**CleverSIDC**

**Harshil Patel 40163431**

**Parth Navsariwala 40178800**

**Design Specifications:**

**Data Structures used**

When number of elements are below 1000, a Linked List ADT is used as an underlying Data Structure, otherwise AVL Tree ADT is used.

Element count < 1000 : CleverSIDC using **Linked List**

Element count > 1000 : CleverSIDC using **AVL Tree**

**Motive behind choosing the listed Data Structures**

**LinkedList** is used for smaller datasets since complexity of most operations resulting in O(n) wont really affect processing speed of the program.

**AVL tree** is used for larger datasets is its ability to perform add, delete, search operations in a time complexity of O(log n) along with taking linear time to return all keys in sorted manner, this is because inorder traversal of AVL tree results in sorted output.

Besides these, a **Hashset** is created with a fixed size of 10^8 to store keys. This Hashset is used to check whether randomly generated key is already present in the data set or not. This was we won’t have to traverse through entire data structure every time you generate a new random key to check whether it already exists or not. Size of Hashset is 10^8 because each key is of 8 digits. So, we directly insert key matching to its index number in Hashset. We can easily look into Hashset at the keys index value to check whether key already exists or not. This effectively gives expected time complexity of O(1) to generate random key value pair.

Based on the key count passed in setSIDCThreshold(size) method, appropriate data structure is created. Since both data structures are free of fixed size, it won’t be an issue to add as many keys as user wants to.

**Time and Space Complexities**

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Method** | **Linked List** | **AVL Tree** |
| **Generate** | **O(n) (Expected O(1))** | **O(n) (Expected O(1))** |
| **allKeys** | **O(n log n)** | **O(n)** |
| **add** | **O(1)** | **O(log n)** |
| **remove** | **O(n)** | **O(log n)** |
| **getValues** | **O(n)** | **O(log n)** |
| **nextKey** | **O(n)** | **O(log n)** |
| **prevKey** | **O(n)** | **O(log n)** |
| **rangeKey** | **O(n)** | **O(n)** |

**Space Complexity**

|  |  |  |
| --- | --- | --- |
| **Method** | **Linked List** | **AVL Tree** |
| **Generate** | **O(100000000)** | **O(100000000)** |
| **allKeys** | **O(n)** | **O(n)** |
| **add** | **O(1)** | **O(1)** |
| **remove** | **O(1)** | **O(1)** |
| **getValues** | **O(1)** | **O(1)** |
| **nextKey** | **O(1)** | **O(1)** |
| **prevKey** | **O(1)** | **O(1)** |
| **rangeKey** | **O(n)** | **O(n)** |

**Pseudo Codes**

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

**if** **isAlreadyPresent**(key) **then**

**Print** “Key already present, not added”

Node t <- new Node()

t.key <- key

t.value <- value

head <- t

size <- size + 1

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

Node t <- head

Node prev <- null

**if** **notNull**(t) **and** t.key <- key **then**

head <- head.next

size <- size – 1

**print** “ key found and removed”

**while** t != null **and** t.key != key **do**

prev <- t

t <- t.next

**if** **isNull**(t) **then**

**print** “key not present”

**return**

prev.next -> t.next

**print** “key found and removed”

**return**

1. **nextKey(key): return next key for given key**

Node t <- head

int next <- Integer.MAX\_VALUE;

**while** **notNull**(t.next) **do**

t <- t.next

**if** t.key >key **and** t.key < next **then**

next = t.key

**if** next == Integer.MAX\_VALUE **then**

**print** “ No next key present”

**else**

**print** “Next key found : “ + next

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

Node t <- head

**while** **notNull**(t) **do**

**if** t.key == key **then**

Return t,value

**else**

t <- t.next

**print** “Key is not present”

**return** “”