树

110. 平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。
本题中，一棵高度平衡二叉树定义为：
一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

示例 1:
给定二叉树 [3,9,20,null,null,15,7]
返回 true 示例 2:
给定二叉树 [1,2,2,3,3,null,null,4,4]
返回 false

513. 找树左下角的值

给定一个二叉树，在树的最后一行找到最左边的值。
注意: 您可以假设树（即给定的根节点）不为 NULL。

144. 二叉树的前序遍历

给定一个二叉树，返回它的 前序 遍历。
输入: [1,null,2,3]
输出: [1,2,3]

230.二叉搜索树中第K小的元素

给定一个二叉搜索树，编写一个函数 kthSmallest 来查找其中第 k 个最小的元素。
说明：
你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。
示例 1:
输入: root = [3,1,4,null,2], k = 1
输出: 1
示例 2:
输入: root = [5,3,6,2,4,null,null,1], k = 3
输出: 3
进阶： 如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化 kthSmallest 函数？

208. 实现 Trie (前缀树)

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

Trie trie = new Trie();

trie.insert("apple"); trie.search("apple"); // 返回 true trie.search("app"); // 返回 false
trie.startsWith("app"); // 返回 true trie.insert("app");
trie.search("app"); // 返回 true 说明:

你可以假设所有的输入都是由小写字母 a-z 构成的。 保证所有输入均为非空字符串。

In [1]:
```python
class TrieNode():
    def __init__(self):
        self.nodes = [None] * 26
        self.last = False
class Trie(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TrieNode()

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: None
        """
        tree = self.root
        while (word):
            temp = word[0]
            word=word[1:]
            if tree.nodes[ord(temp) - ord('a')]==None:
                tree.nodes[ord(temp) - ord('a')] = TrieNode()
            tree = tree.nodes[ord(temp) - ord('a')]
        tree.last = True

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        if word=='':
            return True
        tree=self.root
        while(word):
            temp = word[0]
            word = word[1:]
            if tree.nodes[ord(temp)-ord('a')]==None:
                return False
            tree=tree.nodes[ord(temp)-ord('a')]
        return tree.last


    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that starts with the given prefix.
        :type prefix: str
        :rtype: bool
        """
        tree=self.root
        while(prefix):
            temp=prefix[0]
            prefix=prefix[1:]
            if tree.nodes[ord(temp)-ord('a')]==None:
                return False
            tree=tree.nodes[ord(temp)-ord('a')]
        return True
```

In [2]:
```python
# Your Trie object will be instantiated and called as such:
obj = Trie()
word = "apple"
obj.insert(word)
param_1 = obj.search(word)
param_2 = obj.startsWith("app")
print(param_1)
print(param_2)
```

```
True
True
```

图

785 判断二分图

给定一个无向图graph，当这个图为二分图时返回true。

如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。

graph将会以邻接表方式给出，graph[i]表示图中与节点i相连的所有节点。每个节点都是一个在0到graph.length-1之间的整数。这图中没有自环和平行边： graph[i] 中不存在i，并且graph[i]中没有重复的值。

示例 1:
输入: [[1,3], [0,2], [1,3], [0,2]]
输出: true
我们可以将节点分成两组: {0, 2} 和 {1, 3}
示例 2:
输入: [[1,2,3], [0,2], [0,1,3], [0,2]]
输出: false
我们不能将节点分割成两个独立的子集。

注意:

graph 的长度范围为 [1, 100]。
graph[i] 中的元素的范围为 [0, graph.length - 1]。 graph[i] 不会包含 i 或者有重复的值。
图是无向的: 如果j 在 graph[i]里边, 那么 i 也会在 graph[j]里边。

In [3]:
```python
class Solution:
    def isBipartite(self, graph):
        n = len(graph)
        colors = [0] * n
        # 0 未被染色的 1 是红色 -1 是蓝色
        def dfs(i,color):
            if colors[i] != 0:
                return colors[i] == color
            colors[i] = color
            for j in graph[i]:
                if not dfs(j,-color):
                    return False
            return True
        for  i in range(n):
            if colors[i] == 0 and not dfs(i,1):
                return False
        return True
```

In [4]:
```python
s =Solution()
list1 = [[1,3], [0,2], [1,3], [0,2]]
result1 = s.isBipartite(list1)
print(result1)
list2 = [[1,2,3], [0,2], [0,1,3], [0,2]]
result2 = s.isBipartite(list2)
print(result2)
```

```
True
False
```

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？

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。
通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
拓扑排序也可以通过 BFS 完成。

In [5]:
```python
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]
        finished = [0 for _ in range(numCourses)]
        # 创建图的邻接表形式
        [graph[pair[0]].append(pair[1]) for pair in prerequisites]
        # 访问每个结点，只要有一个不能完成，则整体不能完成
        if min([self.dfs(graph,finished,i) for i in range(numCourses)]) == 0:
            return False
        return True
    def dfs(self, graph, finished, i):
        if finished[i] != 0:
            return finished[i]==1
        # 标记当前结点正在访问
        finished[i] = -1
        # 访问所有依赖的课程结点
        for j in graph[i]:
            if not self.dfs(graph, finished, j):
                return False
        # 如果能到这一步，说明所有依赖的课程都能完成
        finished[i] = 1
        return True
```

In [6]:
```python
s =Solution()
numCourses1 =  2
prerequisites1 = [[1,0]]
result1 =  s.canFinish(numCourses1,prerequisites1)
print(result1)
numCourses2 =  2
prerequisites2 = [[1,0],[0,1]]
result2 =  s.canFinish(numCourses2,prerequisites2)
print(result2)
```

```
True
False
```

684. 冗余连接

在本问题中, 树指的是一个连通且无环的无向图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以边组成的二维数组。每一个边的元素是一对[u, v] ，满足 u < v，表示连接顶点u 和v的无向图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 [u, v] 应满足相同的格式 u < v。

示例 1：
输入: [[1,2], [1,3], [2,3]]
输出: [2,3]

示例 2：
输入: [[1,2], [2,3], [3,4], [1,4], [1,5]]
输出: [1,4]

注意:
输入的二维数组大小在 3 到 1000。
二维数组中的整数在1到N之间，其中N是输入数组的大小。
更新(2017-09-26):
我们已经重新检查了问题描述及测试用例，明确图是无向 图。对于有向图详见冗余连接II。对于造成任何不便，我们深感歉意。

In [7]:
```python
class Solution(object):
    def findRedundantConnection(self, edges):
        """
        :type edges: List[List[int]]
        :rtype: List[int]
        """
        rec = {}
        for (start, end) in edges:
            if start in rec and end in rec:
                p1, p2 = start, end
                while rec[p1] is not None:
                    p1 = rec[p1]
                while rec[p2] is not None:
                    p2 = rec[p2]
                if p1 == p2:
                    return [start, end]
                rec[p1] = p2

            if start not in rec:
                rec[start] = None
            if end not in rec:
                rec[end] = start
            else:
                p2 = end
                while rec[p2] is not None:
                    p2 = rec[p2]
                rec[start] = p2
```

In [8]:
```python
s = Solution()
edges1 = [[1, 2], [1, 3], [2, 3]]
result1 = s.findRedundantConnection(edges1)
print(result1)
edges2 = [[1, 2], [2, 3], [3, 4], [1, 4], [1, 5]]
result2 = s.findRedundantConnection(edges2)
print(result2)
```

```
[2, 3]
[1, 4]
```