Unit 7
Hashing-7

Gulshan Sharma , Assistant Professor
Computer Science & Engineering

Parul® University

DIGITAL LEARNING CONTENT

UNIT-7

Hashing

Hash Table Organizations

Hash table organization is a fundamental concept in data structures and involves the arrangement and management of data within a hash table. Here's a breakdown of key aspects of hash table organization.

1 **Hash Function**: The hash function is at the core of a hash table. It maps keys to indices in the hash table's array. A good hash function aims to distribute keys evenly across the array to minimize collisions.

2 **Array Size**: The size of the array used in the hash table affects its performance. A larger array can reduce the likelihood of collisions but may consume more memory. The array size is typically chosen based on the expected number of elements and the desired trade-off between memory usage and performance.

Hash Table Organizations

3 **Collision Resolution**: Collisions occur when two keys are mapped to the same index by the hash function. There are several methods for resolving collisions:

Separate Chaining: Each bucket in the hash table array contains a linked list (or another data structure) to handle multiple elements hashed to the same index.

Open Addressing: In this approach, when a collision occurs, the algorithm probes the array for an alternative location to store the collided element. This can involve linear probing, quadratic probing, or other techniques.

Hash Table Organizations

Robin Hood Hashing: An extension of open addressing that seeks to reduce the variance in probe lengths by "robbing" elements from bins with shorter probe lengths and placing them in bins with longer probe lengths.

Cuckoo Hashing: This method involves maintaining multiple hash functions and utilizing multiple hash tables to resolve collisions by relocating elements to other tables.

Hash Table Organizations

4 **Load Factor**: The load factor of a hash table is the ratio of the number of elements stored in the table to the total number of slots (or buckets) in the array. It influences the likelihood of collisions and affects the efficiency of operations. A common practice is to resize the hash table when the load factor exceeds a certain threshold to maintain performance.

5 **Resizing**: As the number of elements stored in the hash table increases or decreases, resizing may be necessary to maintain an appropriate load factor and performance. Resizing involves creating a new, larger array and rehashing the elements from the old array into the new one.

Hash Table Organizations

6 **Key-Value Pairs**: Hash tables often store key-value pairs, where each key is associated with a value. The hash function is applied to the keys to determine their storage location in the array.

Overall, effective hash table organization requires careful consideration of the hash function, collision resolution strategy, array size, load factor, and resizing policy to achieve optimal performance and memory usage for a given application.

Hashing Functions

Hashing functions play a crucial role in hash tables and other hashing-based data structures. They are responsible for converting keys (such as strings or integers) into indices within the hash table's array. Here are some common characteristics and types of hashing functions.

Characteristics of a Good Hash Function

Deterministic: A hashing function should always produce the same hash value for the same input key.

Efficient: Hashing functions should be computationally efficient to calculate, ensuring that hashing operations can be performed quickly.

Uniform Distribution: Ideally, a hashing function should evenly distribute keys across the hash table's array, reducing the likelihood of collisions.

Minimal Collisions: While collisions (multiple keys mapping to the same hash value) are inevitable, a good hashing function should minimize collisions, especially for different keys.

Common Hash Functions

1 **Division Method**: One of the simplest hashing functions involves taking the remainder of the key divided by the size of the hash table's array:

hash(key) = key % array_size

While easy to implement, this method may lead to clustering and poor distribution, especially if the array size is not prime.

2 **Multiplication Method**: This method involves multiplying the key by a constant \(A \) and taking the fractional part of the product, then multiplying by the size of the array:

hash(key) = floor(array_size * ((key * A) mod 1))

.

Common Hash Functions

3 **Universal Hashing :** Universal hashing involves randomly selecting a hashing function from a family of hash functions. This approach aims to provide strong probabilistic guarantees against collisions.

4 **Cryptographic Hash Functions:** These hashing functions are designed to produce a fixed-size hash value (digest) from an input of arbitrary size. Examples include SHA-256 and MD5. Cryptographic hash functions have properties such as collision resistance and preimage resistance, making them suitable for security applications.

Common Hash Functions

5 **Custom Hash Functions**: Depending on the characteristics of the keys and the application requirements, custom hashing functions may be designed to achieve better distribution and performance.

Choosing an appropriate hashing function depends on factors such as the nature of the keys, the size of the hash table, and performance considerations. It's essential to evaluate the distribution of hash values and collision rates to ensure the effectiveness of the chosen hashing function.

Static and Dynamic Hashing

Static Hashing:

Static hashing, also known as perfect hashing, is a hashing technique where each key is mapped directly to a unique slot in the hash table. Unlike traditional hashing methods, static hashing ensures that there are no collisions, which simplifies retrieval operations and guarantees constant-time access.

Here's an example of static hashing:

Suppose we have a set of keys that we want to store in a hash table:

Static Hashing

Keys: {10, 22, 31, 4, 15, 28, 17, 88, 59}

We want to create a hash table to store these keys with minimal collisions. We decide to use static hashing with a hash function that maps each key directly to its value.

