COSC 1P03 (Winter 2022) – Assignment 5

Maysara Al Jumaily

April 2^{nd}, 2022

NOTE

Please read the assignment **before** reading this document because you need to familiarize yourself with it first.

Contents

Note for the impatient

If you really want to just complete the assignment and have sufficient knowledge about the topics in the previous page, go to section 2. Refer to the notes skipped when studying for the exam.

1 The Concepts of the Assignment

We will explain the keywords `this`, `interface` and `extends` as well as the notation `<E>`, the `Comparable` interface and `Iterator` interface.

1.1 `this` Keyword

The `this` keyword refers to instance variables and methods in the current class (can call a method as `getX();` or `this.getX();`). **Furthermore**, we can call a constructor from another constructor (a.k.a *constructor chaining*) as well as pass `this` keyword to a parameter.

- Say that we are in the default constructor (doesn't accept parameters) and we wanted to call a constructor that accepts a single `int` inside the default constructor:

```
public Test(){
    this(100);//Calls the constructor that accepts a single int
    //MUST write it in the first line of constructor
    //more code here if needed...
}
public Test(int x){
    System.out.println(x);
}
```

- Passing `this` to a parameter
  - Assume a method `printData` declared *inside* of the `Turtle` class as:

    ```
    public void printData(Turtle t){ ... }
    ```

  - In the `Turtle` class, you can call `printData` and pass `this` which means pass the current values of the `Turtle` object into `printData`
  - The current class is an an object of type `Turtle`, pass the *current* state of to the method (no extra instances/duplicates are created), which include all the values of the instance variables
  - The concept is used in `ConList.java` in the `iterator()` method

1.2 `implements` Keyword

In Java, *polymorphism* (an object that has a definitive structure which is represented by different implementation) is achieved through the usage of interfaces. An *interface* is something that only contains instance variables and public method headers but no implementation. It defines a structure but *not* the implementation. A class can `implements` an interface by coding the methods founds in the interface. Consider the following interface:

```
public interface Calculator {
    public int add(int x, int y);
    public int subtract(int x, int y);
}
```

An implementation class could be[1]:

```
public class SimpleCalculator implements Calculator {
    public SimpleCalculator() { }//empty constructor, that's fine
    public int add(int x, int y) {  return x + y; }
    public int subtract(int x, int y) { return x - y; }
}
```

1.3 `extends` Keyword

In Java, *inheritance* is the idea where a class inherits or builds upon another class (or an interface builds upon another interface) through the usage of the `extends` keyword. A class will automatically have all the public instance variables and methods. Again, the ones with `public` keyword. The same goes with interfaces. Consider the original class:

```
public class Message {//just a class, no interfaces
    public String x = "Some text...";
    public Message() { } //empty constructor, that's fine
    public String getGreetings(String name) { return "Hello " + name; }
    public void sayHi() { System.out.println("Hi"); }
}
```

Let us inherit these methods using the `extends` keyword followed by the class to inherit from:

```
public class PowerfulMessage extends Message {
    public PowerfulMessage() {
        //no instance variables/methods are present but they are inherited
        String s = getGreetings("John");//valid
        sayHi();//prints Hi
        System.out.println(x);//the instance variable, valid!
    }
    //Here, you can add more methods as you already inherit the
    //public methods and instance variables from Message.
}
```

You have to be careful because you can *override* a method. For example, let us say you didn't like the implementation of the `sayHi` method found in the `Message` class. Then you can override it (*i.e.*, explicitly write it in `PowerfulMessage`) using the same method name, return type, and parameters. Here is how the the class will look like:

[1] IntelliJ will automatically add `@Override` annotation which is fine, keep it.

```
public class PowerfulMessage extends Message {
    public PowerfulMessage() {
        //no instance variables/methods are present but they are inherited
        String s = getGreetings("John");//valid
        sayHi();//previously printed Hi, but now prints Hello
        System.out.println(x);//the instance variable, valid!
    }
     public void sayHi() { System.out.println("Hello"); }//overridden
     //brand new method only found in PowerfulMessage class
     public void sayBye() { System.out.println("Bye"); }
}
```

