Lebanese University - Faculty of Science
Branch V

Simulating a Dictionary Using Appropriate Data Structures

I2206 - Data Structures

Submitted by:
Ali Koteich
Mostafa Ismail

Supervisor:
Dr. Hussein Wehbe

2020 - 2021

Table of Content

1. Introduction

Traditional dictionaries used to be in a form of hard copy, which takes several minutes to search for a single word and its synonym, and with the rise of the computers, most dictionaries became available online which facilitates the search process.

As part of our course I2206 we have to implement a program that simulates a dictionary search engine using the appropriate data structures given in the course.

Inorder to simulate an optimised dictionary search engine we take into consideration two major factors: time and space complexities. Time complexity is how long a program takes to process a given input. Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. Not to mention the scalability of the algorithm when simulating a real dictionary.

2. Project Objectives:

This project aims to use the skills acquired in the "Data structures" and "Imperative programming 2" courses in an application framework simulating a dictionary.

Our project do the following:

1. Loads the dictionary in a binary search tree
2. Searches if a word exists in the dictionary
3. Prints the synonyms of a given word
4. Adds words with their synonyms to the dictionary
5. Prints the word with most synonyms
6. Prints the words for which a given word is a synonym
7. Prints the most repeated synonym in the dictionary
8. Checks if the binary search tree is balanced
9. View recent search history

3. User manual:

To run the project:

➔ Access the folder named (**Final Project**).
➔ double click the executable file named (**I2206.exe**)
Warning: If the program crashes on start, delete the file ***NewDictionary.txt*** then rerun the program again !

The folder also contains following files:

➔ **"main.c"** where our functions are called and user interface made,
➔ **"functions.c"** where all the functions are implemented.
➔ **"functions.h"** contains the header functions and structs.
➔ **"I2206.cbp"** where the whole project is stored.
➔ **"WordsSynonyms.txt"** is the text file we were provided with, that contains the dictionary.
➔ **"NewDictionary.txt"** is the text file where we store the dictionary in an alphabetical order and include it next time we run the program.
➔ **"AddedSynonyms.txt"** contains the words and synonyms added by the user to the dictionary.
➔ **"history.txt"** which stores the words we searched for previously.

4. Implementation

4.1 Data structures used in our program

The first data structure used is the **binary search tree (BST)** since we were asked to implement this specific type of data structure to represent the words of the dictionary.

In order to choose the appropriate data structure to store the synonyms associated with each word, we made a study of different algorithms in terms of complexity as shown in table 1.

	Dynamic array	Linked list	Binary search tree	Hash table
Search complexity	O(logn)[1]	O(n)	O(logn)	O(1)
Insert complexity	O(1)	O(1)	O(logn)	O(1)
Space complexity	O(n)	O(n)	O(n)	O(n)

Table 1. Comparison between different data structure in term of time and space complexity

The second data structure used is the **dynamic array** which we found out it fits the best for our program since we can dynamically allocate memory for the synonyms using **malloc** and **realloc** functions, So if we compare it with other ADT's like linked list or binary search tree it is the same, while the hash table uses static table size which makes it inefficient for storing synonyms since some words have 2 synonyms while others can have over 1000 synonyms so hash table is excluded, although its searching is **O(1)** but the synonyms are limited (word **set** has the most synonyms in english with 4900 synonyms and O(log(4900)) ≅ O(12) which is equal to O(1) in this case).

The insertion process in the dynamic array is of time complexity **O(1)** because we are inserting at the end, and since we are provided with a dictionary where its synonyms are mostly sorted, we call the insertion sort the first time only

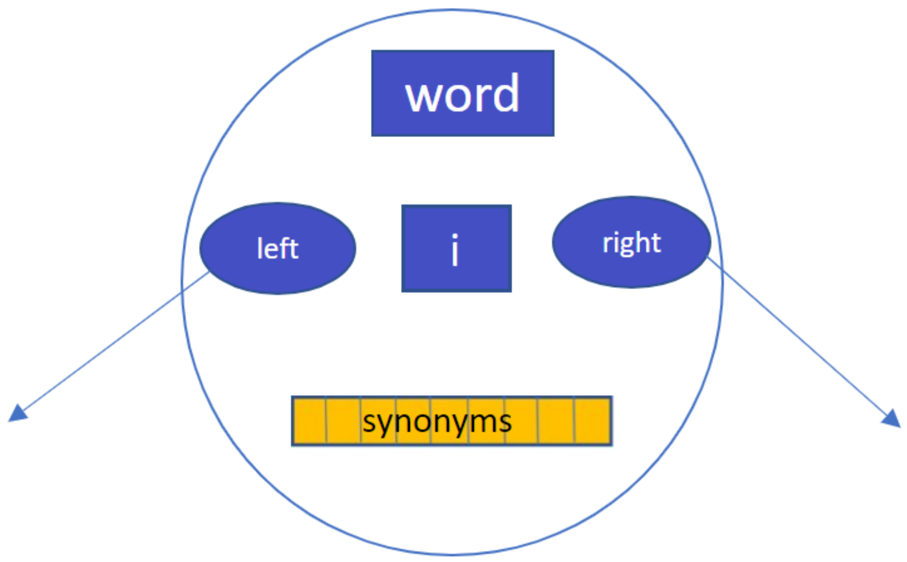
[1] O(logn) since we are using binary search on our array

