Министерство образования и науки Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Тамбовский государственный технический университет»**

**Ю.Ю. Громов, О.Г. Иванова, Ю.В. Кулаков,
Ю.В. Минин, В.Г. Однолько**

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Допущено Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлениям
220100 «Системный анализ и управление»,
230400 «Информационные системы и технологии»

Тамбов
Издательство ФГБОУ ВПО «ТГТУ»
2012

УДК 004.42(075.8)
ББК з973-018я73
М545

Р е ц е н з е н т ы:

Доктор физико-математических наук, профессор,
заслуженный деятель науки РФ
В.Ф. Крапивин

Доктор физико-математических наук, профессор
Ф.А. Мкртчян

М545 Методы программирования : учебное пособие / Ю.Ю. Громов, О.Г. Иванова, Ю.В. Кулаков, Ю.В. Минин, В.Г. Однолько. – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2012. – 144 с. – 100 экз. – ISBN 978-5-8265-1076-6.

Содержит теоретический материал, упражнения, ответы и список рекомендуемой литературы.

Рекомендовано Учёным советом университета в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальностям 090105 «Комплексное обеспечение информационной безопасности автоматизированных систем», 090303 «Информационная безопасность автоматизированных систем», 230104 «Системы автоматизированного проектирования» и направлениям 010400 «Прикладная математика и информатика», 220100 «Системный анализ и управление», 220400 «Управление в технических системах», 230100 «Информатика и вычислительная техника», 230400 «Информационные системы и технологии», и для студентов средних учебных заведений, обучающихся по специальностям 2203 «Программное обеспечение вычислительной техники и автоматизированных систем», 230115 «Программирование в компьютерных системах».

УДК 004.42(075.8)
ББК з973-018я73

ISBN 978-5-8265-1076-6

ВВЕДЕНИЕ

Дисциплина «Методы программирования» имеет целью обучение студентов принципам построения и анализа алгоритмов, способствует развитию логического мышления, формированию научного мировоззрения и привитию склонности к творчеству. При изучении этого курса используются знания, полученные студентами в процессе изучения дисциплин «Математический анализ», «Теория вероятностей» и «Языки программирования высокого уровня». Знания, умения и практические навыки, полученные в курсе «Методы программирования», используются студентами в профессиональных и научных дисциплинах, а также при выполнении лабораторных, курсовых и дипломных работ.

Задачей дисциплины «Методы программирования» является преподавание основ: структур данных; оценки сложности алгоритмов; алгоритмов сортировки; алгоритмов поиска; алгоритмов на графах; алгоритмов генерации случайных последовательностей и подстановок.

В результате изучения дисциплины студенты *должны иметь представление*: о способах оценки сложности работы алгоритмов; о возможности модификации алгоритмов с учётом конкретных практических задач; *знать*: принципы, лежащие в основе алгоритмов сортировки и поиска информации; принципы хранения и обработки информации в алгоритмах сортировки, поиска и алгоритмах на графах; методы генерации случайных последовательностей и подстановок; *уметь*: сформулировать задачу и использовать для её решения известные методы; применять полученные знания к различным предметным областям; реализовывать алгоритмы на языках программирования высокого уровня, выбирая структуры данных для хранения информации; *иметь навыки*: написания и отладки программ, реализующих алгоритмы сортировки, поиска и алгоритмы на графах; получения эмпирических оценок трудоёмкости алгоритма.

В учебном пособии рассмотрены линейные структуры данных; деревья; понятие сложности алгоритмов; алгоритмы внутренней сортировки: сортировку вставками, обменную сортировку, сортировку подсчётом, сортировку посредством выбора, распределяющую сортировку и сортировку слиянием; алгоритмы внешней сортировки; алгоритмы поиска: последовательный поиск, поиск в упорядоченной таблице, поиск по бинарному дереву, сбалансированные деревья и хеширование. Этот материал освещает первые три темы программы [1] изучения дисциплины «Методы программирования».

При изложении основного материала пособия авторы руководствовались работами [2, 3]. Источники [4 – 6] приведены в качестве вспомогательной литературы.

1. ЛИНЕЙНЫЕ ИНФОРМАЦИОННЫЕ СТРУКТУРЫ

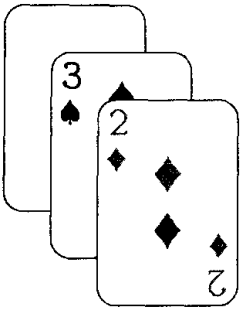
Программы для ЭВМ обычно оперируют с таблицами информации, в которых присутствуют важные структурные отношения между элементами данных. В простейшем случае таблица представляет собой линейный список элементов. В более сложных ситуациях таблица может быть двумерным массивом, имеющим строки и столбцы; *n*-мерным массивом при весьма больших значениях *n*; может иметь структуру дерева, представляющего отношения иерархии или ветвления; наконец, может быть сложной многосвязной структурой, подобной человеческому мозгу. Хорошее понимание структурных отношений позволяет правильно использовать вычислительную машину при решении конкретных задач.

Большую часть материала, который предполагается рассмотреть, будем называть «обработкой списков», поскольку был разработан ряд систем программирования (например, ИПЛ-V, ЛИСП и СЛИП), которые упростили работу с некоторыми общими видами структур, называемых *списками*. Информация в списке представляется множеством записей (объектов), называемых *узлами* (*элементами*) *списка*. Каждый узел списка состоит из одного или нескольких последовательных слов в памяти машины, разделённых на именуемые части, называемые *полями*. Содержимым любого поля в узле могут быть числа, буквы, связи – всё, что только пожелает программист. В качестве примера рассмотрим случай, когда элементы списка представляют игральные карты: мы можем иметь узлы, разбитые на четыре поля – *TAG* (признак), *SUIT* (масть), *RANK* (ранг) и *NEXT* (следующий):

TAG	*SUIT*	*RANK*	*NEXT*

Пусть: *TAG* = 1 означает, что карта повёрнута лицевой стороной вниз, *TAG* = 0 – лицевой стороной вверх; *SUIT* = 1, 2, 3 и 4 – соответственно для трефовых, бубновых, червовых и пиковых карт; *RANK* = 1, 2, …, 13 – для туза, двойки, …, короля; *NEXT* – связь с картой в стопке ниже данной. *Адрес* узла является адресом первого слова в узле. Адрес узла называют *указателем* на этот узел. Тогда некоторая колода из трёх карт может выглядеть так:

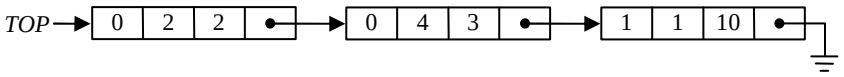
Фактические карты *Машинное представление*



100:	1	1	10	Λ
386:	0	4	3	100
242:	0	2	2	386

В машинном представлении указаны адреса памяти 100, 386 и 242, но на их месте могли быть любые другие числа, поскольку каждая карта ссылается на следующую карту в колоде. Специальный указатель Λ (ламбда) в узле 100 обозначает *пустую связь*, т.е. связь с узлом, которого не существует (десятка треф – нижняя карта в стопке).

Введение связей с другими элементами данных является чрезвычайно важной идеей в программировании – это ключ к представлению сложных информационных структур. В машинном представлении списка связи между узлами удобно изображать стрелками. Так, для нашего примера список приобретает следующий вид:



Фактические адреса 242, 386 и 100 (значение которых не суть важно) отсутствуют в данном представлении, а пустая связь изображена как «заземление» для электрических схем. Символ *TOP* представляет *переменную связи* или, как часто говорят, указатель, т.е. переменную, значением которой является адрес памяти. Для осуществления ссылок на поля узлов будем использовать имя поля, за которым в скобках следует связь с желаемым узлом:

$$TAG\,(TOP) = 0; \quad SUIT\,(TOP) = 2; \quad RANK\,(100) = 10;$$
$$RANK\,(NEXT\,(TOP)) = 3.$$

В качестве примера рассмотрим простой алгоритм, который помещает новую карту в колоду сверху, предполагая, что *NEWCARD* – переменная связи и её значением является связь с новой картой:

*A*1. Установить *NEXT* (*NEWCARD*) ← *TOP*. (Устанавливается соответствующая связь в узле новой карты.)
*A*2. Установить *TOP* ← *NEWCARD*. (Тем самым обеспечивается, что *TOP* по-прежнему указывает на верхнюю карту в колоде.)
*A*3. Установить *TAG* (*TOP*) ← 0. (Отмечается, что карта повёрнута лицевой стороной вверх.) ■

Другим примером является алгоритм, подсчитывающий количество карт в колоде в данный момент:

*B*1. Установить $N \leftarrow 0$, $X \leftarrow TOP$. (*N* – целая переменная, *X* – переменная связи.)
*B*2. Если $X = \Lambda$, то остановиться. (*N* равно числу карт в колоде.)
*B*3. Установить $N \leftarrow N + 1$, $X \leftarrow NEXT(X)$ и вернуться к шагу *B*2. ■

Линейный список – это множество, состоящее из $n \geq 0$ узлов $x[1], x[2], \ldots, x[n]$, структурные свойства которого ограничиваются только линейным относительным положением узлов: $x[1]$ – первый узел; при $1 < k < n$ узлу $x[k]$ предшествует узел $x[k-1]$, а следует за ним узел $x[k+1]$; $x[n]$ – последний узел.

Возможные операции с линейными списками:

1) получение доступа к *k*-му узлу списка, чтобы проанализировать и/или изменить содержимое его полей;
2) включение нового узла в список непосредственно перед *k*-м узлом;
3) исключение *k*-го узла списка;
4) объединение двух (или более) списков в один;
5) разбиение списка на два (или более) списков;
6) копирование списка;
7) определение количества узлов в списке;
8) сортировка узлов списка по некоторым полям;
9) поиск в списке узла с заданным значением в некотором поле.

Случаи при $k = 1$ и $k = n$ в операциях 1), 2), 3) особые, поскольку в линейном списке доступ к первому и последнему узлу получить проще.

Очень часто на практике встречаются линейные списки, в которых включение, исключение и доступ к значениям производятся в первом или последнем узлах. Среди таких списков различают:

- *стек* – список, в котором все включения, все исключения и всякий доступ выполняются в одном конце этого списка;
- *очередь* – список, в котором все включения производятся на одном конце этого списка, а все исключения и всякий доступ – на другом его конце;
- *дек* – список, в котором включения, исключения и всякий доступ делаются на обоих концах этого списка.

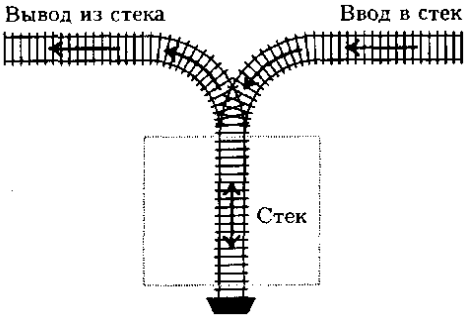


Рис. 1. Стек, представленный в виде железнодорожного разъезда

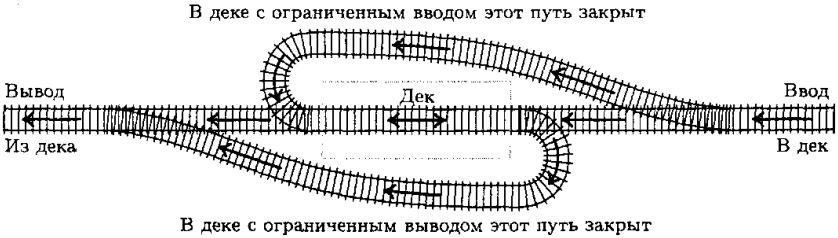


Рис. 2. Дек, представленный в виде железнодорожного разъезда

Среди деков имеются так называемые *деки с ограниченным выходом* и *деки с ограниченным входом*, в которых соответственно исключение и включение допускаются только на одном конце списка.

При описании алгоритмов, использующих такие списки, принята специальная терминология. Так, мы помещаем элемент на *верх* стека или снимаем верхний элемент (рис. 3, *а*). *Внизу* стека находится наименее доступный элемент, который не удаляется до тех пор, пока не будут исключены все другие элементы. Применительно к очередям мы говорим о *начале* и *конце* очереди: объекты встают в конец очереди и удаляются тогда, когда достигают её начала (рис. 3, *б*). Говоря о деках, мы указываем *левый* и *правый* концы (рис. 3, *в*).

Приведём несколько дополнительных способов описания операций, выполняемых над стеками и очередями. Будем писать $A \Leftarrow x$, указывая (если A – стек), что значение x помещается на верх стека или (если A – очередь) что x включается в конец очереди. Подобным же образом запись $x \Leftarrow A$ будет обозначать, что переменная x принимает значение верхнего элемента стека A или начального элемента очереди A, и это значение исключается из A.

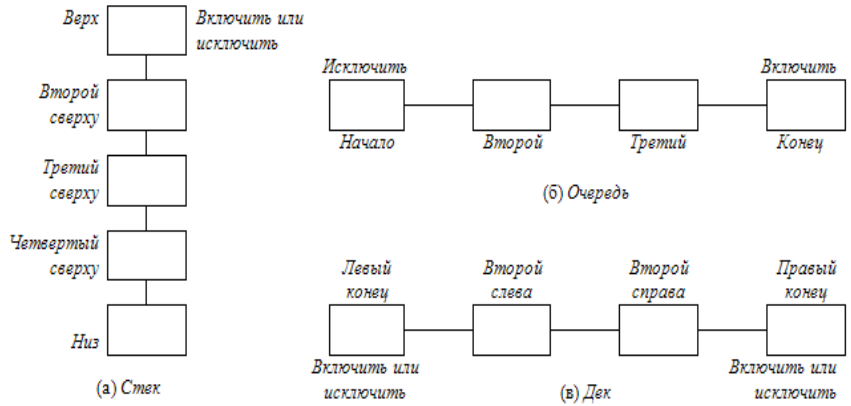


Рис. 3. Три важных класса линейных списков

При непустом стеке A его верхний элемент можно обозначить через top (A).

Упражнения

1. Дек с ограниченным входом является линейным списком, в котором элементы могут включаться только с одного конца, а исключаться с любого конца. Очевидно, что дек с ограниченным входом может работать либо как стек, либо как очередь, если всегда будем удалять элементы с одного из двух концов.

Может ли дек с ограниченным выходом работать либо как стек, либо как очередь?

2. Представьте себе, что четыре железнодорожных вагона находятся на входном пути железнодорожного разъезда, иллюстрирующего стек, и пронумерованы соответственно 1, 2, 3 и 4 (1234). Предположим, что мы выполняем некоторую последовательность операций, которые согласуются с направлением стрелок на рисунке при условии, что вагоны не могут «перепрыгивать» друг через друга. Отправьте: (a) вагон 1 в стек; (b) вагон 2 в стек; (c) вагон 2 на выход; (d) вагон 3 в стек; (e) вагон 4 в стек; (f) вагон 4 на выход; (g) вагон 3 на выход; (h) вагон 1 на выход.

В результате этих операций первоначальный порядок вагонов 1234 изменился на 2431.

Заметим, что (a) – (h) можно описать короче и выразительнее, используя код SSXSSXXX, где S обозначает фразу «отправить вагон из входа в стек», а X − фразу «отправить вагон из стека на выход».

Пусть имеется шесть железнодорожных вагонов, пронумерованных 1, 2, 3, 4, 5 и 6. Установить, можно ли их переставить в порядке:

а) 325641;

б) 154623.

Если такие перестановки возможны, то покажите, как это сделать.

3. Некоторые последовательности из S и X, описывают бессмысленные операции, так как может не оказаться вагонов на указанных путях. Невозможно, например, реализовать последовательность SXXSSXXS.

Будем называть последовательность из S и X, представляющую перестановку из n элементов, *допустимой*, если в ней содержится n символов S и n символов X, и если она задаёт операции, которые можно выполнить.

Сформулируйте правило, которое позволит различать допустимые и недопустимые последовательности.

Покажите, что никакие две различные допустимые последовательности не приводят к одинаковой выходной перестановке.

4. Простая формула для a_n – количества перестановок из n элементов, которые можно получить, используя стек так, как в упр. 2, имеет вид: $a_n = \binom{2n}{n} - \binom{2n}{n-1}$, где $\binom{2n}{n}$ – общее число последовательностей, в которые S и X входят одинаковое число n раз; $\binom{2n}{n-1}$ – число недопустимых последовательностей, в которые S и X входят одинаковое число n раз.

Вычислите значение a_n и определите всевозможные, недопустимые и допустимые последовательности из S и X, а также перестановки, которые будут получены в результате реализации допустимых последовательностей для:

а) $n = 2$;

б) $n = 3$.

5. С использованием стека можно получить перестановку $p_1 p_2 \ldots p_n$ из 1, 2, …, n тогда и только тогда, когда нет таких индексов i, j и k, что $i < j < k$ и $p_j < p_k < p_i$. Применив это условие, поясните, почему из 123456 перестановку 154623 получить нельзя, а перестановку 325641 – можно.

6. Рассмотрите задачу, поставленную в упр. 2, заменив стек на очередь. Какие перестановки из 1, 2, …, n можно получить, используя очередь?

7. Рассмотрите задачу из упр. 2, заменив стек на дек. Найдите перестановки чисел 1, 2, 3 и 4, которые:

а) можно получить в случае дека с ограниченным входом, но нельзя получить для дека с ограниченным выходом;

б) можно получить в случае дека с ограниченным выходом, но нельзя получить для дека с ограниченным входом;

в) нельзя получить ни для дека с ограниченным входом, ни для дека с ограниченным выходом.

8. Существуют ли какие-либо перестановки чисел 1, 2, …, n, которые нельзя получить с помощью дека, не имеющего ни ограниченного входа, ни ограниченного выхода?

2. ПОСЛЕДОВАТЕЛЬНОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ ПРИ ХРАНЕНИИ ЛИНЕЙНЫХ СПИСКОВ

Простейший и наиболее естественный способ хранения линейного списка в памяти машины сводится к размещению элементов списка в последовательных ячейках памяти, один узел за другим. В этом случае

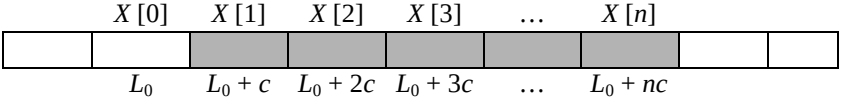
$$LOC\,(X\,[\,j+1\,]) = LOC\,(X\,[\,j\,]) + c,$$

где $LOC\,(X\,[\,j+1\,])$, $LOC\,(X\,[\,j\,])$ – адреса узлов $X\,[\,j+1\,]$ и $X\,[\,j\,]$ соответственно; c – количество слов в одном узле.

В общем случае адрес узла $X\,[\,j\,]$ определяется выражением

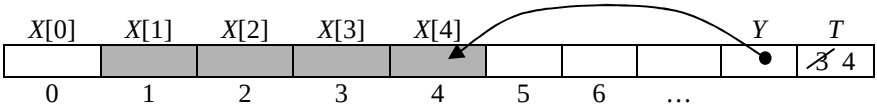
$$LOC\,(X\,[\,j\,]) = L_0 + cj, \qquad (1)$$

где L_0 – константа, называемая *базовым адресом* и являющаяся адресом гипотетического узла $X\,[0]$.



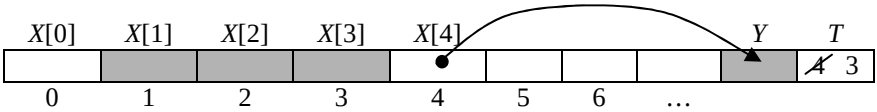
Последовательное распределение очень удобно при работе со стеком. Для этого достаточно иметь *указатель стека* T. Когда стек пуст, $T = 0$. Для того чтобы поместить новый элемент в стек, необходимо (в предположении, что $c = 1$):

$$T \leftarrow T + 1; \quad X\,[T] \leftarrow Y. \qquad (2)$$



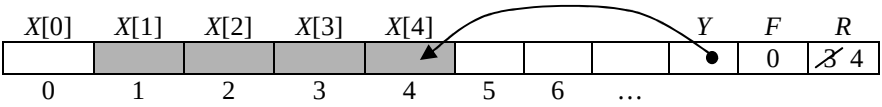
Чтобы переменной Y дать значение верхнего узла непустого стека и исключить этот узел из списка, требуется

$$Y \leftarrow X\,[T]\,;\; T \leftarrow T - 1. \qquad (3)$$



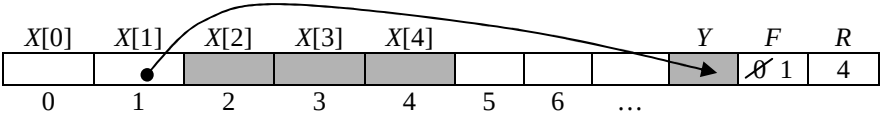
Представление очереди или, более того, дека общего вида требует некоторых ухищрений. Выберем две переменные: указатель F для начала очереди и указатель R для конца очереди. Тогда включение элемента в конец очереди осуществляется следующим образом:

$$R \leftarrow R + 1;\quad X\,[R] \leftarrow Y, \qquad (4)$$



а исключение начального узла (F указывает на место, непосредственно перед началом очереди) так:

$$F \leftarrow F + 1;\quad Y \leftarrow X\,[F\,];\quad \text{если } F = R, \text{ то } F \leftarrow R \leftarrow 0. \qquad (5)$$



Заметим, что в такой ситуации (когда в очереди есть по крайней мере один узел) R и F растут, причём $R > F$. В результате последовательно занимаются ячейки X [1], X [2], … до бесконечности, что свидетельствует о чрезмерно расточительном использовании памяти. Следовательно, простой метод (4), (5) должен использоваться в такой ситуации, когда известно, что указатель F довольно регулярно догоняет указатель R (очередь опустошается).

Обойти проблему выхода очереди за пределы памяти можно зафиксировав M узлов $X[1], \ldots, X[M]$ и неявно «замкнув» их в кольцо так, что за $X[M]$ следует $X[1]$. Тогда процессы (4), (5) преобразуются к виду:

$$\text{если } R = M, \text{ то } R \leftarrow 1; \quad \text{иначе } R \leftarrow R + 1; \quad X[R] \leftarrow Y. \tag{6}$$

$$\text{если } F = M, \text{ то } F \leftarrow 1; \quad \text{иначе } F \leftarrow F + 1; \quad Y \leftarrow X[F]. \tag{7}$$

Очевидно, что в очереди при такой ситуации не может быть более M узлов.

Представим действия с включением и исключением узлов для стека (2), (3) и очереди (6), (7) с учётом того, что в результате включения узла может возникнуть выход за отведённое место в памяти (ПЕРЕПОЛНЕНИЕ) или при попытке исключения отсутствующего узла в списке (НЕХВАТКА):

X $\Leftarrow$ Y (включить в стек);
$$\begin{cases} T \leftarrow T + 1, & \text{если } T > M, \text{ то ПЕРЕПОЛНЕНИЕ;} \\ X[T] \leftarrow Y. \end{cases}$$

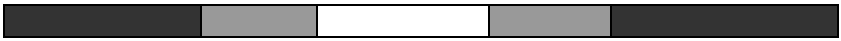
Y $\Leftarrow$ X (исключить из стека);
$$\begin{cases} \text{если } T = 0, \text{ то НЕХВАТКА;} \\ Y \leftarrow X[T], \quad T \leftarrow T - 1. \end{cases}$$

X $\Leftarrow$ Y (включить в очередь);
$$\begin{cases} \text{если } R = M, \text{ то } R \leftarrow 1, \text{ иначе } R \leftarrow R + 1; \\ \text{если } R = F, \text{ то ПЕРЕПОЛНЕНИЕ;} \\ X[R] \leftarrow Y. \end{cases}$$

X $\Leftarrow$ Y (исключить из очереди);
$$\begin{cases} \text{если } R = F, \text{ то НЕХВАТКА;} \\ \text{если } F = M, \text{ то } F \leftarrow 1, \text{ иначе } F \leftarrow F + 1; \\ Y \leftarrow X[F]. \end{cases}$$

Заметим, что ПЕРЕПОЛНЕНИЕ является критической ошибкой.

Рассмотрим случаи, когда программа работает не с единственным списком. Если имеется всего два стека переменного размера, то они могут хорошо сосуществовать вместе, когда будут расти навстречу друг другу:



Начало памяти *Низ* *Верх* *Верх* *Низ* *Конец памяти*

Ситуация ПЕРЕПОЛНЕНИЕ не возникнет до тех пор, пока суммарный объём обоих списков не исчерпает всё свободное пространство в памяти. Такое распределение памяти используется очень часто.

Можно легко убедиться в том, что не существует способа, который позволяет хранить три и более стека переменного размера так, чтобы:

1) ПЕРЕПОЛНЕНИЕ возникало лишь в случае, когда суммарный размер всех списков превысит отведённую для них область памяти;

2) «нижний» элемент каждого списка имел фиксированный адрес.

Если удовлетворять первому условию, то придётся нарушить второе. Это означает, что базовый адрес L_0 в формуле (1) не является более константой и никакой указатель не может быть абсолютным адресом. Кроме того, все ссылки должны быть относительны L_0.

Предположим, что имеется n стеков, $BASE\ [i]$ – базовый адрес, а $TOP[i]$ – указатель i-го стека. Тогда алгоритмы включения и исключения узлов становятся такими:

(Включение) $TOP\ [i] \leftarrow TOP\ [i] + 1$; если $TOP\ [i] > BASE\ [i + 1]$, то ПЕРЕПОЛНЕНИЕ;
в противном случае
$CONTENTS\ (TOP\ [i]) \leftarrow Y$. (8)

(Исключение) если $TOP\ [i] = BASE\ [i]$, то НЕХВАТКА;
в противном случае
$Y \leftarrow CONTENTS\ (TOP\ [i])$, $TOP\ [i] \leftarrow TOP\ [i] - 1$.

В данной ситуации критичность ПЕРЕПОЛНЕНИЯ можно устранить «переупаковав память». Рассмотрим простейший метод переупаковки.

Пусть n стеков располагаются в общей области памяти, состоящей из ячеек с адресами L такими, что $L_0 < L \leq L_\infty$, где L_0, L_∞ – границы области памяти, предоставленной для использования. Можно считать, что вначале все стеки пусты ($BASE\ [i] = TOP\ [i] = L_0$ для всех i). Для правильного выполнения операции включения (8) при $i = n$ базовый адрес $BASE\ [n + 1]$ гипотетического $(n + 1)$-го стека следует считать равным L_∞. Теперь ПЕРЕПОЛНЕНИЕ будет возникать всякий раз, когда в некоторый стек, за исключением n-го, потребуется записать элементов больше, чем когда-либо прежде.

При переполнении i-го стека реализуется одна из трёх возможностей:

1) Определяется наименьшее k (если такое значение существует), для которого $i < k \le n$ и $TOP\,[k] < BASE\,[k+1]$ (k-й список заканчивается раньше, чем начинается $(k+1)$-й). Затем осуществляется сдвиг на одну позицию *вверх*:

$$CONTENTS\,(L+1) \leftarrow CONTENTS\,(L),$$
$$\text{для } L = TOP\,[k], TOP\,[k]-1, \ldots, BASE\,[i+1]+1.$$

И выполняются операторы $BASE\,[j] \leftarrow BASE\,[j] + 1$, $TOP\,[j] \leftarrow TOP\,[j] + 1$ для $j = i+1, i+2, \ldots, k$.

2) Нельзя найти значение k, удовлетворяющее условию 1), но имеется наибольшее k, для которого $1 \le k < i$ и $TOP\,[k] < BASE\,[k+1]$. Теперь осуществляется сдвиг на одну позицию *вниз*:

$$CONTENTS\,(L-1) \leftarrow CONTENTS\,(L),$$
$$\text{для } L = BASE\,[k+1]+1, BASE\,[k+1]+2, \ldots, TOP\,[i].$$

И выполняется $BASE\,[j] \leftarrow BASE\,[j] - 1$, $TOP\,[j] \leftarrow TOP\,[j] - 1$ для $j = k+1, k+2, \ldots, i$.

3) Для всех $k \ne i$ имеет место $TOP\,[k] = BASE\,[k+1]$. Тогда невозможно найти место для нового элемента i-го стека и работу со стеками следует прекратить.

Понятно, что многих первых переполнений стеков можно избежать, если разумно выбрать начальные условия, а не отводить с самого начала всю память под n-й стек. Например, если ожидается, что стеки будут одинакового размера, то можно начать с равномерного распределения памяти:

$$BASE\,[j] = TOP\,[j] = c\left\lfloor \left(\frac{j-1}{n}\right)(L_\infty - L_0)\right\rfloor + L_0.$$

Однако каким бы хорошим ни было начальное распределение, оно позволяет сэкономить лишь фиксированное количество переполнений на ранней стадии работы программы.

Описанный простейший метод можно улучшить, если при каждой переупаковке памяти готовить место для более чем одного нового элемента. При этом выигрыш во времени достигается за счёт того, что сдвиг таблицы на несколько позиций сразу выполняется быстрее, чем несколько сдвигов на одну позицию.

Предлагается при возникшей ситуации переполнения производить полную переупаковку памяти, основываясь на изменении размера каждого стека с момента последней переупаковки. Представленный ниже алгоритм использует дополнительный массив с именем *OLDTOP*, в котором хранятся значения, бывшие в массиве *TOP* непосредственно после пре-

дыдущего распределения памяти. Первоначально таблицы заполняются, как и прежде, причём $OLDTOP\,[i] = TOP\,[i]$, $1 \le i \le n$.

Алгоритм *G*. (*Перераспределение последовательных таблиц.*)

Предполагается, что переполнение случилось в стеке i, в соответствии с операцией включения (8). После выполнения алгоритма окажется, что память либо израсходована вся, либо будет перераспределена так, что можно выполнить $NODE\,(TOP\,[i]) \leftarrow Y$. Заметим, что $TOP\,[i]$ был увеличен ещё в (8), прежде чем начал работу этот алгоритм.

***G*1.** [Начальная установка.] $SUM \leftarrow L_\infty - L_0$, $INC \leftarrow 0$.

***G*2.** [Сбор статистики.] Для $j = 1, 2, \ldots, n$ выполнить $SUM \leftarrow SUM - (TOP\,[j] - BASE\,[j])$. Если $TOP\,[j] > OLDTOP\,[j]$, то $D\,[j] \leftarrow TOP\,[j] - OLDTOP\,[j]$ и $INC \leftarrow INC + D\,[j]$; в противном случае $D\,[j] \leftarrow 0$. (В результате значением SUM будет суммарное количество свободного пространства в памяти, а значением INC – суммарный рост размеров стеков с момента последнего распределения.)

***G*3.** [Память полна?] Если $SUM < 0$, то дальше работать нельзя.

***G*4.** [Вычисление коэффициентов распределения.] $\alpha \leftarrow 0.1 \times SUM / n$, $\beta \leftarrow 0.9 \times SUM / INC$. (Здесь α и β не целые числа, а дроби и вычислять их следует с достаточной точностью. На следующем шаге приблизительно 10% свободного пространства будут поделены поровну между всеми стеками, а остальные 90% свободной памяти – пропорционально росту размера стеков с момента предыдущего распределения.)

***G*5.** [Вычисление новых базовых адресов.] $NEWBASE\,[1] \leftarrow BASE[1]$, $\sigma \leftarrow 0$. Для $j = 2, 3, \ldots, n$ $\tau \leftarrow \sigma + \alpha + D\,[j-1]\,\beta$, $NEWBASE\,[j] \leftarrow NEWBASE\,[j-1] + TOP\,[j-1] - BASE\,[j-1] + \lfloor\tau\rfloor - \lfloor\sigma\rfloor$ и $\sigma \leftarrow \tau$.

***G*6.** [Переупаковка.] $TOP\,[i] \leftarrow TOP\,[i] - 1$. (Этим устанавливается фактический размер i-го стека для того, чтобы не делалось попыток перемещать информацию, выходящую за его границу.) Выполнить алгоритм R, приведённый ниже, и $TOP\,[i] \leftarrow TOP\,[i] + 1$. (Восстановить значение $TOP[i]$.) Затем $OLDTOP\,[j] \leftarrow TOP\,[j]$ для $1 \le j \le n$. ■

Возможно, самой интересной частью всего этого алгоритма является общий процесс переупаковки, который сейчас опишем. Переупаковка оказывается нетривиальной, поскольку одни части памяти должны сдвигаться вверх, а другие вниз. Заметим, что при этих перемещениях важно не затереть какую-либо полезную информацию.

Алгоритм *R*. (*Перемещение последовательных таблиц.*) Для $j = 1, 2, \ldots, n$ информация, специфицированная с помощью $BASE\,[j]$ и $TOP\,[j]$, перемещается на новые места, заданные с помощью $NEWBASE\,[j]$, а значения $BASE\,[j]$ и $TOP\,[j]$ соответствующим образом корректируются.

***R*1.** [Начальная установка.] $j \leftarrow 1$. (Заметим, что никогда не возникает необходимости перемещать стек 1. Поэтому ради эффективности программист должен поставить наибольший стек первым, если он знает его.)

***R*2.** [Поиск начала сдвига.] (Все стеки от 1 до j перемещены нужным образом.) $j \leftarrow j + 1$. Если $NEWBASE[j] < BASE[j]$, то перейти к шагу *R*3; иначе если $NEWBASE[j] > BASE[j]$, то перейти к шагу *R*4; иначе если $j > n$, то окончить алгоритм; иначе выполнить шаг *R*2 сначала.

***R*3.** [Сдвиг вниз.] $\delta \leftarrow BASE[j] - NEWBASE[j]$. Для $L = BASE[j] + 1, BASE[j] + 2, \ldots, TOP[j]$ $CONTENTS(L - \delta) \leftarrow CONTENTS(L)$. (Заметим, что $BASE[j]$ может оказаться равным $TOP[j]$, и в этом случае не требуется никаких действий.) $BASE[j] \leftarrow NEWBASE[j]$, $TOP[j] \leftarrow TOP[j] - \delta$. Перейти к *R*2.

***R*4.** [Поиск верхней границы сдвига.] Найти наименьшее $k \geq j$, для которого $NEWBASE[k + 1] \leq BASE[k + 1]$. (Заметим, что $NEWBASE[n + 1]$ должен быть равен $BASE[n + 1]$ и поэтому такое значение k всегда существует.) Затем выполнить шаг *R*5 для $T = k, k - 1, \ldots, j$; $j \leftarrow k$ и перейти к *R*2.

***R*5.** [Сдвиг вверх.] $\delta \leftarrow NEWBASE[T] - BASE[T]$. Выполнить $CONTENTS(L + \delta) \leftarrow CONTENTS(L)$ для $L = TOP[T], TOP[T] - 1, \ldots, BASE[T] + 1$. (Заметим, что, как и на шаге *R*3, может не потребоваться никаких действий.) $BASE[T] \leftarrow NEWBASE[T]$, $TOP[T] \leftarrow TOP[T] + \delta$. ■

Описанные алгоритмы можно адаптировать и для других относительно адресуемых таблиц, в которых текущая информация располагается между указателями $BASE[j]$ и $TOP[j]$.

При занятой памяти только на половину алгоритм *G* работает хорошо и по крайней мере даёт правильные результаты при почти заполненной памяти. В последнем случае алгоритм *R*, осуществляющий перемещение последовательных таблиц, работает довольно долго. При большом количестве последовательных таблиц переменного размера не следует думать, что удастся использовать пространство памяти на 100% и все-таки избежать переполнения. Для исключения бессмысленных потерь времени можно остановить алгоритм *G* на шаге *G*3, если значение переменной *SUM* станет меньше величины $S_{\min}$, которую задаёт программист, избегая тем самым излишне трудоёмкие переупаковки.

Упражнения

1. Пусть состояние памяти по окончании последней переупаковки было следующим:

1	1					3	3	3		4	4								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Заметим, что число 1 в некоторой ячейке памяти обозначает элемент первого стека, число 2 − элемент второго стека и т.д. Следовательно, базовые адреса стеков имеют значения: $BASE[1] = -1$, $BASE[2] = 2$,

BASE [3] = 5, *BASE* [4] = 9 и адреса верхних элементов стеков − значения: *TOP* [1] = 1, *TOP* [2] = 2, *TOP* [3] = 8, *TOP* [4] =11.

Предположим, что требуется включить новый элемент во второй стек, затем новый элемент в третий стек и ещё раз включить новый элемент в третий стек (обозначим эту последовательность в виде *I*2, *I*3, *I*3). Тогда первые две из этих трёх операций пройдут успешно, а последняя − нет, поскольку возникнет ситуация ПЕРЕПОЛНЕНИЯ третьего стека:

1	1		2			3	3	3	3	4	4								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

при значениях базовых адресов стеков: *BASE* [1] = –1, *BASE* [2] = 2, *BASE* [3] = 5, *BASE* [4] = 9 и адресов верхних элементов: *TOP* [1] = 1, *TOP* [2] = 3, *TOP* [3] = 10, *TOP* [4] =11.

Для исключения этой критической ситуации начинает свою работу алгоритм переупаковки памяти ***G***.

Требуется определить состояние памяти и значения указателей *BASE* [*j*] и *TOP* [*j*] для *j* = 1, 2, 3 и 4 по окончании переупаковки.

3. СВЯЗАННОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ ПРИ ХРАНЕНИИ ЛИНЕЙНЫХ СПИСКОВ

Вместо хранения списка в последовательных ячейках памяти, можно использовать более гибкую схему, в которой каждый узел содержит связь со следующим узлом списка. Ниже представлены схемы последовательного и связанного распределения памяти при хранении линейного списка, состоящего из пяти элементов.

Последовательное распределение

Адрес	Содержимое
$L_0 + c$:	Элемент 1
$L_0 + 2c$:	Элемент 2
$L_0 + 3c$:	Элемент 3
$L_0 + 4c$:	Элемент 4
$L_0 + 5c$:	Элемент 5

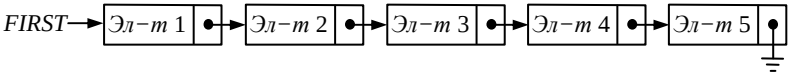
Связанное распределение

Адрес	Содержимое	
A:	Элемент 1	*B*
B:	Элемент 2	*C*
C:	Элемент 3	*D*
D:	Элемент 4	*E*
E:	Элемент 5	Λ

Здесь *A*, *B*, *C*, *D* и *E* – адреса произвольных ячеек в памяти, Λ – пустая связь.

В случае связанного распределения памяти в программе имеется переменная связи или константа, имеющая значение *A*. Отправляясь от узла по адресу *A*, можно найти все другие элементы списка.

Связи часто изображаются просто стрелками:



(9)

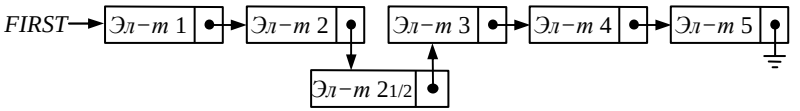
Здесь *FIRST* – указатель на первый узел в списке.

Сопоставим последовательное и связанное распределение:

1. Связанное распределение требует дополнительного пространства в памяти для хранения связей. Однако при использовании связанной памяти часто возникает неявный выигрыш в памяти за счёт совмещения общих частей таблиц. Кроме того, связанное распределение будет более эффективным при плотно загруженной памяти.

2. Легко исключить элемент, находящийся внутри связанного списка. Для этого необходимо только изменить связь в предшествующем элементе. При последовательном же распределении такое исключение обычно требует перемещения значительной части списка вверх на другие места памяти.

3. При связанном распределении легко включить элемент в список. Например, для включения элемента 21/2 в (9) необходимо изменить лишь две связи:



(10)

Такая операция заняла бы значительное время при работе с длинной последовательной таблицей.

4. При связанном распределении памяти значительно медленнее, чем при последовательном, выполняются обращения к произвольным частям списка. Однако в большинстве приложений продвижение по списку осуществляется последовательно, а не произвольным образом. Кроме того, можно создать дополнительные переменные связи, указывающие на соответствующие места в списке.

5. При использовании схемы со связями упрощается задача объединения двух списков или разбиения списка на части.

6. Схема со связями годится для структур более сложных, чем линейные списки.

Рассмотрим организацию включения и исключения узлов для связанного списка.

Пусть каждый узел списка имеет два поля:

INFO	*LINK*	(11)

где *INFO* – полезная информация, *LINK* – связь со следующим узлом.

Использование связанного распределения предполагает поиск пустого пространства для нового узла. Для этого существует специальный список, назовём его *AVAIL*, который содержит все узлы, которые в данный момент не используются, и ничем не отличается от любых других списков. Переменная связи *AVAIL* ссылается на верхний элемент этого списка.

Для выделения узла из списка *AVAIL* с целью его использования необходимо поступить следующим образом:

$$X \leftarrow AVAIL, \quad AVAIL \leftarrow LINK(AVAIL). \tag{12}$$

В результате из стека *AVAIL* исключается верхний узел и переменная связи *X* становится указателем на только что выделенный узел. Операцию (12) для краткости будем обозначать $X \Leftarrow AVAIL$.

Если узел по адресу *X* далее не нужен, то его можно вернуть в список свободного пространства:

$$LINK(X) \leftarrow AVAIL, \quad AVAIL \leftarrow X. \tag{13}$$

Эту операцию будем обозначать $AVAIL \Leftarrow X$.

Для построения списка *AVAIL* в начале работы необходимо (*A*) связать вместе все узлы, которые будут использоваться, (*B*) занести в *AVAIL* адрес первого из этих узлов и (*C*) сделать связь последнего узла пустой. Множество всех узлов, которые могут быть использованы, называют *пулом памяти*.

С учётом проверки на переполнение операцию $X \Leftarrow AVAIL$ следует выполнять так:

Если $AVAIL = \Lambda$, то ПЕРЕПОЛНЕНИЕ;
в противном случае $X \leftarrow AVAIL, \quad AVAIL \leftarrow LINK(AVAIL)$. (14)

Возникшее ПЕРЕПОЛНЕНИЕ означает, что необходимо прекратить выполнение программы.

Рассмотрим несколько наиболее распространённых операций со связанными списками, если работа ведётся со стеками и очередями.

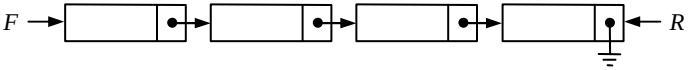
Пусть имеется стек, на верхний элемент которого указывает переменная *T*. Для включения информации из *Y* в стек используется вспомогательный указатель *P*:

$$P \Leftarrow AVAIL, \quad INFO(P) \leftarrow Y, \quad LINK(P) \leftarrow T, \quad T \leftarrow P. \tag{15}$$

Для считывания в *Y* информации из стека следует:

Если $T = \Lambda$, то НЕХВАТКА;
иначе $P \leftarrow T, \ T \leftarrow LINK(P), \quad Y \leftarrow INFO(P), \quad AVAIL \Leftarrow P$. (16)

Связанное распределение особенно удобно в применении к очередям. При этом связи должны быть направлены от начального узла к последнему:



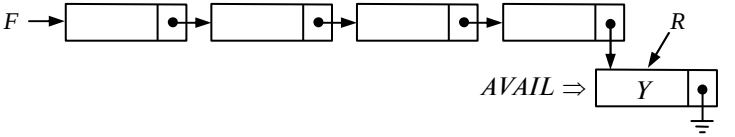
(17)

Здесь F, R – указатели на начало и конец очереди соответственно.

Включение информации из Y в конец очереди выполняется следующим образом:

$$P \Leftarrow AVAIL,\quad INFO(P) \leftarrow Y,\quad LINK(P) \leftarrow \Lambda.$$
$$\text{Если } F = \Lambda, \text{ то } F \leftarrow P, \text{ иначе } LINK(R) \leftarrow P.\quad R \leftarrow P. \tag{18}$$

Если (17) – ситуация перед включением, то после включения в конец очереди схема будет иметь следующий вид:

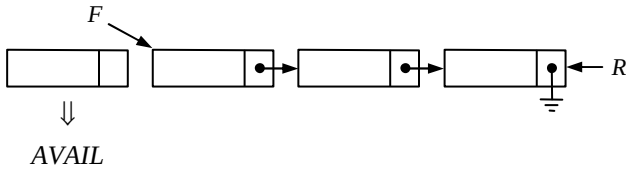


(19)

Исключение информации из начала очереди выполняется следующим образом:

$$\text{Если } F = \Lambda, \text{ то НЕХВАТКА, иначе } P \leftarrow F,\ F \leftarrow LINK(P),$$
$$Y \leftarrow INFO(P),\quad AVAIL \Leftarrow P. \tag{20}$$

Если схема (17) иллюстрирует ситуацию перед исключением, то в результате этой операции она преобразуется в следующую схему:



(21)

Перейдём далее к изучению практического примера *топологической сортировки*, которая оказывается полезной всякий раз, когда мы сталкиваемся с упорядочением.

Пусть имеется следующее частично упорядоченное множество, заданное графом:

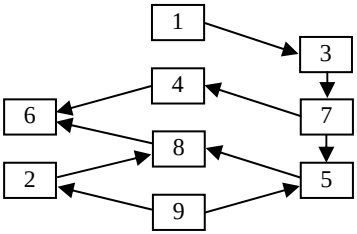


Рис. 4. Исходное частично упорядоченное множество

Здесь, например, работа 1 должна быть выполнена раньше работы 3, работа 3 – раньше работы 7, работа 7 – раньше работ 4, 5 и т.д. (рис. 4).

