CMPS 12B

Introduction to Data Structures

Programming Assignment 3

Objective

The purpose of this assignment is to give you practice implementing binary trees and queues in Java, and also in combining different data structures together in order to solve more interesting problems.

Task

Your task is to write a utility that indexes words in a file. Your program should scan a file looking for words, then print out a table of all words found in lexicographic order together with a list of line numbers indicating where the word was found.

The program will use a binary tree data structure. Words must be able to be inserted and found in the tree in O(*log n*) time. No balancing of the tree needs to be performed, that is beyond the scope of this class. The value at each tree node will be a queue of line numbers. You must implement these queues using linked lists so that they do not run out of space and each operation runs in O(*1*) time.

The specifications for your program are presented in the format of a Unix man page.

NAME

xref - cross reference utility

SYNOPSIS

xref [-d] filename

DESCRIPTION

The xref utility reads in lines from the given file. Once the entire file is finished, the utility produces a printout of each word that appeared anywhere in the file, followed by a list of line numbers where the word occurred. Words are shown in sorted order. All normal output is sent to stdout, all error messages are printed to stderr.

OPTIONS

All options precede all operands.

`-d` Print a debug binary tree instead of a report. This shows all the nodes in the binary tree.

OPERANDS

The optional operand is a filename. When the program starts, a new binary tree is created. Lines are read from the file until end-of-file. Once all the lines have been read the printout is generated.

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.

Demonstration Implementation

To demonstrate what your program is supposed to do, you are provided with a Python implementation of xref. It can be found in the AFS directory for the assignment:

```
/afs/cats.ucsc.edu/users/r/nwhitehe/cmps12/asg3/
```

The Python version does not support debug binary tree output. It also uses data structures suited to Python, not binary trees with queues at the leaf nodes. Nevertheless it can be useful to see what the output of your program should be on different inputs.

Here is example test program test.txt:

```
apple banana cheese
dog elephant human
apple
fairy godmother cheese banana
banana banana
goblin fairy
half human elf
```

Here is an invocation of the Python demonstration program showing the correct output that your program should produce:

```
$ python3.3 xref.py test.txt
apple : 1 3
banana : 1 4 5 5
cheese : 1 4
dog : 2
elephant : 2
elf : 7
fairy : 4 6
goblin : 6
godmother : 4
half : 7
human : 2 7
```

Template Code

In the AFS directory you are given template code:

```
Makefile
Queue.java  Template code for a generic queue with iteration.
Tree.java   Template code for a binary tree with queues as leaves.
xref.java   Template code for main application.
auxlib.java Utility functions for main application.
```

Debug Output

With the "-d" option your application should build up the binary tree as normal, then dump it to stdout using the format described here instead of the normal output. The debug output is an in-order traversal of the tree. Each node should be displayed on one line. First indent by 2 spaces per depth of the node, then show an integer representing the depth of the node. Depth of the root node is 0 (indented 0 spaces). The left and right nodes that are children of the root node are at depth 1 (indented 2 spaces) and so on. After the depth number show the word at that node.

Here is an example debug output after inserting "moon", "cake", "pie", "fudge", and "apple".

```
    2 apple
  1 cake
    2 fudge
0 moon
  1 pie
```

Implementation Sequence

As with all programming projects it is good to organize your work so that you have a working program at all times, even if incomplete. Here is one possible sequence of steps:

1. Experiment with the Python demonstration program, make sure you understand what your application should do.
2. Look through the template code to understand the classes and what you have to implement.
3. Edit the code of xref.java to allow the "-d" option. Add a boolean flag that is set if one of the arguments is "-d". Pass that value to processFile(). Make sure your argument processing works properly.
4. Replace the exception generating code in Tree.java with debug output. Test your application to make sure that the insert() method is being called properly by the main application.
5. Work on implementing insert() in Tree.java. Your implementation will do a binary search on the tree to either find the word already in the tree, or find the location where the word should be inserted. Start by implementing binary search and throw an exception if the word is already in the tree. Print debugging information at each comparison.
6. Once you know where to add new words, update your implementation to actually insert a new word into the tree with a null Queue value.
7. Once you are satisfied that your tree insert() method is doing binary search properly, update it to create new Queue values for new nodes and then call insert() on the queue to add the first line number. If a word already exists in the tree, call insert() on the Queue value to add the new line number to the exiting node.
8. Now work on Queue.java. Convert the exceptions in isempty() and insert() to debugging messages. Once you verify that Tree is calling Queue properly, implement the Queue methods

to make them work.

9. Implement the debug output of trees to help debug any problems with tree insertions. Write code for debugHelper() that follows the debug format specified previously.
10. Fix any bugs in tree insertion. Generate more test files, try them out to make sure they work. You may want to call debug() in Tree.java at different points in your code to be able to see how your tree is changing.
11. Now work on output() in Tree.java. The output format is similar to the debug format. It does not require any indent or numbers but the order of traversal is the same. In addition, for each node you also need to iterate over all the elements in the queue at that node and print out each value. Create a new private method that takes a Queue object and prints out all the numbers in it. Use that method to finish implementing your output() method.
12. Make sure your program is working properly. Compare your output to the output of the Python demonstration program for several different input files.
13. Remove debugging output and tidy up your code. Verify that your makefile works properly. Then submit all your files.

Notes

1. You do not need to do any rebalancing of the binary tree for this assignment.
2. Unit testing is optional for this assignment.
3. Pair programming is allowed.

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. If you are pair programming then both names must be included in every comment block, and both names and CruzIDs must be included in the README.

Submit the following files:

```
README
Makefile
Queue.java
Tree.java
xref.java
auxlib.java
```

To submit, use the submit command.

```
submit cmps12b-nojw.f14 asg3 files
```

Credit: This assignment was created by Wesley Mackey for CMPS 12B, then updated and simplified.