# GEBZE TECHNICAL UNIVERSITY
# CSE222 DATA STRUCTURE
# HOMEWORK 6 REPORT

**225008003102**

**SÜHA BERK KUKUK**

## 1) Info Class

```java
import java.util.ArrayList;

public class info
{
    protected int count;
    protected ArrayList<String> words;

    public info()
          {
              this.count = 0;
              this.words = new ArrayList<>();
          }

          public info(String newWord)
          {
              this.count++;
              this.words = new ArrayList<>();
              this.words.add(newWord);
          }

          public void push(String newWord)
          {
              this.count++;
              this.words.add(newWord);

          }

          public int size()
          {
              return count;
          }

          public ArrayList<String> getWords()
          {
              return words;
          }

          public void setSize(int newCount)
          {
              this.count = newCount;
          }

          @Override
          public String toString()
          {

            String res = "Count: " +this.count + " - " + "Words: " + this.words;
            return res;
        }

    }
```

The "info" class represents a data structure that holds information about a count and a collection of words. The class provides various methods to manipulate and retrieve the data stored within the object.

1. Data Fields:
    - **count**: An integer that represents the count of words stored in the object.
    - **words**: An ArrayList of strings that holds the words.
2. Constructors:
    - **info()**: A default constructor that initializes the count to 0 and creates an empty ArrayList for words.
    - **info(String newWord)**: A constructor that takes a new word as a parameter, sets the count to 1, and initializes the words ArrayList with the new word.
3. Methods:
    - **push(String newWord)**: This method adds a new word to the words ArrayList and increments the count by 1.
    - **size()**: This method returns the current count of words stored in the object.
    - **getWords()**: This method returns the ArrayList of words.
    - **setSize(int newCount)**: This method allows setting a new value for the count.
    - **toString()**: This method overrides the default **toString()** method to provide a string representation of the object. It returns a formatted string displaying the count and the words.

Overall, the "info" class provides a simple structure for storing word-related information. It allows adding new words, retrieving the count and words, and modifying the count if necessary. The **toString()** method provides a convenient way to obtain a string representation of the object's data. I did not check word already in the words arraylist. Because i check in create Hashmap. We will see more detail in relevant part. I use override because when we use **toString()** method, it is easier to applied all same object we dont need rewrite code for specific object.

2) myMap Class

```
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedHashMap;


public class myMap {

    private int mapSize;
    protected String str;
    protected LinkedHashMap<String,info> map;

     public myMap()
     {
        this.mapSize =0 ;
        this.map = new LinkedHashMap<>();

     }

     public myMap(String str)
     {
        this.str = preprocessing(str);
        this.map = new LinkedHashMap<>();

        List<String> items = createWords();
        List<String> letters = createLetter();

        int index = 0;
        for(String ch:letters)
        {
            if(ch.isBlank())
            {
                index++;
            }

            String word = items.get(index);
            if(word.contains(ch))
            {
```

```
            if(map.containsKey(ch))
            {
            info temp = map.get(ch);
            temp.push(str);
            map.put(ch,temp);
            }

            else
            {
                info newInfo = new info(word);
                map.put(ch, newInfo);
            }
        }

    }
    this.mapSize = map.size();
 }

 protected void putInfo(String key,info newInfo)
 {
    map.put(key, newInfo);
 }

 protected info getInfo(String key)
 {
    return map.get(key);
 }

 //helper Functions

 private List<String> createWords()
 {
    List<String> list = new ArrayList<>();
    int start = 0;
    int end = 0;
    while (end < str.length()) {
        while (end < str.length() && str.charAt(end) != ' ') {
            end++;
        }
        list.add(str.substring(start, end));
        start = end + 1;
        end = start;
 }

 return list;
}

private List<String> createLetter()
{
        char[] charArray = str.toCharArray();
        List<String> list = new ArrayList<>();
        for (char ch : charArray) {
            list.add(Character.toString(ch));
        }
        return list;
    }

    public void printMap()
    {
        System.out.println(this.map.toString());
    }

    protected int getCount(String key)
    {
        info temp = map.get(key);
        return temp.size();
    }

    protected String[] getKeys()
    {
        String[] keys = map.keySet().toArray(new String[0]);
        return keys;
    }

    protected boolean isEmpty()
    {
        if(map.isEmpty())
        {
            return true;
        }
        return false;
    }

    protected int mapSize()
    {
        return this.mapSize;
    }

    @Override
    public String toString()
    {
        String res="";
        String[] keys = map.keySet().toArray(new String[0]);
        for(String key:keys)
        {
            String temp = "Letter: " + key + " - " +
map.get(key).toString() + "\n";
            res += temp;
        }

        return res;
    }

    private String preprocessing(String str)
    {
        if(str.isEmpty())
        {
            System.out.println("The stirng is empty. You will get
error");
        }
        else if(str.matches("\\d+"))
        {
            System.out.println("The string just contains number. You
will not run this program with strinh that contains just numbers.");
        }
        else
        {
        String lowercaseString = str.toLowerCase();
        System.out.println("Original String: " +str);
        String processedString = lowercaseString.replaceAll("[^a-z ]",
"");
        System.out.println("Prepocessing String: " +processedString);
        return processedString;
        }
        return "";

    }
}
```

