**《数据结构》**

**实验报告**

**实验名称：树形数据结构及其应用**

|  |  |
| --- | --- |
| **学院** | **网络空间安全学院** |
| **班级** | **2020级软件工程专业软卓2班** |
| **学号** | **202041404130** |
| **姓名** | **张 瀚** |
| **指导教师** | **周 坤 晓** |
| **时间** | **2021/05/08** |

# 需求分析

通过实验达到：

⑴ 理解和掌握树及二叉树的基本概念；

⑵ 理解和掌握二叉树的顺序存储结构、链式存储结构；

⑶ 理解和掌握采用二叉链式存储结构下二叉树的各种遍历操作的思想及其应用；

⑷ 加深对堆栈、队列的概念及其典型操作思想的理解；

⑸ 掌握典型二叉树操作算法的算法分析。

# 实验题目

实验题目：二叉树的建立、遍历及其应用

设树结点的元素类型为ElemType(可以为char或int)，采用二叉链存储，实现以下二叉树的各种基本操作的程序：

① 编写一个创建二叉树的函数，通过文件读取方式，建立不少于10个结点的二叉树T；

② 用非递归方式先序遍历方式输出树T的结点；

③ 用非递归方式中序遍历方式输出树T的结点；

④ 用后序遍历方式输出树T的结点；

⑤ 用层次遍历方式输出树T的结点；

⑥ 输出树T的深度；输出树T的叶子结点和非叶子结点；

⑦ 主函数通过函数调用实现以上各项操作。

选做：

① 8-32判断该二叉树是否为完全二叉树；

② 8-34 根据顺序存储建立二叉链存储的二叉树（与第1个操作类似）；

③ 哈夫曼树编码问题的设计和实现（双亲孩子表示法+flag）。

## 数据结构设计

定义二叉树结点结构体如下：

typedef struct Node //结点

{

ElemType data; //数据域

struct Node \*leftChild; //左子树指针

struct Node \*rightChild; //右子树指针

}BiTreeNode;

## 主要操作算法设计与分析

**通过文件读取方式建立二叉树算法描述：**

void CreateBiTree(BiTreeNode \*\*root,FILE \*fp)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体指针的指针，一个FILE类型的指针

**步骤：**

A.从文件中读取字符ch；

B.判断字符是否为^，遇到^说明为当前结点为空结点，将空结点赋值为NULL，算法结束；

C.若当前结点不是空结点，将字符ch赋值给当前结点；

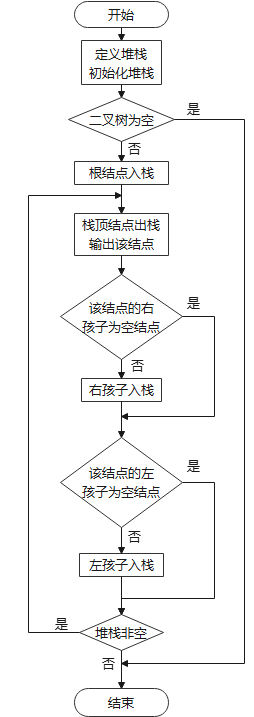
D.递归调用CreateBiTree函数，参数为当前结点的左节点指针，建立当前结点的左子树；

