算法基础（七）：Trie树，并查集，堆

Trie树

基本思想：

用来高效存储和查找字符串集合的数据结构

在用到trie树的时候字符的类型不是很多，比如全是小写或全是大写字符

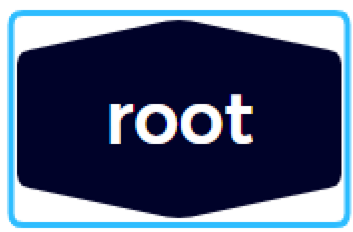
建树过程：

以一个字符串集合`abcd adef bced gfac abc`为例

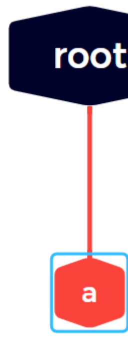
总体思想就是从根节点开始遍历，若当前结点没有该字符的儿子节点，则创建，然后在从当前字符的儿子节点开始重复上述过程，一直递归下去

以`abcd`为例：

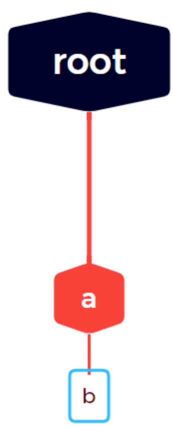
从根节点开始：



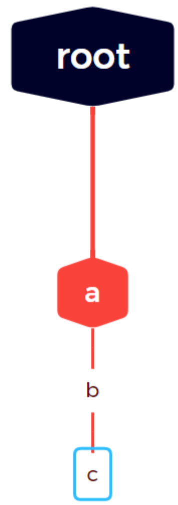
没有字符`a`，创建子结点，并选中`a`结点：



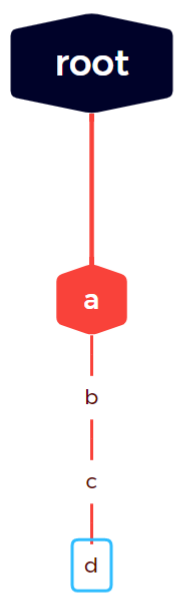
然后在`a`结点开始遍历其子结点，发现没有字符`b`，创建`b`结点，并选中`b`：



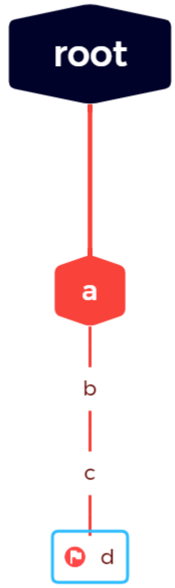
然后遍历`b`的所有子结点，发现没有`c`，创建`c`结点，并选中`c`：



然后遍历`c`结点的所有子结点，发现没有`d`，创建`d`结点，并选中`d`：

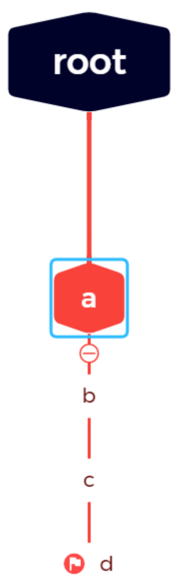


发现`d`是一个字符串的末尾，做上一个标记：

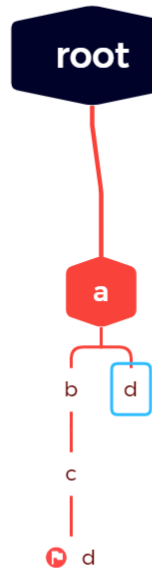


再以`adef`串为例：

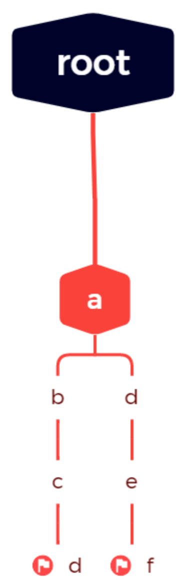
从根节点开始，遍历所有的子结点，发现有`a`结点，则进入`a`结点：



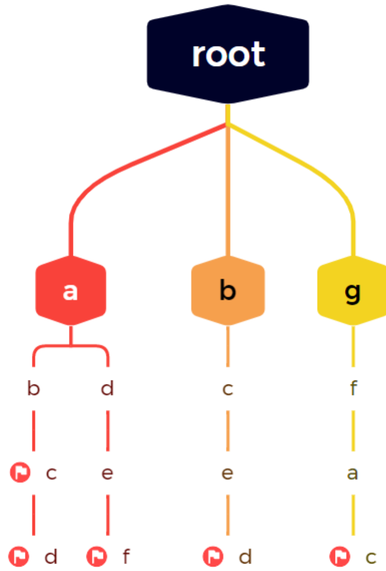
遍历`a`结点的所有子结点，发现没有`d`结点，于是创建`d`结点，然后选中进入`d`结点：



