2. Алгоритмы обработки данных

2.1. *NP*-сложные и труднорешаемые задачи

Для оценки сложности алгоритма (см. введение) используется *O*-символика.

На основе этой оценки можно привести следующую классификацию алгоритмов, при размере входных данных, равном n:

– постоянные – сложность не зависит от n: $O(1)$;

– линейные – сложность $O(n)$;

– полиномиальные – сложность $O(n^m)$, где m – константа;

– экспоненциальные – сложность $O(t^{f(n)})$, где t – константа, которая больше 1, а $f(n)$ – полиномиальная функция;

– суперполиномиальные – сложность $O(c^{f(n)})$, где c – константа, а $f(n)$ – функция возрастающая быстрее постоянной, но медленнее линейной;

Простые задачи (решаемые) – это задачи, решаемые за полиномиальное время.

Труднорешаемые задачи – это задачи, которые не решаются за полиномиальное время, либо алгоритм решения за полиномиальное время не найден.

Помимо этого, как было доказано А. Тьюрингом, существуют принципиально неразрешимые задачи.

Сложность задач не определяется по сложности наилучшего алгоритма, ее решающего. Для оценки сложности вводится классификация по сложности функций, вычисление которых возможно при задаваемых ограничениях на потребляемые ресурсы:

1) класс P – класс задач, которые можно решить за полиномиальное время;

2) класс NP – класс задач, которые можно решить за полиномиальное время только на недетерминированной Машине Тьюринга, которая в отличие от обычной Машины Тьюринга может делать предположения. Это задачи, у которых есть ответ, найти который трудно, но проверить можно за полиномиальное время;

3) класс *NP*-полных задач – класс задач, не менее сложных, чем любая *NP*-задача;

4) класс *EXPTIME* – класс задач, которые решаются за экспоненциальное время;

5) класс *EXPTIME*-полных задач – класс задач, которые не решаются за детерминированное полиномиальное время. Известно, что $P <> EXPTIME$.

Очевидно, что P входит в NP, но вот проверить, что ($P \neq NP$) или ($P = NP$) до сих пор не удалось.

В качестве примера *NP*-полной задачи приведем задачу коммивояжера. Имеется n пунктов, расстояния между любыми двумя из которых известны и представляют собой целые числа. Требуется объехать все пункты, посетив каждый по одному разу и возвратившись в исходный пункт так, чтобы длина полученного кольцевого маршрута была наименьшей.

Общее число обходов n пунктов равно, как легко заметить, $(n - 1)!/2$. Решая эту задачу методом перебора всех возможных кольцевых маршрутов и выбора из них самого короткого, придется выполнить такое же количество итераций. Данный метод решения делает задачу коммивояжера *NP*-полной задачей.

Приняв $n = 100$ и произведя очень грубую (явно заниженную) оценку, можно сделать вывод, что для решения данной задачи придется выполнить не менее 10^{21} операций. Скорость самых современных ЭВМ не превосходит 10^{12} операций в секунду. Следовательно, для получения результата придется ждать 10^9 с, или более 30 лет.

В настоящее время полиномиальное решение задачи коммивояжера неизвестно.

2.2. Методы разработки алгоритмов

2.2.1. Метод декомпозиции

Этот метод, называемый также методом «разделяй и властвуй», или методом разбиения, возможно, является самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов. Он предполагает такую декомпозицию (разбиение) задачи размера n на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи. В качестве примеров применений этого метода можно назвать сортиров-

ку слиянием или применение деревьев двоичного поиска, которые рассматриваются далее.

Проиллюстрируем этот метод на хорошо известном примере (рис. 23). Так, имеются три стержня *A*, *B* и *C*. Вначале на стержень *A* нанизаны несколько дисков: диск наибольшего диаметра находится внизу, а выше – диски последовательно уменьшающегося диаметра. Цель головоломки – перемещать диски (по одному) со стержня на стержень так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы в конце концов все диски оказались нанизанными на стержень *B*. Стержень *C* можно использовать для временного хранения дисков.

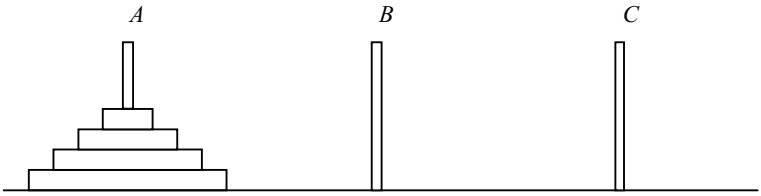


Рис. 23. Головоломка «Ханойские башни»

Задачу перемещения *n* наименьших дисков со стержня *A* на стержень *B* можно представить себе состоящей из двух подзадач размера $n - 1$. Сначала нужно переместить $n - 1$ наименьших дисков со стержня *A* на стержень *C*, оставив на стержне *A* *n*-й наибольший диск. Затем этот диск нужно переместить с *A* на *B*. Потом следует переместить $n - 1$ дисков со стержня *C* на стержень *B*. Это перемещение $n - 1$ дисков выполняется путем рекурсивного применения указанного метода. Поскольку диски, участвующие в перемещениях, по размеру меньше тех, которые в перемещении не участвуют, не нужно задумываться над тем, что находится под перемещаемыми дисками на стержнях *A*, *B* или *C*. Хотя фактическое перемещение отдельных дисков не столь очевидно, а моделирование вручную выполнить непросто из-за образования стеков рекурсивных вызовов, с концептуальной точки зрения этот алгоритм все же довольно прост для понимания и доказательства его правильности (а если говорить о быстроте разработки, то ему вообще нет равных). Именно легкость разработки алгоритмов по методу декомпозиции обусловила огромную популярность этого метода; к тому же во многих случаях эти алгоритмы оказываются более эффективными, чем алгоритмы, разработанные традиционными методами.

2.2.2. Динамическое программирование

Этот метод имеет под собой достаточно серьезную научную основу, однако его суть вполне можно объяснить на простом примере чисел Фибоначчи.

Вычислить *N* чисел в последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, … , где первые два члена равны единице, а все остальные представляют собой сумму двух предыдущих, *N* меньше 100.

Самый очевидный способ «решения» задачи состоит в написании рекурсивной функции примерно следующего вида:

```
function F(X: integer): longint;
begin
  if (X = 1) or (X = 2) then F := 1
                        else F := F(X-1) + F(X-2)
end;
```

При этом на шестом-седьмом десятке программе перестанет хватать временных ресурсов самой современной вычислительной машины. Это происходит по следующим причинам.

Для вычисления, например, `F(40)` вначале вычисляется `F(39)` и `F(38)`. Причем `F(38)` вычисляется заново, без использования значения, которое было вычислено, когда считалось `F(39)`, т. е. значение функции при одном и том же значении аргумента считается много раз. Если исключить повторный счет, то функция станет заметно эффективней. Для этого приходится завести массив, в котором хранятся значения нашей функции:

```
var D: array[1..100] of longint;
```

Сначала массив заполняется значениями, которые заведомо не могут быть значениями функции (чаще всего, это «–1», но вполне пригоден 0). При попытке вычислить какое-то значение, программа смотрит, не вычислялось ли оно ранее, и если да, то берет готовый результат.

Функция принимает следующий вид:

```
function F(X: integer): longint;
begin
  if D[X] = 0 then
    if (X=1) or (X=2) then D[X] := 1
                      else D[X] := F(X-1) + F(X-2);
  F := D[X];
end;
```

Этот подход динамического программирования называется подходом «сверху вниз». Он запоминает решенные задачи, но очередность решения задач все равно контролирует рекурсия.

Можно еще более упростить решение, убрав рекурсию вообще. Для этого необходимо сменить нисходящую логику рассуждения (от того, что надо найти, к тривиальному) на восходящую (соответственно наоборот). В этой задаче такой переход очевиден и описывается простым циклом:

```
D[1] := 1; D[2] := 1;
For i := 3 to N do
  D[i] := D[i-1] + D[i-2];
```

Здесь использован подход «снизу вверх». Чаще всего, такой способ раза в три быстрее. Однако в ряде случаев такой метод приводит к необходимости решать большее количество подзадач, чем при рекурсии.

Очень часто для его написания приходится использовать как промежуточный результат нисходящую форму, а иногда безрекурсивная (итеративная) форма оказывается чрезвычайно сложной и малопонятной.

Таким образом, если алгоритм превышает отведенное ему время на тестах большого объема, то необходимо осуществлять доработку этого алгоритма.

2.2.3. Поиск с возвратом

Иногда приходится иметь дело с задачей поиска оптимального решения, когда невозможно применить ни один из известных методов, способных помочь отыскать оптимальный вариант решения, и остается прибегнуть к последнему средству – полному перебору. Приведем систематическое описание метода полного перебора, называемого поиском с возвратом.

Рассмотрим игры двух лиц, которые обычно описываются множеством «позиций» и совокупностью правил перехода из одной позиции в другую, причем предполагается, что игроки ходят по очереди. Будем считать, что правилами разрешены лишь конечные последовательности позиций и что в каждой позиции имеется лишь конечное число разрешенных ходов (рис. 24). Тогда для каждой позиции p найдется число $N(p)$ такое, что никакая игра, начавшаяся в p, не может продолжаться более $N(p)$ ходов.

Терминальными называются позиции, из которых нет разрешенных ходов. На каждой из них определена целочисленная функция $f(p)$, зада-

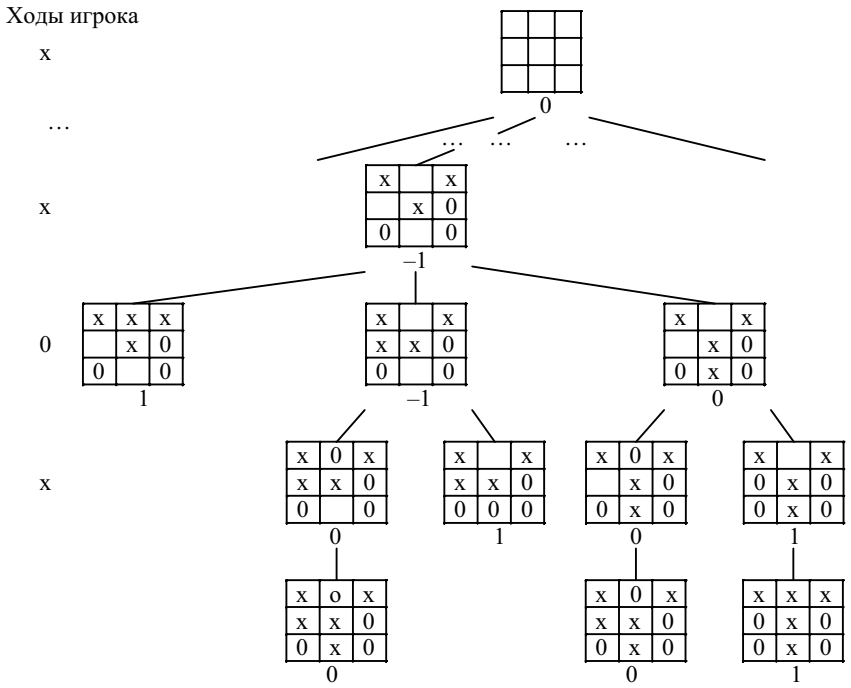


Рис. 24. Дерево игры «крестики-нолики»

ющая выигрыш того из игроков, которому принадлежит ход в этой позиции; выигрыш второго игрока считается равным $-f(p)$.

Если из позиции p имеется d разрешенных ходов $p_1, \ldots, p_d$, возникает проблема выбора лучшего из них. Будем называть ход наилучшим, если по окончании игры он приносит наибольший возможный выигрыш при условии, что противник выбирает ходы, наилучшие для него (в том же смысле). Пусть $f(p)$ есть наибольший выигрыш, достижимый в позиции p игроком, которому принадлежит очередь хода, против оптимальной защиты. Так как после хода в позицию p_i выигрыш этого игрока равен $-f(p_i)$, имеем

$$f(p) = \begin{cases} f(p) & d = 0, \\ \max\{-f(p_1), \ldots, -f(p_d)\}, & \text{если } d > \end{cases}$$

Эта формула позволяет индуктивно определить $f(p)$ для каждой позиции p.

Функция $f(p)$ равна тому максимуму, который гарантирован, если оба игрока действуют оптимально. Следует, однако, заметить, что она отражает результаты весьма осторожной стратегии, которая не обязательно хороша против плохих игроков или игроков, действующих согласно другому принципу оптимальности. Пусть, например, имеются два хода в позиции p_1 и p_2, причем p_1 гарантирует ничью (выигрыш 0) и не дает возможности выиграть, в то время как p_2 дает возможность выиграть, если противник просмотрит очень тонкий выигрывающий ход. В такой ситуации можно предпочесть рискованный ход в p_2, если только нет уверенности в том, что противник всемогущ и всезнающ. Очень возможно, что люди выигрывают у шахматных программ таким именно образом.

Нижеследующий алгоритм, называемый поиском с возвратом, вычисляет $f(p)$.

```
function BackSearch(p: position): integer;
{оценивает и возвращает выигрыш f(p) для позиции p}
var
  m,i,t,d: integer;
begin
  Определить позиции p1,...,pd, подчиненные p;
  if d = 0 then BackSearch := f(p)
  else begin
    m := -∞;
    for i:= 1 to d do begin
      t := - BackSearch(pi);
      if t > m then m:= t;
    end;
    BackSearch := m;
  end;
end;
```

Здесь $+\infty$ обозначает число, которое не меньше $abs(f(p))$ для любой терминальной позиции p; поэтому $-\infty$ не больше $f(p)$ и $-f(p)$ для всех p. Этот алгоритм вычисляет $f(p)$ на основе «грубой силы» – для каждой позиции он оценивает все возможные продолжения.

Если рассмотреть сложную игру (например, шахматы), в которой дерево игры, являясь, в принципе, конечным, столь огромно, что любые попытки оценить его по данному методу обречены на неудачу.

Проблема для наиболее интересных игр состоит в том, что размер этого дерева является чрезвычайно огромным, порядка W^L, где W – сред-

нее количество ходов в позиции, а L – количество уровней дерева. Перебор всего дерева невозможен, главным образом из-за недостатка времени, даже на самых быстрых вычислительных машинах. Все практические алгоритмы поиска являются некоторыми приближениями полного перебора.

2.2.4. Метод ветвей и границ

Перебор, который осуществляет поиск с возвратом, можно уменьшить, используя идею метода «ветвей и границ». Эта идея состоит в том, что можно не искать точную оценку хода, про который стало известно, что он не может быть лучше, чем один из ходов, рассмотренных ранее. Пусть, например, в процессе перебора стало известно, что $f(p_1) = -10$. Отсюда заключаем, что $f(p) \geq 10$, и потому не нужно знать точное значение $f(p_2)$, если каким-либо образом узнали, что $f(p_2) \geq -10$ (поскольку отсюда следует, что $-f(p_2) \leq 10$). Итак, если p_{21} допустимый ход из p_2 и $f(p_{21}) \leq 10$, можно не исследовать другие ходы из p_2. Говорят, что ход в позицию p_2 «опровергается» (ходом в p_1), если у противника в позиции p_2 есть ответ, по крайней мере, столь же хороший, как его лучший ответ в позиции p_1. Ясно, что если ход можно опровергнуть, можно не искать наилучшее опровержение.

Эти рассуждения приводят к методу «ветвей и границ», гораздо более экономному, чем поиск с возвратом. Определим метод «ветвей и границ» как процедуру с двумя параметрами p и *bound*, вычисляющую $f'(p, \text{bound})$. Цель алгоритма – удовлетворить следующим условиям:

$$f'(p, \text{bound}) = f(p), \text{ если } f(p) < \text{bound},$$

$$f'(p, \text{bound}) > \text{bound}, \text{ если } f(p) \geq \text{bound}.$$

Идею метода ветвей и границ реализует следующий алгоритм.

```
Function B&B(p: position, bound: integer): integer;
{оценивает и возвращает выигрыш F'(p) для позиции p}
label done;
var
  m,I,t,d: integer;
begin
  Определить позиции p1,…,pd, подчиненные p;
  if d = 0 then B&B := f(p) else begin
    m := -∞;
    for i:= 1 to d do begin
```

```
      t := - B&B(p_i, -m);
      if t > m then m := t;
      if m >= bound then goto done;
    end;
    done: B&B := m;
  end;
end;
```

2.2.5. Метод альфа-бета отсечения

Метод «ветвей и границ» можно еще улучшить, если ввести не только верхнюю, но и нижнюю границу. Эта идея – ее называют минимаксной альфа-бета процедурой или просто альфа-бета отсечением – является значительным продвижением по сравнению с односторонним методом «ветвей и границ». Определим процедуру f'' с тремя параметрами p, alpha и beta (причем всегда будет выполнено alpha < beta), которая удовлетворяет следующим условиям:

$$f''(p, \text{alpha}, \text{beta}) \le \text{alpha}, \text{ если } f(p) < \text{alpha},$$

$$f''(p, \text{alpha}, \text{beta}) = f(p), \text{ если alpha} < f(p) < \text{beta},$$

$$f''(p, \text{alpha}, \text{beta}) \ge \text{beta}, \text{ если } f(p) \ge \text{beta}.$$

Идею метода альфа-бета отсечения реализует следующий алгоритм:

```
function AB(p: position; alpha, beta: integer): integer;
{оценивает и возвращает выигрыш F''(p) для позиции p}
label done;
var
  m,i,t,d: integer;
begin
  Определить позиции p_1,...,p_d, подчиненные p;
  if d = 0 then AB := f(p) else begin
    m := alpha;
    for i:= 1 to d do begin
      t := -AB(p_i, -beta, -m);
      if t > m then m := t;
      if m >= beta then goto done;
    end;
    done: AB := m;
  end;
end;
```

Выгода от альфа-бета отсечения заключается в более раннем выходе из цикла. Эти отсечения полностью безопасны (корректны), потому что они гарантируют, что отсекаемая часть дерева хуже, чем основной вариант.

При оптимальных обстоятельствах перебор с альфа-бета отсечением должен просмотреть $W^{(L+1)/2} + W^{L/2}-1$ позицию, где W – среднее количество ходов в позиции, а L – количество уровней дерева. Это намного меньше, чем перебор с возвратом. Данное отсечение позволяет достигать примерно вдвое большей глубины за то же самое время.

2.2.6. Локальные и глобальные оптимальные решения

Описанная ниже стратегия нередко приводит к оптимальному решению задачи:

1) находится произвольное решение;

2) для улучшения текущего решения применяется к нему какое-либо преобразование из некоторой заданной совокупности преобразований. Это улучшенное решение становится новым «текущим» решением;

3) указанная процедура повторяется до тех пор, пока ни одно из преобразований в заданной их совокупности не позволит улучшить текущее решение.

Результирующее решение может, хотя и необязательно, оказаться оптимальным. В принципе, если «заданная совокупность преобразований» включает все преобразования, которые берут в качестве исходного одно решение и заменяют его каким-либо другим, процесс «улучшений» не закончится до тех пор, пока не получим оптимальное решение. Но в таком случае время выполнения пункта 2 окажется таким же, как и время, требующееся для анализа всех решений, поэтому описываемый подход в целом окажется достаточно бессмысленным.

Этот метод имеет смысл лишь в том случае, когда можно ограничить совокупность преобразований небольшим ее подмножеством, что дает возможность выполнить все преобразования за относительно короткое время: если «размер» задачи равняется n, то можно допустить $O(n^2)$ или $O(n^3)$ преобразований. Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как «близкие». Такие преобразования называются «локальными», а соответствующий метод называется локальным поиском.

Одной из задач, которую можно решить именно методом локального поиска, является задача нахождения минимального остовного дерева (см. 2.6.4). Локальными преобразованиями являются такие преобразования, в ходе которых берется то или иное ребро, не относящееся к текущему остовному дереву, оно добавляется в это дерево (в результате должен получиться цикл), а затем убирается из этого цикла в точности одно ребро (предположительно, ребро с наивысшей стоимостью), чтобы образовалось новое дерево.

Время, которое занимает выполнение этого алгоритма на графе из n узлов и e ребер, зависит от количества требующихся улучшений решения. Одна лишь проверка того факта, что преобразования уже неприменимы, может занять $O(n \cdot e)$ времени, поскольку для этого необходимо перебрать e ребер, а каждое из них может образовать цикл длиной примерно n. Таким образом, этот алгоритм несколько хуже, чем алгоритмы Прима и Крускала (см. 2.6.4.1 и 2.6.4.2), однако он может служить примером получения оптимального решения на основе локального поиска.

Алгоритмы локального поиска проявляют себя с наилучшей стороны как эвристические алгоритмы для решения задач, точные решения которых требуют экспоненциальных затрат времени (относятся к классу *EXPTIME*). Общепринятый метод поиска состоит в следующем. Начать следует с ряда произвольных решений, применяя к каждому из них локальные преобразования до тех пор, пока не будет получено локально-оптимальное решение, т. е. такое, которое не сможет улучшить ни одно преобразование. Как показывает рис. 25, на основе большинства (или даже всех) произвольных начальных решений нередко будут получаться разные локально-оптимальные решения. Если повезет, одно из них окажется глобально-оптимальным, т. е. лучше любого другого решения.

