Week 4 – Monday

COMP 2100

Last time

- What did we talk about last time?
- Stack implementation with arrays
- Queues

Questions?

Project 1

Bitmap Manipulator

Queues

Queue

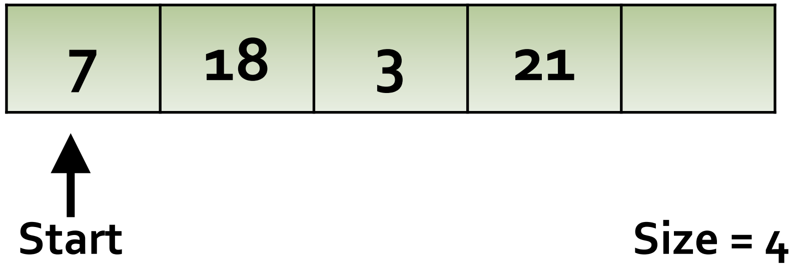
- A **queue** is a simple data structure that has three basic operations (very similar to a stack)
  - **Enqueue** Put an item at the back of the queue
  - **Dequeue** Remove an item from the front of the queue
  - **Front** Return the item at the front of the queue
- A queue is considered FIFO (First In First Out) or LILO (Last In Last Out)

Circular array

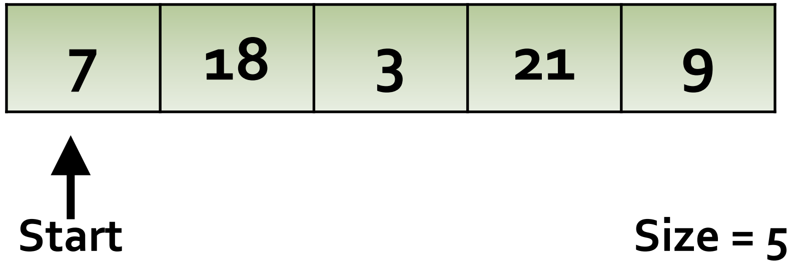
- A **circular array** is just a regular array
- However, we keep a **start** index as well as a **size** that lets us start the array at an arbitrary point
- Then, the contents of the array can go past the end of the array and wrap around
- The modulus operator (%) is a great way to implement the wrap around

