Data Structures and Algorithms (Spring2024)

Week13/14 (29/30/30/05-May-June-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Hashing: Motivation

Imagine a library where each book has a unique ISBN and a corresponding shelf location. Let's use the following books:

- ISBN 978-3-16-148410-0: "Introduction to Algorithms"
- ISBN 978-0-262-03384-8: "The Art of Computer Programming"
- ISBN 978-0-201-83595-1: "Clean Code"
- ISBN 978-0-13-110362-7: "The C Programming Language"
- ISBN 978-0-07-032482-8: "Design Patterns"

Using an Array

In an array, books are stored sequentially and you would typically need an index to access each book, which can be disconnected from the book's actual ISBN.

Array Representation:

```
Index:         0                              1                       2
3                       4
Value: "Intro to Algorithms"  "Art of Computer Programming"  "Clean
Code"  "C Programming Language"  "Design Patterns"
```

- **Accessing "Clean Code"**: `array[2]`
- **Disadvantage**: The index `2` has no direct relation to the book's ISBN.

Using Hashing

With hashing, we can use each book's ISBN directly as a key to store and retrieve the book's location. This mimics how you might look up book information in a library system directly by its ISBN.

Hash Table Representation:

```
Key                  | Value
---------------------|-----------------------------------
978-3-16-148410-0    | "Introduction to Algorithms"
978-0-262-03384-8    | "The Art of Computer Programming"
978-0-201-83595-1    | "Clean Code"
978-0-13-110362-7    | "The C Programming Language"
978-0-07-032482-8    | "Design Patterns"
```

- **Accessing "Clean Code"**: `hashTable["978-0-201-83595-1"]`
- **Advantage**: Direct access using ISBN is intuitive, mimics real-life usage, and is efficient.

Advantages of Using Hashing for Library Systems

1. **Meaningful Keys**:
    - The key (ISBN) is directly used, which is how people typically search for books, ensuring that the key has intrinsic meaning and relevance.
2. **Efficiency in Access**:
    - Hashing allows direct and quick retrieval of book information without needing to know its position in a collection, contrary to arrays where the physical position is crucial.
3. **Space Efficiency**:
    - Hash tables efficiently manage space, especially when book collections are large with non-sequential ISBNs, avoiding the sparse array problem.

Summary of Differences

Array (Horizontal Layout):

- **Index Relationship**: Sequential, often arbitrary with respect to book data.
- **Access Method**: Access by numerical index, requiring mapping between index and ISBN.

Hash Table (Horizontal Layout):

- **Index Relationship**: Uses ISBNs directly.
- **Access Method**: Direct access by ISBN, which is natural for users and systems dealing with books.

Conclusion

For applications like library systems where direct access via specific identifiers (like ISBNs) is necessary, hashing offers significant advantages over arrays. It not only enhances access efficiency but also aligns more closely with how data is actually used and retrieved in real-world scenarios, making it a superior choice for handling such data.

What is Hashing?

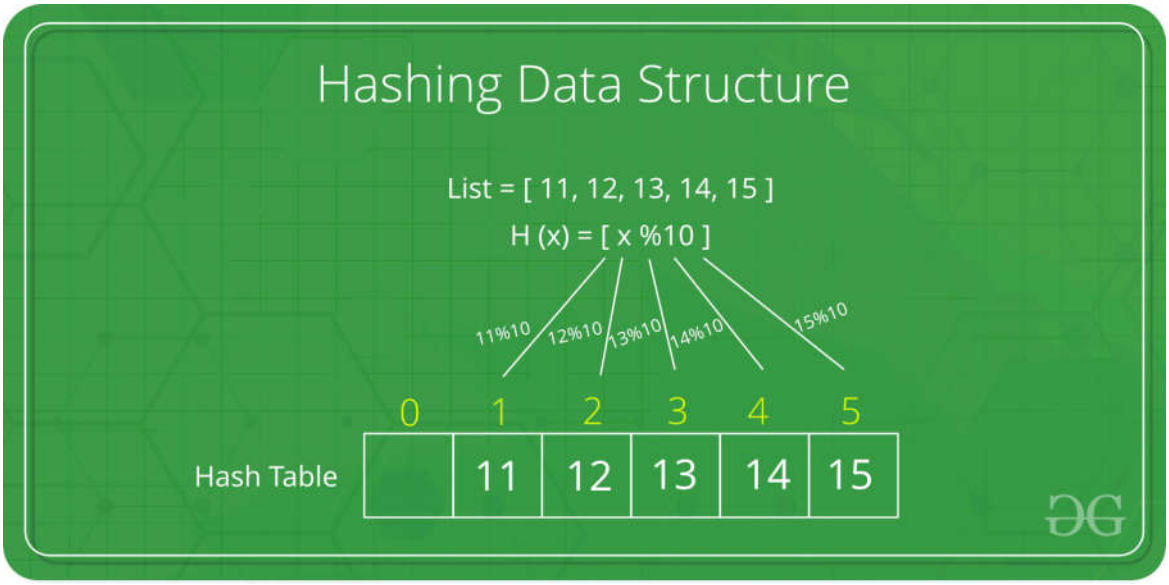
Hashing is a technique used to efficiently store, retrieve, and manage data through a structure called a hash table. It involves converting a given key into an index in an array where the corresponding value is stored. This process is governed by a function known as a hash function.

How It Works

1. **Hash Function:** The core component of hashing is the hash function, which takes an input (or 'key') and returns an index in the hash table array. The efficiency of hashing largely depends on how well the hash function distributes the data across the hash table,

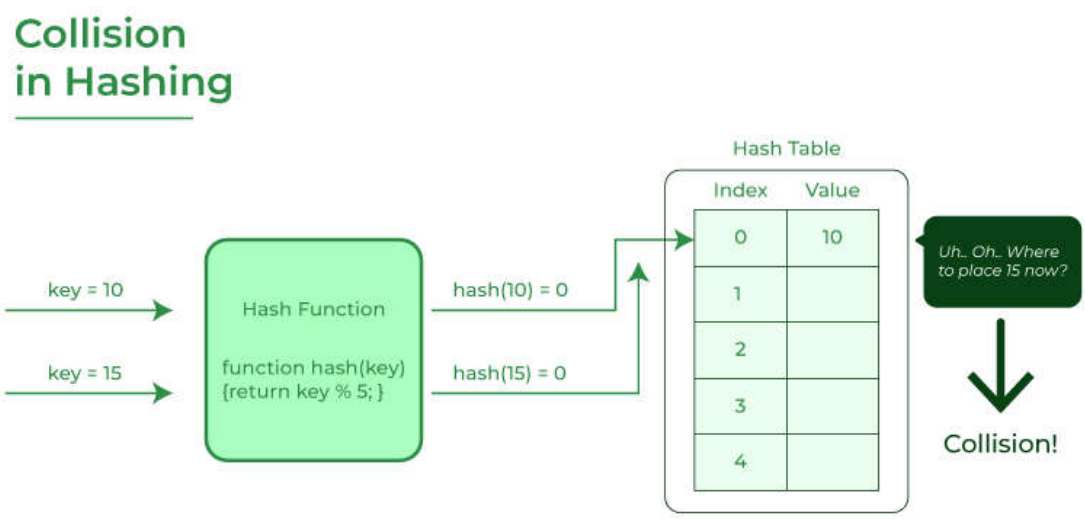
aiming to minimize collisions (where different keys get the same index).

