BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

10: Doubly Linked Lists

Alexandra Papoutsaki
she/her/hers

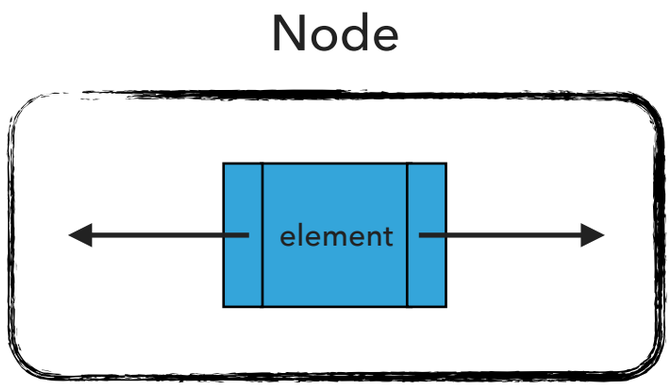
Lecture 10: Doubly Linked Lists

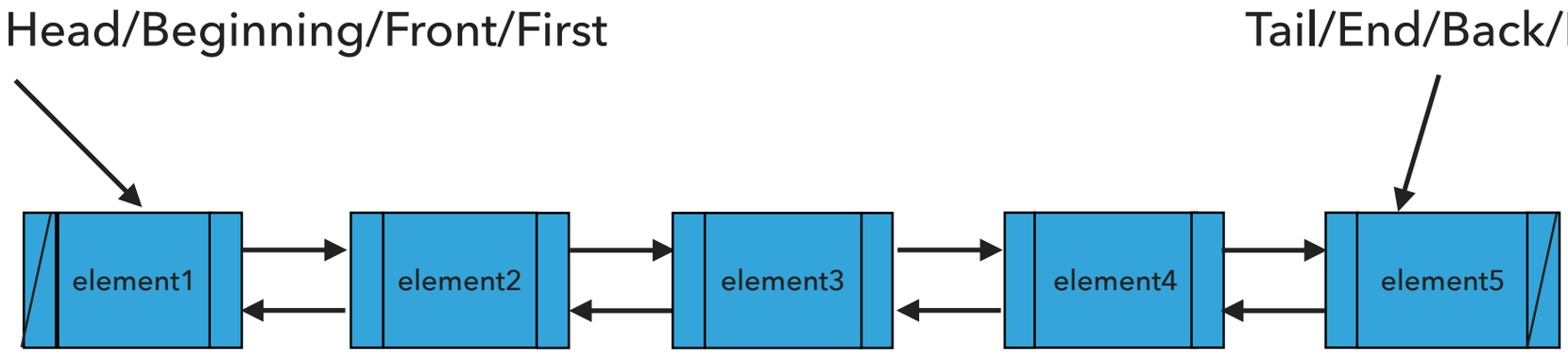
- Doubly Linked Lists
- Java Collections

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

Recursive Definition of Doubly Linked Lists

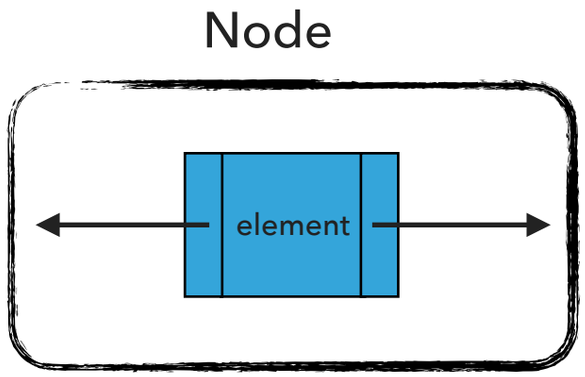
- A doubly linked list is either empty (null) or a node having a reference to a doubly linked list.
- Node: is a data type that holds any kind of data and two references to the previous and next node.





Node

```
private class Node {
    E element;
    Node next;
    Node prev;
}
```



Reminder: Interface List

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

Standard Operations

- `DoublyLinkedList()`: Constructs an empty doubly linked list.
- `isEmpty()`:Returns true if the doubly linked list does not contain any element.
- `size()`: Returns the number of elements in the doubly linked list.
- `E get(int index)`: Returns the element at the specified index.
- `addFirst(E element)`: Inserts the specified element at the head of the doubly linked list.
- `addLast(E element)`: Inserts the specified element at the tail of the doubly linked list.
- `add(E element)`: Inserts the specified element at the tail of the doubly linked list.
- `add(int index, E element)`: Inserts the specified element at the specified index.
- `E set(int index, E element)`: Replaces the specified element at the specified index and returns the old element
- `E removeFirst()`: Removes and returns the head of the doubly linked list.
- `E removeLast()`: Removes and returns the tail of the doubly linked list.
- `E remove()`: Removes and returns the head of the doubly linked list.
- `E remove(int index)`: Removes and returns the element at the specified index.
- `clear()`: Removes all elements.

