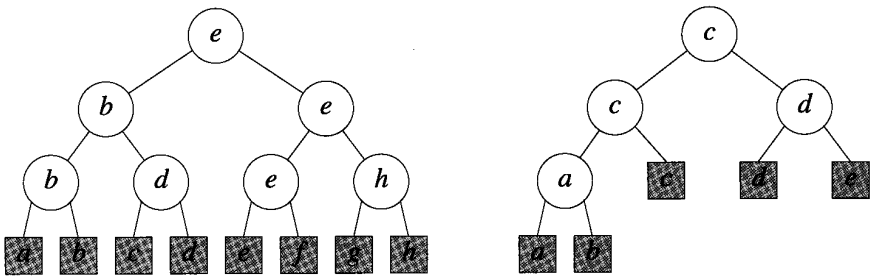
第10章 竞 赛 树

本章将介绍一类新的树，称为竞赛树。象 9.3节的堆一样，竞赛树也是完全二叉树，它可用8.4节定义的公式化描述的二叉树来进行最有效地存储。竞赛树支持的基本操作是替换最大（或最小）元素。如果有 n 个元素，操作的开销为 $\Theta(\log n)$。虽然利用堆和左高树也能获得同样的复杂性 $O(\log n)$，但它们都不能实现可预见的断接操作。所以当需要按指定的方式断开连接时，可选择竞赛树这种数据结构。比如选择最先插入的元素，或选择左端元素（假定所有元素按从左到右的次序排列）。

本章将研究两种竞赛树：赢者树和输者树。尽管赢者树更直观、更接近现实，但输者树的实现更高效。本章的应用部分将考察另一种 NP - 复杂问题——箱子装载。我们将利用竞赛树来高效地实现解决箱子装载问题的两个近似算法。试一试能否用本书迄今为止所介绍的其他任意一种数据结构以相同的时间复杂性来实现这两个算法，从中你会得到一些有益的启示。

10.1 引言

假定在一次乒乓球比赛中有 n 名选手，该比赛采用的是突然死亡（ sudden-death）的比赛规则，即一名选手只要输一场球，就被淘汰，最终必然只剩下一个赢者。定义此“ 幸存 ”的选手为竞赛赢家。图10-1给出由 $a \sim h$ 共8名选手参加的某次比赛。图中用一棵二叉树来描述整个比赛，每个外部节点分别代表一名选手，每个内部节点分别代表一场比赛，参加每场比赛的选手是子节点所对应的两名选手。这里，同一层节点所构成的一轮比赛可以同时进行。在第一轮比赛中 a 与 b、c 与 d、e 与 f、g 与 h 交战。每场比赛的赢者记录在代表该场比赛的内部节点中。在图10-1a 的例子中，第一轮比赛的4个赢家为 b、d、e 和 h，因而 a、c、f、g 被淘汰。下一轮比赛在 b 与 d，e 与 h 之间进行，胜者为 b 和 e ，故由 b、e 参加最后的决赛，最终赢者为 e。图10-1b 给出有 $a \sim e$ 共5名选手参加的比赛，最后的赢家为 c。



a) b)

图10-1 竞赛树

a) 8名选手 b) 5名选手

虽然图10-1的两棵树都是完全二叉树（实际上树a 还是满二叉树），但在现实中竞赛问题所对应的树不一定都是完全二叉树。然而，使用完全二叉树可以减小比赛的场次。对于 n 名选手的比赛，比赛场次为 $\lceil \log_2 n \rceil$ 个。因为图10-1中竞赛树的每一个内部节点都记录了对应比赛的赢

家，所以称之为赢者树（winner tree）。10.4节将介绍另一类型的树，称为输者树（loser tree），故名思义，它的每一个内部节点记录的是比赛的输者。竞赛树在某些情况下也被称为选择树（selection tree）。

为适应计算机的实现，需要对赢者树作一些适当的修改。不妨假定赢者树为完全二叉树。

定义 **[赢者树]** 对于n 名选手，赢者树是一棵含n 个外部节点，$n-1$个内部节点的完全二叉树，其中每个内部节点记录了相应赛局的赢家。

为决定一场比赛的赢家，假设每个选手有一得分且赢者取决于对两选手得分的比较。在最小赢者树（min winner tree）中，得分小的选手获胜；同理，在最大赢者树（max winner tree）中，得分大的选手获胜。有时，也可用左孩子对应的选手代表赢家节点。图 10-2a 给出含8名选手的最小赢者树，而图10-2b 给出含5名选手的最大赢者树。每个外部节点之下的数字表示选手得分。

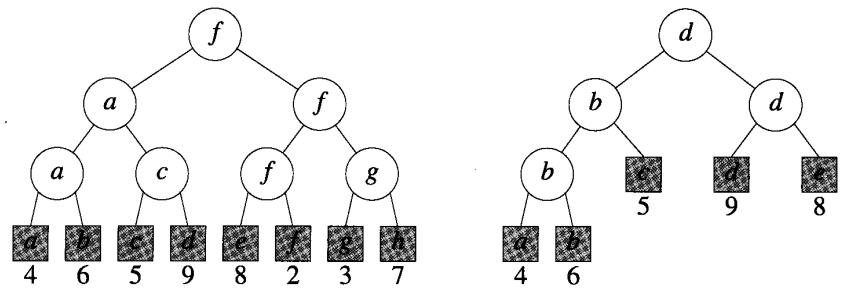


图10-2 赢者树

a) 最小赢者树 b) 最大赢者树

赢者树的一个优点在于即使一名选手得分改变了，也可以较容易地修改此树。如当选手 d 的值由9改为1时，只需沿从d 到根那条路径修改二叉树，而不必改变其他比赛的结果。有时甚至还可以避免重赛某些场次。如在图 10-2a 中当b 值由6改为5时，其父节点的比赛结果中b仍为输家，故此时所有比赛的结果未变，因此不必重赛 b 的祖父及曾祖父节点所对应的比赛。

对于一棵有n 名选手的赢者树，当一个选手的得分发生变化时，需要修改的比赛场次介于$0\sim\log_2 n$ 之间，因此重构赢者树需耗时O(log$n$)。此外，由于n 名选手的赢者树在内部节点中共需进行n-1场比赛（按从最低层到根的次序进行），因此赢者树的初始化时间为 $\Theta(n)$。也可以采用前序遍历方式来完成初始化，方法是在每一访问步中安排一场比赛。

