$Id: asg2j-edfile-dllist.mm,v 1.6 2010-10-19 17:15:26-07 - - $
/afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg2j-edfile-dllist

## 1. Overview

This assignment will make use of doubly-linked lists in order to implement a very simple line editor after the style of **ed**(1). Data Structures goals: Experience with linked lists and pointers (references), and file handling.

## 2. Program Specification

**NAME**

edfile – list text editor

**SYNOPSIS**

**edfile** [**-e**] [*filename*...]

**DESCRIPTION**

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

**OPTIONS**

Options may appear in any order and may appear as separate words on the command line or concatenated together. All options precede all operands.

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

**OPERANDS**

All operands are filenames. When the program begins, all of the lines from all of the filenames given on the command line are read in sequence and inserted into the list. The current line becomes the last line in the list. If there are no filenames given, the list is initially empty.

**COMMANDS**

After all of the files (if any) have been read in, **stdin** is read. Each line of **stdin** contains one command which is applied to the list in various ways. At end of file, the program stops. The editor can always get to the first line and last line in the file in $O(1)$ time. It also maintains a pointer to the current line. Note that there are ***no*** spaces between the command letter and its operand when an operand is permitted. Empty lines and lines consisting only of white space are ignored.

\# *anything*
: Indicates a comment line. The command is ignored. Also, any empty line, or any line consisting only of white space, is ignored.

**$** The current line is set to the last line in the list. The new current line is then printed.

***** All of the lines in the list are printed. The current line becomes the last line in the list.

**.** The current line is printed.

**0** The current line is set to the first line in the list. The new current line is then printed.

**<** The current line is set to the previous line. The new current line is then printed.

**>** The current line is set to the following line. The new current line is then printed.

**a** *inputline*
: The text following the letter ***a*** is inserted ***after*** the current line. The line just inserted becomes the current line, which is then printed.

**d** The current line in the list is deleted. The next line becomes the current line, if any. Otherwise the last line becomes the current line.

**i** *inputline*
: The text following the letter **i** is inserted ***before*** the current line. The line just inserted becomes the current line, which is then printed.

`r` *filename*
: The contents of the specified file are read in and inserted ***after*** the current line. The current line becomes the last line inserted. An error message is printed if the file can not be accessed. If the operation succeeded, the number of lines read in is printed.

`w` *filename*
: All of the lines in the file are written to the specified file. The number of lines written is printed. An error message is printed if the file can not be created.

*(eof)* At end of file, quit the program. This is not a command: it is recognized via EOF on `stdin`.

**EXIT STATUS**

The following exit status codes are returned:

0 No errors were detected.

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

### 3. The Doubly-Linked List

You will be using the doubly-linked list class `dllist`. The list keeps a sequence of objects in order by relative position and has special access to the first and last positions in the list and also another special position known as the current position. The methods are:

`public void setposition (position pos);`
: Changes the current position to be one of the places specified: the first or last positions in the list, or the node immediately previous to or immediately following the current position. An attempt to move before the first or after the last position is silently ignored. Throws an `IllegalArgumentException` if an invalid position code is given.

`public boolean empty ();`
: Reports on whether the list contains any elements. Initially it does not.

`public String getitem ();`
: Returns the object at the current position without changing anything in the list. Throws `java.util.NoSuchElementException` if there are no elements in the list.

`public int getposition ();`
: Returns the relative numerical position of the current element within the list. The first element is 0. Throws `java.util.NoSuchElementException` if there are no elements in the list.

`public void delete ();`
: Deletes the object at the current position and makes the following object the current object. If it was the last element that was deleted the current position is now the new last element. Throws `java.util.NoSuchElementException` if there are no elements in the list.

`public void insert (String item, position pos);`
: Inserts a new element into the list at the position given. The element can be inserted at the first or last positions within the list or immediately previous to or immediately following the current element. The element just inserted becomes the new current element. Throws an `IllegalArgumentException` if an invalid position code is given.

### 4. Implementation Sequence

This program is large and consists of multiple source files. It is very important that you follow an implementation sequence in order to avoid being overwhelmed by its complexity. Begin with a small subset of the required implementation and perform each step in sequence. Make sure that each step is working before you continue on with the next step.

(1) There is a subdirectory called `test` which contains a Perl program called `edfile.perl`. In order to become familiar with the details of what the assignment requires, play around with this program after reading the `man` page above. Post bug reports to the class newsgroup. Java code is

provided in the subdirectory **code/**. Note that your main class must not know anything whatever about linked lists and your linked list class must not know anything whatever about files, arguments, or user interaction.

