树及其应用

树及表示

　我们之前介绍了线性表这一类数据结构，并且学习了如何使用线性表解决一类特定的问题（数据具有明显的前后关系，可以进行线性连接）。我们继续学习一类新的数据结构—树。

树形结构不仅能表示数据间的指向关系，还能表示出数据的层次关系，而有很明显的递归性质。 因此，我们可以利用树的性质解决更多种类的问题

树是非线性数据结构，它能很好地描述数据的层次关系。树形结构的现实场景很常见，例如：

- 文件目录
- 书本的目录

二叉树是最常用的树形结构，特别适合程序设计，常常一般的树转换成二叉树来处理。

二叉树及表示

$Pascal$ 之父尼古拉斯·沃斯(1984年图灵奖）提出一个著名的公式:算法＋数据结构=程序。

编写程序的一个基本问题就是数据处理，包括如何存储输入的数据、如何组织程序中的中间数据等。这个技术就是数据结构。数据结构和算法不同，它并不直接解决问题，但是数据结构是算法不可分割的一部分。数据结构把杂乱无章的数据有序地组织起来，逻辑清晰，易于编程处理；其次，数据结构便于算法高效地访问和处理数据，大大减少空间和时间复杂度。

淘汰赛

题目描述

有 2^n （$n \le 7$）个国家参加世界杯决赛圈且进入淘汰赛环节。已经知道各个国家的能力值，且都不相等。能力值高的国家和能力值低的国家踢比赛时高者获胜。1 号国家和 2 号国家踢一场比赛，胜者晋级。3 号国家和 4 号国家也踢一场，胜者晋级……晋级后的国家用相同的方法继续完成赛程，直到决出冠军。给出各个国家的能力值，请问亚军是哪个国家?

输入格式

第一行一个整数 n，表示一共 2^n 个国家参赛。

第二行 2^n 个整数，第 i 个整数表示编号为 i 的国家的能力值（$1 \le i \le 2^n$）。

数据保证不存在平局。

输出格式

仅一个整数，表示亚军国家的编号。

样例 #1

样例输入 #1

```
3
4 2 3 1 10 5 9 7
```

样例输出 #1

```
1
```

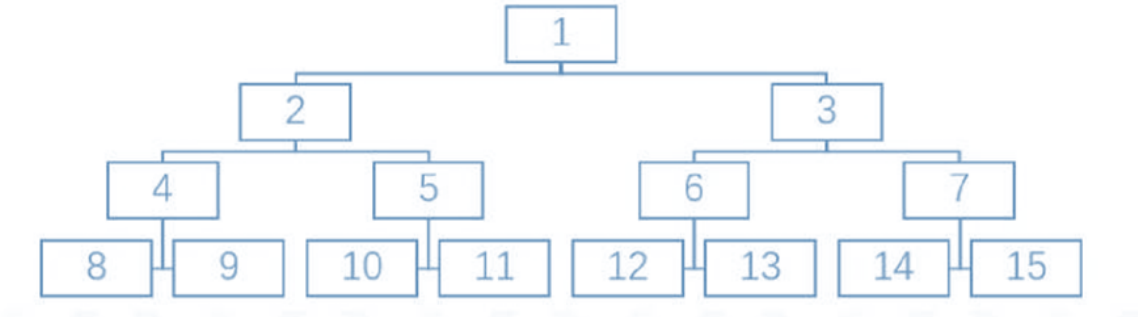
题目分析

假设$n=3$，有8个国家参加，他们在比赛中的能力值分别是（$4,2,3,1,10,5,9,7$），试试看，根据二叉树的特点画出赛程图。

可以看到，从冠军（根结点）往下面看，每个获胜者结点下面都是2个国家。这就是一棵典型的二叉树。更加严格的递归定义是：二叉树要么为空，要么由根结点、左子树、右子树构成, 而左右子树分别还是一棵二叉树。如果一个结点没有任何子树，那就被称为叶子结点。

这个赛程图经过了3轮比赛，一共有4层结点，所以这个二叉树的高度是4。如果一个二叉树高度为h，从第二层开始每一层的结点数都是上一层的两倍，一共有2^h-1个结点的二叉树被称为完美二叉树（你看这棵树多么丰满）。

对于完美二叉树，可以从上到下，从左到右，对各个结点从1开始分配编号。如图：



有没有发现一些规律？对于i号非叶子结点，它的左子树编号是$2\times i$，右子树的编号是$2\times i+1$。这样就可以创建若干足够大的数组将各个结点的信息记录进去，通过计算编号来访问左右子树，并使用递归的方式得到各个子树的统计。

考虑到这是一个完美二叉树，1到$(1<<n)-1$编号都是非叶子结点，编号不小于$1>>n$都是叶子结点。本题使用$value[i]$来记录叶子结点的实力值，或者是非叶子结点的该子树的最大值；使用$winner[i]$来记录该子树的获胜者。当所有比赛模拟完毕，$winner[1]$记录了整场比赛的冠军，$value[i]$记录了冠军的实力值。这时我们要比较一下谁是决赛败者，输出败者的国家编号，即整场比赛的亚军。

由于需要维护和子树相关的两个值——$value$和$winner$，所以建立了这两个数组存储子树的信息。有时只需要维护子树的一个信息,也可不用建立这样的数组，而是将dfs函数定义为具有返回值函数, 然后在递归函数中处理这个值。

参考代码

```
#include<cstdio>
#include<iostream>
using namespace std;
int value[260],winner[260];
int n;
void dfs(int x) {
    if (x>=1<<n)      //如果是叶子结点就不要继续遍历下去了
        return;
    else {
        dfs(2*x); //遍历左子树
        dfs(2*x+1);  //遍历右子树
        int lvalue=value[2*x],rvalue=value[2*x+1];
        if (lvalue>rvalue) {      // 左结点获胜
            value[x]=lvalue;      //记录下获胜方的能力值
            winner[x]=winner[2*x]; // 和获胜方的编号
        } else { // 右结点获胜
            value[x]=rvalue;
            winner[x]=winner[2*x+1];
        }
    }
}
int main() {
    cin>>n;
    for(int i=0;i<1<<n;i++) {
        cin>>value[i+(1<<n)]; //读入各个结点的能力值
```

```
            winner[i+(1<<n)]=i+1; //叶子结点的获胜方就是自己的编号
        }
        dfs(1); //从根结点开始遍历
        if(value[2]>value[3])cout<<winner[3];
        else cout<<winner[2];        //找亚军
        return 0;
}
```

二叉树的性质

·二叉树的每个结点最多有两个子结点，分别是左孩子、右孩子，以它们为根的子树称为左子树、右子树。

·二叉树的第i层最多有2^{i-1}个结点。如果每一层的结点数都是满的，称它为满二叉树。一个 n 层的满二叉树，结点数量一共有2^n-1 个，可以依次编号为 $1,2,3,\cdots,2^n-1$ 。如果满二叉树只在最后一层有缺失，并且缺失的编号都在最后，那么称为完全二叉树。满二叉树和完全二叉树如图所示。

