CMPS 12B

Introduction to Data Structures

Programming Assignment 2

Objective

The purpose of this assignment is to give you practice implementing doubly-linked lists, writing and running unit tests as you develop your data structure, and using your data structure in a working program.

Task

Your task is to write a very simple line-oriented text editor that uses a doubly-linked list data structure to keep track of lines in the file being edited. The specifications for your editor are presented in the format of a Unix man page.

NAME

edfile - line text editor

SYNOPSIS

edfile [-e] [filename]

DESCRIPTION

The edfile utility reads in lines from a file and stores them in a list. Editing operations make changes to this list. Eventually the lines are written out to a file.

OPTIONS

All options precede all operands.

`-e` Each command is echo printed when it is read in.

OPERANDS

The optional operand is a filename. When the program starts, all the lines from the file given are read into a list in memory. The current line becomes the last line in the list. If no filename is specified, the list starts out empty.

COMMANDS

After the file (if given) is read in, input is read from stdin. Each line of stdin contains one command which is interpreted to apply to the list in various ways. The editor must be able to switch to the first and last line in the list in O(1) time. A reference is kept to the current line, which starts out as the last line read from the file.

Note that there are no spaces between the command letter and any other information in the command. Input lines that are empty or entirely whitespace are ignored.

`#` *anything*

A comment line, the command is ignored.

`$` The current line is set to the last line in the list. The new current line is then displayed.

`*` All the lines in the list are displayed. The current line becomes the last line in the list.

`.` The current line is displayed.

`0` The current line is set to the first line in the list. The new current line is displayed.

`<` The current line is set to the previous line. The new current line is displayed.

`>` The current line is set to the next line. The new current line is displayed.

`a` *text* The new text is inserted as a new line after the current line. The new line just inserted becomes the current line, which is displayed.

`d` The current line is deleted. The next line becomes the current line.

`i` *text* The new text is inserted as a new line before the current line. The new line just inserted becomes the current line, which is displayed.

`r` *filename*

The contents of *filename* are read and all the lines are inserted after the current line. The current line becomes the last line inserted. An error is printed if the file cannot be read. If the operation succeeds, the number of new inserted lines is displayed.

w filename

All of the lines are written to *filename*. On success the number of lines written is displayed. An error message is displayed if the file cannot be created or written.

(eof) Quit the program (recognized when stdin reaches end-of-file).

EXIT STATUS

The following exit status codes are returned:

0 No errors were detected.

1 Invalid commands, invalid options, or file access errors were detected.

Doubly-Linked List

You will be implementing a doubly-linked list in your implementation. Your class should be called dllist. This class keeps a sequence of strings in order by relative position and has special access to the first and last positions in the list, and special access to another position called the current position.

Your class must define the following enum:

```
public enum position {FIRST, PREVIOUS, FOLLOWING, LAST};
```

Your class must implement the following methods:

`public void setPosition (position pos);`

Changes the current position to be one of the places specified by the enum argument. This method is used to move the current position to the first, last, previous, or following string. Attempts to move the current position before the first position or following the last position are silently ignored.

`public boolean isEmpty ();`

Returns true if the list is empty, false otherwise.

`public String getItem ();`

Returns the string at the current position without changing anything in the list. Throws the exception `java.util.NoSuchElementException` if there are no elements in the list.

`public int getPosition ();`

Returns the relative numerical position of the current element in the list. The first position in the list is position 0. Throws `java.util.NoSuchElementException` if there are no elements in the list.

`public void delete ();`

Deletes the string at the current position in the list and makes the following string the current position. If the last string in the list is deleted then the current position becomes the new last string. Throws `java.util.NoSuchElementException` if there are no elements in the list.

`public void insert (String item, position pos);`

Inserts a new string into the list at the specified position. The new item can be inserted as the first element, last element, immediately before the current position, or immediately after the current position. The element just inserted becomes the new current element. Throws `IllegalArgumentException` if the position argument does not make sense for the current string.

Implementation

Your implementation will consist of several source files. Make sure you test each step and that it is working before you attempt the next step.

Template code for this assignment can be found here:
`/afs/cats.ucsc.edu/users/r/nwhitehe/cmps12/asg2/`

The template code includes:

auxlib.java	Auxiliary functions for error reporting and error codes
dllist.java	Template implementation for dllist class
dllistTest.java	Template for dllist unit tests
edfile.java	Template implementation for main application
Makefile	Makefile for project

The template code includes empty implementations for all the required methods of the dllist class. It includes a single unit test that checks that isEmpty() returns true on a newly created dllist object. Note that your final implementation of doubly-linked lists should know nothing about file editing, and your application should be able to use doubly-linked lists without knowing anything about their implementation.

