基础数据结构

湖南师大附中 许力

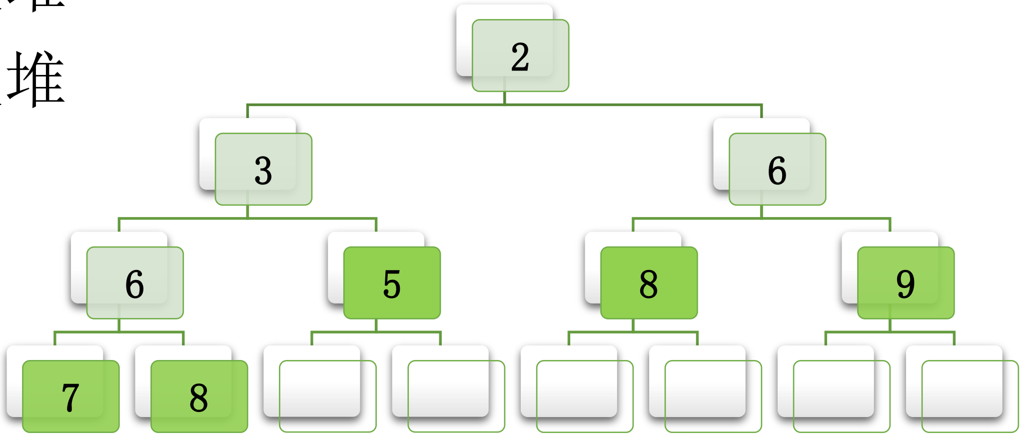
堆（heap）

- 是满足特殊性质的完全二叉树：

1. 若左/右子树不空，则左/右子树均不大于/不小于父节点
2. 左、右子树均符合递归定义

根节点是（所有节点中）最小值的是小根堆

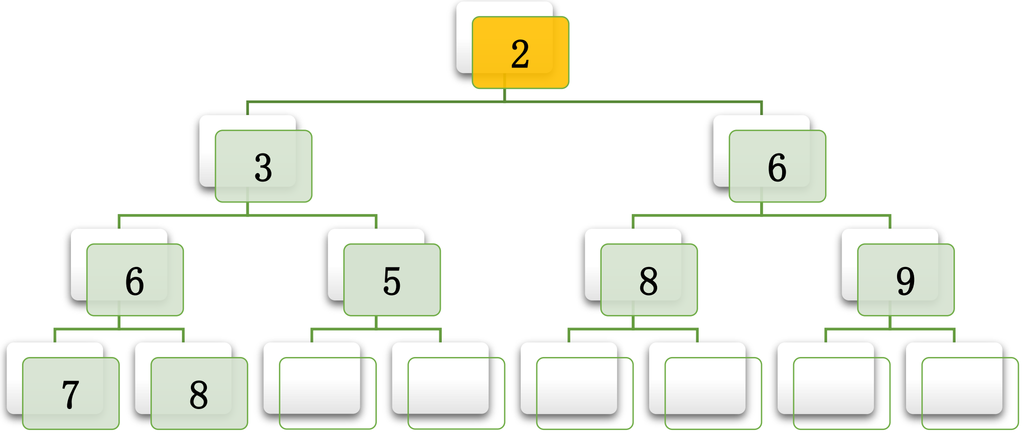
根节点是（所有节点中）最大值的是大根堆



堆

- 根节点（堆顶）是堆中唯一具有价值的节点

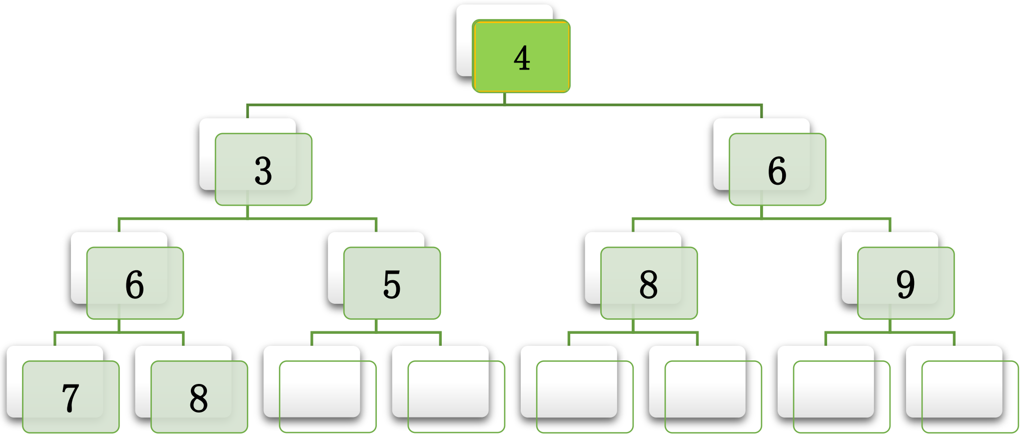
- 堆的意义在于可以在O(logn)的复杂度内维护并查询区间最值
- 比如[2, 3, 6, 6, 5, 8, 9, 7, 8]，建成堆后直接取堆顶即可