The "myMap" class represents a custom map data structure that associates keys with information objects. It provides functionality to preprocess a string, create a map based on the string, and perform various operations on the map.

1. Data Fields:
    - **mapSize**: An integer representing the number of entries in the map.
    - **str**: A string that holds the original input string for preprocessing.
    - **map**: A LinkedHashMap that stores key-value pairs, where the keys are letters and the values are instances of the "info" class.

2. Constructors:
    - **myMap()**: A default constructor that initializes the mapSize to 0 and creates an empty LinkedHashMap.
    - **myMap(String str)**: A constructor that takes a string as input, preprocesses it, and creates the map based on the preprocessed string.
3. Methods:
    - **putInfo(String key, info newInfo)**: Inserts a key-value pair into the map.
    - **getInfo(String key)**: Retrieves the information object associated with the specified key.
    - **printMap()**: Prints the contents of the map.
    - **getCount(String key)**: Retrieves the count value from the information object associated with the specified key.
    - **getKeys()**: Retrieves an array of all the keys in the map.
    - **isEmpty()**: Checks if the map is empty.
    - **mapSize()**: Retrieves the number of entries in the map.
    - **toString()**: Overrides the default **toString()** method to provide a string representation of the map.
4. Helper Functions:
    - **createWords()**: Creates a list of words by splitting the preprocessed string.
    - **createLetter()**: Creates a list of individual letters from the preprocessed string.
5. Preprocessing:
    - The class provides a private method called **preprocessing(String str)** to preprocess the input string. It converts the string to lowercase, removes non-letter characters except spaces using regular expressions, and returns the processed string.

Overall, the "myMap" class encapsulates the functionality to create and manipulate a map where keys are letters and values are information objects. It supports operations like inserting entries, retrieving information, checking map size and emptiness, and providing a string representation of the map. In the create words array list i used the scenario that the input string contains space while traverse the the string if the charecter is space, i get next word why i do that because i have letters uninuqe so if i would not skip the next word, i will obtain same words again and again.  The class also includes preprocessing functionality to ensure that the input string contains only lowercase letters and spaces. I added preprocessing step in myMap class. I think we create new object from myMap so myMap constructor waits String as input, i would check in this step it is easy for create new object. Also i can check directly without using declare method.

