

1 Cardinal Directions (Lecture 7)

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**. You may write each String inside the **s** box rather than drawing an arrow from **s** to a String.

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

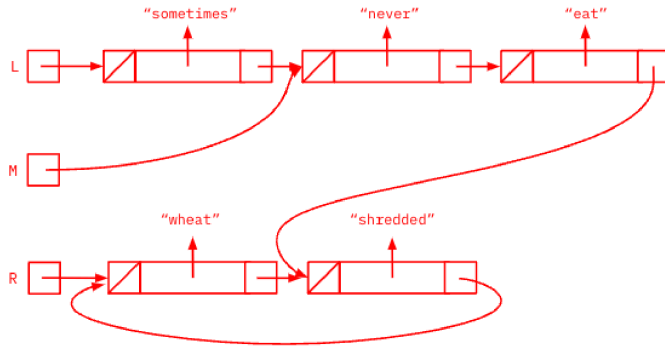
    public DLLStringNode(DLLStringNode prev, String s, DLLStringNode next) {
        this.prev = prev;
        this.s = s;
        this.next = next;
    }

    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;

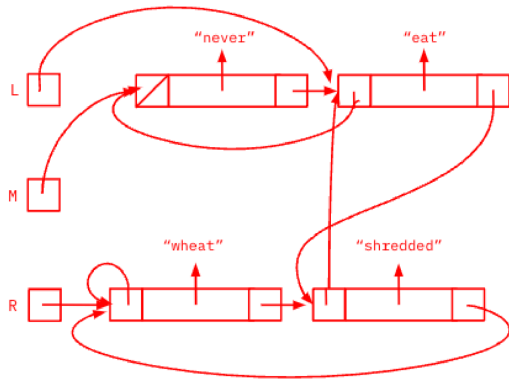
        /* Optional practice below. */

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

Before optional practice:

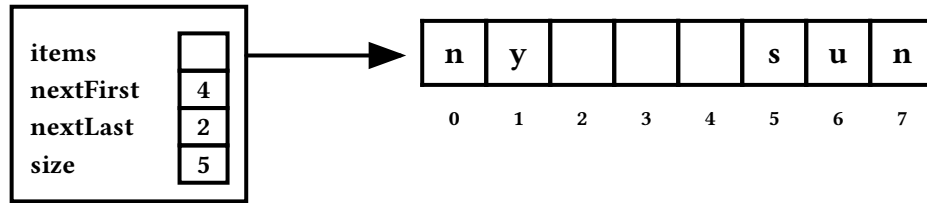


After optional practice:



2 In Circles (Lecture 8)

Consider an ArrayDeque with state shown below.

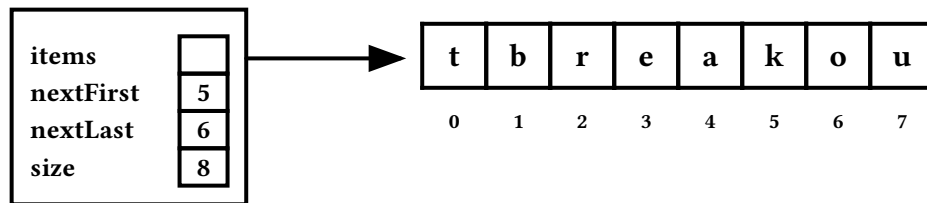


- (a) What abstract list of elements does this data structure represent? Write them out from first to last. How do you know, i.e. what invariant do you have in mind?

“s”, “u”, “n”, “n”, “y”

The **nextFirst** pointer always resides before the first element, so we know the Deque starts at 5.

Consider a completely new **ArrayDeque**. It’s looking pretty cozy in that backing array!



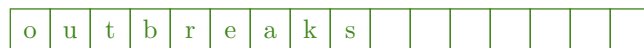
- (b) What list of elements does this new object represent? Write them out from first to last. Note: If your invariant from part (a) was well written, this should help you avoid mistakes.

“o”, “u”, “t”, “b”, “r”, “e”, “a”, “k”

- (c) Draw what the backing array would look like after calling **addLast("s")** on this data structure. What would **nextFirst**, **nextLast**, and **size** be equal to? There is more than one correct solution.

Assume this data structure uses a scaling factor of 2.

The solution space for this problem is large. As long as the array is of size 16, the ordering of elements is preserved, and there are no weird gaps, the answer is correct. Here are two possibilities:



nextFirst: 15, nextLast: 9, size: 9



nextFirst: 9, nextLast: 3, size: 9

3 Off to the Races (Lecture 7)

Consider the following Lists, using the **SLList**, **DLList**, and **AList** classes as implemented in lecture. As a reminder,

- **SLList** is a singly-linked list with a sentinel.
- **DLList** is a circular doubly-linked lists with a sentinel.
- **AList** is an array list with dynamic resizing.

```
public SLList clementineSurprise = new SLList();
public DLList superstitiousRealism = new DLList();
public AList jovialMerrymment = new AList();
```

Assume that each list contains 1000 items. For each set of blocks of code below, circle the one that you'd expect to run **faster** and briefly explain why.

- (a)

<pre>for (int i = 0; i < 100; i += 1) { clementineSurprise.addLast(); }</pre>	<pre>for (int i = 0; i < 200; i += 1) { superstitiousRealism.addLast(); }</pre>
--	--

We'd expect the **right** one to run faster on average. Finding the last element of an **SLList** is a costly operation, requiring us to iterate through the entire list. Each **addLast** operation runs in linear time. In contrast, we can access **sentinel.prev** to **addLast** to a **DLList**, meaning each **addLast** operation runs in linear time. Since our lists contain 1000 elements each, 100 linear-time operations is slower than 200 constant-time operations in this case.

- (b)

<pre>for (int i = 0; i < 900; i += 1) { jovialMerrymment.removeLast(); }</pre>	<pre>for (int i = 0; i < 200; i += 1) { clementineSurprise.removeLast(); }</pre>
---	---

We'd expect the **left** one to run faster on average. Removing the last element of an **SLList** is a costly operation, for the same reasons detailed above. In contrast, **removeLast** runs in constant time for **ALists**, since we can access the last element with **nextLast - 1**.

- (c)

<pre>for (int i = 0; i < 300; i += 1) { superstitiousRealism.get(i); }</pre>	<pre>for (int i = 0; i < 300; i += 1) { jovialMerrymment.get(i); }</pre>
---	---

We'd expect the **right** one to run faster on average. **get** is very slow in general on linked lists, requiring us to iterate through the list until the desired index is reached. It runs in linear time. In contrast, **get** is a constant-time operation for **ALists**.

- (d) Imagine that we add a **last** pointer to our **SLList** class that keeps track of the last element of the list. We then modify **addLast** and **removeLast** to access **last** and update it accordingly. Which of your answers, if any, will change, and why?

Only (a) would change.

Implementing a **last** pointer would greatly speed up successive **addLast** calls, since we can update **last** to the list node that we add. Each **addLast** call now runs in constant time.

However, **removeLast** isn't any faster, since updating **last** requires iterating through the whole list. It's still a linear-time operation, hence why (b) is unchanged.

4 Remove Duplicates (Lecture 7)

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! Do not worry about efficiency.

```
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;
        }
    }
}
```