Требуется расположить элементы множества в линейную последовательность так, чтобы все дуги были ориентированы слева направо (рис. 5):

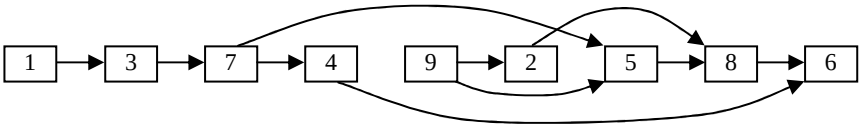


Рис. 5. Частично упорядоченное множество после топологической сортировки

Существует простой способ топологической сортировки вручную. Вначале берётся элемент, которому не предшествует никакой другой элемент. Он помещается первым в выходную последовательность и удаляется из исходного частично упорядоченного множества вместе с инцидентными ему дугами. Затем в графе выбирается ещё один элемент без предшественников, помещается вторым в выходную последовательность и исключается из него. Описанный процесс продолжается до тех пор, пока исходный граф не сократится до пустого графа. Заметим, что линейное размещение всех элементов невозможно, если в графе имеется хотя бы один контур.

Рассмотрим далее реализацию топологической сортировки на машине. При этом будем полагать, что сортируемые элементы пронумерованы числами от 1 до n в любом порядке. Информация вводится парами чисел вида (j, k) или $j < k$. Это означает, что j-й элемент множества предшествует k-му элементу.

Входной информацией применительно к нашему примеру могут быть такие пары:

$$\begin{gathered} 9 < 2,\ \ 3 < 7,\ \ 7 < 5,\ \ 5 < 8, \\ 8 < 6,\ \ 4 < 6,\ \ 1 < 3, \\ 7 < 4,\ \ 9 < 5,\ \ 2 < 8. \end{gathered} \qquad (22)$$

В алгоритме будет использована последовательная таблица $X[1]$, $X[2]$, …, $X[n]$, любой k-й узел которой имеет формат:

COUNT[*k*]	*TOP*[*k*]

где *COUNT*[*k*] – *количество непосредственных предшественников* объекта k (количество встретившихся среди входной информации пар $j < k$), *TOP*[*k*] – связь с началом *списка непосредственных преемников* объекта k. Список непосредственных преемников содержит элементы формата:

SUC	*NEXT*

где *SUC* – непосредственный преемник объекта k, *NEXT* – связь со следующим элементом этого списка.

Для входной информации (22) будем иметь следующую модель памяти:

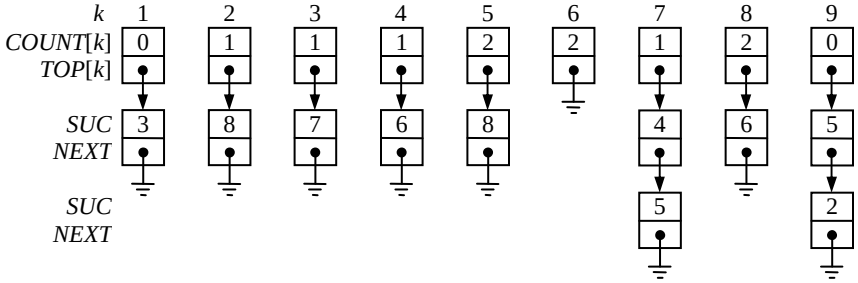


Рис. 6. Машинное представление для отношений (22)

Алгоритм будет заключаться в выводе узлов, у которых поле *COUNT* равно нулю, и последующем уменьшении на один поля *COUNT* у всех преемников этих узлов. Для избежания поиска узлов с нулевым значением поля *COUNT* организуется очередь из этих узлов. Связи для очереди размещаются в поле *COUNT*, которое к данному моменту уже выполнило свою предыдущую функцию. Для большей наглядности в алгоритме используется обозначение *QLINK*[*k*] вместо *COUNT*[*k*], когда это поле уже не используется как счётчик.

Алгоритм *T*. (*Топологическая сортировка.*) Здесь вводятся пары отношений $j < k$ ($1 \le j \le n$, $1 \le k \le n$, $j \ne k$), говорящие, что объект j предшествует объекту k в некотором отношении упорядочения. Результатом является множество всех объектов, расположенных в линейном порядке. Используются внутренние таблицы *QLINK*[0], *COUNT*[1] = *QLINK*[1], *COUNT*[2] = *QLINK*[2], …, *COUNT*[*n*] = *QLINK*[*n*]; *TOP*[1], *TOP*[2], …, *TOP*[*n*]; пул памяти с одним узлом (с полями *SUC* и *NEXT*) для каждой вводимой пары; P – переменная связи для ссылки на узлы в пуле памяти; F и R – целочисленные переменные, используемые для ссылок в начало и

конец очереди, связи которой находятся в таблице *QLINK*; *N* – переменная, указывающая число невыведенных объектов.

***T*1.** [Начальная установка.] Ввести значение *n*. Для $k = 1, 2, \ldots, n$ $COUNT[k] \leftarrow 0$, $TOP[k] \leftarrow \Lambda$. $N \leftarrow n$.

***T*2.** [Следующее отношение.] Ввести следующее отношение $j < k$; если же входная информация исчерпана, то перейти к шагу *T*4.

***T*3.** [Регистрация отношения.] Значение $COUNT[k]$ увеличить на единицу. $P \Leftarrow AVAIL$, $SUC(P) \leftarrow k$, $NEXT(P) \leftarrow TOP[j]$, $TOP[j] \leftarrow P$. Перейти к шагу *T*2.

***T*4.** [Поиск нулей.] (К этому моменту завершена фаза ввода и входная информация преобразована в машинное представление, показанное на рис. 6. Производим теперь начальную установку очереди вывода, которая связывается по полю *QLINK*.) $R \leftarrow 0$, $QLINK[0] \leftarrow 0$. Для $k = 1, 2, \ldots, n$ просмотреть $COUNT[k]$ и, если он нулевой, $QLINK[R] \leftarrow k$, $R \leftarrow k$. После выполнения этих действий для всех *k* реализовать $F \leftarrow QLINK[0]$ (в нём окажется первое встреченное значение *k*, для которого $COUNT[k]$ был нулевым).

***T*5.** [Вывод из начала очереди.] Вывести значение *F*. Если *F* равно 0, то перейти к шагу *T*8; в противном случае $N \leftarrow N - 1$, $P \leftarrow TOP[F]$. (Поскольку таблицы *QLINK* и *COUNT* перекрываются, мы имеем $QLINK[R] = 0$; следовательно, условие $F = 0$ выполняется, когда очередь пуста.)

***T*6.** [Стирание отношения.] Если $P = \Lambda$, то перейти к шагу *T*7; в противном случае уменьшить $COUNT[SUC(P)]$ на единицу и, если при этом он стал нулевым, $QLINK(R) \leftarrow SUC(P)$ и $R \leftarrow SUC(P)$. $P \leftarrow NEXT(P)$ и повторить данный шаг. (Мы исключаем из системы все отношения вида $F < k$ для некоторого *k* и помещаем новые узлы в очередь, когда все их предшественники были выведены.)

***T*7.** [Исключение из очереди.] $F \leftarrow QLINK[F]$ и вернуться к шагу *T*5.

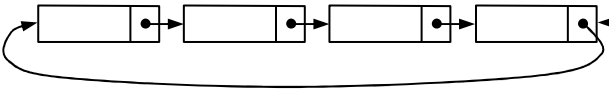
***T*8.** [Окончание процесса.] Конец алгоритма. Если $N = 0$, то выведены все номера элементов в требуемом топологическом порядке и вслед за ними нуль. В противном случае *N* невыведенных номеров содержат цикл. ■

В этом алгоритме удачно сочетаются методы последовательной и связанной памяти. Для главной таблицы $X[1], X[2], \ldots, X[n]$ используется последовательная память, поскольку на шаге *T*3 алгоритм ссылается на «произвольные» части этой таблицы. Связанная память используется для таблиц «непосредственных преемников», поскольку элементы этих таблиц вводятся в произвольном порядке. Очередь узлов, ожидающих вывода, хранится внутри последовательной таблицы, причём узлы связываются в соответствии с порядком их вывода. Для связи вместо адреса используется индекс таблицы, т.е. когда начальным узлом очереди является узел $X[k]$, то мы имеем *F* равно *k*, а не *F* равно $LOC(X[k])$. Операции над оче-

редью, которые использованы на шагах *T*4, *T*6 и *T*7, не идентичны операциям (18) и (20), поскольку в этой системе мы пользуемся специальными свойствами очереди и нет необходимости создавать узлы или возвращать их в свободное пространство.

Время выполнения алгоритма оценивается значением выражения $c_1m + c_2n$, где m – количество вводимых отношений, n – количество объектов; c_1 и c_2 – константы.

Циклический список – связанный список такой, что связь его последнего узла не пустая, а идёт к первому узлу этого же списка:



(23)

Здесь *PTR* – указатель на самый правый узел списка; *LINK*(*PTR*) – адрес самого левого узла.

Наиболее важными операциями являются:

a) Включение *Y* слева: $P \Leftarrow AVAIL$, $INFO(P) \leftarrow Y$, $LINK(P) \leftarrow LINK(PTR)$, $LINK(PTR) \leftarrow P$.

b) Включение *Y* справа: Включение *Y* слева, затем $PTR \leftarrow P$.

c) Запись в *Y* значения из левого узла и исключение этого узла из списка: $P \leftarrow LINK(PTR)$, $Y \leftarrow INFO(P)$, $LINK(PTR) \leftarrow LINK(P)$, $AVAIL \Leftarrow P$.

Описание операций (*a*), (*b*) и (*c*) не учитывает того, что циклический список может быть пустым. Условившись, что *PTR* равен Λ в случае пустого списка, уточним упомянутые операции:

a) Включение *Y* слева: $P \Leftarrow AVAIL$, $INFO(P) \leftarrow Y$. Если $PTR = \Lambda$, то $PTR \leftarrow LINK(P) \leftarrow P$; иначе $LINK(P) \leftarrow LINK(PTR)$, $LINK(PTR) \leftarrow P$.

b) Включение *Y* справа: Включение *Y* слева, затем $PTR \leftarrow P$.

c) Запись в *Y* значения из левого узла и исключение этого узла из списка: Если $PTR = \Lambda$, то НЕХВАТКА; иначе $P \leftarrow LINK(PTR)$, $Y \leftarrow INFO(P)$, $LINK(PTR) \leftarrow LINK(P)$, $AVAIL \Leftarrow P$ и если $PTR = P$, то $PTR \leftarrow \Lambda$.

Заметим, что операции (*a*), (*b*) и (*c*) являются операциями дека, ограниченного по выходу. Это значит, что циклический список можно использовать или как стек (операции (*a*) и (*c*)), или как очередь (операции (*b*) и (*c*)).

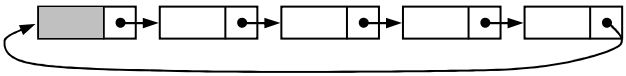
Применительно к циклическим спискам эффективны и некоторые другие важные операции:

d) Очистка списка: Если $PTR \neq \Lambda$, то $P \leftarrow AVAIL$, $AVAIL \leftarrow LINK(PTR)$, $LINK(PTR) \leftarrow P$.

e) Включение циклического списка L_2 в циклический список L_1: Если $PTR_2 \neq \Lambda$ и $PTR_1 \neq \Lambda$, то $P \leftarrow LINK(PTR_1)$, $LINK(PTR_1) \leftarrow LINK(PTR_2)$, $LINK(PTR_2) \leftarrow P$, $PTR_1 \leftarrow PTR_2$, $PTR_2 \leftarrow \Lambda$.

Расщепление одного циклического списка на два представляет ещё одну простую операцию.

Таким образом, циклические списки можно использовать не только для циклических, но и для линейных структур. Тогда возникает вопрос о конце циклического списка. Ответом на него может быть включение в каждый циклический список специального, отличимого от всех, узла, называемого *головой списка*.

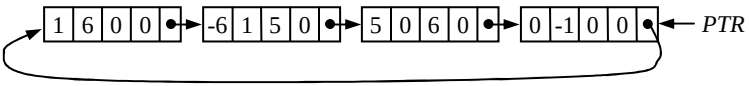


Одним из преимуществ циклического списка с головным узлом является то, что он никогда не будет пустым. При ссылке на списки такого типа вместо указателя на правый конец списка используется обычно голова списка, которая часто находится в фиксированной ячейке памяти.

В качестве примера использования циклических списков рассмотрим арифметическую операцию сложения многочленов. Предположим, что многочлен представлен в виде списка, в котором каждый узел, кроме специального, обозначает ненулевой член и имеет следующий вид:

COEF	*A*	*B*	*C*	*LINK*

Здесь *COEF* является коэффициентом при члене $x^A y^B z^C$. У специального узла *COEF* = 0 и *A* = – 1. Для полей *A*, *B* и *C* узла списка далее будет использоваться обобщённое обозначение *ABC*. Узлы списка всегда располагаются в *порядке убывания* поля *ABC*, за исключением специального узла, после которого находится узел с наибольшим значением *ABC*. Например, многочлен $x^6 - 6xy^5 + 5y^6$ будет представлен следующим образом:



Алгоритм *A*. (*Сложение многочленов.*) Этот алгоритм прибавляет многочлен, на который указывает *P*, к многочлену, на который указывает *Q*. При этом список по указателю *P* не изменяется, а в списке по указателю *Q* будет получена сумма двух многочленов. В конце алгоритма переменные *P* и *Q* принимают свои исходные значения. В алгоритме используются также вспомогательные указатели Q_1 и Q_2.

***A*1.** [Начальная установка.] $P \leftarrow LINK(P)$, $Q_1 \leftarrow Q$, $Q \leftarrow LINK(Q)$. (Теперь P и Q указывают на старшие члены многочленов. Переменная Q_1 в алгоритме почти везде будет «на один шаг отставать» от Q, т.е. будет выполняться равенство $Q = LINK(Q_1)$.)

***A*2.** [Сравнение $ABC(P)$ и $ABC(Q)$.] Если $ABC(P) < ABC(Q)$, то $Q_1 \leftarrow Q$, $Q \leftarrow LINK(Q)$ и выполнить этот шаг сначала. Если $ABC(P) = ABC(Q)$, то перейти к шагу *A*3. Если $ABC(P) > ABC(Q)$, то перейти к шагу *A*5.

***A*3.** [Сложение коэффициентов.] (Найдены два члена с равными степенями.) Если $ABC(P) < 0$, то конец алгоритма; в противном случае $COEF(Q) \leftarrow COEF(Q) + COEF(P)$. Теперь если $COEF(Q) = 0$, то перейти к шагу *A*4; в противном случае $Q_1 \leftarrow Q$, $P \leftarrow LINK(P)$, $Q \leftarrow LINK(Q)$ и вернуться к шагу *A*2.

***A*4.** [Исключение нулевого члена.] $Q_2 \leftarrow Q$, $LINK(Q_1) \leftarrow Q \leftarrow LINK(Q)$ и $AVAIL \Leftarrow Q_2$. (Нулевой член, образовавшийся на шаге *A*3, исключён из многочлена Q.) $P \leftarrow LINK(P)$ и вернуться к шагу *A*2.

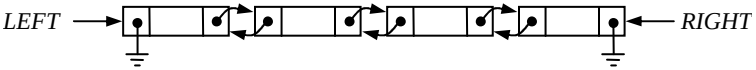
***A*5.** [Включение нового члена.] (Многочлен P содержит член, который не представлен в многочлене Q и поэтому мы включаем его в многочлен Q.) $Q_2 \Leftarrow AVAIL$, $COEF(Q_2) \leftarrow COEF(P)$, $ABC(Q_2) \leftarrow ABC(P)$, $LINK(Q_2) \leftarrow Q$, $LINK(Q_1) \leftarrow Q_2$, $Q_1 \leftarrow Q_2$, $P \leftarrow LINK(P)$ и вернуться к шагу *A*2. ■

В рассмотренном алгоритме значением отношения:

- $ABC(P) < ABC(Q)$ следует считать значение логического выражения $A(P) + B(P) + C(P) < A(Q) + B(Q) + C(Q)$ **или** $A(P) + B(P) + C(P) = A(Q) + B(Q) + C(Q)$ **и** ($A(P) < A(Q)$ **или** $A(P) = A(Q)$ **и** $B(P) < B(Q)$);
- $ABC(P) = ABC(Q)$ – значение логического выражения $A(P) = A(Q)$ **и** $B(P) = B(Q)$ **и** $C(P) = C(Q)$;
- $ABC(P) > ABC(Q)$ – значение **истина**, если ложны первое и второе логические выражения, а значение **ложь** в противном случае.

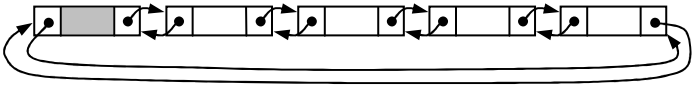
Время выполнения реализующей данный алгоритм программы для гипотетической вычислительной машины *MIX* составляет $(29m' + 18m'' + 29p' + 8q' + 13)u$, где m' – количество подобных и взаимно уничтожающихся членов; m'' – количество подобных, но взаимно не уничтожающихся членов; p' – количество членов в многочлене P, которым нет подобных в многочлене Q; q' – количество членов в многочлене Q, которым нет подобных в многочлене P; u – время выполнения машиной одного такта. Во время выполнения алгоритма в пуле памяти необходимо иметь не менее чем $(2 + p + q)$ и самое большое $(2 + p + q + p')$ узлов.

Списки с двумя связями позволяют достичь ещё большей гибкости в работе с линейными списками. При этом каждый элемент списка имеет две связи *LLINK* и *RLINK*, которые указывают на элементы, находящиеся по обе стороны от данного узла:



(24)

С таким представлением линейного списка легко выполняются операции для дека общего вида. Ещё легче оперировать с двусвязным списком, если его частью является *головной узел*:


(25)

Если такой список пуст, то оба поля связи в его голове указывают на саму голову. Представление (25) полностью удовлетворяет условию

$$RLINK(LLINK(X)) = LLINK(RLINK(X)) = X,$$

где X – адрес любого узла в списке, включая голову. В этом и заключается главная причина предпочтения (25) по сравнению с (24).

Упражнения

1. Выполнить по алгоритму T топологическую сортировку элементов 1, 2, 3, ..., 9 для заданных отношений между ними: $9 < 2$, $3 < 7$, $7 < 5$, $5 < 8$, $8 < 6$, $4 < 6$, $1 < 3$, $7 < 4$, $9 < 5$, $2 < 8$.

2. Используя графическую иллюстрацию модели памяти вычислительной машины, проследить за ходом выполнения алгоритма A, реализующего прибавление многочлена $xy + xz - x$ к многочлену $-xz + 2x - 2y$. Определить последовательность выполнения шагов алгоритма и итоговый многочлен.

4. МАССИВЫ И ОРТОГОНАЛЬНЫЕ СПИСКИ

Массивы (двумерные и более высокой размерности) являются одним из простейших обобщений линейных списков. Рассмотрим в качестве примера матрицу:

$$\begin{pmatrix} A[0,0] & A[0,1] & \ldots & A[0,n] \\ A[1,0] & A[1,1] & \ldots & A[1,n] \\ \ldots & \ldots & \ldots & \ldots \\ A[m,0] & A[m,1] & \ldots & A[m,n] \end{pmatrix}.$$

В таком двумерном массиве некоторый узел $A[j, k]$ принадлежит двум линейным спискам: списку j-й строки $A[j, 0]$, $A[j, 1]$, …, $A[j, n]$ и списку k-го столбца $A[0, k]$, $A[1, k]$, …, $A[m, k]$. Эти списки ортогональных строк и столбцов, по существу, и определяют двумерную структуру матрицы.

Рассмотрим последовательное распределение памяти при хранении массивов.

Если массив хранится в последовательных ячейках памяти, то память обычно распределяется так, что:

$$LOC\,(A\,[\,j,\,k]) = A_0 + A_1 j + A_2 k, \tag{26}$$

где A_0, A_1 и A_2 – константы. Это означает, что при любом изменении j или k легко вычислить адрес узла $A\,[\,j,\,k]$.

Самым естественным и наиболее часто используемым способом распределения памяти является способ, при котором массив располагается в памяти в лексикографическом порядке индексов: $A\,[0,\,0]$, $A\,[0,\,1]$, …, $A\,[0,\,n]$, $A\,[1,\,0]$, $A\,[1,\,1]$, …, $A\,[1,\,n]$, …, $A\,[m,\,0]$, $A\,[m,\,1]$, …, $A\,[m,\,n]$.

В общем случае, если задан k-мерный массив с элементами $A\,[I_1,\,I_2,\,\ldots,\,I_k]$ длины c и $0 \le I_1 \le d_1$, $0 \le I_2 \le d_2$, …, $0 \le I_k \le d_k$, то можно хранить его в памяти так, что:

$$\begin{aligned} LOC\,(A\,[I_1,\,I_2,\,\ldots,\,I_k]) &= LOC\,(A\,[0,\,0,\,\ldots,\,0]) + \\ + c\,(d_2 + 1)\,\ldots\,(d_k + 1)\,I_1 + \ldots + c\,(d_k + 1)\,I_{k-1} &+ c\,I_k = \\ = LOC\,(A\,[0,\,0,\,\ldots,\,0]) + \sum_{1 \le r \le k} a_r I_r\,, & \end{aligned} \tag{27}$$

где $a_r = c \prod_{r<s\le k} (d_s + 1)$.

Например, двумерный массив для $d_1 = 3$, $d_2 = 3$ и $c = 2$ байта будет размещён в памяти следующим образом:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A[0,0]		A[0,1]		A[0,2]		A[0,3]		A[1,0]		A[1,1]		A[1,2]		A[1,3]	

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
A[2,0]		A[2,1]		A[2,2]		A[2,3]		A[3,0]		A[3,1]		A[3,2]		A[3,3]	

В соответствии с формулой (27) адрес его некоторого элемента $A\,[I_1,\,I_2]$ можно определить по выражению:

$$\begin{aligned} LOC\,(A[I_1,\,I_2]) &= LOC\,(A[0,\,0]) + c\,(d_2 + 1)\,I_1 + c\,I_2 = \\ &= LOC\,(A[0,\,0]) + 2\,(3 + 1)\,I_1 + 2\,I_2 = \\ &= LOC\,(A[0,\,0]) + 8\,I_1 + 2\,I_2. \end{aligned}$$

Пусть $I_1 = 3$ и $I_2 = 1$, тогда:

$$LOC\,(A[3, 1]) = LOC\,(A[0, 0]) + 8 \cdot 3 + 2 \cdot 1 =$$
$$= LOC\,(A[0, 0]) + 26 = 0 + 26 = 26.$$

Приведённый выше метод хорош для хранения массивов, имеющих *полную* прямоугольную структуру. Однако во многих случаях приходится иметь дело с *треугольной матрицей* элементов $A\,[\,j, k]$ для $0 \le k \le j \le n$:

$$\begin{pmatrix} A[0, 0] & & & \\ A[1, 0] & A[1, 1] & & \\ \ldots & \ldots & & \\ A[n, 0] & A[n, 1] & \ldots & A[n, n] \end{pmatrix}.$$

Такую матрицу можно использовать, когда известно, что все другие элементы матрицы равны нулю или $A\,[\,j, k] = A\,[k, j]$. Поэтому необходимо и достаточно хранить в памяти немного больше половины значений: $\frac{1}{2}(n+1)(n+2)$. Если хранить эти элементы в последовательных ячейках памяти, то вместо линейного распределения (26) можно использовать распределение в форме:

$$LOC\,(A[\,j, k]) = A_0 + c\,f_1(\,j) + c\,f_2(k),$$

где $f_1(\,j) = \frac{j(j+1)}{2}$, $f_2(k) = k$. Следовательно, получаем довольно простую формулу:

$$LOC\,(A[j, k]) = LOC\,(A\,[0, 0]) + c\,\frac{j(j+1)}{2} + c\,k.$$

Например, для треугольной матрицы вида

$$\begin{pmatrix} A[0, 0] & & & \\ A[1, 0] & A[1, 1] & & \\ A[2, 0] & A[2, 1] & A[2, 2] & \\ A[3, 0] & A[3, 1] & A[3, 2] & A[3, 3] \end{pmatrix}$$

при $c = 2$ получаем такое распределение памяти:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A[0, 0]		A[1, 0]		A[1, 1]		A[2, 0]		A[2, 1]		A[2, 2]		A[3, 0]		A[3, 1]		A[3, 2]		A[3, 3]	

и адрес элемента $A[3, 2]$:

$$LOC\,(A[3, 2]) = LOC\,(A[0, 0]) + 2\,\frac{3(3+1)}{2} + 2 \cdot 2 = 0 + 12 + 4 = 16.$$

В некоторых случаях, когда имеются две треугольные матрицы одинакового размера, их можно объединить в одну. Пусть треугольные матрицы $A[j, k]$ и $B[j, k]$ определены для $0 \le k \le j \le n$. Тогда их можно объединить в матрицу C так, что элемент $C[j, k]$ будет определён для $0 \le j \le n$, $0 \le k \le n + 1$ и

$$A[j, k] = C[j, k], \quad B[j, k] = C[k, j + 1].$$

Таким образом, две треугольные матрицы плотно упаковываются вместе, занимая $(n + 1)(n + 2)$ ячеек:

$$\begin{pmatrix} C[0,0] & C[0,1] & C[0,2] & \dots & C[0,n+1] \\ C[1,0] & C[1,1] & C[1,2] & \dots & C[1,n+1] \\ \dots & \dots & \dots & \dots & \dots \\ C[n,0] & C[n,1] & C[n,2] & \dots & C[n,n+1] \end{pmatrix} \equiv \begin{pmatrix} A[0,0] & B[0,0] & B[1,0] & \dots & B[n,0] \\ A[1,0] & A[1,1] & B[1,1] & \dots & B[n,1] \\ \dots & \dots & \dots & \dots & \dots \\ A[n,0] & A[n,1] & A[n,2] & \dots & B[n,n] \end{pmatrix}$$

и может быть использована линейная адресация вида (26).

Рассмотрим связанное распределение памяти при хранении массивов.

Связанное распределение памяти вполне естественно для многомерных массивов информации. При этом в общем случае любой элемент массива должен содержать k полей связи, по одному на каждый из списков, в которые входит данный элемент. Связанная память обычно используется в тех случаях, когда массивы по своей форме не являются строго прямоугольными.

Использование связанного распределения рассмотрим подробно на примере *разреженных матриц* – матриц, в которых большинство элементов равно нулю. Идея заключается в том, что для экономии памяти не отводить в ней место под нулевые элементы.

Обсудим представление, которое состоит из циклически связанных *ортогональных* списков для каждой строки и каждого столбца матрицы. Пусть узлы списка имеют формат:

LEFT		*UP*
ROW	*COL*	*VAL*

где *ROW* и *COL* – индексы соответственно строки и столбца; *VAL* – значение элемента матрицы; *LEFT* и *UP* – связи со следующими ненулевыми элементами соответственно слева в строке и сверху в столбце. Для каждой i-й строки и j-го столбца имеются специальные головные узлы списков $BASEROW[i]$ и $BASECOL[j]$ такие, что:

$$COL(LOC(BASEROW[i])) < 0 \quad \text{и} \quad ROW(LOC(BASECOL[j])) < 0.$$

Связь *LEFT* в *BASEROW*[*i*] указывает на самый правый узел в *i*-й строке и связь *UP* в *BASECOL*[*j*] – на самый нижний узел в *j*-м столбце.

Например, матрица

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix} \qquad (28)$$

будет представлена так, как показано на рис. 7.

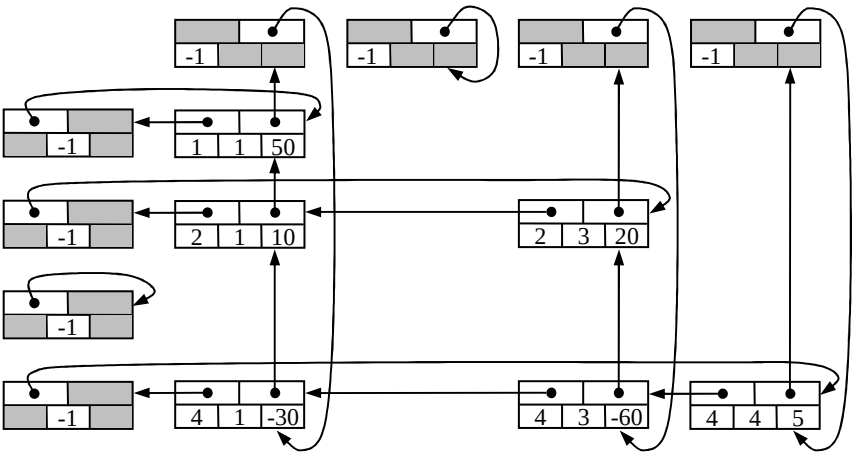


Рис. 7. Машинное представление разреженной матрицы

При последовательном распределении памяти эта матрица заняла бы больше слов, а при увеличении её размеров экономия становится ещё более существенной. При этом время, расходуемое на доступ к произвольному элементу матрицы, остаётся в разумных пределах, поскольку в каждой строке и каждом столбце встречается немного элементов. Кроме того, в большинстве матричных алгоритмов осуществляется последовательное прохождение матрицы и поэтому связанное представление приводит к незначительной потере скорости работы.

В качестве характерного примера рассмотрим операцию *осевого шага* для разреженных матриц, которая играет важную роль в алгоритмах решения систем линейных уравнений, обращения матриц и линейного программирования (симплекс-метод):

$$\begin{array}{l} \\ \text{Осева строка} \\ \\ \text{Люба друга строка} \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & \text{Осевой столбец} & & \text{Любой другой столбец} & \\ \end{array} \\ \begin{pmatrix} & \dots & & \dots & \\ \dots & a & \dots & b & \dots \\ & \dots & & \dots & \\ \dots & c & \dots & d & \dots \\ & \dots & & \dots & \end{pmatrix} \end{array} \quad \begin{array}{c} \text{Осевой шаг} \\ \rightarrow \end{array} \quad \begin{array}{c} \begin{array}{ccccc} & \text{Осевой столбец} & & \text{Любой другой столбец} & \\ \end{array} \\ \begin{pmatrix} & \dots & & \dots & \\ \dots & \dfrac{1}{a} & \dots & \dfrac{b}{a} & \dots \\ & \dots & & \dots & \\ \dots & -\dfrac{c}{a} & \dots & d-\dfrac{bc}{a} & \dots \\ & \dots & & \dots & \end{pmatrix} \end{array} \tag{29}$$

Так, осевой шаг применительно к матрице (28) с осевым элементом 10 приводит к следующей матрице:

$$\begin{pmatrix} -5 & 0 & -100 & 0 \\ 0.1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 \end{pmatrix}.$$

Отметим два случая в реализации осевого шага:

1) если в (29) $b \neq 0$, $c \neq 0$ и $d = 0$, то в представлении разреженной матрицы нет узла для образующегося ненулевого элемента и поэтому необходимо включить новый узел;

2) если $b \neq 0$, $c \neq 0$, $d \neq 0$ и $d - \dfrac{bc}{a} = 0$, то необходимо исключить имеющийся узел.

Алгоритм *S*. (*Осевой шаг в разреженной матрице*.)

Для эффективного выполнения операций включения и исключения узлов вводится таблица указателей *PTR*[*j*], по одному указателю на каждый столбец матрицы. Благодаря этим переменным обеспечивается возможность изменения связей узлов как в горизонтальном, так и в вертикальном измерениях. Алгоритм обрабатывает строки матрицы последовательно снизу вверх, а переменная *PIVOT* указывает на осевой элемент.

***S*1.** [Начальная установка.] *I*0 ← *ROW*(*PIVOT*), *J*0 ← *COL*(*PIVOT*), *ALPHA* ← 1.0/*VAL*(*PIVOT*), *VAL*(*PIVOT*) ← 1.0, *P*0 ← *LOC*(*BASEROW*[*I*0]), *Q*0 ← *LOC*(*BASECOL*[*J*0]).

***S*2.** [Обработка осевой строки.] *P*0 ← *LEFT*(*P*0), *J* ← *COL*(*P*0). Если *J* < 0, то перейти к шагу *S*3 (осевая строка пройдена до конца); в противном случае *PTR*[*J*] ← *LOC*(*BASECOL*[*J*]), *VAL*(*P*0) ← *ALPHA* · *VAL*(*P*0) и повторить шаг *S*2.

***S*3.** [Поиск новой строки.] *Q*0 ← *UP*(*Q*0), *I* ← *ROW*(*Q*0). Если *I* < 0, то конец алгоритма. Если *I* = *I*0, то повторить шаг *S*3; в противном случае *P* ← *LOC*(*BASEROW*[*I*]), *P*1 ← *LEFT*(*P*).

S4. [Поиск нового столбца.] $P0 \leftarrow LEFT(P0)$, $J \leftarrow COL(P0)$. Если $J < 0$, то $VAL(Q0) \leftarrow -ALPHA \cdot VAL(Q0)$ и вернуться к $S3$. Если $J = J0$, то выполнить этот шаг сначала.

S5. [Поиск элемента I, J.] Если $COL(P1) > J$, то $P \leftarrow P1$, $P1 \leftarrow LEFT(P)$ и повторить этот шаг сначала. Если $COL(P1) = J$, то перейти к $S7$; в противном случае перейти к следующему шагу.

S6. [Включение элемента I, J.] Если $ROW(UP(PTR[J])) > I$, то $PTR[J] \leftarrow UP(PTR[J])$ и повторить этот шаг сначала; в противном случае $X \Leftarrow AVAIL$, $VAL(X) \leftarrow 0$, $ROW(X) \leftarrow I$, $COL(X) \leftarrow J$, $LEFT(X) \leftarrow P1$, $UP(X) \leftarrow UP(PTR[J])$, $LEFT(P) \leftarrow X$, $UP(PTR[J]) \leftarrow X$, $P1 \leftarrow X$, $PTR[J] \leftarrow X$.

S7. [Осевая операция.] $VAL(P1) \leftarrow VAL(P1) - VAL(Q0) \cdot VAL(P0)$. Если $VAL(P1) = 0$, то перейти к следующему шагу; в противном случае $PTR[J] \leftarrow P1$, $P \leftarrow P1$, $P1 \leftarrow LEFT(P)$ и вернуться к $S4$.

S8. [Исключение элемента I, J.] Если $UP(PTR[J]) \neq P1$, то $PTR[J] \leftarrow UP(PTR[J])$ и повторить этот шаг сначала; в противном случае $UP(PTR[J]) \leftarrow UP(P1)$, $LEFT(P) \leftarrow LEFT(P1)$, $AVAIL \Leftarrow P1$, $P1 \leftarrow LEFT(P)$. Вернуться к шагу $S4$. ■

Заметим, что в узлах $BASEROW[I]$ и $BASECOL[J]$ используются только два поля. Поэтому для остальных полей можно не отводить место в памяти.

Время работы алгоритма S приблизительно пропорционально количеству элементов матрицы, которые изменяются при выполнении осевой операции.

Упражнения

1. Выполнить по алгоритму S осевой шаг в разреженной матрице с ненулевыми элементами $m[1][1] = 50$, $m[2][3] = 20$, $m[2][1] = 10$, $m[4][4] = 5$, $m[4][3] = -60$, $m[4][1] = -30$, если осевым элементом является $m[2][1] = 10$. Привести последовательность выполненных шагов алгоритма и конечную матрицу в виде, аналогичном виду исходной матрицы (по строкам с убыванием второго индекса в строке).

5. ДЕРЕВЬЯ И ПРЕДСТАВЛЕНИЕ ДЕРЕВЬЕВ

Деревья являются наиболее важными нелинейными структурами. Под *деревом* понимают конечное множество T, состоящее из одного или более узлов таких, что:

a) имеется один специально обозначенный узел, называемый *корнем* данного дерева;

b) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся подмножествах $T_1, T_2, \ldots, T_m$, каждое из которых в

свою очередь является деревом. Деревья T_1, T_2, …, T_m называют *поддеревьями* данного корня.

Каждый узел дерева является корнем некоторого поддерева, которое содержится в этом дереве. Число поддеревьев данного узла называется *степенью* этого узла. Узел с нулевой степенью называется *концевым узлом* (*листом*). Неконцевые узлы часто называют *узлами разветвления*. Каждый узел дерева T находится на некотором *уровне*, при этом корень имеет уровень 1, корни его поддеревьев имеют уровень 2 и т.д.

Например, на рис. 8 узел A – корень дерева { A, B, C, D, E, F, G}; подмножества {B} и {C, D, E, F, G} – поддеревья дерева с корнем A; узел C – корень поддерева {C, D, E, F, G}; узел C имеет уровень 2 относительно всего дерева; дерево {C, D, E, F, G} имеет три поддерева {D}, {E} и {F, G}; степень узла C равна 3; B, D, E и G – концевые узлы; F – единственный узел степени 1; G – единственный концевой узел уровня 4.

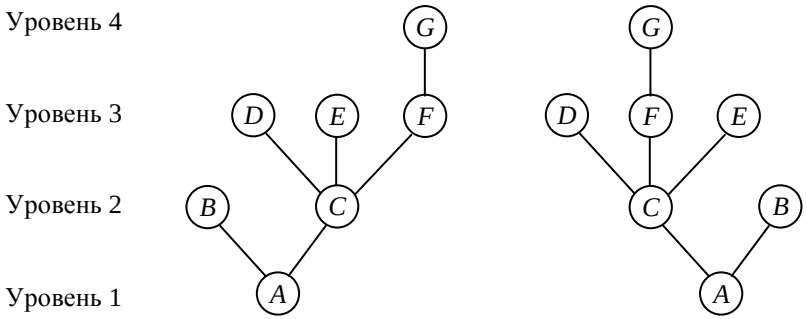


Рис. 8. Деревья

Если в пункте b) в определении дерева имеет значение относительный порядок поддеревьев T_1, T_2, …, T_m, то будем говорить, что дерево является *упорядоченным*; иначе − *ориентированным* деревом. Заметим, что в ориентированном дереве имеет значение только относительная ориентация узлов, а не их порядок. *Все рассматриваемые деревья будем считать упорядоченными, если не оговорено противное.*

Так, деревья на рис. 8 являются различными, если их считать упорядоченными деревьями, и одинаковыми, если эти деревья считать ориентированными.

Лесом называется множество, состоящее из некоторого числа непересекающихся деревьев.

Деревья далее будем изображать с корнем вверху и пользоваться терминологией, употребляемой при описании генеалогических деревьев. Поэтому каждый корень дерева является *отцом* корней своих поддеревьев, а последние являются *братьями* между собой и *сыновьями* своего

отца. Заметим, что корень всего дерева отца не имеет. Кроме того, будем употреблять слова *предок* и *потомок* для обозначения родства, которое может простираться на несколько уровней дерева.

Например, в дереве на рис. 9 узел *C* имеет трёх сыновей *D*, *E* и *F*; узел *E* – отец узла *G*; узлы *B* и *C* – братья; узлы *D*, *E*, *F* и *G* – потомки узла *C*; узлы *A*, *C* и *E* – предки узла *G*.

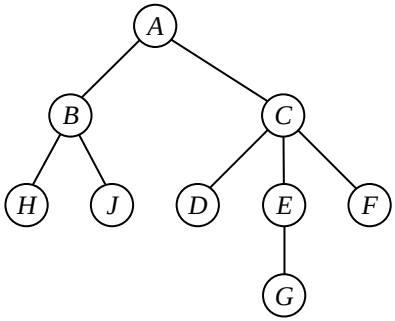


Рис. 9. Дерево с корнем вверху

Наиболее важным типом деревьев являются бинарные деревья.

Бинарное дерево это конечное множество узлов, которое или пусто, или состоит из корня и самое большее двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного дерева.

Так, если два дерева на рис. 10 считать бинарными деревьями, то они различны.

Рис. 10. Различные бинарные деревья

Заметим, что деревья можно представлять и другими способами: вложенными множествами (для ориентированных деревьев), вложенными скобками, уступчатым списком и десятичной системой Дьюи.

Например, дерево, заданное на рис. 9 графом, изображено на рис. 11 с помощью вложенных множеств.

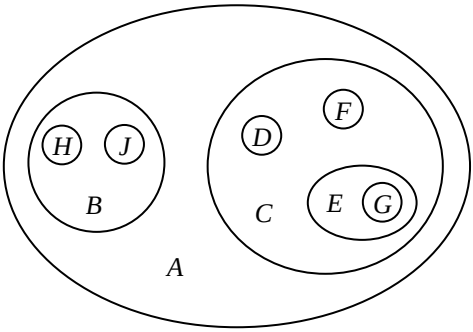


Рис. 11. Дерево, представленное вложенными множествами

Представление этого же дерева с помощью вложенных скобок будет иметь вид (*A* (*B* (*H*) (*J*)) (*C* (*D*) (*E* (*G*)) (*F*))), а уступчатым списком следующим образом:

A xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
B xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
H xxxxxxxxxxxxxxxxxxxxxx
J xxxxxxxxxxxxxxxxxxxxxx
C xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
D xxxxxxxxxxxxxxxxxxxxxx
E xxxxxxxxxxxxxxxxxxxxxx
G xxxxxxxxxxxxxxxxxx
F xxxxxxxxxxxxxxxxxxxxxxx

И, наконец, в десятичной системе Дьюи это дерево может быть задано так: 1 *A* ; 1.1 *B*; 1.1.1 *H*; 1.1.2 *J*; 1.2 *C*; 1.2.1 *D*; 1.2.2 *E*; 1.2.2.1 *G*; 1.2.3 *F*.

Существует тесная связь между десятичной системой Дьюи и способом обозначения переменных, снабжённых индексами. Поэтому, всякий прямоугольный массив можно рассматривать как частный случай древовидной структуры. Например, на рис. 12 представлена в виде дерева матрица *A* размером 2×3.

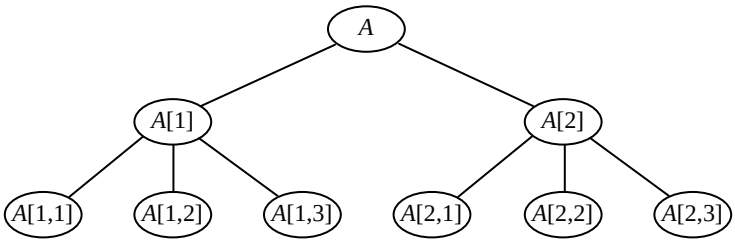


Рис. 12. Матрица, представленная деревом

Заметим, что связь в строках матрицы представлена в дереве явно, а в столбцах – нет.

В виде деревьев также можно представлять и арифметические выражения. Например, бинарное дерево на рис. 13 соответствует арифметическому выражению $a - b \times (c / d + e / f)$.

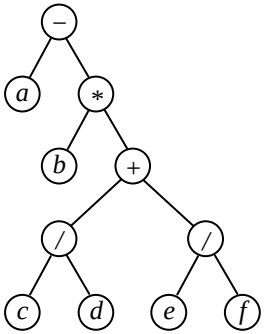


Рис. 13. Арифметическое выражение в виде бинарного дерева

Для представления бинарных деревьев в памяти ЭВМ достаточно иметь указатель T (указатель на дерево) и в каждом узле две связи *LLINK* и *RLINK*. Если дерево пусто, то $T = \Lambda$; в противном случае T – адрес корня этого дерева, а *LLINK*(T) и *RLINK*(T) – указатели соответственно на левое и правое поддерево этого корня.

Так, например, бинарное дерево на рис. 14 в памяти ЭВМ будет представлено в виде, изображённом на рис. 15.

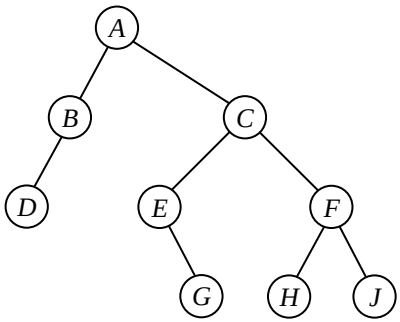


Рис. 14. Бинарное дерево

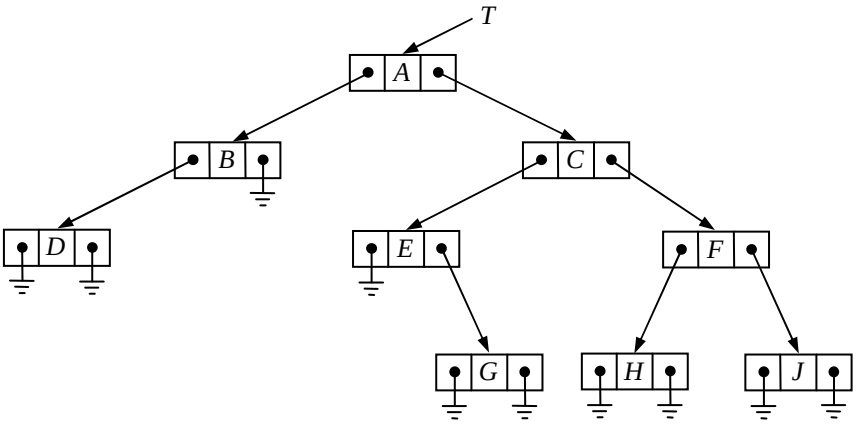


Рис. 15. Представление бинарного дерева в памяти ЭВМ

Упражнения

1. Нарисуйте всевозможные различные деревья с тремя узлами *A*, *B*, *C* и корнем в узле *A*. Сколько существует всего различных деревьев с тремя узлами *A*, *B* и *C*?

