**要求代码和实验报告规范，在算法思想中：对实验涉及的数据结构进行有效设计和分析；对算法进行分析并给出时间、空间复杂度的结论；清晰表达实验思路、出现的问题及解决方法。**

**一、调试成功程序及说明**

**1、**

**题目：**编程实现书P32 ADT Stack 基本操作9个，用顺序存储结构实现；

**算法思想：**利用顺序存储结构实现基本操作，结合书本，与之前顺序线性表基本操作相似，关键在于栈底和栈顶的处理，重视空间的申请，最后由switch菜单一一运行。

**运行结果：**运行出现包含9个基本操作的菜单，初始化和创建之后便可以实现各个命令操作的要求。

**结果分析：**正确

**附源程序。**

#include<iostream>

#include<stdlib.h>

#include<malloc.h>

using namespace std;

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define PARA\_ERROR 0

#define OVERFLOW -2

#define STACKINITSIZE 256

#define STACKINCREMENT 128

typedef int ElemType;

typedef int Status;

typedef struct SeqStack

{

ElemType \*pBase;

ElemType \*pTop;

int stacksize;

}SeqStack;

Status InitStack(SeqStack &S)

{

S.pBase = (ElemType\*)malloc(STACKINITSIZE\*sizeof(ElemType));

if (!S.pBase)

exit (OVERFLOW);

S.pTop = S.pBase;

S.stacksize = STACKINITSIZE;

cout<<"初始化成功！"<<endl;

return OK;

}

Status DestroyStack(SeqStack &S)

{

if(S.pBase !=NULL)

{

free(S.pBase );

S.pBase = NULL;

}

S.pTop = NULL;

S.stacksize = 0;

return OK;

}

Status ClearStack(SeqStack &S)

{

S.pTop = S.pBase ;

return OK;

}

Status StackEmpty(SeqStack S)

{

if (S.pTop == S.pBase )

{

cout<<"是空栈"<<endl;

return TRUE;

}

else

{

cout<<"不是空栈"<<endl;

return FALSE;

}

}

Status StackLength(SeqStack S)

{

int i = S.pTop - S.pBase ;

cout<<"此时栈的长度为"<<i<<endl;

}

Status GetTop(SeqStack &S ,ElemType &e)

{

if(S.pTop == S.pBase ) return ERROR;

e = \*(S.pTop -1);

return OK;

}

Status StackTraverse(SeqStack S)

{

if(S.pTop ==S.pBase )

cout<<"栈为空！"<<endl;

else

{

int i=0;

while(i<S.pTop -S.pBase )

{

cout<<\*(S.pBase +i)<<" ";

i++;

}

}

cout<<endl;

return OK;

}

Status Push(SeqStack &S , ElemType e)

{

if(S.pTop - S.pBase >=S.stacksize )

{

S.pBase = (ElemType\*)realloc(S.pBase ,(S.stacksize + STACKINCREMENT)\*sizeof(ElemType));

if(S.pBase ==NULL) exit(OVERFLOW);

S.pTop = S.pBase + S.stacksize ;

S.stacksize += STACKINCREMENT;

}

\*S.pTop = e;

S.pTop ++;

return OK;

}

Status Pop(SeqStack &S, ElemType &e)

{

if(S.pTop == S.pBase ) return ERROR;

e = \*(S.pTop -1);

S.pTop --;

return OK;

}

void menu()

{

cout<<"1.初始化空栈 2.添加栈顶元素"<<endl;

cout<<"3.销毁栈 4.清空栈"<<endl;

cout<<"5.判断空栈 6.元素个数"<<endl;

cout<<"7.栈顶元素值 8.删除栈顶元素"<<endl;

cout<<"9.遍历输出 "<<endl;

}

int main()

{

SeqStack S;

int choice,e;

while(1)

{

menu();

cout<<"请选择："<<endl;

cin>>choice;

switch(choice)

{

case(1):InitStack(S);break;

case(2):cout<<"请输入元素值:";

cin>>e;

Push(S, e);

cout<<"添加成功！"<<endl;

break;

case(3):DestroyStack(S);break;

case(4):ClearStack(S);break;

case(5):StackEmpty(S);break;

case(6):StackLength(S);break;

case(7):GetTop(S, e);

if(!GetTop(S, e))

cout<<"不存在栈顶元素！"<<endl;

else

cout<<"栈顶元素的值为"<<e<<endl;

break;

case(8):while(Pop(S, e))

