> Oct 4, 2024

> Samrat Baral ID #005030064

> MSCS532 Algorithms and Data Structures - First Bi-term

> Assignment 7

Hash Table Key Concepts and Strategies

Introduction:

A hash Table is a data structure that behaves like a dictionary using key-value pairs mapping those data for insertion, update, and deletion. The hashing concept credit goes to Arnold Dumey, who discussed using remainder modulo a prime as a hash function. Therefore, sometimes hashing algorithms prefer the size to be a prime number. In computer language, it is an associative array that stores a set of (key, value) pairs and allows insertion, deletion, and lookup (search) with the constraint of unique keys. It requires uniform distribution for the hashing function, or else. It will increase the complexity and collision cost. (GeeksforGeeks) The relation size, hash function, and proper table resizing determine a hash table's load factor. The table may be cluttered, so the hash function should avoid clustering. It causes high lookup costs, even though the load factor is low and collisions are infrequent. So, concepts like open addressing mitigate and avoid clustering. Therefore, A search algorithm that uses hashing consists of a part for a key into an array index and another for collision resolution. (Wikipedia)

1. Hash Functions and Their Impact:

Designing Effective Hash Functions:

Hash functions play a critical role in determining the efficiency of a hash table by giving them keys and array indices, known as a hash function. A good hash function has efficient computing and uniformly distributes the keys. If correctly, it is called a perfect hash function (Geeksforgeek). The most practical usage of a hash function is with heuristic techniques, which include division and multiplication. The best results with the division method are achieved when the table size is prime. If many keys can have the same hash, then it causes collision. To mitigate the issue, we keep the key uniformly distributed, which keeps hashing and critical constant. Hence, the hashing function must operate the hashing efficiently. It is hard to read the hash value; instead, it is good practice to attempt to guess the key using the hash value rather than retrieving the original hash. Thus, hashing functions should be flexible as data changes or the formatting scheme changes.

Balancing Speed and Complexity:

We know now that hashing is efficient and effective if the hash function is used on uniform distribution and the key's hash is on the table without collision. Some factors affect real-world scenarios that need trade-offs in designing hash functions that are both computationally efficient and resistant to collisions. In a hash table, information can frequently be used with o(1) time, which is rapid access. The time complexity of O(1) is, on average, the case for insertion, deletion, and update, but the worst case is O(n) due to more extensive data of n-size elements. When more or equal to two keys have the same hashing, a collision impacts the application's speed and complexity. There are techniques such as open addressing or open hashing, close hashing, known as separate chaining, and Robin Hood hashing.

2. Open Addressing vs. Separate Chaining:

Comparing Collision Resolution Strategies:

Open addressing and separate chaining are two primary methods for handling collisions in hash tables. As we know from the above, let us discuss its techniques more. Separate chaining: Add the collided element and the new key to the link list. Why is it done so? Because sometimes, it is unknown how many keys are required when a collision happens. For some practical, such as a password storage system, a slower cryptographic hash function is essential to ensure data security, whereas, for a caching mechanism where speed is critical, a faster hash function with acceptable collision rates is more suitable. Which method is correct? Then, we might have to consider memory constraints, dataset flexibility, and average vs. worst. Case: In applications like caching or dictionaries, where average-case performance is critical, separate chaining can often provide more consistent access times.

Performance in Practice:

Assembling the chain structure ensures the hash table is never complete or needs more elements. Nevertheless, the performance increased due to increased complexity and space, but it is straightforward to implement. Similarly, open-addressing but different implementations ensure collision resolution by keeping in the hash table contact, with the size never being equal to or less than the number of keys present. Linear, double, and quadratic hashing are techniques to mitigate collisions. Compared to linear probing, quadratic probing has a worse cache performance, even though clustering is less concerned with quadratic probing. In contrast, linear has the clustering issue, and the time to find the empty slot is incremental, which happens in the iterative for an empty slot. Double hashing is hashing twice. It can increase time complexity while implying that more space is required. Conversely, separate chaining maintains efficient average-case performance regardless of the load factor, provided that the underlying list is well-managed.

