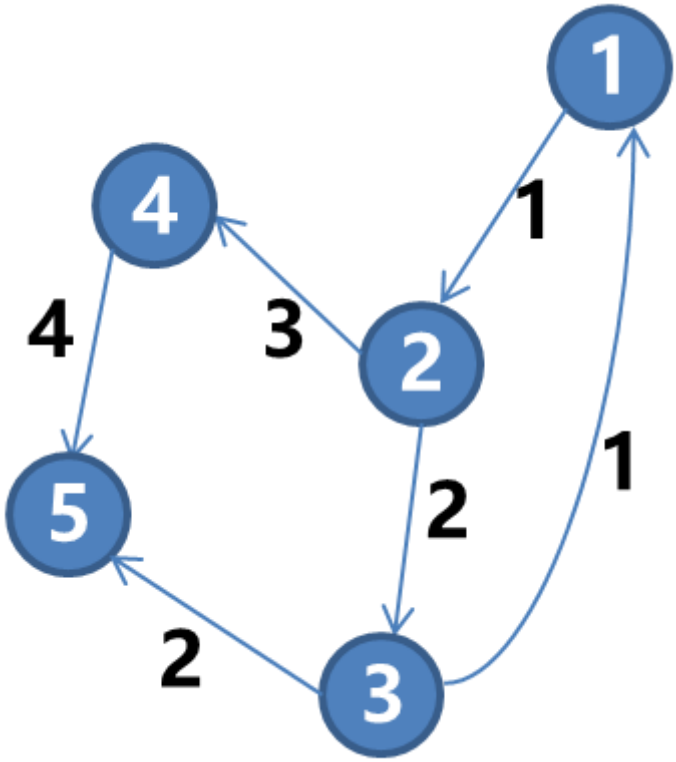
图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象。**图 (Graph)** 是由若干给定的**顶点**及连接两顶点的**边**所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。



基本定义

我们定义一张图 $G = (V, E)$，其中 V 代表点集，E 代表边集。对于 V 中的每个元素，称之为**顶点(Vertex)/节点(Node)**，简称点。E 中每个元素称之为**边（Edge)**，每条边连接着图中的两个节点。

图有很多种，包括无向图，有向图和混合图等等。在无向图中，每条边 $e = (u, v)$ 是一个无序的二元组，称作无向边，代表点 u 与点 v 双向可达，我们称 u, v 为边 e 的端点。在有向图中，每条边 $e = (u, v)$ （或者记作 $u \to v$） 是一个有序的二元组，称作**有向边**，代表点 u 单向可以到达点 v，这里我们把 u 称为起点，v 称为终点。混合图就是既有有向边又有无向边的图。

图中的每个点具有**度**的属性：一个点的度代表与此点相连边的数量，入度代表终点为该点的边的数量，出度代表起点为该点的边的数量。

每条边可以有一个长度，当然也可以没有（默认为1）。图论题面中顶点的个数通常用 n 表示，边的数量通常用 m 表示。

存图

邻接矩阵

我们用一个 $n \times n$ 的二维数组来存储一个图，存储规则：若 x 能到达 y，那么 (x, y) 代表的就是 x 到 y 的距离，如果无法访问，我们一般用 -1 或者 $+\infty$ 来表示。特别地，自己到自己的距离为 0。比如开头那个图就可以这么表示：

	1	2	3	4	5
1	0	1	-1	-1	-1
2	-1	0	2	3	-1
3	1	-1	0	-1	2
4	-1	-1	-1	0	4
5	-1	-1	-1	-1	0

邻接矩阵只适用于没有重边（或重边可以忽略）的情况。其最显著的优点是可以 $O(1)$ 查询一条边是否存在。

由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图(边比点多很多)上使用邻接矩阵。

时间复杂度：查询边 $O(1)$，遍历所有出边 $O(n)$，遍历全图 $O(n^2)$，空间复杂度 $O(n^2)$

```
int G[N][N];

void add(int x,int y) {
    G[x][y] = 1;
}

void init() {
    //初始化整张图
    memset(G,0x3f,sizeof G);
    for (int i = 1;i <= n;++i) G[i][i] = 0;
}
//遍历全图
for (int i = 1;i <= n;++i) {
    for (int j = 1;j <= n;++j) {
        if (i == j) continue;
    }
}
```

邻接表

邻接表（链式前向星）存储图比邻接矩阵更加高效。邻接表由点表（由点构成的表）（上）和边表（由边构成的表）（下）组成。`Head`数组存储的是以该点为起点（按照加入时间顺序）最后加入的一条边。边表 `from` 数组实际操作没有用处，不过在查错环节还是有用的。`Next` 数组是以该边起始点为起点的上一条边（按照加入时间顺序）