{

cout<<"删除成功！被删除的元素值为"<<e<<endl;

break;

}

break;

case(9):StackTraverse(S);

break;

}

cout<<endl;

}

}

**2、**

**题目：**编程实现书P48 ADT Queue 基本操作9个，用链式存储结构实现；

**算法思想：**利用链式存储结构实现基本操作，结合书本，与之前链式线性表基本操作相似，关键在于队头的出和队尾的进，重视空间的申请，最后由switch菜单一一运行。

**运行结果：**运行出现包含9个基本操作的菜单，初始化和创建之后便可以实现各个命令操作的要求。

**结果分析：**正确

**附源程序。**

#include<iostream>

#include<stdlib.h>

#include<malloc.h>

using namespace std;

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define PARA\_ERROR 0

#define OVERFLOW -2

typedef int ElemType;

typedef int Status;

typedef struct LNode

{

ElemType data;

struct LNode \*next;

}LNode, \*QueuePtr;

typedef struct LinkQueue

{

QueuePtr front; //队头指针

QueuePtr rear; //队尾指针

}LinkQueue;

Status InitQueue(LinkQueue &Q)

{

Q.front =Q.rear = (LNode\*)malloc (sizeof(LNode));

if(Q.front ==NULL) exit(OVERFLOW);

Q.front->next = NULL;

cout<<"初始化成功！"<<endl;

return OK;

}

Status DestroyQueue(LinkQueue &Q)

{

if (Q.front ==Q.rear)

{

return ERROR;

}

LNode\* s = Q.front->next;

while (s)

{

Q.front->next = s->next;

if (Q.rear == s)

{

Q.rear = Q.front;

}

free(s);

s = Q.front->next;

}

free(Q.front );

Q.front =Q.rear = NULL;

return OK;

}

Status ClearQueue(LinkQueue &Q)

{

if (Q.front ==Q.rear)

{

return ERROR;

}

LNode\* s = Q.front->next;

while (s)

{

Q.front->next = s->next;

if (Q.rear == s)

{

Q.rear = Q.front;

}

free(s);

s = Q.front->next;

}

return OK;

}

Status QueueEmpty(LinkQueue Q)

{

if (Q.front == Q.rear )

{

cout<<"是空队列"<<endl;

return TRUE;

}

else

{

cout<<"不是空队列"<<endl;

return FALSE;

}

}

Status QueueLength(LinkQueue Q)

{

if (Q.front ==Q.rear)

{

return ERROR;

}

if (Q.front == Q.rear)

{

return 0;

}

LNode\* s = Q.front->next;

int num = 0;

while (s)

{

s = s->next;

num++;

}

cout<<"队列中元素个数为"<<num<<endl;

return OK;

}

Status GetHead(LinkQueue &Q ,ElemType &e)

{

if(Q.front == Q.rear ) return ERROR;

e = Q.front->next->data;

return OK;

}

Status QueueTraverse(LinkQueue Q)

{

if (Q.front ==Q.rear)

{

return ERROR;

}

LNode\* s = Q.front->next;

while (s)

{

cout<<s->data<<" ";

s = s->next;

}

cout<<endl;

return OK;

}

Status EnQueue(LinkQueue &Q , ElemType e)

{

LNode\* s = (LNode\*)malloc(sizeof(LNode));

if (!s)

{

return ERROR;

}

s->next = NULL;

s->data = e;

Q.rear->next = s;//将新结点添加到链表表中

Q.rear = s;//队尾指针指向新的队尾结点

}

Status DeQueue(LinkQueue &Q, ElemType &e)

{

LNode \*p;

if(Q.front ==Q.rear ) return ERROR;

p=Q.front->next ;

e=p->data ;

Q.front ->next = p->next ;

if(Q.rear ==p) Q.rear =Q.front ;

free(p);

return OK;

}

void menu()

{

cout<<"1.初始化队列 2.添加队尾元素"<<endl;

cout<<"3.销毁队列 4.清空队列"<<endl;

cout<<"5.判断空队列 6.元素个数"<<endl;

cout<<"7.队头元素值 8.删除队头元素"<<endl;

cout<<"9.遍历输出 "<<endl;

}

int main()

