IARE
INSTITUTE OF
AERONAUTICAL ENGINEERING

Introduction to Data Structures

- Classification of data structures
- Recursive algorithms
- Abstract data type (ADT), basic operations
- Time & space complexity analysis
- Asymptotic notations (Big-O, Big-Ω, Big-Θ)

Dr. B. Surekha Reddy
IARE10795

Introduction to Data Structures

- In modern computing, programs deal with a large amount of data.
  - To handle this data efficiently, we need a proper way of organizing, storing, and processing it.
  - This is where **Data Structures** play an important role.
- A data structure provides a systematic way of managing data so that operations can be performed efficiently in terms of time and memory.

Introduction to Data Structures

What is a Data Structure?

A **data structure** is a particular way of organizing and storing data in a computer so that it can be accessed and used efficiently.

Definition:

A data structure is a method of organizing data in memory to make it easier to perform operations such as insertion, deletion, searching, and sorting.

Key Objectives:

- Reduce time complexity
- Reduce space complexity
- Improve program performance
- Efficient data management

Introduction to Data Structures

Need Of Data Structure:

The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation. Data structures provide an easy way of organising, retrieving, managing, and storing data.

Here is a list of the needs for data.

- Data structure modification is easy.
- It requires less time.
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database

Introduction to Data Structures

Classification/Types of Data Structures:

1. Linear Data Structure
2. Non-Linear Data Structure.

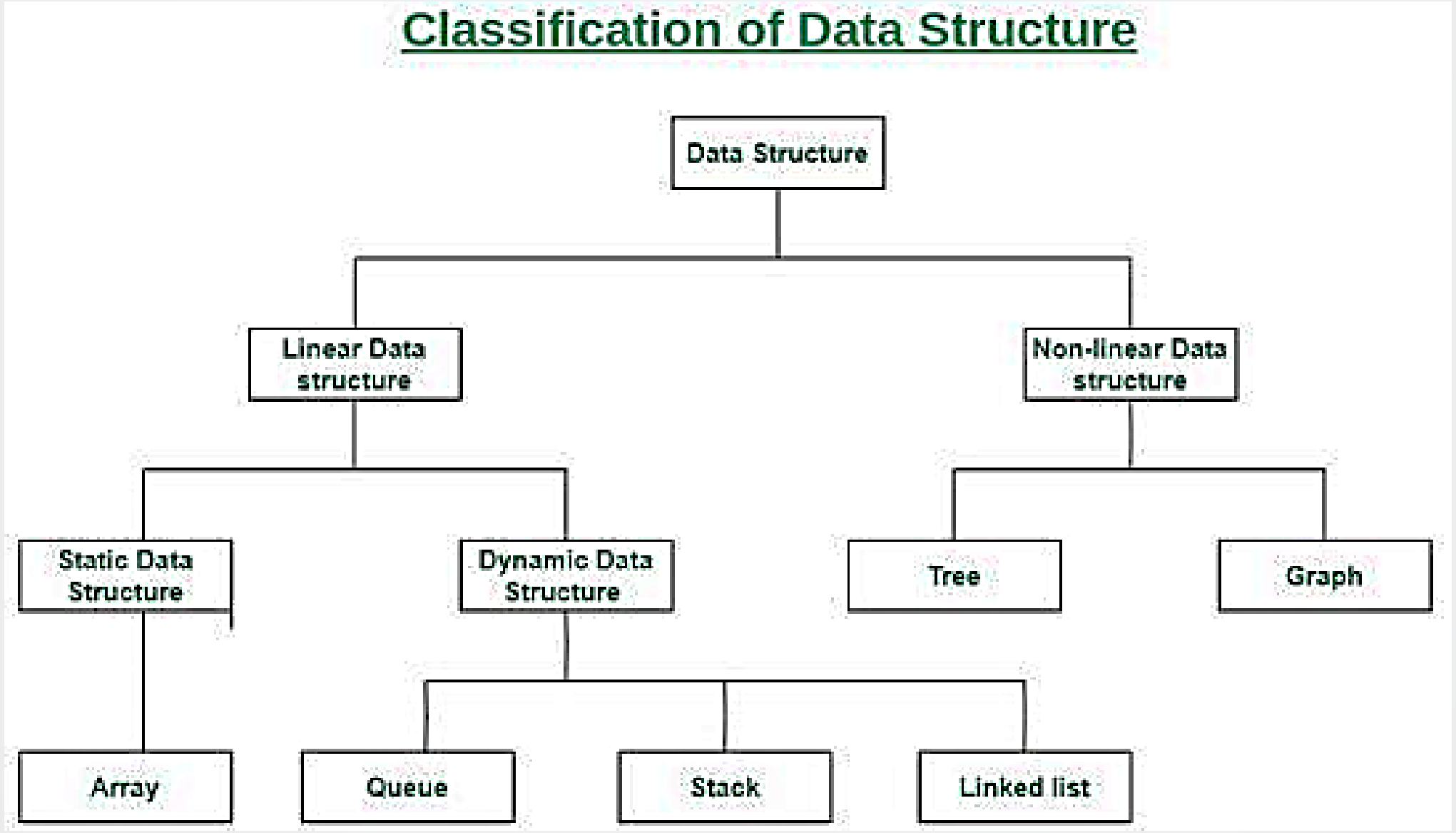
Linear Data Structure:

- Elements are arranged in one dimension ,also known as linear dimension.
- Example: lists, stack, queue, etc.

Non-Linear Data Structure

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.

Introduction to Data Structures



Introduction to Data Structures

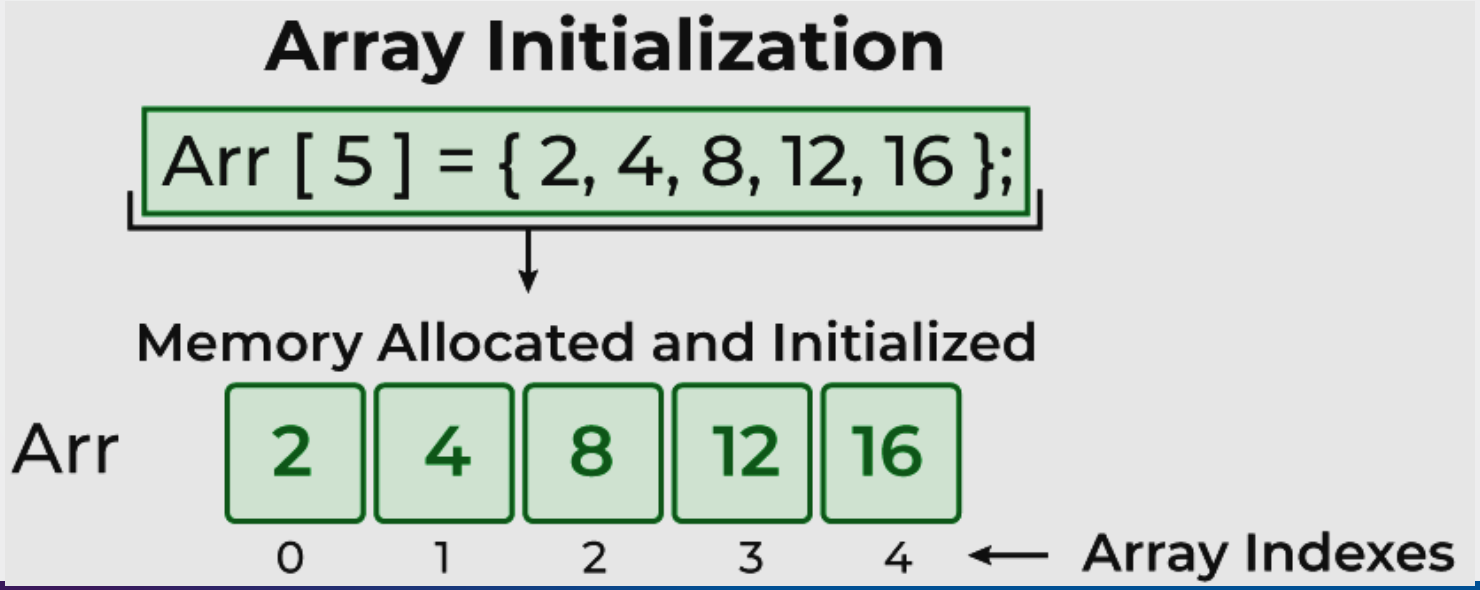
Most Popular Data Structures:

1. Array
2. Linked Lists
3. Stack
4. Queue
5. Binary Tree
6. Binary Search Tree
7. Heap
8. Hash Table Data Structure
9. Matrix
10. Trie
11. Graph

Introduction to Data Structures

Array

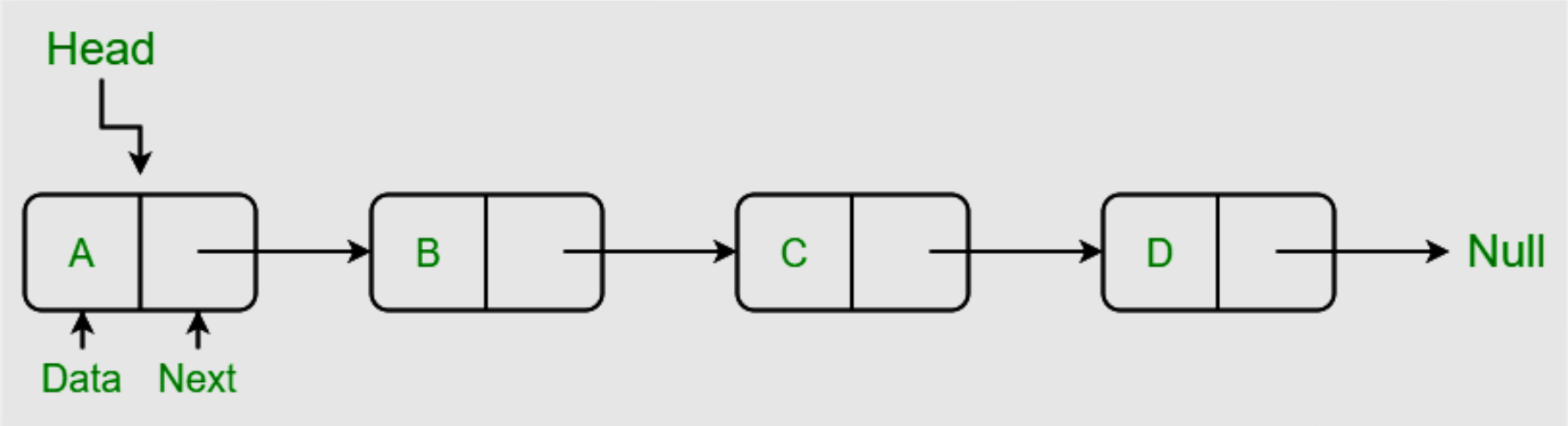
- An array is a collection of data items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Introduction to Data Structures

