**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**

**Program code:**

**package assignment;**

**class Stack {**

**private int maxSize;**

**private int[] stackArray;**

**private int top;**

**public Stack(int size) {**

**maxSize = size;**

**stackArray = new int[maxSize];**

**top = -1;**

**}**

**public void push(int value) {**

**if (top == maxSize - 1) {**

**System.*out*.println("Stack Overflow");**

**} else {**

**stackArray[++top] = value;**

**}**

**}**

**public int pop() {**

**if (top == -1) {**

**System.*out*.println("Stack Underflow");**

**return -1;**

**} else {**

**return stackArray[top--];**

**}**

**}**

**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(stackArray[i]);**

**if (i != top) System.*out*.print(", ");**

**}**

**System.*out*.println("]");**

**}**

**}**

**public static void main(String[] args) {**

**Stack stack = new Stack(5);**

**stack.push(5);**

**stack.push(3);**

**stack.push(7);**

**stack.display();**

**System.*out*.println("Popped element = " + stack.pop());**

**stack.display();**

**stack.push(10);**

**stack.push(20);**

**stack.display();**

**System.*out*.println("Popped element = " + stack.pop());**

**stack.push(15);**

**stack.display();**

**}**

**}**

**Output:**

**Stack = [5, 3, 7]**

**Popped element = 7**

**Stack = [5, 3]**

**Stack = [5, 3, 10, 20]**

**Popped element = 20**

**Stack = [5, 3, 10, 15]**

**3. Explanation**

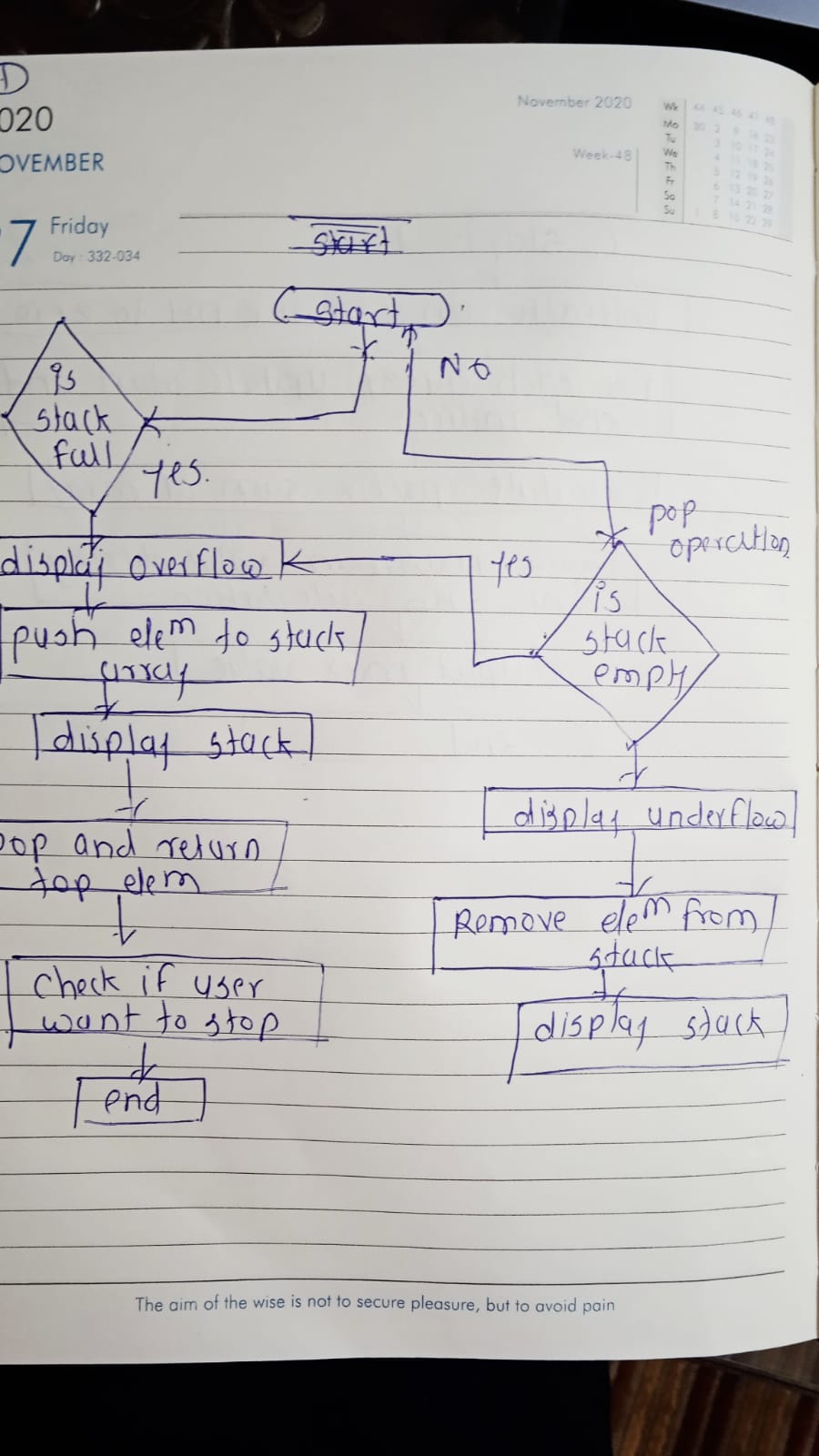
* **Stack is a Last In First Out (LIFO) data structure, where the element that is pushed last is popped first.**
* **In this implementation, we use an integer array to simulate the stack.**
* **The top variable keeps track of the index of the top element in the stack.**
* **Operations:**
  + **Push: Adds an element to the top of the stack. If the stack is full (i.e., the top index is at the maximum size), it throws a stack overflow error.**
  + **Pop: Removes the top element of the stack. If the stack is empty (i.e., top == -1), it throws a stack underflow error.**
  + **Display: Prints the elements currently in the stack.**

** Time Complexity:**

* **Push: O(1) — Direct access to the next index in the array.**
* **Pop: O(1) — Direct access to the top index.**
* **Display: O(n), where n is the number of elements in the stack (for iterating through the array).**

** Space Complexity: O(n),**

**Flowchart:**



**2. Check for balanced parentheses using a stack.**

* **Test Case 1:  
  Input: "({[()]})"  
  Output: Balanced**
* **Test Case 2:  
  Input: "([)]"  
  Output: Not Balanced**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**public class BalancedParentheses {**

