```c
/*
优先队列
    特殊的“队列”，取出元素的顺序依照元素的优先权大小
    而不是元素进入队列的先后顺序

优先队列的实现
    可以采用数组、链表、有序数组、有序链表实现优先队列
    采用上述数据结构，插入和删除的时间复杂度互相影响，性能一般
    建议采用完全二叉树存储结构

堆
    结构性：用数组表示的完全二叉树
    有序性：任一结点的关键字是其子树所有结点的最大值（或最小值）
*/

#include<stdio.h>
#include<malloc.h>
#define MaxData 100000
#define ERROR -1

typedef int ElementType;
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); // 建堆
bool IsFull(MaxHeap H);      // 判断堆是否满
bool Insert(MaxHeap H,ElementType item); // 插入元素
bool IsEmpty(MaxHeap H);     //  判断堆是否为空
ElementType DeleteMax(MaxHeap H);        // 删除并返回堆中最大元素
void LevelOrderTraversal(MaxHeap H);     // 层序遍历

// 建堆
MaxHeap Create(int MaxSize)
{
    MaxHeap H = (MaxHeap)malloc(sizeof(struct HeapStruct));
    // Elements[0] 作为哨兵，堆元素从 Elements[1] 开始存放
    // "哨兵"大于堆中所有可能的值
    H->Elements = (ElementType *)malloc((MaxSize+1)*sizeof(ElementType));
    H->Elements[0] = MaxData;
    H->Size = 0;
    H->Capacity = MaxSize;
    return H;
}

// 插入：从完全二叉树的最后一个位置插入
// 因为哨兵结点作为根结点，取了一个最大值，所以新插入的结点不会转移到根结点
bool Insert(MaxHeap H, ElementType item)
{
    if(IsFull(H))
    {
        printf("堆已满，无法插入！\n");
        return false;
    }
    int i = ++H->Size; // 指向堆中最后一个位置，这儿不能改符号顺序
    // 有哨兵结点存在，不需要另外添加 i>1 的条件
    for(; item>H->Elements[i/2]; i/=2) // 向上找比 item 小的结点
        H->Elements[i] = H->Elements[i/2]; // 父结点向下赋值
    H->Elements[i] = item; // 找到结点最终插入的位置，把 item 值放进去
    return true;
}

// 删除：将根节点弹出
// 如果堆空，无法删除
// 如果堆不空
//     先得到堆的最大值，保存在 Max 中
//     然后用 tmp 得到堆的末尾元素，拟将其放在根结点处
//     tmp 的值不一定是此时堆中的最大值，通过循环寻找 tmp 应保存的位置，保存为 parent
//         在循环中，判断当前 parent 位置的元素是否有左孩子及右孩子
//         如果有孩子，找到值最大的孩子，保存为 child
```

```
//          如果 tmp 的值已经大于 child 的值，则当前 parent 的位置即为 tmp 的最终位置
//          否则，将 child 放在 parent 处，parent 移动到 child 位置，继续寻找
ElementType DeleteMax(MaxHeap H)
{
    int parent, child;
    ElementType Max, tmp;
    if(IsEmpty(H))
    {
        printf("堆为空，无法删除！\n");
        return ERROR;
    }
    Max = H->Elements[1];
    tmp = H->Elements[H->Size--];
    // 判别条件：parent 是否有左孩子结点
    for(parent=1; parent*2<=H->Size; parent=child)
    {
        // 令 child 等于左右孩子中值大的那一个
        child = 2*parent;  // 左孩子结点
        // child!=H->Size 表示 child 不为当前最后一个结点，即 parent 有右孩子结点
        if((child!=H->Size) && (H->Elements[child]<H->Elements[child+1]))
            child++;
        // 给 tmp 找个合适的位置
        // 如果当前左右孩子结点比 tmp 都小，说明 tmp 位置可以保存在当前 parent 位置
        if(tmp > H->Elements[child])
            break;
        else // 否则把较大的孩子结点提上来，parent自己继续下去找
            H->Elements[parent] = H->Elements[child];
    }
    H->Elements[parent] = tmp; // 在合适的位置把 tmp 放进去
    return Max;
}

// 判断堆满
bool IsFull(MaxHeap H)
{
    return (H->Size == H->Capacity);
}

// 判断是否为空
bool IsEmpty(MaxHeap H)
{
    return !H->Size;
}

// 层序遍历
void LevelOrderTraversal(MaxHeap H)
{
    int i;
    printf("层序遍历的结果是：");
    for(i = 1; i<=H->Size; i++)
    {
        printf("%d ", H->Elements[i]);
    }
    printf("\n");
}

int main()
{
    MaxHeap H;
    int MaxSize = 100;
    H = Create(MaxSize);
    Insert(H, 55);
    Insert(H, 66);
    Insert(H, 44);
    Insert(H, 33);
    Insert(H, 11);
    Insert(H, 22);
    Insert(H, 88);
    Insert(H, 99);
    /*
          99
         /  \
        88  66
```

```
147         / \   / \
148        55 11 22 44
149       /
150      33
151       */
152       LevelOrderTraversal(H);
153       DeleteMax(H);
154       LevelOrderTraversal(H);
155       DeleteMax(H);
156       LevelOrderTraversal(H);
157       DeleteMax(H);
158       LevelOrderTraversal(H);
159       DeleteMax(H);
160       LevelOrderTraversal(H);
161       return 0;
162   }
163
```