参考代码（链式前向星）：

```
int head[N],nxt[N],to[N],tot;

//加边
void add(int x,int y) {
    ++tot;
```

```
    nxt[tot] = head[x];
    head[x] = tot;
    to[tot] = y;
}

//遍历x连出的所有边
for (int i = head[x];i;i = nxt[i]) {
    int y = to[i];//i代表 x -> y这条边
}
```

当然更多的是用 `vector` 实现（邻接表）：

```
vector <int> G[N];
//如果需要存带边权的用 pair 就可以啦!

void add(int x,int y) {
    G[x].push_back(y);
}

for (int y : G[x]) {

}
```

时间复杂度：查询边 $O(d(x))$，遍历所有出边 $O(d(x))$，遍历全图 $O(n+m)$，空间复杂度 $O(m)$ （$d(x)$ 代表 x 的出度）

小 Trick：如果用链式前向星，`tot=-1`的话，那么无向边 `i,i^1` 恰好是一对边。

最短路

Floyd

是用来求任意两个结点之间的最短路的。复杂度比较高，但是常数小，容易实现（只有三个 `for`）。适用于任何图，不管有向无向，边权正负，但是最短路必须存在。（不能有个负环）

实现

初始化：

1. 不可以直接到达的 `dis` 设为正无穷($+\infty$)。
2. 自己到自己的距离为0。
3. 题目给定的边的 $dis[i][j]$ 直接赋值为该边长度。双向边需要 $dis[i][j]$ 和 $dis[j][i]$ 均赋值为边长。

Floyd算法使用的思想是：枚举中转节点 k。检查由 x 点经过此点到 y 点的路径是否比原先优。更新由 x 点到 y 点的最短距离。

我们考虑采用动态规划的形式，定义 $f[k][i][j]$ 为由点 i 到 点 j，只允许经过点 $1,2\ldots,k$ 的最短路（即前 k 个点中 i 到 j 的最短路）。

那么我们有初始状态 $f[0][i][j]=dis[i][j]$ 转移方程：

$$f[k][i][j]=\min\{f[k][i][j],f[k-1][i][k]+f[k-1][k][j]\}$$

不难写出代码：

```
void Floyd {
    for (int k = 1; k <= n; k++) {
      for (int x = 1; x <= n; x++) {
        for (int y = 1; y <= n; y++) {
          f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k]
[y]);
        }
      }
    }
}
```

根据我们前几天对于 DP 的学习，我们不难观察出来第一维实际上对结果没有影响，可以直接改成：

$$f[i][j]=\min\{f[i][j],f[i][k]+f[k][j]\}$$

> Q1：为什么第一维对结果没有影响?
>
> 对于给定的 `k`，当更新 `f[k][x][y]` 时，涉及的元素总是来自 `f[k-1]` 数组的第 `k` 行和第 `k` 列。然后我们可以发现，对于给定的 `k`，当更新 `f[k][k][y]` 或 `f[k][x][k]`，总是不会发生数值更新，因为按照公式 `f[k][k][y] = min(f[k-1][k][y], f[k-1][k][k]+f[k-1][k][y])`,`f[k-1][k][k]` 为 0，因此这个值总是 `f[k-1][k][y]`，对于 `f[k][x][k]` 的证明类似。
>
> 因此，如果省略第一维，在给定的 `k` 下，每个元素的更新中使用到的元素都没有在这次迭代中更新，因此第一维的省略并不会影响结果。

```
void Floyd() {
    for (int k = 1;k <= n;++k) {
        for (int i = 1;i <= n;++i) {
            for (int j = 1;j <= n;++j) {
                f[i][j] = min(f[i][j],f[i][k] + f[k][j]);
            }
        }
    }
}
```

时间复杂度$O(N^3)$

> Q2：为什么要先枚举 k 啊?
>
> 否则便过早的把 i 到 j 的最短路径确定下来了，而当后面存在更短的路径时，已经不再会更新了。
>
> 假如我们求 A 到 B 的最短距离。
>
> 在内层枚举 k：只有 $A\to B$唯一一条路线（因为咱们只能枚举一个中转点）

> 在外层枚举 k：可以经过 D 求得 A 到 C 的距离，再经过 C 求得 A 到 B 的距离。

应用

> 例：求最小环 HDU1599
>
> 记原图中 u, v 之间边的边权为 $val\,(u, v)$。
>
> 我们注意到 Floyd 算法有一个性质：在最外层循环到点 k 时（尚未开始第 k 次循环），最短路数组 dis 中，$dis_{u,v}$ 表示的是从 u 到 v 且仅经过编号在 $[1, k)$ 区间中的点的最短路。
>
> 由最小环的定义可知其至少有三个顶点，设其中编号最大的顶点为 w，环上与 w 相邻两侧的两个点为 u, v，则在最外层循环枚举到 $k = w$ 时，该环的长度即为 $dis_{u,v} + val\,(v, w) + val\,(w, u)$。
>
> 故在循环时对于每个 k 枚举满足 $i < k, j < k$ 的 (i, j)，更新答案即可。

