HASHING

Hashing is a technique in data structures used to map data (keys) to specific locations (hash indices) in a data structure (typically a hash table). The primary goal of hashing is to enable fast data retrieval, insertion, and deletion, usually in O(1) average time complexity.

KEY CONCEPTS OF HASHING

HASH FUNCTION:
A function that takes an input (key) and returns an integer (hash code). This hash code is mapped to an index in the hash table.

1. **Division (Modulo) Method:**
   The hash value is obtained by taking the remainder of the key divided by the table size.
   Formula:

   $$h(k)=k \bmod m$$

2. **Multiplication Method:**
   Multiply the key by a constant, extract the fractional part, and scale it to the table size.
   Formula:

   $$h(k)=\lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

   A is a constant, often 0.618033...0.618033...0.618033... (the fractional part of the golden ratio).

3. **Mid-Square Method:**
   Square the key and extract the middle digits to determine the hash value.
   Example:

   Key: 123

   Square: 1232=15129123^2 = 151291232=15129

   Middle digits: 512 (hash value).

4. **Folding Method:**
   Break the key into smaller parts, add the parts, and use modulo to compute the index.
   Example:

   Key: 987654

   Parts: 987,654987, 654987,654

Sum: 987+654=1641987 + 654 = 1641987+654=1641

Index: 1641 mod m

HASH TABLE:
A data structure where the hash codes from the hash function determine the placement of keys or values. It can be implemented using arrays or linked lists.

COLLISION RESOLUTION TECHNIQUE

When two keys produce the same hash code, a collision occurs. Collision resolution strategies include:

1. **Chaining:** In this technique, each hash table slot contains a linked list of all the values that have the same hash value. This technique is simple and easy to implement, but it can lead to poor performance when the linked lists become too long.

2. **Open addressing:** In this technique, when a collision occurs, the algorithm searches for an empty slot in the hash table by probing successive slots until an empty slot is found. This technique can be more efficient than chaining when the load factor is low, but it can lead to clustering and poor performance when the load factor is high.

3. **Double hashing:** This is a variation of open addressing that uses a second hash function to determine the next slot to probe when a collision occurs. This technique can help to reduce clustering and improve performance.

APPLICATIONS OF HASHING

1. Databases: For indexing and searching records.
2. Cryptography: Hash functions secure sensitive data (e.g., SHA, MD5).
3. Compiler Design: Symbol tables for storing identifiers and keywords.
4. Caching: Fast lookup in memory or disk caches.

Advantages of Hashing

1. Fast lookups for keys, especially in large datasets.
2. Efficient storage and retrieval in most cases.

3. Versatile for use in a variety of applications.

DISADVANTAGES OF HASHING

1. Collisions can degrade performance.
2. Poor hash function design can lead to uneven key distribution.
3. Limited by the size of the hash table.