**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 top = stack.pop();**

**if ((ch == ')' && top != '(') ||**

**(ch == '}' && top != '{') ||**

**(ch == ']' && top != '[')) {**

**return false;**

**}**

**}**

**}**

**return stack.isEmpty();**

**}**

**public static void main(String[] args) {**

**String expression1 = "({[()]})";**

**if (*isBalanced*(expression1)) {**

**System.*out*.println("Balanced");**

**} else {**

**System.*out*.println("Not Balanced");**

**}**

**String expression2 = "([)]";**

**if (*isBalanced*(expression2)) {**

**System.*out*.println("Balanced");**

**} else {**

**System.*out*.println("Not Balanced");**

**}**

**}**

**}**

**Output:**

**Balanced**

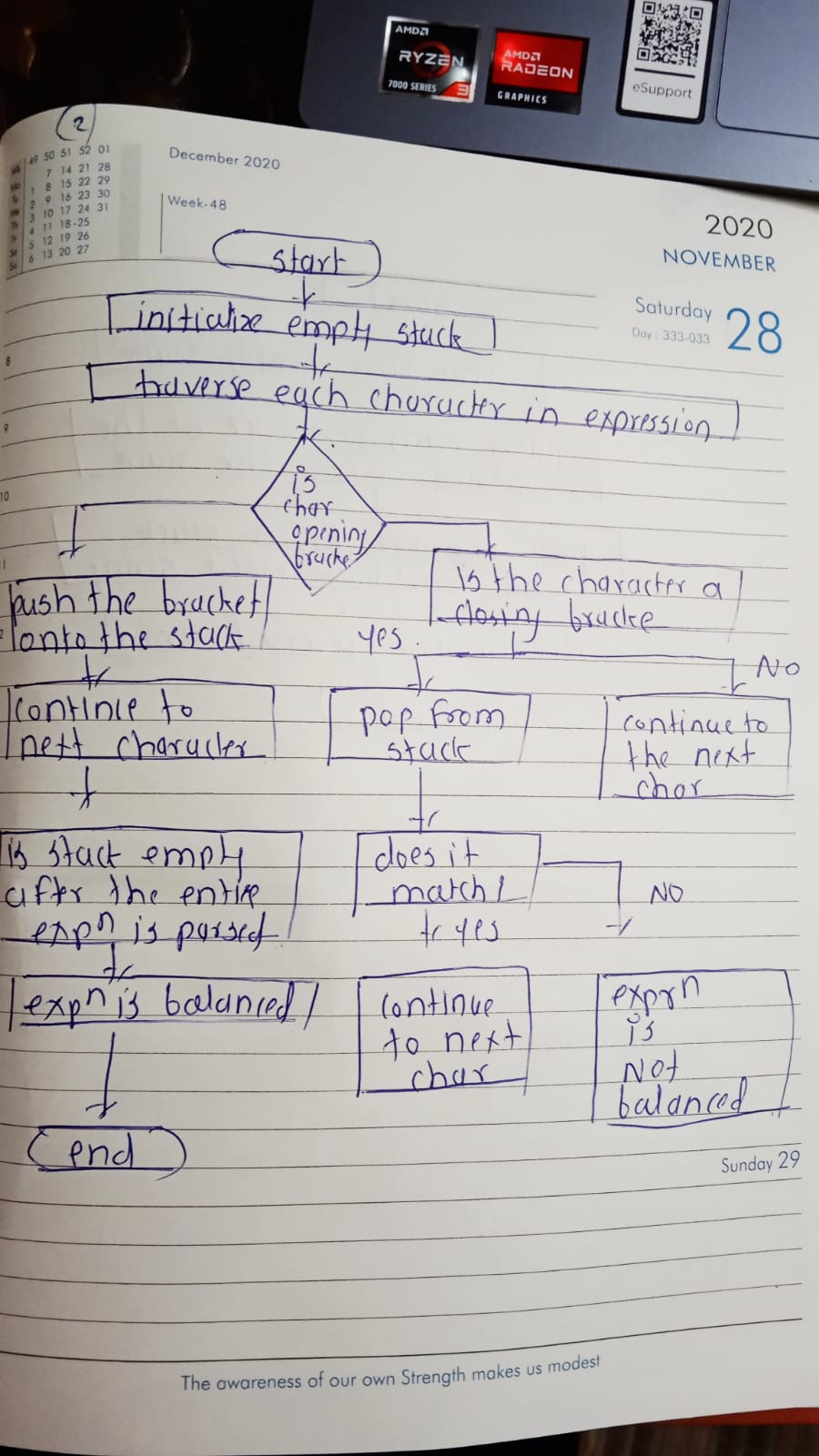
**Not Balanced**

**3. Explanation**

* **Stack: A stack is used to track opening brackets. When an opening bracket ((, {, [) is encountered, it is pushed onto the stack. When a closing bracket is encountered, we check if it matches the top of the stack:**
  + **If it matches, we pop the opening bracket off the stack.**
  + **If it doesn’t match or the stack is empty when encountering a closing bracket, the expression is not balanced.**
* **At the end, if the stack is empty, the expression is balanced; otherwise, it is not balanced.**

**5. Time and Space Complexity:O(n)**

**Flowchart:**



**3. Reverse a string using a stack.**

* **Test Case 1:  
  Input: "hello"  
  Output: "olleh"**
* **Test Case 2:  
  Input: "world"  
  Output: "dlrow"**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**public class ReverseString {**

**public static String reverse(String input) {**

**Stack<Character> stack = new Stack<>();**

**for (char ch : input.toCharArray()) {**

**stack.push(ch);**

**}**

**StringBuilder reversed = new StringBuilder();**

**while (!stack.isEmpty()) {**

**reversed.append(stack.pop());**

**}**

**return reversed.toString();**

**}**

**public static void main(String[] args) {**

**String input1 = "hello";**

**System.*out*.println("Reversed string: " + *reverse*(input1));**

**String input2 = "world";**

**System.*out*.println("Reversed string: " + *reverse*(input2));**

**}**

**}**

**Output code:**

**Reversed string: olleh**

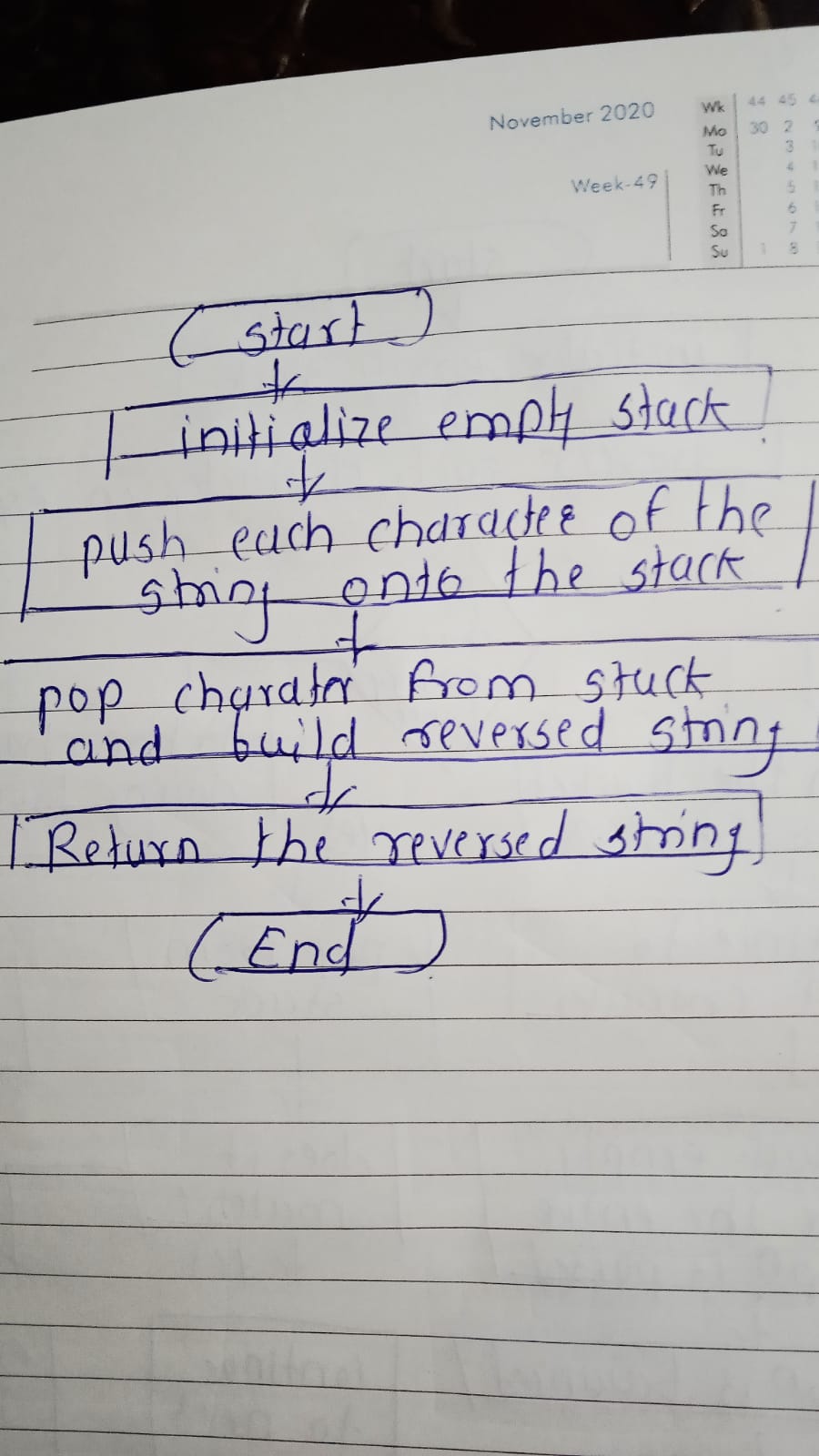
**Reversed string: dlrow**

**3. Explanation**

* **Stack: A stack is used to reverse the string.**
  + **The string's characters are pushed onto the stack one by one.**
  + **When popped from the stack, the characters are retrieved in reverse order.**
* **How it works:**
  + **Each character of the string is pushed onto the stack.**
  + **Once all characters are on the stack, they are popped off and appended to form the reversed string.**

**Time and Space Complexity: O(n)**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**public class PostfixEvaluation {**

