ASSIGNMENT 3

Solve the assignment with following thing to be added in each question.

-Program

-Flow chart

-Explanation

-Output

-Time and Space complexity

1. Implement a Stack using an array.

Test Case 1:

Input: Push 5, 3, 7, Pop

Output: Stack = [5, 3], Popped element = 7

Test Case 2:

Input: Push 10, Push 20, Pop, Push 15

Output: Stack = [10, 15], Popped element = 20

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

    // Constructor to initialize the stack
    public Stack(int size) {
        maxSize = size;
        stack = new int[maxSize];
        top = -1; // Stack is initially empty
    }
```

```java
// Push an element onto the stack
public void push(int value) {
    if (top == maxSize - 1) {
        System.out.println("Stack Overflow");
    } else {
        stack[++top] = value;
    }
}

// Pop an element from the stack
public int pop() {
    if (top == -1) {
        System.out.println("Stack Underflow");
        return -1;
    } else {
        return stack[top--];
    }
}

// Display the current stack
public void display() {
    if (top == -1) {
        System.out.println("Stack is empty");
    } else {
        System.out.print("Stack = [");
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i]);
            if (i < top) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }
}

public static void main(String[] args) {
    Stack stack = new Stack(5);

    // Test Case 1
    stack.push(5);
    stack.push(3);
    stack.push(7);
```

```
        int poppedElement1 = stack.pop();
        stack.display();
        System.out.println("Popped element = " + poppedElement1);

        // Test Case 2
        stack.push(10);
        stack.push(20);
        int poppedElement2 = stack.pop();
        stack.push(15);
        stack.display();
        System.out.println("Popped element = " + poppedElement2);
    }
}
```

Stack = [5, 3]

Popped element = 7

Stack = [10, 15]

Popped element = 20

Flowchart:

- Start
- Initialize stack with given size
- Push or pop based on input
- If push, check for stack overflow and insert element at `top+1`
- If pop, check for stack underflow and remove element from `top`
- Repeat operations until all operations are performed
- End

Explanation

This program implements a stack using a fixed-size array. The stack supports the following operations:

- **Push**: Adds an element to the top of the stack, checking for overflow.
- **Pop**: Removes and returns the element at the top, checking for underflow.
- **Display**: Shows the current stack content.

initialize the stack with a `maxSize`, and the `top` pointer tracks the current top of the stack (starting from -1). In the test case, push and pop elements from the stack and print the stack's state and popped elements.

- **Time Complexity**:
- Push: O(1) (constant time operation)
- Pop: O(1)
- Display: O(n) where n is the number of elements in the stack
- **Space Complexity**: O(n), where n is the maximum size of the stack.

2. Check for balanced parentheses using a stack.

Test Case 1:

Input: "({[()]})"

Output: Balanced

Test Case 2:

Input: "([)]"

Output: Not Balanced

```java
import java.util.Stack;

public class BalancedParentheses {

    // Method to check for balanced parentheses
    public static boolean isBalanced(String expression) {
        Stack<Character> stack = new Stack<>();

        for (char ch : expression.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty()) {
                    return false;
                }
                char openBracket = stack.pop();
                if (!isMatchingPair(openBracket, ch)) {
                    return false;
                }
            }
        }

        return stack.isEmpty(); // Stack should be empty if balanced
    }

    // Helper method to check if the pair of brackets is matching
    private static boolean isMatchingPair(char open, char close) {
        return (open == '(' && close == ')') || (open == '{' && close == '}') ||
(open == '[' && close == ']');
    }

    public static void main(String[] args) {
        // Test Case 1
        String expression1 = "({[()]})";
        System.out.println("Is balanced: " + isBalanced(expression1)); //
Expected Output: true

        // Test Case 2
        String expression2 = "([)]";
        System.out.println("Is balanced: " + isBalanced(expression2)); //
Expected Output: false
    }
}
```

Is balanced: true

Is balanced: false

Flowchart:

1. Start
2. Initialize an empty stack
3. Traverse the string:
    a. If the current character is an opening bracket, push it to the stack
    b. If it's a closing bracket, check if the stack is empty or the top element doesn't match
4. At the end, check if the stack is empty (balanced) or not (unbalanced)
5. End

Step 3: Explanation

This program checks whether a given expression contains balanced parentheses. It uses a stack to keep track of the opening brackets. For every closing bracket, it checks whether the top of the stack has a matching opening bracket. If the stack is empty or mismatched at any point, the expression is considered unbalanced. If at the end the stack is empty, the expression is balanced.

- **Time Complexity**: O(n), where n is the length of the input string.
- **Space Complexity**: O(n), for the stack used to store opening parentheses.

3. Reverse a string using a stack.

Test Case 1:

Input: "hello"

Output: "olleh"

Test Case 2:

Input: "world"

Output: "dlrow"

