

Program #5: Concordance Construction and 2-3-4-5 Trees

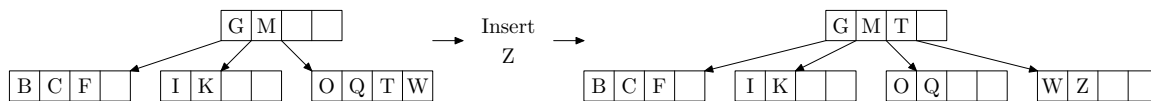
Due Date: April 29th, 2008, at the beginning of class

Overview: A concordance is an alphabetical list of all the words in a document, often with information on the location(s) of the words. Here, you'll create a 2-3-4-5 search tree of words from a text document, and keep track of the general location of each occurrence of each word.

A 2-3-4-5 tree (sometimes called a (2,5) tree) consists of nodes that may contain 1, 2, 3, or 4 data values each, along with 2, 3, 4, or 5 children each, respectively. 2-3-4-5 trees are really just small B-trees (note that “B” ≠ “Binary”). Both are height-balanced search trees; all of their leaves are at the same level in the tree.

Unlike binary search trees (BSTs), 2-3-4-5 trees grow at the root, not at the leaves. To insert, we search (recursively, for this assignment) for the item. If the item is not in the tree, we will reach the leaf node where it should be inserted. If there is room in the leaf (that is, if the leaf has fewer than 4 items), we add the item to the leaf. If the leaf is full and we need to insert into it, we select the median item from the group of 5 items, place the smaller items into a new node, place the larger items into another new node, and promote the median item to the parent. This splitting and promoting can continue back to the root of the tree, which in turn can split and produce a new root.

Here is a sequence of diagrams showing the progress of an insertion that adds a level to a 2-3-4-5 tree of integers:



Assignment: Write a complete, well-documented class that implements a 2-3-4-5 tree and a complete, well-documented program as a separate class that uses the 2-3-4-5 tree class to create a concordance of any UNIX-format text file supplied by the user. The output of your application is essentially an index of the file; a small amount of sample output is shown in the **Output** section, below.

Because words can occur multiple times in a document, your 2-3-4-5 tree will have to be able to keep track of the locations of each occurrence of all words. Rather than store multiple word-paragraph-line triples, you will instead store each word just once, and associate with it a collection of paragraph-line pairs, one pair per occurrence. How you implement this is up to you, but only store each word once in your tree.

Data: As stated above, your program is to accept any UNIX-format text file. Write your program to accept the file name as a command-line argument. Any unused contiguous sequence of characters in that file that starts with a letter or digit and contains letters, digits, hyphens, or apostrophes is considered a word and should be added to the concordance. Other symbols are skipped. For example, in the sequence “<Yawn!> 9:30’s too friggin’ early.”, the words are “Yawn”, “9”, “30’s”, “too”, “friggin’”, and “early”.

Case is not relevant; the words “Yawn” and “yawn” are to be considered to be the same word by your program. Any group of one or more blank lines separate paragraphs. Lines are separated from each other in the normal UNIX way (that is, with a single newline character); thus, two newlines in a row represent a blank line, three in a row represent two blank lines, etc. The first paragraph is paragraph #1, and the first line of any paragraph is line #1.

The file `prog5.dat`, found in `/home/cs345/spring08` on `lectura`, contains the text of a newspaper column by humorist Dave Barry (used completely without permission, but it’s old and its use here is hardly likely to hurt sales of his books; that’s my story and I’m sticking to it).

Output: Your program's output is to be an alphabetical list of the words that appear in the data file, along with an ordered list of locations, one location per occurrence of each word. If you remember how to get an ordered list of the contents of a BST, you should see how to do the same for a 2-3-4-5 tree. Here's a fragment of sample output:

| Word | Occurrences [form: (Paragraph#, Line#)] |
|----------|---|
| ---- | ----- |
| laying | (1,1) (4,1) |
| stupid | (4,2) |
| notaword | (1,1) (2,2) (3,3) (4,4) (5,5) (6,6) (7,7) (8,8) |
| " | (9,9) (10,10) |

Display a maximum of 8 occurrence pairs per line output, as shown. The above output shows that `notaword` occurs 10 times. The ditto mark (") indicates that the occurrences continue from the previous word.

Hand In: On the due date, turn in a printed listing of your well-documented program statements. Due to the size of the sample data file, you should **not** turn in a printout of the output it generates.

As usual, you are required to submit your completed program file(s) to the class submission directory, using the `turnin` command. The `turnin` location for this assignment is `cs345p5`. Name your main program source file `Prog5.java` so that we don't have to guess which file to compile/translate.

Want to Learn More?

- Our text doesn't cover 2-3-4-5 trees, but it does offer considerable coverage of 2-3-4 trees, which are slightly simpler than 2-3-4-5 trees. 2-3-4 and 2-3-4-5 trees can be viewed as special cases of a more general tree structure: B-trees. Resist the temptation to create (or worse, copy) a B-tree implementation for this assignment; 2-3-4-5 trees are much simpler than B-trees.
- Red-Black trees are a binary tree representation of 2-3-4 trees. Our text covers Red-Black trees, and even supplies sample code. Don't bother trying to adapt it for 2-3-4-5 trees; the implementations are quite different.

Other Requirements, Hints, and Miscellaneous Babbling:

- As mentioned above, you are to create a 2-3-4-5 tree class that is separate from the application class. You may create additional classes (e.g., a tree node class). If you are interested in exploring the use of abstract and sentinel nodes, this would be a good opportunity to do so. But, if you would prefer to have null pointers/references in your leaf nodes, that's OK.
- Don't get so into creating the 2-3-4-5 tree class that you neglect the task of figuring out how to fetch the next word from the file. Without correct words, even the best 2-3-4-5 tree implementation won't produce the correct word list.
- You are to code the 2-3-4-5 tree search and insertion operations recursively. While you're looking at those Java binary search tree implementations for reminders of how to do abstract nodes and sentinels, don't forget to pay attention to the recursion.
- Be sure to look at your output and compare it to the data file to verify that the output is correct. Better still, create your own data files and use them for additional testing. We plan to create our own test input files for use when grading your submissions.
- Finally, don't forget to double-check that you are "turnin-ing" all of the files necessary for the TAs to successfully compile and run your program. Remember that `turnin` has an option for listing the files that you have submitted.
- My usual final recommendation: Start early! For most students, a week is not enough time to complete this assignment. Plan your time accordingly.