堆

- 不仅仅可以查询，堆还可以动态维护区间最值！
- 这才是堆存在的真正意义。若只为了静态查询，建堆必要性不够

1. 将根节点修改为4，再询问最小值



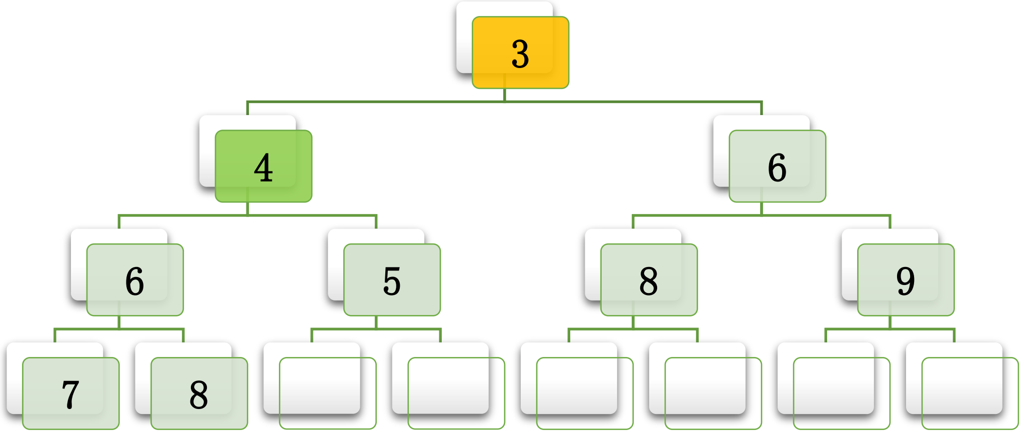
堆

- 不仅查询，还可以动态维护区间最值！
- 这才是堆存在的真正意义，若只为了静态查询，建堆必要性不够

1. 将根节点修改为4，再询问最小值

4≥3（leftchild），破坏了小根堆的性质，需要交换4和3

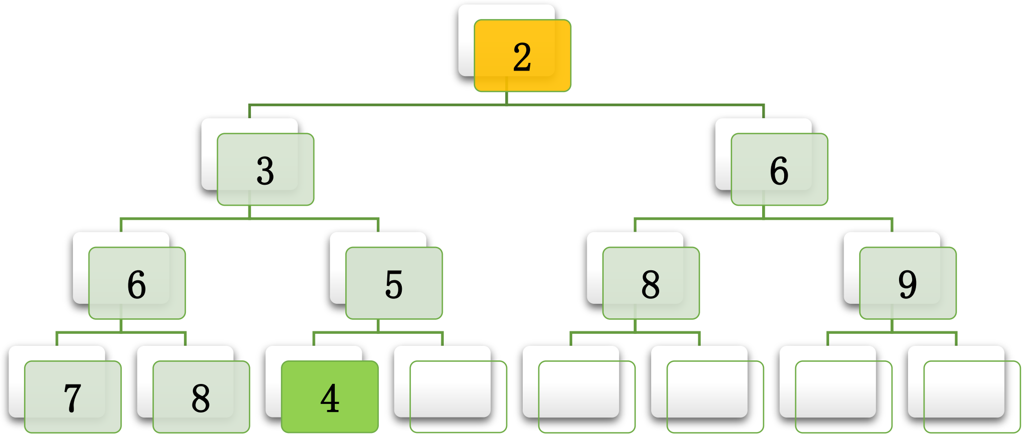
维护后的根节点即为新的最小值



堆

- 不仅查询，还可以动态维护区间最值！
- 这才是堆存在的真正意义，若只为了静态查询，建堆必要性不够

2. 插入新节点4，再询问最小值



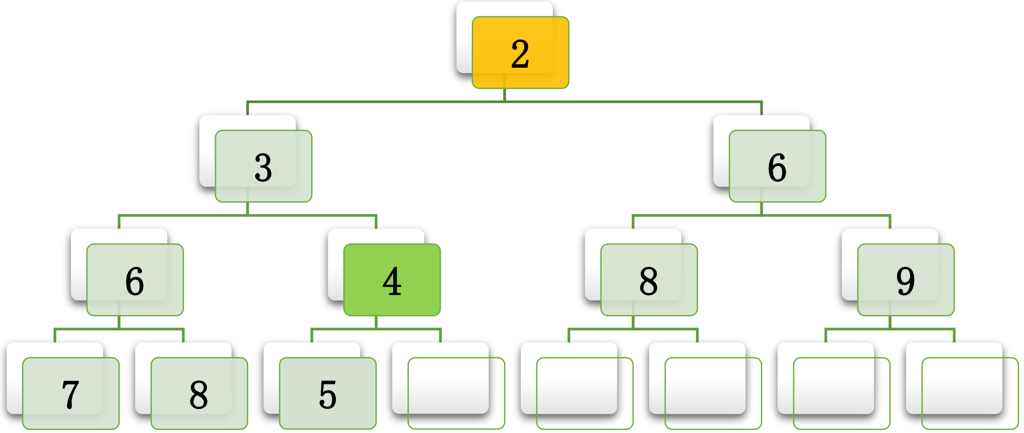
堆

- 不仅查询，还可以动态维护区间最值！
- 这才是堆存在的真正意义，若只为了静态查询，建堆必要性不够

2. 插入新节点4，再询问最小值

4≤5（father），破坏了小根堆的性质，需要交换4和5

维护后的根节点即为新的最小值



堆的代码实现

1. 数组模拟堆