`DoublyLinkedList()`: Constructs an empty DLL

head

tail

.

size

What should happen?

```
DoublyLinkedList<String> dll = new DoublyLinkedList<String>();
```

`DoublyLinkedList()`: Constructs an empty DLL

```
DoublyLinkedList<String> dll = new DoublyLinkedList<String>();
```

head = null

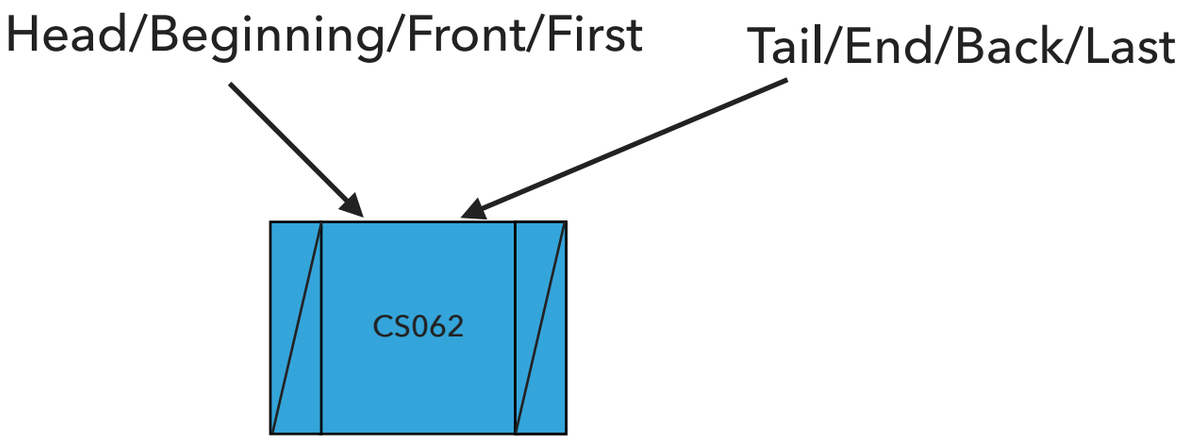
tail = null

size = 0

What should happen?

```
dll.add("CS062");
```

`add(E element)`: Inserts the specified element at the tail of the doubly linked list.



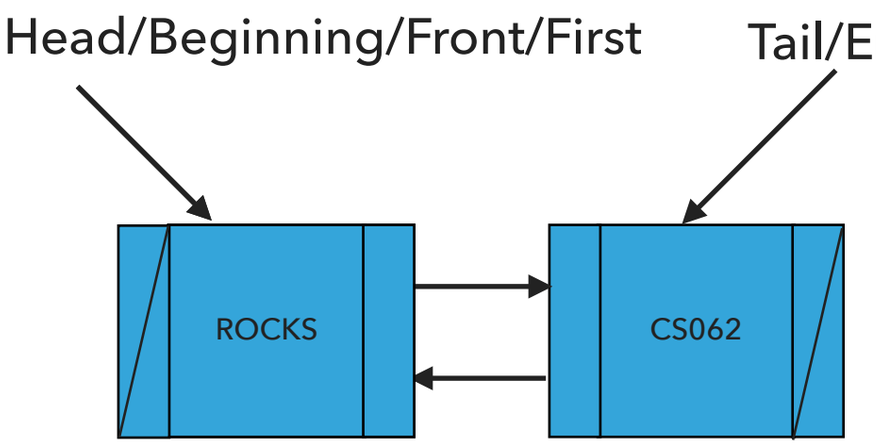

```
dll.add(“CS062”)

size=1
```

What should happen?

```
dll.addFirst(“ROCKS”);
```

`addFirst(E element)`: Inserts the specified element at the head of the doubly linked list



