**实验四**

题目：二叉树的遍历

班级：17级计科一班 姓名：元雨暄 学号：1725111037 完成日期：2018.11.7

**一、需求分析**

(1)输入形式：字符串

输入值的范围：字符范围

(2)输出的形式：字符串表示的二叉树的遍历，凹入表打印二叉树

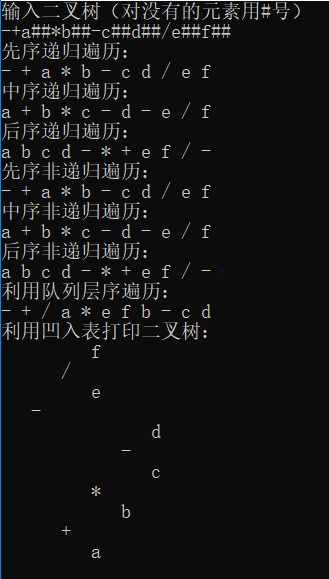
(3)功能：

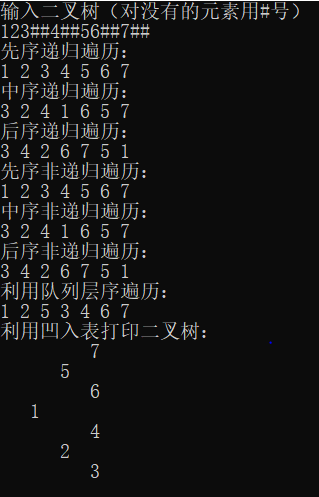
1、以二叉链表为存储结构，实现二叉树的先、中、后三种次序的递归和非递归遍历。

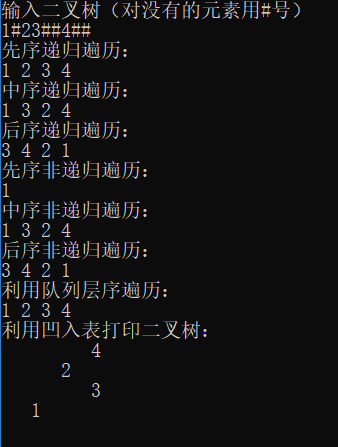
2、借助队列，实现二叉树的层序遍历。

3、按凹入表或树形打印所遍历的二叉树。

(4)测试数据：







**二、概要设计**

(1)抽象数据定义

struct Node //定义树的结点

{

char data;

Node \*lchild,\*rchild;

};

(2)主程序流程

int main()

{

Node \*T;

cout<<"输入二叉树（对没有的元素用#号）"<<endl;

CreateTree(T);

cout<<"先序递归遍历："<<endl;

PreOrderTraverse(T);

cout<<endl;

cout<<"中序递归遍历："<<endl;

InOrderTraverse(T);

cout<<endl;

cout<<"后序递归遍历："<<endl;

PostOrderTraverse(T);

cout<<endl;

cout<<"先序非递归遍历："<<endl;

PreOrderTraverse\_2(T);

cout<<endl;

cout<<"中序非递归遍历："<<endl;

InOrderTraverse\_2(T);

cout<<endl;

cout<<"后序非递归遍历："<<endl;

PostOrderTraverse\_2(T);

cout<<"利用队列层序遍历："<<endl;

q\_Traverse(T);

cout<<endl;

cout<<"利用凹入表打印二叉树："<<endl;

print(T);

return 0;

}

(3)各程序模块之间的调用

主程序模块调用操作函数void CreateTree()，void PreOrderTraverse()，void InOrderTraverse()，Void PostOrderTraverse()，void PreOrderTraverse\_2()，void InOrderTraverse\_2()，Void PostOrderTraverse\_2()，void q\_Traverse(); void print();

**三、详细设计**

(1)数据类型的定义

struct Node

{

struct Node \*lchild,\*rchild;

char data;

};

void createTree(Node \*&T) //创建一棵树，使用递归的方法

{

char c;

cin>>c;

if(c == '#')

{

T=NULL;

}

else

{

T=new Node;

T->data=c;

createTree(T->lchild);

createTree(T->rchild);

}

}

(2)操作的伪码算法

void PreOrderTraverse(Node \*T)

{

if(T)

{

cout<<T->data<<" ";

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

}

void InOrderTraverse(Node \*T)

{

if(T)

{

InOrderTraverse(T->lchild);

cout<<T->data<<" ";

InOrderTraverse(T->rchild);

}

}

void PostOrderTraverse(Node \*T)

{

if(T)

{

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

cout<<T->data<<" ";

}

}

void PreOrderTraverse\_2(Node \*T)

{

stack<Node \*> s;

Node \*p;

s.push(T->rchild);

p=T;

while(!s.empty())

{

while(p->lchild)

{

visit(p->data);

s.push(p->rchild);

p=p->lchild;

}

visit(p->data);

p=s.top();

s.pop();

}

}

void InOrderTraverse\_2(Node \*T)

{

stack<Node \*> s;

Node \*p;

s.push(T);

while(!s.empty())

{

while((p=s.top()) && p)s.push(p->lchild);

p=s.top();

s.pop();

if(!s.empty())

{

p=s.top();

s.pop();

if(!visit(p->data))exit(0);

s.push(p->rchild);

}

}

}

void PostOrderTraverse\_2(Node\* T)

{

if (T == NULL)

return;

stack<Node\*> s;

//pCur:当前访问节点，pLastVisit:上次访问节点

Node\* pCur, \*pLastVisit;

//pCur = root;

pCur = T;

pLastVisit = NULL;

//先把pCur移动到左子树最下边

while (pCur)

{

s.push(pCur);

pCur = pCur->lchild;

}

while (!s.empty())

{

//走到这里，pCur都是空，并已经遍历到左子树底端(看成扩充二叉树，则空，亦是某棵树的左孩子)

pCur = s.top();

s.pop();

//一个根节点被访问的前提是：无右子树或右子树已被访问过

if (pCur->rchild == NULL || pCur->rchild == pLastVisit)

{

cout << pCur->data<< " ";

//修改最近被访问的节点

pLastVisit = pCur;

}

/\*这里的else语句可换成带条件的else if:

else if (pCur->lchild == pLastVisit)//若左子树刚被访问过，则需先进入右子树(根节点需再次入栈)

因为：上面的条件没通过就一定是下面的条件满足。仔细想想！

\*/

else

{

//根节点再次入栈

s.push(pCur);

//进入右子树，且可肯定右子树一定不为空

pCur = pCur->rchild;

while (pCur)

{

s.push(pCur);

pCur = pCur->lchild;

}

}

}

cout << endl;

}

void q\_Traverse(Node \*T)

{

Node \*p;

queue<Node \*> q;

if(T!=NULL)

{

q.push(T);

while(!q.empty())

{

p=q.front();

q.pop();

cout<<p->data<<" ";

if(p->lchild!=NULL)q.push(p->lchild);

if(p->rchild!=NULL)q.push(p->rchild);

}

}

}

void print(Node \*T) //凹入表打印

{

void printHelp(Node\* T,string ss);

string ss="";

printHelp(T,ss);

}

void printHelp(Node\* T,string ss) //凹入表打印树结构

{

if(T==NULL)

return;

ss+=" ";

printHelp(T->rchild,ss); //递归调用函数打印左右孩子

cout<<ss<<T->data<<endl;

printHelp(T->lchild,ss);

}

(3)函数调用关系图

Int main()

PreOrderTraverse()InOrderTraverse()PostOrderTraverse()PreOrderTraverse\_2()InOrderTraverse\_2()PostOrderTraverse\_2()

CreateTree()

q\_Traverse() print()

**四、调试分析**

(1)调试分析中遇到的问题：比如写后序遍历的时候没有考虑到某些结点的值可能因为总是符合条件而使得程序进入无限循环

解决方法：使用一个指针指向最近出栈的元素，这样不需要ppt中的一个数组就可以实现防止一个结点的右孩子被不断访问。

(2) 时间复杂度分析：设n为结点个数

void CreateTree() O(n)

void PreOrderTraverse() O(n)

void InOrderTraverse() O(n)

Void PostOrderTraverse() O(n)

void PreOrderTraverse\_2() O(n)

void InOrderTraverse\_2() O(n)

Void PostOrderTraverse\_2() O(n)

void q\_Traverse() O(n)

void print() O(n)

空间复杂度分析：

void CreateTree() O(n)

void PreOrderTraverse() O(n)

void InOrderTraverse() O(n)

Void PostOrderTraverse() O(n)

void PreOrderTraverse\_2() O(n)

void InOrderTraverse\_2() O(n)

Void PostOrderTraverse\_2() O(n)

void q\_Traverse() O(n)

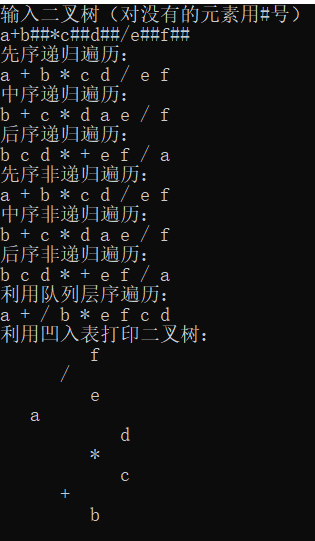
void print() O(n)

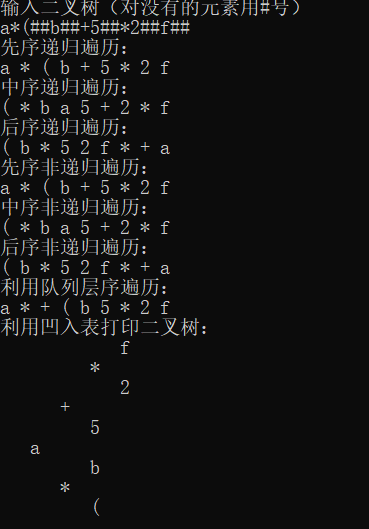
**五、用户使用说明**

根据界面显示的文字提示，利用二叉树的前缀形式输入相应字符，对于叶子结点这类没有孩子的需要输入#号来表示无孩子，缺少一个就输入一个#号，再按下回车键即可。

之后界面上会依次显示输入的二叉树的先序中序以及后序递归遍历，先序中序以及后序非递归遍历，利用队列的层序遍历以及使用凹入表打印二叉树。

**六、测试结果**





**七、附录**

源程序代码：

#include<iostream>

#include<stack>

#include<queue>

using namespace std;

struct Node

{

char data;

Node \*lchild,\*rchild;

};

bool visit(char m)

{

cout<<m<<" ";

return true;

}

void CreateTree(Node \*&T)

{

char a;

cin>>a;

if(a =='#') T=NULL;

else

{

T=new Node;

T->data=a;

CreateTree(T->lchild);

CreateTree(T->rchild);

}

}

void PreOrderTraverse(Node \*T)

{

if(T)

{

cout<<T->data<<" ";

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

}

void InOrderTraverse(Node \*T)

{

if(T)

{

InOrderTraverse(T->lchild);

cout<<T->data<<" ";

InOrderTraverse(T->rchild);

}

}

void PostOrderTraverse(Node \*T)

{

if(T)

{

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

cout<<T->data<<" ";

}

}

void PreOrderTraverse\_2(Node \*T)

{

stack<Node \*> s;

Node \*p;

s.push(T->rchild);

p=T;

while(!s.empty())

{

while(p->lchild)

{

visit(p->data);

s.push(p->rchild);

p=p->lchild;

}

visit(p->data);

p=s.top();

s.pop();

}

}

void InOrderTraverse\_2(Node \*T)

{

stack<Node \*> s;

Node \*p;

s.push(T);

while(!s.empty())

{

while((p=s.top()) && p)s.push(p->lchild);

p=s.top();

s.pop();

if(!s.empty())

{

p=s.top();

s.pop();

if(!visit(p->data))exit(0);

s.push(p->rchild);

}

}

}

//后序遍历

void PostOrderTraverse\_2(Node\* T)

{

if (T == NULL)

return;

stack<Node\*> s;

//pCur:当前访问节点，pLastVisit:上次访问节点

Node\* pCur, \*pLastVisit;

//pCur = root;

pCur = T;

pLastVisit = NULL;

//先把pCur移动到左子树最下边

while (pCur)

{

s.push(pCur);

pCur = pCur->lchild;

}

while (!s.empty())

{

//走到这里，pCur都是空，并已经遍历到左子树底端(看成扩充二叉树，则空，亦是某棵树的左孩子)

pCur = s.top();

s.pop();

//一个根节点被访问的前提是：无右子树或右子树已被访问过

if (pCur->rchild == NULL || pCur->rchild == pLastVisit)

{

cout << pCur->data<< " ";

//修改最近被访问的节点

pLastVisit = pCur;

}

/\*这里的else语句可换成带条件的else if:

else if (pCur->lchild == pLastVisit)//若左子树刚被访问过，则需先进入右子树(根节点需再次入栈)

因为：上面的条件没通过就一定是下面的条件满足。仔细想想！

\*/

else

{

//根节点再次入栈

s.push(pCur);

//进入右子树，且可肯定右子树一定不为空

pCur = pCur->rchild;

while (pCur)

{

s.push(pCur);

pCur = pCur->lchild;

}

}

}

cout << endl;

}

void q\_Traverse(Node \*T)

{

Node \*p;

queue<Node \*> q;

if(T!=NULL)

{

q.push(T);

while(!q.empty())

{

p=q.front();

q.pop();

cout<<p->data<<" ";

if(p->lchild!=NULL)q.push(p->lchild);

if(p->rchild!=NULL)q.push(p->rchild);

}

}

}

void print(Node \*T) //凹入表打印

{

void printHelp(Node\* T,string ss);

string ss="";

printHelp(T,ss);

}

void printHelp(Node\* T,string ss) //凹入表打印树结构

{

if(T==NULL)

return;

ss+=" ";

printHelp(T->rchild,ss);

cout<<ss<<T->data<<endl;

printHelp(T->lchild,ss);

}

int main()

{

Node \*T;

cout<<"输入二叉树（对没有的元素用#号）"<<endl;

CreateTree(T);

cout<<"先序递归遍历："<<endl;

PreOrderTraverse(T);

cout<<endl;

cout<<"中序递归遍历："<<endl;

InOrderTraverse(T);

cout<<endl;

cout<<"后序递归遍历："<<endl;

PostOrderTraverse(T);

cout<<endl;

cout<<"先序非递归遍历："<<endl;

PreOrderTraverse\_2(T);

cout<<endl;

cout<<"中序非递归遍历："<<endl;

InOrderTraverse\_2(T);

cout<<endl;

cout<<"后序非递归遍历："<<endl;

PostOrderTraverse\_2(T);

cout<<"利用队列层序遍历："<<endl;

q\_Traverse(T);

cout<<endl;

cout<<"利用凹入表打印二叉树："<<endl;

print(T);

return 0;

}