重复上述过程，`adef`建串之后得到：



过程类似，串集合`abcd adef bced gfac abc`建树后得到：



查找过程：

与建树过程很类似，不用再赘述了

代码实现

维护一个字符串集合，支持两种操作：

1. `I x` 向集合中插入一个字符串 x；
2. `Q x` 询问一个字符串在集合中出现了多少次。

共有 N 个操作，输入的字符串总长度不超过 10^5，字符串仅包含小写英文字母。

输入格式

第一行包含整数 N，表示操作数。

接下来 N 行，每行包含一个操作指令，指令为 `I x` 或 `Q x` 中的一种。

输出格式

对于每个询问指令 `Q x`，都要输出一个整数作为结果，表示 x 在集合中出现的次数。

每个结果占一行。

数据范围

$1 \le N \le 2 * 10^4$

输入样例：

```
5
I abc
Q abc
Q ab
I ab
Q ab
```

输出样例：

```
1
0
1
```

```
#include<iostream>

using namespace std;
const int N = 100010;

//son[N][26]表示N的结点最多有26个子结点（因为题目中说的是全是小写字母）
//cnt[N]表示，以cnt[i]结点做结尾的字符串有多少个
//idx用来对所有的结点进行编号，根结点的编号是0，每分配一个结点就idx++
//这些编号作为son[][]数组的一维下标来将这个结点作为父结点来遍历
int son[N][26], cnt[N], idx;// 下标是0的点，既是根节点又是空结点

char str[N];


//将一个字符插入到树中
void insert(char str[]){
    int p = 0;//p = 0表示从根节点开始
    for(int i = 0; str[i]; i ++){
        int u = str[i] - 'a';//求出这个字符对应的26个字母的编号
        if(!son[p][u]) son[p][u] = ++ idx;//给分配的结点进行编号，数组的值存放编号
        p = son[p][u];//分配一个结点后就从选中这个结点，下次循环再从这个结点开始遍历
```

```
        //这里采用一个很巧妙的记录方式，数组的一维下标是父节点，二维下标是子结点，并且子结点的值记录以这个子结点作为父节点的一维下标
        //比如根节点是son[0][...]，其中一个子结点是son[0][5]其值为4（idx作为编号分配）
        //那么当我们以这个子结点为父结点开始遍历的时候，就将p设置为4然后遍历son[4][...]的所有子结点
    }
    cnt[p]++;//以这个结点为结尾的字符串的个数++
}

//查询的过程
//思想与插入的过程很类似
//p作为当前选中的结点指针，u用来遍历子结点
int query(char str[]){
    int p = 0;
    for(int i = 0; str[i]; i++){
        int u = str[i] - 'a';
        if(!son[p][u]) return 0;

        p = son[p][u];
    }
    return cnt[p];
}

int main(){
    int n;
    scanf("%d", &n);

    while(n--){
        char op[2];
        scanf("%s%s", op, str);
        if(op[0] == 'I') insert(str);
        else printf("%d\n", query(str));

    }

    return 0;
}
```

并查集

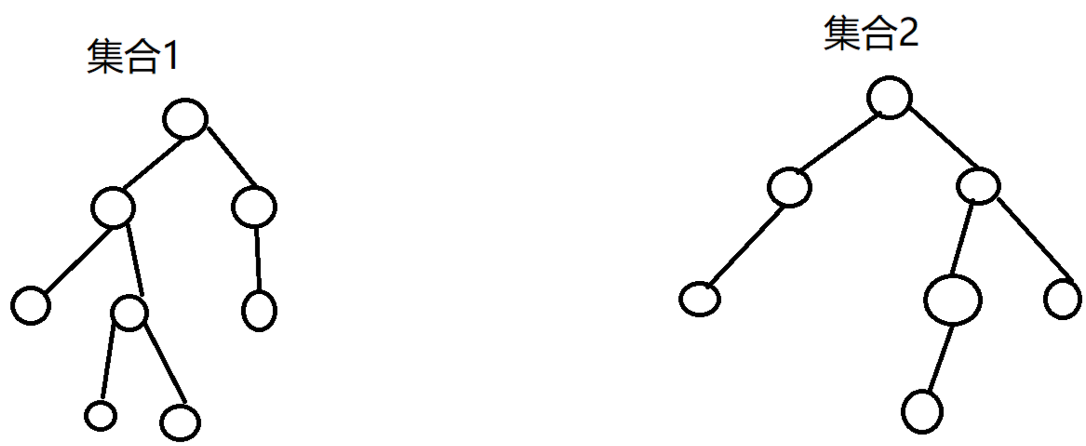
用来快速处理：

1. 将两个集合合并
2. 询问两个元素是否在一个集合当中

并查集可以近乎`O(1)`的时间复杂度之内支持上述两个操作，注意到并查集进行路径压缩之后每个结点都父结点都会指向根结点，所以递归只用调用一次就可以得出结果，所以最后的时间复杂度是近似`O(1)`的：

