Graphs & Search

Graphs:

207. Course Schedule

Topological Sort

解法一：BFS拓扑排序

使用邻接表保存有向图，并使用一个数组保存所有结点的入度。将所有入度为0的点存入栈中，删除入度为0的点（即删除与其他点连接的边），并更新那些点的入度，将新的入度为0的点放入栈中。用一个数组将访问过的点标记。

最后判断是否所有的点都被访问过了。如果有的点没有被访问到则说明有环路，因为这些环路上的点无法达到入度为0，所以无法访问。

---

787. Cheapest Flights Within K Stops

DFS

解法一：DFS

因为题目给定了起始点和终止点，虽然有限制步数，但是可以用深度优先搜索去搜索这个点，遍历所有搜索路径并保存最佳，如果最后没有在规定数量之内找到就返回-1。

需要注意因为需要遍历所有路径，所以已经访问过的点之后还需要继续访问。一种处理方式是把当前点的visited情况先改成true，结束当前点邻接点的遍历之后再改成false，防止绕回来，但是这样还是会TLE。

这个时候需要剪枝，去掉那些不可能的路径，就是说如果走这条路加上下一条路的长度已经比当前最佳答案长了，那么这条路就不用走了，这样剪枝可以保证时间不超时。

886. Possible Bipartition

BFS; DFS

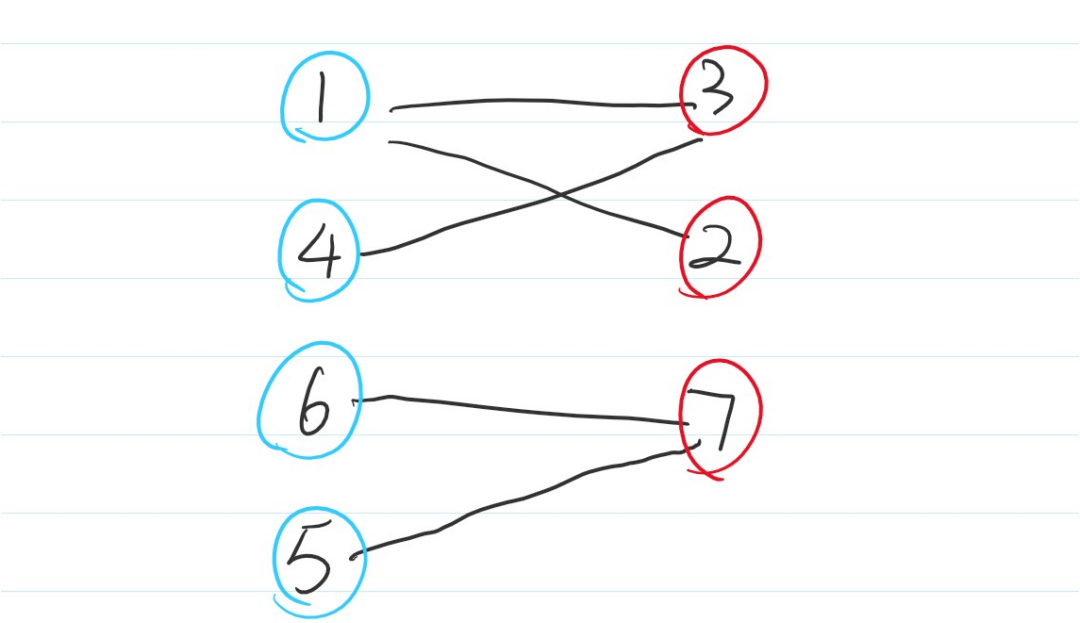
> Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone into two groups of any size.Each person may dislike some other people, and they should not go into the same group.
>
> Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people numbered a and b into the same group.
>
> Return true if and only if it is possible to split everyone into two groups in this way.

解法一：BFS+Coloring判断Bipartite Graph

Bipartite Graph 中的结点可以被分成两个不相交的集合，任意一个集合中的点之间没有边，其中的点只和另一个集合内的点有连接的边。

每两个人之间互相讨厌则视作两个点之间有连线，如果可以被分为两个组则说明这个图是Bipartite Graph。判断是否是Bipartite Graph 则可以对一个点进行上色，然后判断邻近的点是否被上色，如果没有则上相反的颜色，并将这个点加入队列进行下一步判断；如果有则需要判定上色是否合法，不合法则说明不是Bipartite Graph。