我们之前的堆，用数组模拟的话长这样：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	6	6	5	8	9	7	8						

堆也是树，这里为什么用一维数组模拟，而不是开结构体记录左右儿子节点等？

堆的代码实现

1. 数组模拟堆

我们之前的堆，用数组模拟的话长这样：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	6	6	5	8	9	7	8						

堆的操作主要就是两个：

1. 插入一个新节点
2. 删除一个已有节点

堆的代码实现

1. 数组模拟堆

我们之前的堆，用数组模拟的话长这样：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	6	6	5	8	9	7	8						

特别的，建堆可以被视为从空开始，不断执行插入新节点操作

数组模拟堆

- 堆的操作主要就是两个：

1. 插入一个新节点
2. 删除一个已有节点

- 而在这两个基本操作的过程中，需要不断向上或者向下调整堆（否则堆的性质就被破坏，从而失去作用）

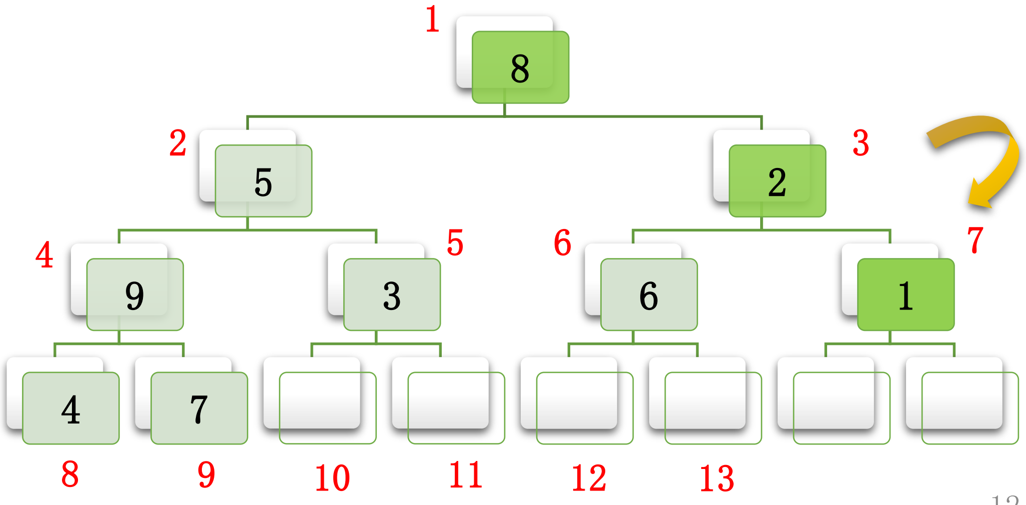
数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