满二叉树　　　　完全二叉树

完全二叉树的性质

一棵结点数量为 k 的完全二叉树，设1号点为根结点，有以下性质：

· $i>1$ 的结点，其父结点是 $i/2$

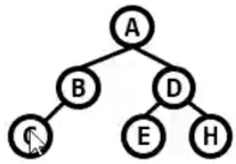
· 如果 $2i>k$，那么 i 没有孩子；如果$2i+1>k$，那么 i 没有右孩子；

·如果结点 i 有孩子，它的左孩子是$2i$，右孩子是$2i+1$。

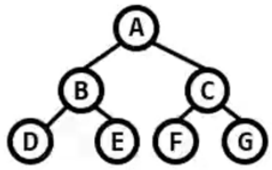
二叉树的存储

顺序存储：对于一个二叉树的所有结点按层编号，将编号为 i 的结点存入一位数组的第 i 个单元。对**完全二叉树**当中的任何一个结点（设编号为 x），启左孩子的编号一定是 $2x$ ，而右孩子的编号一定是 $2x+1$

1	2	3	4	5	6	7
A	B	D	C	#	E	H



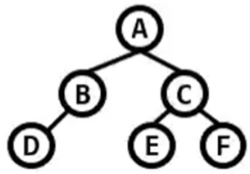
1	2	3	4	5	6	7
A	B	C	D	E	F	G



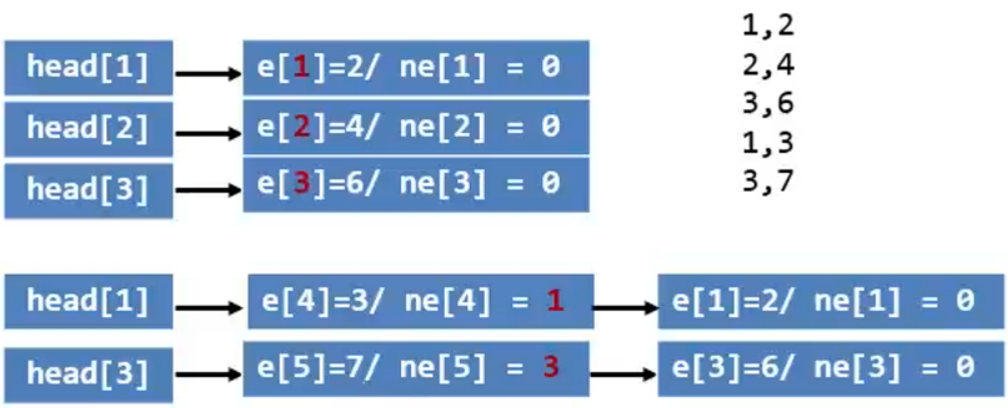
链表存储：结点的左右指针域使用 int 代替，用来表示左右子树的根节点在数组的下标

```
// 链表 存储：结构体
struct node{
    int left;  //左孩子下标
    int right; //右孩子下标
    int val;   //当前结点的数值
}
```

	1	2	3	4	5	6
data	A	B	C	D	E	F
left	2	4	6	0	0	0
right	3	0	7	0	0	0



数组模拟链表：链式前向星



链式前向星一般是用来存图。树属于图的一种。
我们可以发现，数组模拟链表的方式和 $vector$ 动态数组很像，$vector$是从后面压入子结点，而链式前向星是从前面压入子结点。这也是它名字的来源。链式前向星相比$vector$要更灵活一点，在空间非常紧张的情况，$vector$是有可能爆空间的，而链式前向星爆空间的情况较少。

```
// 链式前向星写法
int e[N],ne[N],head[N],cnt=0;
```

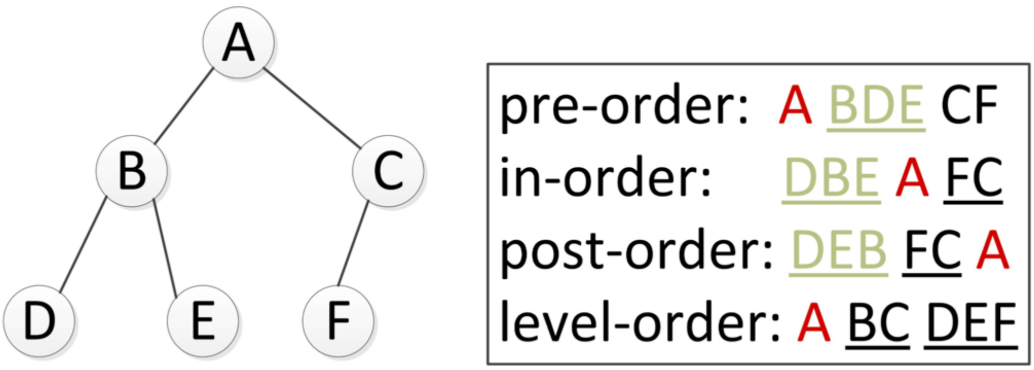
```
add(int a,int b) // 父结点是a
{
    e[cnt]=b; //添加新结点
    ne[cnt]=head[a]; //新结点的ne指向原来head指向的结点
    head[a]=cnt; //头指针指向新结点
    cnt++; // 添加下一个新结点
}
// 链式前向星简写
add(int a,int b)
{
    e[++cnt]=b;
    ne[cnt]=head[a];
    head[a]=cnt;
}

```

二叉树遍历

二叉树的遍历是指通过一定顺序访问二叉树的所有结点。遍历方法一般有四种：先序遍历、中序遍历、后序遍历及层次遍历，其中，前三种一般使用深度优先搜索（DFS）实现，而层次遍历一般用广度优先搜索（BFS）实现。

