COMP 2150 - Winter 2021 - Assignment 1

Due February 11 at 11:59 PM

Typos fixed, January 24: replaced “tutor” with “user” in USER command, removed FAIL option for APPEND command.

In this assignment, you will build the tools to manage a **wiki**. A wiki (like wikipedia.org) is a collection of documents which can be edited by a set of users. Every action that a user takes is logged. Additionally, the wiki should be able to show the history of a document and restore a document to a previous version. The software will be written in Java and driven by a set of text commands.

The **goal of the assignment** is for you to consider how to use OO concepts to complete a larger assignment. Because of this, the assignment is given in terms of behaviour that it should achieve, rather than a how your code should be structured. However, remember that if you don’t use OO concepts in the ways described in this assignment, you can lose marks, even if the assignment works in the way we describe.

Here is an overview of the major components of the assignment, followed by a description of the commands for the system.

Documents

Your wiki should consist of a set of documents. Each document has a title (which is unique among all documents in the wiki) and contents. Each document is a sequence of lines of plain text. When a document is created, it is initially empty.

Users can update the documents in limited ways: they can replace or delete a line (which already exists in the document) or they can add a new line to the end of the document. Each time a document is updated, the change that is made is saved. At any time, the user can ask to see both the most current version of any document and the history of all changes made to a document.

Users

Everyone who wants to edit a document in the wiki must be a registered user. When the user registers, they give a unique userid. After registering for the system, users can create or edit documents. At any time, a user’s history of document changes can be displayed.

Commands

Your program will work by processing a text file that consists of commands. Each command is on its own line in the file. You should read the text files line by line to process command. The name of the text file **must not be hardcoded** into your program -- the user must be prompted for the name of the file (through a text prompt in the program, do not use a command line parameter or a tool like JFileChooser).

In each command, an implicit **time** is specified. In particular, each action is assumed to take place a global time that advances by one for each command. Thus, the first command in the file takes place at time 0, the second command takes place at time 1, etc.

Some commands refer to line numbers in the document. Line numbers are assumed to start at line 0 (the first line in the document). When a document is created, it has no lines in it, so the first line created would be line 0.

1. USER [userid]
   - This action creates a new user.
   - Example: USER mike
   - Outcomes: CONFIRMED, DUPLICATE
   - A user is a duplicate if there is another user with the same userid. Duplicates are ignored after being reported.
   - The userid is a sequence of at most 80 non-whitespace characters (uppercase and lowercase letters and numbers).
2. CREATE [document name] [userid]
   - This action creates a new document with the specified name, which is created by the specified user
   - Example: CREATE Object_Oriented_Programming ali
   - Outcomes: CONFIRMED, DUPLICATE, NOT FOUND
   - A document is a duplicate if has already been created. Duplicates are ignored after being reported.
   - The name of a document is a sequence of at most 80 non-whitespace characters.
   - The command reports not found if the user does not exist in the system. No further processing is done in this case.
   - The created document is initially empty.
3. APPEND [document] [userid] [content]
   - Appends content to a single line at the end of a document.
   - Example: APPEND Object_Oriented_Programming mike OO programming is awesome.
   - Outcomes: NOT FOUND, ~~FAIL~~, SUCCESS
   - An append operation returns not found if the user is not found, or the document has not been created. No further processing occurs in this case.
   - A successful appends adds all the content information (everything after the userid) to a single (new) line at the end of the document. That is, if the document previously had n lines, it now has n+1.
4. REPLACE [document] [userid] [L] [content]
   - Replaces line number L in a document with a new version.
   - Example: REPLACE Object_Oriented_Programming ali 10 Java is an object-oriented programming language.
   - Outcomes: NOT FOUND, FAIL, SUCCESS
   - An append operation returns not found if the user is not found, or the document has not been created. No further processing occurs in this case.
   - An append operation returns fail if there is no line number L in the document
   - A successful replace command replaces the current line L with the content in the command (everything after the line number). All other lines of the document are unchanged.
5. DELETE [document] [userid] [L]
   - Deletes line number L in a document.
   - Example: DELETE Java_Programming_language mike 20
   - Outcomes: NOT FOUND, FAIL, SUCCESS

- A delete operation returns not found if the user is not found, or the document has not been created. No further processing occurs in this case.
- A delete operation returns fail if there is no line number L in the document.
- A successful delete command deletes the current line L in the document. All lines after line L are then numbered one less than their previous number and the document has one less line in total.