Dijkstra

算法原理

实现思路：

1. 将顶点划为两堆，起初第一堆只有起点 S 这一个点。
2. 每次从第二堆里距离 S 点最近的点（这就是贪心了）取出，放入第一堆中，并更新最短路，直到第二堆中没有节点为止。
3. 此时维护出的 $dist[i]$ 就是从 S 点到 i 点的最小距离了。**注意：Dijkstra 只能处理正权边。**

正常实现是 $O(N^2)$ 的，但是通过优先队列优化我们可以以 $O(n \log n)$ 的时间复杂度：也就是步骤 2 中我们找最小点的这个过程可以做到 $O(\log n)$。

复杂度证明可见：最短路 - OI Wiki

参考实现

```
int n,m,s,dis[N];
bool vis[N];

struct node {
    int pos,dis;
    friend bool operator < (const node &a,const node &b) {
        return a.dis > b.dis;
    }
 };

priority_queue <node> q;

void Dijkstra(int s) {
    memset(dis,0x3f,sizeof dis);
    memset(vis,0,sizeof vis);
    q.push((node){s,dis[s] = 0});
    while (!q.empty()) {
```

```
        node p = q.top();q.pop();
        int x = p.pos;
        if (vis[x]) continue;
        vis[x] = 1;
        for (auto e : G[x]) {
            int y = e.first,w = e.second;
            if (dis[y] > dis[x] + w) {
                dis[y] = dis[x] + w;
                q.push((node){y,dis[y]});
            }
        }
    }
}
```

Bellman Ford && SPFA

上面用的 Dijkstra 虽然很强势，但是它只能用来处理非负权边的图。要是图中有负权边那么 Dijkstra 就倒闭了。主播主播有没有好写又跑的还可以的算法呢?

有的兄弟有的，Bellman Ford 算法（当然实际中我们更多应用的是它的队列优化版本，即 SPFA）就是这样的!

Bellman Ford

我们刚才在 Dijkstra 算法中提到的“更新最短路”这个操作，实际上叫做**松弛**操作。也就是对于一条边 (x, y, w)，我们有 $dis_y = \min\{dis_y, dis_x + w\}$

这个操作的含义就是尝试用 (x, y, w) 这条边，加上从原点到 x 的最短路这条路径尝试去更新从原点到 y 的最短路，如果这条路径更优就尝试更新。Bellman-Ford算法做的就是不断尝试对图上每一条边进行松弛操作，每轮循环，我们对所有边都跑一遍。如果一次循环中我们发现图上没有可以松弛的边，操作就停止，我们就求出了最短路。

由于每轮至少更新一条最短路，最短路的边数至多为 $n-1$，所以算法的时间复杂度是 $O(nm)$ 级别的。但是要注意，如果图中存在一个可以从 S 出发到达的环，环的权值之和为负数的话（称之为负环），那么上文的操作就会进行无穷多轮，所以判断是否存在负环的方法也很简单：看一下第 n 轮是否还存在可以松弛的边即可。

参考实现:

```
struct Edge {
  int u, v, w;
};

vector<Edge> edge;

int dis[MAXN], u, v, w;
constexpr int INF = 0x3f3f3f3f;

bool bellmanford(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
```

```
  bool flag = false;  // 判断一轮循环过程中是否发生松弛操作
  for (int i = 1; i <= n; i++) {
    flag = false;
    for (int j = 0; j < edge.size(); j++) {
      u = edge[j].u, v = edge[j].v, w = edge[j].w;
      if (dis[u] == INF) continue;
      // 无穷大与常数加减仍然为无穷大
      // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        flag = true;
      }
    }
    // 没有可以松弛的边时就停止算法
    if (!flag) {
      break;
    }
  }
  // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
  return flag;
}
```

SPFA

很多时候我们并不需要那么多无用的松弛操作。

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。

那么我们用队列来维护「哪些结点可能会引起松弛操作」，就能只访问必要的边了。

SPFA 也可以用于判断 S 点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少 n 条边时，说明 S 点可以抵达一个负环。

在随机图的情况下，SPFA 的时间复杂度为 $O(km)$，其中 k 是一个不大的常数。但是最坏情况下复杂度为 $O(nm)$，并且特别好卡，基本上正式赛场上正权图能卡都会卡一下 SPFA，所以**在没有负权边时最好使用 Dijkstra！！！**

参考实现：

```
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], cnt[MAXN], vis[MAXN];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
```

```
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= n) return false;
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return true;
}
```

三个最短路算法的比较

Floyd算法

优点：

1. **多源最短路径**：Floyd算法可以求解所有顶点对之间的最短路径，即多源最短路径问题。
2. **适合负权边**：Floyd算法可以处理图中存在负权边的情况，但不能处理负权环。
3. **实现简单**：Floyd算法的实现相对简单，代码结构清晰。