基本思想

每个集合用一个树来表示，树根的编号就是这个集合的编号，每个结点存储他的父结点，比如：`p[x] = k` 表示`x`节点的父结点是`k`



1. **如何判断这个结点是树根？**

`p[x] = x`，表示`x`结点的父结点是其本身，也就是说只有根结点的父结点是本身

2. **如何求`x`属于的集合编号？**

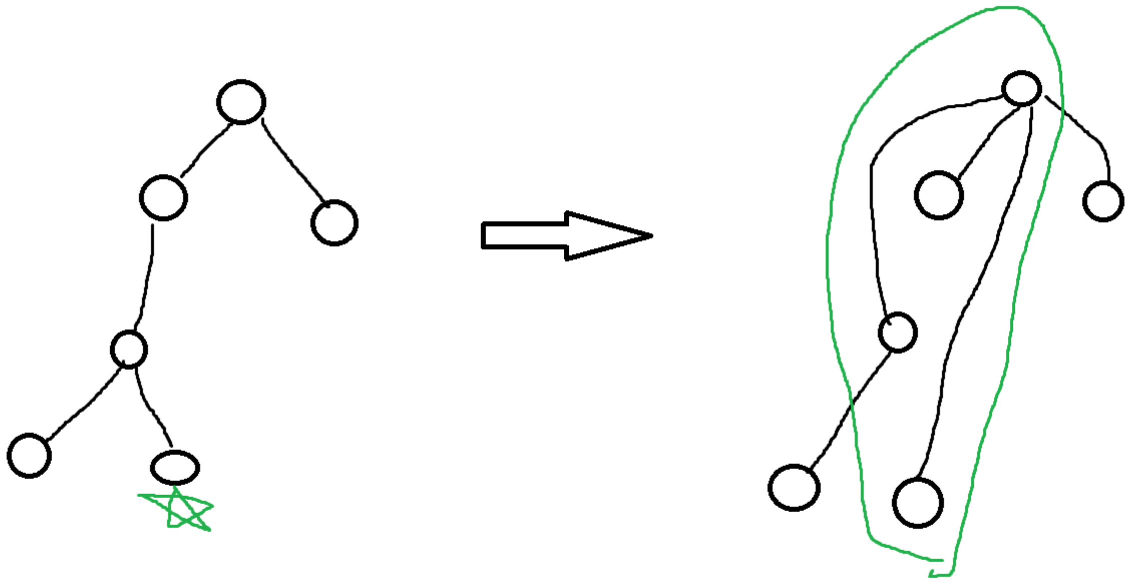
`while(p[x] != x) x = p[x]`，一直向上回溯到树根

3. **如何合并两个集合？**

`p[x]`是`x`的集合编号，`p[y]`是`y`的集合编号，那么直接将一个树当作另外一个树的儿子即可实现两个数的合并，比如将`x`树当作`y`树的儿子，即`p[x] = y`，即`x`集合根结点的父结点的编号设置为`y`集合的根结点

4. **对第二个操作的优化（路径压缩）：**

我们可以看到，第二个操作仍然是与树的高度有关的，并不是常数级别的复杂度，那么我们可以做这样一个优化：在一次搜索的过程中，将搜索路径上的每个点的父结点都更改为根结点，这样在下次寻找的过程中就不用再往前回溯了，于是就将后面的搜索复杂度降低到几乎常数级别



代码实现

一共有 n 个数，编号是 $1 \sim n$，最开始每个数各自在一个集合中。

现在要进行 m 个操作，操作共有两种：

1. `M a b`，将编号为 a 和 b 的两个数所在的集合合并，如果两个数已经在同一个集合中，则忽略这个操作；
2. `Q a b`，询问编号为 a 和 b 的两个数是否在同一个集合中；

输入格式

第一行输入整数 n 和 m。

接下来 m 行，每行包含一个操作指令，指令为 `M a b` 或 `Q a b` 中的一种。

输出格式

对于每个询问指令 `Q a b`，都要输出一个结果，如果 a 和 b 在同一集合内，则输出 `Yes`，否则输出 `No`。

每个结果占一行。

数据范围

$1 \leq n, m \leq 10^5$

输入样例：

```
4 5
M 1 2
M 3 4
Q 1 2
Q 1 3
Q 3 4
```

输出样例：

```
Yes
No
Yes
```

```
#include<iostream>
using namespace std;
const int N = 100010;

int n, m;
int p[N];//父亲数组

//返回x的所在集合编号，加上路径压缩优化
int find(int x){
    //这个递归非常巧妙，先一路回溯到根结点
    //然后再把根结点的值不断返回回赋予回溯路上的p[x],一直到最开始查询的p[x]
    if(p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int main(){
    cin >> n >> m;

    for(int i = 1; i <= n; i++){
```

```
        p[i] = i;//读入n个数，每个数都属于不同的集合，每个数都是自己集合的根结点，父结点就是自己
    }

    while(m--){
        char op[2];
        int a, b;
        scanf("%s%d%d", op, &a, &b);//这里用一个字符串来读入字符，是因为如果用单个字符的话c语言会读入空格，而字符串可以过滤掉空格和换行
        if(op[0] == 'M'){
            p[find(a)] = find(b);
        }else{
            if(find(a) == find(b)) puts("Yes");
            else puts("No");
        }
    }

    return 0;
}


```

并查集的一个变种

给定一个包含 n 个点（编号为 $1 \sim n$）的无向图，初始时图中没有边。

现在要进行 m 个操作，操作共有三种：

1. `C a b`，在点 a 和点 b 之间连一条边，a 和 b 可能相等；
2. `Q1 a b`，询问点 a 和点 b 是否在同一个连通块中，a 和 b 可能相等；
3. `Q2 a`，询问点 a 所在连通块中点的数量；

