CS 218 DATA STRUCTURES
ASSIGNMENT 4
SECTION B, C, D and G
Fall 2020

DUE: Nov 20, 2020
TO SUBMIT: Documented and well written structured code in C++ on classroom. Undocumented code will be assigned a zero.

PROBLEM BACKGROUND
We want to design a small search engine that can help its users to retrieve documents related to the search query from a collection of documents as done in Assignment 1&2.
In assignment 1&2, the unique terms were stored in an unordered array/linked list. The search operation is therefore O(n) in the worst case. If we replace the array/list with a balanced BST, then we can search these terms in O(lgn) time in the worst case. You task is to redo the assignment1&2 but now with AVL.

EXAMPLE
Consider a small example below where we have following documents

Doc 1 breakthrough drug schizophrenia drug released july
Doc 2 new schizophrenia drug breakthrough drug
Doc 3 new approach treatment schizophrenia
Doc 4 new hopes schizophrenia patients schizophrenia cure

Unique terms are breakthrough, drug, schizophrenia, released, july, new, approach, treatment, hopes, patients, cure. The index will be as follows: Terms will stored in an AVL tree and for each term there is a list that contains document ID, and term frequency. See the following figure for the structure of the index.

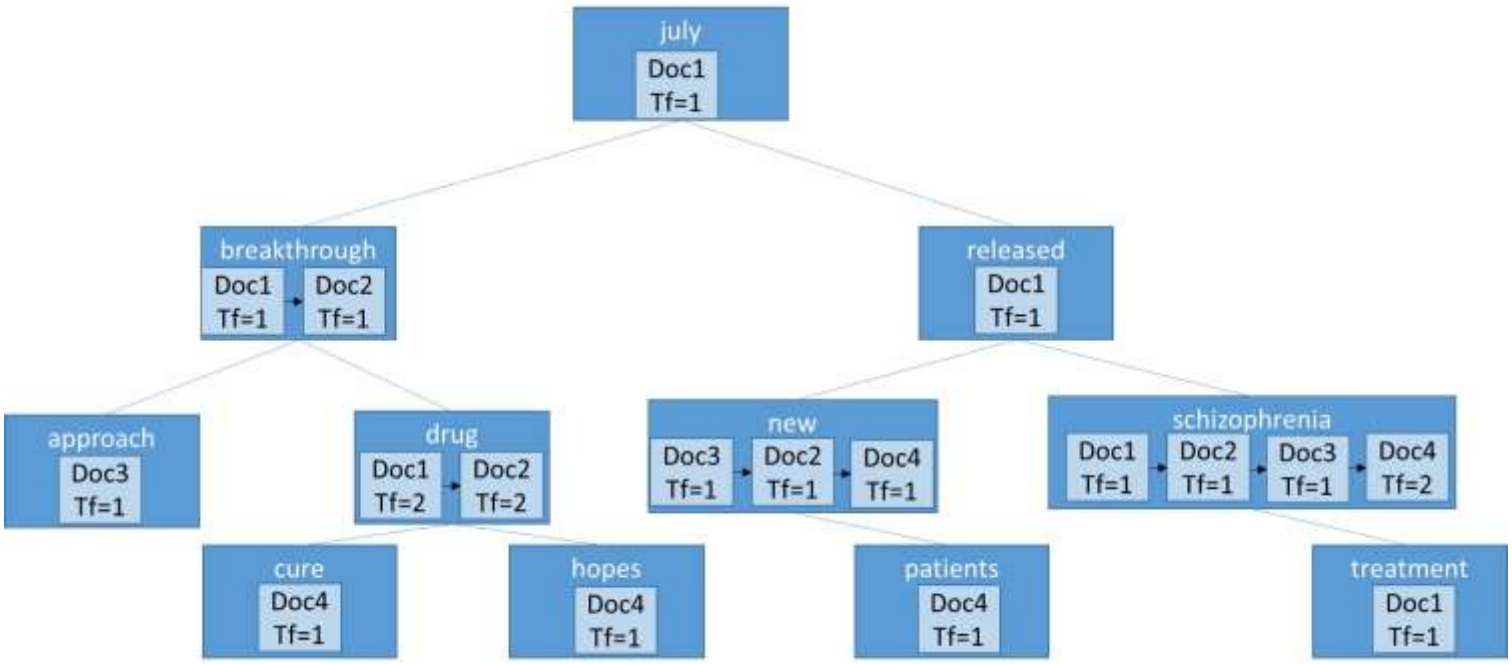

Figure1: Index of Terms

IMPLEMENTATION

Your task is to design a small search engine that can perform following functionality:

Create_Index: Given the collection of documents, tokenize words in each document, compute their term frequencies and create the index.