3) mergeSort Class

```
public class mergeSort {

    protected myMap orginalmyMap;
    protected myMap sortedmyMap;
    private String[] aux;


    public mergeSort(myMap newMyMap)
    {
        if(newMyMap.isEmpty() || newMyMap.mapSize()<2)
        {
            return;
        }
        this.orginalmyMap = newMyMap;
        this.aux = orginalmyMap.getKeys();

        int n = aux.length;
        String[] temp = new String[n];
        mergerHelp(aux, 0, n-1, temp);
        createSortedMap();

        System.out.println(orginalmyMap.toString());
        System.out.println(sortedmyMap.toString());
    }

    private void createSortedMap()
    {
        this.sortedmyMap = new myMap();
        for(String key:aux)
        {
            info temp = orginalmyMap.getInfo(key);
            sortedmyMap.putInfo(key, temp);
        }
    }

    private void mergerHelp(String[] aux,int left, int right, String[]
temp)
    {
        if (left >= right) {
            return;
        }

        int mid = left + (right - left) / 2;
        mergerHelp(aux, left, mid, temp);
        mergerHelp(aux, mid+1, right, temp);
        merge(aux, left, mid, right, temp);
    }

    private void merge(String[] aux,int left, int mid, int right,
String[] temp)
    {
        for (int i = left; i <= right; i++) {
            temp[i] = aux[i];
        }

        int i = left;
        int j = mid + 1;
        int k = left;
        while (i <= mid && j <= right) {
            int tempCount1 = orginalmyMap.getCount(temp[i]);
            int tempCount2 = orginalmyMap.getCount(temp[j]);

            if(tempCount1<tempCount2)
            {
                aux[k++]=temp[i++];

            }
            else if(tempCount1==tempCount2)
            {
                if(i<j)
                {
                    aux[k++]=temp[i++];;

                }
                else
                {
                    aux[k++]=temp[j++];;
                }
            }
            else
            {
                aux[k++]=temp[j++];;
            }
        }
        while(i<=mid)
        {
            aux[k++]=temp[i++];;
        }

        while(j<=mid)
        {
            aux[k++]=temp[j++];;
        }

    }

}
```

The mergeSort class is a Java class that implements the merge sort algorithm to sort the entries of a myMap object. A myMap is a custom map data structure that stores a set of keys and their corresponding info objects, which contain a count and a value. The merge sort algorithm sorts the keys in ascending order based on their corresponding count values.

1. Data Fields:
    - **orginalmyMap**: A "myMap" object representing the original map that needs to be sorted.
    - **sortedmyMap**: A "myMap" object representing the sorted map after performing the merge sort.
    - **aux**: An array of strings used as an auxiliary array for sorting.
2. Constructors:
    - **mergeSort(myMap newMyMap)**: A constructor that takes a "myMap" object as input. It checks if the map is empty or contains only one entry. If so, the constructor returns without performing any sorting. Otherwise, it initializes the data fields, including the auxiliary array, and performs the merge sort

algorithm on the keys of the original map. Finally, it creates a sorted map based on the sorted keys.

3. Methods:

    - **createSortedMap()**: A private helper method that creates a sorted map based on the sorted keys obtained from the merge sort.
    - **mergerHelp(String[] aux, int left, int right, String[] temp)**: A recursive helper method that implements the merge sort algorithm. It splits the array of keys into smaller subarrays, sorts them recursively, and merges them back together in sorted order.
    - **merge(String[] aux, int left, int mid, int right, String[] temp)**: A helper method that performs the merging step of the merge sort algorithm. It compares the counts of the keys and merges them into the auxiliary array in ascending order.

The constructor of the mergeSort class takes a myMap object as a parameter, checks if it is empty or has only one entry, and returns if so. Otherwise, it initializes the orginalmyMap and aux instance variables with the given myMap object, creates a temporary String array with the same size as the aux array, and calls the private mergerHelp method to sort the aux array. After sorting, it calls the private createSortedMap method to create the sortedmyMap object using the sorted keys and the corresponding info objects from the orginalmyMap object.

The createSortedMap method initializes the sortedmyMap object as a new empty myMap object, and iterates over the sorted keys in the aux array. For each key, it retrieves the corresponding info object from the orginalmyMap object, and adds the key-value pair to the sortedmyMap object using the putInfo method.