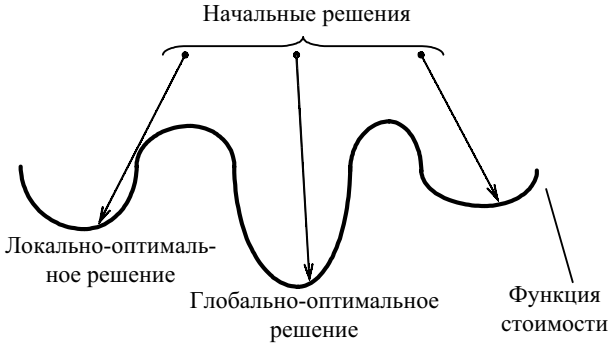


Рис. 25. Локальный поиск в пространстве решений

На практике можно и не найти глобально-оптимального решения, поскольку количество локально-оптимальных решений может оказаться колоссальным. Однако можно, по крайней мере, выбрать локально-оптимальное решение, имеющее минимальную стоимость среди всех найденных решений.

2.3. Алгоритмы поиска

Поиск – процесс отыскания информации во множестве данных (обычно представляющих собой записи) путем просмотра специального поля в каждой записи, называемого ключом. Целью поиска является отыскание записи (если она есть) с данным значением ключа.

Поиск – одно из наиболее часто встречающихся в программировании действий. Существует множество различных алгоритмов этого действия, принципиально зависящих от способа организации данных. Далее рассмотрены алгоритмы поиска в различных структурах данных.

2.3.1. Поиск в линейных структурах

При дальнейшем рассмотрении поиска в линейных структурах определим, что множество данных, в котором производится поиск, описывается как массив фиксированной длины:

```
A: array[1..n] of ItemType;
```

Обычно тип `ItemType` описывает запись с некоторым полем, играющим роль ключа, а сам массив представляет собой таблицу. Так как здесь рассматривается, прежде всего, сам процесс поиска, то будем считать, что тип `ItemType` включает только ключ.

2.3.1.1. Последовательный (линейный) поиск

Последовательный поиск – самый простой из известных. Суть его заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Именно так поступает человек, когда ищет что-то в неупорядоченном множестве. Таким образом, данный алгоритм можно применять

тогда, когда нет никакой дополнительной информации о расположении данных в рассматриваемой последовательности.

Оформим описанный алгоритм в виде функции на Паскале.

```
function LineSearch(Key: ItemType, n: integer;
                    var A: array[1..n] of ItemType): boolean;
{Функция линейного поиска,}
{если элемент найден, то возвращает значение true, иначе – false}
var
  i: integer;
begin
  i := 1;
  while (i <=n ) and (A[i]<>Key) do i := i + 1;
  if A[i]=Key then LineSearch := true
              else LineSearch := false;
end;
```

В среднем этот алгоритм требует $n/2$ итераций цикла. Это означает временную сложность алгоритма поиска, пропорциональную $O(n)$. Никаких ограничений на порядок элементов в массиве данный алгоритм не накладывает. При наличии в массиве нескольких элементов со значением `Key` алгоритм находит только первый из них (с наименьшим индексом).

Здесь можно хранить множество как в массиве (как показано выше), так и в обычном однонаправленном списке.

Существует модификация алгоритма последовательного поиска, которая ускоряет поиск.

Эта модификация поиска является небольшим усовершенствованием предыдущего. В любой программе, имеющей циклы, наибольший интерес представляет оптимизация именно циклов, т. е. сокращение числа действий в них. Посмотрим на алгоритм последовательного поиска. В цикле `while` производятся два сравнения: (`i<=n`) и (`A[i]<>Key`). Избавимся от одного из них (от первого), введя в массив так называемый «барьер», положив `A[n+1] := Key`. В этом случае в цикле обязательно будет найден элемент со значением `Key`. После завершения цикла требуется дополнительная проверка того, был ли найден искомый элемент, или «барьер».

Тогда функция поиска будет выглядеть так:

```
function LineSearchWithBarrier(Key: ItemType, n: integer;
                    var A: array[1..n+1] of ItemType): boolean;
```

```
{Функция линейного поиска с барьером,}
{если элемент найден, то возвращает значение true, иначе – false}
var
  i: integer;
begin
  I := 1;
  A[n+1] := Key;
  while A[i]<>Key do I := I + 1;
  if I <= n then LineSearchWithBarrier := true
            else LineSearchWithBarrier := false;
end;
```

Надо сказать, что хотя такая функция будет работать быстрее, но временная сложность алгоритма остается такой же – $O(n)$. Гораздо больший интерес представляют методы, не только работающие быстро, но и реализующие алгоритмы с меньшей сложностью. Один из таких методов – бинарный поиск.

2.3.1.2. Бинарный поиск

Этот алгоритм поиска предполагает, что множество хранится, как некоторая упорядоченная (например, по возрастанию) последовательность элементов, к которым можно получить прямой доступ посредством индекса. Фактически речь идет о том, что множество хранится в массиве и этот массив отсортирован.

Суть метода заключается в следующем. Областью поиска (L, R) назовем часть массива с индексами от L до R, в которой предположительно находится искомый элемент. Сначала областью поиска будет часть массива (L, R), где $L = 1$, а $R = n$, т. е. вся заполненная элементами множества часть массива. Теперь найдем индекс среднего элемента $m := (L+R)$ div 2. Если Key > A_m, то можно утверждать (поскольку массив отсортирован), что если `Key` есть в массиве, то он находится в одном из элементов с индексами от $L + m$ до R, следовательно, можно присвоить $L := m + 1$, сократив область поиска. В противном случае можно положить $R := m$. На этом заканчивается первый шаг метода. Остальные шаги аналогичны (рис. 26).

На каждом шаге метода область поиска будет сокращаться вдвое. Как только L станет равно R, т. е. область поиска сократится до одного элемента, можно будет проверить этот элемент на равенство искомому и сделать вывод о результате поиска.

Оформим описанный алгоритм в виде функции на Паскале.

```
function BinarySearch(Key: ItemType, n: integer;
                      var A: array[1..n] of ItemType): boolean;
{Функция двоичного поиска,}
{если элемент найден, то возвращает значение true, иначе – false}
var
  L, m, R: integer;
begin
  L := 1; R := n;
  while (L <> R) do begin
    m := (L+R) div 2;
    if Key > A[m] then L := m+1
                  else R := m;
  end;
  if A[L]= Key then BinarySearch := true
              else BinarySearch := false;
end;
```

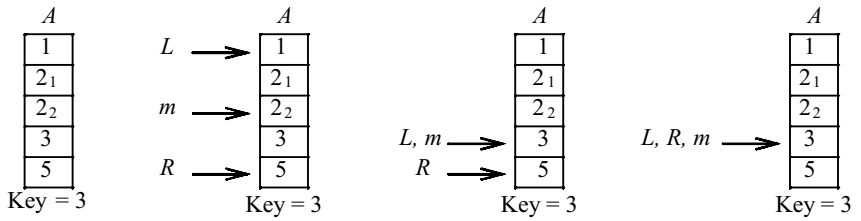


Рис. 26. Бинарный поиск

Как уже было сказано, область поиска на каждом шаге сокращается вдвое, а это означает сложность алгоритма пропорциональную $O(\log n)$.

2.3.2. Хеширование данных

2.3.2.1. Функция хеширования

В рассмотренных выше методах поиска число итераций в лучшем случае было пропорционально $O(\log n)$. Естественно, возникает желание найти такой метод поиска, при котором число итераций не зависело бы от размера таблицы, а в идеальном случае поиск сводился бы к одному шагу.

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ

является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись, если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех ячеек памяти, которые выделяются для хранения таблицы.

Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве реальных задач, однако, размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то, даже ограничив длину ключа 10 символами кириллицы, получаем 33^{10} возможных значений ключей. Даже если ресурсы вычислительной системы и позволят выделить пространство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей.

Из соображений экономии памяти целесообразно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, т. е., преобразование ключа в адрес записи: $a := h(k)$, где a – адрес, k – ключ. Такая функция называется *функцией хеширования* (другие ее названия – функция перемешивания, функция рандомизации).

Идеальной хеш-функцией является такая функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса: $k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$.

При попытке отображения точек из некоторого широкого пространства в узкое неизбежны ситуации, когда разные точки в исходном пространстве отобразятся в одну и ту же точку в целевом пространстве.

Ситуация, при которой разные ключи отображаются в один и тот же адрес записи, называется коллизией, или переполнением, а такие ключи называются синонимами. Коллизии составляют основную проблему для хеш-таблиц и решение ее будет рассмотрено особо.

Если хеш-функция, преобразующая ключ в адрес, может порождать коллизии, то однозначной обратной функции: $k := h'(a)$, позволяющей восстановить ключ по известному адресу, существовать не может. Поэтому ключ должен обязательно входить в состав записи хешированной таблицы как одно из ее полей.

К хеш-функции в общем случае предъявляются следующие требования:

– она должна обеспечивать равномерное распределение отображений фактических ключей по пространству записей (рис. 27);

– она должна порождать как можно меньше коллизий для данного фактического множества записей;

– она не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;

– она должна быть простой и быстрой для вычисления.

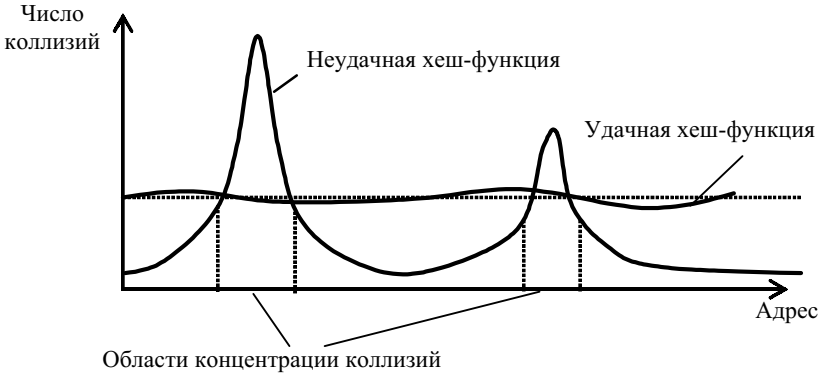


Рис. 27. Распределение коллизий в адресном пространстве таблицы

Приведем обзор и анализ некоторых наиболее простых из применяемых на практике хеш-функций.

Простейшей хеш-функцией является деление по модулю числового значения ключа на размер пространства записи. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция плохо соответствует первым трем требованиям к хеш-функции и сама по

себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей.

Следующей хеш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Еще одной хеш-функцией можно назвать функцию свертки. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся какие-то арифметические или поразрядные логические операции, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ.

В качестве хеш-функции также применяют функцию преобразования системы счисления. Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления $Q > P$. Обычно выбирают $Q = P+1$. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

2.3.2.2. Открытое хеширование

Основная идея базовой структуры при открытом (внешнем) хешировании (рис. 28) заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для B классов, пронумерованных от 0 до $B - 1$, строится хеш-функция $h(x)$ такая, что для любого элемента x исходного множества функция $h(x)$ принимает целочисленное значение из интервала 0, ..., $B - 1$, соответствующее, естественно, классу, которому принадлежит элемент x. Часто «классы» называют сегментами, поэтому будем говорить, что элемент x принадлежит сегменту $h(x)$.

Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0, 1, ..., $B - 1$, содержит заголовки для B списков. Элемент x i-го списка – это элемент исходного множества, для которого $h(x) = i$.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если можно оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один-два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N (или, что эквивалентно, от B).

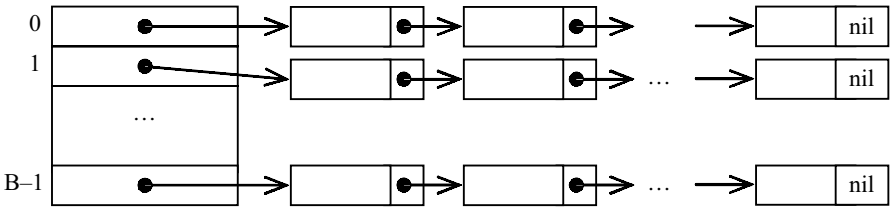


Рис. 28. Организация данных при открытом хешировании

Рассмотрим алгоритмы основных операций с хеш-таблицей, при открытом хешировании. В этих алгоритмах будем использовать структуры данных и операции с линейными однонаправленными списками, которые были описаны в 1.2.6.1. Поле Data в элементах списка будет здесь исполнять роль ключа, а роль указателя на список ptrHead будет играть элемент хеш-таблицы:

```
Const
  B = {подходящая константа};
type
  HtableOpen = array[0..B-1] of Pelement;
```

Кроме того, здесь используется предопределенная функция $h(x)$, которая и является собственно реализацией хеш-функции:

```
Procedure Clear_HtableOpen(var A: HtableOpen);
{Процедура очистки хеш-таблицы}
var
  IndexSeg: integer;
begin
  for IndexSeg:=0 to B-1 do
    while A[IndexSeg] <> nil do
      Del_LineSingleList(A[IndexSeg], A[IndexSeg]);
```

```
end;
function Find_HtableOpen(x: TypeData;
                         var A: HtableOpen;
                         var current: Pelement): boolean;
{функция поиска элемента x в хеш-таблице. Принимает значение
true, если найден и возвращает указатель, который устанавли-
вается на найденный элемент, или принимает значение false и
возвращает nil}
var
  IndexSeg: integer;  {номер сегмента}
begin
  IndexSeg := h(x);
  {начало списка, в котором надо искать, это A[IndexSeg]}
  if Find_LineSingleList(x, A[IndexSeg], current) then
    Find_HtableOpen := true
  else
    Find_HtableOpen := false;
end;
procedure Add_HtableOpen(x: TypeData;  var A: HtableOpen);
{Процедура добавления элемента x в хеш-таблицу}
var
  IndexSeg: integer;  {номер сегмента}
  current: Pelement;
begin
  if not Find_HtableOpen(x, A, current) then begin
  {Если в таблице элемент уже есть, то добавлять не надо}
    IndexSeg := h(x);
    {Добавляем всегда в начало списка}
    InsFirst_LineSingleList(x, A[IndexSeg]);
  end
end;
procedure Del_HtableOpen(x: TypeData;  var A: HtableOpen);
{Процедура удаления элемента x из хеш-таблицы}
var
  IndexSeg: integer;  {номер сегмента}
  current: Pelement;
begin
  if Find_HtableOpen(x, A, current) then begin
  {Если в таблице элемент еще есть, то удаляем}
    IndexSeg := h(x);
    Del_LineSingleList(A[IndexSeg], current);
```

```
  end;
end;
```

Если есть B сегментов и N элементов, хранящихся в хеш-таблице, то каждый сегмент в среднем будет иметь N/B элементов и можно ожидать, что операции добавления, поиска и удаления будут выполняться в среднем за время, пропорциональное $O(1+N/B)$. Здесь константа 1 соответствует поиску сегмента, а N/B – поиску элемента в сегменте. Если B примерно равно N, то время выполнения операторов становится константой, независящей от N.

2.3.2.3. Закрытое хеширование

При закрытом (внутреннем) хешировании в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом хешировании применяется методика повторного хеширования. Если осуществляется попытка поместить элемент x в сегмент с номером $h(x)$, который уже занят другим элементом (такая ситуация называется коллизией), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов $h_1(x)$, $h_2(x)$, ..., куда можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент x добавить нельзя.

При поиске элемента x необходимо просмотреть все местоположения $h(x)$, $h_1(x)$, $h_2(x)$, ..., пока не будет найден x или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. И пусть, для определенности, $h_3(x)$ – первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах $h_4(x)$, $h_5(x)$ и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента $h_3(x)$.

Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента x, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента x. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем deleted (удаленный). В качестве альтерна-

тивы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы deleted и empty – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой deleted, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Существует несколько методов повторного хеширования, т. е. определения местоположений $h(x)$, $h_1(x)$, $h_2(x)$, ... :

– линейное опробование;

– квадратичное опробование;

– двойное хеширование.

Линейное опробование (рис. 29, *а*) сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом:

$$\text{адрес} = h(x) + c \cdot i \ ,$$

где i – номер попытки разрешить коллизию; c – константа, определяющая шаг перебора. При шаге, равном единице, происходит последовательный перебор всех сегментов после текущего.

Квадратичное опробование (рис. 29, *б*) отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

$$\text{адрес} = h(x) + c \cdot i + d \cdot i^2,$$

где i – номер попытки разрешить коллизию, c и d – константы.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов.

Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется *двойным хешированием* (рис. 29, *в*), основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций

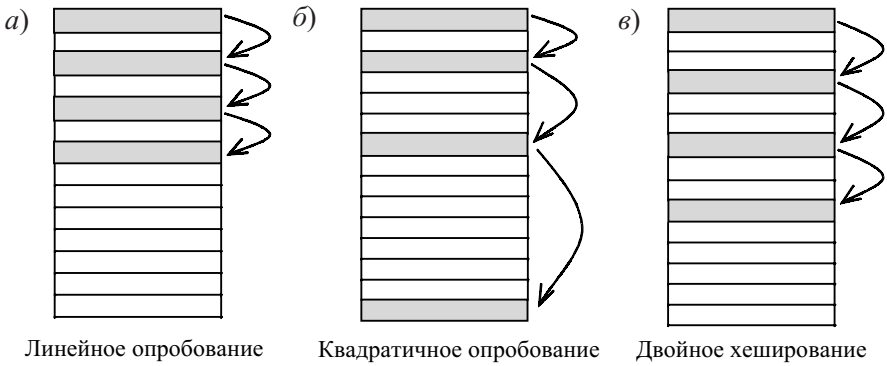


Рис. 29. Организация данных при закрытом хешировании

$$\text{адрес} = h_1(x) + i \cdot h_2(x).$$

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией. С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

Однако в случае многократного превышения адресного пространства и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные сегменты. Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает вероятность нахождения свободных сегментов.

В алгоритмах операций очистки, вставки, поиска и удаления для хеш-таблицы с использованием линейного опробования будем использовать структуры данных:

```
Const
  B = {подходящая константа};
```

```
  empty = '          ';    {10 пробелов}
  deleted = '**********';  {10 символов *}
  c = 1;  {шаг перебора}
  MaxCase = {подходящая константа};  {Max количество попыток}
type
  TypeElem = string[10]
  HTableClose = array[0..B-1] of TypeElem;
```

Теперь приведем сами алгоритмы на примере линейного опробования. Для остальных методов повторного хеширования алгоритмы идентичны:

```
procedure Clear_HTableClose(var A: HTableClose);
{Процедура очистки хеш-таблицы}
var
  IndexSeg: integer;
begin
  for IndexSeg:=0 to B-1 do A[IndexSeg] := empty;
end;
function Find_HTableClose(x: TypeElem;
                          var A: HTableClose;
                          var IndexSeg: integer): boolean;
{функция поиска элемента x в хеш-таблице. Принимает значение
true, если найден элемент, и возвращает номер сегмента, в
котором располагается найденный элемент, или принимает значение
false и возвращает 0}
var
  i: integer;
begin
  i := 0;
  repeat
    IndexSeg := ((h(x) + c*i) mod B +  {лин.опр.с цикл.переходом}
                 (h(x) + c*i) div B  ) {смещение после перехода}
                mod B;                 {ограничение смещения}
    i := i + 1;
  until (A[IndexSeg] = x) or         {нашли}
        (A[IndexSeg] = empty) or     {дальше уже нет}
        (i > MaxCase);               {слишком долго ищем}
  if A[IndexSeg] = x then begin
    Find_HTableClose := true;
  end else begin
    Find_HTableClose := false;
    IndexSeg := 0;
```

```
  end;
end;
function Add_HTableClose(x: TypeElem;
                         var A: HTableClose): boolean;
{Процедура добавления элемента x в хеш-таблицу. Возвращает true,
если элемент добавлен, и false - в обратном}
var
  i,
  IndexSeg: integer;  {номер сегмента}
begin
  if not Find_HTableClose(x, A, IndexSeg) then begin
  {Если в таблице элемент уже есть, то добавлять не надо}
    i := 0;
    repeat
      IndexSeg := ((h(x) + c*i) mod B +
                   (h(x) + c*i) div B  )
                  mod B;
      i := i + 1;
    until (A[IndexSeg] = empty) or    {нашли место}
          (A[IndexSeg] = deleted) or  {тоже можно занять}
          (i > MaxCase);              {слишком долго ищем}
    if (A[IndexSeg] = empty) or
       (A[IndexSeg] = deleted) then begin
      A[IndexSeg] := x;
      Add_HTableClose := true;
    end else begin
      Add_HTableClose := false;
    end;
  end
end;
procedure Del_HTableClose(x: TypeElem;  var A: HTableClose);
{Процедура удаления элемента x из хеш-таблицы}
var
  IndexSeg: integer;  {номер сегмента}
begin
  if Find_HTableClose(x, A, IndexSeg) then begin
  {Если в таблице элемент уже нет, то удалять не надо}
    A[IndexSeg] := deleted;
  end
end;
```

В случае применения схемы закрытого хеширования скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования (опробования) для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты. Например, методика линейного опробования для разрешения коллизий – не самый лучший выбор.

