**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。**

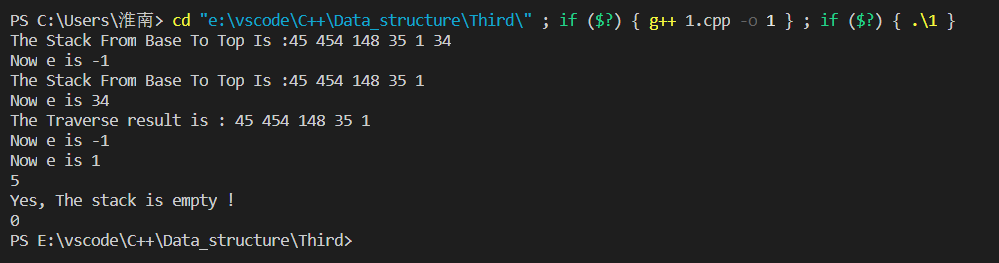
一、调试成功程序及说明

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：利用数组实现栈的基础操作

运行结果：



结果分析：所有操作结果均正确

附源程序。

/\*1．编程实现书P32 ADT Stack 基本操作9个，用顺序存储结构实现；\*/

#include <iostream>

#include <cstdlib>

using namespace std;

#ifndef ElemType

#define ElemType int

const int STACKINITSIZE = 256;

const int STACKINCREMENT = 128;

typedef struct SeqStack

{

ElemType \*pBase;

ElemType \*pTop;

int stacksize;

}SeqStack;

void print(ElemType e)

{

cout << e;

}

//打印当前元素

bool InitStack(SeqStack &S)

{

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

if (S.pBase == NULL)

{

exit(0);

}

S.pTop = S.pBase;

S.stacksize = STACKINITSIZE;

return true;

}

//栈的初始化

bool DestroyStack(SeqStack &S)

{

if (S.pBase != NULL)

{

free(S.pBase);

S.pBase = NULL;

}

S.pTop = NULL;

S.stacksize = 0;

return true;

}

//栈的销毁

bool ClearStack(SeqStack &S)

{

S.pTop = S.pBase; //直接让栈顶指向栈底部

return true;

}

//清空栈中元素

bool StackEmpty(const SeqStack &S)

{

if(S.pBase == S.pTop)

{

return true;

}

else

{

return false;

}

}

//判断栈是否为空栈

int StackLength(const SeqStack &S)

{

ElemType \*p = S.pBase;

int cnt=0;

while(p!=S.pTop)

{

p++;

cnt++;

}

return cnt;

}

//计算栈中元素的长度

bool StackTraverse(const SeqStack &S)

{

ElemType \*p = S.pBase;

cout << "The Traverse result is : ";

while(p != S.pTop)

{

print(\*p);

cout << " ";

p++;

}

cout << endl;

return true;

}

//从栈底向栈顶遍历所有元素

bool GetTop(SeqStack &S, ElemType &e)

{

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

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

return true;

}

//取出栈顶的元素

bool 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)

{

cout << "Memory is overflowed !"<<endl;

exit(0);

}

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

S.stacksize += STACKINCREMENT;

}

\*S.pTop = e;

S.pTop++;

return true;

}

//压栈操作

bool Pop(SeqStack &S, ElemType &e)

{

if (S.pTop <= S.pBase)

{

return false;

}

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

S.pTop--;

return true;

}

//弹栈操作

bool CreateStack(SeqStack &S, int n, ElemType \*Elem) //传入一个有n个数字的数组

{

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

{

Push(S,Elem[i]);

}

return true;

}

//初始化建栈

bool PrintStack(SeqStack S)

{

if (S.pBase == S.pTop)

{

cout << "This Satck is empty"<<endl;

return true;

}

cout << "The Stack From Base To Top Is :";

int \*p;

p = S.pBase;

while (p<S.pTop)

{

cout << \*p << " ";

p++;

}

cout << endl;

return true;

}

//打印一个栈中的元素

int main()

{

SeqStack S;

int e = -1;

int a[50]={45,454,148,35,1,34,899,1565,5448,65,-564,-1545121};