Linked Lists

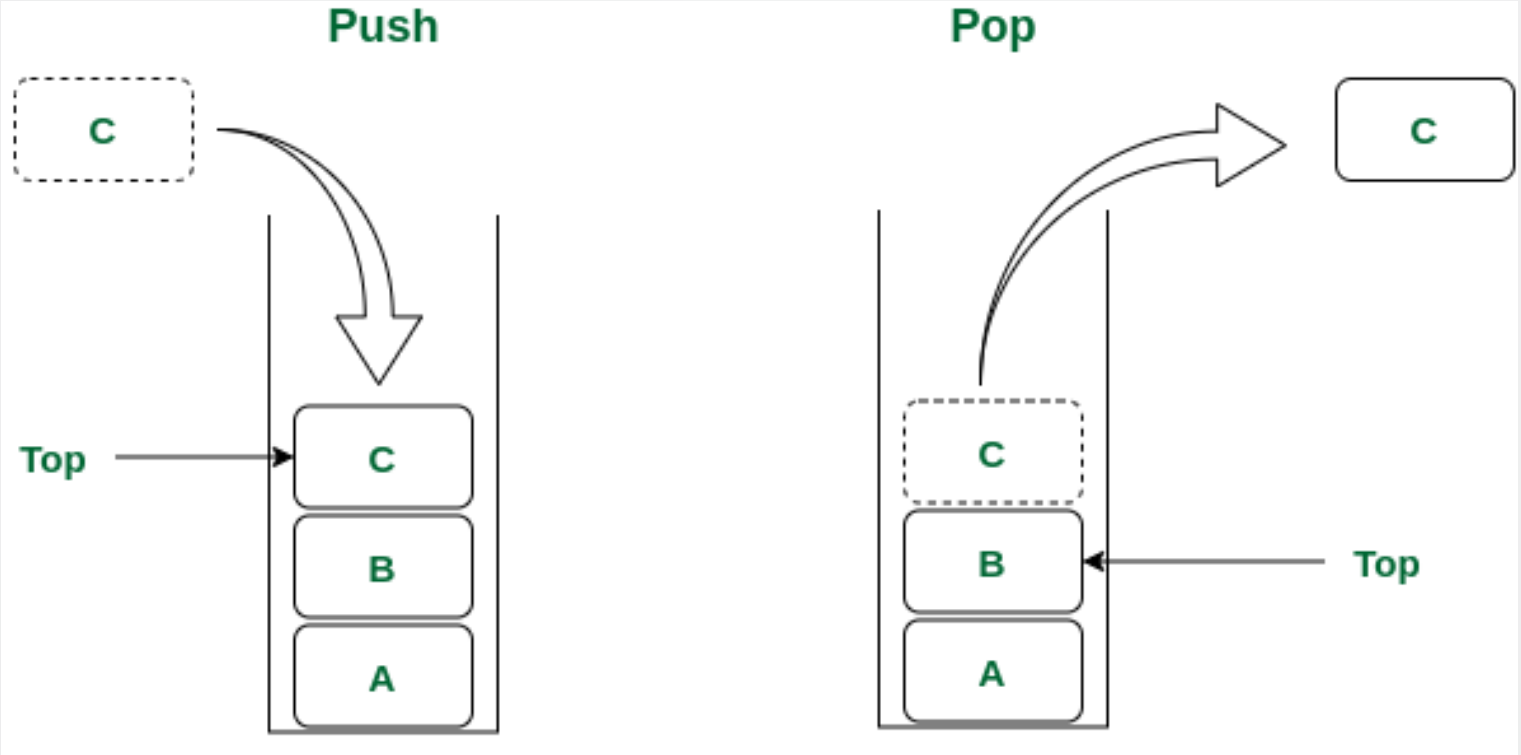
- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Introduction to Data Structures

Stack

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- In stack, all insertion and deletion are permitted at only one end of the list.



Introduction to Data Structures

Stack Operations:

- **push():** When this operation is performed, an element is inserted into the stack.
- **pop():** When this operation is performed, an element is removed from the top of the stack and is returned.
- **top():** This operation will return the last inserted element that is at the top without removing it.
- **size():** This operation will return the size of the stack i.e. the total number of elements present in the stack.
- **isEmpty():** This operation indicates whether the stack is empty or not.

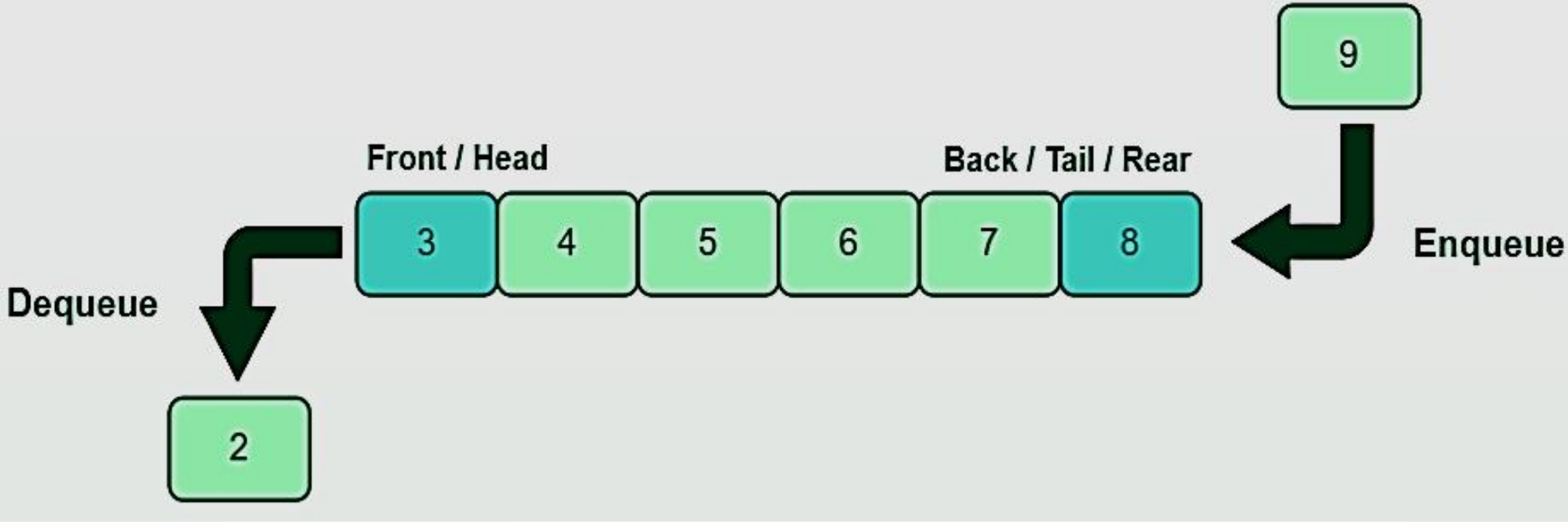
Introduction to Data Structures

Queue

- Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed.
- The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end.
- A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in removing.
- In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Introduction to Data Structures

Queue



Introduction to Data Structures

Queue

Queue Operations:

- **Enqueue():** Adds (or stores) an element to the end of the queue..
- **Dequeue():** Removal of elements from the queue.
- **Peek() or front():** Acquires the data element available at the front node of the queue without deleting it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isFull():** Validates if the queue is full.
- **isNull():** Checks if the queue is empty.

Introduction to Data Structures

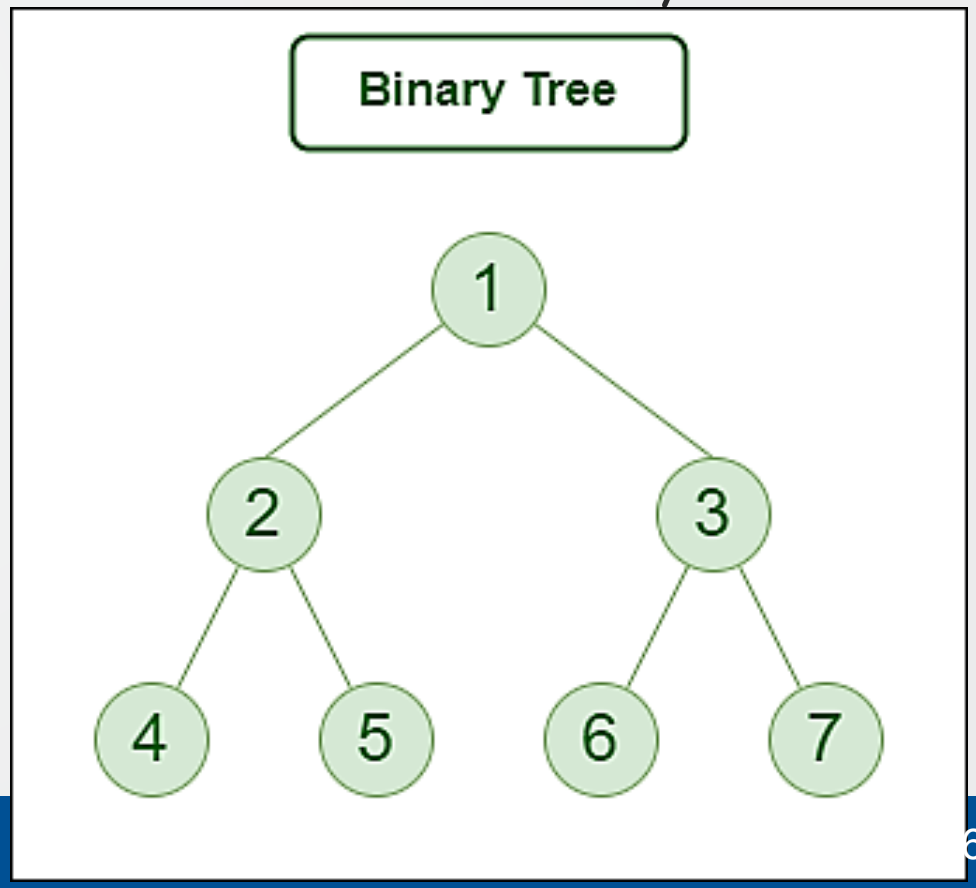
Binary Tree

- Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.
- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- It is implemented mainly using Links.

