#computer #college-studies IT #Sem2

By: Yashraj Maher

Syllabus:

Unit 1: Data Structures and Algorithms

- **Data Structures**
- Introduction to linear and non-linear data structures
- **Algorithm Analysis**
- Growth rates
- Estimating growth rates
- Big O notation

Unit 2: Arrays

- **Need for Arrays**
- **Linear Arrays**
- Representation of linear arrays:
- Row measure order
- Column measure order
- **Operations on Arrays**
- Traversing , Insertion, Modification, Deletion
- Merging Linear Array
- 2-D array introduction
- representation of 2D arrays
- spark matrices
- Need of searching & sorting
- Searching
  - Linear Search
  - Binary Search
- Sorting
  - Selection Sort
  - Insertion Sort

Unit 3: Stack & Queue

- Introduction Stack
- Operation on Stack, Stack Implementation array, application of stacks
- (Explanation, representation, evaluation) Expression Notation (Prefix, infix, postfix), conversion of expression (Infix to Postfix, Prefix to Infix)
- Queue
- Introduction
- types of Queue (Circular & Dequeue)
- Queue implementation using array
- operations on queues (traversing, insertion, deletion & modification) application of queue (priority of queue)

Unit 1: Data Structures and Algorithms

Introduction to Data Structures

Definition

- **Data Structure**: A data structure represents the logical relationship between individual data elements. It is a method of organizing and storing data to facilitate efficient access and modification, considering both the elements and their interconnections.
- **Impact on Programs**: Data structures influence both the structural and functional aspects of a program. A program can be expressed as **Program = Algorithm + Data Structure**, where:
  - **Algorithm**: A step-by-step procedure (set of instructions) to solve a specific task.
  - **Data Structure**: The way data is organized, which directly affects the efficiency of the algorithm's operations.
- **Efficiency**: The performance of an algorithm depends heavily on selecting an appropriate data structure. For handling large datasets, as emphasized in *Data Structures and Algorithm Analysis in C*, careful attention to efficiency is critical.

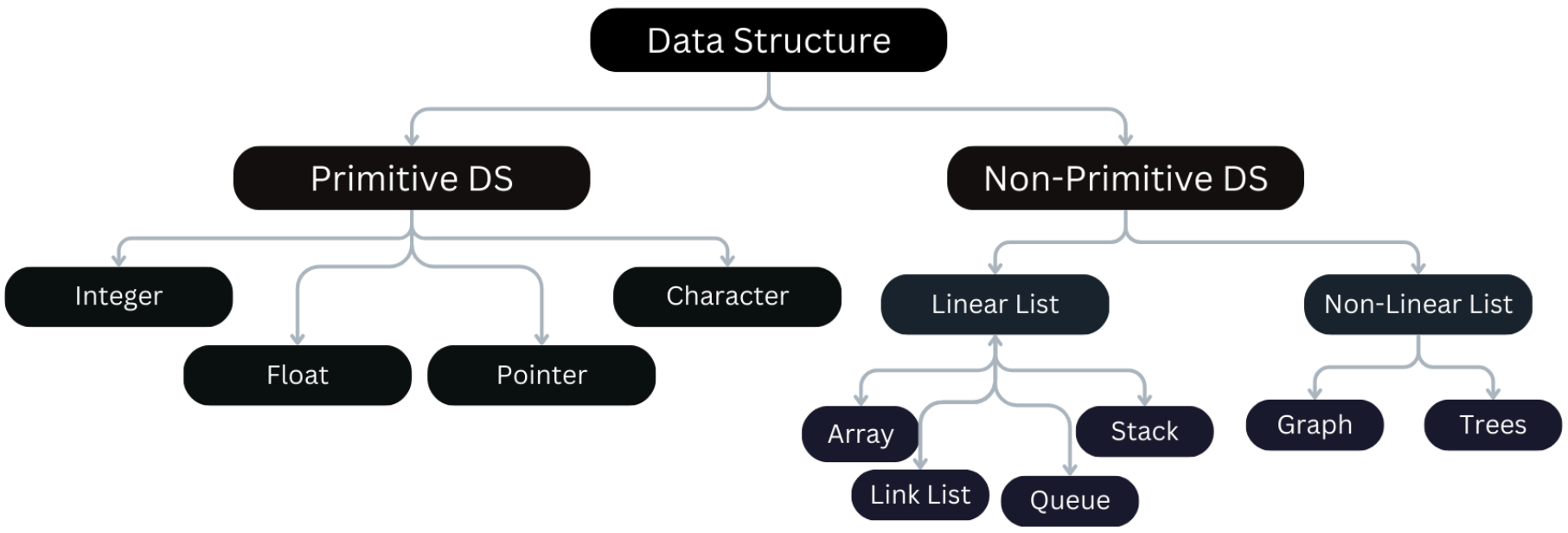
Classification of Data Structures

Data structures are broadly categorized into two types:

1. **Primitive Data Structures**
   - Basic data types directly operated upon by machine instructions.
   - Examples: Integer, Float, Character, Pointer, String Constants.
2. **Non-Primitive Data Structures**
   - Advanced constructs derived from primitive types, designed to manage collections of data.
   - Subcategories:
     - **Linear List Data Structures**: Elements arranged sequentially (e.g., Arrays, Linked Lists, Stacks, Queues).
     - **Non-Linear List Data Structures**: Elements organized hierarchically or in a networked manner (e.g., Trees, Graphs).

Diagram

Below is a conceptual representation of the classification:



Primitive Data Structures

- **Definition**: Foundational data types inherent to programming languages, used to store single values.
- **Types**:
  1. **Integer**: Represents whole numbers (e.g., -5, 0, 42). Used for counting, indexing, and arithmetic.

2. **Float**: Represents real numbers with decimals (e.g., 3.14, -0.001). Used for precision in calculations.
3. **Character**: Represents single symbols (e.g., 'a', '1', '$'). Stored using ASCII/Unicode for text processing.
4. **Pointer**: Stores memory addresses. Essential for dynamic memory management and linking data structures.

- **Significance**: These are the building blocks for all other data structures, directly manipulated by hardware.

Non-Primitive Data Structures

- **Definition**: Complex structures built from primitive types to manage collections of data, emphasizing relationships between elements.
- **Examples**: Arrays, Linked Lists, Stacks, Queues, Trees, Graphs.
- **Design Consideration**: The efficiency of operations (e.g., insertion, deletion) depends on the chosen structure. For instance, frequent insertions favor linked lists over arrays.
- **Common Operations**:
  - **Update/Modification**: Modifying data (e.g., insertion, deletion).
  - **Selection/Access**: Retrieving specific elements (e.g., finding an element).
  - **Searching**: Locating an element by key.
  - **Sorting**: Arranging elements in order.
  - **Merging**: Combining multiple structures.
  - **Traversal**: Visiting all elements.

Algorithm Analysis

Algorithm analysis is a critical component of computer science that evaluates the efficiency and performance of algorithms, particularly as the size of the input data grows. This section provides a detailed exploration of algorithm analysis, covering growth rates, methods for estimating them, Big O notation, and their practical implications, tailored to the context of data structures and programming as outlined in the Semester 2 IT syllabus.

Overview of Algorithm Analysis

Algorithm analysis involves assessing how an algorithm's resource usage—primarily time (runtime) and space (memory)—scales with the input size, denoted as (n). The goal is to predict performance under varying conditions, ensuring that algorithms and their associated data structures are suitable for real-world applications. This process is foundational for selecting appropriate data structures and optimizing program design, as highlighted in the syllabus under Unit 1: Data Structures and Algorithms.

Growth Rates

Definition

Growth rates measure the rate at which an algorithm's resource requirements (time or space) increase as the input size (n) grows. This is typically expressed as a function of (n), reflecting the number of operations or memory units needed.

Factors Influencing Growth Rates

- **Input Size ((n))**: The number of elements or data points processed (e.g., array length, number of nodes in a graph).
- **Operation Complexity**: The type and frequency of operations (e.g., comparisons, assignments, memory allocations).
- **Hardware Dependencies**: While analysis focuses on theoretical behavior, real-world performance may vary due to CPU speed, cache efficiency, or memory access times.

Examples of Growth Patterns

- **Constant Growth**: Resource usage remains unchanged regardless of (n).
- **Linear Growth**: Resource usage increases proportionally with (n).
- **Quadratic Growth**: Resource usage increases with the square of (n), often due to nested loops.
- **Logarithmic Growth**: Resource usage grows slowly, typically seen in divide-and-conquer strategies.

Importance

Understanding growth rates helps developers anticipate how an algorithm will perform with large datasets, guiding the choice between, for instance, an array ($O(1)$ access) versus a linked list ($O(n)$ access for random elements).

Estimating Growth Rates

Methodology

Estimating growth rates involves analyzing the algorithm's steps to determine the dominant operations as (n) increases. This is typically done by:

1. **Counting Operations**: Identify the number of basic operations (e.g., comparisons, assignments) executed.
2. **Identifying the Worst Case**: Focus on the maximum resource usage, which provides an upper bound.
3. **Simplifying the Function**: Ignore lower-order terms and constants, as they become negligible for large (n).

