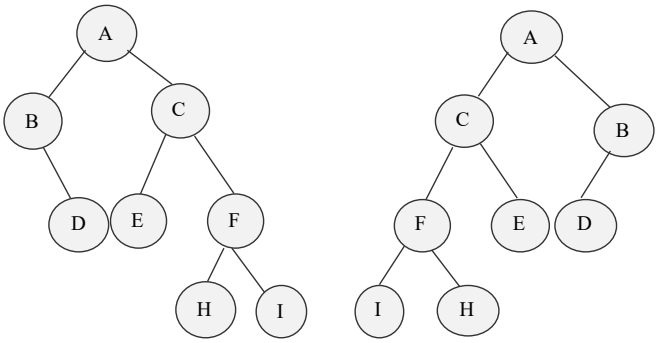
作业 5

第 6 章：树、二叉树

龚舒凯 2022202790 应用经济-数据科学实验班

2023 年 11 月 14 日

1. 两个二叉树镜像对称是指：两个树同为空树；或者两树的根结点相同，并且其中一棵树的左子树与右子树分别与另一棵树的右子树和左子树镜像对称。下图是两个对称二叉树的示例。给出判断两个二叉树是否镜像对称的递归和非递归算法。



解.

分别从第一棵树的左子树和第二棵树的右子树开始，递归地判断两棵树是否镜像对称。

时间复杂度分析： $O(n)$，其中 n 为两棵树中结点的个数。
空间复杂度分析： $O(n)$，其中 n 为两棵树中结点的个数。

```
bool judgeSymmetric(treenode *root1, treenode *root2){
        if (root1 == nullptr && root2 == nullptr){//Both are empty trees
                return true;
        }
        else if (root1 == nullptr || root2 == nullptr){//One of them is empty tree
                return false;
        }
        else if (root1->data != root2->data){//Two trees' roots' values are not equal
                return false;
        }
        else{//Two trees' roots' values are equal
        /*Judge whether the left subtree of the first tree is symmetric to
        the right subtree of the second tree*/
                return judgeSymmetric(root1->left, root2->right) &&
                judgeSymmetric(root1->right, root2->left);
        }
}
```

2. 对二叉树任意两个结点 u 和 v，根是它们的公共祖先。除根之外，可能还有其它公共祖先，而在所有公共祖先中层数最大的结点叫做 u 和 v 的最近公共祖先，即 LCA (Lowest Common Ancestor)。给出求 u 和 v 的最近公共祖先的算法。

解.

首先递归的找到结点 u, v 在树中的位置，这是通过

```
*left = lowestCommonAncestor(root->left, u, v)
*right = lowestCommonAncestor(root->right, u, v)
```

实现的。这样的操作使得在后续的递归回溯中，程序从 u, v 一层一层往上找，直到找到公共祖先为止。同时，这也保证了公共祖先是“最近的”。接下来我们关心`left`，`right`的返回值。`left`(或`right`) 不为空说明在`left`这个分支，找到了 u 或 v。

- 如果 left, right 均非空，说明 u, v 分列 root 的两旁，则最近公共祖先就是 LCA；
- 如果 left, right 中有一个为空，说明 u, v 同属非空的那个子树。继续递归地在非空的那个子树中寻找 LCA。

