Advanced Data Structures

Hash Tables, Trees, Graphs, and More

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Overview of Hash Tables

A hash table, also known as a hash map, is a data structure that provides an efficient way to store and retrieve values using keys. At its core, a hash table is a collection of key-value pairs, where each key is unique. The key is processed through a hash function to produce an index in an array, where the corresponding value is stored. This index helps in directly accessing the data rather than searching through a list, making operations faster.

Key Characteristics

- **Key-Value Pairs:** Data is stored in pairs, with each key mapping to a specific value.
- **Hash Function:** A function that takes a key and computes an index (or hash code) in the array where the value should be stored.
- **Table (Array):** An array that stores the values. The size of this array is typically a prime number to minimize collisions.

Real-World Analogies

To better understand the concept of hash tables, consider the following real-world analogies:

Library Indexing System

Imagine a library where books are indexed by a unique identifier. When you want to find a book, you don't search through every shelf. Instead, you use the identifier to directly locate the book on a specific shelf. The identifier acts as the key, the book as the value, and the indexing system is akin to the hash function directing you to the right shelf (array index).

Phone Directory

A phone directory stores names and their corresponding phone numbers. Instead of flipping through the entire directory to find a number, you use the alphabetical index to quickly locate the name and retrieve the number. Here, the name is the key, the phone number is the value, and the alphabetical index represents the hash function and array index.

Inner Workings

Hash Functions: Properties and Importance

A hash function is a critical component of a hash table. It takes a key as input and produces a hash code, which is then mapped to an index in the array where the corresponding value is stored. The efficiency of a hash table largely depends on the quality of its hash function.

Properties of a Good Hash Function

- **Deterministic:** The hash function should always produce the same hash code for the same key.
- **Uniform Distribution:** The function should distribute keys uniformly across the array to minimize collisions.
- **Fast to Compute:** The hash function should be efficient to calculate to maintain the overall performance of the hash table.
- **Minimizes Collisions:** While collisions (when two keys hash to the same index) are inevitable, a good hash function reduces their frequency.

Collision Resolution Techniques

Despite the best efforts in designing a hash function, collisions can occur. There are several techniques to handle collisions:

Chaining In chaining, each index in the array points to a linked list (or another data structure) of all entries that hash to the same index.

- **Advantages:** Simple to implement and can handle an unlimited number of collisions.
- **Disadvantages:** Can lead to increased memory usage and potentially longer search times if many keys collide at the same index.

Open Addressing Instead of using linked lists, open addressing stores all entries directly in the array. When a collision occurs, the algorithm probes the array to find the next available slot.

Types of Probing

- **Linear Probing:** Check the next slot sequentially until an empty slot is found.
- **Quadratic Probing:** Check slots at increasing intervals (i.e., 1, 4, 9, etc.).
- **Double Hashing:** Use a second hash function to determine the interval for probing.
- **Advantages:** More cache-friendly and can be more space-efficient.
- **Disadvantages:** Can lead to clustering, where a group of consecutive occupied slots forms, increasing the time required to find an empty slot.

Use Cases

Practical Applications in Software Development

Hash tables are widely used in various applications due to their efficiency in data retrieval. Some common use cases include:

Database Indexing Hash tables are often used to index database records. They allow for quick lookup of records by key, such as a primary key in relational databases.

Caches Implementing caches with hash tables enables fast access to frequently accessed data. Web browsers, for instance, use hash tables to cache visited web pages.

Symbol Tables In compilers and interpreters, hash tables store variable names (symbols) and their associated values or references. This enables quick lookup of variable values during program execution.

Sets and Dictionaries Hash tables form the underlying structure for sets and dictionaries in many programming languages, allowing for efficient membership tests, insertions, and deletions.

Routing Tables Networking applications use hash tables to store routing information, mapping IP addresses to routes quickly.

Introduction to Trees

A tree is a hierarchical data structure consisting of nodes, where each node has a value and a list of references to other nodes (its children). The top node in a tree is called the root. Each node (except the root) has exactly one parent. Nodes with no children are called leaves or leaf nodes.

Properties

- Trees represent hierarchical relationships, making them suitable for data that naturally forms a hierarchy.
- Trees do not contain cycles; there is only one path between any two nodes.
- There is a unique path from the root to any other node in the tree.

