CS303 – Data Structure Sem-3rd CSE RGPV

By: Mr. Sonu Kumar

Module-1: Review of C Programming Language

C Programming Language Tutorial

- The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.
- C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C as a mother language

- C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc. 1970.

2) C as a system programming language

- A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**.
- It can't be used for internet programming like Java, .Net, PHP, etc.

3) C as a procedural language

- A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.
- A procedural language breaks the program into functions, data structures, etc.

4) C as a structured programming language

- A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

5) C as a mid-level programming language

- C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**.

- C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).
- A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.
- A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

C Program

- In this tutorial, all C programs are given with C compiler so that you can quickly change the C program code.

File: main.c

```
1. #include <stdio.h>
2. int main() {
3. printf("Hello C Programming\n");
4. return 0;
5. }
```

Output = Hello C Programming

Overview of C Programming

Introduction:

- C is a powerful and widely-used programming language known for its efficiency and versatility.

Some key features of C include:

- **Efficiency:** C is known for its efficiency, making it a popular choice for systems programming and embedded systems.
- **Portability:** C code can be compiled on different platforms with minimal modifications.
- **Modularity:** C supports the creation of reusable modules and libraries.
- **Pointers:** C allows direct memory manipulation through pointers, giving programmers fine-grained control.
- **Standard Library:** C provides a standard library that includes functions for I/O, string manipulation, and more.
- **Procedural Programming:** It follows a structured, procedural approach to programming.

Mr. Sonu Kumar

Introduction to Data Structures:

- Data structures are a fundamental concept in computer science. They are ways of organizing and storing data to perform operations efficiently. In C, data structures are crucial for solving complex problems. Some common data structures include:

- **Arrays:** A collection of elements of the same data type stored in contiguous memory locations.
- **Linked Lists:** A collection of nodes, where each node contains data and a reference to the next node.
- **Stacks and Queues:** Abstract data types that use specific rules for data access (Last-In-First-Out for stacks, and First-In-First-Out for queues).
- **Trees:** Hierarchical data structures with nodes connected by edges, such as binary trees and AVL trees.
- **Graphs:** Collections of nodes and edges used to represent complex relationships.

Concept of Data and Information:

- **Data:** Data refers to raw facts or values, such as numbers, characters, or binary code. It lacks context or meaning on its own.
- **Information:** Information is data that has been processed, organized, or structured to make it meaningful. Information provides context and knowledge.

Classification of Data Structures:

1. **Primitive Data Structures:** These are basic data types provided by the programming language, like integers, floats, characters, and arrays. They can be used to build more complex data structures
2. **Non-primitive Data Structures:** These data structures are created by combining primitive data structures or other non-primitive data structures. They include linked lists, trees, graphs, stacks, and queues. These structures allow for more advanced data organization and manipulation.

Abstract Data Types (ADTs):

- ADTs are high-level, user-defined data types that focus on what the data structure does, rather than how it's implemented. They encapsulate data and operations on that data.

Mr. Sonu Kumar

Key characteristics of ADTs include:

- **Encapsulation:** ADTs hide the implementation details, allowing users to interact with the data structure through defined operations.
- **Data and Operations:** ADTs define the data stored and the operations that can be performed on that data.
- **Reusability:** ADTs can be used as building blocks for solving various problems, promoting code reusability.

Memory Representation:

In C, data structures are represented in memory using different methods:

- **Arrays:** Contiguous block of memory storing elements of the same type. Accessing elements is direct using indices.
- **Pointers:** Used to create dynamic data structures like linked lists and trees. Pointers store memory addresses and allow for non-contiguous memory allocation.
- **Structures:** C struct is a composite data type to store a collection of different data types.

Data Structure Operations and Cost Estimation:

- Every data structure supports specific operations, and their efficiency varies. Common operations on data structures include insertion, deletion, searching, and traversal.
- The cost of operations in a data structure can be estimated in terms of time complexity (how the time required grows with input size) and space complexity (how much memory is used with input size).
- **Time Complexity:** Expressed using Big O notation. It describes the worst-case time taken by an algorithm concerning the input size. For example, O(1) means constant time, O(log n) represents logarithmic time, O(n) indicates linear time, and so on.
- **Space Complexity:** Indicates the amount of memory used by an algorithm concerning the input size.

Introduction to Liear Data Structures: Arrays and Linked Lists:

- **Arrays:** An array is a collection of elements stored in contiguous memory locations.

Mr. Sonu Kumar

- **Implementation:** In C, arrays can be declared with a fixed or dynamic size.
- **Access:** Elements are accessed via indices. The access time for arrays is O(1) since accessing an element requires knowing the base address and adding the index to it.
- **Advantages:** Constant time access, simplicity.
- **Disadvantages:** Fixed size, inefficient insertion/deletion (O(n)) due to shifting elements.

2. Linked Lists:

- **Definition:** A linked list is a collection of nodes where each node holds data and a reference to the next node.
- **Implementation:** Implemented using pointers in C. Nodes are dynamically allocated.
- **Types:** Singly linked lists (each node points to the next node) and doubly linked lists (each node points to the next and previous nodes).
- **Access:** Access time in a linked list is O(n) as elements are not stored contiguously. Traversal is required to reach a specific element.
- **Advantages:** Dynamic size, efficient insertion/deletion (O(1)) by manipulating pointers.
- **Disadvantages:** Higher space overhead due to the storage of pointers, and sequential access is not as efficient as arrays.

Representation of a Linked List in Memory:

- A linked list is a data structure consisting of a collection of nodes, where each node contains two parts: data and a reference (or pointer) to the next node in the sequence.
- Linked lists can have different variations, such as singly linked lists, circular linked lists, and doubly linked lists.
- **Node:** A node is a fundamental building block of a linked list.

It typically consists of two parts:

- **Data:** The actual data or value stored in the node.
- **Next (or Previous in the case of doubly linked lists):** A pointer or reference to the next (or previous) node in the list.

Here's how a basic representation of a singly linked list in memory might look:

Mr. Sonu Kumar

Copy code

```
Node 1          Node 2          Node 3
+------+        +------+        +------+
| Data |  -----> | Data |  -----> | Data |
+------+        +------+        +------+
| Next |        | Next |        | Next |
+------+        +------+        +------+
```

The "Next" field of each node points to the next node in the sequence. The last node's "Next" field typically points to NULL, indicating the end of the list.

