**Beteab Gebru**

**Apr 27 2017**

**CSCI 301 - Section 02**

**CONCORDANCE WITH A BINARY SEARCH TREE**

**Project 10: Program Documentation**

### Introduction

INTRODUCTION

A **concordance** of a text is an alphabetical table of the words that appear in the text and the number of times each word appears. Concordances summarize the frequencies of words in text and are used in statistical analyses of authors' works and to determine authorship of disputed works.

I will design and implement a program that will ask for **name of a file** to be used as the input source. The program will read words from the file and store concordance data in a **Binary Search Tree Structure**. For this intends and purposes; **Any sequence of *letters* is a word**, and all non-letter characters are separators, equivalent to white space. One or more non-letters may separate words, and ends of lines separate words. Differences in capitalization do not make words different; that is, "HOUSE," "house," and "HouSe" are three instances of the same word.

I will use a BST ADT with functions such, Copy, Destroy, Insert, Remove, Print, Empty?, Present? To help it maintain the Tree structure.

**Design Document**

**Data structures**

We will use Binary Tree structure with nodes that have data members **Data**(to store a Distinct word), and **Count**(how many times a text appears in a given input text file) . It also has Data member called Distinct\_Words to keep track of number of Nodes/Words.

**Data Members And Functions of the ADT**

//Name: Beteab Gebru

//proj 10 - Concordance With a BST

//Due Date : April 27 2017

//CSCI 301: Section 02

//CSCI ID : cs301154

typedef string Item;

class BST

{

private:

    struct Node

    {

        Item Data;

        int Count=0;

        Node \*Left;

        Node \*Right;

    };

    Node \*Root;         //the Root node

    int Distinct\_Words;//Number of Nodes

//========================================helper functions

    Node\* Copy(Node \*T);//copy constructor helper

    void Help\_Insert(Node\* &T, const Item& Item);

    void Help\_Remove(Node\* &T, const Item& Item);

    void Remove\_Node(Node\* &T);

    void Help\_Print(std::ostream& Output, Node \*T) const;

public:

    //=========================== Constructors

    BST() { Root = NULL; }

    void BST\_Copy(const BST& T) { Root = Copy(T.Root); }//copy construxtor

    //=========================== Destructor

    ~BST() { Root = NULL; }

    //=========================== member functions

    int Length();

    void Insert(const Item& Entry);

    void Remove(const Item& Removed);

    bool Empty();

    bool Present(const Item& Entry, Node \*Root);

    //int Get\_Count(string Word);

    //=========================== Friend functions

    friend ostream& operator << (ostream& Output, BST& Tree);

};

**Functions**

**Insert()**

-This function Will have a helping function which is recursive and private. It will walk through the tree to find the right location for the entry item. Its only parameter is the Item to be inserted. The helper function Help\_Insert will creste a node with the item in it and initialize count to 1. This function will initialise the pointers to the left and right to be NULL if the tree is getting its first item. If the tree is already populated it will search through the correct side of tree (depending on the item’s magnitude).When item is greater than the node we walk the right edge, when item is smaller than the node we walk to the left enge and compare with the node/vertex on that path.

**Present** (); This function searched through a tree to find if an item is present. For concordance building I have decided to add the job of incrementing count on words(when word is present) into this function. It returns true when item has been found and false when item is not found and we are at the tip of the tree(noted by NULL ptr.

**Length();** This function will return the number of items in the concordance tree. It is always upto date as insertion increments it and removal decrements it. Its up todate at all times.

**Is Empty? Function**

-This function simply returns the truth value the logical statement Root ==NULL.

**BST\_Copy();** This is a copy constructor that will copy all the data(instead of just the datamembers) when copying of a tree is required. It will call a private recursive function to do the actual job of walking through the tree to copy the data.

**Friend function**

-This is the overloaded output stream function that will be able to take the BST ADT object and display its data elements in a chosen format. In my case I have decided to output the elements in a tabular format

**The Main program:**

The main program is used to get the words from the text file input and format them appropriately (all uppercase and max of 8 characters long). It extracts each character using **get()** function and builds words with each character. It will treat white space and non alphabet characters as separators and disregard them in the process. and send this separate words individually to the **My\_Tree.Insert(Word)** function as parameters.

The program makes use of three file names stored in an array to test the program. The three input text files are. I Did this to avoid having to type file names during tests. I will be using array index 0,1,2 to use one of the three files to save time on file name entry.

1. "Text1.txt"

2. "Text2.txt"

3. "Text3.txt”

The Main function includes various loops of instruction to format the word in upper case and cut a substring to max of 8 characters before sending the word to the insertion function.

We also have testing ground for **remove()** and **Length**() functions in the main prog.

We use the main program to print out our list using the overloaded output operator.

### User Document The way to run the program

The program can be run from Microsoft Visual studio 15 or g++. Simply select the file to test on by selecting array 0 for(Text\_1.txt), 1(Text\_1.txt), 2(Text\_1.txt). the program will ask for a word to remove and display a concordance data with the item removed from the tree.

### Testing of the program::Screen shots from my visual studio IDE ..

### Summary

* The program essentially builds concordance data the same as project 4 and 6. The difference being we are building tree structure to store our data. The tree structure has the following characteristics.
* Relatively easier complexity to implement
* inserts are O(logN)
* lookups are O(logN)

It is a great way to visualize data as well.

**Conclusion**

With this project I was able to keep my code more organized than previous projects. I was able to learn to use recursion to keep functions short and apt. It was frustrating when a seemingly logical statements don’t work but I was able to discover ways around some errors during run time by using debugging lines that pause **system(“pause”)**the program so I can look at where the bug occurs.