例10-1 **[排序]** 可用一个最小赢者树在$\Theta(n\log n)$时间内对n 个元素进行排序。首先，用n 个元素代表n 名选手对赢者树进行初始化。利用元素的值来决定每场比赛的结果，最后的赢家为具有最小值的元素，然后将该选手（元素）的值改为最大值 (如∞)，使它赢不了其他任何选手。在此基础上，重构该赢者树，所得到的最终赢家为该排序序列中的下一个元素。以此类推，可以完成n 个元素的排序，时耗为$\Theta(n)$。每次改变竞赛赢家的值并重构赢者树的时耗为 $\Theta(\log n)$，这个过程共需执行n-1次，因此整个排序过程所需要的时间为 $\Theta(n\log n)$。

例10-2 [run **的产生**] 本书前几章所讨论的排序方法（插入排序、堆排序等）都属内部排序法（internal sorting method），待排序的各个元素必须放入计算机内存中。当待排序元素所需的空

间超出内存的最大容量时，内部排序法就需要与存储了部分或所有元素的外部存储介质（如磁盘）打交道，致使排序效率不高。在这种情况下必须采用外部排序法（external sorting method）。外部排序一般包括两步：1) 产生部分排序结果run；2) 将这些run合并在一起得到最终的run。

假设要为含16 000个元素的记录排序，且在内部排序中一次可排序1000个记录。为此在第1）步中，重复以下步骤16次，可得到16个排序结果(run)：

输入1000个记录
用内部排序法对这1000个记录进行排序
输出排序结果run

接下来需产生最终的排序结果，即完成上述第2) 步。在本步中要重复地将k 个run 合并成一个run。如在本例中有16个run（如图10-3所示），它们的编号分别为R_1, R_2, … R_{16}。首先，将R_1,～R_4 合并得到S_1，其长度为4000个记录，然后将R_5～R_8 合并，以此类推。接下来将S_1～S_4合并得到T_1，T_1 便是外部排序的最终结果。

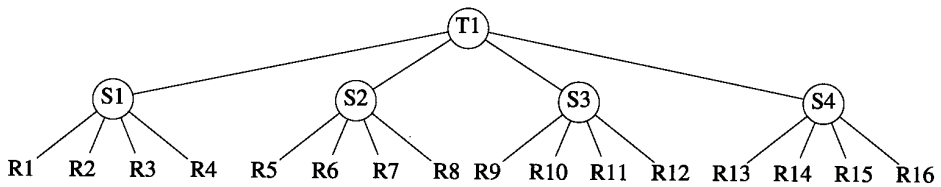


图10-3 16个run的四路合并

一种合并k 个run的简单方法是：从k 个run的前面不断地移出值最小的元素，该元素被移至输出run中。当所有的元素从k 个输入run移至输出run中时，便完成了合并过程。注意到在选择输出run的下一元素时，只需要知道内存中每个输入 run的前面元素的值。故只要有足够内存保存k 个元素值，就可合并任意长度的k 个run。在实际应用上，我们感兴趣的是能一次输入/出更多元素以减少输入/出的次数。

在上面所列举的16 000个记录的例子中，每个run有1000个记录，而内存容量亦为1000个记录。为合并前四个run，可将内存分为五个缓冲区，每个容量为 200个记录。其中前四个为输入run 的缓冲区，第五个为输出 run 的缓冲区，用于存放合并的记录。从四个输入 run中各取200个记录放入输入缓冲区中，这些记录被不断地合并并放入输出缓冲区中，直到以下条件发生：

1) 输出缓冲区已满。
2) 某一输入缓冲区空。

当第一个条件发生时，将输出缓冲区内容写入磁盘，写完之后继续进行合并。当第二个条件发生时，则从对应于空缓冲区的输入 run中继续读取记录放入该缓冲区，读取过程结束后合并继续进行。当这些run中的4000个记录都写入输出run中时，四个run的合并过程结束（更详细的描述参见E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structure in C++*. Computer Science Press, 1995。）

run合并所需时间的决定因素之一是第 1）步所产生的run的个数。通过使用赢者树，可减少run的个数。开始时，用一个含p 名选手的赢者树，其中每个选手对应输入集合中的一个元素。每个选手有一个值（即对应的元素值）和一个 run 号。前p 个元素的run号均为1。当两选手进行比赛时，具有较小run 号的选手获胜。在run 号相同的情况下，按选手的值进行比较，具有

较小值的元素成为赢家。为产生 run，重复地将最终赢者 W 移入它的run号所对应的run中，再用下一个输入元素 N 取代 W 原来的位置。如果 N 的值大于等于 W 的值，则将元素 N 作为相同run的成员输出（它的run号与 W 的相同）。如果 N 的值小于 W 的值，若将 N 放入与 W 相同的run中将会破坏run中元素的排序，因此将 N 的run号设置为 W 的run号加1。

当采用上述方法生成run时，run的平均长度约为 $2p$。当 $2p$ 大于内存容量时，我们希望能得到更少的run（与上述方法相比）。事实上，倘若输入集合已排好序 (或几乎排好序)，只需产生最后的run，则可直接跳到run的合并步，即第2）步。

例10-3 [*k* **路合并**] 在 k 路合并(见例10-2)中，k 个run要合并成一个排好序的run。按例10-2中所述的方法，进行 k 路合并时，因在每一次循环中都需要找到最小值，故将每一个元素合并到输出run中需 $O(k)$ 时间，因此产生大小为 n 的run所需要的时间为 $O(kn)$。若使用赢者树，则可将这个时间缩短为 $\Theta(k+n\log k)$。首先用 $\Theta(k)$ 的时间初始化含 k 个选手的赢者树。这 k 个选手都是 k 个被合并run中的头一个元素。然后将赢者移入输出 run中并用相应的输入 run中的下一个元素替代之。如果在该输入 run中无下一元素，则需用一个 key值很大(不妨为 ∞) 的元素替代之。k 次移入和替代赢家共需耗时 $\Theta(\log k)$。因此采用赢者树进行 k 路合并的总时间为 $\Theta(k+n\log k)$。

