Пирамиды. Общие сведения

Пирамидой называется двоичное дерево, обладающее следующими характеристиками:

- Полнота. Все уровни дерева содержат все возможные узлы (хотя последний уровень может быть заполнен лишь частично). На рис. 12.1 приведены примеры полных и неполных деревьев.
- Пирамида (обычно) реализуется на базе массива. Реализация двоичных деревьев в виде массивов (в отличие от хранения ссылок, определяющих связи между узлами) была описана в главе 8, «Двоичные деревья».
- Для каждого узла в пирамиде выполняется основное условие, гласящее, что ключ каждого узла больше (либо равен) ключей его потомков.

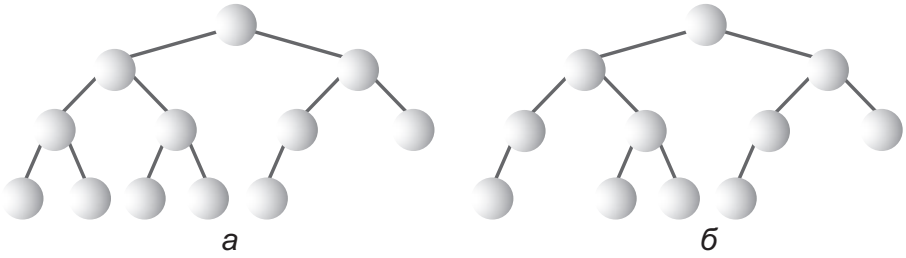

Рис. 12.1. Полные и неполные двоичные деревья: *а* — полное дерево; *б* — неполное дерево

На рис. 12.2 показана пирамида и ее связь с массивом, на базе которого она реализована. В памяти хранится массив, а пирамида является лишь концептуальным представлением. Обратите внимание: дерево является полным, а условие пирамиды выполняется для всех узлов.

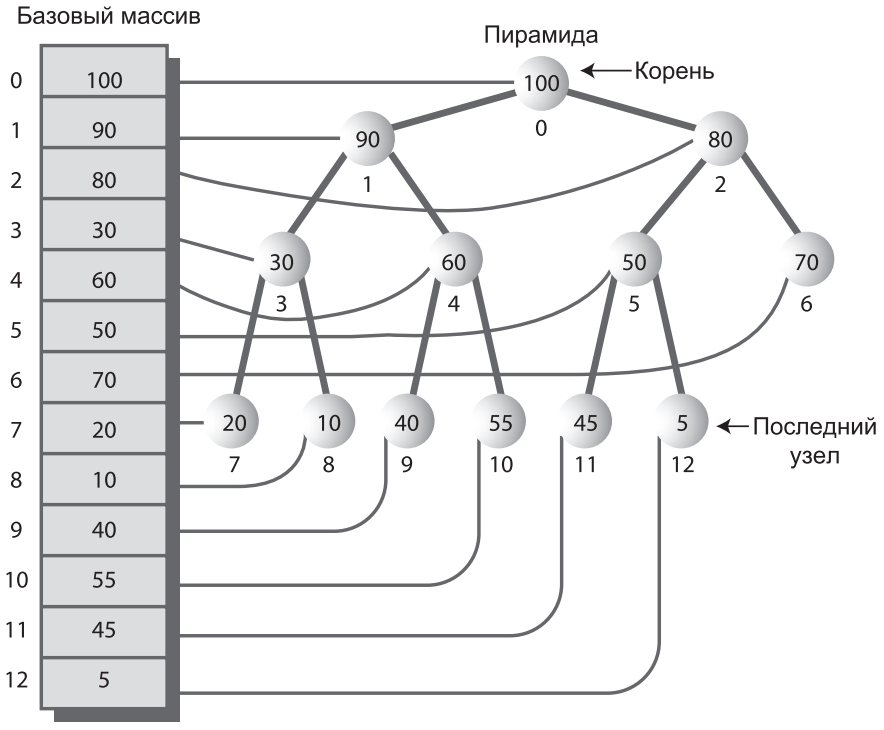

Рис. 12.2. Пирамида и базовый массив

Из того факта, что пирамида является полным двоичным деревом, следует, что в массиве, который используется для ее представления, нет «дыр». Заполнены все ячейки, от 0 до $N - 1$ (на рис. 12.2 $N = 13$).

В этой главе предполагается, что корневой узел содержит наибольший ключ (вместо наименьшего). Приоритетная очередь, созданная на базе такой пирамиды, упорядочена по убыванию приоритетов. (Очереди, упорядоченные по возрастанию приоритетов, рассматривались в главе 4.)

Приоритетные очереди, пирамиды и ADT

Эта глава посвящена пирамидам, хотя пирамиды в основном рассматриваются в контексте реализации приоритетных очередей. Впрочем, между приоритетной очередью и пирамидой, используемой для ее реализации, существует очень тесная связь. Эта связь продемонстрирована в следующем сокращенном фрагменте кода:

```
class Heap
   {
   private Node heapArray[];

   public void insert(Node nd)
      {  }
   public Node remove()
      {  }
   }
class priorityQueue
   {
   private Heap theHeap;

   public void insert(Node nd)
      { theHeap.insert(nd); }
   public Node remove()
      { return theHeap.remove() }
   }
```

Методы класса `priorityQueue` представляют собой простые «обертки» для методов нижележащего класса `Heap`; они обладают той же функциональностью. Пример ясно показывает, что приоритетная очередь представляет собой абстрактный тип данных, который может быть реализован разными способами, тогда как пирамида относится к числу более фундаментальных структур данных. В этой главе для простоты методы пирамиды приводятся в исходном виде, без «упаковки» в методы приоритетной очереди.

Слабая упорядоченность

По сравнению с деревом двоичного поиска, в котором ключ левого потомка каждого узла меньше ключа правого потомка, пирамида является *слабо упорядоченной* (квазиупорядоченной). В частности, ограничения дерева двоичного поиска позволяют выполнить перебор узлов в порядке сортировки по простому алгоритму.

В пирамиде упорядоченный перебор узлов затрудняется тем, что принцип организации пирамиды (условие пирамиды) не так силен, как принцип организации дерева. Единственное, что можно гарантировать по поводу пирамиды — то, что на каждом пути от корня к листу узлы упорядочены по убыванию. Как видно из рис. 12.2, ключи узлов, находящихся слева или справа от заданного узла, на более высоких или низких уровнях (и не принадлежащих тому же пути), могут быть больше или меньше ключа узла. Пути, не имеющие общих узлов, полностью независимы друг от друга.

Вследствие слабой упорядоченности пирамиды некоторые операции с ней затруднены или невозможны. Из-за отсутствия нормальной возможности перебора пирамида не предоставляет удобных средств поиска заданного ключа. Дело в том, что алгоритм поиска не располагает достаточной информацией для принятия решения о том, какого из двух потомков узла следует выбрать для перехода на нижней уровень. Соответственно узел с заданным ключом невозможно удалить (по крайней мере за время $O(\log N)$), потому что его нельзя найти. (Операции можно выполнить последовательным просмотром всех ячеек последовательности, но это может быть сделано только за время $O(N)$.)

Таким образом, структура пирамиды выглядит довольно хаотично. Тем не менее упорядоченности пирамиды достаточно для быстрого удаления наибольшего узла и быстрой вставки новых узлов. Этих операций достаточно для использования пирамиды в качестве приоритетной очереди. Сначала мы в общих чертах разберем, как выполняются эти операции, а затем рассмотрим практические примеры в приложении Workshop.

Удаление

Под удалением подразумевается удаление с наибольшим ключом. Этот узел всегда является корневым, поэтому его удаление выполняется просто. Корень всегда хранится в ячейке 0 базового массива:

```
maxNode = heapArray[0];
```

Проблема в том, что после удаления корня дерево перестает быть полным; в нем появляется пустая ячейка. «Дыру» необходимо заполнить. Все элементы массива можно было бы сдвинуть на одну ячейку, но существует другое, более быстрое решение. Последовательность действий при удалении наибольшего узла выглядит так:

1. Удалить корневой узел.
2. Переместить последний узел на место корневого.
3. Смещать его вниз до тех пор, пока он не окажется ниже большего и выше меньшего узла.

Последним узлом является крайний правый узел самого нижнего занятого уровня дерева. Он соответствует последней заполненной ячейке массива. (На рис. 12.2 это узел с индексом 12 и значением 5.) Копирование этого узла в корень выполняется тривиально:

```
heapArray[0] = heapArray[N-1];
N--;
```

Удаление корня приводит к уменьшению размера массива на 1.

В ходе смещения узел последовательно меняется местами с узлом, находящимся перед ним; после каждой перестановки алгоритм проверяет, оказался ли узел в правильной позиции. На шаге 3 новый корневой узел слишком мал для своей позиции, поэтому он смещается вниз на свое законное место. Код выполнения этой операции будет приведен ниже.

