13.1. Бинарные деревья, представляемые массивами

В гл. 11 для построения бинарных деревьев мы используем узлы дерева. Каждый узел имеет поле данных и поля указателей на правое и левое поддеревья данного узла. Пустое дерево представляется нулевым указателем. Вставки и удаления производятся путем динамического размещения узлов и присвоения значений полям указателей. Это представление используется для целой группы деревьев от вырожденных до законченных. В данном разделе вводится последовательное представление деревьев с помощью массивов. При этом данные хранятся в элементах массива, а узлы указываются индексами. Мы выявим очень близкое родство между массивом и законченным бинарным деревом — взаимосвязь, используемую в пирамидах и очередях приоритетов.

Вспомним из гл. 11, что законченное бинарное дерево глубины n содержит все возможные узлы на уровнях до n-1, а узлы уровня n располагаются слева направо подряд (без дыр). Массив A есть последовательный список, элементы которого могут представлять узлы законченного бинарного дерева с корнем A[0]; потомками первого уровня A[1] и A[2]; потомками второго уровня A[3], A[4], A[5] и A[6] и т.д. Корневой узел имеет индекс 0, а всем остальным узлам индексы назначаются в порядке, определяемом поперечным (уровень за уровнем) методом прохождения. На рис. 13.1 показано законченное бинарное дерево для массива A из десяти элементов.

```
int A[10] = {5, 1, 3, 9, 6, 2, 4, 7, 0, 8}
```

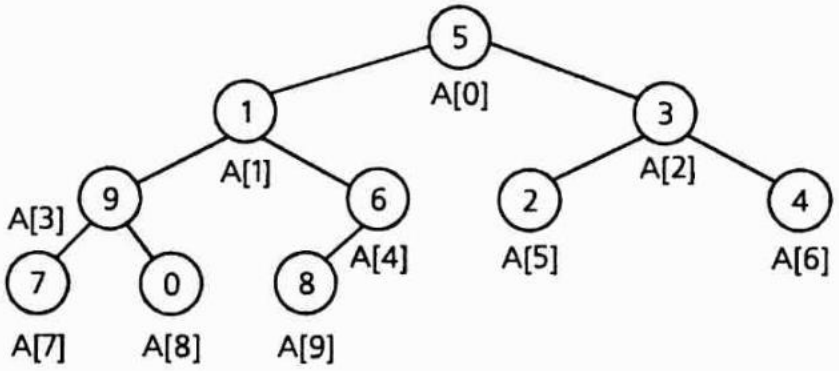
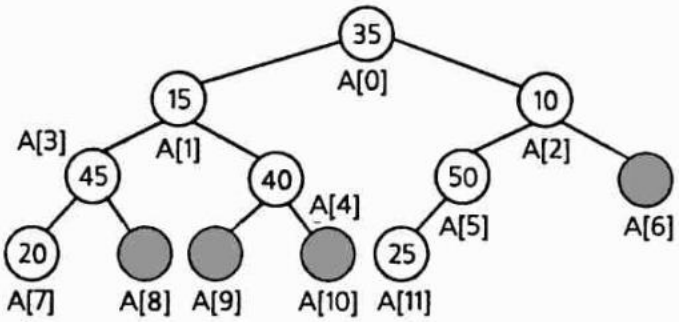


Рис. 13.1. Законченное бинарное дерево для 10-элементного массива A



0	35	6	
1	15	7	20
2	10	8	
3	45	9	
4	40	10	
5	50	11	25

Бинарные деревья

Эквивалентное представление в виде массива

Несмотря на то, что массивы обеспечивают естественное представление деревьев, возникает проблема, связанная с отсутствующими узлами, которым должны соответствовать неиспользуемые элементы массива. В следующем примере массив имеет четыре неиспользуемых элемента, т.е. треть занимаемого деревом пространства. Вырожденное дерево, имеющее только правые поддеревья, дает в этом смысле еще худший результат.

Преимущества представляемых массивами деревьев обнаруживаются тогда, когда требуется прямой доступ к узлам. Индексы, идентифицирующие сыновей и родителя данного узла, вычисляются просто. В таблице 13.1 представлено дерево, изображенное на рис. 13.1. Здесь для каждого уровня указаны узлы, а также их родители и сыновья.

