**数据结构实验报告**

姓名：郭宇瑶 学号：U201717031 班级：软工1704班

**实验五 二叉树的构造等问题的求解**

一、实验描述

1.给出一棵二叉树的先序（或后序）遍历结果，以及中序遍历结果，如何构造这棵树？假定遍历结果以数组方式输入，请写出相应函数，判断是否存在生成同样遍历结果的树，如果存在，构造这棵树。

2. 二叉树的层序遍历。使用队列作为辅助存储，按树的结点的深度，从根开始依次访问所有结点。

二、实验设计

1.根据先序遍历+中序遍历的数组或后序遍历+中序遍历的数组构造唯一的二叉树

首先定义二叉树

typedef struct binTreeNode {

ElementType element;

strcut binTreeNode \*left;

struct binTreeNode \*right;

} BinTreeNode;

先序遍历、中序遍历和后序遍历：递归，

对于先序遍历，先打印根节点，再依次对左子树和右子树进行递归遍历；

中序遍历：先对左子树递归，打印根节点，再递归右子树；

后序遍历：先递归左子树、右子树，再打印根节点。

void pre\_order (BinTreeNode \*root) {

if (root == NULL) return;

print ("%c", root->element);

pre\_order (root->left);

pre\_order (root->right);

}

根据遍历结果构造树：根据遍历序列能确定一棵二叉树，要唯一确定一棵二叉树至少需要两种遍历序列 先序+中序 或 后序+中序，（先序+后序无法唯一确定）

给出 先序+中序 或 后序+中序 时：

1. 首先需要定位根节点：先序序列遍历的第一个元素/后序序列遍历的最后一个元素
2. 树根把中序序列分为3部分 {左子树列}树根{右子树列}
3. 以上分别给出了先序/后序序列的一个划分
4. 使用左子树、右子树的遍历结果来重建子树

2.队列的实现：//链表实现队列类型定义

typedef struct LinkQueueNode

{

binTreeNode \*data;

struct LinkQueueNode \*next;

}LKQueNode;

typedef struct LKQueue

{

LKQueNode \*front, \*rear;

}LKQue;

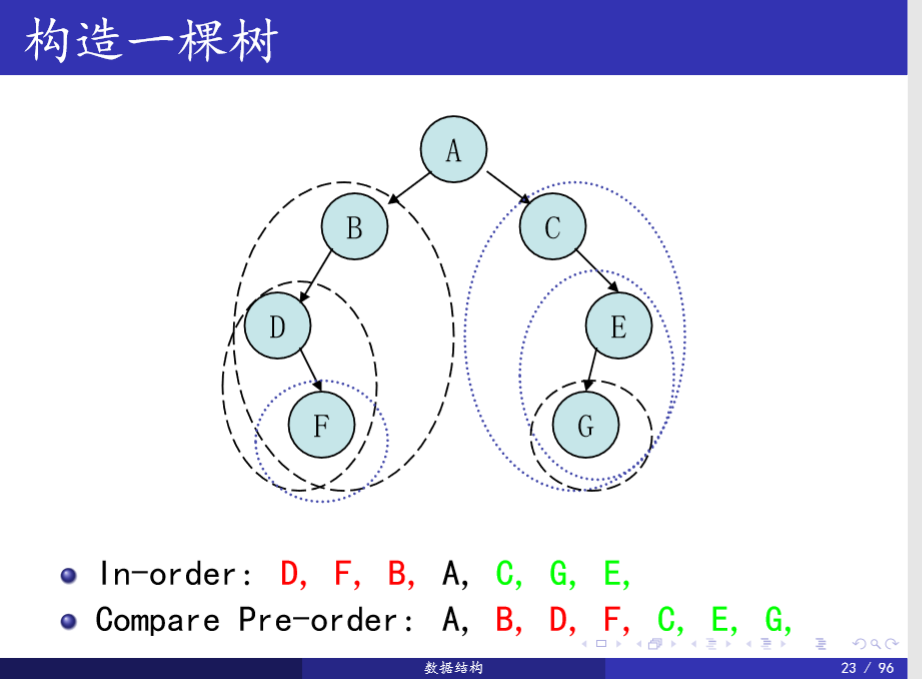
并实现：判断队列是否为空；入队；出队；获取树的高度；获取队列第一个元素等函数

1. 层序遍历

按二叉树从上到下，从左到右依次打印每个节点中存储的数据。

利用队列的实现过程   
(1)首先将二叉树的根节点push到队列中，判断队列不为NULL，就输出队头的元素

1. 判断节点如果有孩子，就将孩子push到队列中，   
   (3)遍历过的节点出队列，   
   (4)循环以上操作，直到Tree == NULL。
2. 测试数据：



