原 java初、中、高级面试题必备——数据结构与算法-链表

2019年06月25日 14:08:58 在IT中穿梭旅行 阅读数 80 编辑

链表

166 链表

数据结构 —— 链表。链表是一种线性数据结构

正如你所看到的，链表中的每个元素实际上是一个单独的对象，而所有对象都通过每个元素中的引用字段链接在一起

链表有两种类型：**单链表和双链表**。上面给出的例子是一个单链表，这里有一个双链表的例子：

167. 单链表

单链表中的**每个结点不仅包含值，还包含链接到下一个结点的引用字段**。通过这种方式，单链表将所有结点按顺序组织起来。

蓝色箭头显示单个链接列表中的结点是如何组合在一起的。 下面是单链表定义：

```
1 public class SinglyListNode {
```

```
    int val;
    SinglyListNode next;
    SinglyListNode(int x) { val = x; }
}
```

在大多数情况下，我们将使用头结点(第一个结点)来表示整个列表。

操作:

与数组不同，我们无法在常量时间内访问单链表中的随机元素。 如果我们想要获得第 i 个元素，我们必须从头结点逐个遍历。 我们按**索引**来**访问元素**平均要花费 O(N) 时间，其中 N 是链表的长度。

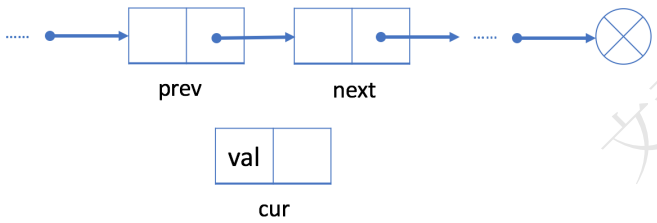
例如，在上面的示例中，头结点是 23。访问第 3 个结点的唯一方法是使用头结点中的“next”字段到达第 2 个结点（结点 6）；然后使用结点 6 的“next”字段，我们能够访问第 3 个结点。

