Interview Questions

2.1 **Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

Hints: #9, #40

Pg 208

2.1 **Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

pg 94

SOLUTION

In order to remove duplicates from a linked list, we need to be able to track duplicates. A simple hash table will work well here.

In the below solution, we simply iterate through the linked list, adding each element to a hash table. When we discover a dupli-cate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list.

```
void deleteDups(LinkedListNode n) {
   HashSet<Integer> set = new HashSet<Integer>();
   LinkedListNode previous = null;
   while (n != null) {
      if (set.contains(n.data)) {
         previous.next = n.next;
      ) else {
         set.add(n.data);
         previous = n;
      )
      n = n.next;
   )
)
```

The above solution takes O(N) time, where N is the number of elements in the linked list.

Follow Up: No Buffer Allowed

If we don't have a buffer, we can iterate with two pointers: `current` which iterates through the linked list, and `runner` which checks all subsequent nodes for duplicates.

```
void deleteDups(LinkedListNode head) {
   LinkedListNode current = head;
   while (current != null) {
      /* Remove all future nodes that have the same value */
      LinkedListNode runner = current;
      while(runner.next     != null)       {
         if (runner.next.data == current.data) {
                runner.next = runner.next.next;
            } else {
                runner = runner.next;
            }
        }
        current = current.next;
    }
}
```

This code runs in `O(1)` space, but `O(N²)` time.

pg 211

2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

```
Input:   3 -> 5 ->  8 -> 5 -> 10 -> 2 -> 1 [partition = 5]
Output:  3 -> 1 -> 2 ->  10 -> 5 -> 5 -> 8
```

Hints: #3, #24

pg 212

2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: `3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1` [partition = 5]

Output: `3 -> 1 -> 2 -> 10 -> 5 -> 5 -> 8`

pg 94

SOLUTION

If this were an array, we would need to be careful about how we shifted elements. Array shifts are very expensive.

However, in a linked list, the situation is much easier. Rather than shifting and swapping elements, we can actually create two different linked lists: one for elements less than x, and one for elements greater than or equal to x.

We iterate through the linked list, inserting elements into our `before` list or our `after` list. Once we reach the end of the linked list and have completed this splitting, we merge the two lists.

This approach is mostly "stable" in that elements stay in their original order, other than the necessary movement around the parti-tion. The code below implements this approach.

```
/* Pass in the head of the linked list and the value to partition around */
LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode beforeStart = null;
   LinkedListNode beforeEnd = null;
   LinkedListNode afterStart = null;
   LinkedListNode afterEnd = null;

   /* Partition list */
   while (node != null) {
      LinkedListNode next = node.next;
      node.next = null;
      if (node.data < x) {
         /* Insert node into end of before list */
         if (beforeStart == null) {
            beforeStart = node;
            beforeEnd = beforeStart;
         } else {
            beforeEnd.next = node;
            beforeEnd = node;
         }
      } else {
         /* Insert node into end of after list */
         if (afterStart == null) {
            afterStart = node;
            afterEnd = afterStart;
         } else {

            afterEnd.next = node;
            afterEnd = node;
         }
      }
      node = next;
   }

   if (beforeStart == null) {
      return afterStart;
   }

   /* Merge before list and after list */
   beforeEnd.next = afterStart;
   return beforeStart;
}
```

If it bugs you to keep around four different variables for tracking two linked lists, you're not alone. We can make this code a bit shorter.

If we don't care about making the elements of the list "stable"(which there's no obligation to, since the interviewer hasn't specified that), then we can instead rearrange the elements by growing the list at the head and tail.

In this approach, we start a "new" list (using the existing nodes). Elements bigger than the pivot element are put at the tail and elements smaller are put at the head. Each time we insert an element, we update either the head or tail.

```
LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode head = node;
   LinkedListNode tail = node;

   while (node != null) {
      LinkedListNode next = node.next;
      if (node.data < x) {
         /* Insert node at head. */
         node.next = head;
         head = node;
      } else {
         /* Insert node at tail. */
         tail.next = node;
         tail = node;
      }
      node = next;
   }
   tail.next = null;

   // The head has changed, so we need to return it to the user.
   return head;
}
```

There are many equally optimal solutions to this problem. If you came up with a different one, that's okay!

SOL

Hints: #7, #30, #71, #95, #109

pg 214

2.6 ✓ **Palindrome:** Implement a function to check if a linked list is a palindrome.