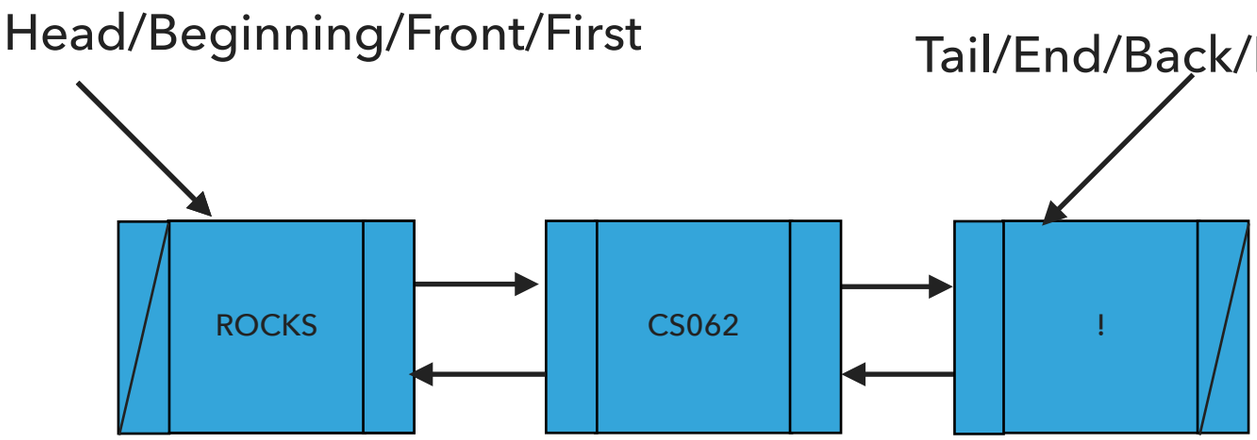

```
dll.addFirst(“ROCKS”)

size=2
```

What should happen?

```
dll.addLast(“!”);
```

`addLast(E element)`: Inserts the specified element at the tail of the doubly linked list



