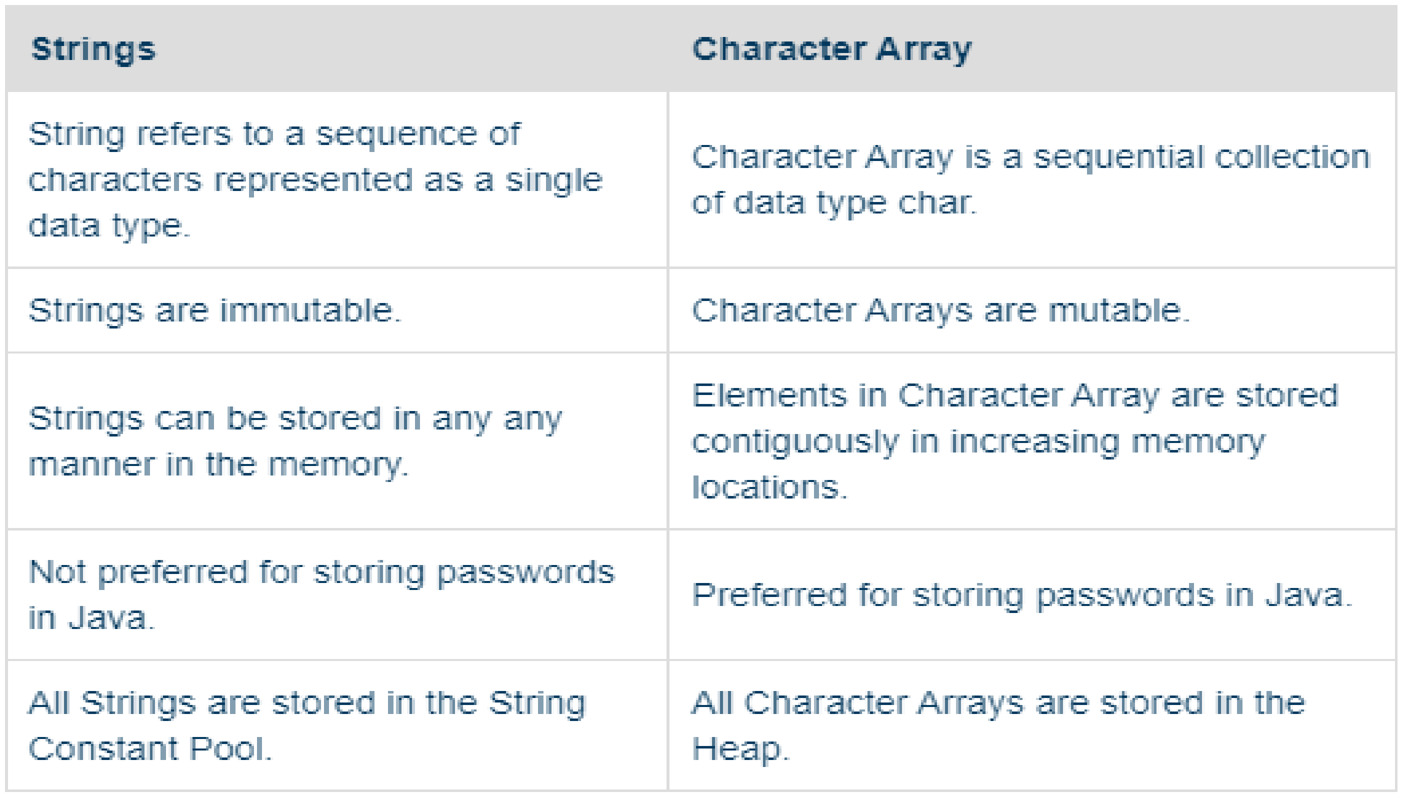
A **data structure** is a fundamental concept in computer science and refers to the way data is **organized** and **stored** in a computer's memory or storage devices. Data structures provide a way to efficiently **store**, **access**, and **manipulate** data, allowing programmers to perform various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



Strings	Character Array
String refers to a sequence of characters represented as a single data type.	Character Array is a sequential collection of data type char.
Strings are immutable.	Character Arrays are mutable.
Strings can be stored in any any manner in the memory.	Elements in Character Array are stored contiguously in increasing memory locations.
Not preferred for storing passwords in Java.	Preferred for storing passwords in Java.
All Strings are stored in the String Constant Pool.	All Character Arrays are stored in the Heap.