when we load the main dictionary from *WordsSynonyms.txt* so the complexity will be **O(n * n')** supposing that (n) is the number of the words and (n') is the number of synonyms in each array. This complexity will hold for the first time only, then after loading the new file *NewDictionary.txt* the synonyms are going to be sorted every time.

This insertion process in the dynamic array is better than the one of the binary search tree **O(logn)** and since the words are mostly sorted so the searching will be around **O(n)**, so to solve this problem we have to balance the tree to keep the searching **O(logn)** so binary search tree is excluded.

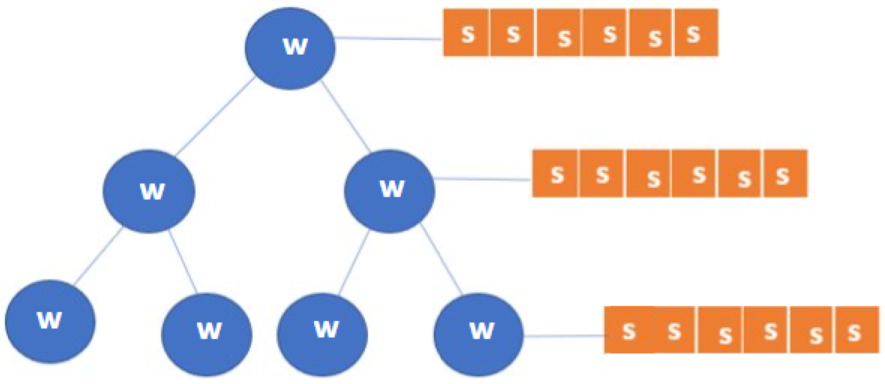
Dynamic array is similar to linked list in many features (insertion, space) but searching a linked list is **O(n)** by using linear search while in dynamic array is **O(logn)** if we use a binary search so we exclude the linked list and finally choose the dynamic array as the ADT that carries the synonyms of the words in our dictionary.

So the structure of each node of the binary search tree consists of a word of type **char*** (struct elt), dynamic array **synonym** of type **elt***, variable **i** of type **int** representing the size of the array, and finally pointers **left** and **right** of the tree. The node is represented as follows:



Node of the binary search tree

The final structure of the dictionary will be as follows:



Third data structure used is the **Hash table** to store all the synonyms of the dictionary and return the most repeated synonym. Our choice was based on the time complexity needed by **brute force** to iterate each word in the array or any other ADT which leads to complexity **$O(n^2)$**, note that we are dealing with all the synonyms of the dictionary so this will affect the searching process time a lot.
On the other hand, a hash table stores all the words with **$O(1)$** insertion and creates a counter for each word that increments when we find the word again. Finally we iterate the table with complexity **$O(n)$** and find the maximum counter then return its word.

So if we take the scalability of the algorithm into consideration, in case we want to simulate a real dictionary that includes about a million words with an average of 20 synonyms for each word, we will find ourselves iterating 20 million iterations to find the most repeated synonym. Using brute force will leave us at complexity of **$O((20\text{million})^2)$** and that is a bad idea. Whereas using the method of hashtable, complexity will be **O(20 million)** which is the best solution we can have in such a scenario.

4.2 Loading the main dictionary into the binary search tree

After taking the first look on the file ***WordsSynonyms.txt*** we notice that every line is in the form of a word and its synonyms separated by (-) and the synonyms are separated from each others by (,) we wanted to extract the words to store them in the binary search tree and extract the synonyms to store them in the dynamic array at their correct node in the tree.

