Linked List

Contents

1. 单链表结点定义

```
typedef struct LinkNode
{
	int data;
	struct LinkNode* next;
}LinkNode_t;
```

2. 单链表从尾到头打印

题目：输入一个链表的头结点，从尾到头反过来打印出每个节点的值。

```
LinkNode_t *reversedLinkedList(LinkNode_t *rootNode){
	LinkNode_t *tempNode, *nextNode;
	LinkNode_t *next;

	next = NULL;
	tempNode = rootNode;

	while (tempNode->next != NULL)
	{
		nextNode = tempNode->next;
		tempNode->next = next;
		next = tempNode;
		tempNode = nextNode;
	}

	tempNode->next = next;

	return tempNode;
}
```

3. 删除链表结点

题目：在 O（1）的时间复杂度内删除链表节点。

给定单向链表的头指针和一个节点指针，定义一个函数在 O（1）时间内删除该节点。

① 如果该节点不是尾节点，那么可以直接将下一个节点的值赋给该节点，然后令该节点指向下下个节点，再删除下一个节点，时间复杂度为 O(1)。

② 否则，就需要先遍历链表，找到节点的前一个节点，然后让前一个节点指向 null，时间复杂度为 O(N)。

```
LinkNode_t *delLinkedList(LinkNode_t *rootNode, LinkNode_t *delNode){
	LinkNode_t *nextNode;
	if (delNode->next == NULL){
		if (rootNode == delNode)
			return NULL;

		nextNode = rootNode;
		while(nextNode->next != delNode){
			nextNode = nextNode->next;
		}

		nextNode->next = NULL;
		return rootNode;
	}


	nextNode = delNode->next;
	delNode->data = nextNode->data;
	delNode->next = nextNode->next;

	return rootNode;
}
```

4. 查找单链表的中间元素

i 走两步，j 走一步……，由此来判断中间的元素

```
LinkNode_t *findMidNode(LinkNode_t *rootNode){
	LinkNode_t *step1, *step2;

	step1 = rootNode;
	step2 = rootNode;
```

```
	while(step2->next != NULL){
		if (step2->next->next == NULL){
			break;
		}

		step1 = step1->next;
		step2 = step2->next->next;
	}


	return step1;
}
```

5. 链表中的倒数第 K 个结点

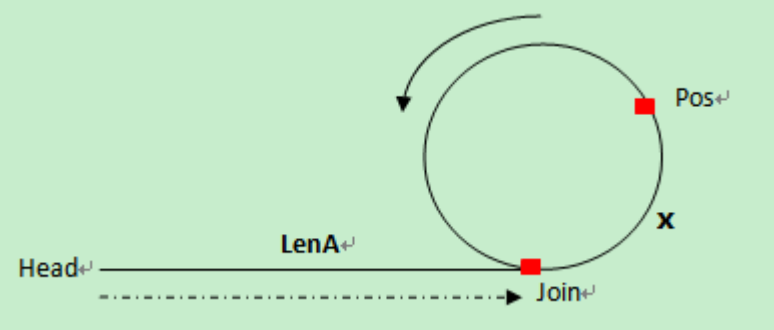
题目：输入一个链表，输出该链表中倒数第 K 个节点。为符合大多数人的习惯，从 1 开始计数，即链表的尾结点是倒数第一个节点。

两个节点 i 和 i+k。

```
LinkNode_t *findinvKNode(LinkNode_t *rootNode, int K){
	LinkNode_t *step1, *step2;
	int cnt = 1;

	step1 = rootNode;
	step2 = rootNode;

	while(cnt < K){
		if (step2 == NULL)
			return NULL;

		step2 = step2->next;
		cnt++;
	}


	while (step2->next != NULL){
		step1 = step1->next;
		step2 = step2->next;
	}

	return step1;
}
```

6. 检测环的存在

使用两个 slow, fast 指针从头开始扫描链表。**指针 slow 每次走 1 步，指针 fast 每次走 2 步。如果存在环，则指针 slow、fast 会相遇**；如果不存在环，指针 fast 遇到 NULL 退出。



求有环单链表的环长

在环上相遇后，记录第一次相遇点为 Pos，之后指针 slow 继续每次走 1 步，fast 每次走 2 步。**在下次相遇的时候 fast 比 slow 正好又多走了一圈，也就是多走的距离等于环长。**

设从第一次相遇到第二次相遇，设 slow 走了 len 步，则 fast 走了 2*len 步，相遇时多走了一圈：

环长=2*len-len。

求有环单链表的环连接点位置

第一次碰撞点 Pos 到连接点 Join 的距离=头指针到连接点 Join 的距离，因此，分别一指针从第一次碰撞点 Pos、另外一指针从头指针 head 开始走，每一步走一个节点，相遇的那个点就是连接点。

在环上相遇后，记录第一次相遇点为 Pos，连接点为 Join，假设头结点到连接点的长度为 **LenA**，连接点到第一次相遇点的长度为 **x**，环长为 **R**。

