ਸੰਤ ਲੌਂਗੋਵਾਲ ਇੰਸਟੀਚਿਊਟ ਆਫ ਇੰਜੀਨੀਅਰਿੰਗ ਐਂਡ ਟੈਕਨਾਲੋਜੀ
संत लौंगोवाल अभियांत्रिकी एवं प्रौद्योगिकी संस्थान
Sant Longowal Institute of Engineering and Technology
(Deemed-to-be-University, under Ministry of Education, Govt. of India)

Course Material for Data Structure

Subject Code: CS215

Class: ICD III Semester

DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that this manual is a bonafide record of **Course Material for Data Structure** in the **Data Structure** in **3rd Semester of II Year ICD (CSE) program** during the academic year **2024-24**. This book was prepared by **Dr. Amar Nath (Assistant Professor),** Department of Computer Science and Engineering.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

PREFACE

This **"Data Structures" Course Material** is crafted to provide a comprehensive understanding of data structure design and analysis, with a focus on practical implementation using the C programming language. It assumes that readers have a foundational knowledge of C and similar procedural languages. Data structures have become a pivotal component in the IT industry, especially in areas like system-level software development.

The material is tailored to enhance procedural programming skills and is enriched with numerous exercises and their solutions, ensuring that students can grasp the concepts quickly and effectively. It serves as a valuable resource for Computer Science and Engineering students, aiding in the practical understanding of data structures. Feedback and suggestions from readers are highly appreciated to improve future editions, as continual refinement is essential for providing the best learning experience.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

ACKNOWLEDGEMENT

It was a wonderful experience working on the "**Course Material for Data Structure**". First, I would like to express my sincere gratitude to **Prof. Birmohan**, Head of the Department of Computer Science and Engineering, for his continuous support and technical guidance in preparing this document. I am deeply indebted and would like to acknowledge the invaluable support and patronage **of Prof. Mani Kant Paswan**, Director of the institute, for providing me with this excellent opportunity and his constant encouragement throughout the process. Finally, I extend my heartfelt thanks to the entire faculty of the CSE Department, whose inspiration and assistance helped me achieve this goal.

BY

Dr. Amar Nath

AP, CSE, SLIET LONGOWAL

Title of the course : **Data Structures**

Subject Code : **CS-215**

Weekly load : 7 Hrs LTP 3-0-4

Credit : 5 (Lecture 3, Practical 2)

Course Outcomes: At the end of the course, the student will be able to:

CO1	Understand the concept of abstract data types and problem solving.
CO2	Develop and analyze algorithms for arrays, stacks, queues and linked list.
CO3	Develop algorithms for binary tree.
CO4	Implement sorting and searching algorithms
CO5	Implement symbol table using hashing techniques

CO/PO Mapping : (Strong(S)/Medium(M)/Weak(W) indicates strength of correlation)										
COs	Programme Outcomes (POs)									
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10
CO1		S	W							S
CO2		S	M							S
CO3		S	W							S
CO4		S	M							S
CO5		S	W							S

Theory

Unit	Main Topics	Course outlines	Lecture(s)
Unit-1	1. Introduction	Data Representation, Abstract data Types, Data Structure and Structured Types, Atomic Type ,Difference between Abstract Data Types, Data Types And Data Structures, Data Types, Linear data type, Non- Linear data type, Primitive data type, Non primitive data type.	05
	2.Fundamental Notations	Problem solving concept, top down and bottom up design, structured programming, Concept of data types, variables and constants, Concept of pointer variables and constants.	05
	3. Arrays	Concept of Arrays, Single dimensional array, Two dimensional array storage strategies of multidimensional arrays.	04
	4. Linked Lists	Introduction to linked list and double linked list, Representation of linked lists in Memory, Traversing a linked list, Searching linked list, Insertion and deletion into linked list	07
Unit-2	5. Stacks	Introduction to stacks, Representation of stacks, Implementation of stacks, Uses of stacks,	04
	6. Queues and Recursions	Introduction to queues, Implementation of queues (with Algorithm), Circular Queues, De-queues, Recursion.	07
	7. Binary search	Traversing Binary Trees (Pre order, Post order and In order),	08

Syllabus in detail

Unit 1: Introduction to Data Structures

1. **Data Representation**
   - **Data Representation** is the form in which data is stored, processed, and transmitted. It can be numerical (integers, floating points), characters (ASCII), or more complex structures like arrays, trees, or graphs.
2. **Abstract Data Types (ADTs)**
   - **Abstract Data Types** are a theoretical model for data structures with hidden implementation details. The focus is on what operations are supported rather than how they are performed. ADTs like stacks, queues, and lists provide a blueprint for specific implementations.
3. **Data Structures and Structured Types**
   - **Data Structure** is an organized way to store, manage, and retrieve data. It focuses on efficient operations like insertion, deletion, and traversal.
   - **Structured Types** include arrays, structs, etc., where multiple elements are grouped and treated as a single entity.
4. **Atomic Type**
   - An **Atomic Type** represents simple, indivisible data like integers, characters, etc.
5. **Difference between ADT, Data Types, and Data Structures**
   - **ADT**: Defines the operations on the data without specifying the implementation.
   - **Data Types**: Specific sets of values and operations (e.g., int, char).
   - **Data Structures**: How data is stored (e.g., arrays, linked lists).
6. **Types of Data Structures**
   - **Primitive Data Types**: Integer, float, character, pointer.
   - **Non-Primitive Data Types**: Arrays, stacks, queues, linked lists, trees, graphs.
   - **Linear Data Types**: Data elements are arranged sequentially (e.g., arrays, linked lists).
   - **Non-Linear Data Types**: Data elements are arranged hierarchically (e.g., trees, graphs).

Unit 2: Fundamental Notations

1. **Problem-Solving Concepts**
   - Involves breaking down a problem into smaller manageable parts. Techniques include **top-down** (starting from the general and moving to specifics) and **bottom-up** (starting with specifics and moving to the general) approaches.
2. **Structured Programming**

- Involves organizing code into functions and modules for readability and maintainability. It includes the use of loops, conditionals, and functions.

3. **Data Types, Variables, and Constants**
   - **Data Types**: Define the type of data a variable can hold (e.g., int, float).
   - **Variables**: Named storage that can hold data.
   - **Constants**: Fixed values that do not change during the program execution.
4. **Pointer Variables and Constants**
   - **Pointers** store the memory address of a variable. They provide direct access to memory locations.
   - **Pointer Constants**: Pointers that cannot be changed after initialization.

Unit 3: Arrays

1. **Concept of Arrays**
   - An **array** is a collection of elements of the same type stored at contiguous memory locations. Each element is accessed using an index.
2. **Single Dimensional Arrays**
   - An array with a single index (e.g., int arr[5];).
3. **Two Dimensional Arrays**
   - A 2D array is like a matrix, with rows and columns (e.g., int arr[3][4];).
4. **Storage Strategies for Multidimensional Arrays**
   - Multidimensional arrays can be stored in **row-major order** (rows first) or **column-major order** (columns first).

Unit 4: Linked Lists

1. **Introduction to Linked List**
   - A **linked list** is a dynamic data structure where each element (node) points to the next. It can grow or shrink dynamically.
2. **Doubly Linked List**
   - In a **doubly linked list**, each node contains pointers to the next and previous nodes, allowing traversal in both directions.
3. **Memory Representation**
   - Linked lists are stored in **non-contiguous** memory locations. Each node contains data and a pointer to the next node.
4. **Operations on Linked Lists**
   - **Traversal**: Visit each element on the list.
   - **Searching**: Finding an element.

- **Insertion**: Adding a new element.
- **Deletion**: Removing an existing element.

Unit 5: Stacks

1. **Introduction to Stacks**
   - A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. The last element inserted is the first to be removed.
2. **Stack Representation**
   - Stacks can be represented using arrays or linked lists.
3. **Stack Operations**
   - **Push**: Insert an element.
   - **Pop**: Remove an element.
   - **Top/Peek**: View the top element without removing it.
4. **Uses of Stacks**
   - Stacks are used in function calls, expression evaluation, parsing, and more.

Unit 6: Queues and Recursion

- **Introduction to Queues**
  - A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. The first element inserted is the first to be removed.
- **Implementation of Queues**
  - Queues can be implemented using arrays or linked lists.
- **Types of Queues**
  - **Circular Queue**: The last position is connected to the first, forming a circle.
  - **Dequeue**: A double-ended queue where elements can be added or removed from both ends.
- **Recursion**
  - **Recursion** is a process where a function calls itself. It is essential in problem-solving techniques like divide and conquers.

Unit 7: Binary Search Tree

1. **Binary Trees**
   - A **binary tree** is a hierarchical structure where each node has at most two children.

2. **Tree Traversals**
    - **Pre-order**: Visit root, left subtree, right subtree.
    - **In order**: Visit the left subtree, root, and right subtree.
    - **Post-order**: Visit the left subtree, right subtree, and root.
3. **Binary Search Tree (BST)**
    - A **BST** is a binary tree where each node's left child contains a value less than the node, and the right child contains a value greater than the node.
4. **Operations on BST**
    - **Searching**: Find an element.
    - **Insertion**: Add an element.
    - **Deletion**: Remove an element.

Unit 8: Sorting and Searching Algorithms

1. **Search Algorithms**
    - **Linear Search**: Sequentially checks each element.
    - **Binary Search**: Divides the array in half repeatedly to find the element (requires sorted array).
2. **Sorting Algorithms**
    - **Bubble Sort**: Repeatedly swaps adjacent elements if they are in the wrong order.
    - **Insertion Sort**: Builds a sorted array by inserting elements into the correct position.
    - **Quick Sort**: A divide-and-conquer algorithm that partitions the array into sub-arrays based on a pivot.
    - **Selection Sort**: Repeatedly selects the minimum element and swaps it with the front element.
    - **Merge Sort**: A divide-and-conquer algorithm that splits the array and merges the sorted sub-arrays.
    - **Heap Sort**: Sorts using a binary heap structure, focusing on the maximum or minimum element.

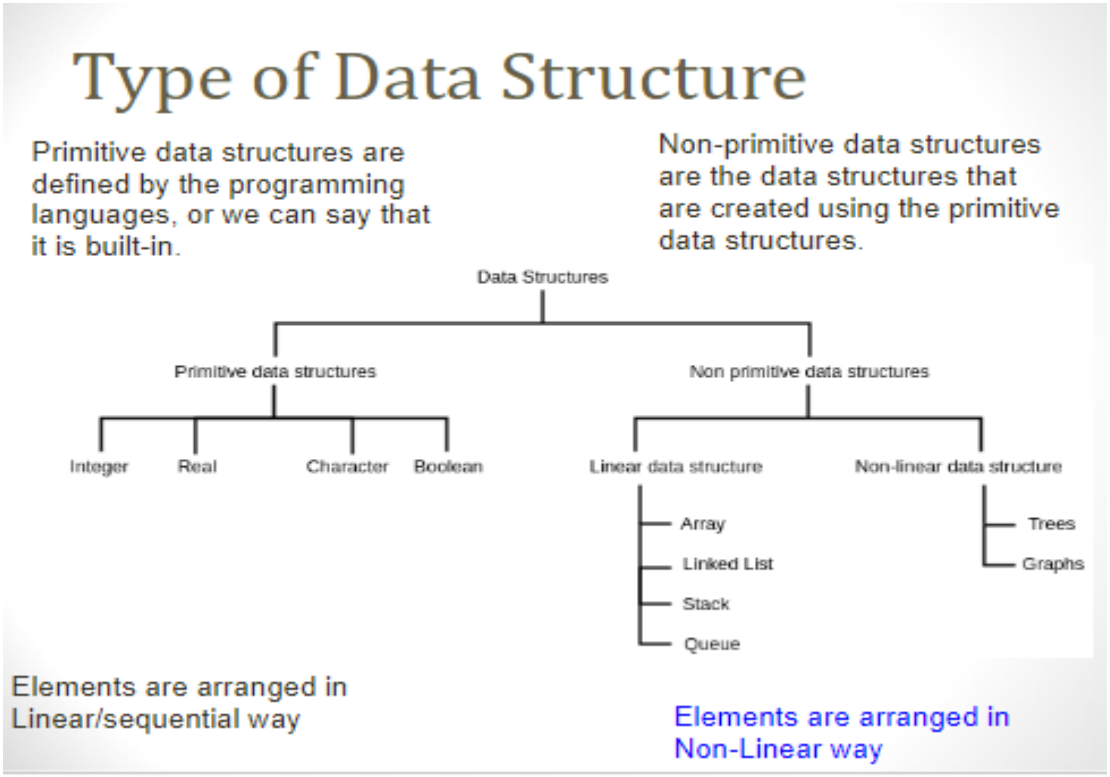
INDEX

Unit 1: Introduction to Data Structures

7. **Data Representation**
   - **Data Representation** is the form in which data is stored, processed, and transmitted. It can be numerical (integers, floating points), characters (ASCII), or more complex structures like arrays, trees, or graphs.
8. **Abstract Data Types (ADTs)**
   - **Abstract Data Types** are a theoretical model for data structures with hidden implementation details. The focus is on what operations are supported rather than how they are performed. ADTs like stacks, queues, and lists provide a blueprint for specific implementations. The definition of ADT only mentions what operations are to be performed rather than how these operations will be implemented.
   - It is called "abstract" because it gives an implementation-independent view.
   - An example of ADT is Queue and **Stack ADT**
   - **Stack ADT**
     i. push() – Insert an element called top at one end of the stack.
     ii. pop() – Remove and return the element at the top of the stack if it is not empty.
     iii. peek() – Return the element at the top of the stack without removing it if the stack is not empty.
     iv. size() – Return the number of elements in the stack.
     v. isEmpty() – Return true if the stack is empty, otherwise, return false.
     vi. isFull() – Return true if the stack is full; otherwise, return false.
   - **Queue ADT**
     - 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 queue element 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.
     - isFull() – Return true if the queue is full; otherwise, return false.

9. **Data Structures and Structured Types**
   - **Data Structure** is an organized way to store, manage, and retrieve data. It focuses on efficient operations like insertion, deletion, and traversal.
   - **Structured Types** include arrays, structs, etc., where multiple elements are grouped and treated as a single entity.

10. **Atomic Type**
    - An **Atomic Type** represents simple, indivisible data like integers, characters, etc.
11. **Difference between ADT, Data Types, and Data Structures**
    - **ADT**: Defines the operations on the data without specifying the implementation.
    - **Data Types**: Specific sets of values and operations (e.g., int, char).
    - **Data Structures**: How data is stored (e.g., arrays, linked lists).
12. **Types of Data Structures**
    - **Primitive Data Types**: Integer, float, character, pointer.
    - **Non-Primitive Data Types**: Arrays, stacks, queues, linked lists, trees, graphs.
    - **Linear Data Types**: Data elements are arranged sequentially (e.g., arrays, linked lists).
    - **Non-Linear Data Types**: Data elements are arranged hierarchically (e.g., trees, graphs).



Unit 2: Fundamental Notations

Problem-Solving Concepts

Problem-solving is solving complex issues by breaking them down into smaller, more manageable parts. Effective problem-solving in programming involves a structured approach to analyzing and addressing each part of the problem. The two major techniques used in problem-solving are **Top-Down Design** and **Bottom-Up Design**. Let's explore these concepts in detail:

1. **Top-Down Design (Stepwise Refinement):**

- **Definition**: Top-down design involves breaking down a complex problem into smaller sub-problems or components. Each component is further divided into more specific sub-components until all parts are manageable enough to be solved directly.
- **Process**:
  - **Step 1**: Identify the overall problem or goal.
  - **Step 2**: Divide the problem into smaller tasks or modules.
  - **Step 3**: Continue dividing tasks into sub-tasks until they can be implemented directly.
  - **Step 4**: Solve or implement the smallest sub-problems first, then combine them to solve larger tasks.
- **Example**:
  - Suppose you need to design software for a library management system.
  - In the top-down approach, the main problem is "Managing a Library."
  - Break this down into smaller problems such as "Add a Book," "Remove a Book," "Search for a Book," etc.
  - Each smaller problem can be broken down further, such as "Search for a Book" can have "Search by Title" and "Search by Author."
- **Advantages**:
  - It is easier to manage and debug as each module can be handled independently.
  - Clear modular structure, improving code reusability.
- **Disadvantages**:
  - May overlook details of the implementation in the early stages.
  - Requires complete understanding of the problem from the beginning.

2. Bottom-Up Design:

- **Definition**: Bottom-up design is an approach where you start by solving small, specific problems and then builds up toward solving the overall larger problem. In this approach, you implement individual components or modules and then integrate them to form a complete solution.
- **Process**:
  - **Step 1**: Identify smaller, independent tasks or modules that can be developed first.
  - **Step 2**: Develop the solution for these smaller components.
  - **Step 3**: Integrate the individual components or modules to form larger systems.
  - **Step 4**: Continue integrating modules until the complete system is formed.
- **Example**:
  - Using the library management system as an example, the bottom-up approach would involve developing individual components first, such as creating a function to handle "Search by Title," another function for "Search by Author," etc.
  - After these smaller functions work independently, they are combined to form a comprehensive "Search" feature.
- **Advantages**:
  - It lets you focus on smaller details and combine them into a larger picture.
  - Encourages the reuse of existing components or libraries.
- **Disadvantages**:
  - This may lead to issues in combining components if the overall system architecture needs to be clarified from the beginning.
  - It may take longer to identify the complete solution.