The mergerHelp method is the main recursive method that sorts the aux array using the merge sort algorithm. It takes the aux array, the left and right indices of the subarray to be sorted, and the temporary array as parameters. If the left index is greater than or equal to the right index, the method returns, as there is only one element in the subarray. Otherwise, it calculates the midpoint of the subarray and recursively calls itself to sort the left and right halves of the subarray. After both halves are sorted, it calls the private merge method to merge the two halves into a single sorted subarray.

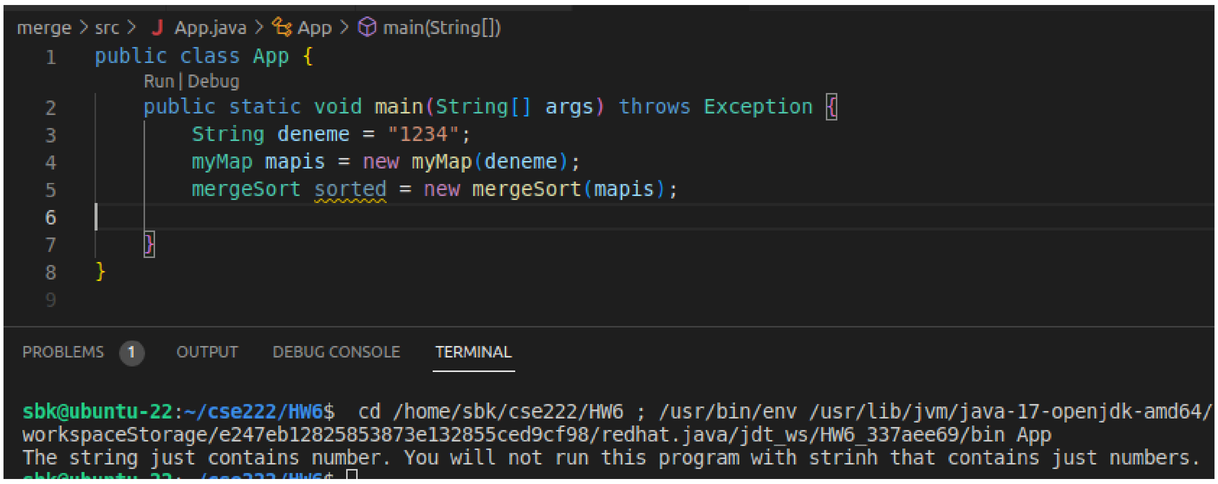
The merge method takes the aux array, the left, midpoint, and right indices of the two halves to be merged, and the temporary array as parameters. It first copies the elements of the aux array within the left and right indices to the temporary array. Then, it initializes three index variables: i for the left half, j for the right half, and k for the merged array. It compares the count values of the info objects corresponding to the keys at positions i and j, and copies the key with the smaller count value to the merged array. If the count values are equal, it copies the key with the smaller index value to the merged array. After one of the halves is exhausted, the method copies the remaining elements of the other half to the merged array.

Time complexity of method which are use for merge algorithm:

1. **createSortedMap() method**: The createSortedMap() method creates a sorted map based on the sorted keys obtained from the merge sort. It iterates over the sorted keys and retrieves the corresponding information from the original map. The time complexity of this method is O(n), where 'n' is the number of keys in the map.
2. **mergerHelp() method**: The mergerHelp() method is a recursive helper method that implements the merge sort algorithm. It splits the array of keys into smaller subarrays, sorts them recursively, and merges them back together in sorted order. The time complexity of this method is O(n log n), where 'n' is the number of keys in the subarray being sorted.
3. **merge() method**: The merge() method performs the merging step of the merge sort algorithm. It compares the counts of the keys and merges them into the auxiliary array in ascending order. The time complexity of this method is O(n), where 'n' is the number of keys being merged.