Circular array example

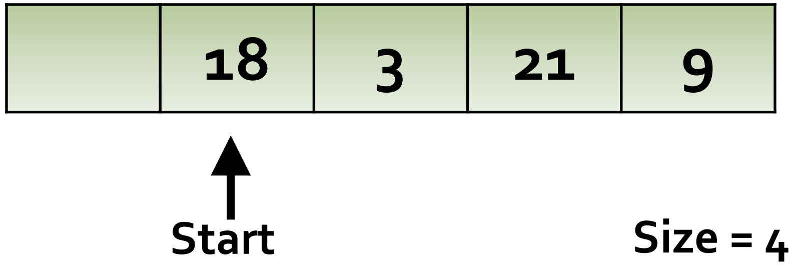
1. Starting array



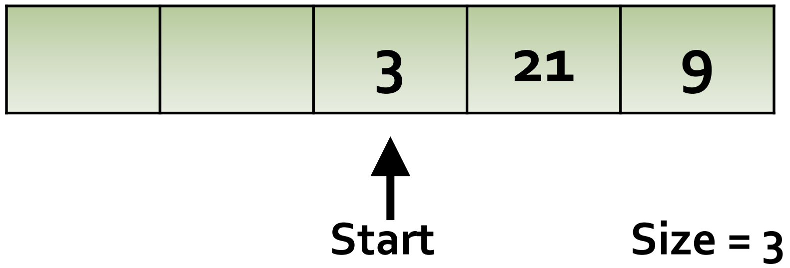
2. Enqueue 9



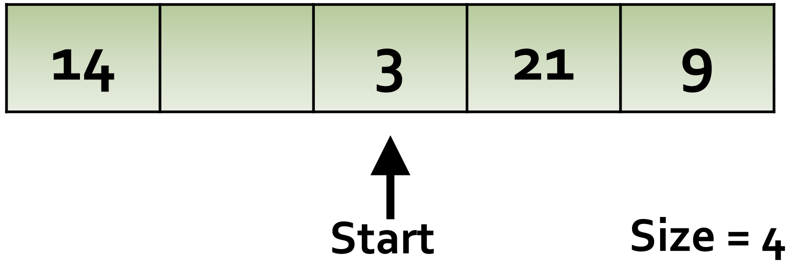
3. Dequeue



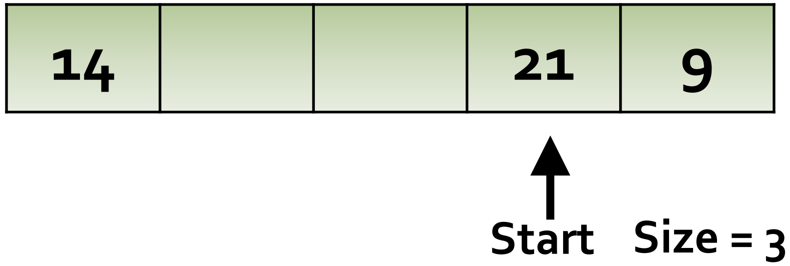
4. Dequeue



5. Enqueue 14



6. Dequeue



Circular array implementation

- Advantages:
  - Dequeue is $\Theta(1)$
  - Front is $\Theta(1)$
- Disadvantages
  - Enqueue is $\Theta(n)$ in the very worst case, but not in the amortized case

Circular array implementation

```
public class ArrayQueue {
  private E[] data = (E[]) new Object[10];
  private int start = 0;
  private int size = 0;

  public void enqueue(E value) {}
  public E dequeue() {}
  public E front() {}
   public int size() {}
}
```

Circular Array Front

Circular Array Get Size

Circular Array Enqueue

Circular Array Dequeue

JCF Stacks and Queues

`Deque<T>`

- Java does have a **`Stack`** class which extends **`Vector`**
- The **`Deque`** (double ended queue, pronounced like *deck*) interface is preferred
- A double ended queue can be used as either stack or queue

Stack Operation	Deque Method
Push	`addFirst(T element)`
Pop	`removeFirst()`
Top	`peekFirst()`
Size	`size()`

Queue Operation	Deque Method
Enqueue	`addLast(T element)`
Dequeue	`removeFirst()`
Front	`peekFirst()`
Size	`size()`

`ArrayDeque<T>`

- Since **`Deque`** is an interface, we have to have classes that can implement it
- **`ArrayDeque`** is an implementation of a double ended queue that uses a circular array for backing
- Probably the best choice for both queues and stacks in terms of speed and memory use
  - **`addFirst()`** (push) is $\Theta(1)$ amortized
  - **`addLast()`** (enqueue) is $\Theta(1)$ amortized
  - **`removeFirst()`** (pop and dequeue) is $\Theta(1)$
  - **`peekFirst()`** (top and front) is $\Theta(1)$

LinkedList<T>

- Good old **LinkedList** is an implementation of a double-ended queue that uses a doubly-linked list for backing
- Generally slower than **ArrayDeque**, but the important operations are Θ(1) without being amortized
  - **addFirst()** (push) is Θ(1)
  - **addLast()** (enqueue) is Θ(1)
  - **removeFirst()** (pop and dequeue) is Θ(1)
  - **peekFirst()** (top and front) is Θ(1)

