第五章 二叉树与图

文章目录

二叉树与图

二叉树

简介

树是n（n>=0）个节点的有限集，且这些节点满足如下关系：

1. 有且仅有一个结点没有父结点，这些结点称为树的根。
2. 除根外，其余每个结点都有且近有一个父结点。
3. 树中的每个结点都构成一个以它为根的树。
   二叉树满足树的条件时，满足如下条件：
   每个结点最多有两个孩子（子树），这两个子树有左右之分，次序不可颠倒。

数据结构与构造

数据结构

```
public class TreeNode {
     int val;
     TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}
```

构造

```
TreeNode node1 = new TreeNode(1);
TreeNode node2 = new TreeNode(2);
TreeNode node3 = new TreeNode(3);
node1.left = null;
node1.right = node2;
node2.left = node3;
node2.right = null;

//输出结果
   1
    \
     2
    /
   3
```

遍历

```
// 1.先序遍历
// 根左右
public void pre_order(Node node){
    if(node != null){
        system.out.println(node.var);
        pre_order(node.left);
        pre_order(node.right);
    }
}

// 2.中序遍历
// 左根右
public void in_order(Node node){
    if(node != null){
        pre_order(node.left);
```

```
        system.out.println(node.var);
        pre_order(node.right);
    }
}

// 3.后序遍历
// 左右根
public void pre_order(Node node){
    if(node != null){
        pre_order(node.left);
        pre_order(node.right);
        system.out.println(node.var);
    }
}
```

图

简介

图（Graph）是由顶点的又穷非空集合和顶点之间的集合组成，通常表示为G（V，E），其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合。

图分无向图和有向图，根据图的边长，又分带权图与不带权图。

构造

（1）邻接矩阵

```
const int MAX_N = 5; // 一共5个顶点
int Graph[MAX_N][MAX_N] = {0}; // 使用邻接矩阵表示
//将图连通，且不带权，一般用邻接矩阵表示稠密图
Graph[0][2] = 1;
Graph[0][4] = 1;
Graph[1][0] = 1;
Graph[1][2] = 1;
```

（2）邻接表

```
// 图的邻接表数据结构
struct GraphNode{
    int label;// 图的顶点的值
    vector<GraphNode *> neighbors;// 相邻节点指针数组
    GraphNode(int x){
        label = x;
    }
}

int main(){
    const int MAX_N = 5;
    GraphNode *Graph[MAX_N]; // 5个顶点
    for(int i=0;i<MAX_N;I++){
        Graph[i] = new GraphNode(i);
}
// 添加边
    GraphNode[0]->neighbors.push_bak(Graph[2]);
    GraphNode[0]->neighbors.push_bak(Graph[4]);
    GraphNode[1]->neighbors.push_bak(Graph[0]);
    GraphNode[1]->neighbors.push_bak(Graph[2]);
    ...
}
```

这里采用的图的数据结构及构造方法如下：

图的数据结构

```
    // 图结点（带权/不带权）
    public class GraphNode{
        public char key;
        public List neighbors;
        // 构造
        public GraphNode(char key) {
            this.key = key;
            this.neighbors = new ArrayList();
        }
        // 添加邻结点
        public void setNeighbors(List neighbors) {
            this.neighbors = neighbors;
        }
    }

    // 带距离邻结点
    public class neighborNode {
        public char key; // 该结点的标识

        public neighborNode(char key) {
                this.key = key;
        }
    };

    // 带距离邻结点
    public class neighborNodeWithDis {
        public char key; // 该结点的标识
        public int dis; // 到相邻结点的值

        public neighborNodeWithDis(char key,int dis) {
                this.key = key;
                this.dis = dis;
        }
    };
```

