

# 学生实验报告册

(人工智能学院)

课程名称：数据结构与算法（双语教学）

课号：2312391

学生学号：2201630216 学生姓名：韦立赞

专业：人工智能 座号：

2023——2024 学年

第 1 学期

## 实验 2 单链表的应用

### 一、实验目的

1. 掌握并运用单链表的基本操作。
2. 理解单链表的应用场景。
3. 实现单链表的插入、排序、合并等功能。

### 二、实验设备

笔记本电脑，代码编辑器，c++编译器

### 三、实验内容及要求

1. 从键盘中输入 5 个无序的整数，插入到单链表 a 中，并保证插入后单链表 a 中的数据要从大到小进行排序。
2. 从键盘中输入 5 个无序的整数，插入到单链表 b 中，并保证插入后单链表 b 中的数据要从小到大进行排序。
3. 输出这两个单链表中的交集和并集。
4. 将单链表 b 倒序。
5. 将这两个有序单链表合并成一个有序单链表（从大到小排序），并将生成的有序单链表输出显示。
6. 复制合并后的链表：例如合并链表为：1 2 3 4 5，复制后的结果是：1 1 2 2 3 3 4 4 5 5

### 四、实验步骤

1. 创建单链表 a 和单链表 b。
2. 从键盘中输入 5 个无序的整数，并将它们插入到单链表 a 中，保证插入后单链表 a 中的数据从大到小排序。
3. 从键盘中输入 5 个无序的整数，并将它们插入到单链表 b 中，保证插入后单链表 b 中的数据从小到大排序。
4. 输出单链表 a 和单链表 b 的内容。
5. 找出单链表 a 和单链表 b 的交集和并集，并输出结果。
6. 将单链表 b 倒序。
7. 输出倒序后的单链表 b 的内容。
8. 将单链表 a 和单链表 b 合并成一个有序单链表，要求合并后的单链表从大到小排序，并输出合并后的结果。
9. 复制合并后的链表，并输出复制后的结果。