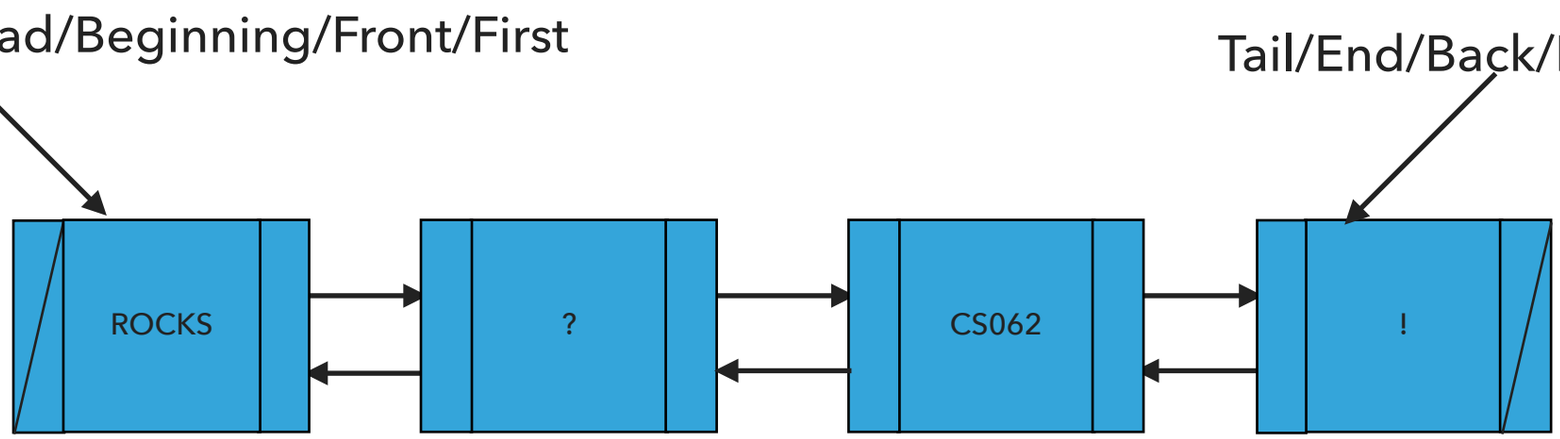

```
dll.addLast(“!”)

size=3
```

What should happen?

```
dll.add(1,“?”);
```

`add(int index, E element)`: Adds element at the specified index



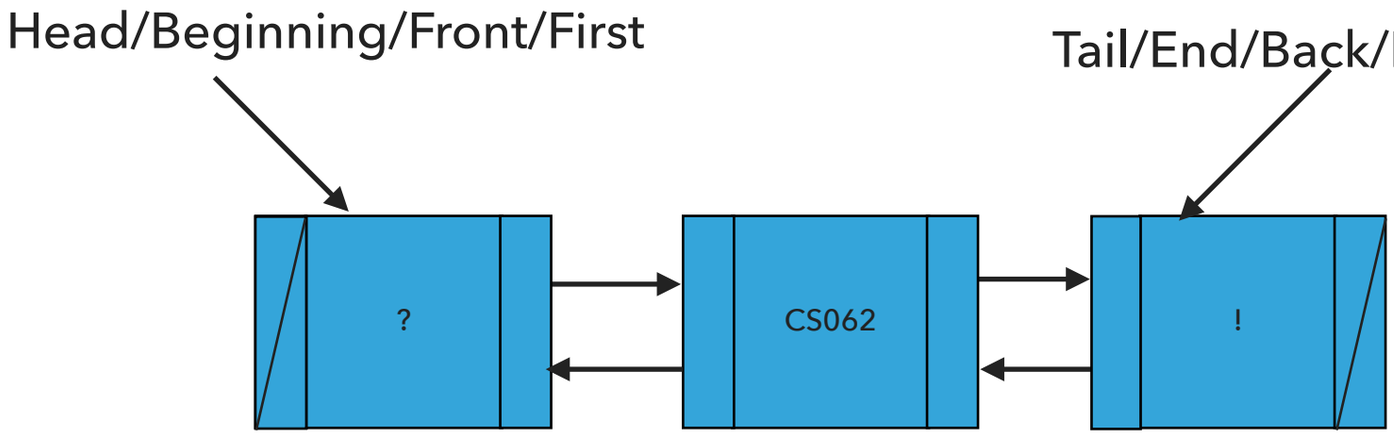

```
dll.add(1,“?”)

size=4
```

What should happen?

```
dll.remove();
```

`remove()`: Removes and returns the head of the doubly linked list



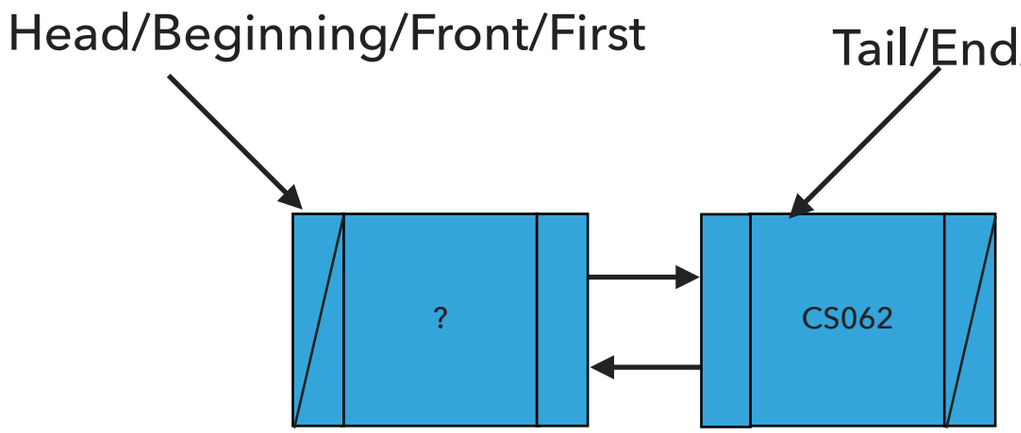

```
dll.remove()

size=3
```

What should happen?

```
dll.removeLast();
```

`removeLast()`: Removes and returns the tail of the doubly linked list



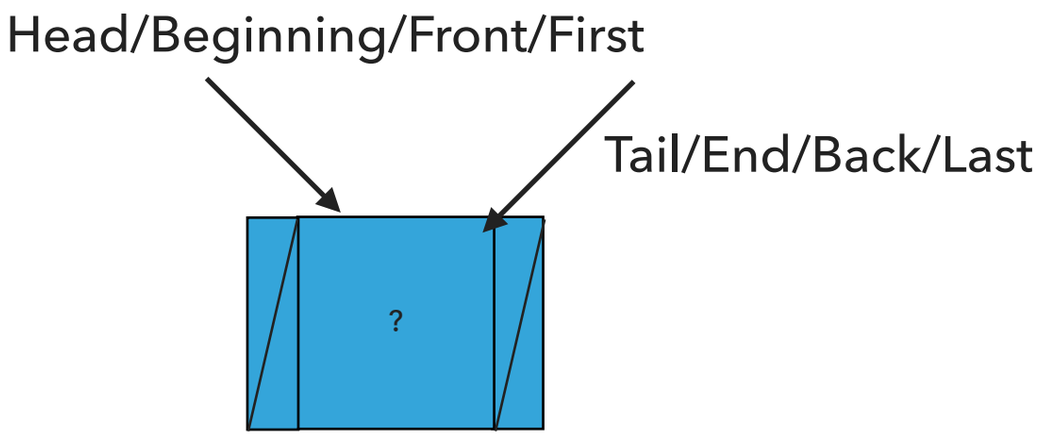

```
dll.removeLast()

size=2
```

What should happen?

```
dll.remove(1);
```

`remove(int index)`: Removes and returns the element at the specified index




```
dll.remove(1)

size=1
```

Our own implementation of Doubly Linked Lists

- We will follow the recommended textbook style.
  - It does not offer a class for this so we will build our own.
- We will work with generics because we want doubly linked lists to hold objects of an type.
- We will implement the List interface we defined in past lectures.
- We will use an inner class Node and we will keep track of how many elements we have in our doubly linked list.