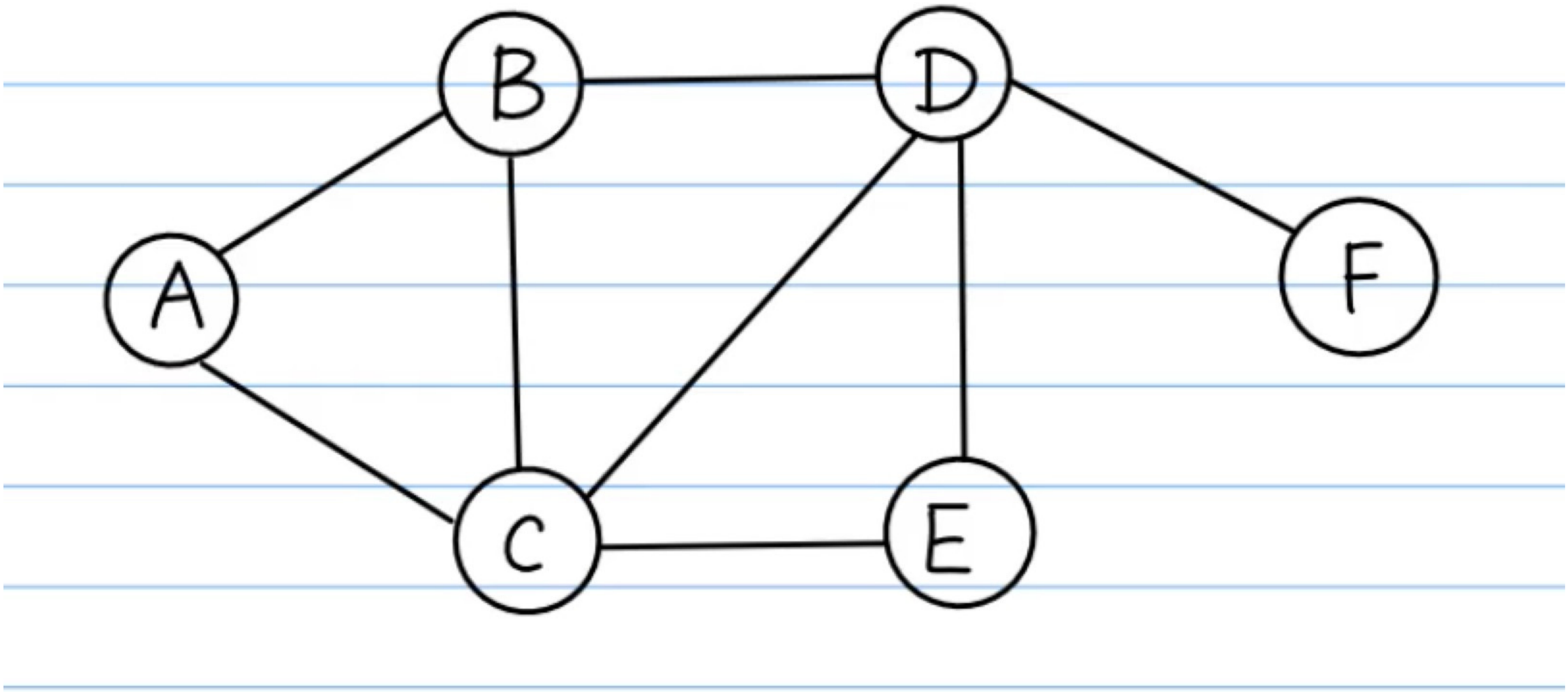
图的构造方法

```
    // 图的构造（不带权）
    public List<GraphNode> initGraphWithoutValue() {
```

```
        // 全局图（非加权）
        List<GraphNode> graph = new ArrayList<>();

        GraphNode A = new GraphNode('A');
        List NeighborsOfNodeA = new ArrayList();
        NeighborsOfNodeA.add(new neighborNode('B'));
        NeighborsOfNodeA.add(new neighborNode('C'));
        A.setNeighbors(NeighborsOfNodeA);

        GraphNode B = new GraphNode('B');
        List NeighborsOfNodeB = new ArrayList();
        NeighborsOfNodeB.add(new neighborNode('A'));
        NeighborsOfNodeB.add(new neighborNode('C'));
        NeighborsOfNodeB.add(new neighborNode('D'));
        B.setNeighbors(NeighborsOfNodeB);

        GraphNode C = new GraphNode('C');
        List NeighborsOfNodeC = new ArrayList();
        NeighborsOfNodeC.add(new neighborNode('A'));
        NeighborsOfNodeC.add(new neighborNode('B'));
        NeighborsOfNodeC.add(new neighborNode('D'));
        NeighborsOfNodeC.add(new neighborNode('E'));
        C.setNeighbors(NeighborsOfNodeC);

        GraphNode D = new GraphNode('D');
        List NeighborsOfNodeD = new ArrayList();
        NeighborsOfNodeD.add(new neighborNode('B'));
        NeighborsOfNodeD.add(new neighborNode('C'));
        NeighborsOfNodeD.add(new neighborNode('E'));
        NeighborsOfNodeD.add(new neighborNode('F'));
        D.setNeighbors(NeighborsOfNodeD);

        GraphNode E = new GraphNode('E');
        List NeighborsOfNodeE = new ArrayList();
        NeighborsOfNodeE.add(new neighborNode('C'));
        NeighborsOfNodeE.add(new neighborNode('D'));
        E.setNeighbors(NeighborsOfNodeE);

        GraphNode F = new GraphNode('F');
        List NeighborsOfNodeF = new ArrayList();
        NeighborsOfNodeF.add(new neighborNode('D'));
        F.setNeighbors(NeighborsOfNodeF);

        graph.add(A);
        graph.add(B);
        graph.add(C);
        graph.add(D);
        graph.add(E);
        graph.add(F);

        return graph;
    }

    // 图的构造（不带权）
    public List<GraphNode> initGraphWithValue() {
        // 全局图（加权）
        List<GraphNode> graphWithValue = new ArrayList<>();

        GraphNode A = new GraphNode('A');
        List NeighborsOfNodeA = new ArrayList();
        NeighborsOfNodeA.add(new neighborNodeWithDis('B',5));
        NeighborsOfNodeA.add(new neighborNodeWithDis('C',1));
        A.setNeighbors(NeighborsOfNodeA);

        GraphNode B = new GraphNode('B');
        List NeighborsOfNodeB = new ArrayList();
        NeighborsOfNodeB.add(new neighborNodeWithDis('A',5));
        NeighborsOfNodeB.add(new neighborNodeWithDis('C',2));
        NeighborsOfNodeB.add(new neighborNodeWithDis('D',1));
        B.setNeighbors(NeighborsOfNodeB);

        GraphNode C = new GraphNode('C');
        List NeighborsOfNodeC = new ArrayList();
        NeighborsOfNodeC.add(new neighborNodeWithDis('A',1));
        NeighborsOfNodeC.add(new neighborNodeWithDis('B',2));
        NeighborsOfNodeC.add(new neighborNodeWithDis('D',4));
        NeighborsOfNodeC.add(new neighborNodeWithDis('E',8));
        C.setNeighbors(NeighborsOfNodeC);

        GraphNode D = new GraphNode('D');
        List NeighborsOfNodeD = new ArrayList();
        NeighborsOfNodeD.add(new neighborNodeWithDis('B',1));
        NeighborsOfNodeD.add(new neighborNodeWithDis('C',4));
        NeighborsOfNodeD.add(new neighborNodeWithDis('E',3));
        NeighborsOfNodeD.add(new neighborNodeWithDis('F',6));
        D.setNeighbors(NeighborsOfNodeD);

        GraphNode E = new GraphNode('E');
        List NeighborsOfNodeE = new ArrayList();
        NeighborsOfNodeE.add(new neighborNodeWithDis('C',8));
        NeighborsOfNodeE.add(new neighborNodeWithDis('D',3));
        E.setNeighbors(NeighborsOfNodeE);

        GraphNode F = new GraphNode('F');
        List NeighborsOfNodeF = new ArrayList();
        NeighborsOfNodeF.add(new neighborNodeWithDis('D',6));
        F.setNeighbors(NeighborsOfNodeF);

        graphWithValue.add(A);
        graphWithValue.add(B);
        graphWithValue.add(C);
        graphWithValue.add(D);
```

```
        graphWithValue.add(E);
        graphWithValue.add(F);

        return graphWithValue;
    }
```

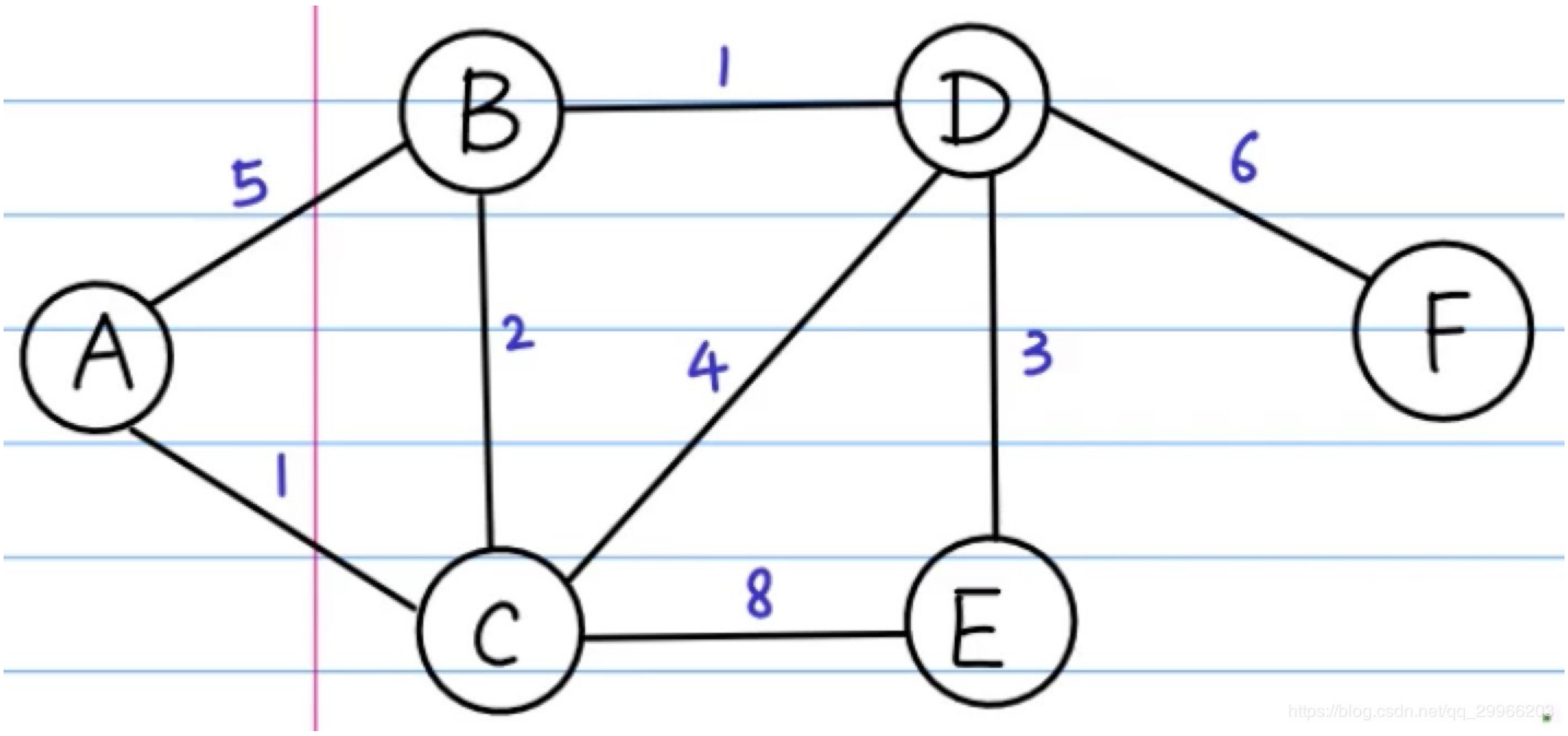
不带权图



字典表示

```
graph = {
      "A": ["B", "C"],
      "B": ["A", "C", "D"],
      "C": ["A", "B", "D", "E"],
      "D": ["B", "C", "E", "F"],
      "E": ["C", "D"],
      "F": ["D"]
}
```


