Week 11 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Dynamic data structures
- Linked lists

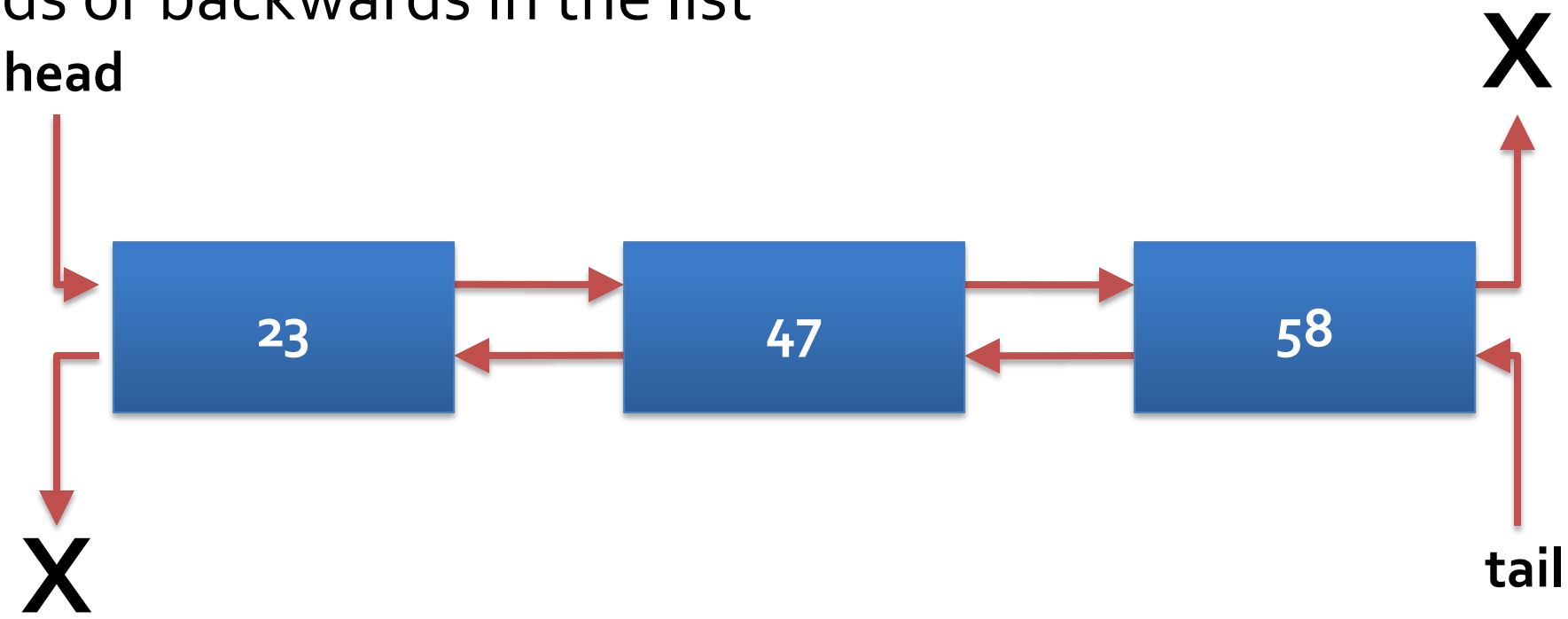
Questions?

Project 3

Linked Lists

Doubly linked list

- The most common library implementation of a linked list is a **doubly linked list**
- Node consists of data, a next pointer, and a previous pointer
- Because we know the next and the previous, we can move forwards or backwards in the list



Definition

- Let's try a simple definition for a doubly linked list that holds an unlimited number of **`String`** values:

```
public class LinkedList {
   private static class Node {
        public String data;
        public Node next;
        public Node previous;
   }

   private Node head = null
   private Node tail = null;
   private int size = 0;
   …
}
```

Find the index of an element

- Method signature:

```
public int indexOf(String value)
```

- Loop through the list until reaching a node whose **`data`** is equal to **`value`**, keeping a counter of the current index
- If **`value`** is found, return the index
- If **`value`** is never found, return **`-1`**