练习

1. 1) 描述如何用最小堆来替代最小赢者树产生 run（见例10-2），产生run中每一元素的时间为多少？
   2) 在此应用中，最小赢者树与堆相比，有哪些优点和缺点？
2. 1) 在进行 k 路合并时（见例10-3），如何用最小堆替代最小赢者树？
   2) 在此应用中，最小赢者树与堆相比有哪些优点和缺点？

10.2 抽象数据类型WinnerTree

在定义抽象数据类型 *WinnerTree* 时，假设选手数是固定的，也就是说，初始化选手数为 n 的树之后，不再增减选手数。选手本身不是赢者树的组成部分，故组成赢者树的成分是图10-1所示的内部节点。因此，赢者树需要支持的操作有：创建空的赢者树、用 n 名选手初始化赢者树、返回赢者、在从选手 i 到根的路径上重新进行比赛。各操作的定义在 ADT10-1中给出。

ADT10-1 赢者树的抽象数据类型描述

```
抽象数据类型 WinnerTree{
实例
    完全二叉树，每个节点代表相应比赛的赢者，外部节点代表选手
操作
    Create( )：创建一个空的赢者树
    Initialize(a, n)：对有n 个选手a[1:n] 的赢者树进行初始化
    Winner( )：返回比赛的赢者
    Replay(i)：选手i 变化时，重组赢者树
 }
```

10.3 类WinnerTree

10.3.1 定义

假设用完全二叉树的公式化描述方法来定义赢者树。 n名选手的赢者树需n - 1个内部节点t[1:n - 1]。选手（或外部节点）用数组e[1:n]表示。故t[i]为数组e的一个索引而且类型为int。t[i]给出了赢者树中节点i 对应比赛的赢者。图10-4给出了5选手赢者树中各节点与数组t和e之间的对应关系。

为实现ADT操作，必须能够确定外部节点e[i]的父节点t[p]。当外部节点的数目为n 时，内部节点数为n - 1。最底层最左端的内部节点的编号为2^s，这里$s=\lfloor \log_2 (n-1) \rfloor$。因此最底层的内部节点数为 $n - 2^s$，最底层的外部节点数$LowExt$ 为内部节点数的2倍。例如，在图10-4的树中，n=5，s=2，最底层最左端的内部节点为 $t[2^s]=t[4]$，该层的内部节点数共为n - 4=1个，最底层外部节点数为2，倒数第2层的最左端的外部节点号为$LowExt$+1。设$offset=2^{s+1}-1$，可知对于任何外部节点e[i]，其父节点t[p]满足以下公式：

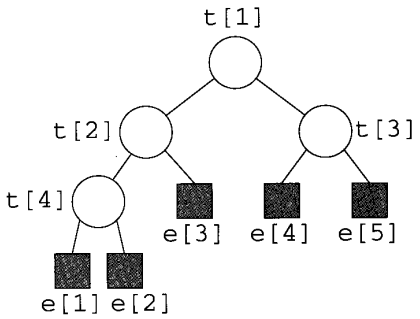


图10-4 树与数组的对应关系

$$p=\begin{cases}(i+offset)/2 & i \leqslant LowExt \\ (i-LowExt+n-1)/2 & i>LowExt\end{cases} \qquad (10\text{-}1)$$

10.3.2 类定义

在程序10-1中给出了赢者树的类定义。私有成员包括： MaxSize（允许的最大选手数）、n（赢者树初始化时的选手数）、t（内部节点数组）、e（外部节点数组）、LowExt和Offset。可以用确定比赛赢者的函数作参数来调用 Initialize和Replay。Winner(a,b,c)返回的是a[b]和a[c]之间比赛的赢者。通过适当定义Winner，可构造最小赢者树、最大赢者树等。

程序10-1 赢者树的类定义

```
template<class T>
class WinnerTree {
  public:
    WinnerTree(int TreeSize = 10);
    ~WinnerTree() {delete [] t;}
    void Initialize(T a[], int size, int(*winner)(T a[], int b, int c));
    int Winner()  const {return (n) ? t[1] : 0;}
    int Winner(int i) const {return (i < n) ? t[i] : 0;}
    void RePlay(int i, int(*winner) (T a[], int b, int c));
  private:
    int MaxSize;
    int n;      //当前大小
```

```
    int LowExt;  //最底层的外部节点
    int offset;  //2^k-1
    int *t;      //赢者树数组
    T *e;        //元素数组
  void Play(int p, int lc, int rc, int(*winner)(T a[], int b, int c));
};
```

10.3.3 构造函数、析构函数及Winner函数

在程序10-2中，构造函数创建了一个初始为空的赢者树（n=0），它可以最多处理MaxSize个选手。可用的内部节点为t[1]～t[MaxSize-1]。析构函数和Winner函数被定义为内部函数，见程序10-1。

程序10-2 创建赢者树

```
template<class T>
WinnerTree<T>::WinnerTree(int TreeSize)
{//构造赢者树
  MaxSize = TreeSize;
  t = new int[MaxSize];
  n = 0;
}
```

10.3.4 初始化赢者树

程序10-3给出了初始化操作的代码。a为选手数组，size表示选手数，winner用于得到a[b]和a[c]之间比赛的赢家。在程序中，首先验证赢者树能否处理size个选手，如果能，则初始化n和e。接下来计算s＝[$\log_2$(n-1)]，再由s得到LowExt和offset。为了完成初始化，从赢者树的外部节点i开始逐层向上进行比赛，其中i依次取为1,2,…, n。注意在这个过程中，仅当从一个节点的右孩子上升到该节点时，才在该节点进行一场比赛。倘若是从左孩子上升到该节点，因其右子树的赢者尚未确定，因而不能在该节点上进行比赛。在第二个 for循环中，各比赛由最底层选手（外部节点）来激活。作为右孩子的选手位于e[2]，e[4]，…，e[LowExt]中。比赛由保护成员函数 Play（见程序10-4）来完成。对于某个 e[i]，其父节点为t[(offset+i)/2]（见公式（10-1）），对手为e[i-1]。为了执行由其他n-LowExt个选手激活的比赛，必须确定n是否为奇数。若n为奇数，则e[LowExt+1]为右孩子，否则为左孩子。当n为奇数时，对手为e[t[n-1]]，父节点为t[(n-1)/2]。最后的for循环语句激活其余外部节点的比赛。

