## 2021．9．17 课后编程训练题（单链表+循环链表）

#### Link.h

#ifndef LINK\_H

#define LINK\_H

#include <cstddef>

/\*单向链表\*/

template <typename E>

class Link

{

private:

static Link<E>\* freelist;

public:

E element;

Link \*next;

Link(const E& elemval , Link\* nextval = nullptr){ element = elemval; next = nextval; }

Link(Link\* nextval = nullptr) { next = nextval; }

void\* operator new(size\_t){

if(freelist == nullptr) return ::new Link;

Link<E>\* temp = freelist;

freelist =freelist->next;

return temp;

}

void operator delete(void\* ptr){

((Link<E>\*)ptr)->next = freelist;

freelist = (Link<E>\*)ptr;

}

};

template <typename E>

Link<E>\* Link<E>::freelist = nullptr;

/\*双向链表版本

template <typename E>

class Link

{

private:

static Link<E>\* freelist;

public:

E element;

Link \*next;

Link \*prev;

Link(const E& it , Link\* prevp , Link\* nextp)

{

element = it;

prev = prevp;

next = nextp;

}

Link(Link\* prevp = nullptr , Link\* nextp = nullptr)

{

next = nextp;

prev = prevp;

}

void\* operator new(size\_t){

if(freelist == nullptr) return ::new Link;

Link<E>\* temp = freelist;

freelist = freelist->next;

return temp;

}

void operator delete(void\* ptr){

((Link<E>\*)ptr)->next = freelist;

freelist = (Link<E>\*)ptr;

}

};

template <typename E>

Link<E>\* Link<E>::freelist = nullptr;

\*/

#endif // LINK\_H

#### List.h

#ifndef LIST\_H

#define LIST\_H

#include <cassert>

#include <cstring>

#include <iostream>

template <typename E>

class List

{

private:

void operator = (const List&) {}

List(const List&) {}

public:

List() {};

virtual ~List() {};

virtual void clear() = 0;

virtual void insert(const E& item) = 0;

virtual E remove() = 0;

virtual void moveToStart() = 0;

virtual void moveToEnd() = 0;

virtual void prev() = 0;

virtual void next() = 0;

virtual int length() const = 0;

virtual int currPos() const = 0;

virtual void moveToPos(int pos) = 0;

virtual const E& getValue() = 0;

};

#endif // LIST\_H

#### Llist.h

#ifndef LLIST\_H

#define LLIST\_H

#define defaultsize 100

#include "link.h"

#include "list.h"

using namespace std;

template <typename E>

class LList : public List<E>

{

private:

Link<E>\* head;

Link<E>\* tail;

Link<E>\* curr;

int cnt;

/\*单向链表版本\*/

void init()

{

curr = tail = head = new Link<E>;

cnt = 0;

}

// void init() //单向循环链表版本

// {

// curr = tail = head = new Link<E>;

// tail->next = head;

// head->next = head;

// cnt = 0;

// }

void removeall()

{

while(head != nullptr){

curr = head;

head = head->next;

delete curr;

}

}

// void removeall() //单向循环链表版本

// {

// while(head->next != head){

// curr = head;

// head = head->next;

// delete curr;

// }

// }

/\*双向链表版本

void init()

{

curr = head = new Link<E>;

tail = new Link<E>;

head->next = tail;

tail->prev = head;

cnt = 0;

}

// void init() //双向循环链表版本

// {

// curr = head = new Link<E>;

// tail = new Link<E>;

// head->next = tail ; head->prev = tail;

// tail->prev = head ; tail->next = head;

// cnt = 0;

// }

void removeall()

{

while(head != nullptr){

curr = head;

head = head->next;

delete curr;

}

}

\*/

public:

void Assert(bool val, string s);

LList(int size =defaultsize);

~LList();

void print() const;

void clear();

void insert(const E& it);

void append(const E& it);

