Page No.
Date

Interview Question

Q. 1. What do you mean by Data structures?

→ DS are specialized formats for organizing processing storing and retrieving data. they enable efficient data management and manipulation, allowing for optimal performance in various computational tasks.

Q. 2 Explain self - referential structure? Explain its need?

→ A self - referential structure is a data structure that contains a reference (or pointer) to itself.
this is commonly seen in linked list, trees, and graphs
the need arises in scenarios where dynamic data allocation is necessary, allowing data structures to grow and shrink in size as needed.

Q. 3 List out the areas in which data structures are applied extensively?

→ Databases (eg B-trees)
Networking (eg routing tables)
Operating Systems (eg queues for Process Scheduling)
Artifical Intelligence (eg tree for search algorithm)
Compilers (eg syntax trees)
Graphics (Spatial data structure)

Q. 4 Explain DS used to perform recursion?

→ The primary data structure used to perform recursion is the call stack.

It stores information about the active subroutines or function calls, including local variables, parameters and the return address.

Q.5 List different types of DS?

1) Arrays
2) Linked lists
3) Stacks
4) Queues
5) Trees (Binary trees, AVL trees, etc)
6) Graphs
7) Hash Tables

Q.6 What is the difference between iterative method and recursive method? which one is feasible?

→ Iterative Method - Uses loops (for, while) to repeat operations. Generally consumes less memory and can be more efficient for problems where the depth of recursion can lead to stack Overflow.

Recursive method - A function call itself to solve smaller subproblems naturally defined in terms of smaller instance (eg factorial)

Feasibility - Depends on the prob recursion is more intuitive for divide and conquer prob, while iteration is often more memory efficient.

Page No.
Date

What are the types of Recursion?

→ Direct Recursion - A function directly calls itself.
Indirect Recursion - A function calls another function which in turn calls the original function
Tail Recursion - The Recursive call is the last statement in the function allowing optimizations by the compiler.

i) Write a Java Program for String Reversal.

→
```
Public class StringReversal {
    Public static String reverse (String str){
        if (str.isEmpty()) {
            return str;
        }
        return reverse (str.substring(1)) +
        str.charAt(0);
    }
    main m {
        String original = "Hello";
        String reversed = reverse (original);
        SOP(reversed); // olleH
    }
}
```

Q. g) What is stack DS? the appln of stack.

→ Stack is a linear DS follows LIFO principal. last element added is the first to be removed

Appln. function call management (call stack)
Expression evaluation and syntax parsing.

Page No.
Date

Backtracking algorithm
Undo mechanism in application

Q. 10 What are different operatn available in stack DS?
→ Push : Add an ele to the TOP
POP : Remove the top ele
Peek / TOP : Get the top ele w/o removing it.
isEmpty : check if the stack is empty
Size : Return the no of ele in the stack

Q. 11 Explain tower of hanoi ?
→ The TOH is a classic puzzle that involves moving a set of disks from one peg to another, using a third peg as an auxilliary following these rules:

only one disk can be moved at a time
A disk can only be moved if it is the uppermost disk on a peg
No disk may be placed on top of a smaller disk.

Q. 12 Which DS is ideal to perform Recursion operation and why ?
→ The call stack is ideal for recursion as it automatically manages function calls and local variables, enabling the system to return to the previous state after a function completes.

Q. 13. Briefly write in brief the approches to develop algorithm?
→ Brute force - Trying all possibilities to find

a solution

Divide and Conquer : Dividing the prob into smaller subproblems, solving them independently, and combining result

Dynamic Programming - Breaking down problems into simpler subproblems and storing results to avoid redundant computations.

Greedy Algorithms - making locally optimal choices at each step with the hope of finding a global optimum.

Q.14) What is Backtracking ?

→ Backtracking is an algorithmic technique for solving problems incrementally by trying partial solutions and abandoning them if they fail to satisfy the conditions of the problem, effectively exploring all possibilities.

Q.15 What is the difference between backtracking and recursion ?

→ Recursion : A programming technique where a function calls itself to solve subproblems, often leading to a single path of execution.

Backtracking : A more strategic form of recursive where multiple paths are explored, and invalid paths are abandoned (pruned), allowing for a search through a solution space.

Q.1 Explicit and Implicit Backtracking -

→ 1. In Explicit Backtracking - the algorithm actively tracks the decision making. Process and explicitly backtracks to previous state when a dead end or infeasible soln is a stack or recursion to store and restore previous choices.