Types of Trees and Their Significance

1. General Tree

Each node can have an arbitrary number of children. General trees are versatile and can represent various hierarchical data such as organizational structures or file systems.

2. Binary Tree

Each node has at most two children, referred to as the left child and the right child. Binary trees are the foundation for many other tree structures and algorithms due to their simplicity and balance properties.

3. Binary Search Tree (BST)

A binary tree where each node follows the property that all nodes in the left subtree are less than the node, and all nodes in the right subtree are greater. BSTs allow for efficient searching, insertion, and deletion operations.

4. Balanced Tree

A tree where the height difference between left and right subtrees is minimized. Balanced trees maintain optimal performance for operations like insertion, deletion, and searching.

5. Heap

A complete binary tree that satisfies the heap property (min-heap or max-heap). Heaps are used in priority queues and for implementing efficient sorting algorithms.

6. Trie

A tree-like structure that stores a dynamic set of strings, where each node represents a single character of a string. Tries are efficient for searching and sorting strings, making them useful in applications like autocomplete and spell checking.

Binary Trees

Structure

Each node in a binary tree has at most two children, commonly referred to as the left and right child. The tree is defined recursively, with the root node containing a value and references to its left and right subtrees.

Types

- **Full Binary Tree:** Each node has either 0 or 2 children. This type ensures that all nodes have the maximum number of children.
- **Complete Binary Tree:** All levels are fully filled except possibly for the last level, which is filled from left to right. This structure is important for efficiently implemented binary heaps.
- **Perfect Binary Tree:** All internal nodes have exactly two children, and all leaf nodes are at the same level. This tree is both full and complete, offering a highly balanced structure.
- **Balanced Binary Tree:** The height difference between the left and right subtrees of any node is at most one. Balanced trees ensure logarithmic height, making operations efficient.

Traversal Methods

Tree traversal refers to the process of visiting each node in a tree data structure, exactly once, in a specific order.

- **In-Order Traversal:** Visit the left subtree, the root node, and then the right subtree. This traversal yields the nodes in non-decreasing order for BSTs.
- **Pre-Order Traversal:** Visit the root node, the left subtree, and then the right subtree. This method is used to create a copy of the tree.
- **Post-Order Traversal:** Visit the left subtree, the right subtree, and then the root node. This traversal is useful for deleting a tree or evaluating postfix expressions.

Binary Search Trees (BST)

Characteristics

- **Ordered Structure:** For any given node, the values of the nodes in the left subtree are smaller, and the values in the right subtree are larger.
- **Dynamic:** BSTs support dynamic set operations, allowing for efficient searching, insertion, and deletion.
- **Recursive Nature:** The properties of BSTs apply to every subtree within the tree.

Operations

Insertion:

- Start at the root and recursively find the correct position for the new node.
- Place the new node as a leaf in its correct position.

Deletion:

- Find the node to be deleted.
- There are three cases to handle:
  - Node with no children (leaf node): Simply remove the node.
  - Node with one child: Remove the node and replace it with its child.
  - Node with two children: Find the in-order successor (smallest node in the right subtree) or in-order predecessor (largest node in the left subtree) and replace the node's value, then delete the successor or predecessor node.

Search:

- Begin at the root and recursively compare the target value with the current node's value.
- Traverse left or right subtrees accordingly until the target value is found or the subtree is null.

Balanced BSTs

1. **AVL Trees:**

   - **Definition:** A self-balancing BST where the height difference (balance factor) between left and right subtrees of any node is at most one.
   - **Operations:** Perform rotations (left, right, left-right, right-left) to maintain balance after insertions and deletions.
   - **Significance:** AVL trees provide guaranteed logarithmic time complexity for search, insertion, and deletion.

2. **Red-Black Trees:**

   - **Definition:** A self-balancing BST with an additional property: nodes are colored either red or black to ensure balanced height.
   - **Properties:**
     - Root is always black.
     - Red nodes cannot have red children (no two reds in a row).
     - Every path from a node to its descendants has the same number of black nodes.
   - **Operations:** Use rotations and color changes to maintain balance after insertions and deletions.
   - **Significance:** Red-black trees offer efficient insertion, deletion, and search operations with good performance in practice.