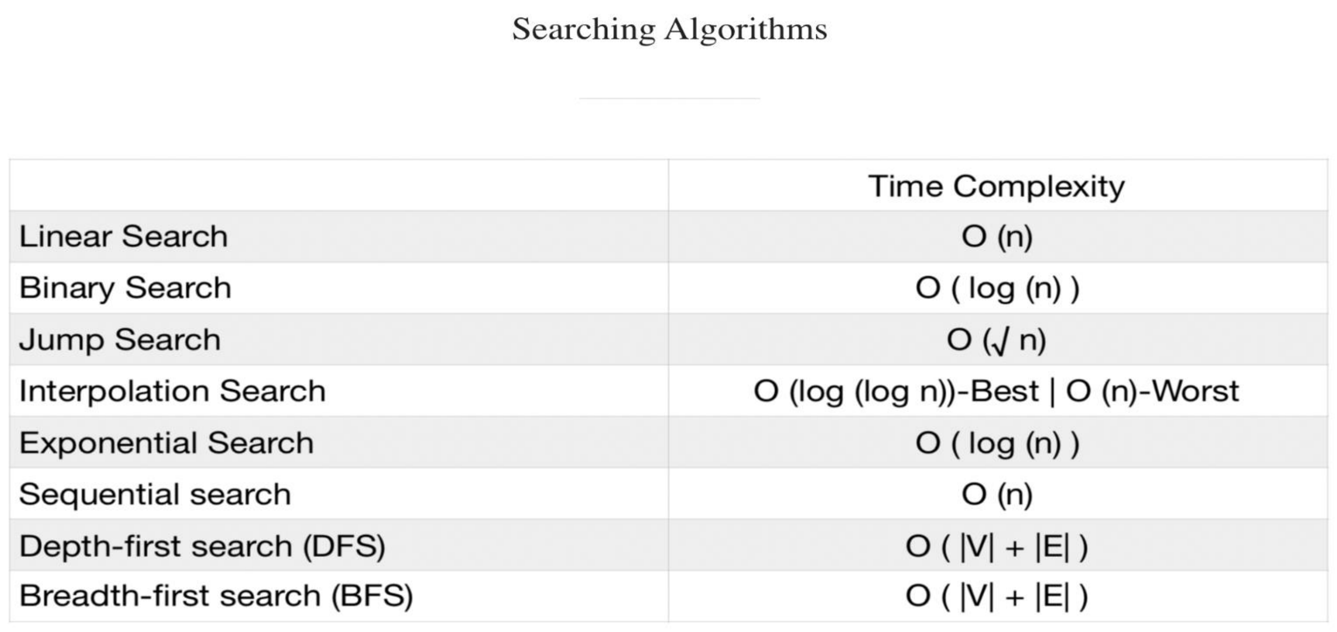
A **hashmap** is a data structure that stores **key-value** pairs, where each **key** is **associated** with a **single value**. A hashmap allows **fast access** to the values by using the **keys as indexes**. A hashmap uses a **hash function** to compute a **hash code** for each key, and then maps the hash code to an **array index**. A hashmap can **handle collisions**, which occur when two different keys have the same hash code, by using various techniques such as **chaining** or **probing**.
A hashmap is also known as a **dictionary**, **associative array**, or **hash table**. It is a common data structure that is used in many applications, such as implementing **caches**, **symbol tables**, **databases**, and **encryption algorithms**. A hashmap can also be used to **store** and **count** the **frequency** of elements in an array or a string. A hashmap can be implemented in different programming languages, such as Java, Python, C++, etc.

- HashMap doesn’t allow duplicate keys but allows duplicate values. That means A single key can’t contain more than 1 value but more than 1 key can contain a single value.
- HashMap allows a null key also but only once and multiple null values.
- This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. It is roughly similar to HashTable but is unsynchronized.

Features of Hashtable

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn’t provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

Hashmap	Hashtable
No method is synchronized.	Every method is synchronized.
Multiple threads can operate simultaneously and hence hashmap’s object is not thread-safe.	At a time only one thread is allowed to operate the Hashtable’s object. Hence it is thread-safe.
Threads are not required to wait and hence relatively performance is high.	It increases the waiting time of the thread and hence performance is low.
Null is allowed for both key and value.	Null is not allowed for both key and value. Otherwise, we will get a null pointer exception.
It is introduced in the 1.2 version.	It is introduced in the 1.0 version.
It is non-legacy.	It is a legacy.



Searching Algorithms

	Time Complexity
Linear Search	O (n)
Binary Search	O (log (n))
Jump Search	O (√ n)
Interpolation Search	O (log (log n))-Best | O (n)-Worst
Exponential Search	O (log (n))
Sequential search	O (n)
Depth-first search (DFS)	O (|V| + |E|)
Breadth-first search (BFS)	O (|V| + |E|)

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.

