A **data structure** is a fundamental concept in computer science and refers to the way data is **organized** and **stored** in a computer's memory or storage devices. Data structures provide a way to efficiently **store**, **access**, and **manipulate** data, allowing programmers to perform various operations on the data effectively.

|  |  |
| --- | --- |
| **Applications of Data Structures** | |
| Databases | Game Development |
| Text Editors and Word Processors | Computer Graphics |
| Operating Systems | Search Engines |
| Compiler Design | Cryptography |
| Networking | Embedded Systems |
| Graphics and Image Processing | Financial Systems |
| Artificial Intelligence and Machine Learning | Bioinformatics |
| Geographic Information Systems (GIS) | AI Planning and Robotics |
| Web Development | Data Compression |

|  |
| --- |
| **The linked list is an example of a linear or non-linear data structure?** |
| A linked list can be a subset of both linear and nonlinear, it depends on its usage & application. If it is used for access strategies, then it is a part of linear structures but when used for data storage, it can be called a non-linear data structure. |

|  |
| --- |
| **What is the difference between NULL and Void?**   * The **void** is a data type **identifier** while **NULL** is a **value**. * Void indicates that the pointer has **no initial size** while NULL indicates an **empty value** for a variable. * Void means that value exists but is not in effect while Null means that the value never existed. |

|  |
| --- |
| **Drawbacks of array implementation of Queue:**   1. Fixed Size 2. Wasted Space 3. Dynamic Resizing Overhead 4. Slower Dequeue Operations 5. Inefficient Enqueue Operations 6. Memory Fragmentation 7. Complex Implementation 8. Not Suitable for Real-time Applications |

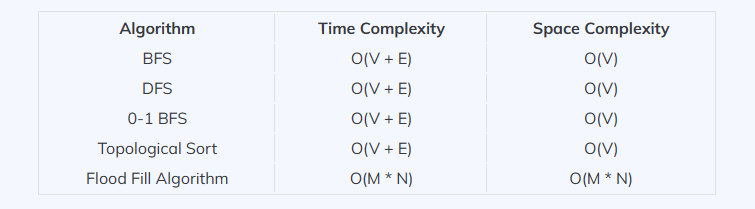
|  |
| --- |
| **The elements of a 2D array are stored in the memory:**   1. Row-Major Order (C/C++ and many other languages) 2. Column-Major Order (Fortran and some other languages) |

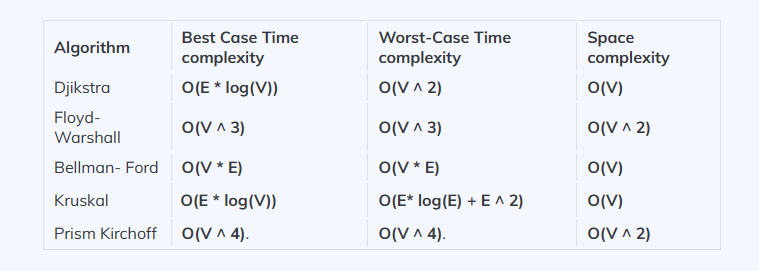
|  |
| --- |
| **Advantages of Linked Lists over Arrays:**   1. Dynamic Size 2. Constant-Time Insertions and Deletions 3. Memory Efficiency 4. No Fixed Size 5. Ease of Insertion at the Beginning 6. Support for Complex Data Structures |
| **Advantages of Arrays over Linked Lists:**   1. Constant-Time Random Access 2. Cache Locality 3. Simplicity 4. Predictable Memory Overhead |

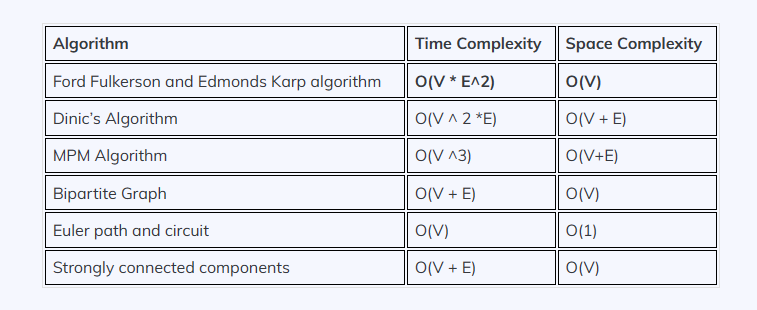
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Different types of sorting algorithm** | | | | |
| **Algorithm Name** | **Time Complexity** | | | **Space Complexity** |
| **Worst-case** | **Average-case** | **Best-case** |
| **Bubble Sort** | O(n^2) | O(n^2) | O(n) | O(1) |
| **Selection Sort** | O(n^2) | O(n^2) | O(n^2) | O(1) |
| **Insertion Sort** | O(n^2) | O(n^2) | O(n) | O(1) |
| **Merge Sort** | O(n log n) | O(n log n) | O(n log n) | O(n) |
| **Quick Sort** | O(n^2) | O(n log n) | O(n log n) | O(log n) to O(n) |
| **Heap Sort** | O(n log n) | O(n log n) | O(n log n) | O(1) |
| **Tim Sort** | O(n log n) | O(n log n) | O(n) | O(n) |
| **Radix Sort** | O(nk) | O(nk) | O(nk) | O(n + k) |
| **Counting Sort** | O(n + k) | O(n + k) | O(n + k) | O(k) |

QuickSort is often considered one of the fastest sorting algorithms in practice for a wide range of scenarios. Quick Sort and Merge Sort, are often preferred in practice for their good average-case performance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Different types of Graph Algorithm, Time and Space Complexity and Use Cases** | | | |
| **Algorithm** | **Time Complexity** | **Space Complexity** | **Use Cases** |
| **Breadth-First Search (BFS)** | O(V + E) | O(V) | find the shortest path in an unweighted graph, explore all vertices at a given depth, finding connected components and detecting cycles |
| **Depth-First Search (DFS)** | O(V + E) | O(V) | used for topological sorting, finding strongly connected components, solving maze problems, and exploring all paths in a graph. |
| **Dijkstra's Algorithm (Single-Source Shortest Path)** | O((V + E) \* log(V)) using a priority queue. | O(V) | finds the shortest path in weighted graphs with non-negative edge weights, such as in routing and navigation systems. |
| **Bellman-Ford Algorithm (Single-Source Shortest Path)** | O(V \* E) | O(V) | handles graphs with negative edge weights and is used when Dijkstra's algorithm is not suitable. |
| **Floyd-Warshall Algorithm (All-Pairs Shortest Path)** | O(V^3) | O(V^2) | finds the shortest paths between all pairs of vertices in weighted graphs and is used in network optimization. |
| **Kruskal's Algorithm** | O(E \* log(E)) or O(E \* log(V)) | O(E + V) | finds the minimum spanning tree in weighted graphs, commonly used in network design and clustering. |
| **Prim's Algorithm** | O(V^2) or O(E + V \* log(V)) | O(V) | finds a minimum spanning tree and is used in network design and clustering problems. |
| **Topological Sorting (DAGs - Directed Acyclic Graphs)** | O(V + E) | O(V) | used in scheduling tasks with dependencies, like project management and build systems. |
| **Strongly Connected Components (Kosaraju's Algorithm or Tarjan's Algorithm)** | O(V + E) | O(V) | Finding strongly connected components helps analyze the structure of directed graphs and solve reachability problems. |
| **Max Flow (Ford-Fulkerson Algorithm or Edmonds-Karp Algorithm)** | O(E \* f) | O(V + E) | used in network flow problems, such as optimizing traffic flow in networks. |







|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Different types of tree algorithm** | | | | | |
| **Algorithm Name** | **Time Complexity** | | | **Space Complexity** | **Use Case** |
| **Insertion** | **Deletion** | **Search** |
| **Binary Search Tree (BST)** | O(log n) | O(log n) | O(log n) | O(n) | maintaining sorted data efficiently and enabling operations like searching, insertion, and deletion. |
| **Balanced Binary Search Trees (AVL Trees, Red-Black Trees)** | O(log n) | O(log n) | O(log n) | O(n) | ensure that the tree remains balanced, providing consistent logarithmic-time performance for various operations |
| **Heap (Binary Heap, Fibonacci Heap)** | O(log n) | O(log n) | O(n) | O(n) | used for priority queues and efficiently finding the minimum or maximum element in a collection. |
| **Binary Tree Traversals (Inorder, Preorder, Postorder):** | | O(n) | | | printing tree elements, evaluating expressions, and building expression trees. |
| **Binary Indexed Tree (Fenwick Tree)** | O(log n) | | | O(n) | used for efficient range queries and updates in arrays, especially in scenarios like cumulative frequency calculations. |
| **Trie (Prefix Tree)** | O(m) |  | O(m) | O(n\*m) | used for efficient retrieval of words with common prefixes in dictionaries, spell checkers, and IP routing. |
| **Segment Tree** | Construction : O(n) | Update: O(log n) | O(log n) | O(4n) | used for efficient range query operations (e.g., minimum, maximum, sum) on arrays. |
| **Binary Search on Trees** | O(log n) | | | O(h) | used to search for elements in sorted trees like BSTs and AVL trees. |
| **Huffman Coding** | O(n log n) | | | O(n) | used for data compression, particularly in file compression algorithms. |
| **Cartesian Tree** | * Construction: O(n) * Range Queries: O(log n) | | | O(n) | used for efficient range queries on arrays and for constructing other data structures like the Cartesian Tree of an array. |