

# Data Structures and Algorithms

## Linear List3 (线性表3)

## ◆ Definition(定义)

◆ **定义**：线性表是 $n$  ( $n$ 大于等于0) 个具有**相同特性的数据元素**的有限序列

## ◆ Characteristics(特点)

- ◆ 存在**唯一**一个被称为**第一个**的数据元素
- ◆ 存在**唯一**一个被称为**最后一个**的数据元素
- ◆ 除第一个以外，集合中每个数据元素均只有一个前驱
- ◆ 除最后一个以外，集合中每个数据元素均只有一个后继

简单来说就是：1. 有限 2. 序列 3. 同构

## ◆ 逻辑结构



## ◆ Linear List 物理结构

### ◆ 顺序存储结构：顺序表（Sequential List）

◆ 定义：用一组地址连续的存储单元存放一个线性表叫做顺序表

◆ 元素地址计算方法：

◆  $LOC(a_i) = LOC(a_1) + (i-1)*L$

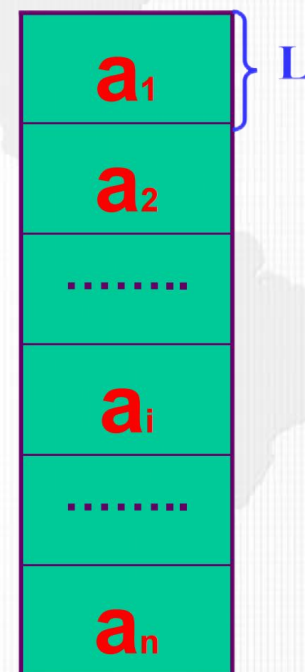
◆  $LOC(a_{i+1}) = LOC(a_i) + L$

◆ 特点：

◆ 实现逻辑上相邻（通过物理地址相邻）

◆ 实现随机存取（random access）

◆ 实现：可用C语言的一维数据实现



## ◆ Linear List 顺序存储结构（顺序表）操作的时间复杂度

- ◆ 插入一个数据元素的时间复杂度为 $O(n)$
- ◆ 删除一个数据元素的时间复杂度为 $O(n)$
- ◆ 读取一个数据元素的时间复杂度为 $O(1)$
- ◆ 修改一个数据元素的时间复杂度为 $O(1)$

## ◆ 顺序存储的优缺点

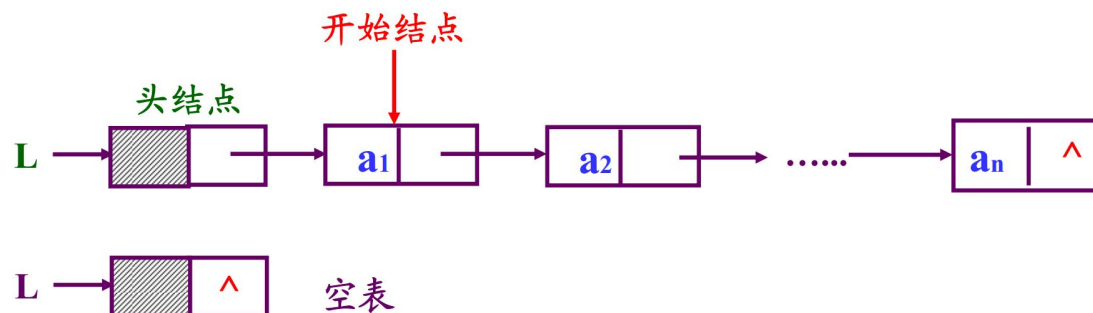
- ◆ 优点
  - ◆ 逻辑相邻，物理相邻
  - ◆ 可随机存取任意元素
  - ◆ 存储空间使用紧凑
- ◆ 缺点
  - ◆ 插入删除操作需要移动大量元素
  - ◆ 需要预先确定数据元素的最大个数