E remove();

void moveToStart();

void moveToEnd();

void prev();

void next();

int length() const;

int currPos() const ;

void moveToPos(int pos);

const E& getValue();

};

#endif // LLIST\_H

#endif // LLIST\_H

#### Llist.cpp

#include "llist.h"

#include "list.h"

/\*单向链表版本\*/

template <typename E>

void LList<E>::Assert(bool val, string s)

{

if (!val) {

cout << "Assertion Failed: " << s << endl;

exit(-1);

}

}

template <typename E>

LList<E>::LList(int size){ init(); }

template <typename E>

LList<E>::~LList() { removeall(); }

template <typename E>

void LList<E>::print() const{

Link<E>\* temp;

temp = head->next;

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

{

cout<<temp->element<<" ";

temp = temp->next;

}

}

template <typename E>

void LList<E>::clear() {

removeall();

init();

}

template <typename E>

void LList<E>::insert(const E& it){

curr->next = new Link<E>(it , curr->next);

if(tail == curr) tail = curr->next;

cnt++;

}

//template <typename E> //curr指向currelement版本

//void LList<E>::insert(const E& it){

// E temp = curr->element;

// curr->next = new Link<E>(temp,curr->next);

// if(tail == curr) tail = curr->next;

// curr->element = it;

// cnt++;

//}

template <typename E>

void LList<E>::append(const E& it){

tail = tail->next = new Link<E>(it , nullptr);

cnt++;

}

//template <typename E>

//void LList<E>::append(const E& it){ //单向循环链表版本

// tail = tail->next = new Link<E>(it , head->next);

// cnt++;

//}

template <typename E>

E LList<E>::remove(){

Assert(curr->next != nullptr , "No element");

E it = curr->next->element;

Link<E>\* ltemp = curr->next;

if(tail == curr->next) tail = curr;

curr->next = curr->next->next;

delete ltemp;

cnt--;

return it;

}

//template <typename E> //curr指向currelement版本

//E LList<E>::remove(){

// E it = curr->element;

// Link<E>\* ltemp = curr;

// curr = head;

// while(curr->next != ltemp)

// curr = curr->next;

// if(ltemp == tail)

// tail = curr; //若curr==tail tail前移

// curr->next = curr->next->next;

// delete ltemp;

// cnt--;

// return it;

//}

template <typename E>

void LList<E>::moveToStart(){ curr = head; }

template <typename E>

void LList<E>::moveToEnd(){ curr = tail; }

template <typename E>

void LList<E>::prev(){

if(curr == head) return;

Link<E>\* temp = head;

while (temp->next != curr) temp = temp->next;

curr = temp;

}

//template <typename E>

//void LList<E>::prev(){ //单向循环链表

// Link<E>\* temp = head;

// while (temp->next != curr) temp = temp->next;

// curr = temp;

//}

template <typename E>

void LList<E>::next(){

if(curr->next != tail) curr = curr->next;

}

//template <typename E>

//void LList<E>::next(){ //单向循环链表

// curr = curr->next;

//}

template <typename E>

int LList<E>::length() const { return cnt; }

template <typename E>

int LList<E>::currPos() const {

Link<E>\* temp = head;

int i;

for( i = 0 ; curr != temp ; i++)

temp = temp->next;

return i;

}

template <typename E>

void LList<E>::moveToPos(int pos){

Assert(pos >= 0 && pos <= cnt , "Position out of range");

curr = head;

for(int i = 0 ; i < pos ; i++) curr = curr->next;

}

//template <typename E> //curr指向currelement版本

//void LList<E>::moveToPos(int pos){

// Assert(pos >= 0 && pos <= cnt , "Position out of range");

// curr = head;

// for(int i = 0 ; i <= pos ; i++) curr = curr->next;

//}

template <typename E>

const E& LList<E>::getValue(){

Assert(curr->next != nullptr , "No value");

return curr->next->element;