3. Structured Programming:

- **Definition**: Structured programming is a programming paradigm that improves clarity, quality, and development time using clear, logical structures such as sequences, loops, and conditionals.
- **Key Principles**:
  - **Sequence**: Instructions are executed in a step-by-step manner.
  - **Selection**: Decision-making processes such as if-else statements.
  - **Iteration**: Repeating processes using loops (for, while).
- **Advantages**:
  - It makes programs more readable and maintainable.
  - Encourages modular programming and promotes the use of functions and procedures.

4. Concept of Data Types:

- **Definition**: A data type specifies the data that can be stored and manipulated within a program. Data types help define the operations that can be performed on that data.
- **Basic Data Types**:

- **Primitive Data Types**: Basic building blocks like int, float, char, and double.
- **Non-Primitive Data Types**: include structures like arrays, linked lists, and user-defined types like struct and enum.

5. Concept of Variables and Constants:

- **Variables**: A variable is a named memory location used to store data that can be changed during program execution.
  - Example: int age = 25; (Here, age is a variable storing the value 25.)
- **Constants**: A constant is a fixed value that cannot be changed during program execution.
  - Example: const int MAX = 100; (Here, MAX is a constant whose value is fixed at 100.)
  -

5. **Structured Programming**
   - Involves organizing code into functions and modules for readability and maintainability. It includes the use of loops, conditionals, and functions.
6. **Data Types, Variables, and Constants**
   - **Data Types**: Define the type of data a variable can hold (e.g., int, float).
   - **Variables**: Named storage that can hold data.
   - **Constants**: Fixed values that do not change during the program execution.
7. **Pointer Variables and Constants**
   - **Pointers** store the memory address of a variable. They provide direct access to memory locations.
   - **Pointer Constants**: Pointers that cannot be changed after initialization.

1. Pointer Variables:

A **pointer** is a special type of programming variable that stores another variable's memory address. Instead of holding the actual data, a pointer "points" to the memory location where the data is stored. Pointers are widely used in programming for efficient memory management, dynamic memory allocation, and manipulating arrays and structures.

Declaration of a Pointer:

- A pointer is declared using the * symbol, which specifies that the variable is a pointer type.

Syntax:

data_type *pointer_name;

int a = 10; // Regular integer variable
int *p = &a; // Pointer 'p' storing the address of variable 'a'

In this example:

- a is a regular integer variable that stores the value 10.
- p is a pointer to an integer that stores the **memory address** of a. The &a, an operator gives the memory address of a, and p holds this address.

Dereferencing a Pointer: To access the value stored at the memory address that the pointer is pointing to, we use the * operator (called **dereferencing**).

printf("Value of a: %d\n", *p); // Output: 10 (Dereferencing pointer 'p' to get the value of 'a')

Here, *p returns the value stored at the memory location that p points to (i.e., the value of a).

2. Pointer Constants:

A **pointer constant** refers to a pointer whose value (i.e., the address it points to) cannot be changed after it has been initialized. There are two types of constants when working with pointers:

- **Constant Pointer (Pointer to a constant)**: The pointer can point to different variables, but the value of the pointer is pointing to cannot be changed through that pointer.
- **Pointer Constant (Constant pointer)**: The pointer always points to the same memory address after initialization, but the value at that address can be modified.

a. Constant Pointer:

A constant pointer ("pointer to a constant") points to a constant value. This means you cannot change the value the pointer points to, but the pointer can point to a different location.

```
const data_type *pointer_name;

int a = 10;
const int *p = &a;

// *p = 20;  // Error: Cannot modify value pointed to by 'p'

int b = 30;
p = &b;  // OK: Pointer 'p' can point to a different location
```

In this example:

- p is a pointer to a constant integer. The value at the address p is pointing to cannot be modified (i.e., you cannot change the value of a through p).
- However, p itself can point to another variable, such as b.

b. Pointer Constant:

A pointer constant (also called a "constant pointer") is a pointer that cannot change the memory address it points to after initialization, but you can modify the value stored at that address.

```
data_type *const pointer_name;
int a = 10;
int *const p = &a;

*p = 20;  // OK: Can modify value pointed to by 'p'

// p = &b;  // Error: Cannot change the address 'p' is pointing to
```

In this example:

- p is a constant pointer. It always points to the same memory address (in this case, the address of a), so we cannot assign p to point to another variable.
- However, we can change the value at the memory address p points to (i.e., we can modify the value of a through p).

3. Constant Pointer to Constant:

It is also possible to have a **constant pointer to a constant**. This means that neither the address the pointer holds nor the value at that address can be changed.

```
const data_type *const pointer_name;

int a = 10;
const int *const p = &a;

// *p = 20;  // Error: Cannot modify value pointed to by 'p'
// p = &b;   // Error: Cannot change the address 'p' is pointing to
```

In this example:

- p is a constant pointer to a constant integer. The value at the address p points to cannot be changed, and the pointer itself cannot point to any other address.

Summary:

- **Pointer Variable**: Stores the memory address of a variable.
- **Constant Pointer (Pointer to a Constant)**: Points to a constant value. The pointer can point to different addresses, but the value at the address cannot be changed via the pointer.
- **Pointer Constant (Constant Pointer)**: The pointer cannot point to any other address after initialization, but the value at the address can be changed.
- **Constant Pointer to Constant**: The pointer cannot change the address it points to, and its value cannot be modified.

Unit 3. Arrays

5. **Concept of Arrays**
    - An **array** is a collection of elements of the same type stored at contiguous memory locations. Each element is accessed using an index.
6. **Single Dimensional Arrays**
    - An array with a single index (e.g., int arr[5];).
7. **Two Dimensional Arrays**
    - A 2D array is like a matrix, with rows and columns (e.g., int arr[3][4];).
8. **Storage Strategies for Multidimensional Arrays**
    - Multidimensional arrays can be stored in **row-major order** (rows first) or **column-major order** (columns first).

1. Concept of Arrays:

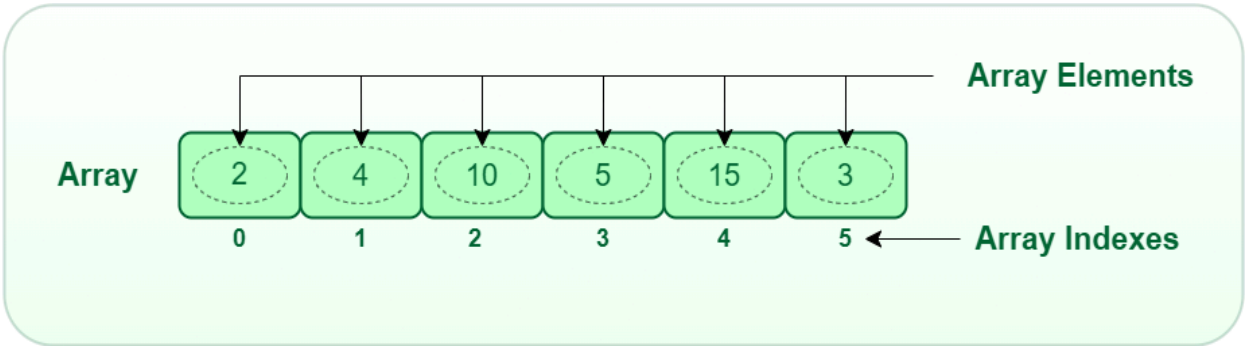
An **array** is a data structure that stores a collection of elements of the **same data type** in a **contiguous memory block**. The array allows for indexed access to each element, which makes retrieval and manipulation of data efficient. Arrays are fundamental data structures used to organize data in a structured format.

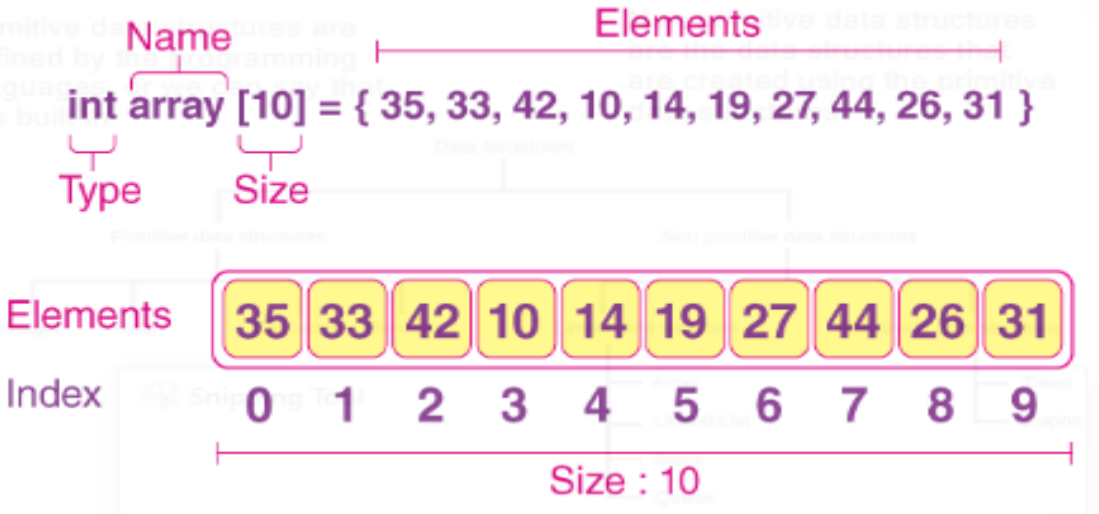
Key Characteristics:

- **Homogeneous**: All elements of the array are of the same data type.
- **Contiguous Memory**: Elements are stored in adjacent memory locations.
- **Fixed Size**: The size of the array is determined at the time of declaration and cannot be changed during runtime.
- **Indexing**: Each element in the array is associated with an index, starting from 0 for the first element.

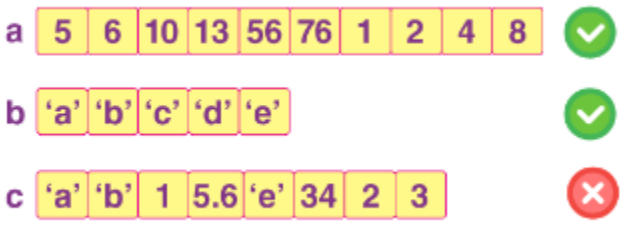
A. **Single Dimensional Arrays**

An array is a collection of items of the same data type stored at contiguous memory locations.





Important: An array can store only the same type of data items. From the below example, you can see how it works:



Program of an int array

```c
#include <stdio.h>
int main()
 {
int arr[] = {10, 20, 30, 40, 50};      // display array
int i = 0;
 while(arr[i]){
printf("%d ", arr[i]);
i++;}
return 0;
 }

//if u dont know how many elements r there in an array
```

Different ways of initialization of an array.

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {

    int arr[]= {10, 20, 30, 40, 50};
    int arr1[5]= {10, 20, 30, 40, 50};

    printf("The Array elements are:\n");
  for(int i=0; i<5; i++) {
     printf("%d ", arr[i]);
  }

    return 0;
}
```

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {

   // int arr[]= {10, 20, 30, 40, 50};
    int arr1[8]= {10, 20, 30, 40, 50};

    printf("The Array elements are:\n");
  for(int i=0; i<8; i++) {
     printf("%d ", arr1[i]);
  }

    return 0;
}
```

```c
// Online C compiler to run C program online
#include <stdio.h>

int main() {

   // int arr[]= {10, 20, 30, 40, 50};
    int arr1[5]= {10, 20, 30, 40, 50};

    printf("The Array elements are:\n");
  for(int i=0; i<7; i++) {
     printf("%d ", arr1[i]);
  }

    return 0;
}
```

Memory Allocation of 1-D Array in C

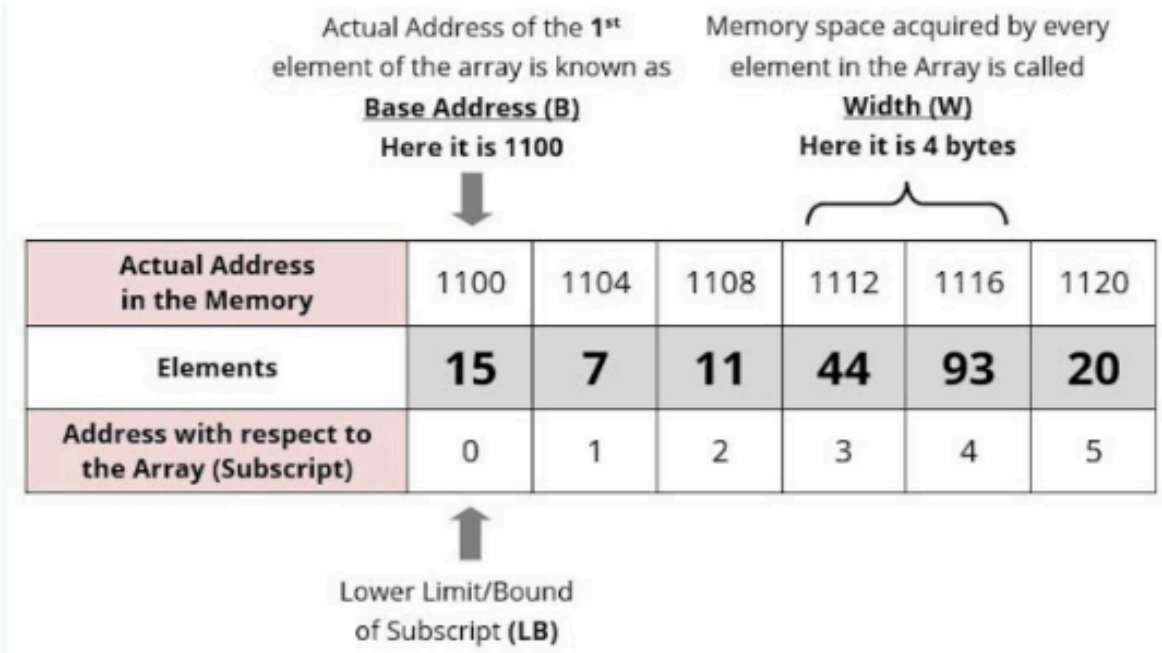
In C, a **1-D array** is a collection of elements of the same data type stored in contiguous memory locations. Each element can be accessed via an index from 0 to n-1, where n is the number of elements in the array. The memory allocation for an array depends on the array elements' data type and the array's number of elements.

Static Memory Allocation

Static memory allocation means the memory for the array is allocated at **compile-time**. The array size is fixed and cannot be altered during program execution. The stack allocates memory for static arrays declared inside a function.

Finding the Address of Elements in a 1-D Array

In a **1-D array**, the elements are stored in contiguous memory locations. The address of any element can be calculated using a **formula** that takes into account the **base address** of the array and the **size of the data type**.



Array of an element of an array say "A[I]" is calculated using the following formula:

$$\text{Address of A [I]} = B + W * (I - LB)$$

Where,
B = Base address
W = Storage Size of one element stored in the array (in byte)
I = Subscript of element whose address is to be found
LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

```
#include <stdio.h>

int main() {
   int arr[5];  // Declares an array of size 5 (static allocation)

   // Memory allocation in the stack, for example:
   // if sizeof(int) is 4 bytes, the memory layout looks like:
   // arr[0] -> address 1000 (4 bytes)
   // arr[1] -> address 1004 (4 bytes)
   // arr[2] -> address 1008 (4 bytes)
   // arr[3] -> address 1012 (4 bytes)
   // arr[4] -> address 1016 (4 bytes)

   for (int i = 0; i < 5; i++) {
      arr[i] = i + 1;  // Assigning values to the array
   }

   // Printing the elements and their addresses
   for (int i = 0; i < 5; i++) {
      printf("arr[%d] = %d, Address = %p\n", i, arr[i], (void*)&arr[i]);
   }

   return 0;
}
```

Output

```
arr[0] = 1, Address = 0x7ffc1c1b60c0
arr[1] = 2, Address = 0x7ffc1c1b60c4
arr[2] = 3, Address = 0x7ffc1c1b60c8
arr[3] = 4, Address = 0x7ffc1c1b60cc
arr[4] = 5, Address = 0x7ffc1c1b60d0
```

Example 1: Array with Custom Negative Indexing (e.g., -10 to 4)

Let's assume an array with a valid index range of -10 to 4. Normally, C arrays are indexed from 0, but we can adjust the calculation of the addresses to simulate this custom index range.

Steps:

1. **Simulate an array with indices from -10 to 4**.
2. Use pointer arithmetic to calculate the address of each element.

Formula for Address with Custom Indexing:

If you want to find the address of an element with index i in a custom-indexed array:

Address of arr[i] = Base address of arr[0] + ((i - start_index) * size of element)

```
#include <stdio.h>

int main() {
   int arr[15]; // Array of 15 elements (from index -10 to 4)

   // Simulating base address for custom indexing
   int *base = &arr[10]; // We treat arr[10] as arr[0] in a normal array

   // Calculating and printing the addresses for the range [-10, 4]
   for (int i = -10; i <= 4; i++) {
      printf("Address of arr[%d] = %p\n", i, (void*)(base + (i + 10)));
   }

   return 0;
}
```

Explanation:

- The array arr[15] has space for 15 integers.
- We simulate the array's base as arr[10], representing arr[0] in a normal array.
- We adjust the index by +10 to get the correct addresses when using negative indices.
    - For example, the address of arr[-10] is calculated as base + (-10 + 10), equivalent to base + 0.

Example 2: Array with Arbitrary Positive Indexing (e.g., 100 to 200)

Now, let's assume an array indexed from 100 to 200. We use the same principle, simulating the array as if it starts from index 100.

