаются одному из файлов (B_1 и B_2 на рис.45 назначены файлу 1). Один из этих буферов будет заполнен частично (в крайнем случае он будет пустым), а другой — полностью. Третий буфер назначается другому файлу, например B_3 на рис.45 назначен файлу 2. Он заполнен частично (в крайнем случае он будет заполнен полностью). Четвертый буфер не назначается ни одному из файлов. На данной стадии он заполняется из одного из этих файлов.

Конечно, сохраняется возможность выполнения слияния параллельно со считыванием: по крайней мере b записей из тех, которые показаны на рисунке 45, должны иметь ключи, не превышающие min(k_1, k_2), где k_1 и k_2— ключи последних имеющихся записей из двух данных файлов. Конфигурацию буферов, которая позволяет выполнять параллельную работу по слиянию и считыванию, назовем безопасной. Поначалу считывается один блок из каждого файла (крайний случай, когда буфер B_1 пустой, а буфер B_3 целиком заполнен); в результате начальная конфигурация оказывается безопасной. Должны (в предположении, что ситуация, представленная на рис.45, соответствует безопасной конфигурации) показать, что конфигурация будет безопасной и после завершения следующего этапа.

Если $k_1 < k_2$, тогда надо заполнить B_4 следующим блоком из файла 1; в противном случае заполняем его из файла 2. Допустим сначала, что $k_1 < k_2$. Поскольку буферы B_1 и B_3 на рисунке 45 содержат в точности b записей, на следующем этапе должны исчерпать B_1; в противном случае исчерпали бы B3 и нарушили безопасность конфигурации, представленной на рисунке 45. Таким образом, по завершении этапа конфигурация примет вид, показанный на рисунке 45а.



Буферы для файла 1 Буфер для файла 2

Рисунок 45 – Схема слияния с четырьмя буферами.

Чтобы убедиться в том, что конфигурация на рис. 45,а действительно безопасна, рассмотрим два случая. Во-первых, если k_3 (последний ключ во вновь прочитанном блоке B_4) оказывается меньше k_2, тогда при целиком заполненном блоке B_4 можно быть уверенным, в наличии b записей, не превышающих $\min(k_1, k_2)$, поэтому соответствующая конфигурация является безопасной. Если $k_2 \le k_3$, тогда в силу предположения, что $k_1 < k_2$ (в противном случае заполнили бы B_4 из файла 2), b записей в B_2 и B_3 имеют ключи, не превышающие $\min(k_2, k_3) == k_2$.

Теперь рассмотрим случай, когда $k_1 \ge k_2$ (см. рис.46). Здесь необходимо считывать следующий блок из файла 2. На рис. 46,б показана итоговая ситуация. Как и в случае $k_1 < k_2$, можно утверждать, что B_1 должен исчерпаться, вот почему на рис. 46,6 показано, что файл 1 имеет только буфер B_2. Доказательство того, что на рис. 46,6 показана безопасная

конфигурация, ничем не отличается от доказательства подобного факта для рис.46а.

Обратите внимание: как и в случае схемы с шестью входными буферами, считываем файл после конца серии. Но если нет необходимости считывать блок из одной из имеющихся серий, можно считать блок из следующей серии в том же файле. Таким образом, появляется возможность считать один блок из каждой из следующих серий и приступить к слиянию серий сразу же после того, как будут выбраны последние записи предыдущей серии.

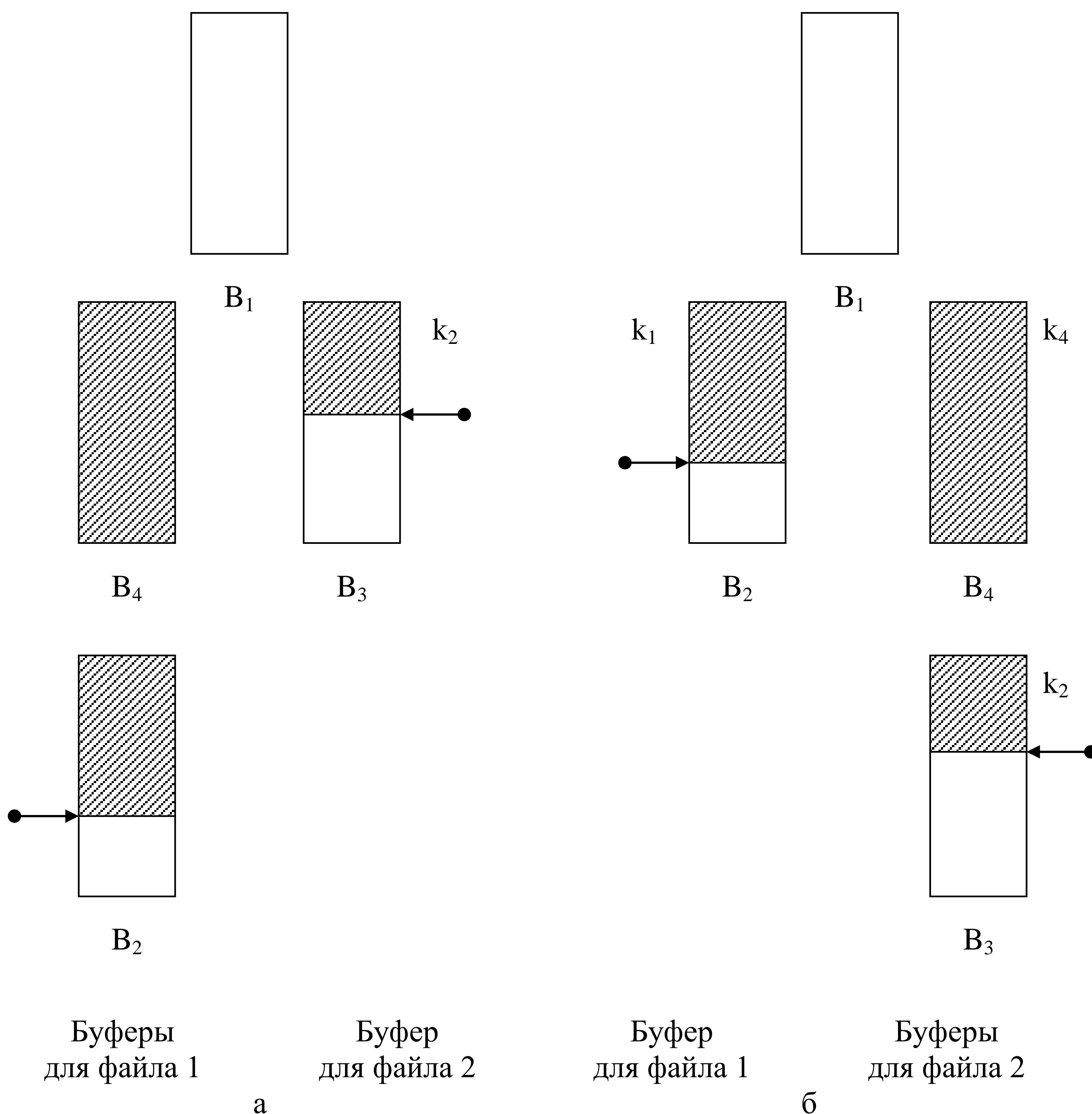


Рисунок 46 – Конфигурация буферов по завершении одного этапа.

6.3. Хранение данных в файлах

В этом разделе рассмотрим структуры данных и алгоритмы для хранения и поиска информации в файлах, находящихся во внешней памяти. Файл будем рассматривать как последовательность записей, причем каждая запись состоит из одной и той же совокупности полей. Поля могут иметь либо фиксированную длину (заранее определенное количество байт), либо переменную. Файлы с записями фиксированной длины широко используются в системах управления базам данных для хранения данных со сложной структурой. Файлы с записями переменной длины, как правило, используются для хранения текстовой информации. В этом разделе будем иметь дело с полями фиксированной длины; рассмотренные методы работы после определенной (несложной) модификации могут использоваться для работы с записями переменной длины.

Рассмотрим следующие операторы для работы с файлами.

1. INSERT вставляет определенную запись в определенный файл.
2. DELETE удаляет из определенного файла все записи, содержащие указанные значения в указанных полях.
3. MODIFY изменяет все записи в определенном файле, задав указанные значения определенным полям в тех записях, которые содержат указанные значения в других полях.
4. RETRIEVE отыскивает все записи, содержащие указанные значения в указанных полях.

Рассматривая операции с файлами, в первом приближении можно считать, что файлы — это просто совокупности записей, над которыми можно выполнять операторы, обсуждавшиеся ранее в АТД. Однако имеются два важных отличия. Во-первых, когда говорим о файлах, хранящихся на устройствах внешней памяти, то при оценивании тех или иных стратегий организации файлов нужно использовать меру затрат, обсуждавшихся в первом разделе данной главы. Другими словами, предполагаем, что файлы хранятся в виде некоторого количества физических блоков, а затраты на выполнение оператора пропорциональны количеству блоков, которые должны считать в основную память или записать из основной памяти на устройство внешней памяти.

Второе отличие заключается в том, что для записей, представляющих собой конкретные типы данных в большинстве языков программирования, могут быть предусмотрены указатели, в то время как для абстрактных элементов из некоторой совокупности никакие "указатели" предусмотреть нельзя. Например, в системах баз данных при организации данных часто используются указатели на записи. Следствием применения подобных указателей является то, что записи часто приходится считать закрепленными: их нельзя перемещать во внешней памяти, поскольку не исключено, что какой-то неизвестный нам указатель

после таких перемещений записи будет указывать неправильный адрес этой записи.

Простой способ представления указателей на записи заключается в следующем. У каждого блока есть определенный физический адрес, т.е. место начала этого блока на устройстве внешней памяти. Отслеживание физических адресов является задачей файловой системы. Одним из способов представления адресов записей является использование физического адреса блока, содержащего интересующую нас запись, со смещением, указывающим количество байт в блоке, предшествующих началу этой записи. Такие пары "физический адрес-смещение" можно хранить в полях типа "указатель на запись".

Простая организация данных

Простейшим (и наименее эффективным) способом реализации перечисленных выше операторов работы с. файлами является использование таких примитивов чтения и записи файлов, которые встречаются в большинстве языков программирования. В случаев пользования подобной "организации" (которая на самом деле является дезорганизацией) записи могут храниться в любом порядке. Поиск записи с указанными значениями в определенных полях осуществляется путем полного просмотра файла и проверки каждой его записи на наличие в ней заданных значений. Вставку в файл можно выполнять путем присоединения соответствующей записи к концу файла.

В случае изменения записей необходимо просмотреть файл, проверить каждую запись и выяснить, соответствует ли она заданным условиям (значениям в указанных полях). Если соответствует, в запись вносятся требуемые изменения. Принцип действия операции удаления почти тот же, но когда находим запись, поля которой соответствуют значениям, заданным в операции удаления, должны найти способ удалить ее. Один из вариантов — сдвинуть все последовательные записи в своих блоках на одну позицию вперед, а первую запись в каждом последующем блоке переместить на последнюю позицию предыдущего блока данного файла. Однако такой подход не годится, если записи являются закрепленными, поскольку указатель на i-ю запись в файле после выполнения этой операции будет указывать на (i + 1)-ю запись.

Если записи являются закрепленными, следует воспользоваться каким-то другим подходом. Должны как-то помечать удаленные записи, но не должны смещать оставшиеся на место удаленных (и не должны вставлять на их место новые записи). Таким образом выполняется логическое удаление записи из файла, но ее место в файле остается незанятым. Это нужно для того, чтобы в случае появления указателя на удаленную запись была возможность, во-первых, понять, что указываемая запись уже удалена, и, во-вторых, предпринять соответствующие меры

(например, присвоить этому указателю значение NIL, чтобы в следующий раз не тратить время на его анализ). Существуют два способа помечать удаленные записи.

1. Заменить запись на какое-то значение, которое никогда не может стать значением "настоящей" записи, и, встретив указатель на какую-либо запись, считать ее удаленной, если она содержит это значение.
2. Предусмотреть для каждой записи специальный бит удаления; этот бит содержит 1 в удаленных записях и 0 — в "настоящих" записях.

Ускорение операций с файлами

Очевидным недостатком последовательного файла является то, что операторы с такими файлами выполняются медленно. Выполнение каждой операции требует, чтобы был прочитан весь файл, а после этого еще и выполнить перезапись некоторых блоков. К счастью, существуют такие способы организации файлов, кот позволяют обращаться к записи, считывая в основную память лишь небольшую часть файла.

Такие способы организации файлов предусматривают наличие у каждой записи файла так называемого ключа, т.е. совокупности полей, которая уникальным образом идентифицирует каждую запись. Например, в файле с полями фамилия, адрес, телефон поле фамилия само по себе может считаться ключом, т.е. можно предположить, что в таком файле не может одновременно быть двух записей с одинаковым значением поля фамилия. Поиск записи, когда заданы значения ее ключевых полей, является типичной операцией, на обеспечение максимальной эффективности которой ориентированы многие широко распространенные способы организации файлов.

Еще одним непременным атрибутом быстрого выполнения операций с файлами является возможность непосредственного доступа к блокам (в отличие от последовательного перебора всех блоков, содержащих файл). Многие структуры данный, которые используются для быстрого выполнения операций с файлами, используют указатели на сами блоки, которые представляют собой физические адреса этих блоков (о физических адресах блоков было сказано выше). К сожалению, во многих языках программирования невозможно писать программы, работающие с данными на уровне физических блоков и их адресов, — такие операции, как правило, выполняются с помощью команд файловой системы. Однако приведем краткое неформальное описание принципа действия операторов, в которых используется прямой доступ к блокам.