带权图



字典表示

```
graph = {
      "A": {"B": 5, "C": 1},
      "B": {"A": 5, "C": 2, "D": 1},
      "C": {"A": 1, "B": 2, "D": 4, "E": 8},
      "D": {"B": 1, "C": 4, "E": 3, "F": 6},
      "E": {"C": 8, "D": 3},
      "F": {"D": 6}
}
```


辅助方法

```java
public GraphNode findGraphNodeByKey(List<GraphNode> graph,char key) {
    // 根据图的 key 找到图结点
    for(int i = 0;i<graph.size();i++)
        if(graph.get(i).key == key)
            return graph.get(i);
    return null;
}
```

BFS

BFS 基本算法

```java
//例1 BFS
public List BFS(List graph, GraphNode start){
    // 利用BFS遍历图 graph，从任意结点 start 出发
    // 通过队列存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表
    // 特别的，对于队列而言，BFS的结果就是队列的层序遍历

    List result = new ArrayList<>(); // 存储遍历结果列表
    Queue<GraphNode> queue = new LinkedList<>(); // 通过队列存储中间处理数据
    //初始化
    queue.offer(start); // 初始结点加入队列
    result.add(start.key); // 对初始结点进行访问

    while(!queue.isEmpty()) {
        // 队列不为空时，弹出队首结点，寻找该结点的邻接结点
        GraphNode head = queue.poll();

        List<neighborNode> neighbors = head.neighbors; // 获取该结点的邻结点
        for(int i=0;i<neighbors.size();i++) {
            // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
            char neighborValue = neighbors.get(i).key;
            if(!result.contains(neighborValue)) { // 结果列表中不包含该结点，则对该结点进行访问
                queue.offer(findGraphNodeByKey(graph,neighborValue)); // 将该结点加入队列
                result.add(neighborValue); // 已访问的结点加入结果列表
            }
        }
    }

    return result;
}
```

BFS 求单源非加权最短路径

```java
//例2 BFS求 单源非加权最短路径
public HashMap ShortestPathWithBFS(List graph, GraphNode start){
    // 利用BFS遍历图 graph，从任意结点 start 出发
    // 通过队列存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表
    // 遍历结点时，用parent 数组存储该结点的上一个结点。最终可以通过parent结点找到从根节点到各个结点的路径（最短路径）
    // 返回值为一个HashMap 记录每一个结点的 key 以及最短路径情况下到达每一个结点的上一个结点的 key

    List seen = new ArrayList<>(); // 存储遍历结果列表，即已经访问过的结点
    Queue<GraphNode> queue = new LinkedList<>(); // 通过优先队列（默认实现小顶堆），存储中间处理数据
    HashMap parent = new HashMap<>(); // parent数组，存储最短路径下遍历到当前结点和该结点的上一个结点，通过parent数组可以通过溯根找到最短路径
    //初始化
    queue.offer(start); // 初始结点加入队列
    seen.add(start.key); // 访问初始结点
    parent.put(start.key, null); // 初始结点的上一个结点为null

    while(!queue.isEmpty()) {
        // 队列不为空时，弹出队首结点，访问该结点的邻接结点
        GraphNode head = queue.poll();

        List<neighborNode> neighbors = head.neighbors; // 获取该结点的邻结点
        for(int i=0;i<neighbors.size();i++) {
            // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
            char neighborValue = neighbors.get(i).key;
            if(!seen.contains(neighborValue)) { // 结果列表中不包含该结点
                queue.offer(findGraphNodeByKey(graph,neighborValue)); // 将该结点加入队列
                seen.add(neighborValue); // 访问该结点
                parent.put(neighborValue,head.key); // 存储该结点的值 与 上一个结点的值
            }
        }
    }

    return parent;
}
```

DFS

```
//例3 DFS
public List<Integer> DFS(List graph, GraphNode start){
    // 利用DFS遍历图 graph，从任意结点 start 出发
    // 通过栈存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表
    // 本质就是将BFS中的队列用栈表示

    List result = new ArrayList<>(); // 存储遍历结果列表
    Stack<GraphNode> stack = new Stack<>(); // 通过队列存储中间处理数据
    //初始化
    stack.push(start); // 初始结点加入栈
    result.add(start.key); // 对初始结点进行访问

    while(!stack.isEmpty()) {
        // 队列不为空时，弹出队首结点，寻找该结点的邻接结点
        GraphNode head = stack.pop();

        List<neighborNode> neighbors = head.neighbors; // 获取该结点的邻结点
        for(int i=0;i<neighbors.size();i++) {
            // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
            char neighborValue = neighbors.get(i).key;
            if(!result.contains(neighborValue)) { // 结果列表中不包含该结点，则对该结点进行访问
                stack.push(findGraphNodeByKey(graph,neighborValue)); // 将该结点加入队列
                result.add(neighborValue); // 已访问的结点加入结果列表
            }
        }
    }

    return result;
}
```