Introduction to Data Structures

Binary Tree

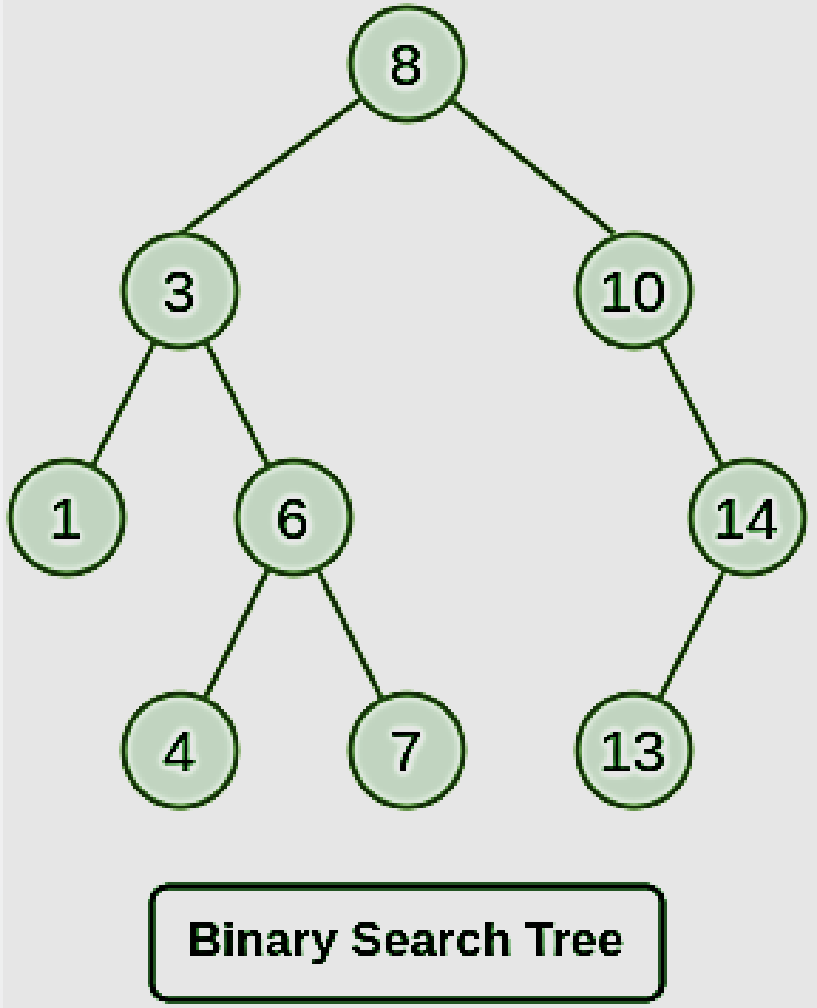
- A Binary Tree is represented by a pointer to the topmost node in the tree.
- If the tree is empty, then the value of root is NULL. A Binary Tree node contains the following parts.
    1. Data
    2. Pointer to left child
    3. Pointer to the right child



Introduction to Data Structures

Binary Search Tree

- A Binary Search Tree is a Binary Tree following the additional properties:
  - The left part of the root node contains keys less than the root node key.
  - The right part of the root node contains keys greater than the root node key.
  - There is no duplicate key present in the binary tree.
- A Binary tree having the following properties is known as Binary search tree (BST).



Introduction to Data Structures

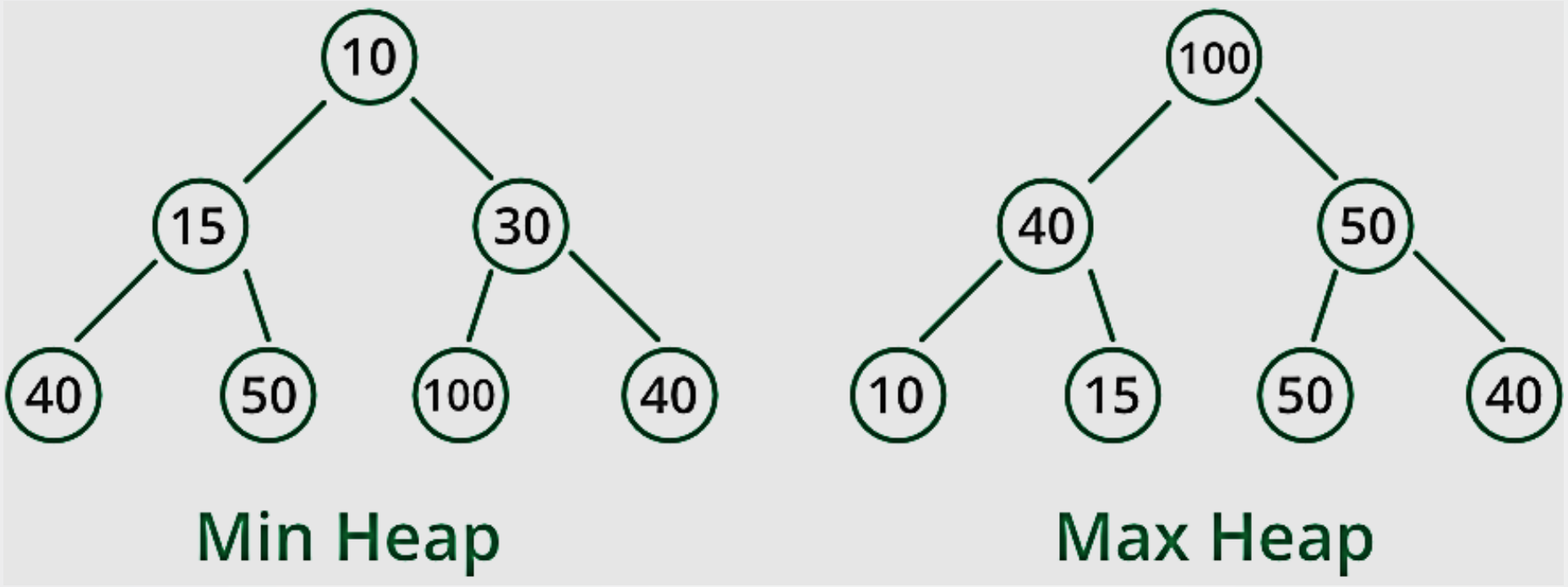
Heap

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

- **Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Introduction to Data Structures

Heap



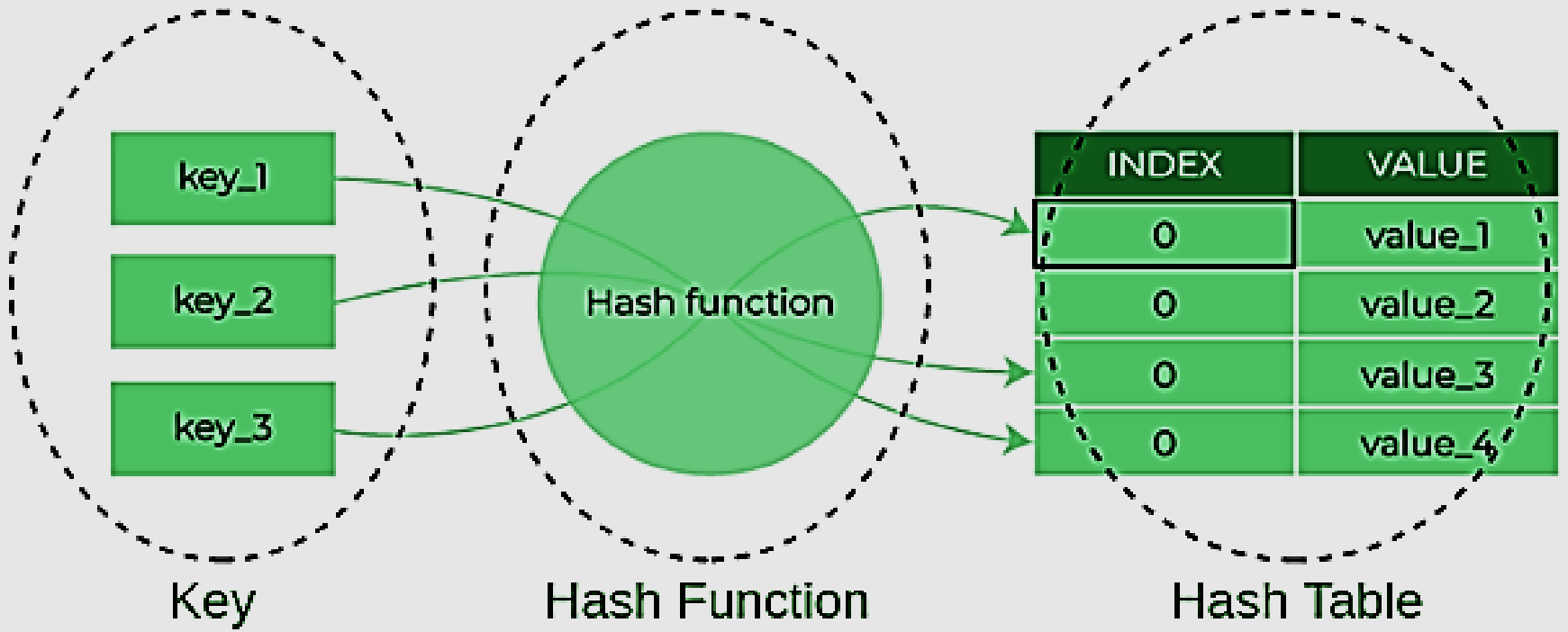
Introduction to Data Structures

Hash Table Data Structure

- Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.
- The efficiency of mapping depends on the efficiency of the hash function used.
- Let a hash function H(x) maps the value x at the index x%10 in an Array. For example, if the list of values is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively.

Introduction to Data Structures

Hash Table Data Structure

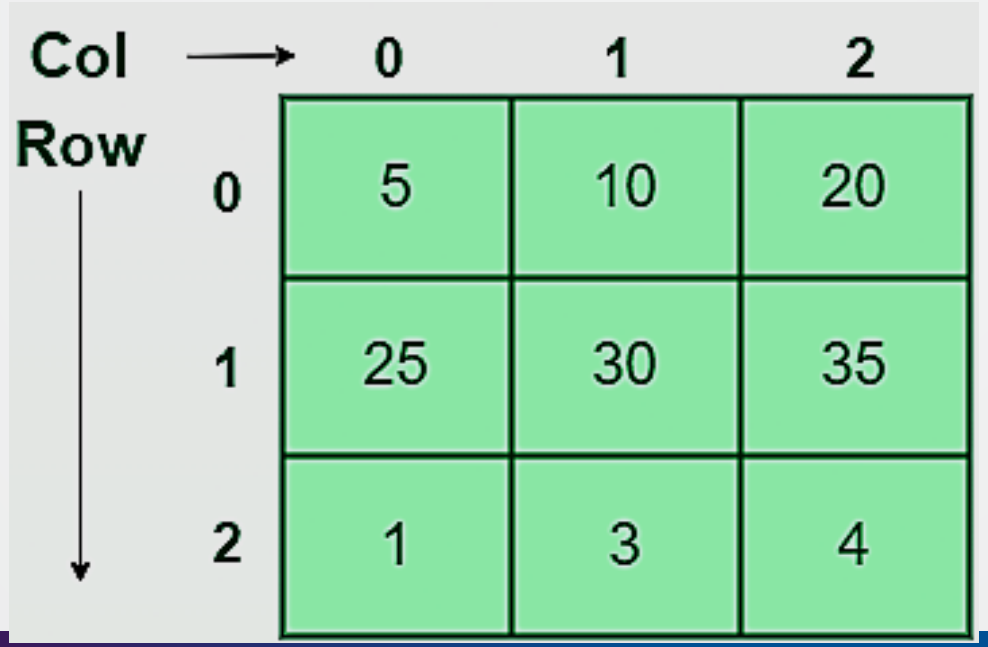


Components of Hashing

Introduction to Data Structures

Matrix

- A matrix represents a collection of numbers arranged in an order of rows and columns.
- It is necessary to enclose the elements of a matrix in parentheses or brackets.
- A matrix with 9 elements is shown below.



