Hashing in Data Structures and Algorithms using C++

Table of Contents

1. Introduction to Hashing

2. Hash Functions

3. Collision Resolution Techniques

4. Types of Hash Tables

5. Implementing Hashing in C++

6. Applications of Hashing

7. Advantages and Disadvantages

8. Summary

---

1. Introduction to Hashing

Definition:

- **Hashing**: A technique that converts input data (keys) into a fixed-size string of characters, which is typically a numerical value. The resultant value, known as a hash code, serves as an index for data storage.

Purpose:

- Efficient data retrieval and storage.
- Reducing time complexity in searching operations.

Characteristics:

- Quick data access and insertion.
- Fixed-size output irrespective of input size.

---

2. Hash Functions

Definition:

- **Hash Function**: An algorithm that transforms input data into a hash code.

Properties of Good Hash Functions:

- **Deterministic**: The same input will always produce the same output.
- **Uniformity**: Each output should distribute input data evenly across the hash table.
- **Quick computation**: Should calculate the hash code quickly.
- **Pre-image Resistance**: Hard to derive the original input from the hash code.

Example of Simple Hash Functions:

1. Modulo Operation:

```
int hash(int key) {
return key % table_size;
}
```

2. Polynomial Rolling Hash:

```
int hash(string str) {
int hash_value = 0;
int p = 31; // A prime number
for (int i = 0; i < str.size(); i++) {
hash_value += (str[i] - 'a' + 1) * pow(p, i);
}
return hash_value % table_size;
}
```

---

3. Collision Resolution Techniques

What is Collision?

- **Collision**: Occurs when two different keys generate the same hash value.

Collision Resolution Techniques:

1. **Chaining**:
   - Each index of the hash table points to a linked list of entries.
   - Pros: Easy to implement and handles dynamic data well.
   - Cons: Can cause slow search times if many collisions occur.

Implementation:

```
class HashTable {
vector table; // ListNode is a structure for linked list nodes
};
```

2. **Open Addressing**:
   - All elements are stored in the hash table itself.
   - If collision occurs, probing is used to find the next open slot.

Types of probing:

- **Linear Probing**
- **Quadratic Probing**
- **Double Hashing**

Implementation of Linear Probing:

```
int hash(int key) {
int index = key % table_size;
while (table[index] != nullptr) {
index = (index + 1) % table_size; // Linear probe
}
return index;
}
```

---

4. Types of Hash Tables

1. **Static Hash Table**:

- Fixed size.
- Better for scenarios where the number of elements is predictable.

2. **Dynamic Hash Table**:

- Can grow or shrink in size.
- Suitable for scenarios with unpredictable quantities of data.

3. **Open Hashing**:

- Uses linked lists to resolve collisions.
- Allows for dynamic resizing easily.

4. **Closed Hashing**:

- Uses probing to resolve collisions, storing all data directly in the table.

---

5. Implementing Hashing in C++

Basic Steps:

1. **Define Hash Table Size**.
2. **Create Hash Table Structure**.
3. **Implement Insertion, Deletion, and Search Functions**.

Example Code:

#include

#include

#include

using namespace std;

```
class HashTable {
private:
vector> table; // for chaining
int table_size;

public:
HashTable(int size) : table_size(size) {
table.resize(size);
}

void insert(int key);
bool search(int key);
void remove(int key);
};

void HashTable::insert(int key) {
int index = hash(key);
table[index].push_back(key);
}

bool HashTable::search(int key) {
int index = hash(key);
for (auto k : table[index])
if (k == key)
return true;
return false;
}

void HashTable::remove(int key) {
int index = hash(key);
table[index].remove(key);
```

}

---

6. Applications of Hashing

Common Applications:

- **Database indexing**: To quickly retrieve records.
- **Caches**: Storing recently used data for quick access.
- **Symbol tables**: In compilers to manage variables and identifiers.
- **Cryptography**: Securely storing passwords and verifying data integrity.

---

7. Advantages and Disadvantages

Advantages:

- Faster data retrieval compared to other data structures like arrays and linked lists.
- Efficient for lookups, insertions, and deletions.

Disadvantages:

- Poor hash functions can lead to many collisions, causing performance issues.
- Memory overhead due to unused spaces in larger hash tables.
- Complexity of design and implementation compared to simpler structures.

---

8. Summary

- **Hashing** is a crucial technique in computer science for efficient data management.
- A good **hash function** is key to minimizing collisions.
- Understanding **collision resolution** techniques is essential for implementing effective hash tables.

- Hash tables can be dynamically or statically sized, with various collision resolution strategies.
- Despite its advantages, hashing has challenges, including design complexity and potential collisions.

With this understanding, students can grasp the importance of hashing within data structures and algorithms and apply these concepts effectively in C++. Practice coding hash tables and experimenting with different hash functions and collision resolution techniques to solidify this knowledge.