Different Implementations of Linked Lists:

- **Singly Linked List:**
  - Each node points to the next node in the sequence.
  - Efficient for forward traversal.
  - Requires less memory as it has only one pointer per node.
  - Inefficient for reverse traversal.
- **Doubly Linked List:**
  - Each node has both a "Next" and a "Previous" pointer.
  - Allows efficient traversal in both forward and reverse directions.
  - Requires more memory due to the extra "Previous" pointers.
- **Circular Linked List:**
  - Similar to singly or doubly linked lists but with the last node pointing back to the first node (in the case of a singly circular linked list) or with both the first and last nodes pointing to each other (in the case of a doubly circular linked list).
  - Suitable for applications that require continuous looping through a list.
- **Singly Linked List:**
  - A singly linked list is a linear data structure where each node points to the next node in the sequence.
  - It consists of nodes where each node has two components: data and a pointer to the next node.

Advantages:

- Efficient for insertion and deletion at the beginning or end (O(1)).
- Low memory overhead compared to doubly linked lists.

Disadvantages:

- Inefficient for accessing nodes in reverse order (O(n)).
- To access a specific node, you must traverse the list sequentially.

Mr. Sonu Kumar

Doubly Linked List:

- A doubly linked list is similar to a singly linked list, but each node has two pointers: one pointing to the next node and the other pointing to the previous node.

Advantages:

- Efficient for traversal in both forward and reverse directions.
- Supports efficient insertion and deletion at both ends (O(1)).

Disadvantages:

- Higher memory overhead due to the extra "Previous" pointers.

Circular Linked List:

- A circular linked list is a variation of linked lists where the last node points back to the first node, creating a circular structure.

Advantages:

- Suitable for applications where you need to continuously loop through a list.
- Similar advantages and disadvantages as singly or doubly linked lists, depending on the variant.

Applications of Linked Lists:

- Linked lists find applications in a wide range of programming scenarios, owing to their flexibility and dynamic memory allocation.

Some common applications include:

- **Dynamic Data Structures:** Linked lists are ideal for creating dynamic data structures, such as stacks, queues, and symbol tables, which can expand or shrink as needed.
- **Memory Allocation:** In languages like C and C++, linked lists are used for dynamic memory allocation when the size of data is not known in advance.
- **File Systems:** File systems use linked lists to manage file and directory structures, organizing the data efficiently.
- **Garbage Collection:** Memory management systems use linked lists to keep track of allocated and deallocated memory blocks.
- **Task Scheduling:** Linked lists are used to manage tasks and their dependencies in task scheduling algorithms.
- **Undo Functionality:** In applications like text editors and graphics software, linked lists are used to implement undo/redo functionality by maintaining a history of operations.
- **Music and Video Playlists:** Media players use linked lists to create and manage playlists.

Polynomial Manipulation Using Linked Lists:

Mr. Sonu Kumar

- Polynomials are mathematical expressions consisting of terms like coefficients and exponents.
- They are commonly used in fields like mathematics, engineering, physics, and computer graphics.
- Polynomials can be efficiently manipulated and evaluated using linked lists.

Polynomial manipulation using a linked list works:

Data Structure for Polynomial:

- Each term of the polynomial can be represented as a node in the linked list.
- The node structure typically contains two fields: coefficient and exponent.
- Each node points to the next node in the list.

Creating a Polynomial:

- To create a polynomial, you add nodes to the linked list, each representing a term
- Nodes are added in descending order of exponents to maintain the polynomial's standard form.

Operations on Polynomials:

- **Addition:** To add two polynomials, you traverse both linked lists simultaneously, combining terms with the same exponent and creating new nodes for terms with different exponents.
- **Subtraction:** Similar to addition, but subtracting coefficients when exponents match.
- **Multiplication:** For polynomial multiplication, you create a new linked list and perform a cross-product of terms from both polynomials, adding like terms to the result.
- **Evaluation:** To evaluate the polynomial for a given value of 'x', you traverse the linked list, substitute 'x' into each term, and sum the results.

Displaying Polynomials:

- You can traverse the linked list to display the polynomial in a human-readable format.

Advantages of Using Linked Lists for Polynomial Manipulation:

- **Dynamic Size:** Linked lists can accommodate polynomials of varying lengths without requiring a fixed-size array.
- **Efficiency:** Polynomial manipulation operations, such as addition, subtraction, and multiplication, can be implemented efficiently using linked lists.
- **Flexibility:** You can easily insert, delete, or modify polynomial terms.

Example:

- Polynomial A: 3x^3 + 2x^2 - 5x + 7
- Polynomial B: 2x^2 + 4x – 9

Mr. Sonu Kumar

Module – 2: Stacks In Data Structure

What is a Stack

- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack.
- a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Standard Stack Operations

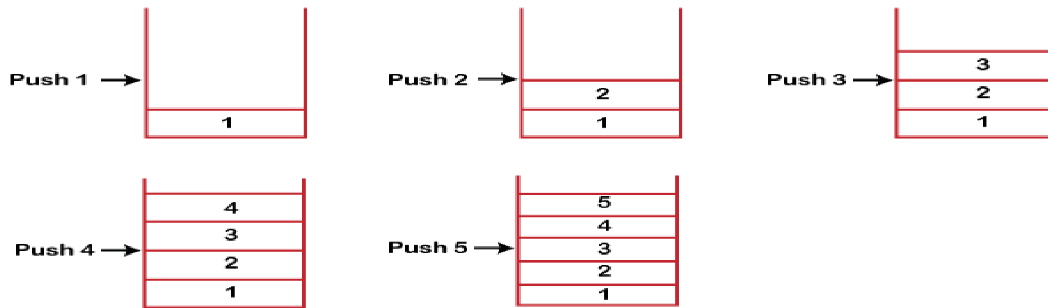
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a 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 means that no element exists in the stack, 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 or not.'
- **peek():** It returns the element at the given position.
- **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.

