LINKED LIST EXERCISES

In this mini-workbook, we'll get you up and running with linked lists. These are used as a building block to the more complicated data structures such as trees, graphs, etc., so it is important to deeply understand their inner workings.

In the guide, we will quickly highlight the key concepts that you need to know for your interview. If you're unfamiliar with any of these, I highly recommend taking some time to review them.

Then, I've provided you with a series of exercises to help you get up and running quickly. I recommend setting aside 2-3 days to work through these exercises. Simply start from the top and work through them as much as you have time.

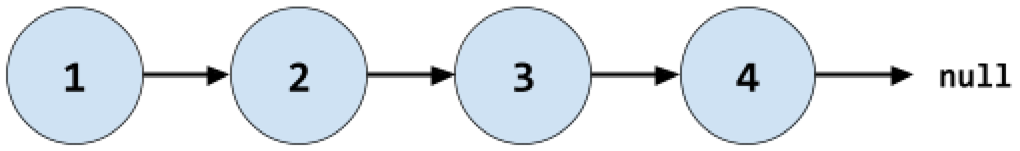
Make sure to track the time you spend here in your Interview Prep Tracker. You can categorize this as "Study" time.

For these exercises, we have complete code samples and templates available here.

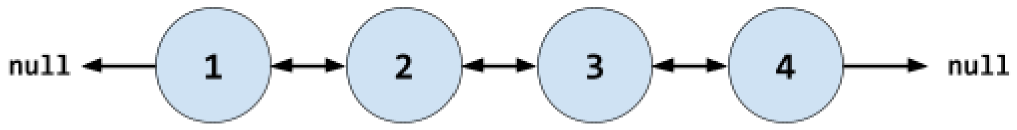
TABLE OF CONTENTS

KEY LINKED LIST TERMINOLOGY

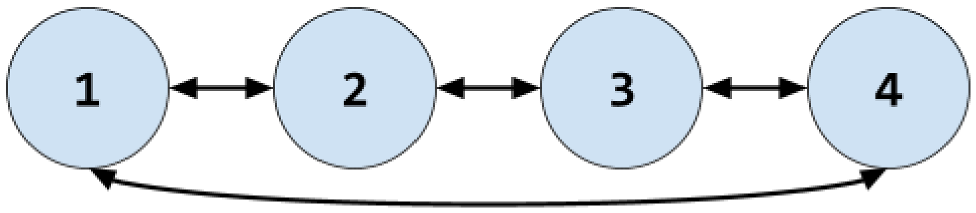
- **Singly Linked List**
  A linked list with nodes only pointing in a singular direction. Nodes have no reference to the node that came before.



- **Doubly Linked List**
  A linked list where nodes point both to the next node and the previous node. This is commonly used in standard language implementations of linked lists. While it adds some complexity to the operations, it can make for much easier access of the data.



- **Circular Linked List**
  A doubly linked list where the tail points to the head and the head points to the tail.



KEY LINKED LIST PATTERNS

- **Accessing and Modifying Values**
  This is the core pattern for every data structure. Without being able to get and set the data in our data structure, it is impossible for us to utilize it in any meaningful way. For our linked lists, this includes adding and removing values from the list as well as iterating over the values in the list.

  It is important to understand how to access and modify both singly- and doubly-linked lists.

- **Updating Pointers in the List**
  This is a subset of accessing and modifying values. However it is worth considering separately. Here, we manipulate the pointers to change the list in various ways. For example, we may reverse our list by switching the directions of the pointers or we may interleave two lists together.

- **Using Multiple Pointers**
  Similar to our arrays and strings, we can use multiple pointers into lists to accomplish a variety of things. These are good for comparing different nodes or different lists. We can also use “runner” pointers to find nodes positioned relative to the end of the list (ie. the nth-to-last node).

- **Using Dummy Nodes**
  Sometimes, we may find it useful to add a “dummy” node to the beginning of our list. This is a node that doesn’t contain any data but just points to the first node of our list. This is often useful when you have a loop referencing the previous node and you want to avoid an NPE on the first iteration.

- **Floyd’s Cycle Detection Algorithm**
  Floyd’s Cycle Detection algorithm is an algorithm to determine whether a Linked LIst loops back on itself. While this is very specific, it has become popular enough in interviews that it is worth knowing.

  This algorithm simply works as follows:
  1. Initialize 2 pointers, one at the first node of the list and one at the second
  2. Each loop, move one pointer forward by one node and the other forward by two nodes
  3. If the fast pointer reaches the end of the list, there is no cycle
  4. Otherwise, check each loop whether the fast and slow pointers point to the same node. If there is a cycle, then there will always be a point at which they are equal

  We will implement this later in the exercises.

