2022212153 陈祥烨 计科22-2班 第五章作业

第三题

源.cpp

#include"List.h"

int main()

{

int A[18] = { 1,1,2,2,2,3,4,5,5,5,6,6,7,7,8,8,8,9};

List list(A, 18);

int n\_move = list.delete\_repetitive\_element();

cout << "原数组为：1,1,2,2,2,3,4,5,5,5,6,6,7,7,8,8,8,9" << endl;

cout << "现数组为： ";

list.print();

cout << "元素移动次数为：" << n\_move - 1 << endl;

return 0;

}

List.h

#pragma once

#include<iostream>

using namespace std;

const int MAXLEN = 1000;

class List

{

public:

List(int A[],int n); //已有数组构造

List(List& list); //拷贝构造函数

int length()const; //求长度

bool get\_element(const int i, int& x)const; //按序号取元素

int Locate(const int x)const; //按值查找元素，返回元素顺序

bool insert(const int i, const int x); //插入元素，第i个往后移

bool delete\_element(const int i); //删除第i个元素

int delete\_repetitive\_element(); //（已升序或降序排好）删除相同元素

void print(); //打印输出

private:

int data[MAXLEN];

int count;

};

List::List(int A[], int n)

{

count = n;

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

{

data[i] = A[i];

}

}

int List::delete\_repetitive\_element()

{

int number[MAXLEN] = { 0 }; //记录不重复元素的位置

int newcount = 0;

int temp = data[0]; //记录前一个数大小，用于比较是否相同，不相同更新

number[0] = 1;

newcount++;

if (count <= 1)return false;

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

{

if (data[i] == temp)

{

data[i] = -1; //用于标记被删除的元素，如果数据为负数，自定义数据类型，添加判断是否已删除添加的变量（bool）

}

else

{

temp = data[i]; //更新

number[newcount++] = i+1; //记录位置，并更新总数量

}

}

count = newcount;

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

{

data[i] = data[number[i] - 1]; //更新数据

}

return count; //返回元素

}

List::List(List& list)

{

count = list.count;

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

{

data[i] = list.data[i];

}

}

int List::length()const

{

return count;

}

bool List::get\_element(const int i, int& x)const

{

if (i <= 0 || i > count || count == 0)

return false;

x = data[i-1];

return true;

}

int List::Locate(const int x)const

{

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

if (data[i] == x) return (i + 1);

return 0;

}

bool List::insert(const int i, const int x)

{

if (count == MAXLEN)

return false;

if (i < 1 || i > length() + 1)

return false;

for (int j = count; j >= i; j--)

data[j] = data[j - 1];

data[i - 1] = x;

count++;

return true;

}

bool List::delete\_element(const int i)

{

if (length() == 0) return false;

if (i < 1 || i > length()) return false;

for (int j = i + 1; j <= length(); j++)

data[j - 2] = data[j - 1];

count--;

return true;

}

void List::print()

