二：链

1. mian.c

#include "SList.h"

#include "DList.h"

SListNode\* partition(SListNode\* pHead, int x) {

// write code here

SListNode \*lt = NULL; // < x 的第一个结点

SListNode \*lt\_tail = NULL; // < x 的最后一个结点

SListNode \*ge = NULL; // >= x 的第一个结点

SListNode \*ge\_tail = NULL; // >= x 的最后一个结点

SListNode \*cur = pHead;

while (cur != NULL) {

SListNode \*next = cur->next;

cur->next = NULL;

if (cur->value < x) {

if (lt\_tail == NULL) {

lt = lt\_tail = cur;

}

else {

lt\_tail->next = cur;

lt\_tail = cur;

}

}

else {

if (ge\_tail == NULL) {

ge = ge\_tail = cur;

}

else {

ge\_tail->next = cur;

ge\_tail = cur;

}

}

cur = next;

}

if (lt\_tail == NULL) {

return ge;

}

else {

lt\_tail->next = ge;

return lt;

}

}

void TestPartition() {

SList list;

SListInit(&list);

int values[] = { 3, 7, 4, 9, 2, 5, 8, 4 };

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

SListPushBack(&list, values[i]);

}

SListPrint(&list);

SListNode \* r = partition(list.first, 6);

list.first = r;

SListPrint(&list);

}

int main() {

TestPartition();

return 0;

}

2.Slist.h文件

#pragma once

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

typedef int SLDataType;

typedef struct SLNode {

SLDataType value;

struct SLNode \*next;

} SListNode;

typedef struct SList {

SListNode \*first; // SLNode \*head;

} SList;

// 初始化

void SListInit(SList \*list) {

assert(list != NULL);

list->first = NULL;

}

// 销毁

void SListDestroy(SList \*list) {

assert(list != NULL);

SListNode \*cur = list->first;

SListNode \*next;

while (cur != NULL) {

next = cur->next;

free(cur);

cur = next;

}

list->first = NULL;

}

// 申请新结点

SListNode \* SListBuyNode(SLDataType value)

{

SListNode \*node = (SListNode \*)malloc(sizeof(SListNode));

assert(node);

node->value = value;

node->next = NULL;

return node;

}

// 增

// 头插

void SListPushFront(SList \*list, SLDataType value)

{

assert(list != NULL);

SListNode \*node = (SListNode \*)malloc(sizeof(SListNode));

assert(node);

node->value = value;

node->next = list->first;

list->first = node;

}

// 尾插

void SListPushBack(SList \*list, SLDataType value)

{

assert(list != NULL);

if (list->first == NULL) {

SListPushFront(list, value);

return;

}

// 链表中有结点的情况

// 找链表中的最后一个结点

SListNode \*cur;

for (cur = list->first; cur->next != NULL; cur = cur->next) {

}

// cur 就是最后一个结点

SListNode \*node = SListBuyNode(value);

cur->next = node;

}

// 删

// 头删

void SListPopFront(SList \*list)

{

assert(list != NULL); // 保证链表是有的

assert(list->first != NULL); // 保证链表不为空

SListNode \*old\_first = list->first;

list->first = list->first->next;

free(old\_first);

}

void SListPopBack(SList \*list)

{

assert(list != NULL); // 保证链表是有的

assert(list->first != NULL); // 保证链表不为空

// 如果只有一个结点

if (list->first->next == NULL) {

SListPopFront(list);

return;

}

SListNode \*cur = list->first;

while (cur->next->next != NULL) {

cur = cur->next;

}

free(cur->next);

cur->next = NULL;

}

// pos 一定是链表中的一个有效结点

void SListInsertAfter(SListNode \*pos, SLDataType value) {

// 给 value 申请结点

SListNode \*node = SListBuyNode(value);

node->next = pos->next;

pos->next = node;

}

void SListEraseAfter(SListNode \*pos) {

SListNode \*next = pos->next;

pos->next = next->next;

free(next);

