5.1.1什么是堆

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）的大小，而不是元素进入队列的先后顺序。

若采用数组或链表实现优先队列

数组：

1. 插入--元素总是插入尾部
2. 删除--查找最大最小关键字，从数组中删去需要移动元素

链表

1. 插入--元素总是插入链表的头部
2. 删除--查找最大最小关键字，删去结点

有序数组

1. 插入--找到合适的位置，移动元素并插入
2. 删除--删去最后一个元素

有序链表

1. 插入--找到合适位置，插入元素
2. 删除--删除首元素或最后元素

是否可以用二叉树存储结构

- 二叉搜索树？
- 如果采用二叉树结构，应该更关注**插入**还是**删除**？
    - 树结点顺序怎么安排？
    - 树结构怎样？

> 应该更加关注如何删去最大值

优先队列的完全二叉树表示

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节点的最大值（或最小值）
    1. **最大堆（MaxHeap）**，也称为大顶堆：最大值
    2. **最小堆（MinHeap）**，也成为小顶堆：最小值

类型名称：**最大堆（MaxHeap）**

数据对象集：**完全二叉树**，每个结点的元素值不小于其子结点的元素值

操作集：最大堆　　，元素　　，主要操作有

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

5.1.2堆的插入

最大堆创建

```
typedef struct HeapStruct *MaxHeap;
typedef int ElementType;
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的最大堆 */
    MaxHeap H=malloc(sizeof(struct HeapStruct));
    H->Elements=malloc((MaxSize+1)*sizeof(ElementType));  /* 堆是从下标为1的地方开始存储的，所以MaxSize要+1 */
    H->Size=0;
    H->Capacity=MaxSize;
    H->Elements[0]=MaxData;
    /* 定义哨兵为大于最大堆中所有可能元素的值，以便于以后更快操作 */
    return H;
}
```

最大堆的插入

算法：将新增结点插入到从其父结点到根结点的有序序列中

```
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 */
}
```

5.1.3堆的删除

最大堆的删除

取出根结点元素，同时删除堆的一个结点

因为堆是用数组实现的，把最后一个结点提到头节点，这样能保证堆的结构仍然是完全二叉树

然后找出比根结点更大的孩子，逐渐更换，保证根节点是最大的，满足有序性

```
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("最大堆已空");
        return ;
    }
    MaxItem=H->Elements[1];  /* 取出根结点的最大值 */
    /* 用最大堆中最后一个元素从根节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;
}
```

5.1.4堆的建立

最大堆的建立

堆的应用：堆排序

需要先建堆

建立最大堆：将　个已经存在的元素按最大堆的要求存放在一个一维数组中

1. 通过插入操作，将　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
2. 在线性时间复杂度下建立最大堆
    1. 将　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

建堆时间复杂性：

树中各结点的高度和

```
typedef struct HNode *Heap; /* 堆的类型定义 */
struct HNode {
    ElementType *Data; /* 存储元素的数组 */
    int Size;          /* 堆中当前元素个数 */
    int Capacity;      /* 堆的最大容量 */
};
typedef Heap MaxHeap; /* 最大堆 */
typedef Heap MinHeap; /* 最小堆 */

#define MAXDATA 1000  /* 该值应根据具体情况定义为大于堆中所有可能元素的值 */

MaxHeap CreateHeap( int MaxSize )
{ /* 创建容量为MaxSize的空的最大堆 */

    MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));
    H->Data = (ElementType *)malloc((MaxSize+1)*sizeof(ElementType));
    H->Size = 0;
    H->Capacity = MaxSize;
    H->Data[0] = MAXDATA; /* 定义"哨兵"为大于堆中所有可能元素的值*/

    return H;
}

bool IsFull( MaxHeap H )
{
    return (H->Size == H->Capacity);
}
```

```
bool Insert( MaxHeap H, ElementType X )
{ /* 将元素X插入最大堆H，其中H->Data[0]已经定义为哨兵 */
    int i;

    if ( IsFull(H) ) {
        printf("最大堆已满");
        return false;
    }
    i = ++H->Size; /* i指向插入后堆中的最后一个元素的位置 */
    for ( ; H->Data[i/2] < X; i/=2 )
        H->Data[i] = H->Data[i/2]; /* 上滤X */
    H->Data[i] = X; /* 将X插入 */
    return true;
}

#define ERROR -1 /* 错误标识应根据具体情况定义为堆中不可能出现的元素值 */

bool IsEmpty( MaxHeap H )
{
    return (H->Size == 0);
}

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, X;

    if ( IsEmpty(H) ) {
        printf("最大堆已为空");
        return ERROR;
    }

    MaxItem = H->Data[1]; /* 取出根结点存放的最大值 */
    /* 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 */
    X = H->Data[H->Size--]; /* 注意当前堆的规模要减小 */
    for( Parent=1; Parent*2<=H->Size; Parent=Child ) {
        Child = Parent * 2;
        if( (Child!=H->Size) && (H->Data[Child]<H->Data[Child+1]) )
            Child++;  /* Child指向左右子结点的较大者 */
        if( X >= H->Data[Child] ) break; /* 找到了合适位置 */
        else  /* 下滤X */
            H->Data[Parent] = H->Data[Child];
    }
    H->Data[Parent] = X;

    return MaxItem;
}

/*------------ 建造最大堆 ------------*/
void PercDown( MaxHeap H, int p )
{ /* 下滤：将H中以H->Data[p]为根的子堆调整为最大堆 */
    int Parent, Child;
    ElementType X;

    X = H->Data[p]; /* 取出根结点存放的值 */
    for( Parent=p; Parent*2<=H->Size; Parent=Child ) {
        Child = Parent * 2;
        if( (Child!=H->Size) && (H->Data[Child]<H->Data[Child+1]) )
            Child++;  /* Child指向左右子结点的较大者 */
        if( X >= H->Data[Child] ) break; /* 找到了合适位置 */
        else  /* 下滤X */
            H->Data[Parent] = H->Data[Child];
    }
    H->Data[Parent] = X;
}

void BuildHeap( MaxHeap H )
{ /* 调整H->Data[]中的元素，使满足最大堆的有序性  */
  /* 这里假设所有H->Size个元素已经存在H->Data[]中 */

    int i;

    /* 从最后一个结点的父节点开始，到根结点1 */
    for( i = H->Size/2; i>0; i-- )
        PercDown( H, i );
}
```