Linked Lists

Lab Setup

1. Create a new Java project called JavaLinkedList
2. Download the provided Java files and place them in the src directory
3. Add a main class

Linked List

A linked list is a data structure used to store elements in linear order. It is like arrays (including ArrayList) but differs in how the list is stored in memory. Arrays store elements contiguously in memory. Linked lists do not store elements contiguously, instead, each element points to the next element in the list. In memory, each element is most likely not stored contiguously.

Linked lists can be thought of as being made up of nodes. A node holds some data, e.g., a number or string, and it also holds a reference to the element that comes after it in the list. Recall that in Java, all Object variables hold the reference to the object instead of the object itself. If a Node only points to the next Node in the list, then the list is singly linked. If a Node points to both the next Node and the previous Node in the list, then the list is doubly linked.

Because a linked list is built using references, we need at least a reference to the first node in the list, also called the head, to have access to the list.

Task 1

1. Declare the class SinglyLinkedList, which is-a LinkedList.
2. Declare the inner class called Node inside of SinglyLinkedList.
3. Define the constructor for SinglyLinkedList which creates an empty list
4. Use Eclipse to create all the stub methods that need to be overridden

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab.

Linked List operations

We will look at how to do basic operations with the node-based infrastructure.

Traversing a list

Unlike an array, which allows for constant time access i.e., given an index you can directly access an element, a linked list must be traversed to access

```
// Iterating the entire list

Node itr = head;
while (itr != null) {
   itr = itr.next;
}

// Iterating to the tail node
Node itr = head;
while (itr.next != null) {
   itr = itr.next;
}

// Iterating until some index
Node itr = head;
for (int i = 0; i < idx; i++) {
   itr = itr.next;
}
```

a given index. We declare a variable that will be used as an iterator, itr and refer it to the head of the list. If we want to access the element after head then we refer itr to head's next reference, which we can access through itr, itr = itr.next. Now, itr is pointing at the second element in the list. We can control how far we iterate through a list using a loop.

We can avoid having to iterate the entire list to reach the tail by keeping a reference to the tail and updating the reference whenever it changes.

Adding an element to the end of the list

Adding an element to the end of the list involves getting there first. If a list is empty, i.e. there is no head node, then we can simply create the first node. If a list is not empty, we need to reach the end of the list. If we do not already have a reference to the tail node, then we must start at the head and iterate until we reach the tail. We know we are at the tail, if the node's next reference is null.

Getting an element from the list

Getting an element from the list given an index is simple. Because the linked list is a linear sequence, we can assign an index to each of the elements in the list. Then we can iterate through the nodes while also counting until we reach the index we want. We can treat index 0 and the last index as special cases to save having to iterate. The last index special case would require a reference to the tail node.

Inserting an element at a position in the list

Inserting an element at a position in the list is a little more complicated. Recall that to insert an element in an array, we need to shift the elements to the right of that position to the right to make space for the new element. This is not required for a linked list, we do not need to do any shifting of the nodes, and we only need to update the links.

For a singly linked list, each node only has a reference to the next node. If we iterate the nodes until we reach the position to which we want to insert the new element at, we would not be able to do the insertion because we need to update the previous index node to reference the new node. So, we need to iterate to the index – 1.

Removing an element from the list

Removing an element from the list follows the same process as inserting an element. We need to iterate to index – 1 so that we can correctly update the links. Removing the node involves removing references to that node, which can be done by pointing the node at index-1 to the node at index+1.

Setting an element in the list

If we want to change the value of an element at a given index, we do not need to create a new Node. Instead, we can update the value the Node holds to be the new value. We still need to traverse the list to reach whichever node we want to update.

Task 2

1. Implement the described operations for a singly-linked list: add, get, insert, remove, and set.
2. Show that the implementation of the linked list is correct using either a JUnit test class or a driver class

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 2. You will submit this copy at the end of this lab.

Lab wrap-up

In this lab we discussed a linear sequential data structure called a linked list. A linked list is made up of nodes which may be places sparsely in memory. This differs from other list-like data structures like the array which stores elements contiguously in memory. Each node in a linked list has a reference to the next node in the sequence. Such linked lists are called singly-linked, and if the nodes also have a reference to the previous node then the linked list is doubly-linked. Because the list is not stored contiguously in memory, we must traverse list using the links if we want to access a certain index.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder "LabLinkedList_firstname_lastname", where first name and last name are
- Submit the zipped file to the respective Canvas link.

Rubrics	
Task 1 (points allotted) - Correct declaration of SinglyLinkedList (2) - Correct declaration of inner class Node (2) - Constructor creates empty list (2) - All stub methods (4)	Task 2 (points allotted) - Implementations of add, get, insert, remove, and set (5) - Driver/JUnit test case demonstrating implementation is correct (5)