**COMPLEXITY ANALYSIS DOCUMETATION:**

**Task 7 (15%):** Write a document that contains the following content:

1. List all the data structures you use in your program and briefly describe the

characteristics of each data structure (2%).

2. Describe your first search algorithm (implemented based on Task 2) and analyze its

complexity using Big-O notation (Hint: Take the size of idea bank as *m*. Assume that

the number of words in each idea is constant. Use pseudo code or simplified source

code for your analysis rather than pure text description.) (5%)

3. Describe your second search algorithm (implemented based on indexing) and analyze

its complexity using Big-O notation (Hint: take the number of total words in your

AVL tree as *n*. Use pseudo code or simplified source code for your analysis rather

than pure text description.) (5%)

4. Justifies your selection of data structures based on your complexity analysis (Hint:

you may assume that if I had used another data structure, the complexity of searching

would be higher. You must justify your claims.) (3%)

1. Data structures used in the program are:
2. List:

Lists are sequence containers. They allow insert and remove with a complexity of O(1). They enable iteration through iterators and do not have random access to its elements. List elements can be anywhere in memory. Each element is linked to the next by pointing to the element using its address in memory.

1. Unordered map:

Associative containers that store key and data pair. Maps allow fast retrieval of individual elements (data) using their keys with a time complexity of O(1) on average.

1. Avl tree:

It is a balanced binary search tree. Avl tree are always balanced with a maximum height difference of 1 between its left and right subtree. Every time new data is inserted or deleted, the tree goes through the rebalancing processes from bottom up rotating the tree left or right where, and if, needed. Retrieval of data has a time complexity of O(log n).

1. Set:

Stores unique elements. The elements in the set are ordered. It has logarithmic time complexity for insertion.

1. ‘searchWord’ function searches if a word is present in an Idea. Word to be searched for, passed in as the parameter, will be referred to as ‘WORD’.

(Idea.h: Line 26)

searchWord:

1 for each keyword: O(x)

2 compare keyword with WORD O(z)

3 return true if match O(1)

4 search content for word using ‘find’ function from string class O(y.z)

5 if found return true O(1)

6

7 return false O(1)

Assumptions: For simplicity, we will consider all Ideas has the same number of words on average, and each word has, again on average, ‘z’ characters. Consider there are ‘x’ number of keywords and ‘y’ number of words in content. Once again, ‘x’, ’y’ and ‘z’ will be considered constant for all Ideas throughout the program.

Analysis:

Lines 1-3: Loop has a complexity of O(x.z)

For each keyword compare function has to compare each character of both words

Lines 4-5: Compliexity of O(y.z)

The find function matches each character of all words in the content until match found. Worst case scenario is word not present and so number of characters times words

Overall, considering all our assumptions, the ‘searchWord’ function has a time complexity of O(xz+yz). Since we are considering x, y and z to be constants for all Ideas we can say the complexity is O(w) where ‘w’ is the number of total words in the idea.

The above function in Idea class is used in the search algorithm.  
Algorithm to seach word directly for Ideas Bank:

(IdeasBank.h: Line 183)

bonus\_search\_bank:

1 Get line from user -|

2 determine if there are boolean operator and multiple words > O(1)

3 set signal of AND operator and separate search words -|

4

5 For each Idea in bank:

6 check if matches the first WORD using searchWord function O(w)

7 if there is a second word

8 check if matches the second WORD using searchWord function O(w)

9 if condition is AND and two words found

10 add Idea ID to list

11 if condition not AND and at least 1 word found

12 add Idea ID to list

13 return list

Assumption: Number of words to be searched for with Either AND or OR is no greater than two.

Analysis:

Lines 1-3 : a set of operations the runs only once per call, so complexity O(1).

Lines 5-12:

In the main loop that runs ‘m’ times, m being the size of the bank, the searchWord function will be called twice in the worst case scenario giving us O(m(2w))

Overall, simplifying the equation we get O(m.w) where m is the number of Ideas in the bank and w number of words in the idea.

1. The search function uses the ‘findIndex’ function which in turn calls the AVL\_Retrieve function form the provided AVL\_ADT class.

The findIndex function call the retrieve function of the AVL tree and returns true if key found or false if not found.

The AVL\_Retrieve function has a complexity of O(log n) where n is the number of nodes in the tree. As AVL are always balanced, on each call the search space gets halved everytime.

(IdeasBank.h: Line 259)

bonus\_search\_tree:

1 Get line from user -|

2 determine if there are boolean operator and multiple words > O(1)

3 set signal of AND operator and separate search words -|

4

5 check tree for first word using findIndex O(log n)

6 if there is a second word

7 check tree for second word using findIndex O(log n)

8

9 if condition is AND and two words found

10 cross check both lists for matching IDs and add to set O(v.u.log x)

11 else

12 add ID list of first word to set O(vlog v)

13 add ID list of second word to set O(ulog u)

14 return set

Assumptions: on a big enough Ideas Bank database: on average, most words will have similar number of relevant ID

Analysis:

Line 5 and 7 has logarithmic complexity. The Avl tree node is copied to a buffer which has a complexity of O(1)

Line 10: nested for each loops for the ID list of the two search words. V is the number of ID is the first word list and u is the number of IDs in the second list. So complexity is O(u.v.log x), x is the number of matching IDs in the two lists

Lines 12 and 13 runs for each loop for each words ID lists thus having the complexities given above in the worst case.

Overall, summing up the operations we get O(2log n + u.v.logx) at worst case. Simplifying the equation we get the complexity in the worst case as O(u.v.log x) where u and v are size of match lists of each search words and x is the total number of matches.

1. Unordered map: This has been chosen as the data structure for the Idea bank to enable fast searching of Ideas using ID from the bank. Maps gives an average complexity of O(1) for insertion and retrieval. Throughout the program most function rely on searching for and Idea using ID and map gives us the fastest way of doing it.

List: Has constant complexity for insertion and deletion. This was used as the data structure for storing idea IDs in the tree because having such small cost for these operations. Updating the list for a word will be a very frequent action and will be needed every time new ideas are inserted in the bank or if any is deleted. Other structures such as array or vectors has limitations with size or has to be deleted and recreated on every update which is very computationally heavy.

AVL tree: This has been chosen for the dictionary as this almost guarantees logarithmic complexity for retrieval of data. Using other tree structures or linear structures may result in the tree being skewered or having to do linear search increasing complexity greatly in both cases.

Set: Set has been used in search function to eliminate duplication of idea IDs. It has logarithmic complexity for insertion. Attempting to eliminate duplicates by cross matching can take complexity up to n\*n in the worst case.

Zarif Jawadul Karim

23/05/2020