Q1. If we apply Quick sort on sortted array then what will be the complexity.

Sol1.

If you apply Quick Sort on a sorted array, the worst-case time complexity will be O(n^2), where n is the number of elements in the array. This is because Quick Sort's worst-case time complexity occurs when the pivot is consistently chosen as the smallest or largest element, leading to unbalanced partitions and suboptimal performance. However, the average-case time complexity of Quick Sort is O(n log n) when the elements are randomly arranged or the pivot selection is optimized.

Q2. If we need to insert a node at the end of the doubly linked list. How many addresses need to be change.

Sol2.

To insert a node at the end of a doubly linked list, you typically need to change two addresses: the `next` pointer of the current last node to point to the new node, and the `prev` pointer of the new node to point back to the current last node.

Q3. In a binary tree with level n (n>=0). Then tree contains atmost _____ number of nodes.

Sol3.

In a binary tree with level n (where n ≥ 0), the maximum number of nodes it can contain is 2^(n+1) - 1. This is because each level of a binary tree can have at most 2^n nodes (as it is a complete binary tree), and summing up the nodes from level 0 to level n gives us the formula 2^(n+1) - 1.

Q4. Write a java method to implement PUSH operation in the Stack.

Sol4.

```
import java.util.*;


public class MyStack {
   private ArrayList<Integer> stack;


   public MyStack() {
      stack = new ArrayList<>();
   }
```

```
    public void push(int element) {
        stack.add(element);
    }

    // Other methods like pop, peek, isEmpty, size, etc. can be implemented here
}
```

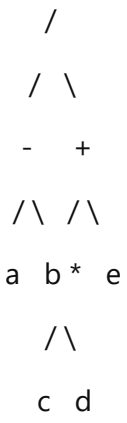
Q5. Construct an expression tree for the following algebraic expression: (a - b) / ((c * d) + e)

Sol5.

To construct an expression tree for the algebraic expression `(a - b) / ((c * d) + e)`, we need to follow the rules of expression trees. We'll break down the expression into its constituent parts and build the tree accordingly.

1. Identify the operators and operands:
   - Operators: `/`, `-`, `*`, `+`
   - Operands: `a`, `b`, `c`, `d`, `e`
2. Determine the precedence and associativity of the operators:
   - Division (`/`) and multiplication (`*`) have higher precedence than subtraction (`-`) and addition (`+`).
   - All operators are left associative.
3. Construct the expression tree recursively based on the precedence and associativity:
   - The operator with the lowest precedence will be the root of the tree.
   - The operands of this operator will be the left and right children of the root.
   - Repeat the process recursively for the operands until all operators and operands are included in the tree.

Here's the expression tree for the given expression:



In this tree:

- The root node represents the division operator `/`.
- The left subtree represents the subtraction operation `a - b`.
- The right subtree represents the addition operation `(c * d) + e`.
- Within the right subtree, the left subtree represents the multiplication operation `c * d`, and the right leaf node represents the operand `e`.

Q6. Write the condition for empty and full circular queue when implementing using array.

Sol6.

When implementing a circular queue using an array, you need to define conditions to check if the queue is empty or full. Here are the conditions:

1. **Empty Circular Queue:**
   - If both the `front` and `rear` pointers are -1, the circular queue is empty.
   - In code:

```
if (front == -1 && rear == -1)
```

2. **Full Circular Queue:**
   - If `(rear + 1) % maxSize` is equal to `front`, the circular queue is full.
   - In code:

```
if ((rear + 1) % maxSize == front)
```

In these conditions:

- `front` points to the front of the circular queue.
- `rear` points to the rear of the circular queue.
- `maxSize` is the maximum size of the circular queue.

Q7. In a complete binary tree if the number of nodes is 1000000. What will be the height of complete binary tree.

Sol7.

In a complete binary tree, the height h is determined by the formula:

$$h=\lfloor \log_2(n) \rfloor$$

where n is the number of nodes.

Given that the number of nodes $n=1000000$, let's calculate the height:

$h=\lfloor \log_2(1000000) \rfloor$

$h=\lfloor 19.9315685693 \rfloor$

$h=19$

So, the height of a complete binary tree with 1,000,000 nodes would be 19.

Q8. Differentiate Strictly Binary Tree and Complete Binary Tree.

Sol8.

A **strictly binary tree** is a type of binary tree in which each node has either 0 or 2 children. In other words, every node in a strictly binary tree must have either no children (a leaf node) or exactly two children. There are no nodes with only one child in a strictly binary tree.

A **complete binary tree**, on the other hand, is a type of binary tree in which all levels are completely filled except possibly for the last level, which is filled from left to right. In other words, in a complete binary tree:

- Every level, except possibly the last, is completely filled with nodes.
- If the last level is not completely filled, the nodes are filled from left to right.

Key differences:

- **Children per node**: In a strictly binary tree, each node can have 0 or 2 children, while in a complete binary tree, each node (except possibly the last level) has exactly 2 children, and the last level is filled from left to right.
- **Structure**: Strictly binary trees have a stricter structure with nodes having either 0 or 2 children, whereas complete binary trees prioritize filling levels from left to right, allowing for varying numbers of nodes in the last level.

Q9. Construct a Binary Search Tree (BST) for the given elements: 15,22,30,10,64,1,3,9,2.

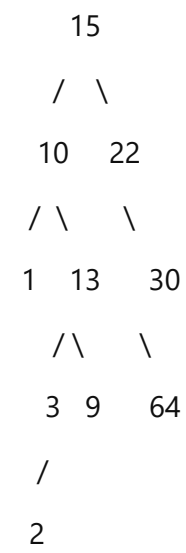
Sol9.

To construct a Binary Search Tree (BST) from the given elements: 15, 22, 30, 10, 64, 1,

3, 9, 2, we need to insert each element into the tree according to the properties of a BST:

1. Start with an empty tree.
2. For each element:
    - If the tree is empty, insert the element as the root.
    - Otherwise, compare the element with the current node:
        - If the element is less than the current node, move to the left subtree.
        - If the element is greater than the current node, move to the right subtree.
        - Repeat the process until a suitable position for the element is found.
    - Insert the element at the suitable position.

Let's construct the BST:



This is the BST constructed from the given elements. Each node follows the property of a BST where the left child is less than the parent and the right child is greater than the parent.

Q10. List the advantages of doubly linked list over single linked list.

Sol10.

Advantages of Doubly Linked List over Single Linked List:

1. **Bidirectional Traversal**: In a doubly linked list, nodes have pointers to both the next and previous nodes. This allows for efficient traversal in both forward and backward directions, whereas in a single linked list, traversal is only possible in one direction (forward).

2. **Insertion and Deletion**: Insertion and deletion operations at the beginning and end of a doubly linked list can be performed in constant time $O(1)$ with pointers manipulation, while in a single linked list, deletion at the end requires traversing the entire list to reach the penultimate node, resulting in $O(n)$ time complexity.
3. **Node Removal without Iteration**: Removing a node from a doubly linked list, given a reference to the node, can be done without traversing the list to find the node's predecessor, as the node has pointers to both its previous and next nodes. This makes removal operations more efficient compared to single linked lists.
4. **Memory Overhead**: Doubly linked lists require additional memory to store the pointers to both the next and previous nodes for each element, leading to a slightly higher memory overhead compared to single linked lists, which only require a single pointer per node.
5. **Implementation of Algorithms**: Certain algorithms, such as reversing a list or implementing advanced data structures like deque (double-ended queue), are more straightforward to implement and more efficient with a doubly linked list due to bidirectional traversal and efficient node removal capabilities.

Overall, the choice between singly linked lists and doubly linked lists depends on the specific requirements of the application, with doubly linked lists offering advantages in scenarios where bidirectional traversal, efficient node removal, and insertion at both ends are important considerations.

Q11. What is the queue's limitation? How the issue is being fixed.

Sol11.

One limitation of a queue data structure is that it has a fixed size or capacity, meaning it can only hold a certain number of elements at a time. When the queue is full, attempting to enqueue (add) additional elements will result in an overflow condition, which can lead to data loss or unexpected behavior.

To address this limitation, there are several approaches to fixing the issue:

1. **Dynamic resizing**: Instead of using a fixed-size array to implement the queue, use a dynamic data structure such as a dynamic array (ArrayList in Java) or a linked list. With dynamic resizing, the size of the underlying data structure is automatically adjusted as needed to accommodate more elements. When the queue reaches its capacity, the underlying data structure is resized to increase its capacity, allowing more elements to be added.

2. **Circular queue**: In a circular queue, elements are stored in a circular manner within a fixed-size array. When the end of the array is reached, the next element is inserted at the beginning of the array if space is available, effectively making use of the available space and avoiding overflow conditions. This allows for efficient use of memory and avoids the need for resizing the underlying data structure.
3. **Priority queue**: In a priority queue, elements are stored based on their priority rather than the order in which they were added. This allows for efficient retrieval of the element with the highest priority, regardless of the order in which elements were added. Priority queues are typically implemented using binary heaps or other heap-based data structures, which have efficient insertion and deletion operations.
4. **Blocking queue**: In a blocking queue, enqueue and dequeue operations are blocked (i.e., the calling thread is paused) if the queue is full or empty, respectively. This allows for synchronization between producer and consumer threads in multi-threaded environments, ensuring that producers do not overwhelm consumers and vice versa.

Each of these approaches addresses the limitation of a fixed-size queue in different ways, allowing for more flexible and efficient queue implementations in various scenarios.

Q12. When would a double linked list be more advantageous than a single linked list?

Sol12.

