实验1代码

（1）#include <iostream>

#include <memory>

// 定义链表结点结构

struct Node {

int data;

std::shared\_ptr<Node> next;

Node(int val) : data(val), next(nullptr) {}

};

// 获取链表中第i个结点的指针

std::shared\_ptr<Node> getNodeAtPosition(const std::shared\_ptr<Node>& head, int i) {

if (i < 0) {

return nullptr;

}

std::shared\_ptr<Node> current = head;

int count = 0;

while (current != nullptr) {

if (count == i) {

return current;

}

count++;

current = current->next;

}

return nullptr;

}

// 打印链表

void printList(const std::shared\_ptr<Node>& head) {

std::shared\_ptr<Node> current = head;

while (current != nullptr) {

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

current = current->next;

}

std::cout << "NULL" << std::endl;

}

int main() {

// 创建链表

auto head = std::make\_shared<Node>(1);

auto current = head;

for (int i = 2; i <= 10; i++) {

current->next = std::make\_shared<Node>(i);

current = current->next;

}

// 测试第一组数据

int testIndices1[] = { 5, 10, 0, 11, 12 };

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

auto result = getNodeAtPosition(head, testIndices1[i]);

if (result != nullptr) {

std::cout << "Node at position " << testIndices1[i] << ": " << result->data << std::endl;

}

else {

std::cout << "Node at position " << testIndices1[i] << ": NULL" << std::endl;

}

}

// 创建空链表

std::shared\_ptr<Node> emptyHead = nullptr;

// 测试第二组数据

int testIndices2[] = { 0, 2 };

for (int i = 0; i < 2; i++) {

auto result = getNodeAtPosition(emptyHead, testIndices2[i]);

if (result != nullptr) {

std::cout << "Node at position " << testIndices2[i] << ": " << result->data << std::endl;

}

else {

std::cout << "Node at position " << testIndices2[i] << ": NULL" << std::endl;

}

}

return 0;

}

（2）#include <iostream>

#include <memory>

// 定义链表结点结构

struct Node {

int data;

std::shared\_ptr<Node> next;

Node(int val) : data(val), next(nullptr) {}

};

// 在链表第i个结点前插入值为x的结点

void insertNodeBeforePosition(std::shared\_ptr<Node>& head, int i, int x) {

if (i < 0) {

std::cout << "Invalid position: " << i << std::endl;

return;

}

auto newNode = std::make\_shared<Node>(x);

if (i == 0) {

newNode->next = head;

head = newNode;

return;

}

std::shared\_ptr<Node> current = head;

int count = 0;

while (current != nullptr && count < i - 1) {

current = current->next;

count++;

}

if (current == nullptr) {

std::cout << "Invalid position: " << i << std::endl;

return;

}

newNode->next = current->next;

current->next = newNode;

}

// 打印链表

void printList(const std::shared\_ptr<Node>& head) {

std::shared\_ptr<Node> current = head;

while (current != nullptr) {

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

current = current->next;

}

std::cout << "NULL" << std::endl;

}

int main() {

// 创建链表

auto head = std::make\_shared<Node>(1);

auto current = head;

for (int i = 2; i <= 10; i++) {

current->next = std::make\_shared<Node>(i);

current = current->next;

}

// 测试第一组数据

int testIndices1[] = { 5, 10, 11, 0, 1, 12 };

int x = 100;

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

insertNodeBeforePosition(head, testIndices1[i], x);

std::cout << "After inserting " << x << " before position " << testIndices1[i] << ": ";

printList(head);

}

// 创建空链表

std::shared\_ptr<Node> emptyHead = nullptr;

// 测试第二组数据

int testIndex2 = 5;

insertNodeBeforePosition(emptyHead, testIndex2, x);

std::cout << "After inserting " << x << " before position " << testIndex2 << ": ";

printList(emptyHead);

return 0;

}

（3）#include <iostream>

#include <memory>

// 定义链表结点结构

struct Node {

int data;

std::shared\_ptr<Node> next;

Node(int val) : data(val), next(nullptr) {}

};

// 删除链表中第i个元素结点

void deleteNodeAtPosition(std::shared\_ptr<Node>& head, int i) {

if (i < 0) {

std::cout << "Invalid position: " << i << std::endl;

return;

}

if (head == nullptr) {

std::cout << "List is empty, cannot delete from empty list." << std::endl;

return;

}

if (i == 0) {

head = head->next;

return;

}

std::shared\_ptr<Node> current = head;

int count = 0;

while (current != nullptr && count < i - 1) {

current = current->next;

count++;

}

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

std::cout << "Invalid position: " << i << std::endl;

return;

}

current->next = current->next->next;

}

// 打印链表

void printList(const std::shared\_ptr<Node>& head) {

std::shared\_ptr<Node> current = head;

while (current != nullptr) {

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

current = current->next;

}

std::cout << "NULL" << std::endl;

}

int main() {

// 创建链表

auto head = std::make\_shared<Node>(1);

auto current = head;

for (int i = 2; i <= 10; i++) {

current->next = std::make\_shared<Node>(i);

current = current->next;

}

// 测试第一组数据

int testIndices1[] = {5, 10, 1, 11, 0};

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

deleteNodeAtPosition(head, testIndices1[i]);

std::cout << "After deleting node at position " << testIndices1[i] << ": ";

printList(head);

}

// 创建空链表

std::shared\_ptr<Node> emptyHead = nullptr;

// 测试第二组数据

int testIndex2 = 5;

deleteNodeAtPosition(emptyHead, testIndex2);

std::cout << "After deleting node at position " << testIndex2 << ": ";

printList(emptyHead);

return 0;

}

（4）#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

void insertSorted(ListNode\*& head, int x) {

ListNode\* newNode = new ListNode(x);

// 如果链表为空或者新节点应该插入在链表头部

if (head == nullptr || head->val >= x) {

newNode->next = head;

head = newNode;

return;

}

// 找到插入位置

ListNode\* current = head;

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

current = current->next;

}

// 插入新节点

newNode->next = current->next;

current->next = newNode;

}

void printList(ListNode\* head) {

ListNode\* current = head;

while (current != nullptr) {

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

current = current->next;

}

std::cout << std::endl;

}

int main() {

// 创建初始链表

ListNode\* head = new ListNode(10);

ListNode\* current = head;

int values[] = { 20, 30, 40, 50, 60, 70, 80, 90, 100 };

for (int val : values) {

current->next = new ListNode(val);

current = current->next;

}

// 测试插入

int testValues[] = { 25, 85, 110, 8 };

for (int x : testValues) {

insertSorted(head, x);

printList(head);

}

// 释放内存

while (head != nullptr) {

ListNode\* temp = head;

head = head->next;

delete temp;

}

return 0;

}

（5）#include <iostream>

using namespace std;

// 定义链表节点结构

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

// 创建链表

ListNode\* createList(const int arr[], int n) {

if (n == 0) return nullptr;

ListNode\* head = new ListNode(arr[0]);

ListNode\* current = head;