CODE IMPLEMENTATION

CHAINING/ SEPARATE CHAINING

```
#include <iostream>
#include <string>
// Node structure for chaining
struct Node {
std::string key;
Node* next;
Node(const std::string& k) : key(k), next(nullptr) {}
};
class HashTable {
private:
static const int TABLE_SIZE = 10; // Size of the hash table
Node* table[TABLE_SIZE]; // Array of pointers to Nodes
// Hash function to calculate the index
int hashFunction(const std::string& key) {
int hashValue = 0;
for (char ch : key) {
hashValue += ch; // Sum ASCII values of characters
}
```

```
return hashValue % TABLE_SIZE;
}
public:
// Constructor to initialize the table
HashTable() {
for (int i = 0; i < TABLE_SIZE; ++i) {
table[i] = nullptr;
}
}
// Destructor to free memory
~HashTable() {
for (int i = 0; i < TABLE_SIZE; ++i) {
Node* current = table[i];
while (current) {
Node* temp = current;
current = current->next;
delete temp;
}
}
}
// Insert a key into the hash table
void insert(const std::string& key) {
int index = hashFunction(key);
Node* newNode = new Node(key);
if (!table[index]) {
table[index] = newNode; // Insert directly if no collision
} else {
```

```
// Collision handling by chaining
Node* current = table[index];
while (current->next) {
current = current->next;
}
current->next = newNode;
}
}
// Search for a key in the hash table
bool search(const std::string& key) {
int index = hashFunction(key);
Node* current = table[index];
while (current) {
if (current->key == key)
return true; // Key found
current = current->next;
}
return false; // Key not found
}
// Remove a key from the hash table
void remove(const std::string& key) {
int index = hashFunction(key);
Node* current = table[index];
Node* prev = nullptr;
while (current) {
if (current->key == key) {
if (prev) {
```

```
prev->next = current->next;
} else {
table[index] = current->next;
}
delete current;
return;
}
prev = current;
current = current->next;
}
}
// Display the hash table
void display() {
for (int i = 0; i < TABLE_SIZE; ++i) {
std::cout << "Bucket " << i << ": ";
Node* current = table[i];
while (current) {
std::cout << current->key << " -> ";
current = current->next;
}
std::cout << "nullptr\n";
}
}
};
// Main function to demonstrate the hash table
int main() {
HashTable ht;
```

```
ht.insert("apple");
ht.insert("banana");
ht.insert("grape");
ht.insert("orange");
ht.insert("melon");
std::cout << "Hash table after insertion:\n";
ht.display();
std::cout << "\nSearching for 'grape': " << (ht.search("grape") ? "Found" : "Not Found") <<
"\n";
std::cout << "Searching for 'cherry': " << (ht.search("cherry") ? "Found" : "Not Found") << "\n";
ht.remove("banana");
std::cout << "\nHash table after removing 'banana':\n";
ht.display();
return 0;
}
```

LINEAR PROBING

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

class HashTable {
private:
    int* table;       // Hash table
    int size;         // Size of the hash table
    int count;        // Number of elements in the hash table

    // Hash function: calculates the index for a given key
```

```
    int hashFunction(int key) {
        return key % size;
    }

public:
    // Constructor: initializes the hash table
    HashTable(int tableSize) {
        size = tableSize;
        count = 0;
        table = new int[size];
        for (int i = 0; i < size; i++) {
            table[i] = -1; // Initialize with -1 to represent an empty slot
        }
    }

    // Insert a key using linear probing
    void insert(int key) {
        if (count == size) {
            cout << "Hash table is full! Cannot insert key " << key << endl;
            return;
        }

        int index = hashFunction(key);

        // Linear probing: find the next empty slot
        while (table[index] != -1) {
            index = (index + 1) % size;
```

```
    }

    table[index] = key;
    count++;
}

// Search for a key in the hash table
bool search(int key) {
    int index = hashFunction(key);

    // Traverse the table using linear probing
    int start = index;
    while (table[index] != -1) {
        if (table[index] == key) {
            return true;
        }
        index = (index + 1) % size;

        // If we've looped back to the start, the key is not present
        if (index == start) {
            break;
        }
    }
    return false;
}

// Display the hash table
```

```
    void display() {
        cout << "Hash Table:" << endl;
        for (int i = 0; i < size; i++) {
            if (table[i] == -1) {
                cout << "Index " << i << ": Empty" << endl;
            } else {
                cout << "Index " << i << ": " << table[i] << endl;
            }
        }
    }

    // Destructor: cleans up memory
    ~HashTable() {
        delete[] table;
    }
};

int main() {
    HashTable ht(7); // Create a hash table of size 7

    // Insert keys
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);
```

```
    // Display the hash table
    ht.display();