InitStack(S);

for (int i=0;i<=5;i++)

{

Push(S,a[i]);

}

PrintStack(S);

cout << "Now e is "<<e<<endl;

Pop(S,e);

PrintStack(S);

cout << "Now e is "<<e<<endl;

//ClearStack(S);

//PrintStack(S);

StackTraverse(S);

e = -1;

cout << "Now e is "<<e<<endl;

GetTop(S,e);

cout << "Now e is "<<e<<endl;

if(StackEmpty(S))

{

cout << "The stack is empty "<<endl;

}

cout << StackLength(S)<<endl;

ClearStack(S);

if(StackEmpty(S))

{

cout << "Yes, The stack is empty !"<<endl;

}

cout << StackLength(S)<<endl;

return 0;

}

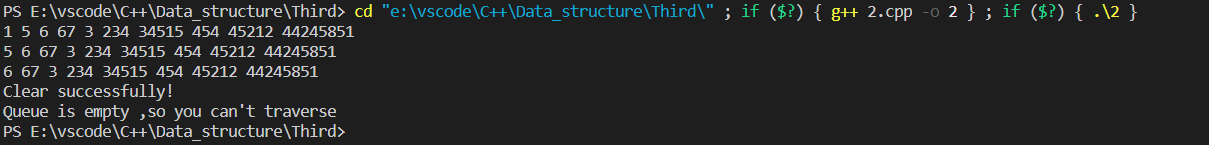
#endif

2、

题目：实现链式储存的队列的基本操作。

算法思想：按照队列的特点进行操作

运行结果：



结果分析：运行结果均正确，实现了队列的基本操作

附源程序。

#define ElemType int

#include <iostream>

#include <cstdlib>

using namespace std;

typedef struct LNode

{

ElemType data;

struct LNode \*next;

}LNode,\*QueuePtr;

typedef struct LinkQueue

{

QueuePtr front;

QueuePtr rear;

}LinkQueue;

void print(ElemType e)

{

cout << e;

}

//打印元素的函数

bool QueueEmpty(const LinkQueue &Q)

{

if(Q.front == Q.rear) //空队的标准是对头和队尾的指针相同

{

return true;

}

else

{

return false;

}

}

//判断队列是否为空队列，是的话返回true，否则返回false

bool InitQueue(LinkQueue &Q)

{

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

if( Q.front == NULL)

{

cout << "Memory is overflow "<<endl;

exit(0);

}

Q.front->next = NULL;

return true;

}

//队列的初始化

bool DestroyQueue(LinkQueue &Q)

{

if(Q.front == NULL && Q.rear == NULL)

{

return true;

}

QueuePtr p,q;

p = Q.front;

while(p != Q.rear)

{

q = p;

p = p->next;

free(q);

}

free(Q.rear);

Q.front = NULL;

Q.rear = NULL;

return true;

}

//销毁队列

bool ClearQueue(LinkQueue &Q)

{

if(QueueEmpty(Q))

{

cout << "This queue is empty ,no need to clear "<<endl;

return false;

}

QueuePtr p,q;

p = Q.front->next;

while(p != Q.rear)

{

q = p;

p = p->next;

free(q);

}

free(Q.rear);

Q.rear = Q.front;

return true;

}

//清空队列

int QueueLength(const LinkQueue &Q)

{

if(QueueEmpty(Q))

{

return 0;//空队列长度直接返回0

}

QueuePtr p;

p = Q.front->next;

int length = 0;

while(p->next != NULL)

{

length++;

p = p->next;

}

return length;

}

//计算队列的长度

bool GetHead(const LinkQueue &Q, ElemType &e)

{

if(Q.front != NULL)

{

return false;

}

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

return true;

}

//取出队头元素

bool QueueTraverse(const LinkQueue &Q)

{

if (QueueEmpty(Q))

{

cout << "Queue is empty ,so you can't traverse"<<endl;

return false;

}

QueuePtr p = Q.front->next;

while(p != NULL)

{

print(p->data);

cout << " ";

p = p->next;

}

cout << endl;

}

//遍历队列中的元素

bool EnQueue(LinkQueue &Q, ElemType e)

