图论基础

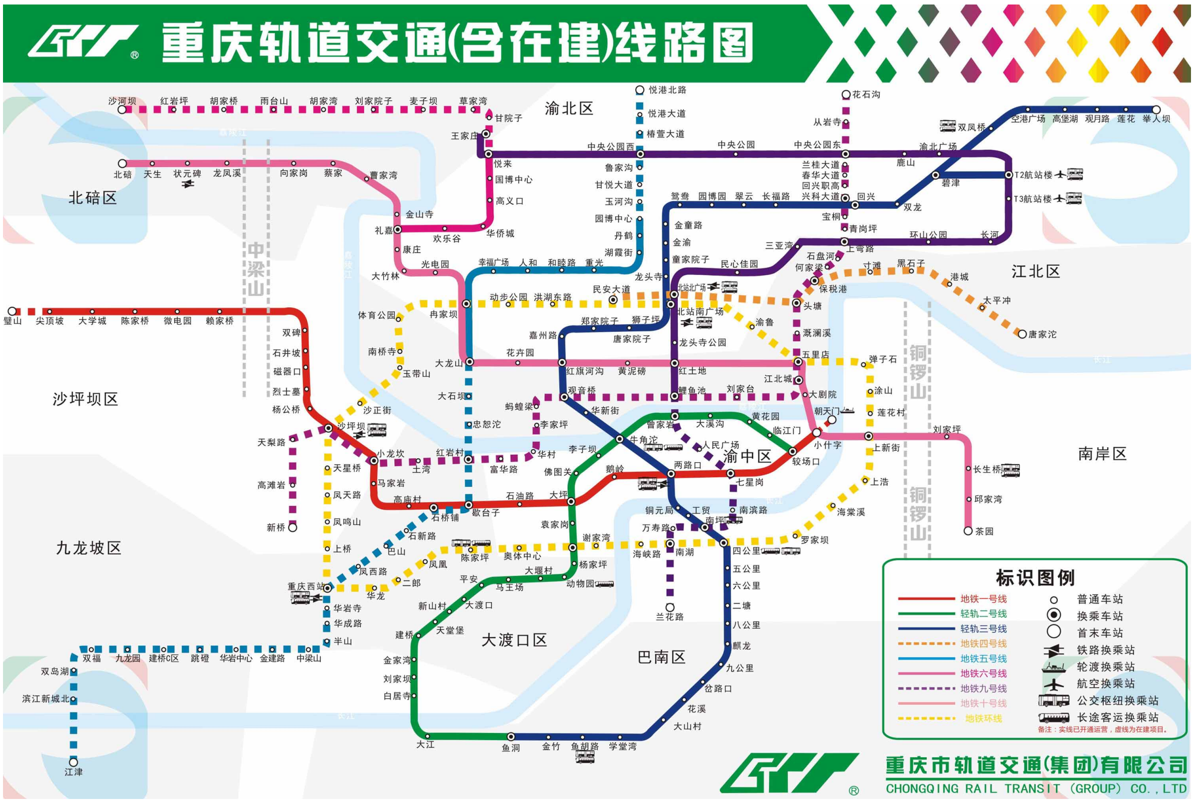
图 (Graph) 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

定义

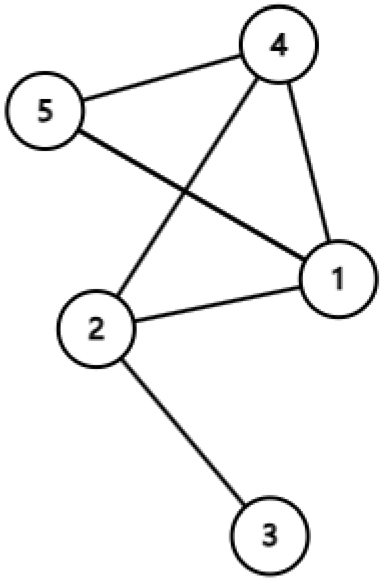
图是一个二元组$G=(V(G),E(G))$，其中$V(G)$非空

通俗来讲，图就是由一个非空的**顶点集合**V和一个**边集合**E（其中，一条边一般链接**两个**顶点）组合而成的集合

下图即为重庆轨道交通线路图，它也可以称为一张图



在信息竞赛中，我们一般将其简化，利用数字作为点的编号，顶点之间连线作为图的边

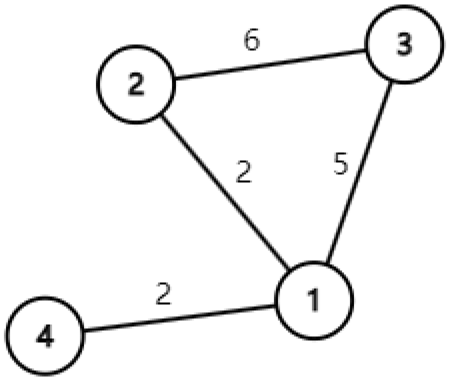


上图就是由点集$\{1,2,3,4,5\}$和边集$\{(1,2),(1,4),(1,5),(2,3),(2,4),(4,5)\}$组成的一张图，其中，边集中的$(u,v)$代表有一条从$u$到$v$的**双向边**（也可以代表有一条从$v$到$u$的**双向边**，即$(v,u)$）