单源加权最短路径（Dijkstra）

```
    // 自定义优先队列比较器
    // 对图中结点距初始结点的距离进行排序
    Comparator<neighborNodeWithDis> cmp = new Comparator<neighborNodeWithDis>() {
        public int compare(neighborNodeWithDis node1, neighborNodeWithDis node2) {
            //升序排序
            return node1.dis - node2.dis;
        }
    };

    void initParent(HashMap parent) {
            parent.put('A', null);
            parent.put('B', null);
            parent.put('C', null);
            parent.put('D', null);
            parent.put('E', null);
            parent.put('F', null);
    }

    void initDistance(HashMap distance){
            distance.put('A', 0);
            distance.put('B', 0);
            distance.put('C', 0);
            distance.put('D', 0);
            distance.put('E', 0);
            distance.put('F', 0);
    }

    // 例4 基于BFS求 单源加权最短路径，用Dijkstra算法
    public void Dijkstra(List graph, GraphNode start){
        // 利用Dijkstra 算法求最短路径。基于BFS算法。
        // 利用 result 记录遍历过的结点
        // 利用 parent 记录最短路径时遍历结点的上一个结点
        // 利用 distance 记录最短路径时从 start 到达各个结点时的最短路径

        List result = new ArrayList<>(); // 存储已经处理结束的结点
        // 通过优先队列存储中间处理数据
        // 优先队列中存储着结点的key 及 当前距离初始结点的距离
        Queue<neighborNodeWithDis> queue = new PriorityQueue<>(cmp);
        HashMap parent = new HashMap<>(); // parent数组，存储最短路径下遍历到当前结点和该结点的上一个结点，通过parent数组可以通过溯根找到最短路径（实时更新）
        HashMap distance = new HashMap<>(); // distance数组，存储最短路径下从根节点遍历到当前结点和该结点的最短路径的距离（实时更新）

        //初始化
        queue.offer(new neighborNodeWithDis(start.key,0)); // 初始结点加入队列
        initParent(parent); //初始化父结点数组
        initDistance(distance); //初始化距离数组

        while(!queue.isEmpty()) {
            // 队列不为空时，弹出队首结点，对该结点进行访问
            neighborNodeWithDis headNode = queue.poll();
            GraphNode head = findGraphNodeByKey(graph,headNode.key);
            result.add(head.key); // 弹出队列，表示该结点已经处理完毕，加入result数组
            int currentDis = headNode.dis;

            List<neighborNodeWithDis> neighbors = head.neighbors; // 获取该结点的邻结点
            for(int i=0;i<neighbors.size();i++) {
                // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
                neighborNodeWithDis neighborNode = neighbors.get(i);
                if(!result.contains(neighborNode.key)) { // 结果列表中不包含该结点
                    int dis = neighborNode.dis; // 获取邻结点与当前结点的距离
                    char key = neighborNode.key; // 获取邻结点的key
                    if(currentDis+dis < (Integer)distance.get(key) || (Integer)distance.get(key)==0) {
                        // 若当前遍历下到该邻接点的距离梗系啊
                        queue.offer(new neighborNodeWithDis(key,currentDis+dis)); // 将该结点加入队列
                        parent.replace(key, head.key); // 存储该结点的值 与 上一个结点的值
                        distance.replace(key,currentDis+dis); // 存储到初始结点更短的距离
                    }
                }
            }
        }
```

```
70
71          System.out.print("A" + " | " + parent.get('A')  + " | " + distance.get('A') + "\n");
72          System.out.print("B" + " | " + parent.get('B')  + " | " + distance.get('B') + "\n");
73          System.out.print("C" + " | " + parent.get('C')  + " | " + distance.get('C') + "\n");
74          System.out.print("D" + " | " + parent.get('D')  + " | " + distance.get('D') + "\n");
75          System.out.print("E" + " | " + parent.get('E')  + " | " + distance.get('E') + "\n");
76          System.out.print("F" + " | " + parent.get('F')  + " | " + distance.get('F') + "\n");
77      }
```

打印结果

```
A | null | 0
B | C | 3
C | A | 1
D | B | 4
E | D | 7
F | D | 10
```

leetcoe

例1：路径之和（113）

题目描述

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

```
给定如下二叉树，以及目标和 sum = 22，

              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
返回:
[
   [5,4,11,2],
   [5,8,4,5]
]
```

解题思路

1. 从根节点深度遍历二叉树，先序遍历时，将该结点值存储至path栈中，使用path_value累加节点值。
2. 当遍历至叶结点时，检查path_value值是否为sum，若为sum，则将path push 进入result结果中。
3. 在后续遍历时，将该节点值从path栈中弹出，path_value减去节点值。

程序代码

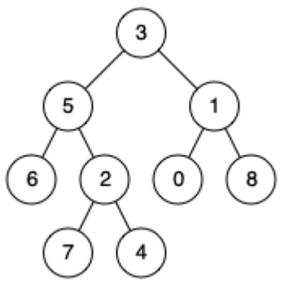
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
   public List<List<Integer>> pathSum(TreeNode root, int sum) {
        // 采用先序遍历，遍历到叶子节点时进行判断
            // 满足条件则加入结果列表中
            List<List<Integer>> result = new ArrayList<>();//结果数组
            List<Integer> path = new ArrayList<>();//某条路径数组
            int path_sum = 0;//某条路径的目标和

            balanceAllPathSum(root,sum,result,path,path_sum);

            return result;
    }

    public void balanceAllPathSum(TreeNode root,int sum,List<List<Integer>> result,List<Integer> path,int path_sum) {
            // 进行遍历时计算路径和，如果等于目标和，则加入结果数组中。
            // 遍历过程中访问的结点均加入列表中，构成节点路径
            if(root == null) return;//如果是空节点，则直接返回
            // 否则对结点进行处理，添加到路径与路径综合
            path.add(root.val);
            path_sum += root.val;
            if(root.left == null && root.right == null && path_sum == sum) {
                // 如果为叶子结点，且此时路径和 = 目标和，则加入结果数组
                result.add(new ArrayList<>(path));
            }
            // 先序遍历
            balanceAllPathSum(root.left,sum,result,path,path_sum);// 遍历左子树
            balanceAllPathSum(root.right,sum,result,path,path_sum);// 遍历右子树

            // 恢复现场（路径与路径和）
            path.remove(path.size()-1);
            path_sum -= root.val;
    }
}
```

例2：最近的公共祖先（236）

题目描述

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”
例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。
```

示例 2:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

说明:
所有节点的值都是唯一的。
p、q 为不同节点且均存在于给定的二叉树中。

解题思路

（1）算法思路：求p结点路径、q节点路径，两个路径上最后一个相同结点。
（2）求根节点至某节点路径（深度搜索）

1. 从根节点遍历（搜索）至该结点，找到该结点后结束搜索。
2. 将遍历过程中遇到的结点按顺序存储，这些结点即路径结点

程序代码

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // 本题的解题思路为：对结点p,q分别求根节点到这两个结点的结点路径
            // 通过list的形式来保存结点路径
            // 从遍历列表元素，找到的最后一个相同的元素即为最近公共结点
            if(root == null)return root;

            List<TreeNode> list_P = new ArrayList<>();// 存储从根节点到结点P的结点路径
            List<TreeNode> list_Q = new ArrayList<>();// 存储从根节点到结点Q的结点路径

            List<List<TreeNode>> list_temp = new ArrayList<>(); // 存储中间路径的结果
            getPathFromRootToNode(root,p,list_temp,new ArrayList(),0);
            getPathFromRootToNode(root,q,list_temp,new ArrayList(),0);
            list_P = list_temp.get(0);
            list_Q = list_temp.get(1);

            Integer min_length = min(list_P.size(),list_Q.size());//p,q结点路径中较小值
            TreeNode result = root;
            for(int i=0;i<min_length;i++)
            {
                // 从根节点遍历p,q的结点路径
                // 最后一个相同结点即为最近公共结点
                if(list_P.get(i).val == list_Q.get(i).val)
                result = list_P.get(i);
            }

            return result;
    }

    public void getPathFromRootToNode(TreeNode root,TreeNode search,List result,List path,Integer finish) {
            // 返回一条结点路径
            // 该路径由根到结点search
            //当结点为空 或 已经找到结点路径时（结束遍历标志符 = 1），便不再继续遍历
            if(root == null || finish == 1)return;
            path.add(root);
            if(root == search) {
                // 如果遍历到该结点，则停止遍历
                finish = 1;// 结束遍历标志符为1
                result.add(new ArrayList<>(path));
                return;
            }
            // 否则继续遍历
            getPathFromRootToNode(root.left,search,result,path,finish);//遍历左子树
            getPathFromRootToNode(root.right,search,result,path,finish);//遍历右子树
            path.remove(path.size()-1);//弹出结点，恢复现场
    }

    int min(int a,int b) {
            return a>=b?b:a;
    }
}
```

例3：二叉树转链表（114）

题目描述

给定一个二叉树，原地将它展开为链表。链表按照先序遍历的顺序，且结构为TreeNode。

