1、数据结构：链表

是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度。

单向链表

链表中最简单的一种是单向链表，它包含两个域，一个信息域和一个指针域。这个链接指向列表中的下一个节点，而最后一个节点则指向一个空值。

一个单向链表的节点被分成两个部分。第一个部分保存或者显示关于节点的信息，第二个部分存储下一个节点的地址。单向链表只可向一个方向遍历。

单向链表是一种线性表，实际上是由节点（Node）组成的，一个链表拥有不定数量的节点。其数据在内存中存储是不连续的，它存储的数据分散在内存中，每个结点只能也只有它能知道下一个结点的存储位置。由N各节点（Node）组成单向链表，每一个Node记录本Node的数据及下一个Node。向外暴露的只有一个头节点（Head），我们对链表的所有操作，都是直接或者间接地通过其头节点来进行的。

双向链表

一种更复杂的链表是“双向链表”或“双面链表”。每个节点有两个连接：一个指向前一个节点，（当此“连接”为第一个“连接”时，指向空值或者空列表）；而另一个指向下一个节点，（当此“连接”为最后一个“连接”时，指向空值或者空列表）

一个双向链表有三个整数值: 数值, 向后的节点链接, 向前的节点链接

循环链表

在一个 循环链表中, 首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。要转换一个循环链表，你开始于任意一个节点然后沿着列表的任一方向直到返回开始的节点。再来看另一种方法，循环链表可以被视为“无头无尾”。这种列表很利于节约数据存储缓存， 假定你在一个列表中有一个对象并且希望所有其他对象迭代在一个非特殊的排列下。

指向整个列表的指针可以被称作访问指针。

用单向链表构建的循环链表

2、Java中的链表

Java中的链表都是双向链表，相关的实现类有LinkedList：

public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, java.io.Serializable

由于LinkedList实现了Deque和Queue接口，可以按照队列、栈和双端队列的方式进行操作：

LinkedList作为队列：

1、在尾部添加元素 (add, offer)：

add()会在长度不够时抛出异常：IllegalStateException; offer()则不会,只返回false

2、查看头部元素 (element, peek)，返回头部元素，但不改变队列

element()会在没元素时抛出异常：NoSuchElementException; peek()返回null;

3、删除头部元素 (remove, poll)，返回头部元素，并且从队列中删除

remove()会在没元素时抛出异常：NoSuchElementException; poll()返回null;

LinkedList作为双端队列：

LinkedList作为栈：

3、链表的相关操作

操作一个链表只需要知道它的头指针就可以做任何操作了。

/**

- Definition for singly-linked list.
- public class ListNode {
- int val;
- ListNode next;
- ListNode(int x) { val = x; }
- } */ 1、链表的反转：（迭代法） public ListNode reverseList(ListNode head) { if（head==null || head.next==null） return head; ListNode prev = null; ListNode curr = head; while (curr != null) { ListNode nextTemp = curr.next; curr.next = prev; prev = curr; curr = nextTemp; } return prev; } 复杂度分析 时间复杂度：O(n)，假设 n 是列表的长度，时间复杂度是 O(n)。 空间复杂度：O(1)。 2、链表的排序：（https://leetcode-cn.com/problems/sort-list/solution/ ） 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。 根据时间复杂度的要求，可以想到归并排序。但是归并排序递归调用的空间复杂度为o(n),因此不能使用递归，采用从底至顶直接合并的方式。 bottom-to-up 的归并思路是这样的：先两个两个的 merge，完成一趟后，再 4 个4个的 merge，直到结束。举个简单的例子：[4,3,1,7,8,9,2,11,5,6]. step=1: (3->4)->(1->7)->(8->9)->(2->11)->(5->6) step=2: (1->3->4->7)->(2->8->9->11)->(5->6) step=4: (1->2->3->4->7->8->9->11)->5->6 step=8: (1->2->3->4->5->6->7->8->9->11) class Solution { public ListNode sortList(ListNode head) { if(head==null || head.next==null) return head; ListNode fast = head.next; ListNode slow = head; while(fast!=null && fast.next!=null){ slow = slow.next; fast = fast.next.next; } ListNode right = slow.next; slow.next = null; return merge(sortList(head),sortList(right)); } private ListNode merge(ListNode l1,ListNode l2){ ListNode head = new ListNode(0); ListNode cur = head; while(l1!=null || l2!=null){ if(l1==null ||(l2!=null && l1.val>l2.val)){ cur.next = l2; l2 = l2.next; }else{ cur.next = l1; l1 = l1.next; } cur = cur.next; } return head.next; } } 3、判断链表是否有环：（https://leetcode-cn.com/problems/linked-list-cycle/ ） 方法一：哈希表 思路 我们可以通过检查一个结点此前是否被访问过来判断链表是否为环形链表。常用的方法是使用哈希表。 算法 我们遍历所有结点并在哈希表中存储每个结点的引用（或内存地址）。如果当前结点为空结点 null（即已检测到链表尾部的下一个结点），那么我们已经遍历完整个链表，并且该链表不是环形链表。如果当前结点的引用已经存在于哈希表中，那么返回 true（即该链表为环形链表）。 public class Solution { public boolean hasCycle(ListNode head) { //直接法： ListNode curr=head; if(head==null||head.next==null) return false; List<ListNode> nodes= new ArrayList(); while(curr!=null){ if(!nodes.contains(curr)){ nodes.add(curr); curr=curr.next; }else{ return true; } } return false; } } 方法二：双指针 通过使用具有 不同速度 的快、慢两个指针遍历链表，空间复杂度可以被降低至 O(1)。慢指针每次移动一步，而快指针每次移动两步。如果有环，最终的slow和fast指针会相遇。 public boolean hasCycle(ListNode head) {if(head==null||head.next==null) return false; ListNode slow=head; ListNode