## ◆ Single Linked List (单链表) :

- ◆ Head pointer: first element of the linked (头指针, 指向链表第一个节点)
- ◆ 0: NULL pointer (空指针, 也表示为NULL或“^”)
- ◆ Head not (头结点, 记录线性表的某些性质信息)

## ◆ 设置头节点的优点

- ◆ 设置头结点, 头指针指向头结点, 不论链表是否为空, 头指针总是**非空**
- ◆ 头结点的设置使得对链表的开始结点的操作与对表中其他结点的操作一致 (均在某一结点之后)





## ◆ 单链表的操作：

◆ 查找：查找单链表中的是否存在节点的数值等于e

◆ 查找操作何时结束？

- `p->data == e`
- `p == NULL`

◆ 插入：已知带头节点单链表L，在第i个元素之前插入新的元素e

◆ 插入操作

- `s->next_ = p->next_`
- `p->next_ = s`

◆ 删除：已知带头节点单链表L，删除第i个节点

◆ 删除操作

- `p -> next_ = p -> next_ -> next;`

◆ 链式存储线性表时，不需要使用地址连续的存储单元，即不要求逻辑上相邻的元素在物理位置上也相邻，插入和删除操作不需要移动元素，而只需修改指针，但也会失去顺序表可随机存取的优点。

## ◆ 顺序表和链表的具体区别

### ◆ 读取（存取）方式

- 顺序表可以顺序存取，也可以随机存取，链表只能从表头顺序存取元素

### ◆ 逻辑结构与物理结构

- 顺序存储逻辑上相邻的元素，对应的物理存储位置也相邻。
- 链式存储逻辑上相邻的元素，物理存储位置不一定相邻，对应的逻辑关系是通过指针链接来表示。

## ◆ Pre Definition for singly-linked list

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7  */
8
9
```



## ◆ 练习1

给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能存储一位数字。请你将两个数相加，并以相同形式返回一个表示和的链表。你可以假设除了数字 0 之外，这两个数都不会以 0 开头

test case1:

输入:  $l1 = [2, 4, 3]$   $l2 = [5, 6, 4]$

输出:  $[7, 0, 8]$

test case2:

输入:  $l1 = [0]$   $l2 = [0]$

输出:  $[0]$

test case3:

输入:  $l1 = [9, 9, 9, 9, 9, 9, 9]$   $l2 = [9, 9, 9, 9]$

输出:  $[8, 9, 9, 9, 0, 0, 0, 0, 1]$

## ◆ 练习1

给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能存储一位数字。**请你将两个数相加，并以相同形式返回一个表示和的链表。**你可以假设除了数字 0 之外，这两个数都不会以 0 开头

1. 将两个链表相加是同时对两个链表进行遍历，如果一个链表较短则在前面补0，比如：987 + 23 = 987 + 023 = 1010
2. 每一位计算的同时需要考虑上一位的进位问题，而当前计算结束后同样需要更新进位值
3. 如果两个链表全部遍历完毕后，进位值位1，则在新链表最前方添加节点1

**tips:** 对于链表问题，返回结果为头节点时候，通常需要先初始化一个预定义的指针pre，该指针的下一个节点指向真正的头节点head。使用预定义指针的目的在于链表初始化时无可用节点值，而且链表构造过程需要指针移动，进而导致头指针丢失，无法返回结果。

## ◆ 练习1

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8
9
10 struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2){
11     struct ListNode* result = malloc(sizeof(struct ListNode));
12     struct ListNode* current = result;
13
14     int carry = 0;
15     while(l1 || l2 || carry){
16         if (l1) {
17             carry += l1->val;
18             l1 = l1->next;
19         }
20         if (l2) {
21             carry += l2->val;
22             l2 = l2->next;
23         }
24         current->next = malloc(sizeof(struct ListNode));
25         current->next->val = carry % 10;
26         current->next->next = NULL;
27         current = current->next;
28         carry /= 10;
29     }
30     return result->next;
31 }
```

时间复杂度?

