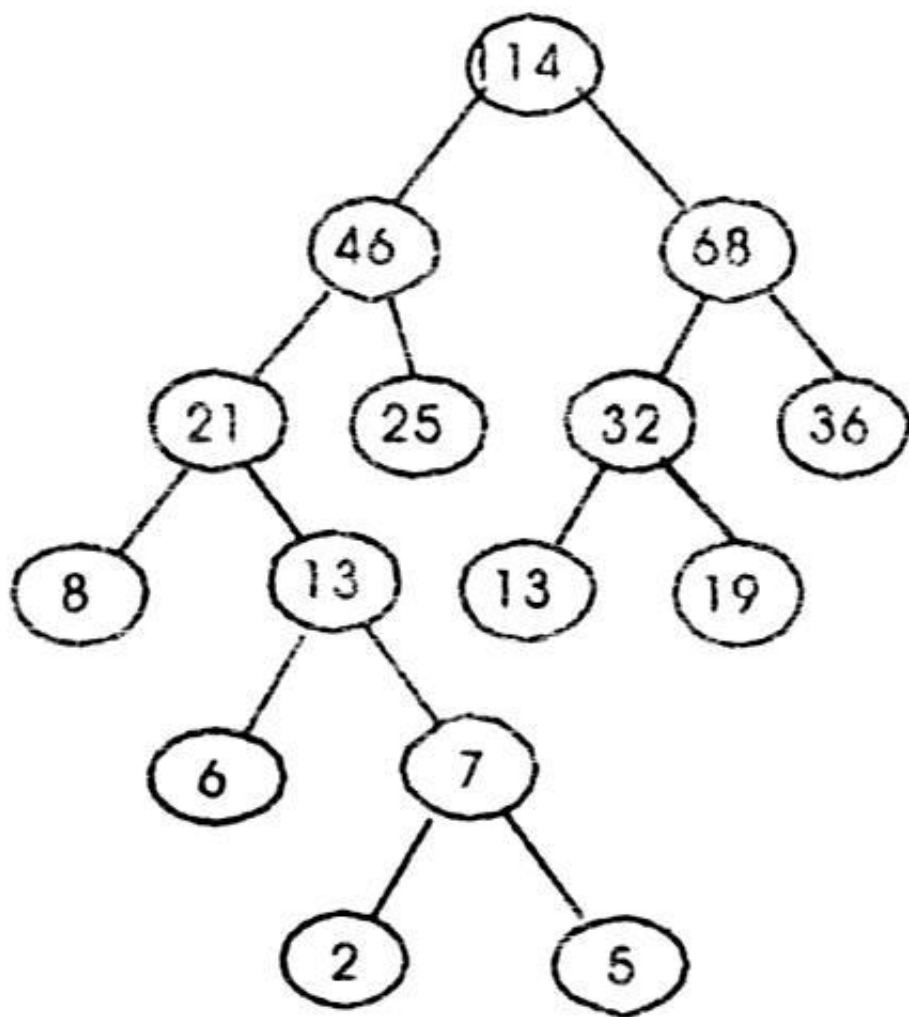


题目七 修理牧场

一.设计思路

分析题目要求不难发现，在给定的目标木头长度里，一段木头锯出时的费用开销与该段木头被锯出的次序直接相关。越晚被锯出，累积的费用开销就越大。要达到锯木头的总费用最小，需要先把大段的木头锯出，小段的木头留在后面。如果逆向来看，所求的锯木头顺序就对应一棵哈夫曼树。权重小的结点位于树的叶端，权重越大越靠近树根。而锯木头最终的费用消耗，就相当于这棵哈夫曼结点的权值累计。哈夫曼树结构结构如下图所示：



为了构造哈夫曼树，需要从木头长度的数据集中每次取出最权值最小的两端（即长度最短的两段），将其权值之和作为新的木头段插入数据集中。一种简单的思路是用一个向量来存储所有数据，并将所有数据进行排序，每次取出首或尾位置的两个元素即可。然而这个数据集具有动态变化的特点，需要不断执行插入和删除。在这种情况下，数组要想维持有序性，其时间开销显然是巨大的。

从本质上讲，我们所需要的是一个优先队列来辅助构建哈夫曼树。而优先队列的较好实现方式就是“堆”。在堆的基础上进行的插入和删除操作都被优化到了 $(\log n)$ 数量级。