A doubly linked list can be more advantageous than a single linked list in several scenarios:

1. **Bidirectional Traversal**: Doubly linked lists allow bidirectional traversal, meaning you can traverse the list both forward and backward. This can be advantageous in scenarios where you need to traverse the list in reverse order, which is not efficiently possible with a single linked list.
2. **Efficient Deletion**: Deleting a node from a doubly linked list, given a reference to the node, can be more efficient compared to a single linked list. In a single linked list, to delete a node, you typically need a reference to its predecessor node, which requires traversing the list from the beginning. In a doubly linked list, you can directly delete a node with just a reference to the node itself, as it contains pointers to both its predecessor and successor nodes.
3. **Implementation of Data Structures**: Certain data structures, such as deque (double-ended queue) or linked list-based implementations of stacks and queues, can be more efficiently implemented with doubly linked lists.

Bidirectional traversal and efficient node removal capabilities of doubly linked lists make them suitable for implementing such data structures.

4. **Reverse Iteration**: Doubly linked lists allow for efficient reverse iteration through the list, which can be useful in scenarios where you need to process elements in reverse order without modifying the original list.

Overall, doubly linked lists are advantageous in scenarios where bidirectional traversal, efficient deletion, or reverse iteration are important requirements, making them a suitable choice over single linked lists in such cases.

Q13. Write the Overflow and underflow conditions in a stack using an array.

In a stack implemented using an array, overflow and underflow conditions occur under the following circumstances:

1. **Overflow**: Overflow occurs when trying to push (add) an element onto a full stack. In other words, the stack is already at its maximum capacity, and attempting to add more elements would exceed this capacity.

Overflow condition in pseudocode:

```
if top == maxSize - 1:
    // Stack is full, cannot push more elements
    print "Stack overflow"
```

2. **Underflow**: Underflow occurs when trying to pop (remove) an element from an empty stack. In other words, the stack is empty, and attempting to remove an element from it results in an underflow condition.

Underflow condition in pseudocode:

```
if top == -1:
    // Stack is empty, cannot pop elements
    print "Stack underflow"
```

In both cases, the conditions check whether the stack is at its limit (overflow) or empty (underflow) before performing the corresponding operation (push or pop). These conditions help ensure the integrity and safety of stack operations to prevent data loss or unexpected behavior

Q14. Differentiate between array and linked list.

Sol14.

Arrays and linked lists are two fundamental data structures used for storing and organizing elements. Here are the main differences between them:

1. **Memory Allocation**:
   - **Array**: Arrays are contiguous blocks of memory where elements are stored sequentially. The size of the array is fixed when it is created, and it cannot be dynamically resized without creating a new array and copying elements.
   - **Linked List**: Linked lists consist of nodes where each node contains the data and a reference (pointer) to the next node in the sequence. Nodes are not necessarily stored sequentially in memory, and they are dynamically allocated as needed. Linked lists can grow or shrink in size dynamically without the need for contiguous memory allocation.
2. **Insertion and Deletion**:
   - **Array**: Insertion and deletion operations in an array may require shifting elements to accommodate new elements or maintain the order of elements. Insertion and deletion at the beginning or middle of an array can be inefficient, especially for large arrays, as it may involve moving many elements.
   - **Linked List**: Insertion and deletion operations in a linked list are generally more efficient compared to arrays. Insertion and deletion of nodes can be done by adjusting pointers, which does not require shifting elements. Insertion and deletion at the beginning or middle of a linked list are performed in constant time $O(1)$ if the position is known.
3. **Random Access**:
   - **Array**: Arrays support random access to elements, meaning you can access any element in constant time $O(1)$ using its index. This is because elements are stored sequentially in memory, and their positions are determined by their indices.
   - **Linked List**: Linked lists do not support random access to elements. To access an element in a linked list, you must traverse the list from the beginning or end until you reach the desired position. As a result, the time complexity for accessing an element in a linked list is linear $O(n)$, where n is the number of elements in the list.
4. **Memory Usage**:
   - **Array**: Arrays have a fixed size determined at compile time, which may lead to wasted memory if the array size is larger than the number of elements it needs to store. Additionally, resizing arrays dynamically can be inefficient and may require copying elements to a new array.
   - **Linked List**: Linked lists use memory more efficiently compared to arrays because they allocate memory for each node dynamically as

needed. This allows linked lists to grow or shrink in size without wasting memory.

5. **Traversal**:
    - **Array**: Arrays support efficient sequential traversal, as elements are stored sequentially in memory. However, traversal in reverse order may be less efficient.
    - **Linked List**: Linked lists support efficient traversal in both forward and reverse directions, as each node contains a reference to the next and/or previous node.

Q15. Differentiate between stack and queue.

Sol15.

Stack and queue are two fundamental data structures with distinct characteristics and usage. Here's how they differ:

1. **Ordering Principle**:
    - **Stack**: Follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed. Elements are added and removed from only one end, typically referred to as the "top" of the stack.
    - **Queue**: Follows the First-In-First-Out (FIFO) principle, where the first element added to the queue is the first one to be removed. Elements are added to the rear (enqueue) and removed from the front (dequeue) of the queue.
2. **Operations**:
    - **Stack**: Supports two primary operations:
        - **Push**: Adds an element to the top of the stack.
        - **Pop**: Removes and returns the top element of the stack.
    - **Queue**: Supports two primary operations:
        - **Enqueue**: Adds an element to the rear of the queue.
        - **Dequeue**: Removes and returns the front element of the queue.
3. **Data Flow**:
    - **Stack**: Operates in a Last-In-First-Out (LIFO) manner, meaning the last element pushed onto the stack is the first one to be popped off.
    - **Queue**: Operates in a First-In-First-Out (FIFO) manner, meaning the first element enqueued into the queue is the first one to be dequeued.
4. **Usage**:
    - **Stack**: Commonly used in scenarios where elements need to be processed in a reverse order or where backtracking is required, such as

expression evaluation, function call management (call stack), and undo operations.
- **Queue**: Commonly used in scenarios involving data processing in the order of arrival or scheduling, such as task scheduling, print queue management, and breadth-first search algorithms.

5. **Implementation**:
   - **Stack**: Can be implemented using arrays or linked lists. Arrays are commonly used due to their simplicity and efficiency in implementing the LIFO behavior.
   - **Queue**: Can also be implemented using arrays or linked lists. Linked lists are commonly used for implementing queues due to their efficient insertion and deletion operations at both ends.

Q16. What is difference between static and dynamic memory allocation.

Sol16.

Static and dynamic memory allocation are two methods used to allocate memory in computer programs, and they differ in terms of when and how memory is allocated and deallocated:

1. **Static Memory Allocation**:
   - In static memory allocation, memory is allocated at compile time, and the size of memory allocated is fixed throughout the program's execution.
   - Memory allocation and deallocation are determined before the program runs, typically based on the program's structure and global variables.
   - Static memory allocation is commonly used for variables declared globally, as well as for variables declared with the `static` keyword inside functions.
   - The memory allocated statically is typically stored in the stack segment of the program's memory space.
   - Static memory allocation is efficient in terms of memory management and access speed, but it lacks flexibility, as the memory size is fixed.
2. **Dynamic Memory Allocation**:
   - In dynamic memory allocation, memory is allocated at runtime, allowing for flexibility in memory usage during program execution.
   - Memory allocation and deallocation are performed explicitly by the programmer using functions such as `malloc`, `calloc`, `realloc`, and `free` in languages like C and C++.

- Dynamic memory allocation is commonly used when the size of data structures or the amount of memory needed cannot be determined at compile time.
- The memory allocated dynamically is typically stored in the heap segment of the program's memory space.
- Dynamic memory allocation provides flexibility in memory management, allowing for allocation and deallocation of memory as needed during program execution. However, it requires careful memory management to avoid memory leaks and fragmentation.

Q17. Calculating minimum and maximum height from 221 number of nodes in a binary tree.

Sol17.

To calculate the minimum and maximum height of a binary tree with 221 nodes, we can use the following formulas:

1. **Minimum Height**:
   - For a binary tree with n nodes, the minimum height (hmin) is calculated as:
   - $h_{min} = \lceil \log_2(n+1) \rceil - 1$
   - Substituting n = 221 into the formula:
   - $h\text{min} = \lceil \log_2(221+1) \rceil - 1$
   - $h\text{min} = \lceil \log_2(222) \rceil - 1$
   - $h_{min} = \lceil 7.807 \rceil - 1$
   - $h\text{min} = 8 - 1$
   - $h_{min} = 7$
2. **Maximum Height**:
   - For a binary tree with n nodes, the maximum height (hmax) is calculated as:
   - $h_{max} = n - 1$
   - Substituting n = 221 into the formula:
   - $h\text{max} = 221 - 1$
   - $h_{max} = 220$

Therefore, for a binary tree with 221 nodes:

- The minimum height is 7.
- The maximum height is 220.

Q18. Write an algorithm or java method to calculate the length of a singly linked list.

Sol18.

```
public class LinkedList {
    private static class Node {
        int data;
        Node next;

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

    private Node head;

    // Method to add a node at the end of the linked list
    public void append(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
```

```
    // Method to calculate the length of the linked list
    public int length() {
        int count = 0;
        Node current = head;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }

    // Main method for testing
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.append(1);
        list.append(2);
        list.append(3);
        list.append(4);
        list.append(5);
        System.out.println("Length of the linked list: " + list.length());
    }
}
```

Q19. Explain the addition of two polynomials using suitable example.

Sol19.

Adding two polynomials involves combining like terms to simplify the expression. Each polynomial consists of terms with coefficients and exponents, where the coefficients represent the numeric values and the exponents represent the powers of variables.