Для каждого узла A[i] в N-элементном массиве индекс его сыновей вычисляется по формулам:

Индекс левого сына = 2*i (неопределен при 2*i + 1 ≥ N)
Индекс правого сына = 2*i + 2 (неопределен при 2*i + 2 ≥ N)

Таблица 13.1

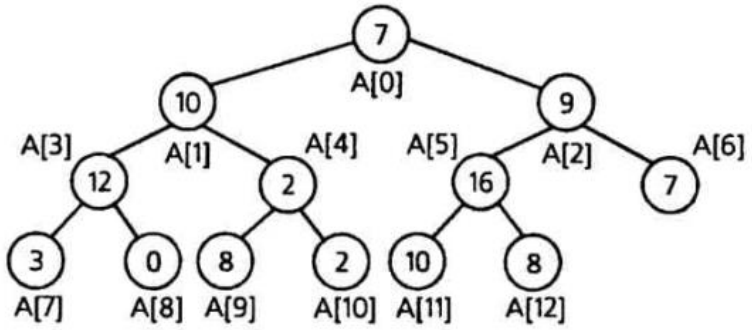
Уровень	Родитель	Значение	Левый сын	Правый сын
0	0	A[0] = 5	1	2
1	1	A[1] = 1	2	4
	2	A[2] = 3	5	6
2	3	A[3] = 9	7	8
	4	A[4] = 6	9	10=NULL
	5	A[5] = 2	11=NULL	12=NULL
	6	A[6] = 4	13=NULL	14=NULL
3	7	A[7] = 7	–	–
	8	A[8] = 0	–	–
	9	A[9] = 8	–	–

Поднимаясь от сыновей к родителю, мы замечаем, что родителем узлов A[3] и A[4] является A[1], родителем A[5] и A[6] — A[2] и т.д. Общая формула для вычисления родителя узла A[i] следующая:

Индекс родителя = (i-1)/2 (неопределен при i=0)

Пример 13.1

Во время прохождения последовательно представленного дерева можно идти вниз к сыновьям или вверх к родителю. Ниже приводятся примеры путей для следующего дерева:



1. Начиная с корня, выбрать путь, проходящий через меньших сыновей.

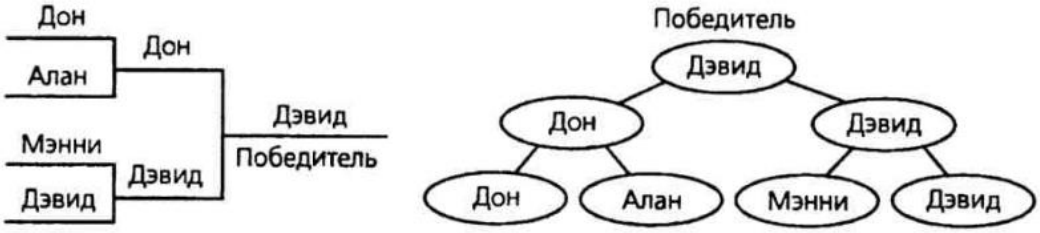
   Путь: A[0] = 7, A[2] = 9, A[6] = 3
2. Начиная с корня, выбрать путь, проходящий через левых сыновей.

   Путь: A[0] = 7, A[1] = 10, A[3] = 12, A[7] = 3
3. Начиная с A[10], выбрать путь, проходящий через родителей.

   Путь: A[10] = 2, A[4] = 2, A[1] = 10, A[0] = 7

Приложение: турнирная сортировка

Бинарные деревья находят важное применение в качестве деревьев принятия решения, в которых каждый узел представляет ситуацию, имеющую два возможных исхода. В частности, для представления спортивного турнира, проводимого по схеме с выбываниями. Каждый нелистовой узел соответствует победителю встречи между двумя игроками. Листовые узлы дают стартовый состав участников и распределение их по парам. Например, победителем теннисного турнира является Дэвид, выигравший финальную встречу с Доном. Оба спортсмена вышли в финал, выиграв предварительные матчи. Дон победил Алана, а Дэвид — Мэнни. Все игры турнира и их результаты могут быть записаны в виде дерева.



В турнире с выбываниями победитель определяется очень скоро. Например, для четырех игроков понадобится всего три матча, а для $2^4 = 16$ участников — $2^4 - 1 = 15$ встреч.