第一次相遇时，slow 走的长度 S = **LenA** + **x**;

第一次相遇时，fast 走的长度 2S = **LenA** + n*R + **x**;

所以可以知道，**LenA** + **x** = n*R;　　**LenA = n*R -x;**

求有环单链表的链表长

上述 2 中求出了环的长度；3 中求出了连接点的位置，就可以求出头结点到连接点的长度。两者相加就是链表的长度。

```
int checkRing(LinkNode_t *rootNode){
	LinkNode_t *slow, *fast;
	int Ring_OK = 0;
```

```
	slow = rootNode;
	fast = rootNode;
	// Need to add whether rootNode is NULL.

	while(fast->next != NULL){
		if (fast->next->next == NULL){
			Ring_OK = 0;
			break;
		}

		slow = slow->next;
		fast = fast->next->next;

		if (fast == slow){
			Ring_OK = 1;
			break;
		}
	}

	printf("Whether exist ring or not: %d. \n", Ring_OK);
	fflush(stdout);


	return 0;
}
```

```
int checkRing_with_RingLen(LinkNode_t *rootNode){
	LinkNode_t *slow, *fast;
	int Ring_OK = 0;
	int meet_cnt = 0;
	int step_cnt = 0; // step count after first meet

	slow = rootNode;
	fast = rootNode;

	while(fast->next != NULL){
		if (fast->next->next == NULL){
			Ring_OK = 0;
			break;
		}

		slow = slow->next;
		fast = fast->next->next;

		if (meet_cnt == 1){
			step_cnt++;
		}
```

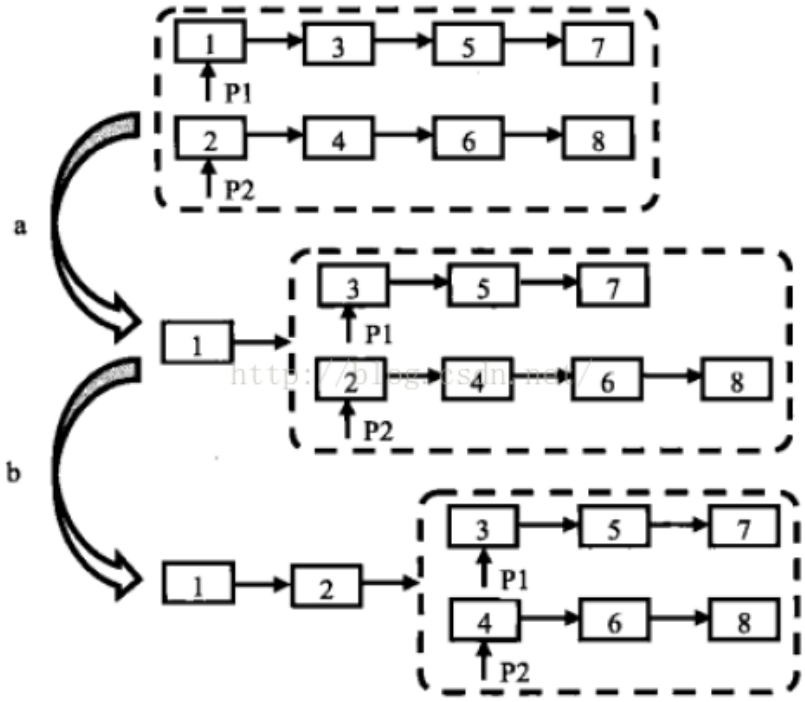
```
			if (fast == slow){
				if (meet_cnt == 1){
					meet_cnt = 2;
					break;
				}

				if (meet_cnt == 0){
					Ring_OK = 1;
					meet_cnt = 1;
				}
			}
		}

		printf("Whether exist ring or not: %d, ring length: %d. \n", Ring_OK,
step_cnt);
		fflush(stdout);


		return 0;
}
```

```
int checkRing_with_RingLen_RingPos(LinkNode_t *rootNode){
		LinkNode_t *slow, *fast;
		int Ring_OK = 0;
		int meet_cnt = 0;
		int step_cnt = 0; // step count

		slow = rootNode;
		fast = rootNode;

		while(fast->next != NULL){
			if (fast->next->next == NULL){
				Ring_OK = 0;
				break;
			}

			slow = slow->next;
			fast = fast->next->next;

			if (fast == slow){
					Ring_OK = 1;
					break;
			}
		}


		slow = rootNode;
```

```
        while (fast!=slow){
                slow = slow->next;  // both one step once
                fast = fast->next; // both one step once, fast also go with one step
        }


        printf("Whether exist ring or not: %d, Ring connection position: %d. \n",
Ring_OK, slow->data);
        fflush(stdout);


        return 0;
}
```