Шаг 2 восстанавливает полноту пирамиды (отсутствие пустых ячеек), а шаг 3 восстанавливает условие пирамиды (каждый узел больше своих потомков). Процесс удаления показан на рис. 12.3.

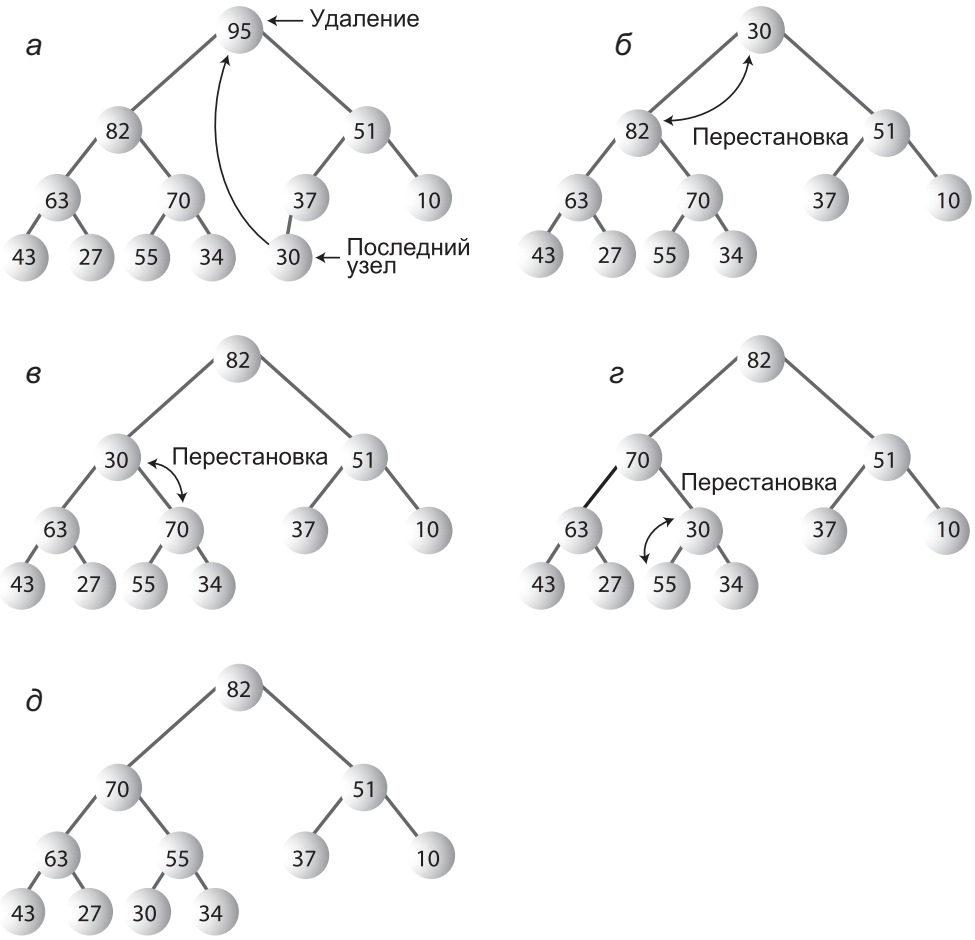

Рис. 12.3. Удаление наибольшего узла

На рис. 12.3, *а* последний узел (30) копируется на место корневого. На рис. 12.3, *б*, *в*, *г* последний узел смещается до своей позиции, которая находится на последнем уровне. (Так бывает не всегда; процесс смещения может остановиться и на одном из промежуточных уровней.) На рис. 12.3, *д* узел занимает правильную позицию.

В каждой промежуточной позиции алгоритм смещения проверяет, какой из потомков больше, и меняет местами целевой узел с бóльшим потомком. Если бы целевой узел поменялся местами с меньшим потомком, то этот потомок стал бы родителем большего потомка, а это было бы нарушением условия пирамиды. Правильные и неправильные перестановки изображены на рис. 12.4.

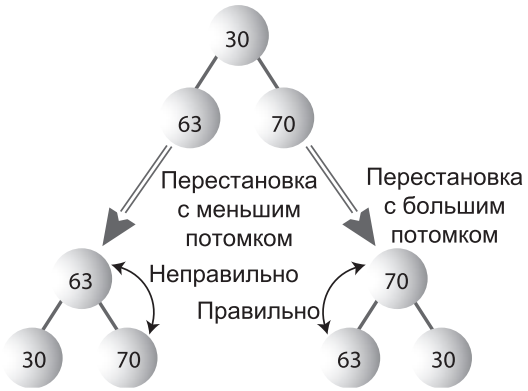

Рис. 12.4. Выбор потомка

Вставка

Вставка узла тоже выполняется относительно просто. При вставке используется смещение вверх (вместо смещения вниз). Сначала вставляемый узел помещается в первую свободную позицию в конце массива, в результате чего размер массива увеличивается на единицу:

```
heapArray[N] = newNode;
N++;
```

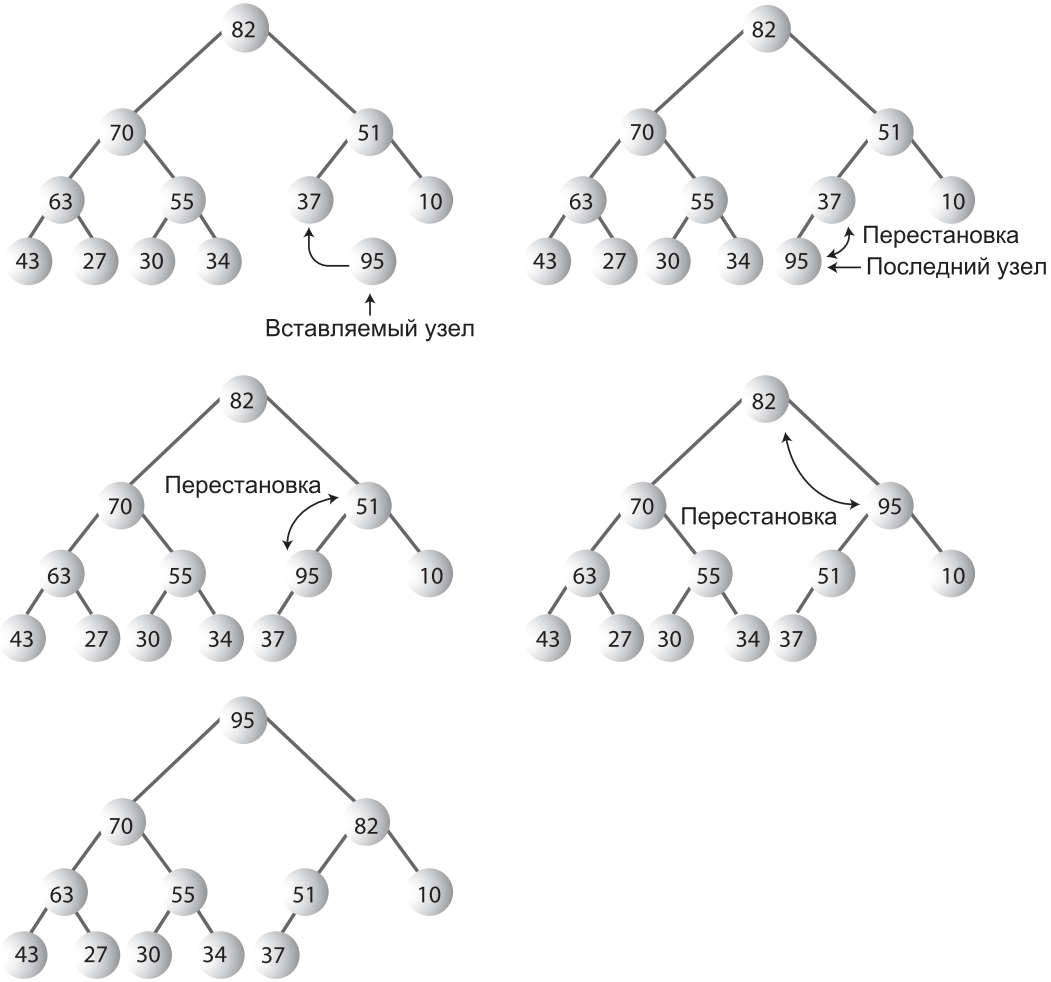

Рис. 12.5. Вставка узла

Такое размещение нарушит условие пирамиды, если ключ нового узла окажется больше ключа его родителя. Так как родитель находится в нижней части пирамиды, его ключ относительно мал, поэтому ключ нового узла с большой вероятностью превысит его. Таким образом, новый узел обычно приходится смещать вверх, пока он не окажется ниже узла с бóльшим ключом, но выше узла с меньшим ключом. Процесс вставки продемонстрирован на рис. 12.5.