此时，每个节点连边的条数就是这个结点的**度数**，记为$d(i)$，例如，
$d(1)=3,d(2)=3,d(3)=1,d(4)=3,d(5)=2$

由于每条边对$\sum d(i)$的贡献为2，因此，我们在有m条边的图中所有节点的度数之和一定是**偶数**，且为该和为$2m$

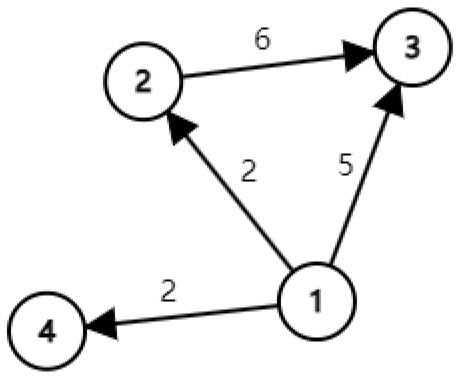
每一条边可以附带属性值，我们一般称之为**边权**，例如**通过该条边所需要的时间**，同样的，顶点也可以有权值，我们称之为**点权**



如图，此时边$(1,2)$的边权就为2，对于**带权边**，我们就可以通过三元组(u,v,w)描述，其中(u,v)定义与上文相同，而增加的w则是该边的边权，对于刚刚这条边，我们就可以以$(1,2,2)$来描述，也可以使用$(2,1,2)$来描述

如果两个顶点之间有不止一条边连接，我们此时称之为**重边**，甚至有些情况下有些边的起点和终点是一样的，我们此时称之为**自环**，在绝大部分情况下，重边和自环都会被简化掉（例如直接删除自环，重边保留权值最小的一条）

然而，有时我们的边不一定能够双向通行，就像单车道一样，如下图所示



此时，边集中描述边的三元组(u, v, w)的意义就变为了：有一条从u出发，到v的，权值为w的**单向边。** 边集为**有向**的图，我们称之为**有向图**，而边集为**无向**的，我们就称之为**无向图**

由于边是**单向的**，因此一旦u, v的顺序改变，边的实际意义也随之改变，因此对于**有向图**而言，(u, v, w_1)和(v, u, w_2)不会被判定成重边，而在无向图中，这就是重边

对于**度数**的定义而言，在有向图中，我们分为了**入度**和**出度**，顾名思义，某个点的入度，就是以这个点为**终点**的边的条数，而出度呢，就是以这个点为**起点**的边的条数

由于每条边对入度和出度的贡献都为1，因此，对于某个有m条边的图而言，其所有节点的**入度**和等于其所有节点的**出度**和等于m

参考链接

[Graph editor]

图的存储

现在，我们希望将上面两张带权的图存入计算机，一般有两种存图方式：**邻接矩阵**和**邻接表**

邻接矩阵

邻接矩阵可以使用一个二维数组$v[i][j]$来表示，$v[i][j]$表示从点i到点j的边权，对于不带权的图，我们也可以定义$v[i][j]$为点i到点j的**可行性**，即0代表(i, j)这条边不存在，1代表(i, j)这条边存在

下表即为第一张带权图的邻接矩阵，其中，0代表点(i, j)之间没有边直接连接

	1	2	3	4

	1	2	3	4
1	0	2	5	2
2	2	0	6	0
3	5	6	0	0
4	2	0	0	0

[# 25447] 邻接矩阵存储图

题目描述

给出一个无向图，顶点数为n，边数为m。

输入格式

第一行两个整数$n, m (1 \le n \le 1000, 1 \le m \le 10000)$，接下来有$m$行，每行两个整数$u, v$,表示点$u$到点$v$有一条边。

输出格式

由小到大依次输出每个点的邻接点，邻接点也按照由小到大的顺序输出。

样例输入

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

样例输出

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

Solution

使用邻接表存图，依次遍历即可

Code

```
void solve() {
    int n, m;
    cin >> n >> m;
    vector e(n + 1, vector<int>(n + 1));
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        e[u][v] = e[v][u] = 1;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
            if (e[i][j])
                cout << j << ' ';
        cout << endl;
    }
}
```

邻接表

邻接矩阵虽然直观，但对于一个n个节点，m条边的图而言，它占用的空间大小是$O(n^2)$的，而对于较为稀疏的图而言（即边的数量m远小于n），会有大量的存储空间浪费，此时我们的邻接表就应运而生了

接下来将以**有向图**解释邻接表是如何存图的

对于某个点u而言，我们在邻接表中，关注的是所有从u出发的边，即我们需要一个数组，存储(v_j, w_j)，其中v为到达的边，w_j为边权，而且数组大小等于点u的出度

因此，我们可以开一个大小等于点u出度的$\text{vector}<\text{pair}<\text{int}, \text{int}>>$，或是在读取边集的时候不断$\text{emplace_back}(\text{v,w})$即可

而对于整张图而言，我们需要开n个这样的vector，一般有两种实现方式：

```
// vector数组
vector<pair<int, int>> e[maxn];
// vector套vector，一般点集标号为1 - index，因此size要开到n + 1
vector e(n + 1, vector<pair<int, int>>());
```

