# 栈，堆和树的C语言实现

1. 栈

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

typedef int Elementtype; // 定义数据类型

// 定义节点

typedef struct Node {

Elementtype Element;

struct Node\* Next;

}NODE, \* PNODE;

// 定义栈结构体

typedef struct Stack {

PNODE PTOP; // 定义栈顶节点

PNODE PBOTTOM; // 定义栈底节点

}STACK, \* PSTACK;

// 函数声明

void InitStack(PSTACK Stack); // 初始化栈

void PushStack(PSTACK Stack, int val); // 入栈函数

void PopStack(PSTACK Stack, int\* val); // 出栈函数

void TraverseStack(PSTACK Stack); // 遍历栈函数

bool IsEmpty(PSTACK Stack); // 判断栈是否为空函数

void ClearStack(PSTACK Stack); // 清空栈函数void sum(PSTACK Stack) {

void sum(PSTACK Stack);

// 主函数

int j = 1;

int main() {

STACK Stack; // 创建一个栈变量

int val = 0; // 定义一个变量

InitStack(&Stack); // 调用初始化栈函数

int i = 10;

while (i != 0)

{

printf("1.入栈操作\n2.出栈操作\n3.遍历栈\n4.判断是否为空\n5.清空栈\n6.求和\n0.退出\n");

scanf\_s("%d", &i);

switch(i) {

case 1:

printf("输入要入栈的数字");

int k;

scanf\_s("%d", &k);

PushStack(&Stack, k);

break;

case 2:

PopStack(&Stack, &j);

if (j != 1) {

printf\_s("%d", j);

}

break;

case 3:

TraverseStack(&Stack);

break;

case 4:

IsEmpty(&Stack);

break;

case 5:

ClearStack(&Stack);

break;

case 6:

sum(&Stack);

break;

}

}

return 0;

}

// 定义初始化栈函数

void InitStack(PSTACK Stack) {

PNODE PNew = (PNODE)malloc(sizeof(NODE)); // 新建一个节点

if (PNew == NULL) { // 判断内存是否分配成功

printf("新节点空间分配失败！\n");

exit(-1);

}

Stack->PTOP = PNew; // 栈顶指针指向新节点

Stack->PBOTTOM = PNew; // 栈底指针指向新节点

PNew->Next = NULL; // 新节点指针指向为空

printf("栈创建成功！\n");

}

// 定义入栈函数

// 从栈顶插入数值val

void PushStack(PSTACK Stack, int val) {

PNODE P = (PNODE)malloc(sizeof(NODE)); // 创建新的节点用于存放变量

if (P == NULL) {

printf("分配空间内存失败！");

exit(-1);

}

P->Element = val; // 变量赋值给节点的数据域

P->Next = Stack->PTOP; // 让新建的节点指向上一个节点

Stack->PTOP = P; // 更新顶部节点，使其指向新建的节点

printf("%d 入栈成功！\n", val);

}

// 定义出栈函数

// 从栈顶弹出，并将地址赋值给变量val

void PopStack(PSTACK Stack, int\* val) {

if (Stack->PBOTTOM == Stack->PTOP) { // 判断栈是否为空

printf("出栈失败，栈为空！\n");

return;

}

PNODE P = Stack->PTOP; // 建立一个临时节点指向栈顶节点

\*val = P->Element; // 把栈顶节点元素弹出，地址放到变量val中

Stack->PTOP = P->Next; // 把栈顶指针指向上一个栈节点

free(P); // 释放节点内存，防止内存空间泄漏

P = NULL; // 防止产生野指针

printf("已将%d从栈弹出！\n", \*val);

}

// 定义判断栈是否为空函数

bool IsEmpty(PSTACK Stack) {

if (Stack->PBOTTOM == Stack->PTOP)

{

printf("栈为空！\n");

return true;

}

else

{

printf("栈不为空\n");

return false;

}

}

// 定义栈的遍历函数

void TraverseStack(PSTACK Stack) {

if (IsEmpty(Stack)) { // 判断栈是否为空

printf("遍历栈失败，栈为空！");

exit(-1);