**public static int evaluatePostfix(String expression) {**

**Stack<Integer> stack = new Stack<>();**

**String[] tokens = expression.split(" ");**

**for (String token : tokens) {**

**if (*isNumeric*(token)) {**

**stack.push(Integer.*parseInt*(token));**

**}**

**else {**

**int operand2 = stack.pop();**

**int operand1 = stack.pop();**

**int result = 0;**

**switch (token) {**

**case "+":**

**result = operand1 + operand2;**

**break;**

**case "-":**

**result = operand1 - operand2;**

**break;**

**case "\*":**

**result = operand1 \* operand2;**

**break;**

**case "/":**

**result = operand1 / operand2;**

**break;**

**}**

**stack.push(result);**

**}**

**}**

**return stack.pop();**

**}**

**private static boolean isNumeric(String str) {**

**try {**

**Integer.*parseInt*(str);**

**return true;**

**} catch (NumberFormatException e) {**

**return false;**

**}**

**}**

**public static void main(String[] args) {**

**String expression1 = "5 3 + 2 \*";**

**System.*out*.println("Output: " + *evaluatePostfix*(expression1)); // Expected output: 16**

**String expression2 = "4 5 \* 6 /";**

**System.*out*.println("Output: " + *evaluatePostfix*(expression2)); // Expected output: 3**

**}**

**}**

**Output:**

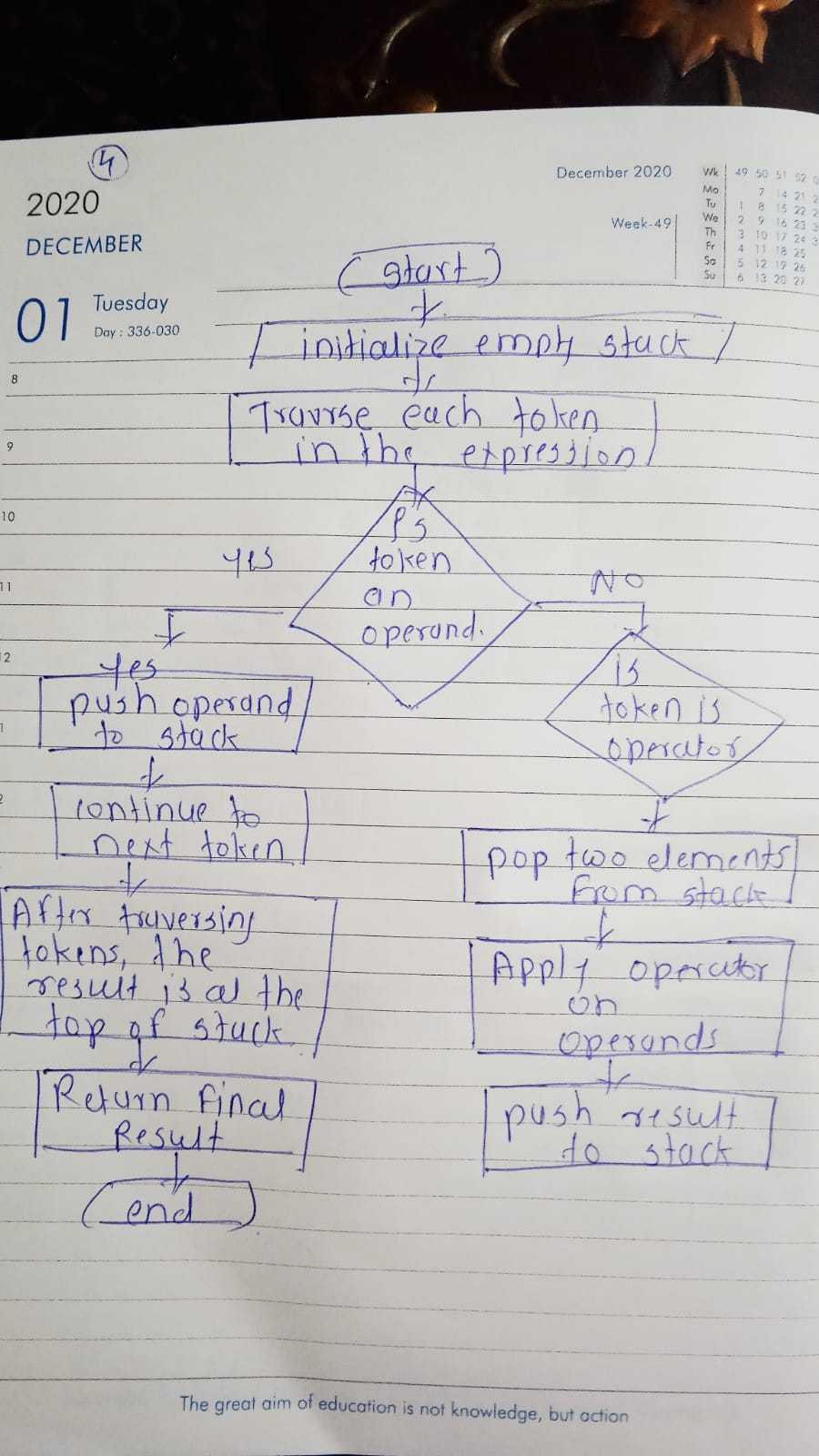
**Output: 16**

**Output: 3**

**3. Explanation**

* **Postfix Expression: A postfix expression (also known as Reverse Polish Notation) places the operator after the operands. For example, "5 3 +" means "5 + 3".**
* **How it works:**
  + **Traverse through the postfix expression token by token.**
  + **If the token is a number (operand), push it onto the stack.**
  + **If the token is an operator (+, -, \*, /), pop two numbers from the stack, apply the operator, and push the result back onto the stack.**
  + **At the end of the traversal, the stack will contain only one element, which is the result of the expression.**
* **Time and Space Complexity: O(n)**

**Flowchart:**



**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 / +"**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**public class InfixToPostfix {**

**private static int precedence(char ch) {**

**switch (ch) {**

**case '+':**

**case '-':**

**return 1;**

**case '\*':**

**case '/':**

**return 2;**

**case '^':**

**return 3;**

**}**

**return -1;**

**}**

**public static String infixToPostfix(String expression) {**

**Stack<Character> stack = new Stack<>();**

**StringBuilder result = new StringBuilder();**

**for (int i = 0; i < expression.length(); i++) {**

**char ch = expression.charAt(i);**

**if (Character.*isLetterOrDigit*(ch)) {**

**result.append(ch).append(" ");**

**}**

**else if (ch == '(') {**

**stack.push(ch);**

**}**

**else if (ch == ')') {**

**while (!stack.isEmpty() && stack.peek() != '(') {**

**result.append(stack.pop()).append(" ");**

**}**

**stack.pop();**

**}**

**else {**

**while (!stack.isEmpty() && *precedence*(ch) <= *precedence*(stack.peek())) {**

**result.append(stack.pop()).append(" ");**

**}**

**stack.push(ch);**

**}**

**}**

**while (!stack.isEmpty()) {**

**result.append(stack.pop()).append(" ");**

**}**

**return result.toString().trim();**

**}**

**public static void main(String[] args) {**

**String expression1 = "A + B \* C";**

**System.*out*.println("Postfix: " + *infixToPostfix*(expression1));**

**String expression2 = "A \* B + C / D";**

**System.*out*.println("Postfix: " + *infixToPostfix*(expression2));**

**}**

**}**

**Output:**

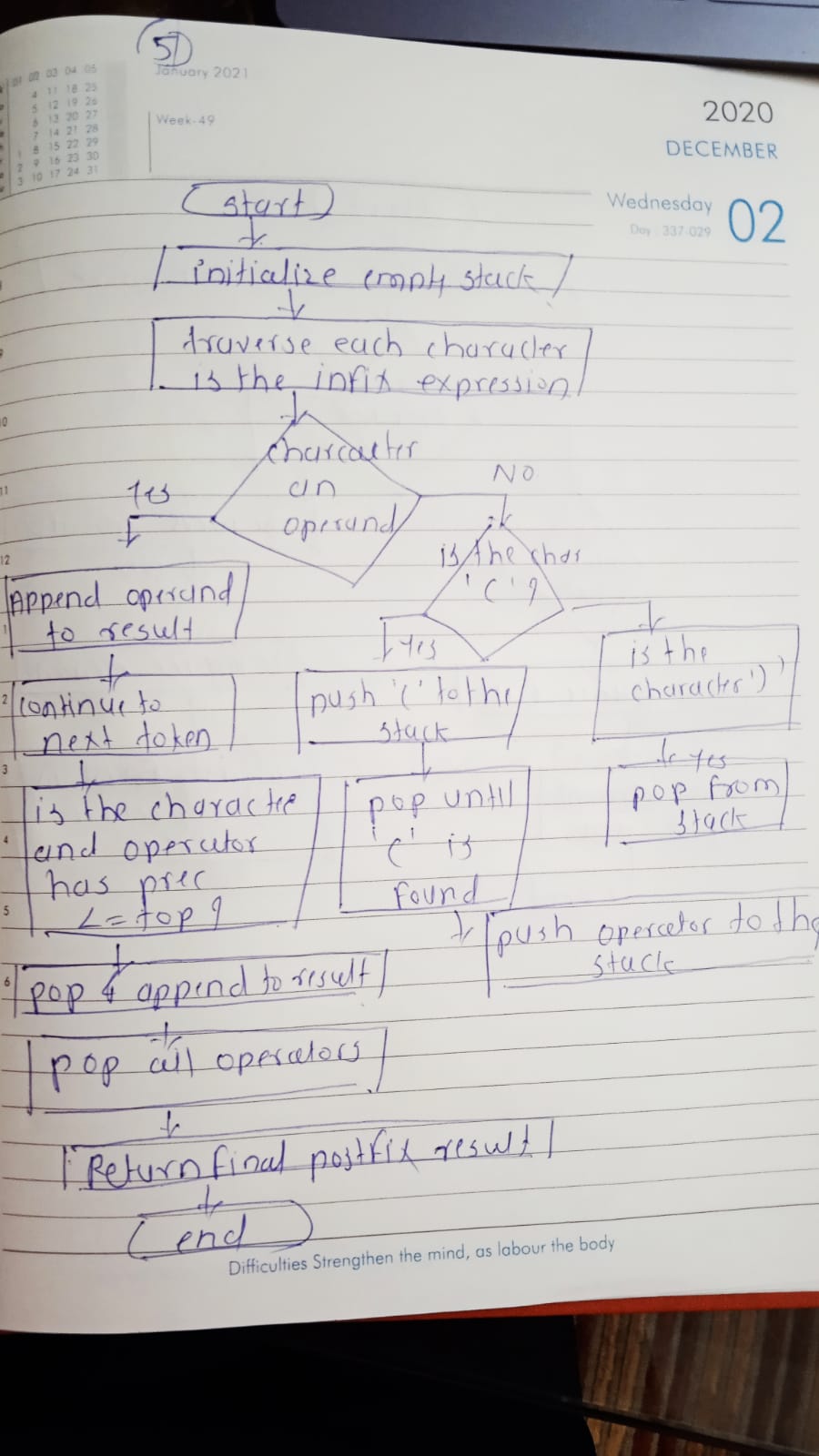
**Postfix: A + B \* C**

**Postfix: A \* B + C / D**

**3. Explanation**

* **Infix Expression: Operators are written between operands (e.g., A + B).**
* **Postfix Expression: Operators follow the operands (e.g., A B +).**
* **Stack:**
  + **Used to temporarily hold operators until they can be added to the postfix expression.**
  + **When an operator is encountered in the infix expression, operators from the stack are popped and added to the result until an operator with lower precedence is found.**
* **Time and Space Complexity: O(n)**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**public class ArrayQueue {**

**private int front, rear, size;**

**private int capacity;**

**private int[] queue;**

**public ArrayQueue(int capacity) {**

**this.capacity = capacity;**

**queue = new int[capacity];**

**front = this.size = 0;**

**rear = capacity - 1;**

**}**

**public boolean isFull() {**

**return size == capacity;**

**}**

**public boolean isEmpty() {**

**return size == 0;**

**}**

**public void enqueue(int item) {**

**if (isFull()) {**

**System.*out*.println("Queue is full");**

**return;**

**}**

**rear = (rear + 1) % capacity;**

**queue[rear] = item;**

**size++;**

**System.*out*.println(item + " enqueued to the queue");**

**}**

**public int dequeue() {**

**if (isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return Integer.*MIN\_VALUE*;**

**}**

**int item = queue[front];**

**front = (front + 1) % capacity;**

**size--;**

**return item;**

**}**

**public void display() {**

**if (isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return;**

**}**

**System.*out*.print("Queue = ");**

**for (int i = 0; i < size; i++) {**

**System.*out*.print(queue[(front + i) % capacity] + " ");**

**}**

**System.*out*.println();**

**}**

**public static void main(String[] args) {**

**ArrayQueue queue = new ArrayQueue(5);**

**queue.enqueue(5);**

**queue.enqueue(10);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.display();**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.display();**