fast=head.next; while(slow!=fast) { if(fast==null || fast.next==null) { return false; } slow=slow.next; fast=fast.next.next; } return true; } 复杂度分析 时间复杂度：O(n)，让我们将 nn 设为链表中结点的总数。为了分析时间复杂度，我们分别考虑下面两种情况。 链表中不存在环： 快指针将会首先到达尾部，其时间取决于列表的长度，也就是 O(n)。 链表中存在环： 我们将慢指针的移动过程划分为两个阶段：非环部分与环形部分： 慢指针在走完非环部分阶段后将进入环形部分：此时，快指针已经进入环中 \text{迭代次数} = \text{非环部分长度} = N迭代次数=非环部分长度=N 两个指针都在环形区域中：考虑两个在环形赛道上的运动员 - 快跑者每次移动两步而慢跑者每次只移动一步。其速度的差值为 1，因此需要经过 \dfrac{\text{二者之间距离}}{\text{速度差值}} 速度差值 二者之间距离 次循环后，快跑者可以追上慢跑者。这个距离几乎就是 "\text{环形部分长度 K}环形部分长度 K" 且速度差值为 1，我们得出这样的结论 \text{迭代次数} = \text{近似于}迭代次数=近似于 "\text{环形部分长度 K}环形部分长度 K". 因此，在最糟糕的情形下，时间复杂度为 O(N+K)O(N+K)，也就是 O(n)O(n)。 空间复杂度：O(1)，我们只使用了慢指针和快指针两个结点，所以空间复杂度为 O(1)。 4、删除链表中倒数第k个节点：（https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/ ）方法一：两次遍历算法，我们注意到这个问题可以容易地简化成另一个问题：删除从列表开头数起的第 (L - n + 1)(个结点，其中 LL 是列表的长度。只要我们找到列表的长度 LL，这个问题就很容易解决。 public ListNode removeNthFromEnd(ListNode head, int n) { ListNode dummy = new ListNode(0); dummy.next = head; int length = 0; ListNode first = head; while (first != null) { length++; first = first.next; } length -= n; first = dummy; while (length > 0) { length--; first = first.next; } first.next = first.next.next; return dummy.next; }

复杂度分析

时间复杂度：O(L)，该算法对列表进行了两次遍历，首先计算了列表的长度 LL 其次找到第 (L - n)个结点。 操作执行了 2L-n2L−n 步，时间复杂度为 O(L)。

空间复杂度：O(1)，我们只用了常量级的额外空间。

方法二：一次遍历算法，双指针

/**

- Definition for singly-linked list.
- public class ListNode {
- int val;
- ListNode next;
- ListNode(int x) { val = x; }
- } */ class Solution { public ListNode removeNthFromEnd(ListNode head, int n) { ListNode dummy = new ListNode(0); dummy.next = head; ListNode first = dummy; ListNode second = dummy; // Advances first pointer so that the gap between first and second is n nodes apart while(n>0){ first=first.next; n--; } // Move first to the end, maintaining the gap while (first.next != null) { first = first.next; second = second.next; } second.next = second.next.next; return dummy.next; }