}

//template <typename E>

//const E& LList<E>::getValue(){ //curr指向currelement版本

// Assert(curr != nullptr , "No value");

// return curr->element;

//}

/\*双向链表版本

template <typename E>

void LList<E>::Assert(bool val, string s)

{

if (!val) {

cout << "Assertion Failed: " << s << endl;

exit(-1);

}

}

template <typename E>

LList<E>::LList(int size){ init(); }

template <typename E>

LList<E>::~LList() { removeall(); }

template <typename E>

void LList<E>::print() const{

Link<E>\* temp;

temp = head->next;

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

{

cout<<temp->element<<" ";

temp = temp->next;

}

}

template <typename E>

void LList<E>::clear() {

removeall();

init();

}

template <typename E>

void LList<E>::insert(const E& it){

curr->next = curr->next->prev = new Link<E>(it , curr , curr->next);

cnt++;

}

template <typename E>

void LList<E>::append(const E& it){

tail->prev = tail->prev->next = new Link<E>(it,tail->prev,tail);

cnt++;

}

template <typename E>

E LList<E>::remove(){

if(curr->next == tail)

return NULL;

E it = curr->next->element;

Link<E>\* ltemp = curr->next;

curr->next->next->prev = curr;

curr->next = curr->next->next;

delete ltemp;

cnt--;

return it;

}

template <typename E>

void LList<E>::moveToStart(){ curr = head; }

template <typename E>

void LList<E>::moveToEnd(){ curr = tail->prev->prev; }

template <typename E>

void LList<E>::prev(){ //双向循环链表版本

if(curr != head)

curr = curr->prev;

else

curr = tail->prev->prev;

}

template <typename E>

void LList<E>::next(){ //双向循环链表版本

if(curr->next != tail)

curr = curr->next;

else

curr = head;

}

template <typename E>

int LList<E>::length() const { return cnt; }

template <typename E>

int LList<E>::currPos() const {

Link<E>\* temp = head;

int i;

for( i = 0 ; curr != temp ; i++)

temp = temp->next;

return i;

}

template <typename E>

void LList<E>::moveToPos(int pos){

Assert(pos >= 0 && pos <= cnt , "Position out of range");

curr = head;

for(int i = 0 ; i < pos ; i++) curr = curr->next;

}

template <typename E>

const E& LList<E>::getValue(){ //双向循环链表版本

if(curr->next != tail)

return curr->next->element;

else

curr = head;

return curr->next->element;

}\*/

#### 1. Write a function to reverses the order of the elements on the singly linked list.

#include <iostream>

#include "llist.cpp"

using namespace std;

template <typename E>

void myReverse(LList<E> &t)

{

int len = t.length();

int right = len-1 ; int left = 0;

for(; left < len/2 ; left++ , right--)

{

E t1 , t2;

t.moveToPos(left) ; t1 = t.getValue();

t.moveToPos(right) ; t2 = t.getValue();

t.remove() ; t.insert(t1);

t.moveToPos(left) ; t.remove() ; t.insert(t2);

}

}

int main()

{

LList<int> a(5);

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

{

int temp ; cin >> temp;

a.append(temp);

}

myReverse(a);

a.print();

}

文本

描述已自动生成

#### 2. Assume there are two ascending ordered singly linked lists L1 and L2, please merge L1 and L2 into a new descending singly linked list L3. There will be no duplicate items in L3.

