作业HW3 实验报告

1. 涉及数据结构和相关背景

- 树、二叉树的概念
- 树的建立
    - 前中建树
    - 中后建树
    - 遍历顺序加上栈的出入顺序建树
    - 前序加上空结点建树
- 树的遍历
    - 前中后序遍历
    - 层序遍历
    - 遍历方式
        - 递归
        - 非递归
- 树的线索化
- 树的转化
    - 转化为二叉树
- 求树的深度
    - dfs
- 树的应用
    - 表达式树

2. 实验内容

2.1 二叉树的同构

2.1.1 问题描述

给定两棵树T1和T2。如果T1可以通过若干次左右孩子互换变成T2，则我们称两棵树是“同构”的。例如图1给出的两棵树就是同构的，因为我们把其中一棵树的结点A、B、G的左右孩子互换后，就得到另外一棵树。而图2就不是同构的。

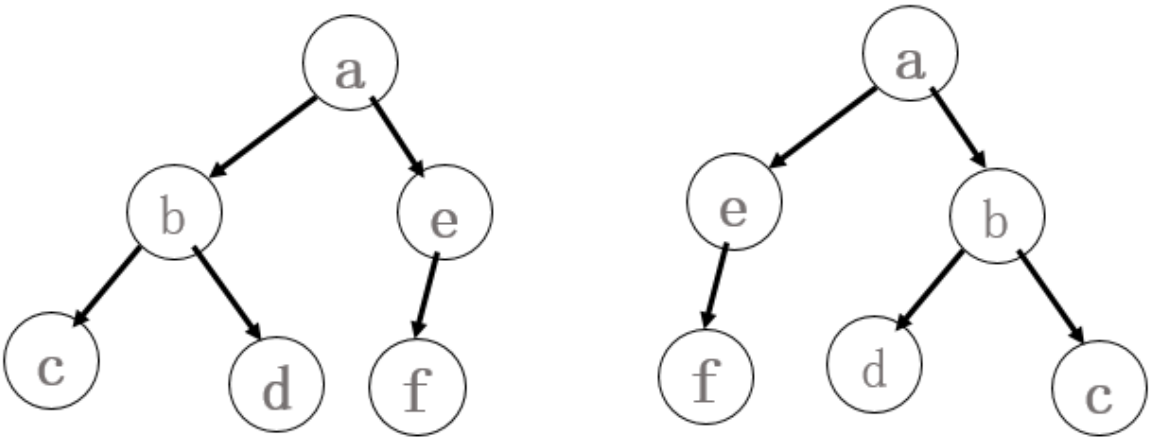


图1

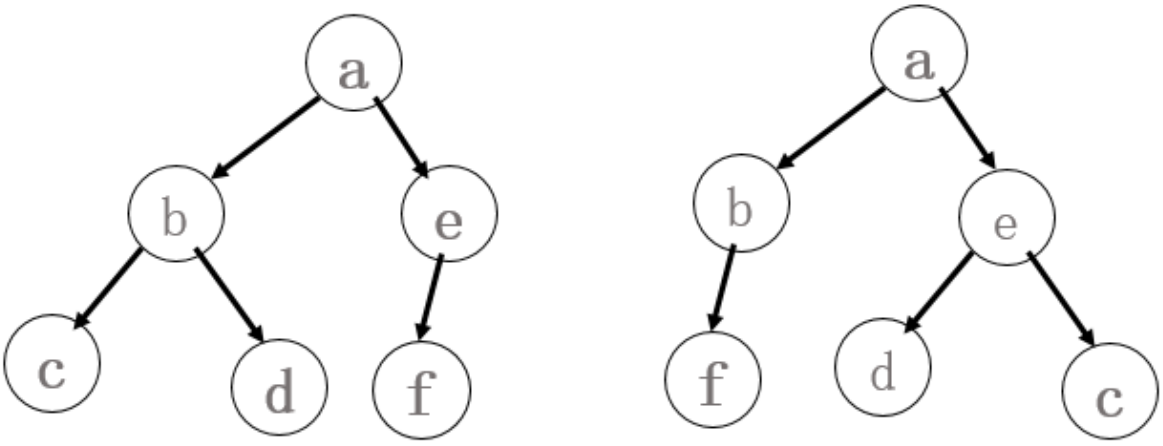


图2

现给定两棵树，请你判断它们是否是同构的。并计算每棵树的深度。

2.1.2 基本要求

- 输入：
- 第一行是一个非负整数N1，表示第1棵树的结点数；

  随后N行，依次对应二叉树的N个结点（假设结点从0到N−1编号），每行有三项，分别是1个英文大写字母、其左孩子结点的编号、右孩子结点的编号。如果孩子结点为空，则在相应位置上给出“-”。给出的数据间用一个空格分隔。

  接着一行是一个非负整数N2，表示第2棵树的结点数；

  随后N行同上描述一样，依次对应二叉树的N个结点。

  对于20%的数据，有0<N1=N2<=10

  对于40%的数据，有0<=N1=N2<=100

对于100%的数据，有0<=N1,N2<=10100

注意：题目不保证每个结点中存储的字母是不同的。

- 输出：
- 共三行。

  第一行，如果两棵树是同构的，输出“Yes”，否则输出“No”。

  后面两行分别是两棵树的深度。

2.1.3 数据结构设计

- 首先判断两个树的深度，当深度不一样时直接判断为不同构
- 对于每一个节点以及这个节点的子树情况进行判断，进行递归调用
- 对于一个位置的节点，在深度相同的情况下，如果有一个为空（证明另外一个也为空），则同构
- 对于一个位置的节点，当其存储的字母不同时，判断为不同构
- 对于有相同存储字母的同一位置节点，如果它是同构，则有以下情况
  - 左子树同构且右子树同构
  - 左右子树互换之后左子树同构且右子树同构同构
- 针对这两种情况进行递归调用即可