最后如果是连通图，则所有点都能被上色，说明可以被分为两组。但如果是非连通图则不能一次性把所有的点都这样上色。因此需要引入一个数组记录结点是否被访问过，出现没有被访问过的结点则需要以这个结点为开头进行BFS访问，直到所有结点都被访问过且都返回True为止。如果任意的搜索返回了False则说明分组失败。



解法二：DFS+Coloring判断Bipartite Graph

参考

也是对图进行上色。对某个点进行上色后，判断其邻近的点，如果有被上色则判断上色是否合法，如果没有被上色则调用DFS对这个点进行上色并判断。

同样针对非连通图需要引入一个数组来判定是否所有的点都被访问到。

---

997. Find the Town Judge

HashMap; Vertex Degree

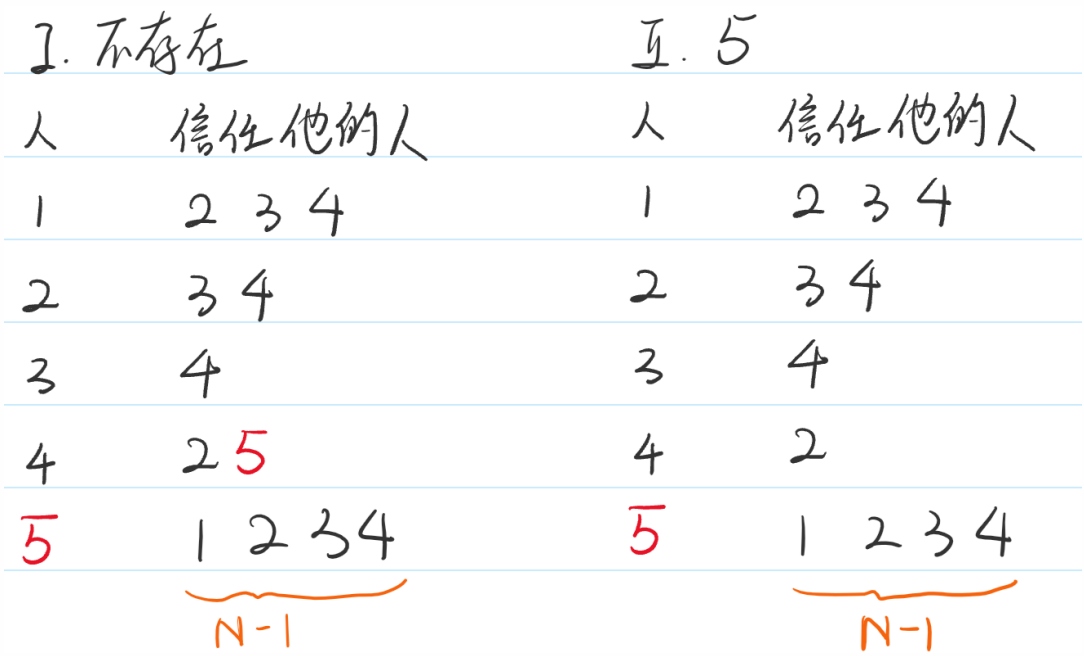
> In a town, there are N people labelled from 1 to N. There is a rumor that one of these people is secretly the town judge.
>
> If the town judge exists, then:
>
> - The town judge trusts nobody.
> - Everybody (except for the town judge) trusts the town judge.
> - There is exactly one person that satisfies properties 1 and 2.
>
> You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.If the town judge exists and

> can be identified, return the label of the town judge. Otherwise, return -1.

解法一：HashMap

用HashMap保存每个人被他人信任的情况，每个人的编号即为一个Key，值使用ArrayList，一旦遇到有人信任这个人就加入ArrayList并在HashMap中更新。遍历所有编号，如果某个人对应的ArrayList长度是N - 1，则这个人有可能是Town Judge。

