

---

## CSI2110 - Assignment 2 (100 points, weight 10%)

---

*The assignment must be uploaded on Brightspace by **Monday June 14 at 10:00 pm**. Late assignments are accepted for a maximum of 24 hours and they will receive a 30% penalty.*

### Part 1 - Short Exercises (45 points)

#### Instructions

- Download the Assignment2.zip file from the course web site and unzip it; this will produce two eclipse ready projects: **Part1** and **Part2**
- Import the **Part1** project into eclipse
- The project contains the following packages
  - **net.datastructures**: contains the implementation of the `net.datastructures.LinkedList` and `net.datastructures.ArrayStack`
  - **csi2110.assignment2**: contains the `csi2110.assignment2.ShortExercises` main class

#### Exercise 1 (15 points)

Change the implementations of the `preOrder` and `inOrder` traversals methods found in the `csi2110.assignment2.ShortExercises` class so that they are no longer recursive. Use the `net.datastructures.ArrayStack` as an auxiliary structure in order to achieve that purpose.

#### Exercise 2 (15 points)

Implement a queue using the `java.util.ArrayList` as an auxiliary structure. The following directives must be respected:

- The queue should be called `ArrayListQueue` and be placed in the `net.datastructures` package
- The queue should implement the interface `net.datastructures.Queue<E>`

### **Exercise 3 (15 points)**

In level order traversal, all the nodes at a level are visited before passing to the next level. The operation starts at the level 0 where the root is visited, and then proceeds to level 1, where the all nodes (children of the root) are visited from left to right, and then proceeds to level 2 where the same process repeats... Therefore, a level order traversal of the tree shown in figure 1 would result in the following sequence of traversal: A, B, C, D, E, F and G.

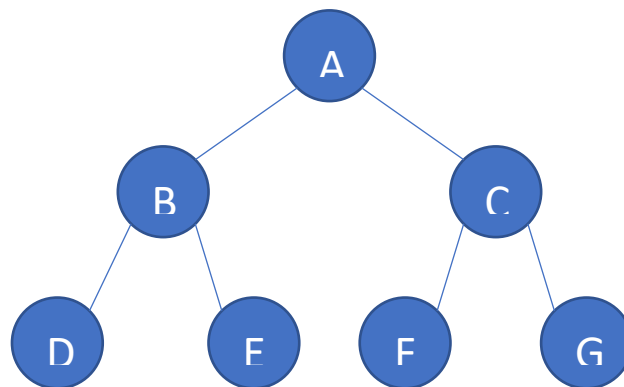


Figure 1: Tree Used to Demonstrate the Level Order Traversal Concept

Use the `net.datastructures.ArrayListQueue` implemented in Exercise 2 as an auxiliary structure in order to complete the implementation of the `levelOrder` method in the `csi2110.assignment2.ShortExecises` class.

## Part 2 - Mini-Project (55 points)

### Background

You have been asked to design and develop a simple Text Editor in Java that allows the user to perform the following functions:

1. Enter textual information into the Text Editor
2. Undo actions performed
3. Open pre-written text files
4. Invoke a spell check operation to verify the correctness of the words in the text document

### Implementation

To implement the spell check operation, a programmer made use of a file that contains the most commonly used English words listed in **alphabetical order**. We will refer to this file as the English Words List (EWL). In his program, he performs the following two steps:

1. Opens the EWL file, reads its content, and stores all the words in a **Node List** structure
2. Scan the text editor and for each word written there, checks if it can be found in the **Node List**. If it cannot be found, then the word is judged to be incorrectly spelled.

To implement the undo operation, the programmer made use of a stack structure to store all user actions. Whenever undo is called, the last user action is popped, examined and its effects are reversed. For instance, if the last user action was adding a letter, the undo operation would delete that same letter.

### Current Issues

The programmer has submitted his implementation to you and to his disappointment, you raised the following issues:


1. The performance of the spell check operation is not up to your expectations. It is too slow. Most of the time is wasted traversing the **Node List** to look for entries.
2. Number of allowable “undos” should have a specific limit, for example maximum ten consecutive undos are allowed, in order to limit the amount of previous actions saved in memory.

The programmer quickly realized why he is facing these issues:

- Searching for a word in a **Node List** is taking too long (worst case:  $O(n)$  word comparisons). He needs to make use of a structure and algorithm(s) that allows him to implement a better search mechanism (worst case:  $O(\log_2(n))$  word comparisons).
- His stack needs to be implemented in a way where its size does not grow beyond a preset limit. Once the size of the stack reaches this limit, it starts to leak and therefore dropping the oldest entries in the structure. He therefore needs a **leaky stack**!

## Your Task

You are going to help the programmer. You will be responsible for the following tasks:

- **Exercise 1 (35 points)** - Employ an **alternative structure** to hold the words instead of the **Node List** (`textEditor.core.structures.NodeList`). **Note that the words are already ordered alphabetically in the EWL file.** Your new structure (which we have already seen in class) should allow the search mechanism to be optimized (refer to the **HINT**  below). ***In order to succeed at this task, you have to:***
  - Implement the new structure **yourself** (you cannot use one of the standard java structures). Note that all of the functions defined in its ADT must be implemented. The new structure should be placed in the `textEditor.core.structures` package.
  - **Integrate an optimized search mechanism** ( $O(\log_2 n)$  complexity) into the new structure. **Assume** that each word comparison (test) is counted as one primitive operation, regardless of the length of the word.
  - Make sure the structure does not have a static size since other words may later be added to the EWL file. The program should be able to accommodate such updates of the EWL file without changing the implementation.
  - Edit the class `textEditor.core.EwlListAdaptor` in order to reference the new structure as opposed to the old one.
- **Exercise 2 (20 points)** - Implement the leaky stack previously discussed using a linked structure. An array based implementation is not acceptable. ***In order to achieve this task, you need to extend the*** `textEditor.core.structures.AbstractStack` ***abstract class.*** Place your new class in the `textEditor.core.structures` package. Also, make sure the `textEditor.gui.TextEditorGui` class references the new stack instead of the old `java.util.Stack`.