节点结构设计：

```
struct Node {
    int lc;
    int rc;
    char ltr;
};
```

2.1.4 功能说明

- 求树的深度:findDepth()

```
int findDepth(int root, Node tree[])
{
    /*root 根节点
    /*tree 结构体数组，相当于是一棵树*/
    /*功能：返回树的深度，作为后续同构判断的与之条件*/
    if (root == -1) return 0;
    queue<int> q;
    int level = 0;
    q.push(root);
    while (!q.empty())
    {
        int len = q.size();
        level++;
        while (len--)
        {
            int temp = q.front();
```

```
            q.pop();
            if (tree[temp].lc != -1) q.push(tree[temp].lc);
            if (tree[temp].rc != -1) q.push(tree[temp].rc);

        }
    }
    return level;

}
```

> 利用bfs求树的深度

- 建树函数builder()
- 关于取根节点：输入子节点并标记，最后整个数组没有被标记的就是根节点

```
int builder(Node T[])
{
    /*Node T[] 节点结构体数组*/
    /*功能：建立树（以结构体数组的形式），并返回根节点的下标*/
    cin >> n;

    if (!n)
    {
        return -1;
    }

    if (n) // 非空
    {
        for (int i = 0; i < n; i++) ck[i] = 0;

        for (int i = 0; i < n; i++)
        {
            cin >> T[i].ltr;
            cin >> cl;
            //读入左孩子
            if (cin.good())
            {
                T[i].lc = cl;
                ck[T[i].lc] = 1;  //将此孩子节点标记为访问过
            }
            else //当输入为'-'的时候就会输入错误，记为空
            {
                T[i].lc = -1;
                cin.clear();
            }

            cin >> cr;
            //读入右孩子
            if (cin.good())
            {
                T[i].rc = cr;
                ck[T[i].rc] = 1;
            }
```

```
            else
            {
                T[i].rc = -1;
                cin.clear();
            }
        }
        //ck数组记录的是节点是否被访问过
        //被访问的都是相对于上一层的孩子节点
        //所以最后剩下的没有被访问过的节点就是根节点
        for (int i = 0; i < n; i++)
        {
            if (!(ck[i]))
            {
                Root = i;
                break;
            }

        }
        return Root;
    }

}
```

> 注意在输入时不能用char类型来接子节点的位置再转换成int类型，因为char本身能接的数在做减法的情况下（-'0'）智能表示1~9，而孩子节点很可能超过两位数

- 判断同构函数tg()

```
int tg(int root1, int root2)
{
    /*root1 root2 两棵树的相应节点*/
    /*功能：判断是否同构*/

    //两个子树均为空(提前判断两棵树的深度相同)
    if (root1 == -1 || root2 == -1)
    {
        return 1;
    }

    //本节点的值不相同，直接判为不同构
    if (tree1[root1].ltr != tree2[root2].ltr)
    {
        return 0;
    }

    //本节点的子树没有交换，分别判断左右子树是否是同构
    int sameSide = tg(tree1[root1].lc, tree2[root2].lc) &&
tg(tree1[root1].rc, tree2[root2].rc);

    //本节点的子树已经做了交换，交换之后判断左右子树知否是同构
    int diffSide = tg(tree1[root1].rc, tree2[root2].lc) &&
tg(tree1[root1].lc, tree2[root2].rc);

    //只要换或者不换二者之一满足左右子树均同构即可
```

```
    return sameSide || diffSide;
}
```

> 主要思想就是判断换和不换的情况都实验，二者之一通过就可以判断是同构

- 主函数调用

```
int main()
{
    //建树
    int root1, root2;
    root1 = builder(tree1);
    root2 = builder(tree2);

    //返回建树的深度
    int size1 = findDepth(root1, tree1);
    int size2 = findDepth(root2, tree2);

    //首先判断深度，再进行同构判断
    if (size1 != size2) cout << "No" << endl;
    else if (tg(root1, root2)) cout << "Yes" << endl;
    else cout << "No" << endl;

    //输出深度
    cout << size1 << endl;
    cout << size2 << endl;

    return 0;
}
```

> 首先进行深度判断可以节省一些递归的时间开销

2.1.5 调试分析

- 关于递归函数：最初尝试是对于当前节点的左右子树考虑情况过于复杂，其中涉及到判断左右子树可能为空，但是这几种情况可以放在递归的下一层进行判断，从而减少了很多非必要的情况判断

  最初尝试的代码：

```
int tg(int root1, int root2)
{
    if (root1 == -1 && root2 == -1)
    {
        return 1; //均空，同构
    }

    if (root1 == -1 && root2 != -1 || root1 != -1 && root2 == -1)
    {
        return 0;  //一棵树为空，不
    }
    if (tree1[root1].ltr != tree2[root2].ltr)
    {
```

```
        return 0;  //树根的元素不同
    }
    if ((tree1[root1].lc == -1) && (tree2[root2].lc == -1))
    {
        return tg(tree1[root1].rc, tree2[root2].rc);  // 左子树都是空，比较右子树
    }
    if ((tree1[root1].lc != -1) && (tree2[root2].lc != -1) && (tree1[tree1[root1].lc].ltr == tree2[tree2[root2].lc].ltr))  //左孩子都有，且值相等，这是不交换，继续向下比较
    {
        return (tg(tree1[root1].lc, tree2[root2].lc) && tg(tree1[root1].rc, tree2[root2].rc));
    }

    else
    {
        return (tg(tree1[root1].rc, tree2[root2].lc) && tg(tree1[root1].lc, tree2[root2].rc));  //剩下情况均需要交换
    }

}
```

> 直接将左右子树进行对换或者不对换会大大简化递归逻辑

- 对于递归可以用栈的方式减少时间开销（针对防止TLE的情况）

```
int tg(int root1, int root2)
{
    queue<pair<int, int>> q;
    q.push({ root1, root2 });
    int flag = 1;

    while (!q.empty())
    {
        int r1 = q.front().first;
        int r2 = q.front().second;
        q.pop();

        if ((r1 == -1) && (r2 == -1))
        {
            continue;
        }
        if ((r1 != -1) && (r2 != -1) && tree1[r1].ltr == tree2[r2].ltr)
        {
            if ((tree1[r1].lc == -1) && (tree2[r2].lc == -1))
            {
                q.push({ tree1[r1].rc, tree2[r2].rc });
                continue;
            }
            if ((tree1[r1].lc != -1) && (tree2[r2].lc != -1) && (tree1[tree1[r1].lc].ltr == tree2[tree2[r2].lc].ltr))
            {
                q.push({ tree1[r1].lc , tree2[r2].lc });
                q.push({ tree1[r1].rc , tree2[r2].rc });
```

```
                continue;
            }

            else
            {
                q.push({ tree1[r1].lc , tree2[r2].rc });
                q.push({ tree1[r1].rc , tree2[r2].lc });
                continue;
            }
        }
        return 0;
    }
    return flag;

}
```

