

Lab 6

Objectives

- More practice with linked data structures
- Introduction to recursion on a list

Recursion overview

In this lab you will be completing the implementation of methods according to the documentation in `IntegerLinkedList` **recursively**.

The following image shows a general template for designing a recursive function that traverses a list. The public method `doSomething` calls the private helper method `recursiveDoSomething` passing it the head (the first element in the list) as an initial Node.

The method `recursiveDoSomething` takes a Node as a parameter. This Node can be one of two things:

- null
- a Node with some data and a next pointer

These two cases provide the basis for the if/else structure of this recursive function template:

- if (n is null) we are in the base case condition
 - o implement the base case answer/result
- otherwise n is not null and we pull apart n into its two pieces
 - o n's data (this is current piece of data – the method can do what it needs with it)
 - o n's next (this is the REST of the list – we pass it to a call to `recursiveDoSomething`)
This call will deal with the REST of the data in the list (all Nodes after n)

```
public ... doSomething() {  
    recursiveDoSomething (head);  
}  
  
private ... recursiveDoSomething (Node n) {  
    if (n== null) { //basecase  
        // basecase result here and return  
    } else {  
        // combine the following:  
  
        // the data in n  
        n.getData();  
  
        //recursive call on rest of the list  
        recursiveDoSomething (n.next);  
    }  
}
```

The diagram illustrates the recursive function template. A green arrow points from the `recursiveDoSomething` call in the public method to the start of the private method. Another green arrow points from the `n.next` argument in the recursive call back to the start of the private method, forming a loop that represents the recursive nature of the function.

Exercises

1. Download `IntegerLinkedList.java`, `Lab6Tester.java`, and `IntegerNode.java` to your Lab6 folder.
2. Start by opening `IntegerLinkedList.java`
 - a. Look at the method implementation for `addOneRecursive`.
 - b. Take time to see the structure of the template shown on the previous page in the implementation of `addOneRecursive`, and compare the structure to this alternative iterative implementation:

```
public void addOne() {
    IntegerNode cur = head;

    while (cur != null) {
        int valPlusOne = cur.getData() + 1;
        cur.setData(valPlusOne);
        cur = cur.next;
    }
}
```

- c. Notice the tests in `Lab6Tester.java` test the empty list case and the case of a list at least 3 elements long.
3. In `IntegerLinkedList.java`, test and implement the next two functions, `doubleValues` and `doubleOddValues`, one at a time. To help you with the implementation, follow these steps:
 - a. Read through the provided tests in `Lab6Tester.java` to ensure you understand how the method works. Add additional tests for longer lists or other edge cases as you see fit.
 - b. Write the template for the private recursive helper function (ie. `addOneRecursiveHelper`) and place a call to that helper function in the public stub method provided for you.
 - c. Add code for your base case answer/result
 - d. Add code to deal with the current Node
 - e. Run the tests you added for that method found in `Lab6Tester.java`

CHECKPOINT – Now might be a good time to check-in with the TA if you are not comfortable in your understanding of how we are expecting you to implement the methods recursively. Remember, please don't hesitate to ask questions if you are unclear about anything.

Look at the implementation of the recursive `Sum` method. This method also has that general structure of the template shown on page 1, but it is returning a result after it traverses the list unlike the previous functions you just wrote (they made changes to each Node as they traversed the list).

Since the method returns a value of type `int`, the result of the call to the private helper function must be returned and the two cases of the helper function must return a result of the expected type.

In this example:

- When `n` is `null` (the empty list case) – the sum is 0, therefore the base case result statement is: `return 0;`
- When `n` is not `null` – the sum will be the value in the current Node + the result of the recursive call on the rest of the list, therefore the result statement is: `return first + sumRest;`

4. In `IntegerLinkedList.java` test and implement the `product` method following the given documentation. Use the process from Step 3 above to help.

CHECKPOINT

Look at the implementation of the recursive `doubleOddPosition` method. This method also has that general structure of the template shown on Page 1, but it has an additional parameter. This additional parameter is called an accumulator which is used to keep track of and pass on context that would otherwise be lost in the recursive call. In this case, the accumulator (`position`) is keeping track of the position of the current Node within the whole list.

The position of the first element of the list is 0, the next element is at position 1, the next at position 2...

There are 4 steps to adding an accumulator to your recursive function:

- Add it to the parameter list in the private helper function:
`public void doubleOddPositionValues(IntegerNode n, int position);`
- Give the accumulator an initial value in the call to the private helper method:
`doubleOddPositionValues(head, 0);`
- Exploit (use) the value of the accumulator as needed within the method:
`if (position % 2 != 0) {`
- Pass the (likely updated) value of the accumulator as a parameter to the recursive call:
`doubleOddPositionValues(n.next, position+1);`

5. In `IntegerLinkedList.java` test and implement the next two functions, `allNegative` and `isSortedAscending` one at a time by following the given documentation.

HINT: Think about if you need an accumulator or not. Is there context that you need to know about from the previous steps of the recursion. One of these methods will need one! Again, use the process from Step 3 above to help and follow the steps given above to systematically add your accumulator.

CHECKPOINT – Remember to demonstrate your progress to your TA.