Steps in Estimation

- **Break Down the Algorithm**: Decompose it into individual steps or loops.
- **Analyze Each Step**: Assign a time complexity to each (e.g., a single loop is $O(n)$).
- **Combine Complexities**: Use rules like addition for sequential steps and multiplication for nested operations.
- **Asymptotic Behavior**: Focus on the behavior as (n) approaches infinity.

Example: Linear Search

- **Algorithm**: Search for an element in an unsorted array of size (n).
- **Steps**: Compare each element with the target, potentially checking all (n) elements in the worst case.
- **Growth Rate**: $O(n)$, as the number of comparisons grows linearly with (n).

Example: Bubble Sort

- **Algorithm**: Repeatedly swap adjacent elements if they are in the wrong order.
- **Steps**: Two nested loops—outer loop runs$(n-1)$times, inner loop up to$(n-1)$times per iteration.
- **Growth Rate**: $O((n^2))$, due to $((n-1) \times (n-1))$ comparisons in the worst case.

Practical Considerations

- **Average Case**: Considers the expected number of operations (e.g., linear search may find the target early, reducing to $O(n/2)$ on average).
- **Best Case**: The minimum resource usage (e.g., $O(1)$ if the target is the first element).
- **Amortized Analysis**: Averages cost over multiple operations, useful for dynamic structures like arrays with resizing.

Big O Notation

Definition

Big O notation describes the upper bound of an algorithm's time or space complexity, providing a worst-case scenario as (n) grows. It abstracts away constants and lower-order terms to focus on the dominant factor.

Mathematical Representation

- If $(T(n))$ is the time complexity function, Big O is defined as:

  $$T(n) = O(f(n))$$

  where $(T(n) \leq c \cdot f(n))$ for some constant $(c > 0)$ and all $(n > n_0)$ (a threshold).

Common Big O Complexities

---

1. $O(1)$ **- Constant Time**
   - **Description**: Execution time is independent of (n).
   - **Example**: Accessing an array element by index (e.g., `arr[5]`).
   - **C Example**:

     ```c
     #include <stdio.h>
     int main() {
         int arr[5] = {1, 2, 3, 4, 5};
         printf("Element at index 2: %d\n", arr[2]); // $O(1)$
         return 0;
     }
     ```

   - **Use Case**: Direct memory access operations.
2. $O(n)$ **- Linear Time**
   - **Description**: Time grows linearly with (n).
   - **Example**: Traversing an array to find the sum.

- **C Example**:

```c
#include <stdio.h>
#define SIZE 5
int main() {
    int arr[SIZE] = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < SIZE; i++) { // O(n)
        sum += arr[i];
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

- **Use Case**: Sequential processing (e.g., linear search).

3. $O(logn)$ **- Logarithmic Time**
   - **Description**: Time grows logarithmically, often due to halving the problem size (e.g., binary search).
   - **Example**: Searching in a sorted array using binary search.
   - **C Example** (Simplified Binary Search):

```c
#include <stdio.h>
#define SIZE 5
int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) { // O(log n)
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
int main() {
    int arr[SIZE] = {1, 2, 3, 4, 5};
    printf("Index of 3: %d\n", binarySearch(arr, 0, SIZE-1, 3));
    return 0;
}
```

   - **Use Case**: Efficient search in sorted data (e.g., binary trees).

4. $O(n^2)$ **- Quadratic Time**
   - **Description**: Time grows with the square of (n), typically from nested loops.
   - **Example**: Bubble sort or nested array comparisons.
   - **C Example** (Bubble Sort):

```c
#include <stdio.h>
#define SIZE 5
void bubbleSort(int arr[]) {
    for (int i = 0; i < SIZE - 1; i++) { // O(n²)
        for (int j = 0; j < SIZE - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
int main() {
    int arr[SIZE] = {5, 4, 3, 2, 1};
    bubbleSort(arr);
    for (int i = 0; i < SIZE; i++) printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

   - **Use Case**: Simple sorting algorithms on small datasets.

Other Notations

- **Ω (Omega) Notation**: Lower bound (best-case complexity).
- **Θ (Theta) Notation**: Tight bound (average-case complexity when upper and lower bounds match).
- **Focus in Big O**: Emphasizes the worst-case scenario, ensuring reliability for all inputs.

Rules for Big O Analysis

- **Drop Constants**: $O(2n)$ simplifies to $O(n)$.
- **Drop Lower-Order Terms**: $O(n^2 + n)$ becomes $O(n^2)$.
- **Multiply Nested Loops**: $O(n)$ inside $O(n)$ is $O(n^2)$.

Practical Example: Array vs. Linked List

- **Array Access**: $O(1)$ due to direct indexing.
- **Linked List Access**: $O(n)$ due to sequential traversal.
- **Choice**: Use arrays for frequent access, linked lists for frequent insertions/deletions.

Purpose of Algorithm Analysis

Guiding Data Structure Selection

- **Matching Operations to Complexity**: If an algorithm requires frequent searches, a data structure with O(log n) search (e.g., BST) is preferable over $O(n)$ (e.g., unsorted array).
- **Scalability**: Ensures the algorithm performs acceptably as (n) increases (e.g., O(n²) sorting may fail for large (n)).

Feasibility Assessment

- **Resource Constraints**: Determines if an algorithm fits within time or memory limits (e.g., $O(n^3)$ may be impractical for large (n)).
- **Optimization**: Identifies bottlenecks (e.g., replacing $O(n^2)$ with $O(nlogn)$ sorting like QuickSort).

Educational Value

- **Understanding Trade-offs**: Teaches the balance between time and space complexity (e.g., hashing offers $O(1)$ access but uses more memory).
- **Problem-Solving**: Encourages designing algorithms with growth rates in mind, aligning with real-world efficiency needs.

Advanced Considerations

Amortized Analysis

- **Definition**: Averages the cost of operations over a sequence, useful for dynamic arrays or hash tables.
- **Example**: Doubling an array's size when full costs $O(n)$ once, but amortized cost per insertion is $O(1)$.

Space Complexity

- **Definition**: Measures memory usage as a function of (n).
- **Examples**:
  - $O(1)$: Using a fixed number of variables.
  - $O(n)$: Storing the input array.
  - $O(nlogn)$: Recursive calls in merge sort.

Real-World Factors

- **Cache Efficiency**: $O(n)$ access may vary if data is cache-friendly.
- **Parallelism**: Some O(n²) algorithms can be optimized with multi-threading.

Unit 2: Arrays

Need for Arrays

- **Purpose**: Arrays provide a simple, efficient way to store and manage a fixed-size collection of elements of the same type.
- **Use Cases**:
  - Storing lists (e.g., grades, temperatures).
  - Enabling fast access via indices.
  - Serving as the foundation for other structures (e.g., stacks, queues).
- **Why Arrays?**: They offer direct memory access and are memory-efficient for static data, unlike dynamic structures like linked lists.

Linear Arrays

- **Definition**: A linear array is a collection of elements stored in contiguous memory locations, with each element having a unique predecessor and successor (except the first and last).
- **Representation**:
  - **Row-Major Order**: Elements stored row-by-row (common in C).
    - Example: For a 2D array `arr[2][3] = {{1, 2, 3}, {4, 5, 6}}`, memory layout is `1, 2, 3, 4, 5, 6`.
  - **Column-Major Order**: Elements stored column-by-column (common in Fortran).
    - Example: For the same array, memory layout is `1, 4, 2, 5, 3, 6`.
- **Memory Calculation**:
  - Base address + (index * size of element).
  - Example: For `int arr[5]` at base address 1000, `arr[2]` is at 1000 + (2 * 4) = 1008 (assuming 4 bytes per int).

Operations on Arrays

- **Traversing**: Visiting each element.
  - Time Complexity: $O(n)$.
- **Insertion**: Adding an element at a specific position, shifting subsequent elements right.
  - Time Complexity: $O(n)$.
- **Modification/Update**: Changing an element's value at a given index.
  - Time Complexity: $O(1)$.
- **Deletion**: Removing an element, shifting subsequent elements left.
  - Time Complexity: $O(n)$.

Advantages of Arrays

- Fast, direct access to elements via indices ($O(1)$).
- Simple implementation and memory-efficient for fixed-size data.

Disadvantages of Arrays

- Fixed size limits flexibility (resizing requires a new array).
- Inefficient for frequent insertions/deletions due to shifting.

Applications

- Storing sequential data (e.g., lists, matrices).
- Implementing stacks, queues, and hash tables.

