Announcements

- Program 4 has been assigned
  - Due Friday 11/10/19 , 11:59PM
  - Extra Credit due: 11/3/19, 11:59 PM
- Midterm 2
  - Wednesday 11/6 (next week)
  - Still working on Practice Problems

Quiz Time ;)

Linked Lists (cont.)

- The **prepend** operation for a doubly-linked list inserts the new node before the list's head node (java tends to call this operation: addFirst)
- Similar to the append operation, there are two cases to be concerned with: behavior when the list is empty versus behavior when the list is non-empty

Linked Lists (cont.)

- Prepend to an empty list: if the list's head pointer is null (empty), the list's head and tail pointers reference the new node.
- Prepend to a non-empty list: if the list's head pointer is not null (i.e there is at least one element in the list), then the new nodes "next" pointer must reference the head node, then point the head node's previous pointer to the new node, and then the head node must reference the new node.

Let's look at an example...

Linked Lists (cont.)

- The **insertAt** operation for a singly-linked list inserts the new node at a provided index. (java tends to call this operation add(int index, E element))
- The element at the desired index and subsequent indices get “shifted” over
- Note: this is different from zybooks which talks about “InsertAfter” which inserts after a specified node, the two operation are very similar.
- There are now four scenarios we have to worry about when doing this operation: empty list, insert at the list’s head node, insert at lists tail node, insert in the middle of the list

Linked Lists (cont.)

- Insert as the list's first node: if the list's head pointer is null, and the index to insert at is zero, the list's head and tail should then point to (i.e reference) the new node
- Insert at the list's head node: This is handled exactly the same as prepend...
- Insert at the tail node: If the list's head pointer is not null (i.e the list is not empty) AND the index is the same as the Tail pointers current index, then we must set the Tail pointers previous nodes next to reference the new node, the new nodes "previous" pointer must reference the Tail pointer's previous node, then the Tail's "previous" pointer must reference the new node, and the new node's next pointer to reference the tail.
    - ~~To do this assume we have access to the previous node, let us call it prevNode.~~

Linked Lists (cont.)

- Insert in the middle of the list: if the list's head pointer is not null:
    - Traverse the list using the linked **list traversal algorithm**, this time a secondary stopping condition will be the desired index at which to stop, this will give us access to a reference called **curNode** that references the node at the specified index location.
    - Set **curNode**'s previous node's "next" pointer to reference the **new node**
    - Set the **new node**'s "previous" pointer to reference **curNode**'s previous node.
    - Set the **new node**'s "next" pointer to reference **curNode**
    - Set **curNode's** "previous" pointer to reference the **new node**

Let's look at an example...

Linked Lists (cont.)

Notice that when we insert an element into the middle of a linked list we must first traverse the linked list in order to get access to that location.

This is a downside to a linked list data structure when compared to an arraylist.

This is because an arraylist holds an array which can be easily indexed into, meaning there is no traversal needed to get access to a specific location.

Linked Lists (cont.)

RemoveAt

I leave it as an exorcise to you, to figure out how having a “previous” pointer might affect this removeAt operation.

Refer to the singly-linked lists implementation for some guidance.

Stacks and Queues

Stacks and queues are both data structures that are often taught together due to their relationship as opposites.

To understand how they are opposites, you must first understand LIFO vs FIFO

- Last In First Out (LIFO) - the last element into the data structure must be the first one to be removed (also known as FILO - First In Last Out)
- First In First Out (FIFO) - the first element into the data structure must be the first one to be removed

Stack and Queue (cont.)

Stack analogy - Stacking concrete blocks

- Cannot pull the blocks from the middle or bottom of the stack
- If want to remove a block, must pull off the block at the top, one at a time...
- In other words, the block at the top was the last in, and is therefore the first out

Queue analogy - Waiting in a line

- The person at the front of the line is the person who has been in the line the longest (i.e. the first to enter the line, First In), and accordingly is the first to leave the line (i.e. First Out).

Stack and Queue (cont.)

A **stack** is a data structure in which items are only inserted on or removed from the top of a stack.

- The **push** operation inserts an item on the top of the stack.
- The **pop** operation removes and returns the item at the top of the stack.
- The **peek** operation returns the item at the top of the stack without removing actually it.
- A stack can be implemented using a linked list or an array (or an arraylist)

Let's look at an example of a stack implementation on the whiteboard….

Stack and Queue (cont.)

A **queue** is a data structure in which items are inserted at the end of the queue and removed from the front of the queue.

- The **push** operation inserts an item at the end of the queue (Sometimes this is referred to as “enqueue”)
- The **pop** operation removes and returns the item at the front of the queue. (Sometimes this is referred to “dequeue”)
- The **peep** operation returns but does not remove the item at the front of the queue.
- A queue can be implemented using a linked list, or an array (or an arraylist)

Let’s look at an example of a stack implementation on the whiteboard….

Stacks and Queues (cont.)

Let’s look at java’s documentation for Stacks and Queues, you will notice some differences in the method names and how they might be handled.