COMPUTER SCIENCE 21A (FALL, 2020)
DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 1 – TEXT EDITOR

Overview:

You have been tasked with writing a simple text editor. Your text editor will allow the user to create, read, and edit text files.

Basic Idea:

Your text editor will be implemented using a modified doubly linked list. In lecture, you have been taught how to traverse, insert into, and delete from doubly linked lists. For this assignment, you will be applying this knowledge but also maintaining an internal "cursor" in your data structure. The cursor is the vertical bar that appears when you are typing.

Implementation Details:

The provided zip file on LATTE contains 9 Java files in 2 packages:

1. `main: Node.java` : this file contains the implementation of a doubly linked list node. Like in lecture, this class has 2 fields "next" and "prev". However, instead of holding integer data, this class holds a character. *Do not edit this file.*

2. `main: Editor.java` : this file will contain the implementation of your text editor. It will contain a variety of methods used for reading, inserting into, deleting from, and tracking the cursor in a text file. *This is one of two classes in the* `main` *package you need to write.*

3. `main: EditorDisplay.java` : this file contains the implementation of a simple graphical user interface (GUI) for you to view your `Editor` class working. *Do not edit this file. (more on page 8)*

4. `main: EditorMain.java` : this file starts the GUI version of your `Editor`. You will find it has 3 commented lines of code. One will start your `Editor` with no input file (line 9). The second will start it with a file containing a single line of text (line 13). The third will start it with a file containing multiple lines of text (line 17). *(more on page 8)*

5. `main: Stack.java:` this file will contain the implementation of a stack. It will be used to store saved versions of the text in your `Editor`. *This is the other class in the* `main` *package that you need to write.*

6. `test: ExampleEditorTests.java` : this file contains basic tests of various capabilities of the text editor. They use and expand on the examples presented in sections I-V of the implementation details of `Editor.java` on pages 2-8. *Do not edit this file.*

7. `test: StudentEditorTests.java` : this file is where you should write your own JUnit tests of the various methods of the `Editor` class. The provided tests do not test edge cases or more advanced functionality. *You should be testing your class thoroughly, and you must write tests for every method that you write.*

8. `test: ExampleStackTests.java:` this file contains basic tests of the methods in your stack data structure. *Do not edit this file.*

9. `test: StudentStackTests.java:` this file is where you should write your own Junit tests for the `Stack` class. The provided tests do not test edge cases or advanced functionality. *You should be testing your class thoroughly, and you must write tests for every method that you write.*

Notes:

- The methods that you will have to implement in `Editor.java` are described in detail below up to page 9.
- Furthermore, there are **six public** fields that you will find in `Editor.java`. *You must use **all** these fields in your implementation.*
- The methods you must implement in `Stack.java` are described on page 9.
- The skeleton also includes 2 text files that are used by lines 11 and 14 of `EditorMain.java`.

Editor.java

Your text editor will have to support a variety of functionality. You have been provided with six instance fields and a variety of functions in the skeleton code, but here the implementation of the various capabilities of the editor will be explained one at a time.

Note: The two constructors for this class are listed first in the skeleton as per Java convention. However, they will be discussed after most of the other required methods.

You cannot use any Java-provided data structures in your implementation.
You cannot use any Java-provided libraries unless they are specified below.

I. The Cursor

Suppose that your text editor contains the string “blue”. In your editor, what are the possible locations of the cursor?

1. Before ‘b’
2. After ‘b’ or before ‘l’
3. After ‘l’ or before ‘u’
4. After ‘u’ or before ‘e’
5. After ‘e’

In general, if your text editor contains a string of **n** characters, the cursor can be placed in **n + 1** possible locations.

For the text editor you will be implementing, if a cursor is at position **k,** then there are **k** characters in the editor’s string before the cursor. Let us consider the example string “blue” and the 5 locations presented above.

1. Before ‘b’. There are *0* characters before the cursor. Hence, the cursor would be at position *0*.
2. After ‘b’ or before ‘l’. There are *1* characters (‘b’) before the cursor. Hence, the cursor would be at position *1*.
3. After ‘l’ or before ‘u’. There are *2* characters (‘b’, ‘l’) before the cursor. Hence, the cursor would be at position *2*.
4. After ‘u’ or before ‘e’. There are *3* characters (‘b’, ‘l’, ‘u’) before the cursor. Hence, the cursor would be at position *3*.
5. After ‘e’. There are *4* characters (‘b’, ‘l’, ‘u’, ‘e’) before the cursor. Hence, the cursor would be at position *4*.

