数据结构：Homework 3

张霄　2019030045

October 23, 2023

Problem 1

1).

```
x->succ = x->succ->succ; // x的后继指向x后继的后继，即跳过x后面的一位
```

2).

```
t->succ = x->succ; // 将新结点t的后继指针指向x结点的后继结点，即将t插入到x结点后面
x->succ = t; // 将x结点的后继指针指向新结点t，完成插入操作
```

3).

```
ListNode* fun(ListNode* x){ // x指向列表的头结点，函数返回反转后的列表的头结点
    ListNode* a = x; // 定义指针a指向原始列表的头结点
    ListNode* b = NULL; // 定义指针b为NULL，用于存储反转后列表的头结点
    while (a != NULL){ // 当原始列表未遍历完时
        ListNode* c = a->succ; // 将指针c指向a的后继结点，保存起来以便后续操作
        a->succ = b; // 将a的后继指针指向b，完成反转操作
        b = a; // 更新b指针，使其指向当前反转后的列表的头结点
        a = c; // 将a指向原始列表中的下一个结点，准备进行下一次反转操作
    }
    return b; // 返回反转后的列表的头结点
}
```

Problem 2

1). 使用两个指针，他们之间的间距为 k，当快指针到达列表的末尾时，慢指针将指向倒数第 k 个节点。这里我们最多遍历一次列表，时间复杂度为 $\mathcal{O}(n)$，使用了常量级的额外指针，空间复杂度为 $\mathcal{O}(1)$。

```
// 定义单向列表的节点结构
struct ListNode {
    int val;
    ListNode* succ;
    ListNode(int x) : val(x), succ(NULL) {}
};
ListNode* findKthFromEnd(ListNode* head, int k) {
    // 定义快指针和慢指针，初始时都指向列表的头节点
    ListNode* fast = head;
    ListNode* slow = head;
    if (k == 0){
    // 单独列出来，因为如果需要输出最后一个节点用双指针相当于遍历了两次
```

```
        while(fast->succ != NULL){
          fast = fast->succ;
        }
        return fast;
    }
    // 将快指针移动k步，使其与慢指针相隔k个节点
    for (int i = 0; i < k; ++i) {
        // 如果快指针已经到达列表末尾，说明列表长度小于n，无法找到倒数第n个节点，返回NULL
        if (fast == NULL) {
            return NULL;
        }
        fast = fast->succ;
    }
    // 同时移动快慢指针，直到快指针到达列表末尾
    while (fast->succ != NULL) {
        fast = fast->succ;
        slow = slow->succ;
    }
    // 此时慢指针指向倒数第k个节点，返回慢指针指向的节点即可
    return slow;
}
```

2). 使用两个指针，一个快指针和一个慢指针，它们之间的间隔为 k+1。当快指针到达列表的末尾时，慢指针将指向要删除节点的前一个节点。然后通过修改慢指针的 succ 指针来删除倒数第 k 个节点。这里我们最多遍历一次列表，因此时间复杂度为 $\mathcal{O}(n)$，使用了常量级的额外指针，空间复杂度为 $\mathcal{O}(1)$。

```
// 定义单向列表的节点结构
struct ListNode {
    int val;
    ListNode* succ;
    ListNode(int x) : val(x), succ(NULL) {}
};
ListNode* removeFromEnd(ListNode* head, int k) {
    // 定义快指针和慢指针，初始时都指向列表的头节点
    ListNode* fast = head;
    ListNode* slow = head;
    // 将快指针移动k+1步，使其与慢指针相隔k个节点
    for (int i = 0; i <= k; ++i) {
        // 如果快指针已经到达列表末尾，说明列表长度小于k，直接返回头节点的下一个节点
        if (fast->succ == NULL) {
            return head->succ;
        }
        fast = fast->succ;
    }
    // 同时移动快慢指针，直到快指针到达列表末尾
    while (fast->succ != NULL) {
        fast = fast->succ;
        slow = slow->succ;
    }
    // 此时慢指针指向倒数第k+1个节点，修改其succ指针，删除倒数第k个节点
    slow->succ = slow->succ->succ;
    return head;// 返回列表的头节点
}
```