    // Search for keys
    cout << "\nSearch for 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;
    cout << "Search for 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;

    return 0;
}
```

QUADRATIC HASING

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

class HashTable {
private:
    int* table;       // Hash table
    int size;         // Size of the hash table
    int count;        // Number of elements in the hash table

    // Hash function: calculates the index for a given key
    int hashFunction(int key) {
        return key % size;
    }

public:
```

```
// Constructor: initializes the hash table
HashTable(int tableSize) {
   size = tableSize;
   count = 0;
   table = new int[size];
   for (int i = 0; i < size; i++) {
      table[i] = -1; // Initialize with -1 to represent an empty slot
   }
}

// Insert a key using quadratic probing
void insert(int key) {
   if (count == size) {
      cout << "Hash table is full! Cannot insert key " << key << endl;
      return;
   }

   int index = hashFunction(key);
   int i = 0;

   // Quadratic probing: find the next empty slot
   while (table[(index + i * i) % size] != -1) {
      i++;
      if (i == size) { // Prevent infinite loop if table is full
         cout << "No available slot for key " << key << endl;
         return;
      }
```

```
    }

    table[(index + i * i) % size] = key;
    count++;
}

// Search for a key in the hash table
bool search(int key) {
    int index = hashFunction(key);
    int i = 0;

    // Traverse the table using quadratic probing
    while (table[(index + i * i) % size] != -1) {
        if (table[(index + i * i) % size] == key) {
            return true;
        }
        i++;
        if (i == size) { // Prevent infinite loop
            break;
        }
    }
    return false;
}

// Display the hash table
void display() {
    cout << "Hash Table:" << endl;
```

```
        for (int i = 0; i < size; i++) {
            if (table[i] == -1) {
                cout << "Index " << i << ": Empty" << endl;
            } else {
                cout << "Index " << i << ": " << table[i] << endl;
            }
        }
    }

    // Destructor: cleans up memory
    ~HashTable() {
        delete[] table;
    }
};

int main() {
    HashTable ht(7); // Create a hash table of size 7

    // Insert keys
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    // Display the hash table
    ht.display();
```

```
    // Search for keys
    cout << "\nSearch for 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;
    cout << "Search for 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;

    return 0;
}
```

DOUBLE HASHING

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

class HashTable {
private:
    int* table;      // Hash table
    int size;        // Size of the hash table
    int count;       // Number of elements in the hash table

    // Primary hash function
    int hash1(int key) {
        return key % size;
    }

    // Secondary hash function
    int hash2(int key) {
        // Ensure the step size is non-zero and relatively prime to the table size
        return 1 + (key % (size - 1));
```

```
    }

public:
    // Constructor: initializes the hash table
    HashTable(int tableSize) {
        size = tableSize;
        count = 0;
        table = new int[size];
        for (int i = 0; i < size; i++) {
            table[i] = -1; // Initialize with -1 to represent an empty slot
        }
    }

    // Insert a key using double hashing
    void insert(int key) {
        if (count == size) {
            cout << "Hash table is full! Cannot insert key " << key << endl;
            return;
        }

        int index = hash1(key);
        int stepSize = hash2(key);
        int i = 0;

        // Double hashing: find the next empty slot
        while (table[(index + i * stepSize) % size] != -1) {
            i++;
```

```
        if (i == size) { // Prevent infinite loop if table is full
            cout << "No available slot for key " << key << endl;
            return;
        }
    }

    table[(index + i * stepSize) % size] = key;
    count++;
}

// Search for a key in the hash table
bool search(int key) {
    int index = hash1(key);
    int stepSize = hash2(key);
    int i = 0;

    // Traverse the table using double hashing
    while (table[(index + i * stepSize) % size] != -1) {
        if (table[(index + i * stepSize) % size] == key) {
            return true;
        }
        i++;
        if (i == size) { // Prevent infinite loop
            break;
        }
    }
    return false;
```

```
    }

    // Display the hash table
    void display() {
        cout << "Hash Table:" << endl;
        for (int i = 0; i < size; i++) {
            if (table[i] == -1) {
                cout << "Index " << i << ": Empty" << endl;
            } else {
                cout << "Index " << i << ": " << table[i] << endl;
            }
        }
    }

    // Destructor: cleans up memory
    ~HashTable() {
        delete[] table;
    }
};

int main() {
    HashTable ht(7); // Create a hash table of size 7

    // Insert keys
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
```

```
    ht.insert(7);
    ht.insert(30);