{

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

if(s == NULL)

{

cout << "Memory is overflow "<<endl;

exit (0);

}

s->data = e;

Q.rear->next = s;

Q.rear = s;

s->next = NULL;

}

//入队操作

bool DeQueue(LinkQueue &Q,ElemType &e)

{

if(QueueEmpty(Q))

{

cout << "This Queue is empty"<<endl;

return false;

}

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

e = p->data;

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

if(Q.rear == p)

{

Q.rear = Q.front;

}

free(p);

return true;

}

//出队操作

bool CreateQueue(LinkQueue &Q, int n, ElemType \*series) //n为个数,series 为一个数组

{

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

{

EnQueue(Q,series[i]);

}

}

//建立一个队列的函数

int main()

{

LinkQueue Q;

InitQueue(Q);

int a[100] = {1,5,6,67,3,234,34515,454,45212,44245851,5454};

int e = -1;

CreateQueue(Q,10,a);

QueueTraverse(Q);

for(int i=1;i<=2;i++)

{

DeQueue(Q,e);

QueueTraverse(Q);

}

// QueueTraverse(Q)

ClearQueue(Q);

if(QueueEmpty(Q))

{

cout <<"Clear successfully! "<<endl;

}

DestroyQueue(Q);

QueueTraverse(Q);

return 0;

