BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

8: Doubly Linked Lists

Alexandra Papoutsaki
she/her/hers

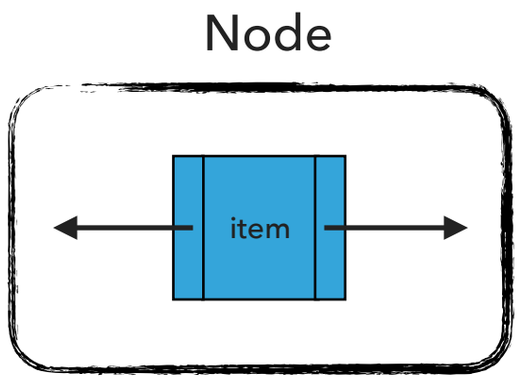
Lecture 8: Doubly Linked Lists

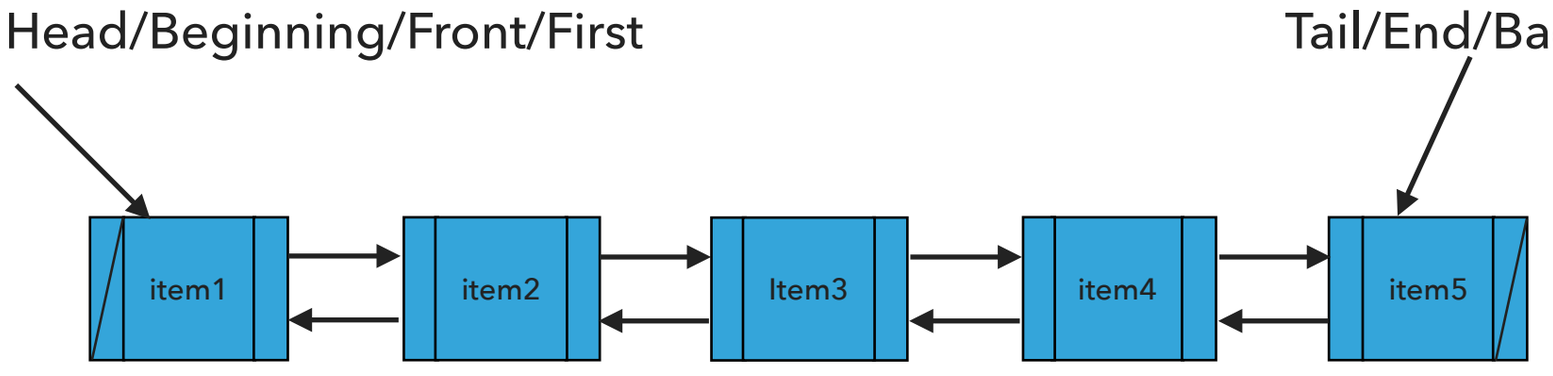
- Doubly Linked Lists
- Java Collections

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

Recursive Definition of Doubly Linked Lists

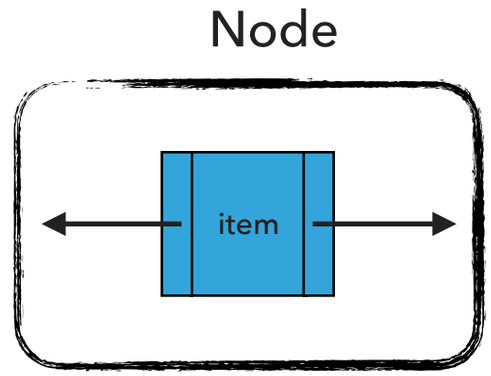
- A doubly linked list is either empty (null) or a node having a reference to a doubly linked list.
- Node: is a data type that holds any kind of data and two references to the previous and next node.





