HOMEWORK PROBLEMS 15

1) What happens when you call `traverse` in Fig. 15.4 when the list is null?

2) What is the effect of

    ```
    p = p.link;
    ```

    where `p` points to a node in a linked list?

3) Change the first line of the `while` loop in `C15h3.java` to

    ```
    while (p.link != null)
    ```

    How does this change affect the behavior of the `traverse` method?

4) Line 27 in Fig. 15.7 is a direct reference to a private variable `x` from outside its class. Why is this legal?

5) Write a `for` loop equivalent to the `while` loop on lines 29 to 33 in Fig. 15.4.

6) Add a `length` method to `C15h6.java` that returns the length of the list (i.e., the number of nodes). Test your `length` method against both null and non-null lists.

7) Add a `getFirst` method to `C15h7.java` that removes the first node on the list and returns its data. If the list is empty, `getFirst` throws an exception. Test your method against both null and non-null lists.

8) Add an `average` method to the `C15h8.java.` Your `average` method should return the average of all the `x` data on the list. If the list is null, your `average` method should throw a `MyListException` (a class you should create). Your `average` method should return a `double` value. Test your `average` method against both non-null and null lists.

9) Add a constructor to the `Node` class in `C15h9.java` that is passed the initial values for the `link` and `x` fields. Simplify the `addFirst` method (this simplification is made possible by the use of the new constructor). Test your new method.

10) Add a method `rtraverse` to `C15h10.java`. Your `rtraverse` method should traverse and display the list in reverse order. Use recursion. Test your new method.

11) Add a copy constructor to the `C15h11.java`. It should create a deep copy. Test your new method.

12) Add an `addLast` method to `C15h12.java`. This method should create a new node for the integer data it is passed and add it to the end of the linked list. On each call of `addLast`, get the pointer to the last node by traversing the list from the first node to the last node. Then, using this pointer, attach the new node to the end of the list. Test your new method. Be sure to consider any special cases. Calling `addLast` when the list is non-null is the normal case. But calling it when the list is null is a special case—you have to treat it differently than the normal case. *Always make sure you correctly handle the special cases as well as the normal cases*.

13) Modify `C15h13.java.` Include a `last` field. This field points to the last node in the linked list, or is `null` if the linked list has no nodes. With `last`, it is easy to add a node to the end of the list. Without it, you have to traverse the list from the beginning to determine where the last node is, as you did in homework problem 12. Test your new `method`.

14) Add a `getLast` method to `C15h14.java`. Your `getLast` method should remove the last node on the list and return its data. If the list is null, `getLast` throws an exception. Test your `getLast` method against a null list, a list with one node, and a list with three nodes.

15) Add an `equals` method and a `toString` method to `C15h15.java`. Create four lists:

    List 1: 1 2 3
    List 2: 1 2 3
    List 3: 1 2 3 4
    List 4: null list

    Display list 3 and list 4 (i.e., display the string returned by your `toString` method). Test your `equals` method by comparing list 1 and list 2, list 1and list 3, and list 3 and list 4.

16) Add a `get` method and a `set` method to `C15h16.java`. `get(i)` should return the `x` data in the node whose index is `i` (indices start at 0—thus the first node has index 0). `get` does not affect the list. `set(i, y)` should set the `x` field in the node whose index is `i` to y. Test your methods by creating a list with 10 nodes, containing 1, 2, ..., 10 (use a `for` loop to do this). Then use a second `for` loop that sets the data to 10, 20, ..., 100, using the `set` method. Finally, use a third `for` loop that gets the data with the `get` method and displays it. If the node index passed to `get` or `set` does not correspond to a pre-existing node, throw a `MyListException` (this is a class you should create).

17) Add an `append` method to `C15h17.java`. `append` has one parameter of type `MyLinkedList`. `append` attaches a *copy* of list it is passed to the end of the list in its object. For example,

    ```
    list1.append(list2);
    ```

    attaches a copy of `list2` to the end of `list1`. Test your `append` method with the following pairs of lists:

    List 1: 1 2 3
    List 2: 4 5

    List 1: 1 2 3
    List 2: null list

    List 1: null list
    List 2: 4 5

    For each pair, you should display the two lists before and after the append operation.

18) Add a `reverseList` method to `C15h18.java` that returns a list in which the data is in reverse order. The returned list should not share any nodes with the original list. Test your method with lists that have zero nodes, one node, and three nodes.

19) Add a `remove` method to `C15h19.java` that removes the node at the specified index and returns its data. For example, if you pass this method the value 1, it removes and returns the second node of the list (node indices start at zero so the node with index 1 is the second node). If the node specified does not exist, `remove` should throw an exception. Test your method by creating a list that contains 1, 2, 3, and 4 in that order. Then remove the nodes that contain 1, 3, 4, and 2 in that order. Display your list after each removal.

20) Add an `addOrdered` method to `C15h20.java` that creates a node for the integer data it is passed and inserts that node into a list before the first node with equal or larger data. Starting with a null list, call `addOrdered` 10 times passing it random integers. Then display the resulting list.

