5，图论

2018年4月27日 17:01

- 图论简介
- 面试题总体分析
- 一些例题
  - 例1 给定二叉树前中序遍历，构造二叉树
  - 例2 二叉树高度、最小深度、二叉搜索树判断、对称判断、平衡判断
  - 例3 二叉树与链表转换
  - 例4 无向图复制
  - 例5 直角路线遍历棋盘
- 总结
- 图论简介
- 图结构
  - 节点
  - 边
- 分类
  - 有向图
  - 无向图
- 特殊的图
  - 二叉树： 二叉搜索树
  - 普通树 （并查集）
  - 堆

- 面试题总体分析
- 图
  - 连通性（割点、边）

- 最小生成树
- 最短路
- 搜索(BFS,DFS)
- 欧拉回路
- 哈密尔顿回路
- 拓扑排序

- 树
  - 树的定义与判断
  - 平衡、二查搜索树、最大（小）高度、最近公共祖先

- 一些例题
  - 例1 给定二叉树前中序遍历，构造二叉树

```
/*
https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/description/
问题： 给定二叉树前、中序遍历，构造二叉树 （Leetcode 105）

分析：首先，前序遍历的第一个节点是根节点x，
      然后在中序遍历中找到节点x，那么x左边序列为左子树的中序遍历，在前序遍历中对应长度的序列为左子树的前序遍历序列
      同样的，x右边的中序遍历为右子树的中序遍历，在前序遍历中对应长度的序列为右子树的前序遍历过程。
      然后递归该过程
*/


/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        return dfs(preorder, inorder, 0, 0, preorder.size());
    }

private:
    TreeNode* dfs(vector<int>& preorder,vector<int>& inorder,int fromp,int fromi,int len)
    {
```

```
        if(len == 0) return NULL;
        TreeNode* root = new TreeNode(preorder[fromp]);
        int i;//在中序中找根节点的位置，定义在外边，是因为后面还要用
        for(i = fromi;inorder[i]!=preorder[fromp];i++);
        root->left = dfs(preorder,inorder,fromp+1,fromi,i-fromi);//左子树的前序开始位置，左子树的中序开始位置，和左子树的序列长度(根据中序来算)
        root->right =dfs(preorder,inorder,fromp+1+i-fromi,i+1,len-1-i+fromi);//建议画图
        return root;
    }
};
```

