第9章 优 先 队 列

与第6章FIFO结构的队列不同，优先队列中元素出队列的顺序由元素的优先级决定。从优先队列中删除元素是根据优先权高或低的次序，而不是元素进入队列的次序。

可以利用堆数据结构来高效地实现优先队列。堆是一棵完全二叉树，可用 8.4节所介绍的公式化描述方法来高效存储完全二叉树。在高度和重量上取得平衡的左高树很适合于用来实现优先队列。本章的内容涵盖了堆和左高树。

在本章的应用部分，利用堆开发了一种复杂性为 $O(n\log n)$ 的排序算法，称为堆排序。在第2章所介绍的对 n 个元素进行排序的算法，其复杂性均为 $O(n^2)$。虽然第3章介绍的箱子排序和基数排序算法的运行时间为 $\Theta(n)$，但算法中元素的取值必须在合适的范围内。堆排序是迄今为止所讨论的第一种复杂性优于 $O(n^2)$ 的通用排序算法，第14章将讨论另一种与堆排序具有相同复杂性的排序算法。

从渐进复杂性的观点来看，堆排序是一种优化的排序算法，因为可以证明，任何通用的排序算法都是通过成对比较元素来获得 $\Omega(n\log n)$ 复杂性的（见14.4.2节）。

本节所考察的另外两个应用是机器调度和生成霍夫曼编码。机器调度问题属于 NP-复杂问题，对于这类问题不存在具有多项式时间复杂性的算法。而第 2章提到的大量事实表明，只有具有多项式时间复杂性的算法才是可行的，因此，经常利用近似算法或启发式算法来解决 NP-完全问题，这些算法能在合理的时间内完成，但并不能保证找到最佳结果。对于机器调度应用，利用堆数据结构获得了有效解决机器调度问题的近似算法。本章没有提供有关左高树的应用，其实6.4.4节中的工厂仿真在这方面是一个很好的应用。

9.1 引言

优先队列（priority queue）是0个或多个元素的集合，每个元素都有一个优先权或值，对优先队列执行的操作有 1) 查找；2) 插入一个新元素；3) 删除。在最小优先队列（min priority queue）中，查找操作用来搜索优先权最小的元素，删除操作用来删除该元素；对于最大优先队列（max priority queue），查找操作用来搜索优先权最大的元素，删除操作用来删除该元素。优先权队列中的元素可以有相同的优先权，查找与删除操作可根据任意优先权进行。

最大优先权队列的抽象数据类型描述如 ADT 9-1所示，最小优先队列的抽象数据类型描述与之类似，只需将最大改为最小即可。

ADT 9-1 最大优先队列的抽象数据类型描述

抽象数据类型 *MaxPriorityQueue*{

实例

有限的元素集合，每个元素都有一个优先权

操作

Create ()：创建一个空的优先队列

Size ()：返回队列中的元素数目

Max ()：返回具有最大优先权的元素

Insert (*x*)：将*x* 插入队列

DeleteMax (*x*)：从队列中删除具有最大优先权的元素，并将该元素返回至 *x*

}

例9-1 假设我们对机器服务进行收费。每个用户每次使用机器所付费用都是相同的，但每个用户所需要服务时间都不同。为获得最大利润，假设只要有用户机器就不会空闲，我们可以把等待使用该机器的用户组织成一个最小优先队列，优先权即为用户所需服务时间。当一个新的用户需要使用机器时，将他/她的请求加入优先队列。一旦机器可用，则为需要最少服务时间（即具有最高优先权）的用户提供服务。

如果每个用户所需时间相同，但用户愿意支付的费用不同，则可以用支付费用作为优先权，一旦机器可用，所交费用最多的用户可最先得到服务，这时就要选择最大优先队列。

例9-2 考察6.4.4节所介绍的工厂仿真问题，对其事件队列所执行的操作有：1）查找具有最小完成时间的机器；2）改变该机器的完成时间。假设我们构造一个最小优先队列，队列中的元素即为机器，元素的优先权为该机器的完成时间。最小优先队列的查找操作可用来返回具有最小完成时间的机器。为了修改此机器的完成时间，可以先从队列中删除具有最小优先权的元素，然后用新的完成时间作为该元素的优先权并将其插入队列。实际上，为了满足事件表的应用，可以在最小优先队列的ADT表中新增加一个操作，用来修改具有最小优先权元素的优先权。

最大优先队列也可用于工厂仿真问题。在6.4.4节中的仿真程序中，每台机器按先进先出的方式来完成等待服务的任务，因此可以为每台机器配置了一个 FIFO队列。但如果将服务规则改为“一旦机器可用，则从等待任务中选择优先权最大的任务进行处理”，每台机器就需要一个最大优先队列。每台机器执行的操作有：1）每当一个新任务到达，将其插入该机器的最大优先队列中；2）一旦机器可以开始运行一个新任务，将具有最大优先权的任务从该机器的队列中删除，并开始执行它。

当每个机器的服务规则如上述改变之后，则需用一个最小优先队列来表示仿真问题中的事件表，用一个最大优先队列来存储每台机器旁的等待任务。在6.4.4节的仿真模型中我们事先已经知道事件表的长度，即机器的台数，并且在整个仿真过程中事件表的长度不会改变，所以可采用一个公式化描述的优先权队列来表示时间表。但更常见的是必须考虑加入新机器或移走旧机器的情况，所以最好使用链表描述的优先队列，这样在队列建立时就不必事先预测队列的最大长度，也不必动态地改变数组的大小。

如同在6.4.4节中选择链表FIFO队列而不是公式化队列一样，每个机器的优先权队列也最好是链表队列。每个机器的队列长度在仿真过程中不断变化，而所有队列的长度之和却总是等于未完成的任务数之和。

本章提供了优先权队列有效的描述方法。鉴于最大优先队列与最小优先队列十分类似，故在此只明确给出了最大优先队列的描述。

9.2 线性表

描述最大优先队列最简单的方法是采用无序线性表。假设有一个具有 n 个元素的优先队列，如果利用公式（2-1），那么插入操作可以十分容易地在表的右端末尾执行，插入所需时间为 $\Theta(1)$。删除操作时必须查找优先权最大的元素，即在未排序的 n 个元素中查找具有最大优先权的元素，所以删除操作所需时间为 $\Theta(n)$。如果利用链表，插入操作在链头执行，时间为 $\Theta(1)$，

而每个删除操作所需时间为$\Theta(n)$。

另一种描述方法是采用有序线性表，当使用公式（ 2-1）时元素按递增次序排列，使用链表时则按递减次序排列，这两种描述方法的删除时间均为$\Theta(1)$，插入操作所需时间为$\Theta(n)$。

练习

1. 利用公式化描述的无序线性表，设计一个C++类来描述最大优先队列 (即利用程序3-1中的Linear List类)，使得插入时间为$\Theta(1)$，对于n个元素的队列删除操作所需的最大时间为$O(n)$。

2. 利用无序链表完成练习1（即利用程序3-8中的类Chain）。

3. 利用公式化描述的有序线性表重做练习 1，使得插入操作的时间为$O(n)$，删除操作的最大时间为$\Theta(1)$。

4. 利用程序7-1中的类ShortedChain重做练习1。

5. 给出抽象数据类型*MinPriorityQueue*的描述，并采用公式化描述的无序线性表，用 C++类设计相应的最小优先队列。

9.3 堆

9.3.1 定义

定义 [最大树（最小树）] 每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。

最大树（max tree）与最小树（min tree）的例子分别如图9-1、9-2所示，虽然这些树都是二叉树，但最大树不必是二叉树，最大树或最小树节点的子节点个数可以大于 2。

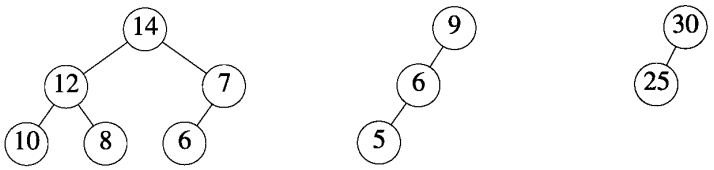

图9-1 最大树

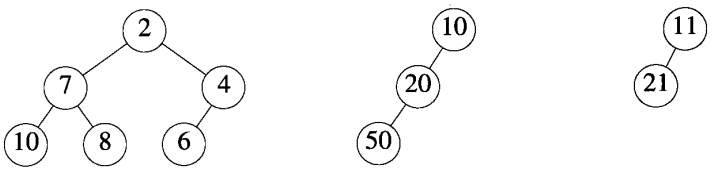

图9-2 最小树

定义 [最大堆（最小堆）] 最大（最小）的完全二叉树。

图9-1b 所示的最大树并不是最大堆（max heap），另外两个最大树是最大堆。图9-2b 所示的最小树不是最小堆（min heap），而另外两个是。

由于堆是完全二叉树，利用 8.4节所介绍的公式化描述方案，可用一维数组有效地描述堆，利用二叉树的特性 4（见8.3节）可将堆中的节点移到它的父节点或它的一个子节点处。在后面的讨论中将用节点在数组中的位置来指定堆中的节点，如根的位置为 1，其左孩子为 2，右孩子为 3，等等。另外，注意到堆是完全二叉树，拥有 n 个元素的堆其高度为 $[\log_2(n+1)]$，因此，如果可在 $O(height)$ 时间内完成插入和删除操作，则这些操作的复杂性为 $O(\log_2 n)$。