缺点：

1. **时间复杂度较高**：Floyd算法的时间复杂度为 $O(N^3)$，其中V是顶点数。对于大规模图，Floyd算法的效率较低。
2. **空间复杂度较高**：Floyd算法需要存储一个 $N \times N$ 的矩阵，空间复杂度为 $O(N^2)$。

Dijkstra

优点：

1. **时间复杂度较低**：Dijkstra算法使用优先队列（如二叉堆）时，时间复杂度为 $O((V+E)logV)$，其中 V 是顶点数，E 是边数。对于稀疏图（边数较少），Dijkstra算法效率较高。
2. **单源最短路径**：Dijkstra算法适用于求解单源最短路径问题，即从一个起点到其他所有顶点的最短路径。
3. **适合正权图**：Dijkstra算法要求图中边的权重为非负数，因此在正权图中表现良好。

SPFA

优点：

1. **可以求解带负权边的图**：本质上是 Bellman-Ford 算法，所以可以在带有负权边的图跑。

总结

- **Dijkstra算法**适用于单源最短路径问题，尤其是在正权稀疏图中表现良好，经常在图的大小为 $10^6, 10^5$ 左右跑。
- **Floyd算法**适用于多源最短路径问题，尤其是在小规模图或存在负权边的情况下，经常在 $n = 500$ 左右跑。
- **SPFA 算法**适用于存在负权边的单源最短路径问题，主要是在含有负权边的图跑，正常情况下最好不要在无负权边的图用。

树

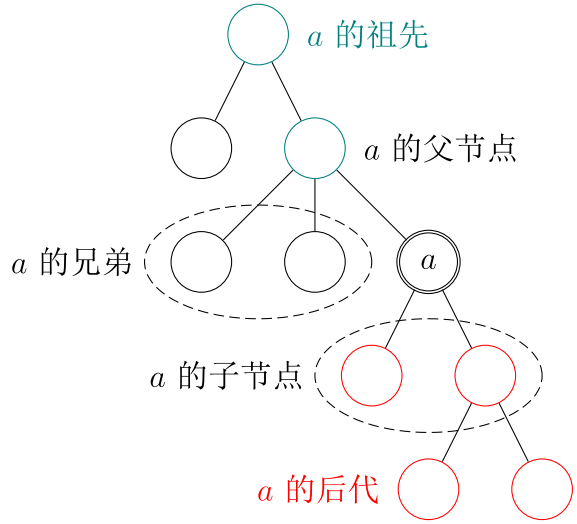
图论中的树和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑。这种数据结构看起来像是一个倒挂的树，因此得名。

定义

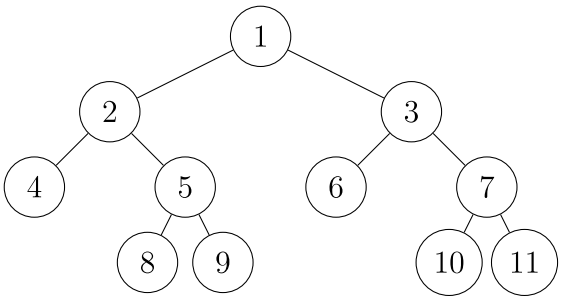
一个没有固定根结点的树称为 **无根树**（unrooted tree）。无根树有几种等价的形式化定义：

- 有 n 个结点，$n-1$条边的连通无向图
- 无向无环的连通图
- 任意两个结点之间有且仅有一条简单路径的无向图

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**（rooted tree）。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。

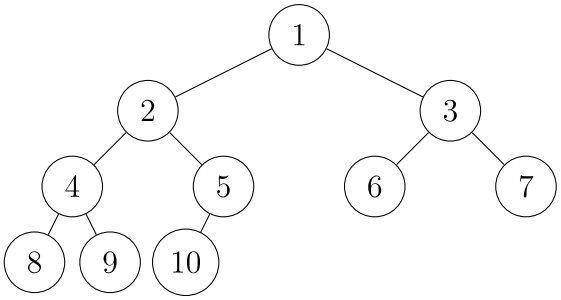
> 相关概念：
>
> - **父亲（parent node）**：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。根结点没有父结点。
> - **祖先（ancestor）**：一个结点到根结点的路径上，除了它本身外的结点。根结点的祖先集合为空。
> - **子结点（child node）**：如果 x 是 y 的父亲，那么 y 是 x 的子结点。子结点的顺序一般不加以区分，二叉树是一个例外。
> - **结点的深度（depth）**：到根结点的路径上的边数。
> - **树的高度（height）**：所有结点的深度的最大值。
> - **兄弟（sibling）**：同一个父亲的多个子结点互为兄弟。
> - **直径：**树上任意两节点之间最长的简单路径即为树的「直径」。
>
> 
>