1.4 implements vs extends

This is in its own subsection because it is an important distinction to know. In a nutshell:

implements vs extends

- Can implement as many interfaces as you want
- **Cannot** implement a class (can implement interfaces only)
- An interface can extend another interface (one interface at most) not a class
- A class can extend another class (one class at most) but not an interface
- Java doesn't support multi-inheritance (that is why it is always one at most)

1.5 Generics (<E> Notation)

The `<E>` notation represents the generics or parametrization concept. Think about a stack (first-in-last-out), wouldn't you agree that the structure is independent of the type being used? For example, if you are using `int`s or `double`s or `String`s, you will always remove the last element inserted regardless of the type being used. Java developers said why not we ignore the type and focus on the structure instead. We will assume the type is `E` and when we want to create that structure, we need to substitute `E` with an actual type (like `String` for example). Consider this simple generic class that stores an object of type `E` and then prints it:

```
public class Print<E> {
    private E data;//local variable
    public Print(E value) { data = value; }//constructor
    public void display() { System.out.println(data); }//prints the data
}
```

The class declaration uses `<E>` to denote "generics". The letter `E` is a placeholder and can be changed to a different letter like `<Z>` (or word) but stick with `<E>` for convention purposes. To run this, we will use the `Test` class (no implements nor extends, just a simple class):

```
public class Test {
    public static void main(String[] args) {
        //Substitute E with String
        Print<String> a = new Print<String>("Hey");
        a.display();//Hey
        //Substitute E with Integer, not int!
        Print<Integer> b = new Print<Integer>(7);
        b.display();//7
        //Substitute E with Double, not double!
        Print<Double> c = new Print<Double>(5.0);
        c.display();//5.0
    }
}
```

When using generics, you will substitute the generic type `E` with an object, *not* a primitive type (this is by definition). Java has eight primitive types and everything else is seen as an `Object`. Each of these eight has its object equivalent class:

Primitive type	Object equivalent
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Table 1: The primitive type and their object equivalent

1.6 `Comparable` interface

The `Comparable` interface (not to confuse it with the `Comparator` interface) is a way for you to compare two objects of the **same** type and is used for sorting. It accepts a generic type `<E>`, where `E` is the type you will compare. It has a single method:

```
public int compareTo(E other);
```

which accepts the generic type object to compare with the current data and returns an `int`. In the case where the instance variable value is the same as the one passed, `return 0;`, if the instance variable value is larger than the one specified, `return +1;`, otherwise, if the instance variable value is smaller, `return -1;`. The two entities you will compare will be explained further in this document. Refer to `KeyedChar` to see this.

1.7 `Iterator` interface

The `Iterator` interface (not to confuse it with the `Iterable` interface) is a way for you to loop through some data structure by specifying what should be looped first, second, etc.

Think about `Pictures` from COSC 1P02, why is it defined that you start top-left, end bottom-right and traverse row by row? Someone had to define a traversal order. They chose that order but it is possible to come up with another order. This allows us to use a for-each loop to loop through a `Picture`, like so:

```
for(Pixel p : aPic){
    //loops through all pixels once and only once starting from top-left
    //and ending bottom-right traversing row by row.
}
```

Since a for-each loop is used, it means that `Picture` implements the `Iterable` interface. **Without implementing the `Iterable` interface, we cannot use a for-each loop**. This interface has the following three methods:

- `public boolean hasNext();`
  - Returns `true` if there are more element(s) to visit, `false` otherwise
- `public E next();`
  - Returns the next element to visit
- `public void remove();`
  - Removes the current element from the data structure but you will not implement this in the assignment. Instead, throw `UnsupportedOperationException`.

The for-each loop is designed to continue looping as long as `hasNext()` returns `true`. Once it returns `false`, it stops looping.

2 Part A

Given Week10 code example, it contains a `List` implementation using an array. Your objective is to implement the same `List` using a linkedlist (from scratch, don't use `import java.util.linkedlist`).

A `List` is defined to have a cursor which marks where the insertion and deletion should occur, instead of just always inserting/deleting from the beginning, for example. It contains following operations (easiest to hardest): `length`, `empty`, `offEnd`, `toFront`, `advance`, `get`, `find` `add`, and `remove`.