{

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

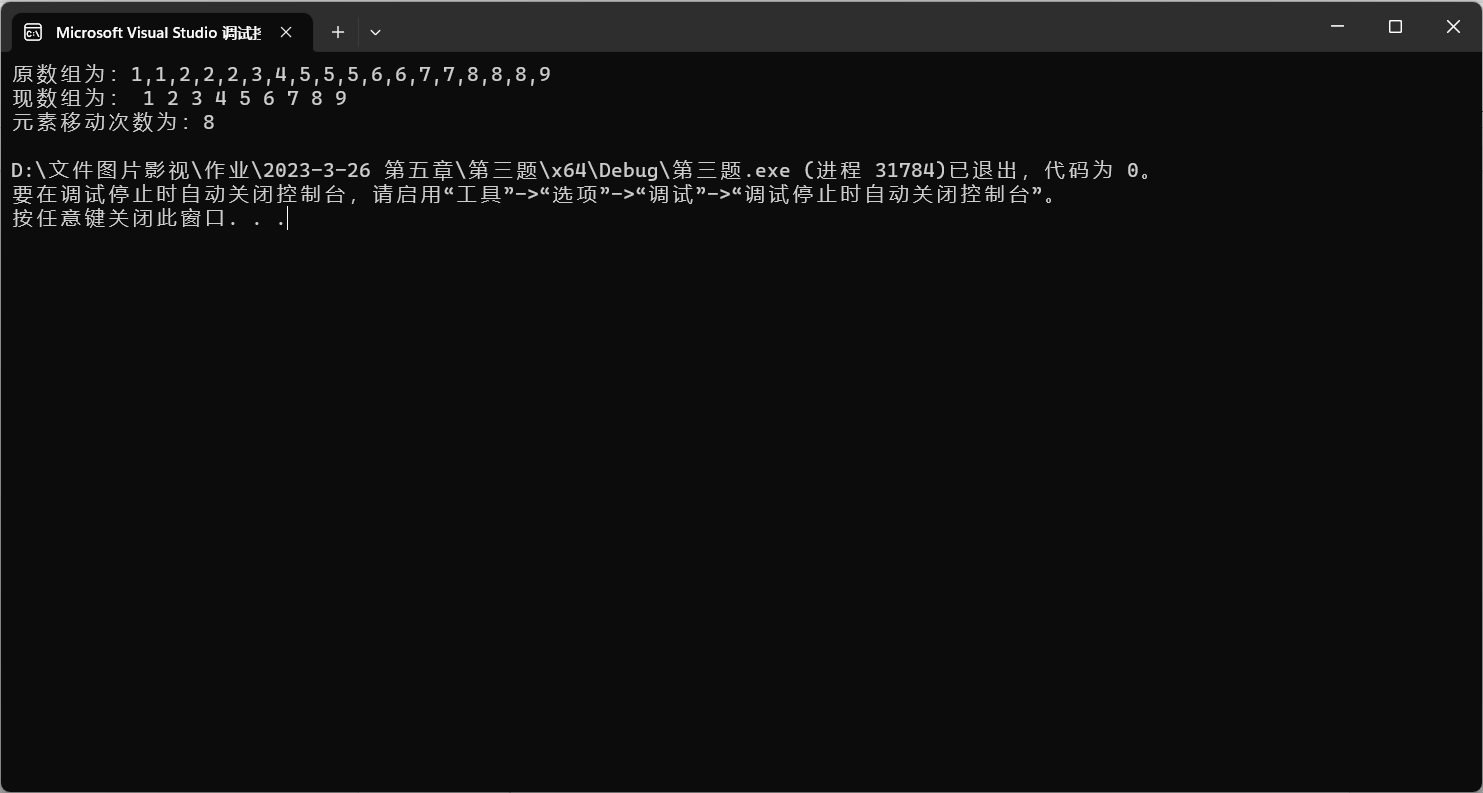
{

cout << data[i] << " ";

}

cout << endl;

}



总结：思路是依次遍历原顺序表，将重复的元素区别于原顺序表的元素（本题的测试集都是正整数，用-1来区别），将不同的元素的所在位置放到新的顺序表（表二）中，遍历完成后，按照表二中记录的位置重新从头依次排序，有多少个不同的元素，移动多少次。

第六题

isson.cpp

#include"List.hpp"

int main()

{

int A[10] = { 1,2, 3,4,5,6,7,8,9,10 };

int B[5] = { 1, 2, 3, 4, 5 };

int C[5] = { 0, 1, 2, 3, 4 };

List<int> lista(A, 10);

List<int> listb(B, 5);

List<int> listc(C, 5);

cout << "ListA: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10" << endl;

cout << "ListB: 1, 2, 3, 4, 5" << endl;

cout << "ListC: 0, 1, 2, 3, 4" << endl;

if (lista.isson(listb))

cout << "ListB is ListA's son" << endl;

else

cout << "ListB isn't ListA's son" << endl;

if (lista.isson(listc))

cout << "ListC is ListA's son" << endl;

else

cout << "ListC isn't ListA's son" << endl;

return 0;

}

List.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct Node

{

T data;

Node<T>\* next;

};

template<class T>

class List //单链表

{

public:

List(); //默认构造函数

List(List<T>& list); //拷贝构造函数

List(const T A[],int n); //数组构造函数

int length()const; //求长度

~List(); //析构函数

bool get\_element(const int i, T& x)const; //按序号取元素

Node<T>\* locate(const T x)const; //搜索元素

bool insert(const int i, const T x); //插入元素(第i个之前，原本的第i个元素变成第i+1个)

bool delete\_end(); //删除尾元素

Node<T>\* get\_head() { return head; } //读取链表表头指针

Node<T>\* get\_rear() { return rear; } //读取链表表尾指针

bool isson(List<T> A); //判断是否为子集

private:

int count;

Node<T>\* head;

Node<T>\* rear;

};

template<class T>

List<T>::List()

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

}

template<class T>

List<T>::List(List<T>& list)

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

Node<T>\* PA = list.get\_head()->next;

Node<T>\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node<T>\* s = new Node<T>; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

rear = s;

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

template<class T>

inline List<T>::List(const T A[], int n)

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = n;

Node<T>\* PA = head; //设置当前表，即B表尾指针

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

{

Node<T>\* s = new Node<T>; //产生结点

s->data = A[i]; //复制结点的值

PA->next = s; // 插入复制的结点到表尾

PA = s; //重新指示表尾，以便下一个元素插入

rear = s;

}

PA->next = NULL; //将B表尾结点的后继指针置为空

}

template<class T>

List<T>::~List()

{

while (head != rear)

{

Node<T>\* temp = head;

head = head->next;

delete temp;

}

delete head;

}

template<class T>

int List<T>::length() const

{

return count;

}

template<class T>

bool List<T>::get\_element(const int i, T& x)const

{

Node<T>\* p = head->next;

int j = 1;

while (p != NULL && j != i) //不是目标结点或不空，就继续搜索

{

p = p->next;

j++;

}

if (p == NULL)return false;

x = p->data;

return true;

}

template<class T>

Node<T>\* List<T>::locate(const T x)const

{

Node<T>\* p = head->next;

while (p != NULL)

{

if (p->data == x)return p;

else p = p->next;

}

return NULL;

}

template<class T>

bool List<T>::insert(const int i, const T x)

{

Node<T>\* p = head;

int j = 0;

for (; j != i - 1 && p != NULL; j++)

p = p->next;

if (i < 1 || i>count + 1) //等价与if(p == NULL)

return false;

Node<T>\* s = new Node<T>; //产生新结点

s->data = x;

s->next = p->next;

p->next = s;

count++;

return true;

}

template<class T>

bool List<T>::delete\_end()

{

if (count == 0)

return false;

Node<T>\* p = head;

while (p->next->next != NULL)

p = p->next;

rear = p;

delete p->next;

p->next = NULL;

return true;

}

template<class T>

bool List<T>::isson(List<T> A)

{

Node<T>\* PA = head->next;

Node<T>\* PB = A.get\_head()->next;

if (A.length() == 0)return true;

while (1)

{

if (PB->data != PA->data)

PA = PA->next;

else

PB = PB->next;

if (PA == NULL)

return false;