> 可能由于这种判断会将一些情况漏判断掉，所以在进行递归判断时应尽可能的将情况进行简化，防止情况重叠或者漏掉情况的问题

2.1.6 总结和体会

- 对于递归情况要尽可能的简化判断条件，防止情况重叠或者漏掉情况的问题；
- 可以通过将这一层需要判读的情况放到下一层递归来简化递归逻辑（判断条件）
- 在接数据时要充分考虑到数据的范围，选择合适的变量来进行存储
- 考虑全递归的出口有几个，分别是什么条件

2.2 二叉树的非递归遍历

2.2.1 问题描述

二叉树的非递归遍历可通过栈来实现。例如对于由abc##d##ef###先序建立的二叉树，如下图1所示，中序非递归遍历（参照课本p131算法6.3）可以通过如下一系列栈的入栈出栈操作来完成：push(a) push(b) push(c) pop pop push(d)pop pop push(e) push(f) pop pop。

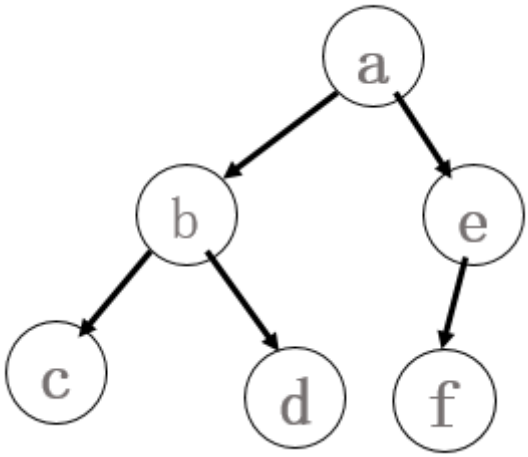


图1

如果已知中序遍历的栈的操作序列，就可唯一地确定一棵二叉树。请编程输出该二叉树的后序遍历序列。

提示：本题有多种解法，仔细分析二叉树非递归遍历过程中栈的操作规律与遍历序列的关系，可将二叉树构造出来。

2.2.2 基本要求

- 输入
  - 第一行一个整数n，表示二叉树的结点个数。
    接下来2n行，每行描述一个栈操作，格式为：push X 表示将结点X压入栈中，pop 表示从栈中弹出一个结点。
    (X用一个字符表示)
  - 对于20%的数据，0<n<=10
    对于40%的数据，0<n<=20
    对于100%的数据，0<n<=83
- 输出
  - 一行，后序遍历序列。

2.2.3 数据结构设计

- 自定义节点类

```
class Node
{
public:
    Node(char d, Node* l=NULL, Node* r=NULL)  //有参构造默认孩子节点是空指针
    {
        data = d;
        lc = l;
        rc = r;
    }

    char data;  //存储的数据
    Node* lc;  //左孩子指针
    Node* rc;  //右孩子指针
};
```

> 注意在定义节点时，要考虑到初始情况，子节点均为空的情况

- 存储左子树没有遍历完的节点的指针

```
stack<Node*> st;  //记录左子树还没有遍历完的节点
```

2.2.4 功能说明

- 建立树并后续遍历输出
- 建树
  - 因为在深度优先搜索的过程中，是先遍历完左子树后才会将这个节点出栈，再遍历右子树，所以需要用指针记录父节点（左子树还没有遍历完的节点），来为这个结点插入右子树进行准备
  - 用栈来记录还没有遍历完左子树的节点的指针，下一次当某个节点的左子树已经遍历完了（有left标记）再进行插入的时候，下一步插入就在栈顶出栈的指针所在位置进行插入

- 建树函数buildTree()

```
void buildTree(int n, Node* root)
{
    /*n : 节点数
    root : 作为建立的树的根节点
    */

    Node* pointer = root;

    bool leftAva = true;

    while (n)
    {

        cin >> opt;
        if (opt == "pop")
        {
            if (leftAva) leftAva = false;  //仅当叶子节点会出现左边没有遍历完就要pop的情况

            else
            {
                if (st.empty()) continue;

                pointer = st.top();  //回退到上一个左子树还没有遍历完的那个节点
                st.pop();
            }
        }

        else
        {
            cin >> dt;
            st.push(pointer);
            //首先插左边，之后看右边能不能插
            if (leftAva)
            {
                pointer->lc = new Node(dt);
                pointer = pointer->lc;
            }

            else
            {
                pointer->rc = new Node(dt);
```

```
                pointer = pointer->rc;

                st.pop();  //能插右边节点，说明这个节点已经满了，这个点之前还被push进栈了，所以这个时候要pop出去
            }

            leftAva = true;  //右子树可以插左孩子
            n--;

        }
    }


}
```

> 注意在进行插入的时候要注意先插入左子树，再插入右子树，并且在插入右子树的时候需要pop，因为此时根节点又一次被添加到了栈里边
>
> 在过程中注意修改left判断值 true代表当前指针所指节点左孩子还可以进行插入
>
> 遍历n个节点，一共n次

- 后序遍历backTranverse()

```
void backTranverse(Node* root)
{
    if (root->lc) backTranverse(root->lc);
    if (root->rc) backTranverse(root->rc);
    cout << root->data;
}
```

> 正常后续遍历输出即可，先遍历左节点，再遍历右节点，最后输出自身

- 主函数调用

```
int main()
{
    int n;
    cin >> n;

    cin >> opt >> dt;  //初始建立根节点

    Node* root = new Node(dt);  //准备根节点

    buildTree(n - 1, root);不用再遍历根节点，所以还需要遍历n-1次

    backTranverse(root); //后序遍历输出


    return 0;
}
```

