ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

NAME	SAAD SHAH
ROLL NO	12373
SUBJECT	DSA
ASSIGNMENT NO	03
SUBMITTED TO	MR JAMAL ABDUL AHAD
DATED	11-19-2023

Q 1.When designing a user authentication system, explain how a hash table can store user credentials securely. Discuss the use of hash functions in password hashing and the importance of collision resistance.

ANSWER:

When designing a user authentication system, hash tables can be utilized to store user credentials securely by employing hash functions for password hashing. Here's a breakdown of how this process works and why it's important:

1. Hashing Passwords:

Hash Functions: A hash function takes an input (in this case, a password) and generates a fixed-size string of characters, known as a hash value or digest. The hashing process is a one-way function, meaning it's computationally infeasible to reverse the hash to obtain the original password.

Storing Hashed Passwords: Instead of storing plain-text passwords, the system stores the hashed values in the hash table. When a user registers or changes their password, the system hashes the password using a secure hash function and stores the hash in the table.

2. Use of Hash Functions:

Security: A secure hash function ensures that even if the hash value is compromised, it's challenging to reverse-engineer the original password. Strong hash functions make it computationally unfeasible for attackers to deduce the password from the hash.

Salt: To enhance security further, a random salt (a unique value) can be added to each password before hashing. This prevents attackers from using precomputed rainbow tables (a list of precomputed hashes for commonly used passwords) because each password hash is unique due to the added salt.

3. Importance of Collision Resistance:

Collision: A collision occurs when two different inputs produce the same hash output. In password hashing, collision resistance is crucial because having different passwords resulting in the same hash could lead to authentication issues.

Relevance to Security: Strong hash functions maintain collision resistance, ensuring that it's highly improbable for attackers to find two different passwords generating the same hash. If a hash function lacks collision resistance, attackers could potentially find different passwords that produce the same hash, compromising the system's security.

4. Secure Hash Algorithms:

- Examples of strong hash functions include SHA-256, SHA-3, bcrypt, and Argon2. These algorithms have been vetted by cryptographic experts and are designed to resist attacks, including pre-image attacks (recovering the original input from the hash) and collisions.

Summary:

Using hash tables to store user credentials securely involves employing robust hash functions like SHA-256 or bcrypt to hash passwords before storing them. Collision-resistant hash functions are crucial to ensure that different passwords don't produce the same hash, thereby maintaining the security of the authentication system.

In essence, by leveraging secure hash functions, salting passwords, and ensuring collision resistance, a hash table can securely store user credentials, mitigating the risks associated with storing plain-text passwords and enhancing overall system security.

Q 2: In a task scheduling application, describe how a hash table can be used to store and quickly retrieve scheduled tasks based on their unique identifiers or names. Discuss the advantages of using a hash table for task management.

ANSWER:

1. Storing Tasks in a Hash Table:

- **Key-Value Pair Structure:** The hash table can use task identifiers or names as keys and the corresponding task details (such as task information, scheduling parameters, etc.) as values.
- **Hashing Function:** The hash function can take the task identifier or name and convert it into an index within the hash table. This allows for quick storage and retrieval of tasks.

2. Advantages of Using a Hash Table:

- **Fast Retrieval:** Hash tables provide constant-time average-case complexity for insertion, deletion, and retrieval of elements. This means that regardless of the size of the hash table, the time taken to retrieve a specific task is typically very efficient.
- **Optimized Search:** Tasks can be accessed directly by their identifiers without needing to iterate through a list or perform time-consuming searches. The hash function maps the identifier directly to the location where the task is stored.
- **Space-Time Tradeoff:** When designed appropriately, hash tables can offer a balance between memory usage and quick access to data. They use memory efficiently while ensuring rapid access to tasks.