**}**

**}**

**Output:**

**5 enqueued to the queue**

**10 enqueued to the queue**

**Dequeued element = 5**

**Queue = 10**

**1 enqueued to the queue**

**2 enqueued to the queue**

**3 enqueued to the queue**

**Dequeued element = 10**

**Dequeued element = 1**

**Queue = 2 3**

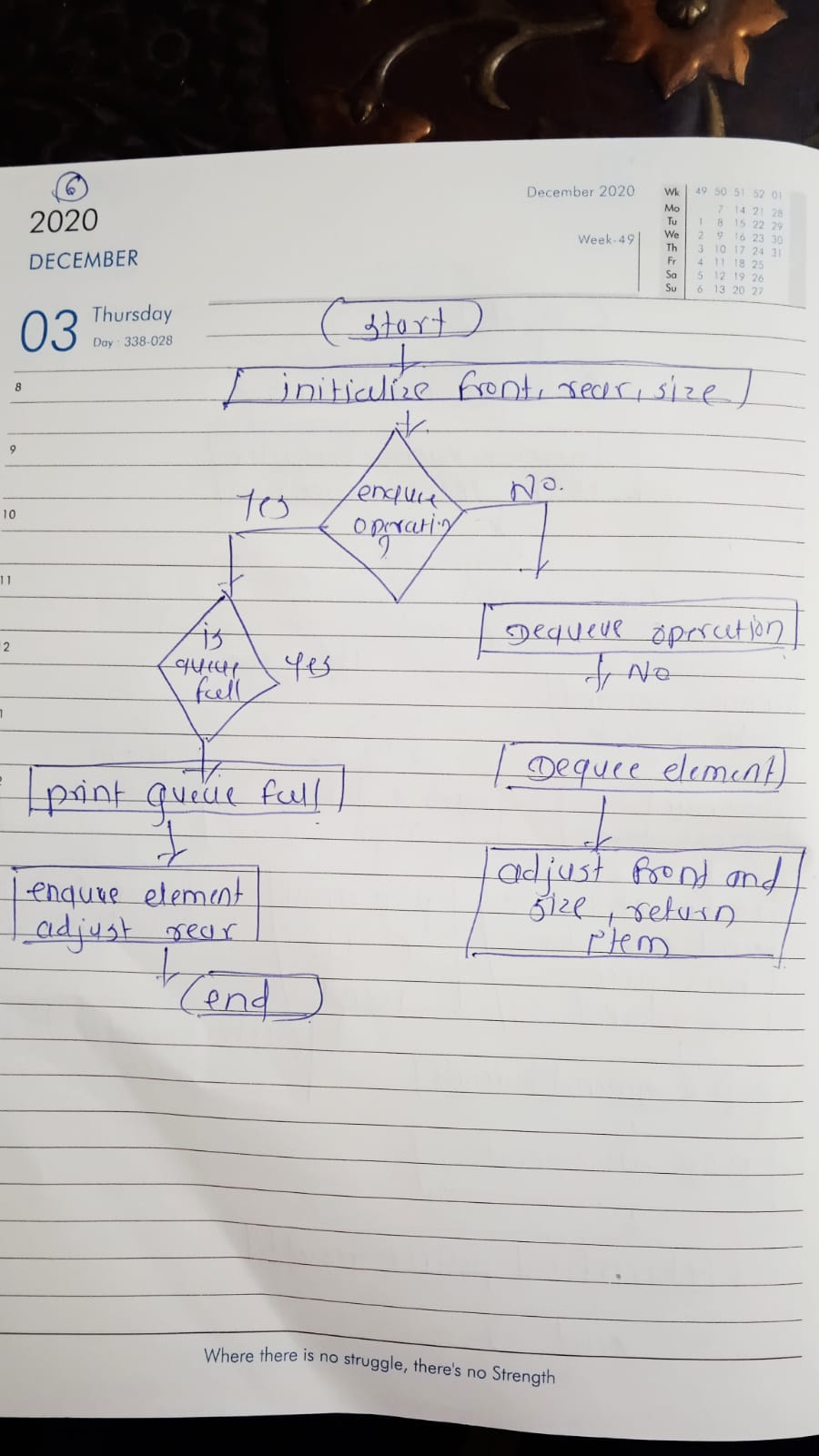
**3. Explanation**

* **Queue: A queue is a linear data structure following the FIFO (First In, First Out) principle.**
* **Array-based Queue Implementation:**
  + **Enqueue: Adds an element to the rear of the queue.**
  + **Dequeue: Removes an element from the front of the queue.**
  + **The queue uses modular arithmetic to wrap around the array when it reaches the end, making it a circular queue.**

**5. Time and Space Complexity**

* **Time Complexity:**
  + **Enqueue/Dequeue operations: O(1) as they involve simple array operations.**
* **Space Complexity: O(n),**

**Flowchart:**



**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]**

**Program code:**

**package assignment;**

**public class CircularQueue {**

**private int front, rear, size;**

**private int capacity;**

**private int[] queue;**

**public CircularQueue(int capacity) {**

**this.capacity = capacity;**

**queue = new int[capacity];**

**front = this.size = 0;**

**rear = capacity - 1;**

**}**

**public boolean isFull() {**

**return size == capacity;**

**}**

**public boolean isEmpty() {**

**return size == 0;**

**}**

**public void enqueue(int item) {**

**if (isFull()) {**

**System.*out*.println("Queue is full");**

**return;**

**}**

**rear = (rear + 1) % capacity;**

**queue[rear] = item;**

**size++;**

**System.*out*.println(item + " enqueued to the queue");**

**}**

**public int dequeue() {**

**if (isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return Integer.*MIN\_VALUE*;**

**}**

**int item = queue[front];**

**front = (front + 1) % capacity;**

**size--;**

**return item;**

**}**

**public void display() {**

**if (isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return;**

**}**

**System.*out*.print("Queue = ");**

**for (int i = 0; i < size; i++) {**

**System.*out*.print(queue[(front + i) % capacity] + " ");**

**}**

**System.*out*.println();**

**}**

**public static void main(String[] args) {**

**CircularQueue queue = new CircularQueue(5);**

**queue.enqueue(4);**

**queue.enqueue(5);**

**queue.enqueue(6);**

**queue.enqueue(7);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.enqueue(8);**

**queue.display();**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**queue.enqueue(4);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.enqueue(5);**

**queue.display();**

**}**

**}**

**Output:**

**4 enqueued to the queue**

**5 enqueued to the queue**

**6 enqueued to the queue**

**7 enqueued to the queue**

**Dequeued element = 4**

**8 enqueued to the queue**

**Queue = 5 6 7 8**

**1 enqueued to the queue**

**Queue is full**

**Queue is full**

**Queue is full**

**Dequeued element = 5**

**Dequeued element = 6**

**5 enqueued to the queue**

**Queue = 7 8 1 5**

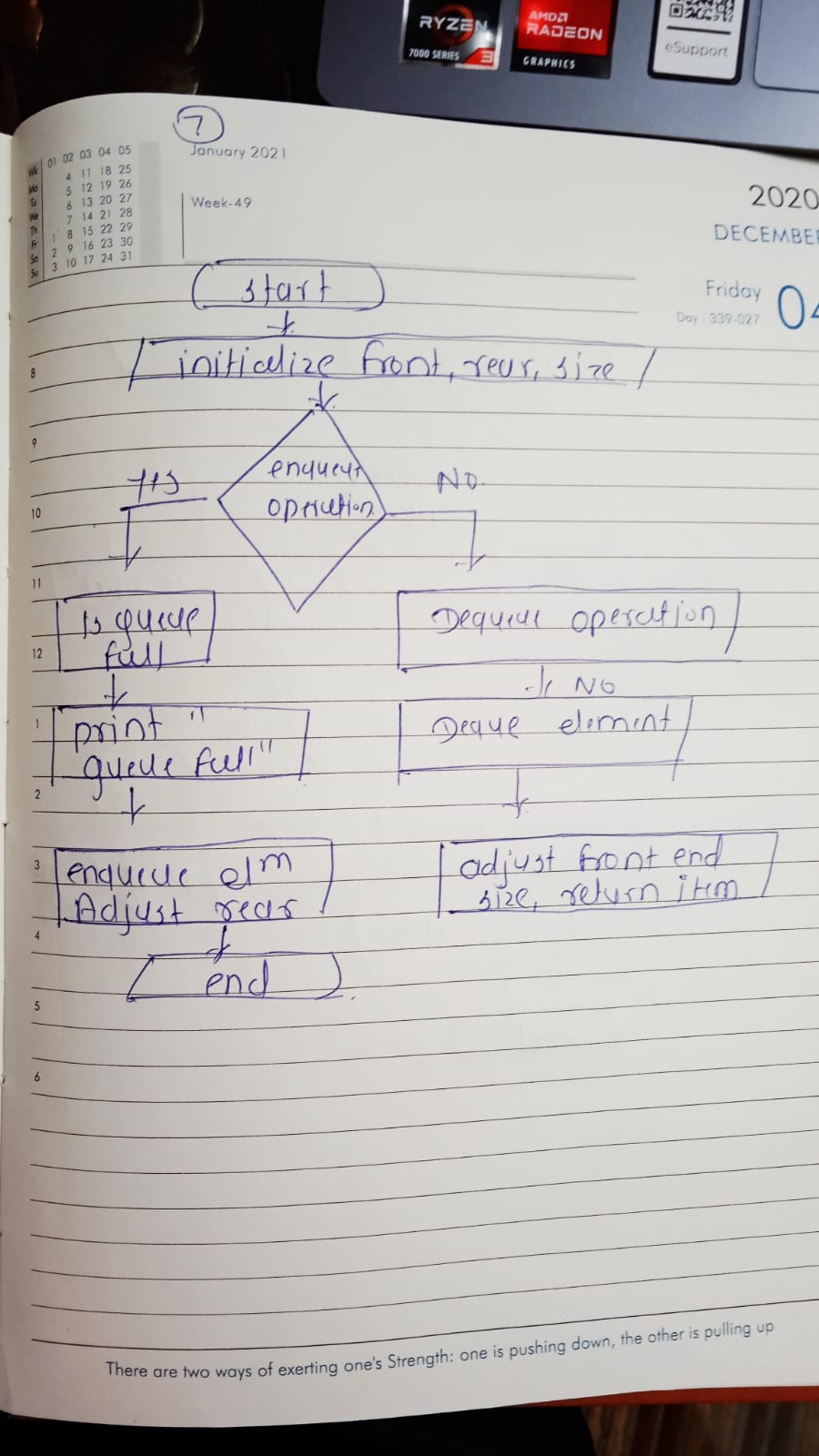
**3. Explanation**

* **Circular Queue: A circular queue is a linear data structure that follows the FIFO (First In, First Out) principle, but the last position of the array is connected to the first position, making it circular.**
  + **Enqueue: Adds an element to the rear of the queue.**
  + **Dequeue: Removes an element from the front of the queue.**
  + **Circular: When the rear reaches the end of the array, it wraps around to the beginning.**