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

current->next = new ListNode(arr[i]);

current = current->next;

}

return head;

}

// 打印链表

void printList(ListNode\* head) {

ListNode\* current = head;

while (current != nullptr) {

cout << current->val << " ";

current = current->next;

}

cout << endl;

}

// 分解链表为奇数项和偶数项

void splitList(ListNode\* head, ListNode\*& oddHead, ListNode\*& evenHead) {

if (head == nullptr) return;

oddHead = new ListNode(0); // 带头结点的单链表

evenHead = new ListNode(0); // 带头结点的单链表

ListNode\* oddTail = oddHead;

ListNode\* evenTail = evenHead;

ListNode\* current = head;

int index = 1;

while (current != nullptr) {

if (index % 2 != 0) {

oddTail->next = current;

oddTail = oddTail->next;

}

else {

evenTail->next = current;

evenTail = evenTail->next;

}

current = current->next;

index++;

}

oddTail->next = nullptr;

evenTail->next = nullptr;

// 去掉头结点

oddHead = oddHead->next;

evenHead = evenHead->next;

}

int main() {

// 第一组数据

int arr1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60 };

ListNode\* head1 = createList(arr1, sizeof(arr1) / sizeof(arr1[0]));

cout << "原链表1: ";

printList(head1);

ListNode\* oddHead1 = nullptr;

ListNode\* evenHead1 = nullptr;

splitList(head1, oddHead1, evenHead1);

cout << "奇数项链表1: ";

printList(oddHead1);

cout << "偶数项链表1: ";

printList(evenHead1);

// 第二组数据

int arr2[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

ListNode\* head2 = createList(arr2, sizeof(arr2) / sizeof(arr2[0]));

cout << "原链表2: ";

printList(head2);

ListNode\* oddHead2 = nullptr;

ListNode\* evenHead2 = nullptr;

splitList(head2, oddHead2, evenHead2);

cout << "奇数项链表2: ";

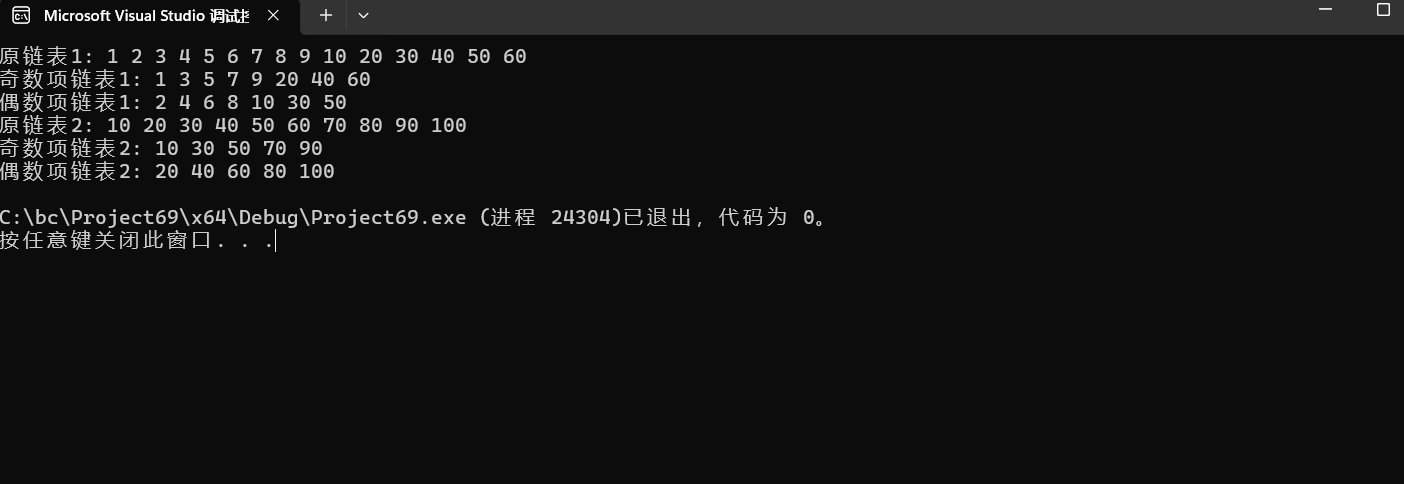
printList(oddHead2);

cout << "偶数项链表2: ";

printList(evenHead2);

return 0;

}



#include <iostream>

using namespace std;

// 定义链表节点结构

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

// 创建链表

ListNode\* createList(const int arr[], int n) {

if (n == 0) return nullptr;

ListNode\* head = new ListNode(arr[0]);

ListNode\* current = head;

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

current->next = new ListNode(arr[i]);

current = current->next;

}

return head;

}

// 打印链表

void printList(ListNode\* head) {

ListNode\* current = head;

while (current != nullptr) {

cout << current->val << " ";

current = current->next;

}

cout << endl;

}

// 找到两个递增有序链表的公共元素，并连接成链表L3

ListNode\* findCommonElements(ListNode\* L1, ListNode\* L2) {

ListNode dummy(0);

ListNode\* tail = &dummy;

ListNode\* p1 = L1;

ListNode\* p2 = L2;

while (p1 != nullptr && p2 != nullptr) {

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

tail->next = new ListNode(p1->val);

tail = tail->next;

p1 = p1->next;

p2 = p2->next;

}

else if (p1->val < p2->val) {

p1 = p1->next;

}

else {

p2 = p2->next;

}

}

return dummy.next;

}

int main() {

// 第一组数据

int arr1[] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

int arr2[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };

ListNode\* L1 = createList(arr1, sizeof(arr1) / sizeof(arr1[0]));

ListNode\* L2 = createList(arr2, sizeof(arr2) / sizeof(arr2[0]));

cout << "第一个链表: ";

printList(L1);

cout << "第二个链表: ";

printList(L2);

ListNode\* L3 = findCommonElements(L1, L2);

cout << "公共元素链表: ";

printList(L3);

// 第二组数据

int arr3[] = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

int arr4[] = { 2, 4, 5, 7, 8, 9, 12, 22 };

ListNode\* L4 = createList(arr3, sizeof(arr3) / sizeof(arr3[0]));

ListNode\* L5 = createList(arr4, sizeof(arr4) / sizeof(arr4[0]));

cout << "第一个链表: ";

printList(L4);

cout << "第二个链表: ";

printList(L5);

ListNode\* L6 = findCommonElements(L4, L5);

cout << "公共元素链表: ";

printList(L6);

// 第三组数据

int arr5[10] = {};

int arr6[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

ListNode\* L7 = createList(arr5, sizeof(arr5) / sizeof(arr5[0]));

ListNode\* L8 = createList(arr6, sizeof(arr6) / sizeof(arr6[0]));

cout << "第一个链表: ";

printList(L7);

cout << "第二个链表: ";

printList(L8);

ListNode\* L9 = findCommonElements(L7, L8);

cout << "公共元素链表: ";

printList(L9);

return 0;