E.递归调用CreateBiTree函数，参数为当前结点的右节点指针，建立当前结点的右子树；

**算法时间复杂度分析：**

函数每次调用都会创建一个新结点（包括空结点），所以时间复杂度为O(n)。

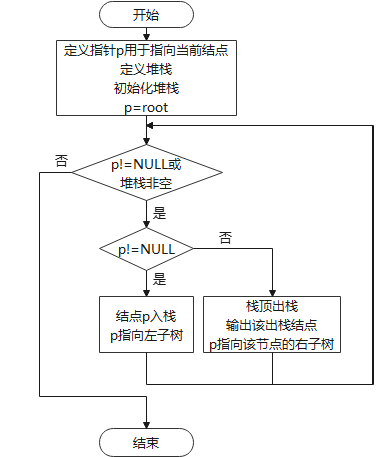
**非递归方式先序遍历输出算法流程图：**



**算法时间复杂度分析：**

算法开始时，堆栈为空。每次循环都会出栈一次，算法结束时，堆栈为空。二叉树中每个结点均入栈一次、出栈一次，因此算法时间复杂度为O(n)。

**非递归方式中序遍历输出算法流程图：**

****

**算法时间复杂度分析：**

算法开始时，堆栈为空。每次循环都会入栈或出栈一次，算法结束时，堆栈为空。二叉树中每个结点均入栈一次、出栈一次，因此算法时间复杂度为O(n)。

**后序遍历输出算法描述：**

void PostOrder(BiTreeNode \*root,void Visit(ElemType item))

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体类型的指针，一个函数虚参

**步骤：**

A.若二叉树为空，则算法结束；

B.若二叉树不为空，递归调用PostOrder函数，遍历左子树；

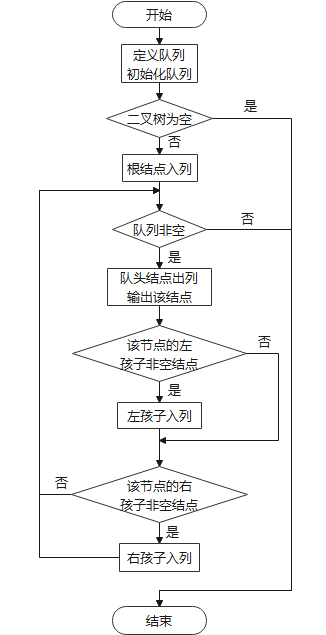
C.递归调用PostOrder函数，遍历右子树；

D.输出根结点；

**算法时间复杂度分析：**

函数每次调用都会访问一个新结点，二叉树每个结点均被访问一次，所以时间复杂度为O(n)。

**层序遍历输出算法流程图：**

****

**算法时间复杂度分析：**

算法开始时，队列为空。每次循环都会出队列一次，算法结束时，队列为空。二叉树中每个结点均入队列一次、出队列一次，因此算法时间复杂度为O(n)。

**计算树T的深度算法描述：**

int TreeDepth(BiTreeNode \*root)

**返回类型：**int型

**是否含参数：**是，含有一个结构体类型的指针

**步骤：**

A.定义左子树深度leftDepth和右子树深度rightDepth，并将其初始化为-1；

B.若二叉树为空，算法结束，返回-1；

C.若二叉树不为空，递归调用TreeDepth函数，参数为当前结点的左节点指针，计算左子树深度，并用leftDepth保存；

D.递归调用TreeDepth函数，参数为当前结点的右节点指针，计算右子树深度，并用rightDepth保存；

E.比较leftDepth和rightDepth，返回其较大值加1；

**算法时间复杂度分析：**

函数每次调用都会访问一个新结点，二叉树每个结点均被访问一次。所以时间复杂度为O(n)。

**输出树T的叶子结点和非叶子结点算法描述：**

void PrintLeaf(BiTreeNode \*root)

void PrintNotLeaf(BiTreeNode \*root)