输入格式

第一行输入整数 n 和 m。

接下来 m 行，每行包含一个操作指令，指令为 `C a b`，`Q1 a b` 或 `Q2 a` 中的一种。

输出格式

对于每个询问指令 `Q1 a b`，如果 a 和 b 在同一个连通块中，则输出 `Yes`，否则输出 `No`。

对于每个询问指令 `Q2 a`，输出一个整数表示点 a 所在连通块中点的数量

每个结果占一行。

数据范围

$1 \le n, m \le 10^5$

输入样例：

```
5 5
C 1 2
Q1 1 2
Q2 1
C 2 5
Q2 5
```

输出样例：

```
Yes
2
3
```

用一个集合来维护一个连通块，连一条边就是把两个集合合并，是否在一个连通块中就是查询是否在同一个集合中

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 100010;
int n, m;
int p[N], Size[N];//只有根结点的size[i]是有意义的


int find(int x){
    if(p[x] != x) p[x] = find(p[x]);
    return p[x];
}


```

```
int main(){
    cin >> n >> m;

    for(int i = 1; i <= n; i ++){
        p[i] = i;
        Size[i] = 1;
    }

    while(m --){
        char op[2];
        int a, b;
        scanf("%s", op);

        if(op[0] == 'C'){

            scanf("%d%d", &a, &b);
            if(find(a) == find(b)) continue;//如果在同一个集合，直接向下进行即可，不用合并
            //计算集合的数量，只需要在集合合并的时候将两个集合的结点的数目相加即可
            Size[find(b)] += Size[find(a)];
            //必须先计算再合并，否则数目会翻倍
            p[find(a)] = find(b);

        }else if(op[1] == '1'){

            scanf("%d%d", &a, &b);
            if(find(a) == find(b)) printf("Yes\n");
            else printf("No\n");

