数据结构实验报告四

学号：201930310012 姓名：张金宇

#include<string.h>

#include<ctype.h>

#include<malloc.h> /\* malloc()等 \*/

#include<limits.h> /\* INT\_MAX等 \*/

#include<stdio.h> /\* EOF(=^Z或F6),NULL \*/

#include<stdlib.h> /\* atoi() \*/

#include<io.h> /\* eof() \*/

#include<math.h> /\* floor(),ceil(),abs() \*/

#include<process.h> /\* exit() \*/

/\* 函数结果状态代码 \*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

/\* #define OVERFLOW -2 因为在math.h中已定义OVERFLOW的值为3,故去掉此行 \*/

typedef int Status; /\* Status是函数的类型,其值是函数结果状态代码，如OK等 \*/

typedef int Boolean; /\* Boolean是布尔类型,其值是TRUE或FALSE \*/

#define MAX\_TREE\_SIZE 100 /\* 二叉树的最大结点数 \*/

typedef int ElemType;

typedef ElemType SqBiTree[MAX\_TREE\_SIZE]; /\* 0号单元存储根结点 \*/

#define Nil 0

#define LIST\_INIT\_SIZE 100

#define LISTINCREMRNT 10

typedef int ElemType;

typedef struct

{

ElemType \*base; //储存空间基地址

ElemType \*top; // 记录当前链表长度

int stacksize; //链表规模

} stack;

typedef struct

{

int level, order; /\* 结点的层,本层序号(按满二叉树计算) \*/

}position;

Status InitBiTree(SqBiTree T)

{ /\* 构造空二叉树T。因为T是固定数组，不会改变，故不需要& \*/

int i;

for (i = 0; i<MAX\_TREE\_SIZE; i++)

T[i] = Nil; /\* 初值为空 \*/

return OK;

}

void DestroyBiTree()

{ /\* 由于SqBiTree是定长类型,无法销毁 \*/

}

Status CreateBiTree(SqBiTree T)

{ /\* 按层序次序输入二叉树中结点的值(字符型或整型), 构造顺序存储的二叉树T \*/

int i = 0;

printf("请按层序输入结点的值(整型)，0表示空结点，输999结束。结点数≤%d:\n", MAX\_TREE\_SIZE);

while (1)

{

scanf("%c", &T[i]);

if (T[i] == '\n')

break;

if (i != 0 && T[(i + 1) / 2 - 1] == Nil&&T[i] != Nil) /\* 此结点(不空)无双亲且不是根 \*/

{

printf("出现无双亲的非根结点%d\n", T[i]);

exit(ERROR);

}

i++;

}

while (i<MAX\_TREE\_SIZE)

{

T[i] = Nil; /\* 将空赋值给T的后面的结点 \*/

i++;

}

return OK;

}

#define ClearBiTree InitBiTree /\* 在顺序存储结构中，两函数完全一样 \*/

Status BiTreeEmpty(SqBiTree T)

{ /\* 初始条件: 二叉树T存在 \*/

/\* 操作结果: 若T为空二叉树,则返回TRUE,否则FALSE \*/

if (T[0] == Nil) /\* 根结点为空,则树空 \*/

return TRUE;

else

return FALSE;

}

int BiTreeDepth(SqBiTree T)

{ /\* 初始条件: 二叉树T存在。操作结果: 返回T的深度 \*/

int i, j = -1;

for (i = MAX\_TREE\_SIZE - 1; i >= 0; i--) /\* 找到最后一个结点 \*/

if (T[i] != Nil)

break;

i++; /\* 为了便于计算 \*/

do

j++;

while (i >= pow(2.0, j));

return j;

}

Status InitStack(stack \*S)

{

(\*S).base = (ElemType\*)malloc(LIST\_INIT\_SIZE\*sizeof(ElemType));

if (!(\*S).base)

{

exit(OVERFLOW);

}

(\*S).top = (\*S).base;