(2) Start with a program that prints its own name as in **edfile.java**

(3) A utility class **auxlib**, with useful functions is provided in **auxlib.java** Remember that normal output is written to **stdout** and error messages are written to **stderr**. Also remember that the program has to properly return the system exit code.

(4) Normally the name of a program comes from **argv[0]**, but Java does not have such a concept, so we need to perform a hack. This hack does not work when Java is not kept in a jar. Instead of the prompt printed by the Perl program, use **auxlib.PROGNAME**

(5) Write code to scan the command line argument list and figure out the options and operands. Print a stub message, later to be removed, which states what options were given and what operands were given. A stub is just a print statement in place of a function call later to be added.

(6) Write code to read lines from **stdin** and parse them. Use a **switch** statement to select the type of line based on the first character (use **charAt**), and print error messages for invalid input. Note that some commands need operands and some don't.

(7) Write a function to read in lines from a file and use this function to read in all lines from all files specified on the command line. Print out each line as it is read.

(8) Integrate the **dllist** class into your program. Each public method of **dllist** should simply be added with a stub body which does nothing other than print out its name and arguments. Run **gmake** to verify that you can construct a proper jar.

(9) Modify each of the stubs in your main class so that they call the appropriate functions in the linked list class. Ignore the read and write commands for now.

(10) To the linked list class, add a private static inner class to keep track of the data and four private fields to keep track of the nodes. See **dllist.java** Note that inner classes, when compiled, produce class files with compound names connected with a dollar sign. You need not have make file targets for these, but they must be placed in the jar file as well as the outer classes.

(11) Unfortunately Java uses a dollar sign in these filenames, which is also a shell metacharacter as well as a **gmake** metacharacter. To use a dollar sign (**$**) inside a make file, it must be doubled, and to avoid the shell doing something with it, it must be escaped. So, to refer to the file **dllist$node.class**, you must type **dllist\$$node.class** See the sample **Makefile**, which you should modify as appropriate.

(12) Write the functions **insert** and **getitem**, ignoring the position. Always insert at the end of the list. Instrument your code so that each operation prints a message as to what it is doing. Print out the node and its links. Since **node** does not have a **toString** method, it will inherit it from **Object** and hence just print its type and id. Also write **empty** which is trivial.

(13) Complete **insert** so that it can function at any position in the list. Also write **setposition** In order to check for a valid position code, use a **switch** statement, with the **default** alternative being the one to throw the exception.

(14) Write code to track the relative numeric position of the current element. Write the function **getposition**

(15) Write **delete**.

(16) Go back to your main class and implement the read and write commands.

(17) Finish off anything else that is missing and submit your program. Actually, you should submit a version of your program as it improves with each milestone and check your submit. Note that if you forget to submit a file or submit the wrong version of a file, your program will not run and you will lose 50% of the value of the assignment.

### 5. What to Submit

Submit the files specified below. Do not submit any files that should be created by your `Makefile`. Filenames must be exactly as given here. The first line of every file you submit must state your name and username in a comment. The second line must contain a comment which originially contained the string `$Id$` before that string was edited by RCS.

`README` contains your name and username and a statement of which major parts of the implementation sequence you have completed, along with a ***brief*** note to the grader, if such a note is needed.

`edfile.java` contains your main function, scanning code, and file access code.

`dllist.java` contains the class `dllist` which implements the data structure.

`auxlib.java` has been written for you and contains various useful functions involving error messages, exit codes, etc. You may modify it as appropriate.

`Makefile` builds the following targets:

- `all:` must be the first target, and builds the jar file `edfile`.
- `clean:` deletes all files generated by `make all`
- `spotless:` depends on `clean` and deletes the jar file.
- `submit:` submits all of the files specified above.
- `ci:` checks in all source files, including the `README` and `Makefile`, into an RCS subdirectory. It may assume the script `cid` is in the path.

`PARTNER` If you are doing pair programming, follow the instructions in `Syllabus/pair-programming/`. Do not submit this file if you are working alone.

***IMPORTANT:*** After you have submitted your files, verify that you have submitted all of the necessary files and that they are all the correct version. See the syllabus and assignment 1 for details of how to do this. If you forget to submit a necessary file or submit the wrong version, you will lose many points! The script `testsubmit` can help with this.