        }else{
            scanf("%d", &a);
            printf("%d\n",Size[find(a)]);
        }
    }
    return 0;
}
```

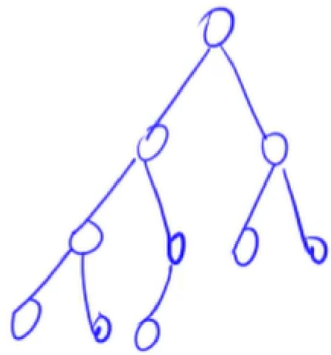
堆（如何手写一个堆

堆的目的：维护一个数据集合，实现以下操作(以小根堆为例)

1. 插入一个数
2. 求集合中的最小值
3. 删除最小值
4. 删除任意一个元素
5. 修改任意一个元素

基本结构与思想

堆是一颗完全二叉树，这颗树的结构是除了最后一层结点以外，上面的所有结点都是满的（指都具有两个子结点），并且最后一层的结点是从左到右排列的



180709

堆需要满足一些性质，比如

小根堆：

1. 每一个结点的值都小于等于左右儿子结点的值
2. 根结点是整棵树的最小值

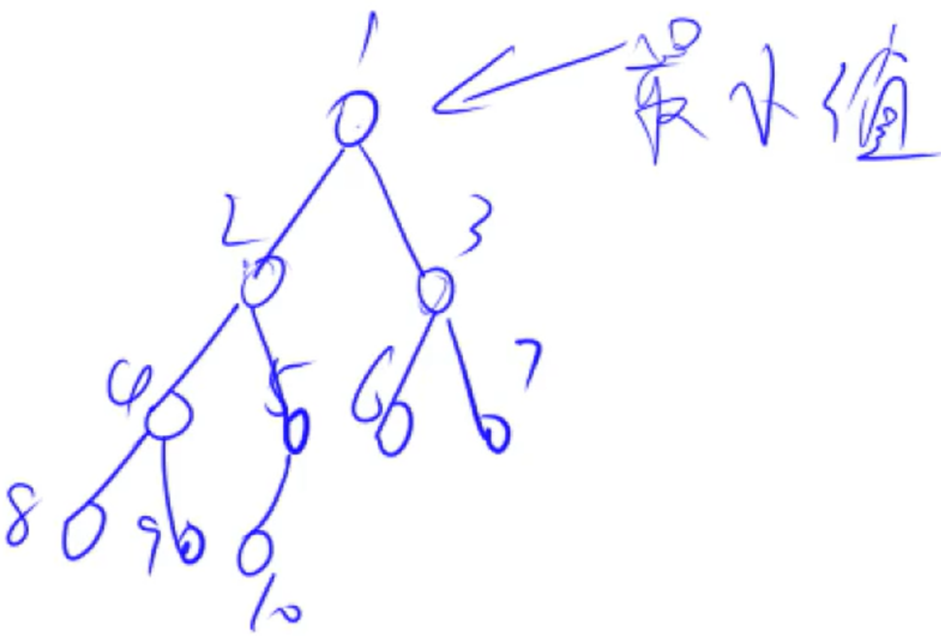
存储方式（用一维数组存储：

1. 下标为`1`的数是根结点
2. `x`结点的左儿子的下标为`2x`，右儿子是`2x + 1`

