```
procedure shellsort;
   const t = 4;
   var i,j,k,s: index;  x: item; m: 1 . . t;
      h: array [1 . . t] of integer;
begin h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
   for m := 1 to t do
   begin k := h[m]; s := -k; {место барьера}
      for i := k+1 to n do
      begin x := a[i]; j := i-k;
         if s=0 then s := -k; s := s+1; a[s] := x;
         while x .key < a[j] .key do
         begin a[j+k] := a[j]; j := j-k
         end ;
         a[j+k] := x
      end
   end
end
```

Программа 2.6. Сортировка Шелла.

происходило как можно чаще. Можно сформулировать следующую теорему:

Если k-рассортированная последовательность i-сортируется, то она остается k-рассортированной.

Кнут [2.8] указывает, что разумным выбором может быть такая последовательность приращений (записанная в обратном порядке):

$$1,\ 4,\ 13,\ 40,\ 121,\ \ldots,$$

где $h_{k-1} = 3h_k + 1$, $h_t = 1$ и $t = [\log_3 n] - 1$. Он рекомендует также последовательность

$$1,\ 3,\ 7,\ 15,\ 31,\ \ldots,$$

где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = [\log_2 n] - 1$. Дальнейший анализ показывает, что в последнем случае затраты, которые требуются для сортировки n элементов с помощью алгоритма сортировки Шелла, пропорциональны n. Хотя это — значительное улучшение по сравнению с n^2, мы не будем в дальнейшем обращаться к этому методу, поскольку известны алгоритмы, работающие еще лучше.

2.2.5. Сортировка с помощью дерева

Метод сортировки простым выбором основан на повторном выборе наименьшего ключа среди n элементов, затем среди $n-1$ элементов и т. д. Понятно, что поиск наименьшего

ключа из n элементов требует $n-1$ сравнений, а поиск его среди $n-1$ элементов требует $n-2$ сравнений. Итак, как можно улучшить эту сортировку выбором? Это можно сделать только в том случае, если получать от каждого прохода больше информации, чем просто указание на один, наименьший элемент. Например, с помощью $n/2$ сравнений можно определить наименьший ключ из каждой пары, при помощи

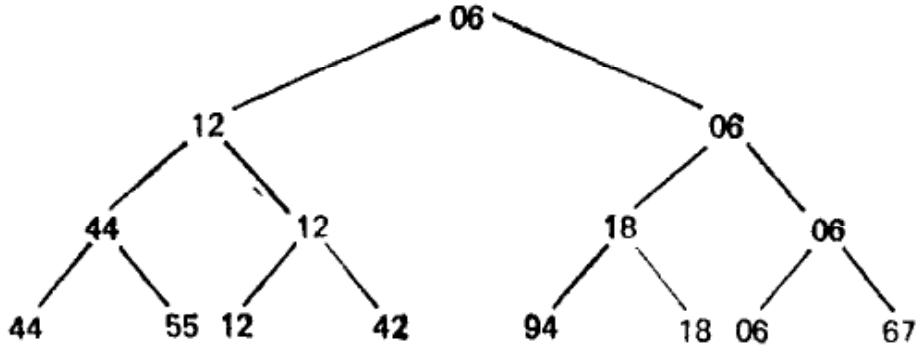

Рис. 2.3. Циклический выбор из двух ключей.

следующих $n/4$ сравнений можно выбрать наименьший из каждой пары таких наименьших ключей и т. д. Наконец, при помощи всего $n-1$ сравнений мы можем построить дерево выбора, как показано на рис. 2.3, и определить корень как наименьший ключ [2.2].

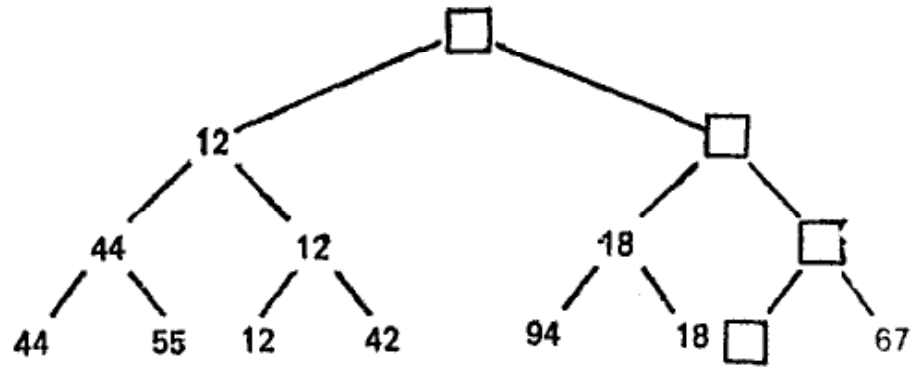

Рис. 2.4. Выбор наименьшего ключа.