```
例如，给定二叉树

    1
   / \
  2   5
 / \   \
3   4   6
将其展开为：

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

解题思路

先序遍历二叉树，将结点指针加入列表，顺序遍历列表，将前面的结点的做指针置空，右指针与后面的结点相连。

程序代码

```
public void flatten(TreeNode root) {
     // 1.利用中间列表存储先序遍历的结果
         // 2.将先序遍历的列表转化为对应链表

         List<TreeNode> preorderList = new ArrayList<>();
         // 将当前树结构按先序顺序存储到链表中
         transFromTreeToLinkedList(root,preorderList);

         for(int i=0;i<=preorderList.size()-2;i++) {
             // 按顺序遍历链表，将当前结点的做指针置空，有指针与下一个结点相连（构造链表结构）
             // 实现原树就地转换为一个单链表
             TreeNode current = preorderList.get(i);
             TreeNode next = preorderList.get(i+1);
             current.left = null;
             current.right = next;
         }
 }

 public void transFromTreeToLinkedList(TreeNode currentNode,List preorderList) {
         //将树按 先序遍历 顺序加入到列表中
         if(currentNode == null)return ;
         preorderList.add(currentNode);//将当前结点加入到列表中
         transFromTreeToLinkedList(currentNode.left,preorderList);//遍历左子树
         transFromTreeToLinkedList(currentNode.right,preorderList);//遍历右子树
 }
```

例4：侧面观察二叉树（199）

题目描述

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

```
输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:
   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---

```

解题思路

层序遍历时，将节点与层数绑定为pair，压入队列时，将节点与层数同时压入队列，并记录每一层中出现的最后一个节点。

在层次遍历中，每一层中的最后一个节点最后遍历到，随时更新对每层的最后一个结点即可。

补充：层序遍历Java解法

```
public class Solution {
    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        ArrayList<Integer> L1=new ArrayList<Integer>();
        Deque<TreeNode> q = new LinkedList<TreeNode>();
// offer()方法是Queue方法，插入到队尾，不成功就返回false
        q.offer(root);
        while(q.peek()!=null){
// peek()和element()都在不移除的情况下返回队列的头部，peek()在队列为空时返回null,element()
// 则会抛出NoSuchElementExcetion异常。
           TreeNode tmp=q.poll();
// poll()和remove()方法移除队列头部，poll（）在队列为空的时候返回null,remove返回
// NoSuchElementException异常
            L1.add(tmp.val);
            if(tmp.left!=null)q.offer(tmp.left);
            if(tmp.right!=null)q.offer(tmp.right);
        }
        return L1;
    }
}
```

程序代码

```
public List<Integer> rightSideView(TreeNode root) {
    // 对二叉树采用层序遍历，每遍历到一个节点，将该结点的 值 和对应的层数保存到中间列表
        // 顺序遍历中间列表，则每个层数的最后一个结点即为最右侧节点。
        // 中间链表，存储层序遍历结点结果
        List<LevelNode> levelList = new ArrayList<>();
        List<Integer> result = new ArrayList<>();
        //若树为空，则返回空结果
        if(root == null) return result;
        // 将树结构层序遍历结果存储列表中
        levelTraversal(root,levelList);
        // 如果只有根，则直接返回根节点
        if(levelList.size()==1) {
            result.add(root.val);
            return result;
        }
        // 如果树中有多个结点，则进行遍历
        LevelNode lastNode;
        LevelNode currentNode;
        for(int i=1;i<levelList.size();i++) {
            // 遍历层序遍历结果，取出每一层最后一个元素（level相同的最后一个结点）加入结果数组
            lastNode = levelList.get(i-1);
            currentNode = levelList.get(i);
            if( lastNode.level != currentNode.level) {
                // 当前结点与上一个结点属于不同层，则说明上一个结点是该层的最后一个结点
                // 加入结果数组中
                    result.add(lastNode.treeNode.val);
            }
            // 最后一个结点则直接加入结果数组中
            if(i == levelList.size()-1)result.add(currentNode.treeNode.val);
        }

        return result;
}

public void levelTraversal(TreeNode root,List levelList) {
        // 层序遍历，存储层序遍历结果
    // 利用队列进行层序遍历
    Deque<LevelNode> queue = new LinkedList<LevelNode>();
    queue.offer(new LevelNode(0,root));
    //加入根节点

    while(queue.peek()!=null) {
        // 加入队首元素
        LevelNode currentNode = queue.poll();
        levelList.add(currentNode);

        if(currentNode.treeNode.left!=null)queue.offer(new LevelNode(currentNode.level+1,currentNode.treeNode.left));
        if(currentNode.treeNode.right!=null)queue.offer(new LevelNode(currentNode.level+1,currentNode.treeNode.right));
    }
}

public class LevelNode{
        int level;// 层数
        TreeNode treeNode;// 树结点

        public LevelNode(int level,TreeNode treeNode) {
            this.level = level;
            this.treeNode = treeNode;
        }
}
```

例5：课程安排（有向图判断环）（207）

题目描述

现在你总共有 n 门课需要选，记为 0 到 n-1。
在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]
给定课程总量以及它们的先决条件，判断是否可能完成所有课程的学习？

```
示例 1:
输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。
示例 2:
输入: 2, [[1,0],[0,1]]
输出: false
解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。
```

说明:
输入的先决条件是由边缘列表表示的图形，而不是邻接矩阵。详情请参见图的表示法。
你可以假定输入的先决条件中没有重复的边。
提示:
这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。

解题思路

n个课程，它们之间有m个依赖关系，可以看成顶点个数为n，边个数为m的有向图。
故，若有向图无环，则可以完成全部课程；否则不能，问题转换成构建图，并判断图是否有环。可以用两种方法判断有向图是否有环。

方法1：深度优先搜索

在深度优先搜索时，如果正在搜索某一顶点（还未退出该顶点的递归深度搜索），又回到了该顶点，即证明图有环。

方法2：拓扑排序（宽度优先搜索）

在宽度优先搜索时，只将入度为0的点添加至队列，当完成一个顶点的搜索（从队列取出），他指向的所有顶点入度都减1，若此时某顶点入度为0则添加至队列，若完成宽度搜索后，所有点入度都为0，则图无环，否则有环。

程序代码

```
    public class GraphNode{
        int label;
        List<GraphNode> neighbors;
```

```
        GraphNode(int label){
            this.label = label;
            this.neighbors = new ArrayList<>();
        }
    }

//  207.课程表
//  现在你总共有 n 门课需要选，记为 0 到 n-1。
//  在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]
//  给定课程总量以及它们的先决条件，判断是否可能完成所有课程的学习？
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // 通过拓扑排序（BFS）判断是否为有向无环图，若无环则可以完成所有课程；否则不能。
            // 在宽度优先搜索时，只将入度为0的点添加至队列。当完成一个顶点的搜索（从队列中取出），
            // 它指向的所有顶点入度减1，若此时某顶点入度为0则添加至队列，
            // 若完成宽度搜索后，所有点的入度都为0，则图无环，否则有环。

            List<GraphNode> graph = new ArrayList<>(); // 构造图的邻接表
            int[] degree = new int[numCourses]; // 存储图结点的度

            // 构造有向图的邻接表
            for(int i=0;i<numCourses;i++) { // 初始化有向图结构，这里结点的label与结点的下标对应
                graph.add(new GraphNode(i));
                degree[i] = 0;
            }
            // 根据课程的依赖关系，为图添加邻接关系和度
            for(int i=0;i<prerequisites.length;i++) {
                // prerequisites中结点:[1,0]表示学习课程1之前，需要学习课程0，因此1是依赖0的。
                // 因此有向图中结点0指向结点1。
                GraphNode start = graph.get(prerequisites[i][1]);
                GraphNode end = graph.get(prerequisites[i][0]);
                start.neighbors.add(end); // 首节点指向尾结点
                degree[end.label]++; // 尾结点入度+1
            }

            // BFS 进行拓扑排序
            // step1:将入度为0的结点加入队列中
            Queue<GraphNode> queue = new LinkedList<>();
            for(int i=0;i<numCourses;i++) {
                if(degree[i] == 0)queue.offer(graph.get(i));
            }

            // step2: 进行宽度优先遍历
            while(!queue.isEmpty()) {
                // step3: 完成一个结点的搜索，从队头取出
                GraphNode head = queue.poll();
                List<GraphNode> neighbors = head.neighbors;
                for(int i=0;i<neighbors.size();i++) {
                    // step4: 将以搜索完毕的结点为首的尾结点入度-1，若该结点的入度为0，则加入队列中
                    GraphNode neighbor = neighbors.get(i);
                    degree[neighbor.label]--;
                    if(degree[neighbor.label]==0)
                        queue.offer(neighbor);
                }
            }

            //step5: 遍历结束，所有点的入度都为0，则图无环，否则有环。
            for(int i=0;i<numCourses;i++)
                if(degree[i]!=0)return false;

            return true;
    }
```