Instance variables and inner class

```
public class DoublyLinkedList<E> implements List<E> {
    private Node head; // head of the doubly linked list
    private Node tail; // tail of the doubly linked list
    private int size; // number of nodes in the doubly linked list

    /**
     * This nested class defines the nodes in the doubly linked list with a value
     * and pointers to the previous and next node they are connected.
     */
    private class Node {
        E element;
        Node next;
        Node prev;
    }
```

Check if is empty and how many elements

```java
/**
 * Returns true if the doubly linked list does not contain any element.
 *
 * @return true if the doubly linked list does not contain any element
 */
public boolean isEmpty() {
    return size == 0; // or return (head == null && tail == null);
}
```

```java
/**
 * Returns the number of elements in the doubly linked list.
 *
 * @return the number of elements in the doubly linked list
 */
public int size() {
    return size;
}
```

PRACTICE TIME: Retrieve element from specified index

```
/**
 * Returns element at the specified index.
 *
 * @param index
 *            the index of the element to be returned
 * @return the element at specified index
 */
public E get(int index) {
    // check whether index is valid

    // if index is 0, return element at head


     // else if index is size-1, return element at tail


    // set a temporary pointer to the head

    // search for index-th element or end of list


    // return the element stored in the node that the temporary pointer points to
}
```

Retrieve element from specified index

```
    /**
     * Returns element at the specified index.
     *
     * @param index
     *            the index of the element to be returned
     * @return the element at specified index
     * @pre 0<=index<size
     */
    public E get(int index) {
        // check whether index is valid
         if (index >= size || index < 0){
                throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
          }
         // if index is 0, return element at head
        if (index == 0){
            return head.element;
         }
         // else if index is size-1, return element at tail
        else if (index == size - 1){
            return tail.element;
         }
        // set a temporary pointer to the head
        Node finger = head;
        // search for index-th element or end of list
        while (index > 0) {
            finger = finger.next;
            index--;
        }
        // return the element stored in the node that the temporary pointer points to
        return finger.element;
    }
```

PRACTICE TIME: Insert element at head of doubly linked list

```
/**
 * Inserts the specified element at the head of the doubly linked list.
 *
 * @param element
 *            the element to be inserted
 */
public void addFirst(E element) {
    // Create a pointer to head

    // Make a new node and assign it to head. Fix pointers and update element



    // if first node to be added, adjust tail to it.

    // else fix previous pointer to head

    // increase number of nodes in doubly linked list.
}
```

Insert element at head of doubly linked list

```
/**
 * Inserts the specified element at the head of the doubly linked list.
 *
 * @param element
 *            the element to be inserted
 */
public void addFirst(E element) {
    // Create a pointer to head
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers and update element
    head = new Node();
    head.element = element;
    head.next = oldHead;
    head.prev = null;

    // if first node to be added, adjust tail to it.
    if (tail == null){
        tail = head;
     }
    else{
        // else fix previous pointer to head
        oldHead.prev = head;
     }
    // increase number of nodes in doubly linked list.
     size++;
}
```

PRACTICE TIME: Insert element at tail of doubly linked list

```
/**
 * Inserts the specified element at the tail of the doubly linked list.
 *
 * @param element
 *            the element to be inserted
 */
public void addLast(E element) {
    // Create a pointer to tail

    // Make a new node and assign it to tail. Fix pointers and update element


    // if first node to be added, adjust head to it.

    // else fix next pointer to tail

    // increase number of nodes in doubly linked list.
}
```

Insert element at tail of doubly linked list

```
/**
 * Inserts the specified element at the tail of the doubly linked list.
 *
 * @param element
 *            the element to be inserted
 */
public void addLast(E element) {
    // Create a pointer to tail
    Node oldTail = tail;

    // Make a new node and assign it to tail. Fix pointers and update element
    tail = new Node();
    tail.element = element;
    tail.next = null;
    tail.prev = oldTail;

    // if first node to be added, adjust head to it.
    if (head == null)
        head = tail;
    else{
         // else fix next pointer to tail
        oldTail.next = tail;
     }
    // increase number of nodes in doubly linked list.
    size++;
}
```

Insert element at tail of doubly linked list

```
/**
 * Inserts the specified element at the tail of the doubly linked list.
 *
 * @param element
 *            the element to be inserted
 */
public void add(E element) {
    // Create a pointer to tail
    addLast(element);

}
```

PRACTICE TIME: Insert element at a specified index

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index to insert the element
 * @param element
 *            the element to insert
 * @pre 0<=index<=size
 */
public void add(int index, E element) {
        // check whether index is valid