2.2.5 调试分析

- 尝试过程中曾经尝试对于给出序列是否可以不通过建树而通过查找关于dfs和后序遍历之间的直接关系，但是发现不同子树之间存在互相嵌套的关系，且也无法确定某节点的左子树是否是完全遍历过的，无法继续研究
- 在建树过程中，将插入左节点与右节的顺序颠倒，导致插入不正确。在深度优先白能力中，最先白能力的就是左节点，所以在建树的过程中要先进行左节点的插入再进行右节点的插入
- 在进行pop的过程中首先要注意如果左节点可以插入时这证明此节点是叶子节点，对于叶子节点的处理是直接将指针返回到上一个左子树还没有遍历完的节点，并把左侧能否插入节点的标志改为false

2.2.6 总结和体会

- 建树是最为简单直接的方法，建立之后可以快速进行后序遍历，输出后序序列
- 对于小规模数据，可以用线性表进行树的模拟，创建结构体数组，结构体中存储这个节点的左指针与右指针，来对树继续模拟

2.3 表达式树

2.3.1 问题描述

任何一个表达式，都可以用一棵表达式树来表示。例如，表达式a+b*c，可以表示为如下的表达式树：

```
   +
  / \
 a   *
    / \
    b c
```

现在，给你一个中缀表达式，这个中缀表达式用变量来表示（不含数字），请你将这个中缀表达式用表达式二叉树的形式输出出来。

2.3.2 基本要求

- 输入
  - 输入分为三个部分。
    第一部分为一行，即中缀表达式(长度不大于50)。
    中缀表达式可能含有小写字母代表变量（a-z），也可能含有运算符（+、-、*、/、小括号），
    不含有数字，也不含有空格。
    第二部分为一个整数n(n <= 10)，表示中缀表达式的变量数。
    第三部分有n行，每行格式为C　x，C为变量的字符，x为该变量的值。
    对于20%的数据，1<=n<=3，1<=x<=5；
    对于40%的数据，1<=n<=5，1<=x<=10；
    对于100%的数据，1<=n<=10，1<=x<=100；
- 输出

- 输出分为三个部分，第一个部分为该表达式的逆波兰式，即该表达式树的后根遍历结果。占一行。
- 第二部分为表达式树的显示，如样例输出所示。
- 如果该二叉树是一棵满二叉树，则最底部的叶子结点，分别占据横坐标的第1、3、5、7……个位置（最左边的坐标是1），
  然后它们的父结点的横坐标，在两个子结点的中间。
- 如果不是满二叉树，则没有结点的地方，用空格填充（但请略去所有的行末空格）。
- 每一行父结点与子结点中隔开一行，用斜杠（/）与反斜杠（\）来表示树的关系。
- /出现的横坐标位置为父结点的横坐标偏左一格，\出现的横坐标位置为父结点的横坐标偏右一格。
- 也就是说，如果树高为m，则输出就有2m-1行。
  第三部分为一个整数，表示将值代入变量之后，该中缀表达式的值。需要注意的一点是，除法代表整除运算，即舍弃小数点后的部分。
  同时，测试数据保证不会出现除以0的现象。

2.3.3 数据结构设计

- 节点类

```
class Node
{
public:
    //Node();
    Node(int sub=-1, char cq=' ', int le = -1, int ri = -1, int val = 0)
    {
        this->subid = sub;
        this->c = cq;
        this->l = le;
        this->r = ri;
        this->val = val;
    }
    int subid;    //此节点在原表达式中的下标
    char c;  //存储元素
    int l;  //左孩子
    int r;  //右孩子
    int val;  //运算表达式值

};
Node tree[N];
```

> 这里有两个特殊量：
>
> - subid：此节点在原数组中的下标位置，便于用线性表对树进行模拟
> - val：表达式的值（在建树过程中同时进行表达式的运算），运算符号中也会存表达式的值作为过程量

- 符号优先级映射

```
map<char, int>symblab;

int main()
{
    symblab.insert({ '+', 1 });
    symblab.insert({ '-', 1 });
    symblab.insert({ '*', 2 });
    symblab.insert({ '/', 2 });
    symblab.insert({ '(', 0 });    //对于左括号右括号需要进行特殊判断，所以优先级并不紧要
    symblab.insert({ ')', 0 });
    ...
    ...
}
```

- 变量与值的映射

```
map<char, int> mp;

 for (int i = 1; i <= vbnum; i++)
 {
     char vb;
     int n;
     cin >> vb >> n;
     mp.insert({ vb, n });
 }
```

2.3.4 功能说明

- 计算函数doMath()

```
int doMath(stack<Node>& num, char sym, Node& fa)
{
    /*
    num:符号栈
    sym：运算符
    fa:符号栈栈顶的的节点
    功能：返回运算结果到符号节点的值中，并且将父节点的左右孩子指针与运算数栈中前两个进行连接
    */
    if (sym == '+')
    {
        int num1 = num.top().val;
        fa.r = num.top().subid;
        num.pop();
        int num2 = num.top().val;
        fa.l = num.top().subid;
        num.pop();
        return num1 + num2;
    }
    else if (sym == '-')
    {
```

```
        int num1 = num.top().val;
        fa.r = num.top().subid;
        num.pop();
        int num2 = num.top().val;
        fa.l = num.top().subid;
        num.pop();
        return num2 - num1;
    }
    else if (sym == '*')
    {
        int num1 = num.top().val;
        fa.r = num.top().subid;
        num.pop();
        int num2 = num.top().val;
        fa.l = num.top().subid;
        num.pop();
        return num1 * num2;
    }
    else if (sym == '/')
    {
        int num1 = num.top().val;
        fa.r = num.top().subid;
        num.pop();
        int num2 = num.top().val;
        fa.l = num.top().subid;
        num.pop();
        return num2 / num1;
    }
    else
        return -1;
}
```