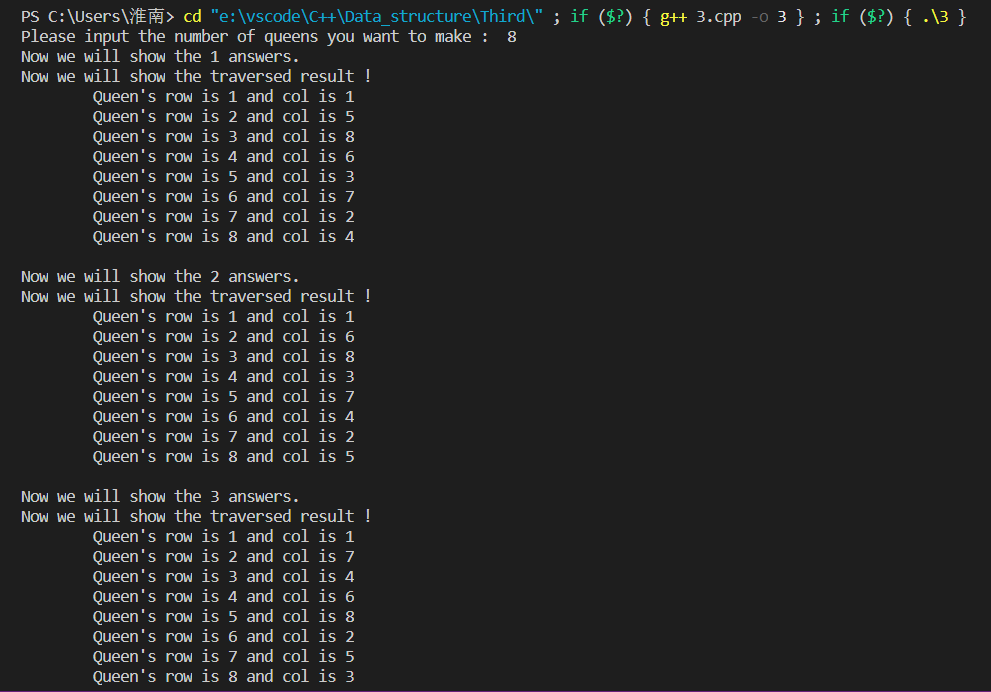
}

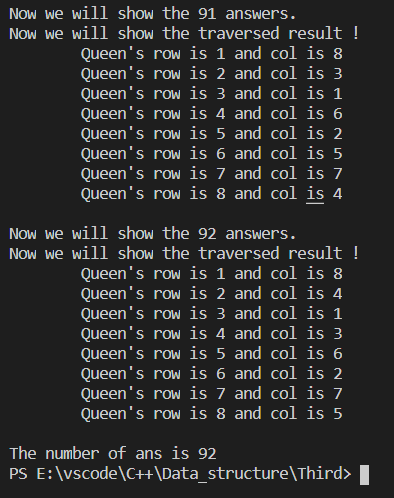
3、

题目：用一个栈解决八皇后问题。

算法思想：先设计数据结构，一个皇后涉及到两个属性，包括行标和列标，把皇后定义为一个包含这两个信息的结构体。利用栈来模拟皇后的放置的过程，因为栈有先进先出的特性，适合穷举搜索时候的回溯。然后设计一个预放置判断的函数，函数实现判断当前栈的状态，能否放置当前传入的皇后，是则返回true，否则返回false。接下来进入主函数，现将坐标为（1，1）的皇后压栈，总体的原则是按照行号每行放一个，对每行中的列进行挨个判断，如果某一行所有列都没有位置可以放，那么回溯继续搜索上一个列的放置方法。如果达到搜索终点，那么进行答案记录，并弹出当前的皇后，按照列号继续进行搜索。

运行结果：





结果分析:运行结果正确，能输出每个情况的状态。

附源程序：//用一个栈解决八皇后问题

typedef struct Queen //定义皇后这个结构体

{

int row; //皇后的行数

int col; //皇后的列数

}Queen;

#ifndef ElemType

#define ElemType Queen

#include "Stack.h"

#include <cmath>

using namespace std;

void print(ElemType e)

{

cout <<"\tQueen's row is "<<e.row<<" and col is "<<e.col<<endl;

}

//打印皇后状态的函数

void printQueen(Stack S,int ans)

{

cout << "Now we will show the "<<ans<< " answers."<<endl;

StackTraverse(S);

}

bool JudgeConfliction(Queen newQueen, Stack QueenStack)

{

ElemType \*pCurQueen;

int x1 = newQueen.col, y1 = newQueen.row;

pCurQueen = QueenStack.pBase;

bool tag = false;

while(pCurQueen<QueenStack.pTop)

{

register int x2 = pCurQueen->col, y2 = pCurQueen->row;

if(x1 == x2)

{

tag = true;

break;

//发生冲突,tag置为true,循环结束

}

if(y1 == y2)

{

tag = true;

break;

//发生冲突,tag置为true,循环结束

}

if(abs(x1-x2) == abs(y1-y2))

{

tag = true;

break;

}

pCurQueen ++;

}

return tag;

}

//判断皇后的冲突，如果冲突就返回true，否则返回false

int main()

{

int number;

cout << "Please input the number of queens you want to make : ";

cin >> number;

Stack QueenStack;

Queen curQueen;

InitStack(QueenStack);

int ans = 0;

curQueen.row = 1;

curQueen.col = 1;

Push(QueenStack,curQueen);

//把第一个皇后压栈

curQueen.row = 2;

while(curQueen.col <= number && curQueen.row <= number)

{

bool tag = true; //默认是冲突的状态

while(curQueen.col <= number)

{

tag = JudgeConfliction(curQueen,QueenStack);

if(!tag)

{

break; //在这一行找到了合适的列

}

curQueen.col += 1;

}

if(!tag)

{

Push(QueenStack,curQueen); //推入当前的皇后，放下一行的

curQueen.row += 1;

curQueen.col = 1;

if(curQueen.row > number) //达到搜索终点

{

ans++;

printQueen(QueenStack,ans);

Pop(QueenStack,curQueen);

curQueen.col+=1;

while(curQueen.col > number && !StackEmpty(QueenStack))

{

Pop(QueenStack,curQueen);

curQueen.col+=1;

}

}

}

else

{

Pop(QueenStack,curQueen);

curQueen.col+=1;

while(curQueen.col > number && !StackEmpty(QueenStack))

{

Pop(QueenStack,curQueen);

curQueen.col+=1;

}

}

}

cout << "The number of ans is " << ans << endl;

DestroyStack(QueenStack);

return 0;

