

**1. What is data structure ? Explain the classification of Data Structure?**

A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and used efficiently<sup>12</sup>. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program<sup>3</sup>. Data structures allow programs to store and process data effectively<sup>4</sup>.

Data structures can be classified into two broad categories<sup>3</sup>.

**Linear data structure:** Data elements are arranged sequentially, and each element is connected to its previous and next element. Examples include arrays, lists, stacks, and queues<sup>5</sup>.

**Non-linear data structure:** Data elements are not arranged in a sequential manner. Examples include trees and graphs<sup>5</sup>.

**What are the different operations that are performed on Data Structure?**

Different operations that can be performed on data structures include:

**Traversing:** Visiting each element of the data structure only once<sup>1</sup>.

**Searching:** Finding an element in a data structure that satisfies one or more conditions<sup>1</sup>.

**Inserting:** Adding elements of the same type in a data structure<sup>1</sup>.

**Deleting:** Removing a node from the data structure<sup>2</sup>.

**Sorting:** Arranging the elements in a certain order<sup>3</sup>.

**What is Algorithm & Flowchart? Differentiate between algorithm & flowchart.**

An **algorithm** is a step-by-step procedure to solve a problem or perform a task<sup>12</sup>. It refers to a set of instructions that define the execution of work to get the expected results<sup>12</sup>. Algorithms can be written in natural language, pseudocode, or programming languages<sup>13</sup>. Sometimes algorithms are difficult to understand, and it is also difficult to show looping and branching using an algorithm<sup>2</sup>.

A **flowchart**, on the other hand, is a visual representation of an algorithm or process<sup>1243</sup>. It uses various symbols to show the operations and decisions to be followed in a program<sup>12</sup>. A flowchart can make it easier to understand the process and identify potential problems with it<sup>4</sup>.

Here are some differences between an algorithm and a flowchart<sup>1</sup>:

An algorithm is a step-by-step procedure to solve a problem, while a flowchart is a diagram created with different shapes to show the flow of data<sup>1</sup>.

The algorithm is complex to understand, while a flowchart is easy to understand<sup>1</sup>.

In the algorithm, plain text is used, while in the flowchart, symbols/shapes are used<sup>1</sup>.

The algorithm is easy to debug, while a flowchart is hard to debug<sup>1</sup>.

The algorithm does not follow any rules, while the flowchart follows rules to be constructed<sup>1</sup>.

The algorithm is the pseudo-code for the program, while a flowchart is just a graphical representation of that logic<sup>1</sup>.

**4. What are the characteristics of Algorithm?**

The characteristics of an algorithm are as follows:

**Well-defined steps:** Algorithms consist of a specific and unambiguous set of instructions or steps that can be followed to perform a task or solve a problem<sup>1</sup>.

**Input and output:** Algorithms take inputs, which are the initial data provided to the algorithm, and produce outputs, which are the results generated by the algorithm after processing the inputs<sup>1</sup>.

**Finiteness:** Algorithms must have a well-defined termination condition, meaning they eventually reach an endpoint or change after a finite number of steps<sup>1</sup>.

**Determinism:** Algorithms are deterministic, meaning that given the same inputs and performed under the same conditions, they will always produce the same outputs<sup>1</sup>.

**Efficiency:** Algorithms strive to be efficient in terms of time and resources. They aim to solve problems or perform tasks in a reasonable amount of time and with optimal use of computational resources<sup>1</sup>.

**Generality:** An algorithm should be designed to solve a specific problem or perform a specific task, but it should also be applicable to a broader class of situations or scenarios<sup>1</sup>.

**What is abstract data type? Explain with a suitable example.**

An **Abstract Data Type (ADT)** is a type or class for objects whose behavior is defined by a set of values and a set of operations<sup>1</sup>. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented<sup>1</sup>. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations<sup>1</sup>. It is called "abstract" because it gives an implementation-independent view<sup>1</sup>. For example, a **List** is an abstract data type that can be implemented using a dynamic array and linked list<sup>2</sup>. A **Queue** is another example of an ADT, which can be implemented using linked list-based queue, array-based queue, and stack-based queue<sup>2</sup>. In these examples, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details<sup>2</sup>. This is because every programming language has a different implementation strategy<sup>2</sup>.

**Write some applications of Data Structure.**

Data structures are fundamental in computer science and have a wide range of applications. Here are some examples:

**Operating Systems:** Data structures are used in the management and organization of data in operating systems<sup>1</sup>.

**Graphics:** In computer graphics, data structures are used to represent images, objects, and other visual elements<sup>1</sup>.

**Compilers:** Compilers use data structures like syntax trees and symbol tables to parse and translate source code<sup>2</sup>.

**Databases:** Databases use data structures like B-trees and hash indexes for efficient data retrieval<sup>2</sup>.

**Artificial Intelligence:** In AI, data structures are used to model and solve complex problems<sup>2</sup>.

**Networking:** Data structures are used in networking to manage and route data efficiently<sup>2</sup>.

**Games:** In game development, data structures are used to store game states, player data, and other information<sup>2</sup>.

These are just a few examples. The choice of data structure depends on the specific requirements of the task at hand

**What is the difference between Linear & Non-Linear Data Structure?**

**Linear Data Structures:** In these structures, data elements are arranged sequentially or linearly, where each element is connected to its previous and next element<sup>1</sup>. Examples include arrays, stacks, queues, and linked lists<sup>1</sup>. Linear data structures are easy to implement because computer memory is arranged in a linear way<sup>1</sup>. They involve only a single level, allowing all elements to be traversed in a single run<sup>1</sup>.

