基础树图

- 2016信息安全邓楚盟
- 微信：Dcm666888
- 给大家安利一波Marp

目录

树

- 什么是树？树的种类、树的存储、树的遍历、树的性质、树的维护

图

- 什么是图？图的种类、图的存储、图的遍历、图的应用、图的性质和定理

并查集

- 简单介绍、带权并查集

1. 树

什么是树

一种重要的非线性数据结构，其中以二叉树最为常用。树结构在客观世界中广泛存在，如社会族谱和组织机构，计算机世界中如语法结构，数据关系。

名词术语

- 边：结点与结点之间是通过一条有向的边(edge)所连接的，一颗结点为n的树有n-1条边，因为除去根结点没有边，其余的结点都有一条边；
- 结点的度:结点的子树个数
- 树的度:树的所有结点中最大的度数
- 叶结点(leaf)： 度为0的结点；
- 路径和路径长度:从结点n_1到结点n_k的路径为一个结点序列$n_1, n_2, ..., n_k$。路径所包含的边的个数为路径的长度
- 祖先结点(ancestor)： 沿树根到某一结点路径上的所有结点都是这个结点的祖先结点；
- 子孙结点(ancestor)：某一结点的子树中的所有结点是这个结点的子孙；
- 树的深度(depth)：树中结点的最大层次称为树的深度(depth)或高度

名词术语

- 二叉树：每个节点最多含有两个子树的树称为二叉树；
- 满二叉树：如果一棵二叉树的结点要么是叶子结点，要么它有两个孩子结点，这样的树就是满二叉树
- 完全二叉树：完全二叉树：叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树
- **树的重心**：树的重心也叫树的质心。找到一个点,其所有的子树中最大的子树节点数最少,那么这个点就是这棵树的重心,删去重心后，生成的多棵树尽可能平衡。
- **树的直径：树中所有最短路径的最大值。**
- **LCA**：最近公共祖先

树的直径

- 第一次BFS/DFS我们可以从任意顶点出发，不妨设该顶点为 t, 找到距离 t 最远的顶点u。则u一定为树的直径中的某一个顶点
- 第二次BFS/DFS我们需要从顶点u出发，这次找到距离u最远的顶点v，则u到v的最短路径就是树的直径。

树的存储

- `father`数组记录父亲
- 孩子数组
  - 动态开点(省空间）/固定点（使用二叉树，简单）
- 当成图一样存（之后讲）

动态开点

一般对空间有较高要求，如可持久化的线段树，动态开点的线段树，trie树，ac自动机等,以后学一些高阶的数据结构会用到，原理类似于指针，只不过是用标号寻找节点.

```
struct Node{
  int child[MAX_CHILD];//每个节点最多的孩子数
  int val;
}node[maxn];
int ct = 0;
void insert(int &o,int v)
{
  if (!o)
  {
    o = ++ct;
    node[o].val = v;
    for(int i = 0;i<MAX_CHILD;i++)node[o].child[i]=0;
    return;
  }
  ......
}
```

固定点

适用二叉树，个人推荐。每个点只有左孩子，右孩子,从1开始。基础线段树都是这么存点的。

```
#define Lson(x) (x<<1)
#define Rson(x) ((x<<1)|1)
#define Father(x) (x>>1)
int val[maxn];
val[x] = 5;
val[Lson(x)] = 6;
val[Rson(x)] = 7;
int lson = Lson(x);
cout <<Father(lson) << endl;
```

当成图一样存

适用绝大多数树的题，个人推荐，也是最常见的树的存储方法，需要手动给一个根。

- 邻接矩阵
- vector
- 链式前向星
- 具体操作等下讲图的时候讲

树的遍历

- DFS
- BFS
- 先序/中序/后序遍历

二叉搜索树

二叉搜索树是一种二叉树的树形数据结构，其定义如下

1. 空树是二叉搜索树。
2. 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值。
3. 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值。
4. 二叉搜索树的左右子树均为二叉搜索树。