The formula for Address Calculation:

Address of arr[i] = Base address of arr[100] + ((i - 100) * size of element)

```
#include <stdio.h>

int main() {
   int arr[101]; // Array of 101 elements (from index 100 to 200)

   // Simulating base address for custom indexing
   int *base = &arr[0]; // Base address for normal indexing

   // Calculating and printing the addresses for the range [100, 200]
   for (int i = 100; i <= 200; i++) {
      printf("Address of arr[%d] = %p\n", i, (void*)(base + (i - 100)));
   }

   return 0;
}
```

Operation on an Array

Arrays are a fundamental data structure that stores multiple values of the same type in contiguous memory locations. They allow efficient random access to elements and provide operations such as insertion, deletion, searching, traversing, updating, and more.

Here are common operations that can be performed on an array:

1. **Traversal**
2. **Insertion**
3. **Deletion**
4. **Searching**
5. **Updating**
6. **Sorting**

1. Traversal

Traversal means accessing each element of an array once to perform some operations such as printing or processing data.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    // Traverse and print each element
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
Output
1 2 3 4 5
```

2. Insertion

Insertion adds a new element at a specific position in the array. However, since arrays have fixed sizes, if the array is full, you may need to shift elements to make room for the new element.

```
#include <stdio.h>

int main() {
    int arr[6] = {1, 2, 3, 4, 5}; // Array has space for 6 elements
    int n = 5; // Current size of the array
    int element = 10;
    int position = 2; // Insert at index 2 (3rd position)

    // Shift elements to the right from the position to make space
    for (int i = n; i > position; i--) {
        arr[i] = arr[i - 1];
    }

    arr[position] = element; // Insert the element
    n++; // Increase the size of the array

    // Print the array after insertion
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }


    return 0;
}
Output

1 2 10 3 4 5
```

3. Deletion

Deletion involves removing an element from the array at a given index and then shifting elements to fill the gap.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int n = 5; // Current size of the array
    int position = 2; // Remove the element at index 2

    // Shift elements to the left to fill the gap
    for (int i = position; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }

    n--; // Decrease the size of the array

    // Print the array after deletion
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output
1 2 4 5

4. Searching

Searching for an element in an array can be done in two ways:

- **Linear Search:** Traverse through the array one by one.
- **Binary Search:** Works on sorted arrays. The array is repeatedly divided into half until the element is found or the search space becomes empty.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int n = 5;
    int key = 3; // Element to be searched
    int found = 0;

    // Linear search
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Element %d found at index %d\n", key, i);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Element %d not found\n", key);
    }

    return 0;
}
```

```
#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x) {
   if (r >= l) {
      int mid = l + (r - l) / 2;

      if (arr[mid] == x)
         return mid;

      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      return binarySearch(arr, mid + 1, r, x);
   }

   return -1;
}

int main(void) {
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr) / sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n - 1, x);
   if (result == -1)
      printf("Element is not present in array\n");
   else
      printf("Element is present at index %d\n", result);
   return 0;
}
```

5. Updating

Updating means modifying the value of an existing element in the array at a particular index.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int position = 2; // Update the element at index 2
    int newValue = 10;

    arr[position] = newValue; // Update the value

    // Print the array after updating
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

Output
1 2 10 4 5
```

6. Sorting

Sorting refers to rearranging the elements in an array in increasing or decreasing order. Common algorithms include Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort.

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[5] = {5, 1, 4, 2, 8};
    int n = 5;

    bubbleSort(arr, n); // Sorting the array

    // Print the sorted array
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Summary of Time Complexities for Array Operations

Operation	Best Case	Worst Case
Traversal	O(n)	O(n)
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Searching	O(1) (Best, if found early)	O(n) (Linear Search)
Sorting	O(n log n) (Merge/Quick)	O(n^2) (Bubble)

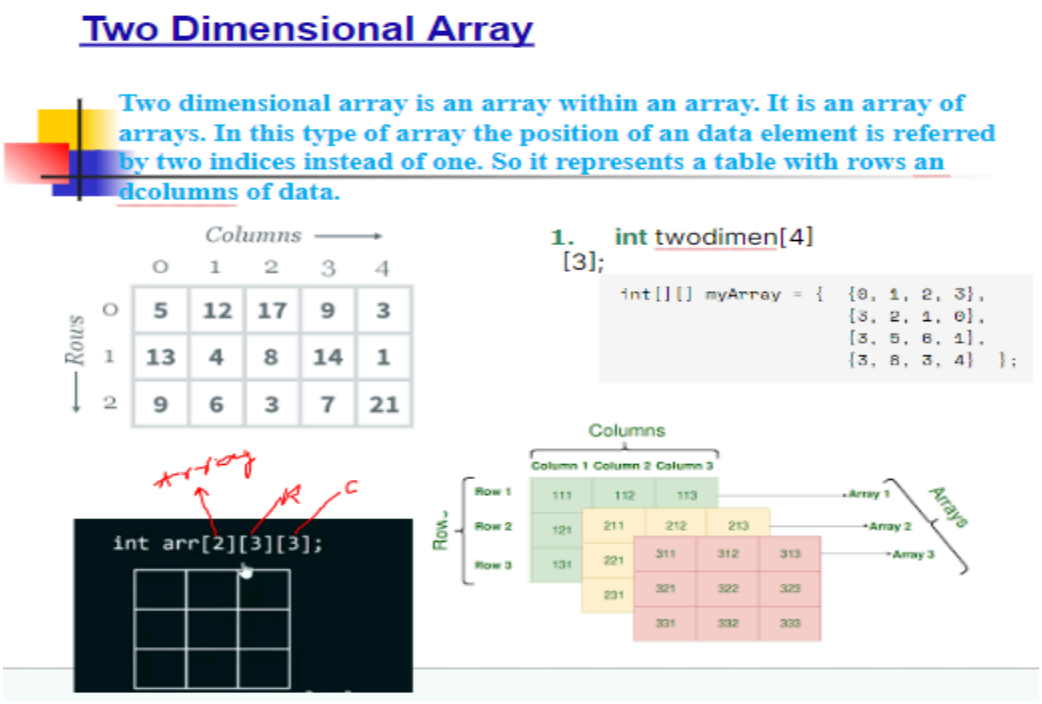
2D Arrays: Concept, Operations, and Examples

A **2D array** (also known as a matrix) is a collection of data elements arranged in rows and columns. It can be visualized as a table where each element is identified by two indices: one for the row and one for the column.

Basic Concept of 2D Arrays

- A **2D array** is an array of arrays. It is used to represent a matrix-like structure where data is organized in rows and columns.
- A 2D array is declared by specifying two dimensions: the number of rows and the number of columns.

Declaration of a 2D Array in C



Memory Representation of 2D Arrays

In memory, a 2D array is stored either in:

1. **Row-Major Order**: Elements of the rows are stored in contiguous memory locations.
2. **Column-Major Order**: Elements of the columns are stored in contiguous memory locations.

- **Row-Major Order**: This is the default storage strategy in most programming languages, including C. The elements of the first row are stored first, followed by the elements of the second row, and so on.
  Memory address of an element in Row-Major Order:

Address(arr[i][j]) = Base_address + [(i * total_columns) + j] * size_of_element

Column-Major Order: In this strategy, the elements of the first column are stored first, followed by the elements of the second column, and so on.

Address(arr[i][j]) = Base_address + [(j * total_rows) + i] * size_of_element

Example: Accessing Elements in a 2D Array

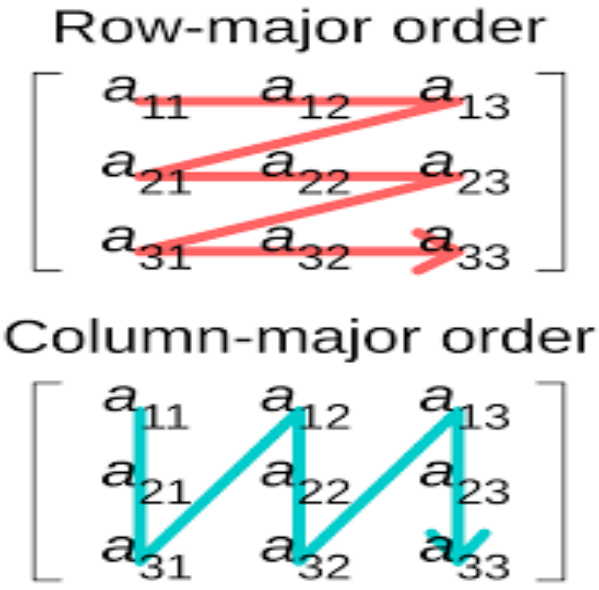
Given the following 2D array:

```
int arr[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
The element arr[0][0] is 1.
The element arr[1][2] is 7.
The element arr[2][3] is 12.
```

Operations on a 2D Array

Traversing a 2D Array

To traverse a 2D array, we must access each element row by row or column by column.

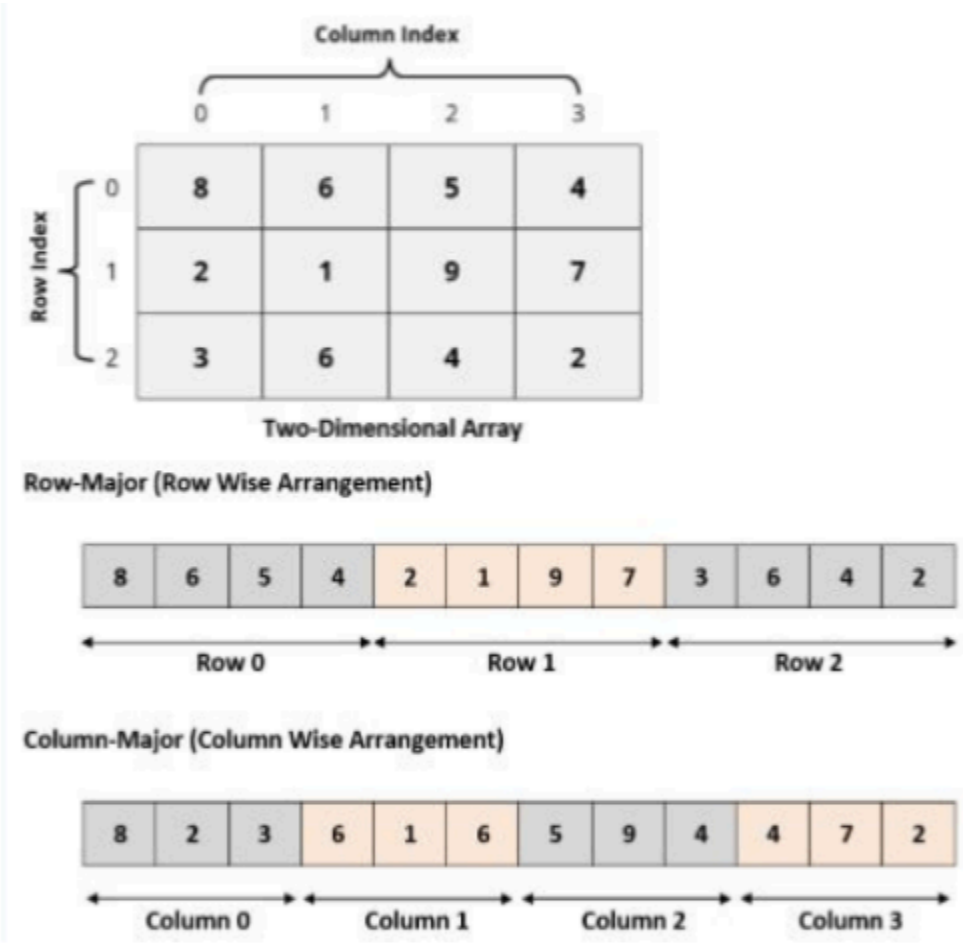


	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Concept:

Row Major Order (RMO)

ELEMENTS	ADDRESS	
A[1,1]	200	Row 1
A[1,2]	201	
A[1,3]	202	
A[1,4]	203	
A[2,1]	204	Row 2
A[2,2]	205	
A[2,3]	206	
A[2,4]	207	
A[3,1]	208	Row 3
A[3,2]	209	
A[3,3]	210	
A[3,4]	211	



Pseudo Code for Traversing (Row-Major Order):

```
Procedure Traverse2DArray(arr, rows, cols):
    For i = 0 to rows - 1:
        For j = 0 to cols - 1:
            Print arr[i][j]

In c.
#include <stdio.h>

int main() {
    int arr[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Insertion in a 2D Array

Inserting an element into a 2D array requires specifying the row and column position where the element is to be inserted.

arr[2][1] = 15; // Inserting 15 into the third row and second column

Searching in a 2D Array

Linear search can be used to find an element in a 2D array by traversing the array and comparing each element with the target.

```
Procedure Search2DArray(arr, rows, cols, key):
   For i = 0 to rows - 1:
      For j = 0 to cols - 1:
         If arr[i][j] == key:
            Print "Element found at", i, j
            Return
   Print "Element not found"

In C
#include <stdio.h>

int main() {
   int arr[3][4] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12}
   };
   int key = 7;
   int found = 0;

   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 4; j++) {
         if (arr[i][j] == key) {
            printf("Element found at (%d, %d)\n", i, j);
            found = 1;
            break;
         }
      }
      if (found) break;
   }

   if (!found) {
      printf("Element not found\n");
   }

   return 0;
}
```

Address Calculation in 2D Arrays

For a **2D array** with base address Base_address, we can calculate the address of an element using **row-major order** with the formula:

Address(arr[i][j]) = Base_address + [(i * total_columns) + j] * size_of_element

Where:

- Base_address is the starting address of the array.
- i is the row index.
- j is the column index.
- total_columns is the number of columns in the array.
- size_of_element is the size of each element (e.g., 4 bytes for an integer).

Example:

Consider a 2D array arr[3][4] of integers starting at address 1000 with size_of_element = 4 bytes.

To find the address of the element at position arr[2][3]:

```
Address(arr[2][3]) = 1000 + [(2 * 4) + 3] * 4
                   = 1000 + (8 + 3) * 4
                   = 1000 + 11 * 4
                   = 1000 + 44
                   = 1044
```

Quiz 1

Consider a 2-d array A[-3.... 7] [6..... 12]. The starting address of array in memory is 1000. Each element occupies 4 memory locations.

1. What is the address of element A[0][9] in row major order?
2. What is the address of element A[5][7] in row major order?

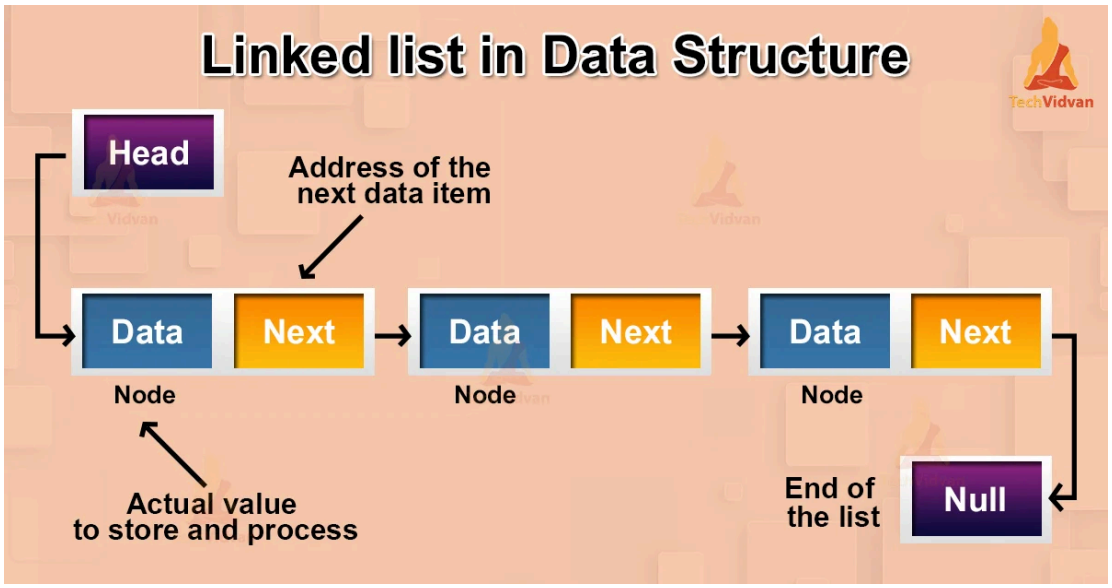
$LB_i = -3$ | $m = 7-(-3)+1 = 11$ | Base = 1000
$LB_j = 6$ | $n = 12-6+1 = 7$ | $w = 4$

1. $Loc(A[0][9]) = 1000 + 4 * [[0-(-3)]\,7 + (9-6)] = 1096$
   $i=0, j=9$

2. $Loc(A[5][7]) = 1000 + 4 * [(5-(-3)) * 7 + (7-6)] = 1228$
   $i=5, j=7$

Unit 4: Linked Lists

5. **Introduction to Linked List**
    - A **linked list** is a dynamic data structure where each element (node) points to the next. It can grow or shrink dynamically.
6. **Doubly Linked List**
    - In a **doubly linked list**, each node contains pointers to the next and previous nodes, allowing traversal in both directions.
7. **Memory Representation**
    - Linked lists are stored in **non-contiguous** memory locations. Each node contains data and a pointer to the next node.
8. **Operations on Linked Lists**
    - **Traversal**: Visit each element on the list.
    - **Searching**: Finding an element.
    - **Insertion**: Adding a new element.
    - **Deletion**: Removing an existing element.



A linked list is a linear collection of data elements called nodes in which linear representation is given by links from one node to the next node. Similar to the array, it is a linear collection of data elements of the same type. Different from an array, data elements of a linked list are generally not lined in consecutive memory space; instead, they are dispersed in various locations.

- **Definition:** A linear data structure where elements are connected using pointers, allowing for dynamic growth and shrinking.

- **Nodes:** Basic units of a linked list, typically containing data and a pointer to the next node.
- **Singly Linked Lists:** Each node has a pointer to the next node.
- **Doubly Linked Lists:** Each node has pointers to the previous and next nodes.