Алгоритм смещения вверх проще алгоритма смещения вниз, потому что он не требует сравнения двух потомков. Узел имеет только одного родителя, так что вставляемый узел просто меняется местами с родителем. На рисунке правильной позицией нового узла оказывается корень дерева, но смещение также может остановиться на одном из промежуточных уровней.

Сравнивая рис. 12.4 и 12.5, легко убедиться, что удаление с последующей вставкой того же узла не обязательно приводит к восстановлению исходной пирамиды. Заданный набор узлов может иметь много действительных представлений в виде пирамиды в зависимости от порядка вставки узлов.

Условные перестановки

На рис. 12.4 и 12.5 показано, как узлы меняются местами в процессе смещения вверх и вниз. На концептуальном уровне такие перестановки упрощают понимание вставок и удалений, и в некоторых реализациях узлы действительно меняются местами. На рис. 12.6, *а* показана упрощенная версия перестановок в процессе смещения вниз. После трех перестановок узел A оказывается в позиции D, а каждый из узлов B, C и D поднимается на один уровень.

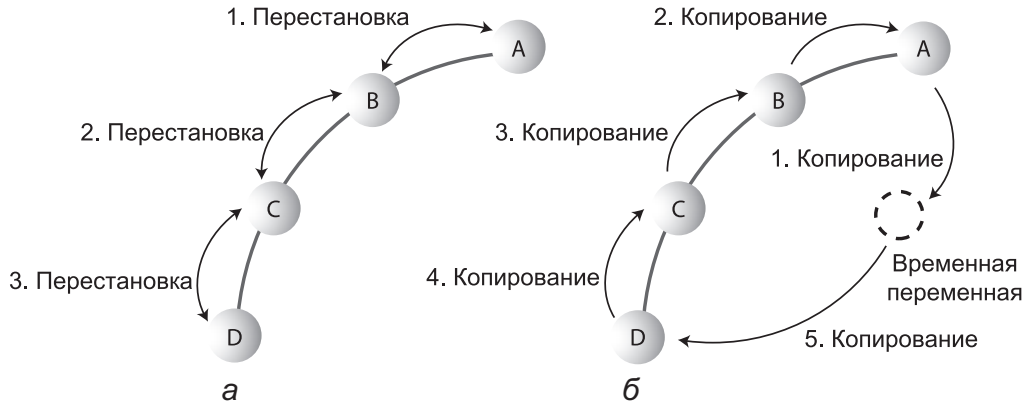

Рис. 12.6. Смещение с перестановками и копированием: *а* — перестановки; *б* — копирование

Однако перестановка требует трех операций копирования; соответственно для трех перестановок на рис. 12.6, *а* потребуется девять копирований. Чтобы сократить общее количество операций копирования при смещении узла, можно заменить перестановки копированиями.

На рис. 12.6, *б* показано, как пять операций копирования выполняют работу трех перестановок. Сначала узел A сохраняется во временной переменной. Затем узел B копируется на место A, узел C — на место B, а узел D — на место C. Наконец,

узел A возвращается из временной переменной на место D. Количество операций копирования сократилось с 9 до 5.

На рисунке узел A перемещается на три уровня. Экономия времени значительно возрастает с увеличением количества уровней, так как две операции копирования во временную переменную и обратно заменяют большее количество операций. Для дерева с многими уровнями сложность сокращается примерно втрое.

Процессы смещения вверх и вниз, выполняемые посредством копирования, также можно представить себе в виде «дыры» (то есть отсутствия узла), движущейся вниз при смещении вверх, и наоборот. Например, на рис. 12.6, *б* копирование A во временную переменную создает «дыру» в A. Вообще говоря, ячейка с «дырой» не совсем пустая — в ней продолжает храниться более ранняя копия узла, но для нас это несущественно. При копировании B в A «дыра» перемещается из A в B в направлении, противоположном смещению узла. Шаг за шагом «дыра» постепенно смещается вниз.

Приложение Heap Workshop

Приложение Heap Workshop демонстрирует, как выполняются операции, описанные в предыдущем разделе: поддерживается вставка новых элементов в пирамиду и удаление наибольшего элемента. Кроме того, вы можете изменить приоритет заданного элемента.

Примерный вид рабочей области приложения Heap Workshop при запуске показан на рис. 12.7.

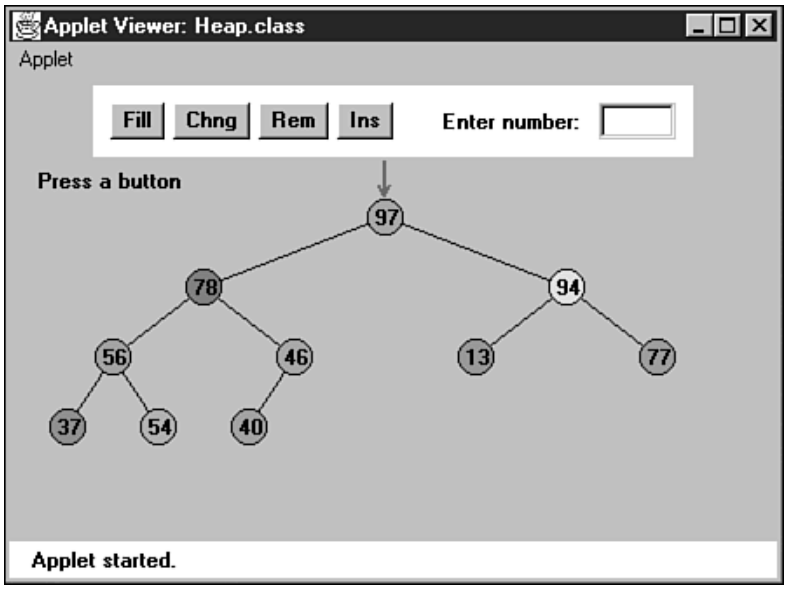

Рис. 12.7. Приложение Heap Workshop

Четыре кнопки — `Fill`, `Chng`, `Rem` и `Ins` — выполняют операции заполнения, изменения приоритета, удаления и вставки соответственно. Давайте посмотрим, как они работают.

Заполнение

При запуске приложения создается пирамида из 10 узлов. При помощи кнопки `Fill` можно создать новую пирамиду с произвольным количеством узлов от 1 до 31. Продолжайте нажимать кнопку `Fill` и введите нужное число по запросу приложения.

Изменение приоритета

Иногда при работе с пирамидой требуется изменить приоритет существующего узла. Такая возможность может пригодиться во многих ситуациях. Скажем, в упомянутом ранее примере с крейсером приближающийся самолет может изменить курс; его приоритет следует понизить в соответствии с изменением обстановки, хотя самолет должен оставаться в очереди до тех пор, пока он не выйдет из зоны действия радара.

Чтобы изменить приоритет узла, нажимайте кнопку `Chng`. При появлении соответствующего запроса щелкните на узле; красная стрелка перемещается к этому узлу. Затем по запросу приложения введите новый приоритет узла.

Если приоритет узла повышается, он смещается вверх к новой позиции. В случае снижения приоритета узел смещается вниз.

Удаление

Кнопка `Rem` удаляет узел с наибольшим ключом, находящийся в корневом узле дерева. Вы увидите, как узел исчезает, а на его месте появляется последний (крайний правый) узел нижнего уровня. Перемещенный узел смещается вниз до тех пор, пока не займет позицию, восстанавливающую иерархию пирамиды.

Вставка

Новый узел всегда изначально вставляется в первую свободную ячейку массива, справа от последнего узла на нижнем уровне пирамиды. Вставленный узел смещается вверх до подходящей позиции. Эта операция выполняется многократным нажатием кнопки `Ins`.

Реализация пирамиды на языке Java

Полный код программы `heap.java` приводится позднее в этом разделе. Но прежде чем рассматривать его, стоит подробнее остановиться на отдельных операциях вставки, удаления и изменения приоритета.

Вспомним несколько фактов из главы 8, касающихся представления дерева в виде массива. Для узла с индексом `x` в массиве:

- индекс родителя равен (`x` – 1)/2;
- индекс левого потомка равен 2*`x` + 1;
- индекс правого потомка равен 2*`x` + 2.

Эти отношения показаны на рис. 12.2.

ПРИМЕЧАНИЕ

Для целых чисел символом / обозначается целочисленное деление с округлением результата в меньшую сторону.

Вставка

Алгоритм смещения вверх выделяется в отдельный метод `trickleUp()`. Код метода `insert()`, включающий вызов `trickleUp()`, получается очень простым:

```
public boolean insert(int key)
   {
   if(currentSize==maxSize)            // Если массив заполнен,
      return false;                    // возвращается признак неудачи
   Node newNode = new Node(key);       // Создание нового узла
   heapArray[currentSize] = newNode;   // Размещение в конце массива
   trickleUp(currentSize++);           // Смещение вверх
   return true;                        // Признак успешной вставки
   }
```

Сначала алгоритм проверяет, что в массиве осталось свободное место, после чего создает новый узел с ключом, переданным в аргументе. Узел вставляется в конец массива. Наконец, вызов метода `trickleUp()` перемещает узел в правильную позицию.