Hints: #5, #13, #29, #61, #101

pg 216

2.7 **Intersection:** Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the `j`th node of the second linked list, then they are intersecting.

2.6 Palindrome: Implement a function to check if a linked list is a palindrome.

pg 95

SOLUTION

To approach this problem, we can picture a palindrome like `0 -> 1 -> 2 -> 1 -> 0`. We know that, since it's a palindrome, the list must be the same backwards and forwards. This leads us to our first solution.

Solution #1: Reverse and Compare

Our first solution is to reverse the linked list and compare the reversed list to the original list. If they're the same, the lists are identical.

Note that when we compare the linked list to the reversed list, we only actually need to compare the first half of the list. If the first half of the normal list matches the first half of the reversed list, then the second half of the normal list must match the second half of the reversed list.

```
class HeadAndTail {
   public LinkedListNode head;
   public LinkedListNode tail;
   public HeadAndTail(LinkedListNode h, LinkedListNode t) {
      head = h;
      tail = t;
   }
}

boolean isPalindrome(LinkedListNode head) {
   HeadAndTail reversed = reverse(head);
   LinkedListNode reversedHead = reversed.head;
   return isEqual(head, reversedHead);
}

HeadAndTail reverse(LinkedistNode head) {
   if (head == null) {
      return null;
   }
   HeadAndTail ht = reverse(head.next);
   LinkedListNode cloneHead = head.clone();
   clonedHead.next = null;

   if (ht == null) {
      return new HeadAndTail(clonedHead, clonedHead);
   }
   ht.tail.next = clonedHead;
   return new HeadAndTail(ht.head, clonedHead);
}

boolean isEqual(LinkedListNode one, LinkedListNode two) {
   LinkedListNode head1 = one;
   LinkedListNode head2 = two;
   while (head1 != null && head2 != null) {
      if (head1.data != head2.data) {
         return false;
      }
      head1 = head1.next;
      head2 = head2.next;
   }
   return head1 == null && head2 == null
}
```

Observe that we've modularized this code into `reverse` and `isEqual` functions. We've also created a new class so that we can return both the head and the tail of this method. We could have also returned a two-element array, but that approach is less maintainable.

Solution #2: Iterative Approach

We want to detect linked lists where the front half of the list is the reverse of the second half. How would we do that? By reversing the front half of the list. A stack can accomplish this.

We need to push the first half of the elements onto a stack. We can do this in two different ways, depending on whether or not we know the size of the linked list.

If we know the size of the linked list, we can iterate through the first half of the elements in a standard for loop, pushing each element onto a stack. We must be careful, of course, to handle the case where the length of the linked list is odd.

If we don't know the size of the linked list, we can iterate through the linked list, using the fast runner / slow runner technique described in the beginning of the chapter. At each step in the loop, we push the data from the slow runner onto a stack. When the fast runner hits the end of the list, the slow runner will have reached the middle of the linked list. By this point, the stack will have all the elements from the front of the linked list, but in reverse order.

Now, we simply iterate through the rest of the linked list. At each iteration, we compare the node to the top of the stack. If we complete the iteration without finding a difference, then the linked list is a palindrome.

```
boolean isPalindrome(LinkedListNode head) {
   LinkedListNode fast = head;
   LinkedListNode slow = head;

   Stack<Integer> stack = new Stack<Integer>();

   /* Push elements from first half of linked list onto stack. When fast runner
    * (which is moving at 2x speed) reaches the end of the linked list, then we
    * know we're at the middle */
   while (fast != null && fast.next != null) {
      stack.push(slow.data);
      slow = slow. next;
      fast = fast.next.next;
   }

   /* Has odd number of elements, so skip the middle element */
   if (fast != null) {
      slow = slow.next;
   }

   while (slow != null) {
      int top = stack.pop().intValue();

      /* If values are different, then it's not a palindrome */
      if (top != slow.data) {
         return false;
      }
      slow = slow.next;
   }
   return true;
}
```

Solution #3: Recursive Approach

First, a word on notation: in this solution, when we use the notation node Kx, the variable K indicates the value of the node data, and x (which is either f or b) indicates whether we are referring to the front node with that value or the back node. For example, in the below linked list, node 2b would refer to the second (back) node with value 2.

