ГЛАВА 6

Пирамидальная сортировка

В этой главе описывается еще один алгоритм сортировки, а именно — пирамидальная сортировка. Время работы этого алгоритма, как и время работы алгоритма сортировки слиянием (и в отличие от времени работы алгоритма сортировки вставкой), равно $O(n \lg n)$. Как и сортировка методом вставок, и в отличие от сортировки слиянием, пирамидальная сортировка выполняется без привлечения дополнительной памяти: в любой момент времени требуется память для хранения вне массива только некоторого постоянного количества элементов. Таким образом, в пирамидальной сортировке сочетаются лучшие особенности двух рассмотренных ранее алгоритмов сортировки.

В ходе рассмотрения пирамидальной сортировки мы также познакомимся с еще одним методом разработки алгоритмов, а именно — использованием специализированных структур данных для управления информацией в ходе выполнения алгоритма. В рассматриваемом случае такая структура данных называется "пирамидой" (heap) и может оказаться полезной не только при пирамидальной сортировке, но и при создании эффективной очереди с приоритетами. В последующих главах эта структура данных появится снова.

Изначально термин "heap" использовался в контексте пирамидальной сортировки (heapsort), но в последнее время его основной смысл изменился, и он стал обозначать память со сборкой мусора, в частности, в языках программирования Lisp и Java (и переводиться как "куча"). Однако в данной книге термину heap (который здесь переводится как "пирамида") возвращен его первоначальный смысл.

6.1 Пирамиды

Пирамида (binary heap) — это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево (см. раздел 5.3 приложения Б, а также рис. 6.1). Каждый узел этого дерева соответствует определенному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненный уровень — это такой, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы. Представляющий пирамиду массив A является объектом с двумя атрибутами: $length\,[A]$, т.е. количество элементов массива, и $heap\_size\,[A]$, т.е. количество элементов пирамиды, содержащихся в массиве A. Другими словами, несмотря на то, что в массиве $A\,[1..\,length\,[A]]$ все элементы могут быть корректными числами, ни один из элементов, следующих после элемента $A\,[heap\_size\,[A]]$, где $heap\_size\,[A] \leqslant length\,[A]$, не является элементом пирамиды. В корне дерева находится элемент $A\,[1]$, а дальше оно строится по следующему принципу: если какому-то узлу соответствует индекс i, то индекс его родительского узла вычисляется с помощью представленной ниже процедуры PARENT(i), индекс левого дочернего узла — с помощью процедуры LEFT(i), а индекс правого дочернего узла — с помощью процедуры RIGHT(i):

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

В пирамиде, представленной на рис. 6.1, число в окружности, представляющей каждый узел дерева, является значением, сортируемым в данном узле. Число над узлом — это соответствующий индекс массива. Линии, попарно соединяющие элементы массива, обозначают взаимосвязь вида "родитель-потомок". Родительские элементы всегда расположены слева от дочерних. Данное дерево имеет высоту, равную 3; узел с индексом 4 (и значением 8) расположен на первом уровне.

На большинстве компьютеров операция $2i$ в процедуре LEFT выполняется при помощи одной команды процессора путем побитового сдвига числа i на один бит влево. Операция $2i + 1$ в процедуре RIGHT тоже выполняется очень быстро — достаточно биты двоичного представления числа i сдвинуть на одну позицию влево, а затем младший бит установить равным 1. Процедура PARENT выполняется путем сдвига числа i на один бит вправо. При реализации пирамидальной сортировки эти функции часто представляются в виде макросов или встраиваемых процедур.

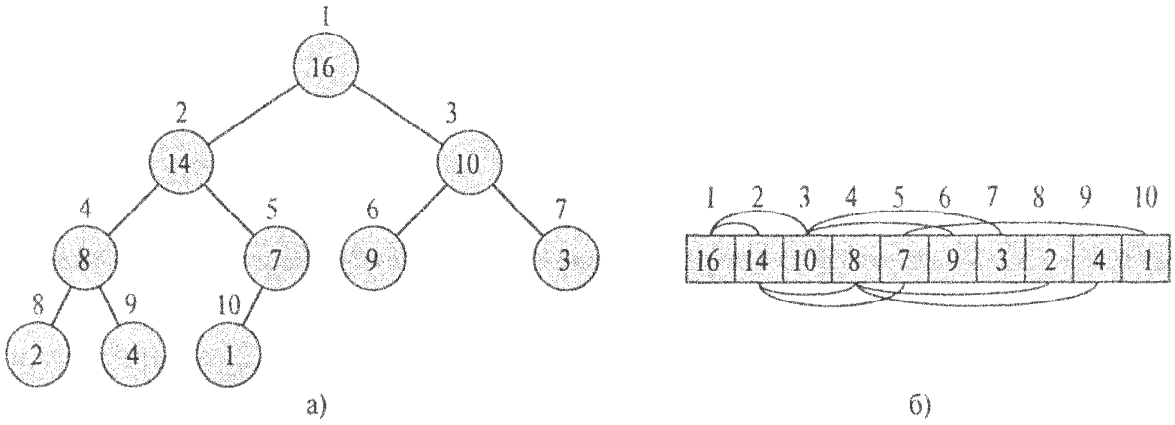

Рис. 6.1. Пирамида, представленная в виде *а*) бинарного дерева и *б*) массива

Различают два вида бинарных пирамид: неубывающие и невозрастающие. В пирамидах обоих видов значения, расположенные в узлах, удовлетворяют ***свойству пирамиды*** (heap property), являющемуся отличительной чертой пирамиды того или иного вида. ***Свойство невозрастающих пирамид*** (max-heap property) заключается в том, что для каждого отличного от корневого узла с индексом i выполняется следующее неравенство:

$$A[\text{PARENT}(i)] \geqslant A[i].$$

Другими словами, значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации ***неубывающей пирамиды*** (min-heap) прямо противоположный. Свойство ***неубывающих пирамид*** (min-heap property) заключается в том, что для всех отличных от корневого узлов с индексом i выполняется такое неравенство:

$$A[\text{PARENT}(i)] \leqslant A[i].$$

Таким образом, наименьший элемент такой пирамиды находится в ее корне.

