

141. 环形链表

141. 环形链表

难度 简单

👍 565

♡ 收藏

🔗 分享

🌐 切换为英文

🔔 关注

💡 反馈

给定一个链表，判断链表中是否有环。

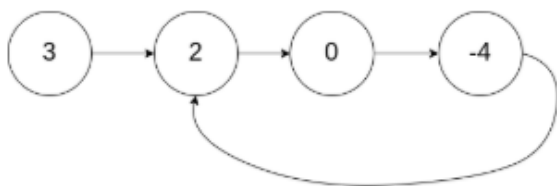
为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。

示例 1:

输入: `head = [3,2,0,-4]`, `pos = 1`

输出: `true`

解释: 链表中有一个环，其尾部连接到第二个节点。



示例 2:

输入: `head = [1,2]`, `pos = 0`

输出: `true`

解释: 链表中有一个环，其尾部连接到第一个节点。



```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head || !head->next) return NULL;
        auto first = head, second = head->next;
        while(first && second)
        {
            if(first == second) return true;
            first = first->next;
            second = second->next;
            if(second) second = second->next;
        }
    }
};
```





```

    }
    return false;
}
};

```

142. 环形链表 II

142. 环形链表 II

难度 中等  441  收藏  分享  切换为英文  关注  反馈

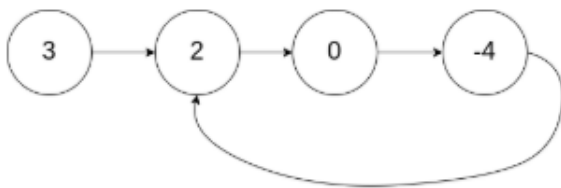
给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。

说明：不允许修改给定的链表。

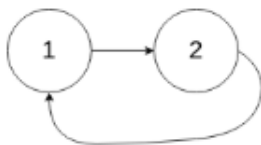
示例 1:

输入: `head = [3,2,0,-4]`, `pos = 1`
 输出: tail connects to node index 1
 解释: 链表中有一个环，其尾部连接到第二个节点。



示例 2:

输入: `head = [1,2]`, `pos = 0`
 输出: tail connects to node index 0
 解释: 链表中有一个环，其尾部连接到第一个节点。



```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        auto fast = head, slow = head;
        while(fast)
        {

```

```




        fast = fast->next;
        slow = slow->next;
        if(fast) fast = fast->next;
        else break;

        if(fast == slow)
        {
            fast = head;
            while(slow != fast)
            {
                fast = fast->next;
                slow = slow->next;
            }
            return slow;
        }
    }
    return NULL;
}
};

```

143. 重排链表

143. 重排链表

难度 中等  200  收藏  分享  切换为英文  关注  反馈

给定一个单链表 $L: L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$ ，
将其重新排列后变为： $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1:

给定链表 1->2->3->4，重新排列为 1->4->2->3。

示例 2:

给定链表 1->2->3->4->5，重新排列为 1->5->2->4->3。

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    void reorderList(ListNode* head) {
        int n = 0;
        for (ListNode *p = head; p; p = p->next) n ++ ;
        if (n <= 2) return;
        ListNode *later = head;
        for (int i = 0; i + 1 < (n + 1) / 2; i ++ )

```

```

        later = later->next;
        ListNode *a = later, *b = later->next;






        while (b)
        {
            ListNode *c = b->next;
            b->next = a;
            a = b;
            b = c;
        }

        later->next = 0;
        while (head && head != a)
        {
            b = a->next;
            a->next = head->next;
            head->next = a;
            head = head->next->next;
            a = b;
        }
    }
};

```

144. 二叉树的前序遍历

144. 二叉树的前序遍历

难度 **中等**  239  收藏  分享  切换为英文  关注  反馈

给定一个二叉树，返回它的 *前序遍历*。

示例:

输入: [1,null,2,3]

```

  1
   \
    2
   /
  3

```

输出: [1,2,3]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {

```



```

vector<int> res;
while (root)
{
    if (!root->left)
    {
        res.push_back(root->val);
        root = root->right;
    }
    else
    {
        TreeNode *pre = root->left;
        while (pre->right && pre->right != root) pre = pre->right;
        if (pre->right) pre->right = 0, root = root->right;
        else
        {
            pre->right = root;
            res.push_back(root->val);
            root = root->left;
        }
    }
}
return res;
};

```

145. 二叉树的后序遍历

145. 二叉树的后序遍历

难度 **困难**  277  收藏  分享  切换为英文  关注  反馈

给定一个二叉树，返回它的 *后序* 遍历。

示例:

输入: [1,null,2,3]

```

  1
   \
    2
   /
  3

```

输出: [3,2,1]

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */

```

```

class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result, left, right;
        if(!root) return result;

        left = postorderTraversal(root->left);
        for(auto &x:left) result.push_back(x);

        right = postorderTraversal(root->right);
        for(auto &x:right) result.push_back(x);




        result.push_back(root->val);

        return result;
    }
};

```

146. LRU缓存机制

146. LRU缓存机制

难度 中等  510  收藏  分享  切换为英文  关注  反馈

运用你所掌握的数据结构，设计和实现一个 **LRU (最近最少使用)** 缓存机制。它应该支持以下操作：获取数据 `get` 和 写入数据 `put` 。

获取数据 `get(key)` - 如果密钥 (`key`) 存在于缓存中，则获取密钥的值（总是正数），否则返回 `-1`。
 写入数据 `put(key, value)` - 如果密钥已经存在，则变更其数据值；如果密钥不存在，则插入该组「密钥/数据值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 **$O(1)$** 时间复杂度内完成这两种操作？

示例:

```

LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);    // 返回 1
cache.put(3, 3);  // 该操作会使得密钥 2 作废
cache.get(2);    // 返回 -1 (未找到)
cache.put(4, 4);  // 该操作会使得密钥 1 作废
cache.get(1);    // 返回 -1 (未找到)
cache.get(3);    // 返回 3
cache.get(4);    // 返回 4

```

```

class LRUCache {

```

```

public:
    struct Node
    {
        int val, key;
        Node *left, *right;
        Node() : key(0), val(0), left(NULL), right(NULL) {}
    };
    Node *hu, *tu; // hu: head_used, tu: tail_used; head在左侧, tail在右侧
    Node *hr, *tr; // hr: head_remains, tr: tail_remains; head在左侧, tail在右侧
    int n;
    unordered_map<int, Node*> hash;

    void delete_node(Node *p)
    {
        p->left->right = p->right, p->right->left = p->left;
    }

    void insert_node(Node *h, Node *p)
    {
        p->right = h->right, h->right = p;
        p->left = h, p->right->left = p;
    }

    LRUCache(int capacity) {
        n = capacity;
        hu = new Node(), tu = new Node();
        hr = new Node(), tr = new Node();
        hu->right = tu, tu->left = hu;
        hr->right = tr, tr->left = hr;

        for (int i = 0; i < n; i++) {
            Node *p = new Node();
            insert_node(hr, p);
        }
    }

    int get(int key) {
        if (hash[key])
        {
            Node *p = hash[key];
            delete_node(p);
            insert_node(hu, p);
            return p->val;
        }
        return -1;
    }

    void put(int key, int value) {
        if (hash[key])
        {
            Node *p = hash[key];
            delete_node(p);
            insert_node(hu, p);
            p->val = value;
            return;
        }
    }

```

```

    if (!n)
    {
        n ++ ;
        Node *p = tu->left;
        hash[p->key] = 0;
        delete_node(p);
        insert_node(hr, p);
    }

    n -- ;
    Node *p = hr->right;
    p->key = key, p->val = value, hash[key] = p;
    delete_node(p);
    insert_node(hu, p);
}