В аргументе метода `trickleUp()` (см. ниже) передается индекс вставленного элемента. Алгоритм находит родителя узла, находящегося в этой позиции, и сохраняет узел в переменной `bottom`. В цикле `while` переменная `index` смещается вверх по пути к корневому узлу, последовательно указывая на каждый узел. Цикл `while` выполняется, пока не был достигнут корневой узел (`index>0`), а ключ (`iData`) родителя `index` меньше ключа нового узла.

В теле цикла `while` выполняется один шаг процесса смещения вверх. Сначала родительский узел копируется в ячейку `index` («дыра» поднимается вверх). Затем `index` перемещается вверх присваиванием индекса родительского узла, а в индекс родительского узла заносится индекс *его* родителя.

```
public void trickleUp(int index)
   {
   int parent = (index-1) / 2;
   Node bottom = heapArray[index];

   while( index > 0 &&
          heapArray[parent].getKey() < bottom.getKey() )
      {
      heapArray[index] = heapArray[parent]; // Узел перемещается вниз
      index = parent;                       // index перемещается вверх
      parent = (parent-1) / 2;              // parent <- его родитель
      }
   heapArray[index] = bottom;
   }
```

При выходе из цикла только что вставленный узел, хранившийся в `bottom`, вставляется в ячейку, на которую ссылается `index`. Это первая ячейка, в которой узел окажется не больше своего родителя, поэтому вставка в нее выполняет условие пирамиды.

Удаление

Алгоритм удаления тоже получается несложным, если выделить алгоритм смещения вниз в отдельную функцию. Мы сохраняем узел из корня, копируем последний узел (с индексом `currentSize-1`) на место корня и вызываем `trickleDown()` для перемещения этого узла в подходящее место.

```
public Node remove()           // Удаление элемента с максимальным ключом
   {                           // (Предполагается, что список не пуст)
   Node root = heapArray[0];   // Сохранение корня
   heapArray[0] = heapArray[--currentSize];  // Корень <- последний узел
   trickleDown(0);             // Корневой узел смещается вниз
   return root;                // Метод возвращает удаленный узел
   }
```

Метод возвращает удаленный узел, который обычно каким-то образом обрабатывается пользователем пирамиды.

Метод смещения вниз `trickleDown()` сложнее метода `trickleUp()`, потому что мы должны определить, какой из двух потомков больше. Сначала узел с индексом `index` сохраняется в переменной с именем `top`. Если метод `trickleDown()` был вызван из `remove()`, то `index` ссылается на корневой узел, но как будет показано ниже, он также может вызываться из других методов.

Цикл `while` выполняется до тех пор, пока `index` не окажется на нижнем уровне дерева, то есть пока у узла имеется хотя бы один потомок. В цикле алгоритм проверяет, существует ли у узла правый потомок (некоторые узлы могут иметь только левого потомка), и если существует — сравнивает ключи потомков, присваивая `largerChild` индекс большего потомка.

Затем алгоритм проверяет, что ключ исходного узла (который теперь находится в `top`) больше ключа `largerChild`. Если условие выполнено, то процесс смещения вниз завершен, и выполнение цикла прерывается.

```
public void trickleDown(int index)
   {
   int largerChild;
   Node top = heapArray[index];       // Сохранение корня
   while(index < currentSize/2)       // Пока у узла имеется
      {                               // хотя бы один потомок
      int leftChild = 2*index+1;
      int rightChild = leftChild+1;
                                      // Определение большего потомка
      if( rightChild < currentSize &&  // (Правый потомок существует?)
                       heapArray[leftChild].getKey() <
                       heapArray[rightChild].getKey() )
         largerChild = rightChild;
      else
         largerChild = leftChild;
                                       // top >= largerChild?
      if(top.getKey() >= heapArray[largerChild].getKey())
```

```
         break;
                                    // Потомок сдвигается вверх
      heapArray[index] = heapArray[largerChild];
      index = largerChild;          // Переход вниз
      }
   heapArray[index] = top;          // index <- корень
   }
```

При выходе из цикла остается лишь вернуть узел, хранящийся в `top`, на его законное место, на которое указывает переменная `index`.

Изменение ключа

С готовыми методами `trickleDown()` и `trickleUp()` мы можем легко реализовать алгоритм изменения приоритета (ключа) узла, с последующим смещением узла вверх или вниз до нужной позиции. Операция изменения ключа выполняется методом `change()`:

```
public boolean change(int index, int newValue)
   {
   if(index<0 || index>=currentSize)
      return false;
   int oldValue = heapArray[index].getKey(); // Сохранение старого ключа
   heapArray[index].setKey(newValue);  // Присваивание нового ключа

   if(oldValue < newValue)             // Если узел повышается,
      trickleUp(index);                // выполняется смещение вверх
   else                                // Если узел понижается,
      trickleDown(index);              // выполняется смещение вниз
   return true;
   }
```

Метод сначала проверяет, что в первом аргументе был передан действительный индекс, и если проверка проходит успешно — записывает в поле `iData` узла с заданным индексом значение, переданное во втором аргументе. Если приоритет увеличился, то узел смещается вверх, а если уменьшился — смещается вниз.

Вообще говоря, в методе не показана самая трудная часть изменения приоритета: поиск изменяемого узла. В приведенном методе `change()` индекс передается в аргументе, а в приложении Heap Workshop пользователь просто щелкает на нужном узле. В реальном приложении потребуется механизм поиска правильного узла; как было сказано ранее, в пирамиде можно легко и просто обратиться только к одному узлу — корневому с наибольшим ключом.

Задача может быть решена последовательным поиском по массиву с линейной сложностью $O(N)$. Также каждое появление узла в приоритетной очереди может сопровождаться обновлением другой структуры данных (например, хеш-таблицы). Отдельная структура данных обеспечит быстрый доступ к любому узлу, но ее обновление само по себе требует затрат времени.

Размер массива

Размер массива (количество узлов в пирамиде) — важнейший атрибут состояния пирамиды и необходимое поле класса `Heap`. Узлы, скопированные из последней ячейки, не стираются, поэтому алгоритм может определить положение последней занятой ячейки только по текущему размеру массива.

Программа heap.java

В программе `heap.java` (листинг 12.1) используется класс `Node`. Единственным полем этого класса является переменная `iData`, в которой хранится ключ узла. Естественно, в реальной программе этот класс содержал бы много других полей. Класс `Heap` содержит методы, упомянутые выше, а также методы `isEmpty()` и `displayHeap()`. Последний метод выводит примитивное, но наглядное символьное представление содержимого пирамиды.

Листинг 12.1. Программа heap.java

```
// heap.java
// Работа с пирамидой
// Запуск программы: C>java HeapApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;              // Данные (ключ)
// -------------------------------------------------------------
   public Node(int key)            // Конструктор
      { iData = key; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void setKey(int id)
      { iData = id; }
// -------------------------------------------------------------
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;            // Размер массива
   private int currentSize;        // Количество узлов в массиве
// -------------------------------------------------------------
   public Heap(int mx)             // Конструктор
      {
      maxSize = mx;
      currentSize = 0;
```

```
      heapArray = new Node[maxSize];  // Создание массива
      }
// -------------------------------------------------------------
   public boolean isEmpty()
      { return currentSize==0; }
// -------------------------------------------------------------
   public boolean insert(int key)
      {
      if(currentSize==maxSize)
         return false;
      Node newNode = new Node(key);
      heapArray[currentSize] = newNode;
      trickleUp(currentSize++);
      return true;
   }
// -------------------------------------------------------------
   public void trickleUp(int index)
      {
      int parent = (index-1) / 2;
      Node bottom = heapArray[index];
      while( index > 0 &&
                heapArray[parent].getKey() < bottom.getKey() )
         {
         heapArray[index] = heapArray[parent];  // Узел смещается вниз
         index = parent;
         parent = (parent-1) / 2;
         }
      heapArray[index] = bottom;
   }
// -------------------------------------------------------------
   public Node remove()           // Удаление элемента с наибольшим ключом
      {                           // (Предполагается, что список не пуст)
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
   }
// -------------------------------------------------------------
   public void trickleDown(int index)
      {
      int largerChild;
      Node top = heapArray[index];       // Сохранение корня
      while(index < currentSize/2)       // Пока у узла имеется
         {                               // хотя бы один потомок
         int leftChild = 2*index+1;
         int rightChild = leftChild+1;
                                         // Определение большего потомка
         if(rightChild < currentSize &&  // (Правый потомок существует?)
                              heapArray[leftChild].getKey() <
```