**返回类型：**无返回值

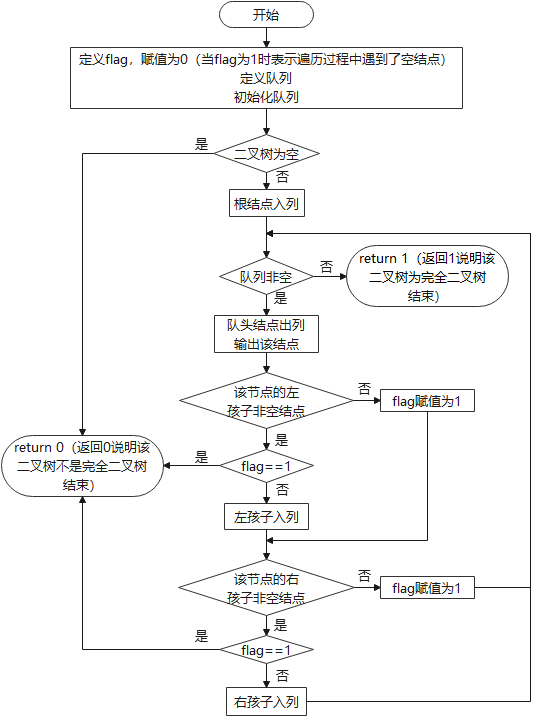
**是否含参数：**是，含有一个结构体类型的指针

**步骤：**

在前序遍历的基础之上，限制输出结点的条件。输出叶子节点的条件是当前结点无左孩子和右孩子；输出非叶子结点的条件是当前结点存在左孩子或右孩子；

**算法时间复杂度：**O(n)

**判断该二叉树是否为完全二叉树算法流程图：**

****

若flag的由0变为1，说明遍历过程中某个结点的一个孩子为空结点。若该二叉树为完全二叉树，队列中的剩下的结点均无左右孩子。

**算法时间复杂度分析：**

算法开始时，队列为空。每次循环都会出队列一次，二叉树中每个结点入队列至多一次、出队列至多一次，因此算法时间复杂度为O(n)。

**根据顺序存储建立二叉链存储的二叉树算法描述：**

void CreateBiTree2(BiTreeNode \*\*root,ElemType BT[],int n)

**返回类型：**无返回值

**是否含参数：**是，含有一个结构体指针的指针，一个数组，一个int型数据

**步骤：**

A.从BT[n]中读取数据，若n超过数组长度，或读取到的数据为^（说明该节点是空结点）时，将空结点赋值为NULL，算法结束；

B.否则，创建新节点，并将BT[n]的数据赋值给当前节点；

C.递归调用CreateBiTree2函数，访问BT[2n+1]，创建当前结点的左孩子。函数参数为(&(\*root)->leftChild,BT,2\*n+1)；

D.递归调用CreateBiTree2函数，访问BT[2n+2]，创建当前结点的右孩子。函数参数为(&(\*root)->rightChild,BT,2\*n+2)；

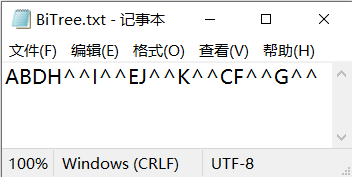
**算法时间复杂度：**O(n)

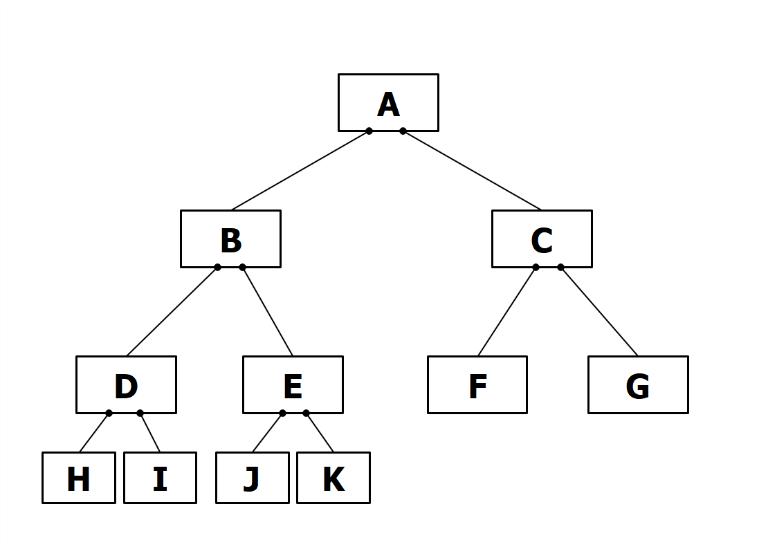
## 程序运行过程及结果（截屏粘贴）

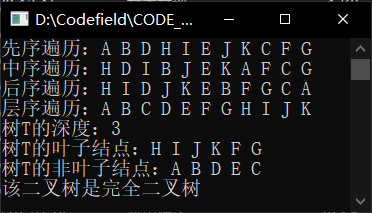
测试数据1：

BiTree.txt（按照先序遍历将二叉树结点写入文件，其中^表示空结点）:

ABDH^^I^^EJ^^K^^CF^^G^^

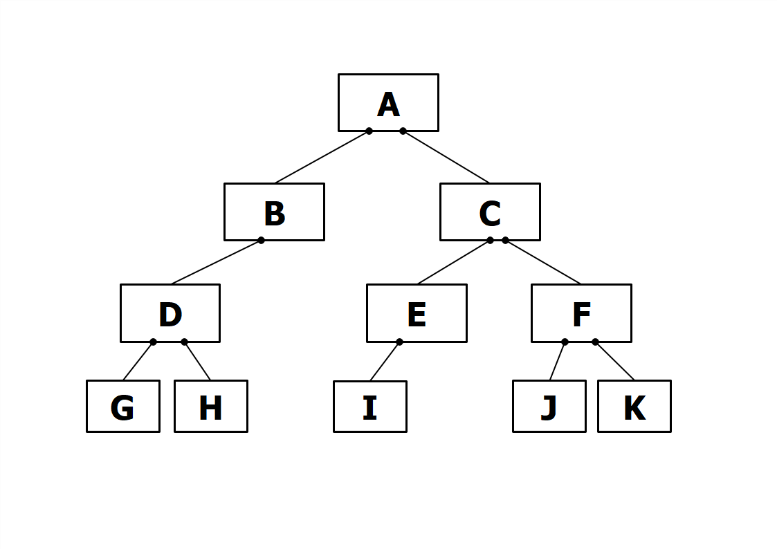


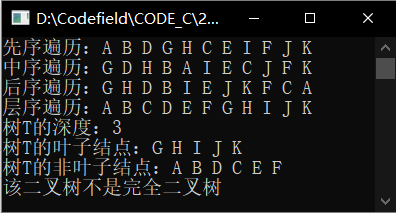




测试数据2：

顺序存储结构二叉树BT[n]：ABCD^EFGH^^I^JK（^表示空结点）





# 问题与总结

1.二叉树的递归定义决定了二叉树的许多操作算法中，涉及到递归算法。其中不少算法都是在前序遍历、中序遍历、后序遍历、层序遍历的基础上实现的；部分递归算法较难理解，通过画图可以方便、形象地地理解。

2.二叉树遍历的递归算法可以写成非递归算法，需要理解对堆栈的操作思想，借助堆栈可以方便实现非递归遍历算法。

3.二叉树的层序遍历算法借助队列可以方便的实现。

4.计算二叉树的深度，首先需要了解深度的定义，同时需要在前序遍历的基础上实现；

5.判断二叉树是否为完全二叉树，首先需要了解完全二叉树的结构特点，同时需要在层序遍历的基础的实现。

6.制作实验报告的过程中，学会制作流程图，可以方便、形象地看出算法的结构、运行步骤，比文字描述更容易理解。

# 附录：源代码

/\* 二叉树头文件 \*/

typedef struct Node //结点

{

ElemType data; //数据域

struct Node \*leftChild; //左子树指针

struct Node \*rightChild; //右子树指针

}BiTreeNode;

typedef BiTreeNode \*DataType; //堆栈、队列的元素为结点

#include "SeqCQueue.h" //队列头文件

#include "SeqStack.h" //堆栈头文件

//1.通过文件读取方式，建立二叉树

void CreateBiTree(BiTreeNode \*\*root,FILE \*fp)

{

char ch;

fscanf(fp,"%c",&ch);

if(ch=='^') //遇到^说明为空结点

{

\*root=NULL;

return;