来自 <http://tool.oschina.net/highlight>

- **例2 二叉树高度、最小深度、二叉搜索树判断、对称判断、平衡判断**

```
/*
问题1：    (Leetcode 124) 二叉树每个节点有一个整数，返回和最大的路径。
https://leetcode.com/problems/binary-tree-maximum-path-sum/description/

分析：这个最大路径有三种可能：
    左子树延伸下去的最大路径，右子树延伸下去的最大路径，和通过根节点的路径

问题2：    二叉树最小深度 （从根到叶子最小经过的节点数）(Leetcode  111) 注意空子树
https://leetcode.com/problems/minimum-depth-of-binary-tree/description/

问题3：  判断平衡（a binary tree in which the depth of the two subtrees of every node never differ by more than 1.），leetcode110
https://leetcode.com/problems/balanced-binary-tree/description/

问题4：  最大深度 leetcode104
https://leetcode.com/problems/maximum-depth-of-binary-tree/description/


问题5：  判断相同 leetcode100
https://leetcode.com/problems/same-tree/description/

问题6：  判断对称 leetcode101
https://leetcode.com/problems/symmetric-tree/description/

问题7：  判断二叉搜索 leetcode 98
https://leetcode.com/problems/validate-binary-search-tree/description/
有很多种方法，一种简单的思路是判断其中序遍历的结果是否是有序的，即可
并且无需存这个序列，只要当前值比上一个大即可。
*/


/**
 * Definition for a binary tree node.
 * struct TreeNode {
```

```
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution1 {
public:
    int maxPathSum(TreeNode* root) {
        if(!root) return NULL;
        int res = root->val;
        dfs(root,res);
        return res;//注意，这里返回的是参数res，而非dfs的返回值
    }
private:
    int dfs(TreeNode *root,int &m)//m传的引用,dfs的返回值为该条路上的最大值，而m为全局最大值
    {
        if(!root) return 0;
        int left = dfs(root->left,m);
        int right = dfs(root->right,m);
        int ret = max(max(left,right),0)+root->val;//左右子树延伸下去的最大路径
        m = max(max(m,ret),left+right+root->val);//左右子树延伸下去的最大路径，原来的m和l+r+root三者的最大值为，全局的
        return ret;
    }
};


class Solution2 {
public:
    //这里需注意的一点是如果左子树为空，那么它最小高度为右子树的高度，而非是1（只包含根节点）
    int minDepth(TreeNode* root) {
        if(!root) return 0;
        if(root->left)
        {
            if(root->right)
                return min(minDepth(root->left),minDepth(root->right))+1;//左右子树都存在
            else//只有左子树
                return minDepth(root->left) +1;
        }
        else if(root->right)//左不在，右在
            return minDepth(root->right)+1;
        else//左不在，右不在
            return 1;
    }
};
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution3 {
public:
    bool isBalanced(TreeNode* root) {
        int h;//树高
        return help(root,h);
    }
private:
    bool help(TreeNode* root,int &h)
    {
        if(root == NULL)
        {
            h = 0;
            return true;
        }
        int h1,h2;
        if(!help(root->left,h1))//如果左子树非平衡
            return false;
        if(!help(root->right,h2))//如果右子树非平衡
            return false;
        h = max(h1,h2)+1;//更新树高
        return (h1>=h2-1)&&(h1<=h2+1);//这个条件很微妙
    }
};

class Solution4 {
public:
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        return max(maxDepth(root->left),maxDepth(root->right))+1;
    }
};

class Solution5 {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p == NULL || q ==NULL) return p == q;
        return (p->val == q->val) &&isSameTree(p->left,q->left)&&isSameTree(p->
right,q->right);

    }
```

```
};

class Solution6 {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return help(root->left,root->right);
    }
private:
    bool help(TreeNode* r1,TreeNode* r2)
    {
        if(r1 == NULL || r2 == NULL) return r1 == r2;
        return (r1->val == r2->val)&&help(r1->left,r2->right)&&help(r1->right,r2->
left);//注意，是对称，轴对称
    }
};

class Solution7 {
public:
    bool isValidBST(TreeNode* root) {
        bool mask = true;//是否为第一次访问的节点
        int val = 0;//中序遍历上一个值
        return help(root,mask,val);
    }
private:
    bool help(TreeNode* root,bool &first,int &last)
    {
        if(!root) return true;
        if(!help(root->left,first,last))//若左子树非，false
            return false;
        if(first)//如果左子树为空，那么first就是true，表示一个元素都没访问过，root为
访问的第一个节点，更新first为false表示已经访问过节点了
        {
            first = false;
            last = root->val;
        }
        else if(last>= root->val)//first为false，说明左子树的确已经遍历过一些点了，那
么last是有意义的，那么判断当前值与前值是否有序
            return false;
        last = root->val;//如果是有序的，上一个值就变成了root->val
        return help(root->right,first,last);//再遍历右子树就可以了


    }
};
```

