C++ 期中上机复习

2024 年春季学期

知识点分布

1. 链表专题 🔗

- 基础操作
  - 创建与删除
  - 插入与遍历
- 进阶技巧
  - 链表逆置
  - 环检测算法
  - 链表合并

2. 排序专题 

- 结构体排序
  - 运算符重载
  - 多级排序
- 算法优化
  - 快速排序改进
  - 特殊情况处理

课程目标

基础能力

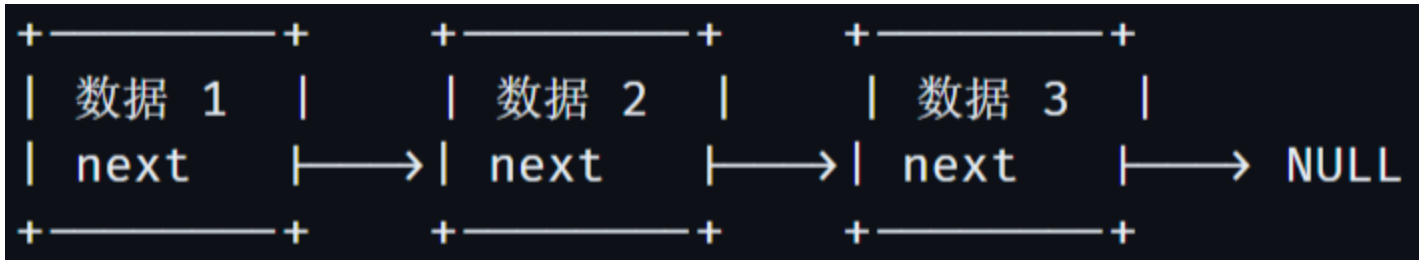
1. 掌握基本数据结构
2. 理解算法设计思想
3. 培养代码实现能力

进阶要求

1. 优化算法性能
2. 提高代码质量
3. 增强问题分析能力

第一部分：链表专题

什么是链表?



特点

- 动态分配内存
- 非连续存储
- 插入删除高效
- 随机访问较慢

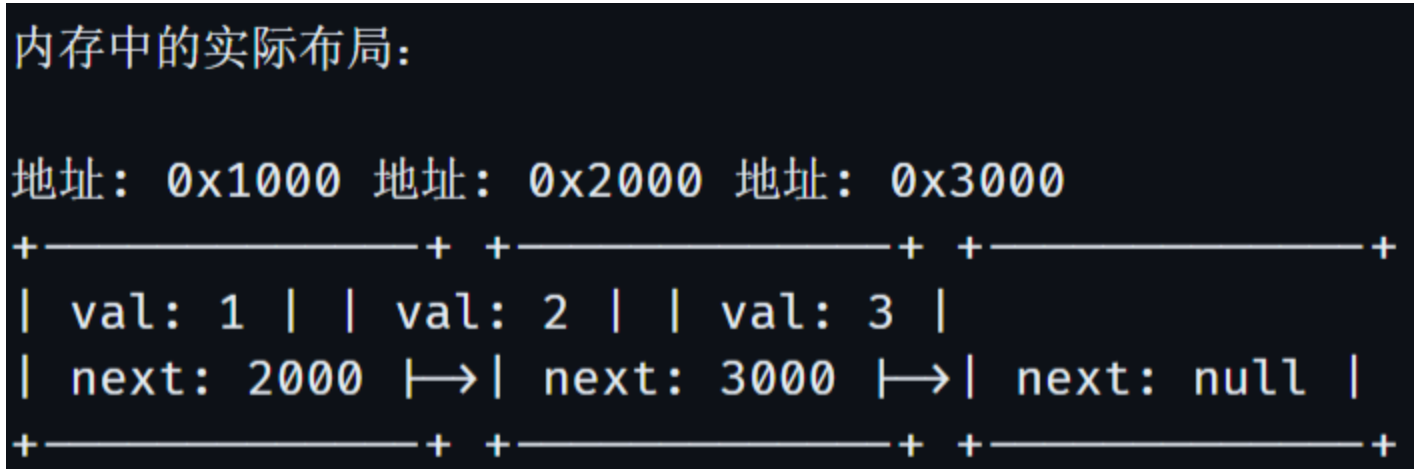
链表节点的设计

```
struct ListNode {
    int val;             // 节点值
    ListNode* next;      // 指向下一个节点的指针
    ListNode(int x = 0) : val(x), next(nullptr) {}
};
```

