**2**

**Linked Lists**

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node.

The following diagram depicts a doubly linked list:



Unlike an array, a linked list does not provide constant time access to a particular "index" within the list. This means that if you'd like to find the Kth element in the list, you will need to iterate through K elements.

The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time. For specific applications, this can be useful.

**Creating a Linked List**

The code below implements a very basic singly linked list.

public class Node

{

Node next = null;

int data;

public Node(int d)

{

data = d;

}

void appendToTail(int d)

{

Node end = new Node(d);

Node n = this;

while (n.next != null)

{

n = n.next;

}

n.next = end;

}

}

In this implementation, we don't have a Linked List data structure. We access the linked list through a reference to the head Node of the linked list. When you implement the linked list this way, you need to be a bit careful. What if multiple objects need a reference to the linked list, and then the head of the linked list changes? Some objects might still be pointing to the old head.

We could, if we chose, implement a Linked List class that wraps the Node class. This would essentially just have a single member variable: the head Node. This would largely resolve the earlier issue.

Remember that when you're discussing a linked list in an interview, you must understand whether it is a singly linked list or a doubly linked list.

**Deleting a Node from a Singly Linked List**

Deleting a node from a linked list is fairly straightforward. Given a node n, we find the previous node prev and set prev.next equal to n.next. If the list is doubly linked, we must also update n.next to set n.next.prev equal to n.prev. The important things to remember are (1) to check for the null pointer and (2) to update the head or tail pointer as necessary.

Additionally, if you implement this code in C, C++ or another language that requires the developer to do memory management, you should consider if the removed node should be deallocated.

Node deleteNode(Node head, int d)

{

Node n = head;

if(n.data == d)

{

return head.next; // move head

}

while (n.next != null)

{

if(n.next.data == d)

{

n.next = n.next.next;

return head;// head didnt change

}

n = n.next;

}

return head;

}

**The “Runner” Technique**

The "runner" (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.

For example, suppose you had a linked list a1 ->a2 -> ••• ->an ->b1 ->b2 -> ••• ->bn and you wanted to rearrange it into a1 ->b1 ->a2 ->b2 -> ••• ->an ->bn. You do not know the length of the linked list (but you do know that the length is an even number).

You could have one pointer pl (the fast pointer) move every two elements for every one move that p2 makes. When pl hits the end of the linked list, p2 will be at the midpoint. Then, move pl back to the front and begin "weaving" the elements. On each iteration, p2 selects an element and inserts it after pl.

**Recursive Problems**

A number of linked list problems rely on recursion. If you're having trouble solving a linked list problem, you should explore if a recursive approach will work. We won't go into depth on recursion here, since a later chapter is devoted to it.

However, you should remember that recursive algorithms take at least O(n) space, where n is the depth of the recursive call. All recursive algorithms can be implemented iteratively, although they may be much more complex.

**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?

**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 duplicate element, we remove the element and continue iterating. We can do this all in one pass since we are using a linked list

private void deleteDups(Node n)

{

HashSet<int> set = new HashSet<int>();

Node 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.

private void deleteDupsNoBuff(Node head)

{

Node current = head;

while (current != null)

{

/\* Remove all future nodes that have the same value \*/

Node 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.

**2.2 Return Kth to Last:** Implement an algorithm to find the kth to last element of a singly linked list.

**SOLUTION**

We will approach this problem both recursively and non-recursively. Remember that recursive solutions are often cleaner but less optimal. For example, in this problem, the recursive implementation is about half the length of the iterative solution but also takes O(n) space, where n is the number of elements in the linked list.

Note that for this solution, we have defined k such that passing ink = 1 would return the last element, k = 2 would return to the second to last element, and so on. It is equally acceptable to define k such that k = 0 would return the last element.

**Solution #1: If linked list size is known**

If the size of the linked list is known, then the kth to last element is the (length - k)th element. We can just iterate through the linked list to find this element. Because this solution is so trivial, we can almost be sure that this is not what the interviewer intended.

**Solution #2: Recursive**

This algorithm recurses through the linked list. When it hits the end, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we know we have reached the kth to last element of the linked list.

Implementing this is short and sweet-provided we have a way of "passing back" an integer value through the stack. Unfortunately, we can't pass back a node and a counter using normal return statements. So how do we handle this?

**Approach A: Don't Return the Element.**

One way to do this is to change the problem to simply printing the kth to last element. Then, we can pass back the value of the counter simply through return values.

private int printKthToLast(Node head, int k)

{

if (head == null)

{

return 0;

}

int index = printKthToLast(head.next, k) + 1;

if (index == k)

{

Debug.WriteLine(k + "th to last node is " + head.data);

}

return index;

}

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.

private class Index

{

public int value;

}

private Node kthTolast(Node head, int k)

{

Index idx = new Index();

return kthToLast(head, k, idx);

}

private Node kthToLast(Node head, int k, Index idx)

{

if (head == null)

{

return null;

}

Node 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.

private Node nthTolast(Node head, int k)

{

Node p1 = head;

Node p2 = head;

/\* Move pl 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 pl hits the end, p2 will be at the right element. \*/

while (p1 != null)

{

p1 = p1.next;

p2 = p2.next;

}

return p2;

}

**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

**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.

private bool deleteNode(Node n)

{

if (n == null || n.next == null)

{

return false; // Failure

}

Node 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

**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\*/

private Node partitionNode(Node node, int x)

{

Node beforeStart = null;

Node beforeEnd = null;

Node afterStart = null;

Node afterEnd = null;

/\* Partition list\*/

while (node != null)

{

Node 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.

private Node partitionN(Node node, int x)

{

Node head = node;

Node tail = node;

while (node != null)

{

Node 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.

**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.

private 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);

}

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 recur sive 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

private 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;

}

}

private PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2)

{

if (l1 == null && l2 == null)

{

PartialSum sum1 = new PartialSum();

return sum1;

}

/\* 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\*/

private 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\*/

private LinkedListNode insertBefore(LinkedListNode list, int data)

{

LinkedListNode node = new LinkedListNode(data);

if (list != null)

{

node.next = list;

}

return node;

}

public int length(LinkedListNode l1)

{

LinkedListNode temp = l1;

int cnt = 0;

while (temp != null)

{

cnt++;

temp = temp.next;

}

return cnt;

}

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

**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.

private bool isPalindrome(LinkedListNode head)

{

LinkedListNode reversed = reverseAndClone(head);

return isEqual(head, reversed);

}

private LinkedListNode reverseAndClone(LinkedListNode node)

{

LinkedListNode head = null;

while (node != null)

{

LinkedListNode n = new LinkedListNode(node.data);

node = node.next;

}

return head;

}

private bool 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.

bool isPalindrome2(LinkedListNode head)

{

LinkedListNode fast = head;

LinkedListNode slow = head;

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

/\* 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();

/\* 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 for 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.

private 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. vl = is Palindrome: 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 = is Palindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5. returns v3
6. returns v2
7. returns vl
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 1 f 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. lfthe values match, it passes a reference to node lb (returned\_node. next) up to line 2.
* Line 2 compares its head (node 1 f) to returned\_node (node lb). 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 bool result;

}

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

1. isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
2. is Palindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
3. is Palindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
4. is Palindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0, len = 3
5. isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
6. returns node 3b, true
7. returns node 2b, true
8. returns node lb, true
9. returns node 0b, true
10. returns null, true

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

bool isPalindrome2(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.