

中山大学计算机学院本科生实验报告

(2025 学年第 1 学期)

课程名称：数据结构与算法实验

任课教师：张子臻

| | | | |
|------|-------------|---------|---------------------|
| 年级 | 2024 级 | 专业 (方向) | 计算机科学与技术 (人工智能与大数据) |
| 学号 | 242325157 | 姓名 | 梁玮麟 |
| 电话 | 18620062182 | Email | 3371676041@qq.com |
| 开始日期 | 2025.9.24 | 结束日期 | 2025.9.28 |

第一题

1、实验题目

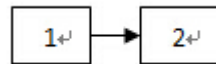
z3-Delete Duplicate

题目描述 已知线性表中的元素以递增有序排列，并以单链表作存储结构。试写一个高效的算法，删除表中所有冗余的结点，即数据域相同的结点只保留一个。



进行操作，将得到：

链表结点定义如下：



```
struct LinkNode {  
    int data;  
    LinkNode *next;  
    LinkNode(int d, LinkNode *add_on = NULL) {  
        data = d;  
        next = add_on;  
    }  
};  
typedef LinkNode *LinkList;
```

请实现函数：

```
void delete_duplicate(LinkList &head);
```

注意内存的回收

Hint 只需提交 `delete_duplicate()` 函数

请记得将头文件包含进去，即 `#include "LinkNode.h"`

2、实验目的

完成删除重复节点的操作，并加深对链表的理解。

3、算法设计

设计思路如下：

1. 有很多方法可以完成。比较通用的方法就是用一个哈希表记录每个节点出现次数，如果遍历链表的时候有相同节点，则删掉当前节点；如果没有，则增加节点。算法空间复杂度为 $O(n)$ ，时间复杂度为 $O(n)$ 。
2. 题目要求高效的算法，通解显然不是最优解。考虑到链表是数据依次递增的结构，我采用以下方法：
 1. 从头节点遍历链表，并记录 head 节点为指针 prev；
 2. 用指针 t 记录当前节点，如果 $t \rightarrow data == prev \rightarrow data$ 则删除当前节点，否则让 prev 指针指向 t 指针指向的节点
 3. 如果 $t \rightarrow next$ 不为空，则让 t 指向 $t \rightarrow next$ ，否则说明链表已经遍历完毕，结束循环。

细节注意:

- 如果只有一个节点，需要直接返回，否则 $head \rightarrow next$ 会报错。
- 释放内存时需要先用一个指针指向 $t \rightarrow next$ ，否则释放内存后再用 $t = t \rightarrow next$ 会报错。

具体实现:

```
//z3-Delete Duplicate
#include <iostream>
#include "LinkNode.h"
#include <map>
using namespace std;
void delete_duplicate(LinkList &head) {
    if (head == NULL){//头节点为空需要直接返回，否则 line 9 会报错
        return;
    }
    LinkList t = head->next;
    LinkList prev = head;
    while (t) {
        if (prev->data == t->data) {
            prev->next = t->next;
            LinkList temp=t->next;//用 temp 记录 t 指向的下一个节点
            //否则 delete 之后 line 16 会报错
            delete t;
            t = temp;
        } else {
            prev = t;
            t = t->next;
        }
    }
}
```

4、程序运行与测试

运行结果:

- 标准输入:
- 实际输出:

```
1
1 2
1
1 2 3
1 2 3 10
```

- 期望输出：

```
1
1 2
1
1 2 3
1 2 3 10
```

5、实验总结与心得

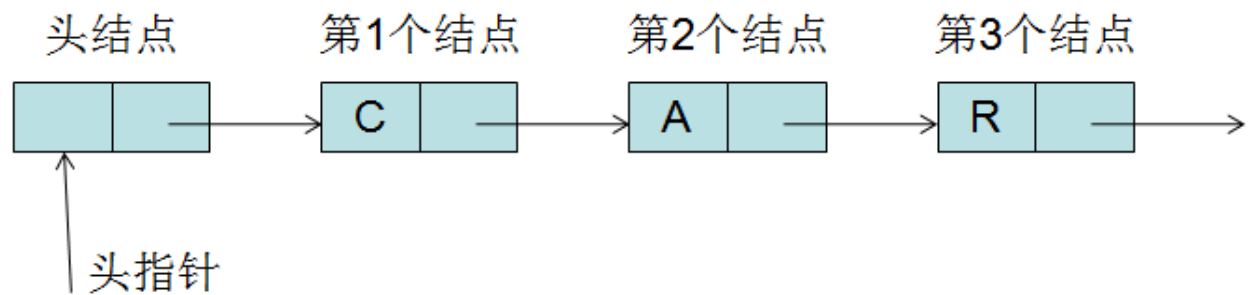
- 这一题的关键在于充分利用给出的链表具有升序结构的特性，时间复杂度与通解一样，但是空间复杂度为 $O(1)$ 。

第二题

1、实验题目

z3-Insert for single link list with head node

题目描述



带虚拟头结点的单链表结点结构如下：

```
struct ListNode
{
    int data;
    ListNode *next;
};
```