成员解析

- `val`：存储节点的数据
- `next`：指向下一个节点的指针
- 构造函数：初始化节点值和指针

链表的内存布局



要点说明

- 节点可以分散在内存各处
- 通过指针连接各个节点
- 最后一个节点指向 nullptr

链表创建：问题分析

要求

- 从标准输入读取数字，以-1 结尾
- -1 不计入链表
- 创建对应的单向链表

示例

输入：1 2 3 -1
输出：1->2->3

关键点

- 使用虚拟头节点简化操作
- 正确处理尾节点
- 注意内存释放

链表创建：代码实现（第一部分）

```
ListNode* createList() {
    // 创建虚拟头节点
    ListNode* dummy = new ListNode(0);
    ListNode* cur = dummy;
    int x;
```

链表创建：代码实现（第二部分）

```
    // 读入数据并构建链表
    while (cin >> x && x != -1) {
        cur->next = new ListNode(x);
        cur = cur->next;
    }

    // 获取真实头节点并释放虚拟头节点
    ListNode* head = dummy->next;
    delete dummy;
    return head;
}
```

链表创建：图解步骤

```
步骤1：创建虚拟头节点
[dummy(0)] -> null

步骤2：读入1
[dummy(0)] -> [1] -> null

步骤3：读入2
[dummy(0)] -> [1] -> [2] -> null

步骤4：读入3
[dummy(0)] -> [1] -> [2] -> [3] -> null

步骤5：返回结果
[1] -> [2] -> [3] -> null
```

链表逆置：问题分析

问题描述

将链表 1->2->3 转变为 3->2->1

解决方案

1. 迭代法
    - 使用三个指针
    - 逐步改变指针方向
2. 递归法
    - 利用递归栈
    - 从后向前处理

链表逆置：迭代法图解

初始状态：

```
1 -> 2 -> 3 -> null
^    ^
cur  next
```

第一步：

```
null <- 1    2 -> 3
       ^     ^
       cur   next
```

第二步：

```
null <- 1 <- 2    3
            ^     ^
            cur   next
```

最终状态：

```
null <- 1 <- 2 <- 3
```

链表逆置：迭代实现

```
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr;   // 前一个节点
    ListNode *cur = head;       // 当前节点

    while (cur) {
        ListNode* next = cur->next;  // 保存下一个节点
        cur->next = prev;            // 反转指针
        prev = cur;                  // 移动prev
        cur = next;                  // 移动cur
    }
    return prev;
}
```

链表逆置：递归分析

递归思路

1. 先递归到链表末尾
2. 反转子链表
3. 处理当前节点
4. 返回新的头节点

关键点

- 基准情况处理
- 避免环形引用
- 正确返回头节点

链表逆置：递归实现

```
ListNode* reverseList(ListNode* head) {
    // 基准情况：空链表或只有一个节点
    if (!head || !head->next) return head;

    // 递归反转后续链表
    ListNode* newHead = reverseList(head->next);

    // 处理当前节点
    head->next->next = head;
    head->next = nullptr;

    return newHead;
}
```

