**‘CONTACT MANAGEMENT SYSTEM’ USING HYBRID DATA STRUCTURE**



**B-Tech/II Year CSE/IV Semester**

**19CSE212/Data Structures and Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Name** | **Roll Number** | **Marks** |
| 1 | Murugesh JLP | CB.EN.U4CSE21137 |  |
| 2 | Nirmal Kanniaiyan Vkl | CB.EN.U4CSE21140 |  |
| 3 | Yashwanth S | CB.EN.U4CSE21151 |  |
|  |  |  |  |

**Amrita School of Engineering, Coimbatore**

**Department of Computer Science and Engineering**

**2022 -2023 Even Semester**

**PROBLEM STATEMENT​:**

* Implementing a ‘Contact management system’ using hybrid data structures.
* A ‘Hybrid data structure’ is a combination of one or more data structures, therefore yielding the advantages of all the data structures and resolving the disadvantages.
* In this Case Study, we have analysed the existing set of data structures in ‘Contact management system’ (existing: Arrays and Linked lists) and resolved them using hybrid data structures( Hashmap and AVL tree).
* Contact management systems play a significant role in computing and have several important benefits.

**APPLICATION OF CONTACT MANAGEMENT SYSTEM​:**

* Contact management and call logs have numerous applications in computing across various domains and have huge amount of relevancy. Here are some common applications:
* Telecommunication Services: Telecommunication service providers utilize call logs to maintain accurate billing records, analyse call patterns, monitor network performance, and detect fraud. Call logs are also crucial for compliance with regulations and law enforcement purposes.
* Personal Contact Management: Individuals can use contact management systems to organize and store contact information for personal use. It helps them maintain an organized address book, manage relationships with friends, family, and acquaintances, and quickly access contact details when needed.
* Thus, ensuring efficiency in search time and insertion time is inevitable.

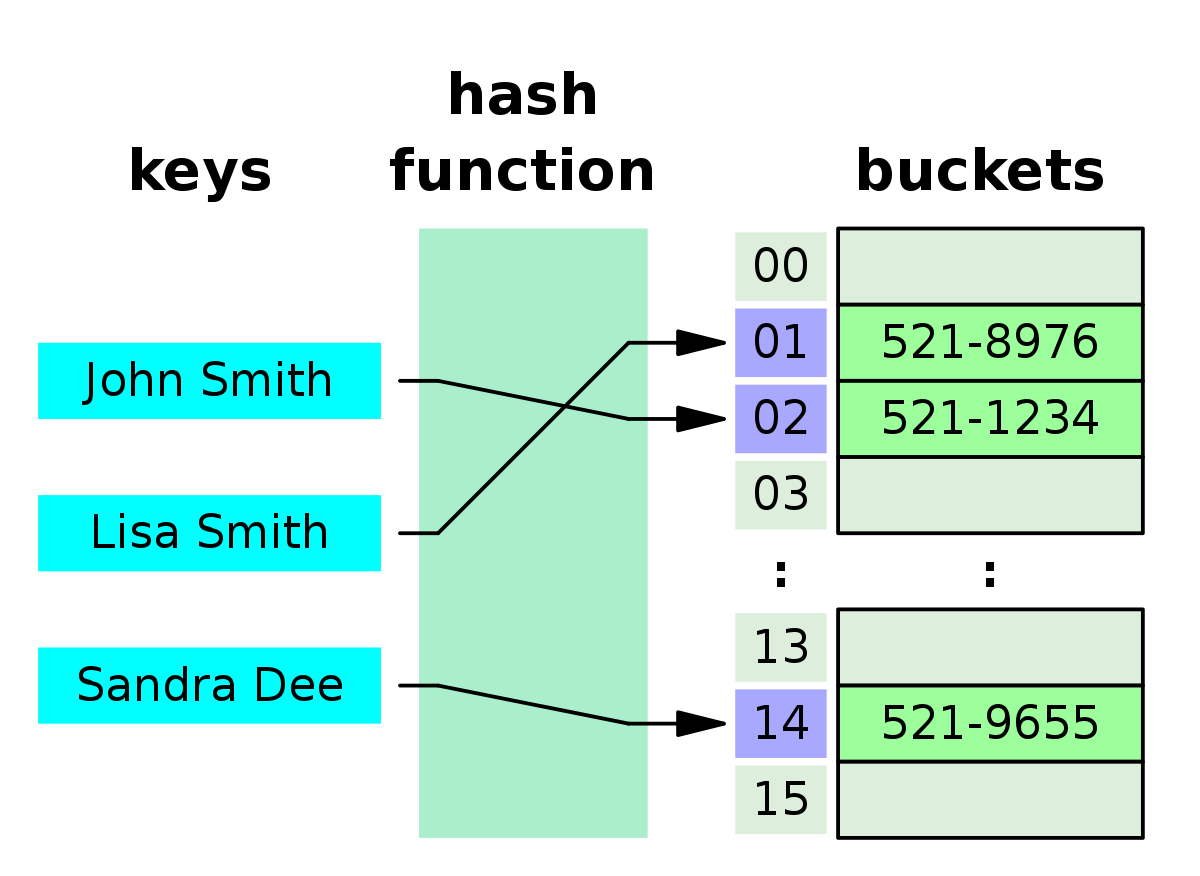
**EXISTING DATASTRUCTURES USED IN THE APPLICATION​ AND LIST OF PROBLEMS IDENTIFIED IN USING EXISTING DATA STRUCTURES FOR THE APPLICATION:**

