1释放动态申请的空间

2#include <iostream>

using namespace std;

typedef int QElemType;

typedef struct QNode // 结点结构

{

QElemType data;

struct QNode\* next;

}QNode, \* QueuePtr;

typedef struct // 队列的链表结构

{

QueuePtr front, rear; // 队头、队尾指针

}LinkQueue;

// 构造一个空队列

void InitQueue(LinkQueue\* Q)

{

Q->front = Q->rear = new QNode;

Q->front->next = Q->front;

}

// 销毁队列

void DestroyQueue(LinkQueue\* Q)

{

while (Q->front) //从队头指针开始销毁节点

{

Q->rear = Q->front->next;

delete Q->front;

Q->front = Q->rear;

}

}

//清空队列

void ClearQueue(LinkQueue\* Q)

{

QueuePtr p, q;

Q->rear = Q->front; //队尾指针指向队头指针

p = Q->front->next; //从队头元素开始清除

Q->front->next = Q->front; //队头指针的尾指针指向自己

while (p != Q->front) //循环清楚

{

q = p;

p = p->next;

delete q;

}

}

// 若Q为空队列,则返回TRUE,否则返回FALSE

bool QueueEmpty(LinkQueue Q)

{

if (Q.front == Q.rear)

return true;

else

return false;

}

// 求队列的长度

int QueueLength(LinkQueue Q)

{

int i = 0;

QueuePtr p;

p = Q.front; //p指向队头指针

while (Q.rear != p)

{

i++;

p = p->next; //p指向后一位置

}

return i;

}

// 返回队头元素

bool GetHead(LinkQueue Q, QElemType\* e)

{

QueuePtr p;

if (Q.front == Q.rear) //队空，无法返回

return false;

p = Q.front->next; //将队头元素返回给e

\*e = p->data;

return true;

}

//进队列

void EnQueue(LinkQueue\* Q, QElemType e)

{

QueuePtr s = new QNode;

s->data = e;

s->next = Q->front; //新节点的后继指向队头指针

Q->rear->next = s; //把拥有元素e的新结点s赋值给原队尾结点的后继

Q->rear = s; //把当前的s设置为队尾结点，rear指向s

}

// 出队列

bool DeQueue(LinkQueue\* Q, QElemType\* e)

{

QueuePtr p;

if (Q->front == Q->rear) //队列空

return false;

p = Q->front->next; //将欲删除的队头结点暂存给p

\*e = p->data; //将欲删除的队头结点的值赋值给e

Q->front->next = p->next; //将p->next赋值给队头结点后继

if (Q->rear == p)

{

Q->rear = Q->front; //若队头就是队尾，则删除后将rear指向头结点

}

delete p;

return true;

}

//从队头到队尾依次输出队列中的元素

void QueueTraverse(LinkQueue Q)

{

QueuePtr p;

p = Q.front->next;

cout << "队头到队尾元素依次是 ： ";

while (p != Q.front)

{

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

p = p->next;

}

cout << endl;

}

int main()

{

LinkQueue Q;

InitQueue(&Q);

for (int i = 0; i < 6; i++)

{

EnQueue(&Q, i);

}

int a;

GetHead(Q, &a);

cout << "队列的长度是 ： " << QueueLength(Q) << endl;

cout << "队头元素是 ： " << a << endl;

QueueTraverse(Q);

cout << "-----------------------------------------" << endl;

cout << "前三个元素出队！！！" << endl;

int x, y, z;

DeQueue(&Q, &x);