На втором шаге мы спускаемся по пути, указанному наименьшим ключом, и исключаем его, последовательно заменяя либо на «дыру» (или ключ ∞), либо на элемент, находящийся на противоположной ветви промежуточного узла (см. рис. 2.4 и 2.5). Элемент, оказавшийся в корне дерева, вновь имеет наименьший ключ (среди оставшихся) и может быть исключен. После n таких шагов дерево становится пустым (т. е. состоит из «дыр»), и процесс сортировки закончен. Отметим, что каждый из n шагов требует лишь $\log_2 n$ сравнений. Поэтому вся сортировка требует лишь порядка $n \cdot \log_2 n$ элементарных операций, не считая n шагов, которые необходимы для построения дерева. Это — значительное улуч-

шение по сравнению с простым методом, требующим n^2 шагов и даже по сравнению с сортировкой Шелла, которая требует $n^{1.2}$ шагов.

Конечно, при сортировке с помощью дерева задача хранения информации стала сложнее и поэтому увеличилась сложность отдельных шагов; в конечном счете для хранения возросшего объема информации, получаемой на начальном проходе, нужно строить некую древовидную структуру. Наша очередная задача — найти способы эффективной организации этой информации.

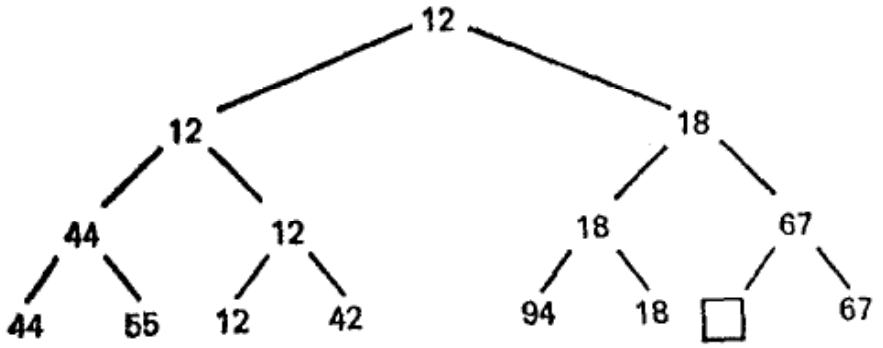

Рис. 2.5. Заполнение «дыр».

Разумеется, было бы весьма желательно избавиться от необходимости в дырах ($-\infty$), которые в конце заполняют все дерево и приводят к большому количеству ненужных сравнений. Кроме того, нужно найти способ представить дерево из n элементов в n единицах памяти вместо $2n-1$ единиц, как показано выше. Это действительно можно сделать с помощью метода, который его изобретатель Дж. Уильямс [2.14] назвал *пирамидальной сортировкой*. Ясно, что этот метод дает существенное улучшение по сравнению с более привычными способами сортировки по дереву.

Пирамида определяется как последовательность ключей

$$h_l,\ h_{l+1},\ \ldots,\ h_r$$

такая, что

$$\begin{aligned} h_i &\leqslant h_{2i}, \\ h_i &\leqslant h_{2i+1} \end{aligned} \qquad (2.13)$$

для всякого $i = l,\ \ldots,\ r/2$. Если двоичное дерево представлено в виде массива, как показано на рис. 2.6, то, следовательно, деревья сортировки на рис. 2.7 и 2.8 являются пирамидами, и, в частности, элемент h_1 пирамиды является ее *наименьшим* элементом

$$h_1 = \min (h_1 \ldots h_n).$$

Теперь предположим, что дана пирамида с элементами $h_{l+1},\ \ldots,\ h_r$ для некоторых значений l и r и нужно добавить

новый элемент x для того, чтобы сформировать расширенную пирамиду $h_l, \ldots, h_r$. Возьмем, например, исходную пирамиду $h_1, \ldots, h_7$, показанную на рис. 2.7, и расширим эту пирамиду «влево», добавив элемент $h_1 = 44$. Новый элемент x сначала помещается в вершину дерева, а затем «просеивается» по пути, на котором находятся меньшие по сравнению с ним элементы, которые одновременно поднимаются вверх; таким

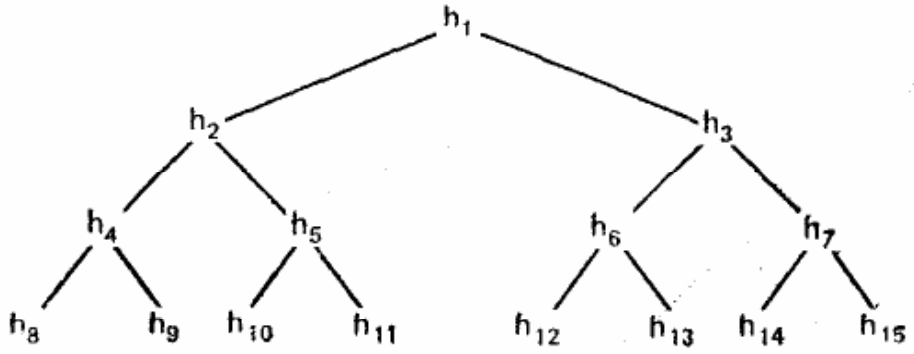

Рис. 2.6. Массив h, расположенный в виде бинарного дерева.