> 注意要同时运算和连接左右孩子
>
> 注意传引用的符号栈

- 建树函数buildTree()

```
int buildTree(string s)
{
    /*s:表达式字符串
    功能：建树并计算表达式的值并返回根节点在原树结构体数组中的下标

    */
    for (int i = 0; i < s.size(); i++)
    {
        Node node = Node(i, s[i]);
        node.c = s[i];
        node.subid = i;
        tree[i] = node;

        //是数
        if (s[i] >= 'a' && s[i] <= 'z')
        {
```

```
            node.val = mp[s[i]];
            numst.push(node);
        }

        //是符号
        else
        {
            //栈空和括号进行特殊判断
            if (symst.empty() || s[i] == '(')
            {
                symst.push(node);
            }
            //运算符优先级关系出栈
            else if (symblab[symst.top().c] < symblab[s[i]])
            {
                symst.push(node);
            }
            else
            {
                //特殊判断是不是右括号
                if (s[i] == ')')
                {
                    while (symst.top().c != '(')
                    {
                        Node nd = symst.top();
                        symst.pop();
                        nd.val = doMath(numst, nd.c, nd);
                        tree[nd.subid] = nd;
                        numst.push(nd);
                    }
                    symst.pop();
                }

                //运算符优先级出栈
                else
                {
                    while (!symst.empty() && (symblab[symst.top().c] >=
symblab[s[i]]))
                    {
                        Node nd = symst.top();
                        symst.pop();
                        nd.val = doMath(numst, nd.c, nd);
                        tree[nd.subid] = nd;
                        numst.push(nd);
                    }
                    symst.push(node);
                }
            }
        }
    }

    //栈不空继续出栈
    while (!symst.empty())
    {
```

```
        Node nd = symst.top();
        symst.pop();
        nd.val = doMath(numst, nd.c, nd);
        tree[nd.subid] = nd;
        numst.push(nd);
    }

    //最后的节点中存储的值就是最后表达式运算的值，存储的节点位置就是根节点在原树的结构体数组中的下标
    root = numst.top().subid;

    //返回的是根节点在原树结构体数组中的下标
    return numst.top().val;

}
```

> 很正常的表达式四则运算，但是同时要注意在过程中进行表达式值的运算，同时设置好节点的左右子节点
>
> 、

- 后序遍历表达式

```
void backTranverse(int pos)
{
    /*pos 根节点在结构体数组中位置*/
    /*功能：返回后续表达式（逆波兰式）*/
    if (tree[pos].l != -1)
    {
        backTranverse(tree[pos].l);
    }
    if (tree[pos].r != -1)
    {
        backTranverse(tree[pos].r);
    }
    cout << tree[pos].c;
}
```

- 求树的深度(在树的展示图中需要)

```
int depth(int root1)
{
    /*root 根节点*/
    /*功能：返回树的深度*/
    int left = 0, right = 0;
    if (tree[root1].l != -1) left = depth(tree[root1].l);
    if (tree[root1].r != -1) right = depth(tree[root1].r);
    return ((left >= right) ? left : right) + 1;
}
```

> 比较左右子树的深度，选择最大的深度返回

- 计算幂次（在树的展示图计算坐标时用到）

```
int fps(int a, int p)
{
    /*快速幂的方法进行求幂*/
    int ans = 1;
    while (p)
    {
        if (p & 1) ans *= a;;

        a *= a;
        p >>= 1;
    }
    return ans;
}
```

> 以幂次二分来使得求幂的过程时间复杂度变为logn

- 填充画图矩阵

```
void bfs(int h)
{
    /*h 树的深度*/
    /*功能：填充展示矩阵*/
    r_x = 0;
    r_y = (fps(2, h) - 1) / 2;

    queue<Node> q;
    queue<pair<int, int>> cod;
    q.push(tree[root]);
    cod.push({ r_x, r_y });

    //bfs
    while (!q.empty())
    {
        Node nd = q.front();
        q.pop();
        pair<int, int> pr = cod.front();
        cod.pop();

        int t_x = pr.first;
        int t_y = pr.second;

        //检测当前位置的头顶上是左还是右节点的边，分别进行偏移量的计算
        if (t_x != 0 && mat[t_x - 1][t_y] == '/')
        {
            t_y -= (fps(2, h - t_x / 2 - 1) - 1);
        }
        else if (t_x != 0 && mat[t_x - 1][t_y] == '\\')
        {
            t_y += (fps(2, h - t_x / 2 - 1) - 1);
        }


        mat[t_x][t_y] = nd.c;

        if (nd.l != -1)  //左节点
```

```
        {
            mat[t_x + 1][t_y - 1] = '/';
            q.push(tree[nd.l]);
            cod.push({t_x + 2, t_y - 1});
        }

        if (nd.r != -1)  //右节点
        {
            mat[t_x + 1][t_y + 1] = '\\';
            q.push(tree[nd.r]);
            cod.push({ t_x + 2, t_y + 1 });

        }
        else  //特殊的要求，在叶子节点处还要将左下右下填充为空格
        {
            mat[t_x + 1][t_y + 1] = ' ';
            mat[t_x + 1][t_y - 1] = ' ';
        }
    }
}
```

> 利用bfs进行矩阵的填充，有很多格式注意事项
>
> - 树的根节点的位置确定
> - 树的不同节点在水平方向的偏移量
> - 节点与边的交替出现
> - 当前节点的左孩子右孩子节点位置计算
> - 叶子节点子节点赋空值