C Program Example: Array Operations

```c
#include <stdio.h>
#define SIZE 5

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

    // Traversing
    printf("Initial array: ");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Insertion (at index 2)
    int value = 10;
    for (int i = SIZE - 1; i > 2; i--) {
        arr[i] = arr[i - 1];
    }
    arr[2] = value;
```

```c
    printf("After insertion at index 2: ");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Modification (update index 1)
    arr[1] = 20;
    printf("After updating index 1: ");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Deletion (at index 3)
    for (int i = 3; i < SIZE - 1; i++) {
        arr[i] = arr[i + 1];
    }
    arr[SIZE - 1] = 0; // Default value
    printf("After deletion at index 3: ");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Initial array: 1 2 3 4 5
After insertion at index 2: 1 2 10 3 4
After updating index 1: 1 20 10 3 4
After deletion at index 3: 1 20 10 4 0
```

Explanation: This program demonstrates traversing (printing all elements), insertion (adding 10 at index 2), modification (updating index 1 to 20), and deletion (removing the element at index 3).

Additional Knowledge: Linked Lists (Introduction)

Definition

- A **Linked List** is a linear data structure where elements are stored in **nodes**, not necessarily contiguous in memory.
- Each node contains:
  - **Data**: The element's value.
  - **Next Pointer**: The memory address of the next node (NULL for the last node).

Advantages over Arrays

- **Dynamic Size**: Can grow/shrink at runtime using dynamic memory allocation.
- **Efficient Insertion/Deletion**: No shifting required; only pointers are adjusted ($O(1)$ if position known, $O(n)$ for traversal).

Core Operations

- **Insertion**: Allocate a new node, adjust pointers.
- **Deletion**: Update the preceding node's pointer, free the deleted node's memory.
- **Traversal**: Visit each node sequentially ($O(n)$).

C Program Example: Basic Linked List Operations

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

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

void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

void traverse(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    printf("Linked List: ");
    traverse(head);
    return 0;
}
```

Output:

```
Linked List: 1 2 3
```

Explanation: This program creates a linked list, inserts elements at the end, and traverses it to print the values.

2-D Arrays

Definition

- A **2-D Array** (Two-Dimensional Array) is an array of arrays. It's a collection of elements of the same data type arranged in a grid-like structure with rows and columns, forming a matrix.
- Each element is identified by *two* indices: a row index and a column index (e.g., `arr[i][j]`). The first index `i` typically represents the row number, and the second index `j` represents the column number.

Representation

- **Conceptual View**: A 2-D array is visualized as a table or a matrix with rows and columns. This makes it easy to understand the organization of data.
- **Memory Representation**: Computers have linear memory (a sequence of memory locations). Therefore, 2-D arrays, despite their grid-like appearance, are stored linearly in memory. There are two primary methods for this linearization:
  1. **Row-Major Order**:
     - **Description**: Elements are stored row by row. All elements of the first row are stored contiguously, followed by all elements of the second row, and so on. This is the most common approach and is used by languages like C, C++, Java, and Python (NumPy).
     - **Example**: Consider the 2-D array: `int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};`
     - **Memory Layout (Row-Major)**: The elements would be stored in memory in the following order: `1 2 3 4 5 6`
     - **Address Calculation**: To access an element `arr[i][j]` in an `m x n` array (m rows, n columns), you need to know:
       - `base`: The base address of the array in memory (the address of the first element, `arr[0][0]`).
       - `size`: The size (in bytes) of each element (e.g., 4 bytes for an `int`).
       - `n`: The number of columns in the array.

       The formula to calculate the memory address of `arr[i][j]` is:

       `Address = base + (i * n + j) * size`

       Explanation of the Formula:
       - `i * n`: This part calculates the number of elements to skip to reach the beginning of the `i`-th row. Since each row has `n` elements, we skip `i` rows.
       - `i * n + j`: This adds the column index `j` to find the offset of the element within its row.
       - `(i * n + j) * size`: Finally, we multiply by the `size` of each element to get the byte offset from the base address.

C Example (Row-Major)

```c
#include <stdio.h>

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}}; // Declare and initialize a 2x3 integer array
    int rows = 2;  // Number of rows
    int cols = 3;  // Number of columns

    // Accessing elements
    printf("Element at arr[0][1]: %d\n", arr[0][1]); // Access and print the element at row 0, column 1 (output: 2)

    // Traversing (printing) the 2-D array
    printf("2-D Array:\n");
    for (int i = 0; i < rows; i++) { // Outer loop iterates through rows
        for (int j = 0; j < cols; j++) { // Inner loop iterates through columns
            printf("%d ", arr[i][j]); // Print the element at arr[i][j]
        }
        printf("\n"); // Print a newline after each row
    }

    return 0;
}
```

Output:

```
Element at arr[0][1]: 2
2-D Array:
1 2 3
4 5 6
```

Explanation of the Row-Major C Code:

1. `#include <stdio.h>`: Includes the standard input/output library for functions like `printf`.

2. `int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};`: Declares a 2-D array named `arr` of type `int`. It has 2 rows and 3 columns. The values are initialized directly.
3. `int rows = 2;` and `int cols = 3;`: Store the number of rows and columns for clarity and easier modification.
4. `printf("Element at arr[0][1]: %d\n", arr[0][1]);`: Demonstrates accessing a specific element. `arr[0][1]` accesses the element at the first row (index 0) and the second column (index 1), which is `2`.
5. `for (int i = 0; i < rows; i++) { ... }`: The outer loop iterates through each row of the array. `i` represents the row index.
6. `for (int j = 0; j < cols; j++) { ... }`: The inner loop iterates through each column *within the current row*. `j` represents the column index.
7. `printf("%d ", arr[i][j]);`: Inside the inner loop, this line prints the value of the element at the current row `i` and column `j`. A space is printed after each element for formatting.
8. `printf("\n");`: After the inner loop completes (i.e., after printing all elements of a row), this line prints a newline character to move to the next line for the next row.

---

2. **Column-Major Order**:
   - **Description**: Elements are stored column by column. All elements of the first column are stored contiguously, followed by all elements of the second column, and so on. This is used by languages like Fortran, MATLAB, R, and Julia.
   - **Example**: Using the same 2-D array:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

   - **Memory Layout (Column-Major)**: The elements would be stored in memory in the following order: `1 4 2 5 3 6`.
   - **Address Calculation**: For an element `arr[i][j]` in an `m x n` array, the formula is:

$$\text{Address} = \text{base} + (j \cdot m + i) \cdot \text{size}$$

     Explanation of the Formula:
     - $(j \cdot m)$: This calculates the number of elements to skip to reach the beginning of the $(j) - th$ column. Since each column has (m) elements, we skip (j) columns.
     - $(j \cdot m + i)$: This adds the row index (i) to find the offset of the element within its column.
     - $((j \cdot m + i) \cdot \text{size})$: This multiplies by the size of each element to get the byte offset.

C Example (Column-Major)

```c
#include <stdio.h>

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}}; // Same array as before
    int rows = 2;
    int cols = 3;

    // We'll simulate column-major access by manually calculating the memory addresses.
    // In a real column-major language, this would be handled by the compiler.

    printf("2-D Array (Column-Major Order):\n");
    for (int j = 0; j < cols; j++) { // Outer loop iterates through columns
        for (int i = 0; i < rows; i++) { // Inner loop iterates through rows

            // Calculate the address manually using the column-major formula
            int *element_ptr = (int *)arr + (j * rows + i);

            printf("%d ", *element_ptr); // Print the value at the calculated address
        }
        printf("\n"); // Newline after each column
    }

    return 0;
}
```

Output:

```
2-D Array (Column-Major Order):
1 4
2 5
3 6
```

Explanation of the Column-Major C Code:

1. `#include <stdio.h>`: Includes the standard input/output library.
2. `int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};`: Declares the same 2x3 array. *Important Note:* C uses row-major order. This example simulates column-major access to demonstrate the concept.
3. `int rows = 2;` and `int cols = 3;`: Store the dimensions.
4. `printf("2-D Array (Column-Major Order):\n");`: Prints a header.
5. `for (int j = 0; j < cols; j++) { ... }`: The *outer* loop now iterates through the *columns* first. `j` represents the column index.
6. `for (int i = 0; i < rows; i++) { ... }`: The *inner* loop iterates through the *rows* within the current column. `i` represents the row index.
7. `int *element_ptr = (int *)arr + (j * rows + i);`: This is the core of the column-major simulation.
   - `(int *)arr`: This casts the array name `arr` to a pointer to an integer (`int *`). In C, the name of an array decays to a pointer to its first element.
   - `+ (j * rows + i)`: This is the column-major address calculation. It's equivalent to `base + (j * m + i) * size`, where `base` is the address of `arr`, `m` is `rows`, `i` is the row index, `j` is the column index, and `size` is implicitly 1 because we're using pointer arithmetic with an `int *`.
   - `element_ptr`: This pointer now holds the calculated memory address of the element at `arr[i][j]` *as if* the array were stored in column-major order.
8. `printf("%d ", *element_ptr);`: This prints the value *pointed to* by `element_ptr`. The `*` operator dereferences the pointer, accessing the value at that memory address.
9. `printf("\n");`: Prints a newline after each column is printed.

Operations on 2-D Arrays

- **Traversal**: Visiting each element in the array. This typically involves nested loops (one for rows and one for columns).
  - **Time Complexity**: O(m*n), where 'm' is the number of rows and 'n' is the number of columns. You must visit every element.
- **Access**: Retrieving the value of an element at a specific row and column (e.g., `arr[i][j]`).
  - **Time Complexity**: O(1). Direct access to the element's memory location is possible using the address calculation formulas.
- **Update**: Modifying the value of an element at a specific row and column.
  - **Time Complexity**: O(1). Similar to access, updating involves direct memory access.
- **Insertion/Deletion**: 2-D arrays in C (and most languages) have a fixed size. True insertion or deletion (changing the dimensions of the array) is not directly supported. If you need to "insert" or "delete" conceptually, you have a few options:
  - **Create a New Array:** Allocate a new 2-D array with the desired dimensions and copy the relevant elements from the original array. This is the most common approach for resizing.
  - **Shifting Elements (Within Fixed Size):** If you have some "empty" space within your fixed-size array (e.g., you're using only a portion of it), you can shift elements to simulate insertion or deletion. This is analogous to 1-D array insertion/deletion and has a time complexity of O(m*n) in the worst case, as you might need to shift a large number of elements.
  - **Use a Different Data Structure:** If you need frequent insertions and deletions that change the size, a 2-D array might not be the best choice. Consider using a more dynamic data structure (e.g., a list of lists, or a custom data structure).

Advantages and Disadvantages of 2-D Arrays

Feature	Advantages	Disadvantages
Access	Fast and direct access to elements using indices (O(1)).	Fixed size; resizing is expensive (requires creating a new array and copying elements).

Feature	Advantages	Disadvantages
Memory	Efficient memory usage for dense matrices (where most elements have non-zero values). Contiguous memory allocation can lead to better cache performance.	Can be inefficient for sparse matrices (where most elements are zero).
Simplicity	Easy to understand and implement. The grid-like structure is intuitive.	Insertion and deletion (changing dimensions) are not directly supported; you need to create a new array or shift elements within the existing array, which can be inefficient.
Use Cases	Suitable for representing matrices, grids, images, tables, game boards, and other data that naturally has a two-dimensional structure.	Not ideal for situations requiring frequent resizing or where the data is very sparse.
Locality	Row Major has good spatial locality.	Resizing is an O(m*n) operation.

Searching

Need for Searching

- **Fundamental Operation**: Searching is one of the most fundamental and frequently performed operations in computer science.
- **Finding Information**: The primary goal is to locate a specific element (the "target" or "key") within a collection of data (e.g., an array, list, database).
- **Ubiquitous**: Searching is essential in a vast range of applications:
  - **Databases**: Finding records that match specific criteria.
  - **Information Retrieval**: Searching for documents, web pages, or other information based on keywords.
  - **Operating Systems**: Finding files, processes, or resources.
  - **Compilers**: Looking up symbols in symbol tables.
  - **Games**: Finding game objects, paths, or solutions.

Types of Searching

---

1. **Linear Search**

- **Algorithm**: Linear search, also known as sequential search, is the simplest searching algorithm. It works by sequentially checking each element in the data structure (typically an array or list) until the target element is found or the end of the data structure is reached.
- **Process**:
  1. Start at the beginning of the array (or list).
  2. Compare the current element with the target element.
  3. If they match, the search is successful; return the index (or position) of the element.
  4. If they don't match, move to the next element.
  5. Repeat steps 2-4 until either the target is found or the end of the array is reached.
  6. If the end is reached without finding the target, the search is unsuccessful; return an indicator (e.g., -1) to signify that the element is not present.
- **Time Complexity**:
  - **Best Case**: O(1). This occurs when the target element is the first element in the array.
  - **Average Case**: O(n). On average, you'll need to examine half of the elements in the array (assuming the target is equally likely to be at any position).
  - **Worst Case**: O(n). This occurs when the target element is the last element in the array or is not present at all. You need to examine every element.
- **Advantages**:
  - **Simple**: Easy to understand and implement.

- **No Sorting Required**: Works on unsorted data.

- **Disadvantages**:
  - **Inefficient**: Can be very slow for large datasets due to its O(n) time complexity.
- **C Example**:

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {
    // arr[]: The array to search in
    // n: The number of elements in the array
    // target: The element to search for

    for (int i = 0; i < n; i++) { // Iterate through the array from index 0 to n-1
        if (arr[i] == target) { // Compare the current element (arr[i]) with the target
            return i; // If they match, return the current index (i) - element found!
        }
    }
    return -1; // If the loop completes without finding the target, return -1 (element not found)
}

int main() {
    int arr[] = {2, 5, 1, 8, 4, 6, 9, 3, 7}; // Example unsorted array
    int n = sizeof(arr) / sizeof(arr[0]);  // Calculate the number of elements in the array
    int target = 8; // The element we want to find
    int index = linearSearch(arr, n, target); // Call the linearSearch function

    if (index != -1) { // Check if the element was found (index is not -1)
        printf("Element found at index: %d\n", index); // Print the index where the element was found
    } else {
        printf("Element not found.\n"); // Print a message if the element was not found
    }
    return 0;
}
```

Explanation of the Linear Search C Code:

1. `int linearSearch(int arr[], int n, int target)`: This defines the `linearSearch` function. It takes three arguments:
   - `arr[]`: The integer array to search within.
   - `n`: The number of elements in the array.
   - `target`: The integer value to search for.
2. `for (int i = 0; i < n; i++) { ... }`: This `for` loop iterates through each element of the array.
   - `int i = 0;`: Initializes the loop counter `i` to 0 (the index of the first element).
   - `i < n;`: The loop continues as long as `i` is less than `n` (the number of elements).
   - `i++;`: Increments `i` by 1 after each iteration, moving to the next element.
3. `if (arr[i] == target) { ... }`: Inside the loop, this `if` statement checks if the current element `arr[i]` is equal to the `target` value.
4. `return i;`: If the `if` condition is true (the element is found), the function immediately returns the current index `i`. This indicates the position of the target element in the array.
5. `return -1;`: If the loop completes without finding the `target` (the `if` condition was never true), the function returns -1. This is a common convention to indicate that the element was not found in the array.
6. `int main() { ... }`: This part shows how the function would be used.

2. **Binary Search**
   - **Requirement**: Binary search *requires* the input array to be sorted (either in ascending or descending order). This is crucial for the algorithm to work.
   - **Algorithm**: Binary search is a much more efficient algorithm than linear search for sorted data. It follows a "divide and conquer" strategy:
     1. **Start with the entire array**: Define a search interval that initially covers the entire array (from index `left` to index `right`).
     2. **Find the middle element**: Calculate the middle index: `mid = left + (right - left) / 2`. (This formula is preferred over `(left + right) / 2` to prevent potential integer overflow for very large arrays).
     3. **Compare**: Compare the target value with the element at the middle index (`arr[mid]`).
     4. **Three Possible Cases**:
        - **Match**: If `arr[mid]` is equal to the target, the search is successful; return `mid`.
        - **Target is Smaller**: If the target is *less than* `arr[mid]`, the target *must* be in the left half of the array (if it exists at all). Recursively (or iteratively) search the left half: `left` remains the same, `right` becomes `mid - 1`.

        - **Target is Larger**: If the target is *greater than* `arr[mid]`, the target *must* be in the right half of the array. Recursively (or iteratively) search the right half: `left` becomes `mid + 1`, `right` remains the same.
    5. **Repeat**: Continue steps 2-4, narrowing the search interval in each step.
    6. **Not Found**: If the search interval becomes empty (`left > right`), the target is not present in the array; return -1.
- **Time Complexity**:
    - **Best Case**: O(1). This occurs when the target element is the middle element in the very first comparison.
    - **Average Case**: O(log n). With each comparison, the search space is halved. This logarithmic behavior makes binary search extremely efficient for large, sorted datasets.
    - **Worst Case**: O(log n). Even in the worst case, the number of comparisons grows logarithmically with the size of the array.
- **Advantages**:
    - **Very Efficient**: Much faster than linear search for large, sorted arrays (O(log n) vs. O(n)).
- **Disadvantages**:
    - **Requires Sorted Data**: The array must be sorted before applying binary search. If the data is not sorted, you'll need to sort it first, which adds to the overall time complexity.
- **C Example**:

```c
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) {
    // arr[]: The sorted array to search in
    // left: The starting index of the search interval
    // right: The ending index of the search interval
    // target: The element to search for

    while (left <= right) { // Continue searching as long as the left index is less than or equal to the right index
        int mid = left + (right - left) / 2; // Calculate the middle index (prevents potential overflow)

        if (arr[mid] == target) { // Check if the middle element is the target
            return mid; // If it is, return the middle index (element found!)
        }

        if (arr[mid] < target) { // If the middle element is less than the target,
            left = mid + 1;  //  search in the right half of the array
        } else {                // Otherwise (if the middle element is greater than the target),
            right = mid - 1; //  search in the left half of the array
        }
    }
    return -1; // If the loop completes without finding the target, return -1 (element not found)
}

int main() {
    int arr[] = {1, 2, 4, 5, 8, 9, 12, 15}; // Example sorted array
    int n = sizeof(arr) / sizeof(arr[0]);    // Calculate the number of elements
    int target = 9;                          // The element to search for
    int index = binarySearch(arr, 0, n - 1, target); // Call binarySearch (start with the entire array: left=0, right=n-1)

    if (index != -1) {
        printf("Element found at index: %d\n", index);
    } else {
        printf("Element not found.\n");
    }
    return 0;
}
```

Explanation of the Binary Search C Code:

1. `int binarySearch(int arr[], int left, int right, int target)`: This defines the `binarySearch` function. It takes four arguments:
    - `arr[]`: The *sorted* integer array.
    - `left`: The starting index of the current search interval.
    - `right`: The ending index of the current search interval.
    - `target`: The integer value to search for.
2. `while (left <= right) { ... }`: This `while` loop is the core of the binary search. It continues as long as the `left` index is less than or equal to the `right` index. If `left` becomes greater than `right`, it means the search interval is empty, and the

target is not present.