2.1 `length` Method

```
public int length() { ... }
```

Returns the length (number of *valid*[2] nodes) in the linkedlist. It initially starts with `0` nodes.

2.2 `empty` Method

[2] A valid node is a node that is not `null`. Don't confuse this with a node that holds a `null` value (*i.e.*,, instead of storing the `item` as `"Hey"` or `7`, `item` would be `null`). The *item* of a valid node can be `null` but not the entire node itself.

```
public boolean empty() { ... }
```

Returns `true` if the linkedlist doesn't have any *valid* nodes, `false` otherwise.

2.3 `offEnd` Method

```
public boolean offEnd();
```

Returns `true` if `cursor` is pointing to `null`, `false` otherwise.

2.4 `toFront` Method

```
public void toFront();
```

Makes `cursor` point to the beginning of the linkedlist.

2.5 `advance` Method

```
public void advance();
```

Makes `cursor` points to the next node (shifts it once to the right). Note that if we are already at the end of the linkedlist (the end is denoted by `null`), we skip the operation (don't throw anything, just ignore it).

2.6 `get` Method

```
public E get();
```

Returns the item of the node `cursor` is pointing to. It is of type `E`, refer to subsection 1.5 if you don't understand generics. In the case where `cursor` is `null`, then throw `NoItemException`.

2.7 `find` Method

```
public void find(E element);
```

It will try to find the element specified. You will start (and including) the node `cursor` is pointing to and continue moving to the next node (*i.e.,* to the right) until you hit `null` (ignore the left side of `cursor`, it will not be searched). In case you found the element somewhere, stop the search and make sure `cursor` is pointing to the node of that element. In case where you had to loop through all remaining elements and `cursor` is `null`, keep `cursor` as `null` and the search is done (no exceptions are thrown).

2.8 Methods Changing the Linkedlist

This is the most difficult part! understand it and draw it first before coding.

Tip

When doing your logic, treat and check`head` separately than `cursor`. In Tutorial03, we only checked for `head`, now, you check both `head` and `cursor`. What are you going to check? Here is an example, `head` is *not* `null` but `cursor` is. Just because `head` isn't `null`, it doesn't mean the same goes to `cursor`.
Note: you can use `length()` and `empty()` here (assuming you implemented them *perfectly*) or check if `head` is `null` instead of `empty()` or `length() == 0`.

Warning

Don't run the code yet! Read the debugging section first: subsection 3.3.

2.8.1 `add` Method

```
public void add(E item);
```

It will insert `item` into a new node using the following syntax:

```
Node<E> n = new Node<E>(_______, _______);
```

Inserting will not be just to the right of the cursor or to the left of it, it will depend on the position of the cursor. There will be no explanation for this but there are visualizations of different scenarios, piece the logic yourself (the visualizations substitute the type `String` in `E`). Please have a look at Figure 1. **Increment the counter!**

2.8.2 `remove` Method

```
public E remove();
```

Similar to insertion, deletion will not be from a single side only. You will throw `NoItemException` if `cursor` is `null`. Just before you delete the node, store its item, then delete the node and return the stored item. Please have a look at Figure 2. **Decrement the counter!**