PUSH operation

The steps involved in the PUSH operation is given below:

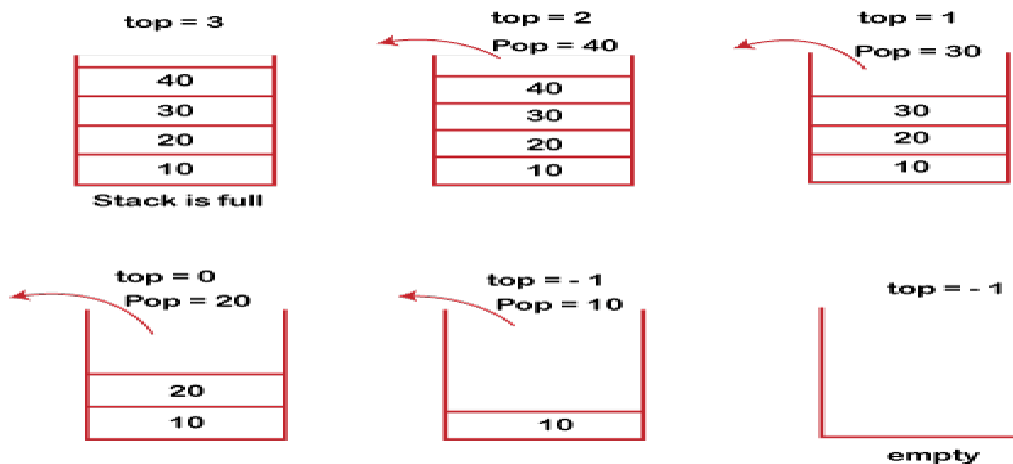
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element 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.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Introduction to Stacks:

- A stack is a fundamental data structure that follows the **Last-In-First-Out (LIFO) principle,** which means that the most recently added item is the first one to be removed.
- Stacks are widely used in computer science and various applications, including parsing expressions, function call management, and backtracking algorithms.

Stack as an Abstract Data Type (ADT):

- An Abstract Data Type (ADT) represents a data structure based on its behavior and operations, rather than its implementation.

For a stack ADT, we define a set of essential operations:

- **Push:** Add an element to the top of the stack.

- ➢ **Pop:** Remove and return the element from the top of the stack.
- ➢ **Peek (or Top):** Return the element at the top of the stack without removing it.
- ➢ **IsEmpty:** Check if the stack is empty.
- ➢ **Size:** Get the number of elements in the stack.

Different Implementations of Stacks:

- There are several ways to implement a stack data structure, each with its advantages and disadvantages. Here are some common implementations:

Array-Based Stack:

- ➢ **Implementation:** Use a one-dimensional array to store the stack elements.

Operations:

- ➢ **Push:** Add an element at the end of the array.
- ➢ **Pop:** Remove and return the last element.
- ➢ **Peek:** Return the last element without removing it.
- **Advantages:** Simple, efficient constant-time (O(1)) access.
- **Disadvantages:** Fixed size, limited capacity.

Linked List-Based Stack:

- ➢ **Implementation:** Use a singly linked list with nodes containing the data and a reference to the next node.

Operations:

- ➢ **Push:** Add a new node at the beginning of the list.
- ➢ **Pop:** Remove and return the first node.
- ➢ **Peek:** Return the first node without removing it.

- **Advantages:** Dynamic size, efficient for inserting and deleting elements.
- **Disadvantages:** Slightly more memory overhead due to pointer storage.

Multiple Stacks:

- Multiple stacks can be implemented using various techniques, allowing you to manage multiple stacks within a single data structure. Here are a couple of ways to implement multiple stacks:

Fixed Division:

- In this approach, you divide the array or linked list into fixed sections, each representing a separate stack.
- Operations for each stack are performed within its designated section.

Dynamic Division:

- In this approach, you allow stacks to grow and shrink as needed without fixed divisions.
- Each stack keeps track of its boundaries and manages its own elements independently.

Mr. Sonu Kumar

Array of Stacks:

- Create an array where each element is a separate stack (array or linked list).
- Operations are performed on the selected stack based on the index in the array.

Variable Division:

- Divide the data structure into variable sections based on the capacity of each stack.
- As each stack grows, it takes available space from the other stacks.

Advantages of Multiple Stacks:

- Efficiently manage multiple stacks within a single data structure.
- Useful in applications like memory management, expression evaluation, and backtracking algorithms.

Applications of Multiple Stacks:

- Expression evaluation: Handling multiple levels of operator precedence.
- Memory management: Allocating and deallocating memory for different data structures.
- Backtracking algorithms: Storing and managing states at different levels of recursion.

Applications of Stacks:

- Stacks are versatile data structures with numerous applications in computer science and everyday computing.

Some common applications include:

- **Function Call Management:** Stacks are used to manage function calls and their return addresses in programming languages. When a function is called, its context is pushed onto the stack, and when it returns, the context is popped off.
- **Expression Evaluation:** Stacks are essential for evaluating expressions, both infix and postfix notations. They help in handling operator precedence and associativity.
- **Undo/Redo Functionality:** Stacks can be used to implement undo and redo functionality in applications, allowing users to revert and reapply changes.
- **Parsing and Syntax Checking:** Stacks are used in parsing algorithms to check and enforce the correct order of parentheses, brackets, or other delimiters in code.
- **Backtracking Algorithms:** Stacks can be employed in backtracking algorithms, where the current state is pushed onto the stack, and if a dead-end is reached, you backtrack by popping the stack.
- **Memory Management:** Stacks are used in memory allocation and deallocation, such as in managing the call stack in a program.

Mr. Sonu Kumar

Recursion:

- Recursion is a programming technique where a function calls itself in order to solve a problem.
- Recursive functions break down a complex problem into simpler instances of the same problem.

Key elements of recursion include:

- **Base Case:** A condition that stops the recursion. When the base case is met, the function returns a result.
- **Recursive Case:** The part of the function where it calls itself with modified parameters to make progress toward the base case.

- Recursion is widely used in various programming scenarios, such as traversing trees and graphs, solving problems like factorials and Fibonacci sequences, and implementing divide-and-conquer algorithms.
- Recursion can lead to elegant and concise code, but it requires careful design to ensure termination and correctness.

Introduction to Queues:

- A queue is a linear data structure that follows the First**-In-First-Out (FIFO) principle**, where the element that is added first is the one that gets removed first. Think of it like a real-life queue of people waiting in line.
- Queues are used to manage data or tasks in an ordered manner, making it an essential data structure in computer science.

Queues as an Abstract Data Type (ADT):

- An Abstract Data Type (ADT) represents a data structure based on its behavior and operations rather than its implementation.

For a queue ADT, the fundamental operations include:

- **Enqueue (Push):** Add an element to the back (rear) of the queue.
- **Dequeue (Pop):** Remove and return the element from the front of the queue.
- **Front (Peek):** Return the element at the front without removing it.
- **IsEmpty:** Check if the queue is empty.
- **Size:** Get the number of elements in the queue.

Different Implementations of Queues:

- Queues can be implemented in several ways to suit different requirements. Common implementations include:

Array-Based Queue:

Mr. Sonu Kumar

- Implemented using an array with two indices, one for the front and one for the rear.
- Enqueue adds elements to the rear, and dequeue removes elements from the front.
- Circular array-based queue can be used to wrap around when the rear index reaches the end of the array.

Linked List-Based Queue:

- Implemented using a singly linked list where elements are added to the rear and removed from the front.
- Efficient for dynamic resizing as the linked list can grow as needed.
- Useful when the size of the queue is not known in advance.