但对于此人还需检查他是否出现在别人对应的ArrayList中。



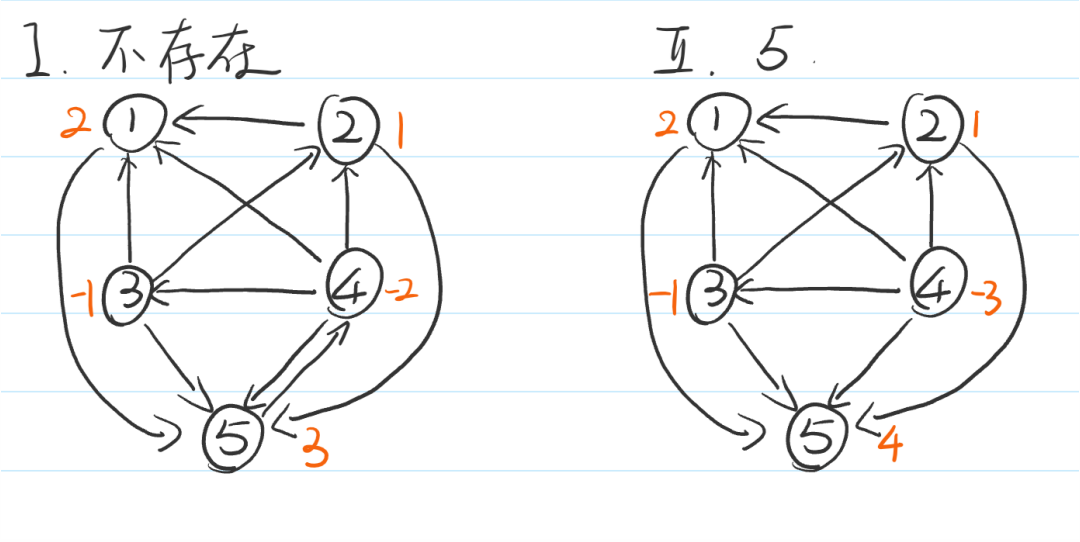
解法二：Graph Vertex Degrees

参考

可以把这道题目想象成一张有向图，对于每个人（结点），如果有人（结点）信任他则将两个结点相连，信任者指向被信任者。最终

每个结点的度数 = 指向其的边 - 其指向外的边

如果有结点度数为N - 1，则是Town Judge




```
class Solution {
    public int findJudge(int N, int[][] trust) {
        int[] ans = new int[N + 1];
        for (int i = 0; i < trust.length; ++i)
        {
            ans[trust[i][1]]++;
            ans[trust[i][0]]--;
        }
        for (int i = 1; i <= N; ++i)
        {
            if (ans[i] == N - 1)
                return i;
        }
        return -1;
    }
}
```

1462. Course Schedule IV

DFS; Floyd-Warshall

解法一：DFS

使用邻接表保存有向图，用DFS从queries的起始点开始进行遍历查找，找到终点则返回True，否则在这个结点的邻接结点中继续进行查找。注意需要引入visited数组来判定结点是否被查找过，否则会死循环。

由于对于每个queries数据都需要进行一遍查找，时间复杂度较高。

解法二：Floyd-Warshall

时间复杂度：O(n^3)； 空间复杂度：O(n^2)

参考1：评论区题解

参考2：Quora

用邻接矩阵保存有向图。

Floyd-Warshall 原本是用于构建最小生成树的算法，但是稍作修改也可以用于判断两个点之间是否连通。

对于两个点i, j之间是否联通，我们需要判断

1. 两点之间是否本身就有路径
2. 两点经过k为中间点是否能有连通路径
    1. i 到 k 之间是联通的
    2. k 到 j 之间是联通的

注意第二个判断的两个子判断本质是对这个问题的一种递归。

需要注意的是，三重循环在这里的顺序**必须要k在最外层**。因为进行判断所需要的情况不仅仅是某个i到k是否一开始就有连接&&某个k到j是否一开始连接。对于i到k是否有连接，也需要通过一样的方法进行更新，所以是在k不断循环向前的情况一下一一判断不同的i, j对是否能够找到连接的路线，每次k更新都要重新把所有的i, j配对重新判断。

最后能把所有结点的情况都更新，根据queries直接输出即可。

```
class Solution {
     public List<Boolean> checkIfPrerequisite(int n, int[][]
prerequisites, int[][] queries)
    {
        boolean[][] connected = new boolean[n][n];
        for (int i = 0; i < prerequisites.length; ++i)
            connected[prerequisites[i][0]][prerequisites[i]
[1]] = true;
        for (int k = 0; k < n; ++k)
        {
            for (int i = 0; i < n; ++i)
            {
                for (int j = 0; j < n; ++j)
                    connected[i][j] = connected[i][j] ||
(connected[i][k] && connected[k][j]);
            }
```

```
            }
            List<Boolean> ans = new ArrayList<>();
            for (int i = 0; i < queries.length; ++i)
                ans.add(connected[queries[i][0]][queries[i][1]]);
            return ans;
        }
    }
```

Trees:

110. Balanced Binary Tree

AVL; Recursion

解法一：递归

递归判断左子树和右子树的高度。

1. 左子树或右子树中有不符合AVL条件的（返回-1），则直接返回-1
2. 左子树右子树高度相差超过1，不符合条件，返回-1
3. 符合条件，返回当前结点所代表的子树的高度，是左右子树高度的max + 1

