# 数据结构实验报告——实验七（2）

## 学号： 20201060330 姓名： 胡诚皓 得分：\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 一、实验目的

1. 复习二叉树的逻辑结构、存储结构及基本操作；
2. 掌握顺序存储结构下二叉树的创建、遍历；
3. 掌握二叉链表及二叉树的基本操作；
4. 了解二叉树的应用。

### 二、实验内容

1. （必做题）链二叉树的基本实现

假设二叉树中数据元素类型是字符型，请采用二叉链表实现二叉树的以下基本操作：

1. 根据二叉树的先序序列和中序序列构造二叉树；
2. 根据先序遍历二叉树；
3. 根据中序遍历二叉树；
4. 根据后序遍历二叉树；
5. 计算二叉树的叶子数目；
6. 计算二叉树的深度。

测试数据包括如下错误数据：

先序：1234；中序：12345

先序：1234；中序：1245

先序：1234；中序：4231

1. （选做题）main函数的完善与补充

请根据给定main函数，补充完善相关数据结构和功能函数，使主程序能顺利运行。

1. （选做题）哈夫曼树构造

给定n个权值，请构造它们的最优二叉树（赫夫曼树）。

### 三、数据结构及算法描述

1. （必做题）链二叉树的基本实现

### 数据结构

链二叉树的树结点定义与之前相同，此处TElemType以char类型为例，BiTNode中的data作为结点的数据域，lchild、rchild分别作为左右指针域。此外，宏定义了最大结点数MAX\_NODE为100。

在主函数中，用pre[MAX\_NODE]存储输入的前序序列、in[MAX\_NODE]存储输入的中序序列用于二叉树的构造，BiTNode类型的root作为根结点。

### 算法描述

Status constructByPreAndIn(BiTree root, const TElemType \*preOrder, int cur, const TElemType \*inOrder, int in\_start, int in\_end)

以root为根结点，preOrder的第cur个开始为当前前序序列，inOrder的in\_start到in\_end为当前中序序列

①将当前前序序列的第一个值即为当前根结点元素的数据域值

②找到当前根结点值在中序序列中的下标以判断其左右子树中的结点

③根据找到的下标，将当前中序序列分为左右两个部分，用左边部分的递归构建左子树，右边部分的递归构建右子树

void preTraverse(BiTNode root)

①用printf访问当前根结点

②若当前根结点有左子树，以左子结点为根结点递归遍历

③若当前根结点有右子树，以右子结点为根结点递归遍历

void inTraverse(BiTNode root)

①若当前根结点有左子树，以左子结点为根结点递归遍历

②用printf访问当前根结点

③若当前根结点有右子树，以右子结点为根结点递归遍历

void postTraverse(BiTNode root)

①若当前根结点有左子树，以左子结点为根结点递归遍历

②若当前根结点有右子树，以右子结点为根结点递归遍历

③用printf访问当前根结点

int countLeaf(BiTNode root)

①初始化局部变量res为0

②若当前根结点是叶结点，res+1

③若当前根结点有左子树，以左子结点为根结点递归计数并加到res中

④若当前根结点有右子树，以右子结点为根结点递归计数并加到res中

int countDepth(BiTree root)

①初始化左子树深度leftDepth、右子树深度rightDepth都为0

②若当前结点为空结点，返回0，否则到③

③分别计算左右子树的深度，并返回较深的子树的深度+1

1. （选做题）main函数的完善与补充

### 数据结构

二叉树结点的数据类型定义为CBTType，其中数据域data为DATA类型，left、right为指向左右子树的指针域。为了进行层序遍历，定义了队列中元素的类型QElemType为指向CBTType的指针。

此处使用链队列，定义链表的结点QNode，其中pt为QElemType类型的数据域，next为指针域；定义链队列queuePtr，head为链表的头结点，即head->next才是队列的第一个元素，rear指向队列尾。

此题中空结点用空字符表示。

### 算法描述

CBTType \*InitTree()

用于初始化一个二叉树的根结点，并返回指向这个新创建的根结点的指针

void AddTreeNode(CBTType \*treeNode)

在以treeNode为根结点的树中加入结点

①跳过回车符读取要插入结点数据域的值为chr

②若以treeNode为根结点的树为空树，直接把chr赋值给treeNode即可；否则，调用InsertNode往树中插入结点

Status InsertNode(CBTType \*treeNode, DATA ch)

此函数递归地向以treeNode为根结点的二叉树中插入结点且尽量使得该树平衡

①调用TreeDepth（与上题算法相同）获取左右子树的深度leftDepth与rightDepth

②若左子树为空，则初始化新结点并作为当前根结点treeNode的左子结点；若右子树为空，则作为treeNode的右子结点；若左右子树都不为空，转到③

③根据leftDepth与rightDepth，递归地将新结点插入深度较小的子树

Status Enqueue(queuePtr q, QElemType elem)

Status Dequeue(queuePtr q, QElemType \*out)

Status Emptyqueue(queuePtr q)

这三个队列基本操作的函数算法与之前均相同，不再赘述

int TreeDepth(CBTType \*root)

与上题中countDepth函数算法相同

Status TreeIsEmpty(CBTType \*treeNode)

根据treeNode的数据域是否为空字符来判断是否为空树

void TreeNodeData(CBTType \*p)

输出p指向结点的数据域

void delete(CBTType \*node)

使用free将node指向的树结点的空间释放

void DLRTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p))

void LDRTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p))

void LRDTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p))

前中后序遍历，与上题中算法思路相同

void LevelTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p))

①初始化层序遍历要使用的列表，即申请并初始化链队列的头结点

②将当前根结点treeNode入队

③判断队列是否为空，若队列为空，直接退出函数

④出队一个结点存在tmp中，若tmp的左子树不为空，将tmp的左子结点入队；若tmp的右子树不为空，将tmp的右子结点入队

CBTType \*TreeFindNode(CBTType \*treeNode, DATA data)

①初始化pointer作为指向树结点的临时变量

②若当前根结点\*treeNode的值与data相同，直接返回treeNode；否则转到③

③若左子树不为空，则在左子树中递归寻找，将寻找结果存在pointer中，找到则直接返回查找结果，否则转到④

④若右子树不为空，则在右子树中递归寻找，将寻找结果存在pointer中，找到则直接返回查找结果，否则转到⑤

⑤返回NULL

void ClearTree(CBTType \*treeNode)

调用后序遍历LRDTree，并把delete作为操作函数，达到清除整棵树的目的

CBTType \*TreeLeftNode(CBTType \*treeNode)

