**辽 宁 大 学 信 息 学 院 实 验 报 告**

实验题目：**Linux操作系统** 专业、班级：计算机科学与技术 1 班

学 号： 161403108

姓 名： 刘长昊

20 19 年 06 月 03 日

**【实验目的】**

**1.** 利用C语言实现数据结构中相关内容。

**【实验代码】**

**1.堆栈**  
#include "stdio.h"

#include<stdlib.h>

#define maxsize 100

#define overflow 1

#define ok 1

#define error 1

#define SElemType int

#define Status int

typedef struct

{

SElemType \*base;

SElemType \*top;

int stacksize;

}SqStack;

Status InitStack(SqStack &S) //顺序栈的初始化

{

//构造一个空栈S

S.base=new SElemType[maxsize]; //为顺序栈动态分配一个最大容量为maxsize的数组空间

if(!S.base) exit(overflow); //存储分配失败

S.top=S.base; //top初始为base，空栈

S.stacksize=maxsize; //stacksize置为栈的最大容量maxsize

return ok;

}

Status Push(SqStack &S,SElemType e) //顺序栈的入栈

{

//插入元素e为新的栈顶元素

if(S.top-S.base==S.stacksize) return error;//栈满

\*S.top++=e; //元素e压入栈顶，栈顶指针加1

return ok;

}

Status Pop(SqStack &S,SElemType &e) //顺序栈的出栈

{

if(S.top==S.base) return error; //栈空

e=\*--S.top; //栈顶指针减1,将栈顶元素赋给e

return ok;

}

SElemType GetTop(SqStack S) //取栈顶元素

{

//返回S的栈顶元素，不修改栈顶指针

if(S.top!=S.base) //栈非空

return \*(S.top-1); //返回栈顶元素的值，栈顶指针不变

}

int main(int argc, char\* argv[])

{

return 0;

}

**2.查找树**

#include <stdio.h>

#include <stdlib.h>

#include <biTree.h>

#define N 10

void printNode(BiTreeNode \*node);

int main(int argc, char \*argv[]) {

BiTreeNode \*root;

int i;

root = NULL;

int data[N] = {10, 23, 11, 98, 111, 87, 34, 11, 33, 8};

for (i = 0; i < N; i++) {

insertNode(&root, data[i]);

}

printf("before delete:\n");

inorderTraversal(&root, printNode);

printf("\n");

deleteNode(&root, 11);

deleteNode(&root, 8);

printf("after delete:\n");

inorderTraversal(&root, printNode);

printf("\n");

exit(0);

}

void printNode(BiTreeNode \*node) {

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

}

**3.Queue**#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

struct Link

{

int data;

struct Link\* next;

};

struct Queue

{

struct Link\* front;

struct Link\* rear;

int size;

};

void QueueInit(struct Queue\* queue)

{

queue->front = NULL;

queue->rear = NULL;

queue->size = 0;

}

int QueueEmpty(struct Queue\* queue)

{

return (queue->size == 0);

}

void QueuePush(struct Queue\* queue, const int data)

{

struct Link\* node;

node = (struct Link\*)malloc(sizeof(struct Link));

assert(node != NULL);

node->data = data;

node->next = NULL;

if(QueueEmpty(queue))

{

queue->front = node;

queue->rear = node;

}

else

{

queue->rear->next = node;

queue->rear = node;

}

++queue->size;

}

int QueuePop(struct Queue\* queue, int\* data)

{

if (QueueEmpty(queue))

{

return 0;

}

struct Link\* tmp = queue->front;

\*data = queue->front->data;

queue->front = queue->front->next;

free(tmp);

--queue->size;

return 1;

}

void QueueDestroy(struct Queue\* queue)

{

struct Link\* tmp;

while(queue->front)

{

tmp = queue->front;

queue->front = queue->front->next;

free(tmp);

}

}

int main(void)

{

struct Queue queue;

int flag =1;

int choice;

int elem;

int i;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("| 1.创建队列 |\n");

printf("| 2.入队 |\n");

printf("| 3.出队 |\n");

printf("| 4.销毁队列 |\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

while(flag)

{

printf("请选择功能：\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

QueueInit(&queue);

break;

case 2:

printf("请输入插入的元素：");

scanf("%d",&elem);