**5. Time and Space Complexity**

* **Time Complexity:**
  + **: O(1)**
* **Space Complexity: O(n),**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**public class QueueUsingTwoStacks {**

**private Stack<Integer> stack1;**

**private Stack<Integer> stack2;**

**public QueueUsingTwoStacks() {**

**stack1 = new Stack<>();**

**stack2 = new Stack<>();**

**}**

**public void enqueue(int item) {**

**stack1.push(item);**

**System.*out*.println(item + " enqueued to the queue");**

**}**

**public int dequeue() {**

**if (stack2.isEmpty()) {**

**if (stack1.isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return Integer.*MIN\_VALUE*;**

**}**

**while (!stack1.isEmpty()) {**

**stack2.push(stack1.pop());**

**}**

**}**

**return stack2.pop();**

**}**

**public void display() {**

**if (stack1.isEmpty() && stack2.isEmpty()) {**

**System.*out*.println("Queue is empty");**

**return;**

**}**

**Stack<Integer> tempStack = new Stack<>();**

**while (!stack2.isEmpty()) {**

**tempStack.push(stack2.pop());**

**}**

**while (!tempStack.isEmpty()) {**

**System.*out*.print(tempStack.pop() + " ");**

**}**

**System.*out*.print("\n");**

**}**

**public static void main(String[] args) {**

**QueueUsingTwoStacks queue = new QueueUsingTwoStacks();**

**queue.enqueue(3);**

**queue.enqueue(7);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.display();**

**queue.enqueue(10);**

**queue.enqueue(20);**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**System.*out*.println("Dequeued element = " + queue.dequeue());**

**queue.display();**

**}**

**}**

**Output:**

**3 enqueued to the queue**

**7 enqueued to the queue**

**Dequeued element = 3**

**7**

**10 enqueued to the queue**

**20 enqueued to the queue**

**Dequeued element = 10**

**Dequeued element = 20**

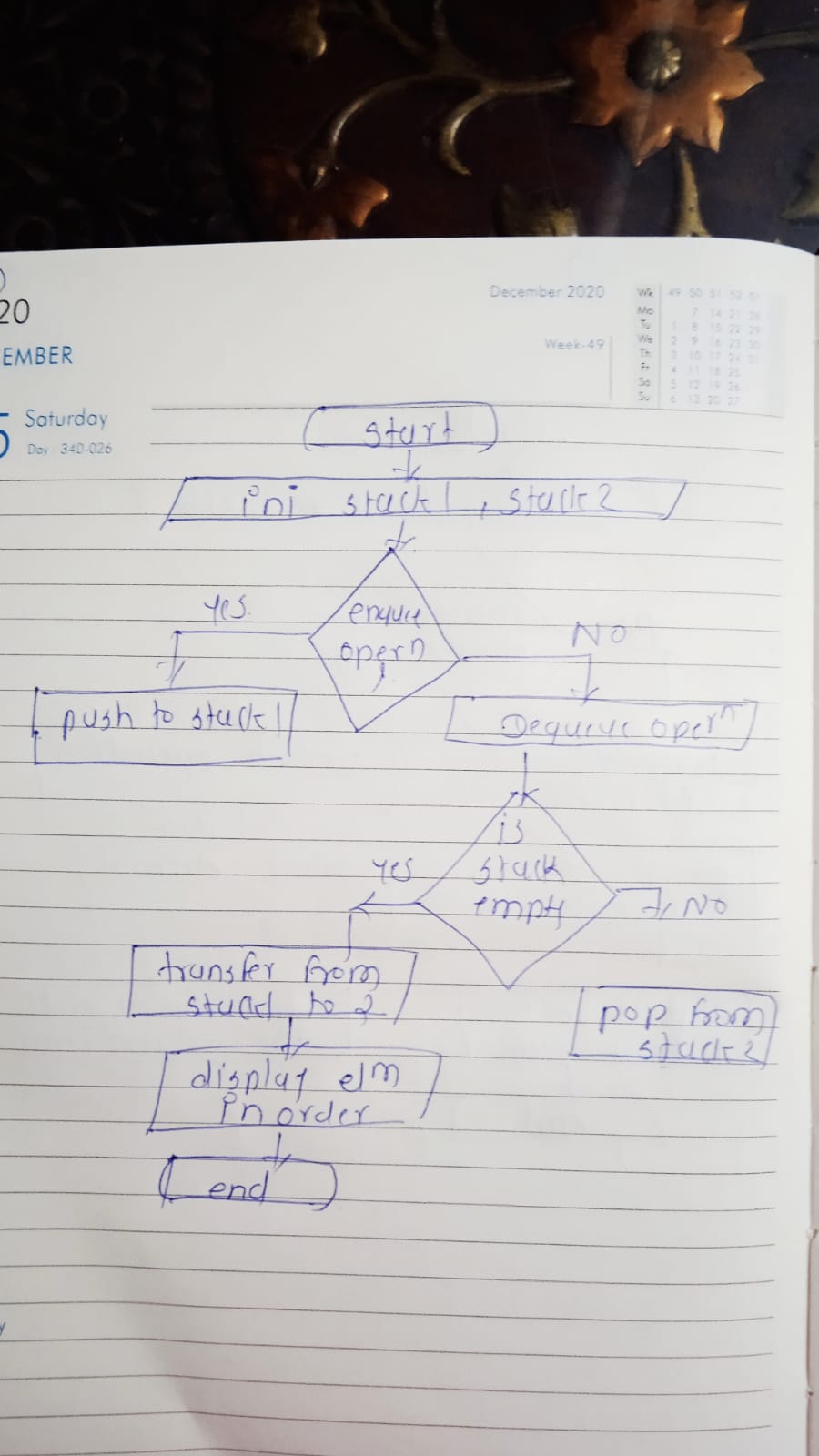
**Queue is empty**

**3. Explanation**

* **Queue Using Two Stacks: The queue is implemented using two stacks.**
  + **Enqueue Operation: Simply push the element onto stack1.**
  + **Dequeue Operation: If stack2 is empty, pop all elements from stack1 and push them onto stack2. Then pop from stack2 to get the front of the queue.**
  + **Displaying the Queue: Use a temporary stack to display elements in the correct order.**