Как только несколько последовательных сегментов будут заполнены (образуя группу), любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операций.

Определим, сколько необходимо сделать проб (проверок) на заполненность сегментов при вставке нового элемента, предполагая, что в хеш-таблице, состоящей из B сегментов, уже находится N элементов и все комбинации расположения N элементов в B сегментах равновероятны.

В этом случае вероятность коллизий равна B/N. Не приводя дальнейших доказательств, отметим, что среднее число проб на один сегмент при заполнении M сегментов равно $(B/M)\ln(B/(B - M + 1))$. Таким образом, для полного заполнения таблицы ($M = B$) требуется в среднем $\ln B$ проб на один сегмент, или всего $B \cdot \ln B$ проб.

При поиске элемента, которого нет в таблице, требуется в среднем такое же число проб, как и при вставке нового элемента при данном заполнении. Поиск существующего элемента требует в среднем столько же проб, сколько необходимо для вставки всех элементов, сделанных до настоящего времени. Удаление требует в среднем столько же проб, сколько и поиск элемента. Здесь следует отметить, что, в отличие от открытого хеширования, удаление элемента из закрытой хеш-таблицы не ускоряет процесс вставки нового элемента или его поиска.

2.3.2.4. Реструктуризация хеш-таблиц

При использовании открытых хеш-таблиц среднее время выполнения операторов возрастает с ростом параметра N/B и особенно быстро

растет при превышении числа элементов над числом сегментов. Подобным образом среднее время выполнения операций также возрастает с увеличением параметра N/B и для закрытых хеш-таблиц (но превышение N над B здесь невозможно).

Чтобы сохранить постоянное время выполнения операторов, которое теоретически возможно при использовании хеш-таблиц, можно предложить при достижении N достаточно больших значений, например при $N \geq 0{,}9B$ для закрытых хеш-таблиц и $N \geq 2B$ – для открытых хеш-таблиц, просто создавать новую хеш-таблицу с удвоенным числом сегментов. Перезапись текущих элементов множества в новую хеш-таблицу в среднем займет меньше времени, чем их ранее выполненная вставка в старую хеш-таблицу меньшего размера. Кроме того, затраченное время на перезапись компенсируется более быстрым выполнением всех операций.

2.3.4. Поиск по вторичным ключам

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Такие ключи называются п е р в и ч н ы м и . Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются в т о р и ч н ы м и ключами.

Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные. Например, поисковые системы InterNet часто организованы как наборы записей, соответствующих Web-страницам. В качестве вторичных ключей для поиска выступают ключевые слова страниц, а сама задача поиска сводится к выборке из таблицы некоторого множества записей, содержащих требуемые вторичные ключи.

2.3.3.1. Инвертированные индексы

Рассмотрим метод организации таблицы с инвертированными индексами (рис. 30). Для таблицы строится отдельный набор данных, содержащий так называемые и н в е р т и р о в а н н ы е индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ.

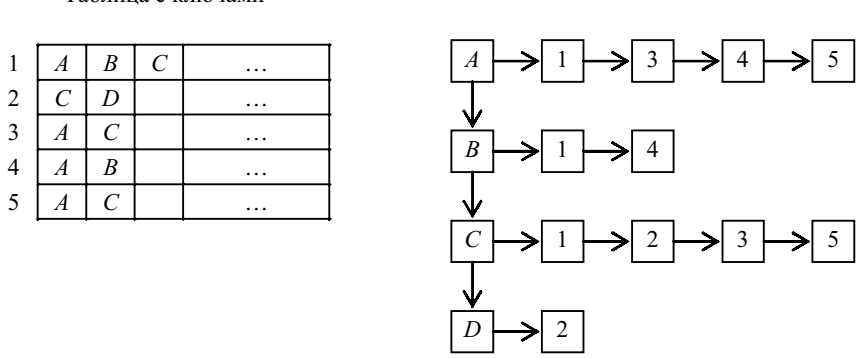


Рис. 30. Организация инвертированных индексов

Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов, является относительно небольшой по сравнению с другими методами организации таблиц.

Недостатками данной системы являются большие затраты времени на составление вспомогательной структуры данных и ее обновление. Причем эти затраты возрастают с увеличение объема базы данных.

Система инвертированных индексов является чрезвычайно удобной и эффективной при организации поиска в больших таблицах.

2.3.3.2. Битовые карты

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт (рис. 31). Для каждого значения вторичного ключа записей основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а нуль – отсутствие.

Основным преимуществом такой организации является очень простая и эффективная организация обработки сложных запросов, которые могут объединять значения ключей различными логическими предикатами. В этом случае поиск сводится к выполнению логических операций запроса непосредственно над битовыми строками и интерпрета-

Таблица с ключами

1	*A*	*B*	*C*	…
2	*C*	*D*		…
3	*A*	*C*		…
4	*A*	*B*		…
5	*A*	*C*		…

Битовая карта

	1	2	3	4	5
A	1	0	1	1	1
B	1	0	0	1	0
C	1	1	1	0	1
D	0	1	0	0	0

Рис. 31. Организация битовых карт

ции результирующей битовой строки. Другим преимуществом является простота обновления карты при добавлении записей.

К недостаткам битовых карт следует отнести увеличение длины строки карты пропорционально длине таблицы. При этом заполненность карты единицами уменьшается с увеличением длины файла. Для таблицы большой длины и редко встречающихся ключей битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.

2.3.4. Использование деревьев в задачах поиска

2.3.4.1. Упорядоченные деревья поиска

Обычные деревья не дают выигрыша при хранении множества значений. При поиске элемента все равно необходимо просмотреть все дерево. Однако можно организовать хранение элементов в дереве так, чтобы при поиске элемента достаточно было просмотреть лишь часть дерева. Для этого надо ввести следующее требование упорядоченности дерева.

Двоичное дерево у п о р я д о ч е н о , если для любой вершины *x* справедливо такое свойство: все элементы, хранимые в левом поддереве, меньше элемента, хранимого в *x*, а все элементы, хранимые в правом поддереве, больше элемента, хранимого в *x*.

Важное свойство упорядоченного дерева: все элементы его различны. Если в дереве встречаются одинаковые элементы, то такое дерево является частично упорядоченным.

В дальнейшем будет идти речь только о двоичных упорядоченных деревьях, опуская слово «упорядоченный».

Итак, основными операциями, производимыми с упорядоченным деревом, являются:

– поиск вершины;
– добавление вершины;
– удаление вершины;
– очистка дерева.

Реализацию этих операций приведем в виде соответствующих процедур.

Алгоритм поиска можно записать в рекурсивном виде. Если искомое значение `Item` меньше `Tree^.Data`, то поиск продолжается в левом поддереве, если равен – поиск считается успешным, если больше – поиск продолжается в правом поддереве; поиск считается неудачным, если достигли пустого поддерева, а элемент найден не был.

```
function Find_Tree(Item: TypeElement; Tree: PTree): boolean;
{Поиск вершины дерева, содержащую значение Item}
var
  Current: PTree;
begin
  Find_Tree := False;
  if Tree <> nil then begin  {Дерево не пустое}
    Current := Tree;
    if Current^.Data = Item then  {Вершина найдена}
      Find_Tree := True
    else
      if Current^.Data > Item then  {Ищем в левом поддереве}
        Find_Tree := Find_Tree(Item, Current^.Left)
      else  {Ищем в правом поддереве}
        Find_Tree := Find_Tree(Item, Current^.Right);
  end;
end;
```

Можно написать аналогичную нерекурсивную функцию. Она позволит избежать избыточного хранения информации. Каждый рекурсивный вызов размещает в стеке локальные переменные `Item` и `Tree`, а также адрес точки возврата из подпрограммы. В нерекурсивном варианте можно обойтись одной переменной `Item` и одной переменной `Tree`.

Как осуществлять нерекурсивный поиск в упорядоченном дереве, рассмотрим на примере *алгоритма добавления элемента* в дерево. Сначала надо найти вершину, к которой в качестве потомка необходимо добавить новую вершину (фактически произвести поиск), а затем присоединить к найденной новую вершину, содержащую значение `Item`

(процедура написана в предположении, что добавляемого элемента в дереве нет):

```
procedure Add_Tree(Item: TypeElement; var Tree: PTree);
{Добавление в дерево вершины со значением Item}
var
  NewNode, Current: PTree;
begin
  if Tree = nil then begin  {Дерево пустое}
  {Создаем корень}
    New(Tree);
    Tree^.Data := Item;
    Tree^.Left := nil;
    Tree^.Right := nil;
  end else begin
    Current := Tree;
    {Поиск вершины}
    while ((Item > Current^.Data) and (Current^.Right <> nil)) or
          ((Item < Current^.Data) and (Current^.Left <> nil)) do
      if Item > Current^.Data then
        Current := Current^.Right
      else
        Current := Current^.Left;
    {Создание новой вершины}
    New(NewNode);
    NewNode^.Data := Item;
    NewNode^.Left := nil;
    NewNode^.Right := nil;
    If Item > Current^.Data then  {Новая вершина больше найденной}
      Current^.Right := NewNode   {Присоединяем новую справа}
    else                          {Новая вершина меньше найденной}
      Current^.Left := NewNode;   {Присоединяем новую слева}
  end;
end;
```

Алгоритм удаления элемента будет несколько сложнее. При удалении может случиться, что удаляемый элемент находится не в листе, т. е. вершина имеет ссылки на реально существующие поддеревья. Эти поддеревья терять нельзя, а присоединить два поддерева на одно освободившееся после удаления место невозможно. Поэтому необходимо поместить на освободившееся место либо самый правый элемент из левого поддерева, либо самый левый из правого поддере-

ва. Нетрудно убедиться, что упорядоченность дерева при этом не нарушится. Договоримся, что будем заменять самый левый элемент из правого поддерева.

Нельзя забывать, что при замене вершина, на которую производится замена, может иметь правое поддерево. Это поддерево необходимо поставить вместо перемещаемой вершины:

```
procedure Del_Tree(Item: TypeElement; var Tree: PTree);
{Удаление из дерева вершины со значением Item}
var
  Parent, Cur, Cur2: PTree;
  Direction: (L, R); {направление движения (влево/вправо)}
begin
  {Поиск удаляемого элемента}
  Parent := nil; {предок удаляемой вершины}
  Cur := Tree;
  while ((Item > Cur^.Data) and (Cur^.Right <> nil)) or
        ((Item < Cur^.Data) and (Cur^.Left <> nil)) do begin
    Parent := Cur;
    if Item > Cur^.Data then begin
      Cur := Cur^.Right; Direction := R;
    end else begin
      Cur := Cur^.Left; Direction := L;
    end;
  end;
  if Cur <> nil then begin   {Вершина найдена}
    {Анализируем наличие потомков у найденной вершины}
    {и удаляем соответствующим образом}
    if (Cur^.Left <> nil) and (Cur^.Right <> nil) then begin
      {Удаление вершины в случае наличия у нее обоих потомков}
      Parent := Cur; Cur2 := Cur^.Right;
      {Поиск кандидатуры на замену}
      while Cur2^.Left <> nil do begin
        Parent := Cur2; Cur2 := Cur2^.Left;
      end;
      {Заменяем}
      Cur^.Data := Cur2^.Data;
      {Спасаем правое поддерево вершины, которой заменяем}
      if Parent <> Cur then
        Parent^.Left := Cur2^.Right
      else
```

```
        Parent^.Right := Cur2^.Right;
      Dispose(Cur2);
    end else begin
      {Удаление вершины в случае наличия у нее}
      {не более одного потомка}
      if (Cur^.Left = nil) then begin
        if Parent <> nil then begin
          if Direction = L then
            Parent^.Left := Cur^.Right
          else
            Parent^.Right := Cur^.Right;
        end else begin
          Tree := Cur^.Right;
        end;
      end;
      if (Cur^.Right = nil) then begin
        if Parent <> nil then begin
          if Direction = L then
            Parent^.Left := Cur^.Left
          else
            Parent^.Right := Cur^.Left;
        end else begin
          Tree := Cur^.Left;
        end;
      end;
      Dispose(Cur);
    end;
  end;
end;
```

Временная сложность этих алгоритмов (она одинакова для этих алгоритмов, так как в их основе лежит поиск) оценим для наилучшего и наихудшего случая. В лучшем случае, т. е. случае полного двоичного дерева, получаем сложность $O_{\min}(\log n)$. В худшем случае дерево может выродиться в список. Такое может произойти, например, при добавлении элементов в порядке возрастания. При работе со списком в среднем придется просмотреть половину списка. Это даст сложность $O_{\max}(n)$.

В *алгоритме очистки дерева* применяется обратный метод обхода дерева. Использование обратного метода обхода гарантирует, что будут сначала посещены и удалены все потомки предка, прежде чем удалится сам предок:

```
Procedure Clear_Tree(var Tree: Ptree);
{Очистка дерева}
begin
  if Tree <> nil then begin
    Clear_Tree(Tree^.Left);
    Clear_Tree(Tree^.Right);
    Dispose(Tree);
    Tree := nil;
  end;
end;
```

Временная сложность этой операции $O(n)$.

2.3.4.2. Случайные деревья поиска

Случайные деревья поиска представляют собой упорядоченные бинарные деревья поиска, при создании которых элементы (их ключи) вставляются в случайном порядке.

При создании таких деревьев используется тот же алгоритм, что и при добавлении вершины в бинарное дерево поиска. Будет ли созданное дерево случайным или нет, зависит от того, в каком порядке поступают элементы для добавления. Примеры различных деревьев, создаваемых при различном порядке поступления элементов, приведены ниже.

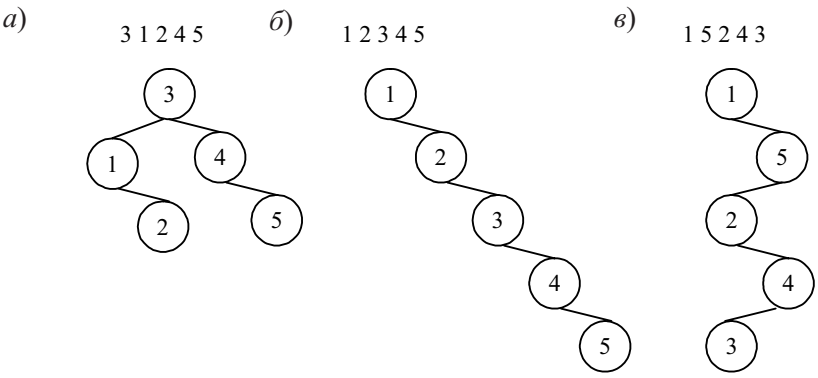


Рис. 32. Случайные и вырожденные деревья поиска

При поступлении элементов в случайном порядке получаем дерево с минимальной высотой h (рис. 32, *а*), а соответственно миними-

зируется время поиска элемента в таком дереве, которое пропорционально $O(\log n)$. При поступлении элементов в упорядоченном виде (рис. 32, *б*) или в несколько необычном порядке (рис. 32, *в*) происходит построение вырожденных деревьев поиска (оно вырождено в линейный список), что нисколько не сокращает время поиска, которое составляет $O(n)$.

2.3.4.3. Оптимальные деревья поиска

В двоичном дереве поиск одних элементов может происходить чаще, чем других, т. е. существуют вероятности p_k поиска k-го элемента и для различных элементов эти вероятности неодинаковы. Можно сразу предположить, что поиск в дереве в среднем будет более быстрым, если те элементы, которые ищут чаще, будут находиться ближе к корню дерева.

Пусть даны $2n+1$ вероятностей $p_1, p_2, \ldots, p_n, q_0, q_1, \ldots, q_n$, где p_i – вероятность того, что аргументом поиска является K_i; q_i – вероятность того, что аргумент поиска лежит между K_i и K_{i+1}; q_0 – вероятность того, что аргумент поиска меньше, чем K_1; q_n – вероятность того, что аргумент поиска больше, чем K_n.

Тогда цена дерева поиска C будет определяться следующим образом:

$$C = \sum_{j=1}^{n} p_j \left(\text{levelroot}_j + 1\right) + \sum_{k=1}^{n} q_k \left(\text{levellist}_k\right),$$

где levelroot_j – уровень узла j, а levellist_k – уровень листа k.

Дерево поиска называется оптимальным, если его цена минимальна или, другими словами, оптимальное бинарное дерево поиска – это бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных.

Существует подход построения оптимальных деревьев поиска, при котором элементы вставляются в порядке уменьшения частот, что дает в среднем неплохие деревья поиска. Однако этот подход может дать вырожденное дерево поиска (см. 2.3.4.2), которое будет далеко от оптимального.

Еще один подход состоит в выборе корня k таким образом, чтобы максимальная сумма вероятностей для вершин левого поддерева или правого поддерева была настолько мала, насколько это возможно. Та-

кой подход также может оказаться плохим в случае выбора в качестве корня элемента с малым значением p_k.

Существуют алгоритмы, которые позволяют построить оптимальное дерево поиска. К ним относится, например, алгоритм Гарсия-Воча. Однако такие алгоритмы имеют временную сложность порядка $O(n^2)$, а некоторые еще имеют такую же пространственную сложность. Таким образом, создание оптимальных деревьев поиска требует больших накладных затрат, что не всегда оправдывает выигрыш при быстром поиске.

2.3.4.4. Сбалансированные по высоте деревья поиска

Как уже говорилось ранее (см. 2.3.4.2), в худшем случае (дерево вырождено в линейный список) хранение данных в упорядоченном бинарном дереве никакого выигрыша в сложности операций (поиск/добавление/удаление) по сравнению с массивом или линейным списком не дает. В лучшем случае (дерево сбалансировано) для всех операций получается логарифмическая сложность, что гораздо лучше.

Идеально сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1. Однако идеальную сбалансированность довольно трудно поддерживать. В некоторых случаях при добавлении/удалении может потребоваться значительная перестройка дерева, не гарантирующая логарифмической сложности. Поэтому в 1962 году два советских математика Г. М. Адельсон-Вельский и Е. М. Ландис ввели менее строгое определение сбалансированности и доказали, что при таком определении можно написать программы добавления/удаления, имеющие логарифмическую сложность и сохраняющие дерево сбалансированным.

Дерево считается *сбалансированным по АВЛ* (в дальнейшем просто «сбалансированным»), если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1. Не всякое сбалансированное дерево идеально сбалансировано, но всякое идеально сбалансированное дерево сбалансировано.

При операциях добавления и удаления может произойти нарушение сбалансированности дерева. В этом случае потребуются некоторые преобразования, не нарушающие упорядоченности дерева и способствующие лучшей сбалансированности.

Рассмотрим такие преобразования.

Пусть вершина a имеет правый потомок b. Обозначим через P левое поддерево вершины a, через Q и R – левое и правое поддеревья вершины b. Упорядоченность дерева требует, чтобы $P < a < Q < b < R$. Точно того же требует упорядоченность дерева с корнем b, его левым потомком a, в котором P и Q – левое и правое поддеревья a, R – правое поддерево b. Поэтому первое дерево можно преобразовать во второе, не нарушая упорядоченности. Такое преобразование называется малым правым вращением (рис. 33). Аналогично определяется симметричное ему малое левое вращение.

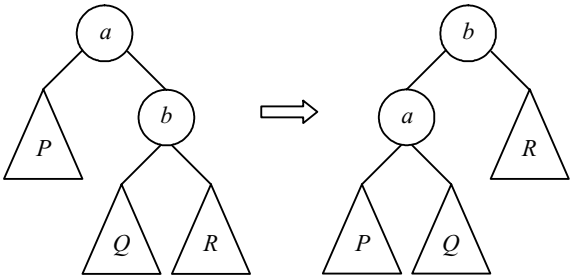


Рис. 33. Малое правое вращение сбалансированного дерева

Пусть b – правый потомок a, c – левый потомок b, P – левое поддерево a, Q и R – левое и правое поддеревья c, S – правое поддерево b. Тогда $P < a < Q < c < R < b < S$. Такой же порядок соответствует дереву с корнем c, имеющим левый потомок a и правый потомок b, для которых P и Q – поддеревья вершины a, а R и S – поддеревья вершины b. Соответствующее преобразование будем называть большим правым вращением (рис. 34). Аналогично определяется симметричное ему большое левое вращение.