最后如果返回-1则说明不是AVL。

---

208. Implement Trie (Prefix Tree)

参考

TreeNode

Trie是一种查找树，同前缀的几个字符串可以共用前缀分支。

一个结点首先有当前结点的值，其次有一个TreeNode数组来保存所有的children，因为只有可能有26个英文字母，所以是长度为26的数组。最后还有isWord用来确定这个结点之前的结点是否组成一个单词。

```
class TrieNode
    {
        public boolean isWord;
        public char val;
        public TrieNode[] children = new TrieNode[26];
        public TrieNode() {}
        TrieNode(char c)
        {
            val = c;
            isWord = false;
        }
    }
```

insert

插入一个单词的时候，循环遍历整个单词，首先从根节点开始，在children数组里查找是否有这个词的第一个字母，如果有则沿着cursor下移，在这个字母的children里找下一个字母是否存在。如果不存在则创造一个新的结点记录在children数组中，并cursor下移，循环到下一个字母继续进行添加。

最后要在cursor所在的结点（比单词后一位）将isWord标记为true。

search

循环遍历整个单词，从根节点开始，在children数组里查找是否有这个单词的第一个字母，如果有则cursor进入这个结点的children数组的对应字符，检查下一个字母。一旦有字母匹配失败则说明单词不存在。

如果顺利完成匹配还需返回cursor的isWord域。

startsWith

和search一样，但是只要完成所有匹配即可，不需要isWord判断。

---

226. Invert Binary Tree

Recursion

解法一：递归

对于任意结点，Base Case是当结点为null的时候，返回null。其他情况下先将左右子树调换，再对左子树和右子树分别调用递归函数，将左子树和右子树也进行调换操作。

315. Count of Smaller Numbers After Self

BST; AVL

解法一：AVL平衡二分查找树

参考1:评论区题解 参考2:Youtube视频

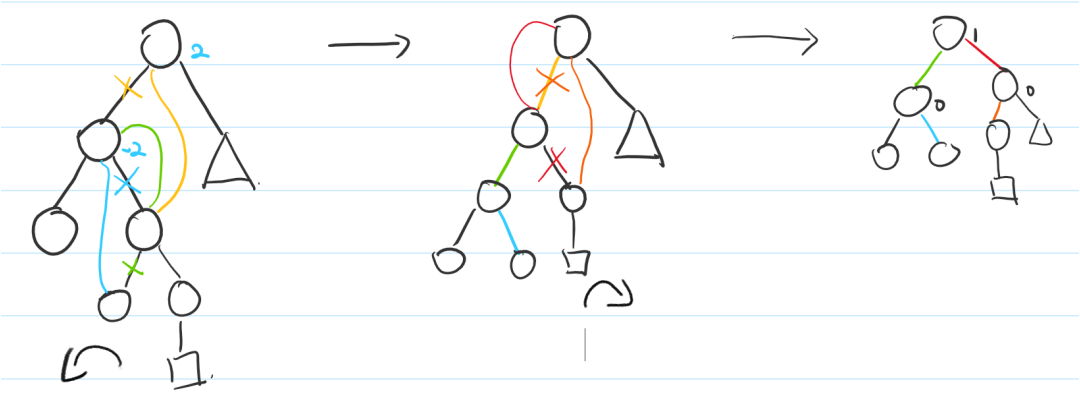
二分查找树可以让我们快速知道树中有多少个结点比当前结点的数值要大/小。所以我们可以从后往前将数字放入二分查找树中，完成放入时因为我们知道结点的位置，所以可以自然知道有多少结点比这个结点小，所以就知道在这个数字之后有多少数字比这个数字小。平衡二分查找树可以降低树的深度，从而节省时间，虽然使用普通二分查找树也是可以的。

首先，在插入的过程中可以用一个counter来记录有多少结点比当前结点小，递归地查找插入位置

1. 如果当前结点和要插入的结点相等，将结点Frequency加一。Counter需要额外加上左子树的大小。
2. 如果比当前结点数值小，在左子树Recursively进行插入。
3. 如果比当前结点数值大，在右子树Recursively进行插入。Counter需要额外加上左子树的大小以及当前结点的Frequency。

接下来需要让树保持平衡。平衡方式为左旋或右旋。总的来说，如果左子树高度过高，则需要右旋，如果右子树高度过高，则需要左旋。但是可能会出现子树需要旋转的方向和当前结点需要旋转的方向不同。所以一共分为四种情况。