}

PNODE P = Stack->PTOP; // 建立一个临时节点指针，初始化时指向栈顶

//使临时节点指针从栈顶开始遍历栈，直到栈底

printf("遍历栈的结果为： ");

while (P != Stack->PBOTTOM) {

printf("%d ", P->Element); // 打印出数据

P = P->Next;

}

printf("\n");

}

// 定义栈的清空函数

void ClearStack(PSTACK Stack) {

if (IsEmpty(Stack)) { // 判断栈是否为空

printf("栈已经为空，无需清空！\n");

return;

}

// 创建两个节点指针用来释放栈

PNODE P = Stack->PTOP;

PNODE Q = NULL;

// 清空栈

while (P != Stack->PBOTTOM) {

Q = P->Next;

free(P);

P = Q;

}

Stack->PTOP = Stack->PBOTTOM; // 栈顶指向栈底指向的初始空节点

printf("已清空栈！\n");

}

void sum(PSTACK Stack) {

int sum = 0;

if (IsEmpty(Stack)) { // 判断栈是否为空

printf("遍历栈失败，栈为空！");

return;

}

PNODE P = Stack->PTOP;

while (P != Stack->PBOTTOM) {

sum = sum + P->Element; // 打印出数据

P = P->Next;

}

printf("链表中的和为：%d", sum);

}

2堆

#include<stdio.h>

#include<stdlib.h>

#define N 10005

typedef int ElementType;

typedef struct HNode {/\* 堆的类型定义 \*/

ElementType\* Data; /\* 存储元素的数组 \*/

int Size; /\* 堆中当前元素个数 \*/

int Capacity; /\* 堆的最大容量 \*/

} HNode,\* Heap;

typedef Heap MaxHeap; /\* 最大堆 \*/

typedef Heap MinHeap; /\* 最小堆 \*/

#define MAXDATA 0/\* 该值应根据具体情况定义为大于堆中所有可能元素的值 \*/

#define MINDATA 0

void PercDown(MinHeap H, int p);

void BuildHeap(MinHeap H);

void heappaixu(MinHeap H);

bool Insert2(MinHeap H, ElementType X);

void PercDown2(MinHeap H, int p);

void BuildHeap2(MinHeap H);

void heappaixu2(MinHeap H);

MinHeap CreateMinHeap(int MaxSize)

{ /\* 创建容量为MaxSize的空的最小堆 \*/

MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));

H->Data = (ElementType\*)malloc((MaxSize + 1) \* sizeof(ElementType));

H->Size = 0;

H->Capacity = MaxSize;

H->Data[0] = MINDATA; /\* 定义"哨兵"为大于堆中所有可能元素的值\*/

return H;

}

MaxHeap CreateMaxHeap(int MaxSize)

{ /\* 创建容量为MaxSize的空的最小堆 \*/

MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));

H->Data = (ElementType\*)malloc((MaxSize + 1) \* sizeof(ElementType));

H->Size = 0;

H->Capacity = MaxSize;

H->Data[0] = MINDATA; /\* 定义"哨兵"为大于堆中所有可能元素的值\*/

return H;

}

bool IsFull(MinHeap H)

{

return (H->Size == H->Capacity);

}

bool Insert(MinHeap H, ElementType X)

{ /\* 将元素X插入最小堆H，其中H->Data[0]已经定义为哨兵 \*/

int i;

if (IsFull(H)) {

printf("最小堆已满");

return false;

}

i = ++H->Size; /\* i指向插入后堆中的最后一个元素的位置 \*/

for (; i!=1&&H->Data[i / 2] > X; i /= 2)

H->Data[i] = H->Data[i / 2]; /\* 上滤X \*/

H->Data[i] = X; /\* 将X插入 \*/

return true;

}

#define ERROR -1 /\* 错误标识应根据具体情况定义为堆中不可能出现的元素值 \*/

bool IsEmpty(MinHeap H)

{

return (H->Size == 0);

}

ElementType Deletemin(MinHeap H)

{

/\* 从最小堆H中取出键值为最小的元素，并删除一个结点 \*/

int Parent, Child;

ElementType MinItem, X;

if (IsEmpty(H)) {

printf("最小堆已为空");

return ERROR;