剑指offer

例1：重建二叉树（4）

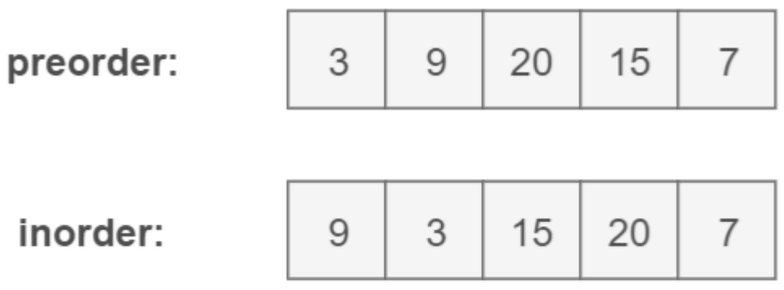
题目描述

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

算法思路

前序遍历的第一个值为根节点的值，使用这个值将中序遍历结果分成两部分，左部分为树的左子树中序遍历结果，右部分为树的右子树中序遍历的结果。

①



CyC2018

程序代码

```
public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        if(pre == null || in == null || pre.length == 0 || in.length == 0 || pre.length != in.length)return null;
        TreeNode root = ConstructBinaryTreeByPreAndInOrder(pre ,in ,0 ,pre.length-1 ,0 ,in.length-1 );
        return root;
}

public TreeNode ConstructBinaryTreeByPreAndInOrder(int[] pre,int[] in, int pre_start, int pre_end, int in_start, int in_end) {
        // 根据 pre[pre_start...pre_end] 与 in[in_start...in_end] 构造 二叉树
        if(pre_start > pre_end || in_start > in_end)return null;
        int node = pre[pre_start];  // 当前结点
        // 1. 构造根节点（先序序列的头结点构造根节点）
        TreeNode root_node = new TreeNode(node);
        // 找到中序遍历的中间结点in_idx
        int in_idx = in_start;
        for(int i=in_start;i<=in_end;i++)
            if(in[i] == node)in_idx = i;
        // 2. 构造左子树
        // 用中序结点中间结点左边的元素 in[in_start...in_idx-1] 及 先序结点对应元素 构造左子树
        root_node.left = ConstructBinaryTreeByPreAndInOrder(pre, in, pre_start+1, pre_start+in_idx-in_start, in_start,in_idx-1);
        // 3. 构造右子树
        // 用中序结点中间结点右边的元素 in[in_idx+1...in_end] 及 先序结点对应元素 构造右子树
        root_node.right = ConstructBinaryTreeByPreAndInOrder(pre, in, pre_end-in_end+in_idx+1, pre_end, in_idx + 1, in_end);

        return root_node;
    }
```

例2：树的子结构（17）

题目描述

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

程序代码

```
// 16.树的子结构
// 输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）
public boolean HasSubtree(TreeNode root1,TreeNode root2) {
    // 对 root1 先序遍历,
        // 若root1 的当前节点的值 = root2头结点，则说明当前root1 结构可能包含root2 -> 判断root1 是否包含root2
        // 判断 root1的左子树 是否包含root2
        // 判断 root1的右子树 是否包含root2
        boolean hasSubTree = false;
        if(root2 == null ) return false;    // 空树不是任意一个树的子结构
        if(root1 == null )return false;
    if(root1.val == root2.val)
            if(isRoot1ContainsRoot2(root1,root2))hasSubTree = true;
    if(HasSubtree(root1.left,root2))hasSubTree = true;
    if(HasSubtree(root1.right,root2))hasSubTree = true;

    return hasSubTree;
}

public boolean isRoot1ContainsRoot2(TreeNode root1,TreeNode root2) {
        // 判断 root1 是否包含 root2
        // 如果 root2 所有节点遍历结束且正确，返回 true
        // 如果 root2 不为空，则若root1 与 root2 均不为空且相同，则判断 root1左右子树是否包含 root2左右子树。若左右子树也包含，则返回 true
        // 否则不包含
        if(root2 == null) return true;
        if(root1!=null && root2!=null && root1.val == root2.val) {
            return isRoot1ContainsRoot2(root1.left,root2.left) && isRoot1ContainsRoot2(root1.right,root2.right);
        }
        return false;
}
```

例3：二叉树的镜像（18）

题目描述

操作给定的二叉树，将其变换为源二叉树的镜像。

程序代码

```
    // 17.二叉树的镜像
    // 操作给定的二叉树，将其变换为源二叉树的镜像。
    public void Mirror(TreeNode root) {
        // 先序遍历二叉树root
            // 若root有左右子树，则颠倒左右子树
            // 采用后序遍历，防止叶子节点颠倒对遍历的影响（重复颠倒）
            if(root == null)return;
            if(root.left==null && root.right == null)return ;   //递归退出条件：该节点为叶子节点
            Mirror(root.left);
            Mirror(root.right);
            // 颠倒左右子树
            TreeNode temp = root.left;
            root.left = root.right;
            root.right = temp;
    }
```

例4：从上往下打印二叉树（22）

题目描述

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

程序代码

```
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
          ArrayList<Integer> resList = new ArrayList<Integer>();
          //实现层序遍历
          Queue queue = new LinkedList<TreeNode>();
          if(root!=null)
              queue.offer(root);
          while(!queue.isEmpty()) {
              TreeNode node = (TreeNode)queue.poll();
              resList.add(node.val);

              if(node.left!=null)queue.offer(node.left);
              if(node.right!=null)queue.offer(node.right);
          }
          return resList;
    }
```

例5：二叉树中和为某一值的路径（24）

题目描述

输入一颗二叉树的根节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)