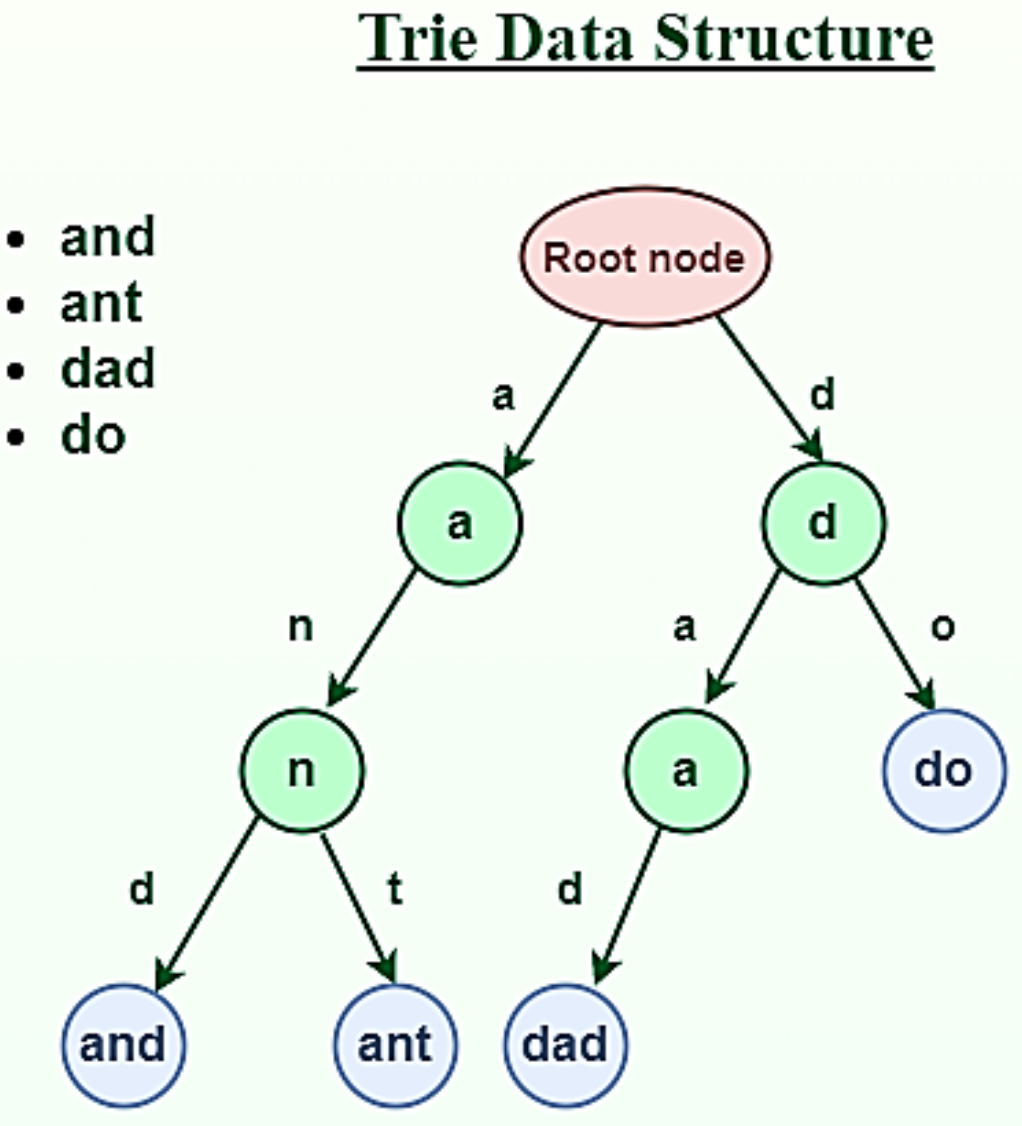
Introduction to Data Structures

Trie

- Trie is an efficient information retrieval data structure.
- Using Trie, search complexities can be brought to an optimal limit (key length).
- If we store keys in the binary search tree, a well-balanced BST will need time proportional to M * log N, where M is maximum string length and N is the number of keys in the tree.
- Using Trie, we can search the key in O(M) time.
- However, the penalty is on Trie storage requirements.

Introduction to Data Structures

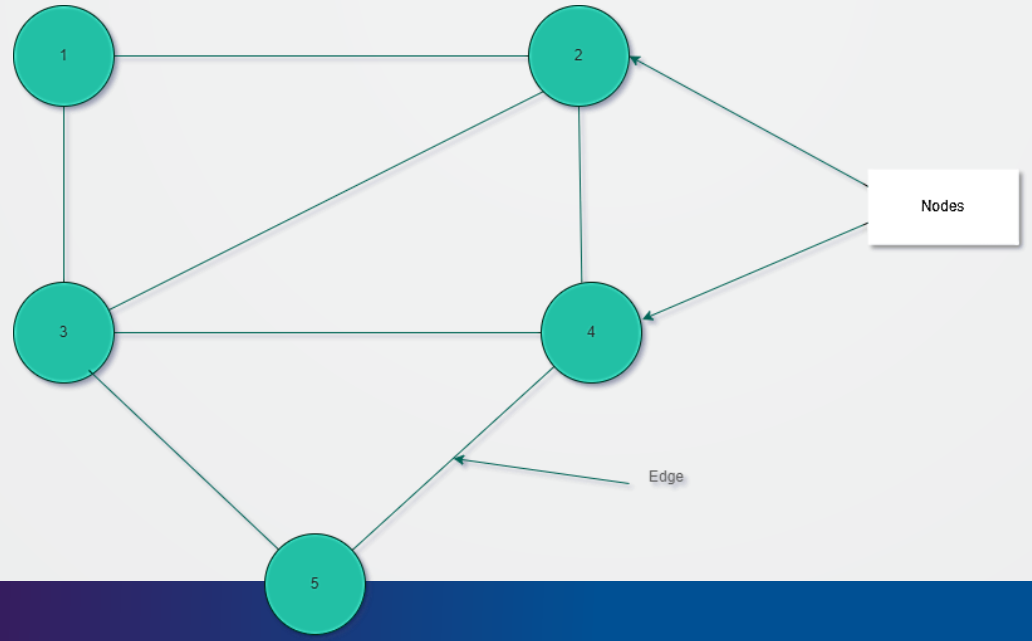
Trie



Introduction to Data Structures

Graph

- Graph is a data structure that consists of a collection of nodes (vertices) connected by edges.
- Graphs are used to represent relationships between objects and are widely used in computer science, mathematics, and other fields.
- Graphs can be used to model a wide variety of real-world systems, such as social networks, transportation networks, and computer networks.



Introduction to Data Structures

Applications of Data Structures:

Data structures are used in various fields such as:

- Operating system
- Graphics
- Computer Design
- Blockchain
- Genetics
- Image Processing
- Simulation,
- etc.

Recursive Algorithms

- Recursion is a technique used in computer science to solve big problems by breaking them into smaller, similar problems.
- *The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a* ***recursive function***.
- Using a recursive algorithm, certain problems can be solved quite easily.

Recursive Algorithms

- A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the solution.
- Since called function may further call itself, this process might continue forever. So, it is essential to provide a base case to terminate this recursion process.

Recursive Algorithms

Steps to Implement Recursion

Step1 - Define a base case: *Identify the simplest (or base) case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.*

Step2 - Define a recursive case: *Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.*

Step3 - Ensure the recursion terminates: *Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.*

Step4 - Combine the solutions: *Combine the solutions of the subproblems to solve the original problem.*

Recursive Algorithms

Example 1 : Sum of Natural Numbers (n=3)

Input *: n = 3*
Output *: 6*
Explanation *: The sum of first 3 natural numbers is 1+2+3 = 6.*

Input *: n = 7*
Output *: 28*
Explanation *: The sum of first 7 natural numbers is 1+2+3+4+5+6+7 = 28.*

Recursive Algorithms

Example 1 : Sum of Natural Numbers (n=3)

Base Case : At n == 1, it returns 1 for n = 3, the recursion reaches this after going through 3 → 2 → 1.

Recursive Case : Each call adds n to sum(n-1) , so sum(3) = 3 + sum(2), sum(2) = 2 + sum(1).

Recursive Algorithms

Example 1 : Sum of Natural Numbers (n=3)

```
public class Main {
    public static int sum(int n) {
        // base condition
        if (n == 1)
            return 1;
        return n + sum(n - 1);
    }
    public static void main(String[] args) {
        int n = 5;
        System.out.println(sum(n));
    }
}
```

Recursive Algorithms

Example 1 : Sum of Natural Numbers (n=3)

Execution Flow of the Recursive Solution

Calls are stacked as sum(3) → sum(2) → sum(1) before any addition happens.

Results are added back in reverse: sum(1) = 1, sum(2) = 3, sum(3) = 6.

Recursive Algorithms

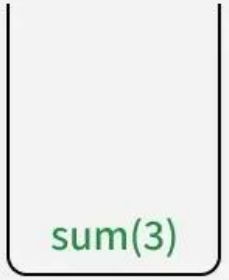
Example 1 : Sum of Natural Numbers (n=3)

Execution Flow of the Recursive Solution

01 Step | Recursively find sum of n natural number (n=3).

sum(3)

sum(3) is added into recursive stack.



