Lesson2

> **Trie树(字典树)**
>
> - 基本作用：高效地**存储和查找字符串集合**的数据结构
>
> 存储
>
> - 以字典形式存储，且字母类型较少
> - 每个单词结尾处打上一个标记，便于检索树
>
> 查找
>
> - e1：字符串存储和查找
>
> ```
> static int N = 100010;
> 
> static int n;
> static int[][] son = new int[N][26];
> static int[] cnt = new int[N];
> static int idx; //编号为0的点既是根节点，也是空结点
> 
> static void insert(String s) {
>     int p = 0;
>     for (int i=0; i<s.length(); i++) {
>         int u = s.charAt(i)-'a';
>         if (son[p][u] == 0) son[p][u] = ++idx;
>         p = son[p][u];
>     }
> 
>     cnt[p]++;
> }
> 
> static int query(String s) {
>     int p = 0;
>     for (int i=0; i<s.length(); i++) {
>         int u = s.charAt(i)-'a';
>         if (son[p][u] == 0) return 0;
>         p = son[p][u];
>     }
> 
>     return cnt[p];
> }
> 
> public static void main(String[] args) throws Exception {
>     n = Integer.parseInt(inb.readLine());
> 
>     while (n-- > 0) {
>         String[] ss = inb.readLine().split(" ");
> 
>         if (ss[0].equals("I")) insert(ss[1]);
>         else out.println(query(ss[1]));
>     }
> 
>     out.flush();
> ```

```
}
```

- e2: 最大异或(^)对

```
static int N = 100010, M = 31*N;

static int n;
static int[] a = new int[N];
static int[][] son = new int[M][2];
static int idx;
//由于每个整数固定用31位表示，故不需要结束标记数组

static void insert(int x) {
    int p = 0;
    for (int i=30; i>=0; i--) {
        int u = x>>i&1;
        if (son[p][u] == 0) son[p][u] = ++idx;
        p = son[p][u];
    }
}

static int query(int x) {    //找出x的异或最大值
    int p = 0, res = 0;
    for (int i=30; i>=0; i--) {
        int u = x>>i&1;
        if (son[p][1-u] != 0) {
            res += 1<<i;
            p = son[p][1-u];    //注意使用son[p][1-u]更新p
        }
        else p = son[p][u];
    }

    return res;
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++) {
        ins.nextToken(); a[i] = (int)ins.nval;
        insert(a[i]);
    }

    int res = 0;
    for (int i=0; i<n; i++) res = Math.max(res, query(a[i]));    //循环求出全局最大

    out.print(res);

    out.flush();
}
```

并查集

基本作用：

近乎 O(1) 时间内完成以下两种操作

- 将两个集合合并
- 询问两个元素是否在一个集合中

基本原理

- 每个集合用一棵树表示（不一定是二叉树），以树根结点的编号作为集合的编号。每个结点存储其父结点信息，p[x]表示x的父结点。
  - q1：如何判断树根？ if (p[x] == x)
  - q2：如何求x的集合编号？ while (p[x] != x) p[x] = find(p[x]); （**路径压缩优化** -> O(1)）
  - q3：如何合并两个集合？px是x的集合编号，py是y的集合编号，则p[x] = py 或者 p[y] = px （按秩合并优化，效果不明显）
- e1: 合并集合

```java
static int N = 100010;

static int n, m;
static int[] p = new int[N];

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

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

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

    while (m-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;

        if (op.equals("M")) p[find(a)] = find(b);
        else {
            if (find(a) == find(b)) out.println("Yes");
            else out.println("No");
        }
    }

    out.flush();
}
```

维护额外信息

- e2: 连通块中点的数量（额外维护每个连通块中点的数量）

```java
static int N = 100010;

static int n, m;
static int[] p = new int[N], cnt = new int[N];

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

public static void main(String[] args) throws Exception {
    String[] ss = inb.readLine().split(" ");
    n = Integer.parseInt(ss[0]); m = Integer.parseInt(ss[1]);

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

    while (m-- > 0) {
        ss = inb.readLine().split(" ");

        if (ss[0].equals("C")) {
            int a = Integer.parseInt(ss[1]), b = Integer.parseInt(ss[2]);

            a = find(a); b = find(b);
            p[a] = b;
            if (a != b) cnt[b] += cnt[a];
        }
        else if (ss[0].equals("Q1")) {
            int a = Integer.parseInt(ss[1]), b = Integer.parseInt(ss[2]);

            if (find(a) == find(b)) out.println("Yes");
            else out.println("No");
        }
        else {
            int a = Integer.parseInt(ss[1]);

            out.println(cnt[find(a)]);
        }
    }

    out.flush();
}
```