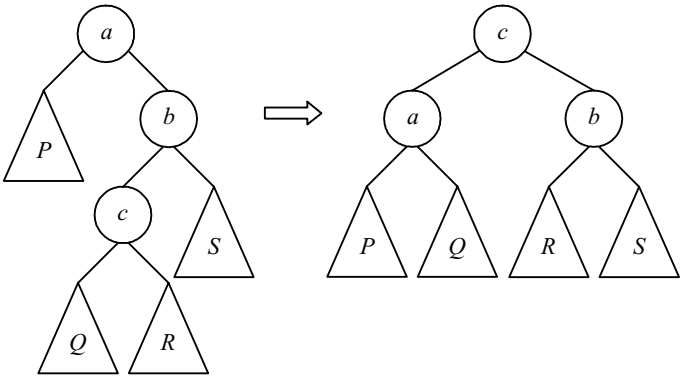


Рис. 34. Большое правое вращение сбалансированного дерева

Теперь приведем алгоритм балансировки на языке Паскаль. В этом алгоритме опущены левые вращения (большое и малое), которые записываются симметрично.

Предварительно договоримся, что в каждой вершине дерева помимо значения элемента будем хранить показатель сбалансированности в данной вершине. Показатель сбалансированности вершины – это разница между высотами правого и левого поддеревьев данной вершины:

```
type
  PTree = ^TTree;
  TTree = record
    Data: TypeElement;    {поле данных}
    Left, Right: PTree;   {указатели на левое и правое подде-
ревья}
    Balance: integer;     {показатель сбалансированности}
  end;
```

В сбалансированном дереве показатели сбалансированности всех вершин лежат в пределах от –1 до 1. При операциях добавления/удаления могут появляться вершины с показателями сбалансированности –2 и 2:

```
procedure TreeBalance(a: PTree; var CurBalance: integer);
{Балансировка двоичного дерева}
procedure SmallRightRotation(a: PTree);
{малое правое вращение поддерева с корнем a}
var
  b: PTree;
  val_a, val_b: TypeElement;
  h_P, h_Q, h_R: integer;
begin
  b := a^.Right;  {b <> null}
  val_a := a^.Data; val_b := b^.Data;
  h_Q := 0;
  h_R := b^.Balance;
  h_P := (max(h_Q, h_R) + 1)- a^.Balance;
  a^.Data := val_b;  b^.Data := val_a;
  a^.Right := b^.Right;  {поддерево R}
  b^.Right := b^.Left;   {поддерево Q}
  b^.Left  := a^.Left;   {поддерево P}
  a^.Left := b;
  b^.Balance := h_Q - h_P;
  a^.Balance := h_R - (max(h_P, h_Q) + 1);
```

```
end;
procedure BigRightRotation(a: PTree);
{большое правое вращение поддерева с корнем a}
var
  b, c: PTree;
  val_a, val_b, val_c: TypeElement;
  h_P, h_Q, h_R, h_S: integer;
begin
  b := a^.Right;  c := b^.Left;  {b <> null,c <> null}
  val_a := a^.Data;  val_b := b^.Data;  val_c := c^.Data;
  h_Q := 0;  h_R := c^.Balance;
  h_S := (max(h_Q, h_R) + 1) + b^.Balance;
  h_P := 1 + max (h_S, h_S-b^.Balance) - a^.Balance;
  a^.Data := val_c;  c^.Data := val_a;
  b^.Left  := c^.Right;  {поддерево R}
  c^.Right := c^.Left;   {поддерево Q}
  c^.Left  := a^.Left;   {поддерево P}
  a^.Left := c;
  b^.Balance := h_S - h_R;
  c^.Balance := h_Q - h_P;
  a^.Balance := max(h_S, h_R) - max(h_P, h_Q);
end;
begin  {-2 <= a^.Balance <= 2}
  if a^.Balance = 2 then begin
    b := a^.Right;
    if b^.Balance = -1 then begin
      BigRightRotation(a);  CurBalance := -1;
    end else begin
      if b^.Balance = 0 then begin
        SmallRightRotation(a);  CurBalance := 0;
      end else begin  {b^.Balance = 1}
        SmallRightRotation(a);  CurBalance := - 1;
      end;
    end;
  end else begin
    if a^.Balance = -2 then begin
      b := a^.Left;
      if b^.Balance = 1 then begin
        BigLeftRotation(a);  CurBalance := -1;
      end else begin
        if b^.Balance = 0 then begin
```

```
          SmallLeftRotation(a);  CurBalance := 0;
        end else begin {b^.Balance = -1}
          SmallLeftRotation(a);  CurBalance := - 1;
        end;
      end;
    end else begin {-2 < a^.Balance < 2, ничего делать не надо}
      CurBalance := 0;
    end;
  end;
end;
```

Схематично алгоритм добавления нового элемента в сбалансированное дерево будет состоять из следующих трех основных шагов:

1) поиск по дереву.

2) вставка элемента в место, где закончился поиск, если элемент отсутствует.

3) восстановление сбалансированности.

Первый шаг необходим для того, чтобы убедиться в отсутствии элемента в дереве, а также найти такое место вставки, чтобы после вставки дерево осталось упорядоченным. Первые два шага ничем не отличаются от алгоритмов, приведенных в 2.3.4.1.

Третий шаг представляет собой обратный проход по пути поиска: от места добавления к корню дерева. По мере продвижения по этому пути корректируются показатели сбалансированности проходимых вершин, и производится балансировка там, где это необходимо. Добавление элемента в дерево никогда не требует более одного поворота.

Алгоритм удаления элемента из сбалансированного дерева будет выглядеть так:

1) поиск по дереву.

2) удаление элемента из дерева.

3) восстановление сбалансированности дерева (обратный проход).

Первый шаг необходим, чтобы найти в дереве вершину, которая должна быть удалена. Первые два шага аналогичны удалению элемента, приведенному в 2.3.4.1.

Третий шаг представляет собой обратный проход от места, из которого взят элемент для замены удаляемого, или от места, из которого удален элемент, если в замене не было необходимости.

Операция удаления может потребовать перебалансировки всех вершин вдоль обратного пути к корню дерева, т. е. порядка log n вершин.

Таким образом, алгоритмы поиска, добавления и удаления элементов в сбалансированном дереве имеют сложность, пропорциональную $O(\log n)$.

Г. М. Адельсон-Вельский и Е. М. Ландис доказали теорему, согласно которой высота сбалансированного дерева никогда не превысит высоту идеально сбалансированного дерева более, чем на 45%.

2.3.5. Поиск в тексте

Часто приходится сталкиваться со специфическим поиском, так называемым поиском слова. Его можно определить следующим образом. Пусть задан массив `Txt` из N элементов, называемый текстом и массив `Wrd` из M элементов, называемый словом, причем $0 < M \leq N$. Описать их можно как строки.

Поиск слова обнаруживает первое вхождение `Wrd` в `Txt`. Это действие типично для любых систем обработки текстов, отсюда и очевидная заинтересованность в поиске эффективного алгоритма для этой задачи.

2.3.5.1. Прямой поиск

Разберем алгоритм поиска, который будем называть прямым поиском строки.

Данный алгоритм заключается в посимвольном сравнении текста со словом. В начальный момент происходит сравнение первого символа текста с первым символом слова, второго символа текста со вторым символом слова и т. д. Если произошло совпадение всех символов, то фиксируется факт нахождения слова. В противном случае производится «сдвиг» слова на одну позицию вправо и повторяется посимвольное сравнение, т. е. сравнивается второй символ текста с первым символом слова, третий символ текста со вторым символом слова и т. д. (рис. 35). Эти «сдвиги» слова повторяются до тех пор, пока конец слова не достиг конца текста или не произошло полное совпадение символов слова с текстом (т. е. слово найдено):

```
function DirectTxtSearch(var Wrd: TWrd;
                         var Txt: TText;
                         var Position: integer): boolean;
{Функция поиска слова Wrd в тексте Txt,}
{если слово найдено, то возвращает значение true}
{и позицию Position начала первого слова Wrd,}
{иначе – false и Position не изменяется}
```

```
var
  i,              {Индекс начала слова в тексте}
  j: integer;     {Индекс текущего символа слова}
begin
  i := 0;
  repeat
    j := 1;  i := i + 1;
    {Осуществляем посимвольное сравнение}
    while (j <= M) and
          (Txt[i+j-1] = Wrd[j]) do
       j := j+1;
  until (j = M+1) or  {Совпало все слово}
       (i >= N-M+1);  {Конец слова за концом текста}
  {Оценка результатов поиска}
  if j = M+1 then begin
    DirectTxtSearch := true;
    Position := i;
  end else begin
    DirectTxtSearch := false;
  end;
end;
```

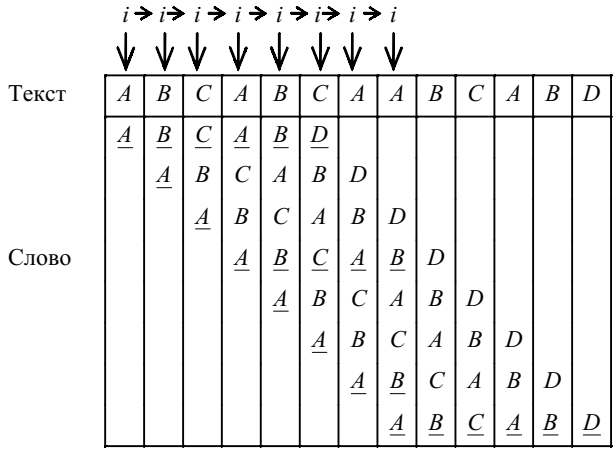


Рис. 35. Алгоритм прямого поиска в тексте

Этот алгоритм работает достаточно эффективно, если допустить, что несовпадение пары символов происходит после незначительного коли-

чества сравнений во внутреннем цикле. При достаточно большом множестве символов это довольно частый случай. Можно предполагать, что для текстов, составленных из 128 символов, несовпадение будет обнаруживаться после одной или двух проверок. Тем не менее, в худшем случае алгоритм будет малоэффективен, так как его сложность будет пропорциональна $O((N - M)\cdot M)$.

2.3.5.2. Алгоритм Кнута, Мориса и Пратта

Приблизительно в 1970 году Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм (КМП-алгоритм), фактически требующий только $O(N)$ сравнений даже в самом плохом случае. Новый алгоритм основывается на том соображении, что после частичного совпадения начальной части слова с соответствующими символами текста фактически известна пройденная часть текста и можно «вычислить» некоторые сведения (на основе самого слова), с помощью которых затем быстро продвинуться по тексту.

Основным отличием КМП-алгоритма от алгоритма прямого поиска является выполнение сдвига слова не на один символ на каждом шаге алгоритма, а на некоторое переменное количество символов. Таким образом, перед тем как осуществлять очередной сдвиг, необходимо определить величину сдвига. Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим.

Если j определяет позицию в слове, содержащую первый несовпадающий символ (как в алгоритме прямого поиска), то величина сдвига Shift определяется как j – LenSuff – 1. Значение LenSuff определяется как размер самой длинной последовательности символов слова, непосредственно предшествующих позиции j (суффикс), которая полностью совпадает с началом слова. LenSuff зависит только от слова и не зависит от текста. Для каждого j будет своя величина Shift, которую обозначим Shift_j.

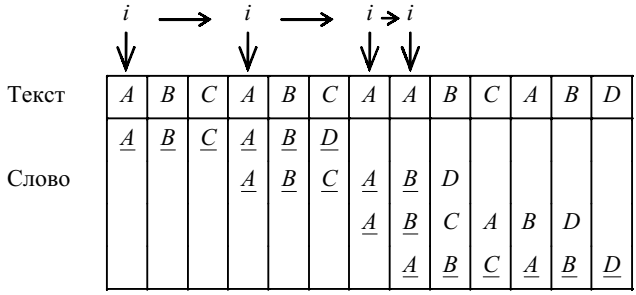


Рис. 36. Пример работы КМП-алгоритма

Приведенный на рис. 36 пример поиска слова ABCABD показывает принцип работы такого алгоритма. Символы, подвергшиеся сравнению, здесь подчеркнуты. Обратите внимание: при каждом несовпадении пары символов слово сдвигается на переменную величину, и меньшие сдвиги не могут привести к полному совпадению.

Так как величины $Shift_j$ зависят только от слова, то перед началом фактического поиска можно вычислить вспомогательную таблицу Shift; эти вычисления сводятся к некоторой предтрансляции слова. Соответствующие усилия будут оправданными, если размер текста значительно превышает размер слова ($M<<N$). Если нужно искать многие вхождения одного и того же слова, то можно пользоваться одной и той же Shift. Приведенные на рис. 37 примеры объясняют назначение Shift.

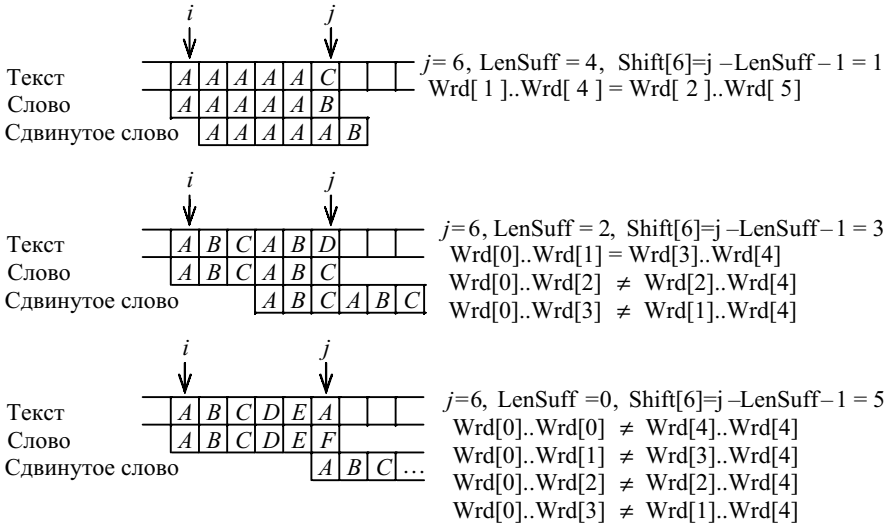


Рис. 37. Частичное совпадение со словом и вычисление $Shift_j$

Следующая функция на языке Паскаль демонстрирует КМП-алгоритм:

```
function KMPTxtSearch(var Wrd: TWrd;
                      var Txt: TTxt;
                      var Position: integer): boolean;
{Функция поиска слова Wrd в тексте Txt,}
{если слово найдено, то возвращает значение true}
{и позицию Position начала первого слова Wrd,}
```

```
{иначе - false и Position не изменяется}
var
  i,                {Индекс начала слова в тексте}
  j,                {Индекс текущ.символа слова}
  k,                {Индекс текущ.символа суффикса слова}
  LenSuff: integer; {Длина суффикса}
  Equal: boolean;   {Признак совпадения суффикса с началом}
  Shift: array[1..M] of integer;          {Массив смещений}
begin
  {Заполнение массива Shift}
  Shift[1] := 1; Shift[2] := 1;  {Для первых двух смещение 1}
  {Вычисляем смещение для остальных M-2 символов слова}
  for j := 3 to M do begin
    Shift[j] := 1;   {Предопределенное значение}
    {Перебираем все возможные длины суффиксов}
    for LenSuff := 1 to j-2 do begin
      Equal:=true;
      {Сравниваем посимвольно суффикс с началом слова}
      for k := 1 to LenSuff do begin
        if Wrd[k] <>
           Wrd[j-LenSuff+k-1]
        then Equal:=false;
      end;
      {Если суффикс совпал, то Shift - это смещение
       от начала слова до начала суффикса}
      if Equal then
        Shift[j] := j - LenSuff - 1;
    end;
  end;
  {Поиск слова Wrd в тексте Txt, аналогичный прямому,
   только смещение не на 1, а на переменный шаг Shift}
  i := 0;  j := 1;   {Начальные значения}
  repeat
    {Смещение слова}
    i := i + Shift[j];
    j := 1;
    {Посимвольное сравнение}
    while (j <= M) and
          (Txt[i+j-1] = Wrd[j]) do
       j := j+1;
  until (j = M+1) or (i >= N-M+1);
```

```
  {Оценка результатов поиска}
  if j = M+1 then begin
    KMPTxtSearch := true;
    Position := i;
  end else begin
    KMPTxtSearch := false;
  end;
end;
```

Точный анализ КМП-алгоритма весьма сложен. Его изобретатели доказывают, что требуется порядка $O(M + N)$ сравнений символов, что значительно лучше, чем $O((N – M)\cdot M)$ сравнений при прямом поиске.

2.3.5.3. Алгоритм Боуера и Мура

КМП-алгоритм дает подлинный выигрыш только тогда, когда неудаче предшествовало некоторое число совпадений. Лишь в этом случае слово сдвигается более чем на единицу. К несчастью, это скорее исключение, чем правило: совпадения встречаются значительно реже, чем несовпадения. Поэтому выигрыш от использования КМП-алгоритма в большинстве случаев поиска в обычных текстах весьма незначителен. Метод же, предложенный Р. Боуером и Д. Муром в 1975 году (БМ-алгоритм), не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях.

БМ-алгоритм основывается на необычном соображении – сравнение символов начинается с конца слова, а не с начала. Как и в случае КМП-алгоритма, перед фактическим поиском на основе слова формируется некоторая таблица. Пусть для каждого символа x из алфавита величина Shift_x – расстояние от самого правого в слове вхождения x до правого конца слова. Представим себе, что обнаружено расхождение между словом и текстом, причем символ в тексте, который не совпал, есть x. В этом случае слово сразу же можно сдвинуть вправо так, чтобы самый правый символ слова, равный x, оказался бы в той же позиции, что и символ текста x. Этот сдвиг, скорее всего, будет на число позиций, большее единицы. Если несовпадающий символ текста x в слове вообще не встречается, то сдвиг становится даже больше: сдвигаем вправо так, чтобы ни один символ слова не накладывался на символ x. На рис. 38 приведен пример, иллюстрирующий этот процесс.

Далее приводится функция на языке Паскаль с упрощенным БМ-алгоритмом, построенная так же, как и предыдущая программа с КМП-алгоритмом:

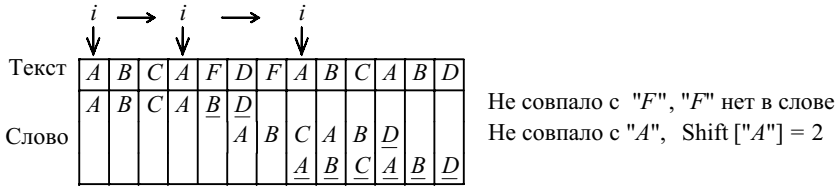


Рис. 38. Пример работы БМ-алгоритма

```
function BMTxtSearch(var Wrd: TWrd;
                     var Txt: TText;
                     var Position: integer): boolean;
{Функция поиска слова Wrd в тексте Txt,}
{если слово найдено, то возвращает значение true}
{и позицию Position начала первого слова Wrd,}
{иначе - false и Position не изменяется}
var
  i,              {Индекс начала слова в тексте}
  j: integer;     {Индекс текущего символа слова}
  ch: char;
  Shift: array[' '..'я'] of integer;    {Массив смещений}
begin
  {Заполнение массива Shift}
  for ch:=' ' to 'я' do Shift[ch] := M;
  for j:=1 to M do Shift[Wrd[j]] := M-j;
  {Поиск слова Wrd в тексте Txt}
  i := 1;    {Начало слова совпадает с началом текста}
  repeat
    j := M+1;  {Сравнивать будем с последнего символа}
    {Посимвольное сравнение слова, начиная с правого символа}
    repeat
      j := j-1;
    until (j < 1) or (Wrd[j] <> Txt[i+j-1]);
    if j >= 1 then
      i := i + (j + Shift[Txt[i+j-1]] - M);  {Сдвиг слова вправо}
  until (j < 1) or (i > N-M+1);  {Оценка результатов поиска}
  if j < 1 then begin
    BMTxtSearch := true;
    Position := i;
  end else begin
```

```
    BMTxtSearch := false;
  end;
end;
```

Почти всегда, кроме специально построенных примеров, данный алгоритм требует значительно меньше $O(N)$ сравнений. В самых же благоприятных обстоятельствах, когда последний символ слова всегда попадает на несовпадающий символ текста, число сравнений пропорционально $O(N/M)$.

Авторы алгоритма приводят и несколько соображений по поводу дальнейших усовершенствований алгоритма. Одно из них – объединить приведенную только что стратегию, обеспечивающую большие сдвиги в случае несовпадения, со стратегией Кнута, Морриса и Пратта, допускающей «ощутимые» сдвиги при обнаружении совпадения (частичного). Такой метод требует двух таблиц, получаемых при предтрансляции: Shift′ – только что упомянутая таблица, а Shift″ – таблица, соответствующая КМП-алгоритму. Из двух сдвигов выбирается больший. Дальнейшее обсуждение этого предмета приводить не будем, поскольку дополнительное усложнение формирования таблиц и самого поиска, возможно, не оправдает видимого выигрыша в производительности. Фактические дополнительные расходы будут высокими и неизвестно, приведут ли все эти ухищрения к выигрышу или проигрышу.