```java
import java.util.Stack;

public class ReverseString {

    // Method to reverse a string using a stack
    public static String reverseString(String str) {
        Stack<Character> stack = new Stack<>();

        // Push all characters of the string into the stack
        for (char ch : str.toCharArray()) {
            stack.push(ch);
        }

        // Pop characters from the stack and build the reversed string
        StringBuilder reversed = new StringBuilder();
        while (!stack.isEmpty()) {
            reversed.append(stack.pop());
        }
        return reversed.toString();
    }

    public static void main(String[] args) {
        // Test Case 1
        String input1 = "hello";
        System.out.println("Reversed string: " + reverseString(input1)); //
Expected Output: "olleh"

        // Test Case 2
        String input2 = "world";
        System.out.println("Reversed string: " + reverseString(input2)); //
Expected Output: "dlrow"
    }
}
```

Flowchart

1. Start
2. Initialize an empty stack
3. Traverse the string:
   a. Push each character onto the stack
4. Pop each character from the stack to form the reversed string
5. End

Step 3: Explanation

This program reverses a string using a stack. It works by pushing all characters of the input string onto a stack and then popping them off in reverse order, thereby constructing the reversed string. The stack ensures that the characters are retrieved in the opposite order to their insertion.

- **Time Complexity**: O(n), where n is the length of the input string.
- **Space Complexity**: O(n), for the stack storing n characters.

4. Evaluate a postfix expression using a stack.

Test Case 1:

Input: "5 3 + 2 *"

Output: 16

Test Case 2:

Input: "4 5 * 6 /"

Output: 3

```java
import java.util.Stack;

public class PostfixEvaluation {

    // Method to evaluate a postfix expression
    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();

        for (char ch : expression.split(" ")) {
            if (Character.isDigit(ch)) {
                stack.push(Character.getNumericValue(ch));
            } else {
                int operand2 = stack.pop();
                int operand1 = stack.pop();
                switch (ch) {
                    case '+':
                        stack.push(operand1 + operand2);
                        break;
                    case '-':
                        stack.push(operand1 - operand2);
                        break;
                    case '*':
                        stack.push(operand1 * operand2);
                        break;
                    case '/':
                        stack.push(operand1 / operand2);
                        break;
                }
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        // Test Case 1
        String expression1 = "5 3 + 2 *";
        System.out.println("Postfix Evaluation: " +
evaluatePostfix(expression1)); // Output: 16

        // Test Case 2
        String expression2 = "4 5 * 6 /";
        System.out.println("Postfix Evaluation: " +
evaluatePostfix(expression2)); // Output: 3
    }
}
```

Flowchart

1. Start
2. Traverse the postfix expression:
    a. Push operands onto the stack
    b. When encountering an operator, pop two operands, apply the operation, and push the result
3. After processing, the final result is at the top of the stack
4. End

Step 3: Explanation

This program evaluates a postfix expression using a stack. The algorithm scans the postfix expression from left to right, pushing numbers onto the stack and applying operations when encountering operators. It pops two operands, performs the operation, and pushes the result back. At the end of the expression, the stack contains the final result.

- **Time Complexity**: O(n), where n is the number of characters in the postfix expression.
- **Space Complexity**: O(n), for storing operands in the stack.

5. Convert an infix expression to postfix using a stack.

Test Case 1:

Input: "A + B * C"

Output: "A B C * +"

Test Case 2:

Input: "A * B + C / D"

Output: "A B * C D / +"

```
import java.util.Stack;

public class InfixToPostfix {

    // Method to check the precedence of operators
    private static int precedence(char operator) {
        switch (operator) {
            case '+':
```

```java
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            case '^':
                return 3;
        }
        return -1;
    }

    // Method to convert infix to postfix expression
    public static String convertToPostfix(String expression) {
        StringBuilder result = new StringBuilder();
        Stack<Character> stack = new Stack<>();

        for (char ch : expression.toCharArray()) {
            if (Character.isLetterOrDigit(ch)) {
                result.append(ch);
            } else if (ch == '(') {
                stack.push(ch);
            } else if (ch == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    result.append(stack.pop());
                }
                stack.pop(); // Remove '(' from the stack
            } else {
                while (!stack.isEmpty() && precedence(ch) <=
precedence(stack.peek())) {
                    result.append(stack.pop());
                }
                stack.push(ch);
            }
        }