QueuePush(&queue, elem);

break;

case 3:

if(!QueueEmpty(&queue))

{

QueuePop(&queue, &i);

printf("出队的元素为：%d\n", i);

}

break;

case 4:

QueueDestroy(&queue);

break;

default:

printf("程序运行结束，请按任意键退出！！！\n");

flag = 0;

}

}

return 0;

}

**4.Stack**#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define EmptyTOS (-1) //数组堆栈需要

#define MinStackSize (5) //数组堆栈需要

//堆栈函数的声明

struct Node;

typedef struct Node \*PtrToNode;

typedef PtrToNode Stack;

typedef int ElementType;

bool IsEmpty(Stack S); //检测堆栈是否为空

Stack CreateStack(); //创建堆栈

void DisposeStack(Stack); //毁灭堆栈

void Push(ElementType X, Stack S); //元素进栈

ElementType Pop(Stack S); //元素出栈

void MakeEmpty(Stack S); //清空堆栈中的元素

//ElementType Pop(Stack S); //元素出栈 (数组实现)

bool IsFull(Stack S); //检测堆栈是否已满 （数组实现需要）

void printStack(Stack S); //输出堆栈中的元素

struct Node { //链表结构

ElementType Element;

PtrToNode Next;

};

//struct Node { //数组结构

// int Capacity; //容量

// int TopOfStack: //栈顶

// ElementType \*Array;

//};

int main()

{

int number, flag;

ElementType X;

Stack S;

flag = 1;

printf("| 堆栈的基本使用 |\n");

printf("|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n");

printf("| 1. 创建堆栈 |\n");

printf("| 2. 销毁堆栈 |\n");

printf("| 3. 入栈 |\n");

printf("| 4. 出栈 |\n");

printf("| 5. 打印堆栈元素 |\n");

printf("|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|\n");

while (flag) {

printf("请选择功能 ： \n");

scanf("%d", &number);

switch(number) {

case 1 :

S = CreateStack();

break;

case 2 :

if (S) {

DisposeStack(S);

} else

printf("堆栈不存在！！！\n");

break;

case 3 :

if (S) {

printf("请输入需要插入的元素：");

scanf("%d", &X);

Push(X, S);

} else

printf("堆栈不存在！！！\n");

break;

case 4 :

if (S) {

Pop(S);

} else

printf("堆栈不存在！！！\n");

break;

case 5 :

if (S) {

printf("堆栈中元素如下：");

printStack(S);

} else

printf("堆栈不存在！！！\n");

break;

default :

printf("程序运行结束，请按任意键退出！！！\n");

flag = 0;

}

}

return 0;

}

//函数定义

bool IsEmpty(Stack S) //检测堆栈是否为空 （链表）

{

return S->Next == NULL;

}

//bool IsEmpty(Stack S) //检测堆栈是否为空 (数组)

//{

// return S->TopOfStack == EmptyTOS;

//}

Stack CreateStack() //创建堆栈 （链表）

{

Stack S;

if (!(S = malloc(sizeof(struct Node))))

exit(-1);

S->Next = NULL;

return S;

}

// Stack CreateStack(int MaxElements) //创建堆栈 (数组)

//{

// Stack S;

//

// if (MaxElements < MinStackSize) {

// printf("堆栈空间太小了！！");

// return -1;

// }

//

// if (!(S = malloc(sizeof(struct Node))))

// exit(-1);

// if (!(S->Array = malloc(sizeof((ElementType) \* MaxStackSize))))

// exit(-1);

// S->Capacity = MaxStackSize;

//

// return S;

// }

void DisposeStack(Stack S) //毁灭堆栈 （链表）

{

PtrToNode P, TmpCell;

P = S->Next;

S->Next = NULL;

while (P) {

TmpCell = P->Next;

free(P);

P = TmpCell;

}

free(S);

}

// void DisposeStack(Stack S) //毁灭堆栈 （数组）

// {

// if (S != NULL) {

// free(S->Array);

// free(S);

// }

// }

void Push(ElementType X, Stack S) //元素进栈 （链表）

{

PtrToNode TmpCell;

if (!(TmpCell = malloc(sizeof(struct Node))))

exit(-1);

else {

TmpCell->Element = X;

