Lap (4)

Digital IC Design

Intake 45

Data Structure and Algorithms

Student Name: Fatma Saad Abdallah

Submitted to Eng. Mina Nagy

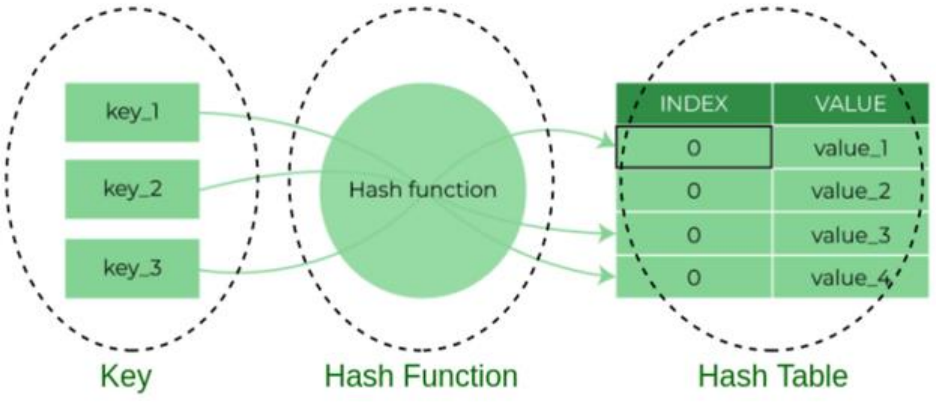
Hashing

Introduction:

Hashing is a technique that maps a large set of data to a small set of data. It uses a hash function for doing this mapping. It is an irreversible process and we cannot find the original value of the key from its hashed value because we are trying to map a large set of data into a small set of data, which may cause collisions. It is not uncommon to encounter collisions when mapping a large dataset into a smaller one. Suppose, We have three buckets and each bucket can store 1L of water in it and we have 5L of water also. We have to put all the water in these three buckets and this kind of situation is known as a collision. URL shorteners are an example of hashing as it maps large size URL to small size

Some Examples of Hash Functions:

- key % number of buckets
- ASCII value of character * PrimeNumber^x. Where x = 1, 2, 3….n
- You can make your own hash function but it should be a good hash function that gives less collisions.



Components of Hashing

Bucket Index:

The value returned by the Hash function is the bucket index for a key in a separate chaining method. Each index in the array is called a bucket as it is a bucket of a linked list.

Rehashing:

Rehashing is a concept that reduces collision when the elements are increased in the current hash table. It will make a new array of doubled size and copy the previous array elements to it and it is like the internal working of vector in C++. Obviously, the Hash function should be dynamic as it should reflect some changes when the capacity is increased. The hash function includes the capacity of the hash table in it, therefore, copying key values from the previous array hash function gives different bucket indexes as it is dependent on the capacity (buckets) of the hash table. Generally, When the value of the load factor is greater than 0.5 rehashing are done.

- Double the size of the array.
- Copy the elements of the previous array to the new array. We use the hash function while copying each node to a new array again therefore, It will reduce collision.
- Delete the previous array from the memory and point your hash map's inside array pointer to this new array.
- Generally, Load Factor = number of elements in Hash Map / total number of buckets (capacity).

Collision:

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.**

We can resolve the hash collision using one of the following techniques.

- Collision resolution by chaining
- Open Addressing: Linear/Quadratic Probing and Double Hashing

1. Collision resolution by chaining

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.

If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j contains NIL.

2. Open Addressing

Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left NIL.

Different techniques used in open addressing are:

i. Linear Probing

In linear probing, collision is resolved by checking the next slot.

$h(k, i) = (h'(k) + i) \bmod m$

where

- $i = \{0, 1, \ldots.\}$
- $h'(k)$ is a new hash function

If a collision occurs at $h(k, 0)$, then $h(k, 1)$ is checked. In this way, the value of i is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

ii. Quadratic Probing

It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

where,

- c_1 and c_2 are positive auxiliary constants,
- $i = \{0, 1, \ldots.\}$

iii. Double hashing

If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

h(k, i) = (h1(k) + ih2(k)) mod m

Good Hash Functions

A good hash function may not prevent the collisions completely however it can reduce the number of collisions.

Here, we will look into different methods to find a good hash function

1. Division Method

If k is a key and m is the size of the hash table, the hash function h() is calculated as:

h(k) = k mod m

For example, If the size of a hash table is 10 and k = 112 then h(k) = 112 mod 10 = 2. The value of m must not be the powers of 2. This is because the powers of 2 in binary format are 10, 100, 1000, …. When we find k mod m, we will always get the lower order p-bits.

