

MCS 360 Programming Project #3 and #4

In Projects #3 and #4, you are to write a program that may be used to spell-check a text file and, optionally, to also count the frequency of words in the file. To spell check a file, you will need a dictionary — a reasonably complete list of English “words” correctly spelled. Here a word will consist of a sequence of letters (lower or upper case) and apostrophes; however, apostrophes at the start or end of a word will not be treated as part of the word. Note that the dictionary will contain only words — not their definitions, information on the pronunciation, or commentary on their usage. On the other hand, the dictionary will need to include all variations of each word. For example, one dictionary contains these variations on the word *abbreviate*:

abbr	abbreviating	abbreviator
abbrev	abbreviation	abbreviator's
abbreviate	abbreviation's	abbreviators
abbreviated	abbreviations	abbrevs
abbreviates		

We will assume that a dictionary is contained in a dictionary file — a text file containing one word per line, beginning in column 1 and continuing to the end of the line. I will provide you several dictionary files.

When your program starts up, it should check if an environment variable named `DICTIONARY` is defined. You may do this with the `getenv()` function, prototyped in `stdlib.h`. If `DICTIONARY` is defined, `getenv()` returns its value (a string). This string should be taken as the name of the dictionary file (possibly including path information). If `DICTIONARY` is not defined, `getenv()` returns `NULL`, and your program should assume the dictionary file is named **dictionary.txt** and lies in the current directory. Your program should attempt to open and read the dictionary file; if it cannot do so, it may print a message and quit. Your program should read this file and construct a binary search tree (Project #3) or open address hash table with linear probing (Project #4) in which each node or table entry contains one word as the key (as well as other fields, discussed below). Once the binary search tree or hash table has been constructed, your program should display some statistics. For Project #3, these should include:

- i) The number of nodes in the binary search tree (i.e., the number of words),
- ii) The height (maximum node depth) of the tree, and
- iii) The mean depth of the nodes in the tree.

For a hash table, you should display

- i) The number of table positions filled (i.e., the number of words), and the load factor.
- ii) The maximum distance between the hash position of a table element and its actual position in the table.
- iii) The mean distance between the hash position of a table element and its actual position.

Your program should then enter a loop. At the beginning of each pass through the loop, it should ask the user to select from four options.

- a) *Add words to the dictionary.* Your program should then prompt the user for the name of a file containing the words to be added. This file should be in the format of a dictionary

file (one word per line, beginning in column 1). Each word in this file should then be inserted into the binary search tree or hash table, unless it is already present. After this is done, you should again print statistics about the entire binary search tree or hash table, as above.

- b) *Remove words from the dictionary.* As in (a) above, except the words in the file are to be removed from the dictionary, if present, rather than added.
- c) *Spell check a text file, and optionally count word frequencies.* Your program should prompt the user to enter the name of the existing text file to be spell checked, the name of the new file to be created to hold the corrected text, whether a list of word frequencies is to be produced, and if so the name of the file that it should be written to. You should then read in the file to be checked and pick out the words. Any character not part of a word should simply be echoed to the new file holding the corrected text. Recall our definition of a word: A nonempty sequence of letters and apostrophes, in which the first and last characters are letters. For each word w in the text file, you should check whether the binary search tree or hash table contains a node or table entry with key w . If w begins with an upper case letter, you should also check for a node or table entry with key equal to w except that the first letter of the key is lower case. (The first letter of w might be capitalized only because it begins a new sentence.) If you find w in binary search tree or hash table, simply write w to the file holding the corrected text. If you don't find w , your program should display a message and provide the user with two choices.
 - i) *Accept the word, and add it to the dictionary.* In this case, you should write w to the file holding the corrected text.
 - ii) *Replace the word.* Your program should prompt the user to enter the replacement, which could be a null string, or could consist of more than one word. Your program should then write the replacement text, rather than w , to the file holding the corrected text.

If you are counting word frequencies, you will need a field in each node or table entry to hold the frequency of the word in that node or table entry. You should initialize or reinitialize this field to zero before you start to spell-check the file. When you find a word in the binary search tree or hash table, you increment its frequency. When the spell check is complete, write the list of words and frequencies to a file of the name designated by the user. Make sure the words are left justified, and the frequencies are right justified, in columns, so that it will be easy to sort the file using an editor or sorting utility

- d) *Quit the program.* Note that changes to the dictionary are lost when the program terminates..

At this point you have completed one pass through the loop. Unless (d) was chosen, you should begin the next pass, keeping any modifications to the dictionary from previous steps, and again prompting the user to choose among options (a), (b), (c), and (d).

In Project #3, you need to implement a binary search tree. The structure of a `TreeNode` should contain at least these fields:

```

TreeEntry entry;      /* A structure, containing the information in the node. */
TreeNode *left;      /* Pointer to left child, or null node has no left child. */
TreeNode *right;     /* Pointer to right child, or null if node has no right child. */

```

You may find it convenient add a pointer to the parent, as well. This pointer should be `NULL` in the root node.

`TreeEntry` is itself a structure type. It should have at least two fields:

```

char *key;           /* A dynamic string, containing a word from the dictionary. */
int freq;           /* Used to count the frequency of the word key in the text. */

```

You may make use of any code from the textbook in Sections 9.1-9.2, except for the deletion functions, `DeleteNodeTree()` and `DeleteKeyTree()`. Deletion makes the tree smaller; it shouldn't increase its height. But these functions can cause the height to increase — in some cases, to nearly double. (See page 408.) In Exercise E7 on page 410 of the textbook, the authors outline a sensible deletion algorithm, which I shall discuss in class also. Please use this algorithm.

In Project #4, you will need to implement an open address hash table with linear probing. I will supply you with more details when we talk about hash tables.

Project #3 should be submitted by Monday, December 3. Project #4 should be submitted by Monday, December 10.