Хешированные файлы

Хеширование — широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Основная идея этого метода подобна открытому хешированию. Записи

файла распределяем между так называемыми сегментами, каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти. Имеется таблица сегментов, содержащая В указателей, — по одному на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента.

Сегменты пронумерованы от 0 до И-1. Хеш-функция h отображает каждое значение ключа в одно из целых чисел от 0 до В-1. Если x — ключ, то h(x) является номером сегмента, который содержит запись с ключом x (если такая запись вообще существует). Блоки, составляющие каждый сегмент, образуют связный список. Таким образом, заголовок i-го блока содержит указатель на физический адрес (i+1)-го блока. Последний блок сегмента содержит в своем заголовке NIL-указатель.

Такой способ организации показан на рис. 47. Основной особенностью данного представления заключается в том, что в данном случае элементы, хранящиеся в с ном блоке сегмента, не требуется связывать друг с другом с помощью указателей связывать между собой нужно только блоки.

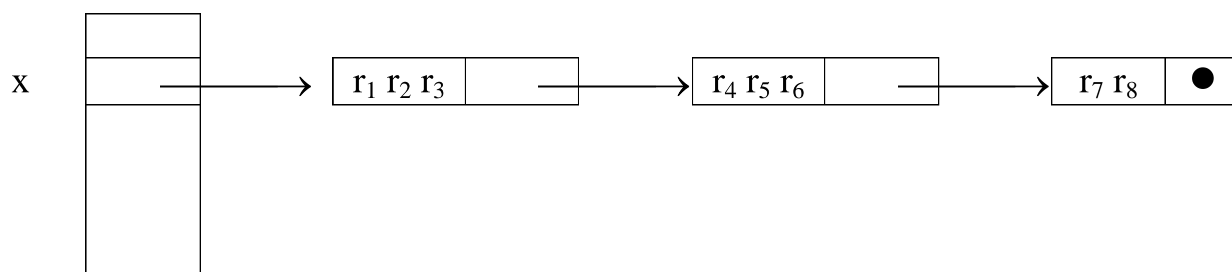


Рисунок 47 – Сегменты, состоящие из связанных блоков.

Если размер таблицы сегментов невелик, ее можно хранить в основной памяти. В противном случае ее можно хранить последовательным способом в отдельных блоках. Если нужно найти запись с ключом x, вычисляется h(x) и находится блок таблицы сегментов, содержащий указатель на первый блок сегмента h(x). Затем последовательно считываются блоки сегмента h(x), пока не обнаружится блок, который содержит запись с ключом x. Если исчерпаны все блоки в связном списке для сегмента h(x), приходим к выводу, что x не является ключом ни одной из записей.

Такая структура оказывается вполне эффективной, если в выполняемом операторе указываются значения ключевых полей. Среднее количество обращений к блокам, требующееся для выполнения оператора, в котором указан ключ записи, приблизительно равняется среднему

количеству блоков в сегменте, которое равно n/bk, если n — количество записей, блок содержит b записей, а k соответствует количеству сегментов. Таким образом, при такой организации данных операторы, использующие значения ключей, выполняются в среднем в k раз быстрее, чем в случае неорганизованного файла. К сожалению, ускорения операций, не основанных на использовании ключей, добиться не удается, поскольку при выполнении подобных операций приходится анализировать практически все содержимое сегментов. Единственным универсальным способом ускорения операций, не основанных на использовав ключей, по-видимому, является применение вторичных индексов.

Чтобы вставить запись с ключом, значение которого равняется x, нужно сначала проверить, нет ли в файле записи с таким значением ключа. Если такая запись есть, выдается сообщение об ошибке, поскольку предполагаем, что ключ уникальным образом идентифицирует каждую запись. Если записи с ключом x нет, то вставляем новую запись в первый же блок цепочки для сегмента h(x), в который эту запись удается вставить. Если запись не удается вставить ни в какой из существующих блоков сегмента h(x), файловой системе выдается команда найти новый блок, в который будет помещена эта запись. Этот новый блок затем добавляется в конец цепочки блоков сегмента h(x).

Чтобы удалить запись с ключом x, нужно сначала найти эту запись, а затем установить ее бит удаления. Еще одной возможной стратегией удаления (которой, впрочем, нельзя пользоваться, если имеем дело с закрепленными записями) является замена удаленной записи на последнюю запись в цепочке блоков сегмента h(x). Если такое изъятие последней записи приводит к опустошению последнего блока в сегменте h(x), этот пустой блок можно затем вернуть файловой системе для повторного использования.

Хорошо продуманная организация файлов с хешированным доступом требует лишь незначительного числа обращений к блокам при выполнении каждой операции с файлами. Если имеем дело с хорошей функцией хеширования, а количество сегментов приблизительно равно количеству записей в файле, деленному на количество записей, которые могут уместиться в одном блоке, тогда средний сегмент состоит из одного блока. Если не учитывать обращения к блокам, которые требуют для просмотра таблицы сегментов, типичная операция поиска данных, основанного на ключах, потребует лишь одного обращения к блоку, а операции вставки, удаления или изменения потребуют двух обращений к блокам. Если среднее количество записей в сегменте намного превосходит количество записей, которые могут уместиться в одном блоке, можно периодически реорганизовывать таблицу сегментов, удваивая количество сегментов и деля каждый сегмент на две части.

Индексированные файлы

Еще одним распространенным способом организации файла записей является поддержание файла в отсортированном (по значениям ключей) порядке. В этом случае файл можно было бы просматривать как обычный словарь или телефонный справочник, когда просматриваем лишь заглавные слова или фамилии на каждой странице. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый разреженным индексом, который состоит из пар (x, b), где x — значение ключа, а b — физический адрес блока, в котором значение ключа равняется x. Этот разреженный индекс отсортирован по значениям ключей.

Пример. На рис. 48 показан файл, а также соответствующий ему файл разрешенного индекса. Предполагается, что три записи основного файла (или три пары индексного файла) умещаются в один блок. Записи основного файла представлены только значениями ключей, которые в данном случае являются целочисленными величинами.

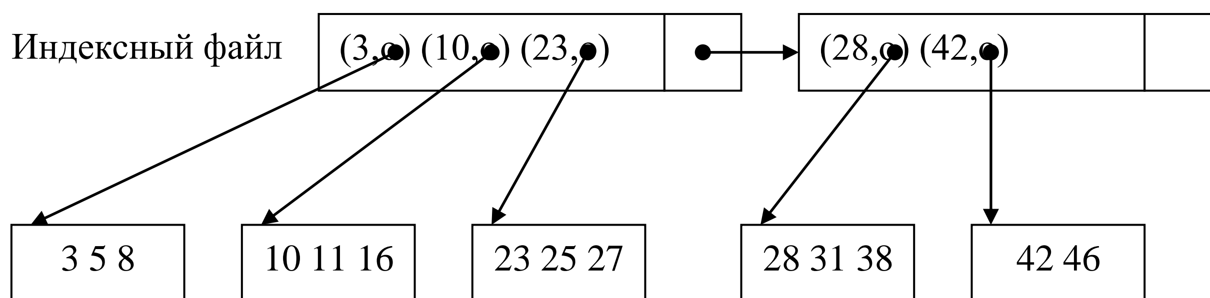


Рисунок 48 – Основной файл и его разреженный индекс.

Чтобы отыскать запись с заданным ключом x, надо сначала просмотреть индексный файл, отыскивая в нем пару (x. b). В действительности отыскивается наибольшее z, такое, что $z \le x$ и далее находится пара (z, b). В этом случае ключ x оказывается в блоке b (если такой ключ вообще присутствует в основном файле).

Разработано несколько стратегий просмотра индексного файла. Простейшей из них является линейный поиск. Индексный файл читается с самого начала, пока встретится пара (x, b) или (y, b), причем $y > x$. В последнем случае у предыдущей пары (z, b') должно быть $z < x$, и если запись с ключом x действительно существует, она находится в блоке b'.

Линейный поиск годится лишь для небольших индексных файлов. Более эффективным методом является двоичный поиск. Допустим, что индексный файл хранится в блоках $b_1, b_2, \ldots, b_n$. Чтобы отыскать значение ключа x, берется средний блок $b_{[n/2]}$ и x сравнивается со значением ключа y в первой паре данного блока. Если $x < y$, поиск повторяется в блоках $b_1, b_2, \ldots, b_{[n/2]-1}$. Если $x \ge y$, но x меньше, чем ключ блока $b_{[n/2]+1}$, используется

линейный поиск, чтобы проверить, совпадает ли x с первым компонентом индексной пары в блоке $b_{[n/2]}$. В противном случае повторяется поиск в блоках $b_{[n/2]+1}$, $b_{[n/2]+2}$, …, b_n . При использовании двоичного поиска нужно проверить лишь $[\log_2(n+1)]$ блоков индексного файла.

Чтобы создать индексированный файл, записи сортируются по значениям их ключей, а затем распределяются по блокам в возрастающем порядке ключей. В каждый блок можно упаковать столько записей, сколько туда умещается. Однако в каждом блоке можно оставлять место для дополнительных записей, которые могут вставляться туда впоследствии. Преимущество такого подхода заключается в том, что вероятность переполнения блока, куда вставляются новые записи, в этом случае оказывается ниже (иначе надо будет обращаться к смежным блокам). После распределения записей по блокам создается индексный файл: просматривается по очереди каждый блок и находится первый ключ в каждом блоке. Подобно тому, как это сделано в основном файле, в блоках, содержащих индексный файл, можно оставить какое-то место для последующего "роста".

Допустим, есть отсортированный файл записей, хранящихся в блоках B_1, B_2, …,B_m. Чтобы вставить в этот отсортированный файл новую запись, используем индексный файл, с помощью которого определим, какой блок B_i должен содержать новую запись. Если новая запись умещается в блок B_i, она туда помещается в правильной последовательности. Если новая запись становится первой записью в блоке B_i, тогда выполняется корректировка индексного файла.

Если новая запись не умещается в блок B_i, можно применить одну из нескольких стратегий. Простейшая из них заключается в том, чтобы перейти на блок B_{i+1} (который можно найти с помощью индексного файла) и узнать, можно ли последнюю запись в B_i переместить в начало B_{i+1}. Если можно, последняя запись перемещается в B_{i+1}, а новую запись можно затем вставить на подходящее место в B_i. В этом случае, конечно, нужно скорректировать вход индексного файла для B_{i+1} (и, возможно, для B_i).

Если блок B_{i+1} также заполнен или если B_i является последним блоком (i=m), из файловой системы нужно получить новый блок. Новая запись вставляется в этот новый блок, который должен размещаться вслед за блоком B_i. Затем используется процедура вставки в индексном файле записи для нового блока.

Несортированные файлы с плотным индексом

Еще одним способом организации файла записей является сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого будут отыскиваться требуемые записи; этот файл называется плотным индексом. Плотный индекс состоит из пар (x, p), где p — указатель на запись с ключом x в основном файле. Эти пары

отсортированы по значениям ключа. В результате структуру, подобную упоминавшемуся выше разреженному индексу (или B-дереву, речь о котором пойдет в следующем разделе), можно использовать для поиска ключей в плотном индексе.

При использовании такой организации плотный индекс служит для поиска в основном файле записи с заданным ключом. Если требуется вставить новую запись, отыскивается последний блок основного файла и туда вставляется новая запись. Если последний блок полностью заполнен, то надо получить новый блок из файловой системы. Одновременно вставляется указатель на соответствующую запись в файле плотного индекса. Чтобы удалить запись, в ней просто устанавливается бит удаления и удаляется соответствующий вход в плотном индексе (возможно, устанавливая и здесь бит удаления).

Вторичные индексы

В то время как хешированные и индексированные структуры ускоряют выполнение операций с файлами, основываясь главным образом на ключах, ни один из этих методов не помогает, когда операция связана с поиском записей, если заданы значения полей, не являющихся ключевыми. Ели требуется найти запись с указанными значениями полей F_1, F_2, …, F_k, понадобится вторичный индекс для этих полей. Вторичный индекс — это файл, состоящий из пар (v, p), где м представляет собой список значений, по одному для каждого из полей F_1, F_2, …, F_k, а p— указатель на запись. В файле вторичного индекса может быть несколько пар с заданным v, и каждый сопутствующий указатель должен указывать на запись в основном файле, которая содержит v в качестве списка значений полей F_1, F_2, …, F_k.

Чтобы отыскать записи, когда заданы значения полей F_1, F_2, …, F_k, отыскиваем в файле вторичного индекса запись (или записи) с заданным списком значений. Сам файл вторичного индекса может быть организован любым из перечисленных выше способов организации файлов по значениям ключей. Таким образом, предполагаем, что v является ключом для пары (v, p).

Например, организация файлов с помощью хеширования практически не зависит от того, уникальны ли ключи, — хотя, если бы оказалось очень много записей с одним и тем же значением "ключа", записи могли бы распределиться по сегментам очень неравномерно, и в результате хеширование не ускорило бы доступ к записям. Рассмотрим крайний случай, когда заданы только два значения для полей вторичного индекса. При этом все сегменты, за исключением двух, были бы пустыми, и таблица хеширования, независимо от имеющегося количества сегментов, ускоряла бы выполнение операций лишь в два раза (и то в лучшем случае). Аналогично, разреженный индекс не требует, чтобы ключи были уникальны, но если они действительно не будут уникальными, тогда в

основном файле может оказаться два или больше блоков, содержащих одинаковые наименьшие значения "ключа", и когда потребуется найти записи с этим значением, необходимо будет просмотреть все такие блоки.