CBTType \*TreeRightNode(CBTType \*treeNode)

分别返回指向当前结点treeNode的左、右结点的指针

1. （选做题）哈夫曼树构造

### 数据结构

二叉树结点定义为BiTNode，其中TElemType类型（此题中宏定义为int）的data作为数据域，lchild、rchild分别作为指向左、右结点的指针。

由于纯C中没有STL模板库，因此手写小顶堆作为存储优先队列的结构。小顶堆中的各元素为指向二叉树中结点的指针。最后以输出先序序列与中序序列的形式来输出二叉树。

### 算法描述

void downAdjust(BiTNode\* heap[], int start, int end)

在堆heap的[start,end]区间内向下调整结点，以start位置的结点作为调整的起点

①初始化局部变量，target始终为正在调整的结点，min\_child保持指向target的较小子结点

②若min\_child未超出范围，则选出target左右结点中较小的，用min\_child存储其下标；否则退出循环

③若当前调整的结点比较小子结点的值大，交换之并更新target与min\_child，转到②；否则退出循环

void upAdjust(BiTNode\* heap[], int start, int end)

在堆heap的[start,end]区间内向上调整结点，以end位置的结点作为调整的起点

①初始化局部变量，target始终为正在调整的结点，father指向target的父结点

②若father未超出范围，且target比父结点father小，交换之并更新target与father，反复执行

void swap(BiTNode\* \*a, BiTNode\* \*b)

交换辅助函数，交换指针a和b

void createHeap(int n, BiTNode\* heap[])

根据完全二叉树与堆的性质，从n/2开始从后往前倒着往前做添加结点的操作

BiTNode\* deleteTop(int \*n, BiTNode\* heap[])

弹出小顶堆堆顶元素并返回。交换堆中第一个和最后一个元素，将堆元素个数-1，再将交换上来的堆顶向下调整。

void insertElem(BiTNode\* newNode, BiTNode\* heap[], int \*n)

在长度为n的结点指针数组heap中插入新结点的指针newNode。将堆元素个数+1，新元素

放在堆的最后，再将此结点向上调整

void displayData(BiTNode \*node)

根据指向结点的指针node输出其数据域

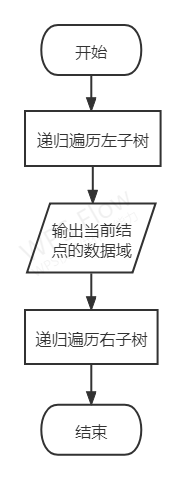
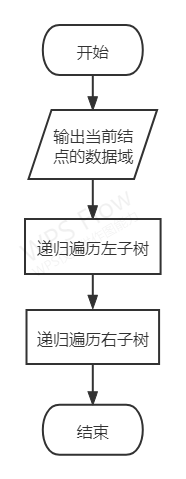
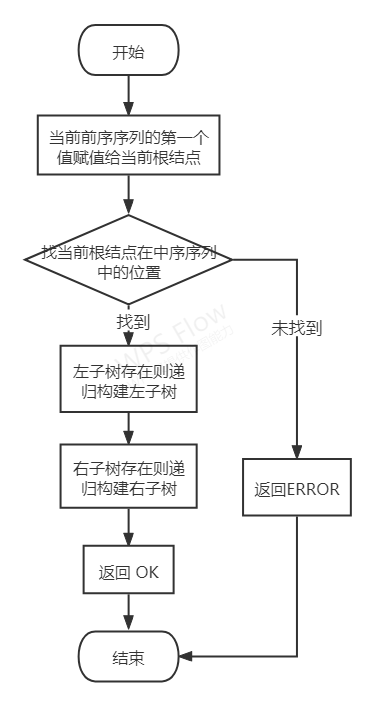
void DLRTree(BiTNode \*treeNode, void TreeNodeData(BiTNode \*))

void LDRTree(BiTNode \*treeNode, void TreeNodeData(BiTNode \*))

前序和中序遍历，与上题中的算法相同

### 详细设计

1. （必做题）链二叉树的基本实现



题8-1

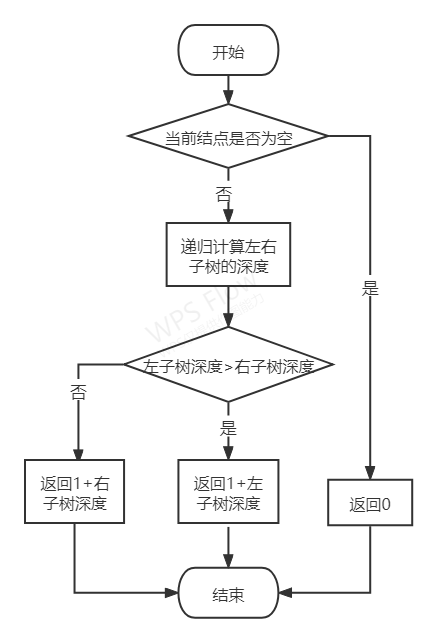
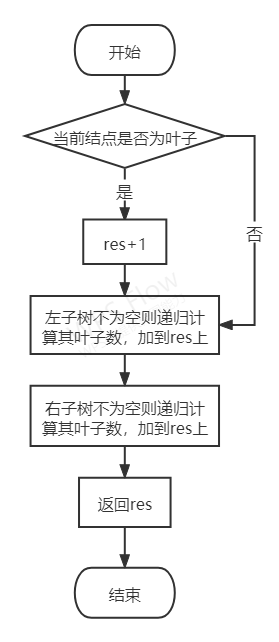
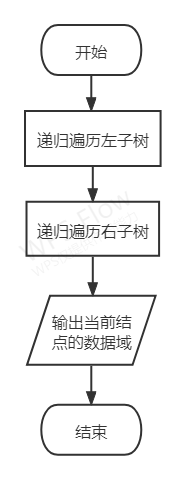
inTraverse函数