下面介绍单链表的**插入**和**删除**操作，你将了解到链表的好处

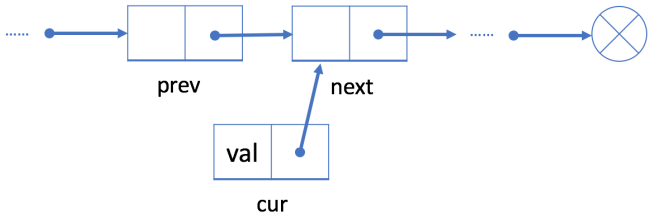
167.添加操作 - 单链表

如果我们想在给定的结点 prev 之后添加新值，我们应该：

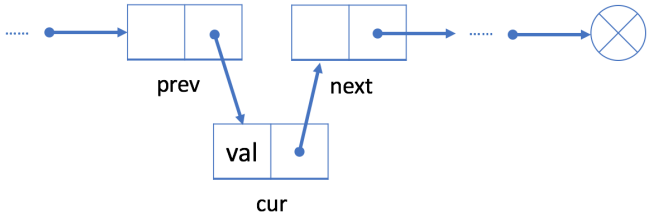
- 使用给定值初始化新结点 cur；



- 将 cur 的“next”字段链接到 prev 的下一个结点 next；

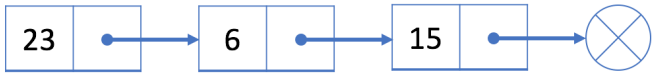


- 将 prev 中的“next”字段链接到 cur 。



与数组不同，我们不需要将所有元素移动到插入元素之后。因此，您可以在 **O(1)** 时间复杂度中将新结点插入到链表中，这非常高效。

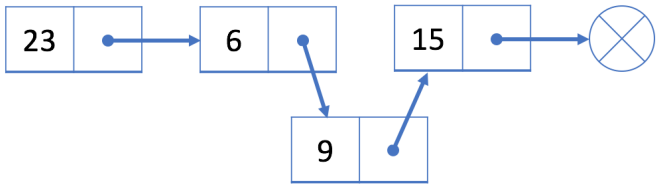
示例：



让我们在第二个结点 6 之后插入一个新的值 9。

我们将首先初始化一个值为 9 的新结点。然后将结点 9 链接到结点 15。最后，将结点 6 链接到结点 9。

插入之后，我们的链表将如下所示：



在开头添加结点：

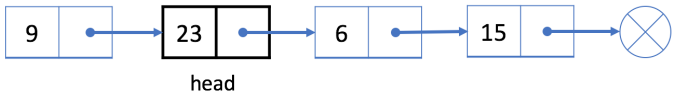
众所周知，我们使用头结点来代表整个列表。

因此，在列表开头添加新节点时更新头结点 **head** 至关重要。

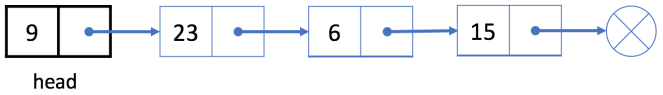
1. 初始化一个新结点 **cur**；
2. 将新结点链接到我们的原始头结点 **head**。
3. 将 **cur** 指定为 **head**。

例如，让我们在列表的开头添加一个新结点 9。

- 我们初始化一个新结点 9 并将其链接到当前头结点 23。



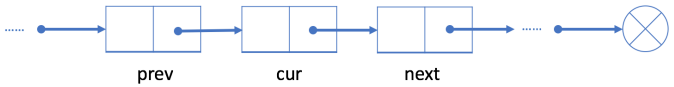
- 指定结点 9 为新的头结点。



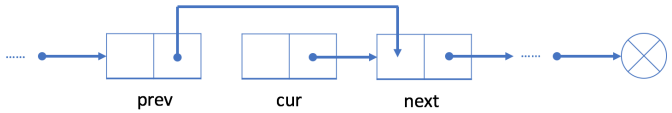
168.删除操作 - 单链表

如果我们想从单链表中删除现有结点 **cur**，可以分两步完成：

- 找到 **cur** 的上一个结点 **prev** 及其下一个结点 **next**；



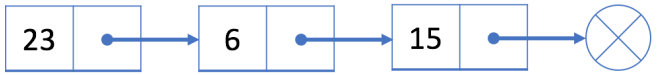
- 接下来链接 **prev** 到 cur 的下一个节点 **next**。



在我们的第一步中，我们需要找出 **prev** 和 **next**。使用 **cur** 的参考字段很容易找出 **next**，但是，我们必须从头结点遍历链表，以找出 **prev**，它的平均时间是 **O(N)**，其中 N 是链表的长度。因此，删除结点的时间复杂度将是 **O(N)**。