二.数据结构实现

1.堆结点类型（member）

定义了结构体 `struct Node` 作为堆的结点，存储堆内元素以及堆的大小等辅助信息。

2.堆类型（Heap）

定义了 `Node` 的指针 `typedef Node* Heap;` 作为堆类型。

3.类成员

```
int* data;  
int size;  
int capacity;
```

`data` 为 `int *` 类型，指向一个数组，存储堆中所有数据。

`size` 存储堆中元素的个数。

`capacity` 存储 `data *` 中已经分配的空间大小。

4.建堆函数

```
Heap creat(int size); //建立一个新的堆
```

建堆函数，用于新建立一个空的堆，堆的预计大小由参数 `int size` 给出。

5.插入函数

```
Heap insert(Heap h,int x); //向堆中插入新元素
```

插入函数，向堆中插入一个新元素，元素权值由参数传入。

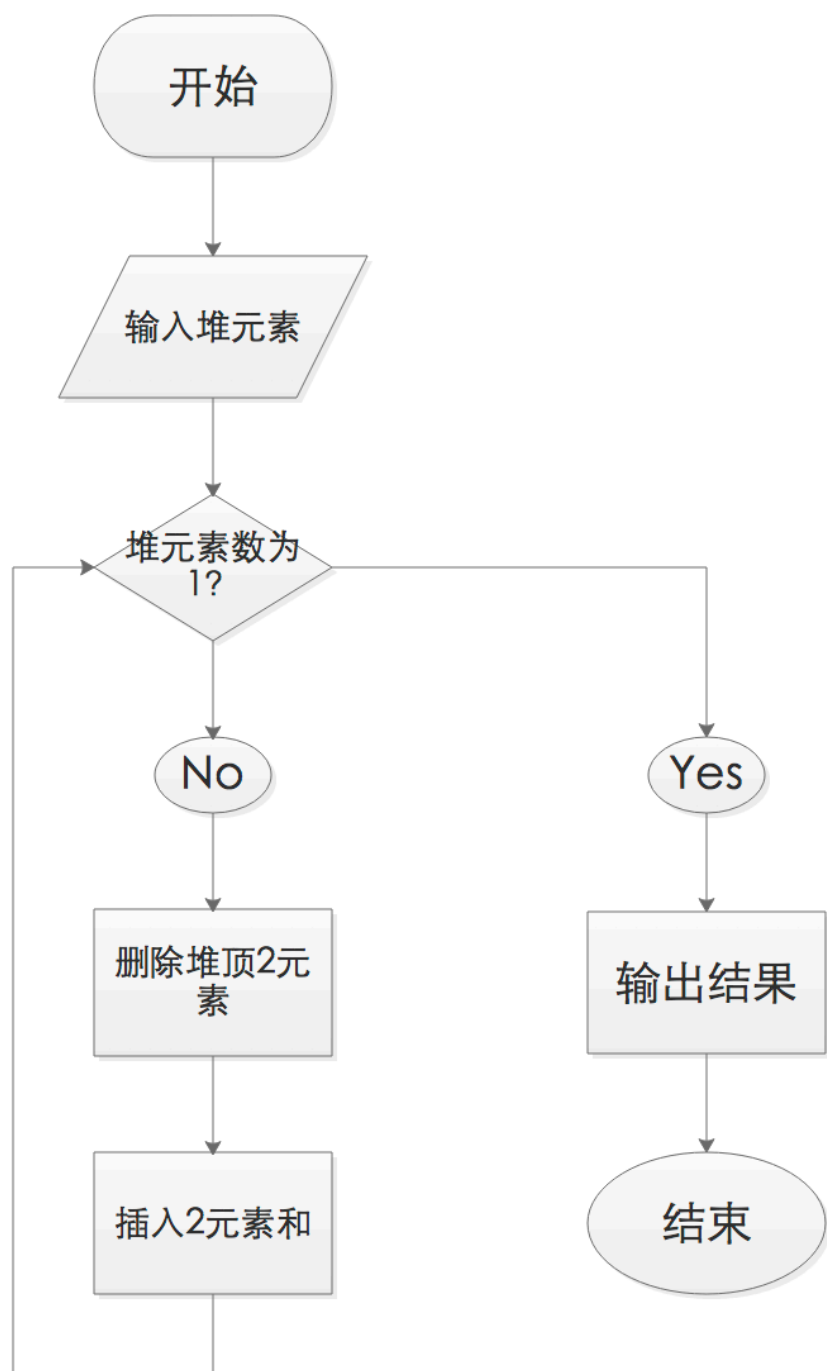
6.删除函数

```
int delet(Heap h); //删除函数
```

