第五节：树（下）

5.1 堆

1.堆的介绍

优先队列：特殊的“队列”，取出元素的顺序是依照元素的优先权（关键字）大小，而不是元素进入队列的先后顺序。

堆的两个特性：

（1）结构性：用数组表示的**完全二叉树**；

（2）有序性：任一结点的关键字是其子树所有结点的的最大值（最小值）：

①最大堆：也称“大顶堆”：最大值

②最小堆，也称“小顶堆”：最小值。

类型名称：最大堆（MaxHeap）
数据对象集：一个有 N>0 个元素的最大堆 H 是一棵完全二叉树，每个结点上的元素值不小于其子结点元素的值。
操作集：对于任意最多有 MaxSize 个元素的最大堆 H ∈ MaxHeap，元素 item ∈ ElementType，主要操作有：

- MaxHeap Create(int MaxSize)：创建一个空的最大堆。
- Boolean IsFull(MaxHeap H)：判断最大堆 H 是否已满。
- Insert(MaxHeap H, ElementType item)：将元素 item 插入最大堆 H。
- Boolean IsEmpty(MaxHeap H)：判断最大堆 H 是否为空。
- ElementType DeleteMax(MaxHeap H)：返回 H 中最大元素(高优先级)。

2.最大堆的操作

（1）最大堆的创建

```
typedef  struct  HeapStruct  *MaxHeap;
struct  HeapStruct {
    ElementType *Elements;  /* 存储堆元素的数组 */
    int Size;                    /* 堆的当前元素个数 */
    int Capacity;              /* 堆的最大容量      */
};
MaxHeap Create( int MaxSize )
{          /* 创建容量为 MaxSize 的空的最大堆 */
MaxHeap H = malloc( sizeof( struct HeapStruct ) );
H->Elements = malloc( (MaxSize+1) * sizeof(ElementType));
H->Size = 0;
H->Capacity = MaxSize;
H->Elements[0] = MaxData;
   /* 定义“哨兵”为大于堆中所有可能元素的值，便于以后更快操作 */
return H;
}
```

（2）最大堆的插入

当插入一个元素时，我们首先要想到将元素插入到最右下角的位置。然而，所插入的值有可能比其父结点还要大，因此我们需要将其与父结点调换位置。若此时该元素仍然要比新位置的父结点大，那么继续调换位置，直到满足要求。代码实现如下所示：

```
void Insert( MaxHeap H, ElementType item )
{ /* 将元素 item 插入最大堆 H，其中 H->Elements[0]已经定义为哨兵 */
    int i;
    if ( IsFull(H) ) {
        printf("最大堆已满");
        return;
    }
  i = ++H->Size; /* i 指向插入后堆中的最后一个元素的位置 */
  for ( ; H->Elements[i/2] < item; i/=2 )
        H->Elements[i] = H->Elements[i/2]; /* 向下过滤结点 */
  H->Elements[i] = item; /* 将 item 插入 */
}
```

注意：H->Element[0] 是哨兵元素，它不小于堆中的最大元素，以便省去 i>=1 的语句，控制顺环结束。

（3）最大堆的删除

取出根结点（最大值）元素，同时删除堆的一个结点。

如图 1 所示。我们想取出最大值 58 这个点。我们需要做的是：

①把 58 取出，然后用最后一个元素来替补根这个位置。

②找出 31 的较大的孩子，然后跟 31 比较。发现孩子 44 要比 31 大，于是将 44 和 31 交换位置。

③再比较 31 和子孩子 35，发现 35 大于 31，因此交换 35 和 31 的位置。完成。

图 1

代码如下：

```
ElementType DeleteMax( MaxHeap H )
{  /* 从最大堆 H 中取出键值为最大的元素，并删除一个结点 */
    int Parent, Child;
    ElementType MaxItem, temp;
    if ( IsEmpty(H) ) {
        printf("最大堆已为空");
        return;
    }
    MaxItem = H->Elements[1]; /* 取出根结点最大值 */
    /* 用最大堆中最后一个元素从根结点开始向上过滤下层结点 */
    temp = H->Elements[H->Size--];
  for( Parent=1; Parent*2<=H->Size; Parent=Child ) {
      Child = Parent * 2;
        if( (Child!= H->Size) &&
     (H->Elements[Child] < H->Elements[Child+1]) )
            Child++;   /* Child 指向左右子结点的较大者 */
        if( temp >= H->Elements[Child] ) break;
```

```
    else   /* 移动 temp 元素到下一层 */
           H->Elements[Parent] = H->Elements[Child];
    }
    H->Elements[Parent] = temp;
    return MaxItem;
}
```