In general, if your text editor contains a string of **n** characters, the cursor’s possible positions are 0, 1, 2, …, n.

This provides motivation for two of the fields you need to use in this class as well as for the implementation of two of the methods you are required to write.

`public int numChars` – this field will maintain a count of the number of characters currently stored in the editor.

`public int curPos` – this field will track the index of the cursor in the editor. It will take on values in the interval [0, numChars].

`public int size()` – this method returns the number of characters stored in the editor.

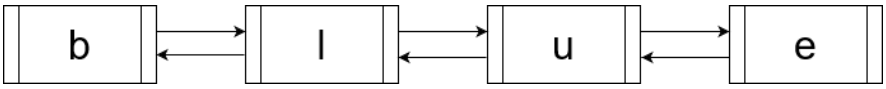
`public int getCursorPosition()` – this method returns the index of the cursor in the editor.

II. Cursor Movement & Default Constructor

At this point, you have been introduced to how the cursor will be tracked and where it will be placed in your editor’s string. In sections III and IV, you will see how insertion and deletion will be done relative to the cursor’s position. However, you will first need to implement the

capability of moving the cursor's current position. This will introduce you to how the doubly linked list will be used as the underlying implementation of the editor.

Returning to our example string "blue" stored in the editor, it will be stored as a doubly linked list of four characters: 'b', 'l', 'u', 'e' with one character per node.



In your editor, you will be using the field `cur` to track the node located *after the current cursor position* or *null*. Consider the 5 cursor locations described in section I for the string "blue":

1. Before 'b'.
    a. `curPos` = 0
    b. `cur` holds a reference to the node containing 'b'
2. After 'b' or before 'l'
    a. `curPos` = 1
    b. `cur` holds a reference to the node containing 'l'
3. After 'l' or before 'u'
    a. `curPos` = 2
    b. `cur` holds a reference to the node containing 'u'
4. After 'u' or before 'e'
    a. `curPos` = 3
    b. `cur` holds a reference to the node containing 'e'
5. After 'e'
    a. `curPos` = 4
    b. `cur` is *null*

In the 5th cursor location, `cur` is null to indicate that the cursor is after the fourth character in the string as opposed to being before it. This distinction will make more sense in sections III and IV.

The user will be able to move the cursor in 4 ways:

1. Moving the cursor right by one character. For instance, if `curPos` = 0 and `cur` references the node containing 'b'. Moving it right should result in `curPos` being 1 and `cur` referencing the node containing 'l'.

2. Moving the cursor left by one character. For instance, if `curPos` = 3 and `cur` references the node containing 'e'. Moving it left should result in `curPos` being 2 and `cur` referencing the node containing 'u'.

3. Moving the cursor to be before the first character. For instance, if `curPos` = 2 and `cur` references the node containing 'u'. This move should result in `curPos` being 0 and `cur` referencing the node containing 'b'.

4. Moving the cursor to be after the last character. For instance, if `curPos` = 2 and `cur` references the node containing 'u'. This move should result in `curPos` being 4 and `cur` being *null*.

This provides the motivation for the existence of the three `Node` variables in the `Editor` as well as the implementation of four of the methods you are required to write.

`public Node head` – this field will hold a reference to the head of the doubly linked list that is used in the underlying implementation of your text editor.

`public Node tail` – this field will hold a reference to the tail of the doubly linked list that is used in the underlying implementation of your text editor.

`public Node cur` – this field will hold a reference to the node that is after the current cursor index or *null* when `curPos` = `numChars`.

`public void moveRight()` – this method should move the cursor one character to the right. As shown in the above example for the string "blue", this should update both `cur` and `curPos`. This method should not throw an exception if the user tries to move right when `curPos` = `numChars`. The movement should be ignored.

`public void moveLeft()` – this method should move the cursor one character to the left. As shown in the above example for the string "blue", this should update both `cur` and `curPos`. This method should not throw an exception if the user tries to move left when `curPos` = 0. The movement should be ignored.