Double-Ended Queue (Deque):

- A deque is a generalized form of a queue that allows elements to be added or removed from both the front and the rear.
- It combines the features of a stack and a queue and is useful in various applications.

Priority Queue:

- A priority queue assigns a priority value to each element and allows you to remove elements based on their priority.
- It is often implemented using data structures like heaps, binary trees, or arrays with a comparison function.

Circular Queues:

- A circular queue is an extension of a regular queue that overcomes the limitation of an array-based queue where the front and rear can't wrap around.
- In a circular queue, when the rear index reaches the end of the array, it wraps around to the beginning. This allows for efficient use of available memory.

Double-Ended Queue (Deque):

- A deque, also known as a **double-ended queue**, is a versatile data structure that allows elements to be added or removed from both ends.
- Deques combine the features of stacks and queues and can be used for various applications like implementing a stack, queue, or as a general-purpose data structure for efficient insertion and deletion at both ends.

Priority Queue:

Mr. Sonu Kumar

- A priority queue is a specialized type of queue where elements are assigned priorities, and elements with higher priorities are dequeued before those with lower priorities.
- Priority queues can be implemented using data structures like heaps, binary trees, or arrays with a comparison function.

Queue Simulation:

- Queue simulations involve modeling real-world systems with queues. This is used to analyze and optimize systems like traffic flow, computer networks, and service centers.
- Queue simulation can help predict and improve system performance.

Applications of Queues:

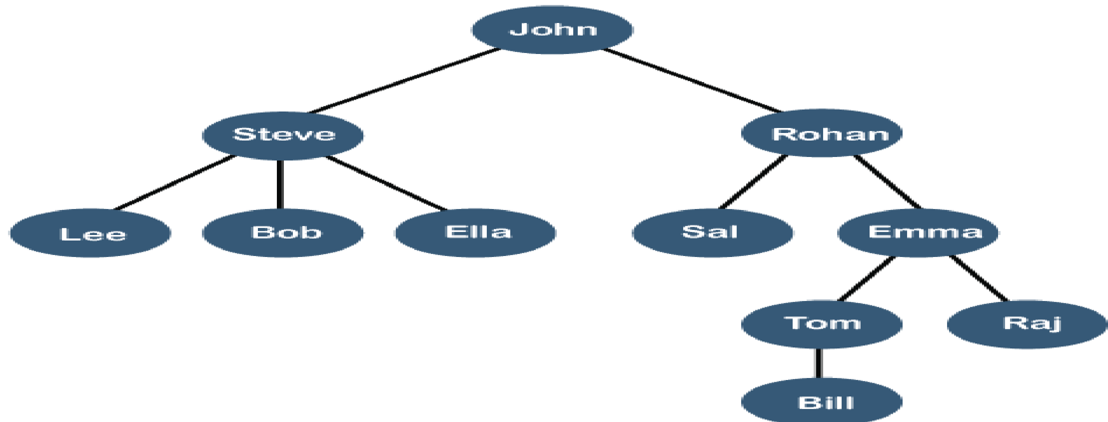
- Queues have a wide range of applications in computer science and various fields:
  - **Task Scheduling:** In operating systems, queues are used to schedule processes for execution based on their priorities.
  - **Print Queue:** In a printing system, print jobs are placed in a queue and printed in the order they are received.
  - **Breadth-First Search (BFS):** In graph algorithms, queues are used to explore nodes level by level.
  - **Buffer Management:** In computer networks, queues are used to manage data packets in routers and switches.
  - **Call Center:** In customer service operations, calls are queued in a call center to be handled by agents.
  - **Baking Industry:** In a bakery, customers queue to order and receive their items.
  - **Waiting Lists:** In many scenarios like airline reservations, customers are placed on waiting lists, and they are served based on their position in the queue.

Mr. Sonu Kumar

Module -3: Trees In Data Structure

Tree Data Structure

- ***A tree*** is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below.



Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.
- Properties of Tree data structure: **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***.

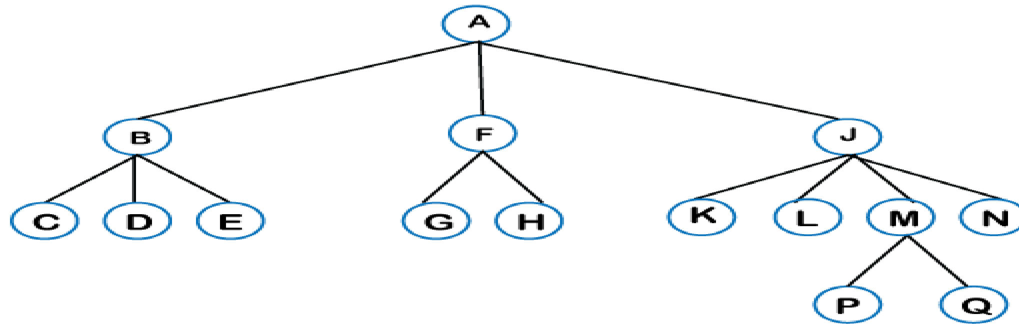
  - **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge.
  - **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path.
  - **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Applications of trees

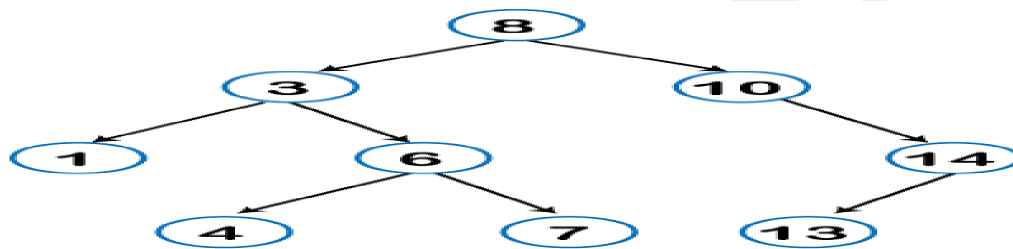
- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

1. **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes.
2. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



3. **Binary tree :** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



4. **Binary Search tree:** Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields
5. **AVL tree**: It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***.
6. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree.
1. **Red-Black Tree**: **The red-Black tree** is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node.
2. **Splay tree:** The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node.
3. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.
1. **Treap:** Treap data structure came from the Tree and Heap data structure. So, it comprises the properties of both Tree and Heap data structures. In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node.