链表加法：问题描述

要求

- 两个链表表示两个数字
- 数字按逆序方式存储
- 计算两数之和

示例

```
输入:
2->4->3 (342)
5->6->4 (465)
输出:
7->0->8 (807)
```

链表加法：解题思路

1. 使用虚拟头节点简化操作
2. 同时遍历两个链表
3. 处理进位情况
4. 处理剩余节点
5. 处理最终进位

链表加法：代码实现（第一部分）

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* dummy = new ListNode(0);
    ListNode* cur = dummy;
    int carry = 0;  // 进位

    while (l1 || l2 || carry) {
        int sum = carry;
```

链表加法：代码实现（第二部分）

```
        // 处理两个链表的节点
        if (l1) {
            sum += l1->val;
            l1 = l1->next;
        }
        if (l2) {
            sum += l2->val;
            l2 = l2->next;
        }

        // 处理进位
        carry = sum / 10;
        cur->next = new ListNode(sum % 10);
        cur = cur->next;
    }
```

链表加法：代码实现（第三部分）

```
    // 获取结果并释放虚拟头节点
    ListNode* result = dummy->next;
    delete dummy;
    return result;
}
```

复杂度分析

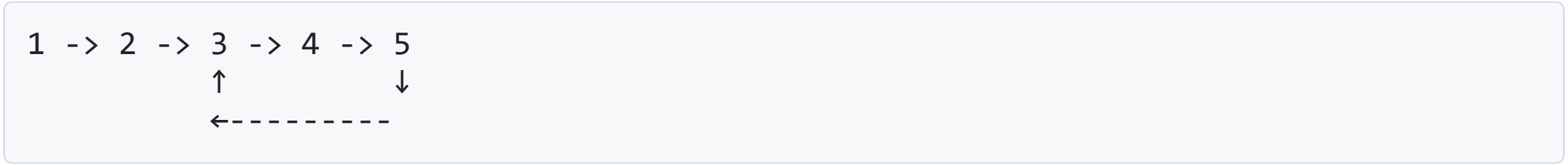
- 时间复杂度：O(max(N,M))
- 空间复杂度：O(max(N,M))
  其中 N,M 为两个链表的长度

链表环检测：问题引入

什么是环形链表?

一个链表中的某个节点的 next 指针指向了链表中的一个前面的节点，形成一个环。

示例



链表环检测：Floyd 算法

算法思想

1. 使用快慢指针
2. 慢指针每次走一步
3. 快指针每次走两步
4. 如果有环，两个指针必然相遇

链表环检测：代码实现

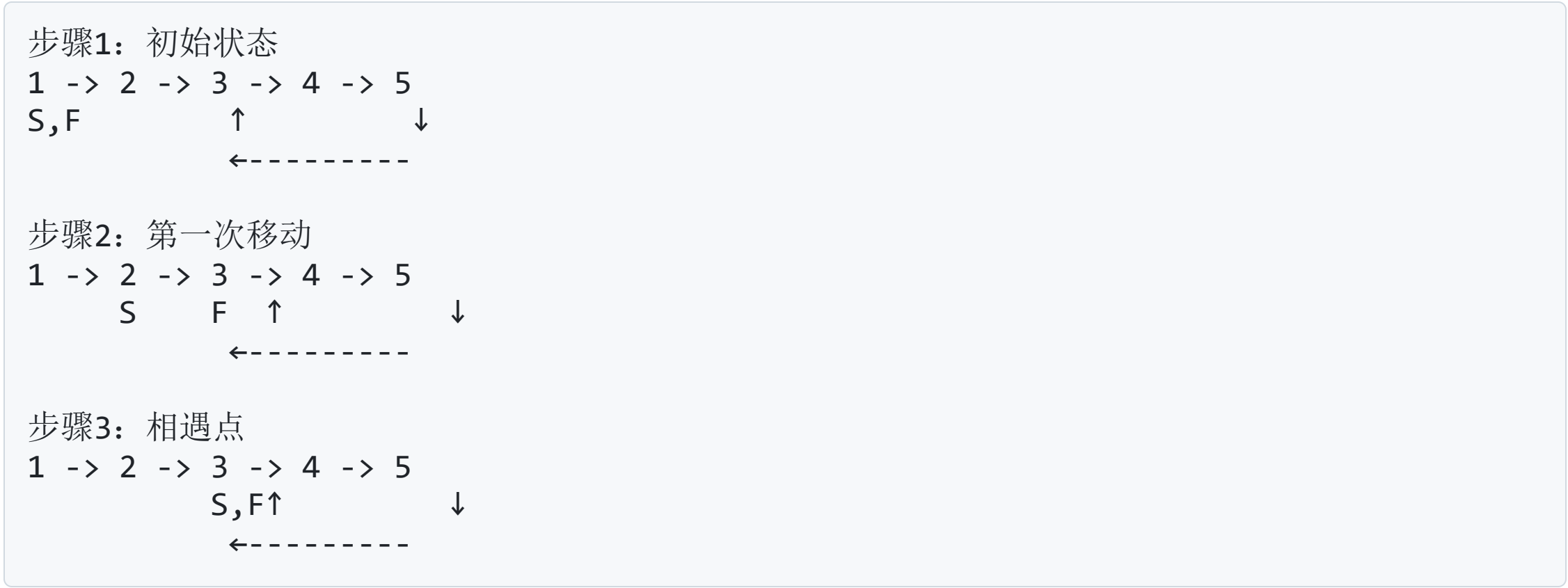
```
bool hasCycle(ListNode* head) {
    if (!head || !head->next) return false;