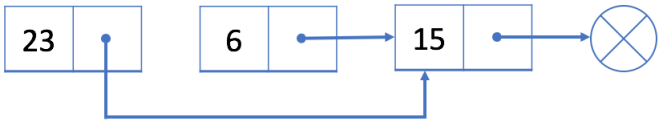
空间复杂度为 **O(1)**，因为我们只需要常量空间来存储指针。

示例



让我们尝试把结点 6从上面的单链表中删除。

1. 从头遍历链表，直到我们找到前一个结点 `prev`，即结点 23
2. 将 `prev`（结点 23）与 `next`（结点 15）链接

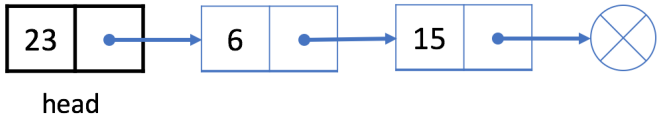


结点 6 现在不在我们的单链表中。

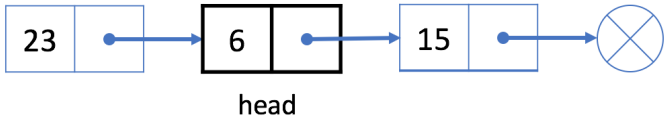
删除第一个结点

如果我们想删除第一个结点，策略会有所不同。

正如之前所提到的，我们使用头结点 **head** 来表示链表。我们的头是下面示例中的黑色结点 **23**。



如果想要删除第一个结点，我们可以简单地**将下一个结点分配给 head**。也就是说，删除之后我们的头将会是结点 6。



链表从头结点开始，因此结点 23 不再在我们的链表中。

169.关于单链表的算法题

1.题目描述

输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

```
/**
*    public class ListNode {
*        int val;
*        ListNode next = null;
*
*        ListNode(int val) {
*            this.val = val;
*        }
*    }
*
*/
import java.util.Stack;
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        Stack<Integer> stack = new Stack<>();
```

```
        while (listNode != null) {18            stack.push(listNode.val);
            listNode = listNode.next;
        }

        ArrayList<Integer> list = new ArrayList<>();
        while (!stack.isEmpty()) {
            list.add(stack.pop());
        }
        return list;
    }
}
```

2. 输入一个链表，反转链表后，输出新链表的表

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
    public ListNode ReverseList(ListNode head) {
        ListNode pre = null;
        ListNode curr = head;
        while(curr!=null){
            ListNode nextNode = curr.next;
            curr.next = pre;
            pre = curr;
             curr = nextNode;
        }
        return pre;
    }
```

```
}
```

3.给定一个链表，判断链表中是否有环。

使用**双指针技巧**有一个更有效的解决方案。

设置快慢指针，让快指针每次移动两步，慢指针每次移动一步，，经过 *M* 次迭代后，快指针肯定会多绕环一周，并赶上慢指针。

1. 如果没有环，快指针将停在链表的末尾。
2. 如果有环，快指针最终将与慢指针相遇。

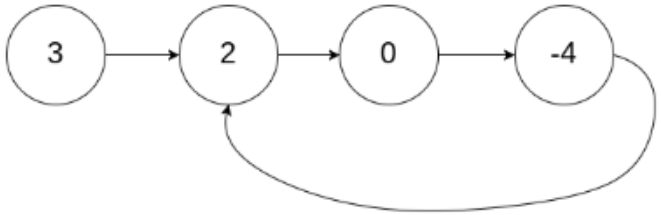
算法实现：

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if(head==null||head.next==null){
            return false;
        }
        ListNode p1 = head;
        ListNode p2 = head;
        while(p2!=null&&p2.next!=null){
            p1 = p1.next;
            p2 = p2.next.next;
            if(p1==p2){
```

```
            return true;
        }
        return false;
    }
}
```

4.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。

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。



这道题和上一道题类似，解题思路如下：让快指针移动两步，慢指针移动一步，如果有环时，慢指针和快指针肯定相交，这时候让慢指针从头指针重新开始走，快指针顺着往下走，快慢指针每次都走一步，最终交替的地方就是入环的第一个节点。

代码如下：

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
```

```
 * }
 */
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head==null||head.next==null){
            return null;
        }
        ListNode slow = head;
        ListNode fast = head;
        while(fast!=null&&fast.next!=null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow==fast){
                slow =head;
                while(slow!=fast){
                    slow = slow.next;
                    fast = fast.next;
                }
                if(slow==fast) return slow;
            }
        }
        return null;
    }
}
```

5.输入两个链表，找出它们的第一个公共结点。

解题思路：1.先求出两个链表分别的长度，然后长度做减法运算
2，如果A -B > 0 说明A链表长于B链表， 则让A链表先往前移动(A-B)步，这样A和B就保持到同一起点位置
然后比较同一起点位置是否相同，如果不相同，让两个链表保持前进，直到相同为止。
3 如果A -B < 0 说明A链表短于B链表， 则让B链表先往前移动(A-B)步，这样A和B就保持到同一起点位置
然后比较同一起点位置是否相同，如果不相同，让两个链表保持前进，直到相同为止。