**HINT:** Since the words are already ordered alphabetically in the EWL file, reading them into an **array list** would allow you to perform a very effective search on the words using one of the algorithms we have seen in class.

**Side note:** if you use an AVL tree, the numerous rebalancing will cause the load time (time it takes to load the words into the structure) to be larger than that of the array list implementation, think about it...

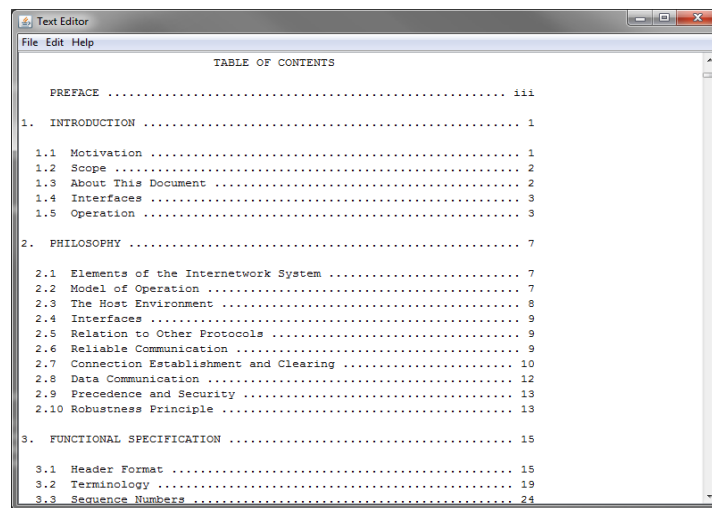
## Instructions

- Download the Assignment2.zip file from the course web site and unzip it; this will produce two eclipse ready projects: **Part1** and **Part2**
- Import the **Part2** project into eclipse
- The contents of the **Part2** directory is organized as follows:
  - **Src folder:** contains all the java source files of the Text Editor program.
  - **Properties folder:** contains the `textEditor.properties` file that configures the Text Editor program.
  - **Config folder:** contains the `ewl.txt` (the file containing a list of the most popular English words) and `about.txt`.
  - **TCP\_RFC.txt file:** contains the RFC of the TCP protocol. This file will be used to test the efficiency of the program.
- Run the Text Editor program (`TextEditorGui` is the main class) (see figure 2)
- Experiment with the Text Editor by testing its spell check and undo functionalities. These two commands can either be called from the menu on the top or using keyboard shortcuts (F7 for spellcheck and `ctrl+z` for undo)
- Open the file `TCP_RFC.txt` contained in the `TextEditor.zip` file. This is the Request for Comment (RFC) for the TCP protocol. Run the spell check command. The TCP RFC is a lengthy file and will give the Text Editor quite a work out. When the spell check operation is finally completed, a list of the incorrectly spelled words will be displayed (see figure 3). Also, the following sentence will be printed in the console: "Time it took to finish spellcheck operation: x ms" (where x is the amount of milliseconds it took to complete the spell check). As you improve your implementation, x should decrease.

## How to Submit

After completing the assignment, make sure you follow these steps for submission:

- Create a zip file out of the Assignment2 directory (should contain the Part1 and Part2 directories)
- Submit the Assignment2.zip on the Brightspace website.
- **Note:** In case your project does not compile, You **MUST** create a readme file that includes the methods that you have implemented and why your project does not compile.



The screenshot shows a 'Text Editor' window with a menu bar (File, Edit, Help) and a document titled 'TABLE OF CONTENTS'. The content is a table of contents for a document, listing sections and their corresponding page numbers.

PREFACE .....	iii
1. INTRODUCTION .....	1
1.1 Motivation .....	1
1.2 Scope .....	2
1.3 About This Document .....	2
1.4 Interfaces .....	3
1.5 Operation .....	3
2. PHILOSOPHY .....	7
2.1 Elements of the Internetwork System .....	7
2.2 Model of Operation .....	7
2.3 The Host Environment .....	8
2.4 Interfaces .....	9
2.5 Relation to Other Protocols .....	9
2.6 Reliable Communication .....	9
2.7 Connection Establishment and Clearing .....	10
2.8 Data Communication .....	12
2.9 Precedence and Security .....	13
2.10 Robustness Principle .....	13
3. FUNCTIONAL SPECIFICATION .....	15
3.1 Header Format .....	15
3.2 Terminology .....	19
3.3 Sequence Numbers .....	24

Figure 2: Text Editor Interface

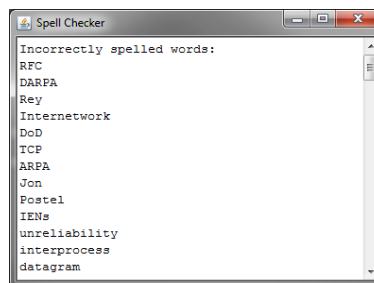


Figure 3: Window Pop-up Displaying Incorrectly Spelled Words



Figure 4: You celebrating when you're done...