(\*S).stacksize = LIST\_INIT\_SIZE;

return OK;

}

Status StackEmpty(stack S)

{ /\* 若栈S为空栈，则返回TRUE，否则返回FALSE \*/

if (S.top == S.base)

return TRUE;

else

return FALSE;

}

Status Push(stack \*S, ElemType e)

{

if ((\*S).top - (\*S).base >= (\*S).stacksize)

{

(\*S).base = (ElemType\*)realloc((\*S).base, ((\*S).stacksize + LISTINCREMRNT) \* sizeof(ElemType));

if (!(\*S).base)

exit(OVERFLOW);

(\*S).top = (\*S).base + (\*S).stacksize;

(\*S).stacksize += LISTINCREMRNT;

}

\*(S->top) = e;

(S->top)++;

}

Status Pop(stack \*S, ElemType \*e)

{

if ((\*S).top == (\*S).base)

return ERROR;

(\*S).top--;

\*e = \*((\*S).top);

}

Status Root(SqBiTree T, ElemType \*e)

{ /\* 初始条件: 二叉树T存在 \*/

/\* 操作结果: 当T不空,用e返回T的根,返回OK;否则返回ERROR,e无定义 \*/

if (BiTreeEmpty(T)) /\* T空 \*/

return ERROR;

else

{

\*e = T[0];

return OK;

}

}

ElemType Value(SqBiTree T, position e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点(的位置) \*/

/\* 操作结果: 返回处于位置e(层,本层序号)的结点的值 \*/

return T[(int)pow(2.0, e.level - 1) + e.order - 2];

}

Status Assign(SqBiTree T, position e, ElemType value)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点(的位置) \*/

/\* 操作结果: 给处于位置e(层,本层序号)的结点赋新值value \*/

int i = (int)pow(2.0, e.level - 1) + e.order - 2; /\* 将层、本层序号转为矩阵的序号 \*/

if (value != Nil&&T[(i + 1) / 2 - 1] == Nil) /\* 给叶子赋非空值但双亲为空 \*/

return ERROR;

else if (value == Nil && (T[i \* 2 + 1] != Nil || T[i \* 2 + 2] != Nil)) /\* 给双亲赋空值但有叶子（不空） \*/

return ERROR;

T[i] = value;

return OK;

}

ElemType Parent(SqBiTree T, ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点 \*/

/\* 操作结果: 若e是T的非根结点,则返回它的双亲,否则返回＂空＂ \*/

int i;

if (T[0] == Nil) /\* 空树 \*/

return Nil;

for (i = 1; i <= MAX\_TREE\_SIZE - 1; i++)

if (T[i] == e) /\* 找到e \*/

return T[(i + 1) / 2 - 1];

return Nil; /\* 没找到e \*/

}

ElemType LeftChild(SqBiTree T, ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点 \*/

/\* 操作结果: 返回e的左孩子。若e无左孩子,则返回＂空＂ \*/

int i;

if (T[0] == Nil) /\* 空树 \*/

return Nil;

for (i = 0; i <= MAX\_TREE\_SIZE - 1; i++)

if (T[i] == e) /\* 找到e \*/

return T[i \* 2 + 1];

return Nil; /\* 没找到e \*/

}

ElemType RightChild(SqBiTree T, ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点 \*/

/\* 操作结果: 返回e的右孩子。若e无右孩子,则返回＂空＂ \*/

int i;

if (T[0] == Nil) /\* 空树 \*/

return Nil;

for (i = 0; i <= MAX\_TREE\_SIZE - 1; i++)

if (T[i] == e) /\* 找到e \*/

return T[i \* 2 + 2];

return Nil; /\* 没找到e \*/

}

ElemType LeftSibling(SqBiTree T, ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点 \*/