```
/**
 * Definition for singly-linked list.
```

```
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA==null||headB==null){
            return null;
        }
        ListNode p1 = headA;
        ListNode p2 = headB;

        int count1 =0;
        while(p1!=null){
            p1 =p1.next;
            count1++;
        }
        int count2 = 0 ;
        while(p2!=null){
            p2 = p2.next;
            count2 ++;
        }
        int flag = count1-count2;
        if(flag>0){
            while(flag>0){
                headA =headA.next;
                flag --;
            }
            while(headA!=headB){
                headA =headA.next;
                headB =headB.next;
            }
            return headA;
```

```
        }
         if(flag<=0){
            while(flag<0){
                headB = headB.next;
                flag++;
            }
            while(headA!=headB){
                headB = headB.next;
                headA = headA.next;

            }
            return headA;
        }
        return null;

    }
}
```

6.删除链表的倒数第N个节点

解题思路：1．使用快慢指针，让快指针先走n步，
2 然后在让快慢指针一起走，直到快指针走到尾节点，则慢指针所在的位置就是要删除的倒数第N个节点．
3．让慢指针所在位置的下一个节点指向才在位置的节点，则删除。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
```

```
public class Test24 {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode fast = head;
        while (n-- >0){
            fast = fast.next;
        }
        ListNode slow = head;
        if (fast == null) return head.next;
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return head;
    }
}
```

7.在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，

重复的结点不保留，返回链表头指针。
例如，链表1->2->3->3->4->4->5 处理后为 1->2->5
解题思路：　利用递归，让下一个节点和当前结点的值比较，不一样，返回当前结点的值，然后继续下一步。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution{
```

```
13
14      public ListNode deleteDuplication(ListNode pHead) {
15          if (pHead == null || pHead.next == null)
16              return pHead;
17          ListNode next = pHead.next;
18          if (pHead.val == next.val) {
19              while (next != null && pHead.val == next.val)
20                  next = next.next;
21              return deleteDuplication(next);
22          } else {
23              pHead.next = deleteDuplication(pHead.next);
24              return pHead;
25          }
26      }
27  }
```

170.介绍双链表

双链表以类似的方式工作，但还有一个引用字段，称为**“prev”**字段。有了这个额外的字段，您就能够知道当前结点的前一个结点。

让我们看一个例子：

绿色箭头表示我们的“prev”字段是如何工作的。

双链表中结点结构的典型定义：