продолжение ⇩

Листинг 12.1 (*продолжение*)

```
                              heapArray[rightChild].getKey())
            largerChild = rightChild;
         else
            largerChild = leftChild;
                                       // top >= largerChild?
         if( top.getKey() >= heapArray[largerChild].getKey() )
            break;
                                       // Потомок сдвигается вверх
         heapArray[index] = heapArray[largerChild];
         index = largerChild;          // Переход вниз
         }
      heapArray[index] = top;          // index <- корень
   }
// -------------------------------------------------------------
   public boolean change(int index, int newValue)
      {
      if(index<0 || index>=currentSize)
         return false;
      int oldValue = heapArray[index].getKey(); // Сохранение старого ключа
      heapArray[index].setKey(newValue);  // Присваивание нового ключа

      if(oldValue < newValue)             // Если узел повышается,
         trickleUp(index);                // выполняется смещение вверх.
      else                                // Если узел понижается,
         trickleDown(index);              // выполняется смещение вниз.
      return true;
      }
// -------------------------------------------------------------
   public void displayHeap()
      {
      System.out.print("heapArray: ");    // Формат массива
      for(int m=0; m<currentSize; m++)
         if(heapArray[m] != null)
            System.out.print( heapArray[m].getKey() + " ");
         else
            System.out.print( "-- ");
      System.out.println();
                                       // Формат пирамиды
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                       // Текущий элемент
      String dots = "................................";
      System.out.println(dots+dots);      // Верхний пунктир

      while(currentSize > 0)              // Для каждого элемента пирамиды
         {
         if(column == 0)                  // Первый элемент в строке?
            for(int k=0; k<nBlanks; k++)  // Предшествующие пробелы
```

```
            System.out.print(' ');
                                       // Вывод элемента
         System.out.print(heapArray[j].getKey());
         if(++j == currentSize)        // Вывод завершен?
            break;

         if(++column==itemsPerRow)     // Конец строки?
            {
            nBlanks /= 2;              // Половина пробелов
            itemsPerRow *= 2;          // Вдвое больше элементов
            column = 0;                // Начать заново
            System.out.println();      // Переход на новую строку
            }
         else                          // Следующий элемент в строке
            for(int k=0; k<nBlanks*2-2; k++)
               System.out.print(' ');  // Внутренние пробелы
         }
      System.out.println("\n"+dots+dots); // Нижний пунктир
      }
// --------------------------------------------------------------
   }  // Конец класса Heap
////////////////////////////////////////////////////////////////
class HeapApp
   {
   public static void main(String[] args) throws IOException
      {
      int value, value2;
      Heap theHeap = new Heap(31);  // Создание пирамиды с максимальным
      boolean success;              // размером 31

      theHeap.insert(70);           // Вставка 10 items
      theHeap.insert(40);
      theHeap.insert(50);
      theHeap.insert(20);
      theHeap.insert(60);
      theHeap.insert(100);
      theHeap.insert(80);
      theHeap.insert(30);
      theHeap.insert(10);
      theHeap.insert(90);

      while(true)           // Пока пользователь не нажмет Ctrl+C
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, remove, change: ");
         int choice = getChar();
         switch(choice)
            {
            case 's':                        // Вывод
```

продолжение ⇩

Листинг 12.1 (*продолжение*)

```
            theHeap.displayHeap();
            break;
         case 'i':                        // Вставка
            System.out.print("Enter value to insert: ");
            value = getInt();
            success = theHeap.insert(value);
            if( !success )
               System.out.println("Can't insert; heap full");
            break;
         case 'r':                        // Удаление
            if( !theHeap.isEmpty() )
               theHeap.remove();
            else
               System.out.println("Can't remove; heap empty");
            break;
         case 'c':                        // Изменение приоритета
            System.out.print("Enter current index of item: ");
            value = getInt();
            System.out.print("Enter new key: ");
            value2 = getInt();
            success = theHeap.change(value, value2);
            if( !success )
               System.out.println("Invalid index");
            break;
         default:
            System.out.println("Invalid entry\n");
         }
      }
   }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
}
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
```

```
      }
//--------------------------------------------------------------
   }  // Конец класса HeapApp
////////////////////////////////////////////////////////////////
```

В массиве корень пирамиды размещается в ячейке с индексом 0. Некоторые реализации пирамиды размещают корень в ячейке 1, а ячейка 0 используется в качестве «сторожа» с наибольшим возможным ключом. Такое решение немного ускоряет некоторые алгоритмы, но усложняет реализацию на концептуальном уровне.

Метод `main()` приложения HeapApp создает пирамиду с максимальным размером 31 (обусловленным ограничениями метода вывода) и вставляет в нее 10 узлов со случайными ключами. Затем программа входит в цикл, в котором пользователь вводит команды *s*, *i*, *r* или *c* (соответственно вывод, вставка, удаление или изменение).

Пример взаимодействия с программой:

```
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40
.............................................................
                              100
               90                             80
        30             60             50             70
    20      10      40
.............................................................
Enter first letter of show, insert, remove, change: i
Enter value to insert: 53
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40 53
.............................................................
                              100
               90                             80
        30             60             50             70
    20      10      40      53
.............................................................
Enter first letter of show, insert, remove, change: r
Enter first letter of show, insert, remove, change: s
heapArray: 90 60 80 30 53 50 70 20 10 40
.............................................................
                               90
               60                             80
        30             53             50             70
    20      10      40
.............................................................
Enter first letter of show, insert, remove, change:
```

Пользователь выводит состояние пирамиды, добавляет элемент с ключом 53, снова выводит, удаляет элемент с наибольшим ключом и выводит состояние пирамиды в третий раз. Метод `show()` выводит оба представления пирамиды: как в виде массива, так и в виде дерева. Сила воображения поможет вам заполнить связи между узлами.

Расширение массива

Что произойдет, если во время выполнения программы в массив пирамиды будет вставлено слишком много элементов? Программа может создать новый массив и скопировать в него данные из старого массива. (В отличие от хеш-таблиц изменение размера пирамиды не требует изменения порядка данных.) Операция копирования выполняется за линейное время, но расширение массива выполняется относительно редко, особенно если размер массива существенно увеличивается при каждом расширении (скажем, вдвое).

ПОЛЕЗНЫЙ СОВЕТ

В языке Java вместо массива можно воспользоваться объектом класса Vector — векторы могут расширяться динамически.

Эффективность операций с пирамидой

Для пирамиды со сколько-нибудь значительным количеством элементов алгоритмы смещения вверх и вниз оказываются наиболее затратными составляющими всех представленных операций. Эти алгоритмы выполняются в цикле: узлы шаг за шагом смещаются вверх и вниз по пути. Количество необходимых операций копирования ограничивается высотой пирамиды: если пирамида состоит из пяти уровней, то четыре операции копирования переместят «дыру» от верхнего уровня до нижнего. (Не будем учитывать два перемещения последнего узла во временное хранилище и обратно; они необходимы всегда и поэтому выполняются за постоянное время.)

В цикле метода `trickleUp()` выполняется всего одна существенная операция: сравнение ключа нового узла с ключом узла в текущей позиции. Метод `trickleDown()` требует двух сравнений: для поиска большего потомка и для его сравнения с «последним» узлом. В обоих случаях операция завершается копированием узла сверху вниз или снизу вверх.

Пирамида является частным случаем двоичного дерева, а как было показано в главе 8, количество уровней L в двоичном дереве равно $\log_2(N+1)$, где N — количество узлов. Циклы методов `trickleUp()` и `trickleDown()` выполняются за $L-1$ итераций, поэтому время выполнения первого пропорционально $\log_2 N$, а время выполнения второго чуть больше из-за дополнительного сравнения. Таким образом, все операции с пирамидой, рассмотренные в этом разделе, выполняются за время $O(\log N)$.

Пирамидальное дерево

На иллюстрациях этой главы пирамиды изображались в виде деревьев, потому что такое представление получается более наглядным. При этом для реализации пирамиды был выбран массив. Тем не менее пирамида может быть реализована и на базе дерева. Такое дерево будет двоичным, но оно не будет деревом двоичного

поиска, потому что, как мы уже видели, оно обладает слабой упорядоченностью. Кроме того, дерево будет полным (то есть в нем не будет отсутствующих узлов). Мы будем называть такое дерево *пирамидальным*.

Одна из проблем при работе с пирамидальными деревьями связана с поиском последнего узла. Эта операция необходима для удаления наибольшего элемента, потому что это последний узел, который вставляется на место удаляемого корня (а затем смещается вниз). Также может возникнуть необходимость в поиске первого пустого узла, потому что в этом месте вставляется новый узел (с последующим смещением вверх). Обнаружить эти узлы посредством поиска не удастся, потому что их значения неизвестны, к тому же дерево не является деревом поиска. Но если реализация отслеживает количество узлов, найти их в полном дереве будет несложно.

Как было показано при обсуждении дерева Хаффмана в главе 8, путь от корня до листа может быть представлен в виде двоичного числа. Двоичные цифры этого числа обозначают направление перехода от каждого родителя к потомку: 0 для левого, 1 для правого.

