GTU Department of Computer Engineering
CSE 222/505 - Spring 2023
Homework 6 Report

Mehmet Mete Şamlıoğlu
200104004093

1-) SYSTEM REQUIREMENTS

The system requires an algorithm that first builds a LinkedHashMap structure after processing the input string. The map structure will be built according to the preprocess input string. After building the map properly, our main task is to sort the map entries by using the count(number of words) value, in cases where the frequency of count values are the same, the second condition that is going to be evaluated is the input order of the key. To perform this operation, it is required to use three classes, one for building the tree, the second one is to perform merge sort. And the very last one is to store the words and the count value for each entry on the map. In the mergeSort class, I used ArrayList to store the entries of the map. As was mentioned in the PDF I used the merge sort algorithm to perform the sorting operation, I wrote my own merge sort algorithm to perform the merge sort on the map structures entries. In the end, I add a method to return a sorted map in the mergeSort class. In the end, I obtained both sorted and unsorted map objects where the mergesort object has been instantiated.

2-) PROBLEM-SOLUTION APPROACH

In this part, I will explain each method of required classes one by one to explain in detail. In the end, all the algorithms will be explained and clarified.

myMap Class methods

- **Constructor public myMap(String input)**

The constructor takes the string as an input and processes the string by calling preprocessString method, after that it creates mapStructer by calling buildMapStructer in it.

- **Default constructor myMap( )**

It creates a default myMap object with a null string input value, mapSize 0 and, not instantiated LinkedHashMap structure.

- **Public void preprocessString (String text)**

The very first thing it performs is replacing capital letters with lowercase letters by calling toLowerCase() method. The second necessary thing is replacing all the letters that are not space or a lowercase letter. To perform this I used a method from the regex library which is replaceAll. In the end, I assigned the preprocessed string to str which is a string that is going to be used to build the map.

- **Public buildMapStructer(String text)**

In this method, firstly I split the array by removing empty spaces and putting each word into the one-dimensional String array which is String words[]. After that, I iterated each word of the words[] array in a for each loop, and in each iteration, I controlled the letters of the word by checking the map structure by using a letter as a key of the map. If it already exists I pushed the related word to the info object, if it does not exist in the tree I created an entry with the key as a letter and a new object info. By iterating the whole words array I pushed each letter of the array as a occurred key of map and value as all the words that the letter occurred in it. The time complexity of the buildMapStructer function was O(n^2), since I used two for a loop.

- **public void printMap( )**

This method prints the map structure by iterating the entries in order. I simply used a for each loop to iterate all the keys(letters) in the map, and in each iteration, I printed all the information that is stored in that entry. For example, if letter a contains 4 words in info class I displayed every one of them in each iteration. In the end, each info object that is related to that key will be displayed properly. The time complexity of this method is O(n^2) since I used two for a loop.

info Class methods

It has one constructor which is public ( ) that instantiates an object with int count = 0 value and String words[1] string array.

- **Public void push(String word)**

It basically pushed the parameter string to the String array words and stores in it, after each push operation the count increases. Increasing count value is important because this information is going to be used for later use when we sort the map entries.

The rest of the methods are to return words array and count value.

mergeSort class

- **public mergeSort(myMap mapObject)**

It has only one constructor which is the one above, it assigns the parameter myMap object to the private myMap originalMap variable in data fields, and also I assign the LinkedHashMap data container of the parameter myMap to a LinkedHasMap map in the data field to use in the sorting stage. After the assignments, it instantiates the myMap sortedMap object from the data field and calls mergeSort method to sort the entries of originalMap and put them into sortedMap. The class does not have any other constructor because without an original myMap, there is nothing to sort.

- **public myMap getSortedMap( )**

It was necessary to return sortedMap to the part of the code where mergeSort class is instantiated. Without this method, there is no other way to return the sortedMap object.

- **public int getIndex(Map.Entry<String, info> entry)**

This method is a helper method for public boolean hasSeenBefore(). It returns the index of parameter entry in LinkedHashMap. As It was mentioned in the PDF if the frequency of count values is the same the second thing that is going to be evaluated is the input order of the key. So it basically finds the index of entry according to input order. The return value of this method is going to be used in hasSeenBefore function.

- **Public boolean hasSeenBefore(entry1 , entry2)**

This method is used in cases where the frequency of count values of two entries is the same. To decide which is going to be put into sortedMap it checks the input order of entry and decides. It iterates the words of each entry in the same for loop, whenever two different words are found it checks the input order of these two occurrences by calling getIndex method. So if the obtained index of entry1 is smaller than entry2's index it will return true which means that the entry1 will be put into the map before entry2, if entry2 has been seen before entry1 it will return false, which means that entry2 will be put into the map before than entry1.