来自 <http://tool.oschina.net/highlight>

例3 二叉树与链表转换

```
/*
问题：二叉树与链表转换，leetcode114
https://leetcode.com/problems/flatten-binary-tree-to-linked-list/description/
分析：二叉树的每个结点有两个指针，在不创建新结点的情况下，更改结点指针的指向可将二叉树转换为链表结构，
二叉树中的左右结点变为链表中的左右结点。使用两个每个结点的两个指针可转换为双链表结构，只使用每个结点的右指针可转换为单链表结构。
1、若要把二叉搜索树转换为排序的双向链表，只需中序遍历树中的每个结点，遍历的过程中始终保存已生成链表的最右端结点，
当遍历父亲结点的时候将父亲结点加入到左子树生成的链表的最右端结点的后面，
接着再遍历父亲结点的右子树，整个遍历完成后的最右端结点即是链表的最右端结点。
2.若要把一般二叉树转换为单链表结构，只需后序遍历每个结点，在遍历父亲结点的时候
将左子树形成的链表插入到父亲结点和右子树形成的链表中间。


右子树先跟得左子树转成的链，然后跟游子数
问题：链表转平衡二叉树 leetcode109
https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/description/
*/


class Solution1 {
public:
    /*
    二叉树的每个结点有两个指针，在不创建新结点的情况下，更改结点指针的指向可将二叉树转换为链表结构，
    二叉树中的左右结点变为链表中的左右结点。使用两个每个结点的两个指针可转换为双链表结构，只使用每个结点的右指针可转换为单链表结构。
    1、若要把二叉搜索树转换为排序的双向链表，只需中序遍历树中的每个结点，遍历的过程中始终保存已生成链表的最右端结点，
    当遍历父亲结点的时候将父亲结点加入到左子树生成的链表的最右端结点的后面，
    接着再遍历父亲结点的右子树，整个遍历完成后的最右端结点即是链表的最右端结点。
    2.若要把一般二叉树转换为单链表结构，只需后序遍历每个结点，在遍历父亲结点的时候
    将左子树形成的链表插入到父亲结点和右子树形成的链表中间。
    */
    void flatten(TreeNode* root) {
        if(!root) return;
        flatten(root->left);
        flatten(root->right);
        if(root->left == NULL) return;

        //三方合并，将左子树形成的链表，插入到root和root->right之间
        TreeNode* p = root->left;
        while(p->right) p = p->right;//寻找左链表的最后一个节点
        p->right = root->right;
        root->right = root->left;
        root->left = NULL;//得到的结果为NULL root root->left pNode->right root->
right
    }
};
```