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

        return result.toString();
    }

    public static void main(String[] args) {
        // Test Case 1
        String expression1 = "A+B*C";
        System.out.println("Postfix: " + convertToPostfix(expression1)); //
Output: A B C * +

        // Test Case 2
        String expression2 = "A*B+C/D";
        System.out.println("Postfix: " + convertToPostfix(expression2)); //
Output: A B * C D / +
    }
}
```

Flowchart

1. Start
2. Traverse the infix expression:
    a. Push opening brackets and operators to the stack
    b. Append operands directly to the output
    c. Pop from the stack for closing brackets or lower precedence operators
3. Append remaining stack contents to the output
4. End

Step 3: Explanation

This program converts an infix expression to a postfix expression using a stack. The algorithm scans the infix expression, pushes operators onto the stack while considering precedence, and outputs operands directly. When encountering closing brackets or lower-precedence operators, it pops operators from the stack until the conditions are met.

- **Time Complexity**: O(n), where n is the number of characters in the infix expression.
- **Space Complexity**: O(n), for the stack used to store operators.

6. Implement a Queue using an array.

Test Case 1:

Input: Enqueue 5, Enqueue 10, Dequeue

Output: Queue = [10], Dequeued element = 5

Test Case 2:

Input: Enqueue 1, 2, 3, Dequeue, Dequeue

Output: Queue = [3], Dequeued elements = 1, 2

```java
public class Queue {
    private int[] queue;
    private int front;
    private int rear;
    private int maxSize;
    private int currentSize;

    // Constructor to initialize the queue
    public Queue(int size) {
        maxSize = size;
        queue = new int[maxSize];
        front = 0;
        rear = -1;
        currentSize = 0;
    }

    // Enqueue an element to the queue
    public void enqueue(int value) {
        if (currentSize == maxSize) {
            System.out.println("Queue Overflow");
        } else {
            rear = (rear + 1) % maxSize;
            queue[rear] = value;
            currentSize++;
        }
    }

    // Dequeue an element from the queue
    public int dequeue() {
        if (currentSize == 0) {
            System.out.println("Queue Underflow");
            return -1;
        } else {
            int value = queue[front];
            front = (front + 1) % maxSize;
            currentSize--;
            return value;
        }
    }

    // Display the current queue
    public void display() {
        if (currentSize == 0) {
            System.out.println("Queue is empty");
        } else {
            System.out.print("Queue = [");
            for (int i = 0; i < currentSize; i++) {
                System.out.print(queue[(front + i) % maxSize]);
                if (i < currentSize - 1) {
                    System.out.print(", ");
                }
            }
            System.out.println("]");
        }
    }

    public static void main(String[] args) {
        Queue queue = new Queue(5);

        // Test Case 1
        queue.enqueue(5);
        queue.enqueue(10);
        int dequeuedElement1 = queue.dequeue();
        queue.display();
        System.out.println("Dequeued element = " + dequeuedElement1);

        // Test Case 2
        queue.enqueue(1);
        queue.enqueue(2);
        queue.enqueue(3);
        int dequeuedElement2 = queue.dequeue();
        int dequeuedElement3 = queue.dequeue();
        queue.display();
        System.out.println("Dequeued elements = " + dequeuedElement2 + ", " +
dequeuedElement3);
    }
}
```

Flowchart

1. Start
2. Initialize the queue with a maximum size
3. Enqueue or dequeue based on input
    a. If enqueue, check for overflow and insert element at `rear`
    b. If dequeue, check for underflow and remove element from `front`
4. End

Step 3: Explanation

This program implements a queue using a circular array. It supports enqueue, dequeue, and display operations. The `enqueue` method adds an element to the queue, wrapping around if needed (circular behavior), and `dequeue` removes the front element. The program maintains the size of the queue and tracks the front and rear indices.

- **Time Complexity**:
- Enqueue: O(1)
- Dequeue: O(1)
- Display: O(n) where n is the current size of the queue
- **Space Complexity**: O(n), where n is the maximum size of the queue.

7. Implement a Circular Queue using an array.

Test Case 1:

Input: Enqueue 4, 5, 6, 7, Dequeue, Enqueue 8

Output: Queue = [8, 5, 6, 7]

Test Case 2:

Input: Enqueue 1, 2, 3, 4, Dequeue, Dequeue, Enqueue 5

Output: Queue = [5, 3, 4]

```
public class CircularQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int maxSize;
    private int currentSize;

