2. Linked Lists

A linked list is a data structure that represents a sequence of nodes.

Unlike an array, a linked list does NOT provide constant time access to a particular "index" within a list. You have to iterate through K elements to get to the Kth element in the list.

The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time.

Implementing a Linked List

The code below implements a very basic singly linked list:

```
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 LinkedList data structure. We access the LinkedList through a reference to the head Node of the linked list. We need to be careful with this implementation. 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 implement a LinkedList class that wraps the Node class. This would just have a single member variable: the head Node. This will resolve the issue above.

Delete a Node from a Singly Linked List

Given a node `n`, we find the previous node `prev` and set `prev.next = n.next`.

If the list is doubly linked, we must also update `n.next` to set `n.next.prev = n.prev`.

Remember to:

1. Check for the null pointer
2. Update the head or tail pointer as necessary.

```
Node deleteNode(Node head, int d) {
  Node n = head;

  if (n.data == d) {
    return head.next; // Moved head
  }

  while (n.next != null) {
    if (n.next.data == d) {
      n.next = n.next.next;
      return head;
    }
    n = n.next;
  }
  return head;
}
```

The "Runner" Technique

The "runner" (or second pointer) technique is used in many linked list problems. You iterate through the linked list with two pointers simultaneously, with one ahead of the other.

Recursive Problems

A number of linked list problems rely on recursion. If you're having trouble solving a linked list problem, try if a recursive approach will work.

Remember that recursive algorithms take at least O(n) space, where n is the depth of the recursive call.

HashTable vs HashMap vs HashSet

HashTable

- Allows duplicate keys.
- Does not allow null for either key or value.
- HashTable does NOT maintain insertion order. The order is defined by the hash function.
- Synchronized, slow, only one thread can access at a time.
- Is thread-safe

- Uses Enumerator to iterate through elements.

HashMap

- Allows duplicate keys.
- Allows one null key and as many null values as you like.
- HashMap does NOT maintain insertion order. The order is defined by the hash function.
- NOT synchronized. Faster than HashTable.
- Not thread-safe, but can use Collections.synchronizedMap(new HashMap<K, V>())

HashSet

- Does not allow duplicate keys.
- Allows a null key.
- Can use contain method to check if object is already in the set.

Questions

Question 1: Write code to remove duplicates from an unsorted linked list.

We first have to track duplicates. A simple hash table will work.

We 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 in one passing.

```
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; // skip pointing to current node
    } else {
      set.add(n.data);
      previous = n;
    }
    n = n.next;
  }
}
```

Time complexity: O(N), where N is the number of elements in the linked list.

Follow Up: No Buffer Allowed

If we cannot have an additional data structure, we can iterate with two pointers.

```
void deleteDups(LinkedListNode head){
  LinkedListNode current = head;
  while (current != null){
    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;
  }
}
```

Time Complexity: O(N^2), space complexity: O(1). Trade-off!

Question 2: Return Kth to Last: Implement an algorithm to find the kth to last element of a singly linked list.

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.

For this implementation, we have defined k such that in k = 1 would return the last element, k = 2 would return the second to last element, and so on.

Solution #1: If the linked list size is known
The kth to last element is the (length - k)th element. We can just iterate through the linked list to find this element. (Trivial answer)

Solution #2: Recursive This algorithm recurses through the linked list. When it hits the last element, the method passes back a counter set to 0. Each parent call adds 1 to this counter. When the counter equals k, we will reach the kth to last element of the list.

Unfortunately, we can't pass back a node and a counter using normal return statements.

Approach A: Don't Return the Element
Simply print the kth to last element. Then we can pass back the value of the counter simply through return values.

```
int printKthToLast(LinkedListNode head, int k) {
  if (head == null) {
    return 0;
  }
```

```
  int index = printKthToLast(head.next, k) + 1;
  if (index == k) {
    System.out.println(k + "th to last node is " + head.data);
  }
  return index;
}
```

Approach B: Create a Wrapper Class As describe earlier, we can't simultaneously return a counter and an index. If we wrap the counter value with a simple class (or an 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, index);
}

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 recurisve calls.

Solution #3: Iterative
More optimal, but less straightforward. We can use 2 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, we move them at the same pace. p1 will hit the end of the linked list after (length - k) steps. At this point, p2 will be (length - k) nodes into the list, or k nodes from the end. Then, just print the value of p2.