В алгоритме пирамидальной сортировки используются невозрастающие пирамиды. Неубывающие пирамиды часто применяются в приоритетных очередях (этот вопрос обсуждается в разделе 6.5). Для каждого приложения будет указано, с пирамидами какого вида мы будем иметь дело — с неубывающими или невозрастающими. При описании свойств как неубывающих, так и невозрастающих пирамид будет использоваться общий термин "пирамида".

Рассматривая пирамиду как дерево, определим ***высоту*** ее узла как число ребер в самом длинном простом нисходящем пути от этого узла к какому-то из листьев дерева. Высота пирамиды определяется как высота его корня. Поскольку n-элементная пирамида строится по принципу полного бинарного дерева, то ее

высота равна $\Theta(\lg n)$ (см. упражнение 6.1-2). Мы сможем убедиться, что время выполнения основных операций в пирамиде приблизительно пропорционально высоте дерева, и, таким образом, эти операции требуют для работы время $O(\lg n)$. В остальных разделах этой главы представлены несколько базовых процедур и продемонстрировано их использование в алгоритме сортировки и в структуре данных очереди с приоритетами.

- Процедура MAX_HEAPIFY выполняется в течение времени $O(\lg n)$ и служит для поддержки свойства невозрастания пирамиды.
- Время выполнения процедуры BUILD_MAX_HEAP увеличивается с ростом количества элементов линейно. Эта процедура предназначена для создания невозрастающей пирамиды из неупорядоченного входного массива.
- Процедура HEAPSORT выполняется в течение времени $O(n \lg n)$ и сортирует массив без привлечения дополнительной памяти.
- Процедуры MAX_HEAP_INSERT, HEAP_EXTRACT_MAX, HEAP_INCREASE_KEY и HEAP_MAXIMUM выполняются в течение времени $O(\lg n)$ и позволяют использовать пирамиду в качестве очереди с приоритетами.

Упражнения

6.1-1. Чему равно минимальное и максимальное количество элементов в пирамиде высотой h?

6.1-2. Покажите, что n-элементная пирамида имеет высоту $\lfloor \lg n \rfloor$.

6.1-3. Покажите, что в любом поддереве невозрастающей пирамиды значение корня этого поддерева не превышает значений, содержащихся в других его узлах.

6.1-4. Где в невозрастающей пирамиде может находиться наименьший ее элемент, если все элементы различаются по величине?

6.1-5. Является ли массив с отсортированными элементами неубывающей пирамидой?

6.1-6. Является ли последовательность $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ невозрастающей пирамидой?

6.1-7. Покажите, что если n-элементную пирамиду представить в виде массива, то ее листьями будут элементы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Поддержка свойства пирамиды

Процедура MAX_HEAPIFY является важной подпрограммой, предназначенной для работы с элементами невозрастающих пирамид. На ее вход подается массив A и индекс i этого массива. При вызове процедуры MAX_HEAPIFY предполагается, что бинарные деревья, корнями которых являются элементы LEFT(i) и RIGHT(i), являются невозрастающими пирамидами, но сам элемент $A[i]$ может быть меньше своих дочерних элементов, нарушая тем самым свойство невозрастающей пирамиды. Функция MAX_HEAPIFY опускает значение элемента $A[i]$ вниз по пирамиде до тех пор, пока поддерево с корнем, отвечающем индексу i, не становится невозрастающей пирамидой:

```
MAX_HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap_size[A] и A[l] > A[i]
 4    then largest ← l
 5    else largest ← i
 6  if r ⩽ heap_size[A] и A[r] > A[largest]
 7    then largest ← r
 8  if largest ≠ i
 9    then Обменять A[i] ↔ A[largest]
10         MAX_HEAPIFY(A, largest)
```

Действие процедуры MAX_HEAPIFY проиллюстрировано на рис. 6.2. На каждом этапе ее работы определяется, какой из элементов — $A[i]$, $A[Left(i)]$ или $A[Right(i)]$ — является максимальным, и его индекс присваивается переменной $largest$. Если наибольший элемент — $A[i]$, то поддерево, корень которого находится в узле с индексом i, — невозрастающая пирамида, и процедура завершает свою работу. В противном случае максимальное значение имеет один из дочерних элементов, и элемент $A[i]$ меняется местами с элементом $A[largest]$. После этого узел с индексом i и его дочерние узлы станут удовлетворять свойству невозрастающей пирамиды. Однако теперь исходное значение элемента $A[i]$ присвоено элементу с индексом $largest$, и свойство невозрастающей пирамиды может нарушиться в поддереве с этим корнем. Для устранения нарушения для этого дерева необходимо рекурсивно вызвать процедуру MAX_HEAPIFY.

На рис. 6.2 показана работа процедуры MAX_HEAPIFY$(A, 2)$. В части *а* этого рисунка показана начальная конфигурация, в которой элемент $A[2]$ нарушает свойство невозрастающей пирамиды, поскольку он меньше, чем оба дочерних узла. Поменяв местами элементы $A[2]$ и $A[4]$, мы восстанавливаем это свойство в узле 2, но нарушаем его в узле 4 (часть *б* рисунка). Теперь в рекурсивном вызове процедуры MAX_HEAPIFY$(A, 4)$ в качестве параметра выступает значение