    // Constructor to initialize the circular queue
    public CircularQueue(int size) {
        maxSize = size;
        queue = new int[maxSize];
        front = 0;
        rear = -1;
        currentSize = 0;
    }
```

```java
    // Enqueue an element to the circular queue
    public void enqueue(int value) {
        if (currentSize == maxSize) {
            System.out.println("Queue Overflow");
        } else {
            rear = (rear + 1) % maxSize;
            queue[rear] = value;
            currentSize++;
        }
    }

    // Dequeue an element from the circular queue
    public int dequeue() {
        if (currentSize == 0) {
            System.out.println("Queue Underflow");
            return -1;
        } else {
            int value = queue[front];
            front = (front + 1) % maxSize;
            currentSize--;
            return value;
        }
    }

    // Display the current queue
    public void display() {
        if (currentSize == 0) {
            System.out.println("Queue is empty");
        } else {
            System.out.print("Queue = [");
            for (int i = 0; i < currentSize; i++) {
                System.out.print(queue[(front + i) % maxSize]);
                if (i < currentSize - 1) {
                    System.out.print(", ");
                }
            }
            System.out.println("]");
        }
    }

    public static void main(String[] args) {
        CircularQueue circularQueue = new CircularQueue(5);
        // Test Case 1
        circularQueue.enqueue(4);
        circularQueue.enqueue(5);
        circularQueue.enqueue(6);
        circularQueue.enqueue(7);
        circularQueue.dequeue();
        circularQueue.enqueue(8);
        circularQueue.display(); // Expected Output: [5, 6, 7, 8]

        // Test Case 2
        circularQueue.enqueue(1);
        circularQueue.enqueue(2);
        circularQueue.enqueue(3);
        circularQueue.dequeue();
        circularQueue.dequeue();
        circularQueue.enqueue(5);
        circularQueue.display(); // Expected Output: [3, 5]
    }
}
```

Flowchart

1. Start
2. Initialize the circular queue with a maximum size
3. Enqueue or dequeue based on input:
    a. If enqueue, insert the element at `rear` and wrap around if needed
    b. If dequeue, remove the element from `front` and adjust indices
4. End

Step 3: Explanation

This program implements a circular queue using an array. The queue supports enqueue and dequeue operations with wrap-around behavior when the rear or front reaches the end of the array. The program manages the `front` and `rear` pointers and uses modulo arithmetic to maintain circular behavior.

Time and Space Complexity

- **Time Complexity**:
    - Enqueue: O(1)
    - Dequeue: O(1)
    - Display: O(n), where n is the current size of the queue
- **Space Complexity**: O(n), where n is the maximum size of the queue.

8. Implement a Queue using two Stacks.

Test Case 1:

Input: Enqueue 3, Enqueue 7, Dequeue

Output: Queue = [7], Dequeued element = 3

Test Case 2:

Input: Enqueue 10, 20, Dequeue, Dequeue

Output: Queue = [], Dequeued elements = 10, 20

```java
import java.util.Stack;
```

```java
public class QueueUsingTwoStacks {
    Stack<Integer> stack1 = new Stack<>();
    Stack<Integer> stack2 = new Stack<>();

    // Enqueue an element to the queue
    public void enqueue(int value) {
        stack1.push(value);
    }

    // Dequeue an element from the queue
    public int dequeue() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        if (!stack2.isEmpty()) {
            return stack2.pop();
        } else {
            System.out.println("Queue is empty");
            return -1;
        }
    }

    // Display the current queue
    public void display() {
        if (stack1.isEmpty() && stack2.isEmpty()) {
            System.out.println("Queue is empty");
        } else {
            System.out.print("Queue = [");
            Stack<Integer> tempStack = new Stack<>();
            while (!stack2.isEmpty()) {
                tempStack.push(stack2.pop());
            }
            while (!tempStack.isEmpty()) {
                System.out.print(tempStack.peek() + " ");
                stack2.push(tempStack.pop());
            }
            for (int i = stack1.size() - 1; i >= 0; i--) {
                System.out.print(stack1.get(i));
                if (i > 0) {
                    System.out.print(", ");
                }
            }
            System.out.println("]");
        }
    }

    public static void main(String[] args) {
        QueueUsingTwoStacks queue = new QueueUsingTwoStacks();

        // Test Case 1
        queue.enqueue(3);
        queue.enqueue(7);
        int dequeuedElement1 = queue.dequeue();
        queue.display(); // Expected Output: [7]
        System.out.println("Dequeued element = " + dequeuedElement1);