代码如下：

```
#include <iostream>

// 定义一个模板结构体，表示链表节点
template<typename T>
struct ListNode {
    T val; // 节点值
    ListNode *next; // 指向下一个节点的指针
    ListNode(T x) : val(x), next(nullptr) {} // 构造函数，初始化节点值和指针
};

// 定义一个模板类，表示链表
template<typename T>
class LinkedList {
public:
    ListNode<T> *head; // 指向链表头节点的指针

    // 构造函数，初始化链表
    LinkedList() : head(nullptr) {}

    // 插入节点的函数
    void insert(T val, bool reverse = false);

    // 打印链表的函数
    void print();

    // 求两个链表的交集的函数
    LinkedList<T> intersection(LinkedList<T> &other);

    // 求两个链表的并集的函数
    LinkedList<T> unionList(LinkedList<T> &other);

    // 判断链表中是否包含某个值的函数
    bool contains(T val);

    // 反转链表的函数
    void reverse();

    // 合并两个有序链表的函数
    LinkedList<T> merge(LinkedList<T> &other);

    // 复制链表的函数
```

```
    LinkedList<T> duplicate();  
};  
  
// 复制链表的函数  
template<typename T>  
LinkedList<T> LinkedList<T>::duplicate() {  
    LinkedList<T> result; // 创建一个新的链表对象  
    ListNode<T> *node = head; // 遍历原链表的指针  
    while (node) {  
        result.insert(node->val); // 将原链表节点的值插入到新链表中两次  
        result.insert(node->val);  
        node = node->next; // 移动到下一个节点  
    }  
    return result; // 返回复制后的链表  
}  
  
// 合并两个有序链表的函数  
template<typename T>  
LinkedList<T> LinkedList<T>::merge(LinkedList<T> &other) {  
    LinkedList<T> result; // 创建一个新的链表对象  
    ListNode<T> *nodeA = head; // 遍历第一个链表的指针  
    ListNode<T> *nodeB = other.head; // 遍历第二个链表的指针  
    while (nodeA && nodeB) {  
        if (nodeA->val > nodeB->val) { // 如果第一个链表节点的值大于第二个链表节点的值  
            result.insert(nodeA->val); // 将第一个链表节点的值插入到新链表中  
            nodeA = nodeA->next; // 移动到下一个节点  
        } else {  
            result.insert(nodeB->val); // 将第二个链表节点的值插入到新链表中  
            nodeB = nodeB->next; // 移动到下一个节点  
        }  
    }  
    while (nodeA) {  
        result.insert(nodeA->val); // 将第一个链表剩余节点的值插入到新链表中  
        nodeA = nodeA->next; // 移动到下一个节点  
    }  
    while (nodeB) {  
        result.insert(nodeB->val); // 将第二个链表剩余节点的值插入到新链表中  
        nodeB = nodeB->next; // 移动到下一个节点  
    }  
    return result; // 返回合并后的链表  
}  
  
// 反转链表的函数
```

```
template<typename T>
void LinkedList<T>::reverse() {
    ListNode<T> *prev = nullptr; // 前一个节点的指针
    ListNode<T> *current = head; // 当前节点的指针
    ListNode<T> *next = nullptr; // 下一个节点的指针
    while (current != nullptr) {
        next = current->next; // 保存下一个节点的指针
        current->next = prev; // 将当前节点的指针指向前一个节点
        prev = current; // 更新前一个节点的指针
        current = next; // 更新当前节点的指针
    }
    head = prev; // 更新链表头节点的指针
}

// 判断链表中是否包含某个值的函数
template<typename T>
bool LinkedList<T>::contains(T val) {
    ListNode<T> *node = head; // 遍历链表的指针
    while (node) {
        if (node->val == val) { // 如果节点的值等于目标值
            return true; // 返回 true
        }
        node = node->next; // 移动到下一个节点
    }
    return false; // 链表中不包含目标值，返回 false
}

// 求两个链表的并集的函数
template<typename T>
LinkedList<T> LinkedList<T>::unionList(LinkedList<T> &other) {
    LinkedList<T> result; // 创建一个新的链表对象
    ListNode<T> *node = head; // 遍历第一个链表的指针
    while (node) {
        result.insert(node->val); // 将第一个链表节点的值插入到新链表中
        node = node->next; // 移动到下一个节点
    }
    node = other.head; // 遍历第二个链表的指针
    while (node) {
        if (!result.contains(node->val)) { // 如果新链表中不包含第二个链表节点的值
            result.insert(node->val); // 将第二个链表节点的值插入到新链表中
        }
        node = node->next; // 移动到下一个节点
    }
}
```

```
        return result; // 返回并集链表
    }

// 求两个链表的交集的函数
template<typename T>
LinkedList<T> LinkedList<T>::intersection(LinkedList<T> &other) {
    LinkedList<T> result; // 创建一个新的链表对象
    ListNode<T> *nodeA = head; // 遍历第一个链表的指针
    while (nodeA) {
        ListNode<T> *nodeB = other.head; // 遍历第二个链表的指针
        bool found = false; // 是否找到交集节点的标志
        while (nodeB) {
            if (nodeA->val == nodeB->val) { // 如果第一个链表节点的值等于第二
个链表节点的值
                found = true; // 设置找到交集节点的标志为 true
                break; // 跳出内层循环
            }
            nodeB = nodeB->next; // 移动到第二个链表的下一个节点
        }
        if (found) {
            result.insert(nodeA->val); // 将交集节点的值插入到新链表中
        }
        nodeA = nodeA->next; // 移动到第一个链表的下一个节点
    }
    return result; // 返回交集链表
}

// 打印链表的函数
template<typename T>
void LinkedList<T>::print() {
    ListNode<T> *node = head; // 遍历链表的指针
    while (node) {
        std::cout << node->val << " "; // 输出节点的值
        node = node->next; // 移动到下一个节点
    }
    std::cout << std::endl; // 输出换行符
}

// 插入节点的函数
template<typename T>
void LinkedList<T>::insert(T val, bool reverse) {
    ListNode<T> *node = new ListNode<T>(val); // 创建一个新的节点
    if (!head || (reverse ? head->val >= node->val : head->val <= node->val))
    {
```

```
        // 如果链表为空或插入的节点值小于等于链表头节点的值（如果 reverse 为 true
        则判断大于等于）
        node->next = head; // 将新节点的指针指向链表头节点
        head = node; // 更新链表头节点的指针为新节点
    } else {
        ListNode<T> *cur = head; // 遍历链表的指针
        while (cur->next && (reverse ? cur->next->val < node->val :
cur->next->val > node->val)) {
            // 遍历链表直到找到插入位置（如果 reverse 为 true 则判断大于）
            cur = cur->next; // 移动到下一个节点
        }
        node->next = cur->next; // 将新节点的指针指向当前节点的下一个节点
        cur->next = node; // 将当前节点的指针指向新节点
    }
}

int main() {
    LinkedList<int> listA; // 创建一个整型链表对象 listA
    LinkedList<int> listB; // 创建一个整型链表对象 listB

    std::cout << "Enter 5 unordered integers for list A: ";
    for (int i = 0; i < 5; ++i) {
        int num;
        std::cin >> num;
        listA.insert(num, true); // 将输入的整数插入到 listA 中，按递减顺序插入
    }

    std::cout << "Enter 5 unordered integers for list B: ";
    for (int i = 0; i < 5; ++i) {
        int num;
        std::cin >> num;
        listB.insert(num); // 将输入的整数插入到 listB 中，按递增顺序插入
    }

    std::cout << "List A: ";
    listA.print(); // 打印 listA

    std::cout << "List B: ";
    listB.print(); // 打印 listB

    LinkedList<int> intersection = listA.intersection(listB); // 求 listA
    和 listB 的交集
    std::cout << "Intersection: ";
```

```
intersection.print(); // 打印交集

LinkedList<int> unionList = listA.unionList(listB); // 求 listA 和
listB 的并集
std::cout << "Union: ";
unionList.print(); // 打印并集

listB.reverse(); // 反转 listB
std::cout << "Reversed list B: ";
listB.print(); // 打印反转后的 listB

LinkedList<int> merged = listA.merge(listB); // 合并 listA 和 listB
std::cout << "Merged list: ";
merged.print(); // 打印合并后的链表

LinkedList<int> duplicated = merged.duplicate(); // 复制合并后的链表
std::cout << "Duplicated list: ";
duplicated.print(); // 打印复制后的链表

return 0;
}
```