}

\*root=(BiTreeNode\*)malloc(sizeof(BiTreeNode)); //创建新结点

(\*root)->data=ch;

CreateBiTree(&(\*root)->leftChild,fp); //创建当前结点的左子树

CreateBiTree(&(\*root)->rightChild,fp); //创建当前结点的右子树

}

//输出结点

void Visit(ElemType item)

{

printf("%c ", item);

}

//2.非递归方式先序遍历

void PreOrder(BiTreeNode \*root,void Visit(ElemType item))

{

BiTreeNode \*p;

SeqStack stack;

StackInitiate(&stack); //堆栈初始化

p=root;

if(p==NULL) printf("二叉树为空！\n");

else

{

StackPush(&stack,p); //根结点入栈

do

{

StackPop(&stack,&p); //结点出栈

Visit(p->data); //输出结点

if(p->rightChild!=NULL)

//当前结点的右孩子入栈

StackPush(&stack,p->rightChild);

if(p->leftChild!=NULL)

//当前结点的左孩子入栈

StackPush(&stack,p->leftChild);

}while(StackNotEmpty(stack)); //当堆栈为空时结束循环

}

}

//3.非递归方式中序遍历

void InOrder(BiTreeNode \*root,void Vist(ElemType item))

{

BiTreeNode \*p;

SeqStack stack;

StackInitiate(&stack); //堆栈初始化

p=root;

if(p==NULL) printf("二叉树为空！\n");

while(p || StackNotEmpty(stack)) //堆栈为空时结束循环

{

if(p)

{

StackPush(&stack,p); //结点入栈

p=p->leftChild; //指向左子树

}

else //左子树为空后

{

StackPop(&stack,&p); //退栈

Visit(p->data); //访问结点。

p=p->rightChild; //指向右子树

}

}

}

//4.后序遍历

void PostOrder(BiTreeNode \*root,void Visit(ElemType item))

{

if(root!=NULL)

{

PostOrder(root->leftChild, Visit);

PostOrder(root->rightChild, Visit);

Visit(root->data);

}

}

//5.层序遍历

void LevelOrder(BiTreeNode \*root,void Visit(ElemType item))

{

BiTreeNode \*p;

SeqCQueue queue;

QueueInitiate(&queue); //队列初始化

p=root;

if(p==NULL) printf("二叉树为空！\n");

else

{

QueueAppend(&queue,p); //根结点入列

while (QueueNotEmpty(queue)) //当队列为空时循环结束

{

QueueDelete(&queue,&p); //结点出列

Visit(p->data); //输出结点

if (p->leftChild!=NULL)

//当前结点的左孩子入列

QueueAppend(&queue,p->leftChild);

if (p->rightChild!=NULL)

//当前结点的右孩子入列

QueueAppend(&queue,p->rightChild);

}

}

}

//6.输出树T的深度

int TreeDepth(BiTreeNode \*root)

{

int leftDepth=-1; //深度初始化为-1

int rightDepth=-1;

if(root==NULL) return -1; //若为空结点，则深度为-1

else

{

leftDepth=TreeDepth(root->leftChild); //计算左子树深度

rightDepth=TreeDepth(root->rightChild); //计算右子树深度

//二叉树的深度为其左右子树的深度较大值加1

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

}

}

//6.输出树T的叶子结点

void PrintLeaf(BiTreeNode \*root)

{

if(root!=NULL)

{

//采用前序遍历

//无左孩子和右孩子

if(root->leftChild==NULL && root->rightChild==NULL)

Visit(root->data);

PrintLeaf(root->leftChild);

PrintLeaf(root->rightChild);

}

}

//6.输出树T的非叶子结点

void PrintNotLeaf(BiTreeNode \*root)

{

//采用前序遍历

if(root!=NULL)

{

//存在左孩子或右孩子

if(root->leftChild!=NULL || root->rightChild!=NULL)

Visit(root->data);