        // Test Case 2
        queue.enqueue(10);
        queue.enqueue(20);
        int dequeuedElement2 = queue.dequeue();
        int dequeuedElement3 = queue.dequeue();
        queue.display(); // Expected Output: []
        System.out.println("Dequeued elements = " + dequeuedElement2 + ", " +
dequeuedElement3);
    }
}
```

Flowchart

1. Start
2. Enqueue elements to `stack1`
3. Dequeue:
    a. If `stack2` is empty, transfer all elements from `stack1` to `stack2` and pop from `stack2`
4. End

Step 3: Explanation

This program implements a queue using two stacks. The first stack (`stack1`) is used for enqueue operations, and the second stack (`stack2`) is used for dequeue operations. When dequeue is called, if `stack2` is empty, elements are transferred from `stack1` to `stack2`, reversing their order and ensuring the correct order for dequeuing.

- **Time Complexity**:
- Enqueue: O(1)
- Dequeue: O(n) in the worst case when transferring elements from `stack1` to `stack2`
- **Space Complexity**: O(n), where n is the number of elements in the queue.

9. Implement a Min-Heap.

Test Case 1:

Input: Insert 10, 15, 20, 17, Extract Min

Output: Min-Heap = [15, 17, 20], Extracted Min = 10

Test Case 2:

Input: Insert 30, 40, 20, 50, Extract Min

Output: Min-Heap = [30, 40, 50], Extracted Min = 20

```
import java.util.PriorityQueue;

public class MinHeap {
    // Method to insert elements into the min-heap
    public static void insert(PriorityQueue<Integer> minHeap, int value) {
        minHeap.add(value);
    }

    // Method to extract the minimum element from the heap
    public static int extractMin(PriorityQueue<Integer> minHeap) {
        return minHeap.poll();
    }

    public static void main(String[] args) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Test Case 1
        insert(minHeap, 10);
        insert(minHeap, 15);
        insert(minHeap, 20);
        insert(minHeap, 17);
        int extractedMin1 = extractMin(minHeap);
        System.out.println("Min-Heap = " + minHeap); // Expected Output: [15, 17,
20]
        System.out.println("Extracted Min = " + extractedMin1); // Expected
Output: 10

        // Test Case 2
        insert(minHeap, 30);
        insert(minHeap, 40);
        insert(minHeap, 20);
        insert(minHeap, 50);
        int extractedMin2 = extractMin(minHeap);
        System.out.println("Min-Heap = " + minHeap); // Expected Output: [30, 40,
50]
        System.out.println("Extracted Min = " + extractedMin2); // Expected
Output: 20
    }
}
```

Flowchart

1. Start
2. Insert elements into the heap
3. Extract the minimum element from the heap
4. End

Step 3: Explanation

This program implements a Min-Heap using Java's `PriorityQueue` class. Elements are inserted into the heap, and the minimum element can be extracted using the `poll()` method, which retrieves and removes the smallest element while maintaining the heap structure.

- **Time Complexity**:
- Insertion: O(log n)
- Extraction: O(log n)
- **Space Complexity**: O(n), where n is the number of elements in the heap.

10. Implement a Max-Heap.

Test Case 1:

Input: Insert 12, 7, 15, 5, Extract Max

Output: Max-Heap = [12, 7, 5], Extracted Max = 15

Test Case 2:

Input: Insert 8, 20, 10, 3, Extract Max

Output: Max-Heap = [10, 8, 3], Extracted Max = 20

```java
import java.util.Collections;
import java.util.PriorityQueue;

public class MaxHeap {

    // Method to insert elements into the max-heap
    public static void insert(PriorityQueue<Integer> maxHeap, int value) {
        maxHeap.add(value);
    }

    // Method to extract the maximum element from the heap
    public static int extractMax(PriorityQueue<Integer> maxHeap) {
        return maxHeap.poll();
    }

    public static void main(String[] args) {
        // Max-Heap using Java's PriorityQueue with reverse order
        PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Collections.reverseOrder());

        // Test Case 1
        insert(maxHeap, 12);
        insert(maxHeap, 7);
        insert(maxHeap, 15);
        insert(maxHeap, 5);
        int extractedMax1 = extractMax(maxHeap);
        System.out.println("Max-Heap = " + maxHeap); // Expected Output: [12, 7,
5]
        System.out.println("Extracted Max = " + extractedMax1); // Expected
Output: 15