//In-order: D, F, B, A, C, G, E, Compare Pre-order: A, B, D, F, C, E, G，节点数n=7

char a[] = { 'A','B','D', 'F', 'C', 'E', 'G' };

char b[] = { 'D', 'F', 'B', 'A', 'C', 'G', 'E' };

int n = 7;

层序遍历期望结果：ABCDEFG

三、实验实现过程

#include<stdio.h>

#include<stdlib.h>

#define MAX\_QUEUE\_SIZE 100

#define ElementType char

#define true 1;

#define false -1;

//定义二叉树结构

typedef struct binTreeNode {

ElementType element;

struct binTreeNode \*left;

struct binTreeNode \*right;

} BinTreeNode;

//链队列类型定义

typedef struct LinkQueueNode

{

binTreeNode \*data;

struct LinkQueueNode \*next;

}LKQueNode;

typedef struct LKQueue

{

LKQueNode \*front, \*rear;

}LKQue;

//初始化队列

void InitQueue(LKQue \*LQ)

{

LKQueNode \*p;

p = (LKQueNode\*)malloc(sizeof(LKQueNode));

LQ->front = p;

LQ->rear = p;

LQ->front->next = NULL;

}

//判断队列是否为空

int EmptyQueue(LKQue \*LQ)

{

if (LQ->front == LQ->rear)

return 1;

else

return 0;

}

//入队列

void EnQueue(LKQue \*LQ, binTreeNode \*x)

{

LKQueNode \*p;

p = (LKQueNode\*)malloc(sizeof(LKQueNode));

p->data = x;

p->next = NULL;

LQ->rear->next = p;

LQ->rear = p;

}

//出队列

int OutQueue(LKQue \*LQ)

{

LKQueNode \*s;

if (EmptyQueue(LQ))

{

exit(0);

return 0;

}

else

{

s = LQ->front->next;

LQ->front->next = s->next;

if (s->next == NULL)

LQ->rear = LQ->front;

free(s);

return 1;

}

}

//取队列首元素

binTreeNode\* GetHead(LKQue \*LQ)

{

LKQueNode \*p;

binTreeNode \*q=NULL;

if (EmptyQueue(LQ))

return q;

else

{

p = LQ->front->next;

return p->data;

}

}

//先序遍历

void pre\_order(BinTreeNode \*root) {

if (root == NULL) return;

printf("%c ", root->element);

pre\_order(root->left);

pre\_order(root->right);

}

//中序遍历

void mid\_order(BinTreeNode \*root) {

if (root == NULL) return;

mid\_order(root->left);

printf("%c ", root->element);

mid\_order(root->right);

}

//后序遍历

void post\_order(BinTreeNode \*root) {

if (root == NULL) return;

post\_order(root->left);

post\_order(root->right);

printf("%c ", root->element);

}

//找到根节点

static int find\_root(char a[], int n, char name) {

int i;

for (i = 0; i < n; i++) { if (a[i] == name) return (i); }

/\* not found \*/ return (-1);

}

//根据先序遍历和中序遍历数组构造二叉树

static binTreeNode \*pre\_and\_mid\_construct(char a[], char b[], int n) {

binTreeNode \*root;

int k, l;

/\* trivial case: tree is empty \*/

if (n == 0) return (NULL);

root = (binTreeNode \*)malloc(sizeof(binTreeNode));

root->element = a[0];

k = find\_root(b, n, a[0]);

if (k < 0) {

printf("node %c not found in in-order traversal\n", a[0]); return (NULL); }

root->left = pre\_and\_mid\_construct(&a[1], b, k);

root->right = pre\_and\_mid\_construct(&a[k + 1], &b[k + 1], n - k - 1);

return (root);

}

//根据后序遍历和中序遍历数组构造二叉树

static binTreeNode \*post\_and\_mid\_construct(char a[], char b[], int n) {

binTreeNode \*root;

int k, l;

/\* trivial case: tree is empty \*/

if (n == 0) return (NULL);

root = (binTreeNode \*)malloc(sizeof(binTreeNode));

root->element = a[n-1];

k = find\_root(b, n, a[n-1]);

if (k < 0) {

printf("node %c not found in in-order traversal\n", a[0]); return (NULL);