#include <iostream>

#include "llist.cpp"

using namespace std;

template <typename E>

void myReverse(LList<E> &t)

{

int len = t.length();

int right = len-1 ; int left = 0;

for(; left < len/2 ; left++ , right--)

{

E t1 , t2;

t.moveToPos(left) ; t1 = t.getValue();

t.moveToPos(right) ; t2 = t.getValue();

t.remove() ; t.insert(t1);

t.moveToPos(left) ; t.remove() ; t.insert(t2);

}

}

template <typename E>

void myUnion(LList<E> &a,LList<E> &b,LList<E> &c)

{

int box[100] {0};

a.moveToStart();

for(; a.currPos() < a.length()-1 ; a.next())

{

int it = a.getValue();

box[it]++;

}

int it = a.getValue(); box[it]++;

b.moveToStart();

for(; b.currPos() < b.length()-1 ; b.next())

{

int it = b.getValue();

box[it]++;

}

it = b.getValue(); box[it]++;

for(int i = 0 ; i < 100 ; i++)

if(box[i])

c.append(i);

}

int main()

{

LList<int> a(5);

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

{

int temp ; cin >> temp;

a.append(temp);

}

LList<int> b(7);

for(int i = 0 ; i < 7 ; i++)

{

int temp ; cin >> temp;

b.append(temp);

}

LList<int> ans;

myUnion(a,b,ans);

myReverse(ans);

ans.print();

}

文本

描述已自动生成

#### 3. You are given a linked list ,L,and another P, P contain integers sorted in ascedding order. The function PrintLots(L,P) will print the elements in L that are in positions specified by P. For instance, if P =1,3,4,6 ,the first, third, fourth, and sixth elements in L are printed. Write the function PrintLots(L,P).

#include <iostream>

#include "llist.cpp"

using namespace std;

template <typename E>

void PrintLots(LList<E> &a , LList<E> &b)

{

b.moveToStart();

for(; b.currPos() < b.length()-1 ; b.next())

{

int it = b.getValue();

a.moveToPos(it);

int ans = a.getValue();

cout<<ans<<" ";

}

int it = b.getValue();

a.moveToPos(it);

int ans = a.getValue();

cout<<ans;

}

int main()

{

LList<int> a(10);

for(int i = 0 ; i < 10 ; i++)

{

int temp ; cin >> temp;

a.append(temp);

}

LList<int> b(4);

for(int i = 0 ; i < 4 ; i++)

{

int temp ; cin >> temp;

b.append(temp);

}

PrintLots(a,b);

}

文本

描述已自动生成

#### 4. In the linked list implementation presented in Section 4.1.2, the current position is implemented using a pointer to the element ahead of the logical current node. The more “natural” approach might seem to be to have curr point directly to the node containing the current element. However, if this was done, then the pointer of the node preceding the current one cannot be updated properly because there is no access to this node from curr. An alternative is to add a new node after the current element, copy the value of the current element to this new node, and then insert the new value into the old current node.

#### (a) What happens if curr is at the end of the list already? Is there still a way to make this work? Is the resulting code simpler or more complex than the implementation of Section 4.1.2?

将curr指向currelement只需

template <typename E>

void LList<E>::moveToPos(int pos){

Assert(pos >= 0 && pos <= cnt , "Position out of range");

curr = head;

for(int i = 0 ; i ***<=*** pos ; i++) curr = curr->next;

} 将<改为<=

template <typename E>

const E& LList<E>::getValue(){

Assert(curr != nullptr , "No value");

return curr->element;

}

template <typename E>

void LList<E>::insert(const E& it){

E temp = curr->element;

curr->next = new Link<E>(temp,curr->next);

if(tail == curr) tail = curr->next;

curr->element = it;

cnt++;

}和原代码复杂程度一样，时间复杂度都是O（1）

#### (b) Will deletion always work in constant time if curr points directly to the current node? In particular, can you make several deletions in a row?

