University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

---

# Implementation #3: Implementing a Generic Singly Linked List

---

# 1 Objectives and Overview

This assignment covers the following

- Nested vs Inner classes
- Singly linked-list structure and operations
- Another implementation of generic data-structures

**NOTE:** Complete all methods in the order they are presented. Do NOT move onto impelementing another method before completing one and testing it. With previous assignment you might have been able to get away with jumping between implementing different methods however as we move into the world of lists and trees doing this without testing and debugging can lead to compounding errors which are difficulty to debug.

# 2 Structures and Specifications

For this assignment you will only be implementing one `.java` file in addition to `main`. The `SinglyLinkedList<E>` class will be the class that represents our linked list as a whole and allows for operations to be performed on that linked list. The individual elements of our linked-list will be represented by the `ListNode<E>` class which will be a *static inner* class of `SinglyLinkedList<E>`.

## 2.1 Nested vs Inner Classes (docs)

```
class OuterClass{
    class InnerClass{
        //...
    }
}
```

```
class OuterClass{
    static class NestedClass{
        //...
    }
}
```

In Java there are two ways of embedding classes within other classes:

- **Inner Class:** This is a class that been declared within another class without the `static` modifier. It has access to all of the methods and attributes *regardless of the access modifier they were declared with.* That is to say, the inner class has access to all of the outer classes private variables and methods. Additionaly, you must first instantiate the outer class before instantiating the inner class.
- **Nested Class:** A nested class is similar to an inner class but is instead declared *with* a `static` modifier. This separates them more from their outer class and does not allow the nested class access to it's outer classes `private` variables. Declaring the embedded class as `static` also allows it to be instantiated regardless of whether its outer class has been instantiated.

In both cases these classes are declared for packaging convinience and the embedded classes are small classes that are only of utility to their outer class. Our `ListNode<E>` class is an example of this since it has no real utility outside the contex of it's usage in `SinglyLinkedList<E>`. It also doesn't need to access any of it's outer classes variables or methods so it makes sense to declare it as a *nested class* rather than an *inner class*. This leaves us with the following class structure for our linked-list.

```
class SinglyLinkedList<E>{
    static class ListNode<E>{
        //...
    }
}
```

# 3 Step 0: Understanding what you are given

Before you begin it is important to understand the code you have been given and it's functionality. As such, *your are highly encouraged* to read through the existing code in the SinglyLinkedListText.java file as well as the provided code in SinglyLinkedList.java, the latter of which is described below.

## 3.1 ListNode

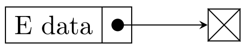



Figure 1: Example ListNode

### 3.1.1 Attributes

1. `E data`: A reference to a generic data. Leave it at the default access level.
2. `ListNode<E> next`: A reference to the next node in the list. Leave it at the default access level.

And that's it! This class doesn't have any methods just for simplicity's sake since it's only purpose is to store the data associated with elements of the linked-list. An example of how list nodes will be represented in future diagrams can be seen in Figure 1. This node has some generic `data` and the `next` attribute is initially `null` (depicted as pointing to a box with an X).

### 3.1.2 Constructor

The constructor for this class takes a single, generic parameter. It initializes the `data` attribute with the parameter and initializes the `next` node to be null.

## 3.2 SinglyLinkedList

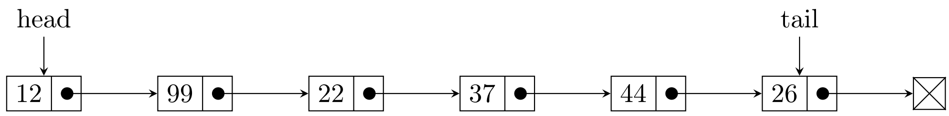



Figure 2: A visual represention of this class as a chain of `ListNode<E>` instances along with a `head` and `tail` reference.

### 3.2.1 Attributes

1. `ListNode<E> head`: This is a reference to the first node in the linked list. This has a `private` access level as we don't want users of the class to directly modify it. Rather, we want to control their access via the add/remove methods.
2. `ListNode<E> tail`: A reference to the last node in the linked list. This should have a `private` access level for the same reasons as stated above
3. `int size`: The current number of nodes in the list. This should have a `private` access level as the class provides a `size()` method with allows read access but no setter so the user of the class can't directly modify it.