- e3: **食物链**

```java
static int N = 50010;

static int n, k;
static int[] p = new int[N], d = new int[N];     //d维护每个结点到根结点的距离信息

static int find(int x) {
    if (x != p[x]) {
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }

    return p[x];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); k = (int)ins.nval;

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

    int res = 0;
    while (k-- > 0) {
        ins.nextToken(); int t = (int)ins.nval;
        ins.nextToken(); int x = (int)ins.nval;
        ins.nextToken(); int y = (int)ins.nval;

        if (x > n || y > n) res++;
        else {
            if (t == 1) {
                int px = find(x), py = find(y);d
                if (px == py && (d[x]-d[y]) % 3 != 0) res++;
                else if (px != py) {
                    p[px] = py;
                    d[px] = d[y]-d[x];
                }
            }
            else if (t == 2){
                int px = find(x), py = find(y);
                if (px == py && (d[x]-d[y]+1) % 3 != 0) res++;
                else if (px != py) {
                    p[px] = py;
                    d[px] = d[y]-d[x]-1;
                }
            }
        }
    }

    out.println(res);

    out.flush();
}
```

堆（手写堆）

堆是一颗**完全二叉树**

支持操作

- 插入一个数

  heap[++cnt] = x; up(cnt);

- 求集合中的最小值

  heap[1];

- 删除最小值

  heap[1] = heap[cnt]; cnt—; down(1);

—————— STL容器（优先队列）不支持的操作 ——————

- 删除任意一个元素

  heap[k] = heap[cnt]; cnt—; down(k); up(k); （down与up只会执行一个）

- 修改任意一个元素

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

基本性质（以小根堆为例）

- 每一个点小于等于左右儿子（递归定义）
- 根结点是最小值（小根堆）

堆的存储（使用**一维数组**进行存储（堆，完全二叉树），**下标从1开始（kmp，前缀差分，堆存储）**）

O(n)建堆方式：从n/2处 down 到 1

基本操作

- down(x)：x增大，往下调整x的位置（小根堆）
- up(x)：x减小，往上调整x的位置（小根堆）
- e1: 堆排序

```
static int N = 100010;

static int n, m;
static int cnt;
static int[] h = new int[N];

static void down(int u) {
    int t = u;
    if (2*u <= cnt && h[2*u]<h[t]) t = 2*u;
    if (2*u+1 <= cnt && h[2*u+1]<h[t]) t = 2*u+1;
    if (t != u) {
        int tp = h[u]; h[u] = h[t]; h[t] = tp;
        down(t);
    }
}

static void up(int u) {
    while (u/2 > 0 && h[u/2] > h[u]) {
        int tp = h[u]; h[u] = h[u/2]; h[u/2] = tp;
        u >>= 1;
    }
```

```java
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) { ins.nextToken(); h[i] = (int)ins.nval; }
    cnt = n;

    for (int i=n/2; i>0; i--) down(i);  //O(n)方式建堆

    while (m-- > 0) {
        out.print(h[1]+" ");
        h[1] = h[cnt--];
        down(1);
    }

    out.flush();
}
```

- e2: 模拟堆

  修改和删除任意一个元素需要额外维护两个数组。

  - ph[k]：第k个插入的数在堆中的下标
- hp[i]：下标是i的点是第几个插入的数

```java
static int N = 100010;

static int n;
static int cnt;
static int[] h = new int[N], ph = new int[N], hp = new int[N];

static void swap(int[] a, int x, int y) {
    int tp = a[x]; a[x] = a[y]; a[y] = tp;
}

static void heap_swap(int a, int b) {
    // swap(ph, hp[a], hp[b]);     //注意函数参数是下标!
    // swap(hp, a, b);
    // swap(h, a, b);
    int tp = ph[hp[a]]; ph[hp[a]] = ph[hp[b]]; ph[hp[b]] = tp;
    tp = hp[a]; hp[a] = hp[b]; hp[b] = tp;
    tp = h[a]; h[a] = h[b]; h[b] = tp;
}

static void down(int u) {
    int t = u;
    if (2*u <= cnt && h[2*u] < h[t]) t = 2*u;
    if (2*u+1 <= cnt && h[2*u+1] < h[t]) t = 2*u+1;
    if (t != u) {
        heap_swap(t, u);
        down(t);
    }
}
```

```java

static void up(int u) {
    while (u/2 > 0 && h[u/2] > h[u]) {
        heap_swap(u/2, u);
        u >>= 1;
    }
}


public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    int kk = 0;
    while (n-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;

        if (op.equals("I")) {
            ins.nextToken(); int x = (int)ins.nval;
            cnt++; kk++;
            ph[kk] = cnt; hp[cnt] = kk;
            h[cnt] = x; up(cnt);
        }
        else if (op.equals("PM")) out.println(h[1]);
        else if (op.equals("DM")) {
            heap_swap(1, cnt--); down(1);
        }
        else if (op.equals("D")) {
            ins.nextToken(); int k = (int)ins.nval;
            k = ph[k];
            heap_swap(k, cnt--); down(k); up(k);
        }
        else {
            ins.nextToken(); int k = (int)ins.nval;
            ins.nextToken(); int x = (int)ins.nval;
            k = ph[k];
            h[k] = x; down(k); up(k);
        }
    }

    out.flush();
}
```