- 主函数调用

```
int printLen[500];  //用于记录每一行输出最多到多少位

int main()
{
    //符号优先级初始化
    symblab.insert({ '+', 1 });
    symblab.insert({ '-', 1 });
    symblab.insert({ '*', 2 });
    symblab.insert({ '/', 2 });
    symblab.insert({ '(', 0 });
    symblab.insert({ ')', 0 });

    //将无关的位置填充为无关字符
    for (int i = 0; i < 500; i++)
    {
        for (int j = 0; j < 500; j++)
        {
            mat[i][j] = 'N';
        }
    }

    cin >> s;
    cin >> vbnum;
```

```
    //载入变量值
    for (int i = 1; i <= vbnum; i++)
    {
        char vb;
        int n;
        cin >> vb >> n;
        mp.insert({ vb, n });
    }

    //求表达式值以及建树
    int res = buildTree(s);

    //求树的深度
    int ht = depth(root);

    //求根节点在输出矩阵中的坐标
    r_x = 0;
    r_y = (fps(2, ht) - 1) / 2;

    //输出后序遍历序列
    backTranverse(root, r_x, r_y);

    //填充输出矩阵
    bfs(ht);

    cout << endl;

    //输出
    for (int i = 0; i < ht * 2 - 1; i++)
    {
        int j = fps(2, ht - 1) * 2 - 1;
        //当第一次遇到有效字符之前，将
        while (mat[i][j] == 'N')
        {
            j--;
        }

        printLen[i] = j;  //记录有效的字符最远到哪一位

        while (j >= 0) //将有效字符左侧的无效字符变为空
        {
            if (mat[i][j] == 'N') mat[i][j] = ' ';
            j--;
        }
    }

    //根据每一行的最远有效字符进行输出
    for (int i = 0; i < ht * 2 - 1; i++)
    {
        for (int j = 0; j <= printLen[i]; j++)
        {
            cout << mat[i][j];
        }
        cout << endl;
```

```
    }

    //输出表达式的值
    cout << endl << res << endl;

    return 0;
}
```

> 这里处理树的输出时需要注意空格的输出，通过记录每一行的最远字符可以实现每一行仅输出到最远有效字符就换行，同时要注意要对相应无效字符进行处理

2.3.5 调试分析

主要是分为建树以及打印树两个部分

- 建树
  - 建树过程中要注意在做运算的过程中一定要将符号栈出栈的节点的左右子树设置好，所以在完成运算时要传入引用，确保这个节点的左右子树连接正确
  - 关于优先级的问题：本题不完全是运算，还涉及到树的输出，所以遇到同级的运算符，与之相等的也要进行先进行运算，确保树的输出正确
- 打印树
  - 树的根节点的位置确定
  - 树的不同节点在水平方向的偏移量
  - 节点与边的交替出现
  - 当前节点的左孩子右孩子节点位置计算
  - 叶子节点子节点赋空值

2.3.6 总结和体会

- 建立表达式树是理解并构建后缀表达式的一种重要方式
- 进一步说明了如果知道树的pop和push顺序（本题中体现为表达式的栈）以及树的一种遍历序列（此为中序序列）就可以构建一棵树
- 格式的要求需要通过文本文件txt来查看，包括空格以及换行和字符间距之类（可利用cmd指令进行文件比较）

2.4 中序遍历线索二叉树

2.4.1 问题描述

练习线索二叉树的基本操作，包括二叉树的线索化，中序遍历线索二叉树，查找某元素在中序遍历的后继结点、前驱结点。

2.4.2 基本要求

- 输入：
  - 2行
    第1行，输入先序序列，内部结点用一个字符表示，空结点用#表示
    第2行，输入一个字符a，查找该字符的后继结点元素和前驱结点元素
  - 接下来包含N组共2N行（N为树种与a相等的所有节点个数）
    每一组按其代表节点的中序遍历顺序排列（即按中序遍历搜索线索树，找到节点即可输出，直到遍历完整棵树）。

- 第2k+3行输出该字符的直接后继元素及该后继元素的RTag，格式为：succ is 字符+RTag
  第2k+4行输出该字符的直接前驱继元素及前驱元素的LTag，格式为：prev is 字符+LTag
  若无后继或前驱，用NULL表示。上述k有（0<=k<N/2）

2.4.3 数据结构设计

- 节点类

```
class Node
{
public:
    Node(const char ch, Node* le=NULL, Node* ri=NULL)
    {
        c = ch;
        l = le;
        r = ri;
    }

    Node* l;
    Node* r;
    char c;

};

Node* root = NULL;  //初始化根节点为空
```

> 初始化根节点为空，防止以后再建立节点的时候操作不统一

2.4.4 功能说明

- 建树函数buildTree()

```
void buildTree(string tree)
{
    stack<Node*> s;
    bool left = false;
    Node* ptr;

    for (int i = 0; i < tree.size(); i++)
    {
        if (tree[i] == '\r') break;

        if (root == NULL)
        {
            root = new Node(tree[i]);
            ptr = root;
            left = true;
        }
        else if (tree[i] == '#')
        {
            if (left) left = false;
            else
            {
                if (s.empty()) break;
                ptr = s.top();
```

```
                s.pop();
            }
        }
        else
        {
            s.push(ptr);

            //首先插左边，之后看右边能不能插
            if (left)
            {
                ptr->l = new Node(tree[i]);
                ptr = ptr->l;
            }

            else
            {
                ptr->r = new Node(tree[i]);
                ptr = ptr->r;

                s.pop();  //能插右边节点，说明这个节点已经满了，这个点之前还被push进栈了，所以这个时候要pop出去
                left = true;  //右子树可以插左孩子
            }


        }
    }
}
```

- 此题建立树的方法与第二题大致相同，但是由于根节点初始化的时候是空指针，所以再建树函数内部进行判断的时候需要对其是不是根节点进行判断
- 本题中的#其实相当于第二题中的pop命令（遇到了叶子节点pop）,字母相当于原来的push指令

- 打印节点的信息（此节点的前驱与后继）

```
void printNode(Node* node, const char ch, char& last_char, bool& last_ltag, bool& print_now, bool& find, string& message)
{
    /*
    node : 本节点
    ch : 要查找的节点的字符
    last_char : 此节点的前驱节点的字符
    last_ltag : 此节点的前驱节点的ltag
    print_now : 打印标识符，在当前找到的节点的前一节点设置，确保是当前节点的前一节点打印，当前节点不打印，当前节点的后继节点打印，通过设置标识符实现不同状态之间的切换
    find : 是否已经找到至少一个目标节点
    message : 前驱/后继的提示信息
    */

    /*

    功能：打印线索树中的节点的前驱和后继
    */
```

```
    if (node->l) printNode(node->l, ch, last_char, last_ltag, print_now,
find, message);

    //此节点是后继节点
    if (print_now)
    {
        //先输出后继。后输出前驱，保证输出顺序
        cout << "succ is " << node->c << (node->r == NULL) << endl;
        cout << message << endl;
        print_now = false;
    }
    //找到目标节点
    if (node->c == ch)
    {
        find = true;
        print_now = true;

        if (last_char == '\0')
        {
            message = "prev is NULL";
        }
        //打印前驱节点
        else
        {
            //不在这里打印时因为其必须保证输出顺序，所以传到递归的下一层进行打印
            message = "prev is " + string(1, last_char) +
to_string(last_ltag);
        }
    }
    //进入下一次前更新char 和 ltag信息
    last_char = node->c;
    last_ltag = node->l == NULL;
    if (node->r) printNode(node->r, ch, last_char, last_ltag, print_now,
find, message);
}
```