3. `int mid = left + (right - left) / 2;`: This line calculates the middle index of the current search interval. This formula is safer than `(left + right) / 2` because it avoids potential integer overflow if `left` and `right` are very large numbers.
4. `if (arr[mid] == target) { return mid; }`: This checks if the element at the middle index `arr[mid]` is equal to the `target`. If it is, the target is found, and the function returns `mid` (the index).
5. `if (arr[mid] < target) { left = mid + 1; }`: If the middle element is *less than* the target, the target must be in the *right half* of the interval. We update `left` to `mid + 1`, effectively discarding the left half.
6. `else { right = mid - 1; }`: If the middle element is *greater than* the target, the target must be in the *left half* of the interval. We update `right` to `mid - 1`, discarding the right half.
7. `return -1;`: If the `while` loop completes without finding the target (i.e., `left` becomes greater than `right`), the function returns -1, indicating that the target is not present in the array.

Sorting

Need for Sorting

- **Organization**: Sorting arranges data elements in a specific order (ascending or descending). This organization is fundamental for many computer science tasks.
- **Facilitates Searching**: Sorted data enables the use of efficient search algorithms like binary search (O(log n) time complexity). Searching unsorted data requires linear search (O(n)), which is much slower for large datasets.
- **Improves Efficiency**: Many algorithms and data structures rely on sorted data for optimal performance. Examples include:
  - **Finding the median or other statistical measures.**
  - **Detecting duplicates in a dataset.**
  - **Implementing priority queues.**
  - **Performing efficient data retrieval in databases.**
  - **Merging datasets.**
  - **Graph algorithms.**

Types of Sorting

---

1. **Selection Sort**

   - **Algorithm**: Selection sort is an in-place comparison-based sorting algorithm. It works by repeatedly finding the minimum element (or maximum, for descending order) from the unsorted portion of the array and swapping it with the element at the beginning of the unsorted portion. The algorithm effectively divides the array into two parts:
     - A **sorted subarray**, which is built up from the beginning of the array.
     - An **unsorted subarray**, which contains the remaining elements.
   - **Process**:
     1. **Find the minimum element**: Iterate through the unsorted subarray to find the minimum element.
     2. **Swap**: Swap the minimum element with the first element of the unsorted subarray.
     3. **Expand the sorted subarray**: The first element of the unsorted subarray is now considered part of the sorted subarray.
     4. **Repeat**: Repeat steps 1-3 until the entire array is sorted.
   - **Time Complexity**:
     - **Best Case**: O(n^2). Even if the array is already sorted, selection sort still performs the same number of comparisons.
     - **Average Case**: O(n^2).
     - **Worst Case**: O(n^2). Selection sort has quadratic time complexity in all cases, making it inefficient for large datasets.
   - **Advantages**:
     - **Simple**: Easy to understand and implement.
     - **In-place**: Sorts the array without requiring significant extra memory (only a few temporary variables are needed).
     - Performs well on small lists.
   - **Disadvantages**:

- **Inefficient**: O(n^2) time complexity makes it very slow for large arrays.

- **C Example**:

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    // arr[]: The array to be sorted
    // n: The number of elements in the array

    for (int i = 0; i < n - 1; i++) { // Outer loop: Iterate through the array (n-1) times
        // i represents the starting index of the unsorted subarray

        int min_idx = i; // Assume the current element (arr[i]) is the minimum
        for (int j = i + 1; j < n; j++) { // Inner loop: Find the minimum element in the unsorted
subarray
            if (arr[j] < arr[min_idx]) { // Compare the current element (arr[j]) with the current
minimum (arr[min_idx])
                min_idx = j;         // If arr[j] is smaller, update min_idx to the index of the new
minimum
            }
        }

        // Swap arr[i] (the first element of the unsorted subarray) with arr[min_idx] (the actual
minimum)
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11, 1, 8, 54, 23, 101, 99, 15}; // Example unsorted array
    int n = sizeof(arr) / sizeof(arr[0]); // Calculate the number of elements
    selectionSort(arr, n);              // Call the selectionSort function to sort the array
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) { // Print the sorted array
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Explanation Selection Sort C Program:

1. `void selectionSort(int arr[], int n)`:
   - Defines the `selectionSort` function. It takes the array `arr` and its size `n` as input. The function does not return a value (void)
2. `for (int i = 0; i < n - 1; i++) { ... }`:
   - This is the outer loop. It iterates from `i = 0` to `n - 2`. The outer loop controls the boundary between the sorted and unsorted portions of the array.
   - We iterate till n-1 because, the last element will automatically be sorted.
3. `int min_idx = i;`:
   - Inside the outer loop, we initialize `min_idx` to `i`. This variable will store the *index* of the minimum element found in the unsorted portion. We initially *assume* the current element at index `i` is the minimum.
4. `for (int j = i + 1; j < n; j++) { ... }`:
   - This is the inner loop. It iterates through the *unsorted* portion of the array (from `i + 1` to the end).
5. `if (arr[j] < arr[min_idx]) { min_idx = j; }`:
   - Inside the inner loop, this `if` statement compares the current element `arr[j]` with the element at the current `min_idx`. If `arr[j]` is smaller, we've found a new minimum, so we update `min_idx` to `j`.
6. `int temp = arr[min_idx];`, `arr[min_idx] = arr[i];`, `arr[i] = temp;`:
   - After the inner loop completes, `min_idx` holds the index of the smallest element in the unsorted portion. These three lines *swap* the element at `arr[i]` (the beginning of the unsorted portion) with the element at `arr[min_idx]` (the actual minimum). This places the minimum element at its correct sorted position.

2. **Insertion Sort**

- **Algorithm**: Insertion sort is another in-place comparison-based sorting algorithm. It works similarly to how you might sort playing cards in your hand. It maintains a sorted subarray at the beginning of the array and iteratively inserts elements from the unsorted portion into their correct position within the sorted subarray.

- **Process**:
    1. **Start with the second element**: Assume the first element is already sorted (a subarray of size 1).
    2. **Pick the next element**: Take the next element from the unsorted portion (this is the 'key' element to be inserted).
    3. **Compare and shift**: Compare the 'key' element with the elements in the sorted subarray, moving from right to left. Shift elements in the sorted subarray that are *greater* than the 'key' element one position to the right. This creates space for the 'key' to be inserted.
    4. **Insert**: Insert the 'key' element into its correct, sorted position in the sorted subarray.
    5. **Repeat**: Repeat steps 2-4 until all elements from the unsorted portion have been inserted into the sorted subarray.
- **Time Complexity**:
    - **Best Case**: O(n). This occurs when the array is already sorted. In this case, the inner loop (the comparison and shifting step) never executes, as each element is already in its correct position.
    - **Average Case**: O(n^2). For randomly ordered data, insertion sort typically requires quadratic time.
    - **Worst Case**: O(n^2). This occurs when the array is sorted in reverse order. In this case, the inner loop has to shift all elements of the sorted subarray for each insertion.
- **Advantages**:
    - **Simple**: Easy to understand and implement.
    - **Efficient for small datasets**: Performs well for small arrays or nearly sorted arrays.
    - **In-place**: Sorts the array in place, requiring only a constant amount of extra memory (O(1) space complexity).
    - **Adaptive**: Efficient for data that is already substantially sorted; the time complexity approaches O(n) as the data becomes more sorted.
    - **Stable**: Maintains the relative order of equal elements.
- **Disadvantages**:
    - **Inefficient for large datasets**: The O(n^2) average and worst-case time complexity makes it slow for large arrays.
- **C Example**:

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    // arr[]: The array to be sorted
    // n: The number of elements in the array
    int i, key, j;
    for (i = 1; i < n; i++) { // Iterate from the second element (index 1) to the end
        key = arr[i];       // 'key' is the element to be inserted into the sorted subarray
        j = i - 1;          // 'j' is the index of the last element in the sorted subarray