```
1  class DoublyListNode {
2      int val;
3      DoublyListNode next, prev;
4      DoublyListNode(int x) {val = x;}
5  }
```

与单链接列表类似，我们将使用**头结点**来表示整个列表。

操作

与单链表类似，我们将介绍在双链表中如何访问数据、插入新结点或删除现有结点。

我们可以与单链表相同的方式访问数据：

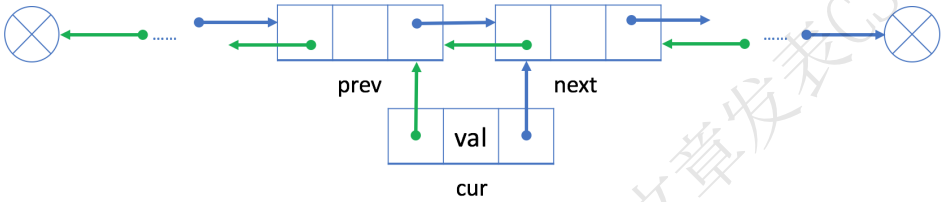
1. 我们不能在常量级的时间内**访问随机位置。**
2. 我们必须从头部遍历才能得到我们想要的第一个结点。
3. 在最坏的情况下，时间复杂度将是 `O(N)`，其中 `N` 是链表的长度。

对于添加和删除，会稍微复杂一些，因为我们还需要处理“prev”字段。在接下来的两篇文章中，我们将介绍这两个操作

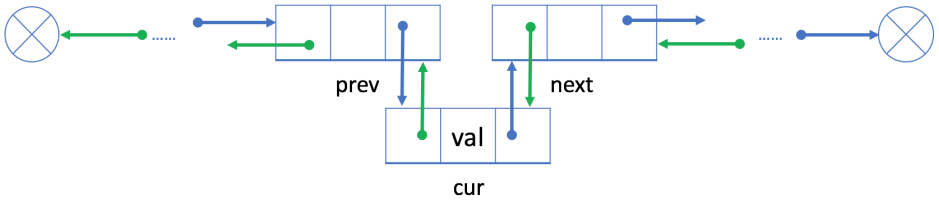
171.添加双链表

如果我们想在现有的结点 `prev` 之后插入一个新的结点 `cur`，我们可以将此过程分为两个步骤：

- 链接 `cur` 与 `prev` 和 `next`，其中 `next` 是 `prev` 原始的下一个节点；



- 用 `cur` 重新链接 `prev` 和 `next`。

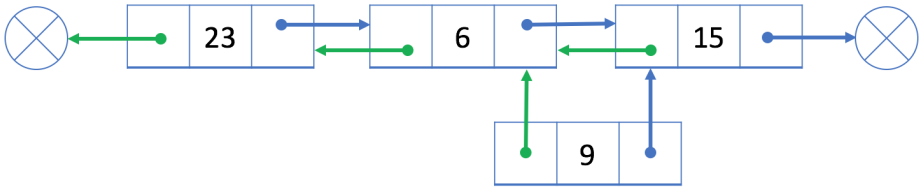


与单链表类似，添加操作的时间和空间复杂度都是 O(1)。

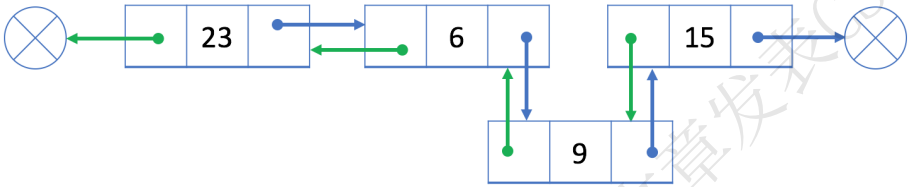
示例

让我们在现有结点 6 之后添加一个新结点 9：

1. 链接 cur（结点 9）与 prev（结点 6）和 next（结点 15）



2.用 cur（结点 9）重新链接 prev（结点 6）和 next（结点 15



172.添加双链表

如果我们想从双链表中删除一个现有的结点 **cur**，我们可以简单地将它的前一个结点 **prev** 与下一个结点 **next** 链接起来。

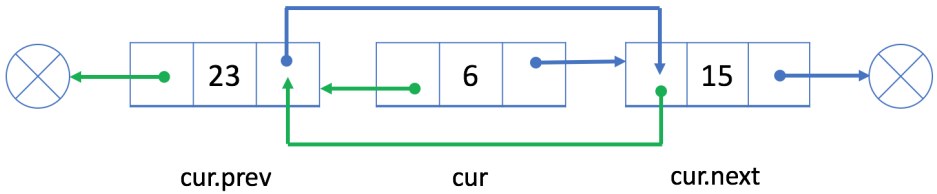
> 与单链表不同，使用“prev”字段可以很容易地在常量时间内获得前一个结点。

因为我们不再需要遍历链表来获取前一个结点，所以时间和空间复杂度都是0(1)。

示例

我们的目标是从双链表中删除结点 6。

因此，我们将它的前一个结点 23 和下一个结点 15 链接起来：



结点 6 现在不在我们的双链表中。

数组链接：https://blog.csdn.net/weixin_38201936/article/details/93628729