### 3.2.2 Constructor

Your linked-list class has the following constructor:

1. `public SinglyLinkedList(E data)`: This constructor initializes `head` and `tail` attributes to `null` and set the initial size of the linked-list to 0. This is because, upon initially creating the list it is empty.

# 4 Step 1: Implementing the addToFront and addToEnd Methods

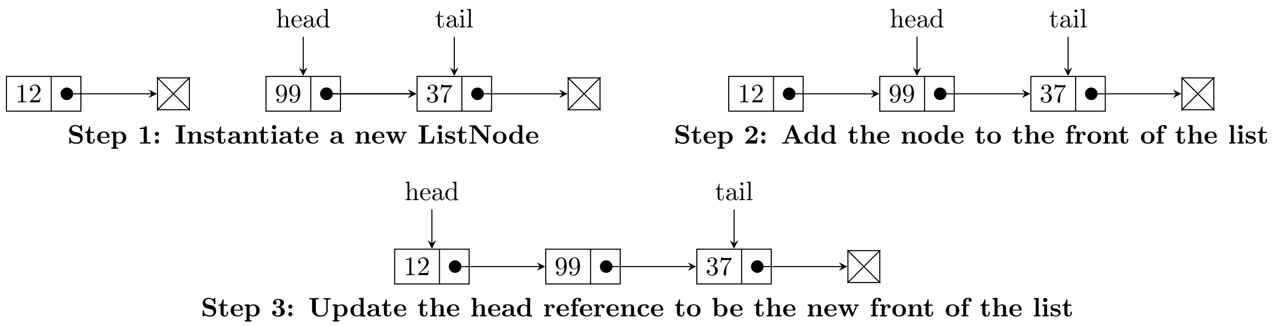



Figure 3: Adding a node to the front of a non-empty LinkedList

`public void addToFront(E data):` This method takes a single generic parameter, generates a new `ListNode<E>` with that data, and adds it to the front of the linked list. When making this method you should consider two sub-cases: (1) the linked-list is empty and (2) the linked list contains nodes. In the case the linked list is empty (i.e., `head` == `null`) you will want to set the tail and the head equal to the new node. Otherwise, you will want to set the new node's next reference equal to the current `head`, update the `head` to be the new node, and increment the size attribute to indicate a new node has been added (see Figure 3).

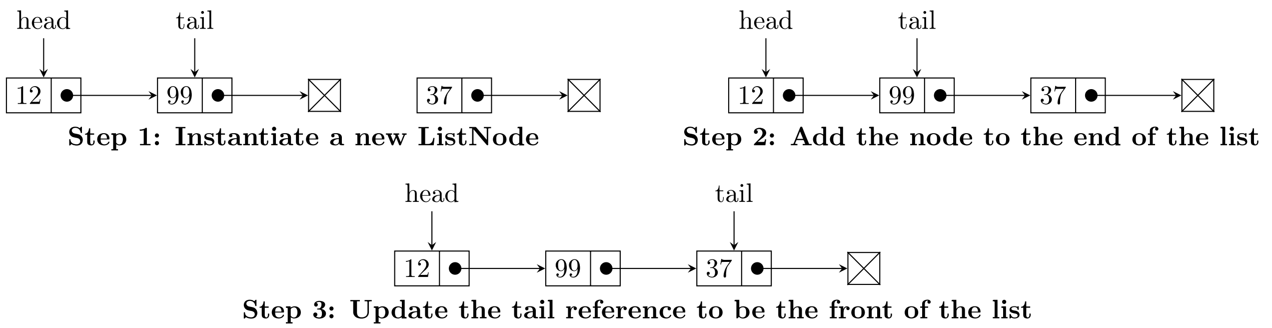



Figure 4: Adding a node to the end of a non-empty LinkedList

`public void addToEnd(E data):` For this method you will be adding to the end of the end of the list. This operation should be fairly similar to the `addToFront` method in that you should consider the following two cases: (1) the linked-list is empty and (2) the linked-list contains some nodes. You will begin by instantiating a new node using the data passed in as a parameter. Next, if the `head` is `null`, we want to set both the `head` and the `tail` to equal the newly created node. If the list is non empty, we want to set the new node be the current `tail`'s next node and then update `tail` to be the new node since it is now at the end of the list (see Figure 4).