程序代码

```
// 24. 二叉树中和为某一值的路径
//输入一颗二叉树的跟节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。
//路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)
public ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {
    // 对二叉树进行DFS
        if(root != null)
        isPathSatisfy(root,target,0,new ArrayList<Integer>());  // DFS遍历二叉树
        Collections.sort(result, listComparator);               // 路径集合对路径的长度按降序排序
        return result;
}

public void isPathSatisfy(TreeNode node, int target, int pathSum, ArrayList<Integer> pathList) {
        // DFS遍历二叉树
        // 若遍历到叶子节点且满足此时路径总和 == target，则加入结果序列
        // 否则继续遍历左右子树
        // * 每次遍历对现场处理 => 该状态节点遍历结束后需弹栈，恢复状态前现场
        if(node == null)return;
        pathSum += node.val;
        pathList.add(node.val);
        if(node.left == null && node.right == null && pathSum == target) {result.add(pathList);return;}
        if(node.left != null)isPathSatisfy(node.left, target, pathSum,new ArrayList<Integer>(pathList));
        if(node.right != null)isPathSatisfy(node.right, target, pathSum,new ArrayList<Integer>(pathList));

        pathSum -= node.val;
        pathList.remove(pathList.size()-1);
}

public static Comparator listComparator = new Comparator() {
    // 比较器，ArrayList中的路径按路径长度排序
    @Override
    public int compare(Object o1, Object o2) {
        // TODO Auto-generated method stub
        return ((ArrayList)o2).size() - ((ArrayList)o1).size();
    }
     };
```

例6：二叉搜索树与双向链表（26）

题目描述

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

程序代码

```
// 26.二叉搜索树与双向链表
// 输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。
public TreeNode lastHandleNode; // 上一次中序遍历处理的节点，即当前节点的左节点
public TreeNode convertHead;        // 转换链表的头结点
public TreeNode Convert(TreeNode pRootOfTree) {
    // 二叉搜索树中序遍历顺序即按升序排列的列表，按中序遍历依此连接双项链表
        if(pRootOfTree != null) {
            lastHandleNode = null;
            convertHead = null;
            inOrderTree(pRootOfTree);
```

```
        }
        return convertHead;
    }

    public void inOrderTree(TreeNode node) {                        // 中序遍历
        if(node != null) {
            if(node.left!=null)inOrderTree(node.left);
            if(convertHead == null) convertHead = node;             // 中序遍历的第一个节点即转换链表的头结点
            if(lastHandleNode!=null) lastHandleNode.right = node; // 上一次中序遍历节点的右结点为当前节点
            node.left = lastHandleNode;                             // 当前节点的左节点为上一次中序遍历节点
            lastHandleNode = node;
            if(node.right!=null)inOrderTree(node.right);
        }
    }
```

例7：二叉树的深度（37）

题目描述

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

程序代码

```
// 37.二叉树的深度
// 输入一棵二叉树，求该树的深度。
// 从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。
Integer maxDepth = 0;
public int TreeDepth(TreeNode root) {
        if(root!=null) getLongestPathOfRoot(root,1);
     return maxDepth;
}

public void getLongestPathOfRoot(TreeNode node,int depth) {
    // 深度遍历二叉树，遍历到叶子节点时计算路径长度
    // 若路径长度大于最大路径长度，则记录
        if(node == null)return;
        else {
            if(node.left == null && node.right == null) {
                maxDepth = depth>maxDepth?depth:maxDepth;
                return;
                }
            else {
                    if(node.left!=null)getLongestPathOfRoot(node.left,depth+1);
                    if(node.right!=null)getLongestPathOfRoot(node.right,depth+1);
                }
        }
}
```

例8：平衡二叉树（38）

题目描述

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

程序代码

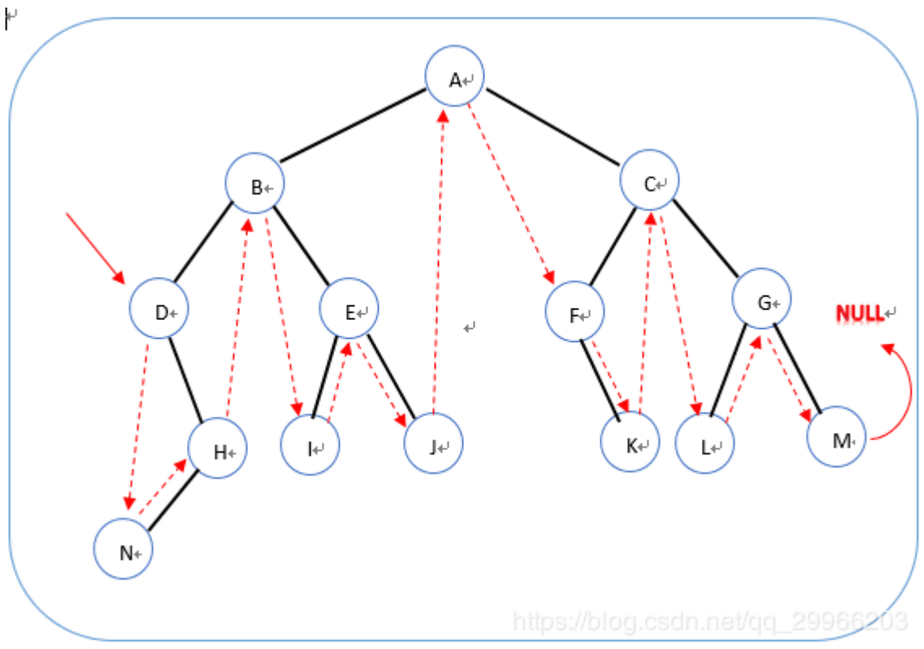
```
// 38.平衡二叉树
// 输入一棵二叉树，判断二叉树是否是平衡二叉树
public boolean IsBalanced_Solution(TreeNode root) {
    // 二叉排序树树满足IsSearch(root)
    // 平衡二叉树满足IsBalanced(root)
        if(IsSearchTree(root) && IsBalanced(root))return true;
        return false;
}

public boolean IsSearchTree(TreeNode root) {
    // 二叉排序树特性:
    //  1. 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
    //  2. 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
    //  3. 它的所有结点的左右子树也分别为二叉排序树。
        if(root!=null) {
        if((root.left!=null && root.left.val>=root.val) || (root.right!=null && root.right.val<=root.val))
            return false;

        if(root.left != null)if(!IsBalanced_Solution(root.left))return false;
        if(root.right != null)if(!IsBalanced_Solution(root.right))return false;
    }
    return true;
}

public boolean IsBalanced(TreeNode root) {
    // 高度差满足:
    // 每个结点的左子树和右子树的高度差至多等于1
        if(root!=null) {
            if(Math.abs(getHeight(root.left)-getHeight(root.right))>1)return false;
            if(root.left!=null)if(!IsBalanced(root.left))return false;
            if(root.right!=null)if(!IsBalanced(root.right))return false;
        }
        return true;
}

public int getHeight(TreeNode root) {
        if(root!=null) {
            int left = root.left==null?0:getHeight(root.left);
            int right = root.right==null?0:getHeight(root.right);

            return left>right?left+1:right+1;
        }else return 0;
}
```

例9：二叉树的下一个节点（56）

题目描述

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

解题思路



结合图，我们可发现分成两大类：1、有右子树的，那么下个结点就是右子树最左边的点；（eg：D，B，E，A，C，G）2、没有右子树的，也可以分成两类，a)是父节点左孩子（eg：N，I，L），那么父节点就是下一个节点 ； b)是父节点的右孩子（eg：H，J，K，M）找他的父节点的父节点的父节点...直到当前结点是其父节点的左孩子位置。如果没有eg：M，那么他就是尾节点。