}

void SListInsertBefore(SList \*list, SListNode \*pos, SLDataType value)

{

SListNode \*cur = list->first;

while (cur->next != pos) {

cur = cur->next;

}

// cur 指向 pos 的前一个结点

SListNode \*node = SListBuyNode(value);

cur->next = node;

node->next = pos;

}

// 改

void SListUpdate(SListNode \*node, SLDataType value) {

node->value = value;

}

// 查

// 去找到链表中遇到的第一个 value，如果没找到，返回 NULL

SListNode \* SListFind(const SList \*list, SLDataType value) {

for (SListNode \*cur = list->first; cur != NULL; cur = cur->next) {

if (cur->value == value) {

return cur;

}

}

return NULL;

}

// 打印

void SListPrint(const SList \*list) {

for (SListNode \*cur = list->first; cur != NULL; cur = cur->next) {

printf("%d --> ", cur->value);

}

printf("NULL\n");

}

void TestSList1() {

SList list;

SListInit(&list);

assert(list.first == NULL);

SListPushBack(&list, 11);

SListPushBack(&list, 12);

SListPushBack(&list, 13);

SListNode \*n12 = SListFind(&list, 12);

assert(n12 != NULL);

SListPrint(&list);

// 11 12 13

SListInsertAfter(n12, 103);

SListPrint(&list);

// 11 12 103 13

SListEraseAfter(n12);

SListPrint(&list);

// 11 12 13

SListInsertBefore(&list, n12, 101);

SListPrint(&list);

// 11 101 12 13

printf("大成功\n");

}

3.Dlist.h文件

#pragma once

#include <stdio.h>

#include <stdlib.h>

typedef int DLDataType;

// 定义链表结点结构

typedef struct DListNode {

DLDataType value; // 头结点的 value 没有意义

struct DListNode \*prev; // 指向前一个结点

struct DListNode \*next; // 指向后一个结点

} DListNode;

// 定义双向链表结构

typedef struct DList {

DListNode \*head; // 表示链表的头结点

} DList;

DListNode \* DListBuyNode(DLDataType value) {

DListNode \* node = (DListNode \*)malloc(sizeof(DListNode));

node->value = value;

node->next = node->prev = NULL;

return node;

}

// 初始化

void DListInit(DList \* dlist) {

dlist->head = DListBuyNode(0); // 0 其实没有意思

// 这里是头结点

dlist->head->next = dlist->head;

dlist->head->prev = dlist->head;

}

// 销毁

// 1. 清空链表

void DListClear(DList \*dlist) {

DListNode \*cur, \*next;

cur = dlist->head->next;

while (cur != dlist->head) {

next = cur->next;

free(cur);

cur = next;

}

dlist->head->next = dlist->head->prev = dlist->head;

}

// 2. 彻底销毁链表

void DListDestroy(DList \*dlist) {

DListClear(dlist);

free(dlist->head);

dlist->head = NULL;

}

// 增

void DListInsert(DListNode \*pos, DLDataType value) {

DListNode \*node = DListBuyNode(value);

node->prev = pos->prev;

node->next = pos;

node->prev->next = node;

pos->prev = node;

}

void DListErase(DListNode \*pos) {

pos->prev->next = pos->next;

pos->next->prev = pos->prev;

free(pos);

}

// 头插

void DListPushFront(DList \* dlist, DLDataType value) {

#if 0

DListNode \*node = DListBuyNode(value);

node->prev = dlist->head;

node->next = dlist->head->next;

dlist->head->next->prev = node;

dlist->head->next = node;

#endif

DListInsert(dlist->head->next, value);

}

void DListPushBack(DList \*dlist, DLDataType value) {

#if 0

DListNode \*node = DListBuyNode(value);

node->prev = dlist->head->prev;

node->next = dlist->head;

node->prev->next = node;

node->next->prev = node;

#endif

DListInsert(dlist->head, value);

}

void DListPopFront(DList \*dlist)

{

assert(dlist->head->next != dlist->head);

