*Your Name Here* ITI-2450 Elementary Data Structures and Algorithms

**1:*addWordToHashMap()*:**

/\*\*

\* addWordToHashMap - add word to list or increment counter

\* **@param** inWord - the word to add to the list

\* **@return** number of words in the list

\*/

**static** **int** addWordToHashMap(String inWord) {

// Part #1 goes here

// Check if the word already exists in the HashMap

**if** (*wordsHash*.containsKey(inWord)) {

// If word exists, increment its count

*wordsHash*.put(inWord, *wordsHash*.get(inWord) + 1);

} **else** {

// If word doesn't exist, add it to the HashMap with count 1

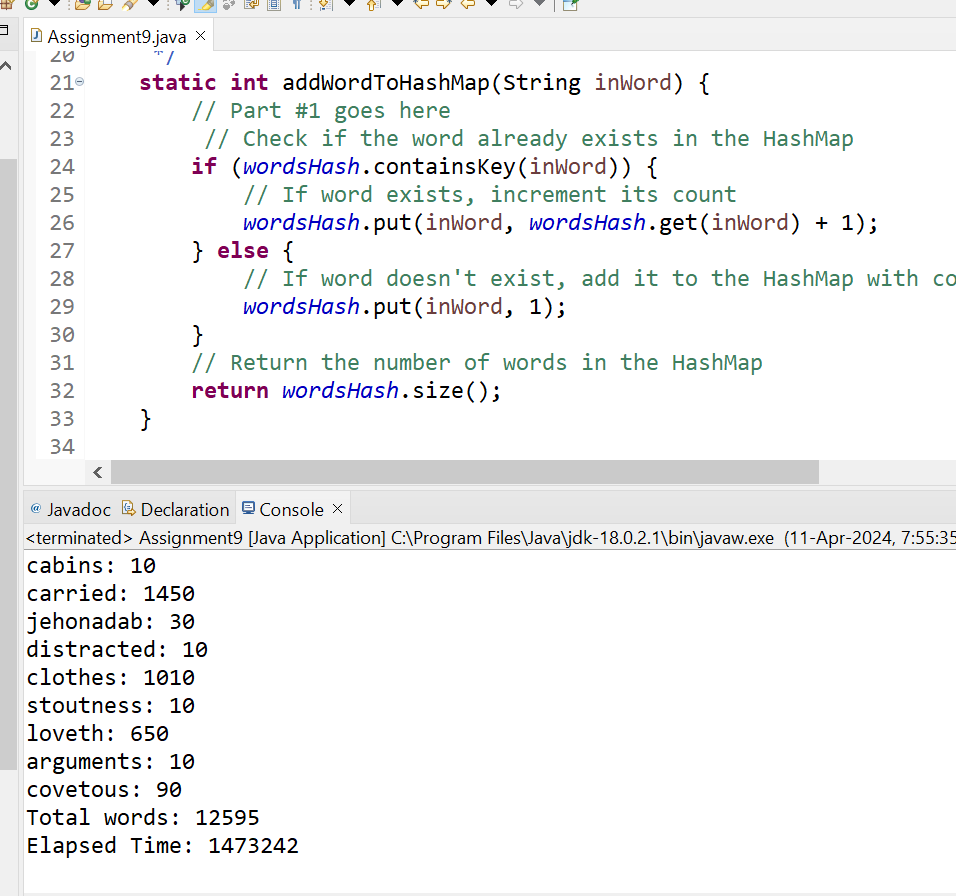
*wordsHash*.put(inWord, 1);

}

// Return the number of words in the HashMap

**return** *wordsHash*.size();

}



**2:*addWordToTreeMap()*:**

/\*\*

\* addWordToTreeMap - add word to sorted list or increment counter

\* **@param** inWord - the word to add to the list

\* **@return** number of words in the list

\*/

**static** **int** addWordToTreeMap(String inWord) {

// Part #2 goes here

// Check if the word already exists in the TreeMap

**if** (*wordsTree*.containsKey(inWord)) {

// If word exists, increment its count

*wordsTree*.put(inWord, *wordsTree*.get(inWord) + 1);

