Learn Data Structures and Algorithms

Table of Contents

Introduction

What is a Data Structure?

Data structure: a named location that can be used to store and organize data.

Example:

- array: a collection of elements stored at contiguous memory locations.

  ```
  char[] array = new char[10];
  ```

What is an Algorithm?

Algorithm: a step-by-step procedure to solve a problem.

Example: Baking a pizza.

1. Heat the oven to 210°C.
2. Place the pizza in the oven.
3. Bake for 15 minutes.

Example: Linear Search Algorithm.

- One by one, examine the elements of an array to find a value.

  ```
   int[] array = {1, 2, 3, 4, 5};
  int value = 3;
  for (int i = 0; i < array.length; i++) {
      if (array[i] == value) {
          return i;
      }
  }
  return -1;
  ```

Why Learn Data Structures and Algorithms?

1. You will write code that is both time and memory efficient.
2. Commonly asked in technical interviews.

Basic Data Structures

Stacks

A **Stack** is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. Stacks are often compared to a stack of plates where you can only add or remove the top plate.

Core Operations

1. **Push**: Adds an element to the top of the stack.
2. **Pop**: Removes the element from the top of the stack and returns it.
3. **Peek (or Top)**: Returns the element at the top of the stack without removing it.
4. **isEmpty**: Checks whether the stack is empty.
5. **Size**: Returns the number of elements in the stack.

Characteristics of Stacks

- **LIFO (Last In, First Out)**: The last element added to the stack is the first one to be removed.
- **No Random Access**: Unlike arrays, you cannot access elements in the middle of a stack directly. You must remove elements from the top one by one until you reach the desired element.
- **Dynamic or Static Size**: Stacks can be implemented using arrays (fixed size) or linked lists (dynamic size).

Applications of Stacks

- **Expression Evaluation**: Stacks are widely used in evaluating arithmetic expressions and parsing expressions in compilers (e.g., converting infix to postfix notation).
- **Undo Mechanism**: Most applications with an undo feature, like text editors, use stacks to keep track of changes.
- **Function Call Management**: Stacks manage function calls in programming languages. The function call stack helps in keeping track of return addresses and local variables.
- **Balanced Parentheses Checking**: Stacks are used to check whether an expression has balanced parentheses, brackets, or braces.

Advantages of Stacks

- **Simple to Use**: The push and pop operations are easy to understand and implement.
- **Efficient Memory Use**: If implemented using a linked list, the memory is utilized efficiently since it grows as needed.

Disadvantages of Stacks

- **Limited Access**: Only the top element is accessible, so accessing other elements can be cumbersome and inefficient.
- **Overflow and Underflow**: In array-based implementations, there's a risk of overflow if too many elements are pushed onto the stack. Similarly, attempting to pop from an empty stack will result in an underflow condition.

Real-World Examples of Stacks

- **Backtracking Algorithms**: Used in algorithms that explore all possible paths, such as maze solving and puzzle solving.
- **Web Browser Navigation**: The back button in a web browser is implemented using a stack to keep track of the history of visited web pages.
- **Programming Language Parsing**: Compilers and interpreters use stacks to parse nested structures such as parentheses or function calls.

Queues

A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element added to the queue is the first element to be removed.

In Java, the Queue interface is implemented by the LinkedLists class.

```
Queue<Integer> queue = new LinkedList<>();
```

Methods:

1. Insert:
    1. enqueue(): adds an element to the rear of the queue.
    2. add(): adds an element to the rear of the queue.
    3. offer(): adds an element to the rear of the queue.
2. Remove:
    1. dequeue(): removes the front element from the queue.
    2. remove(): removes the front element from the queue.
    3. poll(): removes the front element from the queue.

Where are queues used?

1. Keyboard buffers (letters should appear in the order they are typed).
2. Printer queues (documents should be printed in the order they are sent).
3. Used in LinkedLists, Priority Queues, Breadth First Search.

Priority Queues

A priority queue is a data structure that stores elements based on their priority. The element with the highest priority is removed first.

In Java, the PriorityQueue class is used to implement a priority queue.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

Linked Lists

A **Linked List** is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation, which allows them to dynamically grow and shrink in size.

Types of Linked Lists

1. **Singly Linked List**: Each node contains a data element and a reference (or pointer) to the next node in the list. The last node has a reference to `null`, indicating the end of the list.

   ```
   class Node {
       int data;
       Node next;

       Node(int data) {
           this.data = data;
           this.next = null;
       }
   }
   ```

2. **Doubly Linked List**: Each node contains a data element, a reference to the next node, and a reference to the previous node. This allows traversal in both directions (forward and backward).

   ```
   class Node {
       int data;
       Node next;
       Node prev;

       Node(int data) {
           this.data = data;
           this.next = null;
           this.prev = null;
       }
   }
   ```

3. **Circular Linked List**: In this variation, the last node points back to the first node, forming a circle. This can be implemented as either a singly or doubly linked list.

   ```
   class Node {
       int data;
       Node next;

       Node(int data) {
           this.data = data;
           this.next = this; // Points to itself initially
       }
   }
   ```

Basic Operations

- **Insertion**: Adding a new node to the list. Depending on where the new node is added, the operation can be classified as:
  - **At the beginning**
  - **At the end**
  - **In the middle**
- **Deletion**: Removing a node from the list. This can involve:
  - **Removing the first node**
  - **Removing the last node**
  - **Removing a node from the middle**
- **Traversal**: Visiting each node in the list from the head node to the last node (or first node again in a circular list).

```
void traverse(Node head) {
    Node current = head;
    while (current != null) {
        System.out.println(current.data);
        current = current.next;
    }
}
```

- **Search**: Finding a node with a specific value. This involves traversing the list and comparing each node's data with the target value.

```
boolean search(Node head, int key) {
    Node current = head;
    while (current != null) {
        if (current.data == key) {
            return true;
        }
        current = current.next;
    }
    return false;
}
```

Advantages of Linked Lists

- **Dynamic Size**: Unlike arrays, linked lists can easily grow and shrink as needed, making them ideal for scenarios where the number of elements is unknown or fluctuates frequently.
- **Efficient Insertions/Deletions**: Insertion and deletion of nodes are generally more efficient (O(1)) in linked lists compared to arrays, where shifting elements is necessary.

Disadvantages of Linked Lists

- **Memory Overhead**: Each node in a linked list requires extra memory for storing the reference to the next (and possibly previous) node.
- **Sequential Access**: Unlike arrays, linked lists do not provide direct access to elements, making search operations O(n) in the worst case.

Applications

- **Dynamic Memory Allocation**: Linked lists are often used in scenarios where memory allocation is dynamic and unpredictable.
- **Implementing Stacks and Queues**: Linked lists provide a flexible way to implement stack and queue data structures.
- **Graph Adjacency Representation**: In graph theory, linked lists are used to represent adjacency lists, which is a space-efficient way to represent sparse graphs.

Dynamic Arrays

A **Dynamic Array** is a data structure that allows for resizing, unlike a traditional array with a fixed size. Dynamic arrays grow and shrink as needed, making them more flexible and efficient for use cases where the number of elements can change dynamically.

How Dynamic Arrays Work

In a dynamic array, the underlying data structure is typically an array with a predefined initial capacity. When the array becomes full (i.e., when the number of elements equals the capacity), a new array with larger capacity is created, and the elements are copied over to this new array. This process is often referred to as "growing" the array.

When elements are removed and the number of elements falls below a certain threshold, the array may "shrink" to save memory by creating a smaller array and copying the elements over.

Core Operations

1. **Add (or Append)**: Adds a new element to the end of the dynamic array. If the array is full, it triggers a grow operation to increase its capacity before adding the new element.

```
dynamicArray.add("A");
```

2. **Insert**: Adds a new element at a specific index, shifting the subsequent elements to the right. This operation may also trigger a grow operation if the array is full.

```
dynamicArray.insert(1, "B");
```

3. **Delete**: Removes a specific element from the array, shifting the subsequent elements to the left to fill the gap. If the size of the array drops below a certain threshold, it may trigger a shrink operation.

```
dynamicArray.delete("A");
```

4. **Search**: Searches for a specific element in the array and returns its index if found, or `-1` if the element is not present.

```
int index = dynamicArray.search("B");
```

5. **Grow**: Increases the capacity of the array, typically by doubling its size, and copies all the elements to the new array.
6. **Shrink**: Decreases the capacity of the array when a significant portion of the array is empty, usually by halving the size.

Dynamic Arrays in Java: `ArrayList`

In Java, the most common implementation of a dynamic array is through the `ArrayList` class. `ArrayList` automatically handles the growing and shrinking of the array behind the scenes, making it easier to use compared to manually managing a dynamic array.

Advantages of Dynamic Arrays

- **Resizing Flexibility**: Dynamic arrays can grow and shrink as needed, making them suitable for use cases where the number of elements is unknown or fluctuates frequently.
- **Random Access**: Similar to traditional arrays, dynamic arrays provide O(1) time complexity for random access to elements.
- **Efficient Memory Use**: Dynamic arrays only use memory proportional to the number of elements they contain, unlike fixed-size arrays that reserve memory for the maximum capacity.

Disadvantages of Dynamic Arrays

- **Resizing Overhead**: Growing and shrinking the array requires copying elements, which can be time-consuming and lead to performance overhead.
- **Memory Fragmentation**: As the array grows and shrinks, it may lead to memory fragmentation, where memory is allocated in small chunks that are not contiguous.