题8-1

preTraverse函数

题8-1

constructByPreAndIn函数



题8-1

countDepth函数

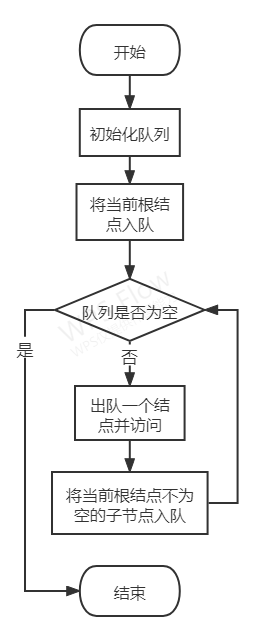
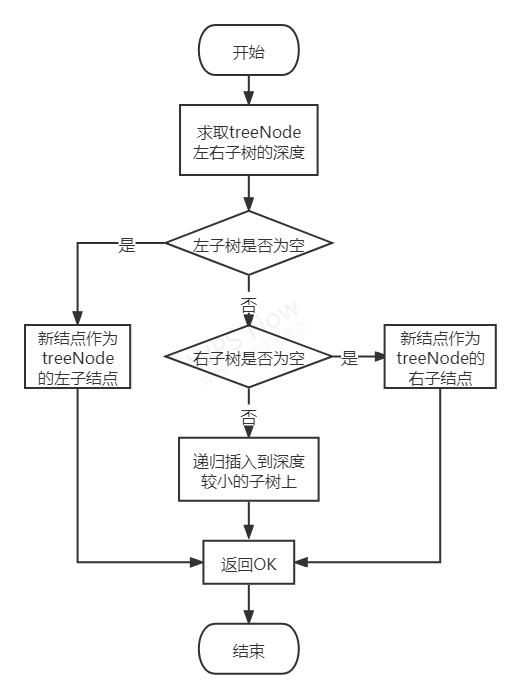
题8-1

countLeaf函数

题8-1

inTraverse函数

1. （选做题）main函数的完善与补充

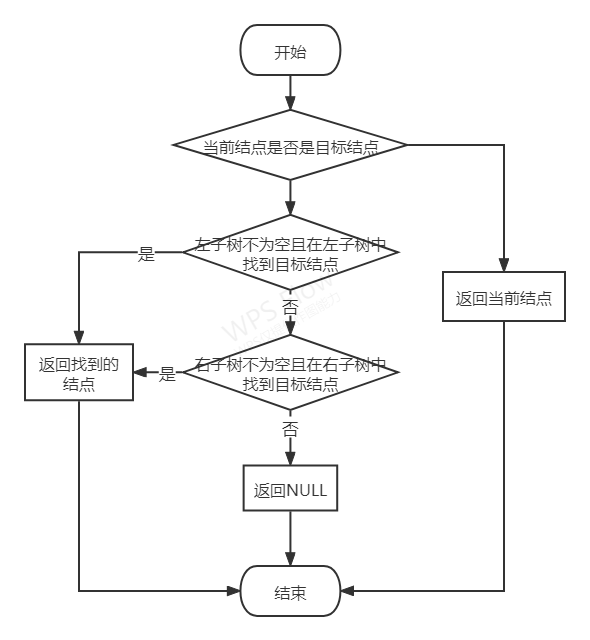


题8-2

InsertNode函数

题8-2

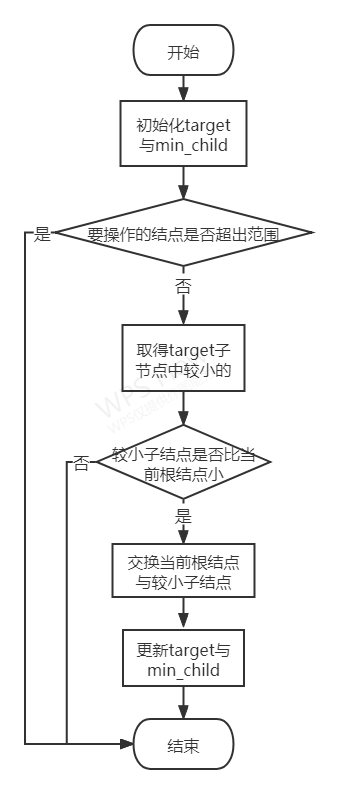
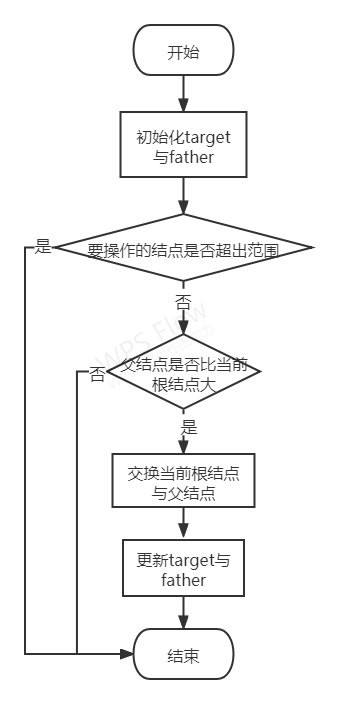
LevelTree函数



题8-2

TreeFindNode函数

1. （选做题）哈夫曼树构造

题8-3

upAdjust函数

题8-3

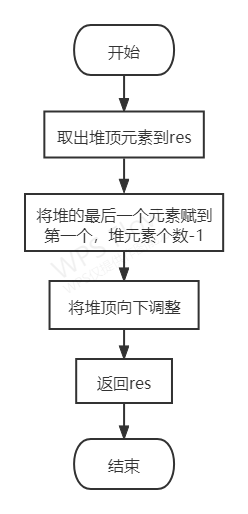
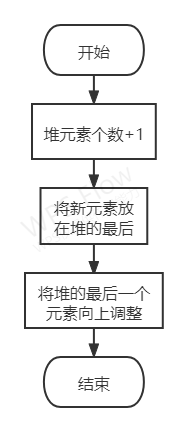
downAdjust函数

题8-3

insertElem函数

题8-3

deleteTop函数

### 五、程序代码

1. （必做题）链二叉树的基本实现



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define Status int

#define TElemType char

#define OK 1

#define ERROR 0

#define MAX\_NODE 100

typedef struct BiTNode {

TElemType data;

struct BiTNode \*lchild, \*rchild;

} BiTNode, \*BiTree;

Status constructByPreAndIn(BiTree, const TElemType \*, int, const TElemType \*, int, int);

void preTraverse(BiTNode root);

void inTraverse(BiTNode root);

void postTraverse(BiTNode root);

int countLeaf(BiTNode root);

int countDepth(BiTree root);

int main() {

TElemType pre[MAX\_NODE];

TElemType in[MAX\_NODE];

BiTNode root;

root.lchild = root.rchild = NULL;

memset(pre, '\0', sizeof(pre));

memset(in, '\0', sizeof(in));

printf("输入前序序列：\n");

gets(pre);

printf("输入中序序列：\n");

gets(in);

if (strlen(pre) != strlen(in)) {

printf("错误！两个序列的结点个数不同");

return 0;

}

if (constructByPreAndIn(&root, pre, 0, in, 0, (int) strlen(pre)-1) == ERROR) {

printf("错误！这两个序列无法构造出二叉树");

return 0;