2. **Storing Data:** When you need to store a key-value pair, the hash function calculates the index based on the key and stores the value at that index in the hash table.
3. **Retrieving Data:** To retrieve the value associated with a particular key, the hash function is used again to find the index where the value is stored, allowing for efficient data retrieval.



Elements of Hashing

1. **Hash Table:** An array that stores data where each element has a unique index derived from its key by the hash function.
2. **Hash Function:** A function that computes an index in a hash table from a given key.
3. **Collisions:** Occur when two keys hash to the same index. Handling collisions is a critical aspect of hash table design, typically managed by methods such as chaining (linking entries with the same hash index) or open addressing (finding another slot using a probing sequence).
4. **Load Factor:** The ratio of the number of entries to the capacity of the hash table, which influences its performance. A high load factor may increase collisions, necessitating resizing of the hash table.



Why Do We Need Hashing?

1. **Efficiency:** Hashing provides a very efficient means of data retrieval that, in the best-case scenario, allows for constant time complexity (O(1)) for both insertion and lookup operations.
2. **Data Management:** It facilitates the management of large datasets by providing rapid data access and insertion capabilities.
3. **Resource Optimization:** Reduces the need for extensive searches, thereby saving on computation and time, especially in applications like database indexing, caching, and handling large data sets.

Incorporating hashing into systems where quick data retrieval is crucial can significantly enhance performance and resource management. It's a fundamental concept in computer science, useful in various applications from software development to systems design.

Example Scenario

Imagine a classroom with 10 students. We want to create a hash table to quickly access their student ID by using their names as keys.

Hash Table

We'll use an array with 10 slots (indices from 0 to 9) to represent our hash table.

Hash Function

For simplicity, let's create a hash function that calculates the index by taking the sum of the ASCII values of the characters in the student's name, then taking the modulus of that sum by 10 (size of the hash table).

```cpp
#include <iostream>
#include <vector>
#include <string>


// Hash function to compute index
int hashFunction(const string& name) {
    int sum = 0;
    for (char c : name) {
        sum += c;
    }
    return sum % TABLE_SIZE;
}

// A hash table where each index has a vector of pairs (name, stude
nt ID)
vector<vector<pair<string, int>>> hashTable(TABLE_SIZE);

// Function to insert a name and student ID into the hash table
void insert(const string& name, int id) {
    int index = hashFunction(name);
    hashTable[index].push_back(make_pair(name, id));
}



// Function to display the hash table
void displayHashTable() {
    for (int i = 0; i < TABLE_SIZE; ++i) {
        cout << "Index " << i << ": ";
        for (auto& p : hashTable[i]) {
            cout << "[" << p.first << ", " << p.second << "] ";
        }
        cout << endl;
    }
}

int main() {
    // Inserting some student names and IDs
    insert("Alice", 12345);
    insert("Bob", 23456);
    insert("Charlie", 34567);
    insert("David", 67890);

    // Display the hash table
    displayHashTable();

    return 0;
}
```

Explanation:

1. **Headers and Namespaces:**
   - We include `<iostream>` for console input/output, `<vector>` for using vectors, and `<string>` for using strings.
   - `using namespace std;` allows for easier usage of standard library types.
2. **Hash Function:**
   - `hashFunction` computes an index by summing the ASCII values of the characters in a name and taking the modulus with `TABLE_SIZE`.
3. **Global Hash Table:**
   - `hashTable` is a vector of vectors of pairs. Each vector entry at an index can hold multiple pairs of names and IDs to handle collisions.
4. **Insert Function:**
   - `insert` adds a name and ID to the hash table. It calculates the index using the hash function and pushes the name and ID pair to the appropriate vector based on the computed index.
5. **Display Function:**
   - `displayHashTable` iterates through each index of the hash table and prints the contents, showing how names and IDs are stored, including any collisions.
6. **Main Function:**
   - The `main` function demonstrates inserting four names with their IDs and then displaying the hash table structure.

Populating the Hash Table

Consider these student names: Alice, Bob, Charlie, David.

Using our hash function:

- Alice -> (65 + 108 + 105 + 99 + 101) % 10 = 478 % 10 = 8
- Bob -> (66 + 111 + 98) % 10 = 275 % 10 = 5
- Charlie -> (67 + 104 + 97 + 114 + 108 + 105 + 101) % 10 = 696 % 10 = 6
- David -> (68 + 97 + 118 + 105 + 100) % 10 = 488 % 10 = 8 (Collision with Alice)

Handling Collision

Since both Alice and David hashed to the same index (8), we need to handle this collision. One common method is chaining, where we keep a list at each index of the hash table.

- Index 8 would contain [("Alice", 12345), ("David", 67890)]

Load Factor

The load factor is the number of entries divided by the number of slots in the hash table. In our case, if we have 4 students and 10 slots, the load factor is 4/10 = 0.4. A lower load factor means fewer collisions and generally faster lookup times.

Visual Representation of the Hash Table

Here's what our hash table looks like after adding the students:

- Index 0: []
- Index 1: []
- Index 2: []
- Index 3: []
- Index 4: []
- Index 5: [("Bob", 23456)]
- Index 6: [("Charlie", 34567)]
- Index 7: []
- Index 8: [("Alice", 12345), ("David", 67890)]
- Index 9: []

Summary

- **Hash Table:** An array of lists (to handle collisions via chaining).
- **Hash Function:** Sums ASCII values of characters in names and takes modulus 10.
- **Collisions:** Occur when two names hash to the same index; handled by chaining in this example.
- **Load Factor:** Number of entries divided by the number of slots, influencing the performance of the hash table.

Hash Functions

Definition

A hash function is a function that takes an input (or "key") and returns a fixed-size string of bytes. The output, typically a hash code or hash value, is usually a number in a specific range. Hash functions are used to map data of arbitrary size to data of fixed size.

Mathematical Definition: $h(x) \rightarrow y$ Where:

- h is the hash function.
- x is the input (key).
- y is the output (hash value).

Properties

1. **Deterministic**: For a given input, the hash function must always produce the same output.
2. **Fixed Output Size**: The output (hash value) should have a fixed size, regardless of the size of the input.
3. **Efficiently Computable**: The hash function should be able to compute the hash value quickly.
4. **Uniform Distribution**: The hash values should be uniformly distributed to minimize the chances of collisions (two different inputs having the same hash value).

5. **Minimize Collisions**: While it's impossible to avoid collisions completely, a good hash function should minimize them.
6. **Avalanche Effect**: A small change in the input should produce a significantly different hash value.