Singly Linked Lists

- **Advantages:**
  - Simple implementation.
  - Efficient insertion and deletion at the beginning or end.
- **Disadvantages:**
  - Inefficient traversal in the reverse direction.
  - Can efficiently access a specific element by traversing from the beginning.

Doubly Linked Lists

- **Advantages:**
  - Efficient traversal in both directions.
  - Efficient insertion and deletion at any position.
- **Disadvantages:**
  - More complex implementation.
  - Slightly more memory overhead due to the additional pointer.

Linked List Operations

- **Initialization:** Creating an empty linked list.
- **Insertion:** Adding a new node at the beginning, end, or a specific position.
- **Deletion:** Removing a node from the beginning, end, or a specific position.
- **Traversal:** Iterating through the linked list to access or modify elements.
- **Searching:** Finding a specific element in the linked list.

Linked List Applications

- **Stacks:** Implementing stacks using linked lists.
- **Queues:** Implementing queues using linked lists.
- **Graphs:** Representing graphs using adjacency lists.
- **Polynomial representation:** Representing polynomials using linked lists.
- **Music playlists:** Storing and managing songs in a linked list.

```
struct Node {
    int data;
    Node* next;
};

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    void insertAtBeginning(int data) {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->next = head;
        head = newNode;
    }

    // ... other operations like insertAtEnd, delete, search, etc.
};
```

Example (Doubly Linked List-insertion)

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// Function to insert at the beginning
struct Node* insertAtBeginning(struct Node* head, int newData) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = head;
    newNode->prev = NULL;

    if (head != NULL) {
        head->prev = newNode;
    }

    return newNode;
}

int main() {
    struct Node* head = NULL;

    // Insert elements
    head = insertAtBeginning(head, 40);
    head = insertAtBeginning(head, 50);

    return 0;
}
```

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Unit 5: Stacks

5. **Introduction to Stacks**
    - A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. The last element inserted is the first to be removed.
6. **Stack Representation**
    - Stacks can be represented using arrays or linked lists.
7. **Stack Operations**
    - **Push**: Insert an element.
    - **Pop**: Remove an element.
    - **Top/Peek**: View the top element without removing it.
8. **Uses of Stacks**
    - Stacks are used in function calls, expression evaluation, parsing, and more.

Introduction to Stacks

A **stack** is a linear data structure that operates on the **Last In, First Out (LIFO)** principle. This means the last element added to the stack is the first to be removed. Stacks are commonly compared to a pile of plates, where you can only remove the topmost plate and add a new one to the top.

Characteristics of Stacks:

- **LIFO** principle: The element added last is removed first.
- Operations are performed only on one end of the stack, called the **top**.

Operations on stack

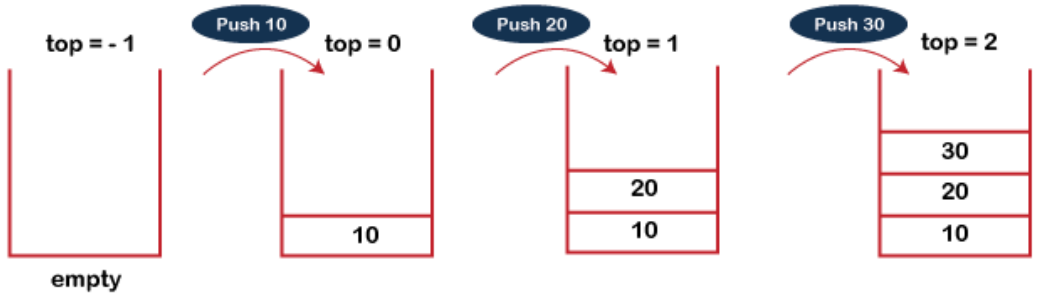
- **push():** When we insert an element in a stack, the operation is known as push. If the stack is full, then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty, no element exists; this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full.
- **peek():** This operation returns the value at the **top of the stack**, allowing you to see what is there, but the value is not removed. This operation is typically used when inspecting the most recent item pushed to the stack.
- **count():** It returns the total number of elements available in a stack.

- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

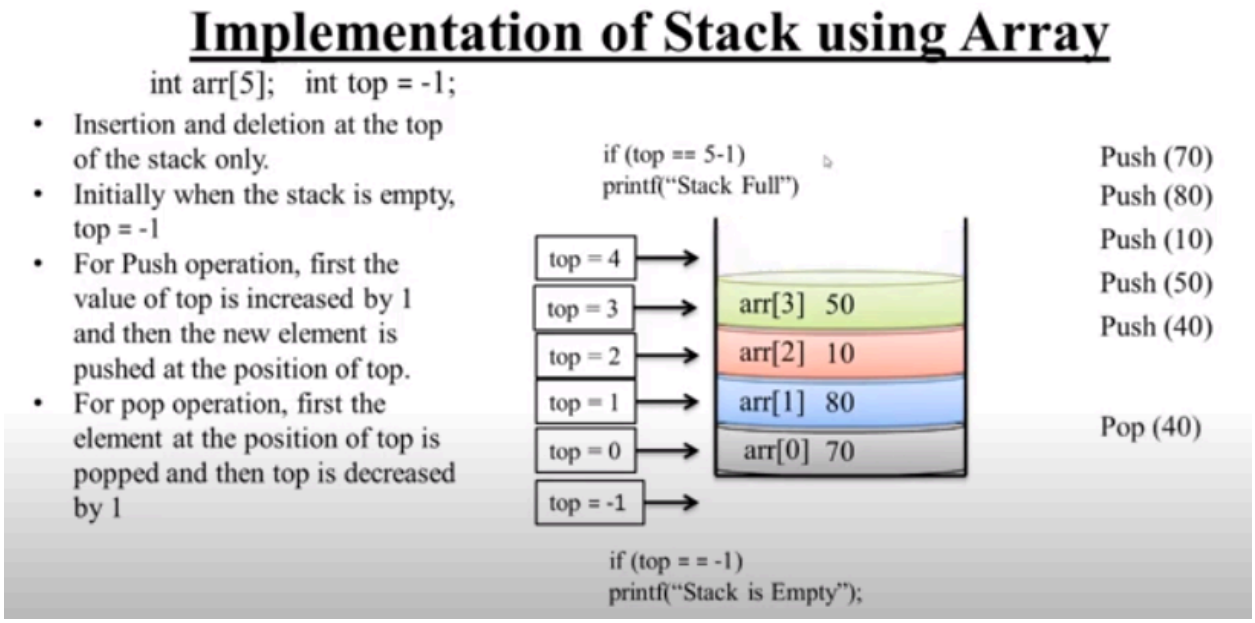
Working of Stack

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



Array Implementation of stack



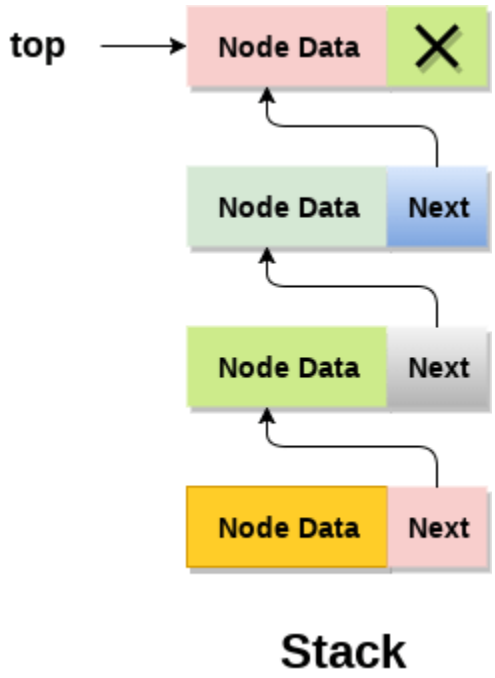
```
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;

class Stack {
public:
    int arr[MAX_SIZE];
    int top;

    Stack() {
        top = -1;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == MAX_SIZE - 1;
    }

    void push(int x) {
        if (isFull()) {
            cout << "Stack Overflow\n";
            return;
        }
        arr[++top] = x;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow\n";
            return -1;
        }
        return arr[top--];
    }

    int peek() {
        if (isEmpty()) {
            cout << "Stack Underflow\n";
            return -1;
        }
        return arr[top];
    }
```

```
};

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    cout << stack.pop() << endl;
    cout << stack.peek() << endl;

    return 0;
}
```

Implementation using Linked List

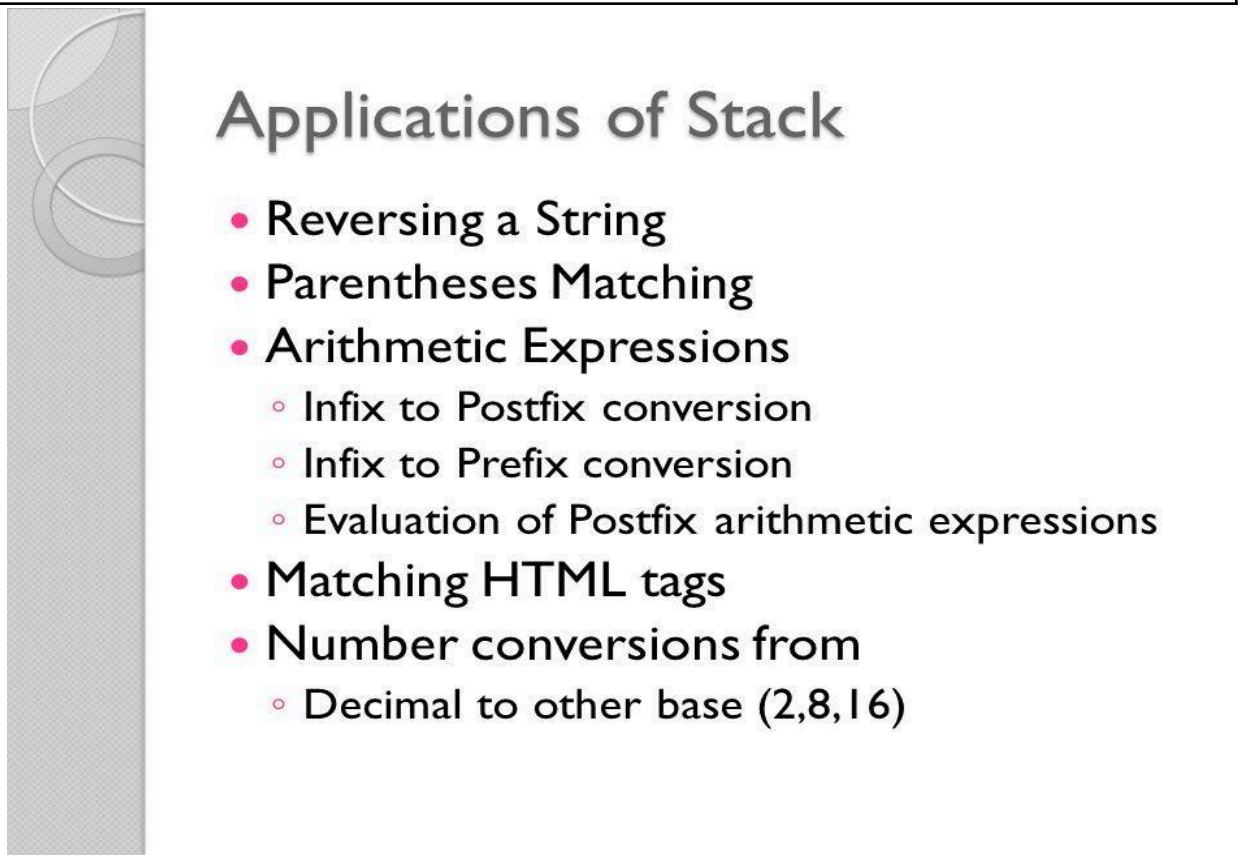
Push function

We should create the linked list in reverse order so that the head node always points to the last inserted data.



```
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* next;
};

class Stack {
public:
    Node* top;

    Stack() {
        top = nullptr;
    }

    bool isEmpty() {
        return top == nullptr;
    }

    void push(int x) {
        Node* newNode = new Node;
        newNode->data = x;
        newNode->next = top;
        top = newNode;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow\n";
            return -1;
        }
        int x = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return x;
    }

    int peek() {
        if (isEmpty()) {
            cout << "Stack Underflow\n";
            return -1;
        }
        return top->data;
```

```
    }
};