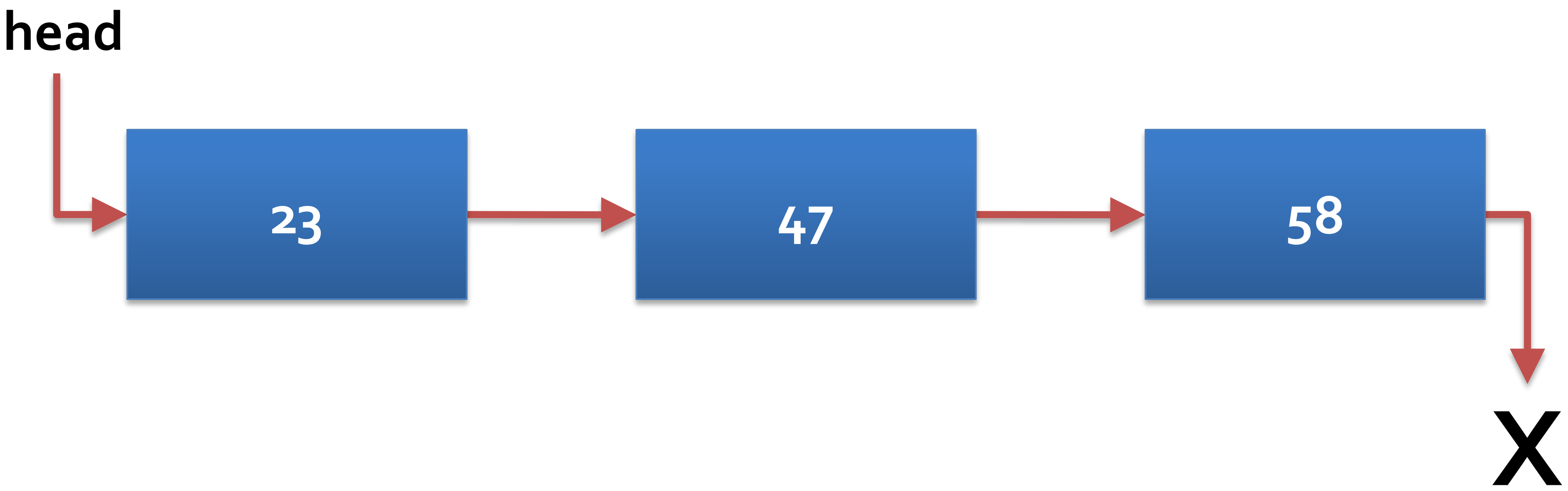
Priority queues

- A **priority queue** is like a regular queue except that items are not always added at the end
- They are added to the place they need to be in order to keep the queue sorted in priority order
- Not all requests are created equal
  - A higher priority job can come along and jump in front of a lower priority job
- Unfortunately, we have to wait for the **heap** data structure to implement priority queues efficiently

Linked Lists

Linked lists

- What is a linked list?
- Why not just use (dynamic) arrays for everything?



Pros

- Insert at front (or back)
  - $\Theta(1)$
- Delete at front (or back)
  - $\Theta(1)$
- Arbitrary amounts of storage with low overhead

Cons

- Search
  - $\Theta(\boldsymbol{n})$
- Go to index
  - $\Theta(\boldsymbol{n})$
- Potentially significant memory overhead if data is small
- Much easier to make pointer and memory errors (especially in C/C++)

Implementations

Levels of flexibility

- Class protecting nodes implementation
- Generic class providing nodes with arbitrary type
- Generic class with the addition of iterators

Wait, what's an iterator?

- I'm glad you asked
- They allow a collection to be used in an enhanced for loop
- So, what's an enhanced for loop?

```
public static int sum(int[] array) {
    int total = 0;
    for (int value: array)
        total += value;
    return total;
}
```

- It allows you to read (but not change) each value in a list

So what?

- Enhanced **`for`** loops work for any iterable list of any type

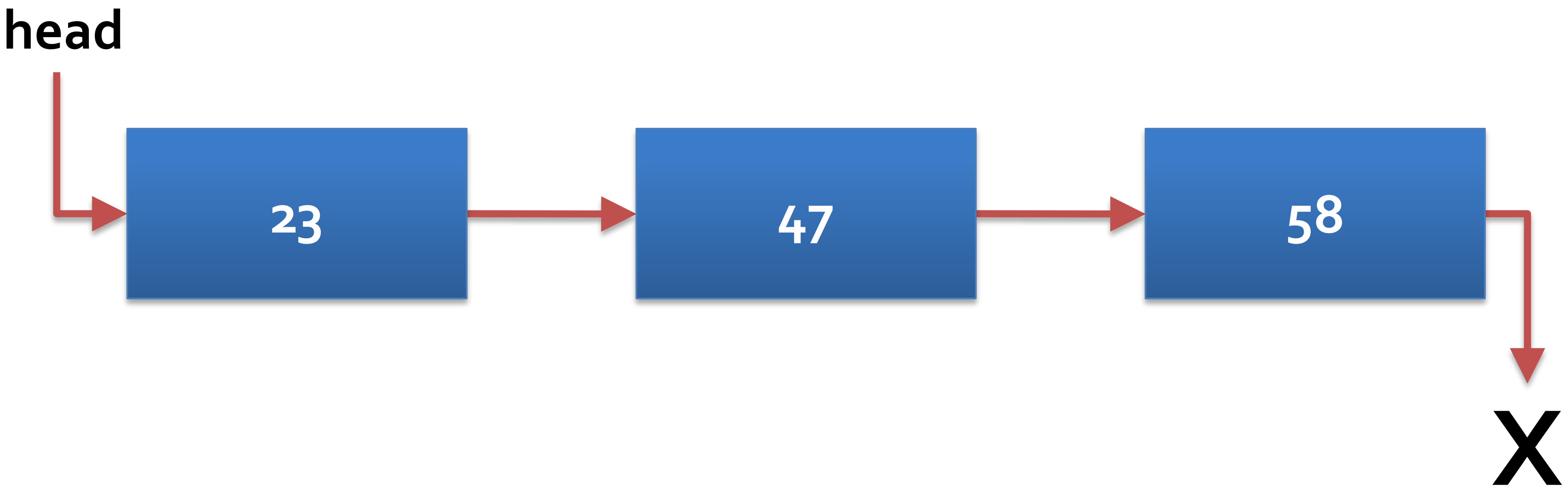
```
public static double weigh(Wombat[] list) {
        double total = 0.0;
        for (Wombat wombat: list)
                total += wombat.getWeight();
        return total;
}
```

```
public static double weigh(ArrayList<Wombat> list) {
        double total = 0.0;
        for (Wombat wombat: list)
                total += wombat.getWeight();
        return total;
}
```

```
public static double weigh(LinkedList<Wombat> list) {
        double total = 0.0;
        for (Wombat wombat: list)
                total += wombat.getWeight();
        return total;
}
```