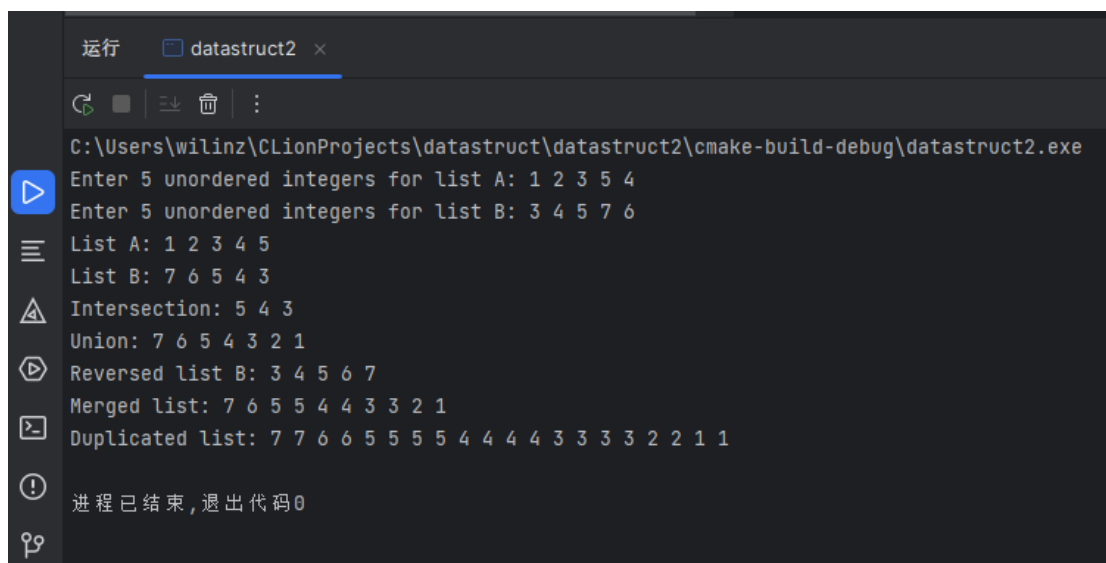
## 五、主要算法描述

1. 定义了一个模板结构体 `ListNode`，表示链表节点。每个节点包含一个值 `val` 和一个指向下一个节点的指针 `next`。
2. `LinkedList` 类包含一个指向链表头节点的指针 `head`，以及各种操作链表的方法。
3. 插入节点（`insert`）：该方法用于将一个新节点插入到链表中。根据参数 `reverse` 的值，决定是按递增还是递减的顺序插入节点。如果链表为空或插入的节点值小于等于链表头节点的值（如果 `reverse` 为 `true` 则判断大于等于），则将新节点作为新的头节点；否则，遍历链表找到合适的位置插入节点。具体实现时，可以使用一个指针从头节点开始遍历链表，找到插入位置的前一个节点，然后将新节点插入到该位置。
4. 打印链表（`print`）：该方法用于打印链表中所有节点的值。遍历链表的节点，依次输出节点的值。
5. 求交集（`intersection`）：该方法用于求两个链表的交集。创建一个新的链表对象 `result` 作为结果链表。遍历第一个链表的节点，对于每个节点，遍历第二个链表的节点，如果找到相同的值，则将该值插入到结果链表 `result` 中。具体实现时，可以使用两个指针分别遍历两个链表，比较节点的值，如果相等则将该值插入到结果链表中。



