Assignment 3 (Part 2)

COMP 6841

Raj Patel

(40085012)

I. Introduction to the methods:

Following the requirements, the IntelligentSIDC system should be implemented with minimal space utilization and time constraints for the given methods. The maximum threshold constraining the total entries/records is nearly 500,000. Considering, given specifications following data structures can be used to achieve the given task.

Hash Map: Due to its least time complexity to search, O(1) (expected) a record, it is one of the most important data type for the given use case. It will allow to quickly retrieve the records. But considering the worst case where due to collisions, search may require O(n) and so we need to adapt a hash function such that it will make sure that the elements will be less clustered within the hash table. This requires increasing the size of hash map when load factor increase by certain amount. There will be more space utilization in the end depending upon the number of collisions occur. We will explore the time and space complexities for the given methods:

1) allKeys(IntelligentSIDC): This method will require traversing the hash table plus sorting the entries. If consecutive elements are stored at different indices, then it will require one more additional sort over the collected elements. Hence,
   Time complexity: O(nlogn)
   Space complexity: O(n).
2) add(IntelligentSIDC,key,value): This method in the worst case can result in O(n), where it requires to iterate through every element to find the place. To improve, if the buckets are implemented using Binary Search Tree, then it can be reduced to O(logn).
3) getValues(IntelligentSIDC,key): In the best case, it will be O(1), but in the worst case, it will be O(n).
4) nextKey(IntelligentSIDC,key) and prevKey(IntelligentSIDC,key) : It will be O(n) in the worst case. Also, the retrieved key will not be in the same order as inserted or sorted.
5) rangeKey(key1, key2): This will require O(n) time complexity in the worst case.

Binary Search Tree (BST): As described above, the time complexity of Search and Insertion is O(N). While, the BST will take O(logn) time complexity. Maintaining the tree through AVL tree will help to maintain these constraints. In this case, the complexities of given methods will be:

1) allKeys(IntelligentSIDC): Inorder traversal in BST will result in sorted sequence of records. The time required for traversal will be the total number of elements in the tree, i.e

Time complexity: O(n)
Space complexity: O(logn).

2) **add(IntelligentSIDC,key,value)**: Adding a record to the tree will be in O(logn) time, as it will be inserted by traversing either to the left or the right tree halving each time.
3) **getValues(IntelligentSIDC,key):** Retrieving a value is followed by searching in BST, that requires O(log n) time.
4) **nextKey(IntelligentSIDC,key) and prevKey(IntelligentSIDC,key) :** These methods will be implemented by traversing the tree in inorder.
   Time complexity will be O(n).
5) **rangeKey(key1, key2)**: Since, the order of the keys is not required, the range of the keys can be find efficiently by **traversing** the tree, will result in time complexity of O(|V|), where v is the vertices in the tree.

Note: Both, the insertion and removal in BST will be followed by balancing the tree, but that will require time complexity of O(log n).

II. Design:

Following the above discussion, the system is designed using BST. It is best suited for retrieving all the records in the ascending order and looking for a value when the number of elements are more.

Pseudo Code for the required methods:

1) **allKeys(IntelligentSIDC)**: This method internally calls inorder traversal method, to perform the task:
   Inorder(root):
   1) Initialize an empty list L
   2) Let R be the root of the tree:
   3) If R is not null, then
      a. Inorder (R)
      b. Add the element to the list
      c. Inorder (R)

   Time complexity: O(n)

2) **add(IntelligentSIDC,key,value)**:

   Insert(key, value, root):
   1) Let R be the root of the element.
   2) If root is null, then we have reached at the leaf,
      a. So create a new node and return that node
   3) Else if the given value is less than the value at the current node:
      a. R.left = insert(key, value, R.left) // Call the left subtree
   4) Else if the given value is more than the value at the current node:

a. R.right = insert(key, value, R.right) // Call the left subtree

Time complexity: O(log n)

3) getValues(IntelligentSIDC,key):

Search(key, root):

1) Let R be the root of the element.
2) If root is null, then we have reached at the leaf,
   Return null // no element found
3) If key == root.key:
   a. Return the root.value
4) Else if the given value is less than the value at the current node:
   a. R.left = search(key, R.left) // Call the left subtree
5) Else if the given value is more than the value at the current node:
   a. R.right = search(key, R.right) // Call the left subtree

Time complexity: O(log n)

4) nextKey(IntelligentSIDC,key): It is implemented by traversing the tree in inorder.
Inorder(root):

1) Initialize an empty list L
2) Let R be the root of the tree:
3) If R is not null, then
   a. Inorder (R)
   b. Add the element to the list (L)
   c. Inorder (R)

While (L is not empty):

1) E = L.getFirst()
2) If E == Key and L.size() > 1:
   a. Return L.getFirst()

Time complexity: O(n)

5) prevKey(IntelligentSIDC,key): It is implemented by traversing the tree in inorder.
Inorder(root):

1) Initialize an empty list L
2) Let R be the root of the tree:
3) If R is not null, then
   d. Inorder (R)
   e. Add the element to the list (L)
   f. Inorder (R)

For i : L.size() to 1:

1) E = L.removeLast() // will not remove the element
2) If E == key
    a. Return L.removeLast()

Time complexity: O(n)

6) **rangeKey(key1, key2)**:
traverse(root, key1, key2):
    1) If root == null:
        Return null
    2) Else if root.key > key1
        a. Traverse (root.left, key1, key2) // traverse to the left sub tree
    3) If root.key >= key1 and root.key <= key2:
        a. L.add(root.key)
    4) If(root.key < key2)
        a. Traverse (root.right, key1, key2) // traverse to the left sub tree

Time complexity: O(|V|), will be O(n)

Space complexity of the algorithm is **O(log n)**

III. Empirical Analysis:

Time is in Milliseconds (ms).

Number of elments	Add (ms)	All keys (ms)	Remove (ms)	Get a value (ms)	Next Key (ms)	RangeKeys (10000000-99999999) (ms)
35,000	1013	275	0	0	16	255
105,000	2630	415	0	0	15	423
250,000	6722	2396	0	0	16	2576
400,000	9300	6680	0	0	47	6349
500,000	12346	7552	0	0	47	9952

IV. Conclusion:

From the above analysis, we can conclude that insertion is the most expensive operation in the BST followed by the RangeKey operation.