**Assignment 3 - Programming**

Q1) Psuedocode for methods:

* allkeys():

Method allkeys(h: HashyTably) returns int[]

Declare an integer array 'array' and initialize it with the keys from hash table 'h'

If the length of 'array' is less than 1000

Call insertionSort on 'array'

Initialize a counter variable to 0

For each element in 'array'

If counter equals 4

Print a newline

Reset counter to 0

Print the element prefixed with " Insrtion-> "

Increment counter

Else

Call radixSort on 'array'

For each element in 'array'

Print the element prefixed with " RadSrtd-> "

Return 'array'

End Method

* rangeKey():

Method rangeKey(h: HashyTably, key1: int, key2: int) returns int

Calculate the index of key1 in the hash table and store it in 'start'

Calculate the index of key2 in the hash table and store it in 'end'

Initialize a counter 'counter' to 0

Check if 'start' or 'end' are outside the bounds of the hash table

If true, print "there is no range between non-existent keys" and return 0

Loop from 'start' to 'end' (exclusive)

If the node at index 'i' is not null and its studentId is positive

Call nextKey with h and 'i'

Increment 'counter'

Print "range between 2 keys: [key1] and [key2] is [counter]"

Return 'counter'

End Method

* Remove():

Method remove(h: HashyTably, key: int)

Check if the slot returned by h.goTo() using h.firstHashFunction(key) is null

If true, print "key does not exits"

Otherwise, check if key is equal to the studentId at the slot returned by h.goTo() using h.firstHashFunction(key)

If true, set the studentId at this slot to -key (indicating removal)

Otherwise, check if the studentId at this slot is less than 0

If true, print "this key is already deleted or does not exits"

Else

Initialize 'index' to the value returned by h.firstHashFunction(key)

Initialize 'counter' to the value returned by h.secondHashFunction(key)

Initialize 'laps' to 0

Enter a loop that continues while the studentId at the slot 'index' is not equal to key

Check if 'laps' is equal to the length of h.studentArray minus 1

If true, print "key does not exist" and exit the method

Increment 'index' by 'counter'

If 'index' is greater or equal to the length of h.studentArray

Adjust 'index' to wrap around the array (index - h.studentArray.length - 1)

If the slot at 'index' is not null and the studentId at this slot is positive and equal to key

Set the studentId at this slot to -key

Increment 'laps'

End Method

* nextKey():

Method nextKey(h: HashyTably, key: int) returns Node

Calculate and store the index of the given key using h.getIndex(key) in 'temp'

Check if 'temp + 1' is within the bounds of the hash table (greater than or equal to 0 and less than h.studentArray.length - 2)

If true:

Retrieve the node at 'temp + 1' and store it in 'tempp'

Check if 'tempp' is null or if 'tempp.studentId' is less than 0 (indicating an empty slot or a deleted key)

If true, return null

Print a new line and the message "next key is: " followed by 'tempp.toString()'

Return 'tempp'

Else

Print "next key does not exist the index is empty"

Return null

End Method

Q2)

1. SetSIDCThreshold (Size)

The complexities depend on the operations involved in switching data structures (e.g., from a linked list to a hash table or a tree).

- Time Complexity: O(n) if it involves rehashing or rebuilding the entire data structure, where n is the number of elements.

- Space Complexity: O(n), where n is the new size threshold.

2. generate()

This method generates a new 8-digit key that does not already exist in the `CleverSIDC`.

- Time Complexity: O(n) in the worst case if it has to check for uniqueness against all existing keys.

- Space Complexity: O(1), as it generates a single key.

3. allKeys(CleverSIDC)

Returns all keys in `CleverSIDC` as a sorted sequence. Based on `HashyTably` and `LinkyListy`:

- Time Complexity: O(n log n) for sorting the keys after retrieval, where n is the number of keys.

- Space Complexity: O(n) for storing the keys.

4. add(CleverSIDC, key, value)

Adds an entry for the given key and value.

- Time Complexity:

- Hash Table (like `HashyTably`): O(1) on average, O(n) in the worst case due to collisions.