> 为保证寻找节点的完整性，按照左节点->此节点->右节点的顺序进行遍历

- 寻找函数finder()

```
void finder(char c)
{
    /*c : 要寻找的字符
    功能：printNode函数没有包含查找到目标节点在线索化的最后一个节点，输出后继为NULL的情
况，同时也没有包含没有找到的情况，本函数添加了这两个补充情况
    */
    char last = '\0';
    string message;
    bool print_now = false, find = false, last_ltag = false;
    printNode(root, c, last, last_ltag, print_now, find, message);
    if (print_now)
```

```
    {
        cout << "succ is NULL" << endl;
        cout << message << endl;
    }
    if (find == false)
        cout << "Not found" << endl;

}
```

- 中序遍历输出

```
void inTranverse(Node* root1)
{

    if (root1->l) inTranverse(root1->l);
    cout << root1->c;
    if (root1->r) inTranverse(root1->r);
}
```

- 主函数调用

```
int main()
{
    string t;
    cin >> t;

    buildTree(t);
    inTranverse(root);
    cout << endl;
    char c;
    cin >> c;
    finder(c);

    return 0;
}
```

2.4.5 调试分析

- 找到并打印所有等于指定字符的元素的前驱、后继节点和对应的tag时，实际上没有建立线索树，但遍历方法和建立线索树的方法一致，也就是在中序遍历中直接输出相应的前驱和后继节点，节省建立线索、遍历线索的时间
- 在尝试中忘记了左节点输出NULL的特判，导致一半测试点有问题，经过添加后解决

2.4.6 总结和体会

- 可以在不同遍历过程中进行相应操作，对线索化之后的遍历进行模拟（其本质就是快速查找到某一前驱或者后继，所以直接进行输出同样也是可以达到目的的）
- 线索化是进行某一遍历之后的记忆化擦奥做，之后再次进行遍历会更加快速一些，因为这时的树在线索化之后在逻辑上相当于变成了一个双向链表

2.5 树的重构

2.5.1 问题描述

- 树木在计算机科学中有许多应用。 也许最常用的树是二叉树，但也有其他的同样有用的树类型。其中一个例子是有序树（任意节点的子树是有序的）。
  每个节点的子节点数是可变的，并且数量没有限制。 一般而言， 有序树由有限节点集合T组成，并且满足：

```
1.其中一个节点置为根节点，定义为root(T);

2.其他节点被划分为若干子集T1,T2,...Tm,每个子集都是一个树.
```

同样定义root(T1),root(T2),...root(Tm)为root(T)的孩子，其中root(Ti)是第i个孩子。节点root(T1),...root(Tm)是兄弟节点。

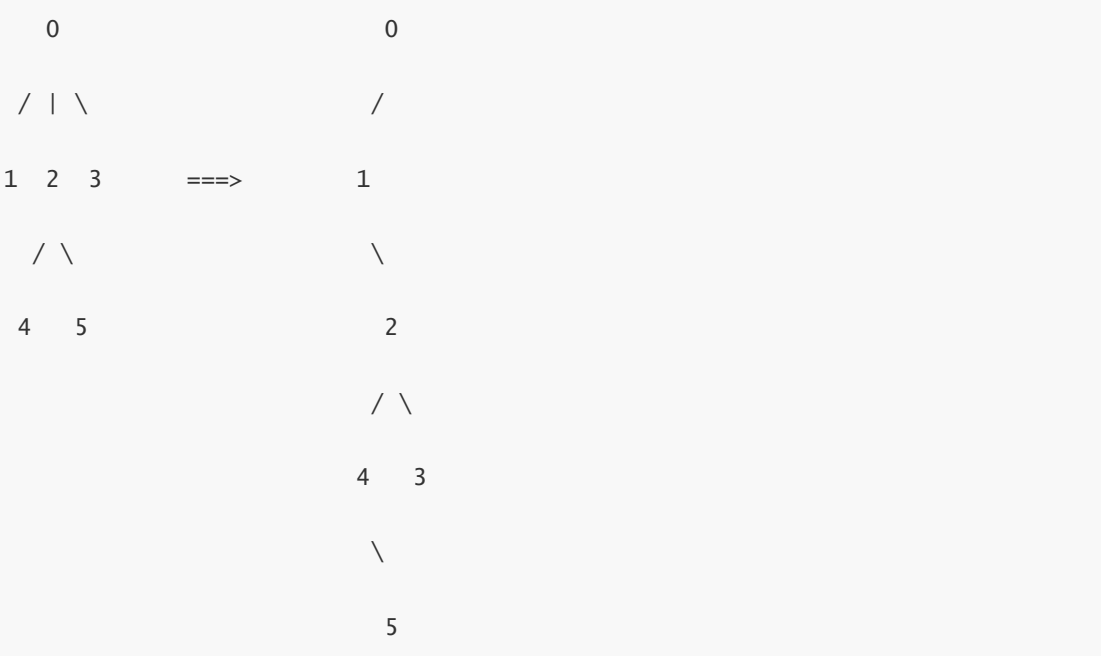
通常将一个有序树表示为二叉树是更加有用的，这样每个节点可以存储在相同内存空间中。有序树到二叉树的转化步骤为：

```
1.去除每个节点与其子节点的边

2.对于每一个节点，在它与第一个孩子节点（如果存在）之间添加一条边，作为该节点的左孩子

3.对于每一个节点，在它与下一个兄弟节点（如果存在）之间添加一条边，作为该节点的右孩子
```