1. 向上调整：

　　从当前节点开始，和它的父节点比较，若是比父节点小，就交换，然后将当前节点下标修改为其原父节点下标

　　比如当前节点7，1<2，那么交换1和2，并将当前节点下标修改为3



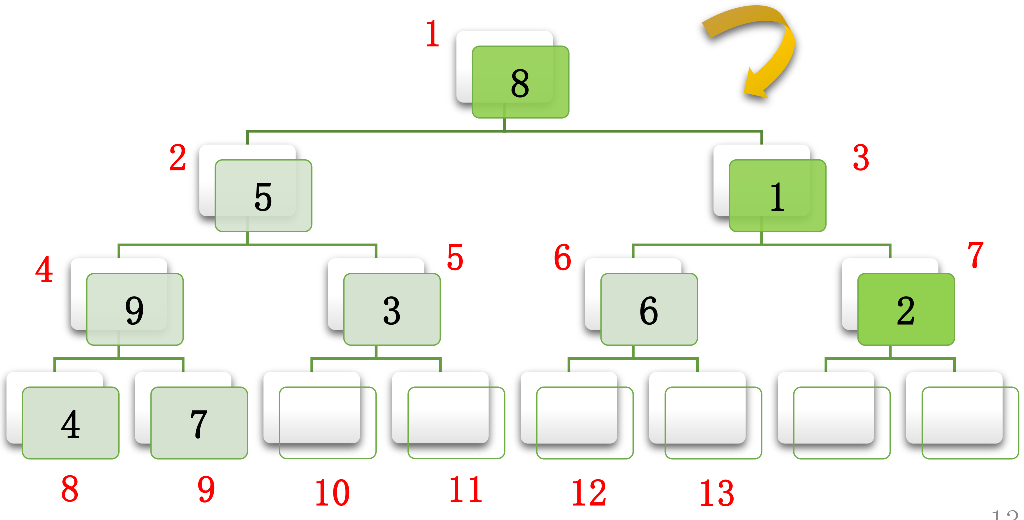
数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