{

LinkQueue Q;

int choice,e;

while(1)

{

menu();

cout<<"请选择："<<endl;

cin>>choice;

switch(choice)

{

case(1):InitQueue(Q);break;

case(2):cout<<"请输入元素值:";

cin>>e;

EnQueue(Q, e);

cout<<"添加成功！"<<endl;

break;

case(3):DestroyQueue(Q);break;

case(4):ClearQueue(Q);break;

case(5):QueueEmpty(Q);break;

case(6):QueueLength(Q);break;

case(7):GetHead(Q, e);

if(!GetHead(Q, e))

cout<<"不存在队头元素！"<<endl;

else

cout<<"队头元素的值为"<<e<<endl;

break;

case(8):while(DeQueue(Q, e))

{

cout<<"删除成功！被删除的元素值为"<<e<<endl;

break;

}

break;

case(9):QueueTraverse(Q);

break;

}

cout<<endl;

}

}

**3、**

**题目：**利用栈操作实现八皇后问题求解。

**算法思想：**在栈的基本操作的基础上，结合书本，针对本题添加条件冲突算法，并且递归实现每层的结果。

**运行结果：**直接输出共92种八皇后问题解法。

**结果分析：**正确

**附源程序。**

#include<iostream>

#include<stdlib.h>

#include<malloc.h>

using namespace std;

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define OVERFLOW -2

#define RET\_OK 0

#define RET\_CONFLICT 1

#define STACKINITSIZE 256

#define STACKINCREMENT 128

int resultCount=0;

typedef int Status;

typedef struct

{

int col; //列坐标值

int row; //行坐标值

}Point;

typedef Point ElemType;

typedef struct SeqStack

{

ElemType \*pBase;

ElemType \*pTop;

int stacksize;

}SeqStack;

Status InitStack(SeqStack &S)

{

S.pBase = (ElemType\*)malloc(STACKINITSIZE\*sizeof(ElemType));

if (!S.pBase)

exit (OVERFLOW);

S.pTop = S.pBase;

S.stacksize = STACKINITSIZE;

cout<<"初始化成功！"<<endl;

return OK;

}

Status Push(SeqStack &S , ElemType e)

{

if(S.pTop - S.pBase >=S.stacksize )

{

S.pBase = (ElemType\*)realloc(S.pBase ,(S.stacksize + STACKINCREMENT)\*sizeof(ElemType));

if(S.pBase ==NULL) exit(OVERFLOW);

S.pTop = S.pBase + S.stacksize ;

S.stacksize += STACKINCREMENT;

}

\*S.pTop = e;

S.pTop ++;

return OK;

}

Status Pop(SeqStack &S, ElemType &e)

{

if(S.pTop == S.pBase ) return ERROR;

e = \*(S.pTop -1);

S.pTop --;

return OK;

}

Status JudgeQueenConfliction( Point newQueen, SeqStack StkQueen )

{

ElemType \*pCurQueen;

int x1, y1, x2, y2, tag;

x1 = newQueen.col;

y1 = newQueen.row;

pCurQueen = StkQueen.pBase ;

tag = RET\_OK;

while( pCurQueen < StkQueen.pTop )

{

x2 = pCurQueen->col;

y2 = pCurQueen->row;

if( x1 == x2 || abs(x1-x2) == abs(y1-y2) ) //在同一列或同一斜线

{

return RET\_CONFLICT;

}

pCurQueen ++;

}

return 0;

}

Status OutputResult( SeqStack &S )

{

Point \*p;

int i, j, tag;

p = S.pBase;

for( i = 0; i < 8; i ++ )

{

p = p + 1;

tag = p->col; //记录本行皇后的位置

for( j = 0; j < 8; j ++ )

{

if( tag == j ) //皇后在的位置

cout<<" x";

else

cout<<" 0";

if( j == 7 )

cout<<endl;

}

}

cout<<endl;

}

Status PlaceQueen( SeqStack &StkQueen, Point &curQueen, int row )

{ //八皇后问题递归实现

int col, ret;

for( col = 0; col < 8; col ++ )

{

curQueen.row = row;

curQueen.col = col;

ret = JudgeQueenConfliction( curQueen, StkQueen );

Push( StkQueen, curQueen );

if( ret == 0 )

{

if( row < 7 )

