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. A search query is a collection of one or more words. A document is said to be related to the search query if it contains one or more query words. The idea is to examine all documents of the collection and output only those that contain the words from the search query. For example, if the search query has the word “velocity”, then we have to find all those documents that contain the word “velocity”.

The job of a search engine is to efficiently retrieve the set of desired documents and then rank them according to their relevance. To make things faster, an index of important terms/words is maintained. For each word in the index, a list is maintained that keeps the record of documents in which that term appears along with some other important information like term frequency, and list of positions where that word occurs.

TF: Term Frequency, which measures how frequently a term occurs in a document.
TF(t,d) = Number of times term t appear in document d

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 be stored in an array. For each term, there is a list that contains document ID, and term frequency. See the following figure for the structure of the index. The word drug is present in Doc1 and Doc2 two times in each document.

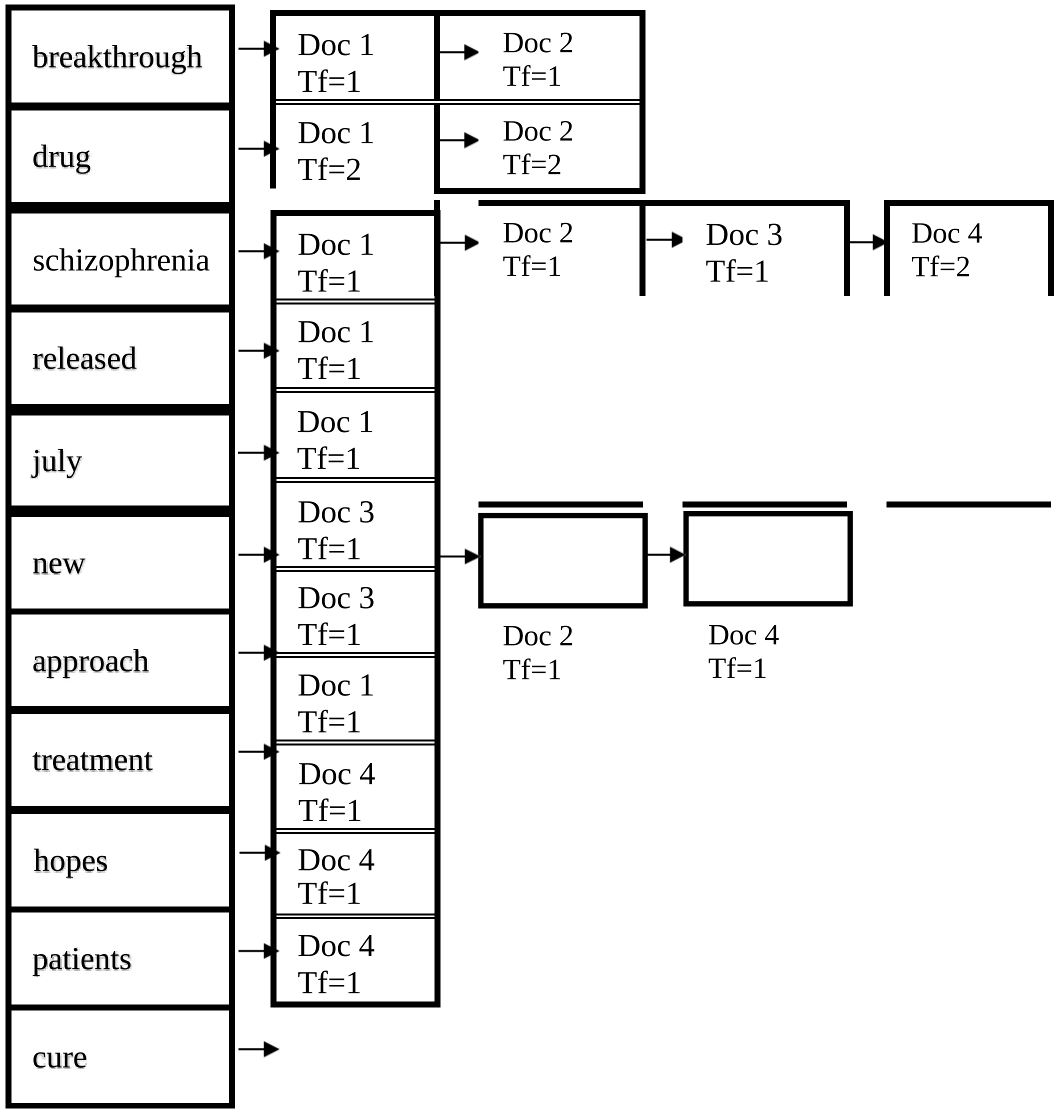

Figure1: Index of Terms

To gain the speed benefits of indexing at retrieval time, we have to build the index in advance. The major steps in constructing an index are given in Algorithm1.

1. Collect the documents to be indexed:
2. Tokenize the text, turning each document into a list of tokens/terms/words
3. Remove stop words (of, for, the etc.), producing a list of important words
4. Put each unique term in an array. For each unique term create a list of documents where that term exists. A sample Index is shown in Figure 1.

Algorithm1: Index creation

The retrieval process is explained in Algorithm2.

1. Input search query
2. Tokenize the text in the query
3. Remove stop words
4. For each query term, search it in the index, collect the list of documents. If there is more than one word in the query, take the union of all the lists.
5. Rank the retrieved documents.

Algorithm2: Document Retrieval

The ranking function works using the following rules:

Rule1: Documents containing more query terms must be ranked higher than the documents containing lesser query terms.