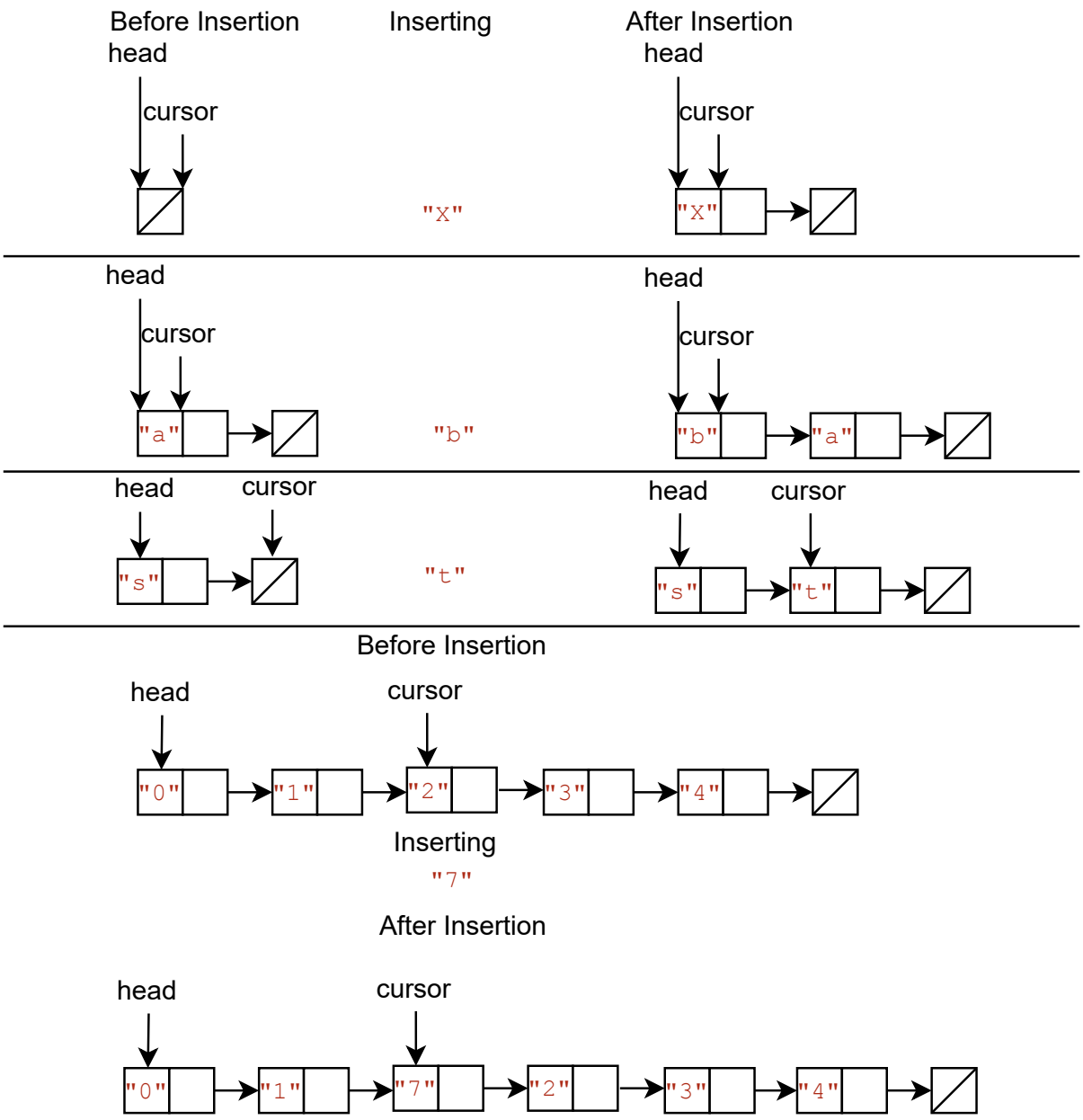

Figure 1: Four different insertion scenarios separated by horizontal lines. The left side is the linkedlist before insertion, the middle is the value to insert and the right side is the linkedlist after insertion. The last scenario is inserting without encountering an edge case.