Good Hash Function Characteristics

1. **Uniformity**:
    - The hash function should distribute hash values uniformly across the hash table.
    - This minimizes clustering and ensures that each hash table bucket is equally likely to be chosen.
2. **Efficiency**:
    - The hash function should be fast to compute.
    - It should handle a large number of inputs efficiently without significant time complexity.
3. **Deterministic**:
    - For any given input, the hash function should consistently return the same hash value.
    - This property ensures reliability and predictability in hash-based data structures.
4. **Low Collision Probability**:
    - The hash function should minimize the probability of collisions.
    - Collisions occur when two different inputs produce the same hash value.
5. **Simplicity**:
    - The hash function should be simple to implement and understand.
    - Complex hash functions might introduce unnecessary computational overhead.
6. **Avalanche Effect**:
    - A small change in the input should result in a significantly different hash value.
    - This ensures that hash values are well-distributed even for similar inputs.

Example of a Good Hash Function

Consider the following simple hash function for strings, known as the "djb2" hash function, created by Daniel J. Bernstein. It is widely used due to its simplicity and effectiveness.

Code Example in C++:

```
#include <iostream>
#include <string>


int main() {
    string key1 = "example";
    string key2 = "Example"; // Different case, different hash valu
e
    string key3 = "another_example";

    cout << "Hash of \"" << key1 << "\": " << djb2Hash(key1) << end
l;
    cout << "Hash of \"" << key2 << "\": " << djb2Hash(key2) << end
l;
    cout << "Hash of \"" << key3 << "\": " << djb2Hash(key3) << end
l;

    return 0;
}
```

Explanation of the Code

1. **Initialization**:
   - The hash value is initialized to 5381, a prime number. This helps in achieving a good distribution of hash values.
2. **Hash Calculation**:
   - For each character in the input string, the hash value is updated using the formula: $\text{hash} = ((\text{hash} \ll 5) + \text{hash}) + c$
   - This formula effectively multiplies the current hash value by 33 and adds the ASCII value of the current character. The bitwise left shift (`<< 5`) operation is equivalent to multiplying by 32, and adding the current hash value again makes it 33.
3. **Result**:
   - The function returns the computed hash value for the input string.

Characteristics of the djb2 Hash Function

1. **Uniformity**: The djb2 hash function provides a good distribution of hash values, reducing clustering and evenly spreading keys across the hash table.
2. **Efficiency**: The hash function is computationally efficient, making it suitable for a wide range of applications.
3. **Deterministic**: For any given input, the djb2 hash function consistently produces the same hash value.
4. **Low Collision Probability**: Although collisions can occur, the djb2 hash function minimizes their probability through its effective distribution.
5. **Simplicity**: The djb2 hash function is simple to understand and implement, with a straightforward calculation method.
6. **Avalanche Effect**: The djb2 hash function exhibits a strong avalanche effect, ensuring that even small changes in the input produce significantly different hash values.

Summary

A good hash function is crucial for the performance of hash-based data structures. It should be efficient, uniform, deterministic, simple, and exhibit a strong avalanche effect. The djb2 hash function is an example of a well-designed hash function that meets these criteria, making it widely used and respected in various applications. By understanding and implementing such hash functions, developers can optimize their data structures for better performance and reliability.

Hash Tables

Concept and Structure

Definition: A hash table is a data structure that maps keys to values. It uses a hash function to compute an index (or hash code) into an array of buckets or slots, from which the desired value can be found.

Structure:

- **Array**: The main component of a hash table is an array where data is stored.
- **Hash Function**: A function that takes a key and computes an index in the array.
- **Buckets/Slots**: Each position in the array, known as a bucket or slot, can hold the key-value pair or a reference to it.

Basic Operations:

1. **Insert**: Add a key-value pair to the hash table.
2. **Search**: Retrieve the value associated with a given key.
3. **Delete**: Remove a key-value pair from the hash table.

Collision Resolution Techniques

Collisions in hashing occur when two different keys hash to the same index in the hash table. This means that multiple elements are assigned to the same slot, despite their distinct keys. This situation necessitates a strategy to handle the overlap so that each element remains accessible.

Example

Imagine a simple hash function for a hash table with 10 slots that assigns an index based on the last digit of a person's ID number. If John's ID is 12345 and Sara's ID is 56755, both would be assigned to index 5 (because the last digit of both IDs is 5). This is a collision. To resolve this, one might use chaining, where each slot in the hash table holds a list of all entries hashing to that index. Thus, John and Sara would both be stored at index 5, but as distinct elements in a linked list.

1. Chaining

In chaining, each bucket of the hash table is a linked list. When a collision occurs, the new element is added to the list at the corresponding bucket.

Advantages:

- Simple to implement.
- Can handle an unlimited number of collisions.

Disadvantages:

- Requires additional memory for pointers.
- Can become slow if many elements are in the same bucket.

Example: Assume we have a hash table with 5 buckets and the hash function $ h(k) = k \mod 5 $.

- Insert keys 10, 15, 20, 25, 30.
- All keys hash to bucket 0: $ 10 \rightarrow 0, 15 \rightarrow 0, 20 \rightarrow 0, 25 \rightarrow 0, 30 \rightarrow 0 $.

Bucket 0: $ 10 \rightarrow 15 \rightarrow 20 \rightarrow 25 \rightarrow 30 $

Code Example for Chaining:

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



class HashTable {
    int BUCKET;
    vector<list<int>> table;

public:
    HashTable(int b) {
        BUCKET = b;
        table.resize(b);
    }

    int hashFunction(int x) {
        return x % BUCKET;
    }
```

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

    void deleteItem(int key) {
        int index = hashFunction(key);
        table[index].remove(key);
    }

    void displayHash() {
        for (int i = 0; i < BUCKET; i++) {
            cout << i;
            for (auto x : table[i])
                cout << " --> " << x;
            cout << endl;
        }
    }
};



int main() {
    int keys[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int n = sizeof(keys)/sizeof(keys[0]);

    HashTable ht(10);

    for (int i = 0; i < n; i++)
        ht.insertItem(keys[i]);

    ht.displayHash();

    ht.deleteItem(20);
    ht.displayHash();

    return 0;
}
```

2. Open Addressing

In open addressing, all elements are stored in the hash table itself. When a collision occurs, the algorithm searches for the next empty slot according to a probing sequence.

Probing Techniques:

1. **Linear Probing**:
    - In linear probing, the interval between probes is fixed, typically 1.
    - Formula: $h'(k, i) = (h(k) + i) \mod m$

- Example: If a collision occurs at index 0, the next index checked is 1, then 2, and so on.

2. **Quadratic Probing**:
   - In quadratic probing, the interval between probes increases quadratically.
   - Formula: $h'(k, i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$
   - Example: If a collision occurs at index 0, the next indices checked are 1, 4, 9, etc.
3. **Double Hashing**:
   - In double hashing, a second hash function is used to calculate the interval between probes.
   - Formula: $h'(k, i) = (h(k) + i \cdot h_2(k)) \mod m$
   - Example: If a collision occurs at index 0, the next index is calculated using a second hash function

Advantages:

- No need for additional memory for pointers.
- Better cache performance due to data locality.