Implementation

Hash table implementation in Python using separate chaining (linked lists) for collision resolution. (GeeksforGeeks)

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]  # Initialize hash table with empty lists for separate chaining

    # Hash function to map a key to an index
    def hash_function(self, key):
        return key % self.size

    # Function to insert a key into the hash table
    def insert(self, key):
        index = self.hash_function(key)
        self.table[index].append(key)
    # Function to remove a key from the hash table
    def remove(self, key):
        index = self.hash_function(key)
        if key in self.table[index]:
            self.table[index].remove(key)
        else:
            print(f"Key {key} not found in the hash table.")

    # Function to display the hash table
    def display(self):
        for i in range(self.size):
            print(f"Bucket {i}: {self.table[i]}")

# Example usage
if __name__ == "__main__":
    hash_table = HashTable(7)  # Create a hash table with 7 buckets

    # Inserting keys into the hash table
    hash_table.insert(15)
    hash_table.insert(11)
    hash_table.insert(27)
    hash_table.insert(8)
    hash_table.insert(12)
    hash_table.insert(33)

    print("Hash Table:")
    hash_table.display()

    # Removing a key from the hash table
    hash_table.remove(12)
    print("\nAfter deleting 12:")
    hash_table.display()
```

GeeksforGeeks

Explanation:

- **Initialization**: The `HashTable` class initializes a list of empty lists (`self.table`) to represent the buckets for separate chaining.
- **Hash Function**: The `hash_function` computes the index by taking the modulus of the key with the size of the table (`key % self.size`).
- **Insertion**: The `insert` method appends the key to the list at the calculated index. This demonstrates how collisions are handled by adding keys to the same bucket.
- **Deletion**: The `remove` method checks if the key exists in the calculated bucket and removes it. If the key is not found, it prints a message.
- **Display**: The `display` method iterates through each bucket and prints its contents.

Output

```
Hash Table:
Bucket 0: []
Bucket 1: [15]
Bucket 2: [8]
Bucket 3: []
Bucket 4: []
Bucket 5: [27, 12]
Bucket 6: [11, 33]

After deleting 12:
Bucket 0: []
Bucket 1: [15]
Bucket 2: [8]
Bucket 3: []
Bucket 4: []
Bucket 5: [27]
Bucket 6: [11, 33]
```

Analysis:

- **Separate Chaining**: This code illustrates how separate chaining works by using lists within a list to handle collisions.
- **Hash Function Design**: The simple modulus-based hash function (`key % size`) demonstrates the importance of choosing an appropriate table size (preferably a prime number) to reduce collisions.
- **Collision Resolution**: By inserting multiple keys that hash to the same index (e.g., keys `27` and `12` both hash to bucket `5`), the example shows how collisions are resolved using separate chaining.
- **Efficiency Considerations**: The example emphasizes the trade-off between simplicity and efficiency in hash table implementations, reflecting the discussions on balancing speed and complexity.
- **Remarks:** Observe the output, which displays the hash table before and after deleting the key `12`.

Conclusion

Hash tables are vital for efficient data retrieval, relying on practical hash functions and collision resolution strategies. Good hash functions ensure uniform distribution and minimize collisions, while open addressing and separate chaining offer different approaches to handle collisions, each with unique advantages depending on memory constraints and data characteristics. Understanding the intricacies of hash functions and collision resolution methods is crucial for optimizing hash table performance in various applications. By selecting appropriate strategies, developers can enhance efficiency and reliability, ensuring that hash tables effectively meet the demands of diverse real-world scenarios. (GeeksforGeeks)

References:

"Hash Table Data Structure." GeeksforGeeks, GeeksforGeeks, 15 Sept. 2024, https://www.geeksforgeeks.org/hash-table-data-structure/.

"What Are Hash Functions and How to Choose a Good Hash Function?" GeeksforGeeks, GeeksforGeeks, 21 Feb. 2023, https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/.

c good hash function. https://pekarenklasok.sk/in-america-frpzfjr/6c88d3-c-good-hash-function

"Hash Table." Wikipedia, Wikimedia Foundation, 27 Sept. 2024, https://en.wikipedia.org/wiki/Hash_table.

"Collision Resolution Techniques." GeeksforGeeks, GeeksforGeeks, 5 Apr. 2024, https://www.geeksforgeeks.org/collision-resolution-techniques/.

c good hash function. https://pekarenklasok.sk/in-america-frpzfjr/6c88d3-c-good-hash-function