- **链（chain/path graph）**：满足与任一结点相连的边不超过 条的树称为链。
- **菊花/星星（star）**：满足存在 使得所有除 以外结点均与 相连的树称为菊花。
- **有根二叉树（rooted binary tree）**：每个结点最多只有两个儿子（子结点）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。
- **完整二叉树（full/proper binary tree）**：每个结点的子结点数量均为 0 或者 2 的二叉树。换言之，每个结点或者是树叶，或者左右子树均非空。



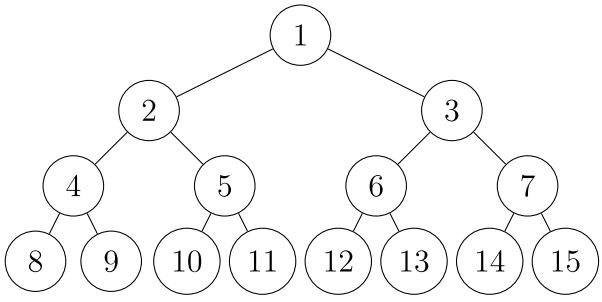
完整二叉树（proper binary tree）

- **完全二叉树（complete binary tree）**：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。



完全二叉树（complete binary tree）

- **完美二叉树（perfect binary tree）**：所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树。



完美二叉树（perfect binary tree）

存储以及遍历

只储存父节点

用一个数组 `fa[N]` 记录每个结点的父亲结点。

这种方式可以获得的信息较少，不便于进行自顶向下的遍历。常用于自底向上的递推问题中。

邻接表

当作无向图来存就 OK 了。两种方法可以一起用

如何遍历？直接 `dfs`！过程中记得记录父亲是谁避免重复访问

```
void dfs(int x,int fa) {
    for (auto y : G[x]) {
        if (y == fa) continue;
        dfs(y,x);
    }
}
调用：dfs(root,0);
```

也可以 `bfs` 因为有天然的层次性：

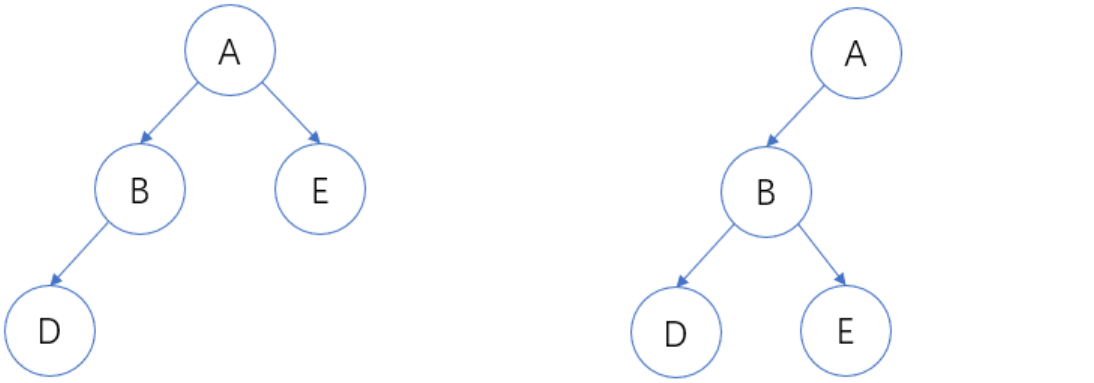
```
queue <int> q;
void bfs(int root) {
    q.push(root);
    while (!q.empty()) {
        int x = q.front();q.pop();
        //do something
    }
}
调用：bfs(root)
```

左孩子右兄弟表示法

也叫树的二叉树表示法

树的左指针指向自己的第一个孩子，右指针指向与自己相邻的兄弟。

结构的最大优点是：它和二叉树的二叉链表表示完全一样。可利用二叉树的算法来实现对树的操作 。



二叉树表示法

首先，给每个结点的所有子结点任意确定一个顺序。

此后为每个结点记录两个值：其 **第一个子结点** `child[u]` 和其 **下一个兄弟结点** `sib[u]`。若没有子结点，则 `child[u]` 为空；若该结点是其父结点的最后一个子结点，则 `sib[u]` 为空