}

实验2代码

（1）#include <iostream>

#include <vector>

// 定义链表节点结构

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

// 创建单循环链表

Node\* createCircularLinkedList(const std::vector<int>& elements) {

if (elements.empty()) return nullptr;

Node\* head = new Node(elements[0]);

Node\* current = head;

for (size\_t i = 1; i < elements.size(); ++i) {

current->next = new Node(elements[i]);

current = current->next;

}

// 将最后一个节点的next指向head，形成循环

current->next = head;

return head;

}

// 依次访问单循环链表的各结点

void traverseCircularLinkedList(Node\* head) {

if (!head) return;

Node\* current = head;

do {

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

current = current->next;

} while (current != head);

std::cout << std::endl;

}

// 释放链表内存

void deleteCircularLinkedList(Node\* head) {

if (!head) return;

Node\* current = head;

Node\* next = nullptr;

do {

next = current->next;

delete current;

current = next;

} while (current != head);

}

int main() {

// 第一组数据

std::vector<int> elements1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60 };

Node\* head1 = createCircularLinkedList(elements1);

std::cout << "第一组数据：";

traverseCircularLinkedList(head1);

deleteCircularLinkedList(head1);

// 第二组数据

std::vector<int> elements2 = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

Node\* head2 = createCircularLinkedList(elements2);

std::cout << "第二组数据：";

traverseCircularLinkedList(head2);

deleteCircularLinkedList(head2);

return 0;

}

（2）#include <iostream>

#include <vector>

#include <cmath>

// 定义链表节点结构

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

// 创建带头结点的单循环链表

Node\* createCircularLinkedList(const std::vector<int>& elements) {

if (elements.empty()) return nullptr;

Node\* head = new Node(0); // 头结点

Node\* current = head;

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

current->next = new Node(elements[i]);

current = current->next;

}

// 将最后一个节点的next指向head->next，形成循环

current->next = head->next;

return head;

}

// 判断链表是否满足条件

bool checkCondition(Node\* head) {

if (!head || !head->next) return true; // 空链表或只有一个头结点

Node\* current = head->next;

int index = 1;

do {

if (std::abs(current->data - index) > 3) {

return false;

}

current = current->next;

++index;

} while (current != head->next);

return true;

}

// 释放链表内存

void deleteCircularLinkedList(Node\* head) {

if (!head) return;

Node\* current = head->next;

Node\* next = nullptr;

do {

next = current->next;

delete current;

current = next;

} while (current != head->next);

delete head; // 删除头结点

}

int main() {

// 第一组数据

std::vector<int> elements1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18 };

Node\* head1 = createCircularLinkedList(elements1);

std::cout << "第一组数据：" << (checkCondition(head1) ? "TRUE" : "FALSE") << std::endl;

deleteCircularLinkedList(head1);

// 第二组数据

std::vector<int> elements2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 20, 18 };

Node\* head2 = createCircularLinkedList(elements2);

std::cout << "第二组数据：" << (checkCondition(head2) ? "TRUE" : "FALSE") << std::endl;

deleteCircularLinkedList(head2);

return 0;

}

（3）#include <iostream>

#include <vector>

// 定义链表节点结构

struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

// 创建单循环链表

Node\* createCircularLinkedList(const std::vector<int>& elements) {

if (elements.empty()) return nullptr;

Node\* head = new Node(elements[0]);

Node\* current = head;

for (size\_t i = 1; i < elements.size(); ++i) {

current->next = new Node(elements[i]);

current = current->next;

}

// 将最后一个节点的next指向head，形成循环

current->next = head;

return head;

}

// 求两个递增有序单循环链表的交集

Node\* intersection(Node\* head1, Node\* head2) {

if (!head1 || !head2) return nullptr;

Node\* resultHead = nullptr;

Node\* resultTail = nullptr;

Node\* current1 = head1;

Node\* current2 = head2;

do {

if (current1->data == current2->data) {

if (!resultHead) {

resultHead = new Node(current1->data);

resultTail = resultHead;

}

else {

if (resultTail->data != current1->data) {

resultTail->next = new Node(current1->data);

resultTail = resultTail->next;

}

}

current1 = current1->next;

current2 = current2->next;

}

else if (current1->data < current2->data) {

current1 = current1->next;

}

else {

current2 = current2->next;

}

} while (current1 != head1 && current2 != head2);

if (resultHead) {

resultTail->next = resultHead;

}

return resultHead;

}

// 求两个递增有序单循环链表的并集

Node\* unionSet(Node\* head1, Node\* head2) {

if (!head1) return head2;

if (!head2) return head1;

Node\* resultHead = nullptr;

Node\* resultTail = nullptr;

Node\* current1 = head1;

Node\* current2 = head2;

do {

if (current1->data == current2->data) {

if (!resultHead) {

resultHead = new Node(current1->data);

resultTail = resultHead;

}

else {

if (resultTail->data != current1->data) {

resultTail->next = new Node(current1->data);

resultTail = resultTail->next;

}

}

current1 = current1->next;

current2 = current2->next;

}

else if (current1->data < current2->data) {

if (!resultHead) {

resultHead = new Node(current1->data);

resultTail = resultHead;

}

else {

if (resultTail->data != current1->data) {

resultTail->next = new Node(current1->data);

resultTail = resultTail->next;

}

}

current1 = current1->next;

}

else {

if (!resultHead) {

resultHead = new Node(current2->data);

resultTail = resultHead;

}

else {

if (resultTail->data != current2->data) {

resultTail->next = new Node(current2->data);

resultTail = resultTail->next;

}

}

current2 = current2->next;

}

} while (current1 != head1 && current2 != head2);

while (current1 != head1) {

if (resultTail->data != current1->data) {

resultTail->next = new Node(current1->data);

resultTail = resultTail->next;

}

current1 = current1->next;

}

while (current2 != head2) {

if (resultTail->data != current2->data) {

resultTail->next = new Node(current2->data);

resultTail = resultTail->next;

}

current2 = current2->next;

}

if (resultHead) {

resultTail->next = resultHead;

}

return resultHead;

}

// 打印单循环链表

void printCircularLinkedList(Node\* head) {

if (!head) return;

Node\* current = head;

do {

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

current = current->next;

} while (current != head);

std::cout << std::endl;

}

// 释放链表内存

void deleteCircularLinkedList(Node\* head) {

if (!head) return;

Node\* current = head;

Node\* next = nullptr;

do {

next = current->next;

delete current;

current = next;

} while (current != head);

}

int main() {

// 第一组数据

std::vector<int> elements1\_1 = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

std::vector<int> elements1\_2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 20, 30 };

Node\* head1\_1 = createCircularLinkedList(elements1\_1);

Node\* head1\_2 = createCircularLinkedList(elements1\_2);

Node\* intersection1 = intersection(head1\_1, head1\_2);

Node\* union1 = unionSet(head1\_1, head1\_2);

std::cout << "第一组交集：";

printCircularLinkedList(intersection1);

std::cout << "第一组并集：";

printCircularLinkedList(union1);

deleteCircularLinkedList(head1\_1);

deleteCircularLinkedList(head1\_2);

deleteCircularLinkedList(intersection1);

deleteCircularLinkedList(union1);