**Non-Linear Data Structures:** In these structures, data elements are not arranged sequentially or linearly<sup>1</sup>. Examples include trees and graphs<sup>1</sup>. Non-linear data structures are not as easy to implement as linear data structures<sup>1</sup>. They involve multiple levels, so all elements cannot be traversed in a single run<sup>1</sup>. Non-linear data structures are more memory-friendly, meaning they use memory more efficiently<sup>1</sup>.

**What is the difference between Primitive & Non-Primitive Data Structure?**

**Primitive Data Structures:** These are the basic data structures that include fundamental data types like integers, floats, characters, and pointers<sup>12</sup>. They can hold a single type of value<sup>12</sup>. For example, an integer variable can hold an integer type of value, a float variable can hold a floating type of value, and a character variable can hold a character type of value<sup>12</sup>.

**Non-Primitive Data Structures:** These are more complex data structures that can store multiple data types in a single entity<sup>12</sup>. They are further classified into linear (like arrays, linked lists, stacks, queues) and non-linear data structures (like trees and graphs)<sup>12</sup>. For example, an array can contain several

integers, whereas a primitive data type like int can only store one integer at a time<sup>12</sup>.

**What is the difference between Static & Dynamic Data Structure?**

**Static Data Structures:** These have a fixed size and memory is allocated at compile-time<sup>1</sup>. The size is fixed and cannot be modified during runtime<sup>1</sup>. This means that their memory size cannot be changed during program execution<sup>1</sup>. Examples of static data structures include arrays<sup>1</sup>.

**Dynamic Data Structures:** These have a variable size and memory is allocated at run-time<sup>1</sup>. The size can be modified during runtime<sup>1</sup>. This means that their memory size can be changed during program execution<sup>1</sup>. Memory can be dynamically allocated or deallocated during program execution<sup>1</sup>. Examples of dynamic data structures include linked lists<sup>1</sup>.

**Unit – 2****1. What is Stack? What is FIFO? How to represent stack into memory?**

**Using the contiguous memory like an array:** In this representation, the stack is implemented using the array<sup>2</sup>. Variables used in this case are STACK (the name of the array storing stack elements), TOP (storing the index where the last element is stored in array representing the stack), and MAX (defining that how many elements, maximum count, can be stored in the array representing the stack)<sup>2</sup>.

**Using the non-contiguous memory like a linked list:** In this representation, the stack is implemented using the dynamic data structure Linked List<sup>2</sup>. Using linked list for Application of stack make a dynamic stack. You don't have the need to define the maximum number of elements in the stack<sup>2</sup>. Pointers (links) to store addresses of nodes used are TOP (storing the address of topmost element of the Linked list storing the stack)<sup>2</sup>.

**2. Write down some applications of Stack**

**A Stack** is a linear data structure that follows a Last In First Out (LIFO) principle and has various applications<sup>1234</sup>.

**Function Calls and Recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution<sup>2</sup>.

**Undo/Redo Operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions<sup>2</sup>.

**Expression Evaluation:** Stacks are used for evaluating arithmetic expressions consisting of operands and operators<sup>13</sup>.

**Memory Management:** Stacks are used for systematic memory management<sup>1</sup>.

**Backtracking:** Stacks are used for backtracking, i.e., to check parenthesis matching in an expression<sup>1</sup>.

**Task Scheduling:** In real-time applications, stacks are used for task scheduling<sup>4</sup>.

**3. Why stack follow LIFO principle.**

A **Stack** is a basic data structure where insertion and deletion of data takes place at one end called the top of the stack<sup>1</sup>. It follows the Last In First Out (LIFO) principle<sup>1</sup>. This principle dictates that the last element added to the stack is the first one to be removed<sup>1</sup>. In other words, the most recently added element is always at the top, and any operations on the stack will affect this top element<sup>1</sup>.

The LIFO principle is used in stacks for several reasons:

**Function Call Tracking:** Stacks help computers remember what functions they're doing. When you call a function, its details are pushed onto the stack. When the function finishes, its details are popped from the stack<sup>2</sup>.

**Undo/Redo Functionality:** We have undo and redo buttons in many apps, like text editors. Stacks help with this. When you do something, it's pushed onto the stack. Undoing means popping the last action and redoing means pushing it back on<sup>2</sup>.

**Expression Evaluation:** Stacks are used for solving math problems with parentheses. When you read a math expression, you can use a stack to track what to do with numbers and symbols, making sure the last thing you put in is the first to be used<sup>2</sup>.

**4. What do you understand by Stack Overflow & Stack Underflow?**

**Stack Overflow and Stack Underflow** are two conditions that can occur in a stack data structure<sup>1234</sup>.

**Stack Overflow:** This occurs when an attempt is made to add (push) an element onto a stack that is already full<sup>124</sup>. In other words, when there is no more space in the stack to hold new elements, a stack overflow error occurs<sup>124</sup>.

**Stack Underflow:** This occurs when an attempt is made to remove (pop) an element from a stack that is already empty<sup>1234</sup>. In other words, when there are no more elements in the stack to be removed, a stack underflow error occurs<sup>12</sup>.

#### 5. Explain different types of expression with example.

Expressions can be of various types, each with its own characteristics and examples:

**Constant Expressions:** These consist of only constant values. A constant value is one that doesn't change<sup>1</sup>.

Example: 5, 10 + 5 / 6.0.

**Integral Expressions:** These produce integer results after implementing all the automatic and explicit type conversions<sup>1</sup>.

Example:  $x * x * y, x + \text{int}(5.0)$  where x and y are integer variables.

**Floating Expressions:** These produce floating point results after implementing all the automatic and explicit type conversions<sup>1</sup>. Example:  $x + y, 10.75$  where x and y are floating point variables.