程序10-3 初始化赢者树

```
template<class T>
void WinnerTree<T>::Initialize(T a[], int size, int(*winner)(T a[], int b, int c))
 {//对于数组a初始化赢者树
    if (size > MaxSize || size < 2)
      throw BadInput();
   n = size;
```

```
    e = a;

    //计算 s = 2^log (n-1)
    int i, s;
    for (s = 1; 2*s <= n-1; s += s);

    LowExt = 2*(n-s);
    offset = 2*s-1;

    //最底层外部节点的比赛
    for (i = 2; i <= LowExt; i += 2)
      Play((offset+i)/2, i-1, i, winner);
    //处理其余外部节点
    if (n % 2) {//当n奇数时，内部节点和外部节点的比赛
          play(n/2 , t[n-1],LowExt+1,winner);
          i = LowExt+3;}
    else i = LowExt+2;

    //i为最左剩余节点
     for (; i <= n; i += 2)
          play((i-LowExt+n-1)/2, i-1,i, winner);
    }
```

程序10-4 通过比赛对树进行初始化

```
template<class T>
void WinnerTree<T>::Play(int p, int lc, int rc, int(*winner)(T a[], int b, int c))
{//在t[p]处开始比赛
 //lc和rc是t[p]的孩子
   t[p] = winner(e, lc, rc);

   //若在右孩子处，则可能有多场比赛
   while (p > 1 && p % 2) {//在右孩子处
      t[p/2] = winner(e, t[p-1], t[p]);
      p /= 2;  //到父节点
      }
}
```

函数Play首先在内部节点t[p]处进行比赛，然后沿赢者树不断向上移动并进行比赛，直到从一个左孩子移到其父节点为止。

为了弄清Initialize是如何工作的，现在来考察图 10-4所示的5选手例子。在第二个for循环中，e[1]和e[2]之间有一场比赛，该比赛的赢者记录在 t[4]中。此时t[2]对应的比赛尚未进行。这是因为t[4]是其父节点t[2]的左孩子。此时n 为奇数，因而在t[2]处进行e[t[4]]和e[3]间的比赛。赢者记录在t[2]。在第三个for 循环中，首先进行t[3]处的比赛（e[4] 和e[5]间的比赛)。因t[3]为其父节点的右子女，故此时t[1]对应的比赛也可进行。

现在来分析函数Initialize的复杂性。 s 的计算需Θ(logn)时间。第二次和第三次 for 循环

（包括函数Play）共需Θ(n)时间。调用所有Play(共n-1次)所需时间为Θ(n)，因此Initialize 总的复杂性为Θ(n)。

10.3.5 重新组织比赛

当选手i 相应的值改变后，需要重新进行某些甚至所有的从外部节点 e[i]到根t[l]路径上的比赛。为简单起见，将再次执行该路径上的所有比赛。实际上，在例 10-1、10-2和10-3中，只有赢者的值会发生变化。一个赢者的值发生变化必然会导致重新执行从赢者对应的外部节点开始到根的路径上的所有比赛。程序10-5给出了相应的代码。

程序10-5 当元素i 改变时重新组织比赛

```
template<class T>
void WinnerTree<T>::RePlay(int i, int(*winner)(T a[], int b, int c))
{//针对元素i重新组织比赛
  if (i <= 0 || i > n) throw OutOfBounds();

  int p,   //比赛节点
    lc,  //p的左孩子
    rc;  // p的右孩子
 //找到第一个比赛节点及其子女
 if (i <= LowExt) {//从最底层开始
   p = (offset + i)/2;
   lc = 2*p - offset; // p的左孩子
   rc = lc+1; }
  else {p = (i-LowExt+n-1)/2;
       if (2*p == n-1) {lc = t[2*p];
                        rc = i;}
       else {lc = 2*p - n + 1 + LowExt;
             rc = lc+1;}
       }
t[p] = winner(e, lc, rc);

//剩余节点的比赛
p /= 2;  //移到父节点处
 for (; p >= 1; p /= 2)
    t[p] = winner(e, t[2*p], t[2*p+1]);
 }
```

为了重新进行比赛，需要利用公式（ 10-1）来确定第一次比赛的选手。 t[p]处的比赛是e[lc]和e[rc]之间的比赛，赢者记录在t[p]中。其余比赛都在for循环中完成。比赛的总次数为Θ(赢者树的高度)＝Θ(logn)。

练习

3. 修改函数RePlay (见程序10-5)，要求避免进行一些不必要的比赛。通常来说，当一次比赛的赢者与上一次该比赛的赢者相同时，可不再进行该场比赛。

4. 编写一个排序程序，该程序用赢者树来重复地将元素插入到已排好序的序列中去（见例10-1）。

10.4 输者树

仔细观察赢者树的RePlay操作。不难发现，在许多应用中，只是在前一赢者被新的选手取代后才执行此操作（见例10-1、10-2和10-3）。在这些应用中，从取代赢者的外部节点到根节点的路径上的所有比赛都要重新进行。考察图 10-2a 中的最小赢者树。当赢者 f 被值为5的选手f'取代时，第一场比赛在e 和f' 之间进行，其中e 是该节点处上一次与f 比赛的输方。而f' 作为新的赢者，在内部节点t[3]上与g 进行下一场比赛。注意g 是t[3]处与f 的前一场比赛的输者，而t[3]上g 与f' 的比赛的赢者为g。接下来，g 与根节点上的a 比赛，而a 是在根节点处上一场比赛的输方，

如果在每个内部节点中记录的是该节点比赛的输者而不是赢者，那么当 e[i]发生变化时，就可以减少确定比赛选手的工作量（从节点 e[i]到根节点的路径上）。最终的赢者可记录在t[0]中。图10-5a 为对应于图10-2a 8名选手的输者树。现在当赢者f 的值变成5时，首先移动到它的父节点t[6]，对应的比赛为e[t[6]]和e[6]间的比赛。欲确定f'= e[6]的对手，只需检查t[6]，而在赢者树中还必须检查t[6]的其他子女。在t[6]的比赛完成后，输者e 被记录于此节点中，f ’继续与t[3]中前一场比赛的输家g 进行比赛。此次的输家为f ’，记录于t[3]，赢家g则与t[1]中上一场比赛的输家a 比赛，输者为a，记录于t[1]。新的输者树如图10-5b 所示。