## Testing

You have provided test cases for both of the above methods. Upon running the `SinglyLInkedListTest.java` file

```
addToFront tests
List 1: 3 --> 2 --> 1 --> null
------------------------------------------

addToEnd tests
List 2: 1 --> 2 --> 3 --> null
------------------------------------------
```

# 5 Step 2: getNodeAtPosition

Much like the `indexOf` method in the last assignment you will find the following method quite useful, particularly when getting nodes that occur before other nodes. For example, if we want to add a node at a given `pos` we need a reference to the node that precedes it. This can be done by calling the method below as: `getNodeAtPosition(pos -1)`. This is why we are implementing this method now given it will greatly simplify our code later on.

`public ListNode<E> getNodeAtPosition(int pos)` This method should return a reference to a node at a given position (`pos`). In the event the position (`pos`) exceeds the last valid index or the position is negative of the list this method should throw an `IndexOutOfBoundsException`. Otherwise, you should create a `tmp` pointer to the head, advance it `pos` number of times, and return `tmp`.

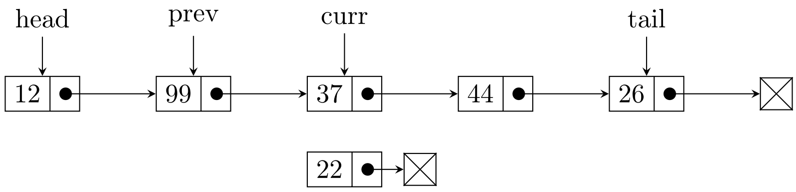
## Testing

The provided tests for get node at position should output the following after you have implemented this method:

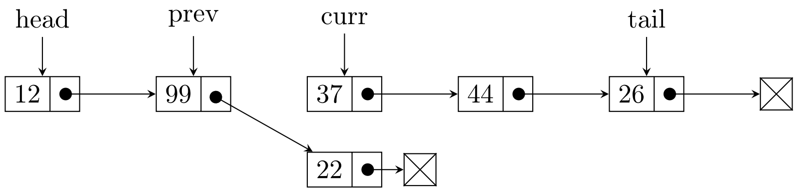
```
getNodeAtPosition Tests
The node containing 1 is at position 0 in List 2
The node containing 2 is at position 1 in List 2
The node containing 3 is at position 2 in List 2
------------------------------------------
```

As was the case with the last one, you should review the test code in SinglyLinkedListTest.java and add a few more.

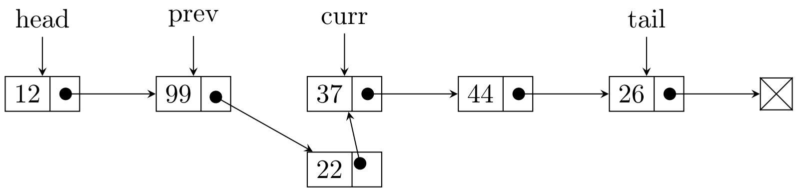
# 6 Step 3: addNodeAtPosition




**Step 1: Search for and find a reference to the node you want to remove and the node that precedes it.**




**Step 2: Update previous node's next node to be a reference to the new node.**




**Step 3: Update the new nodes net node to be a reference to the current node**

Figure 5: Add node to the middle of a non-empty LinkedList

`public void addNodeAtPosition(int pos, E data)` : This method should instantiate a new node with the data passed in as a parameter and insert it at a given position. Completing this method has three distinct stages:

1. In the event `pos` is not a valid index (e.g., pos $>$ size or pos $<$ 0) the method should throw an `IndexOutOfBoundsException`.
2. If the node position (`pos`) is the front (Case 1; pos $==$ 0) or the end (Case 2; pos $==$ size) it should call the appropriate method (i.e., addToFront, addToBack) to insert node.
3. If it is somewhere in the middle of the list (e.g., 0 $<$ pos $<$ size) it should search for the node that currently occupies that position and insert the new node before it (Case 3; Figure 5). As a reminder, this is where we can use `getNodeAtPosition(pos)` and `getNodeAtPosition(pos - 1)` to get `curr` and `prev` respectively.