}

printf("先序遍历：");

preTraverse(root);

printf("\n");

printf("中序遍历：");

inTraverse(root);

printf("\n");

printf("后序遍历：");

postTraverse(root);

printf("\n");

printf("叶子数目为：%d\n", countLeaf(root));

printf("深度为：%d\n", countDepth(&root));

system("pause");

return 0;

}

//中序区间为[in\_start, in\_end]

Status constructByPreAndIn(BiTree root, const TElemType \*preOrder, int cur, const TElemType \*inOrder, int in\_start,

int in\_end) {

//当前前序序列的开始第一个即为当前根结点元素

root->data = preOrder[cur];

BiTNode \*tmp;

int i;

//在中序序列中找当前根结点

for (i = in\_start; i <= in\_end; i++) {

if (inOrder[i] == preOrder[cur])

break;

}

//找不到结点，说明序列有错误

if (i == in\_end+1)

return ERROR;

//构建左子树（前提是左子树有结点）

if (i - in\_start > 0) {

tmp = (BiTNode \*) malloc(sizeof(BiTNode));

tmp->lchild = tmp->rchild = NULL;

root->lchild = tmp;

if (constructByPreAndIn(root->lchild, preOrder, cur + 1, inOrder, in\_start, i - 1) == ERROR)

return ERROR;

}

//构建右子树（前提是右子树有结点）

if (in\_end - i > 0) {

tmp = (BiTNode \*) malloc(sizeof(BiTNode));

tmp->lchild = tmp->rchild = NULL;

root->rchild = tmp;

if (constructByPreAndIn(root->rchild, preOrder, cur + i - in\_start + 1, inOrder, i + 1, in\_end) == ERROR)

return ERROR;

}

return OK;

}

void preTraverse(BiTNode root) {

printf("%c ", root.data);

if (root.lchild != NULL)

preTraverse(\*root.lchild);

if (root.rchild != NULL)

preTraverse(\*root.rchild);

}

void inTraverse(BiTNode root) {

if (root.lchild != NULL)

inTraverse(\*root.lchild);

printf("%c ", root.data);

if (root.rchild != NULL)

inTraverse(\*root.rchild);

}

void postTraverse(BiTNode root) {

if (root.lchild != NULL)

postTraverse(\*root.lchild);

if (root.rchild != NULL)

postTraverse(\*root.rchild);

printf("%c ", root.data);

}

int countLeaf(BiTNode root) {

int res = 0;

if (root.lchild == NULL && root.rchild == NULL)

res++;

if (root.lchild != NULL)

res += countLeaf(\*root.lchild);

if (root.rchild != NULL)

res += countLeaf(\*root.rchild);

return res;

}

int countDepth(BiTree root) {

int leftDepth = 0, rightDepth = 0;

if (root == NULL)

return 0;

else {

leftDepth = countDepth(root->lchild);

rightDepth = countDepth(root->rchild);

return 1 + (leftDepth > rightDepth ? leftDepth : rightDepth);

}

}

1. （选做题）main函数的完善与补充



#include <stdio.h>

#include <stdlib.h>

#define Status int

#define OK 1

#define ERROR 0

//定义数据元素类型

typedef char DATA;

// 定义二叉树结点类型

typedef struct CBT{

DATA data;//结点数据

struct CBT \*left;//左子树指针

struct CBT \*right;//右子树指针

} CBTType;

typedef CBTType\* QElemType;

typedef struct QNode{

QElemType pt;

struct QNode \*next;

} QNode;

typedef struct LinkQueue {

QNode \*head, \*rear;

} \*queuePtr;

/\* 实现二叉树基本操作：

\* 1. 初始化二叉树、查找结点、添加结点

\* 2. 获取左子树、获取右子树、显示结点数据

\* 3. 计算二叉树深度、清空二叉树、判断空树

\* 4. 遍历二叉树（按层、先序、中序、后序）

\*/

/\* 初始化二叉树，即返回一个空结点 \*/

CBTType \*InitTree();

/\* 查找数据域为data的结点 \*/

CBTType \*TreeFindNode(CBTType \*treeNode, DATA data);

/\* 在二叉树中添加结点的操作 \*/

void AddTreeNode(CBTType \*treeNode);

Status InsertNode(CBTType \*root, DATA ch);

/\* 获取左子树 \*/

CBTType \*TreeLeftNode(CBTType \*treeNode);

/\* 获取右子树 \*/

CBTType \*TreeRightNode(CBTType \*treeNode);

/\* 输出p所指结点的数据域 \*/

void TreeNodeData(CBTType \*p);

/\* 计算二叉树的深度 \*/

int TreeDepth(CBTType \*root);

/\* 清空二叉树 \*/

void ClearTree(CBTType \*treeNode);

/\* 判断是否为空树 \*/

int TreeIsEmpty(CBTType \*treeNode);

/\* 队列基本操作 \*/

Status Enqueue(queuePtr, QElemType);

Status Dequeue(queuePtr, QElemType \*);

Status Emptyqueue(queuePtr q);

/\* 分别为先序、中序、后序、层序遍历 \*/

void DLRTree(CBTType \*treeNode, void(\*TreeNodeData)(CBTType \*p));

void LDRTree(CBTType \*treeNode, void(\*TreeNodeData)(CBTType \*p));

void LRDTree(CBTType \*treeNode, void(\*TreeNodeData)(CBTType \*p));

void LevelTree(CBTType \*treeNode, void(\*TreeNodeData)(CBTType \*p));

int main() {

CBTType \*root = NULL;//定义root为指向根结点的指针

CBTType \*tmp;

DATA ch;

char menuSel;

void (\*Visit)();//指向函数的指针

Visit = TreeNodeData;//指向具体函数

root = InitTree();//初始化二叉树

//添加结点

do {

printf("请选择菜单添加二叉树结点\n");

printf("0.退出\t");

printf("1.添加结点\n");

while ((menuSel = (char) getchar())=='\n' || menuSel == '\r');

switch (menuSel) {

case '1':

AddTreeNode(root);

break;

case '0':

break;

default:;

}

} while (menuSel != '0');

//遍历二叉树

do {

printf("请选择菜单遍历二叉树，输入0退出\n");

printf("1.DLR先序遍历\n");

printf("2.LDR中序遍历\n");

printf("3.LRD后序遍历\n");

printf("4.按层遍历\n");

while ((menuSel = (char) getchar())=='\n' || menuSel == '\r');

switch (menuSel) {

case '0':

break;

case '1':

printf("\n先序遍历二叉树结果为：");