    ListNode *slow = head;
    ListNode *fast = head;

    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }

    return false;
}
```

链表环检测：图解过程



第二部分：排序专题

结构体排序：基本概念

运算符重载

- 重载小于运算符
- 自定义比较规则
- 支持多重条件

示例场景

- 商品管理系统
- 学生成绩排序
- 文件系统管理

结构体排序：日期结构体

```
struct Date {
    int year, month, day;

    bool operator<(const Date& other) const {
        if (year != other.year) return year < other.year;
        if (month != other.month) return month < other.month;
        return day < other.day;
    }
};
```

结构体排序：商品结构体

```
struct Product {
    string id;           // 商品编号
    string name;         // 商品名称
    double price;        // 价格
    int stock;          // 库存
    Date expiry;        // 过期日期
    string category;     // 商品类别
    double discount;     // 折扣率

    double getDiscountPrice() const {
        return price * discount;
    }
};
```

商品排序规则：第一部分

```
class ProductSorter {
public:
    // 按折后价格升序，同价格按库存降序
    static bool byPriceAndStock(const Product& a,
                                const Product& b) {
        double priceA = a.getDiscountPrice();
        double priceB = b.getDiscountPrice();
        if (abs(priceA - priceB) > 1e-6)
            return priceA < priceB;
        return a.stock > b.stock;
    }
```

商品排序规则：第二部分

```
    // 按过期日期升序，同日期按价格升序
    static bool byExpiryAndPrice(const Product& a,
                                 const Product& b) {
        if (a.expiry < b.expiry) return true;
        if (b.expiry < a.expiry) return false;
        return a.getDiscountPrice() < b.getDiscountPrice();
    }
};
```

快速排序优化：基本思路

优化方向

1. 基准值选择
    - 三数取中法
    - 随机选择
2. 小规模优化
    - 使用插入排序
    - 优化递归
3. 特殊情况
    - 处理重复元素
    - 三路快排

快速排序：辅助函数

```
// 插入排序：用于小规模数组
void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int temp = arr[i];
        int j = i;
        while (j > left && arr[j-1] > temp) {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = temp;
    }
}
```

快速排序：优化实现（第一部分）

```
void quickSort(vector<int>& arr, int left, int right) {
    // 小规模数组使用插入排序
    if (right - left <= 16) {
        insertionSort(arr, left, right);
        return;
    }