If we take a closer look to the function **load_main_file_into_bst(BST t, char* filePath)** it takes the tree and the file path as arguments. It opens the file and reads line by line till the end of file is reached. Inorder to extract the words alone and the synonyms alone we used a function called **strtok(char* str, delimiters)** this function is included in the library **<string.h>** and it splits a given string according to the delimiters we pass it. In our case we used the comma, space, and dash as delimiters, and we inserted the word to the tree using the function **insert_word_from_main_file(BST t, char* w)**, then take the synonyms by looping on them and store them in their dynamic array using function **insert_synonym(BST t, char* w, char*s)**. Also we used a function called **new_line_removal** which removes the **\n** from the content of the last synonym of each word at the end of the line in the file, then we call the **insertion_sort** function to sort the array. We repeat this process till the end of the file is reached. Finally we have our binary search tree ready with all the synonyms sorted in the dynamic array at their correct position.

4.3 Balancing the binary search tree

When inserting a word using the function **insert_word_from_main_file,** which does not include a call for the function **Balance_tree,** then when loading the file ***WordsSynonyms.txt*** the tree will not be balanced. After storing the tree in the file ***NewDictionary.txt*** and loading it every time, we use the functions **insert_word_from_new_file** that includes a call for function **Balanace_tree,** the tree will always be completely balanced to lower the complexity when traversing through its nodes. We achieved a balanced tree after each insertion by calculating the height using the function **height_of_tree** of the root's left tree and root's right tree by taking the max depth of their left and right subtrees recursively. After the calculations we subtract them, if the absolute value of their difference is greater than 1 and the height of the left subtree is greater than that of the right subtree, we rotate the tree to the right using the function **right_rotate**. The rotation to the right is done by disconnecting the root **X** of the tree from its left subtree. Then the root **Y** of the left subtree is the new root of the main tree, and in case **Y** has a right node connected to it we disconnect it and reconnect it to the left node of **X** and after that **X** is connected to the right node of **Y**. If the absolute value of the difference between the heights of the left and the right subtree is greater than 1 and the height of the right subtree is greater than that of the left subtree, we rotate the tree to the left by the function **left_rotate**. The rotation to the left works the same as the rotation to the right. After that we recursively balance the right subtree and the left subtree using the function **balance_tree** until the whole binary search tree is fully balanced.

4.4 Features implementation

A. First feature is searching for a specific word in the dictionary and for that we used the function **word_exist(BST t, char* s)** this function takes a string and searches the binary search tree recursively and returns 1 if the word exists and 0 otherwise the complexity for searching os **O(logn)**.

B. Second feature is printing the synonyms of a given word, for this one we used the function **print_synonyms_of_word(BST t, char* s)** which takes a string, searches for it using the **word_exist** function, and prints all its synonyms, this function has the same complexity of **word_exist** plus **O(n')** for printing the synonyms.

C. Third feature is inserting a word and its synonym to the dictionary. We use the function **insert_synonym_into_dictionary(BST t, char* w, char* s)** which takes a word and a synonym. The function searches if the word exists in the tree, if yes it inserts the synonym among its synonyms, sorts the array and saves the word and the synonym in the file ***AddedSynonyms.txt***. Otherwise if the word does not exist we insert the word to the tree and insert its synonym too and save the word with its synonym in the file ***AddedSynonyms.txt***. The complexity for inserting is **O(logn)** for searching for the word in the tree and **O(n')** for inserting the synonym among its synonyms (n' is the size of the array we are inserting the synonym to) so the complexity as overall is **O(logn)+O(n')**. Our choice to save the words and the synonyms inserted by the user in a separate file is to lower the complexity as much as we can, becuase the only other way is to insert the word and the synonym to the tree and open the file ***NewDictionary.txt*** in write mode and save the content of the tree (words and their synonyms) every time we insert a word(in this case we are deleting and saving the whole binary search tree every time we insert a word), while instead in our case we just save them in a separate file and load both files ***NewDictionary.txt*** and ***AddedSynonyms.txt*** using **load_new_file_into_BST** and **load_temp_file_into_BST** respectively.

D. Fourth feature is finding the word that has the most number of synonyms. First we created a function called **find_max** that traverses the tree and returns the maximum **t->i** (t->i represents the size of each array in each node of the tree), then we call this function in **word_with_most_synonyms** which takes the maximum and traverses the tree recursively and prints the word where this maximum belongs to, the complexity overall is **O(2*n)** since we traversed the tree twice in a row.

E. Fifth feature is taking a synonym as input from the user and printing all the words of this synonym. This feature demands searching all the synonyms in the dictionary, so we used the function **print_words_of_synonym** that takes a synonym and the binary search tree as parameters, it traverses the tree and uses **binary searching** on each array and when it finds the synonym it prints its word, until it reaches end of the tree. This process costs **O(n)** for traversing the tree and **O(logn')** for searching in the arrays, so the time complexity will be **O(nlogn')**.

