Hash Tables and Graphs Data Structures

1. Hash Tables

What Are Hash Tables?

Hash tables are a data structure that stores key-value pairs, allowing for efficient data retrieval. The key is processed through a **hash function**, which generates an index that indicates where the corresponding value is stored in an array. This enables average-case constant time complexity O(1)*O*(1) for search, insert, and delete operations, making hash tables highly efficient for various applications, including associative arrays and database indexing.

The architecture of a hash table involves:

- **Hash Function**: Converts the key into a hash value.
- **Index Calculation**: The hash value is then mapped to an index using the formula:
- **Buckets**: To handle collisions (when multiple keys hash to the same index), hash tables use buckets, often implemented as linked lists

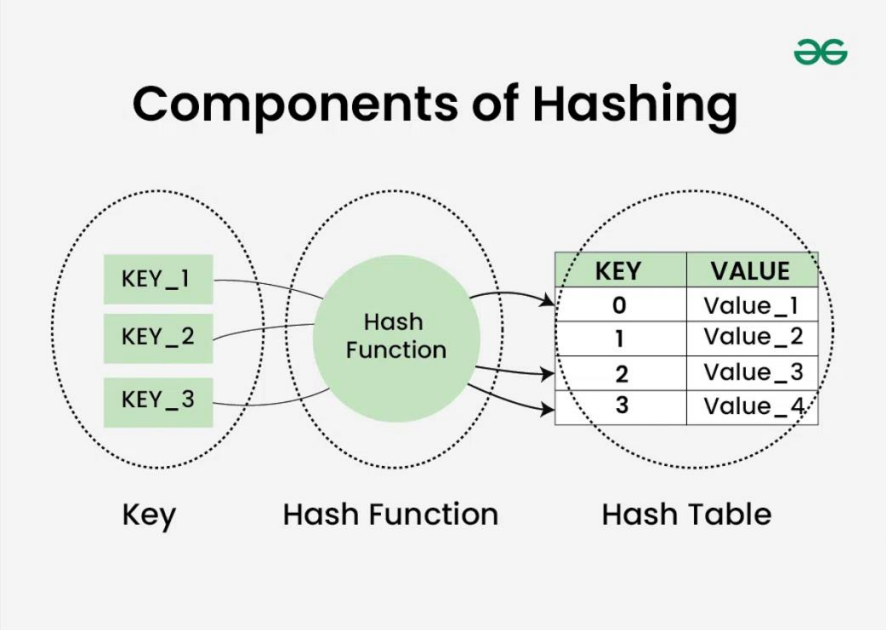

Figure 1: Hash Function and Table.

Implementation Outlines

The implementation of a hash table typically follows these steps:

1. **Initialization**: Create an array of a fixed size to hold the buckets.
2. **Hash Function**: Define a hash function to compute the index from the key.
3. **Insertion**:
    - Compute the hash of the key.
    - Calculate the index using the modulo operation.
    - Insert the key-value pair into the appropriate bucket.
4. **Searching**:
    - Compute the hash and index for the key.
    - Traverse the bucket at that index to find the corresponding value.
5. **Deletion**:
    - Similar to searching, but remove the key-value pair from the bucket

Applications of Hash Tables

Hash tables are versatile data structures widely used in computer science and software development due to their efficiency in data retrieval and storage. Here are some key applications of hash tables:

1. **Associative Arrays**:
    - Hash tables are commonly used to implement associative arrays, which map unique keys to values. This allows for quick insertion, deletion, and lookup operations, making them ideal for scenarios where fast access to data is required.
2. **Database Indexing**:
    - In databases, hash tables facilitate rapid data access by indexing records. They enable efficient searching and retrieval of data, significantly improving performance compared to traditional search methods like linear search.
3. **Caching**:
    - Hash tables are extensively used in caching mechanisms to store frequently accessed data. By using a hash function to quickly locate cached items, systems can reduce access times and enhance overall performance.

4. **Symbol Tables in Compilers**:
    - In programming language compilers, hash tables serve as symbol tables that store variable names and their associated information (like type and scope). This allows for efficient lookups during the compilation process.
5. **Spell Checkers**:
    - Hash tables are utilized in spell checkers to store dictionaries of words. The ability to quickly check the existence of a word enhances the efficiency of text processing applications.
6. **Frequency Analysis**:
    - They are employed in applications that require counting occurrences of elements, such as analyzing text or data streams. By hashing each element, systems can efficiently track how many times each item appears.
7. **Cryptography**:
    - Hash tables play a crucial role in cryptographic applications, where they help map plaintext to ciphertext securely. They are also used in creating digital signatures and ensuring data integrity through hashing function.
8. **Load Balancing**:
    - In distributed systems, hash tables help evenly distribute incoming requests across multiple servers based on hashed keys, which is essential for maintaining system performance and reliability.
9. **Detecting Duplicates**:
    - Hash tables can efficiently identify duplicate entries in datasets by storing previously seen items as keys and checking against them during insertion operations.

