About This Lecture

- In this lecture we will learn about an implementation of the List Interface called Doubly-Linked List.

Outline

- Drawing Doubly Linked Lists
- DoublyLinkedListElement class
- DoublyLinkedList class

Doubly-Linked List Diagrams

- A doubly-linked list node has two links, one forward and one backward.
- The doubly-linked list has references to its head and tail nodes.
- This symmetry makes the implementation simpler.

Constructing a Node

- When a DoublyLinkedListElement (node) is constructed, four links may need to be set.
- If one or both of the “neighbouring” nodes is null then fewer links must be set.

```java
public class DoublyLinkedListElement {
    protected Object data;
    protected DoublyLinkedListElement next;
    protected DoublyLinkedListElement previous;
    public DoublyLinkedListElement(Object element, DoublyLinkedListElement nextNode, DoublyLinkedListElement previousNode) {
        this.data = element;
        this.next = nextNode;
        this.previous = previousNode;
        if (nextNode != null) nextNode.previous = this;
        if (previousNode != null) previousNode.next = this;
    }
}
```
DoublyLinkedListElement – slide 2

```java
public DoublyLinkedListElement(Object element) {
    // post: initializes the receiver node to be the only node
    // in a list.
    this(element, null, null);
}
```

DoublyLinkedListElement – slide 3

- The implementation of the other methods in
  DoublyLinkedListElement are straightforward:
  - `next()` - return the next node.
  - `previous()` - return the previous node.
  - `setNext(DoublyLinkedListElement)` - set the next node to the
    given node.
  - `setPrevious(DoublyLinkedListElement)` - set the previous
    node to the given node.
  - `getValue(Object)` - set the element in the receiver node to the
    given object.
  - `getValue()` - return the element contained in the receiver node.

DoublyLinkedList - State and Constructor

```java
public class DoublyLinkedList implements List {
    protected int count;
    protected DoublyLinkedListElement head;
    protected DoublyLinkedListElement tail;

    public DoublyLinkedList() {
        // post: initializes the list to be empty.
        this.head = null;
        this.tail = null;
        this.count = 0;
    }
```

DoublyLinkedList - Store Interface

```java
/* Interface Store Methods */
public int size() {
    // post: returns the number of elements in the list.
    return this.count;
}

public boolean isEmpty() {
    // post: returns the true iff store is empty.
    return this.size() == 0;
}

public void clear() {
    // post: clears the list so that it contains no elements.
    this.head = null;
    this.tail = null;
    this.count = 0;
}
```

DoublyLinkedList - Collection Interface 1

```java
/* Interface Collection Methods */
public boolean contains(Object anObject) {
    // pre: anObject is non-null
    // post: returns true iff the collection contains the object
    DoublyLinkedListElement cursor;
    cursor = this.head; // partial traversal
    while ((cursor != null) && (!cursor.value().equals(anObject))
        cursor = cursor.next();
    return cursor != null;
}
```

DoublyLinkedList Collection Interface 2

```java
public void add(Object anObject) {
    // pre: anObject is non-null
    // post: the object is added to the head of the
    // of the collection.
    this.addToHead(anObject);
}
```

```java
public Object remove(Object anObject) {
    // pre: anObject is non-null
    // post: removes object "equal" to anObject and returns
    // it, otherwise returns null.
    // left as an exercise - try to adapt the
    // remove method from SinglyLinkedList
}
```

```java
public Iterator elements() {
    // post: return an iterator for traversing the collection
    // Ignore this one until iterators lecture
}
```
DoublyLinkedList - peek() and tailPeek()

```java
public Object peek() {
    // pre: list is not empty
    // post: returns the first object in the list without
    // modifying the list
    return this.head.value();
}
public Object tailPeek() {
    // pre: list is not empty
    // post: returns the last object in the list without
    // modifying the list
    return this.tail.value();
}
```

DoublyLinkedList - addToHead() vers 1

```java
/** Interface List Methods */
public void addToHead(Object anObject) {
    // pre: anObject is non-null
    // post: the object is added to the beginning of the list
    this.head = new DoublyLinkedListElement(anObject,
        this.head, null);
    this.count++;
}
```

DoublyLinkedList - addToHead() problem

```java
/** Interface List Methods */
public void addToHead(Object anObject) {
    // pre: anObject is non-null
    // post: the object is added to the beginning of the list
    this.head = new DoublyLinkedListElement(anObject,
        this.head, null);
    this.count++;
}
```

DoublyLinkedList - removeFromHead() vers 1

```java
public Object removeFromHead() {
    // pre: list is not empty
    // post: removes and returns first object from the list
    DoublyLinkedListElement temp;
    temp = this.head;
    this.head = this.head.next();
    if (this.head != null) {
        this.head.setPrevious(null);
        this.count--;
    }
    return temp.value();
}
```

DoublyLinkedList - removeFromHead() problem

```java
public Object removeFromHead() {
    // pre: list is not empty
    // post: removes and returns first object from the list
    DoublyLinkedListElement temp;
    temp = this.head;
    this.head = this.head.next();
    if (this.head != null) {
        this.head.setPrevious(null);
        this.count--;
    }
    return temp.value();
}
```
DoublyLinkedList - removeFromHead()

```java
public Object removeFromHead() {
    // pre: list is not empty
    // post: removes and returns first object from the list
    DoublyLinkedListElement temp;
    temp = this.head;
    this.head = this.head.next();
    if (this.head != null)
        this.head.setPrevious(null);
    else
        this.tail = null;
    this.count--;
    return temp.value();
}
```

DoublyLinkedList - addToTail()

```java
public void addToTail(Object anObject) {
    // pre: anObject is non-null
    // post: the object is added at the end of the list
    if (this.head == null)
        this.head = this.tail = new DoublyLinkedListElement(anObject, null, this.tail);
    else
        this.tail = this.tail.next(anObject);
}
```

DoublyLinkedList - removeFromTail()

```java
public Object removeFromTail() {
    // pre: list is not empty
    // post: the last object in the list is removed and returned
    DoublyLinkedListElement temp;
    temp = this.tail;
    this.tail = this.tail.previous();
    if (this.tail == null)
        this.head = null;
    else
        this.tail.setNext(null);
    this.count--;
    return temp.value();
}
```