二叉搜索树--复杂度

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比。对于一个有n个结点的二叉搜索树中，这些操作的最优时间复杂度为$O(log_2n)$，最坏为$O(n)$ 。随机构造这样一棵二叉搜索树的期望高度为$O(log_2n)$。

(这里埋一个伏笔 , 二叉搜索树为了尽可能降低各种操作的时间复杂度，会用到旋转，使它高度尽可能小，改变树的结构，同时依然满足左小右大的性质。如果以后有幸能讲splay树，我们再学习）

二叉搜索树--常见操作

- 插入
- 删除
- 求排名
- 排名为k的元素

插入

定义 `insert(o,v)` 为在以o为根节点的二叉搜索树中插入一个值为的新节点。
分类讨论如下：

若o为空，直接返回一个值为v的新节点。
若o的权值大于v，在o的左子树中插入权值为v的节点。
若o的权值等于v，该节点的附加域该值出现的次数自增1。
若o的权值小于v，在o的右子树中插入权值为v的节点。

```
void insert(int &o, int v) {
  if (!o)
  {
      o = ++ct;
      val[o] = v;
      siz[o] = cnt[o] = 1;
      return;
  }
  siz[o]++;
  if (val[o] > v) insert(lc[o], v);
  if (val[o] == v) {
    cnt[o]++;
    return;
  }
  if (val[o] < v) insert(rc[o], v);
}
```

删除

定义 `delete(o,v)` 为在以o为根节点的二叉搜索树中删除一个值为v的节点。
先在二叉搜索树中找到权值为v的节点，分类讨论如下：

若该节点的附加cnt为1：

1. 若o为叶子节点，直接删除该节点即可。
2. 若o为链节点，即只有一个儿子的节点，返回这个儿子。
3. 若o有两个非空子节点，一般是用它右子树的最小值代替它，然后将它删除。

```cpp
int deletemin(int &o) {
  if (!lc[o]){
    int ret = o;o = rc[o]; return ret;
  }
  else
    return deletemin(lc[o]);
}

void delte (int& o, int v) {
  siz[o]--;
  if (val[o] == v ) {
    if(cnt[o] > 1){cnt[o]--;return;}
    if (lc[o] && rc[o]){
        int t = deletemin(rc[o]);rc[t] = rc[o];lc[t] = lc[o];
        o = t;
    }
    else
      o = lc[o] + rc[o];
    return;
  }
  if (val[o] > v) delte (lc[o], v);
  if (val[o] < v) delte (rc[o], v);
}
```

求元素的排名

排名定义为将数组元素排序后第一个相同元素之前的数的个数$+1$

维护每个根节点的子树大小siz。查找一个元素的排名，首先从根节点跳到这个元素，若向右跳，答案加上左儿子节点个数加当前节点重复的数个数，最后答案加上终点的左儿子子树大小$+1$。

```
int qrnk(int o, int v) {
  if (val[o] == v) return siz[lc[o]] + 1;
  if (val[o] > v) return qrnk(lc[o], v);
  if (val[o] < v) return qrnk(rc[o], v) + siz[lc[o]] + cnt[o];
}
```

查找排名为 k 的元素

在一棵子树中，根节点的排名取决于其左子树的大小。

若其左子树的大小大于等于k，则该元素在左子树中

若其左子树的大小在区间$[k-1,k+cnt-1]$中，则该元素为子树的根节点；

若其左子树的大小小于$[k+cnt-1]$，则该元素在右子树中。

```
int querykth(int o, int k) {
  if (siz[lc[o]] >= k) return querykth(lc[o], k);
  if (siz[lc[o]] < k + cnt[o] - 1)
    return querykth(rc[o], k - siz[lc[o]] - cnt[o] + 1);
  return o;
}
```