2. **B-tree**: B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children.
3. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

Introduction to Trees:

- A tree is a widely used data structure in computer science and programming. It's a hierarchical structure composed of nodes connected by edges, with the following key features:
- **Root Node:** The topmost node in the tree, from which all other nodes are descended.
- **Parent Node:** A node that has one or more child nodes.
- **Child Node:** A node that has a parent node.
- **Leaf Node:** A node with no children.
- **Edge:** The link between two nodes.
- **Subtree:** A portion of the tree that itself is a tree.
- **Path:** A sequence of nodes connected by edges.
- **Depth:** The level or distance of a node from the root.
- **Height:** The length of the longest path from a node to a leaf.

Height and Depth:

- **Height:** The height of a tree is defined as the length of the longest path from the root to a leaf node. It represents the depth of the deepest leaf node.
- **Depth:** The depth of a node is the length of the path from the root to that particular node.

Order and Degree:

- **Order:** The order of a tree is the total number of child nodes that a parent node can have. For example, in a binary tree, each parent node can have at most two child nodes, so it's a binary tree with an order of 2.
- **Degree:** The degree of a node in a tree is the number of children it has. A leaf node has a degree of 0, a node with one child has a degree of 1, and so on.

Binary Search Tree (BST) - Operations, Traversal, and Search:

- A Binary Search Tree (BST) is a binary tree data structure with a specific property: for each node, all elements in its left subtree are less than the node's value, and all elements in its right subtree are greater.

Operations:

- **Insertion:** To insert a value into a BST, start from the root and traverse the tree. Compare the value to be inserted with the current node's value.
- If it's smaller, move to the left child; if larger, move to the right child. Repeat this process until you find an empty spot for insertion.

Mr. Sonu Kumar

- **Deletion:** To delete a node with a specific value from a BST, there are three cases to consider: a) If the node has no children, simply remove it. b) If the node has one child, replace it with the child. c) If the node has two children, replace it with the in-order successor (smallest value in the right subtree) or the in-order predecessor (largest value in the left subtree), and then delete the successor or predecessor.
- **Traversal:** Traversal of a BST involves visiting all the nodes in a specific order. Common traversal methods are:
- **In-Order:** Left, Root, Right (visits nodes in ascending order in a BST).
- **Pre-Order:** Root, Left, Right (used to create a copy of the tree).
- **Post-Order:** Left, Right, Root (used for deleting the entire tree).

Search:

- Searching in a BST is efficient due to its hierarchical structure. To search for a value:

Start at the root.

- If the value matches the current node's value, you've found it.
- If the value is less, move to the left subtree; if greater, move to the right subtree.
- Repeat this process until you either find the value or reach a leaf node (indicating the value isn't in the tree).

AVL Tree:

- An AVL tree is a self-balancing binary search tree (BST). It's characterized by the balance factor, which ensures that the height difference between the left and right subtrees of any node is at most 1.
- AVL trees maintain their balance through rotations, which are performed after insertion or deletion operations. These rotations ensure that the tree remains relatively balanced, which results in efficient search, insert, and delete operations with a time complexity of O(log n).
- **Heap:** A heap is a specialized tree-based data structure that satisfies the heap property, which depends on the type of heap:
- **Max Heap:** In a max heap, for any given node C, the value of C is greater than or equal to the values of its children.
- **Min Heap:** In a min heap, for any given node C, the value of C is less than or equal to the values of its children.

Comparison of Various Types of Trees:

- **BST (Binary Search Tree):** Good for searching and retrieval, but unbalanced BSTs can degrade to O(n) time complexity for insertions and deletions.
- **AVL Tree:** Self-balancing BST, ensuring O(log n) time complexity for insertions, deletions, and lookups.
- **Heap:** Used for priority queues, efficient for finding the maximum (in a max heap) or minimum (in a min heap) element.

Mr. Sonu Kumar

Introduction to Forest:

- A forest is a collection of trees, where each tree is a separate, disjoint set of nodes with no cycles. Forests are often used in graph theory and data structures.
- A forest can be considered a collection of individual trees.

Multi-Way Tree:

- A multi-way tree is a tree data structure where each node can have more than two children.
- Unlike binary trees, where nodes have at most two children, multi-way trees allow nodes to have multiple children, making them more versatile for certain applications.

B-Tree:

- A B-tree is a self-balancing tree structure designed to store large amounts of data and provide efficient disk I/O operations. B-trees have multiple child nodes per parent and maintain balance through a specific set of rules.
- They are commonly used in databases and file systems to ensure fast search and retrieval.

B+ Tree:

- A B+ tree is a variant of the B-tree that is optimized for disk storage and range queries. In a B+ tree, all data is stored in leaf nodes, which are linked together in a linked list for efficient range queries.
- Internal nodes serve as keys to navigate the tree structure.

B Tree:*

- A B* tree is another variant of the B-tree, aiming to reduce the amount of disk access during tree traversal.
- It has more restrictive rules regarding splitting and merging nodes compared to a traditional B-tree.

Red-Black Tree:

- A red-black tree is a self-balancing binary search tree that ensures that the tree remains approximately balanced during insertions and deletions.
- It uses a set of rules involving coloring nodes red and black and rotations to maintain balance. Red-black trees are commonly used in language libraries for associative containers (e.g., C++'s std::map and std::set).

Comparison of B-Tree Variants:

- **B-tree:** Good for general-purpose storage, efficient for search and insertions.
- **B+ tree:** Optimized for range queries and bulk loading, suitable for databases and file systems.
- **B tree:*** A variant aiming for fewer disk accesses during navigation.
- **Red-Black Tree:** Suitable for general-purpose associative containers in language libraries, but less efficient for large-scale data storage compared to B-trees.

Mr. Sonu Kumar

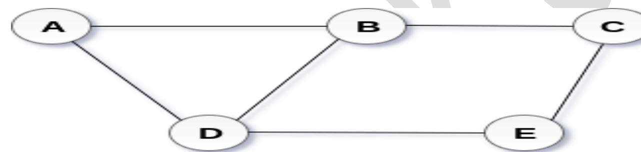
Module – 4 : Graph In Data Structure

Graph:

- A graph can be defined as group of vertices and edges that are used to connect these vertices.
- A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

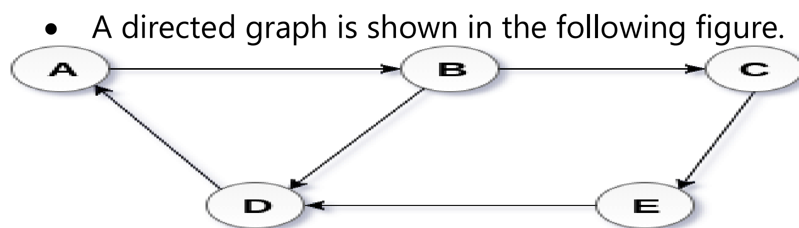
- A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
- A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

Directed and Undirected Graph

- A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.
- In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.
  - A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

- **Path :** A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path:** A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.
- **Simple Path:** If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.
- **Cycle :-** A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- **Connected Graph** :- a connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete Graph** :- a complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.
- **Weighted Graph :-** a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.
- **Digraph :- a** digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop:-** An edge that is associated with the similar end points can be called as Loop.
- **Adjacent Nodes :-** If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the Node** :- A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Introduction to Graphs:

- A graph is a fundamental data structure used to represent relationships between a collection of objects. These objects are called vertices or nodes, and the relationships between them are represented by edges.
- Graphs are used to model a wide range of real-world problems, from social networks and transportation systems to computer networks and data structures.