层次遍历

直接对二叉树进行广度优先搜索（即队列），将根结点放入初始队列中，取出每次出队的结点，即可得到层次遍历。取出时别忘了把这个结点的子结点放到队伍的末尾。



前中后遍历

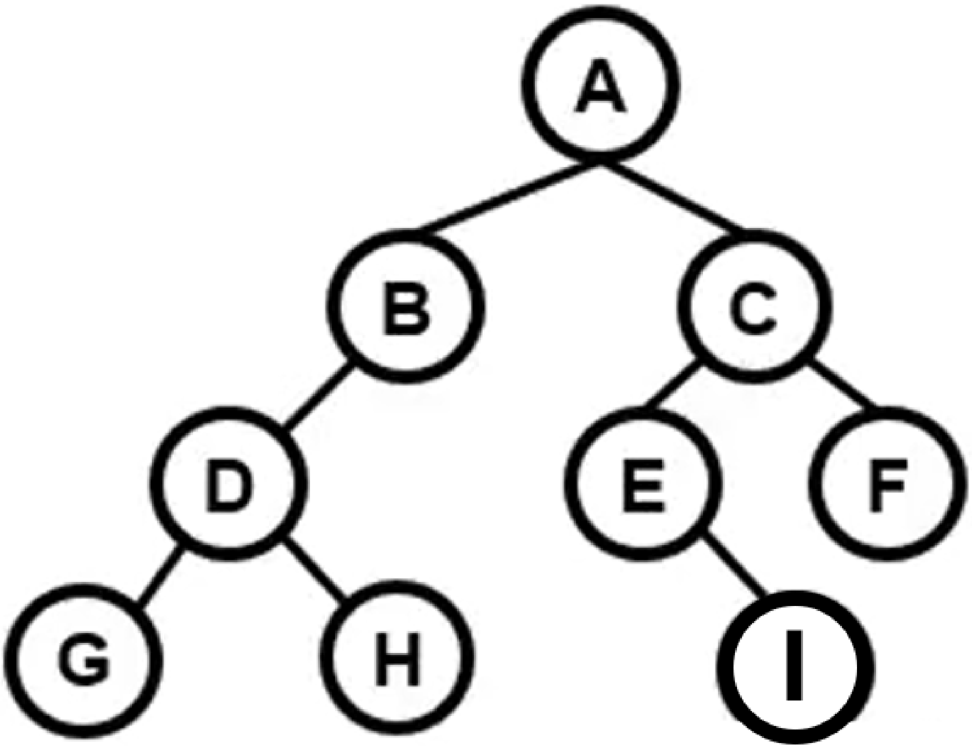
对于任意给定结点，可以访问该结点本身、遍历左子树、遍历右子树。根据在某个结点中遍历的顺序不同，有以下三种遍历方式：

1.前序遍历： 根结点、左子树、右子树

2.中序遍历：左子树、根结点、右子树

3.后续遍历：左子树、右子树、根结点

无论是哪种遍历方式，本质上都是深度优先遍历，只是递归的顺序不一样。其实二叉树 的遍历本质上也是**深度优先搜索**。



先序遍历的访问顺序是：根结点→左子树→右子树。访问顺序为：$A(BDGH)(CEIF)$

中序遍历的访问顺序是：左子树→根结点→右子树。访问顺序为：$(GDHB)A(EICF)$

后序遍历的访问顺序是：左子树→右子树→根结点。访问顺序为：$(GHDB)(IEFC)A$

注意：无论是哪一种遍历，左子树一定优先于右子树遍历。

```
#include<bits/stdc++.h>
using namespace std;
const int N=10010;

struct node {
    char data;
    int Left,Right;
}a[N];

void pre_order(int root)
{
    // 先序遍历：根左右
    if (root==0) return ;
    cout<<a[root].data<<" ";
    pre_order(a[root].Left);
    pre_order(a[root].Right);
}

void in_order(int root)
{
    // 中序遍历：左根右
```

```
    if (root==0) return ;
    in_order(a[root].Left);
    cout<<a[root].data<<" ";
    in_order(a[root].Right);
}

void post_order(int root)
{
    // 后序遍历：左右根
    if (root==0) return ;
    post_order(a[root].Left);
    post_order(a[root].Right);
    cout<<a[root].data<<" ";
}

int main()
{
    // 根据图创建二叉关系图
    a[1].data='A',a[1].Left=2,a[1].Right=3;
    a[2].data='B',a[2].Left=4,a[2].Right=0;
    a[3].data='C',a[3].Left=5,a[3].Right=6;
    a[4].data='D',a[4].Left=7,a[4].Right=8;
    a[5].data='E',a[5].Left=0,a[5].Right=9;
    a[6].data='F',a[6].Left=0,a[6].Right=0;
    a[7].data='G',a[7].Left=0,a[7].Right=0;
    a[8].data='H',a[8].Left=0,a[8].Right=0;
    a[9].data='I',a[9].Left=0,a[9].Right=0;
    pre_order(1);
    cout<<endl;
    in_order(1);
    cout<<endl;
    post_order(1);
    return 0;
}
```

新二叉树

题目描述

输入一串二叉树，输出其前序遍历。

输入格式

第一行为二叉树的节点数 n。$(1 \le n \le 26)$

后面 n 行，每一个字母为节点，后两个字母分别为其左右儿子。特别地，数据保证第一行读入的节点必为根节点。

空节点用 `*` 表示

输出格式

二叉树的前序遍历。

样例 #1

样例输入 #1

```
6
abc
bdi
cj*
d**
i**
j**
```

样例输出 #1

```
abdicj
```

题目分析

输入一串二叉树，用先序遍历输出。

这道题需要解决的是确定每个字母对应的编号，比如 $a \to 1$

参考代码

```
#include<bits/stdc++.h>
using namespace std;
const int N=30;
int n;
struct node{
    char data;
    int Left,Right;
}a[N];

void dfs(int root)
{
    if (root==0) return ;
    cout<<a[root].data;
    dfs(a[root].Left);
    dfs(a[root].Right);
 }

 int main()
 {
    cin>>n;
    char x,y,z;
    int root=0; // 记录根节点
    for (int i=1;i<=n;i++)
    {
        cin>>x>>y>>z;
        if (!root) root=x-'a'+1;
```

```
            int t=x-'a'+1;
            a[t].data=x;
            if (y>='a' && y<='z') a[t].Left=y-'a'+1;
            if (z>='a' && z<='z') a[t].Right=z-'a'+1;
        }
    dfs(root);
    return 0;
}
```