1. 向上调整：

　　从当前节点开始，和它的父节点比较，若是比父节点小，就交换，然后将当前节点下标修改为其原父节点下标

　　当前节点3，1＜8，那么交换1和8，并将当前节点下标修改为1

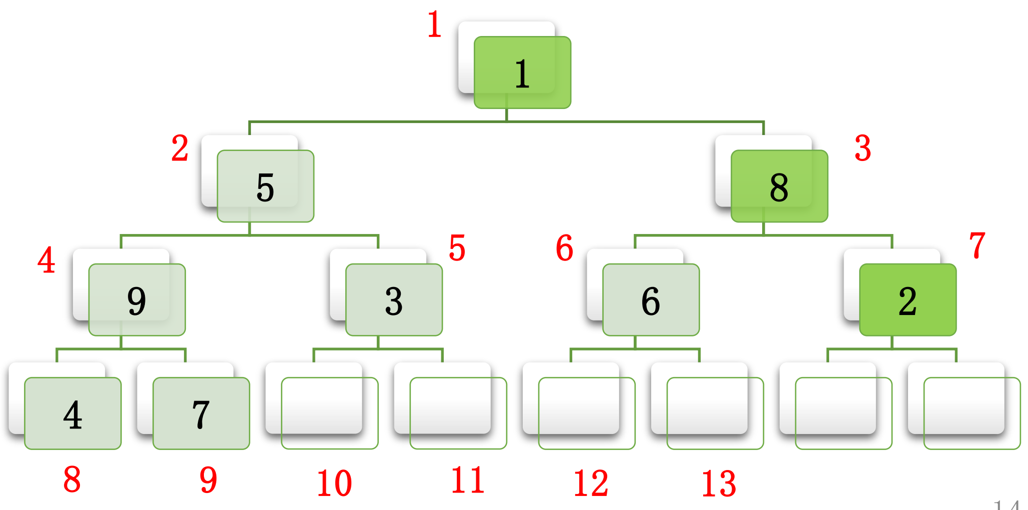


数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

1. 向上调整：

　　从当前节点开始，和它的父节点比较，若是比父节点小，就交换，然后将当前节点下标修改为其原父节点下标



数组模拟堆

1. 向上调整：

从当前节点开始，和它的父节点比较，若是比父节点小，就交换，然后将当前节点下标修改为其原父节点下标

这里的swap函数最好自己手写

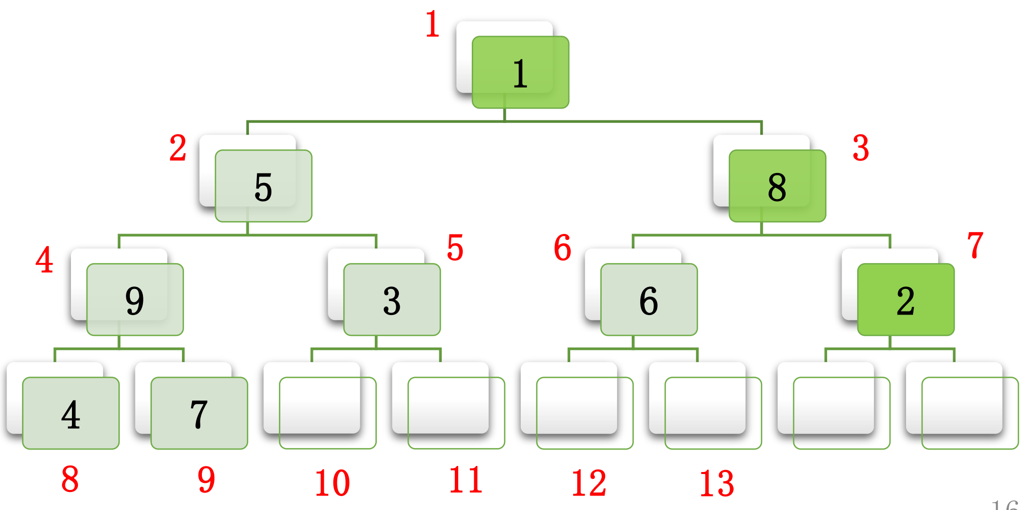
```
void up(int i)  //对当前节点 i执行向上调整
{
    while (i / 2)  //如果父节点不空
    {
        if (h[i] < h[i/2])  //如果当前节点小于父节点
        {
            swap(h[i], h[i/2]);  //交换当前节点和父节点
            i /= 2;  //更新为原父节点下标
        }
        else break;
    }
}
```

数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

2. 向下调整：

此时堆顶到位，但是3号位置不符合堆的性质，向下调整



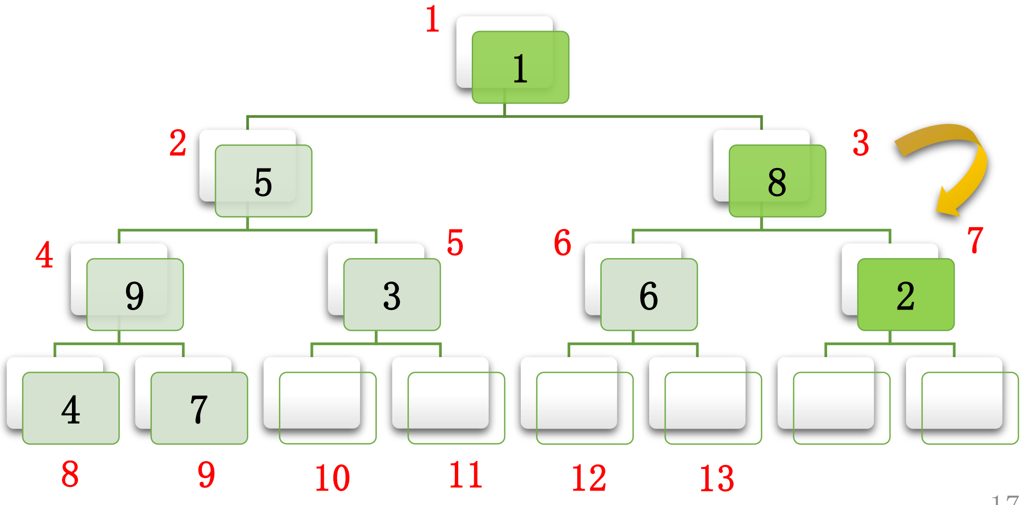
数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