**Relational Expressions:** These yield results of type bool which takes a value true or false<sup>1</sup>. Example:  $x <= y, x + y > 2$ .

**Logical Expressions:** These combine two or more relational expressions and produces bool type results<sup>1</sup>. Example:  $x > y \& x == 10, x == 10 || y == 5$ .

**Pointer Expressions:** These produce address values<sup>1</sup>.

Example:  $\&x, \text{ptr}, \text{ptr}++$  where x is a variable and ptr is a pointer.

**Bitwise Expressions:** These are used to manipulate data at bit level<sup>1</sup>. Example:  $x \ll 3$  shifts three bit position to left,  $y \gg 1$  shifts one bit position to right.

#### 6. What is the difference between Push & Pop?

**Push and Pop** are two fundamental operations performed on the stack data structure<sup>1234</sup>:

**Push:** This operation adds a new item to the top of the stack<sup>1234</sup>. If the stack is empty, the push operation initializes it<sup>1234</sup>. The item being added (pushed) becomes the new top of the stack<sup>1234</sup>.

**Pop:** This operation removes an item from the top of the stack<sup>1234</sup>. The item being removed (popped) is always the one that was most recently added to the stack, following the Last In First Out (LIFO) principle<sup>1234</sup>. After a pop operation, the item that was previously second from the top becomes the new top of the stack<sup>1</sup>.

**7. Write down algorithm for PUSH & POP operation with related to the stack. (M-5)**

#### PUSH Operation Algorithm<sup>1</sup>:

```
if TOP = MAX-1
    return "Overflow"
else
    top = top + 1
    stack[top] = item
end
```

The steps for the PUSH operation are as follows<sup>1</sup>:

1. Check if the stack has some space or if the stack is full.
2. If the stack has no space then display "overflow" and exit.
3. If the stack has space then increase top by 1 to point to the next empty space.
4. Add the item to the newly stack location, where top is pointing.

#### POP Operation Algorithm<sup>1</sup>:

```
if TOP = -1
    return "Underflow"
else
    item = stack[top]
    top = top - 1
    return item
end
```

The steps for the POP operation are as follows<sup>1</sup>:

1. Check if the stack has some element or if the stack is empty.
2. If the stack has no element (i.e., it is empty) then display "underflow".
3. If the stack has some element, access the data element at which top is pointing.
4. Decrease the value of top by 1.

#### 8. Write prefix form for the expression : $(A+B*C) - (D/E)$ .

The prefix form of an expression is a notation where the operator is placed before its operands. This is also known as Polish notation<sup>1</sup>.

To convert the infix expression  $(A+B*C) - (D/E)$  to prefix notation, we follow these steps<sup>12</sup>:

1. Reverse the infix expression:  $E/D - (C*B+A)$
2. Replace each '(' with ')' and each ')' with '(' :  $E/D ) - (C*B+A)$

3. Convert the modified expression to postfix:  $ED/BC+AB-$
4. Reverse the postfix expression to get the prefix expression:  $-+AB*BC/ED$

So, the prefix form of the expression  $(A+B*C) - (D/E)$  is  $-+AB*BC/ED$ .

#### 9. Write postfix form for the above expression

The postfix form of an expression is a notation where the operator is placed after its operands. This is also known as Reverse Polish notation<sup>1</sup>.

To convert the infix expression  $(A+B*C) - (D/E)$  to postfix notation, we follow these steps<sup>12</sup>:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
  - If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.
  - Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered. Remove both '(' and ')'.  
Repeat steps 2-6 until the expression is scanned.
6. Pop and output from the stack until it is not empty.

So, the postfix form of the expression  $(A+B*C) - (D/E)$  is  $ABC*DE/-$ .

#### 10. Infix to prefix conversion by using a Stack. Infix to Postfix conversion by using a Stack.

#### Infix to Prefix Conversion using a Stack<sup>123</sup>:

1. Reverse the infix expression. Note that while reversing, each '(' will become ')' and each ')' becomes '('.
2. Convert the reversed infix expression to "nearly" postfix expression. While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
3. Reverse the postfix expression to get the prefix expression.

#### Infix to Postfix Conversion using a Stack<sup>456</sup>:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. If the scanned character is an operator, then:

If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '('), then push it in the stack.

Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.

4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until the expression is scanned.
7. Pop and output from the stack until it is not empty.

#### 11. Evaluation of prefix expression or Postfix expression.

#### Evaluation of Prefix and Postfix Expressions:

#### Prefix Expression Evaluation<sup>1</sup>:

1. Read the prefix expression from right to left.
2. If the scanned character is an operand, push it to the stack.
3. If the scanned character is an operator, pop two operands from the stack, perform the operation, and push the result back to the stack.
4. Repeat steps 2-3 until all characters are scanned.
5. The final result will be at the top of the stack.

#### Postfix Expression Evaluation<sup>2</sup>:

1. Read the postfix expression from left to right.
2. If the scanned character is an operand, push it to the stack.
3. If the scanned character is an operator, pop two operands from the stack, perform the operation, and push the result back to the stack.
4. Repeat steps 2-3 until all characters are scanned.
5. The final result will be at the top of the stack.

#### 12. Evaluate the postfix expression : $4\ 2\ \$\ 3\ *\ 3 - 8\ 4 / 1\ 1\ + /$

To evaluate the postfix expression  $4\ 2\ \$\ 3\ *\ 3 - 8\ 4 / 1\ 1\ + /$ , you can use a stack and follow these steps<sup>1234</sup>:

1. Read the postfix expression from left to right.
2. If the scanned character is an operand (number), push it onto the stack.
3. If the scanned character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.
4. Repeat steps 2 and 3 until all characters are scanned.
5. The final result will be at the top of the stack.