`public void moveToHead()` – this method should move the cursor to the first character in the editor (i.e. to `curPos` = 0).

`public void moveToTail()` – this method should move the cursor to the end of the string in the text editor (i.e. to `curPos` = `numChars`).

Note that `curPos` *should remain in the interval [0,* `numChars`*] when any of the 4 methods above are called.*

Since you are now familiar with all 6 fields you will need to implement this class, you can write the default constructor which should initialize the `Editor`'s fields to signify that it has no characters in it.

`public Editor()` – this method constructs an empty `Editor`.

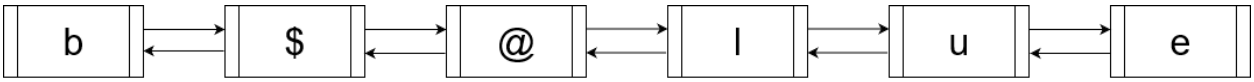
Make sure you initialize all your fields in the constructor.

III. Insertion

One reason why a linked list should be used over an array in the implementation of this text editor is because the user should be able to insert characters at any cursor index in the editor's string. If one were to use an array, a great deal of shifting would have to occur every time a character is inserted. *For this text editor, insertions will always be done **before** the* `Node` *referenced by* `cur`.

Consider the example string "blue" from section I. If one wanted to insert '$' after 'b' then `cur` would have to reference the `Node` with data 'l' and `curPos` = 1. The resulting doubly linked list should then be:

However, there is still the question, what should the values of `cur` and `curPos` be after the insertion of '$'? Suppose we were to then insert the character '@'. We would want it to follow '$' in the resulting string. That, is the new doubly linked list would be:



Since insertions must occur **before** the `Node` referenced by `cur`, this would mean `cur` would still reference the `Node` with data 'l'. However, the `Node` with data 'l' now has 2 nodes ('b', '$') before it. Therefore, the cursor index is no longer 1. It is 2.

With `cur` remaining referencing the `Node` with 'l' and `curPos` has been properly updated to 2, we can insert '@' into the doubly linked list before `cur`. This insertion will cause `curPos` to be incremented again as `cur` now has 3 nodes before it. Therefore, following the insertions of '$' and '@' into the list `cur` remains referencing the `Node` with 'l' and `curPos` = 3.

This should allow you to implement the required method `insert` below:

`public void insert(char c)` – this should insert the provided character before `cur` into the doubly linked list. This includes insertion after the last `Node` in the list.

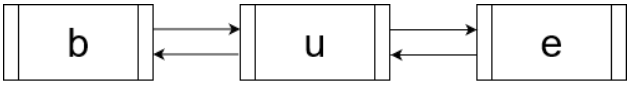
Insertions should result in the fields `head`, `tail`, *and* `numChars` *described in section II should be updated appropriately in addition to* `curPos`.

IV. Delete and Backspace

Another reason that a linked list should be used in place of an array in the implementation of a text editor is because the user should be able to remove characters from any cursor index in the editor's string. If one were to use an array, a great deal of shifting would have to occur every time a character is removed. For this text editor, removal can be done in two ways via delete and

backspace. *Using delete will remove the* `Node` *referenced by* `cur` *from the doubly linked list.* On the other hand, *using backspace will remove the* `Node` *before the* `Node` *referenced by* `cur` *from the doubly linked list*.

Suppose that we wanted to remove the character 'l' from the example string "blue" in section I. This could be done using delete or backspace. To delete 'l', `cur` should reference the `Node` containing 'l' and `curPos` = 1 to remove 'l' using the 'delete' key on a keyboard. This is because the cursor would visually be between 'b' and 'l' at cursor index 1. The resulting doubly linked list following the deletion of 'l' should be:



Consider this as a real-world text editor, the visual cursor should still be between 'b' and 'u'. This would mean a future delete would remove 'u'. Therefore, following the deletion of 'l', `cur` should reference the `Node` containing 'u'. However, the cursor still only has 1 character 'b' before it. Therefore, `curPos` should remain as 1. A following deletion of 'u' would result in `cur` referencing the `Node` with 'e' and `curPos` remaining as 1.
One can also remove 'l' using backspace. To backspace 'l', `cur` should reference the `Node` containing 'u' and `curPos` = 2 to remove 'l' using the 'backspace' key on a keyboard. This is because the cursor would visually be between 'l' and 'u' at cursor index 2. The resulting doubly linked list following the backspace of 'l' is the same as the list shown above.