            // if index is 0, call addFirst

            // if index is size, call addLast

            // else
                 // Make two new Node references, previous and finger. Set previous to null and finger to head

             // search for index-th position. Set previous to finger and move finger to next position


             // create new Node, update its element, and fix its pointers taking into account where finger and previous
are

                  // increase number of nodes

    }
```

Insert element at a specified index

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index to insert the element
 * @param element
 *            the element to insert
 * @pre 0<=index<=size
 */
public void add(int index, E element) {
       // check whether index is valid
       if (index > size || index < 0){
           throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }
        // if index is 0, call addFirst
    if (index == 0) {
         addFirst(element);
    // if index is n, call addLast
} else if (index == size()) {
     addLast(element);
    // else
} else {
         // Make two new Node references, previous and finger. Set previous to null and finger to head
     Node previous = null;
     Node finger = head;
     // search for index-th position. Set previous to finger and move finger to next position
     while (index > 0) {
          previous = finger;
          finger = finger.next;
          index--;
     }
     // create new Node, update its element, and fix its pointers taking into account where finger and previous are
     Node current = new Node();
     current.element = element;
     current.next = finger;
     current.prev = previous;
     previous.next = current;
     finger.prev = current;
          // increase number of nodes
     size++;
}
```

Replace element at a specified index

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index of the element to replace
 * @param element
 *            the element to be stored at the specific index
 * @return the old element that was replaced
 * @pre 0<=index<size
 */
public E set(int index, E element) {
        // check that index is within range
       if (index >= size || index < 0){
            throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }

        Node finger = head;
        // search for index-th position by pointing previous to finger and advancing finger
        while (index > 0) {
             finger = finger.next;
             index--;
         }
         // reference old element
         E old = finger.element;
         // update element at finger
         finger.element = element;
         // return old element
         return old;
    }
}
```

PRACTICE: Retrieve and remove head

```
/**
 * Removes and returns the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public E removeFirst() {
    // Create a pointer to head

    // Move head to next

    // if there was only one node left in doubly linked list

        // remove tail by setting it to null

    // else

        // set previous pointer of head to null

    // set old head's next pointer to null

    // decrease number of nodes

    // return old head's element
}
```

Retrieve and remove head

```
/**
 * Removes and returns the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public E removeFirst() {
    // Create a pointer to head
    Node oldHead = head;
    // Move head to next
    head = head.next;
      // if there was only one node in the doubly linked list.
    if (head == null) {
        tail = null
    } else {
      head.prev= null;
    }
    // decrease number of nodes
    size--;
    // return old head's element
    return oldHead.element;
}
```

PRACTICE TIME: Retrieve and remove tail

```
/**
 * Removes and returns the tail of the doubly linked list.
 *
 * @return the tail of the doubly linked list.
 */
public E removeLast() {
    // Create a pointer to tail

    // Move tail to previous

    // if removed the last node

        // set head to null

    // else

        // set new tail's next to null
    }
    // decrease number of nodes

    // return old tail's element

}
```

Retrieve and remove tail

```
/**
 * Removes and returns the tail of the doubly linked list.
 *
 * @return the tail of the doubly linked list.
 */
public E removeLast() {
    // Create a pointer to tail
    Node temp = tail;
    // Move tail to previous
    tail = tail.prev;
    // if removed the last node
    if (tail == null) {
        // set head to null
        head = null;
    // else
    } else {
        // set new tail's next to null
        tail.next = null;
    }
    // decrease number of nodes
    size--;
    // return old tail's element
    return temp.element;
}
```

Retrieve and remove head

```
/**
 * Removes and returns the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public E remove() {
    return removeFirst();
}
```

PRACTICE TIME: Retrieve and remove element from a specific index

```
/**
 * Removes and returns the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the element previously at the specified index
 * @pre 0<=index<size
 */
public E remove(int index) {
      // check whether index is valid
      // if index is 0
            // return removeFirst
      // else if index is size-1
            // return removeLast
      // else
            // Make two new Node references, previous and finger. Set previous to null and finger to head
          // search for index-th position. Set previous to finger and move finger to next position


          // update pointers for previous and finger


          // decrease number of nodes
          // return the element that finger points to
     }
}
```

Retrieve and remove element from a specific index

```
/**
 * Removes and returns the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the element previously at the specified index
 * @pre 0<=index<size
 */
public E remove(int index) {
      // check whether index is valid
      if (index >= size || index < 0){
          throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
      }
      // if index is 0
     if (index == 0) {
            // return removeFirst
          return removeFirst();
      // else if index is size-1
     } else if (index == size - 1) {
            // return removeLast
          return removeLast();
      // else
     } else {
            // Make two new Node references, previous and finger. Set previous to null and finger to head
          Node previous = null;
          Node finger = head;
          // search for index-th position. Set previous to finger and move finger to next position
          while (index > 0) {
               previous = finger;
               finger = finger.next;
               index--;
          }
          // update pointers for previous and finger
          previous.next = finger.next;
          finger.next.prev = previous;
          // decrease number of nodes
          size--;
          // return the element that finger points to
          return finger.element;
     }

