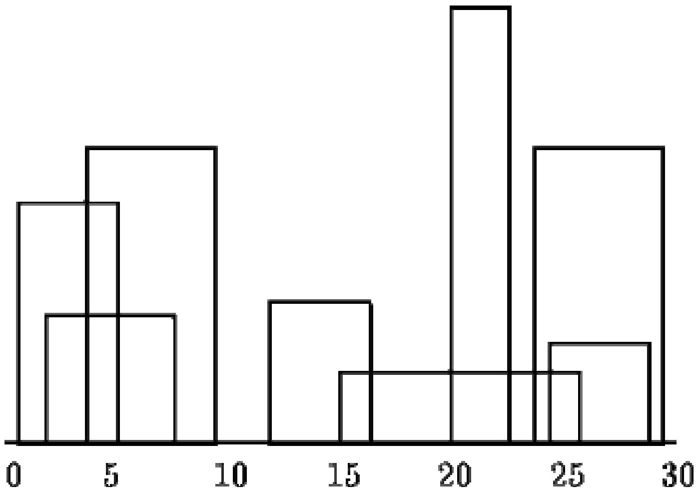
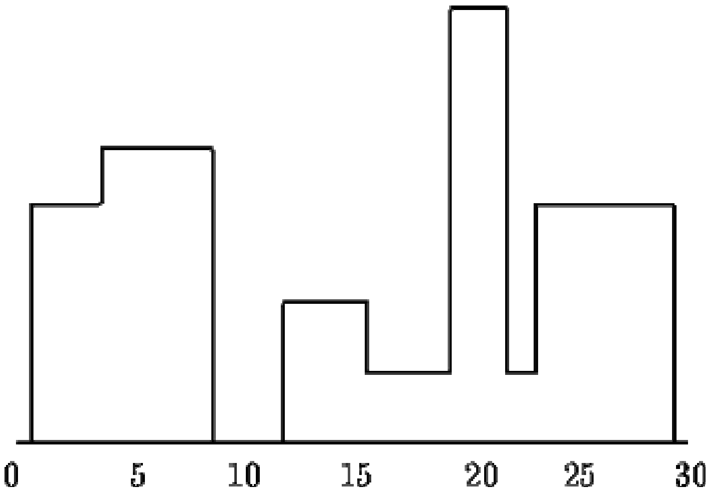
轮廓线

- 每一个建筑物用一个三元组表示(L, H, R), 表示左边界, 高度和右边界
- 轮廓线用X, Y, X, Y…这样的交替式表示
- 右图的轮廓线为: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)
- 给N个建筑，求轮廓线





二叉堆与优先队列

优先队列

- 优先队列(priority queue): 可以把元素加入到优先队列中, 也可以从队列中取出优先级最高的元素, 即以下ADT
  - **Insert(T, x):** 把x加入优先队列中
  - **DeleteMin(T, x):** 获取优先级最高的元素x, 并把它从优先队列中删除

堆的操作

- 用二叉堆(binary heap)很容易实现优先队列
- 除了实现优先队列, 堆还有其他用途, 因此操作比优先队列多
  - **Getmin(T, x):** 获得最小值
  - **Delete(T, x):** 删除任意已知结点
  - **DecreaseKey(T, x, p):** 把x的优先级降为p
  - **Build(T, x):** 把数组x建立成最小堆

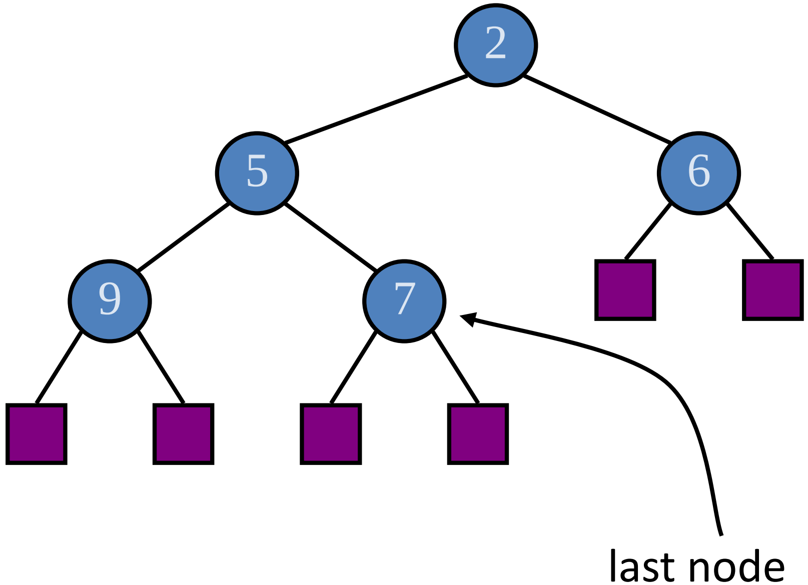
堆的定义

- 堆是一个完全二叉树
  - 所有叶子在同一层或者两个连续层
  - 最后一层的结点占据尽量左的位置
- 堆性质
  - 为空, 或者最小元素在根上
  - 两棵子树也是堆

What is a heap?

