About This Lecture

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

Outline

- Drawing Lists and List Nodes
- SinglyLinkedListElement class
- SinglyLinkedList class

List Interface

```java
public interface List extends Collection{
    public void addToHead(Object anObject);
    public void addToTail(Object anObject);
    public Object peek();
    public Object tailPeek();
    public Object removeFromHead();
    public Object removeFromTail();
}
```

External Container Diagrams

- We can draw an external or implementation-independent diagram of a container by just showing its elements.
- For example, here is the diagram for a general container where the elements are not even ordered:

Indexed Container Diagrams

- We can modify the diagram when the interface is more specific.
- For example, here is a diagram for an indexed container.
- Note that it still might be implemented as a vector or as an array.
External List Diagrams

- Since a List is not indexed, the elements are not numbered.
- However, since the elements are ordered, they must be "connected" somehow to maintain this order.
- Since different implementation classes "connect" the elements differently, we do not show the connections in an external diagram.

How to implement a list

- **Linked list**
  ```plaintext```
  ![Linked list diagram]
  ```plaintext```

  - **Parallel Arrays**
    ```plaintext```
    ```plaintext```
    ```plaintext```
  ```plaintext```

Internal List Diagrams - Nodes

- However, when we want to highlight a particular class implementation of the List interface, we add internal structure.
- Each element is put in a list node and the nodes connect to each other with links.
- The end of the list is denoted by a dot instead of a link to another node.

Complex List Node Diagrams

- If the elements of a list are complex objects, it is not always possible to draw the elements inside the node.
- In this case, an arrow is used in the node to represent a reference to the element.
- This diagram is actually more accurate since the node always contains a reference to an object instead of an object.

Terminology - Elements and Nodes

- In these notes we use the term element to refer to the individual values or objects that collectively make up the list.
- We use the term node to refer to an object that contains an element object and links to adjacent list nodes.

Singly-Linked Lists

- In a singly-linked list, each list node contains an element and a link to the "next" node in the list.
- Since a node contains a link to the next node, this is an example of self-referencing of objects.
- We need to define two classes to implement a singly-linked list, one for the nodes and one for the list itself.
- In the structure package, the list node class is called SinglyLinkedListElement and the list class is called SinglyLinkedList.
SinglyLinkedListElement Class

- Each instance of SinglyLinkedListElement represents a single list node with two instance variables.
- The instance variable, data, is a reference to an element object.
- The instance variable, next, is a reference to the next node (another instance of SinglyLinkedListElement) or null if it is the last node (tail node).

```
public class SinglyLinkedListElement {
    protected Object data;
    protected SinglyLinkedListElement next;
    public SinglyLinkedListElement(Object element, SinglyLinkedListElement nextNode) {
        // post: initializes the node to contain the given object and link to the given next node.
        this.data = element;
        this.next = nextNode;
    }
    public SinglyLinkedListElement(Object element) {
        // post: initializes the node to be a tail node containing the given value.
        this(element, null);
    }
}
```

```
public SinglyLinkedListElement next() {
    // post: returns the next node.
    return this.next;
}
public void setValue(Object anObject) {
    // post: sets the element in this node to the given object
    this.data = anObject;
}
```

SinglyLinkedList Class

- How many instance variables?
  1. One SinglyLinkedListElement pointing to the head
  2. One integer to store the number of elements
  3. One Element pointing to the tail?

- How many constructors?
  1. Constructor for the empty list

- How many access methods?
  All the signatures specified in the Store, Collection and List interfaces
Caching the List Size

- The list size could be computed by traversing the list and counting nodes.
- However, the size is cached as an instance variable so that the size() method can be computed faster.
- The disadvantage of caching the size as an instance variable is that the instance variable must be updated each time an element is added or removed from the list.

List Traversal

- Many list methods will involve traversal of the list elements from the head to the tail or from the head to a particular element or location.
- We use a cursor for traversal.

SinglyLinkedList - State and Constructor

```java
public class SinglyLinkedList implements List {
    protected int count;
    protected SinglyLinkedListElement head;

    public SinglyLinkedList() {
        this.head = null;
        this.count = 0;
    }