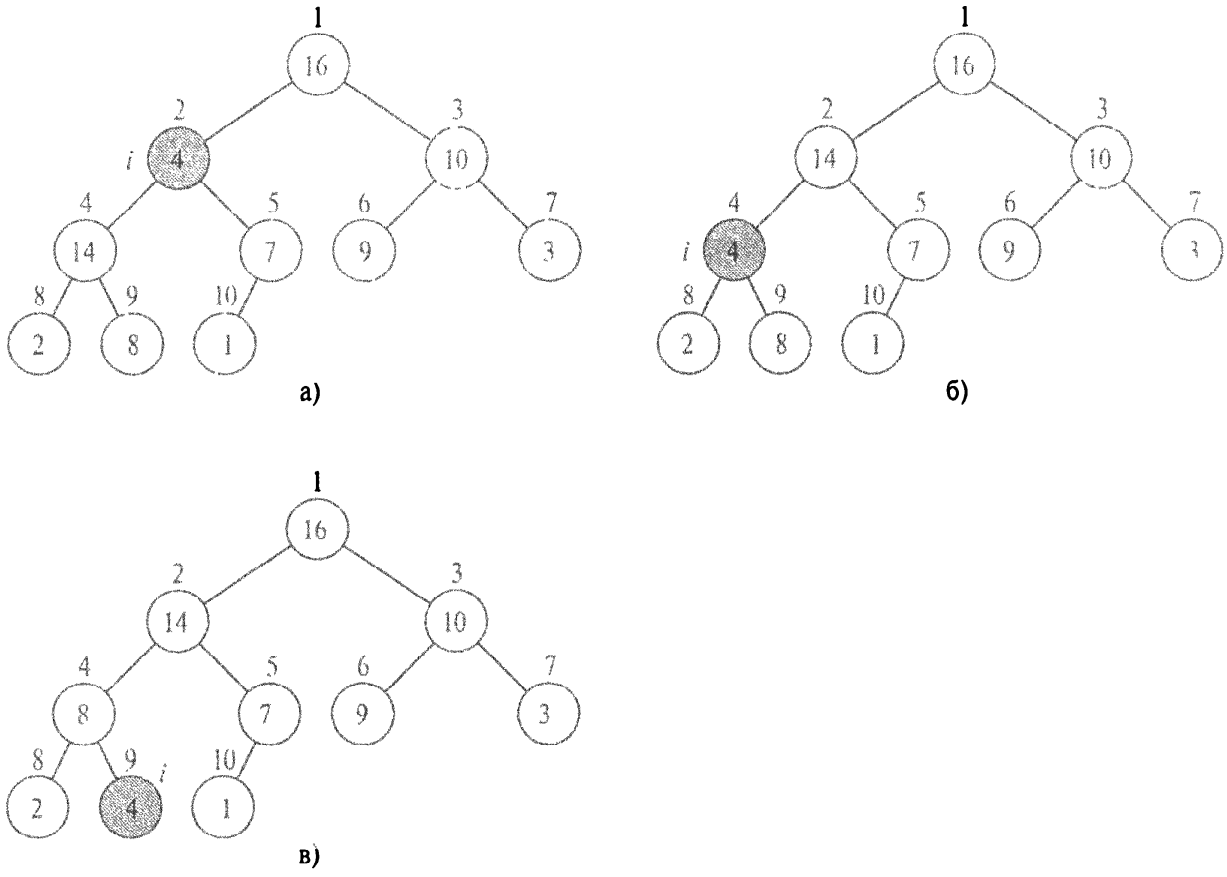

Рис. 6.2. Работа процедуры MAX_HEAPIFY$(A, 2)$ при $heap\_size\,[A] = 10$

$i = 4$. После перестановки элементов $A\,[4]$ и $A\,[9]$ (часть *в* рисунка) ситуация в узле 4 исправляется, а рекурсивный вызов процедуры MAX_HEAPIFY$(A, 9)$ не вносит никаких изменений в рассматриваемую структуру данных.

Время работы процедуры MAX_HEAPIFY на поддереве размера n с корнем в заданном узле i вычисляется как время $\Theta\,(1)$, необходимое для исправления отношений между элементами $A\,[i]$, $A\,[Left\,(i)]$ или $A\,[Right\,(i)]$, плюс время работы этой процедуры с поддеревом, корень которого находится в одном из дочерних узлов узла i. Размер каждого из таких дочерних поддеревьев не превышает величину $2n/3$, причем наихудший случай — это когда последний уровень заполнен наполовину. Таким образом, время работы процедуры MAX_HEAPIFY описывается следующим рекуррентным соотношением:

$$T\,(n) \leqslant T\,(2n/3) + \Theta\,(1)\,.$$

Решение этого рекуррентного соотношения, в соответствии со вторым случаем основной теоремы (теоремы 4.1), равно $T\,(n) = O\,(\lg n)$. По-другому время работы процедуры MAX_HEAPIFY с узлом, который находится на высоте h, можно выразить как $O\,(h)$.

Упражнения

6.2-1. Используя в качестве модели рис. 6.2, проиллюстрируйте работу процедуры MAX_HEAPIFY$(A, 3)$ с массивом $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

6.2-2. Используя в качестве отправной точки процедуру MAX_HEAPIFY, составьте псевдокод процедуры MIN_HEAPIFY(A, i), выполняющей соответствующие действия в неубывающей пирамиде. Сравните время работы этих двух процедур.

6.2-3. Как работает процедура MAX_HEAPIFY(A, i) в случае, когда элемент $A[i]$ больше своих дочерних элементов?

6.2-4. К чему приведет вызов процедуры MAX_HEAPIFY(A, i) при $i > > heap\_size[A]/2$?

6.2-5. Код процедуры MAX_HEAPIFY достаточно рационален, если не считать рекурсивного вызова в строке 10, из-за которого некоторые компиляторы могут сгенерировать неэффективный код. Напишите эффективную процедуру MAX_HEAPIFY, в которой вместо рекурсивного вызова использовалась бы итеративная управляющая конструкция (цикл).

6.2-6. Покажите, что в наихудшем случае время работы процедуры MAX_HEAPIFY на пирамиде размера n равно $\Omega(\lg n)$. (*Указание*: в пирамиде с n узлами присвойте узлам такие значения, чтобы процедура MAX_HEAPIFY рекурсивно вызывалась в каждом узле, расположенном на пути от корня до одного из листьев.)