}

#endif

/\*

这是栈的头文件，调用时注意 宏定义ElemType

1.InitStack

2.DestroyStack

3.ClearStack

4.GetTop

5.Push

6.Pop

7.CreateStack

\*/

#include <cstdlib>

#include <iostream>

using namespace std;

const int STACKINITSIZE = 256;

const int STACKINCREMENT = 128;

extern void print(ElemType e);

typedef struct Stack

{

ElemType \*pBase;

ElemType \*pTop;

int stacksize;

}Stack;

bool InitStack(Stack &S)

{

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

if (S.pBase == NULL)

{

exit(0);

}

S.pTop = S.pBase;

S.stacksize = STACKINITSIZE;

return true;

}

//栈的初始化

bool DestroyStack(Stack &S)

{

if (S.pBase != NULL)

{

free(S.pBase);

S.pBase = NULL;

}

S.pTop = NULL;

S.stacksize = 0;

return true;

}

//栈的销毁

bool ClearStack(Stack &S)

{

S.pTop = S.pBase; //直接让栈顶指向栈底部

return true;

}

//清空栈中元素

bool StackEmpty(const Stack &S)

{

if(S.pBase == S.pTop)

{

return true;

}

else

{

return false;

}

}

//判断栈是否为空栈

int StackLength(const Stack &S)

{

ElemType \*p = S.pBase;

int cnt=0;

while(p!=S.pTop)

{

p++;

cnt++;

}

return cnt;

}

//计算栈中元素的长度

bool StackTraverse(const Stack &S)

{

ElemType \*p = S.pBase;

cout << "Now we will show the traversed result !"<<endl;

while(p != S.pTop)

{

print(\*p);

cout << " ";

p++;

}

cout << endl;

return true;

}

//从栈底向栈顶遍历栈中的所有元素,print为打印当前的ElemType的函数

bool GetTop(Stack &S, ElemType &e)

{

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

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

return true;

}

//取出栈顶的元素

bool Push(Stack &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)

{

cout << "Memory is overflowed !"<<endl;

exit(0);

}

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

S.stacksize += STACKINCREMENT;

}

\*S.pTop = e;

S.pTop++;

return true;

}

//压栈操作

bool Pop(Stack &S, ElemType &e)

{

if (S.pTop <= S.pBase)

{

return false;

}

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

S.pTop--;

return true;

}

//弹栈操作

bool CreateStack(Stack &S, int n, ElemType \*Elem) //传入一个有n个Elem类型的数组

{

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

{

Push(S,Elem[i]);

}

return true;

}

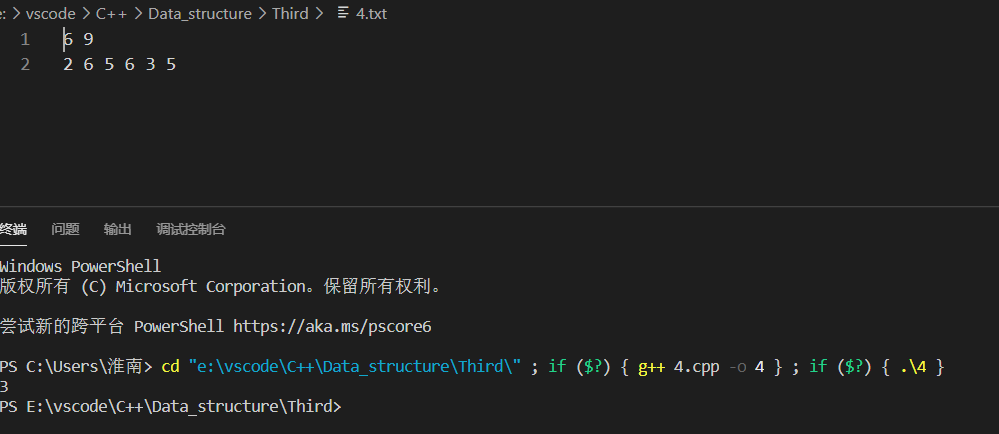
//初始化建栈

4、

题目：问题描述：小明今天生日，他有n块蛋糕要分给朋友们吃，这n块蛋糕（编号为1到n）的重量分别为a1, a2, …, an。小明想分给每个朋友至少重量为k的蛋糕。小明的朋友们已经排好队准备领蛋糕，对于每个朋友，小明总是先将自己手中编号最小的蛋糕分给他，当这个朋友所分得蛋糕的重量不到k时，再继续将剩下的蛋糕中编号最小的给他，直到小明的蛋糕分完或者这个朋友分到的蛋糕的总重量大于等于k。  
　　请问当小明的蛋糕分完时，总共有多少个朋友分到了蛋糕。