2.4. Алгоритмы кодирования (сжатия) данных

2.4.1. Основные виды сжатия

Сжатие сокращает объем пространства, требуемого для хранения файлов в ЭВМ, и количество времени, необходимого для передачи информации по каналу установленной ширины пропускания. Это есть форма кодирования. Другими целями кодирования являются поиск и исправление ошибок, а также шифрование. Процесс поиска и исправления ошибок противоположен сжатию – он увеличивает избыточность данных, когда их не нужно представлять в удобной для восприятия человеком форме. Удаляя из текста избыточность, сжатие способствует шифрованию, что затрудняет поиск шифра доступным для взломщика статистическим методом.

В этом подразделе рассматривается обратимое сжатие или сжатие без наличия помех, где первоначальный текст может быть в точнос-

ти восстановлен из сжатого состояния. Необратимое или ущербное сжатие используется для цифровой записи аналоговых сигналов, таких как человеческая речь или рисунки. Обратимое сжатие особенно важно для текстов, записанных на естественных и на искусственных языках, поскольку в этом случае ошибки обычно недопустимы. Хотя первоочередной областью применения рассматриваемых методов есть сжатие текстов, однако, эта техника может найти применение и в других случаях, включая обратимое кодирование последовательностей дискретных данных.

Существует много веских причин осуществлять сжатие данных, так как более быстрая передача данных и сокращение пространства для их хранения позволяют сберечь значительные средства и зачастую улучшить показатели ЭВМ. Сжатие, вероятно, будет оставаться в сфере внимания из-за все возрастающих объемов хранимых и передаваемых в ЭВМ данных, кроме того, его можно использовать для преодоления некоторых физических ограничений, таких как, например, сравнительно низкая ширина пропускания телефонных каналов.

Существуют два основных способа проведения сжатия: статистический и словарный. Лучшие статистические методы применяют кодирование Хаффмана, лучшие словарные – метод Зива-Лемпела. В статистическом сжатии каждому символу присваивается код, основанный на вероятности его появления в тексте. Высоковероятные символы получают короткие коды, и наоборот. В словарном методе группы последовательных символов или «фраз» заменяются кодом. Замененная фраза может быть найдена в некотором «словаре». Только в последнее время было показано, что любая практическая схема словарного сжатия может быть сведена к соответствующей статистической схеме сжатия, и найден общий алгоритм преобразования словарного метода в статистический. Поэтому при поиске лучшего сжатия статистическое кодирование обещает быть наиболее плодотворным, хотя словарные методы и привлекательны своей быстротой.

Далее более подробно рассмотрим кодирование Хаффмана.

2.4.2. Метод Хаффмана. Оптимальные префиксные коды

В этом методе при сжатии данных, как уже говорилось выше, каждому символу присваивается оптимальный префиксный код, основанный на вероятности его появления в тексте.

Префиксные коды – это коды, в которых никакое кодовое слово не является префиксом любого другого кодового слова. Эти коды имеют переменную длину.

Префикс, применительно к цепочке *a* – это какая-либо строка *b*, где *a* – конкатенация *bc* для некоторой цепочки *c*.

Оптимальный префиксный код – это префиксный код, имеющий минимальную среднюю длину.

Алгоритм Хаффмана можно разделить на два этапа:

1) определение вероятности появления символов в файле;

2) нахождение оптимального префиксного кода.

На первом этапе необходимо прочитать файл полностью и подсчитать вероятности появления символов в файле (иногда подсчитывают, сколько раз встречается каждый символ). Если при этом учитываются все 256 символов, то не будет разницы в сжатии текстового или файла иного формата.

Далее находятся два символа *a* и *b* с наименьшими вероятностями появления и заменяются одним фиктивным символом *x*, который имеет вероятность появления, равную сумме вероятностей появления символов *a* и *b*. Затем, используя эту процедуру рекурсивно, находится оптимальный префиксный код для меньшего множества символов (где символы *a* и *b* заменены одним символом *x*). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 или 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа *a* будет соответствовать коду *x* с добавленным нулем перед этим кодом, а для символа *b* перед кодом символа *x* будет добавлена единица.

Можно рассматривать префиксные коды как пути в двоичном дереве: прохождение от узла к его левому потомку соответствует 0 в коде, а к правому потомку – 1. Если пометить листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева.

2.4.3. Кодовые деревья

Рассмотрим реализацию алгоритма Хаффмана с использованием кодовых деревьев.

Кодовое дерево – это бинарное дерево, у которого:

– листья помечены символами, для которых разрабатывается кодировка;

– узлы (в том числе корень) помечены суммой вероятностей появления всех символов, соответствующих листьям поддерева, корнем которого является соответствующий узел.

Существует два подхода к построению кодового дерева: от корня к листьям и от листьев к корню. Рассмотрим первый подход в виде процедуры Паскаля. Входными параметрами этой процедуры являются массив используемых символов, отсортированный в порядке убывания вероятности появления символов (вначале идут символы с максимальными вероятностями, а в конце – с минимальными), а также указатель на создаваемое кодовое дерево, описание которого идентично описанию бинарного дерева в виде списков (см. 1.3.4.4), за исключением того, что поле Data здесь принимает символьный тип.

```
procedure CreateCodeTable(UseSimbol: TUseSimbol;
          var CodeTree: PTree);
{Процедура создания кодового дерева по методу Хаффмана}
var
  Current: PTree;
  SimbolCount: integer;
begin
  {Создаем корень кодового дерева}
  new(CodeTree);
  Current := CodeTree;
  Current^.Left := nil; Current^.Right := nil;
  SimbolCount := 1;
  {Создаем остальные вершины}
  while SimbolCount <= n-1 do begin
    {Создаем лист с символом в виде левого потомка}
    new(Current^.Left);
    Current^.Left^.Data := UseSimbol[SimbolCount];
    Current^.Left^.Left := nil; Current^.Left^.Right := nil;
    {Создаем следующий узел в виде правого потомка}
    new(Current^.Right);
    Current := Current^.Right;
    Current^.Left := nil; Current^.Right := nil;
    SimbolCount := SimbolCount + 1;
  end;
  {Последний узел превращаем в лист с символом}
  Current^.Data := UseSimbol[SimbolCount];
end;
```

Пример построения кодового дерева приведен на рис. 39.

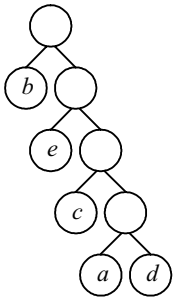
Исходная последовательность символов: aabbbbbbbbcccdeeeee

Исходный объем: 20 байт (160 бит)

Вероятности появления символов

Символ	Вероятность
a	0,10
b	0,40
c	0,15
d	0,05
e	0,25

Кодовое дерево



Оптимальный префиксный код

Символ	Код
a	1110
b	0
c	110
d	1111
e	10

Сжатый код: 11101110000000001101101101111101010101 0

Объем сжатого кода: 37 бит (~ 24,5% исходного)

Рис. 39. Создание оптимальных префиксных кодов

Созданное кодовое дерево может быть использовано для кодирования и декодирования текста. Как осуществляется кодирование уже говорилось выше. Теперь рассмотрим процесс декодирования. Алгоритм распаковки кода можно сформулировать следующим образом:

```
procedure DecodeHuffman(CodeTree: PTree; var ResultStr:
string);
{Процедура декодирования по методу Хаффмана из некоторой битовой}
{последовательности в результирующую строку ResultStr}
var
  Current: PTree;          {Указатель в дереве}
  CurrentBit,              {Значение текущего бита кода}
  CurrentSimbol: integer;{Индекс распаковываемого символа}
  FlagOfEnd: boolean;      {Флаг конца битовой последовательности}
begin
  {Начальная инициализация}
  FlagOfEnd := false; CurrentSimbol := 1; Current := CodeTree;
  {Пока не закончилась битовая последовательность}
  while not FlagOfEnd do begin
    {Пока не пришли в лист дерева}
    while (Current^.Left <> nil) and
          (Current^.Right <> nil) and
```

```
          not FlagOfEnd do begin
        {Читаем значение очередного бита}
        CurrentBit := ReadBynary(FlagOfEnd);
        {Бит – 0, то идем налево, бит – 1, то направо}
        if CurrentBit = 0 then Current := Current^.Left
                          else Current := Current^.Right;
      end;
      {Пришли в лист и формируем очередной символ}
      ResultStr[CurrentSimbol] := Current^.Data;
      CurrentSimbol := CurrentSimbol + 1;
      Current := CodeTree;
    end;
end;
```

В приведенном алгоритме используется функция ReadBinary, которая читает битовую последовательность и возвращает целое значение 0, если текущий бит равен 0 и возвращает 1, если бит равен 1. Кроме того, она формирует флаг конца битовой последовательности: он принимает значение `true`, если последовательность исчерпана.

Для осуществления распаковки необходимо иметь кодовое дерево, которое приходится хранить вместе со сжатыми данными. Это приводит к некоторому незначительному увеличению объема сжатых данных. Используются самые различные форматы, в которых хранят это дерево. Здесь следует отметить, что узлы кодового дерева являются пустыми. Иногда хранят не само дерево, а исходные данные для его формирования, т. е. сведения о вероятностях появления символов (или их количествах). При этом процесс декодирования предваряется построением нового кодового дерева, которое будет таким же, как и при кодировании.

2.5. Алгоритмы сортировки

2.5.1. Основные виды сортировки

Сортировка – это процесс упорядочения некоторого множества элементов, на котором определены отношения порядка >, <, ≥, ≤, =. Когда говорят о сортировке, подразумевают упорядочение множества элементов по возрастанию или убыванию. В случае наличия элементов с одинаковыми значениями, в упорядоченной последовательности они располагаются рядом друг с другом в любом порядке. Хотя иногда, бывает полезно сохранить первоначальный порядок элементов с одинаковыми значениями.

Алгоритмы сортировки имеют большое практическое применение. Их можно встретить почти везде, где речь идет об обработке и хранении больших объемов информации. Некоторые задачи обработки данных решаются проще, если данные упорядочены.

Традиционно различают внутреннюю сортировку, в которой предполагается, что данные находятся в оперативной памяти, и важно оптимизировать число действий программы (для методов, основанных на сравнении, число сравнений, обменов элементов и пр.), и внешнюю, в которой данные хранятся на внешнем устройстве с медленным доступом (диск, лента и т. д.) и, прежде всего, надо снизить число обращений к этому устройству.

2.5.2. Алгоритмы внутренней сортировки

При дальнейшем рассмотрении внутренней сортировки определим, что множество данных, которые упорядочиваются, описывается как массив фиксированной длины:

```
A: array[1..n] of ItemType;
```

Обычно тип `ItemType` описывает запись с некоторым полем, играющим роль ключа, а сам массив представляет собой таблицу. Так как здесь рассматривается, прежде всего, сам процесс сортировки, то будем считать, что тип `ItemType` включает только ключ целого типа.

2.5.2.1. Сортировка подсчетом

Суть метода заключается в том, что на каждом шаге подсчитывается, в какую позицию результирующего массива **B** надо записать очередной элемент исходного массива **A** (рис. 40). Если некоторый элемент **A**[i] помещается в результирующий массив в позицию $k + 1$, то слева от **B**[$k + 1$] должны стоять элементы меньшие или равные **B**[$k + 1$]. Значит, число k складывается из количества элементов меньших **A**[i] и, возможно, некоторого числа элементов, равных **A**[i]. Условимся, что из равных будут учитываться только те элементы, которые в исходном массиве стоят левее **A**[i]:

```
procedure NumerSort(n: integer;
                    var A: array [1..n] of integer);
{Процедура сортировки подсчетом}
var
  i, j, k: integer;
  B: array[1..n] of integer;
```

```
begin
  for i := 1 to n do begin
    {Вычисляем положение элемента в результирующем массиве}
    k := 1;
    for j := 1 to n do
      if (A[j] < A[i]) or
         ((A[j] = A[i]) and (j < i)) then k := k+1;
    {Включаем очередной элемент в результирующий массив}
    B[k] := A[i];
  end;
  for i := 1 to n do A[i] := B[i];
end;
```

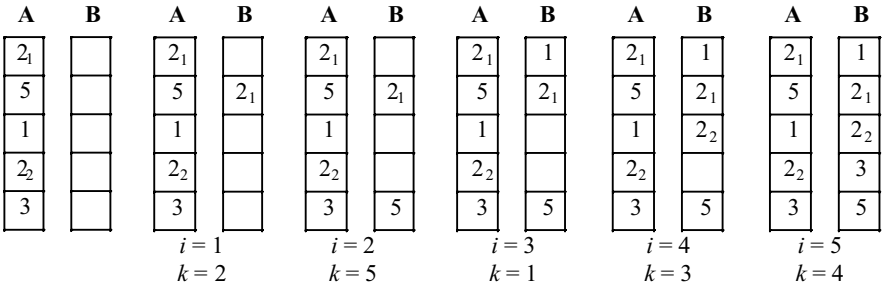


Рис. 40. Сортировка подсчетом

Легко видеть, что этот алгоритм всегда имеет временную сложность, пропорциональную $O(n^2)$ (два вложенных цикла, зависящих от n линейно и не зависящих от порядка элементов) и пространственную сложность, пропорциональную $O(n)$ (результирующий массив). Также следует отметить, что данный алгоритм сохраняет порядок элементов с одинаковыми значениями.

2.5.2.2. Сортировка простым включением

В этой сортировке массив делится на две части: отсортированную и неотсортированную. На каждом шаге берется очередной элемент из неотсортированной части и «включается» в отсортированную часть массива (рис. 41).

Пусть отсортировано начало массива **A**[1], **A**[2], ..., **A**[i–1], а остаток массива **A**[i], ..., **A**[n] содержит неотсортированную часть. На очередном шаге будем включать элемент **A**[i] в отсортированную часть, ставя его на соответствующее место. При этом придется сдвинуть часть эле-

ментов, больших **A**[*i*], (если таковые есть) на одну позицию правее, чтобы освободить место для элемента **A**[*i*]. Но при сдвиге будет потеряно само значение **A**[*i*], поскольку в эту позицию запишется первый (самый правый – с самым большим индексом) сдвигаемый элемент. Поэтому прежде чем производить сдвиг элементов необходимо сохранить значение **A**[*i*] в промежуточной переменной.

Так как массив из одного элемента можно считать отсортированным, начнем с $i = 2$.

Выглядит это в виде следующей процедуры:

```
procedure InsertSort(n: integer;
                     var A: array[1..n] of integer);
{Процедура сортировки простым включением}
var
  i, j, Tmp: integer;
begin
  for i := 2 to n do begin
    {Сохраняем текущий элемент}
    Tmp := A[i];
    {Сдвигаем элементы, большие, чем текущий}
    j := i-1;
    while (A[j] > Tmp) and (j > 1) do begin
      A[j+1] := A[j];
      j := j-1;
    end;
    {Вставляем текущий элемент}
    A[j+1] := Tmp;
  end;
end;
```

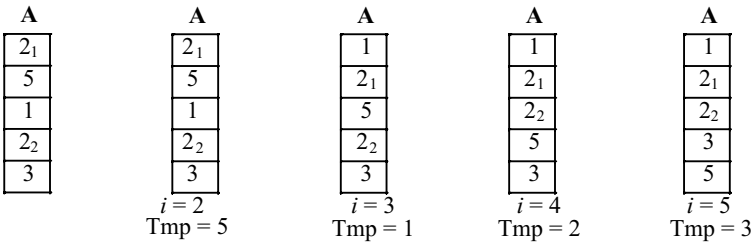


Рис. 41. Сортировка простым включением

Этот алгоритм также имеет максимальную и среднюю временную сложности, пропорциональные $O(n^2)$, но в случае исходно отсортированного

массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет временную сложность $T_{\min}(n)$, пропорциональную $O(n)$. Можно заметить, что метод использует любой частичный порядок, и чем в большей степени массив исходно упорядочен, тем быстрее он закончит работу. В отличие от предыдущего метода, этот не требует дополнительной памяти, но сохраняет порядок элементов с одинаковыми значениями.

2.5.2.3. Сортировка методом Шелла

Метод Шелла является усовершенствованием метода простого включения, который основан на том, что включение использует любой частичный порядок. Но недостатком простого включения является то, что во внутреннем цикле элемент **A**[*i*] фактически сдвигается на одну позицию. И так до тех пор, пока он не достигнет своего места в отсортированной части. (На самом деле передвигалось место, оставленное под **A**[*i*]). Метод Шелла позволяет преодолеть это ограничение следующим интересным способом (рис. 42).

Вместо включения **A**[*i*] в подмассив предшествующих ему элементов, его включают в подсписок, содержащий элементы **A**[$i - h$], **A**[$i - 2h$], **A**[$i - 3h$] и так далее, где h – положительная константа. Таким образом, формируется массив, в котором «h-серии» элементов, отстоящие друг от друга на h, сортируются отдельно.

Конечно, этого недостаточно: процесс возобновляется с новым значением h, меньшим предыдущего. И так до тех пор, пока не будет достигнуто значение $h = 1$.

В настоящее время неизвестна последовательность h_i, h_{i-1}, h_{i-2}, ..., h_1, оптимальность которой доказана. Для достаточно больших массивов рекомендуемой считается такая последовательность, что $h_{i+1} = 3h_i + 1$, а $h_1 = 1$. Начинается процесс с h_m, что $h_m \geq [n/9]$. Иногда значение h вычисляют проще: $h_{i+1} = h_i/2$, $h_1 = 1$, $h_m = n/2$. Это упрощенное вычисление h и будем использовать далее.

Теперь запишем алгоритм:

```
procedure ShellSort(n: integer;
                    var A: array[1..n] of integer);
{Процедура сортировки Шелла}
var
  h, i, j, Tmp: integer;
begin
  {Вычисляем величину h}
```

```
  h := n div 2;
  {Собственно сортировка}
  while h > 0 do begin
    for i := 1 to n-h do begin
      j := i;
      while j > 0 do begin
        {Сравниваем элементы, отстоящие друг от друга}
        {на расстояние, кратное h}
        if A[j] > A[j+h] then begin
          {Меняем элементы}
          Tmp := A[j+h];
          A[j+h] := A[j];
          A[j] := Tmp;
          j := j - h;
        end else begin
          {Досрочно завершаем цикл с параметром j}
          j := 0;
        end;
      end;
    end;
    {Уменьшаем размер серии}
    h := h div 2;
  end;
end;
```

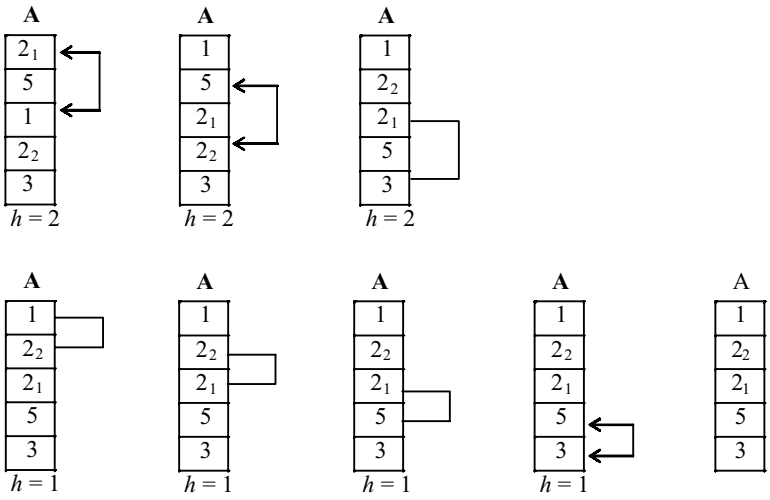


Рис. 42. Сортировка Шелла

Как показывают теоретические выкладки, которые здесь приводить не будем, сортировке методом Шелла требуется в среднем $1{,}66n^{1{,}25}$ перемещений. Порядок элементов влияет на количество итераций внутреннего цикла `while`. Дополнительной памяти данный алгоритм не требует, но и не гарантирует сохранение порядка элементов с одинаковыми значениями.

2.5.2.4. Сортировка простым извлечением.

В этом методе массив также делится на уже отсортированную часть **A**[*i*+1], **A**[*i*+1], ..., **A**[*n*] и еще не отсортированную **A**[1], **A**[2], ..., **A**[*i*]. Но здесь из неотсортированной части на каждом шаге извлекается максимальный элемент, просматривая ее заново на каждом шаге. Этот элемент будет минимальным элементом отсортированной части, так как все большие его элементы были извлечены на предыдущих шагах, поэтому ставим извлеченный элемент в начало отсортированной части, точнее меняем его с **A**[*i*] местами (рис. 43).