将curr指向currelement只需

template <typename E>

void LList<E>::moveToPos(int pos){

Assert(pos >= 0 && pos <= cnt , "Position out of range");

curr = head;

for(int i = 0 ; i ***<=*** pos ; i++) curr = curr->next;

} 将<改为<=

template <typename E>

const E& LList<E>::getValue(){

Assert(curr != nullptr , "No value");

return curr->element;

}

Remove函数

E LList<E>::remove(){

E it = curr->element;

Link<E>\* ltemp = curr;

curr = head;

while(curr->next != ltemp) //将curr指向currelement前一个

curr = curr->next;

if(ltemp == tail)

tail = curr; //若curr==tail tail前移

curr->next = curr->next->next;

delete ltemp;

cnt--;

return it;

}

以此实现remove

Main

int main()

{

LList<int> b(4);

for(int i = 0 ; i < 4 ; i++)

{

int temp ; cin >> temp;

b.append(temp);

}

b.moveToPos(3);

b.remove();

b.print();

cout<<endl;

b.remove();

b.print();

}

文本

描述已自动生成

且可以多次remove，其中每次移除的元素是上一次移除元素的前一位。

#### 5. A circular linked list is one in which the next ﬁeld for the last link node of the list points to the ﬁrst link node of the list. This can be useful when you wish to have a relative positioning for elements, but no concept of an absolute ﬁrst or last position.

#### (a) Modify the code of Figure 4.8 to implement circular singly linked lists.

修改init函数为

void init()

{

curr = tail = head = new Link<E>;

tail->next = head;

head->next = head;

cnt = 0;

}

修改removeall函数为

void removeall()

{

while(head->next != head){

curr = head;

head = head->next;

delete curr;

}

}

修改append函数为

template <typename E>

void LList<E>::append(const E& it){ //单向循环链表版本

tail = tail->next = new Link<E>(it , head->next);

cnt++;

}

修改prev函数为

template <typename E>

void LList<E>::prev(){ //单向循环链表

Link<E>\* temp = head;

while (temp->next != curr) temp = temp->next;

curr = temp;

}

修改next函数为

void LList<E>::next(){ //单向循环链表

curr = curr->next;

}

Main函数

int main()

{

LList<int> a(5);

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

{

int temp ; cin >> temp;

a.append(temp);

}

a.moveToStart();

for(int i = 0 ; i < 10 ; i++)

{

int it = a.getValue();

cout<<it<<" ";

a.next();

}

}

文本

描述已自动生成

#### (b) Modify the code of Figure 4.14 to implement circular doubly linked lists.

修改init

void init() //双向循环链表版本

{

curr = head = new Link<E>;

tail = new Link<E>;

head->next = tail ; head->prev = tail;

tail->prev = head ; tail->next = head;

cnt = 0;

}

修改insert append remove next prev getvalue

template <typename E>

void LList<E>::insert(const E& it){

curr->next = curr->next->prev = new Link<E>(it , curr , curr->next);

cnt++;

}

template <typename E>

void LList<E>::append(const E& it){

tail->prev = tail->prev->next = new Link<E>(it,tail->prev,tail);

cnt++;

}

template <typename E>

E LList<E>::remove(){

if(curr->next == tail)

return NULL;

E it = curr->next->element;

Link<E>\* ltemp = curr->next;

curr->next->next->prev = curr;

curr->next = curr->next->next;

delete ltemp;

cnt--;

return it;

}

template <typename E>

void LList<E>::prev(){ //双向循环链表版本

if(curr != head)

curr = curr->prev;

else

curr = tail->prev->prev;

}

template <typename E>

void LList<E>::next(){ //双向循环链表版本

if(curr->next != tail)

curr = curr->next;

else

curr = head;

}

template <typename E>

const E& LList<E>::getValue(){ //双向循环链表版本

if(curr->next != tail)

return curr->next->element;

else

curr = head;

return curr->next->element;

}

Main函数

int main()

{

LList<int> a(5);

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

{

int temp ; cin >> temp;

a.append(temp);

}

a.moveToEnd();

cout<<a.getValue()<<" ";

a.next();

cout<<a.getValue()<<" ";

a.moveToStart();

cout<<a.getValue()<<" ";

a.prev();

cout<<a.getValue()<<" "<<endl;

a.moveToStart();

for(int i = 0 ; i < 10 ; i++)

{

cout<<a.getValue()<< " ";

a.next();

}

}

文本

描述已自动生成