2. 向下调整：

　　当前节点的左、右儿子节点做比较，哪个比较小，就和当前节点交换，然后将当前节点下标修改为该儿子节点下标

　　比如当前节点3，2<6，和右儿子节点交换，并修改当前节点下标为7



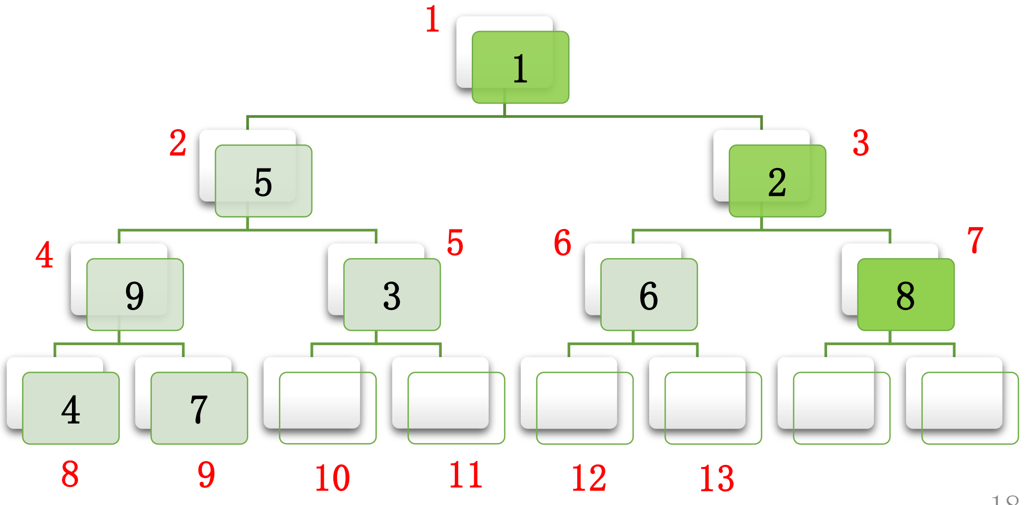
数组模拟堆

- 下面我们以小根堆为例讲解向上和向下调整操作

2. 向下调整：

　　当前节点的左、右儿子节点做比较，哪个比较小，就和当前节点交换，然后将当前节点下标修改为该儿子节点下标

　　比如当前节点3，2<6，和右儿子节点交换，并修改当前节点下标为7



数组模拟堆

2. 向下调整：

当前节点的左、右儿子节点做比较，哪个比较小，就和当前节点交换，然后将当前节点下标修改为该儿子节点下标

```
void down(int i)  //对当前节点 i执行向下调整
{
    while (2 * i <= n)  //儿子节点编号还在范围内
    {
        int temp = 2 * i;  //初始默认左儿子较小
        if (temp + 1 <= n && h[temp + 1] < h[temp]) temp ++;
        //如果右儿子更小就记录右儿子
        if (h[i] < h[temp])  //比较当前节点和较小的儿子节点
        {
            swap(h[i], h[temp]);
            //交换当前节点和较小的儿子节点
            i = temp;  //更新为儿子节点下标
        }
        else break;
    }
}
```

数组模拟堆

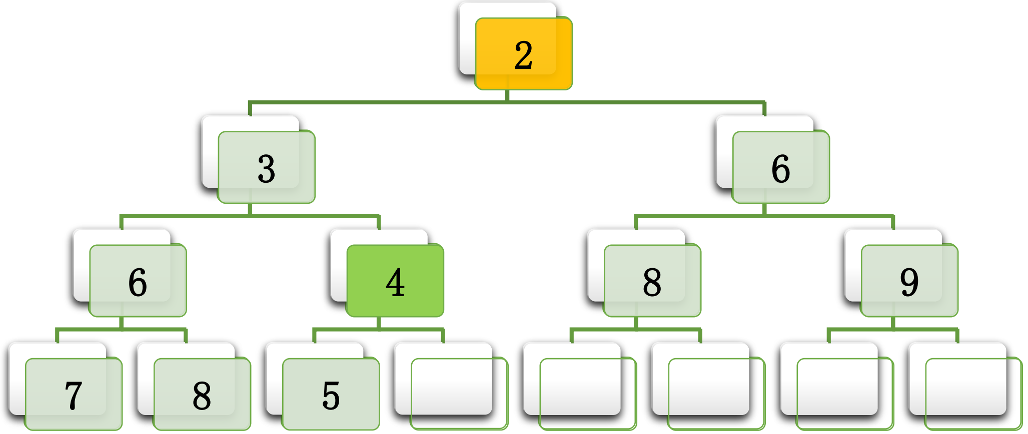
• 我们实现了向上和向下调整，就可以写插入和删除了

1. 插入操作

插入操作十分简单：

① 插入节点至堆尾

② 执行向上调整以维持堆的性质



数组模拟堆

1. 插入操作

插入操作十分简单：

① 插入节点至堆尾

② 执行向上调整以维持堆的性质

```
void push(int x)  //插入节点 x
{
    h[++n] = x;  //插入 x至堆尾
    up(n);     //对当前节点 n执行向上调整
 }
```