Overall, the time complexity of the "mergeSort" class is dominated by the time complexity of the merge sort algorithm, which is O(n log n). The additional operations performed in the class, such as creating a sorted map and merging the keys, have time complexities of O(n).

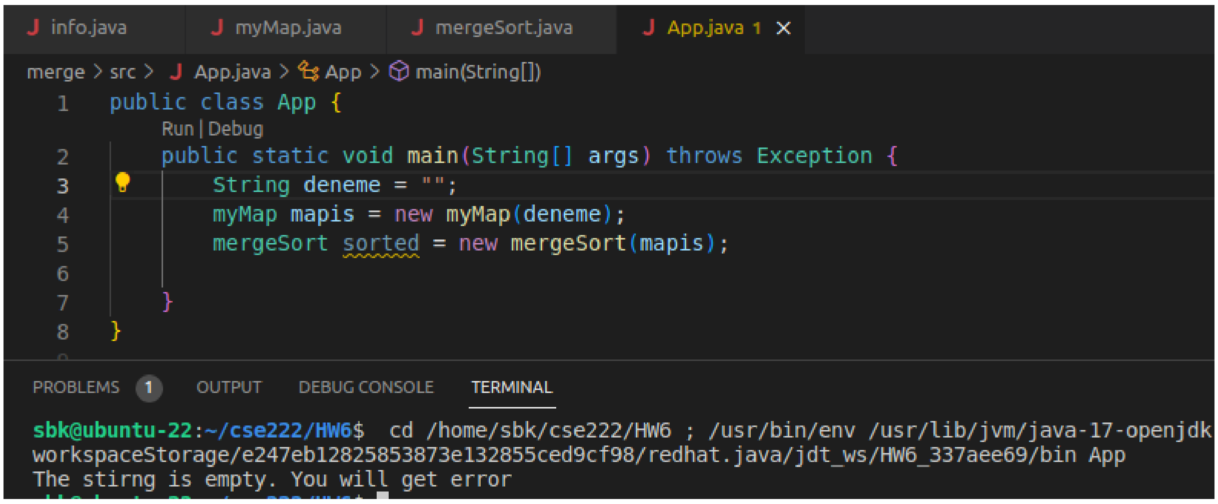
Some Examples :



```java
public class App {
    public static void main(String[] args) throws Exception {
        String deneme = "1234";
        myMap mapis = new myMap(deneme);
        mergeSort sorted = new mergeSort(mapis);

    }
}
```