        // Move elements of arr[0..i-1] (the sorted subarray) that are greater than 'key'
        // to one position ahead of their current position.
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // Shift the element to the right
            j = j - 1;          // Move to the next element in the sorted subarray (towards the left)
        }
        arr[j + 1] = key; // Insert 'key' into its correct sorted position
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 1, 9, 8, 55, 2}; // Example unsorted array
    int n = sizeof(arr) / sizeof(arr[0]);           // Calculate the number of elements

    insertionSort(arr, n);                          // Call insertionSort to sort the array

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

    return 0;
}
```

Explanation of the Insertion Sort C Code:

1. `void insertionSort(int arr[], int n)`:
    - Defines the `insertionSort` function, taking the array `arr` and its size `n` as input.
2. `for (i = 1; i < n; i++) { ... }`:
    - The outer loop starts at index 1 (the second element). We assume the first element (at index 0) is already a sorted subarray of size 1.
3. `key = arr[i];`:

- `key` stores the value of the current element to be inserted into the sorted subarray. This is the element we're working with in this iteration of the outer loop.

4. `j = i - 1;`:
   - `j` is initialized to the index of the last element *in the sorted subarray*.
5. `while (j >= 0 && arr[j] > key) { ... }`:
   - This `while` loop is the core of the insertion process. It does two things:
     - `j >= 0`: Checks if we've reached the beginning of the array (we don't want to go out of bounds).
     - `arr[j] > key`: Checks if the current element in the sorted subarray (`arr[j]`) is *greater* than the `key`. If it is, we need to shift `arr[j]` to the right to make space for `key`.
   - `arr[j + 1] = arr[j];`: Shifts the element at `arr[j]` one position to the right (to `arr[j + 1]`).
   - `j = j - 1;`: Decrements `j` to move to the next element to the left in the sorted subarray.
6. `arr[j + 1] = key;`:
   - After the `while` loop finishes, `j` will either be -1 (meaning `key` is smaller than all elements in the sorted subarray) or `j` will be the index where `key` should be inserted. We insert `key` at `j + 1`.

Unit 3: Stack & Queue

Stack

Introduction

- **Definition**: A Stack is a linear data structure that follows the **LIFO (Last-In, First-Out)** principle. This means that the last element added to the stack is the first element to be removed. It's like a stack of plates or a stack of books – you add items to the top and remove items from the top.
- **Analogy**: The most common analogy is a stack of plates. You can only add a new plate to the top of the stack, and you can only remove a plate from the top. You can't access the plates in the middle without first removing the ones on top. Other analogies include:
  - A stack of books.
  - A Pez dispenser.
  - The "back" button in a web browser (it takes you to the last page you visited).

Operations on Stack

- **Push**: Adds an element to the *top* of the stack.
  - Time Complexity: O(1) - Constant time, as it only involves updating the top pointer.
- **Pop**: Removes and returns the element from the *top* of the stack. If the stack is empty, this is called an "underflow" condition.
  - Time Complexity: O(1)
- **Peek/Top**: Returns the value of the top element *without* removing it. Useful for inspecting the top element. If the stack is empty, this operation is usually undefined or returns a special value (like NULL or -1).
  - Time Complexity: O(1)
- **isEmpty**: Checks if the stack is empty. Returns `true` if the stack is empty, `false` otherwise.
  - Time Complexity: O(1)
- **isFull**: Checks if the stack is full. This is relevant when the stack has a fixed size (e.g., when implemented using an array). If the stack is implemented using a linked list, it's typically considered to be dynamically sized and doesn't have a "full" state in the same way.
  - Time Complexity: O(1)

Stack Implementation (Array)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h> // For using 'bool', 'true', and 'false'

#define MAX_SIZE 10 // Define a maximum size for the stack

typedef struct {
    int items[MAX_SIZE]; // Array to store the stack elements
    int top;             // Index of the top element
} Stack;

// Initialize the stack
void initialize(Stack *s) {
    s->top = -1; // -1 indicates an empty stack (no elements yet)
}

// Check if the stack is empty
bool isEmpty(Stack *s) {
    return s->top == -1; // Returns 'true' if top is -1, 'false' otherwise
}

// Check if the stack is full
bool isFull(Stack *s) {
    return s->top == MAX_SIZE - 1; // Returns 'true' if top is at the last valid index, 'false' otherwise
}

// Push an element onto the stack
void push(Stack *s, int value) {
    if (isFull(s)) {        // Check for stack overflow
        printf("Stack Overflow!\n");
        return;             // Don't push if the stack is full
    }
    s->top++;              // Increment top (move it to the next available position)
    s->items[s->top] = value; // Store the value at the new top position
}

// Pop an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {       // Check for stack underflow
        printf("Stack Underflow!\n");
        return -1;          // Return -1 (or another error indicator) if the stack is empty
    }
    int value = s->items[s->top]; // Get the value at the top
    s->top--;                     // Decrement top (move it down)
    return value;                 // Return the popped value
}

// Peek at the top element without removing it
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return -1;
    }
    return s->items[s->top]; // Return the value at the top
}

int main() {
    Stack s;            // Declare a stack variable
    initialize(&s);   // Initialize the stack

    push(&s, 10);     // Push some values onto the stack
    push(&s, 20);
    push(&s, 30);

    printf("Top element: %d\n", peek(&s)); // Peek at the top element (should be 30)

    printf("Popped: %d\n", pop(&s));    // Pop the top element (30)
    printf("Popped: %d\n", pop(&s));    // Pop the next element (20)

    printf("Top element: %d\n", peek(&s)); // Peek again (should be 10)

    push(&s, 40);
```

```
    printf("Popped: %d\n", pop(&s));
    printf("Popped: %d\n", pop(&s));
    printf("Popped: %d\n", pop(&s));

    return 0;
}
```

Explanation of the Stack C Code:

1. `#include <stdio.h>`, `#include <stdlib.h>`, `#include <stdbool.h>`: Include necessary header files. `stdbool.h` is for using the `bool` type (true/false).
2. `#define MAX_SIZE 10`: Defines a constant `MAX_SIZE` to represent the maximum capacity of the stack (10 elements in this case). This is for the array-based implementation.
3. `typedef struct { ... } Stack;`: Defines a structure named `Stack` to represent the stack. It contains:
   - `int items[MAX_SIZE];`: An array of integers to hold the stack elements.
   - `int top;`: An integer variable `top` that stores the *index* of the top element in the `items` array.
4. `void initialize(Stack *s)`:
   - Initializes the stack. It sets `s->top` to -1. The value -1 is used to indicate that the stack is initially empty.
5. `bool isEmpty(Stack *s)`:
   - Checks if the stack is empty. It returns `true` if `s->top` is -1 (empty), and `false` otherwise.
6. `bool isFull(Stack *s)`:
   - Checks if the stack is full. It returns `true` if `s->top` is equal to `MAX_SIZE - 1` (the last valid index in the `items` array), and `false` otherwise.
7. `void push(Stack *s, int value)`:
   - Adds an element (`value`) to the top of the stack.
   - `if (isFull(s)) { ... }`: Checks for *stack overflow* (trying to push onto a full stack). If the stack is full, it prints an error message and returns without pushing.
   - `s->top++;`: Increments the `top` index to point to the next available position in the array.
   - `s->items[s->top] = value;`: Stores the `value` at the new `top` position in the `items` array.
8. `int pop(Stack *s)`:
   - Removes and returns the element at the top of the stack.
   - `if (isEmpty(s)) { ... }`: Checks for *stack underflow* (trying to pop from an empty stack). If the stack is empty, it prints an error message and returns -1 (as an error indicator).
   - `int value = s->items[s->top];`: Stores the value of the top element in the `value` variable.
   - `s->top--;`: Decrements the `top` index, effectively removing the top element.
   - `return value;`: Returns the value that was removed.
9. `int peek(Stack *s)`:
   - Returns the value of the top element *without* removing it.
   - `if (isEmpty(s)) { ... }`: Checks if the stack is empty. If it is, it prints an error message and returns -1.
   - `return s->items[s->top];`: Returns the value at the current `top` index.
10. `int main() { ... }`:
    - Demonstrates the usage of the stack functions.

Applications of Stacks

- **Function Call Stack**: When a function is called in a program, its information (local variables, return address, etc.) is pushed onto the *call stack*. When the function returns, its information is popped off the stack. This LIFO mechanism ensures that functions return in the correct order. This is fundamental to how most programming languages work.
- **Expression Evaluation**: Stacks are used to evaluate arithmetic expressions, especially those involving parentheses and operator precedence. Infix, postfix, and prefix notations are related to this.
- **Backtracking Algorithms**: In algorithms that involve exploring multiple possibilities (like searching a maze or solving a puzzle), stacks can be used to keep track of the path taken and to backtrack when a dead end is reached. Depth-first search (DFS) is a classic example.
- **Undo/Redo Functionality**: In text editors, graphics programs, and other applications, stacks can store the history of actions. "Undo" pops the last action off the stack, and "Redo" pushes it back on.
- **Parenthesis Matching**: Stacks can be used to check if parentheses (or other delimiters like brackets and braces) in an expression are balanced. For each opening parenthesis, you push it onto the stack. For each closing parenthesis, you pop an

opening parenthesis from the stack. If the stack is empty at the end, and you never try to pop from an empty stack, the parentheses are balanced.
- **Web Browser History**: The history of the pages visited are managed using Stack.