数的各种结点的下标如下图所示，我们会发现，结点从上到下，从左到右，刚好就是数组的顺序，如果我们用数组`heap[N]`来存储堆的元素，那么`heap[1]`就是根结点，`heap[size]`就是最后一个结点

而且对于完全二叉树而言，下标从`2/n+1`到 `n` 的节点都是叶子节点，`n/2`就是最后一个非叶子结点

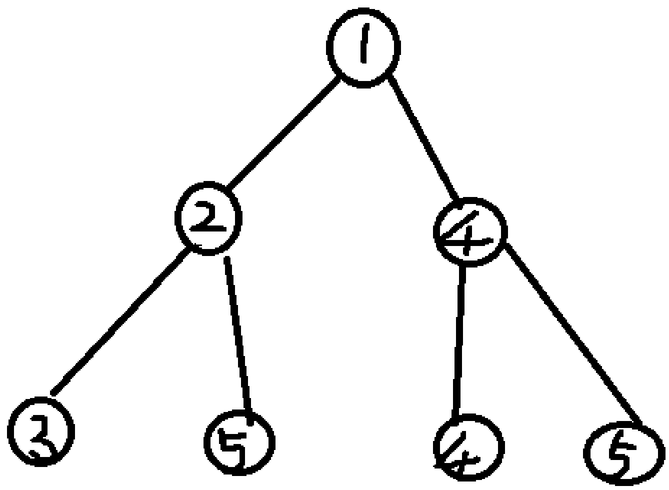
180709



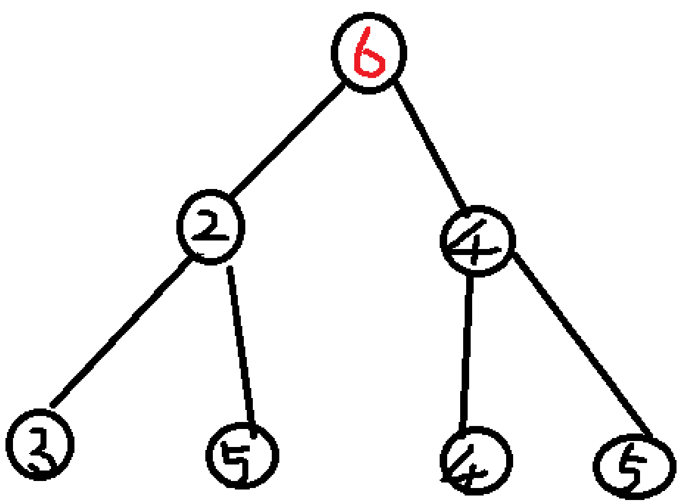
基本操作：

`down(x)`，向下堆化操作：

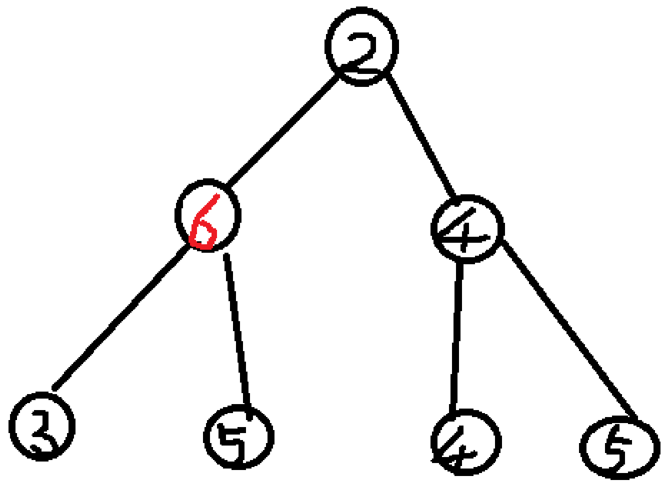
这个操作是将一个结点的值往下移，比如一个堆原来的各结点的值是下图所示：



现在我们将根结点的值换为`6`，那么肯定，这个结点需要往下移动，`down(1)`就是往下移动的过程

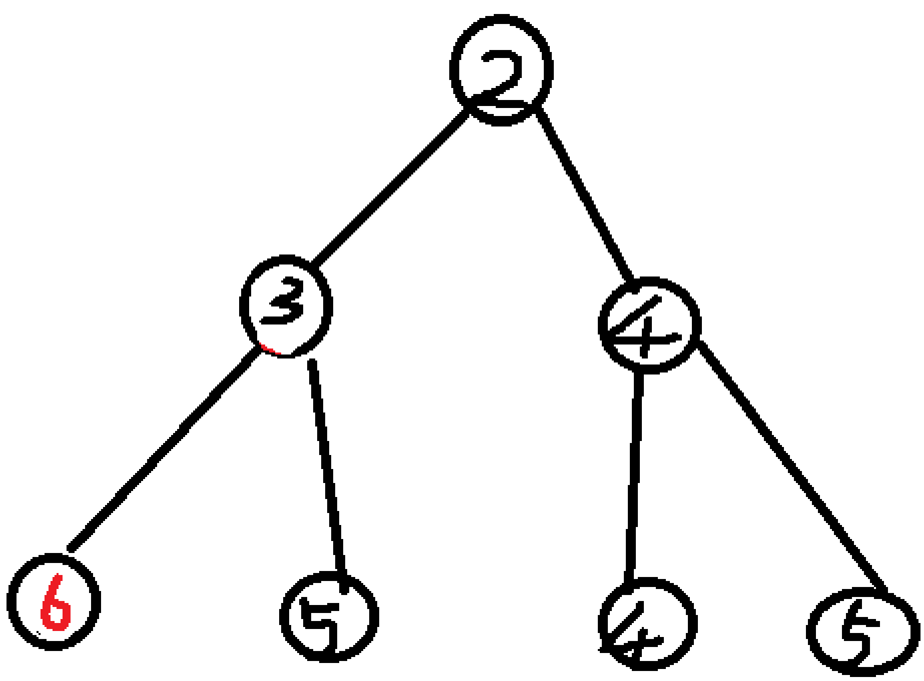


第一，以6为父结点的三个点`2 6 4`中，`2`的值最小，`6`与`2`交换：



这样交换的话仍然满足堆的结构

第二，继续看，以`6`为父结点的三个点`6 3 5`中，`3`的值最小，于是交换：

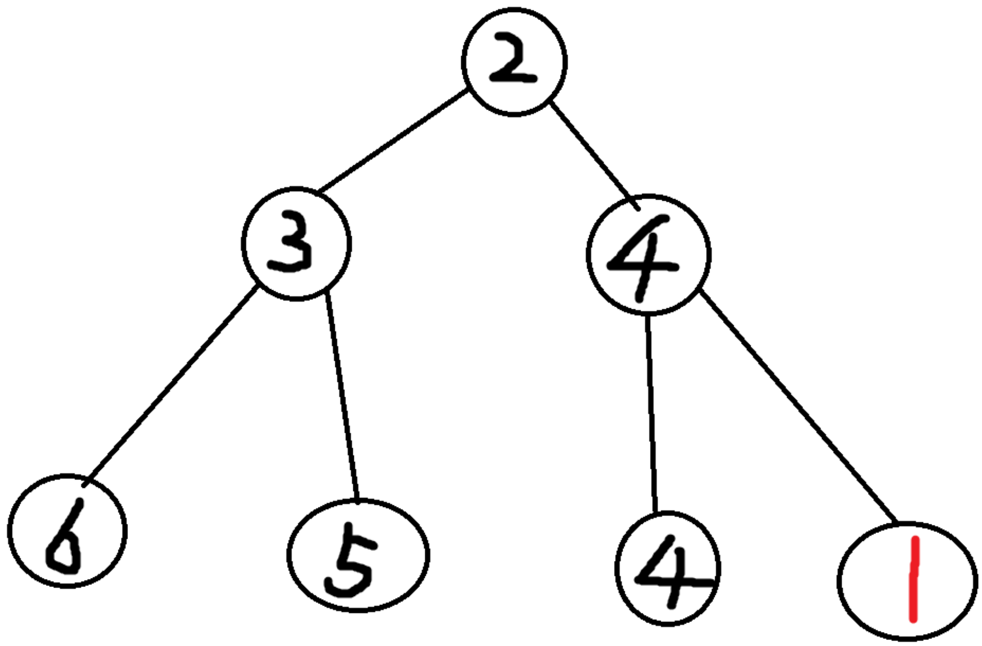