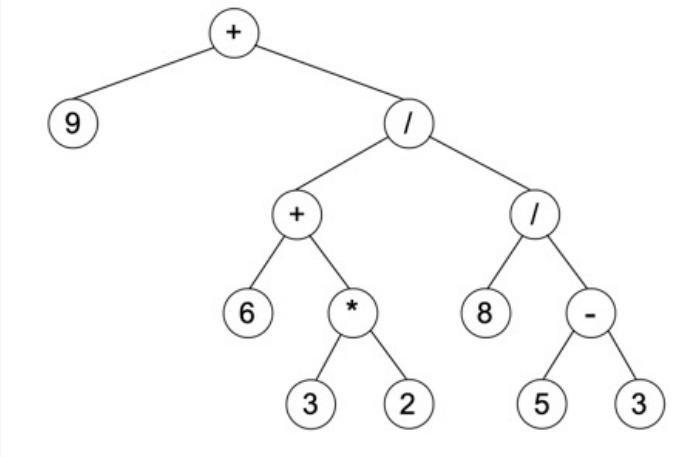
时间复杂度分析： $O(n)$，其中 n 为树中结点的个数。
空间复杂度分析： $O(n)$，其中 n 为树中结点的个数。

```
treenode* lowestCommonAncestor(treenode *root, int u, int v){
        if (root == nullptr || root->data == u || root->data == v){
                return root;
        }
        //Search from left subtree
        treenode *left = lowestCommonAncestor(root->left, u, v);
        //After this step, left is either u or v or nullptr

        //Search from right subtree
        treenode *right = lowestCommonAncestor(root->right, u, v);
        //After this step, right is either u or v or nullptr

        if (left && right){//both left and right are not empty
                //u and v are in different sides of the root
                //root is the LCA
                return root;
        }
        else return left ? left : right;
        //either left or right is empty,
        //meaning that the LCA is in the other subtree or its ancestors
}
```

3. 设计算法将中缀表达式转换为表达式树。

解.

以中缀表达式"9+(6+3*2)/(8/(5-3))"为例，我们要想将其转化为下面的表达式树，可以考虑如下流程：



1. 找到最后一个运算符，即'+'，将其作为根结点；
2. 将'+'左边的子串"9"作为根结点的左孩子；
3. 将'+'右边的子串"6+3*2/(8/(5-3))"作为根结点的右孩子；
4. 按照 1,2,3，递归地将"6+3*2/(8/(5-3))"转化为表达式树。
5. 以此类推，直到将整个中缀表达式转化为表达式树。

时间复杂度分析：记 n 为中缀表达式的长度，中缀表达式中有 m 个运算符 (不含括号)。表达式树的分支数也就是运算符的个数，即 m。每次递归调用`findLastOperator`的时间复杂度为 $O(n)$(子串的长度必然 $< n$，$O(n)$ 是`findLastOperator`的一个上界)，因此总的时间复杂度为 $O(mn)$。