DLRTree(root, Visit);

printf("\n");

break;

case '2':

printf("\n中序遍历二叉树结果为：");

LDRTree(root, Visit);

printf("\n");

break;

case '3':

printf("\n后序遍历二叉树结果为：");

LRDTree(root, Visit);

printf("\n");

break;

case '4':

printf("\n按层遍历二叉树结果为：");

LevelTree(root, Visit);

printf("\n");

break;

default:;

}

} while (menuSel != '0');

printf("请输入要查找的数据：\n");

while ((ch = (char) getchar())=='\n' || ch == '\r');

tmp = TreeFindNode(root, ch);

if (tmp != NULL)

printf("二叉树中存在'%c'", tmp->data);

else

printf("二叉树中不存在'%c'", ch);

printf("\n二叉树深度为：%d\n", TreeDepth(root));

ClearTree(root);

root = NULL;

system("pause");

return 0;

}

void TreeNodeData(CBTType \*p) {

printf("%c ", p->data);

}

CBTType \*InitTree() {

CBTType \*tmp = malloc(sizeof(CBTType));

tmp->data = '\0';

tmp->left = tmp->right = NULL;

return tmp;

}

void AddTreeNode(CBTType \*treeNode) {

DATA chr;

printf("插入结点，请输入要插入的结点值（单个字符）：\n");

while ((chr = (char) getchar()) == '\n' || chr == '\r');

//特殊处理第一次插入的结点（作为根结点的值）

if (TreeIsEmpty(treeNode))

treeNode->data = chr;

else

InsertNode(treeNode, chr);

}

Status InsertNode(CBTType \*treeNode, DATA ch) {

int leftDepth= TreeDepth(treeNode->left);

int rightDepth = TreeDepth(treeNode->right);

//用于指向新创建的空结点

CBTType \*newNode;

//左/右子树为空，直接插入

if (leftDepth == 0) {

newNode = (CBTType \*) malloc(sizeof(CBTType));

newNode->left = newNode->right = NULL;

newNode->data = ch;

treeNode->left = newNode;

} else if (rightDepth == 0) {

newNode = (CBTType \*) malloc(sizeof(CBTType));

newNode->left = newNode->right = NULL;

newNode->data = ch;

treeNode->right = newNode;

} else {

//左右子树都不为空，插在深度较小的子树上，使二叉树尽量平衡

if (leftDepth <= rightDepth) {

InsertNode(treeNode->left, ch);

} else {

InsertNode(treeNode->right, ch);

}

}

return OK;

}

Status Enqueue(queuePtr q, QElemType elem) {

QNode\* tmp = (QNode \*) malloc(sizeof(QNode));

if (tmp == NULL)

return ERROR;

tmp->pt = elem;

tmp->next = NULL;

q->rear->next = tmp;

q->rear = tmp;

return OK;

}

Status Dequeue(queuePtr q, QElemType \*out) {

if (Emptyqueue(q) == OK)

return ERROR;

QNode \*tmp=q->head->next;

\*out = q->head->next->pt;

q->head->next = tmp->next;

//由于队列是有头结点的，对出队后变为空队列的情况做特殊处理

if (q->head->next == NULL)

q->rear = q->head;

free(tmp);

return OK;

}

Status Emptyqueue(queuePtr q) {

if (q->head->next == NULL)

return OK;

return ERROR;

}

int TreeDepth(CBTType \*root) {

int leftDepth = 0, rightDepth = 0;

if (root == NULL)

return 0;

else {

leftDepth = TreeDepth(root->left);

rightDepth = TreeDepth(root->right);

return 1 + (leftDepth > rightDepth ? leftDepth : rightDepth);

}

}

Status TreeIsEmpty(CBTType \*treeNode) {

if (treeNode->data == '\0')

return OK;

else

return ERROR;

}

void DLRTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p)) {

//访问当前根结点

TreeNodeData(treeNode);

//左子树不为空，则遍历左子树

if (treeNode->left != NULL)

DLRTree(treeNode->left, TreeNodeData);

//右子树不为空，则遍历右子树

if (treeNode->right != NULL)

DLRTree(treeNode->right, TreeNodeData);

}

void LDRTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p)) {

//左子树不为空，则遍历左子树

if (treeNode->left != NULL)

LDRTree(treeNode->left, TreeNodeData);

//访问当前根结点

TreeNodeData(treeNode);

//右子树不为空，则遍历右子树

if (treeNode->right != NULL)

LDRTree(treeNode->right, TreeNodeData);

}

void LRDTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p)) {

//左子树不为空，则遍历左子树

if (treeNode->left != NULL)

LRDTree(treeNode->left, TreeNodeData);

//右子树不为空，则遍历右子树

if (treeNode->right != NULL)

LRDTree(treeNode->right, TreeNodeData);

//访问当前根结点

TreeNodeData(treeNode);

}

void LevelTree(CBTType \*treeNode, void TreeNodeData(CBTType \*p)) {

//初始化层序遍历要使用的队列

queuePtr queue = (queuePtr) malloc(sizeof(struct LinkQueue));

queue->rear = queue->head = (QNode \*) malloc(sizeof(QNode));

queue->head->pt = NULL, queue->head->next = NULL;

QElemType tmp;

Enqueue(queue, treeNode);

while (Emptyqueue(queue) != OK) {

//出队一个元素（指向二叉树中某个结点的指针）

Dequeue(queue, &tmp);

TreeNodeData(tmp);//访问当前结点

//左子树不为空，将左结点入列

if (tmp->left != NULL)

Enqueue(queue, tmp->left);

//右子树不为空，将右结点入列

if (tmp->right != NULL)

Enqueue(queue, tmp->right);

}

}

CBTType \*TreeFindNode(CBTType \*treeNode, DATA data) {

CBTType \*pointer=NULL;

//当前根结点数据域就是data，直接返回

if (treeNode->data == data)

return treeNode;

else {

//在左子树中找

if (treeNode->left != NULL) {

pointer = TreeFindNode(treeNode->left, data);

//在左子树中找到了，就直接返回

if (pointer != NULL)

return pointer;

}

//在右子树中找

if (treeNode->right != NULL) {

pointer = TreeFindNode(treeNode->right, data);

//在右子树中找到了，就直接返回

if (pointer != NULL)

return pointer;

}

}

//以上都没有返回，说明当前根结点、左子树、右子树都找不到

return NULL;

}

void delete(CBTType \*node) {

free(node);

}

void ClearTree(CBTType \*treeNode) {

LRDTree(treeNode, delete);

}

CBTType \*TreeLeftNode(CBTType \*treeNode) {

return treeNode->left;

