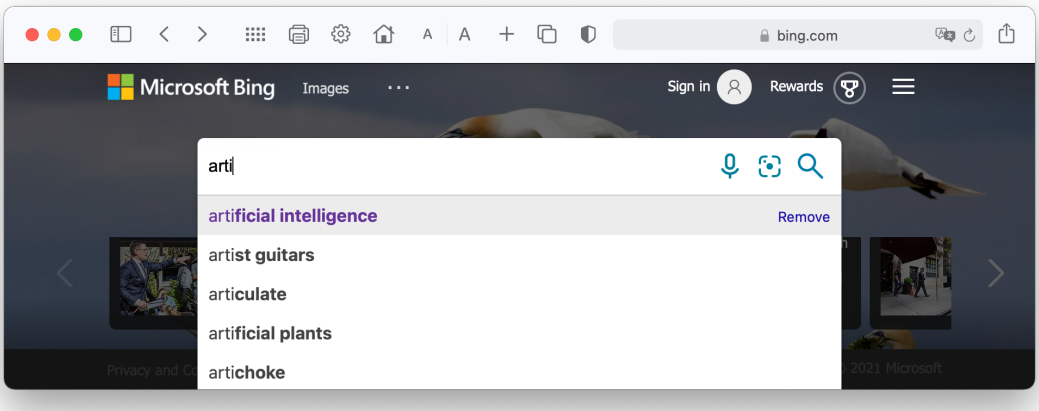
Assignment 2

Submit Deadline: 10pm Friday 23 May 2021

1. Problem

Almost with any search engine, such as Google and Bing, when you input a sequence of characters, the input panel will display a list of words/phrases as input suggestions for a quick access.



For instance, if you input "`arti`", you will see a list of words/phrases starting with "`arti`". If any of the suggested words/phrases is what you are looking for, you may click on it without further input. The list will dynamically update when you input more characters.

To achieve speedy search, a technology called *Indexing* is widely used by many search engines[1]. Indexing is a process of collecting information from data sources to help access the data effectively. A simple implementation of indexing is to create a data structure called "*inverted index*", which stores a mapping from words/phrases to their locations in data sources (documents or web pages)[2]. An inverted index looks like a dictionary which record location or other information for each word/phrase.

As an example, consider the following texts

Text1: `"It is what it is."`
Text2: `"What is it?"`

[1] https://en.wikipedia.org/wiki/Search_engine_indexing
[2] https://www.geeksforgeeks.org/inverted-index/ or http://en.wikipedia.org/wiki/Inverted_index

Text3: `"It is a banana."`

An inverted index for these texts is a table (Table 1), which maps each word to a list of indices (after trimming out the punctuations and ignoring cases). Each index consists of the *name of document* that contains the word and the *word's location* in the document[3]:

Table 1: *Inverted index*

Word	**Index**
a	{(Text3, 3)}
banana	{(Text3, 4)}
is	{(Text1, 2), (Text1, 5), (Text2, 2), (Text3, 2)}
it	{(Text1, 1), (Text1, 4), (Text2,3), (Text3,1)}
what	{(Text1, 3), (Text2,1)}

With the inverted index, whenever a user searches for a word, the search engine knows which documents contains the word and where, without having to traverse the original files. This makes search much more efficient.

An index can also contain other information related to search content, such as the frequency a word appears in all documents (Table 2). Such information can help a search engine to rank input suggestions as shown above.

Table 2: *Inverted index with frequency*

Word	**Index**	
a	1	{(Text3, 3)}
banana	1	{(Text3, 4)}
is	4	{(Text1, 2), (Text1, 5), (Text2, 2), (Text3, 2)}
it	4	{(Text1, 1), (Text1, 4), (Text2,3), (Text3,1)}
what	1	{(Text1, 3), (Text2,1)}

This assignment is to implement a data structure of *inverted index* with word/phrase frequency to facilitate input suggestions. We will call an inverted index an *index dictionary*. You are required to create an AVL tree to store indices by learning a set of text files. You may choose a book or a set of text files for you to learn the inverted index. If it is a book, view different chapters as different documents (use chapter number as the name of documents). You can learn the index dictionary by traversing all the documents, recording the name of documents and the location of each appearance for each word/phrase, and then counting the total appearances of each word/phrase. The frequency of a word/phrase can be represented as the number of appearances per thousand words.

[3] Location can be measured in different ways, for instance, the number of characters or the number of words from the beginning of the document. In this example, location represents the number of words from the beginning.

The information of each word/phrase must be stored as a node in an AVL tree. The generated AVL tree will be the index dictionary.