Если файл вторичного индекса организован по методу хеширования или разреженного индекса, может возникнуть желание сэкономить память, объединив все записи с одинаковыми значениями, т.е. пары (v, p_1), (v, p_2), ... , (v, p_m) можно заменить на v, за которым следует список $p_1, p_2, \ldots, p_m$.

Может возникнуть вопрос, нельзя ли оптимизировать время выполнения операторов, создав вторичный индекс для каждого поля или даже для всех подмножеств полей. К сожалению, придется "платить" за каждый вторичный индекс, который хотим создать. Во-первых, для хранения вторичного индекса тоже требуется место на диске, а этот ресурс (объем внешней памяти) тоже, хоть и не всегда, может быть дефицитным. Во-вторых, каждый создаваемый вторичный индекс замедляет выполнение всех операций вставки и удаления. Дело в том, что когда вставляется запись, надо также вставить точку входа для этой записи в каждый вторичный индекс, чтобы эти вторичные индексы по-прежнему правильно представляли соответствующий файл. Обновление вторичного индекса означает, что потребуется выполнить не менее двух обращений к блокам, поскольку должны и прочитать, и записать блок. Однако таких обращений к блокам может оказаться значительно больше, поскольку еще нужно найти требуемый блок, а любой способ организации файла, который используется для вторичного индекса, потребует в среднем еще нескольких обращений, чтобы найти нужный блок. Все сказанное относится и к операции удаления записей. Отсюда следует вывод, что решение использовать вторичные индексы требует взвешенного подхода, поскольку надо определить, какие совокупности полей будут указываться в операциях, выполняемых с файлами, настолько то время, которое собираемся сэкономить за счет применения вторичных индексов, превосходит затраты на обновление этого индекса при выполнении каждой операции вставки и удаления.

6.4. Внешние деревья поиска

Древовидные структуры данных, которые обсуждались ранее можно использовать для представления внешних файлов. B-дерево, являющееся обобщением 2-3 дерева, особенно удачно подходит для представления внешней памяти и стало стандартным способом организации индексов в системах баз данных. В этом разделе будут приведены базовые методы поиска, вставки и удаления информации в B-деревьях.

Разветвленные Деревья поиска

Обобщением дерева двоичного поиска является m-арное дерево, в котором каждый узел имеет не более m сыновей. Так же, как и для

деревьев двоичного поиска, будем считать, что выполняется следующее условие: если n_1 и n_2 являются двумя сыновьями одного узла и n_1 находится слева от n_2, тогда все элементы, исходящие вниз от n_1, оказываются меньше элементов, исходящих вниз от n_2. Операторы MEMBER, INSERT и DELETE для m-арного дерева поиска реализуются путем естественного обобщения тех операций для деревьев двоичного поиска, которые обсуждались ранее.

Однако в данном случае нас интересует проблема хранения записей в файлах, когда файлы хранятся в виде блоков внешней памяти. Правильным применением идеи разветвленного дерева является представление об узлах как о физических блоках. Внутренний узел содержит указатели на своих m сыновей, а также m-1 ключевых значений, которые разделяют потомков этих сыновей. Листья также являются блоками; эти блоки содержат записи основного файла.

Если использовать дерево двоичного поиска из n узлов для представления файла, хранящегося во внешней памяти, то для поиска записи в таком файле потребовалось бы в среднем $\log_2 n$ обращений к блокам. Если вместо дерева двоичного поиска использовать для представления файла m-арное дерево поиска, то для поиска записи в таком файле потребуется лишь $\log_m n$ обращений к блокам. В случае $n = 10^6$ дерево двоичного поиска потребовало бы примерно 20 обращений к блокам, тогда как 128-арное дерево поиска потребовало бы лишь 3 обращения к блокам.

Однако нельзя сделать m сколь угодно большим, поскольку чем больше m, тем больше должен быть размер блока. Более того, считывание и обработка более крупного блока занимает больше времени, поэтому существует оптимальное значение m, которое позволяет минимизировать время, требующееся для просмотра внешнего m-арного дерева поиска. На практике значение, близкое к такому минимуму, получается для довольно широкого диапазона величин m.

В-деревья

В-дерево — это особый вид сбалансированного m-арного дерева, который позволяет нам выполнять операции поиска, вставки и удаления записей из внешнего файла с гарантированной производительностью для самой неблагоприятной ситуации. Оно представляет собой обобщение 2-3 дерева. С формальной точки зрения В-дерево порядка m представляет собой m-арное дерево поиска, характеризующееся следующими свойствами.

1. Корень либо является листом, либо имеет по крайней мере двух сыновей.

2. Каждый узел, за исключением корня и листьев, имеет от [m/2] до m сыновей.

3. Все пути от корня до любого листа имеют одинаковую длину.

Обратите внимание: каждое 2-3 дерево является В-деревом порядка 3, т.е. 3-арным. На рис. 49 показано В-дерево порядка 5, здесь предполагается, что в блоке листа умещается не более трех записей.

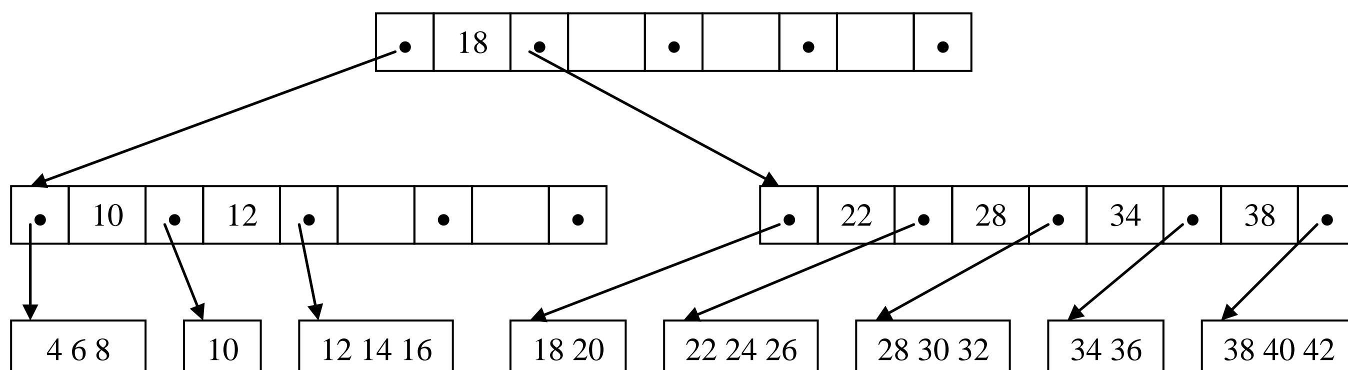


Рисунок 49 – В-дерево порядка 5.

В-дерево можно рассматривать как иерархический индекс, каждый узел в котором занимает блок во внешней памяти. Корень В-дерева является индексом первого уровня. Каждый нелистовой узел на В-дереве имеет форму (p_0, k_1, p_1, k_2, p_2, …, k_n, p_n), где p_i является указателем на i-го сына, $0 \le i \le n$, а k_i — ключи в узле упорядочены, поэтому $k_1 < k_2 < \ldots < k^n$. Все ключи в поддереве, на которое указывает p_0 меньше, чем k_1. В случае $1 \le i < n$ все ключи в поддереве, на которое указывает p_i, имеют значения, не меньшие, чем k_i, и меньшие, чем k_{i+1}. Все ключи в поддереве, на которое указывает p_n, имеют значения, не меньшие, чем k_n.

Существует несколько способов организации листьев. В данном случае предполагаем, что записи основного файла хранятся только в листьях. Предполагается также, что каждый лист занимает один блок.

Поиск записей

Если требуется найти запись r со значением ключа x, нужно проследить путь от корня до листа, который содержит r, если эта запись вообще существует в файле. Проходим этот путь, последовательно считывая из внешней памяти в основную, внутренние узлы (p_0, k_1, p_1, …, k_n, p_n) и вычисляя положение x относительно ключей k_1, k_2, …, k_n. Если $k_i \le x < k_{i+1}$, тогда в основную память считывается узел, на который указывает p_i, и повторяется описанный процесс. Если $x < k_i$, для считывания в основную память используется указатель p_0; если $x \ge k_n$, тогда используется p_n. Когда в результате этого процесса попадаем на какой-либо лист, то пытаемся найти запись со значением ключа x. Если количество входов в узле невелико, то в этом узле можно использовать линейный поиск; в противном случае лучше воспользоваться двоичным поиском.

Вставка записей

Если требуется вставить в B-дерево запись r со значением ключа x, нужно сначала воспользоваться процедурой поиска, чтобы найти лист L, которому должна принадлежать запись r. Если в L есть место для этой записи, то она вставляется в L в требуемом порядке. В этом случае внесение каких-либо изменений в предков листа L не требуется.

Если же в блоке листа L нет места для записи r, у файловой системы запрашивается новый блок L' и перемещается последняя половина записей из L в L', вставляя r в требуемое место в L или L'. Допустим, узел P является родителем узла L. P известен, поскольку процедура поиска отследила путь от корня к листу L через узел P. Теперь можно рекурсивно применить процедуру вставки, чтобы разместить в P ключ k' и указатель l' на L'; k' и l' вставляют сразу же после ключа и указателя для листа L. Значение k' является наименьшим значением ключа в L'.

Если P уже имеет m указателей, вставка k' и l' в P приведет к расщеплению P и потребует вставки ключа и указателя в узел родителя P. Эта вставка может произвести "эффект домино", распространяясь на предков узла L в направлении корня (вдоль пути, который уже был пройден процедурой поиска). Это может даже привести к тому, что понадобится расщепить корень (в этом случае создается новый корень, причем две половины старого корня выступают в роли двух его сыновей). Это единственная ситуация, при которой узел может иметь менее m/2 потомков.

Удаление записей

Если требуется удалить запись r со значением ключа x, нужно сначала найти лист L, содержащий запись г. Затем, если такая запись существует, она удаляется из L. Если r является первой записью в L, переходим после этого в узел P (родителя листа L), чтобы установить новое значение первого ключа для L. Но если L является первым сыном узла P, то первый ключ L не зафиксирован в P, а появляется в одном из предков P. Таким образом, надо распространить изменение в наименьшем значении ключа L в обратном направлении вдоль пути от L к корню.

Если блок листа L после удаления записи оказывается пустым, он отдается файловой системе. После этого корректируются ключи и указатели в P, чтобы отразить факт удаления листа L. Если количество сыновей узла P оказывается теперь меньшим, чем m/2, проверяется узел P' расположенный в дереве на том же уровне непосредственно слева (или справа) от P. Если узел P' имеет по крайней мере [m/2]+2 сыновей, ключи и указатели распределяются поровну между P и P' так, чтобы оба эти узла имели по меньшей мере [m/2] потомков, сохраняя, разумеется, упорядочение записей. Затем надо изменить значения ключей для P и P' в родителе P и, если необходимо, рекурсивно распространить воздействие

внесенного изменения на всех предков узла P, на которых это изменение отразилось.

Если у P' имеется ровно [m/2] сыновей, то объединяем P и P' в один узел с 2[m/2] - 1 сыновьями. Затем необходимо удалить ключ и указатель на P' из родителя для P'. Это удаление можно выполнить с помощью рекурсивного применения процедуры удаления.

Если "обратная волна" воздействия удаления докатывается до самого корня, возможно, придется объединить только двух сыновей корня. В этом случае новым корнем становится результирующий объединенный узел, а старый корень можно вернуть файловой системе. Высота B-дерева уменьшается при этом на единицу.

Время выполнения операций с B-деревом

Допустим, есть файл с n записями, организованный в виде B-дерева порядка m. Если каждый лист содержит в среднем b записей, тогда дерево содержит примерно [n/b] листьев. Самые длинные пути в таком дереве образуются в том случае, если каждый внутренний узел имеет наименьшее возможное количество сыновей, т.е. m/2. В этом случае будет примерно 2[n/b]/m родителей листьев, $4[n/b]/m^2$ родителей родителей листьев и т.д.

Если вдоль пути от корня к листу встречается j узлов, в этом случае $2^{j-1}[n/b]/m^{j-1} \geq 1$, в противном случае на уровне корня окажется меньше одного узла. Таким образом, $[n/b] \geq (m/2)^{j-1}$, а $j \leq 1 + \log_{m/2}[n/b]$. Например, если $n = 10^6$, b = 10, а m = 10, то j ≤ 3,9. Обратите внимание, что b является не максимальным количеством которые можно поместить в блок, а средним или ожидаемым количеством. Однако, перераспределяя записи между соседними блоками каждый раз, когда один из них опустошается больше чем наполовину, можем гарантировать, что b будет равняться по крайней мере половине максимального значения. Обратите внимание на высказанное нами выше предположение о том, что каждый внутренний узел имеет минимально возможное количество сыновей. На практике количество сыновей среднего внутреннего узла будет больше минимума, и приведенный выше анализ отражает, таким образом, самый "консервативный" случай.

В случае вставки или удаления записи для поиска нужного листа потребуется j обращений к блокам. Точное количество дополнительных обращений к блокам, необходимое для выполнения вставки или удаления записи и распространения воздействия этих операций по дереву, подсчитать весьма затруднительно. Чаще всего требуется перезаписать только один блок — лист, содержащий интересующую запись. Таким образом, в качестве ориентировочного количества обращений к блокам при выполнении вставки или удаления записи можно принять значение $2 + \log_{m/2}[n/b]$.