        // Test Case 2
        insert(maxHeap, 8);
        insert(maxHeap, 20);
        insert(maxHeap, 10);
        insert(maxHeap, 3);
        int extractedMax2 = extractMax(maxHeap);
        System.out.println("Max-Heap = " + maxHeap); // Expected Output: [10, 8,
3]
        System.out.println("Extracted Max = " + extractedMax2); // Expected
Output: 20
```

```
    }
}
```

Flowchart

1. **Start**
2. **Insert elements into the max-heap**
    a. Use `insert` method to add elements.
3. **Extract the maximum element**
    a. Use `extractMax` method to remove the maximum.
4. **Display the heap and extracted element**
5. **End**

Step 3: Explanation

This program implements a Max-Heap using Java's `PriorityQueue` with a custom comparator (`Collections.reverseOrder()`) to maintain the heap in descending order. The `insert` method adds elements to the heap, and the `extractMax` method removes the maximum element (the root of the heap). The `main` method tests the implementation with the provided test cases.

- **Time Complexity**:
- Insertion: O(log n)
- Extraction: O(log n)
- **Space Complexity**: O(n), where n is the number of elements in the heap.

11. Sort an array using a heap (Heap Sort).

Test Case 1:

Input: [5, 1, 12, 3, 9]

Output: [1, 3, 5, 9, 12]

Test Case 2:

Input: [20, 15, 8, 10]

Output: [8, 10, 15, 20]

```java
import java.util.PriorityQueue;

public class HeapSort {

    // Method to sort an array using heap sort
    public static void heapSort(int[] arr) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Insert all elements into the min-heap
        for (int value : arr) {
            minHeap.add(value);
        }

        // Extract the elements back into the array in sorted order
        int index = 0;
        while (!minHeap.isEmpty()) {
            arr[index++] = minHeap.poll();
        }
    }

    public static void main(String[] args) {
        // Test Case 1
        int[] arr1 = {5, 1, 12, 3, 9};
        heapSort(arr1);
        System.out.print("Sorted Array: ");
        for (int value : arr1) {
            System.out.print(value + " ");
        }
        System.out.println(); // Expected Output: [1, 3, 5, 9, 12]

        // Test Case 2
        int[] arr2 = {20, 15, 8, 10};
        heapSort(arr2);
        System.out.print("Sorted Array: ");
        for (int value : arr2) {
            System.out.print(value + " ");
        }
        System.out.println(); // Expected Output: [8, 10, 15, 20]
    }
}
```

Flowchart

1. **Start**
2. **Build a max-heap from the array**
   a. Call `heapify` for all non-leaf nodes starting from `n/2 - 1` down to `0`.
3. **Extract elements from the heap one by one**
   a. Swap the root (largest value) with the last element.
   b. Reduce the heap size by one.
   c. Heapify the root element again.
4. **Repeat until the heap size is 1**
5. **End**

Step 3: Explanation

Heap Sort is a comparison-based sorting technique based on a binary heap data structure. The algorithm first builds a max-heap from the input data and then repeatedly extracts the maximum element from the heap and rebuilds the heap until all elements are sorted. The `heapify` function ensures the subtree rooted at a given node satisfies the heap property.

Time and Space Complexity

- **Time Complexity**: O(n log n)
- **Space Complexity**: O(1) (in-place sorting)

12. Find the kth largest element in a stream of numbers using a heap.

Test Case 1:

Input: Stream = [3, 10, 5, 20, 15], k = 3

Output: 10

Test Case 2:

Input: Stream = [7, 4, 8, 2, 9], k = 2

Output: 8

```
import java.util.PriorityQueue;

public class KthLargestElement {

    // Method to find the kth largest element in a stream
    public static int findKthLargest(int[] stream, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);

        for (int value : stream) {
            if (minHeap.size() < k) {
                minHeap.add(value);
            } else if (value > minHeap.peek()) {
                minHeap.poll();
                minHeap.add(value);
            }
        }

        return minHeap.peek(); // The root of the heap will be the kth largest
    }

    public static void main(String[] args) {
        // Test Case 1
        int[] stream1 = {3, 10, 5, 20, 15};
        int k1 = 3;
        int result1 = findKthLargest(stream1, k1);
        System.out.println("kth largest element = " + result1); // Expected
Output: 10