并且由于本来左边的数（`3 5`），都是大于等于换到根结点的数（`2`）的，所以哪怕最底部的那个数（`3`）与`6`交换，它仍然是大于等于根结点的，仍然可以保证堆的结构不发生改变

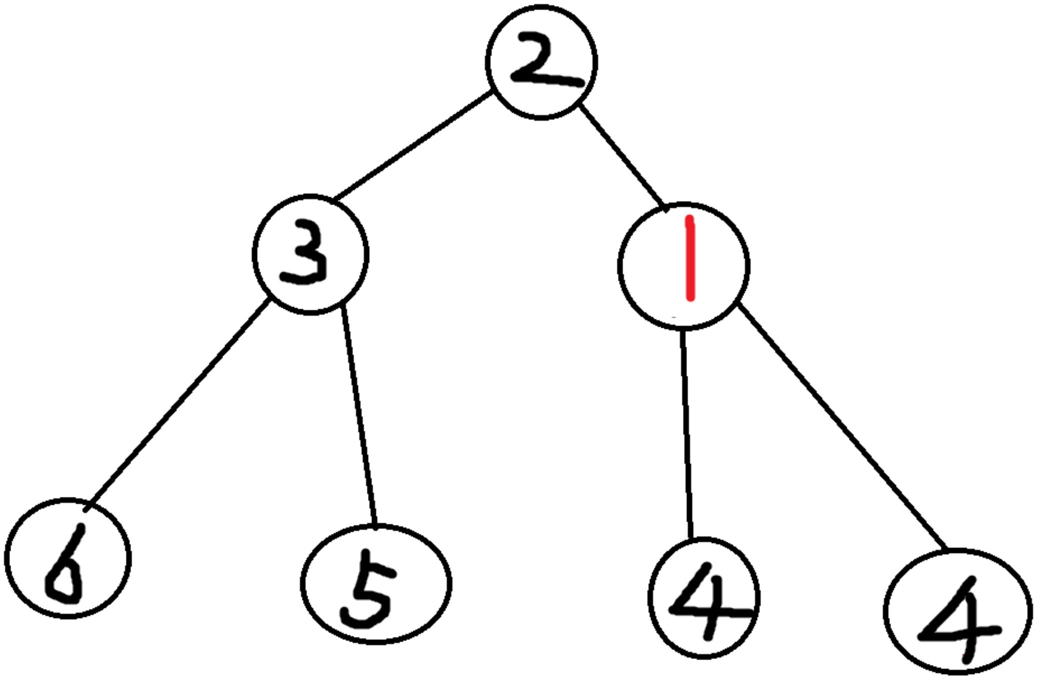
我们发现，通过把改变的这个数与以这个数为父结点的三个数做上述的交换，每一次操作都可以保证操作的那个小三角堆的结构不发生改变，并且重复这个操作，我们就可以将改变值后的堆恢复到原来的结构。

`up(x)`，向上堆化操作：

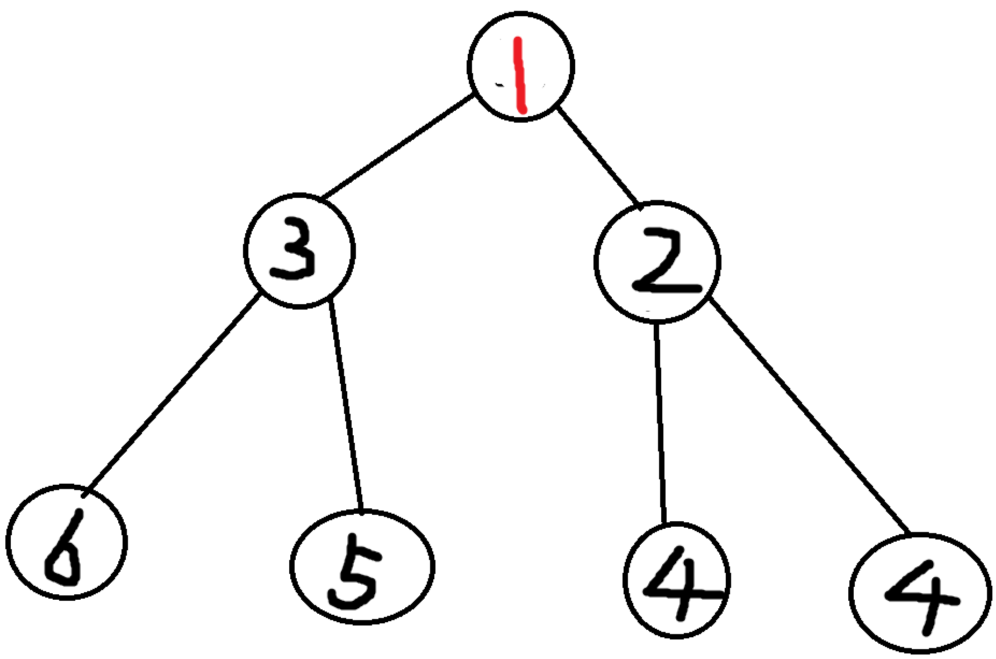
如果我们将最右下角的数变成`1`，则此时不满足了堆的结构，显然，我们需要将`1`移动往上移动即`up(7)`



改变后不满足堆的性质的原因就是这个数小于父结点，而由于其父结点的另一个儿子的值必然大于父结点的值，所以改变的这个值也必然小于其父结点另外一个儿子的值，所以**我们只需要进行这个结点与父结点的交换**



当1与2交换的时候，由于2作为根结点始终小于右子树的所有值，所以交换完成之后仍然保证堆的结构



插入操作：

我们统一在堆的最后面插入一个元素，然后对这个元素不断进行`up`操作

