**NAME SYED ALI BUKHARI**

**CLASS BSSE**

**ASSINGMENT 3 DSA**

**12398**

**QUESTION 1**

1. **Hash Functions:**
   * A hash function takes input data (in this case, passwords) and produces a fixed-size string of characters, which is typically a hash value.
   * The key properties of a good hash function include determinism (the same input will always produce the same hash), efficiency (quick to compute), and unpredictability (a small change in input should produce a significantly different hash).
   * Hash functions are a crucial component of password hashing in authentication systems because they transform passwords into a format that is not easily reversible.
2. **Password Hashing:**
   * Instead of storing plain text passwords in a database, the system stores the hash values of the passwords.
   * When a user creates an account or updates their password, the system hashes the provided password using a secure hash function and then stores the hashed value in the database.
   * During authentication, the system hashes the entered password and compares it to the stored hash. If they match, access is granted.
3. **Hash Table for User Credentials:**
   * In the context of user authentication, a hash table can be used to efficiently store and retrieve user credentials. The hash table might use the usernames as keys and the hashed passwords as values.
   * The hash function is used to map each username to a specific location (bucket) in the hash table, making it faster to retrieve user information.
4. **Importance of Collision Resistance:**
   * Collision resistance is a crucial property of hash functions in password hashing.
   * A collision occurs when two different inputs produce the same hash output. In the context of user authentication, a collision would mean that two different passwords result in the same hash value.
   * Collision resistance is essential because if two different passwords hash to the same value, an attacker could potentially log in with a different password than the one originally set. This undermines the security of the authentication system.
5. **Salting:**
   * To enhance security further, a system should use a unique random value called a "salt" for each user. The salt is combined with the user's password before hashing.
   * Salting prevents attackers from using precomputed tables (rainbow tables) for common passwords since the same password will have different hash values for different users.

Top of Form

**QUESTION 2**

1. **Task Identification:**
   * Each scheduled task in the application is assigned a unique identifier or name. This could be a task ID, a task name, or any other unique attribute that distinguishes one task from another.
2. **Hash Function:**
   * A hash function is used to map the unique identifiers or names of tasks to specific locations (buckets) in the hash table. The goal is to distribute tasks across the hash table evenly.
3. **Storing Tasks in the Hash Table:**
   * The hash table is used to store the scheduled tasks. The unique identifier or name of each task is hashed using the chosen hash function, and the resulting hash value is used as the index to store the task in the hash table.
4. **Advantages of Using a Hash Table for Task Management:**

a. **Quick Retrieval:**

* + Hash tables offer constant-time average case complexity for insertion, deletion, and retrieval operations. This means that regardless of the number of tasks, the time it takes to retrieve a task based on its identifier is consistent and fast.

b. **Efficient Search Operations:**

* + Hash tables are designed for efficient search operations. When a task needs to be retrieved or updated, the application can use the task's identifier to compute the hash value and directly access the corresponding location in the hash table.

c. **Space Efficiency:**

* + Hash tables provide efficient space utilization. The size of the hash table can be dynamically adjusted based on the number of tasks, ensuring that memory is used optimally.

d. **Scalability:**

* + As the number of tasks in the scheduling application increases, a well-designed hash table can scale effectively. The constant-time complexity for key operations makes hash tables suitable for managing a large number of tasks without a significant increase in time complexity.

e. **Collision Handling:**

* + While collisions (two tasks mapping to the same hash value) can occur, they can be mitigated through techniques such as open addressing or chaining. Proper collision handling ensures that tasks are not lost or overwritten in the hash table.

f. **Simplified Implementation:**

* + Hash tables provide a straightforward and effective way to implement task management. The use of a hash function simplifies the association between task identifiers and their corresponding locations in memory.

In summary, a hash table offers a fast and efficient solution for storing and retrieving scheduled tasks in a task scheduling application. It provides constant-time complexity for key operations, making it well-suited for managing tasks with unique identifiers or names. The advantages include quick retrieval, efficient search operations, space efficiency, scalability, and simplified implementation.

