CS 61B Spring 2026

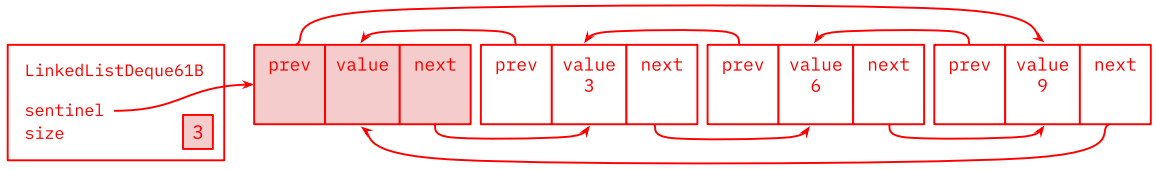
Linked Lists, ArrayLists, Deques, Default

Discussion 04: February 09, 2026

1 LinkedListDeque Rotation

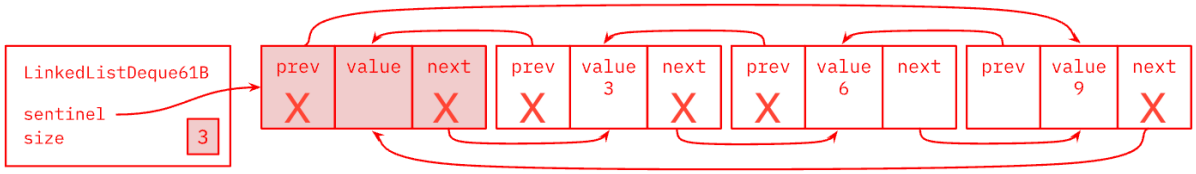
```
public class LinkedListDeque61B<T> implements Deque61B<T> {
    private Node sentinel;
    private int size;
    public LinkedListDeque61B() { ... }
    private class Node {
        private T value;
        private Node next, prev;
        ...
    }
    ...
```

(a) Draw a box and pointer diagram for a **LinkedListDeque61B** that contains the items 3, 6, then 9 in that order. The **LinkedListDeque61B** object should have two boxes for **sentinel** and **size** respectively. Each **Node** should have three memory boxes for **prev**, **value**, and **next** respectively.



(b) The Deque you drew has the numbers [3, 6, 9]. Suppose we rotate the list left by one. Left rotation means that every number moves left by one, and the leftmost number wraps around to the end. Thus after one rotation we'd have [6, 9, 3].

Mark the memory boxes on the diagram that must be changed in order to carry out a rotation. There should be six.



(c) 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** Each rotation operation should set six pointers. Don't spend too much time on this problem. If it gets too annoying, you'll see a much much simpler way to do this in part d.

```
/** 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) {
  if (size <= 1) { return; }

  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;
  }
```

(d) 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) {

  if (size <= 1) { return; }
  for (int i = 0; i < x % size; i++) {
      addLast(removeFirst());
  }

}
```

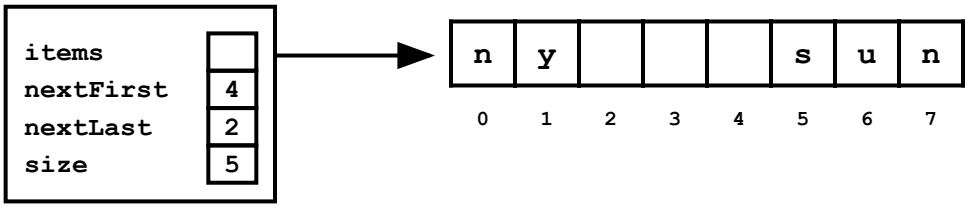
2 Default Queue

When we write **`default`** methods, we have no choice but to use our existing abstractions. This is because we don't have any knowledge of or access to the underlying base reality. Suppose we have a **`MyQueue`** interface that we want to implement. We want to include three default methods in the interface: **`clear`**, **`remove`** and **`appendAll`**. Fill in these methods in the code below.

```
public interface MyQueue<E> {
    void enqueue(E element); // adds an element to the end of the queue
    E dequeue();             // removes and returns the front element of the queue
    boolean isEmpty();       // returns true if the queue is empty
    int size();              // returns the number of elements in the queue
    // removes all items from the queue
    default void clear() {

      while (!isEmpty()) {
        dequeue();
      }

    }
    // removes all items equal to item from the queue
    // the remaining items should be in the same order as they were before
    // use .equals to compare items rather than ==
    default void remove(E item) {

      int i = 0;
      int currSize = size();
      while (i < currSize) {
          E currItem = dequeue();
          if (!currItem.equals(item)) {
            enqueue(currItem);
          }
          i += 1;
      }

    }
    // appends all items from the other queue into this queue
    // this method should be non-destructive on the otherQueue!
    default void appendAll(MyQueue<E> otherQueue) {

      int n = otherQueue.size();
      for (int i = 0; i < n; i += 1) {
        E item = otherQueue.dequeue();
        enqueue(item);
        otherQueue.enqueue(item);
      }

    }
}
```

3 In Circles

Consider an ArrayDeque with state shown below.

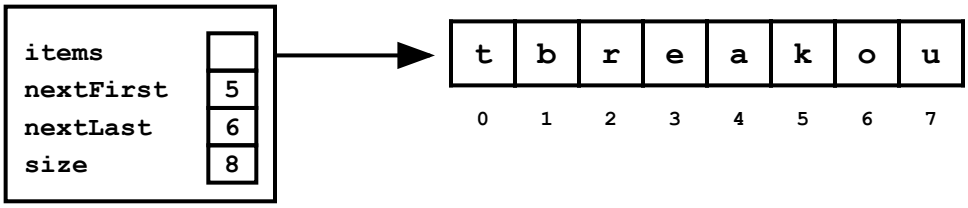


(a) What abstract list of elements does this data structure represent? Write them out from first to last. How do you know?

"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.

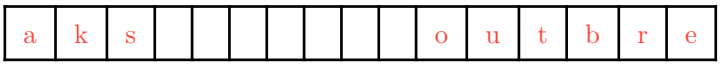
"o", "u", "t", "b", "r", "e", "a", "k"

(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