#if 0

DListNode \*cur = dlist->head->next;

dlist->head->next = cur->next;

cur->next->prev = dlist->head;

free(cur);

#endif

DListErase(dlist->head->next);

}

void DListPopBack(DList \*dlist)

{

assert(dlist->head->next != dlist->head); // 确保链表不为空

#if 0

DListNode \*cur = dlist->head->prev;

cur->prev->next = dlist->head; //dlist->head->prev->prev->next;

cur->next->prev = cur->prev; //dlist->head->prev;

free(cur);

#endif

DListErase(dlist->head->prev);

}

DListNode \* DListFind(const DList \*dlist, DLDataType value) {

DListNode \*cur;

for (cur = dlist->head->next; cur != dlist->head; cur = cur->next) {

if (cur->value == value) {

return cur;

}

}

return NULL;

}

void DListPrint(const DList \*dlist) {

for (DListNode \*cur = dlist->head->next;

cur != dlist->head; cur = cur->next) {

printf("%d --> ", cur->value);

}

printf("\n");

}

void TestDList1() {

DList list;

DListInit(&list);

DListPrint(&list);

DListPushBack(&list, 1);

DListPushBack(&list, 2);

DListPushBack(&list, 3);

DListPrint(&list);

// 1 2 3

DListPushFront(&list, 11);

DListPushFront(&list, 12);

DListPushFront(&list, 13);

DListPrint(&list);

// 13 12 11 1 2 3

}

1. OJ.h

// 删除所有

struct ListNode\* removeElements(struct ListNode\* head, int val) {

if (head == NULL) {

return NULL;

}

struct ListNode \*prev = head;

struct ListNode \*cur = head->next;

while (cur != NULL) {

if (cur->val == val) {

prev->next = cur->next;

free(cur);

cur = prev->next;

}

else {

prev = cur;

cur = cur->next;

}

}

if (head->val == val) {

struct ListNode \*newHead = head->next;

free(head);

return newHead;

}

else {

return head;

}

}

// 反转单链表 第一种方式

struct ListNode\* reverseList(struct ListNode\* head) {

struct ListNode \*ret = NULL;

while (head != NULL) {

// 头删

struct ListNode \*node = head;

head = head->next;

// 头插 node -> ret

node->next = ret;

ret = node;

}

return ret;

}

// 反转单链表 第二种方式

struct ListNode\* reverseList(struct ListNode\* head) {

if (head == NULL) {

return head;

}

struct ListNode \*prev, \*cur, \*next;

prev = NULL;

cur = head;

next = head->next;

while (cur != NULL) {

cur->next = prev;

prev = cur;

cur = next;

if (next != NULL) {

next = next->next;

}

}

return prev;

}

// 找到中间结点

struct ListNode\* middleNode(struct ListNode\* head) {

if (head == NULL) {

return NULL;

}

struct ListNode \*fast = head;

struct ListNode \*slow = head;

while (1) {

fast = fast->next;

if (fast == NULL) {

break;

}

slow = slow->next;

fast = fast->next;

if (fast == NULL) {

break;

}

}

return slow;

}

struct ListNode\* mergeTwoLists(struct ListNode\* l1, struct ListNode\* l2) {

if (l1 == NULL) {

return l2;

}

else if (l2 == NULL) {

return l1;

}

struct ListNode \*cur1 = l1;

struct ListNode \*cur2 = l2;

struct ListNode \*result = NULL;

struct ListNode \*result\_tail = NULL; // result 链表的最后一个结点

while (cur1 != NULL && cur != NULL) {

if (cur1->val <= cur2->val) {

// 把 cur1 的结点尾插到 result

if (result\_tail != NULL) {

result\_tail->next = cur1;

result\_tail = cur1;

}

else {

result = cur1;

result\_tail = cur1;

}

cur1 = cur1->next;

}

else {

// 把 cur2 的结点尾插到 result

if (result\_tail != NULL) {

result\_tail->next = cur2;

result\_tail = cur2;

}

else {

result = cur2;

result\_tail = cur2;

}

cur2 = cur2->next;

