

# DATA STRUCTURES AND ALGORITHMS



## DSA QUESTIONS/ANSWERS

### Questions [KG Sir]

1. How can we determine that an Algorithms is efficient or not? List out the measures.
2. What are time and space complexity?
3. What are Asymptotic notation? Explain Big O notation and its importance.
4. What are data structures and explain its operations.
5. What is Linked list? Difference between Linked List and Array.
6. Implement the insertion and deletion operation in singular, doubly, circular linked list.
7. Reverse a string using stack.
8. Check a given number is palindrome or not using stack.
9. convert the given infix expression to postfix expression.
10. What is the importance of converting infix expression to postfix expression.
11. Implement the queue using Array.
12. Implement the queue using Linked List.
13. What is advantage of Circular Queue and DE Queue over queue.
14. Find the factorial of number using recursion.
15. Find the nth term of Fibonacci using recursion.
16. Print the nodes of linked list in reverse order.
17. Count the nodes of linked list using recursion.
18. What is recursion and differentiate between recursion and loop.

## Solutions

### ▼ Solution 1

The time and space an algorithm uses, are the two major measures of the efficiency of an algorithm.

#### 1. Time Efficiency →

The time required by the algorithms to perform a certain task.

The Quicker the work is done, the better.

Time depends on several factors such as amount of data, speed of compute...

#### 2. Space Efficiency →

The memory required by the algorithms while a process is being performed.

Components of Space use are:

1. Instruction space
2. Data Space
3. run-time stack space.

### ▼ Solution 2

Time and Space complexity are the measures to compare the working of an algorithms and to measure the efficiency of it.

Time Complexity → Tells us about the time usage by an algorithm for an input size of  $n$ .

There are two approaches for time complexity measurement

1. Asymptotic categorization: This gives the general idea about the time usage of an algorithm for an input size  $n$ .
2. Estimation of running time: this is done by analyzing the code
  - Operation count → Select operation that occur most frequently and calculate their occurrences.
  - Step count → Determine total number of steps, which are executed by the program.

The time complexity has 3 cases: Worst Case, Average Case, Best Case

Asymptotic categorization eliminates hardware from the consideration and expresses efficiency in terms of data size,  $N$ .

There are several notations such as: Big O, Omega, Theta, Little O etc.

Space Complexity → This tells us about the space/memory usage of an algorithm. Now-a-days the space complexity is of little use as memory is cheap now.

### ▼ Solution 3

**Asymptotic Notation:** It is used to describe the running time of an algorithm in terms of the input size  $n$ . There are three different notations: big O, big Theta ( $\Theta$ ), and big Omega ( $\Omega$ ).

**Big O Notation:**

The Big-O notation describes the worst-case running time of a program. We compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of N.

Several Big O Complexities:

1. Linear Search:  $O(n)$
2. Binary Search:  $O(\log n)$
3. Bubble Sort:  $O(n^2)$
4. Merge-Sort:  $O(n \log n)$

**▼ Solution 4**

**Data Structure:** The logical or mathematical model of a particular organization of data is called a data structure. There are several types of data structure such as linear or non-linear. Example:

1. Array: This is the simplest, linear data structure. The data is stored in the linear fashion, one after another. The data is stored in the Random Access Memory.
2. Linked List: Another type of DS which holds memory in form of Nodes. Each Node holds some data and has a pointer to the next node.
3. Trees: This DS is used to represent hierarchical storing of data and in this DS, each node can have multiple child nodes.

and there are other data Structures as well.

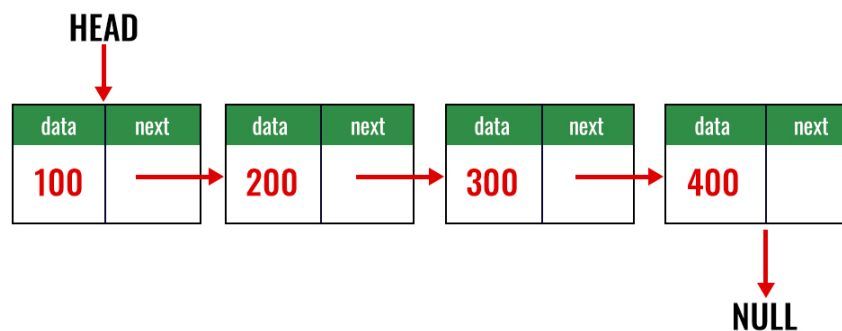
**Operations:** The operations are the actions that we perform on the data which is stored in the Data Structure. Some Frequently used operations are:

1. Traversing: Accessing each record in the DS exactly one so that certain items in the record may be processed. It is also called "Visiting".
2. Searching: Finding the location of the record with a given key value or finding the location of all records which satisfy one or more conditions.
3. Inserting: Adding a new record to the structure.
4. Deleting: Removing a record from the structure.