程序代码

```
 // 56. 二叉树的下一个节点
// 给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。
// 注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。
public class TreeLinkNode {
    int val;
    TreeLinkNode left = null;
    TreeLinkNode right = null;
    TreeLinkNode next = null;

    TreeLinkNode(int val) {
        this.val = val;
    }
}

public TreeLinkNode GetNext(TreeLinkNode pNode)
{
        if(pNode == null)return null;
        if(pNode.right!=null)return getLeftLeaf(pNode.right);
        else if(pNode.left!=null)return pNode.next;
        else {
            TreeLinkNode parent = pNode.next;
            while(parent != null) {
                if(parent.left == pNode)return parent;
                else {
                    pNode = parent;
                    parent = pNode.next;
                }
            }
            return null;
        }
}

public TreeLinkNode getLeftLeaf(TreeLinkNode node) {
        // 获取节点node的最左节点
        // 1. 如果有左子树，则下一个节点为左子树的最左值
    // 2. 否则，如果有右子树，则下一个节点为右子树的最左值
        while(node.left!=null)node = node.left;
        return node;
}
```

例10：对称的二叉树（57）

题目描述

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

程序代码

```
 // 57. 对称的二叉树
// 请实现一个函数，用来判断一颗二叉树是不是对称的。
// 注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。
boolean isSymmetrical(TreeNode pRoot)
{
    // 从根节点依次向下递归判断
        // 保证左子树的左子树 = 右子树的右子树
        //    左子树的右子树 = 右子树的左子树
        if(pRoot == null)return true;
        else return isNodeSymmetrical(pRoot.left,pRoot.right);
}

public boolean isNodeSymmetrical(TreeNode leftNode,TreeNode rightNode) {
        if(leftNode == null)return rightNode == null;
        if(rightNode == null)return leftNode == null;
        if(leftNode.val != rightNode.val)return false;
        if(!(isNodeSymmetrical(leftNode.left,rightNode.right) && isNodeSymmetrical(leftNode.right,rightNode.left)))return false;
        return true;
}
```

例11：按之字形顺序打印二叉树（58）

题目描述

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

程序代码

```java
// 58.按之字形顺序打印二叉树
 // 请实现一个函数按照之字形打印二叉树，
 // 即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。
 public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        // 1. 层序遍历各个节点，记录当前行的最右节点
        // 2. 若当前遍历节点为最右节点，则换行操作
        // 3. 按之字形顺序打印，故若为偶数行，则打印时需逆序打印
        Queue<TreeNode> queue = new LinkedList<TreeNode>(); // 定义队列，用于层序遍历
        Integer row = 1;        // 记录当前遍历二叉树的层数
        TreeNode lastNodeOfRow = pRoot;     // 记录当前行的最右节点
        TreeNode lastNodeOfNextRow = pRoot;     // 记录下一行可能的最右节点
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>(); // 结果数组
        ArrayList<Integer> rowList = new ArrayList<Integer>();
        if(pRoot == null)return result;
        queue.offer(pRoot);
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            rowList.add(node.val);
            // 每次加入左右节点，均为下一行可能的最右节点（最后一个加入的节点即为最右节点）
            if(node.left!=null) {queue.offer(node.left);lastNodeOfNextRow = node.left;}
            if(node.right!=null) {queue.offer(node.right);lastNodeOfNextRow = node.right;}
            if(node == lastNodeOfRow) {
                // 如果该节点为当前行的最后节点，换行的各种操作
                if(row % 2 == 0) reverseList(rowList);  // 偶数行，则转置
                result.add(rowList);                        // 将该行的结果加入结果集
                row++;                                  // 行数+1，定义新行数
                rowList = new ArrayList<Integer>();
                lastNodeOfRow = lastNodeOfNextRow;      // 换行后该行的最右节点
            }
        }
        return result;
    }

    public void reverseList(List list) {
        for(int i=0;i<list.size()/2;i++)
        {
            Object temp = list.get(i);
            list.set(i, list.get(list.size()-1-i));
            list.set(list.size()-1-i, temp);
        }
    }
```

例12：把二叉树打印成多行（59）

题目描述

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

程序代码

```java
// 59.将二叉树打印成多行
 // 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。
 ArrayList<ArrayList<Integer> > Print2(TreeNode pRoot) {
    // 1. 层序遍历各个节点，记录当前行的最右节点
    // 2. 若当前遍历节点为最右节点，则换行操作
    Queue<TreeNode> queue = new LinkedList<TreeNode>(); // 定义队列，用于层序遍历
    Integer row = 1;        // 记录当前遍历二叉树的层数
    TreeNode lastNodeOfRow = pRoot;     // 记录当前行的最右节点
    TreeNode lastNodeOfNextRow = pRoot;     // 记录下一行可能的最右节点
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>(); // 结果数组
    ArrayList<Integer> rowList = new ArrayList<Integer>();
    if(pRoot == null)return result;
    queue.offer(pRoot);
    while(!queue.isEmpty()) {
        TreeNode node = queue.poll();
        rowList.add(node.val);
        // 每次加入左右节点，均为下一行可能的最右节点（最后一个加入的节点即为最右节点）
        if(node.left!=null) {queue.offer(node.left);lastNodeOfNextRow = node.left;}
        if(node.right!=null) {queue.offer(node.right);lastNodeOfNextRow = node.right;}
        if(node == lastNodeOfRow) {
            // 如果该节点为当前行的最后节点，换行的各种操作
            result.add(rowList);                        // 将该行的结果加入结果集
            row++;                                  // 行数+1，定义新行数
            rowList = new ArrayList<Integer>();
            lastNodeOfRow = lastNodeOfNextRow;      // 换行后该行的最右节点
        }
    }
    return result;
 }
```

例13：序列化二叉树（60）

题目描述

请实现两个函数，分别用来序列化和反序列化二叉树

二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，序列化的结果是一个字符串，序列化时通过 某种符号表示空节点（#），以 ！ 表示一个结点值的结束（value!）。

二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。

程序代码

```java
    // 60. 序列化二叉树
    // 请实现两个函数，分别用来序列化和反序列化二叉树
    // 二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。
    // 序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，
    // 序列化的结果是一个字符串，序列化时通过 某种符号表示空节点（#），以 ！ 表示一个结点值的结束（value!）。
    // 二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。
    int count = 0;
    String Serialize(TreeNode root) {
        // 返回层序序列字符串
        if(root == null)return null;
```

```
        countNodeNumber(root);  // 计算树的节点个数
        String serializeStr = getRowOrderList(root);
        return serializeStr;
    }

    public void countNodeNumber(TreeNode root) {
        if(root!=null) {
            count++;
            countNodeNumber(root.left);
            countNodeNumber(root.right);
        }
    }

    public String getRowOrderList(TreeNode root) {
        if(root == null)return null;
        String serializeStr = "";
        Integer nodeNumber = 0;
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if(node == null){   // 若为空节点，则也加入序列化字符串（构造完全二叉树）
                serializeStr += "#,";
                queue.offer(null);
                queue.offer(null);
            }
            else {
                serializeStr += node.val + ",";
                if(++nodeNumber == count)break; // 遍历到最后一个节点停止
                queue.offer(node.left);
                queue.offer(node.right);
            }
        }
        return serializeStr;
}

    TreeNode Deserialize(String str) {
        if(str == null || str == "")return null;
        String[] strArray = str.split(",");
        TreeNode root = buildTreeByRow(0,strArray);
        return root;
    }

    public TreeNode buildTreeByRow(int i,String[] str) {
        // 构造第i个节点的子树
        if(i>=str.length)return null;
        if(str[i].equals("#")) return null;
        else {
            // 根据完全二叉树节点下标的特点，还原树
            TreeNode node = new TreeNode(Integer.parseInt(str[i]));
            int leftIdx = i*2 +1;
            int rightIdx = i*2 +2;
            if(leftIdx < str.length)node.left = buildTreeByRow(leftIdx,str);
            if(rightIdx < str.length)node.right = buildTreeByRow(rightIdx,str);
            return node;
        }
    }
```