然而，上面讲的都是基础，单独考平衡二叉树的题比较少（面试题除外）

这一切都是在为一个叫做Splay树的东西做准备

然而，Splay树今天不讲。

抛砖引玉

acm中的数据结构也是充满了未知和挑战，希望大家能在其中找到属于自己的乐趣。

其他以后会学/听到的其他名词

- 树状数组：十分十分优雅的数据结构，常用来解决区间问题。
- 线段树：近些年的常见考点，也可以说是必备技能了。
- ~~划分树（用来解决区间第K大/前K大）（常数贼大）（不会，不讲）~~
- 虚树：浓缩一个树，保留所有关键点以及任意一对关键点的公共祖先。
- Treap，弱平衡的二叉搜索树，treap 的每个结点上要额外储存一个值**priority**。treap 除了要满足二叉搜索树的性质之外，还需满足父节点**priority**大于等于两个儿子的。而 是每个结点建立时随机生成的，因此 treap 是期望平衡的。（旋转和无旋两种）
- ~~树套树：就是树里有树。（不会，不讲）~~
- **Splay树**：通过旋转，使其始终满足左儿子的值<根节点的值<右儿子的值,可以用来解决很多区间操作的问题，以前比较火，现在见得少了，splay能做的，线段树基本都能做，线段树常数还小。

其他以后会学/听到的其他名词

- AVL树：是一种平衡的二叉搜索树，在每次插入后，会根据平衡因子决定是否需要旋转
- kd树：以树的形式存储n维空间各个点的信息，可以快速查找距离某个点最近的k个点
- 替罪羊树：是一种依靠重构操作维持平衡的重量平衡树。替罪羊树会在插入、删除操作时，检测途经的节点，若发现失衡，则将以该节点为根的子树重构（简单说就是把树拍成一个链，然后在提起来）。有时候会和kd树一起使用。
- 树链剖分：把一棵树拆成几个链，以线段的形式维护。
- **Link-Cut Tree**： LCT牛逼！Tarjan牛逼！Splay+树链剖分，解决一类动态树问题，比如把一棵树的某条边割掉，接到另一颗树上，还要快速维护一些信息。
- **树分治**：又叫点分治，强烈建议学一下。

其他以后会学/听到的其他名词

- 最小生成树：这个下周就会讲
- 基环树 ： n个点n条边
- 可持久化数据结构：常用来解决有保存历史信息需求的数据结构，可以随时回滚、查询任何一个操作之后的状态。
- 仙人掌 ：是**不含自环**的,一条边最多属于一个简单环的无向连通图.

2. 图

基础概念

- 图
  - 节点
  - 边
  - 路径/回路
  - 环
  - 链
  - 度数
  - 补图
  - 子图/导出子图
- 简单图/多重图
  - 重边
  - 自环
- 有向图/无向图
  - 有向边/无向边
  - 混合图
- 完全图
  - 团
  - 有向完全图
  - 竞赛图
- 连通性
  - 连通图/连通分量
  - 双联通图/双连通分量
  - 强连通图/强连通分量
  - 割点/桥
  - 割
- 网络流
  - 最大流/最小割
  - 最小割树
- 树
  - 根
  - 叶子
  - 最近公共祖先
  - 生成树
  - 有向生成树（树形图）
  - 森林
  - DFS树/森林
  - 仙人掌/沙漠
- 二分图
  - 染色
  - 匹配
  - 独立集
- 平面图
  - 对偶图
- 弦图
  - 弦
  - 区间图
- （k-）正则图
- 线图
- 完美图
  - 色数
- ……

什么是图？

一个图G是一个二元组，即序偶$< V, E >$，或记作$G =< V, E >$，其中V是有限非空集合，称为G的顶点集，V中的元素称为顶点或结点；E称为G的边的集合$\forall e_i \in E$，都有V中的结点与之对应，称e_i为G的边。

简单来说，图就是节点集合和边集合。

名词术语

- 有向边和无向边/有向图和无向图
- 有限图：一个图的点集和边集都是有穷集的图。
- 平凡图：仅有一个结点而没有边构成的图。
- 关联：若有$e_i = (u, v)$且$e_i \in E$，则称u是和v相关联的。
- 自环：若一条边所关联的两个结点重合，则称此边为自环。
- 邻接：关联于同一条边的两个点u和v称为邻接的；关联于同一个点的两条边e_1和e_2是邻接的（或相邻的）。
- **度数**：关联于结点v的边的条数叫度数，对于有向图，又有出度和入度两种
- 连通图：图上任意两点可以互相到达
- 树：边数比结点少一的连通图