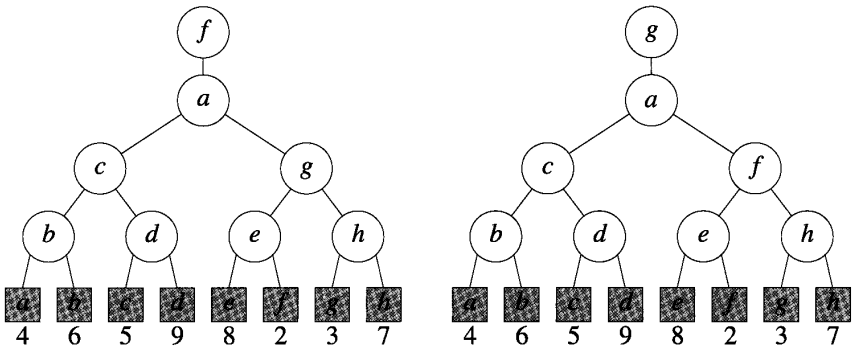


图10-5 8个选手的最小输者树

a) 初始状态 b) 当e[b]改变之后

虽然使用输者树可简化一个赢者发生变化后重新比赛的过程，但它并不能简化其他选手发生改变时的情况。例如，当选手d 的值由9变为3，t[5]、t[2]、t[1]上的比赛将重新进行。在t[5]处，d 必须与c 比赛，但c 不是该节点上一场比赛的输家。在t[2]处，d 必须与a 比赛，但a 也不是该节点上一场比赛的输家。在t[1]处，d 必须与f 比赛，f 同样不是该节点上一场比赛的输家。为了更容易地重新进行这些比赛，还得用到赢者树。因此仅当 e[i]为前次比赛的赢家时，对于函数RePlay(i)，采用输者树比采用赢者树执行效率更高。

练习

5. 1) 模仿赢者树的类定义（见程序10-1)设计一个C++类LoserTree。比赛赢者可记录在t[0]

中。可定义函数RePlay（）代替原来的共享成员函数RePlay(i)，该函数从上一次比赛的赢者开始重新组织比赛。

2) 一种对输者树进行初始化的简单方法是先构造一棵赢者树，然后按层次遍历赢者树，遍历过程中用输者替代每个内部节点。遍历从顶层到底层进行。 t[i]的孩子会告诉我们在t[i]处参加比赛的选手，然后根据选手信息确定谁是输家。采用上述策略编写一个初始化函数 Initialize。证明代码能在$\Theta(n)$的时间内对n个选手的输者树进行初始化。

3) 使用程序10-3中的策略编写Initialize函数。尽可能进行比赛并记录比赛输者。当某一比赛无法进行时，记录下该比赛对应的唯一选手。证明代码能在$\Theta(n)$时间对含n 个选手的输者树进行初始化。

6. 编写一个排序程序，利用输者树不断地将元素插入到已排好序的序列中。指出程序的复杂性。

10.5 应用

10.5.1 用最先匹配法求解箱子装载问题

在箱子装载问题中，有若干个容量为c 的箱子和n 个待装载入箱子中的物品。物品i需占s[i]个单元（0< s[i]≤c)。所谓成功装载（feasible packing)，是指能把所有物品都装入箱子而不溢出，而最优装载（optimal packing）则是指使用了最少箱子的成功装载。

例10-4 [卡车装载] 某一运输公司需把包裹装入卡车中，每个包裹都有一定的重量，且每辆卡车也有其载重限制（假设每辆卡车的载重都一样)。在卡车装载问题中，希望用最少的卡车来装载包裹。可将此问题转化为箱子装载问题，即卡车对应于箱子，包裹对应于物品。

例10-5 [集成片分布] 在给定宽度的电路板上按行布设一些电路集成片。集成片高度一致但宽度各不相同。电路板的最小高度由所使用的最小行数决定。集成片分布问题也可转化为箱子装载问题，即电路板的每行对应为一箱子，每个集成片对应为一个需装载的物品。电路板的宽度为箱子容量，而集成片的长度便相当于相应物品的容量。

箱子装载问题和机器调度问题（见8.5.2节）一样，也是NP-复杂问题。因此可用近似的算法求解。在箱子装载问题中，该算法可得到一个接近于最少箱子个数的解。对于箱子装载问题，有4种流行的求解算法：

1) 最先匹配法（First Fit, FF）

物品按1，2，…，n的顺序装入箱子。假设箱子从左至右排列。每一物品 i放入可盛载它的最左箱子。

2) 最优匹配法（Best Fit, BF）

设cAvail[j]为箱子j的可用容量。初始时，所有箱子的可负载容量为c 。物品i放入具有最小cAvail且容量大于s[i]的的箱子中。

3) 最先匹配递减法（First Fit Decreasing, FFD）

此方法与FF 类似，区别在于各物品首先按容量递减的次序排列，即对于 l≤i <n，有s[i]≥s[i+1]。

4) 最优匹配递减法（Best Fit Decreasing, BFD）

此法与BF相似，区别在于各物品首先按容量递减的次序排列，即对于 l≤i <n，有s[i]≥

s[i+1]。

可以看出，以上4种方法中没有一种能保证获得最优装载。但这4种方法都很直观实用。

设I为箱子装载问题的任一实例。$b(I)$为用于最优装载的箱子数。FF和BF中所用的箱子数不会超过$[(17/10)b(I)]$，而FFD和BFD中的箱子数不会超过（11/9）$b(I)$+4个。以上结论的证明很复杂，可参考M.Garey, K.Graham, D.Johnson, A.Yao. Resource Constrained Scheduling as Generalized Bin-Packing. *Journal of Combinatorial Theory, Series A,* 1976, 257~298和D. Johnson, A.Demers, J.Ullman, M.Garey, R.Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal on Computing,* 1974, 299~325。

例10-6 把四件物品s[1:4]=[3,5,2,4]放入容量为7的箱子中。使用FF法时，物品1放入箱子1；物品2放入箱子2；此时箱子1还可以放得下物品3，故物品3放入箱子1；物品4无法再放入前面这两个箱子，因此需再使用一个箱子。最后的方案为：使用三个箱子，物品 1和3放入箱子1；物品2放入箱子2；物品4放入箱子3。