**5. Time and Space Complexity**

* **Time Complexity:**
  + **: O(1)**
* **Space Complexity: O(n),**



**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**

**Program code:**

**package assignment;**

**import java.util.Arrays;**

**public class MinHeap {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**public MinHeap(int capacity) {**

**this.capacity = capacity;**

**this.size = 0;**

**heap = new int[capacity];**

**}**

**public void insert(int key) {**

**if (size == capacity) {**

**System.*out*.println("Heap is full");**

**return;**

**}**

**heap[size] = key;**

**size++;**

**minHeapifyUp(size - 1);**

**System.*out*.println(key + " inserted into the Min-Heap");**

**}**

**public int extractMin() {**

**if (size == 0) {**

**System.*out*.println("Heap is empty");**

**return Integer.*MIN\_VALUE*;**

**}**

**int min = heap[0];**

**heap[0] = heap[size - 1];**

**size--;**

**minHeapifyDown(0);**

**return min;**

**}**

**private void minHeapifyUp(int index) {**

**while (index != 0 && heap[parent(index)] > heap[index]) {**

**swap(index, parent(index));**

**index = parent(index);**

**}**

**}**

**private void minHeapifyDown(int index) {**

**int smallest = index;**

**int leftChild = leftChild(index);**

**int rightChild = rightChild(index);**

**if (leftChild < size && heap[leftChild] < heap[smallest]) {**

**smallest = leftChild;**

**}**

**if (rightChild < size && heap[rightChild] < heap[smallest]) {**

**smallest = rightChild;**

**}**

**if (smallest != index) {**

**swap(index, smallest);**

**minHeapifyDown(smallest);**

**}**

**}**

**private void swap(int i, int j) {**

**int temp = heap[i];**

**heap[i] = heap[j];**

**heap[j] = temp;**

**}**

**private int parent(int index) {**

**return (index - 1) / 2;**

**}**

**private int leftChild(int index) {**

**return 2 \* index + 1;**

**}**

**private int rightChild(int index) {**

**return 2 \* index + 2;**

**}**

**public void display() {**

**System.*out*.println("Min-Heap = " + Arrays.*toString*(Arrays.*copyOf*(heap, size)));**

**}**

**public static void main(String[] args) {**

**MinHeap minHeap = new MinHeap(10);**

**minHeap.insert(10);**

**minHeap.insert(15);**

**minHeap.insert(20);**

**minHeap.insert(17);**

**System.*out*.println("Extracted Min = " + minHeap.extractMin());**

**minHeap.display();**

**minHeap.insert(30);**

**minHeap.insert(40);**

**minHeap.insert(20);**

**minHeap.insert(50);**

**System.*out*.println("Extracted Min = " + minHeap.extractMin());**

**minHeap.display();**

**}**

**}**

**Output:**

**10 inserted into the Min-Heap**

**15 inserted into the Min-Heap**

**20 inserted into the Min-Heap**

**17 inserted into the Min-Heap**

**Extracted Min = 10**

**Min-Heap = [15, 17, 20]**

**30 inserted into the Min-Heap**

**40 inserted into the Min-Heap**

**20 inserted into the Min-Heap**

**50 inserted into the Min-Heap**

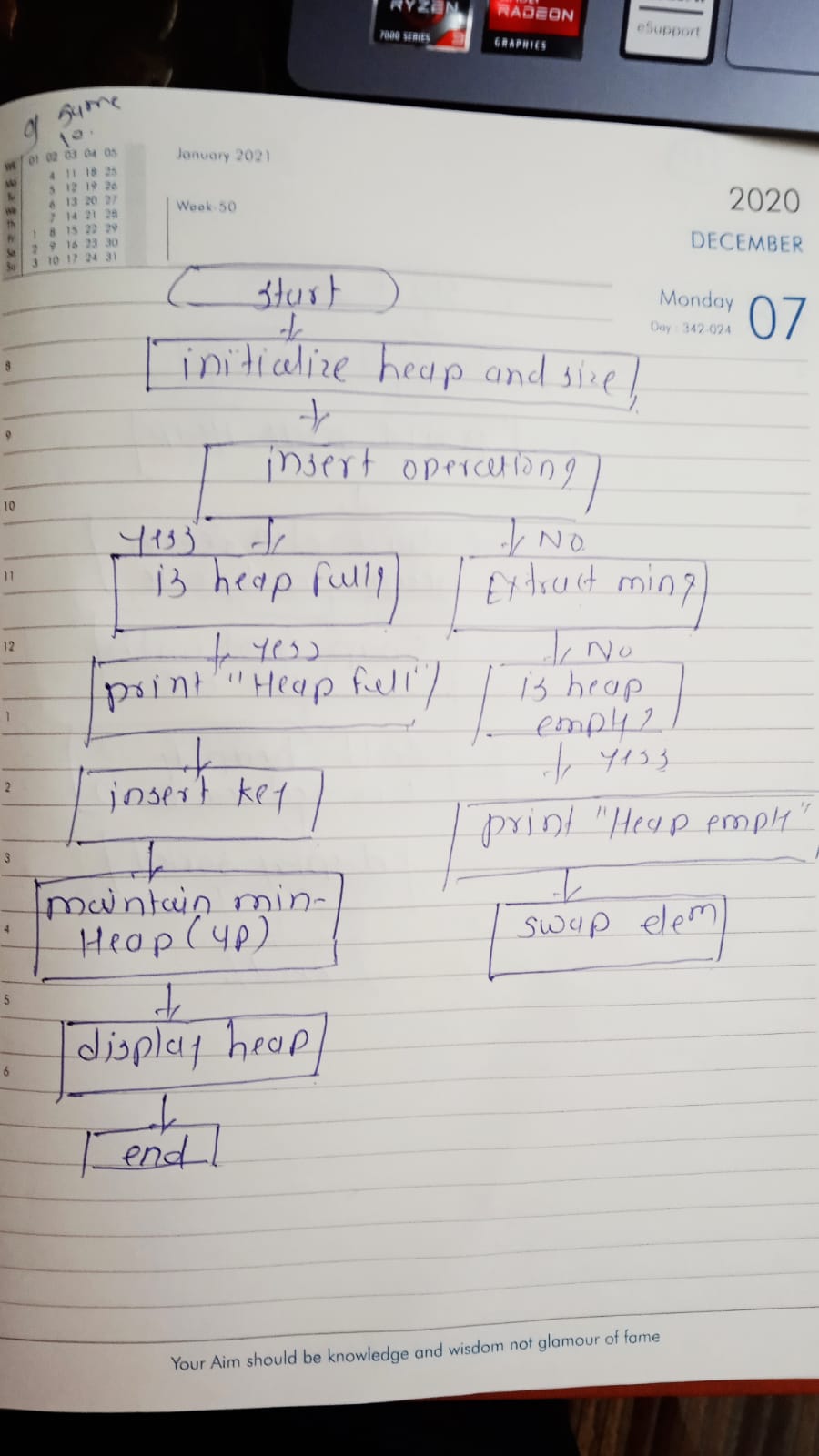
**Extracted Min = 15**

**Min-Heap = [17, 30, 20, 50, 40, 20]**

**3. Explanation**

* **Min-Heap: A Min-Heap is a complete binary tree where the value of each node is less than or equal to the values of its children. The minimum element is at the root.**
  + **Insert Operation: When inserting, the element is added to the end of the heap (the next available position), and the heap property is maintained by comparing the new element with its parent and swapping them if necessary.**
  + **Extract Min Operation: The minimum element (root) is removed, and the last element in the heap replaces the root. The heap property is then maintained by comparing the new root with its children and swapping if necessary.**

**5. Time and Space Complexity**

* **Time Complexity:**
  + **Insert operation: O(log n)**
* **Space Complexity: O(n)**
* **Flowchart:**
* 