删除函数，删除堆顶节点，返回该节点权值。

三.系统实现

1.系统执行框架



首先由用户输入堆中各个元素的值，将其插入堆中，构建一个最小堆。

核心代码如下：

```
int n,x;
cin>>n;      //输入堆的最大规模
Heap hp = creat(n);    //建立新的堆
for(int i = 0;i < n;++i){    //循环向堆中插入元素
    cin>>x;
    insert(hp,x);
}
```

程序执行情况如下：



随后开始模拟哈夫曼树的构建过程。不断取出堆顶两个元素，将其求和后计入累加和，然后继续压入堆中。直到堆中只剩下一个元素，循环终止。此时的累加和即位锯木头的总开销。

核心代码如下：

```
int answer = 0;    //记录锯木头费用
while(hp->size > 1){
    int p = delet(hp); //连续两次取出堆顶元素
    p += delet(hp);
    answer += p;    //记录累计和
    insert(hp,p);   //将本次产生的元素插入堆
}

cout<<answer<<endl;    //输出答案
```

程序执行情况如下：



```
Run proj7
/Users/lixiangzhen/CLionProjects/proj7/cmake-build-debug/proj7
3
8 7 5
32
Process finished with exit code 0
```

2.建堆功能

核心代码如下：

```
/*
 * 新建立一个空的堆
 * 堆的预计大小由参数 ` int size ` 给出
 * */
Heap creat(int size){
    Heap h = new Node;
    h->data = new int[size + 1];
    h->size = 0;
    h->capacity = size;
    h->data[0] = -10006;

    return h;
}
```

- 根据参数大小新建数组
- 初始化堆的大小
- 置堆顶标记为一个极小数

关联调用情况如下：

```
87     int n,x;
88     cin>>n;      //输入堆的最大规模
89     Heap hp = creat(n);    //建立新的堆
90     for(int i = 0;i < n;++i){    //循环向堆中插入元素
91         cin>>x;
92         insert(hp,x);
```


3.插入功能

核心代码如下：

```
/*
 * 插入函数
 * 向堆中插入一个新元素
 * 元素权值由参数传入
 * */
Heap insert(Heap h,int x){

    if(h->size == h->capacity){
        cout<<"Heap is FULL\n";
        return h;
    }

    int i = ++h->size;

    for( ; h->data[i/2] > x; i /= 2){
        h->data[i] = h->data[i/2];
    }
    h->data[i] = x;
    return h;
}
```

- 判断堆是否满了
- 新节点插在尾部
- 向上过滤寻找合适位置
- 插入到正确位置

关联调用情况如下：

```
90     for(int i = 0; i < n; ++i){           //循环向堆中插入元素
91         cin >> x;
92         insert(hp, x);
93     }
94
```

```
8     p += delet(hp);
9     answer += p;    //记录累计和
0     insert(hp, p);  //将本次产生的元素插入堆
1 }
2
```

4.删除功能

核心代码如下：

```

/*
*删除函数
*删除堆顶节点
*返回该节点权值
* */
int delet(Heap h){
    if(h->size == 0){
        cout<<"堆为空! \n";
        return -1;
    }
    int result = h->data[1];
    int tag = h->data[h->size];
    --h->size;
    int p,c;
    for(p = 1;p*2 <= h->size;p = c){
        c = p*2;
        if(c + 1 <= h->size && h->data[c] > h->data[c + 1])
            ++c;
        if(h->data[c] >= tag)
            break;
        else
            h->data[p] = h->data[c];
    }
    h->data[p] = tag;
    return result;
}