名词术语

- 简单图：不含重边和自环的图。
- 完全图：任意两点之间都有边的简单无向图
- 图论的知识点和算法比较多，可以参考Oi-wiki-图论

如何存图？

- 存边
- 邻接矩阵
- 邻接表
- 链式前向星

树本质上也是一种图
会存图、遍历图了，自然树也就会了

存边

每个边记录左右端点、权值、id，每个点记录边的标号,遍历的时候根据每个点对应的边，即可找到下一个点。

```
struct Edge{
  int u,v,w,id;
}edge[maxn];
vector<int>node[maxn];//记录每个点对应的边
int ct = 0;//记录当前有多少边
void addedge(int u,int v,int w){
  edge[ct].u = u;
  edge[ct].v = v;
  edge[ct].w = w;
  edge[ct].id = id;
  node[u].push_back(ct);
  ct++;
}
```

邻接矩阵

说白了就是二维数组，`mp[i][j]`表示点i和j之间边的权值或有无。

```
int mp[maxn][maxn]
void addedge(int u,int v,int w)
{
  mp[u][v] = w;//视情况而定
}
```

邻接表

每个点记录与其连接的其他点，可以用vector、list等,如果需要权值，那还是需要存边。

```
vector<int>V[maxn];//每个点的链接情况
void addedge(int u,int v)
{
  V[u].push_back(v);
}

//遍历
for(int i = 0;i<V[fron].size();i++){
  int to = V[from][i];
  ......
}
```

链式前向星！

非常巧妙且推荐的存图方式，编写简介，效率够高

```
int cnt = 0;
int head[maxn];
struct Edge{
  int to,w,next;
}edge[maxn];

void init(){
  cnt=0;
  for(int i = 0;i<maxn;i++)head[i]=-1;
}
void addedge(int u,int v,int w){
  edge[cnt].w=w;
  edge[cnt].to=v;
  edge[cnt].next=head[u];
  head[u]=cnt++;
}

//遍历的时候
for(int i = head[u];i!=-1;i=edge[i].next){
  int to = edge[i].to;
  int weight = edge[i].w;
  ......
}
```

- 图的知识点也不少，今天只是入个门，之后咱们慢慢讲

某段神秘代码

```
struct node
{
    int v,next,w;
}edge[maxn*2];
int head[maxn],tot;
int size[maxn];//树的大小
int maxv[maxn];//最大孩子节点的size
int vis[maxn];
int dis[maxn];
int num;
void init()
{
    tot=0;
    ans=0;
    memset(head,-1,sizeof(head));
    memset(vis,0,sizeof(vis));
}
void add_edge(int u,int v,int w)
{
    edge[tot].v=v;
    edge[tot].w=w;
    edge[tot].next=head[u];
    head[u]=tot++;
}
```

```
//处理子树的大小
void dfssize(int u,int f)
{
    size[u]=1;
    maxv[u]=0;
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].v;
        if(v==f||vis[v])continue;
        dfssize(v,u);
        size[u]+=size[v];
        if(size[v]>maxv[u])maxv[u]=size[v];
    }
}
```

```
//找重心
void dfsroot(int r,int u,int f)
{
    if(size[r]-size[u]>maxv[u])maxv[u]=size[r]-size[u];
    //size[r]-size[u]是u上面部分的树的尺寸,
    //跟u的最大孩子比,找到最大孩子的最小差值节点
    if(maxv[u]<Max)Max=maxv[u],root=u;
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].v;
        if(v==f||vis[v])continue;
        dfsroot(r,v,u);
    }
}
```

```cpp
//求每个点离重心的距离
void dfsdis(int u,int d,int f)
{
    dis[num++]=d;
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].v;
        if(v!=f&&!vis[v])
            dfsdis(v,d+edge[i].w,u);
    }
}
```

