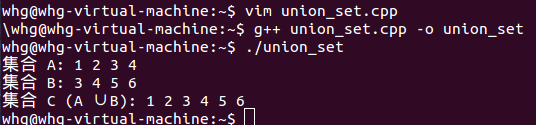
1.

代码如下:

#include <iostream>

#include <set>

#include <vector>

int main() {

// 定义集合 A 和 B

std::vector<int> A;

A.push\_back(1);

A.push\_back(2);

A.push\_back(3);

A.push\_back(4);

std::vector<int> B;

B.push\_back(3);

B.push\_back(4);

B.push\_back(5);

B.push\_back(6);

// 使用 set 存储并集

std::set<int> unionSet;

// 将集合 A 的元素插入并集

for (size\_t i = 0; i < A.size(); ++i) {

unionSet.insert(A[i]);

}

// 将集合 B 的元素插入并集

for (size\_t i = 0; i < B.size(); ++i) {

unionSet.insert(B[i]);

}

// 将结果输出

std::cout << "集合 A: ";

for (size\_t i = 0; i < A.size(); ++i) {

std::cout << A[i] << " ";

}

std::cout << std::endl;

std::cout << "集合 B: ";

for (size\_t i = 0; i < B.size(); ++i) {

std::cout << B[i] << " ";

}

std::cout << std::endl;

std::cout << "集合 C (A ∪ B): ";

// 使用常规 for 循环输出 set 中的元素

for (std::set<int>::iterator it = unionSet.begin(); it != unionSet.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

return 0;

}

2.#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 链表结点结构

struct LinkNode {

ElemType data; // 数据域

LinkNode\* next; // 指针域

};

// 插入函数

bool InsertList(LinkNode\* L, ElemType a, ElemType x) {

// 找到数据域为 a 的结点

LinkNode\* current = L->next; // 从头结点的下一个结点开始

while (current != nullptr && current->data != a) {

current = current->next; // 遍历链表

}

// 如果没有找到数据域为 a 的结点，返回 false

if (current == nullptr) {

return false;

}

// 创建新结点

LinkNode\* newNode = new LinkNode;

newNode->data = x; // 设置新结点的数据域

newNode->next = current->next; // 新结点的指针域指向当前结点的下一个结点

current->next = newNode; // 当前结点的指针域指向新结点

return true; // 插入成功

}

// 测试代码

int main() {

// 创建一个带头结点的链表

LinkNode\* head = new LinkNode; // 头结点

head->next = nullptr; // 初始化链表为空

// 添加一些结点

LinkNode\* node1 = new LinkNode{1, nullptr};

LinkNode\* node2 = new LinkNode{2, nullptr};

head->next = node1; // 头结点指向第一个结点

node1->next = node2; // 第一个结点指向第二个结点

// 插入新结点

if (InsertList(head, 1, 3)) {

std::cout << "插入成功" << std::endl;

} else {

std::cout << "插入失败" << std::endl;

}

// 输出链表

LinkNode\* current = head->next;

std::cout << "链表内容: ";

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

// 释放内存

current = head;

while (current != nullptr) {

LinkNode\* temp = current;

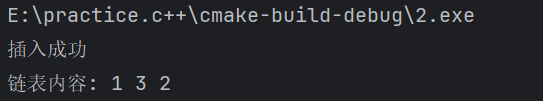
current = current->next;

delete temp;

}

return 0;

}



#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 链表结点结构

struct LinkNode {

ElemType data; // 数据域

LinkNode\* next; // 指针域

};

// 删除函数

void DeleteNodes(LinkNode\* head, int i, int n) {

LinkNode\* current = head; // 从头结点开始

int count = 0;

// 找到第 i 个结点

while (current != nullptr && count < i) {

current = current->next;

count++;

}

// 如果当前结点为空，说明 i 超出了链表的范围

if (current == nullptr) {

std::cout << "i 超出链表范围" << std::endl;

return;