空间复杂度?

## ◆ 练习2

给你单链表的头节点head，请你反转链表，并返回反转后的链表

test case1:

输入: head = [1, 2, 3, 4, 5]

输出: head = [5, 4, 3, 2, 1]

test case2:

输入: head = [1, 2]

输出: head = [2, 1]

test case3:

输入: head = []

输出: head = []

## ◆ 练习2

给你单链表的头节点head，请你反转链表，并返回反转后的链表  
方法一：

1. 定义两个指针，prev和current；prev在前，current在后
2. 每次让prev的next指向current，实现一次局部的反转
3. 局部反转完成之后，prev和current同时往前移动一个位置。
4. 循环上述过程，直至prev到达链表尾部。



## ◆ 练习2

给你单链表的头节点head，请你反转链表，并返回反转后的链表

方法一：

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8
9
10 struct ListNode* reverseList(struct ListNode* head){
11     struct ListNode *prev = NULL;
12     struct ListNode *current = head;
13     while(current) {
14         struct ListNode* next = current->next;
15         current->next = prev;
16         prev = current;
17         current = next;
18     }
19
20     return prev;
21 }
```

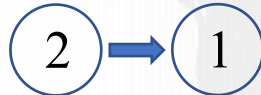
时间复杂度?

空间复杂度?

## ◆ 练习2

给你单链表的头节点**head**，请你反转链表，并返回反转后的链表

方法二：递归

1. 反转链表最终可以转化为如图所示的子问题：  
A diagram showing two nodes, each represented by a circle containing a number. The first circle contains the number '2' and the second circle contains the number '1'. A blue arrow points from the '2' node to the '1' node.
2. 将两个节点反转的操作为：`head.next.next = head`

## ◆ 练习2

给你单链表的头节点head，请你反转链表，并返回反转后的链表  
方法二：

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8
9
10 struct ListNode* reverseList(struct ListNode* head){
11     if (head == NULL || head->next == NULL) {
12         return head;
13     }
14     struct ListNode* new_head = reverseList(head->next);
15     head->next->next = head;
16     head->next = NULL;
17     return new_head;
18 }
19
```

时间复杂度?

空间复杂度?

## ◆ 练习3

给你一个链表，删除链表的倒数第 $n$ 个结点，返回链表的头节点。

test case1:

输入:  $\text{head} = [1, 2, 3, 4, 5]$ ,  $n = 2$

输出:  $\text{head} = [1, 2, 3, 5]$

test case2:

输入:  $\text{head} = [1]$ ,  $n = 1$

输出:  $\text{head} = []$

test case3:

输入:  $\text{head} = [1, 2]$ ,  $n = 1$

输出:  $\text{head} = [1]$

## ◆ 练习3

给你一个链表，删除链表的倒数第 $n$ 个节点，返回链表的头节点。

方法一：

1. 首先从头节点开始对链表进行一次遍历，得到链表的长度 $L$
2. 在从头节点开始对链表进行一次遍历，当遍历到第 $L-n+1$ 个节点时，它就是我们需要删除的节点

方法二：

1. 首先设立预先指针 $cur$ ，预先指针是一个小技巧，不用重复第二遍遍历
2. 设预先指针 $cur$ 指向  $head$
3.  $fast$  先向前移动 $n-1$ 步
4. 之后 $fast$ 和 $low$ 共同向前移动，此时二者的距离为  $n$ ，当  $fast$  到尾部时， $low$  的位置恰好为倒数第  $n$  个节点，删除即可