6. 求并集 (**unionList**)：该方法用于求两个链表的并集。创建一个新的链表对象 **result** 作为结果链表。遍历第一个链表的节点，将每个节点的值插入到结果链表 **result** 中。然后遍历第二个链表的节点，如果结果链表 **result** 中不包含该节点的值，则将该值插入到结果链表中。具体实现时，可以使用两个指针分别遍历两个链表，将节点的值插入到结果链表中，并使用一个集合或哈希表来记录已经插入的值，以避免重复插入。
7. 判断是否包含某个值 (**contains**)：该方法用于判断链表中是否包含某个值。遍历链表的节点，如果找到与目标值相等的节点，则返回 **true**；否则返回 **false**。具体实现时，可以使用一个指针从头节点开始遍历链表，比较节点的值与目标值是否相等。
8. 反转链表 (**reverse**)：该方法用于将链表中的节点顺序反转。使用三个指针 **prev**、**current** 和 **next**，分别指向前一个节点、当前节点和下一个节点。遍历链表的过程中，将当前节点的指针指向前一个节点，然后更新三个指针的位置。具体实现时，可以使用一个指针从头节点开始遍历链表，依次修改节点的指针指向。
9. 合并有序链表 (**merge**)：该方法用于合并两个有序链表。创建一个新的链表对象作为结果链表，同时遍历两个链表的节点。如果第一个链表节点的值大于第二个链表节点的值，则将第一个链表节点的值插入到结果链表中，并移动第一个链表的指针；否则，将第二个链表节点的值插入到结果链表中，并移动第二个链表的指针。最后，将剩余的节点插入到结果链表中。具体实现时，可以使用两个指针分别遍历两个有序链表，比较节点的值，将较小的值插入到结果链表中，并移动相应的指针。
10. 复制链表 (**duplicate**)：该方法用于复制链表。创建一个新的链表对象作为结果链表，遍历原链表的节点，将每个节点的值插入到结果链表中两次，然后移动到下一个节点。最后返回复制后的链表。具体实现时，可以使用一个指针从头节点开始遍历原链表，将每个节点的值插入到结果链表中两次，并移动相应的指针。
11. 主函数 (**main**)：在主函数中，创建两个整型链表对象 **listA** 和 **listB**，分别输入 5 个无序的整数，并按照要求进行插入。然后调用打印链表的方法，分别打印链表 **listA** 和 **listB** 的值。接着调用求交集和求并集的方法，得到结果链表，并打印结果。然后调用反转链表的方法，得到反转后的链表，并打印。最后调用合并链表和复制链表的方法，得到结果链表，并打印。

## 六、实验结果分析与总结



The screenshot shows a terminal window titled "运行" (Run) for a file named "datastruct2". The command executed is "C:\Users\wilinz\CLionProjects\datastruct\datastruct2\cmake-build-debug\datastruct2.exe". The program prompts for two lists of 5 unordered integers. List A is "1 2 3 5 4" and List B is "3 4 5 7 6". The program then displays the following results: List A sorted as "1 2 3 4 5", List B sorted as "7 6 5 4 3", their intersection as "5 4 3", their union as "7 6 5 4 3 2 1", List B reversed as "3 4 5 6 7", the merged list as "7 6 5 5 4 4 3 3 2 1", and the duplicated list as "7 7 6 6 5 5 5 5 4 4 4 4 3 3 3 3 2 2 1 1". The process ends with the message "进程已结束,退出代码0".

```
运行 datastruct2 x
C:\Users\wilinz\CLionProjects\datastruct\datastruct2\cmake-build-debug\datastruct2.exe
Enter 5 unordered integers for list A: 1 2 3 5 4
Enter 5 unordered integers for list B: 3 4 5 7 6
List A: 1 2 3 4 5
List B: 7 6 5 4 3
Intersection: 5 4 3
Union: 7 6 5 4 3 2 1
Reversed list B: 3 4 5 6 7
Merged list: 7 6 5 5 4 4 3 3 2 1
Duplicated list: 7 7 6 6 5 5 5 5 4 4 4 4 3 3 3 3 2 2 1 1
进程已结束,退出代码0
```

根据实验结果，单链表的应用实验完成了以下操作：

1. 输入的无序整数列表插入到单链表 A 后，单链表 A 的数据从大到小排序为：  
1 2 3 4 5。
2. 输入的无序整数列表插入到单链表 B 后，单链表 B 的数据从小到大排序为：  
7 6 5 4 3。
3. 单链表 A 和单链表 B 的交集为：5 4 3。
4. 单链表 A 和单链表 B 的并集为：7 6 5 4 3 2 1。
5. 单链表 B 的倒序为：3 4 5 6 7。
6. 将单链表 A 和单链表 B 合并成一个有序单链表，从大到小排序为：7 6 5 5 4 4  
3 3 2 1。
7. 复制合并后的链表，结果为：7 7 6 6 5 5 5 5 4 4 4 4 3 3 3 3 2 2 1 1。

通过这次实验，加深了对单链表的了解和使用。