Top of Form

**QUESTION 3**

Binary Search is a search algorithm that efficiently finds the position of a target element in a sorted list by repeatedly dividing the search range in half. Here's how Binary Search can be applied to identify the position of a particular student exam score in a sorted list:

1. **Initial Setup:**
   * The first step is to consider the entire sorted list as the initial search range.
2. **Comparison:**
   * Calculate the middle element of the current search range.
   * Compare the middle element with the target exam score.
3. **Updating Search Range:**
   * If the middle element is equal to the target score, the search is complete, and you have found the position of the score in the list.
   * If the middle element is greater than the target score, update the search range to the lower half of the current range (excluding the middle element).
   * If the middle element is less than the target score, update the search range to the upper half of the current range (excluding the middle element).
4. **Repeat:**
   * Repeat steps 2 and 3 until the target score is found, or the search range becomes empty (indicating that the score is not in the list).
5. **Identifying Position:**
   * If the target score is found, the position is the index of the middle element in the sorted list. If the search range is empty, the target score is not in the list.

**Assumptions and Requirements for Binary Search:**

1. **Sorted List:**
   * The list of student exam scores must be sorted in ascending (or descending) order for Binary Search to be applicable. If the list is not sorted, the algorithm will not work correctly.
2. **Random Access:**
   * Binary Search assumes random access to the elements of the list. This means that you can directly access any element in the list in constant time. Arrays or data structures that provide random access, such as arrays or certain types of lists, are suitable for Binary Search.
3. **Comparable Elements:**
   * The elements in the list must be comparable. This means there should be a way to determine whether one element is greater than, equal to, or less than another element. For numeric scores, this comparison is straightforward.
4. **Efficiency:**
   * Binary Search is more efficient than linear search, especially for large sorted datasets. However, its efficiency comes into play when the list is relatively large. For small lists, the overhead of dividing the range may outweigh the benefits.
5. **No Duplicates (Optional):**
   * Binary Search assumes that there are no duplicate elements in the list or that, if duplicates exist, you are looking for the first or last occurrence.

In summary, Binary Search is a powerful algorithm for efficiently finding the position of a target score in a sorted list. It requires a sorted list, random access to elements, comparability of elements, and assumes no duplicates (or a specific strategy for handling duplicates).

**QUESTION 4**

1. **Sorted Data Points:**
   * Ensure that the collected data points are sorted based on the parameter of interest. This could be, for example, time, concentration, temperature, or any other relevant parameter.
2. **Identifying the Target Data Point:**
   * When you want to locate a specific data point, you need to know the value of the parameter associated with that data point.
3. **Binary Search Algorithm:**
   * Apply the Binary Search algorithm to efficiently locate the target data point in the sorted dataset.
   * The steps involve comparing the parameter value of the middle data point with the target parameter value and adjusting the search range accordingly.
4. **Updating Search Range:**
   * If the parameter value of the middle data point is equal to the target parameter value, the search is successful, and you have located the specific data point.
   * If the parameter value of the middle data point is greater than the target parameter value, update the search range to the lower half of the current range.
   * If the parameter value of the middle data point is less than the target parameter value, update the search range to the upper half of the current range.
5. **Repeat:**
   * Repeat the comparison and updating steps until the target data point is found or the search range becomes empty, indicating that the data point is not in the dataset.

**Scalability of Binary Search for Large Datasets:**

Binary Search is highly scalable for large datasets due to its logarithmic time complexity. The time complexity of Binary Search is O(log n), where n is the number of elements in the dataset. This means that as the size of the dataset increases, the time it takes to perform Binary Search grows at a much slower rate compared to linear search algorithms.

Advantages of Binary Search Scalability:

1. **Efficiency:**
   * Binary Search is more efficient than linear search, especially as the dataset size increases. It dramatically reduces the number of comparisons needed to locate a specific data point.
2. **Consistent Performance:**
   * The time complexity of Binary Search remains relatively constant as the dataset grows. It's particularly advantageous when dealing with large datasets, where the time complexity of linear search algorithms would grow linearly with the dataset size (O(n)).
3. **Reduced Search Time:**
   * Binary Search's logarithmic time complexity ensures that the search time scales well with large datasets. This makes it suitable for applications where quick retrieval of specific data points is essential.

In summary, Binary Search is a scalable and efficient algorithm for locating specific data points in large datasets, especially when the data points are sorted based on a parameter. Its logarithmic time complexity makes it well-suited for applications in scientific experiments and other scenarios where quick and efficient data retrieval is crucial.

Top of Form