}

MinItem = H->Data[1]; /\* 取出根结点存放的最小值 \*/

//printf("%d\n", H->Size); /\* 用最小堆中最后一个元素从根结点开始向上过滤下层结点 \*/

X = H->Data[H->Size--];

H->Data[1] = X;

//printf("%d\n", X);/\* 注意当前堆的规模要减小 \*/

//printf("%d\n", H->Size);

BuildHeap(H);

return MinItem;

}

ElementType Deletemin2(MinHeap H)

{

/\* 从最小堆H中取出键值为最小的元素，并删除一个结点 \*/

int Parent, Child;

ElementType MinItem, X;

if (IsEmpty(H)) {

printf("最小堆已为空");

return ERROR;

}

MinItem = H->Data[1]; /\* 取出根结点存放的最小值 \*/

//printf("%d\n", H->Size); /\* 用最小堆中最后一个元素从根结点开始向上过滤下层结点 \*/

X = H->Data[H->Size--];

H->Data[1] = X;

//printf("%d\n", X);/\* 注意当前堆的规模要减小 \*/

//printf("%d\n", H->Size);

BuildHeap2(H);

return MinItem;

}

/\*----------- 建造最小堆 -----------\*/

void PercDown(MinHeap H, int p)

{ /\* 下滤：将H中以H->Data[p]为根的子堆调整为最小堆 \*/

int Parent, Child;

ElementType X;

X = H->Data[p]; /\* 取出根结点存放的值 \*/

for (Parent = p; Parent \* 2 <= H->Size; Parent = Child) {

Child = Parent \* 2;

if ((Child != H->Size) && (H->Data[Child] > H->Data[Child + 1]))

Child++; /\* Child指向左右子结点的较小者 \*/

if (X <= H->Data[Child]) break; /\* 找到了合适位置 \*/

else /\* 下滤X \*/

H->Data[Parent] = H->Data[Child];

}

H->Data[Parent] = X;

}

void BuildHeap(MinHeap H)

{ /\* 调整H->Data[]中的元素，使满足最小堆的有序性 \*/

/\* 这里假设所有H->Size个元素已经存在H->Data[]中 \*/

int i;

/\* 从最后一个结点的父节点开始，到根结点1 \*/

for (i = H->Size / 2; i > 0; i--)

PercDown(H, i);

}

int main()

{

MinHeap minheap;

int c, g;

int a, b, i;

printf("1.建一个最大堆\n2.建一个最小堆\n");

scanf\_s("%d", &c);

while (c!=1&&c!=2)

{

printf("请输入1或者2\n");

scanf\_s("%d", &c);

}

printf("堆的容量");

scanf\_s("%d", &a);

minheap = CreateMinHeap(a + 1);

switch (c) {

case 1:

printf("输入堆中的数字，中间用空格分开");

for (i = 1; i < a + 1; i++) {

scanf\_s("%d", &b);

Insert2(minheap, b);

}

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

break;

case 2:

printf("输入堆中的数字，中间用空格分开");

for (i = 1; i < a + 1; i++) {

scanf\_s("%d", &b);

Insert(minheap, b);

}

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

break;

}

printf("1.进行堆排序\n2.插入新的数字\n3删除堆中的元素\n");

int m;

scanf\_s("%d", &m);

int f;

switch (m)

{

case 1:

if (c == 1)

{

heappaixu2(minheap);

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

}

else {

heappaixu(minheap);

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

}

case 2:printf("输入插入元素的个数\n");

scanf\_s("%d",&f);

minheap->Data= (int\*)realloc(minheap->Data, (f+minheap->Capacity) \* sizeof(int));

minheap->Capacity += f;

printf("输入插入堆中的数字，中间用空格分开");

if (c == 1)

{

for (i = 1; i <=f; i++) {

scanf\_s("%d", &b);

Insert2(minheap, b);

}

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

break;

}

else {

for (i = 1; i <= f; i++) {

scanf\_s("%d", &b);

Insert(minheap, b);

}

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

break;

}

case 3:printf("输入删除元素个数：");

int n=0;

scanf\_s("%d",&n);

if (c == 1) {

for (; n> 0; n--)