对于一条边，我们只会插入一个vector里面，因此，占用空间是$O(n+m)$的

我们以**有向图**为例讲解了邻接表是如何存图的，对于**无向图**而言，其某条边(u, v, w)其实可以**拆**成两条有向边$(u, v, w), (v, u, w)$，此时存图就可以和**有向图**一样了

```
// 不带权的单向边
vector<int> e[maxn];

int n, m; cin >> n >> m;
for (int i = 0; i < m; i++) {
    int u, v; cin >> u >> v;
    e[u].push_back(v);
}
```

```cpp
// 带权的双向边
vector<pair<int, int>> e[maxn];

int n, m; cin >> n >> m;
for (int i = 0; i < m; i++) {
    int u, v, w; cin >> u >> v >> w;
    e[u].emplace_back(v, w);
    e[v].emplace_back(u, w);
}
```

对于邻接表和邻接矩阵而言，其各有优劣，例如：

1. 邻接表占用空间的大小为$O(n+m)$，邻接矩阵占用空间大小为$O(n^2)$，由于去掉重边、自环情况下，一个图最多有$\frac{n(n-1)}{2}$条边，因此邻接表占用空间不会大于邻接矩阵
2. 邻接表能快速的得到某个点的出边条数（$\mathrm{e[u].size()}$）
3. 邻接表能够快速地清空某一个点的所有边（$\mathrm{e[u].clear()}$）
4. 邻接矩阵能够以$O(1)$的时间复杂度得到某条特定边(i,j)的信息，而邻接表需要$O(m)$

[# 25448] 邻接表存储图

题目描述

给出一个无向图，顶点数为n，边数为m。

输入格式

第一行两个整数$n, m(1 \le n \le 1000, 1 \le m \le 10000)$，接下来有$m$行，每行两个整数$u, v$,表示点$u$到点$v$有一条边。

输出格式

第i行输出点i的所有邻接点，邻接点按照度数由小到大输出,如果度数相等，则按照由小到大输出。

样例输入

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

样例输出

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

Solution

使用邻接表存图，同时存储每个点的度数，在输出前排序即可

Code

```
void solve(){
    int n, m; cin >> n >> m;
    vector e(n + 1, vector<int>());
    vector<int> d(n + 1);
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v;
        e[u].push_back(v); e[v].push_back(u);
        d[u]++; d[v]++;
    }

    for (int i = 1; i <= n; i++) {
        sort(all(e[i]), [&](int a, int b){
            if (d[a] == d[b]) return a < b;
            return d[a] < d[b];
        });
        for (auto v: e[i]) cout << v << ' ';
        cout << endl;
    }
}
```

图的遍历

深度优先遍历(dfs)

假设某个点有两条边，分别连向u, v，我们先遍历u及其所有边连向的节点，在回来遍历v及其所有边连向的节点，叫做深度优先遍历，其代码一般由递归实现，如下所示

```
void dfs(int u) {
    // u已经被访问过
    if (vis[u]) return;
    xxx; vis[u] = 1;
    for (auto v: e[u]) dfs(v);
}
```

[# 3376] 有向图的DFS

题目描述

给定一个有向图，有N个顶点，M条边，顶点从$1, 2, \cdots N$依次编号，求出字典序最小的深度优先搜索顺序。

输入格式

第1行：2个整数，$N(1 \le N \le 200)$ 和$M(2 \le M \le 5000)$接下来M行，每行2个整数i, j，描述一条边从顶点i指向顶点j

输出格式

仅一行，一个顶点编号序列，表示**字典序最小**的深度优先搜索序列，顶点之间用一个空格分开

样例输入

```
3 3
1 2
1 3
2 3
```

样例输出

```
1 2 3
```

Solution

由于要求**字典序**最小，因此我们以1为起点遍历，在预处理时对所有与$1(2, 3, \cdots n)$连边的节点按照**字典序**进行排序，递归进行即可

Code

```
void solve() {
    int n, m;
    cin >> n >> m;
    vector e(n + 1, vector<int>());
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
    }

    vector<int> vis(n + 1);
    for (auto& vec : e) sort(all(vec));

    auto dfs = [&](int u, auto dfs) -> void {
        if (vis[u])
            return;
        vis[u] = 1;
        cout << u << ' ';
        for (auto v : e[u]) dfs(v, dfs);
    };

    for (int i = 1; i <= n; i++)
        if (!vis[i])
            dfs(i, dfs);
    cout << endl;
}
```

广度优先遍历(bfs)

假设某个点有两条边，分别连向u, v，我们先将u, v都遍历一遍，再去遍历他们所有边连向的节点，叫做广度优先遍历，其代码一般由迭代实现，大致如下

```
void bfs(int start) {
    // 用队列来存储待访问元素
    queue<int> q; q.push(start);
    while (!q.empty()) {
        auto u = q.front(); q.pop();
        if (vis[u]) continue;
        xxx; vis[u] = 1;
        for (auto v: e[u]) q.push(v);
    }
}
```

[# 3377] 有向图的BFS

题目描述

给定一个有向图，有N个顶点，M条边，顶点从$1, 2, \cdots N$依次编号，求出字典序最小的广度优先搜索顺序。

注意： 本题图可能不连通

输入格式

第1行：2个整数，$N(1 \le N \le 200$）和$M(2 \le M \le 5000)$接下来M行，每行2个整数i, j，描述一条边从顶点i指向顶点j

