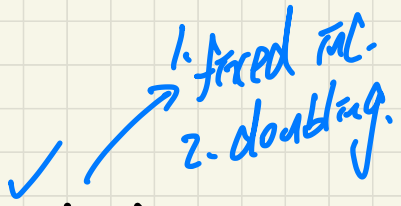
EECS2011 Fundamentals of Data Structures (Winter 2022)

Q&A - Lectures 5 & 6

Wednesday, March 2

Announcements

- Lecture W7 released
  - + Circular Arrays
  - + Amortized Analysis of Dynamic Arrays
  - + Binary Search
  - + RT of Recursive Algorithms
- A1 grading still going on...

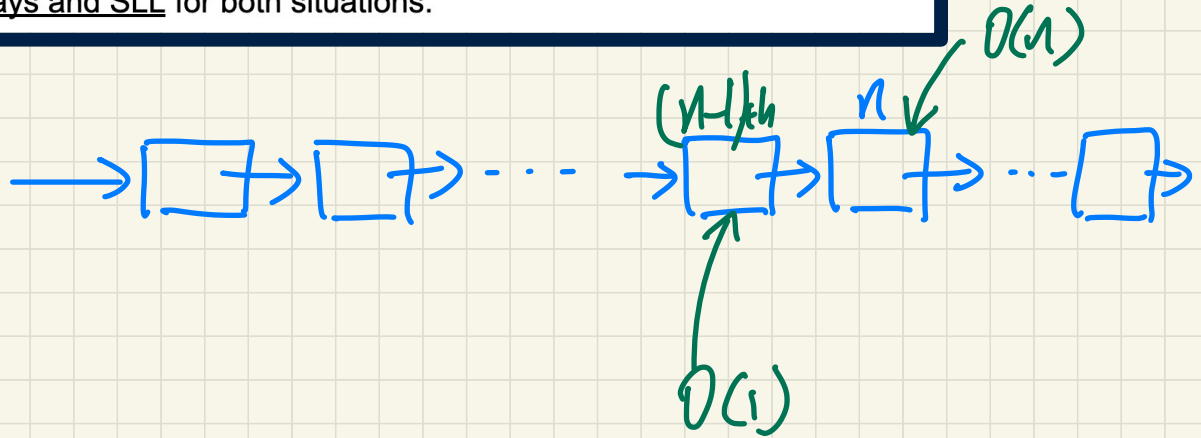


Regarding Written Test 1:
Could you explain some of the answers for this question?

> Given an array and a singly-linked list (declared with attributes `size`, `head`, and `tail`), both of size n, for each of the following operations, select the data structure that offer **better** performance (i.e., lower running time, asymptotically speaking), or select "Either" to indicate that it makes no difference by choosing either.

Specifically:
"Adding an element at index i, with the reference to the (i - 1)th position given" → **SLL**
"Adding an element at index i, with the reference to the (i)th position given." → **Either**

- I don't understand how these would have different answers. I would just expect a linear time for both arrays and SLL for both situations.



- Your development under your control
  - ↳ always DLL

- ProgTest, WT, Exam
  - ↳ SLL is more likely.

Problem on SLL: Shifting the List to the Right

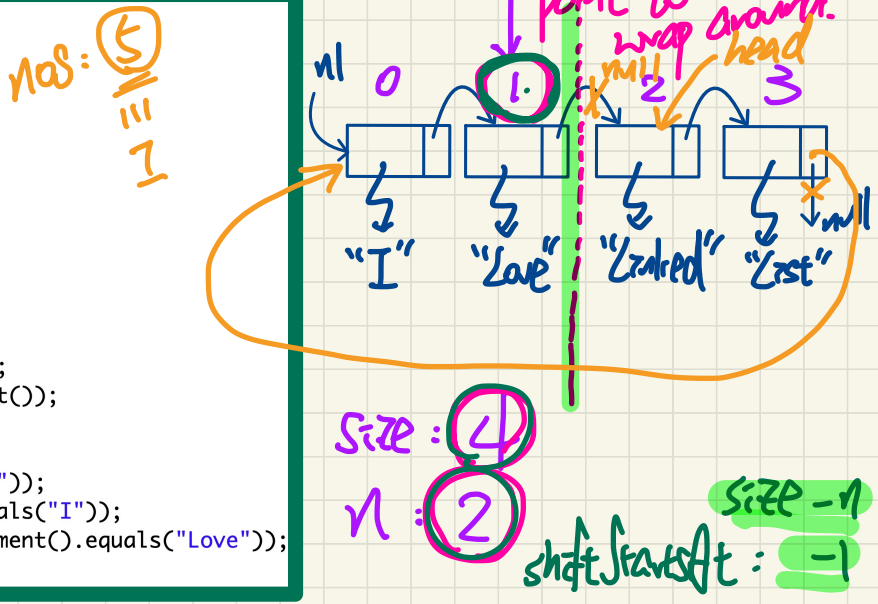
You are asked to program this method:

public Node<S.> shiftedToRightBy(**Node<S.>** head, **int** n)

Return the same chain, with nodes being shifted to the right by n positions.

Assumptions: head is not null and n ≥ 0

```
@Test
public void test2() {
    ListUtilities<String> util = new ListUtilities<>();
    Node<String> n4 = new Node<>("Lists", null);
    Node<String> n3 = new Node<>("Linked", n4);
    Node<String> n2 = new Node<>("Love", n3);
    Node<String> n1 = new Node<>("I", n2);

    Node<String> output = util.shiftedToRightBy(n1, 2);
    assertTrue(output == n3);
    assertTrue(output.getNext() == n4);
    assertTrue(output.getNext().getNext() == n1);
    assertTrue(output.getNext().getNext().getNext() == n2);
    assertNull(output.getNext().getNext().getNext().getNext());

    assertTrue(output.getElement().equals("Linked"));
    assertTrue(output.getNext().getElement().equals("Lists"));
    assertTrue(output.getNext().getNext().getElement().equals("I"));
    assertTrue(output.getNext().getNext().getNext().getElement().equals("Love"));
}
```

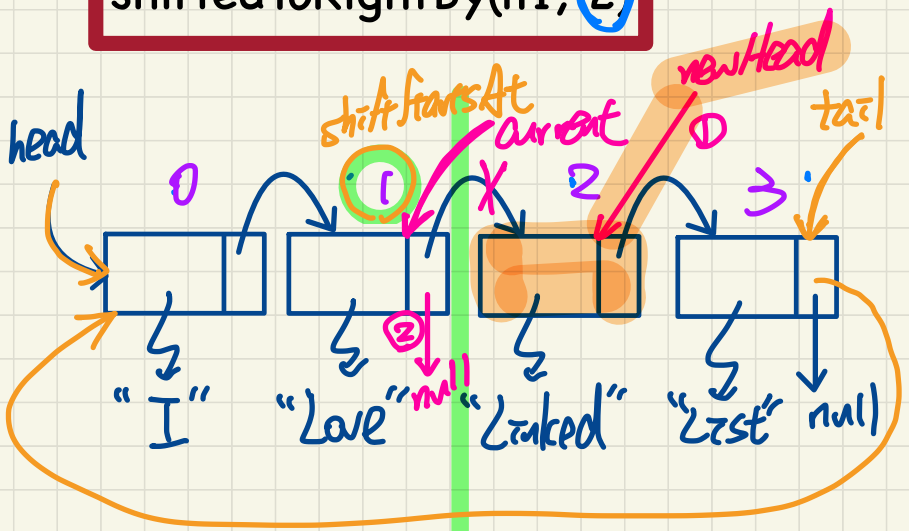


Tracing: shiftedToRightBy

```
public Node<E> shiftedToRightBy(Node<E> head, int n) {
    int size = 0;
    Node<E> current = head;
    Node<E> tail = null;
    while(current != null) {
        size ++;
        tail = current;
        current = current.getNext();
    }

    if(size == 0) {
        return null;
    }
    else {
        int nos = n % size; /* number of shifts */
        int shiftStartsAt = size - nos - 1;

        if(n == 0 || nos == 0) {
            return head;
        }
        else {
            current = head;
            for(int i = 0; i < shiftStartsAt; i ++) {
                current = current.getNext();
            }
            Node<E> newHead = current.getNext();
            current.setNext(null);
            tail.setNext(head);

            return newHead;
        }
    }
}
```

2 % 4 = 2

shiftedToRightBy(n1, 2)



nos: 2

4 - 2 - 1 = 1

Implementing an algorithm

1. Visualize the normal cases
2. Implement the normal cases
3. Write test cases to make sure 2) correct.
4. Shift the focus from code to Junit tests

e.g. shiftedToRightBy(n1, n)