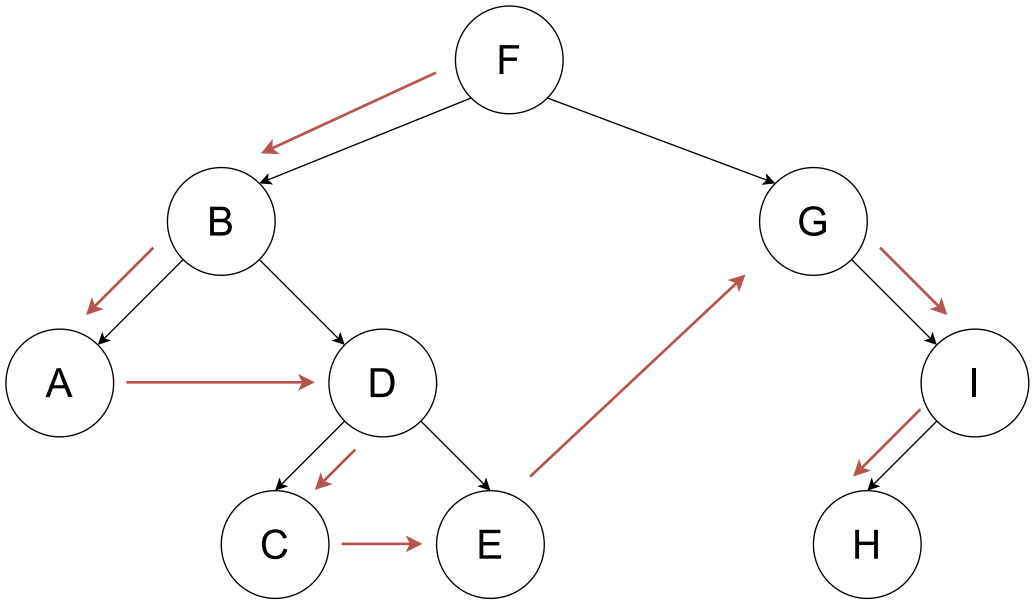
如何遍历：

```
int v = child[u];  // 从第一个子结点开始
while (v != EMPTY_NODE) {
  // ...
  // 处理子结点 v
  // ...
  v = sib[v];  // 转至下一个子结点，即 v 的一个兄弟
}
```