Disadvantages:

- More complex than chaining.
- Clustering can occur, leading to performance degradation.

Code Example for Open Addressing (Linear Probing):

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



class HashTable {
    int BUCKET;
    vector<int> table;
    vector<bool> filled;

public:
    HashTable(int b) : BUCKET(b), table(b, -1), filled(b, false) {}

    int hashFunction(int x) {
        return x % BUCKET;
    }
```

```
    void insertItem(int key) {
        int index = hashFunction(key);
        int i = 0;
        while (filled[index]) {
            index = (index + 1) % BUCKET;
            i++;
            if (i == BUCKET) {
                cout << "Hash table is full\n";
                return;
            }
        }
        table[index] = key;
        filled[index] = true;
    }



    void deleteItem(int key) {
        int index = hashFunction(key);
        int i = 0;
        while (filled[index]) {
            if (table[index] == key) {
                table[index] = -1;
                filled[index] = false;
                return;
            }
            index = (index + 1) % BUCKET;
            i++;
            if (i == BUCKET) {
                cout << "Key not found\n";
                return;
            }
        }
        cout << "Key not found\n";
    }



    void displayHash() {
        for (int i = 0; i < BUCKET; i++) {
            if (filled[i])
                cout << i << " --> " << table[i] << endl;
            else
                cout << i << " --> " << "empty" << endl;
        }
    }
};
```

```
int main() {
    int keys[] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int n = sizeof(keys)/sizeof(keys[0]);

    HashTable ht(10);

    for (int i = 0; i < n; i++)
        ht.insertItem(keys[i]);

    ht.displayHash();

    ht.deleteItem(20);
    ht.displayHash();

    return 0;
}
```

Summary

Chaining:

- Uses linked lists to handle collisions.
- Each bucket contains a list of all elements that hash to the same index.
- Simple to implement but requires extra memory for pointers.

Open Addressing:

- All elements are stored in the hash table itself.
- Uses probing sequences to find the next available slot.
- Methods include linear probing, quadratic probing, and double hashing.
- No need for additional memory but can suffer from clustering.

Both chaining and open addressing are effective methods for resolving collisions in hash tables. The choice of method depends on the specific requirements of the application, such as memory constraints and the expected load factor of the hash table.

Implementation of Hash Table in C++

Hash tables are a critical data structure that provides efficient lookup, insertion, and deletion operations. In C++, a hash table can be implemented using arrays and hash functions, and collision resolution techniques can be applied to handle scenarios where multiple keys hash to the same index.

Key Concepts

1. **Hash Function**: Converts a key into an index in the hash table.
2. **Bucket/Slot**: Each index in the hash table array is a bucket that can hold a key-value pair.
3. **Collision**: Occurs when two keys hash to the same index.

4. **Collision Resolution**: Techniques to handle collisions, such as chaining and open

Example: Implementing a Hash Table with Chaining

We'll implement a hash table using chaining, where each bucket is a linked list that stores all elements hashing to the same index.

HashTable.h

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



class HashTable {
private:
    vector<list<pair<int, string>>> table;
    int BUCKET;

    int hashFunction(int key) {
        return key % BUCKET;
    }

public:
    HashTable(int b);
    void insertItem(int key, const string& value);
    void deleteItem(int key);
    string searchItem(int key);
    void displayHash();
};
```

HashTable.cpp

```
#include "HashTable.h"

HashTable::HashTable(int b) {
    BUCKET = b;
    table.resize(BUCKET);
}

void HashTable::insertItem(int key, const string& value) {
    int index = hashFunction(key);
    table[index].push_back(make_pair(key, value));
}
```

```
void HashTable::deleteItem(int key) {
    int index = hashFunction(key);
    auto& cell = table[index];
    auto it = cell.begin();
    bool keyFound = false;
    for (; it != cell.end(); it++) {
        if (it->first == key) {
            keyFound = true;
            break;
        }
    }
    if (keyFound) {
        cell.erase(it);
        cout << "Key " << key << " deleted." << endl;
    } else {
        cout << "Key " << key << " not found." << endl;
    }
}



string HashTable::searchItem(int key) {
    int index = hashFunction(key);
    auto& cell = table[index];
    for (auto it = cell.begin(); it != cell.end(); it++) {
        if (it->first == key) {
            return it->second;
        }
    }
    return "Key not found.";
}



void HashTable::displayHash() {
    for (int i = 0; i < BUCKET; i++) {
        cout << "Bucket " << i << ": ";
        for (auto& kv : table[i]) {
            cout << "[" << kv.first << ": " << kv.second << "] ";
        }
        cout << endl;
    }
}
```

Main.cpp

```
#include "HashTable.h"

int main() {
    int BUCKET = 10;
    HashTable ht(BUCKET);

    ht.insertItem(10, "Value10");
    ht.insertItem(20, "Value20");
    ht.insertItem(30, "Value30");
    ht.insertItem(40, "Value40");

    cout << "Displaying hash table:" << endl;
    ht.displayHash();

    cout << "\nSearching for key 20: " << ht.searchItem(20) << end
l;
    cout << "Searching for key 99: " << ht.searchItem(99) << endl;

    ht.deleteItem(20);
    ht.deleteItem(99);

    cout << "\nDisplaying hash table after deletions:" << endl;
    ht.displayHash();

    return 0;
}
```

Explanation

HashTable Class:

- **Private Members**:
  - `vector<list<pair<int, string>>> table` : A vector of lists to store key-value pairs, where each list handles collisions via chaining.
  - `int BUCKET` : The number of buckets in the hash table.
  - `int hashFunction(int key)` : A simple hash function that computes the index for a given key using the modulo operation.
- **Public Methods**:
  - `HashTable(int b)` : Constructor that initializes the hash table with `b` buckets.
  - `void insertItem(int key, const string& value)` : Inserts a key-value pair into the hash table.
  - `void deleteItem(int key)` : Deletes a key-value pair from the hash table.
  - `string searchItem(int key)` : Searches for a key in the hash table and returns the associated value.
  - `void displayHash()` : Displays the contents of the hash table.

Main Function:

- **Initialization**:
  - Creates a hash table with 10 buckets.

- Inserts several key-value pairs into the hash table.
- Displays the hash table.
- Searches for keys 20 and 99.
- Deletes keys 20 and 99.
- Displays the hash table again after deletions.

Example: Implementing a Hash Table with Open Addressing (Linear Probing)

We'll also implement a hash table using open addressing with linear probing to resolve collisions.

