Josh Martin

Dr. Snehasis Mukhopadhyay

CSCI 36200

19 March 2018

Programming Assignment 2: Lexicographic Tree

**Project Description**

The main problem of which this project is trying to fix is constructing a cross-reference index for a given text file. A cross-reference index is an instance within a document which refers to related information elsewhere in the same document. In both printed and online dictionaries cross-references are important because they form a network structure of relations.

We can do this by using a data structure called a binary tree. A binary tree is a data structure in which each node has at most two children (which are referred as left child and right child). This data structure is used to store a list of words passed in by the user from a file and then sorts the words alphabetically in the file by use of a binary search. And stores line number of occurrence. Binary Search (also known as half-interval search) is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array.

*An example of using this cross-reference would be using the following three words:*

*"abracadabra*","hocuspocus", and *"watchamaycalli"*

*"abracadabra"*- would be the root of the tree

*"hocuspocus"* - would be the left child of the root

*"watchamaycalli"* - would be the right child of the root

**Source Code**

included source files and header files:

lexicographic/lexicographic.cpp

lexicographic/lexicographic.h

node/node.h

main.cpp

input\_files/another\_text\_input.txt

input\_files/text\_input.txt

readme.md

these to files are just for formatting strings and I found it on stackoverflow:

string\_formater/string\_formater.cpp

string\_formater/string\_formater.h

**Sample Output**

WELCOME

If you would like to exit type: `#`

civilizati 1

destiny 3

is 2 3

knowledge 2 3

of 1

our 3

science 1 2

**Comments**

*The Basic Approach*

I chose to write my program in C++ because it is one of the languages that I am most comfortable programming in. I have been using it since high school. C++ also offers the advantage of maximum control of what your program does and how it manages memory. My basic approach was to design the program so it could take in standard input. That way it could read in files as well as input from the user. Most of the text processing happens in the main file.

The main file checks if first a '#' has been enter the program will halt; if not the program will continue processing the current line of text. Then passed to a parser function (that I created), that will normalize the text. breaking the sentence into individual words, then making the text all lower case, checking if each word starts with a letter, and removing punctuation marks. Each word is then sent to be sorted in the tree alphabetically. Each word in the tree is stored in a node that has a vector which stores on what lines the words shows up, each node also has a left & right child node.

when we insert a node, my algorithm first checks if the tree is empty then will make the new node the root of the tree. If not, my algorithm will search if the node already exists in the tree. If it finds no such node it will compare each word in the tree with the new node and on what child it belongs to be inserted.

*Binary Search*

In my research it wasn't that hard to find great resources on how to do binary search on a tree. It seems to be easier than doing a binary search on an array. CS50 on YouTube was of good use to me in the process of writing the Binary Search algorithm because before this program I have never tried to implement this search algorithm. it did seem quite scary at first but once I watched a few videos it wasn't that bad.

*Runtime Complexity*

There is only one main algorithm in this program, that is a binary search of the tree when inserting new elements which at worst case is O(logn). To print each element, it would at worst case take O(n) because there is N elements in the tree.

We could improve the time complexity using a Trie data structure, instead of a binary search tree. A Trie stores every letter into a node instead of a whole word. In my potential implementation we could also store the line numbers in the last letter of the key. A Trie is beneficial because it would not depend on the size of the tree (N) but the size of the string we are searching for (M). Therefore, our time complexity would be O(M) and not O(N). Which could make things a lot faster.