* In the contact management system, both arrays and linked lists can be used to store and manage contact information.
* Storing Contacts: Each element of the array holds the details of a contact, such as name, phone number, email address, and other relevant information. For example, you can create an array of contact structures or objects where each element represents a contact.
* Accessing Contacts: Contacts in an array can be accessed directly using their indices. For instance, you can retrieve a specific contact by accessing the corresponding array element based on its index. This direct access makes it easy to search, retrieve, and update contacts in the array.
* Traversing and Manipulating Contacts: To access or manipulate contacts in a linked list, you typically start from the head of the list and traverse through the nodes until the desired contact is found. This traversal allows for operations like searching, insertion, deletion, and updating of contacts in the linked list.
* Insertion and Deletion Efficiency: Linked lists excel at efficient insertion and deletion operations, as they require updating only the affected nodes. In contrast, inserting or deleting elements in an array may require shifting elements or resizing the array.
* While linear data structures like arrays and linked lists can be useful for storing and managing contact information, they do have some disadvantages compared to our hybrid data structures like hashmaps and AVL trees:
* Search Efficiency: Linear data structures typically have linear search time complexity, meaning the time required to search for a specific contact increases linearly with the number of contacts. In large contact lists, this can result in slower search operations compared to more efficient data structures like hash tables or balanced search trees.

Time complexity: O(n)

* Insertion and Deletion Efficiency: Inserting or deleting contacts in linear data structures can be less efficient compared to specialized data structures like AVL trees or hash tables. In linked lists, insertion and deletion operations require traversing the list, which can be time-consuming for large lists. Similarly, arrays may require shifting elements or resizing, resulting in additional overhead.
* Lack of Built-in Key-Value Mapping: Linear data structures like arrays and linked lists do not provide inherent key-value mapping. While you can use indices or positions to represent keys, it may not be as convenient or efficient as directly mapping contacts to specific keys or identifiers using hash tables or trees.

**DIRECTION/IDEA FOR PROPOSING DATA STRUCTURE​:**

****

**Code:**

class TreeNode:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

self.height = 1

class AVLTree:

def \_\_init\_\_(self):

self.root = None

def \_get\_height(self, node):

if node is None:

return 0

return node.height

def \_update\_height(self, node):

if node is None:

return

node.height = 1 + max(self.\_get\_height(node.left), self.\_get\_height(node.right))

def \_get\_balance(self, node):

if node is None:

return 0

return self.\_get\_height(node.left) - self.\_get\_height(node.right)

def \_rotate\_left(self, z): # LL rotation

y = z.right

T2 = y.left

y.left = z

z.right = T2

self.\_update\_height(z)

self.\_update\_height(y)

return y

def \_rotate\_right(self, y): # RR rotation

x = y.left

T2 = x.right

x.right = y

y.left = T2

self.\_update\_height(y)

self.\_update\_height(x)

return x

def \_insert(self, node, value):

if node is None:

return TreeNode(value)

elif value < node.value:

node.left = self.\_insert(node.left, value)

else:

node.right = self.\_insert(node.right, value)

self.\_update\_height(node)

balance = self.\_get\_balance(node) # After inserting, we're checking continously the balane factor of all nodes.