#### 13. Consider the following arithmetic infix expression Q. Q : $A + (B * C - (D / E ^ F) * G) * H$ Transform a into its equivalent postfix expression.

To convert the infix expression  $A + (B * C - (D / E ^ F) * G) * H$  to postfix notation, we follow these steps<sup>123</sup>:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. If the scanned character is an operator, then:
  - If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '('), then push it in the stack.
  - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until the expression is scanned.
7. Pop and output from the stack until it is not empty.

So, the postfix form of the expression  $A + (B * C - (D / E ^ F) * G) * H$  is  $ABC*DE^F/G*-H*$ .

#### 14. Evaluate the following postfix expression : $P : 5, 6, 2, +, *, 12, 4, /,-$

To evaluate the postfix expression  $5, 6, 2, +, *, 12, 4, /,-$ , you can use a stack and follow these steps<sup>1234</sup>:

1. Read the postfix expression from left to right.
2. If the scanned character is an operand (number), push it onto the stack.
3. If the scanned character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.
4. Repeat steps 2 and 3 until all characters are scanned.
5. The final result will be at the top of the stack.

Following these steps, the given postfix expression evaluates to 37

#### 15. What is Polish Notation?

#### Polish Notation<sup>1234</sup>:

Polish notation, also known as prefix notation, is a mathematical notation in which operators precede their operands<sup>1234</sup>. This is in contrast to the more common infix notation, where operators are placed between operands, and reverse Polish notation (RPN), where operators follow their operands<sup>1234</sup>. It was invented by the logician Jan Lukasiewicz in 1924<sup>1234</sup>.

#### 16. What is the difference between Array & Stack. Difference between Array and Stack<sup>567</sup>.

**Definition:** A stack is a linear data structure represented by a sequential collection of elements in a fixed order<sup>5</sup>. An array is a collection of related data values called elements each identified by an indexed array<sup>5</sup>.

**Principle:** Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list<sup>5</sup>. In the array the elements belong to indexes<sup>5</sup>.

**Operations:** Insertion and deletion in stacks take place only from one end of the list called the top<sup>5</sup>. Insertion and deletion in the array can be done at any index in the array<sup>5</sup>.

**Storage:** The stack has a dynamic size<sup>5</sup>. The array has a fixed size<sup>5</sup>.

**Data Types:** The stack can contain elements of different data types<sup>5</sup>. The array contains elements of the same data type<sup>5</sup>.

**Methods:** There are limited number of operations can be performed on a stack: push, pop, peek, etc<sup>5</sup>. It is rich in methods or operations that can be perform on it like sorting, traversing, reverse, push, pop, etc<sup>5</sup>.

**Pointers:** It has only one pointer- the top<sup>5</sup>. This pointer indicates the address of the topmost element or the last inserted one of the stack<sup>5</sup>. In arrays, memory can be allocated in compile-time and is also known as static memory allocation<sup>5</sup>.

**Complexity Analysis:** Operation Stack Operation Array push O(1) Insert O(n) pop O(1) Delete O(n) peek O(1) Access O(1) isEmpty O(1) Traversal O(n)<sup>5</sup>.

## QUEUE

**1. Define Queue? Explain the array representation of Queue.**  
A Queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order<sup>1</sup>. This means that the element which is first pushed into the queue is the first one to be removed<sup>1</sup>. It is like a line waiting to purchase tickets, where the first person in line is the first person served<sup>1</sup>.

### Array Representation of Queue<sup>2134</sup>.

Queues can be represented using arrays<sup>2134</sup>. In this representation, the queue is implemented using an array<sup>2134</sup>. There are two variables, front and rear, that are implemented in the case of every queue<sup>2134</sup>. front and rear variables point to the position from where insertions and deletions are performed in a queue<sup>2134</sup>. Initially, the value of front and rear is -1 which represents an empty queue<sup>2134</sup>. After inserting an element into the queue, the value of rear will increase while the value of front remains the same<sup>2134</sup>. After deleting an element, the value of front will increase<sup>2134</sup>.

### 2. What is Priority Queue?

A Priority Queue is an abstract data type that behaves similarly to a regular queue or stack data structure, but with a key difference: each element in a priority queue has an associated priority<sup>123</sup>. Elements with higher priority values are typically retrieved before elements with lower priority values<sup>123</sup>.

In a priority queue, elements with high priority are served before elements with low priority<sup>123</sup>. If two elements have the same priority, they are served according to their order in the queue<sup>1</sup>.

### 3. Difference between Input restricted Queue & Output restricted Queue.

### Input Restricted Queue and Output Restricted Queue

are two special cases of a double-ended queue<sup>12</sup>.

**Input Restricted Queue:** In this type of queue, data can be inserted from one end (rear) but can be removed from both ends (front and rear)<sup>1</sup>. This kind of queue does not follow the First In First Out (FIFO) principle<sup>1</sup>. The main operations performed on an input restricted queue

are insertRear(), deleteFront(), and deleteRear()<sup>1</sup>.

**Output Restricted Queue:** In this type of queue, data can be removed from one end (front) but can be inserted from both ends (front and rear)<sup>1</sup>. This kind of queue also does not follow the FIFO principle<sup>2</sup>. The main operations performed on an output restricted queue

are insertRear(), insertFront(), and deleteFront()<sup>2</sup>.

### 4. Write down the algorithm for QUEUE INSERTION & DELETION.

the algorithm for insertion (enqueue) and deletion (dequeue) operations in a queue data structure.

#### Queue Insertion (Enqueue) Algorithm<sup>123</sup>:

1. Check if the queue is full. If it is, return an overflow error.
2. If the queue is not full, insert the new element at the rear of the queue.
3. If the front of the queue is -1 (indicating the queue was empty), set the front of the queue to 0.
4. Increment the rear of the queue.