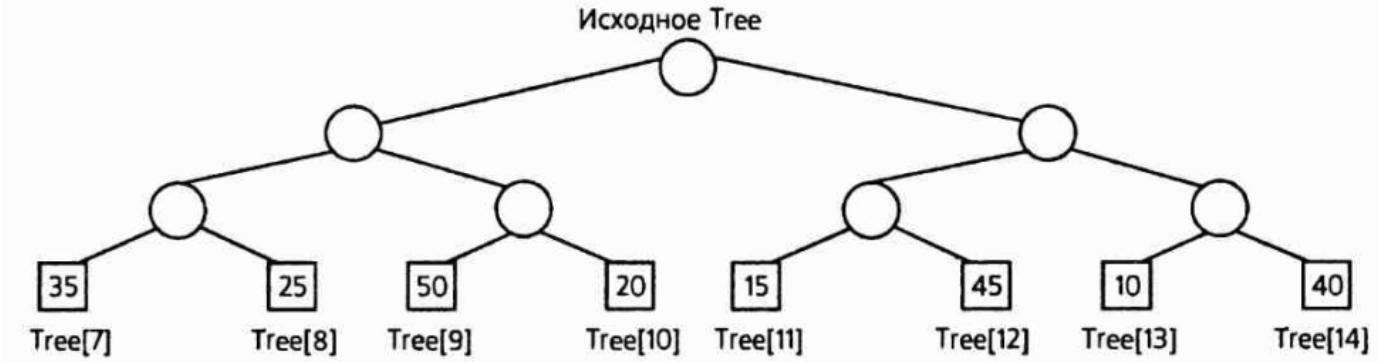
Турнир выявляет победителя, но со вторым лучшим игроком пока не все ясно. Поскольку Дон проиграл финал победителю турнира, он может и не оказаться вторым лучшим игроком. Нам нужно дать шанс Мэнни, так как тот играл матч первого круга с, быть может, единственным игроком, способным его победить. Чтобы выявить второго лучшего игрока, нужно исключить Дэвида и реорганизовать турнирное дерево, устроив матч между Доном и Мэнни.

Как только определится победитель этого матча, мы сможем правильно распределить места.

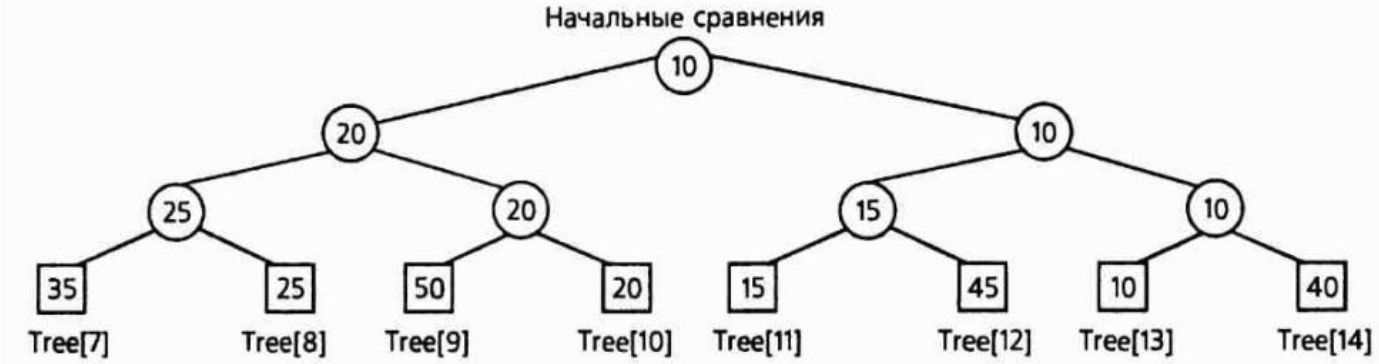
Выиграл Мэнни:	Места	Дэвид Мэнни Дон Алан
Выиграл Дон:	Места	Дэвид Дон Мэнни Алан

Турнирное дерево может использоваться для сортировки списка из N элементов. Рассмотрим эффективный алгоритм, использующий дерево, представленное в виде массива. Пусть имеется последовательно представленное дерево, содержащее N элементов — листовых узлов в нижнем ряду. Эти элементы запоминаются на уровне k, где $2^k \geq N$. Предположим, что список сортируется по возрастанию. Мы сравниваем каждую пару элементов и запоминаем меньший из них (победителя) в родительском узле. Процесс продолжается до тех пор, пока наименьший элемент (победитель турнира) не окажется в корневом узле. Например, приведенное ниже дерево задает следующее начальное состояние массива из $N = 8$ целых чисел. Элементы запоминаются на уровне 3, где $2^3 = 8$.

