Assignment 3

Part 2

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

a) **Generate():**

```
Algorithm generate()
Output 8-digit unique key
Start
    Generate 8-digit random key
    While (key is already present in the data structure) then
    Do
        Generate 8-digit random key
    End while
    Generate random value
    Insert (key, value) pair in the data structure
End
```

b) **Add(key, value):**

```
Algorithm add(key, value)
Input 8-digit key and value
Output adds the (key, value) pair in the data structure
Start
    Search for the key
    If (key is not present) then
        Call insert method of the data structure and insert (key, value) pair
    Else
        Display "Key already present"
    End if
End
```

c) **remove(key):**

```
Algorithm remove(key)
Input 8-digit key
Output removes the key
Start
    If (data structure is empty) then
        Display "Empty data structure"
    End if
    Search for the key
    If (key is present) then
            Call remove method of the data structure and remove (key, value) pair
    Else
        Display "Key is not present"
    End if
End
```

d) **nextKey(key):**

```
Algorithm nextKey(key)
Input 8-digit key
Output next key of the given key
Start
    If (data structure is empty) then
        Display "Empty data structure"
    End if
    Search for the key
    If (key is not present) then
        Display "Key is not present"
    Else if (key is present at the last node) then
        Display "No next key is present"
    Else
        Return the next key value
    End if
End
```

2. Write the java code that implements all the above methods.

- The java code of all the above methods is submitted as java files with submission.

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

- The time and space complexity changes depending upon the underlying data structure.
- Here, I have implemented two data structures from scratch.
- Depending upon the threshold value the data structure is selected to store the data.
- If the threshold is less than 100, then **Binary Search Tree (BST)** is used, otherwise, **linked hash table** is used.

❖ setThreshold():
  - This method is independent of the underlying data structure.
  - Depending upon the threshold value, it selects the data structure to store the data and perform operations.
  - So time complexity is **O(1)**, space complexity **O(1)** as I haven't created any extra space.

❖ generate():
  - This method generates the 8-digit unique key.
  - It searches for the generated key in the data structure. If present, then it generates random key until it is not found.
  - For that, algorithm searches for the key every time when it generates the key.
  - Searching in BST takes O(log n) time, and for linked hash table amortized time complexity is O(1), worst case is O(n).
  - Space complexity for both is O(1).

- allKeys():
  - This method returns all the keys in sorted order.
  - Inorder traversal of BST returns all keys in sorted order. So, time complexity is O(n).
  - For linked hash table, first we traverse through all keys and store them in array list.
  - It takes O(n) time. After we sort it using radix sort. It takes O(n) time. So total time complexity is O(2n), which is O(n).

- add(key, value):
  - This method adds the (key, value) pair in data structure.
  - It first search for the key in the data structure.
  - For BST, it takes O(log n) time and for linked hash table it takes O(1) time.
  - For inserting, BST takes O(log n) time and linked hash table takes O(1) time.
  - So, time complexity for BST is O(log n) and amortized time complexity for linked hash table is O(1).
  - Space complexity is O(1) for both, as I haven't created any extra space.

- remove(key):
  - This method removes (key, value) pair.
  - It first search for the key in the data structure.
  - For BST, it takes O(log n) time and for linked hash table it takes O(1) time.
  - Then removing for BST takes O(1) time and for linked hash table it is O(1).
  - So, time complexity for BST is O(log n) and amortized time complexity for linked hash table is O(1).
  - Space complexity is O(1) for both, as I haven't created any extra space.

- getValues(key):
  - This method returns the value associated with the key.
  - It first search for the key in the data structure.
  - For BST, it takes O(log n) time and for linked hash table it takes O(1) time.
  - So, time complexity for BST is O(log n) and amortized time complexity for linked hash table is O(1).
  - Space complexity is O(1) for both, as I haven't created any extra space.

- nextKey(key):
  - This method returns the next key of the given key.
  - It first search for the key in the data structure and returns the next key of the given key if the key is present and it is not at the last position.
  - For BST, it takes O(log n) time and for linked hash table it takes O(1) time.
  - So, time complexity for BST is O(log n) and amortized time complexity for linked hash table is O(1).
  - Space complexity is O(1) for both, as I haven't created any extra space.

- prevKey(key):
  - This method returns the previous key of the given key.
  - It first search for the key in the data structure and returns the previous key of the given key if the key is present and it is not at the first position.
  - For BST, it takes O(log n) time and for linked hash table it takes O(1) time.