}

CBTType \*TreeRightNode(CBTType \*treeNode) {

return treeNode->right;

}

1. （选做题）哈夫曼树构造



#include <stdio.h>

#include <stdlib.h>

#define TElemType int

#define MAX\_NODE 200

//二叉树结点

typedef struct BiTNode {

TElemType data;

struct BiTNode \*lchild, \*rchild;

} BiTNode;

/\* 堆的操作 \*/

/\* 堆的向下调整 \*/

void downAdjust(BiTNode\* [], int, int);

/\* 堆的向上调整 \*/

void upAdjust(BiTNode\* [], int, int);

/\* 弹出堆顶的最小结点（的指针） \*/

BiTNode\* deleteTop(int \*n, BiTNode\* heap[]);

/\* 指向结点的指针数组，创建树的结点并创建最小堆 \*/

void createHeap(int, BiTNode\* []);

/\* 在最小堆中插入一个新结点（的指针） \*/

void insertElem(BiTNode\* newNode, BiTNode\* heap[], int \*n);

/\* 交换辅助函数，用于在堆中交换两个树结点（的指针） \*/

void swap(BiTNode\* \*, BiTNode\* \*);

/\* 输出数据域的辅助函数 \*/

void displayData(BiTNode \*node);

/\* 分别为先序、中序遍历，以此唯一确定一棵二叉树 \*/

void DLRTree(BiTNode \*treeNode, void(\*TreeNodeData)(BiTNode \*p));

void LDRTree(BiTNode \*treeNode, void(\*TreeNodeData)(BiTNode \*p));

/\* 由于纯C中没有STL，手打小顶堆以实现优先队列来完成哈夫曼树构建 \*/

/\* 堆是完全二叉树，就使用顺序存储结构（从下标1开始）来存储堆 \*/

int main() {

int n = 0;

//使用指针作为最小堆中的元素，即使用指针数组来表示

BiTNode\* trees[MAX\_NODE+1];

BiTNode \*tmp, \*min1, \*min2;

printf("请输入数据（总结点）个数\n");

scanf("%d", &n);

printf("请输入%d个数据（空格或回车分隔）\n", n);

for (int i = 1; i <= n; i++) {

tmp = malloc(sizeof(BiTNode));

scanf("%d", &tmp->data);

tmp->lchild = tmp->rchild = NULL;

trees[i] = tmp;

}

createHeap(n, trees);

while (n > 1) {

min1 = deleteTop(&n, trees);

min2 = deleteTop(&n, trees);

tmp = (BiTNode \*) malloc(sizeof(BiTNode));

tmp->data = min1->data+min2->data;

tmp->lchild = min1;

tmp->rchild = min2;

insertElem(tmp, trees, &n);

}

printf("哈夫曼树构造完成！\n前序遍历为：\n");

DLRTree(trees[1], displayData);

printf("\n中序遍历为：\n");

LDRTree(trees[1], displayData);

printf("\n");

system("pause");

return 0;

}

void swap(BiTNode\* \*a, BiTNode\* \*b) {

BiTNode\* tmp = \*a;

\*a = \*b;

\*b = tmp;

}

/\* 在[start, end]之间向下调整，一般start为开始调整的结点 \*/

void downAdjust(BiTNode\* heap[], int start, int end) {

int target = start, min\_child = start \* 2;

while (min\_child <= end) {

//选出target左右结点中较小的

//首先要存在右结点，才能与左结点比较

if (min\_child + 1 <= end) {

//右结点更小

if (heap[min\_child]->data > heap[min\_child+1]->data)

min\_child = min\_child + 1;

}

//有子结点更小，交换

if (heap[min\_child]->data < heap[target]->data) {

swap(&heap[min\_child], &heap[target]);

//还有继续向下调整的可能

target = min\_child;

min\_child = target \* 2;

} else {

//已经没得调整了，说明heap[start]已经调整到合适位置

break;

}

}

}

/\* 在[start, end]之间向上调整，一般end为开始调整的结点 \*/

void upAdjust(BiTNode\* heap[], int start, int end) {

int target = end, father = end/2;

while (father >= start) {

//父结点比待调整结点大，需要交换

if (heap[father]->data > heap[target]->data) {

swap(&heap[father], &heap[target]);

//还有继续向上调整的可能

target = father;

father = target / 2;

} else {

//已经没得调整了，说明heap[end]已经调整到合适位置

break;

}

}

}

/\* 根据结点指针数组heap构建小顶堆 \*/

void createHeap(int n, BiTNode\* heap[]) {

//根据完全二叉树与堆的性质，从n/2倒着往前操作

for (int i = n / 2; i >= 1; i--) {

downAdjust(heap, i, n);

}

}

/\* 弹出堆顶（即指向最小的结点的指针） \*/

BiTNode\* deleteTop(int \*n, BiTNode\* heap[]) {

BiTNode\* res;

res = heap[1];

//堆中最后一个元素放到第一个

heap[1] = heap[\*n];

(\*n)--;

downAdjust(heap, 1, \*n);

return res;

}

/\* 在长度为n的结点指针数组heap中插入新结点的指针newNode \*/

void insertElem(BiTNode\* newNode, BiTNode\* heap[], int \*n) {

(\*n)++;

heap[\*n] = newNode;

upAdjust(heap, 1, \*n);

}

/\* 根据指向结点的指针输出其数据域 \*/

void displayData(BiTNode \*node) {

printf("%d ", node->data);

}

/\* 递归前序遍历 \*/

void DLRTree(BiTNode \*treeNode, void TreeNodeData(BiTNode \*)) {

//访问当前根结点

TreeNodeData(treeNode);

//左子树不为空，则遍历左子树

if (treeNode->lchild != NULL)

DLRTree(treeNode->lchild, TreeNodeData);

//右子树不为空，则遍历右子树

if (treeNode->rchild != NULL)

DLRTree(treeNode->rchild, TreeNodeData);

}

/\* 递归中序遍历 \*/

void LDRTree(BiTNode \*treeNode, void TreeNodeData(BiTNode \*)) {

//左子树不为空，则遍历左子树

if (treeNode->lchild != NULL)

LDRTree(treeNode->lchild, TreeNodeData);

//访问当前根结点

TreeNodeData(treeNode);

//右子树不为空，则遍历右子树

if (treeNode->rchild != NULL)

LDRTree(treeNode->rchild, TreeNodeData);