- Linked List (like `LinkyListy`): O(n) since it may need to traverse the list to find the insertion point.

- Space Complexity: O(1) for adding a single key-value pair.

5. remove(CleverSIDC, key)

Removes the entry for the given key.

- Time Complexity:

- Hash Table: O(1) on average, O(n) in the worst case due to collisions.

- Linked List: O(n) since it may need to traverse the list to find the key.

- Space Complexity: O(1).

6. getValues(CleverSIDC, key)

Returns the values of the given key.

- Time Complexity:

- Hash Table: O(1) on average, O(n) in the worst case.

- Linked List: O(n) to find the key.

- Space Complexity: O(1).

7. nextKey(CleverSIDC, key)

Returns the key for the successor of the given key.

- Time Complexity:

- Hash Table: O(1) on average, O(n) in the worst case.

- Linked List: O(n) if it needs to find the key and then its successor.

- Space Complexity: O(1).

8. prevKey(CleverSIDC, key)

Returns the key for the predecessor of the given key.

- Time Complexity:

- Hash Table: O(1) on average, O(n) in the worst case.

- Linked List: O(n) if it needs to find the key and then its predecessor.

- Space Complexity: O(1).

9. rangeKey(key1, key2)

Returns the number of keys within the specified range.

- Time Complexity:

- Hash Table: O(n) since it might need to check each key.

- Linked List: O(n), especially if the keys are not sorted.

- Space Complexity: O(1) as it counts keys without additional storage.

Report

Design Decisions and Rationale:

1. Data Structures:

- `HashyTably` uses an array-based hash table.

- `LinkyListy` implements a doubly linked list.

- This choice suggests a focus on optimizing search, insertion, and deletion operations. The hash table offers average-case O(1) time complexity for these operations, while the linked list provides efficient insertion and deletion.

2. Hash Functions (HashyTably):

- Two hash functions are used, indicating a strategy to manage collisions and distribute keys evenly.

- This improves the performance of the hash table by reducing the likelihood of clustering.

3. Dynamic Structure Adjustment (SetSIDCThreshold):

- The system adjusts its data structure (between a hash table, tree, AVL tree, etc.) based on the size threshold.

- This decision supports scalability and efficiency. Different structures are more efficient at different sizes and operation types.

4. Node Structure:

- Nodes store student IDs and additional information of the student.

- Nodes in `LinkyListy` include pointers to both next and previous nodes, allowing bidirectional traversal.

5. Collision Handling (HashyTably):

- Collision handling via linear probing is evident in `HashyTably`.

- This approach is simple and efficient for moderate load factors but can lead to clustering.

Assumptions:

1. Key Uniqueness:

- It is assumed that all keys (student IDs) are unique, as evident in the generate method.

2. Load Factor and Size Management:

- The system likely assumes a manageable load factor for the hash table to maintain efficiency.

3. Balancing Efficiency and Complexity:

- A balance is assumed between the need for efficient operations (like search and insert) and the complexity of the data structure.

Semantics:

1. `allKeys` Method:

- Sorts and returns all keys, reflecting a need for ordered data retrieval.

2. `add` and `remove` Methods:

- These methods suggest a dynamic dataset where entries can be frequently added or removed.

3. `nextKey` and `prevKey` Methods:

- Providing successors and predecessors indicates a use case for ordered traversal or sequential data processing.

4. `rangeKey` Method:

- Supports querying for a range of keys, useful in scenarios requiring data analysis within specific bounds.

5. Error Handling and Edge Cases:

- The code includes considerations for null values and empty data structures, which is crucial for robustness.

Conclusion:

The CleverSIDC ADT is designed with flexibility and efficiency in mind, adapting its underlying data structure based on size and operation requirements. The use of hash tables and linked lists caters to a wide range of operations with varying efficiency needs. The system seems well-suited for scenarios where rapid access to student data is essential, with particular emphasis on handling large datasets dynamically. The choice of data structures and methods indicates a focus on practical application in educational or administrative settings where student data is frequently accessed and modified.