## ◆ 练习3

给你一个链表，删除链表的倒数第 $n$ 个节点，返回链表的头节点。

方法二：

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8
9
10 struct ListNode* removeNthFromEnd(struct ListNode* head, int n){
11     struct ListNode* dummy = malloc(sizeof(struct ListNode));
12     dummy->val = 0, dummy->next = head;
13     struct ListNode* first = head;
14     struct ListNode* second = dummy;
15     for (int i = 0; i < n; ++i) {
16         first = first->next;
17     }
18     while (first) {
19         first = first->next;
20         second = second->next;
21     }
22     second->next = second->next->next;
23     struct ListNode* ans = dummy->next;
24     free(dummy);
25     return ans;
26 }
```

时间复杂度?

空间复杂度?

## ◆ 练习4

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成。

test case1:

输入:  $l1 = [1, 2, 4]$ ,  $l2 = [1, 3, 4]$

输出:  $[1, 1, 2, 3, 4, 4]$

test case2:

输入:  $l1 = []$ ,  $l2 = []$

输出:  $[]$

test case3:

输入:  $head = [1, 2]$ ,  $n = 1$

输出:  $head = [1]$

## ◆ 练习4

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成。

方法一：

1. 首先，我们设定一个哨兵节点 `prehead`，这可以在最后让我们比较容易地返回合并后的链表。我们维护一个 `prev` 指针，我们需要做的是调整它的 `next` 指针
2. 然后，我们重复以下过程，直到 `l1` 或者 `l2` 指向了 `null`：
  - 如果 `l1` 当前节点的值小于等于 `l2`，我们就把 `l1` 当前的节点接在 `prev` 节点的后面同时将 `l1` 指针往后移一位
  - 否则，我们对 `l2` 做同样的操作。不管我们将哪一个元素接在了后面，我们都需要把 `prev` 向后移一位

## ◆ 练习4

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成。

方法一：

在循环终止的时候，l1 和 l2 **至多**有一个是非空的。由于输入的两个链表都是有序的所以不管哪个链表是非空的，它包含的所有元素都比前面已经合并链表中的所有元素都要大。这意味着我们只需要简单地将非空链表接在合并链表的后面，并返回合并链表即可。



## ◆ 练习4

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成。

方法一：

```
1 /**
2  * Definition for singly-linked list.
3  * struct ListNode {
4  *     int val;
5  *     struct ListNode *next;
6  * };
7  */
8 struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2){
9     if(list1 == NULL) return list2;
10    if(list2 == NULL) return list1;
11
12    struct ListNode* pre_head = (struct ListNode*)malloc(sizeof(struct ListNode));
13    struct ListNode* prev = pre_head;
14
15    while(list1 != NULL && list2 != NULL) {
16        if (list1->val < list2->val) {
17            prev->next = list1;
18            list1 = list1->next;
19        } else {
20            prev->next = list2;
21            list2 = list2->next;
22        }
23        prev = prev->next;
24    }
25    prev->next = list1 == NULL ? list2 : list1;
26
27    return pre_head->next;
28 }
```

时间复杂度?

空间复杂度?



## ◆ 练习4

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成。

方法二：递归

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8  struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2){
9      if(list1 == NULL) return list2;
10     if(list2 == NULL) return list1;
11
12     if(list1->val < list2->val) {
13         list1->next = mergeTwoLists(list1->next, list2);
14         return list1;
15     } else {
16         list2->next = mergeTwoLists(list1, list2->next);
17         return list2;
18     }
19 }
20
```



时间复杂度?



空间复杂度?

## ◆ 练习5

给你一个链表的数组，每个链表都已经按照升序排列。请你将所有链表合并到一个升序链表中，返回合并后的链表。

test case1:

输入：lists = [[1, 4, 5], [1, 3, 4], [2, 6]]

输出：[1, 1, 2, 3, 4, 4, 5, 6]

## ◆ 练习5

给你一个链表的数组，每个链表都已经按照升序排列。请你将所有链表合并到一个升序链表中，返回合并后的链表

方法一：

每次取出K个链表中最小的元素，不断放入结果数组中，直至所有链表遍历完毕

方法二：

用分治的方法进行合并

1. 将  $k$  个链表配对并将同一对中的链表合并；
2. 第一轮合并以后， $k$  个链表被合并成了  $k/2$  个链表，平均长度为  $2n/k$ ，然后是  $k/4$  个链表， $k/8$  个链表等等；
3. 重复这一过程，直到我们得到了最终的有序链表