Classification of Graphs:

- **Directed vs. Undirected Graphs:**
- **Undirected Graph:** In an undirected graph, edges have **no direction.** They simply connect two vertices without indicating a one-way relationship.

Mr. Sonu Kumar

- If there is an edge from vertex A to vertex B, it implies an edge from B to A as well. Undirected graphs are symmetrical.
- **Directed Graph (Digraph):** In a directed graph, edges have **a direction**. They indicate a one-way relationship from one vertex to another.
- If there is a directed edge from vertex A to vertex B, it does not imply a directed edge from B to A. Directed graphs can represent asymmetric relationships.

Weighted vs. Unweighted Graphs:

- **Unweighted Graph:** In an unweighted graph, all edges are considered equal in terms of their relationship. They have no associated values or weights.
- Unweighted graphs are used to represent simple connections.
- **Weighted Graph:** In a weighted graph, edges have associated values or weights. These weights can represent various attributes, such as distances, costs, or strengths, depending on the application.
- Weighted graphs are used in problems where the strength or cost of relationships matters.

Cyclic vs. Acyclic Graphs:

- **Cyclic Graph:** A cyclic graph contains at least one cycle, which is a closed path of edges where you can traverse from a vertex and return to it without retracing any edges. Cyclic graphs may have loops or multiple paths between the same vertices.
- **Acyclic Graph:** An acyclic graph has no cycles. It is a tree-like structure where there are no closed loops or paths, making it a directed acyclic graph (DAG) if it is a directed graph. Trees are a common example of acyclic graphs.

Sparse vs. Dense Graphs:

- **Sparse Graph:** A sparse graph has relatively fewer edges compared to the number of vertices. It has many vertices with only a few edges connecting them.
- **Dense Graph:** A dense graph has a significant number of edges relative to the number of vertices. It is characterized by many connections between its vertices.

Connected vs. Disconnected Graphs:

- **Connected Graph:** A connected graph is one where there is a path between every pair of vertices. There are no isolated or disconnected components.
- **Disconnected Graph:** A disconnected graph consists of multiple isolated components, each of which is connected within itself but not to other components.

Mr. Sonu Kumar

Representation of Graphs:

- **Adjacency Matrix:** In an adjacency matrix, a 2D array is used to represent a graph. The rows and columns correspond to vertices, and the matrix cells indicate whether there is an edge between the vertices.
- If there is an edge, the cell contains a value (usually 1 for an unweighted graph or the weight for a weighted graph), and if there is no edge, the cell contains 0. This representation is suitable for dense graphs but can be memory-inefficient for sparse graphs.
- **Adjacency List:** In an adjacency list, each vertex is associated with a list of its adjacent vertices. This representation is memory-efficient for sparse graphs and is commonly used for various graph algorithms.

Graph Traversal:

- Graph traversal involves visiting all the vertices and edges of a graph in a systematic way. Two common methods for graph traversal are:
- **Depth-First Search (DFS):** DFS explores as far as possible along a branch before backtracking. It can be implemented using recursion or a stack.
- DFS is useful for tasks like topological sorting, cycle detection, and path finding. It is often used to explore deeper into a graph.
- **Breadth-First Search (BFS):** BFS explores all the neighbors of a vertex before moving to the next level of neighbors. It is implemented using a queue.
- BFS is useful for tasks like finding the shortest path, level order traversal, and connected component analysis. It is often used to explore broader in a graph.

Graph Algorithms:

- Several fundamental graph algorithms can be applied to solve various problems:

Minimum Spanning Tree (MST):

- **Kruskal's Algorithm:** Kruskal's algorithm finds the MST by iteratively adding the shortest edge that doesn't form a cycle.
- **Prim's Algorithm:** Prim's algorithm finds the MST by iteratively adding the shortest edge connected to the current tree.

Mr. Sonu Kumar

- **Dijkstra's Shortest Path Algorithm:** Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted, non-negative graph.
- It uses a priority queue to select the next vertex with the smallest known distance.

Comparison Between Graph Algorithms:

- **MST (Kruskal vs. Prim):** Both algorithms find the MST, but Kruskal's algorithm is edge-based, while Prim's algorithm is vertex-based
- Kruskal's works for disconnected graphs and is more versatile, whereas Prim's is typically faster for dense graphs.
- **Dijkstra vs. BFS:** Dijkstra's algorithm finds the shortest path in weighted graphs, while BFS finds the shortest path in unweighted graphs.
- Dijkstra's algorithm uses a priority queue to efficiently explore the graph, while BFS uses a queue for level-based traversal.

Applications of Graphs:

- Graphs are used in numerous real-world applications, including:
- **Social Networks:** Modeling friendships and connections between individuals.
- **Transportation Networks:** Representing road networks, flight connections, and public transportation systems.
- **Computer Networks:** Modeling network topology and routing algorithms.
- **Recommendation Systems:** Analyzing user preferences and recommending items.
- **Game Development:** Designing game maps and pathfinding for characters.
- **Data Structures:** Building complex data structures like trees, linked lists, and hash tables.
- **Natural Language Processing:** Analyzing language structures and semantic relationships.
- **Network Analysis:** Detecting patterns, vulnerabilities, and clusters in networks.
- **Bioinformatics:** Analyzing biological networks and genetic relationships.
- **Geographical Information Systems (GIS):** Mapping geographical data and spatial relationships.

Mr. Sonu Kumar

Module -5 : Sorting In Data Structure

Introduction:

- Sorting is a fundamental computer science operation that entails putting a group of objects in a specific order.
- It is extensively utilised in many different applications, including database management, data analysis, and searching.
- In data structures, different sorting techniques are employed to organize and manipulate large sets of data efficiently.