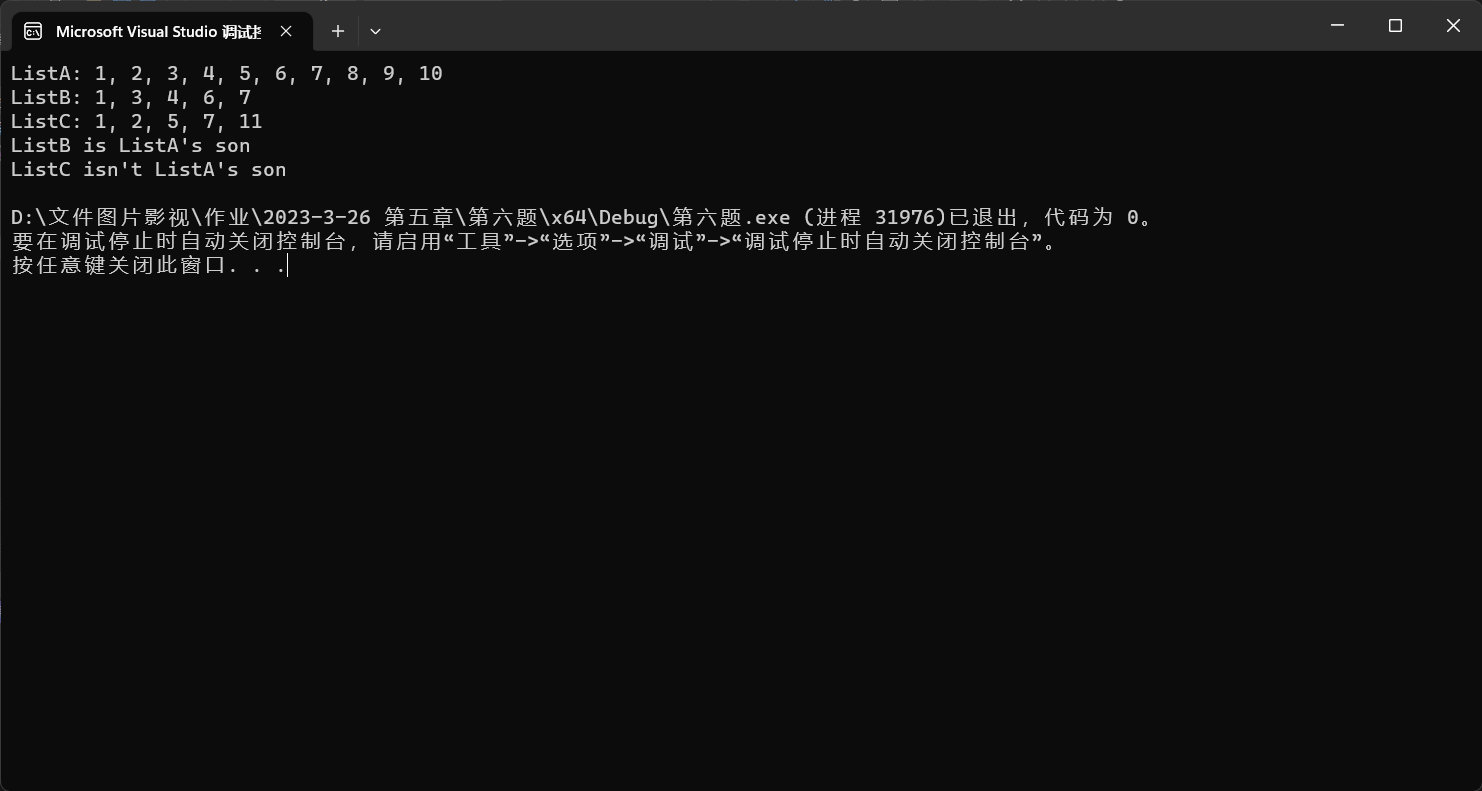
if (PB == NULL)

return true;

}

}





总结：因为是已经排好序的，所以直接遍历比较是否相等。设置两个指针，比较元素是否相等，相同则判断下一个是不是相同，子链表指针后移；不相同则再到父链表里找有没有相同的，父链表指针后移。时间复杂度主要有链表长度决定，基本小于等于两链表长度之和。

第七题

getintersection.cpp

#include"List.hpp"

int main()

{

int A[5] = { 0,1,2,3,4 };

int B[5] = { 3,4,5,6,7 };

List<int> listA(A, 5);

List<int> listB(B, 5);

List<int> listA\_B((listA.getintersection(listB)));

cout << "listA: " << endl;

listA.print();

cout << "listB: " << endl;

listB.print();

cout << "listA\_B: " << endl;

listA\_B.print();

return 0;

}

List.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct Node

{

T data;

Node<T>\* next;

};

template<class T>

class List //单链表

{

public:

List(); //默认构造函数

List(List<T>& list); //拷贝构造函数

List(const T A[],int n); //数组构造函数

int length()const; //求长度

~List(); //析构函数

bool get\_element(const int i, T& x)const; //按序号取元素

Node<T>\* locate(const T x)const; //搜索元素

bool insert(const int i, const T x); //插入元素(第i个之前，原本的第i个元素变成第i+1个)

bool delete\_end(); //删除尾元素

Node<T>\* get\_head() { return head; } //读取链表表头指针

Node<T>\* get\_rear() { return rear; } //读取链表表尾指针

List<T>& getintersection(List<T> A); //求交集

bool append(const T x); //尾增

bool print() const; //输出元素

private:

int count;

Node<T>\* head;

Node<T>\* rear;

};

template<class T>

List<T>::List()