    // If we call clear instead of setting the instance
    // variables directly, we could change instance variables
    // without changing both the clear method and this method.
}
```

SinglyLinkedList - 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
    // call to size allows us to change instance variables
    public void clear() {
        // post: clears the list so that it contains no elements.
        this.head = null;
        this.count = 0;
    }
```

SinglyLinkedList - Collection Interface 1

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

SinglyLinkedList Collection Interface 2

```java
public void add(Object anObject) {
    // pre: anObject is non-null
    // post: the object is added to the collection. The
    // replacement policy is: add the new object at the front.
    this.addToList(anObject);
}

public Object remove(Object anObject) {
    // pre: anObject is non-null
    // post: remove object "equal" to anObject and return it,
    // otherwise returns null
}

public Iterator elements() {
    // post: return an iterator for traversing the collection
    // Ignore this one until Iterators Lecture
}
SinglyLinkedList - peek() and tailPeek()

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
  // left as an exercise
}

SinglyLinkedList - removeFromHead()

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

SinglyLinkedList - addToHead()

/* 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 SinglyLinkedListElement(anObject, this.head);
  this.count++;
  aList.addToHead("Pebbles");
}

"Fred" -> "Wilma" -> "Barney"

SinglyLinkedList - addToTail() version 1

public void addToTail(Object anObject) {
  // pre: anObject is non-null
  // post: the object is added at the end of the list
  SinglyLinkedListElement temp;
  SinglyLinkedListElement cursor;
  temp = new SinglyLinkedListElement(anObject, null);
  cursor = this.head;
  while (cursor.next() != null) {
    cursor = cursor.next();
  }
  cursor.setNext(temp);
  this.count++;
}

SinglyLinkedList - addToTail() problem

public void addToTail(Object anObject) {
  // pre: anObject is non-null
  // post: the object is added at the end of the list
  SinglyLinkedListElement temp;
  SinglyLinkedListElement cursor;
  temp = new SinglyLinkedListElement(anObject, null);
  if (this.head == null) {
    this.head = temp;
    this.head = new SinglyLinkedListElement(anObject, null);
  } else {
    cursor = this.head;
    while (cursor.next() != null) {
      cursor = cursor.next();
    }
    cursor.setNext(temp);
  }
  this.count++;
}

check boundaries
null

In this case we just want to bind this.head to the new node
Removal From Tail - Problem

- We cannot remove from the tail by traversing to the last node and removing it.
- We need a reference to the second last node so we can set its "next" reference to null.
- To find the second last node, we need to traverse the list with a second cursor following the first.

Removal From Tail - Solution

SinglyLinkedList - removeFromTail() v1 p1

```java
public Object removeFromTail() {
    // pre: list is not empty
    // post: the last object in the list is removed and returned
    SinglyLinkedListElement previous;
    // continues on next slide...
```

SinglyLinkedList - removeFromTail() v1 p2

```java
public Object removeFromTail() {
    // pre: list is not empty
    // post: the last object in the list is removed and returned
    SinglyLinkedListElement cursor;
    SinglyLinkedListElement previous;
    cursor = this.head;
    while (cursor.next() != null) {
        previous = cursor;
        cursor = cursor.next();
    }
    previous.setNext(null);
    this.count--;
    return cursor.value();
```

SinglyLinkedList - removeFromTail() problem

```java
    cursor = this.head;
    while (cursor.next() != null) {
        previous = cursor;
        cursor = cursor.next();
    }
    previous.setNext(null);
    this.count--;
    return cursor.value();
```

SinglyLinkedList - removeFromTail()
SinglyLinkedList - remove

```java
public Object remove(Object anObject) {
    // pre: anObject is non-null
    // post: remove object "equal" to anObject and return it,
    // otherwise returns null

    // As in removeFromTail, run two pointers through
    // the list, but stop when cursor is pointing at a
    // node with an element equal to anObject.
    previous.setNext(_________);

    Special Cases ???
```

Linked-List Implementation Advice

- When manipulating references, draw pictures.
- Every public method of an object should leave
  the object in a consistent state.
- Symmetry is good.
- Test the boundaries of your structures and
  methods.