}

复杂度分析

时间复杂度：O(L)，该算法对含有 LL 个结点的列表进行了一次遍历。因此时间复杂度为 O(L)。

空间复杂度：O(1)，我们只用了常量级的额外空间。

5、给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。(https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/)

```
public ListNode deleteDuplicates(ListNode head) {
ListNode current = head;
while (current != null && current.next != null) {
if (current.next.val == current.val) {
current.next = current.next.next;
} else {
current = current.next;
}
}
return head;
}
```

复杂度分析

时间复杂度：O(n)，因为列表中的每个结点都检查一次以确定它是否重复，所以总运行时间为 O(n)O(n)，其中 nn 是列表中的结点数。

空间复杂度：O(1)，没有使用额外的空间。

6、编写一个程序，找到两个单链表相交的起始节点（ https://leetcode-cn.com/problems/intersection-of-two-linked-lists/ ）

方法一: 暴力法

对链表A中的每一个结点 a_ia ，遍历整个链表 B 并检查链表 B 中是否存在结点和 a_i相同。

复杂度分析

时间复杂度 : (mn)(mn)。

空间复杂度 : O(1)O(1)。

方法二: 哈希表法

遍历链表 A 并将每个结点的地址/引用存储在哈希表中。然后检查链表 B 中的每一个结点 b_i是否在哈希表中。若在，则 b_i为相交结点。

复杂度分析

时间复杂度 : O(m+n)O。

空间复杂度 : O(m)或 O(n)。

方法三：双指针法

创建两个指针 pApA 和 pBpB，分别初始化为链表 A 和 B 的头结点。然后让它们向后逐结点遍历。

当 pA 到达链表的尾部时，将它重定位到链表 B 的头结点 (你没看错，就是链表 B); 类似的，当 pBpB 到达链表的尾部时，将它重定位到链表 A 的头结点。

若在某一时刻 pApA 和 pBpB 相遇，则 pApA/pBpB 为相交结点。

想弄清楚为什么这样可行, 可以考虑以下两个链表: A={1,3,5,7,9,11} 和 B={2,4,9,11}，相交于结点 9。

由于 B.length (=4) < A.length (=6)，pBpB 比 pApA 少经过 22 个结点，会先到达尾部。将 pBpB 重定向

到 A 的头结点，pApA 重定向到 B 的头结点后，pBpB 要比 pApA 多走 2 个结点。因此，它们会同时到达交点。

如果两个链表存在相交，它们末尾的结点必然相同。因此当 pA/pB 到达链表结尾时，记录下链表 A/B 对应的元素。若最后元素不相同，则两个链表不相交。

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

```
    if (headA == null || headB == null)     return null;
    ListNode a = headA, b = headB;
    while(a != b) {
        a = (a == null ? headB : a.next);
        b = (b == null ? headA : b.next);
    }
    return a;
}

```

}

复杂度分析

时间复杂度 : O(m+n)。

空间复杂度 : O(1)。

7、链表的遍历：

public void getNode(ListNode head){

ListNode curr = head;

while(curr!=null){

System.out.println(curr.data);

curr=curr.next;

}

8、查询链表的中间节点（https://leetcode-cn.com/problems/middle-of-the-linked-list/ ）

方法一：输出到数组

思路和算法

按顺序将每个结点放入数组 A 中。然后中间结点就是 A[A.Length/2]，因为我们可以通过索引检索每个结点。

C++JavaPythonJavaScript

class Solution {

public ListNode middleNode(ListNode head) {

ListNode[] A = new ListNode[100];

ListNode curr=head;

```
int t = 0;
while (curr != null) {
A[t++] = curr;
curr= curr.next;
}
return A[t / 2];
}
}
```

复杂度分析

时间复杂度：O(N)，其中 NN是给定列表中的结点数目。

空间复杂度：O(N)，A 用去的空间。

方法二：快慢指针法

思路和算法

当用慢指针 slow 遍历列表时，让另一个指针 fast 的速度是它的两倍。

当 fast 到达列表的末尾时，slow 必然位于中间。

```
class Solution {
public ListNode middleNode(ListNode head) {
ListNode slow = head, fast = head;
while (fast != null && fast.next != null) {
slow = slow.next;
fast = fast.next.next;
}
return slow;
}
}
```