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

}

template<class T>

List<T>::List(List<T>& list)

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

Node<T>\* PA = list.get\_head()->next;

Node<T>\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node<T>\* s = new Node<T>; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

rear = s;

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

template<class T>

inline List<T>::List(const T A[], int n)

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = n;

Node<T>\* PA = head; //设置当前表，即B表尾指针

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

{

Node<T>\* s = new Node<T>; //产生结点

s->data = A[i]; //复制结点的值

PA->next = s; // 插入复制的结点到表尾

PA = s; //重新指示表尾，以便下一个元素插入

rear = s;

}

PA->next = NULL; //将B表尾结点的后继指针置为空

}

template<class T>

List<T>::~List()

{

while (head != rear)

{

Node<T>\* temp = head;

head = head->next;

delete temp;

}

delete head;

}

template<class T>

int List<T>::length() const

{

return count;

}

template<class T>

bool List<T>::get\_element(const int i, T& x)const

{

Node<T>\* p = head->next;

int j = 1;

while (p != NULL && j != i) //不是目标结点或不空，就继续搜索

{

p = p->next;

j++;

}

if (p == NULL)return false;

x = p->data;

return true;

}

template<class T>

Node<T>\* List<T>::locate(const T x)const

{

Node<T>\* p = head->next;

while (p != NULL)

{

if (p->data == x)return p;

else p = p->next;

}

return NULL;

}

template<class T>

bool List<T>::insert(const int i, const T x)

{

Node<T>\* p = head;

int j = 0;

for (; j != i - 1 && p != NULL; j++)

p = p->next;

if (i < 1 || i>count + 1) //等价与if(p == NULL)

return false;

Node<T>\* s = new Node<T>; //产生新结点

s->data = x;

s->next = p->next;

p->next = s;

count++;

return true;

}

template<class T>

bool List<T>::delete\_end()

{

if (count == 0)

return false;

Node<T>\* p = head;

while (p->next->next != NULL)

p = p->next;

rear = p;

delete p->next;

p->next = NULL;

return true;

}

template<class T>

List<T>& List<T>::getintersection(List<T> A)

{

List<T>\* list = new List<T>;

Node<T>\* PA = head->next;

Node<T>\* PB = A.get\_head()->next;

if (A.length() == 0)return \*list;

for(;PB!= NULL;PB = PB->next)

for (; PA != NULL; PA = PA->next)

if (PA->data == PB->data)

{

list->append(PB->data);

break;

}

return \*list;

}

template<class T>

bool List<T>::append(const T x)

{

Node<T>\* s = new Node<T>;

s->data = x;

s->next = NULL;

rear->next = s;

rear = s;

count++;

return true;

}

template<class T>

bool List<T>::print() const //输出元素

{

//输出空集

if (count == 0)

{

cout << "{ }";

return true;

}

//输出空集

Node<T>\* PA = head->next;

cout << "{ ";

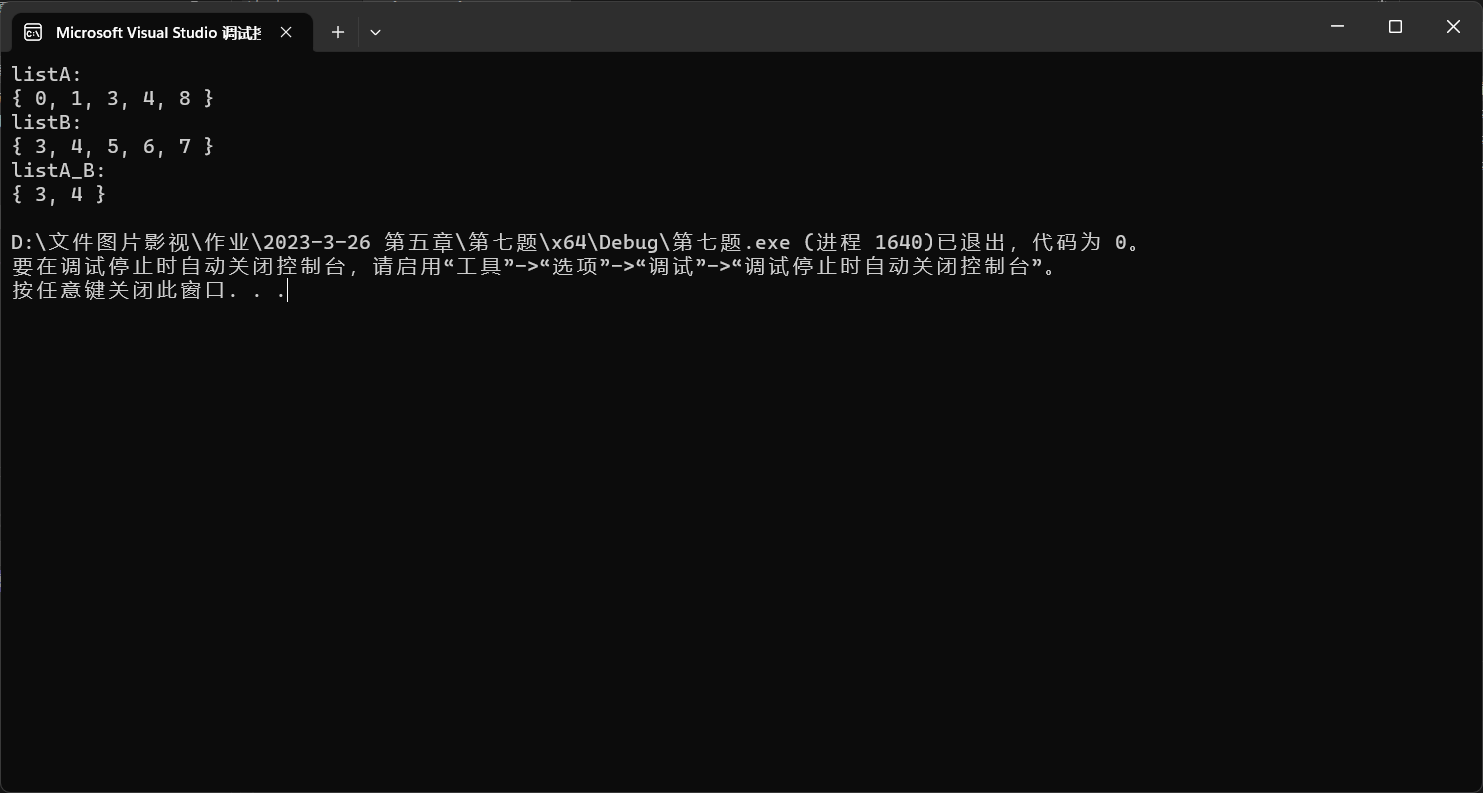
for (; PA->next != NULL; PA = PA->next)