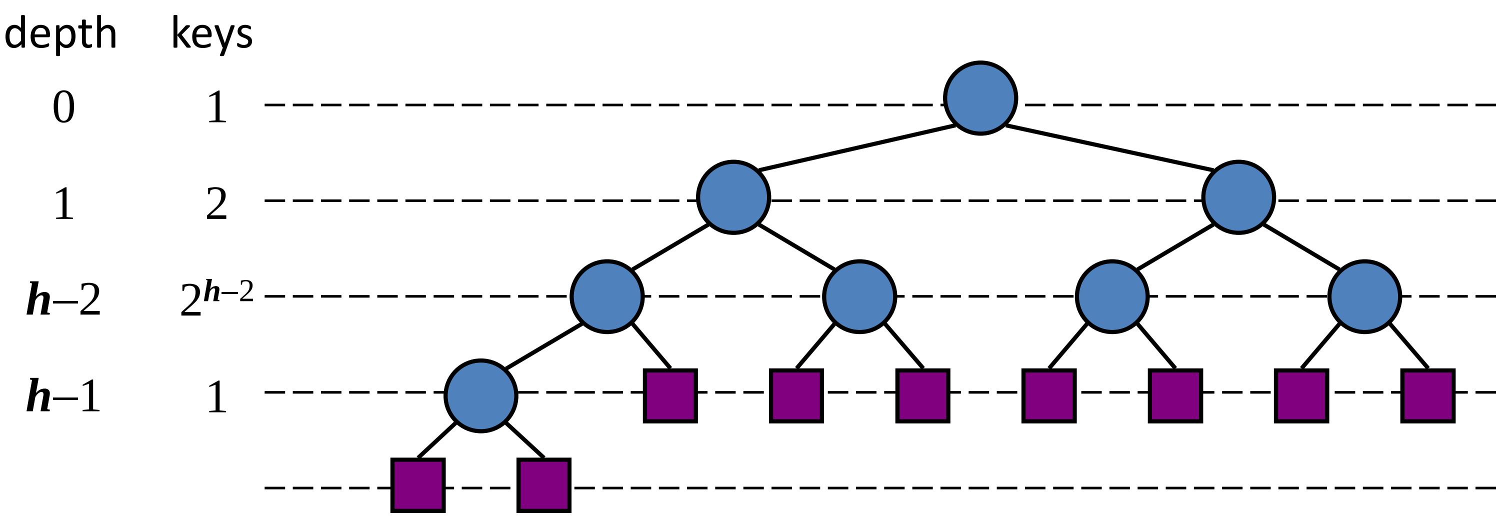
- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
    - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$, there are $2^{\boldsymbol{i}}$ nodes of depth $\boldsymbol{i}$
    - at depth $\boldsymbol{h} - 1$, the internal nodes are to the left of the leaf nodes

- The last node of a heap is the rightmost internal node of depth $\boldsymbol{h} - 1$