2. Нарисуйте всевозможные различные ориентированные деревья с тремя узлами *A*, *B*, *C* и корнем в узле *A*. Сколько существует всего различных ориентированных деревьев с тремя узлами *A*, *B* и *C*?

3. Узел *A* имеет трёх братьев, а узел *B* – его отец. Нарисуйте соответствующий фрагмент дерева и установите, чему равна степень узла *B*.

4. Предположим, что узел *X* некоторого дерева в десятичной системе Дьюи имеет обозначение $a_1.a_2. \ldots a_k$. Каковы обозначения узлов этого дерева на пути от узла *X* к корню?

5. Разработайте систему условных обозначений для узлов бинарных деревьев, которая соответствовала бы системе обозначений Дьюи, и приведите пример бинарного дерева с обозначениями узлов в разработанной системе.

6. Нарисуйте бинарное дерево, соответствующее арифметическому выражению:

а) $2\ (a - b/c)$;

б) $a + b + 5c$.

6. ПРОХОЖДЕНИЕ ДЕРЕВЬЕВ

Все алгоритмы обработки древовидных структур неизменно связаны с прохождением деревьев. Под *прохождением дерева* понимается способ методичного исследования его узлов, при котором каждый узел проходится точно один раз. Полное прохождение дерева даёт линейную расстановку его узлов.

Рассмотрим три способа прохождения бинарного дерева: прямой порядок, обратный порядок и концевой порядок. Каждый из них выполняется в три этапа.

Прямой порядок: попасть в корень, пройти левое поддерево, пройти правое поддерево.

Обратный порядок: пройти левое поддерево, попасть в корень, пройти правое поддерево.

Концевой порядок: пройти левое поддерево, пройти правое поддерево, попасть в корень.

При прямом порядке прохождения дерева (см. рис. 14 из предыдущего параграфа) узлы располагаются в порядке *A, B, D, C, E, G, F, H, J*; при обратном − в порядке *D, B, A, E, G, C, H, F, J* и при концевом − *D, B, G, E, H, J, F, C, A*.

Рассмотрим алгоритм прохождения бинарного дерева в обратном порядке.

Алгоритм *T*. (*Обход бинарного дерева в обратном порядке*.)

Пусть T − указатель на бинарное дерево, имеющее представление, аналогичное рис. 15; A − вспомогательный стек.

***T*1.** [Начальная установка.] Очистить стек A; $P \leftarrow T$.

***T*2.** [$P = \Lambda$?] Если $P = \Lambda$, перейти к шагу *T*4.

***T*3.** [Стек $\Leftarrow P$] (Теперь P указывает на непустое бинарное дерево, которое нужно пройти.) $A \Leftarrow P$, $P \leftarrow LLINK(P)$ и вернуться к шагу *T*2.

***T*4.** [$P \Leftarrow$ стек] Если стек A пуст, то работа алгоритма заканчивается; в противном случае $P \Leftarrow A$.

***T*5.** [Попасть в P] «Попасть» в $NODE(P)$, $P \leftarrow RLINK(P)$ и вернуться к шагу *T*2. ■

Для удобства дальнейших рассуждений введём следующие обозначения. Если P указывает на узел бинарного дерева, то:

P* – адрес преемника узла $NODE(P)$ в прямом порядке;

P\$ – адрес преемника узла $NODE(P)$ в обратном порядке;

P# – адрес преемника узла $NODE(P)$ в концевом порядке;

*P – адрес предшественника узла $NODE(P)$ в прямом порядке;

\$$P$ – адрес предшественника узла $NODE(P)$ в обратном порядке;

#P – адрес предшественника узла $NODE(P)$ в концевом порядке.

Тогда *(P*) = (**P*)* = P, \$(P\$) = (\$$P$)\$ = P и т.д.

Пусть *INFO*(*P*) – буква, изображённая в поле *INFO* узла *NODE*(*P*) дерева, представленного на рис. 15. Тогда, если *P* указывает на корень этого дерева, имеем: *INFO*(*P*) = *A*, *INFO*(*P**) = *B*, *INFO*(*P*$) = *E*, *INFO*($*P*) = *B*, *INFO*(#*P*) = *C*.

Заметим, что в рассматриваемом дереве, как, впрочем, и в любом бинарном дереве, имеется больше нулевых связей, чем других указателей. Поэтому для более эффективного использования памяти ЭВМ применяют представления бинарных деревьев в виде так называемых *прошитых* деревьев. При этом нулевые связи заменяют *связями-нитями*, идущими в другие части дерева и облегчающими его прохождение. Прошитое дерево, эквивалентное дереву на рис. 15, изображено на рис. 16. На нём штриховые линии обозначают связи-нити, которые всегда идут к узлу, находящемуся выше. Теперь каждый узел имеет две связи: или две обычные связи, идущие к левому и правому поддеревьям (например, узел *C*); или две связи-нити (например, узел *H*); или по одной связи каждого типа (например, узел *B*). Нити, исходящие из *D* и *J*, особые, они возникают в «самом левом» и «самом правом» узлах, а их значение будет объяснено позже.

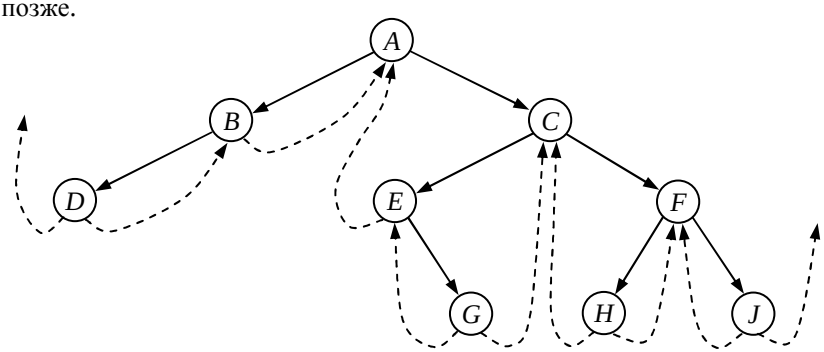


Рис. 16. Прошитое бинарное дерево

Различие штриховых и сплошных связей обеспечивается с помощью двух дополнительных однобитовых полей *LTAG* и *RTAG* в каждом узле списка таких, что:

если *LTAG*(*P*) = “−”, то *LLINK*(*P*) = $*P*;
если *LTAG*(*P*) = “+”, то *LLINK*(*P*) = *Q* ≠ Λ;
(30)
если *RTAG*(*P*) = “−”, то *RLINK*(*P*) = *P*$;
если *RTAG*(*P*) = “+”, то *RLINK*(*P*) = *Q* ≠ Λ;

На рисунках 17 и 18 представлены схемы проведения левых и правых связей-нитей в простейшем (k = 0) и общем (k > 0) случаях. Здесь

ломаные линии обозначают обычные связи или связи-нити, ведущие к другим узлам дерева.

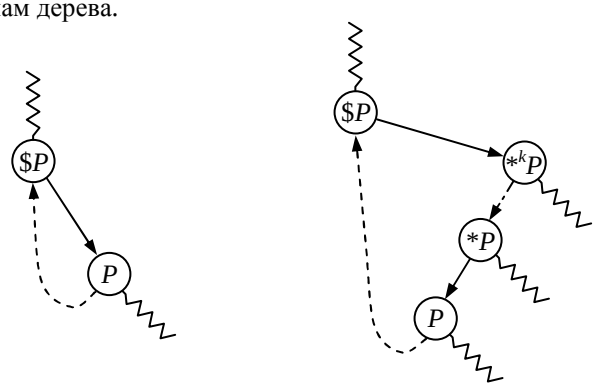


Рис. 17. Схемы проведения левых связей-нитей

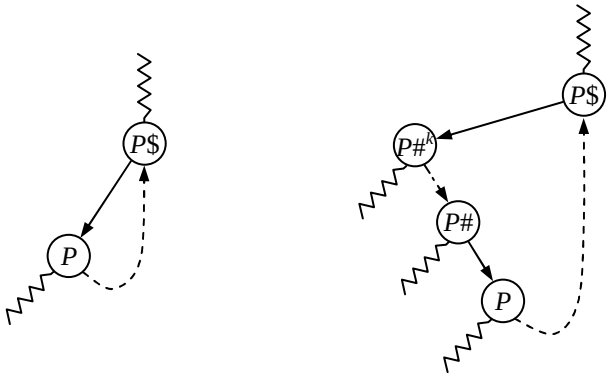


Рис. 18. Схемы проведения правых связей-нитей

Таким образом, каждая связь-нить указывает непосредственно на предшественника или на преемника данного узла в дереве при обратном (симметричном) порядке прохождения.

Большое преимущество прошитых деревьев состоит в том, что упрощаются алгоритмы их прохождения. Рассмотрим алгоритм, который определяет *P*\$ при заданном *P*.

Алгоритм *S*. (*Обратный* (*симметричный*) *преемник в прошитом бинарном дереве*.)

Если *P* указывает на узел бинарного дерева, то этот алгоритм переменной *Q* присваивает значение *P*\$.

***S*1.** [*RLINK*(*P*) − нить?] *Q* ← *RLINK*(*P*). Если *RTAG*(*P*) = ”−”, то закончить алгоритм.

S2. [Поиск влево.] Если *LTAG*(*Q*) = ”+”, то *Q* ← *LLINK*(*Q*) и повторить этот шаг; в противном случае закончить алгоритм. ■

Заметим, что в данном случае стек не требуется, как при решении этой же задачи с помощью алгоритма *T*.

Если в алгоритме *S* поменять *L* на *R* и *R* на *L*, перевернув тем самым дерево так, что левые и правые поддеревья поменяются местами, то получим следующий алгоритм определения $*P* по заданному *P*.

Алгоритм *S’*. (*Обратный* (*симметричный*) *предшественник в прошитом бинарном дереве*.)

Если *P* указывает на узел бинарного дерева, то этот алгоритм переменной *Q* присваивает значение $*P*.

S’1. [*LLINK*(*P*) − нить?] *Q* ← *LLINK*(*P*). Если *LTAG*(*P*) = ”−”, то закончить алгоритм.

S’2. [Поиск вправо.] Если *RTAG*(*Q*) = ”+”, то *Q* ← *RLINK*(*Q*) и повторить этот шаг; в противном случае закончить алгоритм. ■

Представление в памяти ЭВМ бинарного дерева (см. рис. 14) как прошитого дерева может иметь вид связанного списка с головным узлом, изображённого на рис. 19. Заметим, что особые нити, возникшие на рис. 16 в «самом левом» и «самом правом» узлах этого дерева, здесь направлены к головному узлу.

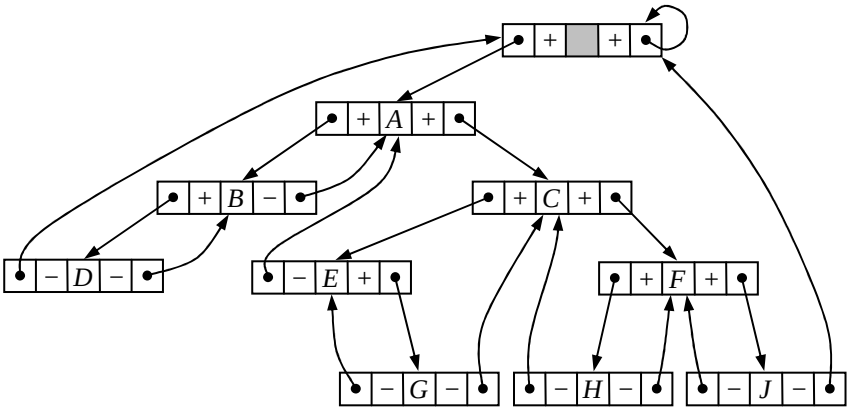


Рис. 19. Представление прошитого бинарного дерева в памяти ЭВМ

Следует отметить, что для выполнения алгоритма *T* прохождения дерева с пустыми связями требуется (15*n* + *a* + 3) единиц времени, а для алгоритма прохождения прошитого дерева – (11*n* − *a* + 8) единиц, где *n* – число узлов в дереве, *a* – число концевых правых связей.

Анализируя два рассмотренных подхода к прохождению бинарных деревьев, можно сделать следующие выводы:

− прохождение прошитых деревьев осуществляется несколько быстрее, чем непрошитых;

− для алгоритма *T* требуется больший объём памяти ЭВМ из-за необходимости применения стека;

− использование алгоритма *S* даёт возможность перейти от *P* к *P*\$, не проходя всё бинарное дерево.

Таким образом, прошитое бинарное дерево с точки зрения его прохождения имеет преимущества перед непрошитым. Заметим, что упомянутые преимущества прошитых деревьев свелись бы на нет, если бы в самом начале было затруднительно проведение связей-нитей. Однако идея прошитых деревьев вполне работоспособна в силу того, что прошить деревья довольно легко. Это можно сделать с помощью следующего алгоритма.

Алгоритм *I*. (*Включение узла в прошитое бинарное дерево*.)

Данный алгоритм присоединяет одиночный узел *NODE*(*Q*) в качестве правого поддерева узла *NODE*(*P*), если правое поддерево пусто (т.е. если *RTAG*(*P*) = ”−”), и включает *NODE*(*Q*) между *NODE*(*P*) и *NODE*(*RLINK*(*P*)) в противном случае. Предполагается, что бинарное дерево, в котором производится включение, является прошитым как на рис. 19.

***I*1.** [Установка признаков и связей.]

RLINK(*Q*) ← *RLINK*(*P*), *RTAG*(*Q*) ← *RTAG*(*P*), *RLINK*(*P*) ← *Q*, *RTAG*(*P*) ← ”+”, *LLINK*(*Q*) ← *P*, *LTAG*(*Q*) ← ”−”.

***I*2.** [*RLINK*(*P*) − нить?]

Если *RTAG*(*Q*) = ”+”, то *LLINK*(*Q*\$) ← *Q*. (Здесь адрес *Q*\$ определяется алгоритмом *S*, который будет работать должным образом, даже если *LLINK*(*Q*\$) теперь указывает на узел *NODE*(*P*), а не на узел *NODE*(*Q*). Необходимость в этом шаге возникает только в том случае, когда происходит включение в середину прошитого дерева, а не просто включение «нового листа».) ■

Заметим, что поменяв ролями левое и правое поддеревья, а также заменив *Q*\$ на \$*Q* на шаге *I*2, получим алгоритм *I* ’, производящий подобным же образом включение влево.

Кроме рассмотренных представлений нигде не прошитых и всюду прошитых деревьев имеется представление, называемое *правопрошитыми* бинарными деревьями. В них пустые правые поддеревья представляются связями-нитями *RLINK*, а для пустых левых поддеревьев *LLINK* = Λ. Аналогичным способом строятся *левопрошитые* бинарные деревья. Если в алгоритме *S* на шаге *S*2 проверку *LTAG* = ”+” заменить на ”*LLINK* ≠ Λ”, то получим алгоритм для прохождения в обратном порядке правопрошитых бинарных деревьев. Отметим, что на практике чаще всего встречается случай, когда требуется проведение нитей только с одной стороны.

Далее рассмотрим алоритм получения копии бинарного дерева на новом месте памяти.

Алгоритм *C*. (*Копирование бинарного дерева*.)

Пусть *HEAD* – адрес головы списка бинарного дерева *T* (т.е. *T* является левым поддеревом узла по адресу *HEAD*; *LLINK*(*HEAD*) − указатель на дерево). Пусть *NODE*(*U*) – узел (не дерева *T*) с пустым левым поддеревом. Рассматриваемый алгоритм копирует дерево *T*, и копия становится левым поддеревом узла *NODE*(*U*). В частности, если *NODE*(*U*) является головой списка пустого бинарного дерева, то алгоритм превращает пустое дерево в копию дерева *T*.

***C*1.** [Начальная установка.]

$P \leftarrow HEAD$, $Q \leftarrow U$. Перейти к шагу *C*4.

***C*2.** [Поместить справа ?]

Если узел *NODE*(*P*) имеет непустое правое поддерево, то $R \Leftarrow AVAIL$ и присоединить узел *NODE*(*R*) справа от узла *NODE*(*Q*). (В начале шага *C*2 правое поддерево узла *NODE*(*Q*) пусто.)

***C*3.** [Скопировать *INFO*.]

INFO(*Q*) ← *INFO*(*P*). (Здесь *INFO* обозначает информацию из всех полей узла, кроме информации из полей *LLINK*, *RLINK*, *LTAG*, *RTAG*.)

***C*4.** [Поместить слева?]

Если узел *NODE*(*P*) имеет непустое левое поддерево, то $R \Leftarrow AVAIL$ и присоединить узел *NODE*(*R*) слева от узла *NODE*(*Q*). (В начале шага *C*4 левое поддерево узла *NODE*(*Q*) пусто.)

***C*5.** [Вперёд.]

$P \leftarrow P^*$, $Q \leftarrow Q^*$.

***C*6.** [Конец?]

Если $P = HEAD$ (или, что то же самое, если $Q = RLINK(U)$, в предположении, что узел *NODE*(*U*) имеет непустое правое поддерево), работа алгоритма заканчивается; в противном случае вернуться к шагу *C*2. ■

Заметим, что этот алгоритм служит типичным примером применения процедуры прохождения деревьев. В данном случае он применим к прошитым, непрошитым и частично прошитым деревьям. Для прошитых деревьев включение узлов на шагах *C*2 и *C*4 осуществляется с помощью алгоритмов *I* и *I* ’. Определение прямых преемников P^* и Q^* на шаге *C*5 в прошитом дереве может выполнить описанный ниже алгоритм, который для заданного значения *P* присваивает переменной *Q* значение P^*.

Если $LTAG(P) = \text{”+”}$, то $Q \leftarrow LLINK(P)$ и конец алгоритма; в противном случае $Q \leftarrow P$ и выполнять $Q \leftarrow RLINK(Q)$ до тех пор, пока не получим $RTAG(Q) = \text{”+”}$ (может быть и ни разу), $Q \leftarrow RLINK(Q)$. ■

Упражнения

1. С помощью алгоритма *T* прохождения бинарного дерева в обратном порядке определить линейное расположение узлов бинарного дерева, заданного в системе обозначений Дьюи (0 соответствует левому, а 1 – правому поддереву): 1 *A*; 10 *B*; 11 *C*; 101 *D*; 111 *E*; 1010 *F*; 1011 *G*; 1111 *H*; 10100 *J*.

2. Прошить бинарное дерево из упр. 1 и пройти полученное прошитое дерево с помощью алгоритма *S*.

3. Используя графическую иллюстрацию, выполните с помощью алгоритма *I* включение нового узла *K* в бинарное дерево, представленное на рис. 19, если переменная *P* указывает на:

а) узел *D*;

б) узел *C*.

4. Выполните с помощью алгоритма *I'* (используя графическую иллюстрацию) включение нового узла *K* в бинарное дерево, представленное на рис. 19, если переменная *P* указывает на:

а) узел *E*;

б) узел *C*.

5. Используя графическую иллюстрацию, выполните с помощью алгоритма *C* копирование бинарного дерева, представленного на рис. 19, если узел *NODE*(*U*) является головой списка пустого бинарного дерева.

7. ПРЕДСТАВЛЕНИЕ ЛЕСОВ БИНАРНЫМИ ДЕРЕВЬЯМИ

Лесом называется упорядоченное множество, состоящее из некоторого, быть может и равного нулю, числа деревьев. Всякий лес может быть естественным образом представлен в виде бинарного дерева. Рассмотрим, например, лес из двух деревьев, изображённый на рис. 20.

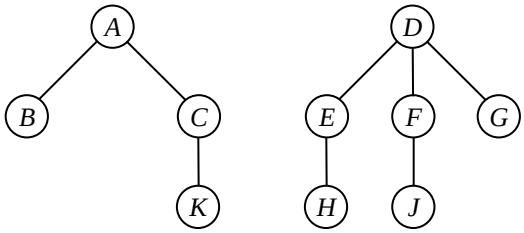


Рис. 20. Лес из двух деревьев

Соответствующее этому лесу бинарное дерево можно получить, если соединить вместе корни деревьев, а также сыновей в каждой семье и убрать затем негоризонтальные связи, оставив только связи, идущие от отцов к их первым сыновьям (рис. 21).

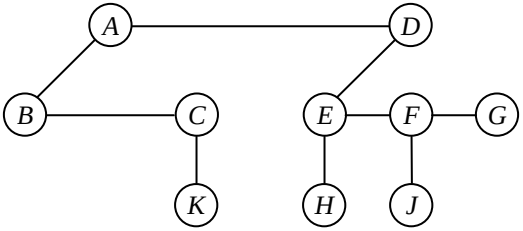


Рис. 21. Преобразованный лес из двух деревьев

Повернув затем некоторые части графа примерно на 45^{o} по часовой стрелке вокруг соответствующих узлов, получим искомое представление бинарного дерева, изображённое на рис. 22.

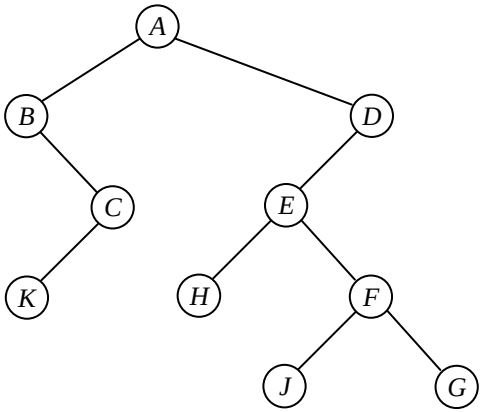


Рис. 22. Бинарное дерево, представляющее лес из двух деревьев

Выполняя обратные действия в обратном порядке, можно из бинарного дерева получить соответствующий ему лес.

Описанные преобразования называются *естественным соответствием* между лесами и бинарными деревьями.

Иногда для удобства граф бинарного дерева изображают так, как на рис. 21, т.е. не выполняя поворотов по часовой стрелке. Прошитое бинарное дерево, соответствующее рассматриваемому лесу, представлено на рис. 23. Здесь штриховые линии обозначают правые, а пунктирные – левые связи-нити. Заметим, что правые связи-нити идут от самого правого сына к отцу семьи.

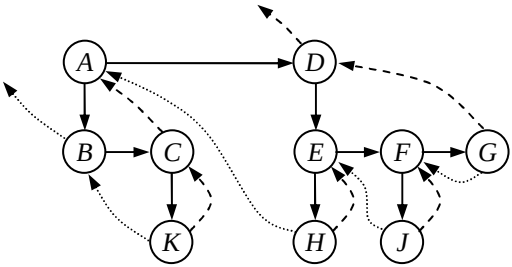


Рис. 23. Прошитое бинарное дерево, представляющее лес из двух деревьев

Исходя из отмеченного выше естественного соответствия между лесами и бинарными деревьями, идеи прохождения, изложенные для бинарных деревьев, можно применить к лесам (и, следовательно, к деревьям). Известные нам три порядка прохождения принимают такой вид.

Прямой порядок: попасть в корень первого дерева, пройти поддеревья первого дерева (в прямом порядке), пройти оставшиеся деревья (в прямом порядке).

Обратный порядок: пройти поддеревья первого дерева (в обратном порядке), попасть в корень первого дерева, пройти оставшиеся деревья (в обратном порядке).

Концевой порядок: пройти поддеревья первого дерева (в концевом порядке), пройти оставшиеся деревья (в концевом порядке), попасть в корень первого дерева.

Для того чтобы понять значение этих трёх методов прохождения, рассмотрим следующую запись леса с помощью вложенных скобок:

$$(A\ (B,\ C\ (K)),\ D\ (E\ (H),\ F\ (J),\ G)) \qquad (31)$$

Эта запись соответствует нашему лесу из двух деревьев. В ней дерево представлено так, что сначала записан корень дерева, а затем представлены его поддеревья аналогичным образом. Следовательно, представлением непустого леса служит заключённый в скобки список представлений его деревьев, разделяемых запятыми. Если наш лес проходится в прямом порядке, то получается последовательность *ABCKDEHFJG*, а это не что иное, как выражение (31) без скобок и запятых. Заметим, что прямой порядок является династическим порядком. После смерти короля, его титул переходит к его первому сыну, а затем к потомкам первого сына, и если же все таковые умирают, то титул таким же самым образом переходит к другим сыновьям этой семьи.

Обратным порядком прохождения узлов этого же леса является последовательность *BKCAHEJFGD*. Она получается таким же способом, как и для прямого порядка, но только в соответствующем представлении леса с помощью вложенных скобок каждый узел записывается после своих потомков, а не перед ними:

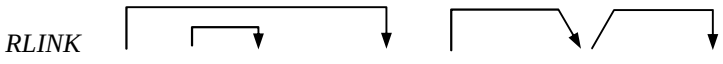
$$((B, (K)\ C)\ A, ((H)\ E, (J)\ F, G)\ D) \qquad (32)$$

Узлы нашего леса в концевом порядке образуют последовательность *KCBHJGFEDA*. Это на первый взгляд несколько странное расположение узлов довольно просто объясняется. В строке символов (31) или (32) необходимо двигаться слева направо до тех пор, пока не будет достигнута первая правая скобка. Затем начать проходить узлы, двигаясь справа налево до тех пор, пока не будет достигнута левая скобка. Потом из всей строки вычеркнуть полностью группу символов, включая эти скобки. В полученной строке опять двигаться слева направо до первой правой скобки и проходить узлы справа налево до тех пор, пока не будет достигнута левая скобка и так далее, пока строка не станет пустой.

Помимо рассмотренного выше представления лесов, которое условно можно назвать «*левый сын – правый брат*», существует ещё много способов представления древовидных структур в компьютере. Из них, прежде всего, используют методы *последовательного* распределения памяти.

Один из способов представления лесов методом последовательного распределения памяти заключается, по существу, в том, что опускаются связи *LLINK*.

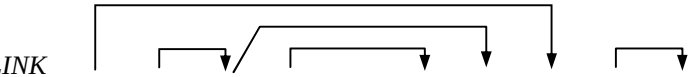
В случае представления нашего леса в прямом порядке при последовательном распределении узлы располагаются в прямом порядке с полями *INFO*, *RLINK*, *LTAG* в каждом узле:



(33)

Здесь непустые связи *RLINK* отмечены стрелками, а связи *LLINK* не нужны, поскольку такая связь была бы нулевой или указывала бы на следующий элемент последовательности.

Реверсивный к концевому порядок представления леса при последовательном распределении памяти отличается от прямого порядка тем, что опускаются связи *RLINK*, а связи *LLINK* сохраняются:



LLINK											
INFO	*A*	*D*	*E*	*F*	*G*	*J*	*H*	*B*	*C*	*K*	(34)
RTAG	+		+	+				+			

Порядок, реверсивный к концевому, называют ещё *фамильным* порядком, поскольку все сыновья одной семьи стоят рядом, например *EFG*.

Третье последовательное представление леса в обратном порядке несколько отличается от первых двух. Здесь узлы размещаются в обратном порядке и регистрируются не связи, а степень каждого узла:

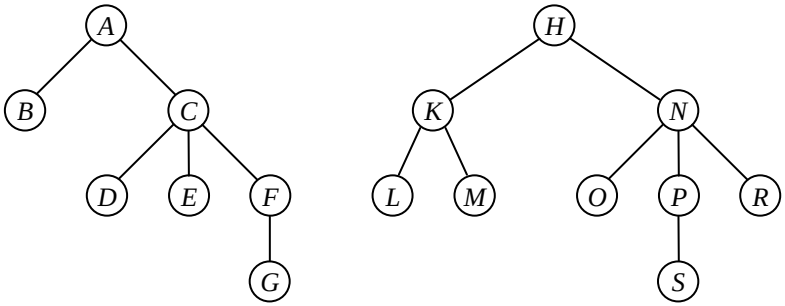
Степень	0	0	1	2	0	1	0	1	0	3
INFO	*B*	*K*	*C*	*A*	*H*	*E*	*J*	*F*	*G*	*D*

(35)

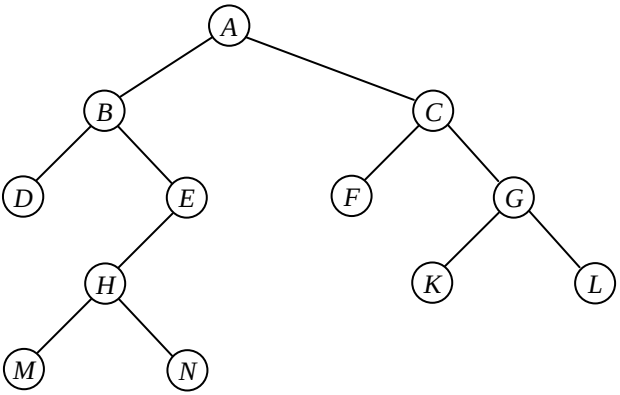
Этой информации достаточно для того, чтобы задать древовидную структуру.

Упражнения

1. Представить в виде бинарного дерева лес:



2. Получить лес, соответствующий бинарному дереву:



3. Определить последовательность прохождения узлов леса из упр. 1, если порядок прохождения узлов:

а) прямой; б) обратный; в) концевой.

4. Представьте лес из упр. 1 с помощью последовательного распределения памяти, если порядок прохождения его узлов:

а) прямой; б) фамильный; в) обратный.

8. ИСПОЛЬЗОВАНИЕ ПОНЯТИЯ ДЕРЕВА ДЛЯ РЕШЕНИЯ ПРИКЛАДНЫХ ЗАДАЧ

Для полной спецификации всякого ориентированного дерева или леса достаточно связи *FATHER* (отец), поскольку зная все восходящие связи можно нарисовать дерево или лес.

Рассмотрим важное практическое приложение, в котором можно обойтись одними восходящими связями, – алгоритм, решающий задачу эквивалентности.

Отношением эквивалентности, обозначаемым символом «≡», называется бинарное отношение между элементами множества M, удовлетворяющее следующим трём свойствам для любых (не обязательно различных между собой) элементов x, y, z из M:

а) если $x \equiv y$ и $y \equiv z$, то $x \equiv z$ (транзитивность);

б) если $x \equiv y$, то $y \equiv x$ (симметричность);

в) $x \equiv x$ (рефлексивность).

Примерами отношений эквивалентности могут служить отношение «=», отношение сравнимости по модулю m для целых чисел, отношение подобия между деревьями и др.

Задача эквивалентности состоит в том, что указываются пары эквивалентных элементов множества M и необходимо установить возможность доказательства эквивалентности двух определённых элементов из этого множества.

Пусть множество $M = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ и даны следующие пары эквивалентных элементов:

$$1 \equiv 5,\ 6 \equiv 8,\ 7 \equiv 2,\ 9 \equiv 8,\ 3 \equiv 7,\ 4 \equiv 2,\ 9 \equiv 3. \qquad (36)$$

Отсюда доказывается, что $2 \equiv 6$, поскольку $2 \equiv 7 \equiv 3 \equiv 9 \equiv 8 \equiv 6$, но нельзя доказать, что $1 \equiv 6$. В самом деле, пары эквивалентных элементов из (36) разбивают множество M на два класса эквивалентности:

$$K_1 = \{1, 5\} \text{ и } K_2 = \{2, 3, 4, 6, 7, 8, 9\}, \qquad (37)$$

такие, что элементы 1 и 6 принадлежат различным классам, а, следовательно, не эквивалентны.

Для решения сформулированной выше задачи эквивалентности необходимо сначала определить разбиение множества M на классы эквивалентности типа (37), а затем сделать соответствующие выводы.

Разбиение множества M на классы эквивалентности можно начать с того крайнего случая, когда каждый его элемент один составляет весь свой класс:

$$K_1 = \{1\},\ K_2 = \{2\},\ K_3 = \{3\},\ K_4 = \{4\},\ K_5 = \{5\},$$

$$K_6 = \{6\},\ K_7 = \{7\},\ K_8 = \{8\},\ K_9 = \{9\}. \qquad (38)$$

Теперь, с учётом первого заданного соотношения $1 \equiv 5$, следует объединить классы $K_1 = \{1\}$ и $K_5 = \{5\}$ в один класс $K_1 = \{1, 5\}$. После обработки соотношений $1 \equiv 5$, $6 \equiv 8$ и $7 \equiv 2$ распределение элементов по классам (38) примет такой вид:

$$K_1 = \{1, 5\},\ K_2 = \{2, 7\},\ K_3 = \{3\},\ K_4 = \{4\},\ K_5 = \{6, 8\},\ K_6 = \{9\}. \qquad (39)$$

Далее на основании соотношения $9 \equiv 8$ следует объединить классы $K_5 = \{6, 8\}$ и $K_6 = \{9\}$ с образованием класса $K_5 = \{6, 8, 9\}$ и т.д.

В приведённном ниже алгоритме для представления ситуаций типа (37) – (39) в компьютере и эффективного объединения классов используются древовидные структуры. Элементы множества M становятся узлами леса и *два узла эквивалентны* тогда и только тогда, когда они *принадлежат одному и тому же дереву* (*классу*). Заметим, что два узла принадлежат одному и тому же дереву, когда они расположены ниже одного и того же корня. При объединении классов два дерева легко соединить, просто полагая одно из них новым поддеревом корня другого дерева.

Алгоритм *E*. (*Обработка соотношений эквивалентности*.)

Пусть M – множество целых чисел 1, 2, …, n и пусть $FATHER[1]$, $FATHER[2]$, …, $FATHER[n]$ – целочисленные переменные. Пусть входными данными для этого алгоритма служит некоторое множество соотношений вида (36).

***E*1.** [Начальная установка.] $FATHER[k] \leftarrow 0$ для $k = 1, 2, \ldots, n$. (Это означает, что все деревья первоначально состоят только из корня, как в (38).)

***E*2.** [Ввод новой пары.] Взять из входного потока пару эквивалентных элементов вида $j \equiv k$. Если входной поток исчерпан, то конец алгоритма.

***E*3.** [Найти корни.] Если $FATHER[j] \neq 0$, то $j \leftarrow FATHER[j]$ и повторить этот шаг. Если $FATHER[k] \neq 0$, то $k \leftarrow FATHER[k]$ и повторить этот шаг. (После этой операции j и k перешли в корни двух деревьев, которые надо соединить. Заметим, что вводимое соотношение $j \equiv k$ избыточно тогда и только тогда, когда $j = k$.)

***E*4.** [Соединение деревьев.] Если $j \neq k$, то $FATHER[j] \leftarrow k$. Вернуться на шаг *E*2. ■

Для входных данных (36), вводимых по порядку слева направо, по окончании алгоритма таблица *FATHER* получит следующее заполнение:

FATHER

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

,

что представляет лес, изображённый на рис. 24.

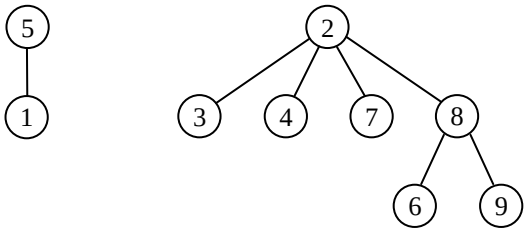


Рис. 24. Лес, определяющий разбиение (37)

Рассмотрим ещё одно важное практическое приложение, использующее понятие дерева при анализе вычислительных алгоритмов. В нём решается задача определения, сколько раз выполняется каждый шаг некоторого алгоритма при однократном его исполнении. Применение деревьев позволяет в этой задаче определить независимые неизвестные и даёт систематический метод их обнаружения.

Пусть алгоритм представлен блок-схемой, изображённой на рис. 25. В ней блоки обозначают шаги алгоритма, буква или цифра внутри блока – сколько раз должен быть выполнен этот шаг за один проход алгоритма. Стрелки указывают возможные переходы между блоками.

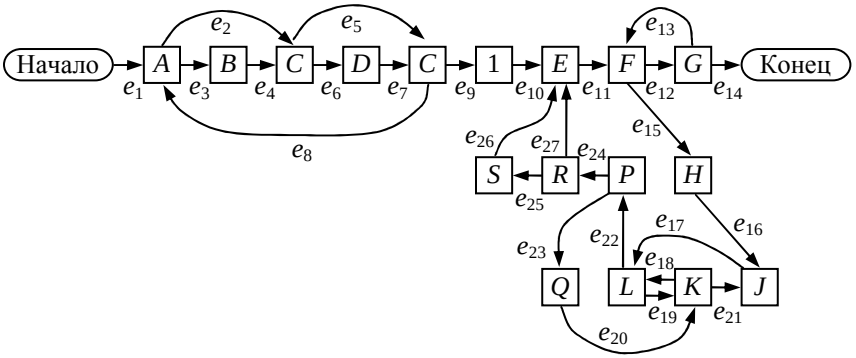


Рис. 25. Блок-схема алгоритма

В задаче требуется найти значения всех величин *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *J*, *K*, *L*, *P*, *Q*, *R* и *S*.

Для её решения вначале обозначим через E_j число, показывающее, сколько раз в процессе выполнения алгоритма проходится стрелка e_j. Тогда, согласно закону Кирхгофа:

$$\begin{gathered}\text{(сумма величин } E\text{, входящих в блок)} = \\ = \text{(величина, находящаяся в блоке)} = \\ = \text{(сумма величин } E\text{, выходящих из блока).}\end{gathered} \tag{40}$$

Например, для блока *K* имеем:

$$E_{19} + E_{20} = K = E_{18} + E_{21}. \tag{41}$$

Далее кроме величин *A*, *B*, …, *S* будем считать неизвестными и величины $E_1, E_2, \ldots, E_{27}$.

Затем абстрагируем блок-схему так, что она примет вид ориентированного графа, изображённого на рис. 26. При этом дуга e_{13} разделена на дуги e'_{13} и e''_{13} для того, чтобы контур состоял не менее чем из трёх дуг. Таким же образом разделена дуга e_{19}. Аналогично нужно было бы поступить и с петлями, если бы они имели место в графе. Для того чтобы закон Кирхгофа был одинаково применим ко всем частям данного графа, введена дуга e_0, направленная из блока «Конец» в блок «Начало».

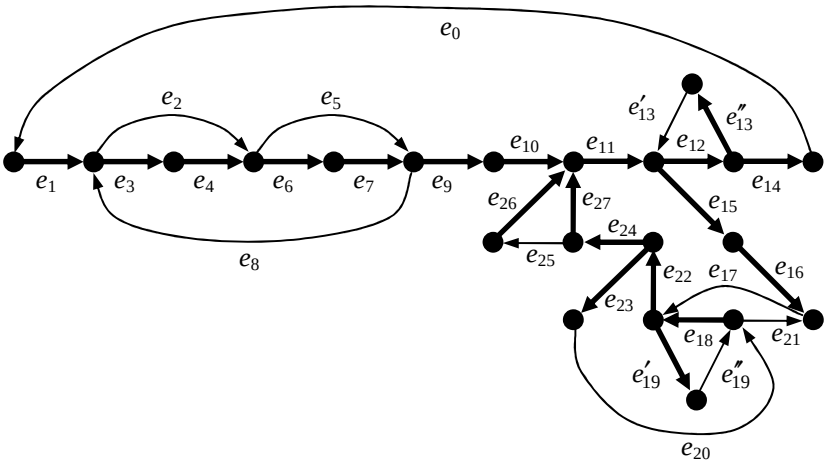


Рис. 26. Абстрактная блок-схема алгоритма

На этом рисунке более жирными линиями выделен остов графа. Добавив к остову любую хорду (дугу графа, не вошедшую в остов), полу-

чим точно один цикл. Например, добавив дугу e_2 к заданному остову, получаем цикл (e_2, e_4, e_3), который алгебраически можно записать в виде «$e_2 - e_3 - e_4$», где знаки плюс и минус показывают, идёт ли цикл по направлению соответствующей дуги или против него.

Выполнив такую процедуру для каждой хорды относительно заданного остова, получим все базисные циклы:

$$
\begin{aligned}
C_0 &: \quad e_0 + e_1 + e_3 + e_4 + e_6 + e_7 + e_9 + e_{10} + e_{11} + e_{12} + e_{14};\\
C_2 &: \quad e_2 - e_3 - e_4;\\
C_5 &: \quad e_5 - e_7 - e_6;\\
C_8 &: \quad e_8 + e_3 + e_4 + e_6 + e_7;\\
C''_{13} &: \quad e''_{13} + e_{12} + e'_{13};\\
C_{17} &: \quad e_{17} + e_{22} + e_{24} + e_{27} + e_{11} + e_{15} + e_{16};\\
C''_{19} &: \quad e''_{19} + e_{18} + e'_{19};\\
C_{20} &: \quad e_{20} + e_{18} + e_{22} + e_{23};\\
C_{21} &: \quad e_{21} - e_{16} - e_{15} - e_{11} - e_{27} - e_{24} - e_{22} - e_{18};\\
C_{25} &: \quad e_{25} + e_{26} - e_{27}.
\end{aligned}
\tag{42}
$$

Величины E_0, E_2 , E_5, E_8, E''_{13} , E_{17}, E''_{19} , E_{20}, E_{21} и E_{25}, соответствующие представленным циклам, следует считать *независимыми* неизвестными величинами, поскольку для любого набора значений этих величин имеется решение уравнений Кирхгофа вида (40), причём единственное:

$$
\begin{aligned}
&E_1 = E_0, && E_{11} = E_0 + E_{17} - E_{21}, && E'_{19} = E''_{19},\\
&E_3 = E_0 - E_2 + E_8, && E_{12} = E_0 + E''_{13}, && E_{22} = E_{17} + E_{20} - E_{21},\\
&E_4 = E_0 - E_2 + E_8, && E'_{13} = E''_{13}, && E_{23} = E_{20},\\
&E_6 = E_0 - E_5 + E_8, && E_{14} = E_0, && E_{24} = E_{17} - E_{21},\\
&E_7 = E_0 - E_5 + E_8, && E_{15} = E_{17} - E_{21}, && E_{26} = E_{25},\\
&E_9 = E_0, && E_{16} = E_{17} - E_{21}, && E_{27} = E_{17} - E_{21} - E_{25}.\\
&E_{10} = E_0, && E_{18} = E''_{19} + E_{20},
\end{aligned}
\tag{43}
$$

Отметим, что при получении соотношений (43) для каждой дуги e_j заданного остова записываем все взятые с соответствующим знаком величины E_k, для которых дуга e_j входит в цикл C_k. Таким образом, матрица коэффициентов системы (43) является просто транспонированной по отношению к матрице коэффициентов системы (42).

Задав значения независимых неизвестных величин E_0, E_2, E_5, E_8, E''_{13} , E_{17}, E''_{19} , E_{20}, E_{21} и E_{25}, можно вычислить по соотношениям (43) величины E_1, E_3, E_4, E_6, E_7, E_9, E_{10}, E_{11}, E_{12}, E'_{13} , E_{14}, E_{15}, E_{16}, E_{18}, E'_{19} , E_{22}, E_{23}, E_{24}, E_{26} и E_{27}. Отметим, что блоки «Начало» и «Конец» алгорит-

ма выполняются точно один раз, поэтому значение E_0 следует считать равным 1.

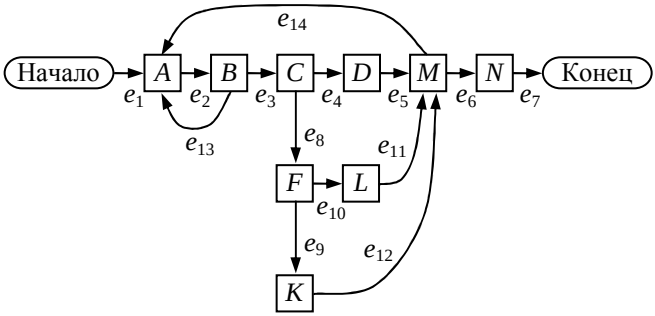
Заканчивая решение задачи, по известным теперь величинам E_0, E_1, E_2, …, E_{27}, можно определить с использованием блок-схемы (см. рис. 25) и соотношений типа (40) сколько раз выполняется каждый блок нашего алгоритма:

$A = E_1 + E_8$,
$B = E_3$,
$C = E_2 + E_4 = E_7$,
$D = E_6$,
$E = E_{10} + E_{26} + E_{27}$,
$F = E_{11} + E_{13}$,
$G = E_{12}$,
$H = E_{15}$,
$J = E_{16} + E_{21}$,
$K = E_{19} + E_{20}$,
$L = E_{17} + E_{18}$,
$P = E_{22}$,
$Q = E_{23}$,
$R = E_{24}$,
$S = E_{25}$.

Упражнения

1. Для множества $M = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ и пар эквивалентных элементов $1 \equiv 6$, $5 \equiv 2$, $7 \equiv 3$, $4 \equiv 7$, $8 \equiv 5$, $6 \equiv 8$, $9 \equiv 8$ выполните алгоритм E обработки соотношений эквивалентности, постройте соответствующий полученным результатам лес и определите разбиение множества M на классы эквивалентности.

2. Применительно к блок-схеме алгоритма:



а) среди величин E_1, E_2, …, E_{14}, обозначающих число прохождений по дугам e_1, e_2, …, e_{14} этой схемы, выбрать (с использованием понятия дерева) независимые величины, а зависимые величины выразить через независимые;

б) вычислить значения зависимых величин, задавшись значениями выбранных независимых величин для однократного прохода по алгоритму;

в) определить, сколько раз будет выполнен каждый шаг этого алгоритма для данных, полученных в предыдущем пункте.

9. СЛОЖНОСТЬ АЛГОРИТМОВ

Рассмотрим задачу нахождения максимального элемента массива среди n элементов $x[1], x[2], \ldots, x[n]$.