算法思想：从前向后遍历，一个临时变量存上加到现在的重量，如果大于K则答案加一，临时变量清零。

运行结果：



结果分析:结果正确，通过CSP测试

附源程序：#include <iostream>

#include <fstream>

using namespace std;

int n=0,k=0;

int a[10005];

int main()

{

fstream file("4.txt",ios::in);

file >> n >>k;

//cin >> n >> k;

int ans=0;

for (int i = 1; i<=n; i++ )

{

file >> a[i];

//cin >> a[i];

}

int cnt=0,temp=0;

while(cnt <= n)

{

temp += a[cnt];

if (temp>=k)

{

ans++;

temp=0;

}

cnt++;

}

if (temp != 0) ans++;

cout << ans;

return 0;

}

5、

题目：问题描述：给定n个数，请找出其中相差（差的绝对值）最小的两个数，输出它们的差值的绝对值。

输入格式：

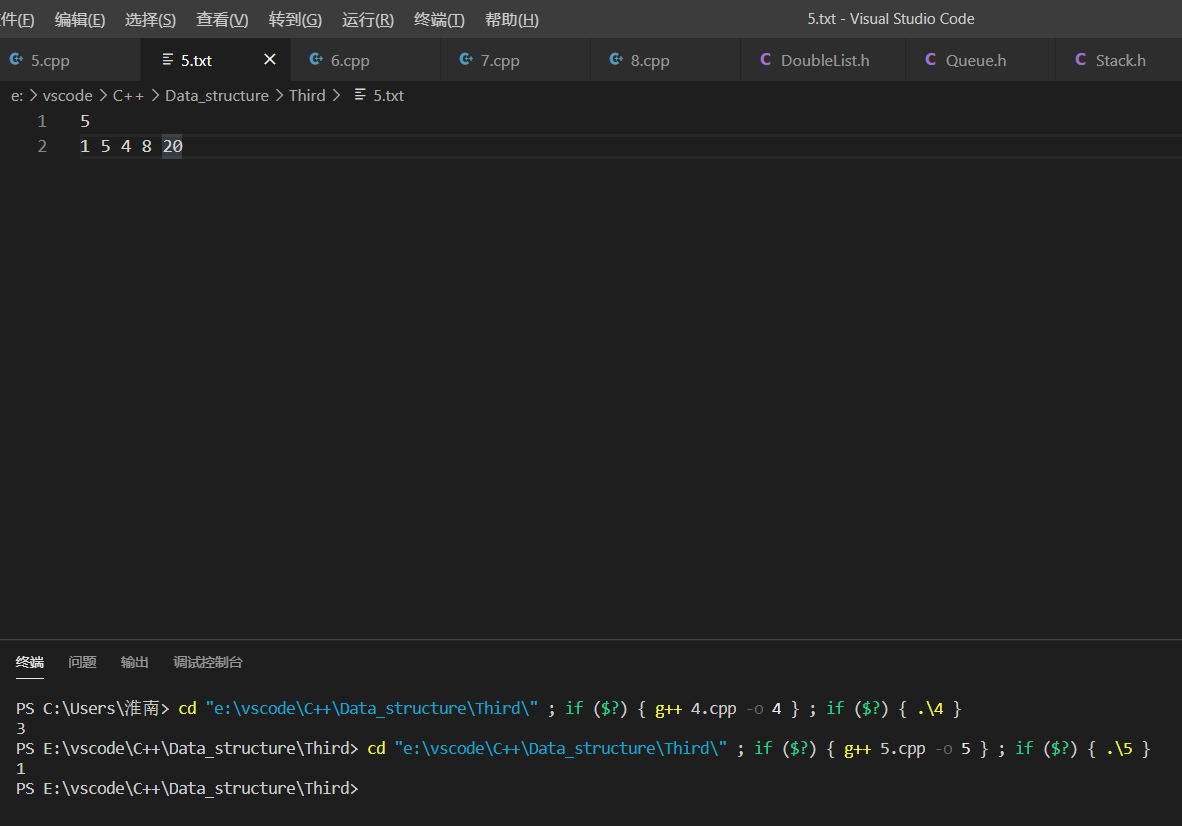
　　输入第一行包含一个整数n。  
　　第二行包含n个正整数，相邻整数之间使用一个空格分隔。