HashTableOpenAddressing.h

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

class HashTableOpenAddressing {
private:
    vector<pair<int, string>> table;
    vector<bool> filled;
    int BUCKET;

    int hashFunction(int key) {
        return key % BUCKET;
    }



public:
    HashTableOpenAddressing(int b);
    void insertItem(int key, const string& value);
    void deleteItem(int key);
    string searchItem(int key);
    void displayHash();
};
```

HashTableOpenAddressing.cpp

```cpp
#include "HashTableOpenAddressing.h"

HashTableOpenAddressing::HashTableOpenAddressing(int b) : BUCKET
(b), table(b, make_pair(-1, "")), filled(b, false) {}

void HashTableOpenAddressing::insertItem(int key, const string& val
ue) {
    int index = hashFunction(key);
    int i = 0;
    while (filled[index]) {


void HashTableOpenAddressing::deleteItem(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (filled[index]) {
        if (table[index].first == key) {
            table[index] = make_pair(-1, "");
            filled[index] = false;
            cout << "Key " << key << " deleted." << endl;
            return;
        }
        index = (index + 1) % BUCKET;
        i++;
        if (i == BUCKET) {
            cout << "Key not found\n";
            return;
        }
    }
    cout << "Key not found\n";
}



string HashTableOpenAddressing::searchItem(int key) {
    int index = hashFunction(key);
    int i = 0;
    while (filled[index]) {
        if (table[index].first == key) {
            return table[index].second;
        }
        index = (index + 1) % BUCKET;
        i++;
        if (i == BUCKET) {
            break;
        }
    }
    return "Key not found.";
}
```

```
void HashTableOpenAddressing::displayHash() {
    for (int i = 0; i < BUCKET; i++) {
        if (filled[i])
            cout << i << " --> [" << table[i].first << ": " << tabl
e[i].second << "]" << endl;
        else
            cout << i << " --> empty" << endl;
    }
}
```

MainOpenAddressing.cpp

```
#include "HashTableOpenAddressing.h"

int main() {
    int BUCKET = 10;
    HashTableOpenAddressing ht(BUCKET);

    ht.insertItem(10, "Value10");
    ht.insertItem(20, "Value20");
    ht.insertItem(30, "Value30");
    ht.insertItem(40, "Value40");
    ht.insertItem(50, "Value50");
    ht.insertItem(15, "Value15");
    ht.insertItem(25, "Value25");



    cout << "Displaying hash table:" << endl;
    ht.displayHash();

    cout << "\nSearching for key 20: " << ht.searchItem(20) << end
l;
    cout << "Searching for key 99: " << ht.searchItem(99) << endl;

    ht.deleteItem(20);
    ht.deleteItem(99);

    cout << "\nDisplaying hash table after deletions:" << endl;
    ht.displayHash();

    return 0;
}
```

Explanation

HashTableOpenAddressing Class:

- **Private Members**:

    - `vector<pair<int, string>> table` : A vector to store key-value pairs.
    - `vector<bool> filled` : A vector to keep track of filled slots.
    - `int BUCKET` : The number of buckets in the hash table.
    - `int hashFunction(int key)` : A simple hash function that computes the index for a given key using the modulo operation.
- **Public Methods**:
    - `HashTableOpenAddressing(int b)` : Constructor that initializes the hash table with `b` buckets.
    - `void insertItem(int key, const string& value)` : Inserts a key-value pair into the hash table using linear probing.
    - `void deleteItem(int key)` : Deletes a key-value pair from the hash table using linear probing.
    - `string searchItem(int key)` : Searches for a key in the hash table and returns the associated value using linear probing.
    - `void displayHash()` : Displays the contents of the hash table.

Main Function:

- **Initialization**:
    - Creates a hash table with 10 buckets.
    - Inserts several key-value pairs into the hash table.
    - Displays the hash table.
    - Searches for keys 20 and 99.
    - Deletes keys 20 and 99.
    - Displays the hash table again after deletions.

Summary

This detailed example covers the implementation of a hash table in C++ using both chaining and open addressing (linear probing) collision resolution techniques. By understanding these implementations, one can leverage the power of hash tables to perform efficient lookups, insertions, and deletions in various applications.

Applications of Hashing

Hashing is a powerful technique widely used in computer science for efficient data retrieval and storage. Some common applications of hashing include dictionaries, caches, and sets. Here's an in-depth look at each application, along with examples and code snippets.

1. Dictionaries

Concept:

- A dictionary (or hashmap) is a data structure that stores key-value pairs.
- Hashing is used to map keys to their corresponding values, allowing for efficient data retrieval.

Use Case:

- Storing and retrieving information quickly, such as a phonebook, where you can look up a person's phone number by their name.

Example: Consider a dictionary that maps employee IDs to their names.

Code Example in C++:

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

int main() {
    unordered_map<int, string> employeeDirectory;

    // Inserting key-value pairs
    employeeDirectory[101] = "Alice";
    employeeDirectory[102] = "Bob";
    employeeDirectory[103] = "Charlie";



    // Retrieving values using keys
    cout << "Employee 101: " << employeeDirectory[101] << endl;
    cout << "Employee 102: " << employeeDirectory[102] << endl;
    cout << "Employee 103: " << employeeDirectory[103] << endl;

    // Checking if a key exists
    int key = 104;
    if (employeeDirectory.find(key) != employeeDirectory.end()) {
        cout << "Employee " << key << ": " << employeeDirectory[ke
y] << endl;
    } else {
        cout << "Employee " << key << " not found." << endl;
    }



    // Deleting a key-value pair
    employeeDirectory.erase(102);

    // Displaying the dictionary
    cout << "Employee Directory:" << endl;
    for (auto& pair : employeeDirectory) {
        cout << "ID: " << pair.first << ", Name: " << pair.second <
< endl;
    }

    return 0;
}
```

2. Caches

Concept:

- A cache is a temporary storage area that holds frequently accessed data to reduce access time.
- Hashing is used in caches to quickly map requests to stored data.

Use Case:

- Web browsers use caching to store web pages, images, and other resources for faster access on subsequent requests.

Code Example in C++:

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

// Function to simulate an expensive computation
int expensiveComputation(int x) {
    return x * x; // Just a placeholder for a real expensive comput
ation
}

class Cache {
private:
    unordered_map<int, int> cache;



public:
    int get(int key) {
        if (cache.find(key) != cache.end()) {
            return cache[key];
        } else {
            int result = expensiveComputation(key);
            cache[key] = result;
            return result;
        }
    }

    void displayCache() {
        cout << "Cache contents:" << endl;
        for (auto& pair : cache) {
            cout << "Key: " << pair.first << ", Value: " << pair.se
cond << endl;
        }
    }
};
```

```
int main() {
    Cache myCache;

    // Performing computations and caching results
    cout << "Result for 5: " << myCache.get(5) << endl;
    cout << "Result for 10: " << myCache.get(10) << endl;
    cout << "Result for 5 (cached): " << myCache.get(5) << endl;

    // Displaying cache contents
    myCache.displayCache();

    return 0;
}
```

3. Sets

Concept:

- A set is a collection of unique elements.
- Hashing is used to quickly check for the existence of elements, ensuring that duplicates are not added.

Use Case:

- Tracking unique visitors to a website.

Example: Consider a set that stores unique visitor IDs.

Code Example in C++:

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