Пусть переменная m обозначает максимальный элемент массива, переменная k – порядковый номер элемента массива и переменная j – номер максимального элемента массива. Тогда:

$$m = \max_{1 \le k \le n} x[k] = x[j].$$

Предположим, что значение j при этом должно быть максимально возможным.

Поставленная задача отыскания максимального элемента в массиве может быть решена с помощью алгоритма M, блок-схема которого изображена на рис. 27.

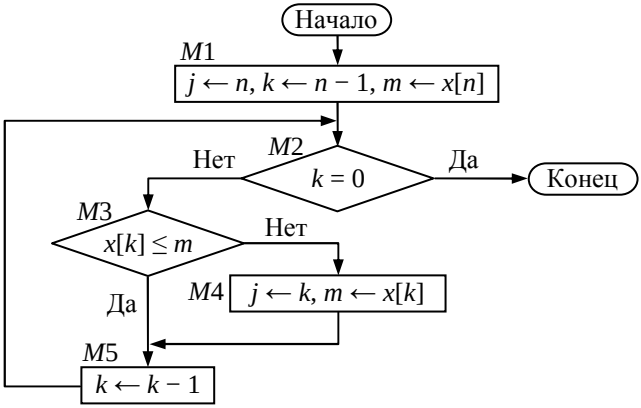


Рис. 27. Блок-схема алгоритма

Проследим за работой этого алгоритма с использованием модели памяти ЭВМ на следующем примере исходных данных:

n	j	k	m
5	5	4	7
	4	3	8
	2	2	10
		1	
		0	

x

6	10	7	8	7
1	2	3	4	5

Здесь блок $M4$ алгоритма был выполнен два раза.

В общем случае анализ сложности алгоритма включает в себя анализ затрачиваемой памяти и анализ времени работы. Заметим, что анализ сложности нескольких алгоритмов, предназначенных для решения одной

и той же задачи, позволяет выбрать среди них наилучший алгоритм (наименьшей сложности).

Приведённый выше алгоритм *M* тривиален, но, тем не менее, на его примере удобно проиллюстрировать детальный анализ алгоритма на сложность.

Отметим, что перед началом работы этого алгоритма необходимо выделить место в оперативной памяти для хранения значений переменных *j*, *k*, *m*, *n* и массива *x*, а в ходе его выполнения дополнительное место в памяти не требуется. Поэтому, с точки зрения *анализа затрачиваемой памяти*, для алгоритма *M* достаточно запоминающего устройства некоторого фиксированного объёма.

Сосредоточим основное внимание на *анализе времени работы* алгоритма *M* и оценим вначале, сколько раз будет выполнен каждый его шаг:

Шаг	Число выполнений
*M*1	1
*M*2	n
*M*3	$n - 1$
*M*4	A
*M*5	$n - 1$

Заметим, что здесь символом *A* обозначено число замен значений текущего максимума, которое является неизвестной случайной величиной, зависящей от конкретных значений элементов массива $x[1], x[2], \ldots, x[n]$. Поскольку время работы алгоритма *M* зависит от случайной величины *A*, анализ заключается в нахождении: минимального, максимального и среднего значения величины *A*, а также квадратичного отклонения величины *A*.

Минимальное значение величины *A* равно нулю. Это случай, когда

$$x[n] = \max_{1 \le k \le n} x[k] .$$

Максимальному значению $n - 1$ величина *A* равна в случае, когда $x[1] > x[2] > \ldots > x[n]$. Среднее значение величины *A* находится между минимальным и максимальным значениями.

Рассмотрим всевозможные случаи соотношений между значениями элементов массива и соответствующие значения величины *A* при условии, что $n = 3$.

Случай	Значение *A*	Случай	Значение *A*
$x[1] < x[2] < x[3]$	0	$x[2] < x[3] < x[1]$	1
$x[1] < x[3] < x[2]$	1	$x[3] < x[1] < x[2]$	1
$x[2] < x[1] < x[3]$	0	$x[3] < x[2] < x[1]$	2

Вероятность того, что случайная величина A имеет значение k обозначим через $p_{n\,k}$. Значение этой вероятности может быть вычислено по формуле:

$$p_{nk} = \frac{(\text{число перестановок порядка } n, \text{ для которых } A = k)}{n!}.$$

Из приведённой выше таблицы следует, что $p_{30} = 2 / 3! = 2 / 6 = 1 / 3$, $p_{31} = 3 / 6 = 1 / 2$, $p_{32} = 1 / 6$.

Среднее значение или математическое ожидание A_n случайной величины A можно определить по формуле

$$A_n = \sum_k k p_{nk} .$$

Заметим, что математическое ожидание A_n обозначают и через *ave A* (от англ. *average* – среднее число, средний).

В нашем примере математическое ожидание величины A:

$$A_3 = \sum_{k=0}^{2} k p_{3k} = 0 \cdot p_{30} + 1 \cdot p_{31} + 2 \cdot p_{32} = 0 \cdot \frac{1}{3} + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{6} = \frac{1}{2} + \frac{1}{3} = \frac{5}{6}.$$

Дисперсия V_n случайной величины A является математическим ожиданием случайной величины $(A - A_n)^2$:

$$V_n = \sum_k (k - A_n)^2 \, p_{nk} = \sum_k k^2 \, p_{nk} - 2 A_n \sum_k k \, p_{nk} + A_n^2 \sum_k p_{nk} =$$

$$= \sum_k k^2 \, p_{nk} - 2 A_n A_n + A_n^2 = \sum_k k^2 \, p_{nk} - A_n^2 .$$

Дисперсию V_n случайной величины A обозначают и как *var A* (от англ. *variation* – изменение).

В отношении к нашему примеру

$$V_3 = \sum_k k^2 \, p_{nk} - A_n^2 = \sum_{k=0}^{2} k^2 p_{3k} - A_3^2 = 0^2 \cdot p_{30} + 1^2 \cdot p_{31} + 2^2 \cdot p_{32} - \left(\frac{5}{6}\right)^2 = =$$

$$0^2 \cdot \frac{1}{3} + 1^2 \cdot \frac{1}{2} + 2^2 \cdot \frac{1}{6} - \left(\frac{5}{6}\right)^2 = \frac{1}{2} + \frac{2}{3} - \frac{25}{36} = \frac{17}{36}.$$

Квадратичное отклонение σ_n случайной величины A (от её математического ожидания) определяют как квадратный корень из дисперсии:

$$\sigma_n = \sqrt{V_n}$$

и обозначают также через *dev A* (от англ. *deviate* – отклонение).

Для нашего примера $\sigma_3 = \sqrt{V_3} = \sqrt{\frac{17}{36}} \approx 0{,}69$.

Опуская вывод зависимостей для математического ожидания *ave A* и квадратичного отклонения *dev A* при произвольном значении *n*, запишем статистические характеристики случайной величины *A* в виде:

$$A = (\min 0,\ \text{ave}\ (H_n - 1),\ \max\ (n-1),\ \text{dev}\ \sqrt{H_n - H_n^{(2)}}\ ),$$

где $H_n = \sum_{k=1}^{n} \frac{1}{k}$ – гармонический числовой ряд, $H_n^{(2)} = \sum_{k=1}^{n} \frac{1}{k^2}$ – гармонический числовой ряд второго порядка.

Подтверждая полученные ранее значения математического ожидания и квадратичного отклонения случайной величины *A* при $n = 3$, получим:

$$H_3 - 1 = \sum_{k=1}^{3} \frac{1}{k} - 1 = 1 + \frac{1}{2} + \frac{1}{3} - 1 = \frac{1}{2} + \frac{1}{3} = \frac{5}{6};$$

$$H_3^{(2)} = \sum_{k=1}^{3} \frac{1}{k^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} = 1 + \frac{1}{4} + \frac{1}{9} = 1\frac{13}{36};$$

$$\sqrt{H_3 - H_3^{(2)}} = \sqrt{1\frac{5}{6} - 1\frac{13}{36}} = \sqrt{\frac{17}{36}} \approx 0{,}69.$$

Время работы алгоритма *M* для решения рассматриваемой задачи отыскания максимального элемента в массиве можно оценить по значению выражения $(7 + 5n + 4A)u$, где *u* – время выполнения одного такта гипотетической вычислительной машиной *MIX*.

Упражнения

1. Оцените минимальное, максимальное и среднее время работы алгоритма *M* при $n = 5$, если этот алгоритм выполняет гипотетическая машина *MIX*.

10. СОРТИРОВКА. ВНУТРЕННЯЯ СОРТИРОВКА. СОРТИРОВКА ПОДСЧЁТОМ

Под *сортировкой* будем понимать переразмещение некоторых объектов в возрастающем или убывающем порядке. Сортировка может быть использована в различных случаях, например, для:

а) решения задачи «группировки», когда нужно собрать вместе все элементы с одинаковым значением некоторого признака. Эту задачу можно решить, расположив элементы в неубывающем порядке;

б) отыскания общих элементов в двух или более файлах. Отсортировав все файлы в одном и том же порядке, можно отыскать в них все общие элементы за один последовательный просмотр без возвратов;

в) поиска нужной информации в листинге (напечатанном машиной документе). Поиск будет гораздо удобнее, если листинг будет отсортирован в алфавитном порядке.

Методы сортировки могут служить хорошей иллюстрацией идей *анализа алгоритмов*, позволяющих оценивать рабочие характеристики алгоритмов и разумно выбирать один среди, казалось бы, равноценных алгоритмов.

Введём некоторые понятия и определим *задачу сортировки*.

Пусть необходимо упорядочить N элементов $R_1, R_2, \ldots, R_N$. Назовём эти элементы *записями*, а совокупность N записей – *файлом*. Каждая запись R_j имеет, кроме всего прочего, *ключ* K_j, который и управляет процессом сортировки.

Задача сортировки заключается в том, чтобы найти такую перестановку $p(1)\ p(2)\ \ldots\ p(N)$ записей, в которой ключи расположились бы в неубывающем порядке:

$$K_{p(1)} \le K_{p(2)} \le \ldots \le K_{p(N)}. \qquad (44)$$

Сортировка называется *устойчивой*, если она удовлетворяет дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке:

$$p(i) < p(j), \text{ если } K_{p(i)} = K_{p(j)} \text{ и } i < j. \qquad (45)$$

В ряде случаев при сортировке может потребоваться физически перемещать записи в памяти; в других случаях достаточно создать вспомогательную таблицу, которая некоторым образом описывает перестановку и обеспечивает доступ к записям в соответствии с порядком их ключей.

Некоторые методы сортировки предполагают существование величин «∞» и «$-\infty$». Величина «∞» считается большей, а величина «$-\infty$» меньшей любого ключа:

$$-\infty < K_j < \infty, \quad j = 1, 2, \ldots, N. \qquad (46)$$

Обычно сортировку подразделяют на два класса: *внутреннюю*, когда все записи хранятся в оперативной памяти, и *внешнюю*, когда все они там не помещаются. При внутренней сортировке имеются более гибкие возможности для построения структур данных и доступа к ним. Внешняя сортировка показывает, как поступать в условиях сильно ограниченного доступа к данным.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время порядка $N \log N$, при этом требуется около $\log N$ «проходов» по данным.

Возможны следующие основные решения задачи внутренней сортировки:

а) *сортировка вставками*. Элементы просматриваются по одному, и каждый новый элемент вставляется в подходящее место среди ранее упорядоченных элементов;

б) *обменная сортировка*. Если два элемента расположены не по порядку, то они меняются местами. Этот процесс повторяется до тех пор, пока все элементы не будут упорядочены;

в) *сортировка посредством выбора*. Сначала выбирается наименьший (или наибольший) элемент и каким-либо образом отделяется от остальных, затем выбирается наименьший (наибольший) из оставшихся и так далее до тех пор, пока не будут выбраны все элементы;

г) *сортировка подсчётом*. Каждый элемент сравнивается со всеми остальными. Окончательное положение элемента определяется после подсчёта числа меньших ключей.

На самом деле разработано множество различных алгоритмов сортировки и каждый из них имеет свои преимущества и недостатки. Только изучив характеристики каждого алгоритма сортировки, можно производить разумный выбор алгоритмов для конкретных приложений.

Рассмотрим *сортировку подсчётом*.

Ниже приведён алгоритм сортировки записей R_1, R_2, …, R_N по ключам K_1, K_2, …, K_N, который использует для подсчёта числа ключей, меньших данного, вспомогательную таблицу *COUNT*[1], *COUNT*[2], …, *COUNT*[*N*]. После завершения алгоритма величина *COUNT*[*j*] + 1 определяет окончательное положение записи R_j.

Алгоритм *C*. (*Сравнение и подсчёт.*)

***C*1.** [Сброс счётчиков.] Установить значения элементов *COUNT*[1], *COUNT*[2], …, *COUNT*[*N*] равными нулю.

***C*2.** [Цикл по *i*.] Выполнить шаг *C*3 для $i = N, N-1, \ldots, 2$ и завершить работу алгоритма.

***C*3.** [Цикл по *j*.] Выполнить шаг *C*4 для $j = i-1, i-2, \ldots, 1$.

***C*4.** [Сравнение ключей K_i, K_j.] Если $K_i < K_j$, то значение *COUNT*[*j*] увеличить на один; в противном случае значение *COUNT*[*i*] увеличить на один. ■

Заметим, что этот алгоритм даёт верный результат независимо от числа равных ключей.

Пусть $N = 16$, а записи в памяти размещены так, что образуют следующую последовательность ключей:

K															
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Тогда по окончании работы алгоритма *C* вспомогательная таблица *COUNT* будет иметь следующий вид:

COUNT

6	1	8	0	15	3	14	4	10	5	2	7	9	11	13	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

и, например, величина $COUNT[3] + 1 = 9$ определяет окончательное положение записи R_3 с ключом 512.

Время работы алгоритма *C* изменяется в пределах от $(3N^2 + 10N - 4)u$ до $(5.5N^2 + 7.5N - 4)u$ в зависимости от исходной последовательности ключей. Среднее время работы находится где-то посередине. Множитель N^2 в выражениях свидетельствует о том, что алгоритм не даёт эффективного способа сортировки при больших значениях *N*.

Рассмотрим ещё одну разновидность сортировки посредством подсчёта. Она эффективна в том случае, когда имеется много равных ключей, и все они попадают в достаточно узкий диапазон.

Алгоритм *D*. (*Распределяющий подсчёт.*)

Этот алгоритм сортирует записи $R_1, R_2, \ldots, R_N$, в которых все ключи целые и лежат в диапазоне $[u, v]$. Он использует вспомогательную таблицу $COUNT[u], \ldots, COUNT[v]$ и в конце своей работы все записи в требуемом порядке переносит в область вывода $S_1, S_2, \ldots, S_N$.

***D*1.** [Сброс счётчиков.] Установить значения $COUNT[u], \ldots, COUNT[v]$ равными нулю.

***D*2.** [Цикл по *j*.] Выполнить шаг *D*3 при $1 \le j \le N$, а затем перейти к шагу *D*4.

***D*3.** [Увеличение $COUNT[K_j]$.] Увеличить значение $COUNT[K_j]$ на 1.

***D*4.** [Суммирование.] (К этому моменту значение $COUNT[i]$ есть число ключей, равных *i*.) $COUNT[i] \leftarrow COUNT[i] + COUNT[i - 1]$ для $i = u + 1, u + 2, \ldots, v$.

***D*5.** [Цикл по *j*.] (К этому моменту значение $COUNT[i]$ есть число ключей, меньших или равных *i*, в частности $COUNT[v] = N$.) Выполнить шаг *D*6 для $j = N, N - 1, \ldots, 1$ и завершить работу алгоритма.

***D*6.** [Вывод R_j.] $i \leftarrow COUNT[K_j]$, $S_i \leftarrow R_j$, $COUNT[K_j] \leftarrow i - 1$. ■

При сформулированных выше условиях это очень быстрая процедура сортировки.

Упражнения

1. Используя алгоритм *C*, отсортировать записи, которые образуют последовательность ключей 5, 0, 5, 0, 9, 1, 8, 2, 6, 4, 1, 5, 6, 6, 7, 7. Установить, является ли сортировка по алгоритму *C* устойчивой.

2. С помощью алгоритма *D* отсортировать записи 5 *T*, 0 *C*, 5 *U*, 0 *O*, 9 ., 1 *N*, 8 *S*, 2 *R*, 6 *A*, 4 *A*, 1 *G*, 5 *L*, 6 *T*, 6 *I*, 7 *O* и 7 *N*, в которых цифры обозначают ключи, а символы после цифр – сопутствующую информацию в записях.

11. СОРТИРОВКА ВСТАВКАМИ

Сортировка вставками заключается в следующем: предполагается, что перед рассмотрением записи R_j предыдущие записи R_1, R_2, …, R_{j-1} уже упорядочены, и запись R_j вставляется в подходящее место. На основе этой схемы возможны несколько любопытных вариаций.

Сортировка *простыми вставками* относится к наиболее очевидным сортировочным процедурам. Пусть $1 < j \le N$, а записи R_1, R_2, …, R_{j-1} уже размещены так, что $K_1 \le K_2 \le \ldots \le K_{j-1}$. Будем сравнивать ключ K_j по очереди с ключами K_{j-1}, K_{j-2}, … до тех пор, пока не обнаружим, что запись R_j следует вставить между записями R_i и R_{i+1}. Тогда передвинем записи R_{i+1}, …, R_{j-1} на одно место вверх и поместим новую запись в позицию $i + 1$.

В представленном ниже алгоритме записи R_1, R_2, …, R_N переразмещаются на том же месте памяти. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$.

Алгоритм *S*. (*Сортировка простыми вставками.*)

S1. [Цикл по *j*.] Выполнить шаги *S*2…*S*5 для $j = 2, 3, \ldots, N$ и завершить алгоритм.

***S*2.** [Установка *i*, *K*, *R*.] Выполнить присваивания: $i \leftarrow j - 1$, $K \leftarrow K_j$, $R \leftarrow R_j$. (В последующих шагах делается попытка вставить запись *R* в нужное место, сравнивая ключ *K* с ключами K_i при убывающих значениях *i*.)

***S*3.** [Сравнение *K* с K_i.] Если $K \ge K_i$, то перейти к шагу *S*5. (Найдено искомое место для записи *R*.)

***S*4.** [Перемещение R_i, уменьшение *i*.] Выполнить: $R_{i+1} \leftarrow R_i$, $i \leftarrow i - 1$. Если $i > 0$, то вернуться к шагу *S*3. (Если $i = 0$, то *K* – наименьший ключ из рассмотренных до сих пор ключей, а, значит, запись *R* должна занять первую позицию.)

***S*5.** [Запись *R* на место R_{i+1}.] Выполнить присваивание $R_{i+1} \leftarrow R$. ■

Заметим, что время работы алгоритма равно $(9B + 10N - 3A - 9)\,u$, где *N* – число сортируемых записей; *A* – число случаев, когда на шаге *S*4 значение *i* убывает до нуля; *B* – число перемещений. Среднее время работы алгоритма *S* в предположении, что все ключи различны и расположены в случайном порядке, равно $(2.25N^2 + 7.75N - 3H_N - 6)\,u$.

Обсудим так называемые *бинарные вставки*. Когда при сортировке простыми вставками обрабатывается *j*-я запись, её ключ сравнивается в среднем с $j/2$ ранее отсортированными ключами. Поэтому общее число сравнений равно приблизительно $N^2/4$, а это очень много, даже если *N* умеренно велико.

Метод бинарного поиска позволяет определить место, куда вставлять *j*-й элемент после приблизительно $\log_2 j$ соответствующим образом выбранных сравнений. Например, если вставляется 64-я запись, можно сначала сравнить ключ K_{64} с K_{32}; затем, если он меньше, сравнить его с

ключом K_{16}, если больше – с ключом K_{48} и т.д. В результате место для записи R_{64} будет найдено после всего лишь шести сравнений. Общее число сравнений для N вставляемых элементов равно приблизительно $N \log_2 N$, что существенно лучше, чем $N^2/4$.

К сожалению, бинарные вставки решают задачу только наполовину: после того, как найдено место для записи R_j, всё равно нужно переместить примерно $j/2$ ранее отсортированных записей, чтобы освободить место под запись R_j. Поэтому общее время работы алгоритма остаётся, по существу, пропорциональным N^2.

Рассмотрим механизм, с помощью которого записи при сортировке перемещаются большими скачками, а не маленькими шагами. Его называют *методом Шелла* или *сортировкой с убывающим шагом*.

Пусть имеется шестнадцать записей с ключами:

503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Сначала разделим эти записи на восемь групп по две записи в каждой группе: (R_1, R_9), (R_2, R_{10}), …, (R_8, R_{16}). В результате сортировки каждой из восьми групп записей в отдельности (8-*сортировки*) получим последовательность записей со следующим распределением ключей:

503	87	154	61	612	170	765	275	653	426	512	509	908	677	897	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Затем разделим все записи на четыре группы по четыре записи в каждой: (R_1, R_5, R_9, R_{13}), …, (R_4, R_8, R_{12}, R_{16}), опять отсортируем каждую группу в отдельности (4-*сортировка*) и получим:

503	87	154	61	612	170	512	275	653	426	765	509	908	677	897	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Далее выполняется 2-*сортировка*, в которой сортируются две группы (R_1, R_3, …, R_{15}), (R_2, R_4, …, R_{16}) по восемь записей в каждой группе. Она даст следующую последовательность ключей:

154	61	503	87	512	170	612	275	653	426	765	509	897	677	908	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Процесс завершается 1-*сортировкой*, в которой сортируются все шестнадцать записей и в результате получается последовательность:

61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Заметим, что в каждом из описанных промежуточных процессов сортировки участвуют либо сравнительно короткие файлы, либо сравни-

тельно хорошо упорядоченные файлы. Поэтому здесь можно пользоваться простыми вставками и записи будут довольно быстро достигать своего конечного положения.

Последовательность шагов 8, 4, 2, 1 не следует считать незыблимой, а можно пользоваться *любой* последовательностью h_t, h_{t-1}, …, h_1, в которой последний шаг h_1 равен 1. Например, те же шестнадцать записей можно отсортировать с шагами 7, 5, 3, 1. Отметим, что из всех возможных последовательностей шагов одни последовательности могут оказаться гораздо лучше других.

Алгоритм *D*. (*Сортировка с убывающим шагом.*)

Записи $R_1, R_2, \ldots, R_N$ переразмещаются на том же месте памяти. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$. Для управления процессом сортировки используется вспомогательная последовательность шагов $h_t, h_{t-1}, \ldots, h_1$, где $h_1 = 1$. Выбрав правильную последовательность шагов, можно значительно сократить время сортировки. При $t = 1$ этот алгоритм сводится к алгоритму *S*.

***D*1.** [Цикл по *s*.] Выполнить шаг *D*2 при $s = t, t-1, \ldots, 1$ и закончить алгоритм.

***D*2.** [Цикл по *j*.] Реализовать присваивание $h \leftarrow h_s$ и шаги *D*3…*D*6 при $j = h+1, h+2, \ldots, N$. (Для сортировки элементов, отстоящих друг от друга на h позиций, воспользуемся простыми вставками и в результате получим $K_i \le K_{i+h}$ при $1 \le i \le N-h$. Шаги *D*3…*D*6, по существу, такие же, как соответственно шаги *S*2…*S*5 в алгоритме *S*.)

***D*3.** [Установка *i*, *K*, *R*.] Присвоить $i \leftarrow j-h$, $K \leftarrow K_j$, $R \leftarrow R_j$.

***D*4.** [Сравнение *K* с K_i.] Если $K \ge K_i$, то перейти к шагу *D*6.

***D*5.** [Перемещение R_i.] Выполнить присваивания $R_{i+h} \leftarrow R_i$, $i \leftarrow i-h$. Если после этого будет $i > 0$, то вернуться к шагу *D*4.

***D*6.** [Запись *R* на место R_{i+h}.] Присвоить $R_{i+h} \leftarrow R$. ■

Эксперименты по оценке времени работы алгоритма *D* показали, что для диапазона $100 \le N \le 250\,000$ «наиболее подходящими» формулами оценки являются $1.21\, N^{1.26}$ и $0.39N\,(\ln N)^2 - 2.33N \ln N$. Последовательность же шагов $h_t, h_{t-1}, \ldots, h_1$ разумно выбирать по правилу:

$$h_1 \leftarrow 1,\ h_{s+1} \leftarrow 3h_s + 1 \text{ и остановиться на } h_t, \text{ когда } h_{t+2} \ge N. \quad (47)$$

Это правило, например для $N = 100$, даёт последовательность шагов 13, 4, 1.

Оставим метод Шелла и рассмотрим другие пути усовершенствования метода простых вставок. Одним из самых важных среди общих способов улучшения алгоритма сортировки является способ, который основывается на тщательном анализе структур данных и называется *вставками в список*.

Заметим, что последовательность записей является линейным списком и алгоритм S обрабатывает его, используя последовательное распределение памяти. Именно поэтому для выполнения каждой операции вставки необходимо переместить примерно половину записей. Но, как известно, для вставок идеально подходит связанное распределение памяти, так как при этом требуется изменить всего лишь несколько связей. Поскольку последовательности записей при сортировке простыми вставами просматриваются всегда в одном и том же направлении, то их достаточно представлять односвязными списками.

Приведём алгоритм, предполагающий, что записи $R_1, R_2, \ldots, R_N$ содержат ключи $K_1, K_2, \ldots, K_N$ и поля связи $L_1, L_2, \ldots, L_N$, в которых могут храниться числа от 0 до N. Кроме того, имеется ещё одно поле связи L_0 в некоторой искусственной записи R_0 в начале списка. Алгоритм устанавливает поля связи так, что записи оказываются связанными в возрастающем порядке. Так, если $p(1) \ldots p(N)$ является устойчивой перестановкой такой, что $K_{p(1)} \le \ldots \le K_{p(N)}$, то в результате применения алгоритма получим $L_0 = p(1)$; $L_{p(i)} = p(i + 1)$ при $i = 1, 2, \ldots, N - 1$; $L_{p(N)} = 0$.

Алгоритм *L*. (*Вставки в список.*)

***L*1.** [Цикл по j.] Присвоить $L_0 \leftarrow N$, $L_N \leftarrow 0$. (Поскольку L_0 является «головой» списка, 0 – пустой связью, то список, по существу, циклический.) Выполнить шаги $L2 \ldots L5$ для $j = N - 1, N - 2, \ldots, 1$ и завершить работу алгоритма.

***L*2.** [Установка p, q, K.] Выполнить $p \leftarrow L_0$, $q \leftarrow 0$, $K \leftarrow K_j$. (В последующих шагах запись R_j встанет в нужное место в связанном списке путём сравнения ключа K с предыдущими ключами в возрастающем порядке. Переменные p и q служат указателями на текущее место в списке, причём $p = L_q$, т.е. q всегда на один шаг отстаёт от p.)

***L*3.** [Сравнение K и K_p]. Если $K \le K_p$, то перейти к шагу $L5$. (Найдено искомое положение записи R в списке между записями R_q и R_p.)

***L*4.** [Продвижение p и q]. Присвоить $q \leftarrow p$, $p \leftarrow L_q$. Если $p > 0$, то вернуться к шагу $L3$. (Если $p = 0$, то K – наибольший ключ, обнаруженный до сих пор. Следовательно, запись R должна попасть в конец списка, между записями R_q и R_0.)

***L*5.** [Вставка в список]. Выполнить $L_q \leftarrow j$, $L_j \leftarrow p$. ■

Заметим, что этот алгоритм важен не только как простой метод сортировки, но и потому, что он часто используется в других алгоритмах обработки списков. Время его выполнения равно $(7B + 14N - 3A - 6)u$, где N – число записей, A – число правосторонних максимумов, B – число инверсий в исходной перестановке. По сравнению с алгоритмом S, здесь удаётся сэкономить примерно 22% времени работы, но оно по-прежнему остаётся в среднем пропорциональным N^2.

Упражнения

1. Используя алгоритм *S* сортировки простыми вставками, отсортировать следующую последовательность из двенадцати записей (5 «И»), (7 «,»), (9 «употреблённое»), (12 «не»), (5 «сало»), (15 «меру»), (18 «,»), (21 «может»), (12 «в»), (28 «.»), (27 «вред») и (22 «причинить»), в которых цифры обозначают ключи, а строки символы после цифр – сопутствующую информацию в записях.

2. С помощью алгоритма *D* сортировки с убывающим шагом отсортировать одиннадцать записей: (101 «сам»), (85 «человека»), (25 «робей»), (113 «.»), (48 «врагом»), (32 «перед»), (59 «,»), (99 «он»), (11 «Не»), (60 «лютейший») и (72 «враг»), используя при этом последовательность шагов 5, 3, 1.

3. Определить по правилу (48) разумную последовательность шагов для алгоритма *D* сортировки с убывающим шагом при $N = 20\,000$.

4. Используя графическую иллюстрацию формируемого циклического списка, выполнить алгоритм *L* сортировки вставками в список применительно к последовательности записей: (4 «В»), (47 «известных»), (85 «.»), (35 «жильцов»), (76 «обрящешь»), (21 «без»), (55 «насекомых»), (69 «не»), (12 «доме»).

12. ОБМЕННАЯ СОРТИРОВКА

Семейство алгоритмов обменной сортировки предусматривает систематический обмен местами элементов пар, в которых нарушена упорядоченность до тех пор, пока таких пар не останется.

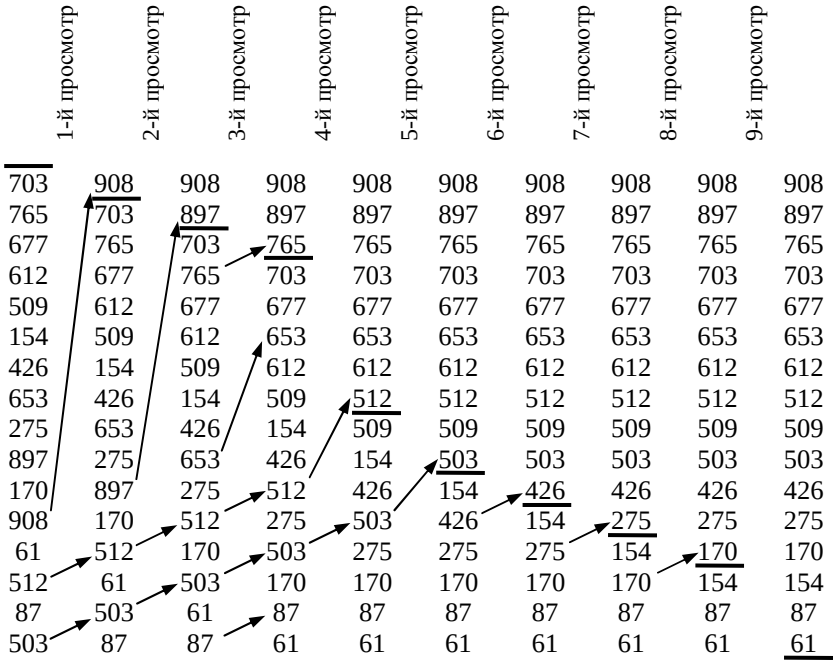
Наиболее очевидным способом обменной сортировки является *метод пузырька*. Он предполагает вначале сравнение ключа K_1 с ключом K_2 и обмен местами записей R_1 и R_2, если эти ключи не упорядочены. Затем подобное проделывается с записями R_2 и R_3, R_3 и R_4, и т.д. Отметим, что при выполнении этой последовательности операций записи с большими ключами будут продвигаться вправо и запись с наибольшим ключом займёт положение R_N. При повторном выполнении этого процесса соответствующие записи попадут в позиции R_{N-1}, R_{N-2} и т.д., так что все записи будут упорядочены. Метод получил своё название, потому что большие элементы, подобно пузырькам, «всплывают» на соответствующую позицию в противоположность методу простых вставок, в котором элементы «погружаются» на соответствующий уровень.

Первый просмотр шестнадцати традиционных ключей, записанных в порядке снизу вверх, повлечёт тринадцать обменов:

K_{16}	703	703	703	703	703	703	703	703	703	703	703	703	703	703	**908**
K_{15}	765	765	765	765	765	765	765	765	765	765	765	765	765	**908**	703
K_{14}	677	677	677	677	677	677	677	677	677	677	677	677	**908**	765	765
K_{13}	612	612	612	612	612	612	612	612	612	612	612	**908**	677	677	677
K_{12}	509	509	509	509	509	509	509	509	509	509	**908**	612	612	612	612
K_{11}	154	154	154	154	154	154	154	154	154	**908**	509	509	509	509	509
K_{10}	426	426	426	426	426	426	426	426	**908**	154	154	154	154	154	154
K_9	653	653	653	653	653	653	**908**	426	426	426	426	426	426	426	426
K_8	275	275	275	275	275	**908**	653	653	653	653	653	653	653	653	653
K_7	897	897	897	897	**908**	275	275	275	275	275	275	275	275	275	275
K_6	170	170	170	**908**	897	897	897	897	897	897	897	897	897	897	897
K_5	**908**	**908**	**908**	170	170	170	170	170	170	170	170	170	170	170	170
K_4	61	61	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**	**512**
K_3	**512**	**512**	61	61	61	61	61	61	61	61	61	61	61	61	61
K_2	87	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**	**503**
K_1	**503**	87	87	87	87	87	87	87	87	87	87	87	87	87	87

В результате первого просмотра записи с выделенными полужирным шрифтом ключами 503 и 512 всплывут на одну позицию, а запись с также выделенным ключом 908 – на одиннадцать позиций.

Рассмотрим процесс сортировки методом пузырька укрупнённо:



После каждого просмотра все записи, начиная с последней, которая участвовала в обмене и выше, занимают свои окончательные позиции, так что их не нужно проверять при последующих просмотрах. Под такой группой записей на рисунке поставлена черта. Последним просмотром следует считать просмотр, при котором не было произведено ни одного обмена.

Алгоритм *B*. (*Метод пузырька.*)

Записи $R_1, \ldots, R_N$ переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$.

***B*1.** [Начальная установка *BOUND*.] Присвоить $BOUND \leftarrow N$. (*BOUND* – индекс самого верхнего элемента, о котором еще не известно, занял ли он уже свою окончательную позицию. Таким образом, к этому моменту еще ничего не известно.)

***B*2.** [Цикл по *j*.] Присвоить $t \leftarrow 0$. Выполнить шаг *B*3 для $j = 1, 2, \ldots, BOUND - 1$ и перейти к шагу *B*4.

***B*3.** [Сравнение/обмен R_j и R_{j+1}.] Если $K_j > K_{j+1}$, то поменять местами R_j и R_{j+1}, а также $t \leftarrow j$.

***B*4.** [Были обмены?] Если $t = 0$, то завершить работу алгоритма; в противном случае $BOUND \leftarrow t$ и вернуться к шагу *B*2. ■

Минимальное, среднее и максимальное время работы этого алгоритма оценивается последовательностью $(\min\ (8N + 1)u$, ave $(5.75N^2 + O(N \ln N))u$, max $(7.5N^2 + 0.5N + 1)u)$.

Параллельной сортировкой Бэтчера называется обменная сортировка, в которой сравниваются подбираемые пары несоседних ключей. Последовательность сравнений, предназначенную для отыскания возможных обменов, открыл К.Е. Бэтчер в 1964 г. В его далеко не очевидном методе, требующем весьма сложного доказательства, выполняется относительно мало сравнений.

Схема сортировки Бэтчера несколько напоминает сортировку Шелла, но сравнения выполняются по-другому так, что, по существу, происходит слияние пар отсортированных последовательностей. Поэтому алгоритм Бэтчера называют ещё *обменной сортировкой со слиянием*.

Алгоритм *M*. (*Обменная сортировка со слиянием.*)

Записи $R_1, \ldots, R_N$ переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$. Предполагается, что $N \ge 2$.

***M*1.** [Начальная установка *p*.] Присвоить $p \leftarrow 2^{t-1}$, где $t = \lceil \log_2 N \rceil$ – наименьшее целое число, такое, что $2^t \ge N$. (Шаги *M*2, …, *M*5 будут выполняться для $p = 2^{t-1}, 2^{t-2}, \ldots, 1$.)

***M*2.** [Начальная установка *q, r, d*.] Выполнить присваивания $q \leftarrow 2^{t-1}$, $r \leftarrow 0$, $d \leftarrow p$.

***M*3.** [Цикл по i.] Для всех i, таких, что $0 \le i < N - d$ и $i \wedge p = r$, выполнить шаг *M*4. Затем перейти к шагу *M*5. (Здесь через $i \wedge p$ обозначена операция «логическое и» над двоичными представлениями чисел i и p. Так, например, $13 \wedge 21 = (1101)_2 \wedge (10101)_2 = (00101)_2 = 5$. К этому моменту d – нечётное кратное p (т.е. частное от деления d на p нечётно); а p – степень двойки, так что $i \wedge p \ne (i + d) \wedge p$; отсюда следует, что действия шага *M*4 можно выполнять при всех нужных значениях i в любом порядке или далее одновременно.)

***M*4.** [Сравнение/обмен R_{i+1} и R_{i+d+1}.] Если $K_{i+1} > K_{i+d+1}$, то поменять местами записи R_{i+1} и R_{i+d+1}.

***M*5.** [Цикл по q.] Если $q \ne p$, то выполнить присваивания $d \leftarrow q - p$, $q \leftarrow q/2$, $r \leftarrow p$ и вернуться к шагу *M*3.

***M*6.** [Цикл по p.] (К этому моменту перестановка K_1, K_2, …, K_N будет p-упорядочена.) Присвоить $p \leftarrow \lfloor p/2 \rfloor$. Если $p > 0$, то вернуться к шагу *M*2. ■

Заметим, что в методе Бэтчера последовательность сравнений предопределена, т.е. каждый раз сравниваются одни и те же пары ключей, независимо от информации о файле, которую можно получить из предыдущих сравнений.

Обратимся к другой стратегии, в которой результат каждого сравнения определяет то, какие ключи сравнивать следующими. Такая стратегия не годится для параллельных вычислений, но может оказаться плодотворной для вычислительных машин, работающих последовательно.

Рассмотрим схему сравнений/обменов этой стратегии. Пусть имеются указатели i и j, причём вначале $i = 1$, а $j = N$. Сравним ключи K_i, K_j и если обмен не требуется, то уменьшим j на 1 и повторим этот процесс. После первого обмена увеличим i на 1 и будем продолжать сравнения, увеличивая i, пока не произойдёт ещё один обмен. Тогда опять уменьшим j и т.д. Таким образом, будем «сжигать свечку с обоих концов» до тех пор, пока i не станет равно j. Посмотрим, например, что произойдёт с нашим файлом из шестнадцати чисел:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	**703**
уменьшение j и 1-й обмен															
154	87	512	61	908	170	897	275	653	426	**503**	509	612	677	765	703
увеличение i и 2-й обмен															
154	87	**503**	61	908	170	897	275	653	426	**512**	509	612	677	765	703
уменьшение j и 3-й обмен															
154	87	**426**	61	908	170	897	275	653	**503**	512	509	612	677	765	703

увеличение i и 4-й обмен															
154	87	426	61	**503**	170	897	275	653	**908**	512	509	612	677	765	703
уменьшение j и 5-й обмен															
154	87	426	61	**275**	170	897	**503**	653	908	512	509	612	677	765	703
увеличение i и 6-й обмен															
154	87	426	61	275	170	**503**	**897**	653	908	512	509	612	677	765	703

Здесь для того, чтобы выявить значения указателей i и j непосредственно после обменов, ключи K_i и K_j выделены полужирным шрифтом. Заметим, что в каждом сравнении этого примера участвовал ключ 503. В общем случае в каждом сравнении будут участвовать исходный ключ K_1, потому что он будет продолжать обмениваться местами с другими ключами каждый раз, когда мы переключаем направление. К моменту, когда i сравняется с j, исходная запись R_1 займёт свою окончательную позицию, потому что слева от неё не будет больших ключей, а справа – меньших. В результате исходный файл окажется разделённым таким образом, что первоначальная задача сведётся к двум более простым: сортировке файла R_1, …, R_{i-1} и (независимо) сортировке файла R_{i+1}, …, R_N. К каждому из этих подфайлов можно применить тот же самый метод.

Ниже показано, как выбранный нами для примеров файл полностью сортируется при помощи этого метода за одиннадцать стадий.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(l, r)	Стек
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703	(1, 16)	–
154	87	426	61	275	170	503	897	653	908	512	509	612	677	765	703	(1, 6)	(8, 16)
61	87	154	426	275	170	503	897	653	908	512	509	612	677	765	703	(1, 2)	(4, 6), (8, 16)
61	87	154	426	275	170	503	897	653	908	512	509	612	677	765	703	(4, 6)	(8, 16)
61	87	154	170	275	426	503	897	653	908	512	509	612	677	765	703	(4, 5)	(8, 16)
61	87	154	170	275	426	503	897	653	908	512	509	612	677	765	703	(8, 16)	–
61	87	154	170	275	426	503	703	653	765	512	509	612	677	897	908	(8, 14)	–
61	87	154	170	275	426	503	677	653	612	512	509	703	765	897	908	(8, 12)	–
61	87	154	170	275	426	503	509	653	612	512	677	703	765	897	908	(8, 11)	–
61	87	154	170	275	426	503	509	653	612	512	677	703	765	897	908	(9, 11)	–
61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908	(9, 10)	–
61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908	–	–

Здесь в скобки заключены подфайлы, которые ещё предстоит отсортировать. В вычислительной машине эти подфайлы можно представлять парой (l, r), содержащей указатели границ рассматриваемого в данный момент файла, а также стеком дополнительных пар (l_k, r_k), которые ещё предстоит рассмотреть. Каждый раз, когда файл подразделяется, в стек

помещаем больший из подфайлов и начинаем обрабатываться оставшийся, и так до тех пор, пока не придём к тривиально коротким файлам. Такая процедура гарантирует, что в стеке никогда не будет находиться более чем примерно $\log_2 N$ элементов.

Описанную процедуру сортировки можно назвать *обменной сортировкой с разделением*. Ч.Э.Р. Хоар опубликовал её в 1962 г. под названием «*quicksort*» (*быстрая сортировка*), и это соответствует действительности, так как весь процесс сортировки нашего файла из шестнадцати записей требует всего сорок восемь сравнений (меньше любого другого уже встречавшегося метода за исключением бинарных вставок, требующих сорок семь сравнений).

Заметим, что процедура быстрой сортировки посредством разделений пригодна в основном для больших значений N. Поэтому короткие подфайлы желательно сортировать особым образом, как это делается в следующем алгоритме.

Алгоритм *Q*. (*Обменная сортировка с разделением.*)

Записи $R_1, \ldots, R_N$ переразмещаются на том же месте памяти. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$. Нужен вспомогательный стек для хранения не более чем $\log_2 N$ элементов. Этот алгоритм соответствует процедуре «быстрой сортировки», приведённой выше, с небольшими изменениями в целях повышения эффективности:

а) предполагается наличие искусственных ключей $K_0 = -\infty$ и $K_{N+1} = +\infty$ таких, что

$$K_0 \le K_i \le K_{N+1} \quad \text{для} \quad 1 \le i \le N; \tag{48}$$

б) подфайлы, состоящие из M и менее элементов, сортируются простыми вставками. Следовательно, $M \ge 1$ – выбираемый параметр сортировки;

в) на некоторых стадиях сортировки делается одно или два дополнительных сравнения (допускается перекрытие указателей i, j), чтобы основные циклы сравнения выполнялись настолько быстро, насколько это возможно;

г) записи с одинаковыми ключами меняются местами, хотя это не является строго необходимым.

***Q*1.** [Начальная установка.] Очистить стек и выполнить присваивания $l \leftarrow 1$, $r \leftarrow N$.

***Q*2.** [Начало новой стадии.] (Нам хотелось бы отсортировать файл $R_l, \ldots, R_r$.) Если $r - l < M$, то перейти к шагу *Q*8; в противном случае $i \leftarrow l$, $j \leftarrow r$, $K \leftarrow K_l$, $R \leftarrow R_l$.

***Q*3.** [Сравнение K и K_j.] Если $K < K_j$, то j уменьшить на 1 и повторить этот шаг.

Q4. [Запись R на место R_i.] (К этому моменту K_i – ключ, не меньший чем K и находящийся не на своём месте, кроме того $K \geq K_j$.) Если $j \leq i$, то $R_i \leftarrow R$ и перейти к шагу $Q7$; в противном случае $R_i \leftarrow R_j$ и увеличить i на 1.

Q5. [Сравнение K_i и K.] Если $K_i < K$, то i увеличить на 1 и повторить этот шаг.

Q6. [Запись R на место R_j.] (К этому моменту K_j – ключ, не больший чем K и находящийся не на своём месте, кроме того $K \leq K_i$.) Если $j \leq i$, то $R_j \leftarrow R$ и $i \leftarrow j$; в противном случае $R_j \leftarrow R_i$, значение j уменьшить на 1 и вернуться к шагу $Q3$.

Q7. [Включение в стек.] (Теперь подфайл R_l, …, R_i, …, R_r разделён так, что $K_k \leq K_i$ при $l \leq k \leq i$ и $K_i \leq K_k$ при $i \leq k \leq r$.) Если $r - i \geq i - l$, то пару $(i + 1, r)$ включить в стек и $r \leftarrow i - 1$; в противном случае пару $(l, i - 1)$ включить в стек и $l \leftarrow i + 1$. (Каждый элемент стека вида (a, b) является заявкой на сортировку подфайла R_a, …, R_b в одной из последующих стадий.) Вернуться к шагу $Q2$.