数组模拟堆

- 我们实现了向上和向下调整，就可以写插入和删除了

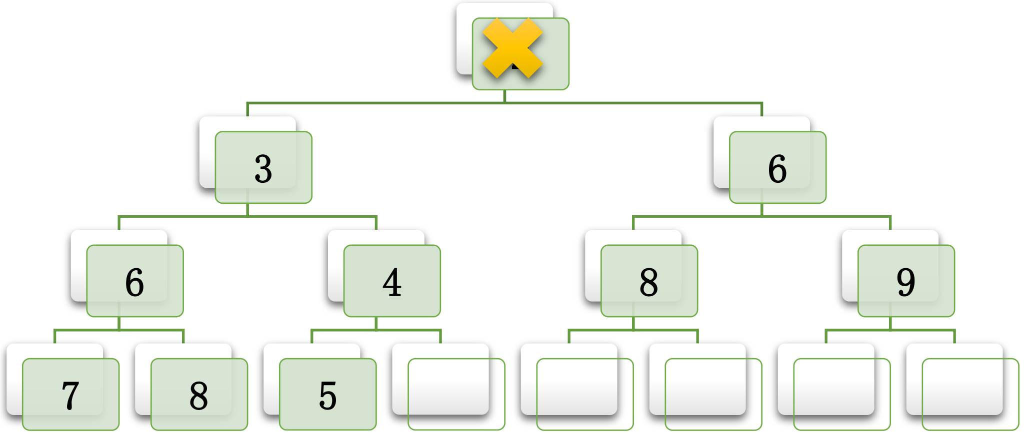
2. 删除操作

删除操作比插入操作稍复杂一点：

① 删除堆顶

② 执行向下调整以维持堆的性质

但是：删掉2后，这棵树裂开了。。。



数组模拟堆

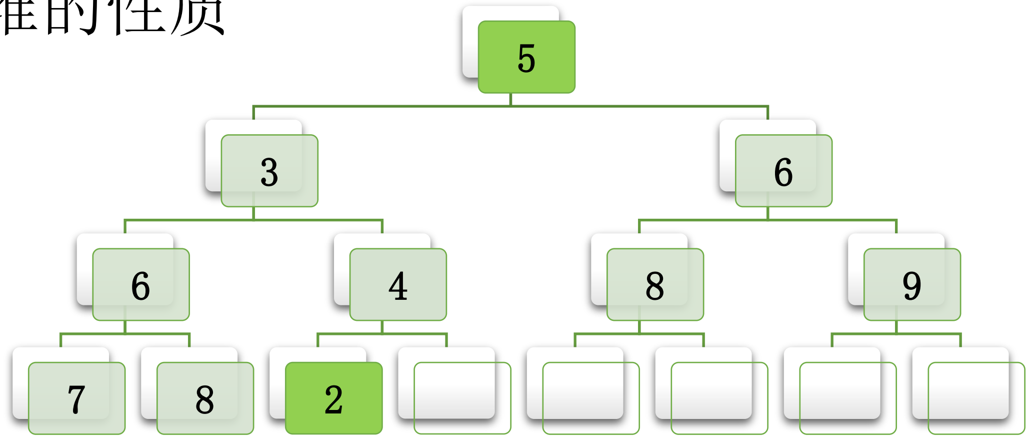
- 我们实现了向上和向下调整，就可以写插入和删除了

2. 删除操作

这就是删除操作比插入操作稍复杂一点的地方：

① 先交换堆顶和堆尾，然后把堆尾排除在堆之外

② 再从根节点开始执行向下调整以维持堆的性质



数组模拟堆

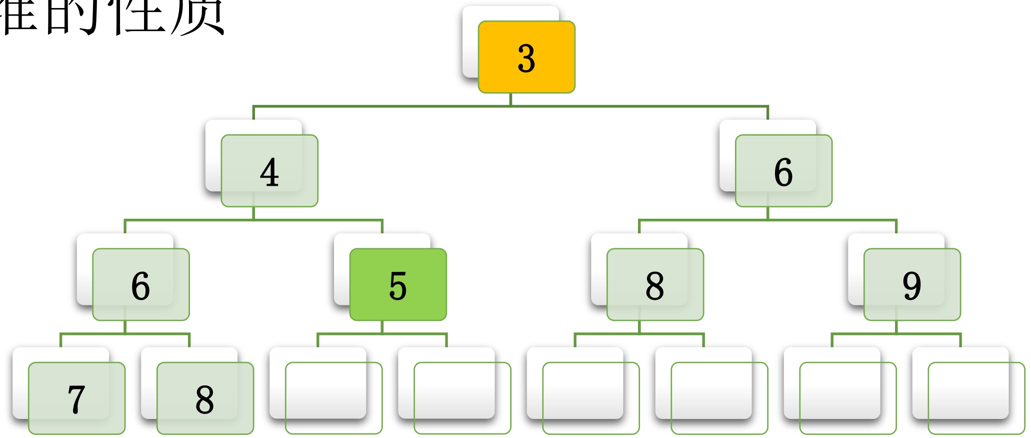
- 我们实现了向上和向下调整，就可以写插入和删除了

2. 删除操作

这就是删除操作比插入操作稍复杂一点的地方：

① 先交换堆顶和堆尾，然后把堆尾排除在堆之外

② 再从根节点开始执行向下调整以维持堆的性质



数组模拟堆

2. 删除操作

这就是删除操作比插入操作稍复杂一点的地方：

① 先交换堆顶和堆尾，然后把堆尾排除在堆之外

② 再从根节点开始执行向下调整以维持堆的性质

```
void pop()  //删除堆顶
{
    swap(h[1], h[n]);  //交换堆顶和堆尾
    n --;   //把现在的堆尾排除在堆外
    down(1);  //从根节点开始执行向下调整
 }
```

数组模拟堆

3. 建堆操作

建堆操作其实就是从空开始

不断插入新的节点

```
while (n --)
    scanf("%d", &x), push(x);
```

- Tips：建堆过程自底向上，可以保证O(n)的复杂度，而如果自上而下，复杂度将被均摊到O(nlogn)

数组模拟堆

4. 取堆顶操作

这其实已经不能算一个操作了

直接返回h[1]即可

```
return h[1];
```

堆的代码实现

2. STL自带标准堆：**heap**。具体用法如下：

1. 建堆：**make_heap()** //默认建大根堆，同priority_queue
2. 插入堆尾：**push_heap()**
3. 删除堆顶：**pop_heap()** //并未真删除，实际是交换到堆尾然后挤出堆
4. 堆排序：**sort_heap()** //堆排序后就不一定还是堆了

Tips：heap的内部是基于vector实现的，但是也可以用数组模拟

STL中的heap

```
4
4 3 9 9
4
2
2

--------------------------------