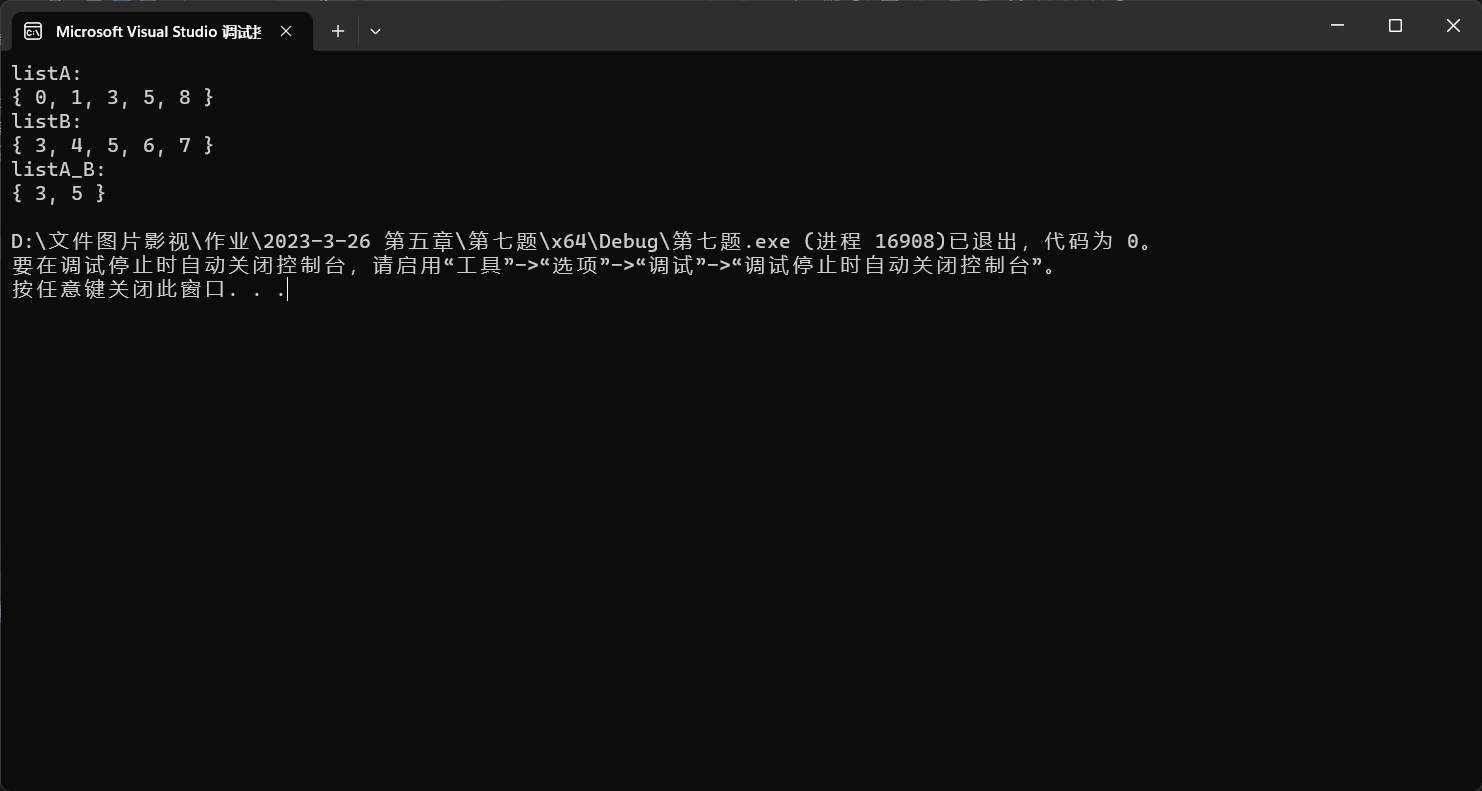
cout << PA->data << ", ";