6.3 Создание пирамиды

С помощью процедуры MAX_HEAPIFY можно преобразовать массив $A[1..n]$, где $n = length[A]$, в невозрастающую пирамиду в направлении снизу вверх. Из упражнения 6.1-7 известно, что все элементы подмассива $A[(\lfloor n/2 \rfloor + 1)..n]$ являются листьями дерева, поэтому каждый из них можно считать одноэлементной пирамидой, с которой можно начать процесс построения. Процедура BUILD_MAX_HEAP проходит по остальным узлам и для каждого из них выполняет процедуру MAX_HEAPIFY:

```
BUILD_MAX_HEAP(A)
1  heap_size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3      do MAX_HEAPIFY(A, i)
```

Пример работы процедуры BUILD_MAX_HEAP показан на рис. 6.3. В части *a* этого рисунка изображен 10-элементный входной массив A и представляющее

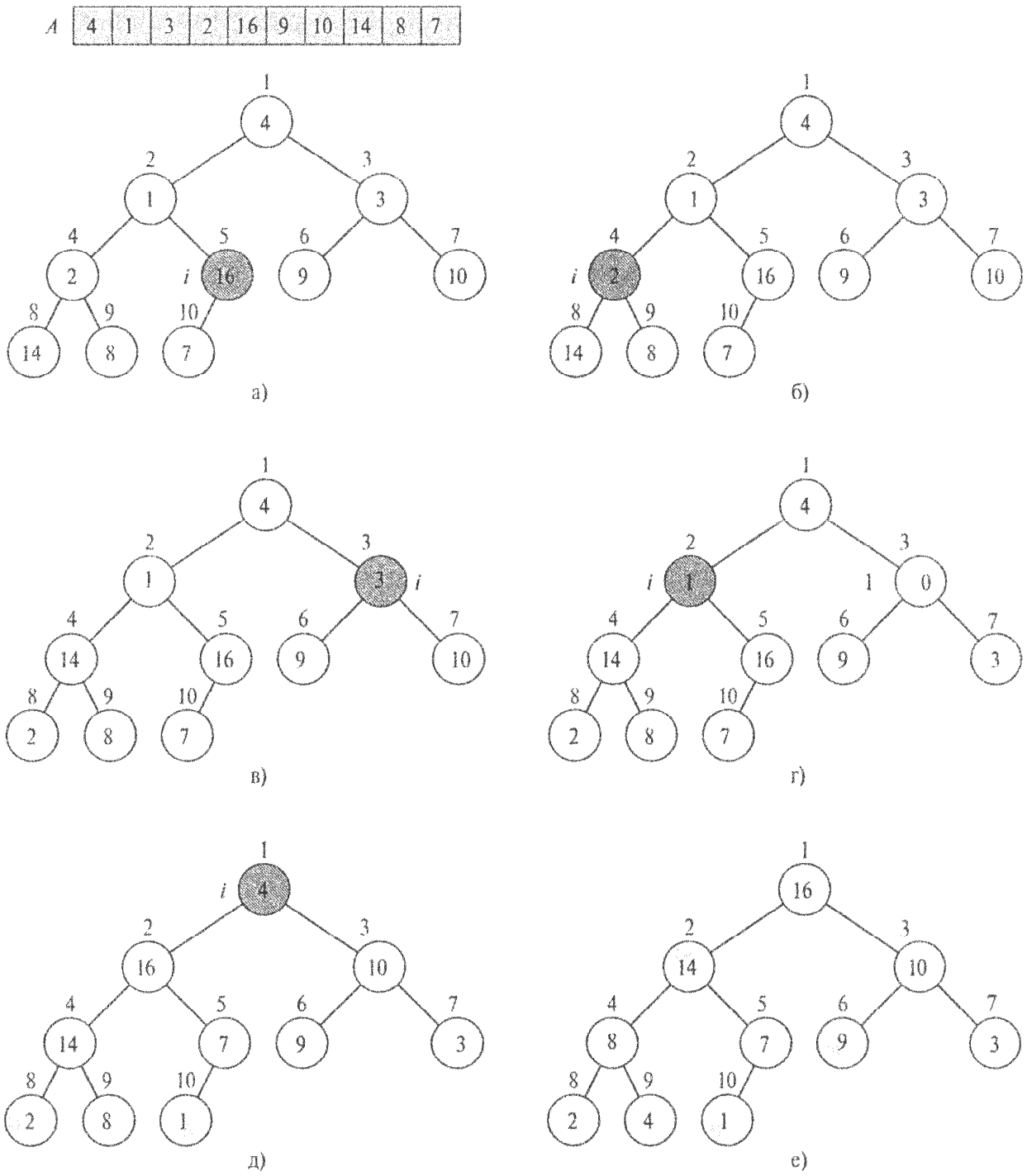

Рис. 6.3. Схема работы процедуры BUILD_MAX_HEAP

его бинарное дерево. Из этого рисунка видно, что перед вызовом процедуры MAX_HEAPIFY(A, i) индекс цикла i указывает на 5-й узел. Получившаяся в результате структура данных показана в части *б*. В следующей итерации индекс цикла i указывает на узел 4. Последующие итерации цикла **for** в процедуре BUILD_MAX_HEAP показаны в частях *в–д* рисунка. Обратите внимание, что при вызове процедуры MAX_HEAPIFY для любого узла поддеревья с корнями в его дочерних узлах являются невозрастающими пирамидами. В части *е* показана невозрастающая пирамида, полученная в результате работы процедуры BUILD_MAX_HEAP.

Чтобы показать, что процедура BUILD_MAX_HEAP работает корректно, воспользуемся сформулированным ниже инвариантом цикла.

> Перед каждой итерацией цикла **for** в строках 2–3 процедуры BUILD_MAX_HEAP все узлы с индексами $i+1, i+2, \ldots, n$ являются корнями невозрастающих пирамид.

Необходимо показать, что этот инвариант справедлив перед первой итерацией цикла, что он сохраняется при каждой итерации, и что он позволяет продемонстрировать корректность алгоритма после его завершения.