输出格式：

　　输出一个整数，表示答案。

算法思想:先排序，然后选出最小的差值。

运行结果：



结果分析:结果正确，通过CSP测试。

附源程序：/\*问题描述：给定n个数，请找出其中相差（差的绝对值）最小的两个数，输出它们的差值的绝对值。

输入格式：

　　输入第一行包含一个整数n。

　　第二行包含n个正整数，相邻整数之间使用一个空格分隔。

输出格式：

　　输出一个整数，表示答案。\*/

#include <iostream>

#include <fstream>

#include <algorithm>

using namespace std;

int main()

{

fstream file ("5.txt",ios::in);

int n;

int a[1005];

file >> n;

//cin >> n;

for (int i=1; i<=n; i++)

{

file >> a[i];

//cin >> a[i];

}

sort(a+1,a+n+1);

int ans = 9999999;

for (int i = 1; i<n; i++)

{

ans = min(ans,a[i+1]-a[i]);

}

cout << ans;

return 0;

}

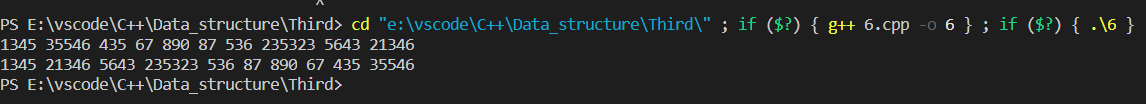
6、

题目：\*已知有一个单向循环链表，其中每个节点中含三个域:prior,data,和next，

其中data为数据域，next为指向后继的指针域，prior也为指针域，但它的值为空(NULL),试编写算法将此单向链表改为双向循环链表，即使prior成为指向前驱结点的指针域\*/

算法思想：设计两个指针，将后一个指针的前驱指向前一个指针指向的位置。

运行结果：



结果分析:运行结果正确，实现了所需功能

附源程序：

//2.32 2.37 2.38

/\*已知有一个单向循环链表，其中每个节点中含三个域:prior,data,和next，

其中data为数据域，next为指向后继的指针域，prior也为指针域，但它的值为空(NULL),试编写算法将此单向链表改为双向循环链表，即使prior成为指向前驱结点的指针域\*/

#define ElemType int

#include <iostream>

#include <cstdlib>

using namespace std;

typedef struct LNode

{

LNode \*prior,\*next;

ElemType data;

}LNode,\*LinkList;

void CreateList(LinkList &L)

{

int a[100] = {-1111,1345,35546,435,67,890,87,536,235323,5643,21346,23464,134564,46421323,111};

int N = 10;

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

L->next = NULL;

L->data = 0;

LNode \*p = NULL, \*q=L;

for (int i = 1; i <= N; i++)

{

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

p->data = a[i];

p->next = NULL;

q->next = p;

q = p;

}

q->next = L->next;

return;

}

void print(LinkList &L)

{

if(L == NULL )

{

return;

}

LNode \*p = L->next;

while(p != NULL )

{

cout << p->data<< " ";

p = p->next;

if(p == L->next)

{

break;

}

}

cout << endl;

return;

}

void ListReverse(LinkList &L)

{

LNode \*p = L->next, \*q = L->next;

do

{

p = p->next;

p->prior = q;

q = p;

//cout << p->data<<endl;

}while(p!=L->next);

//cout << p->data <<" "<< q->data<<endl;

return ;

}

void RevPrint(LinkList &L)

{

if(L == NULL )

{

return;

}

LNode \*p = L->next;

while(p != NULL )

{

cout << p->data<< " ";

p = p->prior;

if(p == L->next)

{

break;

}

}

cout << endl;

return;

}

int main()

{

LinkList L;

CreateList(L);

print(L);

ListReverse(L);

RevPrint(L);

return 0;

}

......