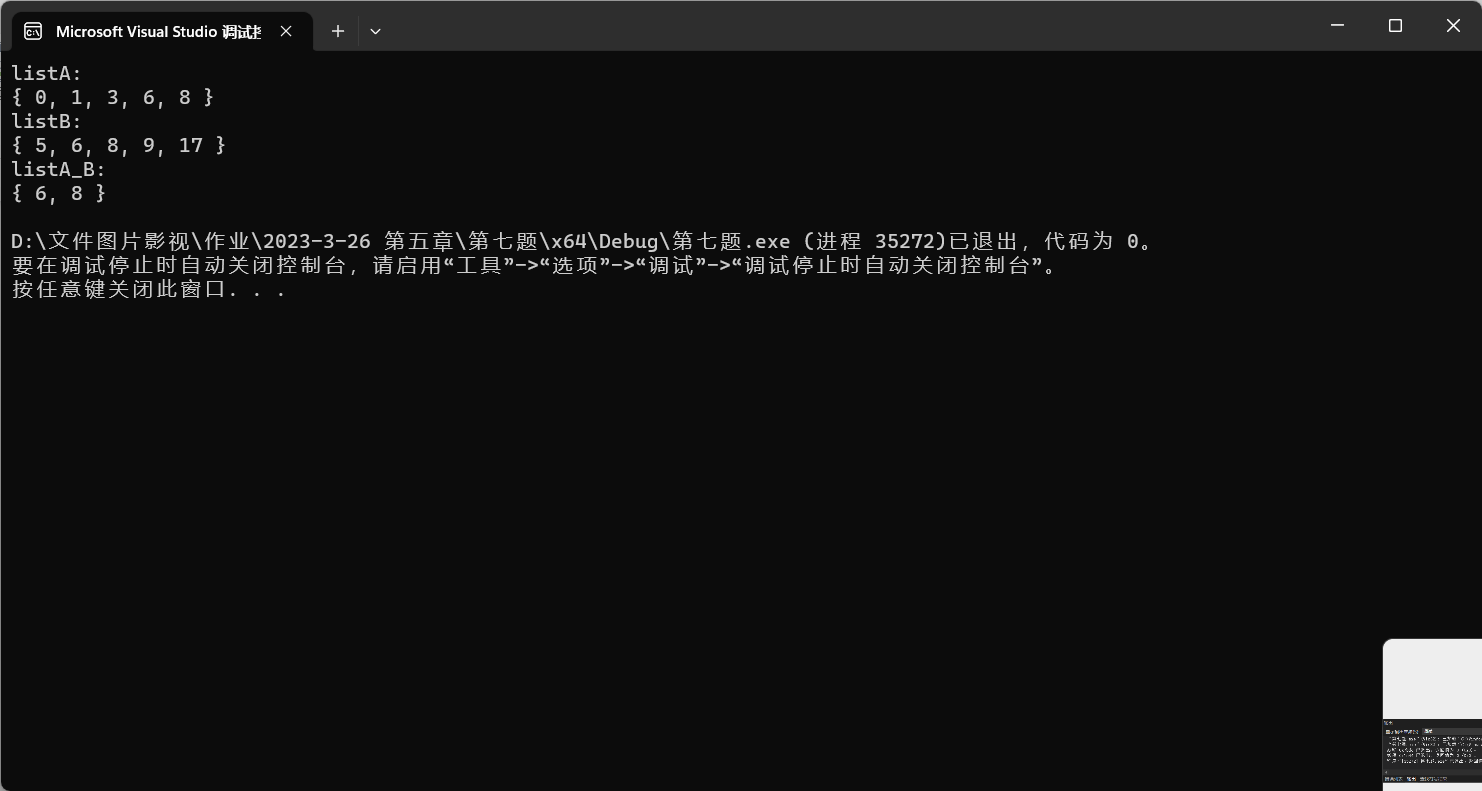
cout << PA->data << " }" << endl;

return true;

}







总结：先设一个空链表C用于存储相同元素，因为是已知的递增数列，用两个指针遍历，取A的元素与B做判断，相同则C尾增，不相同则A偏移，取下一个问题。但这样会有个问题，就是B的指针碰到不同的元素会指到表尾，A后面重复的元素就不会被记录，所以需要一个指针储存上一个相同的B指针，每次做完判断后重置B指针。

时间复杂度是两个循环x2（x为链表长）。

第八题

invert.cpp

#include"Double List.hpp"

int main()

{

int A[5] = { 1,2,3,4,5 };

DoubleList<int> list1(A, 5);

cout << "list1:" << endl;

list1.print();

cout << "After inverted, list1:" << endl;

list1.invert();

list1.print();

return 0;

}

Double List.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct DoubleNode

{

DoubleNode<T>\* prior;

T data;

DoubleNode<T>\* next;

};

template<class T>

class DoubleList

{

public:

DoubleList(); //构造函数

DoubleList(const T A[], const int n); //数组构造函数

DoubleList(DoubleList<T>& list); //拷贝构造函数

~DoubleList(); //析构函数

DoubleNode<T>\* get\_front() const; //得到头结点指针

DoubleNode<T>\* get\_rear() const; //得到尾结点指针

int length() const; //得到链表长度

void print() const; //控制台输出元素

bool invert(); //倒置

private:

DoubleNode<T>\* front;

DoubleNode<T>\* rear;

int count;

};

template<class T>

DoubleList<T>::DoubleList()

{

DoubleNode<T>\* node = new DoubleNode<T>;

rear = node;

front = node;

node->next = front;

node->prior = rear;

count = 0;

}

template<class T>

DoubleList<T>::DoubleList(const T A[], const int n)

{

DoubleNode<T>\* PA = new DoubleNode<T>;

rear = PA;

front = PA;

PA->next = front;

PA->prior = rear;

count = 0;

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

{

DoubleNode<T>\* node = new DoubleNode<T>;

PA->next = node; //更新前结点

node->data = A[i]; //更新节点变量

node->prior = PA;

node->next = front;

rear = node; //更新链表变量

front->prior = node;

count++;