2. Graphs

What Are Graphs?

Graphs are a **non-linear data structure** that consist of a set of vertices (also called nodes) and edges that connect these vertices. This structure allows for the representation of complex relationships and networks, making graphs suitable for various applications, such as social networks, transportation systems, and circuit designs. In graph theory, a graph is formally defined as a pair of sets G(*V*,*E*), where V is the set of vertices and E is the set of edges connecting these vertices.

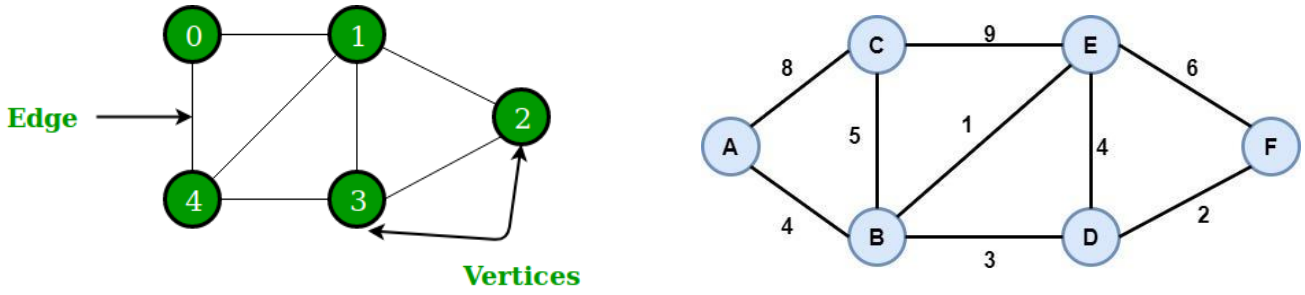

Figure 2: Graphs.

Key Terminologies

Understanding graphs involves familiarizing oneself with several key terms:

- **Vertex (Node)**: A fundamental unit of a graph representing an object.
- **Edge**: A connection between two vertices, indicating a relationship.
- **Adjacent Vertices**: Two vertices connected by an edge.
- **Degree of a Vertex**: The number of edges connected to a vertex.
- **Path**: A sequence of vertices where each adjacent pair is connected by an edge.
- **Cycle**: A path that starts and ends at the same vertex without repeating any edges or vertices.

Types of Graphs

Graphs can be categorized into several types based on their properties:

1. **Directed Graphs (Digraphs)**: In these graphs, edges have a direction, indicating a one-way relationship between vertices. For example, if there is an edge from vertex A to vertex B, it does not imply there is an edge from B to A.
2. **Undirected Graphs**: These graphs feature edges without direction, meaning the relationship between connected vertices is bidirectional.

3. **Weighted Graphs**: Each edge in a weighted graph has an associated weight or cost, representing distances or other metrics relevant to the relationship between the vertices.
4. **Unweighted Graphs**: Edges do not have weights; they simply indicate connections between vertices.
5. **Connected Graphs**: In these graphs, there exists a path between every pair of vertices, ensuring no isolated nodes.
6. **Complete Graphs**: Every vertex is connected to every other vertex in the graph.
7. **Cyclic Graphs**: These contain at least one cycle, while acyclic graphs do not contain cycles.

Implementation Outlines of Graphs

Graphs can be implemented using various methods, primarily through **adjacency matrices** or **adjacency lists**. Each method has its advantages and is suitable for different types of applications. Below are the outlines for implementing graphs using both methods, along with examples in Python.

1. Adjacency Matrix Implementation

An **adjacency matrix** is a 2D array where each cell at position (i, j) indicates whether there is an edge between vertex i and vertex j. This method is particularly efficient for dense graphs.

Implementation Steps

1. **Initialize the Matrix**: Create a square matrix of size n × n initialized to 0, where n is the number of vertices.
2. **Add Edges**: For an undirected graph, set both matrix[i][j] and matrix[j][i] to 1 (or the weight if it's a weighted graph). For directed graphs, only set matrix[i][j].
3. **Add Vertex Data**: Optionally maintain an array to store data associated with each vertex.

2. Adjacency List Implementation

An **adjacency list** consists of an array of lists. The index represents the vertex and each element in the list contains the vertices that are adjacent to it. This method is more space-efficient for sparse graphs.

Implementation Steps

1. **Initialize the List**: Create an empty list for each vertex.
2. **Add Edges**: Append the destination vertex to the source vertex's list and vice versa for undirected graphs.
3. **Add Vertex Data**: Optionally maintain a separate structure for vertex data.

Applications of Graphs

Graphs have numerous practical applications across various fields:

- **Social Networks**: Representing users as vertices and their relationships (friends, followers) as edges.
- **Transportation Networks**: Modeling routes and connections in road systems or flight paths.
- **Web Page Linking**: Pages are represented as vertices and hyperlinks as edges, facilitating search engine indexing.
- **Network Routing**: Helping in determining optimal paths for data transmission across networks.