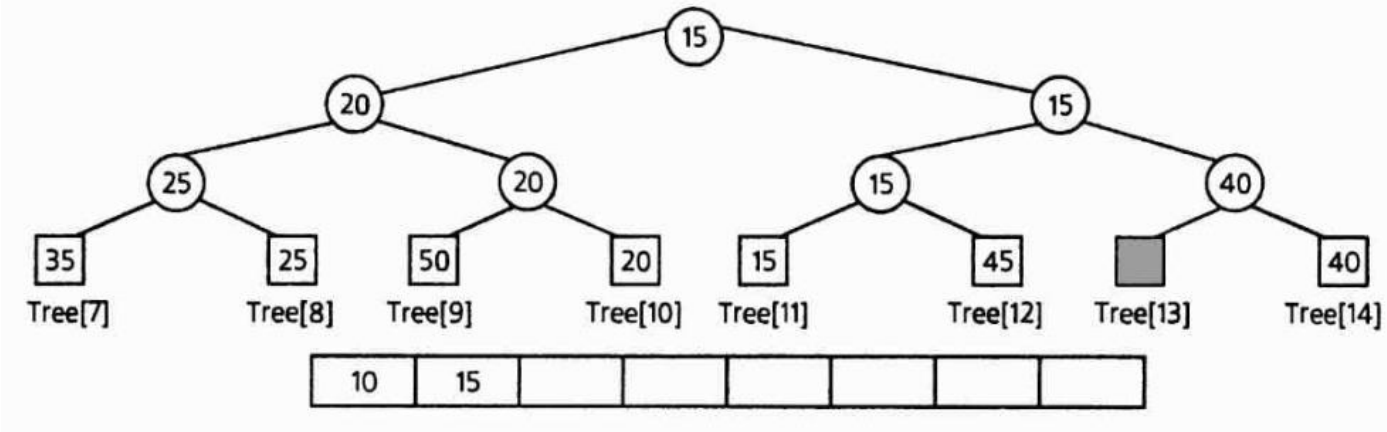
```
A[8] = {35, 25, 50, 20, 15, 45, 10, 40}
```



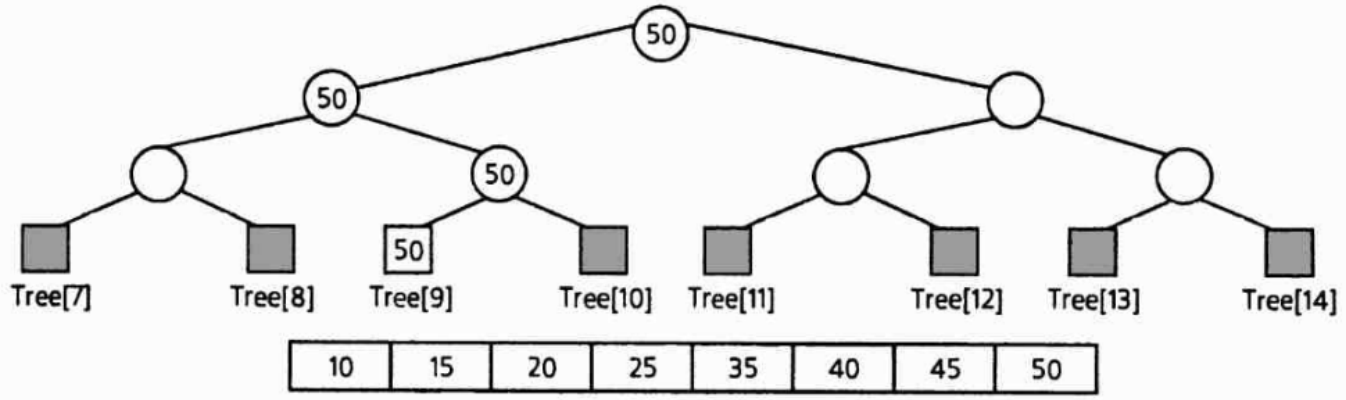
Со второго уровня начинаются "игры" — в родительские узлы помещаются наименьшие значения в парах. Например, "игру" между элементами Tree[7] и Tree[8] выигрывает меньший из них, и значение 25 записывается в Tree[3]. Подобные сравнения проводятся также на втором и первом уровнях. В результате последнего сравнения наименьший элемент попадает в корень дерева на уровне 0.



Как только наименьший элемент оказывается в корневом узле, он удаляется со своего старого места на дереве и копируется в массив. В первый раз в A[0] записывается 10, а затем дерево обновляется для поиска следующего наименьшего элемента. В турнирной модели некоторые матчи должны быть сыграны повторно. Поскольку число 10 изначально было в A[13], проигравший в первом круге A[14] = 40 должен снова участвовать в турнире. A[14] копируется в свой родительский узел A[6], а затем снова проводятся матчи в индексе 6 (15 побеждает 40) и в индексе 2 (15 побеждает 20). В результате 15 попадает в корень и становится вторым наименьшим элементом списка. Корень копируется в A[1], и процесс продолжается.



Процесс продолжается до тех пор, пока все листья не будут удалены. В нашем примере последний (наибольший) узел играет серию матчей, в которых побеждает всех по умолчанию. После копирования числа 50 в A[7] мы получаем отсортированный список.