Node

```
private class Node {
    Item item;
    Node next;
    Node prev;
}
```



Standard Operations

- `DoublyLinkedList()`: Constructs an empty doubly linked list.
- `isEmpty()`:Returns true if the doubly linked list does not contain any item.
- `size()`: Returns the number of items in the doubly linked list.
- `get(int index)`: Returns the item at the specified index.
- `addFirst(Item item)`: Inserts the specified item at the head of the doubly linked list.
- `addLast(Item item)`: Inserts the specified item at the tail of the doubly linked list.
- `add(int index, Item item)`: Inserts the specified item at the specified index.
- `Item removeFirst()`: Retrieves and removes the head of the doubly linked list.
- `Item removeLast()`: Retrieves and removes the tail of the doubly linked list.
- `Item remove(int index)`: Retrieves and removes the item at the specified index.

`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

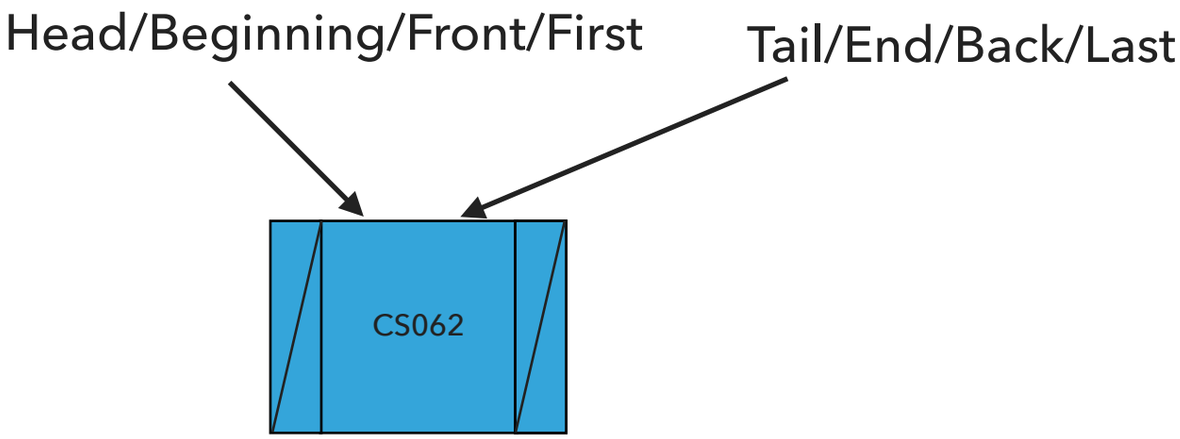
tail = null

size = 0

What should happen?

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

`addFirst(Item item)`: Inserts the specified item at the head of the doubly linked list

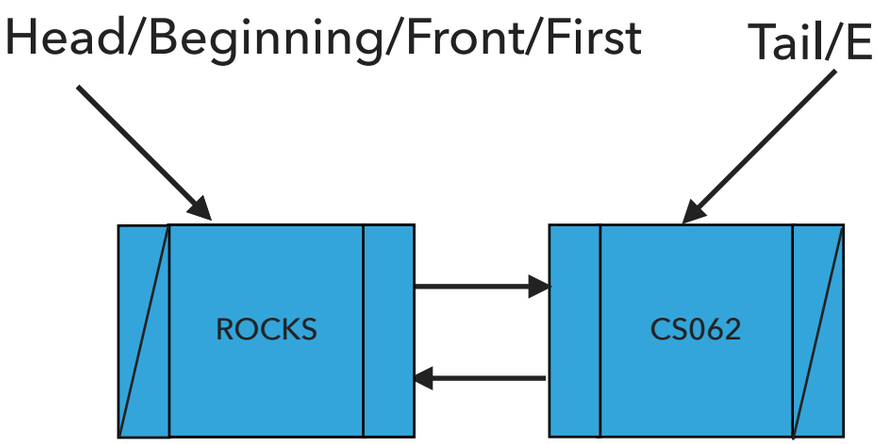



```
dll.addFirst(“CS062”)

size=1
```