}
```

Clear the singly linked list of all elements

```
/**
 * Clears the doubly linked list of all elements.
 *
 */
public void clear(
        head = null;
        tail = null;
        size = 0;

}
```

addFirst() in doubly linked lists is $O(1)$ for worst case

```
public void addFirst(E element) {
        // Save the old node
        Node oldHead = head;

        // Make a new node and assign it to head. Fix pointers.
        head = new Node();
        head.element = element;
        head.next = oldHead;
    head.prev = null;

    // if first node to be added, adjust tail to it.
    if (tail == null)
        tail = head;
    else
        oldHead.prev = head;

    size++; // increase number of nodes in doubly linked list.
}
```

addLast() in doubly linked lists is $O(1)$ for worst case

```
public void addLast(E element) {
       // Save the old node
       Node oldTail = tail;

       // Make a new node and assign it to tail. Fix pointers.
       tail = new Node();
       tail.element = element;
       tail.next = null;
       tail.prev = oldTail;

       // if first node to be added, adjust head to it.
       if (head == null)
          head = tail;
       else
          oldTail.next = tail;

       size++;
}
```

get(int index) in doubly linked lists is $O(n)$ for worst case

```
/**
 * Returns element at the specified index.
 *
 * @param index
 *            the index of the element to be returned
 * @return the element at specified index
 * @pre 0<=index<size
 */
public E get(int index) {
    // check whether index is valid
     if (index >= size || index < 0){
          throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
      }
     // if index is 0, return element at head
    if (index == 0){
         return head.element;
     }
     // else if index is size-1, return element at tail
    else if (index == size - 1){
         return tail.element;
     }
    // set a temporary pointer to the head
    Node finger = head;
    // search for index-th element or end of list
    while (index > 0) {
         finger = finger.next;
         index--;
    }
    // return the element stored in the node that the temporary pointer points to
    return finger.element;
}
```

`add(int index, E element)` in doubly linked lists is $O(n)$ for worst case

```
public void add(int index, E element) {
        if (index == 0) {
            addFirst(element);
        } else if (index == size()) {
            addLast(element);
        } else {

            Node previous = null;
            Node finger = head;
            // search for index-th position
            while (index > 0) {
                previous = finger;
                finger = finger.next;
                index--;
            }
            // create new value to insert in correct position
            Node current = new Node();
            current.element = element;
            current.next = finger;
            current.prev = previous;
            previous.next = current;
            finger.prev = current;

            size++;
        }
    }
```

`set(int index, E element)` in singly linked lists is $O(n)$ for worst case

```
    /**
     * Inserts the specified element at the specified index.
     *
     * @param index
     *            the index of the element to replace
     * @param element
     *            the element to be stored at the specific index
     * @return the old element that was replaced
     * @pre 0<=index<size
     */
    public E set(int index, E element) {
        // check that index is within range
        if (index >= size || index < 0){
            throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }

        Node finger = head;
        // search for index-th position by pointing previous to finger and advancing finger
        while (index > 0) {
            finger = finger.next;
            index--;
        }
        // reference old element
        E old = finger.element;
        // update element at finger
        finger.element = element;
        // return old element
        return old;
    }
}
```

`removeFirst()` in doubly linked lists is $O(1)$ for worst case

```
public E removeFirst() {
        Node oldHead = head;
        // Fix pointers.
        head = head.next;
        // if there was only one node in the doubly linked list.
        if (head == null) {
            tail = null
        } else {
            head.next= null;
        }

        size--;

        return oldHead.element;
    }
```

removeLast() in doubly linked lists is $O(1)$ for worst case

```
public E removeLast() {

    Node temp = tail;
    tail = tail.prev;

    // if there was only one node in the doubly linked list.
    if (tail == null) {
        head = null;
    } else {
        tail.next = null;
    }
    size--;
    return temp.element;
}
```

`remove(int index)` in doubly linked lists is $O(n)$ for worst case

```
public E remove(int index) {
    if (index == 0) {
        return removeFirst();
    } else if (index == size() - 1) {
        return removeLast();
    } else {
        Node previous = null;
        Node finger = head;
        // search for value indexed, keep track of previous
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        previous.next = finger.next;
        finger.next.prev = previous;

        size--;
        // finger's value is old value, return it
        return finger.element;
    }