PA = node; //更新临时结点，为下一次添加做准备

}

}

template<class T>

DoubleList<T>::DoubleList(DoubleList<T>& doublelist)

{

DoubleNode<T>\* PA = new DoubleNode<T>;

DoubleNode<T>\* PB = doublelist.get\_front()->next;

//构造头结点

rear = PA;

front = PA;

PA->next = front;

PA->prior = rear;

count = 0;

//添加数据

for (; PB != doublelist.get\_front(); PB = PB->next)

{

DoubleNode<T>\* node = new DoubleNode<T>;

PA->next = node; //更新前结点

node->data = PB->data; //更新节点变量

node->prior = PA;

node->next = front;

rear = node; //更新链表变量

front->prior = node;

count++;

PA = node; //更新临时结点，为下一次添加做准备

}

}

template<class T>

DoubleNode<T>\* DoubleList<T>::get\_front() const

{

return front;

}

template<class T>

DoubleNode<T>\* DoubleList<T>::get\_rear() const

{

return rear;

}

template<class T>

int DoubleList<T>::length() const

{

return count;

} //得到链表长度

template<class T>

void DoubleList<T>::print() const

{

DoubleNode<T>\* PA = front->next;

if (count == 0)

{

cout << "{ }" << endl;

return;

}

cout << "{ ";

for (; PA->next != front; PA = PA->next)

{

cout << PA->data << ", ";

}

cout << PA->data << " }" << endl;

}

template<class T>

inline bool DoubleList<T>::invert()

{

if (count < 2)return true;

rear = front->next;

DoubleNode<T>\* temp,\*PA = front;

while (PA != rear)

{

temp = PA->prior;

PA->prior = PA->next;

PA->next = temp;

PA = PA->next;

}

temp = PA->prior;

PA->prior = PA->next;

PA->next = temp;

return true;

}

template<class T>

DoubleList<T>::~DoubleList()

{

DoubleNode<T>\* PA = rear;

DoubleNode<T>\* temp = PA->prior;

while (PA->next != front)

{

delete PA;

PA = temp;

temp = PA->prior;

count--;

}

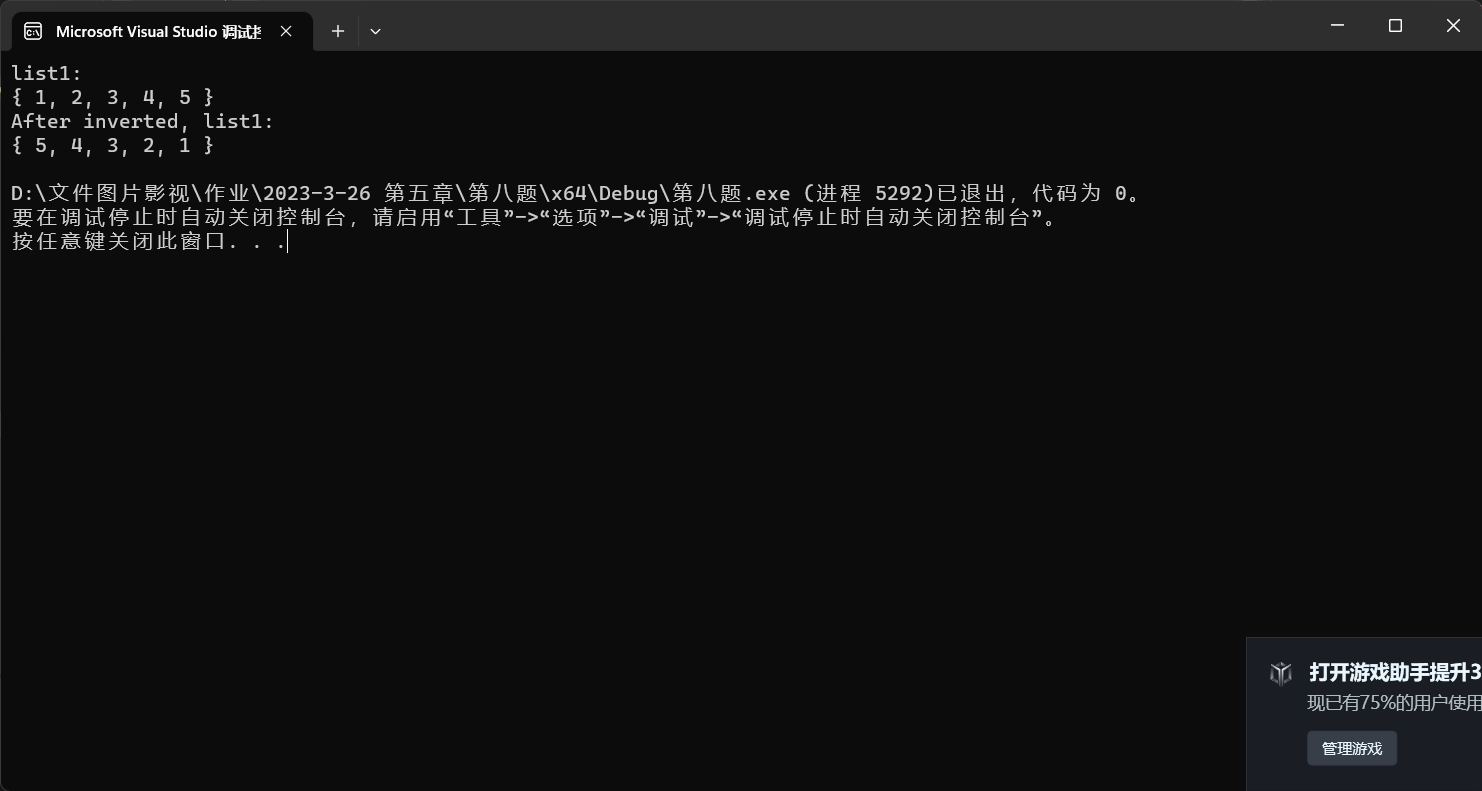
delete PA;

delete front;

front = NULL;

rear = NULL;

