Topics: ListNodes, LinkedLists, Introduction to Recursive Data Structures
Ref: CSC20 PAL WS LinkedLists

The ListNode Class

The building block of a linked list is a 'node' object of the *ListNode* class. It has two fields, one to hold data and the other (called *next*) to hold the address of another *ListNode* object. This structure makes it easy to chain *ListNodes* into a *Linked List*

```
public class ListNode {
     public int data; //data field
     public ListNode next; // points to the next ListNode

     public ListNode() { // default constructor
          data=0;
          next=null;
          // this(0, null); another way to do the above.
     }
}
```

Here is how we create an empty *ListNode* object:

```
ListNode list = new ListNode();
```

This node's data field is default 0, and the 'next' variable is set to default null. We then update the data field to contain a specific value:

```
list.data = 3;
```

Let's create a second node to add to the existing list:

```
ListNode node2 = new ListNode();
node2.data = 7;
```

Here is how we connect to the second node, using the 'next' variable to store the address of the new node. This gives us a list of two 'linked' nodes:

```
list.next = node2;
```

Ex.1) Create a linked list of three connected nodes, with the following data values: [5, 7, 9]. Start with an empty node object.

After creating each additional node, link it with the previous node as shown above.

Now let's create a linked list without having to create each node individually. We will do this with modified constructors belonging to the *ListNode* class.

```
public class ListNode {
     public int data;
     public ListNode next;

     // default constructor
     public ListNode() {
          this(0, null);
     }

     // modified constructor1 for data only
     public ListNode(int data) {
          this(data, null);
     }

     // modified constructor2 for data and next node
     public ListNode(int data, ListNode next) {
          this.data = data;
          this.next = next;
}
```

With the first modified constructor, we can specify the node's data value:

```
ListNode list = new ListNode(3);
```

With the second modified constructor, we can specify the node's data and 'next' variables:

```
ListNode list = new ListNode(3, null);
```

We can also create an additional node through the 'next' parameter. The additional node here utilizes the modified constructor1.

```
ListNode list = new ListNode(3, new ListNode(7));
```

Ex 2) Use modified constructor1 to re-create the linked list from Problem 1.

Ex 3) Redo the problem using only modified constructor2.

One way of accessing and manipulating nodes is to use the next pointer to traverse. BJP Chapter 16 has several examples on how to search, add and remove list nodes. For example, given a list
list: [5] → [7] → [9] → [3],

Accessing the node 9 can be done as *list.next.next.data.*
Deleting node 7 can be done as *list.next=list.next.next.*
Deleting node 3 can be done *as list.next.next.next=null.*
Inserting a node 8 between 7 and 9 can be done as

```
ListNode eight = new ListNode(8,list.next.next)
        // create a ListNode with data =8 and next pointing to 9
List.next.next=eight; // connect 7 to 8
```

As you can see, managing multiple "next" hops can be confusing and error prone.

Linked lists may contain hundreds or thousands of connected nodes. We need a method to traverse a list, using a temporary variable (i.e. 'current') which points to specific nodes and can be moved.

In the following for-loop, we set a temporary variable called *current* pointing to the first list node. Then we re-position it to the following node in each iteration. The for-loop halts when *current* has moved past the last list node and has taken the value 'null'.

```
for(ListNode current = list; current != null; current =
current.next) {

}
```

With such a for-loop, we can: count the number of nodes in the list, search the list for a specific data value, count the number of nodes with a specific value, check if the linked list is sorted, etc.

Ex 4) Using the for-loop above, write code to count the number of nodes in the given linked list. Use a 'count' variable which is incremented by one for each loop iteration. Be sure to handle the case if the linked list is empty.

list: [5] → [7] → [9] → [3]

Ex 5) Using the for-loop above, write code to find this linked list's maximum data value. Be sure to check if the linked list is empty.

list: [5] → [7] → [9] → [3]

6) Write code to find if the linked list is sorted in ascending order. Hint: you will need to compare two successive nodes, *'current'* and *'current.next'*. Modify the for-loop condition to accommodate for this.

list: [5] → [7] → [9] → [3]

Inserting a new node into a linked list is simple if we need to place the node at either the front or end position. For example, here is to insert a node at the front of a list:

```
new_node = new ListNode(2);

new_node.next = list; // new node attaches to front of list

list = new_node;    // list now starts at the new node
```

Ex.7) Write code to insert a new node at the end of a linked list. You will need to iterate to the end of the list first, using the for-loop method provided.

Inserting into the middle of the list, however, requires two pointer re-assignments. In the linked

list [5] → [7] → [9] → [3], if we want to insert new node [2] between [9] and [3], we must re-assign [9]'s *'next'* pointer to store [2], and [2]'s *'next'* pointer to store [3]:

```
new_node = new ListNode(2);
for(ListNode current = list; current.next != null; current =
current.next) {
     if current.data == 9:
          ListNode temp = current.next; // store location of [3]
```

```
            current.next = new_node; // attach [9] to [2]

            new_node.next = temp; // attach [2] to [3]
}
```

Ex.8) In the following sorted linked list, insert the following new nodes (or sub-lists) such that the

list remains sorted in ascending order.

list: [2] → [5] → [8] → [12]

8a) Insert [1]

8b) Insert [3]

8c) Insert [7]

8d) Insert sub-list [10] → [11]

Deleting a node from a linked list requires re-assignment of node pointers as well. To delete [9] from the linked list [5] → [7] → [9] → [3], we will re-assign [7]'s next pointer to bypass [9] and store [3]. Java's garbage collection utility will erase the node [9], which is still pointing to [3].

```
for(ListNode current = list; current.next != null; current =
current.next) {
     if current.data == 7:
          current.next = current.next.next; //connect [7] to [3]
}
```

Ex. 9) Write code to remove all nodes with value less than 5 from list [8] → [4] → [6] → [2] → [9]. Do so without specifically halting the list iteration at specific nodes.

Additionally, do *Practice-it SelfCheck* exercises 16.9-16.14

The LinkedIntList Class

The *LinkedIntList* Class (in BJP Chapter 16) is a class that can be used to create a *LinkedIntList* object that can be used to store a list of integers. Each element of this list is a *ListNode* object that contains an integer in its data field and an address of the next *ListNode* in the next field.

A *LinkedIntList* object needs only to have the address of the front or the first node in the list and this can be used to traverse and manage the list. Methods for this *LinkedIntList* object are written to offer an *api* to do search and retrieve operations, and perform other updates to the linked list.
Refer to Practice-it Exercises for Chapter 16.

```
public class LinkedIntList {
    private ListNode front;   // node holding first value in
list (null if empty)
    private String name = "front";  // string to print for front
of list

    // Constructs an empty list.
    public LinkedIntList() {
        front = null;
    }
```

Recursion:

The recursive definition of List Node Objects offers the possibility of writing recursive solutions for LinkedList methods. Recursion will be covered in detail in another worksheet