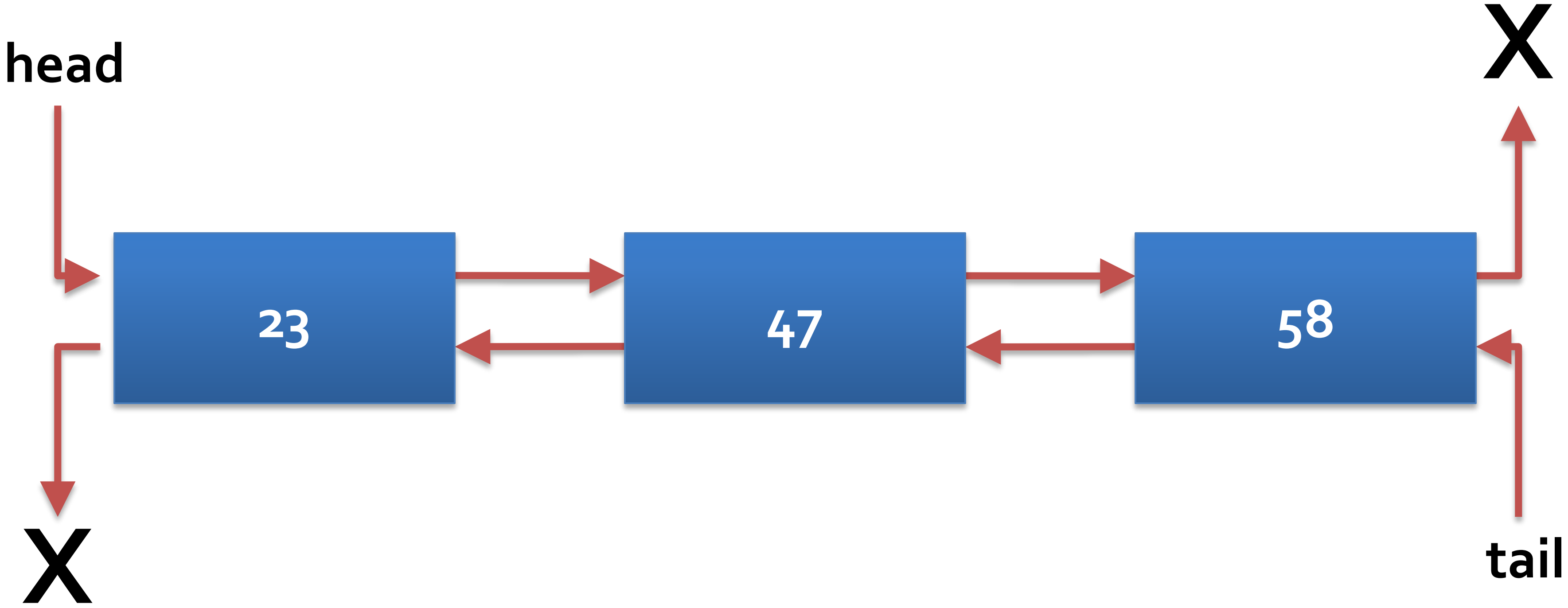
Singly linked list

- Node consists of data and a single next pointer
- Advantages: fast and easy to implement
- Disadvantages: forward movement only



Doubly linked list

- Node consists of data, a next pointer, and a previous pointer
- Advantages: bi-directional movement
- Disadvantages: slower, 4 pointers must change for every insert/delete



Interview question

- You are given a singly linked list
- It may have a loop in it, that is, a node that points back to an earlier node in the list
- If you try to visit every node in the list, you'll be in an infinite loop
- How can you see if there is a loop in a linked list?

Upcoming

Next time…

- Implementation of a linked list
- Circular linked lists and skip lists
- Implementing a stack with a linked list

Reminders

- Keep reading section 1.3
- Keep working on Project 1
  - **Due this Friday, September 20 by midnight**
- Exam 1 next Monday