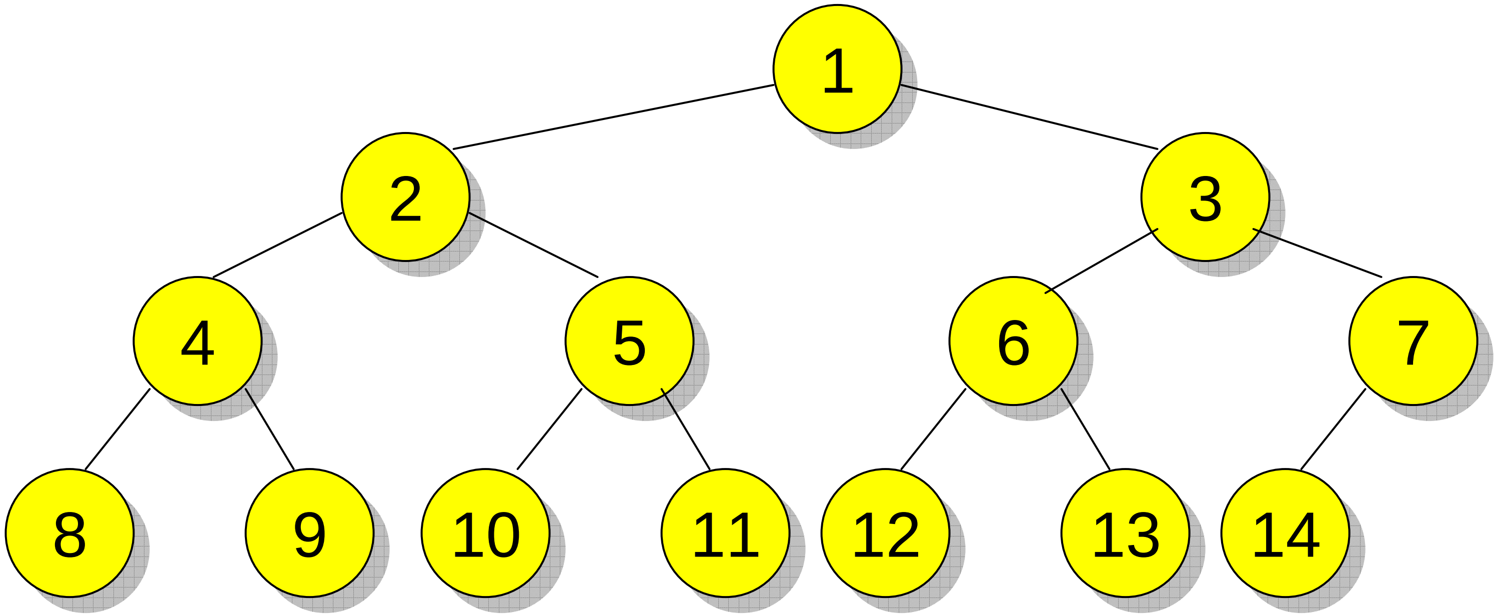
Height of a Heap

- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



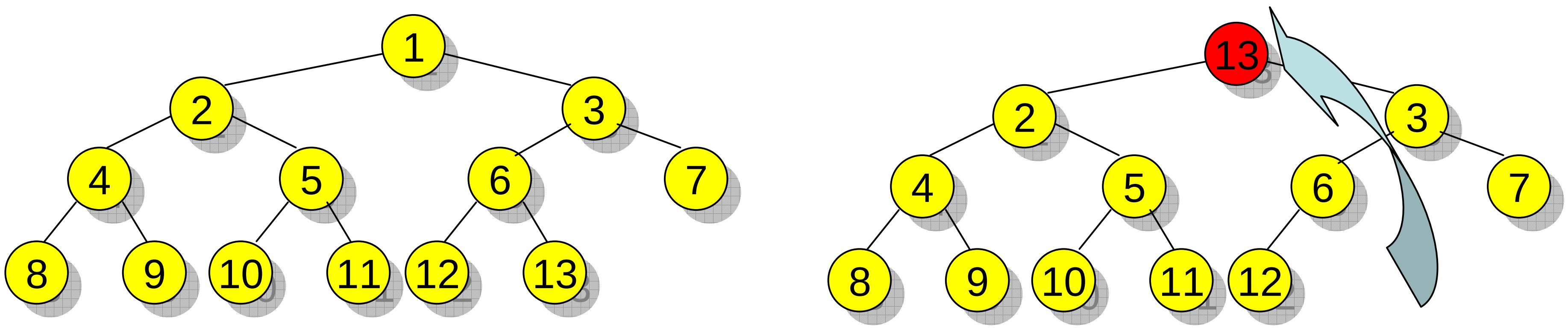
存储方式

- 最小堆的元素保存在heap[1..hs]内
  – 根在heap[1]
  – K的左儿子是2k, K的右儿子是2k+1,
  – K的父亲是[k/2]

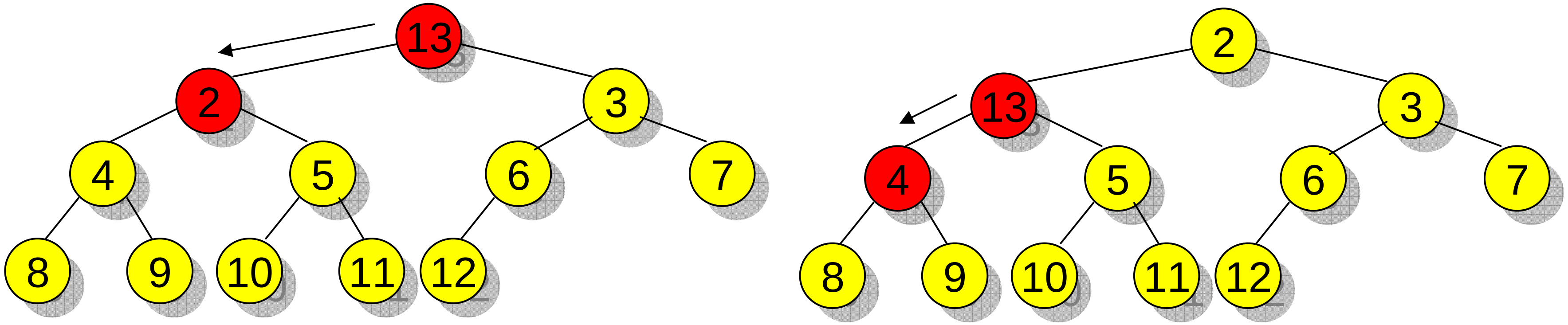


删除最小值元素

- 三步法
  - 直接删除根
  - 用最后一个元素代替根上元素
  - <u>向下调整</u>



- 首先选取当前结点p的较小儿子. 如果比p大, 调整停止, 否则交换p和儿子, 继续调整




```
void sink(int p){
    int q=p<<1, a = heap[p];
    while(q<=hs){
        if(q<hs&&heap[q+1]<heap[q])q++;
        if(heap[q]>=a) break;
        heap[p]=heap[q]; p=q; q=p<<1;
    }
    heap[p] = a;
}
```