    // 三数取中选择基准
    int mid = left + (right - left) / 2;
    if (arr[left] > arr[mid]) swap(arr[left], arr[mid]);
    if (arr[left] > arr[right]) swap(arr[left], arr[right]);
    if (arr[mid] > arr[right]) swap(arr[mid], arr[right]);
```

快速排序：优化实现（第二部分）

```
    // 三路快排实现
    int pivot = arr[mid];
    int lt = left;     // less than
    int gt = right;    // greater than
    int i = left + 1;  // current

    while (i <= gt) {
        if (arr[i] < pivot)
            swap(arr[lt++], arr[i++]);
        else if (arr[i] > pivot)
            swap(arr[i], arr[gt--]);
        else
            i++;
    }

    // 递归处理
    quickSort(arr, left, lt-1);
    quickSort(arr, gt+1, right);
}
```

考试实战技巧

解题步骤

1. 审题阶段

- 仔细阅读题目要求
- 明确输入输出格式
- 找出关键算法和数据结构

2. 设计阶段

- 先写注释设计流程
- 确定核心数据结构
- 规划主要函数接口

代码实现技巧

命名规范

```
// 推荐写法
ListNode* reverseList(ListNode* head)
vector<int> sortedArray;
bool isValidBST(TreeNode* root)

// 不推荐写法
ListNode* reverse(ListNode* h)
vector<int> a;
bool check(TreeNode\* r)
```

边界处理示例

空指针检查

```
ListNode* mergeLists(ListNode* l1, ListNode* l2) {
    // 边界条件处理
    if (!l1) return l2;  // l1为空返回l2
    if (!l2) return l1;  // l2为空返回l1

    // 主要逻辑
    if (l1->val < l2->val) {
        l1->next = mergeLists(l1->next, l2);
        return l1;
    } else {
        l2->next = mergeLists(l1, l2->next);
        return l2;
    }
}
```

调试技巧

输出检查

```
void debugList(ListNode* head) {
    cout << "List: ";
    while (head) {
        cout << head->val << "->";
        head = head->next;
    }
    cout << "null" << endl;
}
```

断言使用

```
#include <cassert>
void checkSortedArray(vector<int>& arr) {
    for (int i = 1; i < arr.size(); i++) {
        assert(arr[i] >= arr[i-1]);  // 确保数组有序
    }
}
```

代码优化示例

优化前

```
bool containsDuplicate(vector<int>& nums) {
    for (int i = 0; i < nums.size(); i++) {
        for (int j = i + 1; j < nums.size(); j++) {
            if (nums[i] == nums[j]) return true;
        }
    }
    return false;
}
```

优化后

```
bool containsDuplicate(vector<int>& nums) {
    unordered_set<int> seen;
    for (int num : nums) {
        if (seen.count(num)) return true;
        seen.insert(num);
    }
```

常见错误避免

1. 内存泄漏

```
// 错误示例
ListNode* createNode() {
    ListNode* node = new ListNode(0);
    if (someCondition) return nullptr;  // 内存泄漏!
    return node;
}

// 正确示例
ListNode* createNode() {
    ListNode* node = new ListNode(0);
    if (someCondition) {
        delete node;  // 释放内存
        return nullptr;
    }
    return node;
}
```

代码风格示例

良好的注释风格

```
ListNode* mergeSortedLists(ListNode* l1, ListNode* l2) {
    // 创建虚拟头节点简化边界处理
    ListNode* dummy = new ListNode(0);
    ListNode* curr = dummy;

    // 同时遍历两个链表
    while (l1 && l2) {
        if (l1->val < l2->val) {
            curr->next = l1;
            l1 = l1->next;
        } else {
            curr->next = l2;
            l2 = l2->next;
        }
        curr = curr->next;
    }

    // 处理剩余节点
    curr->next = l1 ? l1 : l2;

    // 获取结果并释放虚拟头节点
    ListNode* result = dummy->next;
    delete dummy;
```

祝大家考试顺利