}

}

// 一个链表已经处理完了

if (cur1 != NULL) {

result\_tail->next = cur1;

}

else {

result\_tail->next = cur2;

}

}

ListNode\* partition(ListNode\* pHead, int x) {

ListNode \*lt = NULL; // < x 的第一个结点

ListNode \*lt\_tail = NULL; // < x 的最后一个结点

ListNode \*ge = NULL; // >= x 的第一个结点

ListNode \*ge\_tail = NULL; // >= x 的最后一个结点

ListNode \*cur = pHead;

while (cur != NULL) {

ListNode \*next = cur->next;

//cur->next = NULL;

if (cur->val < x) {

if (lt\_tail == NULL) {

// 比 x 小的链表现在是空的

lt = lt\_tail = cur;

}

else {

lt\_tail->next = cur;

lt\_tail = cur;

}

}

else {

if (ge\_tail == NULL) {

ge = ge\_tail = cur;

}

else {

ge\_tail->next = cur;

ge\_tail = cur;

}

}

cur = next;

}

if (ge\_tail != NULL) {

ge\_tail->next = NULL;

}

if (lt\_tail == NULL) {

return ge;

}

else {

lt\_tail->next = ge;

return lt;

}

}

ListNode\* deleteDuplication(ListNode\* pHead)

{

if (pHead == NULL) {

return NULL;

}

ListNode \*prev = NULL; // 用于删除的结点，是 p1 的前一个结点

ListNode \*p1 = pHead;

ListNode \*p2 = pHead->next;

ListNode \*result = pHead; // 用于返回的第一个结点

while (p2 != NULL) {

if (p1->val != p2->val) {

prev = p1;

p1 = p2;

p2 = p2->next;

}

else {

while (p2 != NULL && p2->val == p1->val) {

p2 = p2->next;

}

// 删除

ListNode \*next;

for (ListNode \*node = p1; node != p2; node = next) {

next = node->next;

free(node);

}

// 重新拼接链表

if (prev != NULL) {

prev->next = p2;

}

else {

// 1 --> 1 --> 1 --> 2 --> NULL

// 的情况

result = p2;

}

p1 = p2;

if (p2 != NULL) {

p2 = p2->next;

}

}

}

return result;

}

// 回文

ListNode\* middleNode(ListNode\* head) {

if (head == NULL) {

return NULL;

}

ListNode \*fast = head;

ListNode \*slow = head;

while (1) {

fast = fast->next;

if (fast == NULL) {

break;

}

slow = slow->next;

fast = fast->next;

if (fast == NULL) {

break;

}

}

return slow;

}

ListNode\* reverseList(ListNode\* head) {

if (head == NULL) {

return head;

}

ListNode \*prev, \*cur, \*next;

prev = NULL;

cur = head;

next = head->next;

while (cur != NULL) {

cur->next = prev;

prev = cur;

cur = next;

if (next != NULL) {

next = next->next;

}

}

return prev;

}

bool chkPalindrome(ListNode\* A) {

ListNode \* middle = middleNode(A);

// middle 是中间的第一个

// 所以逆置middle 的next

ListNode \* r = reverseList(middle->next);

ListNode \*n1 = A, \*n2 = r;

while (n1 != NULL && n2 != NULL) {

if (n1->val != n2->val) {

return false;

}

n1 = n1->next;

n2 = n2->next;

}

return true;

}

int getLength(struct ListNode \*head) {

int n = 0;

struct ListNode \*cur = head;

while (cur != NULL) {

cur = cur->next;

n++;

}

return n;

}

struct ListNode \*getIntersectionNode(struct ListNode \*headA, struct ListNode \*headB) {

int lenA = getLength(headA);

int lenB = getLength(headB);

struct ListNode \*longHead;

struct ListNode \*shortHead;

int diff;

if (lenA >= lenB) {

longHead = headA;

shortHead = headB;

diff = lenA - lenB;

}

else {

longHead = headB;

shortHead = headA;

diff = lenB - lenA;