};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */

```

147. 对链表进行插入排序

147. 对链表进行插入排序

难度 中等

150

收藏

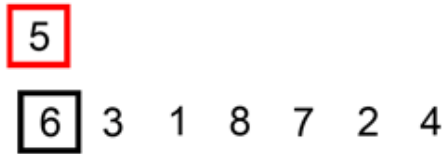
分享

切换为英文

关注

反馈

对链表进行插入排序。



插入排序的动画演示如上。从第一个元素开始，该链表可以被认为已经部分排序（用黑色表示）。每次迭代时，从输入数据中移除一个元素（用红色表示），并原地将其插入到已排好序的链表中。

插入排序算法：

1. 插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。
2. 每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。
3. 重复直到所有输入数据插入完为止。

示例 1:

输入：4->2->1->3

输出：1->2->3->4

示例 2:

输入：-1->5->3->4->0

输出：-1->0->3->4->5

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        auto dummy = new ListNode(-1);
        while(head)
        {
            auto p = head->next, q = dummy;
            while(q->next && q->next->val <= head->val) q = q->next;

            head->next = q->next;
```

```

        q->next = head;

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

```

148. 排序链表

148. 排序链表

难度 中等  494  收藏  分享  切换为英文  关注  反馈

在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

输入: 4->2->1->3
输出: 1->2->3->4

示例 2:

输入: -1->5->3->4->0
输出: -1->0->3->4->5

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        int n = 0;
        for (ListNode *p = head; p; p = p->next) n ++ ;

        ListNode *dummy = new ListNode(-1);
        dummy->next = head;
        for (int i = 1; i < n; i += 2)
        {
            ListNode *begin = dummy;
            for (int j = 0; j + i < n; j += i * 2)
            {
                ListNode *first = begin->next, *second = first;
                for (int k = 0; k < i; k ++ )
                    second = second->next;
                int f = 0, s = 0;
                while (f < i && s < i && second)
                    if (first->val < second->val)
                    {
                        begin->next = first;

```

```

        first = first->next;
        f ++ ;
    }
    else
    {
        begin = begin->next = second;
        second = second->next;
        s ++ ;
    }

    while (f < i)
    {
        begin = begin->next = first;
        first = first->next;
        f ++ ;
    }
    while (s < i && second)
    {
        begin = begin->next = second;
        second = second->next;
        s ++ ;
    }

    begin->next = second;
}

return dummy->next;
};

```

149. 直线上最多的点数

149. 直线上最多的点数

难度 **困难**

131

收藏

分享

切换为英文

关注

反馈

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

示例 1:

输入: `[[1,1],[2,2],[3,3]]`

输出: 3

解释:

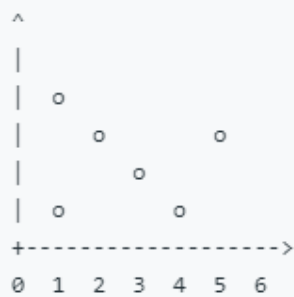


示例 2:

输入: `[[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]`

输出: 4

解释:



```
/**
 * Definition for a point.
 * struct Point {
 *     int x;
 *     int y;
 *     Point() : x(0), y(0) {}
 *     Point(int a, int b) : x(a), y(b) {}
 * };
 */
class Solution {
public:
    int maxPoints(vector<Point>& points) {
        if (points.empty()) return 0;
        int res = 1;
        for (int i = 0; i < points.size(); i++) {
            unordered_map<long double, int> map;
            int duplicates = 0, verticals = 1;

            for (int j = i + 1; j < points.size(); j++) {
                if (points[i].x == points[j].x) {
                    verticals++;
                }
            }
        }
    }
};
```