Сравнение методов

Итак, в качестве возможных методов организации внешних файлов были обсуждены хеширование, разреженные индексы и В-деревья. Интересно было бы сравнить количество обращений к блокам, связанное с выполнением той или иной операции с файлами, у разных методов.

Хеширование зачастую является самым быстрым из трех перечисленных методов; оно требует в среднем двух обращений к блокам по каждой операции (не считая обращений к блокам, которые требуются для просмотра самой таблицы сегментов), если количество сегментов достаточно велико для того, чтобы типичный сегмент использовал только один блок. Но в случае хеширования не легко обращаться к записям в отсортированной последовательности.

Разреженный индекс для файла, состоящего из n записей, позволяет выполнять операции с файлами, ограничиваясь использованием примерно $2 + \log(n/bb')$ обращений к блокам в случае двоичного поиска, где b — количество записей, помещающихся в один блок, а b'— количество пар "ключ-указатель", умещающихся в один блок для индексного файла. В-деревья позволяют выполнять операции с файлами с использованием примерно $2 + \log_{m/2}[n/b]$ обращений к блокам, где m — максимальное количество сыновей у внутренних узлов, что приблизительно равняется b'. Как разреженные указатели, так и В-деревья допускают обращение к записям в отсортированной последовательности.

Все перечисленные методы намного эффективнее обычного последовательного просмотра файла. Временное различия между ними, однако, невелики и не поддаются точной аналитической оценке, особенно с учетом того, что соответствующие параметры, такие как ожидаемая длина файла и коэффициенты заполненности блоков, трудно прогнозировать заранее.

Похоже на то, что В-деревья приобретают все большую популярность как средство доступа к файлам в системах баз данных. Причина этой популярности частично заключается в их способности обрабатывать запросы, запрашивая записи с ключами, относящимися к определенному диапазону (при этом используется преимущество упорядоченности записей в основном файле в соответствии с последовательностью сортировки). Разреженный индекс обрабатывает подобные запросы также достаточно эффективно — хотя, как правило, все же менее эффективно, чем В-деревья. На интуитивном уровне причина предпочтения В-деревьев (в сравнении с разреженным индексом) заключается в том, что В-дерево можно рассматривать как разреженный индекс для разреженного индекса для разреженного индекса и т.д. (Однако ситуации, когда требуется более трех уровней индексов, встречаются довольно редко.)

В-деревья также зарекомендовали себя относительно неплохо при использовании их в качестве вторичных указателей, когда "ключи" в действительности не определяют ту или иную уникальную запись. Даже если записи с заданным значением для указанных полей вторичного индекса охватывают несколько блоков, можно прочитать все эти записи, выполнив столько обращений к блокам, сколько существует блоков, содержащих эти записи, плюс число их предшественников в В-дереве. Для сравнения: если бы эти записи плюс еще одна группа такого же размера оказались хешированными в одном и том же сегменте, тогда поиск любой из этих групп в таблице хеширования потребовал бы такого количества обращений к блокам, которое приблизительно равняется удвоенному числу блоков, требующемуся для размещения любой из этих групп. Возможно, есть и другие причины популярности В-деревьев, например их высокая эффективность в случае, когда к такой структуре одновременно обращается несколько процессов.

7. ХЕШИРОВАНИЕ. ПРИКЛАДНЫЕ АЛГОРИТМЫ

Для работы с словарем требуются *поиск*, *вставка* и *удаление*. Один из наиболее эффективных способов реализации словаря – хеш-таблицы. Среднее время поиска элемента в них есть $\boldsymbol{O}(1)$, время для наихудшего случая – $\boldsymbol{O}(n)$. Прекрасное изложение хеширования можно найти в работах Кормена и Кнута. Чтобы читать статьи на эту тему, вам понадобится владеть соответствующей терминологией. Здесь описан метод, известный как связывание или открытое хеширование. Другой метод, известный как замкнутое хеширование или закрытая адресация, здесь не обсуждаются.

7.1. Теория

Хеш-таблица – это обычный массив с необычной адресацией, задаваемой хеш-функцией. Например, на **hashTable** рис. 50 – это массив из 8 элементов. Каждый элемент представляет собой указатель на линейный список, хранящий числа. Хеш-функция в этом примере просто делит ключ на 8 и использует остаток как индекс в таблице. Это дает нам числа от 0 до 7 Поскольку для адресации в **hashTable** нам и нужны числа от 0 до 7, алгоритм гарантирует допустимые значения индексов.

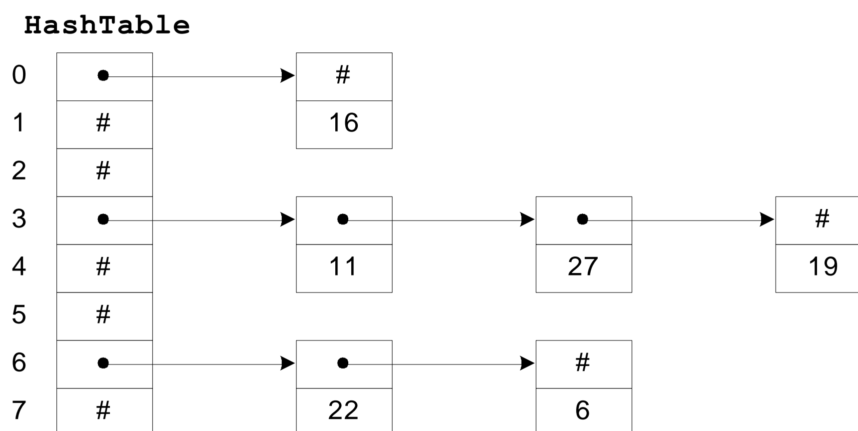


Рисунок 50 – Хеш-таблица.

Чтобы вставить в таблицу новый элемент, мы хешируем ключ, чтобы определить список, в который его нужно добавить, затем вставляем элемент в начало этого списка. Например, чтобы добавить 11, мы делим 11 на 8 и получаем остаток 3. Таким образом, 11 следует разместить в списке, на начало которого указывает **hashTable[3]**. Чтобы найти число, мы его хешируем и проходим по соответствующему списку. Чтобы удалить число, мы находим его и удаляем элемент списка, его содержащий.

Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов, то хеширование эффективно разбивает множество ключей. Наихудший случай – когда все ключи

хешируются в один индекс. При этом мы работаем с одним линейным списком, который и вынуждены последовательно сканировать каждый раз, когда что-нибудь делаем. Отсюда видно, как важна хорошая хеш-функция. Здесь мы рассмотрим лишь несколько из возможных подходов. При иллюстрации методов предполагается, что *unsigned char* располагается в 8 битах, *unsigned short int* – в 16, *unsigned long int* – в 32.

- *Деление (размер таблицы* **hashTableSize** *– простое число).* Этот метод использован в последнем примере. Хеширующее значение **hashValue**, изменяющееся от 0 до (**hashTableSize - 1**), равно остатку от деления ключа на размер хеш-таблицы. Вот как это может выглядеть:

```
typedef int hashIndexType;

hashIndexType hash(int Key) {
   return Key % hashTableSize;
}
```

  Для успеха этого метода очень важен выбор подходящего значения **hashTableSize**. Если, например, **hashTableSize** равняется двум, то для четных ключей хеш-значения будут четными, для нечетных – нечетными. Ясно, что это нежелательно – ведь если все ключи окажутся четными, они попадут в один элемент таблицы. Аналогично, если все ключи окажутся четными, то **hashTableSize**, равное степени двух, попросту возьмет часть битов **Key** в качестве индекса. Чтобы получить более случайное распределение ключей, в качестве **hashTableSize** нужно брать простое число, не слишком близкое к степени двух.

- *Мультипликативный метод (размер таблицы* **hashTableSize** есть степень *2^n).* Значение **Key** умножается на константу, затем от результата берется необходимое число битов. В качестве такой константы Кнут рекомендует золотое сечение $\left(\sqrt{5}-1\right)/2=$ 0.6180339887499. Пусть, например, мы работаем с байтами. Умножив золотое сечение на 2^8, получаем 158. Перемножим 8-битовый ключ и 158, получаем 16-битовое целое. Для таблицы длиной 2^5 в качестве хеширующего значения берем 5 младших битов младшего слова, содержащего такое произведение. Вот как можно реализовать этот метод:

```
/* 8-bit index */
typedef unsigned char hashIndexType;
static const hashIndexType K = 158;
```

```
/* 16-bit index */
typedef unsigned short int hashIndexType;
static const hashIndexType K = 40503;

/* 32-bit index */
typedef unsigned long int hashIndexType;
static const hashIndexType K = 2654435769;

/* w=bitwidth(hashIndexType), size of table=2**m */
static const int S = w - m;
hashIndexType hashValue = (hashIndexType)(K * Key) >> S;
```

Пусть, например, размер таблицы **hashTableSize** равен 1024 (2^{10}). Тогда нам достаточен 16-битный индекс и **S** будет присвоено значение 16 – 10 = 6. В итоге получаем:

```
typedef unsigned short int hashIndexType;

hashIndexType hash(int Key) {
   static const hashIndexType K = 40503;
   static const int S = 6;
   return (hashIndexType)(K * Key) >> S;
}
```

- *Аддитивный метод для строк переменной длины (размер таблицы равен 256).* Для строк переменной длины вполне разумные результаты дает сложение по модулю 256. В этом случае результат **hashValue** заключен между 0 и 244.

```
typedef unsigned char hashIndexType;

hashIndexType hash(char *str) {
   hashIndexType h = 0;
   while (*str) h += *str++;
   return h;
}
```

- *Исключающее ИЛИ для строк переменной длины (размер таблицы равен 256).* Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод, как легко догадаться, заключается в том, что к элементам строки последовательно применяется операция “исключающее или”. В нижеследующем алгоритме добавляется случайная компонента, чтобы еще улучшить результат.

```
typedef unsigned char hashIndexType;
```

```
unsigned char Rand8[256];

hashIndexType hash(char *str) {
   unsigned char h = 0;
   while (*str) h = Rand8[h ^ *str++];
   return h;
}
```

Здесь **Rand8** – таблица из 256 восьмибитовых случайных чисел. Их точный порядок не критичен. Корни этого метода лежат в криптографии; он оказался вполне эффективным [4].

- *Исключающее ИЛИ для строк переменной длины (размер таблицы ≤ 65536).* Если мы хешируем строку дважды, мы получим хеш-значение для таблицы любой длины до 65536. Когда строка хешируется во второй раз, к первому символу прибавляется 1. Получаемые два 8-битовых числа объединяются в одно 16-битовое.

```
typedef unsigned short int hashIndexType;
unsigned char Rand8[256];

hashIndexType hash(char *str) {
   hashIndexType h;
   unsigned char h1, h2;

   if (*str == 0) return 0;
   h1 = *str; h2 = *str + 1;
   str++;
   while (*str) {
      h1 = Rand8[h1 ^ *str];
      h2 = Rand8[h2 ^ *str];
      str++;
   }

   /* h is in range 0..65535 */
   h = ((hashIndexType)h1 << 8)|(hashIndexType)h2;
   /* use division method to scale */
   return h % hashTableSize
}
```

Размер хеш-таблицы должен быть достаточно велик, чтобы в ней оставалось разумное число пустых мест. Как видно из таблицы, чем меньше таблица, тем больше среднее время поиска ключа в ней. Хеш-таблицу можно рассматривать как совокупность связанных списков. По

мере того, как таблица растет, увеличивается количество списков и, соответственно, среднее число узлов в каждом списке уменьшается. Пусть количество элементов равно *n*. Если размер таблицы равен 1, то таблица вырождается в один список длины *n*. Если размер таблицы равен 2 и хеширование идеально, то нам придется иметь дело с двумя списками по *n*/100 элементов в каждом. Как мы видим в таблице, имеется значительная свободы в выборе длины таблицы.

size	*time*	*size*	*time*
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8192	3

Таблица: Зависимость среднего времени поиска (μs) от **hashTableSize**. Сортируются 4096 элементов.

7.2. Реализация

Реализация алгоритма на Си находится в конце раздела. Операторы typedef **T** и **compGT** следует изменить так, чтобы они соответствовали данным, хранимым в массиве. Для работы программы следует также определить **hashTableSize** и отвести место под **hashTable**. В хеш-функции **hash** использован метод деления. Функция **insertNode** отводит память под новый узел и вставляет его в таблицу. Функция **deleteNode** удаляет узел и освобождает память, где он располагался. Функция **findNode** ищет в таблице заданный ключ.

```
typedef int T;                      /* type of item to be sorted */
#define compEQ(a,b) (a == b)

typedef struct Node_ {
    struct Node_ *next;         /* next node */
    T data;                     /* data stored in node */
} Node;

typedef int hashTableIndex;

Node **hashTable;
int hashTableSize;

hashTableIndex hash(T data) {

   /************************************
```

```
    *  hash function applied to data  *
    ***********************************/

    return (data % hashTableSize);
}

Node *insertNode(T data) {
    Node *p, *p0;
    hashTableIndex bucket;

   /************************************************
    *  allocate node for data and insert in table  *
    ************************************************/

    /* insert node at beginning of list */
    bucket = hash(data);
    if ((p = malloc(sizeof(Node))) == 0) {
        fprintf (stderr, "out of memory (insertNode)\n");
        exit(1);
    }
    p0 = hashTable[bucket];
    hashTable[bucket] = p;
    p->next = p0;
    p->data = data;
    return p;
}

void deleteNode(T data) {
    Node *p0, *p;
    hashTableIndex bucket;

   /********************************************
    *  delete node containing data from table  *
    ********************************************/

    /* find node */
    p0 = 0;
    bucket = hash(data);
    p = hashTable[bucket];
    while (p && !compEQ(p->data, data)) {
        p0 = p;
        p = p->next;
    }
    if (!p) return;

    /* p designates node to delete, remove it from list */
    if (p0)
        /* not first node, p0 points to previous node */
        p0->next = p->next;
    else
        /* first node on chain */
        hashTable[bucket] = p->next;

    free (p);
}

Node *findNode (T data) {
    Node *p;

   /*******************************
    *  find node containing data  *
```

```
    ******************************/

    p = hashTable[hash(data)];
    while (p && !compEQ(p->data, data))
        p = p->next;
    return p;
}
```

Лабораторные работы

Создать исполнитель с указанным типом данных, явно отображающий систему предписаний исполнителя. Все элементы предписания должны отображаться в течение всего процесса выполнения программы. Предписания исполнителя должны быть созданы в виде отдельных процедур и/или функций. При невозможности выполнения какого-либо предписания генерируется результат "отказ работы" с указанием ошибки.