Q8. [Сортировка простыми вставками.] Для $j = l + 1, l + 2, \ldots, r$ выполнять следующие операции: сначала $K \leftarrow K_j$, $R \leftarrow R_j$, $i \leftarrow j - 1$; затем $R_{i+1} \leftarrow R_i$ и $i \leftarrow i - 1$ нуль или более раз, пока выполняется условие $K_i > K$; наконец $R_{i+1} \leftarrow R$. (Это, по существу, алгоритм S, применённый к подфайлу из M или менее элементов.)

Q9. [Исключение из стека.] Если стек пуст, то конец алгоритма; в противном случае взять верхний элемент стека (l', r'), присвоить $l \leftarrow l'$, $r \leftarrow r'$ и вернуться к шагу $Q2$. ■

Отметим, что «наилучшее» значение M (для гипотетической машины MIX) равно девяти. Если $M = 9$, то при больших значениях N среднее время работы алгоритма примерно равно $(12.67(N + 1) \ln N - 1.92 N - 14.59)\, u$.

Следовательно, алгоритм Q работает в среднем довольно быстро, кроме того, он требует очень мало памяти. Но следует отметить, что наихудшим случаем для алгоритма Q является тот, когда исходный файл уже упорядочен. При этом каждая операция «разделения» становится почти бесполезной, поскольку каждый выделяемый подфайл состоит из одного элемента. И самый, казалось бы, простой для сортировки файл будет сортироваться по алгоритму Q с затратами времени, пропорциональными N^2.

Таким образом, в отличие от рассмотренных ранее алгоритмов сортировки, алгоритм Q предпочитает неупорядоченные файлы.

Упражнения

1. Методом пузырька по алгоритму B отсортируйте последовательность из тринадцати записей: (47 «придерживай»), (79 «за»), (94 «козырёк»), (41 «высокие»), (30 «людей»), (10 «на»), (12 «высоких»), (77 «свой»), (36 «и»), (96 «.»), (7 «Взирая»), (44 «предметы»), (69 «картуз»).

2. Используя алгоритм *M* обменной сортировки со слиянием, отсортируйте последовательность из шестнадцати записей: (503 «коли»), (87 «скажут»), (512 «сам»), (61 «Что»), (908 «?»), (170 «тебе»), (897 «можешь»), (275 «другие»), (653 «себе»), (426 «,»), (154 «о»), (509 «ты»), (612 «о»), (677 «ничего»), (765 «не»), (703 «сказать»).

3. Выполните с параметром *M* = 5 алгоритм *Q* обменной сортировки с разделением для последовательности из четырнадцати записей: (39 «,»), (16 «у»), (80 «отдохнуть), (59 «;»), (16 «тебя»), (40 «заткни»), (22 «есть»), (13 «Если»), (86 «фонтану»), (63 «дай»), (81 «и»), (93 «.»), (23 «фонтан»), (44 «его»).

13. СОРТИРОВКА ПОСРЕДСТВОМ ВЫБОРА

Простейшая сортировка посредством выбора сводится к следующему:

а) выбрать в области ввода запись с наименьшим ключом; скопировать эту запись в область вывода и заменить ключ этой записи в области ввода значением «∞», которое по предположению больше любого реального ключа;

б) повторить шаг (а). На этот раз в область вывода будет скопирована запись с ключом, наименьшим из невыбранных ещё записей, так как ранее наименьший ключ был заменён на «∞»;

в) повторять шаг (а) до тех пор, пока в область вывода не будут скопированы все *N* записей.

Для наших традиционных шестнадцати ключей после первого выполнения шага (а) содержимое областей ввода и вывода будет следующим:

Область ввода

503	87	512	∞	908	170	897	275	653	426	154	509	612	677	765	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Область вывода

61															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

После второго выполнения шага (а):

Область ввода

503	∞	512	∞	908	170	897	275	653	426	154	509	612	677	765	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Область вывода

61	87														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Наконец, после шестнадцатого выполнения:

Область ввода

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Область вывода

61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Описанный метод требует $(N - 1)$ сравнений каждый раз, когда выбирается очередная запись, и требует отдельной области вывода в памяти. Ликвидировать отмеченные недостатки можно следующим образом. Не использовать ключ «∞» и выбранную запись перемещать в соответствующую позицию, а запись, которая занимала эту позицию, переносить на место выбранной. Тогда упомянутую выбранную запись не нужно рассматривать вновь при последующих выборах.

Алгоритм *S*. (*Сортировка посредством простого выбора*.) Записи $R_1, \ldots, R_N$ переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$. Сортировка основана на описанном выше методе, но для удобства сначала выбирается наибольший элемент, затем – наибольший из оставшихся элементов и т.д.

***S*1.** [Цикл по j.] Выполнить шаги *S*2 и *S*3 при $j = N, N - 1, \ldots, 2$.

***S*2.** [Поиск max $\{K_1, \ldots, K_j\}$.] Найти наибольший из ключей $K_j, K_{j-1}, \ldots, K_1$. Пусть это будет ключ K_i, где i выбирается как можно большим.

***S*3.** [Обмен.] Поменять местами записи R_i и R_j. (Теперь записи $R_j, \ldots, R_N$ занимают свои окончательные позиции.) ■

Действие этого алгоритма на последовательность из шестнадцати ключей будет следующим:

503	87	512	61	**908**	170	897	275	653	426	154	509	612	677	765	**703**
503	87	512	61	703	170	**897**	275	653	426	154	509	612	677	**765**	908
503	87	512	61	703	170	**765**	275	653	426	154	509	612	**677**	897	908
503	87	512	61	**703**	170	677	275	653	426	154	509	**612**	765	897	908
503	87	512	61	612	170	**677**	275	653	426	154	**509**	703	765	897	908

. . . . . .

61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Здесь полужирным шрифтом отмечены ключи обменивающихся местами записей.

Отметим, что среднее время работы алгоритма *S* равно $(2.5N^2 + 3(N + 1)H_N + 3.5N - 11)u$, т.е. он лишь немного медленнее простых вставок.

С целью усовершенствования алгоритма простого выбора проанализируем поиск максимума на шаге *S*2. Для определения максимума из N элементов, основанном на сравнении пар элементов, действительно

необходимо выполнить по крайней мере (N – 1) сравнений. Но, заметим, что это справедливо только к *первому* выбору. При *последующих* выборах можно использовать извлечённую ранее информацию.

Рассмотрим те самые шестнадцать ключей, которые разобьём на четыре группы по четыре числа и найдём максимальный элемент в каждой группе, выполнив при этом 12 сравнений. Наибольший элемент 908 из максимальных элементов в группах 512, 908, 653, 765 найдём путём выполнения трёх сравнений:

908															
512				908				653				765			
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703

Следовательно, на определение наибольшего элемента 908 во всём файле затрачивается 15 сравнений.

Для получения второго по величине элемента 897 в файле достаточно определить максимальный элемент из оставшихся трёх элементов 170, 897, 275 во второй группе, выполнив при этом два сравнения, и найти наибольший из элементов 512, 897, 653, 765 за три сравнения:

897															
512				897				653				765			
503	87	512	61		170	897	275	653	426	154	509	612	677	765	703

Таким образом, на определение второго по величине элемента 897 затрачивается всего пять сравнений.

Для определения третьего по величине элемента достаточно найти максимальный из элементов 170, 275 и наибольший из элементов 512, 275, 653, 765. Третий по величине элемент 765 определяется за четыре сравнения и т.д.

В общем случае, если N является точным квадратом некоторого натурального числа, то файл можно разделить на $\sqrt{N}$ групп по $\sqrt{N}$ элементов в каждой группе. Тогда любой выбор, кроме первого, потребует не более чем ($\sqrt{N}$ – 1) сравнений внутри группы ранее выбранного элемента плюс ($\sqrt{N}$ – 1) сравнений среди «лидеров» групп. Этот метод получил название *квадратичного выбора*, общее время работы которого имеет порядок $N\sqrt{N}$, что существенно лучше, чем N^2.

Метод квадратичного выбора можно обобщить и получить в результате метод выбора третьей, четвёртой, …, n-й степени. Например, в методе кубического выбора файл разбивается на $\sqrt[3]{N}$ больших групп, каждая из которых содержит по $\sqrt[3]{N}$ малых групп по $\sqrt[3]{N}$ записей. Время работы кубического выбора пропорционально $N\sqrt[3]{N}$.

Если N является точным значением 2^n , то можно организовать выбор n-й степени. При этом файл разбивают на две группы по 2^{n-1} элементов, каждую группу – на две подгруппы по 2^{n-2} элементов и т.д. Таким образом, выбор n-й степени основан на структуре бинарного дерева, его называют также *выбором из дерева*. Время его работы пропорционально значению $N \log N$.

На рисунках 28 – 30 наши 16 чисел сортируются посредством выбора из дерева.

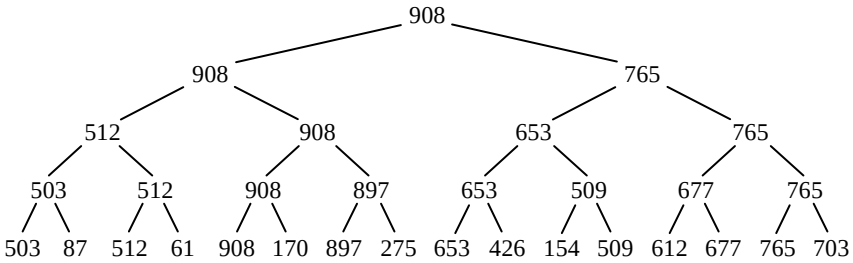


Рис. 28. Исходная конфигурация дерева

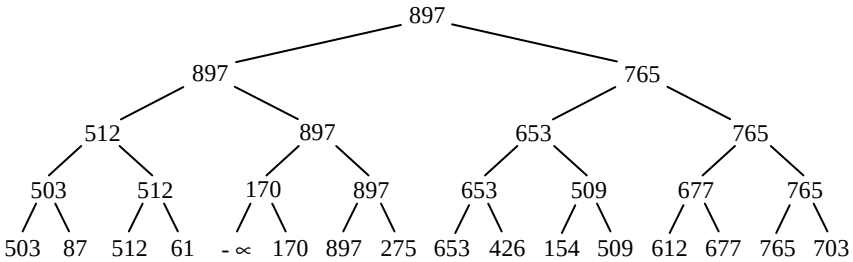


Рис. 29. Дерево после вывода элемента 908

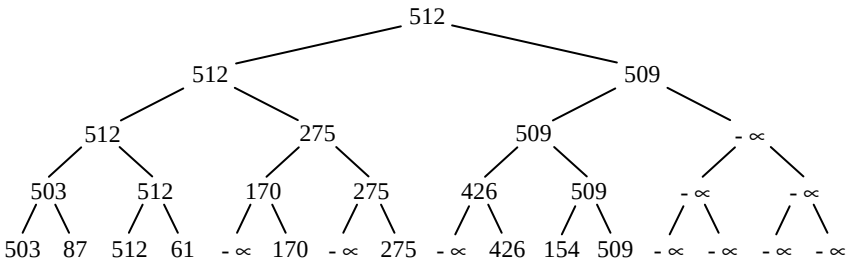


Рис. 30. Дерево после вывода элементов 908, 897, 765, 703, 677, 653 и 612

Заметим, что для того, чтобы знать, куда вставлять следующий элемент «–∞», необходимо помнить, откуда пришёл ключ, находящийся в корне. Поэтому узлы разветвления должны содержать указатели, описывающие позицию ключа, а не сам ключ. Отсюда следует, что необходима память для N исходных записей, $(N-1)$ указателей и N выводимых записей.

Теперь рассмотрим одну из модификаций выбора из дерева, которая устраняет необходимость указателей. Когда некоторый ключ поднимается вверх с нижнего уровня, то на нижнем уровне его сразу можно заменить на «–∞». Если же ключ перемещается вверх с одного разветвления на другое, то этот ключ можно заменить наибольшим из двух ключей, который в конце концов всё равно должен подняться на его прежнее место. Выполнив эту операцию (*корпоративной системы выдвижений*) столько раз, сколько возможно, от исходной конфигурации дерева, представленной на рис. 28, прийдём к дереву, изображённому на рис. 31.

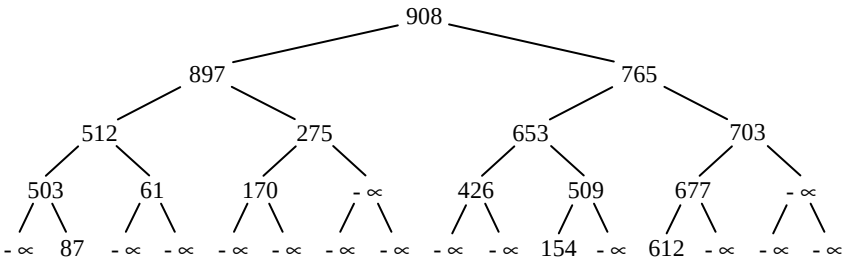


Рис. 31. Дерево, в котором каждый ключ поднялся на свой уровень в иерархии

Имея такое дерево, можно продолжать сортировку не *восходящим* (см. рис. 28 – 30), а *нисходящим методом*: выводится элемент, находящийся в корне, перемещается вверх наибольший из его потомков, перемещается вверх наибольший из потомков последнего и так далее до первого перемещения вверх «–∞». Этот метод имеет важное достоинство – он позволяет избежать лишних сравнений «–∞» с «–∞».

Отметим, что до сих пор говорилось только о полных бинарных деревьях, содержащих $N = 2^n$ узлов. В действительности можно работать с произвольным значением N, поскольку для любого N нетрудно построить полное бинарное дерево с N концевыми узлами.

Полные бинарные деревья удобно хранить в последовательных ячейках памяти так, как показано на рис. 32 для дерева с шестнадцатью узлами. Числа в узлах этого дерева обозначают не значения ключей, а номера ячеек памяти, в которых хранятся ключи.

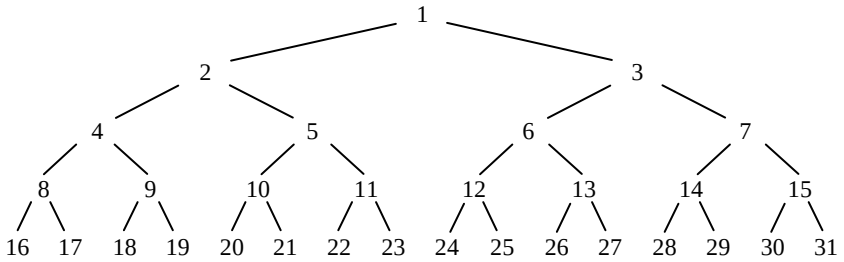


Рис. 32. Последовательное распределение памяти для полного бинарного дерева с 16 узлами

Заметим, что отцом узла номер k является узел с номером $\lfloor k/2 \rfloor$, а его потомками – узлы с номерами $2k$ и $2k + 1$.

Наконец рассмотрим нисходящий метод сортировки, который обходится совсем без ключей «$-\infty$», т.е. вся существенная информация, имеющаяся на рис. 31, располагается в ячейках 1 – 16 полного бинарного дерева без всяких бесполезных ячеек, содержащих «$-\infty$». Этот метод, называемый *пирамидальной сортировкой*, сортирует N записей на том же месте в памяти без вспомогательной области вывода.

Файл ключей $K_1, K_2, \ldots, K_N$ будем называть *пирамидой*, если

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad \text{при } 1 \leq \lfloor j/2 \rfloor < j \leq N. \tag{49}$$

В этом случае $K_1 \geq K_2$, $K_1 \geq K_3$, $K_2 \geq K_4$ и т.д. (см. рис. 31), т.е. наибольший ключ оказывается на *вершине* пирамиды:

$$K_1 = \max (K_1, K_2, \ldots, K_N). \tag{50}$$

Рассмотрим задачу преобразования произвольного исходного файла в пирамиду. Пусть нам удалось расположить файл таким образом, что:

$$K_{\lfloor j/2 \rfloor} \geq K_j \quad \text{при } l < \lfloor j/2 \rfloor < j \leq N, \tag{51}$$

где $l \geq 1$. Заметим, что в исходном файле это условие выполняется «автоматически» для $l = \lfloor N/2 \rfloor$, поскольку ни один индекс j не удовлетворяет условию $\lfloor N/2 \rfloor < \lfloor j/2 \rfloor < j \leq N$. Нетрудно понять, как, изменяя лишь поддерево с корнем в узле l, преобразовать файл, чтобы распространить неравенства (51) и на случай, когда $\lfloor j/2 \rfloor = l$. Следовательно, можно уменьшать l на единицу до тех пор, пока в конце концов не будет достигнуто условие (49). Эти идеи приводят к изящному алгоритму, который заслуживает пристального изучения.

Алгоритм *H*. (*Пирамидальная сортировка*.) Записи $R_1, \ldots, R_N$ переразмещаются на том же месте; после завершения сортировки их ключи будут упорядочены: $K_1 \le \ldots \le K_N$. Сначала файл перестраивается в пирамиду, после чего вершина пирамиды многократно исключается и записывается на своё окончательное место. Предполагается, что $N \ge 2$.

***H*1.** [Начальная установка.] $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$.

***H*2.** [Уменьшить l или r.] (Если на данный момент $l > 1$, то происходит преобразование исходного файла в пирамиду. Если же $l = 1$, то это значит, что ключи $K_1, K_2, \ldots, K_r$ уже образуют пирамиду.) Если $l > 1$, то $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$; в противном случае $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$, $r \leftarrow r - 1$. Если в результате оказалось, что $r = 1$, то $R_1 \leftarrow R$ и завершить работу алгоритма.

***H*3.** [Приготовиться к «протаскиванию».] $j \leftarrow l$. (К этому моменту

$$K_{\lfloor j/2 \rfloor} \ge K_j \quad \text{при } l < \lfloor j/2 \rfloor < j \le r, \tag{52}$$

а записи R_k, $r < k \le N$, занимают свои окончательные места. Шаги *H*3…*H*8 называются алгоритмом «протаскивания»; их действие эквивалентно присваиванию $R_l \leftarrow R$ с последующим перемещением записей $R_l, \ldots, R_r$ таким образом, чтобы условие (52) выполнялось и при $l = \lfloor k/2 \rfloor$.)

***H*4.** [Продвинуться вниз.] $i \leftarrow j$, $j \leftarrow 2j$. (В последующих шагах $i = \lfloor j/2 \rfloor$.) Если $j < r$, то перейти к шагу *H*5; если $j = r$, то перейти к шагу *H*6; если же $j > r$, то перейти к шагу *H*8.

***H*5.** [Найти большего сына.] Если $K_j < K_{j+1}$; то $j \leftarrow j + 1$.

***H*6.** [Больше, чем K?] Если $K \ge K_j$, то перейти к *H*8.

***H*7.** [Поднять его вверх.] Присвоить $R_i \leftarrow R_j$ и вернуться к шагу *H*4.

***H*8.** [Занести R.] Присвоить $R_i \leftarrow R$. (На этом алгоритм «протаскивания», начатый на шаге *H*3, заканчивается.) Вернуться к шагу *H*2. ■

Упражнения

1. Отсортируйте методом кубического выбора последовательность из двадцати семи ключей: 48, 7, 22, 10, 51, 69, 5, 62, 26, 16, 60, 78, 97, 14, 66, 9, 8, 83, 96, 20, 56, 41, 98, 63, 28, 66, 9 и установите количество сравнений пар ключей, которые будут при этом выполнены.

2. Для традиционной последовательности шестнадцати ключей:

503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

а) постройте полное бинарное дерево с 16 ячейками (без всяких бесполезных ячеек, содержащих «$-\infty$») и расположите ключи в соответствующих ячейках;

б) выполните по алгоритму *H* пирамидальную сортировку этих ключей, прослеживая изменения, происходящие в построенном бинарном дереве.

14. СОРТИРОВКА СЛИЯНИЕМ

Слияние означает объединение двух или более упорядоченных подфайлов в один упорядоченный файл. Можно, например, слить два подфайла – 503 703 765 и 87 512 677, получив 87 503 512 677 703 765. Простой способ сделать это – сравнить наименьшие элементы подфайлов, вывести в файл наименьший из них и повторить эту процедуру.

Начав с

$$\begin{cases} 503 & 703 & 765 \\ 87 & 512 & 677 \end{cases},$$

получим

$$87 \begin{cases} 503 & 703 & 765 \\ 512 & 677 \end{cases}.$$

Затем

$$87 \quad 503 \begin{cases} 703 & 765 \\ 512 & 677 \end{cases}$$

и т.д. Заметим, что при этом необходимо позаботиться о действиях на случай, когда исчерпается один из подфайлов.

Весь процесс слияния подробно описан в следующем алгоритме.

Алгоритм *M*. (*Двухпутевое слияние.*)

Этот алгоритм осуществляет слияние двух упорядоченных подфайлов $x_1 \le x_2 \le \ldots \le x_m$ и $y_1 \le y_2 \le \ldots \le y_n$ в один файл $z_1 \le z_2 \le \ldots \le z_{m+n}$.

***M*1.** [Начальная установка.] Установить $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$.

***M*2.** [Найти наименьший элемент.] Если $x_i \le y_j$, то перейти к шагу *M*3; в противном случае – к шагу *M*5.

***M*3.** [Вывести x_i.] Установить $z_k \leftarrow x_i$, $k \leftarrow k + 1$, $i \leftarrow i + 1$. Если $i \le m$, то вернуться к шагу *M*2.

***M*4.** [Вывести $y_j, \ldots, y_n$.] Установить $(z_k, \ldots, z_{m+n}) \leftarrow (y_j, \ldots, y_n)$ и завершить работу алгоритма.

***M*5.** [Вывести y_j.] Установить $z_k \leftarrow y_j$, $k \leftarrow k + 1$, $j \leftarrow j + 1$. Если $j \le n$, то вернуться к шагу *M*2.

***M*6.** [Вывести $x_i, \ldots, x_m$.] Установить $(z_k, \ldots, z_{m+n}) \leftarrow (x_i, \ldots, x_m)$ и завершить работу алгоритма. ■

Эта простая процедура, по существу, «наилучший из возможных» способов слияния на традиционной ЭВМ при условии, что $m \approx n$. Но, если m гораздо меньше n, можно разработать более эффективные алгоритмы сортировки, хотя в общем случае они довольно сложны. Алгоритм M без особой потери эффективности можно немного упростить, добавив в конец исходных файлов искусственных «стражей» $x_{m+1} = y_{n+1} = \infty$, и останавливаться перед выводом ∞.

Общий объём работы, выполняемой алгоритмом M, по существу, пропорционален $m + n$.

Понятно, что слияние является более простой задачей по сравнению с сортировкой. Однако задачу сортировки можно свести к слияниям, сливая все более длинные подфайлы до тех пор, пока не будет отсортирован весь файл. Такой подход можно рассматривать как развитие идеи сортировки вставками, так как вставка нового элемента в упорядоченный файл является частным случаем слияния при $n = 1$.

С исторической точки зрения метод слияний это один из самых первых методов, предназначенных для решения задачи сортировки на ЭВМ; он был предложен Джоном фон Нейманом ещё в 1945 г.

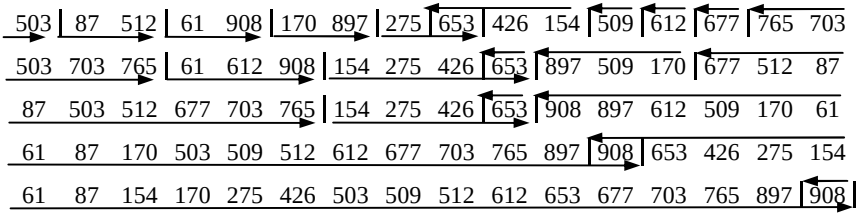
Слияния более подробно будут рассмотрены в связи с алгоритмами внешней сортировки, а здесь сосредоточим своё внимание на сортировке в быстрой памяти с произвольным доступом.

Таблица 1 иллюстрирует сортировку слиянием, когда «свечка сжигается с обоих концов», подобно тем процедурам просмотра элементов файла, которые применялись при быстрой сортировке, поразрядной обменной сортировке и т.д. В ней исходный файл анализируется слева и справа в направлении к его середине. Процесс слияния имеет смысл описать на примере перехода от второй строки к третьей. Во второй строке слева имеет место возрастающий отрезок 503 703 765, а справа, если читать справа налево, – возрастающий отрезок 87 512 677. Слияние этих двух последовательностей даёт подфайл 87 503 512 677 703 765, который помещается *слева* в третью строку. Затем возрастающая последовательность ключей 61 612 908 второй строки сливается с последовательностью ключей 170 509 897 этой же строки, и результат (61 170 509 612 897 908) записывается *справа* в третью строку. Наконец, последовательность 154 275 426 653 сливается с ключом 653 (при этом перекрытие ключей не приводит к вредным последствиям), и результат записывается *слева*. Аналогичным образом вторая строка получилась из первой, четвёртая – из третьей и пятая из четвёртой.

Вертикальными линиями в табл. 1 отмечены границы между отрезками. Это так называемые *ступеньки вниз*, где меньший элемент следует за большим. В середине файла обычно возникает двусмысленная ситуация, когда при движении с обоих концов прочитывается один и тот же ключ, однако в представленном ниже алгоритме это не приводит к ос-

ложнениям. Этот метод по традиции называется *естественным слиянием*, так как он использует отрезки, которые естественно образуются в исходном файле.

Таблица 1



Алгоритм *N*. (*Сортировка естественным двухпутевым слиянием.*) При сортировке записей $R_1, \ldots, R_N$ используются две области памяти, каждая из которых может содержать N записей. Для удобства обозначим записи, находящиеся во второй области, через $R_{N+1}, \ldots, R_{2N}$, хотя в действительности запись R_{N+1} может и не примыкать непосредственно к R_N. Начальное содержимое записей $R_{N+1}, \ldots, R_{2N}$ не имеет значения. После завершения сортировки ключи будут упорядочены: $K_1 \le \ldots \le K_N$.

***N*1.** [Начальная установка.] Установить $s \leftarrow 0$. (При $s = 0$ записи из области $(R_1, \ldots, R_N)$ пересылаются в область $(R_{N+1}, \ldots, R_{2N})$; при $s = 1$ области по отношению к пересылкам поменяются ролями.)

***N*2.** [Подготовка к просмотру.] Если $s = 0$, то присвоить $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N + 1$, $l \leftarrow 2N$. Если $s = 1$, то присвоить $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 1$, $l \leftarrow N$. (Переменные i, j, k, l указывают текущие позиции во входных файлах, откуда идёт чтение, и в выходных файлах, куда идёт запись.) Присвоить $d \leftarrow 1$, $f \leftarrow 1$. (Переменная d определяет текущее направление вывода, f устанавливается равной 0, если необходимы дальнейшие просмотры.)

***N*3.** [Сравнение K_i и K_j.] Если $K_i > K_j$, то перейти к шагу $N8$. Если $i = j$, то присвоить $R_k \leftarrow R_i$ и перейти к шагу $N13$.

***N*4.** [Пересылка R_i.] (Шаги $N4 \ldots N7$ аналогичны шагам $M3 \ldots M4$ алгоритма M.) Установить $R_k \leftarrow R_i$, $k \leftarrow k + d$.

***N*5.** [Ступенька вниз?] Увеличить i на 1. Затем, если $K_{i-1} \le K_i$, то вернуться к шагу $N3$.

***N*6.** [Пересылка R_j.] Присвоить $R_k \leftarrow R_j$, $k \leftarrow k + d$.

***N*7.** [Ступенька вниз?] Уменьшить j на 1. Затем, если $K_{j+1} \le K_j$, то вернуться к шагу $N6$; в противном случае перейти к шагу $N12$.

***N*8.** [Пересылка R_j.] (Шаги $N8 \ldots N11$ двойственны по отношению к шагам $N4 \ldots N7$.) Установить $R_k \leftarrow R_j$, $k \leftarrow k + d$.

***N*9.** [Ступенька вниз?] Уменьшить j на 1. Затем, если $K_{j+1} \le K_j$, то вернуться к шагу $N3$.

***N*10.** [Пересылка R_i.] Присвоить $R_k \leftarrow R_i$, $k \leftarrow k + d$.

***N*11.** [Ступенька вниз?] Увеличить i на 1. Затем, если $K_{i-1} \le K_i$, то вернуться к шагу *N*10.

***N*12.** [Переключение направления.] Присвоить $f \leftarrow 0$, $d \leftarrow -d$ и взаимозаменить $k \leftrightarrow l$. Вернуться к шагу *N*3.

***N*13.** [Переключение областей.] Если $f = 0$, то присвоить $s \leftarrow 1 - s$ и вернуться к шагу *N*2. В противном случае сортировка завершена; если $s = 0$, то присвоить $(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N})$. (Если результат можно оставить в области $(R_{N+1}, \ldots, R_{2N})$, то выполнение последнего копирования необязательно.) ■

Проанализируем данный алгоритм. Если файл случаен, то в нём имеется около $\frac{1}{2}N$ возрастающих отрезков, так как $K_i > K_{i+1}$ с вероятностью $\frac{1}{2}$. При каждом просмотре число отрезков сокращается вдвое, за исключением необычных случаев. Таким образом, число просмотров, как правило, составляет около $\log_2 N$. При каждом просмотре должны быть переписаны все N записей, и большая часть времени затрачивается в шагах *N*3, *N*4, *N*5, *N*8 и *N*9. Если считать, что равные ключи встречаются с малой вероятностью, то при каждом просмотре на каждую запись затрачивается 12.5 единиц времени, и общее время работы асимптотически приближается к $12.5N \log_2 N$ как в среднем, так и в наихудшем случае. Это медленнее быстрой сортировки и не настолько лучше времени работы пирамидальной сортировки ($16N \log_2 N$), чтобы оправдать вдвое больший расход памяти.

В алгоритме *N* границы между отрезками полностью определяются «ступеньками вниз». Такой подход обладает тем возможным преимуществом, что исходные файлы с преобладанием возрастающего или убывающего расположения элементов могут обрабатываться очень быстро, но при этом замедляется основной цикл вычислений. Вместо проверок ступенек вниз можно принудительно установить длину отрезков, считая, что все отрезки исходного файла имеют длину 1, после первого просмотра все отрезки (кроме, возможно, последнего) имеют длину 2, …, после k-го просмотра все отрезки (кроме, возможно, последнего) имеют длину 2^k. В отличие от «естественного» слияния в алгоритме *N* такой способ называется *простым двухпутевым слиянием*.

Алгоритм простого двухпутевого слияния очень напоминает алгоритм *N*, тем не менее, методы достаточно отличаются друг от друга.

Алгоритм *S*. (*Сортировка простым двухпутевым слиянием.*)

Как и в алгоритме *N*, при сортировке записей $R_1, \ldots, R_N$ используются две области памяти.

***S*1.** [Начальная установка.] Присвоить $s \leftarrow 0$, $p \leftarrow 1$. (Смысл переменных s, i, j, k, l и d приведён в описании алгоритма *N*. Здесь p – размер

возрастающих отрезков, которые будут сливаться во время текущего просмотра; q и r – количества неслитых элементов в отрезках.)

S2. [Подготовка к просмотру.] Если $s = 0$, то присвоить $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N$, $l \leftarrow 2N + 1$; если $s = 1$, то присвоить $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 0$, $l \leftarrow N + 1$. Затем присвоить $d \leftarrow 1$, $q \leftarrow p$, $r \leftarrow p$.

S3. [Сравнение K_i и K_j.] Если $K_i > K_j$, то перейти к шагу *S*8.

S4. [Пересылка R_i.] Присвоить $k \leftarrow k + d$, $R_k \leftarrow R_i$.

S5. [Конец отрезка?] Присвоить $i \leftarrow i + 1$, $q \leftarrow q - 1$. Если $q > 0$, то вернуться к шагу *S*3.

S6. [Пересылка R_j.] Присвоить $k \leftarrow k + d$. Затем, если $k = l$, то перейти к шагу *S*13; в противном случае присвоить $R_k \leftarrow R_j$.

S7. [Конец отрезка?] Присвоить $j \leftarrow j - 1$, $r \leftarrow r - 1$. Если $r > 0$, то вернуться к шагу *S*6; в противном случае перейти к шагу *S*12.

S8. [Пересылка R_j.] Присвоить $k \leftarrow k + d$, $R_k \leftarrow R_j$.

S9. [Конец отрезка?] Установить $j \leftarrow j - 1$, $r \leftarrow r - 1$. Если $r > 0$, то вернуться к шагу *S*3.

S10. [Пересылка R_i.] Установить $k \leftarrow k + d$. Затем, если $k = l$, то перейти к шагу *S*13; в противном случае присвоить $R_k \leftarrow R_i$.

S11. [Конец отрезка?] Выполнить присваивания $i \leftarrow i + 1$, $q \leftarrow q - 1$. Если $q > 0$, то вернуться к шагу *S*10.

S12. [Переключение направления.] Присвоить $q \leftarrow p$, $r \leftarrow p$, $d \leftarrow -d$ и взаимозаменить $k \leftrightarrow l$. Если $j - i < p$, то вернуться к шагу *S*10; в противном случае вернуться к шагу *S*3.

S13. [Переключение областей.] Установить $p \leftarrow p + p$. Если $p < N$, то присвоить $s \leftarrow 1 - s$ и вернуться к шагу *S*2. В противном случае сортировка завершена. Если $s = 0$, то присвоить $(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N})$. (Независимо от распределения записей в исходном файле, последнее копирование будет выполнено тогда и только тогда, когда значение $\lceil \log_2 N \rceil$ нечётно. Таким образом, можно заранее предсказать положение отсортированного файла, и тогда копирование не потребуется.) ■

Пример работы алгоритма сортировки простым двухпутевым слиянием представлен в табл. 2. Отметим, что метод справедлив и тогда, когда число сортируемых записей N не является степенью двойки. В отличие от алгоритма *N*, здесь вместо проверок ступенек вниз уменьшаются значения переменных q и r до достижения равенства нулю. Благодаря этому время работы алгоритма *S* асимптотически приближается к $11N \log_2 N$ единицам, что несколько лучше, чем в алгоритме *N*.

Заметим, что на практике имеет смысл комбинировать алгоритм *S* с простыми вставками. Например, вместо первых четырёх просмотров алгоритма *S* можно простыми вставками отсортировать группы из 16 элементов, исключив довольно расточительные вспомогательные операции, связанные со слиянием коротких файлов.

Таблица 2

503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703
503	703	512	677	509	908	426	897	653	275	170	154	612	61	765	87
87	503	703	765	154	170	509	908	897	653	426	275	677	612	512	61
61	87	503	512	612	677	703	765	908	897	653	509	426	275	170	154
61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Рассмотрим теперь алгоритмы N и S с точки зрения структур данных. Для работы этих алгоритмов необходима память под $2N$ записей, поскольку в каждом просмотре работа ведётся с четырьмя списками переменного размера: двумя «входными списками» и двумя «выходными списками». С целью экономии памяти можно воспользоваться её *связанным* распределением. При этом к каждой из N записей добавить по одному полю связи, и все необходимые слияния проделать, манипулируя со связями и не перемещая сами записи. Заметим, что добавление N полей связи, как правило, выгоднее, чем добавление пространства памяти ещё под N записей, и, кроме того, за счёт отказа от перемещения записей может быть также сэкономлено и время.

Алгоритм *L*. (*Сортировка посредством слияния списков.*) Предполагается, что записи $R_1, \ldots, R_N$ содержат ключи $K_1, \ldots, K_N$ и «поля связи» $L_1, \ldots, L_N$, в которых могут храниться числа от минус $(N + 1)$ до $(N + 1)$. В начале и конце файла имеются искусственные записи R_0 и R_{N+1} с полями связи L_0 и L_{N+1}. Этот алгоритм сортировки списков устанавливает поля связи таким образом, что записи оказываются связанными в возрастающем порядке. После завершения сортировки L_0 указывает на запись с наименьшим ключом; при $1 \le k \le N$ связь L_k указывает на запись, следующую за R_k, а если R_k является записью с наибольшим ключом, то связь $L_k = 0$.

В процессе выполнения этого алгоритма записи R_0 и R_{N+1} служат «головами» двух линейных списков, подсписки которых в данный момент сливаются. Отрицательная связь означает конец подсписка, о котором известно, что он упорядочен. Нулевая связь означает конец всего списка. Предполагается, что число записей $N \ge 2$.

Через оператор вида $|L_s| \leftarrow p$ обозначено присваивание переменной L_s значения p или $-p$ с сохранением прежнего знака значения указателя L_s.

***L*1.** [Подготовка двух списков.] Присвоить $L_0 \leftarrow 1$, $L_{N+1} \leftarrow 2$, $L_i \leftarrow -(i + 2)$ при $1 \le i \le N - 2$ и $L_{N-1} \leftarrow L_N \leftarrow 0$. (Созданы два списка, содержащие соответственно записи $R_1, R_3, R_5, \ldots$ и $R_2, R_4, R_6, \ldots$. Отрицательные связи говорят о том, что каждый упорядоченный «подсписок» состоит всего лишь из одного элемента.)

***L*2.** [Начало нового просмотра.] Присвоить $s \leftarrow 0$, $t \leftarrow N + 1$, $p \leftarrow L_s$, $q \leftarrow L_t$. Если $q = 0$, то работа алгоритма завершена. (При каждом про-

смотре p и q пробегают по спискам, которые подвергаются слиянию; s обычно указывает на последнюю обработанную запись текущего подсписка, а t – на конец только что выведенного подсписка.)

***L*3.** [Сравнение K_p и K_q.] Если $K_p > K_q$, то перейти к шагу *L*6.

***L*4.** [Продвижение p.] Присвоить $|L_s| \leftarrow p$, $s \leftarrow p$, $p \leftarrow L_p$. Если $p > 0$, то вернуться к шагу *L*3.

***L*5.** [Завершение подсписка.] Присвоить $L_s \leftarrow q$, $s \leftarrow t$. Затем присвоить $t \leftarrow q$ и $q \leftarrow L_q$ один или более раз, пока не станет $q \le 0$, после чего перейти к шагу *L*8.

***L*6.** [Продвижение q.] (Шаги *L*6 и *L*7 двойственны по отношению к *L*4 и *L*5.) Присвоить $|L_s| \leftarrow q$, $s \leftarrow q$, $q \leftarrow L_q$. Если $q > 0$, вернуться к шагу *L*3.

***L*7.** [Завершение подсписка.] Присвоить $L_s \leftarrow p$, $s \leftarrow t$. Затем присвоить $t \leftarrow p$ и $p \leftarrow L_p$ один или более раз, пока не станет $p \le 0$.

***L*8.** [Конец просмотра?] (К этому моменту $p \le 0$ и $q \le 0$, так как оба указателя продвинулись до конца соответствующих подсписков.) Присвоить $p \leftarrow -p$, $q \leftarrow -q$. Если $q = 0$, то присвоить $|L_s| \leftarrow p$, $|L_t| \leftarrow 0$ и вернуться к шагу *L*2; в противном случае вернуться к шагу *L*3. ■

Пример работы этого алгоритма иллюстрирует табл. 3, в которой представлены значения связей L_j к моменту непосредственно перед выполнением шага *L*2. По окончании работы алгоритма *L* записи можно переразместить так, чтобы их ключи были упорядочены.

Таблица 3

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
K_j	–	503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703	–
L_j	1	–3	–4	–5	–6	–7	–8	–9	–10	–11	–12	–13	–14	–15	–16	0	0	2
L_j	2	–6	1	–8	3	–10	5	–11	7	–13	9	12	–16	14	0	0	15	4
L_j	4	3	1	–11	2	–13	8	5	7	0	12	10	9	14	16	0	15	6
L_j	4	3	6	7	2	0	8	5	1	14	12	10	13	9	16	0	15	11
L_j	4	12	11	13	2	0	8	5	10	14	1	6	3	9	16	7	15	0

Время работы алгоритма *L* в среднем равно приблизительно $(10N \log_2 N + 4.92N)$ единиц с небольшим стандартным отклонением порядка $\sqrt{N}$.

Таким образом, связанное распределение памяти имеет бесспорные преимущества по сравнению с последовательным, поскольку при этом требуется меньше памяти, а программа работает на 10% быстрее.

Упражнения

1. Слейте по алгоритму *M* два упорядоченных подфайла 12, 13, 14, 14, 25, 42, 47, 62 и 16, 35, 46, 52, 61, 73, 91, 103 в один файл и установите сколько раз при этом будут выполнены шаги *M*3 и *M*5.

2. Отсортируйте естественным двухпутевым слиянием по алгоритму *N* последовательность из девятнадцати записей: (11 «смотри»), (9 «воду»), (92 «забавой»), (35 «:»), (38 «иначе»), (9 «камешки»), (26 «,»), (23 «круги»), (31 «образуемые»), (69 «такое»), (4 «в»), (96 «.»), (29 «ими»), (11 «на»), (1 «Бросая»), (88 «пустою»), (82 «будет»), (71 «бросание»), (10 «,») и установите, сколько при этом будет сделано просмотров (выполнений шага *N*2).

3. Выполните по алгоритму *S* сортировку простым двухпутевым слиянием последовательность из пятнадцати записей: (13 «,»), (43 «свои»), (26 «прищурь»), (54 «разглядишь»), (7 «на»), (57 «на»), (71 «пятна»), (97 «.»), (69 «нем»), (43 «и»), (4 «Взирая»), (49 «ты»), (10 «солнце»), (49 «смело»), (29 «глаза») и установите размещение ключей в двух используемых областях памяти до сортировки, а также по окончании каждого просмотра.

4. Отсортируйте посредством слияния списков по алгоритму *L* последовательность из пятнадцати записей: (70 «хорошо»), (85 «иногда»), (66 «иногда»), (43 «белой»), (55 «:»), (67 «напишется»), (0 «Память»), (93 «.»), (3 «есть»), (83 «а»), (88 «дурно»), (5 «лист»), (2 «человека»), (74 «,»), (45 «бумаги») и установите распределение связей для двух линейных списков перед началом сортировки, а также по окончании каждого из просмотров.

15. РАСПРЕДЕЛЯЮЩАЯ СОРТИРОВКА

Рассмотрим интересный класс методов сортировки, который, по существу, прямо *противоположен* слиянию. Предположим, что нужно отсортировать колоду из 52 игральных карт. Для этого введём упорядочение по старшинству (достоинству) карт в масти

$$\text{Т} < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < \text{В} < \text{Д} < \text{К},$$

а также по масти

$$\clubsuit < \blacklozenge < \heartsuit < \spadesuit .$$

Заметим, что одна какая-нибудь карта предшествует другой, если она либо младше по масти; либо масти обеих карт одинаковы, но она младше по достоинству. Таким образом, отсортированная колода карт должна быть упорядочена следующим образом:

$$\text{Т}\clubsuit < 2\clubsuit < \ldots < \text{К}\clubsuit < \text{Т}\blacklozenge < \ldots < \text{Д}\spadesuit < \text{К}\spadesuit.$$

Естественно было бы отсортировать карты сначала по масти, разложив их в четыре стопки, а затем перекладывать карты внутри каждой стопки до тех пор, пока они не будут упорядочены по достоинству.

Но существует более быстрый способ. Он заключается в том, что сначала карты следует разложить в 13 стопок лицевой стороной вверх по их достоинству. Затем собрать все стопки вместе: снизу тузы, затем двойки, тройки и т.д. и сверху короли (лицевой стороной вверх). Перевернуть колоду рубашками вверх и снова разложить, на этот раз в четыре стопки по масти. Сложив вместе полученные стопки так, чтобы внизу были трефы, затем бубны, черви и пики, получим упорядоченную колоду карт.

Идея такой сортировки является правильной, потому что, если две карты при последнем раскладе попали в разные стопки, то они имеют разные масти, так что карта с меньшей мастью младше другой карты. Если же две карты имеют одну и ту же масть, то они уже находятся в нужном порядке благодаря предварительной сортировке. Заметим, что это доказательство можно обобщить на сортировку любого множества в лексикографическом порядке.

Такой метод упорядочения естественным образом приводит к идее поразрядной сортировки. Чтобы выполнить поразрядную сортировку с помощью ЭВМ, необходимо определиться со способом представления стопок. Если имеется M стопок, то можно выделить M областей памяти и пересылать каждую исходную запись в соответствующую область. Но при этом в каждой области должно быть достаточно места для хранения N элементов, и тогда потребуется пространство под $(M+1)N$ записей. Однако того же эффекта можно добиться, имея в распоряжении пространство всего под $2N$ записей и M счётчиков. Сделав один предварительный просмотр данных, можно установить, сколько элементов попадает в каждую область, что даст возможность точно распределить память под стопки. Такая идея уже была применена при сортировке распределяющим подсчётом (алгоритм D).

Следовательно, поразрядную сортировку можно выполнить так. Сначала произвести распределяющую сортировку *по младшим цифрам ключей* (в системе счисления с основанием M), переместив записи из области ввода во вспомогательную область. Затем выполнить ещё одну распределяющую сортировку, но уже по следующей цифре, переместив записи обратно в исходную область. Продолжать подобные действия до тех пор, пока после завершающей сортировки по старшей цифре все ключи не окажутся расположенными в нужном порядке.