插入元素和向上调整

- 插入元素是先添加到末尾, 再向上调整
- 向上调整: 比较当前结点p和父亲, 如果父亲比p小，停止; 否则交换父亲和p, 继续调整

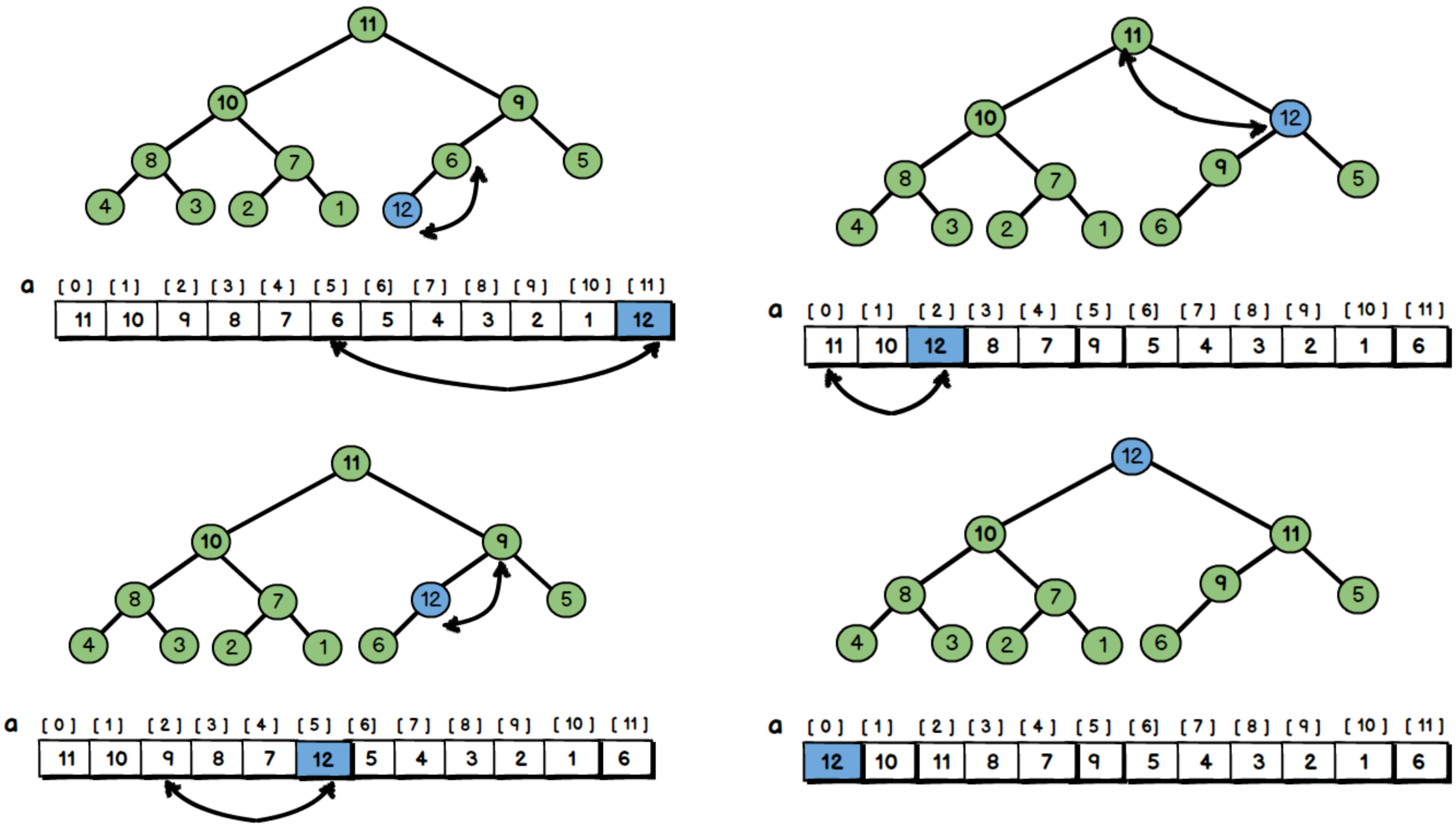
```
void swim(int p){
    int q = p>>1, a = heap[p];
    while(q && a<heap[q]){ heap[p]=heap[q]; p=q; q=p>>1; }
    heap[p] = a;
}
```

堆的建立

- 从下往上逐层向下调整. 所有的叶子无需调整, 因此从hs/2开始. 可用数学归纳法证明循环变量为i时, 第i+1, i+2, …n均为最小堆的根

```
void insert(int a)
{ heap[++hs]=a; swim(hs); }
int getmin()
{ int r=heap[1]; heap[1]=heap[hs--];
  sink(1); return r; }
int decreaseKey(int p, int a)
{ heap[p]=a; swim(p); }
void build()
{ for(int i=hs/2;i>0;i--) sink(i); }
```