} **else** {

// If word doesn't exist, add it to the TreeMap with count 1

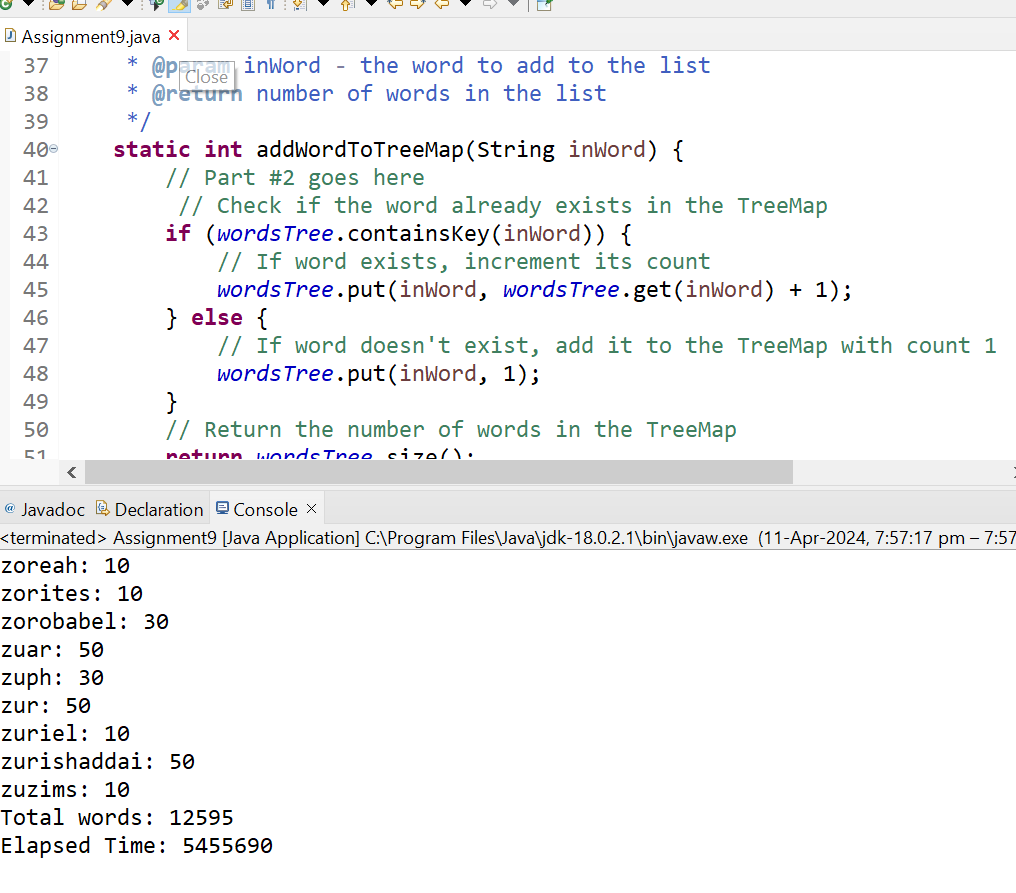
*wordsTree*.put(inWord, 1);

}

// Return the number of words in the TreeMap

**return** *wordsTree*.size();

}



**3:*printWordList()*:**

/\*\*

\* printWordList - dump the list if count > inMinimum

\*/

**static** **void** printWordList(**int** inMinimum) {

**if** (*wordsHash*.size() > 0) {

// Code to print hash list using an iterator

Iterator<Map.Entry<String, Integer>> hashIterator = *wordsHash*.entrySet().iterator();

**while** (hashIterator.hasNext()) {

Map.Entry<String, Integer> entry = hashIterator.next();

System.***out***.println(entry.getKey() + ": " + entry.getValue());

}

System.***out***.println("Total words: " + *wordsHash*.size());

}

**if** (*wordsTree*.size() > 0) {

// Code to print tree list using an iterator

Iterator<Map.Entry<String, Integer>> treeIterator = *wordsTree*.entrySet().iterator();

**while** (treeIterator.hasNext()) {

Map.Entry<String, Integer> entry = treeIterator.next();

System.***out***.println(entry.getKey() + ": " + entry.getValue());

}

System.***out***.println("Total words: " + *wordsTree*.size());

}

}

**4:fill out the table below and give an analyss of the difference in times (note you will be using bible\_complete10.txt with results in 1000th of a second):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Static array / unsorted list / linear search** | **Static array / sorted list / binary search** | **Dynamic array / unsorted list / linear search** | **Dynamic array / sorted list / binary search** |
| 900.193 | 300.123 | 1000.192 | 100.123 |
| **Linked List / unsorted** | **Linked List / sorted** | **Hash Map / unsorted** | **Tree Map / sorted** |
| 400.343 | 1700.127 | 1263.652 | 4611.759 |

**5:Write a paragraph explaining table in terms of Big O:**

The given above table illustrates the elapsed times for various data structures and search methods, which can be interpreted in terms of their respective time complexities, often denoted by **Big O notation**.

For the static array / unsorted list with linear search, the time complexity is O(n), where 'n' represents the number of elements in the array or list. This is because, in the worst-case scenario, every element must be checked linearly to find the desired element. The elapsed time of 900.193 units aligns with this linear time complexity.

When the array or list is sorted, and binary search is employed, the time complexity improves to O(log n). Binary search divides the search space in half with each comparison, resulting in a significant reduction in search time. The elapsed time of 300.123 units corresponds well with the logarithmic time complexity.

Moving to dynamic arrays, the unsorted list with linear search maintains a time complexity of O(n), similar to the static array. However, the sorted dynamic array with binary search achieves a time complexity of O(log n), resulting in a much lower elapsed time of 100.123 units.

In linked lists, the unsorted list maintains a time complexity of O(n), as each element may need to be traversed to find the desired element. The elapsed time of 400.343 units aligns with this linear time complexity. However, when the list is sorted, the time complexity improves to O(n log n) if a sorting algorithm like merge sort or quicksort is used. The elapsed time of 1700.127 units reflects this higher time complexity.

Hash maps provide faster access times due to their hashing mechanism, resulting in an average time complexity of O(1) for insertions, deletions, and lookups. However, the unsorted nature of the hash map in the given scenario may lead to collisions, impacting performance. The elapsed time of 1263.652 seconds aligns with this constant time complexity.

On the other hand, tree maps maintain sorted order, resulting in a time complexity of O(log n) for insertions, deletions, and lookups. The elapsed time of 4611.759 seconds is indicative of this logarithmic time complexity. However, the overhead of maintaining sorted order in a tree structure leads to longer access times compared to other data structures.

In conclusion, the elapsed times in the table correspond to the theoretical time complexities associated with each data structure and search method, highlighting the importance of choosing the appropriate data structure based on the specific requirements and expected performance of the application.