}

for (int i = 0; i < diff; i++) {

longHead = longHead->next;

}

while (longHead != shortHead) {

longHead = longHead->next;

shortHead = shortHead->next;

}

return longHead;

}

struct ListNode \*detectCycle(struct ListNode \*head) {

if (head == NULL) {

return NULL;

}

struct ListNode \*fast = head;

struct ListNode \*slow = head;

while (1) {

fast = fast->next;

if (fast == NULL) {

return NULL;

}

fast = fast->next;

if (fast == NULL) {

return NULL;

}

slow = slow->next;

if (fast == slow) {

break;

}

}

struct ListNode \*n1 = head;

struct ListNode \*n2 = slow;

while (n1 != n2) {

n1 = n1->next;

n2 = n2->next;

}

return n1;

}

typedef struct RandomListNode RandomListNode;

// 复杂链表复制

struct RandomListNode \*copyRandomList(struct RandomListNode \*head) {

if (head == NULL) {

return NULL;

}

struct RandomListNode \*cur = head;

while (cur != NULL) {

RandomListNode \*node = (RandomListNode \*)malloc(sizeof(RandomListNode));

node->label = cur->label;

node->random = NULL;

RandomListNode \*next = cur->next;

cur->next = node;

node->next = next;

cur = next;

}

// 复制 random

cur = head;

while (cur != NULL) {

RandomListNode \*node = cur->next;

if (cur->random != NULL) {

node->random = cur->random->next;

}

cur = node->next;

}

cur = head;

RandomListNode \*ret = head->next;

while (cur != NULL) {

RandomListNode \*node = cur->next;

cur->next = node->next;

if (node->next != NULL) {

node->next = cur->next->next;

}

cur = cur->next;

}

return ret;

}

1. partict.h文件

#pragma once

typedef struct ListNode {

int value;

struct ListNode \*next;

} ListNode;

// 逆置

// 头删(head) + 头插(ret)

ListNode \* reverseList\_1(ListNode \*head) {

ListNode \*ret = NULL;

ListNode \*cur = head;

while (cur != NULL) {

ListNode \*next = cur->next;

ListNode \*node = cur; // node 结点从 head 上头删

// node 结点准备头插到 ret

node->next = ret; // node 是 ret 新的第一个结点

// node->next 应该是原来的第一个结点

// ret 中记录着 原来的第一个结点

ret = node; // 更新 ret 的第一个结点地址

cur = next;

}

return ret;

}

// 逆置

// 三个指针法

ListNode \* reverseList\_2(ListNode \*head) {

if (head == NULL) {

return NULL;

}

else {

ListNode \*p1 = NULL;

ListNode \*p2 = head;

ListNode \*p3 = head->next;

while (p2 != NULL) {

p2->next = p1;

p1 = p2;

p2 = p3;

if (p3 != NULL) {

p3 = p3->next;

}

}

return p1;

}

}

ListNode \* PushBack(ListNode \*head, ListNode \*tail, ListNode \*node) {

if (head = NULL) {

tail = node;

return node;

}

else {

tail->next = node;

tail = node;

return head;

}

}

// 合并两个有序链表

ListNode \* merge(ListNode \*headA, ListNode \*headB) {

if (headA == NULL) {

return headB;

}

else if (headB == NULL) {

return headA;

}

ListNode \*curA = headA;

ListNode \*curB = headB;

ListNode \*ret = NULL;

ListNode \*tail = NULL;

while (curA != NULL & curB != NULL) {

if (curA->value <= curB->value) {

ListNode \*next = curA->next;

if (ret == NULL) {

tail = curA;

ret = curA;

}

else {

tail->next = curA;

tail = curA;

}

curA = next;

}

else {

ListNode \*next = curB->next;

if (ret == NULL) {

tail = curB;

ret = curB;

}

else {

tail->next = curB;

tail = curB;

}

curB = next;

curB = next;

}

}

if (curA == NULL) {

tail->next = curB;

}

else {

tail->next = curA;

}

return ret;

}