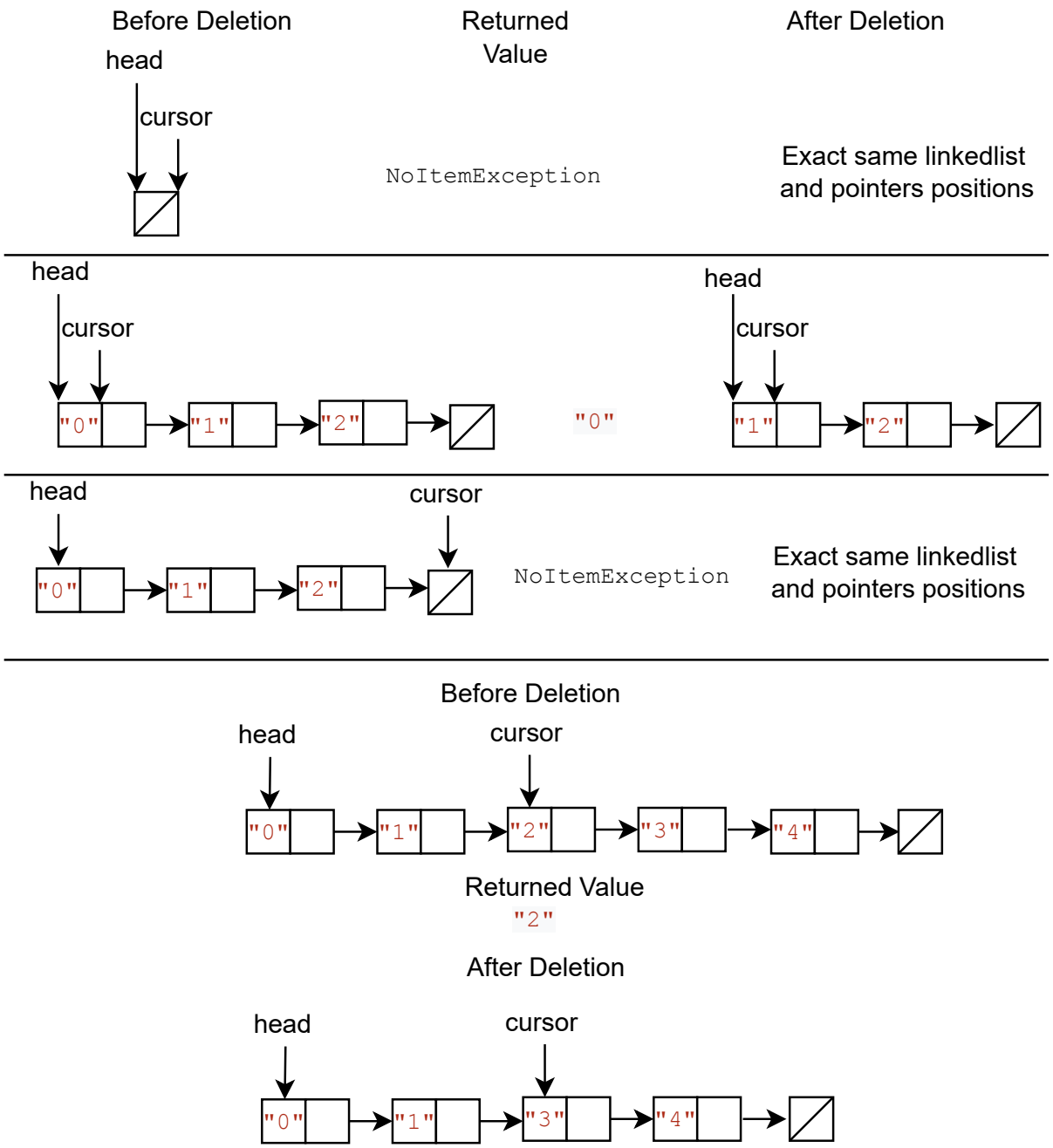

Figure 2: Four different deletion scenarios separated by horizontal lines. The left side is the linkedlist before deletion, the middle is the value to return and the right side is the linkedlist after deletion. In the case where `NoItemException` is thrown, then nothing in the linkedlist will change. The last scenario is deleting without encountering an edge case.

3 Part A Implementation Classes

There will be two implementation classes: `DynamicList`[3] and `DynamicListIterator`.
Note: None of the classes in this assignment throws `NoSpaceException` but keep the file in the project so that `ConList` would also work.