Rule2: If two documents have the same number of query terms then rank the document higher that has higher collective term frequency.
Rule3: If two documents have the same number of query terms and the same collective term frequency then rank the document alphabetically by Doc ID

Table1: Ranking rules

Suppose you are given this Query= "drug schizophrenia"

Word drug occurs in Doc1 and Doc2. Schizophrenia occurs in Doc1, Doc2, Doc3, and Doc4.
Doc1 and Doc 2 will be ranked higher than Doc3 and Doc4 because of *Rule1*.
Doc 4 will be ranked higher than Doc3 because of *Rule2* as Doc4 has high term frequency.
Doc1 will be ranked higher than Doc2 because the number of query terms in both documents is the same i.e. 2. Also the collective term frequency is 3 in both documents so using *Rule3* Doc1 will be ranked higher.

The final ranked document will be
Doc1
Doc2
Doc4
Doc3

IMPLEMENTATION
PART (A)

Your task is to design a small search engine that creates an index of important key terms for efficient searching. Following is the list of classes that you need to implement.

IMPORTANT CLASSES

You have to implement the following classes

1. **Class List**

Implement doubly linked list in class List and the iterator class as nested class of List. The List class must have the following data members: head, tail, and size. Head and tail will store the first and last node address respectively, and size counts the total number of data items in the list. Implement all the required operations including constructor, destructor, copy constructor, and overloaded assignment operator.

2. **Class Doc_Info**

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

3. **Class Term_Info**

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

4. **Class Search_Engine**

This must contain a dynamic array **Index** of type Term_Info, array_size, and array max_size as data members. Following is the list of required functions:

i. **Constructor**: Create the array Index dynamically of size max_size. Initially, the index is empty so array_size will be zero.

ii. **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*** perform the following tasks.

    a. Tokenize the words separated by white space and compute the list of words
    b. For each word in the list, insert it into the array Index such that each unique term exists only one in the index.

iii. **Destructor**: Delete all the memory dynamically allocated memory.