}

// 删除 n 个结点

LinkNode\* toDelete = current->next; // 从第 i + 1 个结点开始删除

for (int j = 0; j < n && toDelete != nullptr; j++) {

LinkNode\* temp = toDelete; // 保存要删除的结点

toDelete = toDelete->next; // 移动到下一个结点

delete temp; // 删除结点

}

// 连接剩余的链表

current->next = toDelete; // 将第 i 个结点的 next 指向第 i + n + 1 个结点

}

// 测试代码

int main() {

// 创建一个带头结点的链表

LinkNode\* head = new LinkNode; // 头结点

head->next = nullptr; // 初始化链表为空

// 添加一些结点

for (int i = 1; i <= 10; i++) {

LinkNode\* newNode = new LinkNode{i, nullptr};

newNode->next = head->next; // 新结点插入到链表头部

head->next = newNode;

}

// 删除从第 3 个结点起的 4 个结点

DeleteNodes(head, 3, 4);

// 输出链表

LinkNode\* current = head->next; // 从第一个有效结点开始

std::cout << "链表内容: ";

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

// 释放内存

current = head;

while (current != nullptr) {

LinkNode\* temp = current;

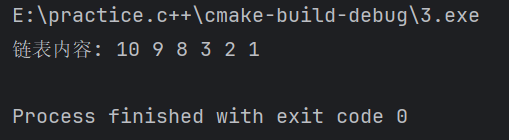
current = current->next;

delete temp;

}

return 0;

}



4.#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 链表结点结构

struct LinkNode {

ElemType data; // 数据域

LinkNode\* next; // 指针域

};

// 计算数据域为 x 的结点数量

int CountList(LinkNode\* L, ElemType x) {

int count = 0; // 计数器

LinkNode\* current = L->next; // 从头结点的下一个结点开始

// 遍历链表

while (current != nullptr) {

if (current->data == x) {

count++; // 如果数据域等于 x，计数器加一

}

current = current->next; // 移动到下一个结点

}

return count; // 返回计数结果

}

// 测试代码

int main() {

// 创建一个带头结点的链表

LinkNode\* head = new LinkNode; // 头结点

head->next = nullptr; // 初始化链表为空

// 添加一些结点

for (int i = 1; i <= 10; i++) {

LinkNode\* newNode = new LinkNode{i, nullptr};

newNode->next = head->next; // 新结点插入到链表头部

head->next = newNode;

}

// 在链表中插入一些重复的值

LinkNode\* newNode1 = new LinkNode{5, nullptr};

newNode1->next = head->next;

head->next = newNode1; // 插入一个值为 5 的结点

LinkNode\* newNode2 = new LinkNode{5, nullptr};

newNode2->next = head->next;

head->next = newNode2; // 再插入一个值为 5 的结点

// 计算值为 5 的结点数量

int x = 5;

int count = CountList(head, x);

std::cout << "值为 " << x << " 的结点数量: " << count << std::endl;

// 释放内存

LinkNode\* current = head;

while (current != nullptr) {

LinkNode\* temp = current;

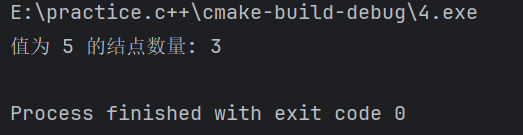
current = current->next;

delete temp;

}

return 0;

}



5.

#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 链表结点结构

struct LinkNode {

ElemType data; // 数据域

LinkNode\* next; // 指针域

};

// 插入函数

void InsertInOrder(LinkNode\* head, ElemType x) {

LinkNode\* newNode = new LinkNode; // 创建新结点

newNode->data = x; // 设置新结点的数据域

newNode->next = nullptr;

LinkNode\* current = head; // 从头结点开始

LinkNode\* previous = nullptr;

// 找到插入位置

while (current != nullptr && current->data < x) {

previous = current;

current = current->next;