3.1 The `DynamicList` Class

This implements the `List` interface (and other matter) using generics but **not** the `Iterator` interface (`Iterator` will be discussed next subsection). The class will be declared as the following (you **MUST** use this declaration, otherwise, your assignment is not complete[4]):

```
public class DynamicList<E extends Comparable<E>> implements List<E> { ... }
```

Let us explain that:

- `List<E>` is the interface that uses generics
- `DynamicList<E extends Comparable<E>>`
  - We know that `DynamicList< ... >` means it is a generic type
  - The text inside of `< ... >`, which is `E extends Comparable<E>` means accept a class `E` that implements[5] the `Comparable` interface and it compares the same type
  - For example, if you create a class named `Data`, then `DynamicList` accepts `Data` only if `Data` uses `Comparable` and compares `Data` objects
  - An example that is **not** valid is having `Data` use `Comparable` but compares anything *other than* `Data` objects (maybe it compares `String`s for example)
  - Out of the classes given to you, which one implements the `Comparable` interface and compares the same objects?

head vs cursor

`head` denotes the beginning of the linkedlist. `cursor` (pointing to somewhere on the list or `null`) is a pointer accessible to *all* methods, where should it be declared?

3.1.1 Required: `getFront` Method in Implementation Class

REQUIRED

This method is **REQUIRED** for the iterator portion! You cannot complete the assignment without it. Insert it in the implementation class `DynamicList` (not in the `List` interface) and will be discussed next page:

```
public Node<E> getFront() {
    return head;//the front of the linkedlist
}
```

[3] The professor named it `LnkList` but this document uses `DynamicList`
[4] The `ConList` class implemented the `Serializable` interface but not needed here
[5] Yes, it is implements, *not* extends but Java developers said whether you are using implements or extends, always use the `extends` keyword in generics delimiters `< ... >`

3.2 The `DynamicListIterator` Class

Now that `DynamicList` is done, create the iterator! Use the following class declaration:

```
public class DynamicListIterator<E extends Comparable<E>> implements Iterator<E> { ... }
```

- `Iterator<E>` means we will create the logic of traversal of some generic element `E`
- Similar to `DynamicList`, this class accepts a type `E` that implements the `Comparable` interface comparing objects of the same type (refer previous page for more details)
- Create a constructor that accepts a generic `DynamicList` to iterate through
  - **DO NOT** use `cursor` found in `DynamicList`, you will create a new cursor (a generic `Node`) that will start at the *very* beginning of the linkedlist specified and visits all the nodes (left to right until it is `null`)
  - Does it click now as to why you need the `getFront()` method?
- Look at `ConListIterator` which is given in week 10 example
  - Note that `ConListIterator` uses an `int` as the cursor because arrays are accessed by indices. However, in our case, we have a linkedlist of `Node`s so we have to use a `Node` type (make sure it is generic `Node`)
- Implement the `hasNext()` and `next()` methods
- Don't implement the `remove()` method, keep it as:

```
public void remove() {// not supported...
    throw new UnsupportedOperationException();
}
```

- This code should be very minimal and the logic is simple! The entire class (without comment) should be no more than 40 lines

3.3 Debugging Part A

This is just for testing purposes, don't submit the code provided here:

- Create a `print()` method in `DynamicList` so that you would print out the linkedlist once testing (use `System.out.println(...);` to do so)
- When inserting/deleting, you should also know where is `cursor` before and after the operation (*i.e.,* print `cursor.item`) as well as length
- You will see outputs like `KeyedChar@13969fbe`
  - To see the actual character store in the node, go to `KeyedChar` class and add:

```
public String toString() {
    return theChar + "";
}
```

- The methods `length`, `empty`, `offEnd`, `toFront`, `advance`, `get` and `find` should be completed first (in that order)
  - They are very simple but if implemented incorrectly your code will never work
  - Look at the `ConList` class for aid
  - Without `add` and `remove` methods, you are still able to complete the `DynamicListIterator` class

IMPORTANT

DON'T use the code in the `TestLists` class for debugging (use it for **submission** but not testing). It will ruin your logic because you insert/delete relative to `cursor`. So, if `cursor` is moved programmatically in the test code, it will change the entire representation of the list. Instead, use something like this:

```
//delete the testList method and replace it with:
private void testList(List<KeyedChar> l) {
    l.toFront();//start at beginning (very far-left)
    System.out.println("adding: " + "A");
    l.add(new KeyedChar('A'));
    System.out.println("adding: " + "B");
    l.add(new KeyedChar('B'));
    System.out.println("adding: " + "C");
    l.add(new KeyedChar('C'));
    l.advance();//shift once to right
    System.out.println("removed: " + l.remove());
}
```

Now, **YOU** can move `cursor` manually using `l.toFront();` or `l.advance();`.