```
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;
    p1 = p1.next;
  }

  // Move them at the same pace
  while (p1 != null){
    p1 = p1.next;
    p2 = p2.next;
  }
  return p2;
}
```

Time complexity: `O(n)`, where n is the length of the linked list. Space complexity: `O(1)` (in-place).

Question 3: Delete Middle Node. Implement an algorithm to delete a node in the middle (i.e. any node but the first and last node) of a singly linked list, given only access to that node.

Example:
Input: Node C from the linked list a -> b -> c -> d -> e -> f Output: Nothing is returned, but the new linked list looks like a -> b -> d -> e -> f

Solution:
Since you're only given access to that node, simple copy the data from the next node over to the current node, then delete the next node.

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

Question 5: Sum Lists. You have two numbers represented by a linked list, where each node contains a digit. 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: What if the digits are stored in forward order?

Solution: We can mimic the addition process recursively by adding node by node, carrying over excess data to the next node.

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

We do the following:

1. Add 7 and 5 first, getting 12. 2 Becomes the first node in our linked list and we carry the 1 to the next sum.
   List: 2 -> ?
2. Add 1 and 9 and the carry, getting. 1 Becomes the second element and we carry over 1 to the next sum.
   List: 2 -> 1 -> ?
3. Add 2 and 6 and the carry, getting 9.
   List: 2 -> 1 -> 9

This is the algorithm:

```
LinkedListNode addLists(LinkedListNode node1, LinkedListNode node2, int carry) {
  if (node1 == null && node2 == null && carry = 0) {
    return null;
  }

  LinkedListNode result = new LinkedListNode();
  int value = carry;
  if (node1 != null) {
    value += node1.data;
  }
  if (node2 != null){
    value += node2.data;
  }

  result.data = value % 10;

  // Recursion
  if (node1 != null || node2 != null){
    LinkedListNode nextNode = addLists(node1 == null ? null : node1.next,
                                       node2 == null ? null : node2.next,
                                       value >= 10 ? 1 : 0);
    result.setNext(nextNode);
  }
  return result;
}
```

Remember to handle the condition when one linked list is shorter than another so that we don't get a null pointer exception.

FOLLOW UP Solution
Part B is conceptually the same (recurse, carry the excess), but has additional complications to it:

1. One list may be shorter than the other and we cannot handle this on the fly. We can accomplish this by comparing the lengths of the lists and padding the shorter list with zeros.
2. In the first part, successive results were added to the tail. In this case, they are added to the head. The recursive call must return the result, as well as the carry. We can solve this issue by creating a wrapper class.

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

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

  if (len1 < len2) {
    l1 = padList(l1, len2 - len1);
  } else {
    l2 = padList(l2, len1 - len2);
  }

  PartialSum sum = addListsHelper(l1, l2);

  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
  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;
}

LinkedListNode padList(LinkedListNode node, int padding) {
  LinkedListNode head = node;
  for (int i = 0; i < padding; i++){
    head = insertBefore(head, 0);
  }
  return head;
}

LinkedListNode insertBefore(LinkedListNode list, int data){
  LinkedListNode node = new LinkedListNode(data);
  if (list != null){
    node.next = list;
  }
  return node;
}
```

Question 6: Palindrome. Check if a linked list is a palindrome.

Solution 1: Reverse and Compare.
We only actually need to compare the first half of the 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;
  }
}

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;
}
```

Solution 2:
If we know the size of the linked list, we can iterate through the first half of the elements and push each element onto a stack. (Be careful 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 technique. At each step in the loop, push the data from the slow runner onto a stack. When the fast runner reached the end of the loop, the slow runner will be at the middle of the linked list.

Either way, by this point the stack will have all the elements from the front of the linked list to the middle part, but in reverse order.

So all we have to do is to continue our iteration through the linked list (from the middle point) and compare the node with the top of the stack.

Question 8: Loop Detection. Given a circular linked list, check if the linked list is corrupt.

EXAMPLE
Input: A -> B -> C -> D -> E -> C (the same C as earlier)
Output: C

SOLUTION
This is a modification of a classic interview problem: detect if a linked list has a loop.

Part 1: Detect if linked list has a loop
An easy way to detect if a linked list has a loop is through the Fast Runner technique. If there is a loop, they will eventually meet.

Part 2: when do they collide?
Let's assume that the linked list has a non-looped part of size k.

If we apply our algorithm from part 1, when will FastRunner and SlowRunner collide?

We know that for every `p` steps that SlowRunner takes, FastRunner has taken `2p` steps. Therefore, when SlowRunner enters the looped portion after `k` steps, FastRunner has taken `2k` steps total and `(2k - k)` steps, or `k` steps, into the looped portion.

Since `k` might be much larger than the loop length, we can write this as `mod(k, LOOP_SIZE)` steps, which we will denote as `K`.

So now we know:

1. SlowRunner is 0 steps into the loop
2. FastRunner is K steps into the loop
3. SlowRunner is K steps behind the FastRunner

4. FastRunner is LOOP_SIZE - K steps behind SlowRunner
5. FastRunner catches up to SlowRunner at a rate of 1 step per unit of time

FastRunner will meet SlowRunner after LOOP_SIZE - K steps. At this point, they will be K steps before the head of the loop. Let's call this point CollisionSpot.

Part 3: How do you find the start of the loop?
We now know that CollisionSpot is K nodes before the start of the loop because K = mod(k, LOOP_SIZE). In other words, k = K + M * LOOP_SIZE, for any integer M.

Hence, both CollisionSpot and LinkedListHead are k nodes from the start of the loop. Now if we keep one pointer at CollisionSpot and move the other one to LinkedListHead, they will each be k nodes from LoopStart. Moving the 2 pointers at the same speed will cause them to collide again, this time after k steps, at which point they will be both at LoopStart. Return this node.