02 Step | Function sum(3) calls sum(2)

sum(3)

3 + sum(2)

Recursive Stack

03 Step | sum(2) calls sum(1)

sum(3)

3 + sum(2)

2 + sum(1)

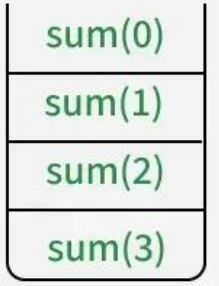
04 Step | sum(1) calls sum(0) and base condition is met.

sum(3)

3 + sum(2)

2 + sum(1)

1 + sum(0)



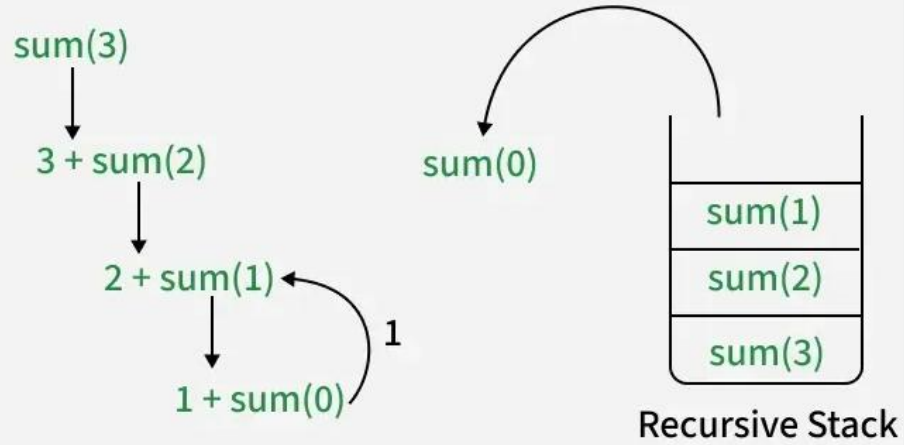
Recursive Stack

Recursive Algorithms

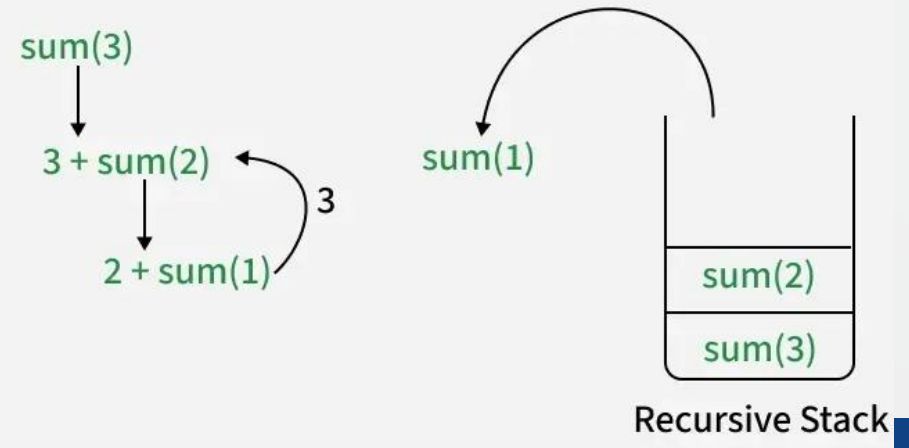
Example 1 : Sum of Natural Numbers (n=3)

Execution Flow of the Recursive Solution

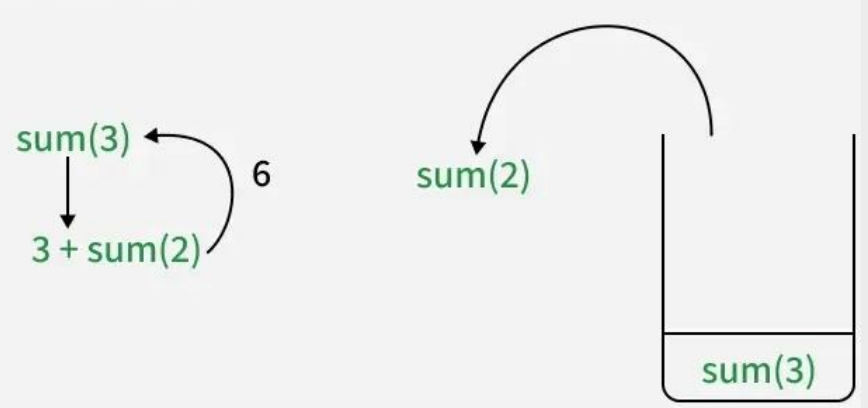
05 Step | Base case reached: sum(0)=0



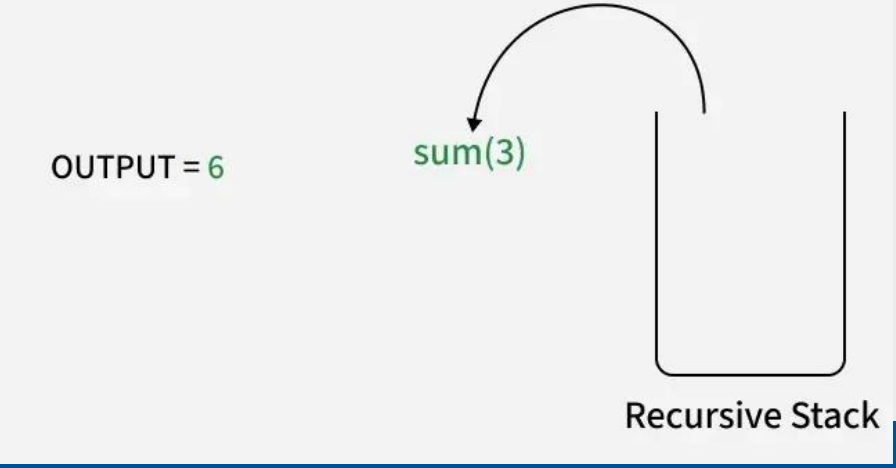
06 Step | Returning sum(1)



07 Step | Returning sum(2)



08 Step | Returning sum(3)



Recursive Algorithms

Need of Recursion

- Recursion helps in logic building. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.

- Recursive solutions work as a basis for Dynamic Programming and Divide and Conquer algorithms.

Recursive Algorithms

Example 2 : Factorial of a Number

- The factorial of a number n (where n >= 0) is the product of all positive integers from 1 to n.
- To compute the factorial recursively, we calculate the factorial of n by using the factorial of (n-1).
- The base case for the recursive function is when n = 0, in which case we return 1.

Recursive Algorithms

Example 2 : Factorial of a Number

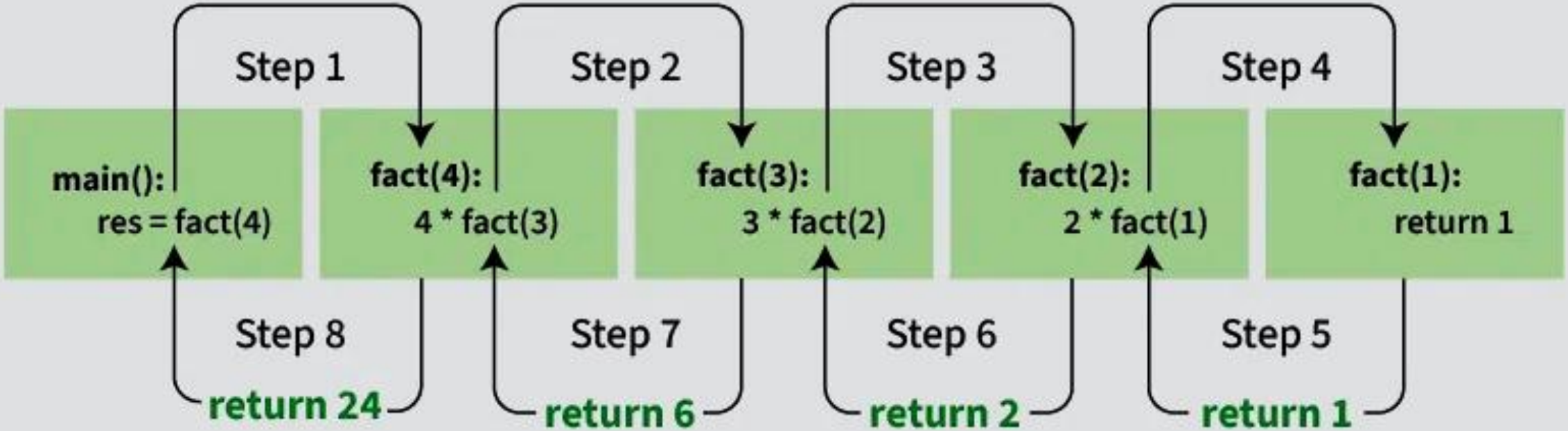
```
public class FactorialN {
    public static int fact(int n) {
        // BASE CONDITION
        if (n == 0)
            return 1;

        return n * fact(n - 1);
    }

    public static void main(String[] args) {
        System.out.println("Factorial of 5 : " + fact(5));
    }
}
```

Output

Factorial of 5 : 120

Recursive Algorithms

Example 2 : Factorial of a Number



Recursive Algorithms

Stack Overflow error occur in recursion

If the base case is not reached or not defined, then the stack overflow problem may arise.

```
int fact(int n)
{
// wrong base case (it may cause stack overflow).
if (n == 100)
return 1;
else
return n*fact(n-1);
}
```

In this example, if fact(10) is called, the function will recursively call fact(9), then fact(8), fact(7), and so on. However, the base case checks if n == 100. Since n will never reach 100 during these recursive calls, the base case is never triggered. As a result, the recursion continues indefinitely.

Recursive Algorithms

Stack Overflow error occur in recursion

- This continuous recursion consumes memory on the function call stack. If the system's memory is exhausted due to these unending function calls, a **stack overflow error** occurs.
- To prevent this, it's essential to define a proper base case, such as if **(n == 0)** to ensure that the recursion terminates and the function doesn't run out of memory.

Recursive Algorithms

Direct and Indirect recursion

- A function is called **direct recursive** if it calls itself directly during its execution. In other words, the function makes a recursive call to itself within its own body.
- An **indirect recursive function** is one that calls another function, and that other function, in turn, calls the original function either directly or through other functions. This creates a chain of recursive calls involving multiple functions, as opposed to direct recursion, where a function calls itself.

Recursive Algorithms

Direct and Indirect recursion

```
// An example of direct recursion
void directRecFun()
{
// Some code....

directRecFun();

// Some code...
}
```

```
// An example of indirect recursion
void indirectRecFun1()
{
// Some code...

indirectRecFun2();

// Some code...
}
void indirectRecFun2()
{
// Some code...

indirectRecFun1();

// Some code...
}
```

Recursive Algorithms

Memory allocation to different function calls in recursion:

Recursion uses more memory to store data of every recursive call in an internal function call stack.

- Whenever we call a function, its record is added to the stack and remains there until the call is finished.
- The internal systems use a stack because function calling follows LIFO structure, the last called function finishes first.