链表类接口如下：

```
class List
{
public:
    List()
    {
        head = new ListNode;
        head->next = NULL;
    }
    ~List()
    {
        ListNode* curNode;
        while( head->next )
        {
            curNode = head->next;
        }
    }
};
```

```

        head->next = curNode->next;
        delete curNode;
    }
    delete head;
}

//在链表第 pos(pos>0) 个结点之前插入新结点, 新结点的值为 toadd
//链表实际结点从 1 开始计数。
//调用时需保证 pos 小等于链表实际结点数
void List::insert(int toadd, int pos);

// Data field
ListNode *head; //head 指向虚拟头结点, head-next 指向第一个实际结点
};

```

请实现如下函数：

```
void List::insert(int toadd, int pos)
```

Hint 只提交 insert 函数实现，不需要提交类定义及 main 函数。

请记得将头文件包含进去，即 #include "ListNode.h"

2、实验目的

完成 insert 函数，加深对链表结构的理解。

3、算法设计

设计思路如下： 因为是单向链表，而且已经给出了头指针 head 指针和插入位置 pos，直接找到对应的位置的前一个位置，然后插入就行。

1. 初始化一个指针 t 指向头节点。
2. 用 for 循环，用 t 遍历链表并指向对应的节点的前一个节点。
3. 用指针 next 指向 t 的下一个节点。
4. 新开一个指针 cur 并分配内存，并给这个节点的 data 赋值。
5. 操作 t, next, cur 三个节点，让 t->next 指向 cur，cur->prev 指向 t，next->prev 指向 cur，cur->next 指向 next，完成插入操作。

细节注意：

- for 循环中，要设置成 for(int i=1; i<pos; i++)，因为要找到 pos 的前一个位置，如果设置成 int i=0 就会找到了 next 指针指向的位置，而且可能直接指向结尾的 nullptr。

具体实现：

```

//z3-Insert for single link list with head node
#include<iostream>
#include"ListNode.h"
using namespace std;
void List::insert(int toadd, int pos){
    ListNode* t=head; //指向头节点
    for(int i =1; i<pos; i++){
        t=t->next;
    }
    ListNode* next=t->next;
    ListNode* cur=new ListNode;

```

```

    cur->data=toadd;
    t->next=cur;
    cur->next=next;
}

```

4、程序运行与测试

运行结果：

- 标准输入：

- 实际输出：

```
1 2 3 0 4 5 6 7 8 9 10 11 11 10 9 8 7 6 5 4 3 2 1
```

- 期望输出：

```
1 2 3 0 4 5 6 7 8 9 10 11 11 10 9 8 7 6 5 4 3 2 1
```

5、实验总结与心得

- 这一题要实现的也是链表的基本操作，实现难度并不高。只要理解好链表的结构就能顺利完成。

第三题

1、实验题目

z3-Loops in the Linked List

题目描述 给定链表结点类型定义，要求实现 check 函数返回对于给定链表是否存在环，注意链表的边界情况。

```

struct node{
    node *next;
    int val;
};

// check if there exists a loop
bool check(node *head){
}

```

Hint 只需要提交 check 函数实现，不需要提交 main 函数

请记得将头文件包含进去，即 #include "Node.h"

2、实验目的

通过完成习题，掌握快慢指针遍历链表判断是否有环的经典算法，加深对链表的理解。

3、算法设计

设计思路如下：

1. 首先判断链表是否为空或者只有一个节点，如果只有一个节点则直接返回 false。
2. 设置快慢指针 fast 和 slow，并让 slow 指向 head，fast 指向 head->next。

3. 将 fast 的速度设置为 2, 将 slow 的速度设置为 1。如果有环, 假设环中有 n 个数, 那么 slow 第一次走完所有未走过的节点 (记作 tail) 所需次数为

$$n - 1$$

, 而 fast 从开始的位置第二次遍历到 tail 所需次数为

$$(n + n - 2) / 2 = n - 1$$

。所以无论环中有奇数个还是偶数个元素, 保证两个指针最终都会碰到。

4. 用一个 while 循环:
 1. 首先判断 fast 是否等于 slow, 如果相等直接返回 true。
 2. 如果 fast, slow 都不为空, 则继续循环:
 - 1. slow 直接赋值为 slow->next;
 - 2. 判断 fast->next 是否为空:
 - 如果是, 则直接返回 false;
 - 否则 false=false->next。
 3. 否则直接跳出循环, 返回 false(因为遇到了链表的末尾如果是环, 没有 nullptr 的结尾)。

细节注意:

- 在一开始判断是否空链表或者只有一个节点的时候, 需要把空链表的判断前置, 否则链表为空时 head->next 会报错
- 对每一次循环结束时, 记得要判断 fast->next 是否为空, 否则如果不是环, fast 正好处于尾节点的时候会报错。

具体实现:

```
//z3-Loops in the Linked List
#include <iostream>
#include "Node.h"
using namespace std;
bool check(node *head) {
    if (!head || !head->next) //不能反了
        return false;
    node *fast = head->next;
    node *slow = head;
    while (fast && slow) {
        if (slow == fast)
            return true;
        slow = slow->next;
        if (fast->next) { //防止报错
            fast = fast->next->next;
        } else
            return false;
    }
    return false;
}
```