$T(N) = O(\log N)$

3.最大堆的建立

堆的一个重要应用就是**堆排序**。要进行排序，首先需要建立堆。

建立最大堆：将已经存在的 N 个元素按最大堆的要求存放在一个一位数组中。

方法 1：通过插入操作，将 N 个元素一个个相继插入到一个初始为空的堆中去，其时间代价最大为 O(NlogN)。不合适。

方法 2：在限行时间复杂度下建立最大堆。

（1）将 N 个元素按输入顺序存入，先满足完全二叉树的结构特性。

（2）调整各结点位置，以满足最大堆的有序特性。

要这样做，首先要寻找到最后一个由儿子的结点。由右往左，由下往上，直到根结点。

代码如下：

```
MaxHeap BuildMaxHeap( MaxHeap H )
{    /* 这里假设所有 H->Size 个元素已经存在 H->Elements[]中        */
     /* 本函数将 H->Elements[]中的元素调整，使满足最大堆的有序性 */
int Parent, Child, i;
ElementType temp;
for( i = H->Size/2; i>0; i-- ){ /*从最后一个结点的父结点开始 */
 temp = H->Elements[i];
 for( Parent=i; Parent*2<=H->Size; Parent=Child ) {
     /* 向下过滤 */
        Child = Parent * 2;
          if( (Child!= H->Size) &&
     (H->Elements[Child] < H->Elements[Child+1]) )
                Child++;   /* Child 指向左右子结点的较大者 */
          if( temp >= H->Elements[Child] ) break;
    else   /* 移动 temp 元素到下一层 */
             H->Elements[Parent] = H->Elements[Child];
  } /* 结束内部 for 循环对以 H->Elements[i]为根的子树的调整 */
  H->Elements[Parent] = temp;
 }
 return H;
}
```

5.2 哈夫曼树和哈夫曼编码

1.哈夫曼编码介绍

设一棵二叉树有 n 个叶子结点，每个叶子结点带有权值 w_k，从根结点到每个叶子结点的长度为 l_k，则每个叶子结点的带权路径长度之和就是这棵树的“带权路径长度（Weighted

Path Length，简称 WPL）”。

假设有 n 个权值$\{w_1, w_2, \cdots\cdots, w_n\}$，构造有 n 个叶子的二叉树，每个叶子的权值是 n 个权值之一。这样的二叉树也许可以构造多个，其中必有一个（或几个）是带权路径长度 WPL 最小的。达到 WPL 最小的二叉树就称为最优二叉树或哈夫曼树。

2.哈夫曼树的构造

每次把权值最小的两棵二叉树合并

```
typedef struct TreeNode *HuffmanTree;
struct TreeNode{
int Weight;
HuffmanTree Left, Right;
}
HuffmanTree Huffman( MinHeap H )
{    /* 假设 H->Size 个权值已经存在 H->Elements[]->Weight 里 */
   int i;   HuffmanTree T;
      BuildMinHeap(H); /*将 H->Elements[]按权值调整为最小堆*/
      for (i = 1; i < H->Size; i++) { /*做 H->Size-1 次合并*/
            T = malloc( sizeof( struct TreeNode) ); /*建立新结点*/
            T->Left = DeleteMin(H);
                        /*从最小堆中删除一个结点，作为新 T 的左子结点*/
            T->Right = DeleteMin(H);
                        /*从最小堆中删除一个结点，作为新 T 的右子结点*/
            T->Weight = T->Left->Weight+T->Right->Weight;
                        /*计算新权值*/
            Insert( H, T ); /*将新 T 插入最小堆*/
   }
   T = DeleteMin(H);
   return T;
}
```

3.哈夫曼树的特点

（1）没有度为 1 的结点；

（2）n 个叶子结点的哈夫曼树共有 2n-1 个结点；

（3）哈夫曼树的任意非叶节点的左右子树交换后仍是哈夫曼树。

5.3 集合

1.集合基本介绍

集合运算包括交、并、补、差以及判定一个元素是否是某一集合中等。逻辑上，可以用树结构表示集合，树的每个结点代表一个集合元素。

但是更好的方法是用数组存储形式。

数组中每个元素的类型描述为：

```
typedef struct {
      ElementType Data;
      int Parent;
} SetType;
```

数组每个分量都是一个结构，包含两个域：一个是 Data，代表结点的信息；一个是 Parent，是一个下标数值，指向父结点的下标，若为-1，代表其为根结点。这样，图 2 左侧的数组就可以用来表示右侧 3 棵树。

图 2

2.集合运算

（1）集合的查找操作（用根结点表示）

```
int Find( SetType S[ ], ElementType X )
{   /* 在数组 S 中查找值为 X 的元素所属的集合 */
    /* MaxSize 是全局变量，为数组 S 的最大长度 */
    int i;
    for ( i=0; i < MaxSize && S[i].Data != X; i++) ;
    if( i >= MaxSize ) return -1; /* 未找到 X，返回-1 */
    for( ; S[i].Parent > 0; i = S[i].Parent ) ;
    return i; /* 找到 X 所属集合，返回树根结点在数组 S 中的下标 */
}
```

这段代码的意思是，首先寻找到 X 的下标 i，如果 X 不在集合中，那么返回-1；若 X 在集合中，则需要返回其根结点的下标，这时候就利用若 i 的父结点大于等于 0，那么久把 i 的父结点赋值给 i，直到找到根结点，返回树根结点在数组 S 中的下标。

（2）集合的并操作

首先分别找到 X1 和 X2 两个元素所在集合树的根结点，如果它们不同根，则将其中一个根结点的父结点指针设置成另一个根结点的数组下标就行了。

```
void Union( SetType S[ ], ElementType X1, ElementType X2 )
{
    int Root1, Root2;
    Root1 = Find(S, X1);
    Root2 = Find(S, X2);
    if （ Root1 != Root2 ） S[Root2].Parent = Root1;
}
```

为了改善合并以后的查找性能，可以采用**小的集合合并到相对大的集合**中。为了区分小的和大的，所以可以将根结点的 Parent 由-1 改为-A，其中 A 代表该集合有多少元素。这样我们就可以知道哪个集合小，哪个集合大了。