Recursive Algorithms

Memory allocation to different function calls in recursion:

```
// A Java program to demonstrate working of
// recursion
class MemoryRecursion {
    static void printFun(int test)
    {
        if (test < 1)
            return;
        else {
            System.out.printf("%d ", test);
            printFun(test - 1); // statement 2
            System.out.printf("%d ", test);
            return;
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        int test = 3;
        printFun(test);
    }
}
```

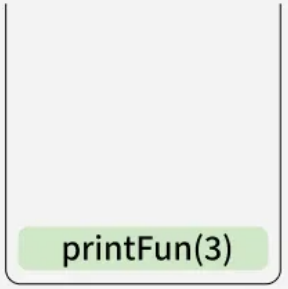
Recursive Algorithms

The memory stack grows with each function call and shrinks as the recursion unwinds, following the LIFO structure.

```
printFun(int test)
{
   if (test < 1)
      return;
   else {
      print (test);
      printFun(test - 1);
      print (test);
      return;
   }
}
```

1 printFun(3) is called by main

```
test = 3
printFun(3) {
print (3);
printFun(2);
print(3);
return;
}
```

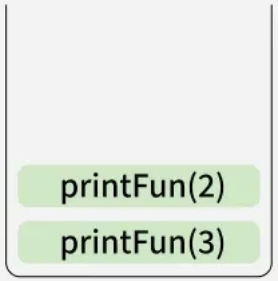


memory stack

Output: 3

2 printFun(3) calls printFun(2)

```
test = 2
printFun(2) {
print (2);
printFun(1);
print(2);
return;
}
```



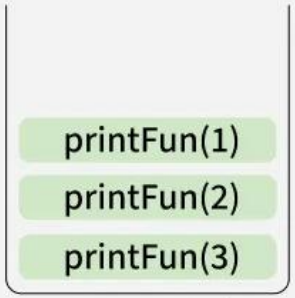
memory stack

Output: 3 2

printFun(2) calls printFun(1)

3

```
test = 1
printFun(1) {
print (1);
printFun(0);
print(1);
return;
}
```



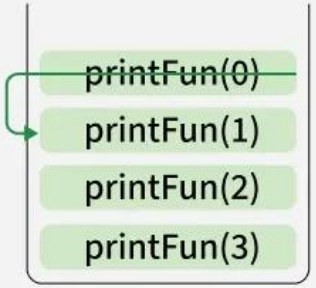
memory stack

Output: 3 2 1

printFun(1) calls printFun(0)

4

```
test = 0
printFun(0) {
if (test < 1)
   return;
}
```



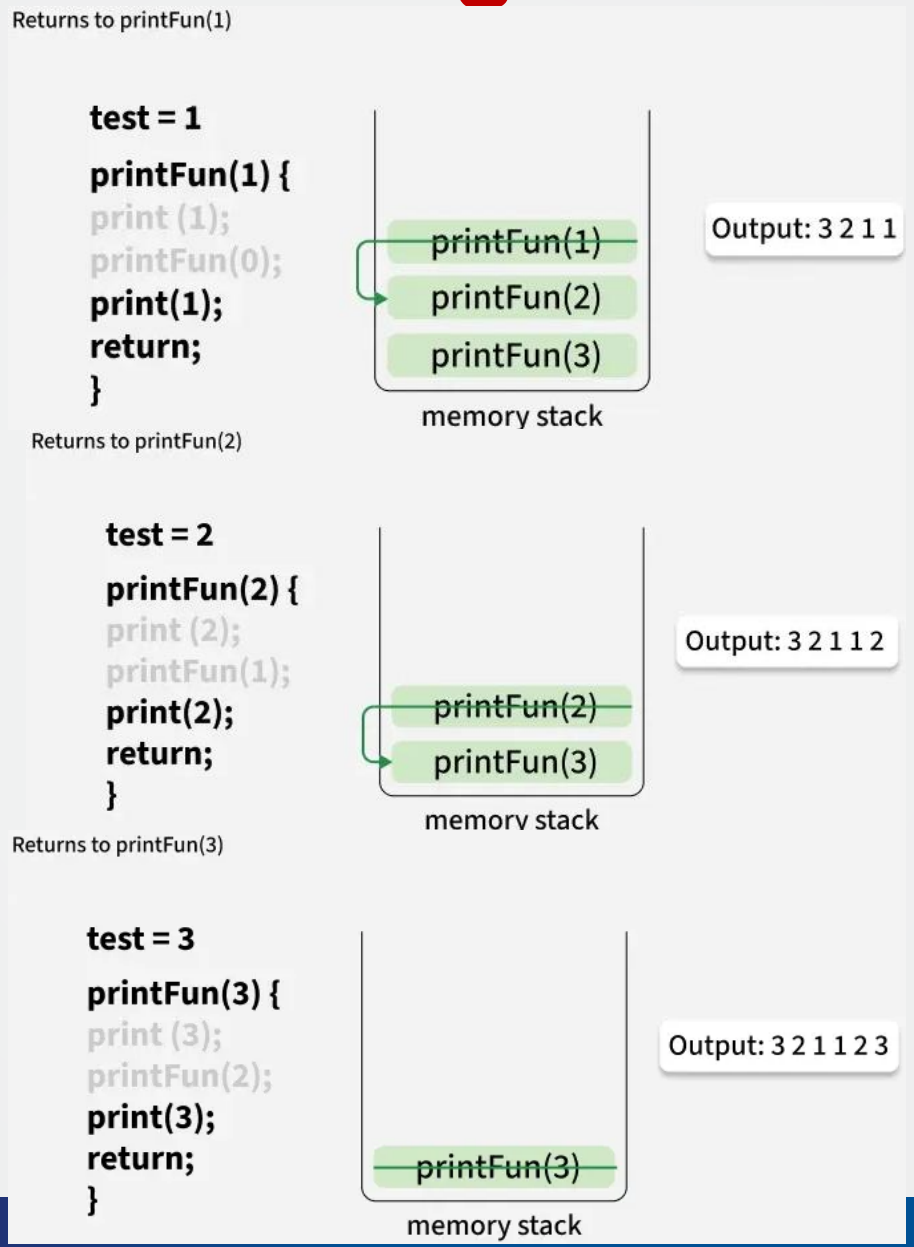
memory stack

Output: 3 2 1

After printFun(0) returns, the recursion uses backtracking, with each function call completing in reverse order

Recursive Algorithms

```
printFun(int test)
{
   if (test < 1)
      return;
   else {
      print (test);
      printFun(test - 1);
      print (test);
      return;
   }
}
```



Recursive Algorithms

Recursion vs Iteration Programming

Aspect	Recursion	Iteration
Definition	A function calls itself to solve smaller sub-problems	Repeated execution of statements using loops (`for`, `while`, `do-while`)
Code simplicity	Often **shorter and cleaner** for problems like trees, graphs, divide-and-conquer	Usually **longer but straightforward**
Readability	More **intuitive** for naturally recursive problems	Easier to understand for simple repetitive tasks
Memory usage	Uses **extra stack memory** due to function calls	Uses **constant memory** (no call stack overhead)
Execution speed	Generally **slower** due to function call overhead	Generally **faster**

Recursive Algorithms

Recursion vs Iteration Programming

Aspect	Recursion	Iteration
Risk of overflow	Risk of **stack overflow** for deep recursion	No risk of stack overflow
Debugging	Harder to debug and trace	Easier to debug
Performance control	Less control over memory and execution	Better control over performance
Problem suitability	Best for **tree traversal, DFS, divide & conquer**	Best for **loops, counting, linear tasks**
Termination condition	Needs a **base case**	Needs a **loop condition**
Optimization	Tail recursion *may* be optimized (language dependent)	Naturally optimized by compiler
Implementation complexity	Simpler logic, complex execution flow	Slightly complex logic, simple execution flow

Recursive Algorithms

Example 3 : Fibonacci with Recursion

Write a program and recurrence relation to find the **Fibonacci** series of n where n >= 0.

Mathematical Equation:

n if n == 0, n == 1;

fib(n) = fib(n-1) + fib(n-2) otherwise;

Recurrence Relation:

T(n) = T(n-1) + T(n-2) + O(1)

Recursive Algorithms

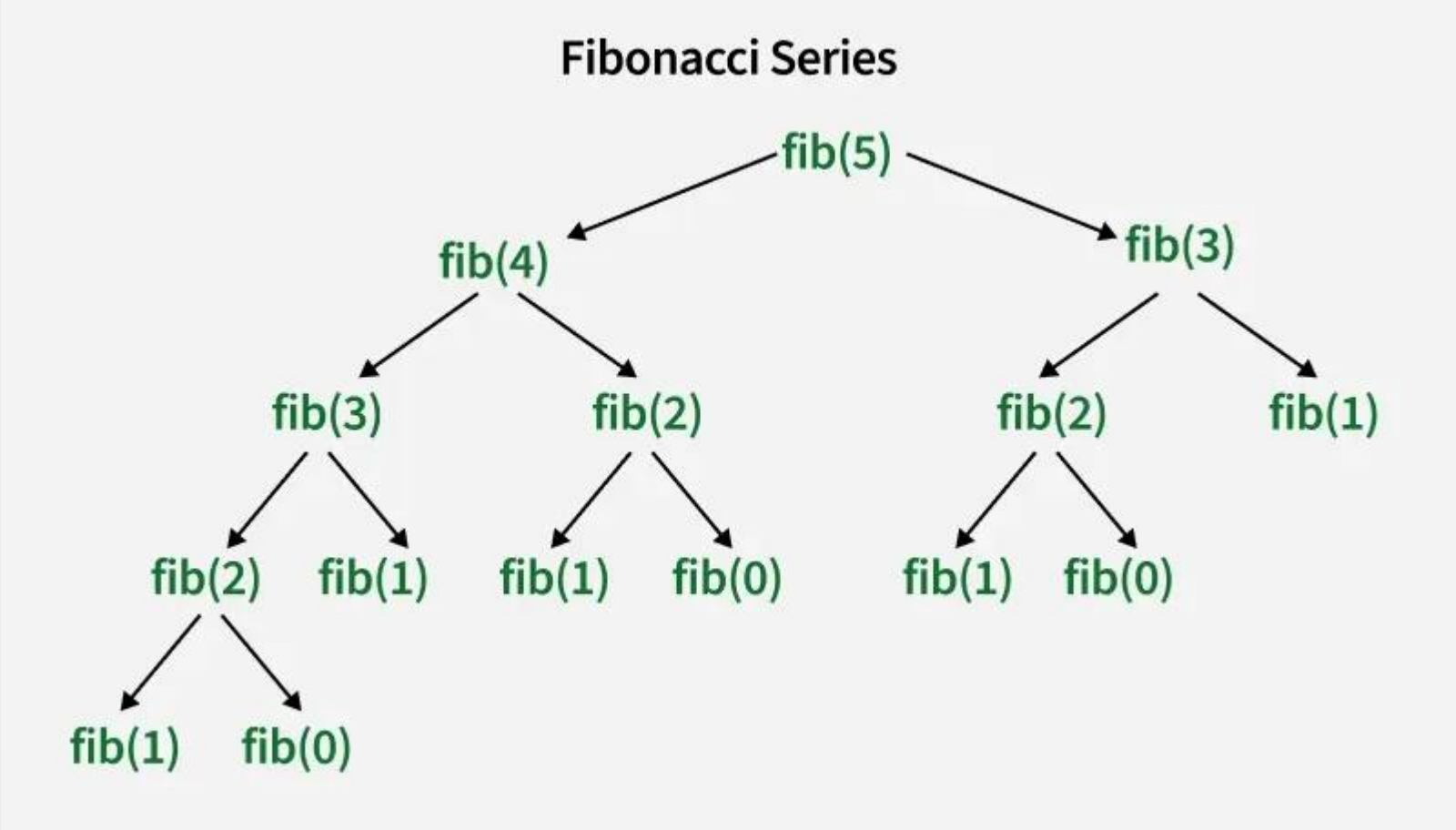
```
// Java code to implement Fibonacci series
import java.util.*;
class GFG
{
// Function for fibonacci
static int fib(int n)
{
    // Stop condition
    if (n == 0)
        return 0;
    // Stop condition
    if (n == 1 || n == 2)
        return 1;
    // Recursion function
    else
        return (fib(n - 1) + fib(n - 2));
}
```

```
// Driver Code
public static void main(String []args)
{

    // Initialize variable n.
    int n = 5;
    System.out.print("Fibonacci series of 5
numbers is: ");

    // for loop to print the fibonacci series.
    for (int i = 0; i < n; i++)
    {
        System.out.print(fib(i)+" ");
    }
}
}
```

Recursive Algorithms

Output

```
Fibonacci series of 5 numbers is: 0 1 1 2 3
```



Recursive Algorithms

Tail Recursion

- Tail recursion is defined as a recursive function in which the recursive call is the last statement that is executed by the function.
- So, basically nothing is left to execute after the recursion call.

```
// An example of tail recursive function
static void print(int n)
{
    if (n < 0)
        return;
    System.out.print(" " + n);
    // The last executed statement is recursive call
    print(n - 1);
}
```

Recursive Algorithms

Key Characteristics of Tail Recursion

- Recursive call at the **end**
- No pending operations
- Can be optimized into iteration (in some languages)

Advantages

- Uses less memory (with optimization)
- Faster than non-tail recursion
- Easy to convert into a loop

Recursive Algorithms

Non-Tail Recursion

A recursion is called **non-tail recursion** when **some operations remain after** the recursive call.