2 Implicit Backtracking -

Implicit backtracking also known as chronological backtracking doesn't explicitly store previous choices, instead it relies on the problem's structure to automatically backtrack when necessary.

This often happens in algorithm like depth-first search, where the search path is implicitly traced and reversed when a dead-end is reached.

Q.2 Greedy Algorithms

→ GA are algorithms that make locally optimal choices at each step, hoping to achieve a globally optimal solution. They often work well for problems where the optimal solution can be constructed by making a series of locally optimal decisions.

Q.3 Example of Greedy Algorithms?

→ 1 Dijkstra's Algorithm - finds the shortest path between two nodes in a graph.

2 Prim's algorithm - constructs a minimum spanning tree of a weighted graph.

3. K mi gr

4. Hu va the

Q. 4 →

Q. 5 →

Q. 5 →

Q. 6 →

3. Kruskal's algorithm - constructs a minimum spanning tree of a weighted graph.

4. Huffman coding - compresses data by assiging variable-length codes to characters based on their frequency.

Q.4 Difference Between Greedy and Brute force?
→ GA : make locally optimal choices at each step, aiming for a globally optimal solut n. They can be efficient but might not always find the best solution
BF : Ex exhaustively explores all possible solutions, guaranteeing the optimal soln but can be computationally expensive for large problems

Q.5 Stack
→ Stack is linear DS.
follow LIFO
Elements are added and removed from top.

Q.5 Applications of stack
→ Function calls and returns.
Expression evaluation.
Backtracking algorithms.
Undo / redo Operations.

Q.6 Infix to postfix Conversion and Evalution?
→
```
import java.util.stack;
Public class InfixToPostfix {
    String infix = "2+3*4";
    String postfix = infixToPostfix(infix);
    SOPln("Postfix:" + postfix);
    int result = evaluatePostfix(postfix);
```

```
System.out.println("Result : " + result);
}
private static String infixToPostfix
        (String infix) {
Stack<Character> stack = new
        Stack<>();
StringBuilder postfix = new StringBuilder();

for (char c : infix.toCharArray()) {
    if (Character.isDigit(c))
{
        postfix.append(c);
    } else if (c == '(') {
        stack.push(c);
    } else if (c == ')') {
        while (!stack.isEmpty() &&
stack.peek() != '(') {
        postfix.append(stack.pop());
    }
    stack.pop(); // pop the '('
    } else
{ // Operator
        while (!stack.isEmpty() &&
precedence(c) <= precedence
        (stack.peek())) {
    postfix.append(stack.pop());
}
    stack.push(c);
}
}
    while (!stack.isEmpty()) {
    postfix.append(stack.pop());
} return postfix.toString();
}
```

```
private static int evaluatePostfix (String postfix)
{Stack<Integer> stack = new Stack<>();
  for (char c : postfix.toCharArray()){
    if (Character.isDigit(c)){
      Stack.push(Integer.parseInt(
      String.valueOf(c)));
    }else{
      int operand2 = stack.pop();
      int operand1 = stack.pop();
      switch(c){
        case '+':
          stack.push(operand1 + operand2);
          break;
```

Q.7 Queue -

→ A queue is linear DS that FIFO Elements are added to the rear and removed from the front.

Q.8 Types of Queue -

→ Simple queue, circular queue, priority queue, Double ended queue (deque)

Q.8 Queue Full condition for Circular Queue.

→ In a circular queue, the queue is considered full when the front and rear pointer are equal, and the next available index is occupied.

Q.9 Implementing Two stacks using a Single Array

→ You can allocate half of the array to one stack and the other half to the other stack. Use two pointers to track the top elements of each stack.

Q.10 Implementing stack and Queue using linked list.