}

root->left = post\_and\_mid\_construct(&a[1], b, k);

root->right = post\_and\_mid\_construct(&a[k + 1], &b[k + 1], n - k - 1);

return (root);

}

//访问节点

void Visit(binTreeNode \*p)

{

printf("%c", p->element); //输出是char

}

//树的高度

int height(binTreeNode \*t)

{

int ld, rd;

if (t == NULL)

return 0;

else

{

ld = height(t->left);

rd = height(t->right);

return 1 + (ld>rd ? ld : rd);

}

}

//层次遍历

void LevelOrder(binTreeNode\* bt)

{

LKQue Q;

binTreeNode\* p;

InitQueue(&Q);

if (bt != NULL)

{

EnQueue(&Q, bt);

while (!EmptyQueue(&Q))

{

p = GetHead(&Q);

OutQueue(&Q);

Visit(p);

if (p->left != NULL)

EnQueue(&Q, p->left);

if (p->right != NULL)

EnQueue(&Q, p->right);

}

}

}

void main() {

//In-order: D, F, B, A, C, G, E, Compare Pre-order: A, B, D, F, C, E, G

char a[] = { 'A','B','D', 'F', 'C', 'E', 'G' };

char b[] = { 'D', 'F', 'B', 'A', 'C', 'G', 'E' };

int n = 7;

binTreeNode \*p=pre\_and\_mid\_construct(a, b, n);

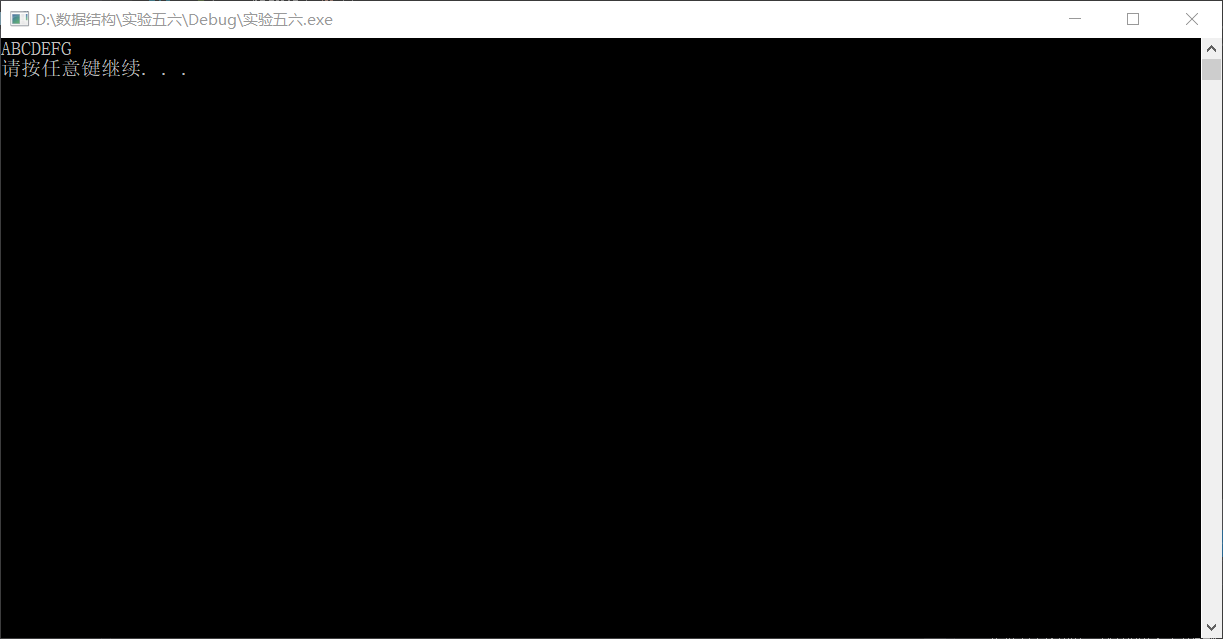
LevelOrder(p);

printf("\n");

system("PAUSE");

}

1. 实验结果



五、实验分析与结论

输出结果与期望值一致。

复杂度分析：

平凡情况: T(0) = 1

递归式: T(n) = T(k) + T(n–k-1) + 非递归运算

最佳情况：只有右子树, T(n) = T(n-1) + O(1),复杂度 O(n)

最坏情况：只有左子树, T(n) = T(n-1) + O(n), 复杂度O(n^2)

平衡情况: T(n) = 2T(n/2) + O(n), 复杂度O(nlogn)