PrintNotLeaf(root->leftChild);

PrintNotLeaf(root->rightChild);

}

}

//选做1.判断该二叉树是否为完全二叉树

int ComBiTree(BiTreeNode \*root) //函数返回1说明该二叉树为完全二叉树

{

int flag=0; //当flag为1时表示已遇到空结点

BiTreeNode \*p;

SeqCQueue queue;

QueueInitiate(&queue); //队列初始化

p=root;

if(p==NULL)

{

printf("二叉树为空！\n");

return 0;

}

else

{

QueueAppend(&queue,p); //根结点入列

while (QueueNotEmpty(queue)) //当队列为空时结束循环

{

QueueDelete(&queue,&p); //结点出列

if(p->leftChild!=NULL)

{

//遍历过程中若遇到空结点后仍有非空结点入列

//则说明该二叉树不是完全二叉树

if(flag==1) return 0;

//当前结点的左孩子入列

QueueAppend(&queue,p->leftChild);

}

else flag=1; //若遇到空结点，flag的值由0变为1

if(p->rightChild!=NULL)

{

if(flag==1) return 0;

//当前结点的右孩子入列

QueueAppend(&queue,p->rightChild);

}

else flag=1;

}

}

return 1; //若完成上述循环，说明该二叉树为完全二叉树

}

//选做2.根据顺序存储建立二叉链存储的二叉树

void CreateBiTree2(BiTreeNode \*\*root,ElemType BT[],int n)

{

//访问BT[n]

//当n超过数组长度或数组元素为^时，返回

if(n>strlen(BT)-1 || BT[n]=='^')

{

\*root=NULL;

return;

}

\*root=(BiTreeNode\*)malloc(sizeof(BiTreeNode)); //创建新节点

(\*root)->data=BT[n];

//访问BT[2n+1]，创建左孩子

CreateBiTree2(&(\*root)->leftChild,BT,2\*n+1);

//访问BT[2n+2]，创建右孩子

CreateBiTree2(&(\*root)->rightChild,BT,2\*n+2);

}

//撤销二叉树

void Destroy(BiTreeNode \*\*root)

{

if((\*root)!=NULL)

{

Destroy(&(\*root)->leftChild);

Destroy(&(\*root)->rightChild);

free(\*root);

}

}

/\* 源文件 \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MaxQueueSize 100 //队列最大元素个数

#define MaxStackSize 100 //堆栈最大元素个数

typedef char ElemType;

#include "BiTree.h"

int main() //主函数

{

BiTreeNode \*T;

FILE \*fp;

char BT[]="ABCD^EFGH^^I^JK";

int depth;

//1.通过文件读取创建

// fp=fopen("BiTree.txt","r");

// CreateBiTree(&T,fp);

//fclose(fp);

//选做2.通过BT[n]创建

CreateBiTree2(&T,BT,0);

//2.非递归方式先序遍历

printf("先序遍历：");

PreOrder(T,Visit);

printf("\n");

//3.非递归方式中序遍历

printf("中序遍历：");

InOrder(T,Visit);

printf("\n");

//4.后序遍历

printf("后序遍历：");

PostOrder(T,Visit);

printf("\n");

//5.层序遍历

printf("层序遍历：");

LevelOrder(T,Visit);

printf("\n");

//6.输出树T的深度

printf("树T的深度：%d\n",TreeDepth(T));

//6.输出树T的叶子结点

printf("树T的叶子结点：");

PrintLeaf(T);

printf("\n");

//6.输出树T的非叶子结点

printf("树T的非叶子结点：");

PrintNotLeaf(T);

printf("\n");

//选做1.判断该二叉树是否为完全二叉树

if(ComBiTree(T)) printf("该二叉树是完全二叉树\n");

else printf("该二叉树不是完全二叉树\n");

//撤销二叉树

Destroy(&T);

}

/\* 队列头文件SeqCQueue.h和堆栈头文件SeqStack.h略 \*/