```

- 判断堆是否为空
- 记录堆顶元素
- 将堆尾元素插到堆顶
- 向下过滤调整

关联调用情况如下：

```
int answer = 0;    //记录锯木头费用
while (hp->size > 1){
    int p = delet(hp); //连续两次取出堆顶元素
    p += delet(hp);
    answer += p;    //记录累计和
}
```

5.测试辅助函数

核心代码如下：

```

/*
 * 测试辅助函数
 * 对于建堆，堆的删除，堆的插入等操作进行测试
 * */
void test(){
    cout<<"输入测试元素个数：\n";
    int n,x;
    cin>>n;
    cout<<"输入各个元素值：\n";
    Heap hp = creat(n);        //建立新的堆
    for(int i = 0;i < n;++i){    //循环向堆中插入元素
        cin>>x;
        insert(hp,x);
    }

    cout<<"堆中元素序列为：\n";
    for(int i = 1;i <= n;++i)
        cout<<hp->data[i]<<" ";
    cout<<endl;

    cout<<"被删除的堆顶元素为："<<delet(hp)<<endl;
    cout<<"删除后堆中元素为："<<endl;
    for(int i = 1;i < n;++i)
        cout<<hp->data[i]<<" ";
    cout<<endl;
    cout<<"请输入要新插入的元素\n";
    cin>>x;
    insert(hp,x);
    cout<<"插入后堆中元素为：\n";
    for(int i = 1;i <= n;++i)
        cout<<hp->data[i]<<" ";
    cout<<endl;
}

```

- 测试建堆，堆的删除，堆的插入等操作
- 在正式程序中不调用

四.测试

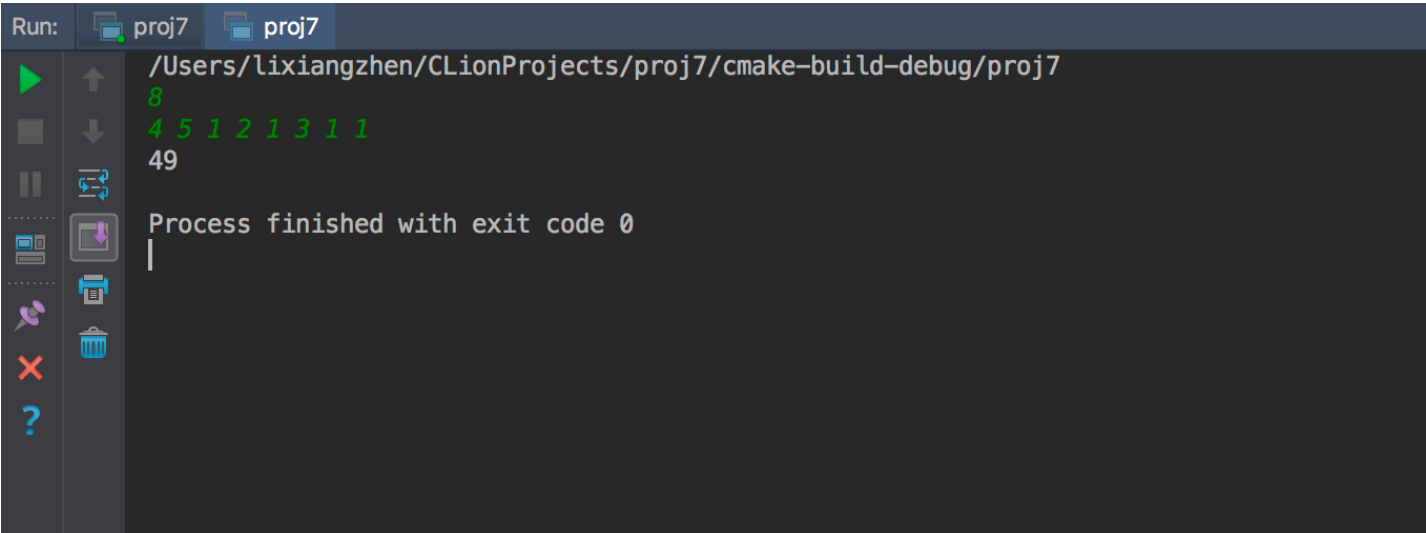
1.基本功能测试

测试用例

木头数目： 8

目标长度： 4 5 1 2 1 3 1 1

程序执行情况如下：



```
Run: proj7 proj7
/Users/lixiangzhen/CLionProjects/proj7/cmake-build-debug/proj7
8
4 5 1 2 1 3 1 1
49
Process finished with exit code 0
```

