Assignment

smccd.mrooms.net/mod/assign/view.php

Assignment # 4

Recursion and Trees Application – Building a Word Index

Make sure you have read and understood

- ***lesson modules week 10 and 11***
- ***chapters 9 and 10 of our text***
- ***Coding Style Guidelines (module week 1)***

before submitting this assignment. Hand in only one program, please.

Background:

In many applications, the composition of a collection of data items changes over time. Not only are new data items added and existing ones removed, but data items may be duplicated. A list data structure is a member of the general category of abstract data types called containers, whose purpose is to hold other objects. In C++, lists are provided in the Standard Template Library. However, for this assignment you will design and write your own linked list tree based implementation to support the ADT operations specified below.

Objective:

Working from a problem description design the data structures and operations needed to develop an application to solve the specified task.

Requirements:

Define a linked list tree based container and develop a set of operations for creating a word index that satisfies the application requirement specifications.

Problem Description:

A publisher is in need of an index to be produced for an upcoming data structures textbook publication. You have agreed to take on the task.

The first step in this process is to decide which words should go in the index. The second step is to generate a list of the pages where each word occurs.

The publisher has aided us on our way with this task by providing a list of all unique words used in the manuscript and their frequency of occurrence. Our job now becomes to review the list and choose which words to include in the index.

Discussion:

The primary goal in this problem is to identify a word with its associated frequency. Therefore, the first thing we must do is to define a "word".

Looking through our own data structures textbook, you may want to consider what is a tentative definition of "word" in this context? Considerations may include "something between two blanks" or a "character string between two blanks". The definition that works here is for you to decide. You want a definition that works for most words in an index. However, all words before "." and "," would have the "." and "," attached. Also, words surrounded by quotes would cause a problem. You need to create a working definition to take care of the problem at hand.

A suggestion could be:

A word is a string of alphanumeric characters between markers where markers are whitespace and all punctuation marks.

Yes, let's decide this is a feasible working definition of the kind of word that would be a candidate for an index term in this project. We can use function isalnum, available in , to determine if a character is an alphanumeric character. We can skip leading nonnumeric characters and store and read characters until we encounter a nonalphanumeric character (isalnum returns false) or inFile goes into the fail state. If we do not encounter any alphanumeric characters we return the empty string.

This process ignores quotation marks, but leaves contractions as a problem. A possible strategy here is to examine a few candidate words and see if a solution can be found. For instance, the common contractions "let's," "couldn't," "can't," and "that's" all have only one letter after the single quote. The algorithm would return the characters up to the single quote as one word and the character following the single quote as one word. Hmmm..what we want is to do is ignore the characters after the single quote. One idea is to require that words must be at least three characters long to be considered for the index. Besides, ignoring words of fewer than three letters also removes from consideration such words as "a," "is," "to," "do," and "by" that do not belong in an index.

Brainstorming:

It is now time to list objects that will prove to be useful in solving this problem. Scanning the problem description, the following nouns can be identified: publisher, index, text, word, list, pages, manuscript, frequency, and occurrence. It is now our job to select nouns that would serve as part of the solution. Suggestions here to pick include manuscript, word, list, and frequency.

Scenarios:

At this point a single scenario for this problem can be defined to allow us to begin designing our solution to build this index. Here we go: Read a file (i.e. the manuscript), break it into words, process the words, and output the results. To process each word, check its length. If it is three characters or longer, then check whether it is a word that has been processed already. If it is, increment its frequency; if not, add it to the list of words with a frequency of 1. In order to allow the publisher the additional flexibility to consider 'qualifying words' let the user enter the minimum number of characters.

Looking closer, the noun *frequency* can be seen as a property of a word in this case. This suggests to combine word and frequency into aWordType object. A container object (list) in which to store the items of WordType will be needed. To have the output file list the words in alphabetical order, a Sorted List ADT makes sense.

This is a good time to introduce a commonly used brainstorming tool used in the design of object-oriented software called class-responsibility-collaboration (CRC) cards.

The WordType card should look something like this:

Classname Word Type	
Responsibilities Initialize(word) Increment frequency GetWord GetFrequency	Collaborators

The ListType card should look something like this:

Classname ListType	
Responsibilities Initialize PutOrIncrement Print its contents in alphabetical order	Collaborators WordType String

Before going any further, it is time to decide on an implementation structure for ListType. No limit has been placed on the number of items in the list, so a linked implementation is appropriate. For each word, the list must be searched, either to insert a new word or to increment the frequency of an already identified word. A tree-based list would be the most efficient because its search has $O(\log_2 N)$complexity. A struct can be used to define a tree node.

In our original discussion, WordType was developing into a class with member functions to initialize itself, compare itself, and increment its frequency. Since the '*container*' class is being designed especially for this problem, make WordType a struct rather than a class and let the list be responsible for the processing.

We are now ready to draw out the algorithms to develop our solution:

Algorithms:

Main

```
Open input file
Open output file
Get file label
Print file label on output file
Get minimum word length
Set letters to GetString(input file)
while more data
    if letters.GetLength() >= minimum word length
        list.InsertOrIncrement(tree, letters)
    Set letters to GetString(input file)
list.Print(output file)
```

GetString(inFile) returns string

```
Set letters to empty string
```

```
Get a letter
while (NOT isalnum(letter) AND inFile)
    Get a letter
if (NOT inFile)
   return letters
else
  do
    Set letter to tolower(letter);
    Set letters to letters + letter;
    Get a letter
  while (isalnum(letter) AND inFile);
```

PutOrIncrement

```
if tree is NULL
   Get a new node for tree to point to
   Set word memberof Info(tree) to letters
   Set count member of Info(tree) to 1
   Set Left(tree) to NULL
   Set Right(tree) to NULL
else if word memberof Info(tree) is equal to letters
   Increment count member of Info(tree)
else if letters is less than the word member of Info(tree)
   PutOrIncrement(Left(tree), letters)
else
   PutOrincrement(Right(tree), letters)
```

Print

```
if tree is not NULL
    Print(tree, outFile)
    word member of Info(tree).PrintToFile(TRUE, outFile)
    outFile << word
    Print(tree, outFile)
```

You are now ready to code the algorithms.

Notes:

Use a defined const MAX_LETTERS to set the maximum word length to 20 characters.

Optional: Your solution can be in one file **index.cpp** to include data structures, operations and driver or in separate files.

Test Run Requirements:

As a test plan for this program use the provided **index.in** file to manually calculate the words and frequencies. Write the test results to the**a4indexFirstnameLastname** file (for FirstnameLastname *your* Firstname and Lastname).

Grading Criteria:

ListType class is correctly defined and implemented.

structs WordType and TreeNode are correctly defined.

Defined const used to set maximum word length to 20 characters.

Program compiles and runs.

Implementation supports the operations given in the algorithm functional requirements.

A test driver is included to satisfy the test run demonstration.

A copy of your test run output display is included in an output file named a4indexFirstnameLastname

Be sure to include (at minimum) 2 separate files:

index.cpp (complete solution in one file) *or (separate files) AND*

a4indexFirstnameLastname