二叉树遍历

二叉树是一种特殊的树，有三种遍历方式：前序遍历、中序遍历、后序遍历

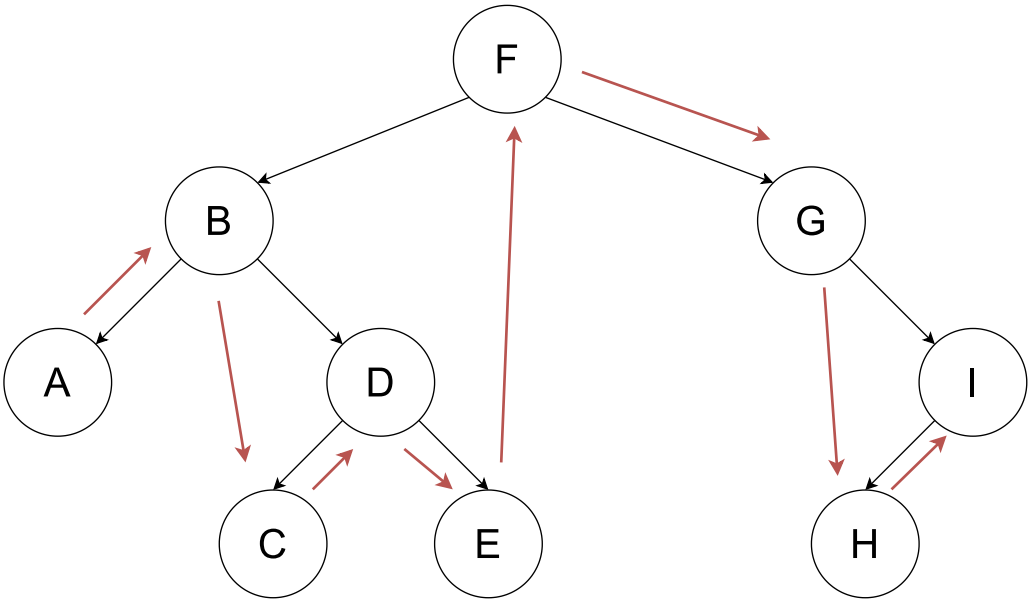
先序遍历



Preorder:	F	B	A	D	C	E	G	I	H

按照 **根，左，右** 的顺序遍历二叉树。

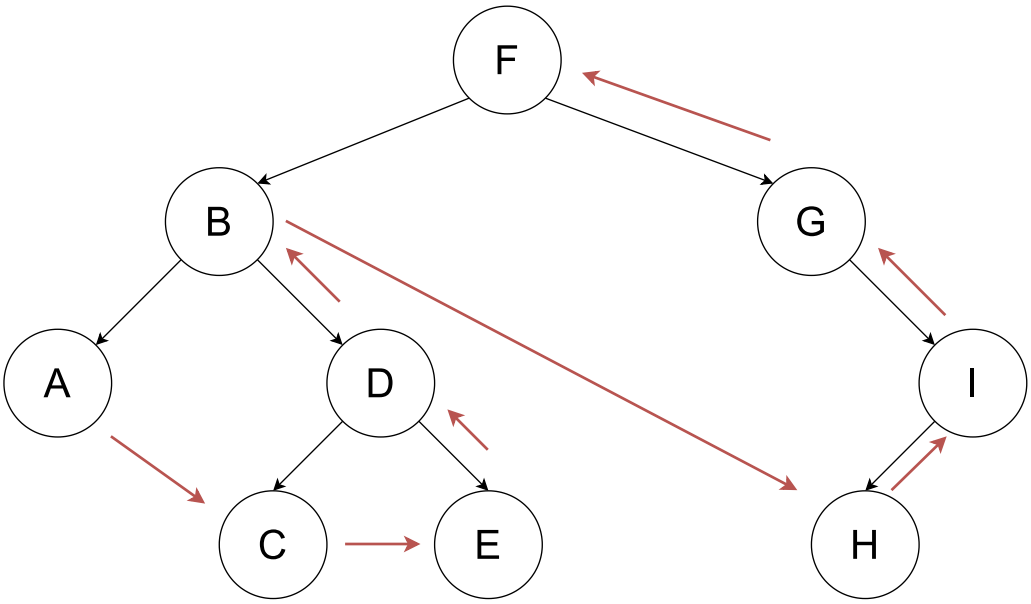
中序遍历



Inorder:	A	B	C	D	E	F	G	H	I

按照 **左，根，右** 的顺序遍历二叉树。

后序遍历



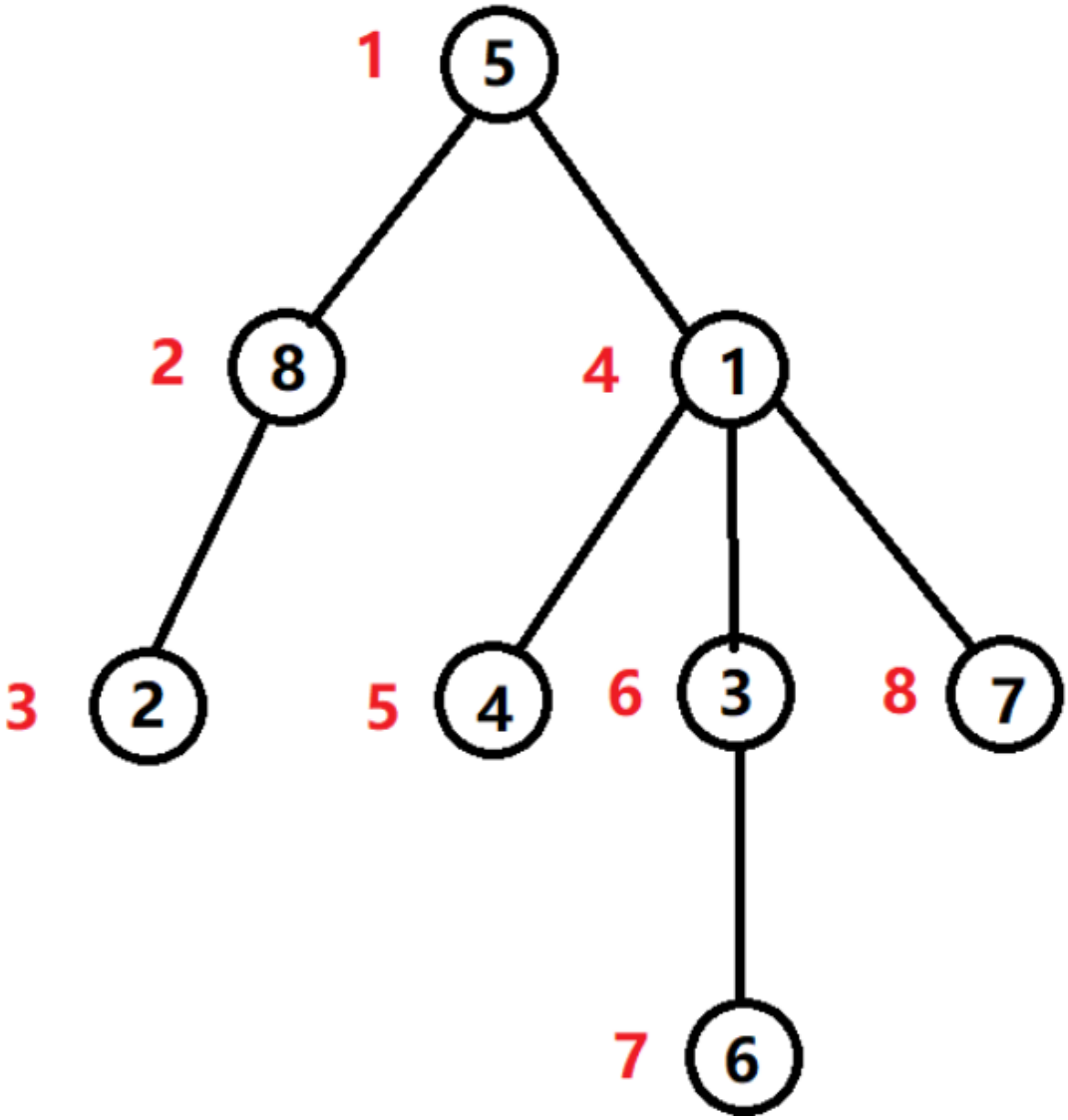
Postorder:	A	C	E	D	B	H	I	G	F

按照 **左，右，根** 的顺序遍历二叉树。

DFS 序

DFS 序是指 DFS 调用过程中访问的节点编号的序列。我们发现，每个子树都对应 DFS 序列中的连续一段（一段区间）。

树是一种非线性的数据结构，它的一些数据调用肯定是没有线性结构来得方便的。所以基于 DFS 函数，我们可以在遍历的同时记录下每个节点进出栈的时间序列。然后我们就把一棵树变成了一个序列，你就可以用很多数据结构做很多问题啦！



实际实现很简单，我们只需要在 DFS 开头加上一句就可以：

```
void dfs(int x,int fa) {
    dfn[x] = ++cnt;
    for (auto y : G[x]) {
        if (y == fa) continue;
        dfs(y,x);
    }
}
```

树形 DP

树形 DP，即在树上进行的 DP。由于树固有的递归性质，树形 DP 一般都是递归进行的。

根据我们上一节课学到的知识，我们一般以 f_x 代表以 x 为根的子树，这个是我们的状态。至于阶段的划分一般是根据子树来划分。如果需要多涵盖一些信息，一般用 $f_{x,i,j,\dots}$ 的状态来刻画。

具体来说，在树形动态规划当中，我们一般先算子树再进行合并，在实现上与树的后序遍历相似，都是先遍历子树，遍历完之后将子树的值传给父亲。简单来说我们动态规划的过程大概就是先递归访问所有子树，再在根上合并。

了解了树形动态规划的基本思想后，我们可以通过一些例题来了解一下树形 DP。

> 例1： 给一棵 n 个点的无权树，求树的重心?
>
> 重心定义为，删去该点之后，图中的所有连通块的 最大尺寸最小
>
> 其中 $0 \leq n \leq 10^5$

我们用 siz_x 代表 x 子树的节点数，那么有 $siz_x = \sum\limits_{y \in son(x)} siz_y$，

用 mx_x 代表删去 x 之后的最大子树节点数，那么有 $mx_x = \max\limits_{y \in son(x)} \{siz_y, n - siz_x\}$，因为不仅要考虑 x 的某个子树 y，还要考虑去掉 x 这整棵子树的 $n - siz_x$ 这些点。

