Data Structures - Quick Notes

Data structures are fundamental components in computer science, designed to organize, manage, and store data efficiently to enable ease of access and modification. They serve as the backbone for writing efficient algorithms and are essential for problem-solving in various domains, from software development to data science and artificial intelligence. The choice of data structure can greatly impact the performance and complexity of a program. There are several types of data structures, each serving different purposes and offering unique benefits depending on the use case.

Arrays are one of the simplest and most widely used data structures. They store elements in contiguous memory locations and allow fast access through indexing. Arrays are ideal when the number of elements is fixed and frequent read operations are required. However, insertion and deletion operations can be costly as they require shifting of elements. Linked Lists, on the other hand, consist of nodes that are dynamically allocated. Each node contains data and a reference to the next node. This dynamic nature allows efficient insertions and deletions, especially in situations where the size of the dataset changes frequently. There are various types of linked lists such as singly, doubly, and circular linked lists, each with specific advantages in different contexts.

Stacks and Queues are linear data structures that manage data in a particular order. A Stack follows the Last In First Out (LIFO) principle, where the most recently added item is the first to be removed. It supports operations like push, pop, and peek, and is commonly used in scenarios like expression evaluation, function call management, and undo mechanisms. A Queue follows the First In First Out (FIFO) principle, where the oldest item is processed first. It is widely used in scheduling tasks, handling requests in systems, and managing buffers. Variations of queues include circular queues, priority queues, and double-ended queues (deques), which allow for more flexible data handling.

Data Structures - Quick Notes

Trees are hierarchical data structures made up of nodes connected by edges. Each tree has a root node and subtrees of children, forming a parent-child hierarchy. Binary Trees, where each node has at most two children, are the most common. Binary Search Trees (BSTs) are structured such that the left child is less than the parent and the right child is greater, allowing efficient searching, insertion, and deletion. More advanced trees like AVL trees and B-trees are used to maintain balance and enable fast access to data. Trees are extensively used in databases, compilers, and file systems.

Graphs are complex data structures used to represent networks. A graph consists of vertices (or nodes) and edges connecting them. Graphs can be directed or undirected, weighted or unweighted, and are commonly used in scenarios like social networking, route planning, and web page ranking. They can be represented using adjacency matrices or adjacency lists depending on the density of the graph. Graph traversal algorithms like Depth First Search (DFS) and Breadth First Search (BFS) are key for exploring these structures.

Hashing is another important concept that allows for efficient data retrieval. Hash tables use a hash function to compute an index into an array of buckets from which the desired value can be found. They provide average-case constant time complexity for search, insert, and delete operations. Collision resolution techniques such as chaining and open addressing are used to handle scenarios when two keys hash to the same index.

Understanding the time and space complexity of operations on different data structures is critical. This is typically represented using Big O notation, which describes the worst-case performance of an algorithm. For example, accessing an element in an array is O(1), whereas searching in a linked list is O(n). Selecting the appropriate data structure involves evaluating the types of operations needed and balancing trade-offs between time complexity, space efficiency, and code maintainability.