Once you have an index dictionary, you then implement an algorithm to simulate input suggestions. Your algorithm should accept an input of a string of characters, generate a priority queue of words/phrases that match user's input in the order or frequency, and display the top part of the queue, say five items. For instance, if you input "`arti`", you will provide a list of words/phrases starting with "`arti`" in the order of their frequency. Display does not need to be in "real-time" appearing immediately after you key in a character as seen in Google or Bing. The list can be displayed after the *return key* is pressed[4]. Your list can be empty if there is no word/phase that matches the input string.

2. Requirements

2.1 Coding (85% of marks)

Basic task (40 marks): Implement an index dictionary. For learning purpose, we require that an index dictionary must be stored in an AVL tree based on provided AVLTree ADTs. The AVLTree ADTs can be downloaded on vUWS (in C++ or Java). You may use other AVLTree ADT implementation from other resources for either more functionalities or higher efficiency, but you must include their source code. You are not allowed to use STL `map` or other built-in AVL tree implementation to store the dictionary (you may use `map` for other purpose in the assignment). You can change the provided AVLTree ADT code for your purpose. Your program should have the following functionalities:

1. Build an index dictionary, represented in an AVL tree, by reading any text files. Each node of the AVL tree represents a word (as the key) and its index - the frequency and a set of (*document name, location*) pairs (as the data)[5].
2. Each word must be trimmed so that a word contains only alphabetic letters.
3. Print all the nodes by using in-order traversal with information of the words and associated indices.
4. Print the AVL tree with the information of each node on the AVL tree: the key, balance factor and the level of the node (you may use the built-in functions of AVLTree ADT).

You must complete the basic task. You can claim extra marks by completing the following add-on tasks in any order.

Add-on task 1 (10 marks): Extend your code to implement the following functionalities:

1. Save your index dictionary into a text file.
2. Rebuilt your dictionary by reading the file saved previously (note that the way you save the dictionary determines the efficiency you rebuild the dictionary).

[4] Just to avoid the complexity of system-depended programming.
[5] See Table 2.

Make sure your code can learn from multiple text files (or a text file with separate chapters/sections).

Add-on task 2 (10 marks): Extend your index dictionary so that it can also include phrases. You can limit the length of a phrase to two to three words and include only highly frequently appeared phrases. Note that you might have to make changes for other functions so that the input and output work well with phrases.

Add-on task 3 (10 marks): Extend your index dictionary so that it can delete words/phrases with lower frequency (users supply a threshold, say 0.1 per thousand words). If you implement it by adding more functions in the AVL ADT, you receive 10 marks. If you implement it by exporting the inverted index into a file and then reload the dictionary from the file, you receive 5 marks.

Add-on task 4 (15 marks): Extend your code to implement input suggestions. Your program should have the following functionalities:

1. Generate a priority queue for each input of characters in the order of frequency as recorded in the index dictionary.
2. Display a list of words/phrases from the priority queue according to your input. You may limit the length of the list to be at most five items.

Note: *To avoid spending too much time on the assignment, you do not have to implement a fancy GUI. Focus on your algorithms and data structures.*

2.2 Documentation (15 marks)

In this assignment, you are required to submit a formal document no matter which level of code you have implemented. The document should explain the algorithms and data structures you have used in your code. You should also use the document to demonstrate the depth of your understanding of the data structures you have used in your code. Your document should include the following sections:

1. **Introduction:** demonstrate your understanding of the problem. (2 marks)
2. **Algorithms:** Choose two major algorithms (sets of functions in classes to implement a functionality) in your program to explain your idea of implementation using pseudocode or simplified source code (list the major operations only). You should choose the most significant algorithms to present in order to demonstrate your skill of programming and the depth of your understanding. (3 marks)
3. **Data Structures:** List all the major data structures you have used for your implementation and briefly explain why they need to be used. (3 marks)
4. **Complexity analysis:** Depending on which tasks you have implemented, you would need to show the complexity of the following operations:

a. The complexity of building up an index dictionary assuming that the total number of words in the text file is n.[6] (3 marks)
b. The complexity of rebuilding a dictionary from a dictionary file (you cannot claim this mark if you do not have an implementation for add-on task 1). (3 marks)

5. **Conclusion**: summarize what you did. (1 marks)

3. Submission

Both the documentation and source code should be submitted via vUWS before the deadline. Your programs (.h, .cpp, .java and .txt) can be put in separate files (executable file is not required). All these files should be zipped into one file **with your student id as the zipped file name**. Submission that does not follow the format will not be accepted.

Email submission is not acceptable (strict rule).

4. Demonstration

You are required to demonstrate your program during **your scheduled** practical session. We will check your code and your understanding of the code. **You will receive no marks if you fail the demonstration, especially if you miss the demo time.** Note that it is students' responsibility to get the appropriate compilers or IDEs to run their programs. You are allowed to run your program from your laptop at the specified demo time. **The feedback to your work will be delivered orally during the demonstration.** No further feedback or comments are given afterward.

[6] You may consider that the dictionary contains words only without phrases.