What should happen?

```
dll.addFirst(“ROCKS”);
```

`addFirst(Item item)`: Inserts the specified item at the head of the doubly linked list

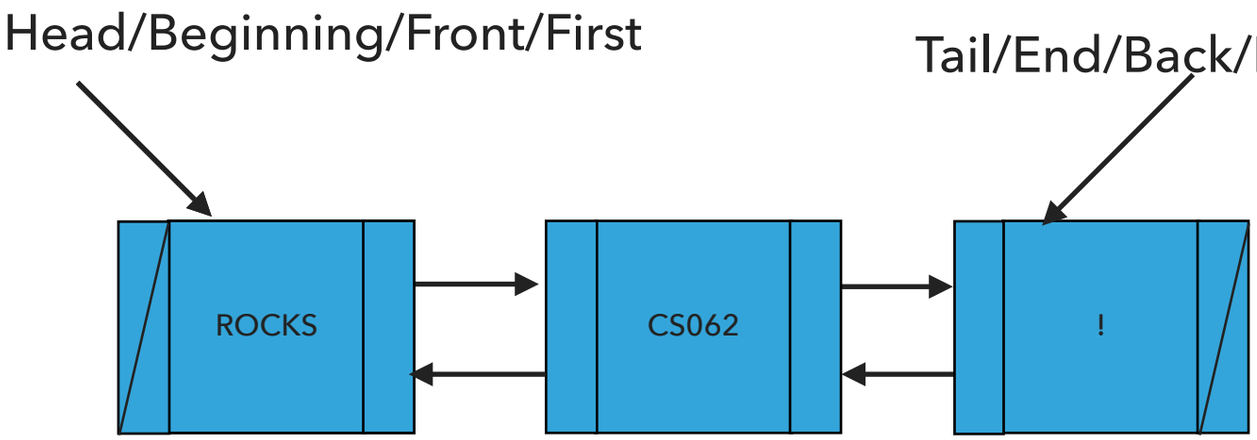



```
dll.addFirst(“ROCKS”)

size=2
```

What should happen?

```
dll.addLast(“!”);
```

`addLast(Item item)`: Inserts the specified item at the tail of the doubly linked list