Let's consider the addition of two polynomials:

1. Polynomial A: $3x^3+2x^2-x+5$
2. Polynomial B: $2x^2+4x-3$

To add these polynomials, we follow these steps:

1. **Combine like terms with the same exponent**:
   - First, we add the coefficients of terms with the same exponent.
   - For Polynomial A, the terms with the same exponent are:
     - $3x^3$
     - $2x^2$
     - $-x$
     - The constant term 5
   - For Polynomial B, the terms with the same exponent are:
     - $2x^2$
     - $4x$
     - The constant term -3
2. **Add the coefficients**:
   - For terms with the same exponent, we add their coefficients:
     - $3x^3+0x^3=3x^3$ (since Polynomial B does not have a term with x^3)
     - $2x^2+2x^2=4x^2$
     - $-x+4x=3x$
     - $5-3=2$
3. **Combine like terms**:
   - Combine the resulting terms to get the simplified polynomial:
     - $3x^3+4x^2+3x+2$

So, the addition of Polynomial A and Polynomial B is $3x^3+4x^2+3x+2$.

In general, when adding polynomials, we combine like terms by adding their coefficients while keeping the exponents unchanged. This process simplifies the expression and yields the sum of the two polynomials.

Q20. Write the Overflow and underflow conditions in a circular queue using an array.

Sol20

In a circular queue implemented using an array, overflow and underflow conditions occur as follows:

1. Overflow: This happens when the rear pointer (index) of the queue reaches the maximum capacity of the array and there is an attempt to enqueue (add)

more elements. It means the queue is full, and no more elements can be added until some elements are dequeued (removed).
2. Underflow: This occurs when the front pointer (index) of the queue reaches the same position as the rear pointer (index), indicating that the queue is empty, and there is an attempt to dequeue (remove) an element. It means the queue is already empty, and no elements can be dequeued until new elements are enqueued.

Circular queues implemented using arrays overcome the limitations of traditional linear queues by wrapping around to the beginning of the array when reaching the end, thus allowing for efficient use of memory and avoiding unnecessary shifting of elements.

Q21. The height of a tree is the length of the longest root-to-leaf path in it. Calculate the maximum and the minimum number of nodes in a binary tree of height 5.

Sol21.

To calculate the maximum and minimum number of nodes in a binary tree of height 5, we can use the following formulas:

1. Maximum number of nodes in a binary tree of height h:

   MaximumNodes = 2^(h+1) - 1

2. Minimum number of nodes in a binary tree of height h:

   MinimumNodes = h + 1

Using these formulas:

1. For a binary tree of height 5:

   - Maximum number of nodes = 2^(5+1) - 1 = 63

   - Minimum number of nodes = 5 + 1 = 6

So, the maximum number of nodes in a binary tree of height 5 is 63, and the minimum number of nodes is 6.

Q22. How to delete a node in binary search tree? Explain with the help of example.

Sol22.

To delete a node in a binary search tree (BST), we need to consider three cases:

1. If the node to be deleted has no children (leaf node): In this case, we simply remove the node from the tree.

2. If the node to be deleted has only one child: We remove the node and replace it with its child.

3. If the node to be deleted has two children: We find the node with the smallest value in the right subtree (or the node with the largest value in the left subtree) and replace the node to be deleted with this node. Then, we recursively delete this replacement node from its original position.

Let's illustrate this process with an example:

Consider the following binary search tree:

```
        9
      /   \
     5     12
    / \   / \
   3   7 10  15
```

Let's say we want to delete node 5 from this tree.

1. Node 5 has two children (3 and 7). We find the smallest value in its right subtree, which is 7. So, we replace node 5 with node 7.

```
        9
      /   \
     7     12
    /     / \
   3     10  15
```

2. Now, node 5 has only one child (3). We can simply remove node 5 and replace it with its child (3).

```
        9
      /   \
     7     12
      \    / \
       3  10  15
```

Now, node 5 has been successfully deleted from the binary search tree.

Keep in mind that after deleting a node, we need to ensure that the BST property is maintained, which means that for every node, its left child's value is less than its value, and its right child's value is greater than its value. This may involve rearranging the tree structure after deletion.

Q23. How the choice of pivot element effects the effciency of Quick sort algorithm with suitable example.

Sol23.

The choice of pivot element significantly affects the efficiency of the Quick Sort algorithm. When selecting a pivot, ideally, we want it to divide the input array into two roughly equal

partitions. However, if the pivot is poorly chosen, it can lead to uneven partitioning, resulting in inefficient performance.

Let's consider an example to illustrate this:

Suppose we have an input array `[5, 9, 3, 7, 2, 8, 6]` and we choose the first element (`5`) as the pivot.

1. First partition step:
   - Pivot: 5
   - After partitioning: `[3, 2, 5, 9, 7, 8, 6]`

   In this partition, all elements smaller than the pivot are on the left, and all elements larger are on the right. However, the left partition contains only two elements, while the right partition contains five elements.

2. Second partition step (recursively sorting the left and right partitions):
   - Left partition `[3, 2]` (choosing the first element `3` as the pivot):
     - Pivot: 3
     - After partitioning: `[2, 3]`

   - Right partition `[9, 7, 8, 6]` (choosing the first element `9` as the pivot):
     - Pivot: 9
     - After partitioning: `[7, 8, 6, 9]`

   Both left and right partitions are already sorted because they contain only one or two elements each.

After these steps, the array is partially sorted but not completely sorted. The left partition `[2, 3]` and the right partition `[7, 8, 6, 9]` are not combined correctly, leading to inefficient sorting.

If we had chosen a better pivot, such as the median-of-three (median of the first, middle, and last elements), or the randomized pivot selection, it would have resulted in more balanced partitions and improved efficiency.

Therefore, the choice of pivot greatly influences the efficiency of the Quick Sort algorithm, and selecting a good pivot strategy is crucial for achieving optimal performance.

Q24. Implement PUSH and POP operations in stack using array.

Sol24.

```
public class Stack {
    private int maxSize;
    private int[] stackArray;
    private int top;

    // Constructor
    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }

    // Method to check if the stack is empty
    public boolean isEmpty() {
        return (top == -1);
    }

    // Method to check if the stack is full
    public boolean isFull() {
        return (top == maxSize - 1);
    }
```

```
// Method to push an element onto the stack
public void push(int element) {
    if (isFull()) {
        System.out.println("Stack Overflow: Cannot push element, stack is full.");
        return;
    }
    stackArray[++top] = element;
}

// Method to pop an element from the stack
public int pop() {
    if (isEmpty()) {
        System.out.println("Stack Underflow: Cannot pop element, stack is empty.");
        return -1; // Return a default value indicating underflow
    }
    return stackArray[top--];
}

// Method to peek at the top element of the stack without removing it
public int peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
        return -1; // Return a default value indicating empty stack
    }
    return stackArray[top];
}

public static void main(String[] args) {
    Stack stack = new Stack(5); // Creating a stack with a maximum size of 5
```

```
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);
        stack.push(5);

        System.out.println("Stack peek: " + stack.peek()); // Output: 5

        stack.push(6); // Output: Stack Overflow: Cannot push element, stack is full.

        System.out.println("Popped element: " + stack.pop()); // Output: 5

        System.out.println("Popped element: " + stack.pop()); // Output: 4

        System.out.println("Stack peek: " + stack.peek()); // Output: 3
    }
}
```

Q25. Write an algorithm or java method to concatinate two singly linked list.

Sol25.

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```
class LinkedList {
    Node head;

    public void append(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }

    public void concatenate(LinkedList list2) {
        if (head == null) {
            head = list2.head;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = list2.head;
    }
```

```
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList list1 = new LinkedList();
        list1.append(1);
        list1.append(2);
        list1.append(3);

        LinkedList list2 = new LinkedList();
        list2.append(4);
        list2.append(5);
        list2.append(6);

        System.out.println("List 1:");
        list1.display();

        System.out.println("List 2:");
        list2.display();

        list1.concatenate(list2);
```

```
        System.out.println("Concatenated List:");
        list1.display();
    }
}
```

Q26. Write an java method or algorithm to search a key in Binary Search Tree (BST).

Sol26.

```
class Node {
    int data;
    Node left, right;

    public Node(int value) {
        data = value;
        left = right = null;
    }
}

class BST {
    Node root;

    public BST() {
        root = null;
    }

    public void insert(int key) {
        root = insertRec(root, key);
    }

    private Node insertRec(Node root, int key) {
```

```
    if (root == null) {
        root = new Node(key);
        return root;
    }

    if (key < root.data)
        root.left = insertRec(root.left, key);
    else if (key > root.data)
        root.right = insertRec(root.right, key);

    return root;
}

public boolean search(int key) {
    return searchRec(root, key);
}

private boolean searchRec(Node root, int key) {
    if (root == null)
        return false;

    if (root.data == key)
        return true;

    if (key < root.data)
        return searchRec(root.left, key);
    else
        return searchRec(root.right, key);
}
```

```
}

public class Main {
    public static void main(String[] args) {
        BST bst = new BST();
        bst.insert(50);
        bst.insert(30);
        bst.insert(70);
        bst.insert(20);
        bst.insert(40);
        bst.insert(60);
        bst.insert(80);

        int keyToSearch = 40;
        if (bst.search(keyToSearch))
            System.out.println(keyToSearch + " found in BST.");
        else
            System.out.println(keyToSearch + " not found in BST.");
    }
}
```

Q27. Initially, there are elements {A, R, E} in a stack and E is the Top element of the stack. Do the following operations with suitable diagram, POP(), PUSH{W}, POP(), POP(), PUSH{G}, POP().

Sol27.

Sure, let's go through the operations step by step:

1. Initially, the stack contains elements {A, R, E}, with E as the top element:

```
| E |
```

| R |

| A |

```

2. POP(): Remove the top element from the stack.

After performing POP(), the stack becomes:

```

| R |

| A |

```

3. PUSH{W}: Push the element W onto the stack.

After performing PUSH{W}, the stack becomes:

```

| W |

| R |

| A |

```

4. POP(): Remove the top element from the stack.

After performing POP(), the stack becomes:
```

```
| R |
| A |
-----
```

5. POP(): Remove the top element from the stack.

After performing POP(), the stack becomes:

```
| A |
-----
```

6. PUSH{G}: Push the element G onto the stack.

After performing PUSH{G}, the stack becomes:

```
| G |
| A |
-----
```

7. POP(): Remove the top element from the stack.

After performing POP(), the stack becomes:

```
| A |
-----
```

So, the final stack contains only the element A.

Q28. Construct a Max-Heap for the given set of elements: 1, 22, 52, 57, 47, 61, 27, 12, 15, 9, 11. Show all the steps with a suitable diagram.

Sol28.

To construct a Max-Heap from the given set of elements, we'll follow these steps:

1. Insert the elements one by one into an initially empty heap.

2. After each insertion, ensure that the Max-Heap property is maintained by swapping elements if necessary to move the newly inserted element to its correct position.

Here are the steps illustrated with a suitable diagram:

1. Insert 1:
```
   1
```

2. Insert 22:
```
    22
   /
  1
```

3. Insert 52:

```
   52
  / \
 1   22
```

4. Insert 57:

```
   57
  / \
 1   52
/
22
```

5. Insert 47:

```
   57
  / \
 47  52
/ \
22 1
```

6. Insert 61:

```
   61
  /  \
```

```
 47    57
/ \   /
22  1 52
```

7. Insert 27:

```
    61
  /   \
 47    57
/ \   / \
22  1 52 27
```

8. Insert 12:

```
    61
  /   \
 47    57
/ \   / \
22  1 52 27
/
12
```

9. Insert 15:

```
    61
  /   \
```

```
  47    57
 / \   / \
22  1 52 27
 / \
12 15
```

10. Insert 9:

```
    61
   /   \
  47    57
 / \   / \
22  1 52 27
 / \ / \
12 15 9
```

11. Insert 11:

```
    61
   /   \
  47    57
 / \   / \
22  1 52 27
 / \ / \ /
12 15 9 11
```

Now, the Max-Heap is constructed from the given set of elements. Each parent node is greater than or equal to its children, ensuring the Max-Heap property is satisfied.

Q29. Initially, there are elements {B, P, Q, D} in a queue D is the front element and B is the rear of the queue. Do the following operations with a suitable diagram, DELETE(), DELETE(), DELETE(), DELETE(), INSERT(M), DELETE(), DELETE().

Sol29.

Sure, let's go through the operations step by step:

1. Initially, the queue contains elements {B, P, Q, D}, with D as the front element and B as the rear element:

```
Front -> | D | P | Q | B | <- Rear
```

2. DELETE(): Remove the front element from the queue.

After performing DELETE(), the queue becomes:

```
Front -> | P | Q | B | <- Rear
```

3. DELETE(): Remove the front element from the queue.

After performing DELETE(), the queue becomes:

```
Front -> | Q | B | <- Rear
```

```

4. DELETE(): Remove the front element from the queue.

After performing DELETE(), the queue becomes:

```
Front -> | B | <- Rear
```

5. DELETE(): Remove the front element from the queue.

After performing DELETE(), the queue becomes empty:

```
Empty Queue
```

6. INSERT(M): Insert the element M into the rear of the queue.

After performing INSERT(M), the queue becomes:

```
Front -> | M | <- Rear
```

7. DELETE(): Remove the front element from the queue.

After performing DELETE(), the queue becomes empty:
```

```

Empty Queue

```

8. DELETE(): Attempting to delete from an empty queue, no change in the queue:

```

Empty Queue

```

So, after performing the given operations, the final state of the queue is an empty queue.

Q30. Construct a BST for the given set of elements: 21, 32, 12, 65, 45, 67, 34, 2, 5, 9, 1, 10. Show all the steps with a suitable diagram.

Sol30.

Sure, let's construct a Binary Search Tree (BST) for the given set of elements: 21, 32, 12, 65, 45, 67, 34, 2, 5, 9, 1, 10.

Here are the steps:

1. Start with the root node: 21.

2. Insert 32, it's greater than 21 so it goes to the right of 21.

3. Insert 12, it's less than 21 so it goes to the left of 21.

4. Insert 65, it's greater than 32 so it goes to the right of 32.

5. Insert 45, it's less than 65 but greater than 32, so it goes to the left of 65.

6. Insert 67, it's greater than 65, so it goes to the right of 65.

7. Insert 34, it's greater than 32 but less than 65, so it goes to the left of 65 but to the right of 32.

8. Insert 2, it's less than 21 so it goes to the left of 21.

9. Insert 5, it's greater than 2 but less than 12, so it goes to the right of 12 but to the left of 21.

10. Insert 9, it's greater than 5 but less than 12, so it goes to the right of 5 but to the left of 12.

11. Insert 1, it's less than 2, so it goes to the left of 2.

12. Insert 10, it's greater than 9 but less than 12, so it goes to the right of 9 but to the left of 12.

Here's the final Binary Search Tree (BST) diagram:

```
      21
     /  \
    12   32
   /  \   \
  2   13   65
 / \  \  / \
1   5  10 45 67
     \
      9
```

This diagram represents the Binary Search Tree (BST) for the given set of elements.

Q31. Write a java program to delete a node from Kth position in singly linked list.

Sol31.

```
class Node {
    int data;
    Node next;

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

class SinglyLinkedList {
    Node head;

    public void deleteNodeAtK(int k) {
        if (head == null || k <= 0) {
            return;
        }

        if (k == 1) {
            head = head.next;
            return;
        }

        Node current = head;
        Node prev = null;
        int count = 1;
```

```
        while (current != null && count != k) {
            prev = current;
            current = current.next;
            count++;
        }

        if (current == null) {
            return;
        }

        prev.next = current.next;
    }

    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);
```

```
        list.head.next.next.next.next = new Node(5);

        System.out.println("Original list:");
        list.display();

        int k = 3; // Position of the node to delete
        list.deleteNodeAtK(k);

        System.out.println("List after deleting node at position " + k + ":");
        list.display();
    }
}
```

Q32. Convert the following infix expression into prefix expression using stack. A*(B+D)/E-F*(G+H/K)

Sol32.

To convert an infix expression to a prefix expression using a stack, we can follow these steps:

1. Reverse the infix expression.

2. Scan the reversed expression from left to right.

3. If an operand is encountered, add it to the result string.

4. If an operator is encountered:

   - If it's a closing parenthesis, push it onto the stack.

   - If it's an opening parenthesis, pop operators from the stack and add them to the result string until a closing parenthesis is encountered.

   - If it's any other operator, compare its precedence with the precedence of the top operator on the stack. If the precedence of the incoming operator is lower, push it onto the stack. If it's higher or equal, pop operators from the stack and add them to the result string until a lower precedence operator is encountered or the stack is empty, then push the incoming operator onto the stack.

5. Reverse the result string to get the final prefix expression.

Let's apply these steps to the infix expression A*(B+D)/E-F*(G+H/K):

1. Reverse the infix expression: K/H+G)*F-E/(D+B)*A

2. Start scanning the reversed expression from left to right.

Here's the step-by-step conversion:

1. Start with the reversed infix expression: K/H+G)*F-E/(D+B)*A

2. Initialize an empty stack and an empty result string.

3. Scan the reversed expression:

   - K: Operand, add to the result string.

   - /: Operator, push onto the stack.

   - H: Operand, add to the result string.

   - +: Operator, push onto the stack.

   - G: Operand, add to the result string.

   -): Closing parenthesis, push onto the stack.

   - *: Operator, push onto the stack.

   - F: Operand, add to the result string.

   - -: Operator, push onto the stack.

   - E: Operand, add to the result string.

   - /: Operator, push onto the stack.

   - (: Opening parenthesis, pop operators from the stack and add them to the result string until a closing parenthesis is encountered.

   - D: Operand, add to the result string.

   - +: Operator, push onto the stack.

   - B: Operand, add to the result string.

   -): Closing parenthesis, pop operators from the stack and add them to the result string until an opening parenthesis is encountered.

   - *: Operator, push onto the stack.

   - A: Operand, add to the result string.

4. Pop any remaining operators from the stack and add them to the result string.

5. Reverse the result string to get the final prefix expression.

After reversing the result string, we get the prefix expression: *A/-*EF+K/*HDGB

So, the prefix expression for the infix expression A*(B+D)/E-F*(G+H/K) is *A/-*EF+K/*HDGB.

Q33. What is circular Queue? Write a pseudocode or method in java to insert an element in circular queue?

Sol33.

A circular queue is a data structure that follows the FIFO (First-In-First-Out) principle, similar to a regular queue, but with a fixed size. In a circular queue, the last element is connected to the first element, forming a circle. This allows efficient use of space because when the queue becomes full, inserting new elements causes the oldest elements to be overwritten.

Here's a pseudocode to insert an element in a circular queue:

```
function enqueue(element):
    if (queue is full):
        return "Queue is full"
    else if (queue is empty):
        front = rear = 0
        queue[rear] = element
    else:
        rear = (rear + 1) % queue_size
        queue[rear] = element
    return "Element inserted successfully"
```

In this pseudocode:

- `queue` is the array representing the circular queue.
- `queue_size` is the maximum size of the circular queue.
- `front` is the index of the front element in the circular queue.
- `rear` is the index of the rear element in the circular queue.
- `element` is the element to be inserted.

And here's a Java method to insert an element in a circular queue:

```java
class CircularQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    public CircularQueue(int capacity) {
        this.capacity = capacity;
        queue = new int[capacity];
        front = rear = -1;
        size = 0;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == capacity;
```

```
    }

    public void enqueue(int element) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        } else if (isEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % capacity;
        }
        queue[rear] = element;
        size++;
    }
}
```

Q34. The order of nodes of a binary tree in inorder and postorder traversal are as follows: In order : B, I, D, A, C, G, E, H, F. Post order: I, D, B, G, C, H, F, E, A. (i) Draw the corresponding binary tree. (ii) Write the pre order traversal of the same tree.

Sol34.

To draw the corresponding binary tree and write the pre-order traversal, we can follow these steps:

1. Use the given inorder and postorder traversal sequences to construct the binary tree.

2. Once the binary tree is constructed, perform pre-order traversal to get the desired sequence.

Given inorder traversal: B, I, D, A, C, G, E, H, F.

Given postorder traversal: I, D, B, G, C, H, F, E, A.

(i) Drawing the corresponding binary tree:

```
      A
     / \
    B   E
   / \   \
  I   D   F
       \
        C
         \
          G
           \
            H
```

(ii) Pre-order traversal of the binary tree: A, B, I, D, E, F, C, G, H.

So, the pre-order traversal of the binary tree is: A, B, I, D, E, F, C, G, H.

Q35. Discuss doubly linked list. Write an algorithm to insert a node after a given node in singly linked list.

Sol35.

A doubly linked list is a type of linked list where each node contains two pointers: one that points to the next node in the sequence (next pointer) and another that points to the previous node in the sequence (previous pointer). This allows traversal in both forward and backward directions.

Key features of a doubly linked list:

- Each node contains data and two pointers: next and previous.

- The last node's next pointer points to null, and the first node's previous pointer points to null, indicating the end of the list in both directions.

- Insertion and deletion operations are more efficient compared to a singly linked list as we can directly access the previous node using the previous pointer.

Algorithm to insert a node after a given node in a singly linked list:

```plaintext
Algorithm InsertAfter(nodeToInsertAfter, newData):
    // Create a new node with the given data
    newNode = Node(newData)

    // Check if the given node is null (if the list is empty)
    if nodeToInsertAfter == null:
        // The list is empty, so set the new node as the head
        head = newNode
    else:
        // Update the next pointer of the new node to point to the next node of the given node
        newNode.next = nodeToInsertAfter.next

        // Update the next pointer of the given node to point to the new node
        nodeToInsertAfter.next = newNode

    // The new node has been inserted after the given node
```

In this algorithm:
- `nodeToInsertAfter` is the node after which we want to insert the new node.
- `newData` is the data of the new node to be inserted.
- `newNode` is the new node to be inserted.

This algorithm handles two cases:

1. If the given node is null (indicating an empty list), the new node becomes the head of the list.

2. If the given node is not null, the new node is inserted after the given node by updating the next pointers accordingly.

Q36. Write an algorithm for Merge Sort. Explains with the help of suitable example.

Sol36.

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves. It works as follows:

1. **Divide**: Split the input array into two halves.

2. **Conquer**: Recursively sort each half.

3. **Merge**: Merge the two sorted halves to produce the final sorted array.

Here's the algorithm for Merge Sort:

```
Algorithm MergeSort(arr):
    if length of arr <= 1:
        return arr

    // Divide
    mid = length of arr / 2
    left_half = MergeSort(arr[0:mid])
    right_half = MergeSort(arr[mid:end])

    // Merge
    sorted_arr = Merge(left_half, right_half)
    return sorted_arr
```

```
Algorithm Merge(left_arr, right_arr):
    merged_arr = []
    left_index = 0
    right_index = 0

    // Compare elements from left and right arrays and merge them into merged_arr
    while left_index < length of left_arr and right_index < length of right_arr:
        if left_arr[left_index] <= right_arr[right_index]:
            merged_arr.append(left_arr[left_index])
            left_index++
        else:
            merged_arr.append(right_arr[right_index])
            right_index++

    // Add remaining elements from left_arr (if any)
    while left_index < length of left_arr:
        merged_arr.append(left_arr[left_index])
        left_index++

    // Add remaining elements from right_arr (if any)
    while right_index < length of right_arr:
        merged_arr.append(right_arr[right_index])
        right_index++

    return merged_arr
```

Explanation with an example:

Consider an unsorted array: [38, 27, 43, 3, 9, 82, 10]

1. Divide: Divide the array into two halves: [38, 27, 43] and [3, 9, 82, 10].

2. Conquer: Recursively sort each half.

   - Sort [38, 27, 43]: [27, 38, 43]

   - Sort [3, 9, 82, 10]: [3, 9, 10, 82]

3. Merge: Merge the sorted halves.

   - Merge [27, 38, 43] and [3, 9, 10, 82]: [3, 9, 10, 27, 38, 43, 82]

The final sorted array is [3, 9, 10, 27, 38, 43, 82].

Q37. Write an algorithm for Quick sort. Use Quick sort algorithm to sort the following elements: 2, 8, 7, 1, 3, 5, 6, 4.

Sol37.

Quick Sort is a divide-and-conquer 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 recursively sorted.

Here's the algorithm for Quick Sort:

```
Algorithm QuickSort(arr, low, high):
    if low < high:
        // Partition the array
        pivot_index = Partition(arr, low, high)

        // Recursively sort the sub-arrays
        QuickSort(arr, low, pivot_index - 1)
        QuickSort(arr, pivot_index + 1, high)

Algorithm Partition(arr, low, high):
    // Select the pivot element (choose the last element in this case)
    pivot = arr[high]
```

```
    i = low - 1

    // Partition the array
    for j from low to high - 1:
        if arr[j] <= pivot:
            i++
            swap arr[i] and arr[j]

    // Swap the pivot element with the element at index i+1
    swap arr[i + 1] and arr[high]

    // Return the index of the pivot element after partitioning
    return i + 1
```

Now, let's use Quick Sort algorithm to sort the given elements: 2, 8, 7, 1, 3, 5, 6, 4.

Step-by-step:
1. Choose a pivot (let's choose the last element, 4).
2. Partition the array around the pivot:
   - [2, 1, 3, 4, 8, 7, 6, 5]
3. Recursively sort the sub-arrays:
   - Left sub-array: [2, 1, 3]
   - Right sub-array: [8, 7, 6, 5]
4. Partition the left sub-array:
   - [1, 2, 3, 4, 8, 7, 6, 5]
5. Recursively sort the sub-arrays:
   - Left sub-array: [1, 2, 3]
   - Right sub-array: [8, 7, 6, 5]

6. Partition the right sub-array:

- [1, 2, 3, 4, 5, 7, 6, 8]

7. Recursively sort the sub-arrays:

- Left sub-array: [1, 2, 3, 4, 5]

- Right sub-array: [7, 6, 8]

8. Partition the right sub-array:

- [1, 2, 3, 4, 5, 6, 7, 8]

9. The array is now sorted: [1, 2, 3, 4, 5, 6, 7, 8].

So, the sorted elements using Quick Sort algorithm are: 1, 2, 3, 4, 5, 6, 7, 8.

Q38. Write an algorithm to evaluate postfix expression also find the value of 7,5,2,-,*,4,1,5,-,/,+.

Sol38.

To evaluate a postfix expression, we can use a stack to keep track of operands and perform operations when operators are encountered. Here's the algorithm to evaluate a postfix expression:

Algorithm EvaluatePostfix(expression):

1. Create an empty stack.

2. For each character in the expression:

- If the character is an operand, push it onto the stack.

- If the character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.

3. At the end, the stack will contain the final result.

Here's the algorithm to evaluate the given postfix expression:

Algorithm EvaluatePostfix("7,5,2,-,*,4,1,5,-,/,+"):

1. Create an empty stack.

2. Split the expression by comma to get individual elements: ["7", "5", "2", "-", "*", "4", "1", "5", "-", "/", "+"]

3. For each element in the expression:

   - If the element is an operand, push it onto the stack.

   - If the element is an operator:

     - Pop two operands from the stack.

     - Perform the operation.

     - Push the result back onto the stack.

4. At the end, the stack will contain the final result.

Let's apply the algorithm to evaluate the given postfix expression:

```plaintext
Expression: 7,5,2,-,*,4,1,5,-,/,+

Stack: []
```

1. Element: "7"

   - Push 7 onto the stack: [7]

2. Element: "5"

   - Push 5 onto the stack: [7, 5]

3. Element: "2"

   - Push 2 onto the stack: [7, 5, 2]

4. Element: "-"

   - Pop two operands from the stack (2 and 5).

   - Perform the operation (5 - 2 = 3).

   - Push the result (3) onto the stack: [7, 3]

5. Element: "*"
   - Pop two operands from the stack (3 and 7).
   - Perform the operation (3 * 7 = 21).
   - Push the result (21) onto the stack: [21]

6. Element: "4"
   - Push 4 onto the stack: [21, 4]

7. Element: "1"
   - Push 1 onto the stack: [21, 4, 1]

8. Element: "5"
   - Push 5 onto the stack: [21, 4, 1, 5]

9. Element: "-"
   - Pop two operands from the stack (5 and 1).
   - Perform the operation (1 - 5 = -4).
   - Push the result (-4) onto the stack: [21, 4, -4]

10. Element: "/"
    - Pop two operands from the stack (-4 and 4).
    - Perform the operation (4 / -4 = -1).
    - Push the result (-1) onto the stack: [21, -1]

11. Element: "+"
    - Pop two operands from the stack (-1 and 21).
    - Perform the operation (21 + (-1) = 20).
    - Push the result (20) onto the stack: [20]

The final result is 20.

