CIS2168 RECURSION

Chapter 5 Part 3

Key Topics

- Recursive data structures and recursive methods for a `LinkedList` class

Recursive Data Structures

Section 5.4

Recursive Data Structures

- Computer scientists often encounter data structures that are defined recursively
  - Each with another version of itself as a component
- Linked lists and trees (Chapter 6) can be defined as recursive data structures

Class `LinkedListRec`

- We define a class `LinkedListRec<E>` that implements several list operations using recursive methods

```
public class LinkedListRec<E> {
   private Node<E> head;

   // inner class Node<E> here
   // (from chapter 2)
}
```

Recursive `size` Method

```
/** Finds the size of a list.
    @param head The head of the current list
    @return The size of the current list
*/
private int size(Node<E> head) {
    if (head == null)
        return 0;
    else
        return 1 + size(head.next);
}

/** Wrapper method for finding the size of a list.
    @return The size of the list
*/
public int size() {
    return size(head);
}
```

Recursive `toString` Method

```
/** Returns the string representation of a list.
    @param head The head of the current list
    @return The state of the current list
*/
private String toString(Node<E> head) {
    if (head == null)
        return "";
    else
        return head.data + "\n" + toString(head.next);
}

/** Wrapper method for returning the string representation of a list.
    @return The string representation of the list
*/
public String toString() {
    return toString(head);
}
```

Recursive `replace` Method

```
/** Replaces all occurrences of oldObj with newObj.
    post: Each occurrence of oldObj has been replaced by newObj.
    @param head The head of the current list
    @param oldObj The object being removed
    @param newObj The object being inserted
*/
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldObj.equals(head.data))
            head.data = newObj;
        replace(head.next, oldObj, newObj);
    }
}

/*  Wrapper method for replacing oldObj with newObj.
    post: Each occurrence of oldObj has been replaced by newObj.
    @param oldObj The object being removed
    @param newObj The object being inserted
*/
public void replace(E oldObj, E newObj) {
    replace(head, oldObj, newObj);
}
```

Note: the book code here replaces ALL occurrences of the oldObj.

Recursive `replace` Method

Note: The code below replaces only the first occurrence of the oldObj.

```
/** Replaces the FIRST occurrence of oldObj with newObj.
   @post FIRST occurrence of oldObj has been replaced by newObj.
   @param head The head of the current list
   @param oldObj The object being replaced
   @param newObj The object being inserted
 */
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldObj.equals(head.data)) {
            head.data = newObj;
            return;
        }
        replace(head.next, oldObj, newObj);
    }
}
```

Recursive `add` Method

```
/** Adds a new node to the end of a list.
    @param head The head of the current list
    @param data The data for the new node
*/
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<E>(data);
    else
        add(head.next, data);      // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list.
    @param data The data for the new node
*/
public void add(E data) {
    if (head == null)
        head = new Node<E>(data);  // List has 1 node.
    else
        add(head, data);
}
```

Recursive `remove` Method

```
/** Removes a node from a list.
    post: The first occurrence of outData is removed.
    @param head The head of the current list
    @param pred The predecessor of the list head
    @param outData The data to be removed
    @return true if the item is removed
            and false otherwise
*/
private boolean remove(Node<E> head, Node<E> pred, E outData) {
    if (head == null)  // Base case - empty list.
        return false;
    else if (head.data.equals(outData)) {  // 2nd base case.
        pred.next = head.next;  // Remove head.
        return true;
    } else
        return remove(head.next, head, outData);
}
```

Recursive `remove` Method (cont.)

```
/** Wrapper method for removing a node (in LinkedListRec).
    post: The first occurrence of outData is removed.
    @param outData The data to be removed
    @return true if the item is removed,
            and false otherwise
*/
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
```