Heaps

Min-Heap:

- **Structure:** A complete binary tree where the value of each node is greater than or equal to the value of its parent.
- **Property:** The root node contains the minimum value in the heap.
- **Operations:** Support efficient extraction of the minimum element, insertion of new elements, and heapify operations to maintain the heap property.

Max-Heap:

- **Structure:** A complete binary tree where the value of each node is less than or equal to the value of its parent.
- **Property:** The root node contains the maximum value in the heap.
- **Operations:** Support efficient extraction of the maximum element, insertion of new elements, and heapify operations to maintain the heap property.

Applications in Priority Queues

Heaps are widely used to implement priority queues, which are abstract data types where each element has a priority assigned to it. In a priority queue, elements with higher priority are served before elements with lower priority.

Operations in Priority Queues:

- **Insertion:** Add a new element with an associated priority.
- **Find-Min/Find-Max:** Retrieve the element with the highest priority (minimum or maximum in the context of heaps).
- **Extract-Min/Extract-Max:** Remove and return the element with the highest priority.
- **Decrease-Key/Increase-Key:** Adjust the priority of an element and reheapify to maintain the heap property.

Priority queues are essential in algorithms like Dijkstra's shortest path and Huffman coding.

Trie

Structure and Use Cases in Text Processing

Structure:

- A trie (pronounced "try") is a tree-like data structure that stores a dynamic set of strings, where each node represents a single character of a string.
- The root node is empty, and each edge represents a character.
- Paths from the root to the leaf nodes correspond to strings stored in the trie.
- Nodes can store additional information such as end-of-word markers to denote complete strings.

Use Cases:

- **Autocomplete:** Tries are used to suggest completions for a given prefix. By traversing the trie from the root to the end of the prefix, all possible completions can be efficiently retrieved.
- **Spell Checking:** Tries can quickly verify the presence of a word and suggest corrections by traversing paths corresponding to valid words.
- **Longest Prefix Matching:** Used in networking to determine the longest matching prefix of an IP address.
- **Text Search:** Efficiently search for patterns within text, especially useful in applications like search engines.

Tries offer efficient solutions for problems involving a large set of strings, providing operations like insertion, deletion, and search in a time complexity that is proportional to the length of the string being processed.

Introduction to Graphs

Definition and Terminology

Graphs are fundamental data structures used in computer science to represent networks of interconnected entities. A graph G is defined as an ordered pair $G = (V, E)$, where:

- V is a set of vertices (also called nodes or points).
- E is a set of edges (also called links or lines) that connect pairs of vertices.

Vertices

Vertices (or Nodes): The fundamental units of which graphs are formed. In a graph $G = (V, E)$, the set V contains the vertices. Each vertex can be uniquely identified, often by a label or number.

Edges

Edges (or Links): The connections between vertices. In $G = (V, E)$, the set E contains the edges. Each edge is defined by a pair of vertices it connects. For example, an edge connecting vertex u and vertex v can be represented as $e = (u, v)$.

Adjacency

Adjacency: Two vertices are said to be adjacent if there is an edge connecting them. In other words, vertex u is adjacent to vertex v if and only if there exists an edge $(u, v) \in E$.

Types of Graphs

Undirected Graph: A graph in which edges have no direction. The edge (u, v) is identical to the edge (v, u). Undirected graphs are useful for representing bidirectional relationships, such as friendships in a social network.

Example:

- Vertices: A, B, C
- Edges: A - B, B - C, A - C

Directed Graph (Digraph): A graph in which edges have a direction. The edge (u, v) is not the same as the edge (v, u). Directed graphs are useful for representing one-way relationships, such as following on Twitter or web page links.

Example:

- Vertices: A, B, C
- Edges: A $\rightarrow$ B, B $\rightarrow$ C, C $\rightarrow$ A

Unweighted Graph: A graph in which all edges are treated equally. There is no weight or cost associated with traversing an edge.

