Of course, this is only a valid solution if the interviewer says it is valid.

Approach B: Use C++.

A second way to solve this is to use C++ and to pass values by reference. This allows us to return the node value, but also update the counter by passing a pointer to it.

```
node* nthToLast(node* head, int k, int& i) {
  if (head == NULL) {
    return NULL;
  }
  node* nd = nthToLast(head->next, k, i);
  i = i + 1;
  if (i == k) {
    return head;
  }
  return nd;
}

node* nthToLast(node* head, int k) {
  int i = 0;
  return nthToLast(head, k, i);
}
```

Approach C: Create a Wrapper Class.

We described earlier that the issue was that we couldn't simultaneously return a counter and an index. If we wrap the `counter` value with simple class (or even a single element array), we can mimic passing by reference.

```
class Index {
  public int value = 0;
}

LinkedListNode kthToLast(LinkedListNode head, int k) {
  Index idx = new Index();
  return kthToLast(head, k, idx);
}

LinkedListNode kthToLast(LinkedListNode head, int k, Index idx) {
  if (head == null) {
    return null;
  }
  LinkedListNode node = kthToLast(head.next, k, idx);
  idx.value = idx.value + 1;
  if (idx.value == k) {
    return head;
  }
  return node;
}
```

Each of these recursive solutions takes `O(n)` space due to the recursive calls.

There are a number of other solutions that we haven't addressed. We could store the counter in a static variable. Or, we could create a class that stores both the node and the counter, and return an instance of that class. Regardless of which solution we pick, we need a way to update both the node and the counter in a way that all levels of the recursive stack will see.

Solution #3: Iterative

A more optimal, but less straightforward, solution is to implement this iteratively. We can use two pointers, p1 and p2. We place them k nodes apart in the linked list by putting p2 at the beginning and moving p1 k nodes into the list. Then, when we move them at the same pace, p1 will hit the end of the linked list after `LENGTH - k` steps. At that point, p2 will be `LENGTH - k` nodes into the list, or k nodes from the end.

The code below implements this algorithm.

```
LinkedListNode nthToLast(LinkedListNode head, int k) {
  LinkedListNode p1 = head;
  LinkedListNode p2 = head;

  /* Move p1 k nodes into the list.*/
  for (int i = 0; i < k; i++) {
    if (p1 == null) return null; // Out of bounds
    p1 = p1.next;
  }

  /* Move them at the same pace. When p1 hits the end, p2 will be at the right
   * element. */
  while (p1 != null) {
    p1 = p1.next;
    p2 = p2.next;
  }
  return p2;
}
```

This algorithm takes `O(n)` time and `O(1)` space.

2.3 **Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node `c` from the linked list `a->b->c->d->e->f`

Result: nothing is returned, but the new linked list looks like `a->b->d->e->f`

pg 94

SOLUTION

In this problem, you are not given access to the head of the linked list. You only have access to that node. The solution is simply to copy the data from the next node over to the current node, and then to delete the next node.

The code below implements this algorithm.

```
boolean deleteNode(LinkedListNode n) {
  if (n == null || n.next == null) {
    return false; // Failure
  }
  LinkedListNode next = n.next;
  n.data = next.data;
  n.next = next.next;
  return true;
}
```

Note that this problem cannot be solved if the node to be deleted is the last node in the linked list. That's okay—your interviewer wants you to point that out, and to discuss how to handle this case. You could, for example, consider marking the node as dummy.

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 partition. 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!

2.5 **Sum Lists:** You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: `(7-> 1 -> 6) + (5 -> 9 -> 2)`. That is, `617 + 295`.

Output: `2 -> 1 -> 9`. That is, `912`.

FOLLOW UP

Suppose the digits are stored in forward order. Repeat the above problem.

Input: `(6 -> 1 -> 7) + (2 -> 9 -> 5)`. That is, `617 + 295`.

Output: `9 -> 1 -> 2`. That is, `912`.

pg 95

SOLUTION

It's useful to remember in this problem how exactly addition works. Imagine the problem:

```
  6 1 7
+ 2 9 5
```

First, we add 7 and 5 to get 12. The digit 2 becomes the last digit of the number, and 1 gets carried over to the next step. Second, we add 1, 1, and 9 to get 11. The 1 becomes the second digit, and the other 1 gets carried over the final step. Third and finally, we add 1, 6 and 2 to get 9. So, our value becomes 912.