Process exited after 10.16 seconds with return value 0
```

```cpp
#include<bits/stdc++.h>
using namespace std;
bool cmp(int a, int b) {return a > b;}  //建小根堆
int main()
{
    vector<int> h;
    int n, x;
    scanf("%d", &n);
    for (int i = 0; i < n; i ++)
        scanf("%d", &x), h.push_back(x);  //压入vector
    make_heap(h.begin(), h.end(), cmp);  //建堆

    pop_heap(h.begin(), h.end(), cmp);  //删堆顶
    printf("%d\n", h[0]);

    scanf("%d", &x);
    h.push_back(x);  // x节点插入vector
    push_heap(h.begin(), h.end(), cmp);  //插入堆尾
    printf("%d\n", h[0]);
    return 0;
 }
```

堆排序 sort_heap

- 堆排序的原理，借助了堆的特性：根节点就是最大值/最小值
- 以从小到大排序为例：建立小根堆，然后每次返回根节点（堆顶）并执行删除操作，直至堆空为止

- 堆排序的复杂度O(n*logn)，也是最优秀的排序算法之一

- Tips：

堆排序、快速排序、归并排序的复杂度都是O(n*logn)

冒泡排序、选择排序、插入排序的复杂度都是O(n^2)

经典问

- 询问区间[L,R]内的第K大数

先sort，然后返回a[k]。

复杂度O(n*logn)

经典问

- 询问区间[L,R]内的第K大数

扫k次，每次把当前的最大值删除，返回最后一次的最大值。

复杂度O(n*k)，适合k<logn的时候

经典问

- 询问区间[L,R]内的第K大数

利用快排的思想，找到前面有k-1个数时的key即为答案。

并不要求前k-1个数有序

复杂度O(n*logk)

经典问

- 询问区间[L,R]内的第K大数

利用堆排的思想：建大小为k的小根堆，后续元素逐个与堆顶比较，比堆顶大则替换堆顶，小则舍弃，最后堆顶即为答案。

复杂度O((n-k)*logk)

课外加练

- luogu 3378 堆
- luogu 1168 中位数
- luogu 1878 舞蹈课
- luogu 2085 最小函数值