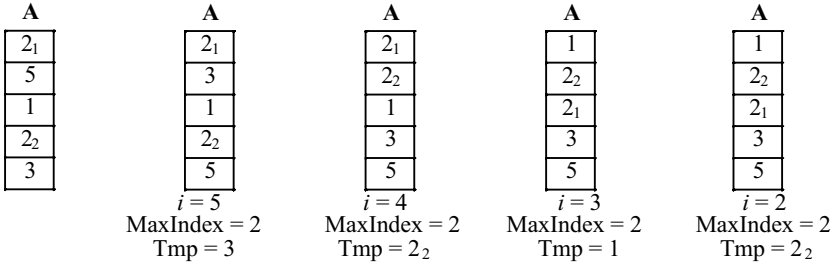


Рис. 43. Сортировка простым извлечением

Теперь запишем алгоритм.

```
procedure ExtractSort(n: integer;
                      var A: array[1..n] of integer);
{Процедура сортировки простым извлечением}
var
  i, j, MaxIndex, Tmp: integer;
begin
  for i := n downto 2 do begin
    {Ищем максимальный элемент в неотсортированной части}
    MaxIndex := 1;
    for j := 2 to i do
      if A[j] > A[MaxIndex] then MaxIndex := j;
    {Меняем найденный элемент с первым из отсортированных}
```

```
    Tmp := A[i]; A[i] := A[MaxIndex];
    A[MaxIndex] := Tmp;
  end;
end;
```

Простое извлечение во всех случаях имеет временную сложность, пропорциональную $O(n^2)$ (два вложенных цикла, зависящих от n линейно и не зависящих от порядка элементов). Также следует отметить, что данный алгоритм не требует дополнительной памяти и не гарантирует сохранение порядка элементов с одинаковыми значениями.

2.5.2.5. Древесная сортировка

При использовании этой сортировки в массиве постоянно поддерживается такой порядок, при котором максимальный элемент всегда будет оказываться в **A**[1]. Сортировка называется древесной, потому что в этом методе используется структура данных, называемая двоичным деревом. При чем используется представление дерева в виде массива (см. п. 1.3.4.4) и при сортировке используется тот же способ расположения вершин дерева в массиве.

Пусть **A**[1]...**A**[n] – массив, подлежащий сортировке. Вершинами дерева будут числа от 1 до n; о числе **A**[i] будем говорить как о числе, стоящем в вершине i. В процессе сортировки количество вершин дерева будет сокращаться. Число вершин текущего дерева будем хранить в переменной k. Таким образом, в процессе работы алгоритма массив **A**[1]...**A**[n] делится на две части: в **A**[1]...**A**[k] хранятся числа на дереве, а в **A**[k+1]...**A**[n] хранится уже отсортированная в порядке возрастания часть массива – элементы, уже занявшие свое законное место (рис. 44).

На каждом шаге алгоритм будет изымать максимальный элемент дерева, и помещать его в отсортированную часть, на освободившееся в результате сокращения дерева место.

Договоримся о терминологии. Вершинами дерева считаются числа от 1 до текущего значения переменной k. У каждой вершины s могут быть потомки $2s$ и $2s + 1$. Если оба этих числа больше k, то потомков нет; такая вершина называется листом. Если $2s = k$, то вершина s имеет ровно одного потомка ($2s$).

Для каждого s из 1...k рассмотрим «поддерево» с корнем в s: оно содержит вершину s и всех ее потомков. Вершина s называется р е г у л я р н о й , если стоящее в ней число – максимальный элемент s-поддерева; s-поддерево называется регулярным, если все его вершины регу-

лярны. В частности, любой лист образует регулярное одноэлементное поддерево.

Теперь запишем алгоритм сортировки:

```
procedure TreeSort (n: integer;
                    var A: array[1..n] of integer);
{Процедура древесной (пирамидальной) сортировки}
var
  u, k: integer;
procedure Exchange(i, j: integer);
{Процедура обмена двух элементов}
var
  Tmp: integer;
begin
  Tmp  := A[i];
  A[i] := A[j];
  A[j] := Tmp;
end;  {Exchange}
procedure Restore(s: integer);
{Процедура восстановления регулярности поддерева с корнем s}
var
  t: integer;
begin
  t:=s; {начинаем с корня поддерева}
  while ((2*t+1 <= k) and (A[2*t+1] > A[t])) or
        ((2*t <= k) and (A[2*t] > A[t])) do begin
  {Пока не просмотрено все поддерево и вершина t нерегулярна}
    if (2*t+1 <= k) and (A[2*t+1] >= A[2*t]) then begin
    {Меняем корень поддерева с его правым потомком}
      Exchange(t, 2*t+1);
      t := 2*t+1;  {переход к правому потомку}
    end else begin
    {Меняем корень поддерева с его левым потомком}
      Exchange(t, 2*t);
      t := 2*t;    {переход к правому потомку}
    end;
  end;
end;  {Restore}
begin
  k:= n;
  u:= n;
  while u <> 0 do begin
```

```
    Restore(u);
    u := u - 1;
  end;
  while k <> 1 do begin
    Exchange(1, k);
    k := k - 1;
    Restore(1);
  end;
end;  {TreeSort}
```

В качестве вспомогательных процедур используются процедуры обмена двух элементов `Exchange` и процедура восстановления регулярности *s*-поддерева в корне – `Restore`. Первая процедура введена исключительно для лучшей наглядности. Вторая требуется для восстановления регулярности поддерева, которая может нарушиться после обменов элементов.

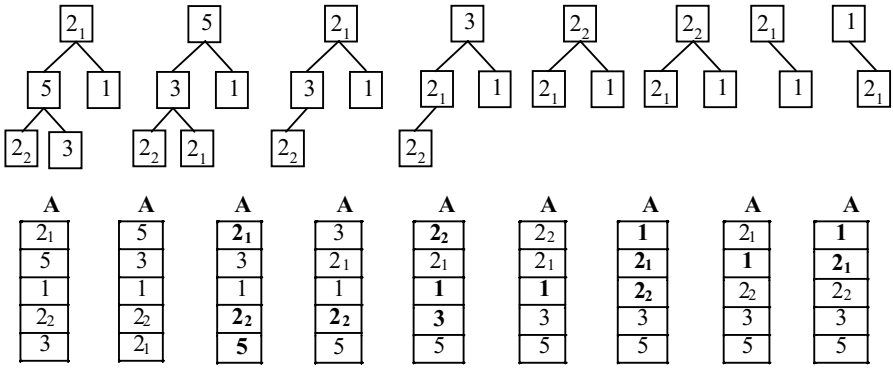


Рис. 44. Древесная сортировка

Рассмотрим восстановление регулярности подробнее. Пусть в *s*-поддереве все вершины, кроме разве что вершины *t*, регулярны. Рассмотрим потомков вершины *t*. Они регулярны, и потому содержат наибольшие числа в своих поддеревьях. Таким образом, на роль наибольшего числа в *t*-поддереве могут претендовать число в самой вершине *t* и числа, содержащиеся в ее потомках (в первом случае вершина *t* регулярна, и все в порядке).

После обмена вершина *t* становится регулярной (в нее попадает максимальное число *t*-поддерева). Не принявший участия в обмене потомок остается регулярным, а принявший участие может и не быть регу-

лярным. В остальных вершинах *s*-поддерева не изменились ни числа, ни поддеревья их потомков (разве что два элемента поддерева поменялись местами), так что регулярность не нарушилась.

Эта же процедура может использоваться для того, чтобы сделать 1-поддерево регулярным на начальной стадии сортировки (см. первый цикл в теле основной процедуры).

Преимущество этого алгоритма перед другими в том, что он, имея максимальную временную сложность $T_{\max}(n)$, пропорциональную $O(n\log n)$ (внутри внешнего цикла зависящего от n линейно вызывается процедура `Restore`, требующая порядка $\log n$ действий), не требует дополнительной памяти порядка $O(n)$.

2.5.2.6. Сортировка методом пузырька

Сортировка методом пузырька – один из наиболее широко известных алгоритмов сортировки. В этом методе массив также делится на две части: отсортированную и неотсортированную. На каждом шаге метода осуществляется просмотр от меньших индексов к большим по неотсортированной части, каждый раз сравнивая два соседних элемента. Если они не упорядочены между собой (меньший следует за большим), то меняем их местами. Тем самым за один проход путем последовательных обменов наибольший элемент неотсортированной части сдвинется к ее концу (рис. 45).

Алгоритм называют пузырьковой сортировкой, потому что на каждом шаге наибольший элемент неотсортированной части подобно пузырьку газа в воде всплывает к концу массива.

Заметим, что в том случае, когда за очередной проход не было сделано ни одного обмена, массив уже отсортирован, и следующие проходы можно пропустить. Для отслеживания такой ситуации введем логическую переменную Flag – признак совершения обмена на очередном проходе.

Теперь запишем алгоритм:

```
procedure BubleSort(n: integer;
                    var A: array[1..n] of integer);
{Процедура сортировки методом пузырька}
var
  i, j, Tmp: integer;
  Flag: boolean;
begin
```

```
  for i := n-1 downto 1 do begin
    Flag := false;
    for j := 1 to i do
      if A[j] > A[j+1] then begin
        Tmp := A[j]; A[j] := A[j+1]; A[j+1] := Tmp;
        Flag := true;
      end;
    if not Flag then Break;
  end;
end;
```

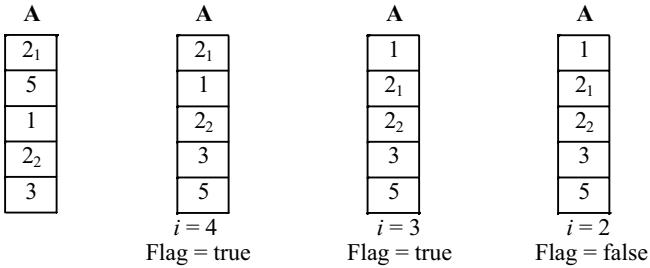


Рис. 45. Сортировка методом пузырька

Этот алгоритм имеет среднюю и максимальную временные сложности, пропорциональную $O(n^2)$ (два вложенных цикла, зависящих от n линейно) и не требует дополнительной памяти. Введение переменной Flag и прерывание работы в случае отсортированного массива позволяет свести минимальную временную сложность к $O(n)$. Также следует отметить, что данный алгоритм не требует дополнительной памяти и сохраняет порядок элементов с одинаковыми значениями.

2.5.2.7. Быстрая сортировка (Хоара)

Эту сортировку называют быстрой, потому что на практике она оказывается самым быстрым алгоритмом сортировки из тех, что оперируют сравнениями.

Этот алгоритм является ярким примером реализации принципа «разделяй и властвуй». Как показывают теоретические выкладки, наиболее эффективным в общем случае оказывается разделение задачи на две равные по сложности части, что здесь и делается.

На каждом шаге алгоритма сначала выбирается «средний» элемент, затем переставляются элементы массива так, что массив разделился на две части. Первая часть содержит элементы меньше «среднего» и, воз-

можно, равные ему. Вторая часть содержит элементы больше «среднего» и, возможно, равные ему. После такого деления массива остается только отсортировать его части по отдельности, с которыми поступаем аналогично (делим на две части). И так до тех пор, пока эти части не окажутся состоящими из одного элемента, а массив из одного элемента всегда отсортирован (рис. 46). В случае, когда массив содержит только одинаковые элементы, выбор «среднего» элемента не производится и сортировка не осуществляется.

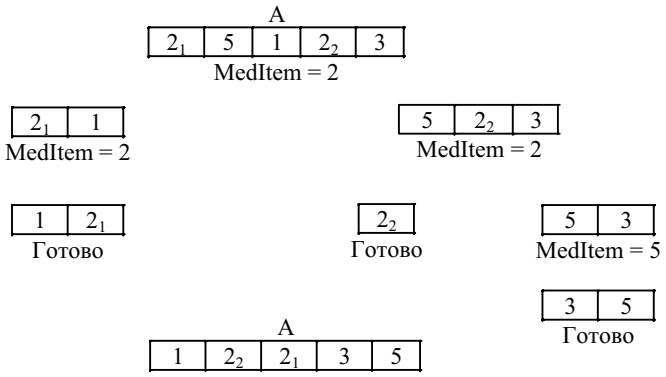


Рис. 46. Быстрая сортировка Хоара

Разделение массива на две части производится следующим образом. Устанавливаем один курсор на левую границу массива, а второй – на правую границу. Затем осуществляем перемещение курсоров навстречу друг другу до тех пор, пока они не пересекутся. При перемещении курсоров сравниваем значения текущих элементов со «средним». Находим левый текущий элемент, больший «среднего», и правый текущий элемент, меньший «среднего» (т. е. элементы, которые находятся «не на своем месте»). Осуществляем обмен этих элементов.

Выбор «среднего» – задача непростая, так как требуется, не производя сортировку, найти элемент со значением максимально близким к среднему. Здесь, конечно, можно просто выбрать произвольный элемент (обычно выбирают элемент, стоящий в середине сортируемого подмассива), но пойдем чуть дальше: из трех элементов (самого левого, самого правого и стоящего посередине) выберем средний:

```
procedure HoarSort(n: integer;
                   var A: array[1..n] of integer);
{Процедура сортировки Хоара}
```

```
function FindMedium(L, R: integer): integer;
{Нахождение индекса "среднего" элемента}
var
  MedIndex,           {индекс "среднего" элемента}
  Left, Right, Median: integer;
begin
  Left := A[L];  Right := A[R];  Median := A[(L+R) div 2];
  {Берем два крайних элемента и один из середины массива}
  if (Left = Median) and (Median = Right) then begin
    {Если все три элемента одинаковы, то ищем неравный им}
    i := L;
    while (A[i] = Median) and (i < R) do i := i + 1;
    {Если найден неравный элемент, то берем его третьим}
    if A[i] <> Median then Median := A[i];
  end;
  if (Left = Median) and (Median = Right) then begin
    {Все элементы массива одинаковы и "средний" не найден}
    FindMedium := 0;
  end else begin
    {Выбираем "средний" из трех разных элементов}
    if Left <= Median then
      if Median <= Right then
        MedIndex := (L+R) div 2
      else
        if Left <= Right then MedIndex := R
        else MedIndex := L
    else
      if Left >= Right then
        MedIndex := (L+R) div 2
      else
        if Left >= Right then
          MedIndex := R
        else
          MedIndex := L;
    FindMedium := MedIndex;
  end;
end;  {FindMedium}
procedure QuickSort(L, R: integer);
var
  MedItem,            {значение "среднего" элемента}
  MedIndex,           {индекс "среднего" элемента}
```

```
  Tmp, i, j: integer;  {вспомогательные переменные}
begin
  MedIndex := FindMedium(L, R);
  if MedIndex <> 0 then begin
    {Сортируем, если найден "средний" элемент}
    MedItem := A[MedIndex];
    {Разбиваем массив на две части}
    i := L; j := R;
    while i <= j do begin
      {Ищем первый слева элемент, больший, чем MedItem}
      while A[i] < MedItem do i := i + 1;
      {Ищем первый справа элемент, меньший, чем MedItem}
      while A[j] > MedItem do j := j - 1;
      if i <= j then begin  {Меняем местами найденные элементы}
        Tmp  := A[i];
        A[i] := A[j];
        A[j] := Tmp;
        i := i + 1;
        j := j - 1;
      end;
    end;
    {Сортируем две части массива по отдельности}
    if L < j then QuickSort(L, j);
    if i < R then QuickSort(i, R);
  end;
end;    {QuickSort}
begin           {HoarSort}
  QuickSort(1, n);
end;             {HoarSort}
```

Заметим, что предложенный способ нахождения «среднего» элемента подмассива в худшем случае приведет к тому, что после деления, например, правая часть поделенного массива будет содержать один элемент, а левая – все остальные. В этом случае получается порядка n рекурсивных вызовов. Это значит, что необходимо будет завести дополнительную память размером, пропорциональным n, и пространственная сложность $V_{\max}(n)$ будет пропорциональна $O(n)$. В среднем и лучшем случае, можно говорить о пространственной сложности, пропорциональной $O(\log n)$.

В худшем случае этот алгоритм дает временную сложность $T_{\max}(n)$, пропорциональную $O(n^2)$ (для случая, когда все выборки «среднего»

элемента оказались неудачны), но как показывают теоретические исследования, вероятность такого случая очень мала. В среднем же и в лучшем случае получим временную сложность $T(n)$, пропорциональную $O(n\log n)$.

2.5.2.8. Сортировка слиянием

Этот метод сортирует массив последовательным слиянием пар уже отсортированных подмассивов.

Пусть k – положительное целое число. Разобьем массив **A**[1]...**A**[n] на участки длины k. (Первый – **A**[1]...**A**[k], затем **A**[k+1]...**A**[2k] и т. д.) Последний участок будет неполным, если n не делится нацело на k. Назовем массив k-упорядоченным, если каждый из этих участков длины k упорядочен.

Ясно, что любой массив 1-упорядочен, так как его участки длиной 1 можно считать упорядоченными. Если массив k-упорядочен и $n \le k$, то он упорядочен.

Рассмотрим процедуру преобразования k-упорядоченного массива в 2k-упорядоченный. Сгруппируем все участки длины k в пары участков. Теперь пару упорядоченных участков сольем в один упорядоченный участок. Проделав это со всеми парами, получим 2k-упорядоченный массив (рис. 47).

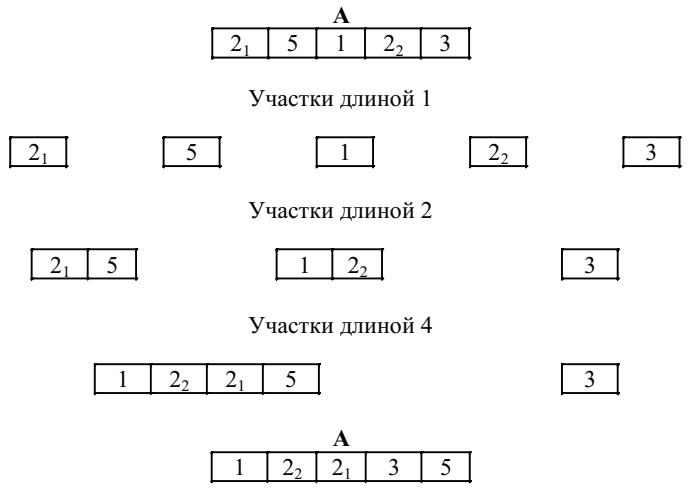


Рис. 47. Сортировка слиянием

Слияние требует вспомогательного массива **B** для записи результатов слияния. При слиянии сравниваем наименьшие элементы участков рассматриваемой пары, и меньший из них заносим в массив **B**. Повторяем описанные действия до тех пор, пока не исчерпается один из участков. После чего заносим в массив **B** все оставшиеся элементы другого участка. Затем переходим к следующей паре участков:

```
procedure MergeSort(n: integer;
                    var A: array[1..n] of integer);
{Процедура сортировки слиянием}
var
  i, j, k, t, s, Start1, Fin1, Fin2: integer;
  B: array[1..n] of integer;
begin
  k := 1;   {Начальное значение длины участков}
  while k < n do begin  {пока участок не весь массив}
    t := 0;             {начало 1-й пары участков}
    while t+k < n do begin  {пока не все участки просмотрели}
      {Определяем границы рассматриваемых участков}
      Start1 := t+1;  Fin1 := t+k;  {Начало и конец 1-го участка}
      if t+2*k > n then Fin2 := n
                   else Fin2 := t+2*k; {Конец 2-го участка}
      i := Start1;    {Начальное значение индекса в 1-м участке}
      j := Fin1 + 1;  {Начальное значение индекса в 2-м участке}
      s := 1;         {Начальное значение индекса в массиве B}
      {Заполняем B элементами из двух участков}
      while (i <= Fin1) and (j <= Fin2) do begin
        {Сравниваем попарно элементы из двух участков}
        if A[i] < A[j] then begin
        {Вставляем элемент из 1-го участка}
          B[s] := A[i];
          i := i + 1;
        end else begin
        {Вставляем элемент из 2-го участка}
          B[s] := A[j];
          j := j + 1;
        end;
        s := s + 1;
      end;
      {Добавляем в массив B оставшиеся элементы из 1-го участка}
```

```
      while (i <= Fin1) do begin
        B[s] := A[i];
        i := i + 1; s := s + 1;
      end;
      {Добавляем в массив B оставшиеся элементы из 2-го участка}
      while (j <= Fin2) do begin
        B[s] := A[j];
        j := j + 1; s := s + 1;
      end;
      t := Fin2;    {Переходим к следующей паре участков}
    end;
    k := k * 2;  {Удваиваем значение длины участков}
    {Сохраняем полученный промежуточный результат}
    for s := 1 to t do A[s] := B[s];
  end;
end;
```

Сразу же бросается в глаза недостаток алгоритма – он требует дополнительную память размером порядка n (для хранения вспомогательного массива). Кроме того, он не гарантирует сохранение порядка элементов с одинаковыми значениями. Но его временная сложность всегда пропорциональна $O(n\log n)$ (так как преобразование k-упорядоченного массива в $2k$-упорядоченный требует порядка n действий и внешний цикл по k совершает порядка $\log n$ итераций).