Applications of Dynamic Arrays

- **Dynamic Lists**: Used in scenarios where the number of elements is not known in advance, such as dynamic collections of data.
- **Buffering Data**: Dynamic arrays are commonly used to store data buffers that grow and shrink dynamically based on input/output operations.
- **Implementation of Higher-Level Data Structures**: Dynamic arrays serve as the underlying data structure for more complex data structures like stacks, queues, and hash tables.

LinkedLists vs ArrayLists

Overview

In Java, both `LinkedList` and `ArrayList` are part of the Java Collections Framework and implement the `List` interface. However, they have different underlying data structures and thus perform differently in various scenarios.

- **`ArrayList`**: Uses a dynamic array to store elements. It provides O(1) time complexity for accessing elements by index, but can be slower for insertions and deletions, especially in the middle of the list, as elements need to be shifted.
- **`LinkedList`**: Uses a doubly linked list to store elements. It provides O(1) time complexity for insertions and deletions at both ends, but O(n) for accessing elements by index, since it requires traversing the list.

Performance Comparison

1. **Access by Index**:
   - **ArrayList**: Accessing elements by index is very fast (O(1)) because it uses an underlying array.
   - **LinkedList**: Accessing elements by index is slower (O(n)) because it requires traversing the list from the beginning or the end.
2. **Insertion and Deletion**:
   - **ArrayList**: Inserting or deleting elements at the end of the list is fast (O(1)), but at the beginning or middle of the list, it can be slow (O(n)) because elements need to be shifted.
   - **LinkedList**: Insertion and deletion are faster (O(1)) at the beginning or end of the list, and generally more efficient than `ArrayList` for these operations in such cases. However, inserting or deleting in the middle requires traversal (O(n)).

Situational Recommendations

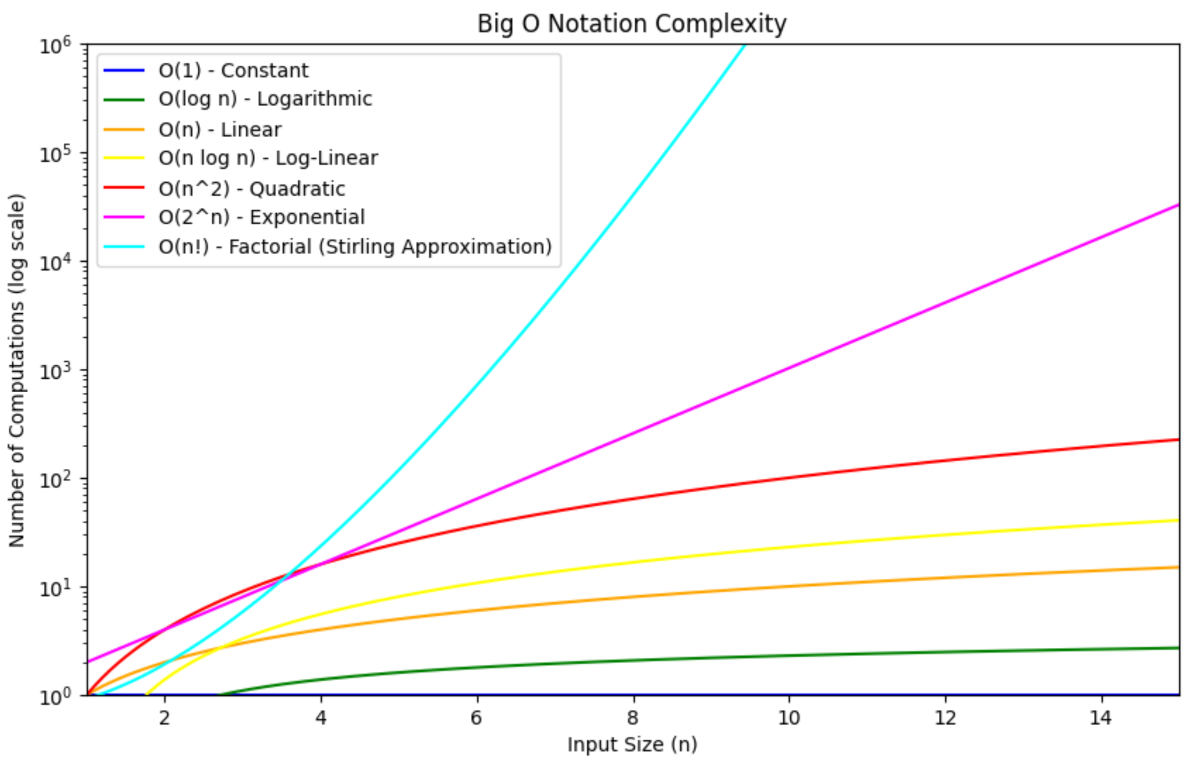
- **When to use `ArrayList`**:
  - If your application requires frequent access to elements by index, such as in cases where you're performing a lot of random access operations.
  - When the size of the list changes infrequently, and most operations are appending to the end of the list.
  - Ideal for use cases involving data storage where reads outnumber writes.
- **When to use `LinkedList`**:
  - If your application requires frequent insertions and deletions, especially at the beginning or end of the list.
  - When working with large lists where memory reallocation during resizing could be costly.
  - Suitable for scenarios where you need a queue or stack-like behavior, as it efficiently supports operations like adding/removing from both ends.

Big O Notation

Big O Notation is a mathematical notation used to describe the upper bound of an algorithm's runtime or space complexity in terms of the input size. It provides a way to express how the runtime or space requirements of an algorithm grow as the input size increases. Big O focuses on the worst-case scenario, ensuring that an algorithm will perform within a specific time or space limit, regardless of the input.

Common Big O Notations

The following are some of the most common Big O notations, listed from the most efficient to the least efficient:



- **O(1) - Constant Time**: The runtime does not change regardless of the input size. This is the most efficient complexity as the algorithm takes a constant amount of time, no matter how large the input is.
  - **Examples**:
    - Accessing an element in an array by index: `array[index]`.
    - Inserting an element at the beginning of a `LinkedList`.
- **O(log n) - Logarithmic Time**: The runtime grows logarithmically as the input size increases. Algorithms with logarithmic time complexity are very efficient for large datasets.
  - **Examples**:
    - Binary search in a sorted array.
    - Finding an element in a balanced binary search tree (BST).
- **O(n) - Linear Time**: The runtime grows linearly with the input size. If the input size doubles, the runtime also doubles. This is typical of algorithms that need to examine each element in the input.
  - **Examples**:
    - Looping through elements in an array.
    - Searching through a `LinkedList`.
- **O(n log n) - Log-Linear Time**: The runtime grows faster than linear time but slower than quadratic time. This complexity is common in efficient sorting algorithms.
  - **Examples**:
    - Quick sort.
    - Merge sort.
    - Heap sort.
- **O(n^2) - Quadratic Time**: The runtime grows quadratically as the input size increases. If the input size doubles, the runtime quadruples. This is common in algorithms with nested loops.
  - **Examples**:
    - Insertion sort.
    - Selection sort.
    - Bubble sort.
- **O(2^n) - Exponential Time**: The runtime doubles with each additional input element. This is very inefficient for large inputs and is often seen in algorithms that solve problems by brute force.
  - **Examples**:
    - Solving the Towers of Hanoi.
    - Certain recursive algorithms that explore all possibilities.
- **O(n!) - Factorial Time**: The runtime grows factorially with the input size. This is the least efficient time complexity and is typically found in algorithms that generate all possible permutations of an input.
  - **Examples**:
    - Solving the Traveling Salesman Problem (TSP) with brute force.

        - Generating all permutations of a set.

Searching Algorithms

Linear Search

Iterate through a collection one element at a time.

- **Runtime Complexity**: O(n)

Disadvantages:

- Slow for large data sets

Advantages:

- Fast for searches of small to medium data sets
- Does not need to be sorted
- Useful for data structures that do not have random access (Linked List)

Binary Search

Binary Search is an efficient algorithm used to find the position of a target value within a sorted array or list. The key idea behind Binary Search is to repeatedly divide the search interval in half, reducing the problem size exponentially with each step.

How It Works

1. **Start with the Middle Element**: The algorithm begins by comparing the target value with the middle element of the array.
2. **Adjust the Search Interval**:
    - If the target value is equal to the middle element, the search is complete.
    - If the target value is less than the middle element, the search continues on the left half of the array.
    - If the target value is greater than the middle element, the search continues on the right half of the array.
3. **Repeat**: This process is repeated, halving the search interval each time, until the target value is found or the interval is empty.

Time Complexity

- **O(log n)** — With each step, the search space is halved, leading to a logarithmic time complexity, which is much faster than linear search for large datasets.

Advantages

- **Efficiency**: Binary Search is significantly faster than Linear Search, especially for large, sorted datasets, because it reduces the number of comparisons needed to find the target.
- **Logarithmic Time**: The algorithm's time complexity of O(log n) makes it highly efficient for searching through large volumes of data.

Disadvantages

- **Requires Sorted Data**: Binary Search can only be applied to data that is already sorted. If the data is unsorted, it must first be sorted, which can add to the overall time complexity.
- **Not Ideal for Dynamic Data**: If the data is frequently updated with insertions and deletions, maintaining a sorted order can be inefficient, making Binary Search less practical in such scenarios.
- **Less Effective for Small Data Sets**: For very small data sets, the overhead of dividing the search space might not be worth the performance gain over simpler algorithms like Linear Search.

Use Cases

Binary Search is ideal for scenarios where:

- The dataset is large and sorted.
- Fast retrieval of elements is critical.
- The dataset remains relatively static (few insertions/deletions).

Binary Search is widely used in applications such as searching in databases, looking up records in a sorted array, or even in various algorithms that require efficient search capabilities.

Interpolation Search

Interpolation Search is an improved variant of Binary Search, designed for searching in a uniformly distributed, sorted array. Unlike Binary Search, which always checks the middle element, Interpolation Search estimates the position of the target value based on the distribution of the data, making it potentially faster in certain scenarios.

How It Works

1. **Estimate Position**: The algorithm estimates the position of the target value using the formula:

```
int probe = low + (high - low) * (target - array[low]) /
        (array[high] - array[low]);
```

Here, `low` and `high` represent the indices of the current search interval, and `target` is the value being searched.

2. **Compare and Adjust**:
   - If the target value matches the element at the estimated position, the search is complete.
   - If the target is less than the estimated element, the search continues in the lower subarray.
   - If the target is greater, the search continues in the upper subarray.
3. **Repeat**: This process is repeated, narrowing down the search interval based on the estimated position, until the target is found or the search interval is empty.

Time Complexity

- **O(log log n)** — Interpolation Search can be very fast, with a time complexity of O(log log n) in the best case for uniformly distributed data.
- However, in the worst case, particularly with non-uniform distributions, the time complexity can degrade to **O(n)**.

Advantages

- **Potentially Faster Than Binary Search**: For large, uniformly distributed datasets, Interpolation Search can be more efficient than Binary Search, as it uses a more refined estimate to locate the target.
- **Effective for Specific Data Types**: It excels in scenarios where the data is uniformly distributed, such as indexes in a database or sorted numerical data.

Disadvantages

- **Requires Uniformly Distributed Data**: The algorithm's efficiency depends on the data being uniformly distributed. If the data is not well-distributed, the performance can degrade significantly.
- **Complexity in Implementation**: Interpolation Search is more complex to implement than Binary Search, requiring a deeper understanding of the data distribution.

Use Cases

Interpolation Search is most effective in the following scenarios:

- **Large, Uniformly Distributed Datasets**: Ideal for searching in large datasets where the values are evenly spread out, such as indexes in databases or numerical ranges.
- **Specific Applications**: Used in situations where the distribution of the data is known and can be leveraged to improve search performance.

While Interpolation Search can offer significant performance improvements over Binary Search in the right context, it is essential to consider the nature of the data before opting for this method.

Sorting Algorithms

Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

How It Works

1. **Compare Adjacent Elements**: Start with the first element of the array and compare it to the next element.
2. **Swap If Necessary**: If the first element is greater than the second, swap them. Otherwise, move to the next pair of elements.
3. **Repeat**: Continue this process for each pair of adjacent elements in the array. After one full pass through the array, the largest element will have "bubbled up" to its correct position at the end.
4. **Iterate Until Sorted**: Repeat the process for the remaining unsorted elements. With each pass, the next largest element is placed in its correct position, reducing the number of elements to check in subsequent passes.

Time Complexity

- **Worst-Case and Average Case**: O(n^2) — This occurs when the array is in reverse order or unordered, requiring the maximum number of comparisons and swaps.
- **Best Case**: O(n) — This occurs when the array is already sorted, and no swaps are needed. The algorithm can terminate early after one pass.

Advantages

- **Simplicity**: Bubble Sort is easy to understand and implement.
- **No Additional Space Required**: It sorts the array in place, so no additional memory is required beyond the input data.
- **Early Termination**: In the best-case scenario (an already sorted array), the algorithm can terminate early, making it more efficient than other O(n^2) algorithms in such cases.

Disadvantages

- **Inefficiency**: Bubble Sort is not suitable for large datasets because of its O(n^2) time complexity in the average and worst cases.
- **Slowness**: Even for relatively small lists, other sorting algorithms like Quick Sort, Merge Sort, or Insertion Sort are generally faster.

Use Cases

Bubble Sort is generally used for educational purposes to demonstrate how sorting algorithms work. It's rarely used in practical applications due to its inefficiency. However, it can be useful in situations where the dataset is nearly sorted, and its simplicity is a significant advantage.

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It sorts an array by repeatedly finding the smallest (or largest) element in the array and swapping it with the element at the current position.

How It Works

1. **Start with the First Element**: Begin with the first element in the array.
2. **Find the Minimum Element**: Search through the array to find the smallest element.
3. **Swap**: Swap this smallest element with the element at the current position.
4. **Move to the Next Position**: Move to the next element in the array and repeat the process until the entire array is sorted.

Time Complexity

- **O(n^2)**

Advantages

- **Simplicity**: Easy to implement and understand.
- **In-Place Sorting**: No additional storage space is required.

Disadvantages

- **Inefficiency**: Not suitable for large datasets due to O(n^2) time complexity.
- **Unstable**: May not preserve the relative order of equal elements.

Use Cases

Selection Sort is best used for small datasets or in educational contexts to demonstrate basic sorting concepts.

Insertion Sort

Insertion Sort is a simple and intuitive comparison-based sorting algorithm. It builds the final sorted array one element at a time, by repeatedly taking the next unsorted element and inserting it into its correct position relative to the already checked elements in the array.

How It Works

1. **Start with the Second Element**: Begin with the second element in the array because it needs to be compared with the first element to determine its correct position.
2. **Compare and Shift**: Compare the current element with the elements before it in the array. Shift the larger elements to the right until the correct position for the current element is found.
3. **Insert the Element**: Insert the current element into its correct position within the array.
4. **Repeat**: Move to the next element and repeat the process until the entire array is sorted.

Time Complexity

- **O(n^2)** — In the worst and average cases, Insertion Sort requires comparing and shifting elements, leading to a quadratic time complexity.
- **O(n)** — In the best case, when the array is already sorted, Insertion Sort performs at linear time.

Advantages

- **Simplicity**: Easy to understand and implement.
- **Efficient for Small or Nearly Sorted Data**: Performs well on small datasets or datasets that are already partially sorted.
- **Stable**: Preserves the relative order of equal elements.
- **In-Place Sorting**: Requires no additional memory for sorting, as it rearranges the elements within the original array.

Disadvantages

- **Inefficiency on Large Data Sets**: Due to its O(n^2) time complexity, it is not suitable for large datasets.

Use Cases

Insertion Sort is particularly useful for:

- **Small Data Sets**: Where its simplicity and efficiency outweigh its quadratic time complexity.
- **Nearly Sorted Data**: Ideal for situations where the data is already partially sorted, making the algorithm very efficient.
- **Real-Time Applications**: Because it can sort elements as they arrive, Insertion Sort is sometimes used in real-time systems.

Insertion Sort is often used in practice for small arrays or as a final step in more complex algorithms, such as Quick Sort or Merge Sort, where the overhead of more sophisticated methods is unnecessary.

Recursion

Recursion is a programming technique where a function calls itself in order to solve a problem. This method involves breaking down a problem into smaller, more manageable sub-problems that are easier to solve. Recursion is often used for tasks that can be defined in terms of similar subtasks, such as computing factorials, traversing data structures, or solving puzzles like the Tower of Hanoi.

How It Works

Recursion typically involves two main components:

1. **Base Case**: The condition under which the recursion stops. Without a base case, the function would call itself indefinitely, leading to a stack overflow. The base case defines the simplest instance of the problem, which can be solved directly without further recursion.
2. **Recursive Case**: The part of the function where it calls itself with a modified argument, gradually working towards the base case. Each recursive call reduces the problem's complexity, inching closer to the base case until the recursion can terminate.

Types of Recursion

Recursion can be classified into several types, including**Tail Recursion** (final recursion), **Non-Tail Recursion** (non-final recursion), **Direct Recursion**, and **Indirect Recursion**.

1. Tail Recursion (Final Recursion)

Tail Recursion occurs when the recursive call is the last operation in the function. In other words, there is nothing left to do after the recursive call returns. This type of recursion is more memory-efficient because many compilers and interpreters can optimize tail-recursive functions to avoid adding a new frame to the call stack for each recursive call. This optimization is known as *tail call optimization (TCO)*.

```
private static void tailRecursionWalk(int steps) {
    if (steps == 0) return;  // Base case
    System.out.println("You take a step");
    tailRecursionWalk(steps - 1);  // Recursive call
}
```

2. Non-Tail Recursion (Non-Final Recursion)

Non-Tail Recursion occurs when there are still operations left to perform after the recursive call returns. This means that each recursive call must wait for the subsequent recursive calls to complete before it can finish its execution, which can lead to higher memory usage as each call is kept on the stack until all calls are resolved.

```
private static int nonTailRecursionFactorial(int num) {
    if (num < 1) return 1;  // Base case
    return num * nonTailRecursionFactorial(num - 1);  // Recursive call followed by multiplication
}
```

3. Direct Recursion

Direct Recursion occurs when a function calls itself directly. This is the most common form of recursion and is what most people think of when they hear the term "recursion."

```
private static void directRecursion(int n) {
    if (n <= 0) return;  // Base case
    directRecursion(n - 1);  // Recursive call
}
```

4. Indirect Recursion

Indirect Recursion occurs when a function calls another function, which eventually leads to the original function being called again. This can involve two or more functions calling each other in a cycle.

```
private static void functionA(int n) {
    if (n <= 0) return;  // Base case
    functionB(n - 1);  // Call to another function
}

private static void functionB(int n) {
    if (n <= 0) return;  // Base case
    functionA(n - 1);  // Call back to the original function
}
```

Advantages of Recursion

- Simplifies Complex Problems: Recursion can simplify the solution to complex problems by breaking them down into smaller, more manageable pieces.
- Natural Fit for Certain Problems: Some problems, particularly those involving hierarchical data structures (like trees or graphs), are naturally suited to recursive solutions.

Disadvantages of Recursion

- Performance Overhead: Each recursive call adds a new layer to the call stack, which can lead to high memory usage and slower performance compared to iterative solutions.
- Risk of Stack Overflow: If the base case is not properly defined or if the recursion is too deep (i.e., too many recursive calls), the program can run out of stack memory, resulting in a stack overflow error.
- Complexity: Recursive solutions can be more difficult to understand and debug, especially for those new to the concept.

Use Cases

Recursion is particularly useful in scenarios where a problem can be naturally divided into similar subproblems, such as:

- Mathematical Calculations: Functions like factorials, Fibonacci sequences, or powers.
- Data Structure Traversal: Navigating trees, graphs, or other nested data structures.
- Algorithms: Recursive algorithms like quicksort, merge sort, and binary search.

Recursion is a powerful tool in a programmer's arsenal, but it should be used judiciously, especially when performance and memory usage are critical considerations.

Merge Sort

Merge Sort is a divide-and-conquer algorithm that efficiently sorts an array by dividing it into smaller subarrays, sorting those subarrays, and then merging them back together. This approach ensures that the entire array is sorted in a methodical and efficient manner.

How It Works

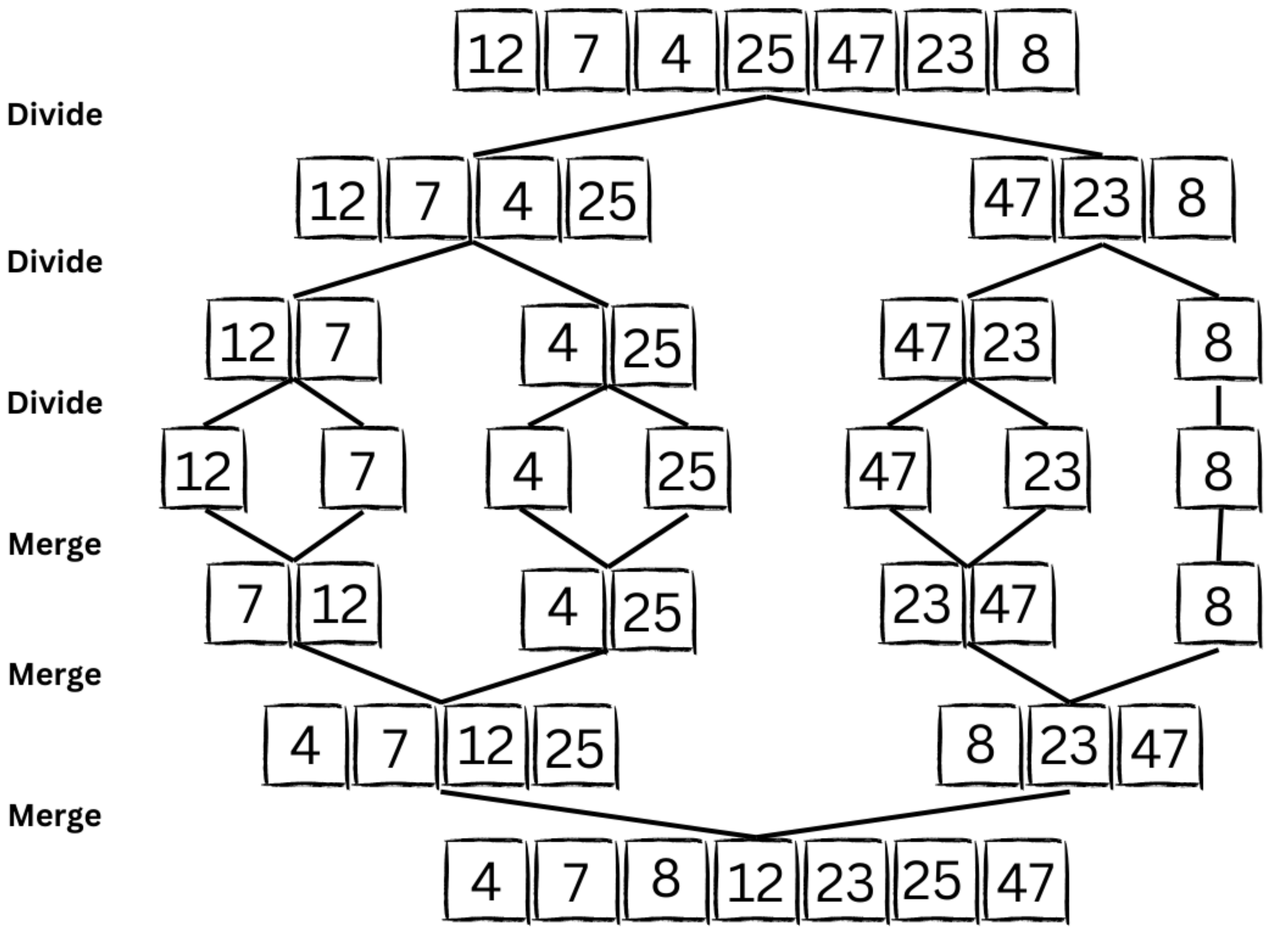
1. **Divide**: The algorithm starts by dividing the array into two roughly equal halves. This process is repeated recursively until each subarray contains only one element. At this point, each element is considered sorted by definition.

2. **Conquer**: Once the array has been divided into individual elements, the algorithm begins to merge these elements back together. During the merging process, the elements are compared and sorted, ensuring that the resulting merged array is in order.

3. **Combine**: Finally, the sorted subarrays are combined back together to form a fully sorted array.

Visualization of the Process

Here is a visual representation of how Merge Sort works:



This image illustrates the process of dividing the array into smaller subarrays, sorting them, and then merging them back together.

Time Complexity

- **O(n log n)** — Merge Sort consistently operates with a time complexity of O(n log n), making it efficient for large datasets. This efficiency comes from the fact that the array is divided in half at each step (log n divisions) and each level requires a linear amount of work to merge the subarrays (n operations).

Space Complexity

- **O(n)** — Merge Sort requires additional memory proportional to the size of the array being sorted. This space is used to hold the left and right subarrays during the merge process. The need for extra storage makes Merge Sort less space-efficient compared to in-place sorting algorithms like Quick Sort.

Advantages

- **Stable Sorting**: Merge Sort is stable, meaning that it preserves the relative order of equal elements in the array.
- **Predictable Performance**: It has a consistent O(n log n) time complexity, regardless of the initial order of elements.
- **Efficient for Large Datasets**: Particularly useful for sorting large datasets due to its efficient handling of data.

Disadvantages

- **Additional Memory Usage**: Merge Sort requires additional memory proportional to the size of the array being sorted, as it needs space to hold the left and right subarrays during the merge process.
- **Not In-Place**: Unlike some other sorting algorithms, Merge Sort does not sort the array in place; it requires extra storage, which can be a disadvantage in memory-constrained environments.

Use Cases

Merge Sort is particularly useful in scenarios where:

- **Stable Sort is Required**: Situations where maintaining the relative order of records with equal keys is important.
- **Large Data Volumes**: Handling large datasets that need to be sorted efficiently, especially when the data is stored on external storage or needs to be processed in chunks.
- **External Sorting**: Commonly used in external sorting algorithms where the data is too large to fit into memory and needs to be sorted using external storage.

Merge Sort is a powerful sorting algorithm, especially when working with large datasets or when stability is a concern. It is widely used in various applications, from simple data sorting to more complex algorithms in computer science.

QuickSort

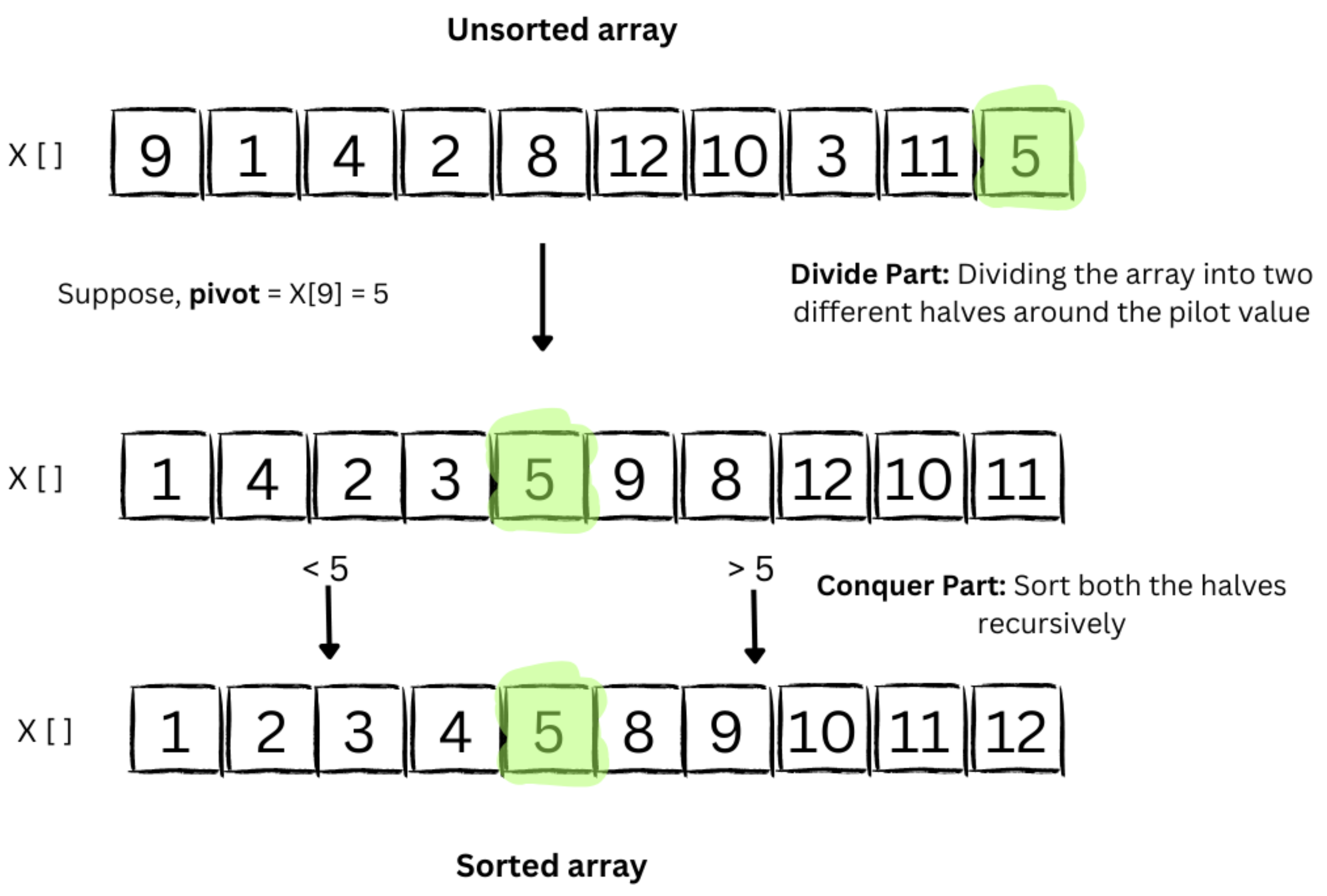
QuickSort is a highly efficient sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. This process results in a sorted array.

How It Works

1. **Choose a Pivot**: Select an element from the array to serve as the pivot. Common choices include picking the first element, the last element, or a random element.
2. **Partitioning**: Rearrange the array so that all elements less than the pivot are on the left side of the pivot and all elements greater than the pivot are on the right side. The pivot element is then placed in its correct sorted position.
3. **Recursively Sort Sub-arrays**: Recursively apply the above steps to the sub-arrays on the left and right of the pivot.
4. **Base Case**: The recursion terminates when the sub-array has fewer than two elements, meaning it is already sorted.

Visualization of the Process

Here is a visual representation of how QuickSort works:


Unsorted array
X [ ]
9 1 4 2 8 12 10 3 11 5
Suppose, **pivot** = X[9] = 5
Divide Part: Dividing the array into two different halves around the pilot value
X [ ]
1 4 2 3 5 9 8 12 10 11
< 5
> 5
Conquer Part: Sort both the halves recursively
X [ ]
1 2 3 4 5 8 9 10 11 12
Sorted array

This image illustrates the process of choosing a pivot, partitioning the array around the pivot, and recursively sorting the sub-arrays.

Time Complexity

- **Best and Average Case**: O(n log n) — QuickSort is efficient when the pivot divides the array into two nearly equal sub-arrays.
- **Worst Case**: O(n^2) — This occurs when the pivot is consistently the smallest or largest element, leading to unbalanced partitions.

Space Complexity

- **O(log n)** — QuickSort is an in-place sorting algorithm, meaning it requires only a small, constant amount of extra space for the recursion stack. However, in the worst case, the space complexity can degrade to O(n) due to deep recursion.

Advantages

- **Efficiency**: QuickSort is generally faster in practice than other O(n log n) algorithms like Merge Sort, especially for large datasets.
- **In-Place Sorting**: It sorts the array in place, requiring only a small amount of extra memory.
- **Flexibility**: It can be implemented with different strategies for choosing the pivot, which can be tailored to the specific characteristics of the data.

Disadvantages

- **Worst-Case Performance**: In its basic form, QuickSort has a worst-case time complexity of O(n^2), which can occur if the pivot selection is poor.
- **Not Stable**: QuickSort is not a stable sort, meaning that the relative order of equal elements may not be preserved.

Use Cases

QuickSort is particularly useful in scenarios where:

- **Large Datasets**: It is often the fastest algorithm for large datasets, especially when the dataset can be partitioned effectively.
- **In-Place Sorting**: When memory usage is a concern, QuickSort's in-place sorting is an advantage.
- **General Purpose**: QuickSort is a good default choice for a general-purpose sorting algorithm, with a good balance of efficiency and ease of implementation.

QuickSort is one of the most widely used sorting algorithms due to its efficiency and performance in average cases, making it a popular choice in many real-world applications.

Hash Tables

A **HashTable** is a data structure that stores key-value pairs. Each key in a HashTable is unique, and it is associated with a specific value. This structure allows for efficient data retrieval based on keys, making operations like insertion, deletion, and lookup very fast.

Key Concepts

- **Hashing**: Hashing is the process of converting a key into a specific index within the HashTable. The key is passed through a hash function, which computes an integer (hash code). The hash code is then used to calculate an index within the HashTable using the modulus operator (`%`) with the table's capacity.

```
key.hashCode() % capacity = index
```

- **Bucket**: A bucket is an indexed storage location in the HashTable where one or more entries (key-value pairs) are stored. If multiple keys hash to the same index, these entries are stored together in the same bucket, often implemented as a linked list.
- **Collision**: A collision occurs when two different keys produce the same hash code and thus map to the same index in the HashTable. Collisions are managed by storing both entries in the same bucket. However, the more collisions there are, the less efficient the HashTable becomes. Reducing collisions by choosing a good hash function and an appropriate table size is critical to maintaining efficiency.

Runtime Complexity

- **Best Case: O(1)** — In the best case, the hash function distributes keys uniformly across the table, and there are no collisions, allowing for constant-time operations.
- **Worst Case: O(n)** — In the worst case, all keys hash to the same index, causing all entries to be stored in a single bucket, resulting in linear-time operations as it degrades to a linked list.

Differences Between sortingAlgorithms.HashTables and HashMaps

- Thread Safety:
  - `HashTable` is synchronized, meaning it is thread-safe and can be shared between multiple threads without additional synchronization.
  - `HashMap` is not synchronized by default, making it faster but not thread-safe. If thread safety is required, HashMap can be synchronized externally using Collections.synchronizedMap().
- Null Values:
  - `HashTable` does not allow null keys or values. If you try to insert a null key or value, a NullPointerException is thrown.
  - `HashMap` allows one null key and multiple null values.
- Legacy Status:
  - `HashTable` is considered a legacy class from earlier versions of Java and has largely been replaced by HashMap for most use cases.
  - `HashMap` is part of the Java Collections Framework and is generally preferred in modern Java programming due to its flexibility and better performance.

sortingAlgorithms.HashTables are effective for quick data access and management, especially in multi-threaded environments where synchronization is necessary. However, for most modern applications, HashMap is preferred due to its flexibility and better performance.

Graphs

Introduction to Graphs

A **graph** is a fundamental data structure in computer science used to model relationships between objects. A graph consists of a set of vertices (or nodes) and a set of edges that connect pairs of vertices. Graphs are widely used in various fields, including computer networks, social networks, transportation systems, and many more.

Types of Graphs

Graphs can be classified into several types based on their properties:

1. **Directed Graph (Digraph)**: In a directed graph, the edges have a direction, meaning they go from one vertex to another. The direction of the edge is indicated by an arrow. This type of graph is useful for representing one-way relationships, such as the flow of information or traffic.

2. **Undirected Graph**: In an undirected graph, the edges do not have a direction, indicating a bidirectional relationship between vertices. This type of graph is useful for representing mutual relationships, such as friendships in a social network.

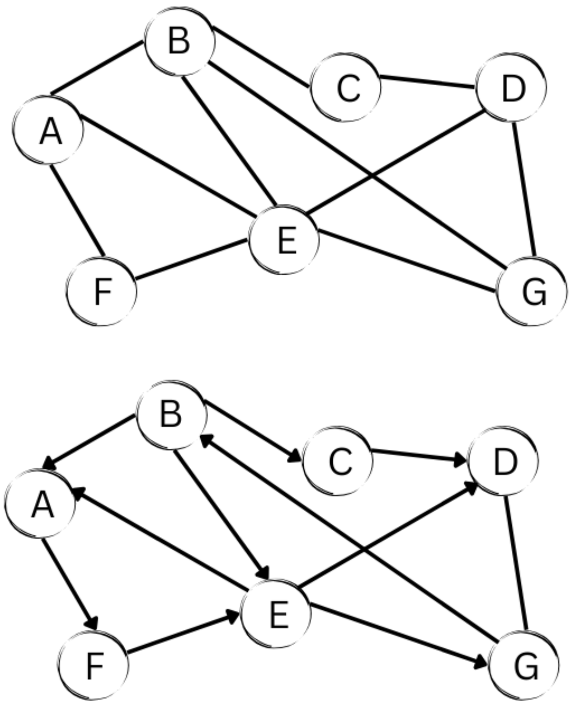
3. **Weighted Graph**: In a weighted graph, each edge has an associated numerical value, called a weight. These weights can represent costs, distances, or any other quantitative relationship between the vertices.

4. **Unweighted Graph**: An unweighted graph is a graph where all edges are considered to have the same weight, usually interpreted as 1. This is typical in simple graphs where only the presence or absence of a connection matters.

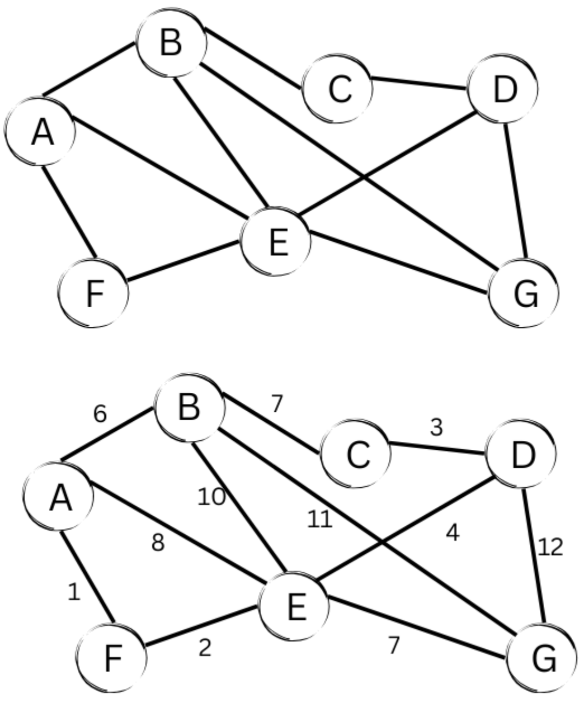
Here is a visual representation of these types of graphs:

Types of Graphs

DIRECTED VS UNDIRECTED



WEIGHTED VS UNWEIGHTED



Adjacency Matrix

An **adjacency matrix** is a 2D array of size VxV where V is the number of vertices in a graph. The matrix is used to represent which vertices (or nodes) are adjacent to which other vertices.

- **Directed Graph**: In an adjacency matrix for a directed graph, if there is an edge from vertex `v1` to vertex `v2`, then the matrix entry at `[v1][v2]` is 1 (or the weight of the edge if the graph is weighted). Otherwise, it is 0.
- **Undirected Graph**: For an undirected graph, the adjacency matrix is symmetric. If there is an edge between `v1` and `v2`, then both `[v1][v2]` and `[v2][v1]` will be 1 (or the corresponding weight).

Time Complexity:

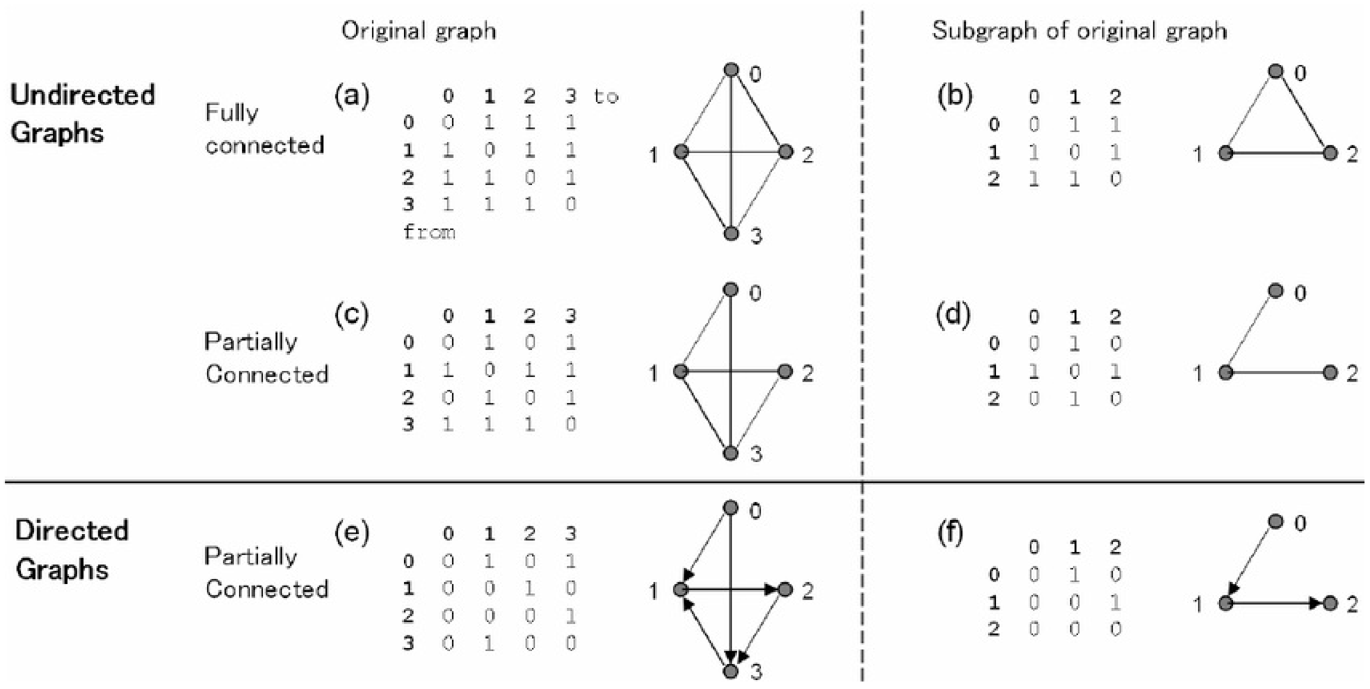
- **Edge Lookup**: O(1) — Checking whether an edge exists between two vertices is constant time since you can directly access the matrix entry.
- **Iteration Over All Edges**: O(V^2) — Iterating through all the edges requires checking each entry in the VxV matrix.

Space Complexity:

- **O(V^2)** — The space complexity is proportional to the number of vertices squared. This makes adjacency matrices less space-efficient for sparse graphs where the number of edges is much smaller than V^2.

The adjacency matrix is simple to implement and allows for quick edge lookups, but it can be memory inefficient for sparse graphs (graphs with few edges).

Here is a visual representation of graphs and their corresponding adjacency matrices:



(a)

	0	1	2	3 to
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0
from				

(b)

	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

(c)

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	1	0	1
3	1	1	1	0

(d)

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

(e)

	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

(f)

	0	1	2
0	0	1	0
1	0	0	1
2	0	0	0

Adjacency List

An **adjacency list** is another way to represent a graph. Instead of a 2D array, an adjacency list uses an array of lists. The array represents all the vertices, and each entry in the array points to a list of nodes that are adjacent to it.

- **Space Efficiency**: Adjacency lists are more space-efficient for sparse graphs compared to adjacency matrices because they only store the edges that exist.
- **Traversal Efficiency**: Adjacency lists allow for efficient traversal of the graph, particularly for algorithms that need to explore all adjacent nodes (like depth-first search or breadth-first search).

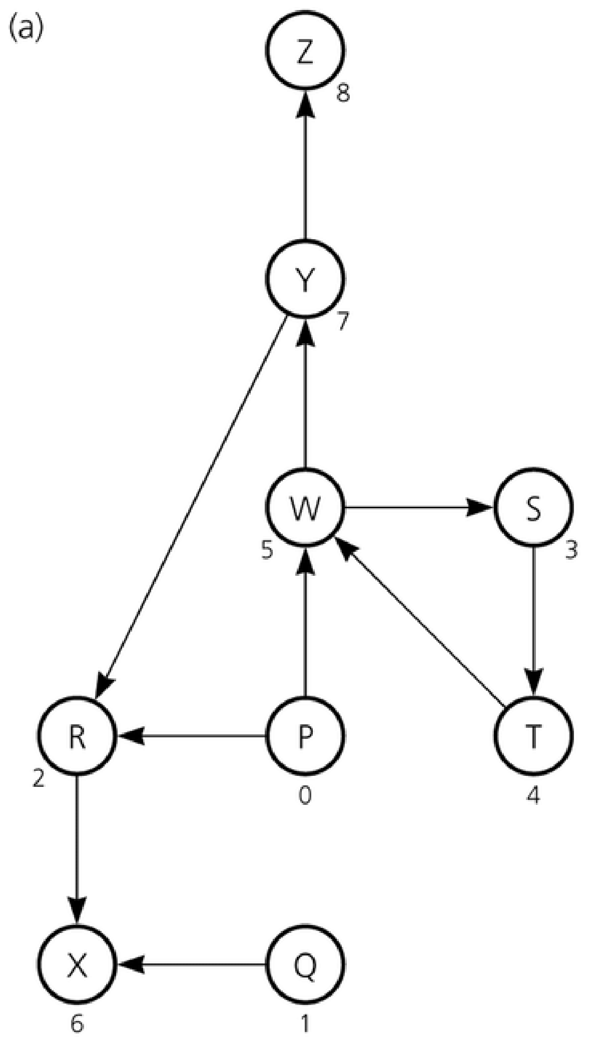
Time Complexity:

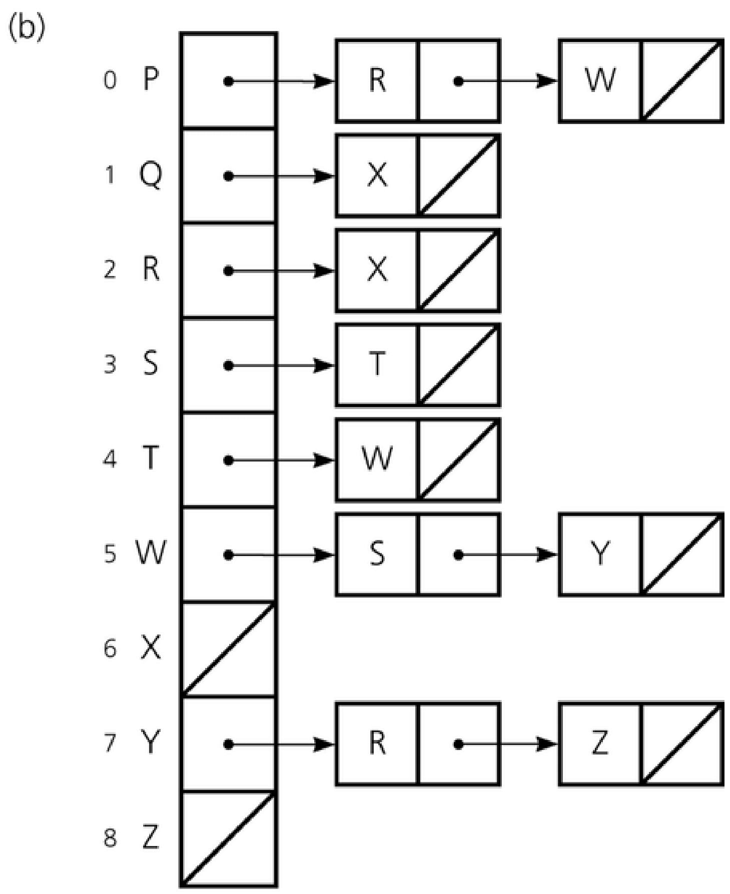
- **Edge Lookup**: O(V) — Checking whether an edge exists between two vertices may require iterating through a list of adjacent vertices, which can take linear time in the number of vertices.
- **Iteration Over All Edges**: O(V + E) — Iterating over all edges requires visiting each vertex and traversing its adjacency list, resulting in time complexity proportional to the number of vertices and edges.

Space Complexity:

- **O(V + E)** — The space complexity is proportional to the sum of the number of vertices and the number of edges. This makes adjacency lists much more space-efficient than adjacency matrices for sparse graphs.

The following image shows a directed graph and its corresponding adjacency list representation:





Depth-First Search (DFS)

Depth First Search (DFS) is a fundamental algorithm in graph theory used to traverse or search through graph data structures. The algorithm starts at a given node (referred to as the "starting" node) and explores as far as possible along each branch before backtracking. This approach allows the algorithm to explore the depth of the graph before moving horizontally to other nodes.

How DFS Works

1. **Start at the Chosen Node**: DFS begins at a specified starting node, marking it as visited. This node is often referred to as the "root" in tree structures, though in a general graph, it can be any node.
2. **Explore Each Branch Recursively**: From the current node, DFS recursively visits each unvisited adjacent node. This process continues deeper into the graph until it reaches a node that has no unvisited adjacent nodes.
3. **Backtrack**: Once the algorithm reaches a node with no unvisited adjacent nodes, it backtracks to the previous node and continues exploring any other unvisited branches from that node.
4. **Complete the Search**: This process repeats until all nodes reachable from the starting node have been visited. If the graph is disconnected, DFS can be restarted from any unvisited node to explore other components of the graph.

Applications of DFS

- **Path Finding**: DFS is used in algorithms that need to find a path between two nodes in a graph.
- **Cycle Detection**: DFS can be used to detect cycles in a graph by checking if there is a back edge in the recursion stack.
- **Topological Sorting**: DFS is used to perform topological sorting in a directed acyclic graph (DAG).
- **Connectivity**: DFS can help determine if a graph is connected, meaning that there is a path between every pair of vertices.
- **Maze and Puzzle Solving**: DFS is useful in solving puzzles or mazes where the solution is deep within the structure.

Breadth-First Search (BFS)

Breadth First Search (BFS) is a graph traversal algorithm that explores the nodes of a graph level by level. It starts at a given node (referred to as the "starting" node) and visits all its neighbors before moving on to the next level of nodes. BFS is particularly useful for finding the shortest path in unweighted graphs.

How BFS Works

1. **Start at the Chosen Node**: BFS begins at a specified starting node, marking it as visited and placing it in a queue. The queue is used to keep track of the nodes that need to be explored.
2. **Explore Neighboring Nodes**: The algorithm dequeues the front node from the queue and visits all its adjacent, unvisited nodes. Each of these nodes is then

marked as visited and added to the queue.

3. **Move to the Next Level**: BFS continues to dequeue nodes from the front of the queue, visiting their unvisited neighbors, and adding those neighbors to the queue. This process repeats until the queue is empty, meaning all reachable nodes have been visited.

4. **Complete the Search**: BFS ensures that all nodes are visited level by level, meaning it explores the closest nodes first before moving on to nodes that are farther away.

Common Applications of BFS

- **Shortest Path**: BFS is commonly used to find the shortest path in unweighted graphs.
- **Level Order Traversal**: BFS is ideal for traversing a graph level by level, making it useful for tasks like level-order traversal in trees.
- **Connected Components**: BFS can be used to find all connected components in a graph, identifying clusters of connected nodes.
- **Bipartite Graph Checking**: BFS is used to check if a graph is bipartite, meaning its vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent.

BFS vs DFS

Feature	BFS (Breadth First Search)	DFS (Depth First Search)
Traversal Approach	Traverses a graph level by level.	Traverses a graph branch by branch.
Data Structure	Utilizes a Queue.	Utilizes a Stack.
Best Use Case	Better if the destination is on average close to the start.	Better if the destination is on average far from the start.
Order of Visit	Siblings are visited before children.	Children are visited before siblings.
Popularity	Common for social networks, web crawlers, finding shortest paths.	More popular for games, puzzles, and problems requiring deep exploration.

Trees

Introduction to Trees

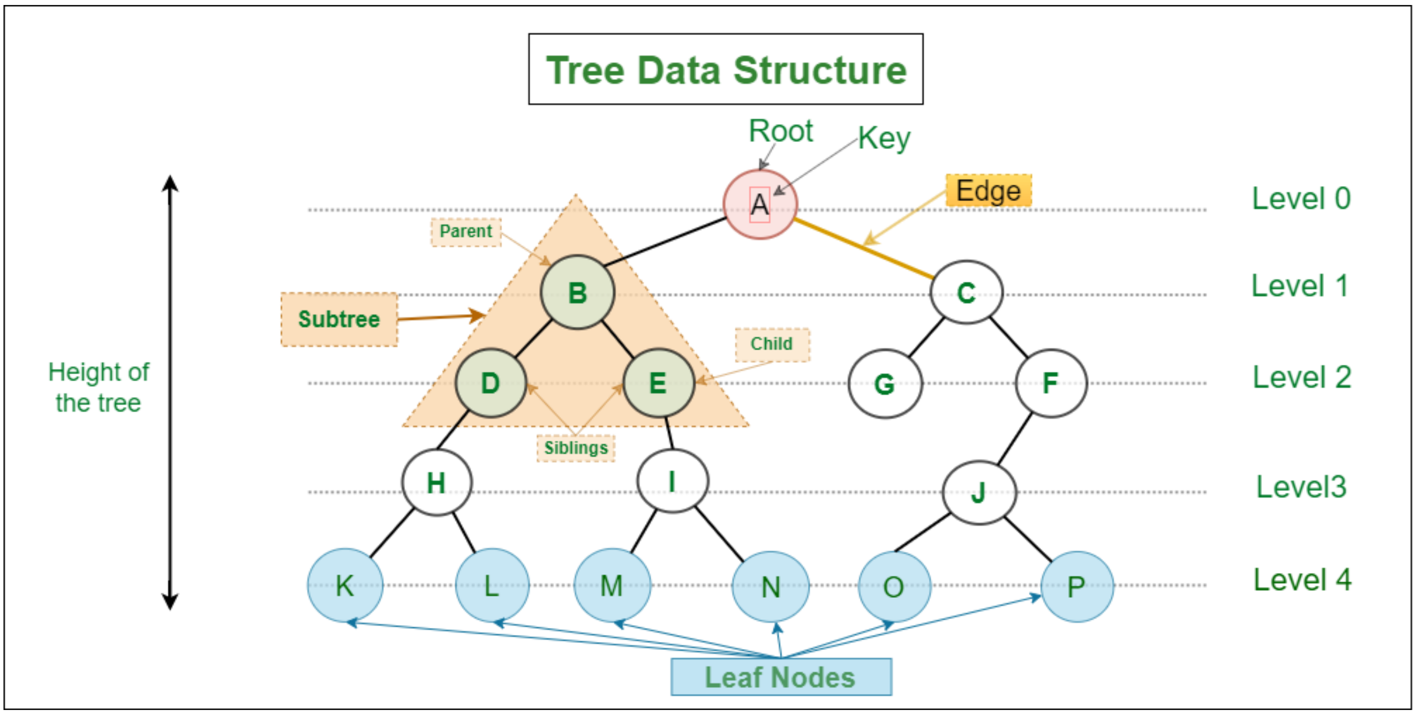
A **tree** is a widely used data structure in computer science that is a hierarchical, non-linear collection of nodes. Each node contains a value or key and can have child nodes, forming a parent-child relationship. Trees are essential for modeling hierarchical structures like file systems, organizational charts, and more.

Key Terminology

- **Root**: The topmost node in a tree, from which all other nodes descend. The root node does not have a parent.
- **Child**: A node that descends from another node. In the diagram, node B is a child of node A.
- **Parent**: A node that has one or more children. Node A is the parent of node B.
- **Subtree**: A tree formed by a node and all of its descendants. For example, nodes B, D, and E form a subtree.
- **Siblings**: Nodes that share the same parent. In the diagram, nodes D and E are siblings as they both descend from node B.
- **Leaf Nodes**: Nodes that do not have any children. These nodes are the "end" of the tree. In the diagram, nodes K, L, M, N, O, and P are leaf nodes.
- **Edge**: The connection between two nodes. An edge connects a parent to its child.
- **Height**: The height of a tree is the number of edges on the longest downward path from the root to a leaf.
- **Level**: The level of a node is determined by the distance from the root node. The root is at level 0, its children are at level 1, and so on.