Вычислительная эффективность. Эффективность турнирной сортировки составляет $O(n \log_2 n)$. В массиве, содержащем $n = 2^k$ элементов, для выявления наименьшего элемента требуется $n-1$ сравнений. Это становится ясным, когда мы замечаем, что половина участников выбывает после каждого круга по мере продвижения к корню. Общее число матчей равно

$$2^{k-1} + 2^{k-2} + \ldots + 2^1 + 1 = n-1$$

Дерево обновляется, и оставшиеся n-1 элементов.обрабатываются посредством $k-1$ сравнений вдоль пути, проходящего через родительские узлы. Общее число сравнений равно

$$(n-1) + (k-1)*(n-1) = (n-1) + (n-1)*(\log_2 n-1) = (n-1)\log_2 n$$

Хотя количество сравнений в турнирной сортировке составляет $O(n \log_2 n)$, использование пустот значительно менее эффективно. Дереву требуется $2 * n-1$ узлов, чтобы вместить $k-1$ кругов соревнования.

Алгоритм TournamentSort. Для реализации турнирной сортировки определим класс DataNode и создадим представленное массивом дерево из объектов этого типа. Членами класса являются элемент данных, его место в нижнем ряду дерева и флажок, показывающий, участвует ли еще этот элемент в турнире. Для сравнения узлов используется перегруженный оператор "<=".

```
template <class T>
class DataNode
{
  public:
    // элемент данных, индекс в массиве, логический флажок
    T data;
    int index;
    int active;

    friend int operator <= (const DataNode<T> &x,
                            const DataNode<T> &y);
};
```

Сортировка реализуется с помощью функции TournamentSort и утилиты UpdateTree, которая производит сравнения вдоль пути предков. Полный листинг функций и переменных, обеспечивающих турнирную сортировку, находится в файле toursort.h.

```
// сформировать последовательное дерево, скопировать туда элементы массива;
// отсортировать элементы и скопировать их обратно в массив

template <class T>
void TournamentSort (T a[], int n)
{
  DataNode<T> *tree;  // корень дерева
  DataNode<T> item;

  // минимальная степень двойки, большая или равная n
  int bottomRowSize;

  // число узлов в полном дереве, нижний ряд которого
  // имеет bottomRowSize узлов
  int treesize;

  // начальный индекс нижнего ряда узлов
  int loadindex;
  int i, j;

  // определить требуемый размер памяти для нижнего ряда узлов
  bottomRowSize = PowerOfTwo(n);

 // вычислить размер дерева и динамически создать его узлы
 treesize = 2 * bottomRowSize - 1;
 tree = new DataNode<T>[treesize];
 // скопировать массив в дерево объектов типа DataNode
j = 0;
for (i=loadindex; i<treesize; i++)
 {
   item.index = i;
   if (j < n)
   {
     item.active = 1;
     item.data = a[j++];
   }
   else
     item.active = 0;
   tree[i] = item;
 }

 // выполнить начальные сравнения для определения наименьшего элемента
 i = loadindex;
 while (i > 0)
 {
  j = i;
  while (j < 2*i);   // обработать пары    соревнующихся
  {
     // проведение матча. сравнить tree[j] с его     соперником tree[j+1]
     // скопировать победителя в      родительский узел
     if (!tree[j+1].active || tree[j] <      tree[j+1])
        tree[(j-1)/2] = tree[j];
     else
        tree[(j-1)/2] = tree[j+1];
     j += 2;         // перейти к следующей паре
  }
```

```
 // обработать оставшиеся n-1 элементов. скопировать победителя
 // из корня в массив. сделать победителя неактивным. обновить
 // дерево, разрешив сопернику победителя снова войти в турнир
 for (i=0; i<n-1; i++)
 {
  a[i] =    tree[0].data;
  tree[tree[0].index].active = 0;
  UpdateTree(tree, tree[0].index);
 }
 // скопировать  наибольшее значение в массив
 a[n-1] = tree[0].data;
}
```

В функцию UpdateTree передается индекс i, указывающий исходное положение наименьшего текущего элемента в нижнем ряду дерева. Это—удаляемый узел (становится неактивным). Значению, которое "проиграло" предварительный раунд последнему победителю (наименьшему значению), разрешается снова войти в турнир.

```
// параметр i есть начальный индекс текущего наименьшего элемента
// в списке (победителя турнира)
template <class T>
void UpdateTree(DataNode<T> *tree, int i)
{
  int j;

    // определить соперника победителя. позволить ему продолжить
    // турнир, копируя его в родительский узел.
    if (i % 2 == 0)
      tree [(i-1)/2] = tree[i-1]; // соперник – левый узел
    else
      tree [(i-1)/2] = tree[i+1]; // соперник – правый узел

        // переиграть те матчи, в которых принимал участие
        // только что исключенный из турнира игрок
        i = (i-1)/2;
        while (i > 0)
        {
          // соперником является правый или левый узел?
          if (i % 2 == 0)
            j = i-1;
          else
            j = i+1;
          // проверить, является ли соперник активным
          if (!tree[i].active || !tree[j].active)
            if (tree[i].active)
              tree[(i-1)/2] = tree[i];
            else
              tree[(i-1)/2] = tree[j];
          // устроить соревнование.
          // победителя скопировать в родительский узел
          else
            if (tree[i] < tree[j])
              tree[(i-1)/2] = tree[i];
            else
              tree[(i-1)/2] = tree[j];
          // перейти к следующему кругу соревнования (родительский уровень)
       i = (i-1)/2;
        }
        // Турнир с новым соперником закончен.
        // очередное наименьшее значение находится в корневом узле
      }
```