二叉堆的插入

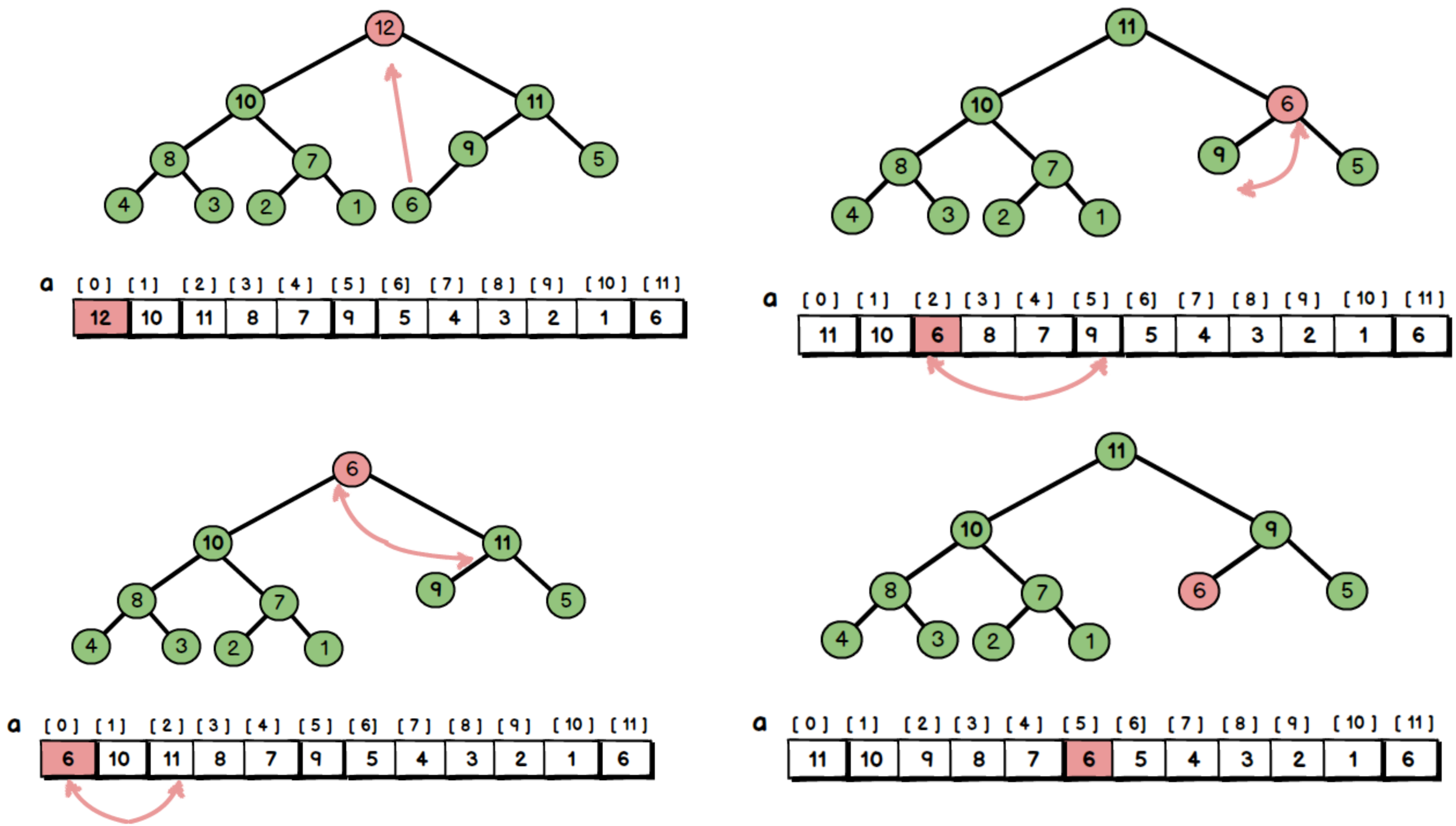


a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
11 10 9 8 7 6 5 4 3 2 1 12
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
11 10 12 8 7 9 5 4 3 2 1 6
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
11 10 9 8 7 12 5 4 3 2 1 6
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
12 10 11 8 7 9 5 4 3 2 1 6

二叉堆的插入

```
void Insert(T value) //value表示待插值，size表示已插的数量，capacity表示数组大小
    {
        if (size < capacity)  //当前已插入的数量小于数组大小
        {
            heap[size++] = value;   //将插入值，放到数组的有效元素最后一个位置
            int index = size;
            while (index>1)
            {
                if (heap[index] > heap[index / 2]) //比较双亲结点的值与子节点的值
                    swap(heap[index], heap[index / 2]); //如果双亲结点的值大于子节点，则交换
                index /= 2;     //向上不断调整
            }
        }
        else            //数组已满，退出
            return;

```

二叉堆的删除



a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
12 10 11 8 7 9 5 4 3 2 1 6
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
11 10 6 8 7 9 5 4 3 2 1 6
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
6 10 11 8 7 9 5 4 3 2 1 6
a [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
11 10 9 8 7 6 5 4 3 2 1 6

二叉堆的删除

```
void DeleteMax()
    {
        if (size == 0)
            return;
        swap[heap[1], heap[size--]];  //用最后一个结点将第一个结点替换，
                                                                //替换结束后最后一个结点为原本第一个结点
                                                                //size--表示删除最后一个结点，
        int index = 1;                  //也就是将第一个节点间接删除
        while (index <= size)
        {
            int p = 2 * index;
            if (p < size && heap[p] < heap[p + 1]) //找孩子节点中最大的一个
                p++;
            swap(heap[p], heap[index]);
            index = p;    //继续向下调整
        }
    }
```

二叉堆的建立

```
void BuildHeap(int res[], int size)  //用数组创建堆
{
    for (int i = size / 2; i >= 1; --i)
    {
        int temp = res[i];
        int index = 2 * i;
        while (index <= size)
        {
            if (index < size &&res[index] < res[index + 1])
                index++;
            if (res[index] > temp)
            {
                res[index / 2] = res[index];
                index *= 2;
            }
            else
                break;
        }
        res[index / 2] = temp;
    }
}
```

时间复杂度分析

- 向上调整/向下调整
  - 每层是常数级别, 共logn层, 因此O(logn)
- 插入/删除
  - 只调用一次向上或向下调整, 因此都是O(logn)
- 建堆
  - 高度为h的结点有$n/2^{h+1}$个,总时间为

时间复杂度分析

- 向上调整/向下调整
  - 每层是常数级别, 共logn层, 因此O(logn)
- 插入/删除
  - 只调用一次向上或向下调整, 因此都是O(logn)
- 建堆
  - 高度为h的结点有$n/2^{h+1}$个,总时间为

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

例1. k路归并问题

- 把k个有序表合并成一个有序表.
- 元素共有n个.