В таблице показано применение поразрядной сортировки к нашим 16 ключам при $M = 10$. При таких малых N поразрядная сортировка, как правило, неэффективна, так что этот пример предназначен главным образом для того, чтобы продемонстрировать достаточность применяемого метода:

Область ввода	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
Счётчики для младших цифр							1	1	2	3	1	2	1	3	1	1

Распределение памяти							1	2	4	7	8	10	11	14	15	16
Вспом. область	170	061	512	612	503	653	703	154	275	765	426	087	897	677	908	509
Счётчики для средних цифр							4	2	1	0	0	2	2	3	1	1
Распределение памяти							4	6	7	7	7	9	11	14	15	16
Область ввода	503	703	908	509	512	612	426	653	154	061	765	170	275	677	087	897
Счётчики для старших цифр							2	2	1	0	1	3	3	2	1	1
Распределение памяти							2	4	5	5	6	9	12	14	15	16
Вспом. область	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Заметим, что идея счётчиков для распределения памяти привязана к «старомодным» понятиям о последовательном представлении данных, а для работы с множеством таблиц переменной длины специально придумано *связанное* распределение. Поэтому для поразрядной сортировки естественно будет воспользоваться связанными структурами данных. Так как каждая стопка просматривается последовательно, при каждом элементе достаточно иметь ссылку на следующий элемент. Кроме того, при связанном распределении, как известно, нет необходимости перемещать записи, а достаточно лишь должным образом скорректировать связи в списках. При таком подходе необходима память в объёме $(1 + \varepsilon)N + 2\varepsilon M$ записей, где ε – пространство, занимаемое одним полем связи.

Рассмотрим алгоритм, который является ярким примером типичных манипуляций со структурами данных, соединяющих в себе последовательное и связанное распределение памяти.

Алгоритм *R*. (*Поразрядная сортировка списка*.)

Предполагается, что каждая из записей $R_1, \ldots, R_N$ содержит поле связи *LINK*, а ключи представляют собой последовательность из p элементов:

$$(a_p, \ldots, a_2, a_1),\ 0 \le a_i < M.$$

Отношение порядка – лексикографическое, т.е.

$$(a_p, \ldots, a_2, a_1) < (b_p, \ldots, b_2, b_1)$$

тогда и только тогда, когда существует такой индекс j, $1 \le j \le p$, что

$$a_i = b_i \text{ при } i > j, \text{ но } a_j < b_j.$$

Ключи могут быть представлены, в частности, как числа в системе счисления с основанием M:

$$a_p M^{p-1} + \ldots + a_2 M + a_1,$$

и в этом случае лексикографическое отношение порядка соответствует обычному упорядочению множества неотрицательных чисел. Ключи также могут быть цепочками букв алфавита и т.д.

Во время сортировки формируется M «стопок», которые фактически представляют собой очереди, поскольку они всегда просматриваются по принципу «первым включается – первым исключается». Для каждой стопки имеется два указателя: $TOP[i]$ и $BOTM[i]$, $0 \le i < M$.

***R*1.** [Цикл по k.] Вначале установить $P \leftarrow LOC(R_N)$ – указатель на последнюю запись. Затем выполнить шаги с *R*2 по *R*6 при $k = 1, 2, \ldots, p$ (шаги с *R*2 по *R*6 составляют один «просмотр») и завершить работу алгоритма. Переменная P будет указывать на запись с наименьшим ключом, $LINK(P)$ – на запись со следующим по величине ключом, $LINK(LINK(P))$ – на следующую и т.д. Поле $LINK$ последней записи будет равно Λ.

***R*2.** [Очистка стопок.] При $0 \le i < M$ установить $TOP[i] \leftarrow \Lambda$ и $BOTM[i] \leftarrow \Lambda$.

***R*3.** [Выделение k-й цифры ключа.] Пусть $KEY(P)$ – ключ записи, на которую указывает P, и равен $(a_p, \ldots, a_2, a_1)$. Присвоить $i \leftarrow a_k$ (k-я младшая цифра этого ключа).

***R*4.** [Коррекция связей.] Если $BOTM[i] = \Lambda$, то $BOTM[i] \leftarrow P$ и $TOP[i] \leftarrow P$; иначе $LINK(TOP[i]) \leftarrow P$ и затем $TOP[i] \leftarrow P$.

***R*5.** [Переход к следующей записи.] Если $k = 1$ (первый просмотр) и если $P = LOC(R_j)$ при некотором $j \ne 1$, то присвоить $P \leftarrow LOC(R_{j-1})$ и вернуться к шагу *R*3. Если $k > 1$ (не первый просмотр), то присвоить $P \leftarrow LINK(P)$ и если $P \ne \Lambda$, то вернуться к шагу *R*3.

***R*6.** [Вызов алгоритма H.] (Теперь все элементы уже распределены по стопкам.) Выполнить приведённый ниже алгоритм H, который сцепляет отдельные «стопки» в один список, подготавливая их к следующему просмотру. Затем выполнить $P \leftarrow BOTM[0]$ – указатель на первый элемент объединённого списка. ■

Алгоритм *H*. (*Сцепление очередей*.) Из M данных очередей со связями, удовлетворяющими соглашениям алгоритма R, данный алгоритм создаёт одну очередь, меняя при этом не более чем M связей. В результате $BOTM[0]$ указывает на первый элемент, и стопка 0 предшествует стопке 1, …, стопка $(M-2)$ предшествует стопке $(M-1)$.

***H*1.** [Начальная установка.] Присвоить $i \leftarrow 0$.

***H*2.** [Указатель на вершину стопки.] Выполнить $P \leftarrow TOP[i]$.

***H*3.** [Следующая стопка.] Увеличить i на 1. Если $i = M$, то присвоить $LINK(P) \leftarrow \Lambda$ и конец алгоритма.

***H*4.** [Стопка пуста?] Если $BOTM[i] = \Lambda$, то вернуться к шагу *H*3.

***H*5.** [Сцепить стопки.] Присвоить $LINK(P) \leftarrow BOTM[i]$ и вернуться к шагу *H*2. ■

На рисунках 33 – 35 показано содержимое стопок после каждого из трёх просмотров, выполняемых при сортировке наших шестнадцати чисел с $M = 10$. Алгоритм R очень просто запрограммировать, если только найти удобный способ изменять от просмотра к просмотру действия в шагах $R3$ и $R5$.

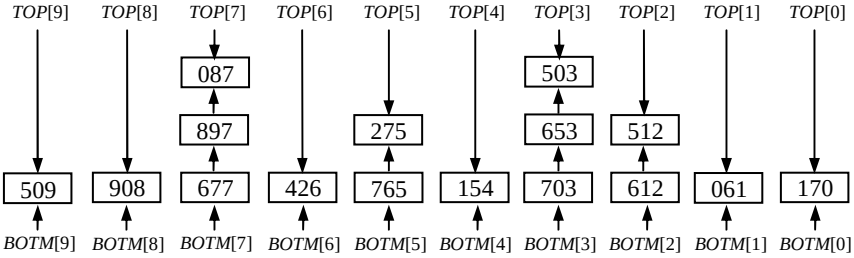


Рис. 33. Содержимое стопок после первого просмотра

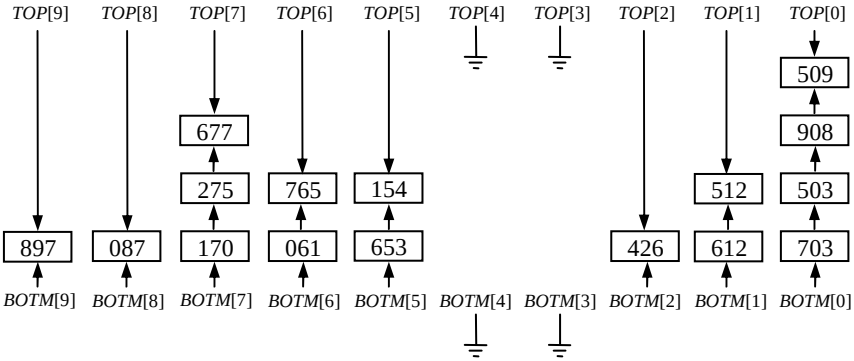


Рис. 34. Содержимое стопок после второго просмотра

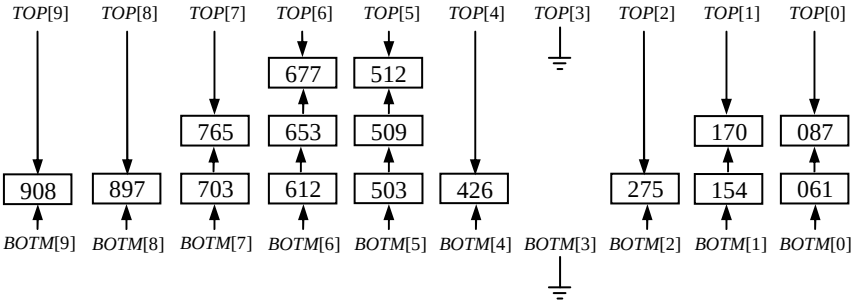


Рис. 35. Содержимое стопок после третьего просмотра

Время работы алгоритма R равно $32N + 48M + 38 - 4E$, где N – число исходных записей, M – основание системы счисления (число стопок), E – число встретившихся пустых стопок. Сравнение с другими алгоритмами, построенными на основе аналогичных предположений (например, Алгоритм L. Сортировка посредством слияния списков), говорит явно в пользу алгоритма R. Заметим, что если ключи уж очень длинные, поразрядная сортировка не так эффективна, многие просмотры будут выполняться почти впустую.

Упражнения

1. Выполните по алгоритму R поразрядную сортировку последовательности из пятнадцати записей: (623 «к»), (390 «:»), (858 «обнажаются»), (874 «ноги»), (379 «одеялу»), (555 «когда»), (47 «распутного»), (12 «Достаток»), (701 «носу»), (589 «натянешь»), (841 «,»), (167 «равняется»), (322 «короткому»), (610 «его»), (905 «.») и установите содержимое стопок после каждого из трёх просмотров, выполняемых при сортировке этих записей с $M = 10$.

16. ВНЕШНЯЯ СОРТИРОВКА

Рассмотрим задачи, возникающие при внешней сортировке, т.е. в случае, когда общий объём сортируемых записей превышает объём оперативного запоминающего устройства. Внешняя сортировка принципиально отличается от внутренней сортировки и это отличие объясняется тем, что время доступа к файлам на внешних носителях информации жёстко лимитирует процесс. Структура данных должна быть такой, чтобы сравнительно медленные периферийные запоминающие устройства (ленты, диски, барабаны и т.д.) могли справиться с потребностями алгоритма сортировки. Большинство изученных до сих пор методов внутренней сортировки (вставка, обмен, выбор) фактически бесполезно для внешней сортировки.

Предположим, например, что предназначенный для сортировки *файл последовательного доступа* состоит из 5 000 000 записей $R_1\ R_2\ \ldots\ R_{5\,000\,000}$. Рассмотрим случай, когда во внутренней памяти ЭВМ помещается только 1 000 000 из этих записей.

Сразу напрашивается такое решение: начать с сортировки каждого из пяти подфайлов $R_1 \ldots R_{1\,000\,000}$, $R_{1\,000\,001} \ldots R_{2\,000\,000}$, … , $R_{4\,000\,001} \ldots R_{5\,000\,000}$ по отдельности и затем слить полученные подфайлы. Отметим, слияние оперирует только очень простыми структурами данных, а именно линейными списками, пройти которые можно последовательным образом.

Заметим, что описанный процесс внешней сортировки, заключающийся во внутренней сортировке с последующим «внешним слиянием», является весьма популярным.

Рассмотрим процесс внешней сортировки, использующий *сбалансированное двухпутевое слияние*, в основе которого лежит идея, применённая ранее в алгоритмах *N*, *S* и *L* (см. параграф 14). В процессе слияния нам потребуются четыре *рабочих файла*.

На протяжении первой фазы возрастающие отрезки, получаемые при внутренней сортировке, помещаются поочередно в файлы 1 и 2 до тех пор, пока не исчерпаются исходные данные. Затем указатели файлов 1 и 2 обнуляются и сливаются отрезки, находящиеся в этих файлах, и получаются новые отрезки, вдвое длиннее исходных. Эти новые отрезки записываются по мере их формирования попеременно в файлы 3 и 4. (Если в файле 1 на один отрезок больше, чем в файле 2, то предполагается, что файл 2 содержит дополнительный «фиктивный» отрезок длины 0.) Затем файловые указатели всех файлов обнуляются, и содержимое файлов 3 и 4 сливается в удвоенные по длине отрезки, записываемые поочередно в файлы 1 и 2. Процесс продолжается (при этом длина отрезков каждый раз удваивается) до тех пор, пока не останется один отрезок (а именно весь упорядоченный файл). Если после внутренней сортировки было получено S отрезков, причём $2^{k-1} < S \le 2^k$, то процедура сбалансированного двухпутевого слияния произведёт ровно $k = \lceil \log_2 S \rceil$ проходов по всем данным.

Например, в рассмотренной выше ситуации, когда требуется упорядочить 5 000 000 записей, а объём внутренней памяти составляет 1 000 000 записей, мы имеем $S = 5$ и $k = 3$. Начальная распределительная фаза процесса сортировки поместит пять отрезков в файлы следующим образом:

$$\begin{array}{ll} \text{Файл 1} & R_1 \ldots R_{1\,000\,000};\ R_{2\,000\,001} \ldots R_{3\,000\,000};\ R_{4\,000\,001} \ldots R_{5\,000\,000}. \\ \text{Файл 2} & R_{1\,000\,001} \ldots R_{2\,000\,000};\ R_{3\,000\,001} \ldots R_{4\,000\,000}. \\ \text{Файл 3} & \text{(пустой)} \\ \text{Файл 4} & \text{(пустой)} \end{array} \qquad (52)$$

Поскольку записей в файле 2 оказалось меньше, чем в файле 1, в конец файла 2 неявно добавляется фиктивный отрезок.

После первого прохода слияния в файлах 3 и 4 получатся в два раза более длинные отрезки, чем в файлах 1 и 2:

$$\begin{array}{ll} \text{Файл 3} & R_1 \ldots R_{2\,000\,000};\ R_{4\,000\,001} \ldots R_{5\,000\,000}. \\ \text{Файл 4} & R_{2\,000\,001} \ldots R_{4\,000\,000}. \end{array} \qquad (53)$$

Заметим, что за счёт фиктивного отрезка в файле 2 отрезок $R_{4\,000\,001} \ldots R_{5\,000\,000}$ просто был скопирован из файла 1 в файл 3. После обнуления указателей всех четырёх файлов будет выполнен второй проход по данным и их слияние, что приведёт к такому результату:

Файл 1 $R_1 \ldots R_{4\,000\,000}$.
Файл 2 $R_{4\,000\,001} \ldots R_{5\,000\,000}$. (54)

Наконец, после еще одного обнуления указателей и слияния данных в файле 3 окажется отрезок $R_1 \ldots R_{5\,000\,000}$, и сортировка закончится.

Сбалансированное слияние легко обобщается на случай T файлов для любого $T \geq 3$. Выберем произвольное число P такое, что $1 \leq P < T$, и разделим T файлов на два «банка»: P файлов в левом банке и $(T - P)$ файлов в правом банке. Распределим исходные отрезки как можно равномернее по P файлам левого «банка», затем выполним P-путевое слияние слево направо, после этого – $(T - P)$-путевое слияние справа налево и т.д., пока сортировка не завершится. Обычно значение P лучше всего выбирать равным $\lceil T / 2 \rceil$.

Отметим, что частный случай сбалансированного слияния при $T = 4$ и $P = 2$ соответствует сбалансированному двухпутевому слиянию.

Вновь рассмотрим предыдущий пример, но с использованием большего количества файлов. Если $T = 6$ и $P = 3$, то начальное распределение будет следующим:

Файл 1 $R_1 \ldots R_{1\,000\,000}$; $R_{3\,000\,001} \ldots R_{4\,000\,000}$.
Файл 2 $R_{1\,000\,001} \ldots R_{2\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$. (55)
Файл 3 $R_{2\,000\,001} \ldots R_{3\,000\,000}$.

Первый проход слияния приведёт к состоянию:

Файл 4 $R_1 \ldots R_{3\,000\,000}$.
Файл 5 $R_{3\,000\,001} \ldots R_{5\,000\,000}$. (56)
Файл 6 (пустой)

При этом предполагается, что в файле 3 помещён фиктивный отрезок. По окончании второго прохода слияния сортировка завершается, и отрезок $R_1 \ldots R_{5\,000\,000}$ помещается в файл 1.

Отметим, что рассмотренный случай для $T = 6$ эквивалентен случаю для $T = 5$, так как шестой файл будет задействован при сортировке лишь для $S \geq 7$.

Сбалансированное слияние кажется очень простым и естественным. Но если приглядеться внимательнее, то видно, что это не наилучший способ в рассмотренных выше частных случаях. Вместо того чтобы переходить от (52) к (53) и обнулять указатели всех файлов, следовало бы остановить первое слияние, когда файлы 3 и 4 содержали соответственно $R_1 \ldots R_{2\,000\,000}$ и $R_{2\,000\,001} \ldots R_{4\,000\,000}$, а файл 1 был готов к считыва-

нию $R_{4\,000\,001} \ldots R_{5\,000\,000}$. Затем указатели файлов 2, 3, 4 могли быть обнулены, и сортировка завершилась бы трёхпутевым слиянием в файл 2. Общее число записей, прочитанных из файлов в ходе этой процедуры, составило бы 4 000 000 + 5 000 000 = 9 000 000 против 5 000 000 + 5 000 000 + + 5 000 000 = 15 000 000 в сбалансированной схеме.

Имея пять отрезков и четыре файла, можно поступить ещё лучше, распределив отрезки следующим образом:

$$\begin{array}{ll} \text{Файл 1} & R_1 \ldots R_{1\,000\,000};\ R_{3\,000\,001} \ldots R_{4\,000\,000}. \\ \text{Файл 2} & R_{1\,000\,001} \ldots R_{2\,000\,000};\ R_{4\,000\,001} \ldots R_{5\,000\,000}. \\ \text{Файл 3} & R_{2\,000\,001} \ldots R_{3\,000\,000}. \\ \text{Файл 4} & \text{(пустой)} \end{array} \qquad (57)$$

Теперь, выполнив трёхпутевое слияние в файл 4, затем обнуление указателей файлов 3 и 4 с последующим трёхпутевым слиянием в файл 3, можно было бы завершить сортировку, прочитав всего 3 000 000 + + 5 000 000 = 8 000 000 записей.

Наконец, если бы имелось шесть файлов, то можно было бы записать исходные отрезки в файлы 1 – 5 и закончить сортировку за один проход, выполнив пятипутевое слияние в файл 6. Рассмотрение этих случаев показывает, что простое сбалансированное слияние не является наилучшим, и поэтому имеет смысл поиск улучшенных схем слияния.

Далее внешнюю сортировку в следующем параграфе исследуем более глубоко. Рассмотрим фазу внутренней сортировки, порождающей начальные отрезки. Здесь особый интерес представляет технология «выбора с замещением», которая порождает длинные отрезки, значительно превосходящие ёмкость внутренней памяти. Обсудим структуры данных, удобные для целей многопутевого слияния, а также важнейшие схемы слияния.

Упражнения

1. В данном параграфе внешней сортировке предшествует фаза внутренней сортировки. Установите, можно ли не выполняя внутренней сортировки, производить слияние записей во все более и более длинные отрезки с самого начала.

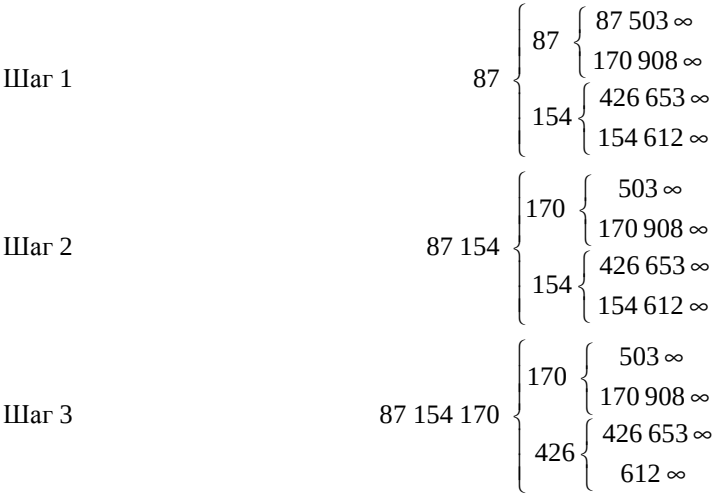
2. Установите, каким будет содержимое файлов (аналогичное (52) – (54)), если записи $R_1 \ldots R_{5\,000\,000}$ сортируются с помощью трёхфайлового сбалансированного метода при $P = 2$. Сравните этот случай со слиянием на четырёх файлах. Оцените, сколько проходов по всем данным будет сделано после первоначального распределения отрезков.

17. МНОГОПУТЕВОЕ СЛИЯНИЕ И ВЫБОР С ЗАМЕЩЕНИЕМ

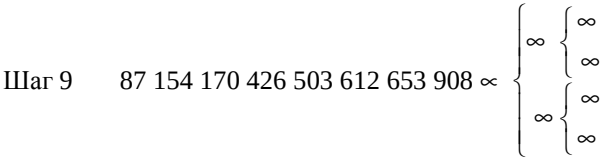
Расширим понятие двухпутевого слияния до понятия P-путевого слияния, когда P входных отрезков объединяются в один выходной.

Пусть имеется P возрастающих отрезков, т.е. последовательностей записей, ключи которых расположены в неубывающем порядке. Очевидным способом их слияния будет следующий: посмотреть на первые записи каждого отрезка и выбрать из них ту, которой соответствует минимальный ключ; эта запись передаётся на выход и исключается из входных данных, затем процесс повторяется.

Пока P не слишком велико, этот выбор удобно осуществлять, просто выполняя $P-1$ сравнений для нахождения наименьшего из текущих ключей. Но если, скажем, $P \geq 8$, то можно сократить работу, используя *дерево выбора* (см. параграф 13). При этом каждый раз требуется только приблизительно $\log P$ сравнений (после начального формирования дерева). Рассмотрим, например, четырёхпутевое слияние с двухуровневым деревом выбора:



. . . . . .



В этом примере в конце каждого отрезка помещён добавочный ключ ∞, чтобы слияние заканчивалось естественно. Так как внешнее слияние обычно имеет дело с очень длинными отрезками, то эта добавочная запись с ключом ∞ не увеличит существенно длину данных и объём работы при слиянии. Кроме того, такая «концевая» запись часто служит удобным способом разделения записей файла.

В рассматриваемом процессе каждый шаг, кроме первого, состоит из замещения наименьшего элемента следующим элементом из того же отрезка и изменения соответствующего пути в дереве выбора. Так, на шаге 2 изменяется 3 узла, которые содержали ключ 87 на шаге 1. На шаге 3 изменяется 3 узла, содержавшие ключ 154 на шаге 2, и т.д. Процесс замещения в дереве выбора одного ключа другим называется *выбором с замещением*.

Каждый концевой узел дерева выбора представляет один из отрезков, используемых в процессах слияния, а само дерево, по существу, используется как приоритетная очередь с дисциплиной «наименьший из включённых первым исключается».

На рисунке 36 изображено дерево «проигравших», представляющее собой полное бинарное дерево с 12 концевыми и 11 внутренними узлами. В концевых узлах записаны ключи, а во внутренних – «победители», если дерево рассматривать как турнир для выбора наименьшего ключа. Числа, стоящие рядом с вершинами дерева, обозначают позиции записей при последовательном распределении памяти для полного бинарного дерева.

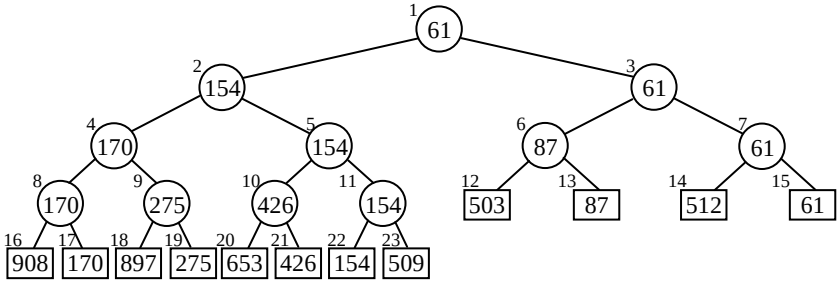


Рис. 36. Турнир, в котором выбирается наименьший ключ

Чтобы определить новое состояние дерева выбора, когда наименьший ключ 61 будет заменён другим ключом, нужно рассмотреть только ключи 512, 87 и 154. Если интерпретировать дерево как турнир, эти три ключа представляют проигравших в матчах с игроком 61. Поэтому во внутренние узлы дерева необходимо записывать проигравшего в каждом матче, а не победителя, и тогда информация, необходимая для изменения этого дерева, будет легкодоступной.

На рисунке 37 изображено то же дерево, что и на рис. 36, но вместо победителей в нём представлены проигравшие. Дополнительный узел с номером 0 добавлен на вершине дерева для указания чемпиона турнира. Заметим, что каждый ключ, кроме чемпиона, является проигравшим ровно один раз. Таким образом, каждый ключ появляется один раз в концевом узле и один раз во внутреннем.

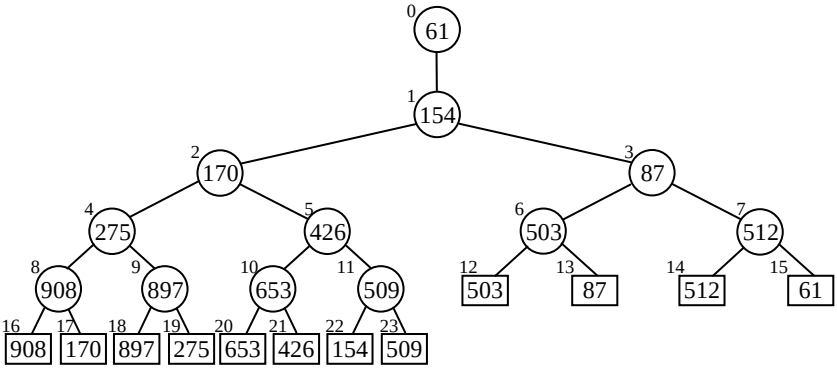


Рис. 37. Турнир, в котором показаны проигравшие, а не победители

На практике конечным узлам дерева на рис. 37 будут соответствовать весьма длинные записи, расположенные в памяти ЭВМ, а внутренним узлам – указатели на эти записи. Заметим, что P-путевое слияние требует ровно P конечных и P внутренних узлов – по одному в соседних группах. Это наводит на мысль о возможности использования известных эффективных методов распределения памяти.

Технология выбора с замещением может быть использована на первой фазе внешней сортировки при получении начальных отрезков, если фактически выполнить P-путевое слияние входных данных с самими собой. В этом случае P выбирается достаточно большим, чтобы заполнить, по существу, всю внутреннюю память. Каждая запись при выводе замещается очередной записью из исходных данных. Если у этой новой записи ключ меньше ключа выведенной записи, то она не включается в текущий отрезок. В противном случае она обычным образом включается в дерево выбора и становится частью отрезка, порождаемого в данный момент. Таким образом, каждый отрезок может содержать больше P записей, хотя в любой момент в дереве выбора находится не более P записей.

Таблица 4 иллюстрирует описанный процесс для $P = 4$. В ней числа, заключённые в скобки, ожидают включения в следующий отрезок.

Таблица 4

Содержимое памяти				Вывод
503	87	512	61	61
503	87	512	908	87
503	170	512	908	170
503	897	512	908	503
(275)	897	512	908	512
(275)	897	653	908	653
(275)	897	(426)	908	897
(275)	(154)	(426)	908	908
(275)	(154)	(426)	(509)	(конец отрезка)
275	154	426	509	154
275	612	426	509	275
677	612	426	509	426
677	612	765	509	509
677	612	765	703	612
677	∝	765	703	677
∝	∝	765	703	703
∝	∝	765	∝	765
∝	∝	∝	∝	(конец отрезка)

Доказано, что для случайных входных данных ожидаемая длина отрезков равна $2P$. Однако во многих практических приложениях входные данные нельзя считать полностью случайными, если в них уже существует определённый порядок. Следовательно, отрезки, порождаемые выбором с замещением, имеют тенденцию содержать даже больше, чем $2P$ записей. Поскольку время, нужное для внешней сортировки слиянием, в значительной степени зависит от количества отрезков, порождаемых начальной распределительной фазой, то выбор с замещением становится особенно привлекательным. Другие способы внутренней сортировки порождали бы примерно вдвое больше начальных отрезков, поскольку размеры памяти ограничены.

Теперь детально рассмотрим процесс создания начальных отрезков посредством выбора с замещением. Следующий алгоритм включает в себя прекрасный способ начального формирования дерева выбора и разделения записей, принадлежащих разным отрезкам, а также вывода последнего отрезка по единообразной и сравнительно простой логике. Заметим, что надлежащая обработка последнего отрезка, порождённого выбором с замещением, оказывается довольно сложной и часто бывает камнем преткновения для программиста.

Основная идея заключается в том, чтобы рассматривать каждый ключ как пару (S, K), где K – первоначальный ключ, а S – номер отрезка,

которому принадлежит данная запись. Выходную последовательность, порождённую выбором с замещением, можно получить, если лексикографически упорядочить эти расширенные ключи. При этом S считается старше чем K.

В приведённом ниже алгоритме для представления дерева выбора используется структура данных, состоящая из P узлов. Предполагается, что j-й узел $X[j]$ содержит c слов, начинающихся с $LOC(X[j]) = L_0 + c\,j$ при $0 \le j < P$. Он представляет как внутренний узел с номером j, так и концевой узел с номером $P + j$ (см. рис. 37). В каждом узле имеется несколько полей:

KEY – ключ, находящийся в данном концевом узле;

RECORD – запись, находящаяся в данном концевом узле (включая *KEY* как подполе);

LOSER – указатель на «проигравшего» в данном внутреннем узле;

RN – номер отрезка, содержащего запись, на которую указывает *LOSER*;

PE – указатель на внутренний узел, расположенный в дереве выбора выше данного конечного узла;

PI – указатель на внутренний узел, расположенный в дереве выбора выше данного внутреннего узла.

Например, при $P = 12$ и внутренний узел с номером 5 и конечный узел с номером 17 на рис. 37 будут представлены узлом $X[5]$ с полями $KEY = $ $= 170$, $LOSER = L_0 + 9c$ (адрес конечного узла с номером 21), $PE = L_0 + 8c$, $PI = L_0 + 2c$.

Значения в полях *PE* и *PI* являются константами, и поэтому нет необходимости хранить их в явном виде. Однако иногда на начальной фазе внешней сортировки для быстрой работы с устройствами ввода/вывода выгоднее хранить эту избыточную информацию, чем вычислять её каждый раз заново.

Алгоритм *R*. (*Выбор с замещением.*)

Этот алгоритм читает записи последовательно из входного файла и записывает их последовательно в выходной файл, производя *RMAX* отрезков длиной P или больше (за исключением последнего отрезка). Имеется $P \ge 2$ узлов $X[0], \ldots, X[P-1]$ с полями, описанными выше.

***R*1.** [Начальная установка.] Присвоить $RMAX \leftarrow 0$, $RC \leftarrow 0$, $LASTKEY \leftarrow \infty$, $Q \leftarrow LOC(X[0])$ и $RQ \leftarrow 0$. (RC – номер текущего отрезка, а *LASTKEY* – ключ последней выведенной записи. Начальное значение *LASTKEY* должно быть больше любого возможного ключа.) Для $0 \le j < P$, если обозначить $J = LOC(X[j])$, начальное содержимое $X[j]$ установить следующим образом:

$$LOSER(J) \leftarrow J; \qquad RN(J) \leftarrow 0;$$
$$PE(J) \leftarrow LOC(X[\lfloor (P + j)/2 \rfloor]); \; PI(J) \leftarrow LOC(X[\lfloor j/2 \rfloor]).$$

(Установка $LOSER(J)$ и $RN(J)$ представляет собой искусственный способ образования начального дерева путём рассмотрения фиктивного отрезка с номером 0, который никогда не выводится.)

***R*2.** [Конец отрезка?] Если $RQ = RC$, то перейти к шагу *R*3. (В противном случае $RQ = RC + 1$, и обработка отрезка с номером RC завершена. Здесь следует выполнить те специальные действия, которые соответствуют схеме слияния для последующих этапов сортировки.) Если $RQ > RMAX$, то алгоритм завершён; в противном случае присвоить $RC \leftarrow RQ$.

***R*3.** [Вывод вершины дерева.] (Сейчас Q указывает на «чемпиона» и RQ – номер его отрезка.) Если $RQ \neq 0$, то вывести $RECORD(Q)$ и присвоить $LASTKEY \leftarrow KEY(Q)$.

***R*4.** [Ввод новой записи.] Если входной файл исчерпан, присвоить $RQ \leftarrow RMAX + 1$ и перейти к шагу *R*5. В противном случае поместить новую запись из входного файла в $RECORD(Q)$. Если $KEY(Q) < LASTKEY$ (т.е. эта запись не принадлежит текущему отрезку), то $RQ \leftarrow RQ + 1$, и теперь, если $RQ > RMAX$, присвоить $RMAX \leftarrow RQ$.

***R*5.** [Подготовка к изменению.] (Сейчас Q указывает на новую запись с номером отрезка RQ.) Присвоить $T \leftarrow PE(Q)$. (T – переменный указатель, который будет двигаться по дереву.)

***R*6.** [Установка нового проигравшего.] Если $RN(T) < RQ$ или если $RN(T) = RQ$ и $KEY(LOSER(T)) < KEY(Q)$, то поменять местами $LOSER(T) \leftrightarrow Q$, $RN(T) \leftrightarrow RQ$. (В переменных Q и RQ запоминается текущий победитель и номер его отрезка.)

***R*7.** [Сдвиг вверх.] Если $T = LOC(X[1])$, то вернуться к шагу *R*2; в противном случае $T \leftarrow PI(T)$ и вернуться к шагу *R*6. ■

В алгоритме *R* говорится о вводе и выводе записей по одной, тогда как практически оказывается лучше читать и записывать относительно большие блоки записей. Следовательно, на самом деле необходимы буферы ввода и вывода. Отметим, что их присутствие в памяти приводит к уменьшению значения P.

Упражнения

1. С помощью алгоритма *R* выбора с замещением (для $P = 4$) установите образуемые в выходном файле отрезки, если во входном файле находится последовательность из шестнадцати записей с ключами:

а) 79, 69, 4, 25, 64, 87, 13, 94, 3, 20, 16, 15, 30, 44, 9, 72;

б) 77, 92, 42, 94_1, 74, 96, 88, 56, 94_2, 45, 55, 40, 36, 37, 22, 5;

в) 3, 64, 34, 61, 32, 98, 12, 65, 21, 73, 35, 82, 38, 81, 54, 69.

18. ПОИСК. ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

Рассмотрим вопросы накопления информации в памяти компьютера и способы её быстрого извлечения. При этом главным образом будем изучать простейшую задачу: поиск информации, сохранённой с конкретным идентификатором. Например, поиск $f(x)$ по заданному x в таблице значений функции f или поиск перевода на английский язык некоторого русского слова.

Пусть имеется набор из N записей, а задача состоит в нахождении одной из них. Как и в случае сортировки предполагается, что каждая запись включает в себя специальное поле, хранящее её *ключ*. При этом требуется N различных значений ключа для того, чтобы каждый из них однозначно идентифицировал связанную с ним запись. Набор всех записей обычно именуется *таблицей*, *файлом* или *базой данных*.

Алгоритм поиска имеет так называемый *аргумент* K, а задача заключается в нахождении записи, для которой K служит ключом. В результате поиска может возникнуть одна из двух ситуаций: либо поиск завершился *успешно* (уникальная запись, содержащая ключ K, найдена), либо поиск окончился *неудачно* (запись с ключом K не найдена). После неудачного поиска иногда требуется внести в таблицу новую запись, содержащую ключ K. Такой метод называется *алгоритмом поиска и вставки*.

Хотя целью поиска является нахождение информации, хранящейся в записи, идентифицируемой по ключу K, для простоты будем рассматривать алгоритмы, предназначенные для поиска K. После нахождения K поиск связанной с ним информации становится тривиальной задачей, зависящей от способа хранения этой информации. Поэтому будем считать задачу поиска выполненной в тот момент, когда найдём ключ K или убедимся в его отсутствии.

Поиск обычно является наиболее «времяёмкой» составляющей многих программ, и замена плохого метода поиска хорошим может значительно увеличить скорость работы программы.

Методы поиска могут быть классифицированы несколькими способами. Во-первых, можно разделить их на *внутренний* и *внешний* поиск так же, как различают алгоритмы сортировки внутренней и внешней.

Во-вторых, возможно деление на *статические* и *динамические* методы поиска. При этом статический поиск предполагает, что содержимое таблицы остаётся неизменным, а главная задача заключается в уменьшении времени поиска. Динамический поиск проводится в таблице, которая часто изменяется путём вставки, а возможно, и удаления элементов.

Третья возможная классификация методов поиска зависит от того, на чём они основаны: на сравнении ключей или на некоторых числовых свойствах ключей.

И, наконец, можно разделить методы сортировки на методы с использованием ключей действительных и преобразованных.

Поиск и сортировка зачастую тесно связаны между собой. Например, рассмотрим следующую задачу. Даны числовые множества $A = \{a_1, a_2, ..., a_m\}$, $B = \{b_1, b_1, ..., b_n\}$ и необходимо определить, является ли множество A подмножеством множества B: $A \subseteq B$. Очевидными представляются следующие варианты её решения:

а) сравнивать каждое $a_i \in A$ со всеми $b_j \in A$ последовательно до нахождения совпадения;

б) сначала отсортировать множества A и B, а затем сделать только один последовательный проход по обоим файлам, сравнивая их элементы;

в) внести все элементы b_j с арифметически преобразованными ключами (*хеширование*) в таблицу и выполнить поиск каждого значения a_i.

Каждое из этих решений имеет свои достоинства для различных значений m и n. Решение (а) требует порядка c_1mn единиц времени, где c_1 – некоторая константа. Решение (б) требует порядка $c_2(m \lg m + n \lg n)$ единиц времени, где c_2 – некоторая константа. При выборе подходящего метода хеширования решение (в) займёт около $c_3m + c_4n$ единиц времени для некоторых (ещё бóльших) констант c_3 и c_4. Отсюда следует, что решение (а) подходит для очень небольших значений m и n. При увеличении мощности множеств лучшим становится решение (б). Затем наилучшим станет решение (в) – до тех пор, пока n не превысит размер внутренней памяти. После этого наилучшим обычно вновь становится решение (б), пока n не вырастет до совсем уж громадных значений. Таким образом, существуют ситуации, в которых сортировка служит хорошей заменой поиску, а поиск – отличной заменой сортировке.

Более сложные задачи поиска зачастую сводятся к простым. Предположим, что ключи представляют собой слова, которые могут быть записаны с небольшими ошибками и необходимо найти запись, невзирая на ошибки в ключах. При этом можно сделать две копии файла, в одном из которых ключи будут расположены в обычном лексикографическом порядке, а в другом – в лексикографическом порядке при чтении слова задом наперёд. Тогда искажённый аргумент поиска будет, вероятно, совпадать до половины (или более) своей длины с ключом некоторой записи в одном из файлов.

Подобные задачи привлекли внимание учёных в связи с вводом в действие систем резервирования билетов на самолёты и других подобных им систем, в которых велика вероятность возникновения ошибки в фамилии из-за плохой слышимости или некаллиграфического почерка. Целью исследований был поиск метода преобразования аргумента в некий код, который позволил бы группировать различные варианты одной фамилии. Далее описан метод *Soundex*, нашедший широкое применение для кодирования фамилий:

Шаг 1. Оставить первую букву фамилии и удалить из неё все буквы *a*, *e*, *h*, *i*, *o*, *u*, *w*, *y* в других позициях;

Шаг 2. Назначить оставшимся буквам (кроме первой) следующие числовые значения:

b, *f*, *p*, *v* $\rightarrow$ 1;	*l* $\rightarrow$ 4;
c, *g*, *j*, *k*, *q*, *s*, *x*, *z* $\rightarrow$ 2;	*m*, *n* $\rightarrow$ 5;
d, *t* $\rightarrow$ 3;	*r* $\rightarrow$ 6.

Шаг 3. Если в исходном слове до выполнения шага 1 две или более буквы с одним и тем же кодом стояли рядом, удалить коды всех букв, кроме первой.

Шаг 4. Преобразовать полученный код в формат «буква, цифра, цифра, цифра» (приписывая необходимое количество нулей справа, если в результате получилось меньше трёх цифр, или отбрасывая лишние цифры, если их больше трёх). ■

Например, фамилии *Euler*, *Gauss*, *Hilbert*, *Knuth*, *Lloyd* и *Lukasiewicz* имеют коды *E*460, *G*200, *H*413, *K*530, *L*300 и *L*222 соответственно. Естественно, разные слова могут иметь одинаковые коды. Так, приведённые выше коды могут быть получены из следующих фамилий: *Ellery*, *Ghosh*, *Heilbronn*, *Kant*, *Liddy* и *Lissajous*. С другой стороны, такие схожие слова, как *Rogers* и *Rodgers*, или *Tchebysheff* и *Chebyshev*, имеют разные коды. Тем не менее, коды *Soundex* существенно увеличивают вероятность нахождения имени по одному из вариантов написания.

Прежде чем перейти к собственно изучению методов поиска, полезно взглянуть на историю данного вопроса. В докомпьютерную эру имелось множество книг с таблицами логарифмов, тригонометрическими и другими таблицами (например «таблицы Брадиса»). И многие математические вычисления фактически были сведены к поиску. Однако с появлением компьютеров с хранимыми программами стало иногда проще вычислить значение $\log x$ или $\cos x$ заново, чем найти его в таблице. В некоторых других ситуациях более выгодным оказывается предварительное вычисление таблиц функций, а в процессе работы – осуществление поиска их значений. Заметим, что в процессе поиска можно использовать более быструю целочисленную арифметику.

Хотя задача сортировки привлекала большое внимание ещё на заре компьютерной эры, алгоритмы поиска оставались в забвении достаточно долгое время. Из-за малой внутренней памяти и наличия для хранения больших файлов только устройств последовательного доступа (наподобие лент) делали поиск либо тривиальным, либо невозможным. Развитие и удешевление памяти с произвольным доступом привело к пониманию того, что проблема поиска важна и интересна.

Рассмотрим далее так называемый *последовательный поиск*, заключающийся в следующем. Начать с начала и продолжать до тех пор, пока не будет найден искомый ключ, и затем остановиться. Эта процедура представляет собой очевидный путь поиска и может служить отправной точкой для рассмотрения множества алгоритмов поиска, поскольку они основаны на последовательном поиске. Заметим, что за простотой последовательного поиска скрывается ряд интересных, несмотря на их простоту, идей.

Алгоритм *S*. (*Последовательный поиск*.)

Дана таблица записей $R_1, R_2, \ldots, R_N$ с ключами $K_1, K_2, \ldots, K_N$ соответственно. Алгоритм предназначен для поиска записи с заданным ключом K. Предполагается, что $N \geq 1$.

***S*1.** [Инициализация.] Присвоить $i \leftarrow 1$.

***S*2.** [Сравнение.] Если $K = K_i$, то успешное завершение алгоритма.

***S*3.** [Продвижение.] Увеличить i на 1.

***S*4.** [Конец файла?] Если $i \leq N$, то вернуться к шагу *S*2. В противном случае алгоритм заканчивается неудачно. ■

Отметим, что этот алгоритм может завершиться успешно (искомый ключ найден) и неудачно (искомый ключ отсутствует), что характерно для большинства алгоритмов поиска.

Анализ алгоритма *S* несложен. Время его выполнения составляет $(5C - 2R + 3)$ единиц времени и зависит от количества сравнений ключей C, а также результата R ($R = 1$ при успешном окончании поиска и $R = 0$ при неудачном). Если при поиске успешно найден ключ $K_i = K$, то полное время работы составляет $(5i + 1)\,u$. Если поиск окажется неудачным, то время работы – $(5N + 3)\,u$. Если все ключи поступают на вход алгоритма с одинаковой вероятностью, то среднее значение C в случае успешного поиска равно

$$\frac{1+2+\ldots+N}{N} = \frac{N+1}{2}\,.$$

Данный алгоритм, несомненно, знаком всем программистам, это не самая лучшая реализация последовательного поиска. При небольшом изменении алгоритм выполняется существенно быстрее (если записей не слишком мало).

Алгоритм *Q*. (*Быстрый последовательный поиск*.)

В данном алгоритме, по сравнению с алгоритмом *S*, имеется дополнительное предположение о наличии фиктивной записи R_{N+1} в конце файла.

***Q*1.** [Инициализация.] Присвоить $i \leftarrow 1$ и $K_{N+1} \leftarrow K$.

***Q*2.** [Сравнение.] Если $K = K_i$, то перейти к шагу *Q*4.

***Q*3.** [Продвижение.] Увеличить i на 1 и вернуться к шагу *Q*2.

***Q*4.** [Конец файла?] Если $i \leq N$, то алгоритм заканчивается успешно; в противном случае – неудачно ($i = N + 1$). ■

Этот алгоритм выполняет поиск за $(4C - 4R + 10)$ единиц времени. Следовательно, выигрыш по сравнению с предыдущим алгоритмом получается в случае $C \geq 6$ для успешного поиска и $N \geq 8$ – для неудачного.

При переходе от алгоритма S к алгоритму Q использован важный ускоряющий принцип – несколько проверок во внутреннем цикле свести к одной проверке по его окончании.

Другим улучшенным алгоритмом поиска можно воспользоваться, если ключи в таблице расположены в порядке возрастания.