Expression Notation

Types of Notation

---

1. **Infix Notation**:
   - **Definition**: The standard way we write mathematical expressions. The operator is placed *between* its operands.
   - **Example**: `A + B`, `(A + B) * C`, `A - B / (C + D)`
   - **Characteristics**:
     - Requires parentheses to indicate the order of operations (precedence) when the default precedence rules are not sufficient.
     - Familiar and easy to read for humans.
   - **Advantages**:
     - Intuitiveness for human readability.
   - **Disadvantages**:
     - Requires rules for precedence and associativity, and often requires parentheses, making parsing more complex for computers.
2. **Prefix Notation (Polish Notation)**:
   - **Definition**: The operator is placed *before* its operands.
   - **Example**: `+ A B`, `* + A B C`, `- A / B + C D`
   - **Characteristics**:
     - Does *not* require parentheses to specify precedence. The order of operations is unambiguous.
     - Developed by the Polish logician Jan Łukasiewicz.
   - **Advantages:**
     - No need for parentheses or precedence rules, simplifying parsing.
     - Easier to evaluate using a stack.
   - **Disadvantages:**
   - Less natural than infix for human reading, require some practice.
3. **Postfix Notation (Reverse Polish Notation)**:
   - **Definition**: The operator is placed *after* its operands.
   - **Example**: `A B +`, `A B + C *`, `A B C D + / -`
   - **Characteristics**:
     - Does *not* require parentheses.
     - Used in some calculators (especially HP calculators) and in programming languages like Forth and PostScript.
   - **Advantages:**
     - No need for parenthesis or precedence rules, simplifying parsing.
     - Very easy to evaluate using a stack.
   - **Disadvantages:**
     - Less natural than infix for human reading, require some practice.

Conversion of Expressions

- Converting between these notations is a classic application of stacks. The most common conversion is from infix to postfix (or prefix), as postfix and prefix are easier for computers to evaluate.

Infix to Postfix Conversion (Using a Stack)

1. **Initialize**: Create an empty stack (to hold operators) and an empty output string (or list).
2. **Scan**: Scan the infix expression from left to right, one token (operand, operator, or parenthesis) at a time.
3. **Process Each Token**:
   - **Operand**: If the token is an operand (a variable or a number), append it directly to the output string.
   - **Opening Parenthesis '(':** Push it onto the stack.
   - **Closing Parenthesis ')':**
     - Pop operators from the stack and append them to the output string *until* you encounter an opening parenthesis '('.
     - Discard both the opening and closing parentheses (don't add them to the output).
   - **Operator**:
     - While the stack is *not* empty AND the operator at the top of the stack has *equal or higher precedence* than the current operator:
       - Pop the operator from the stack and append it to the output string.
     - Push the current operator onto the stack. (Note: You need to define the precedence of operators beforehand. For example, `*` and `/` have higher precedence than `+` and `-`.)
4. **End of Expression**: After scanning the entire infix expression, pop any remaining operators from the stack and append them to the output string.

Example: A + B *C - D* -> *A B C* + D -

Token	Stack (Bottom to Top)	Output	Explanation
A		A	Operand: Append to output.
+	+	A	Operator: Push onto stack.
B	+	A B	Operand: Append to output.
*	+ *	A B	Operator: `*` has higher precedence than `+`, so push `*` onto the stack.
C	+ *	A B C	Operand: Append to output.
-	-	A B C * +	Operator: `-` has lower precedence than `*`, so pop `*` and append to output. `+` has equal/higher precedence to `-`, so pop `+` and append it to the output. Then, push `-` onto stack.
D	-	A B C * + D	Operand: Append to output.
		A B C * + D -	End of expression: Pop remaining operators from the stack and append to output.

Prefix to Infix

To convert a prefix expression to an infix expression, we use a stack. The algorithm is as follows:

1. **Read**: Read the prefix expression from *right to left* (opposite of the usual left-to-right scan).
2. **Initialize**: Create an empty stack.
3. **Process Each Symbol**:
   - **Operand**: If the symbol is an operand, push it onto the stack.
   - **Operator**: If the symbol is an operator:
     - Pop *two* operands from the stack (let's call them `operand1` and `operand2`). *Important*: The order matters. The first operand popped is the *right* operand, and the second operand popped is the *left* operand.
     - Combine the two operands with the operator, enclosing the entire expression in parentheses: `(operand2 operator operand1)`.
     - Push the resulting string back onto the stack.
4. **Result**: When you've processed the entire prefix expression, the stack will contain a single string, which is the equivalent infix expression.

Example:

```
Prefix expression: * + A B - C D
```

Symbol	Stack (Bottom to Top)	Explanation
D	D	Operand: Push D
C	C, D	Operand: Push C
-	(C - D)	Operator: Pop C, D; Combine: (C - D); Push (C - D)
B	B, (C - D)	Operand: Push B
A	A, B, (C - D)	Operand: Push A
+	(A + B), (C - D)	Operator: Pop A, B; Combine: (A + B); Push (A + B)
*	((A + B) * (C - D))	Operator: Pop (A + B), (C - D); Combine: ((A + B) *(C - D)); Push ((A + B)* (C - D))

Resulting infix expression: `((A + B) * (C - D))`

Queue

Introduction

- **Definition**: A Queue is a linear data structure that follows the **FIFO (First-In, First-Out)** principle. This means that the first element added to the queue is the first element to be removed. It's like a line of people waiting for a service – the person who arrives first is served first.
- **Analogy**: The most common analogy is a queue of people waiting in line (e.g., at a bank, a store, or a bus stop). Other analogies include:
  - Print jobs waiting to be printed.
  - Tasks waiting to be processed by a CPU.
  - Data packets waiting to be transmitted over a network.

Operations on Queue

- **Enqueue**: Adds an element to the *rear* (end) of the queue.
  - Time Complexity: O(1)
- **Dequeue**: Removes and returns the element from the *front* of the queue. If the queue is empty, this is called an "underflow" condition.
  - Time Complexity: O(1)
- **Front/Peek**: Returns the value of the front element *without* removing it. Useful for inspecting the element at the front.
  - Time Complexity: O(1)
- **Rear**: Returns the value of the rear element without removing it. (Less commonly used than `front`, but sometimes helpful.)
  - Time Complexity: O(1)
- **isEmpty**: Checks if the queue is empty. Returns `true` if empty, `false` otherwise.
  - Time Complexity: O(1)
- **isFull**: Checks if the queue is full. Relevant for fixed-size queues (e.g., array-based implementations).
  - Time Complexity: O(1)

Queue Implementation (Array)

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

#define MAX_SIZE 10 // Maximum size of the queue

typedef struct {
```

```c
    int items[MAX_SIZE]; // Array to store queue elements
    int front;           // Index of the front element
    int rear;            // Index of the rear element
} Queue;

// Initialize the queue
void initialize(Queue *q) {
    q->front = -1; // -1 indicates an empty queue
    q->rear = -1;
}

// Check if the queue is empty
bool isEmpty(Queue *q) {
    return q->front == -1;
}

// Check if the queue is full (Circular Queue implementation)
bool isFull(Queue *q) {
   return (q->rear + 1) % MAX_SIZE == q->front;
   // This checks if the next position for rear is the same as front, which is the condition of full in
Circular Queue.
}

// Add an element to the rear of the queue (Enqueue)
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow!\n"); // Queue is full
        return;
    }
    if (isEmpty(q)) {
        q->front = 0; // If the queue was empty, set front to 0
    }
    q->rear = (q->rear + 1) % MAX_SIZE; // Circular increment: Move rear to the next position (wrap around
if necessary)
    q->items[q->rear] = value;         // Store the value at the new rear position
}

// Remove an element from the front of the queue (Dequeue)
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow!\n"); // Queue is empty
        return -1;                    // Return -1 (or another error indicator)
    }
    int value = q->items[q->front];    // Get the value at the front
    if (q->front == q->rear) {        // If there was only one element in the queue
        initialize(q);               // Reset the queue to empty
    } else {
        q->front = (q->front + 1) % MAX_SIZE; // Circular increment: Move front to the next position (wrap
around if necessary)
    }
    return value;                     // Return the dequeued value
}

// Get the value of the front element without removing it
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    return q->items[q->front]; // Return the value at the front
}

int main() {
    Queue q;
    initialize(&q); // Initialize the queue

    enqueue(&q, 10); // Enqueue some elements
    enqueue(&q, 20);
    enqueue(&q, 30);

    printf("Front element: %d\n", front(&q)); // Peek at the front (should be 10)

    printf("Dequeued: %d\n", dequeue(&q)); // Dequeue (removes 10)
    printf("Dequeued: %d\n", dequeue(&q)); // Dequeue (removes 20)
```