[NOIP2001 普及组] 求先序排列

题目描述

给出一棵二叉树的中序与后序排列。求出它的先序排列。（约定树结点用不同的大写字母表示，且二叉树的节点个数 ≤ 8）。

输入格式

共两行，均为大写字母组成的字符串，表示一棵二叉树的中序与后序排列。

输出格式

共一行一个字符串，表示一棵二叉树的先序。

样例 #1

样例输入 #1

```
BADC   // 中序
BDCA   // 后序
```

样例输出 #1

```
ABCD
```

提示

【题目来源】

NOIP 2001 普及组第三题

题目分析

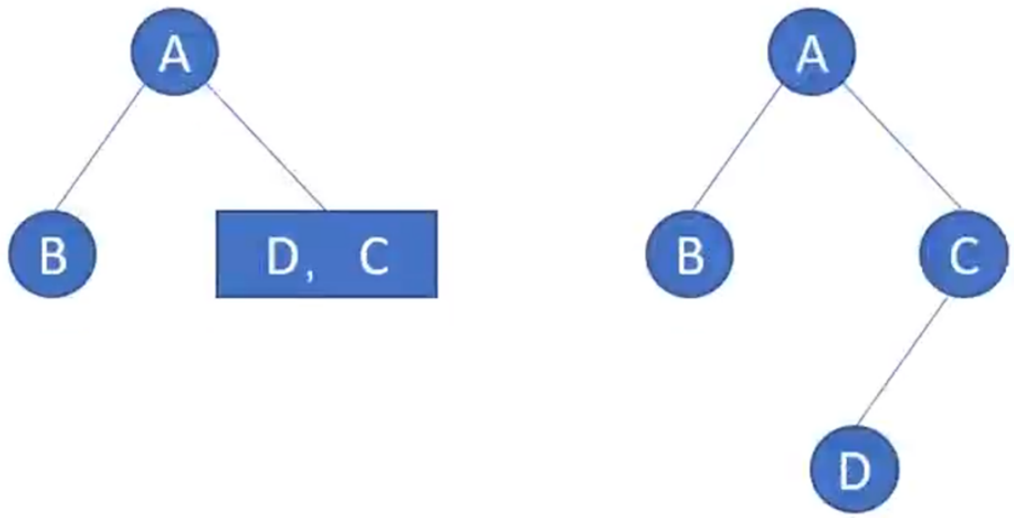
已知中序遍历，后序遍历求先序遍历。

中序遍历：$BADC$

后序遍历：$BDCA$

后序遍历最后一个数是整棵树的根。知道了\\A//是根，对照中序遍历，\\A//左边的\\B//为左子树，\\DC//为右子树。

递归上述过程。后序遍历最后一个数是树的根。知道\\C//是根，对照中序遍历，\\C//的左子树为\\D//。



参考代码1

```
#include<bits/stdc++.h>
using namespace std;
string os,ps;
char root;
void dfs(string a,string b)
{
    // 中序字符，后序字符
    int alen=a.length(),blen=b.length();
    if (alen<=0) return; // 如果当前中序只有1就不需要再搜索左右子树了
    root=b[blen-1]; // 获取根节点
    int dir=a.find(root); // 找到中序遍历该点的位置
    cout<<root; // 输出根节点
    dfs(a.substr(0,dir),b.substr(0,dir)); //先搜索左子树
    dfs(a.substr(dir+1),b.substr(dir,blen-dir-1)); // 搜索右子树
 }
 int main()
 {
    cin>>os>>ps;
    dfs(os,ps);
    return 0;
  }
```

参考代码2

```
#include<bits/stdc++.h>
using namespace std;
const int N=10;
char os[N],as[N]; // 中序与后序
```

```
int len;
int Find(char root)
{
    // 注意传递进来的地址，不可能计算出字符串长度的
    for (int i=0;i<len;i++)
    {
        if (os[i]==root) return i;
     }
    return -1; // 如果没找到 -1
 }

 void dfs(int L1,int R1,int L2,int R2)
 {
    // 中序后序的起点和终点
     int k=Find(as[R2]); // 根节点
     cout<<as[R2]; // 先输出根结点
     if (k>L1) dfs(L1,k-1,L2,R2-R1+k-1); //搜搜左子树
     if (k<R1) dfs(k+1,R1,L2-L1+k,R2-1); //搜索右子树
  }

  int main()
  {
    cin>>os>>as;
    len=strlen(as); // 计算长度
    dfs(0,len-1,0,len-1);
    return 0;
   }
```

美国血统 American Heritage

题目描述

农夫约翰非常认真地对待他的奶牛们的血统。然而他不是一个真正优秀的记帐员。他把他的奶牛 们的家谱作成**二叉树**，并且把二叉树以更线性的"**树的中序遍历**"和"**树的前序遍历**"的符号加以记录而 不是用图形的方法。

你的任务是在被给予奶牛家谱的"树中序遍历"和"树前序遍历"的符号后，创建奶牛家谱的"树的 **后序遍历**"的符号。每一头奶牛的姓名被译为一个唯一的字母。（你可能已经知道你可以在知道树的两 种遍历以后可以经常地重建这棵树。）显然，这里的树不会有多于 26 个的顶点。 这是在样例输入和 样例输出中的树的图形表达方式：

```
               C
             /   \
            /     \
           B       G
          / \     /
         A   D   H
            / \
           E   F

```

树的中序遍历是按照左子树，根，右子树的顺序访问节点。

树的前序遍历是按照根，左子树，右子树的顺序访问节点。

树的后序遍历是按照左子树，右子树，根的顺序访问节点。

输入格式

第一行： 树的中序遍历

第二行： 同样的树的前序遍历

输出格式

单独的一行表示该树的后序遍历。

样例 #1

样例输入 #1

```
ABEDFCHG
CBADEFGH
```

样例输出 #1

```
AEFDBHGC
```

提示

题目翻译来自NOCOW。

USACO Training Section 3.4

题目分析

从题意可知:

先序遍历和中序遍历，求后序遍历

参考代码

```
#include<bits/stdc++.h>
using namespace std;
string mid,pre;
void dfs(string a,string b)
{
    if (a.size()<=0) return ;
    char fa=b[0]; // 父结点是前序遍历的第一个字母
    int idx=a.find(fa);
    dfs(a.substr(0,idx),b.substr(1,idx)); // 左子树
    dfs(a.substr(idx+1),b.substr(idx+1)); // 右子树
```

```
    cout<<fa;
}
int main()
{
    cin>>mid>>pre; // 输入中序遍历和前序遍历
    dfs(mid,pre); // 搜索
    return 0;
}
```

二叉树深度

题目描述

有一个 $n(n \le 10^6)$ 个结点的二叉树。给出每个结点的两个子结点编号（均不超过 n），建立一棵二叉树（根节点的编号为 1），如果是叶子结点，则输入 `0 0`。

建好这棵二叉树之后，请求出它的深度。二叉树的**深度**是指从根节点到叶子结点时，最多经过了几层。

输入格式

第一行一个整数 n，表示结点数。

之后 n 行，第 i 行两个整数 l、r，分别表示结点 i 的左右子结点编号。若 $l=0$ 则表示无左子结点，$r=0$ 同理。

输出格式

一个整数，表示最大结点深度。

样例 #1

样例输入 #1

```
7
2 7
3 6
4 5
0 0
0 0
0 0
0 0
```

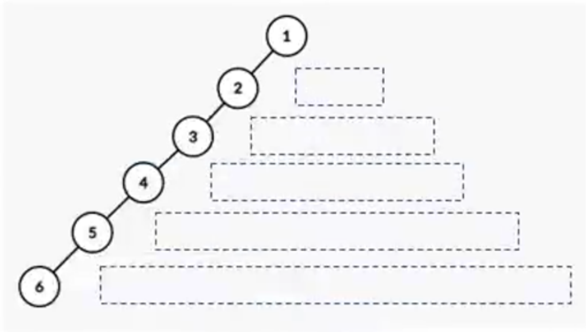
样例输出 #1

```
4
```

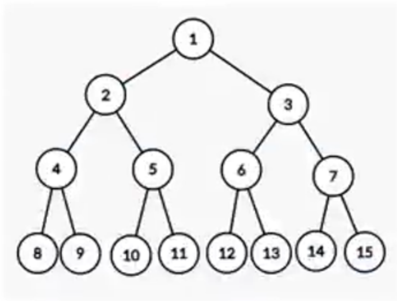
题目分析

数组存储子结点：

　　用数组存储子结点的话，一般需要把数组开大。如果是完全二叉树或者是满二叉树的话，数组开到10^6的4倍差不多就够了。但是如果这根树"退化"成链状，即每个结点只有左儿子，没有右儿子(树)(但是右儿子(树)的空间是开好的，却没用上)，势必会造成极大的浪费。退化后成链后，数组得开到2^n(n为深度，不是题目的结点)。所以深度最好不能超过24。否则空间容易爆。