TmpCell->Next = NULL;

}

TmpCell->Next = S->Next;

S->Next = TmpCell;

}

//void Push(ElementType X, Stack S) //元素进栈 （数组）

//{

// if (!IsFull(S))

// S->Array[++TopOfStack] = X;

// else

// printf("堆栈已满！！\n");

//}

ElementType Pop(Stack S) //元素出栈 （链表）

{

Stack TmpCell;

ElementType X;

if (IsEmpty(S)) {

printf("堆栈为空！！！\n");

return -1;

}

else {

TmpCell = S->Next;

S->Next = TmpCell->Next;

X = TmpCell->Element;

free(TmpCell);

return X;

}

}

//ElementType Pop(Stack S) //元素出栈 （数组）

//{

// if (!IsEmpty(S))

// return S->Array[TopOfStack--];

// else

// printf("堆栈为空，出栈异常！！！\n");

//}

void MakeEmpty(Stack S) //清空堆栈中的元素 （链表）

{

PtrToNode P, TmpCell;

P = S->Next;

S->Next = NULL;

while (P) {

TmpCell = P->Next;

free(P);

P = TmpCell;

}

}

//void MakeEmpty(Stack S) //清空堆栈中的元素 （数组）

//{

// S->TopOfStack == EmptyTOS;

//}

//bool IsFull(Stack S) //检测堆栈是否已满 （数组实现需要）

//{

// return S->(++TopOfStack) == S->Capacity;

//}

//

void printStack(Stack S) //打印堆栈中元素

{

ElementType P;

while (!IsEmpty(S)) {

P = Pop(S);

printf("%d ", P);

}

printf("\n");

}

**5.Binarytree**#include<iostream>

#include<stdlib.h>

#include<malloc.h>

using namespace std;

typedef struct BinaryTreeNode{

char value;

struct BinaryTreeNode \*left;

struct BinaryTreeNode \*right;

};

void CreateBinaryTree(BinaryTreeNode \* &T)

{

char data;

cin>>data;

if(data=='#')

T=NULL;

else

{

T=new BinaryTreeNode;

T->value=data;

CreateBinaryTree(T->left);

CreateBinaryTree(T->right);

}

}

BinaryTreeNode \* CreateBinaryTree2()

{

BinaryTreeNode \*T;

char data;

cin>>data;

if(data=='#')

T=NULL;

else

{

T=new BinaryTreeNode;

T->value=data;

T->left=CreateBinaryTree2();

T->right=CreateBinaryTree2();

}

return T;

}

void PreorderTree(BinaryTreeNode \*T)

{

if(T==NULL)

return;

cout<<T->value<<" ";

PreorderTree(T->left);

PreorderTree(T->right);

}

void InorderTree(BinaryTreeNode \*T)

{

if(T==NULL)

return;

InorderTree(T->left);

cout<<T->value<<" ";

InorderTree(T->right);

}

void PostorderTree(BinaryTreeNode \*T)

{

if(T==NULL)

return;

PostorderTree(T->left);

PostorderTree(T->right);

cout<<T->value<<" ";

}

void main()

{

cout<<"---先序创建二叉树的第一种方法 输入序列为 A B C # # # D # #---------- "<<endl;

cout<<"请输入先序创建二叉树序列,以'#'号表示该节点为NULL，序列如上所示："<<endl;

BinaryTreeNode \*T;

CreateBinaryTree(T);

cout<<"方法1的先序序列为：";

PreorderTree(T);

cout<<endl;

cout<<"方法1的中序序列为：";

InorderTree(T);

cout<<endl;

cout<<"方法1的后序序列为：";

InorderTree(T);

cout<<endl;

cout<<"-----先序创建二叉树的第一种方法 输入序列为 A B C # # # D # #-------- "<<endl;

cout<<"请输入先序创建二叉树序列，以'#'号表示该节点为NULL，序列如上所示："<<endl;

BinaryTreeNode \*T1;

T1=CreateBinaryTree2();

cout<<"方法2的先序序列为：";

PreorderTree(T1);

cout<<endl;

cout<<"方法2的中序序列为：";

InorderTree(T1);

cout<<endl;

cout<<"方法2的后序序列为：";

InorderTree(T1);

cout<<endl;

}