Алгоритм *T*. (*Последовательный поиск в упорядоченной таблице*.)

В заданной таблице записей $R_1, R_2, \ldots, R_N$ ключи расположены в порядке возрастания: $K_1 < K_2 < \ldots < K_N$. Алгоритм предназначен для поиска записи с заданным ключом K. Для удобства и ускорения работы алгоритма предполагается наличие фиктивной записи R_{N+1} с ключом $K_{N+1} = \infty > K$.

***T*1.** [Инициализация.] Присвоить $i \leftarrow 1$.

***T*2.** [Сравнение.] Если $K \leq K_i$, то перейти к шагу *T*4.

***T*3.** [Продвижение.] Увеличить i на 1 и вернуться к шагу *T*2.

***T*4.** [Равенство?] Если $K = K_i$, то алгоритм заканчивается успешно. В противном случае – неудачное завершение алгоритма. ■

В предположении, что все входные аргументы-ключи равновероятны, алгоритм по скорости работы в случае успешного поиска аналогичен алгоритму Q. При неудачном поиске отсутствие нужного ключа определяется примерно вдвое быстрее.

Во всех приведённых здесь алгоритмах использовалась запись с индексами для элементов таблиц (она более удобна для описания алгоритмов). Однако все описанные алгоритмы применимы и к другим типам данных, например, к таблицам со связанным представлением данных, поскольку в них данные также расположены последовательно. Например, алгоритм *S* последовательного поиска преобразуется к следующему виду.

Алгоритм *S'*. (*Последовательный поиск в связанном списке*.)

Дана таблица записей $R_1, R_2, \ldots, R_N$. Если P указывает на некоторую запись, то $KEY(P)$ – ключ, $INFO(P)$ – соответствующая ключу информация и $LINK(P)$ – указатель на следующую запись. Переменная *FIRST* указывает на первую запись, а в поле *LINK* последней записи содержится пустая связь Λ. Алгоритм предназначен для поиска записи с заданным ключом K. Предполагается, что $N \geq 1$.

***S'*1.** [Инициализация.] Присвоить $P \leftarrow FIRST$.

***S'*2.** [Сравнение.] Если $K = KEY(P)$, то успешное завершение алгоритма.

***S'*3.** [Продвижение.] Присвоить $P \leftarrow LINK(P)$.

***S'*4.** [Конец файла?] Если $P \neq \Lambda$, то вернуться к шагу *S'*2. В противном случае алгоритм заканчивается неудачно. ■

Упражнения

1. Выполните методом *Soundex* пошаговое преобразование фамилий *Euler*, *Gauss*, *Hilbert*, *Knuth*, *Lloyd* и *Lukasiewicz* в соответствующие им коды.

2. Определите количество сравнений $K = K_i$ и количество сравнений $i \le N$, производимых алгоритмом *S* и алгоритмом *Q* при успешном поиске ключа 765 и неудачном поиске ключа 843 в последовательности ключей 503, 87, 512, 61, 908, 170, 897, 275, 653, 426, 154, 509, 612, 677, 765, 703. Проанализируйте полученные результаты.

3. Установите общее количество сравнений, производимых алгоритмом *Q*, и общее количество сравнений, производимых алгоритмом *T*, при успешном поиске ключей 61, 512, 908 и неудачном поиске ключей 172, 769 в упорядоченной последовательности ключей 61, 87, 154, 170, 275, 426, 503, 509, 512, 612, 653, 677, 703, 765, 897, 908. Проанализируйте полученные результаты.

19. ПОИСК В УПОРЯДОЧЕННОЙ ТАБЛИЦЕ

Далее обсудим методы поиска, основанные на линейном упорядочении ключей (числовом или алфавитном):

$$K_1 < K_2 < \ldots < K_N .$$

При поиске такими методами после сравнения аргумента *K* с ключом K_i из таблицы поиск продолжается одним из трёх путей, в зависимости от того, какое из трёх условий будет выполнено:

– если $K < K_i$, то записи R_i , R_{i+1}, …, R_N исключаются из рассмотрения;

– если $K = K_i$, то поиск завершён;

– если $K > K_i$, то записи R_1, R_2, …, R_i исключаются из рассмотрения.

Рассмотрим метод, заключающийся в следующем. Сначала, сравнив аргумент *K* со средним ключом в таблице, определяют, в какой половине таблицы находится искомый ключ. Затем применяют ту же процедуру к половине таблицы, четверти таблицы и т.д. Максимум за величину порядка lg *N* сравнений находят искомый ключ (либо устанавливают, что его нет в таблице). Такой метод известен как *логарифмический поиск*, или *метод деления пополам*, или *бинарный поиск*.

Хотя основная идея бинарного поиска сравнительно проста, детали его реализации нетривиальны. В одной из наиболее популярных форм реализации метода используются два указателя *l* и *u*, которые обозначают нижнюю и верхнюю границы поиска соответственно.

Алгоритм *B*. (*Бинарный поиск*).

Дана таблица записей $R_1, R_2, \ldots, R_N$, ключи которых расположены в порядке возрастания: $K_1 < K_2 < \ldots < K_N$. Алгоритм используется для поиска в этой таблице заданного аргумента K.

***B*1.** [Инициализация.] Установить $l \leftarrow 1$, $u \leftarrow N$.

***B*2.** [Получение середины.] (На этом шаге известно, что если K имеется в таблице, то справедливо условие $K_l \le K \le K_u$.) Если $u < l$, алгоритм завершается неудачно; в противном случае необходимо присвоить $i \leftarrow \lfloor (l+u)/2 \rfloor$, чтобы i соответствовало примерно середине рассматриваемой части таблицы.

***B*3.** [Сравнение.] Если $K < K_i$, то перейти к шагу *B*4; если $K > K_i$, то перейти к шагу *B*5; если $K = K_i$, то алгоритм завершается успешно.

***B*4.** [Изменение u.] Присвоить $u \leftarrow i - 1$ и вернуться к шагу *B*2.

***B*5.** [Изменение l.] Присвоить $l \leftarrow i + 1$ и вернуться к шагу *B*2. ■

В таблицах 5 и 6 показаны два случая проведения бинарного поиска. В первом случае осуществляется поиск числа 653, имеющегося в списке, а во втором – числа 400, которое в списке отсутствует. Скобки показывают положение указателей l и u, а подчёркнутое число – значение K_i. В обоих случаях поиск завершается после выполнения четырёх сравнений.

Таблица 5

[61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
61	87	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
61	87	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908
61	87	154	170	275	426	503	509	512	612	[653]	677	703	765	897	908

Таблица 6

[61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
[61	87	154	170	275	426	503]	509	512	612	653	677	703	765	897	908
61	87	154	170	[275	426	503]	509	512	612	653	677	703	765	897	908
61	87	154	170	[275]	426	503	509	512	612	653	677	703	765	897	908
61	87	154	170	275]	[426	503	509	512	612	653	677	703	765	897	908

Для лучшего понимания работы алгоритма *B* представим описанную процедуру поиска в виде бинарного дерева принятия решений, изображённого на рис. 38 для $N = 16$.

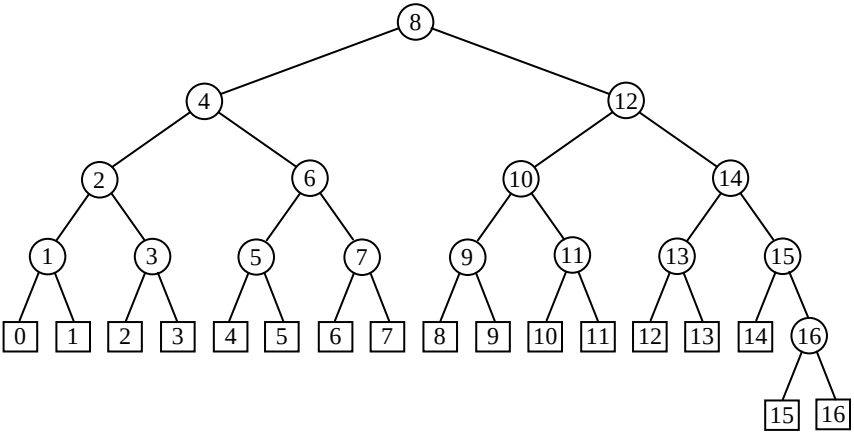
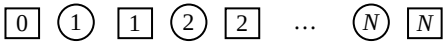


Рис. 38. Дерево сравнений бинарного поиска

Сравнение K и K_8 на рисунке представлено корневым узлом номер 8. Если $K < K_8$, то алгоритм переходит в левое поддерево и сравниваются K и K_4. В противном случае, если $K > K_8$, то алгоритм переходит в правое поддерево и сравниваются K и K_{12}. В противном случае при равенстве K и K_8 алгоритм завершается успешно. Вообще при успешном завершении поиска алгоритм останавливается в одном из внутренних «круглых» узлов. Например, в круглом узле номер 3 алгоритм остановится при $K = K_3$. Неудачный же поиск приведёт в один из внешних «квадратных» узлов, пронумерованных от 0 до N. Например, квадратный узел номер 6 будет достигнут при $K_6 < K < K_7$, квадратный узел номер 7 – при $K_7 < K < K_8$.

Бинарное дерево, соответствующее бинарному поиску в таблице из N записей, может быть построено следующим образом. Если $N = 0$, то дерево представляет собой просто квадратный узел номер 0. В противном случае корневым узлом дерева будет круглый узел с номером $\lceil N / 2 \rceil$. При этом левое поддерево представляет собой бинарное дерево с $(\lceil N / 2 \rceil - 1)$ круглыми узлами, а правое поддерево – бинарное дерево с $\lfloor N / 2 \rfloor$ круглыми узлами. Отметим, что все числа в узлах правого поддерева увеличены на $\lceil N / 2 \rceil$ по сравнению с числами в соответствующих узлах левого поддерева.

Точно так в виде бинарного дерева с N узлами, в котором все узлы пронумерованы числами от 1 до N, может быть представлен *любой* алгоритм поиска в упорядоченной таблице длиной N (если только алгоритм не выполняет излишние сравнения). Верно и обратное – любое бинарное дерево соответствует некоторому методу поиска в упорядоченной таблице. При этом достаточно просто пометить узлы



в симметричном порядке слева направо.

Теорема В. *Для* $2^{k-1} \le N < 2^k$ *успешный поиск по алгоритму B требует минимум одно и максимум k сравнений*, а *неудачный поиск при* $2^{k-1} \le N < 2^k - 1$ *требует* $(k-1)$ *или k сравнений*. ■

Заметим, что никакой другой метод поиска, основанный на сравнении ключей, не может превзойти этот результат. Среднее время работы алгоритма *B* в предположении равновероятных исходов поиска составляет приблизительно $(18 \lg N - 16)$ *u* для успешного и $(18 \lg N + 12)$ *u* для неудачного поиска.

Упражнения

1. Постройте бинарное дерево (подобное изображённому на рис. 38), которое соответствует бинарному поиску в упорядоченной таблице из десяти записей с ключами: 61, 87, 154, 170, 275, 426, 503, 509, 512 и 612. Проследите за поиском по алгоритму *B* ключей 50, 61, 72, 87, 112, 154, 163, 170 и 275, отслеживая при этом соответствующие поиску пути в бинарном дереве.

20. ПОИСК ПО БИНАРНОМУ ДЕРЕВУ

Из предыдущего параграфа видно, что неявная структура бинарного дерева облегчает понимание бинарного поиска. Однако рассмотренный там метод годится в основном для таблиц фиксированного размера, поскольку последовательное расположение записей делает вставку и удаление записей весьма трудоёмкими. В случае когда таблица динамически изменяется, можно потратить на её постоянное упорядочение куда больше времени, чем сэкономить на бинарном поиске.

Явное использование структуры бинарного дерева для хранения таблицы позволяет быстро вставлять и удалять записи, а также эффективно выполнять поиск. Заметим, что при этом требуется два дополнительных поля ссылок в каждой записи таблицы.

Технологии поиска в растущей таблице часто называют *алгоритмами таблиц символов*, так как ассемблеры, компиляторы и другие системные программы обычно применяют их для хранения пользовательских символов. Например, ключом записи в компиляторе может быть символьный идентификатор переменной в некоторой программе на языке *FORTRAN* или *C*, а остальная часть записи может содержать информацию о типе переменной и её адресе. Алгоритм поиска со вставкой по дереву, который будет рассмотрен ниже, достаточно эффективен в качестве алгоритма таблиц символов, особенно в приложениях, в которых может оказаться нежелательным вывод списка символов в алфавитном порядке.

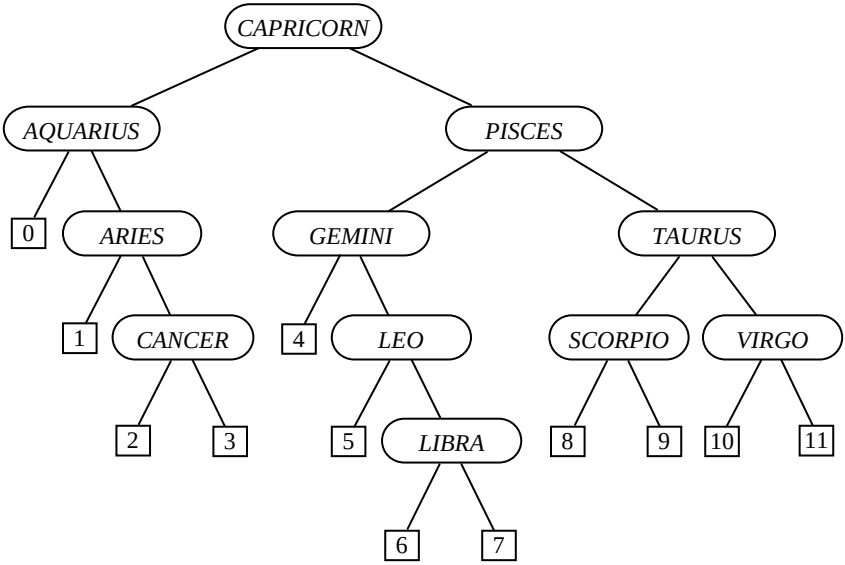


Рис. 39. Бинарное дерево поиска

На рисунке 39 показано бинарное дерево поиска, содержащее названия одиннадцати знаков зодиака[*]. Если приступить к поиску двенадцатого имени, *SAGITTARIUS*, начиная от корня дерева, то выяснится, что оно больше (лексикографически), чем *CAPRICORN*, и нужно идти вправо. Оно больше, чем *PISCES*, и необходимо опять идти вправо. Оно меньше, чем *TAURUS*, и нужно идти влево. Оно меньше, чем *SCORPIO*, и мы попадаем во внешний узел номер 8. Поиск оказался неудачным, и теперь можно вставить имя *SAGITTARIUS* в то место, где завершился поиск, т.е. на место внешнего узла номер 8. Таким образом, таблица может расширяться без перемещения уже существующих записей.

Заметим, что дерево на рис. 39 было получено путём последовательной вставки в пустое дерево ключей *CAPRICORN*, *AQUARIUS*, *PISCES*, *ARIES*, *TAURUS*, *GEMINI*, *CANCER*, *LEO*, *VIRGO*, *LIBRA*, *SCORPIO* в указанном порядке. В этом дереве ключи левого поддерева меньше (в лексикографическом порядке), чем *CAPRICORN*, а все ключи правого поддерева – больше его. То же самое справедливо для левого и правого поддеревьев любого узла. Отсюда следует, что при обходе дерева в симметричном порядке (левое поддерево, корень, правое поддерево)

[*] Список знаков зодиака, упорядоченный по месяцам таков: Козерог (*Capricorn*), Водолей (*Aquarius*), Рыбы (*Pisces*), Овен (*Aries*), Телец (*Taurus*), Близнецы (*Gemini*), Рак (*Cancer*), Лев (*Leo*), Дева (*Virgo*), Весы (*Libra*), Скорпион (*Scorpio*), Стрелец (*Sagittarius*).

будет получен список ключей в алфавитном порядке: *AQUARIUS, ARIES, CANCER, CAPRICORN, GEMINI, LEO, …, VIRGO*.

Ниже приводится подробное описание процесса поиска со вставкой.

Алгоритм *T*. (*Поиск по дереву со вставкой*.)

Алгоритм предназначен для поиска заданного аргумента K в таблице записей, образующей бинарное дерево. Если ключ K в таблице отсутствует, то новый узел, содержащий этот ключ, вставляется в дерево в надлежащее место. Предполагается, что узлы дерева содержат по крайней мере следующие поля:

$KEY(P)$ – ключ, хранящийся в узле $NODE(P)$;
$LLINK(P)$ – указатель на левое поддерево узла $NODE(P)$;
$RLINK(P)$ – указатель на правое поддерево узла $NODE(P)$.

Пустые поддеревья (внешние узлы на рис. 39) представляются пустыми указателями Λ. Переменная $ROOT$ указывает на корень дерева. Для удобства полагаем, что дерево не пусто, т.е. $ROOT \neq \Lambda$, так как при $ROOT = \Lambda$ необходимые операции тривиальны.

T1. [Инициализация.] Присвоить $P \leftarrow ROOT$. (Переменная-указатель P будет перемещаться вниз по дереву.)

T2. [Сравнение.] Если $K < KEY(P)$, то перейти к шагу $T3$; если $K > KEY(P)$, перейти к шагу $T4$; и если $K = KEY(P)$, поиск успешно завершён.

T3. [Перемещение влево.] Если $LLINK(P) \neq \Lambda$, то присвоить $P \leftarrow LLINK(P)$ и вернуться к шагу $T2$; в противном случае перейти к шагу $T5$.

T4. [Перемещение вправо.] Если $RLINK(P) \neq \Lambda$, то присвоить $P \leftarrow RLINK(P)$ и вернуться к шагу $T2$.

T5. [Вставка в дерево.] (Поиск оказался неудачным и требуется поместить ключ K в дерево.) $Q \Leftarrow AVAIL$ (выделить новый узел). Присвоить $KEY(Q) \leftarrow K$, $LLINK(Q) \leftarrow RLINK(Q) \leftarrow \Lambda$. (На практике следует инициализировать и другие поля нового узла.) Если $K < KEY(P)$, то выполнить присваивание $LLINK(P) \leftarrow Q$; в противном случае – присваивание $RLINK(P) \leftarrow Q$. (Теперь можно присвоить $P \leftarrow Q$ и считать поиск успешно завершившимся.) ■

Среднее время работы алгоритма составляет величину, равную примерно $(7.5C - 2.5S + 4)u$, где C – количество выполненных сравнений; $S = 1$ при успешном и 0 при неудачном поиске. Следовательно, это лучшее значение, чем при бинарном поиске с неявными деревьями.

Упражнения

1. Выполните алгоритм T в условиях последовательной вставки в пустое дерево ключей *JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER* и постройте бинарное дерево (подобное изображённому на рис. 39), которое при этом получится.

21. СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

Рассмотренный в предыдущем параграфе алгоритм поиска со вставкой в дерево даёт хорошие результаты при использовании случайных входных данных, но существует неприятная возможность того, что при этом будет построено вырожденное дерево (линейный список). Для поддержания бинарного дерева в так называемом *хорошем состоянии* его можно реорганизовывать на некоторых этапах построения.

Красивое решение задачи реорганизации бинарного дерева было открыто в 1962 г. советскими математиками Г.М. Адельсон-Вельским и Е.М. Ландисом. Их метод требует двух дополнительных битов на каждый узел дерева и никогда не использует более $O(\log N)$ операций для поиска по дереву или вставки нового элемента. Предложенный подход также даёт общую технологию представления линейных списков длиной N таким образом, что любая из следующих операций может быть выполнена за время порядка $\log N$:

а) поиск элемента по заданному ключу;
б) поиск k-го элемента по заданному k;
в) вставка элемента в определённое место;
г) удаление определённого элемента.

При последовательном расположении в памяти элементов линейных списков операции (а) и (б) выполняются очень эффективно, в то время как операции (в) и (г) требуют порядка N шагов. При использовании связанных элементов, напротив, эффективно будут выполняться операции (в) и (г), а операции (а) и (б) потребуют порядка N шагов. Представление линейных списков в виде дерева позволяет выполнить *каждую из четырёх* операций за $O(\log N)$ шагов. При этом можно более или менее эффективно выполнять и другие операции, например сцепление списка из M элементов со списком из N элементов за $O(\log(M + N))$ шагов.

Метод, предоставляющий все эти преимущества, будем называть сбалансированными деревьями (*AVL-деревьями* – по первым буквам фамилий авторов).

Высотой дерева называется его максимальный уровень, т.е. длина самого длинного пути от корня к внешнему узлу (листу). Бинарное дерево называют *сбалансированным*, если высота левого поддерева любого узла отличается не более чем на ±1 от высоты правого поддерева. На рисунке 40 изображено сбалансированное дерево высотой 5 с 17 внутренними узлами. *Фактор сбалансированности* обозначен внутри каждого узла знаками «+», «•» и «–» в соответствии с величиной разности между высотами правого и левого поддеревьев (+1, 0 или –1).

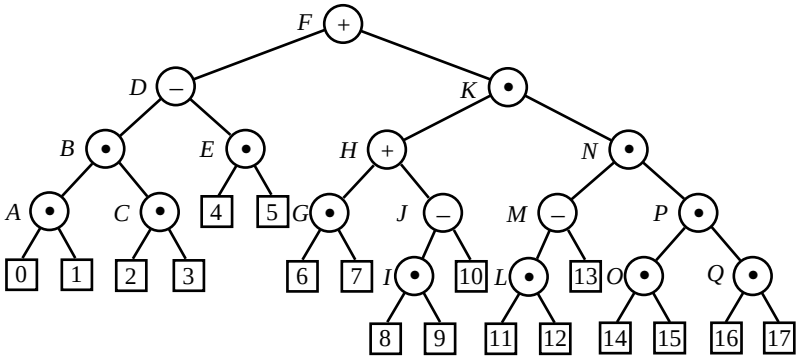


Рис. 40. Сбалансированное бинарное дерево

В качестве примера *несбалансированного* бинарного дерева может служить дерево знаков зодиака (см. рис. 39), поскольку для узлов *AQUARIUS* и *GEMINI* нарушено условие для величины разности высот поддеревьев.

Подчеркнём, что сбалансированные бинарные деревья представляют собой компромисс между *оптимальными* бинарными деревьями (все внешние узлы которых должны размещаться на двух соседних уровнях) и произвольными бинарными деревьями. А путь поиска по сбалансированному дереву не превышает пути поиска по оптимальному дереву более чем на 45%.

Теорема (Адельсон-Вельского и Ландиса). *Высота сбалансированного дерева с N внутренними узлами ограничена значениями* $\lg(N+1)$ *и* $1.4405\lg(N+2)-0.3277$.

Рассмотрим, что произойдёт при вставке нового узла в сбалансированное дерево с использованием алгоритма *T* (поиска по дереву со вставкой) из предыдущего параграфа. Дерево, представленное на рис. 40, останется сбалансированным, если новый узел займёт место одного из внешних узлов 4, 5, 6, 7, 10 или 13, а в других случаях потребуется определённая реорганизации бинарного дерева.

Вообще необходимость реорганизации дерева возникает, когда в нём имеется узел с фактором сбалансированности +1 и после вставки нового узла его правое поддерево становится выше, или в случае, когда фактор сбалансированности узла равен –1 и выше становится его левое поддерево.

На рисунке 41 большие прямоугольники α, β, γ и δ представляют поддеревья с соответствующими высотами. Случай *а*) имеет место, когда новый элемент увеличивает высоту правого поддерева узла *B* с *h* до *h* + 1, а случай *б*) – когда увеличивается высота левого поддерева узла *B*. В по-

следнем случае либо $h = 0$ (X представляет собой новый узел), либо узел X имеет два поддерева с высотами $(h - 1, h)$ или $(h, h - 1)$. Отметим, что два других случая потери сбалансированности бинарного дерева могут быть получены из указанных на рис. 41 деревьев при зеркальном отражении относительно вертикальной оси.

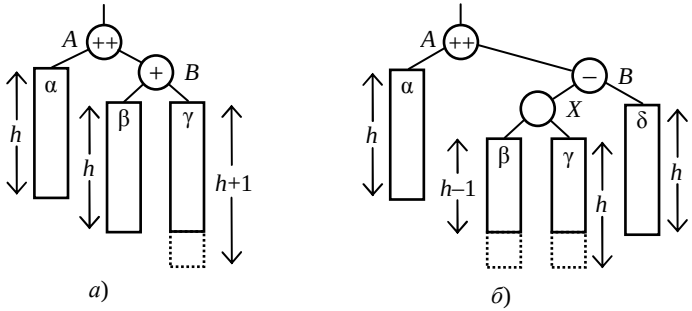


Рис. 41. Два случая потери сбалансированности

Выполнив простые преобразования, как показано на рис. 42, можно восстановить баланс в обоих случаях. При этом симметричный порядок узлов дерева сохранится.

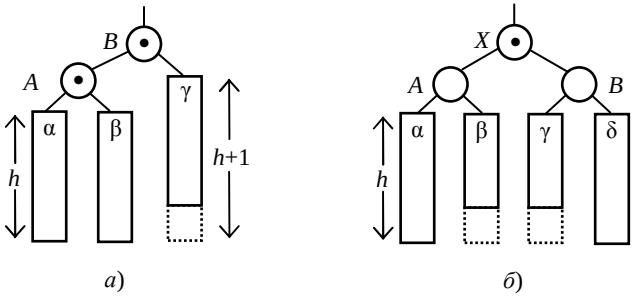


Рис. 42. Два случая восстановления сбалансированности

В случае *а*) дерево просто «поворачивают» налево, присоединяя поддерево β к узлу A вместо B. Такое преобразование подобно применению ассоциативного закона к алгебраической формуле, заменяющему α(βγ) на (αβ)γ. В случае *б*) используется двойной поворот: сначала (X, B) поворачивают направо и затем (A, X) – налево. В обоих случаях в дереве требуется изменить лишь несколько ссылок. Кроме того, новые деревья имеют высоту $h + 2$, точно равную высоте, которая была до вставки. Следовательно, остаток дерева (если таковой имеется), который изначально находился над узлом A, всегда остаётся сбалансированным.

На рисунке 43 изображено дерево, представленное на рис. 40, после вставки нового ключа *R* и выполнения балансировки. Например, если вставить новый *R* узел в позицию 17 дерева, представленного на рис. 40, то после однократного поворота получится сбалансированное дерево, изображённое на рис. 43. Заметим, что при этом изменились некоторые факторы сбалансированности.

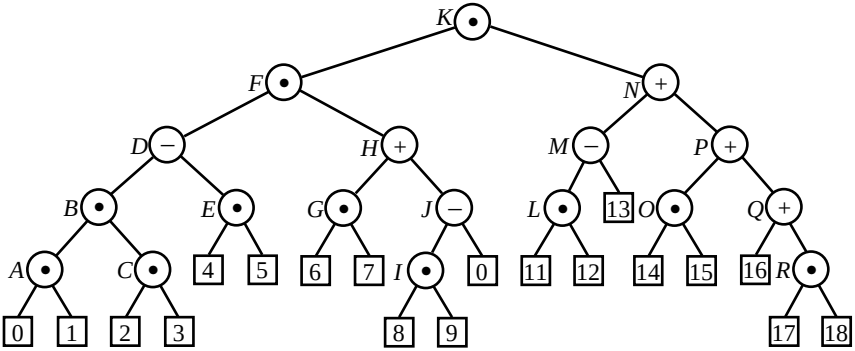


Рис. 43. Сбалансированное дерево после вставки нового ключа *R*

Детали процедуры вставки могут быть разработаны различными способами. При этом кажется, что невозможно избежать использования вспомогательного стека, поскольку требуется запоминать узлы, затронутые при вставке. Однако описанный ниже алгоритм позволяет обойтись без стека.

Алгоритм *A*. (*Поиск со вставкой по сбалансированному дереву*.)

Дана таблица записей в форме сбалансированного бинарного дерева, описанного выше. Алгоритм производит поиск данного аргумента *K*. Если *K* отсутствует в таблице, то в соответствующее место дерева вставляется узел, в котором содержится *K*, и дерево при необходимости балансируется.

Предполагается, что, как и в алгоритме *T* из предыдущего параграфа, узлы дерева имеют поля *KEY*, *LLINK* и *RLINK*. Кроме того, имеется новое поле

$$B(P) - \text{ фактор сбалансированности узла } NODE(P),$$

представляющий разность высот его правого и левого поддеревьев. В нём всегда содержится только одно из трёх значений: +1, 0 или –1. На вершине дерева по адресу *HEAD* расположен специальный узел, поле *RLINK* которого указывает на корень дерева, а *LLINK*(*HEAD*) используется для хранения полной высоты дерева (знание высоты не является необходимым для этого алгоритма, однако полезно при конкатенации, которая

будет обсуждаться ниже). Также предполагается, что дерево *непустое*, т.е. $RLINK(HEAD) \neq \Lambda$.

Для удобства описания в алгоритме используется обозначение $LINK(a, P)$ для $LLINK(P)$ при $a = -1$ и для $RLINK(P)$ при $a = +1$.

***A*1.** [Инициализация.] Присвоить $T \leftarrow HEAD$, $S \leftarrow P \leftarrow RLINK(HEAD)$. (Переменная-указатель P будет двигаться вниз по дереву; S будет указывать на место, где может потребоваться балансировка, а T – на родительский по отношению к S узел.)

***A*2.** [Сравнение.] Если $K < KEY(P)$, то перейти к шагу $A3$; если $K > KEY(P)$, перейти к шагу $A4$; если $K = KEY(P)$, поиск успешно завершён.

***A*3.** [Перемещение влево.] Присвоить $Q \leftarrow LLINK(P)$. Если $Q = \Lambda$, то выполнить $Q \Leftarrow AVAIL$ и $LLINK(P) \leftarrow Q$ и перейти к шагу $A5$. В противном случае, если $B(Q) \neq 0$, то присвоить $T \leftarrow P$ и $S \leftarrow Q$. И, наконец, присвоить $P \leftarrow Q$ и вернуться к шагу $A2$.

***A*4.** [Перемещение вправо.] Присвоить $Q \leftarrow RLINK(P)$. Если $Q = \Lambda$, то выполнить $Q \Leftarrow AVAIL$, $RLINK(P) \leftarrow Q$ и перейти к шагу $A5$. В противном случае, если $B(Q) \neq 0$, то присвоить $T \leftarrow P$ и $S \leftarrow Q$. И, наконец, присвоить $P \leftarrow Q$ и вернуться к шагу $A2$.

***A*5.** [Вставка.] (Только что новый узел $NODE(Q)$ вставлен в дерево и теперь следует инициализировать его поля.) Установить $KEY(Q) \leftarrow K$, $LLINK(Q) \leftarrow RLINK(Q) \leftarrow \Lambda$ и $B(Q) \leftarrow 0$.

***A*6.** [Корректировка факторов сбалансированности.] (В настоящий момент следует изменить факторы сбалансированности узлов между S и Q с 0 на ±1.) Если $K < KEY(S)$, то присвоить $a \leftarrow -1$; в противном случае присвоить $a \leftarrow +1$. Затем присвоить $R \leftarrow P \leftarrow LINK(a, S)$ и при необходимости повторить следующую операцию несколько раз, пока P не станет равным Q: если $K < KEY(P)$, то присвоить $B(P) \leftarrow -1$ и $P \leftarrow LLINK(P)$; если $K > KEY(P)$, то присвоить $B(P) \leftarrow +1$ и $P \leftarrow RLINK(P)$. (Если $K = KEY(P)$, то $P = Q$ и следует перейти к следующему шагу.)

***A*7.** [Балансировка.] На этом шаге возможны несколько вариантов.

1) если $B(S) = 0$ (дерево стало выше), то присвоить $B(S) \leftarrow a$, $LLINK(HEAD) \leftarrow LLINK(HEAD) + 1$ и прекратить выполнение алгоритма;

2) если $B(S) = -a$ (дерево стало более сбалансированным), то присвоить $B(S) \leftarrow 0$ и прекратить выполнение алгоритма;

3) если $B(S) = a$ (дерево разбалансировано), то перейти к шагу $A8$ при $B(R) = a$ или к шагу $A9$ при $B(R) = -a$.

(Случай (3) соответствует ситуации (см. рис. 41) при $a = +1$. При этом S и R указывают на узлы A к B соответственно, $LINK(-a, S)$ указывает на α и т.д.)

***A*8.** [Однократный поворот.] Установить $P \leftarrow R$, $LINK(a, S) \leftarrow LINK(-a, R)$, $LINK(-a, R) \leftarrow S$, $B(S) \leftarrow B(R) \leftarrow 0$. Перейти к шагу $A10$.

***A*9.** [Двукратный поворот.] Установить $P \leftarrow LINK(-a, R)$, $LINK(-a, R) \leftarrow LINK(a, P)$, $LINK(a, P) \leftarrow R$, $LINK(a, S) \leftarrow LINK(-a,P)$, $LINK(-a, P) \leftarrow S$; присвоить сначала

$$(B(S), B(R)) \leftarrow \begin{cases} (-a, 0), & \text{если } B(P) = a; \\ (0, 0), & \text{если } B(P) = 0; \\ (0, a), & \text{если } B(P) = -a; \end{cases}$$

а затем $B(P) \leftarrow 0$.

***A*10.** [Последний штрих.] (Балансирующее преобразование (рис. 41) в (рис. 42) завершено. P указывает на корень нового поддерева, а T – на родительский по отношению к корню старого поддерева узел S.) Если $S = RLINK(T)$, то присвоить $RLINK(T) \leftarrow P$; в противном случае присвоить $LLINK(T) \leftarrow P$. ■

Этот алгоритм достаточно длинный, однако разделяется на три простые части: на шагах $A1 \ldots A4$ осуществляется поиск, на шагах $A5 \ldots A7$ выполняется вставка нового узла и на шагах $A8 \ldots A10$ при необходимости балансируется дерево. Отметим, что алгоритм может использоваться и для прошитых деревьев.

Известно, что для этого алгоритма требуется около $C \log N$ единиц времени, где C – некоторая константа.

Упражнения

1. Выполните алгоритм A в условиях вставки в корень пустого дерева ключа *JANUARY* и дальнейшей последовательной вставки ключей *FEBRUARY*, *MARCH*, *APRIL*, *MAY*, *JUNE*, *JULY*, *AUGUST*, *SEPTEMBER*, *OCTOBER*, *NOVEMBER*, *DECEMBER* и постройте сбалансированное бинарное дерево, которое при этом получится.

22. ХЕШИРОВАНИЕ

До сих пор были рассмотрены методы поиска, основанные на сравнении данного аргумента K с ключами в таблице. Другой путь заключается в так называемом *хешировании*, когда отказываются от поиска по данным и выполняют арифметические действия над K, позволяющие вычислить некоторую *хеш-функцию* $h(K)$. Последняя укажет адрес в таблице, где хранится K и связанная с ним информация.

Заметим, что поиск таких функций $h(K)$ является довольно сложной задачей, поскольку функции, дающие неповторяющиеся значения, встречаются на удивление редко. Например, в соответствии со знаменитым «парадоксом дней рождения» получается, что в компании из двадцати трёх человек более вероятно совпадение дней рождения у двух из них,

чем то, что у всех дни рождения окажутся различными. Иными словами, если выбрать случайную функцию, отображающую 23 ключа в таблицу размером 365, вероятность того, что никакие два ключа не будут отображены в одно и то же место таблицы, составляет 0.4927, т.е. менее половины.

Событие, заключающееся в том, что для двух различных ключей K_i и K_j имеет место одинаковое значение хеш-функции: $h(K_i) = h(K_j)$, называется *коллизией*. Поэтому для использования хеширования программист должен решить две практически независимые задачи – выбрать хеш-функцию и способ разрешения коллизий.

Далее предполагается, что хеш-функция имеет не более M различных значений, причём для любого ключа K справедливо соотношение

$$0 \le h(K) < M. \tag{58}$$

Многочисленные тесты показали хорошую работу хеш-функции, представляющей собой остаток от деления на M:

$$h(K) = K \bmod M. \tag{59}$$

Самым очевидным путём решения проблемы возникающих коллизий является поддержка M связных списков, по одному для каждого возможного значения хеш-кода. При этом следует иметь M головных узлов списков, пронумерованных, например, от 1 до M, а каждый узел связанного списка должен иметь поле *LINK* для размещения связи со следующим узлом. После хеширования ключа просто выполняется последовательный поиск в списке с номером $h(K) + 1$.

На рисунке 44 представлена простая схема с цепочками при $M = 9$ для последовательности из ключей $K = 830, 585, 516, 432, 571, 161$ и 433, имеющих хеш-коды $h(K) + 1 = 3, 1, 4, 1, 5, 9$ и 2 соответственно. В первом списке содержится два элемента, три списка пусты.

В общем случае, если имеется N ключей и M списков, средний размер списка будет равен N/M, следовательно, хеширование приводит к уменьшению среднего количества работы, необходимой для последовательного поиска, примерно в M раз.

Для повышения быстродействия желательны большие значения M, однако в этом случае слишком многие списки будут пусты и будут зря расходовать память на N записей и $M + N$ ссылок. Иногда можно совершать проход по всем данным для поиска заголовков списков, которые будут использоваться, а затем при втором проходе вставлять «переполняющие» записи в пустые места. Однако зачастую это непрактично или невозможно, поэтому хотелось бы иметь метод, работающий с каждой записью только раз при помещении её в систему. Следующий алгоритм позволяет быстро решить эту задачу.

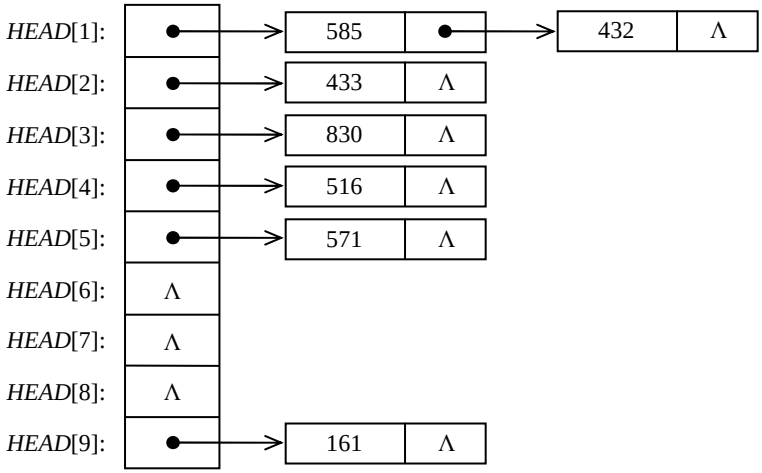


Рис. 44. Раздельные цепочки

Алгоритм *C*. (*Поиск и вставка в хеш-таблице с цепочками*.)

Этот алгоритм ищет заданный ключ K в таблице из M узлов. Если ключ K в таблице отсутствует, а таблица не заполнена, он вставляется в таблицу.

Узлы таблицы обозначаются через $TABLE[i]$, $0 \le i \le M$, и могут быть двух различных типов – *пустыми* и *занятыми*. В занятом узле содержатся поле ключа $KEY[i]$, поле ссылки $LINK[i]$ и, возможно, другие поля.

Алгоритм использует хеш-функцию $h(K)$. Применяется также вспомогательная переменная R для упрощения поиска пустых мест. Если таблица пуста, то $R = M + 1$. По мере выполнения вставок всегда будет оставаться истинным утверждение, что узлы $TABLE[j]$ заняты для всех j в диапазоне $R \le j \le M$. По соглашению узел $TABLE[0]$ всегда пуст.

***C*1.** [Хеширование.] Присвоить $i \leftarrow h(K) + 1$. (Теперь $1 \le i \le M$.)

***C*2.** [Есть ли список?] Если узел $TABLE[i]$ пуст, то перейти к шагу *C*6 (В противном случае $TABLE[i]$ занят и необходимо приступить к поиску в списке, начинающемся в этом узле.)

***C*3.** [Сравнение.] Если $K = KEY[i]$, то алгоритм успешно завершается.

***C*4.** [Переход к следующему.] Если $LINK[i] \ne 0$, то присвоить $i \leftarrow LINK[i]$ и вернуться к шагу *C*3.

***C*5.** [Поиск пустого узла.] (Поиск был неудачным, и необходимо найти пустое место в таблице.) Уменьшать R один или более раз, пока не будет найдено такое значение, при котором узел $TABLE[R]$ будет пуст. Если $R = 0$, то алгоритм завершается в связи с переполнением (свободных узлов больше нет); в противном случае присвоить $LINK[i] \leftarrow R$, $i \leftarrow R$.

C6. [Вставка нового ключа.] Пометить узел *TABLE*[*i*] как занятый и присвоить *KEY*[*i*] ← *K*, *LINK*[*i*] ← 0. ■

Алгоритм позволяет объединить несколько списков, поэтому записи не нужно перемещать после вставки в таблицу. Например, на рис. 45 ключ 161 попадает в список, содержащий 585 и 432, поскольку последний уже был вставлен в позицию 9.

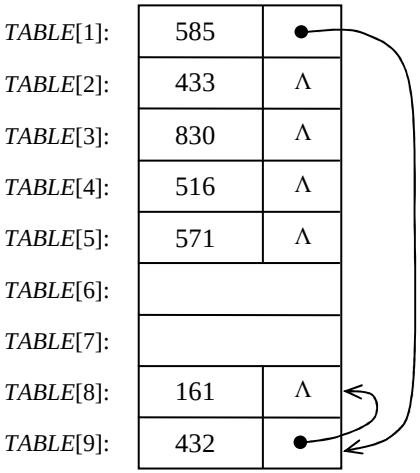


Рис. 45. Сросшиеся цепочки

Другой путь решения проблемы, связанной с коллизиями, состоит в том, чтобы полностью отказаться от ссылок, просто просматривая различные записи таблицы одну за одной до тех пор, пока не будет найден ключ *K* или пустая позиция. Идея заключается в формулировании правила, согласно которому по данному ключу *K* определяется «пробная последовательность», т.е. последовательность позиций таблицы, которые должны быть просмотрены при вставке или поиске *K*. Если при поиске *K* согласно определённой этим ключом последовательности встречается пустая позиция, можно сделать вывод о том, что *K* в таблице отсутствует (поскольку та же последовательность проверок должна выполняться при каждом поиске *K*). Этот общий класс методов назван *открытой адресацией*.

Простейшая схема открытой адресации, известная как *линейное исследование*, использует циклическую последовательность проверок:

$$h(K),\ h(K)-1,\ ...,\ 0,\ M-1,\ M-2\ ,\ ...,\ h(K)+1 \qquad (60)$$

и описывается следующим образом.

Алгоритм L. (*Линейное исследование и вставка*.)

Этот алгоритм выполняет поиск данного ключа K в таблице с M узлами. Если K отсутствует в таблице и таблица не полна, ключ K будет вставлен в таблицу.

Узлы таблицы обозначаются как $TABLE[i]$, $0 \le i < M$, и могут быть двух типов – *пустыми* и *занятыми*. В занятых узлах содержатся ключи $KEY[i]$ и, возможно, другие поля. Вспомогательная переменная N используется для отслеживания количества занятых узлов и увеличивается на 1 при каждой вставке нового ключа.

Алгоритм использует хеш-функцию $h(K)$ и линейную последовательность проб (60) для адресации таблицы.

L1. [Хеширование.] Присвоить $i \leftarrow h(K)$. (Теперь $0 \le i < M$.)

L2. [Сравнение.] Если узел $TABLE[i]$ пуст, перейти к шагу $L4$. В противном случае, если $KEY[i] = K$, алгоритм успешно завершается.

L3. [Переход к следующему.] Присвоить $i \leftarrow i - 1$. Если теперь $i < 0$, присвоить $i \leftarrow i + M$. Вернуться к шагу $L2$.

L4. [Вставка.] (Поиск оказался неудачным.) Если $N = M - 1$, алгоритм завершается в связи с переполнением. В противном случае $N \leftarrow N + 1$, пометить узел $TABLE[i]$ как занятый и присвоить $KEY[i] \leftarrow K$. ■

0	585
1	433
2	830
3	516
4	571
5	
6	
7	161
8	432

Рис. 46. Линейная открытая адресация

На рисунке 46 показано, что происходит при вставке семи ключей K = 830, 585, 516, 432, 571, 161 и 433 нашего примера с хеш-кодами $h(K)$ = 2, 0, 3, 0, 4, 8 и 1 соответственно. При этом ключи 432 и 161 смещены со своих начальных позиций $h(K)$.

Упражнения

1. Выполните с применением алгоритма C, использующего хеш-функцию $h(K) = K \bmod M$, последовательную вставку ключей 443, 564, 434, 704, 157, 541, 240 в пустую хеш-таблицу из $M = 9$ узлов.

2. Примените алгоритм L, использующий хеш-функцию $h(K) = K \bmod M$, для последовательной вставки ключей 601, 372, 268, 365, 622, 609, 133 в пустую хеш-таблицу из $M = 9$ узлов.

ОТВЕТЫ К УПРАЖНЕНИЯМ

§ 1

1. Да, если всегда включать все элементы в один из двух концов.

2. **а)** Да, если выполнить последовательность операций SSSXXSSXSXXX; **б)** Нет (может получиться только 154632), поскольку 2 может предшествовать 3 тогда и только тогда, когда 2 удаляется из стека до занесения в него 3.

3. Допустимой последовательностью из S и X является та, в которой при просмотре её слева направо текущее число вхождений X никогда не превышает текущего числа вхождений S.