9.3.2 最大堆的插入

图9-3a 给出了一个具有 5个元素的最大堆。由于堆是完全二叉树，当加入一个元素形成6元素堆时，其结构必如 9-3b 所示。如果插入元素的值为 1，则插入后该元素成为 2的左孩子，相反，若新元素的值为 5，则该元素不能成为 2的左孩子（否则将改变最大树的特性），应把2下移为左孩子（如图 9-3c所示），同时还得决定在最大堆中 5是否占据2原来的位置。由于父元素20大于等于新插入的元素 5，因此可以在原 2所在位置插入新的元素。假设新元素的值为21而不是5，这时，同图 9-3c 一样，把2下移为左孩子，由于 21比父元素值大，所以 21不能插入原来 2所在位置，因此把 20移到它的右孩子所在位置， 21插入堆的根节点（如图 9-3d 所示）。

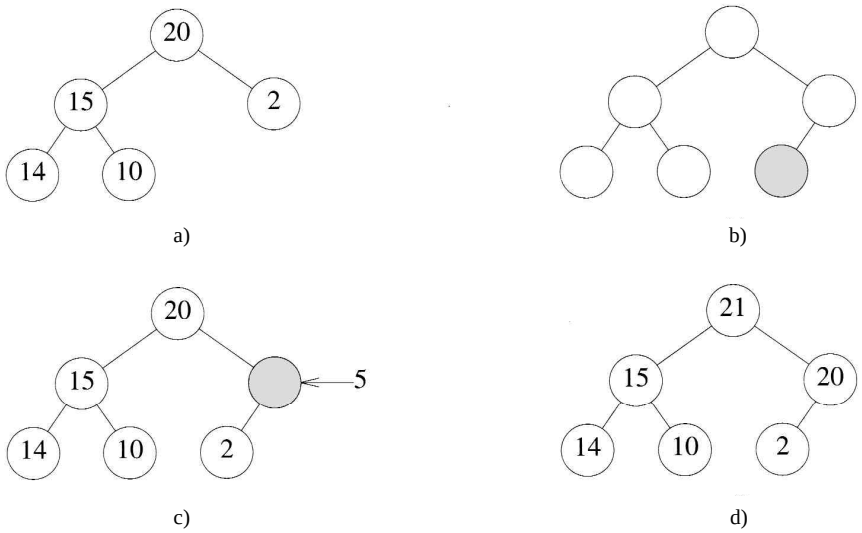

图9-3 最大堆的插入

插入策略从叶到根只有单一路径，每一层的工作需耗时 $\Theta(1)$，因此实现此策略的时间复杂性为$O(height)=O(\log_2 n)$

9.3.3 最大堆的删除

从最大堆中删除一个元素时，该元素从堆的根部移出。例如，对图 9-3d 的最大堆进行删除操作即是移去元素21，因此最大堆只剩下五个元素。此时，图 9-3d 中的二叉树需要重新构造，以便仍然为完全二叉树。为此可以移动位置6中的元素，即2。这样就得到了正确的结构（如图 9-4a所示），但此时根节点为空且元素 2不在堆中，如果2直接插入根节点，得到的二叉树不是

最大树，根节点的元素应为2、根的左孩子、根的右孩子三者中的最大值。这个值是 20，它被移到根节点，因此在位置3形成一个空位，由于这个位置没有孩子节点， 2可以插入，最后形成的最大堆如图9-3a 所示。

现在假设要删除20，在删除之后，堆的二叉树结构如图9-4b 所示，为得到这个结构，10从位置5移出，如果将10放在根节点，结果并不是最大堆。把根节点的两个孩子（15和2）中较大的一个移到根节点。假设将10插入位置2，结果仍不是最大堆。因此将14上移，10插入到位置4，最后结果如图9-4c 所示。

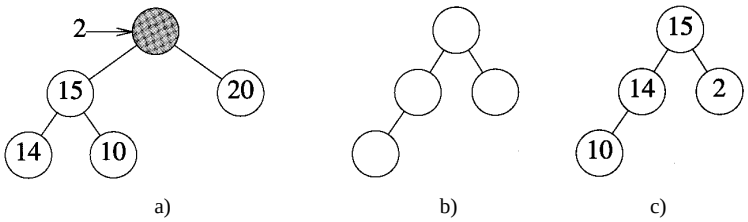

图9-4 最大堆的删除

删除策略产生了一条从堆的根节点到叶节点的单一路径，每层工作需耗时 $\Theta(1)$，因此实现此策略的时间复杂性为O(*height*)=O ($\log_2 n$)

9.3.4 最大堆的初始化

在最大堆的几种应用中，包括 6.4.4节中工厂仿真问题的事件表，开始时堆中已经含有 n ($n>0$) 个元素。我们可以通过在初始为空的堆中执行 n 次插入操作来构建非空的堆，插入操作所需总时间为O($n\log n$)，也可利用不同的策略在 $\Theta(n)$ 时间内完成堆的初始化。

假设开始数组a 中有n 个元素，另有n=10，a[1:10]中元素的关键值为[20，12，35，15，10，80，30，17，2，1]，这个数组可以用来表示如图 9-5a 所示的完全二叉树，这棵完全二叉树不是最大树。

为了将图9-5a 的完全二叉树转化为最大堆，从第一个具有孩子的节点开始（即节点 10），这个元素在数组中的位置为 $i=[n/2]$，如果以这个元素为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为堆。随后，继续检查以 $i-1$，$i-2$等节点为根的子树，直到检查到整个二叉树的根节点（其位置为1）。

下面对图 9-5a中的二叉树完成这一系列工作。最初， i=5，由于 10>1，所以以位置 i 为根的子树已是最大堆。下一步，检查根节点在位置 4的子树，由于 15<17，因此它不是最大堆，为将其变为最大堆，可将 15与17进行交换，得到的树如图 9-5b 所示。然后检查以位置 3为根的子树，为使其变为最大堆，将 80与35进行交换。之后，检查根位于位置 2 的子树，通过重建过程使该子树成为最大堆。将该子树重构为最大堆时需确定孩子中较大的一个，因为 12<17，所以17成为重构子树的根，下一步将 12与位置4的两个孩子中较大的一个进行比较，由于 12<15，15被移到位置 4，空位8没有孩子，将 12插入位置 8，形成的二叉树如图 9-5c。最后，检查位置 1，这时以位置 2或位置 3为根的子树已是最大堆了，然而 20 <（max[17,80]），因此80成为最大堆的根，当 80移入根，位置 3空出。由于 20<（max[35,30]），位置 3被35占据，最后 20占据位置 6。图 9-5d 显示了最终形成的最大堆。

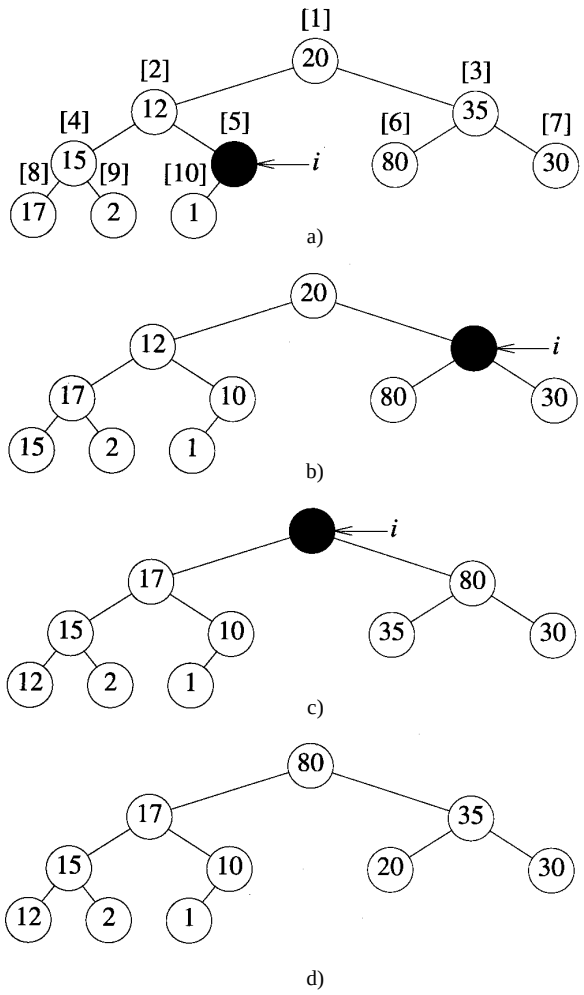

图9-5 最大堆的初始化

9.3.5 类MaxHeap

程序9-1给出了最大堆的类定义。n 是私有成员，代表目前堆中元素的个数；MaxSize是堆的最大容量；heap为存贮堆元素的数组，省缺堆的大小为10个元素。

程序9-1 类MaxHeap

```
template<class T>
class MaxHeap {
  public:
    MaxHeap(int MaxHeapSize = 10);
    ~MaxHeap() {delete [] heap;}
    int Size() const {return CurrentSize;}
    T Max() {if (CurrentSize == 0) throw OutOfBounds();
          return heap[1];}
    MaxHeap<T>& Insert(const T& x);
```

```
    MaxHeap<T>& DeleteMax(T& x);
    void Initialize(T a[], int size, int ArraySize);
  private:
    int CurrentSize, MaxSize;
    T *heap;  // 元素数组
};
```

堆的构造函数见程序9-2，构造函数只是简单地开辟一个足够大的数组使之能够存贮当元素个数达到最大值时的所有元素，它并不处理由new引发的NoMem异常。析构函数将删除此数组。size函数尤其简单，仅仅返回CurrentSize的值。另一个简单的函数是Max，如果最大堆为空，它将引发OutOf Bounds异常，如果不为空，则返回最大树的根的值。

程序9-2 MaxHeap的构造函数

```
template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{// 构造函数
  MaxSize = MaxHeapSize;
  heap = new T[MaxSize+1];
  CurrentSize = 0;
}
```

在Insert (见程序9-3)和DeleteMax (见程序9-4)的代码中假设已重载了关系操作符<、>和>=。这些代码反映了9.3.2与9.3.3节所讨论内容。