#### Queue Deletion (Dequeue) Algorithm<sup>2</sup>:

1. Check if the queue is empty. If it is, return an underflow error.
2. If the queue is not empty, access the data element at the front of the queue.
3. Increment the front of the queue.
4. If the front of the queue is greater than the rear after the increment, reset the front and the rear to -1 to show that the queue is empty.

### 5. Write two application of Queue.

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It is used in various applications where things don't have to be processed immediately, but have to be processed in the order they arrived. Here are two applications of Queue:

**Task Scheduling:** Queues can be used to schedule tasks based on priority or the order in which they were received. For example, operating systems often use queues to manage processes and resources<sup>1</sup>.

**Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time. For instance, in printing systems, queues are used to manage the order in which print jobs are processed<sup>1</sup>.

### 6. Write an algorithm for implement the circular Queue using Array

A circular queue is a variant of the queue data structure that uses a circular increment process to optimize memory utilization. When the rear of the queue reaches the end, it wraps around to the start of the queue. Here's a basic algorithm to implement a circular queue using an array:

#### Initialization:

Initialize an array queue of size n, where n is the maximum number of elements that the queue can hold.

Initialize two variables front and rear to -1.

#### Enqueue Operation (Insertion):

1. Increment rear by 1. If rear is equal to n, set rear to 0.

If front is -1 (indicating the queue is empty), set front to 0.

Add the new element at the position pointed to by rear.

#### Dequeue Operation (Deletion):

Check if the queue is empty by checking if front is -1. If it is, return an error message indicating that the queue is empty.

If the queue is not empty, then get the element at the position pointed by front and remove it from the queue.

Increment front by 1 and if front is equal to n, set front to 0.

### 7. What is Circular Queue? Write down the advantage of Circular Queue over Linear Queue

A Circular Queue is a variant of the linear queue data structure where the front and rear ends are connected. This forms a circular structure, hence the name "Circular Queue". The primary advantage of a Circular Queue over a Linear Queue lies in its efficient utilization of memory and its flexibility in performing operations<sup>12345</sup>.

#### Advantages of Circular Queue over Linear Queue:

**Efficient Space Utilization:** In a Circular Queue, once an item is removed, that spot can be filled again, so no space is wasted<sup>15</sup>. This is not the case in a Linear Queue, where insertion is not possible once the rear reaches the last index even if there are empty locations present in the queue<sup>1</sup>.

**Ease of Performing Operations:** In a Linear Queue, FIFO (First In, First Out) is followed, so the element inserted first is the element to be deleted first. This is not the scenario in the case of the Circular Queue as the rear and front are not fixed so the order of insertion-deletion can be changed, which is very useful<sup>1</sup>.

**Support for Cyclic Operations:** The circular nature of this queue allows for continuous enqueue and dequeue operations, making it especially useful for cyclic processes in computer systems<sup>4</sup>.

**Minimizing Memory Wastage:** Compared to a linear queue, a circular queue minimizes memory wastage by wrapping around at the end of the queue<sup>4</sup>.

### 8. What is De-Queue

A Deque, also known as a Double-Ended Queue, is a type of queue in which insertion and removal of elements can be performed from both the front and the rear<sup>1234</sup>. This means it does not strictly follow the First-In-First-Out (FIFO) principle that a regular queue does<sup>4</sup>.

There are two types of deques<sup>12</sup>.

- **Input Restricted Deque:** In this type of deque, insertion can only be performed at one end, but allows deletion from both the ends<sup>12</sup>.
- **Output Restricted Deque:** In this type of deque, deletion can only be performed at one end, but allows insertion at both the ends<sup>12</sup>.

The operations that can be performed on a deque include<sup>12</sup>:

- Insertion at the front
- Insertion at the rear
- Deletion at the front
- Deletion at the rear

## RECURSION

### 1. What is Recursion? Explain the types of Recursion

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem<sup>1</sup>. It involves a function calling itself directly or indirectly<sup>21</sup>. The primary property of recursion is the ability to solve a problem by breaking it down into smaller sub-problems<sup>2</sup>. A recursive function must have a base case or stopping criteria to avoid infinite recursion<sup>2</sup>. There are two primary types of recursion<sup>2345</sup>.

**Direct Recursion:** When a function calls itself directly, it is called direct recursion<sup>24</sup>. This can be further categorized into four types: Tail recursion, Head recursion, Tree recursion, and Nested recursion<sup>2</sup>.

**Indirect Recursion:** Indirect recursion occurs when a function calls another function that eventually calls the original function, forming a cycle<sup>24</sup>. This results in a more complex structure

### 2. Write down the recursive algorithm for factorial of a number.

recursive algorithm for calculating the factorial of a number: PythonAI-generated code. Review and use carefully. More info on FAQ.

```
def factorial(n):  
    # Base case: factorial of 0 or 1 is 1  
    if n == 0 or n == 1:  
        return 1  
    # Recursive case: n! = n * (n-1)!  
    else:  
        return n * factorial(n - 1)
```

### 3. Write down the recursive algorithm for finding GCD of a number

The Greatest Common Divisor (GCD) of two or more integers is the largest positive integer that divides both numbers without leaving any remainder<sup>1</sup>. A recursive algorithm for finding the GCD of two numbers, often referred to as Euclid's algorithm, can be implemented as follows<sup>231</sup>.