## ◆ 练习6

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头结点。你必须在  
不修改结点内部的值的情况下完成本题（只能进行结点交换）

test case1:

输入：head = [1, 2, 3, 4]

输出：[2, 1, 4, 3]

test case2:

输入：head = []

输出：[]

test case3:

输入：head = [1]

输出：[1]

## ◆ 练习6

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头结点。你必须在  
不修改结点内部的值的情况下完成本题（只能进行结点交换）

方法一：

1. 这里我们需要三个指针，`a`，`b`，`tmp`。假设链表是`1->2->3->4->5->6`
2. 在迭代的时候，每次处理两个节点，于是第一轮 `a` 指向 1，`b` 指向 2。
3. 第二轮的时候 `a` 指向 3，`b` 指向 4。第三轮的时候 `a` 指向 5，`b` 指向 6。
4. 我们通过 `a.next = b.next`，以及 `b.next = a` 就把两个指针的位置反转了，于是`1->2`就变成`2->1`。

题目要求，最终应该是`2->1->4->3`。



## ◆ 练习6

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头结点。你必须在  
不修改结点内部的值的情况下完成本题（只能进行结点交换）

方法一：

```
struct ListNode* swapPairs(struct ListNode* head) {  
    struct ListNode dummyHead;  
    dummyHead.next = head;  
    struct ListNode* temp = &dummyHead;  
    while (temp->next != NULL && temp->next->next != NULL) {  
        struct ListNode* node1 = temp->next;  
        struct ListNode* node2 = temp->next->next;  
        temp->next = node2;  
        node1->next = node2->next;  
        node2->next = node1;  
        temp = node1;  
    }  
    return dummyHead.next;  
}
```

## ◆ 练习6

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头结点。你必须在不修改结点内部的值的情况下完成本题（只能进行结点交换）

方法二：

递归

```
1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     struct ListNode *next;
6   * };
7   */
8
9
10 public ListNode swapPairs(ListNode head) {
11     if(head == null || head.next == null) return head;
12
13     ListNode t1 = head.next;
14     ListNode t2 = head.next.next;
15     t1.next = head;
16     head.next = swapPairs(t2);
17     return t1;
18 }
19 }
```

## ◆ 练习7

给你一个链表的头节点**head**，每**k**个节点一组进行翻转，请你返回修改后的链表。**k**是一个正整数，它的值小于或等于链表的长度。如果节点总数不是**k**的整数倍，那么请将最后的剩余的节点保持原有顺序。你不能只是单纯的改变结点内部的值，而是需要实际进行结点交换。

test case1:

输入:  $\text{head} = [1, 2, 3, 4, 5]$ ,  $k = 2$

输出:  $[2, 1, 4, 3, 5]$

test case2:

输入:  $\text{head} = [1, 2, 3, 4, 5]$ ,  $k = 3$

输出:  $[3, 2, 1, 4, 5]$

## ◆ 练习7

```
int Length(struct ListNode* head){
    int num = 0;
    while(head){
        num++;
        head = head->next;
    }
    return num;
}

struct ListNode* reverseKGroup(struct ListNode* head, int k){
    int time = Length(head) / k;
    struct ListNode* dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;
    struct ListNode* pre = dummy;
    struct ListNode* rear = dummy->next;
    struct ListNode* temp;
    int num;
    while(time){
        num = 1;
        while(num < k){
            temp = rear->next;
            rear->next = temp->next;
            temp->next = pre->next;
            pre->next = temp;
            num++;
        }
        pre = rear;
        rear = pre->next;
        time--;
    }
    return dummy->next;
}
```

# Homework



leetcode上的练习题：全部AC



# Thanks!



**See you in the next session!**