1. 当前结点Balance < -1，即右子树高度太高
    1. 对右子树是否平衡进行计算，如果右子树的Balance > 1，即其左子树高度太高，则先右子树进行右旋，再对整棵树进行左旋
    2. 否则直接对整棵树进行左旋
2. 当前结点Balance > 1，即左子树高度太高
    1. 对左子树是否平衡进行计算，如果左子树的Balance < -1，即其右子树高度太高，则先对左子树进行左旋，再对整棵树进行右旋
    2. 否则直接对整棵树进行右旋



旋转的时候总是先选出即将做新的根节点的点，然后把这个结点原本的左（右）结点移给原来的根节点做它的右（左）子树，然后再把旧的根节点接给新的根节点。

解法二：普通二分查找树

参考

用普通二分查找树也可以实现，只是不去平衡，可能会导致树的高度过高从而增加时间消耗。每个结点保存当前结点的Freq和左子树的大小。

1. Base Case：如果root是nullptr，则新建Node并返回
2. Base Case：如果遇到数字一样的结点，则更新Freq，Count要增加leftChildSize
3. 如果数字比当前结点大，往右子树插入，Count要额外加上leftChildSize和当前结点Freq
4. 如果数字比当前结点小，往左子树插入，更新左子树大小

```
class Solution {

class Node {
    public:
    int val;
    int freq;
    int leftChildSize;
    Node* left;
    Node* right;

    Node(int num)
    {
        val = num;
        freq = 1;
        leftChildSize = 0;
        left = nullptr;
        right = nullptr;
```

```
    }

};
public:
    vector<int> countSmaller(vector<int>& nums) {
        Node* root = nullptr;
        vector<int> ans;
        for (int i = nums.size() - 1; i >= 0; --i)
        {
            int counter = 0;
            root = insert(root, nums[i], &counter);
            ans.insert(ans.begin(), counter);
        }
        return ans;
    }

    Node* insert(Node* root, int num, int* count)
    {
        if (root == nullptr)
        {
            root = new Node(num);
            *count = 0;
            return root;
        }
        if (num == root->val)
        {
            root->freq++;
            *count = *count + root->leftChildSize;
        }
        else if (num < root->val)
        {
            root->left = insert(root->left, num, count);
            root->leftChildSize++;
        }
        else if (num > root->val)
        {
            root->right = insert(root->right, num, count);
               *count = *count + root->leftChildSize + root-
>freq;
        }
        return root;

    }
};
```

700. Search in a Binary Search Tree

BST; Recursion

1008. Construct Binary Search Tree from Preorder Traversal

BST; Recursion

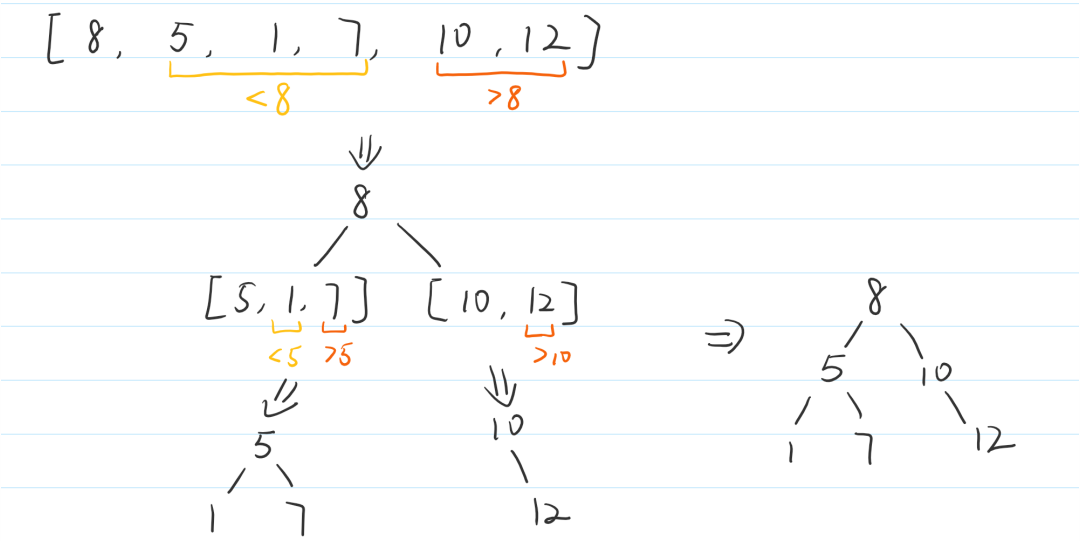
> Return the root node of a binary search tree that matches the given preorder traversal.