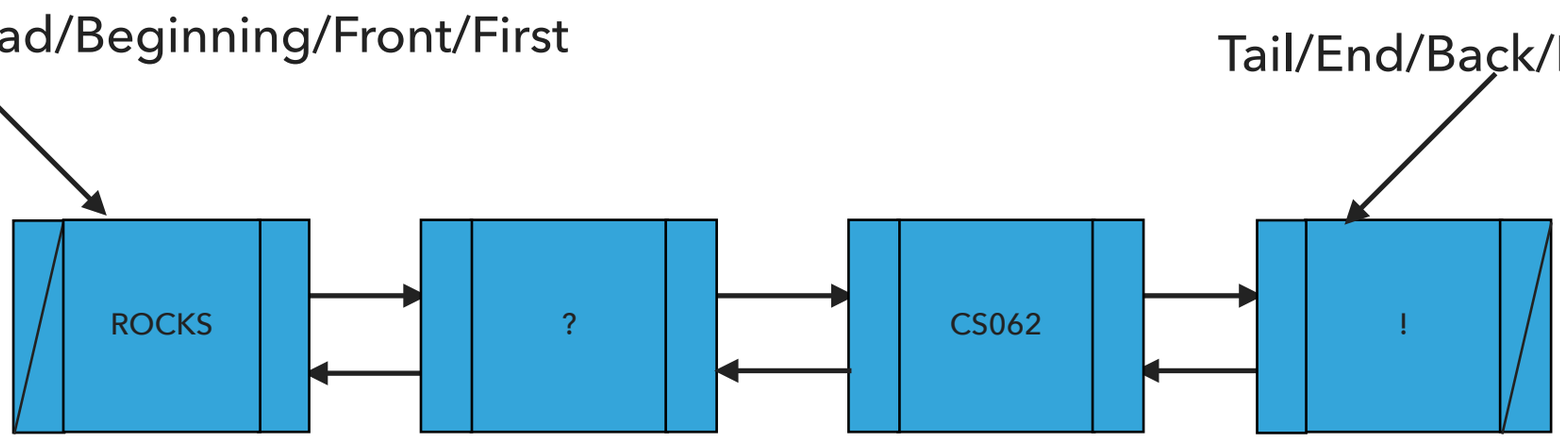



```
dll.addLast(“!”)

size=3
```

What should happen?

```
dll.add(1,“?”);
```

`add(int index, Item item)`: Adds item at the specified index



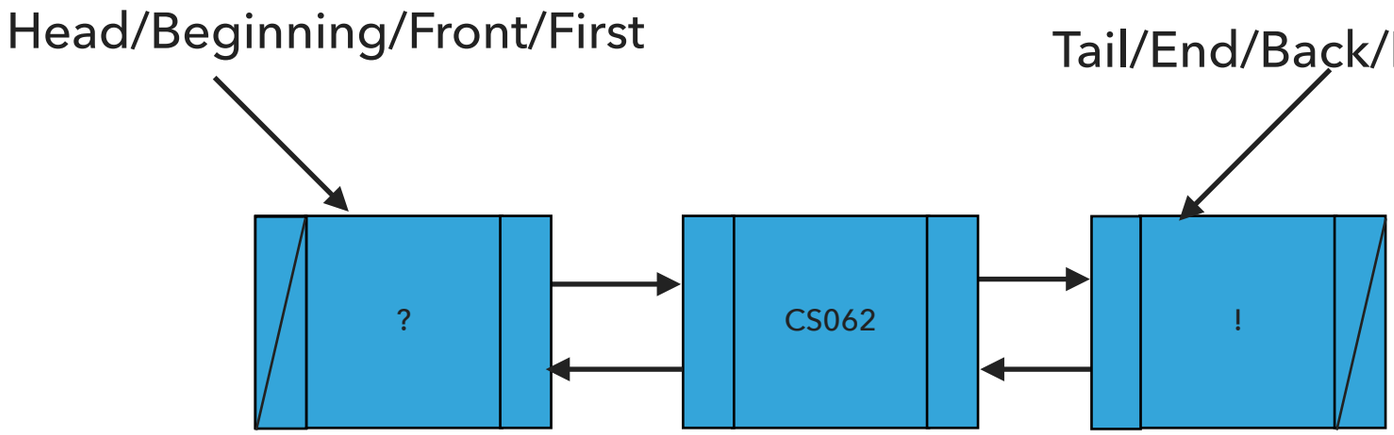
`dll.add(1,“?”)`

`size=4`

What should happen?

```
dll.removeFirst();
```

`removeFirst()`: Retrieves and removes the head of the doubly linked list

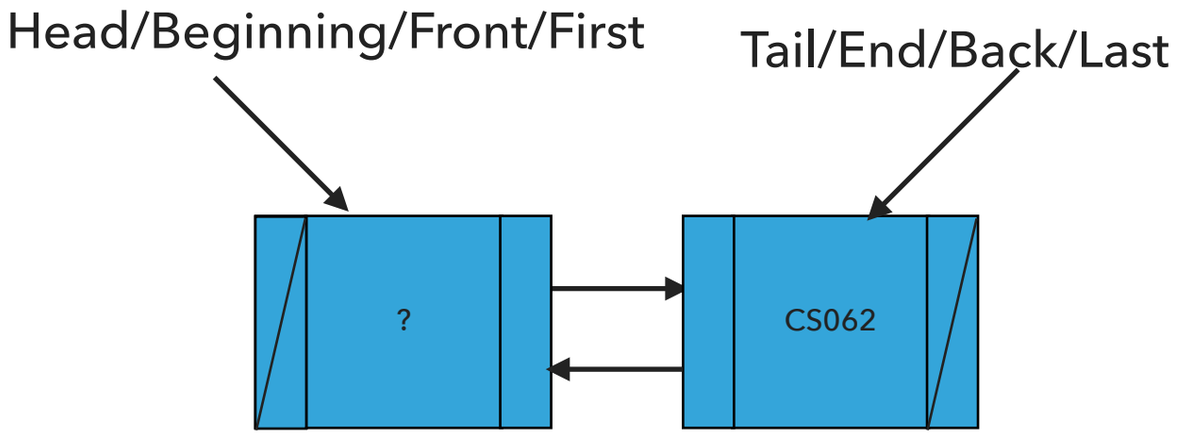



```
dll.removeFirst()

size=3
```

What should happen?

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

`removeLast()`: Retrieves and removes the tail of the doubly linked list

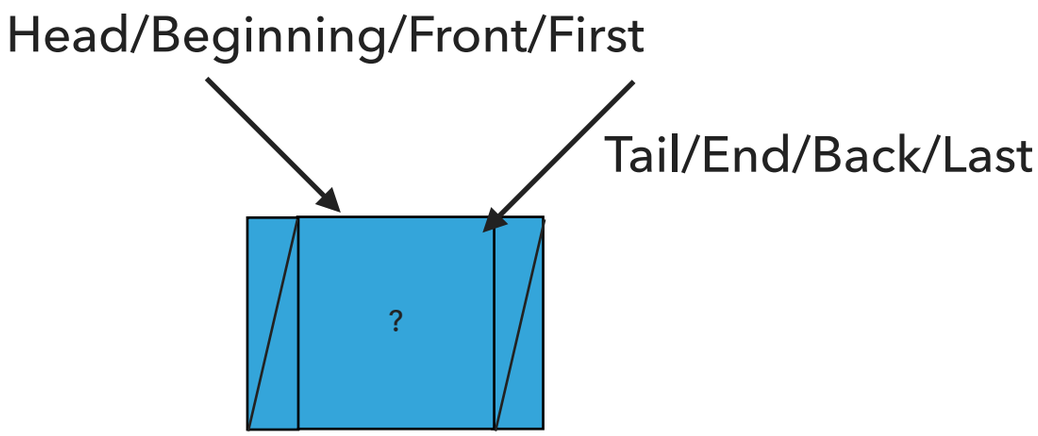



```
dll.removeLast()

size=2
```

What should happen?

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

`remove(int index)`: Retrieves and removes the item at the specified index




```
dll.remove(1)

size=1
```

Our own implementation of Doubly Linked Lists

- We will follow the textbook style.
  - It does not offer a class for this so we will build our own.
- We will work with generics because we don't want to offer multiple implementations.
- 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<Item> implements Iterable<Item> {
    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 {
        Item item;
        Node next;
        Node prev;
    }
```

PRACTICE TIME: Check if is empty and how many items

```
/**
 * Returns true if the doubly linked list does not contain any item.
 *
 * @return true if the doubly linked list does not contain any item
 */
public boolean isEmpty() {

}
```

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

}
```

Check if is empty and how many items

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

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

PRACTICE TIME: Retrieve item from specified index

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

    // if index is 0, return item at head


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


    // set a temporary pointer to the head

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


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

Retrieve item from specified index

```
/**
 * Returns item at the specified index.
 *
 * @param index
 *            the index of the item to be returned
 * @return the item at specified index
 * @pre 0<=index<size
 */
public Item 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 item at head
    if (index == 0){
         return head.item;
     }
     // else if index is size-1, return item at tail
    else if (index == size - 1){
         return tail.item;
     }
    // 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 item stored in the node that the temporary pointer points to
    return finger.item;
}
```