程序9-3 最大堆的插入

```
template<class T>
MaxHeap<T>& MaxHeap<T>::Insert(const T& x)
{// 把 x 插入到最大堆中
  if (CurrentSize == MaxSize)
    throw NoMem(); // 没有足够空间

  //为 x寻找应插入位置
  // i 从新的叶节点开始，并沿着树上升
  int i = ++CurrentSize;
  while (i != 1 && x > heap[i/2]) {
    // 不能够把 x 放入 heap[i]
    heap[i] = heap[i/2]; // 将元素下移
    i /= 2;            // 移向父节点
    }

  heap[i] = x;
  return *this;
}
```

在插入代码中，i 从新创建的叶节点位置CurrentSize开始，对从该位置到根的路径进行遍历。对于每个位置i，都要检查是否到达根（i =1）或在i 处插入新元素不会改变最大树的性质（x.key≤heap[i /2].key）。只要这两个条件中有一个满足，就可以在 i 处插入x，否则，将执行while 循环体，把位于i/2处的元素移到i 处并把i 处元素移到父节点（i/2）。对于一个具有*n* 个元

素的最大堆（即CarrentSize=n），while 循环的执行次数为O(*height*)=O(log*n*)，且每次执行所需时间为Θ(1)，因此Insert 的时间复杂性为O(log*n*)。

程序9-4 最大堆的删除

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{// 将最大元素放入 x，并从堆中删除最大元素
  // 检查堆是否为空
  if (CurrentSize == 0)
    throw OutOfBounds(); // 队列空

  x = heap[1]; // 最大元素

  // 重构堆
  T y = heap[CurrentSize--]; // 最后一个元素

  // 从根开始，为y 寻找合适的位置
  int i = 1,  // 堆的当前节点
      ci = 2; // i的孩子
  while (ci <= CurrentSize) {
    // heap[ci] 应是 i的较大的孩子
    if (ci < CurrentSize &&
        heap[ci] < heap[ci+1]) ci++;

    // 能把 y 放入heap[i]吗?
    if (y >= heap[ci]) break;   // 能

    // 不能
    heap[i] = heap[ci]; // 将孩子上移
    i = ci;                  //下移一层
    ci *= 2;
    }
  heap[i] = y;

  return *this;
}
```

在DeleteMax操作中，堆的根（即最大元素）heap[1]被保存到变量x中，堆的最后一个元素heap[CurrentSize]被保存到变量y中，堆的大小（CurrentSize）被减1。在while 循环中，开始查找一个合适的位置以便重新将 y插入。从根部开始沿堆向下查找，对于具有 *n* 个元素的堆，while 循环的执行次数为O(log*n*)，且每次执行所花时间为Θ(1)，因此，DeleteMax 操作总的时间复杂性为O(log*n*)。注意到即使堆的元素个数为0，代码也能正确执行，在这种情况下不执行While 循环，对堆的位置1进行赋值是多余的。

Initialize函数（见程序9-5）使用数组a中的元素对最大堆进行初始化。初始时删除私有成员heap当前所指的数组，并使 heap指向a[0]。size为数组a中的元素个数， ArraySize是假设从a[1]算起数组*a* 中所能容纳的最大元素个数。程序 9-5的最初4行代码重新设置了最大堆的私有成员，使数组a 代替数组heap。在for 循环中，从数组heap（即数组a）的二叉树表示中最后一

个具有一个孩子的节点开始进行初始化，直至到达根节点。对于每个位置 i，在while 循环中都保证根节点为i的子树已是最大堆。请注意for循环体与DeleteMax（见程序9-4）代码的相似性。

程序9-5 初始化一个非空最大堆

```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size, int ArraySize)
{// 把最大堆初始化为数组 a.
  delete [] heap;
  heap = a;
  CurrentSize = size;
  MaxSize = ArraySize;

  // 产生一个最大堆
  for (int i = CurrentSize/2; i >= 1; i--) {
    T y = heap[i]; // 子树的根

    // 寻找放置 y的位置
    int c = 2*i; // c的父节点是y的目标位置
    while (c <= CurrentSize) {
      // heap[c] 应是较大的同胞节点
      if (c < CurrentSize &&
          heap[c] < heap[c+1]) c++;

      // 能把 y 放入heap[c/2]吗?
      if (y >= heap[c]) break;  // 能

      // 不能
      heap[c/2] = heap[c]; // 将孩子上移
      c *= 2; // 下移一层
      }
    heap[c/2] = y;
    }
}
```

在Initialize 中将整个数组的元素放入一个类中，这样一来，当超出了最大堆的范围而又仍想访问数组a 中的元素时将会产生问题。为避免这个问题，在 MaxHeap 类的定义中加入一个共享成员函数Deactivate：