输出格式

仅一行，一个顶点编号序列，表示**字典序最小**的广度优先搜索序列，顶点之间用一个空格分开

样例输入

```
3 3
1 2
1 3
2 3
```

样例输出

```
1 2 3
```

Solution

类似有向图的DFS，因此我们以1为起点遍历，在预处理时对所有与$1(2,3,\cdots n)$连边的节点按照**字典序**进行排序，迭代进行即可

Code

```
void solve() {
    int n, m;
    cin >> n >> m;
    vector e(n + 1, vector<int>());
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
    }

    vector<int> vis(n + 1);
    for (auto& vec : e) sort(all(vec));

    for (int i = 1; i <= n; i++)
        if (!vis[i]) {
            queue<int> q;
            q.push(i);
            while (!q.empty()) {
                auto u = q.front();
                q.pop();
                if (vis[u])
                    continue;
                vis[u] = 1;
                cout << u << ' ';
                for (auto v : e[u]) q.push(v);
            }
        }
    cout << endl;
}
```

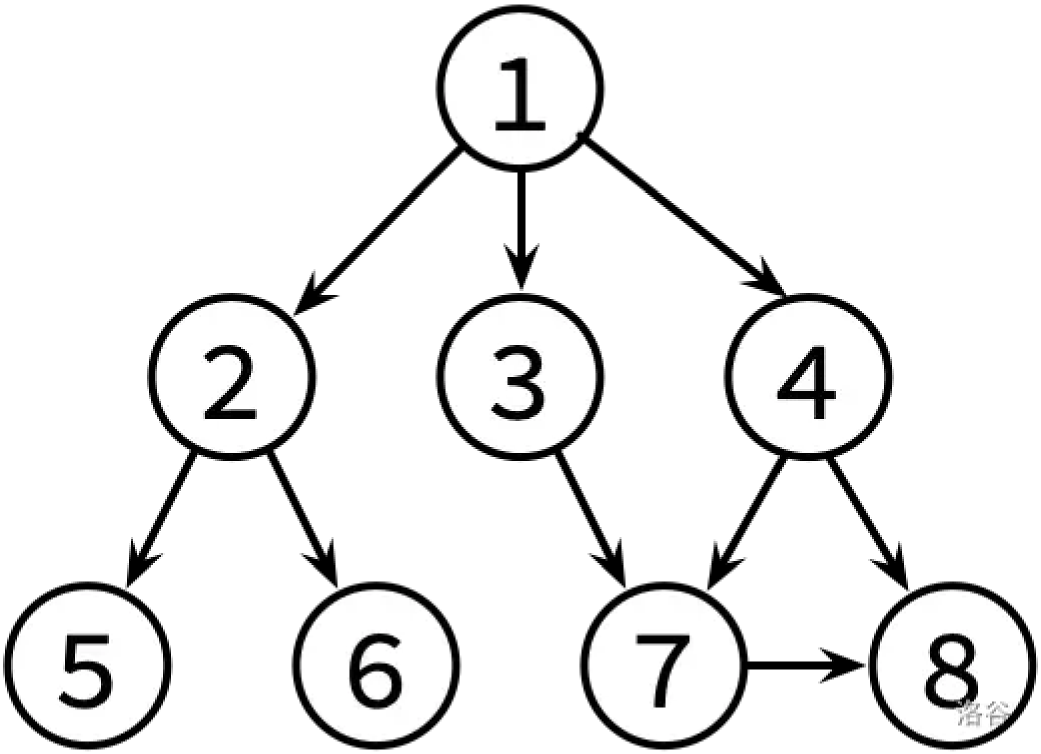
[P5318] 查找文献

题目描述

小 K 喜欢翻看洛谷博客获取知识。每篇文章可能会有若干个（也有可能没有）参考文献的链接指向别的博客文章。小 K 求知欲旺盛，如果他看了某篇文章，那么他一定会去看这篇文章的参考文献（如果他之前已经看过这篇参考文献的话就不用再看它了）。

假设洛谷博客里面一共有 $n(n \le 10^5)$ 篇文章（编号为 1 到 n）以及 $m(m \le 10^6)$ 条参考文献引用关系。目前小 K 已经打开了编号为 1 的一篇文章，请帮助小 K 设计一种方法，使小 K 可以不重复、不遗漏的看完所有他能看到的文章。

这边是已经整理好的参考文献关系图，其中，文献 X → Y 表示文章 X 有参考文献 Y。不保证编号为 1 的文章没有被其他文章引用。



请对这个图分别进行 DFS 和 BFS，并输出遍历结果。如果有很多篇文章可以参阅，请先看编号较小的那篇(因此你可能需要先排序)。

输入格式

共 $m+1$ 行，第 1 行为 2 个数，n 和 m，分别表示一共有 $n(n \leq 10^5)$ 篇文章（编号为 1 到 n）以及 $m(m \leq 10^6)$ 条参考文献引用关系。

接下来 m 行，每行有两个整数 X, Y 表示文章 X 有参考文献 Y。

输出格式

共 2 行。
第一行为 DFS 遍历结果，第二行为 BFS 遍历结果。

样例输入

```
8 9
1 2
1 3
1 4
2 5
2 6
3 7
4 7
4 8
7 8
```

样例输出

```
1 2 5 6 3 7 8 4
1 2 3 4 5 6 7 8
```

Solution

将所有从点u边输入后，按照终点从小到大排序，之后进行dfs和bfs即可

Code

```
void solve(){
    int n, m; cin >> n >> m;
    vector e(n + 1, vector<int>());
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v;
        e[u].push_back(v);
    }

    vector<int> vis(n + 1);
    for (auto& vec: e) sort(all(vec));

    auto dfs = [&](int u, auto dfs) -> void {
        if (vis[u]) return;
        vis[u] = 1; cout << u << ' ';
        for (auto v: e[u]) dfs(v, dfs);
    };

    dfs(1, dfs); cout << endl;

    fill(all(vis), 0);
    queue<int> q; q.push(1);
    while (!q.empty()) {
        auto u = q.front(); q.pop();
        if (vis[u]) continue;
        vis[u] = 1; cout << u << ' ';
        for (auto v: e[u]) q.push(v);
    }
}
```