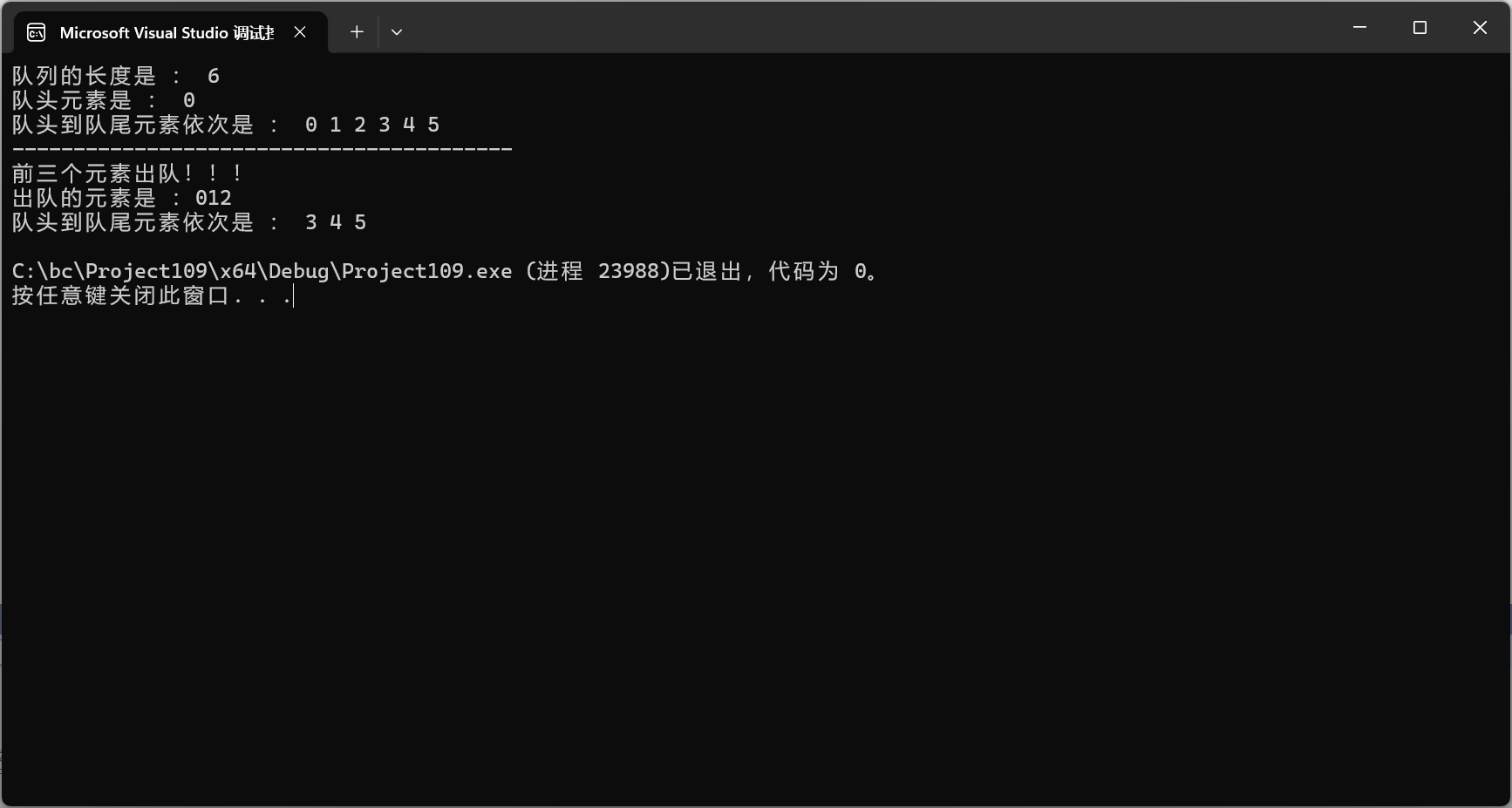
DeQueue(&Q, &y);

DeQueue(&Q, &z);

cout << "出队的元素是 ：" << x << y << z << endl;

QueueTraverse(Q);

}



//如果采用带尾指针的单循环链表作为队列的存储结构，设计算法以实现队列的各运算

public void convert\_List(Node L) {

Node p, t;

p = L.next; //此时指针p指向链表L的第一个结点

while (p.next != null) { //第二个结点不为空，因为逆置，需要链表有两个及以上的元素逆置才有意义

t = p.next;

p.next = t.next; //此时跳过了t结点元素，也就是说t结点元素脱离了链表

t.next = L.next; //将t结点元素放在头结点后面，即t结点元素成为第一个结点

L.next = t; //头结点的指针指向t

}

}

4

public class SingleList {

/\*

\* 将单链表A、B合并成一个递增有序的新链表C

\*/

public void single\_List(Node A, Node B, Node C) {

A = A.next; //指针指向第一个元素

B = B.next; //指针指向第一个元素

while (A != null && B != null) {//需要A、B链表同时不为空

if (A.data < B.data) {

C.next = A;

C = A;

A = A.next;

}

else if (A.data == B.data) {

C.next = A;

A = A.next; //如果A、B的元素相等，则A、B的指针向后移位

B = B.next;

}

else if (A.data > B.data) {

C.next = B;

C = B;

B = B.next;

}

}

if (A != null) { //如果B遍历完了，A未遍历完，则将A的剩余部分接到C链表尾部

C.next = A;

}

if (B != null) { //如果A遍历完了，B未遍历完，则将B的剩余部分接到C链表尾部

C.next = B;

}

}

}

就是说要插入节点，要考虑最后的判断，即这两个链表其中遍历完的时候，要把下一个链表的剩下的借到新链表上。