```

        if (points[i].y == points[j].y) duplicates ++ ;
    }

    for (int j = i + 1; j < points.size(); j ++ )
        if (points[i].x != points[j].x)
        {
            long double slope = (long double)(points[i].y - points[j].y)
/ (points[i].x - points[j].x);
            if (map[slope] == 0) map[slope] = 2;
            else map[slope] ++ ;
            res = max(res, map[slope] + duplicates);
        }

    res = max(res, verticals);
}
return res;
}
};

```

150. 逆波兰表达式求值

150. 逆波兰表达式求值

难度 中等

125

收藏

分享

切换为英文

关注

反馈

根据逆波兰表示法，求表达式的值。

有效的运算符包括 `+`、`-`、`*`、`/`。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

输入：["2", "1", "+", "3", "*"]

输出：9

解释：((2 + 1) * 3) = 9

示例 2：

输入：["4", "13", "5", "/", "+"]

输出：6

解释：(4 + (13 / 5)) = 6

示例 3：

输入：["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]

输出：22

解释：



```
((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> sta;
        for (auto &t : tokens)
            if (t == "+" || t == "-" || t == "*" || t == "/")
            {
                int a = sta.top();
                sta.pop();
                int b = sta.top();
                sta.pop();
                if (t == "+") sta.push(a + b);
                else if (t == "-") sta.push(b - a);
                else if (t == "*") sta.push(a * b);
                else sta.push(b / a);
            }
            else sta.push(atoi(t.c_str()));
        return sta.top();
    }
}
```

```
};
```

151. 翻转字符串里的单词

151. 翻转字符串里的单词

难度 中等  161  收藏  分享  切换为英文  关注  反馈

给定一个字符串，逐个翻转字符串中的每个单词。

示例 1:

输入: "the sky is blue"
输出: "blue is sky the"

示例 2:

输入: " hello world! "
输出: "world! hello"
解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。

示例 3:

输入: "a good example"
输出: "example good a"
解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

说明:

- 无空格字符构成一个单词。
- 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
- 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

进阶:

请选用 C 语言的用户尝试使用 $O(1)$ 额外空间复杂度的原地解法。

```
class Solution {
public:
    string reverseWords(string s) {
        int k = 0;
        for(int i = 0; i < s.size(); i++)
        {
            int j = i;
            while(j < s.size() && s[j] == ' ') j++;
            if(j == s.size()) break;
            i = j;
            while(j < s.size() && s[j] != ' ') j++;
            reverse(s.begin() + i, s.begin() + j);
            if(k) s[k++] = ' ';
            while(i < j) s[k++] = s[i++];
        }
    }
};
```

```
s.erase(s.begin() + k,s.end());
reverse(s.begin(),s.end());

return s;
}
};
```

152. 乘积最大子数组

152. 乘积最大子数组

难度 中等 466 分享 切换为英文 关注 反馈

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字）。

示例 1:

输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。

示例 2:

输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2，因为 [-2,-1] 不是子数组。







```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int pos = 0, neg = 0;
        int res = INT_MIN;
        for (auto x : nums)
        {
            if (x > 0)
            {
                if (pos > 0) pos *= x;
                else pos = x;
                neg *= x;
            }
            else if (x < 0)
            {
                int np = pos, ng = neg;
                np = neg * x;
                if (pos > 0) ng = pos * x;
                else ng = x;
                pos = np, neg = ng;
            }
            else
            {
                pos = neg = 0;
            }
            res = max(res, x);
            if (pos != 0) res = max(res, pos);
        }
    }
};
```



```
    }  
    return res;  
}  
};
```

153. 寻找旋转排序数组中的最小值

153. 寻找旋转排序数组中的最小值

难度 中等  171  收藏  分享  切换为英文  关注  反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0, 1, 2, 4, 5, 6, 7]` 可能变为 `[4, 5, 6, 7, 0, 1, 2]`)。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

输入: `[3,4,5,1,2]`
输出: `1`

示例 2:

输入: `[4,5,6,7,0,1,2]`
输出: `0`