LINKED LIST EXERCISES

Exercise Set #1 - Accessing and Modifying Values

1. Implement a singly-linked list class with the following methods:
    a. Constructor
    b. `insert(int n)` - Inserts the value n at the front of the list
    c. `delete(int n)` - Removes the first occurrence of a node with the value `n` from the list. Return true if the list contains the given value
    d. `size()` - Return the length of the linked list
    e. `toString()` - Convert the list into a string. This should be of the form “1 -> 2 -> 3 -> null”

2. Implement a doubly-linked list class with the following methods:
    a. Constructor
    b. `insert(int n)` - Inserts the value n at the front of the list
    c. `delete(int n)` - Removes the first occurrence of a node with the value `n` from the list. Return true if the list contains the given value
    d. `size()` - Return the length of the linked list
    e. `toString()` - Convert the list into a string. This should be of the form “null <- 1 <-> 2 <-> 3 -> null”

3. For each of your functions, compute the time and space complexity

Exercise Set #2 - Updating Pointers

For these and the remaining exercises in this workbook, treat `list` as a pointer to the first node in the list. Do not use your list class from eariler, although you may reuse the `ListNode` classes.

1. Write a function that swaps the nodes at two indices in a doubly-linked list.

    You should accomplish this by modifying the pointers in the list and not create a new list.

    eg.
    ```
    list = null <- 1 <-> 2 <-> 3 <-> 4 <-> 5 -> null
    swap(list, 1, 3)
    list = null <- 1 <-> 4 <-> 3 <-> 2 <-> 5 -> null
    ```

    What is the time and space complexity of your solution?

2. Write a function that removes the odd-indexed values from a singly-linked list.

   You should accomplish this by modifying the pointers in the list and not create a new list.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5 -> null
   removeOdd(list)
   list = 1 -> 3 -> 5 -> null
   ```

   What is the time and space complexity of your solution?

3. Write a function that de-interleaves the even and odd indices in a singly-linked list. Your resulting list should have all the even indices first followed by all the odd indices.

   You should accomplish this by modifying the pointers in the list and not create a new list.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> null
   deinterleave(list) = 1 -> 3 -> 5 -> 2<-> 4 -> 6 -> null
   ```

   What is the time and space complexity of your solution?

4. Write a function that reverses a singly-linked list.

   You should accomplish this by modifying the pointers in the list and not create a new list.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5 -> null
   reverse(list) = 5 -> 4 -> 3 -> 2 -> 1 -> null
   ```

   What is the time and space complexity of your solution?

Exercise Set #3 - Using Multiple Pointers

1. Write a function that compares 2 singly-linked lists and returns true if the two lists are identical.

   eg.
   ```
   list1 = 1 -> 2 -> 3 -> null
   list2 = 2 -> 2 -> 3 -> null
   areEqual(list1, list2) = false
   ```

   What is the time and space complexity of your solution?

2. Write a function that returns the nth-to-last value in a singly-linked list.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5 -> null
   n = 2
   nthToLast(list, n) = 4
   ```

   What is the time and space complexity of your solution?

3. Write a function that returns the value at the midpoint of a singly-linked list. You can assume the length of the list is odd.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5 -> null
   result = 3
   ```

   What is the time and space complexity of your solution?

Exercise Set #4 - Using Dummy Nodes

1. Remove all occurrences of n from a singly-linked list.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 1 -> 5 -> null
   n = 1
   result = 2 -> 3 -> 5 -> null
   ```

   What is the time and space complexity of your solution?

Exercise Set #5 - Floyd's Cycle Detection Algorithm

1. Given a singly-linked list, determine if the list contains a cycle. DO NOT use Floyd's algorithm. FInd some other method for identifying a cycle.

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5
               ^              |
               ---------------

   containsCycle(list) = true
   ```

   What is the time and space complexity of your solution?

2. Now implmement the same thing again using Floyd's Algorithm. As a reminder, here is how it works:
   a. Initialize 2 pointers, one at the first node of the list and one at the second
   b. Each loop, move one pointer forward by one node and the other forward by two nodes
   c. If the fast pointer reaches the end of the list, there is no cycle
   d. Otherwise, check each loop whether the fast and slow pointers point to the same node. If there is a cycle, then there will always be a point at which they are equal

   eg.
   ```
   list = 1 -> 2 -> 3 -> 4 -> 5
               ^              |
               ---------------

   containsCycle(list) = true
   ```

   What is the time and space complexity of your solution?

Exercise Set #6 - Additional Practice

You do not need to complete these problems on your initial pass of this workbook. These are good exercises to revisit later if you would like to go deeper on Linked Lists.

- Remove duplicates from a sorted list (Full Problem Definition)
- Is list palindromic? (Full Problem Definition)
- Pairs with Sum (Full Problem Definition)
- Merge two sorted lists (Full Problem Definition)
- Copy list with random pointer (Full Problem Definition)
- Reorder list (Full Problem Definition)