```
void Deactivate( ) {heap=0;}
```

通过使用该函数，可以防止在调用堆的析构函数时将数组 a 删除。

Initialize 函数的复杂性

如果元素个数为n，Initialize 函数（见程序9-5）中for 循环每次所花时间为$O(\log n)$，循环次数为$n/2$，所以总的复杂性为 $O(n\log n)$。注意符号 O代表算法复杂性的上限。实际应用中，Initialize 的复杂性要比$O(n\log n)$ 好一些。经过更仔细的分析，我们发现其真正的复杂性为$\Theta(n)$。

Initialize 的while 循环每次所花费时间为$O(h_i)$，其中h_i 是以i 位置为根节点的子树的高度。完全二叉树a[1:n]的高度为$h=\lceil \log_2 (n+1) \rceil$。在第$j$ 层最多含有2^{j-1} 个节点，因此最多有2^{j-1} 个节点具有相同的高度$h_i = h-j+1$，所以最大堆的初始化时间为：

$$O(\sum_{j=1}^{h-1} 2^{j-1}(h-j+1)) = O(\sum_{k=1}^{h-1} k2^{h-k}) = O(2^h \sum_{k=1}^{h-1} (k/2^k)) = O(2^h) = O(n)$$

由于for 循环执行了$n/2$次，其复杂性为$\Omega(n)$。将两者综合考虑，可以得到 Initialize的复杂性为$\Theta(n)$。

练习

6. 在MaxHeap类中加入共享成员函数 IsEmpty和IsFull，前者当且仅当最大堆为空时返回true。后者当且仅当最大堆已满时返回true。

7. 模仿最大堆的类定义构造一个最小堆的定义，并测试代码的正确性。

8. 在MaxHeap类中加入一个共享成员函数ChangeMax(x)，将当前最大元素改为元素x，x的值可以大于或小于当前最大元素的值，与删除最大元素的操作一样，代码沿着从根开始的一条向下的路径遍历。执行过程中当最大堆为空时，引发一个 OutOfBounds异常，代码的复杂性应为O(log$n$)，其中n是最大堆中元素个数，证明这个结论。

9. 由于在删除操作（见程序9-4）中重新插入最大堆的元素y是从堆的底部移出的那个元素，我们期望它仍插在接近底部的地方。重新写一个 DeleteMax函数，让根节点的空位置先移到叶节点，然后y通过这个叶节点往上找到它的合适位置。通过实验比较新代码是否比旧代码执行速度快。

10. 根据假设：

1) 在创建堆时，提供两个元素MaxElement和MinElement，堆中没有元素比MaxElement大，也没有元素比MinElement小。

2) 一个具有n 个元素的堆需要一个数组heap[0:2n+1]。

3) n 个元素，如本节所描述的那样存贮在heap[1:n]中。

4) MaxElement存贮在heap[0]中。

5) MinElement存贮在heap[n+1:2n+1]中。

重写MaxHeap中的成员函数。这些假设可使Insert和Delete操作得以简化。将本练习与本节中的实现作一对比。

9.4 左高树

9.4.1 高度与宽度优先的最大及最小左高树

9.3节的堆结构是一种隐式数据结构（implicit data structure），用完全二叉树表示的堆在数组中是隐式存贮的（即没有明确的指针或其他数据能够重构这种结构）。由于没有存贮结构信息，这种描述方法空间利用率很高，事实上没有空间浪费。尽管堆结构的时间和空间效率都很高，但它不适合于所有优先队列的应用，尤其是当需要合并两个优先队列或多个长度不同的队列时。因此需要借助于其他数据结构来实现这类应用，左高树（ leftist tree）就能满足这种要求。

考察一棵二叉树，它有一类特殊的节点叫做外部节点（external node），用来代替树中的空子树，其余节点叫做内部节点（ internal node）。增加了外部节点的二叉树被称为扩充二叉树（extended binary tree），图9-6a 给出了一棵二叉树，其相应的扩充二叉树如图 9-6b 所示，外部节点用阴影框表示，为了方便起见，这些节点用 a～f标注。

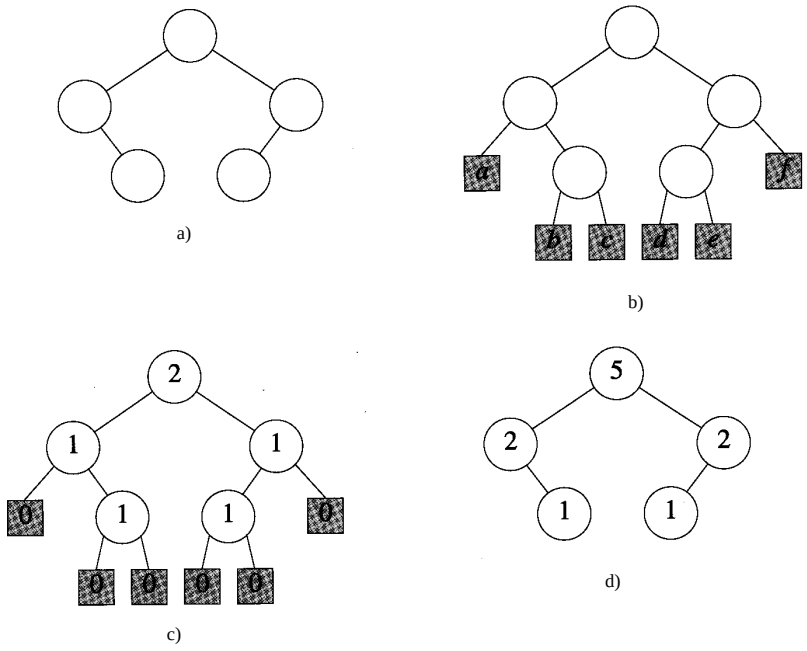

图9-6 s 和w 的值

a) 一棵二叉树 b) 扩充二叉树 c) s 的值 d) w的值

令$s(x)$ 为从节点x 到它的子树的外部节点的所有路径中最短的一条，根据 $s(x)$的定义可知，若x 是外部节点，则s 的值为0，若x 为内部节点，则它的s 值是：

$$\min\{s(L), s(R)\}+1$$

其中L与R分别为x 的左右孩子。扩充二叉树（如图9-6b所示）各节点的s 值如图9-6c 所示。

定义 **[高度优先左高树]** 当且仅当一棵二叉树的任何一个内部节点，其左孩子的 s 值大于等于右孩子的s 值时，该二叉树为高度优先左高树（height-biased leftist tree, HBLT）。

图9-6a 所示的二叉树不是HBLT。考察外部节点a 的父节点，它的左孩子的s 值为0，而右孩子的为1，所有其他内部节点均满足HBLT的定义。因此，若将图9-6a 中节点a 的父节点的左右子树进行交换，所得到的二叉树即为HBLT。

定理9-1 令x 为一个HBLT的内部节点，则

1) 以x 为根的子树的节点数目至少为$2^{s(x)}-1$。

2) 若子树x 有m 个节点，$s(x)$ 最多为$\log_2(m+1)$。

3) 通过最右路径（即，此路径是从x 开始沿右孩子移动）从x 到达外部节点的路径长度为$s(x)$。

证明 根据$s(x)$ 的定义可知，从x节点往下第$s(x)-1$层没有外部节点（否则 x 的s 值将更小）。以x为根的子树在当前层只有一个节点x，下一层有两个，再下一层有四个，……，x 层往下第$s(x)-1$层有个$2^{s(x)-1}$，在$s(x)-1$层以下可能还有其他节点，因此子树x的节点数目至少为$\sum_{i=0}^{s(x)-1} 2^i = 2^{s(x)}-1$。从1) 可以推出2)。根据$s$ 的定义以及HBLT一个节点的左孩子的s 值总是大于等于其右孩子的s 值，可以推得3）成立。

定义 [**最大**HBLT] 即同时又是最大树的HBLT；[**最小**HBLT] 即同时又是最小树的HBLT。

图9-1的最大树及图9-2的最小树都是HBLT。因此，图9-1的树是最大HBLT，图9-2中的树是最小HBLT。最大优先队列可以用最大HBLT表示，最小优先队列可用最小HBLT表示。

可以通过考察子树的节点数目而不是从根到外部节点的路径来得到另一类左高树。定义 x 的重量 $w(x)$ 为以 x 为根的子树的内部节点数目。注意到若 x 是外部节点，则其重量为 0；若 x 为内部节点，其重量为其孩子节点的重量之和加 1，图9-6a 中二叉树各节点的重量如图 9-6d 所示。

定义 [**重量优先左高树**] 当且仅当一棵二叉树的任何一个内部节点，其左孩子的 w 值大于等于右孩子的 w 值时，该二叉树为重量优先左高树（weight-biased leftist tree, WBLT）；[**最大（小）**WBLT] 即同时又是最大（小）树的WBLT。

同HBLT类似，具有 m 个节点的WBLT的最右路径长度最多为 $\log_2(m+1)$。可以对WBLT与HBLT执行优先队列的查找、插入、删除操作，其时间复杂性与堆的相应操作相同。同堆一样，WBLT与HBLT可在线性时间内完成初始化。用 WBLT或HBLT描述的两个优先队列可在对数时间内合并为一个，而当用堆描述优先队列时，则不能在对数时间内完成合并。

WBLT中的查找、插入、删除、合并和初始化操作与 HBLT中的相应操作很相似，因此，以下仅介绍有关HBLT的操作。WBLT的操作将留做练习（练习12）。

9.4.2 最大HBLT的插入

最大HBLT的插入操作可借助于最大 HBLT的合并操作来完成。假设将元素 x 插入到名为 H 的最大HBLT中，如果建造一棵仅有一个元素 x 的最大HBLT然后将它与 H 进行合并，结果得到的最大HBLT将包括 H 中的全部元素及元素 x。因此插入操作只需先建立一棵仅包含欲插入元素的HBLT，然后将它与原来的HBLT合并即可。

9.4.3 最大HBLT的删除

根是最大元素，如果根被删除，将留下分别以其左右孩子为根的两棵 HBLT的子树。将这两棵最大HBLT合并到一起，便得到包含除删除元素外所有元素的最大 HBLT，所以删除操作可以通过删除根元素并对两个子树进行合并来实现。

9.4.4 合并两棵最大HBLT

具有 n 个元素的最大 HBLT，其最右路径的长度为 $O(\log n)$。合并操作仅需遍历欲合并的HBLT的最右路径。由于在两条最右路径的每个节点上只需耗时 O(1)，因此将两棵HBLT进行合并具有对数复杂性。通过以上观察，在我们所设计的合并算法中，仅需移动右孩子。

合并策略最好用递归来实现。令 A、B 为需要合并的两棵最大HBLT，如果其中一个为空，则将另一个作为合并的结果，因此可以假设两者均不为空。为实现合并，先比较两个根元素，较大者作为合并后的HBLT的根。假定 A 具有较大的根，且其左子树为 L，C 是由 A 的右子树与 B 合并而成的HBLT。A 与 B 合并所得结果即是以 A 的根为根，L 与 C 为左右子树的最大HBLT。如果 L 的 s 值小于 C 的 s 值，则 C 为左子树，否则 L 为左子树。

例9-3 考察图9-7a 所示的两棵最大HBLT。每个节点的 s 值显示在节点的外侧，元素的值标在

节点内部。图中总是将要合并的两棵最大 HBLT中具有较大根的树画在左边。根据这种约定，左边HBLT的根总是最后所得到的HBLT的根。此外，右边HBLT的根用阴影来标出。

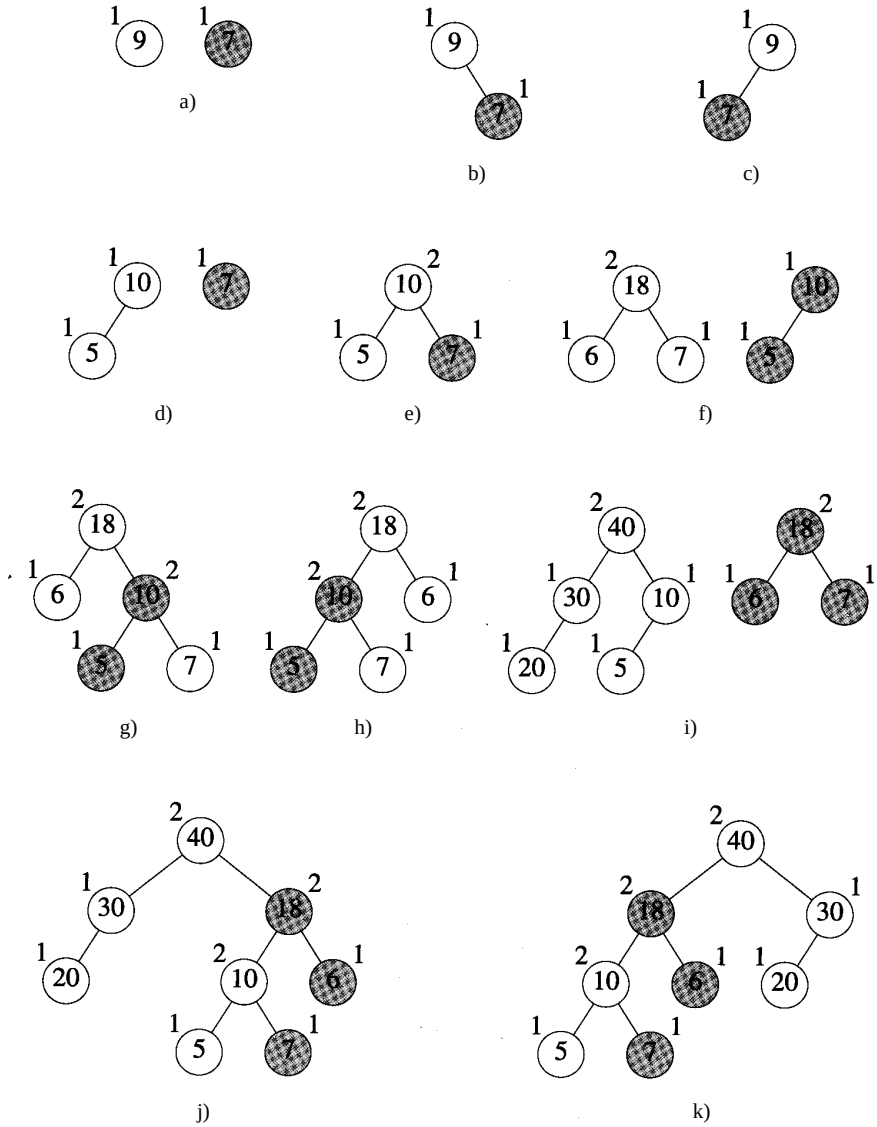

图9-7 最大HBLT的合并

因为9的右子树为空，因此将9的右子树与以7为根的HBLT进行合并，得到的仍是以7为根的树，把这棵树暂时作为9的右子树，可以得到如图9-7b 所示的最大树。由于此时9的左子树s值为0而右子树s 值为1，因此将其左右子树进行交换，得到如图9-7c 所示的最大HBLT。

下面来考察合并图9-7d 所示的两棵最大HBLT。毫无疑问，左子树的根将作为最终的根。当10的右子树与根为7的HBLT进行合并时，所得结果仍为后者。如果把这棵 HBLT作为10的右子树，可得到图9-7e 所示的最大树。比较10的左右孩子的*s* 值，发现没有必要再进行交换。

现在考察合并图9-7f 中的两棵最大HBLT。左子树的根无疑将作为最终的根。首先把 18的

右子树与以10为根的最大HBLT进行合并，其过程与图9-7d 中的情形完全一致，合并的结果为图9-7e 所示的最大HBLT。把这棵树作为18的右子树，即可得到图9-7f 所示的最大树。比较18的左右子树的s 值，可知这两棵子树必须交换，交换后所得结果如图9-7h 所示。

作为最后一个例子，来合并图9-7i 中的两棵最大HBLT。同前面的例子一样，左子树的根将成为最终的根。首先把40的右子树与根为18的最大HBLT进行合并，这个过程与图9-7f 中的合并过程完全一致，合并的结果为图 9-7g 所示的最大HBLT。把图9-7g 中的最大HBLT作为40的右子树，即可得到图9-7j 所示的最大树。由于40的左子树的s 值比其右子树的s 值小，因此将这两棵子树进行交换，最后得到图 9-7k 所示的最大HBLT。注意到在合并图9-7i 中的最大HBLT时，先移动到40的右孩子，再移动到18的左孩子，最后移动到10的右孩子，因此所有移动都是按照当前最大HBLT的最右路径进行的。

9.4.5 初始化最大HBLT

通过将n 个元素插入到最初为空的最大 HBLT中来对其进行初始化，所需时间为 O(log$n$)。为得到具有线性时间的初始化算法，首先创建n 个最大HBLT，每个树中仅包含n 个元素中的某一个，这n 棵树排成一个FIFO队列，然后从队列中依次删除两个 HBLT，将其合并，然后再加入队列末尾，直到最后只有一棵HBLT。

例9-4 我们希望构造具有五个元素：7,1,9,11,2的一棵最大HBLT。为此，首先构造五个单元素的最大HBLT，并形成一个FIFO队列。把最前面的两个最大 HBLT 7和1从队列中删除并进行合并，所得结果如图9-8a 所示，将该结果加入到队列中。接下来从队列中删除两棵最大 HBLT 9和11并进行合并，所得结果如图9-8b 所示，也将该结果加入队列中。现在继续从队列中删除最大HBLT 2及图9-8a 所得到的HBLT，并进行合并，所得结果（如图9-8c所示）加入队列。下一对从队列中删除的最大 HBLT如图9-8b 与9-8c 所示，经合并得到的最大 HBLT如图9-8d 所示，将该结果插入到队列中。至此，队列中只有一棵最大 HBLT，初始化工作完成。

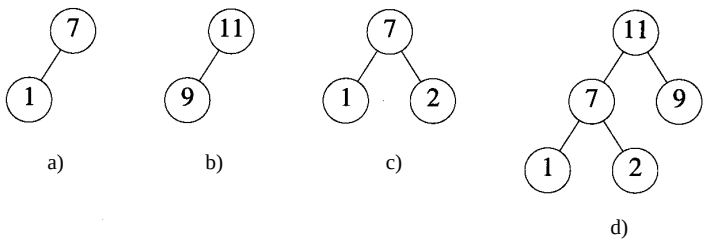

图9-8 最大HBLT的初始化

9.4.6 类MaxHBLT

最大HBLT的每个节点均需要 data、LeftChild、RightChild和s四个域，相应的节点类为HBLTNode（见程序9-6）。在HBLTNode的构造函数中要求提供 data和s，同时将LeftChild和RightChild置为0。

程序9-6 HBLT的节点类