2.建堆功能测试

测试用例

元素数目： 6

元素： 2 3 1 77 5 9

程序执行情况如下：

```
Run: proj7 proj7 proj7 122 cin>>x;
/Users/lixiangzhen/CLionProjects/proj7/cmake-build-debug/proj7
输入测试元素个数:
6
输入各个元素值:
2 3 1 77 5 9
堆中元素序列为:
1 3 2 77 5 9
```

3.堆删除功能测试

测试用例

元素数目: 6

元素: 2 3 1 77 5 9

程序执行情况如下:

```
Run: proj7 proj7 proj7 proj7
/Users/lixiangzhen/CLionProjects/proj7/cmake-build-debug/proj7
输入测试元素个数:
6
输入各个元素值:
2 3 1 77 5 9
堆中元素序列为:
1 3 2 77 5 9
被删除的堆顶元素为: 1
删除后堆中元素为:
2 3 9 77 5
```

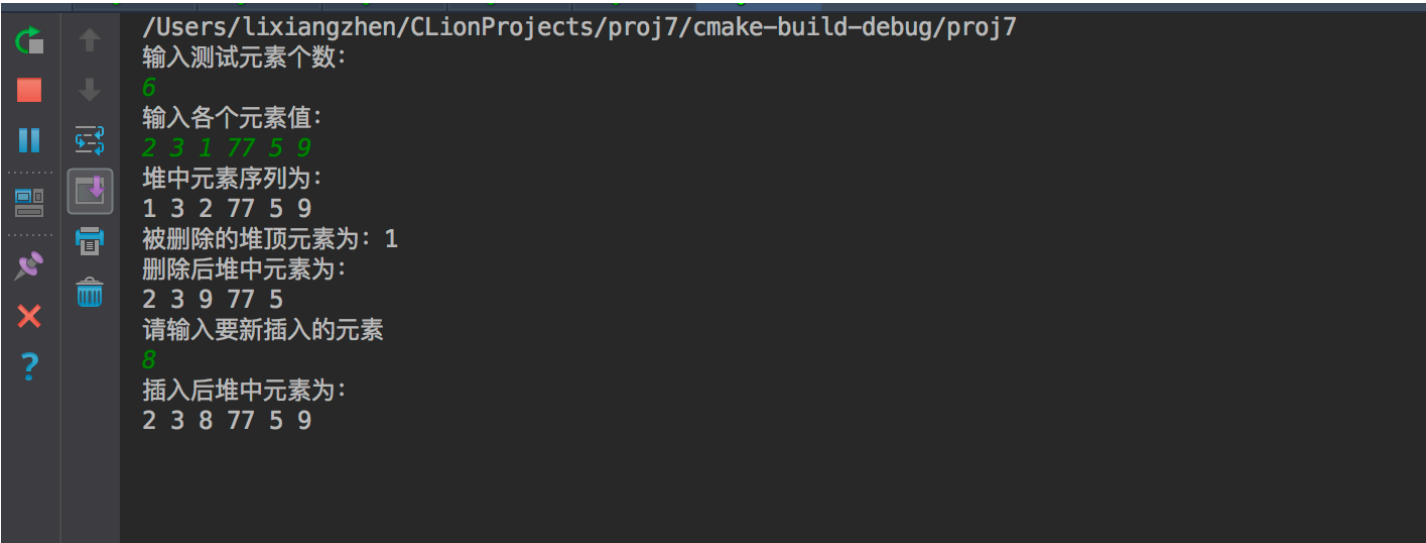
4.堆删除功能测试

测试用例

元素数目：

元素：

程序执行情况如下：



```
/Users/lixiangzhen/CLionProjects/proj7/cmake-build-debug/proj7
输入测试元素个数:
6
输入各个元素值:
2 3 1 77 5 9
堆中元素序列为:
1 3 2 77 5 9
被删除的堆顶元素为: 1
删除后堆中元素为:
2 3 9 77 5
请输入要新插入的元素
8
插入后堆中元素为:
2 3 8 77 5 9
```

5.边界测试

测试用例

木头数目：

目标长度：

程序执行情况如下：