分析

- 每个表的元素都是从左到右移入新表
- 把每个表的当前元素放入二叉堆中, 每次删除最小值并放入新表中, 然后加入此序列的下一个元素
- 每次操作需要logk时间, 因此总共需要nlogk的时间

例2. 序列和的前n小元素

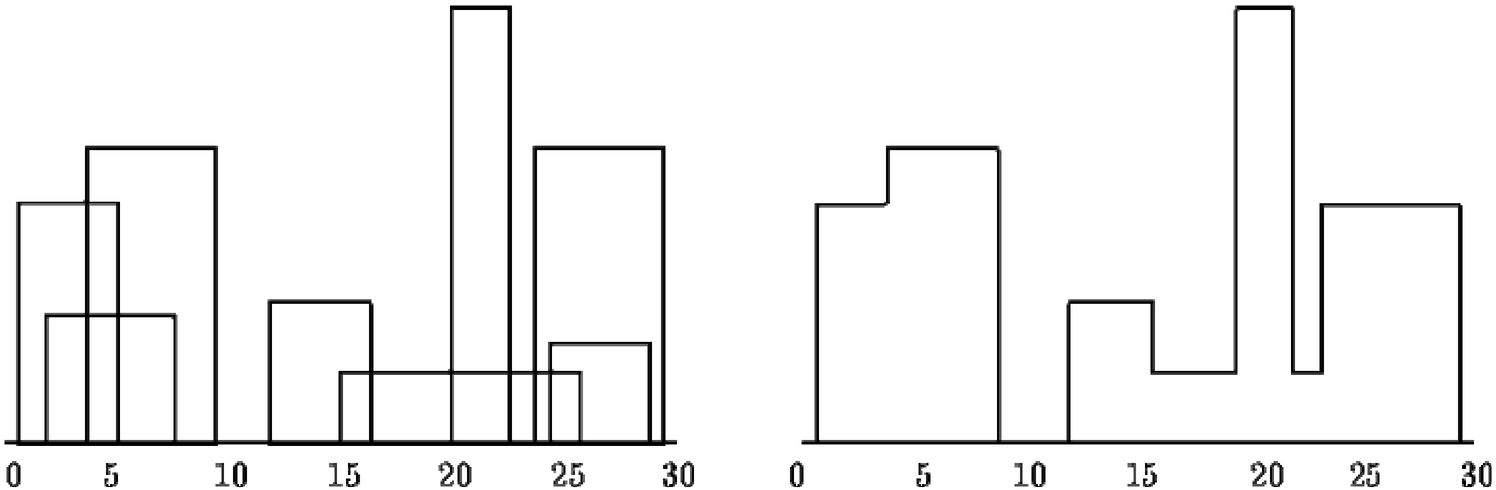
- 给出两个长度为n的有序表A和B, 在A和B中各任取一个, 可以得到n^2个和. 求这些和最小的n个

分析

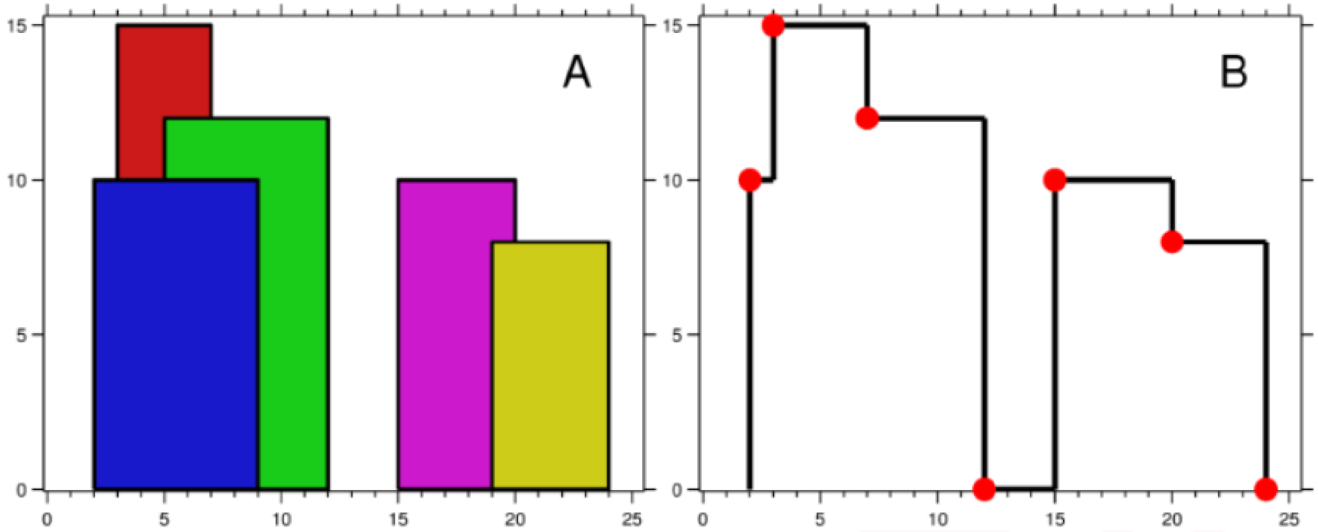
- 可以把这些和看成n个有序表:
  - A[1]+B[1] <= A[1]+B[2] <= A[1]+B[3] <=…
  - A[2]+B[1] <= A[2]+B[2] <= A[2]+B[3] <=…
  - …
  - A[n]+B[1] <= A[n]+B[2] <= A[n]+B[3] <=…
- 类似刚才的算法, 每次O(logn), 共取n次最小元素,共O(nlogn)

例3. 轮廓线

- 每一个建筑物用一个三元组表示(L, H, R), 表示左边界, 高度和右边界
- 轮廓线用X, Y, X, Y…这样的交替式表示
- 右图的轮廓线为: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)
- 给N个建筑，求轮廓线