4、程序运行与测试

运行结果:

数据点 0

- 标准输入:

8

14349 1

877 23486 3504 25047 5984 20729 1279 14378 7136 30797 3787 16558 8414 23792 6211 18276 18345 19213 2783

- 实际输出：

YES

YES

NO

NO

NO

NO

NO

YES

- 期望输出：

YES

YES

NO

NO

NO

NO

NO

YES

数据点 1

- 标准输入：

5

0 0

1 0

1

2 1

1 1

2 0

1 2

3 1

1 2 1

- 实际输出：

NO

NO

YES

NO

YES

- 期望输出：

NO

NO

YES

NO

YES

5、实验总结与心得

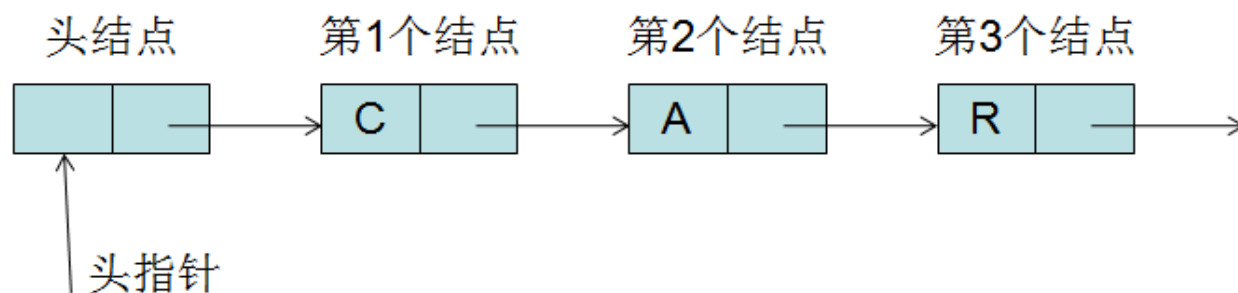
- 这一题学到真东西了，第一次见到快慢指针的写法，设计的很巧妙，完全利用了环的特点。
- 另外，我的算法还可以优化。在对 `fast` 进行更新的时候，判断 `fast->next` 是否为空时，我还应该判断 `fast->next` 是否等于 `slow`，让程序更高效。

第四题

1、实验题目

z3-Remove for single link list with head node

题目描述



带虚拟头结点的单链表结点结构如下：

```
struct ListNode
{
    int data;
    ListNode *next;
};

class List
{
public:
    List()
    {
        head = new ListNode;
        head->next = NULL;
    }

    ~List()
    {
        ListNode* curNode;
        while( head->next )
        {
            curNode = head->next;
            head->next = curNode->next;
            delete curNode;
        }
        delete head;
    }

    //在链表第 pos(pos>0) 个结点之前插入新结点，新结点值为 toadd
    //链表实际结点从 1 开始计数。
    //调用时需保证 pos 小等于链表实际结点数
};
```



```

void insert(int toadd, int pos)

//删除链表的第 pos(pos>0) 个结点
//链表实际结点从 1 开始计数。
//调用时需保证 pos 小等于链表实际结点数
void remove(int pos);

// Data field
ListNode *head; //head 指向虚拟头结点, head-next 指向第一个实际结点
};

```

请实现如下函数：

```
void List::remove(int pos)
```

Hint 只提交 remove 函数实现，不要提交类定义及 main 函数。

请记得将头文件包含进去，即 #include "ListNode.h"

2、实验目的

通过完成习题，知道从链表中移除节点的操作步骤，加深对链表的理解。

3、算法设计

设计思路如下： 因为是单向链表，而且已经给出了头指针 head 指针和插入位置 pos，直接找到对应的位置的前一个位置，然后删除就行。

1. 初始化一个指针 t 指向头节点。
2. 用 for 循环，用 t 遍历链表并指向对应的节点的前一个节点。
3. 用指针 cur 指向 t 的下一个节点。
4. 用指针 next 指向 cur 的下一个节点。
5. 操作 t,next,cur 三个节点，让 t->next 指向 next，释放 cur 指针指向地址的内存，完成删除操作。

具体实现：

```

//z3-Remove for single link list with head node
#include<iostream>
#include"ListNode.h"
using namespace std;
void List::remove(int pos){
    ListNode* t=head;
    for (int i=1;i<pos;i++){
        t=t->next;
    }
    ListNode* cur=t->next;
    ListNode* next=cur->next;
    t->next = next;
    delete cur;
}

```

4、程序运行与测试

运行结果：

- 标准输入：

- 实际输出：

11 10 8 6 4 2 1

- 期望输出：

11 10 8 6 4 2 1

5、实验总结与心得

- 这一题难度比较低，找到对应的位置，然后直接操作即可。

总结

- 总体都是完成链表的基本操作函数。第三题的快慢指针让我积累了一种新的做题思路。收获还是挺多的。