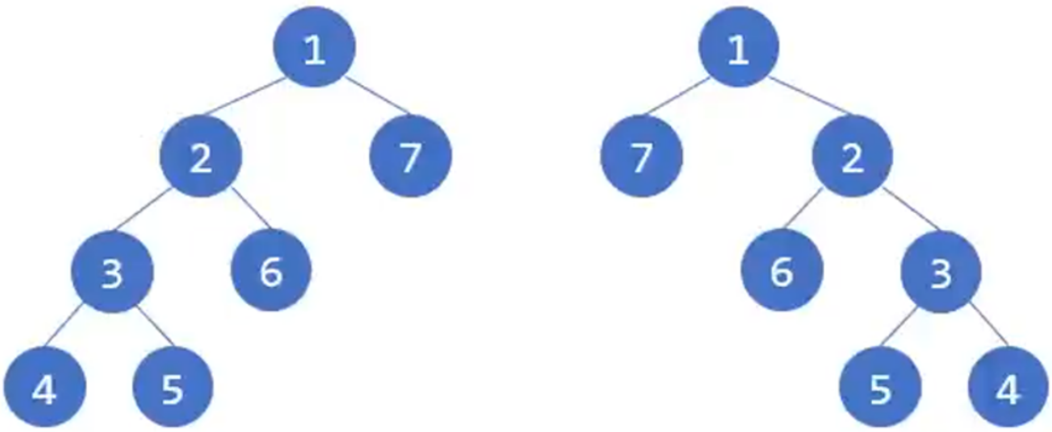
树退化成链



二叉树

结构体存储子结点：

　　每个结点只有两个儿子，在一个结构体内定义两个成员变量。有结点就添加结点。空间为$2\times n+10$的大小(n为结点)。



思路：

每次搜索左右子树，深度 $+1$ 。

注意取左右子树的最大值。

参考代码

```
#include<iostream>
using namespace std;
const int N=1000010; // 结点数
struct node{
    int left,right;
```

```
}a[N];

int dfs(int x)
{
    if (!x) return 0; // 如果没有子结点了
    return max(dfs(a[x].left),dfs(a[x].right))+1;
 }

int main()
{
    int n;cin>>n;
    for (int i=1;i<=n;i++)
    {
        int x,y;
        cin>>x>>y;
        a[i].left=x;
        a[i].right=y;
    }
    int dep=dfs(1); // 从第一层开始
    cout<<dep;
    return 0;
}
```

普通二叉树（简化版）

题目描述

您需要写一种数据结构，来维护一些数（都是 10^9 以内的数字）的集合，最开始时集合是空的。其中需要提供以下操作，操作次数 q 不超过 10^4：

1. 查询 x 数的排名（排名定义为比当前数小的数的个数 $+1$。若有多个相同的数，应输出最小的排名）。
2. 查询排名为 x 的数。
3. 求 x 的前驱（前驱定义为小于 x，且最大的数）。若未找到则输出 -2147483647。
4. 求 x 的后继（后继定义为大于 x，且最小的数）。若未找到则输出 2147483647。
5. 插入一个数 x。

输入格式

第一行是一个整数 q，表示操作次数。

接下来 q 行，每行两个整数 op, x，分别表示操作序号以及操作的参数 x。

输出格式

输出有若干行。对于操作 $1, 2, 3, 4$，输出一个整数，表示该操作的结果。

样例 #1

样例输入 #1

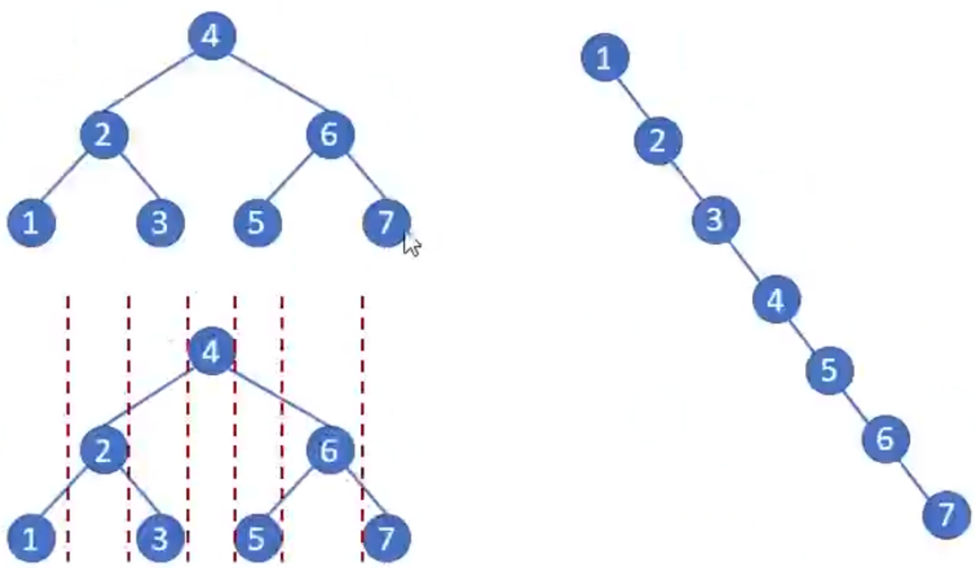
```
7
5 1
5 3
5 5
1 3
2 2
3 3
4 3
```

样例输出 #1

```
2
3
1
5
```

题目分析

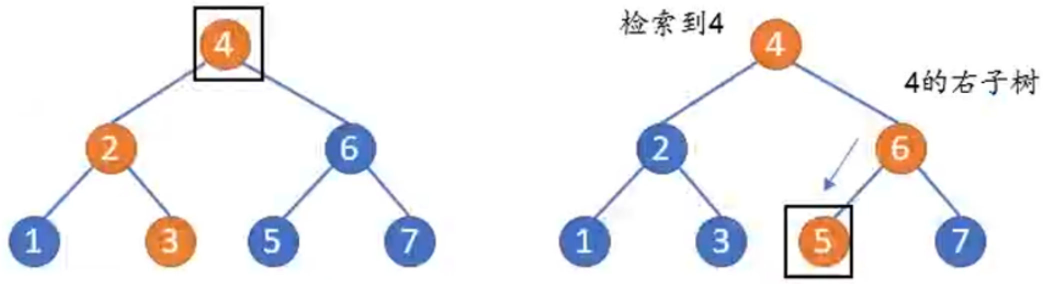
二叉搜索树BST($BinarySearchTree$，二叉搜索树）是非常有用的数据结构，它的结构精巧、访问高效。主要用来维护一组序列。数据的基本操作是插入，查询、删除。给定一个数据序列，如何实现BST?
建树和插入。以第1个数据x为根结点，逐个插入其他所有数据。插入过程从根结点，如果数据y比根结点x小，就往x的左子树上插入，否则就往右子树上插入；如果子树为空，就直接放在这个空位上，如果非空，就与子树的值进行比较，再进入子树的下一层，直到找到一个空位置。**新插入的数据肯定位于一个最底层的叶子结点，而不是插到中间某个结点上替代原来的数据。**从建树的过程可知，如果有一组数据是$(4, 2, 1, 3, 6, 5, 7)$可以得到一棵平衡二叉搜索树。访问的时间复杂度为$O(log_2n)$。如果有一组数据为$(1, 2, 3, 4, 5, 6, 7)$，按顺序插入，会全部插入到右子树上，BST退化成一只包含右子树的链表。其访问的时间复杂度为$O(n)$。