// 第二组数据

std::vector<int> elements2\_1 = { 1, 3, 6, 10, 15, 16, 17, 18, 19, 20 };

std::vector<int> elements2\_2 = { 2, 4, 5, 7, 8, 9, 12, 22 };

Node\* head2\_1 = createCircularLinkedList(elements2\_1);

Node\* head2\_2 = createCircularLinkedList(elements2\_2);

Node\* intersection2 = intersection(head2\_1, head2\_2);

Node\* union2 = unionSet(head2\_1, head2\_2);

std::cout << "第二组交集：";

printCircularLinkedList(intersection2);

std::cout << "第二组并集：";

printCircularLinkedList(union2);

deleteCircularLinkedList(head2\_1);

deleteCircularLinkedList(head2\_2);

deleteCircularLinkedList(intersection2);

deleteCircularLinkedList(union2);

// 第三组数据

std::vector<int> elements3\_1 = {};

std::vector<int> elements3\_2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

Node\* head3\_1 = createCircularLinkedList(elements3\_1);

Node\* head3\_2 = createCircularLinkedList(elements3\_2);

Node\* intersection3 = intersection(head3\_1, head3\_2);

Node\* union3 = unionSet(head3\_1, head3\_2);

std::cout << "第三组交集：";

printCircularLinkedList(intersection3);

std::cout << "第三组并集：";

printCircularLinkedList(union3);

deleteCircularLinkedList(head3\_1);

deleteCircularLinkedList(head3\_2);

deleteCircularLinkedList(intersection3);

deleteCircularLinkedList(union3);

return 0;

}

（4）#include <iostream>

#include <vector>

// 定义双循环链表节点结构

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// 创建带头结点的双循环链表

Node\* createDoublyCircularLinkedList(const std::vector<int>& elements) {

if (elements.empty()) return nullptr;

// 创建头结点

Node\* head = new Node(0);

Node\* current = head;

// 创建链表节点

for (int element : elements) {

Node\* newNode = new Node(element);

newNode->prev = current;

current->next = newNode;

current = newNode;

}

// 将最后一个节点的next指向头结点的next，形成循环

current->next = head->next;

// 将头结点的next的prev指向最后一个节点，形成循环

head->next->prev = current;

return head;

}

// 打印双循环链表

void printDoublyCircularLinkedList(Node\* head) {

if (!head || !head->next) return;

Node\* current = head->next;

do {

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

current = current->next;

} while (current != head->next);

std::cout << std::endl;

}

// 释放双循环链表内存

void deleteDoublyCircularLinkedList(Node\* head) {

if (!head || !head->next) return;

Node\* current = head->next;

Node\* next = nullptr;

do {

next = current->next;

delete current;

current = next;

} while (current != head->next);

delete head; // 删除头结点

}

int main() {

// 第一组数据

std::vector<int> elements1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

Node\* head1 = createDoublyCircularLinkedList(elements1);

std::cout << "第一组数据：";

printDoublyCircularLinkedList(head1);

deleteDoublyCircularLinkedList(head1);

// 第二组数据

std::vector<int> elements2 = { 10, 30, 40, 55, 60, 70, 88, 99, 100 };

Node\* head2 = createDoublyCircularLinkedList(elements2);

std::cout << "第二组数据：";

printDoublyCircularLinkedList(head2);

deleteDoublyCircularLinkedList(head2);

return 0;

}

（5）

};

bool isSymmetric(Node\* head) {

if (head->next == head) {

// 空链表或只有一个元素的链表是对称的

return true;

}

Node\* front = head->next;

Node\* back = head->prev;

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

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

return false;

}

front = front->next;

back = back->prev;

}

return true;

}

// 辅助函数：创建双循环链表

Node\* createDoublyCircularLinkedList(int arr[], int n) {

Node\* head = new Node();

head->next = head;

head->prev = head;

Node\* current = head;

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

Node\* newNode = new Node();

newNode->data = arr[i];

newNode->next = head;

newNode->prev = current;

current->next = newNode;

head->prev = newNode;

current = newNode;

}

return head;

}

// 辅助函数：打印链表（用于调试）

void printList(Node\* head) {

Node\* current = head->next;

while (current != head) {

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

current = current->next;

}

std::cout << std::endl;

}

int main() {

int arr1[] = { 1, 2, 3, 4, 5, 4, 3, 2, 1 };

int arr2[] = { 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 };

int arr3[] = { 1, 2, 3, 4, 5, 6, 3, 2, 1 };

int arr4[] = { 1, 2, 3, 4, 5, 5, 6, 4, 3, 2, 1 };

Node\* head1 = createDoublyCircularLinkedList(arr1, sizeof(arr1) / sizeof(arr1[0]));

Node\* head2 = createDoublyCircularLinkedList(arr2, sizeof(arr2) / sizeof(arr2[0]));

Node\* head3 = createDoublyCircularLinkedList(arr3, sizeof(arr3) / sizeof(arr3[0]));

Node\* head4 = createDoublyCircularLinkedList(arr4, sizeof(arr4) / sizeof(arr4[0]));

std::cout << "第一组数据: " << (isSymmetric(head1) ? "TRUE" : "FALSE") << std::endl;

std::cout << "第二组数据: " << (isSymmetric(head2) ? "TRUE" : "FALSE") << std::endl;

std::cout << "第三组数据: " << (isSymmetric(head3) ? "TRUE" : "FALSE") << std::endl;

std::cout << "第四组数据: " << (isSymmetric(head4) ? "TRUE" : "FALSE") << std::endl;

return 0;

}

实验3代码

（1）#include <iostream>

#include <stack>

#include <string>

#include <cctype>

#include <cmath>

bool isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/');

}

int getPrecedence(char op) {

if (op == '+' || op == '-') {

return 1;

}

else if (op == '\*' || op == '/') {

return 2;

}

return 0;

}

void applyOperator(std::stack<int>& operands, char op) {

int right = operands.top();

operands.pop();

int left = operands.top();

operands.pop();

switch (op) {

case '+':

operands.push(left + right);

break;

case '-':

operands.push(left - right);

break;

case '\*':

operands.push(left \* right);

break;

case '/':

operands.push(left / right);

break;

}

}

int evaluateExpression(const std::string& expression) {

std::stack<int> operands;

std::stack<char> operators;

for (int i = 0; i < expression.length(); ++i) {

if (expression[i] == ' ') {

continue;

}

else if (isdigit(expression[i])) {

int val = 0;

while (i < expression.length() && isdigit(expression[i])) {

val = val \* 10 + (expression[i] - '0');

i++;

}

i--;

operands.push(val);

}

else if (expression[i] == '(') {

operators.push(expression[i]);

}

else if (expression[i] == ')') {

while (!operators.empty() && operators.top() != '(') {

applyOperator(operands, operators.top());

operators.pop();