PRACTICE TIME: Insert item at head of doubly linked list

```
/**
 * Inserts the specified item at the head of the doubly linked list.
 *
 * @param item
 *            the item to be inserted
 */
public void addFirst(Item item) {
    // Create a pointer to head
    // Make a new node and assign it to head. Fix pointers and update item


    // 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 item at head of doubly linked list

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

    // Make a new node and assign it to head. Fix pointers and update item
    head = new Node();
    head.item = item;
    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 item at tail of doubly linked list

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

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


    // 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 item at tail of doubly linked list

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

    // Make a new node and assign it to tail. Fix pointers and update item
    tail = new Node();
    tail.item = item;
    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++;
}
```

PRACTICE TIME: Insert item at a specified index

```
/**
 * Inserts the specified item at the specified index.
 *
 * @param index
 *            the index to insert the item
 * @param item
 *            the item to insert
 * @pre 0<=index<=size
 */
public void add(int index, Item item) {
        // 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 item, and fix its pointers taking into account where finger and previous are

              // increase number of nodes
}
```

Insert item at a specified index

```
/**
 * Inserts the specified item at the specified index.
 *
 * @param index
 *            the index to insert the item
 * @param item
 *            the item to insert
 * @pre 0<=index<=size
 */
public void add(int index, Item item) {
       // 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(item);
    // if index is n, call addLast
} else if (index == size()) {
     addLast(item);
    // 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 item, and fix its pointers taking into account where finger and previous are
     Node current = new Node();
     current.item = item;
     current.next = finger;
     current.prev = previous;
     previous.next = current;
     finger.prev = current;
          // increase number of nodes
     size++;
}
```

PRACTICE: Retrieve and remove head

```
/**
 * Retrieves and removes the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public Item 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 item
}
```

Retrieve and remove head

```
/**
 * Retrieves and removes the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public Item 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;
    }
    // set old head's next pointer to null
    oldHead.next = null;
    // decrease number of nodes
    size--;
    // return old head's item
    return oldHead.item;
}
```

PRACTICE TIME: Retrieve and remove tail

```
/**
 * Retrieves and removes the tail of the doubly linked list.
 *
 * @return the tail of the doubly linked list.
 */
public Item 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 item

}
```

Retrieve and remove tail

```
/**
 * Retrieves and removes the tail of the doubly linked list.
 *
 * @return the tail of the doubly linked list.
 */
public Item 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 item
    return temp.item;
}
```

PRACTICE TIME: Retrieve and remove element from a specific index

```
/**
 * Retrieves and removes the item at the specified index.
 *
 * @param index
 *            the index of the item to be removed
 * @return the item previously at the specified index
 * @pre 0<=index<size
 */
public Item 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 item that finger points to

      }

}
```

Retrieve and remove element from a specific index

```
/**
 * Retrieves and removes the item at the specified index.
 *
 * @param index
 *            the index of the item to be removed
 * @return the item previously at the specified index
 * @pre 0<=index<size
 */
public Item 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 item that finger points to
          return finger.item;
     }

}
```

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

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

        // Make a new node and assign it to head. Fix pointers.
        head = new Node();
        head.item = item;
        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(Item item) {
       // Save the old node
       Node oldTail = tail;

       // Make a new node and assign it to tail. Fix pointers.
       tail = new Node();
       tail.item = item;
       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++;
}
```

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

```
public void add(int index, Item item) {
        if (index == 0) {
            addFirst(item);
        } else if (index == size()) {
            addLast(item);
        } 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.item = item;
            current.next = finger;
            current.prev = previous;
            previous.next = current;
            finger.prev = current;

            size++;
    }
}
```

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

```
public Item 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;
       }
       oldHead.next = null;

       size--;

       return oldHead.item;
}
```

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