21) Construct a class similar to `MyLinkedList` that creates a doubly-linked list. Include iterative `rtraverse` and `traverse` methods. Use a `head` and a `last` variable to point to the first and last nodes, respectively.

22) Give an application where a **circular list** can be put to good use. *Hint*: what is **round-robin** service? A circular list is a list in which the last node points to the first node. How can a circular list be traversed without the danger of an infinite loop occurring.

23) Change the list structure in `C15h23.java` to use a singly-linked circular list. Add a method that returns the pointer to the next node in round-robin fashion.

24) A one-dimensional array can be used to hold a linked list. Its downside is that it cannot easily change size like the list in `MyLinkedList`. Each node is represented by a pair of adjacent slots in the array. The first slot in the pair contains the index of the node it points to. The second slot would contain the data. Construct a class similar to `MyLinkedList`, but use an array to hold the list. Use the file name `C15h24.java`.

25) Draw a picture of the linked data structure that is built by the program below. What does the program display? The private `inorder` method performs an **inorder traversal** of the tree that `main` constructs. In an inorder traversal, at each node first the left subtree is traversed, then the root is visited, then the right subtree is traversed.

```
class C15h25
{
   public static void main(String[] args)
   {
      MyTree t = new MyTree();

      t.insert(5);
      t.insert(10);
      t.insert(2);
      t.insert(9);
      t.insert(3);
      t.insert(1);
      t.inorder();
   }
}
//======================================================
class MyTree
{
   private class Node
   {
      private int x;
      private Node left;
      private Node right;
   }
   //---------------------------------------
   private Node rootptr = null;

   public void insert(int xx)
   {
      Node n = new Node();
      n.x = xx;
      if (rootptr == null)
         rootptr = n;
      else
      {
         Node trailing = null, leading = rootptr;
         while (leading != null)
         {
            trailing = leading;
            if (xx < leading.x)
               leading = leading.left;
            else
               leading = leading.right;
         }
         if (xx < trailing.x)
            trailing.left = n;
         else
            trailing.right = n;
       }
   }
   //----------------------------------------
   public void inorder()
   {
      inorder(rootptr);
   }
   //----------------------------------------
   private void inorder(Node p)     // uses double recursion
   {
      if (p != null)
      {
         inorder(p.left);           // recursive call
         System.out.println(p.x);
         inorder(p.right);          // second recursive call
      }
   }
}
```

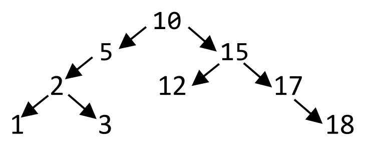
26) Replace the `insert` method in `C15h26.java` with an equivalent recursive method (difficult).

27) Replace the double recursive `inorder` method in `C15h27.java` with a method that has only one recursive call. *Hint*: The second recursive call in `C15h27.java` is tail recursive.

28) Add to the `MyTree` class in `C15h28.java` a method that does a **preorder traversal** of the tree constructed. In a preorder traversal, at each node, first visit that node, then traverse the left subtree of that node, then the right subtree of that node. *Hint*: Only a minor modification is required to convert the recursive traversal method: Simply switch the first recursive all with the `println`.

29) Add to the `MyTree` class in `C15h29.java` a method that does a **postorder traversal** of the tree constructed. In a postorder traversal, at each node, first the left subtree of that node is traversed, then the right subtree of that node, then that node is visited. *Hint*: Only a minor modification is required to convert the recursive traversal method (the private one) in homework problem 21: Simply switch the `println` with second the recursive call.

30) What is displayed when the following tree is traversed with preorder traversal algorithm? With an inorder algorithm? With a postorder algorithm? *Note*: Only the data in each node is shown in the diagram this diagram.



31) In what order does the data have to entered by `main` to result in the tree in homework problem 30? What tree would result if the data were entered in ascending order? In descending order?

32) Suppose a set of numbers were entered in random order by `main` in the `C15h25.java` program in homework problem 25. In what order would they be displayed by the `inorder` traversal?

33) Add a method to the `MyTree` class in `C15h33.java` that returns the sum on the integer data in the tree. Add code to `main` to call this method and display the value it returns.

34) Add a method to the `MyTree` class in `C15h34.java` that displays (in any order) the data in every node of a tree that has a left child but no right child.

35) Add a method to the `MyLinkedList` class in `C15h35.java` that converts the list to a circular list (i.e, the last nodes points to the first node. Your method should also initialize a `last` variable that points to the last node of the list. Modify the `traverse` method so that it works properly with the new list structure.

36) Would it ever make sense for the first variable in a linked list be a node rather than a pointer to a node? Redo homework program 19 with this modification. Use `C15h36.java`.

37) Write a recursive method that is passed the head pointer to a linked list. Each node has a `link` field and an `int x` field. Your method should return true if the number 7 appears one or more times on the list. Otherwise, it should return false.

38) Double recursion in the implementation of the `fib` method in homework problem 7 in Chapter 14 is extremely inefficient. Thus, the `fib` method should *not* be implemented in this way. But the double recursion in `C15h25` is not similarly inefficient. In fact, the double recursion implementation is generally preferred. Why the difference?