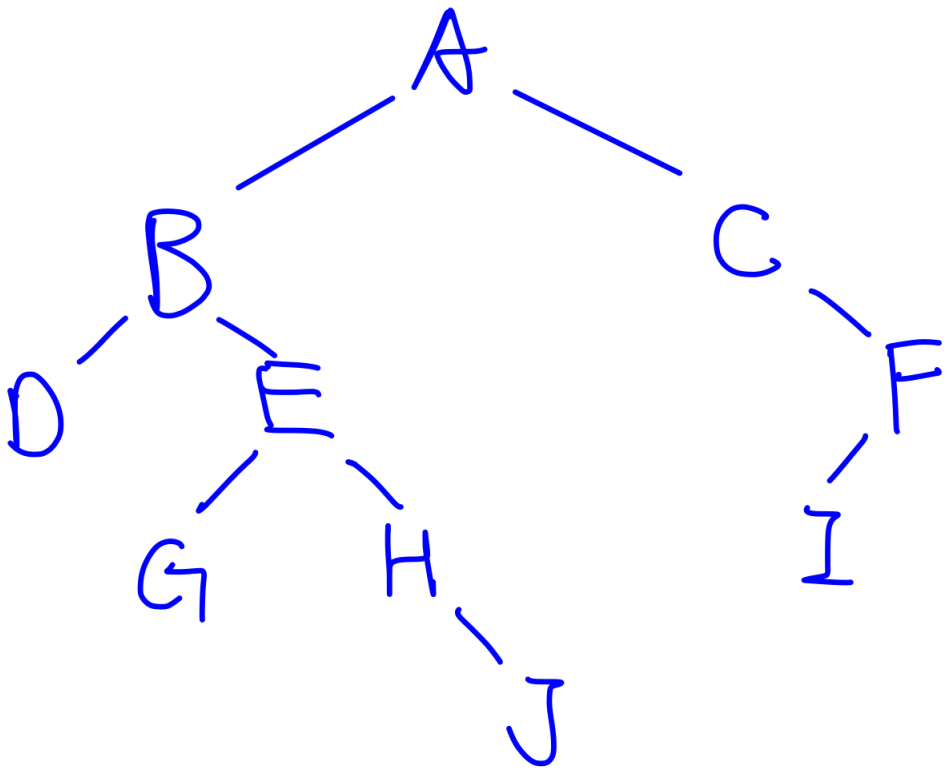
空间复杂度分析：转化为表达式树所需空间为 $O(n)$，其中 n 为中缀表达式的长度。

```
int findLastOperator(string s) {
        /*
        The logic is as followed:
        E.g. "3+(5-2)*6"
        We want to find the last operator, which is '+'.
        Operands in between brackets should be ignored,
        so we use a bracketCount to count the number of brackets.

        In the rule of Poland Expression computation,
        '*' and '/' have higher priority than '+' and '-',
```

```
        and should be firstly computed.

        The operator with minimal priority should be the last operator,
        in this case, '+'.
        We use a minPriority to record the operand with minimal priority
        */
        int opIndex = -1;
        int bracketCount = 0;
        int minPriority = INF;
        for (int i = s.size() - 1; i >= 0; i--){
                if (s[i] == ')') {
                        bracketCount++;
                }
                else if (s[i] == '('){
                        bracketCount--;
                }
                else if (bracketCount == 0 && isOperator(s[i])){
                        int precedence = getPriority(s[i]);
                        if (precedence <= minPriority) {
                                minPriority = precedence;
                                opIndex = i;
                        }
                }
        }
        return opIndex;
}

treenode* infixToTree(string s) {
        /*
        E.g. "1+2*(3-4)""
        The last operator to be executed in the infix expression is the tree root.
        In this case, '+'.
        Devide the infix expression into two parts:
        the left child of the root is "1",
        the right child of the root is "2*(3-4)".
        the infix expression tree can then be generate by recursively apply the above
        steps to the left and right child.
        */
        if (s.size() == 0){
                return nullptr;
        }
```

```
        if (s.size() == 1){
                return new treenode(s[0]);
        }
        int opIndex = findLastOperator(s);
        //Special process: in cases like (3-4), the last operator is '-'
        //The brackets should therefore be removed to get the correct operator.
        if (opIndex == -1 && s[0] == '(' && s[s.size()-1] == ')'){
                s = s.substr(1, s.size()-2);
                opIndex = findLastOperator(s);
        }

        treenode* root = new treenode(s[opIndex]);
        string front = s.substr(0, opIndex);
        string back = s.substr(opIndex + 1);
        root->left = infixToTree(front);
        root->right = infixToTree(back);
        return root;
}
```

4. 假设一棵二叉树的层序序列为 ABCDEFGHIJ 和中序序列为 DBGEHJACIF。请画出该树。



5. 试编写算法，求一棵以孩子-兄弟链表表示的树的深度。

解.

本质上，以孩子-兄弟链表表示的树就是一棵二叉树，因此求深度的算法与二叉树的求深度算法相同。

时间复杂度分析： $O(n)$，其中 n 为树中结点的个数。
空间复杂度分析： $O(n)$，其中 n 为树中结点的个数。

```
int treeDepth(treeNode *root){
        if (root == nullptr){
                return 0;
        }
        else{
                int childDepth = treeDepth(root->child);
                int sibDepth = treeDepth(root->sibling);
                return childDepth > sibDepth ? (childDepth + 1) : (sibDepth + 1);
        }
}
```