## Testing

The provided tests for get node at position should output the following after you have implemented this method:

```
addNodeAtPosition Tests
List 3: 1 --> 2 --> 3 --> 2 --> 1 --> null
------------------------------------------
```

As was the case with the last one, you should review the test code in SinglyLinkedListTest.java and add a few more.

# 7 Step 4: removeFromFront and removeFromEnd

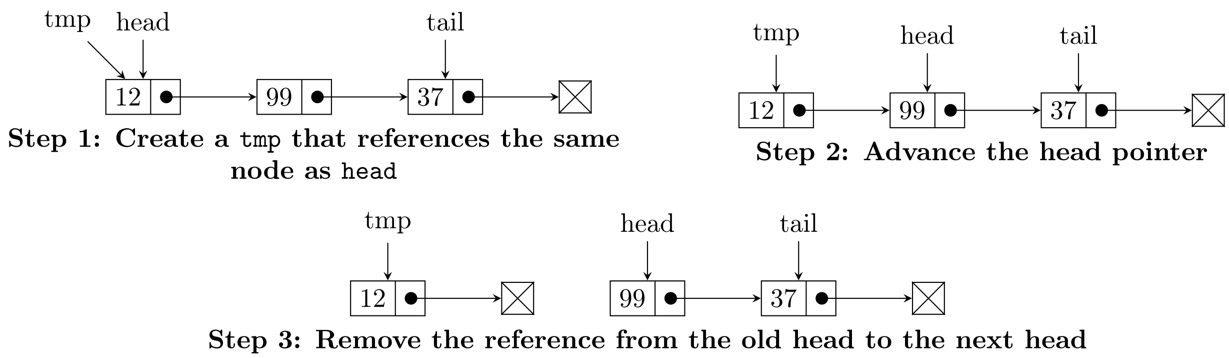



Figure 6: Removing a node from the front of a non-empty LinkedList

`public void removeFromFront():` As the name suggests you will implement a method that removes the node from the front of the linked-list. Implementing this method has three cases to consider:

1. If the list is empty, the method should throw `IndexOutOfBoundsException`.
2. If the list has one item (e.g, `head` == `tail`) we must set the `head` AND `tail` to null.
3. Otherwise, we remove and update the head pointer as shown in Figure 6.

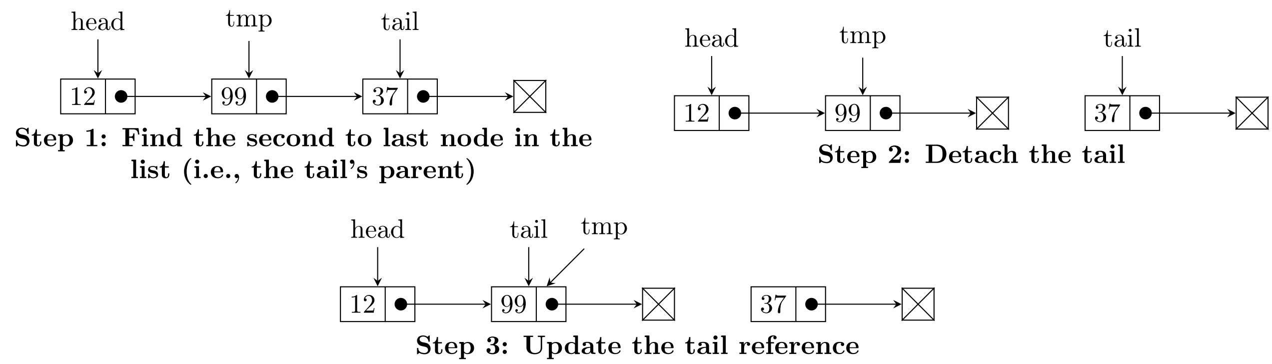



Figure 7: Removing a node to the end of a non-empty LinkedList

`public void removeFromEnd(E data):` Similar to `removeFromFront` you will implement a method that removes the node from the back of the linked-list. This also has three cases to consider:

1. Again, if the list is empty, the method should throw `IndexOutOfBoundsException`.
2. If the list has one item (e.g, `head` == `tail`) we must set the `head` AND `tail` to null.
3. Otherwise, we remove and update the head pointer (see Figure 6). For this, you can once again your `getNodeAtPosition` method you implemented earlier to get the second to last node.