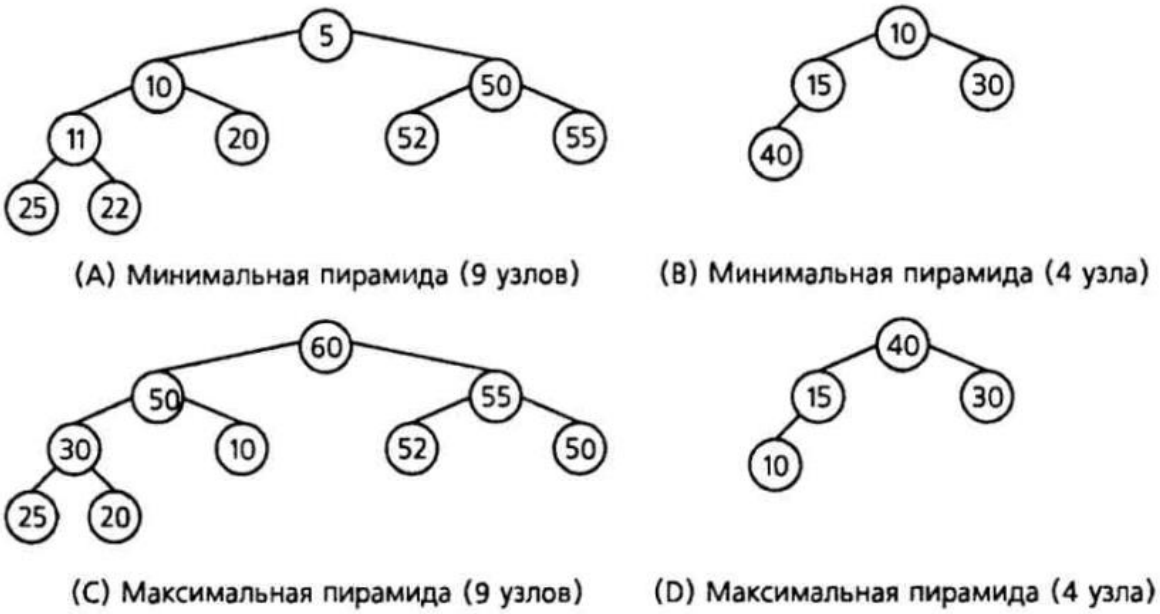
13.2. Пирамиды

Представляемые массивами деревья находят применение в имеющих большое значение приложениях с **пирамидами (heaps)**, являющимися законченными бинарными деревьями, имеющими упорядочение узлов по уровням. В **максимальной пирамиде (maximum heap)** родительский узел больше или равен каждому из своих сыновей. В **минимальной пирамиде (minimum heap)** родительский узел меньше или равен каждому из своих сыновей. Эти ситуации изображены на рис. 13.2. В максимальной пирамиде корень содержит наибольший элемент, а в минимальной — наименьший. В этой книге рассматриваются минимальные пирамиды.

Пирамида как список

Пирамида является списком, который хранит некоторый набор данных в виде бинарного дерева. **Пирамидальное упорядочение** предполагает, что каждый узел пирамиды содержит значение, которое меньше или равно значению любого из его сыновей. При таком упорядочении корень содержит наименьшее значение данных. Как абстрактная списковая структура пирамида допускает добавление и удаление элементов. Процесс включения не подразумевает, что новый

элемент занимает конкретное место, а лишь требует, чтобы поддерживалось пирамидальное упорядочение. Однако при удалении из списка выбрасывается наименьший элемент (корень). Пирамида используется в тех приложениях, где клиенту требуется прямой доступ к минимальному элементу. Как список пирамида не имеет операции поиска и осуществляет прямой доступ к минимальному элементу в режиме "только чтение". Все алгоритмы обработки пирамид сами должны обновлять дерево и поддерживать пирамидальное упорядочение.