Key Characteristics

- Recursive call not the last statement
- Pending operations stored in stack
- Cannot be optimized easily

Disadvantages

- Higher memory usage
- Slower execution
- Risk of stack overflow

```
int sum(int n) {
    if (n == 0) return 0;
    return n + sum(n - 1);   // addition after recursion → non-tail
}
```

Recursive Algorithms

- **In tail recursion, the recursive call is the last operation, whereas in non-tail recursion, additional computation is performed after the recursive call.**

Abstract Data Types

- An **Abstract Data Type (ADT)** is a conceptual model that defines a set of operations and behaviors for a data structure, **without specifying how these operations are implemented** or how data is organized in memory.
- The definition of ADT only mentions what **operations are to be performed** but not **how** these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- **It is called "abstract" because it provides an implementation-independent view.**
- The process of providing only the essentials and hiding the details is known as abstraction.

Abstract Data Types

Features of ADT

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided. gives us a better conceptualization of the real world.
- **Robust:** The program is robust and
- **Better Conceptualization:** ADT has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.

Abstract Data Types

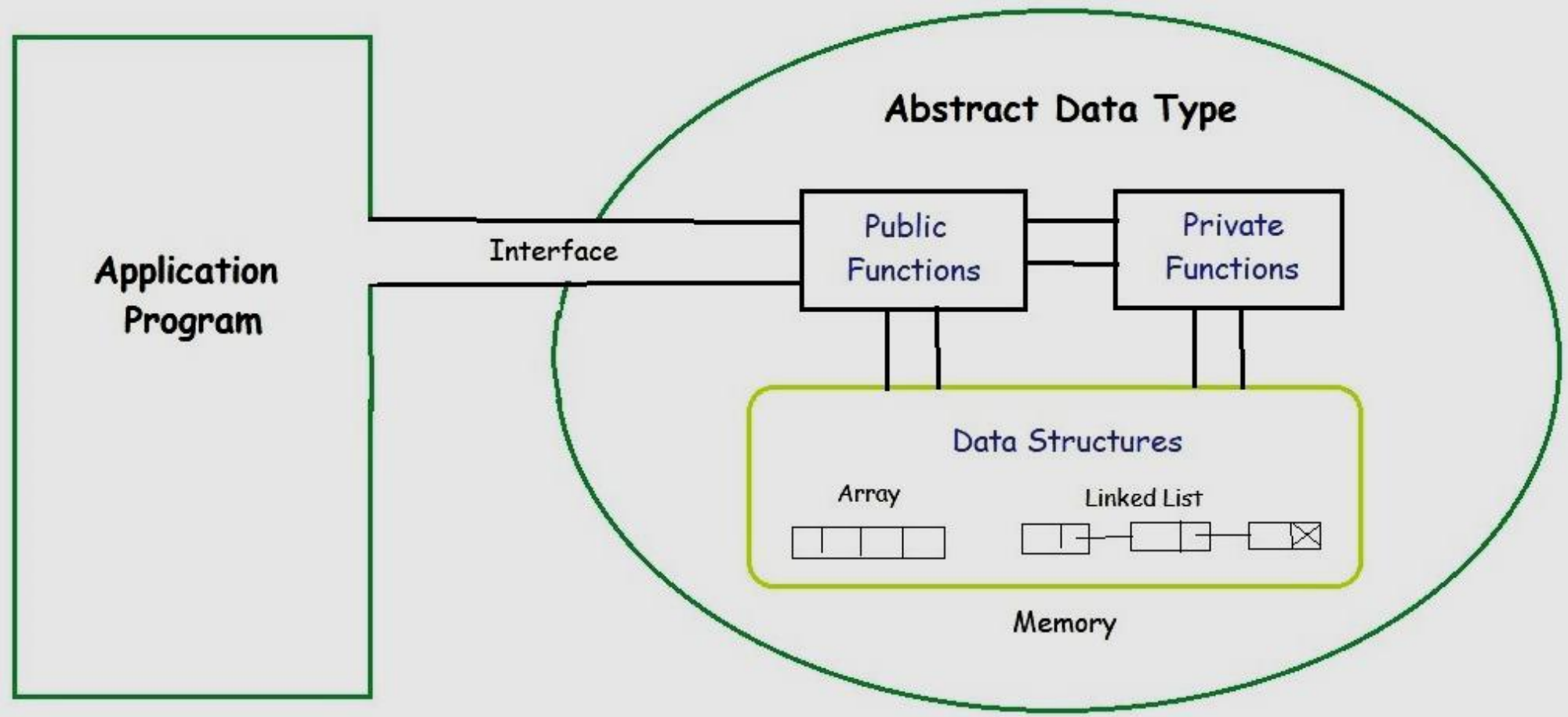
Features of ADT

- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

Abstract Data Types

Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.



Abstract Data Types

The key reasons to use ADTs in Java are listed below:

- **Encapsulation:** Hides complex implementation details behind a clean interface.
- **Reusability**: Allows different internal implementations (e.g., array or linked list) without changing external usage.
- **Modularity:** Simplifies maintenance and updates by separating logic.
- **Security:** Protects data by preventing direct access, minimizing bugs and unintended changes.

Abstract Data Types

Difference Between ADTs and UDTs

Aspect	Abstract Data Types (ADTs)	User-Defined Data Types (UDTs)
Definition	Defines a class of objects and the operations that can be performed on them, along with their expected behavior (semantics), but without specifying implementation details.	A custom data type created by combining or extending existing primitive types, specifying both structure and operations.
Focus	What operations are allowed and how they behave, without dictating how they are implemented.	How data is organized in memory and how operations are executed.

Abstract Data Types

Difference Between ADTs and UDTs

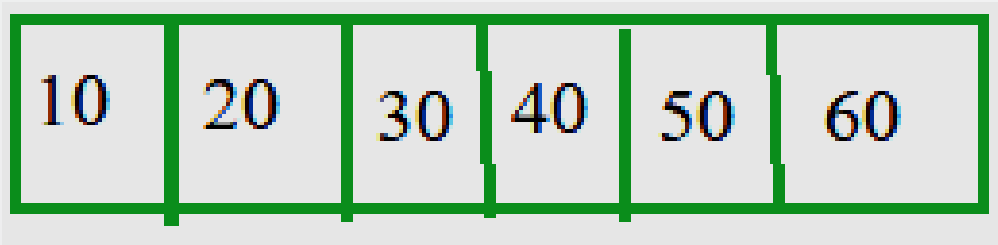
Aspect	Abstract Data Types (ADTs)	User-Defined Data Types (UDTs)
Purpose	Provides an abstract model to define data structures in a conceptual way.	Allows programmers to create concrete implementations of data structures using primitive types.
Implementation Details	Does not specify how operations are implemented or how data is structured.	Specifies how to create and organize data types to implement the structure.
Usage	Used to design and conceptualize data structures.	Used to implement data structures that realize the abstract concepts defined by ADTs.
Example	List ADT, Stack ADT, Queue ADT.	Structures, classes, enumerations, records.

Abstract Data Types

Examples of ADTs

1. List ADT

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations **without specifying the internal implementation**. It provides an ordered way to store, access, and modify data.



Abstract Data Types

1. List ADT Operations

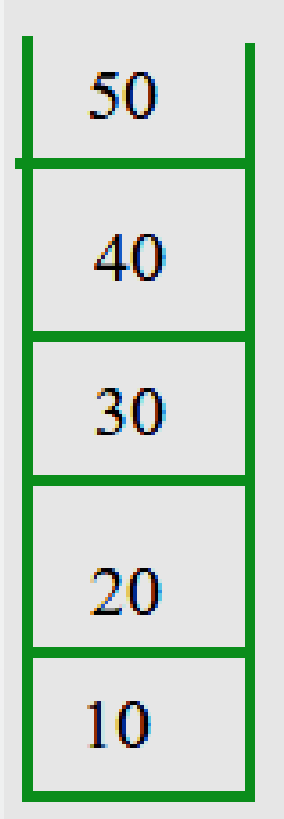
The List ADT need to store the required data in the sequence and should have the following operations:

- **get():** Return an element from the list at any given position.
- **insert():** Insert an element at any position in the list.
- **remove():** Remove the first occurrence of any element from a non-empty list.
- **removeAt():** Remove the element at a specified location from a non-empty list.
- **replace():** Replace an element at any position with another element.
- **size():** Return the number of elements in the list.
- **isEmpty():** Return true if the list is empty; otherwise, return false.
- **isFull():** Return true if the list is full, otherwise, return false. Only applicable in fixed-size implementations (e.g., array-based lists).

Abstract Data Types

2. Stack ADT

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.



Abstract Data Types

2. Stack ADT Operations

In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

- **push():** Insert an element at one end of the stack called the top.
- **pop():** Remove and return the element at the top of the stack, if it is not empty.
- **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
- **size():** Return the number of elements in the stack.
- **isEmpty():** Return true if the stack is empty; otherwise, return false.
- **isFull():** Return true if the stack is full; otherwise, return false. Only relevant for fixed-capacity stacks (e.g., array-based).

Abstract Data Types

3. Queue ADT Operations

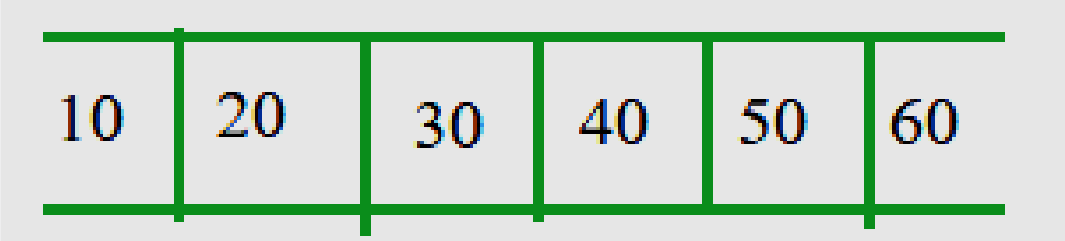
The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

- **enqueue():** Insert an element at the end of the queue.
- **dequeue():** Remove and return the first element of the queue, if the queue is not empty.
- **peek():** Return the element of the queue without removing it, if the queue is not empty.
- **size():** Return the number of elements in the queue.
- **isEmpty():** Return true if the queue is empty; otherwise, return false.