Инициализация. Перед первой итерацией цикла $i = \lfloor n/2 \rfloor$. Все узлы с индексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ — листья, поэтому каждый из них является корнем тривиальной невозрастающей пирамиды.

Сохранение. Чтобы убедиться, что каждая итерация сохраняет инвариант цикла, заметим, что дочерние по отношению к узлу i имеют номера, которые больше i. В соответствии с инвариантом цикла, оба эти узла являются корнями невозрастающих пирамид. Это именно то условие, которое требуется для вызова процедуры MAX_HEAPIFY(A, i), чтобы преобразовать узел с индексом i в корень невозрастающей пирамиды. Кроме того, при вызове процедуры MAX_HEAPIFY сохраняется свойство пирамиды, заключающееся в том, что все узлы с индексами $i+1, i+2, \ldots, n$ являются корнями невозрастающих пирамид. Уменьшение индекса i в цикле **for** обеспечивает выполнение инварианта цикла для следующей итерации.

Завершение. После завершения цикла $i = 0$. В соответствии с инвариантом цикла, все узлы с индексами $1, 2, \ldots, n$ являются корнями невозрастающих пирамид. В частности, таким корнем является узел 1.

Простую верхнюю оценку времени работы процедуры BUILD_MAX_HEAP можно получить следующим простым способом. Каждый вызов процедуры MAX_HEAPIFY занимает время $O(\lg n)$, и всего имеется $O(n)$ таких вызовов. Таким образом, время работы алгоритма равно $O(n \lg n)$. Эта верхняя граница вполне корректна, однако не является асимптотически точной.

Чтобы получить более точную оценку, заметим, что время работы MAX_HEAPIFY в том или ином узле зависит от высоты этого узла, и при этом большинство узлов расположено на малой высоте. При более тщательном анализе принимается во внимание тот факт, что высота n-элементной пирамиды равна $\lfloor \lg n \rfloor$ (упражнение 6.1-2) и что на любом уровне, находящемся на высоте h, содержится не более $\lceil n/2^{h+1} \rceil$ узлов (упражнение 6.3-3).

Время работы процедуры MAX_HEAPIFY при ее вызове для работы с узлом, который находится на высоте h, равно $O(h)$, поэтому верхнюю оценку полного времени работы процедуры BUILD_MAX_HEAP можно записать следующим

образом:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Сумму в последнем выражении можно оценить, подставив $x = 1/2$ в формулу (A.8), в результате чего получим:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Таким образом, время работы процедуры BUILD_MAX_HEAP можно ограничить следующим образом:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

Получается, что время, которое требуется для преобразования неупорядоченного массива в невозрастающую пирамиду, линейно зависит от размера входных данных.

Неубывающую пирамиду можно создать с помощью процедуры BUILD_MIN_HEAP, полученной в результате преобразования процедуры BUILD_MAX_HEAP путем замены в строке 3 вызова функции MAX_HEAPIFY вызовом функции MIN_HEAPIFY (см. упражнение 6.2-2). Процедура BUILD_MIN_HEAP создает неубывающую пирамиду из неупорядоченного линейного массива за время, линейно зависящее от размера входных данных.

Упражнения

6.3-1. Используя в качестве модели рис. 6.3, проиллюстрируйте работу процедуры BUILD_MAX_HEAP с входным массивом $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2. Почему индекс цикла i в строке 2 процедуры BUILD_MAX_HEAP убывает от $\lfloor length[A]/2 \rfloor$ до 1, а не возрастает от 1 до $\lfloor length[A]/2 \rfloor$?

6.3-3. Покажите, что в любой n-элементной пирамиде на высоте h находится не более $\lceil n/2^{h+1} \rceil$ узлов.

6.4 Алгоритм пирамидальной сортировки

Работа алгоритма пирамидальной сортировки начинается с вызова процедуры BUILD_MAX_HEAP, с помощью которой из входного массива $A[1..n]$, где

$n = length[A]$, создается невозрастающая пирамида. Поскольку наибольший элемент массива находится в корне, т.е. в элементе $A[1]$, его можно поместить в окончательную позицию в отсортированном массиве, поменяв его местами с элементом $A[n]$. Выбросив из пирамиды узел n (путем уменьшения на единицу величины $heap\_size[A]$), мы обнаружим, что подмассив $A[1..(n-1)]$ легко преобразуется в невозрастающую пирамиду. Пирамиды, дочерние по отношению к корневому узлу, после обмена элементов $A[1]$ и $A[n]$ и уменьшения размера массива остаются невозрастающими, однако новый корневой элемент может нарушить свойство невозрастания пирамиды. Для восстановления этого свойства достаточно вызвать процедуру MAX_HEAPIFY$(A, 1)$, после чего подмассив $A[1..(n-1)]$ превратится в невозрастающую пирамиду. Затем алгоритм пирамидальной сортировки повторяет описанный процесс для невозрастающих пирамид размера $n-1, n-2, \ldots, 2$. (См. упражнение 6.4-2, посвященное точной формулировке инварианта цикла данного алгоритма.)

```
HEAPSORT(A)
1  BUILD_MAX_HEAP(A)
2  for i ← length[A] downto 2
3      do Обменять A[1] ↔ A[i]
4         heap_size[A] ← heap_size[A] − 1
5         MAX_HEAPIFY(A, 1)
```

На рис. 6.4 показан пример пирамидальной сортировки после предварительного построения невозрастающей пирамиды. В каждой части этого рисунка изображена невозрастающая пирамида перед выполнением очередной итерации цикла **for** в строках 2–5. В части *а*) этого рисунка показана исходная невозрастающая пирамида, полученная при помощи процедуры BUILD_MAX_HEAP. В частях *б*)–*к*) показаны пирамиды, получающиеся в результате вызова процедуры MAX_HEAPIFY в строке 5. В каждой из этих частей указано значение индекса i. В пирамиде содержатся только узлы, закрашенные светло-серым цветом. В части *л*) показан получившийся в конечном итоге массив A.