Generics

Containers

- When you write a container class (like a list), you have to write it to contain something
  - A list of **`String`** values
  - A list of **`Wombat`** values
  - A list of **`int`** values
- What if we could design a list class and not specify what its contents are?
- Someone has to say what it contains only when they make a particular list

Generics

- That's the idea behind **generics** in Java
  - The name is because it lets you make a generic list instead of a specific kind of list
- You can make classes (often, but not always, containers)
- These classes have one or more **type parameters**
- The type parameters are like variables that hold type information
- When you make such an object, you have to say what its types are

Angle brackets

- Influenced by templates in C++, Java puts type parameters in angle brackets (**<>**)
- For example, we can declare the following **`LinkedList`** objects defined in the Java Collections Framework

```
LinkedList<String> words = new LinkedList<String>();
LinkedList<Wombat> zoo = new LinkedList<Wombat>();
LinkedList<Integer> numbers = new LinkedList<Integer>();
```

- For technical reasons, you can only use reference types for type parameters, never primitive types

Details

- You can only use type parameters on classes that were designed from the beginning to be generic
  - You can't force a class to take type parameters
- But you can leave off type parameters, what are called raw types
  - You'll get a warning
  - Java assumes that you use **`Object`** as the type parameter by default
- For convenience, you can often leave them out in the instantiation step (after the **`new`** keyword)
- Java can often infer what the types should be:

```
LinkedList<String> words = new LinkedList<>();
```

Primitive types

- Although you can't use primitive types as type parameters, every primitive type has a corresponding wrapper type
  - `boolean:` `Boolean`
  - `byte:` `Byte`
  - `char:` `Character`
  - `short:` `Short`
  - `int:` `Integer`
  - `long:` `Long`
  - `float:` `Float`
  - `double:` `Double`

Boxing and unboxing

- If you use the wrapper class as the type parameter, Java will automatically convert primitive types to and from the wrapper class
- This is called boxing and unboxing
- For example:

```
LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(7);
numbers.add(15);
int value = numbers.get(0);   // Holds 7
```

- For the most part, it magically works
- However, storing primitive types is less efficient

Creating Generic Classes

Creating generic classes

- For the most part, you will use libraries that have generic classes in them
- You will rarely need to design your own generic class
- Nevertheless, you will sometimes need to extend generic classes or implement generic interfaces
- It's good to know how it all works

Type parameter syntax

- When declaring a generic class, put angle brackets and the type parameter after the name of the class
- The type parameter is often called **T**, standing for type
- Consider a simple generic class that holds a pair of…anything

```
public class Pair<T> {
    private T x;
    private T y;
    public Pair(T x, T y) {
        this.x = x;
        this.y = y;
    }
}
```

Definition

- Instead of **`String`** values, we can write a doubly linked list class that holds anything

```
public class LinkedList<T> {
   private static class Node<T> {
        public T data;
        public Node<T> next;
        public Node<T> previous;
   }

   private Node<T> head = null
   private Node<T> tail = null;
   private int size = 0;
   …
}
```

Generic add to the end of the list

- Method signature:

```
public void add(T value)
```

- The method creates a new node
- If the list is empty, it points **`head`** at the new node
- Otherwise, it points the **`tail`** node's **`next`** at the new node and the new node's previous at the **`tail`** node
- It updates the **`tail`** to point at the new node
- It increases **`size`** by one

Generic get an element from the list

- Method signature:

```
public T get(int index)
```

- If **`index`** is illegal, throw an **`IndexOutOfBoundsException`**
- Loop through the list until reaching the node at location **`index`** (using 0-based indexing)
- Return the **`data`** of the node in question

Generic remove the first element

- Method signature:

```
public T remove()
```

- If the list is empty, throw a **`NoSuchElementException`**
- Point a temporary variable at the **`head`** node
- Point **`head`** at the next node
- If the next node is null, point **`tail`** at **`null`**
- Otherwise, point the next node's **`previous`** at **`null`**
- Return the **`data`** of the temporary node

Upcoming

Next time…

- Java Collections Framework
- Lists
- Sets

Reminders

- **Finish Project 3**
  - **Due tonight by midnight!**
- Keep reading Chapter 18