Отчет по лабораторной работе должен содержать:

- титульный лист, на котором привести данные о кафедре и институте имя студента, группу, номер варианта, город, год.
- текст задания, содержание отчета;
- теоретическую часть исполнителя;
- обобщенную блок-схему алгоритма программы работы исполнителя;
- описание входные и выходные типов данных процедур, функций и основной программы;
- результаты работы программы на тестовых примерах;
- описание случаев отказа работы исполнителя и реакцию созданной программной системы на них.

Варианты первого задания приведены в таблице 1, второго — в таблице 2.

Таблица 1 – Варианты заданий

Тип данных	Стек	Дек	Очередь	Последовательность
Байт	1	2	3	4
Слово	5	6	7	8
Двойное слово	9	10	11	12
Символ	13	14	15	16
Строка (до 255 символов)	17	18	19	20

Таблица 2 – Варианты заданий

Тип данных	Множество	Нагруженное множество	Л1 список	Л2 список
Байт	20	19	18	17
Слово	16	15	14	13
Двойное слово	12	11	10	9
Символ	8	7	6	5
Строка (до 255 символов)	4	3	2	1

Контрольные работы

Контрольная 1

Пояснение:
1. Задачи:
   1) Работа с матрицами.
   2) Сортировка.
   3) Деревья.
   4) Динамические структуры.
2. Необходимо создать алгоритм (представить в виде блок-схемы) работы. В алгоритме следует обратить внимание на поставленную задачу. Все остальное указывать в обобщенном виде.
3. В тех задачах, где не указаны способы заполнения матрицы или другой структуры данных, заполнение производить случайным образом (не используя ввод с клавиатуры) или из файла. Обратить внимание, чтобы были созданы ситуации, которые оговорены в условии задачи, например, отрицательные числа.
4. Деревья реализовать с помощью матрицы смежности.
5. Динамические структуры реализовать с помощью вектора (массива).
6. Сделать отчет по ГОСТ ТУСУР.

Вариант 1

1. Дана действительная квадратная матрица $[a_{ij}]_{i,j=1,...,n}$. Получить две квадратные матрицы $[b_{ij}]_{i,j=1,...,n}$, $[c_{ij}]_{i,j=1,...,n}$, для которых

$$b_{ij} = \begin{cases} a_{ij} & \text{при } j \geq 1, \\ a_{ij} & \text{при } j < i, \end{cases} \qquad c_{ij} = \begin{cases} a_{ij} & \text{при } j < i, \\ -a_{ij} & \text{при } j \geq i. \end{cases}$$

2. Сортировка вставками просто массив.
3. Задано дерево. Задать обход дерева снизу вверх.
4. Реализовать стек. Даны целые числа $a_1, ..., a_{50}$. Получить сумму тех чисел данной последовательности, которые
   а) кратны 5;
   б) нечётны и отрицательны;

Вариант 2

1. Даны натуральное число n, действительная матрица размера n x 9. Найти:
среднее арифметическое:
   а) каждого из столбцов;

б) каждого из столбцов, имеющих чётные номера.
2. Обменная сортировка просто массив
3. Задано дерево. Вставить элемент.
4. Реализовать очередь. Даны натуральное число n, целые числа a_1, ..., a_n. Найти количество и сумму тех членов данной последовательности, которые делятся на 5 и не делятся на 7.

Вариант 3
1. Дана действительная матрица размера nxm, в которой не все элементы равны нулю. Получить новую матрицу путём деления всех элементов данной матрицы на её наибольший по модулю элемент.
2. Реализовать сортировку методом Шелла просто массив.
3. Задано дерево. Найти максимальный элемент.
4. Реализовать дек. Даны натуральное число n, действительные числа $a_1,...,a_n$. В последовательности $a_1,...,a_n$ все отрицательные члены увеличить на 0.5, а все неотрицательные заменить на 0.1.

Вариант 4
1. Дана действительная квадратная матрица порядка 12. Заменить нулями все её элементы, расположенные на на главной диагонали и выше неё.
2. Сортировка посредством выбора матрицы по столбцам.
3. Задано дерево. Найти сумму всех четных элементов.
4. Реализовать дек. Даны натуральное число n, действительные числа $x_1,...,x_n$. В последовательности $x_1,...,x_n$ все члены, меньшие двух, заменить нулями. Кроме
того, получить сумму членов, принадлежащих отрезку [3,7], а также число таких членов.

Вариант 5
1. Даны действительные числа $x_1,...,x_8$. Получить действительную квадратную матрицу порядка 8:

а) $\begin{bmatrix} x_1 x_2 \dots x_8 \\ x_1^2 x_2^2 \dots x_8^2 \\ \dots\dots\dots\dots \\ x_1^8 x_2^8 \dots x_8^8 \end{bmatrix}$; б) $\begin{bmatrix} 1 \;\; 1 \dots 1 \\ x_1 \;\; x_2 \dots x_8 \\ \dots\dots\dots\dots \\ x_1^7 \;\; x_2^7 \dots x_8^7 \end{bmatrix}$

2. Метод подсчета матрицы по строкам.

3. Задано дерево. Обменять значения полей для ключей 8 и 3.
4. Реализовать очередь. Даны натуральное число n, целые числа $a_1,...,a_n$. Получить сумму положительных и число отрицательных членов последовательности $a_1,...,a_n$.

Вариант 6

1. Дана действительная матрица размера nxm. Определить числа $b_1,...,b_m$, равные соответственно:
   а) суммам элементов строк;
   б) произведениям элементов строк;
   в) наименьшим значениям элементов строк;
   г) значениям средних арифметических элементов строк;
   д) разностям наибольших и наименьших значений элементов строк.
2. Сортировка вставками матрицы по столбцам.
3. Найти значения самого левого и самого правого элементов.
4. Реализовать стек. Даны натуральное число n, целые числа $a_1,...,a_n$. Заменить все большие семи члены последовательности $a_1, ..., a_n$ числом 7. Вычислить количество таких членов.

Вариант 7

1. Все элементы с наибольшим значением в данной целочисленной квадратной матрице порядка 10 заменить нулями.
2. Отсортировать методом Шелла сумму по строкам (в каждой).
3. Задано дерево. Найти первый элемент (значение поля) больше 12.
4. Реализовать очередь. Даны целые числа $a_1,...,a_{50}$. Получить последовательность $b_1,...,b_{50}$, которая отличается от исходной тем, что все нечётные члены удвоены.

Вариант 8

1. Дана действительная матрица размера $m \times n$. Найти среднее арифметическое наибольшего и наименьшего значений её элементов.
2. Сортировка вставками строки по диагональным элементам и отсортировать по этим элементам.
3. Задано сбалансированное дерево. Найти произведение правой ветви.
4. Реализовать стек. Даны натуральное число n, действительные числа $a_1,...,a_n$. Верно ли, что отрицательных членов в последовательности $a_1,...,a_n$ больше, чем положительных?

Вариант 9

1. Дана действительная матрица размера nxm. Найти сумму наибольших значений элементов её строк.
2. Обменная сортировка матрицы по столбцам.
3. Задано дерево. Задать обход сверху вниз.
4. Реализовать очередь. Даны натуральное число n, действительные числа $a_1,...,a_n$. Получить max(|a1|,...,|an|). .

Вариант 10

1. В данной действительной матрице размера 6x9 поменять местами строку, содержащую элемент с наибольшим значением, со строкой, содержащей элемент с наименьшим значением. Предполагается, что эти элементы единственны.
2. Реализовать метод Шелла для матрицы по столбцам.
3. Поменять местами элементы с четными и нечетными номерами.
4. Реализовать дек. Дано натуральное число n. Найти наибольшее среди чисел (k=1,...,n), а также сумму всех этих чисел.

Вариант 11

1. Дана действительная матрица размера nxm, все элементы которой различны. В каждой строке выбирается элемент с наименьшим значением, затем среди этих чисел выбирается наибольшее. Указать индексы элемента с найденным значением.
2. Сортировка посредством выбора строки по диагональным элементам и отсортировать по этим элементам.
3. Задано дерево. Найти минимальный элемент.
4. Реализовать стек. Дано натуральное число n. Выбросить из записи числа n цифры 0 и 5, оставив прежним порядок остальных цифр. Например, из числа 59015509 должно получиться 919.

Вариант 12

1. Дана действительная матрица размера nxm. Получить последовательность $b_1,...,b_n$, где b_k - это
   а) наибольшее из значений элементов k-й строки;
   б) сумма наибольшего и наименьшего из значений элементов k-й строки;
   в) число отрицательных элементов в k-й строке;
2. метод Шелла строки по диагональным элементам и отсортировать по этим элементам.
3. Найти значение элемента самого правого узла.
4. Реализовать очередь. Пусть $x_1 = 0.3$; $x_2 = -0.3$; $x_i = i + \sin(x_{i-2})$, i = 3,4... Среди $x_1,...,x_{100}$ найти ближайшее к какому-нибудь целому.

Вариант 13

1. Дана целочисленная квадратная матрица порядка n. Найти номера строк:
   а) все элементы которых - нули;
   б) элементы в каждой из которых одинакова;
   в) все элементы которых чётны;
   г) элементы каждой из которых образуют монотонную последовательность (монотонно убывающую или монотонно возрастающую);
   д) элементы которых образуют симметричные последовательности (палиндромы).
2. Сортировка посредством выбора матрицы по строкам.
3. Задано дерево. Задать обход дерева снизу вверх.
4. Реализовать стек. Пусть $a_o = \cos^2 1$; $a_i = -\sin^2 1$; $a_k = 2a_{k-1} - a_{k-2}$, $k = 2,3,...$ Найти сумму квадратов тех чисел $a_1,...,a_{100}$, которые не превосходят двух.

Вариант 14

1. Дана действительная квадратная матрица порядка 10. В строках с отрицательным элементом на главной диагонали найти:
   а) сумму всех элементов;
   б) наибольший из всех элементов.
2. сортировка посредством выбора матрицы по столбцам
3. Задано сбалансированное дерево. Подсчитать сумму элементов левой ветви.
4. Реализовать очередь. Даны натуральное число n, действительные числа $a_1,...,a_n$. В последовательности $a_1,...,a_n$ определить число соседств:
   а) двух положительных чисел;
   б) двух чисел разного знака;
   в) двух чисел одного знака, причём модуль первого числа должен быть больше модуля второго числа.

Вариант 15

1. Даны натуральное число n, действительная квадратная матрица порядка n. Построить последовательность $b_1,...,b_n$ из нулей и единиц, в которой b_i = 1 тогда и только тогда, когда в i-й строке матрицы есть хотя бы один отрицательный элемент.
2. Сортировка вставками просто массива.
3. Найти нулевые элементы и заменить их на значение максимального элемента.
4. Реализовать дек. Даны целые числа $c_1,...,c_{95}$. Имеются ли в последовательности $c_1,...,c_{95}$:
   а) два идущих подряд нулевых члена;
   б) три идущих подряд нулевых члена?

Вариант 16

1. Дана действительная квадратная матрица порядка 9. Вычислить сумму тех из её элементов, расположенных на главной диагонали и выше её, которые превосходят по величине все элементы, расположенные ниже главной диагонали. Если на главной диагонали и выше неё нет элементов с указанным свойством, то ответом должно служить сообщение об этом.
2. Реализовать метод Шелла для матрицы по строкам.
3. Задано дерево. Удалить запись.
4. Реализовать стек. Даны натуральное число n, действительные числа $y_1,...,y_n$. Найти:

$$\max(|z_1|, \ldots, |z_n|), \text{ где } z_i = \begin{cases} y_i & \text{при } |y_i| \le 2, \\ 0.5 & \text{в противном случае;} \end{cases}$$

Вариант 17

1. Таблица футбольного чемпионата задана квадратной матрицей порядка n, в которой все элементы, принадлежащие главной диагонали, равны нулю, а каждый элемент, не принадлежащий главной диагонали, равен 2,1 или 0 (число очков, набранных в игре: 2 - выигрыш, 1 - ничья, 0 - проигрыш).

   а) Найти число команд, имеющих больше побед, чем поражений.

   б) Определить номера команд, прошедших чемпионат без поражений.