Search_Documents: Given the query word(s), output a ranked list of related documents as done in assignment 1&2.

Add_Doc_to_Index: Given a new document (DOC ID and the text), tokenize its words, compute their term frequencies. If a word is already present in the index, add the Doc ID and computed term frequency at the end of the corresponding list. Otherwise add the new word in the AVL tree. For example if we have to add a new document:

"Doc5 miracle drug" to the index shown in Figure1, then updated index will be as follows:

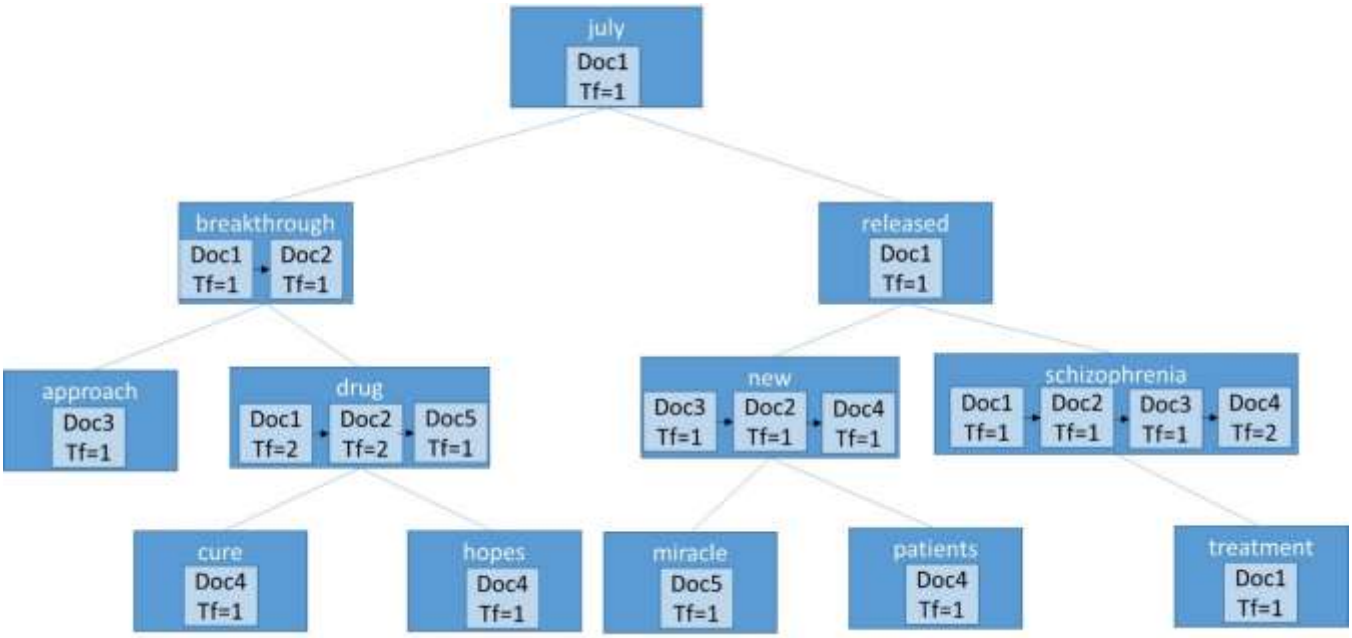

Figure2: Updated Index

Important Note: For simplicity, you can assume that all the documents and query do not contain any stop word and all words are separated by white space.

IMPORTANT CLASSES

You have to implement the following classes

Class List

Use the class List that you designed in assignment 1&2

Class AVLTree

Implement AVLTree class and Anode class. The Anode class must have following data members: data, left, right and height. The AVLTree class will have one data member root. Implement all the required operations including constructor, destructor, copy constructor.

Class Doc_Info

This class must contain two data members DocID and term frequency of a particular term

Class Term_Info

This must contain a key term and a list of Doc_Info as its data members.

Class Search_Engine
This must contain an AVLTree **Index** of type Term_Info, Following is the list of required functions:

Create_Index: This function takes an array ***Docs*** of type strings/char* that contains the file names and an integer ***n***. Here n is the size of array ***Docs***. For each file in ***Docs***: call **Add_Doc_to_Index**

Search_Documents: Given the query word(s), output a ranked list of related documents as done in assignment 1&2.

Add_Doc_to_Index: Given a file name: open it, read it and tokenize words on white space. Also, compute the term frequency of each unique word in the file. For each unique word, if the word is already present in the index, add the Doc ID and computed term frequency at the end of the corresponding Doc_Info list. Otherwise add the new word in the Index.

Add any function that you find important to implement above mentioned functions.
For simplicity you can declare Doc_info and Term_info as friends of Search_Engine.