**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**

**Program code:**

**package assignment;**

**import java.util.Arrays;**

**public class MaxHeap {**

**private int[] heap;**

**private int size;**

**private int capacity;**

**public MaxHeap(int capacity) {**

**this.capacity = capacity;**

**this.size = 0;**

**heap = new int[capacity];**

**}**

**public void insert(int key) {**

**if (size == capacity) {**

**System.*out*.println("Heap is full");**

**return;**

**}**

**heap[size] = key;**

**size++;**

**maxHeapifyUp(size - 1);**

**System.*out*.println(key + " inserted into the Max-Heap");**

**}**

**public int extractMax() {**

**if (size == 0) {**

**System.*out*.println("Heap is empty");**

**return Integer.*MIN\_VALUE*;**

**}**

**int max = heap[0];**

**heap[0] = heap[size - 1];**

**size--;**

**maxHeapifyDown(0);**

**return max;**

**}**

**private void maxHeapifyUp(int index) {**

**while (index != 0 && heap[parent(index)] < heap[index]) {**

**swap(index, parent(index));**

**index = parent(index);**

**}**

**}**

**private void maxHeapifyDown(int index) {**

**int largest = index;**

**int leftChild = leftChild(index);**

**int rightChild = rightChild(index);**

**if (leftChild < size && heap[leftChild] > heap[largest]) {**

**largest = leftChild;**

**}**

**if (rightChild < size && heap[rightChild] > heap[largest]) {**

**largest = rightChild;**

**}**

**if (largest != index) {**

**swap(index, largest);**

**maxHeapifyDown(largest);**

**}**

**}**

**private void swap(int i, int j) {**

**int temp = heap[i];**

**heap[i] = heap[j];**

**heap[j] = temp;**

**}**

**private int parent(int index) {**

**return (index - 1) / 2;**

**}**

**private int leftChild(int index) {**

**return 2 \* index + 1;**

**}**

**private int rightChild(int index) {**

**return 2 \* index + 2;**

**}**

**public void display() {**

**System.*out*.println("Max-Heap = " + Arrays.*toString*(Arrays.*copyOf*(heap, size)));**

**}**

**public static void main(String[] args) {**

**MaxHeap maxHeap = new MaxHeap(10);**

**maxHeap.insert(12);**

**maxHeap.insert(7);**

**maxHeap.insert(15);**

**maxHeap.insert(5);**

**System.*out*.println("Extracted Max = " + maxHeap.extractMax());**

**maxHeap.display();**

**maxHeap.insert(8);**

**maxHeap.insert(20);**

**maxHeap.insert(10);**

**maxHeap.insert(3);**

**System.*out*.println("Extracted Max = " + maxHeap.extractMax());**

**maxHeap.display();**

**}**

**}**

**Output:**

**12 inserted into the Max-Heap**

**7 inserted into the Max-Heap**

**15 inserted into the Max-Heap**

**5 inserted into the Max-Heap**

**Extracted Max = 15**

**Max-Heap = [12, 7, 5]**

**8 inserted into the Max-Heap**

**20 inserted into the Max-Heap**

**10 inserted into the Max-Heap**

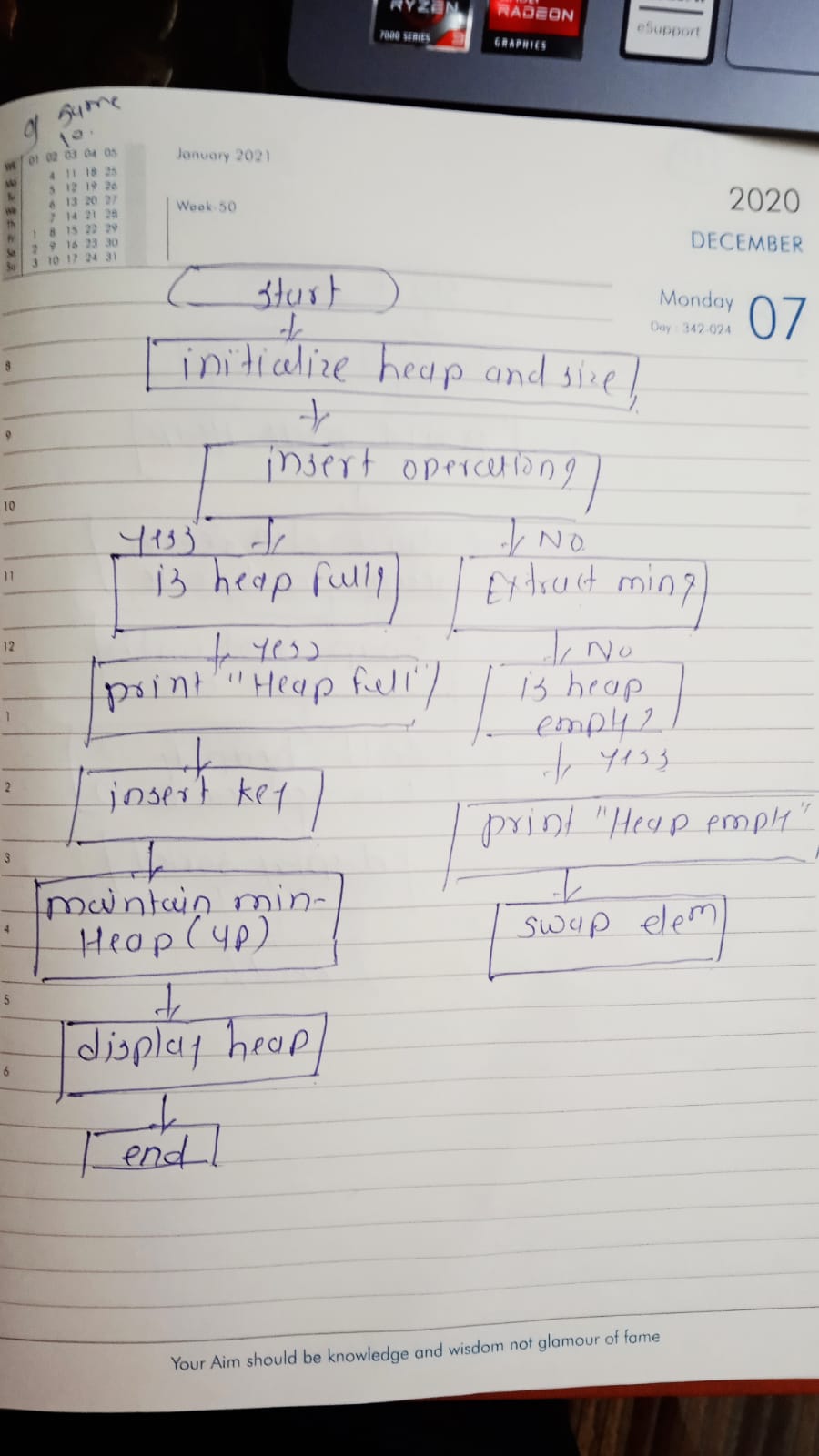
**3 inserted into the Max-Heap**

**Extracted Max = 20**

**Max-Heap = [12, 8, 10, 7, 3, 5]**

**3. Explanation**

* **Max-Heap: A Max-Heap is a complete binary tree where the value of each node is greater than or equal to the values of its children. The maximum element is at the root.**
  + **Insert Operation: When inserting, the element is added to the end of the heap (the next available position), and the heap property is maintained by comparing the new element with its parent and swapping them if necessary.**
  + **Extract Max Operation: The maximum element (root) is removed, and the last element in the heap replaces the root. The heap property is then maintained by comparing the new root with its children and swapping if necessary.**
* **Time Complexity:**
  + **Insert operation: O(log n)**
* **Space Complexity: O(n)**
* **Flowchart:**



**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]**

**Program code:**

**package assignment;**

**import java.util.Arrays;**

**public class HeapSort {**

**public static void heapSort(int[] array) {**

**int n = array.length;**

**for (int i = n / 2 - 1; i >= 0; i--) {**

***heapify*(array, n, i);**

**}**

**for (int i = n - 1; i > 0; i--) {**

***swap*(array, 0, i);**

***heapify*(array, i, 0);**

**}**

**}**

**private static void heapify(int[] array, int n, int i) {**

**int largest = i;**

**int leftChild = 2 \* i + 1;**

**int rightChild = 2 \* i + 2;**

**if (leftChild < n && array[leftChild] > array[largest]) {**

**largest = leftChild;**

**}**

**if (rightChild < n && array[rightChild] > array[largest]) {**

**largest = rightChild;**

**}**

**if (largest != i) {**

***swap*(array, i, largest);**

***heapify*(array, n, largest);**

**}**

**}**

**private static void swap(int[] array, int i, int j) {**

**int temp = array[i];**

**array[i] = array[j];**

**array[j] = temp;**

**}**

**public static void main(String[] args) {**

**int[] array1 = {5, 1, 12, 3, 9};**

***heapSort*(array1);**

**System.*out*.println("Sorted array: " + Arrays.*toString*(array1)); // Expected output: [1, 3, 5, 9, 12]**

**int[] array2 = {20, 15, 8, 10};**

***heapSort*(array2);**

**System.*out*.println("Sorted array: " + Arrays.*toString*(array2)); // Expected output: [8, 10, 15, 20]**

**}**

**}**

**Output:**

**Sorted array: [1, 3, 5, 9, 12]**

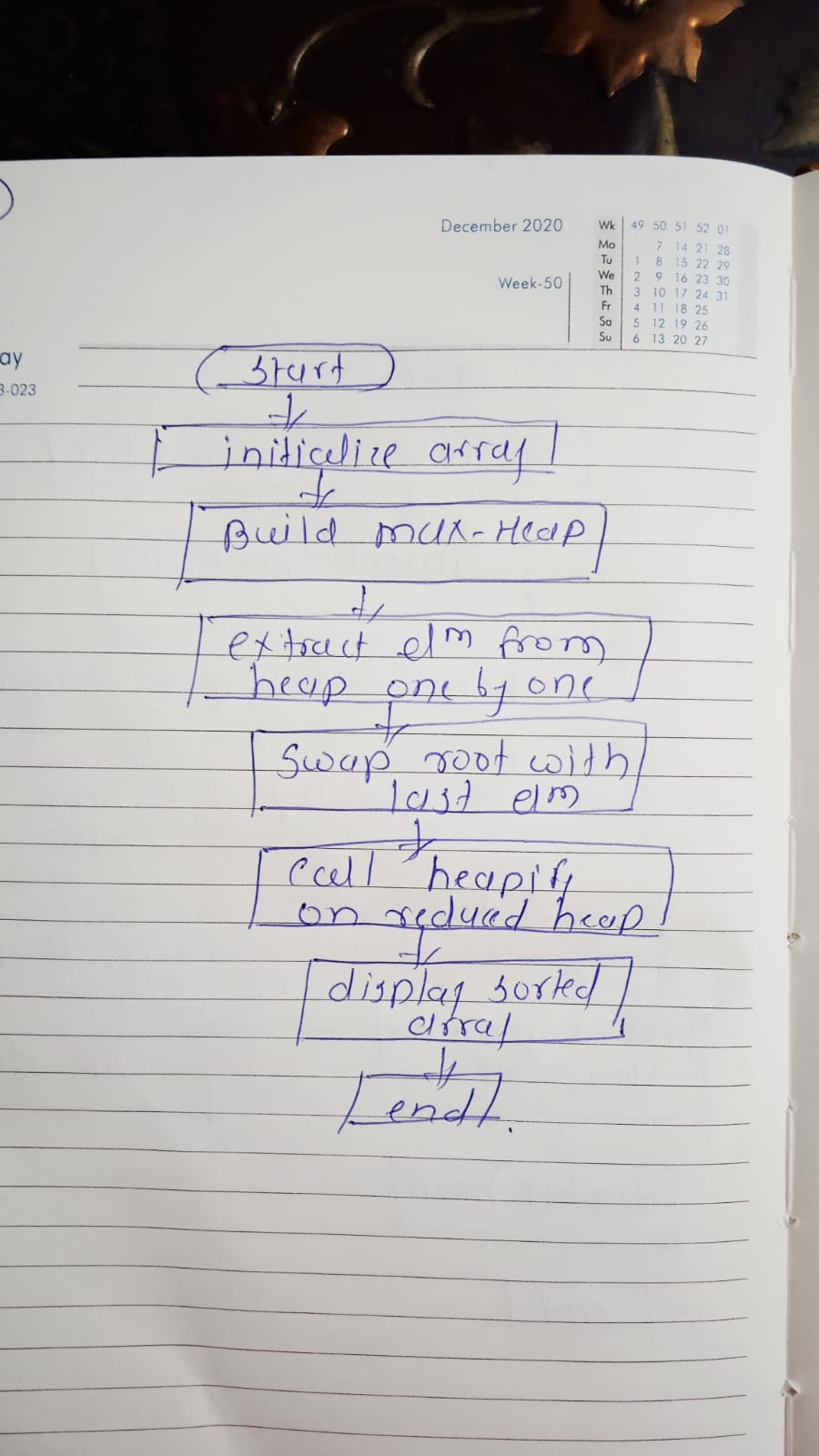
**Sorted array: [8, 10, 15, 20]**

* **Time Complexity:**
  + **Insert operation: O( n)**
* **Space Complexity: O(1)**

**3. Explanation**

* **Heap Sort: Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure. The algorithm can be divided into two main phases:**
  1. **Building a Max-Heap: The first step is to convert the array into a Max-Heap, a complete binary tree where the value of each node is greater than or equal to the values of its children. This is done by calling the heapify function from the last non-leaf node down to the root.**
  2. **Sorting: Once the Max-Heap is built, the maximum element (the root) is swapped with the last element of the heap. The size of the heap is reduced, and the heapify function is called again to maintain the heap property. This process is repeated until all elements are sorted.**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**import java.util.PriorityQueue;**

**public class KthLargest {**

**private PriorityQueue<Integer> minHeap;**

**private int k;**

**public KthLargest(int k, int[] stream) {**

**this.k = k;**

**minHeap = new PriorityQueue<>(k);**

**for (int num : stream) {**

**add(num);**

**}**

**}**

**public int add(int num) {**

**if (minHeap.size() < k) {**

**minHeap.offer(num);**

**} else if (num > minHeap.peek()) {**

**minHeap.poll();**

**minHeap.offer(num);**

**}**

**return minHeap.peek();**

**}**

**public static void main(String[] args) {**

**// Test Case 1**

**int[] stream1 = {3, 10, 5, 20, 15};**

**KthLargest kthLargest1 = new KthLargest(3, stream1);**

**System.*out*.println("3rd largest: " + kthLargest1.add(0)); // Output: 10**

**// Test Case 2**

**int[] stream2 = {7, 4, 8, 2, 9};**

**KthLargest kthLargest2 = new KthLargest(2, stream2);**

**System.*out*.println("2nd largest: " + kthLargest2.add(6)); // Output: 8**

**}**

**}**

**Output:**

**3rd largest: 10**

**2nd largest: 8**

** Time Complexity:**

* **The add method takes O(log k) time, as it involves inserting or removing elements from a heap of size k.**
* **For n elements in the stream, the overall complexity is O(n log k).**

