COMP6481: Programming and Problem Solving
Assignment # 3
PART 2

AUTHORS: Sraddha Bhattacharjee – 40221370 & Sai Shrawan Malyala - 40236492

1. **A detailed report about your design decisions and specification of your ElasticERL ADT including a rationale and comments about assumptions and semantics.**

- For datasets containing 100,000 or more entries, we utilized an AVL tree to store hospital IDs, while for datasets ranging in size from 100 to 100,000 entries, we used an array.
- Additionally, we stored student IDs and their corresponding data as key-value pairs in a HashMap.
- We opted to use an **AVL tree** for storing hospital IDs because it offers a time complexity of O (log n) for both node insertion and deletion operations. Furthermore, for retrieving the elements in sorted order, we performed an in-order traversal of the tree, which has a time complexity of O(n).
- We selected a **HashMap** for storing student IDs and their data because it provides a time complexity of O (1) for both inserting new key-value pairs and retrieving values based on their keys. This makes it an efficient data structure for our use case.
- We created a HashMap tailored to our needs by utilizing Array and LinkedList data structures and implemented dynamic resizing of the array. When the number of elements in the HashMap grows to twice the size of the array, we resize the array to reduce collisions and maintain optimal performance
- To generate the hash value for our HashMap, we utilized the built-in hash function provided by Java's Objects class
- As per the project's specifications, for datasets containing a large number of entries (exceeding 1000 or even tens of thousands of elements), we opted to use **Merge Sort.** Although Merge Sort may require more memory than other sorting algorithms, it is well-suited for large inputs due to its speed and guaranteed time complexity of O (n log n). This is achieved by splitting the list into sub-lists and merging them back together.

2. **Discuss how both the time and space complexity change for each of the above methods depending on the underlying structure of your ElasticERL (i.e. whether it is an array, linked list, etc.)?**

1. **SetEINThreshold (Size)**: - where 100 ≤ Size ≤ ~500,000 is an integer number that defines the size of the list
    - Time and space complexity are both O(1) as we are only assigning the size to a variable

2. **generate()** - randomly generates new non-existing key of 8 digits

    - The time and space complexity for our operation are both O(1). We achieve this by generating a random value and checking if it already exists in our

HashMap. Since lookups in a HashMap have a time complexity of O(1), we can efficiently check whether the generated value already exists. If the value does exist, we generate a new random number until we find one that does not already exist in our HashMap.

➢

3. **allKeys()** - return all keys in ElasticERL as a sorted sequence;

    ➢ For sizes greater than or equal to 100,000 we use AVL Tree. We perform in-order traversal to get all keys in sorted order, thus both the time and space complexity are O(n).
    ➢ For sizes less than 100,000 but greater than 100, we are using Merge Sort so we have guaranteed time complexity of O(n log n) and space complexity of O(n)

4. **add(ElasticERL,key,value):** add an entry for the given key and value;
    ➢ For sizes greater than or equal to 100,000 , we use AVL Tree so the time complexity for insertion is O(log n)
    ➢ For sizes less than 100,000, we use Arrays, so the time complexity for insertion is O(1)
    ➢ Space complexity in both cases would be O(1)

5. **remove(ElasticERL , key)** - remove the entry for the given key
    ➢ For sizes greater than or equal to 100,000 we use AVL Tree so the time complexity for deletion is O(log n)
    ➢ For sizes of less than 100,000, we are using an Array data structure. In this case, the time complexity for deletion is O(n), as we need to shift all the elements from the index of the element to be deleted to the end of the array to maintain continuity. This is because arrays have a fixed size and contiguous memory, so we need to rearrange the remaining elements to maintain the correct order.
    ➢ No space is used in both cases so the space complexity would be O (1)

6. **getValues(ElasticERL , key)** - return the values of the given key
    ➢ Since HashMaps provide constant-time (O(1)) access to values based on their keys, retrieving a value associated with a given student ID has an efficient time complexity. Additionally, the space complexity for storing each key-value pair in the HashMap is also O(1), making it a suitable choice for our use case.

7. **nextKey(ElasticERL ,key)** - return the key for the successor of key
    ➢ In our method, we retrieve all keys from the HashMap and sort them using the allKeys() function. Then, we iterate over the sorted keys to obtain the next key. While retrieving all keys takes O(n log n) time, the overall time complexity for iterating over the sorted keys is O(n), resulting in an overall time complexity of O(n log n). Additionally, the space complexity for this method is O(n), as we need to call the allKeys() function to retrieve all keys from the HashMap.

8. **prevKey(ElasticERL ,key)** - return the key for the predecessor of key

- ➢ In our method, we retrieve all keys from the HashMap and sort them using the allKeys() function. Then, we iterate over the sorted keys to obtain the previous key. While retrieving all keys takes O(n log n) time, the overall time complexity for iterating over the sorted keys is O(n), resulting in an overall time complexity of O(n log n). Additionally, the space complexity for this method is O(n), as we need to call the allKeys() function to retrieve all keys from the HashMap.

9. **rangeKey(key1, key2)** - returns the number of keys that are within the specified range of the two keys key1 and key2
    - ➢ In our method, we obtain all keys from the HashMap and sort them using the allKeys() function. Next, we iterate over the sorted keys to find the indices of key1 and key2. We then create a subarray from the index of key1 to the index of key2. While obtaining all keys and sorting them takes O(n log n) time, the overall time complexity for finding the indices and creating the subarray is O(n), resulting in an overall time complexity of O(n log n). Additionally, the space complexity for this method is O(n), as we need to call the allKeys() function to retrieve all keys from the HashMap.

3. Write the pseudo code for at least 4 of the above methods

1. allKeys()

```
BEGIN
if (threshold >= 100000) then
return avltree.inordertraversal()
else
return mergesort.sort(hosplist)
end if
END
```

2. add(ElasticERL,key,value):

```
BEGIN
if(!hashmap.contains(key)) then
hashmap.put(key,value))
if(threshold >= 100000) then
avltree.insertnode(key)
else
studentlist[size] = key
end if
size++
else
print("key already exists")
END
```

3. remove(ElasticERL , key)

```
BEGIN
hashmap.remove(key)
if (thresholdValue >= 100000) then
avlTree.deleteNode(key)
else
removeEleFromArray(key)
end if
END
```

4. **prevKey(ElasticERL ,key)**

```
 BEGIN
sortedKeysList = allKeys()
for i = 0 to length of sortedKeysList do
if sortedKeysList[i] == key then
if i-1 >= 0 then
return sortedKeysList[i-1]
end if
end if
end for
return -1
END
```