        // Test Case 2
        int[] stream2 = {7, 4, 8, 2, 9};
        int k2 = 2;
        int result2 = findKthLargest(stream2, k2);
        System.out.println("kth largest element = " + result2); // Expected
Output: 8
    }
}
```

Flowchart

1. **Start**
2. **Initialize a min-heap with capacity k**
3. **For each number in the stream:**
    a. If heap size < k, add number to heap
    b. Else if number > heap's root, replace root with the new number
4. **After processing all numbers, the root of the heap is the kth largest**
5. **End**

Step 3: Explanation

The program maintains a min-heap of size k. As it processes the stream, it keeps only the k largest elements seen so far. The smallest element in the heap (the root) is the kth largest element in the stream. By the end of the stream, the root of the heap represents the kth largest element overall.

- **Time Complexity**: O(n log k), where n is the number of elements in the stream.
- **Space Complexity**: O(k), for the heap.

13. Implement a Priority Queue using a heap.

Test Case 1:

Input: Enqueue with priorities: 3 (priority 1), 10 (priority 3), 5 (priority 2), Dequeue

Output: Dequeued element = 10 (highest priority), Priority Queue = [5, 3]

Test Case 2:

Input: Enqueue with priorities: 7 (priority 4), 8 (priority 2), 6 (priority 3), Dequeue

Output: Dequeued element = 7, Priority Queue = [6, 8]

```java
import java.util.PriorityQueue;
import java.util.Comparator;

class Element {
    int value;
    int priority;

    public Element(int value, int priority) {
        this.value = value;
        this.priority = priority;
    }
}

public class PriorityQueueHeap {

    // Comparator for the priority queue to order by priority
    public static class ElementComparator implements Comparator<Element> {
        @Override
        public int compare(Element e1, Element e2) {
            return Integer.compare(e2.priority, e1.priority); // Max-Heap based
on priority
        }
    }

    public static void enqueue(PriorityQueue<Element> priorityQueue, int value,
int priority) {
        priorityQueue.add(new Element(value, priority));
    }

    public static Element dequeue(PriorityQueue<Element> priorityQueue) {
        return priorityQueue.poll();
    }

    public static void main(String[] args) {
        // Priority queue with custom comparator based on priority
        PriorityQueue<Element> priorityQueue = new PriorityQueue<>(new
ElementComparator());

        // Test Case 1
        enqueue(priorityQueue, 3, 1);
        enqueue(priorityQueue, 10, 3);
        enqueue(priorityQueue, 5, 2);
        Element dequeuedElement1 = dequeue(priorityQueue);
        System.out.println("Dequeued element = " + dequeuedElement1.value + "
(priority " + dequeuedElement1.priority + ")");
        System.out.println("Priority Queue size = " + priorityQueue.size());

        // Test Case 2
        enqueue(priorityQueue, 7, 4);
        enqueue(priorityQueue, 8, 2);
        enqueue(priorityQueue, 6, 3);
        Element dequeuedElement2 = dequeue(priorityQueue);
        System.out.println("Dequeued element = " + dequeuedElement2.value + "
(priority " + dequeuedElement2.priority + ")");
        System.out.println("Priority Queue size = " + priorityQueue.size());
    }
}
```

Flowchart

1. **Start**
2. **Define a custom comparator for priority**
3. **Initialize a priority queue with the comparator**
4. **Enqueue elements with their priorities**
5. **Dequeue elements based on priority**
6. **Display dequeued element and remaining queue**
7. **End**

Step 3: Explanation

This program implements a priority queue using a heap. Elements are inserted into the priority queue along with their priorities. The priority queue is implemented using Java's `PriorityQueue` with a custom comparator to sort the elements based on their priority (higher priority first). Dequeue operations remove the element with the highest priority.

- **Time Complexity**:
  - Enqueue: O(log n)

    - Dequeue: O(log n)
- **Space Complexity**: O(n), where n is the number of elements in the priority queue.

14. Design an algorithm to implement a stack with a getMin() function to return the minimum element in constant time.

Test Case 1:

Input: Push 5, Push 3, Push 7, Get Min

Output: Min = 3

Test Case 2:

Input: Push 10, Push 8, Push 6, Push 12, Get Min

Output: Min = 6

```java
import java.util.Stack;

public class MinStack {
    Stack<Integer> stack = new Stack<>();
    Stack<Integer> minStack = new Stack<>();

    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public void pop() {
        if (stack.peek().equals(minStack.peek())) {
            minStack.pop();
        }
        stack.pop();
    }
    public int getMin() {
        return minStack.peek();
    }

    public static void main(String[] args) {
        MinStack minStack = new MinStack();

        // Test Case 1
        minStack.push(5);
        minStack.push(3);
        minStack.push(7);
        System.out.println("Minimum = " + minStack.getMin()); // Expected Output:
3


