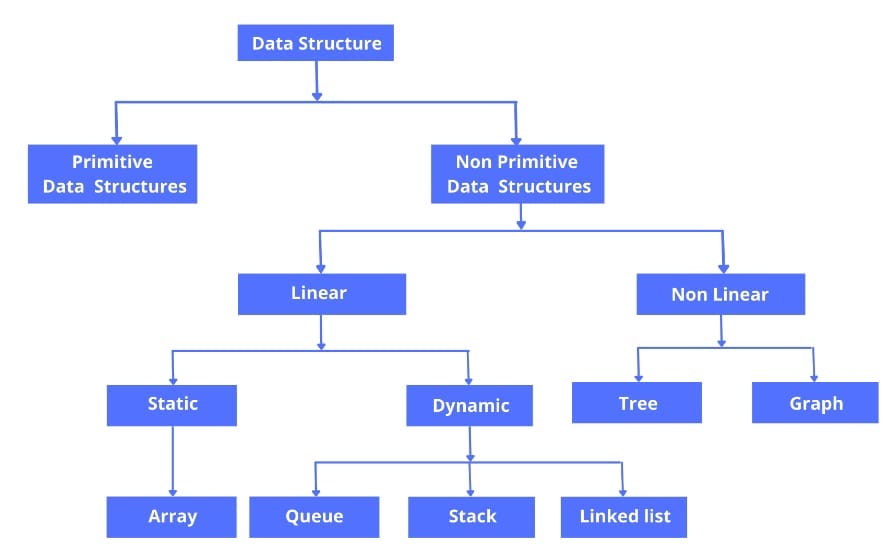
* **Data Structure**
* Data Structure is storage that is used to store and organize data.
* It is a way of arranging the data on a computer so it can be accessed and updated efficiently.
* Data Structure is not only for organizing, it is also for processing, retrieving, storing the data.



* **Linear Data Structure**
* Data Structures which the data elements are arranged in sequentially or linearly where the each data elements are attached with its previous and next adjacent elements.

|  |  |
| --- | --- |
| **Types of Linear Data Structure** | |
| **Static** | **Dynamic** |
| Static Data Structure has fixed size. | Has no fixed size. |
| Elements in static data structure is easier to access. | The elements can be updated randomly during the runtime. |
| Eg: Array | Eg: Linked List, Queue, Stack |

* **Non-Linear Data Structure**
* In Data Structures, the data elements are not arranged in a sequential or linear order.
* And all elements in a non linear data structure cannot traverse in a single run.
* Eg: Trees, Graph, Hash Table
* **What is the need of Data Structure?**
* Data Structures provides the easy ways of organizing, managing, retrieving, storing the data.’
* It requires less time.
* Saves storage space.
* Data Structure modification is easy.
* Easy access to the large database.
* Data representation is easy.
* **What is the concept of Arrays?**
* Array is a linear data structure in which the data is stored contiguous memory allocations.
* Array is index based, so identifying the elements are easier.
* It stores multiple values of same type. (But in Java Script, the arrays are more flexible, it can store any data type in an array).
* It can handle complex data structure via two-dimensional array.
* Search process in array is easy.
* **Applications of Arrays.**
* Arrays used in solving matrix problems.
* Database records are also implemented as arrays.
* It is used to implement other data structures like stack, queue, heaps, hash-tables.
* An array can be used for CPU scheduling
* **Real Life Applications of Array.**
* Array is used for computing mathematical operations.
* Used for image processing.
* Used for record management.
* Used in ordering boxes.
* **What is an Algorithm?**
* An Algorithm is the step by step process of solving a problem or performing a task.
* In data structure, the algorithms are used to organize and manipulate the data in a efficient and effective manner.
* The common algorithms in data structure are Searching Algorithms, Sorting Algorithms, Traversing Algorithms.
* The efficiency of algorithm is often measured in terms of time and space complexity.
* **Memory Allocation**
* Memory Allocation is refers to the allocating and reallocating the memory space to store a data in computer memory.
* In other words, it is the process of reserving a block of memory for the program or application.
* In data structure, the memory allocation is a critical aspects that determines the efficiency and performance of algorithms and programs.

|  |  |
| --- | --- |
| **Types of Memory Allocations** | |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| **Allocated at compile time**: The memory is reserved during the compile time. | **Allocated at runtime**: Memory is reserved during the program’s execution based on current needs. |
| **Fixed size**: The amount of memory needed must be known before the program runs, and it cannot be changed during execution. | **Not fixed size**: You can request and allocate different amounts of memory as required, allowing for more flexibility. |
| **No manual reallocation**: Memory is automatically freed when the program ends or when the variable goes out of scope. | **Manual reallocation**: You need to manually free the memory when it’s no longer needed (using functions like free() in C, or delete in C++).  (JavaScript has a built-in **garbage collector** that automatically frees up memory when objects or variables are no longer needed) |
| **Fast access**: Since memory is allocated before execution, accessing variables is faster. | **Slower access**: Since memory is allocated during execution, there is some overhead, which can make access slightly slower. |
| **Examples**: Global variables, static variables, local variables and parameters in a function. | **Examples**: Using malloc(), calloc(), or new to allocate memory in C/C++. |

* **What is Memory Leak?**
* A **memory leak** happens when a program keeps using memory that it no longer needs, but doesn't let go of it. This makes the program use more and more memory over time, which can slow it down or even cause it to crash.
* It’s when a program uses memory but doesn’t free it, even when it’s not needed anymore.
* Over time, the program takes up too much memory, making it slow or even causing it to stop working.
* **Complexity Analysis**
* Complexity Analysis is the process of analyzing the performance of algorithms & data structures in terms of time and space complexities.
* It helps to understand how much time and memory space taken for an algorithm or data structure.

| **Data Structure** | **Access** | **Search** | **Insertion** | **Deletion** | **Space Complexity** |
| --- | --- | --- | --- | --- | --- |
| **Array** | O(1) | O(n) | O(n) | O(n) | O(n) |
| **Linked List** | O(n) | O(n) | O(1) | O(1) | O(n) |
| **Doubly Linked List** | O(n) | O(n) | O(1) | O(1) | O(n) |
| **Stack (Array-based)** | O(1) | O(n) | O(1) | O(1) | O(n) |
| **Stack (Linked List)** | O(n) | O(n) | O(1) | O(1) | O(n) |
| **Queue (Array-based)** | O(1) | O(n) | O(1) | O(1) | O(n) |
| **Queue (Linked List)** | O(n) | O(n) | O(1) | O(1) | O(n) |
| **Hash Table** | O(1) | O(1) | O(1) | O(1) | O(n) |
| **Binary Search Tree (BST)** | O(log n) | O(log n) | O(log n) | O(log n) | O(n) |
| **AVL Tree** | O(log n) | O(log n) | O(log n) | O(log n) | O(n) |
| **Heap (Min/Max Heap)** | O(log n) | O(n) | O(log n) | O(log n) | O(n) |
| **Graph (Adjacency List)** | O(V + E) | O(V + E) | O(1) | O(1) | O(V + E) |
| **Graph (Adjacency Matrix)** | O(1) | O(V) | O(1) | O(1) | O(V²) |

· **Arrays** are fast for accessing elements but slow for insertions and deletions.

· **Linked Lists** are great for insertions and deletions but slower for access.

· **Hash Tables** offer fast lookups but can degrade in performance with too many collisions.

· **Trees and Heaps** provide a good balance of performance for various operations.

· **Graphs** can be represented in different ways depending on the use case (dense vs sparse graphs).

* **Asymptotic Analysis**
* **Asymptotic analysis** is a method used to describe the performance of an algorithm as the size of the input grows. It helps us understand how the running time or space required by an algorithm changes when we work with large inputs.
* It helps you compare different algorithms and figure out which one will be faster or use less memory, especially for large inputs.
* You can predict how an algorithm will perform without having to run it for every possible input size.
* **Asymptotic Notation**
* This is the mathematical notation used to describe the time or space complexity derived from the asymptotic analysis.
* Common Types

· **Big O (O)**: Worst-case scenario, upper bound.

· **Big Omega (Ω)**: Best-case scenario, lower bound.

· **Big Theta (Θ)**: Average or tight bound.

* **Linked List**
* Linked List is a linear data structure in which each item is connected to the next one in the sequence.
* Each piece of data is stored in a **no**d**e**.
* A node has two parts:

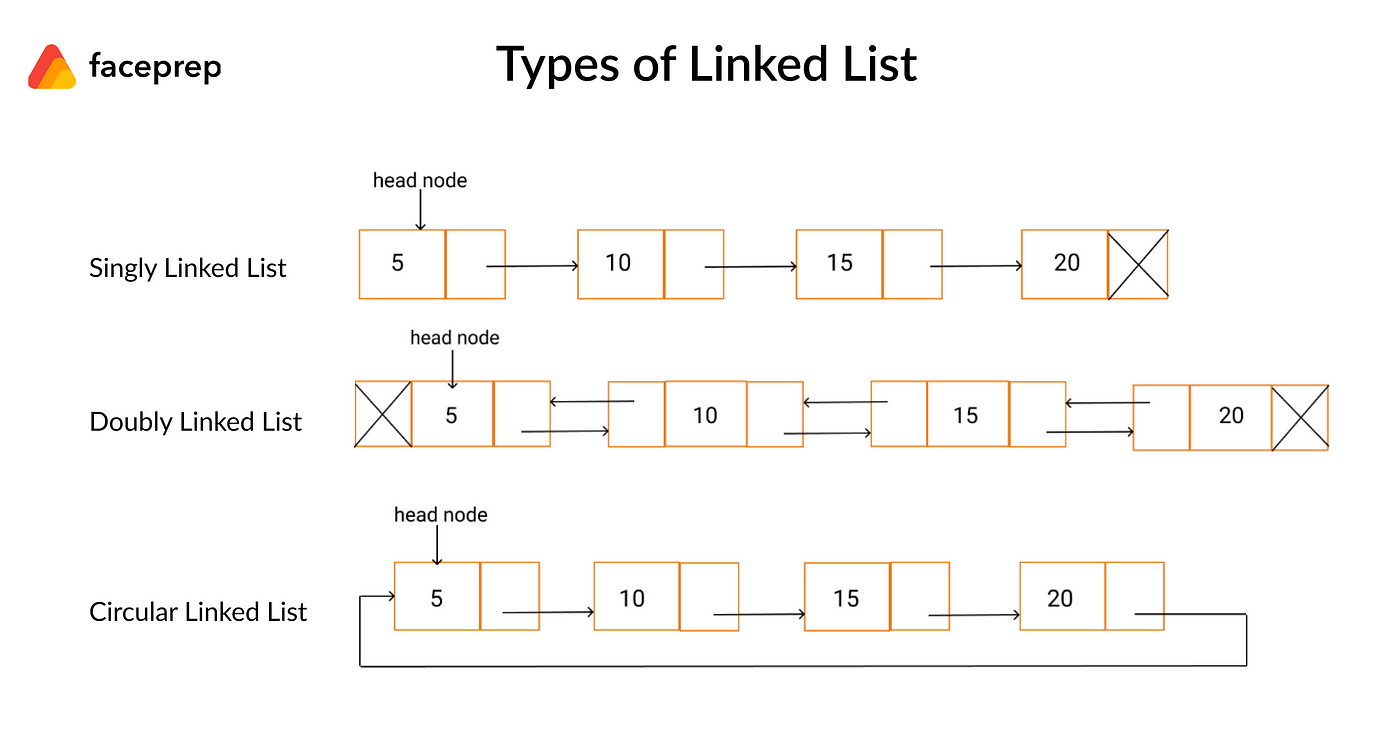
· **Data**: The actual information.

· **Next**: A pointer to the next node in the list.

* Each node points to the next node, and the last node points to nothing (null).

### **Types of Linked Lists:**

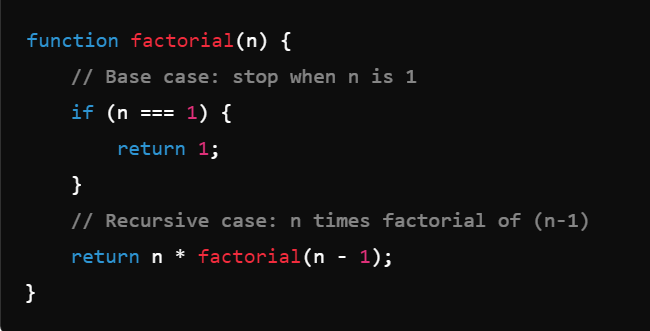
1. **Singly Linked List**: Each node points to the next node, forming a one-way chain.
2. **Doubly Linked List**: Each node points to both the next and the previous node, allowing traversal in both directions.
3. **Circular Linked List**: The last node points back to the first node, creating a circular structure.



* **Linked List Advantages:**
* **Dynamic Size:** Unlike arrays, which have a fixed size, linked lists can grow or shrink as needed during program execution.
* **Efficient Insertions/Deletions**: Adding or removing elements in a linked list is more efficient, especially at the beginning or middle. In an array, you have to shift other elements around, but in a linked list, you only need to update the pointers (links).
* **No Need for Contiguous Memory**: In arrays, all elements must be stored together in memory, which can be a problem when memory is fragmented. Linked lists store elements in different locations, connected by pointers, making them ideal when you can't get a large block of continuous memory.
* **Linked List Drawbacks:**
* **No Random Access**: Linked lists don't support direct (random) access to elements. To access a specific element, you need to traverse the list from the beginning, which makes accessing elements slower than in an array.
* **More Memory Usage**: Each node in a linked list requires extra memory for storing a pointer (or two in the case of doubly linked lists), leading to higher memory consumption compared to arrays.
* **Applications of Linked List:**
* Implementation of Stacks and Queues
* Graph Representation
* Browser's Back and Forward Buttons
* Undo/Redo Functionality in Applications
* Music and Video Playlists
* Implementation of Hash Tables. Each buckets of hash table can be a linked list
* **Recursive Function**
* Recursion is a technique where a problem is divided into smaller, more manageable sub-problems that are similar in structure to the original problem.
* Recursive Function is a function that calls itself in order to solve smaller instances of the same problem.

### How Recursive Functions Work:

1. The function keeps calling itself with a simpler version of the original input.
2. When the base case is met, the function stops calling itself.
3. The intermediate results are returned and combined as the recursion "unwinds."



### Pros of Recursion:

* Simplifies complex problems by breaking them into smaller sub-problems.
* Some problems, like tree traversal and the Tower of Hanoi, are naturally recursive and easier to solve this way.

### Cons of Recursion:

* It can be less efficient in terms of memory and processing time because it uses the call stack to store function calls.
* Recursion can lead to a **stack overflow** if the base case isn't reached or if the input is too large.

### When to Use Recursion:

* When the problem can naturally be divided into similar sub-problems.
* When iterative solutions are harder to understand or implement.
* **Binary Search**
* **Binary Search** is an efficient algorithm used to find an item in a **sorted list**.
* It works by splitting the list in half over and over. Each time, it ignores the half where the item can't be, and keeps searching in the other half. This makes it quicker than checking every item one by one.
* It's much faster than linear search for large sorted lists, which takes O(n) time.
* Binary search works only on **sorted data**.

### **How Binary Search works:**

1. Look at the **middle** item in the sorted list.
2. If the middle item is what you're looking for, you're done.
3. If the middle item is **bigger** than what you're looking for, focus on the **left half** of the list.
4. If the middle item is **smaller** than what you're looking for, focus on the **right half** of the list.
5. Keep repeating this process until you find the item or there’s no more left to check.

### Example of Binary Search:

Suppose you have a sorted list: **[10, 20, 30, 40, 50, 60, 70]** and you want to find the number **30**.

1. Check the middle item: **40**. since **30** is smaller than **40**, search the left half **[10, 20, 30]**
2. Check the middle item: 20. since **30** is greater than **20**, search the right half **[30]**
3. Check the middle item: 30. It matches the target, so the search is complete.

If the target was **25**:

1. Check the middle item: **40**. Since **25** is smaller than **40**, search the left half **[10, 20, 30]**.
2. Now, the middle of **[10, 20, 30]** is **20**. Since **25** is larger than **20**, search the right half **[30]**.
3. The middle of **[30]** is **30**, but since there's no more elements left, the search ends, and the target isn't found.

* **Linear Search**
* **Linear search** is a simple searching algorithm that checks each element of a list or array sequentially until the desired element (target) is found or the list ends.
* It is one of the most straightforward search methods and does not require the data to be sorted.
* Example: search from array or linked list.
* **How Linear Search Works:**
* Start at the first element of the list.
* Compare the target value with the current element.
* If the current element matches the target, return its position (or true, if you're only checking for existence).
* If not, move to the next element and repeat the process.
* If the end of the list is reached and the target is not found, return a failure indication (e.g., -1 or false).
* **Jagged Array & Nested Array**
* Jagged Array is
* **Sorting**
* **Bubble Sort:**
* Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is repeated until the list is sorted.
* Start at the beginning of the list
* **Compare each pair of adjacent elements**: If the element on the left is greater than the element on the right, **swap them**.
* **Move to the next pair of elements** and repeat the comparison and swapping process until you reach the end of the list.
* **Time Complexity**: O(n2) for average and worst cases, O(n) for best case (when the list is already sorted).
* **Space Complexity**: O(1) since it’s an in-place sorting algorithm.
* **Stability**: Bubble Sort is stable because it doesn’t change the relative order of equal elements.
* Bubble Sort is generally inefficient for large datasets but is sometimes used for small or nearly sorted arrays because of its simplicity.
* **Insertion Sort:**
* Insertion Sort is a straightforward sorting algorithm that builds the final sorted array one item at a time. It is similar to how people sort playing cards in their hands: you take each card and place it in the correct position among the previously sorted cards.
* Start from the second element (as the current element)
* Compare the current element with the all the previous elements.
* If the current element is smaller, shift the larger elements one position to the right to make space.
* **Insert the current element into its correct position**.And continues the process until all the elements are placed inserted correct position.
* **Time Complexity**: O(n2) in the average and worst cases, but O(n) in the best case (when the array is already sorted).
* **Space Complexity**: O(1) as it’s an in-place sorting algorithm.
* **Stability**: Insertion Sort is stable, preserving the relative order of equal elements.
* Insertion Sort is efficient for small or nearly sorted arrays, making it useful in practice for such cases.
* **Selection Sort:**
* Selection Sort is a simple comparison-based sorting algorithm that works by repeatedly finding the minimum (or maximum) element from the unsorted part of the list and moving it to the sorted part.
* **Divide the list into a sorted and an unsorted part**.
* **Find the minimum element** in the unsorted part of the list.
* **Swap** the minimum element with the first element in the unsorted part.
* Move the boundary between the sorted and unsorted parts one element to the right.
* **Time Complexity**: O(n2) for all cases (best, average, and worst), as it always performs n2 comparisons.
* **Space Complexity**: O(1), since it sorts in place.
* **Stability**: Selection Sort is **not stable** because it may change the relative order of equal elements.
* **Quick Sort:**
* Quick Sort is a fast way to sort a list of numbers or items by using a "pivot" to split the list into smaller parts.
* **Pick a pivot**: Choose one item from the list to be the "pivot." This can be any item, like the last one in the list.
* **Split the list**: Put all items smaller than the pivot on its left and all items larger than the pivot on its right.
* **Sort each part**: Repeat the same steps on the left and right parts (pick a pivot, split, and sort) until each part is small (only one item). When that happens, the list is sorted.

###### Advantages Quick Sort

* **Fast**: For most lists, it can sort in a short amount of time.
* **Efficient**: It doesn’t need extra memory, so it’s good for large lists.

### Drawback of Quick Sort

* If the list is already nearly sorted, Quick Sort can be slow unless the pivot is chosen carefully.
* **Merge Sort:**
* Merge Sort is a sorting algorithm using "divide and conquer" method. It breaks the list into smaller parts, sorts them, and then combines (merges) them back together in order.

### How Merge Sort Works

1. **Divide**: Keep splitting the list into two halves until each part has only one item (or no items). A single-item list is considered "sorted" because there's nothing to compare it with.
2. **Merge**: Start combining (merging) each of the small parts back together in the correct order. While merging, keep the smallest items first so the list stays sorted as you put it back together.

· **Base Case**: If the list has 1 or 0 items, it's already sorted, so just return it.

· **Divide**: The list is split into left and right halves.

· **Merge**: We merge the two sorted halves by comparing the items from each half and placing them in order in the result list.

· **Time Complexity**: Always O(n log n), which means it is efficient for large lists.

· **Space Complexity**: Uses extra space O(n) for the temporary arrays.

· **Stability**: Merge Sort is stable, which means it keeps the order of equal elements the same.

* **Heap Sort**
* **Stack Overflow**
* Stack Overflow is
* **Stack**
* A **Stack** is a linear data structure that stores items in a **Last-In, First-Out (LIFO)** order. This means that the last item added to the stack will be the first one to be removed, like a stack of books where you add books on top and can only take the top one off first.

### Basic Operations in a Stack

1. **Push**: Adds an item to the top of the stack.
2. **Pop**: Removes the item from the top of the stack.
3. **Peek** (or Top): Returns the item at the top without removing it.
4. **isEmpty**: Checks if the stack is empty.

### How Stacks Are Used in Real Life

* **Undo Functionality**: In applications like text editors, stacks store actions to undo the last one performed.
* **Browser History**: Browsers use stacks to store pages, allowing users to go back to the previous page.
* **Function Calls**: Programming languages often use a call stack to keep track of function calls, allowing functions to return to their caller in reverse order.

### Different Types of Stacks

* **Linear Stack (Simple Stack)**
* **Dynamic Stack**
* **Array Based Stack**
* **Linked List Based Stack**
* **Double Ended Stack (Deque Stack)**
* **Multiple Stacks**
* **Priority Stack**
* **Concurrent Stack**
* **Blocking Stack**
* **Non-Blocking Stack**
* **Min Stack / Max Stack**
* **Circular Stack**
* **Queue**
* A **Queue** is a linear data structure that organizes items in a **First-In, First-Out (FIFO)** order.
* This means that the first item added to the queue will be the first one to be removed, much like a line of people waiting for service: the first person in line is the first to be served.

### Basic Operations in a Queue

1. **Enqueue**: Adds an item to the end of the queue.
2. **Dequeue**: Removes the item from the front of the queue.
3. **Peek** (or Front): Returns the item at the front without removing it.
4. **isEmpty**: Checks if the queue is empty.

### How Queues Are Used in Real Life

* **Customer Service**: Queues represent a line where people wait for service, such as in a bank or a customer service call center.
* **Print Queue**: In computers, print jobs are processed in the order they were sent to the printer.
* **Task Scheduling**: Operating systems use queues to manage tasks waiting for CPU time.

### Different Types of Queues

* **Linear Queue (Simple Queue)**
* **Circular Queue**
* **Circular Dequeue**
* **Double Ended Queue (Dequeue)**
* **Double Ended Priority Queue (DEPQ)**
* **Priority Queue**
* **Concurrent Queue**
* **Blocking Queue**
* **Non-Blocking Queue**
* **Input-Restricted and Output-Restricted Queues**
* **Hash Table**
* A **Hashtable** (also called a **Hash Map**) is a data structure that stores key-value pairs.
* It uses a hash function to compute an index (or hash) into an array of buckets, where the desired value can be stored or found.
* Hashtables allow for very fast data retrieval because, ideally, each key is directly mapped to a specific location in memory.

### How a Hashtable Works

1. **Hash Function**: A function that takes a key and converts it into a hash code, which is an integer. This hash code determines where in the array the key-value pair will be stored. For example, a key like "name" might be converted into an index like 5.
2. **Buckets**: Each position in the array is called a "bucket," which can hold one or more key-value pairs. Ideally, each bucket holds only one pair, but sometimes, due to **collisions**, multiple pairs might end up in the same bucket.
3. **Collision Handling**: When two keys hash to the same bucket, there’s a **collision**. Hashtables use various methods to handle this, such as:
4. **Load Factor**: Measures how full the hashtable is. High load factors (like over 0.75) often trigger re-sizing of the table to reduce collisions.

### Basic Operations in a Hashtable

1. **Set (Insert)** : Store a key-value pair.
2. **Get (Search)** : Retrieve the value associated with a key.
3. **Remove (Delete)** : Remove a key-value pair.

### Advantages of Hashtables

* **Fast Lookups**: Access time is approximately constant (O(1)) because the hash function directly maps keys to indices.
* **Efficient for Large Datasets**: Hashtables are efficient when the number of items is large and keys need to be mapped uniquely.

### Disadvantages of Hashtables

* **Collisions**: When multiple keys hash to the same index, collision handling is needed, which can slow down operations.
* **Memory Usage**: Hashtables may consume more memory than other data structures due to empty or sparsely populated buckets.
* **Poor Performance in Worst Case**: If many collisions occur or if the hash function is poorly designed, the performance can degrade.

### Real-World Uses of Hashtables

* **Dictionaries**: Used in dictionaries to store word definitions (key-value pairs).
* **Database Indexing**: Used to quickly locate data records in databases.
* **Caches**: Used in caching mechanisms where items are stored and retrieved based on keys.

### Applications of Hash Table

* **Database Indexing**: Used for indexing databases to enable quick data retrieval.
* **Cryptography**: For secure data hashing in encryption, digital signatures, and password storage.
* **Networking**: Efficiently map IP addresses to MAC addresses. Resolving domain names to IP addresses (**DNS Lookup**).
* **Search Engines**: Store and quickly retrieve search terms and associated documents. Map partial queries to possible completions.
* **Collision :**
* In a hash table, collision happens when two different keys hash to the same index or location in the table. This means both keys want to be stored in the same spot, which creates a conflict.
* Collision handling methods are used in such cases to avoid the conflicts.
* Collision Handling Methods:
* **Chaining (Separate Chaining)**: Each bucket holds an array or linked list of items, so multiple items can exist in the same bucket.
* **Open Addressing (Linear Probing)**: When a collision occurs, the hashtable finds the next available slot. All elements are stored directly within the hash table (no need for array or linked list).
* **Double Hashing**
* **Quadratic Probing**
* **Cuckoo Hashing**
* **Rehashing**
* **Load Factor :**
* The load factor in a hash table is a measure of how full the table is. It is calculated using this formula:

**Load Factor = Number of Elements in the Table / Total Size of the Table**

* Example: If a hash table can hold 10 elements (total size) and currently has 7 elements stored, the load factor is: **7/10 = 0.7**
* Why is it important?
* - A high load factor (e.g., close to 1) means the table is nearly full, increasing the chances of collisions.
* - A low load factor (e.g., 0.2) means there is plenty of space, reducing collisions but wasting memory.
* - Hash tables often resize (rehash) when the load factor exceeds a threshold, like 0.7 or 0.75, to keep performance optimal.

- double hashing

- linear probing vs quadratic probing

- SHA1, popular hashing algorithms

- hashing vs encryption

- priority queue

- double ended queue

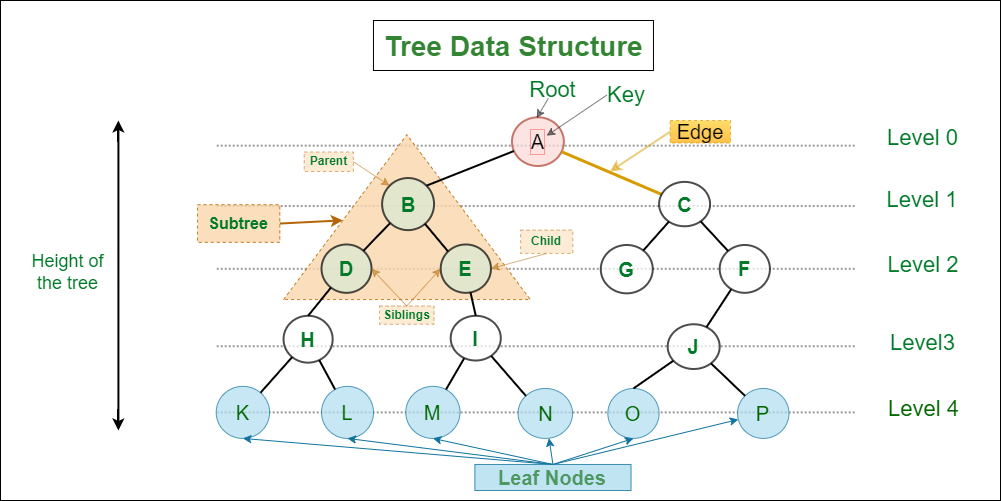
- circular queue

- attempt to implement hashtable without using arrays and using another hashing algorithm

* **Tree**
* **Tree is a hierarchical (non-linear) data structure that consists of nodes connected by edges, forming a parent-child relationship.**
* **One Root: A tree has exactly one root node.**
* **No cycles:** **A tree will not contain any loops or cycles.**
* **It is very quicker to search / access the data in tree when compared to the linear data structure like arrays, linkedlist etc..**

### **Key Components of a Tree**

* **Root:**
  + The topmost node of the tree.
  + It serves as the starting point for traversing the tree.
* **Node:**
  + Represents a single element in the tree.
  + Contains data and links to other nodes.
* **Edge (Connection):**
  + The connection between two nodes.
* **Child:**
  + A node that is directly connected to another node when moving away from the root.
* **Parent:**
  + A node that has one or more children.
* **Leaf:**
  + A node with no children.
* **Subtree:**
  + A smaller tree within the larger tree, starting at any node.
* **Height:**
  + The longest path from the root to a leaf.
* **Depth:**
  + The length of the path from the root to a particular node.
* **Level:**
  + All nodes at the same distance from the root.



* **Types of Trees**

1. **Binary Tree:**

* Each node has at most two children (left and right).

1. **Binary Search Tree (BST):**

* A binary tree where the left child contains values less than the parent node, and the right child contains values greater than the parent node.

1. **AVL Tree:**

* A self-balancing binary search tree.

1. **B-Tree:**

* Used for database and file systems, with more than two children per node.

1. **Heap:**

* A special tree used in priority queues and heap-sort.

1. **Trie:**

* Used to store strings, like in auto-complete systems.

1. **N-ary Tree:**

* A tree where each node can have at most n children.
* **Usage of Trees**
* **File System - for directory structure**
* **A Family Tree**
* **An Organization Tree**
* **DOM**
* **Chat Bots**
* **Abstract Syntax Trees**
* **Binary Tree**
* **Binary Tree is a hierarchical data structure where **each node has at most two child nodes**. These children are referred to as the left child and right child.**
* **Types of Binary Trees**

1. **Full Binary Tree:**

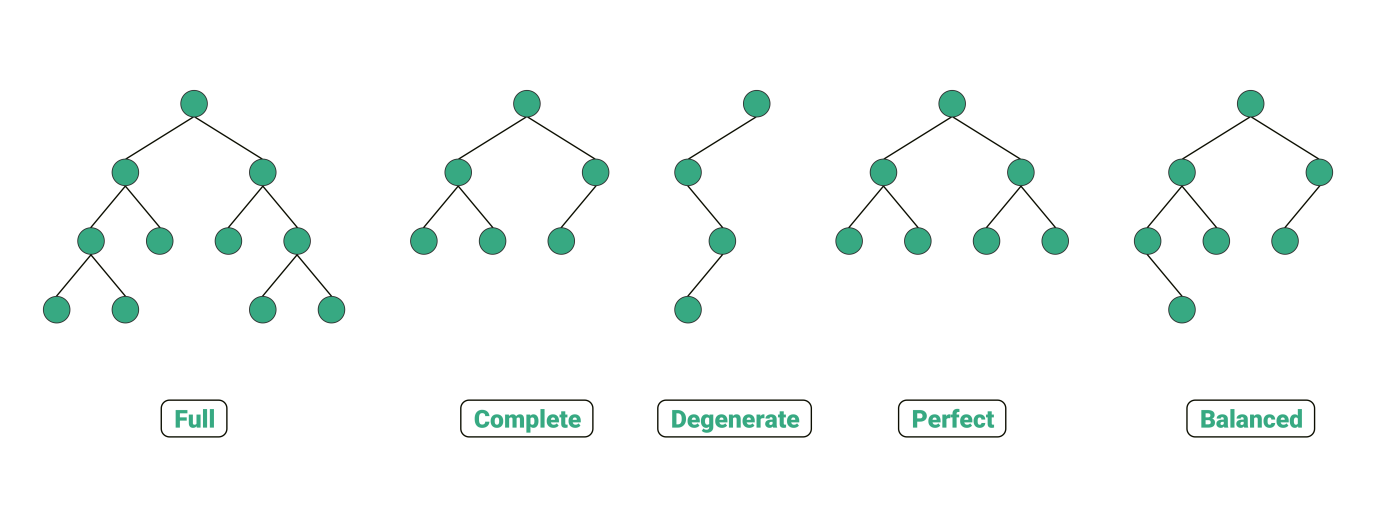
* Every node has either 0 or 2 children.

1. **Complete Binary Tree:**

* All levels, except possibly the last, are fully filled.
* The last level has nodes as far left as possible.

1. **Perfect Binary Tree:**

* All internal nodes have exactly two children, and all leaf nodes are at the same level.

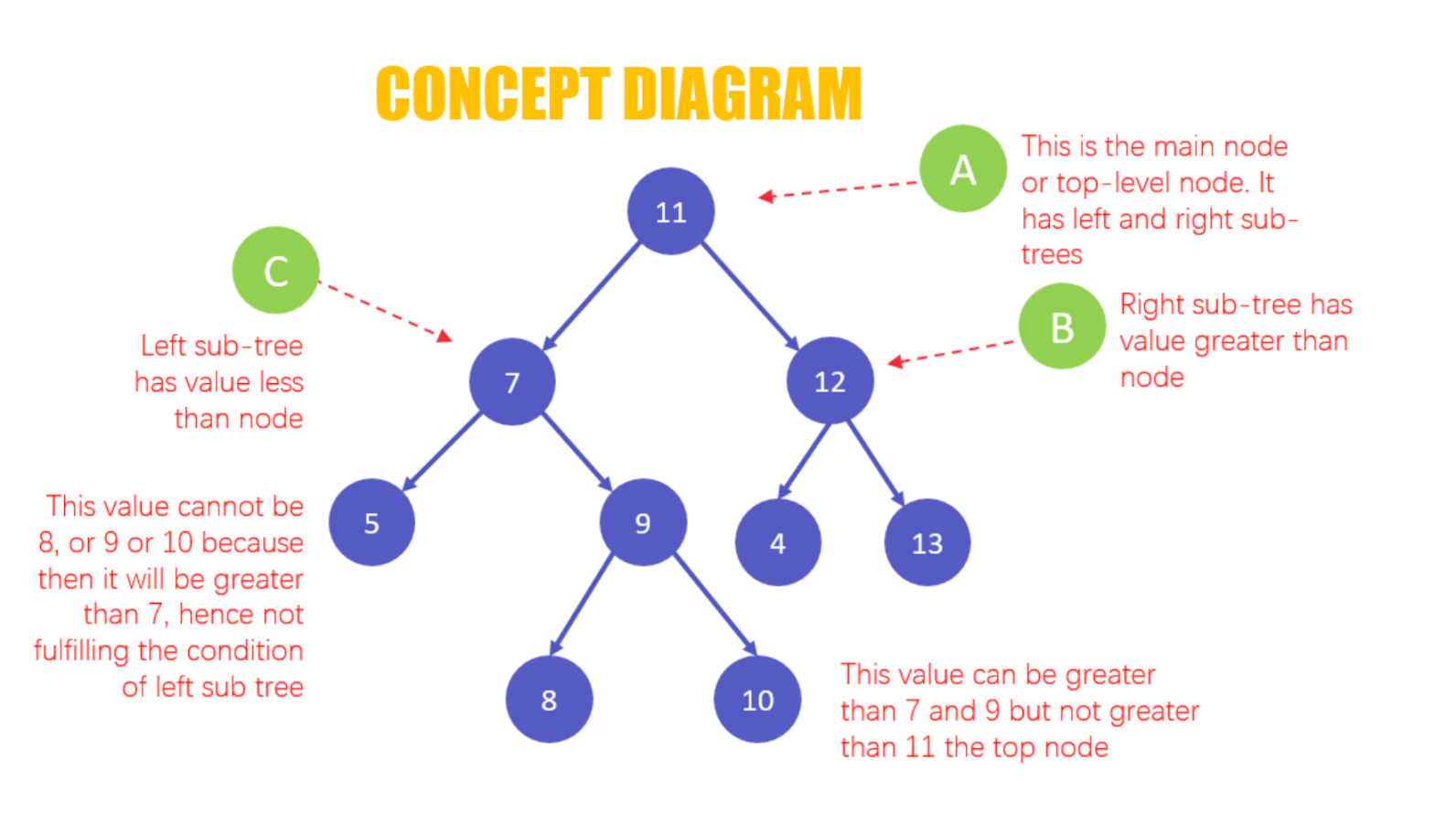


* **Binary Search Tree (BST)**
* **Binary Tree is a specialized binary tree where each nodes are arranged in specific order to enable efficient searching, insertion and deletion.**
* **Key Characteristics of a BST**

1. Each node has at most **two children**.
2. The **left child** of a node contains values **less than** the parent node's value.
3. The **right child** of a node contains values **greater than** the parent node's value.
4. No duplicate values are typically allowed (but this can vary depending on implementation).

#### ****Advantages of a BST****

* Searching, inserting, and deleting operations are efficient, with an average time complexity of **O(log n)** for balanced BSTs.
* Enables **sorted order traversal** using in-order traversal.



### ****Differences Between Binary Tree & Binary Search Tree****

| **Aspect** | **Binary Tree** | **Binary Search Tree (BST)** |
| --- | --- | --- |
| **Node Arrangement** | No specific order among nodes. | Nodes are arranged based on their values. |
| **Efficiency** | Operations like search are less efficient (O(n) in the worst case). | Searching, insertion, and deletion are faster O(log n) for balanced BSTs). |
| **Traversal** | Can be traversed in any order (pre-order, in-order, post-order). | In-order traversal gives nodes in sorted order. |
| **Use Case** | Suitable for representing hierarchical data. | Ideal for applications requiring quick lookup, insertion, and deletion. |

* **TRAVERSE in Binary Search Tree**

1. **Depth-First Traversal (DFS)**

* **The DFS algorithm starts from at the root node and explores as far as possible along each branch before backtracking.**
* **Visit the root node, visit all the nodes in the left sub-tree, and visit all the nodes in the right sub-tree.**
* **There are 3 types of DFS methods (In-Order, Pre-Order, Post-Order).**
* ****In-order Traversal (Left → Root → Right)**:**

· **Order**: Left sub-tree, Root node, Right sub-tree.

· **Use Case**: Retrieves elements in **sorted order** for a BST.

* **Pre-order Traversal (Root → Left → Right)**:

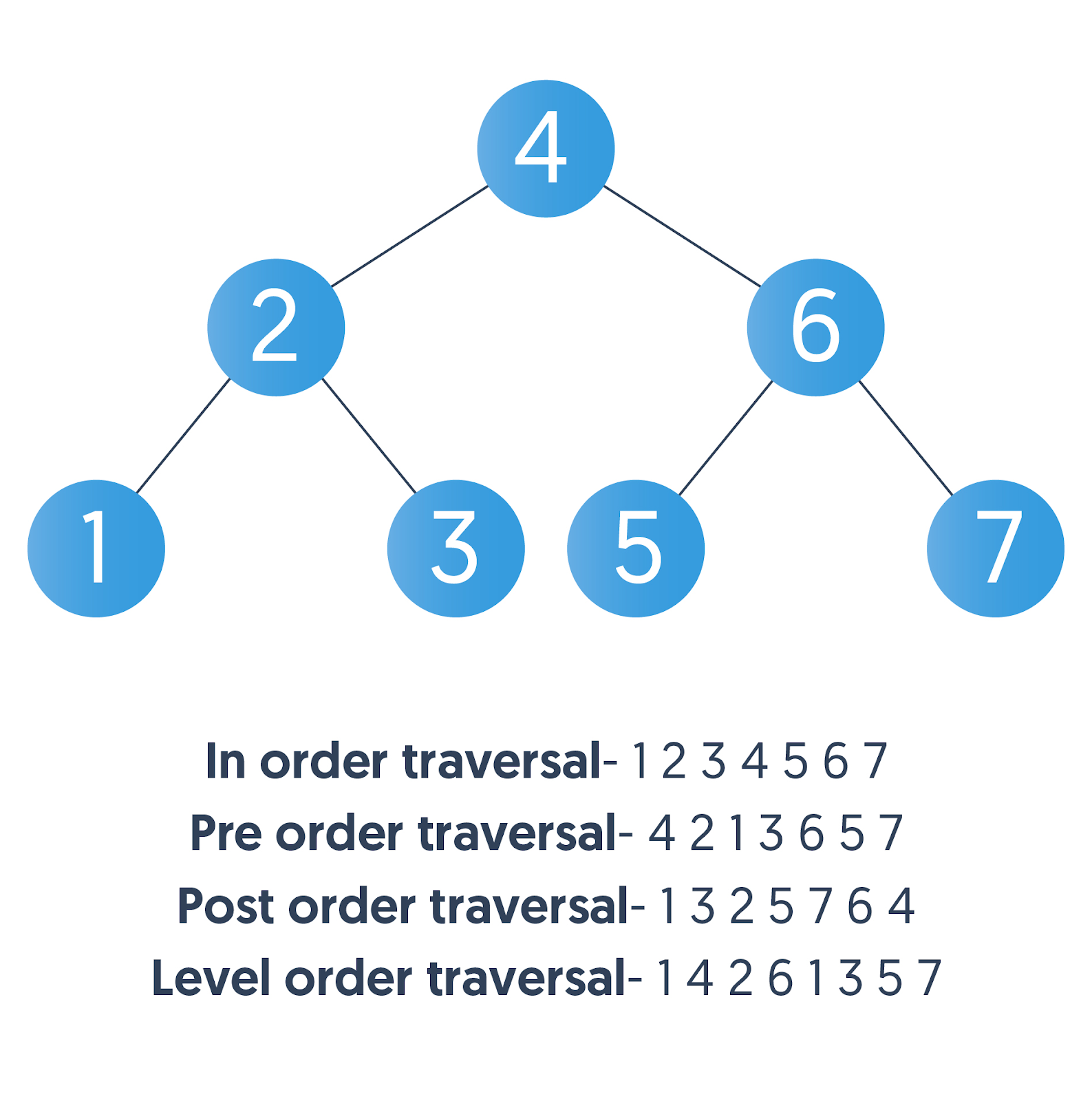
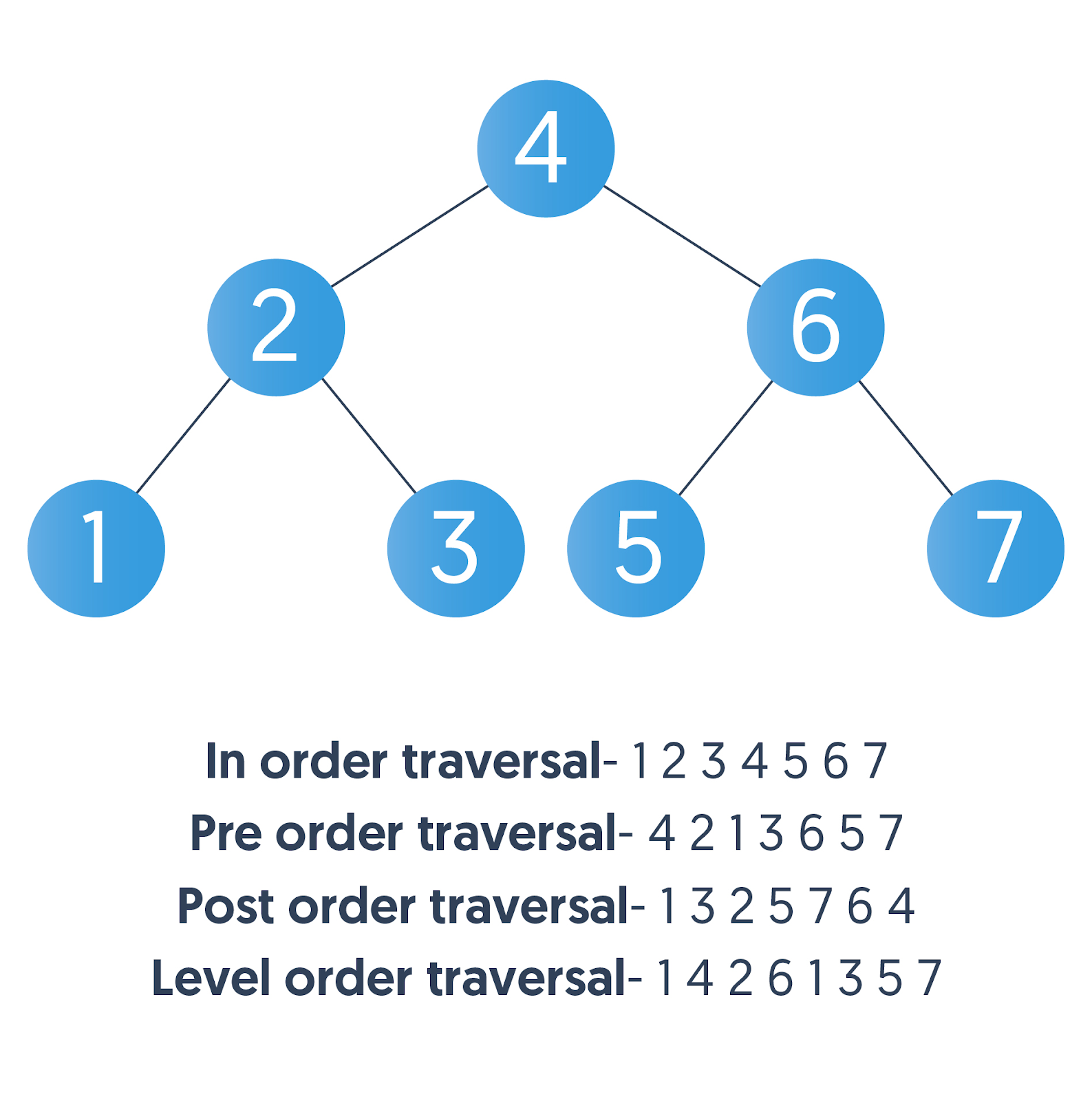
· **Order**: Root node, Left sub-tree, Right sub-tree.

· **Use Case**: Useful for **creating a copy** of the tree or serializing it.

* **Post-order Traversal (Left → Right → Root)**:

· **Order**: Left sub-tree, Right sub-tree, Root node.

· **Use Case**: Useful for **deleting the tree nodes** or evaluating expressions in expression trees.



1. **Breadth-First Traversal (Level Order)**

* **BFS explores all the nodes at present depth and moving on to the nodes at next depth level. (Level by Level from top to down)**

· **Order**: Visit nodes **level by level** (Top → Down). Starting from the Root and moving down each level from left to right.

· **Use Case**: Useful for **shortest path problems** and **visualizing** the tree structure

· **Implementation**: Uses a **queue**.