}

### 六、测试和结果

1. （必做题）链二叉树的基本实现

**Input:**

1234

12345

**Output:**

错误！两个序列的结点个数不同

**Input:**

1234

1245

**Output:**

错误！这两个序列无法构造出二叉树

**Input:**

1234

4321

**Output:**

错误！这两个序列无法构造出二叉树

**Input:**

ABDGECF

GDBEAFC

**Output:**

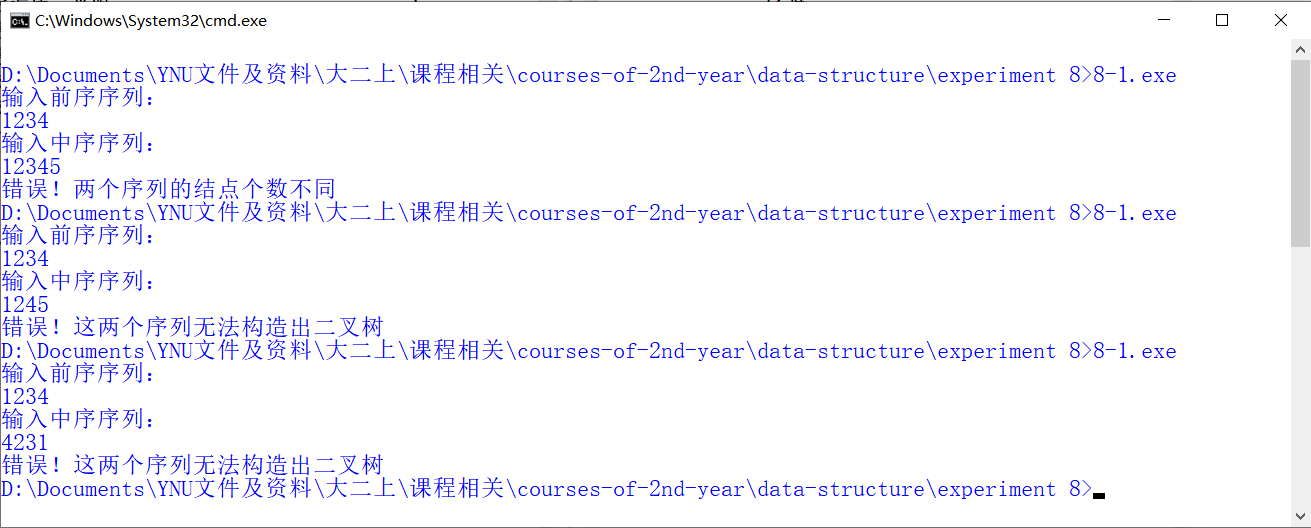
先序遍历：A B D G E C F

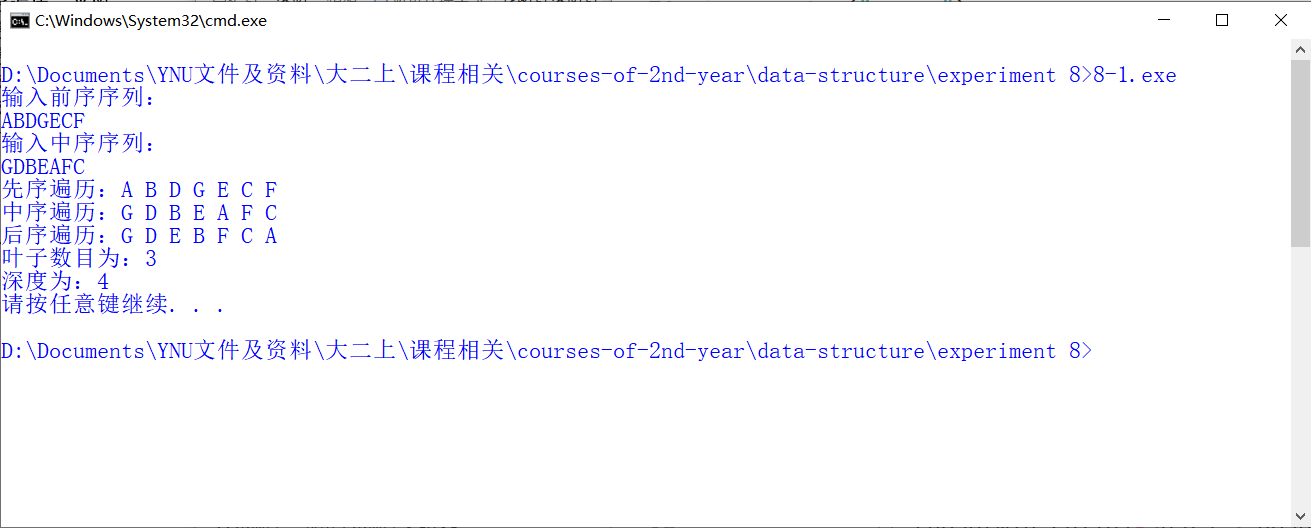
中序遍历：G D B E A F C

后序遍历：G D E B F C A

叶子数目为：3

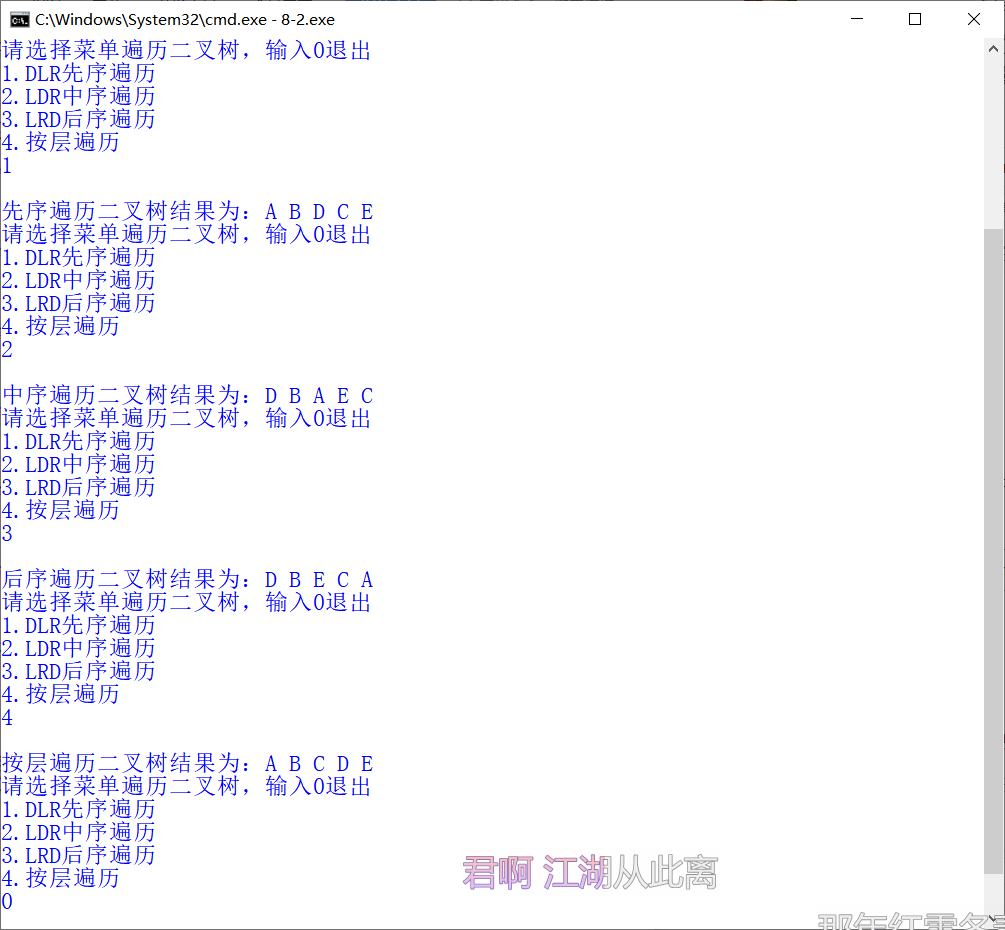
深度为：4

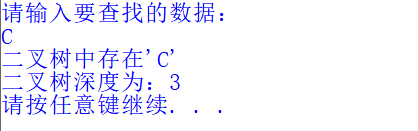




1. （选做题）main函数的完善与补充







1. （选做题）哈夫曼树构造

**Input:**

5

5 6 10 11 12

**Output:**

哈夫曼树构造完成！