若使用BF法，物品1和2分别放入箱子1和箱子2，由于箱子2比箱子1更适合放置物品3，因此将物品3放入箱子2，这样物品4又可放入箱子1。这种装载方案只用了两个箱子：物品1、4放入箱子1；物品2、3放入箱子2。

对于FFD和 BFD方法，首先将物品按2、4、1、3排序，最后所得装载方案均为：使用两个箱子，物品2、3放入箱子1；物品1、4放入箱子2。

可使用赢者树来实现FF和FFD算法，所需时间为Θ(nlogn)。因最多会用到n 个箱子，故可用n 个空箱子作为初始条件。设 avail[j]为箱子j 的可用空间。初始化时，对于所有箱子有avai1[j]=c。接下来，用avail[j]作为选手对最大赢者树进行初始化。图 10-6a 给出了在n=8和c=10条件下的最大赢者树。外部节点从左到右对应为箱子1到箱子8。在外部节点之下的数字为该箱子的装载容量。假设s[1]=8，为找到装载物品1的最左箱子，从根root[1]开始搜索。根据定义可知avail[t[1]]≥s[1]，因此至少有一个箱子可装载此物品。为找到最左箱子，要确定在箱子1～箱子4中是否有箱子含足够的空间来装载物品 1。当且仅当avai1[t[2]]≥S[1]时这些箱子之一有足够空间。在本例中此条件满足，因此可从根为2的子树开始继续搜索。现在要确定2的左子树（即根为4的子树）所包含的箱子中是否有容量合适的箱子，若有，则不必再考虑2的右子树。在本例中，因avail[t[4]]≥S[1]，故移动到左子树。由于树4的左子树是一个外部节点，所以可将s[1]放入节点4的任一个孩子之中，若左孩子有足够的空间则将其放入左孩子。当物品 1放入箱子1时， avail[1]减为2，然后从avail[2]开始重新比赛。新的赢者树如图 10-6b所示。现假设s[2]=6。因avai1[t[2]]≥6，可知在左子树中有一个箱子有足够的空间，因此可以先移动到该箱子处，然后再移到左子树4，并将物品2放入箱子2。新的结果如图10-6c所示。当s[3] = 5时，搜寻将进入根2的子树。对于其左子树， avail1[t[4]]<s[3]，故根为4的子树所包含的箱子均没有足够空间。因此，移到右子树 5，并将物品放入箱 3。图10-6d 给出了相应的结果。接下来，设s[4] = 3，从根为4的子树开始搜寻，因avail[t4]≥s[3]故将物品3加入箱子2。

根据前面的讨论，可以编写采用最先匹配策略的程序。主程序见程序 10-6。该程序首先要求输入物品数量n 及每个箱子的容量c。假定容量及物品的空间需求均为整数，并且假定程序只在n≥2 的情况下运行，因为n=0或1时装载方法显而易见。接下来程序要求输入n个物品并限制每个物品的空间≤c。最后，再调用函数FirstFitPack（见程序10-7），将物品分派到各个箱子中。对于使用FFD策略的程序，只需将源程序稍作修改，即在调用 FirstFitPack之前按递减顺序对物品进行排序。

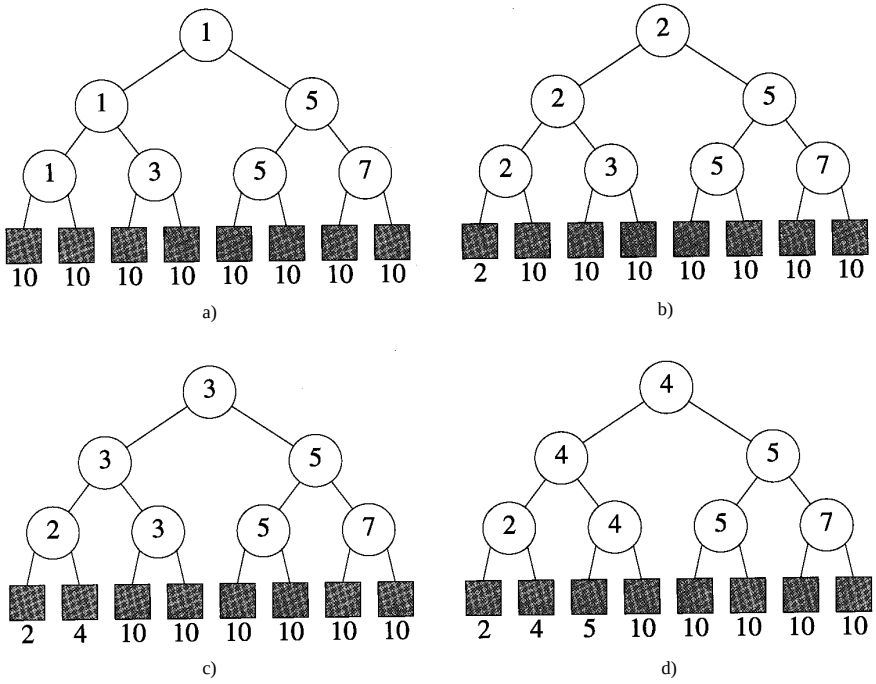


图10-6 最先匹配最大赢者树

a) 初始状态 b) 装载S[1]之后 c) 装载S[2]之后 d) 装载S[3]之后

程序10-6 最先匹配箱子装载法

```
void main(void)
{
    int n, c; // 物品数量和箱子容量
    cout <<  "Enter number of objects and bin capacity" << end1;
    cin >> n>> c;
    if (n< 2) {cout << "Too few objects" << end1;
              exit(1);}
    int *s = new int[n+1];

    for (int i = 1; i <= n;  i ++)  {
        cout  <<  "Enter space requirement of object" << i << end1;
        cin >> s[i] ;
        if ( s[i] > c)  {
              cout  << "Object  too  large  to  fit in a bin" << end1;
              exit (1) ; }
          }
        FirstFitPack(s, n, c);
    }
```