}

// 插入新结点

if (previous == nullptr) {

// 插入到头部

newNode->next = head->next; // 新结点指向原链表的第一个结点

head->next = newNode; // 头结点指向新结点

} else {

// 插入到中间或尾部

newNode->next = current; // 新结点指向当前结点

previous->next = newNode; // 前一个结点指向新结点

}

}

// 测试代码

int main() {

// 创建一个带头结点的链表

LinkNode\* head = new LinkNode; // 头结点

head->next = nullptr; // 初始化链表为空

// 添加一些结点

for (int i = 1; i <= 5; i++) {

LinkNode\* newNode = new LinkNode{i, nullptr};

newNode->next = head->next; // 新结点插入到链表头部

head->next = newNode;

}

// 按顺序插入新结点

int x = 3; // 要插入的值

InsertInOrder(head, x);

// 输出链表

LinkNode\* current = head->next; // 从第一个有效结点开始

std::cout << "链表内容: ";

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

// 释放内存

current = head;

while (current != nullptr) {

LinkNode\* temp = current;

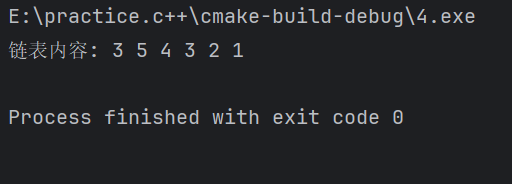
current = current->next;

delete temp;

}

return 0;

}



6.

#include <iostream>

#include <limits>

typedef int ElemType; // 定义元素类型为整型

// 链表结点结构

struct LinkNode {

ElemType data; // 数据域

LinkNode\* next; // 指针域

};

// 删除最大值结点的函数

void DeleteMaxNode(LinkNode\* head) {

if (head == nullptr || head->next == nullptr) {

std::cout << "链表为空或只有头结点，无法删除最大结点。" << std::endl;

return;

}

LinkNode\* current = head->next; // 从第一个有效结点开始

LinkNode\* maxNode = current; // 假设第一个结点是最大值结点

LinkNode\* previous = head; // 记录前一个结点

// 遍历链表，找到最大值结点

while (current != nullptr) {

if (current->data > maxNode->data) {

maxNode = current; // 更新最大值结点

previous = head; // 重置前一个结点

LinkNode\* temp = head->next; // 从头结点开始查找前一个结点

while (temp != nullptr && temp != maxNode) {

previous = temp;

temp = temp->next;

}

}

current = current->next; // 移动到下一个结点

}

// 删除最大值结点

if (maxNode != nullptr) {

previous->next = maxNode->next; // 前一个结点指向最大值结点的下一个结点

delete maxNode; // 释放内存

}

}

// 测试代码

int main() {

// 创建一个带头结点的链表

LinkNode\* head = new LinkNode; // 头结点

head->next = nullptr; // 初始化链表为空

// 添加一些结点

for (int i = 1; i <= 5; i++) {

LinkNode\* newNode = new LinkNode{i, nullptr};

newNode->next = head->next; // 新结点插入到链表头部

head->next = newNode;

}

// 输出链表

std::cout << "链表内容: ";

LinkNode\* current = head->next; // 从第一个有效结点开始

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

// 删除最大值结点

DeleteMaxNode(head);

// 输出链表

std::cout << "删除最大值结点后的链表内容: ";

current = head->next; // 从第一个有效结点开始

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

// 释放内存

current = head;

while (current != nullptr) {

LinkNode\* temp = current;

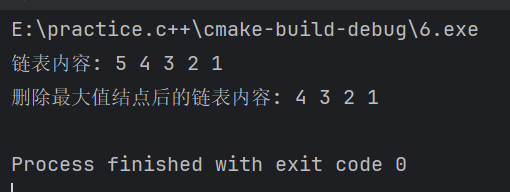
current = current->next;

delete temp;

}

return 0;

}



7.