}



总结：倒置其实只需要访问顺序倒置就可以了，头结点时空指针让其next指向rear，其prior指向原先头结点的下一个，rear指向原头结点的下一个。但需要注意的是，每个元素的prior和next需要做个调换就可以了。

第九题

strcmp.cpp

#include"List.h"

#include<iostream>

using namespace std;

int strcmp(List list1, List list2) //1大为1,1小为-1，相等为0

{

Node\* p1 = list1.get\_head()->next;

Node\* p2 = list2.get\_head()->next;

while (1)

{

if (p1->data == p2->data)

{

p1 = p1->next;

p2 = p2->next;

if (p1 == NULL && p2 == NULL)

return 0;

else if (p1 == NULL)

return -1;

else if (p2 == NULL)

return 1;

}

else if (p1->data > p2->data) return 1;

else return -1;

}

}

void print\_result(List list1, List list2)

{

int result = strcmp(list1, list2);

if (result == 1)

cout << "B is big" << endl;

else if (result == 0)

cout << "is equal" << endl;

else

cout << "B is small" << endl;

}

int main()

{

char A[3] = { 'a','b','c' };

char B[3] = { 'a','b','d' };

char C[3] = { 'a','b','e' };

char D[4] = { 'a','b','d','e'};

cout << "A[3] = { 'a','b','c' };" << endl;

cout << "B[3] = { 'a','b','d' };" << endl;

cout << "C[3] = { 'a','b','e' };" << endl;

cout << "D[4] = { 'a','b','d','e'};" << endl;

List listA(A, 3);

List listB(B, 3);

List listC(C, 3);

List listD(D, 4);

cout << "compare B with A:" << endl;

print\_result(listB, listA);

cout << "compare B with B:" << endl;

print\_result(listB, listB);

cout << "compare B with C:" << endl;

print\_result(listB, listC);

cout << "compare B with D:" << endl;

print\_result(listB, listD);

return 0;

}

List.h

#pragma once

struct Node

{

char data;

Node\* next;

};

class List //单链表

{

public:

//构造

List(); //默认构造函数

List(List& list); //拷贝构造函数

List(char A[], int n); //数组构造函数

~List(); //析构函数

//获取链表成员变量

int length()const; //求长度

Node\* get\_head() { return head; } //读取链表表头指针

Node\* get\_rear() { return rear; } //读取链表表尾指针

//操作元素

bool get\_element(const int i, char& x)const;//按序号取元素

Node\* locate(const char x)const; //搜索元素

bool insert(const int i, const char x); //插入元素(第i个之前，原本的第i个元素变成第i+1个)

bool delete\_end(); //删除尾元素

bool append(const char x); //尾增

//操作链表

void copy(List A); //A链表复制到B链表

bool isequal(List A); //判断链表是否相等

private:

int count;

Node\* head;

Node\* rear;

};

List.cpp

#include"List.h"

#include<iostream>

using namespace std;

List::List()

{

head = new Node;

head->next = NULL;

rear = head;

count = 0;

}

List::List(List& list)

{

head = new Node;

head->next = NULL;

rear = head;

count = 0;

Node\* PA = list.get\_head()->next;

Node\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node\* s = new Node; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

rear = s;

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

List::List(char A[], int n)

{

head = new Node;

head->next = NULL;

rear = head;

Node\* PA = head;

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

{

Node\* s = new Node;

s->data = A[i];

s->next = NULL;

PA->next = s;

PA = s;

}

rear = PA;

count = n;

}

List::~List()

{

while (head != rear)

{

Node\* temp = head;

head = head->next;

delete temp;

}

delete head;

}

int List::length() const

{

return count;

}

bool List::get\_element(const int i, char& x)const

{

Node\* p = head->next;

int j = 1;

while (p != NULL && j != i) //不是目标结点或不空，就继续搜索

{

p = p->next;

j++;

}

if (p == NULL)return false;

x = p->data;

return true;

}

Node\* List::locate(const char x)const

{

Node\* p = head->next;

while (p != NULL)

{

if (p->data == x)return p;

else p = p->next;

}

return NULL;

}

bool List::insert(const int i, const char x)

{

Node\* p = head;

int j = 0;

for (; j != i - 1 && p != NULL; j++)

p = p->next;

if (i < 1 || i>count + 1) //等价与if(p == NULL)

return false;

Node\* s = new Node; //产生新结点

s->data = x;

s->next = p->next;

p->next = s;

count++;

return true;

}

bool List::delete\_end()

{

if (count == 0)

return false;

Node\* p = head;

while (p->next->next != NULL)

p = p->next;

rear = p;

delete p->next;

p->next = NULL;

return true;

}

void List::copy(List A)

{

Node\* PA = A.get\_head()->next;

Node\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node\* s = new Node; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

bool List::append(const char x)

{

Node\* s = new Node;

s->data = x;

s->next = NULL;

rear->next = s;

rear = s;

count++;

return true;

}

bool List::isequal(List A)

{

if (count != A.count)

return false;

else

{

Node\* PA = head->next;

Node\* PB = A.head->next;

while (PA != NULL)

{

if (PA->data != PB->data)

return false;

PA = PA->next;

PB = PB->next;

}

return true;

}

}

