**Programming and Problem Solving**

**Assignment # 3**

**PART 2**

**SUBMITTED BY**

Manan Dineshbhai Paruthi - 40192620

&

Harman Preet Kaur - 40198317

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

* We have stored student id in ***AVL Tree*** for datasets with size in range of 100,000 to 500,000 & in ***Array*** for datasets with size in the range of 100 to 99,999 - along with that we have stored the student id and student data as a key value pair in ***Hashmap***
* ***AVL Tree*** is used as it has O(log n) time complexity for insertion of a node & deletion of a node and for getting elements in sorted order we just need to do inorder traversal which has time complexity of O(n)
* ***Array*** is used as it has it has O(1) time complexity for insertion and O(n) time complexity for deletion
* ***Hashmap*** is used as it has O(1) time complexity for insertion and lookup
* We have made a custom ***Hashmap*** using ***Array*** and ***LinkedList*** and we are doing ***dynamic resizing of array*** when the number of elements increases more than twice the size of array in order to decrease the collision and to compute the hash value we are using in-built hash function of Objects available in Java
* As it was mentioned in the assignment, for large number of entries (greater than 1000 or even in the range of tens of thousands of elements), it might have a higher memory requirement but faster(sorting) algorithms - we have used ***Merge Sort*** - as it is fast for huge inputs and it requires space as it splits list into sublist and merge them again - it has guaranteed complexity of O(n log n)
* And as it stated that for a small number of entries (e.g., few hundreds), it might use less memory overhead but slower (sorting) algorithms - we have used ***Quick Sort*** - as it is is faster for large inputs and does in-place sorting so do not require much extra space - it has randomised pivot selection so it’s complexity is O(n log n) expected and it is not guaranteed

1. **Discuss how both the time and space complexity change for each of the methods depending on the underlying structure of your CleverSIDC (i.e. whether it is an array, linked list, etc.)?**
2. ***setSIDCThreshold (size)*** - where 100 ≤ Size ≤ ~500,000 is an integer number that defines the size of the list

* Its time and space complexity is O(1) as we are only assigning the size to a variable

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

* Its time and space complexity is O(1) as we are generating a random value and checking if it already exists in Hashmap and lookup in Hashmap is O(1) - if it exists then we are generating a new number

1. ***allKeys()*** - return all keys in CleverSIDC as a sorted sequence

* For size greater than equal to 100,000 , we are using AVL Tree so in that we are doing inorder traversal to get all keys in sorted order thus the time and space complexity is O(n)
* For size greater than equal to 1000 , we are using Merge Sort so we have guaranteed time complexity of O(n log n) and space complexity of O(n)
* For size less than 1000 , we are using Quick Sort so we have expected time complexity of O(n log n) and space complexity of O(log n)

1. ***add(key,value)*** - add an entry for the given key and value

* For size greater than equal to 100,000 , we are using AVL Tree so time complexity for insertion is O(log n)
* For size less than 100,000 , we are using Array so time complexity for insertion is O(1)
* Space complexity in both case would be O(1)

1. ***remove(key)*** - remove the entry for the given key

* For size greater than equal to 100,000 , we are using AVL Tree so time complexity for deletion is O(log n)
* For size less than 100,000 , we are using Array so time complexity for deletion is O(n) as we require shifting of elements from the index of element to be deleted till the last index of array
* No space is used in both case so space complexity would be O(1)

1. ***getValues(key)*** - return the values of the given key

* We are storing student id and student data as key value pair in hashmap so getting value from hashmap has O(1) time complexity and space complexity will be O(1)

1. ***nextKey(key)*** - return the key for the successor of key
2. ***prevKey(key)*** - return the key for the predecessor of key

* For both the above methods, We are getting all keys in sorted format using allKeys() function and then iterating over it to get the next / previous key so it takes O(n) time but as the time required to get allKeys() is O(n log n) the overall time complexity would be O(n log n) and the space complexity would be O(n) as we are calling allKeys() function

1. ***rangeKey(key1, key2)*** - returns the number of keys that are within the specified range of the two keys key1 and key2

* We are getting all keys in sorted format using allKeys() function and then iterating over it to find the index of key1 & key2 and then to make a subarray from index of key1 to index of key2 so it takes O(n) time but as the time required to get allKeys() is O(n log n) the overall time complexity would be O(n log n) and the space complexity would be O(n) as we are calling allKeys() function

1. **Write the pseudo code for at least 4 of the above methods**
2. **METHOD ALLKEYS()**

BEGIN

IF (THRESHOLD >= 100000) THEN

RETURN AVLTREE.INORDERTRAVERSAL()

ELSE IF(THRESHOLD >= 1000) THEN

RETURN MERGESORT.SORT(STUDENTLIST)

ELSE

RETURN QUICKSORT.SORT(STUDENTLIST)

END IF

END METHOD

1. **METHOD ADD(key,value)**

BEGIN

IF(!HASHMAP.CONTAINS(key)) THEN

HASHMAP.PUT(key,value))

IF(THRESHOLD >= 100000) THEN

AVLTREE.INSERTNODE(Key)

ELSE IF (THRESHOLD >= 1000) THEN

STUDENTLIST[SIZE] = key

ELSE

STUDENTLIST[SIZE] = key

END IF

SIZE++

ELSE

PRINT("KEY ALREADY EXISTS")

END IF

END METHOD

1. **METHOD REMOVE(key)**

BEGIN

HASHMAP.REMOVE(key)

IF(THRESHOLD >= 100000) THEN

AVLTREE.DELETENODE(key)

ELSE

INDEXTOBEDELETED = -1;

FOR i = 0 TO SIZE

IF(STUDENTLIST[i] == key)

INDEXTOBEDELETED = i;

break;

END IF

END FOR

IF(INDEXTOBEDELETED == -1) THEN

PRINT(KEY + " does not exist")

ELSE IF(INDEXTOBEDELETED == SIZE-1) THEN

STUDENTLIST[SIZE-1] = 0;

SIZE--;

ELSE

FOR i = indexToBeDeleted TO size-1

STUDENTLIST[i] = STUDENTLIST[i+1]

END FOR

STUDENTLIST[SIZE-1] = 0;

SIZE--;

END IF

END IF

END METHOD

1. **METHOD NEXTKEY(key)**

BEGIN

SORTEDKEYSLIST = ALLKEYS()

FOR i= 0 to SORTEDKEYSLIST.LENGTH

IF SORTEDKEYSLIST[i] = key THEN

IF i+1 < SIZE THEN

RETURN SORTEDKEYSLIST[i+1]

END IF

END IF

END FOR

END METHOD