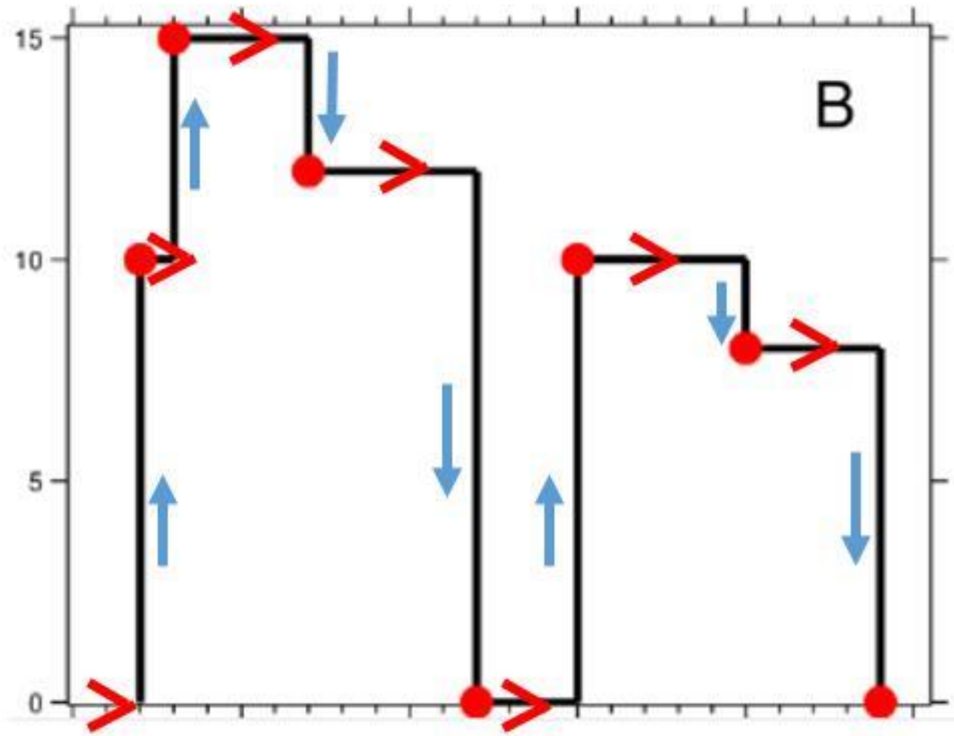
例3. 轮廓线



建筑物：[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]

天际线点：[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24 0]

例3. 轮廓线



理解关键点：关键点指导画出轮廓。我们只需要从原点向右出发，沿着水平方向一直画线。如果在正上方或者正下方遇到关键点，就拐向关键点。到达关键点后继续向右水平画线，重复上边的过程即可。

分析

- 算法一: 用数组记录每一个元线段的高度
  - 离散化, 有n个元线段
  - 每次插入可能影响n个元线段, O(n), 共$O(n^2)$
  - 从左到右扫描元线段高度, 得轮廓线
- 算法二：每个建筑的左右边界为事件点
  - 把事件点排序, 从左到右扫描
  - 维护建筑物集合, 事件点为线段的插入删除
  - 需要求最高建筑物, 用堆, 共O(nlogn)

例3. 轮廓线

只考虑每个建筑物的左上角和右上角坐标，将所有点按 x 坐标排序，然后开始遍历，并且用一个优先队列来存储遍历坐标的高度，也就是 y 轴坐标。

- 遇到左上角坐标，将其 y 坐标加入到优先队列中。
- 遇到右上角坐标，将其 y 坐标从优先队列中删除，也就是删除了其对应的左上角坐标的 y 值。
- 最后判断优先队列中的最高高度相对于之前是否更新，如果更新了的话，就将当前的 x 以及更新后的最高高度作为一个坐标加入到最终结果中。

例4. 丑数

- 素因子都在集合{2, 3, 5, 7}的数称为ugly number
- 求第n大的丑数

分析

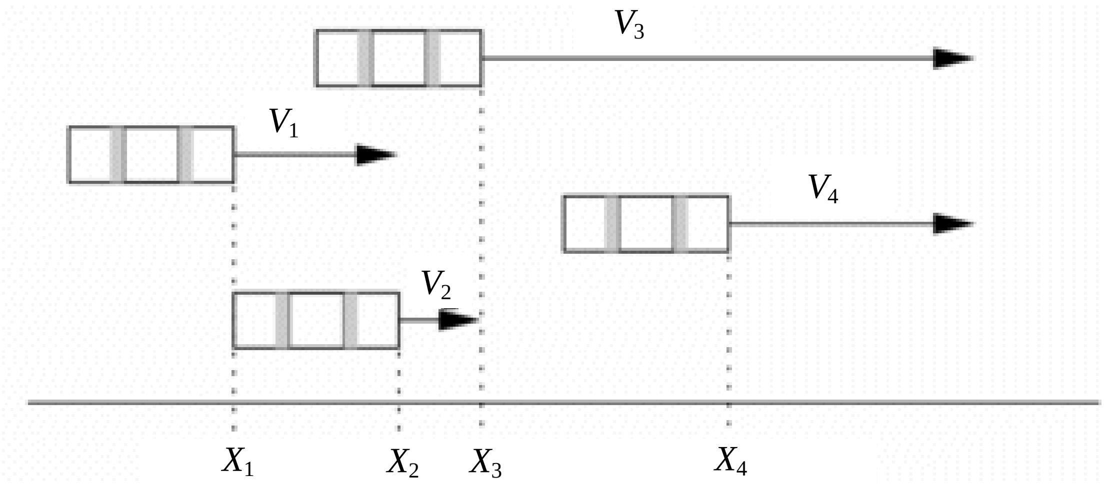
- 初始：把1放入优先队列中
- 每次从优先队列中取出一个元素k，把2k, 3k, 5k, 7k放入优先队列中
- 从2开始，取出的第n个元素就是第n大丑数
- 每取出一个数，插入4个数，因此任何堆里的元素是O(n)的，时间复杂度为O(nlogn)
- 思考: 如果集合元素个数m与n同阶, 时间复杂度将变为怎样? 如何优化?