Время работы процедуры HEAPSORT равно $O(n \lg n)$, поскольку вызов процедуры BUILD_MAX_HEAP требует времени $O(n)$, а каждый из $n-1$ вызовов процедуры MAX_HEAPIFY — времени $O(\lg n)$.

Упражнения

6.4-1. Используя в качестве модели рис. 6.4, проиллюстрируйте работу процедуры HEAPSORT с входным массивом $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2. Докажите корректность процедуры HEAPSORT с помощью сформулированного ниже инварианта цикла.

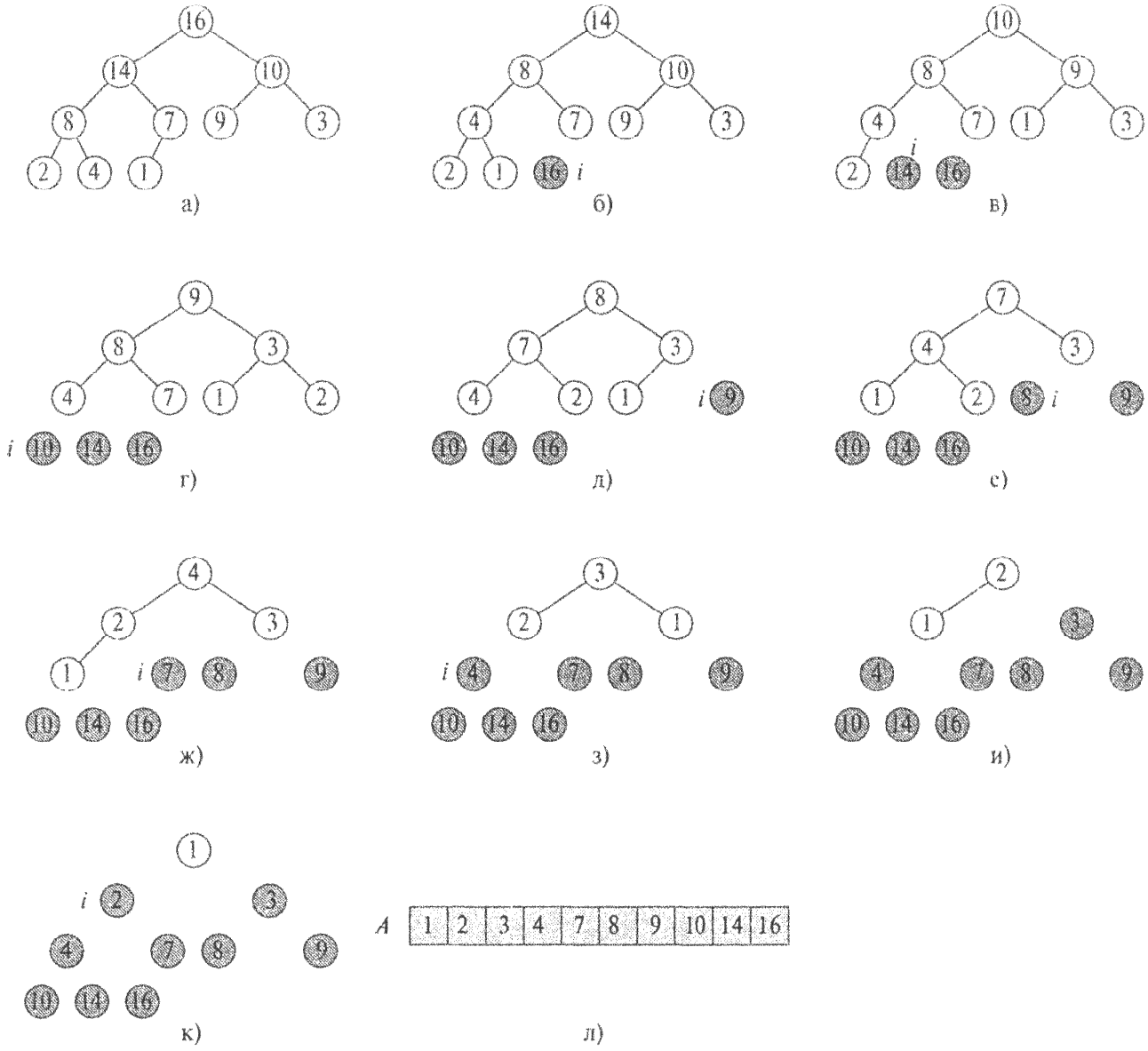

Рис. 6.4. Работа процедуры HEAPSORT

В начале каждой итерации цикла **for** в строках 2–5 подмассив $A[1..i]$ является невозрастающей пирамидой, содержащей i наименьших элементов массива $A[1..n]$, а в подмассиве $A[i+1..n]$ находятся $n-i$ отсортированных наибольших элементов массива $A[1..n]$.

6.4-3. Чему равно время работы алгоритма пирамидальной сортировки массива A длины n, в котором элементы отсортированы и расположены в порядке возрастания? В порядке убывания?

6.4-4. Покажите, что время работы алгоритма пирамидальной сортировки в наихудшем случае равно $\Omega(n \lg n)$.

★ 6.4-5. Покажите, что для массива, все элементы которого различны, время работы пирамидальной сортировки в наилучшем случае равно $\Omega(n \lg n)$.

6.5 Очереди с приоритетами

Пирамидальная сортировка — превосходный алгоритм, однако качественная реализация алгоритма быстрой сортировки, представленного в главе 7, на практике обычно превосходит по производительности пирамидальную сортировку. Тем не менее, структура данных, использующаяся при пирамидальной сортировке, сама по себе имеет большую ценность. В этом разделе представлено одно из наиболее популярных применений пирамид — в качестве эффективных очередей с приоритетами. Как и пирамиды, очереди с приоритетами бывают двух видов: невозрастающие и неубывающие. Мы рассмотрим процесс реализации невозрастающих очередей с приоритетами, которые основаны на невозрастающих пирамидах; в упражнении 6.5-3 требуется написать процедуры для неубывающих очередей с приоритетами.