```
public Item 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.item;
}
```

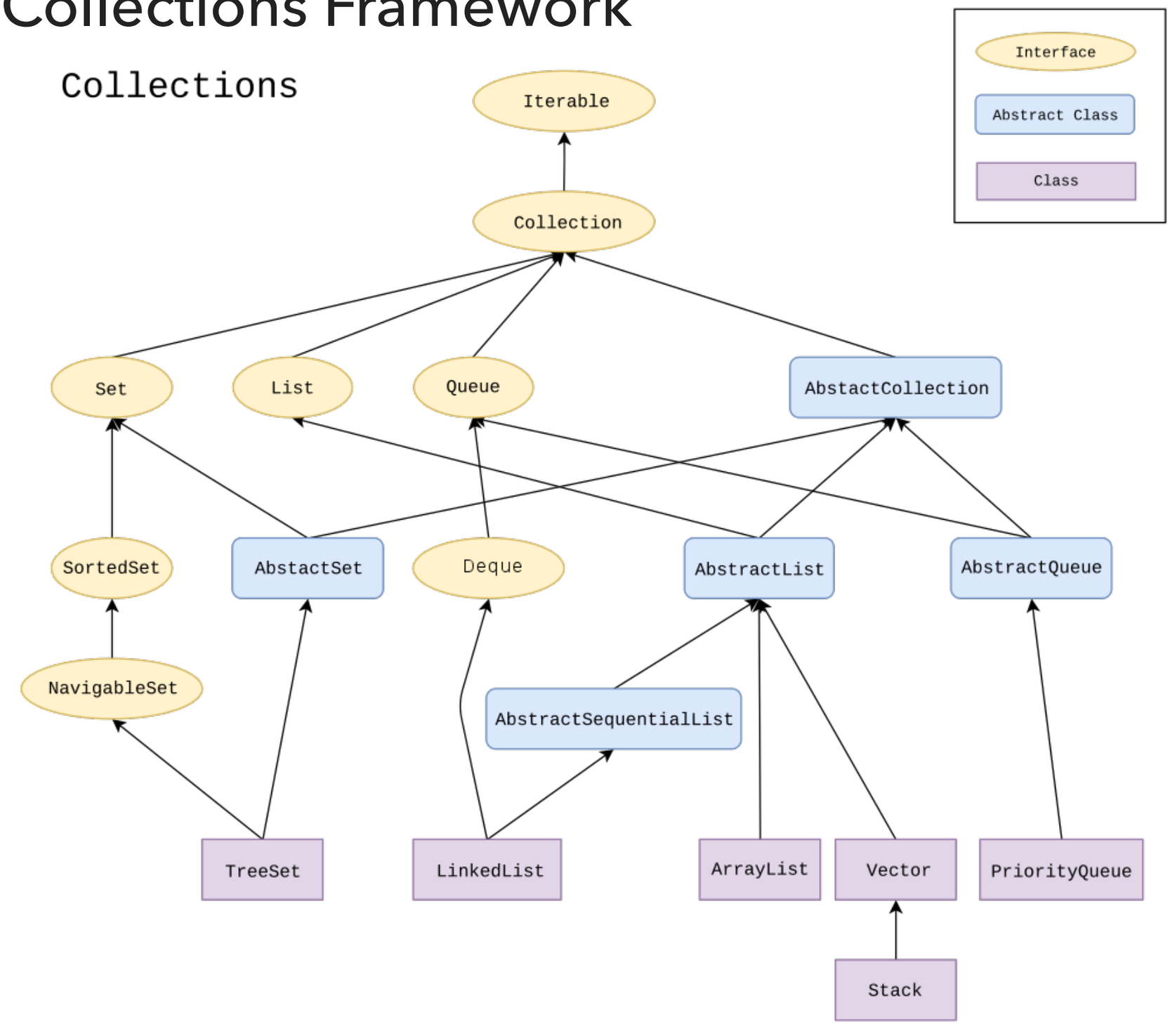
`remove(int index)` in doubly linked lists is $O(n)$ for worst case

```
public Item 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.item;
    }

}
```

Lecture 8: Doubly Linked Lists

- Doubly Linked Lists
- Java Collections

The Java Collections Framework



https://en.wikipedia.org/wiki/Java_collections_framework

`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 8: 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 8 code

Practice Problems:

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