```
template <class T>
```

```
class HBLTNode {
  friend MaxHBLT<T>;
  public:
    HBLTNode(const T& e, const int sh)
     {data = e;
      s = sh;
      LeftChild = RightChild = 0;}
  private:
    int s; // 节点的s 值
    T data;
    HBLTNode<T> *LeftChild, *RightChild;
};
```

最大HBLT可用程序9-7中的MaxHBLT类来实现，MaxHBLT类的每个对象都有一个唯一的私有成员root，用来指向最大HBLT的根。构造函数在初始化时将root置为0，因此初始的最大HBLT为空。析构函数通过调用私有成员函数Free来删除HBLT中的所有节点。Free函数按后序遍历整棵HBLT，每访问一个节点就删除该节点。Free的代码与8.9节中二叉树的私有成员函数Free的代码相同。

程序9-7 MaxHBLT类

```
template<class T>
class MaxHBLT {
  public:
    MaxHBLT() {root = 0;}
    ~MaxHBLT() {Free(root);}
    T Max() {if (!root) throw OutOfBounds();
          return root->data;}
    MaxHBLT<T>& Insert(const T& x);
    MaxHBLT<T>& DeleteMax(T& x);
    MaxHBLT<T>& Meld(MaxHBLT<T>& x) {
          Meld(root,x.root);
          x.root = 0;
          return *this;}
    void Initialize(T a[], int n);
  private:
    void Free(HBLTNode<T> *t);
    void Meld(HBLTNode<T>* &x, HBLTNode<T>* y);
    HBLTNode<T> *root;  // 指向树根的指针
};
```

由于插入、删除最大元素及初始化操作都需要使用合并操作，因此首先考察合并操作。共享成员函数Meld用于合并两棵最大HBLT，不过它是通过调用私有成员函数Meld来完成合并操作的。私有成员函数Meld用来实际完成合并任务，它带有两个参数x 和y，二者分别指向欲合并的最大HBLT的根，合并所得到的HBLT的根保存在x 中。在共享函数Meld中，当root与x.root合并完成之后，该函数将x 的根域置为0（这是为了防止x 的原节点被意外删除），并返回所得到的合并树的引用。

程序9-8给出了私有函数Meld。该函数首先处理要合并的树中至少有一个为空的特殊情况。当没有空树时要确保x 指向根值较大的树，如果x 不是指向根值较大的树，则将x与y的指针进行交换。接下来把x 的右子树与以y为根的最大HBLT进行递归合并。合并后为保证整棵树为最大HBLT，x 的左右孩子可能需要交换，这是通过计算x 的s 值来确定的。

程序9-8 合并两棵左高树

```
template<class T>
void MaxHBLT<T>::Meld(HBLTNode<T>* &x, HBLTNode<T>* y)
{// 合并两棵根分别为*x和 *y的左高树
 // 返回指向新根 x的指针
  if (!y) return; // y 为空
  if (!x) // x为空
    {x = y;
     return;}

  // x和y 均为空
  if (x->data < y->data) Swap(x,y);
  // 现在 x->data >= y->data
  Meld(x->RightChild,y);
  if (!x->LeftChild) {// 左子树为空
       // 交换子树
       x->LeftChild = x->RightChild;
       x->RightChild = 0;
       x->s = 1;}
  else {// 检查是否需要交换子树
       if (x->LeftChild->s < x->RightChild->s)
         Swap(x->LeftChild,x->RightChild);
       x->s = x->RightChild->s + 1;}
}
```

为了将元素x 插入到一棵最大HBLT中，程序9-9中的代码先产生一棵只有一个元素x的最大HBLT，然后通过私有成员函数 Meld将此树与欲插入的最大 HBLT进行合并。函数将返回一个指向所得最大HBLT的指针。该函数并不捕获可能由new产生的异常。

程序9-9 最大HBLT的插入

```
template<class T>
MaxHBLT<T>& MaxHBLT<T>::Insert(const T& x)
{// 把 x 插入到左高树中
 // 创建带有一个节点的树
  HBLTNode<T> *q = new HBLTNode<T> (x,1);
  // 把q与原树进行合并
  Meld(root,q);
  return *this;
}
```

在DeleteMax代码（见程序9-10）中，若最大HBLT为空，则引发OutOfBounds异常；若不为空，则将其左右子树的根分别保存在指针L与R中，然后将根删除，并把子树L和 R合并。

程序9-10 从最大HBLT中删除最大元素

```
template<class T>
MaxHBLT<T>& MaxHBLT<T>::DeleteMax(T& x)
{// 删除最大元素，并将其放入x
  if (!root) throw OutOfBounds();

  // 树不为空
  x = root->data;  // 最大元素
  HBLTNode<T> *L = root->LeftChild;
  HBLTNode<T> *R = root->RightChild;
  delete root;
  root = L;
  Meld(root,R);
  return *this;
}
```

最大HBLT的初始化代码见程序9-11。代码中利用一个公式化FIFO队列来保存初始化过程中所产生的最大HBLT。在第一个for循环中，产生了n 个仅含一个元素的最大HBLT并依次加入最初为空的队列。在下一个for 循环中，每次从队列中删除两个最大HBLT，将其合并，并把结果加入队列中。当for 循环结束时，队列中仅含一棵有n 个元素的最大HBLT（假设n>1）。

程序9-11 最大HBLT的初始化

```
template<class T>
void MaxHBLT<T>::Initialize(T a[], int n)
{// 初始化有n个元素的 HBLT 树
  Queue<HBLTNode<T> *> Q(n);
  Free(root);  // 删除老节点
  // 对树的队列进行初始化
  for (int i = 1; i <= n; i++) {
    // 创建只有一个节点的树
    HBLTNode<T> *q = new HBLTNode<T> (a[i],1);
    Q.Add(q);
    }