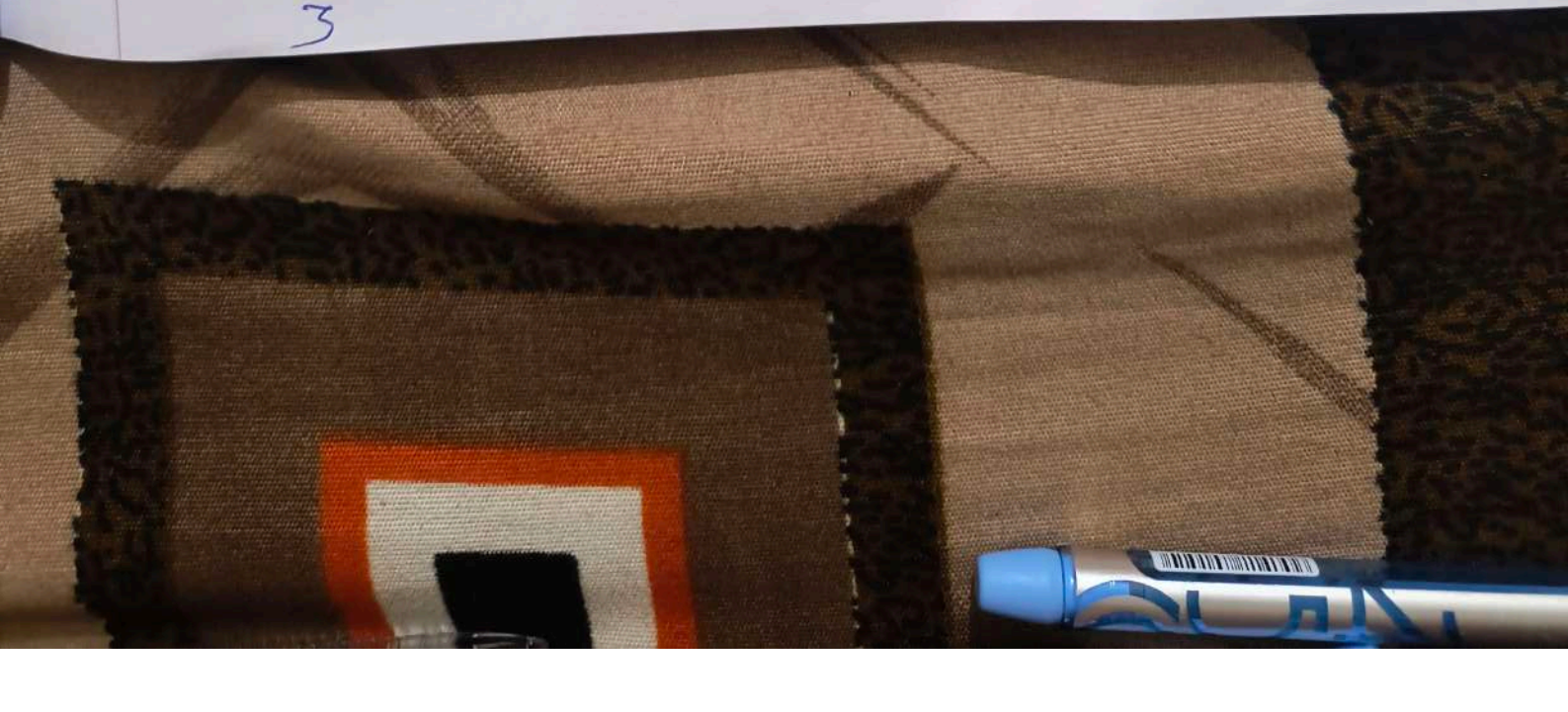
```
new_node.prev Prev_node.next = prev_node;
        if (new_node.next != null)
            new_node.next.prev = new_node;
}

case '-':
    stack.push (operand1 - operand2);
    break;

case '*':
    stack.push (operand1 * operand2);
    break;

case '/':
    stack.push (operand1 / operand2);
    break;

}
}
}
    return stack.pop();
}
}
```



→ Stack - Use a singly LL where top of ele at the head
Queue - Use a singly LL where the front is at the head and the rear is at the tail.

Q.11 Shortcomings of Array over LL;
→ You can allocate half of the array to one stack and the other half to the other stack. Use two pointers to track the top elements of each stack.

Q.12 Linked list
→ A LL is linear DS where elements are connected by pointers.
each element called a node
Contains data and a pointer to the next node.

Q.13 Reversing LL write program ?
→
```
public class LL {
    Node head;
    static class Node {
        int data;
        Node next;
    Node (int d) {
        data = d;
        next = null;
    }
}

public void reverse() {
    Node prev = null;
    Node current = head;
    Node next = null;
```

Q.14 Re
wr
→

Q.15
→

```
while (current != null) {
    next = current.next;
    current.next = prev;
    prev = current;
    current = next;
}
    head = prev;
}
```

Q.14 Reversing LL Using Single pointer write program ?

→
```
Public void reverse () {
    Node current = head;
    while (current != null && current.next
    != null) {
        Node next = current.next;
        current.next = next.next;
        next.next = head;
        head = next;
    }
}
```

Q.15 If you are using C language to implement Heterogeneous LL in C what pointer type you use

→ In C you can use void* pointer to store elements of different data types in a heterogenous LL. This allow you to create a flexible LL that can hold various type of data.