Problem 3

仍旧使用双指针的方法，两个列表的头部分别为 **len1**，**len2**，然后遍历计算两个列表的长度。因为它们在第一个公共节点之后的部分应该一样，所以我们计算两个列表的长度差，把较长列表的指针 **node1** 先向后移 **abs**(**len1** − **len2**)，较短的列表指针 **node2** 仍旧指在开头。接着同时移动两个指针，他们的 **succ** 指向同一个节点为止，输出节点即输出指向这个节点的指针。由于我们最多需要遍历两个列表各两次，因此时间复杂度是 $\mathcal{O}(n)$，而因为只用了常量级的额外空间，空间复杂度为 $\mathcal{O}(1)$。

Algorithm 1 problem 3

```
len1 = 0
len2 = 0
node1 = head1
node2 = head2
while node1 is not NULL do:                          ▷ 计算列表 1 的长度
    len1++
    node1 = node1 → succ
end while
while node2 is not NULL do:                          ▷ 计算列表 2 的长度
    len2++
    node2 = node2 → succ
end while                                            ▷ 调整列表起始位置
node1 = head1
node2 = head2
diff = abs(len1 − len2)
if len1 > len2 then:
    for i from 1 to diff do:
        node1 = node1 → succ
    end for
else:
    for i from 1 to diff do:
        node2 = node2 → succ
    end for
end if
                                                     ▷ 同步移动两个指针，直到找到第一个公共节点
while node1 is not NULL and node2 is not NULL do:
    if node1 == node2 then:
        return node1                                 ▷ 找到第一个公共节点
    end if
    node1 = node1 → succ
    node2 = node2 → succ
end while
return NULL                                          ▷ 没有找到公共节点
```

Problem 4

Algorithm 2 problem 4

```
function BinarySearch(vector, target):
```

```
    left = 0
    right = length(vector) − 1
    while left ≤ right do:
        mid = left + (right − left) / 2
        if vector[mid] == target then:
            return mid                                  ▷ 找到目标，返回目标元素的索引
        else if vector[mid] < target then:
            left = mid + 1                              ▷ 目标在右半部分，缩小搜索范围
        else:
            right = mid − 1                             ▷ 目标在左半部分，缩小搜索范围
        end if
    end while
    return -1                                           ▷ 没有找到目标，返回-1
end function
```

因为是一个已经排列好的向量，每次比较都将搜索范围缩小为原来的一半。在最坏情况下，二分查找最多需要 $\mathcal{O}(\log n)$ 次比较来找到目标元素

Problem 5

Algorithm 3 problem 5

```
function BinarySearch(head, target)
    left = head
    right = NULL                                        ▷ 右指针初始化为 NULL，表示列表结束
    while left ≠ right and left ≠ NULL do
        mid = FindMiddle(left, right)                   ▷ 找到中间节点
        if mid → val = target then
            return mid                                  ▷ 找到目标节点，返回节点指针
        else if mid → val < target then
            left = mid → next                           ▷ 目标在右半部分，移动左指针
        else
            right = mid                                 ▷ 目标在左半部分，调整右指针
        end if
    end while
    return NULL                                         ▷ 没有找到目标节点，返回 NULL
end function
function FindMiddle(left, right)
    slow = left
    fast = left
    while fast ≠ right and fast → next ≠ right do
        slow = slow → next
        fast = fast → next → next
    end while
    return slow                                         ▷ 返回中间节点指针
end function
```

由于每一次比较都将搜索范围缩小为原来的一半，所以在有序单向列表上进行二分查找的时间复杂度是

$\mathcal{O}(\log n)$, 其中 n 是列表的长度。但找到单向列表的中值需要对列表进行遍历, 所以总的时间复杂度是 $\mathcal{O}(n)$。