    // Display the hash table
    ht.display();

    // Search for keys
    cout << "\nSearch for 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;
    cout << "Search for 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;

    return 0;
}
```

REHASHING

Rehashing is a technique used to address the issue of hash table overflow and improve performance when the load factor becomes too high. It involves creating a new hash table with a larger size and reinserting all existing elements into it using a new hash function or the same hash function adapted to the new table size.

WHY REHASHING?

1. High Load Factor: When the hash table becomes crowded (load factor exceeds a threshold, e.g., 0.7 or 0.75), collisions increase, degrading performance.

   Load Factor = Number of Elements / Table Size

2. Efficiency: Rehashing reduces collisions and ensures faster insertion, deletion, and search operations.

STEPS FOR REHASHING

1. Determine the New Table Size:

   Typically, the new size is chosen as a prime number greater than twice the current size.

2. Create a New Hash Table:

   Allocate a larger table.

3. Rehash All Elements:

   Traverse the old table and insert all non-empty elements into the new table using the new hash function.

CODE IMPLEMENTATION

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

class HashTable {
private:
   int* table;  // Dynamic array to store hash table
   int size;    // Current size of the hash table
   int count;   // Number of elements in the table
   const float loadFactorThreshold = 0.75;

   // Hash function
   int hashFunction(int key) {
      return key % size;
   }

   // Helper function to find the next prime number
   int nextPrime(int n) {
      while (true) {
         n++;
         bool isPrime = true;
```

```
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) return n;
    }
}

// Rehash the table
void rehash() {
    cout << "\nRehashing...\n";

    // Find new size
    int oldSize = size;
    size = nextPrime(2 * oldSize);

    // Create a new table
    int* oldTable = table;
    table = new int[size];
    for (int i = 0; i < size; i++) {
        table[i] = -1; // Initialize all slots as empty (-1)
    }

    // Reinsert all elements
    for (int i = 0; i < oldSize; i++) {
```

```
            if (oldTable[i] != -1) { // Ignore empty slots
                insert(oldTable[i]);
            }
        }

        // Free memory of old table
        delete[] oldTable;

        cout << "Rehashing complete. New size: " << size << endl;
    }

public:
    // Constructor
    HashTable(int initialSize) {
        size = initialSize;
        count = 0;
        table = new int[size];
        for (int i = 0; i < size; i++) {
            table[i] = -1; // Initialize all slots as empty (-1)
        }
    }

    // Insert a key
    void insert(int key) {
        if (count >= loadFactorThreshold * size) {
            rehash();
        }
```

```
    int index = hashFunction(key);

    // Linear probing for collision resolution
    while (table[index] != -1) {
        index = (index + 1) % size;
    }

    table[index] = key;
    count++;
}

// Search for a key
bool search(int key) {
    int index = hashFunction(key);

    // Linear probing for search
    int start = index;
    while (table[index] != -1) {
        if (table[index] == key) {
            return true;
        }
        index = (index + 1) % size;

        // Stop if we circle back to the start index
        if (index == start) break;
    }
```

```
        return false;
    }

    // Display the hash table
    void display() {
        cout << "Hash Table:" << endl;
        for (int i = 0; i < size; i++) {
            if (table[i] == -1) {
                cout << "Index " << i << ": Empty" << endl;
            } else {
                cout << "Index " << i << ": " << table[i] << endl;
            }
        }
    }

    // Destructor
    ~HashTable() {
        delete[] table;
    }
};

int main() {
    HashTable ht(5); // Initial size of hash table

    // Insert keys
    ht.insert(10);
```

```
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    // Display the hash table
    ht.display();

    // Insert more keys to trigger rehashing
    ht.insert(40);
    ht.insert(50);

    // Display the hash table after rehashing
    ht.display();

    // Search for keys
    cout << "\nSearch for 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;
    cout << "Search for 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;

    return 0;
}
```