Hashing in Data Structures

Hashing is a technique used to store and retrieve data efficiently using a **hash function** that converts keys into a fixed-size numerical value (hash code). This hash code is used as an index in a **hash table**, enabling **constant-time (O(1)) average-case complexity** for insertions, deletions, and lookups.

Key Components of Hashing

1. **Hash Function**: Maps a key to an index in the hash table.
2. **Hash Table**: An array where data is stored based on the computed index.
3. **Collision Handling**: Since different keys can produce the same index, techniques like chaining and open addressing are used.

Types of Hashing and C++ Examples

1. Direct Hashing

In **direct hashing**, the key itself is used as the index. It is used when the key values are within a small range.

Example:

```cpp
#include <iostream>
using namespace std;

#define SIZE 100  // Assuming keys range from 0-99

int hashTable[SIZE] = {0};  // Initialize hash table

void insert(int key, int value) {
    hashTable[key] = value;  // Directly store at index = key
}

int search(int key) {
    return hashTable[key];  // Directly retrieve value
}

int main() {
    insert(25, 100);
    insert(50, 200);

    cout << "Value at key 25: " << search(25) << endl;
    cout << "Value at key 50: " << search(50) << endl;

    return 0;
```

```
}
```

□ **Best for**: Small, fixed key ranges (e.g., student roll numbers).
□ **Downside**: Inefficient if keys are large and sparse.

---

2. Modulo Division Hashing

This method calculates the index as:

index=keymod table_size\text{index} = \text{key} \mod \text{table_size}

It distributes keys more evenly across the table.

Example:

```
#include <iostream>
using namespace std;

#define TABLE_SIZE 10  // Hash table of size 10

int hashTable[TABLE_SIZE] = {0};

int hashFunction(int key) {
    return key % TABLE_SIZE;  // Modulo operation
}

void insert(int key, int value) {
    int index = hashFunction(key);
    hashTable[index] = value;
}

int search(int key) {
    int index = hashFunction(key);
    return hashTable[index];
}

int main() {
    insert(25, 100);
    insert(50, 200);

    cout << "Value at key 25: " << search(25) << endl;
    cout << "Value at key 50: " << search(50) << endl;

    return 0;
```

```
}
```

□ **Best for**: Evenly distributed data (e.g., hashing user IDs).
□ **Downside**: **Collisions** occur when multiple keys map to the same index.

---

3. Multiplication Hashing

This method uses a constant **A** (between 0 and 1) to generate an index:

index=⌊table_size×(key×Amod 1)⌋\text{index} = \lfloor \text{table_size} \times (key \times A \mod 1) \rfloor

It avoids clustering issues common in modulo hashing.

Example:

```
#include <iostream>
#include <cmath>
using namespace std;

#define TABLE_SIZE 10
const double A = 0.6180339887;  // Commonly used constant

int hashTable[TABLE_SIZE] = {0};

int hashFunction(int key) {
    return floor(TABLE_SIZE * fmod(key * A, 1));  // Multiplication method
}

void insert(int key, int value) {
    int index = hashFunction(key);
    hashTable[index] = value;
}

int search(int key) {
    int index = hashFunction(key);
    return hashTable[index];
}

int main() {
    insert(25, 100);
    insert(50, 200);

    cout << "Value at key 25: " << search(25) << endl;
```

```
    cout << "Value at key 50: " << search(50) << endl;

    return 0;
}
```

 Best for: Uniform distribution of keys.
 Downside: More complex than modulo hashing.

---

4. Collision Handling (Chaining)

If two keys hash to the same index, they are stored in a linked list at that index.

Example (Separate Chaining with Linked List):

```
    #include <iostream>
#include <list>
using namespace std;

#define TABLE_SIZE 10

class HashTable {
    list<int> table[TABLE_SIZE];

public:
    int hashFunction(int key) {
        return key % TABLE_SIZE;  // Modulo division
    }

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

    void search(int key) {
        int index = hashFunction(key);
        for (int val : table[index]) {
            if (val == key) {
                cout << key << " found at index " << index << endl;
                return;
            }
        }
        cout << key << " not found" << endl;
    }

    void display() {
```

```cpp
        for (int i = 0; i < TABLE_SIZE; i++) {
            cout << "Index " << i << ": ";
            for (int val : table[i])
                cout << val << " -> ";
            cout << "NULL" << endl;
        }
    }
};

int main() {
    HashTable h;
    h.insert(10);
    h.insert(20);
    h.insert(30);
    h.insert(40);

    h.display();
    h.search(20);
    h.search(25);

    return 0;
}
```

 Best for: Handling frequent collisions.
 Downside: Requires additional memory (linked list).

Conclusion

Hashing Method	Pros	Cons
Direct Hashing	Fast lookups	Wastes memory for large keys
Modulo Hashing	Simple, effective	Can cause collisions
Multiplication Hashing	Distributes keys better	More complex than modulo
Chaining (Linked List)	Handles collisions well	Extra memory overhead

Each hashing method is suitable for different scenarios. Which one do you need help with?