Deletemin2(minheap);

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

}

else {

for (; n> 0; n--)

Deletemin(minheap);

for (int i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

}

}

}

/\*int a, b, i;

printf("最小堆的容量");

scanf\_s("%d", &a);

MinHeap minheap = CreateMinHeap(a + 1);

printf("输入堆中的数字，中间用空格分开");

for (i = 1; i < a + 1; i++) {

scanf\_s("%d", &b);

Insert(minheap, b);

}

for (i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

Deletemin2(minheap);

for (i = 1; i <= minheap->Size; i++)

printf("%d\t", minheap->Data[i]);

printf("\n");

heappaixu(minheap);\*/

void heappaixu(MinHeap H){

int i = H->Size, j, m = H->Size;

for (; m >0; m--) {

j = H->Data[H->Size];

H->Data[H->Size]=H->Data[1];

H->Data[1] = j;

H->Size--;

BuildHeap(H);

}H->Size = i;

}

bool Insert2(MinHeap H, ElementType X)

{ /\* 将元素X插入最小堆H，其中H->Data[0]已经定义为哨兵 \*/

int i;

if (IsFull(H)) {

printf("最大堆已满");

return false;

}

i = ++H->Size; /\* i指向插入后堆中的最后一个元素的位置 \*/

for (;(i!=1)&&( H->Data[i / 2])<X; i /= 2)

H->Data[i] = H->Data[i / 2]; /\* 上滤X \*/

H->Data[i] = X; /\* 将X插入 \*/

return true;

}

void heappaixu2(MinHeap H) {

int i = H->Size, j, m = H->Size;

for (; m > 0; m--) {

j = H->Data[H->Size];

H->Data[H->Size] = H->Data[1];

H->Data[1] = j;

H->Size--;

BuildHeap2(H);

}H->Size = i;

}

void PercDown2(MinHeap H, int p)

{ /\* 下滤：将H中以H->Data[p]为根的子堆调整为最大堆 \*/

int Parent, Child;

ElementType X;

X = H->Data[p]; /\* 取出根结点存放的值 \*/

for (Parent = p; Parent \* 2 <= H->Size; Parent = Child) {

Child = Parent \* 2;

if ((Child != H->Size) && (H->Data[Child] < H->Data[Child + 1]))

Child++; /\* Child指向左右子结点的较小者 \*/

if (X >= H->Data[Child]) break; /\* 找到了合适位置 \*/

else /\* 下滤X \*/

H->Data[Parent] = H->Data[Child];

}

H->Data[Parent] = X;

}

void BuildHeap2(MinHeap H)

{ /\* 调整H->Data[]中的元素，使满足最小堆的有序性 \*/

/\* 这里假设所有H->Size个元素已经存在H->Data[]中 \*/

int i;

/\* 从最后一个结点的父节点开始，到根结点1 \*/

for (i = H->Size / 2; i > 0; i--)

PercDown2(H, i);

}

3.树

#include<stdio.h>

#include<stdlib.h>

typedef struct BiTNode

{

char data;

struct BiTNode\* lchild, \* rchild;

}BiTNode, \* BiTree;

void PreOrderTraverse(BiTree T)//二叉树的先序遍历

{

if (T == NULL)

return;

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

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

void InOrderTraverse(BiTree T)//二叉树的中序遍历

{

if (T == NULL)

return;

InOrderTraverse(T->lchild);

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

InOrderTraverse(T->rchild);

}

void PostOrderTraverse(BiTree T)//后序遍历

{

if (T == NULL)

return;

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

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

}

void CreateBiTree(BiTree \* T)

{

char ch;

scanf\_s("%c", &ch);

if (ch == '#')

\* T = NULL;

else

{

\*T = (BiTree)malloc(sizeof(BiTNode));

if (!\*T)

exit(-1);

(\*T)->data = ch;

CreateBiTree(&(\*T)->lchild);

CreateBiTree(&(\*T)->rchild);

}

}

int main()

{

BiTree T;

CreateBiTree(&T);

PreOrderTraverse(T);

InOrderTraverse(T);

PostOrderTraverse(T);

return 0;

}