```
int main() {
    unordered_set<int> visitorIDs;

    // Adding elements to the set
    visitorIDs.insert(101);
    visitorIDs.insert(102);
    visitorIDs.insert(103);
    visitorIDs.insert(101); // Duplicate element, will not be added

    // Checking for the existence of an element
    int visitorID = 102;
    if (visitorIDs.find(visitorID) != visitorIDs.end()) {
        cout << "Visitor " << visitorID << " exists." << endl;
    } else {
        cout << "Visitor " << visitorID << " does not exist." << en
dl;
    }



    // Deleting an element from the set
    visitorIDs.erase(102);

    // Displaying the set
    cout << "Visitor IDs:" << endl;
    for (auto id : visitorIDs) {
        cout << id << " ";
    }
    cout << endl;

    return 0;
}
```

Summary

Dictionaries:

- **Purpose**: Store key-value pairs.
- **Example**: Employee directory.
- **Operations**: Insert, retrieve, check existence, delete.

Caches:

- **Purpose**: Store frequently accessed data temporarily.
- **Example**: Storing results of expensive computations.
- **Operations**: Retrieve (with caching logic), display.

Sets:

- **Purpose**: Store unique elements.
- **Example**: Tracking unique visitor IDs.

- **Operations**: Insert, check existence, delete, display.

Hashing plays a crucial role in these applications by providing efficient mechanisms for data retrieval and storage. Understanding and implementing these applications can significantly enhance the performance of systems and software solutions.

Rehashing

Concept and Necessity

Concept: Rehashing is the process of resizing a hash table and re-inserting all its elements into the new table. This involves creating a larger hash table, recalculating the hash codes for each element, and inserting them into the new table. The main goal of rehashing is to maintain efficient performance by reducing the load factor, which is the ratio of the number of elements to the number of buckets in the hash table.

Necessity:

1. **Load Factor**:
    - A high load factor increases the chances of collisions, leading to longer chains in chaining or more probes in open addressing, which degrades performance.
    - A common practice is to keep the load factor below a certain threshold (e.g., 0.75).

2. **Performance**:
    - As the number of elements grows, the time complexity for operations like insert, search, and delete increases if the load factor is too high.
    - Rehashing ensures that these operations remain efficient by distributing the elements more evenly across the new table.
3. **Capacity**:
    - Initially, a hash table may be created with a small number of buckets to save space.
    - As more elements are added, the table needs to be resized to accommodate the growing number of elements without sacrificing performance.

Implementation in C++

Let's implement a hash table with chaining and include rehashing. We'll start with a basic hash table implementation and then add rehashing functionality.

HashTable.h

```
#include <iostream>
#include <list>


    int hashFunction(int key) {
        return key % BUCKET;
    }

    void rehash();

public:
    HashTable(int b);
    void insertItem(int key, const string& value);
    void deleteItem(int key);
    string searchItem(int key);
    void displayHash();
};
```

HashTable.cpp

```
#include "HashTable.h"

HashTable::HashTable(int b) {
    BUCKET = b;
    size = 0;
    table.resize(BUCKET);
}

void HashTable::insertItem(int key, const string& value) {
    int index = hashFunction(key);
    table[index].push_back(make_pair(key, value));
    size++;

    // Check load factor and rehash if necessary
    if (size > BUCKET * 0.75) {
        rehash();
    }
}
```

```
void HashTable::deleteItem(int key) {
    int index = hashFunction(key);
    auto& cell = table[index];
    auto it = cell.begin();
    bool keyFound = false;
    for (; it != cell.end(); it++) {
        if (it->first == key) {
            keyFound = true;
            break;
        }
    }
    if (keyFound) {
        cell.erase(it);
        size--;
        cout << "Key " << key << " deleted." << endl;
    } else {
        cout << "Key " << key << " not found." << endl;
    }
}



string HashTable::searchItem(int key) {
    int index = hashFunction(key);
    auto& cell = table[index];
    for (auto it = cell.begin(); it != cell.end(); it++) {
        if (it->first == key) {
            return it->second;
        }
    }
    return "Key not found.";
}



void HashTable::displayHash() {
    for (int i = 0; i < BUCKET; i++) {
        cout << "Bucket " << i << ": ";
        for (auto& kv : table[i]) {
            cout << "[" << kv.first << ": " << kv.second << "] ";
        }
        cout << endl;
    }
}

void HashTable::rehash() {
    cout << "Rehashing..." << endl;
```

```
    // Save old table
    vector<list<pair<int, string>>> oldTable = table;

    // Double the number of buckets
    BUCKET = 2 * BUCKET;
    table.clear();
    table.resize(BUCKET);
    size = 0;

    // Reinsert all elements into the new table
    for (auto& cell : oldTable) {
        for (auto& kv : cell) {
            insertItem(kv.first, kv.second);
        }
    }
}
```

Main.cpp

```
#include "HashTable.h"

int main() {
    int BUCKET = 5;
    HashTable ht(BUCKET);

    ht.insertItem(1, "Value1");
    ht.insertItem(2, "Value2");
    ht.insertItem(3, "Value3");
    ht.insertItem(4, "Value4");
    ht.insertItem(5, "Value5");
    ht.insertItem(6, "Value6");
    ht.insertItem(7, "Value7");
```

```
    cout << "Displaying hash table:" << endl;
    ht.displayHash();

    cout << "\nSearching for key 2: " << ht.searchItem(2) << endl;
    cout << "Searching for key 8: " << ht.searchItem(8) << endl;

    ht.deleteItem(2);
    ht.deleteItem(8);

    cout << "\nDisplaying hash table after deletions:" << endl;
    ht.displayHash();