二、未调试成功程序及说明

1、

题目:设带头结点的双向循环链表的线性表L = (a1,a2,a3.....an),设计时间复杂度为O(n)的算法

，将其改造为L=(a1,a3...an...a4,a2)

算法思想：先统计链表的长度,奇数和偶数的操作有所不同，因为连接的点不同。从头开始向遍历，隔一个进行连接操作，到达链表头的前一个结点，然后进行向前访问的操作，进行连接。

错误原因：奇数和偶数的未开处理为处理好，结束条件未正确判断。

附源程序。

/\*2.37 设带头结点的双向循环链表的线性表L = (a1,a2,a3.....an),设计时间复杂度为O(n)的算法

，将其改造为L=(a1,a3...an...a4,a2)\*/

#include "DoubleList.h"

void choice(LinkList &L)

{

if (L == NULL)

{

return ;

}

LNode \*p = L->next, \*q = NULL, \*r = L->next->prior;

cout << r->data<<"hhhhhhh"<<endl;

int length = 0;

do

{

length ++;

p = p->next;

}while(p != L->next);

//cout << "Length is "<<length<<endl;

p = L->next;

if (length % 2 == 1) //当长度为奇数的时候

{

do

{

q = p->next->next;

p->next = q;

q->prior = p;

p = q;

}while(p->next != L->next);

p = p->next;

q = p;

do

{

q = p->prior->prior;

p->next = q;

q->prior = p;

p = q;

}while(p->prior != L->next);

}

else //当长度为偶数的时候

{

do

{

q = p->next->next;

p->next = q;

q->prior = p;

p = q;

cout << "p->data :"<<p->data<<endl;

}while(p != L->next);

cout << "hhh now data is "<< p->data<<endl;

q = r;

cout << q->data<<endl;

do

{

r = q->prior->prior;

p->next = q;

q->prior = p;

p = q;

q = r;

//cout << "p->data :" <<p->data<<endl;

}while(p->prior != L->next);

}

return;

}

//根据题目要求，进行单数双数的选择

int main()

{

LinkList L;

CreateList(L);

print(L);

choice(L);

print(L);

return 0;

}

//双向链表的数据结构

#define ElemType int

#include <iostream>

#include <cstdlib>

using namespace std;

typedef struct LNode

{

LNode \*prior,\*next;

ElemType data;

}LNode,\*LinkList;

void CreateList(LinkList &L)

{

int a[100] = {-1111,1345,35546,435,67,890,87,536,235323,5643,21346,23464,134564,46421323,111};

int N = 10;

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

L->next = NULL;

L->data = 0;

LNode \*p = NULL, \*q=L;

for (int i = 1; i <= N; i++)

{

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

p->data = a[i];

p->next = NULL;

q->next = p;

p->prior = q;

q = p;

}

q->next = L->next;

L->next->prior = q;

return;

}

void print(LinkList &L)

{

if(L == NULL )

{

return;

}

LNode \*p = L->next;

while(p != NULL )

{

cout << p->data<< " ";

p = p->next;

if(p == L->next)

{

break;

}

}

cout << endl;

return;

}

void ListReverse(LinkList &L)

{

LNode \*p = L->next, \*q = L->next;

do

{

p = p->next;

p->prior = q;

q = p;

//cout << p->data<<endl;

}while(p!=L->next);

//cout << p->data <<" "<< q->data<<endl;

return ;

}

void RevPrint(LinkList &L)

{

if(L == NULL )

{

return;

}

LNode \*p = L->next;

while(p != NULL )

{

cout << p->data<< " ";

p = p->prior;

if(p == L->next)

{

break;

}

}

cout << endl;

return;

}

2、

题目：

算法思想：

错误原因：

附源程序。

......

三、代码行数及小结