// ... rest of the code remains the same
```



Expression Evaluation of Infix Expression Using Stack

Infix expressions are the standard way of writing mathematical expressions where operators are placed between operands (e.g., A + B). The challenge with infix expressions is that operators have precedence and associativity rules, making evaluation more complex than postfix expressions. To evaluate an infix expression using stacks, we convert it to postfix or evaluate it directly using two stacks: one for operands and another for operators.

Steps to Evaluate Infix Expression Using Stack:

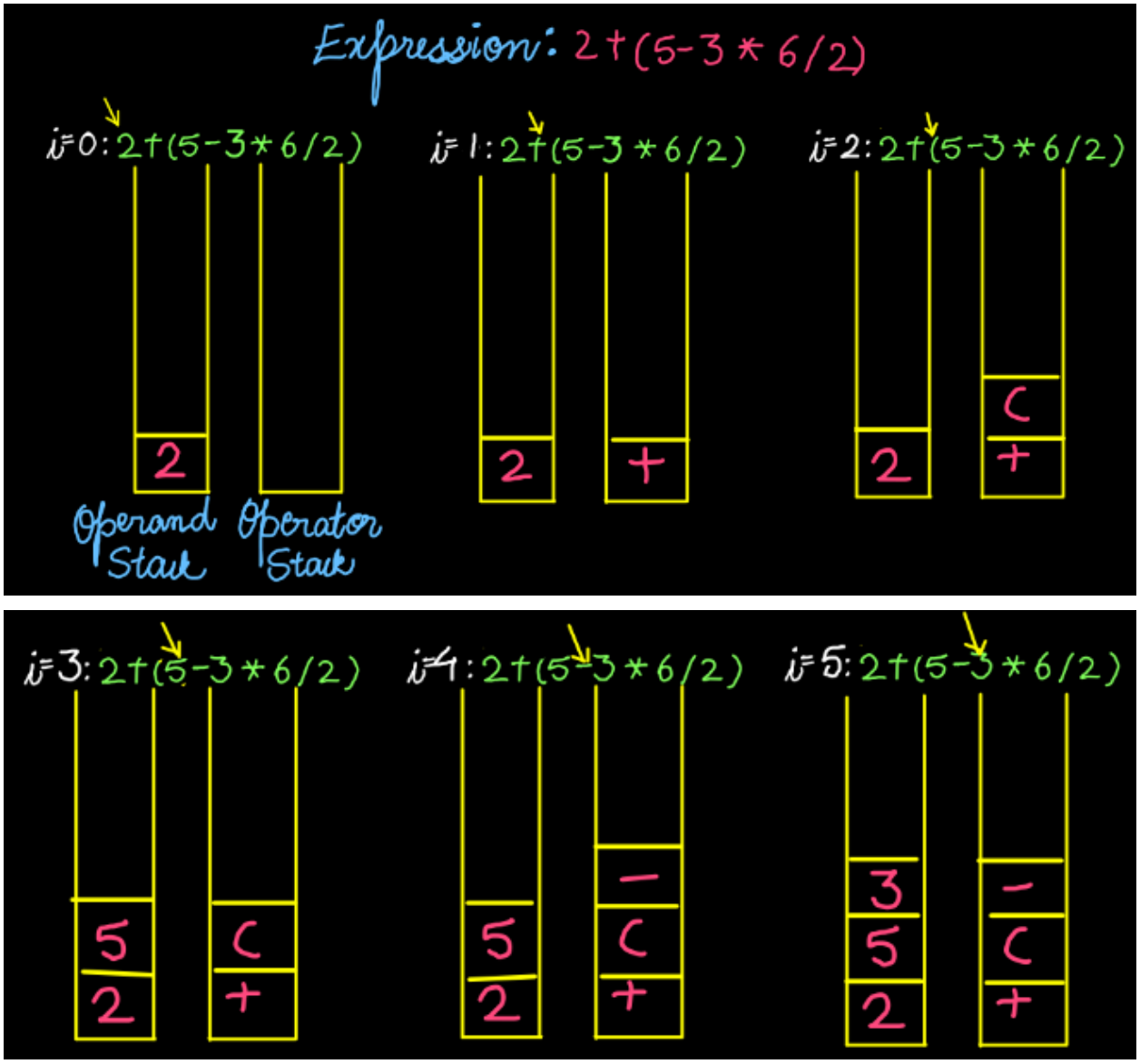
1. **Initialize two stacks**:
    - **Operand Stack**: To store numbers or variables (operands).
    - **Operator Stack**: Store operators like +, -, *, /.
2. **Scan the expression from left to right**:

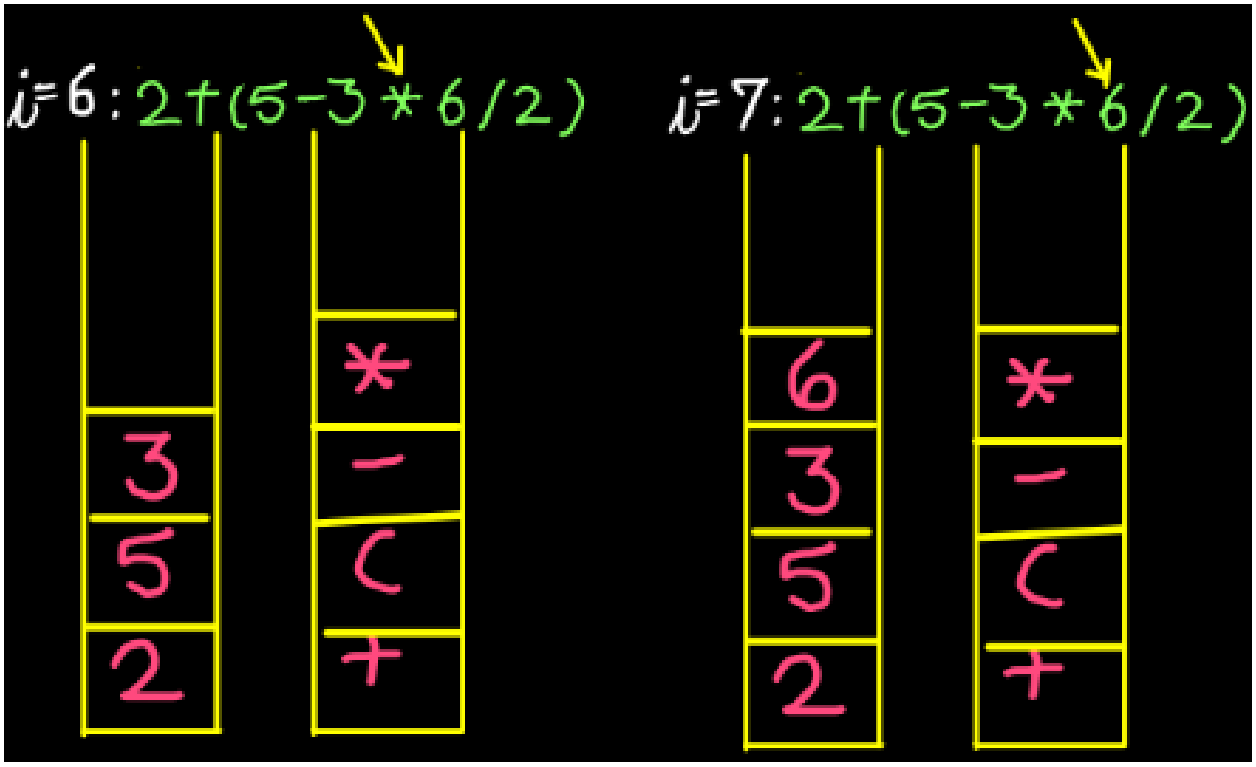
- If you encounter an **operand** (a number or variable), push it onto the operand stack.
- If you encounter an **opening parenthesis (()**, push it onto the operator stack.
- If you encounter a **closing parenthesis ())**:
  - Pop from the operator stack and apply the operator to the top two operands in the operand stack until you find the opening parenthesis.
  - Pop the opening parenthesis from the operator stack.
- If you encounter an **operator**:
  - While the operator stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator on top of the operator stack, apply the operator on top of the stack to the top two operands in the operand stack.
  - Push the current operator onto the operator stack.

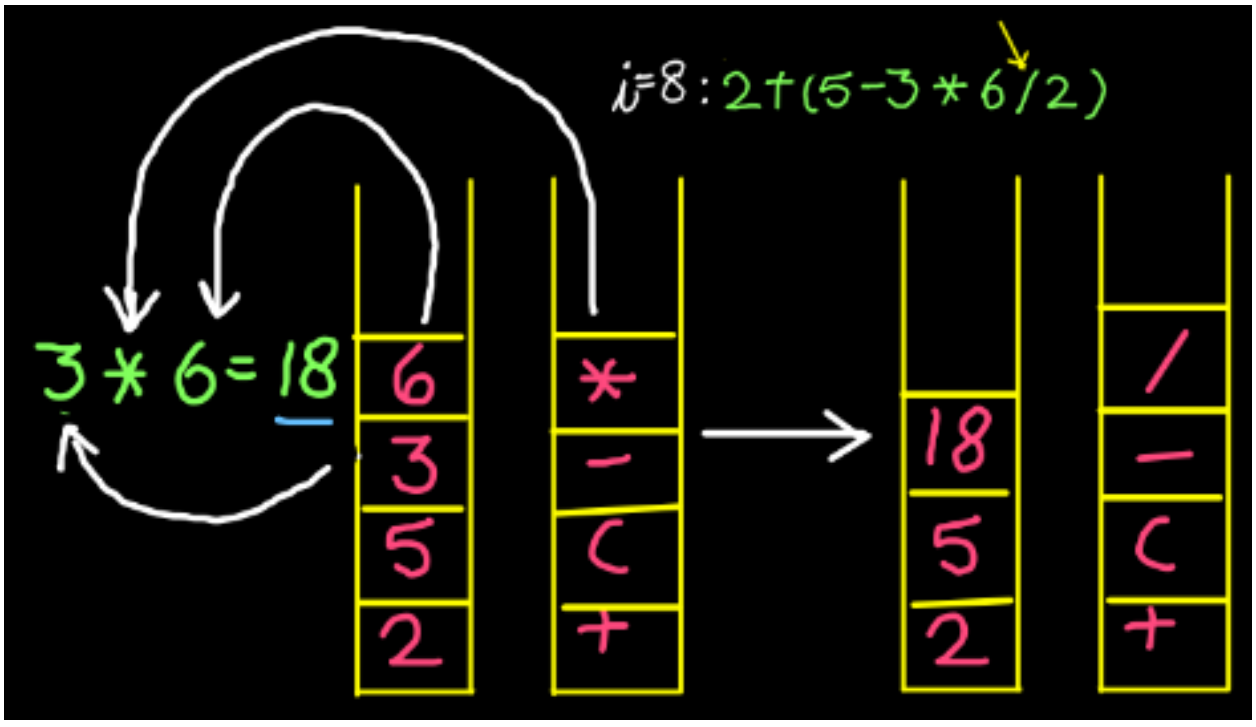
3. **When the expression is completely scanned**, apply any operators in the operator stack to the operands in the operand stack.
4. **Result**: The expression will result in the operand stack's final value.

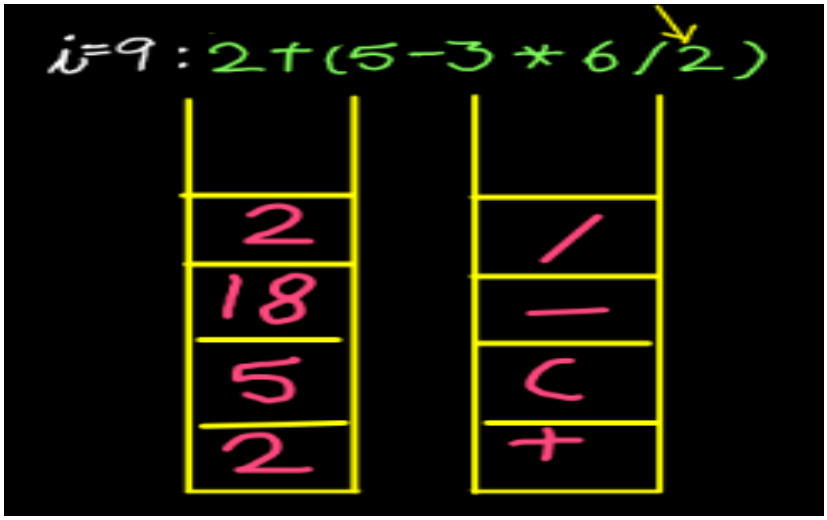
```
Procedure EvaluateInfix(expression):
   Create empty Operand Stack
   Create empty Operator Stack

   For each character in the expression:
      If the character is a number or operand:
         Push it onto the Operand Stack
      If the character is an opening parenthesis '(':
         Push it onto the Operator Stack
      If the character is a closing parenthesis ')':
         While the top of the Operator Stack is not '(':
            Operator = Pop from Operator Stack
            Operand2 = Pop from Operand Stack
            Operand1 = Pop from Operand Stack
            Result = Apply Operator to Operand1 and Operand2
            Push Result onto Operand Stack
         Pop '(' from the Operator Stack
      If the character is an operator:
         While Operator Stack is not empty and precedence of character <= precedence of top
of Operator Stack:
            Operator = Pop from Operator Stack
            Operand2 = Pop from Operand Stack
            Operand1 = Pop from Operand Stack
            Result = Apply Operator to Operand1 and Operand2
            Push Result onto Operand Stack
         Push the current character (operator) onto the Operator Stack

   While Operator Stack is not empty:
      Operator = Pop from Operator Stack
      Operand2 = Pop from Operand Stack
      Operand1 = Pop from Operand Stack
      Result = Apply Operator to Operand1 and Operand2
      Push Result onto Operand Stack

   Return the top of Operand Stack (final result)
```

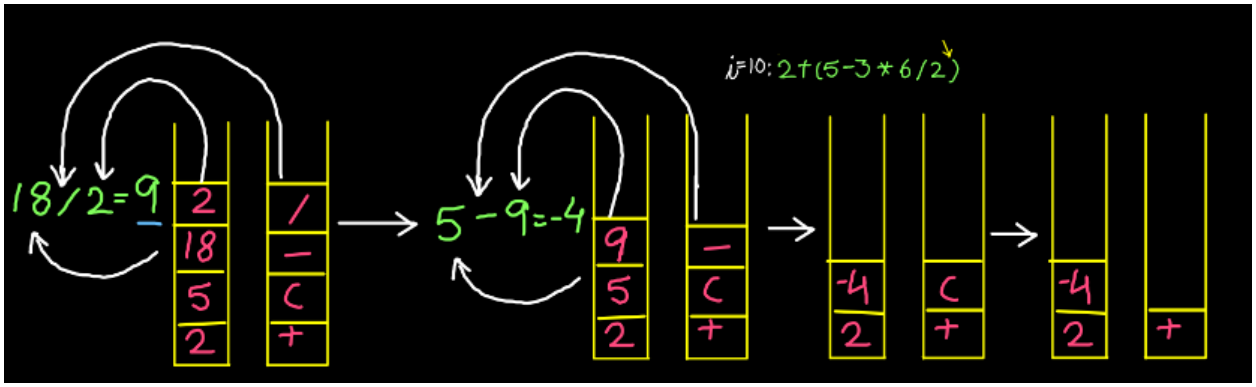
Example

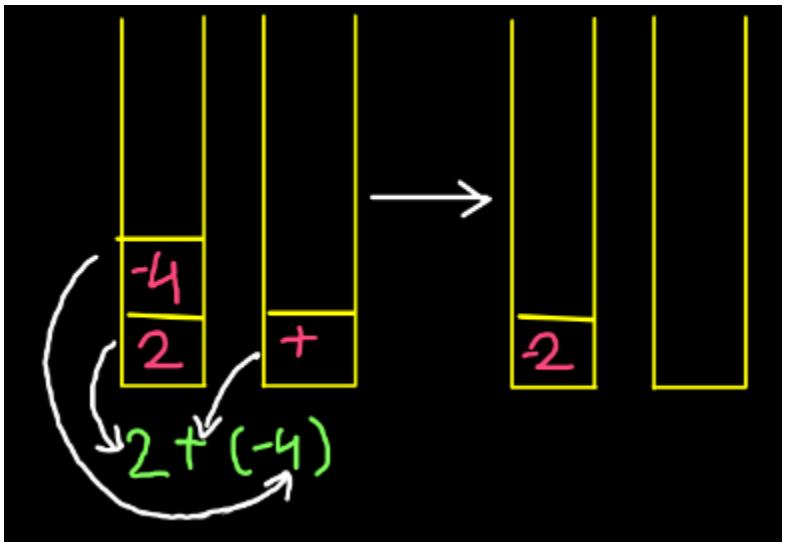

Expression: 2↑(5-3*6/2)
i=0: 2↑(5-3*6/2)
i=1: 2↑(5-3*6/2)
i=2: 2↑(5-3*6/2)
2
(
+
Operand Stack
Operator Stack
i=3: 2↑(5-3*6/2)
i=4: 2↑(5-3*6/2)
i=5: 2↑(5-3*6/2)
5
2
3
-


i=6 : 2+(5-3*6/2)
3
5
2
*
-
(
+
i=7 : 2+(5-3*6/2)
6
3
5
2
*
-
(
+


i=8 : 2+(5-3*6/2)
3*6=18
6
3
5
2
*
-
(
+
18
5
2
/
-
(
+


i=9 : 2+(5-3*6/2)
2
18
5
2
/
-
C
+


i=10: 2+(5-3*6/2)
18/2=9
5-9=-4
2
18
5
2
/
-
C
+
9
5
2
-
C
+
-4
2
C
+
-4
2
+

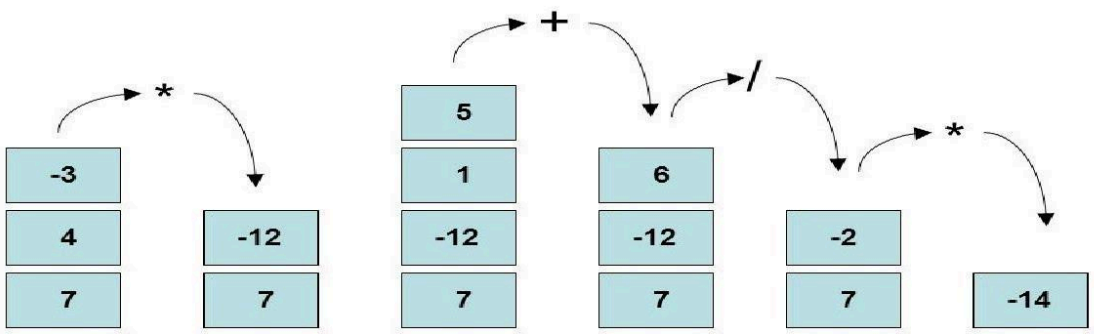

-4
2
+
-2
2+(-4)

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

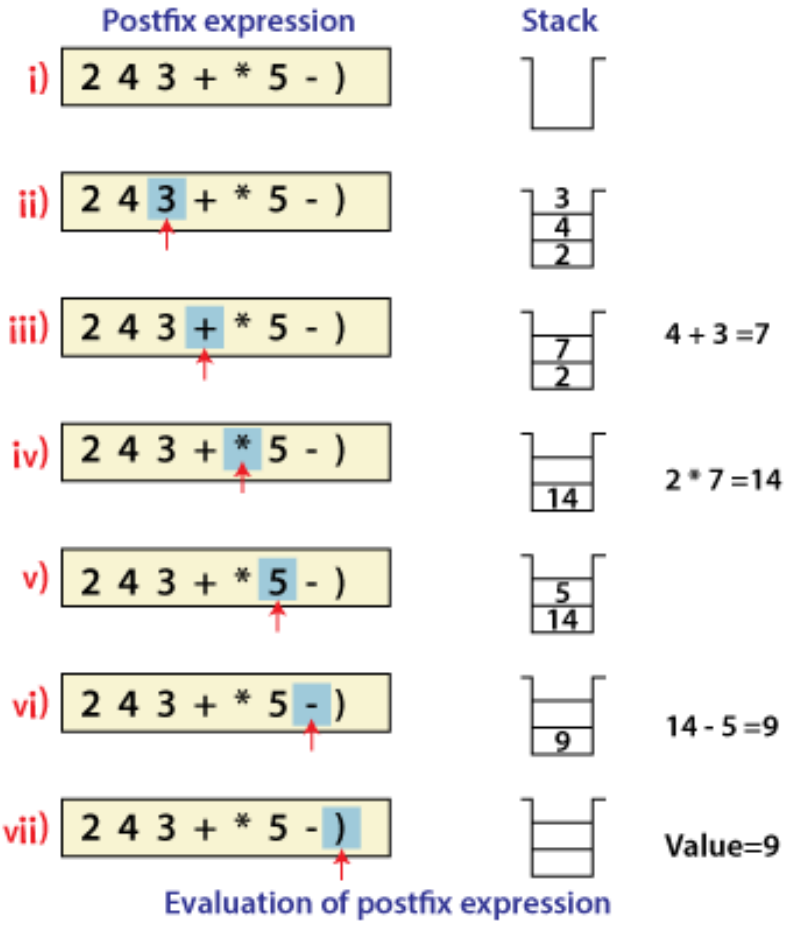
Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	? × (4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *



7



Example 1: Evaluate the following postfix expression: 6 2 3 + – 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

CONVERT INTO POST EXPRESSION

Input Character	Operations on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till Empty		**AB*CD+-E+**

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Algorithm infix to prefix

Step 1. Reverse the input string
Step 2. Examine the next element in the input
Step 3. If it is operand add it to the output string
Step 4. If it is closing parenthesis push it on stack
Step 5. If it is an operator then
  a. If stack is empty, push operation on stack
  b. If the top of stack is ")" push operator on stack
  c. If it has same or higher priority then the top of stack, push it
  d. Else pop the operator & add it to output string, repeat 5
Step 6. If it is "(" pop operator and add them to string s until a ")" is encountered. POP and discard ")"
Step 7. If there is more input go to step 2
Step 8. If there is no more input, unstack the remaining operators & add them
Step 9. Reverse the output string

Infix to Prefix Conversion

Infix Expression: (P + (Q * R) / (S - T))

Note: - Read the infix string in revers

Symbol Scanned	Stack	Output
)	)	-
)	))	-
T	))	T
-	))-	T
S	))-	ST
(	)	-ST
/	)/	-ST
)	)/)	-ST
R	)/)	R-ST
*	)/)*	R-ST
Q	)/)*	QR-ST
(	)/	*QR-ST
+	)+	/*QR-ST
P	)+	P/*QR-ST
(	Empty	+P/*QR-ST

Prefix Expression: +P/*QR-ST

Convert the following postfix expression A B C * D E F ^ / G * – H * + into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) – ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) – ((D/(E^F))*G)) H	Push H
*	A (((B*C) – ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

Application of Stack for recursion evaluation

Application of Stack for Recursion Evaluation

Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem. Each recursive call adds a new layer to the call stack, storing information about the function's execution state, including local variables and the return address. When the base case is reached, the function returns and the stack unwinds to the initial call.

How Stack Works in Recursion

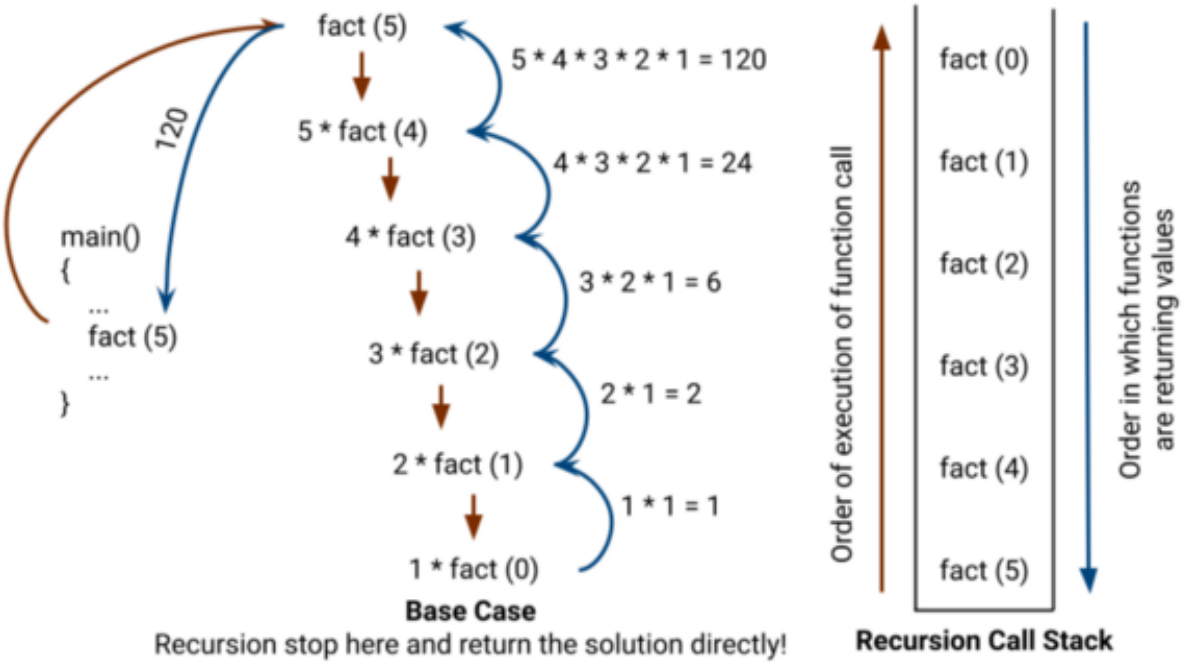
1. Function Call: When a recursive function is called, a new stack frame is created, which contains:
    - The function's parameters.
    - Local variables.

- The address of the next instruction to be executed after the function returns.

2. Stack Growth: Each recursive call pushes a new frame onto the call stack until the base case is reached.
3. Base Case: The function returns a value once the base case is met. The stack unwinds as each frame is popped off, returning control to the previous frame.
4. Return Value: The final return value is passed through the stack frames until it reaches the original caller.

Example of Recursion: Factorial Calculation

Factorial Function: The factorial of a non-negative integer nnn is the product of all positive integers up to nnn. It is defined recursively as:

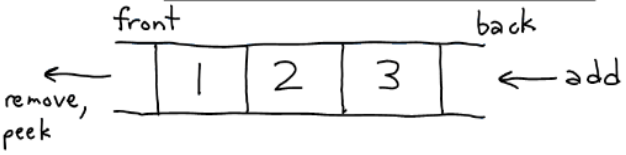