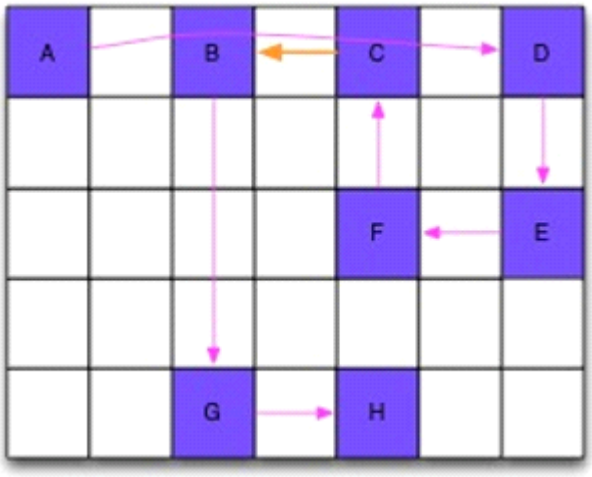
来自 <http://tool.oschina.net/highlight>

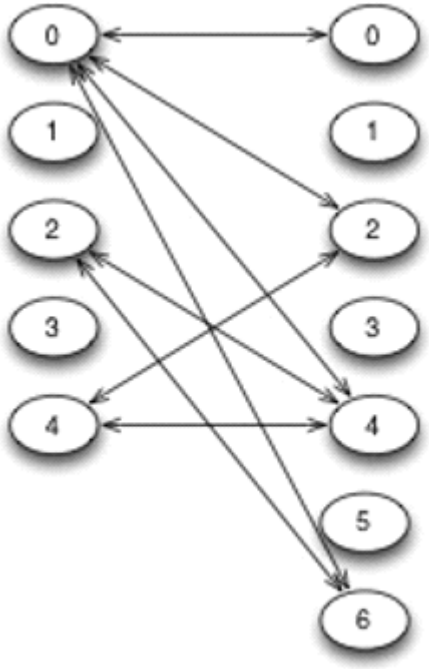
例4 无向图复制

```
/*

问题：无向图的复制，leetcode 133
https://leetcode.com/problems/clone-graph/description/

*/
/**
 * Definition for undirected graph.
 * struct UndirectedGraphNode {
 *     int label;
 *     vector<UndirectedGraphNode *> neighbors;
 *     UndirectedGraphNode(int x) : label(x) {};
 * };
 */
class Solution {
public:
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
        /*
        //BFS解法
        if(!node) return NULL;
        //unordered_map<UndirectedGraphNode*,UndirectedGraphNode*> mp;
        queue<UndirectedGraphNode*> toVisit;//待访问队列
        UndirectedGraphNode * copy = new UndirectedGraphNode(node->label);
        mp[node] = copy;
        toVisit.push(node);//记住，这里放入的是原node，因为copy现在还是个空壳子，需要跟着node，继续填充其邻居
        while(!toVisit.empty())
        {
            UndirectedGraphNode* cur = toVisit.front();
            toVisit.pop();
            for(auto neigh: cur->neighbors)//访问原节点的邻居节点
            {
                if(mp.find(neigh) == mp.end())//表示该节点并未被拷贝过，则拷贝并存入mp表示已被拷贝过，并将邻居节点本身加入toVisit，然后将邻居拷贝添加至拷贝的邻居
                {
                    UndirectedGraphNode* neigh_copy = new UndirectedGraphNode(neigh->label);
                    mp[neigh] = neigh_copy;//同样放入邻居节点和它的拷贝
                    toVisit.push(neigh);//并将邻居节点本身加入待访问队列

                }

                mp[cur]->neighbors.push_back(mp[neigh]);//注意，这里开始将原节点的邻居节点的拷贝，赋值给原节点拷贝的邻居节点

            }
        }
        return copy;*/
        //DFS解法
        if(!node) return NULL;
        if(mp.find(node) == mp.end())
        {
            mp[node] = new UndirectedGraphNode(node->label);
            for(auto neigh: node->neighbors)
                mp[node]->neighbors.push_back(cloneGraph(neigh));
```

```
        }
        return mp[node];
    }

private:
    unordered_map<UndirectedGraphNode*,UndirectedGraphNode*> mp;//标记是否被访问过,存放原节点及其拷贝,DFS解法mp必须放这里
};
```

来自 <http://tool.oschina.net/highlight>

- 例5 直角路线遍历棋盘
  - 

- 思考题(密码锁问题)
  - 


```
/*
问题： 给定矩形棋盘，再给你若干个位置(x,y)，你可以从任何给定的位置出发，只能在给定位置之间移动。
每次移动只能是沿着水平和竖直方向走，并且这次走的方向和上次不同（交错方向），每个位置只能经过一次，是否可行？ （直角遍历棋盘）
```

分析：我们把所有节点的x放在一起X，所有y放在一起Y。每走一步，要么从X跳到Y，要么从Y跳到X。（水平跳，变y，垂直跳，变x

这样就变成一个欧拉路的问题，即一笔画问题，参考ppt里面的图。每个节点能经过n次，但每条边只能走一遍

思考题： 一个密码锁，密码是4位数字，操作是
（1） 扔掉高位数字
（2） 把低位数字移动到高位
（3） 添加任意低位数字
即abcd变为bcd(前挪)e(新加），问从任意数字开始，是否可以经过0000-9999仅一次？
提示：
节点：任何3位数字从"000"到"999"
边： 后两位等于前两位 a bc->bc d 相当于一条边 代表abcd的组合
*/

来自 <http://tool.oschina.net/highlight>

- 总结
- 理解递归
- 熟悉树的遍历(递归、非递归)
- 其他问题
  - 最近公共祖先
    - 二叉树
    - 非二叉树
    - 二查搜索树
    - 离线算法－在线算法
  - (隐式）图搜索 （bfs/dfs)——（强）连通分量
    - 自己建图
  - 拓扑排序