**▼ Solution 5****Linked List:**

- It is a one-way list and is a linear collection of data elements called *Nodes* whereas the linear order is maintained by means of pointer.

- Each node is divided into two parts:  
*Data* which holds the data in the structure and  
*Link Field* which holds the link to the next pointer.
- It is a dynamic data structure.
- Memory allocation is runtime and no upper bound on size.
- There are several types of Linked Lists such as: Doubly Linked List, Circular Linked List.



Linked List representation

### ***Difference b/w Linked List and Array:***

S No.	Linked List	Array
1.	It is Dynamic data structure,	It is Static data structure.
2.	Insertion and deletion of elements are easy	Insertion and Deletion of elements is difficult.
3.	Random Access is not allowed	Random Access is Allowed
4.	Slower execution time	Faster Execution time.

### ***Code:***

```

import java.util.Scanner;

class Node {
    int rollNo;
    Node next;
}

class LinkedList {
    Node START;

    void insert(int data) {
        Node newNode = new Node();
        newNode.rollNo = data;
    }
}
  
```

```

        if(START == null)
            START = newNode;
        else {
            Node current = START;
            while(current.next != null)
                current = current.next;
            current.next = newNode;
        }
    }

    void traverse() {
        if(START == null)
            System.out.println("List Empty!!");
        else {
            Node current;
            for (current = START; current != null; current = current.next)
                System.out.print(" " + current.rollNo);
        }
    }

    public static void main(String[] args) {
        SinglyLinkedList obj = new SinglyLinkedList();
        Scanner sc = new Scanner(System.in);

        obj.insert(10);
        obj.insert(20);
        obj.traverse();
    }
}

```

## ▼ Solution 6

Implementations of Insertion and Deletion operation Linked List.

## ▼ Singular LL

```

class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}

public class LinkedList {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if(head == null)
            head = newNode;
        else {
            Node temp = head;
            while(temp.next!=null)
                temp = temp.next;
            temp.next = newNode;
        }
    }
}

```

```

    }
}

public void delete() {
    if(head != null) {
        System.out.println("Data deleted : " + head.data);
        head = head.next;
    }
    else
        System.out.println("Empty List!!");
}

public void display() {
    Node temp = head;
    if(head!=null)
        while(temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    else
        System.out.print("Empty List!!");

    System.out.println();
}

public static void main(String[] args) {
    LinkedList LL = new LinkedList();
    LL.insert(10);
    LL.insert(20);
    LL.display();
    LL.delete();
    LL.display();
}
}

```

## ▼ Doubly LL

```

class Node {
    int data;
    Node next;
    Node prev;
    Node(int data) {
        this.data = data;
    }
}

public class DoublyLL {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if(head == null) {
            head = newNode;
        }
        else {
            Node temp = head;
            while(temp.next!=null)

```

```

        temp = temp.next;
        temp.next = newNode;
        newNode.prev = temp;
    }
}

public void delete() {
    if(head != null) {
        System.out.println("Data deleted : " + head.data);
        if(head.next == null)
            head = null;
        else {
            head = head.next;
            head.prev = null;
        }
    }
    else
        System.out.println("Empty List!!");
}

public void display() {
    Node temp = head;
    if(head!=null)
        while(temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    else
        System.out.print("Empty List!!");

    System.out.println();
}

public static void main(String[] args) {
    DoublyLL LL = new DoublyLL();
    LL.insert(10);
    LL.insert(20);
    LL.display();
    LL.delete();
    LL.display();
}
}

```

## ▼ Circular LL

```

class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}

public class CircularLL {

    Node head;
}

```

```

public void insert(int data) {
    Node newNode = new Node(data);
    if(head == null) {
        head = newNode;
        head.next = head;
    }
    else {
        Node temp = head;
        while(temp.next != head)
            temp = temp.next;
        temp.next = newNode;
        newNode.next = head;
    }
}

public void delete() {
    if(head != null) {
        System.out.println("Data deleted : " + head.data);
        if(head.next == head)
            head = null;
        else {
            Node temp = head;
            while(temp.next != head)
                temp = temp.next;
            head = head.next;
            temp.next = head;
        }
    }
    else
        System.out.println("Empty List!!");
}

public void display() {
    Node temp = head;
    if(head != null) {
        if(head.next != head) {
            while(temp.next != head) {
                System.out.print(temp.data + " ");
                temp = temp.next;
            }
            System.out.print(temp.data + " ");
        }
        else
            System.out.print(head.data + " ");
    }
    else
        System.out.print("Empty List!!");
    System.out.println();
}

public static void main(String[] args) {
    CircularLL LL = new CircularLL();
    LL.insert(10);
    LL.insert(20);
    LL.display();
    LL.delete();
    LL.display();
}
}