平衡的二叉搜索树（BST）和退化的BST

查询。建树过程实际上也是一个查询过程。所以查询仍然是从根结点开始的**递归**过程。访问的复杂度取决于BST的形态。以下以查询后驱为例。查询一般有三种可能结果。设ans为val的最小后驱,val为所求键值。

1. 没有找到val。根据val与根结点的关系搜索左/右子树，此时val的后继就在已经经过的结点中，ans即为所求(对应图的键值4)。
2. 找到了键值为val的结点p，但p没有右子树。与上一种情况相同。
3. 找到了键值为val的结点p，且p有右子树。从p的右子结点出发，一直向左走，就找到了val的后继(对应图的键值5)。



求3的后继　　　　求4的后继

```
// 查询：默认二叉树初始化有两个点 a[1]=-INF,a[2]=INF
int GetNext(int val)
{
    int ans=2; // a[2].val=INF
    int p=root;
    while(p)
    {
        if (val==a[p].val)
        {
            // 查询成功
            if (a[p].right>0)
            {
                // 有右子树
                p=a[p].r;
                // 右子树一直向左走
                while(a[p].left>0) p=a[p].left;
                ans=p;
             }
            break;
        }
        // 每经过一个结点，都尝试更新后继
        if (a[p].val>val  && a[p].val<a[ans].val) ans=p;
        p=val<a[p].val?a[p].l:a[p].right;
    }
    return ans;
 }
```

参考代码

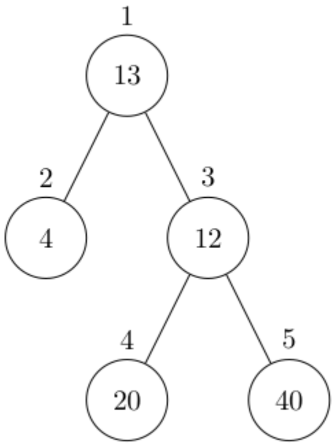
```cpp
#include<iostream>
#include<cstdio>
#include<vector>
#define pb push_back
const int N = 10010;
const int INF = 0x7fffffff;
inline int read() {
    int r = 0; bool w = 0; char ch = getchar();
    while(ch < '0' || ch > '9') w = ch == '-' ? 1 : w, ch = getchar();
    while(ch >= '0' && ch <= '9') r = (r << 3) + (r << 1) + (ch ^ 48), ch = getchar();
    return w ? ~r + 1 : r;
}
#define ls tree[x].son[0]
#define rs tree[x].son[1]
struct Node {
    int val, siz, cnt, son[2];
}tree[N];
int n, root, tot;
inline void add(int v) {
    if(!tot) {
        root = ++tot;
        tree[tot].cnt = tree[tot].siz = 1;
        tree[tot].son[0] = tree[tot].son[1] = 0;
        tree[tot].val = v;
        return ;
    }
    int x = root, last = 0;
    do {
        ++tree[x].siz;
        if(tree[x].val == v) {
            ++tree[x].cnt;
            break;
        }
        last = x;
        x = tree[last].son[v > tree[last].val];
        if(!x) {
            tree[last].son[v > tree[last].val] = ++tot;
            tree[tot].son[0] = tree[tot].son[1] = 0;
            tree[tot].val = v;
            tree[tot].cnt = tree[tot].siz = 1;
            break;
        }
    } while(true);//Code by do_while_true qwq
}
int queryfr(int val) {
    int x = root, ans = -INF;
    do {
        if(x == 0) return ans;
        if(tree[x].val >= val) {
            if(ls == 0) return ans;
            x = ls;
        }
```

```
        else {
            if(rs == 0) return tree[x].val;
            ans = tree[x].val;
            x = rs;
        }
    } while(true);
}
int queryne(int v) {
    int x = root, ans = INF;
    do {
        if(x == 0) return ans;
        if(tree[x].val <= v) {
            if(rs == 0) return ans;
            x = rs;
        }
        else {
            if(ls == 0) return tree[x].val;
            ans = tree[x].val;
            x = ls;
        }
    } while(true);
}
int queryrk(int rk) {
    int x = root;
    do {
        if(x == 0) return INF;
        if(tree[ls].siz >= rk) x = ls;
        else if(tree[ls].siz + tree[x].cnt >= rk) return tree[x].val;
        else rk -= tree[ls].siz + tree[x].cnt, x = rs;
    } while(true);
}
int queryval(int v) {
    int x = root, ans = 0;
    do {
        if(x == 0) return ans;
        if(tree[x].val == v) return ans + tree[ls].siz;
        else if(tree[x].val > v) x = ls;
        else ans += tree[ls].siz + tree[x].cnt, x = rs;
    } while(true);
}
int main() {
    n = read();
    while(n--) {
        int opt = read(), x = read();
        if(opt == 1) printf("%d\n", queryval(x) + 1);
        if(opt == 2) printf("%d\n", queryrk(x));
        if(opt == 3) printf("%d\n", queryfr(x));
        if(opt == 4) printf("%d\n", queryne(x));
        if(opt == 5) add(x);
    }
    return 0;
}
```

医院设置

题目描述

设有一棵二叉树，如图：



其中，圈中的数字表示结点中居民的人口。圈边上数字表示结点编号，现在要求在某个结点上建立一个医院，使所有居民所走的路程之和为最小，同时约定，相邻接点之间的距离为 1。如上图中，若医院建在 1 处，则距离和 $=4+12+2\times20+2\times40=136$；若医院建在 3 处，则距离和 $=4\times2+13+20+40=81$。

输入格式

第一行一个整数 n，表示树的结点数。

接下来的 n 行每行描述了一个结点的状况，包含三个整数 w, u, v，其中 w 为居民人口数，u 为左链接（为 0 表示无链接），v 为右链接（为 0 表示无链接）。