{

PlaceQueen( StkQueen, curQueen, row+1 ); //未到达最后一行，继续递归下一行

}

else if( row == 7 )

{

OutputResult(StkQueen);

resultCount ++;

}

}

Pop(StkQueen, curQueen);

}

return 0;

}

int main()

{

SeqStack StkQueen;

Point curQueen;

InitStack(StkQueen);

int row = 0, result = 0;

PlaceQueen( StkQueen, curQueen, row );

cout<<"共有"<<resultCount<<"种解法"<<endl;

}

**4、**

**题目：**CSP题目 最佳阈值

**算法思想：**依旧用的是遍历算法，数组记录与比较，算出所有的情况，按题目要求输出最佳阈值。

**运行结果：**输入安全指数和result后，输出最佳阈值。

**结果分析：**正确

**附源程序。**

#include <iostream>

using namespace std;

#define M 100000

int main()

{

int m,x;

int y[M],result[M];

int maxnumber=0,number=0,flag,max;

cin>>m;

for(int i=0;i<m;i++)

{

cin>>y[i];

cin>>result[i];

}

for(int i=0;i<m;i++)

{

x=y[i];

for(int j=0;j<m;j++)

{

if(y[j]<x)

{

flag=0;

}

else

{

flag=1;

}

if(flag==result[j])

number++;

}

if(number>=maxnumber)

{

maxnumber=number;

max=x;

}

number=0;

}

cout<<max;

}

**5、**

**题目：**CSP题目 窗口

**算法思想：**利用链式线性表来记录各个窗口及上下位置，记录后先逆置线性表，使得调整上下顺序时是从表头由上到下比较，并且调换顺序较为方便。（用数组数据域记录坐标时配合number会出错，未能找出原因。）

**运行结果：**输入坐标，会呈现最上方窗口。

**结果分析：**正确

**附源程序。**

#include<iostream>

#include<stdlib.h>

#include<malloc.h>

using namespace std;

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define PARA\_ERROR 0

#define OVERFLOW -2

typedef int ElemType;

typedef int Status;

typedef struct LNode

{

ElemType x1,x2,y1,y2; //数据域

ElemType number;

struct LNode \*next; //指针域

}LNode, \*LinkList;

Status InitList(LinkList &L)

{

L=(LNode\*) malloc (sizeof(LNode));

if(L==NULL) exit(OVERFLOW);

L->next=NULL;

return OK;

}

Status CreateList(LinkList &L,int n)

{

int x;

LNode \*p,\*q;

p=L;

for(int i=0;i<n;i++)

{

q=(LNode\*)malloc(sizeof(LNode));

if(q==NULL)

exit(OVERFLOW);

cin>>q->x1;

cin>>q->y1;

cin>>q->x2;

cin>>q->y2;

q->number = i+1;

p->next =q;

p=q;

p->next =NULL;

}

}

LinkList ReverseList(LinkList &L)

{

LNode \*p, \*q;

p=L->next;

L->next=NULL;

while(p)

{

q=p->next;

p->next=L->next;

L->next=p;

p=q;

}

return L;

}

Status TopList(LinkList &L,int m)

{

LNode \*p,\*q;

int flag=0;

p=L;

q=L->next ;

int M[1][m];

for(int i=0;i<m;i++)

{

cin>>M[0][i];

cin>>M[1][i];

}

for(int i=0;i<m;i++)

{

p=L;

q=L->next ;

while(q!=NULL)

{

flag=0;

if(M[0][i]>=q->x1&&M[0][i]<=q->x2&&M[1][i]>=q->y1&&M[1][i]<=q->y2 )

{

p->next =q->next ;

q->next =L->next ;

L->next =q;

cout<<q->number<<"\n";

flag=1;

break;

}

else

{

p=q;

q=q->next;

}

}

if(flag==0)

cout<<"IGNORED"<<"\n";

}

}

int main()

{

int m,n;

LinkList L;

InitList(L);

cin>>n;

cin>>m;

CreateList(L,n);

ReverseList(L);

TopList(L,m);

}

1. **代码行数及小结**

**行数：**合计700行左右

**小结：**本次是对栈和队列基本操作的巩固，与线性表有一些共通之处，且更多考察的是书本上的知识。这次的CSP题目做得时间相较前两次增长，主要在于第二道的链式线性表的应用，这也说明掌握书本知识的重要性。