Now, like many linked list problems, you can approach this problem recursively. We may have some intuitive idea that we want to compare element 0 and element n - 1, element 1 and element n-2, element 2 and element n-3, and so on, until the middle element(s). For example:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

In order to apply this approach, we first need to know when we've reached the middle element, as this will form our base case. We can do this by passing in `length - 2` for the `length` each time. When the length equals `0` or `1`, we're at the center of the linked list. This is because the length is reduced by 2 each time. Once we've recursed $N/2$ times, `length` will be down to 0.

```
recurse(Node n, int length) {
   if (length == 0 || length == 1) {
      return [something]; // At middle
   }
   recurse(n.next, length - 2);
   ...
}
```

This method will form the outline of the `isPalindrome` method. The "meat" of the algorithm though is comparing node `i` to node n - `i` to check if the linked list is a palindrome. How do we do that?

Let's examine what the call stack looks like:

```
1  v1 = isPalindrome: list = 0 ( 1 ( 2  ( 3 ) 2 )  1 ) 0. length = 7
2    v2 = isPalindrome: list = 1 ( 2 ( 3  ) 2 ) 1  ) 0. length = 5
3      v3 = isPalindrome: list = 2 ( 3 ) 2 )  1 ) 0. length = 3
4        v4 = isPalindrome: list = 3 ) 2 )  1 ) 0. length = 1
5        returns v3
6      returns v2
7    returns v1
8  returns ?
```

In the above call stack, each call wants to check if the list is a palindrome by comparing its head node with the corresponding node from the back of the list. That is:

- Line 1 needs to compare node `0f` with node `0b`
- Line 2 needs to compare node `1f` with node `1b`
- Line 3 needs to compare node `2f` with node `2b`
- Line 4 needs to compare node `3f` with node `3b`.

If we rewind the stack, passing nodes back as described below, we can do just that:

- Line 4 sees that it is the middle node (since `length = 1`), and passes back `head.next`. The value head equals node `3`, so `head.next` is node `2b`.
- Line 3 compares its head, node `2f`, to `returned_node` (the value from the previous recursive call), which is node `2b`. If the values match, it passes a reference to node `1b` (`returned_node.next`) up to line 2.
- Line 2 compares its head (node `1f`) to `returned_node` (node `1b`). If the values match, it passes a reference to node `0b` (or, `returned_node.next`) up to line 1.
- Line 1 compares its head, node `0f`, to `returned_node`, which is node `0b`. If the values match, it returns true.

To generalize, each call compares its head to `returned_node`, and then passes `returned_node.next` up the stack. In this way, every node `i` gets compared to node `n - i`. If at any point the values do not match, we return `false`, and every call up the stack checks for that value.

But wait, you might ask, sometimes we said we'll return a `boolean` value, and sometimes we're returning a `node`. Which is it?

It's both. We create a simple class with two members, a `boolean` and a `node`, and return an instance of that class.

```
class Result {
   public LinkedListNode node;
   public boolean result;
}
```

The example below illustrates the parameters and return values from this sample list.

```
isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
  isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
    isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
      isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
        isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
        returns node 3b, true
      returns node 2b, true
    returns node 1b, true
  returns node 0b, true
returns null, true
```

Implementing this code is now just a matter of filling in the details.

```
boolean isPalindrome(LinkedListNode head) {
   int length = lengthOfList(head);
   Result p = isPalindromeRecurse(head, length);
   return p.result;
}

Result isPalindromeRecurse(LinkedListNode head, int length) {
   if (head == null || length <= 0) { // Even number of nodes
      return new Result(head, true);
   } else if (length == 1) { // Odd number of nodes
      return new Result(head.next, true);
   }

   /* Recurse on sublist. */
   Result res = isPalindromeRecurse(head.next, length - 2);

   /* If child calls are not a palindrome, pass back up
    * a failure. */
   if (!res.result || res.node == null) {
      return res;
   }

   /* Check if matches corresponding node on other side. */
   res.result = (head.data == res.node.data);

   /* Return corresponding node. */
   res.node = res.node.next;

   return res;
}

int lengthOfList(LinkedListNode n) {
   int size = 0;
   while (n != null) {
      size++;
      n = n.next;
   }
   return size;
}
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```
bool isPalindromeRecurse(Node head, int length, Node** next) {
   ...
}
```

It's ugly, but it works.

2.7 **Intersection:** Given two (singly)