动态规划

```java
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        //昨晚看了一下，应该是动规
        //所有质因数都出现在数组里，应该是用数组中的数动规
        //刚开始数组中总共l+1个数，然后就是里面的数互相乘，得到下一个数
        //其实和丑数2差不多，那应该怎么做呢
        //dp[i][j]

        int l = primes.length;

        //想想动规，每个数有l个数可以乘
        //然后标记哪个数被乘过了，如果都被乘过了，那么就下一个数
        //但是怎么取下一个数呢
        //对于第一个例子，比如第一个数是1，那么它有l个数可以乘
        //从小到大，得到第二个数是2，dp[1] = 1; dp[2] = 2;
        //然后比较1*7和2 *2 ，得到第三个数是4
        //然后比较1*7和4*2
        //有了，就是对于primes中的每个数，都有一个当前的最小数，然后根据这个最小数，往后计算

        //对应primes中每个位置的最小数
        int[] base = new int[l];
        //刚开始所有的最小数都为1
        Arrays.fill(base, 1);

        //记录当前丑数
        int[] dp = new int[n + 1];
        dp[1] = 1;
        for(int i = 2; i <= n; i++){
            //计算当前的超级丑数
            int res = Integer.MAX_VALUE;
            for(int j = 0; j < l; j++){
                if(dp[base[j]] * primes[j] < res){
                    res = dp[base[j]] * primes[j];
                }
            }
            //如果当前最小数是t位置取的，那么就把t位置的下标往后移一位
            //但是因为会有重复的，所以需要判断
            for(int j = 0; j < l; j++){
                if(dp[base[j]] * primes[j] == res){
                    base[j]++;
                }
            }
            dp[i] = res;
            //System.out.println(res);
        }
        return dp[n];
    }
}
```

例5. 赛车

- 有n辆赛车从各不相同的地方以各种的速度(速度 $0<v_i<100$)开始往右行驶，不断有超车现象发生。



- 给出n辆赛车的描述（位置x_i，速度v_i），赛车已按照位置排序（$x_1<x_2<\ldots<x_n$）
- 输出超车总数以及按时间顺序的前m个超车事件

分析

- 事件个数$O(n^2)$, 因此只能一个一个求
- 给定两辆车，超越时刻预先可算出
- 第一次超车可能在哪些辆车之间？
  - 维护所有车的前方相邻车和追上时刻
    - 局部：此时刻不一定是该车下个超车时刻！
    - 全局：所有时刻的最小值就是下次真实超车时刻
- 维护：超车以后有什么变化？
  - 相对顺序变化…改变三个车的前方相邻车
  - 重新算追上时刻，调整三个权
  - 简单的处理方法：删除三个再插入三个

例7. 黑匣子

- 我们使用黑匣子的一个简单模型。它能存放一个整数序列和一个特别的变量*i*。在初始时刻，黑匣子为空且*i*等于0。这个黑匣子执行一序列的命令。有两类命令：
- ADD(*x*)：把元素*x*放入黑匣子；
- GET：*i*增1的同时，输出黑匣子内所有整数中第*i*小的数。牢记第*i*小的数是当黑匣子中的元素以非降序排序后位于第*i*位的元素

例7. 黑匣子

编号	命令	i	黑匣子内容	输出
1	ADD(3)	0	3	
2	GET	1	**3**	3
3	ADD(1)	1	1, 3	
4	GET	2	1, **3**	3
5	ADD(-4)	2	-4, 1, 3	
6	ADD(2)	2	-4, 1, 2, 3	
7	ADD(8)	2	-4, 1, 2, 3, 8	
8	ADD(-1000)	2	-1000, -4, 1, 2, 3, 8	
9	GET	3	-1000, -4, **1**, 2, 3, 8	1
10	GET	4	-1000, -4, 1, **2**, 3, 8	2
11	ADD(2)	4	-1000, -4, 1, 2, 2, 3, 8	

分析

- 降序堆$H_{\geq}$和升序堆$H_{\leq}$如图放置
- $H_{\geq}$根节点的值$H_{\geq}[1]$在堆$H_{\geq}$中最大，$H_{\leq}$根节点的值$H_{\leq}[1]$在堆$H_{\leq}$中最小,并满足
  - $H_{\geq}[1] \leq H_{\leq}[1]$
  - $size[H_{\geq}]=i-1$
- ADD(*x*): 比较*x*与$H_{\geq}[1]$，若$x \geq H_{\geq}[1]$，则将*x*插入$H_{\leq}$，否则从$H_{\geq}$中取出$H_{\geq}[1]$插入$H_{\leq}$，再将*x*插入$H_{\geq}$
- GET:　　$H_{\leq}[1]$就是待获取的对象。输出$H_{\leq}[1]$，同时从$H_{\leq}$中取出$H_{\leq}[1]$插入$H_{\geq}$，以维护条件(2)