2. Multiplication Method

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

where,

- kA mod 1 gives the fractional part kA,
- $\lfloor\ \rfloor$ gives the floor value
- A is any constant. The value of A lies between 0 and 1. But, an optimal choice will be ≈ (√5-1)/2 suggested by Knuth.

3. Universal Hashing

In Universal hashing, the hash function is chosen at random independent of keys.

Applications for Hash Table

Hash tables are implemented where

- constant time lookup and insertion is required
- cryptographic applications
- indexing data is required

<u>Graph</u>

Graphs are a fundamental data structure in computer science used to model relationships between objects. In C++, graphs can be represented and manipulated in various ways depending on the problem requirements. This report provides an overview of graph data structures, their types, and how they are implemented in C++.

What is a Graph?

A graph is a collection of:

- Nodes (or vertices): Represent entities, objects, or points.
- Edges: Represent the connections or relationships between nodes.

Types of Graphs

1. **Directed Graph (Digraph)**: Edges have a direction, going from one vertex to another.
2. **Undirected Graph**: Edges do not have a direction and connect two vertices bidirectionally.
3. **Weighted Graph**: Edges have associated weights, representing costs, distances, or other measures.
4. **Unweighted Graph**: Edges have no associated weights.
5. **Cyclic Graph:** Contains at least one cycle (a path that starts and ends at the same vertex).
6. **Acyclic Graph**: Does not contain any cycles.

Representing Graphs in C++

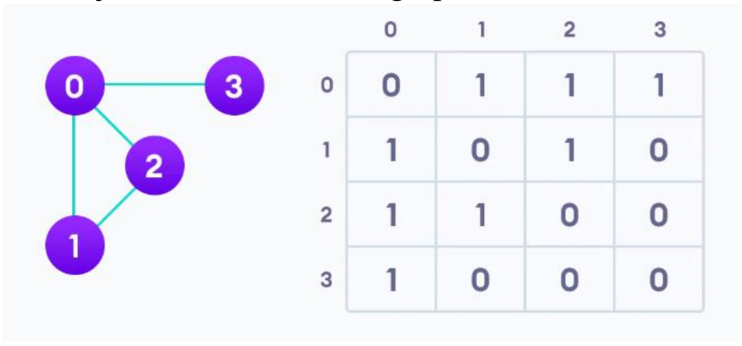
Graphs can be represented in C++ in the following ways:

1. Adjacency Matrix

An adjacent matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacent matrix for the graph we created above is



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

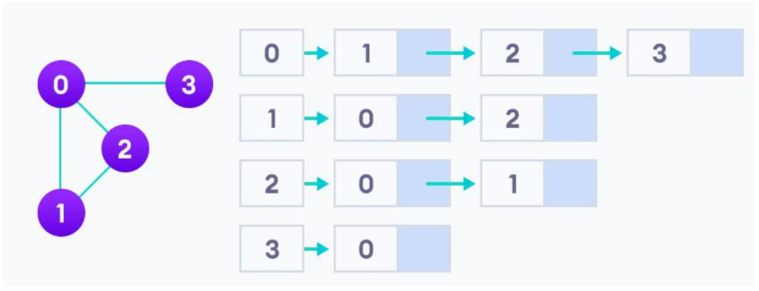
Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacent matrix representation but we have to reserve space for every possible link between all vertices (V x V), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

3. Edge List

A list of all edges in the graph, represented as pairs of vertices. This is useful for algorithms that process edges directly.

Traversing Graphs

Two common methods to traverse graphs are:

1. Depth First Search (DFS)

DFS explores as far along a branch as possible before backtracking.

2. Breadth First Search (BFS)

BFS explores all neighbors of a vertex before moving to the next level.

Graph Operations

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements (vertex, edges) to graph
- Finding the path from one vertex to another

Applications of Graphs

1. **Social Networks**: Representing connections between users.
2. **Navigation Systems**: Finding the shortest path.
3. **Web Crawling**: Representing links between web pages.
4. **Scheduling**: Representing dependencies in tasks.
5. **Networking**: Modeling computer networks.

Conclusion

Graphs are versatile data structures that can model a wide variety of real-world problems. In C++, graphs can be implemented efficiently using adjacency matrices, adjacency lists, or edge lists, depending on the application. Mastering graph traversal algorithms like DFS and BFS is essential for solving complex problems in fields such as networking, artificial intelligence, and logistics.