函数FirstFitPack首先对n 名选手的最大赢者树进行初始化。选手 i 代表箱子i当前的容量。所有箱子的容量初始化为 c。该函数假定，当比赛开始时左边选手是赢者，除非右边选手比较大。同时还假定WinnerTree（见程序10-1）的类定义中已增加一个共享成员函数：

```
int Winner(int i) const {return(i < n ) ? t[i] : 0;}
```

该共享函数用于返回在内部节点 i 比赛的赢者。在第二个 for 循环中，物品被依次分派到各箱子中。物品i 的分派过程，是按从根节点到满足该物品的最左箱子的路径进行的。从当前的位置可以判断左子树（根为 p）是否包含有足够容量的箱子。若无，则确保右子树（根为 p+1)包含这样的箱子。当然，将优先使用左子树中满足条件的箱子。一旦确定了要移到哪个子树，就把p 修改为其左子树的根，若当前节点的左子树为一外部节点时（即 p≥n），while循环将结束。注意到我们的程序代码不能准确地记录当前位置。不过，在退出 while循环时常用p 除以2来计算当前位置。当 n 为奇数时，当前位置可以是一个外部节点，这时 p 等于n。对于其他所有情况，p 均为内部节点。当p 为外部节点时，该节点对应的箱子是其父节点比赛的赢者，也就是说它是箱子t[p/2]。当p 为内部节点时，可以确信t[p]有足够容量。然而，倘若该箱子不是其父节点的左孩子，它可能不是最左箱子，故我们从该箱子的左边进行检查。一旦确定了用箱子b 来装载物品i，该箱子的可用容量应减少 s[i]，并且沿着赢者树中从该箱子到根的路径重新进行比赛。

程序10-7第二个for循环中的每次循环需Θ(logn)的时间，因此，该循环共需耗时Θ(nlogn)。该函数其余部分所需的时间为Θ(n)，所以总时耗为Θ(nlogn)。

程序10-7 函数FirstFitPack

```
void FirstFitPack(int s[ ], int n, int c)
{//c为箱子容量
 //n为物品数量， s[ ]为各物品所需要的空间

    WinnerTree<int> *W = new WinnerTree<int> (n);
    int *avail = new int[n+1];//箱子

    //初始化n个箱子和赢者树
    for (int i = 1; i <= n; i++)
        avail[i] = c; //初始可用的容量
    W-> Initialize( avail, n, winner);

    //将物品放入箱子中
    for (int i = 1; i <= n; i++) {//将s[i]放入箱子
        //找到有足够容量的第一个箱子
        int p = 2; //从根的左子树开始查询
        while (p < n) {
            int winp = W->Winner(p);
            if (avail[winp] < s[i]) //第一个箱子在右子树中
                p++;
            p *= 2; //移到左孩子
            }

        int b;
        p / = 2;
        if (p < n) {//在一树节点处
          b = W->Winner(p);
```

```
            //若b是右孩子，需要检查箱子b-1。
            //即使b是左孩子，这种检查也没有什么副作用
            if (b > 1 && avail[b-1] >= s[i])
              b - -;}
          else //当n为奇数时
            b = W->Winner (p/2) ;

        cout << "Pack object " << i << " in bin " << b << end1;
        avail[b] -= s[i]; //更新可用容量
        w->RePlay(b, winner);
        }
}
```

评价

函数FirstFitPack利用了在赢者树建立过程中所规定的某些特有的细节。如，赢者树为用数组表示的完全二叉树，因此能够按数组下标乘2或加1的方式在竞赛树中向下移动。按这种方式在竞赛树中向下移动破坏了使用类的一个目标——信息隐藏。我们希望类的实现细节应对用户透明。当用户与细节隔离时，我们可在保持类的共享成员不发生改变的情况下修改类的具体实现，而这种修改并不会影响应用的正确性。利用信息隐藏的特点，可以扩充 WinnerTree的类定义，增加从一个内部节点移动到其左、右孩子的共享函数，然后在函数 FirstFitPack中应用这些函数。

10.5.2 用相邻匹配法求解箱子装载问题

在相邻匹配（Next Fit）策略中，一次只将一个物品放入一个箱子中。开始时将物品 1放入箱子1。对于其余物品，则从最后使用的箱子的下一个箱子开始，用循环的方式轮流查询能够装载该物品的非空箱子。比如，在该轮询法中，若箱子 1～箱子b 正在使用，则可认为这些箱子排列成环状。i≠b 时，i 的下一个箱子为i +1 ；i =b时，i 的下一个箱子为箱子1。若上一个物品放入了箱子j，则从箱子j 的下一个箱子开始不断地查找后续箱子，直到找到具有足够空间的箱子或者又回到箱子j。若没有找到合适的箱子，则启用一新箱子，并将物品放入该箱子中。

例10-7 欲将6个物品s[1: 6] = [3, 5, 3, 4, 2, 1]放入容量为7的箱子中。用相邻匹配装载法，首先将物品1放入箱子1。物品2无合适的箱子，故插入一个新的箱子——箱子 2。对于物品3，从下一箱子开始搜寻非空的合适箱子。上一次使用的箱子为箱子 2，故下一个箱子为箱子 1。箱子1有足够的空间，所以将物品 3放入箱子1。对于物品4，因箱子1是上一次使用的箱子，所以从箱子2开始轮询。箱子 2无足够的空间，而箱子 2的下一个箱子（箱子 1）也无足够的空间，因此启用新箱子——箱子 3，并将物品4放入其中。装载物品 5的过程是从查寻箱子 3的下一个箱子开始的，箱子 3的下一个箱子为箱子 1，按上述步骤 ，可查知箱子2是合适的，因此将物品5放入箱子2。对于最后一个物品6，从箱子3开始检查，因该箱有足够空间，可将物品 6放入其中。

上述相邻匹配策略与另一种同名的动态存储分配策略很类似，即每次装载一个物品，若一个物品不能装入当前箱子，则将当前箱子关闭并启用一个新的箱子。本节不准备介绍这种匹配策略。

可用最大赢者树来高效地实现相邻匹配策略。与最先匹配法一样，外部节点代表各箱子，比赛是依据各箱子的最大容量来进行的。对于 n 个物品的装载问题，从 n 个箱子（外部节点）开始工作。观察图10-7的最大赢者树，其中有6/8的箱子已被使用，各标号的约定同图10-6。虽然当n = 8时，图10-7所示的情况不会出现，但它演示了如何确定装载下一个物品的箱子。若上一个物品被放入箱子last中且当前已使用了b个箱子，则搜索下一个可用的箱子可按如下两个步骤来进行：