```



### ▼ Solution 7

Reversing a string using Stack

```
import java.util.Scanner;

class Node {
    char c;
    Node next;
    Node(char c) {
        this.c = c;
    }
}

public class Main {

    Node top;

    public void push(char c) {
        Node newNode = new Node(c);
        if(top == null)
            top = newNode;
        else {
            newNode.next = top;
            top = newNode;
        }
    }

    public char pop() {
        if(top == null)
            return '\n';
        else {
            char c = top.c;
            top = top.next;
            return c;
        }
    }

    public static void main(String[] args) {
        Main stack = new Main();
        Scanner sc = new Scanner(System.in);
        String str = sc.next();
        for (char c : str.toCharArray() )
            stack.push(c);
        for (int i=0; i < str.length() ; i++)
            System.out.print(stack.pop());
    }
}
```

### ▼ Solution 8

Check if a number is palindrome or not using Stack.

```
import java.util.Scanner;
```

```

class Node {
    char c;
    Node next;
    Node(char c) {
        this.c = c;
    }
}

public class Main {

    Node top;

    public void push(char c) {
        Node newNode = new Node(c);
        if(top == null)
            top = newNode;
        else {
            newNode.next = top;
            top = newNode;
        }
    }

    public char pop() {
        if(top == null)
            return '\n';
        else {
            char c = top.c;
            top = top.next;
            return c;
        }
    }

    public static boolean isPalindrome(String str) {
        Main stack1 = new Main();
        Main stack2 = new Main();

        int len = str.length();
        for (int i=0; i<len/2 ; i++) {
            stack1.push(str.charAt(i));
            stack2.push(str.charAt(len-1-i));
        }
        boolean flag = true;
        for (int i=0; i<len/2 ; i++) {
            if(stack1.pop() != stack2.pop()) {
                flag = false;
                break;
            }
        }

        return flag;
    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        String str = sc.next();
        var flag = isPalindrome(str);

        if(flag)
            System.out.println("Palindrome!");
        else
            System.out.println("Not Palinedrome!");
    }
}

```

```
}  
}
```

#### ▼ Solution 9

<https://www.javatpoint.com/convert-infix-to-postfix-notation>

You can read it here.

#### ▼ Solution 10

While writing a mathematical expression, when we write an infix expression the operator are in between the operands and thus it was difficult for the computer to evaluate the expression as it needs to follow the law of associativity and operator precedence. But in a postfix expression the computer does not have to follow the associativity or precedence law, rather it only performs the operation as they arrive. This saves lot of computational power and coding challenges while writing the program.

We convert the infix expression to postfix expression using the stack data structure.

Infix → a + b

Postfix → ab+

#### ▼ Solution 11

Queue Implementation using Array

```
public class Main {  
  
    int[] ar;  
    int front;  
    int rear;  
  
    final int size = 5;  
  
    Main() {  
        ar = new int[size];  
        front = rear = -1;  
    }  
  
    public void enqueue(int inp) {  
        if(rear == -1) {  
            front = rear = 0;  
            ar[rear] = inp;  
        }  
        else {  
            if(rear+1 == size)  
                System.out.println("Queue Full!");  
            else {  
                rear++;  
                ar[rear] = inp;  
            }  
        }  
    }  
  
    public void dequeue() {
```

```

        if(rear == -1) {
            System.out.println("Empty Queue!!");
        }
        else {
            System.out.println("data : " + ar[front]);
            if(front == rear)
                front = rear = -1;
            else if(front < rear)
                front++;
        }
    }

    public void view() {
        if(rear != -1)
            for (int i=front; i<=rear; i++)
                System.out.print(ar[i] + " ");
        else
            System.out.print("Empty Queue!!");
        System.out.println();
    }

    public static void main(String[] args) {
        Main Queue = new Main();
        Queue.enqueue(10);
        Queue.enqueue(20);
        Queue.view();
        Queue.dequeue();
        Queue.view();
        Queue.enqueue(30);
        Queue.view();
    }
}

```

## ▼ Solution 12

### Queue implementation using Linked List

```

class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}

public class Main {

    Node front;
    Node rear;

    public void enqueue(int inp) {
        Node newNode = new Node(inp);
        if(front == null) {
            front = newNode;
            rear = front;
        }
        else {
            rear.next = newNode;
            rear = newNode;
        }
    }
}