  // 不断合并队列中的树
  HBLTNode<T> *b, *c;
  for (i = 1; i <= n - 1; i++) {
    // 删除并合并两棵树
    Q.Delete(b).Delete(c);
    Meld(b,c);
    // 把合并后所得到的树放入队列
    Q.Add(b);
    }

  if (n) Q.Delete(root);
}
```

复杂性分析

构造函数需耗时$\Theta(1)$，而析构函数需耗时$\Theta(n)$，其中n为欲删除的最大HBLT中的元素个数。Max函数的复杂性为$\Theta(1)$。Insert、DeleteMax及共享成员函数Meld的复杂性与私有成员函数Meld的复杂性相同。由于私有成员函数Meld仅在以*x和*y为根的树的右子树中移动，因此该函数的复杂性为O(x->s+y->s)。由于*x与*y的最大s值分别为$\log_2(m+1)$与$\log_2(n+1)$，其中m与n分别是以*x和*y为根的最大HBLT中的元素个数，所以私有成员函数Meld的复杂性为O(logmn)。

为了分析Initialize的复杂性，为简单起见，假设n是2的幂次方。首先合并n/2对具有一个元素的最大HBLT，然后合并$n/4$个含有两个元素的最大HBLT，继而合并$n/8$个含有4个元素的最大HBLT，…。由于合并两棵含2^i个元素的最大HBLT需耗时O(i+1)，因此Initialize所花费的总时间为：

$$O(n/2 + 2*(n/4) + 3*(n/8) + \cdots) = O(n\sum\frac{i}{2^i}) = O(n)$$

练习

11. 写出MinHBLT类的代码，该类与MaxHBLT类的差别仅在于原来最大HBLT中的类成员现在是最小HBLT的成员，用Min与DeleteMin操作来代替Max与DeleteMax操作。

12. 1) 图9-1中哪些（如果有）二叉树是WBLT？

2) 令x为WBLT中的一个节点，对$w(x)$进行归纳，来证明从x出发到达一个外部节点的最右路径的最大长度为$\log_2(w(x)+1)$。

3) 用WBLTNode类来描述重量优先左高树的节点，每个类成员具有 w（重量）、data、LeftChild和RightChild域。

4) 设计类MaxWBLT来描述最大WBLT。类中应该包括Max、Insert、 DeleteMax、Meld及Initialize函数，这些函数分别对应于最大HBLT中的相应函数。代码中每个函数应与MaxHBLT中的相应函数具有相同的复杂性。用非递归的代码实现私有成员函数Meld。注意由于每个节点的w值可在向下移动的过程中计算出来，因此在使用WBLT时，自底向上的递归展开过程（见程序9-8）是多余的，但若使用HBLT则是必须的。

5) 试比较用最大WBLT、最大HBLT及最大堆来实现最大优先队列时各自的优缺点。

13. 在描述一棵最大HBLT时，可采用一个指向值为MinElement节点的指针（见练习10）来代替NULL（或0）指针。根据这一变化来修改最大HBLT的代码。新代码是否比原代码执行得更快？

9.5 应用

9.5.1 堆排序

你也许已注意到可利用堆来实现n个元素的排序，所需时间为O($n$log$n$)。可先将要排序的n个元素初始化为一个最大堆，然后每次从堆中提取（即删除）元素。各元素将按递减次序排列。初始化所需要的时间为$\Theta(n)$，每次删除所需要的时间为O(log$n$)，因此总时间为O($n$log$n$)，这个时间要比第2章中的排序算法（时间为$O(n^2)$）好得多。

上述的排序策略称为堆排序，其实现代码见程序9-12。

程序9-12 堆排序

```
template <class T>
void HeapSort(T a[], int n)
{// 利用堆排序算法对 a[1:n] 进行排序
  // 创建一个最大堆
  MaxHeap<T> H(1);
  H.Initialize(a,n,n);

  // 从最大堆中逐个抽取元素
  T x;
  for (int i = n-1; i >= 1; i--) {
    H.DeleteMax(x);
    a[i+1] = x;
    }

  // 在堆的析构函数中保存数组 a
  H.Deactivate();
}
```

图9-9给出了程序9-12的for 循环中最初几个i 值的运行过程，循环开始时的最大堆如图9-5d所示。

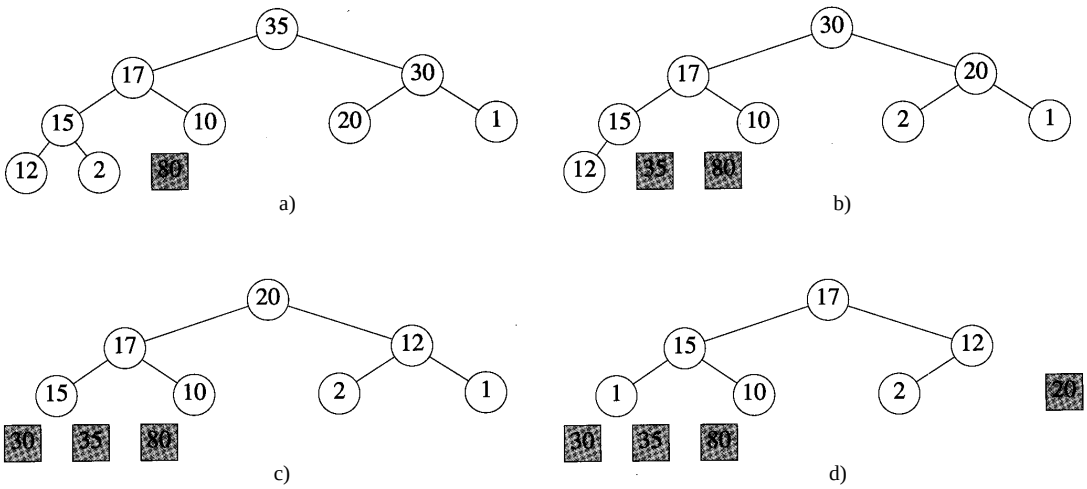

图9-9 堆排序

9.5.2 机器调度

考察一个机械厂，其中有 m 台一模一样的机器。现有 n 个作业需要处理，设作业 i 的处理时间为 t_i，这个时间为从将作业放入机器直到从机器上取下作业的时间。所谓调度（ schedule）是指按作业在机器上的运行时间对作业进行分配，使得：

- 一台机器在同一时间内只能处理一个作业。
- 一个作业不能同时在两台机器上处理。
- 作业 i 一旦运行，则需要 t_i 个时间单位。

假设每台机器在0时刻都是可用的，完成时间（或调度长度）是指完成所有作业的时间。在一个非抢先调度（nonpreemptire schedule）中，作业从s_i时刻起在某台机器上进行处理，其完成时刻为s_i+t_i，这里仅考虑非抢先调度。

图9-10给出了七个作业在三台机器上进行调度的情形，七个作业所需时间分别为（ 2, 14, 4, 16, 6, 5, 3）。三台机器分别被编号为M1、M2 和M3。每个阴影区代表作业的运行区间，阴影区的编号代表作业的索引号。作业4 在0 到16 时刻被调度到机器1（M1）上运行，在这16个时间单位中，机器1完成了对作业4的处理。作业2在0到14时刻被调度到机器2上处理，之后机器2在14到17时刻处理作业7。在机器3上，作业5在0～6时刻内完成，作业6在6～8时刻内完成，作业3在11～15时刻内完成，作业1在15～17时刻内完成。注意到每个作业只能在一台机器上s_i从时刻到s_i+t_i时刻内完成且任何机器在同一时刻仅能处理一个作业，完成所有作业的时刻为 17，因此完成时间或调度长度为17。

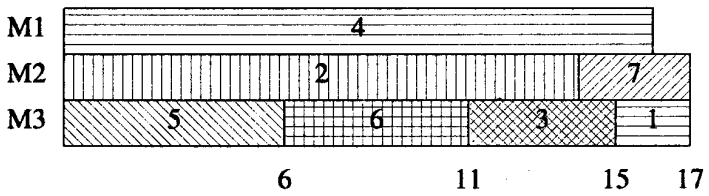

图9-10 三台机器的调度

我们的任务是写一个程序，以便确定如何进行调度才能使在 m 台机器上执行给定的n 个作业时所需要的处理时间最短。建立这种调度非常难。实际上，没有人能够设计一个具有多项式时间复杂性的算法（即一个复杂性为 $O(n^k m^l)$ 的算法，k 和l 为常数）来解决最小调度时间问题。

调度问题是著名的NP-复杂问题（NP表示nondeterministic polynornial) 中的一种。NP-复杂及NP-完全问题是指尚未找到具有多项式时间复杂性算法的问题。 NP-完全问题是一类判断问题，也就是说，这类问题的答案为是或否。机器调度问题不是一个判断题，因为它的答案是给出作业在机器间的分配方案（使完成时间最少）。可以构造另一类机器调度问题，除了给定任务和机器外，还给定了时间 TMin，这类问题要求确定是否存在一种调度仅需 TMin或更少的时间，这种问题应属于判断问题，因而是 NP-完全问题。NP-复杂问题可以是判断问题，也可以不是判断问题。

成千上万的具有实际意义的问题都是NP-复杂或NP-完全问题，如果有人能对一个NP-复杂或NP-完全问题找到一个多项式算法，那么他 /她同时也找到了能在多项时间内解决所有 NP-复杂或NP-完全问题的方法。虽然不能证明 NP-完全问题不能在多项式时间内得到解决，但大家都认为这已是一个事实。因此， NP-复杂问题的优化问题经常用近似算法（ approximation algorithms）解决，虽然近似算法不能保证得到最优解，但能保证所获得的是近似最优解。

在调度问题中，采用了一个称为最长处理时间优先（longest processing time first, LPT）的简单调度策略，它可以帮助我们获得一个较理想的调度长度，该长度为最优调度长度的 $4/3-1/(3m)$。在LPT算法中，作业按它们所需处理时间的递减顺序排列。在分配一个作业时，总是将其分配给最先变为空闲的机器。

如图9-10的例子，为了获得一个LPT调度，首先根据作业所需处理时间把作业按递减次序排列，所得排列结果为（4,2,5,6,3,7,1）。首先为作业4分配机器，由于三台机器均是从0时刻开

始可用，作业4可分配给任何一台。假设将其分配给机器 1，则机器1直到第16时刻才可用。下面可分配作业2，可以将其分配给机器2或机器3，假设将作业2分配给机器2，这样机器2直到第14时刻才可用。然后在0～6时刻将作业5分配给机器3处理。对于下一个待分配的作业 6，最先可用的机器为机器3，它在6时刻变为空闲，因此将作业6在6～11时刻分配给机器3。机器3下一次可用的时间为第 11时刻，因此可在11时刻调度作业3。按此方法继续下去，即可得到图 9-10的调度方案。

定理9-2 [Graham] 令$F^*(I)$为在m 台机器上执行作业集合I 的最佳调度完成时间，$F(I)$）为采用LPT调度策略所得到的调度完成时间，则

$$\frac{F(I)}{F^*(I)} \leqslant \frac{4}{3} - \frac{1}{3m}$$

证明 参见1996年纽约计算机科学出版社出版的 *Computer Algorithm C++*，作者E.Horowitz, S.Sahni和S.Rajasekeran。

实际上，LPT调度比定理9-2所给界限更接近最佳算法。利用堆可在 O($n$log$n$) 时间内建立LPT调度方案。首先，当 $n \leq m$时，只需要将作业 i在0～t_i 时刻内分配到机器 i 上去处理。当$n > m$时，可以首先利用HeapSort（见程序9-12）将作业按处理时间递增的顺序排列。为了建立LPT调度，作业按相反次序进行分配。为决定作业分配给哪一台机器，必须知道哪台机器最先可用。为此，维持一个m 台机器的最小堆，这个最小堆的每个元素为MachineNode类型（见程序9-13）。avail是机器可用的时刻，ID为机器编号。DeleteMin用来获取最先可用的机器。当机器的可用时间增加后，需将其再次插入到最小堆中。由于所有机器的最初可用时间都为 0时刻，所以这些机器的avail值都为0。程序9-14给出了实现代码。类型T至少必须包括time与ID域。程序9-13为*a和T提供了一个比较合适的类型 JobNode。ID为每个作业的唯一标识，time为每个作业所需的处理时间。

程序9-13 JobNode **及**MachineNode **数据类型**