Две различные допустимые последовательности должны дать разный результат по следующей причине. Если две последовательности совпадают до некоторого момента, а затем в одной из них встречается S и в другой – X, то вторая последовательность выведет символ, который нельзя будет вывести раньше символа, посланного в стек операцией S в первой последовательности. Например, SSSSXXXX → 4321, SSSXSXXX → 3421.

4. **а)** $a_2 = \binom{2\cdot 2}{2} - \binom{2\cdot 2}{2-1} = \binom{4}{2} - \binom{4}{1} = \frac{4!}{2!\,2!} - \frac{4!}{1!\,3!} = \frac{1\cdot 2\cdot 3\cdot 4}{1\cdot 2\cdot 1\cdot 2} - \frac{1\cdot 2\cdot 3\cdot 4}{1\cdot 1\cdot 2\cdot 3} = 3\cdot 2 - 4 = 6 - 4 = 2.$

Всевозможные последовательности: SSXX, XXSS, SXSX, XSXS, SXXS, XSSX.

Недопустимые последовательности: XXSS, XSXS, SXXS, XSSX.

Допустимые последовательности: SSXX, SXSX.

В результате реализации этих последовательностей будут получены перестановки: 21, 12;

б) $a_3 = \binom{2\cdot 3}{3} - \binom{2\cdot 3}{3-1} = \binom{6}{3} - \binom{6}{2} = \frac{6!}{3!\,3!} - \frac{6!}{2!\,4!} = \frac{1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6}{1\cdot 2\cdot 3\cdot 1\cdot 2\cdot 3} - \frac{1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6}{1\cdot 2\cdot 1\cdot 2\cdot 3\cdot 4} = 20 - 15 = 5.$

Допустимые последовательности SSSXXX, SSXSXX, SSXXSX, SXSSXX, SXSXSX.

В результате реализации этих последовательностей будут получены перестановки: 321, 231, 213, 132, 123.

5. Перестановку 154623 получить нельзя, так как существуют индексы $i = 2$, $j = 5$ и $k = 6$, при которых $p_j = 2 < p_k = 3 < p_i = 5$. Перестановку 325641 получить можно, поскольку не существует таких индексов i, j и k, при которых $i < j < k$ и $p_j < p_k < p_i$.

6. Только тривиальную перестановку 1, 2, …, n.

7. Если в деке с ограниченным входом первым на выходе появляется величина n, то это означает, что в дек были введены 1, 2, …, n в данном порядке. На выходе, пользуясь той свободой вывода, которой обладает дек, получим некоторую перестановку вида n, … . Например, при $n = 4$ получаем перестановки 4123, 4132, 4321, 4312. Если в деке с ограниченным выходом первым на выходе появляется величина n, то это означает, что в дек была введена некоторая перестановка $p_1\ p_2\ \ldots\ p_n$, которую можно получить из 1, 2, …, n, пользуясь свободой входа, причём в зависимости от ограничений, наложенных на выход, либо $p_1 = n$, либо $p_n = n$. Например, при $n = 4$ получаем перестановки 4321, 4312, 4213, 4123. Поэтому получаем: **а)** 4132; **б)** 4213; **в)** 4231.

8. Если, например, $n = 4$, то таких перестановок не существует. Если же $n = 5$, то их четыре: 51324, 51342, 52314, 52341.

§ 2

1. Состояние памяти после переупаковки:

1	1	2				3	3	3	3								4	4	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Базовые адреса стеков: *BASE* [1] = –1, *BASE* [2] = 1, *BASE* [3] = 5, *BASE* [4] = 16 и адреса верхних элементов стеков: *TOP* [1] = 1, *TOP* [2] = 2, *TOP* [3] = 10, *TOP* [4] = 18 получены в результате выполнения алгоритма ***G***.

Исходными данными для этого алгоритма послужили: *OLDTOP* [1] = 1, *OLDTOP* [2] = 2, *OLDTOP* [3] = 8, *OLDTOP* [4] = 11; номер переполнившегося стека $i = 3$, *BASE* [1] = –1, *BASE* [2] = 2, *BASE* [3] = 5, *BASE* [4] = 9, *BASE* [5] = 19, *TOP* [1] = 1, *TOP* [2] = 3, *TOP* [3] = 10, *TOP* [4] = 11, *NEWBASE* [5] = 19, $L_\infty = 19$, $L_0 = -1$.

Приведём последовательность выполнения шагов алгоритма: *G*1, *G*2, *G*3, *G*4, *G*5, *G*6, *R*1, *R*2, *R*3, *R*4, *R*5, Конец алгоритма *R*, Конец алгоритма *G*.

§ 3

1. Узлы будут выведены в порядке: 1, 9, 3, 2, 7, 4, 5, 8 и 6. Приведём последовательность выполнения шагов алгоритма: *T*1, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*2, *T*3, *T*4, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*6, *T*6, *T*7, *T*5, *T*8.

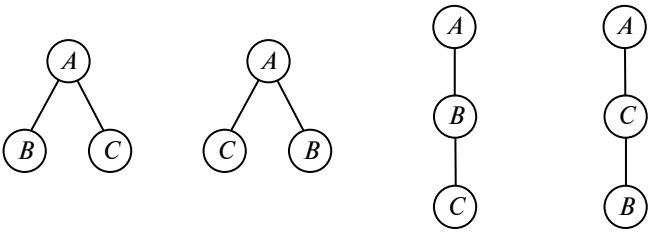
2. Прибавление многочлена $xy + xz - x$ к многочлену $-xz + 2x - 2y$ влечёт последовательное выполнение шагов *A*1, *A*2, *A*5, *A*2, *A*3, *A*4, *A*2, *A*3, *A*2, *A*2, *A*3 алгоритма и даёт итоговый многочлен $xy + x - 2y$.

§ 4

1. В результате решения будет получена следующая последовательность шагов алгоритма: *S*1, *S*2, *S*2, *S*2, *S*3, *S*4, *S*5, *S*5, *S*7, *S*8, *S*4, *S*4, *S*3, *S*3, *S*4, *S*5, *S*6, *S*6, *S*7, *S*4, *S*4, *S*3, а конечная матрица будет содержать ненулевые элементы: $m[1][3] = -100$, $m[1][1] = -5$, $m[2][3] = 2$, $m[2][1] = 0.1$, $m[4][4] = 5$ и $m[4][1]=3$.

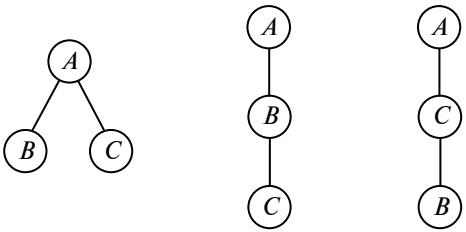
§ 5

1.



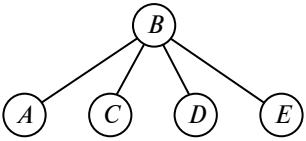
Существует всего 12 различных (упорядоченных) деревьев с тремя узлами *A*, *B* и *C*.

2.



Существует всего 9 различных ориентированных деревьев с тремя узлами *A*, *B* и *C*.

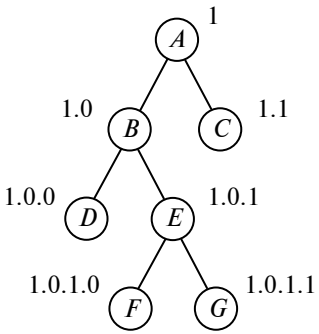
3.



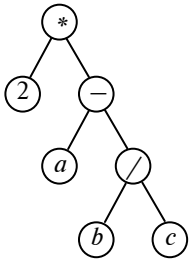
Степень узла *B* равна 4.

4. Последовательность обозначений узлов на пути от узла *X* к корню: $a_1.a_2. \ldots . a_k$, $a_1.a_2. \ldots . a_{k-1}$, $a_1.a_2. \ldots . a_{k-2}$, …, a_1.

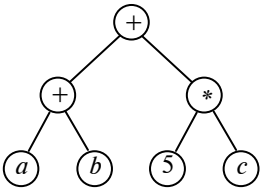
5. Обозначим корень бинарного дерева цифрой 1, корень левого поддерева узла с обозначением α – через α.0, а корень правого поддерева узла с обозначением α – через α.1. Например:



6. **а)**



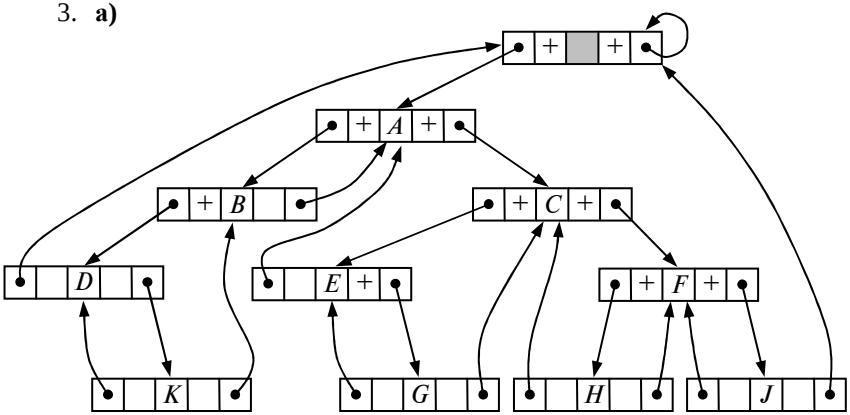
б)



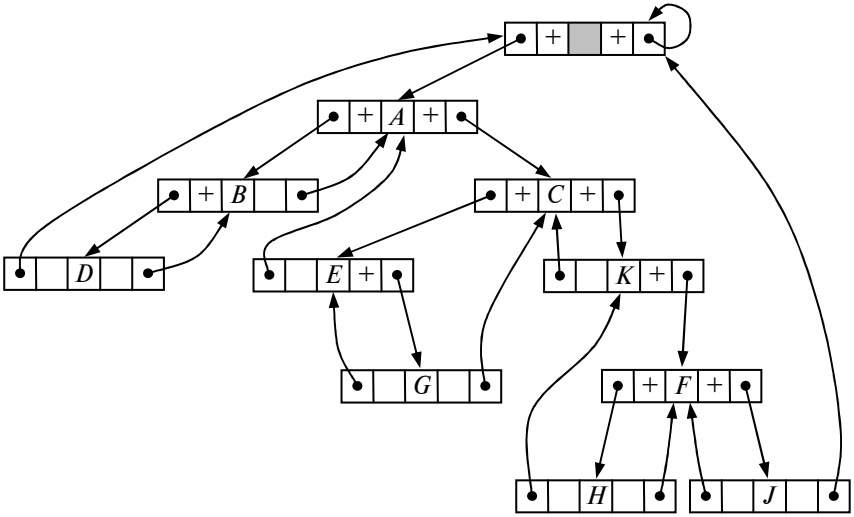
§ 6

1. Линейное расположение узлов: *BJFDGACEH*.

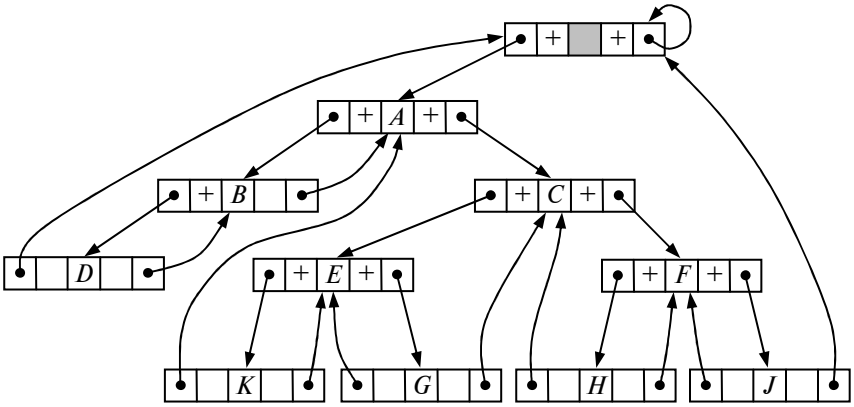
3. **а)**



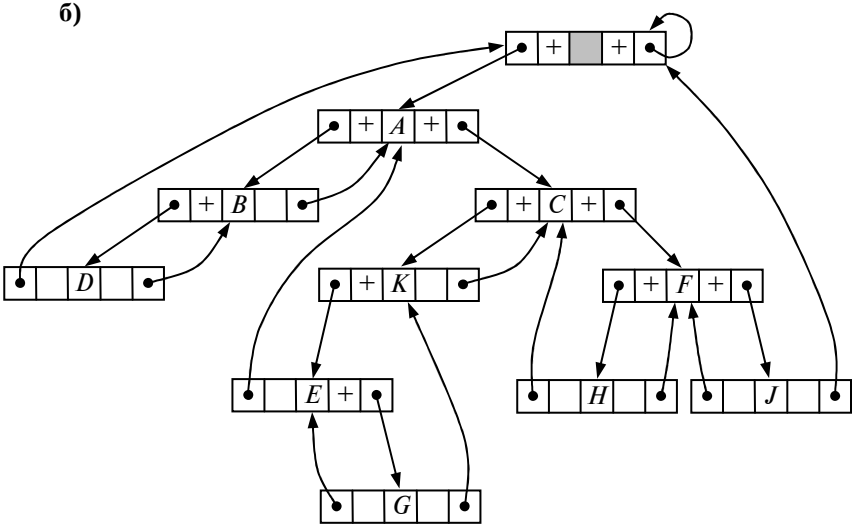
б)



4. **а)**

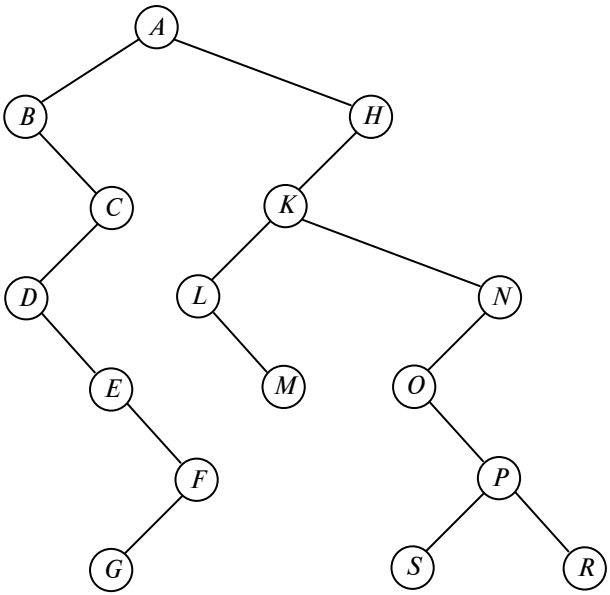


б)

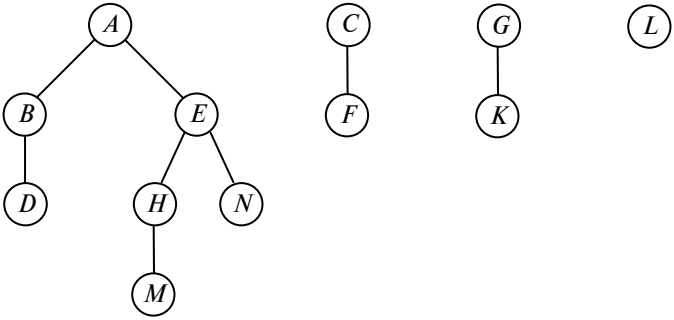


§ 7

1. Бинарное дерево

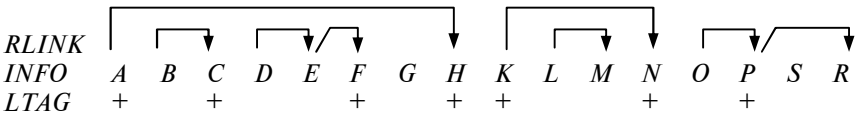


2. Лес

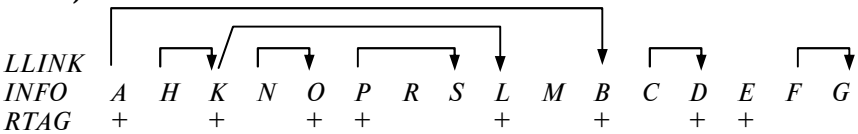


3. **а)** *ABCDEFGHKLMNOPSR*; **б)** *BDEGFCALMKOSPRNH*; **в)** *GFED CBMLSRPONKHA*.

4. **а)**



б)

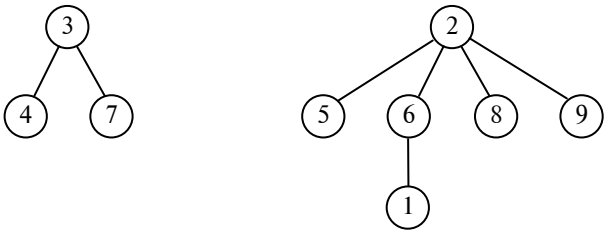


в)

Степень	0	0	0	0	1	3	2	0	0	2	0	0	1	0	3	2
INFO	*B*	*D*	*E*	*G*	*F*	*C*	*A*	*L*	*M*	*K*	*O*	*S*	*P*	*R*	*N*	*H*

§ 8

1. Лес из двух деревьев



определяет разбиение множества M на два класса эквивалентности: $K_1 = \{1, 2, 5, 6, 8, 9\}$ и $K_2 = \{3, 4, 7\}$.

2. **а)** Независимые величины E_0, E_{11}, E_{12}, E''_{13}, E_{14} и выражения для зависимых величин:

$E_1 = E_0$, $E_5 = E_0 - E_{11} - E_{12} + E_{14}$, $E_9 = E_{12}$,

$E_2 = E_0 + E''_{13} + E_{14}$, $E_6 = E_0$, $E_{10} = E_{11}$,

$E_3 = E_0 + E_{14}$, $E_7 = E_0$, $E'_{13} = E''_{13}$;

$E_4 = E_0 - E_{11} - E_{12} + E_{14}$, $E_8 = E_{11} + E_{12}$,

б) если $E_0 = 1$, $E_{11} = 2$, $E_{12} = 3$, $E''_{13} = 2$ и $E_{14} = 6$, то $E_1 = 1$, $E_2 = 9$, $E_3 = 7$, $E_4 = 2$, $E_5 = 2$, $E_6 = 1$, $E_7 = 1$, $E_8 = 5$, $E_9 = 3$, $E_{10} = 2$ и $E'_{13} = 2$;

в) $A = 9$, $B = 9$, $C = 7$, $D = 2$, $F = 5$, $K = 3$, $L = 2$, $M = 7$ и $N = 1$.

§ 9

1. Минимальное время 32 *и*, максимальное время 48 *и*, среднее время $37\frac{2}{15}$ *и*.

§ 10

1. $N = 16$, а записи в памяти размещены так, что образуют последовательность ключей:

K

5	0	5	0	9	1	8	2	6	4	1	5	6	6	7	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

По окончании работы алгоритма *C* вспомогательная таблица *COUNT* будет иметь следующий вид:

COUNT

6	0	7	1	15	2	14	4	9	5	3	8	10	11	12	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Сортировка является устойчивой, поскольку если $K_i = K_j$ и $j > i$, то $COUNT[j] > COUNT[i]$.

2. $N = 16$, а записи в памяти размещены так, что образуют последовательность:

R

5 *T*	0 *C*	5 *U*	0 *O*	9 .	1 *N*	8 *S*	2 *R*	6 *A*	4 *A*	1 *G*	5 *L*	6 *T*	6 *I*	7 *O*	7 *N*
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

По окончании работы алгоритма *D* в области вывода записи будут размещены следующим образом:

S

0 *C*	0 *O*	1 *N*	1 *G*	2 *R*	4 *A*	5 *T*	5 *U*	5 *L*	6 *A*	6 *T*	6 *I*	7 *O*	7 *N*	8 *S*	9 .
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Заметим, что в результате сортировки сформировалось слово «*CONGRATULATIONS*.» – поздравления.

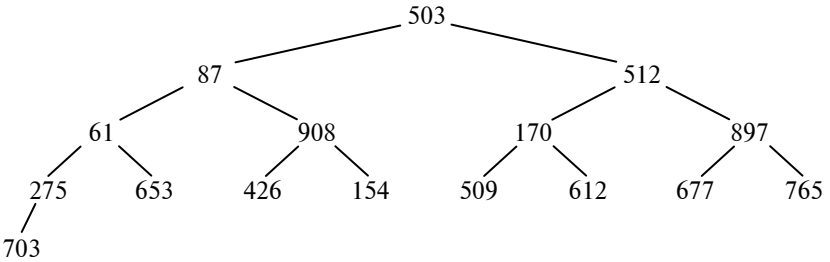
§ 11

3. Последовательность шагов: 3280, 1093, 364, 121, 40, 13, 4, 1.

§ 13

1. Будет выполнено 117 сравнений: 26 сравнений (для выбора ключа 98), 5 сравнений (для выбора 97), 5 (96), 5 (83), 5 (78), 5 (69), (66_2), 5 (66_1), 4 (63), 4 (62), 5 (60), 4 (56), 4 (51), 4 (48), 5 (41), 3 (28), 3 (26), 4 (22), 4 (20), 2 (16), 3 (14), 2 (10), 3 (9_2), 1 (9_1), 1 (8), 0 (7), 0 (5).

2**. а)** исходное полное бинарное дерево с 16 ячейками (без всяких бесполезных ячеек, содержащих «–∞»)



б) выполнение пирамидальной сортировки по алгоритму *H* с выводом последовательности ключей, а также значений переменных *l*, *r* и *K* на шаге *H*2 непосредственно перед проверкой равенства значения переменной *r* числу 1 даёт следующие результаты:

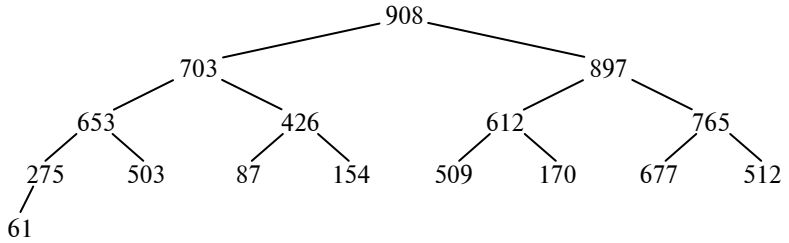
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	*l*	*r*	*K*
503	87	512	61	908	170	897	275	653	426	154	509	612	677	765	703	8	16	275
503	87	512	61	908	170	897	703	653	426	154	509	612	677	765	275	7	16	897
503	87	512	61	908	170	897	703	653	426	154	509	612	677	765	275	6	16	170
503	87	512	61	908	612	897	703	653	426	154	509	170	677	765	275	5	16	908
503	87	512	61	908	612	897	703	653	426	154	509	170	677	765	275	4	16	61
503	87	512	703	908	612	897	275	653	426	154	509	170	677	765	61	3	16	512
503	87	897	703	908	612	765	275	653	426	154	509	170	677	512	61	2	16	87
503	908	897	703	426	612	765	275	653	87	154	509	170	677	512	61	1	16	503
908	703	897	653	426	612	765	275	503	87	154	509	170	677	512	908	1	15	61
897	703	765	653	426	612	677	275	503	87	154	509	170	61	897	908	1	14	512
765	703	677	653	426	612	512	275	503	87	154	509	170	765	897	908	1	13	61
703	653	677	503	426	612	512	275	61	87	154	509	703	765	897	908	1	12	170
677	653	612	503	426	509	512	275	61	87	154	677	703	765	897	908	1	11	170
653	503	612	275	426	509	512	170	61	87	653	677	703	765	897	908	1	10	154
612	503	512	275	426	509	154	170	61	612	653	677	703	765	897	908	1	9	87
512	503	509	275	426	87	154	170	512	612	653	677	703	765	897	908	1	8	61
509	503	154	275	426	87	61	509	512	612	653	677	703	765	897	908	1	7	170
503	426	154	275	170	87	503	509	512	612	653	677	703	765	897	908	1	6	61

426	275	154	61	170	426	503	509	512	612	653	677	703	765	897	908	1	5	87
275	170	154	61	275	426	503	509	512	612	653	677	703	765	897	908	1	4	87
170	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	3	61
154	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	2	61
87	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	1	61

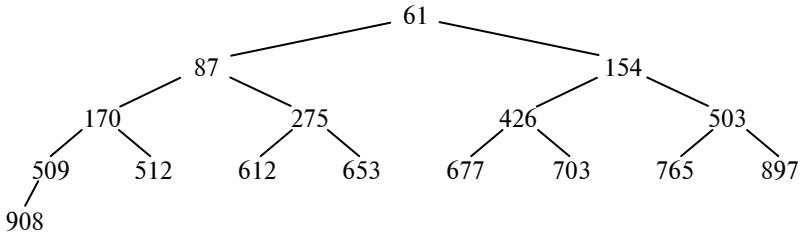
Последовательность ключей по окончании сортировки

61	87	154	170	275	426	503	509	512	612	653	677	703	765	897	908
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Заметим, что в процессе сортировки исходное бинарное дерево сначала перестраивается в пирамиду



а после многократного исключения вершины пирамиды и записи её на своё окончательное место бинарное дерево приобретает итоговый вид



§ 14

1. В результате слияния по алгоритму *M* двух упорядоченных подфайлов *x* = (12, 13, 14, 14, 25, 42, 47, 62) и *y* = (16, 35, 46, 52, 61, 73, 91, 103) получен упорядоченный файл *z* = (12, 13, 14, 14, 16, 25, 35, 42, 46, 47, 52, 61, 62, 73, 91, 103). При этом шаг *M*3 был выполнен восемь, а шаг *M*5 – пять раз.

2. Сортировка естественным двухпутевым слиянием по алгоритму *N* даёт в итоге следующую последовательность из девятнадцати записей: (1 «Бросая»), (4 «в»), (9_1 «воду»), (9_2 камешки), (10 «,»), (11_1 «смотри»), (11_2 «на»), (23 «круги»), (26 «,»), (29 «ими»), (31 «образуемые»), (35 «:»), (38 «иначе»), (69 «такое»), (71 «бросание»), (82 «будет»), (88 «пустою»), (92 «забавой»), (96 «.»). При этом сделано четыре просмотра (выполнения шага *N*2).

3. Выполнение по алгоритму *S* сортировки простым двухпутевым слиянием даёт в результате следующую последовательность из пятнадцати записей: (4 «Взирая»), (7 «на»), (10 «солнце»), (13 «,»), (26 «прищурь»), (29 «глаза»), (43_1 «свои»), (43_2 «и»), (49_1 «ты»), (49_2 «смело»), (54 «разглядишь»), (57 «на»), (69 «нем»), (71 «пятна»), (97 «.»).

При этом размещение ключей в двух используемых областях памяти до сортировки, а также по окончании каждого просмотра было следующим:

До сортировки	K_j	13	43	26	54	7	57	71	97	69	43	4	49	10	49	29
	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	K_j	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	j	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1-й просмотр	K_j	13	43	26	54	7	57	71	97	69	43	4	49	10	49	29
	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	K_j	13	29	10	26	4	7	69	71	97	57	43	54	49	49	43
	j	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
2-й просмотр	K_j	13	29	43	49	4	7	43	57	97	71	69	54	49	26	10
	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	K_j	13	29	10	26	4	7	69	71	97	57	43	54	49	49	43
	j	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
3-й просмотр	K_j	13	29	43	49	4	7	43	57	97	71	69	54	49	26	10
	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	K_j	10	13	26	29	43	49	49	54	97	71	69	57	43	7	4
	j	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
4-й просмотр	K_j	4	7	10	13	26	29	43	43	49	49	54	57	69	71	97
	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	K_j	10	13	26	29	43	49	49	54	97	71	69	57	43	7	4
	j	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

4. Сортировка посредством слияния списков по алгоритму *L* даёт в результате следующую последовательность из пятнадцати записей: (0 «Память»), (2 «человека»), (3 «есть»), (5 «лист»), (43 «белой»), (45 «бумаги»), (55 «:»), (66 «иногда»), (67 «напишется»), (70 «хорошо»), (74 «,»), (83 «а»), (85 «иногда»), (88 «дурно»), (93 «.»).

При этом распределение связей для двух линейных списков перед началом сортировки, а также по окончании каждого из просмотра было следующим:

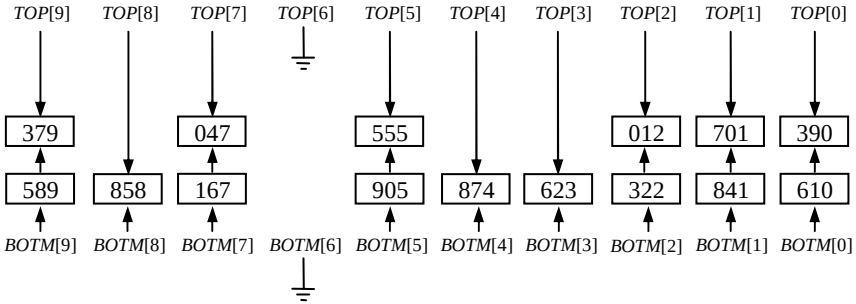
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K_j	–	70	85	66	43	55	67	0	93	3	83	88	5	2	74	45	–
L_j	1	–3	–4	–5	–6	–7	–8	–9	–10	–11	–12	–13	–14	–15	0	0	2

L_j	1	2	–5	–7	3	6	–9	8	–12	10	–13	–15	11	14	0	0	4
L_j	4	2	–9	1	3	6	8	5	–13	12	11	0	10	15	0	14	7
L_j	7	2	8	6	5	3	1	4	0	12	11	0	15	9	10	14	13
L_j	7	14	11	6	15	3	1	13	0	12	2	8	4	9	10	5	0

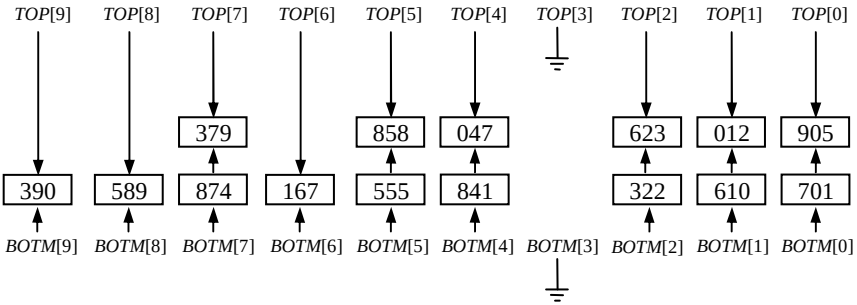
§ 15

1. В результате поразрядной сортировки по алгоритму R получается последовательность из пятнадцати записей: (12 «Достаток»), (47 «распутного»), (167 «равняется»), (322 «короткому»), (379 «одеялу»), (390 «:»), (555 «когда»), (589 «натянешь»), (610 «его»), (623 «к»), (701 «носу»), (841 «,»), (858 «обнажаются»), (874 «ноги») и (905 «.»), а ключи в стопках после каждого из трёх просмотров, выполняемых при сортировке с $M = 10$, разместятся следующим образом:

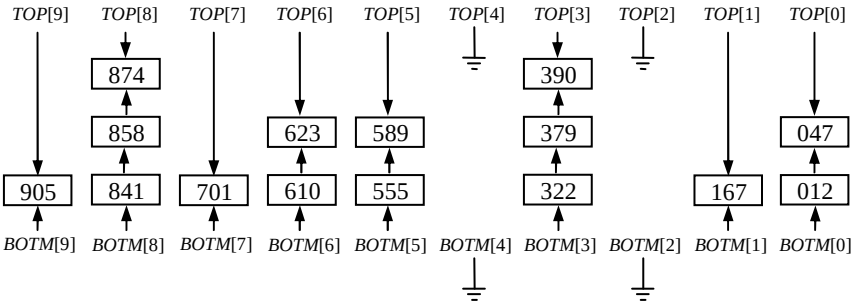
Содержимое стопок после первого просмотра



Содержимое стопок после второго просмотра



Содержимое стопок после третьего просмотра



§ 16

1. Без предварительной фазы внутренней сортировки обойтись можно, но при этом сортировка значительно замедлится, поскольку возрастёт число считываний каждой части данных из внешней памяти и записи в неё.

2. Первоначальное распределение отрезков по файлам будет таким:

Файл 1 $R_1 \ldots R_{1\,000\,000}$; $R_{2\,000\,001} \ldots R_{3\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 2 $R_{1\,000\,001} \ldots R_{2\,000\,000}$; $R_{3\,000\,001} \ldots R_{4\,000\,000}$.

Файл 3 (пустой)

Затем будет выполнено двухпутевое слияние из первого и второго файлов в третий файл:

Файл 1 $R_1 \ldots R_{1\,000\,000}$; $R_{2\,000\,001} \ldots R_{3\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 2 $R_{1\,000\,001} \ldots R_{2\,000\,000}$; $R_{3\,000\,001} \ldots R_{4\,000\,000}$.

Файл 3 $R_1 \ldots R_{2\,000\,000}$; $R_{2\,000\,001} \ldots R_{4\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

После обнуления указателей всех файлов и «однопутевого» слияния (распределения) из файла 3 в файлы 1 и 2 будет получено следующее состояние файлов:

Файл 1 $R_1 \ldots R_{2\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 2 $R_{2\,000\,001} \ldots R_{4\,000\,000}$.

Файл 3 $R_1 \ldots R_{2\,000\,000}$; $R_{2\,000\,001} \ldots R_{4\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Затем файлы 1 и 2 сливаются в файл 3:

Файл 1 $R_1 \ldots R_{2\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 2 $R_{2\,000\,001} \ldots R_{4\,000\,000}$.

Файл 3 $R_1 \ldots R_{4\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Информация распределяется:

Файл 1 $R_1 \ldots R_{4\,000\,000}$.

Файл 2 $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 3 $R_1 \ldots R_{4\,000\,000}$; $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

и сливается ещё раз:

Файл 1 $R_1 \ldots R_{4\,000\,000}$.

Файл 2 $R_{4\,000\,001} \ldots R_{5\,000\,000}$.

Файл 3 $R_1 \ldots R_{5\,000\,000}$.

В итоге имеем пять проходов. В общем случае эта процедура аналогична четырёхфайловому сбалансированному слиянию, но выполняется распределение между любыми двумя слияниями, что приводит к удвоенному числу проходов минус один проход.

§ 17

1. С помощью алгоритма *R* выбора с замещением (для $P = 4$) установите образуемые в выходном файле отрезки, если во входном файле находится последовательность из шестнадцати записей с ключами:

а) отрезок 1: 4, 25, 64, 69, 79, 87, 94; отрезок 2: 3, 13, 15, 16, 20, 30, 44, 72; отрезок 3: 9;

б) отрезок 1: 42, 74, 77, 88, 92, 94_1, 94_2, 96; отрезок 2: 40, 45, 55, 56; отрезок 3: 5, 22, 36, 37;

в) отрезок 1: 3, 32, 34, 61, 64, 65, 73, 98; отрезок 2: 12, 21, 35, 38, 54, 69, 81, 82.

§ 18

1. Таблица пошагового преобразования фамилий *Euler*, *Gauss*, *Hilbert*, *Knuth*, *Lloyd* и *Lukasiewicz* в соответствующие им коды методом *Soundex*:

	Шаг 1	Шаг 2	Шаг 3	Шаг 4
Euler	*Elr*	*E*46	*E*46	*E*460
Gauss	*Gss*	*G*22	*G*2	*G*200
Hilbert	*Hlbrt*	*H*4163	*H*4163	*H*416
Knuth	*Knt*	*K*53	*K*53	*K*530
Lloyd	*Lld*	*L*43	*L*3	*L*300
Lukasiewicz	*Lkscz*	*L*2222	*L*222	*L*222

2. Количество сравнений $K = K_i$ и количество сравнений $i \le N$, производимых алгоритмом *S* и алгоритмом *Q*, при поиске в последовательности ключей 503, 87, 512, 61, 908, 170, 897, 275, 653, 426, 154, 509, 612, 677, 765, 703:

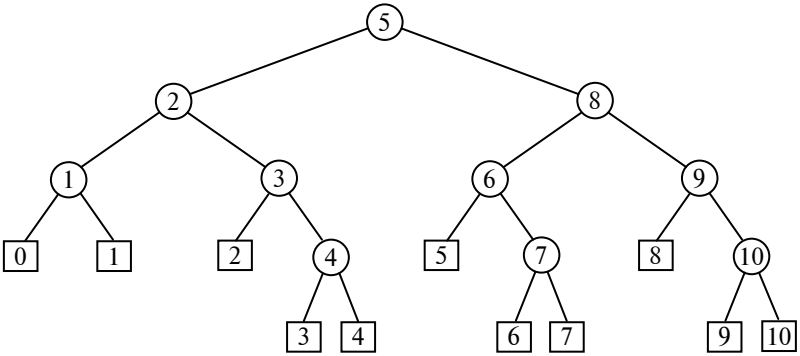
	Успешный поиск ключа 765		Неудачный поиск ключа 843	
	Количество сравнений $K = K_i$	Количество сравнений $i \le N$	Количество сравнений $K = K_i$	Количество сравнений $i \le N$
Алгоритм *S*	15	14	16	16
Алгоритм *Q*	15	1	17	1

3. Общее количество сравнений, производимых алгоритмом *Q*, и общее количество сравнений, производимых алгоритмом *T*, при поиске в упорядоченной последовательности ключей 61, 87, 154, 170, 275, 426, 503, 509, 512, 612, 653, 677, 703, 765, 897, 908:

	Успешный поиск ключа			Неудачный поиск ключа	
	61	512	908	172	769
Алгоритм *Q*	2	10	17	18	18
Алгоритм *T*	2	10	17	6	16

§ 19

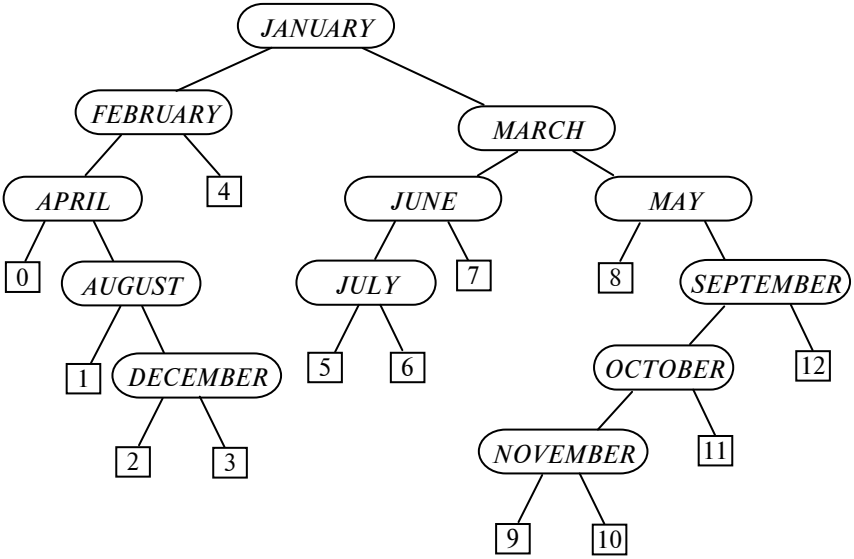
1. Бинарное дерево (подобное изображённому на рис. 38), которое соответствует бинарному поиску в упорядоченной таблице из десяти записей имеет следующий вид:



Аргументы поиска и соответствующие им пути в бинарном дереве: 50 – (5, 2, 1, 0); 61 – (5, 2, 1); 72 – (5, 2, 1, 1); 87 – (5, 2); 112 – (5, 2, 3, 2); 154 – (5, 2, 3); 163 – (5, 2, 3, 4, 3); 170 – (5, 2, 3, 4); 275 – (5).

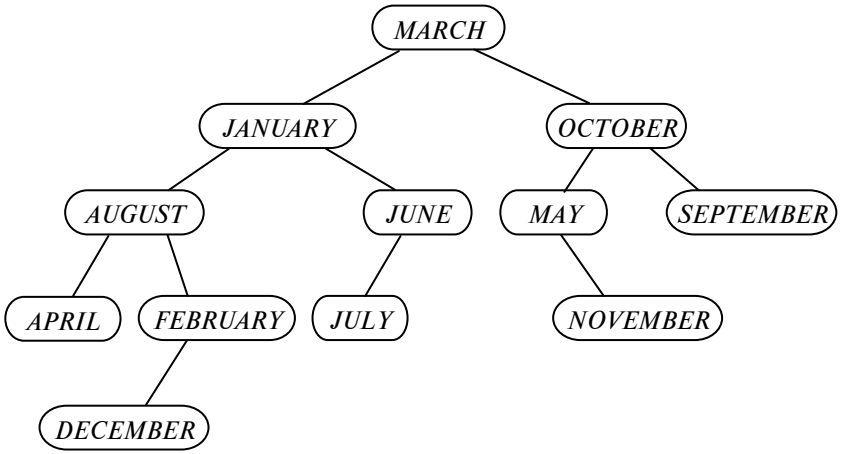
§ 20

1. Бинарное дерево, полученное в результате выполнения алгоритма *T* при последовательной вставке в пустое дерево ключей *JANUARY*, *FEBRUARY*, *MARCH*, *APRIL*, *MAY*, *JUNE*, *JULY*, *AUGUST*, *SEPTEMBER*, *OCTOBER*, *NOVEMBER*, *DECEMBER*:



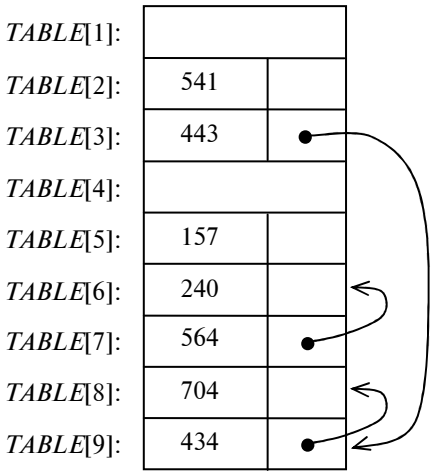
§ 21

1. В результате выполнения алгоритма *A* в условиях вставки в корень пустого дерева ключа *JANUARY* и дальнейшей последовательной вставки ключей *FEBRUARY*, *MARCH*, *APRIL*, *MAY*, *JUNE*, *JULY*, *AUGUST*, *SEPTEMBER*, *OCTOBER*, *NOVEMBER*, *DECEMBER* и получится следующее сбалансированное бинарное дерево.



§ 22

1. В результате последовательной вставки ключей 443, 564, 434, 704, 157, 541, 240 в пустую хеш-таблицу из $M = 9$ узлов в соответствии с алгоритмом C ключи в ней разместятся следующим образом:



2. При последовательной вставке с помощью алгоритма L ключей 601, 372, 268, 365, 622, 609, 133 в пустую хеш-таблицу из $M = 9$ узлов, они разместятся в ней следующим образом:

0	
1	622
2	133
3	372
4	609
5	365
6	268
7	601
8	

ЗАКЛЮЧЕНИЕ

В рамках изучаемой дисциплины «Методы программирования», а также родственных с ней дисциплин «Технологии и методы программирования», «Технология программирования», «Теория и технология программирования», «Основы алгоритмизации и языки программирования» студенты слушают лекции, участвуют в практических занятиях, выполняют лабораторные и курсовые работы.

Рассмотренные в пособии линейные информационные структуры, *n*-мерные массивы и такие важные нелинейные структуры, как деревья, дают возможность понимания статических и динамических свойств этих структур, средств выделения памяти для хранения структурированных данных, эффективных алгоритмов создания, изменения и удаления данных, а также всякого доступа к ним.

Владение анализом занимаемой памяти и быстродействия алгоритмов позволяет программисту выбрать из нескольких алгоритмов решения одной и той же конкретной задачи наилучший алгоритм.

Знание различных методов сортировки и поиска предоставляет возможность их эффективного применения в таких важных областях, как решение задачи группирования элементов с одинаковыми значениями некоторого признака, поиска общих элементов в двух или более массивах за один последовательный их просмотр без возвратов, а также задачи поиска информации по заданным значениям ключей.

СПИСОК ЛИТЕРАТУРЫ

1. Кулаков, Ю.В. Методы программирования: программа, метод. указ. и контр. вопросы / сост. : Ю.В. Кулаков, В.Н. Шамкин. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2006. – 32 с.

2. Кнут, Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы / Д. Кнут. – М. : Мир, 1976. – 736 с.

3. Кнут, Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск / Д. Кнут. – М. : Мир, 1978. – 846 с.

4. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл. – М. : Техносфера, 2004. – 368 с.

5. Нивергельт, Ю. Машинный поход к решению математических задач / Ю. Нивергельт, Дж. Фаррар, Э. Рейнголд. – М. : Мир, 1977. – 352 с.

6. Уайс, М.А. Организация структур данных и решение задач на C++ / М.А. Уайс. – М. : ЭКОМ Паблишерз, 2008. – 896 с.

ОГЛАВЛЕНИЕ

Учебное издание

ГРОМОВ Юрий Юрьевич,
ИВАНОВА Ольга Геннадьевна,
КУЛАКОВ Юрий Владимирович,
МИНИН Юрий Викторович,
ОДНОЛЬКО Валерий Григорьевич

МЕТОДЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор Л.В. Комбарова
Инженер по компьютерному макетированию М.Н. Рыжкова

Подписано в печать 19.03.2012.
Формат 60 × 84 / 16. 8,37 усл. печ. л. Тираж 100 экз. Заказ № 112

Издательско-полиграфический центр ФГБОУ ВПО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14