Following the backspace of 'l', the visual cursor should still be between 'b' and 'u'. This would mean a future backspace would remove 'b'. Therefore, following the backspace of 'l', `cur` should remain referencing the `Node` containing 'u'. However, the cursor now has only one `Node` before it (with data 'b'). Before, when the cursor was after 'l', there were two `Nodes` ('b', 'l') before the cursor. Hence, `curPos` must be decremented from 2 to 1. A following backspace of 'b' would result in `cur` still referencing the `Node` with 'u' and `curPos` decremented to 0.

This should allow you to implement the two required methods:

`public void delete()` – this method should remove the `Node` referenced by `cur`. If the user calls `delete()` when the cursor has no following characters, you should not throw an exception. The movement should be ignored.

`public void backspace()` – this method should remove the `Node` before the `Node` referenced by `cur`. If the user calls `backspace()` when the cursor has no preceding characters, you should not throw an exception. The movement should be ignored.

Removals via `delete()` *or* `backspace()` *should result in the fields* `head`, `tail`, *and* `numChars` *described in section II should be updated appropriately in addition to* `cur` *and* `curPos`.

V. Second Constructor & Remaining Methods

The text editor has two modes. The first is where the editor has no characters in it. The constructor relating to this is the default constructor which was presented in section II. The second mode is where the editor can be loaded with characters from a text file. Thus, you are required to implement the following constructor:

`public Editor(String filepath)` – this should construct an `Editor` with the characters from a text `file` at the provided file path. To do this, you will be required to iterate over the text file character by character and insert them into the `Editor`. To do this, you will be required to use: `java.util.Scanner` and `java.io.File`.

Notes:

- You will find that the method (in the skeleton) has already been declared with a `throws` clause which you should leave there. *You do not need to use try-catch.*
- The cursor should be after the *last character* in the input file (**not on a new line**) after this constructor executes.

There are five additional methods that you are required to implement for this class:

`public String toString()` – this method should return a `String` concatenation of all the characters stored in the text editor. If the `Editor` contains the characters 'b', 'l', 'u', and 'e' then `toString()` should return "blue".

`public void clear()` – this method should remove all of the characters stored in the text editor. *Hint:* This can be done in O(1) time, but make sure that you update *all* relevant fields.

`public void export(String savepath)` – this method should export the contents of the text editor to a file at the provided save path. To do this, you will be required to use: `java.io.PrintStream` and `java.io.File`

Note: you will find that the method (in the skeleton) has already been declared with a `throws` clause which you should leave there. *You do not need to use try-catch.*

`public void save()` – this method should save the string of characters currently in the text editor by pushing the string onto the `savedVersions` Stack. You will implement the Stack from scratch in the `Stack.java` file, which is described in detail below.

A new string should only be pushed onto the Stack if it is different in some way from the most recently saved version. If the current contents of the editor are "abc" and `save()` is called 5 times in a row, with no changes made to the contents of the editor, then "abc" should NOT be pushed onto the Stack 5 times. It should be pushed only once. If the final character 'c' is deleted from the editor, resulting in "ab", and then `save()` is called, "ab" would be pushed onto the

Stack. If ‘c’ is then added, resulting in “abc” and `save()` is called, “abc” can be pushed onto the Stack again, because it is different from the most recently saved version.

`public void undo()` – this method should revert the contents of the text editor to the most recently saved version. The cursor should be after the last character in the `Editor` after the contents have been updated with the most recently saved version. *If there are no saved versions, this method should not throw an exception, it should just be ignored.*

Stack.java

The stack has one field already created for you called `stack`. It is a generic Array that will hold the elements in your Stack. You may not use any other data structure to implement your stack. There is one line of code (commented out) in the constructor, which will help you initialize a generic array.

You may add any fields that you find necessary for the implementation of your Stack. Be sure to comment clearly, explaining what they do.

Required Methods:

`public void push(T x)` – Pushes element x onto the Stack. If the internal Array that holds the elements in your stack is filled to capacity and `push(T x)` is called, the internal Array must be resized (made larger) to create enough capacity to accommodate the new element.