Оказывается, между количеством узлов в дереве и двоичным числом, кодирующим путь к последнему узлу, существует простая связь. Допустим, корню присвоен номер 1; на втором уровне находятся узлы 2 и 3; на третьем — узлы 4, 5, 6, 7 и т. д. Возьмите номер искомого узла — последнего узла или первого пустого узла. Преобразуйте номер узла в двоичное представление. Предположим, дерево состоит из 29 узлов, и вы хотите найти последний узел. Десятичное число 29 соответствует 11101 в двоичной записи. Удалите начальную единицу; остается 1101. Эта запись определяет путь от корня к узлу 29: направо, направо, налево, направо. Номер первого свободного узла 30 в двоичной записи имеет вид 1110 (после удаления начальной единицы): направо, направо, направо, налево.

Двоичное представление числа может быть получено многократным применением оператора `%` для получения остатка от деления номера узла на 2 (0 или 1) и последующего целочисленного деления *n* на 2 оператором `/`. Когда значение *n* станет меньше единицы, вычисления закончены. Последовательность остатков, которую можно сохранить в массиве или строке, определяет двоичное число. (Младшие биты находятся в начале строки.) Реализация может выглядеть так:

```
while(n >= 1)
   {
   path[j++] = n % 2;
   n = n / 2;
   }
```

Также существует рекурсивное решение: остатки вычисляются при каждом рекурсивном вызове, а соответствующее направление выбирается при возврате управления.

После того как нужный узел (или пустой потомок) будет найден, операции с пирамидой становятся тривиальными. При смещении вверх или вниз структура дерева не изменяется, поэтому выполнять фактическое перемещение узлов не нужно — достаточно копировать данные из одного узла в следующий. Это позволяет избежать присоединения и отсоединения всех потомков и родителей при простом

перемещении. Класс Node должен содержать поле родительского узла, потому что смещение вверх требует обращения к родителю. Реализация пирамидального дерева предоставляется читателю в виде программного проекта.

Операции с пирамидальным деревом выполняются за время $O(\log N)$. Как и при реализации пирамиды на базе массива, время тратится в основном на операции смещения вверх и вниз, сложность которых пропорциональна высоте дерева.

Пирамидальная сортировка

Эффективность пирамиды как структуры данных является одной из предпосылок на удивление простого и эффективного алгоритма сортировки, называемого *пирамидальной сортировкой*.

Алгоритм строится на вставке всех неупорядоченных элементов в пирамиду обычным методом insert(). Затем многократные вызовы remove() извлекают элементы в порядке сортировки. Реализация может выглядеть примерно так:

```
for(j=0; j<size; j++)
   theHeap.insert( anArray[j] );   // Из несортированного массива
for(j=0; j<size; j++)
   anArray[j] = theHeap.remove();  // В отсортированный массив
```

Так как методы insert() и remove() выполняются за время $O(\log N)$, и каждый из них должен быть выполнена N раз, вся сортировка выполняется за время $O(N^*\log N)$ — такое же, как при быстрой сортировке. Однако по скорости этот алгоритм все же уступает быстрой сортировке — отчасти из-за того, что во внутреннем цикле while метода trickleDown() выполняется больше операций, чем во внутреннем цикле быстрой сортировки.

Однако существует пара приемов, повышающих эффективность пирамидальной сортировки. Первый экономит время, а второй — память.

Ускоренное смещение вниз

Вставляя в пирамиду N новых элементов, мы применяем метод trickleUp() N раз. Однако все элементы можно вставить в случайных позициях массива, а затем восстановить иерархию пирамиды всего $N/2$ применениями trickleDown(). Этот прием обеспечивает небольшой прирост скорости.

Две правильные субпирамиды образуют правильную пирамиду

Чтобы понять, как работает это решение, необходимо знать, что trickleDown() создает правильную пирамиду в том случае, если при нарушении корневым элементом условия пирамиды обе дочерних субпирамиды этого корня являются правильными пирамидами. (Корневой элемент может быть корнем как субпирамиды, так и всей пирамиды.) Ситуация показана на рис. 12.8.

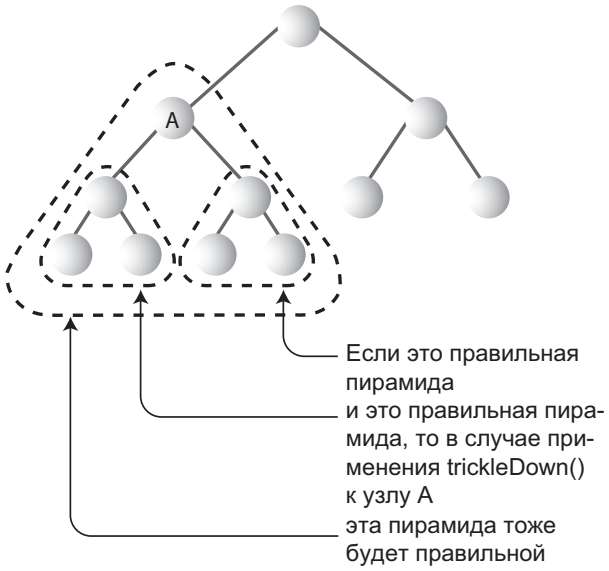

Рис. 12.8. Обе субпирамиды должны быть правильными

Из этого факта следует способ преобразования неупорядоченного массива в пирамиду. Мы можем применить `trickleDown()` к узлам у низа (будущей) пирамиды, то есть в конце массива, и двигаться вверх к корню с индексом 0. На каждом шаге нижние субпирамиды будут правильными, потому что метод `trickleDown()` к ним уже был применен. После применения `trickleDown()` к корню неупорядоченный массив будет преобразован в пирамиду.

Однако следует заметить, что узлы нижнего уровня (не имеющие потомков) уже являются правильными пирамидами, потому что они представляют собой деревья из одного узла; у них просто нет связей, которые могли бы быть нарушены. Следовательно, применять к этим узлам `trickleDown()` не нужно. Начинать следует с узла $N/2 - 1$, крайнего правого узла с потомком, вместо последнего узла $N - 1$. А следовательно, по сравнению с N-кратным использованием `insert()` количество смещений сокращается вдвое. На рис. 12.9 показан порядок применения алгоритма смещения вниз, начиная с узла 6 в пирамиде из 15 узлов.

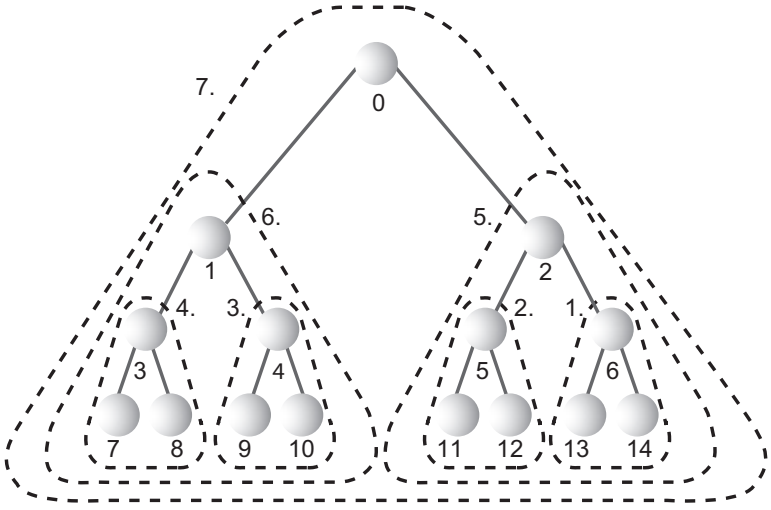

Рис. 12.9. Порядок применения trickleDown()

В следующем фрагменте кода метод `trickleDown()` применяется ко всем узлам, кроме узлов нижнего уровня, от позиции $N/2 - 1$ и до корня дерева:

```
for(j=size/2-1; j >=0; j--)
   theHeap.trickleDown(j);
```

Рекурсивное решение

Для построения пирамиды из массива также можно воспользоваться рекурсией. Метод `heapify()` вызывается для корня дерева. Он вызывает себя для двух потомков корня, затем для двух потомков каждого из этих потомков и т. д. В конечном итоге он доберется до нижнего уровня, где немедленно вернет управление при обнаружении узла, не имеющего потомков.

Вызвав себя для двух дочерних поддеревьев, метод `heapify()` затем вызывает `trickleDown()` для корня поддерева. Этот вызов гарантирует, что поддерево представляет собой правильную пирамиду. Затем `heapify()` возвращает управление, а работа с поддеревом продолжается уровнем выше.

```
heapify(int index)      // Преобразование массива в пирамиду
   {
   if(index > N/2-1)    // Если узел не имеет потомков,
      return;           // возврат управления
   heapify(index*2+2);  // Правое поддерево преобразуется в пирамиду
   heapify(index*2+1);  // Левое поддерево преобразуется в пирамиду
   trickleDown(index);  // Узел смещается вниз
   }
```

Вероятно, по эффективности рекурсивное решение уступает простому циклу.