然后找 mx_x 最小的点 x 即可。

实现用 DFS 实现:

```
void dfs(int x,int fa) {
    siz[x] = 1;
    for (int y : G[x]) {
        if (y == fa) continue;
        dfs(y,x);
        siz[x] += siz[y];
        mx[x] = max(mx[x],siz[y]);
    }
    mx[x] = max(mx[x],n - siz[x]);
}
```

> 例2：P1352 没有上司的舞会
>
> 某大学有 n 个职员，编号为 $1 \ldots n$。
>
> 他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。
>
> 现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 r_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。
>
> 所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

简化一下，如果一个节点的父节点被选中，则该节点不能被选中，求最大点权和。

我们可以定义 $f_{i,0}$ 为该节点不选中时的最大值，$f_{i,1}$ 为该节点被选中的最大值，p_x 为 x 的子节点个数，$son（x）$为 x 的子节点点集。

转移方程就呼之欲出了:

$$f_{x,0} = \sum_{y \in son(x)} \max(f_{y,0}, f_{y,1})$$

$$f_{x,1} = \sum_{y \in son(x)} f_{y,0}$$

如果当前节点不选，则子节点选不选都可以，求最大值就行。

如果当前节点选，显然子节点只能都不选，暴力求和即可。

代码实现

```
void dp(int x) {
    f[x][0] = a[x],f[x][1] = 0;//每次的初始化，将选择这个节点的值设为这个节点的值，不选择这个节点的值设为0
    int p = G[x].size();
    for (int i = 0;i < p;++i) {
        int y = G[x][i];//遍历儿子
        dp(y);//对每一个儿子进行dp
        f[x][1] += max(f[y][0],f[y][1]);//不选择这个节点，则两种决策都不会受到影响
        f[x][0] += f[y][1];//选择这个节点，子节点只能不选
    }
}
```

最后答案就是 $\max(f_{n,0}, f_{n,1})$

拓展阅读：[学习笔记]换根dp - 洛谷专栏，树形背包导论

这俩都是树形 DP 里特别经典的模型扩展！推荐有兴趣的同学学习。