```

So, the value of the given postfix expression "7,5,2,-,*,4,1,5,-,/,+" is 20.

Q39. What is Stack? Write a algorithm or program for linked list implementation of stack.

Sol39.

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning that the element inserted last is the first one to be removed. It operates on two main operations: push (to add an element to the top of the stack) and pop (to remove the top element from the stack). Additionally, it may also support other operations such as peek (to view the top element without removing it) and isEmpty (to check if the stack is empty).

Here's an algorithm for the linked list implementation of a stack:

Algorithm StackLinkedList:

- Define a Node class with two attributes: data and next (pointer to the next node).
- Define a StackLinkedList class with one attribute: top (pointer to the top node of the stack).
- Initialize an empty stack (top = null).
- Implement the following operations:
  - Push(data): Create a new node with the given data. Set the next pointer of the new node to point to the current top node. Update the top pointer to point to the new node.
  - Pop(): If the stack is empty (top = null), return an error. Otherwise, store the data of the top node, update the top pointer to point to the next node, and return the stored data.
  - Peek(): If the stack is empty, return an error. Otherwise, return the data of the top node.
  - isEmpty(): Return true if the stack is empty (top = null), otherwise return false.

Here's a Java program for the linked list implementation of a stack:

```java
class Node {

```
    int data;
    Node next;

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

class StackLinkedList {
    private Node top;

    public StackLinkedList() {
        this.top = null;
    }

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        int data = top.data;
        top = top.next;
        return data;
```

```
    }

    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return top.data;
    }

    public boolean isEmpty() {
        return top == null;
    }
}

public class Main {
    public static void main(String[] args) {
        StackLinkedList stack = new StackLinkedList();
        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Top element: " + stack.peek());
        System.out.println("Pop: " + stack.pop());
        System.out.println("Pop: " + stack.pop());
        System.out.println("Pop: " + stack.pop());
        System.out.println("Is stack empty? " + stack.isEmpty());
    }
}
```

This Java program demonstrates the linked list implementation of a stack. It creates a stack, pushes elements onto it, pops elements from it, and checks if it's empty.

Q40. The keys 12, 17, 13, 2, 5, 43, 5 and 15 are inserted into an initially empty hash table of length 15 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

Sol40.

To solve this problem, let's go step by step:

1. Initialize an empty hash table of length 15.

2. Define the hash function h(k) = k mod 10.

3. Use linear probing to resolve collisions.

Here's how it's done:

1. Insert 12: h(12) = 12 mod 10 = 2. Insert 12 at index 2.

2. Insert 17: h(17) = 17 mod 10 = 7. Insert 17 at index 7.

3. Insert 13: h(13) = 13 mod 10 = 3. Insert 13 at index 3.

4. Insert 2: h(2) = 2 mod 10 = 2. Collision at index 2, linear probing to next available slot. Insert 2 at index 3.

5. Insert 5: h(5) = 5 mod 10 = 5. Insert 5 at index 5.

6. Insert 43: h(43) = 43 mod 10 = 3. Collision at index 3, linear probing to next available slot. Insert 43 at index 4.

7. Insert 5: h(5) = 5 mod 10 = 5. Collision at index 5, linear probing to next available slot. Insert 5 at index 6.

8. Insert 15: h(15) = 15 mod 10 = 5. Collision at index 5, linear probing to next available slot. Insert 15 at index 7.

The resultant hash table is:

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Value: - - 12 13 43 5 15 17 - - - - - - -

So, the resultant hash table after inserting the given keys is as shown above.

Q41. What is Hashing? Explain division method to compute the hash function and also explain the collision resolution strategies used in hashing.

Sol41.

Hashing is a technique used in computer science to efficiently map data of arbitrary size to fixed-size values, typically for faster data retrieval. It involves applying a hash function to a key to generate an index or address within a data structure like an array, where the corresponding value or data associated with that key can be stored or retrieved.

Division Method for Hash Function:

The division method is one of the simplest techniques to compute a hash function. It involves taking the key and dividing it by a constant, typically the size of the hash table, and using the remainder as the hash value.

Hash Function:

\[h(k) = k \mod m \]

Where:

- \(h(k) \) is the hash value for key \(k \).

- \(k \) is the key.

- \(m \) is the size of the hash table.

Collision Resolution Strategies:

Collisions occur when two different keys hash to the same index. Several strategies are used to handle collisions:

1. **Chaining:** In chaining, each bucket in the hash table is associated with a linked list. Colliding keys are stored in this linked list at the corresponding bucket.

2. **Open Addressing:**

   - **Linear Probing:** If a collision occurs, linear probing involves placing the colliding key in the next available (empty) slot in the hash table. This process continues until an empty slot is found.

   - **Quadratic Probing:** Similar to linear probing, but the interval between probes is increased by a quadratic function instead of a linear one. This helps to distribute keys more evenly in the hash table.

   - **Double Hashing:** In double hashing, an alternate hash function is used to determine the step size for probing. If a collision occurs, the step size is recomputed using the alternate hash function, and the process continues until an empty slot is found.

3. **Robin Hood Hashing:** Similar to linear probing, but when a new key collides with an existing key, it compares the number of probes needed to insert both keys. If the new key requires fewer probes, it displaces the existing key to minimize the average search time.

Each collision resolution strategy has its trade-offs in terms of simplicity, efficiency, and performance under different scenarios, and the choice depends on factors such as the expected data distribution and the specific requirements of the application.

Q42. Write an algorithm to reverse an integer number using stack.

Sol42.

```
import java.util.Stack;


public class ReverseInteger {
    public static int reverseInteger(int num) {
        // Convert the integer to a string
        String numStr = Integer.toString(num);

        // Create a stack to store the digits
        Stack<Character> stack = new Stack<>();

        // Push each digit of the integer onto the stack
        for (char digit : numStr.toCharArray()) {
```

```
            stack.push(digit);
        }

        // Pop digits from the stack to build the reversed integer string
        StringBuilder reversedNumStr = new StringBuilder();
        while (!stack.isEmpty()) {
            reversedNumStr.append(stack.pop());
        }

        // Convert the reversed integer string back to an integer
        int reversedNum = Integer.parseInt(reversedNumStr.toString());

        return reversedNum;
    }

    public static void main(String[] args) {
        int num = 12345;
        int reversedNum = reverseInteger(num);
        System.out.println("Original number: " + num);
        System.out.println("Reversed number: " + reversedNum);
    }
}
```

Q43. Write an algorithm for Heap Sort. Use Heap sort algorithm, sort the following sequence: 18, 25, 45, 34, 36, 51, 43, and 24.

Sol43.

Here's the algorithm for Heap Sort followed by sorting the given sequence:

Heap Sort Algorithm:

1. Build a max heap from the input array.

2. Swap the root (maximum element) with the last element and remove it from the heap (reducing the heap size by 1).

3. Heapify the root of the heap.

4. Repeat steps 2 and 3 until the heap is empty (heap size becomes 0).

5. The array is now sorted in ascending order.

Heapify Algorithm (to maintain the max heap property):

1. Compare the parent node with its left and right child nodes.

2. If the left child is larger than the parent, swap them.

3. If the right child is larger than the parent (or left child, if already swapped), swap them.

4. Repeat steps 1-3 until the parent is larger than both its children or reaches a leaf node.

Sorting the given sequence (18, 25, 45, 34, 36, 51, 43, 24) using Heap Sort:

1. Build a max heap from the given sequence: [51, 36, 45, 25, 18, 34, 43, 24]

2. Swap the root (51) with the last element (24) and heapify the root: [45, 36, 43, 25, 18, 34, 24, 51]

3. Repeat step 2 until the heap is empty: [24, 25, 34, 36, 43, 45, 51]

4. The sorted sequence is [24, 25, 34, 36, 43, 45, 51].

Here's the Java implementation of Heap Sort:

```java
public class HeapSort {
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
```

```
        heapify(arr, n, i);
    }

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Swap root (maximum element) with last element
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Heapify root element
        heapify(arr, i, 0);
    }
}

public static void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
```

```
        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    public static void main(String[] args) {
        int[] arr = {18, 25, 45, 34, 36, 51, 43, 24};
        heapSort(arr);

        // Print sorted array
        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
Sorted array:
18 24 25 34 36 43 45 51
```

Q44. Write a program in java or algorithm to reverse a singly linked list.

Sol44.

Sure, here's a Java program to reverse a singly linked list:

```java
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

public class ReverseLinkedList {
    public ListNode reverse(ListNode head) {
        ListNode prev = null;
        ListNode current = head;
        ListNode nextNode = null;

        while (current != null) {
            nextNode = current.next; // Save next node
            current.next = prev;     // Reverse the link
            prev = current;          // Move pointers one position ahead
            current = nextNode;      // Move pointers one position ahead
        }

        // Update the head to the last node
        head = prev;
```

```
        return head;
    }

    public static void main(String[] args) {
        // Create a sample linked list: 1 -> 2 -> 3 -> 4 -> 5
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);

        ReverseLinkedList solution = new ReverseLinkedList();
        ListNode reversedHead = solution.reverse(head);

        // Print the reversed linked list
        while (reversedHead != null) {
            System.out.print(reversedHead.val + " ");
            reversedHead = reversedHead.next;
        }
    }
}
```

Output:

```
5 4 3 2 1
```

Q45. From the given preorder traversal of BST, print its postorder traversal. Input : 40 30 32 35 80 90 100 120.

Sol45.

To print the postorder traversal of a binary search tree (BST) given its preorder traversal, we can follow these steps:

1. Construct the BST from the given preorder traversal.

2. Perform a postorder traversal of the constructed BST to print its nodes in the postorder sequence.

Here's a Java program to achieve this:

```java
import java.util.Stack;

class TreeNode {
    int val;
    TreeNode left, right;

    TreeNode(int val) {
        this.val = val;
        left = right = null;
    }
}

public class PreorderToPostorderBST {
    public TreeNode constructBST(int[] preorder) {
        if (preorder == null || preorder.length == 0) {
            return null;
        }
```

```
        TreeNode root = new TreeNode(preorder[0]);
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        for (int i = 1; i < preorder.length; i++) {
            TreeNode temp = null;

            // Keep popping elements from the stack while the current element is greater than the top element
            while (!stack.isEmpty() && preorder[i] > stack.peek().val) {
                temp = stack.pop();
            }

            // If temp is not null, make the current element the right child of temp
            if (temp != null) {
                temp.right = new TreeNode(preorder[i]);
                stack.push(temp.right);
            } else { // Otherwise, make the current element the left child of the top element of the stack
                temp = stack.peek();
                temp.left = new TreeNode(preorder[i]);
                stack.push(temp.left);
            }
        }

        return root;
    }

    public void postorderTraversal(TreeNode root) {
```

```
        if (root == null) {
            return;
        }

        postorderTraversal(root.left);
        postorderTraversal(root.right);
        System.out.print(root.val + " ");
    }

    public static void main(String[] args) {
        int[] preorder = {40, 30, 32, 35, 80, 90, 100, 120};
        PreorderToPostorderBST solution = new PreorderToPostorderBST();
        TreeNode root = solution.constructBST(preorder);

        System.out.println("Postorder traversal of the constructed BST:");
        solution.postorderTraversal(root);
    }
}
```

Output:

```
35 32 30 120 100 90 80 40
```

Q46. Write an algorithm or java method to insert a node at the end in a Doubly Circular linked list.

Sol46.

```
class ListNode {
    int val;
    ListNode prev, next;
```

```
    ListNode(int val) {
        this.val = val;
        this.prev = null;
        this.next = null;
    }
}

public class DoublyCircularLinkedList {
    ListNode head;

    public void insertAtEnd(int val) {
        ListNode newNode = new ListNode(val);

        if (head == null) {
            head = newNode;
            head.prev = head;
            head.next = head;
        } else {
            ListNode last = head.prev; // Get the last node
            last.next = newNode;       // Make the last node point to the new node
            newNode.prev = last;       // Make the new node's previous pointer point to the last
node
            newNode.next = head;       // Make the new node's next pointer point to the head
            head.prev = newNode;       // Make the head's previous pointer point to the new
node
        }
    }

    public void display() {
```

```
        if (head == null) {
            return;
        }

        ListNode current = head;
        do {
            System.out.print(current.val + " ");
            current = current.next;
        } while (current != head);
    }

    public static void main(String[] args) {
        DoublyCircularLinkedList list = new DoublyCircularLinkedList();

        // Insert nodes at the end
        list.insertAtEnd(1);
        list.insertAtEnd(2);
        list.insertAtEnd(3);

        // Display the doubly circular linked list
        list.display();
    }
}
```

OUTPUT:-

1 2 3

Q47. Write an algorithm or java method to insert a node at the end in a Doubly Circular linked list.

Sol47.

```
class ListNode {
```

```
    int val;
    ListNode prev, next;

    ListNode(int val) {
        this.val = val;
        this.prev = null;
        this.next = null;
    }
}

public class DoublyCircularLinkedList {
    ListNode head;

    public void insertAtEnd(int val) {
        ListNode newNode = new ListNode(val);

        if (head == null) {
            head = newNode;
            head.prev = head;
            head.next = head;
        } else {
            ListNode last = head.prev; // Get the last node
            newNode.next = head;       // Make the new node point to the head
            newNode.prev = last;       // Make the new node's previous pointer point to the last
node
            last.next = newNode;       // Make the last node's next pointer point to the new node
            head.prev = newNode;       // Make the head's previous pointer point to the new
node
        }
    }
```

```
    public void display() {
        if (head == null) {
            return;
        }

        ListNode current = head;
        do {
            System.out.print(current.val + " ");
            current = current.next;
        } while (current != head);
    }

    public static void main(String[] args) {
        DoublyCircularLinkedList list = new DoublyCircularLinkedList();

        // Insert nodes at the end
        list.insertAtEnd(1);
        list.insertAtEnd(2);
        list.insertAtEnd(3);

        // Display the doubly circular linked list
        list.display();
    }
}
```

Q48. Write an algorithm to convert an infix expression to a prefix expression. Also, convert the given infix expression to a prefix expression. (A – B/C) * (A/K-L).

Sol48.

To convert an infix expression to a prefix expression, you can use the following algorithm:

1. Reverse the infix expression.
2. Replace each opening parenthesis '(' with a closing parenthesis ')' and vice versa.
3. Obtain the postfix expression from the modified infix expression.
4. Reverse the postfix expression to get the prefix expression.

Now, let's apply this algorithm to the given infix expression: $(A - B/C) \times (A/K-L)$.

1. Reverse the infix expression: $)L-K/A) \times)C/B-A($.
2. Replace '(' with ')' and vice versa: $)L-K/A(\times)C/B-A)$.
3. Obtain the postfix expression: $LKA/-CB-A\times/$.
4. Reverse the postfix expression to get the prefix expression: $\times-A-B/C/A-KL$.

So, the converted prefix expression is $\times-A-B/C/A-KL$.

Here's a Java program that performs the conversion:

```java
import java.util.Stack;

public class InfixToPrefixConverter {

    // Function to check if a character is an operator
    private static boolean isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/';
    }

    // Function to check if a character is an operand
    private static boolean isOperand(char c) {
        return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
```

```
}

// Function to get the precedence of an operator
private static int getPrecedence(char operator) {
    switch (operator) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}

// Function to convert infix to prefix expression
public static String infixToPrefix(String infix) {
    StringBuilder prefix = new StringBuilder();
    Stack<Character> stack = new Stack<>();

    // Step 1: Reverse the infix expression
    StringBuilder reversedInfix = new StringBuilder(infix).reverse();

    // Step 2: Replace '(' with ')' and vice versa
    for (int i = 0; i < reversedInfix.length(); i++) {
        char currentChar = reversedInfix.charAt(i);
        if (currentChar == '(') {
            reversedInfix.setCharAt(i, ')');
```

```
            } else if (currentChar == ')') {
                reversedInfix.setCharAt(i, '(');
            }
        }

        // Step 3: Obtain postfix expression
        for (int i = 0; i < reversedInfix.length(); i++) {
            char currentChar = reversedInfix.charAt(i);

            if (isOperand(currentChar)) {
                prefix.append(currentChar);
            } else if (isOperator(currentChar)) {
                while (!stack.isEmpty() && getPrecedence(stack.peek()) >=
getPrecedence(currentChar)) {
                    prefix.append(stack.pop());
                }
                stack.push(currentChar);
            } else if (currentChar == '(') {
                while (!stack.isEmpty() && stack.peek() != ')') {
                    prefix.append(stack.pop());
                }
                stack.pop(); // Pop the closing parenthesis
            }
        }

        // Pop any remaining operators from the stack
        while (!stack.isEmpty()) {
            prefix.append(stack.pop());
        }
```

```
        // Step 4: Reverse the prefix expression to get the final result
        return prefix.reverse().toString();
    }

    public static void main(String[] args) {
        String infixExpression = "(A - B/C) * (A/K-L)";
        String prefixExpression = infixToPrefix(infixExpression);
        System.out.println("Infix Expression: " + infixExpression);
        System.out.println("Prefix Expression: " + prefixExpression);
    }
}
```

Output:

```
Infix Expression: (A - B/C) * (A/K-L)
Prefix Expression: *-A/BCKL/A
```

Q49. Write an algorithm for converting infix expression into postfix expression. Convert following infix expression into its equivalent postfix expression: A + (B * C – (D / E ^ F) * G) * H

Sol49.