```

```

    }
}

public void deQueue() {
    if(front != null) {
        System.out.println("data : " + front.data);
        if(front == rear)
            front = rear = null;
        else {
            front = front.next;
        }
    }
    else
        System.out.println("Empty List!!");
}

public void display() {
    if(front != null) {
        Node temp = front;
        while(temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }
    else
        System.out.print("Empty Queue!!");
    System.out.println();
}

public static void main(String[] args) {
    Main Queue = new Main();
    Queue.enqueue(10);
    Queue.enqueue(20);
    Queue.display();
    Queue.deQueue();
    Queue.display();
    Queue.enqueue(30);
    Queue.display();
}
}

```

### ▼ Solution 13

#### ***Circular Queue:***

- Easier insertion and deletion in the circular queue, as rear and front are connected so newer element can be inserted from the front of the queue again.
- Efficient Utilization of Memory as there is no memory wastage as newer elements occupy previously empty state.

#### ***Deque:***

- Easier insertion and deletion in the deque as elements can be inserted and deleted from both (front and rear) ends.
- The deque supports clockwise and anticlockwise rotations in constant time.

#### ▼ Solution 14

Factorial program using Recursion

```
public class Main {  
  
    public static int recurFact(int num) {  
        if(num == 1 || num == 0)  
            return 1;  
        else  
            return num * recurFact(num-1);  
    }  
  
    public static void main(String[] args) {  
        System.out.println(recurFact(5));  
    }  
}
```

#### ▼ Solution 15

nth Fibonacci term program using recursion:

```
import java.util.Scanner;  
  
public class Main {  
  
    public static int recurFibo(int num) {  
        if(num == 0)  
            return 0;  
        else if(num == 1 || num == 2)  
            return 1;  
        else  
            return recurFibo(num-1) + recurFibo(num-2);  
    }  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter position : ");  
        int num = sc.nextInt();  
        System.out.println("Value : " + recurFibo(num-1));  
    }  
}
```

#### ▼ Solution 16

Reversing the Linked List

```
import java.util.Scanner;  
  
class Node {  
    int data;  
    Node next;
```

```

    Node(int data) {
        this.data = data;
    }
}

public class Main {

    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if(head == null)
            head = newNode;
        else {
            Node temp = head;
            while(temp.next != null)
                temp = temp.next;
            temp.next = newNode;
        }
    }

    public void reverseTheLL() {

        Node currNext = null;
        Node curr = head;
        Node prev = null;

        while(curr != null) {
            currNext = curr.next;
            curr.next = prev;
            prev = curr;
            curr = currNext;
        }
        head = prev;
    }

    public void display() {
        if(head != null) {
            Node temp = head;
            while(temp != null) {
                System.out.print(temp.data + " ");
                temp = temp.next;
            }
        }
        else
            System.out.print("Empty List!!");
        System.out.println();
    }

    public static void main (String[] args) {
        Main LL = new Main();

        for (int i=1; i<=5; i++)
            LL.insert(i);

        LL.reverseTheLL();
        LL.display();
    }
}

```

## ▼ Solution 17

### Count nodes in Linked List using recursion

```
import java.util.Scanner;

class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}

public class Main {

    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if(head == null)
            head = newNode;
        else {
            Node temp = head;
            while(temp.next != null)
                temp = temp.next;
            temp.next = newNode;
        }
    }

    public int countNodes(Node curr) {
        if(curr == null)
            return 0;
        return 1 + countNodes(curr.next);
    }

    public void display() {
        if(head != null) {
            Node temp = head;
            while(temp != null) {
                System.out.print(temp.data + " ");
                temp = temp.next;
            }
        }
        else
            System.out.print("Empty List!!");
        System.out.println();
    }

    public static void main (String[] args) {
        Main LL = new Main();

        for (int i=1; i<=5; i++)
            LL.insert(i);

        System.out.println(LL.countNodes(LL.head));
        LL.display();
    }
}
```




### ▼ Solution 18

**Recursion:** When a function calls itself within the function, it is called recursion.

S No.	Recursion	Looping
1.	Recursion is a method of calling a function within the same function.	Loop is a control structure that allows executing a block of code repeatedly within the program.
2.	It is slower	It is faster
3.	Recursion uses stack to store variables	Loop does not uses stack.
4.	More Readable	Less Readable

### GitHub Repository

GitHub - aryan-upa/Java\_DSA: This repository is the collection of Java programs done by me during the 4th Semester while learning Data Structures and Algorithms.

 [https://github.com/aryan-upa/Java\\_DSA](https://github.com/aryan-upa/Java_DSA)