    return 0;
}
```

Explanation

HashTable Class:

- **Private Members**:
  - `vector<list<pair<int, string>>> table` : A vector of lists to store key-value pairs, where each list handles collisions via chaining.
  - `int BUCKET` : The number of buckets in the hash table.
  - `int size` : The number of elements in the hash table.
  - `int hashFunction(int key)` : A simple hash function that computes the index for a given key using the modulo operation.
  - `void rehash()` : Resizes the hash table and re-inserts all elements.

- **Public Methods**:
  - `HashTable(int b)` : Constructor that initializes the hash table with `b` buckets.
  - `void insertItem(int key, const string& value)` : Inserts a key-value pair into the hash table. It checks the load factor and calls `rehash()` if necessary.
  - `void deleteItem(int key)` : Deletes a key-value pair from the hash table.
  - `string searchItem(int key)` : Searches for a key in the hash table and returns the associated value.
  - `void displayHash()` : Displays the contents of the hash table.

Main Function:

- **Initialization**:
  - Creates a hash table with 5 buckets.
  - Inserts several key-value pairs into the hash table.
  - Displays the hash table.
  - Searches for keys 2 and 8.
  - Deletes keys 2 and 8.
  - Displays the hash table again after deletions.

Rehashing in Detail

1. **Triggering Rehashing**:
    - Rehashing is triggered when the load factor exceeds a certain threshold (e.g., 0.75). This threshold ensures that the hash table remains efficient by preventing too many collisions.
2. **Rehashing Process**:
    - **Save Old Table**: The current table is saved.
    - **Resize Table**: The number of buckets is typically doubled to reduce the load factor.
    - **Reinsert Elements**: All elements from the old table are reinserted into the new table. This step involves recomputing the hash codes for each element since the number of buckets has changed.
3. **Benefits of Rehashing**:
    - **Improved Performance**: By reducing the load factor, rehashing decreases the likelihood of collisions, thereby improving the performance of insertion, deletion, and search operations.
    - **Scalability**: Rehashing allows the hash table to grow dynamically, accommodating an increasing number of elements without sacrificing efficiency.

Summary

Rehashing is an essential technique to maintain the efficiency of hash tables as the number of elements grows. It involves resizing the hash table and re-inserting all elements, ensuring that the load factor remains within an optimal range. By implementing rehashing, hash tables can provide consistent and efficient performance for a wide range of applications. The provided code examples demonstrate how to implement a hash table with chaining and rehashing in C++.

Comparison of Hashing with Other Data Structures

Hashing is a powerful technique used in hash tables to provide efficient data storage and retrieval. However, it is essential to understand how hashing compares to other data structures to make informed decisions about which structure to use based on specific requirements. Below, we compare hash tables with arrays, linked lists, binary search trees (BST), and balanced trees (AVL and Red-Black Trees) based on various criteria.

Comparison Criteria

1. **Time Complexity**:
    - Insertion
    - Deletion
    - Search
2. **Space Complexity**
3. **Order Preservation**
4. **Implementation Complexity**
5. **Use Cases**

Comparison Table

Data Structure	Insertion	Deletion	Search	Space Complexity	Order Preservation	Implementation Complexity	Use Ca
Hash Table	$O(1)$	$O(1)$	$O(1)$	$O(n)$	No	Medium	Dictiona caches,
Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$	Yes	Low	Fixed-size c frequent inde
Linked List	$O(1)$	$O(1)$	$O(n)$	$O(n)$	Yes	Low	Dynamic c freq insertions/delet
BST	$O(n)$	$O(n)$	$O(n)$	$O(n)$	Yes	Medium	Hierarchical c ordered
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Yes	High	Self-balan ordered
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Yes	High	Self-balan ordered

Detailed Comparison

1. Hash Table

Operations:

- **Insertion**: $O(1)$ average, due to direct access using hash function.
- **Deletion**: $O(1)$ average, using hash function to locate the element.
- **Search**: $O(1)$ average, due to direct access using hash function.

Space Complexity:

- $O(n)$, where n is the number of elements. Additional space is required for storing pointers in chaining.

Order Preservation:

- No. Hash tables do not maintain any order of elements.

Implementation Complexity:

- Medium. Requires a good hash function and handling of collisions (chaining or open addressing).

Use Cases:

- Dictionaries, caches, sets, and scenarios requiring fast lookups.

2. Array

Operations:

- **Insertion**: $O(n)$ in the worst case, as elements may need to be shifted.

- **Deletion**: $O(n)$ in the worst case, due to shifting elements.
- **Search**: $O(n)$ in the worst case, for unsorted arrays. $O(\log n)$ for sorted arrays with binary search.

Space Complexity:

- $O(n)$, where n is the number of elements.

Order Preservation:

- Yes. Arrays maintain the order of elements.

Implementation Complexity:

- Low. Arrays are simple to implement.

Use Cases:

- Fixed-size data, frequent indexing, and scenarios where the order of elements is crucial.

3. Linked List

Operations:

- **Insertion**: $O(1)$ at the beginning, $O(n)$ if inserting at a specific position.
- **Deletion**: $O(1)$ if deleting the first element, $O(n)$ if deleting a specific element.
- **Search**: $O(n)$, as it requires traversal from the head to the desired element.

Space Complexity:

- $O(n)$, where n is the number of elements. Additional space for pointers.

Order Preservation:

- Yes. Linked lists maintain the order of elements.

Implementation Complexity:

- Low. Linked lists are straightforward to implement.

Use Cases:

- Dynamic data, frequent insertions and deletions, and scenarios where the order of elements is crucial.

4. Binary Search Tree (BST)

Operations:

- **Insertion**: $O(n)$ in the worst case, if the tree becomes unbalanced.
- **Deletion**: $O(n)$ in the worst case, for the same reason.
- **Search**: $O(n)$ in the worst case, for unbalanced trees.

Space Complexity:

- $O(n)$, where n is the number of elements.

Order Preservation:

- Yes. BSTs maintain the order of elements.

Implementation Complexity:

- Medium. Requires careful implementation to maintain the BST properties.

Use Cases:

- Hierarchical data, ordered data, and scenarios where ordered traversal is required.

5. AVL Tree

Operations:

- **Insertion**: $O(\log n)$, as the tree is self-balancing.
- **Deletion**: $O(\log n)$, due to self-balancing.
- **Search**: $O(\log n)$, as the tree is balanced.

Space Complexity:

- $O(n)$, where n is the number of elements.

Order Preservation:

- Yes. AVL trees maintain the order of elements.

Implementation Complexity:

- High. Requires additional logic to maintain balance after insertions and deletions.

Use Cases:

- Self-balancing scenarios, ordered data, and applications needing efficient search, insert, and delete operations.

6. Red-Black Tree

Operations:

- **Insertion**: $O(\log n)$, due to self-balancing properties.
- **Deletion**: $O(\log n)$, for the same reason.
- **Search**: $O(\log n)$, as the tree remains balanced.

Space Complexity:

- $O(n)$, where n is the number of elements.

Order Preservation:

- Yes. Red-Black trees maintain the order of elements.

Implementation Complexity:

- High. Requires additional rules and balancing logic.

Use Cases:

- Self-balancing scenarios, ordered data, databases, and scenarios requiring efficient and reliable operations.

Summary

Hash tables provide constant time complexity on average for insertion, deletion, and search operations, making them highly efficient for scenarios requiring fast lookups and updates. However, they do not maintain order and require careful handling of collisions. Arrays and linked lists offer simplicity and order preservation but can have higher time complexity for operations. Binary search trees and their balanced variants (AVL and Red-Black trees) maintain order and provide efficient operations, albeit with higher implementation complexity.

