第7章 完全二叉堆

上一章实际讲的其实是词典，哈希表只是一种实现，
本章实际讲的是优先级队列，完全二叉堆也只是一种实现，
因此，我们从要实现的优先级队列讲起

一、优先级队列的定义

由前面几个章节的学习经验，一个数据结构的定义必定包含以下几个方面：
1.数据以什么形式存储（向量？列表？某种树？）
2.这些数据具备什么样的性质（先进先出？先进后出？从小到大？头大脚轻？）
3.实现了什么样的接口（增删查改？遍历？）

优先级队列的定义：
1.首先容器中的数据以某种优先级次序进行排列（这里先理解为数值大就是优先级高）
2.必须实现下面几个接口：
元素入队按优先级插入到合适的位置，最高优先级的元素出队，获取最高优先级的元素

优先级队列

```
template <typename T> struct PQ { //priority queue
    virtual void insert( T ) = 0;
    virtual T getMax() = 0;
    virtual T delMax() = 0;
}; //作为ADT的PQ有多种实现方式，各自的效率及适用场合也不尽相同
```

- Stack和Queue，都是PQ的特例——优先级完全取决于元素的插入次序
- Steap和Queap，也是PQ的特例——插入和删除的位置受限

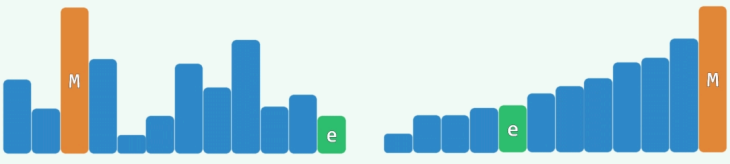
In [1]:
```
#include <iostream>
using namespace std;
```

In [2]:
```
template <class T>
struct PQ{
    virtual void insert(T) = 0;
    virtual T getmax() = 0;
    virtual T delmax() = 0;
};
```

二、优先级队列的向量实现和列表实现

向量实现
如果数据用一个向量存储，那么这个向量必须维护从小大大的顺序这么一个性质，
首先最后一个向量就是最大元素，访问和删掉都是O(1)的复杂度，但是新元素入队，
需要插入到合适的位置，保证有序这么一个性质，需要花费O(n)的复杂度，不是很好

Vector



Vector	getMax()	delMax()	insert()
Unsorted	traverse() Θ(n)	remove(traverse()) Θ(n) + O(n) = Θ(n)	insertAsLast(e) O(1)
Sorted	[n - 1] O(1)	remove(n - 1) O(1)	insert(1 + search(e), e) O(logn) + O(n) = O(n)

链表实现
操作的理论复杂度和向量一致，但是由于内存地址不连续，无法利用到缓存机制，所以也不好

List	getMax()	delMax()	insert()
Unsorted	traverse() Θ(n)	remove(traverse()) Θ(n) + O(1) = Θ(n)	insertAsFirst(e) O(1)
Sorted	first() O(1)	remove(first()) O(1)	insert(search(e), e) O(n) + O(1) = O(n)

建表复杂度
假如此时有n个元素，需要立刻变成优先级队列，使用向量和链表的实现方式一个一个插入，
相当于把插入算法实现了一遍，时间复杂度为O(n^2),这个复杂度无法接受，当然也可以
先按照无序存储，再按照归并排序或者快排实现元素有序，建表复杂度确实可以接受，
但是既然要用这个优先级队列，肯定它的基本插入删除是频繁使用的，这个插入操作复杂度不能接受
那么有没有一种数据结构，建立它不超过O(nlogn)的复杂度，每种操作不超过O(logn)的复杂度呢?
答案是肯定的，AVL树，Splay数，红黑树，都行，不过这里不讲这么高级的数据结构，这里使用
的是一个头大脚轻的完全二叉树，也叫完全二叉堆，大顶堆

三、优先级队列的完全二叉堆实现

完全二叉堆的定义及存储

在逻辑上，这是一颗完全二叉树，但是可以用向量进行存储，实现物理上的连续性，
实现方式如下，对于任何一个节点i，它的父亲，左孩子，右孩子的下标按如下公式直接计算

In [3]:
```
#define Parent(i) ( ((i) - 1) >> 1 )
#define LChild(i) ( 1 + ((i) << 1) )
#define RChild(i) ( (1 + (i)) << 1 )
#define HasParent(i) ( (((i) - 1) >> 1) > -1 )
#define HasLChild(i) (LChild(i) < size)
#define HasRChild(i) (RChild(i) < size)
#define HasChild(i) (HasLChild(i) || HasRChild(i))
```