F. Sixth feature is finding the most repeated synonym in the dictionary. This feature requires a hash table for better optimised program, where we created a hash table using **ht_create** then used a hashcode for each synonym (value = value * 37 + key[i]) using the function **hash** then store all the synonyms with a counter for each synonym in the hash table using **ht_set,** we increment the counter every time we insert the same word to the table, and finally we iterate the hash table and print the synonym that has the maximum counter.(The complexity in this part is explained briefly in 4.1 last paragraph).

4.5 Additional features implementation

A. First additional feature, is checking if the binary search tree is balanced or not, by using the function **tree_is_balanced** that calculates the height of both subtrees left and right and checks if the absolute value of their difference is greater than one then the tree is unbalanced, while it recursively checks both subtrees as long as the difference of their heights is less than or equal one, then the tree is said to be balanced.

B. Second additional feature Levenshtein algorithm, which is an algorithm that calculates the minimum edit distance between 2 strings, this algorithm uses dynamic programming inorder to be implemented, it compares 2 strings and returns the number of operations needed to be applied on one of the strings so they become similar. The operations that this algorithm holds are insert, delete and replace. In our program we implemented this algorithm and called it when the user searches if a word exists in the dictionary. If the user enters a word that does not exist, we calculate the edit distance using the functions **distance** and **suggestion** between his word and the words of the dictionary. If the distance is less than 2(the word needs 2 operations or less to become similar to one of the words in the dictionary) we suggest the words for the user in case he misspelled the word. The complexity in this case in **O(n)** as we only traversed the binary search tree once.

C. Third additional feature is the history, where the user can check the search history. Every time the user searches if a word exists in the dictionary, his input is saved in a file called ***history.txt*** and the content of the file is printed on the screen whenever he chooses to view the history.

4.6 Saving the dictionary sorted into new file

After loading the dictionary from the main file we were provided with, and inserting the data to the binary search tree for the first time, and after implementing all the features mentioned above. We store the dictionary in a new file called ***NewDictionary.txt***. We traversed the binary search tree, took the synonyms of each word and concatenate them into a single string using the function **synonyms_concatenating** then we concatenate their word with the string(the array of synonyms) using the function **word_and_synonyms_concatenating.** Then we stored the final string in an inorder way(storing from the binary search tree from bottom left all the way to the bottom right) using the function **store_line_into_dictionary** then we did this process recursively in the function **store_new_dictionary** so the words will be sorted and stored in an alphabetical order.

After storing the dictionary in the new file, we include this new file using the function **load_new_file_into_BST** and we start loading the dictionary from ***NewDictionary.txt*** every time we run the program instead of ***WordsSynonym.txt***, so we lower the complexity by loading the words and their synonyms sorted then there is no need to sort them again.

5. Possible future improvements

We can improve the time complexity further more by creating an index table and we break down the binary search tree into binary search subtrees according to their alphabetical order, we link every subtree to the index table, so for example I want to search for word “cat”, i skip indexes ‘a’ and ‘b’ and jump strictly towards ‘c’ and search for the word. The complexity of this process will cost **O(1)** for accessing the index table and **O(logn)** for searching in the subtree and n is the number of words in this subtree.

6. Difficulties and bugs encountered during the implementation

➔ Working with the binary search tree made most of our functions limited with the idea of recursion, where some codes cannot be implemented easily and directly.

➔ Dealing with files since they are limited in some ways if we try to edit, insert or delete data in them.

➔ While we were testing the program for the last time, hours before the deadline, we encountered a bug when storing the data into the new file ***NewDictioanry.txt***. We tried to balance the tree obtained after loading the main file ***WordSynonyms.txt*** , we achieved the balance but failed to store all the data in the new file ***NewDictionary.txt*** for an unknown reason. After debugging the code we decided to balance only the tree obtained after loading the new file ***NewDictionary.txt***.

➔ The program sometimes crashes on start for unknown reasons!

7. Conclusion

In this project we worked on simulating a dictionary by using the appropriate data structures where we tried to come out with a very well optimised program, with the least time and space complexity, taking into consideration the scalability of the program. Every ADT and every step was chosen wisely to keep the efficiency no matter the size of the dictionary we are dealing with because we showed that in some places time complexity has the priority over space complexity, while in other cases we found out that space complexity matters more. All these details in order to implement the best dictionary we are able to.