}
```

Lecture 10: Doubly Linked Lists

- Doubly Linked Lists
- Java Collections

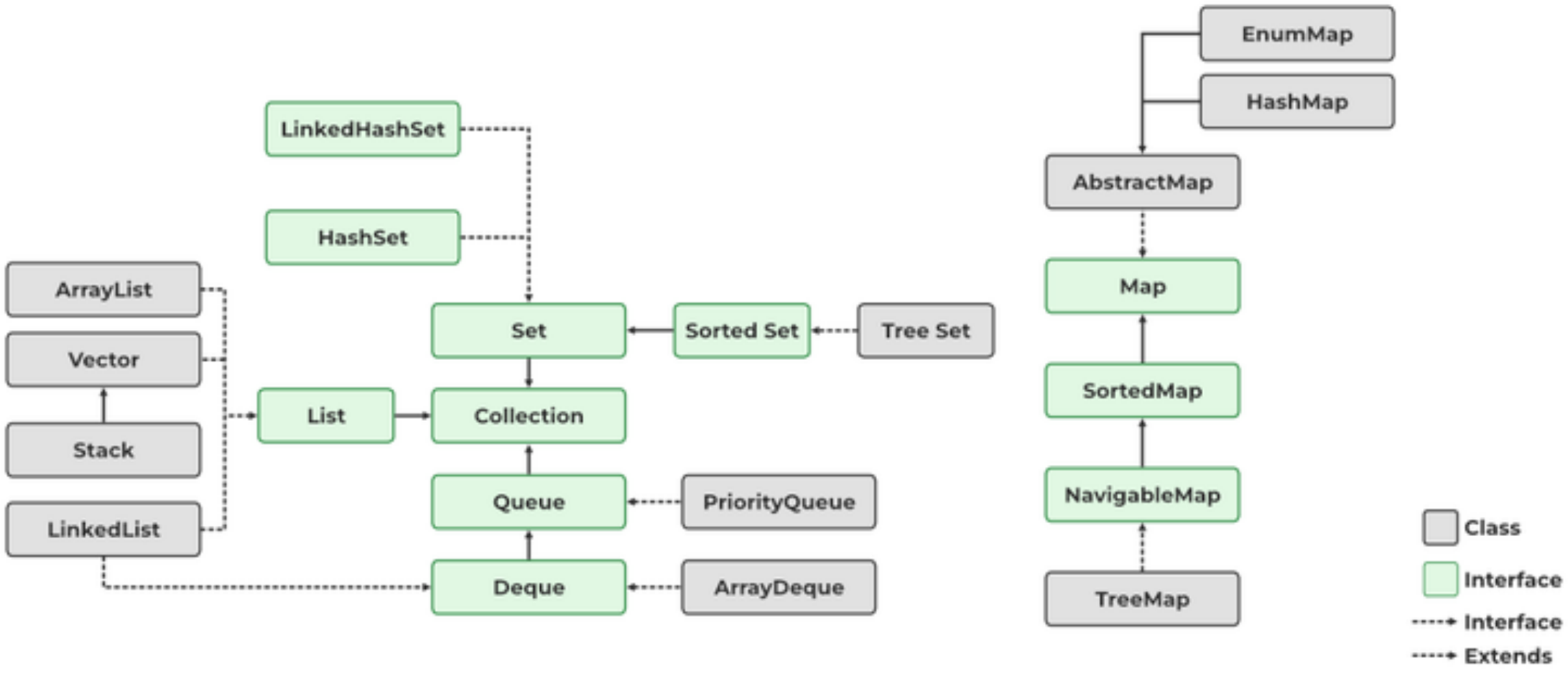
`clear()` in singly linked lists is $O(1)$ for worst case

```
/**
 * Clears the doubly linked list of all elements.
 *
 */
public void clear(
        head = null;
        tail = null;
        size = 0;
}
```

Lecture 10: Doubly Linked Lists

- Doubly Linked Lists
- Java Collections

The Java Collections Framework



https://www.geeksforgeeks.org/collections-in-java-2/

`LinkedList` in Java Collections

- Doubly linked list implementation of the `List` and `Deque` (stay tuned) interfaces.

`java.util.LinkedList;`

```
public class LinkedList<E> extends
AbstractSequentialList<E> implements List<E>, Deque<E>
```

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Lecture 10: Doubly Linked Lists

- Doubly Linked Lists
- Java Collections

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Linked Lists: https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
- Recommended Textbook:
  - Chapter 1.3 (Page 142–146)
- Recommended Textbook Website:
  - Linked Lists: https://algs4.cs.princeton.edu/13stacks/

Code

- Lecture 10 code

Practice Problems:

- 1.3.18–1.3.27 (approach them as doubly linked lists).