Sorting Techniques:

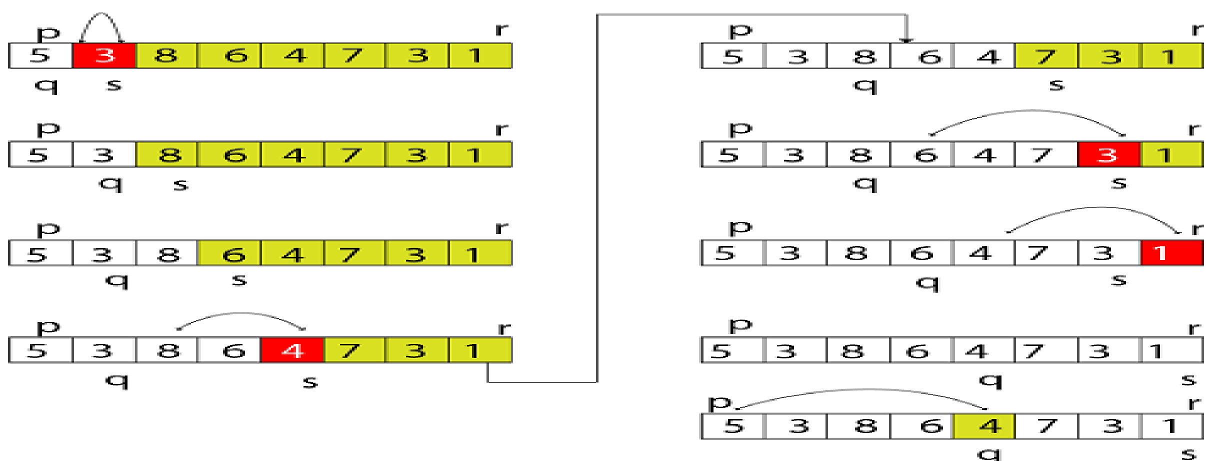
There are various sorting techniques, which are listed in the table below:

Algorithm	Description	Time Complexity	Space Complexity
Bubble Sort	Compares nearby elements, swaps any that are out of order, and repeats until the array is sorted.	O(n^2)	O(1)
Selection Sort	Divides the input array into two sections: one that is sorted and one that is not. It does this by repeatedly choosing the smallest element in the unsorted zone and moving it to the sorted region.	O(n^2)	O(1)
Insertion Sort	By continuously inserting each unsorted element into the appropriate location within the sorted subarray, the final sorted array is built one element at a time.	O(n^2)	O(1)
Merge Sort	Merge Sort splits the input array in half, sorts each half iteratively, and then combines the two sorted halves to create the final sorted array.	O(n log n)	O(n)
Quick Sort	Selects a pivot element and divides the array so that everything that is smaller than the pivot is put in front of it, and everything that is bigger than the pivot is put in back. The subarrays prior to and following the pivot are then sorted recursively.	O(n log n)	O(log n)

Heap Sort	Creates a max-heap from the input array, removes the highest element from the heap periodically, swaps it for the last element, and modifies the heap until the array is sorted.	O(n log n)	O(1)
Counting Sort	creates a count array to keep track of how many distinct elements there are in the input array. Then it determines the position of each element by calculating the cumulative sum of counts.	O(n + k)	O(n + k)
Bucket Sort	Divides the input array into a set of buckets, where each bucket represents a range of values. It distributes elements into the respective buckets based on their values. Then, each bucket is sorted separately, usually using a different sorting method.	O(n^2) or O(n + k)	O(n)
ix Sort	Sorts elements by processing individual digits or characters of the elements from the least significant digit (LSD) to the most significant digit (MSD). As a subroutine, it may employ a reliable sorting algorithm.	O(d * (n + k))	O(n + k)

Bubble Sort:

- The straightforward sorting method known as "Bubble Sort" analyses neighbouring elements, iteratively goes over the list, and swaps out any elements that are out of order. To sort the complete list, repeat this step.



Introduction to Sorting:

- Sorting is a fundamental operation in computer science and data processing, where the elements of a dataset are arranged in a specific order.
- The order can be ascending or descending based on a defined comparison criterion. Sorting plays a crucial role in data retrieval, data analysis, and the efficient organization of data.

Bubble Sort:

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- This process continues until the entire list is sorted. Bubble Sort has a worst-case and average-case time complexity of O(n^2) and is not the most efficient sorting algorithm, but it is easy to implement.

Quick Sort:

- Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively. Quick Sort is often faster than other sorting algorithms and has an average-case time complexity of O(n log n).

Selection Sort:

- Selection Sort is an in-place comparison sorting algorithm that divides the input list into two parts: a sorted region and an unsorted region.
- It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region.
- Selection Sort has a worst-case and average-case time complexity of O(n^2).

Heap Sort:

- Heap Sort is a comparison-based sorting algorithm that first transforms the input into a max-heap, a specialized binary tree where each parent node is greater than or equal to its child nodes.
- The root element is then swapped with the last element in the heap, removing it from the heap and placing it in its correct position.
- Heap Sort has a time complexity of O(n log n).

Insertion Sort:

- Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time.
- It is much less efficient on large lists than other algorithms such as Quick Sort, Merge Sort, or Heap Sort.
- Insertion Sort has a worst-case and average-case time complexity of O(n^2).

Mr. Sonu Kumar

Shell Sort:

- Shell Sort is an improvement over Insertion Sort that breaks the original list into smaller sub-lists.
- Each sub-list is sorted using Insertion Sort, with the sub-lists being progressively sorted until the entire list is sorted.
- Shell Sort has a time complexity that depends on the chosen gap sequence but is generally better than O(n^2).

Merge Sort:

- Merge Sort is an efficient, stable, and comparison-based sorting algorithm that divides the unsorted list into n sub-lists, each containing one element, and then repeatedly merges sub-lists to produce new sorted sub-lists until there is only one sub-list remaining.
- Merge Sort has a time complexity of O(n log n).

Radix Sort:

- Radix Sort is a non-comparative integer sorting algorithm that works by processing individual digits of the numbers to be sorted.
- It processes the least significant digit first, then the next most significant digit, and so on.
- Radix Sort can have linear time complexity for certain cases.

Comparison of Sorting Techniques:

- The choice of sorting algorithm depends on factors such as the size of the dataset, the presence of duplicate values, and the specific use case.
- Quick Sort and Merge Sort are typically preferred for general sorting. Radix Sort is suitable for sorting integers, and Heap Sort is a good choice when in-place sorting is needed.

- ➢ Quick Sort and Merge Sort are efficient with a time complexity of O(n log n).
- ➢ Bubble Sort and Selection Sort are simple but less efficient with a time complexity of O(n^2).
- ➢ Radix Sort is efficient for sorting integers with a linear time complexity.
- ➢ Shell Sort is a good choice for medium-sized datasets.
- ➢ Insertion Sort is efficient for small datasets or mostly sorted data.
- ➢ Heap Sort is used when in-place sorting is needed and is generally efficient.
- ➢ The best sorting algorithm depends on the specific use case and data characteristics.