/\* 操作结果: 返回e的左兄弟。若e是T的左孩子或无左兄弟,则返回＂空＂ \*/

int i;

if (T[0] == Nil) /\* 空树 \*/

return Nil;

for (i = 1; i <= MAX\_TREE\_SIZE - 1; i++)

if (T[i] == e&&i % 2 == 0) /\* 找到e且其序号为偶数(是右孩子) \*/

return T[i - 1];

return Nil; /\* 没找到e \*/

}

ElemType RightSibling(SqBiTree T, ElemType e)

{ /\* 初始条件: 二叉树T存在,e是T中某个结点 \*/

/\* 操作结果: 返回e的右兄弟。若e是T的右孩子或无右兄弟,则返回＂空＂ \*/

int i;

if (T[0] == Nil) /\* 空树 \*/

return Nil;

for (i = 1; i <= MAX\_TREE\_SIZE - 1; i++)

if (T[i] == e&&i % 2) /\* 找到e且其序号为奇数(是左孩子) \*/

return T[i + 1];

return Nil; /\* 没找到e \*/

}

void Move(SqBiTree q, int j, SqBiTree T, int i) /\* InsertChild()用到。加 \*/

{ /\* 把从q的j结点开始的子树移为从T的i结点开始的子树 \*/

if (q[2 \* j + 1] != Nil) /\* q的左子树不空 \*/

Move(q, (2 \* j + 1), T, (2 \* i + 1)); /\* 把q的j结点的左子树移为T的i结点的左子树 \*/

if (q[2 \* j + 2] != Nil) /\* q的右子树不空 \*/

Move(q, (2 \* j + 2), T, (2 \* i + 2)); /\* 把q的j结点的右子树移为T的i结点的右子树 \*/

T[i] = q[j]; /\* 把q的j结点移为T的i结点 \*/

q[j] = Nil; /\* 把q的j结点置空 \*/

}

Status InsertChild(SqBiTree T, ElemType p, Status LR, SqBiTree c)

{ /\* 初始条件: 二叉树T存在,p是T中某个结点的值,LR为0或1,非空二叉树c与T \*/

/\* 不相交且右子树为空 \*/

/\* 操作结果: 根据LR为0或1,插入c为T中p结点的左或右子树。p结点的原有左或 \*/

/\* 右子树则成为c的右子树 \*/

int j, k, i = 0;

for (j = 0; j<(int)pow(2.0, BiTreeDepth(T)) - 1; j++) /\* 查找p的序号 \*/

if (T[j] == p) /\* j为p的序号 \*/

break;

k = 2 \* j + 1 + LR; /\* k为p的左或右孩子的序号 \*/

if (T[k] != Nil) /\* p原来的左或右孩子不空 \*/

Move(T, k, T, 2 \* k + 2); /\* 把从T的k结点开始的子树移为从k结点的右子树开始的子树 \*/

Move(c, i, T, k); /\* 把从c的i结点开始的子树移为从T的k结点开始的子树 \*/

return OK;

}

Status(\*VisitFunc)(ElemType); /\* 函数变量 \*/

void PreTraverse(SqBiTree T, int e)

{ /\* PreOrderTraverse()调用 \*/

VisitFunc(T[e]);

if (T[2 \* e + 1] != Nil) /\* 左子树不空 \*/

PreTraverse(T, 2 \* e + 1);

if (T[2 \* e + 2] != Nil) /\* 右子树不空 \*/

PreTraverse(T, 2 \* e + 2);

}

Status PreOrderTraverse(SqBiTree T, Status(\*Visit)(ElemType))

{ /\* 初始条件: 二叉树存在,Visit是对结点操作的应用函数 \*/

/\* 操作结果: 先序遍历T,对每个结点调用函数Visit一次且仅一次。 \*/

/\* 一旦Visit()失败,则操作失败 \*/

VisitFunc = Visit;

if (!BiTreeEmpty(T)) /\* 树不空 \*/

PreTraverse(T, 0);

printf("\n");

return OK;