   в) Выяснить, имеется хотя бы одна команда, выигравшая более половины игр.
2. Сортировка посредством выбора суммы по строкам (в каждой).
3. Задано дерево. Задать обход справа налево.
4. Реализовать очередь. Даны натуральное число n, действительные числа $y_1,...,y_n$. Найти:

$(\sqrt{z_1} - z_2)^2 + \ldots + (\sqrt{z_n} - z_n)^2$,

г) $(\sqrt{z_1} - z_2)^2 + \ldots + (\sqrt{z_n} - z_n)^2$,

$$\text{где } z_i = \begin{cases} y_i & \text{при } 0 < y_i \le 15, \\ 2.7 & \text{в противном случае;} \end{cases}$$

Вариант 18

1. Дана символьная квадратная матрица порядка 10. Заменить буквой **а** все её элементы, расположенные выше главной диагонали.
2. Сортировка посредством выбора просто массив
3. Задано дерево. Заменить значения левой ветви на значения правой.
4. Реализовать стек. Даны целые числа $a_1, a_2, \ldots$ Известно, что $a_1 > 0$ и что

среди a_2, a_3 и т.д. есть хотя бы одно отрицательное число. Пусть $a_1,...,a_n$ – члены данной последовательности, предшествующие первому отрицательному члену (n заранее неизвестно). Получить: $\max(a^2_1, \ldots, a^2_n)$;

Вариант 19

1 Будем называть соседями элемента с индексами i, j некоторой матрицы такие элементы этой матрицы, соответствующие индексы которых отличаются от i и j не более чем на единицу. Для данной целочисленной матрицы $[a_{ij}]_{i=1,...,n;j=1,...,m}$ найти матрицу из нулей и единиц $[b_{ij}]_{i=1,...,n;j=1,...m}$, элемент которой b_{ij} равен единице, когда

а) все соседи a_{ij} меньше самого a_{ij};

б) все соседи a_{ij} и само a_{ij} равны нулю;

в) среди соседей a_{ij} есть не менее двух совпадающих с a_{ij}.

2. Реализовать метод Шелла для матрицы по строкам.
3. Задано дерево. Подсчитать сумму всех элементов.
4. Реализовать очередь. Даны натуральные числа m, n ($m \neq 0$, $n \neq 0$). Получить все их общие делители (положительные и отрицательные).

Вариант 20

1. Даны натуральное число n, действительная квадратная матрица порядка n. Построить последовательность $b_1,...,b_n$ из нулей и единиц, в которой $b_i = 1$ тогда и только тогда, когда в i-й строке матрицы есть хотя бы один отрицательный элемент.
2. Реализовать обменную сортировку строки по диагональным элементам и отсортировать по этим элементам
3. Задано дерево. Задать обход слева направо.
4. Реализовать дек. Даны натуральное число n, действительные числа $a_1,...,a_n$. Получить $\max(|a1|,...,|an|)$.

Вариант 21

1. Даны действительная квадратная матрица порядка n, натуральные числа i,j
($1 \leq i \leq n$, $1 \leq j \leq n$). Из матрицы удалить i-ю строку и j-й столбец.
2. Сортировка вставками суммы по строкам (в каждой).
3. Задано дерево. Найти произведение всех элементов.
4. Реализовать очередь. Даны целые числа a_1, ..., a_{50}. Получить сумму тех чисел данной последовательности, которые

а) кратны 6;

б) чётны и отрицательны;

Вариант 22

1. Дана действительная квадратная матрица порядка n. Получить $x_1x_n + x_2x_{n-1} + ... + x_nx_1$, где x_k - наибольшее значение элементов k-й строки данной матрицы.
2. Реализовать метод подсчета строки по диагональным элементам и отсортировать по этим элементам.
3. Задано дерево. Определить сумму максимального и минимального элементов.
4. Реализовать стек. Даны целые числа p, q, a_1, ..., a_{67} ($p > q \geq 0$). В последовательности a_1, ..., a_{67} заменить нулями члены, модуль которых при делении на p даёт в остатке q.

Вариант 23

1. Дана действительная квадратная матрица порядка 9. Получить целочисленную квадратную матрицу того же порядка, в которой элемент равен единице, если соответствующий ему элемент исходной матрицы больше элемента, расположенного в его строке на главной диагонали, и равен нулю в противном случае.
2. Сортировка посредством выбора сумму по строкам (в каждой).
3. Задано дерево. Определить сумму самого левого и самого правого элементов.
4. Реализовать дек. Даны натуральное число n, действительные числа $a_1,...,a_n$. В последовательности $a_1,...,a_n$ все неотрицательные члены, не принадлежащие отрезку [1,2], заменить на единицу. Кроме того, получить число отрицательных членов и число членов, принадлежащих отрезку [1,2].

Вариант 24

1. Дана целочисленная квадратная матрица порядка 8. Найти наименьшее из значений элементов столбца, который обладает наибольшей суммой модулей элементов. Если таких столбцов несколько, то взять первый из них.
2. Сортировать методом подсчета матрицу по столбцам.
3. Задано дерево. Определить количество отрицательных элементов.
4. Реализовать стек. Даны целые числа $a_1,...,a_{50}$. Получить последовательность $b_1,...,b_{50}$, которая отличается от исходной тем, что все чётные члены утроены.

Вариант 25

1. Даны натуральное число n, целочисленная матрица $[a_{ij}]_{i=1,2;j=1,...,m}$. Найти сумму тех из элементов a_{2j} (j = 1,...,m), для которых a_{1j} имеет значение наибольшего среди значений $a_{11},a_{12},...,a_{1m}$.
2. Сортировать матрицу методом Шелла матрицу по столбцам.
3. Задано дерево. Определить количество и сумму четных элементов.
4. Реализовать дек. Даны натуральное число n, действительные числа $a_1,...,a_n$. Получить min(|a1|,...,|an|).

Вариант 26

1. Все элементы с наибольшим значением в данной целочисленной квадратной матрице порядка 10 заменить единицами.
2. Реализовать обменную сортировку суммы по строкам (в каждой) и потом отсортировать
3. Задано дерево. Поменять значения самого левого и самого правого элементов.
4. Реализовать стек. Дано натуральное число n. Выбросить из записи числа n цифры 1 и 2, оставив прежним порядок остальных цифр. Например, из числа 59015509 должно получиться 919.

Вариант 27

1. Даны натуральное число m, целые числа $a_1,...,a_m$ и целочисленная квадратная матрица порядка m. Строку с номером i матрицы назовём отмеченной, если ai > 0, и неотмеченной в противном случае.

   а) Нужно все элементы, расположенные в отмеченных строках матрицы, преобразовать по правилу: отрицательные элементы заменить на -1, положительные - на 1, а нулевые оставить без изменения.

   б) Подсчитать число отрицательных элементов матрицы, расположенных в отмеченных строках.
2. Отсортировать методом подсчета сумму по строкам (в каждой.
3. Задано дерево. Изменить значение на сумму значений предков.
4. Реализовать очередь. Даны целые числа $c_1,...,c_{95}$. Имеются ли в последовательности $c_1,...,c_{95}$:

   а) два идущих подряд нулевых члена;

   б) три идущих подряд нулевых члена?

Вариант 28

1. Дано натуральное число n. Выяснить, сколько положительных элементов
содержит матрица $[a_{ij}]_{i,j=1,...,n}$, если

а) $a_{ij} = \sin(i + j/2)$;
б) $a_{ij} = \cos(i^2 + n)$;
в) $a_{ij} = \sin\left(\frac{i^2 - j^2}{n}\right)$.

2. Сортировка вставками матрицы по строкам.
3. Задано дерево. Изменить значение на сумму значений потомков.
4. Реализовать дек. Даны целые числа $a_1, ..., a_{50}$. Получить сумму тех чисел данной последовательности, которые
   а) кратны 7;
   б) отрицательны.

Вариант 29

1. Получить действительную матрицу $[a_{ij}]_{i,j=1,...,7}$, первая строка которой задаётся формулой $a_{1j} = 2j + 3$ (j=1,...,7), вторая строка задается формулой

$a_{2j} = j - \frac{3}{2 + 1/j}$ (j = 1, …, 7), а каждая следующая строка есть сумма двух предыдущих.

2. Реализовать обменную сортировку матрицы по строкам.
3. Задано дерево. Изменить значение на разность значений левого и правого сына.
4. Реализовать очередь. Даны целые числа p, q, $a_1, ..., a_{27}$ $(p > q \geq 0)$. В последовательности $a_1, ..., a_{67}$ заменить нулями члены, модуль которых при делении на p+1 даёт в остатке q-1.

Вариант 30

1. Дано натуральное число n. Получить действительную матрицу $[a_{ij}]_{i,j=1,...,n}$, для которой

а) $a_{ij} = \frac{1}{i + j}$;

б) $$a_{ij} = \begin{cases} \sin(i + j) & \text{при } i < j, \\ 1 & \text{при } i = j, \\ \arcsin\frac{i + j}{2i + 3j} & \text{в остальных случаях.} \end{cases}$$

2. Сортировать методом подсчета просто массив.

3. Задано дерево. Найти среднее геометрическое значений дерева.
4. Реализовать стек. Даны натуральное число n, действительные числа a_1, ..., a_n. Получить удвоенную сумму всех положительных членов последовательности a_1, ..., a_n.

Контрольная 2

1. Выбрать задачи предыдущих вариантов:
   1) Деревья.
   2) Динамические структуры.
2. Необходимо создать алгоритм (представить в виде блок-схемы) работы. В алгоритме следует обратить внимание на поставленную задачу. Все остальное указывать в обобщенном виде.
3. В тех задачах, где не указаны способы заполнения матрицы или другой структуры данных, заполнение производить случайным образом (не используя ввод с клавиатуры) или из файла. Обратить внимание, чтобы были созданы ситуации, которые оговорены в условии задачи, например, отрицательные числа.
4. Деревья реализовать с помощью динамического распределения памяти указателей.
5. Динамические структуры реализовать с помощью Л2-списков на указателях.
6. Сделать отчет по ГОСТ ТУСУР.

ПРИЛОЖЕНИЯ

Таблицы кодировок русского языка

Символ	DOS	Windows	КОИ-8
А	128	192	225
Б	129	193	226
В	130	194	247
Г	131	195	231
Д	132	196	228
Е	133	197	229
Ё	240	168	179
Ж	134	198	246
З	135	199	250
И	136	200	233
Й	137	201	234
К	138	202	235
Л	139	203	236
М	140	204	237
Н	141	205	238
О	142	206	239
П	143	207	240
Р	144	208	242
С	145	209	243
Т	146	210	244
У	147	211	245
Ф	148	212	230
Х	149	213	232
Ц	150	214	227
Ч	151	215	254
Ш	152	216	251
Щ	153	217	253
Ъ	154	218	255
Ы	155	219	249
Ь	156	220	248
Э	157	221	252
Ю	158	222	224
Я	159	223	241

Символ	DOS	Windows	КОИ-8
а	160	224	193
б	161	225	194
в	162	226	215
г	163	227	199
д	164	228	196
е	165	229	197
ё	241	184	163
ж	166	230	214
з	167	231	218
и	168	232	201
й	169	233	202
к	170	234	203
л	171	235	204
м	172	236	205
н	173	237	206
о	174	238	207
п	175	239	208
р	224	240	210
с	225	241	211
т	226	242	212
у	227	243	213
ф	228	244	198
х	229	245	200
ц	230	246	195
ш	232	248	219
щ	233	249	221
ъ	234	250	223
ы	235	251	217
ь	236	252	216
э	237	253	220
ю	238	254	192
я	239	255	209

Исходные тексты программ

Приведенные примеры исходных текстов программ никоим образом не претендуют на совершенный стиль программирования. Кроме того, в некоторых программах есть лишние элементы. Читателям предлагается усовершенствовать программы.

Работа с матрицами

```
uses crt;
const n=8;{размерность матрицы}
var m : array [1..n,1..n] of integer;{задаем матрицу nxn типа целое}
    i,j : integer;{}
    min : integer;{}

procedure show;
{отображает матрицу}
var i,j : integer;
begin
for i:=1 to n do begin
 for j:=1 to n do write(m[i,j]:4);{выводим строку}
 writeln;{переход на новую строку}
end;
end;

BEGIN
clrscr; {очистка экрана}
randomize;{запускаем датчик случайных чисел}
for i:=1 to n do
 for j:=1 to n do m[i,j]:=random(100)-50;{задаем матрицу случайным образом}

show;

min:=m[1,1];
for i:=1 to n do
 for j:=1 to n do
 if min>m[i,j] then min:=m[i,j];{находим минимальный}
writeln('минимальное значение матрицы',min:4)

END.
```

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
    {
        const n=9;
        int m[n][n]; //задаем матрицу
        randomize(); //запускаем датчик случайных чисел
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
               m[i][j]=rand()%100;
               printf("%d \t", m[i][j]); // выводим на экран
            }
            printf("\n"); // перевод на новую строку
        }
```

```
        int min=m[1][1]; // поиск минимального
        for(i=0; i<n; i++)
        for(int j=0; j<n; j++){
            if (min>m[i][j]) min=m[i][j];
        };
        printf("%d\n", min);
    return 0;
}
```

Работа с деревьями