树

图论中的**树**和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑。这种数据结构看起来像是一个倒挂的树，因此得名。

定义

无根树

一个没有固定根结点的树称为**无根树**。无根树有几种等价的形式化定义：

1. 有 n个点，$n-1$条边的**连通无向图**
2. 无向无环的连通图
3. 任意两个结点之间都只有一条**简单路径**的无向图
4. 没有**环**，且在**任意**不同两点添加一条边后，能形成唯一一个环的图

在无根树的基础上，指定一个结点称为**根**，则形成一棵**有根树**。有根树在很多时候仍以无向图表示，只是规定了结点之间的上下级关系

有根树

父亲：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。根结点没有父结点。

祖先：一个结点到根结点的路径上，除了它本身外的结点。根结点的祖先集合为空。

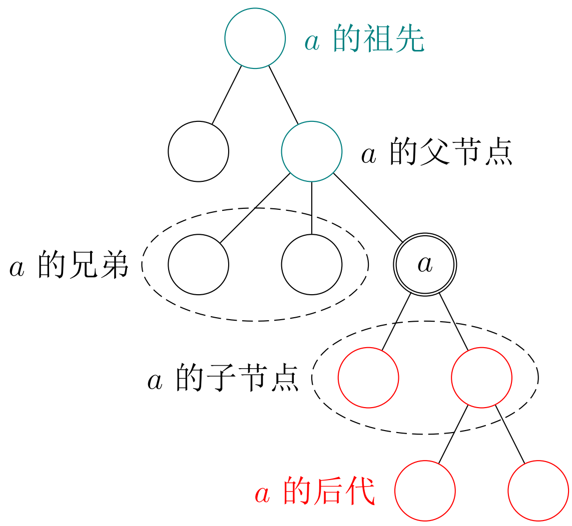
子结点：如果u是v的父亲，那么v是u的子结点。子结点的顺序一般不加以区分，二叉树是一个例外。

