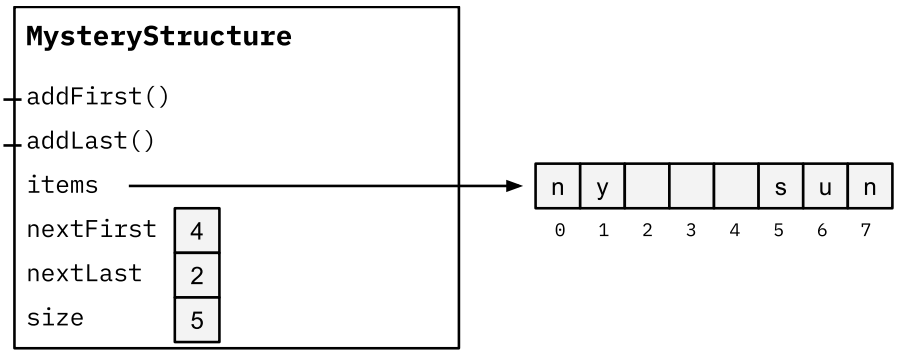
CS 61B
Fall 2025

Inheritance, Comparators, Generic Functions

Discussion 03: September 15, 2025

1 In Circles

Consider the mystery class **`MysteryStructure`**, with the state shown below.



(a) Based on what you can infer from the class's associated methods and instance variables, what data structure does this class represent?

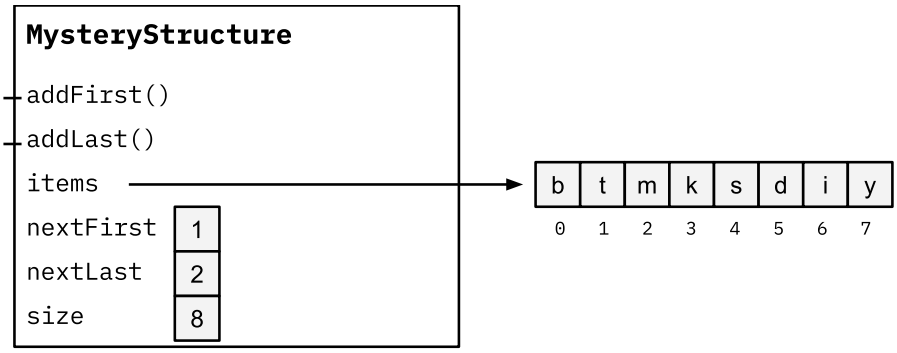
This is a Deque, as is evident by the **`addFirst`** and **`addLast`** methods. It is implemented using a circular backing array.

(b) What 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 **`MysteryStructure`**. It's looking pretty cozy in that backing array!



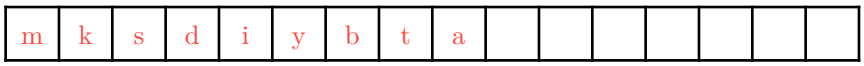
(c) What list of elements does this new object represent? Write them out from first to last.

"m", "k", "s", "d", "i", "y", "b", "t"

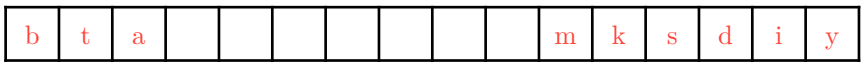
(d) Draw what the backing array would look like after calling **`addLast("a")`** on this data structure. What would **`nextFirst`**, **`nextLast`**, and **`size`** be equal to?

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

2 It's a Bird! It's a Plane! It's a CatBus!

(a) On a research expedition studying air traffic, we discovered a new species: the Flying Interfacing **CatBus**, which acts like a vehicle and has the ability to make noise (safety is important!).

Given the **Vehicle** and **Noisemaker** interfaces, fill out the **CatBus** class so that **CatBus**es can rev their engines and make noise at other **CatBus**es with a **CatBus**-specific sound.

```
interface Vehicle {
    public void revEngine();
}

interface Noisemaker {
    public void makeNoise();
}


public class CatBus implements Vehicle, Noisemaker {
    @Override
    public void revEngine { /* CatBus revs engine, code not shown */ }

    @Override
    public void makeNoise { /* CatBus makes noise, code not shown */ }

    /** Allows CatBus to make noise at other CatBuses. */
    public void conversation(CatBus target) {
        makeNoise();
        target.makeNoise();
    }
}
```

(b) It's a lovely morning in the skies and we've encountered a horrible **Goose**, which also **implements Noisemaker** (it has a knife in its beak!). Modify the **conversation** method signature so that **CatBus**es can **makeNoise** at both **CatBus** and **Goose** objects while only having one argument, **target**.

We can change the method signature so that the type of the parameter **target** is **Noisemaker** (both **CatBus** and **Goose** implement **Noisemaker**):

```
/** Allows CatBus to make noise at other both CatBuses and Gooses. */
public void conversation(Noisemaker target) {
    makeNoise();
    target.makeNoise();
}
```

3 Default

Suppose we have a **MyQueue** interface that we want to implement. We want to add two default methods to the interface: **clear**, **remove** and **max**. 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
    default void remove(E item) {

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

    }

    // returns the maximum element in the queue according to the comparator
    // the items in the queue should be in the same order as they were before
    // assume the queue is not empty
    default E max(Comparator<E> c) {

      int removed = 0;
      int currSize = size();
      E currMax = null;
      while (removed < currSize) {
        E currItem = dequeue();
        if (currMax == null) {
            currMax = currItem;
        } else if (c.compare(currItem, currMax) > 0) {
            currMax = currItem;
        }
        enqueue(currItem);
        removed++;
      }
      return currMax;
```

```
    }
}
```