输出格式

一个整数，表示最小距离和。

样例 #1

样例输入 #1

```
5
13 2 3
4 0 0
12 4 5
20 0 0
40 0 0
```

样例输出 #1

```
81
```

提示

数据规模与约定

对于 100% 的数据，保证 $1 \leq n \leq 100$，$0 \leq u,v \leq n$，$1 \leq w \leq 10^5$。

题目分析

从指定结点开始，使用深度优先搜索。对于某个结点来说，搜索的深度就是源点到这个点的距离，单点贡献（该点所有居民到医院的距离之和）就是源点到这个结点的距离$\times$该点的居民数量。然后再加上自己的父结点和左右子结点的贡献，并返回统计的和。这里vis数组保证每个结点只统计一次。

算法模拟：假设医院设置在结点2，那么从结点2出发进行遍历。

· 遍历左结点(空)，右结点(空)，父结点1，每个结点当前距离$+1\times$*居民数，累加；

· 遍历结点1，左结点2(访问过)，父结点(空)，右结点3，每个结点当前距离$+1\times$居民数，累加；

·遍历结点3，左结点4，右结点5，父结点1（访问过)。每个结点加上当前距离$+1\times$居民数，累加；

搜索返回一个距离ans。最后对每个结点的距离进行取最小值即可。

并查集

并查集是一种非常精巧而且实用的数据结构，它主要用于处理一些不相交**集合的合并问题**。经典的例子有：判断连通图是否有环，最小生成树$Kruskal$算法,，最近公共祖先(LCA)等。

作用

在一个城市中有n个人，他们分成不同的帮派；给出一些人的关系，例如1号、2号是朋友，1号、3号也是朋友，那么他们都属于一个帮派；在分析完所有的朋友关系之后，问有多少个帮派，每人属于哪个帮派，给出的n可能是10^6的。使用并查集可以让其复杂度到达$O(log_2n)$。

亲戚

题目背景

若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

题目描述

规定：x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x，y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。