(A) Минимальная пирамида (9 узлов) (B) Минимальная пирамида (4 узла)

(C) Максимальная пирамида (9 узлов) (D) Максимальная пирамида (4 узла)

Рис. 13.2. Максимальные и минимальные пирамиды

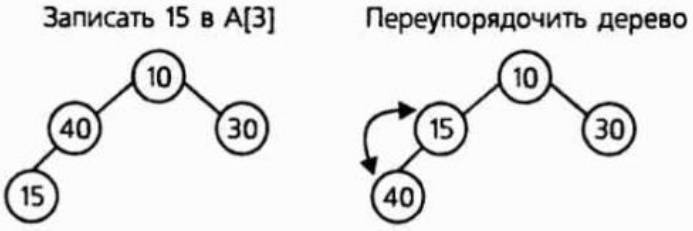
Пирамида является очень эффективной структурой управления списками, которая пользуется преимуществами полного бинарного дерева. При каждой операции включения или удаления пирамида восстанавливает свое упорядочение посредством сканирования только коротких путей от корня вниз до конца дерева. Важными приложениями пирамид являются очереди приоритетов и сортировка элементов списка. Вместо того чтобы использовать более медленные алгоритмы сортировки, можно включить элементы списка в пирамиду и отсортировать их, постоянно удаляя корневой узел. Это дает чрезвычайно быстрый алгоритм сортировки.

Обсудим внутреннюю организацию пирамиды в нашем классе Heap. Алгоритмы включения и исключения элементов представляются в реализации методов Insert и Delete. Пример 13.2 исследует пирамиды и иллюстрирует некоторые операции над ними.

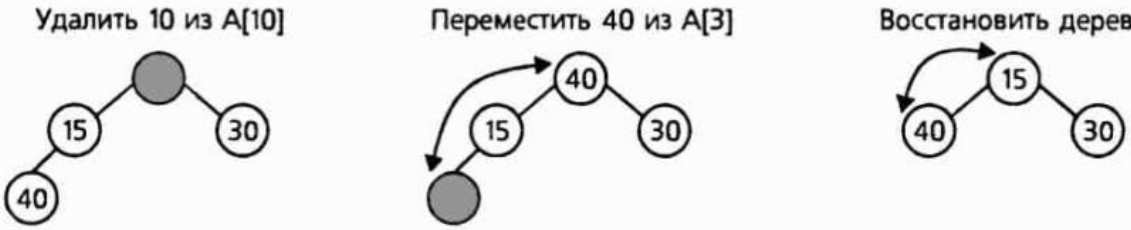
Пример 13.2

1. *Создание пирамиды.* Массив имеет соотвествующее представление в виде дерева. В общем случае это дерево не является пирамидой. Пирамида создается переупорядочением элементов массива.

2. *Вставка элемента.* Новый элемент добавляется в конец списка, а затем дерево реорганизуется с целью восстановления пирамидальной структуры. Например, для добавления в список числа 15 производятся следующие действия:



3. *Удаление элемента.* Удаляется всегда корень дерева (A[0]). Освободившееся место занимает последний элемент списка. Дерево реорганизуется с целью восстановления пирамидальной структуры. Например, для исключения числа 10 производятся следующие действия:



Класс Heap

Как и любой линейный или нелинейный список, класс пирамид имеет операции включения и исключения элементов, а также операции, которые возвращают информацию о состоянии объекта, например, размер списка.

Спецификация класса Heap

ОБЪЯВЛЕНИЕ

```
#include <iostream.h>
#include <stdlib.h>

template <class T>
class Heap
{
  private:
    // hlist указывает на массив, который может быть динамически создан
    // конструктором (inArray == 0) или передан как параметр (inArray == 1)
    T *hlist;
    int inArray;

    // максимальный и текущий размеры пирамиды
    int maxheapsize;
    int heapsize;       // определяет конец списка

    // функция вывода сообщений об ошибке
    void error(char errmsg[]);

    // утилиты восстановления пирамидальной структуры
    void FilterDown(int i);
    void FilterUp(int i);

  public:
    // конструкторы и деструктор
    Heap (int maxsize);           // создать пустую пирамиду
    Heap (T arr[], int n);        // преобразовать arr в пирамиду
    Heap (const Heap<T>& H);      // конструктор копий
    ~Heap(void);                  // деструктор

    // перегруженные операторы: "=", "[]", "T*"
    Heap<T> operator= (const Heap<T>& rhs);
    const T& operator[] (int i);

    // методы обработки списков
    int ListSize(void) const;
    int ListEmpty(void) const;
    int ListFull(void) const;
    void Insert(const T& item);
    T Delete(void);
    void ClearList(void);
};
```

ОПИСАНИЕ

Первый конструктор принимает параметр size и использует его для динамического выделения памяти под массив. В исходном состоянии пирамида пуста, и новые элементы включаются в нее с помощью метода Insert. Деструктор, конструктор копирования и оператор присваивания поддерживают использование динамической памяти. Второй конструктор принимает в качестве параметра массив и преобразует его в пирамиду. Таким образом, клиент может навязать пирамидальную структуру любому существующему массиву и воспользоваться свойствами пирамиды.