```
class Solution {  
public:  
    int findMin(vector<int>& nums) {  
        int l = 0, r = nums.size() - 1;  
        while(l < r)  
        {  
            int mid = l + r >> 1;  
            if(nums[mid] <= nums.back()) r = mid;  
            else l = mid + 1;  
        }  
        return nums[r];  
    }  
};
```

154. 寻找旋转排序数组中的最小值 II

154. 寻找旋转排序数组中的最小值 II

难度 困难

97

收藏

分享

切换为英文

关注

反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0, 1, 2, 4, 5, 6, 7]` 可能变为 `[4, 5, 6, 7, 0, 1, 2]`)。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1:

输入: `[1, 3, 5]`

输出: `1`

示例 2:

输入: `[2, 2, 2, 0, 1]`

输出: `0`

说明:

- 这道题是 [寻找旋转排序数组中的最小值](#) 的延伸题目。
- 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int l = 0, r = nums.size() - 1;
        while (l < r && nums[r] == nums[l]) r -- ;
        // 其实这块写 r - l + 1 < 2 就OK，不过思考的时候可以偷个懒，直接把小于5的全部特盘掉了。
        if (r - l + 1 < 5)
        {
            int res = INT_MAX;
            for (int x : nums) res = min(res, x);
            return res;
        }
        if (nums[0] <= nums[r]) return nums[0];
        else
        {
            int end = nums[r];
            while (l < r)
            {
                int mid = (l + r + 1) / 2;
                if (nums[mid] >= nums[0]) l = mid;
                else r = mid - 1;
            }
            return nums[r + 1];
        }
    }
};
```

155. 最小栈

155. 最小栈

难度 简单

👍 446

❤ 收藏

🔗 分享

🌐 切换为英文

🔔 关注

📝 反馈

设计一个支持 `push` , `pop` , `top` 操作，并能在常数时间内检索到最小元素的栈。

- `push(x)` —— 将元素 `x` 推入栈中。
- `pop()` —— 删除栈顶的元素。
- `top()` —— 获取栈顶元素。
- `getMin()` —— 检索栈中的最小元素。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

```
class MinStack {
public:
    stack<int> stk,stk_min;
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        stk.push(x);
        if(stk.empty() || stk_min.top() >= x)
            stk_min.push(x);
    }

    void pop() {
        if(stk_min.top() == stk.top()) stk_min.pop();
        stk.pop();
    }

    int top() {
        return stk.top();
    }

    int getMin() {
        return stk_min.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

160. 相交链表

160. 相交链表

难度 简单

619

收藏

分享

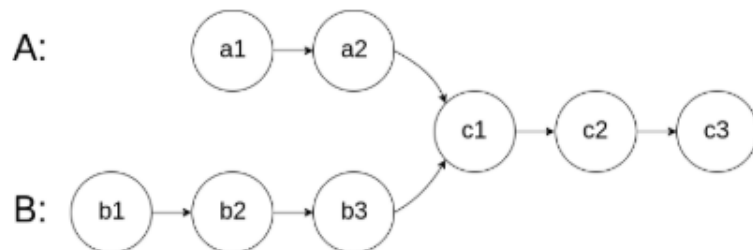
切换为英文

关注

反馈

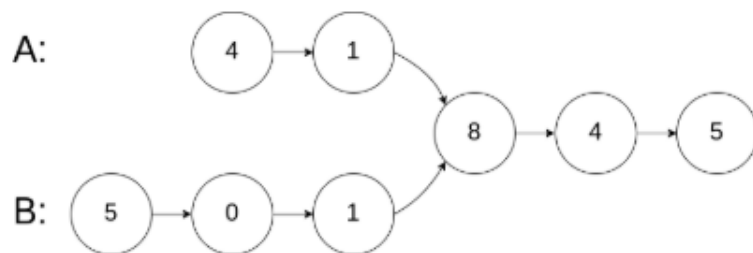
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1:



输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3

输出：Reference of the node with value = 8

输入解释：相交节点的值为 8（注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        auto p = headA, q = headB;
        while(p != q)
        {
            if(p) p = p->next;
            else p = headB;
            if(q) q = q->next;
            else q = headA;
        }
        return p;
    }
};

```

```
        else q = headA;  
    }  
    return p;  
}  
};
```