1 **Hash Function**: For static hashing, the hash function is simple: it directly maps each key to its value. For example:

hash(key) = key

In this case, the hash value of a key is the key itself.

Static Hashing

2 **Hash Table Creation**: Based on the range of keys, we create a hash table with slots corresponding to each possible key value. In this example, the keys range from 4 to 88, so we create a hash table with slots from 4 to 88.

3 **Insertion**: We insert each key into its corresponding slot in the hash table based on the hash function. Since static hashing guarantees no collisions, each key is inserted directly into its assigned slot.

Static Hashing

Slot 4: 4
Slot 10: 10
Slot 15: 15
Slot 17: 17
Slot 22: 22
Slot 28: 28
Slot 31: 31
Slot 59: 59
Slot 88: 88

As you can see, each key is stored directly in its assigned slot without any collisions.

Static Hashing

4 Retrieval: Retrieving a key from a static hash table is straightforward. We calculate the hash value of the key, which is the key itself, and then access the corresponding slot in the hash table.

For example, if we want to retrieve the key 15, we calculate the hash value:

hash(15) = 15

We access slot 15 in the hash table, which contains the key 15.

Static Hashing

Static hashing is efficient for datasets where the keys are known in advance and do not change frequently. It provides constant-time access without the need for collision resolution mechanisms. However, it may not be suitable for dynamic datasets where keys are inserted or deleted frequently, as resizing the hash table can be challenging.

Dynamic Hashing:

Dynamic hashing is a technique used in computer science to handle data that might grow or shrink unpredictably. It's particularly useful in scenarios where you need to efficiently store and retrieve data in a hash table, but you don't know beforehand how many items will be stored or what their distribution will be like.

In dynamic hashing, the number of buckets in the hash table is not fixed. Instead, it adjusts dynamically based on the number of items being stored and retrieved. This helps in maintaining a good balance between space efficiency and lookup efficiency.

Here's how dynamic hashing works with a simplified example:

Dynamic Hashing

Let's say we're implementing a hash table to store the names of students along with their corresponding grades in a class. We want to efficiently retrieve the grade of any student given their name.

1 **Initialization**: Initially, we start with a small number of buckets in our hash table. Let's say we start with 4 buckets

2 **Hashing**: Each student's name is hashed to determine which bucket it should go into. For simplicity, let's use a basic hash function that takes the first letter of the student's name and maps it to a bucket

Dynamic Hashing

3 **Insertion**: We start inserting students into buckets based on their hashed values. For example:

"Alice" hashes to bucket 1
"Bob" hashes to bucket 2
"Charlie" hashes to bucket 3
"David" hashes to bucket 4
"Eva" also hashes to bucket 1 (hash collision)

Dynamic Hashing

4 **Collision Handling**: When two or more items hash to the same bucket, we typically use techniques like chaining (maintaining a linked list of items in each bucket) or open addressing (probing for an empty bucket nearby) to handle collisions.

5 **Expansion**: As more items are inserted into the hash table, if the load factor (the ratio of the number of items to the number of buckets) exceeds a certain threshold, we dynamically increase the number of buckets. For example, if the load factor exceeds 0.75, we can double the number of buckets to 8.

6 **Rehashing**: When expanding the number of buckets, all existing items need to be rehashed and redistributed into the new bucket structure. This ensures that the distribution remains balanced and efficient.

Dynamic Hashing

7 **Lookup**: When we want to retrieve the grade of a student, we hash their name to find the corresponding bucket and then search within that bucket for the student's name.

8 **Deletion and Contraction**: Similarly, if the number of items decreases significantly, we can dynamically reduce the number of buckets to save space. This involves redistributing items and possibly rehashing again to maintain efficiency.

Dynamic hashing ensures that the hash table can adapt to changing storage requirements while still providing efficient lookup and insertion times. It's a powerful technique used in many real-world applications where the size and distribution of data can vary unpredictably.

Extendible Hashing

Directory: An array of pointers to buckets.

Buckets: Store actual entries.

Directory Doubling: When a bucket overflows, the directory size is doubled, and existing buckets are split based on a bit from the hash value.

Bucket Splitting: Only the overflowing bucket is split, reducing the overhead compared to resizing the entire table.

Linear Hashing

Buckets: Organized in a sequence.

Level: Indicates the current round of splitting.

Splitting Rule: Buckets are split one at a time in a linear order, which allows for gradual growth.

Hash Functions: Two hash functions, hh and $h'h'$, are used. When a bucket overflows, the next bucket in the sequence is split.

Parul® University

DIGITAL LEARNING CONTENT

Conclusion

Hashing is a fundamental technique in computer science for efficient data retrieval. Understanding the various hash table organizations, hashing functions, and the differences between static and dynamic hashing is crucial for designing effective and scalable data structures

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in