#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 双链表结点结构

struct DLinkNode {

ElemType data; // 数据域

DLinkNode\* next; // 指向下一个结点的指针

DLinkNode\* prev; // 指向前一个结点的指针

};

// 创建一个双链表并返回头结点

DLinkNode\* CreateDList(int n) {

DLinkNode\* head = new DLinkNode; // 头结点

head->next = nullptr;

head->prev = nullptr;

DLinkNode\* tail = head; // 记录链表的尾部

// 添加结点

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

DLinkNode\* newNode = new DLinkNode{i, nullptr, nullptr};

tail->next = newNode; // 将新结点链接到链表

newNode->prev = tail; // 新结点的前指针指向当前尾

tail = newNode; // 更新尾结点

}

return head; // 返回头结点

}

// 变换双链表的函数

void TransformDList(DLinkNode\* head) {

if (head == nullptr || head->next == nullptr) {

std::cout << "链表为空或只有头结点，无法变换。" << std::endl;

return;

}

DLinkNode\* front = head->next; // 从第一个有效结点开始

DLinkNode\* back = head; // 从头结点开始

// 找到链表的尾部

while (back->next != nullptr) {

back = back->next;

}

// 进行变换

while (front != back && front->prev != back) {

// 交换数据

std::swap(front->data, back->data);

front = front->next; // 移动到下一个结点

back = back->prev; // 移动到前一个结点

}

}

// 打印链表

void PrintDList(DLinkNode\* head) {

DLinkNode\* current = head->next; // 从第一个有效结点开始

while (current != nullptr) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

}

// 释放链表内存

void FreeDList(DLinkNode\* head) {

DLinkNode\* current = head;

while (current != nullptr) {

DLinkNode\* temp = current;

current = current->next;

delete temp;

}

}

// 测试代码

int main() {

// 创建一个带头结点的双链表

int n = 5;

DLinkNode\* head = CreateDList(n);

// 输出链表

std::cout << "链表内容: ";

PrintDList(head);

// 变换链表

TransformDList(head);

// 输出变换后的链表

std::cout << "变换后的链表内容: ";

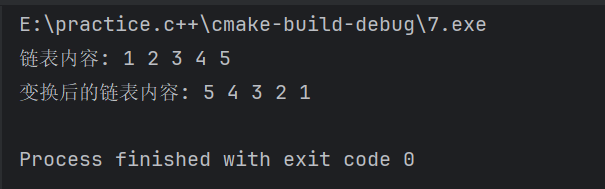
PrintDList(head);

// 释放内存

FreeDList(head);

return 0;

}



8.

#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 循环双链表结点结构

struct DLinkNode {

ElemType data; // 数据域

DLinkNode\* next; // 指向下一个结点的指针

DLinkNode\* prev; // 指向前一个结点的指针

};

// 创建一个循环双链表并返回头结点

DLinkNode\* CreateCircularDList(int n) {

DLinkNode\* head = new DLinkNode; // 头结点

head->next = head;

head->prev = head;

DLinkNode\* tail = head; // 记录链表的尾部

// 添加结点

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

DLinkNode\* newNode = new DLinkNode{i, nullptr, nullptr};

tail->next = newNode; // 将新结点链接到链表

newNode->prev = tail; // 新结点的前指针指向当前尾

tail = newNode; // 更新尾结点

}

// 形成循环

tail->next = head; // 尾结点指向头结点

head->prev = tail; // 头结点的前指针指向尾结点

return head; // 返回头结点

}

// 删除第一个值为 x 的结点

bool DeleteNodeByValue(DLinkNode\* head, ElemType x) {

if (head == nullptr || head->next == head) {

return false; // 链表为空或只有头结点

}

DLinkNode\* current = head->next; // 从第一个有效结点开始

// 遍历链表查找值为 x 的结点

do {

if (current->data == x) {

// 找到结点，进行删除

current->prev->next = current->next; // 前一个结点的 next 指向当前结点的下一个结点

current->next->prev = current->prev; // 下一个结点的 prev 指向当前结点的前一个结点

delete current; // 释放内存

return true; // 返回 true

}

current = current->next; // 移动到下一个结点

} while (current != head); // 循环直到回到头结点

return false; // 未找到值为 x 的结点

}

// 打印链表

void PrintCircularDList(DLinkNode\* head) {

if (head == nullptr) return;

DLinkNode\* current = head->next; // 从第一个有效结点开始

do {

std::cout << current->data << " ";

current = current->next;

} while (current != head);

std::cout << std::endl;

}

// 释放链表内存

void FreeCircularDList(DLinkNode\* head) {

if (head == nullptr) return;