In [4]:
```
(0-1) >> 1
```

Out[4]:
```
-1
```

In [5]:
```
(7-1) >> 1
```

Out[5]:
```
3
```

In [6]:
```
HasParent(0)
```

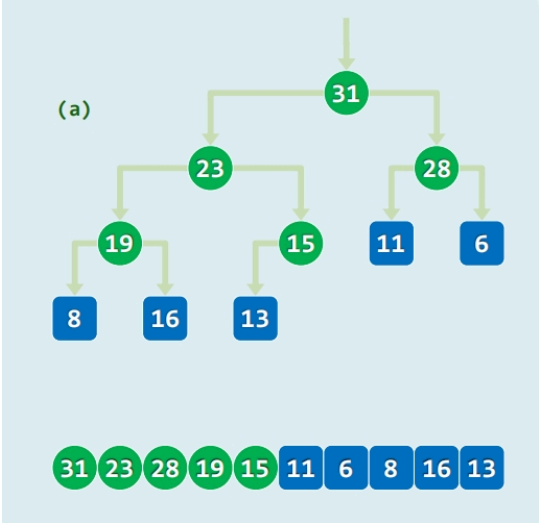
Out[6]:
```
false
```

In [7]:
```
HasParent(1)
```

Out[7]:
```
true
```

堆序性

简单来说就是头大脚轻，父亲的值 > 左右孩子的值，如下



完全二叉堆的实现

直接继承向量继续写它独有的接口

In [8]:
```
#include "vector.h"
template<class T>
class BinaryHeap: public PQ<T>, public Vector<T>{
    public:
        using Vector<T>::size;
        using Vector<T>::data;
        using Vector<T>::insert;

        BinaryHeap():Vector<T>(0){}   // 构造函数，初始化一个空堆

        // 优先级队列必须实现的接口
        void insert(T);
        T getmax();
        T delmax();

};
```

访问最大值

最大值就是二叉树的根，也就是向量的首元素

In [9]:
```
template<class T>
T BinaryHeap<T>::getmax(){
    return data[0];
}
```

按优先级插入元素

首先将元素插入最后一个位置，然后跟父亲比较，如果比父亲大破坏了堆序性，则交换

In [10]:
```
template<class T>
void BinaryHeap<T>::insert(T e){
    insert(size,e);    // 先插入到最后一个位置
```

```
    // 下面进行元素上浮，保证堆序性
    int i = size - 1;
    int j;
    while(HasParent(i)){
        j = Parent(i);
        if (data[i] > data[j]) {
            swap(data[i],data[j]);
            i = j;
        }
        else break;
    }
}
```

删除最大元素

直接将最后一个元素覆盖到第一个元素上，然后跟两个孩子的最大值比较，如果比孩子小，和最大的孩子交换

In [11]:
```
template<class T>
T BinaryHeap<T>::delmax(){
    T max_ = data[size - 1];
    data[0] = data[size - 1];
    size--;

    // 下面进行元素下潜
    int i = 0;
    int j;
    while(HasChild(i)){
        if (HasLChild(i) && HasRChild(i)){
            j = data[LChild(i)] > data[RChild(i)] ? LChild(i) : RChild(i);
            if (data[i] < data[j]) {
                swap(data[i],data[j]);
                i = j;
            }
        }
        else if (HasLChild(i)){
            j = LChild(i);
            if (data[i] < data[j]) {
                swap(data[i],data[j]);
                i = j;
            }
        }
        else{
            j = RChild(i);
            if (data[i] < data[j]) {
                swap(data[i],data[j]);
                i = j;
            }
        }
    }
    return max_;
}
```

使用插入构造一个完全二叉堆

实际建堆还有别的更好的办法，这里不提，因为这里仅仅理解优先级队列是怎么回事

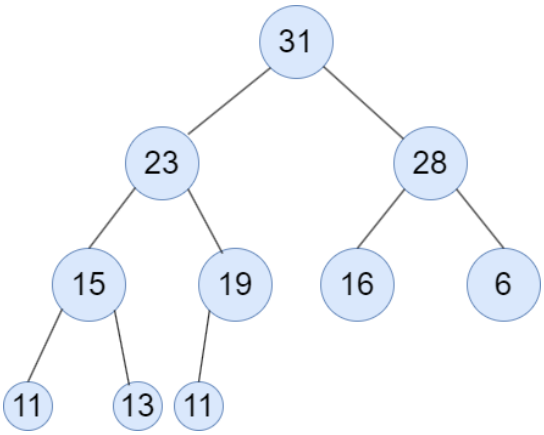
In [12]:
```
BinaryHeap<int> heap1;
heap1.size
```

Out[12]: 0

In [13]:
```
heap1.insert(23);
heap1.insert(28);
heap1.insert(31);
heap1.insert(13);
heap1.insert(6);
heap1.insert(16);
heap1.insert(6);
heap1.insert(11);
heap1.insert(15);
heap1.insert(19);
```

In [14]:
```
for (int i = 0; i < heap1.size; i++)
    cout << heap1.data[i] << " ";
```

```
31 23 28 15 19 16 6 11 13 6
```



由上图可以看出，这个堆完全满足堆序性

删除最大元素

In [15]:
```
heap1.delmax();
for (int i = 0; i < heap1.size; i++)
    cout << heap1.data[i] << " ";
```

```
28 23 16 15 19 6 6 11 13
```

获取当前队列最大元素

In [16]:
```
heap1.getmax()
```

Out[16]: 28

四、堆排序

堆排序的步骤是这样的，首先将你的数据使用大顶堆结构保存，这步使用模板库效率为O(n),
然后采用选择排序的思路，把依次取出最大元素放在后面，这里就是把堆顶和堆尾交换，然后堆size-1，
这时候堆就分成两部分了，前面的堆是待排序的，后面的是已经排好序的，别忘了把交换到堆顶的元素下潜恢复堆序性。
接着执行前面同样的步骤即可完成，因为每步选最大元素仅O(1),时间花在下潜上了，为O(logn)，然后
所以对所有元素都执行这些动作的总复杂度就是O(nlogn)，综合前面建堆的效率，因此整个算法的复杂度就是O(nlogn)

由于我们前面写的下潜算法针对整个堆，因此这里需要修改一下，需要支持下潜范围是前面多少个元素，
所以里面的判断有没有孩子的范围也必须改，HasChild(i)需要改成HasChild(i,k)，下潜算法是针对
前k个元素组成的堆进行下潜。原理很简单，这里就不重构上面的算法了。

In [ ]: