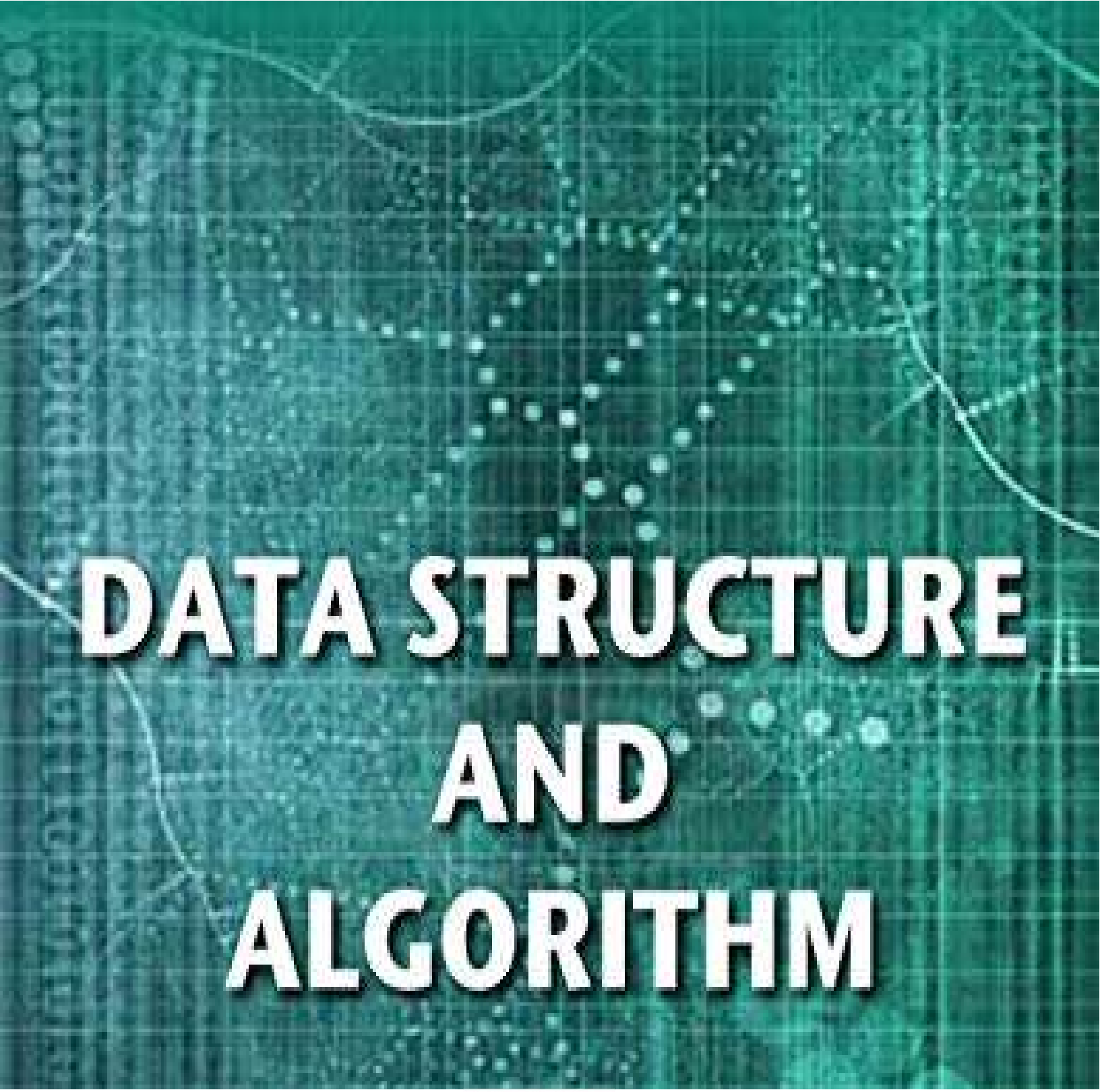


सी डैक
CDAC

BENGALURU

Data Structure & Algorithm.

What is DSA?

DSA stands for Data Structures and Algorithms, which is a fundamental concept in computer science and programming.

Data Structures:

Data structures is a way of organizing and storing data in a computer's memory so that it can be accessed and manipulated (like addition, deletion, traversal, se arching, sorting etc...) efficiently.

Types of Data Structures are as Follows. For Better understanding Follow fig 1.:

- **Linear / Basic data structures**: Data elements get stored / arranged into the memory in a linear manner (e.g. sequentially) and hence can be accessed linearly / sequentially.
  - Array- Structure and Union.
  - Linked List.
  - Stack.
  - Queue
- **Non-Linear / Advanced data structures:** Data elements gets stored / arranged into the memory in a non-linear manner (e.g. hierarchical manner) and hence can be accessed non linearly.
  - Tree (Hierarchical manner).
  - Graph.

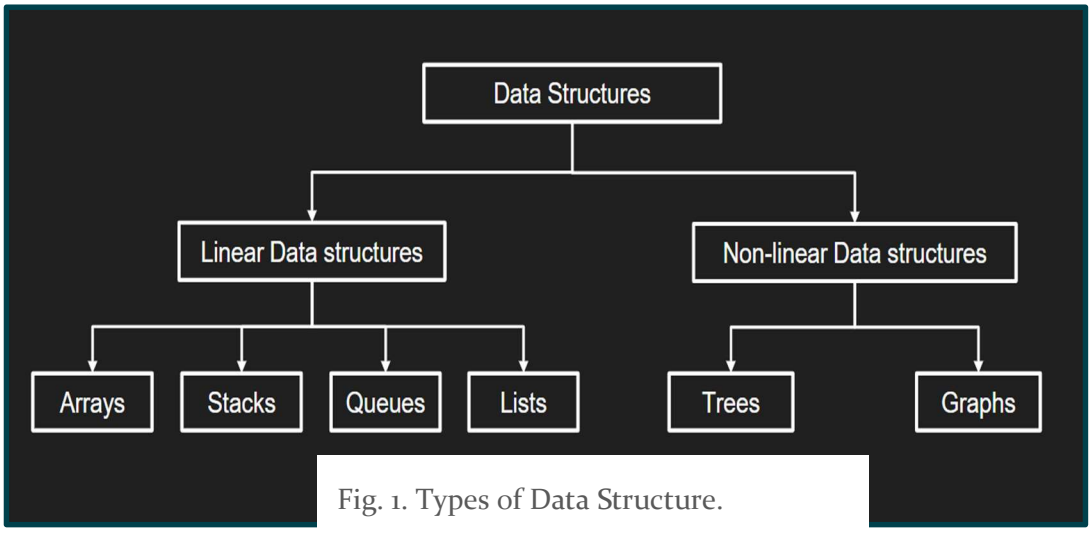


Fig. 1. Types of Data Structure.

Why there is a need of data structure?

There is a need of data structure to achieve 3 things in programming:

1. Efficiency.
2. Abstraction.
3. Reusability.

Define the problem.

- Clearly understanding the problem is the first and most crucial step in problem-solving.
- Break down the problem statement and identify the key components, such as input, output, constraints, and requirements.
- Ask clarifying questions if any part of the problem statement is ambiguous or unclear.

Identify the problem

- Recognize the type of problem you are dealing with (e.g., searching, sorting, graph traversal, dynamic programming, etc.).
- Identify the core data structures and algorithms that might be applicable to the problem.
- Determine the time and space complexity requirements, if specified.

Introduction to Problem Solving

- Problem-solving is a systematic process of finding a solution to a well-defined problem.
- It involves breaking down a complex problem into smaller, manageable parts and finding a way to solve each part.
- The goal is to develop an efficient and effective algorithm or solution that meets the given requirements.

Problem solving basics

- Understand the problem statement thoroughly, including input/output examples and edge cases.
- Identify the constraints and limitations of the problem (e.g., time complexity, space complexity, specific data structures allowed).
- Break down the problem into smaller sub-problems or tasks, if applicable.
- Explore different approaches and strategies (e.g., brute force, divide and conquer, greedy, dynamic programming, etc.).
- Consider using appropriate data structures and algorithms as building blocks for the solution.
- Analyze the time and space complexity of your proposed solution.
- Test your solution with various test cases, including edge cases and corner cases.
- Optimize and refine your solution, if necessary.

Introductory Concepts

Algorithm

- A step-by-step procedure or set of instructions to solve a problem or accomplish a task.
- Algorithms must have a clear starting point, well-defined steps, and a termination condition.
- They should be efficient, correct, and have a finite number of steps.

For Example. Write a Algorithm for Finding the Largest No. from the given 3 numbers.

Solution:

1. Start with three input numbers: a, b, and c.
2. Compare a and b.
   If a is greater than b, proceed to step 3.
   If b is greater than or equal to a, proceed to step 5.
3. Compare a and c.
4. If a is greater than c, then a is the largest number. Output "a is the largest" and terminate the algorithm.
5. If c is greater than or equal to a, then c is the largest number. Output "c is the largest" and terminate the algorithm.
6. (This step is skipped because the algorithm has already terminated in step 3.)
7. Compare b and c.
8. If b is greater than c, then b is the largest number. Output "b is the largest" and terminate the algorithm.
9. If c is greater than or equal to b, then c is the largest number. Output "c is the largest" and terminate the algorithm.
10. End of the algorithm.

Flow-Chart:

A Flowchart is a type of Diagram that represents a workflow or process. A flowchart can also be defined as Diagrammatic representation of an Algorithm, a step-by-step approach to solving a task.

Fig 2. Shows the Flow-Chart for a program to Find largest no. from the 3 given no.

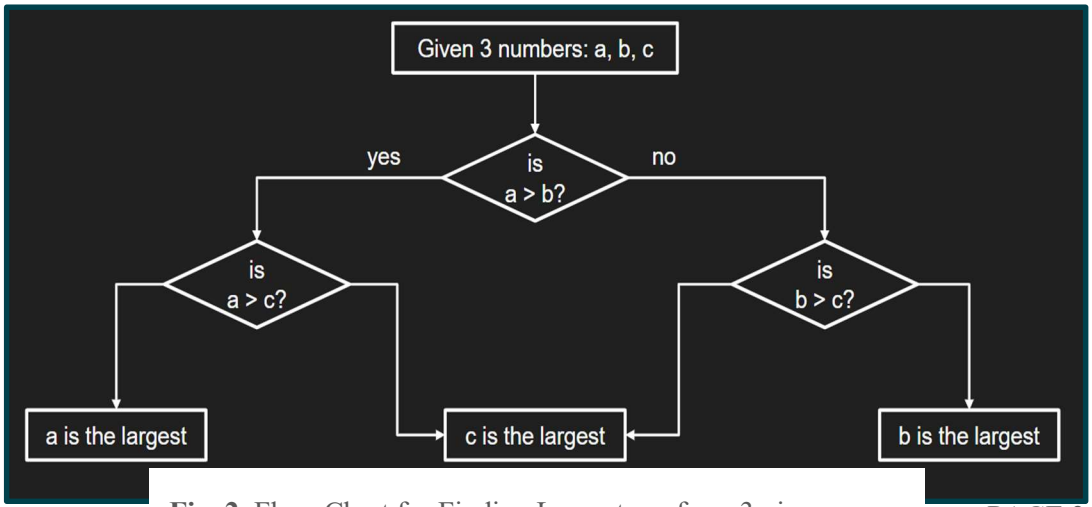


Fig. 2. Flow-Chart for Finding Largest no. from 3 given nos.

Pseudo-Code:

Pseudocode is an informal way of describing an algorithm or a computer program using a combination of natural language and programming language constructs. It provides a high-level, human-readable representation of the logic and flow of an algorithm without adhering to the strict syntax rules of any specific programming language. The pseudocode for provided Fig 2. (an algorithm that finds the largest among three given numbers (a, b, and c)). Here's an explanation of the pseudocode using the definition and characteristics of pseudocode:

```
FUNCTION findLargest(a, b, c)
    IF a > b AND a > c
        RETURN "a is the largest"
    ELSE IF b > a AND b > c
        RETURN "b is the largest"
    ELSE
        RETURN "c is the largest"
    END IF
END FUNCTION
```

In this pseudocode:

1. FUNCTION “findLargest(a, b, c)” defines a function named “findLargest” that takes three parameters ‘a,’ ‘b’, and ‘c’, which represent the three numbers to be compared.
2. IF a > b is a conditional statement that checks if the value of a is greater than the value of b.
3. If the condition a > b is true, it proceeds to the next nested IF statement: IF a > c. This checks if the value of a is also greater than the value of c.
    - If a > c is true, it means a is the largest among the three numbers, so the pseudocode RETURN "a is the largest" indicates that the function should return the string "a is the largest".
    - If a > c is false (i.e., c is greater than or equal to a), it means c is the largest, so the pseudocode RETURN "c is the largest" indicates that the function should return the string "c is the largest".
4. If the initial condition a > b is false (i.e., b is greater than or equal to a), it proceeds to the ELSE block, which checks IF b > c.

- If b > c is true, it means b is the largest among the three numbers, so the pseudocode RETURN "b is the largest" indicates that the function should return the string "b is the largest".
- If b > c is false (i.e., c is greater than or equal to b), it means c is the largest, so the pseudocode RETURN "c is the largest" indicates that the function should return the string "c is the largest".

5. END IF and END FUNCTION are used to mark the end of the conditional blocks and the function definition, respectively.

This pseudocode effectively describes the logic of finding the largest among three given numbers by comparing them using a series of conditional statements. It uses a combination of natural language descriptions and programming language constructs like FUNCTION, IF, ELSE, and RETURN to convey the algorithm in a clear and human-readable manner.

Complexity analysis of algorithms (Big-O notation):

Complexity refers to the analysis of the time and space requirements of an algorithm or a computational problem. Understanding complexity is crucial for designing efficient algorithms and evaluating their performance. Here are some key points about complexity:

1. Time Complexity

- Measures how the running time of an algorithm grows as the input size increases.
- Expressed using Big O notation (e.g., O(n), O(n^2), O(log n), etc.).
- Common time complexities:

  - O(1) - Constant time (ideal, does not depend on input size)
  - O(log n) - Logarithmic time (efficient for divide-and-conquer algorithms)
  - O(n) - Linear time (common for iterating through data structures)
  - O(n log n) - Linearithmic time (efficient sorting algorithms like Merge Sort)
  - O(n^2) - Quadratic time (inefficient nested loops)
  - O(2^n) - Exponential time (very inefficient, avoidable in most cases)

2. Space Complexity

- Measures how the memory usage of an algorithm grows as the input size increases.
- Also expressed using Big O notation.
- Common space complexities:

  - O(1) - Constant space (ideal, does not depend on input size)
  - O(n) - Linear space (common for data structures like arrays, linked lists)
  - O(n^2) - Quadratic space (inefficient, usually avoidable)
  - O(log n) - Logarithmic space (efficient for recursive algorithms)

3. **Analyzing Complexity**

- Identifying the time and space complexities of an algorithm is crucial for understanding its efficiency and scalability.
- Best-case, worst-case, and average-case complexities should be considered.
- Auxiliary space complexity (temporary space used by an algorithm) is also important.
- Optimizing algorithms often involves trading off time complexity for space complexity, or vice versa.

Importance of Complexity Analysis:

- Allows comparing the efficiency of different algorithms for the same problem.
- Helps determine the scalability of an algorithm for large input sizes.
- Guides the selection of appropriate algorithms and data structures for specific problems.
- Aids in optimizing algorithms by identifying performance bottlenecks.

Q. What is Best, Worst, Average Case?

1. **Best Case**:

   - The best case represents the most favorable situation for the algorithm, where the input data is arranged or structured in a way that allows the algorithm to perform optimally.
   - In the best case, the algorithm typically exhibits the lowest time and space complexity.
   - For example, in the case of a sorting algorithm like Quicksort, the best case occurs when the pivot element divides the array into two nearly equal parts, resulting in a time complexity of O(n log n).

2. **Worst Case**:

   - The worst case represents the most unfavorable situation for the algorithm, where the input data is arranged or structured in a way that causes the algorithm to perform poorly.
   - In the worst case, the algorithm typically exhibits the highest time and space complexity.
   - For example, in the case of a sorting algorithm like Bubble Sort, the worst case occurs when the input array is in reverse order, resulting in a time complexity of O(n^2).

3. **Average Case**:

   - The average case represents the expected or typical performance of the algorithm, assuming that the input data is randomly distributed.
   - The average case complexity is often more challenging to analyze than the best and worst cases, as it requires considering the probability distribution of the input data.
   - For many algorithms, the average case complexity is the most relevant metric, as it represents the algorithm's performance in real-world scenarios.

Memory Allocation:

In Java, memory allocation works similarly to other programming languages, but with some differences due to the managed nature of the Java Virtual Machine (JVM) and the automatic memory management handled by the garbage collector.

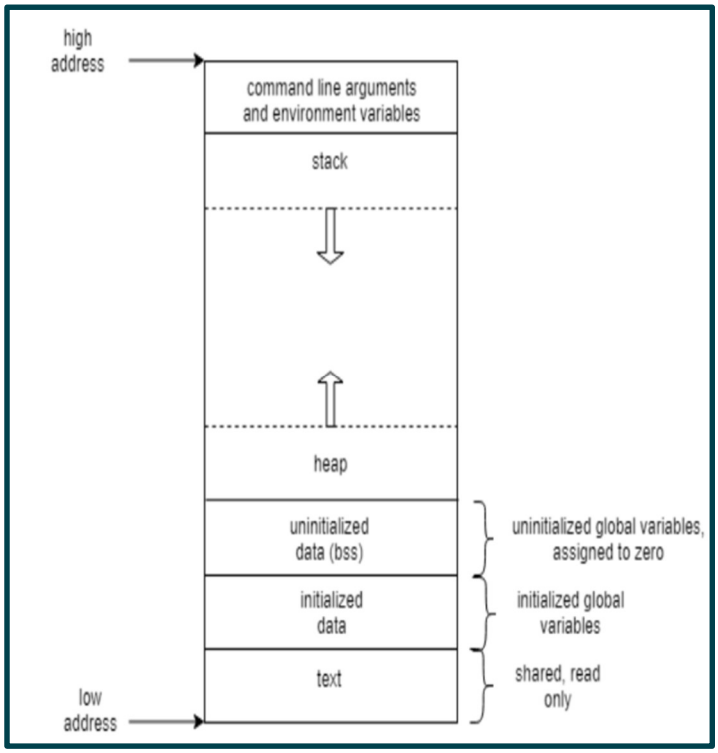
Here's how memory allocation works in Java:

1) **Stack Memory:**
   - The JVM creates a stack for each thread in the program.
   - The stack is used to store local variables and method call information (like arguments, return addresses, etc.)
   - When a method is called, a new stack frame is created and pushed onto the stack.
   - When a method returns, its stack frame is popped off the stack.
   - The stack follows a Last-In-First-Out (LIFO) order.
   - Variables stored on the stack have a limited lifetime and are automatically deallocated when the method returns.

2) **Heap Memory:**
   - The heap is a shared memory area used for dynamic memory allocation.
   - Objects and arrays are stored on the heap.
   - When you create an object using the `new` keyword, the JVM allocates memory for that object on the heap.
   - Objects on the heap have a longer lifetime and can be accessed by multiple threads.
   - The garbage collector automatically manages and deallocates objects on the heap that are no longer referenced.

3) **Method Area (or Permanent Generation):**
   - The method area is a shared memory area used to store class-level data, like class code, static variables, and constants.
   - This area is created when the JVM starts and remains until the JVM terminates.
   - Data in the method area is shared among all threads.

Here's a visual representation of memory allocation in Java:



In Java, you don't have direct access to low-level memory allocation and deallocation like in languages like C or C++. Instead, the JVM manages memory allocation and deallocation for you, including:

- Allocating memory for objects and arrays on the heap using the `new` keyword.
- Automatically deallocating memory for objects on the heap that are no longer referenced, through the garbage collector.
- Allocating and deallocating memory for local variables and method call information on the stack.

The Java memory management model simplifies memory management for developers and helps reduce common memory-related issues like memory leaks and buffer overflows. However, it's still important to understand memory allocation concepts and write efficient code to avoid excessive memory usage and improve performance.

OO design: Abstract Data Types (ADTs)

In object-oriented design, an Abstract Data Type (ADT) is a conceptual model or a high-level description of a data structure and its associated operations. ADTs define the logical behavior and properties of a data structure without specifying the implementation details.

ADT: ADT is a set of data values and the operations that can be performed on those data values. It specifies the contract or interface of the data structure, without revealing the underlying implementation.

ADT consists of two main components:

1. Data: The set of values or objects that the ADT can store or represent.
2. Operations: The set of operations or methods that can be performed on the data values

Abstraction and Encapsulation:

1. ADTs provide abstraction by hiding the implementation details from the user.
2. They encapsulate the data representation and the implementation of operations within the data structure.

Examples:

Common examples of ADTs include:

1. List (insert, remove, search, etc.)
2. Stack (push, pop, peek, etc.)
3. Queue (enqueue, dequeue, front, etc.)
4. Set (insert, remove, contains, etc.)
5. Map or Dictionary (put, get, remove, etc.)

Benefits:

1. ADTs promote code reusability, modularity, and maintainability.
2. They allow for multiple implementations of the same ADT, as long as they follow the specified contract.
3. ADTs provide a clear separation between the interface and implementation, enabling easier modification and extension.

Implementation:

1. ADTs can be implemented using various data structures, such as arrays, linked lists, trees, or hash tables.
2. The choice of implementation depends on factors like performance requirements, memory constraints, and the specific use case.

Basic Data Structures

1.Arrays

- An array is a collection of elements of the same data type stored in contiguous memory locations.
- Arrays provide constant-time access to elements by index.
- Basic operations: accessing an element, traversing, inserting, and deleting elements.
- Static arrays have a fixed size, while dynamic arrays can resize dynamically.
- Example of Static.

```java
// Creating an array
int[] arr = new int[5]; // Array of size 5

// Assigning values
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;

// Accessing array elements
System.out.println("Element at index 2: " + arr[2]); // Output: 30

// Iterating over the array
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + " "); // Output: 10 20 30 40 50
}
```

- Example Dynamic Array.

```java
import java.util.Arrays;
public class ArraySS  2 usages
{
    private int [] item;  9 usages
    private int count;  3 usages
    public ArraySS(int lenght)  1 usage
    {
        item = new int[lenght];
    }
    public void insert(int i)  5 usages
    {
        if (item.length==count)//if the array is full resize it
        {
            //create an array
            int [] newItem = new int[item.length*2];
            //copy the array
            System.arraycopy(item,  srcPos: 0, newItem,  destPos: 0, item.length);
            //set item to new array
            item = newItem;

        }
        item[count++]=i; //Adding an item at the end
    }
    public void print()  1 usage
    {
        for (int i = 0; i < count; i++)
        {
            System.out.println( item [i] );
        }
        System.out.print(Arrays.toString(item) );
    }
}

```

```java
public class Main {
    public static void main(String[] args)
    {
        ArraySS row = new ArraySS( lenght: 3);
        row.insert( i: 10);
        row.insert( i: 20);
        row.insert( i: 30);
        row.insert( i: 40);
        row.insert( i: 50);
        row.print();
    }
}
```

Output:

```
10
20
30
40
50
[10, 20, 30, 40, 50, 0]
Process finished with exit code 0
```

2.Stacks

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle.
- Basic operations: push (insert at the top), pop (remove from the top), peek (get the top element), and is Empty.
- Stacks can be implemented using arrays or linked lists.
- Applications: function call stack, expression evaluation, backtracking algorithms.

```java
// Creating a stack
Stack<Integer> stack = new Stack<>();

// Push elements onto the stack
stack.push(10);
stack.push(20);
stack.push(30);

// Peek the top element
System.out.println("Top element: " + stack.peek()); // Output: 30

// Pop an element from the stack
System.out.println("Popped element: " + stack.pop()); // Output: 30

// Check if the stack is empty
System.out.println("Is stack empty? " + stack.isEmpty()); // Output: false
```

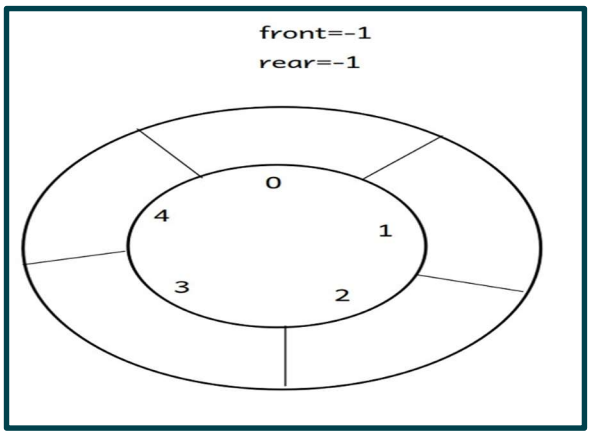
3.Queues

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle.
- Basic operations: enqueue (insert at the rear), dequeue (remove from the front), front (get the front element), and is Empty.
- Queues can be implemented using arrays or linked lists.
- Applications: job scheduling, printer spooling, breadth-first search.

```java
// Creating a queue
Queue<String> queue = new LinkedList<>();

// Add elements to the queue
queue.offer("Apple");
queue.offer("Banana");
queue.offer("Cherry");

// Retrieve and remove the front element
System.out.println("Front element: " + queue.poll()); // Output: Apple

// Peek the front element without removing it
System.out.println("Front element: " + queue.peek()); // Output: Banana

// Check if the queue is empty
System.out.println("Is queue empty? " + queue.isEmpty()); // Output: false
```

4.Circular Queues

- A circular queue is a variation of a regular queue where the last position is connected to the first position, forming a circle.
- This allows efficient use of the underlying array or linked list by reusing the empty spaces in a circular manner.
- Circular queues can handle overflow conditions more gracefully than regular queues.



```
class CircularQueue {
    private int[] arr;
    private int front, rear, size, capacity;

    CircularQueue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = rear = -1;
        size = 0;
    }

    boolean isFull() {
        return (size == capacity);
    }

    boolean isEmpty() {
        return (size == 0);
    }
```

```
    void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = x;
        if (front == -1)
            front = 0;
        size++;
    }

    int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        int x = arr[front];
        if (front == rear)
            front = rear = -1;
        else
            front = (front + 1) % capacity;
        size--;
        return x;
    }
}

// Usage example
CircularQueue queue = new CircularQueue(5);
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
System.out.println("Front element: " + queue.dequeue()); // Output: 10
System.out.println("Front element: " + queue.dequeue()); // Output: 20
```

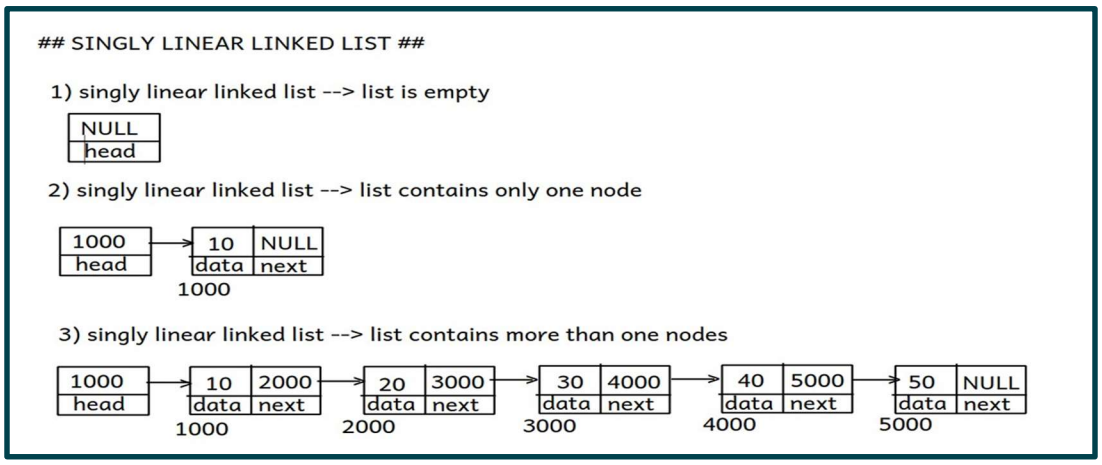
Linked lists

- A linked list is a linear data structure where elements are not stored in contiguous memory locations.
- Each element (node) contains data and a reference (link) to the next node in the sequence.

1) Singly linked lists

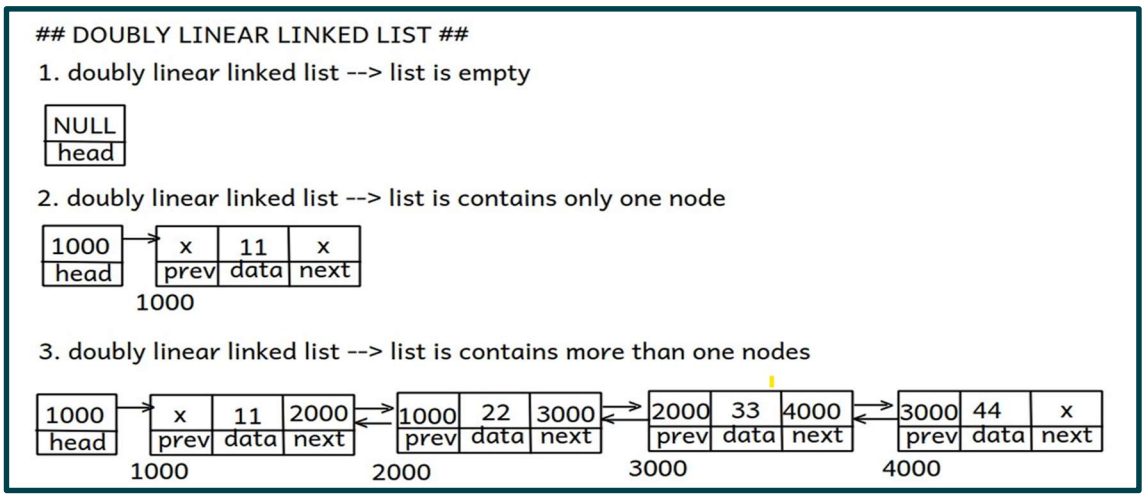
- In a singly linked list, each node contains data and a single reference to the next node.
- Basic operations: insert at the beginning, insert at the end, insert after a given node, delete a node, and traverse the list.

```
// Node class to represent a node in the linked list
class Node {
    int data; // Data stored in the node
    Node next; // Reference to the next node

    Node(int data) {
        this.data = data;
        next = null; // Initially, the next pointer is null
    }
}

class SinglyLinkedList {
    Node head; // Head of the linked list

    // Method to insert a new node at the end of the linked list
    void insertAtEnd(int data) {
        Node newNode = new Node(data); // Create a new node with the given data

        // If the list is empty, make the new node as the head
        if (head == null) {
            head = newNode;
            return;
        }

        // Traverse to the last node
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }

        // Insert the new node at the end
        current.next = newNode;
    }

    // Method to print the linked list
    void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next; // Move to the next node
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insertAtEnd(10); // Insert 10 at the end
        list.insertAtEnd(20); // Insert 20 at the end
        list.insertAtEnd(30); // Insert 30 at the end
        list.printList(); // Print the list: 10 20 30
    }
}
```



2) Doubly linked lists

- In a doubly linked list, each node contains data and two references: one to the next node and one to the previous node.
- This allows traversal in both forward and backward directions.
- Basic operations are similar to singly linked lists, with the addition of backward traversal and deletion based on the previous node reference.

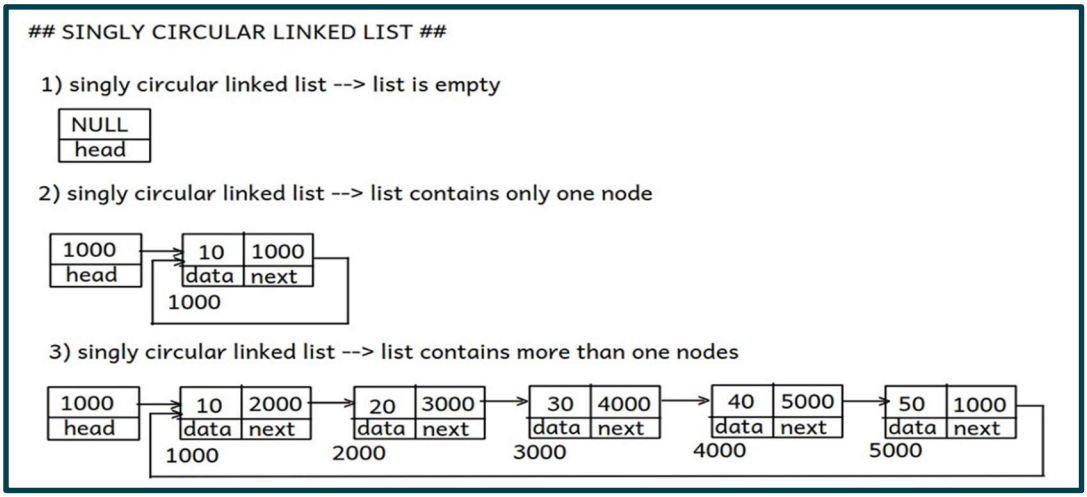


```java
// Node class to represent a node in the doubly linked list
class Node {
    int data; // Data stored in the node
    Node prev; // Reference to the previous node
    Node next; // Reference to the next node

    Node(int data) {
        this.data = data;
        prev = null; // Initially, the previous pointer is null
        next = null; // Initially, the next pointer is null
    }
}

class DoublyLinkedList {
    Node head, tail; // Head and tail of the doubly linked list

    // Method to insert a new node at the end of the doubly linked list
    void insertAtEnd(int data) {
        Node newNode = new Node(data); // Create a new node with the given data

        // If the list is empty, make the new node as both head and tail
        if (head == null) {
            head = tail = newNode;
            return;
        }

        // Insert the new node at the end
        tail.next = newNode; // Update the next pointer of the current tail
        newNode.prev = tail; // Update the previous pointer of the new node
        tail = newNode; // Make the new node as the new tail
    }

    // Method to print the doubly linked list
    void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next; // Move to the next node
        }
        System.out.println();
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        list.insertAtEnd(10); // Insert 10 at the end
        list.insertAtEnd(20); // Insert 20 at the end
        list.insertAtEnd(30); // Insert 30 at the end
        list.printList(); // Print the list: 10 20 30
    }
}
```

3) Singly Circular linked lists

- A singly circular linked list is a variation of a singly linked list where the last node's next pointer points back to the first node (head), forming a circular loop. This means that the list has no null-terminator at the end, and the nodes are connected in a circular manner.
- In a singly circular linked list, the next pointer of the last node points to the head node, creating a circular loop. This allows traversal of the list to continue from the last node back to the first node, making it easier to insert or remove nodes at the beginning or end of the list.

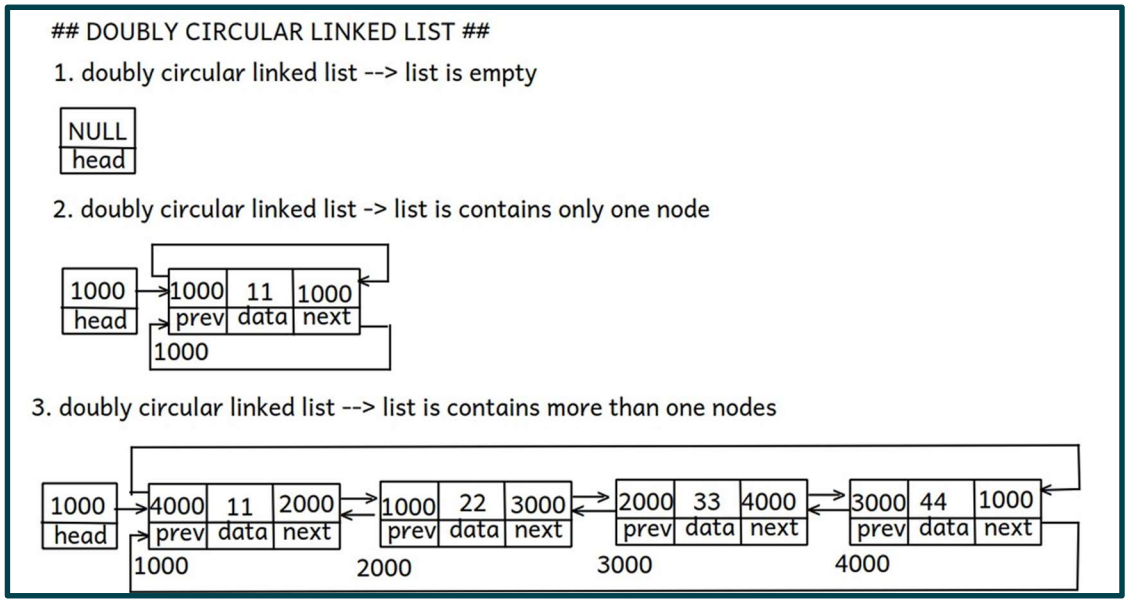
- The main advantage of a singly circular linked list over a regular singly linked list is that it simplifies the task of inserting or removing nodes at the end of the list, as the last node can be easily accessed through the circular loop.




```java
class Node {
    int data; // Data stored in the node
    Node next; // Reference to the next node

    Node(int data) {
        this.data = data;
        next = null; // Initially, the next pointer is null
    }
}

class SinglyCircularLinkedList {
    Node head; // Head of the singly circular linked list

    // Method to insert a new node at the end of the singly circular linked list
    void insertAtEnd(int data) {
        Node newNode = new Node(data); // Create a new node with the given data

        // If the list is empty, make the new node as the head and point to itself
        if (head == null) {
            head = newNode;
            newNode.next = head;
            return;
        }

        // Traverse to the last node
        Node current = head;
        while (current.next != head) {
            current = current.next;
        }

        // Insert the new node at the end and make it point to the head
        current.next = newNode;
        newNode.next = head;
    }

    // Method to print the singly circular linked list
    void printList() {
        if (head == null) {
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " ");
            current = current.next; // Move to the next node
        } while (current != head); // Loop until we reach the head again
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyCircularLinkedList list = new SinglyCircularLinkedList();
        list.insertAtEnd(10); // Insert 10 at the end
        list.insertAtEnd(20); // Insert 20 at the end
        list.insertAtEnd(30); // Insert 30 at the end
        list.printList(); // Print the list: 10 20 30
    }
}
```

4) **Doubly Circular Linked List**:

- A doubly circular linked list is a variation of a doubly linked list where the next pointer of the last node points to the head node, and the prev pointer of the head node points to the last node, creating a circular loop in both directions.
- In a doubly circular linked list, each node has two pointers: next and prev. The next pointer points to the next node in the sequence, and the prev pointer points to the previous node. The last node's next pointer points to the head node, and the head node's prev pointer points to the last node, creating a circular loop in both directions.
- The main advantage of a doubly circular linked list over a regular doubly linked list is that it simplifies the task of inserting or removing nodes at the beginning or end of the list, as the first and last nodes can be easily accessed through the circular loop in both directions.



```java
// Node class to represent a node in the doubly circular linked list
class Node {
    int data; // Data stored in the node
    Node prev; // Reference to the previous node
    Node next; // Reference to the next node

    Node(int data) {
        this.data = data;
        prev = null; // Initially, the previous pointer is null
        next = null; // Initially, the next pointer is null
    }
}

class DoublyCircularLinkedList {
    Node head; // Head of the doubly circular linked list

    // Method to insert a new node at the end of the doubly circular linked list
    void insertAtEnd(int data) {
        Node newNode = new Node(data); // Create a new node with the given data

        // If the list is empty, make the new node as the head and point to itself
        if (head == null) {
            head = newNode;
            newNode.next = newNode;
            newNode.prev = newNode;
            return;
        }

        // Insert the new node at the end
        Node last = head.prev; // Get the last node
        last.next = newNode; // Update the next pointer of the last node
        newNode.prev = last; // Update the previous pointer of the new node
        newNode.next = head; // Make the new node's next pointer point to the head
        head.prev = newNode; // Update the previous pointer of the head
    }

    // Method to print the doubly circular linked list
    void printList() {
        if (head == null) {
            return;
        }

        Node current = head;
        do {
            System.out.print(current.data + " ");
            current = current.next; // Move to the next node
        } while (current != head); // Loop until we reach the head again
        System.out.println();
    }

    public static void main(String[] args) {
        DoublyCircularLinkedList list = new DoublyCircularLinkedList();
        list.insertAtEnd(10); // Insert 10 at the end
        list.insertAtEnd(20); // Insert 20 at the end
        list.insertAtEnd(30); // Insert 30 at the end
        list.printList(); // Print the list: 10 20 30
    }
}
```

5) Node-based storage with arrays

- Node-based storage with arrays is a technique where nodes are stored in an array, and each node contains an index to the next node in the array.
- This approach combines the benefits of arrays (constant-time access) and linked lists (dynamic memory allocation).
- It is useful when the maximum number of nodes is known in advance and memory usage needs to be optimized.

```
class Node {
    int data;
    int next; // Index of the next node in the array

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

class NodeStorage {
    private static final int MAX_NODES = 100; // Maximum number of nodes
    private Node[] nodes = new Node[MAX_NODES]; // Array to store nodes
    private int freeIndex = 0; // Index of the first free slot in the array
    private int head = -1; // Index of the head node

    public void insert(int data) {
        if (freeIndex >= MAX_NODES) {
            System.out.println("Storage is full");
            return;
        }

        nodes[freeIndex] = new Node(data, -1); // Create a new node with -1 as the next index
        if (head == -1) {
            head = freeIndex; // If the list is empty, make the new node the head
        } else {
            int curr = head;
            while (nodes[curr].next != -1) {
                curr = nodes[curr].next; // Traverse to the last node
            }
            nodes[curr].next = freeIndex; // Update the next pointer of the last node
        }
        freeIndex++; // Move to the next free slot
    }

    public void print() {
        int curr = head;
        while (curr != -1) {
            System.out.print(nodes[curr].data + " ");
            curr = nodes[curr].next; // Move to the next node
        }
        System.out.println();
    }

    public static void main(String[] args) {
        NodeStorage storage = new NodeStorage();
        storage.insert(10);
        storage.insert(20);
        storage.insert(30);
        storage.print(); // Output: 10 20 30
    }
}
```

What is recursion?

Recursion is a programming technique where a function calls itself with a smaller input or a slightly different set of parameters to make the problem smaller. It is a way of solving complex problems by breaking them down into simpler instances of the same problem.

```java
package DSA;

import java.util.Scanner;

public class Fibonacci
{
	public static void main(String[] args)
	{
		System.out.print("Enter the Series Upto You Want : ");
		@SuppressWarnings("resource")
		Scanner scan = new Scanner(System.in); // Using scanner for taking input from user
		int n= scan.nextInt(); // taking value from user and storing in Variable n

		System.out.println("Your Fabonacci Series is: \n");

		for (int i=0 ; i<n ; i++)
		{
			System.out.print(fib(i)+" "); //While printing the value Function is called and value is passed
		}
	}

	public static int fib(int n)  // function for finding Fibonacci Series
	{
		if (n==0 || n==1)
			return n;
		else
			return fib(n-1)+fib(n-2); // Recursion occurs as the function is calling itself.
	}

}
```

Memoization:

- It is an optimization technique used primarily to speed up computer program by caching the result of expensive function calls and returning the cached result when the same input occurs again.
- Ex:
  - FabanocciByMethodMemoization (int n);
  - Int [] array = new int [100];

What is the base condition in recursion?

The base condition, also known as the base case or stopping condition, is a crucial part of any recursive function. It defines the termination condition for the recursion, ensuring that the function does not call itself indefinitely. The base condition is typically a simple case where the problem can be solved without further recursion.

Direct and indirect recursion.

- **Direct Recursion**: A function is directly recursive if it calls itself within its body.
- **Indirect Recursion**: A set of functions are indirectly recursive if function A calls function B, which in turn calls function C, and so on, until one of the functions eventually calls function A again, forming a recursive loop.

Memory is allocated to different function calls in recursion.

In recursion, each recursive call creates a new instance of the function with its own set of parameters and local variables. This means that memory is allocated on the call stack for each recursive call, allowing the function to maintain its own state and data. When the base case is reached, the recursive calls start unwinding, and the memory allocated for each call is deallocated (released) from the call stack.

Pro and cons of recursion

Pros:

1. Recursion provides a clean and elegant solution to many problems, making the code more readable and easier to understand.
2. Recursive solutions can be more concise and expressive compared to iterative solutions for certain problems.
3. Recursion is well-suited for problems that can be naturally broken down into smaller instances of the same problem.

Cons:

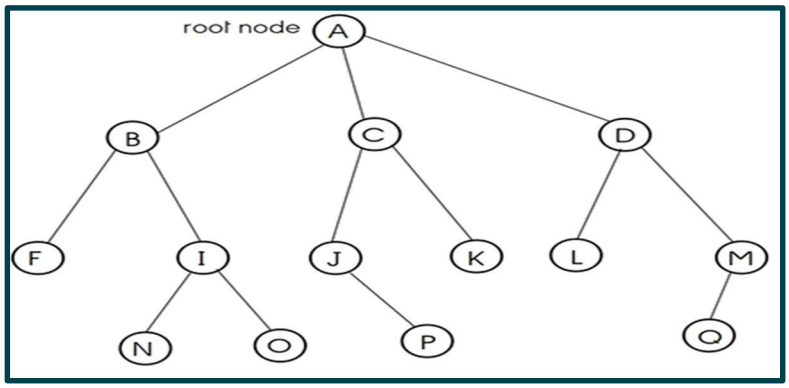
1. Recursive functions may have a higher overhead due to the repeated function calls and memory allocation on the call stack.
2. Recursive solutions can lead to stack overflow errors if the recursion goes too deep and exceeds the available call stack space.
3. Debugging recursive functions can be more challenging compared to iterative solutions.

Function complexity during recursion

The time and space complexity of a recursive function depend on the number of recursive calls made and the work done in each call. The time complexity is determined by the number of recursive calls and the operations performed in each call, while the space complexity is determined by the maximum depth of the recursion (the maximum number of active recursive calls on the call stack at any given time).

Introduction to trees

A tree is a non-linear hierarchical data structure that consists of nodes connected by edges. It is a collection of entities called nodes, linked together to represent a hierarchy. Trees are widely used in computer science for data organization, manipulation, and storage.



Trees and terminology

- **Root**: The topmost node in the tree hierarchy.
- **Parent Node**: A node that has one or more child nodes.
- **Child Node**: A node that has a parent node.
- **Leaf Node**: A node that has no children.
- **Sibling Nodes**: Nodes that share the same parent.
- **Depth**: The number of edges from the root node to a particular node.
- **Height**: The number of edges in the longest path from the root node to a leaf node.

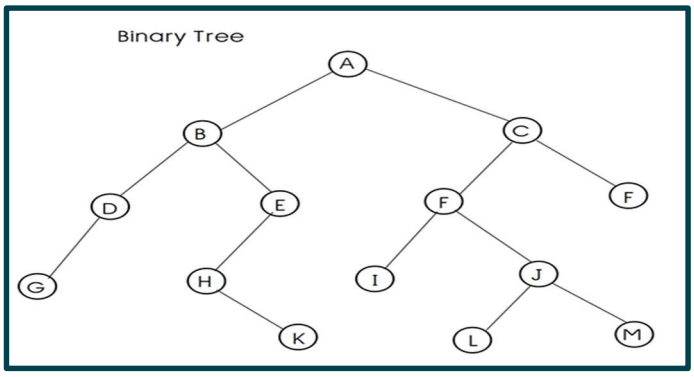
Tree traversals

Tree traversal refers to the process of visiting or accessing each node in the tree exactly once. The three primary tree traversal techniques are:

1. **Preorder Traversal**: Visit the root node, then recursively traverse the left subtree, and finally the right subtree.
2. **In-Order Traversal**: Recursively traverse the left subtree, visit the root node, and then recursively traverse the right subtree.
3. **Post-Order Traversal**: Recursively traverse the left subtree, then the right subtree, and finally visit the root node.

Binary trees

A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.



Complete binary trees / Almost complete binary tree (ACBT)

- **Complete Binary Tree**: A binary tree in which all levels are completely filled except possibly the last level, and all nodes on the last level are as far left as possible.
- **Almost Complete Binary Tree (ACBT)**: A binary tree in which all levels are completely filled except possibly the last level, and all nodes on the last level are as far left as possible, with the exception of one node.

Array implementation of A.C.B.T

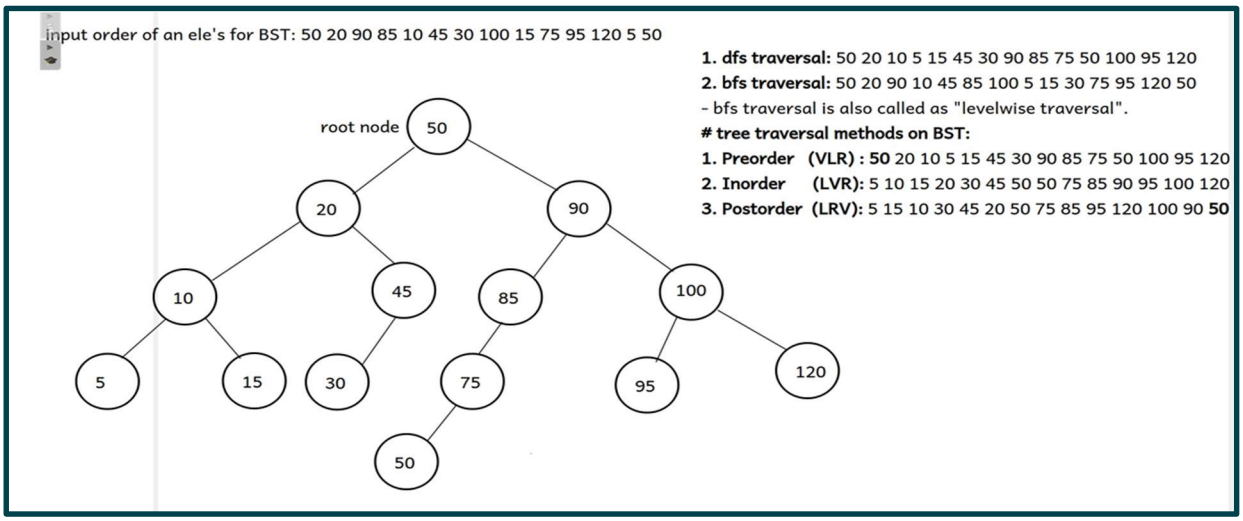
An Almost Complete Binary Tree (ACBT) can be efficiently represented using an array. The root node is stored at index 0, and for any node at index i, its left child is at index 2i+1, and its right child is at index 2i+2. This implementation allows constant-time access to the children of a node and efficient memory utilization.

Binary search trees

A binary search tree (BST) is a binary tree with the following properties:

1. The value of each node is greater than or equal to the values in its left subtree.
2. The value of each node is less than or equal to the values in its right subtree.
3. The left and right subtrees must also be binary search trees.

BSTs are commonly used for efficient searching, insertion, and deletion operations, with an average time complexity of O(log n) for these operations.



AVL tree

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. This property allows the AVL tree to maintain a balanced structure, ensuring that all operations (search, insertion, deletion) have a time complexity of O(log n), where n is the number of nodes in the tree.

Objectives of Searching

The primary objectives of searching algorithms are:
1. To determine whether a given element is present in a data structure or not.
2. To locate the position or index of an element in a data structure, if it exists.

1. The Sequential Search

The sequential search (or linear search) is a simple searching algorithm that iterates over a data structure (e.g., an array or a linked list) and compares each element with the target value until a match is found or the end of the data structure is reached.

2. Analysis of Sequential Search

The time complexity of the sequential search algorithm depends on the position of the target element (if present) or whether it is absent. In the worst case, when the element is not present or is at the end of the data structure, the time complexity is O(n), where n is the size of the data structure.

3. The Binary Search

The binary search is an efficient searching algorithm that works on sorted arrays or sorted data structures. It repeatedly divides the search interval in half by comparing the target value with the middle element until the target is found or the search interval becomes empty.

```java
package Dsa;

public class BinarySearchArray {
    public int binarySearch(int temp, int[] c) {
        int low = 0, high = c.length - 1, mid = 0;
        while (low <= high) {
            mid = (low + high) / 2; // compute the mid array
            if (temp > c[mid]) {
                low = mid + 1; // if true implies that element is in the right half
            } else if (temp < c[mid]) {
                high = mid - 1; // if true implies that element is in the left half
            } else { // if true implies that element in search is present at the Center
                return mid;
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] array = {4, 9, 11, 16, 19, 21, 25, 28};
        BinarySearchArray binarySearch = new BinarySearchArray();
        int index = binarySearch.binarySearch(21, array);
        System.out.println("Index of 21 in the array: " + index); // Output: Index of 21
    }
}
```

Analysis of Binary Search:

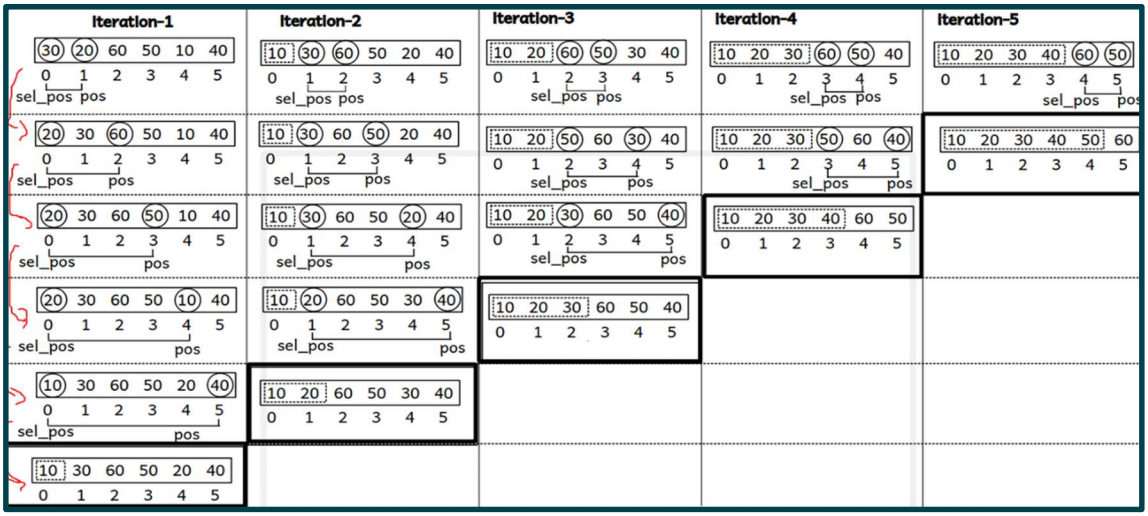
The time complexity of the binary search algorithm is O(log n) in the average and best cases, where n is the size of the sorted data structure. This makes it significantly more efficient than the sequential search for large data sets.

Introduction to Sorting

Sorting is the process of arranging elements in a specific order (ascending or descending) based on a comparison operation. Sorting algorithms are fundamental in computer science and are used in various applications, such as database operations, data processing, and numerical analysis.

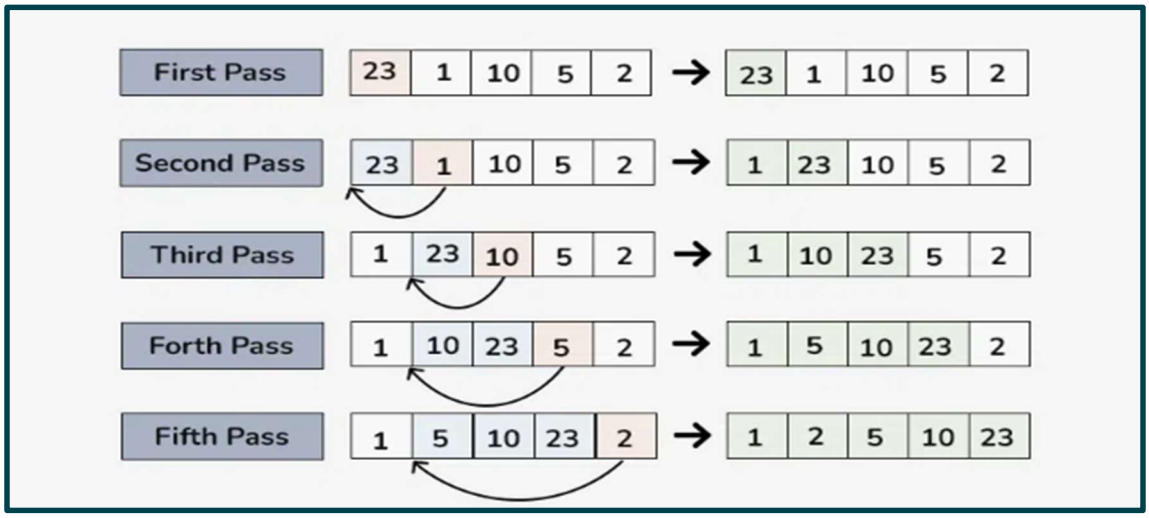
1. Selection Sort

Selection sort is an in-place comparison-based sorting algorithm that works by repeatedly finding the minimum element from the unsorted part and swapping it with the first element of the unsorted part.



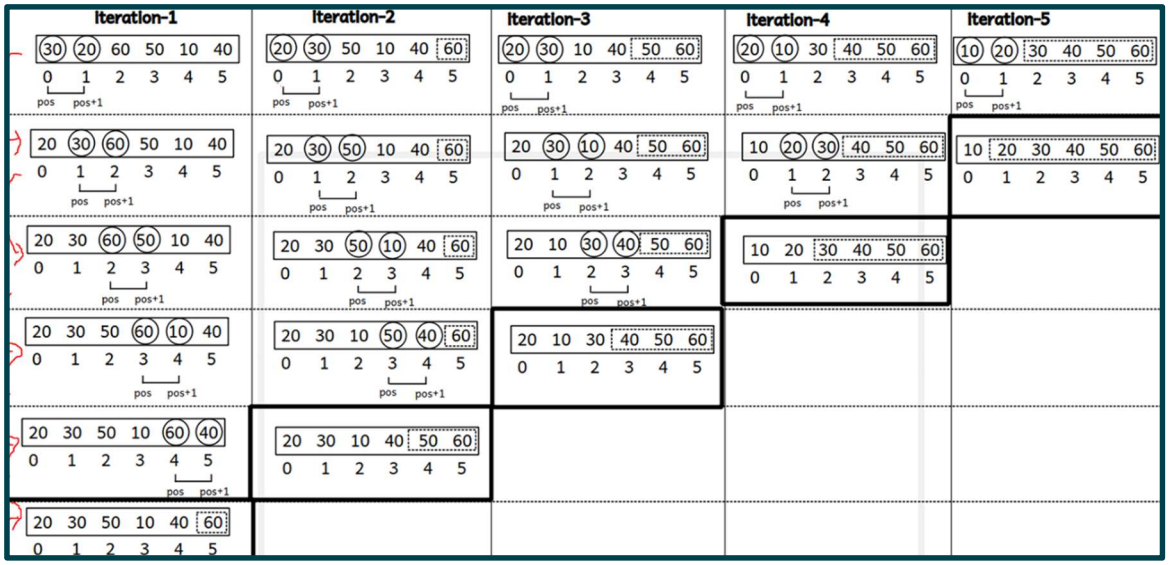
2. Insertion Sort

Insertion sort is an in-place comparison-based sorting algorithm that builds the final sorted array one element at a time by inserting each element into its correct position in the sorted subarray.



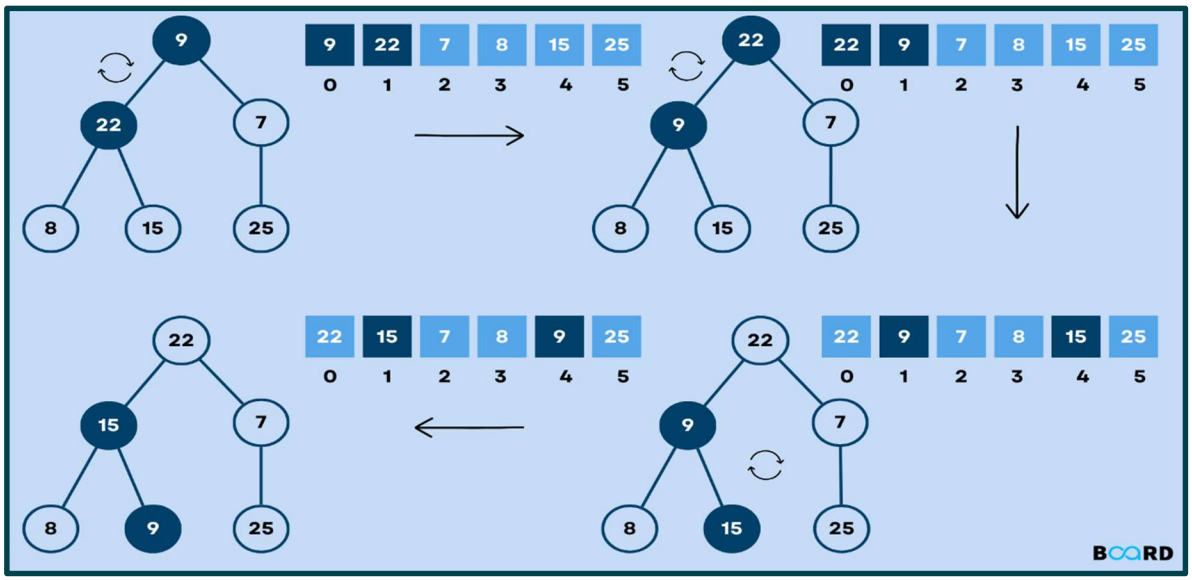
3. Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order, "bubbling" larger elements toward the end of the array.



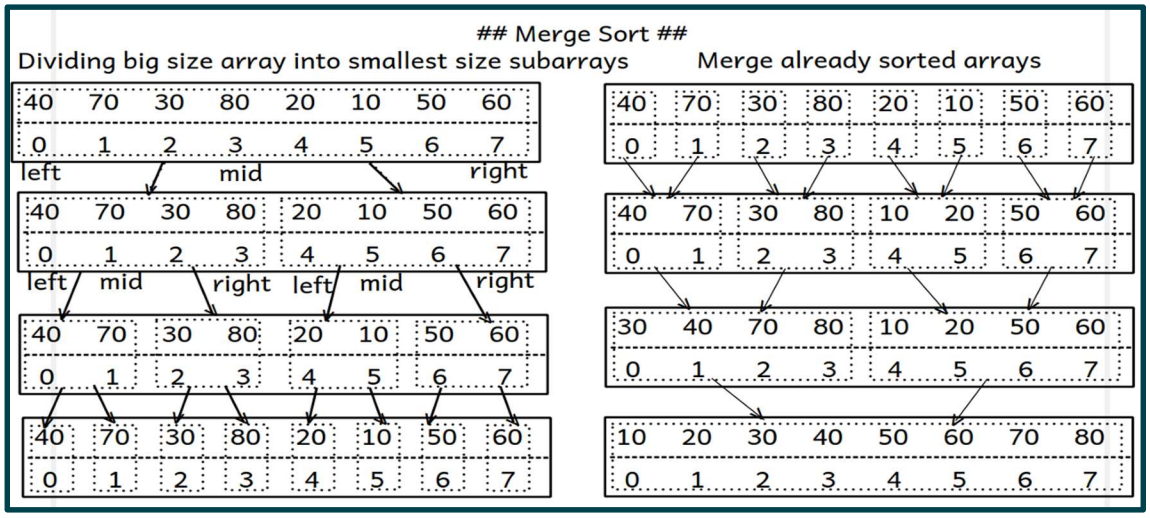
4. Heap sort

Heapsort is a comparison-based sorting algorithm that works by first building a binary heap data structure from the input array, and then repeatedly extracting the maximum element from the heap and placing it at the end of the sorted portion of the array.



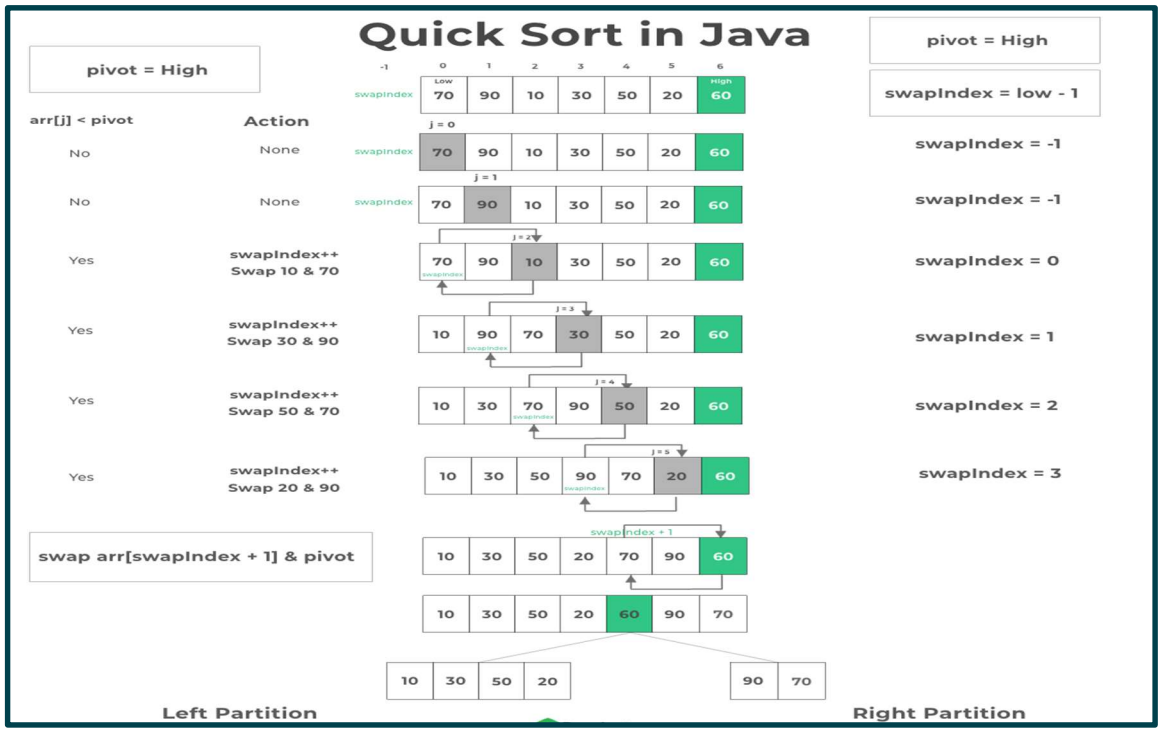
5. Merge sort

Merge sort is a divide-and-conquer sorting algorithm that recursively divides the unsorted array into two halves, sorts each half, and then merges the two sorted halves.



6. Quick sort

Quicksort is a divide-and-conquer sorting algorithm that partitions the array around a pivot element and then recursively sorts the sub-arrays on either side of the pivot.



Analysis of Sorting Algorithms

The time complexity of sorting algorithms is a crucial factor in determining their efficiency. Common time complexities include O(n^2) for simple sorts like selection, insertion, and bubble sort, O(n log n) for more efficient sorts like merge sort, heapsort, and quicksort (average case), and O(n) for non-comparison sorts like counting sort and radix sort (for specific input types).

Hashing & Introduction to Hash Tables

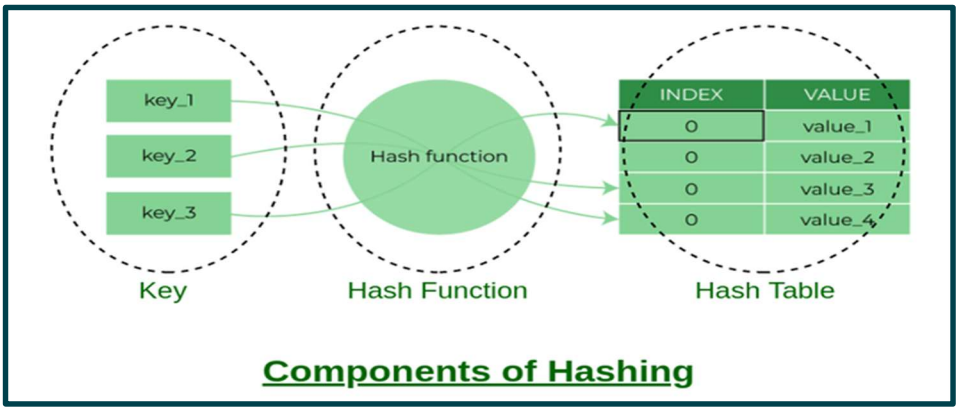
Hashing is a technique used to map data of an arbitrary size to a fixed-size value, called a hash value or hash code. Hash tables are data structures that use hashing to store and retrieve key-value pairs efficiently.

Need Of Hashing:

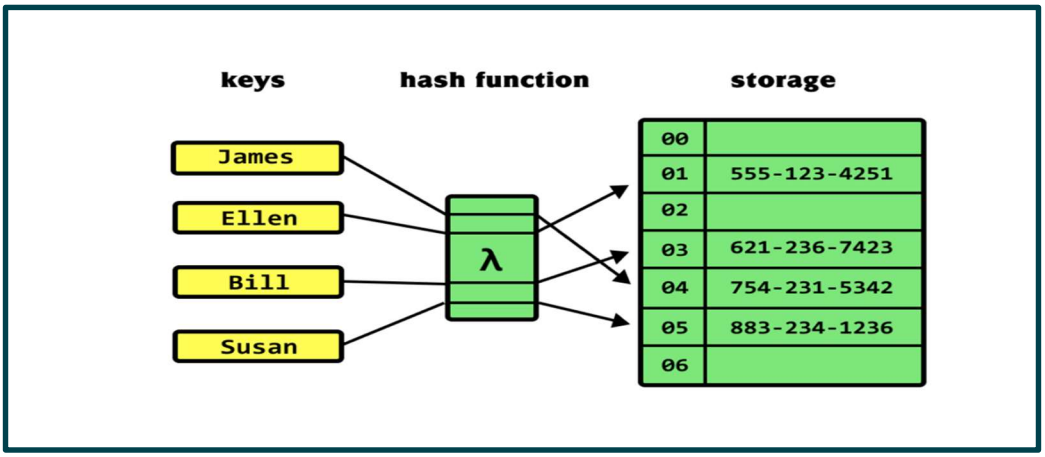
1. **Fast Data Retrieval**:
   Hashing provides a way to quickly locate and retrieve data from a large dataset. It allows constant-time average-case access to data, which is much faster than linear or logarithmic search algorithms.
2. **Data Organization**:
   Hashing is used to organize and store data in data structures like hash tables, which provide efficient insertion, deletion, and lookup operations.
3. **Caching**:
   Hashing is extensively used in caching mechanisms, such as CPU caches, web browser caches, and database caches. It allows for quick access to frequently used data, improving overall system performance.
4. **Cryptography**:
   Hashing is essential in cryptography for tasks like data encryption, message authentication, and digital signatures. Cryptographic hash functions are designed to be one-way and collision-resistant, making them suitable for secure data transmission and storage.
5. **Data Deduplication**:
   Hashing is used in data deduplication techniques, where it helps identify and eliminate redundant data by computing and comparing hash values of data blocks.
6. **Load Balancing**:
   In distributed systems and computer networks, hashing is used for load balancing, where it helps distribute workloads evenly across multiple servers or resources based on the hash values of incoming requests or data.
7. **Randomization**:
   Hashing can be used to introduce randomization in various algorithms and data structures, such as hash tables, bloom filters, and randomized algorithms, which can improve their performance and security.
8. **Fingerprinting**:
   Hashing is used for fingerprinting large data sets, where a small hash value can represent a much larger amount of data. This is useful for data integrity checking and duplicate detection.
9. **Indexing and Sorting**:
   Hashing is sometimes used in indexing and sorting algorithms, such as bucket sorting and radix sort, where it can provide efficient ways to group and organize data based on hash values

Components of Hashing:

1. **Key :**
   The key is the data item or value that needs to be stored or retrieved from the hash table. It can be a string, integer, or any other data type depending on the application.
2. **Hash Functions:**
   A hash function is a mathematical function that takes an input (key) and produces a hash value or hash code, which is a fixed-size integer or bit string.
3. **Hash Table:**
   The hash table is the data structure used to store and retrieve key-value pairs. It is an array-like structure that uses the hash values to determine the position (index) where the key-value pairs are stored.



Example Of Hashing:



Different Types of Hash Functions

Common hash functions include:
1. Division method
2. Multiplication method
3. Universal hashing
4. Cryptographic hash functions (e.g., MD5, SHA-256)

Collision Resolution Technique

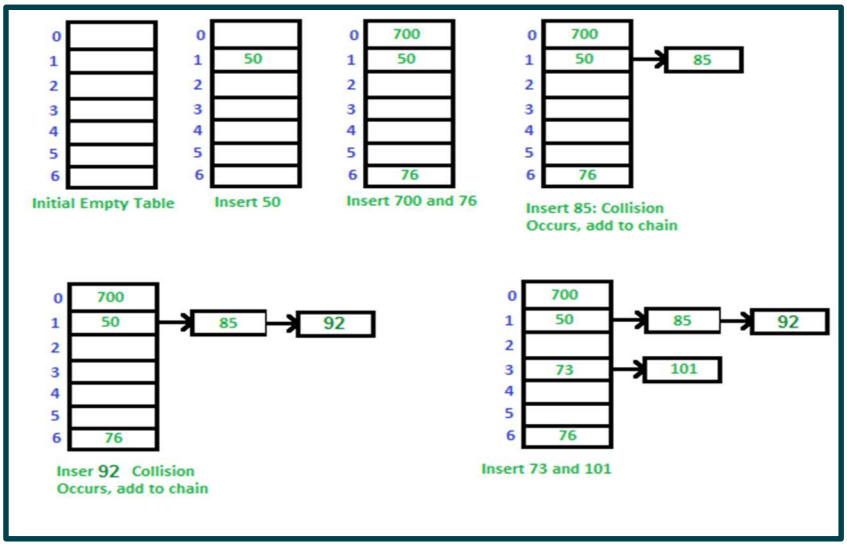
Collisions occur when two or more keys are mapped to the same hash value by the hash function. Collision resolution techniques are used to handle such situations.
There are two types of Collision Resolution Techniques:

A. Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called chain. Separate Chaining is one of the most popular and commonly used techniques in order to handle collisions

Ex:

Let us consider a simple hash function as "key mod 7" and a sequence of keys are 50, 700, 76, 85, 92, 73,101.



B. Open Addressing.

1. Linear Probing

Linear probing is a collision resolution technique in which the hash table is treated as a circular array, and when a collision occurs, the next available slot is probed linearly until an empty slot is found

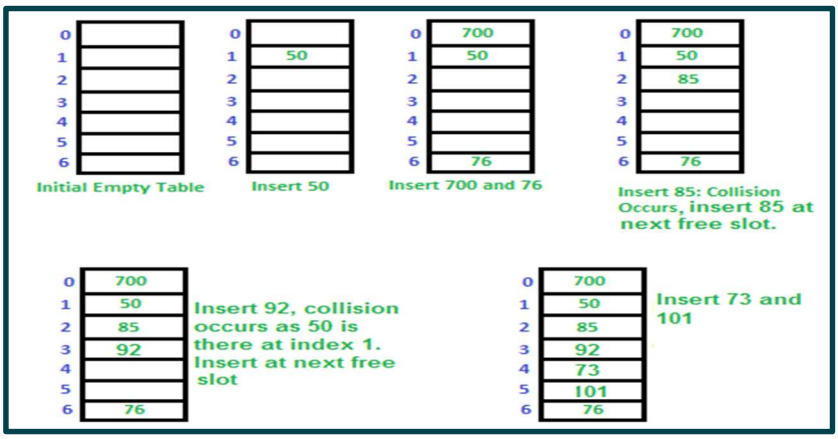
function used for rehashing is as follows:

"rehash(key) = (n+1) % table-size".

Ex: The typical gap between two probes is 1 as seen in example below:

Let **hash(x)** be the slot index computed using a hash function and **S** be the table size
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101, which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table, although it is fixed internally with a pre-defined formula.



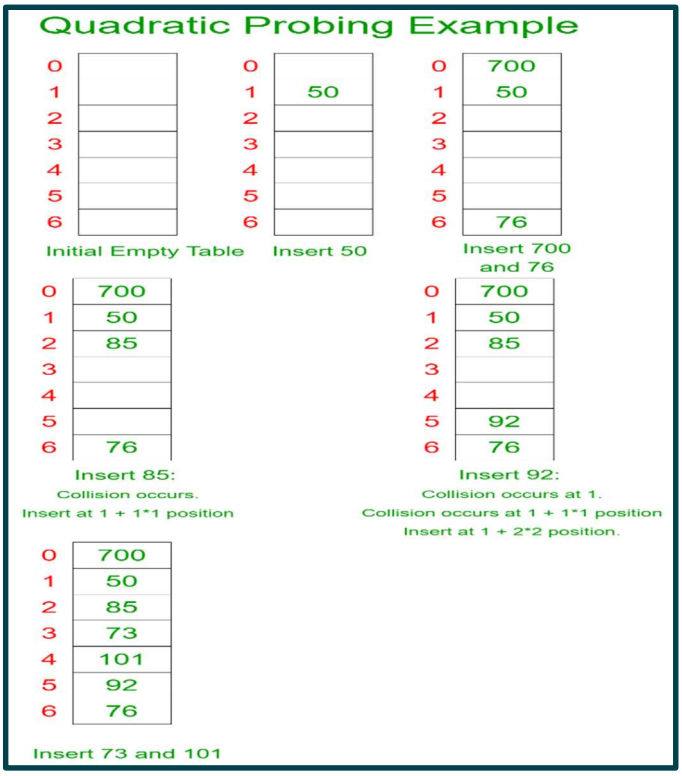
2. Quadratic Probing

Quadratic probing is a collision resolution technique that uses a quadratic function to determine the next slot to probe when a collision occurs.

Let hash(x) be the slot index computed using the hash function.

- If the slot hash(x) % S is full, then we try (hash(x) + 1*1) % S.
- If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S.
- If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S.
- This process is repeated for all the values of i until an empty slot is found.

For example: Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101



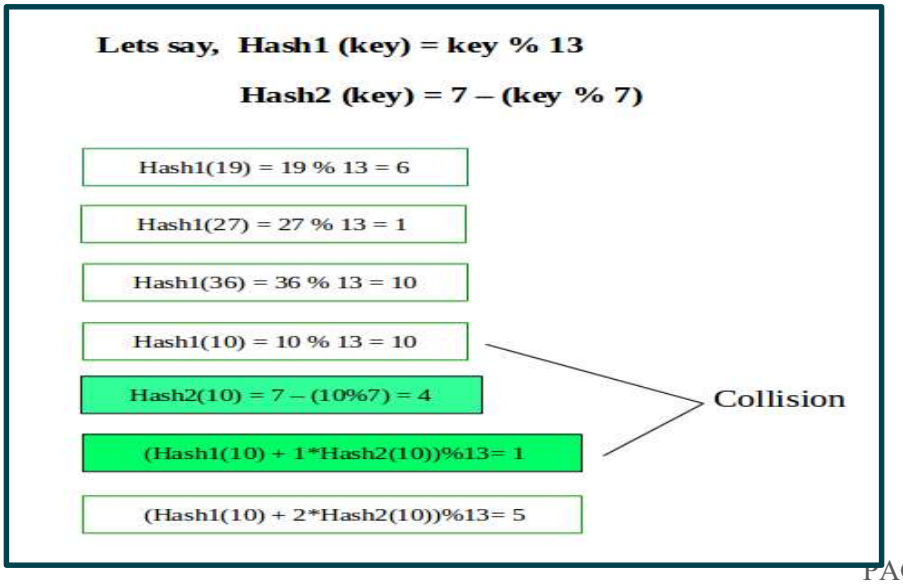
3. Double Hashing

Double hashing is a collision resolution technique that uses a second hash function to determine the step size for probing when a collision occurs.

Double hashing can be done using:
First hash function is typically hash1(key) = key % TABLE_SIZE
A popular second hash function is hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

A good second Hash function is:

1. It must never evaluate to zero
2. Just make sure that all cells can be probed



Inserting and Deleting an Element from a Hash Table

Inserting an element into a hash table involves computing the hash value of the key, probing for an empty slot using the chosen collision resolution technique, and storing the key-value pair in the empty slot. Deleting an element involves finding the slot containing the key-value pair and marking it as deleted or removing it from the table.

Insert Operation:

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
import java.util.Arrays;
public class Main {
  static final int SIZE = 4; // Define the size of the hash table
  static class DataItem {
    int key;
  }
  static DataItem[] hashArray = new DataItem[SIZE]; // Define the hash table as an array of DataItem pointers
  static int hashCode(int key) {
    // Return a hash value based on the key
    return key % SIZE;
  }
  static void insert(int key) {
    // Create a new DataItem
    DataItem newItem = new DataItem();
    newItem.key = key;
    // Initialize other data members if needed

    // Calculate the hash index for the key
    int hashIndex = hashCode(key);

    // Handle collisions (linear probing)
    while (hashArray[hashIndex] != null) {
      // Move to the next cell
      hashIndex++;
      // Wrap around the table if needed
      hashIndex %= SIZE;
    }
    // Insert the new DataItem at the calculated index
    hashArray[hashIndex] = newItem;
  }
public static void main(String[] args) {
  // Call the insert function with different keys to populate the hash table
  insert(42); // Insert an item with key 42
  insert(25); // Insert an item with key 25
  insert(64); // Insert an item with key 64
  insert(22); // Insert an item with key 22
  // Output the populated hash table
```

```
// Output the populated hash table
  for (int i = 0; i < SIZE; i++) {
    if (hashArray[i] != null) {
      System.out.println("Index " + i + ": Key " + hashArray[i].key);
    } else {
      System.out.println("Index " + i + ": Empty");
    }
  }
  }
}
```

Output:

```
Index 0: Key 64
Index 1: Key 25
Index 2: Key 42

Index 3: Key 22
```

Deletion Operation: Add the Following code in the Class Main() of Insertion operation.

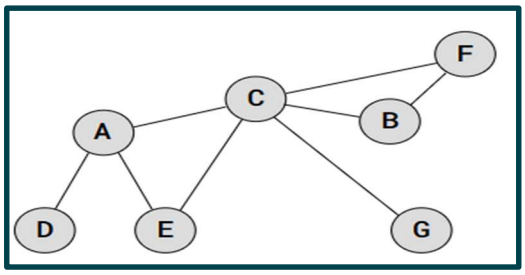
```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {
      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];
         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL; }
```

And add following to main() function

```
delete(1); // Delete an item with key 2

delete(2); // Delete an item with key 4
```

Introduction to Graph Theory

Graph theory is a branch of mathematics that studies the properties and behavior of graphs, which are mathematical structures used to model pairwise relations between objects.



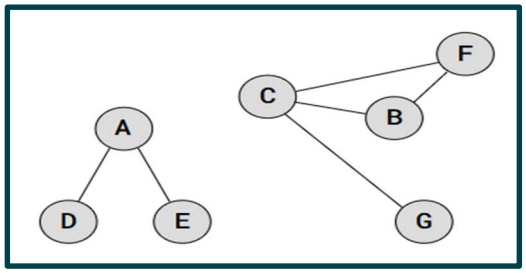
Graph Terminology

- **Vertex (Node):** An object or entity in a graph.
- **Edge:** A connection or link between two vertices.
- **Degree:** The number of edges connected to a vertex.
- **Path:** A sequence of vertices connected by edges.
- **Cycle:** A path that starts and ends at the same vertex.

Different Types of Graphs

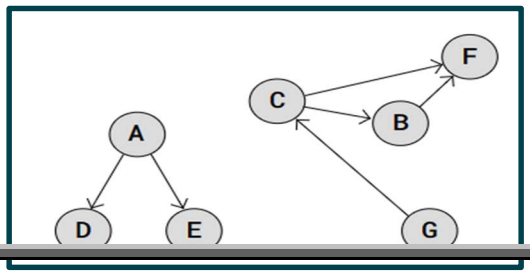
1. **Undirected graphs:**

   A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional.



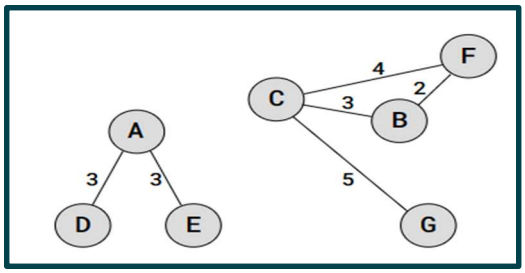
2. **Directed graphs:**

   A **directed** Graph, also known as a digraph, is when the edges between the vertex pairs have a direction. The direction of an edge can represents things like hierarchy or flow.



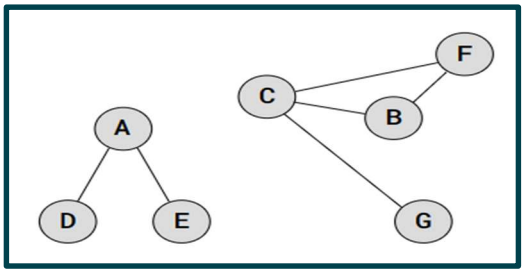
3. **Weighted graphs:**

A **weighted** Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability

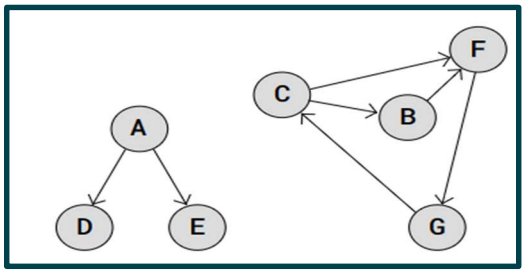


4. **Cyclic graphs:**

A Graph that contains at least one cycle which is a path that begins and ends at the same node, without passing through any other node twice.



5. **Acyclic graphs**



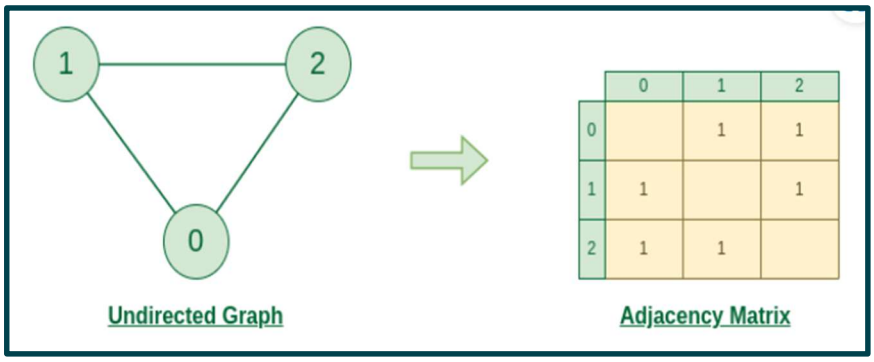
Representation of Graphs

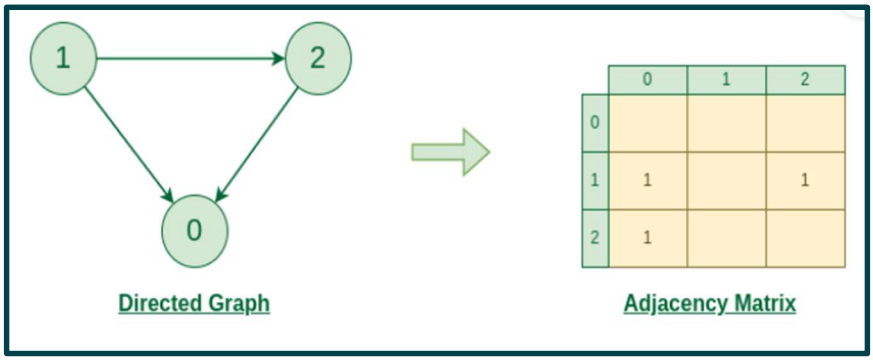
Graphs can be represented in various ways, including:

1. Adjacency Matrix

An adjacency matrix is a two-dimensional array representation of a graph, where each element represents the presence or absence of an edge between two vertices.

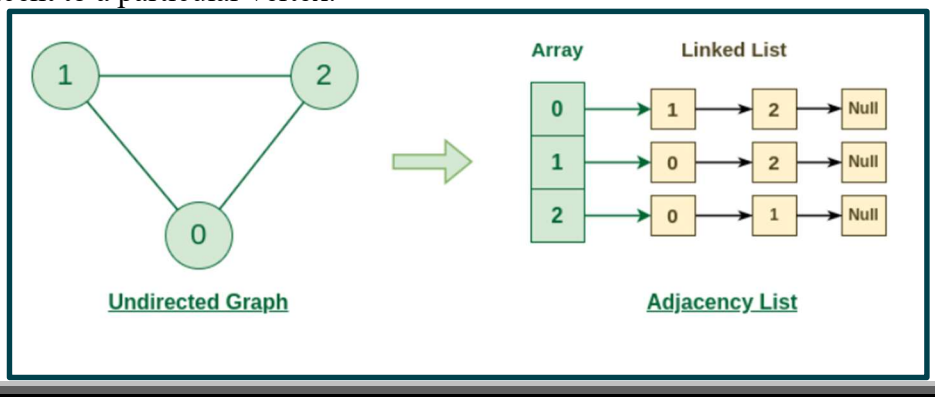
Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

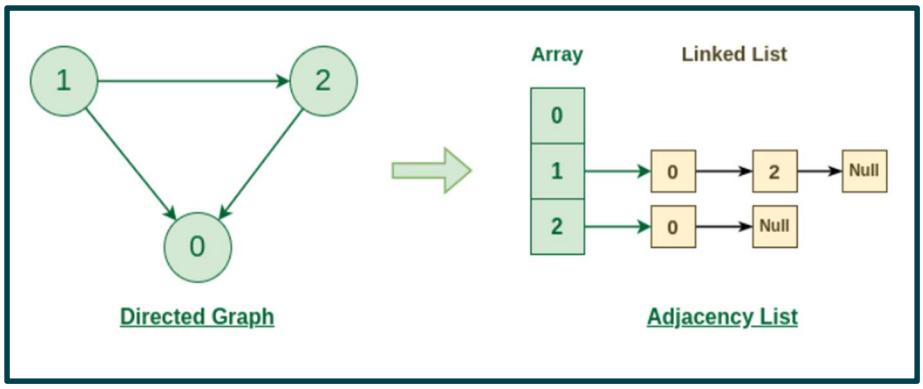




2. Adjacency List

An adjacency list is a collection of lists, where each list represents the set of vertices adjacent to a particular vertex.





Real-Time Applications of Graph:

1) **Social media analysis**:
   Social media platforms generate vast amounts of data in real-time, which can be analyzed using graphs to identify trends, sentiment, and key influencers. This can be useful for marketing, customer service, and reputation management.

2) **Network monitoring:**
   Graphs can be used to monitor network traffic in real-time, allowing network administrators to identify potential bottlenecks, security threats, and other issues. This is critical for ensuring the smooth operation of complex networks.

3) **Financial trading:**
   Graphs can be used to analyze real-time financial data, such as stock prices and market trends, to identify patterns and make trading decisions. This is particularly important for high-frequency trading, where even small delays can have a significant impact on profits.

4) **Internet of Things (IoT) management:**
   IoT devices generate vast amounts of data in real-time, which can be analyzed using graphs to identify patterns, optimize performance, and detect anomalies. This is important for managing large-scale IoT deployments.

5) **Autonomous vehicles:**
   Graphs can be used to model the real-time environment around autonomous vehicles, allowing them to navigate safely and efficiently. This requires real-time data from sensors and other sources, which can be processed using graph algorithms.

6) **Disease surveillance**:
   Graphs can be used to model the spread of infectious diseases in real-time, allowing health officials to identify outbreaks and implement effective containment

strategies. This is particularly important during pandemics or other public health emergencies.

7) The best example of graphs in the real world is Facebook. Each person on Facebook is a node and is connected through edges. Thus, A is a friend of B. B is a friend of C, and so on.

Advantages:

1. **Representation of Relationships**: Graphs are an excellent way to represent and model relationships between objects or entities. They can represent complex networks, such as social networks, transportation networks, and computer networks, in an intuitive and efficient manner.

2. **Path Finding**: Graphs are widely used in pathfinding algorithms, such as finding the shortest path between two points in a network. Algorithms like Dijkstra's algorithm and the A* algorithm is used for path finding in graphs

3. **Network Analysis**: Graphs are useful for analyzing various properties of networks, such as connectivity, centrality measures, and community detection. These analyses are important in fields like social network analysis, computer network analysis, and bioinformatics.

4. **Efficient Representation**: Depending on the graph representation (adjacency list or adjacency matrix), graphs can provide efficient storage and retrieval of data, especially for sparse graphs (graphs with few edges compared to the maximum possible number of edges).

5. **Generalization**: Graphs can be generalized to represent different types of data structures, such as trees, which can be considered as a special case of graphs.

Disadvantages:

1. **Memory Overhead**:
   Graphs can consume a significant amount of memory, especially for dense graphs (graphs with many edges) or large-scale graphs. The memory requirement grows with the number of vertices and edges in the graph.

2. **Traversal Complexity**:
   Graph traversal algorithms, such as breadth-first search (BFS) and depth-first search (DFS), can have higher time complexities compared to simpler data structures like arrays or linked lists, especially for dense graphs.

3. **Algorithm Complexity:**
   Many graph algorithms, such as finding the minimum spanning tree or solving the traveling salesman problem, have high time complexities and can become computationally expensive for large graphs.

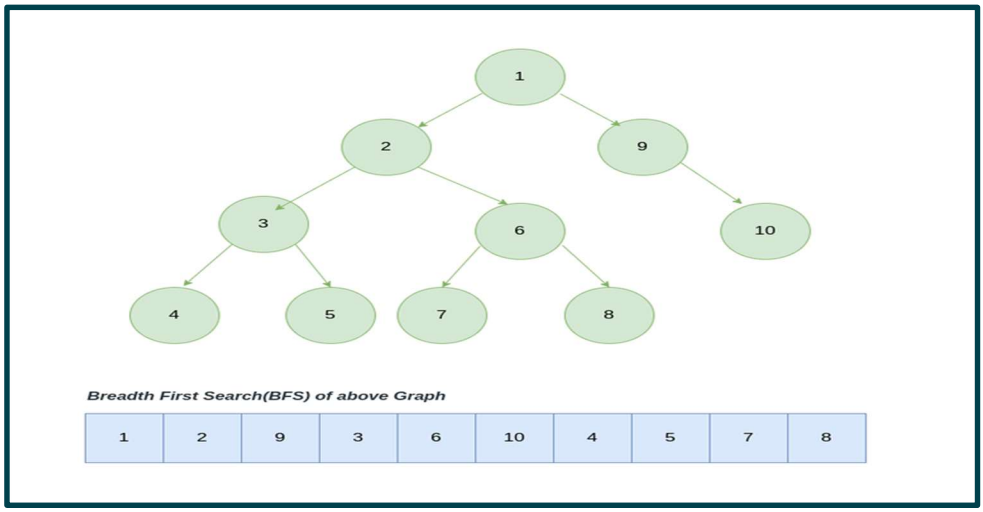
4. **Cyclic Nature**:
   Graphs can contain cycles, which can introduce additional complexity when designing and implementing algorithms to handle cycles appropriately.

5. **Serialization and Deserialization**:
   Serializing and deserializing graphs can be challenging, especially for large and complex graphs, as the relationships between vertices need to be preserved accurately.

Graph Traversal Algorithms

Graph traversal algorithms are used to visit or explore all vertices in a graph. The two main traversal algorithms are:

a) **Breadth-First Search (BFS):** Explores all vertices at the current level before moving to the next level.



b) **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking.

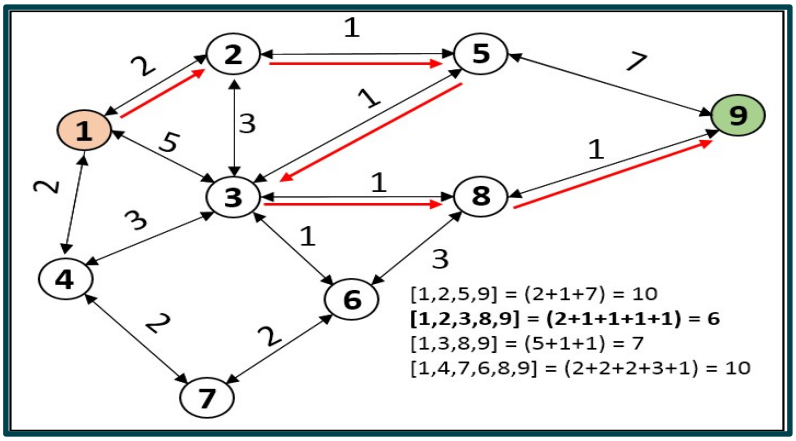
Shortest Path

Finding the shortest path between two vertices in a graph is a common problem in graph theory.

i) Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm used to find the shortest path between a single source vertex and all other vertices in a weighted graph with non-negative edge weights.

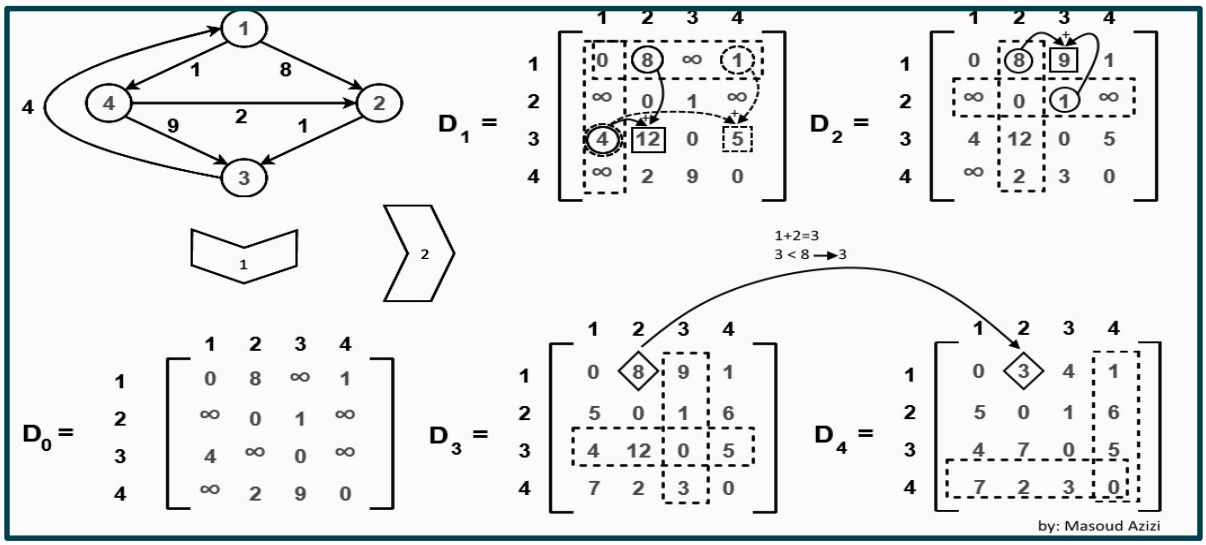
EX:



In the above Example, Route '1-2-3-8-9' is the Shortest Path for reaching From Source '1' to Destination '9'.

ii) Floyd-Warshall Algorithm

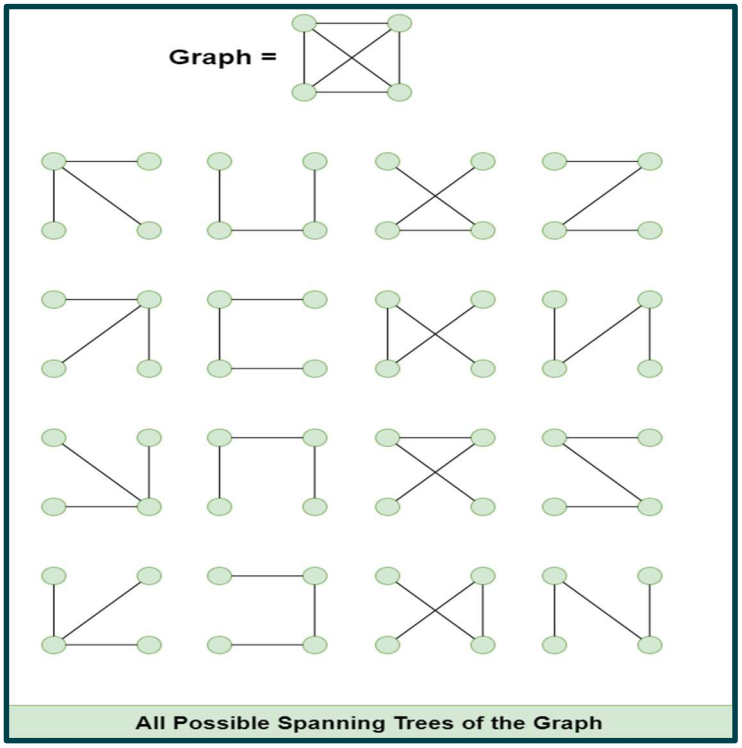
The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph.



In D4 we can see the calculated Shortest paths to reach the next point.

Spanning Trees

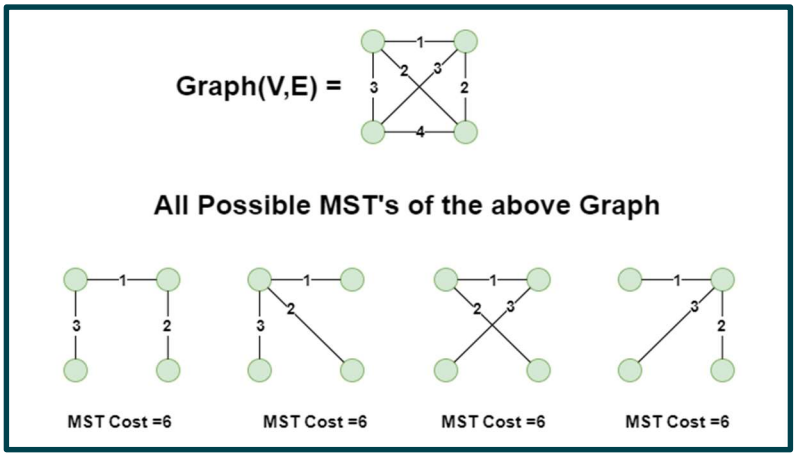
A Spanning tree is a subset of a graph that includes all vertices and forms a tree structure (i.e., it is acyclic and connected).



Types of Spanning Tree Algorithms:

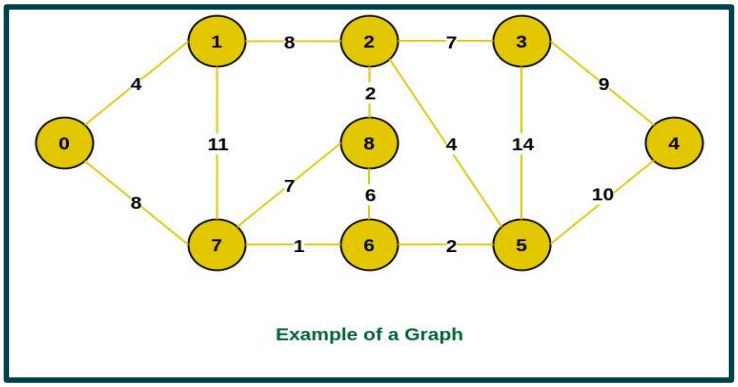
1. **Minimum Spanning Tree Algorithms**

   Minimum spanning tree algorithms are used to find the minimum-weight spanning tree of a weighted undirected graph.

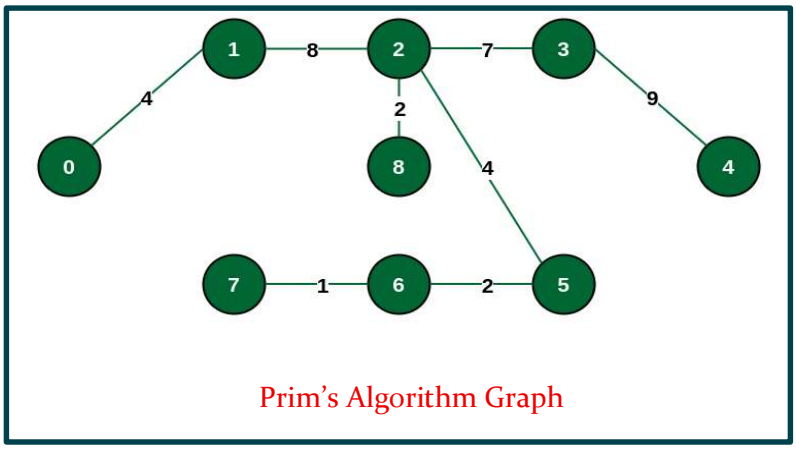


2. Prim's Algorithm

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree of a weighted undirected graph.



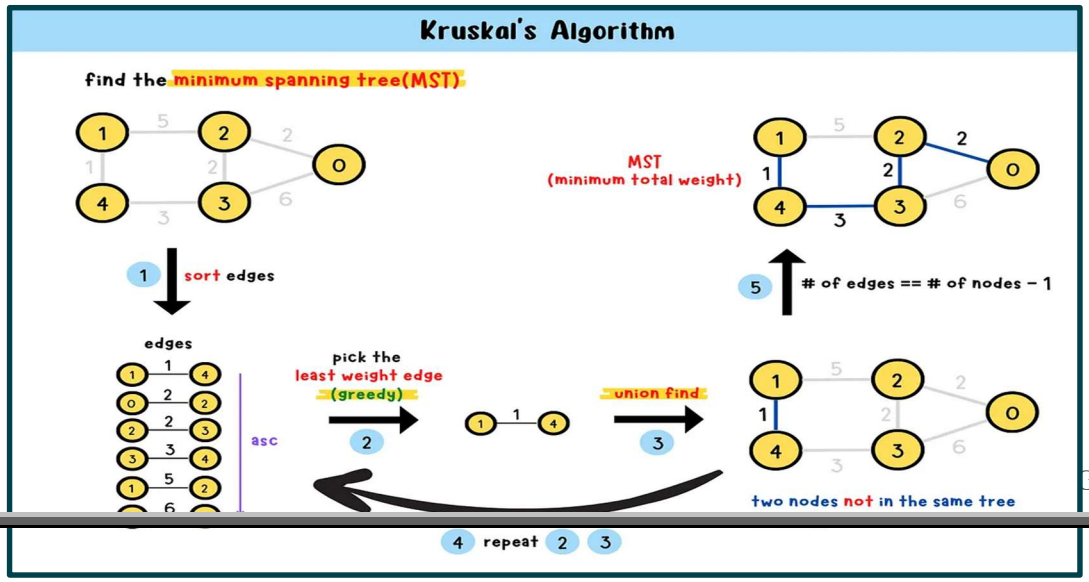
Example of a Graph



Prim's Algorithm Graph

3. Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that finds the minimum spanning tree of a weighted undirected graph by adding edges in increasing order of weight.



Classes of Algorithms

Algorithms can be classified based on their computational complexity, design techniques, or specific problem domains. Some common classes of algorithms include:

1. Brute force algorithms
2. Divide and conquer algorithms
3. Greedy algorithms
4. Dynamic programming algorithms
5. Backtracking algorithms
6. Branch and bound algorithms
7. Randomized algorithms

Writing Efficient Algorithms

To write efficient algorithms, one should consider factors such as:

1. Time and space complexity
2. Input size and potential growth
3. Choice of appropriate data structures
4. Use of efficient algorithm design techniques
5. Optimization techniques (e.g., memoization, pruning)
6. Trade-offs between time and space complexity

Algorithm Design Techniques

Algorithm design techniques are strategies and methodologies used to develop efficient algorithms for solving computational problems. Some common algorithm design techniques include:

1. **Divide and Conquer:** This technique involves breaking a problem into smaller sub-problems, solving each sub-problem independently, and then combining the solutions to solve the original problem.

2. **Greedy Approach:** A greedy algorithm makes locally optimal choices at each step with the hope of finding a global optimum. It builds a solution step by step, making the best choice at each step without consideration for future consequences.

3. **Dynamic Programming:** Dynamic programming is a technique for solving complex problems by breaking them down into simpler sub-problems, solving each sub-problem once, and storing the solutions for future use.

4. **Brute Force:** The brute force approach involves systematically enumerating all possible candidates for the solution and checking each candidate to see if it satisfies the problem's statement.

5. **Backtracking:** Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

6. **Branch and Bound:** Branch and bound is a technique used for solving optimization problems by exploring all possible solutions in a strategic way and pruning the search space based on specific criteria.

7. **Randomized Algorithms:** Randomized algorithms incorporate randomness as part of their logic, often to improve efficiency or simplify complex problems.

Analysis of an Algorithm

1. Asymptotic Analysis

Asymptotic analysis is a technique used to analyze the time and space complexity of an algorithm as the input size grows towards infinity. It focuses on the dominant term of the complexity function, ignoring constant factors and lower-order terms.

2. Algorithm Analysis

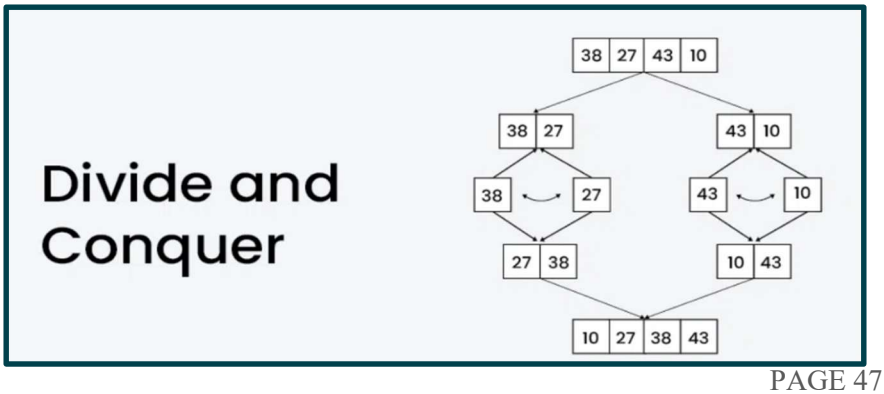
Algorithm analysis involves determining the time and space complexity of an algorithm using various techniques, such as:

- **Analytical Methods:** Mathematical analysis of the algorithm's logic and operations to derive the complexity function.
- **Experimental Methods:** Empirical evaluation of the algorithm's performance on different input sizes and test cases.

Analysis of Different Types of Algorithms

1. Divide and Conquer Algorithms

Divide and conquer algorithms have a time complexity that is typically expressed using recurrence relations, which can be solved using techniques like the master method or substitution method.



Examples

• Quick-Sort:
  • Partition the array into two parts, and quicksort each of the parts
  • No additional work is required to combine the two sorted parts

• Merge-Sort:
  • Cut the array in half, and merge-sort each half
  • Combine the two sorted arrays into a single sorted array by merging them

2. Greedy Algorithms

The time complexity of greedy algorithms is usually determined by analyzing the number of steps required to make locally optimal choices and construct the solution.

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be: At each step, take the largest possible bill or coin that does not overshoot
- Example:

  To make $6.39, you can choose:
  • a $5 bill
  • a $1 bill, to make $6
  • a 25¢ coin, to make $6.25
  • A 10¢ coin, to make $6.35
  • four 1¢ coins, to make $6.39
  • For US money, the greedy algorithm always gives the optimum solution

3. Dynamic Programming Algorithms

The time complexity of dynamic programming algorithms depends on the number of sub-problems and the time required to solve each sub-problem, while the space complexity depends on the size of the data structures used to store the solutions to sub-problems.

Example:

- Fibonacci Number Series,
- Factorial etc.

4. Brute Force Algorithms

Brute force algorithms often have exponential time complexity, as they typically involve enumerating all possible solutions.

Example: Finding the best path for a travelling salesman

• Satisficing: Stop as soon as a solution is found that is good enough

Example: Finding a travelling salesman path that is within 10% of optimal

5. Backtracking Algorithms

The time complexity of backtracking algorithms is often exponential in the worst case, as they may need to explore all possible solutions in the search space.

Examples:

a) **N-Queens Problem**:
Place N queens on an N×N chessboard such that no two queens attack each other (i.e., they are not in the same row, column, or diagonal). The backtracking algorithm builds candidates for the solutions and abandons a candidate ("backtracks") as soon as it determines that this candidate cannot possibly be completed to a valid solution.

b) **Sudoku Solver**:
Solve a partially filled Sudoku puzzle by backtracking and trying different values in the empty cells until a valid solution is found.

c) **Graph Coloring**:
Assign colors to the vertices of a graph such that no two adjacent vertices have the same color, using the minimum number of colors. Backtracking is used to explore different color assignments and backtrack when a constraint is violated.

6. Branch and Bound Algorithms

The time complexity of branch and bound algorithms depends on the effectiveness of the pruning strategy and the size of the search space that needs to be explored.

Example:

- **Traveling Salesman Problem (TSP)**:
Find the shortest possible tour that visits all cities exactly once and returns to the starting city. The branch and bound algorithm explores the solution space by branching (generating candidate solutions) and bounding (calculating lower bounds on the cost of a candidate solution) to prune the search tree and avoid exploring unnecessary branches.

- **Knapsack Problem**:
Given a set of items with weights and values, determine the optimal subset of items to include in a knapsack such that the total weight does not exceed the

knapsack's capacity, and the total value is maximized. Branch and bound is used to explore the solution space efficiently by pruning suboptimal branches.

7. Stochastic Algorithms

The time complexity of stochastic algorithms can be analyzed using probabilistic methods, considering the expected number of iterations or operations required to achieve a desired result.

Examples:

- **Simulated Annealing**:

  A probabilistic technique for approximating the global optimum of a given function. It is inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to minimize its energy state. In the algorithm, a random solution is generated, and then small perturbations are made to the solution. If the new solution is better, it is accepted; if it is worse, it may still be accepted with a certain probability that decreases over time, allowing the algorithm to escape local optima.

- **Genetic Algorithms**:

  A metaheuristic inspired by the process of natural selection. It operates on a population of candidate solutions, which are iteratively evolved through the application of genetic operators (e.g., mutation, crossover) to find better solutions. The fitness function evaluates the quality of each solution, and the fittest solutions are more likely to be selected for the next generation.

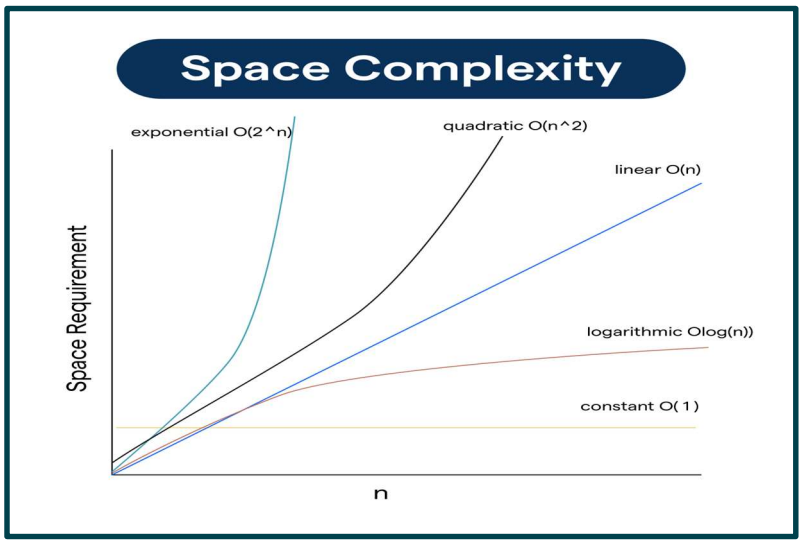
Complexity

Complexity Analysis

Complexity analysis involves determining the time and space complexity of an algorithm, which provides insights into its efficiency and scalability.

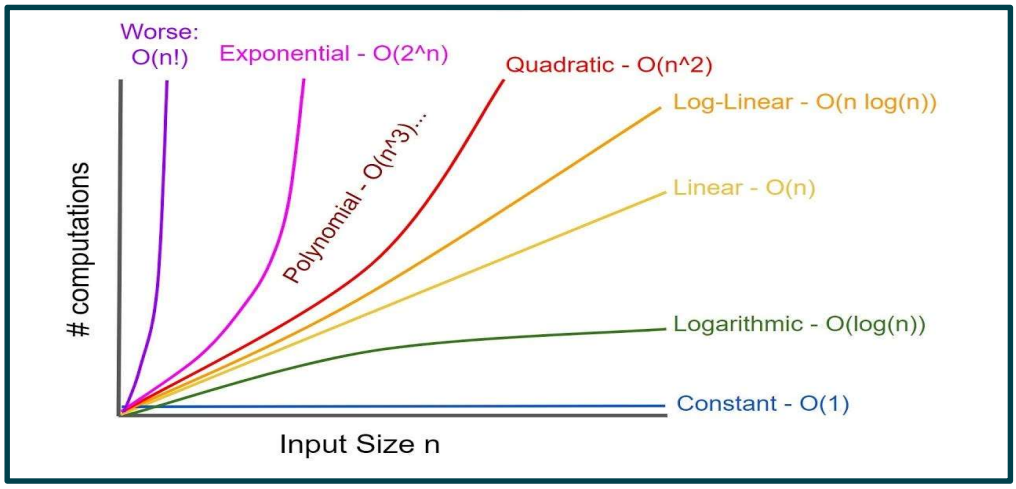
1. Space Complexity of an Algorithm

Space complexity refers to the amount of memory or storage space required by an algorithm as a function of the input size. It is typically expressed using Big O notation (e.g., O(n), O(log n), O(1)).



2. Time Complexity of an Algorithm

Time complexity refers to the amount of time required by an algorithm to run as a function of the input size. It is also typically expressed using Big O notation (e.g., O(n^2), O(n log n), O(n)).



Case Study on Algorithm Design Techniques

Case studies on algorithm design techniques can involve analyzing real-world problems or applications and applying various design techniques to develop efficient solutions. These case studies can cover topics such as optimization problems, graph algorithms, string algorithms, and computational geometry problems.

Application of Data Structures

Data structures are fundamental building blocks in the design and implementationss of efficient algorithms. Different data structures have their own strengths and weaknesses, and choosing the right data structure is crucial for achieving optimal performance. Some common applications of data structures include:

1. Arrays and Linked Lists for storing and manipulating sequential data.
2. Stacks and Queues for managing operations in a specific order.
3. Trees and Heaps for hierarchical data and priority queue operations.
4. Graphs for representing and analyzing interconnected data.
5. Hash Tables for efficient key-value storage and retrieval.
6. Tries for efficient string operations and information retrieval.

The selection and application of appropriate data structures can significantly impact the efficiency and performance of algorithms, making it an essential aspect of algorithm design and analysis.