```
class  JobNode {
  friend void LPT(JobNode *, int, int);
  friend void main(void);
  public:
    operator int () const {return time;}
  private:
    int ID,    // 作业号
        time;  // 处理时间
};

class  MachineNode {
  friend void LPT(JobNode *, int, int);
  public:
    operator int () const {return avail;}
  private:
    int ID,     // 机器号
        avail;  // 何时变空闲
};
```

程序9-14 构造LPT调度

```
template <class T>
void LPT(T a[], int n, int m)
{// 构造一个有 m 台机器的 LPT 调度
  if (n <= m) {
    cout << "Schedule one job per machine." << endl;
    return;}

  HeapSort(a,n); // 按升序排列
  // 对 m 台机器及最小堆进行初始化
  MinHeap<MachineNode> H(m);
  MachineNode x;
  for (int i = 1; i <= m; i++) {
     x.avail = 0;
     x.ID = i;
     H.Insert(x);
     }

  // 构造调度
  for (i = n; i >= 1; i--) {
    H.DeleteMin(x);  // 获取第一台空闲的机器
    cout << "Schedule job " << a[i].ID
         << " on machine " << x.ID << " from "
         << x.avail << " to "
         << (x.avail + a[i].time) << endl;
    x.avail += a[i].time;  // 新的可用时间
    H.Insert(x);
    }
}
```

LPT 复杂性分析

当n≤m时，LPT函数所花费时间为$\Theta(1)$。当n > m时，堆排序花费时间为O(nlogn)。虽然在堆的初始化时做了m 次插入，但由于所有元素具有相同的值，所以每次插入实际开销为$\Theta(1)$，因此初始化所花总时间为 $\Theta(m)$。在第二个for循环中，执行了n 次DeleteMin和n 次Insert操作，每次需O(log*m*)的时间，因此需时O(nlogm)。所以总的时间为：O(nlogn+nlogm)=O(nlogn) (因为n> m)。

9.5.3 霍夫曼编码

在7.5.2节中介绍了一种基于LZW算法的文本压缩工具，这种算法利用了字符串在文本中重复出现的规律。霍夫曼编码（Huffman code）是另外一种文本压缩算法，它根据不同符号在一段文字中的相对出现频率来进行压缩编码。假设文本是由 *a*,*u*,*x*,*z* 组成的字符串，若这个字符串的长度为1000，每个字符用一个字节来存贮，共需1000个字节（即8000位）的空间。如果每个字符用2位二进制来编码（00=*a*, 01=*x*, 10=*u*, 11=*z*），则用2000位二进制即可表示1000个字符。此外，还需要一定的空间来存放编码表，可采用如下格式来存储：

符号个数，代码1，符号1，代码2，符号2，…

符号个数及每个符号分别用 8位二进制来表示，每个代码需占用 [$\log_2$ (符号个数)] 位二进制。因此，在本例中，代码表需占用5*8+4*2=48位，压缩比为8000/2048=3.9。

利用上述编码方法，字符串 *aaxuaxz* 的压缩编码为二进制串00000110000111，每个字符的编码具有相同的位数（两位）。从左到右依次从位串中取出 2位，通过查编码表便可获得原字符串。

在字符串*aaxuaxz*中，*a* 出现了三次。一个字符出现的次数称为频率（ frequency），*a,x,u,z* 在这个字符串中出现的频率分别为3，2，1，1。当每个字符出现的频率有很大变化时，可以通过可变长的编码来降低每个位串的长度。如果使用编码（ 0=*a*, 10=*x*, 110=*u*, 111=*z*），则aaxuaxz的压缩编码为0010110010111。编码长度为13位，比原来每个字符用2位、总长为14位要稍好一些。当频率相差大时这种差别会更为明显。如果四个字符的出现频率分别为（ 996,2,1,1），则每个字符用2位编码所得到编码的长度为2000位，而用可变长编码则仅为1000位。

但是怎样对位串进行解码呢？若每个代码为 2位，则解码很容易——只需每次取出2位并利用编码表来得到这两位代表什么。若使用可变长编码，则并不知道每次应取出多少位，字符串aaxuaxz经编码为001011001011，当从左至右解码时，必须知道第一个字符的代码是 0，00还是001。由于没有哪个字符的代码以00打头，因此第一个字符的代码必为0，根据编码表可知该字符为*a*。下一个代码为0，01或010，同理，由于不存在以01打头的字符代码，因此代码必为 0，相应元素为*a*。根据这种方法不断进行下去，就可以对整个位串进行解码。

为什么能够采用上述方法进行解码呢？通过仔细观察所使用的 4种代码（0，10，110，111），可以发现没有任何一个代码是另一代码的前缀。因此，当从左到右检查代码时，可以很确定地得到与实际代码相匹配的字符。

可以利用扩充二叉树（见9.4.1节定义）来派生一个实现可变长编码的特殊类，该类满足上述前缀性质，被称为霍夫曼编码。

在扩充二叉树中，可对从根到外部节点的路径进行编码，方法是向左孩子移动时取 0，向右孩子移动时取1。在图9-6b 中从根到外部节点*b*的路径编码为010，到节点（*a,b,c,d,e,f*）的路径编码分别为（00，010，011，100，101，11）。注意到每条从根到外部节点的路径的编码不会是另一条路径编码的前缀，因此，这种编码可用来对字符 *a,b,⋯,f* 分别编码。令*s* 为由这些字符组成的字符串，*F*(*x*)为字符*x*(*x* {*a,s,c,d,e,f* } 的频率。若利用上述代码对 *s* 进行编码，则编码后的串长为：

$$2*F(a)+3*F(b)+3*F(c)+3*F(d)+3*F(e)+2*F(f)$$

对于一棵具有外部节点1，⋯ *n* 的扩充二叉树，对应的压缩编码串的长度为：

$$WEP=\sum_{i=1}^{n}L(i)*F(i)$$

其中$L(i)$ 为从根到达外部节点 *i* 的路径长度（即路径的边数）；WEP为二叉树的加权外部路径长度（weighted external path length)。为了减小压缩编码串的长度，必须利用二叉树中的编码，其中二叉树的外部节点对应于字符串中被编码的字符，且它的加权外部路径长度最小。若二叉树对于给定的频率具有最小加权外部路径长度，则这棵树被称为霍夫曼树（ Huffman tree)。

为了利用霍夫曼编码对字符串或一段文本进行压缩编码，必须：

1) 获得不同字符的频率。

2) 建立具有最小加权外部路径的二叉树（即霍夫曼树），树的外部节点用字符串中的字符表示，外部节点的权重（weight）即为该字符的频率。

3) 遍历从根到外部节点的路径得到每个字符的编码。

4) 用字符的编码来代替字符串中的字符。

为了方便解码，需要保存字符代码映射表或每个字符的频率表。对于后一种情况，可以采用方法2）来重构霍夫曼编码树，以得到相应的霍夫曼编码。后面将详细讨论方法 2)。

为了构造霍夫曼树，首先从仅含一个外部节点的二叉树集合开始，每个外部节点代表字符串中一个不同的字符，其权重等于该字符的频率。此后不断地从集合中选择两棵具有最小权重的二叉树，并把它们合并成一棵新的二叉树，合并方法是把这两棵二叉树分别作为左右子树，然后增加一个新的根节点。新二叉树的权重为两棵子树的权重之和。这个过程可一直持续到仅剩下一棵树为止。

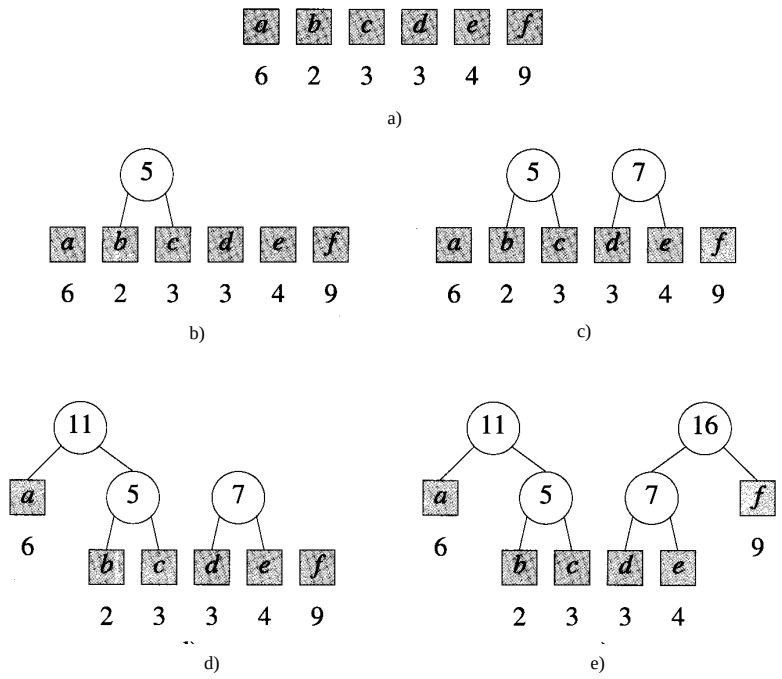

图9-11 建立霍夫曼树

a) 初始集合 b) 第一次合并之后 c) 第二次合并之后 d) 第三次合并之后 e) 第四次合并之后

下面利用上述方法来构造由6个字符（*a*,*b*,*c*,*d*,*e*,*f*）构成的霍夫曼树，这6个字符的频率分别为（6,2,3,3,4,9)。初始的二叉树集合如图 9-11a 所示，方框外部的数字为树的权重。首先选择具有最小权重的树*b* 和具有次小权重的树*c*，将*b* 与*c* 合并，所得到的结果如图 9-11b 所示。接下来选择具有最小权重的两棵树*d* 和*e* 进行合并，合并后的结果如图 9-11c 所示。从图9-11c 的四棵树中选出具有最小权重的树 *a* 和权重为5的树，将它们合并后得到权重为 11的树。从剩下的三棵树（如图9-11d所示）中选取树f及权重为7的树，并进行合并。至此，还剩下图 9-11e 所示的两棵树，将这两棵树合并后即得到图 9-6b 所示的二叉树，其权重为27。

定理9-3 上述过程所建立的二叉树具有最小加权外部路径。

证明 留作练习（练习19)。

霍夫曼树的建立过程可以利用最小堆来实现，最小堆用来存贮二叉树集合。最小堆中的每个元素包括一棵二叉树及其权重值，二叉树本身是 8.8节所定义的BinaryTree类的一个成员。

对于外部节点其data域设置为它所代表的字符，内部节点的data域设置为0。为了方便，可以假设字符用数字1到n表示。程序9-15中的HuffmanTree函数假定已经定义了类Huffman（见程序9-16）。

程序9-15 建立霍夫曼树