Example:

- Vertices: A, B, C
- Edges: A - B, B - C, A - C

Weighted Graph: A graph in which each edge has an associated weight (or cost). This weight can represent various quantities, such as distance, time, or capacity. Weighted graphs are crucial for problems involving optimization, such as finding the shortest path or the minimum spanning tree.

Example:

- Vertices: A, B, C
- Edges: A - B (weight: 3), B - C (weight: 5), A - C (weight: 2)

Special Graphs

Bipartite Graph

A graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. Bipartite graphs are used in modeling problems like job assignments and matching problems.

Example:

- Sets of Vertices: {U1, U2} and {V1, V2, V3}
- Edges: U1 - V1, U1 - V2, U2 - V2, U2 - V3

Complete Graph

A graph in which every pair of distinct vertices is connected by a unique edge. A complete graph with n vertices is denoted as K_n and has $\frac{n(n-1)}{2}$ edges.

Example:

- Vertices: A, B, C, D
- Edges: A - B, A - C, A - D, B - C, B - D, C - D

Cyclic Graph

A graph that contains at least one cycle, which is a path that starts and ends at the same vertex without traversing any edge more than once.

Example:

- Vertices: A, B, C, D
- Edges: A - B, B - C, C - D, D - A

Acyclic Graph

A graph with no cycles. Acyclic graphs include trees (connected acyclic undirected graphs) and directed acyclic graphs (DAGs), which are used in modeling hierarchical structures and scheduling problems.

Example:

- Vertices: A, B, C, D
- Edges: A - B, A - C, B - D

Graph Representations

Example Graph for Representation:

- Vertices: A, B, C
- Edges: A - B, B - C

Adjacency Matrix

An adjacency matrix is a 2-dimensional array (or matrix) used to represent a graph. The rows and columns represent vertices, and the entries indicate the presence (and weight) of edges.

Example:

	A	B	C
A	0	1	0
B	1	0	1
C	0	1	0

Structure: For a graph with n vertices, the adjacency matrix is an $n \times n$ matrix A, where $A[i][j]$ represents the edge between vertex i and vertex j.

- In an unweighted graph, $A[i][j] = 1$ if there is an edge between i and j; otherwise, $A[i][j] = 0$.
- In a weighted graph, $A[i][j] = w$ if there is an edge with weight w between i and j; otherwise, $A[i][j] = 0$.

Advantages: Easy to implement and efficient for dense graphs where the number of edges is close to the maximum number of edges $\left(\frac{n(n-1)}{2}\right)$ for undirected graphs.

Disadvantages: Space inefficient for sparse graphs because it requires $O(n^2)$ space regardless of the number of edges.

Adjacency List

An adjacency list is a collection of lists or arrays used to represent a graph. Each vertex has a list of adjacent vertices.

Example:

- A: B
- B: A, C
- C: B

Structure: For a graph with n vertices, the adjacency list consists of n lists. The i-th list contains all the vertices adjacent to the i-th vertex.

- In an unweighted graph, the list contains the vertices directly connected by edges.
- In a weighted graph, each element in the list is a pair, consisting of a vertex and the weight of the edge connecting them.

Advantages: Space efficient for sparse graphs, using $O(n + e)$ space, where e is the number of edges. It is also efficient for iterating over all edges.

Disadvantages: Can be less intuitive than an adjacency matrix and slightly more complex to implement.

Edge List

An edge list is a list of all edges in the graph. Each edge is represented as a pair (or tuple) of vertices, and optionally a weight.

Example:

- (A, B)
- (B, C)

Structure: For a graph with e edges, the edge list contains e pairs. Each pair (u, v) (or (u, v, w) for weighted graphs) represents an edge from vertex u to vertex v (with weight w if applicable).

Advantages: Simple to implement and efficient for certain operations like edge iteration. Space efficient for sparse graphs.

Disadvantages: Less efficient for operations that require checking for the existence of an edge or finding all adjacent vertices.

Introduction to Circular Buffers

A circular buffer, also known as a ring buffer, is a fixed-size data structure that uses a single, contiguous block of memory arranged in a circular fashion. This means that the end of the buffer wraps around to the beginning, creating a continuous loop.

Circular buffers are used to efficiently manage data streams where continuous reading and writing operations occur. They are particularly useful in situations where the buffer size is known in advance and memory efficiency is crucial. Typical applications include buffering data between producers and consumers at different rates, managing real-time data streams, and handling multimedia data.

Key Characteristics

- **Fixed Size:** The buffer has a predetermined size, defined at initialization.
- **Efficient Memory Usage:** Only a fixed block of memory is used, avoiding dynamic memory allocation.
- **Wrap-Around:** When the end of the buffer is reached, operations continue from the beginning, making the structure circular.
- **Concurrency:** Circular buffers are suitable for concurrent access, with one thread writing data (producer) and another reading data (consumer).

Structure of a Circular Buffer

Buffer Size and Capacity

The buffer size is set at the time of creation and remains constant throughout its lifecycle. This size determines the maximum number of elements the buffer can hold at any given time. The capacity is crucial for ensuring that the buffer does not overflow or underflow during operations.

Head and Tail Pointers

Two pointers, often referred to as the head and tail, are used to manage the circular buffer:

- **Head Pointer:** Indicates the position from which the next element will be read (or dequeued).
- **Tail Pointer:** Indicates the position where the next element will be written (or enqueued).

The positions of these pointers change dynamically as data is added or removed from the buffer.

Wrap-Around Mechanism

When either the head or tail pointer reaches the end of the buffer, it wraps around to the beginning. This circular behavior ensures continuous operation without needing to shift elements within the buffer.

Operations on Circular Buffers

Enqueue (Add)

Adding an element to the circular buffer is called enqueuing.

- **Detailed Steps:**
  1. Check if the buffer is full.
  2. If not full, place the new element at the tail position.
  3. Move the tail pointer to the next position.
  4. If the tail reaches the end of the buffer, wrap it around to the beginning.
- **Handling Full Buffers:** If the buffer is full, typically one of two actions is taken: either overwrite the oldest data or reject new data until space is available.

Dequeue (Remove)

Removing an element from the circular buffer is called dequeuing.

- **Detailed Steps:**
  1. Check if the buffer is empty.
  2. If not empty, retrieve the element at the head position.
  3. Move the head pointer to the next position.
  4. If the head reaches the end of the buffer, wrap it around to the beginning.
- **Handling Empty Buffers:** If the buffer is empty, the dequeue operation cannot proceed and typically an error or a special empty indicator is returned.

Peek (Read without Removing)

Peeking allows reading the value at the head without removing it.

- **Steps and Use Cases:**
  1. Check if the buffer is empty.
  2. If not empty, read the value at the head position.

  The head pointer remains unchanged. Peeking is useful in situations where data needs to be inspected without consuming it, such as in monitoring applications.

IsEmpty and IsFull

- **Conditions and Checks:**
  - **IsEmpty:** The buffer is empty if the head and tail pointers are equal and no data is available to read.
  - **IsFull:** The buffer is full if the next position of the tail pointer equals the head pointer, indicating no space for new data.

Advantages and Disadvantages

Memory Efficiency

Circular buffers use a fixed amount of memory, making them predictable in terms of memory usage. This is particularly beneficial in embedded systems and real-time applications where memory is constrained.

Performance Benefits

Circular buffers provide constant time complexity (O(1)) for both enqueue and dequeue operations, ensuring fast and predictable performance. The fixed size and lack of dynamic memory allocation contribute to their high efficiency.

Limitations

- **Fixed Size:** The fixed size can be a limitation if the buffer needs to handle variable amounts of data, potentially leading to overflow or underutilization.
- **Complexity in Handling:** Managing wrap-around behavior and ensuring correct concurrent access require careful implementation.

Use Cases of Circular Buffers

Real-Time Data Streaming

Circular buffers are ideal for real-time data streaming applications, such as video playback and audio processing, where continuous, smooth data flow is essential.

Multithreaded Programming

In multithreaded environments, circular buffers effectively manage data between producer and consumer threads, preventing race conditions and ensuring thread-safe operations.

Network Packet Management

Network applications use circular buffers to manage incoming and outgoing packets efficiently, ensuring that data packets are processed in a timely manner without loss.

Audio and Video Buffers

Circular buffers handle audio and video data streams, providing stable playback and recording by managing data buffering effectively, even when data is produced and consumed at different rates.