结点的深度：到根结点的路径上的边数。

树的高度：所有结点的深度的最大值。

兄弟：同一个父亲的多个子结点互为兄弟。

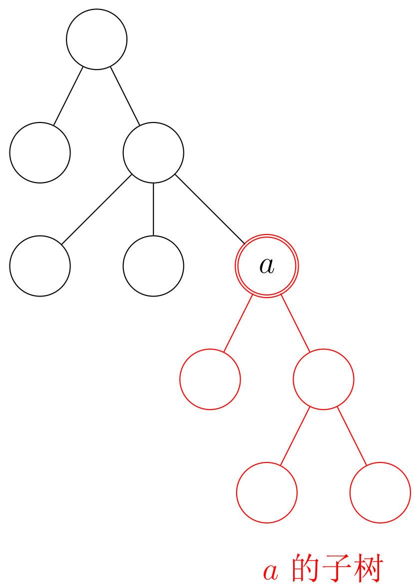
后代：子结点和子结点的后代。或者理解成：如果u是v的祖先，那么v是u的后代。



子树：删掉与父亲相连的边后，该结点所在的子图

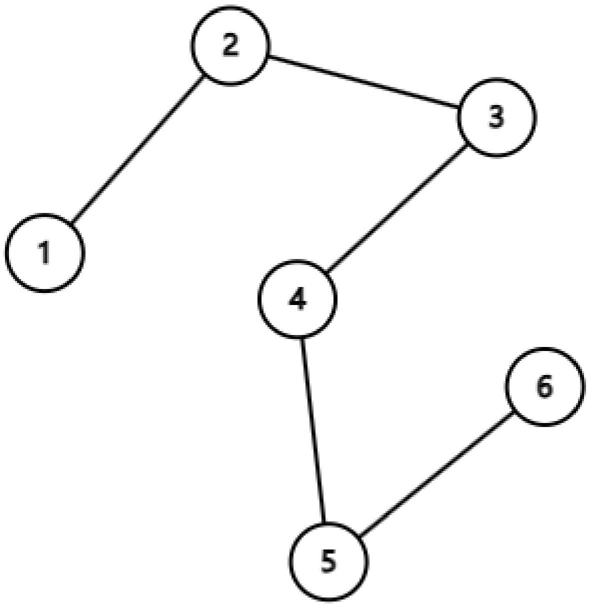
有根树的叶节点：没有子节点的节点

无根树的叶节点：度**小于等于1**的节点（为什么小于1？考虑$n = 1$的情况）

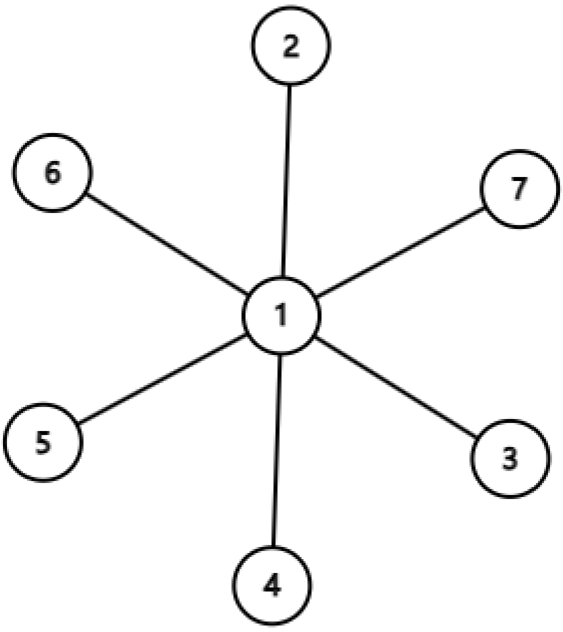


特殊的树

链：满足与任一结点相连的边不超过2条的树称为链



菊花/星星：满足存在u使得所有除u以外结点均与u相连的树称为菊花



有根二叉树：每个结点最多只有**两个儿子**的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。大多数情况下，**二叉树** 一词均指有根二叉树

树的存储

只记录父结点

利用一个`int parent[maxn]`数组，记录每个节点的**父亲**节点，这种记录方式能存储的信息比较少，通常用作与**自底向上**的递推问题中

邻接表

对于**无根树**而言，我们可以使用邻接表（思考：为什么不使用邻接矩阵），将树视为**无向图**进行存储

而对于**有根树**而言，我们可以将每条边**定向**，方向为**父亲节点**指向**儿子节点**，这样就可以转换成**有向图**进行处理；若有需要，还可在另一个数组中记录其父结点

```
std::vector<int> e[maxn];
int parent[maxn];
```

[# 3693] 树的存储

题目描述

告诉你一颗树有n个结点，然后告诉你n个父子结点关系，请按顺序输出$1 \sim n$个结点的父结点和子结点，根目录的父结点输出0

输入格式

第一行一个数$n(1 \leq n \leq 1000)$，表示树的结点数

然后n行，每行二个数$a_i, a_j(1 \leq a_i, a_j \leq n)$，结点$a_i$表示孩子，结点$a_j$表示父亲，即$a_j$是$a_i$的父亲。保证$a_j$为$0$或在前面已出现过。

输出格式

依次输出每个结点的父结点和孩子结点，每个结点输两行。

第一行表示该结点以及该结点的父亲，中间用`:`隔开；

第二行表示当前结点的孩子，孩子与孩子之间用空格隔开， 且孩子的编号要按从小到大排列。

对于叶结点，虽然它没有孩子，但是也要输出一个字符串`nobody`。

输入样例

```
7
1 0
2 1
3 1
4 2
5 3
7 4
6 2
```

输出样例

```
1:0
2 3
2:1
4 6
3:1
5
4:2
7
5:3
nobody
6:2
nobody
7:4
nobody
```

Solution

对于每个节点，我们开一个数组$e[i]$去维护它的**儿子集合**，同时，我们开一个大数组$fa[maxn]$，其中$fa[i]$用于记录i节点的父亲节点

在输出时，只需要注意提前把儿子集合排序即可

Code

```
void solve(){
    int n; cin >> n;
    vector<int> fa(n + 1);
    vector e(n + 1, vector<int>());

    for (int i = 0; i < n; i++) {
        int u, v; cin >> u >> v;
        fa[u] = v;
        e[v].push_back(u);
    }

    for (int i = 1; i <= n; i++) {
        cout << i << ':' << fa[i] << '\n';
        sort(all(e[i]));
        if (!e[i].size()) cout << "nobody";
        else for (auto v: e[i]) cout << v << ' ';
        cout << '\n';
    }
}
```

树的遍历

树上dfs

树上dfs类似于图的dfs：**先访问根节点**，然后分别访问根节点每个儿子的**子树**

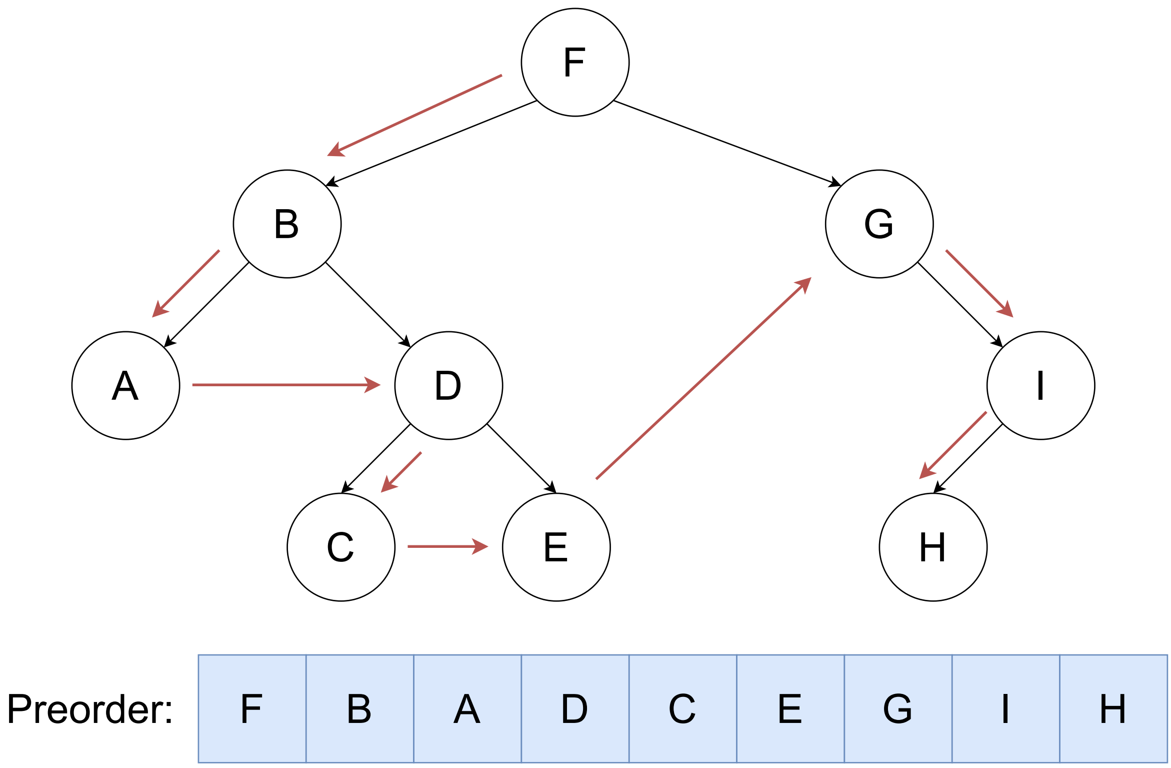
```
void dfs(int u, int fa) {
    // 对根节点做相应处理
    xxx;
    for (auto v: e[u]) {
        // 如果该节点是父亲节点，则不访问
        if (v == fa) continue;
        // v的父亲节点是u
        dfs(v, u);
    }
}
```

二叉树的dfs遍历

一般而言，使用dfs实现的树的遍历有三种方式，分别为：先序遍历，中序遍历，后序遍历

先序遍历

按照 **根，左，右** 的顺序遍历二叉树



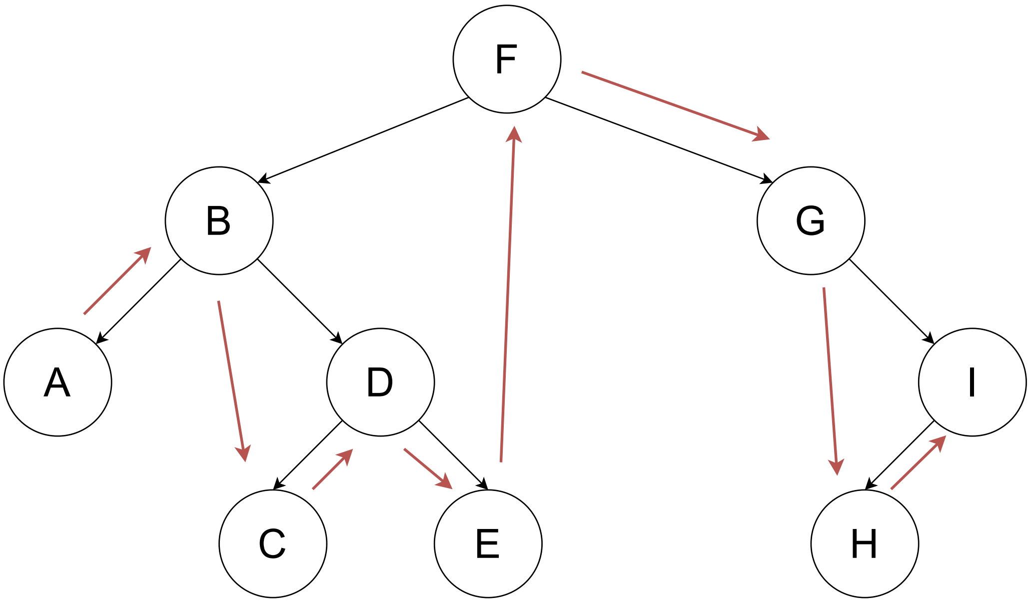

```cpp
// 使用lc / rc 数组存储二叉树
void preorder(int u) {
    // 访问根节点
    vis(u);
    // 访问左子树
    preorder(lc[u]);
    // 访问右子树
    preorder(rc[u]);
}

// 使用struct存储节点
struct node {
    int lc, int rc;
};

void preorder(const node& u) {
    // 访问根节点
    vis(u);
    // 访问左子树
    preorder(u.lc);
    // 访问右子树
    preorder(u.rc);
}
```

中序遍历

按照 **左，根，右** 的顺序遍历二叉树。

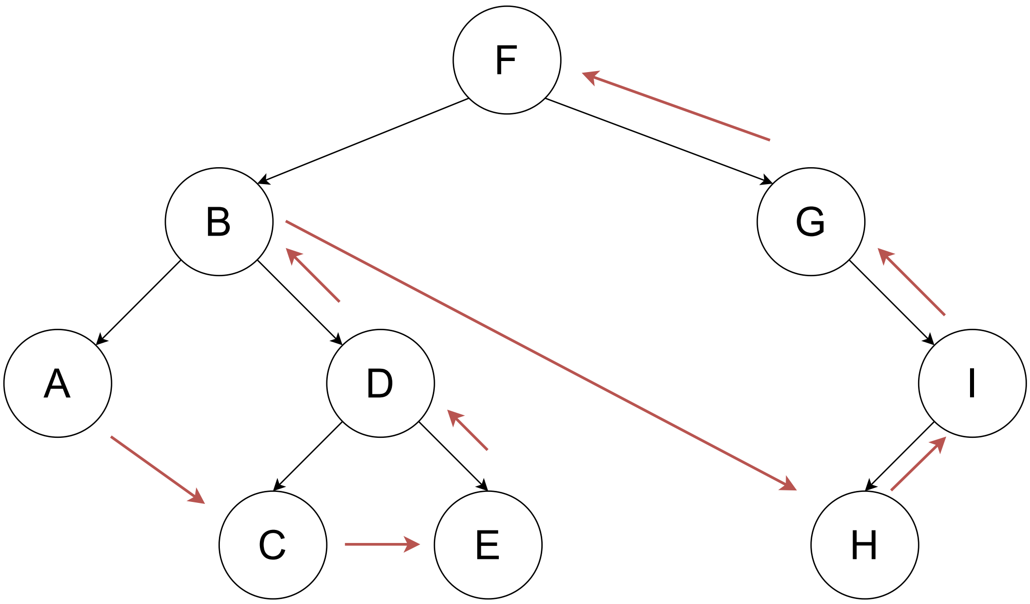


Inorder:	A	B	C	D	E	F	G	H	I

```
void inorder(int u) {
    // 访问左子树
    inorder(lc[u]);
    // 访问根节点
    vis(u);
    // 访问右子树
    inorder(rc[u]);
}
```

后序遍历

按照 **左，右，根** 的顺序遍历二叉树。



Postorder:	A	C	E	D	B	H	I	G	F

[# 3443] 二叉树的遍历

题目描述

给出一棵二叉树，分别输出先序、中序、后序遍历结果。

输入格式

第1行：结点数$n(1 \le n \le 100)$

以下若干行,每行3个整数，分别表示结点编号、左孩子编号、右孩子编号。若没有孩子，对应的整数为0.

输出格式

第1行：树根

第2行：先序遍历结果，数字间用1个空格分开。

第3行：中序遍历结果，数字间用1个空格分开。

第4行：后序遍历结果，数字间用1个空格分开。

样例输入

```
8
1 2 4
2 0 0
4 8 0
3 1 5
5 6 0
6 0 7
8 0 0
7 0 0
```

样例输出

```
3
3 1 2 4 8 5 6 7
2 1 8 4 3 6 7 5
2 8 4 1 7 6 5 3
```

Solution

按照上文所讲内容，简单实现即可

注意当某个正在访问或即将访问的节点为0节点时，需要直接`return`

Code

```
void solve(){
    int n; cin >> n;
    vector<int> lc(n + 1), rc(n + 1), fa(n + 1);
```

```
    for (int i = 1; i <= n; i++) {
        int u, l, r; cin >> u >> l >> r;
        lc[u] = l; rc[u] = r; fa[l] = fa[r] = u;
    }

    int rt = 1; while (fa[rt]) rt = fa[rt];

    auto preorder = [&](int u, auto preorder) -> void {
        if (!u) return;
        cout << u << ' ';
        preorder(lc[u], preorder);
        preorder(rc[u], preorder);
    };

    auto inorder = [&](int u, auto inorder) -> void {
        if (!u) return;
        inorder(lc[u], inorder);
        cout << u << ' ';
        inorder(rc[u], inorder);
    };

    auto backorder = [&](int u, auto backorder) -> void {
        if (!u) return;
        backorder(lc[u], backorder);
        backorder(rc[u], backorder);
        cout << u << ' ';
    };

    cout << rt << endl;
    preorder(rt, preorder); cout << endl;
    inorder(rt, inorder); cout << endl;
    backorder(rt, backorder); cout << endl;
}
```

[# 2912] 求二叉树后序遍历

题目描述

输入一棵二叉树的先序和中序遍历，输出其后序遍历序列。

输入格式

共两行，第一行一个字符串s_1，表示树的先序遍历，第二行一个字符串s_2，表示树的中序遍历。树的节点一律用小写字母表示。

保证$|s_1| = |s_2| \leq 26$

输出格式

仅一行，表示树的后序遍历。

样例输入

```
abdec
dbeac
```

样例输出

```
debca
```

Solution

对于一棵二叉树而言，其根节点在**先序遍历**时一定处于首位，并且，其所有的**左子树**节点都会先于**右子树**节点进行遍历

而在**中序遍历**时，在根节点**前面**的点是根节点的**左子树**，而在根节点**后面**的点是根节点的**右子树**

此时，我们可以利用**中序遍历**中，根节点的**位置**，得到左右两棵子树的大小信息，从而得到**先序遍历**中，哪一部分代表左子树，哪一部分代表右子树

此时，我们已经获得了左右两棵子树分别的先 / 中序遍历的结果；如此一来，我们便能将问题归结成获得左子树的后序遍历序列，获得右子树的后序遍历序列两个子问题，递归求解即可

Code

```
void solve() {
    string po, io;
    cin >> po >> io;

    auto process = [&](string po, string io, auto process) -> void {
        if (!po.size()) return;
        auto rt = po[0];
        int idx = 0;
        while (io[idx] != rt) idx++;

        process(po.substr(1, idx), io.substr(0, idx), process);
        process(po.substr(1 + idx), io.substr(idx + 1), process);
        cout << po[0];
    };

    process(po, io, process);
}
```