Choosing the appropriate data structure depends on the specific requirements of the application, such as the need for order preservation, performance constraints, and the nature of the operations performed frequently. Understanding these trade-offs helps in making informed decisions to optimize performance and resource utilization.

MCQs

Hash Functions

1. **What is a hash function?**
    - A. A function that maps data of fixed size to data of arbitrary size
    - B. A function that maps data of arbitrary size to data of fixed size
    - C. A function that sorts data in ascending order
    - D. A function that finds the shortest path in a graph
    - **Correct Answer: B. A function that maps data of arbitrary size to data of fixed size**
2. **Which of the following is a characteristic of a good hash function?**
    - A. High collision probability
    - B. Deterministic
    - C. Slow computation
    - D. Fixed output size for variable input size
    - **Correct Answer: B. Deterministic**
3. **What is the avalanche effect in hash functions?**
    - A. Small changes in the input produce significantly different hash values.
    - B. Large changes in the input produce similar hash values.
    - C. Small changes in the input produce similar hash values.
    - D. Large changes in the input produce significantly different hash values.
    - **Correct Answer: A. Small changes in the input produce significantly different hash values**

Hash Tables

4. **What is the primary purpose of a hash table?**

- A. To sort data in ascending order
- B. To provide efficient data storage and retrieval
- C. To detect cycles in a graph
- D. To find the shortest path in a graph
- **Correct Answer: B. To provide efficient data storage and retrieval**

5. **What is a collision in a hash table?**

   - A. When two keys hash to different indices
   - B. When two keys hash to the same index
   - C. When a key cannot be hashed
   - D. When a hash table is full
   - **Correct Answer: B. When two keys hash to the same index**

6. **Which of the following is a collision resolution technique that uses linked lists at each bucket?**

   - A. Open Addressing
   - B. Chaining
   - C. Linear Probing
   - D. Quadratic Probing
   - **Correct Answer: B. Chaining**

Open Addressing

7. **Which probing technique in open addressing uses a fixed interval between probes?**

   - A. Linear Probing
   - B. Quadratic Probing
   - C. Double Hashing
   - D. Chaining
   - **Correct Answer: A. Linear Probing**

8. **In double hashing, what determines the interval between probes?**

   - A. The primary hash function
   - B. A second hash function
   - C. The size of the hash table
   - D. The load factor
   - **Correct Answer: B. A second hash function**

9. **Which of the following is an advantage of open addressing over chaining in hash tables?**

   - A. No need for additional memory for pointers
   - B. Better performance for large datasets
   - C. Simpler implementation
   - D. Better handling of high load factors
   - **Correct Answer: A. No need for additional memory for pointers**

Rehashing

10. **What is rehashing in the context of hash tables?**

    - A. The process of finding the hash value of a key

- B. The process of resizing the hash table and re-inserting all elements
- C. The process of handling collisions using chaining
- D. The process of sorting the elements in the hash table
- **Correct Answer: B. The process of resizing the hash table and re-inserting all elements**

11. **When is rehashing typically triggered in a hash table?**

    - A. When the hash

    function fails

    - B. When the load factor exceeds a certain threshold
    - C. When a collision occurs
    - D. When the hash table is empty
    - **Correct Answer: B. When the load factor exceeds a certain threshold**

12. **What is the main benefit of rehashing?**

    - A. Reduced memory usage
    - B. Improved performance by reducing collisions
    - C. Simplified hash function
    - D. Increased load factor
    - **Correct Answer: B. Improved performance by reducing collisions**

Comparison of Hashing with Other Data Structures

13. **Which data structure provides average O(1) time complexity for insertion, deletion, and search operations?**

    - A. Array
    - B. Linked List
    - C. Binary Search Tree (BST)
    - D. Hash Table
    - **Correct Answer: D. Hash Table**

14. **Which data structure maintains elements in a sorted order?**

    - A. Hash Table
    - B. Array
    - C. Linked List
    - D. AVL Tree
    - **Correct Answer: D. AVL Tree**

15. **What is the time complexity for searching an element in a balanced binary search tree (AVL Tree)?**

    - A. O(1)
    - B. O(log n)
    - C. O(n)
    - D. O(n log n)
    - **Correct Answer: B. O(log n)**

Hash Functions Examples

16. **What property of a hash function ensures that small changes in the input produce significantly different hash values?**

- A. Deterministic
- B. Uniform distribution
- C. Avalanche effect
- D. Fixed output size
- **Correct Answer: C. Avalanche effect**

17. **Which characteristic of a good hash function minimizes the probability of collisions?**

    - A. High computation time
    - B. Uniform distribution
    - C. Fixed output size
    - D. Deterministic
    - **Correct Answer: B. Uniform distribution**

18. **What does it mean for a hash function to be deterministic?**

    - A. It produces different outputs for the same input each time.
    - B. It produces the same output for the same input each time.
    - C. It produces random outputs for different inputs.
    - D. It is based on a cryptographic algorithm.
    - **Correct Answer: B. It produces the same output for the same input each time**

Hash Tables Examples

19. **What is the primary advantage of using a hash table over an array?**

    - A. Hash tables use less memory.
    - B. Hash tables provide O(1) average time complexity for insertions, deletions, and searches.
    - C. Hash tables maintain the order of elements.
    - D. Hash tables do not need collision resolution.
    - **Correct Answer: B. Hash tables provide O(1) average time complexity for insertions, deletions, and searches**

20. **Which collision resolution technique uses an array of linked lists to handle collisions?**

    - A. Open Addressing
    - B. Linear Probing
    - C. Chaining
    - D. Quadratic Probing
    - **Correct Answer: C. Chaining**

21. **Which collision resolution technique involves finding another slot within the array when a collision occurs?**

    - A. Open Addressing
    - B. Chaining
    - C. Separate Chaining
    - D. Double Hashing
    - **Correct Answer: A. Open Addressing**

Open Addressing Examples

22. **Which probing technique in open addressing uses a quadratic function to find the next available slot?**

    - A. Linear Probing
    - B. Quadratic Probing
    - C. Double Hashing
    - D. Chaining
    - **Correct Answer: B. Quadratic Probing**
23. **What is a disadvantage of using linear probing for collision resolution?**

    - A. It requires additional memory for pointers.
    - B. It can lead to clustering of entries.
    - C. It is complex to implement.
    - D. It does not guarantee finding an empty slot.
    - **Correct Answer: B. It can lead to clustering of entries**
24. **Which of the following is true about double hashing?**

    - A. It uses two hash functions to calculate the interval between probes.
    - B. It uses a fixed interval between probes.
    - C. It resolves collisions using linked lists.
    - D. It does not require a secondary hash function.
    - **Correct Answer: A. It uses two hash functions to calculate the interval between probes**

Rehashing Examples

25. **What is the purpose of rehashing in a hash table?**

    - A. To change the hash function
    - B. To resize the hash table and re-insert all elements
    - C. To handle clustering
    - D. To sort the elements in the hash table
    - **Correct Answer: B. To resize the hash table and re-insert all elements**
26. **What typically triggers rehashing in a hash table?**

    - A. High collision rate
    - B. Low load factor
    - C. High load factor
    - D. Change in hash function
    - **Correct Answer: C. High load factor**
27. **Which of the following is a benefit of rehashing?**

    - A. Increased load factor
    - B. Improved performance by reducing collisions
    - C. Simplified implementation
    - D. Reduced memory usage
    - **Correct Answer: B. Improved performance by reducing collisions**

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind