# 数据结构第四周实验报告

2018 软件工程创新班 邰思童 2018204493 / 2019/12/04

## 实验题目

二叉树

## 实验环境

Windows 10

gcc (tdm64-2) 4.8.1

std=c99

## 实验内容

实现一棵二叉树及其三种遍历方式，可以统计叶子节点的个数以及树的深度。 其中遍历方式又有递归与非递归的两种实现方式。

## 算法流程

根据读入的先序遍历序列构造一棵二叉树，在建树的过程中顺便计算出数的深度以及叶子的个数。 节点的定义如下：

typedef struct Node {  
 struct Node \*lc, \*rc;  
 struct Node \*fa;  
 char data;  
  
 int depth, height;  
 int leaves;  
} Node;

其中， depth表示这个节点的深度，即与整棵树的根节点距离的层数。由父节点递推得到。 height表示以这个节点为根节点的子树的高度，即从这个点到这颗子树的叶节点距离的层数。由子节点递归得到。

递归遍历相对简单，重点在非递归的实现上。

非递归中需要自己手动实现一个栈，这里直接使用了第三周实验中实现的栈 stack.h 与 stack.c，只需要include 一个自己写过的头文件，而不是重新实现遍，这也体现了模块化与代码重用的巨大优势。

永远不要重新发明轮子。

### 先序遍历

非递归的先序遍历实现最为简单。

先把根扔到栈中，只要栈不空，就每次取出栈顶元素输出然后依次将其左右儿子扔到栈里。

### 中序遍历

对于一棵子树，把根节点和它所有的左儿子扔到栈里，再依次出栈，当根节点出栈时，把它所有的右儿子都扔到栈里，这里用到了一点递归的思想，但是程序实现上仍然是非递归的。

### 后序遍历

非递归的后序遍历在三种遍历方式中实现起来最为复杂，究其原因，是难以在操作左右子树的时候保存下来根节点的信息（递归的时候系统会保存当前节点的栈帧）。

所以，这里用到两个栈，一个用来保存根的信息，另一个栈处理根的左右节点。

当处理到一个点的时候，把它扔到栈S2中，它的左右孩子依次扔到栈s2中，以此类推，处理完所有节点后，栈S1中保存的就是后序遍历的序列。

## 程序代码

//BinTree.h  
#ifndef BINTREE\_H\_  
#define BINTREE\_H\_  
  
typedef struct Node {  
 struct Node \*lc, \*rc;  
 struct Node \*fa;  
 char data;  
  
 int depth, height;  
 int leaves;  
} Node;  
  
//private  
Node \*newNode(char data, Node \*fa);  
Node \*build();  
  
void inOrder(Node \*root);  
void postOrder(Node \*root);  
void preOrder(Node\*root);  
  
#endif

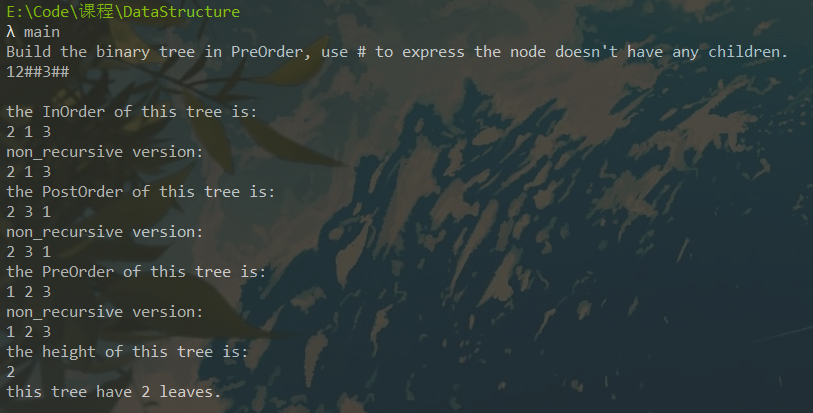
//BinTree.c
  
#include <stdio.h>
  
#include <stdlib.h>
  
   
#include "BinTree.h"
  
#include "stack.h"
  
   
#define MAXSIZE 100
  
#define MAX(a, b) (a) > (b) ? (a) : (b)
  
   
Node \*newNode(char data, Node \*fa) {
  
 Node \*ret = (Node\*)malloc(sizeof(Node));
  
   
 ret->data = data;
  
 ret->lc = ret->rc = NULL;
  
 ret->depth = fa ? fa->depth + 1 : 1;
  
 ret->height = 1;
  
 ret->leaves = 0;
  
   
 ret->fa = fa;
  
   
 return ret;
  
}
  
   
int getHeight(Node \*x) {
  
 return x ? x->height : 0;
  
}
  
   
int getLeaves(Node \*x) {
  
 return x ? x->leaves : 0;
  
}
  
   
Node \*build(Node \*fa) {
  
 char data;
  
 scanf("%c", &data);
  
   
 Node \*ret = NULL;
  
 if (data != '#') {
  
 ret = newNode(data, fa);
  
 ret->lc = build(ret);
  
 ret->rc = build(ret);
  
   
 ret->height += MAX(getHeight(ret->lc), getHeight(ret->rc));
  
 ret->leaves += (getLeaves(ret->lc) + getLeaves(ret->rc));
  
 // printf("data: %c. lc->leaves: %d. rc->leaves :%d\n", ret->data, getLeaves(ret->lc), getLeaves(ret->rc));
  
 } else {
  
 fa->leaves = 1;
  
 }
  
   
 return ret;
  
}
  
   
void inOrder(Node \*root) {
  
 if (root->lc) inOrder(root->lc);
  
 printf("%c ", root->data);
  
 if (root->rc) inOrder(root->rc);
  
}
  
   
void nonrecInOrder(Node \*root) {
  
 Stack \*s = newStack(MAXSIZE);
  
   
 Node \*v = root;
  
 while (v || !empty(s)) {
  
 if (v) {
  
 push(s, v);
  
 v = v->lc;
  
 } else {
  
 Node \*tmp = top(s);
  
 printf("%c ", tmp->data);
  
 pop(s);
  
 v = tmp->rc;
  
 }
  
 }
  
}
  
   
void preOrder(Node \*root) {
  
 printf("%c ", root->data);
  
 if (root->lc) preOrder(root->lc);
  
 if (root->rc) preOrder(root->rc);
  
}
  
   
void nonrecPreOrder(Node \*root) {
  
 Stack \*s = newStack(MAXSIZE);
  
   
 Node \*v = root;
  
 push(s, v);
  
 while (!empty(s)) {
  
 Node \*tmp = top(s);
  
 pop(s);
  
 printf("%c ", tmp->data);
  
   
 if (tmp->rc) push(s, tmp->rc);
  
 if (tmp->lc) push(s, tmp->lc);
  
 }
  
}
  
   
void postOrder(Node \*root) {
  
 if (root->lc) postOrder(root->lc);
  
 if (root->rc) postOrder(root->rc);
  
 printf("%c ", root->data);
  
}
  
   
void nonrecPostOrder(Node \*root) {
  
 Stack \*s1 = newStack(MAXSIZE);
  
 Stack \*s2 = newStack(MAXSIZE);
  
 push(s1, root);
  
   
 while (!empty(s1)) {
  
 Node \*tmp = top(s1);
  
 pop(s1);
  
   
 push(s2, tmp);
  
 if (tmp->lc) push(s1, tmp->lc);
  
 if (tmp->rc) push(s1, tmp->rc);
  
 }
  
   
 while (!empty(s2)) {
  
 printf("%c ", top(s2)->data);
  
 pop(s2);
  
 }
  
}

//main.c
  
#include <stdio.h>
  
   
#include "BinTree.h"
  
   
int main() {
  
 printf("Build the binary tree in PreOrder, use # to express the node doesn't have any children.\n");
  
 Node \*root = build(NULL);
  
   
 printf("\nthe InOrder of this tree is:\n");
  
 inOrder(root);
  
 printf("\nnon\_recursive version:\n");
  
 nonrecInOrder(root);
  
   
 printf("\nthe PostOrder of this tree is:\n");
  
 postOrder(root);
  
 printf("\nnon\_recursive version:\n");
  
 nonrecPostOrder(root);
  
   
 printf("\nthe PreOrder of this tree is:\n");
  
 preOrder(root);
  
 printf("\nnon\_recursive version:\n");
  
 nonrecPreOrder(root);
  
   
 printf("\nthe height of this tree is:\n");
  
 printf("%d", root->height);
  
   
 printf("\nthis tree have %d leaves.\n", root->leaves);
  
 return 0;
  
}

//Stack.h
  
#ifndef STACK\_H
  
#define STACK\_H
  
   
#include "BinTree.h"
  
   
typedef Node\* element\_type;
  
   
typedef struct Stack {
  
 element\_type \*stack; //一个不定长的数组，表示栈本身
  
   
 int top; //栈顶的位置，指向栈顶元素的下一个位置
  
 int capacity; //栈的容量
  
} Stack;
  
   
Stack\* newStack(int size); //构造一个新的栈，初始大小为 size
  
void push(Stack \*s, element\_type x); //向栈顶压入一个元素
  
void pop(Stack \*s); //弹出栈顶元素
  
element\_type top(Stack \*s); //获取栈顶元素的值
  
int empty(Stack \*s); //判断栈是否为空，1表示空，0表示非空
  
int size(Stack \*s); //获取当前栈的大小（栈内的元素个数）
  
   
#endif

//Stack.c
  
#include <stdlib.h>
  
#include <string.h>
  
#include <stdio.h>
  
   
#include "stack.h"
  
#include "BinTree.h"
  
   
typedef Node\* element\_type;
  
   
Stack\* newStack(int size) {
  
 Stack \*ret = (Stack\*)malloc(sizeof(Stack));
  
 ret->capacity = size;
  
 ret->top = 0;
  
 ret->stack = (element\_type\*)malloc(size \* sizeof(element\_type));
  
   
 return ret;
  
}
  
   
int size(Stack \*s) {
  
 return s->top;
  
}
  
   
int empty(Stack \*s) {
  
 return s->top == 0;
  
}
  
   
void push(Stack \*s, element\_type x) {
  
 if (s->top == s->capacity - 1) { //若栈已满，则将栈的空间扩大至原来的两倍
  
 element\_type \*tmp = (element\_type\*)malloc(s->capacity \* 2 \* sizeof(element\_type));
  
 memcpy(tmp, s->stack, sizeof(element\_type) \* s->capacity);
  
   
 s->stack = tmp;
  
 s->capacity \*= 2;
  
 }
  
   
 s->stack[s->top++] = x;
  
}
  
   
void pop(Stack \*s) {
  
 if (empty(s)) printf("The Stack is empty\n");
  
 else {
  
 s->top--;
  
 }
  
}
  
   
element\_type top(Stack \*s) {
  
 if (empty(s)) printf("The Stack is empty\n");
  
 else return s->stack[s->top - 1];
  
}

## 运行结果



## 注意事项

二叉树的实现涉及到一些稍微复杂的指针操作，应该时刻注意由指针引发的内存问题与段错误。