Сортировка «на месте»

В исходном фрагменте кода неупорядоченные данные хранятся в массиве. Затем эти данные вставляются в пирамиду, затем извлекаются из нее и записываются обратно в массив в порядке сортировки. Для этой процедуры необходимы два массива размера *N*: исходный и массив, используемый для хранения пирамиды.

В действительности для хранения пирамиды и исходных данных достаточно одного массива. Это вдвое сокращает объем памяти, необходимой для пирамидальной сортировки; алгоритм не требует дополнительных затрат памяти за пределами исходного массива.

Мы уже видели, что массив может быть преобразован в пирамиду применением `trickleDown()` к половине элементов массива. Преобразование неупорядоченного массива в пирамиду осуществляется «на месте»; для выполнения этой операции достаточно одного массива. Таким образом, для первого шага пирамидальной сортировки достаточно одного массива.

Однако ситуация усложняется с повторным применением `remove()` к пирамиде. Где хранить извлекаемые элементы?

При каждом извлечении элемента из пирамиды в конце массива появляется свободная ячейка; размер пирамиды уменьшается на единицу. Только что из-

влеченный элемент можно сохранить в освободившейся ячейке. С извлечением других элементов массив пирамиды становится все меньше, а массив упорядоченных данных — все больше. Таким образом, при соответствующем планировании упорядоченный массив и массив пирамиды смогут совместно использовать одно пространство (рис. 12.10).

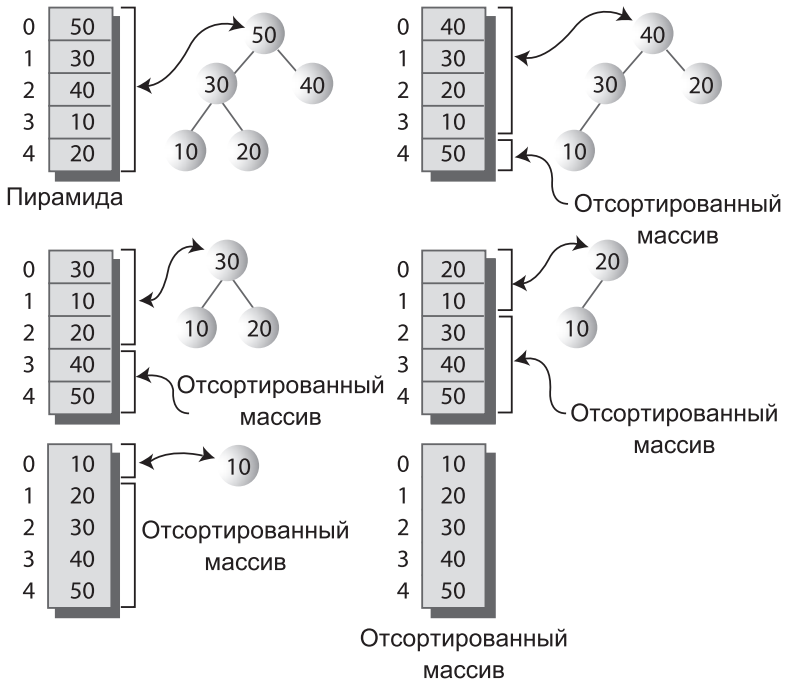

Рис. 12.10. Массив двойного назначения

Программа heapSort.java

Мы объединим эти два приема — применение `trickleDown()` без `insert()` и использование одного массива для хранения исходных данных и пирамиды — в программе для выполнения пирамидальной сортировки. В листинге 12.2 приведен полный код программы `heapSort.java`.

Листинг 12.2. Программа heapSort.java

```
// heapSort.java
// Пирамидальная сортировка
// Запуск программы: C>java HeapSortApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;              // Данные (ключ)
// -------------------------------------------------------------
   public Node(int key)            // Конструктор
      { iData = key; }
// -------------------------------------------------------------
```

продолжение ⇩

Листинг 12.2 (*продолжение*)

```
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;           // Размер массива
   private int currentSize;
// -------------------------------------------------------------
   public Heap(int mx)
      {
      maxSize = mx;
      currentSize = 0;
      heapArray = new Node[maxSize];
      }
// -------------------------------------------------------------
   public Node remove()
      {
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
      }
// -------------------------------------------------------------
   public void trickleDown(int index)
      {
      int largerChild;
      Node top = heapArray[index];        // Сохранение корня
      while(index < currentSize/2)        // до нижнего уровня
         {
         int leftChild = 2*index+1;
         int rightChild = leftChild+1;
                                          // Определение большего потомка
         if(rightChild < currentSize &&   // (Правый потомок существует?)
                           heapArray[leftChild].getKey() <
                           heapArray[rightChild].getKey())
            largerChild = rightChild;
         else
            largerChild = leftChild;
                                          // top >= largerChild?
         if(top.getKey() >= heapArray[largerChild].getKey())
            break;
                                          // Потомок сдвигается вверх
         heapArray[index] = heapArray[largerChild];
         index = largerChild;             // Переход вниз
         }
      heapArray[index] = top;             // index <- корень
      }
```

```
// -------------------------------------------------------------
   public void displayHeap()
      {
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                          // Текущий элемент
      String dots = "................................";
      System.out.println(dots+dots);      // Верхний пунктир

      while(currentSize > 0)              // Для каждого элемента пирамиды
         {
         if(column == 0)                  // Первый элемент в строке?
            for(int k=0; k<nBlanks; k++)  // Предшествующие пробелы
               System.out.print(' ');
                                          // Вывод элемента
         System.out.print(heapArray[j].getKey());

         if(++j == currentSize)           // Вывод завершен?
            break;

         if(++column==itemsPerRow)        // Конец строки?
            {
            nBlanks /= 2;                 // Половина пробелов
            itemsPerRow *= 2;             // Вдвое больше элементов
            column = 0;                   // Начать заново
            System.out.println();         // Переход на новую строку
            }
         else                             // Следующий элемент в строке
            for(int k=0; k<nBlanks*2-2; k++)
               System.out.print(' ');     // Внутренние пробелы
         }
      System.out.println("\n"+dots+dots); // Нижний пунктир
      }
// -------------------------------------------------------------
   public void displayArray()
      {
      for(int j=0; j<maxSize; j++)
      System.out.print(heapArray[j].getKey() + " ");
      System.out.println("");
      }
// -------------------------------------------------------------
   public void insertAt(int index, Node newNode)
      { heapArray[index] = newNode; }
// -------------------------------------------------------------
   public void incrementSize()
      { currentSize++; }
// -------------------------------------------------------------
   }  // Конец класса Heap
```

продолжение ⇩

Листинг 12.2 (*продолжение*)

```
////////////////////////////////////////////////////////////////
class HeapSortApp
   {
   public static void main(String[] args) throws IOException
      {
      int size, j;
      System.out.print("Enter number of items: ");
      size = getInt();
      Heap theHeap = new Heap(size);

      for(j=0; j<size; j++)       // Заполнение массива
         {                        // случайными данными
         int random = (int)(java.lang.Math.random()*100);
         Node newNode = new Node(random);
         theHeap.insertAt(j, newNode);
         theHeap.incrementSize();
         }

         System.out.print("Random: ");
            theHeap.displayArray();  // Вывод случайного массива

         for(j=size/2-1; j>=0; j--)  // Преобразование массива в пирамиду
            theHeap.trickleDown(j);

         System.out.print("Heap:   ");
         theHeap.displayArray();     // Вывод массива
         theHeap.displayHeap();      // Вывод пирамиды

         for(j=size-1; j>=0; j--)    // Извлечение из пирамиды
            {                        // с сохранением в конце массива
            Node biggestNode = theHeap.remove();
            theHeap.insertAt(j, biggestNode);
            }
         System.out.print("Sorted: ");
         theHeap.displayArray();     // Вывод отсортированного массива
         }
// --------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
```

```
      return Integer.parseInt(s);
      }
// --------------------------------------------------------
   }  // Конец класса HeapSortApp
```

Класс `Heap` почти не отличается от одноименного класса из программы `heap.java` (см. листинг 12.1), если не считать того, что для экономии места из него были исключены методы `trickleUp()` и `insert()`, лишние для пирамидальной сортировки. В класс включен метод `insertAt()` для прямой вставки в массив пирамиды.

Учтите, что это добавление не соответствует духу объектно-ориентированного программирования. Интерфейс класса `Heap` должен ограждать пользователей класса от подробностей реализации пирамиды. Базовый массив должен оставаться невидимым, однако метод `insertAt()` позволяет выполнять прямую вставку. В такой ситуации нарушение принципов ООП приемлемо, потому что массив так тесно связан с архитектурой пирамиды.