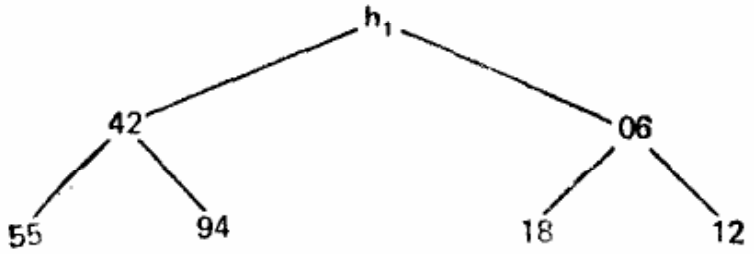

Рис. 2.7. Пирамида из семи элементов.

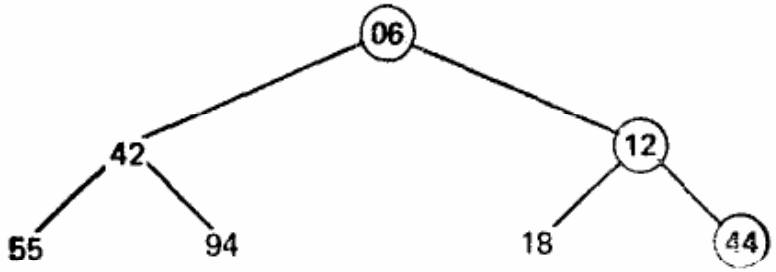

Рис. 2.8. Просеивание ключа 44 через пирамиду.

образом формируется новая пирамида. В данном примере значение 44 сначала меняется местами с 06, затем с 12, и так формируется дерево, показанное на рис. 2.8. Далее мы процесс просеивания будем формулировать следующим образом: i, j — пара индексов, обозначающих элементы, которые нужно менять местами на каждом шаге просеивания. Мы предоставляем читателю возможность самому убедиться, что предложенный способ просеивания действительно позволяет сохранить условия (2.13), определяющие пирамиду.

Изящный способ построения пирамиды *in situ* был предложен Р. У. Флойдом. В нем используется следующая про-

цедура просеивания (программа 2.7). Дан массив $h_1, \ldots, h_n$; ясно, что элементы $h_{n/2+1} \ldots h_n$ уже образуют пирамиду, поскольку не существует двух индексов i, j, таких, что $j = 2i$ (или $j = 2i + 1$). Эти элементы составляют последовательность, которую можно рассматривать как нижний ряд соответствующего двоичного дерева (см. рис. 2.6), где не тре-

```
procedure sift(l,r: index);
    label 13;
    var i,j: index;  x: item;
begin i := l; j := 2*i; x := a[i];
    while j ≤ r do
    begin if j < r then
              if a[j] .key > a[j+1] .key then j := j+1;
          if x .key ≤ a[j] .key then goto 13;
          a[i] := a[j]; i := j; j := 2*i {sift}
    end;
13: a[i] := x
end
```

Программа 2.7. Просеивание.

буется никакого упорядочения. Теперь пирамида расширяется влево: на каждом шаге добавляется новый элемент и при помощи просеивания помещается на соответствующее место.

Таблица 2.6. Построение пирамиды

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

Этот процесс иллюстрируется табл. 2.6 и приводит к пирамиде, показанной на рис. 2.6. Следовательно, процесс построения пирамиды из n элементов *in situ* можно описать следующим образом:

```
l := (n div 2) + 1;
while l > 1 do
    begin l := l−1; sift(l,n)
    end
```

Для того чтобы рассортировать элементы, надо выполнить n шагов просеивания: после каждого шага очередной элемент берется с вершины пирамиды. Вновь встает вопрос, куда помещать элементы с вершины и возможна ли сортировка *in situ*. Да, такое решение существует! На каждом шаге из пирамиды выбирается последняя компонента (скажем, x), верхний элемент пирамиды помещается на освободившееся место x, а x просеивается на свое место. В этом случае необходимо совершить $n-1$ шагов, что показано на примере

Таблица 2.7. Пример пирамидальной сортировки

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

пирамиды, приведенной в табл. 2.7. Этот процесс описывается с помощью процедуры *sift* (программа 2.7) следующим образом:

```
r := n;
while r > 1 do
    begin x := a[1]; a[1] := a[r]; a[r] := x;
        r := r-1; sift(1,r)
    end
```

Из табл. 2.7 видно, что на самом деле в результате мы получаем последовательность в обратном порядке. Но это легко можно исправить, изменив направление отношения порядка в процедуре *sift*. В результате мы получаем процедуру *Heapsort*, показанную в программе 2.8.