- Now that you read the big note above, code `add(...)` correctly by looking at the scenarios shown in Figure 1. Once you are happy with it, replace the minimal testing code with the original. Look only at the insertion portion in the output.
  - Only go to the `remove` method once you figure out this one
- Now the `remove()` method
  - Code the method correctly by looking at the scenarios shown in Figure 2
  - Use your minimal and customized testing code first
  - Once you are happy (and everything else is completed including the iterator), run the testing code provided by the professor and you should get the **exact** same output as using `ConList`
- When testing the code provided by the prof, you can:
  - Comment out the line:

    ```
    l = new ConList<KeyedChar>(100);
    ```

  - Type the following line below it:

    ```
    l = new DynamicList<KeyedChar>();
    ```

  - Now you can comment/uncomment one line run one of them quickly
- Make sure to change the output text at the very beginning in the `ASCIIDisplayer`!
- **For submission, make sure you undo all of the debugging code and run the exact same code provided to you in week 10 example**

4 Part B

- Create a new interface named `SortableList<...>` which `extends` the `List<...>` interface
  - Use the name `E` as the generic type
  - You will have to use `<E extends Comparable<E>>` somewhere in the interface header (hint: look at `List` interface file)
  - The extra method to include is:

    ```
    public int Sort();
    ```
- Create a new class named `DynamicSortableList` which is the copy of `DynamicList` (create the class and copy/paste the code into it and make sure the class name and constructor is `DynamicSortableList`)
  - Instead of implementing the `List` interface, we implement the `SortableList` interface (use generics!)
  - You need to also implement the one extra method (which we will discuss in the next subsection) in `SortableList`:

    ```
    public int Sort() {
        return 0;//dummy value, end of page 16 shows what to return
    }
    ```
  - IntelliJ will go crazy because of the `iterator()` method
- To fix the `iterator()` method, you need to create a new class named `DynamicSortableListIterator`
  - Copy/paste the code found in `DynamicListIterator` into it and change the class/constructor name
  - Change the list type from `DynamicList<E>` to `DynamicListIterator<E>` (in the instance variable and parameter in the constructor)
- Go back to the `DynamicSortableList` class to fix the `iterator()` method because you now have a class `DynamicSortableListIterator` that has an iterator that traverses a linkedlist of type `DynamicSortableList` (don't forget to use generics)
- Now, your code should work fine without issues
- To summarize:
  - An new interface `SortableList` which extends the `List` interface with an extra method named `Sort()` is created
  - A `DynamicSortableList` class is created similar to the `DynamicList` class which implements the `SortableList` interface and contains the `Sort` method
  - A `DynamicSortableListIterator` class is created similar to the `DynamicListIterator` which iterates through a list of type `DynamicSortableList`
  - Everything works well except the hardcoded logic of `Sort` which returns `0`

4.1 The `Sort` method

The `Sort` method will use Bubble sort (also known as Exchange sort) algorithm to sort the linkedlist in an ascending order so that the smallest value will be the left node (and `head`

will point to) and the far-right node the largest value (which then points to `null`). After the sorting completes, the item of the node `cursor` is pointing to will probably change but that is fine. Keep the cursor where it is (*i.e.,* don't play with it). Here is a modified version of the code found on Week 12 slideshow slide 39:

```
private int[] sort(int[] list) {
    int n = list.length;//one-based
    for (int i = n - 1; i >= 1; i--) {
        for (int j = 0; j < i; j++) {
            if (list[j + 1] < list[j]) {//swap values in indices j and j + 1
                int temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
            }
        }
    }
    return list;
}
```

Let us explain this logic:

- `n - 1` because indices are zero-based
- `i >= 1` because the left-most element not required in the first loop
- Inner loop starts from `0` to last array index and then places the largest value at the far-right index
- That far-right cell will not be altered anymore, that is why every time we loop back up, `i` becomes smaller and smaller. Hence, `j` will not span the entire array because `j < i`
- The right side will be sorted until `j` is just a single element which will be sorted by default (that is why `i` doesn't visit the far-left element because `j < i` will be `j < 0` and `j` starts at `0`). You can have `i >= 0` but not required and not wrong either.