PythonAI-generated code. Review and use carefully. More info on FAQ.

```
def gcd(a, b):  
    # Base case: if b is 0, a is the GCD  
    if b == 0:  
        return a  
    # Recursive case: gcd of b and a%b  
    else:  
        return gcd(b, a % b)
```

### 4. Write down the recursion algorithm for Fibonacci Series.

The Fibonacci Series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1<sup>123</sup>. The recursive algorithm for generating the Fibonacci series can be implemented as follows:

PythonAI-generated code. Review and use carefully. More info on FAQ.

```
def fibonacci(n):  
    # Base case: Fibonacci of 0 or 1 is the same number  
    if n <= 1:  
        return n  
    # Recursive case: Fibonacci of n is the sum of the two preceding numbers  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)
```

### 5. Write down the advantages & disadvantages of Recursion

Recursion is a powerful tool in computer science and programming, allowing problems to be solved by breaking them down into smaller sub-problems<sup>12</sup>. However, like any tool, it has its advantages and disadvantages<sup>12345</sup>.

#### Advantages of Recursion:

**Simplicity:** Recursive solutions are often more straightforward and easier to understand than their iterative counterparts<sup>12</sup>.

**Problem Solving:** Recursion is excellent for solving problems that can be broken down into smaller, similar problems<sup>12</sup>.

**Code Readability:** Recursive code can be more readable and easier to understand than iterative code<sup>2</sup>.

#### Disadvantages of Recursion:

**Memory Usage:** Recursive functions can consume more memory as each function call needs to be stored on the stack until the base case is reached<sup>1234</sup>.

**Efficiency:** Recursive functions may be less efficient than iterative solutions in terms of memory and performance<sup>1234</sup>.

**Debugging Difficulty:** Recursive functions can be more challenging to debug and understand than iterative solutions<sup>2</sup>.

#### 6. Which Data Structure can be used to implement Recursion

The data structure used to implement recursion is the **Stack**<sup>1234</sup>. The Stack's Last-In-First-Out (LIFO) property makes it ideal for implementing recursion.

When a recursive function is called, the current function's state, including its variables and the location of any return point, is pushed onto the Stack<sup>1234</sup>. When the recursive call is finished, the function's state is popped from the Stack, and execution continues at the return point<sup>1234</sup>. This allows the function to be suspended and resumed, enabling recursion.

#### 7. The Ackermann Function Problems

The Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive<sup>12</sup>. This means that while the function can be computed by a Turing machine or a general-purpose computer language, it cannot be computed by a primitive recursive function<sup>12</sup>.

The Ackermann function is defined for nonnegative integers m and n and is usually given in the following recursive form<sup>1</sup>:

PythonAI-generated code. Review and use carefully. [More info on FAQ](#).

```
def Ackermann(m, n):
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return Ackermann(m - 1, 1)
    else:
        return Ackermann(m - 1, Ackermann(m, n - 1))
```

The Ackermann function grows very rapidly. For example, Ackermann (3, 3) results in 29, an integer of two decimal digits.

#### 9. Difference between recursive function & iterative function.

A recursive function is one that calls itself within its code to solve a problem, while an iterative function uses loops to repeat some part of the code<sup>1234</sup>. Here are some key differences between them:

**Code Size:** Recursive functions usually have a smaller code size compared to iterative functions. However, iterative functions can have a larger code size but are generally more efficient in terms of time complexity<sup>12</sup>.

**Memory Usage:** Recursive functions require more memory due to the overhead of repeated function calls. Each recursive call is added to the call stack and takes up memory space. On the other hand, iterative functions require less memory as they do not involve such overhead<sup>12</sup>.

**Speed:** Recursive functions are generally slower than iterative functions due to the overhead of repeated function calls and return operations<sup>12</sup>.

**Usage:** If time complexity is the point of focus, and the number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go<sup>1</sup>.

#### 10. Write down the recursive algorithm for Tower of Hanoi.

The Tower of Hanoi is a mathematical puzzle that consists of three rods and a number of disks of different sizes which can slide onto any rod<sup>12345</sup>. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape<sup>12345</sup>. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules<sup>12345</sup>.

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.

No disk may be placed on top of a smaller disk<sup>12345</sup>.

Here is a recursive algorithm to solve the Tower of Hanoi problem:

```
PythonAI-generated code. Review and use carefully. More info on FAQ.
def TowerOfHanoi(n , source, destination,
auxiliary):
    if n==1:
        print("Move disk 1 from
rod",source,"to rod",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary,
destination)
    print("Move disk",n,"from rod",source,"to
rod",destination)
    TowerOfHanoi(n-1, auxiliary, destination,
source)
```

Unit – 4

#### 1. What is non-linear data structure?

**Non-linear Data Structure**<sup>1234</sup>: Non-linear data structures are those where data elements are not arranged in a sequential manner. In these data structures, elements are stored in a hierarchical or a network-based structure that does not follow a sequential order. They do not follow a linear progression or a simple order, unlike linear data structures such as arrays or linked lists<sup>1234</sup>.

#### 2. What is Tree? Write down the properties of a Tree.

**Tree and Its Properties**<sup>56</sup>: A Tree is a connected acyclic undirected graph<sup>5</sup>. There is a unique path between every pair of vertices in the tree<sup>5</sup>. A tree with N number of vertices contains (N-1) number of edges<sup>5</sup>. The vertex which is of 0 degree is called the root of the tree<sup>5</sup>. The vertex which is of 1 degree is called a leaf node of the tree and the degree of an internal node is at least 2<sup>5</sup>. Trees represent hierarchical relationships between individual elements or nodes<sup>56</sup>.

#### 5. How to represent Binary Tree using Array & Linked List

A Binary Tree can be represented in memory using either an Array or a Linked List<sup>12345</sup>.