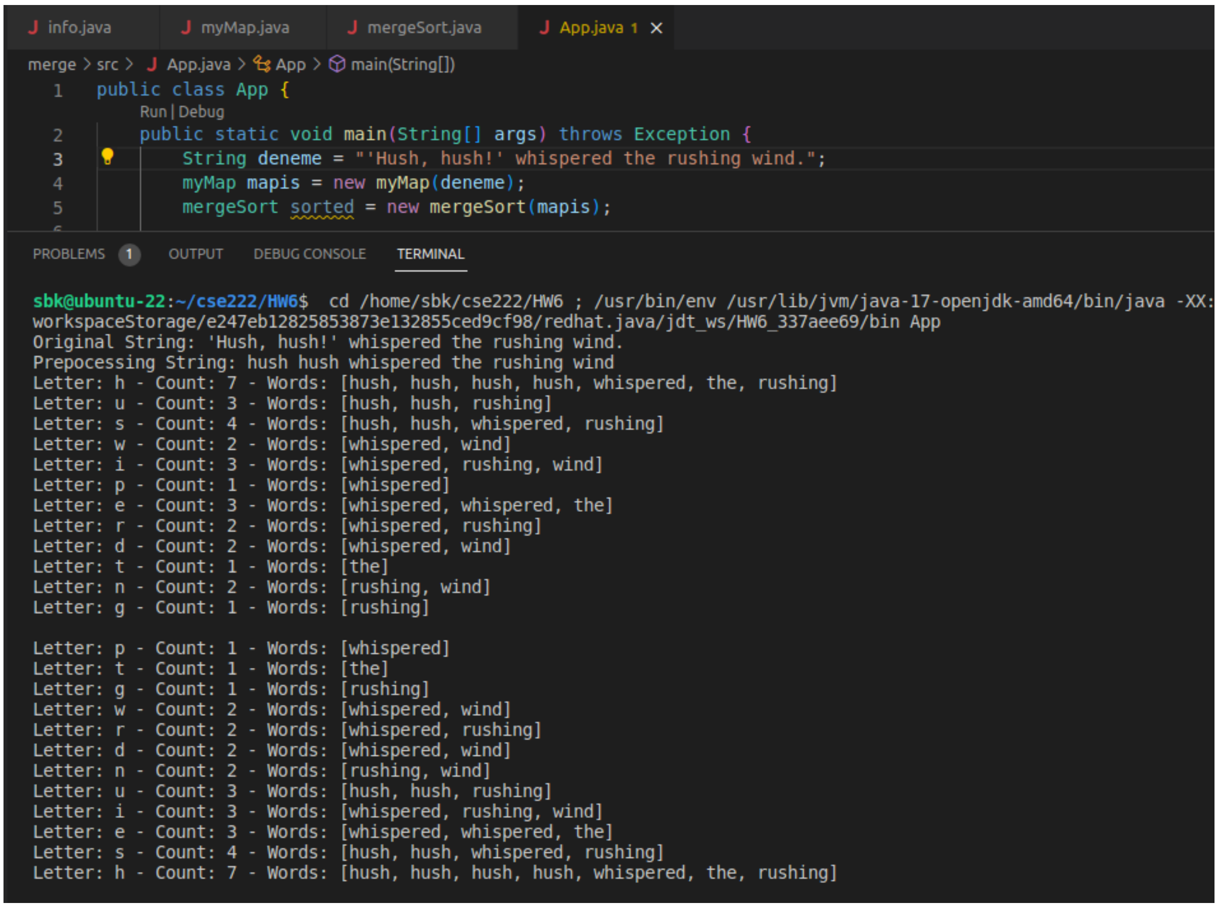
```
sbk@ubuntu-22:~/cse222/HW6$  cd /home/sbk/cse222/HW6 ; /usr/bin/env /usr/lib/jvm/java-17-openjdk-amd64/
workspaceStorage/e247eb12825853873e132855ced9cf98/redhat.java/jdt_ws/HW6_337aee69/bin App
The string just contains number. You will not run this program with strinh that contains just numbers.
```



```java
public class App {
    public static void main(String[] args) throws Exception {
        String deneme = "";
        myMap mapis = new myMap(deneme);
        mergeSort sorted = new mergeSort(mapis);

    }
}
```

```
sbk@ubuntu-22:~/cse222/HW6$  cd /home/sbk/cse222/HW6 ; /usr/bin/env /usr/lib/jvm/java-17-openjdk
workspaceStorage/e247eb12825853873e132855ced9cf98/redhat.java/jdt_ws/HW6_337aee69/bin App
The stirng is empty. You will get error
```



```java
public class App {
    public static void main(String[] args) throws Exception {
        String deneme = "'Hush, hush!' whispered the rushing wind.";
        myMap mapis = new myMap(deneme);
        mergeSort sorted = new mergeSort(mapis);
```

```
sbk@ubuntu-22:~/cse222/HW6$  cd /home/sbk/cse222/HW6 ; /usr/bin/env /usr/lib/jvm/java-17-openjdk-amd64/bin/java -XX:
workspaceStorage/e247eb12825853873e132855ced9cf98/redhat.java/jdt_ws/HW6_337aee69/bin App
Original String: 'Hush, hush!' whispered the rushing wind.
Prepocessing String: hush hush whispered the rushing wind
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]

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]
```