输入格式

第一行：三个整数 n, m, p，（$n, m, p \le 5000$），分别表示有 n 个人，m 个亲戚**关系**，询问 p 对亲戚关系。

以下 m 行：每行两个数 M_i，M_j，$1 \le M_i$，$M_j \le N$，表示 M_i 和 M_j 具有亲戚关系。

接下来 p 行：每行两个数 P_i, P_j，询问 P_i 和 P_j 是否具有亲戚关系。

输出格式

p 行，每行一个 `Yes` 或 `No`。表示第 i 个询问的答案为"具有"或"不具有"亲戚关系。

样例 #1

样例输入 #1

```
6 5 3
1 2
1 5
3 4
5 2
1 3
1 4
2 3
5 6
```

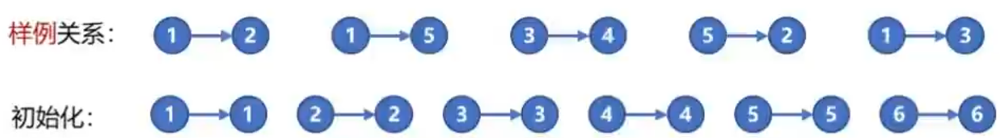
样例输出 #1

```
Yes
Yes
No
```

题目分析

并查集的主要操作有：初始化、合并与查找

1. 初始化：默认当前结点（下标）的父结点是它自己（即初始化自己就是一个集合）



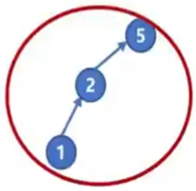
i	1	2	3	4	5	6
F[i]	1	2	3	4	5	6

2. 合并：

把结点1合并到结点2。1的父结点更改为2。根也修改为2

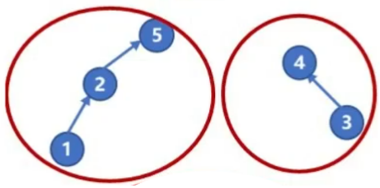
i	1	2	3	4	5	6
F[i]	2	2	3	4	5	6

将结点1的根，结点2与结点5合并，根修改为5



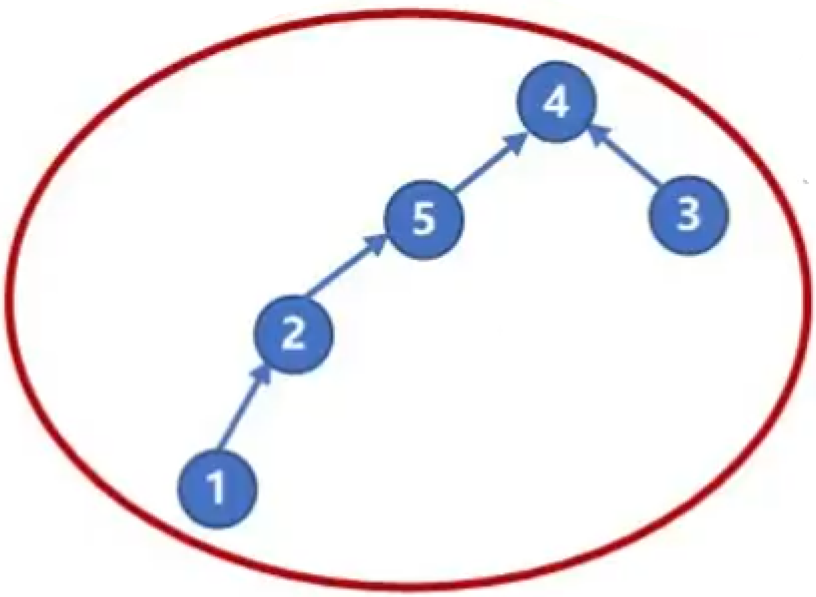
i	1	2	3	4	5	6
F[i]	2	5	3	4	5	6

把结点3合并到结点4，3的父结点更改为4，根节点修改为4

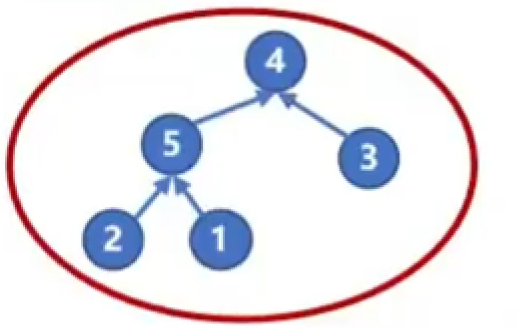


i	1	2	3	4	5	6
F[i]	5	5	4	4	4	6

将两个集合进行合并，5的父结点更改为4，根修改为4



将结点1的根改为5，结点3的根改为4，进行合并



3. 查找

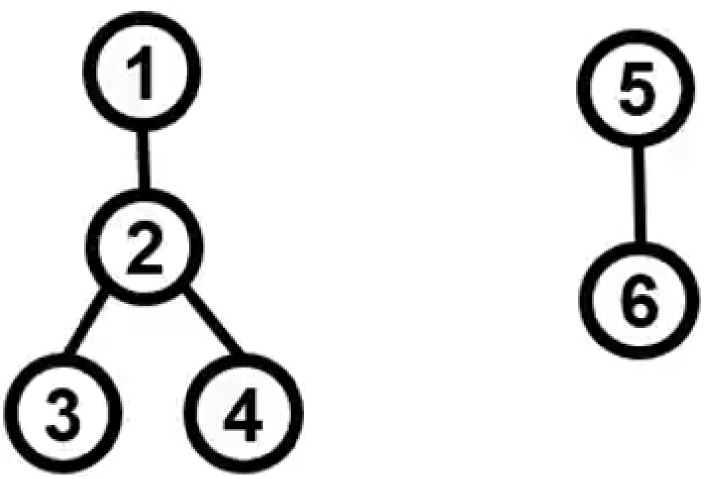
查找：

在上面步骤中已经有查找操作。查找元素的集合(根)是一个递归过程，直到结点和它的父结点相等就找到了根结点。

(注意：最后一次合并采用父结点的合并方式(而不是直接把结点1连接到结点3），左边集合的结点1的父结点需要递归修改成根结点5)，这种合并方式称为**路径压缩**，防止并查集在合并的时候退化成链状。
初始化。每个元素都是独立的一个集合，因此需要令所有$father[i]$等于i

```
for (int i=1;i<=n;i++)
{
    fa[i]=i;
}
```

查找。由于规定同一个集合中只存在一个根结点，因此查找操作就是对给定的结点**寻找其根结点**的过程。实现的方式可以是递推或是递归，但是其思路都是一样的，即反复寻找父亲结点，直到找到根结点（即$father[i] == i$的结点)




```
/*
要查找元素4的根节点是谁，应按照递推方法，流程如下：
x=4,father[4]=2,因此 4!=father[4],继续查
x=2,father[2]=1,因此 2!=fahter[2],继续查
x=1,father[1]=1,因此 1==father[1],找到根结点，返回1
*/
```

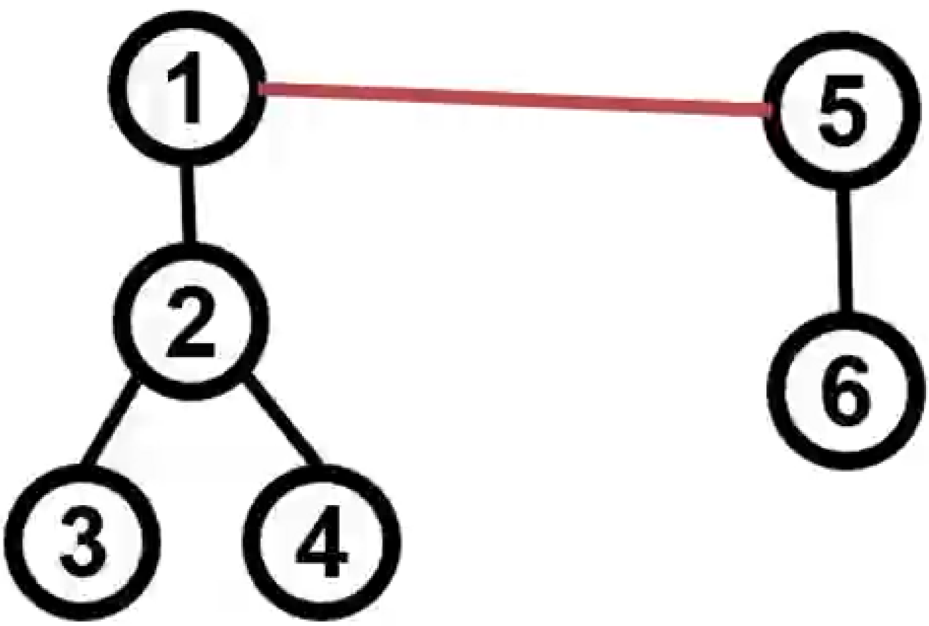
```
//递推写法
int Find(int x)
{
    while(x!=fa[x])
    {
        x=fa[x];
    }
    return x;
}

// 递归写法
int Find(int x)
{
    if(x==fa[x]) return x;
    return Find(fa[x]);
}
```

这个查找函数是没有经过优化的。在极端情况下效率极低。如果总共有10^5个结点合并成一条链，那么这个查找函数每次查询最后的根节点都需要10^5 的计算量，这显然无法承受。

路径压缩，在递归过程中，整个搜索路径上的元素(从元素i到根结点的所有元素)所属集都被改为根结点。路径压缩不仅优化了下次查询，而且优化了合并，因为合并时也用到了查询。

```
// 寻找根节点的路径压缩（递归写法）
int Find(int x)
{
    if (x==fa[x]) return x;
    return fa[x]=Find(fa[x]);
    // 递归过程查找根节点，同时把根节点赋值给fa[x]
}
```

合并。合并是指两个集合合并成一个集合，题目中一般给出两个元素，要求把这两个元素所在的集合合并。具体实现上一般是**判断**两个元素是否属于同一个集合，只有当两个元素属于不同集合时才合并，而合并的过程一般是把其中一个集合的**根结点的父亲**指向另一个集合的根结点。



```
void Union(int a,int b)
{
    int faA=Find(a); // 查找a的根节点，记为faA
    int faB=Find(b); // 查找b的根节点，记为faB
    if (faA!=faB)
    {
        Find(faA)=faB; // 合并
    }
}
```

并查集性质

在合并过程中，只对两个下同的集合进行合并，如果两个元素在相同的集合中，那么就不会对它们进行操作。这就保证了在同一个集合中一定不会产生环，即**并查集产生的每一个集合都是一棵树。**

代码参考

```
#include<bits/stdc++.h>
using namespace std;
const int N=5010;
int n,m,t;
int f[N]; // 点与父结点
int Find(int x)
{
    // 查找与修改当前点的父结点为根节点
    if (x!=f[x]) return f[x]=Find(f[x]);// 如果当前父结点不是根结点，则递归查找
    return x; // 返回根节点
 }

 int main()
 {
    cin>>n>>m>>t;
    for (int i=1;i<=n;i++) f[i]=i ; // 初始化父结点为自己
    int x,y; // 表示关系链
    for (int i=1;i<=m;i++)
    {
        cin>>x>>y; // 输入关系 1 2
        f[Find(x)]=Find(y); // 查询并合并两点，将左图中的根节点5合并到右图结点5（原来父结点是自己）的父结点修改为4
        // for (int i=1;i<=n;i++) cout<<f[i]<<" " ;可以查看过程
     }
    for (int i=1;i<=t;i++)
    {
        cin>>x>>y; //检查两者是否有关系，即是否在一个集合
        f[x]=Find(x),f[y]=Find(y); // 查找两点的根节点
        if (f[x]==f[y] )cout<<"Yes"<<endl; // 如果根节点一样，则在同一个集合
        else cout<<"No"<<endl;
     }
    return 0;
  }
```