$Id: asg3j-xref-bstreeque.mm,v 1.15 2012-11-05 18:04:19-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg3j-xref-bstreeque
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/asg3j-xref-bstreeque

1. Overview

In this assignment, you will implement a cross reference utility that will scan files for words and for each word found, print a list of those words in lexicographic order along with a count of the number of times each occurs and a list of line numbers where the words were found.

This program will make use of a binary search tree data structure, which means that each search and insertion of a word will run in $O(\log_2 n)$ time. As a value in each tree node, there will be an integer and queue. you will use the array implementation of the queue where each operation runs in $O(1)$ time (amortized).

2. Program specification

Once again we present the specification in the form of a Unix **man**(1) page.

NAME

jxref — word cross reference utility

SYNOPSIS

jxref [**-cdf**] [*filename* . . .]

DESCRIPTION

The options word, if present, must precede all operands. Each file is read in sequence and a printout of the words found in the file is generated at the end of each file, one word per line, each followed by a count of the number of times the word appears and a list of line numbers where it appears. All output is to **stdout**, except for error messages, which are printed to **stderr**.

OPTIONS

Options may appear in any order either clustered or as separate words, but all must appear before operands.

-c The cross reference list for each word is suppressed and only the words and counts are printed.

-d Instead of producing normal output, the tree is dumped in debug format, showing each key and value and also the level of each node within the tree.

-f Upper case letters are folded into lower case before insertion into the binary search tree.

OPERANDS

Each operand is a filename, which is processed in sequence, each file causing to be created a new tree. If any filename is specified as a minus sign (**-**), **stdin** is read at that point. If no filenames are specified, **stdin** is read. As an output filename, the minus sign is used as the name of **stdin**.

EXIT STATUS

0 No errors occurred.

1 Errors occurred, either in scanning options or opening files. Messages were printed to **stderr**.

3. A Tour of the code and possible implementation sequence

You have been provided with some source code to start working from. Study this code before you begin. Print out the code listing when you print out this file. Both are important.

(1) The Perl program **pxref.perl** is a reference implementation, which your program should emulate. Study the behavior of this program. Do not submit this program. It is not part of your project.

(2) The class `options` keeps track of the options and arguments to the program. Use its constructor to initialize its various fields. Set the `filenames` field to an array containing the filenames, but not the option word, from `args`. If there are no filenames, initialize that field to an array containging a single string which is a minus sign (`"-"`).

(3) The class `queue` is a linked list implementation of a generic queue. An `Iterator` has been implemented, which allows elements of the queue to be processed in sequence from front to rear without looking at the internals of the queue. So if we have a `queue<Integer> queue`, it may be used with code such as:

```
for (int n: queue) f(n);
```

(4) The class `counted_queue` is a little class which you need not modify. It is intended to show how you can use object-oriented programming to extend a class. In this case we want to keep a count of the number of items in the queue as well as the queue items themselves. The rest of the methods are inherited from `queue`. The `toString` function is used for debugging and extends the inherited function.

(5) The class `treemap` is the major data structures project for this assignment. Replace enough `UnsupportedOperationException` calls to make your program work. You will need to implement `put`, `get`, and `visit_all`. Implementing `remove` is optional.

(a) Important note: balancing the binary search tree is beyond the scope of this course and you should not attempt it for the purposes of this project. If you really want to know, see your Data Structures or Algorithms book, or one of:
`http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree`
`http://en.wikipedia.org/wiki/AVL_tree`
`http://en.wikipedia.org/wiki/Red_black_tree`

(b) Implementing `get`: Perform the standard binary search algorithm on a tree (one was presented in class). If the key is found, return the associated value. If not, return `null`.

(c) Implementing `put`: Use an algorithm very similar to `get`, except that you do one of the following:

(i) If the key is found in the tree, then the new value replaces the old value, and the old value is returned.

(ii) If not, create a new node as a leaf node, in such a way as to maintain the binary search tree property. Its parent will have a null pointer, which is changed to point at the new node. Insert the key and value. Return `null`.

(iii) In order to remember the node under which the new node is to become a child, keep a record of a previous or parent pointer in the same way as was done with insert ascending. Also keep a variable to remember whether the previous step was down to the left or to the right.

(d) You need not implement `remove`.

(e) Implement `visit_all_inorder`, which should perform an inorder traversal of the tree, and for each node found, pass the key and value field to the visitor. It will need a service function in order to do the recursion.

(f) The function `debug_dump_inorder`, provides a debug dump of the tree to verify its structure and balancing.

(g) The interface `visitor` provides a link between iteration done in the main module and it implementation in the tree module. Iteration over a tree is more complicated. This provides a way of passing a Java function through a parameter list. Unfortunately, in Java, functions are not first-class objects, and must always by wrapped in classes.

(h) The main program is the class `jxref`, which is partially implemented for you.

(a) The `main` function scans the options and iterates over all of the files whose names are given in `args`, if any, otherwise over `stdin`. It also causes a filename of hyphen (`-`) to be interpreted as `stdin`. You need not change this function.

(b) The class `options` scans the option word and converts args to a more useable format.

(c) The function `scanfile` only illustrates how to pick words out of lines. It consists of an outer loop which scans the file for lines, and an inner loop which uses a `Matcher` to pick words out of a line and process them. Replace much code in this function.

(i) Create a tree before the outer loop starts.

(ii) Inside the inner loop, every time a word is found, put it, along with its line number into the `counted_queue`.

(iii) First, if the word is in the tree, get its `counted_queue` and append the current line number to the end of the queue.

(iv) If not in the tree, create a new `counted_queue`, append the current line number to the end of that queue, and then `put` it into the tree.

(v) At end of file, depending on the options, call one or the other of the tree iteration functions. In the case of normal output, make use of the `wxvisitor`, provided for you.

(i) The `Makefile` is a little more involved in terms of doing clever things. Specifically, look at the following lines:

(a) `CLASSES = ${JAVASRC:%.java=%.class}`
All of the files listed in `JAVASRC` are class files to be made, with the suffix "`.java`" replaced by "`.class`".

(b) `INNCLASSES = ${CLASSES:%.class=%\\$$*.class}`
The inner classes have names the same as the outer classes, except that the outer class name is appended with a dollar sign and the name of the inner class. We specify them via wildcards rather than entering each name individually. The extra backslash (\) is needed because we run this expression through the shell twice.

(c) `LS_INNER = `(ls ${INNCLASSES} 2>/dev/null || true)`
After compiling all of the class files, we use the command `ls`(1) to find the names of all of the inner classes. Since some of them may not exist (not all classes have inner classes), we redirect (hide) the `stderr` to `/dev/null`, the bit bucket. We also hide any non-zero exit status from this command with `|| true`. This is not a pipe, but a logical or. Finally the backticks (``) cause the command to be passed through the shell in order to find the inner class files.

(d) `JARCLASSES = ${CLASSES} ${LS_INNER}`
This specifies what classes belong in the jar. If inner classes are not present in the jar, a `NoClassDefFoundError` is thrown when the jar is executed.

(j) You need not understand regular expressions in detail. The regex used to recognize words is given for you, namely the string `"\\w+([-'.:/]\\w+)*"`, which represents the regex `\w+([-'.:/]\w+)*`. In Java, two backslashes are needed to put one in a string.

4. What to submit

All of the Java source files that are part of your program, `Makefile`, and `README`. Finish the `submit` target in the `Makefile`. If you are doing pair programming, see instructions in `SCORE.pair`.