# Abnormal balance factors

# If it is greater than 1

# Left subtree is longer than the right sub tree

if balance > 1:

if value < node.left.value:

return self.\_rotate\_right(node)

else:

node.left = self.\_rotate\_left(node.left)

return self.\_rotate\_right(node)

# If it is lesser than -1

# If the right subtree is larger than the left subtree

if balance < -1:

if value > node.right.value:

return self.\_rotate\_left(node)

else:

node.right = self.\_rotate\_right(node.right)

return self.\_rotate\_left(node)

return node

def insert(self, value):

self.root = self.\_insert(self.root, value)

class PhoneHistoryManager:

def \_\_init\_\_(self):

self.hash\_table = {}

def \_get\_hash\_key(self, phone\_number, name):

return hash(phone\_number + name) % 1000

def \_get\_height(self, node):

if node is None:

return 0

return node.height

def \_update\_height(self, key):

if key in self.hash\_table:

avl\_tree = self.hash\_table[key]

avl\_tree.\_update\_height(avl\_tree.root)

# If a user is added we'll atomically put it as the key of the hash table and we'll assign a AVL tree value for it

def add\_user(self, phone\_number, name):

key = self.\_get\_hash\_key(phone\_number, name)

if key not in self.hash\_table:

self.hash\_table[key] = AVLTree()

def remove\_user(self, phone\_number, name):

key = self.\_get\_hash\_key(phone\_number, name)

if key in self.hash\_table:

del self.hash\_table[key]

def add\_history(self, phone\_number, name, call\_time):

key = self.\_get\_hash\_key(phone\_number, name)

if key in self.hash\_table:

avl\_tree = self.hash\_table[key]

avl\_tree.insert(call\_time)

self.\_update\_height(key)

else:

self.add\_user(phone\_number, name)

self.add\_history(phone\_number, name, call\_time)

def remove\_history(self, phone\_number, name, call\_time):

key = self.\_get\_hash\_key(phone\_number, name)

if key in self.hash\_table:

avl\_tree = self.hash\_table[key]

avl\_tree.root = self.\_remove\_call\_history(avl\_tree.root, call\_time)

def \_remove\_call\_history(self, node, call\_time):

if node is None:

return None

if node.value == call\_time:

if node.left is None and node.right is None:

return None

elif node.left is None:

return node.right

elif node.right is None:

return node.left

else:

successor = self.\_get\_successor(node.right)

node.value = successor.value

node.right = self.\_remove\_call\_history(node.right, successor.value)

elif node.value < call\_time:

node.right = self.\_remove\_call\_history(node.right, call\_time)

else:

node.left = self.\_remove\_call\_history(node.left, call\_time)

self.\_update\_height(node)

balance = self.\_get\_balance(node)

if balance > 1:

if self.\_get\_balance(node.left) >= 0:

return self.\_rotate\_right(node)

else:

node.left = self.\_rotate\_left(node.left)

return self.\_rotate\_right(node)

if balance < -1:

if self.\_get\_balance(node.right) <= 0:

return self.\_rotate\_left(node)

else:

node.right = self.\_rotate\_right(node.right)

return self.\_rotate\_left(node)

return node

def \_get\_successor(self, node):

current = node

while current.left is not None:

current = current.left

return current

def get\_history\_before(self, phone\_number, name, call\_time):

key = self.\_get\_hash\_key(phone\_number, name)

if key in self.hash\_table:

avl\_tree = self.hash\_table[key]

return self.\_get\_call\_history\_before(avl\_tree.root, call\_time)

return None

def \_get\_call\_history\_before(self, node, call\_time):