We can mimic this process recursively by adding node by node, carrying over any "excess" data to the next node. Let's walk through this for the below linked list:

```
    7 -> 1 -> 6
+   5 -> 9 -> 2
```

We do the following:

1. We add 7 and 5 first, getting a result of 12. 2 becomes the first node in our linked list, and we "carry" the 1 to the next sum.

   `List: 2 -> ?`

2. We then add 1 and 9, as well as the "carry," getting a result of 11. 1 becomes the second element of our linked list, and we carry the 1 to the next sum.

   `List: 2 -> 1 -> ?`

3. Finally, we add 6, 2 and our "carry," to get 9. This becomes the final element of our linked list.

   `List: 2 -> 1 -> 9.`

The code below implements this algorithm.

```
LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2, int carry) {
  if (l1 == null && l2 == null && carry == 0) {
    return null;
  }

  LinkedListNode result = new LinkedListNode();
  int value = carry;
  if (l1 != null) {
    value += l1.data;
  }
  if (l2 != null) {
      value += l2.data;
    }

    result.data = value % 10; /* Second digit of number */

    /* Recurse */
    if (l1 != null || l2 != null) {
      LinkedListNode more = addLists(l1 == null ? null : l1.next,
                                     l2 == null ? null : l2.next,
                                     value >= 10 ? 1 : 0);
      result.setNext(more);
    }
    return result;
  }
```

In implementing this code, we must be careful to handle the condition when one linked list is shorter than another. We don't want to get a null pointer exception.

Follow Up

Part B is conceptually the same (recurse, carry the excess), but has some additional complications when it comes to implementation:

1. One list may be shorter than the other, and we cannot handle this "on the fly." For example, suppose we were adding (1 -> 2 -> 3 -> 4) and (5 -> 6 -> 7). We need to know that the 5 should be "matched" with the 2, not the 1. We can accomplish this by comparing the lengths of the lists in the beginning and padding the shorter list with zeros.
2. In the first part, successive results were added to the tail (i.e., passed forward). This meant that the recursive call would be *passed* the carry, and would return the result (which is then appended to the tail). In this case, however, results are added to the head (i.e., passed backward). The recursive call must return the result, as before, as well as the carry. This is not terribly challenging to implement, but it is more cumbersome. We can solve this issue by creating a wrapper class called Partial Sum.

The code below implements this algorithm.

```
class PartialSum {
  public LinkedListNode sum = null;
  public int carry = 0;
}

LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
  int len1 = length(l1);
  int len2 = length(l2);

  /* Pad the shorter list with zeros - see note (1) */
  if (len1 < len2) {
    l1 = padList(l1, len2 - len1);
  } else {
    l2 = padList(l2, len1 - len2);
  }

  /* Add lists */
  PartialSum sum = addListsHelper(l1, l2);

  /* If there was a carry value left over, insert this at the front of the list.
   * Otherwise, just return the linked list. */
  if (sum.carry == 0) {
      return sum.sum;
    } else {
      LinkedListNode result = insertBefore(sum.sum, sum.carry);
      return result;
    }
}

PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
    if (l1 == null && l2 == null) {
      PartialSum sum = new PartialSum();
      return sum;
    }
    /* Add smaller digits recursively */
    PartialSum sum = addListsHelper(l1.next, l2.next);

    /* Add carry to current data */
    int val = sum.carry + l1.data + l2.data;

    /* Insert sum of current digits */
    LinkedListNode full_result = insertBefore(sum.sum, val % 10);

    /* Return sum so far, and the carry value */
    sum.sum = full_result;
    sum.carry = val / 10;
    return sum;
}

/* Pad the list with zeros */
LinkedListNode padList(LinkedListNode l, int padding) {
    LinkedListNode head = l;
    for (int i = 0; i < padding; i++) {
      head = insertBefore(head, 0);
    }
    return head;
}

/* Helper function to insert node in the front of a linked list */
LinkedListNode insertBefore(LinkedListNode list, int data) {
    LinkedListNode node = new LinkedListNode(data);
    if (list != null) {
      node.next = list;
    }
    return node;
}
```

Note how we have pulled `insertBefore()`, `padList()`, and `length()` (not listed) into their own methods. This makes the code cleaner and easier to read—a wise thing to do in your interviews!

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.

```
boolean isPalindrome(LinkedListNode head) {
  LinkedListNode reversed = reverseAndClone(head);
  return isEqual(head, reversed);
}

LinkedListNode reverseAndClone(LinkedListNode node) {
  LinkedListNode head = null;
  while (node != null) {
    LinkedListNode n = new LinkedListNode(node.data); // Clone
    n.next = head;
    head = n;
    node = node.next;
  }
  return head;
}

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