7. 合并两个有序链表

题目：输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。



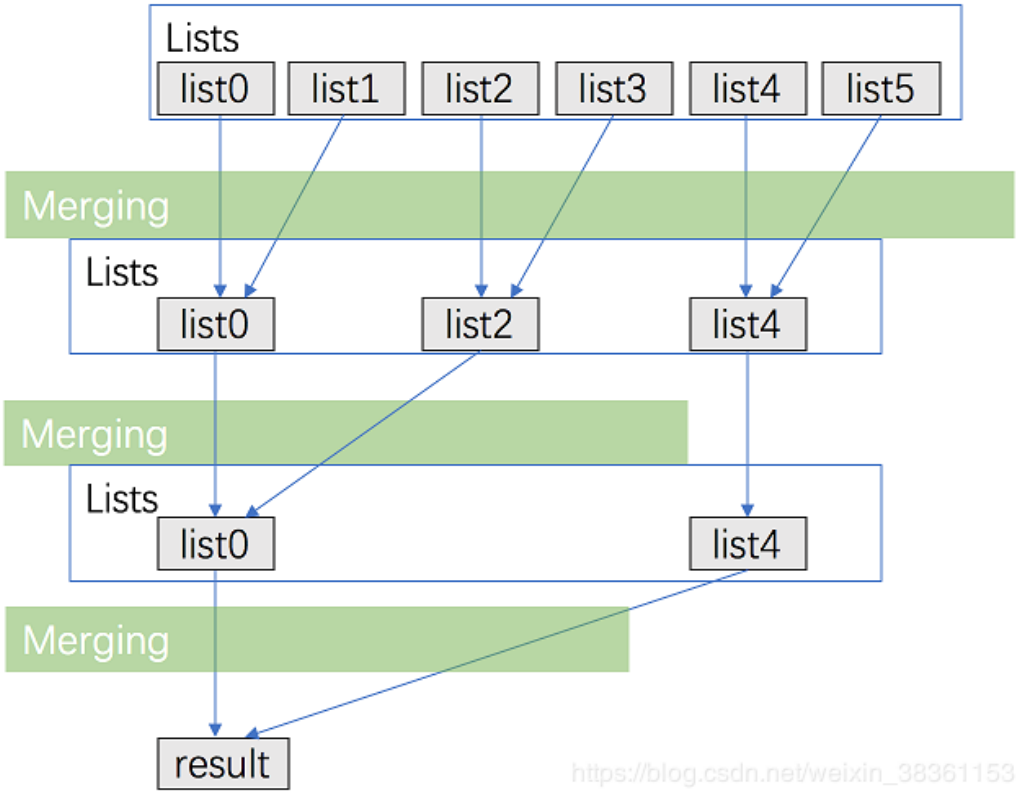

```
LinkNode_t *combineLinkedList(LinkNode_t *list_A, LinkNode_t *list_B){
        LinkNode_t *p1, *p2, *pT;
        LinkNode_t *targetNode;
```

```
        int cnt = 1;

        targetNode = (LinkNode_t *)malloc(sizeof(LinkNode_t));

        p1 = list_A;
        p2 = list_B;
        pT = targetNode;

        if (p1 == NULL)
                return list_B;

        if (p2 == NULL)
                return list_A;


        if (p1->data > p2->data){
                pT->data = p2->data;
                p2 = p2->next;
        }else{
                pT->data = p1->data;
                p1 = p1->next;
        }

        while ((p1!=NULL) && (p2!=NULL)){
                if (p1->data > p2->data){
                        pT->next = p2;
                        p2 = p2->next;
                }else{
                        pT->next = p1;
                        p1 = p1->next;
                }

                pT = pT->next;
        }


        if (p1==NULL){
                pT->next = p2;
        }

        if (p2==NULL){
                pT->next = p1;
        }

        return targetNode;
}
```

8. 合并 k 个排序链表

返回合并后的排序链表。请分析和描述算法的复杂度。

k 个有序链表合并这个问题，可以看作是归并排序回溯过程中的一个状态，使用分治的思想求解，不过和归并排序不同的是，这里只有治而没有分。
下面这个图详细体现了算法过程，并且我们可以原地归并，不需要申请新的数组空间：



原地归并的算法实现，其实就是一个找规律问题。第一轮的归并是，0 和 1，2 和 3，4 和 5 ...；第二轮的归并是，0 和 2，4 和 6 ...；第三轮的归并是，0 和 4，4 和 8 ...；... 直到两归并链表的间距大于等于数组长度为止。

类似归并排序的回溯过程，两两合并。下面解法的时间复杂度也为 O(n * logk)，k 为链表个数，n 为总的结点数，空间复杂度为 O(1)。
时间复杂度分析：两两归并，每个结点会被归并 logk 次，所以总的时间复杂度为 O(n * logk)。

参考资源：

[1] https://www.cnblogs.com/xudong-bupt/p/3667729.html

[2] https://blog.csdn.net/weixin_38361153/article/details/93514302