2.5.2.9. Сортировка распределением

Сортировка распределением интересна тем, что она сортирует массив, не сравнивая элементы друг с другом.

Рассмотрим сначала вырожденный случай сортировки распределением, а затем более общий.

При вырожденном распределении предполагается, что каждый элемент массива может принимать m (например, от 1 до m) фиксированных значений. Заведем массив **Amount** размерностью m, первоначально обнулив его. Затем для каждого i подсчитаем количество элементов массива **A**, равных i, и занесем это число в **Amount**[i]. После чего, в первые **Amount**[1] элементов массива **A** запишем 1, в следующие **Amount**[2] элементов массива **A** запишем 2 и т. д. до тех пор, пока не дойдем до конца массива **A** (заметим, что в то же время мы окажемся в конце массива **Amount**).

Теперь запишем алгоритм:

```
procedure DispersSort(n, m: integer;
                      var A: array[1..n] of integer);
{Процедура сортировки вырожденным распределением}
var
  i, j, k: integer;
  Amount: array[1..m] of integer;
begin
  {Обнуляем массив Amount}
  for i := 0 to m do Amount[i] := 0;
  {Заполняем массив Amount}
  for i := 1 to n do Amount[A[i]] := Amount[A[i]] + 1;
  {Заполняем массив A}
  k := 1;
  for i := 0 to M do
    for j := 1 to Amount[i] do begin
      A[k] := i;
      k := k + 1;
    end;
end;
```

Временную сложность метода можно оценить как $O(m+n)$ (m появляется в сумме, так как изначально надо обнулить массив **Amount**, а это требует m действий). Пространственная сложность в этом случае пропорциональна $O(m)$, поскольку требуется дополнительная память размером порядка m.

Недостатком этого метода является то, что требуется дополнительная память размером порядка m, а это может оказаться недопустимым из-за большого значения m. Но, если $m>>n$, то имеется способ уменьшить объем требуемой дополнительной памяти, который сейчас и рассмотрим, как общий случай сортировки распределением.

Пусть выделяется дополнительная память размером $b+n$, а элементы массива могут принимать значения от 0 до s, причем $s>>b$.

Каждый элемент этого массива можно представить в b-ичной системе счисления и разбить на k цифр этой системы счисления.

Заведем списки L_1, L_2, …, L_b общей суммарной длиной порядка n (это можно сделать, ограничившись дополнительной памятью $O(b+n)$) (рис. 48).

Тогда алгоритм сортировки распределением можно представить следующим образом:

```
for i := k downto 1 do begin
  for j := 1 to n do begin
    if p = i-й цифре A[j] в b-й системе счисления then
      занести A[j] в L[p] список;
  end;
  Очистить массив A;
  for j := 1 to b do
  Дописать элементы L[j] в массив A;
end;
```

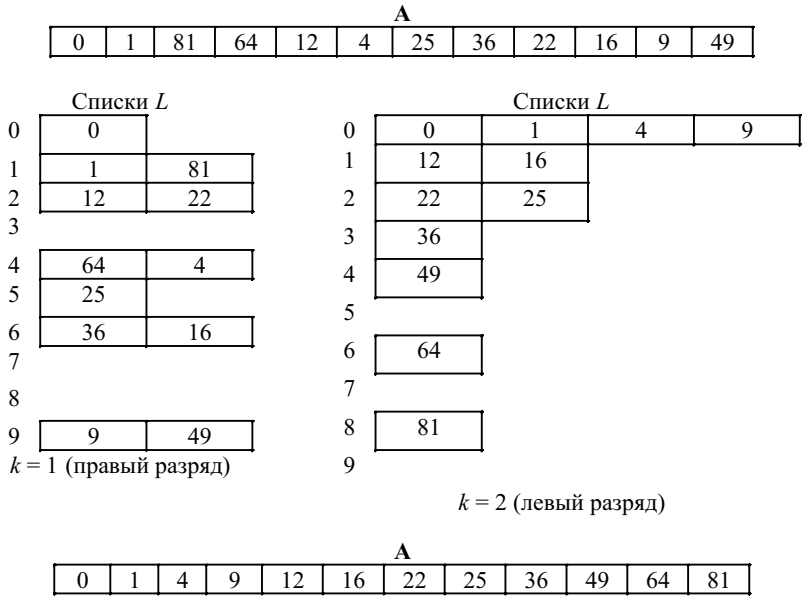


Рис. 48. Сортировка распределением

Итак, как видно из приведенной выше программы, на каждом шаге метода производится сортировка элементов массива по значению *i*-ого разряда. При этом производится промежуточное распределение элементов массива по спискам в зависимости от значения соответствующего разряда этих элементов. Во время распределения очень важно сохранить при записи в списки порядок следования элементов, чтобы не нарушить порядок, достигнутый на предыдущих шагах.

Индукцией по *i* легко доказать, что после *i* шагов любые два числа, отличающиеся только в *i* последних разрядах, идут в правильном порядке.

Достигнув $i = 1$, получаем полностью отсортированный массив.

Как нетрудно заметить, если положить $s = b$, то отпадает необходимость заводить списки и производить запись в них: в j-ый список будут попадать только числа, равные j. В этом случае достаточно хранить лишь размеры списков, т. е. подсчитать количество элементов, равных j, для всех j от 1 до s. А потом просто заново заполнить массив **A** в соответствии с этими количествами, т. е. получаем вырожденную сортировку.

Рассмотрим на примере задачу сортировки 12 целых чисел из интервала от 0 до 99, т. е. $n = 12$, $b = 10$ (десятичная система счисления), $s = 99$, $k = 2$ (два разряда). При этом будем считать, что числа, содержащие только один разряд, дополняются слева нулем, т. е. число «0» будет «00», число «1» будет «01» и т. д.

Интересно, что временная сложность этого алгоритма пропорциональна $O(k \cdot n)$, а если учесть, что k фактически является константой, то получаем гарантированную (минимальную, среднюю и максимальную) линейную сложность. Но недостатком этого метода является необходимость выделять дополнительную память размером порядка $b + n$. Если бы не это ограничение, можно было бы считать этот метод самым эффективным при больших значениях n.

2.5.2.10. Сравнение алгоритмов внутренней сортировки

Выше было рассмотрено достаточно большое количество алгоритмов внутренней сортировки. Возникает вопрос: зачем тогда нужно такое разнообразие алгоритмов сортировок, если есть возможность раз и навсегда определить алгоритм с наилучшим показателем эффективности и оставить «право на жизнь» исключительно за ним? Ответ прост: в реальных задачах имеются ограничения, определяемые как логикой задачи, так и свойствами конкретной вычислительной среды, которые могут существенно влиять на эффективность данной конкретной реализации алгоритма. Поэтому выбор того или иного алгоритма всегда остается за разработчиком программного обеспечения.

Теоретические временные и пространственные сложности рассмотренных методов сортировки показаны в табл. 4.

Эта таблица позволяет сделать ряд выводов.

1. На небольших наборах данных целесообразнее использовать сортировку включением, так как из всех методов, имеющих очень простую программную реализацию, этот на практике оказывается самым быст-

Таблица 4

Метод сортировки	Характеристики			
	T_{max}	T_{mid}	T_{min}	V_{max}
Подсчет	$O(n^2)$			$O(n)$
Включение	$O(n^2)$		$O(n)$	$O(1)$
Шелла	$O(n2)$	$O(n^{1,25})$	$O(n)$	$O(1)$
Извлечение	$O(n^2)$			$O(1)$
Древесная	$O(n*\log n)$			$O(1)$
Пузырьковая	$O(n^2)$		$O(n)$	$O(1)$
Быстрая	$O(n^2)$	$O(n*\log n)$		$O(\log n)$
Слияние	$O(n*\log n)$			$O(n)$
Распределение	$O(n)$			$O(n)$

рым и при размерностях меньше ~3000 дает вполне приемлемую для большинства случаев скорость работы. Еще одно преимущество этого метода заключается в том, что он использует полную или частичную упорядоченность входных данных и на упорядоченных данных работает быстрее, а на практике данные, как правило, уже имеют хотя бы частичный порядок.

2. Алгоритм пузырьковой сортировки, причем в той его модификации, которая не использует частичный порядок данных исходного массива, хотя и часто используется, но имеет плохие показатели даже среди простых методов с квадратичной сложностью.

3. Сортировка Шелла оказывается лишь красивым теоретическим методом, потому что на практике использовать его нецелесообразно: он сложен в реализации, но не дает такой скорости, какую дают сравнимые с ним по сложности программной реализации методы.

4. При сортировке больших массивов исходных данных лучше использовать быструю сортировку.

5. Если же добавляется требование гарантировать приемлемое время работы метода (быстрая сортировка в худшем случае имеет сложность, пропорциональную $O(n^2)$, хотя вероятность такого случая очень мала), то надо применять либо древесную сортировку, либо сортировку слиянием. Как видно из таблиц, сортировка слиянием работает быстрее, но следует помнить, что она требует дополнительную память размером порядка n.

6. В тех же случаях, когда есть возможность использовать дополнительную память размером порядка n, имеет смысл воспользоваться сортировкой распределением.

2.5.3. Алгоритмы внешней сортировки

Как уже говорилось выше, внешняя сортировка – это упорядочивание данных, которые хранятся на внешнем устройстве с медленным доступом (диск, лента и т. д.), и прежде всего надо уменьшить число обращений к этому устройству, т. е. число проходов через файл.

Обычно данные, хранящиеся на внешних устройствах, имеют большой объем, что не позволяет их целиком переместить в оперативную память, отсортировать с использованием одного из алгоритмов внутренней сортировки, а затем вернуть их на внешнее устройство. В этом случае осуществлялось бы минимальное количество проходов через файл: однократное чтение и однократная запись данных. Однако на практике приходится осуществлять чтение, обработку и запись данных в файл по блокам, размер которых зависит от операционной системы и имеющегося объема оперативной памяти, что приводит к увеличению числа проходов через файл и заметному снижению скорости сортировки.

Применение большинства алгоритмов внутренней сортировки для сортировки файлов требует порядка $O(n)$ проходов. Однако, если несколько модифицировать алгоритм сортировки слиянием (см. п. 2.5.2.8), то можно произвести сортировку, осуществляя порядка $O(\log n)$ проходов.

Основное отличие сортировки слиянием для файлов, заключается в следующем. Вся сортируемая последовательность данных разбивается на два файла f_1 и f_2. Желательно, чтобы количество записей в этих файлах было поровну. Как и в алгоритме внутренней сортировки, считаем, что любой файл состоит из участков длиной 1. Затем можно объединить участки длины 1 и распределить их по файлам g_1 и g_2 в виде участков длины 2. После этого делаем f_1 и f_2 пустыми и объединяем g_1 и g_2 в f_1 и f_2, которые затем можно организовать в виде участков длины 4 и т. д.

После выполнения i подобного рода проходов получатся два файла, состоящие из участков длины 2^i. Если $2^i \geq n$, тогда один из этих двух файлов будет пустым, а другой будет содержать единственный участок длиной n, т. е. будет отсортирован. Так как $2^i \geq n$ при $i \geq \log n$, то не-

трудно заметить, что в этом случае будет достаточно порядка $O(\log n)$ проходов по данным.

Пример внешней сортировки слиянием приведен на рис. 49.

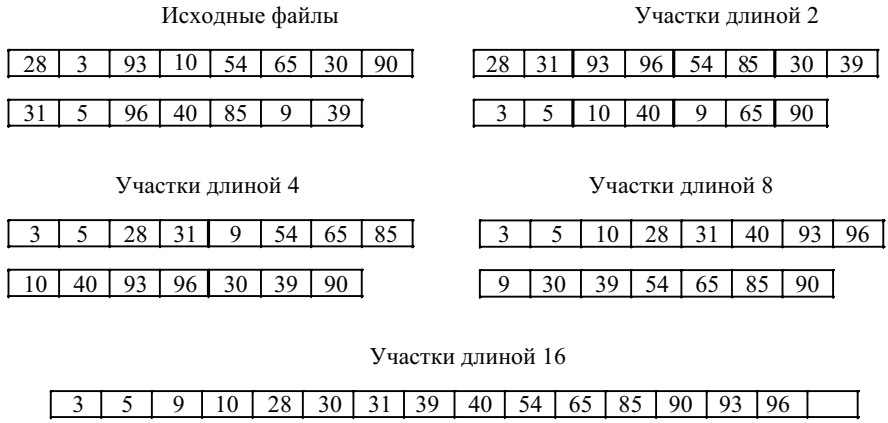


Рис. 49. Внешняя сортировка слиянием

При такой сортировке не требуется, чтобы отдельный участок полностью находился в оперативной памяти (при большой длине он может не поместиться в буфер). Участок считывается и записывается последовательно запись за записью. Именно такой подход заставляет использовать два входных файла. В противном случае можно было бы читать по два участка из одного файла одновременно.

2.6. Алгоритмы на графах

2.6.1. Алгоритм определения циклов

Наличие циклов в графе можно определить с помощью эффективного и простого алгоритма. Алгоритм может быть реализован как для матричного, так и для спискового способа представления графа. В случае неориентированного графа его ребра считаются двунаправленными.

Принцип выделения циклов следующий. Если вершина имеет только входные или только выходные дуги, то она явно не входит ни в один цикл. Можно удалить все такие вершины из графа вместе со связанными с ними дугами. В результате появятся новые вершины, имеющие только входные или только выходные дуги. Они также удаляются. Итерации повторяются до тех пор, пока граф не перестанет изменяться.

Отсутствие изменений свидетельствует об отсутствии циклов, если все вершины были удалены. Все оставшиеся вершины обязательно принадлежат циклам (рис. 60).

Сформулируем алгоритм, используя матрицу смежности (см. 1.3.3.2):

```
var
  M: TadjacencyMatrix;
repeat
  for i := 1 to n begin
    if строка M(i ,*) = 0 then обнулить столбец M(*, i);
    if столбец M(*, i) = 0 then обнулить строку M(i ,*);
  end;
until M не изменилась;
if M нулевая then граф ациклический
             else граф содержит циклы;
```

Достоинством данного алгоритма является то, что происходит одновременное определение цикличности или ацикличности графа и исключение дуг, не входящих в циклы. После завершения алгоритма остается матрица смежности, соответствующая подграфу, содержащему все циклы исходного графа.

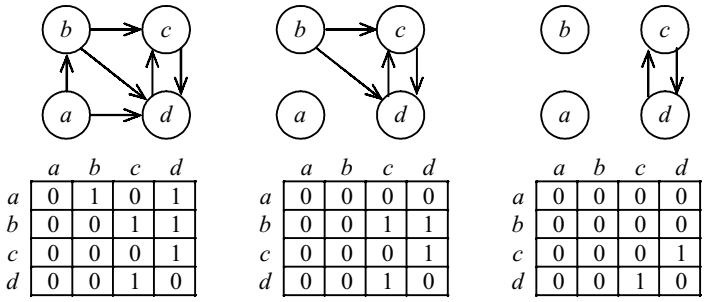


Рис. 50. Определение циклов

В худшем случае этот алгоритм дает временную сложность $T_{\max}(n)$, пропорциональную $O(n^3)$.

2.6.2. Алгоритмы обхода графа

При решении многих задач, касающихся графов, необходимы эффективные методы систематического обхода вершин и ребер графов. К таким методам относятся:

– поиск в глубину;

– поиск в ширину.

Эти методы чаще всего рассматриваются на ориентированных графах, но они применимы и для неориентированных, ребра которых считаются двунаправленными.

2.6.2.1. Поиск в глубину

Поиск в глубину является обобщением метода обхода дерева в прямом порядке (см. 1.3.4.2).

Предположим, что есть ориентированный граф G, в котором первоначально все вершины помечены как непосещенные. Поиск в глубину начинается с выбора начальной вершины v графа G, и эта вершина помечается как посещенная. Затем для каждой вершины, смежной с вершиной v и которая не посещалась ранее, рекурсивно применяется поиск в глубину. Когда все вершины, которые можно достичь из вершины v, будут «удостоены» посещения, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа G.

Этот метод обхода вершин орграфа называется поиском в глубину, поскольку поиск непосещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно. Например, пусть x – последняя посещенная вершина. Для продолжения процесса выбирается какая-либо нерассмотренная дуга $x \rightarrow y$, выходящая из вершины x. Если вершина y уже посещалась, то ищется другая вершина, смежная с вершиной x. Если вершина y ранее не посещалась, то она помечается как посещенная и поиск начинается заново от вершины y. Пройдя все пути, которые начинаются в вершине y, возвращаемся в вершину x, т. е. в ту вершину, из которой впервые была достигнута вершина y. Затем продолжается выбор нерассмотренных дуг, исходящих из вершины x, и так до тех пор, пока не будут исчерпаны все эти дуги (рис. 51).

Для представления вершин, смежных с вершиной v, можно использовать список смежных (см. 1.3.3.2), а для определения вершин, которые ранее посещались, – массив `Visited`:

```
Graph: TAdjacencyList;
Visited: array[1..n] of boolean;
```

Чтобы применить эту процедуру к графу, состоящему из n вершин, надо сначала присвоить всем элементам массива `Visited` значение `false`, затем начать поиск в глубину для каждой вершины, помеченной как `false`.

```
Procedure DepthSearch(v: integer);
begin
  Visited[v] := true;
  for каждой вершины y, смежной с v do
    if not Visited[y] then
      DepthSearch(y);
end;
begin
  while есть непомеченные вершины do begin
    v := любая непомеченная вершина;
    DepthSearch(v);
  end;
end.
```

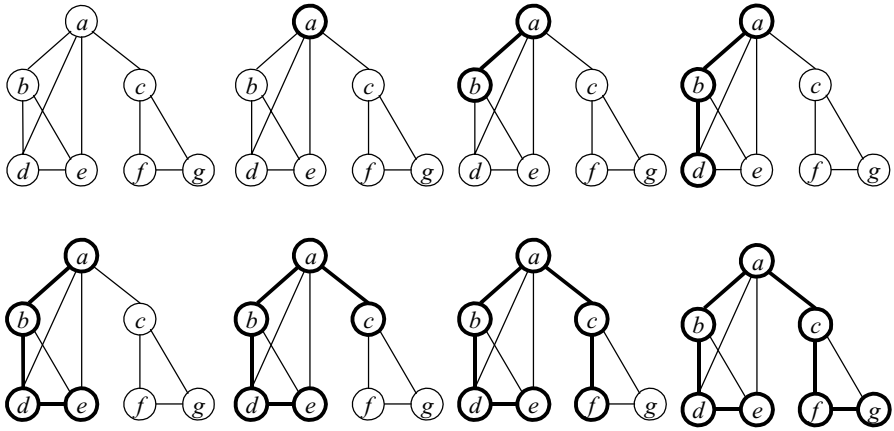


Рис. 51. Поиск в глубину

Поиск в глубину для полного обхода графа с n вершинами и m дугами требует общего времени порядка $O(\max(n, m))$. Поскольку обычно $m \geq n$, то получается $O(m)$.

2.6.2.2. Поиск в ширину (волновой алгоритм)

Этот алгоритм поиска в графе также называют волновым алгоритмом из-за того, что обход графа идет по принципу распространения волны. Волна растекается равномерно во все стороны с одинаковой скоростью. На i-м шаге будут помечены все вершины, достижимые за i ходов, если ходом считать переход из одной вершины в другую.

Метод поиска в ширину получается из программы поиска в глубину (см. 2.6.2.1), если заменить стек возврата на очередь. Эта простая замена модифицирует порядок обхода вершин так, что обход идет равномерно во все стороны, а не вглубь как при поиске в глубину (рис. 52).

Здесь используются те же структуры `Graph` и `Visited`, что были описаны в алгоритме поиска в глубину.

```
Procedure WidthSearch(v: integer);
var
  Delayed: array[1..n] of integer; {Очередь}
  Count,                           {Хвост очереди}
  Head: integer;                   {Голова очереди}
  Current, j: integer;
begin
  Count := 1; Head := 0; Delayed[Count] := v;
  Visited[v] := true;
  repeat
    Head := Head +1;  Current := Delayed[Head];
    for каждой вершины y, смежной с Current do
      if not Visited[y] then begin
        Count := Count + 1;
        Delayed[Count] := Graph[y];
        Visited[y] := true;
      end;
  until Count = Head;
end;
begin
  while есть непомеченные вершины do begin
    v := любая непомеченная вершина;
    WidthSearch(v);
  end;
end.
```

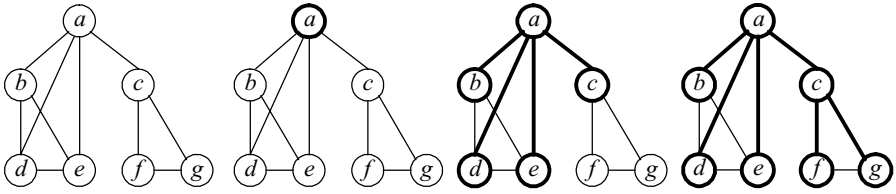


Рис. 52. Поиск в ширину

Поиск в ширину для полного обхода графа с n вершинами и m дугами требует столько же времени, как и поиск в глубину, т. е. времени порядка $O(\max(n, m))$. Поскольку обычно $m \geq n$, то получается $O(m)$.