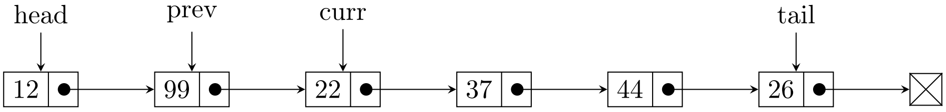
## Testing

The provided tests for get node at position should output the following after you have implemented this method:

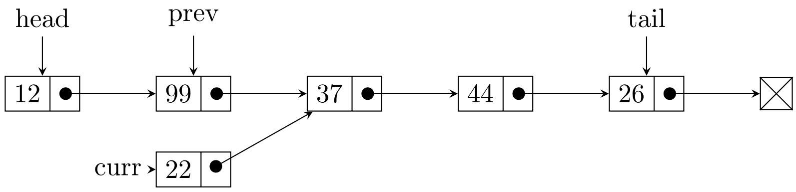
```
Remove from front of List 1
List 1 after front removal: 2 --> 1 --> null

Remove from end of List 2
List 2 after front removal: 1 --> 2 --> null
```

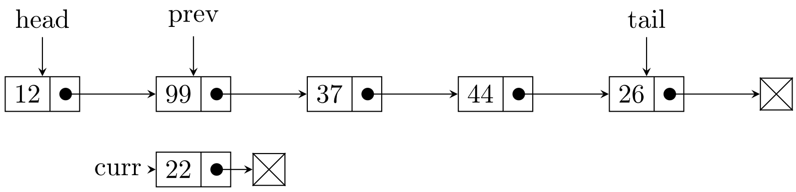
# 8 Step 5: removeNodeAtPostion




**Step 1: Search for and find a reference to the node you want to remove and the node that precedes it.**




**Step 2: Update previous node's next node to be the current nodes next node.**




**Step 3: Null out the current nodes next node to fully remove it from the list**

Figure 8: Removing a node from the middle of a non-empty LinkedList

`public void removeNodeAtPosition(int pos)` : This method should find the node at a given position. If the node position is at the head (Case 1; pos == 0) or at the end (Case 2; pos == size - 1) it should call the appropriate method to remove that node (i.e., the ones you implemented at the beginning of the assessment). If it is somewhere in the middle of the list (e.g., 0 < pos < size -1) it should search for that node and remove it (Case 3; Figure 5). In the event pos is not a valid index (e.g., pos > size - 1 or pos is negative) the method should throw an `IndexOutOfBoundsException`.

```
Remove position (front, end, middle) from List 3
List 3 after removing middle, front, and end: 2 --> 2 --> null
```

# 9 Hints

For this assignment you will be creating and throwing exceptions when invalid operations are attempted to be performed. Exceptions, in Java are created via two elements:

1. The method header specifies that it throws an exception, as shown in the following template: `public void foo()throws Exception{ /*... */}`.
2. A new exception is created and thrown: `throw new Exception();`

In this assignment you are told at a number of points to throw an IndexOutOfBoundsException. You are given the method headers so all you need to do is use a conditional to determine when the exception should be thrown then fill in the code that throws the IndexOutOfBoundsException in the body of that if statement. Example code for this is shown below.

```
public ListNode<E> getNodeAtPosition(int pos) throws
    IndexOutOfBoundsException{
        if(pos > size - 1 || pos < 0) {
                throw new IndexOutOfBoundsException();
        }

        return new ListNode<>((E) new Object()); // Remove this line once
            you begin to implement this method
}
```

## 9.1 Checklist

- ListNode: All the stuff here is given to you so check off if you didn't modify it :)
- SinglyLinkedList
  - ☐ The following accessors and list modification methods have been implemented:
    - ☐ `getNodeAtPosition`
    - ☐ `addToFront`
    - ☐ `removeFromFront`
    - ☐ `addToEnd`
    - ☐ `removeFromEnd`
    - ☐ `addNodeAtPosition`
    - ☐ `removeNodeAtPosition`