Перегруженный оператор индекса "[]" позволяет клиенту обращаться к объекту типа пирамиды как к массиву. Поскольку этот оператор возвращает ссылку на константу, доступ осуществляется лишь в режиме "только чтение".

Методы ListEmpty, ListSize и ListFull возвращают информацию о текущем состоянии пирамиды.

Метод Delete всегда исключает из пирамиды первый (наименьший) элемент. Метод Insert включает элемент в список и поддерживает пирамидальное упорядочение.

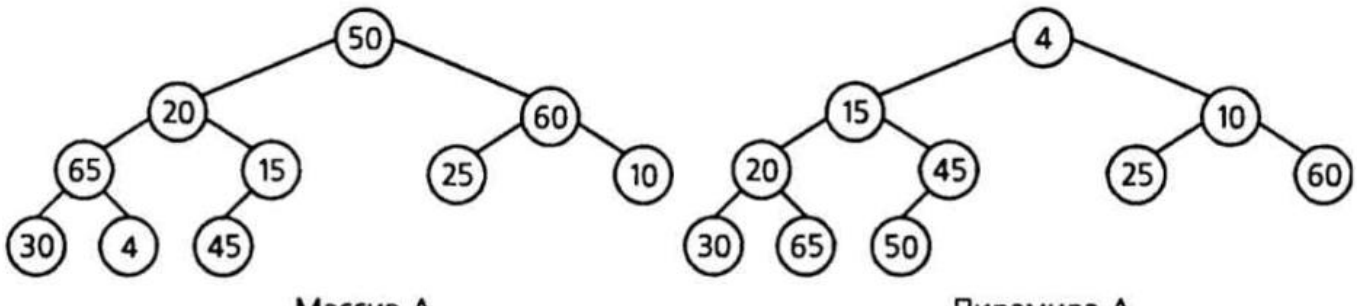
ПРИМЕР

```
Heap<int> H(4);                 // 4-элементная пирамида целых чисел
int A[] = {15, 10, 40, 30};     // 4-элементный массив
Heap<int> K(A, 4);              // преобразовать массив A в пирамиду K

H.Insert(85);                   // вставить 85 в пирамиду H
H.Insert(40);                   // вставить 40 в пирамиду H
cout << H.Delete();             // напечатать 40 — наименьший элемент в H

// распечатать массив, представляющий пирамиду A
for (int i=0; i<4; i++)
  cout << K[i] << " ";          // напечатать 10 15 40 30

K[0] = 99;                      // недопустимый оператор
```

Программа 13.1. Иллюстрация класса Heap

Эта программа начинается с инициализации массива A, а затем преобразует его в пирамиду.

A: 50, 20, 60, 65, 15, 25, 10, 30, 4, 45



Массив A Пирамида A

Элементы исключаются из пирамиды и распечатываются до тех пор, пока пирамида не опустеет. Поскольку пирамида реорганизуется после каждого исключения, элементы распечатываются по возрастанию.

```
#include <iostream.h>

#include "heap.h"

// распечатать массив, состоящий из n элементов
template <class T>
void PrintList (T A[], int n)
{
  for (int i=0; i<n; i++)
    cout << A[i] << "  ";
  cout << endl;
}

void main(void)
{
  // исходный массив
  int A[10] = {50, 20, 60, 65, 15, 25, 10, 30, 4, 45}

  cout << "Исходный массив:" << endl;
  PrintList(A, 10);

  // преобразование A в пирамиду
  heap<int> H(A,10);

  // распечатать новую версию массива A
  cout << "Пирамида:" << endl;
  PrintList(A, 10);

  cout << "Удаление элементов из пирамиды:" << endl;
  // непрерывно извлекать наименьшее значение
  while (!H.ListEmpty())
    cout << H.Delete() << "  ";
  cout << endl;
}
/*
<Прогон программы 13.1>

Исходный массив:
50  20  60  65  15  25  10  30  4  45
Пирамида:

4  15  10  20  45  25  60  30  65  50
Удаление элементов из пирамиды:
4  10  15  20  25  30  45  50  60  65
*/
```

Источники:

1. У. Топп, У. Форд – Структуры данных в C++
2. https://habr.com/ru/post/587228/