Example Scenario:

Suppose you have a task scheduling application where each task has a unique identifier (e.g., task ID or name). The hash table would allow you to quickly access specific tasks:

- **Storing Tasks:** When a new task is scheduled, its details are stored in the hash table with its unique identifier as the key and task information as the value.
- **Retrieval:** When you need to access a particular task, you provide its identifier. The hash function immediately determines the location in the hash table where the task details are stored, facilitating rapid retrieval.

Q3: Suppose you have a sorted list of student exam scores. Explain how Binary Search can be applied to identify the position of a particular score in the list. Discuss any assumptions or requirements for using Binary Search in this scenario.

ANSWER:

Binary Search is a powerful algorithm used to find the position of a specific element within a sorted list efficiently. Here's how it can be applied to identify the position of a particular score in a sorted list of student exam scores:

How Binary Search Works:

1. **Initial Steps:**
   - Binary Search begins by examining the middle element of the sorted list.
   - If the middle element matches the target score, the search ends.
   - If the target score is less than the middle element, the search continues in the lower half of the list; otherwise, it continues in the upper half.
2. **Divide and Conquer:**
   - The algorithm repeatedly divides the search interval in half until the target score is found or the interval becomes empty.
3. **Efficiency:**
   - Binary Search is highly efficient as it discards half of the elements at each step, making it a logarithmic time complexity algorithm (O(log n)) for searching in a sorted list.

Requirements and Assumptions for Binary Search:

1. **Sorted List:** Binary Search requires the list to be sorted in ascending or descending order. This assumption is vital for dividing the list and deciding which half to search in.
2. **Random Access:** Binary Search assumes that the elements of the list can be accessed randomly, meaning that accessing the middle element or any element based on its index is possible in constant time.
3. **Comparable Elements:** It assumes the elements in the list are comparable, i.e., you can determine if one element is greater than, less than, or equal to another element

Applying Binary Search to Identify a Score's Position:

Suppose you have a sorted list of student exam scores, for example

65	70	75	80	85	90	95	100

1. To find the position of a specific score (e.g., 85):
   - Begin by examining the middle element (**80**) of the list.
   - As 85 is greater than 80, the search continues in the upper half of the list (**[85, 90, 95, 100]**).
   - The algorithm proceeds by checking the middle element (**90**).
   - As 85 is smaller than 90, the search continues in the lower half (**[85]**).
2. Finally, the algorithm identifies the position of the score **85** in the sorted list, indicating it's at index 4 (considering zero-based indexing).

Q 4: In a scientific experiment, data points are collected and sorted based on a parameter. Explain how Binary Search could be applied to locate specific data points efficiently. Discuss the scalability of Binary Search for large datasets.

ANSWER:

Binary Search is a highly efficient algorithm used to locate a specific element within a sorted array or list of elements by repeatedly dividing the search interval in half. This algorithm works particularly well when the data is already sorted.

In the context of a scientific experiment where data points are collected and sorted based on a parameter, Binary Search could be applied as follows:

1. Initial Sort: The collected data points need to be sorted based on the parameter by which you want to conduct the search.

2. Implementation of Binary Search:

- Identify the middle element of the sorted data.

- Compare the parameter of interest with the middle element.

- If the parameter matches the middle element, the search is successful.

- If the parameter is less than the middle element, then the search continues on the lower half of the sorted array.

- If the parameter is greater than the middle element, then the search continues on the upper half of the sorted array.

- Repeat these steps until the desired element is found or until the search interval becomes empty.

3. Efficiency of Binary Search: Binary Search has a time complexity of O (log n), where 'n' is the number of elements in the dataset. This means that as the dataset size increases, the time taken by Binary Search to find an element grows logarithmically rather than linearly. Therefore, for large datasets, Binary Search is highly efficient compared to linear search algorithms.

4. Scalability for Large Datasets: Binary Search scales efficiently for large datasets because its time complexity grows very slowly as the dataset size increases. Even with a substantial increase in the number of elements, the number of operations required by Binary Search does not grow proportionally. This makes it a highly scalable algorithm for searching within large datasets, providing a significant advantage over linear search methods.