}

operators.pop(); // Discard the '('

}

else if (isOperator(expression[i])) {

while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(expression[i])) {

applyOperator(operands, operators.top());

operators.pop();

}

operators.push(expression[i]);

}

}

while (!operators.empty()) {

applyOperator(operands, operators.top());

operators.pop();

}

return operands.top();

}

int main() {

std::string expression = "((8 + 2) \* 5) / (1 + 1)";

int result = evaluateExpression(expression);

std::cout << "Result: " << result << std::endl;

return 0;

}

（2）#include <iostream>

#include <vector>

int josephus(int n, int k, bool isJosephus) {

if (n == 1) {

return 0;

}

else {

if (isJosephus) {

return (josephus(n - 1, k, true) + k) % n;

}

else {

return (josephus(n - 1, k, true) - k + n) % n;

}

}

}

int main() {

int n = 41; // 41个人

int k = 3; // 数到3的人被杀

int josephusPosition = josephus(n, k, true) + 1; // +1 是因为索引从0开始

int friendPosition = josephus(n, k, false) + 1; // +1 是因为索引从0开始

std::cout << "约瑟夫的位置为：" << josephusPosition << std::endl;

std::cout << "朋友的位置为：" << friendPosition << std::endl;

return 0;

}

(3) #include <iostream>

#include <stack>

#include <string>

#include <cctype>

class ChainString {

private:

std::string str;

public:

ChainString(std::string s) : str(s) {}

int compare(ChainString s) {

if (str < s.str) return -1;

else if (str == s.str) return 0;

else return 1;

}

};

int main() {

// Compare ChainString

ChainString s1("abc");

ChainString s2("def");

int comparison = s1.compare(s2);

std::cout << "Comparison result: " << comparison << std::endl;

return 0;

}

实验4代码

（1）#include <iostream>

#include <vector>

class SparseMatrix {

private:

int rows;

int cols;

std::vector<std::vector<int>> data;

public:

SparseMatrix(int r, int c, const std::vector<std::vector<int>>& d) : rows(r), cols(c), data(d) {}

SparseMatrix operator+(const SparseMatrix& other) {

// Add the two sparse matrices

if (rows != other.rows || cols != other.cols) {

throw std::invalid\_argument("Matrix dimensions don't match");

}

std::vector<std::vector<int>> result(rows, std::vector<int>(cols, 0));

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

for (int j = 0; j < cols; j++) {

result[i][j] = data[i][j] + other.data[i][j];

}

}

return SparseMatrix(rows, cols, result);

}

SparseMatrix operator\*(const SparseMatrix& other) {

// Multiply the two sparse matrices

if (cols != other.rows) {

throw std::invalid\_argument("Matrix dimensions don't match for multiplication");

}

std::vector<std::vector<int>> result(rows, std::vector<int>(other.cols, 0));

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

for (int j = 0; j < other.cols; j++) {

for (int k = 0; k < cols; k++) {

result[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return SparseMatrix(rows, other.cols, result);

}

void print() {

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

for (int j = 0; j < cols; j++) {

std::cout << data[i][j] << " ";

}

std::cout << std::endl;

}

}

};

int main() {

// Test cases

std::vector<std::vector<int>> data1 = { {1, 0, 0}, {0, 2, 0}, {0, 0, 3} };

std::vector<std::vector<int>> data2 = { {4, 0, 0}, {0, 5, 0}, {0, 0, 6} };

SparseMatrix matrix1(3, 3, data1);

SparseMatrix matrix2(3, 3, data2);

// Addition

SparseMatrix sum = matrix1 + matrix2;

sum.print();

// Multiplication

SparseMatrix product = matrix1 \* matrix2;

product.print();

return 0;

}

（2）#include <iostream>

#include <string>

#include <vector>

class Undergraduate {

public:

std::string name;

// other information about undergraduate

};

class Graduate {

public:

std::string name;

std::vector<Undergraduate> undergraduates;

// other information about graduate

};

class Teacher {

public:

std::string name;

Graduate graduate;

std::vector<Undergraduate> undergraduates;

void displayTeacherInfo() const {

std::cout << "Teacher: " << name << std::endl;

std::cout << "Graduate Student: " << graduate.name << std::endl;

std::cout << "Undergraduate Students: ";

for (const auto& undergrad : undergraduates) {

std::cout << undergrad.name << " ";

}

std::cout << std::endl;

}

};

class GeneralizedList {

private:

std::vector<Teacher> teachers;

public:

void insertTeacher(const Teacher& teacher) {

teachers.push\_back(teacher);

}

void deleteTeacher(const std::string& teacherName) {

for (auto it = teachers.begin(); it != teachers.end(); ++it) {

if (it->name == teacherName) {

teachers.erase(it);

break;

}

}

}

void queryTeacher(const std::string& teacherName) {

for (const auto& teacher : teachers) {

if (teacher.name == teacherName) {

teacher.displayTeacherInfo();

}

}

}

void outputList() {

for (const auto& teacher : teachers) {

teacher.displayTeacherInfo();

}

}

};

int main() {

// Test case

GeneralizedList list;

Teacher teacher1;

teacher1.name = "Dr. Smith";

Graduate graduate1;

graduate1.name = "Jack";

Undergraduate undergrad1;

undergrad1.name = "Alice";

graduate1.undergraduates.push\_back(undergrad1);

teacher1.graduate = graduate1;

Undergraduate undergrad2;

undergrad2.name = "Bob";

teacher1.undergraduates.push\_back(undergrad2);

Teacher teacher2;

teacher2.name = "Prof. Johnson";

Graduate graduate2;

graduate2.name = "Emily";

Undergraduate undergrad3;

undergrad3.name = "Carol";

graduate2.undergraduates.push\_back(undergrad3);

teacher2.graduate = graduate2;

Undergraduate undergrad4;

undergrad4.name = "David";

teacher2.undergraduates.push\_back(undergrad4);

list.insertTeacher(teacher1);

list.insertTeacher(teacher2);

list.outputList();

list.queryTeacher("Dr. Smith");

list.deleteTeacher("Prof. Johnson");

list.outputList();

return 0;

}

实验5代码

（1）#include <iostream>

#include <vector>

// 定义二叉树节点结构

struct TreeNode {

char value;

int level; // 节点所在的层次

TreeNode\* left;

TreeNode\* right;

TreeNode(char val) : value(val), level(0), left(nullptr), right(nullptr) {}

};

// 构建二叉树并记录每个节点的层次数

void buildTreeWithLevel(const std::vector<std::pair<char, std::pair<int, int>>>& nodes, std::vector<TreeNode\*>& treeNodes, TreeNode\* root, int level) {

if (root == nullptr) return;

root->level = level;

if (nodes[root->value - 'A'].second.first != -1) {

root->left = treeNodes[nodes[root->value - 'A'].second.first];

buildTreeWithLevel(nodes, treeNodes, root->left, level + 1);