The makefile includes the following phony targets:

all	Builds the final executable
test	Build and run the unit tests
clean	Clean up compiled classes
spotless	Clean up everything including final executable
submit	Submit files for grading

Here are the required steps to completing this assignment:

1. Get the template code, try "make test" and "make all". Examine the code.
2. Start by modifying edfile.java to print a welcome message.
3. Write code to scan the command line argument list and figure out the options and operand. Print out which options were given and what the operand is (if any). This code should be removed for the final version but is useful for debugging and testing.
4. Write a function to read in all the lines of a file. Put in debugging output so you can verify it is processing lines properly.
5. Within edfile, create a new dllist object in the main app.
6. Replace the stub code in edfile.java to call the appropriate method in the linked list class. Ignore the read and write commands at first.
7. Modify your application so that it checks for a filename, if present it then reads all the lines in the file and inserts them into the linked list.
8. Now start working on making your linked list actually work.
9. Write another unit test for empty, for example after one insert at the end of the list empty should return false. Implement the isEmpty() method and verify it passes both unit tests.
10. Write a unit test to verify that insert at the end of the list and getItem work properly for some simple examples. For example, a single insert at the end of the list followed by a getItem should return the item inserted. Also, two inserts at the end of the list followed by a getItem should return the second item inserted.
11. Verify that the unit tests fail (because the methods are not implemented). Implement insert with

position at the end of the list and getItem. Make your unit tests pass. Include as much debugging output as you want.

12. Write more unit tests that test insert putting new items at the front of the list. For example, inserting two elements at the front of the list followed by getItem should return the second item.
13. Implement insert for inserting new items at the front of the list and verify that the unit tests pass.
14. Write unit tests for setPosition where setPosition moves to the beginning or end of the list. You should have a unit test that calls insert two times at the end of the list, then setPosition to the beginning of the list, then call getItem. This should return the first item inserted. Also write a unit test that inserts twice at the beginning of the list, then calls setPosition to move to the end of the list. Then getItem should return the first item inserted.
15. Write code for setPosition to move to the beginning or end of the list and verify it passes the unit tests.
16. Write a unit test that builds up a list using insert. Insert strings "A", "B", and "C" at the end of the list. Then call insert to insert "D" immediately before the current position ("C"). Then move to the end of the list with setPosition, and verify that getItem returns "C". Write a unit test that inserts "A", "B", and "C" at the beginning of the list, the inserts "D" immediately after the current position ("C"). Move to the beginning of the list with setPosition and verify that getItem returns "C".
17. Implement insert for the case where the position is PREVIOUS or FOLLOWING and verify it passes your unit tests.
18. Invent a unit test that uses insert and setPosition to create a list of 5 elements in a random order, then uses setPosition to navigate to each position in the list using PREVIOUS and FOLLOWING and verifies that getItem returns the correct element in each position.
19. Update your implementation code to make the unit test pass.
20. Write a unit test similar to the previous one but that calls getPosition at each position to verify the position is correct instead of verifying the item in that position.
21. Imlement getPosition and verify it passes your unit tests.
22. Come up a with a plan for testing delete. Write unit tests following your plan, alternating writing unit tests and implementing the delete functionality.
23. Write unit tests to cover error cases and verify that your implementation handles error cases properly.
24. Go back to your application and verify that linked lists are working in your application.
25. Implement reading and writing files in your main application.
26. Test your application on a text file to find bugs. Fix any bugs you find.
27. Tidy up your application and linked list implementation to not print out debugging information.
28. Submit (actually you should do this as you complete milestones throughout the assignment).

What to Submit

All files you turn in for every assignment and lab should begin with a comment block that includes your name, CruzID, class, date, filename, short description of the file's role in the assignment, and any special instructions related to the file. Also create a file called README. The README file should have the normal comment block, then list all the files being submitted (including itself) along with any special notes to the graders.

Submit the following files:

README	Contains your name, username, optional brief note to grader
auxlib.java	Auxiliary functions for error reporting and error codes
dllist.java	Implementation for dllist class
dllistTest.java	Unit tests for dllist
edfile.java	Main application
Makefile	Makefile for project

To submit, use the submit command.

```
submit cmps12b-nojw.f14 asg2 files
```

Credit: This assignment was created by Wesley Mackey for CMPS 12B and adapted to use unit testing.