В классе пирамиды также появился новый метод `incrementSize()`. Может показаться, что его можно объединить с `insertAt()`, но при вставке в массив в его качестве упорядоченного массива размер пирамиды увеличиваться не должен, поэтому мы разделяем эти две функции.

Метод `main()` класса `HeapSortApp`:

1) запрашивает у пользователя размер массива;
2) заполняет массив случайными данными;
3) преобразует массив в пирамиду $N/2$ с применениями метода `trickleDown()`;
4) извлекает элементы из кучи и записывает их обратно в конец массива.

После каждого шага выводится содержимое массива. Пример вывода программы `heapSort.java`:

```
Enter number of items: 10
Random: 81 6 23 38 95 71 72 39 34 53
Heap:   95 81 72 39 53 71 23 38 34 6
..............................................................
                               95
               81                              72
        39             53             71             23
    38      34      6
..............................................................
Sorted: 6 23 34 38 39 53 71 72 81 95
```

Эффективность пирамидальной сортировки

Как упоминалось ранее, пирамидальная сортировка выполняется за время $O(N^*\log N)$. Хотя по скорости она слегка уступает быстрой сортировке, ее преимуществом перед быстрой сортировкой является меньшая чувствительность к исходному распределению данных. При некоторых конфигурациях ключей быстрая сортировка замедляется до $O(N^2)$, тогда как пирамидальная сортировка выполняется за время $O(N^*\log N)$ независимо от распределения данных.

Итоги

- Приоритетная очередь представляет собой абстрактный тип данных (ADT) с методами для вставки данных и извлечения наибольшего (или наименьшего) элемента.
- Пирамида является эффективной реализацией приоритетной очереди ADT.
- Пирамида обеспечивает извлечение наибольшего элемента и вставку за время $O(N^*\log N)$.
- Наибольший элемент всегда находится в корне.
- Пирамида не поддерживает упорядоченный обход данных, поиск элемента с заданным ключом или удаление.
- Пирамида обычно реализуется на базе массива, представляющего полное двоичное дерево. Корневой элемент хранится в ячейке с индексом 0, а последний элемент — в ячейке с индексом $N - 1$.
- Ключ каждого узла меньше ключа его родителя и больше ключей его потомков.
- Вставляемый элемент всегда размещается в первой свободной ячейке массива, а затем смещается вверх до нужной позиции.
- Элемент, извлекаемый из корня, заменяется последним элементом массива, который затем смещается вниз до нужной позиции.
- Смещения вверх и вниз можно представить в виде последовательности перестановок элементов, но они более эффективно реализуются как последовательность операций копирования.
- Пирамида позволяет изменить приоритет произвольного элемента. Сначала изменяется ключ элемента. Если ключ увеличился, то элемент смещается вверх, а если уменьшился, то элемент смещается вниз.
- Пирамида может быть реализована на базе двоичного дерева (не дерева поиска), воспроизводящего структуру пирамиды; такое дерево называется пирамидальным.
- Существуют алгоритмы поиска последнего занятого или первого свободного узла в пирамидальном дереве.
- Пирамидальная сортировка — эффективный алгоритм поиска, выполняемый за время $O(N^*\log N)$.
- На концептуальном уровне пирамидальная сортировка состоит из N вставок в пирамиду с последующими N извлечениями.
- Пирамидальную сортировку можно ускорить, применяя алгоритм смещения вниз напрямую к $N/2$ элементам неупорядоченного массива (вместо вставки N элементов).
- Неупорядоченные данные, массив пирамиды и итоговые отсортированные данные могут храниться в одном массиве. Таким образом, пирамидальная сортировка не требует дополнительных затрат памяти.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении B.

1. Что означает термин «полный» применительно к двоичным деревьям?
   a) Все необходимые данные вставлены в дерево.
   b) Все уровни заполнены узлами (возможно, кроме нижнего).
   c) Все существующие узлы содержат данные.
   d) Конфигурация узлов соответствует условию пирамиды.
2. Что означает термин «слабая упорядоченность» применительно к двоичным деревьям?
3. Узлы в пирамиде всегда извлекаются из _____.
4. В ходе смещения вверх в пирамиде, упорядоченной по убыванию:
   a) узел многократно меняется местами со своим родителем, пока не станет больше родителя;
   b) узел многократно меняется местами со своим потомком, пока не станет больше потомка;
   c) узел многократно меняется местами со своим потомком, пока не станет меньше потомка;
   d) узел многократно меняется местами со своим родителем, пока не станет меньше родителя.
5. Пирамида может быть представлена в виде массива, потому что пирамида:
   a) полна;
   b) обладает слабой упорядоченностью;
   c) является двоичным деревом;
   d) удовлетворяет условию пирамиды.
6. Последний узел пирамиды:
   a) всегда является левым потомком;
   b) всегда является правым потомком;
   c) всегда находится на нижнем уровне;
   d) никогда не бывает меньше своего «брата».
7. Пирамида по отношению к приоритетной очереди является тем же, чем _____ является по отношению к стеку.
8. При вставке в пирамиде, упорядоченной по убыванию, используется смещение _____.
9. Пирамидальная сортировка основана на:
   a) извлечении данных из пирамиды и их повторной вставке;
   b) вставке данных в пирамиду и их последующем извлечении;

c) копировании данных из одной пирамиды в другую;

d) копировании данных в пирамиду из массива, представляющего пирамиду.

10. Сколько массивов, размер каждого из которых достаточен для хранения всех данных, требуется для сортировки пирамиды?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Влияет ли порядок вставки данных в пирамиду на конфигурацию узлов? Воспользуйтесь приложением Heap Workshop для проверки.
2. Вставьте в приложении Workshop 10 элементов, упорядоченных по возрастанию, в пустую пирамиду (кнопка `Ins`). Если теперь извлекать эти элементы кнопкой `Rem`, будут ли они извлекаться в противоположном порядке?
3. Вставьте несколько элементов с одинаковыми ключами, затем извлеките их. Можно ли по результатам определить, является ли пирамидальная сортировка устойчивой? Цвет узлов является вторичным элементом данных.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

12.1. Преобразуйте программу `heap.java` (см. листинг 12.1), чтобы пирамида была упорядочена по возрастанию, а не по убыванию (То есть корневой узел является наименьшим, а не наибольшим.) Убедитесь в том, что все операции работают правильно.

12.2. В программе `heap.java` метод `insert()` вставляет в пирамиду новый узел и проверяет выполнение условия пирамиды. Напишите метод `toss()`, который помещает новый узел в массив пирамиды без проверки условия. (Например, каждый новый элемент просто помещается в конец массива.) Затем напишите метод `restoreHeap()`, который восстанавливает условие для всей пирамиды. При вставке большого количества элементов многократные вызовы `toss()` с одним итоговым вызовом `restoreHeap()` более эффективны, чем многократные вызовы `insert()`. Чтобы протестировать программу, вставьте в пирамиду несколько элементов, потом добавьте еще несколько при помощи `toss()` и восстановите пирамиду.

12.3. Реализуйте класс `PriorityQ` в программе `priorityQ.java` (см. листинг 4.6) на базе пирамиды (вместо массива). Вы можете использовать класс `Heap` из программы `heap.java` (см. листинг 12.1) без каких-либо изменений. Сделайте очередь упорядоченной по убыванию (с извлечением наибольшего элемента).

12.4. Одной из проблем с реализацией приоритетной очереди в виде пирамиды, реализованной на базе массива, является фиксированный размер массива. Если данные выйдут за границу массива, массив необходимо расширить, как это делалось для хеш-таблиц в программном проекте 11.4 главы 11, «Хеш-таблицы». Проблему можно обойти, реализуя приоритетную очередь на базе обычного дерева двоичного поиска (вместо пирамиды). Такое дерево может разрастаться до неограниченных размеров (если не считать ограничений объема системной памяти).

Начните с класса `Tree` программы `tree.java` (см. листинг 8.1). Внесите изменения в этот класс, соответствующие особенностям приоритетной очереди — добавьте метод извлечения наибольшего элемента `removeMax()`. Для пирамиды метод реализуется просто, для дерева задача усложняется. Как найти наибольший элемент дерева? Нужно ли беспокоиться об обоих потомках извлекаемого узла? Реализация `change()` не является обязательной — в дереве двоичного поиска она легко решается удалением старого элемента и вставкой нового с другим ключом.

Приложение должно работать с классом `PriorityQ`; класс `Tree` должен оставаться невидимым для метода `main()` (допускается только вывод дерева в процессе отладки). Метод вставки и `removeMax()` должны выполняться за время $O(\log N)$.

12.5. Напишите программу, реализующую пирамидальное дерево (реализацию пирамиды на базе дерева), по описанию в тексте главы. Программа должна поддерживать извлечение наибольшего элемента, вставку и изменение ключа.