Linked Lists are better than arrays in the following ways:

- Array sizes are fixed at run-time and can't be modified later
- Linked Lists are not stored contiguously in the memory, as a result, they are a lot more memory efficient than arrays that are statically stored.
- there is no memory wasted because linked list is allocated memory in runtime.
- Insertion and deletion process is expensive in an array.

Heap memory is a type of memory allocation that allows programmers to **request** and **release memory dynamically** during the execution of a program. Heap memory is different from stack memory, which is allocated and deallocated automatically when a function is called and returned. **Heap memory** is also known as **dynamic memory** or **free store**.

Heap memory can be **accessed** by using **pointers**, which are variables that store the address of another variable or object. **Pointers** can **point** to any **location** in the heap **memory**, regardless of the scope or lifetime of the variable or object. This gives heap memory more flexibility and versatility, but also more complexity and risk. For example, **heap** memory can **cause** memory **leaks**, which occur when a program does not release the memory it has allocated, leading to wasted resources and performance degradation.

Heap memory is also **used** to **store objects** that are created **dynamically** in some programming languages, such as Java, Python, and C#. These languages have a feature called **garbage collection**, which is a process that automatically identifies and removes unused objects from the heap memory, freeing up space for new objects. **Garbage collection** can **improve** the **efficiency** and **reliability** of heap memory management, but it can also introduce some overhead and unpredictability.

To summarize, heap memory is a way of allocating memory on demand for variables and objects that need to have a dynamic size or lifetime. Heap memory has some advantages and disadvantages over stack memory, and it requires careful handling by the programmer or the language runtime.

Stack memory is a type of memory allocation that **stores temporary variables** and **function** calls in a **sequential** order. Stack memory is different from heap memory, which is allocated and deallocated dynamically during the execution of a program. Stack memory is also known as **static** memory or **automatic** memory.

Stack memory is implemented with a data structure called a stack, which follows the Last In First Out (**LIFO**) principle. This means that the last variable or function that is pushed onto the stack is the first one that is popped off the stack. Stack memory is **allocated** and **deallocated automatically** by the **compiler** or the **language runtime**, without any intervention from the programmer. Stack memory is also **faster** and more **efficient** than heap memory, as it does not involve any fragmentation or overhead.

Stack memory can only store variables and functions that have a fixed size and lifetime, which are determined at compile time. **Stack** memory is **limited** in size and can cause stack overflow errors if it is exhausted by too many or too large variables or functions. Stack memory can also cause **stack corruption** errors if a **variable** or function **tries** to access or **modify** the memory **outside** its **scope**.

To summarize, stack memory is a way of allocating memory for variables and functions that need to have a static size and lifetime. Stack memory has some advantages and disadvantages over heap memory, and it requires less handling by the programmer or the language runtime.

Why is Quick Sort preferred for Arrays?

- One of the main reasons for efficiency in quick sort is **locality of reference**, which makes it easy for the computer system to access memory locations that are near to each other, which is faster than memory locations scattered throughout the memory which is the case in merge sort.
- Quick sort is an **in-place sorting algorithm** i.e. it does not require any extra space, whereas Merge sort requires an additional linear space, which may be quite expensive. In merge sort, the allocation and deallocation of the **extra space increases the running time of the algorithm**.

Why is Merge Sort preferred for Linked Lists?

- In the case of linked lists, the nodes may not be present at adjacent memory locations, therefore Merge Sort is used.
- Unlike arrays, in linked lists, we can insert items in the middle in O(1) extra space and O(1) time if we are given a reference/pointer to the previous node. Therefore, we can implement the merge operation in the merge sort without using extra space.

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.

There are 3 main **approaches** to developing **algorithms**:

- **Divide and Conquer:** Involves dividing the entire problem into a number of sub-problems and then solving each of them independently.
- **Dynamic Programming:** Identical to the divide and conquer approach with the exception that all sub-problems are solved together
- **Greedy Approach:** Finds a solution by choosing the next best option.

Backtracking: Backtracking is a general algorithmic technique that systematically explores potential solutions to a problem by **incrementally** making choices, with the ability to backtrack and **undo choices** that lead to dead ends. It is often used to solve **combinatorial** problems, **optimization** problems, and **constraint satisfaction** problems.

Brute Force: Brute force is a **straightforward** and often **naive** approach to problem-solving that exhaustively considers all possible solutions to a problem, without employing any **optimization** or **heuristic** strategies. It systematically checks each candidate solution until a **valid one** is found or all possibilities have been explored. Brute force is characterized by its lack of sophistication and efficiency.

Heap is a special tree-based non-linear data structure in which the tree is a complete binary tree. It can be of two types:

- **Min-Heap:** The parent node has a key value less than its children.
- **Max-Heap:** The parent node has a key value greater than its children.

Heaps are structures meant to allow quick access to the min or the max element.

Hashing is a technique that converts any data into a **fixed-length string** of characters, called a hash or a hash value. Hashing is useful for security, encryption, authentication, and data management purposes. A hash function is a mathematical formula that takes the data as input and produces the hash as output. Different data should produce different hashes, and the same data should always produce the same hash. It should be very hard to reverse the hashing process and find the original data from the hash.

A **Trie** (pronounced "try") is a tree-like data structure used for efficient retrieval of a set of **keys** or **strings**. It is particularly well-suited for tasks involving **prefix matching** or searching for strings with **common prefixes**. Tries are commonly used in **dictionary** applications, **autocomplete** systems, **spell**-checking, **IP** routing tables, and various string-based algorithms.

The main differences between pointers and reference parameters are -

- References are used to refer an existing variable in another name whereas pointers are used to store address of variable
- References cannot have a null value assigned but pointer can.
- A reference variable can be referenced by pass by value whereas a pointer can be referenced by pass by reference.
- A reference must be initialized on declaration while it is not necessary in case of pointer.

- A reference shares the same memory address with the original variable but also takes up some space on the stack whereas a pointer has its own memory address and size on the stack

For example, in C++: **int x = 10; int& ref = x;**
For example, in C or C++: **int x = 10; int* ptr = &x;**

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

What is the **minimum** number of **queues** required to implement a **priority queue**?
The minimum number of queues required to implement a priority queue efficiently is typically two. This approach involves using two separate queues:

1. **Main Queue:** This queue stores the elements of the priority queue. It can be implemented using a standard data structure like an array, linked list, or dynamic array. Elements are enqueued (inserted) into this queue without considering their priority.
2. **Priority Queue:** This queue or data structure is used to maintain the order or priority of elements in the main queue. The priority queue contains references or pointers to elements in the main queue and ensures that the elements are dequeued (removed) from the main queue in order of their priority.

Some common **applications** of **stacks**:

1. **Function Call Management:** Stacks are used by programming languages and compilers to manage function calls and track the execution flow. Each function call's context is pushed onto the stack, allowing for proper return and cleanup when the function completes.
2. **Expression Evaluation:** Stacks are used to evaluate arithmetic expressions, especially those involving infix notation. They help convert infix expressions to postfix (or prefix) notation and then evaluate them efficiently.
3. **Parentheses Matching:** Stacks are employed to check and validate the matching of parentheses, brackets, and braces in expressions and programming code.
4. **Undo Mechanism:** Many applications, such as text editors and graphics software, use stacks to implement undo and redo functionality, allowing users to revert actions.
5. **Backtracking Algorithms:** In various algorithmic problems, stacks are used to store and manage states, making backtracking algorithms like depth-first search (DFS) possible.
6. **Memory Management:** Stacks are used in memory management to keep track of allocated and deallocated memory blocks, helping to prevent memory leaks.
7. **Expression Parsing:** Stacks assist in parsing and evaluating complex expressions, including those in programming languages, query languages, and formula calculators.
8. **Task Scheduling:** Stacks are used in scheduling algorithms to maintain a stack of tasks or processes that need to be executed.
9. **Compiler Syntax Analysis:** Stacks are used in compilers to perform syntax analysis, such as parsing the source code based on a context-free grammar.
10. **Postfix Evaluation:** Stacks are used to evaluate postfix (reverse Polish notation) expressions efficiently.
11. **Web Browsers:** Stacks can be used to implement the back and forward navigation history in web browsers.
12. **Call Stack in Debugging:** Debugging tools often use a call stack, which is essentially a stack data structure, to display the call hierarchy during program execution.
13. **Expression Conversion:** Stacks are used to convert expressions between different notations, such as infix to postfix or infix to prefix.
14. **Task Management:** Stacks are used in operating systems to manage tasks, interrupts, and context switching between processes.
15. **Resource Allocation:** In resource-constrained systems, stacks can be used to allocate and deallocate resources like memory, I/O buffers, and hardware devices.
16. **Evaluation of Logical Expressions:** Stacks are used to evaluate logical expressions, such as those found in rule-based systems and expert systems.
17. **Graph Algorithms:** Stacks are used in various graph algorithms, such as depth-first search (DFS), to explore and traverse graphs efficiently.
18. **Simulation:** Stacks are used in discrete event simulation to manage events and their scheduling.

Asymptotic Notations:

- **O notation**: asymptotic “upper bound”
- **Ω notation**: asymptotic “lower bound”
- **Θ notation**: asymptotic “tight bound”

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.

The **topological** sort algorithm takes a **directed graph** and returns an **array** of the nodes where each node appears before all the nodes it points to.

- time complexity is **O(M+N)**
- space complexity**: O(N)**

How do you check if a given graph is tree or not?

1. For an **undirected** graph, you can use either breadth-first search (**BFS**) or depth-first search (**DFS**) to traverse the graph from any vertex and mark the visited vertices. If you encounter a vertex that is **already visited**, then the graph has a **cycle** and is **not** a **tree**.
2. For a **directed** graph, you need to find the **root** of the tree, which is the vertex that has **no incoming edges**. If there is no such vertex, or there are more than one such vertices, then the **graph is not a tree**.
3. **Count** the number of **edges** in the graph. A tree with n vertices must have exactly **n - 1** edges. If the graph has more or less edges than n - 1, then it is not a tree.

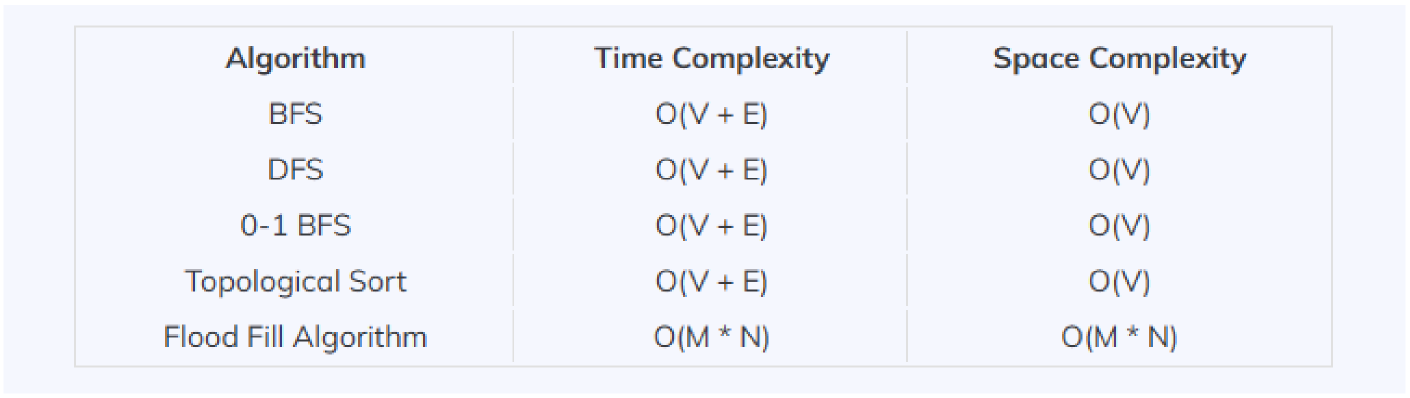
A **bipartite graph** is a type of graph in which the set of vertices can be divided into **two disjoint** sets such that no two vertices within the **same set** are **adjacent**. In other words, it's a graph in which you can color the vertices with two colors (usually referred to as "red" and "blue") such that no two adjacent vertices have the same color.

Here's a high-level algorithm to detect whether a graph is bipartite:

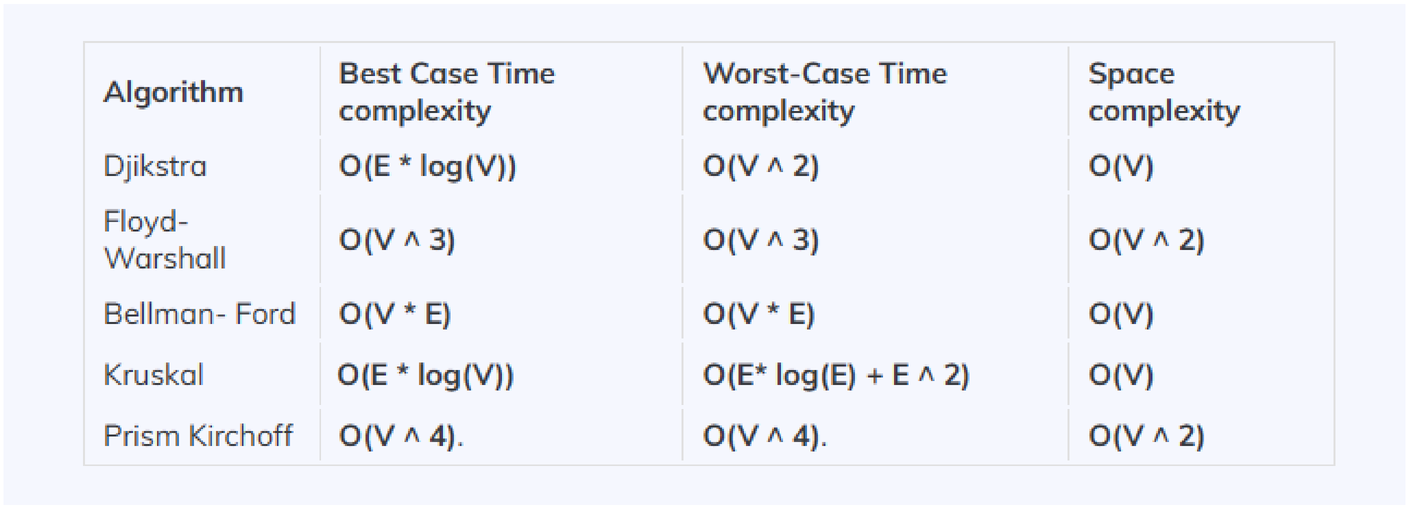
1. Choose an arbitrary vertex as the starting point.
2. Assign an initial color (e.g., "red") to the starting vertex.
3. Perform a DFS or BFS traversal of the graph, visiting each vertex:
    - When visiting a neighbor, assign the opposite color (e.g., "blue") to that neighbor.
    - If a neighbor already has the same color as the current vertex, the graph is not bipartite.
4. Continue the traversal until all vertices are visited or until a violation is detected.

Different types of Graph Algorithm, Time and Space Complexity and Use Cases

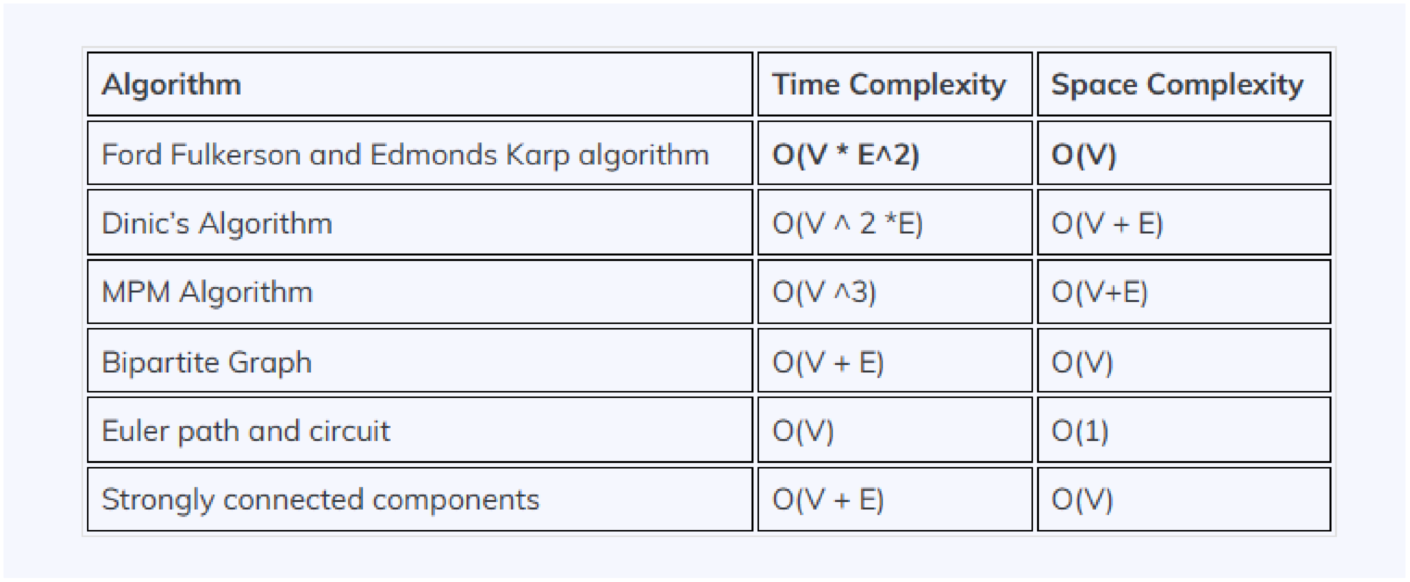
Algorithm	**Time Complexity**	**Space Complexity**	**Use Cases**
Breadth-First Search (BFS)	O(V + E)	O(V)	find the **shortest path** in an unweighted graph, **explore all vertices** at a given depth, finding **connected components** and **detecting cycles**
Depth-First Search (DFS)	O(V + E)	O(V)	used 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.



Algorithm	Time Complexity	Space Complexity
BFS	O(V + E)	O(V)
DFS	O(V + E)	O(V)
0-1 BFS	O(V + E)	O(V)
Topological Sort	O(V + E)	O(V)
Flood Fill Algorithm	O(M * N)	O(M * N)



Algorithm	Best Case Time complexity	Worst-Case Time complexity	Space complexity
Djikstra	**O(E * log(V))**	**O(V ^ 2)**	**O(V)**
Floyd-Warshall	**O(V ^ 3)**	**O(V ^ 3)**	**O(V ^ 2)**
Bellman- Ford	**O(V * E)**	**O(V * E)**	**O(V)**
Kruskal	**O(E * log(V))**	**O(E* log(E) + E ^ 2)**	**O(V)**
Prism Kirchoff	**O(V ^ 4)**.	**O(V ^ 4)**.	**O(V ^ 2)**



Algorithm	Time Complexity	Space Complexity
Ford Fulkerson and Edmonds Karp algorithm	**O(V * E^2)**	**O(V)**
Dinic's Algorithm	O(V ^ 2 *E)	O(V + E)
MPM Algorithm	O(V ^3)	O(V+E)
Bipartite Graph	O(V + E)	O(V)
Euler path and circuit	O(V)	O(1)
Strongly connected components	O(V + E)	O(V)

Some common **applications** of the **graph** data structure:

1. **Social Networks:** Graphs represent social networks like Facebook, Twitter, and LinkedIn, with users as nodes and connections (friendships, follows) as edges.
2. **Recommendation Systems:** Graphs are used to build recommendation systems by analyzing user behavior and connections to suggest products, services, or content.
3. **Transportation Networks:** Graphs model transportation systems, including road networks, airline routes, and public transportation, for route planning and optimization.

4. **Network Routing:** Graphs are used in computer networks to find optimal routes for data packets, ensuring efficient data transmission.
5. **Dependency Analysis:** Graphs represent dependencies between software components, facilitating software build and deployment systems.
6. **Web Page Link Analysis:** In web search engines, graphs represent web pages as nodes and hyperlinks as edges, helping rank and index web pages.
7. **Circuit Design:** Graphs are used in electrical engineering for circuit design, analysis, and optimization.
8. **Biology and Bioinformatics:** Graphs model protein-protein interaction networks, gene expression, and phylogenetic trees for biological research.
9. **Semantic Web:** RDF graphs represent linked data on the semantic web, enabling machines to understand and connect information.
10. **Geographical Information Systems (GIS):** Graphs model geographic data, including road networks, terrain, and infrastructure, for spatial analysis.
11. **Game Development:** Graphs represent game maps and navigation meshes for character pathfinding and decision-making in video games.
12. **Epidemiology:** Graphs model disease transmission and contact networks for epidemiological studies and disease control.
13. **Natural Language Processing (NLP):** Graphs represent syntactic and semantic relationships between words in text data, aiding in language processing tasks.
14. **Fraud Detection:** Graphs help identify fraudulent activities by modeling connections and patterns in financial transactions or social interactions.
15. **Knowledge Graphs:** Graphs represent structured knowledge with nodes representing concepts and edges representing relationships between them.
16. **Supply Chain Management:** Graphs model supply chains and logistics networks to optimize inventory, shipping, and production.
17. **Graph Databases:** Specialized databases like Neo4j and OrientDB use graphs to store and query interconnected data efficiently.
18. **Image Analysis:** In image processing, graphs represent image structures for tasks like image segmentation and object recognition.

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 ra**nge query operations** (e.g., minimum, maximum, sum) on arrays.
Binary Search on Trees	O(log n)			O(h)	used to s**earch 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.

Different types of tree, their definition and use case:		
Name	**Description**	**Use Case**
Binary Tree	A binary tree is a tree data structure where each node has at most two children: left and right.	• Binary Search Trees (BSTs) for efficient data searching and retrieval. • Expression trees for representing and evaluating mathematical expressions. • Huffman trees for data compression.
Binary Search Tree (BST)	A binary search tree is a binary tree where nodes are ordered such that for each node, all nodes in its left subtree have values less than it, and all nodes in its right subtree have values greater than it.	• Efficient searching and retrieval of data. • Implementing dictionary-like data structures. • Database indexing and symbol tables.
AVL Tree	An AVL tree is a self-balancing binary search tree where the heights of the two child subtrees of every node differ by at most one.	• Efficient data searching and retrieval while maintaining balanced height. • Real-time systems and applications where predictable performance is critical.

Red-Black Tree	A red-black tree is a self-balancing binary search tree with rules to ensure balanced height and efficient operations.	• General-purpose balanced tree for efficient searching and insertion. • Memory allocation and deallocation in many programming languages.
B-Tree	A B-tree is a self-balancing tree data structure with a variable number of children per node, designed for efficient disk storage and retrieval.	• File systems and databases for managing large datasets efficiently. • Disk-based data structures for external storage.
Trie (Prefix Tree)	A trie is a tree data structure used for storing a dynamic set of strings, where each node represents a character, and paths from the root to the leaves form words.	• Fast string searching, such as autocomplete and spell-checking. • IP routing and network protocols.
Heap	A heap is a binary tree with specific rules (min-heap or max-heap) that maintain the minimum or maximum element at the root, respectively.	• Priority queues for efficient retrieval of the highest-priority element. • Heap-based sorting algorithms like heapsort.
Suffix Tree	A suffix tree is a tree data structure that represents all the suffixes of a given string, used in pattern matching and text indexing.	• Pattern matching in DNA sequence analysis. • Search engines for full-text search.

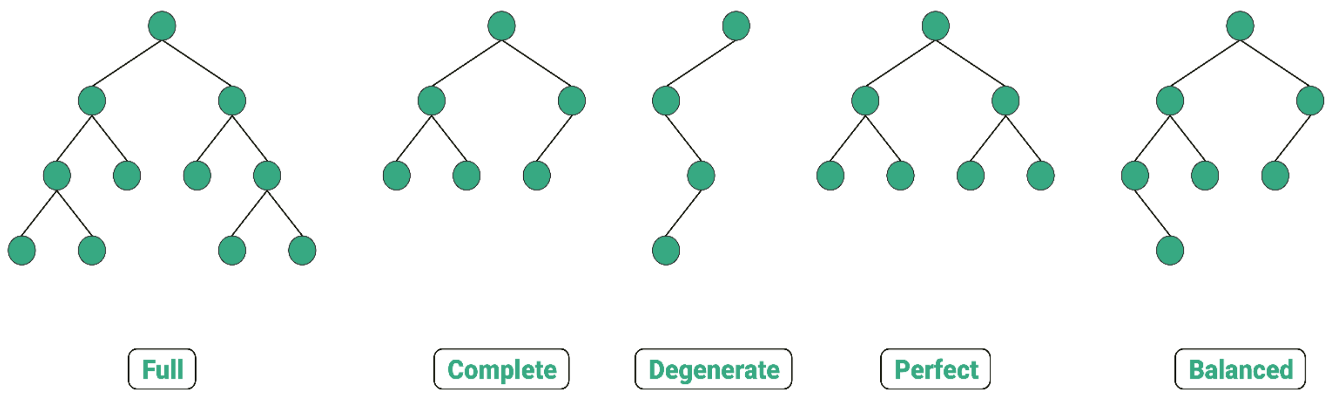


Figure: Example of **Binary Tree**

A **binary tree** is a tree in which each node has at most two children, usually called the left and right child. Some common types of binary trees are:
Full binary tree: A binary tree in which every node has either zero or two children.
Complete binary tree: A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
Degenerate or pathological binary tree: A binary tree in which every node has only one child, making it look like a linked list.

Perfect binary tree: A binary tree in which every node has two children and all leaf nodes are at the same level.

Balanced binary tree: A binary tree in which the height of the left and right subtrees of every node differ by at most one.

Print all of the leaves of a binary tree:

```
Algorithm PrintLeaves(root):
1. if root is null, return
2. if root has no left and right children (i.e., it's a leaf node):
    1. Print the value of root
3. Recursively call PrintLeaves(root.left)
4. Recursively call PrintLeaves(root.right)
```

Circuit - it is a closed path where initial vertex and end vertex are identical to each other. And any vertex can be repeated.

Path - They are the sequence of adjacent vertices that are connected by edges and have no restrictions.

Cycle - It is also a closed path where the initial vertex is identical to the closed vertex but the vertex in the path cannot be visited twice.

Tree **data** structures find **applications** in various domains, including:

- **File systems:** Trees are used to organize files and directories in a hierarchical structure.
- **Organization charts:** Trees represent the hierarchical structure of an organization, with employees and their reporting relationships.
- **Decision-making processes:** Trees are employed in decision trees and game trees to model different outcomes and choices.
- **Family trees:** Trees are used to depict genealogical relationships within families.
- **HTML/XML parsing:** Trees are utilized to represent the structure of web pages and XML documents.