**Array Representation**<sup>24</sup>: In an array representation of a binary tree, if a node is stored at index i, its left child is stored at index 2\*i+1 and its right child is stored at index 2\*i+2<sup>24</sup>. This representation is efficient for complete binary trees, but for sparse trees, it may lead to wastage of memory space<sup>24</sup>.

**Linked List Representation**<sup>135</sup>: In a linked list representation of a binary tree, each node is a separate object with three fields: one for storing the data and two for storing the references to the left and right child nodes<sup>135</sup>. This representation is more memory-efficient for sparse trees.

#### 6. Explain the different types of Binary Tree. (Definition, Diagram, Example) [Rooted Binary Tree, Full, Complete/Perfect Binary Tree, Almost Binary Tree, Skewed Binary Tree]

**Rooted Binary Tree**<sup>12</sup>: A rooted binary tree is a binary tree in which a special node, known as the root, represents the starting point. Each node in a rooted binary tree has at most two children<sup>12</sup>.

**Full Binary Tree**<sup>34</sup>: A full binary tree, also known as a proper binary tree, is a binary tree in which every node has either zero or two children<sup>34</sup>. This means that no node in a full binary tree has exactly one child<sup>34</sup>.

**Complete/Perfect Binary Tree**<sup>567</sup>: A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible<sup>567</sup>. A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level<sup>567</sup>.

**Almost Binary Tree**<sup>89</sup>: The term "Almost Binary Tree" seems to be a typo or a misunderstanding. It could be referring to the use of binary trees in Artificial Intelligence (AI). Binary trees are used in AI for various purposes, including decision trees<sup>89</sup>.

**Skewed Binary Tree**<sup>101112</sup>: A skewed binary tree is a binary tree in which all the nodes have only either one child or no child. There are two types of skewed binary trees: left-skewed binary tree and right-skewed binary tree<sup>101112</sup>.

#### 7. Define Binary Search Tree with example

A Binary Search Tree (BST) is a node-based binary tree data structure with the following properties<sup>123</sup>:

The left subtree of a node contains only nodes with keys lesser than the node's key<sup>123</sup>.

The right subtree of a node contains only nodes with keys greater than the node's key<sup>123</sup>.

The left and right subtree each must also be a binary search tree<sup>123</sup>.

For example, consider a BST with 8 as the root. The left child of 8 is 3 and the right child is 10. The tree further branches out with 3 having 1 as its left child and 6 as its right child, and so on. This tree follows the BST property where all nodes in the left subtree of a node are less than the node, and all nodes in the right subtree are greater.

#### 8. Problems on Binary Search Tree(BST).

Problems on Binary Search Trees (BSTs) often involve various operations such as insertion, deletion, and searching<sup>12</sup>. Here are some common problems:

- Insertion in a BST**<sup>12</sup>: Given a BST and a key, the task is to insert the key in the BST at the correct position to maintain the BST property<sup>12</sup>.
- Search a given key in BST**<sup>12</sup>: Given a BST and a key, the task is to find the node with the given key in the BST<sup>12</sup>.
- Deletion from BST**<sup>12</sup>: Given a BST and a key, the task is to delete the node with the given key from the BST<sup>12</sup>.
- Determine whether a given binary tree is a BST or not**<sup>12</sup>: Given a binary tree, the task is to determine if it is a BST<sup>12</sup>.
- Find k'th smallest and k'th largest element in a BST**<sup>12</sup>: Given a BST and an integer k, the task is to find the k'th smallest and k'th largest element in the BST<sup>12</sup>.

**Find the Lowest Common Ancestor (LCA) of two nodes in a BST**<sup>12</sup>: Given a BST and two nodes, the task is to find the LCA of the two nodes<sup>12</sup>.

#### 9. What is B Tree? How to insert and delete node in a B Tree?

**B-Tree**<sup>1234</sup>: A B-Tree is a self-balancing tree data structure that maintains sorted data and allows for efficient searches, sequential access, insertions, and deletions<sup>1234</sup>. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height<sup>1</sup>. This shallow height leads to less disk I/O, which results in faster search and insertion operations<sup>1</sup>. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs<sup>1</sup>.

**Insertion in a B-Tree**<sup>5</sup>: Insertion in a B-Tree is performed at the leaf node level. The algorithm first traverses the B-Tree to find the appropriate leaf node where the key can be inserted. If the leaf node has space for the key, it is inserted there. If the leaf node is full, it is split into two nodes, and the median key is moved up to its parent node<sup>5</sup>.

**Deletion in a B-Tree**<sup>67</sup>: Deletion in a B-Tree involves searching for the node where the key to be deleted exists, deleting the key, and balancing the tree if required<sup>6</sup>. If the key to be deleted is in an internal node, it is replaced by its inorder predecessor or successor<sup>6</sup>. If deleting a key causes a node to have fewer than the minimum number of keys, the tree is rebalanced by borrowing a key from a sibling node or merging nodes<sup>6</sup>.

#### 10. Problems on B-Tree?

Problems on B-Trees often involve various operations such as insertion, deletion, and searching<sup>1</sup>. Here are some common problems:

**Insertion in a B-Tree**<sup>1</sup>: Given a B-Tree and a key, the task is to insert the key in the B-Tree at the correct position to maintain the B-Tree property<sup>1</sup>.

**Search a given key in B-Tree**<sup>1</sup>: Given a B-Tree and a key, the task is to find the node with the given key in the B-Tree<sup>1</sup>.

**Deletion from B-Tree**<sup>1</sup>: Given a B-Tree and a key, the task is to delete the node with the given key from the B-Tree<sup>1</sup>.

**Determine whether a given binary tree is a B-Tree or not**<sup>1</sup>: Given a binary tree, the task is to determine if it is a B-Tree<sup>1</sup>.