1) 找到第一个箱子j，j >last。当箱子总数为n 时，这样的j 总存在。若该箱子非空（即j≤b），则使用之。

2) 若1) 未找到合适的箱子，则在树中搜索适合该物品的最左箱子，这个箱子是目前正在使用的箱子。

现在来考察图10-7中的情形。假设下一个物品需7个单元的空间。若last = 3，则在1) 中可确定箱子5有足够的空间。因箱子5是非空箱子，可将物品放入其中。另一方面，若last=5，则在1) 中获知箱子7有足够空间，因箱子7为空，故移到2)，找到有足够空间的最左箱子为箱子1，将物品放入其中。

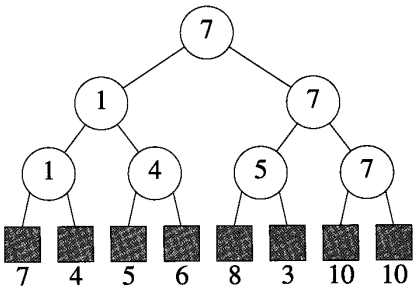


图10-7 相邻匹配法的最大赢者树

为了实现1)，从箱子j=last+1开始，其中last 为上一个箱子的编号。注意到若 last=n，则所有n个物品都已被装载并且使用了 n 个箱子，每个物品分别放在一个箱子中。因此 j≤n。图10-8的伪代码描述了从箱子j 开始查询合适箱子的过程。一般遍历从箱子j 到根的路径，查询各右子树直到找到第一个含有合适箱子的右子树。当找到该子树时，该子树中具有合适容量的最左箱子就是所查找的箱子。

```
//寻找距last的右孩子最近的箱子，用来存放物品 i
 j = last + 1;
 if (avail [j] >= s[i] ) return j;
 if (avail[j+1] >=s[i] ) return j+1;
 p = avail[j]的父节点 ;
 if (p  = = n-1 ) {// 特殊情况
   令 q 为t[p]右孩子中的外部节点;
   if (avail[q] >= s[i]) return q;}

//向树根移动，寻找第一个含有一个有足够容量的箱子的右子树
// p 的右子树为 p+1

p /= 2; //移动到父节点
while (avail[t[p+1]] < s[i])
     p /= 2;

return 子树 p+1中第一个能够存放物品i的箱子;
```

图10-8 查找合适箱子的伪代码

考察图10-7的赢者树。假设last=1且s[i]=7。从j=2开始，首先可以确定箱子2无足够的容量。接着，查询箱子j+1=3，它也没有足够的容量。因此移到j的父节点并取p等于4。因p≠n-1，我们到达while循环并得知根为5的子树不含合适的箱子。接下来，移到节点2并得知根3的子树包含合适的箱子。所需的箱子应是该树中容量大于或等于7的最左箱子。按程序10-7的策略可找到这个箱子——箱子5。若初始假设为last=3且s[i]=8，则将从箱子4开始查询。箱子4和箱子5都无足够的容量，故取p等于5并到达while循环。在第一次循环中检查avail[t[6]]，并得知根为6的子树不含合适的箱子。然后，p移到节点2，并得知根为3的子树包含合适的箱子。用程序10-7的策略可找到这个合适箱子——箱子7。因箱子7是空的，故移到2)，确定使用箱子7。

1) 要求我们按树的某条路径向下遍历以找到最左合适箱子，所需的时耗为O(log$n$)。利用程序10-7中的策略，2) 所需的时耗为Θ(log$n$)。因此相邻匹配策略总的时间复杂性为Θ(n log$n$)。

练习

7. 函数FirstFitPack（见程序10-7）将一个物品分配给一个箱子的时耗为Θ(log$n$)，即使当前所使用的箱子数目远远少于n。如果从包含箱子1和箱子b（b为当前已使用的最右箱子）的最小子树的根开始搜索，可以进一步减少分配一个箱子的时间，也就是说，从箱子1和箱子b的最近祖先开始搜索。例如，当b为3时，从节点2开始搜索。如果在箱子1到箱子b中没有一个箱子的容量符合条件，则将b增1。另外，在重新组织比赛时，只需重赛位于箱子1和箱子b的最近公共祖先之前的比赛。按照上述思想重写程序10-7，并在n=1000、5000、50 000、和100 000的情况下，比较前后两个程序版本的时间消耗。

8. 1) 扩充类WinnerTree，增加共享函数LeftChild（i）和RightChild（i），函数LeftChild（i）和RightChild（i）分别返回内部节点i的左右孩子，当i = 0时返回值均为0。

2) 重写FirstFitPack（见程序10-7），使之符合10.5.1节所述的信息隐藏原理。

9. 虽然证明最先匹配法和最优匹配法的箱子数不会超过$\lceil(17/10)b[I]\rceil$很困难（$b(I)$为实例I下所需的最少箱子数），但比较容易证明箱子数不会超过$2b(I)$，请证明之。

10. 最差匹配法（Worst Fit）是又一种箱子装载策略。同最先匹配法一样，一次只将一个物品放入箱子中。若要装载一个物品，则将其放入可选的具有最大容量的非空箱子中，如果没有这样的箱子则启用一新的箱子来装载此物品。最差匹配法可用最大堆来实现，相应的时间复杂性应为O($n$log$n$)，其中n为物品的数量。

1) 采用最差匹配法编写一个不同于函数FirstFitPack的函数WorstFitPack，使用一个初始为空的最大堆（即：没有非空箱子）。每次装载一个物品时，搜索具有最大可用空间的箱子，若该箱子无足够的空间则启用一个新箱子并将其加入堆中。

2) 比较在n=500、1000、2000和5000的情况下，最差匹配法和最先适配法各自所用的箱子数。

11. 1) 利用10.5.2节所述的两个步骤及图10-8中的伪代码，编写采用相邻匹配策略实现箱子装载的C++程序。

2) 利用随机产生的箱子装载实例比较相邻匹配策略和最先适配策略所需要的箱子数。