```
{пример работы с деревьями}
uses crt;
type ptree=^tree;
     tree = record
        d : integer; {данные}
       pl : ptree; {указатель left}
       pr : ptree; {указатель rigth}
      end; {tree}
var
  p_root,
  p_current, {указатель на текущий элемент}
  p_temp :  ptree; {временный указатель}
  i,j : integer;
{-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure show_tree (pt:ptree);
 {показывает список top-down}
begin
 if pt<>nil then begin
  writeln(pt^.d:6);
  show_tree(pt^.pl);
  show_tree(pt^.pr);
 end; {while}
end; {show_tree}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure add_element (d:integer; f:boolean; pt : ptree);
 {вставляет элемент за указателем}
begin
 if f then begin {left}
  if pt^.pl=nil then begin
   new(pt^.pl);
   pt^.pl^.d:=d;
   pt^.pl^.pl:=nil;
   pt^.pl^.pr:=nil;
  end;{if pl}
 end {left}
  else begin {rigth}
   if pt^.pr=nil then begin
    new(pt^.pr);
    pt^.pr^.d:=d;
    pt^.pr^.pl:=nil;
    pt^.pr^.pr:=nil;
   end; {if pr}
  end; {rigth}
end; {add_element}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure delete_element (pt : ptree; f : boolean);
 {удаляет текущий элемент}
begin
 if f then begin {left}
```

```
  if (pt^.pl^.pl=nil) and (pt^.pl^.pl=nil) then begin
   dispose(pt^.pl);
   pt^.pl:=nil;
  end;
 end {left}
  else begin {rigth}
   if (pt^.pr^.pl=nil) and (pt^.pr^.pl=nil) then begin
    dispose(pt^.pr);
    pt^.pr:=nil;
   end;
  end; {rigth}
end; {delete_element}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-}
BEGIN {main program}
 clrscr;
 new (p_root);
 p_root^.d:=1;
 p_root^.pl:=nil;
 p_root^.pr:=nil;
 add_element (2, true , p_root);
 add_element (3, false, p_root);
 p_temp:=p_root^.pl;
 add_element (4, true , p_temp);
 add_element (5, false, p_temp);
 p_temp:=p_root^.pr;
 add_element (6, true , p_temp);
 add_element (7, false, p_temp);
 show_tree(p_root);
 delete_element(p_temp, true);
 writeln;
 show_tree(p_root);
END.
```

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <iostream.h>


typedef struct tree {
    int d;
    struct tree *pl;
    struct tree *pr;
};


void show_tree (tree *pt) {
//показывает список top-down
    if (pt != NULL) {
        cout << (*pt).d)<< "\n";
        show_tree ((*pt).pl);
        show_tree ((*pt).pr);
    }
}

void add_element (int d, int f, tree *pt) {
// вставляет элемент за указателем
   if (pt != NULL) {
       if (f) { //left
           if ((*pt).pl == NULL) {
```

```
                (*pt).pl = new tree;
                (*(*pt).pl).d = d;
                (*(*pt).pl).pl = NULL;
                (*(*pt).pl).pr = NULL;
            } //left
        }
        else { //rigth
            if ((*pt).pr == NULL) {
                (*pt).pr = new tree;
                (*(*pt).pr).d = d;
                (*(*pt).pr).pl = NULL;
                (*(*pt).pr).pr = NULL;
            }
        }
    }
}
void delete_element (tree *pt, int f) {
//удаляет текущий элемент
    if (pt != NULL) {
        if (f) { //left
            if (((*(*pt).pl).pl == NULL) && ((*(*pt).pl).pl == NULL)) {
                delete (*pt).pl;
                 (*pt).pl = NULL;
            }
        } //left
        else { //rigth
            if (((*(*pt).pr).pl == NULL) && ((*(*pt).pr).pl == NULL)) {
                delete (*pt).pr;
                 (*pt).pr = NULL;
            }
        }
    }
}

int main (void)
    {
    tree *a;
    a = new tree;
    (*a).d = 1;
    (*a).pl = NULL;
    (*a).pr = NULL;
    add_element (2, 1, a);
    add_element (3, 0, a);
    add_element (4, 1, (*a).pl);
    add_element (5, 0, (*a).pl);
    add_element (6, 1, (*a).pr);
    add_element (7, 0, (*a).pr);
    cout << (*a).d <<"\n";

    cout << "---------------\n";
    show_tree (a);

    delete_element ((*a).pl, 0);

    cout << "---------------\n";
    show_tree (a);

    return 0;
}
```

Работа со списками

```
{пример работы двунаправленного списка}
uses crt;
type
  l2_element=record
          d : integer;{данные}
         p1 : pointer;{указатель на следующий}
         p2 : pointer;{указатель на предыдущий}
  end;
var
  l2_first, {указатель на первый элемент}
  l2_current, {указатель на текущий элемент}
  l2_temp :  ^l2_element; {временный указатель}
  i,j : integer;
{-=-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure show_list;
 {показывает список}
begin
 l2_temp:=l2_first;
 while l2_temp^.p1<>nil do begin
{  writeln(l2_temp^.d:6,' ',seg(l2_temp^),' ',ofs(l2_temp^));}
  writeln(l2_temp^.d:3);
  l2_temp:=l2_temp^.p1;
 end;{while}
 if l2_temp<>nil then writeln(l2_temp^.d:3);
end;{show_list}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure add_element (d : integer);
 {вставляет элемент за указателем}
var p : pointer;
begin
 new (l2_temp);
{ writeln(i,' ',seg(l2_temp^),' ',ofs(l2_temp^),' ',seg(l2_current^),'
',ofs(l2_current^));}
 l2_temp^.d:=i;
 l2_temp^.p1:=l2_current^.p1;
 l2_temp^.p2:=l2_current;
 l2_current^.p1:=l2_temp;
 p:=l2_temp;
 l2_temp:=l2_temp^.p1;
 l2_temp^.p2:=p;
{ writeln(i,' ',seg(p^),' ',ofs(p^));}
end;{add_element}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-=-}
procedure delete_element ;
 {удаляет текущий элемент}
begin
 if l2_current^.p1<>nil then begin
  l2_temp:=l2_current^.p1;
  l2_temp^.p2:=l2_current^.p2;
 end;
 if l2_current^.p2<>nil then begin
  l2_temp:=l2_current^.p2;
  l2_temp^.p1:=l2_current^.p1;
 end;
 release (l2_current);
end;{add_element}
{-=-=-=-=-=-=-=-=-=-=-=-=-=-=-}
BEGIN
 new (l2_first);
```

```
 l2_first^.d:=0;
 l2_first^.p1:=nil;
 l2_first^.p2:=nil;

 l2_current:=l2_first;
 for i:=1 to 5 do add_element(i);
 show_list;

 l2_current:=l2_first^.p1;
 delete_element;
 show_list;

END.

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <iostream.h>

typedef struct l2_element {
    int d;
    struct l2_element *p1;
    struct l2_element *p2;
};

void show_list (l2_element *pt) {
//показывает список
    l2_element *pp;
    pp = pt;
    while ((*pp).p1 != NULL) {
        cout << (*pp).d << " ";
        pp = (*pp).p1;
    }
    if (pp != NULL) cout << (*pp).d << "\n";
}

void add_element (int d, l2_element *pt) {
// вставляет элемент за указателем
    if (pt != NULL) {
        l2_element *pp, *t;
        pp = new l2_element;
        (*pp).d = d;
        (*pp).p1 = (*pt).p1;
        (*pp).p2 = pt;
        (*pt).p1 = pp;
        t = pp;
        pp = (*pp).p1;
        (*pp).p1 = t;
    }
}

void delete_element (l2_element *pt) {
//удаляет текущий элемент
    if (pt != NULL) {
        l2_element *pp;
        if ((*pt).p1 != NULL) {
            pp = (*pt).p1;
            (*pp).p2 = (*pt).p2;
        }
        if ((*pt).p2 != NULL) {
            pp = (*pt).p2;
```

```
            (*pp).p1 = (*pt).p1;
        }
        delete pt;
    }
}


int main (void)
    {
    l2_element *a, *c;
    a = new l2_element;
    (*a).d = 0;
    (*a).p1 = NULL;
    (*a).p2 = NULL;

    c = a;
    for (int i = 1; i < 6; i++) {
        add_element (i, c);
        c = (*c).p1;
    }
    c = a;
    show_list (c);

    c = (*a).p1;
    delete_element (c);

    c = a;
    show_list (a);

    return 0;
}
```

Пример оформления контрольной работы.

1. Титульный лист:

Министерство образования Российской Федерации
Томский государственный университет систем
управления и радиоэлектроники

Кафедра комплексной информационной безопасности
электронно-вычислительных систем

Контрольная работа № 1 по курсу
"Структуры данных и прикладные алгоритмы"
вариант 0

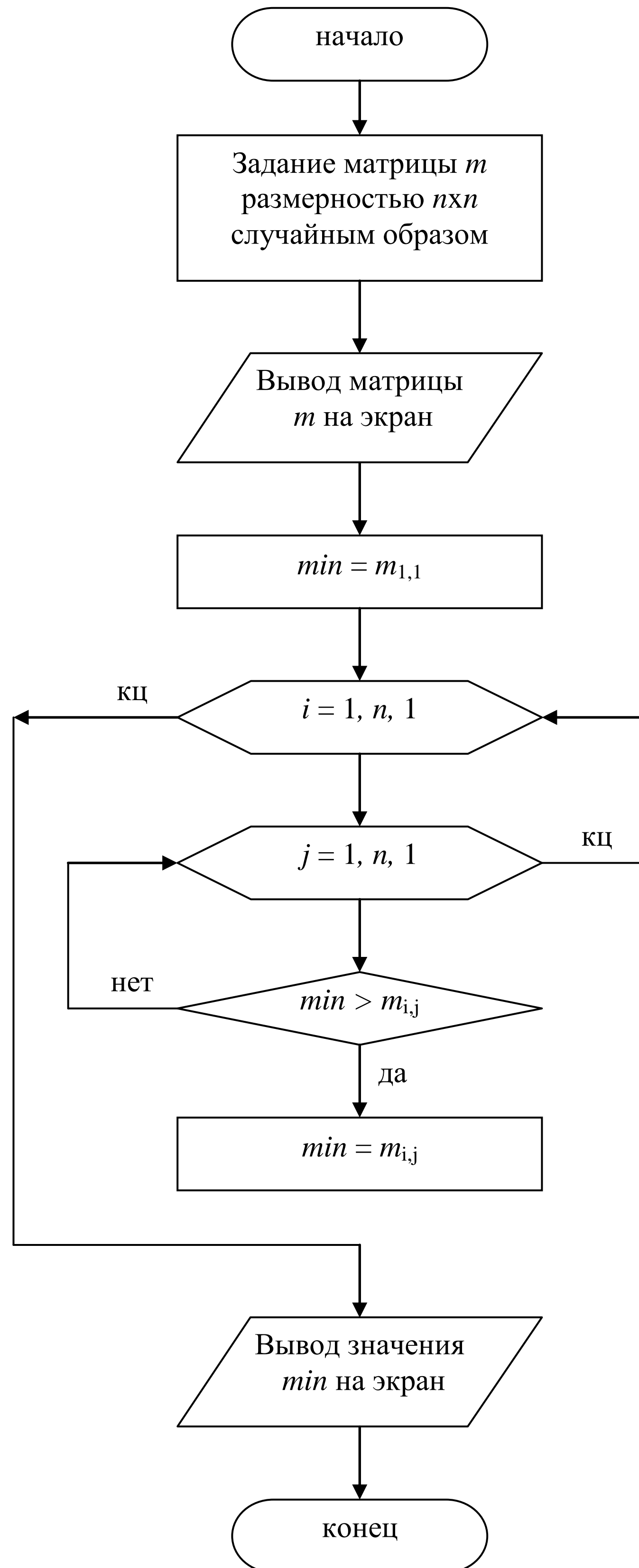
выполнил студент
группы 200
город Томск
Иванов Иван Иванович

проверил
доцент кафедры КИБЭВС
Мещеряков Р.В.

Томск
2002

2. Задание
1. Дана действительная квадратная матрица порядка 8. Найти минимальное значение матрицы.

3. Обобщенная блок-схема алгоритма.



Алгоритм программы состоит из последовательности следующих шагов:
1) Задание матрицы случайным образом.
2) Отображение матрицы на экран.
3) Поиск минимального значения матрицы.
4) Вывод найденного значения матрицы на экран.

Наиболее важная часть: поиск минимального значения.
Вначале предполагаем, что элемент матрицы с индексами 1,1 , т.е. $m_{1,1}$ является самым минимальным элементом. Затем проводится прохождение по всем элементам матрицы (два цикла по индексам). Если находится элемент, который меньше текущего минимального, то текущему минимальному присваивается значение текущего элемента. По окончанию сравнения текущего минимального со всеми элементами матрицы значение текущего минимального будет равно минимальному элементу матрицы.

4. Исходный текст программы.
```
uses crt;
const n=8;{размерность матрицы}
var m : array [1..n,1..n] of integer;{задаем матрицу nxn типа целое}
    i,j : integer;{}
    min : integer;{}

procedure show;
{отображает матрицу}
var i,j : integer;
begin
for i:=1 to n do begin
 for j:=1 to n do write(m[i,j]:4);{выводим строку}
 writeln;{переход на новую строку}
end;
end;

BEGIN
clrscr; {очистка экрана}
randomize;{запускаем датчик случайных чисел}
for i:=1 to n do
 for j:=1 to n do m[i,j]:=random(100)-50;{задаем матрицу случайным образом}

show;

min:=m[1,1];
for i:=1 to n do
 for j:=1 to n do
  if min>m[i,j] then min:=m[i,j];{находим минимальный}
writeln('минимальное значение матрицы',min:4)

END.
```

5. В основной программе используются следующие типы данных:

n=6	Константа, определяющая размерность матрицы. Является входным параметром.
m : array [1..n,1..n] of integer	Матрица целых чисел *m* размерностью *n*x*n* .
i,j : integer	Переменные типа целое для организации индексов матрицы.
min : integer	Переменная типа целое для поиска результата. Является выходным значением.