`public T pop()` – Removes the top element from the Stack, and returns it. If pop is called when there are no elements in the Stack, you should throw an `IllegalStateException`.

`public T top()` – Returns the element that is at the top of the Stack. If there are no elements in the Stack, you should return `null`.

`public int size()` – Returns the number of elements in the Stack.

`public boolean isEmpty()` – Returns a boolean corresponding to whether or not this Stack is empty.

`public String toString()` – Returns a string representation of the contents of this Stack. The string representation should contain all elements inside of the stack, with each element on its own line. There should be a single new line after the final element in the stack. There should be no additional/extraneous whitespace of any kind added to or between elements of the stack. For example if we pushed “foo”, “bar” onto the stack, then the string representation would look like:

bar
foo

or, “bar\nfoo\n”.

There are tests for this method included in the ExampleStackTests.java file. It is extremely important that you have the correct formatting for the toString() method, so make sure you are passing the toString() tests.

Using the GUI

While we recommend that you use JUnit tests to thoroughly test your code, you can utilize the provided GUI to see your text editor in action. To run the editor on an empty input file, uncomment line 9 of `EditorMain.java`. A resizable GUI window will then open that uses your `Editor` in the background.

- To insert characters (including new lines), just type the desired characters.
- In the previous pages, 'delete' means to remove the character *after* the cursor.
    - On Windows, use the 'delete' key on the keyboard.
    - On Mac, use Ctrl + D.
- Furthermore, 'backspace' means to remove the character *before* the cursor.
    - On Windows, use the 'backspace' key on the keyboard.
    - On Mac, use the 'delete' key.
- To move left and right, use the left and right arrow keys.
- To jump to the beginning of the text, use Ctrl + H.
- To jump to the end of the text, use Ctrl + E.
- To clear the contents of the editor, use Ctrl + K.
- To export the contents of the editor, use Ctrl + S. Then, you can type the name of the save file in the text area below the text displayed on the window and press enter.
    - If you do not provide a full system file path, the file will be in the same folder as your Eclipse project.
    - If you close the window without saving, your changes will be lost.
    - If you provide the name of an existing file, it will be overwritten with the current contents of the editor.
- To save the contents of the editor so that they can be restored, click on the Save button.
- To restore the most recently saved version of the contents of the editor, click on the Undo button.

These features will allow you to test most of the methods of the editor class. Furthermore, you can uncomment lines 13 or 17 of `EditorMain.java` to create the GUI with some text loaded from an input text file and test the second `Editor` constructor.

IMPORTANT NOTES

- For the classes in which specific fields are included, **you must be using those fields and using the original visibility modifier.**

- Make sure that you have implemented **all of the methods** listed on pages 2 through 8 in `Editor.java`, and all methods on page 9 in `Stack.java`.
- You should use the provided JUnit tests in `ExampleEditorTests.java` and `ExampleStackTests.java` to test your code based on the examples on this document and to ensure you understand the basic functionality of the `Editor` and `Stack`.
- However, you need to write more rigorous tests considering more advanced operations and edge cases of the various methods in the `Editor` and `Stack` classes in the empty files `StudentEditorTests.java` and `StudentStackTests.java`. *Hint:* some of those edge cases have been mentioned in the implementation details of the `Editor` and `Stack` classes.
- You must write JUnit tests to test **every method** that you write.
- **You cannot use any Java provided data structures.**

Submission Details

- For every non-test method you write, you must provide a **proper JavaDoc** comment using `@param`, `@return`, etc.
- In addition to regular JavaDoc information, please include each **non-test method's runtime.**
- At the top of **every file** that you submit please leave a comment with the following format. Make sure to include **each of these 6 items.**

```
/**
 * <A description of the class>
 * Known Bugs: <Explanation of known bugs or "None">
 *
 * @author Firstname Lastname
 * <your Brandeis email>
 * <Month Date, Year>
 * COSI 21A PA1
 */

>Start of your class here<
```

- Use the provided skeleton Eclipse project.
- Submit your assignment via Latte as a zipped Eclipse project by the due date.
- Your Eclipse project should be named **LastnameFirstname-PA1**. To rename a project in Eclipse, right-click on it and choose Refactor > Rename.
- Name your .zip file **LastnameFirstname-PA1.zip**
- Late submissions will not receive credit.