Abstract Data Types

3. Queue ADT

The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).



Abstract Data Types

Basic Operations of ADT

The basic operations depend on the type of ADT, but generally include:

Operation	Description
Create()	Creates an empty data structure
Insert()	Adds an element
Delete()	Removes an element
Search()	Finds an element
Access/Retrieve()	Gets a specific element
Update()	Modifies an element
Traverse()	Visits all elements
IsEmpty()	Checks if structure is empty
IsFull()	Checks if structure is full
Size()	Returns number of elements

Abstract Data Types

ADT vs Data Structure

ADT	Data Structure
Logical concept	Physical implementation
Specifies operations	Specifies storage
Independent of language	Language dependent
Example: Stack	Array, Linked List

Time & Space Complexity Analysis

Time Complexity Analysis

- **Time complexity** measures the **amount of time an algorithm takes to run** as a function of the **input size (n)**.
- It helps compare algorithms **independent of machine speed**.
- Expressed using **Big-O notation**.

Time & Space Complexity Analysis

Common Time Complexities

Big-O Notation	Name	Example
O(1)	Constant time	Accessing array element
O(log n)	Logarithmic	Binary search
O(n)	Linear	Linear search
O(n log n)	Linearithmic	Merge sort
$O(n^2)$	Quadratic	Bubble sort
$O(2^n)$	Exponential	Recursive Fibonacci
O(n!)	Factorial	Traveling salesman

Time & Space Complexity Analysis

Types of Time Complexity

Type	Meaning
Best Case	Minimum time required
Average Case	Expected time
Worst Case	Maximum time required

Time & Space Complexity Analysis

Types of Time Complexity

Example (Linear Search)

```
for(int i = 0; i < n; i++) {
    if(arr[i] == key)
        return i;
}
```

Case	Time Complexity
Best case	O(1)
Average case	O(n)
Worst case	O(n)

Time & Space Complexity Analysis

Space Complexity Analysis

Space complexity measures the **total memory used** by an algorithm, including:

- Input space
- Auxiliary (extra) space

Time & Space Complexity Analysis

Components of Space Complexity

Component	Description
Input space	Memory to store input data
Auxiliary space	Extra memory used during execution
Recursive stack	Memory for function calls

Example (Recursive Sum)

```
int sum(int n) {
    if (n == 0) return 0;
    return n + sum(n - 1);
}
```

- Time complexity → **O(n)**
- Space complexity → **O(n)** (recursive stack)

Time & Space Complexity Analysis

Time vs Space Complexity

Aspect	Time Complexity	Space Complexity
Measures	Execution time	Memory usage
Focus	Speed	Storage
Optimization	Faster algorithms	Memory-efficient algorithms
Unit	Number of operations	Memory units

Asymptotic notations

Asymptotic notations are mathematical tools used to **describe the growth rate of an algorithm's time or space complexity** as the input size (n) becomes very large.

- They ignore constants and lower-order terms.

Big-O Notation (O)

- **Big-O** gives the **upper bound** on the growth rate of an algorithm.
  It represents the **worst-case time complexity**.

"The algorithm will not take more than this time."

Asymptotic notations

An algorithm is **O(f(n))** if there exist constants $c > 0$ and n_0 such that:

$$T(n) \leq c \cdot f(n) \text{ for all } n \geq n_0$$

Example

```
for(int i = 0; i < n; i++) {
    System.out.println(i);
}
```

Time Complexity → **O(n)**

Asymptotic notations

Big-Ω Notation (Ω)

- **Big-Ω** gives the **lower bound** on the growth rate of an algorithm.
- It represents the **best-case time complexity**.

"The algorithm will take at least this time."

An algorithm is **Ω(f(n))** if there exist constants `c > 0` and n_0 such that:

$$T(n) \geq c \cdot f(n) \text{ for all } n \geq n_0$$

Example

Accessing an array element:

```
int x = arr[0];
```

Time Complexity → **Ω(1)**

Asymptotic notations

Big-Θ Notation (Θ)

- **Big-Θ** gives the **tight bound** on the growth rate of an algorithm.
- It represents **both upper and lower bounds**.

"The algorithm always grows at this rate."

An algorithm is **Θ(f(n))** if there exist constants $c_1, c_2 > 0$ and n_0 such that:

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \text{ for all } n \geq n_0$$

Example

```
for(int i = 0; i < n; i++) {
    System.out.println(i);
}
```

Time Complexity → **Θ(n)**

Asymptotic notations

Notation	Meaning	Case	Bound Type
Big-O (O)	Upper bound	Worst case	Maximum
Big-Ω (Ω)	Lower bound	Best case	Minimum
Big-Θ (Θ)	Tight bound	Average / Exact	Exact

Asymptotic notations

Example Summary (Linear Search)

Case	Complexity
Best case	Ω(1)
Worst case	O(n)
Average case	Θ(n)

Note

- Big-O → **worst-case analysis**
- Big-Ω → **best-case analysis**
- Big-Θ → **exact growth rate**
- Constants and lower terms are ignored

Asymptotic notations

Asymptotic notations describe the growth rate of an algorithm using Big-O for upper bound, Big-Ω for lower bound, and Big-Θ for tight bound.

Programs

1a. Sum of last digits of two given numbers
1c. Combine Strings
1f. Alternate String Combiner
1h. Leaders in an array

Programs

1a. Sum of last digits of two given numbers

Rohit wants to add the last digits of two given numbers. For example, If the given numbers are 267 and 154, the output should be 11.

Below is the explanation –

Last digit of the 267 is 7

Last digit of the 154 is 4

Sum of 7 and 4 = 11

Write a program to help Rohit achieve this for any given two numbers.

The prototype of the method should be –

int addLastDigits(int input1, int input2);

where input1 and input2 denote the two numbers whose last digits are to be added.

Note: The sign of the input numbers should be ignored.

Programs

1a. Sum of last digits of two given numbers

if the input numbers are 267 and 154, the sum of last two digits should be 11
if the input numbers are 267 and -154, the sum of last two digits should be 11
if the input numbers are -267 and 154, the sum of last two digits should be 11
if the input numbers are -267 and -154, the sum of last two digits should be 11

Input: 267 154 **Output**: 11
Input: 267 -154 **Output**: 11
Input: -267 154 **Output**: 11
Input: -267 -154 **Output**: 11

Programs

```
import java.util.Scanner;
class AddLastDigits {
    static int addLastDigits(int input1, int input2) {
        int lastDigit1 = Math.abs(input1) % 10;
        int lastDigit2 = Math.abs(input2) % 10;
        return lastDigit1 + lastDigit2;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        System.out.println(addLastDigits(a, b));
    }
}
```

Programs

1c. Combine Strings

Given 2 strings, a and b, return a new string of the form short+long+short, with the shorter string on the outside and the longer string in the inside. The strings will not be the same length, but they may be empty (length 0).
If input is "hi" and "hello", then output will be "hihellohi"
Input:
Enter the first string: “hi”
Enter the second string: “hello”
Output: “hihellohi”

Input:
Enter the first string: “iare”
Enter the second string: “college”
Output: “iarecollegeiare

Programs

```
import java.util.Scanner;
class ShortLongShort {
    static String shortLongShort(String a, String b) {
        if (a.length() < b.length()) {
            return a + b + a;
        } else {
            return b + a + b;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the first string: ");
        String a = sc.nextLine();
        System.out.print("Enter the second string: ");
        String b = sc.nextLine();
        System.out.println("Output: " + shortLongShort(a, b));
    }
}
```

Programs

1f. Alternate String Combiner

Given two strings, a and b, print a new string which is made of the following combination-first character of a, the first character of b, second character of a, second character of b and so on. Any characters left, will go to the end of the result.

Hello,World
HWeoIrllod

Input: “Hello,World”
Output: “HWeoIrllod”

Input: “Iare,College”
Output: “ICaorlelege”

Programs

```
import java.util.Scanner;
class MergeStrings {
    static String mergeStrings(String a, String b) {
        StringBuilder result = new StringBuilder();
        int minLength = Math.min(a.length(), b.length());
        for (int i = 0; i < minLength; i++) {
            result.append(a.charAt(i));
            result.append(b.charAt(i));
        }
        if (a.length() > minLength)
            result.append(a.substring(minLength));
        else
            result.append(b.substring(minLength));
        return result.toString();
    }
```

```
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter the first string: ");
    String a = sc.nextLine();

    System.out.print("Enter the second string: ");
    String b = sc.nextLine();

    System.out.println("Output: " + mergeStrings(a, b));
     }
}
```

Programs

1h. Leaders in an array

Given an array arr of n positive integers, your task is to find all the leaders in the array. An element of the array is considered a leader if it is greater than all the elements on its right side or if it is equal to the maximum element on its right side. The rightmost element is always a leader.

Input: n = 6, arr[] = (16, 17, 4, 3, 5, 2} Output: 17 5 2

Input: n = 5, arr[] = {10, 4, 2, 4, 1} Output: 10 4 4 1

Input: n = 4, arr[] = {5, 10, 20, 40} Output: 40

Input: n = 4, arr[] = {30, 10, 10, 5} Output: 30 10 10 5

```
import java.util.Scanner;
class LeadersInArray {
    static void printLeaders(int[] arr, int n) {
        int[] leaders = new int[n];   // array to store leaders
        int count = 0;
        // rightmost element is always a leader
        int max = arr[n - 1];
        leaders[count++] = max;
        // traverse from right to left
        for (int i = n - 2; i >= 0; i--) {
            if (arr[i] >= max) {
                max = arr[i];
                leaders[count++] = max;
            }
        }
        // print leaders in correct order
        System.out.print("Leaders: ");
        for (int i = count - 1; i >= 0; i--) {
            System.out.print(leaders[i] + " ");
        }
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of elements: ");
        int n = sc.nextInt();

        int[] arr = new int[n];
        System.out.println("Enter array elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        printLeaders(arr, n);
    }
}
```

Programs

```
import java.util.*;
class LeadersInArray {
    static void printLeaders(int[] arr, int n) {
        ArrayList<Integer> leaders = new ArrayList<>();
        int maxFromRight = arr[n - 1];
        leaders.add(maxFromRight);
        for (int i = n - 2; i >= 0; i--) {
            if (arr[i] >= maxFromRight) {
                maxFromRight = arr[i];
                leaders.add(maxFromRight);
            }
        }
        // reverse to maintain left-to-right order
        Collections.reverse(leaders);
        for (int x : leaders)
            System.out.print(x + " ");
    }
    public static void main(String[] args) {
        int[] arr = {16, 17, 4, 3, 5, 2};
        int n = arr.length;
        printLeaders(arr, n);
    }
}
```