if node is None:

return []

if node.value < call\_time:

left\_history = self.\_get\_call\_history\_before(node.left, call\_time)

right\_history = self.\_get\_call\_history\_before(node.right, call\_time)

return left\_history + [node.value] + right\_history

elif node.value == call\_time:

return [node.value]

else:

return self.\_get\_call\_history\_before(node.left, call\_time)

def \_get\_balance(self, node):

if node is None:

return 0

return self.\_get\_height(node.left) - self.\_get\_height(node.right)

def print\_contacts(self):

for key in self.hash\_table:

phone\_number, name = divmod(key, 1000)

print(f"Phone Number: {phone\_number}, Name: {name}")

# Create a PhoneHistoryManager instance

phone\_manager = PhoneHistoryManager()

# Add users and call history

phone\_manager.add\_user("555-1234", "John")

phone\_manager.add\_user("555-5678", "Jane")

phone\_manager.add\_history("555-1234", "John", "2023-06-10 14:45")

phone\_manager.add\_history("555-1234", "John", "2022-06-11 14:45")

phone\_manager.add\_history("555-1234", "John", "2023-06-11 14:45")

phone\_manager.add\_history("555-5678", "Jane", "2023-06-12 09:15")

# Retrieve call history before a specific time

call\_time = "2023-06-12 11:45"

history = phone\_manager.get\_history\_before("555-1234", "John", call\_time)

if history:

print(f"Call history for John (555-1234) before {call\_time}:")

for item in history:

print(item)

else:

print(f"No call history found for John (555-1234) before {call\_time}")

# Remove a call history entry

call\_time\_to\_remove = "2022-06-11 14:45"

phone\_manager.remove\_history("555-1234", "John", call\_time\_to\_remove)

# Retrieve call history after removing the entry

history = phone\_manager.get\_history\_before("555-1234", "John", call\_time)

if history:

print(f"Updated call history for John (555-1234) before {call\_time}:")

for item in history:

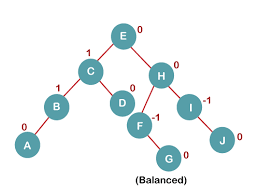
print(item)

else:

print(f"No call history found for John (555-1234) before {call\_time}")

# Remove a user

phone\_manager.remove\_user("555-5678", "Jane")

****

* In the given code, we have used hash tables and AVL trees to address some of the limitations of arrays and linked lists in a contact management system. Here's how we have solved those problems:
* Efficient Search: By using a hash table, we have implemented a mapping between a unique key (generated from phone number and name) and the corresponding AVL tree that stores the call history. This allows for efficient search operations by directly accessing the AVL tree associated with a specific contact, without having to traverse through the entire list of contacts.

Time complexity:O( log n )

* Dynamic Size and Flexibility: The use of hash tables and AVL trees provides flexibility for dynamically changing the size of the contact management system. New contacts can be added by creating a new AVL tree and linking it to the hash table, while removal of contacts involves deleting the corresponding AVL tree from the hash table.
* Efficient Insertion and Deletion: AVL trees are self-balancing binary search trees that provide efficient insertion and deletion operations while maintaining a balanced structure. This helps in minimizing the time complexity of these operations compared to linked lists or arrays, which may require shifting or resizing.
* Sorting and Indexing: Although not explicitly shown in the provided code, AVL trees have built-in support for maintaining a sorted order of elements based on the key values. This can be useful for efficiently retrieving call history in a sorted manner if needed.
* Key-Value Mapping: By using the hash table, we have established a key-value mapping between the unique key (generated from phone number and name) and the corresponding AVL tree that stores the call history. This allows for direct access to the AVL tree associated with a specific contact using the key-value pair, providing efficient retrieval and manipulation of call history.
* By leveraging hash tables and AVL trees,we have addressed the limitations of linear data structures like arrays and linked lists in terms of search efficiency, dynamic size, insertion and deletion operations, sorting, and key-value mapping. This approach enhances the performance and flexibility of the contact management system, making it more suitable for scenarios with a large number of contacts and efficient retrieval and manipulation requirements.