Очередь с приоритетами (priority queue) — это структура данных, предназначенная для обслуживания множества S, с каждым элементом которого связано определенное значение, называющееся ***ключом*** (key). В ***невозрастающей очереди с приоритетами*** поддерживаются следующие операции.

- Операция INSERT(S, x) вставляет элемент x в множество S. Эту операцию можно записать как $S \leftarrow S \cup \{x\}$.
- Операция MAXIMUM(S) возвращает элемент множества S с наибольшим ключом.
- Операция EXTRACT_MAX(S) возвращает элемент с наибольшим ключом, удаляя его при этом из множества S.
- Операция INCREASE_KEY(S, x, k) увеличивает значение ключа, соответствующего элементу x, путем его замены ключом со значением k. Предполагается, что величина k не меньше текущего ключа элемента x.

Одна из областей применения невозрастающих очередей — планирование заданий на компьютере, который совместно используется несколькими пользователями. Очередь позволяет следить за заданиями, которые подлежат выполнению, и за их приоритетами. Если задание прервано или завершило свою работу, из очереди с помощью операции EXTRACT_MAX выбирается следующее задание с наибольшим приоритетом. В очередь в любое время можно добавить новое задание, воспользовавшись операцией HEAP_INSERT.

В ***неубывающей очереди с приоритетами*** поддерживаются операции INSERT, MINIMUM, EXTRACT_MIN и DECREASE_KEY. Очереди такого вида могут использоваться в моделировании систем, управляемых событиями. В роли элементов очереди в таком случае выступают моделируемые события, для каждого из которых сопоставляется время происшествия, играющее роль ключа. Элементы должны моделироваться последовательно согласно времени событий, поскольку

процесс моделирования может вызвать генерацию других событий. Моделирующая программа выбирает очередное моделируемое событие с помощью операции EXTRACT_MIN. Когда инициируются новые события, они помещаются в очередь с помощью процедуры INSERT. В главах 23 и 24 нам предстоит познакомиться и с другими случаями применения неубывающих очередей с приоритетами, когда особо важной становится роль операции DECREASE_KEY.

Не удивительно, что приоритетную очередь можно реализовать с помощью пирамиды. В каждом отдельно взятом приложении, например, в планировщике заданий, или при моделировании событий элементы очереди с приоритетами соответствуют объектам, с которыми работает это приложение. Часто возникает необходимость определить, какой из объектов приложения отвечает тому или иному элементу очереди, или наоборот. Если очередь с приоритетами реализуется с помощью пирамиды, то в каждом элементе пирамиды приходится хранить ***идентификатор*** (handle) соответствующего объекта приложения. То, каким будет конкретный вид этого идентификатора (указатель, целочисленный индекс или что-нибудь другое), — зависит от приложения. В каждом объекте приложения точно так же необходимо хранить идентификатор соответствующего элемента пирамиды. В данной книге таким идентификатором, как правило, будет индекс массива. Поскольку в ходе операций над пирамидой ее элементы изменяют свое расположение в массиве, при перемещении элемента пирамиды необходимо также обновлять значение индекса в соответствующем объекте приложения. Так как детали доступа к объектам приложения в значительной мере зависят от самого приложения и его реализации, мы не станем останавливаться на этом вопросе. Ограничимся лишь замечанием, что на практике необходима организация надлежащей обработки идентификаторов.

Теперь перейдем к реализации операций в невозрастающей очереди с приоритетами. Процедура HEAP_MAXIMUM реализует выполнение операции MAXIMUM за время $\Theta(1)$:

```
HEAP_MAXIMUM(A)
1  return A[1]
```

Процедура HEAP_EXTRACT_MAX реализует операцию EXTRACT_MAX. Она напоминает тело цикла **for** в строках 3–5 процедуры HEAPSORT:

```
HEAP_EXTRACT_MAX(A)
1  if heap_size[A] < 1
2     then error "Очередь пуста"
3  max ← A[1]
4  A[1] ← A[heap_size[A]]
5  heap_size[A] ← heap_size[A] − 1
6  MAX_HEAPIFY(A, 1)
7  return max
```

Время работы процедуры HEAP_EXTRACT_MAX равно $O(\lg n)$, поскольку перед запуском процедуры MAX_HEAPIFY, выполняющейся в течение времени $O(\lg n)$, в ней выполняется лишь определенная постоянная подготовительная работа.