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

Important Note: For simplicity, you can assume that all the documents and queries do not contain any stop word and all words are separated by white space. We will remove the shortcoming of using an array of key terms in the upcoming assignments

PART (B)
In this part you will implement the remaining functionalities of the search engine. Add the Following functions in the class Search_Engine

iv. **Search_Documents:** Given the query word(s), tokenize them by white space, compute the list of documents related to query and output a ranked list of related documents as explained in Algorithm2 and Table1.

v. **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 at the end of the array Index. If array Index is already full double its size and then add. For example, if we have to add a new document: Doc5miracle drug" to the index shown in Figure1, then the updated index will be as follows:

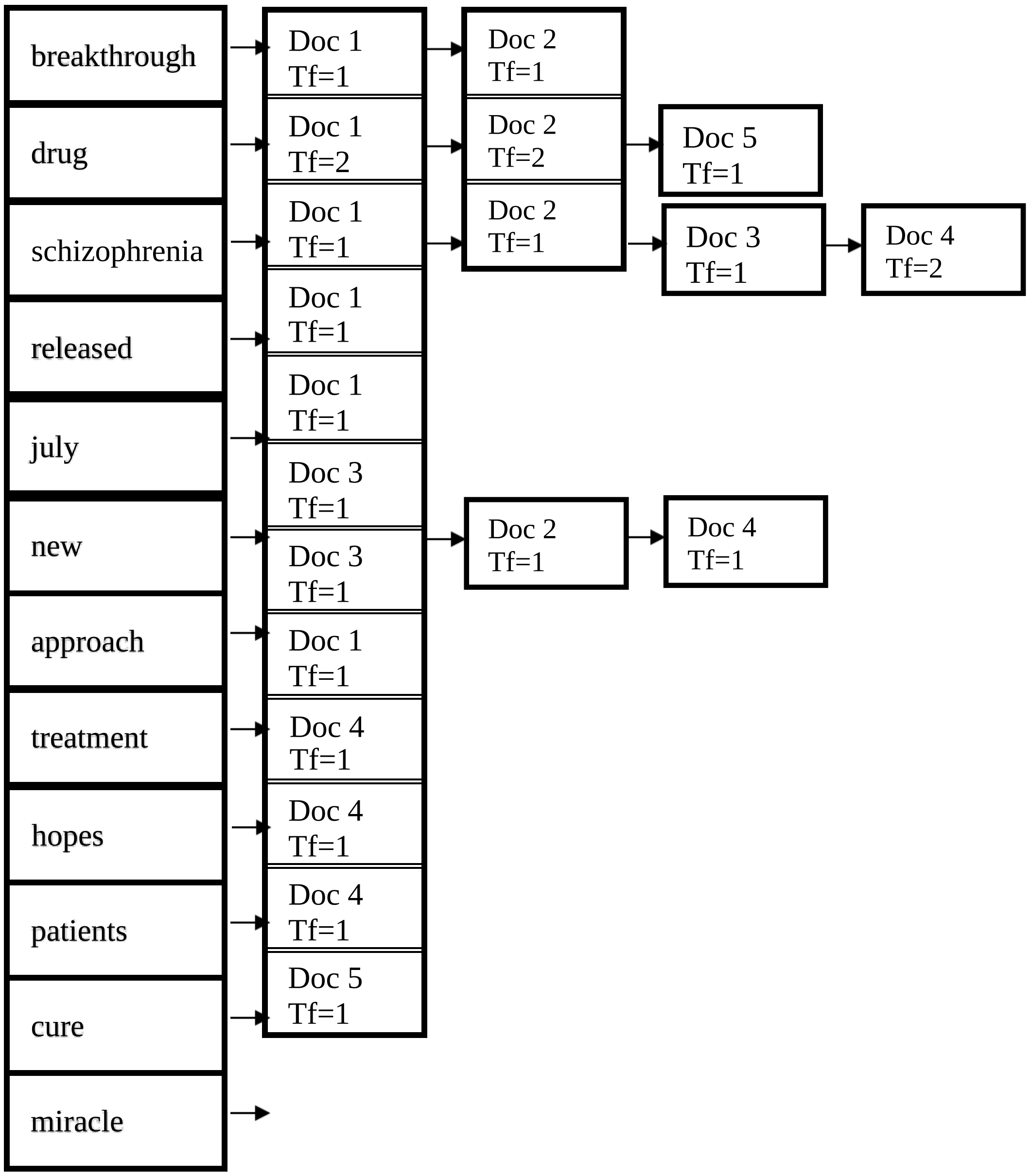

Figure2: Updated Index