2.6.3. Нахождение кратчайшего пути

Здесь рассматриваются алгоритмы нахождения путей в ориентированном графе. Эти алгоритмы работают на ориентированном графе, у которого все дуги имеют неотрицательные метки (стоимости дуг). Задача алгоритмов состоит в нахождении кратчайших путей между вершинами графа. Длина пути здесь определяется как сумма меток (длин) дуг, составляющих путь.

2.6.3.1. Алгоритм Дейкстры

Этот алгоритм находит в графе кратчайший путь из заданной вершины, определенной как источник, во все остальные вершины.

В процессе своей работы алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. При этом используется массив **D**, в который записываются длины кратчайших путей для каждой вершины. Когда множество S будет содержать все вершины графа, тогда массив **D** будет содержать длины кратчайших путей от источника к каждой вершине.

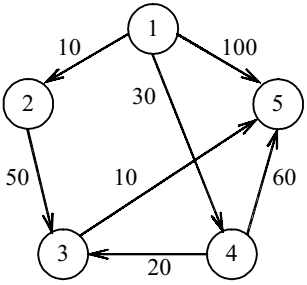
Помимо указанных массивов, в алгоритме Дейкстры используется матрица длин **C**, где элемент **C**$[i, j]$ – метка (длина) дуги (i, j), если дуги нет, то ее длина полагается равной бесконечности, т. е. больше любой фактической длины дуг. Фактически, матрица **C** представляет собой матрицу смежности, в которой все нулевые элементы заменены на бесконечность.

Для определения самого кратчайшего пути (т. е. последовательности вершин) необходимо ввести еще один массив **P** вершин, где **P**$[v]$ содержит вершину, непосредственно предшествующую вершине v в кратчайшем пути (рис. 53).

Алгоритм:

```
procedure Dijkstra;
begin
  S := источник;
  for i := 2 to n do begin
```

```
    D[i] := C[источник, i];
    P[i] := источник;
  end;
  for i := 1 to n-1 do begin
    выбор из множества V\S такой вершины w,
      что значение D[w] минимально;
    добавить w к множеству S;
    for каждая вершина v из множества V\S do begin
      D[v] := min(D[v], D[w] + C[w, v]);
      if D[w] + C[w, v]< D[v] then P[v] := w;
    end;
  end;
end;
```



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

Массив **P**:

	1	4	1	3

Кратчайший путь из 1 в 5: {1, 4, 3, 5}

Рис. 53. Алгоритм Дейкстры

После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива **P**, начиная от конечной вершины к источнику.

Время выполнения этого алгоритма, если для представления графа используется матрица смежности, имеет порядок $O(n^2)$, где n – количество вершин графа.

2.6.3.2. Алгоритм Флойда

Этот алгоритм решает задачу нахождения кратчайших путей между всеми парами вершин графа. Более строгая формулировка этой задачи следующая: есть ориентированный граф $G = (V, E)$, каждой дуге (v, w) этого графа сопоставлена неотрицательная стоимость **C**$[v, w]$. Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин (v, w) любого пути из вершины v

в вершину w, длина которого минимальна среди всех возможных путей из вершины v к w.

Можно решить эту задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника. Но существует прямой способ решения данной задачи, использующий алгоритм Флойда. Для определенности положим, что вершины графа последовательно пронумерованы от 1 до n. Алгоритм Флойда использует матрицу **A** размера $n \times n$, в которой вычисляются длины кратчайших путей. В начале $\mathbf{A}[i, j] = \mathbf{C}[i, j]$ для всех $i <> j$. Если дуга (i, j) отсутствует, то $\mathbf{C}[i, j] = \infty$. Каждый диагональный элемент матрицы **A** равен 0.

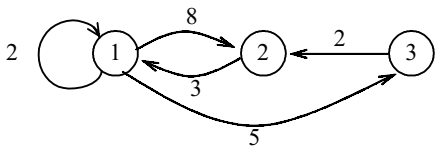
Над матрицей **A** выполняется n итераций. После k-й итерации $\mathbf{A}[i, j]$ содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k (рис. 54).

На k-й итерации для вычисления матрицы **A** применяется следующая формула: $\mathbf{A}_k[i, j] = \min(\mathbf{A}_{k-1}[i, j], \mathbf{A}_{k-1}[i, k] + \mathbf{A}_{k-1}[k, j])$.

Нижний индекс k обозначает значение матрицы **A** после k-й итерации, но это не означает, что существует n различных матриц, этот индекс используется для сокращения записи.

Равенства $\mathbf{A}_k[i, k] = \mathbf{A}_{k-1}[i, k]$ и $\mathbf{A}_k[k, j] = \mathbf{A}_{k-1}[k, j]$ означают, что на k-й итерации элементы матрицы **A**, стоящие в k-й строке и k-м столбце, не изменяются. Более того, все вычисления можно выполнить с применением только одного экземпляра матрицы **A**. Представим алгоритм Флойда в виде следующей процедуры:

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



	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$\mathbf{A}_0$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$\mathbf{A}_1$

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

$\mathbf{A}_2$

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

$\mathbf{A}_3$

Рис. 54. Алгоритм Флойда

Следует заметить, что если в графе существует контур отрицательной суммарной длины, то вес любого пути, проходящего через вершину из этого контура, можно сделать сколь угодно малой, «прокрутившись» в контуре необходимое количество раз. Поэтому поставленная задача разрешима не всегда. В случае, описанном выше, алгоритм Флойда не применим. Останавливаясь подробнее надо заметить, что если граф неориентированный, то ребро с отрицательным весом является как раз таким контуром (проходя по нему в обоих направлениях столько раз пока не сделаем вес достаточно малым).

Заметим, что если граф неориентированный, то все матрицы, получаемые в результате преобразований симметричны и, следовательно, достаточно вычислять только элементы расположенные выше главной диагонали.

Время выполнения этого алгоритма, очевидно, имеет порядок $O(n^3)$, поскольку в нем присутствуют вложенные друг в друга три цикла.

2.6.3.3. Переборные алгоритмы

Рассмотрим переборные алгоритмы, основанные на методах обхода графа (см. п. 2.6.2) на примере задачи нахождения кратчайшего пути в лабиринте. Поскольку существует два метода обхода графа, то и переборных алгоритмов будем рассматривать два.

Лабиринт, состоящий из проходимых и непроходимых клеток, задан матрицей **A** размером $m \times n$. Элемент матрицы $\mathbf{A}[i, j] = 0$, если клетка (i, j) проходима. В противном случае $\mathbf{A}[i, j] = \infty$.

Требуется найти кратчайший путь из клетки (1, 1) в клетку (m, n).

Фактически дана инвертированная матрица смежности (в ней нули заменены ∞, а единицы – нулями). Лабиринт представляет собой граф.

Метод перебора с возвратом (по-английски называемый backtracking) основан на методе поиска в глубину. Перебор с возвратом – это метод проб и ошибок («попробуем сходить в эту сторону: не получится – вернемся и попробуем в другую»). Поскольку речь идет о переборе вариантов методом поиска в глубину, то во время работы алгоритма надо хранить текущий путь в дереве. Этот путь представляет собой стек `Way`. Кроме того, необходим массив `Dest`, размерность которого соответствует количеству вершин графа, хранящий для каждой вершины расстояние от нее до исходной вершины.

Вернемся к нашей задаче. Пусть текущей является некоторая клетка (в начале работы алгоритма – клетка (1, 1)). Далее

```
if для текущей клетки есть клетка-сосед Neighbor, такая что:
  (отсутствует в Way) and
  (Dist[Neighbor]=0 or (Dist[Neighbor] > Length(Way))) then begin
  добавить Neighbor в Way;
  текущая клетка := Neighbor;
end else
  извлечь из Way;
```

Из приведенного выше фрагмента ясно, почему этот метод называется перебором с возвратом. Возврату здесь соответствует операция «извлечь из `Way`», которая уменьшает длину `Way` на 1.

Перебор заканчивается, когда `Way` пуст и делается попытка возврата назад. В этой ситуации возвращаться уже некуда (рис. 55).

`Way` – это путь текущий, но в процессе работы необходимо хранить еще и оптимальный путь `OptimalWay`.

Заметим, что алгоритм можно усовершенствовать, если не позволять, чтобы длина `Way` была больше или равна длине `OptimalWay`. В этом случае если и будет найден какой-то вариант большей длины, он заведомо не будет оптимальным. Такое усовершенствование в общем случае означает, что как только текущий путь станет заведомо неоптимальным, надо вернуться назад. В некоторых случаях это улучшение алгоритма позволяет сильно сократить перебор.

Переборный алгоритм, основанный на поиске в ширину, состоит из двух этапов:

1) распространение волны;

2) обратный ход.

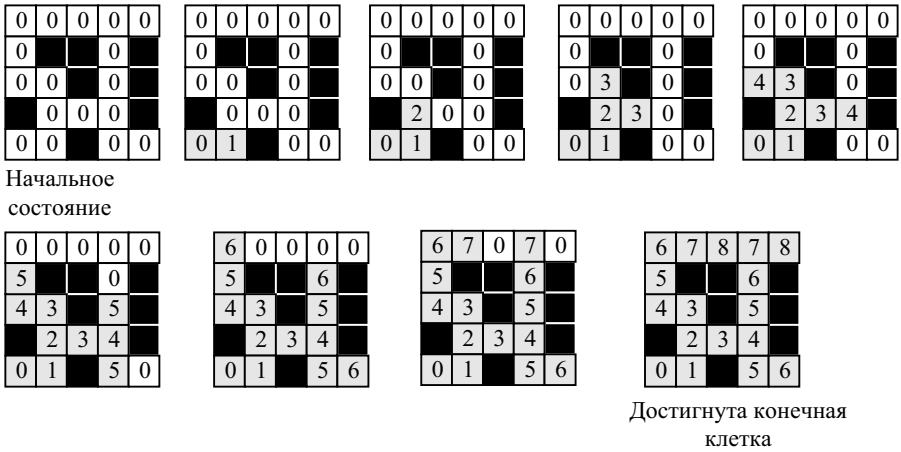
Распространение волны и есть собственно поиск в ширину (см. п. 2.6.2.2), при котором клетки помечаются номером шага метода, на

OptimalWay = (1,1), (1,2), (2,2), (2,3), (2,4), (3,4), (4,4), (5,4), (5,5)

Length(Way) = 8

Рис. 55. Перебор методом поиска в глубину

котором клетка посещается. При обратном ходе, начиная с конечной вершины, идет восстановление кратчайшего пути, по которому в нее попали путем включения в него клеток с минимальной пометкой.



OptimalWay = (1,1), (1,2), (2,2), (2,3), (2,4), (3,4), (4,4), (5,4), (5,5);

Length(Way) = 8

Рис. 56. Перебор методом поиска в ширину

Важно, что восстановление начинается с конца (с начала оно зачастую невозможно) (рис. 56).

Надо сказать, что перебор методом поиска в ширину по сравнению с перебором с возвратом, как правило, требует больше дополнительной памяти, расходуемой на хранение информации нужной для построения пути при обратном ходе и пометки посещенных вершин, но и работает быстрее, так как совершенно исключается посещение одной и той же клетки более, чем один раз.

2.6.4. Нахождение минимального остовного дерева

Приведем без доказательства следующее свойство MST (minimal spanning tree – минимальное остовное дерево на английском языке).

В графе $G = (V, E)$ рассмотрим U – некоторое подмножество V, такое что U и $V \backslash U$ не пусты. Пусть (u, v) – ребро наименьшей стоимости, одна вершина которого – $u \in U$, а другая – $v \in V \backslash U$. Тогда существует некоторое MST, содержащее ребро (u, v). Пример графа G и его минимального основного дерева приведен на рис. 57.

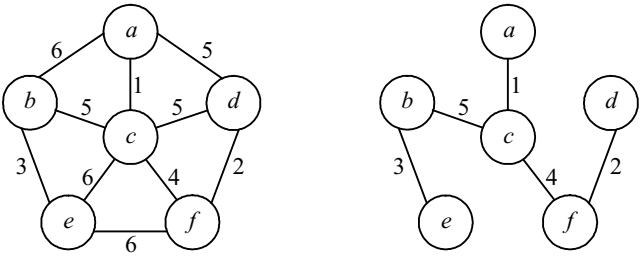


Рис. 57. Граф и его остовное дерево минимальной длины

На этом свойстве основаны два известных алгоритма.

2.6.4.1. Алгоритм Прима

В этом алгоритме строится множество вершин U, из которого «вырастает» остовное дерево (рис. 58).

Сначала $U = \varnothing$. На каждом шаге алгоритма находится ребро наименьшей стоимости (u, v) такое, что $u \in U$, $v \in V \backslash U$, затем вершина v переносится из $V \backslash U$ в U. Этот процесс продолжается до тех пор, пока множество U не станет равным множеству V:

```
U := ∅;
U:= U ∪ любая вершина;
```

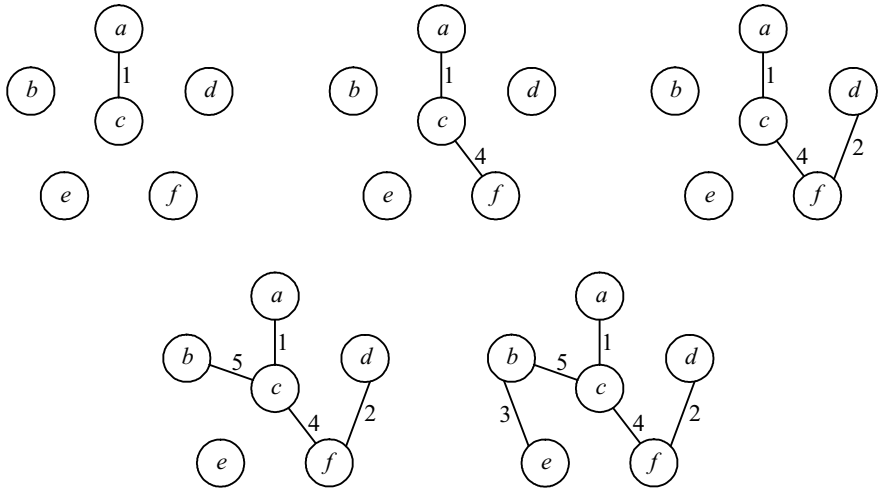


Рис. 58. Алгоритм Прима

```
while V\U <> ∅ do begin
  Выбрать ребро (u, v) наименьшей стоимости, где u∈U, v∈V\U;
  U:= U ∪ v;
  V\U := (V\U) ∩ v;
end;
```

Очевидно, данный алгоритм для графа с n вершинами имеет сложность, пропорциональную $O(n^2)$

2.6.4.2. Алгоритм Крускала

Здесь построение MST начинается с графа, состоящего только из n вершин графа G и не имеющего ребер. Таким образом, каждая вершина является связанной (сама с собой) компонентой. Это дает n связанных компонентов. В процессе выполнения алгоритма связанные компоненты постепенно объединяются друг с другом, формируя остовное дерево (рис. 59).

При построении постепенно возрастающих связанных компонент поочередно проверяются ребра из множества E в порядке возрастания их длин. Если очередное ребро связывает две вершины из разных компонент, тогда оно добавляется в остовное дерево. Если это ребро связывает две вершины из одной компоненты, то оно отбрасывается, так как его добавление в связанную компоненту может

привести к образованию цикла. Число ребер, необходимое для остовного дерева равно n–1. Граф связан, а значит E содержит как минимум такое их количество. Когда остовное дерево будет содержать n–1 ребер, алгоритм завершается:

```
Создать список ребер L, в неубывающем по длине порядке
while число отмеченных ребер < n-1 do begin
  Удалить w из головы списка L;
  if w соединяет две несвязанных компоненты then
    отметить w и добавить к MST
  else                          {w - внутри компоненты}
    не отмечать w               {это приведет к циклу в MST}
end;
```

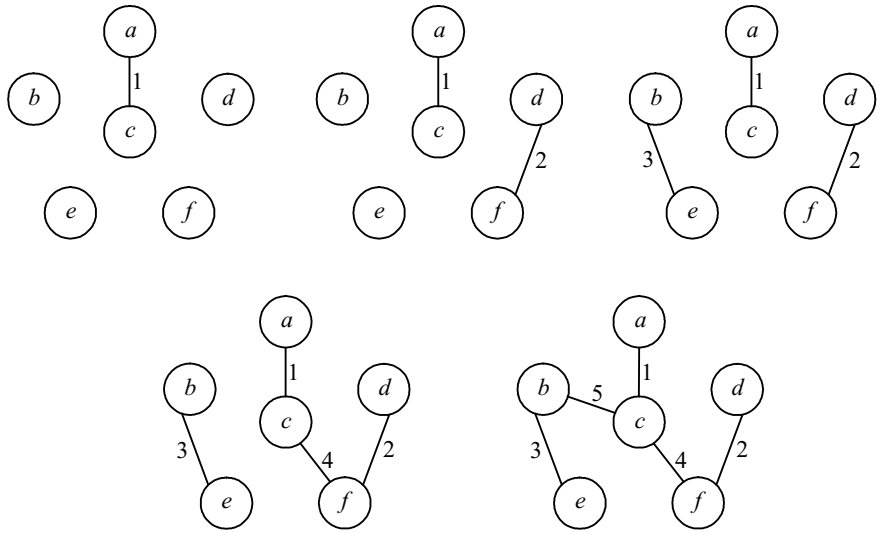


Рис. 59. Алгоритм Крускала

Сложность алгоритма для графа с n вершинами и m ребрами пропорциональна $O(m\log m)$.

Библиографический список

1. **Ахо А., Хопкрофт Дж., Ульман Дж*. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

2. *Ахо А., Хопкрофт Дж., Ульман Дж*. Структуры данных и алгоритмы. М.: Вильямс, 2001. 384 с.

3. **Бентли Д*. Жемчужины творчества программистов. М.: Радио и связь, 1990.

4. **Вирт Н.* Алгоритмы + структуры данных = программы. М.: Мир, 1985.

5. **Вирт Н.* Алгоритмы и структуры данных. М: Мир, 1989. 360 с.

6. **Грин Д., Кнут Д.* Математические методы анализа алгоритмов. М: Мир, 1987.

7. **Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Мир, 1981.

8. **Дейкстра Э.* Дисциплина программирования. М: Мир, 1978.

9. **Кнут Д. Е.* Искусство программирования для ЭВМ: В 3 т. М.: Мир, 1976.

10. *Кнут Д. Е.* Искусство программирования: В 3 т. М.: Вильямс, 2000.

11. **Кормен Т., Лейзерсон Ч., Ривест Р*. Алгоритмы: Построение и анализ. М.: МЦНМО, 2001.

12. *Лэгсам Й., Огенстайн М.* Структуры данных для персональных ЭВМ. М.: Мир, 1989. 586 с.

13. *Структуры и алгоритмы обработки данных/ *В. А. Матьяш, В. А. Путилов, В. В. Фильчаков, С. В. Щекин.* Апатиты: КФ ПетрГУ, 2000. 80 с.

14. **Оре О*. Графы и их применение. М.: Мир, 1965.

15. **Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980.

16. **Сибуя М., Ямамото Т*. Алгоритмы обработки данных. М.: Мир, 1986. 218 с.

17. **Успенский В. А., Семенов А. Л.* Теория алгоритмов: основные открытия и приложения. М.: Наука, 1987.

18. **Харари Ф*. Теория графов. М.: Мир, 1973.

---

* Имеются в наличии в библиотеке ГУАП.

Русскоязычные ресурсы Internet

1. http://algo.4u.ru/
2. http://algolist.manual.ru/
3. http://alglib.chat.ru/
4. http://algo.do.ru/
5. http://hcinsu.chat.ru/
6. http://algolist.da.ru/
7. http://progstone.narod.ru/links/wantalgo.html
8. http://www.sevmashvtuz.edu/links/algorithms.html

ОГЛАВЛЕНИЕ

Учебное издание

Ключарев Александр Анатольевич
Матьяш Валерий Анатольевич
Щекин Сергей Валерьевич

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Редактор *Г. Д. Бакастова*
Компьютерная верстка *А. Н. Колешко*

Сдано в набор 03.02.04. Подписано к печати 05.07.04. Формат 60×84 1/16. Бумага офсетная. Печать офсетная. Усл. печ. л. 9,99. Усл. кр.-отт. 10,12. Уч. -изд. л. 10,34. Тираж 200 экз. Заказ №

Редакционно-издательский отдел
Отдел электронных публикаций и библиографии библиотеки
Отдел оперативной полиграфии
СПбГУАП
190000, Санкт-Петербург, ул. Б. Морская, 67