```
import java.util.Stack;

public class InfixToPostfixConverter {
    public static String infixToPostfix(String infix) {
        StringBuilder postfix = new StringBuilder();
        Stack<Character> stack = new Stack<>();

        // Define operator precedence
```

```
        int precedence[] = {1, 1, 2, 2, 3}; // +, -, *, /, ^

        for (char c : infix.toCharArray()) {
            if (Character.isLetterOrDigit(c)) { // Operand
                postfix.append(c);
            } else if (c == '(') {
                stack.push(c);
            } else if (c == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    postfix.append(stack.pop());
                }
                stack.pop(); // Discard the opening parenthesis
            } else { // Operator
                while (!stack.isEmpty() && precedence[operatorIndex(stack.peek())] >=
precedence[operatorIndex(c)]) {
                    postfix.append(stack.pop());
                }
                stack.push(c);
            }
        }

        while (!stack.isEmpty()) {
            postfix.append(stack.pop());
        }

        return postfix.toString();
    }

    private static int operatorIndex(char op) {
        switch (op) {
```

```
            case '+':
                return 0;
            case '-':
                return 1;
            case '*':
                return 2;
            case '/':
                return 3;
            case '^':
                return 4;
            default:
                return -1; // Unknown operator
        }
    }

    public static void main(String[] args) {
        String infixExpression = "A + (B * C - (D / E ^ F) * G) * H";
        String postfixExpression = infixToPostfix(infixExpression);
        System.out.println("Infix Expression: " + infixExpression);
        System.out.println("Postfix Expression: " + postfixExpression);
    }}
```

OUTPUT:-

Infix Expression: A + (B * C - (D / E ^ F) * G) * H

Postfix Expression: ABC*DEF^/G*-H*+

Q50. Write short notes on the following: (i) Priority Queue. (ii) Threaded Binary tree. (iii) Representation of Graph in memory

Sol50.

Sure, here are short notes on each topic:

(i) **Priority Queue**:

- A priority queue is an abstract data type similar to a regular queue or stack, but with each element having a priority associated with it.

- The elements with higher priority are dequeued before elements with lower priority.

- Priority queues can be implemented using various data structures such as binary heaps, Fibonacci heaps, or self-balancing binary search trees.

- They are commonly used in algorithms where ordering by priority is required, such as Dijkstra's shortest path algorithm or Huffman coding.

(ii) **Threaded Binary Tree**:

- A threaded binary tree is a binary tree in which every node has an additional pointer called a thread that either points to the inorder successor (right thread) or inorder predecessor (left thread) of the node.

- Threaded binary trees can be used to perform inorder traversal without using recursion or an explicit stack, making traversal more memory-efficient.

- There are two types of threaded binary trees: singly threaded binary tree (with only right threads) and doubly threaded binary tree (with both left and right threads).

- Threaded binary trees can be constructed from regular binary trees by performing a threading traversal (inorder traversal while creating threads).

(iii) **Representation of Graph in Memory**:

- Graphs can be represented in memory using various data structures, each suitable for different operations and space-time complexities.

- The two most common representations are:

  1. **Adjacency Matrix**: A 2D array where the value at index (i, j) represents the weight of the edge between vertices i and j. Suitable for dense graphs but requires more memory.

  2. **Adjacency List**: An array of linked lists (or dynamic arrays) where each element represents a vertex, and the linked list contains the adjacent vertices. Suitable for sparse graphs and requires less memory.

Q51. Write a java program to implement a circular linked list.

Sol51.

```
class Node {

    int data;
```

```
    Node next;

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

public class CircularLinkedList {
    private Node head;
    private int size;

    public CircularLinkedList() {
        head = null;
        size = 0;
    }

    // Method to insert a node at the end of the circular linked list
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head; // Circular linking
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
```

```
            newNode.next = head; // Circular linking
        }
        size++;
    }

    // Method to display the circular linked list
    public void display() {
        if (head == null) {
            System.out.println("Circular linked list is empty.");
            return;
        }
        Node current = head;
        do {
            System.out.print(current.data + " ");
            current = current.next;
        } while (current != head);
        System.out.println();
    }

    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();
        cll.insert(1);
        cll.insert(2);
        cll.insert(3);
        cll.insert(4)
        System.out.println("Circular linked list:");
        cll.display();
    }
}
```

OUTPUT:-

Circular linked list:

1 2 3 4

Q52. Write a java program to implement a circular linked list.

Sol52.

```
class Node {
    int data;
    Node next;

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

public class CircularLinkedList {
    private Node head;
    private Node tail;

    public CircularLinkedList() {
        head = null;
        tail = null;
    }

    // Method to check if the circular linked list is empty
    public boolean isEmpty() {
        return head == null;
    }
```

```
// Method to insert a node at the end of the circular linked list
public void insert(int data) {
    Node newNode = new Node(data);
    if (isEmpty()) {
        head = newNode;
        tail = newNode;
        newNode.next = head; // Circular linking
    } else {
        tail.next = newNode;
        tail = newNode;
        tail.next = head; // Circular linking
    }
}

// Method to display the circular linked list
public void display() {
    if (isEmpty()) {
        System.out.println("Circular linked list is empty.");
        return;
    }
    Node current = head;
    do {
        System.out.print(current.data + " ");
        current = current.next;
    } while (current != head);
    System.out.println();
}
```

```
    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();
        cll.insert(1);
        cll.insert(2);
        cll.insert(3);
        cll.insert(4);

        System.out.println("Circular linked list:");
        cll.display();
    }
}
```

Output:

Circular linked list:

1 2 3 4

Q53. Define tree, binary tree, complete binary tree and full binary tree. Write algorithms or function to obtain traversals of a binary tree in preorder, postorder and inorder.

Sol53.

1. **Tree**:

   - A tree is a hierarchical data structure consisting of nodes connected by edges.

   - It has a root node, and each node has zero or more child nodes.

   - Nodes are connected in a way that there is exactly one path between any two nodes.

2. **Binary Tree**:

   - A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.

3. **Complete Binary Tree**:

   - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- In a complete binary tree, the last level is filled from left to right.

4. **Full Binary Tree**:

- A full binary tree is a binary tree in which every node other than the leaves has two children.

Now, let's provide algorithms or functions for obtaining traversals of a binary tree in preorder, postorder, and inorder:

- **Preorder Traversal**:
  - Visit the root node.
  - Traverse the left subtree recursively.
  - Traverse the right subtree recursively.

- **Postorder Traversal**:
  - Traverse the left subtree recursively.
  - Traverse the right subtree recursively.
  - Visit the root node.

- **Inorder Traversal**:
  - Traverse the left subtree recursively.
  - Visit the root node.
  - Traverse the right subtree recursively.

Here are the algorithms/functions in Java for obtaining traversals of a binary tree:

```java
class TreeNode {
    int val;
    TreeNode left, right;
```

```
    TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}

public class BinaryTreeTraversal {
    // Preorder Traversal
    public static void preorder(TreeNode root) {
        if (root == null) {
            return;
        }
        System.out.print(root.val + " ");
        preorder(root.left);
        preorder(root.right);
    }

    // Postorder Traversal
    public static void postorder(TreeNode root) {
        if (root == null) {
            return;
        }
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.val + " ");
    }
```

```
// Inorder Traversal
public static void inorder(TreeNode root) {
    if (root == null) {
        return;
    }
    inorder(root.left);
    System.out.print(root.val + " ");
    inorder(root.right);
}

public static void main(String[] args) {
    // Constructing a binary tree
    TreeNode root = new TreeNode(1);
    root.left = new TreeNode(2);
    root.right = new TreeNode(3);
    root.left.left = new TreeNode(4);
    root.left.right = new TreeNode(5);

    // Preorder traversal
    System.out.print("Preorder Traversal: ");
    preorder(root);
    System.out.println();

    // Postorder traversal
    System.out.print("Postorder Traversal: ");
    postorder(root);
    System.out.println();

    // Inorder traversal
```

```
        System.out.print("Inorder Traversal: ");
        inorder(root);
        System.out.println();
    }
}
```

Output:

Preorder Traversal: 1 2 4 5 3

Postorder Traversal: 4 5 2 3 1

Inorder Traversal: 4 2 5 1 3

Q54. Design a method for keeping two stacks within a single linear array so that neither stack overflow until all the memory is used.

Sol54.

To design a method for keeping two stacks within a single linear array so that neither stack overflows until all the memory is used, we can use a "two ends" approach. In this approach, each stack starts from one end of the array and grows towards the center. When the stacks meet in the middle, we can check if there is still space available in the array and redistribute the elements accordingly.

Here's a high-level overview of the method:

1. Initialize two pointers, one pointing to the beginning of the array (for stack 1) and the other pointing to the end of the array (for stack 2).

2. When pushing elements into either stack, increment or decrement the respective pointer accordingly.

3. When popping elements from either stack, decrement or increment the respective pointer accordingly.

4. Before pushing an element into either stack, check if the stacks meet in the middle of the array. If they do, check if there is still space available in the array.

5. If there is space available, redistribute the elements by moving them towards the opposite ends of the array.

6. Continue pushing and popping elements until all memory is used.

Here's a Java implementation of this method:

```java
public class TwoStacksInArray {
    private int[] array;
    private int top1, top2;
    private int capacity;

    public TwoStacksInArray(int capacity) {
        this.capacity = capacity;
        this.array = new int[capacity];
        this.top1 = -1;
        this.top2 = capacity;
    }

    public void pushStack1(int data) {
        if (top1 + 1 < top2) {
            array[++top1] = data;
        } else {
            System.out.println("Stack 1 overflow");
        }
    }

    public void pushStack2(int data) {
        if (top2 - 1 > top1) {
            array[--top2] = data;
        } else {
            System.out.println("Stack 2 overflow");
        }
```

```
}

public int popStack1() {
    if (top1 >= 0) {
        return array[top1--];
    } else {
        System.out.println("Stack 1 underflow");
        return -1;
    }
}

public int popStack2() {
    if (top2 < capacity) {
        return array[top2++];
    } else {
        System.out.println("Stack 2 underflow");
        return -1;
    }
}

public static void main(String[] args) {
    TwoStacksInArray stacks = new TwoStacksInArray(5);
    stacks.pushStack1(1);
    stacks.pushStack2(2);
    stacks.pushStack1(3);
    stacks.pushStack2(4);
    stacks.pushStack1(5);
    stacks.pushStack2(6); // Stack 2 overflow
    System.out.println("Popped from stack 1: " + stacks.popStack1()); // 5
```

```
        System.out.println("Popped from stack 2: " + stacks.popStack2()); // 4
        stacks.pushStack2(6); // Stack 2 overflow
    }
}
```

Output:

```
Stack 2 overflow
Popped from stack 1: 5
Popped from stack 2: 4
Stack 2 overflow
```