前序遍历为：

44 21 10 11 5 6 23 11 12

中序遍历为：

10 21 5 11 6 44 11 23 12

**Input:**

3

1 5 3

**Output:**

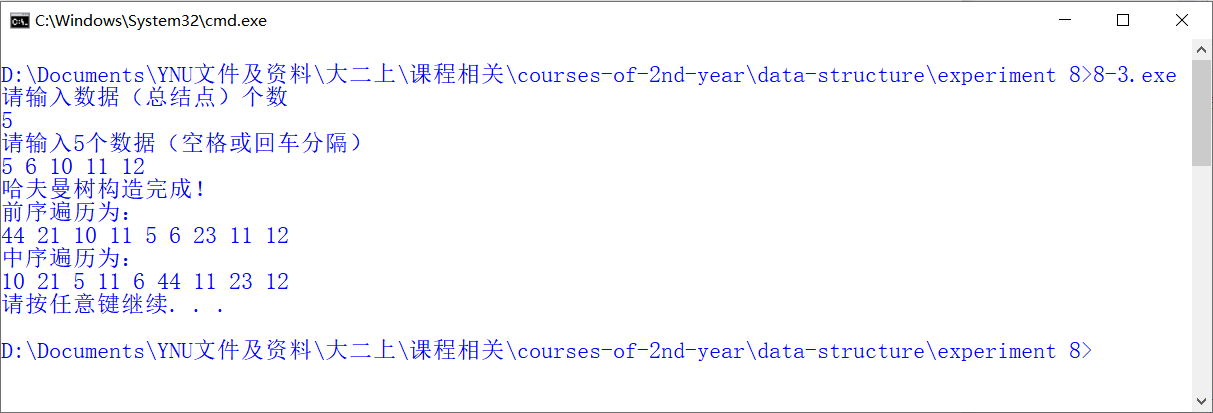
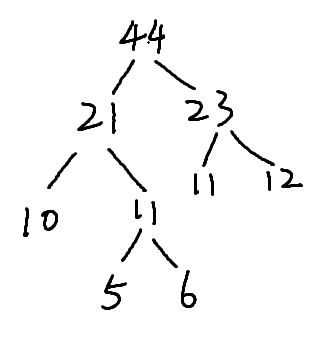
哈夫曼树构造完成！

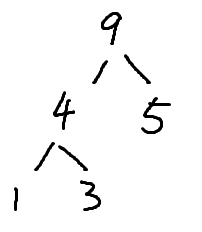
前序遍历为：

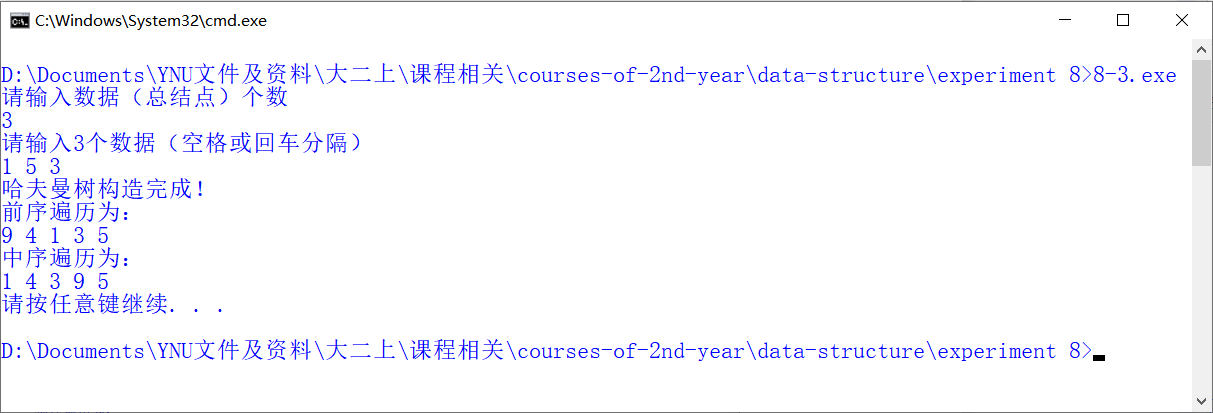
9 4 1 3 5

中序遍历为：

1 4 3 9 5







### 用户手册

1. （必做题）链二叉树的基本实现

各个结点中的数据域均为字符型，因此输入时每个字符都被当成一个结点的数据域。

1. （选做题）main函数的完善与补充

各个结点中的数据域均为字符型，因此输入时每个字符都被当成一个结点的数据域。由于TreeFindNode是根据数据域来判断结点是否相等的，输入的各个数据需要各不相同；若有相同的，输出的结点实际上是遍历中遇到第一个。

1. （选做题）哈夫曼树构造

各个数据必须在int范围内，并且各个数据的构成的二叉树的最小带权路径长度也不能超出int的范围。