}

if (nodes[root->value - 'A'].second.second != -1) {

root->right = treeNodes[nodes[root->value - 'A'].second.second];

buildTreeWithLevel(nodes, treeNodes, root->right, level + 1);

}

}

// 中序遍历输出节点值及对应的层次数

void inorderTraversalWithLevel(TreeNode\* root, std::vector<std::pair<char, int>>& result) {

if (root == nullptr) return;

inorderTraversalWithLevel(root->left, result);

result.push\_back(std::make\_pair(root->value, root->level));

inorderTraversalWithLevel(root->right, result);

}

// 前序遍历

void preorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

std::cout << root->value << " ";

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// 后序遍历

void postorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

std::cout << root->value << " ";

}

// 在指定位置插入节点

void insertNode(TreeNode\* root, char parentValue, char newValue, bool isLeft) {

if (root == nullptr) return;

if (root->value == parentValue) {

if (isLeft) {

TreeNode\* newNode = new TreeNode(newValue);

newNode->left = root->left;

root->left = newNode;

}

else {

TreeNode\* newNode = new TreeNode(newValue);

newNode->right = root->right;

root->right = newNode;

}

return;

}

insertNode(root->left, parentValue, newValue, isLeft);

insertNode(root->right, parentValue, newValue, isLeft);

}

int main() {

// 构建二叉树

std::vector<std::pair<char, std::pair<int, int>>> nodes = {

{'A', {1, 2}}, {'B', {3, 4}}, {'C', {5, 6}}, {'D', {-1, -1}}, {'E', {-1, -1}}, {'F', {-1, -1}}, {'G', {-1, -1}}

};

std::vector<TreeNode\*> treeNodes(nodes.size(), nullptr);

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

treeNodes[i] = new TreeNode(nodes[i].first);

}

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

if (nodes[i].second.first != -1) {

treeNodes[i]->left = treeNodes[nodes[i].second.first];

}

if (nodes[i].second.second != -1) {

treeNodes[i]->right = treeNodes[nodes[i].second.second];

}

}

buildTreeWithLevel(nodes, treeNodes, treeNodes[0], 1);

// 中序遍历输出节点值及对应的层次数

std::vector<std::pair<char, int>> result;

inorderTraversalWithLevel(treeNodes[0], result);

std::cout << "中序遍历输出节点值及对应的层次数: ";

for (const auto& p : result) {

std::cout << p.first << ":" << p.second << ", ";

}

std::cout << std::endl;

// 遍历二叉树

std::cout << "前序遍历: ";

preorderTraversal(treeNodes[0]);

std::cout << std::endl;

// 插入新节点

insertNode(treeNodes[0], 'D', 'H', true);

// 遍历二叉树以验证插入结果

std::cout << "插入新节点后的前序遍历: ";

preorderTraversal(treeNodes[0]);

std::cout << std::endl;

return 0;

}

（2）#include <iostream>

#include <vector>

#include <fstream>

using namespace std;

struct TreeNode {

char val;

vector<TreeNode\*> children;

TreeNode(char x) : val(x) {}

};

struct Forest {

vector<TreeNode\*> trees;

};

struct BinaryTreeNode {

char val;

BinaryTreeNode\* left;

BinaryTreeNode\* right;

BinaryTreeNode(char x) : val(x), left(nullptr), right(nullptr) {}

};

void displayMenu() {

cout << "1. 采用指定输入建树或森林" << endl;

cout << "2. 读入文件建树或森林" << endl;

cout << "3. 实现各遍历算法" << endl;

cout << "4. 与二叉树的相互转换" << endl;

cout << "5. 退出" << endl;

}

void buildSubTree(TreeNode\* node) {

int numChildren;

cout << "输入节点 " << node->val << " 的子节点数量: ";

cin >> numChildren;

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

char childVal;

cout << "输入子节点 " << i + 1 << " 的值: ";

cin >> childVal;

TreeNode\* child = new TreeNode(childVal);

node->children.push\_back(child);

buildSubTree(child);

}

}

void buildSubTreeFromFile(TreeNode\* node, ifstream& file) {

int numChildren;

file >> numChildren;

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

char childVal;

file >> childVal;

TreeNode\* child = new TreeNode(childVal);

node->children.push\_back(child);

buildSubTreeFromFile(child, file);

}

}

TreeNode\* buildTreeFromFile(ifstream& file) {

char rootVal;

file >> rootVal;

TreeNode\* root = new TreeNode(rootVal);

// 递归构建子树

buildSubTreeFromFile(root, file);

return root;

}

TreeNode\* buildTreeFromInput() {

char rootVal;

cout << "输入根节点的值: ";

cin >> rootVal;

TreeNode\* root = new TreeNode(rootVal);

// 递归构建子树

buildSubTree(root);

return root;

}

void postOrderTraversal(TreeNode\* node) {

if (node) {

for (TreeNode\* child : node->children) {

postOrderTraversal(child);

}

cout << node->val;

}

}

BinaryTreeNode\* convertToBinaryTree(TreeNode\* node) {

if (!node) return nullptr;

BinaryTreeNode\* bNode = new BinaryTreeNode(node->val);

if (!node->children.empty()) {

bNode->left = convertToBinaryTree(node->children[0]);

}

BinaryTreeNode\* current = bNode->left;

for (int i = 1; i < node->children.size(); ++i) {

current->right = convertToBinaryTree(node->children[i]);

current = current->right;

}

return bNode;

}

void preOrderTraversal(BinaryTreeNode\* node) {

if (node) {

cout << node->val;

preOrderTraversal(node->left);

preOrderTraversal(node->right);

}

}

void inOrderTraversal(BinaryTreeNode\* node) {

if (node) {

inOrderTraversal(node->left);

cout << node->val;

inOrderTraversal(node->right);

}

}

void postOrderTraversal(BinaryTreeNode\* node) {

if (node) {

postOrderTraversal(node->left);

postOrderTraversal(node->right);

cout << node->val;

}

}

int main() {

int choice;

Forest forest;

while (true) {

displayMenu();

cout << "选择一个选项: ";

cin >> choice;

switch (choice) {

case 1: {

TreeNode\* tree = buildTreeFromInput();

forest.trees.push\_back(tree);

break;

}

case 2: {

ifstream file("tree\_data.txt");

if (file.is\_open()) {

TreeNode\* tree = buildTreeFromFile(file);

forest.trees.push\_back(tree);

file.close();

}

else {

cout << "无法打开文件" << endl;

}

break;

}

case 3: {

cout << "后序遍历森林: ";

for (TreeNode\* tree : forest.trees) {

postOrderTraversal(tree);

}

cout << endl;

break;

}

case 4: {

cout << "转换为二叉树并遍历: " << endl;

for (TreeNode\* tree : forest.trees) {

BinaryTreeNode\* bTree = convertToBinaryTree(tree);

cout << "前序遍历: ";

preOrderTraversal(bTree);

cout << endl;

cout << "中序遍历: ";

inOrderTraversal(bTree);

cout << endl;

cout << "后序遍历: ";

postOrderTraversal(bTree);

cout << endl;