如图所示：



在大多数情况下，树的深度（从根节点到叶子节点的边数的最大值）都会在转化后增加。这是不希望发生的事情因为很多算法的复杂度都取决于树的深度。
现在，需要你实现一个程序来计算转化前后的树的深度。

2.5.2 基本要求

- 输入
  - 输入由多行组成，每一行都是一棵树的深度优先遍历时的方向。其中d表示下行(down)，u表示上行(up)。
    例如上面的树就是dudduduudu,表示从0下行到1,1上行到0,0下行到2等等。输入的截止为以#开始的行。
    可以假设每棵树至少含有2个节点，最多10000个节点。
  - 对每棵树，打印转化前后的树的深度，采用以下格式 Tree t: h1 => h2。其中t表示样例编号(从1开始)，h1是转化前的树的深度，h2是转化后的树的深度
- 输出
  - 对每棵树，打印转化前后的树的深度，采用以下格式 Tree t: h1 => h2。其中t表示样例编号(从1开始)，h1是转化前的树的深度，h2是转化后的树的深度。

2.5.3 数据结构设计

- 根据题目要求，这是一个深度优先遍历的过程，所以可以在模拟深度优先遍历的过程中对于深度记录，进行相应值的动态维护
- 关键点：转化为二叉树的时候，有以下的式子

$$\text{当前节点深度} = \text{父节点深度} + \text{当前节点左兄弟数量}$$

可利用此式子进行dfs递归

2.5.4 功能说明

- dfs寻找转化为二叉树之后的树的深度

```
void dfs(int& i, int fatherdepth)
{
    bdp = max(bdp, fatherdepth);
    int cnt = 0;

    while (s[i])  //遍历当前父节点的全部子节点
    {
        if (s[i] == 'd')  //当此节点为向下走
        {
            dfs(++i, fatherdepth + cnt + 1);   //遍历所有当前父节点的子节点的子树，比较深度
            cnt++;  //统计左兄弟节点数量
        }
        else  //当前节点是向上走,说明此父节点的子节点全部遍历完
        {
            ++i;
            return;
        }
    }
}
```

> 利用dfs进行树的深度遍历，利用当前节点深度=父节点深度+当前节点左兄弟数量进行递归
>
> 每次如果父节点深度（当前节点所在深度）更大一些就更新深度最大值

- 求原来树的深度

```
stack<char> st;
for (int i = 0; i < s.size(); i++)
{
    if (s[i] == 'd')
    {
        st.push(s[i]);
    }

    else
    {
        //pop时比较当前栈的高度是不是更高，是就更新最大值
        int t = st.size();
        h = max(h, t);
        st.pop();

    }
}
```

> 主要想法就是维护栈的最大高度，当出栈时比较大小即可
>
> 因为在栈里的都是向下走的深度，当字符是u的时候出栈说明要向上走了，此时栈里装的向下走的数量就是最大的树的深度

- 主函数调用

```
int main()
{
    int cnt = 0;
    while (1)
    {
        int h = 0;
        cin >> s;

        if (s == "#") break;


        //求原来树的深度
        stack<char> st;
        for (int i = 0; i < s.size(); i++)
        {
            if (s[i] == 'd')
            {
                st.push(s[i]);
            }

            else
            {

                int t = st.size();
                h = max(h, t);
```

```
                st.pop();

            }
        }


        //求转换后的树的深度
        int ptr = 0;
        dfs(ptr, 0);
        cout << "Tree " << ++cnt << ": " << h << " => " << bdp << endl;

        bdp = -1;

    }

    return 0;
}
```

2.5.5 调试分析

- 主要是在递归时要确定如何确定当前节点的深度，关键是利用当前节点深度=父节点深度+当前节点左兄弟数量进行递归

2.5.6 总结和体会

- 将树转化为二叉树的深度，求某节点的深度可以由

  $$当前节点深度 = 父节点深度 + 当前节点左兄弟数量$$

  来确定

3. 实验总结

- 树的建立
  - 可以通过确定前序中序或者后序序列进行建树，或者通过进栈出栈顺序进行建树
  - 更特殊的，可以通过前序序列加上空节点进行建树（相当于给出了进栈出栈的顺序）

- 树的遍历
  - 前中后序遍历
  - 层序遍历
    - 利用bfs实现层序遍历
  - 遍历方式
    - 递归 通过变化递归顺序来进行遍历
      - 事实上可以通过增加一个参数来进行模式的选择

```
void PrintByMode(Node* node, int mode)
{
    if( mode == 1 ) out << c;
    if( node->l ) PrintByMode(node->l, mode);
    if( mode == 2 ) out << c;
    if( node->r ) PrintByMode(node->r, mode);
    if( mode == 3 ) out << c;
}
```

        - 非递归
            - 利用入栈出栈的先后顺序模拟递归
            - 其中前序：打印根节点， 左子树遍历完，出栈完，右子树遍历完，出栈完
            - 中序：左子树遍历完，出栈完，打印根节点， 右子树遍历完，出栈完
            - 后序： 左子树遍历完，出栈完，右子树遍历完，出栈完，打印根节点， 打印根节点（利用指针记录已经访问过的节点或者在节点处增加计数器，记录访问次数，两次就将根节点输出）
    - 树的线索化
        - ltag, rtag
        - 如果当前节点左子树是空，前一个节点的右子树是空， 就将其线索化（在这一个节点与前一个节点之间建立双向指针）

```
void inTheading(Node* p)
{
    if (p)
    {
        inTheading(p->l);

        if (!p->l)
        {
            p->ltag = 1;
            p->l = pre;
        }

        if (pre != NULL && !pre->r)
        {
            pre->rtag = 1;
            pre->r = p;
        }
        pre = p;

        inTheading(p->r);
    }
    //return;
}
```

        - 注意线索化是逻辑结构，不是物理结构，在实际的图表示中，并不是真正的双向链表连接
- 树的转化

    - 转化为二叉树
        - 求二叉树的深度
        - 二叉树节点的深度
- 求树的深度
    - dfs
    - 二叉树的深度
        - $$当前节点深度 = 父节点深度 + 当前节点左兄弟数量$$

- 树的应用
    - 表达式树
    - 树的打印