Анализ пирамидальной сортировки. С первого взгляда неочевидно, что этот метод сортировки дает хорошие результаты. Ведь элементы с большими ключами вначале просеиваются влево, прежде чем, наконец, окажутся справа. Действительно, эта процедура не рекомендуется для такого небольшого числа элементов, как, скажем, в нашем примере. Однако для больших n пирамидальная сортировка оказы-

```
procedure heapsort;
   var l,r: index;  x: item;
   procedure sift;
      label 13;
      var i,j: index;
   begin i := l; j := 2*i; x := a[i];
      while j ≤ r do
      begin if j < r then
                  if a[j] .key < a[j+1] .key then j := j+1;
         if x .key ≥ a[j] .key then goto 13;
         a[i] := a[j]; i := j; j := 2*i
      end ;
   13: a[i] := x
   end ;
begin l := (n div 2) + 1; r := n;
   while l > 1 do
      begin l := l-1; sift
      end ;
   while r > 1 do
      begin x := a[1]; a[1] := a[r]; a[r] := x;
         r := r-1; sift
      end
end {heapsort}
```

Программа 2.8. Пирамидальная сортировка.

вается очень эффективной, и чем больше n, тем она эффективнее — даже по сравнению с сортировкой Шелла.

В худшем случае необходимы $n/2$ шагов, которые просеивают элементы через $\log(n/2)$, $\log(n/2-1)$, ..., $\log(n-1)$ позиций (здесь берется целая часть логарифма по основанию 2). Следовательно, на фазе сортировки происходит $n-1$ просеиваний с самое большее $\log(n-1)$, $\log(n-2)$, ..., 1 пересылками. Кроме того, требуются $n-1$ пересылок для того, чтобы отложить просеянный элемент вправо. Отсюда видно, что пирамидальная сортировка требует $n \cdot \log(n)$ шагов даже *в худшем случае*. Такие отличные характеристики для худшего случая — одно из самых выгодных качеств пирамидальной сортировки.

Не совсем ясно, в каких случаях можно ожидать наименьшей (или наибольшей) эффективности. Но в принципе для пирамидальной сортировки, видимо, больше всего подходят случаи, когда элементы более или менее рассортированы в обратном порядке, т. е. для нее характерно неестественное

поведение. Очевидно, что при обратном порядке фаза построения пирамиды не требует никаких пересылок. Для восьми элементов из нашего примера минимальное и максимальное количества пересылок дают следующие исходные последовательности:

$M_{\min} = 13$ для последовательности
94 67 44 55 12 42 18 6

$M_{\max} = 24$ для последовательности
18 42 12 44 6 55 67 94

Среднее число пересылок равно приблизительно $\frac{1}{2} n \cdot \log n$ и отклонения от этого значения сравнительно малы.

2.2.6. Сортировка с разделением

После того как мы обсудили два усовершенствованных метода сортировки, основанных на принципах включения и выбора, мы введем третий, улучшенный метод, основанный на принципе обмена. Учитывая, что сортировка методом пузырька в среднем была наименее эффективной из трех алгоритмов простой сортировки, мы должны требовать значительного улучшения. Однако неожиданно оказывается, что усовершенствование сортировки, основанной на обмене, которое мы здесь будем обсуждать, дает вообще лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель К. Хоор окрестил его *быстрой сортировкой* [2.5, 2.6].

Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях. Предположим, что нам даны n элементов с ключами, расположенными в обратном порядке. Их можно рассортировать, выполнив всего $n/2$ обменов, если сначала поменять местами самый левый и самый правый элементы и так постепенно продвигаться с двух концов к середине. Разумеется, это возможно, только если мы знаем, что элементы расположены строго в обратном порядке. Но все же этот пример наводит на некоторые мысли.

Попробуем рассмотреть следующий алгоритм: выберем случайным образом какой-то элемент (назовем его x), просмотрим массив, двигаясь слева направо, пока не найдем элемент $a_i > x$, а затем просмотрим его справа налево, пока не найдем элемент $a_j < x$. Теперь поменяем местами эти два элемента и продолжим процесс «просмотра с обменом», пока два просмотра не встретятся где-то в середине массива. В результате массив разделится на две части: левую — с клю-