解法一：Recursion+遍历查找

给定前序遍历的数组，则任意这样一个数组或是表达一个子树的数组都有第一个点为根节点，剩下分成两部分，前一部分组成左子树，后一部分组成右子树。

另外根据二分查找树的性质，左子树中所有结点必定小于根节点数值，右子树中所有结点必定大于根节点数值，所以在根节点之后的数组中必定有一个点可以分割左右子树，只需遍历找到第一个比根节点数值大的结点即可。

接下来递归调用子树构建即可。




```
class Solution {
    public TreeNode bstFromPreorder(int[] preorder) {
        return bstHelper(preorder, 0, preorder.length - 1);
    }

    private TreeNode bstHelper(int[] arr, int st, int ed)
    {
        int val = arr[st];
        TreeNode root = new TreeNode(val);
```

```
        int i = st + 1;
        while (i <= ed && arr[i] < val)
            i++;
        if (st + 1 <= i - 1)
            root.left = bstHelper(arr, st + 1, i - 1);
        if (i <= ed)
            root.right = bstHelper(arr, i, ed);
        return root;
    }
}
```

解法二：直接Recursion

参考

时间复杂度：O(n)； 空间复杂度：O(1)

根据二分查找树和前序遍历的性质，数组中永远是先出现根节点数值再出现左子树和右子树数值。所以可以按顺序一一构建。给定一个上限值（对于左子树是根节点数值，对于右子树可以是上一个Parent结点的数值，不过右子树的上限一开始从无穷大开始），程序不断地生成结点并用一个计数器计数，直到达到上限所有结点都被构建完成。

这样省去了查找分界点的时间，因为程序不需要预先知道哪里是分界点，只要一边构造一边检测直到到达边界则返回nullptr即可。

```
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int i = 0;
        return bstPreorder(preorder, i, INT_MAX);
    }

     TreeNode* bstPreorder(vector<int>& A, int& i, int bound)
{
        if (i == A.size() || A[i] > bound)
            return nullptr;
        TreeNode* root = new TreeNode(A[i++]);
        root->left = bstPreorder(A, i, root->val);
        root->right = bstPreorder(A, i, bound);
        return root;
    }
};
```

1457. Pseudo-Palindromic Paths in a Binary Tree

DFS; HashSet; Stack

> Given a binary tree where node values are digits from 1 to 9. A path in the binary tree is said to be pseudo-palindromic if at least one permutation of the node values in the path is a palindrome.
>
> Return the number of pseudo-palindromic paths going from the root node to leaf nodes.

解法一：DFS+Stack记录结点

DFS

对于任意结点，首先将当前数值入栈。如果左右子树都不存在则说明是叶子结点，这个时候检查保存结点的栈内的结点是否构成伪回文字符串。如果是则对ans增加1。如果左子树或右子树不为空则对左子树或右子树进行DFS。最后左右遍历都结束时将当前结点出栈。

检查伪回文字符串

检查伪回文字符串只需要把栈转化为数组，检测每个数值的出现次数，只有奇数长度的栈能有至多一个数的出现次数是奇数次，其他都必须是偶数次，才算一个伪回文字符串。

解法二：DFS+Array直接记录结点

参考

可以改进这个解法，只需要一个一维数组存储数值出现频率（因为数字只有0-9），之后根据频率检查即可。之前出栈的操作可以用改变数字频率来替代。

Search:

130. Surrounded Regions

DFS

解法一：DFS反向查找

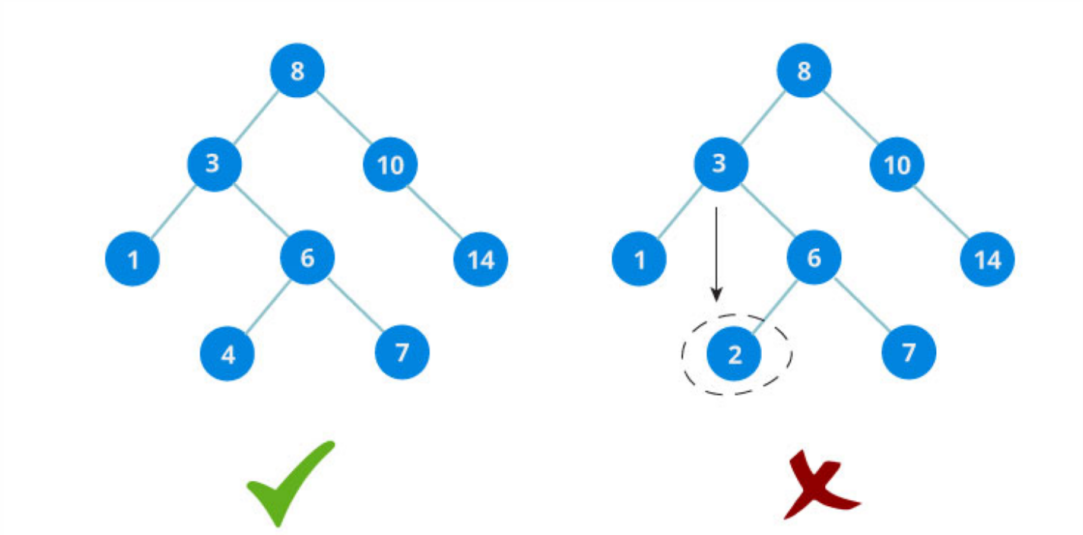
一开始没有想到因为一直在用DFS考虑如何找到整块被包裹的O区域，但是很难实现，因为要考虑四个方向，且情况比较多。所以简单的方法是先找出不需要做改变的O，然后把剩下的O变成X。

找出不需要做改变的O比较简单，只要从边界开始，如果遇到O就进行DFS搜索，可以先把O变成另一个字母P，查找出所有能从边界能走到的O，他们没有被X完全包裹。接下来把剩下的所有O都变成X，最后把P变回O即可。

230. Kth Smallest Element in a BST

DFS

> Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.



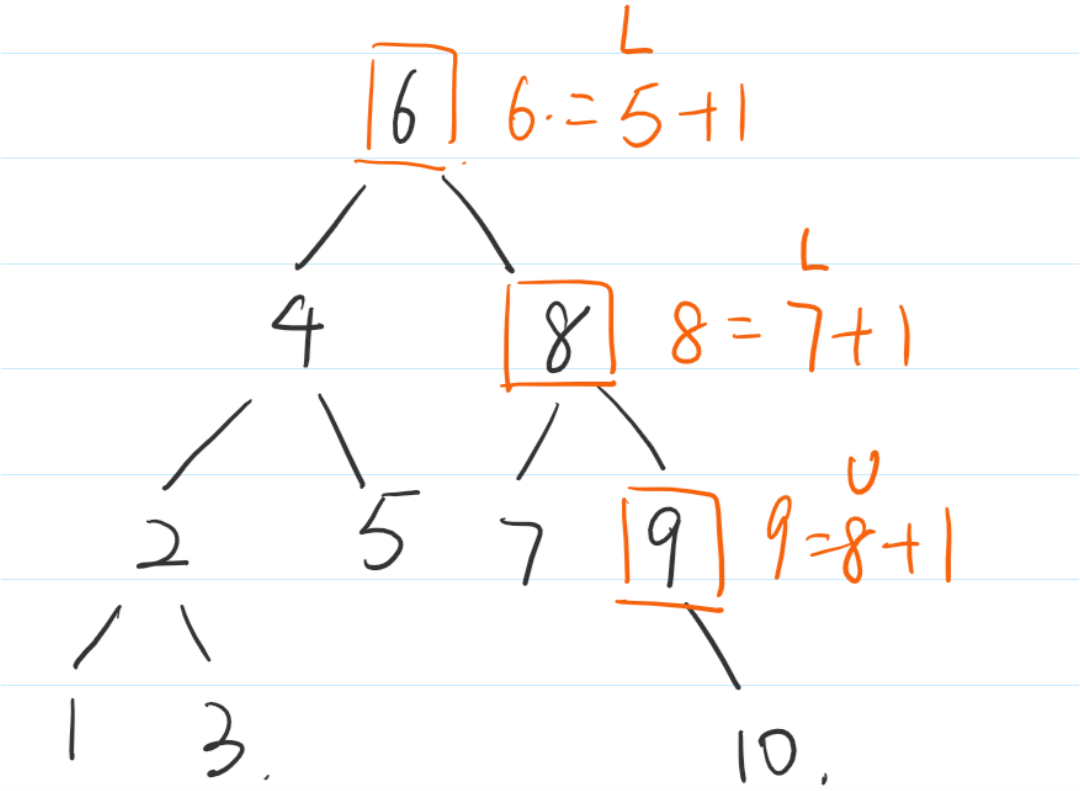
解法一：DFS

时间复杂度：O(n)； 空间复杂度：O(1)