```
heap[++size];
up(size)
```

求最小值：

在小根堆中就是`heap[1]`

删除最小值：

在小根堆中删除第一个元素比较困难，但是可以这样操作：

1. 用最后一个点覆盖掉第一个点，然后堆的总元素`--`
    1. 这一步相当于删除掉最后一个结点，但是由于根结点被最后一个元素覆盖，所以最后一个元素并没有被真的删除
2. 然后再使用`down()`操作，这样就实现了对第一个元素的删除

```
heap[1] = heap[size];
size--;
down(1);
```

删除任意一个元素：

跟上面删除最小元素类似，我们仍然用最后一个元素去覆盖这个元素，然后`size--`表示删除最后一个元素

但是需要注意的是，这个元素被覆盖后可能有两种情况，有可能变大也有可能变小，这时其实我们需要判断一下，不过也可以不用判断，直接两个操作都写上，因为不管怎样都只会执行一个操作

```
heap[k] = heap[size];
size--;
up(k);
down(k);
```

修改任意一个元素:

跟上述类似，不再赘述

```
heap[k] = x;
up(k);
down(k);
```

代码实现

建堆

我们有两种建堆方式

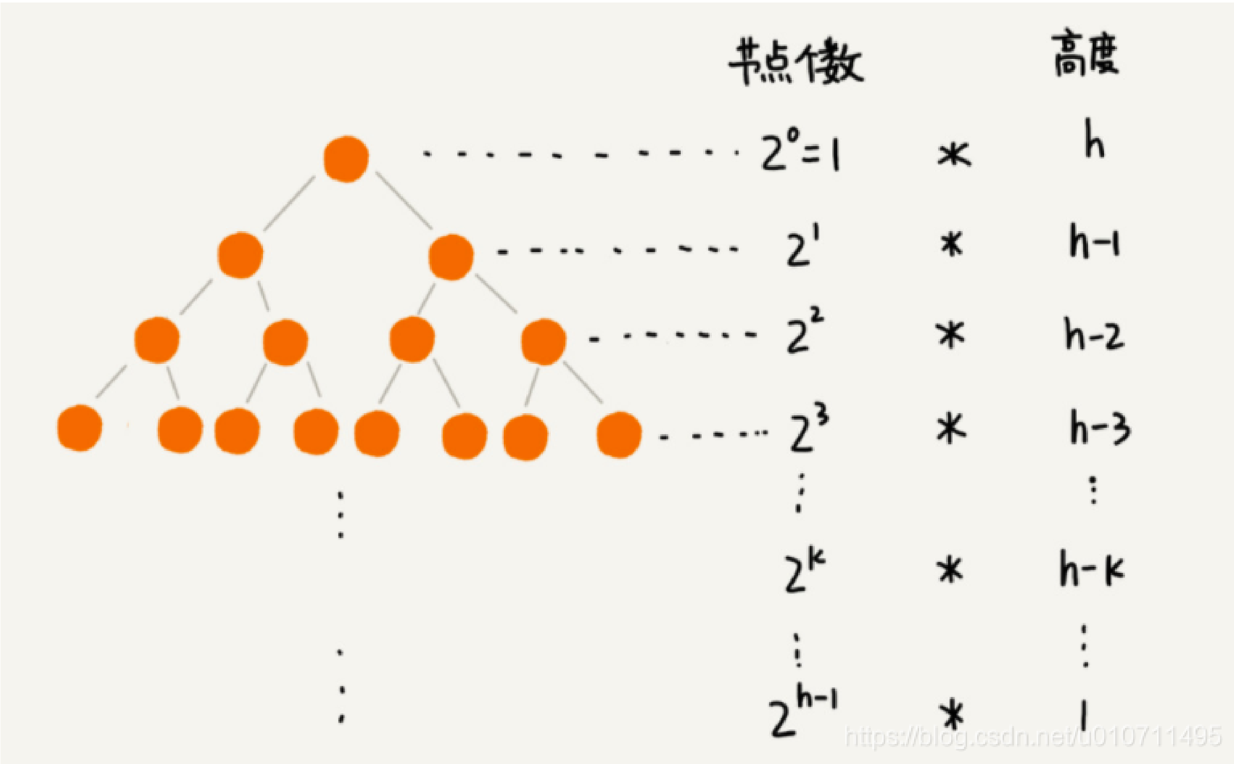
第一种，每读入一个数进入数组中，就将其看作对原堆的插入操作，每读入一个数，就进行一次向上堆化，最后可以得到一个完整的堆

但是这种操作的时间复杂度是`O(nlogn)`的，因为读入`n`个数，每个数都要进行一次向上堆化，而每个结点堆化的时间复杂度是`O(logn)`所以是`O(nlogn)`

第二种，先读入所有的元素到数组中，然后从后往前依次向下堆化

由于叶结点向下堆化的时候只和自己比较不用操作，所以我们从最后一个非叶结点的结点开始向下堆化，在完全二叉树中，这个结点的下标就是`n/2`

我们用一个满完全二叉树（非叶结点数最多）来证明一下时间复杂度



首先，堆化的复杂度与高度是成正比的，我们将所有需要建堆的高度总和求出来，就得到了总的复杂度

根据各种操作，等比数列啥的，这里不再赘述具体参考：[这篇文章]

可以得到总的高度和为：`S = 2^(h+1) - h - 2`

又`h = log2n`，带入可得`S = O(n)`

输入一个长度为 n 的整数数列，从小到大输出前 m 小的数。

输入格式

第一行包含整数 n 和 m。

第二行包含 n 个整数，表示整数数列。

输出格式

共一行，包含 m 个整数，表示整数数列中前 m 小的数。

数据范围

$1 \le m \le n \le 10^5$，
$1 \le \text{数列中元素} \le 10^9$

输入样例：

```
5 3
4 5 1 3 2
```

输出样例：

```
1 2 3
```

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 100010;

int n, m, Size;
int h[N];

void down(int u){
    int t = u;//t来存储三个点中最小值的下标
    //如果左儿子存在并且，左儿子结点的值小于当前结点的值，则t记录为左儿子的下标
    if(u * 2 <= Size && h[u * 2] < h[t]) t = u * 2;
    //上面语句执行后t始终存的是两者比较最小值的下标

    //如果右儿子存在，并且右儿子结点的值小于当前的最小值，则t记录为右儿子的下标
    if(u * 2 + 1 <= Size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;

    if(u != t){//表示不满足堆的结构，需要调整
        swap(h[u], h[t]);//将父结点的值与三者最小的进行交换

        //此时之前存最小值的结点中存放了父结点的值
        //这时需要继续向下堆化，一直到结构符合堆的结构
        down(t);
    }


}
//向上堆化
```

```
void up(int u){
    //当这个结点的父结点存在并且，父结点的值比这个结点的值要大，则需要进行调整
    //如果父结点不存在，表示到了根结点，停止
    //如果父结点比这个结点小，表示符合堆的结构，停止
    while(u / 2 && h[u / 2] > h[u]){
        //交换这两个结点的值
        swap(h[u / 2], h[u]);
        //然后从父结点开始向上递归
        u /= 2;
    }
}

int main(){
    scanf("%d%d", &n, &m);

    for(int i = 1; i <= n; i++){
        scanf("%d", &h[i]);
    }
    Size = n;
    for(int i = n / 2; i >= 1; i --){
        down(i);
    }

    while(m--){//求前m个最小元素
        printf("%d ", h[1]);//先输出堆顶元素
        //再删除堆顶元素，然后调整堆
        h[1] = h[Size];
        Size--;
        down(1);
    }

}
```