CS 61B Spring 2026

Linked Lists, ArrayLists, Deques, Default

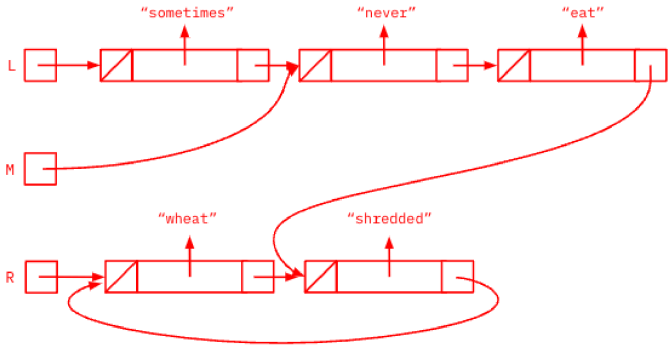
Exam-Level Discussion 04: February 09, 2026

1 Cardinal Directions

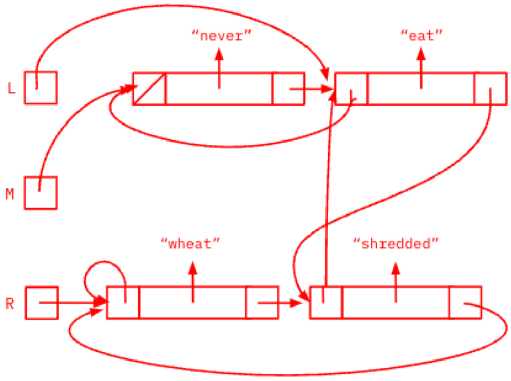
Draw the box-and-pointer diagram that results from running the following code. A `DLLStringNode` is similar to a `Node` in a `DLList`. It has 3 instance variables: `prev`, `s`, and `next`.

```
public class DLLStringNode {
  DLLStringNode prev;
  String s;
  DLLStringNode next;

  public DLLStringNode(DLLStringNode prev, String s, DLLStringNode next) {
    this.prev = prev;
    this.s = s;
    this.next = next;
  }

  public static void main(String[] args) {
    DLLStringNode L = new DLLStringNode(null, "eat", null);
    L = new DLLStringNode(null, "bananas", L);
    L = new DLLStringNode(null, "never", L);
    L = new DLLStringNode(null, "sometimes", L);
    DLLStringNode M = L.next;
    DLLStringNode R = new DLLStringNode(null, "shredded", null);
    R = new DLLStringNode(null, "wheat", R);
    R.next.next = R;
    M.next.next.next = R.next;
    L.next.next = L.next.next.next;

    /* Optional practice below. */

    L = M.next;
    M.next.next.prev = R;
    L.prev = M;
    L.next.prev = L;
    R.prev = L.next.next;
  }
}
```

Before optional practice:



After optional practice:



2 LinkedListDeque Rotation

```
public class LinkedListDeque61B<T> implements Deque61B<T> {
    private Node sentinel;
    private int size;
    public LinkedListDeque61B() { ... }
    private class Node {
        private T value;
        private Node next, prev;
        ...
    }
    ...
```

(a) Write the method **`rotateLeft(int x)`**. It should rotate the items in the list left by **`x`** positions. You may assume **`x`** is non-negative. **Do not call any other methods**

```
/** Rotates the Deque left by x places. Assume x is non-negative.
 *  Example: [3, 6, 9, 12, 15, 18].rotateLeft(4) yields [15, 18, 3, 6, 9, 12]. */
public void rotateLeft(int x) {
```

```
public void rotateLeft(int x) {
  if (size <= 1) {
      return;
  }

  // you can also mode x by size for efficiency
  for (int i = 0; i < x; i += 1) {
    Node oldLast = sentinel.prev;
    Node oldFirst = sentinel.next;
    Node oldSecond = oldFirst.next;

    sentinel.next = oldSecond;
    oldSecond.prev = sentinel;

    oldFirst.prev = oldLast;
    oldFirst.next = sentinel;

    oldLast.next = oldFirst;
    sentinel.prev = oldFirst;

    // note: you must have either if (size <= 1)
    // OR you must mode x by size, otherwise there is a
    // very subtle bug which causes the list to
    // enter a bad state if you rotate a size one list.
    // try it!
  }

}
```

(b) Often, it's easier to write methods (e.g. **`resize`** in project 2) in terms of other methods you've already written. Or more generally, when creating abstractions, you should rely on abstractions you've already created, rather than going all the way back down to the "base reality". Rewrite the **`rotateLeft`** method in terms of other **`LinkedListDeque`** methods.

```
public void rotateLeft(int x) {
```

```
        // note: you can also mod x by size for efficiency
        for (int i = 0; i < x; i++) {
            addLast(removeFirst());
        }

    }
```

3 Remove Duplicates

Using the simplified **DLList** class defined below, implement the removeDuplicates method. removeDuplicates should remove all duplicate items from the **DLList**. For example, if our initial list is [8, 4, 4, 6, 4, 10, 12, 12], our final list should be [8, 4, 6, 10, 12]. You may **not** assume that duplicate items are grouped together, or that the list is sorted!

```
public class DLList {
  Node sentinel;
  public DLList() { // Constructor not shown... }
  public class Node { int item; Node prev; Node next; }
  public void removeDuplicates() {

    Node ref = sentinel.next;

    Node checker;

    while (ref != sentinel) {

      checker = ref.next;

      while (checker != sentinel) {

        if (ref.item == checker.item) {

          Node checkerPrev = checker.prev;
          Node checkerNext = checker.next;

          checkerPrev.next = checker.next;

          checkerNext.prev = checker.prev;

        }

        checker = checker.next;

      }

      ref = ref.next;

    }
  }
}
```

4 Stack Inheritance

Suppose we have a **MyStack** interface that we want to implement. We want to add two default methods to the interface: **insesrtAtBottom** and **flip**. Fill in these methods in the code below.

```
public interface MyStack<E> {
    void push(E element); // adds an element to the top of the stack
    E pop();              // removes and returns the top element of the stack
    boolean isEmpty();    // returns true if the stack is empty
    int size();           // returns the number of elements in the stack

    // inserts the item at the bottom of the stack using push, pop, isEmpty, and size
    private void insertAtBottom(E item) {

    if (isEmpty()) {
        push(item);
        return;
    }

    E topElem = pop();
    insertAtBottom(item);
    push(topElem);

    }

    // flips the stack upside down (hint: use insertAtBottom)
    default void flip() {

    // Base case
    if (isEmpty()) {
        return;
    }
    // Pop top
    E topElem = pop();
    // Recursively reverse the remainder
    flip();
    // Insert the popped element at the bottom
    insertAtBottom(topElem);

    }
}
```