基础算法与数据结构（一） 链表

本部分内容一般选用C++，如有Java代码，将有明确标注。

链表的基础数据结构

```
struct ListNode{
    int val;          //may be the other data type
    ListNode* next;
    ListNode(int val){
        this.val = val;
        this.next = null;
    }
}
```

链表的基本性质

- 链表克服了数组增删改操作需要移动大量元素的缺点，它不要求逻辑上相邻的两个元素在物理实现上相邻（数组要求）。但失去了数组可以随机存取的优点。
- 链表的存取必须从第一个结点开始。所以对链表进行增删改操作时的最坏时间复杂度应为O(n)
- 静态链表：用数组模仿链表，数组的第二个数据存下一个结点的位置

链表的基础增删以及逆转操作

- 新增元素：

```
//pos is the number which count from the 0 and don't have the dommy node
void addNodes(ListNode* head,int val,int pos){
    ListNode* cur = head;
    while(pos-- && cur){
        cur = cur->next;
    }
    if(cur){
        ListNode* next = cur->next;
        cur->next = new ListNode(val);
        cur->next->next = next;
    }
}
```

- 删除元素

```
  //pos is the number which count from the 0 and don't have the dommy node
  void deleteNodes(ListNode* head,int pos){
      ListNode* cur = head;
      int nowPos = j;
      while(cur && j<i-1){
          cur = cur->next;
          j++;
      }
      ListNode *del = cur->next;
      cur-next = del->next;
  }
```

- 逆转操作

```
  // the head node is the node which is the fist node of the sublist which will
be reversed.
  //no recursion
  ListNode* Reverse(ListNode* head){
      ListNode* pre = null;
      ListNode* next = null;
      while(head){
          next = head->next;
          head->next = next;
          pre = head;
          head = next;
      }
      return pre;
  }

  //recursion
  ListNode* Reverse(ListNode* head){
      if(!head || !head->next) return head;
      ListNode* thead = Reverse(head->next);
      head->next->next = head;
      head->next = NULL;
      return thead;
```

```
}
```

链表题 – 快慢指针

基本思想为 快指针一次走两步，慢指针一次走一步，根据产生的效果判断一些问题。注意点即指针的有效性问题。

判断单链表是否为循环链表

- 基本理念：如果存在循环，总有fast和slow相遇的时候，如果相遇则退出。如果是因为这个退出循环的，那么总存在fast或者fast的next节点不为空，即返回真值。如果是因为fast节点指向空才跳出的循环，那么就不存在循环。

```
bool isExistLoop(ListNode* head){
    ListNode* fast,slow;
    fast = slow = head;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast) break;
    }
    return (!fast || !fast->next)
}
```

在有序列表中寻找中位数（在链表中找到中间）

```
int findMid(ListNode* head){
    ListNode* slow,fast;
    slow = fast = head;
    while(fast->next && fast->next->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    if(!fast->next) return slow->val;
    else return ((slow->val)+(slow->next->val))/2;
}
```

有环链表寻找入口点

- 原理： fast和slow的相遇点和起点分别设为两个指针，每次各走一步，则这两个指针的相遇点即为入口点。
- 证明：

  设 起点到环的入口的长度为a，入口处到快慢指针相遇处的长度为b
  那么慢指针从开始到相遇处走了(a+b)步，那么快指针走了2(a+b)步
  设快指针此时绕着环走了n圈，环的长度为r
  那么有
  2(a+b) = (a+b) + n * r
  → a+b = nr
  → a = (n-1) * r + r - b

相遇点开始的走了n-1圈和r-b的长度回到起点的同时，起点开始的来到了环的起点。

代码略

双指针问题

基础思想： 慢指针从头开始，快指针从k开始，当快指针走到结束时，慢指针走到倒数第k个的位置。

例题： LEETCode 234 Palindrome Linked List

- 一般链表解法： 用双指针找到中点位置 → 中点的下一个结点开始到末尾逆转链表 → 从头尾开始向中间，如果出现不同的点则不为回文
- 巧解： 转为字符串，并且reverse，如果为相同字符串那么为回文串

链表相交问题

首先明确，如果两个链表有相交，一定出现Y字形的相交，相交点之后的两个链表相同，不可能出现X型的相交。

分类讨论链表是否有环。

- 两个无环链表： 判断末尾是否相同，如果为同一个节点那么相交（判断地址）
  求交点： 遍历两个链表得到两个链表的长度len1,len2，从长的链表先走|len2-len1|的长度，在让另一指针从短的头开始走，两个指针相等的那个结点即为交点

- 可能有环
  判断有无环
  - 两个均无环： 划归为1
  - 一个有环一个无环： 根据首先明确的，两个不可能相交

- 两个有环： 从一个链表上得到快慢指针的第一个相交点，判断这个相交点是否在第二个链表上，如果在第二个链表的环上那么一定相交。然后划归为1（链表长度为链表起点到环的入口处）

链表题 – 归并有序链表

两个链表的归并

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
     ListNode* start = new ListNode(0);
     ListNode* pre = start;
     if(!l1 ||!l2 ) return l1?l1:l2;
     while(l1 != NULL && l2 != NULL){
         if(l1->val < l2->val){
             pre->next = l1;
             l1 = l1->next;
         }else{
             pre->next = l2;
             l2 = l2->next;
         }
         pre = pre->next;
     }
     pre->next = l1?l1:l2;
     return start->next;
  }
```

k个链表的归并

- 两两归并 O(kn^2)
- 通过优先队列每次取出最小值节点归并
- 分治法归并

分治法归并代码：

```
ListNode* mergeKLists(vector<ListNode*>& lists) {
      if(lists.empty()) return NULL;
      return divideAndConquer(lists,0,lists.size()-1);
```

```
    }

    ListNode* divideAndConquer(vector<ListNode*> &lists,int left,int right){
        if(left == right) return lists[right];
        int mid = (left+right)/2;
        ListNode* left_merge =
          divideAndConquer(lists,left,mid);
        ListNode* right_merge =
          divideAndConquer(lists,mid+1,right);
        return mergeTwoLists(left_merge,right_merge);
    }
```