如果上面的步骤你都懂了，那么恭喜你，树分治你已经会了一半了！虽然今天不讲，但是还是要抛砖引玉。
洛谷日报#23

并查集！

一种超级优雅的数据结构

畅通工程

> 某省调查城镇交通状况，得到现有城镇道路统计表，表中列出了每条道路直接连通的城镇。省政府“畅通工程”的目标是使全省任何两个城镇间都可以实现交通（但不一定有直接的道路相连，只要互相间接通过道路可达即可）。问最少还需要建设多少条道路？

输入

> 测试输入包含若干测试用例。每个测试用例的第1行给出两个正整数，分别是城镇数目N (< 1000)和道路数目M；随后的M行对应M条道路，每行给出一对正整数，分别是该条道路直接连通的两个城镇的编号。为简单起见，城镇从1到N编号。

输出

> 对每个测试用例，在1行里输出最少还需要建设的道路数目。

分析

在各个连通块之间加路就行了（连通块可以理解为一个集合），每个连通块内部，两两可达，m个连通块，则需要m-1条路

如何求连通块?DFS?BFS?没那么麻烦，并查集登场!

为什么说它优雅？

```
int fa[maxn];
void init(int n)
{
    for(int i = 0;i<n;i++)fa[i]=i;
}
int find(int x)
{
    return (fa[x] == x)?x:fa[x] = find(fa[x]);
}
void merge(int x,int y)
{
    int xx = find(x),yy = find(y);
    if(xx != yy) fa[xx] = yy;
}
```

```
int find(int x)
{
    if(fa[x] == x)return x;
    int f = find(fa[x]);
    fa[x] = f;
    return fa[x];
}
```

解释

- `fa[maxn]`数组记录每个节点的father，初始每个人都是自己的（也可以认为一开始每个人都在自己集合里）。这个father由于存在路径压缩，会破坏原来图的结构，如果需要
- 路径压缩就是，在向上找父亲的过程中，把所有遇到的点的父亲都变成目前这个祖先，画图理解会比较容易一些
- `merge(int x,int y)`就是给两个点的祖先建立父子关系，相当于把两个集合并成一个集合了

然后我们就知道每个城镇属于哪个集合，就能统计出一共有多少个集合，这个问题就能解决了！
传闻，并查集的时间复杂度非常小。

带权并查集

有时候我们除了需要知道自己所属的集合，还需要知道自己和父亲的关系，就需要带权并查集了，然而，路径压缩的时候会改变权值关系，一定要注意维护!

食物链

> 动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B， B吃C，C吃A。
> 现有N个动物，以1－N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。
> 有人用两种说法对这N个动物所构成的食物链关系进行描述：
> 第一种说法是"1 X Y"，表示X和Y是同类。
> 第二种说法是"2 X Y"，表示X吃Y。
> 此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
> 1） 当前的话与前面的某些真的话冲突，就是假话；
> 2） 当前的话中X或Y比N大，就是假话；
> 3） 当前的话表示X吃X，就是假话。
> 你的任务是根据给定的N（1 <= N <= 50,000）和K句话（0 <= K <= 100,000），输出假话的总数。

输入

> 第一行是两个整数N和K，以一个空格分隔。
> 以下K行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中D表示说法的种类。
> 若D=1，则表示X和Y是同类。
> 若D=2，则表示X吃Y。

输出

> 只有一个整数，表示假话的数目。

分析

- 三种关系，同类，吃，被吃
- `val[x]`表示x和father的关系，0表示同类，1表示吃父亲，2表示被吃
- 路径压缩改变father的时候怎么维护权值？
  - `val[fx] = (val[y]-val[x]+t+3)%3`
  - 举个例子，如果y吃父亲，x被父亲吃，x又吃y，那么y的父亲和x是同类，`val[fx]=(1-2+1)%3=0`
  - 向上传递是加模
  - merge之前先判断貌不矛盾
  -

End，谢谢大家