        // Test Case 2
        minStack.push(10);
        minStack.push(8);
        minStack.push(6);
        minStack.push(12);
        System.out.println("Minimum = " + minStack.getMin()); // Expected Output:
6
    }
}
```

Flowchart

1. **Start**
2. **Initialize two stacks: mainStack and minStack**
3. **Push operation:**
    a. Push to mainStack
    b. If minStack is empty or new element <= minStack.peek(), push to minStack
4. **Pop operation:**
    a. Pop from mainStack

        b. If popped element == minStack.peek(), pop from minStack
    5. **getMin operation:**
        a. Return minStack.peek()
    6. **End**

Step 3: Explanation

The `MinStack` class maintains two stacks: one for all the elements (`mainStack`) and one for tracking the minimum elements (`minStack`). When pushing a new element, it is compared with the current minimum. If it is smaller or equal, it is also pushed onto `minStack`. When popping, if the popped element is the current minimum, it is also popped from `minStack`. The `getMin()` method returns the top of `minStack`, providing the minimum element in O(1) time.

- **Time Complexity**:
- Push: O(1)
- Pop: O(1)
- getMin: O(1)
- **Space Complexity**: O(n), for storing elements in both stacks.

15. Design a Circular Queue with a fixed size, supporting enqueue, dequeue, and isFull/isEmpty operations.

Test Case 1:

Input: Size = 4, Enqueue 1, 2, 3, 4, isFull()

Output: True

Test Case 2:

Input: Size = 3, Enqueue 5, 6, Dequeue, Enqueue 7, isEmpty()

Output: False

```java
public class FixedCircularQueue {

    private int[] queue;

    private int front;

    private int rear;

    private int maxSize;

    private int currentSize;



    // Constructor to initialize the circular queue

    public FixedCircularQueue(int size) {

        maxSize = size;

        queue = new int[maxSize];

        front = 0;

        rear = -1;

        currentSize = 0;

    }
```

```
// Enqueue operation

public void enqueue(int value) {

    if (isFull()) {

        System.out.println("Queue is Full");

    } else {

        rear = (rear + 1) % maxSize;

        queue[rear] = value;

        currentSize++;

    }

}



// Dequeue operation

public int dequeue() {

    if (isEmpty()) {

        System.out.println("Queue is Empty");

        return -1;

    } else {

        int value = queue[front];

        front = (front + 1) % maxSize;

        currentSize--;
```

```java
            return value;
        }
    }

    // Check if the queue is full
    public boolean isFull() {
        return currentSize == maxSize;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return currentSize == 0;
    }

    // Display the queue
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is Empty");
        } else {
            System.out.print("Queue = [");
            for (int i = 0; i < currentSize; i++) {

                System.out.print(queue[(front + i) % maxSize]);

                if (i < currentSize - 1) {

                    System.out.print(", ");

                }

            }

            System.out.println("]");

        }

    }



    public static void main(String[] args) {

        // Test Case 1

        FixedCircularQueue queue1 = new FixedCircularQueue(4);

        queue1.enqueue(1);

        queue1.enqueue(2);

        queue1.enqueue(3);

        queue1.enqueue(4);

        System.out.println("isFull(): " + queue1.isFull()); // Output: true



        // Test Case 2
```

```
        FixedCircularQueue queue2 = new FixedCircularQueue(3);

        queue2.enqueue(5);

        queue2.enqueue(6);

        queue2.dequeue();

        queue2.enqueue(7);

        System.out.println("isEmpty(): " + queue2.isEmpty()); // Output: false

    }

}
```

Flowchart

1. **Start**
2. **Initialize circular queue with fixed size**
3. **Enqueue operation:**
    a. If not full, insert element at `rear`, update `rear` and `currentSize`
4. **Dequeue operation:**
    a. If not empty, remove element from `front`, update `front` and `currentSize`
5. **isFull operation:**
    a. Return true if `currentSize == maxSize`
6. **isEmpty operation:**
    a. Return true if `currentSize == 0`
7. **End**

Step 3: Explanation

This program implements a fixed-size circular queue that supports enqueue, dequeue, `isFull`, and `isEmpty` operations. The queue uses a circular array and maintains the `front`, `rear`, and `currentSize` variables to manage its state. The enqueue method adds elements if the queue is not full, and the dequeue method removes elements if the queue is not empty. The `isFull` and `isEmpty` methods check the queue's state.

Time and Space Complexity

- **Time Complexity**:
  - Enqueue: O(1)
  - Dequeue: O(1)
  - isFull: O(1)
  - isEmpty: O(1)
- **Space Complexity**: O(n), where n is the fixed size of the queue.