}

void InTraverse(SqBiTree T, int e)

{ /\* InOrderTraverse()调用 \*/

if (T[2 \* e + 1] != Nil) /\* 左子树不空 \*/

InTraverse(T, 2 \* e + 1);

VisitFunc(T[e]);

if (T[2 \* e + 2] != Nil) /\* 右子树不空 \*/

InTraverse(T, 2 \* e + 2);

}

Status InOrderTraverse(SqBiTree T, Status(\*Visit)(ElemType))

{ /\* 初始条件: 二叉树存在,Visit是对结点操作的应用函数 \*/

/\* 操作结果: 中序遍历T,对每个结点调用函数Visit一次且仅一次。 \*/

/\* 一旦Visit()失败,则操作失败 \*/

VisitFunc = Visit;

if (!BiTreeEmpty(T)) /\* 树不空 \*/

InTraverse(T, 0);

printf("\n");

return OK;

}

void PostTraverse(SqBiTree T, int e)

{ /\* PostOrderTraverse()调用 \*/

if (T[2 \* e + 1] != Nil) /\* 左子树不空 \*/

PostTraverse(T, 2 \* e + 1);

if (T[2 \* e + 2] != Nil) /\* 右子树不空 \*/

PostTraverse(T, 2 \* e + 2);

VisitFunc(T[e]);

}

Status PostOrderTraverse(SqBiTree T, Status(\*Visit)(ElemType))

{ /\* 初始条件: 二叉树T存在,Visit是对结点操作的应用函数 \*/

/\* 操作结果: 后序遍历T,对每个结点调用函数Visit一次且仅一次。 \*/

/\* 一旦Visit()失败,则操作失败 \*/

VisitFunc = Visit;

if (!BiTreeEmpty(T)) /\* 树不空 \*/

PostTraverse(T, 0);

printf("\n");

return OK;

}

void LevelOrderTraverse(SqBiTree T, Status(\*Visit)(ElemType))

{ /\* 层序遍历二叉树 \*/

int i = MAX\_TREE\_SIZE - 1, j;

while (T[i] == Nil)

i--; /\* 找到最后一个非空结点的序号 \*/

for (j = 0; j <= i; j++) /\* 从根结点起,按层序遍历二叉树 \*/

if (T[j] != Nil)

Visit(T[j]); /\* 只遍历非空的结点 \*/

printf("\n");

}

void Print(SqBiTree T)

{ /\* 逐层、按本层序号输出二叉树 \*/

int j, k;

position p;

ElemType e;

for (j = 1; j <= BiTreeDepth(T); j++)

{

printf("第%d层: ", j);

for (k = 1; k <= pow(2.0, j - 1); k++)

{

p.level = j;

p.order = k;

e = Value(T, p);

if (e != Nil)

printf("%d:%c ", k, e);

}

printf("\n");

}

}

Status Inorder (SqBiTree T,Status(\*Visit)(ElemType))//中序遍历非递归

{

stack S;

InitStack(&S);

while (T ||！ StackEmpty(S))

{

if (T)

{

Push(&S, T);

T = T->ichild;

}

else {

Pop(&5, &T);

if (!Visit(T->data))

return ERROR;

T = T->rchild;

}

}

return OK;

}

Status Visit(ElemType x)

{

printf("%c", x);

return 0;

}

void main()

{

SqBiTree T;

InitBiTree(T);

CreateBiTree(T);

PreOrderTraverse(T, Visit);

InOrderTraverse(T, Visit);

PostOrderTraverse(T, Visit);

//Print(T);

}

1. 一开始遍历根节点的所有左节点，并把他们压栈。
2. 弹栈一个节点并访问它。遍历这个节点的右孩子结点并压栈。
3. 依次扫描右孩子节点的所有左节点并压栈。
4. 我对树的掌握不够好，比如对节点查找的顺序。中序遍历是先左子树然后根节点然后右子树。