```
template <class T>
BinaryTree<int> HuffmanTree(T a[], int n)
{// 根据权重 a[1:n]构造霍夫曼树
  // 创建一个单节点树的数组
  Huffman<T> *w = new Huffman<T> [n+1];
  BinaryTree<int> z, zero;
  for (int i = 1; i <= n; i++) {
    z.MakeTree(i, zero, zero);
    w[i].weight = a[i];
    w[i].tree = z;
    }

  // 把数组变成一个最小堆
  MinHeap<Huffman<T> > H(1);
  H.Initialize(w,n,n);

  // 将堆中的树不断合并
  Huffman<T> x, y;
  for (i = 1; i < n; i++) {
    H.DeleteMin(x);
    H.DeleteMin(y);
    z.MakeTree(0, x.tree, y.tree);
    x.weight += y.weight; x.tree = z;
    H.Insert(x);
    }

  H.DeleteMin(x); // 最后的树
  H.Deactivate();
  delete [] w;
  return x.tree;
}
```

程序9-16 Huffman类

```
template<class T>
class  Huffman {
  friend BinaryTree<int> HuffmanTree(T [], int);
  public:
    operator T () const {return weight;}
  private:
    BinaryTree<int> tree;
    T weight;
};
```

HuffmanTree输入n 个频率（即权重，存放在数组a 中）的集合并返回一个霍夫曼树。它首先建立n 棵二叉树，每棵树仅由一个外部节点构成。这些树用数组 w 来存贮，后面将把w初始化为一个最小堆。第二个 for循环从最小堆中取出权重最小的两棵二叉树并将它们合并成一棵二叉树，然后将结果插入最小堆中。

HuffmanTree 函数的复杂性

当T为内部数据类型时，构造和删除数组 w 所需时间为Θ(1)，而当T为用户自定义的类时需耗时Θ(n)。第一个for循环和堆的初始化需Θ(n) 时间。第二个for循环中，总共执行了2(n-1)次删除最小元素及n-1次插入操作，需O(nlogn) 的时间。函数其余部分花费的时间为Θ(1)。因此HuffmanTree函数总的时间复杂性为O(nlogn)。

练习

14. 比较在最坏情况下堆排序与插入排序的执行时间。对于堆排序，利用一些随机序列来估计最坏情况下的执行时间。当n 取什么值时，堆排序比插入排序所用时间少？

15. 利用练习9与10的思想，实现一种比程序9-12更块的堆排序。利用随机数据比较两种实现的执行时间。

16. 一种稳定的排序算法是指具有相同值的记录在排序前与排序后具有相同的顺序。假设记录3与记录10的关键值相同，对于一个稳定排序，排序后记录 3仍在记录10的前面。请问堆排序是稳定排序吗？插入排序呢？

17. 在程序9-14中，第二个for循环每次执行一次最小元素删除和一次插入，这两个操作使最小关键值的量得以增加，所增加的量为刚被调度的作业的处理时间。可以利用一个扩充的最小优先队列来加快程序 9-14的执行。这种扩充包括最小优先队列通常支持的函数以及函数IncreaseMinkey (x,e)，后者用于将最小关键值增加x 并将结果返回到e 中，e 即原来的含有最小关键值的元素。IncreaseMinkey 函数首先将根的值增加 x，然后沿堆向下移动（尤如最小元素删除操作），并将元素e 上移直到找到一个合适的位置。

1) 设计一个新类 ExtendedMinHeap，提供 MinHeap类中出现的所有成员函数及IncreaseMinkey函数。template<class Te, class Tk> class ExtendMinHeap 应从类MinHeap<Te>中派生而来。Te 为元素的数据类型，Tk 为关键值的数据类型，因此在IncreaseMinKey(x,e)中，x 是Tk类型，e 为Te 类型。可以假设操作符+=已被重载，因此e += x 语句表示将e 的关键值增加x。

2) 利用函数IncreaseMinKey重写程序9-14。

3) 比较代码与程序9-14的代码。

18. 将n 件物品装入容器，第i 件占用的空间为s_i，且每个容器的容量为c。装入过程采用最不合适法则（worst-fit rule），每次只能给容器分配一件物品。在分配物品时，寻找具有最大剩余容量的容器，若容器可以装下这件物品，则可分配，否则需使用一个新的容器。

1) 编写一个程序，输入为n，s_i 和c，输出为物品在容器中的分配方案。利用最大堆来记录容器的可用空间。

2) 程序的复杂性是多少？（复杂性应为n 及容器个数m 的函数）

*19. 利用外部节点的数目进行归纳证明定理 9-3。在归纳步中可以假定存在一棵二叉树，该树拥有最小加权外部路径，且存在一棵子树，子树中有一个内部节点和两个外部节点，外部节点分别对应两个最小频率。

20. 写一个程序，其输入为HuffmanTree函数（见程序9-15）所建立的霍夫曼树，输出为编码表。程序的复杂性为多少？

*21. 设计一个完整的，基于霍夫曼编码的压缩-解压缩软件包，并用适当的文本文件来检验其正确性。

*22. 欲存储0到511之间的n 个整数，利用霍夫曼编码编写一个压缩-解压缩包来实现。

23. run 是一个有序元素序列。假设两个 run 可以在$\Theta(r+s)$ 时间内合并为一个run，其中r与s 分别为要合并的两个run 的长度。通过不断地合并两个run，可将n 个不同长度的run 最终合并成一个。解释如何运用霍夫曼树来合并n 个run，以使时间开销最小。

9.6 参考及推荐读物

想更详细地了解优先队列及其各种变化，可参考E.HoroWitz, S.Sahni, D.mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1994。

高度优先左高树可参考专题论文 R.Tarjan. *Data Stuctures and Network Algorithms*. SIAM, 1983。权重优先左高树可参见论文S.Cho, S.Sahni. Weight Biased Leftist Trees and Modified Skip Lists. *Proceedings, Second International Conference COCOON '96*, Lecture Notes in Computer Science, Springer Verlag 1090, 1996, 361~370。

可以从 *Computer and Intractability* 一书中找到更多的 NP-复杂问题，如： M.Gorey, D.Johnson. *A Guide to the Theory of NP-completeness*. W.H.Freeman, 1979; E.Horowitz, S.Sahni, S.Rajasekeran. *Computer Algorithms/C++*. Computer Science, 1996。