单链表集体思路

合并两个有序链表

```
class Solution {
public:
  ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode* dummy = new ListNode();  // 虚拟头结点
    ListNode* p     = dummy;
    while (l1 && l2) {
      if (l1->val <= l2->val) {
        p->next = l1;
        l1      = l1->next;
        p       = p->next;
      } else {
        p->next = l2;
        l2      = l2->next;
        p       = p->next;
      }
    }

    if (l1) {
      p->next = l1;
    }
    if (l2) {
      p->next = l2;
    }
    p = dummy->next;
    delete dummy;

    return p;
  }
};
```

合并k个有序链表

```
class Solution {
public:
  ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    // 构建一个小根堆
    std::priority_queue<ListNode*, std::vector<ListNode*>, greater> pq;
    for (auto head : lists) {
      if (head != nullptr) {
        pq.push(head);
      }
    }

    ListNode* dummy = new ListNode();
    ListNode* p     = dummy;
```

```
    while (pq.size()) {
      ListNode* node = pq.top();
      pq.pop();
      if (node->next != nullptr) {
        pq.push(node->next);
      }

      p->next = node;
      p       = node;
    }

    p = dummy->next;
    delete dummy;
    return p;
  }

private:
  struct greater {
    bool operator()(const ListNode* t1, const ListNode* t2) {
      return t1->val >= t2->val;
    }
  };
};
```

链表的第k个节点

```
class Solution {
public:
  ListNode* removeNthFromEnd(ListNode* head, int n) {
    if (head == nullptr || n < 1) {
      return head;
    }

    ListNode* dummy = new ListNode(-1);
    dummy->next     = head;
    ListNode* p     = FindKthNode(dummy, n + 1);

    p->next = p->next->next;

    p = dummy->next;
    delete dummy;
    return p;
  }

private:
  ListNode* FindKthNode(ListNode* head, int k) {
    ListNode* p = head;
    for (int i = 1; i <= k; i++) {
      p = p->next;
    }
```

```
    ListNode* q = head;
    while (p != nullptr) {
      p = p->next;
      q = q->next;
    }

    return q;
  }
};
```

链表中点

```
class Solution {
public:
  ListNode* middleNode(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
    while (fast && fast->next) {
      slow = slow->next;
      fast = fast->next->next;
    }

    return slow;
  }
};
```

链表是否包含环

```
void hasCycle(ListNode *head) {
  ListNode *slow = head;
  ListNode *fast = head;
  while(fast && fast->next) {
    slow = slow->next;
    fast = fast->next->next;

    if(fast == slow){
      return true;
    }
  }
  return false;
}
```

求链表环起点

```
ListNode *detectCycle(ListNode *head) {
  // 判断是否存在环
  ListNode *slow = head;
  ListNode *fast = head;
  while(fast && fast->next){
    slow = slow->next;
    fast = fast->next->next;
    if(slow == fast){
      break;
    }
  }

  // 不存在环
  if(fast == nullptr || fast->next == nullptr){
    return nullptr;
  }

  // 存在环
  slow = head;
  while(slow != fast) {
    slow = slow->next;
    fast = fast->next;
  }

  return slow;
}
```

两个链表是否相交

```
class Solution {
public:
  ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *p1 = headA;
    ListNode *p2 = headB;

    while (p1 != p2) {
      if (p1 == nullptr) {
        p1 = headB;
      } else {
        p1 = p1->next;
      }

      if (p2 != nullptr) {
        p2 = headA;
      } else {
        p2 = p2->next;
      }
    }

    return p1;
```

```
    }
};
```