#### 11. What is AVL Tree? Explain the different rotation of a AVL Tree?

**AVL Tree**<sup>1234</sup>: An AVL tree, named after its inventors Georgy Adelson-Velsky and Evgenii Landis, is a self-balancing binary search tree<sup>1234</sup>. In an AVL tree, the heights of the two child subtrees of any node differ by at most one<sup>1234</sup>. If at any time they differ by more than one, rebalancing is done to restore this property<sup>1234</sup>.

**Rotations in AVL Tree**<sup>15678</sup>: An AVL tree may perform four kinds of rotations to keep itself balanced<sup>15678</sup>:

**Left Rotation (LL Rotation)**: Performed when a node is added into the right subtree of the right subtree, and the tree gets out of balance<sup>15</sup>.

**Right Rotation (RR Rotation)**: Performed when a node is added to the left subtree of the left subtree, and the tree gets out of balance<sup>15</sup>.

**Left-Right Rotation (LR Rotation)**: A combination in which first left rotation takes place, followed by a right rotation<sup>15</sup>.

**Right-Left Rotation (RL Rotation)**: A combination in which first right rotation takes place, followed by a left rotation<sup>15</sup>. These rotations help the AVL tree maintain its balance, ensuring efficient operations<sup>15678</sup>.

#### 12. Problems on AVL Tree

Problems on AVL Trees often involve various operations such as insertion, deletion, and searching<sup>1</sup>. Here are some common problems:

**Insertion in an AVL Tree**<sup>1</sup>: Given an AVL Tree and a key, the task is to insert the key in the AVL Tree at the correct position to maintain the AVL Tree property<sup>1</sup>.

**Search a given key in AVL Tree**<sup>1</sup>: Given an AVL Tree and a key, the task is to find the node with the given key in the AVL Tree<sup>1</sup>.

**Deletion from AVL Tree**<sup>1</sup>: Given an AVL Tree and a key, the task is to delete the node with the given key from the AVL Tree<sup>1</sup>.  
**Determine whether a given binary tree is an AVL Tree or not**<sup>1</sup>: Given a binary tree, the task is to determine if it is an AVL Tree<sup>1</sup>.

**13.What is Tree traversal? Write down the concept of preorder, inorder, postorder traversal.**

**Tree Traversal**<sup>12345</sup>: Tree traversal is the process of systematically visiting each node in a tree data structure. It involves exploring the nodes in a specific order, such as inorder, preorder, or postorder, to perform operations or retrieve information<sup>12345</sup>.

**Preorder Traversal**<sup>6718</sup>: In preorder traversal, the nodes are visited in the following sequence: Root->Left->Right<sup>6718</sup>. The algorithm starts by visiting or printing the root node, and then traversing the left subtree. Once the left subtree has been fully traversed, the algorithm then moves on to traverse the right subtree<sup>6718</sup>.

**Inorder Traversal**<sup>971011</sup>: In inorder traversal, the nodes are visited in the following sequence: Left->Root->Right<sup>971011</sup>. The algorithm starts by traversing the left subtree, then visiting or printing the root node, and finally traversing the right subtree<sup>971011</sup>.

**Postorder Traversal**<sup>1213114</sup>: In postorder traversal, the nodes are visited in the following sequence: Left->Right->Root<sup>1213114</sup>. The algorithm starts by traversing the left subtree, then moving on to traverse the right subtree. Once the right subtree has been fully traversed, the algorithm then visits the root node<sup>1213114</sup>.

**15.Write a recursive algorithm for Binary Tree traversal with example**

Binary Tree traversal is a process of visiting each node in a tree data structure exactly once in a systematic way<sup>1234</sup>. Here are the recursive algorithms for three common types of binary tree traversals:

**Preorder Traversal**<sup>1</sup>: In a preorder traversal, the root node is visited first, then the left subtree, and finally the right subtree<sup>1</sup>. Here is a recursive algorithm for preorder traversal:

PythonAI-generated code. Review and use carefully. [More info on FAQ](#)

```
def Preorder(node):
    if node is not null:
        print(node.value)
        Preorder(node.left)
        Preorder(node.right)
```

**Inorder Traversal**<sup>1234</sup>: In an inorder traversal, the left subtree is visited first, then the root node, and finally the right subtree<sup>1234</sup>. Here is a recursive algorithm for inorder traversal:

PythonAI-generated code. Review and use carefully. [More info on FAQ](#)

```
def Inorder(node):
    if node is not null:
        Inorder(node.left)
        print(node.value)
        Inorder(node.right)
```

**Postorder Traversal**<sup>1</sup>: In a postorder traversal, the left subtree is visited first, then the right subtree, and finally the root node<sup>1</sup>. Here is a recursive algorithm for postorder traversal:

PythonAI-generated code. Review and use carefully. [More info on FAQ](#)

```
def Postorder(node):
    if node is not null:
        Postorder(node.left)
        Postorder(node.right)
        print(node.value)
```

**16.What are the purpose of AVL tree.**

An AVL tree, named after its inventors Georgy Adelson-Velsky and Evgenii Landis, is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one<sup>12</sup>. The purpose of an AVL tree is to maintain this balance in order to provide efficient operations<sup>12</sup>. Here are some of the main purposes of AVL trees<sup>13</sup>:

**Efficient Searching**: AVL trees are used to index large records in a database and to efficiently search within that data<sup>3</sup>.

**In-Memory Collections**: AVL trees are used for all types of in-memory collections, including sets and dictionaries<sup>3</sup>.

**Database Applications**: AVL trees are used in database applications where insertions and deletions are less frequent, but there are frequent data lookups<sup>3</sup>.

**Optimized Search**: AVL trees are used in applications that require optimized search<sup>3</sup>.