Процедура show осуществляет отображение матрицы на экран

i,j : integer	Переменные типа целое для организации индексов матрицы.
m []	Используется как глобальная переменная из основной программы. Является входным параметром.

6. Результаты работы программы на тестовых примерах;

```
  -7  30  -8 -40 -33  48 -44 -11
  35 -31  11 -10   7 -22  15 -29
  30  40 -46  46  29  36 -21 -16
  48  36   9   3   8 -17  -2 -43
  40   5  47  -9  44  -6  21 -46
 -22   2 -11  15 -18  48   5  -1
  -6 -49   1  42   8  10  49 -44
 -12  10  22  25  28  25 -39 -25
минимальное значение матрицы -49
```

Вопросы к экзамену

1. Простые структуры данных.
2. Представление целых чисел.
3. Представление вещественных чисел (с фиксированной и плавающей точкой).
4. Логические типы данных.
5. Перечисления.
6. Диапазоны.
7. Кодировки символов.
8. Представление строк в виде строки со счетчиком.
9. Представление строк в виде строки с признаком конца строки.
10. Структура данных стек.
11. Структура данных очередь.
12. Структура данных дек.
13. Структура данных последовательность.
14. Структура данных множество, нагруженное множество.
15. Структура данных однонаправленный список.
16. Структура данных двунаправленный список.
17. Структура данных вектор.
18. Структура данных матрица.
19. Структура данных динамический вектор.
20. Структура данных граф.
21. Представление графов.
22. Структура данных дерево.
23. Представление деревьев.
24. Бинарное дерево.
25. Правило обхода дерева внизу вверх.
26. Правило обхода дерева сверху вниз.
27. Ориентированные графы.
28. Задача нахождения наикратчайшего пути.
29. Сортировка.
30. Условия, при которых возможна сортировка.
31. Внутренняя сортировка.
32. Сортировка посредством выбора.
33. Обменная сортировка.
34. Сортировка вставкой.
35. Сортировка методом подсчета.
36. Метод сортировки Шелла.
37. Внешняя сортировка.
38. Сортировка слиянием.
39. Хранение данных в файлах.

40.Поиск информации. Критерий поиска.
41.Последовательный метод поиска.
42.Двоичный метод поиска.
43.Блочный поиск.
44.Бинарный поиск.
45.Хеширование.

СПИСОК РЕКОМЕНДУЕМЫХ ИСТОЧНИКОВ

1. Абель П. Язык ассемблера для IBM PC и программирование: Пер. с англ. – М.: Высшая школа, 1992 – 477с.,
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир. 1979 – 536с.
3. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. – М.: Издательский дом "Вильямс", 2000. – 384с. : ил.
4. Б. Керниган, Д. Ритчи, А. Фьюер. Язык программирования С. Задачи по языку С. М.: "Финансы и статистика". 1984
5. Бен-Ари М. Языки программирования. Практический сравнительный анализ: Пер. с англ. – М.: Мир, 2000. – 366с., ил.
6. Братко И. Программирование на языке Пролог для искусственного интеллекта: Пер. с англ. – М.: Мир, 1990. – 560с., ил.
7. Брой М. Информатика. Вычислительные структуры и машинно-ориентированное программирование: В 4-х ч. Ч. 2./ Пер. с нем. – М.: Диалог-МИФИ, 1996 –224с.
8. Брой М. Информатика. Основополагающее введение: В 4-х ч. Ч. 1./ Пер. с нем. – М.: Диалог-МИФИ, 1996 –229с.
9. Брой М. Информатика. Структуры систем и системное программирование: В 4-х ч. Ч. 3./ Пер. с нем. – М.: Диалог-МИФИ, 1996 –224с.
10. Вайнер Р., Пинсон Л. С++ изнутри. Пер. с англ. – Киев.: "Диасофт", 1993. – 304с., ил.
11. Джордейн Р. Справочник программиста персональных комьютеров типа IBM PC, XT и AT: Пер. с англ./ Предисловие Н.В. Гайского. – М.: Финансы и статистика, 1991. – 544с.
12. Дьюхарст С., Старк К. Программирование на С++. Пер. с англ. – Киев: "ДиаСофт", 1993. – 272с., ил.
13. Зуев Е.А. Язык программирования Turbo Pascal 6.0. – М.: Унитех, 1992. – 292с.
14. Кнут Д. Искусство программирования: – М.: Наука. 1978.
15. Котов В.Е., Сабельфельд В.К. Теория схем программ – М.: Наука. Гл. ред. физ.-мат. Лит., 1991. – 248с.
16. Кушниренко А.Г., Лебедев Г.В. Программирование для математиков: Уч. пособие для вузов – М.: Наука. 1988. – 384с.
17. Лукас П. С++ под рукой: Пер. с англ. – Киев: "ДиаСофт", 1993. – 176с., ил.
18. Марселлус Д. Программирование экспертных систем на Турбо Прологе: Пер. с англ. / Предисл. С.В. Трубицына. – М.: Финансы и статистика, 1994. – 256с., ил.

19. Нортон П., Сохуэ Д. Язык ассемблера для IBM PC: Пер. с англ. – М.: Изд-во "Компьютер"; Финансы и статистика, 1992. – 352с.
20. Общая теория систем. Пер. с англ. – М.: Мир, 1966, - 188с.
21. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: "ДИАЛОГ-МИФИ", 1994. – 288 с.
22. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера Пер. с англ. – М.: Радио и связь, 1991. – 336с.
23. Соха Дж., Рахмел Д., Холл Д. Изучи сам Visual Basic 5 / Пер. с англ. А.Н. Филимонов. – Мн.: ООО "Попури", 1998. – 320с., ил.
24. Страуструп Б Язык программирования С++. Пер. с англ. – Киев.: "Диасофт", 1993.
25. Тьюринг А. Может ли машина мыслить? – Саратов. Изд-во ГосУНЦ "Колледж", 1999. 100с.
26. Урнов В.А., Климов Д.Ю. Преподавание информацтики в компьютерном классе. – М.: Просвещение, 1990. –206с., ил.
27. Фаронов В.В. Турбо Паскаль (в 3-х книгах). Кн.1. Основы Турбо Паскаля. – М.: Учебно-инженерный центр "МВТУ – ФЕСТО ДИДАКТИК", 1993. – 304с., ил.
28. Фаронов В.В. Турбо Паскаль (в 3-х книгах). Кн.3. Практика программирования. Часть.2. – М.: Учебно-инженерный центр "МВТУ – ФЕСТО ДИДАКТИК", 1993. – 256с., ил.
29. Фаронов В.В. Турбо Паскаль (в 3-х книгах). Кн.3. Практика программирования. Часть.1. – М.: Учебно-инженерный центр "МВТУ – ФЕСТО ДИДАКТИК", 1993. – 304с., ил.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **3**

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ. СТРУКТУРЫ ДАННЫХ **4**

1.1 Способы задания элементов определенного типа. 4
1.2 Стек элементов типа Е 6
1.3 Очередь элементов типа Е 7
1.4 Дек элементов типа Е 8
1.5 Множество элементов типа Е 9
1.6 Нагруженное множество 10
1.7 Последовательность элементов типа Е 10
1.8 Л2-список элементов типа Е 11
1.9 Л1-список элементов типа Е 13
1.10 Вектор элементов типа Е с произвольным доступом 13
1.11 Матрица элементов типа Е с индексами типа И1, И2. 14
1.12 Динамический вектор элементов типа Е 15
1.13 Графы 15
1.14 Деревья 18

2 ПРАКТИЧЕСКАЯ ЧАСТЬ. СТРУКТУРЫ ДАННЫХ **20**

2.1 Общие сведения. 20
2.2 Ассемблер (Assembler) 24
Директивы определения данных 24
Директива DB (db) 24
Директива DW (dw) 27
Директива DD (dd) 29
Директивы эквивалентности и присваивания 30
2.3 Си/Си++ (С/С++) 33
Константы 33
Символьная константа 34
Константное выражение 35
Строчная константа 35
Описания 36
Преобразование типов 37
Инициализация 40
Макроподстановка 41
Указатели и массивы 43
Указатели и адреса 43
Указатели и массивы 45
Многомерные массивы 47
Массивы указателей; указатели указателей 49
Указатели и многомерные массивы 49
Структуры 49
Поля 52
Объединения 54
2.4 Паскаль (Pascal) 57
Идентификаторы 57
Типы 57
Записи 58
Порядковые типы 59
Вещественные типы 60
Строковые типы 61
Символьный тип 62
Булевы типы 62
Массив 63
Указательные типы 63
Файлы 65
Целочисленные типы 66
Переменные 66

Постоянные выражения *66*
Константы *67*
Типизированные константы *67*
Стандартная директива Absolute *72*
2.5 Бейсик (BASIC) 74
2.6 Пролог (Prolog) 75
Объекты данных *75*
Атомы и числа. *75*
Переменные. *77*
Структуры *77*
Списки *78*

3. ДЕРЕВЬЯ. ПРИКЛАДНЫЕ АЛГОРИТМЫ 81

3.1. Основная терминология 81
Порядок узлов *83*
Прямой, обратный и симметричный обходы дерева *84*
Помеченные деревья и деревья выражений *86*
Вычисление "наследственных" данных *88*
3.2. Абстрактный тип данных TREE 88
3.3. Реализация деревьев 91
Представление деревьев с помощью массивов *91*
Представление деревьев с использованием списков сыновей *93*
Представление левых сыновей и правых братьев *96*
3.4. Двоичные деревья 100
Представление двоичных деревьев *102*
Пример: коды Хаффмана *102*
3.5. Реализация двоичных деревьев с помощью указателей 108

4. ОРИЕНТИРОВАННЫЕ ГРАФЫ. ПРИКЛАДНЫЕ АЛГОРИТМЫ 109

4.1. Основные определения 109
4.2. Представления ориентированных графов 111
АТД для ориентированных графов *113*
4.3. Задача нахождения кратчайшего пути 114
Обоснование алгоритма Дейкстры *117*
Время выполнения алгоритма Дейкстры *119*
4.4. Нахождение кратчайших путей между парами вершин 119
Сравнение алгоритмов Флойда и Дейкстры. *122*
Вывод на печать кратчайших путей *123*
Транзитивное замыкание *124*

5. СОРТИРОВКА. ПРИКЛАДНЫЕ АЛГОРИТМЫ 126

5.1. Модель внутренней сортировки 126
5.2. Простые схемы сортировки 127
Сортировка вставками *128*
Сортировка посредством выбора *129*
Временная сложность методов сортировки *130*
Подсчет перестановок *131*
Ограниченность простых схем сортировки *132*
5.3. Быстрая сортировка 133
Временная сложность быстрой сортировки *137*
Время выполнения быстрой сортировки в среднем *138*
Реализация алгоритма быстрой сортировки *142*
5.4. Пирамидальная сортировка 143
Анализ пирамидальной сортировки *146*
5.5. Карманная сортировка 147
Анализ "карманной" сортировки *150*
5.6. Сортировка множеств с большими значениями ключей 151
5.7. Общая поразрядная сортировка 153
Анализ поразрядной сортировки *155*
5.8. Время выполнения сортировок сравнениями 156

Деревья решений *156*
Размер дерева решений *157*
Анализ времени выполнения в среднем *158*
5.9. Порядковые статистики 159
5.10. Вариант быстрой сортировки 159
5.11. Линейный метод нахождения порядковых статистик 160
Случай равенства некоторых значений ключей *163*
5.12. Краткое описание наиболее распространных сортировок 164
Метод выбора *164*
Метод обмена (метод «пузырька») *164*
Метод вставок *165*
Метод подсчета *165*
Метод сортировки Шелла *166*

6. АЛГОРИТМЫ ДЛЯ ВНЕШНЕЙ ПАМЯТИ. ПРИКЛАДНЫЕ АЛГОРИТМЫ 167

6.1. Модель внешних вычислений 167
Стоимость операций со вторичной памятью *168*
6.2. Внешняя сортировка 169
Сортировка слиянием *169*
Ускорение сортировки слиянием *173*
Минимизация полного времени выполнения *173*
Многоканальное слияние *174*
Многофазная сортировка *175*
Когда скорость ввода-вывода не является „узким местом" *177*
Схема с шестью входными буферами *179*
Схема с четырьмя буферами *180*
6.3. Хранение данных в файлах 183
Простая организация данных *184*
Ускорение операций с файлами *185*
Хешированные файлы *185*
Индексированные файлы *188*
Несортированные файлы с плотным индексом *189*
Вторичные индексы *190*
6.4. Внешние деревья поиска 191
Разветвленные Деревья поиска *191*
B-деревья *192*
Поиск записей *193*
Вставка записей *194*
Удаление записей *194*
Время выполнения операций с B-деревом *195*
Сравнение методов *196*

7. ХЕШИРОВАНИЕ. ПРИКЛАДНЫЕ АЛГОРИТМЫ 198

7.1. Теория 198
7.2. Реализация 202

ЛАБОРАТОРНЫЕ РАБОТЫ 205

КОНТРОЛЬНЫЕ РАБОТЫ 206

Контрольная 1 206
Контрольная 2 216

ПРИЛОЖЕНИЯ 217

Таблицы кодировок русского языка 217
Исходные тексты программ 219
Работа с матрицами *219*
Работа с деревьями *220*
Работа со списками *223*
Пример оформления контрольной работы. 226

ВОПРОСЫ К ЭКЗАМЕНУ 230

СПИСОК РЕКОМЕНДУЕМЫХ ИСТОЧНИКОВ **232**

СОДЕРЖАНИЕ **234**