}

cout << endl;

break;

}

case 5: {

return 0;

}

default: {

cout << "无效选项" << endl;

break;

}

}

}

return 0;

}

（3）#include <iostream>

#include <fstream>

#include <queue>

#include <unordered\_map>

#include <vector>

#include <string>

using namespace std;

// 哈夫曼树节点

struct HuffmanNode {

char data;

int frequency;

HuffmanNode\* left, \* right;

HuffmanNode(char data, int frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}

};

// 用于优先队列的比较函数

struct Compare {

bool operator()(HuffmanNode\* a, HuffmanNode\* b) {

return a->frequency > b->frequency;

}

};

// 生成哈夫曼树

HuffmanNode\* buildHuffmanTree(const unordered\_map<char, int>& frequencyMap) {

priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, Compare> minHeap;

for (const auto& pair : frequencyMap) {

minHeap.push(new HuffmanNode(pair.first, pair.second));

}

while (minHeap.size() != 1) {

HuffmanNode\* left = minHeap.top();

minHeap.pop();

HuffmanNode\* right = minHeap.top();

minHeap.pop();

HuffmanNode\* newNode = new HuffmanNode('$', left->frequency + right->frequency);

newNode->left = left;

newNode->right = right;

minHeap.push(newNode);

}

return minHeap.top();

}

// 生成哈夫曼编码

void generateHuffmanCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& huffmanCodes) {

if (root == nullptr) return;

if (root->data != '$') {

huffmanCodes[root->data] = code;

}

generateHuffmanCodes(root->left, code + "0", huffmanCodes);

generateHuffmanCodes(root->right, code + "1", huffmanCodes);

}

// 压缩文件

void compressFile(const string& inputFile, const string& outputFile, const unordered\_map<char, string>& huffmanCodes) {

ifstream inFile(inputFile, ios::in | ios::binary);

ofstream outFile(outputFile, ios::out | ios::binary);

char ch;

string encodedString = "";

while (inFile.get(ch)) {

encodedString += huffmanCodes.at(ch);

}

// 将编码后的字符串写入输出文件

string compressedString = "";

int i = 0;

while (i < encodedString.size()) {

unsigned char byte = 0;

for (int j = 0; j < 8 && i < encodedString.size(); ++j, ++i) {

byte <<= 1;

if (encodedString[i] == '1') {

byte |= 1;

}

}

compressedString += byte;

}

outFile.write(compressedString.c\_str(), compressedString.size());

inFile.close();

outFile.close();

}

int main() {

string inputFile = "input.txt";

string outputFile = "compressed.bin";

// 读取文件并计算字符的权重

ifstream inFile(inputFile, ios::in | ios::binary);

unordered\_map<char, int> frequencyMap;

char ch;

while (inFile.get(ch)) {

frequencyMap[ch]++;

}

inFile.close();

// 生成哈夫曼树

HuffmanNode\* root = buildHuffmanTree(frequencyMap);

// 生成哈夫曼编码

unordered\_map<char, string> huffmanCodes;

generateHuffmanCodes(root, "", huffmanCodes);

// 输出每个字符的哈夫曼编码

for (const auto& pair : huffmanCodes) {

cout << pair.first << ": " << pair.second << endl;

}

// 压缩文件

compressFile(inputFile, outputFile, huffmanCodes);

cout << "文件压缩完成，输出文件为 " << outputFile << endl;

return 0;

}

实验6代码

（1）#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_set>

class Graph {

private:

int V; // 顶点数

std::vector<std::unordered\_set<int>> adj; // 邻接表

public:

Graph(int V) : V(V), adj(V) {}

void addEdge(int u, int v) {

adj[u].insert(v);

adj[v].insert(u);

}

int countEdges() {

int count = 0;

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

count += adj[i].size();

}

return count / 2; // 每条边被计算了两次

}

bool isTree() {

std::vector<bool> visited(V, false);

if (isCyclic(0, -1, visited)) {

return false;

}

for (bool v : visited) {

if (!v) return false; // 如果有顶点未访问到，则不是树

}

return true;

}

bool isCyclic(int v, int parent, std::vector<bool>& visited) {

visited[v] = true;

for (int u : adj[v]) {

if (!visited[u]) {

if (isCyclic(u, v, visited)) {

return true;

}

}

else if (u != parent) {

return true;

}

}

return false;

}

std::vector<int> shortestPath(int src, int dest) {

std::vector<bool> visited(V, false);

std::vector<int> parent(V, -1);

std::queue<int> q;

visited[src] = true;

q.push(src);

while (!q.empty()) {

int u = q.front();

q.pop();

if (u == dest) {

std::vector<int> path;

for (int v = dest; v != -1; v = parent[v]) {

path.push\_back(v);

}

std::reverse(path.begin(), path.end());

return path;

}

for (int v : adj[u]) {

if (!visited[v]) {

visited[v] = true;

parent[v] = u;

q.push(v);

}

}

}

return {}; // 如果没有路径，返回空向量

}

};

int main() {

// 测试案例1：图的边数和是否是树

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(0, 3);

g1.addEdge(1, 4);

std::cout << "Graph 1 has " << g1.countEdges() << " edges." << std::endl;

std::cout << "Graph 1 is " << (g1.isTree() ? "a tree." : "not a tree.") << std::endl;

// 测试案例2：寻找最短路径

Graph g2(6);

g2.addEdge(0, 1);

g2.addEdge(0, 2);

g2.addEdge(1, 3);

g2.addEdge(2, 3);

g2.addEdge(3, 4);

g2.addEdge(4, 5);

std::vector<int> path = g2.shortestPath(0, 5);

std::cout << "Shortest path from 0 to 5: ";

for (int v : path) {

std::cout << v << " ";

}

std::cout << std::endl;

return 0;

}

（2）#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <limits.h>

class Graph {

public:

Graph(int vertices);

void addEdge(int u, int v, int weight);

void findCriticalPath();

private:

int vertices;

std::vector<std::vector<std::pair<int, int>>> adjList;

std::vector<int> topologicalSort();

void longestPath(int start, std::vector<int>& dist, std::vector<int>& pred);

};

Graph::Graph(int vertices) : vertices(vertices) {

adjList.resize(vertices);

}

void Graph::addEdge(int u, int v, int weight) {

adjList[u].push\_back({ v, weight });

}

std::vector<int> Graph::topologicalSort() {

std::vector<int> inDegree(vertices, 0);

for (int u = 0; u < vertices; ++u) {

for (auto& edge : adjList[u]) {

int v = edge.first;

inDegree[v]++;

}

}

std::queue<int> q;

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

if (inDegree[i] == 0) {

q.push(i);

}

}

std::vector<int> topoOrder;

while (!q.empty()) {

int u = q.front();

q.pop();

topoOrder.push\_back(u);

for (auto& edge : adjList[u]) {

int v = edge.first;

if (--inDegree[v] == 0) {

q.push(v);

}

}

}

return topoOrder;