```c
    printf("Front element: %d\n", front(&q)); // Peek again (should be 30)

    enqueue(&q, 40);
    enqueue(&q, 50);

    printf("Dequeued: %d\n", dequeue(&q)); // Output: 30
    printf("Dequeued: %d\n", dequeue(&q)); // Output: 40
    printf("Front element: %d\n", front(&q)); // Output: 50
    return 0;
}
```

Explanation of the Queue C Code (Circular Queue):

1. `#include <stdio.h>`, `#include <stdlib.h>`, `#include <stdbool.h>`: Includes headers.
2. `#define MAX_SIZE 10`: Defines the maximum queue size.
3. `typedef struct { ... } Queue;`: Defines the `Queue` structure:
   - `int items[MAX_SIZE];`: The array to hold elements.
   - `int front;`: Index of the front element.
   - `int rear;`: Index of the rear element.
4. `void initialize(Queue *q)`:
   - Initializes `front` and `rear` to -1, indicating an empty queue.
5. `bool isEmpty(Queue *q)`:
   - Returns `true` if `front` is -1 (empty queue), `false` otherwise.
6. `bool isFull(Queue *q)`:
   - This implements the *circular queue* logic for checking fullness.
   - `return (q->rear + 1) % MAX_SIZE == q->front;`: This is the key to the circular queue. It checks if the position *after* the current `rear` would be the same as `front`. If they are the same, the queue is full. The modulo operator (`%`) handles the wrap-around.
7. `void enqueue(Queue *q, int value)`:
   - Adds an element to the rear.
   - `if (isFull(q)) { ... }`: Checks for overflow.
   - `if (isEmpty(q)) { q->front = 0; }`: If the queue was empty, sets `front` to 0 (the first element will be at index 0).
   - `q->rear = (q->rear + 1) % MAX_SIZE;`: This is the circular increment for `rear`. It moves `rear` to the next position, wrapping around to 0 if `rear` reaches the end of the array.
   - `q->items[q->rear] = value;`: Stores the `value` at the new `rear` position.
8. `int dequeue(Queue *q)`:
   - Removes and returns the element at the front.
   - `if (isEmpty(q)) { ... }`: Checks for underflow.
   - `int value = q->items[q->front];`: Gets the value at the front.
   - `if (q->front == q->rear) { initialize(q); }`: If there was only one element in the queue (front and rear were at the same position), we reset the queue to empty by calling `initialize()`.
   - `else { q->front = (q->front + 1) % MAX_SIZE; }`: This is the circular increment for `front`. It moves `front` to the next position, wrapping around if necessary.
   - `return value;`: Returns the dequeued value.
9. `int front(Queue *q)`:
   - Returns the value at the front without removing it.
   - Checks for an empty queue.
10. `int main() { ... }`: Demonstrates the usage.

Types of Queues

---

1. **Circular Queue**:
   - **Definition**: A circular queue is a variation of a queue where the rear and front can "wrap around" to the beginning of the array. This avoids wasted space that can occur in a simple linear queue when elements are dequeued from the front.
   - **Advantages**: More efficient use of memory compared to a simple linear queue, especially when there are frequent enqueue and dequeue operations.
   - **Implementation**: The C code example above demonstrates a circular queue. The key is using the modulo operator (`%`) when incrementing `rear` and `front`.

- **Disadvantages**: Implementation is slightly more complex.

2. **Dequeue (Double-Ended Queue)**:
   - **Definition**: A deque (pronounced "deck") is a double-ended queue. It allows insertion and deletion at *both* ends (front and rear).
   - **Operations**:
     - `enqueueFront`: Adds an element to the front.
     - `enqueueRear`: Adds an element to the rear.
     - `dequeueFront`: Removes an element from the front.
     - `dequeueRear`: Removes an element from the rear.
     - `getFront`: Gets the front element (without removing).
     - `getRear`: Gets the rear element (without removing).
     - `isEmpty`: Checks if empty.
     - `isFull`: Checks if full.
   - **Advantages**: Very flexible; can be used as a stack, a queue, or a combination of both.
   - **Disadvantages**: More complex to implement than a simple queue.
3. **Priority Queue**:
   - **Definition**: A priority queue is a queue where each element has an associated *priority*. Elements are dequeued based on their priority (highest priority first). Elements with the *same* priority are typically dequeued in the order they were enqueued (FIFO within the same priority).
   - **Analogy**: An emergency room in a hospital, where patients with the most severe conditions are treated first, regardless of when they arrived. Another analogy is a task scheduler in an operating system, where tasks with higher priorities (e.g., real-time tasks) are executed before lower-priority tasks.
   - **Operations**:
     - `enqueue(element, priority)`: Adds an element to the queue with a given priority.
     - `dequeue()`: Removes and returns the element with the *highest* priority.
     - `peek()`: Returns the highest-priority element *without* removing it.
     - `isEmpty()`: Checks if the queue is empty.
     - `isFull()`: Checks if the queue is full (relevant for array-based implementations).
   - **Implementation**: Priority queues can be implemented using various underlying data structures, each with different performance characteristics:
     - **Unsorted Array/List**: `enqueue` is O(1) (just add to the end), but `dequeue` and `peek` are O(n) (you have to search for the highest priority element).
     - **Sorted Array/List**: `enqueue` is O(n) (you have to insert in the correct sorted position), but `dequeue` and `peek` are O(1) (the highest priority element is at the front or end, depending on the sort order).
     - **Heap (Binary Heap)**: This is the *most common and efficient* implementation. A heap is a tree-based data structure that satisfies the heap property:
       - **Min-Heap**: The value of each node is less than or equal to the value of its children (for a min-priority queue, where *lower* values represent *higher* priority).
       - **Max-Heap**: The value of each node is greater than or equal to the value of its children (for a max-priority queue, where *higher* values represent *higher* priority).
       - Using a heap, both `enqueue` and `dequeue` operations have a time complexity of O(log n), which is very efficient. `peek` is O(1).
     - **Self-Balancing Binary Search Trees (e.g., AVL Trees, Red-Black Trees)**: Can also be used, providing O(log n) performance for all operations, but are more complex to implement than heaps.
   - **Advantages**:
     - Efficiently manages elements with priorities.
     - Provides fast access to the highest-priority element.
   - **Disadvantages**:
     - More complex to implement than simple queues, especially when using heaps or self-balancing trees.

Applications of Queues

Queues are fundamental data structures used in a wide variety of applications, including:

- **Operating Systems**:
  - **CPU Scheduling**: Processes waiting for execution by the CPU are often managed in a queue (or multiple queues with different priorities). The operating system uses scheduling algorithms (e.g., Round Robin, Priority Scheduling) to determine which process to run next.
  - **I/O Request Handling**: Requests to access I/O devices (e.g., printers, disks) are typically queued. This ensures that requests are handled in an orderly fashion.

    - **Interrupt Handling**: Interrupts (signals from hardware or software) are often placed in a queue to be processed by the operating system.
    - **Process management**: Maintaining a queue of processes that are ready to run.
    - **Memory management**: Managing free and allocated memory blocks.
- **Networking**:
    - **Packet Queuing**: Data packets arriving at a network router or switch are placed in queues to be transmitted. Queuing mechanisms (e.g., FIFO, Weighted Fair Queuing) manage the order of packet transmission.
    - **Buffering**: Queues are used to buffer data streams, ensuring smooth data flow even when there are temporary variations in transmission rates.
- **Computer Simulations**:
    - **Event Queues**: Simulations often involve events that occur at different times. A priority queue can be used to store events, with the priority representing the time of the event. The simulation processes events in chronological order.
- **Printing**:
    - **Printer Spooling**: Print jobs sent to a printer are placed in a queue. The printer processes the jobs one at a time, in the order they were received (or based on priority).
- **Web Servers**:
    - **Request Queues**: Web servers use queues to manage incoming requests from clients. This prevents the server from being overwhelmed by a large number of simultaneous requests.
- **Graphs and Trees**:
    - **Breadth-First Search (BFS)**: BFS is a graph traversal algorithm that uses a queue to explore nodes level by level. It's used for finding the shortest path in an unweighted graph, among other applications.
- **Data Buffers**:
    - **Keyboard Input**: Characters typed on a keyboard are often stored in a queue (a buffer) before being processed by an application.
    - **Asynchronous Operations**: In applications involving asynchronous operations (e.g., reading data from a file or network), queues can buffer data between the producer (the part of the application that generates data) and the consumer (the part that processes the data).
- **Call Centers**: Call centers use queues to hold callers waiting to speak to a representative.
- **Messaging Systems**: Message queues are used in distributed systems to enable asynchronous communication between different parts of an application or between different applications.
- **Task Scheduling**: Queues can be used to manage tasks that need to be executed in the background, such as sending emails, processing images, or generating reports.