Searching:

- Searching is the process of finding a specific item in a collection of data. Common searching algorithms include:

Mr. Sonu Kumar

- **Linear Search:** It sequentially checks each element in a list until a match is found or the end of the list is reached. It has a time complexity of O(n).
- **Binary Search:** Binary Search is a divide-and-conquer algorithm that works on sorted data. It repeatedly divides the search interval in half until the desired item is found.
- It has a time complexity of O(log n).
- **Hashing:** Hashing is a technique that converts a key into an index of an array.
- It is used in data structures like hash tables and has an average time complexity of O(1).
- **Interpolation Search:** Interpolation Search is an improved variant of binary search that works well for uniformly distributed data.
- It has an average time complexity of O(log log n).

Introduction to Sequential Search:

- Sequential Search, also known as **Linear Search,** is a straightforward searching algorithm that involves scanning a list or array of elements one by one, in a sequence, to find a specific target element.
- It starts at the beginning of the data and continues until the target is found or until the entire list has been examined.
- If the target element is found, the search terminates, and the position or index of the target in the list is returned.
- If the target is not present in the list, the search reaches the end, and it returns a signal to indicate that the target is not in the list.
- Sequential Search has a **time complexity of O(n)** in the worst case, where "**n**" is **the number of elements** in the list.

Introduction to Binary Search:

- Binary Search is a highly efficient searching algorithm that is applicable to sorted lists or arrays.
- It operates by repeatedly dividing the search interval in half, discarding half of the elements in each step, until the target element is found.
- Binary Search works on the principle that the data must be sorted, and it can quickly eliminate half of the remaining elements in each iteration.
- This results in a time complexity of O(log n) in the worst case, making it much faster than Sequential Search for large datasets.

Mr. Sonu Kumar

Sequential Search vs. Binary Search:

- **Data Type:** Sequential Search can be used on both sorted and unsorted data, whereas Binary Search requires data to be sorted.
- **Efficiency:** Binary Search is much more efficient than Sequential Search for large datasets due to its O(log n) time complexity. In contrast, Sequential Search has O(n) time complexity.
- **Applicability:** Sequential Search can be applied to any data, while Binary Search is only suitable for sorted data.
- **Memory Usage:** Sequential Search doesn't require extra memory, while Binary Search uses recursion or an iterative approach and might need additional stack space.
- **Complexity:** Binary Search is more complex to implement but provides significant performance benefits on large datasets.

Hashing and Indexing:

- **Hashing:** Hashing is a technique that involves mapping a key (e.g., a search value) to an index in an array using a hash function.
- It allows for quick access to the data when the key is known.
- Hashing is commonly used in data structures like hash tables, providing an average time complexity of O(1) for searching.
- **Indexing: I**ndexing involves creating a separate data structure, such as a B-tree or a simple index, that stores references or pointers to the actual data.
- This index structure is used for quickly accessing data based on specific keys or criteria.
- Indexing can provide efficient search operations and is commonly used in databases and search engines.

Application of Various Data Structures in Operating Systems:

- Data structures play a crucial role in the design and implementation of operating systems. Here are some case studies highlighting the use of various data structures in operating systems:

File Allocation Table (FAT) in File Systems:

- **Data Structure:** Linked List
- **Case Study:** In the FAT file system, a linked list data structure is used to keep track of the clusters (disk blocks) allocated to files.
- Each entry in the table points to the next cluster in the file.

Mr. Sonu Kumar

- Linked lists efficiently manage file fragmentation and are easy to extend for large storage.

Page Tables in Memory Management:

- **Data Structure:** Hash Table or Multilevel Page Table
- **Case Study:** Page tables are used to map virtual addresses to physical addresses in virtual memory systems.
- Hash tables or multilevel page tables help efficiently locate the corresponding physical page.
- Hash tables reduce the search time, while multilevel page tables help save memory space.

Process Control Blocks (PCB):

- **Data Structure:** Linked List
- **Case Study:** In process management, PCBs store information about each process.
- These PCBs are linked together in a queue, allowing the operating system to efficiently switch between processes, manage their state, and maintain a list of active processes.

File Descriptors and Open File Tables:

- **Data Structure:** Arrays and Hash Tables
- **Case Study:** File descriptors in Unix-like operating systems are maintained in the form of an array.
- An open file table is a hash table used to keep track of open files.
- These data structures enable efficient file operations and access control.

Application of Data Structures in Database Management Systems (DBMS):

- DBMSs rely on various data structures to optimize data storage, retrieval, and management.
- Here are some case studies illustrating the use of data structures in DBMS:

B-Tree and B+ Tree Indexes:

- **Data Structure:** B-Tree or B+ Tree
- **Case Study:** In DBMS, B-Tree and B+ Tree indexes are widely used to efficiently search, insert, and delete records in large databases.
- These self-balancing tree structures help maintain data integrity and enable logarithmic-time search operations.

Mr. Sonu Kumar

Hash Tables for Caching:

- **Data Structure:** Hash Table
- **Case Study:** Many DBMSs employ hash tables to implement query caches for frequently executed queries.
- Hashing allows for quick retrieval of previously computed query results, reducing the computational load on the database server.

Linked Lists for Linked Data Structures:

- **Data Structure:** Linked List
- **Case Study:** In graph databases and certain NoSQL databases, linked lists are used to represent relationships between entities or nodes.
- Linked data structures efficiently manage complex relationships and enable graph-based queries.

Red-Black Trees for Balanced Search Trees:

- **Data Structure:** Red-Black Tree
- **Case Study:** Red-Black trees are utilized to maintain balanced search trees in DBMS indexes.
- These trees offer efficient search, insertion, and deletion operations, ensuring data remains balanced for optimal query performance.

Priority Queues for Transaction Scheduling:

- **Data Structure:** Priority Queue
- **Case Study:** DBMSs often use priority queues to schedule database transactions based on their importance and deadline.
- Priority queues ensure that high-priority transactions are executed in a timely manner.

Segment Trees for Range Queries:

- **Data Structure:** Segment Tree
- **Case Study:** Segment trees are employed in databases for efficiently handling range queries. They store aggregated data for a range of values, facilitating complex queries like range-based aggregations.

CS303 – Data Structure Sem-3rd CSE RGPV

By: Mr. Sonu Kumar

Contact Number: 6200638476

Thank You !!!

Mr. Sonu Kumar