对于任意结点在所有结点中的排序号，

1. 如果该结点有Left Child，则需要加上Left Child的排序值
2. 如果该结点有Parent Node，则需要加上Parent Node的排序值
3. 之后再给结果加1得到该点的序号

注意：这里结点的值也指**以这个结点为根的整个子树**的排序值。



当我们每到达一个点的时候，我们需要知道前置点Parent Node的排序值，因此需要prev参数。同时我们也要把这个点的排序值送给下一个结点。

为了知道Left Child的值，我们先对Left Child进行DFS，需要pass in 从上一个点得到的prev，将返回的值加上1之后得到这个点的值，与k进行比较，如果和k相等则存入ans中。继续Right Child的查找，需要将这个结点的值作为prev传递给接下来的搜索中。

最终遍历整棵树得到结果。

```
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        int[] ans = new int[1];
        dfs(root, k, 0, ans);
        return ans[0];
    }
    public int dfs(TreeNode root, int k, int prev, int ans[])
    {
        if (root == null)
            return t;
        int cur = dfs(root.left, k, prev, ans) + 1;
        if (k == cur)
            ans[0] = root.val;
        return dfs(root.right, k, cur, ans);
    }
}
```

733. Flood Fill

DFS

解法一：DFS

对于每个点，只要这个点的颜色符合要求就对上下左右四个方向进行染色，因为被染色的格子的颜色已经有变化，所以不会出现重复染色的情况。但是要注意如果初始点和要被染色的颜色一样，就不进行操作，否则会死循环。

---

993. Cousins in Binary Tree

解法一：分步DFS

首先对于两个给定的点先分别数深度。数深度的时候，如果找到该点，则返回0，如果已经走到尽头则返回-300（因为题目设定的树的深度肯定小于300）之后返回的值应该是左子树搜索值和又子树搜索值中更大的那个加1，即为该点的深度。

其次对于搜索得到的确定深度一样的点，再次判断他们是否来自同一个Parent。直接成对搜索，如果一个点左右子树都存在且值符合给的两个值，说明两个值来自同一个Parent。

1. 如果该点的Left Child和Right Child都存在，则进行数值比较，如果不成功则分别向左和向右搜索，只要两者有其一返回True则True，否则False。
2. 如果只有Left或Right Child存在，对这个方向进行搜索。
3. 否则返回False。

解法二：直接DFS

这个过程可以简化到直接用一个DFS完成。多Pass in两个parameter，一个记录depth，一个记录parent，只要查找到了题目给定的点就把这两个信息储存下来，最后在主程序中进行比较。

1466. Reorder Routes to Make All Paths Lead to the City Zero

解法一：DFS

参考

一开始没想出来，尤其说是Minimal change容易想偏掉。其实因为给定了条件每两个点之前都只有一条确定的路线，所以一旦遇到一条背离0点的路线，必须要进行翻转。所以只需要从0点开始搜索，遇到远离这个点的就将答案加一，然后进入这个点进行继续搜索，因为这个点可以直接到达0，所以远离这个点的路径都需要做反转，然后继续搜索。

使用邻接表存储路线的过程中，每次存储将相反的路线用负数表示存入图中，因此在搜索的时候如果遇到路线是负数且接下来那个点没有被搜索过，就需要记录并进入那个点进行继续搜素。

```
class Solution {
    public int minReorder(int n, int[][] connections)
    {
        List<Integer>[] adjList = new ArrayList[n];
        for (int i = 0; i < n; ++i)
            adjList[i] = new ArrayList<Integer>();
        for (int i = 0; i < connections.length; ++i)
        {
                adjList[connections[i][1]].add(connections[i]
[0]);
               adjList[connections[i][0]].add(-connections[i]
[1]);
        }
        boolean[] visited = new boolean[n];
        int[] ans = new int[1];
        dfs(0, adjList, visited, ans);
        return ans[0];
    }

       public  void  dfs(int  src,  List<Integer>[]  adjList,
boolean[] visited, int[] count)
    {
        visited[src] = true;
        for (int i : adjList[src])
        {
            if (i >= 0)
            {
                if (!visited[i])
                    dfs(i, adjList, visited, count);
            }
```

```
            else if (!visited[-i])
            {
                count[0]++;
                dfs(-i, adjList, visited, count);
            }
        }
        return;
    }
}
```