DLinkNode\* current = head->next; // 从第一个有效结点开始

DLinkNode\* temp;

while (current != head) {

temp = current;

current = current->next;

delete temp;

}

delete head; // 释放头结点

}

// 测试代码

int main() {

// 创建一个循环双链表

int n = 5;

DLinkNode\* head = CreateCircularDList(n);

// 输出链表

std::cout << "链表内容: ";

PrintCircularDList(head);

// 删除值为 3 的结点

if (DeleteNodeByValue(head, 3)) {

std::cout << "删除值为 3 的结点成功。" << std::endl;

} else {

std::cout << "未找到值为 3 的结点。" << std::endl;

}

// 输出删除后的链表

std::cout << "删除后的链表内容: ";

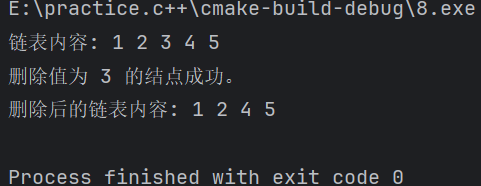
PrintCircularDList(head);

// 释放内存

FreeCircularDList(head);

return 0;

}



9.

#include <iostream>

typedef int ElemType; // 定义元素类型为整型

// 循环双链表结点结构

struct DLinkNode {

ElemType data; // 数据域

DLinkNode\* next; // 指向下一个结点的指针

DLinkNode\* prev; // 指向前一个结点的指针

};

// 创建一个循环双链表并返回头结点

DLinkNode\* CreateCircularDList(int\* arr, int n) {

DLinkNode\* head = new DLinkNode; // 头结点

head->next = head;

head->prev = head;

DLinkNode\* tail = head; // 尾结点

// 添加结点

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

DLinkNode\* newNode = new DLinkNode{arr[i], nullptr, nullptr};

tail->next = newNode; // 将新结点链接到链表

newNode->prev = tail; // 新结点的前指针指向当前尾

tail = newNode; // 更新尾结点

}

// 形成循环

tail->next = head; // 尾结点指向头结点

head->prev = tail; // 头结点的前指针指向尾结点

return head; // 返回头结点

}

// 判断循环双链表中数据结点是否对称

bool IsSymmetric(DLinkNode\* head) {

if (head == nullptr || head->next == head) {

return true; // 空链表或只有头结点为对称

}

DLinkNode\* front = head->next; // 从第一个有效结点开始

DLinkNode\* back = head->prev; // 从最后一个有效结点开始

// 使用双指针比较数据

while (front != back && front->prev != back) {

if (front->data != back->data) {

return false; // 一旦不相等，则不是对称的

}

front = front->next; // 向后移动

back = back->prev; // 向前移动

}

return true; // 所有数据相等，链表是对称的

}

// 打印链表

void PrintCircularDList(DLinkNode\* head) {

if (head == nullptr) return;

DLinkNode\* current = head->next; // 从第一个有效结点开始

do {

std::cout << current->data << " ";

current = current->next;

} while (current != head);

std::cout << std::endl;

}

// 释放链表内存

void FreeCircularDList(DLinkNode\* head) {

if (head == nullptr) return;

DLinkNode\* current = head->next; // 从第一个有效结点开始

DLinkNode\* temp;

while (current != head) {

temp = current;

current = current->next;

delete temp;

}

delete head; // 释放头结点

}

// 测试代码

int main() {

// 创建一个循环双链表（可以根据需要修改此处的数组）

int arr[] = {1, 2, 3, 2, 1}; // 例子：对称的链表

int n = sizeof(arr) / sizeof(arr[0]);

DLinkNode\* head = CreateCircularDList(arr, n);

// 输出链表

std::cout << "链表内容: ";

PrintCircularDList(head);

// 判断数据结点是否对称

if (IsSymmetric(head)) {

std::cout << "链表中的数据结点是对称的。" << std::endl;

} else {

std::cout << "链表中的数据结点不是对称的。" << std::endl;

}

// 释放内存

FreeCircularDList(head);

return 0;

}