Процедура HEAP_INCREASE_KEY реализует операцию INCREASE_KEY. Элемент очереди с приоритетами, ключ которого подлежит увеличению, идентифицируется в массиве с помощью индекса i. Сначала процедура обновляет ключ элемента $A[i]$. Поскольку это может нарушить свойство невозрастающих пирамид, после этого процедура проходит путь от измененного узла к корню в поисках надлежащего места для нового ключа. Эта операция напоминает реализованную в цикле процедуры INSERTION_SORT (строки 5–7) из раздела 2.1. В процессе прохода выполняется сравнение текущего элемента с родительским. Если оказывается, что ключ текущего элемента превышает значение ключа родительского элемента, то происходит обмен ключами элементов и процедура продолжает свою работу на более высоком уровне. В противном случае процедура прекращает работу, поскольку ей удалось восстановить свойство невозрастающих пирамид. (Точная формулировка инварианта цикла этого алгоритма приведена в упражнении 6.5-5.)

```
HEAP_INCREASE_KEY(A, i, key)
1  if key < A[i]
2     then error "Новый ключ меньше текущего"
3  A[i] ← key
4  while i > 1 и A[PARENT(i)] < A[i]
5        do Обменять A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

На рис. 6.5 приведен пример работы процедуры HEAP_INCREASE_KEY. В части *а* этого рисунка изображена невозрастающая пирамида, в которой будет увеличен ключ узла, выделенного темно-серым цветом (кроме выделения цветом, возле этого узла указан индекс i). В части *б* рисунка показана эта же пирамида после того, как ключ выделенного узла был увеличен до 15. В части *в* обрабатываемая пирамида изображена после первой итерации цикла **while** (строки 4–6). Здесь видно, как текущий и родительский по отношению к нему узлы обменялись ключами, и индекс i перешел к родительскому узлу. В части *г* показана эта же пирамида после еще одной итерации цикла. Теперь условие невозрастающих пирамид выполняется, и процедура завершает работу. Время обработки n-элементной пирамиды с помощью этой процедуры равно $O(\lg n)$. Это объясняется тем, что длина пути от обновляемого в строке 3 элемента до корня равна $O(\lg n)$.

Процедура MAX_HEAP_INSERT реализует операцию INSERT. В качестве параметра этой процедуре передается ключ нового элемента. Сначала процедура добавляет в пирамиду новый лист и присваивает ему ключ со значением $-\infty$. Затем вызывается процедура HEAP_INCREASE_KEY, которая присваивает корректное зна-

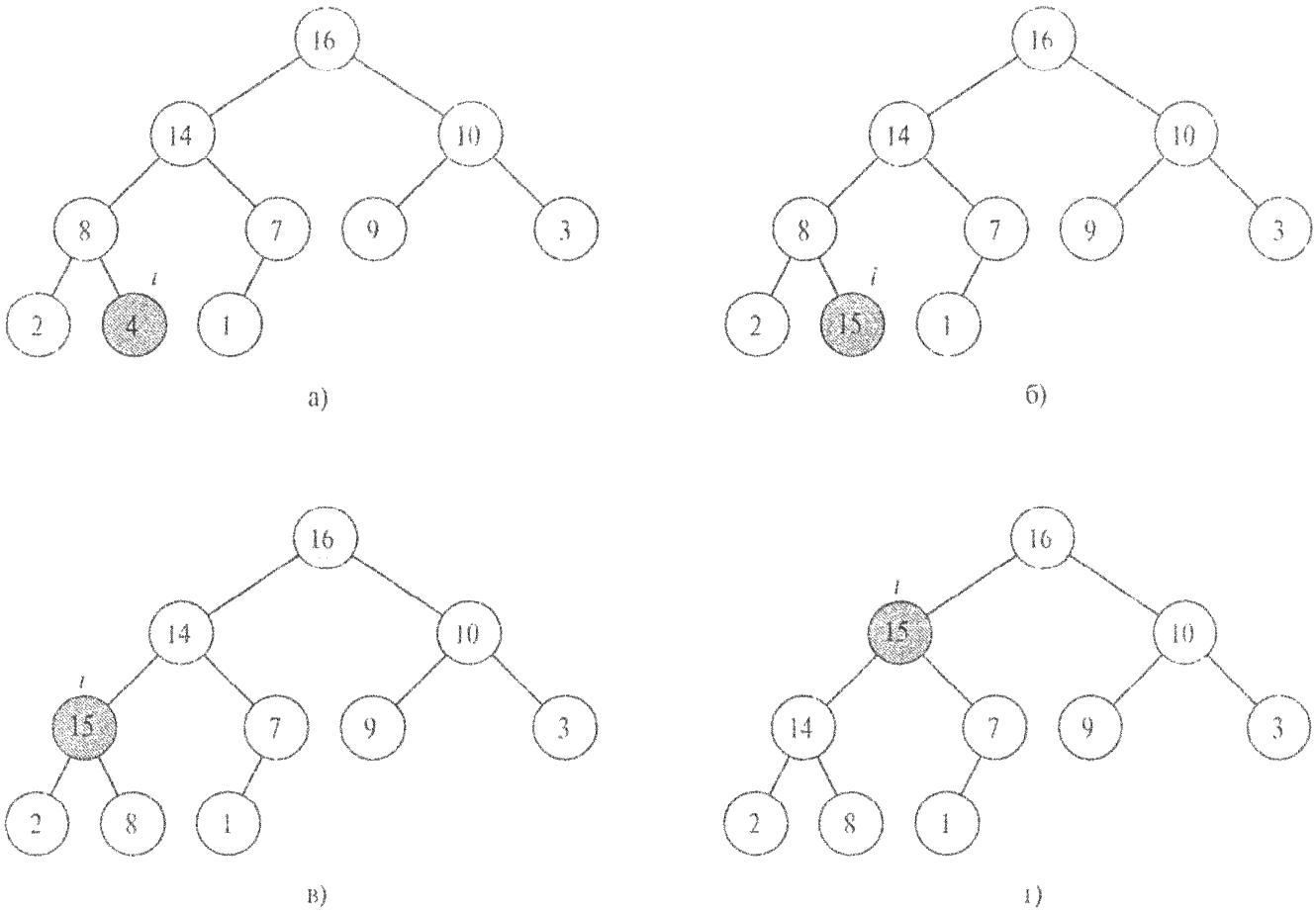

Рис. 6.5. Работа процедуры HEAP_INCREASE_KEY

чение ключу и помещает его в надлежащее место, чтобы не нарушалось свойство невозрастающих пирамид:

MAX_HEAP_INSERT(A, key)
1 $heap\_size[A] \leftarrow heap\_size[A] + 1$
2 $A[heap\_size[A]] \leftarrow -\infty$
3 HEAP_INCREASE_KEY($A, heap\_size[A], key$)

Время вставки в n-элементную пирамиду с помощью процедуры MAX_HEAP_INSERT составляет $O(\lg n)$.

Подводя итог, заметим, что в пирамиде время выполнения всех операций по обслуживанию очереди с приоритетами равно $O(\lg n)$.

Упражнения

6.5-1. Проиллюстрируйте работу процедуры HEAP_EXTRACT_MAX с пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2. Проиллюстрируйте работу процедуры MAX_HEAP_INSERT($A, 10$) с пирамидой $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Воспользуйтесь в качестве модели рисунком 6.5.