Tree Structure Visualization

Here is a visual representation of a tree data structure:



Characteristics of Trees

1. **Hierarchical Structure**: Unlike linear data structures like arrays or linked lists, trees organize data hierarchically, allowing efficient insertion, deletion, and search operations.
2. **Single Root**: Each tree has only one root node from which all other nodes descend.
3. **Recursive Nature**: Trees are inherently recursive structures since each subtree is itself a tree.

Applications of Trees

- **File Systems**: Trees are used to represent hierarchical file structures, where folders contain subfolders and files.
- **Binary Search Trees (BST)**: Trees are used to efficiently manage and retrieve sorted data.
- **Heaps**: A type of tree used in priority queues.
- **Expression Trees**: Trees are used to represent mathematical expressions, where each node is an operator or operand.

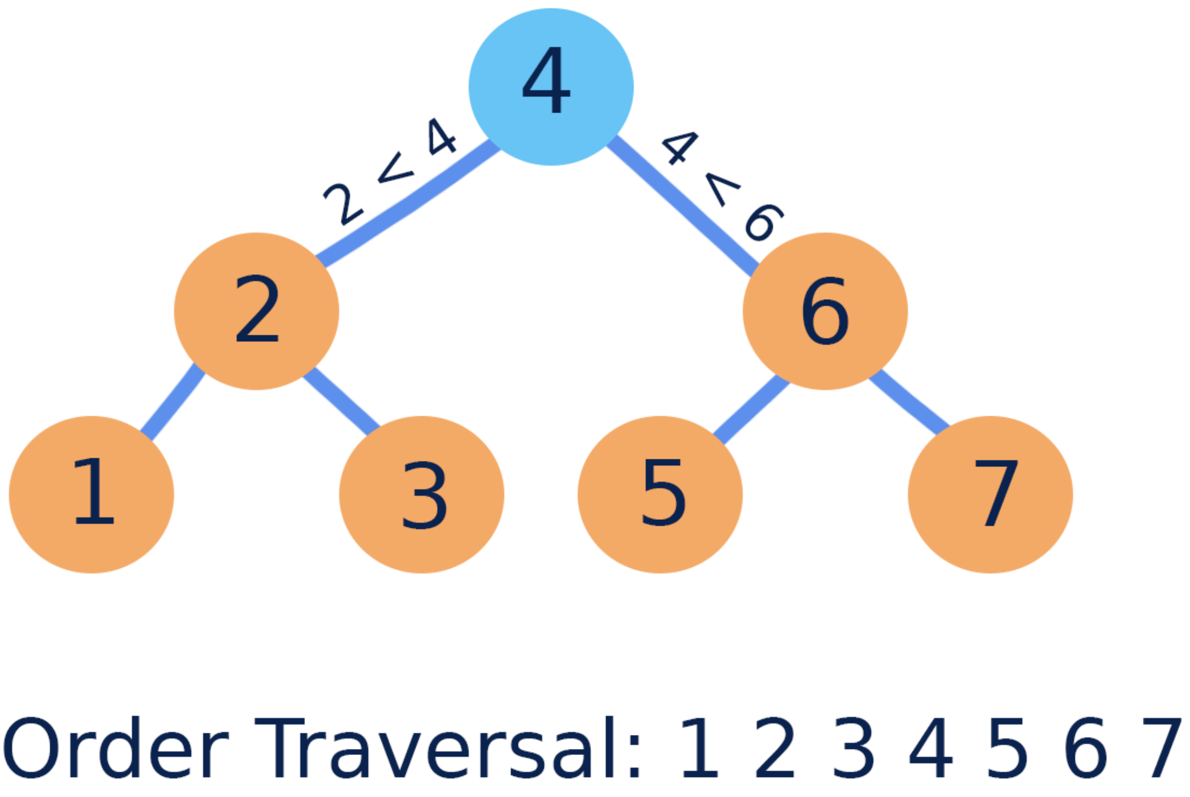
Binary Search Trees (BST)

A **Binary Search Tree (BST)** is a type of binary tree in which each node has at most two children, referred to as the left child and the right child. What distinguishes a BST from other binary trees is the binary search property, which ensures that for each node:

- All the nodes in its left subtree have values less than the node's value.
- All the nodes in its right subtree have values greater than the node's value.

Key Operations in a BST

1. **Insertion**: To insert a new node into a BST, you start at the root and compare the value of the new node with the current node. If the value is smaller, you proceed to the left child; if it is larger, you proceed to the right child. This process repeats until an appropriate null spot is found for the new node.
2. **Search**: Searching in a BST follows the same logic as insertion. Starting at the root, you compare the target value with the current node's value. You traverse the tree to the left or right, depending on whether the target is smaller or larger, until you either find the target or reach a null reference, which indicates the target is not present in the tree.
3. **Removal**: Removal in a BST is more complex because it requires maintaining the binary search property after a node is deleted. There are three cases to handle:
    - **Node is a leaf**: Simply remove the node.
    - **Node has one child**: Remove the node and connect its parent directly to its child.
    - **Node has two children**: Find the in-order successor (smallest node in the right subtree) or the in-order predecessor (largest node in the left subtree) to replace the deleted node and then adjust the tree accordingly.



Time Complexity

The time complexity of a Binary Search Tree depends on whether the tree is balanced or unbalanced:

- **Balanced BST**: O(log n) for insertion, search, and deletion. This is the ideal case, where the height of the tree is kept logarithmic relative to the number of nodes.
- **Unbalanced BST**: O(n) for insertion, search, and deletion. This occurs when the tree becomes skewed, resembling a linked list due to operations like inserting elements in sorted order.

Advantages of Binary Search Trees

- **Efficient Searching**: Due to the binary search property, a well-balanced BST allows for efficient searching, insertion, and deletion operations.
- **Sorted Order**: An in-order traversal of a BST returns the elements in sorted order, which makes it useful for sorting applications.

Disadvantages

- **Imbalance**: If the BST becomes unbalanced (for example, if the nodes are inserted in increasing or decreasing order), the tree degenerates into a linked list, and the time complexity of operations degrades to O(n).

Applications of Binary Search Trees

- **Search Applications**: BSTs are commonly used in search applications where data is constantly inserted, deleted, and queried.
- **Sorting**: In-order traversal of a BST is an efficient way to retrieve elements in a sorted manner.
- **Database Indexing**: Many databases use BSTs or variations (like B-trees) to optimize search operations.

Tree Traversal

Tree Traversal refers to the process of visiting each node in a tree data structure in a systematic way. Traversal methods are essential for performing operations such as searching, updating, or displaying the contents of a tree. There are several types of tree traversal techniques, categorized into two main groups: **Depth-First Traversal** and **Breadth-First Traversal**.

Types of Tree Traversal

1. Depth-First Traversal (DFS)

Depth-First Traversal explores as far down a branch as possible before backtracking. It is generally implemented using recursion or a stack.

The three common types of DFS tree traversals are:

- **In-Order Traversal**:
  - Process: Left subtree → Root → Right subtree
  - Application: In binary search trees (BST), an in-order traversal visits nodes in ascending order.
  - Example: For a BST, the nodes will be visited in the order that they would naturally be sorted.
- **Pre-Order Traversal**:

    - Process: Root → Left subtree → Right subtree
    - Application: Pre-order traversal is often used to create a copy of the tree or to prefix notation (Polish notation) in expressions.
    - Example: Visit the root first, then recursively visit the left subtree and finally the right subtree.

- **Post-Order Traversal**:

    - Process: Left subtree → Right subtree → Root
    - Application: Post-order traversal is useful for deleting or freeing nodes in a tree. It is also used in postfix notation (Reverse Polish notation) in expressions.
    - Example: Visit both subtrees before visiting the root node, which is useful for cleanup operations.

2. Breadth-First Traversal (BFS)

Breadth-First Traversal, also known as **Level-Order Traversal**, explores the nodes level by level, starting from the root and moving horizontally across the tree. It is typically implemented using a queue.

- **Level-Order Traversal**:
    - Process: Visit nodes level by level, starting from the root.
    - Application: BFS is useful for scenarios where we need to explore nodes closest to the root first. It is commonly used in algorithms like finding the shortest path in an unweighted tree or graph.
    - Example: In a binary tree, the nodes are visited level by level from top to bottom, starting with the root, followed by its children, and so on.

Visual Example of Tree Traversals

Consider the following tree structure:

```
      1
   /     \
  2       3
 / \     / \
4   5   6   7
```

- **In-Order Traversal**: 4 → 2 → 5 → 1 → 6 → 3 → 7
- **Pre-Order Traversal**: 1 → 2 → 4 → 5 → 3 → 6 → 7
- **Post-Order Traversal**: 4 → 5 → 2 → 6 → 7 → 3 → 1
- **Level-Order Traversal**: 1 → 2 → 3 → 4 → 5 → 6 → 7

Applications of Tree Traversal

1. **In-Order Traversal**:

    - Sorting the elements in a binary search tree.
    - Converting a tree to a sorted list.

2. **Pre-Order Traversal**:

    - Creating a copy of the tree.
    - Expression evaluation in prefix notation (Polish notation).

3. **Post-Order Traversal**:

    - Deleting a tree.
    - Expression evaluation in postfix notation (Reverse Polish notation).

4. **Level-Order Traversal**:

    - Finding the shortest path in unweighted trees or graphs.
    - Implementing algorithms like breadth-first search in graph structures.

Summary

Tree traversal methods allow you to explore all nodes in a tree systematically. The choice of traversal technique depends on the specific task at hand:

- Use **in-order traversal** when you need sorted data from a BST.
- Use **pre-order traversal** when you need to process the root before its children.
- Use **post-order traversal** when the children need to be processed before the root.
- Use **level-order traversal** when you want to explore nodes level by level.