Data Structures & Algorithms (C++)

Hashing: Bucket Arrays, Hash Functions, Hash Codes, Compression Functions

1. Bucket Arrays

Introduction

A bucket array is a fundamental structure used in hashing where data is stored in an array of "buckets", and each bucket can hold one or more elements.

Each bucket corresponds to an index generated by a hash function. Instead of searching the whole dataset, we directly jump to a specific bucket.

Bucket arrays are widely used in hash tables, dictionaries, and database indexing systems.

Concept

Index → Bucket
0 → []
1 → []
2 → [10, 20]
3 → []

Why Bucket Arrays are Used

- Fast access using index
- Reduces search time
- Handles large datasets efficiently
- Forms the base of hash tables

Types of Buckets

- Single-slot bucket (one value per index)
- Chaining bucket (linked list per index)

2. Hash Functions

Introduction

A hash function is a mathematical function that converts input data (key) into a fixed-size integer value called a hash code.

This hash code is then used as an index in a bucket array.

A good hash function distributes values uniformly to minimize collisions.

Definition

A hash function maps:

Key → Hash Value → Index

Example

Key = 25
Hash Function = h(x) = x % 10

Result = 25 % 10 = 5

So, key 25 goes to bucket index 5.

Properties of a Good Hash Function

- Fast computation
- Uniform distribution
- Minimizes collisions
- Deterministic (same input → same output)

Example Hash Functions

- Division method: h(k) = k % m
- Multiplication method
- Mid-square method

3. Hash Codes

Introduction

A hash code is an integer representation of an object or key generated before applying compression.

It is the intermediate step between the key and the final index.

Hash codes allow complex data (strings, objects) to be converted into integers.

Example

For string "ABC":

A = 65
B = 66
C = 67

Hash Code = 65 + 66 + 67 = 198

Purpose

- Convert data into numeric form
- Prepare input for hash function
- Enable fast indexing

Characteristics

- Must be consistent
- Should minimize collisions
- Should distribute values evenly

4. Compression Functions

Introduction

A compression function takes a large hash code and reduces it into a valid index range for a bucket array.

Since arrays have fixed size, compression ensures the hash code fits within array limits.

Definition

Hash Code → Compression Function → Index

Example

If:

Hash Code = 198
Table size = 10

Compression:

Index = 198 % 10 = 8

Common Compression Methods

1. Division Method

h(k) = k % m

2. Folding Method

Split number into parts and add them.

3. Mid-square Method

Square key and take middle digits.

Requirements of Good Compression Function

- Uniform distribution
- Reduces collisions
- Keeps values in valid range (0 to m-1)

COMPLETE C++ PROGRAM (HASH TABLE USING BUCKET ARRAY + HASH FUNCTION)

This program demonstrates:

- Bucket array
- Hash function

- Compression function
- Collision handling (chaining)

C++ Implementation

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

class HashTable {
private:
   int size;
   list<int>* table;

   // Hash Function + Compression Function
   int hashFunction(int key) {
      return key % size;
   }

public:
   HashTable(int s) {
      size = s;
      table = new list<int>[size];
   }

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

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

      for (int val : table[index]) {
         if (val == key)
            return true;
      }
      return false;
   }

   // Display hash table
   void display() {
      for (int i = 0; i < size; i++) {
         cout << i << " --> ";
         for (int val : table[i]) {
            cout << val << " ";
         }
         cout << endl;
      }
```

```
    }
};

int main() {
    HashTable ht(10);

    ht.insert(10);
    ht.insert(20);
    ht.insert(30);
    ht.insert(25);
    ht.insert(35);

    cout << "Hash Table:\n";
    ht.display();

    cout << "\nSearch 25: "
         << (ht.search(25) ? "Found" : "Not Found") << endl;

    cout << "Search 100: "
         << (ht.search(100) ? "Found" : "Not Found") << endl;

    return 0;
}
```

Complexity Analysis

Operation	Average Case	Worst Case
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Comparison Table

Concept	Purpose
Bucket Array	Stores data in indexed slots
Hash Function	Converts key to hash value
Hash Code	Numeric representation of key
Compression Function	Maps hash code into array range

Key Takeaways

- Hashing provides **fast O(1) average lookup**
- Bucket arrays store values at computed indices
- Hash functions generate numeric values from keys
- Hash codes represent raw numeric transformation of keys

- Compression functions ensure values fit into array size
- Collisions are handled using chaining (linked lists)
- Hashing is used in:
    - Databases
    - Compilers
    - Caches
    - Dictionaries (STL unordered_map)