Recursion and Linked Lists

Recursive list definition, list construction, delegation, operations

Recursive list definition

Recursive definition:

A **list**:
is empty
-or-
consists of a single element and another list

Consider the list 23, 45, 16. According to the recursive definition:

The list consists of 23 and a second list.
The second list consists of 45 and a third list.
The third list consists of 16 and a fourth list.
The fourth list is empty.

A way to write the list is (23 (45 (16 ()))).

Setting up the list

- The initial setup for a recursively defined list is similar to setting up an ordinary linked list.
- List has an instance variable of type List. This is identical to Node except:
  - the name (List vs Node)
  - the generic type element (<E>)
  - List is public (Node is private)
- List constructor requires a List element.

```
public class List<E>{
    public E element;
    public List next;

    public List(E element, List next) {
        this.element = element;
        this.next = next;
    }
}
```

Defining size( )

```
public class List<E> {
    public E element;
    public List next;

    public List(E element, List next) {
        this.element = element;
        this.next = next;
    }

    public int size() {
        if (next == null)
            return 1;
        else
            return 1 + next.size();
     }
}
```

List and size( ) observations

- The size of a list is at least 1.
  - When you create a new List, you must specify an element and another List (could be null).
  - *Our current definition cannot accommodate an empty list.*
- size() is recursive.
  - The stop condition occurs when the last element in the list has been counted (next == null).
  - The recursive call is to "another list," which is part of the recursive definition of a list.

Constructing an outer class

- We need to accommodate empty lists!
- Solution:
  - Nest List inside another class.
  - Make the inner List class private.
  - Put the element type as a parameter to the outer class.
- Structure:

```
public class RecursiveList<E> {
    private List list;
    // RecursiveList constructor and methods
    private class List {
        // List data (element, next)
        // List constructor and methods
    }
}
```

RecursiveList construction

```
public class RecursiveList<E> {
    private List list;

    public RecursiveList() {
        list = null;
    }
    public void addToFront(E element) {
        list = new List(element, list);
    }
    private class List {
        // List content go here:
        //    data, constructor, size()
        }
    }
}
```

Construction discussion

- The instance variable list mimics the abstract data type list.
- To add an element:

```
list = new List(element, list)
```

  So the list is null or element + another list

- Consider this statement:

```
RecursiveList<String> roster = new
                               RecursiveList<String>();
```

  `roster.list` is null, and `roster` represents a list with no elements. A `RecursiveList<String>` is created as an empty list.

- Add some elements to the front:

```
roster.addToFront("Sophie");
roster.addToFront("Anna");
roster.addToFront("Charles");
```

- The list is now ["Charles", "Anna", "Sophie”].

Delegation

Client code cannot call List.size(), so define RecursiveList.size()

```
public class RecursiveList<E> {
    private List list;
    // Constructor, addToFront go here

    public int size() {
        if (list == null)
            return 0;
        return list.size();
    }
    private class List {
        // Data members, constructor go here
        public int size() {
            if (next == null)
                return 1;
            else
                return 1 + next.size();
        }
     }
```

Delegation (continued)

- Recall roster is a RecursiveList<String> with 3 elements.
- Call trace on roster.size():
  - roster.size()
  - -- roster.list.size()
  - --- roster.list.next.size() --- 1 + (1 + (1))
  - ----- roster.list.next.next.size() --- 1 + (1)
  - ---------roster.list.next.next.next.size(); -- 1
- Since roster.list is not null, the responsibility for finding the size of the list goes from the roster object to its list object.
- This is an example of **delegation**, in which one object uses another object to handle some of its work.

Adding to the end of the list

- Adding an element to the front of a linked list is a one-step process. No delegation is required. Removing the front element is equally trivial.
- Adding an element to the end of a linked list is definitely not a one-step process (unless you have implemented the linked list to handle that problem efficiently).
- Our recursive list implementation delegates that responsibility from the outer list, down the chain to the last non-null List object.

Adding to the end - code

```
public class RecursiveList<E> {
    // data, constructor. addToFront, size go here

    public void addToEnd(E element) {
        if (list == null)
            addToFront(element);
        else
            list.addToEnd(element);
   }

    private class List {
   // data, constructor, size go here

        public void addToEnd(E element) {
            if (next == null)
                next = new List(element, null);
            else
                next.addToEnd(element);
        }
```

Get an ArrayList<E> equivalent

ArrayList<E> gives cheap access to individual list elements.

```
public class RecursiveList<E> {
    // data, constructor. addToFront, addToEnd, size go here

    public ArrayList<E> toArray() {
        if (list == null)
            return null;
        return list.toArray(new ArrayList<E>());
    }

    private class List {
   // data, constructor, addToEnd, size go here

       public ArrayList<E> toArray(ArrayList<E> al) {
           al.add(element);
           if (next != null)
               next.toArray(al);
           return al;
       }
```

Removing an element

Start with RecursiveList<E>.remove().

```
public class RecursiveList {
    // data, constructor, methods go here
    public void remove(E element) {
        if (list != null) {
            if (list.element.equals(element))
                list = list.next;
            else
                list.remove(element);
        }
    }
    private class List {
       // data, constructor, methods go here
       public void remove(E element) {
         // code with recursive call here
       }
```

Removing an element (cont)

Now let List do the heavy lifting.

```
public class RecursiveList<E> {
    // data, constructor, methods go here
    public void remove(E element) { // Recursive list
       // call: list.remove(element);

    private class List {
       // data, constructor, methods go here
       public void remove(E element) {
            if (next != null) {
                if (next.element.equals(element))
                    next = next.next;
                else
                    next.remove(element);
            }
        }
```