- **public void mergeSortMap( )**

This method provokes the mergeSort method with int left = 0 and int right = map.size( ) – 1 values and sorts the ArrayList which contains the entries of LinkedHashMap according to the count values. In the end, the aux ArrayList contains sorted map entries. In the last step, a loop iterates through ArrayList's elements and puts each entry to the myMap sorted map which is a variable of the mergeSort class. In the end, we obtained originalMap and sortedMap inside of the mergeSort class.

- **public void mergeSort(ArrayList<> aux, int left, int right)**

It is basically sorting the entries of aux by dividing an ArrayList into smaller subArrayLists, and after that, it is sorting each small subarray. In the end, it is merging the sorted subArrayLists back together to form the final sorted ArrayList with entries. Simply, the sorting mechanism is to divide the ArrayList into two halves, sort each half, and in the end merge the sorted halves back together and continue

doing it until the entire array is sorted. The time complexity of mergeSort is O(nlogn) so if the array is large it becomes an efficient way to sort it. To create every combination of subArrays it requires a recursive method which is this. In each recursive calls it calculates the middle point by doing (left + right)/2 and after that it makes two recursive calls first one checks the left part of the ArrayList(left to the middle), and the second one checks the right part of the ArrayList (middle + 1 to right). In the end, it calls the sort() method to sort each created subArrayLists and merges them in this method. So by doing two recursive calls and one last call to sort() method it performs the sorting operation.

- **public void sort(ArrayList<>entires , int left, int middle, int right)**

To sort each sub ArrayList that is created by the recursive mergeSort method this method has been used. So it basically gets the information of which part of the ArrayList is going to be sorted from the mergeSort method as an int left, int middle, and right variable and it sorts that particular part of the ArrayList. In a for loop whose index is stating from int left parameter it iterates the loop until one of the size of either left part or right part has been reached. In each iteration, it checks if the count value of the left entry is smaller or greater than the right entry's count value and also it checks whether the frequency of count values are equal or not. So if the count value of the left entry is smaller than the right entry's it puts the LeftEntry before, if the condition is the opposite it puts the right entry before. The challenging problem is to decide which one will be put first when the count frequencies are exactly the same. In this case, it calls the hasSeenBefore method and passes entry1 and entry2 to this method. As was mentioned above in the hasSeenBefore method part, this method returns true if entry1 occurs before entry2 in terms of input order, and false otherwise.

So If the return value of this method is true it puts the left entry to aux before, if the return value is false it puts the right entry before. At the end of this loop the sorting will be performed, to put the entries that have not been reached there are two consecutive while loops after the ending of this for loop to put the rest of the entries of left and right entries to the ArrayList if there are any.

Summation:

So whenever a myMap object is instantiated with a string, first the string will be preprocessed, after that myMap object will build a tree by using this preprocessed string. After that, If the mergeSort object is instantiated with this myMap it will be sorted by using mergeSort. In this mergeSort object you can find both the sorted map and the original unsortedMap. To obtain a reference of the sorted myMap object use getSortedMap( ), to get an unsorted original map use getUnsortedMap( ).

3-) Test Cases

```
public class TestClass
{




    public static void main(String[] args)
    {

        String input = "Buzzing bees buzz.";
        //String input = "'Hush, hush!' whispered the rushing wind."

        myMap hashmap = new myMap(input);
        System.out.printf("The original (unsorted) map:\n");
        hashmap.printMap( );

        mergeSort mergeSortmyMap = new mergeSort(hashmap);
        myMap sortedMap = mergeSortmyMap.getSortedMap( );

        System.out.printf("\n\n");
        System.out.printf("The sorted map:\n");
        sortedMap.printMap( );
    }
}
```

4-) Running Results

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw6$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw6$ java Driver
Original String: Buzzing bees buzz.
Preprocessed String: buzzing bees buzz


The original (unsorted) map:
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
Letter: i - Count: 1 - Words: [buzzing]
Letter: n - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: s - Count: 1 - Words: [bees]


The sorted map:
Letter: i - Count: 1 - Words: [buzzing]
Letter: n - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: s - Count: 1 - Words: [bees]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
meterose@DESKTOP-3HDDHUD:/mnt/c/hw6$
```

```
meterose@DESKTOP-3HDDH

The original (unsorted) map:
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: p - Count: 1 - Words: [whispered]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: t - Count: 1 - Words: [the]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: g - Count: 1 - Words: [rushing]


The sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
meterose@DESKTOP-3HDDHUD:/mnt/c/hw6$
```