4.1.1 Sorting in the Assignment

You can use the same code provided above but no arrays/array indices will be used. Furthermore, the swap statement will similar to what is found in the by using `compareTo` method. Have a look at the original `if` statement from Week 12:

```
if ( list[j+1].getName().compareTo(list[j].getName()) < 0 ) {
    ...
}
```

We will *not* use indices nor `getName` and the inequality operator (*i.e.,* `<`) might or might not be changed. Keep the value `0` at the very end of the line as it is. There are two ways to do this. The first way is easier and the second is more involved.

- Understand that linkedlists don't have indices. However, we can *mimic* the idea of indices. Assume we have `n` nodes in the linkedlist. Let us use the convention that the far-left node is at index `0` and the far-right node (which points to `null`) is at index `n-1` (zero-based), like so:

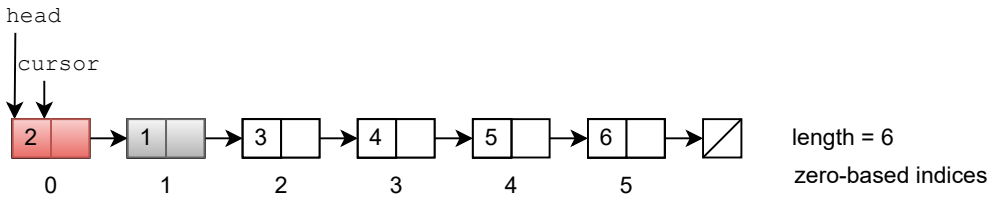

Figure 3: Almost sorted linkedlist with indices. The indices *don't* exist in the program but we are able to create a logic mimicking as if they are there. The colours of the nodes will be discussed in the next figures.

- This will *not* change the nodes but only the `items` inside the nodes
- Is there a way for you to get the element in the `x`th (zero-based) node? For example, let us say you want to get the element `2` (in index `3`), **without** using `cursor`, is there a way to start at the `head`, traverse `x` amount of times (including `x` or not?) and get the item of the node in the specified index? Maybe put all of this into a method? And the method returns the item in that index?
- That way, we can compare one element versus another without the need to worry about linkedlists and `Nodes`. Link that to the modified Bubble sort found above.

- The second approach also uses indices but instead of just swapping the `items` inside the nodes, it will swap the entire nodes. It requires to keep two extra pointers to `j - 1`th and `j + 2`th (four in total). That way:
  - `j - 1`th points to `j + 1`th node
  - `j + 1`th points to `j`th node
  - `j`th points to `j + 2`th node
  - Worry about edge case (if `j` is `0`)

Both approaches will sort the linkedlist but pay attention to the colours of the nodes:

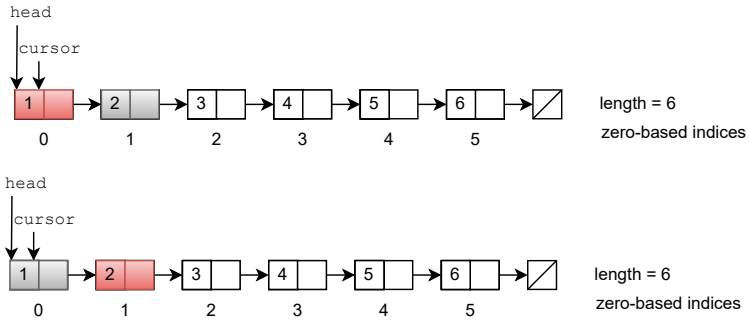

Figure 4: Sorted linkedlist with indices. The top linkedlist (first approach) keeps the nodes the same but swaps elements whereas the second approach swaps entire nodes.

Every time you **swap**, increment some counter in the method and this will denote the time units the assignment was talking about. Return that counter (should be random each run).

4.2 Testing Part B

Lastly, create a class named `TestSorting` and following the instructions in the assignment to test your sorting using a for each loop (it should include the `main` method).