- So, time complexity for BST is O(log n) and amortized time complexity for linked hash table is O(1).
- Space complexity is O(1) for both, as I haven’t created any extra space.

- rangeKey(key1, key2):
  - This method returns the number of keys in the data structure which are within the specified range.
  - For this, it traverses through the entire data structure and checks every key.
  - For BST and linked hash table, it takes O(n) time.
  - Space complexity for both is O(1), as I haven’t created any extra space.

Design decision and Specification

- CleverSIDC is an Abstract Data Type, which stores (key, value) pairs.
- Keys of CleverSIDC entries are long integers of 8 digits, and one can retrieve the keys/values of an CleverSIDC or access a single element by its key.
- Here, each key is a unique entry.
- This CleverSIDC ADT uses two custom data structures, which I have implemented from scratch.
    - Binary Search Tree (BST)
    - Linked Hash Table
- So, every time when the object of CleverSIDC is created, it instantiates these two data structures.
- Which data structure to use to store the data is dependent upon the **threshold** value.
- As described in project description, if threshold value is less than 100, then BST is selected, otherwise Linked Hash Table is selected.
- For that, I have used two variables, named method_1 and method_2. These variables are set to 1 depending upon the threshold value. To select BST method_1 is used, whereas for Linked Hash Table method_2 variable is used.

Then **generate()** method generates 8-digit unique key. It uses Random built-in class to generate 8-digit key. Every time when it creates random key, it checks for its existence in the data structure depending upon the value of variables method_1 and method_2. If value is already present, then it generates random key again. This process is repeated until unique key is generated.

The **allKeys()** method returns all the keys in sorted order. BST stores values in following order (left < root < right), it means for any node, it’s left child value is less than the node and right child value is greater than the node. Here, I have created custom array list class from scratch. I have implemented all the necessary methods. So inorder traversal of BST returns tree in sorted order. And these keys are directly stored in custom array list. For Linked Hash table, keys are arranged depending upon the hash value of the key. So, keys are always unsorted. So, we must have to traverse through entire hash table and store those keys in the custom array list. Then we must sort this array list. Here I have used radix sort to sort array list.

The **add(key, value)** method stores the given (key, value) pair in the data structure depending upon the value of the variables method_1 and method_2. As it is mentioned in the requirement, all keys must be unique. So, to avoid duplication, firstly, we must have to check whether the given key is already present or not. If present, then display error message. Otherwise, we can simply insert (key, value) pair. For BST, we traverse through the tree until we reach an appropriate location for the key. Once we reach at the location, we insert the node storing given (key, value) pair.

The **remove(key)** method removes the given key from the data structure depending upon the value of the variables method_1 and method_2. Algorithm first search for the key, if key is already present then it simply removes the key, otherwise, it displays an error message. For Linked hash table, this method is quite straight forward. We simply calculate the hash value of the given key and check for that key at the location pointed by the corresponding hash value.

The **getValues(key)** method returns the value associated with the given key. firstly, we must have to check whether the given key is already present or not. If present, then simply return the value associated with the given key. Otherwise, display an error message.

The **nextKey(key)** method returns the next key of the given key. Here, two things are possible, given key is present, or given key is absent. If key is absent, then it simply displays an error message. If key is present, then again two things are possible, key is at the last position, or it is somewhere in the middle. If the key is at last position, then it displays an error message as there is no next key. Otherwise, it returns the next key of the given key. In BST, if the given key is the leaf node, then it returns null as there is no next key. If the given key is an internal node, then it returns its left or right child. In Linked Hash Table, as insertion order is maintained, it will return the next inserted key after the given key.

The **prevKey(key)** method returns the previous key of the given key. Here, two things are possible, given key is present, or given key is absent. If key is absent, then it simply displays an error message. If key is present, then again two things are possible, key is at the first position, or it is somewhere in the middle. If the key is at first position, then it displays an error message as there is no previous key. Otherwise, it returns the previous key of the given key. In BST, if the given key is the root node, then it returns null as there is no parent of the root. If the given key is an internal node, then it returns its parent. In Linked Hash Table, as insertion order is maintained, it will return the previously inserted key before the given key.

The **rangeKey(key1, key2)** method returns the number of keys that are within the specified range of the two keys *key1* and *key2*. Algorithm first stores all the keys in the array list. Then it traverses through the entire array list and checks whether the key is within the specified range. If so, then it increments the counter, otherwise, it moves to next entry in array list. Finally, it returns the count.