```
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case
    if (n == 0) {
        return 1;
    } else {
        // Recursive case
        return n * factorial(n - 1);
    }
}

int main() {
    int number = 5;
    printf("Factorial of %d is %d\n", number, factorial(number));
    return 0;
}
```

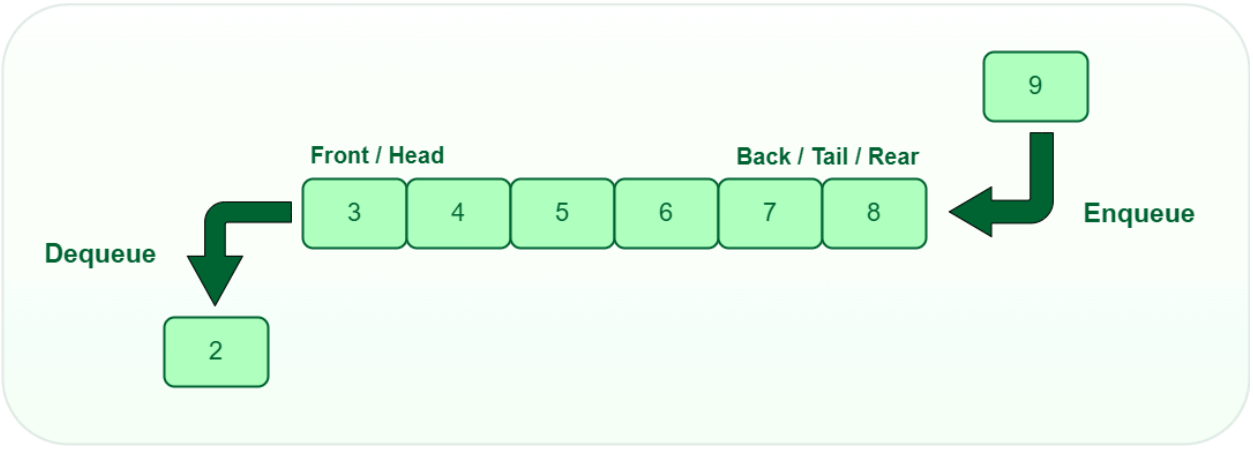
Unit 6: Queues and Recursion

- **Introduction to Queues**
  - A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. The first element inserted is the first to be removed.

- **queue**: A list with the restriction that insertions are done at one end and deletions are done at the other
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the element at the front.



`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

Priority Queue:

- This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority.
- The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values.
- The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.

Working of Queue

Queue operations work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue
- `REAR` track the last element of the queue
- initially, set value of `FRONT` and `REAR` to -1

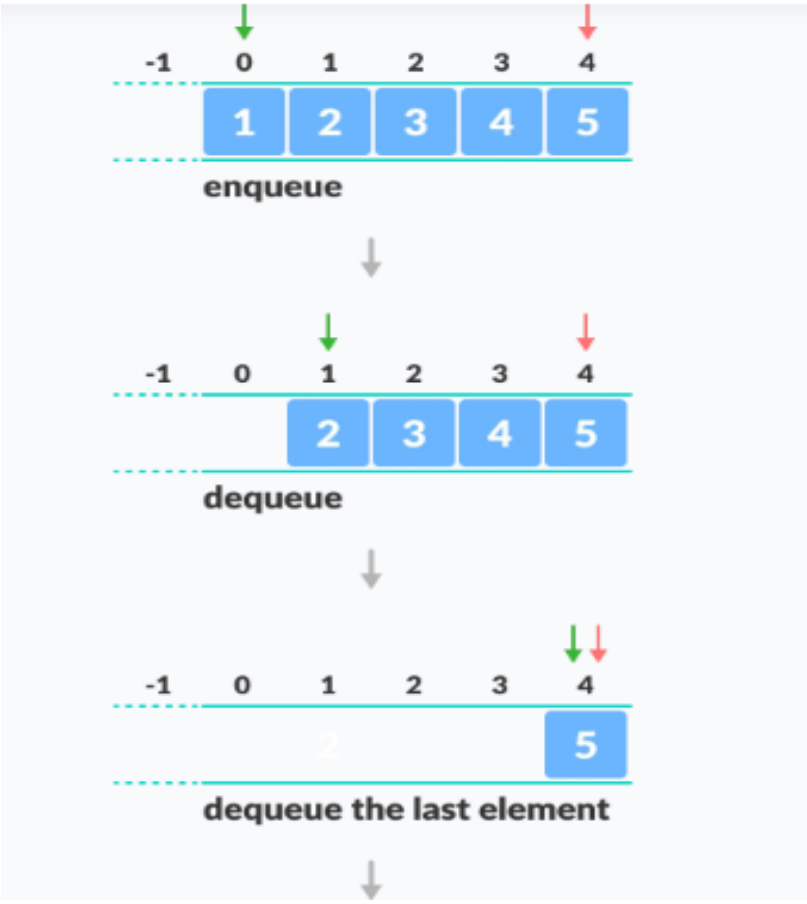
Enqueue Operation

- check if the queue is full
- for the first element, set the value of `FRONT` to 0
- increase the `REAR` index by 1
- add the new element in the position pointed to by `REAR`

Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1


FRONT
REAR
-1
0
1
2
3
4
empty queue
1
enqueue the first element
2
enqueue



- **Implementation of Queues**
  - Queues can be implemented using arrays or linked lists.
  -

```c
/*
 * C Program to Implement a Queue using an Array
 */
#include <stdio.h>

#define MAX 50

void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
        printf("1.Insert element to queue \n");
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);

switch (choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
```

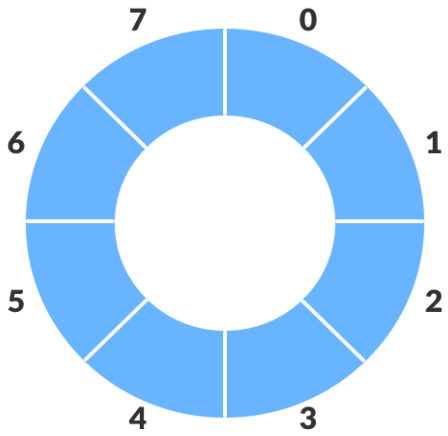
```
            case 3:
            display();
            break;
            case 4:
            exit(1);
            default:
            printf("Wrong choice \n");
        } /* End of switch */
    } /* End of while */
} /* End of main() */
```

```
1.  void insert()
2.  {
3.     int add_item;
4.     if (rear == MAX - 1)
5.     printf("Queue Overflow \n");
6.     else
7.     {
8.          if (front == - 1)
9.          /*If queue is initially empty */
10.         front = 0;
11.         printf("Inset the element in
    queue : ");
12.         scanf("%d", &add_item);
13.         rear = rear + 1;
14.         queue_array[rear] = add_item;
15.    }
16. } /* End of insert() */
17.
```

```c
void delete()
{
   if (front == - 1 || front > rear)
   {
        printf("Queue Underflow \n");
        return ;
   }
   else
   {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
   }
} /* End of delete() */

void display()
{
   int i;
   if (front == - 1)
        printf("Queue is empty \n");
   else
   {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
   }
} /* End of display() */
```

- **Types of Queues**
  - **Circular Queue**: The last position is connected to the first, forming a circle.
  - **Dequeue**: A double-ended queue where elements can be added or removed from both ends.

- **Circular Queue Data Structure:** A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus forming a circle-like structure.



Circular Queue Operations

The circular queue work as follows:

- two pointers `FRONT` and `REAR`
- `FRONT` track the first element of the queue
- `REAR` track the last elements of the queue
- initially, set value of `FRONT` and `REAR` to -1

1. Enqueue Operation

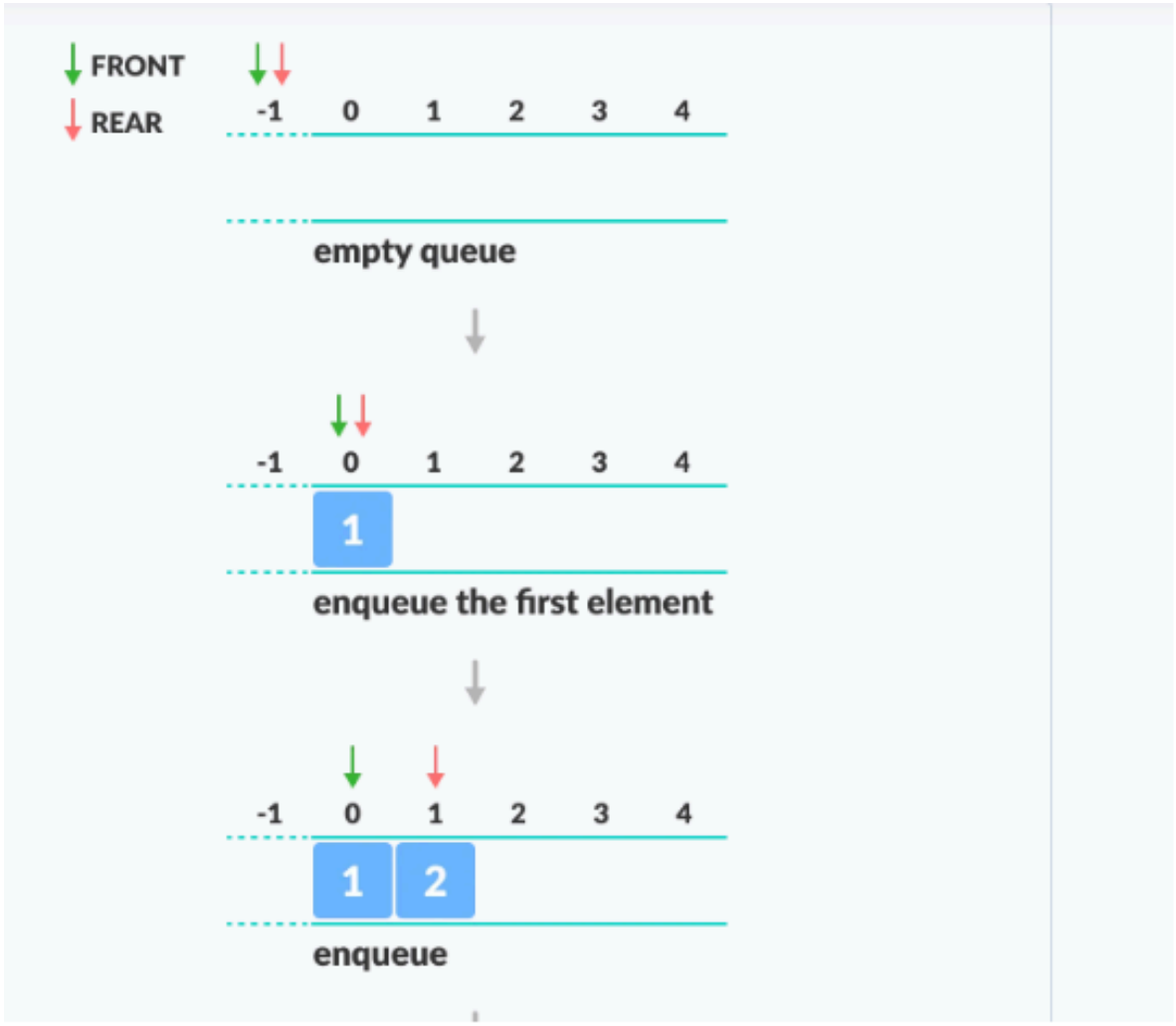
- check if the queue is full
- for the first element, set value of `FRONT` to 0
- circularly increase the `REAR` index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by `REAR`

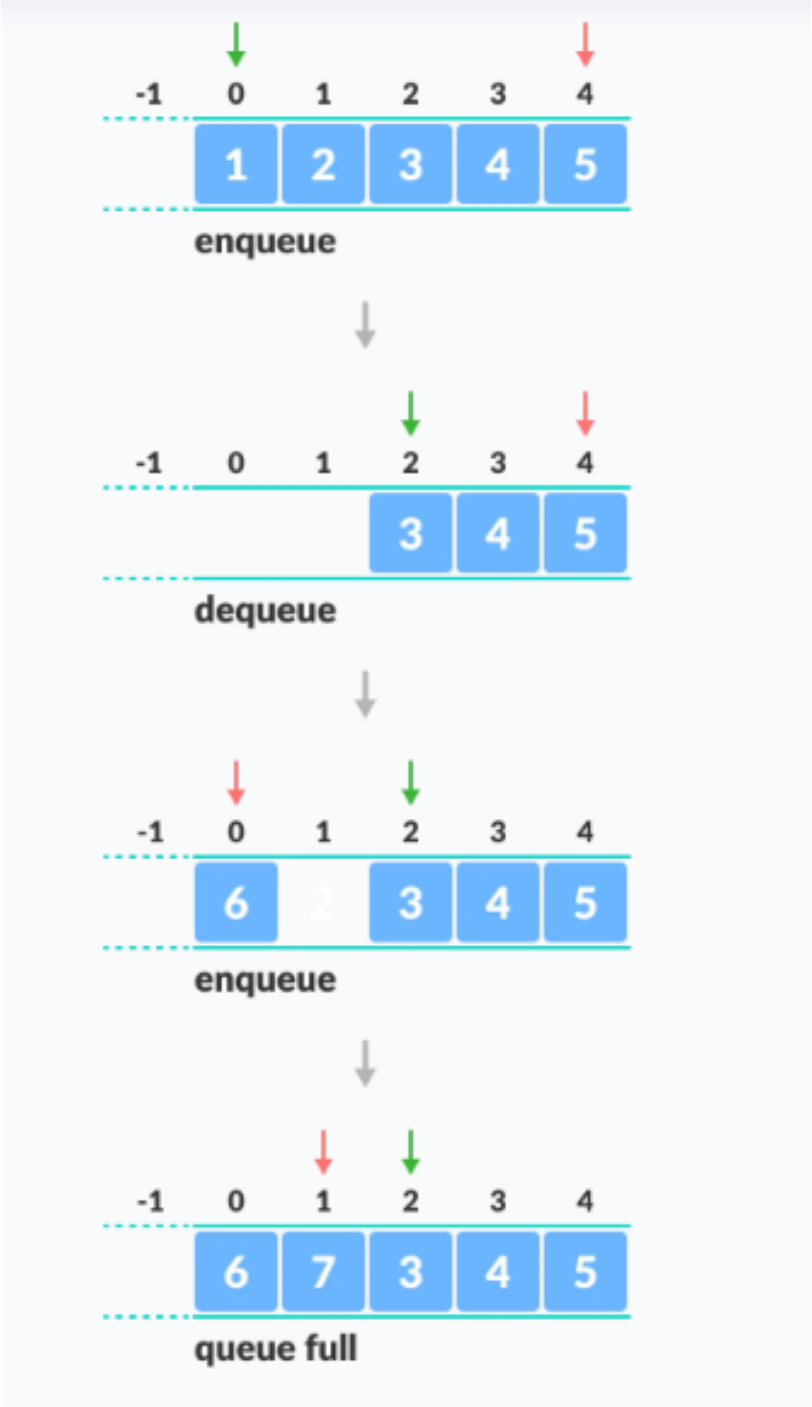
2. Dequeue Operation

- check if the queue is empty
- return the value pointed by `FRONT`
- circularly increase the `FRONT` index by 1
- for the last element, reset the values of `FRONT` and `REAR` to -1

However, the check for full queue has a new additional case:

- Case 1: `FRONT` = 0 && `REAR == SIZE - 1`
- Case 2: `FRONT = REAR + 1`


FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue



- **Recursion**
  - **Recursion** is a process where a function calls itself. It is essential in problem-solving techniques like divide and conquer.

Recursion: A Detailed Explanation

Recursion is a powerful and fundamental programming concept where a function calls itself to solve a problem. This self-referential behavior is useful for breaking down complex problems into smaller, more manageable subproblems. When used effectively, recursion simplifies code, making it more readable and often leading to elegant solutions for problems that would otherwise be complicated to solve using iterative approaches.

Key Concepts of Recursion:

1. **Base Case**: The most crucial part of any recursive function is its base case, which acts as a stopping condition. Without a base case, the function would continue to call itself indefinitely, leading to infinite recursion and a stack overflow error. The base case tells the function when to stop the recursion and return a result directly.
2. **Recursive Case**: Besides the base case, a recursive function includes a recursive case where the function breaks the problem down into smaller subproblems and calls itself with updated parameters. The solution to the larger problem is then constructed using the results of these smaller subproblems.
3. **Stack Mechanism**: Recursion relies on the function call stack, a data structure that stores information about function calls. Each time a recursive function is called, a new frame is pushed onto the stack, storing the function's local variables, parameters, and return address. When the base case is reached, the function returns values, and the stack frames are popped off in reverse order, ultimately leading to the final result.

Example: Factorial of a Number

The factorial of a number n (denoted as $n!$) is a classic example of recursion. The factorial is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

Alternatively, it can be defined recursively as:

$$n! = n \times (n-1)!$$

with the base case being $0! = 1$.

Here's the Python implementation using recursion:

```
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case
    if (n == 0) {
        return 1;
    }
    // Recursive case
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    int result = factorial(num);
    printf("Factorial of %d is %d\n", num, result);

    return 0;
}
```

In this example:

- The base case is when n=0, where the function returns 1.
- The recursive case is when n is greater than 0, where the function calls itself with $n-1$.

Example: Fibonacci Sequence

The Fibonacci sequence is another example where recursion is useful. The sequence is defined as:

$$F(n)=F(n-1)+F(n-2)$$

with the base cases being:
$F(0)=0$ and $F(1)=1$

Types of Recursion

1. **Direct Recursion**: This is the most straightforward form where a function calls itself directly.

```
#include <stdio.h>

void func() {
    func(); // Recursive call without base case
}

int main() {
    func(); // Calling the recursive function
    return 0;
}
```

2. **Indirect Recursion**: In indirect recursion, a function calls another function, and that function, in turn, calls the original function, creating a recursive loop.

```
#include <stdio.h>

void funcB(); // Forward declaration of funcB

void funcA() {
    funcB();  // funcA calls funcB
}

void funcB() {
    funcA();  // funcB calls funcA
}

int main() {
    funcA();  // Start the mutual recursion
    return 0;
}
```

3. **Tail Recursion**: A special form of recursion where the recursive call is the last operation in the function. Some compilers and interpreters optimize tail recursion to avoid stack overflow issues.

```
#include <stdio.h>

// Tail recursive function to calculate factorial
unsigned long long tail_recursive_factorial(int n, unsigned long long accumulator) {
    if (n == 0) {
        return accumulator;
    } else {
        return tail_recursive_factorial(n - 1, n * accumulator);
    }
}

int main() {
    int num = 5;  // Example number
    unsigned long long result = tail_recursive_factorial(num, 1);
    printf("Factorial of %d is %llu\n", num, result);
    return 0;
}
```

Pros and Cons of Recursion

Advantages:

- **Simpler Code**: For problems like tree traversal, the recursive approach is often more intuitive and leads to shorter, cleaner code.
- **Problem Solving**: Recursion is naturally suited for problems broken down into smaller, similar subproblems (e.g., divide and conquer strategies like quicksort mergesort).

Disadvantages:

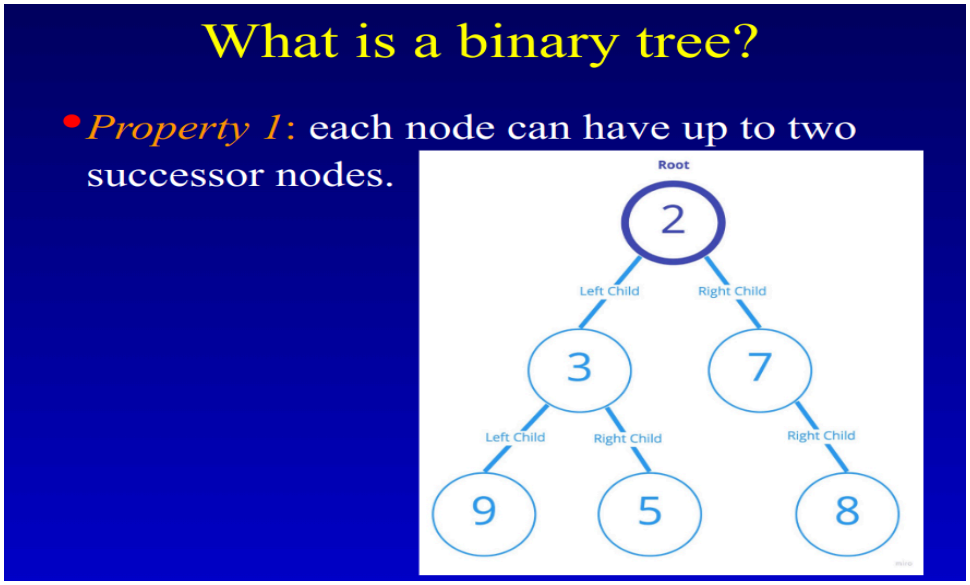
- **Memory Overhead**: Each recursive call requires extra memory to store the function call on the stack. This can lead to memory inefficiency and, in extreme cases, stack overflow.
- **Slower Execution**: Recursion can sometimes be slower than an iterative approach because each function call adds to the overhead.

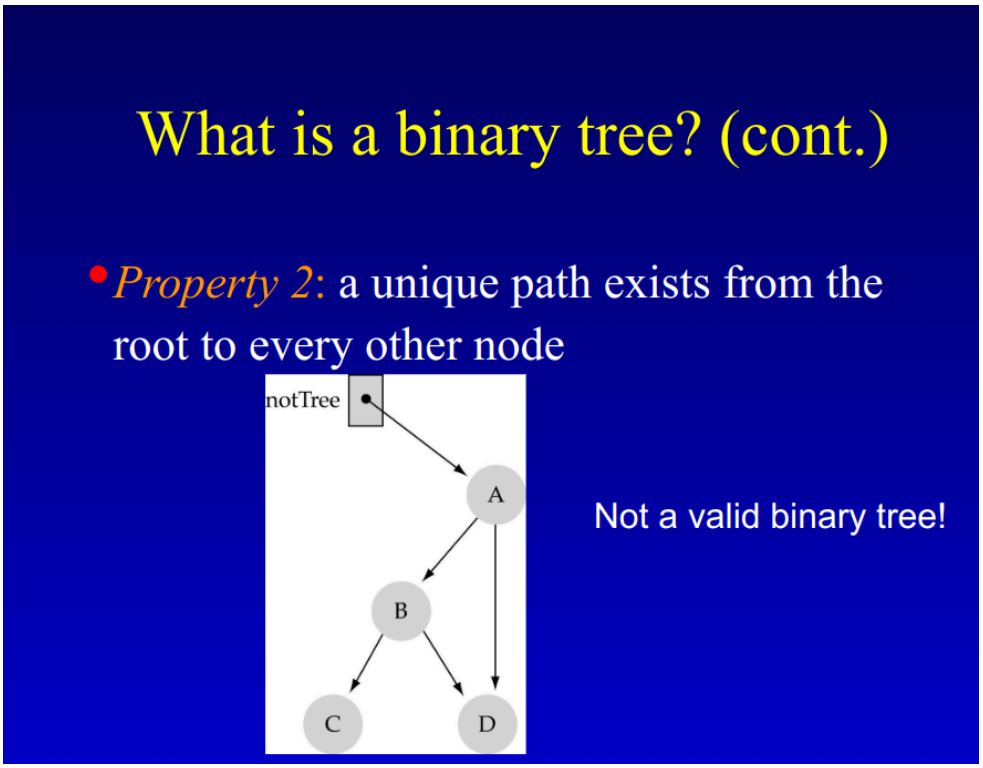
Real-World Applications of Recursion

1. **Tree Traversals**: Recursion is widely used to traverse hierarchical data structures like trees (e.g., binary trees, n-ary trees). Common tree traversals such as in-order, pre-order, and post-order are typically implemented recursively.

2. **Divide and Conquer Algorithms**: Algorithms like merge sort, quicksort, and binary search use recursion to break problems down into smaller subproblems.
3. **Dynamic Programming**: Problems like the Fibonacci sequence, longest common subsequence, and matrix chain multiplication use recursion and memoization to optimize solutions.
4. **Graph Algorithms**: Depth-first search (DFS) is a graph traversal algorithm often implemented using recursion.
5. **Backtracking**: Recursion is fundamental in solving problems where all possible solutions must be explored (e.g., N-queens problem, Sudoku solver).

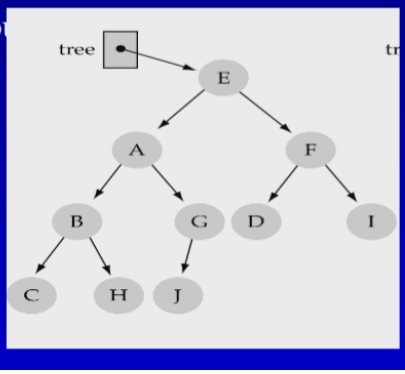
Unit 7: Binary Search Tree

5. **Binary Trees**
   - A **binary tree** is a hierarchical structure where each node has at most two children.
   - 

   - 

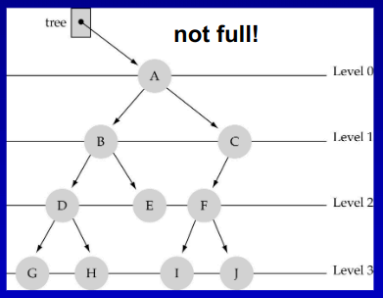
Some terminology

- The successor nodes of a node are called its *children*
- The predecessor node of a node is called its *parent*
- The "beginning" node is called the *root* (has no parent)
- A node witho



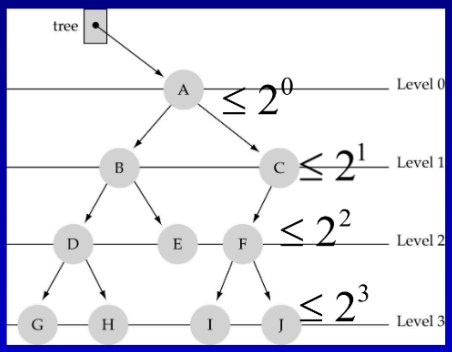
Some terminology (cont'd)

- Nodes are organize in levels (indexed from 0).
- **Level (or depth) of a node**: number of edges in the path from the root to that node.
- **Height of a tree h**: #levels = L (Warning: some books define h as #levels-1).
- **Full tree:** every node has exactly two children ***and*** all the leaves are on the same level



What is the max #nodes at some level *l*?

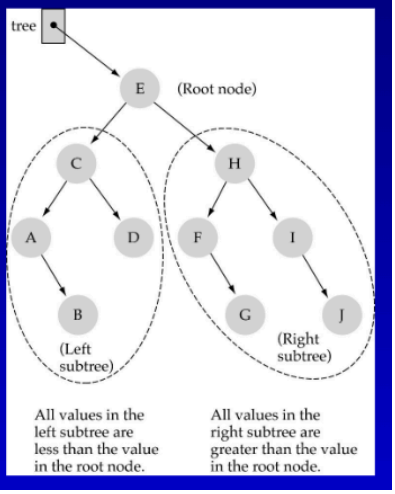
The max #nodes at level l is 2^l where $l=0,1,2,\ldots,L-1$

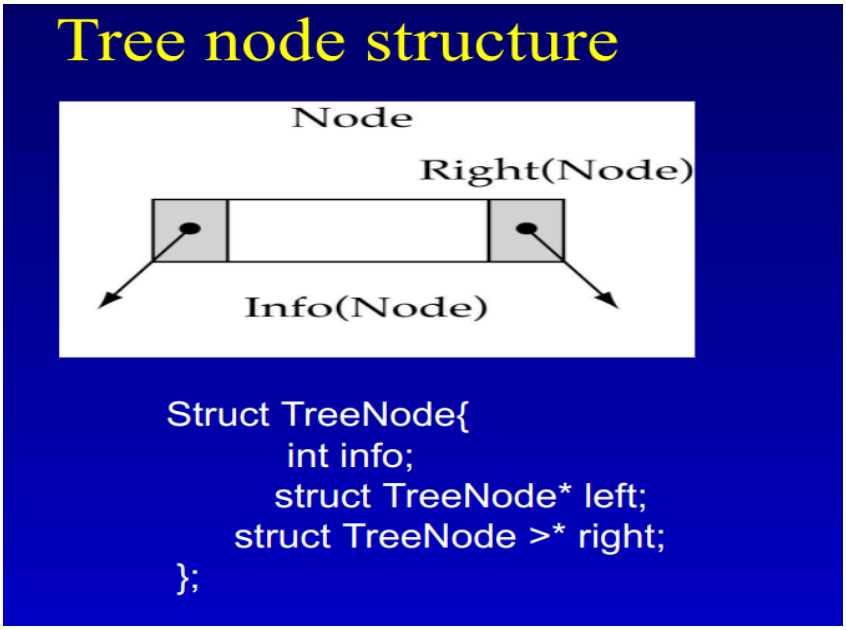


Binary **Search** Trees (BSTs)

- **Binary Search Tree Property**:

The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



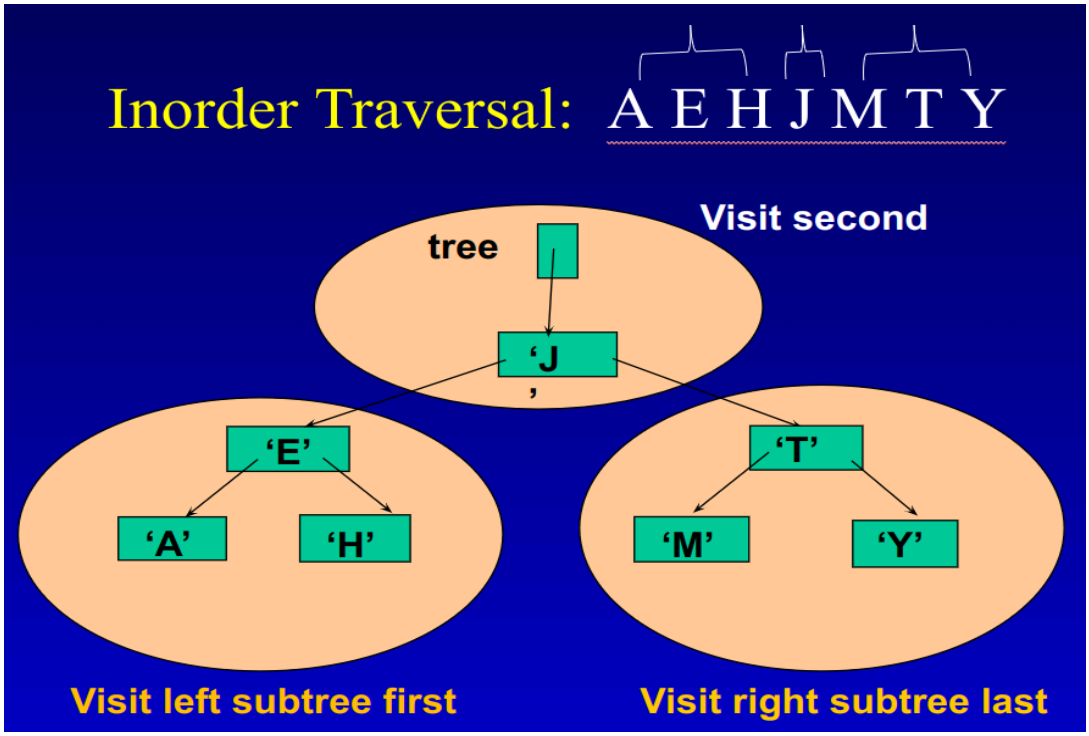


6. **Tree Traversals**
    - **Pre-order**: Visit root, left subtree, right subtree.
    - **In order**: Visit the left subtree, root, and right subtree.
    - **Post-order**: Visit the left subtree, right subtree, and root.

1. In **inorder traversal**, we visit the left subtree first, then the root node, and finally the right subtree. This traversal is commonly used for binary search trees (BST) because it visits the nodes in ascending order.

    Steps:

1. Traverse the left subtree by recursively calling the in-order function.
2. Visit the root node.
3. Traverse the right subtree by recursively calling the in-order function.

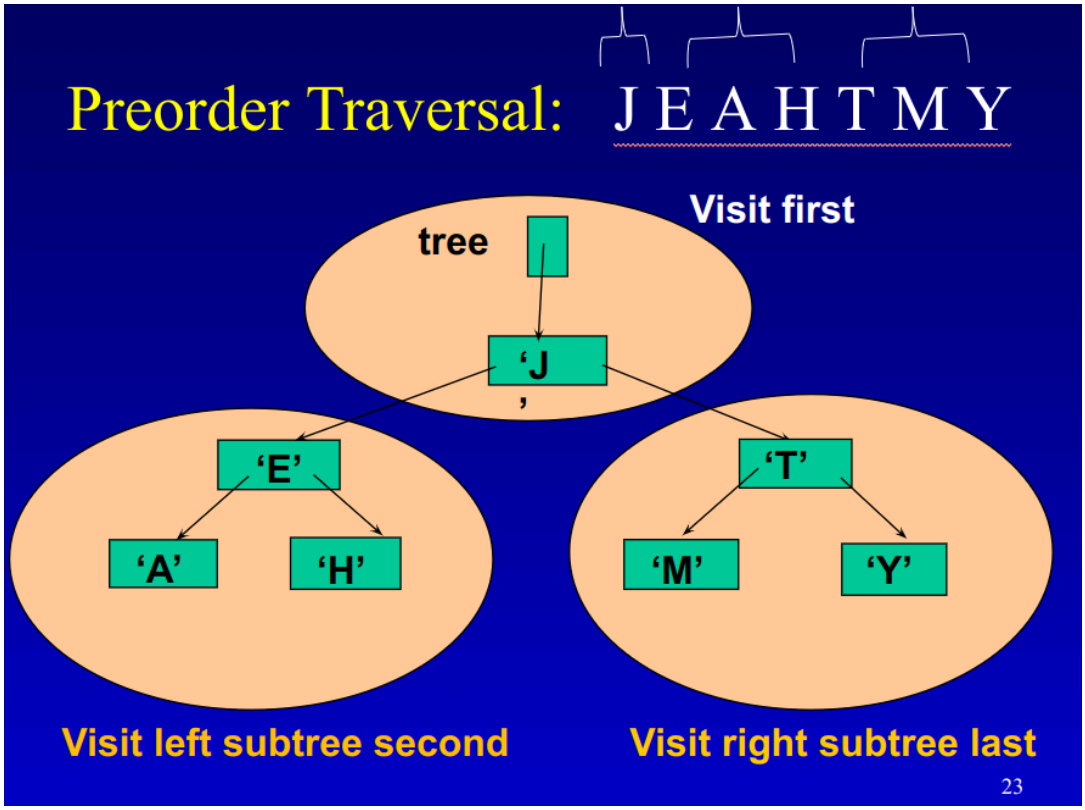


2. Preorder Traversal (Root, Left, Right)

In **preorder traversal**, we visit the root node first, the left subtree, and finally the right subtree. Preorder traversal is useful when copying or printing the tree structure.

Steps:

1. Visit the root node.
2. Traverse the left subtree by recursively calling the preorder function.
3. Traverse the right subtree by recursively calling the preorder function.

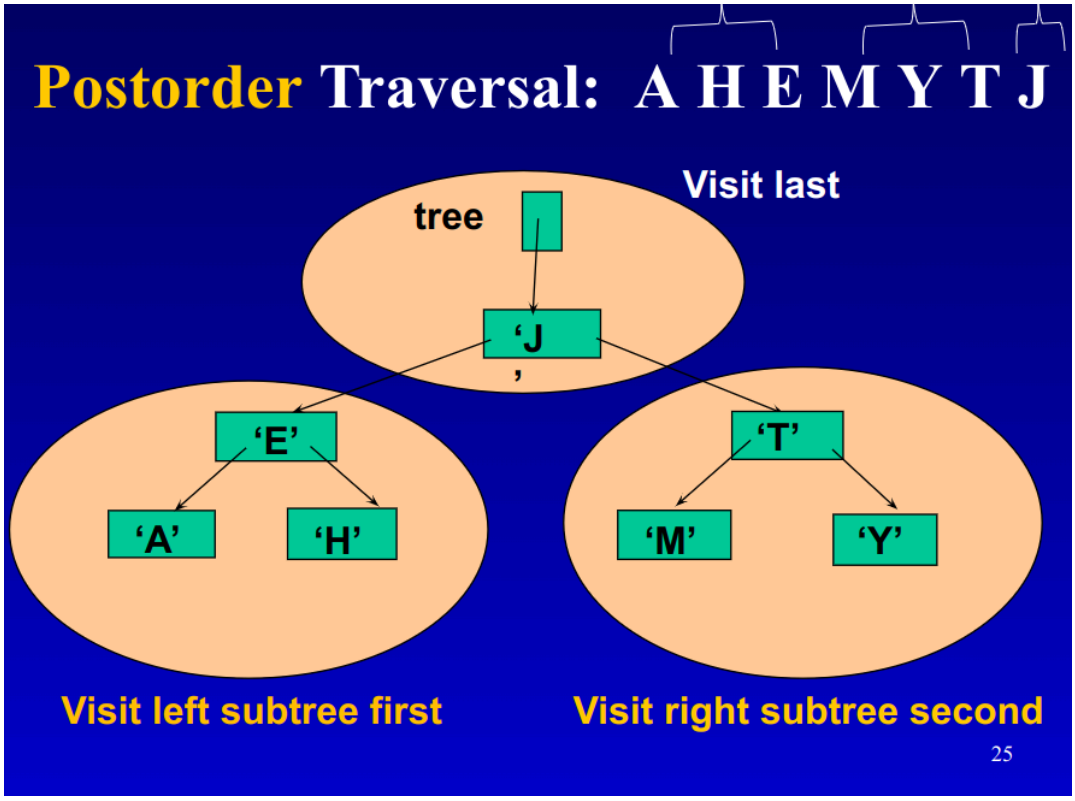


3. Postorder Traversal (Left, Right, Root)

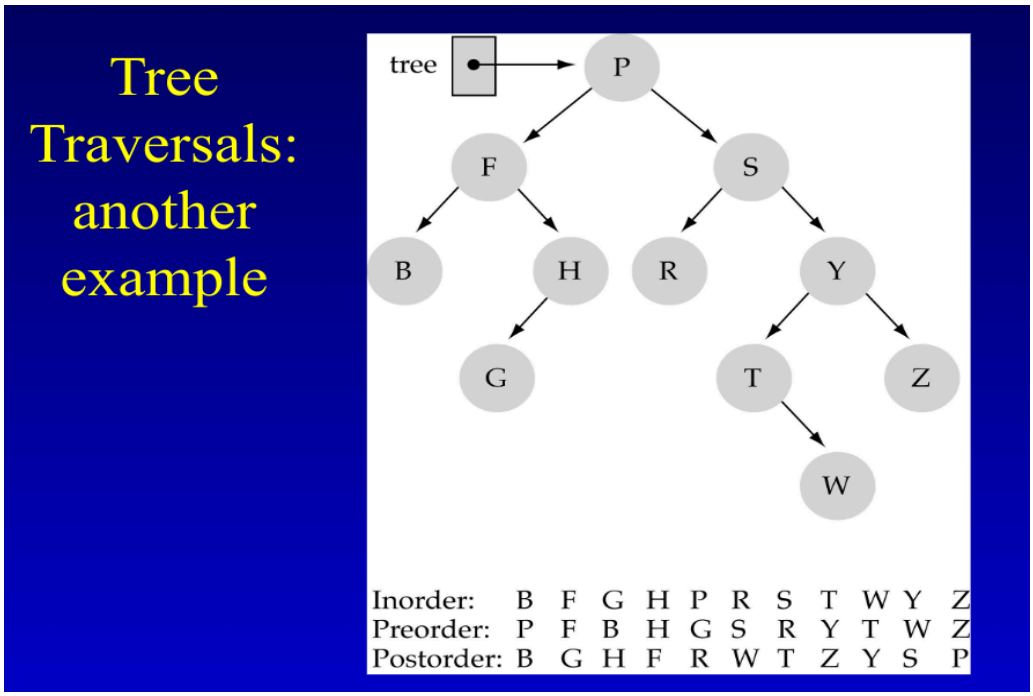
In **postorder traversal**, we traverse the left subtree first, then the right subtree, and finally the root node. Postorder is useful for deleting or evaluating trees.

Steps:

1. Traverse the left subtree by recursively calling the postorder function.
2. Traverse the right subtree by recursively calling the postorder function.
3. Visit the root node.


Postorder Traversal: A H E M Y T J
Visit last
tree
'J'
'E'
'T'
'A'
'H'
'M'
'Y'
Visit left subtree first
Visit right subtree second
25

Example:


Tree Traversals: another example
tree
P
F
S
B
H
R
Y
G
T
Z
W
Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Unit 8: Sorting and Searching Algorithms

3. **Search Algorithms**
    - **Linear Search**: Sequentially checks each element.
    - **Binary Search**: Divides the array in half repeatedly to find the element (requires sorted array).
4. **Sorting Algorithms**
    - **Bubble Sort**: Repeatedly swaps adjacent elements if they are in the wrong order.
    - **Insertion Sort**: Builds a sorted array by inserting elements into the correct position.
    - **Quick Sort**: A divide-and-conquer algorithm that partitions the array into sub-arrays based on a pivot.
    - **Selection Sort**: Repeatedly selects the minimum element and swaps it with the front element.
    - **Merge Sort**: A divide-and-conquer algorithm that splits the array and merges the sorted sub-arrays.
    - **Heap Sort**: Sorts using a binary heap structure, focusing on the maximum or minimum element.

Search Algorithms

1. **Linear Search**:
    - **Concept**: Linear search is the simplest search algorithm where each element of the array is checked sequentially to find the target element.
    - **Time Complexity**: O(n), where n is the number of elements in the array.
    - **Advantages**: Works on unsorted arrays.
    - **Disadvantages**: Inefficient for large datasets as it may require scanning all elements.
2. **Example**:

```
int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i;
      }
   }
   return -1;
}
```

Binary Search:

- **Concept**: Binary search divides the sorted array in half and eliminates one half of the search space in each iteration. It requires the array to be sorted.
- **Time Complexity**: O(log n), where n is the number of elements in the array.
- **Advantages**: Efficient for large datasets if the array is sorted.
- **Disadvantages**: Requires the array to be sorted.

Example:

```
int binarySearch(int arr[], int left, int right, int target) {
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == target) return mid;
      else if (arr[mid] < target) left = mid + 1;
      else right = mid - 1;
   }
   return -1;
}
```

Sorting Algorithms

1. **Bubble Sort**:
    - **Concept**: In this sorting algorithm, adjacent elements are repeatedly compared and swapped if they are in the wrong order. This continues until the array is sorted.
    - **Time Complexity**: $O(n^2)$, where n is the number of elements in the array.
    - **Advantages**: Simple to understand and implement.
    - **Disadvantages**: Not efficient for large datasets.

    Example:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Insertion Sort:

- **Concept**: Builds the final sorted array one element at a time by picking elements and placing them in their correct positions in the sorted portion of the array.
- **Time Complexity**: O(n²) for worst-case scenarios, but O(n) for nearly sorted data.
- **Advantages**: More efficient than bubble sort on small or nearly sorted arrays.
- **Disadvantages**: Inefficient on large datasets.

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Quick Sort:

- **Concept**: A divide-and-conquer algorithm that picks a pivot element, partitions the array into two sub-arrays, and recursively sorts the sub-arrays.
- **Time Complexity**: O(n log n) on average, but O(n²) in the worst case if the pivot is poorly chosen.

- **Advantages**: Very efficient for large datasets and generally faster than merge sort.
- **Disadvantages**: Worst-case performance can be poor; requires careful selection of the pivot.

```
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j < high; j++) {
      if (arr[j] <= pivot) {
         i++;
         int temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
   }
   int temp = arr[i + 1];
   arr[i + 1] = arr[high];
   arr[high] = temp;
   return (i + 1);
}

void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
```

Selection Sort:

- **Concept**: Repeatedly selects the minimum element from the unsorted portion of the array and swaps it with the element at the beginning of the unsorted section.
- **Time Complexity**: $O(n^2)$, where n is the number of elements.
- **Advantages**: Simple to understand and implement, reduces the number of swaps compared to bubble sort.
- **Disadvantages**: Not efficient for large datasets

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Merge Sort:

- **Concept**: A divide-and-conquer algorithm that splits the array into two halves, recursively sorts each half, and then merges them to form a sorted array.
- **Time Complexity**: O(n log n), where n is the number of elements.
- **Advantages**: Stable and works well on large datasets.
- **Disadvantages**: Requires extra space for merging, which can be a disadvantage in memory-constrained environments.

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Heap Sort:

- **Concept**: Sorts an array by first building a binary heap, either max-heap (for ascending order) or min-heap (for descending order), and then repeatedly extracting the root (maximum or minimum element).
- **Time Complexity**: O(n log n), where n is the number of elements.
- **Advantages**: More space-efficient compared to merge sort.
- **Disadvantages**: Not a stable sorting algorithm, as the relative order of equal elements may not be preserved.

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp
```

Comparison of Sorting Algorithms

Algorithm	Time Complexity	Space Complexity	Stability	Best Use Case	Advantages	Disadvantages
Bubble Sort	Best: O(n) Average: O(n²) Worst: O(n²)	O(1)	Yes	Small datasets or nearly sorted arrays	Simple to implement, stable, and detects small issues in nearly sorted data.	Inefficient for large datasets with a worst-case time complexity of O(n²).
Selection Sort	Best: O(n²) Average: O(n²) Worst: O(n²)	O(1)	No	Small datasets where memory is constrained	Simple to implement, low memory usage.	Always has O(n²) time complexity, even for already sorted arrays.
Insertion Sort	Best: O(n) Average: O(n²) Worst: O(n²)	O(1)	Yes	Nearly sorted datasets	Efficient for small datasets or nearly sorted arrays. Stable and simple.	O(n²) time complexity for unsorted arrays.

Quick Sort	Best: O(n log n)	O(log n)	No	Large datasets with random distribution	Very efficient for large datasets, faster than merge sort in practice. Divide-and-conquer approach.	Worst-case time complexity of $O(n^2)$ if pivot is poorly chosen.
	Average: O(n log n)					Recursive approach, requiring additional stack space for function calls.
	Worst: $O(n^2)$					
Merge Sort	Best: O(n log n)	O(n)	Yes	Linked lists or large datasets where stability is crucial	Always O(n log n) time complexity regardless of input. Stable and efficient for large datasets. Suitable for linked lists and external sorting.	Requires extra space proportional to the size of the array.
	Average: O(n log n)					
	Worst: O(n log n)					

Heap Sort	Best: O(n log n) Average: O(n log n) Worst: O(n log n)	O(1)	No	Large datasets where memory usage is a concern and stability is not required	Doesn't require additional memory. Always O(n log n) time complexity. Efficient for large datasets, especially when memory space is limited.	Not stable, and slower than quick sort in practice. Complex to implement.
Radix Sort	Best: O(nk) Average: O(nk) Worst: O(nk)	O(n + k)	Yes	Sorting integers or strings when the length of the elements (k) is small compared to the number of elements (n)	Efficient for sorting integers or strings. Stable and non-comparative. Works well when range of input data is known and fixed.	Limited to certain types of data (e.g., integers, strings) and can require additional space. Depends on digit length and base.

Key Points of Comparison:

1. **Time Complexity**:

- **Best Performers**: Quick Sort, Merge Sort, and Heap Sort have average time complexities of O(n log n), making them efficient for large datasets.
- **Slow Performers**: Bubble Sort, Selection Sort, and Insertion Sort have $O(n^2)$ worst-case time complexities, making them less suitable for large arrays.

2. **Space Complexity**:
   - **In-Place Algorithms**: Algorithms like Quick Sort, Heap Sort, and Bubble Sort have O(1) space complexity.
   - **Extra Space**: Merge Sort requires O(n) space due to the need to merge arrays, while Radix Sort also requires extra space based on the number of digits.
3. **Stability**:
   - **Stable Algorithms**: Insertion Sort, Merge Sort, Radix Sort, and Bubble Sort are stable, meaning they maintain the relative order of equal elements.
   - **Unstable Algorithms**: Quick Sort, Selection Sort, and Heap Sort are unstable by default.
4. **Efficiency**:
   - **Quick Sort** is often preferred for large datasets due to its average O(n log n) time complexity and practical efficiency.
   - **Merge Sort** is a good choice when stability is important or for linked lists.
   - **Heap Sort** is useful in memory-constrained environments.
5. **Best Use Cases**:
   - **Bubble Sort & Insertion Sort**: Good for small or nearly sorted datasets.
   - **Quick Sort & Merge Sort**: Excellent for large datasets.
   - **Heap Sort**: Efficient when space is limited, and stability is not a concern.
   - **Radix Sort**: Good for integer or string sorting when the dataset fits its constraints.

Thank you.

Dr. Amar Nath

AP, CSE