}

void Graph::longestPath(int start, std::vector<int>& dist, std::vector<int>& pred) {

std::vector<int> topoOrder = topologicalSort();

dist.assign(vertices, INT\_MIN);

pred.assign(vertices, -1);

dist[start] = 0;

for (int u : topoOrder) {

if (dist[u] != INT\_MIN) {

for (auto& edge : adjList[u]) {

int v = edge.first;

int weight = edge.second;

if (dist[v] < dist[u] + weight) {

dist[v] = dist[u] + weight;

pred[v] = u;

}

}

}

}

}

void Graph::findCriticalPath() {

std::vector<int> dist, pred;

longestPath(0, dist, pred);

int maxDist = INT\_MIN;

int endVertex = -1;

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

if (dist[i] > maxDist) {

maxDist = dist[i];

endVertex = i;

}

}

std::stack<int> path;

for (int v = endVertex; v != -1; v = pred[v]) {

path.push(v);

}

std::cout << "Critical Path: ";

while (!path.empty()) {

std::cout << path.top() + 1 << " "; // 将每个顶点的值加1以使得点的值与实际顶点号匹配

path.pop();

}

std::cout << "\nLength of Critical Path: " << maxDist << std::endl;

}

int main() {

Graph g(8);

g.addEdge(0, 1, 3);

g.addEdge(0, 2, 4);

g.addEdge(1, 3, 5);

g.addEdge(1, 4, 6);

g.addEdge(2, 3, 8);

g.addEdge(2, 5, 7);

g.addEdge(3, 4, 3);

g.addEdge(4, 6, 9);

g.addEdge(5, 6, 4);

g.addEdge(6, 7, 2);

g.findCriticalPath();

return 0;

}

实验7代码

（1）#include <iostream>

#include <vector>

#include <unordered\_map>

#include <algorithm>

#include <ctime>

#include <cstdlib>

#include <chrono>

class SearchAlgorithms {

public:

// 基于顺序表的查找算法

static int linearSearch(const std::vector<int>& arr, int key) {

for (int i = 0; i < arr.size(); ++i) {

if (arr[i] == key) {

return i;

}

}

return -1;

}

static int binarySearch(const std::vector<int>& arr, int key) {

int left = 0, right = arr.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == key) {

return mid;

}

else if (arr[mid] < key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1;

}

// 基于树表的查找算法

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

static TreeNode\* insertBST(TreeNode\* root, int key) {

if (root == nullptr) {

return new TreeNode(key);

}

if (key < root->val) {

root->left = insertBST(root->left, key);

}

else {

root->right = insertBST(root->right, key);

}

return root;

}

static TreeNode\* searchBST(TreeNode\* root, int key) {

if (root == nullptr || root->val == key) {

return root;

}

if (key < root->val) {

return searchBST(root->left, key);

}

else {

return searchBST(root->right, key);

}

}

// 基于散列表的查找算法

static int hashTableSearch(const std::unordered\_map<int, int>& hashTable, int key) {

auto it = hashTable.find(key);

if (it != hashTable.end()) {

return it->second;

}

return -1;

}

static int hashTableSearch2(const std::vector<int>& arr, int key) {

std::unordered\_map<int, int> hashTable;

for (int i = 0; i < arr.size(); ++i) {

hashTable[arr[i]] = i;

}

return hashTableSearch(hashTable, key);

}

static std::vector<int> generateRandomData(int N) {

std::vector<int> data(N);

std::srand(std::time(0));

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

data[i] = std::rand() % (N \* 10);

}

return data;

}

static void timeComparison(int N) {

std::vector<int> data = generateRandomData(N);

int key = data[std::rand() % N];

auto start = std::chrono::high\_resolution\_clock::now();

linearSearch(data, key);

auto end = std::chrono::high\_resolution\_clock::now();

std::cout << "Linear Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";

std::vector<int> sortedData = data; // 创建一个副本进行排序

std::sort(sortedData.begin(), sortedData.end());

start = std::chrono::high\_resolution\_clock::now();

binarySearch(sortedData, key);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Binary Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";

TreeNode\* root = nullptr;

for (int num : data) {

root = insertBST(root, num);

}

start = std::chrono::high\_resolution\_clock::now();

searchBST(root, key);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Binary Search Tree Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";

std::unordered\_map<int, int> hashTable;

for (int i = 0; i < data.size(); ++i) {

hashTable[data[i]] = i;

}

start = std::chrono::high\_resolution\_clock::now();

hashTableSearch(hashTable, key);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Hash Table Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";

start = std::chrono::high\_resolution\_clock::now();

hashTableSearch2(data, key);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Hash Table Search 2 Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";

}

};

int main() {

int N = 100000; // 可以根据需要调整数据规模

SearchAlgorithms::timeComparison(N);

return 0;

}

（2）#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <cstdlib>

#include <queue>

#include <list>

class Sorting {

public:

// 顺序表排序：快速排序

void quickSort(std::vector<int>& arr, int left, int right) {

if (left >= right) return;

int i = left, j = right;

int pivot = arr[left + (right - left) / 2];

while (i <= j) {

while (arr[i] < pivot) i++;

while (arr[j] > pivot) j--;

if (i <= j) {

std::swap(arr[i], arr[j]);

i++;

j--;

}

}

quickSort(arr, left, j);

quickSort(arr, i, right);

}

// 树表排序：堆排序

void heapSort(std::vector<int>& arr) {

std::priority\_queue<int, std::vector<int>, std::greater<int>> minHeap;

for (int num : arr) {

minHeap.push(num);

}

for (int i = 0; i < arr.size(); ++i) {

arr[i] = minHeap.top();

minHeap.pop();

}

}

// 链式基数排序

void radixSort(std::vector<int>& arr) {

const int base = 10;

std::vector<std::list<int>> buckets(base);

int maxVal = \*std::max\_element(arr.begin(), arr.end());

for (int exp = 1; maxVal / exp > 0; exp \*= base) {

for (int num : arr) {

buckets[(num / exp) % base].push\_back(num);

}

int i = 0;

for (auto& bucket : buckets) {

for (int num : bucket) {

arr[i++] = num;

}

bucket.clear();

}

}

}

// 生成随机数据

std::vector<int> generateRandomData(int size) {

std::vector<int> data(size);

std::srand(std::time(0));

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

data[i] = std::rand() % 100000;

}

return data;

}

// 测试排序算法性能

void testSortingAlgorithms(int N) {

std::vector<int> data = generateRandomData(N);

std::vector<int> dataCopy = data;

// 快速排序

clock\_t start = clock();

quickSort(dataCopy, 0, dataCopy.size() - 1);

clock\_t end = clock();

std::cout << "Quick Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";

dataCopy = data;

// 堆排序

start = clock();

heapSort(dataCopy);

end = clock();

std::cout << "Heap Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";

dataCopy = data;

// 基数排序

start = clock();

radixSort(dataCopy);

end = clock();

std::cout << "Radix Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";

}

};

int main() {

Sorting sorting;

sorting.testSortingAlgorithms(10000);

return 0;

}