6. PRINT [document]
   - Prints the current contents of the document.
   - Example: PRINT Java_programming_language
   - Outcomes: NOT FOUND, print document
   - A print command returns not found if the document has not been created. No further processing occurs in this case.
   - A successful print outputs the title of the document on a line, followed by the contents of the document, with the number of each line before each line.
7. RESTORE [userid] [document] [time]
   - Restores a document to a particular time.
   - Example: RESTORE mike History_of_programming_languages 10
   - Outcomes: NOT FOUND, restore document
   - The command returns not found if the user is not found, the document has not been created, or if the document did not exist at the given time. No further processing occurs in this case.
   - A successful restore prints the contents of a document at a given time. If no edit command (append, restore, delete, replace) occurred to the document at that time, the contents of the document after the last edit before that time should be reported. If an edit occurred at the time request, the contents of the document **after** that edit should be reported.
8. HISTORY [document]
   - Prints the history of the document.
   - Example: HISTORY History_of_programming_languages
   - Outcomes: NOT FOUND, print history
   - A history command returns not found if the document has not been created. No further processing occurs in this case.
   - A successful history outputs all edits that have occurred for a document. This includes the document creation, and all replace, delete, restore and append commands. For each of these commands, the user who performed the operation should be reported, along with the type of edit and all relevant information (line numbers, old version in case of replace). The commands should be output in order from newest to oldest.
   - **Note**: If a restore command was issued, the document should be restored to the version at that time, but there is no change to the history – all edits that were undone still appear in the history. Further, the restore command also adds an entry to the history.
9. USERREPORT [userid]
   - Prints a list of all edits made by a user.
   - Example: USERREPORT ali
   - Outcomes: NOT FOUND, print report
   - A user report command returns not found if the user does not exist. No further processing occurs in this case.
   - A successful user report command outputs the user id and then all edits that the user has made. This includes creation, append, delete, restore and replace commands. The commands should be output in order from oldest to newest.

- If an edit was later undone by a restore command, the command should still be reported in the list of edits for the user.

10. QUIT
    - Ends the program. No other commands are read after this command.
    - The message "BYE" should be printed and the program should terminate.
    - If the end of the program is encountered without a "QUIT" command, the program should report that the "QUIT" command was missing and then terminate.
11. COMMENTS
    - Any line that starts with the character # is a comment.
    - The comment is ignored by the system.
    - Please note: **students have failed assignments in previous years because the assignments couldn't process comments**. Please don't forget about comments.

In all commands, whitespace (tabs and spaces) is ignored (i.e., there can be leading whitespace on any lines, or multiple whitespace between tokens in a line). However, each command is guaranteed to appear on a single line.

For error checking, you should ignore commands that are not one of the previous kinds. You can assume that integers are valid integers in the input and don't need to do error checking to see if an integer is actually an integer (but you do need to check ranges for delete and replace commands).

The format of all output statements is **not strict,** but all prompts and output should provide enough information for the marker to identify correctness quickly. It is suggested that some output be provided for every command in the input file. Questions on the format of output will not be answered – you should simply use your best judgement to create useful, meaningful output for all commands.

Object Oriented Programming

Your assignment should use OO programming. In particular:

1. You should have code reuse as much as possible. If you have duplicated code for the same tasks (i.e., inserting an item into a data structure), you will lose marks.
2. You should design your code to use OO tools (hierarchies and polymorphism) to make code as general purpose as possible.
3. You should have as little static code as possible. You should aim to have **ten lines of code or less in static methods** in your entire assignments. More than this will result in lost marks.

Data Structures

There are some restrictions on data structures you must use for the assignment. **Do not use ANY built-in Java collection classes** (e.g., ArrayLists, Hash tables) **or Java Generics in this assignment**: any linked structures must be your own, built from scratch. You should **not use arrays** other than for temporary operations (e.g., split() for strings).
IF YOU USE GENERICS, ARRAYS OR COLLECTION DATA STRUCTURES, YOU WILL LOSE MARKS.

For this assignment, you do not need to concern yourself with data structure time efficiency. Any searching through data structures can be exhaustive search and you do not to maintain any data structures in sorted order. You also do not need to worry about space efficiency. You can store more information that you need to make

When dealing with hierarchies and data structures, you must use safe casting practices.

Unit Testing

Construct a set of unit tests for your code. To do unit testing in Java, you should:

1. Install junit jar files (either through your IDE or from https://junit.org/junit5/).
2. Create a new class for the tests.
3. Imports: add the following import statements.

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

4. Create public void methods in your class to test conditions about your code.
5. Annotate each test method with the line "@Test" before the method.

Your tests should focus on the data structures you create for your project. You should include at least ten meaningful tests for your code, including a minimum of five for your data structures.

You will be graded on your tests, so write useful tests. The markers will be running your tests, as well, so ensure that they pass. If tests do not pass or your code does not compile, you will lose a substantial number of marks.

Hand-in

Submit all your source code for all classes. Each class should be declared as public in its own file.

You **MUST** submit all your files in a zip file. Additionally, you **MUST** follow these rules:

- You should follow the programming standards given on the course website.
- All the files required for your project should be in a single directory (this probably means that you should **not** have any package statements in your code.)
- Include a README.TXT file that describes **exactly how to compile and run your code** from the command line. The markers will be using these instructions exactly and if your code does not compile directly, you will lose marks. Code will be run on a standard linux environment (**aviary**). Your instructions must include details on how to read the text files.
- You should also submit a text document giving the output on the official test data. (Official test data will not be released until **just before** the due date.)

The easier it is for your assignment to mark, the more marks you are likely to get. Do yourself a favour. Submit all files on umlearn.