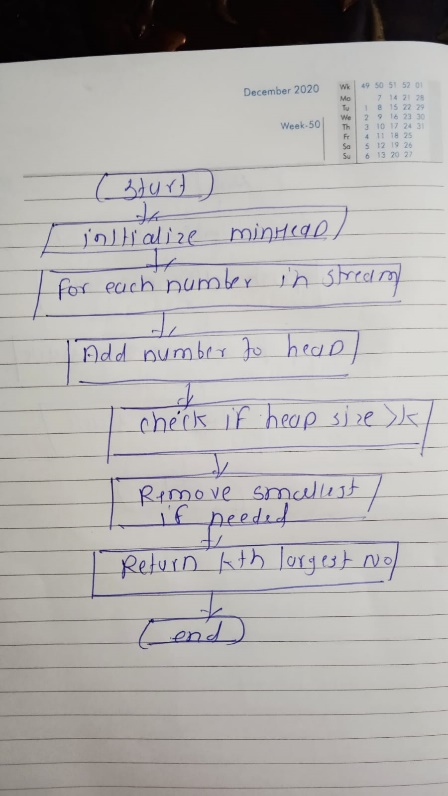
** Space Complexity:**

* **O(k)**

**Explanation**

* **Min-Heap Usage: The problem is solved using a min-heap of size k. The heap stores the k largest elements seen so far. If a new element is larger than the smallest element in the heap (the root), the root is removed, and the new element is added.**

**Flowchart:**



**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]**

**Program code:**

**package assignment;**

**import java.util.PriorityQueue;**

**class PriorityQueueExample {**

**public static void main(String[] args) {**

**System.*out*.println("Test Case 1:");**

**PriorityQueue<Element> pq1 = new PriorityQueue<>();**

**pq1.add(new Element(3, 1));**

**pq1.add(new Element(10, 3));**

**pq1.add(new Element(5, 2));**

**Element dequeued1 = pq1.poll();**

**System.*out*.println("Dequeued element = " + dequeued1.value);**

**System.*out*.print("Priority Queue = ");**

***printPriorityQueue*(pq1);**

**System.*out*.println("\nTest Case 2:");**

**PriorityQueue<Element> pq2 = new PriorityQueue<>();**

**pq2.add(new Element(7, 4));**

**pq2.add(new Element(8, 2));**

**pq2.add(new Element(6, 3));**

**Element dequeued2 = pq2.poll();**

**System.*out*.println("Dequeued element = " + dequeued2.value);**

**System.*out*.print("Priority Queue = ");**

***printPriorityQueue*(pq2);**

**}**

**private static void printPriorityQueue(PriorityQueue<Element> pq) {**

**for (Element elem : pq) {**

**System.*out*.print(elem.value + " ");**

**}**

**System.*out*.println();**

**}**

**}**

**class Element implements Comparable<Element> {**

**int value;**

**int priority;**

**Element(int value, int priority) {**

**this.value = value;**

**this.priority = priority;**

**}**

**@Override**

**public int compareTo(Element other) {**

**return Integer.*compare*(other.priority, this.priority);**

**}**

**}**

**Output code:**

**Test Case 1:**

**Dequeued element = 10**

**Priority Queue = 5 3**

**Test Case 2:**

**Dequeued element = 7**

**Priority Queue = 6 8**

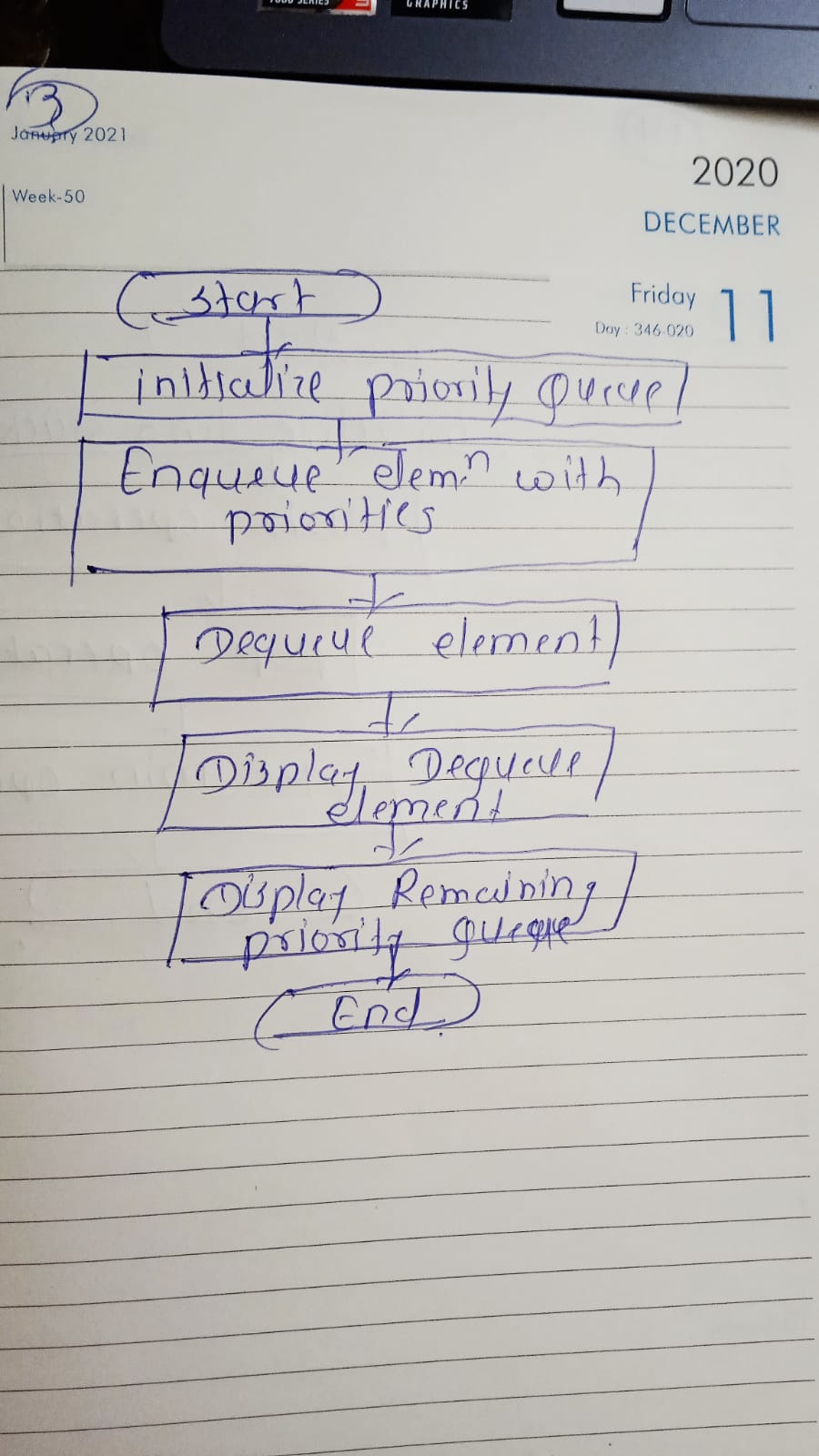
**Time and Space Complexity**

* **Time Complexity:**
  + **Enqueue (insert): O(log n), where n is the number of elements in the queue.**
  + **Dequeue (remove): O(log n), due to heap restructuring.**
* **Space Complexity: O(n), where n is the number of elements stored in the priority queue.**

**Explanation**

* **Priority Queue: A data structure where each element has a priority. Elements with higher priority are dequeued before elements with lower priority.**
* **Implementation: We use Java's PriorityQueue class, which uses a binary heap under the hood. The Element class contains the value and priority, implementing Comparable to order elements by priority.**
* **Enqueue: When adding elements, they are placed in the heap according to their priority.**
* **Dequeue: The element with the highest priority is removed from the queue.**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**import java.util.Stack;**

**class MinStack {**

**private Stack<Integer> stack;**

**private Stack<Integer> minStack;**

**public MinStack() {**

**stack = new Stack<>();**

**minStack = new Stack<>();**

**}**

**public void push(int value) {**

**stack.push(value);**

**if (minStack.isEmpty() || value <= minStack.peek()) {**

**minStack.push(value);**

**}**

**}**

**public int pop() {**

**int poppedValue = stack.pop();**

**if (poppedValue == minStack.peek()) {**

**minStack.pop();**

**}**

**return poppedValue;**

**}**

**public int getMin() {**

**return minStack.peek();**

**}**

**public void printStack() {**

**System.*out*.println("Stack: " + stack);**

**}**

**}**

**public class MinStackExample {**

**public static void main(String[] args) {**

**System.*out*.println("Test Case 1:");**

**MinStack minStack1 = new MinStack();**

**minStack1.push(5);**

**minStack1.push(3);**

**minStack1.push(7);**

**System.*out*.println("Min = " + minStack1.getMin()); // Output: Min = 3**

**System.*out*.println("\nTest Case 2:");**

**MinStack minStack2 = new MinStack();**

**minStack2.push(10);**

**minStack2.push(8);**

**minStack2.push(6);**

**minStack2.push(12);**

**System.*out*.println("Min = " + minStack2.getMin()); // Output: Min = 6**

**}**

**}**

**Output:**

**Test Case 1:**

**Min = 3**

**Test Case 2:**

**Min = 6**

**Explanation**

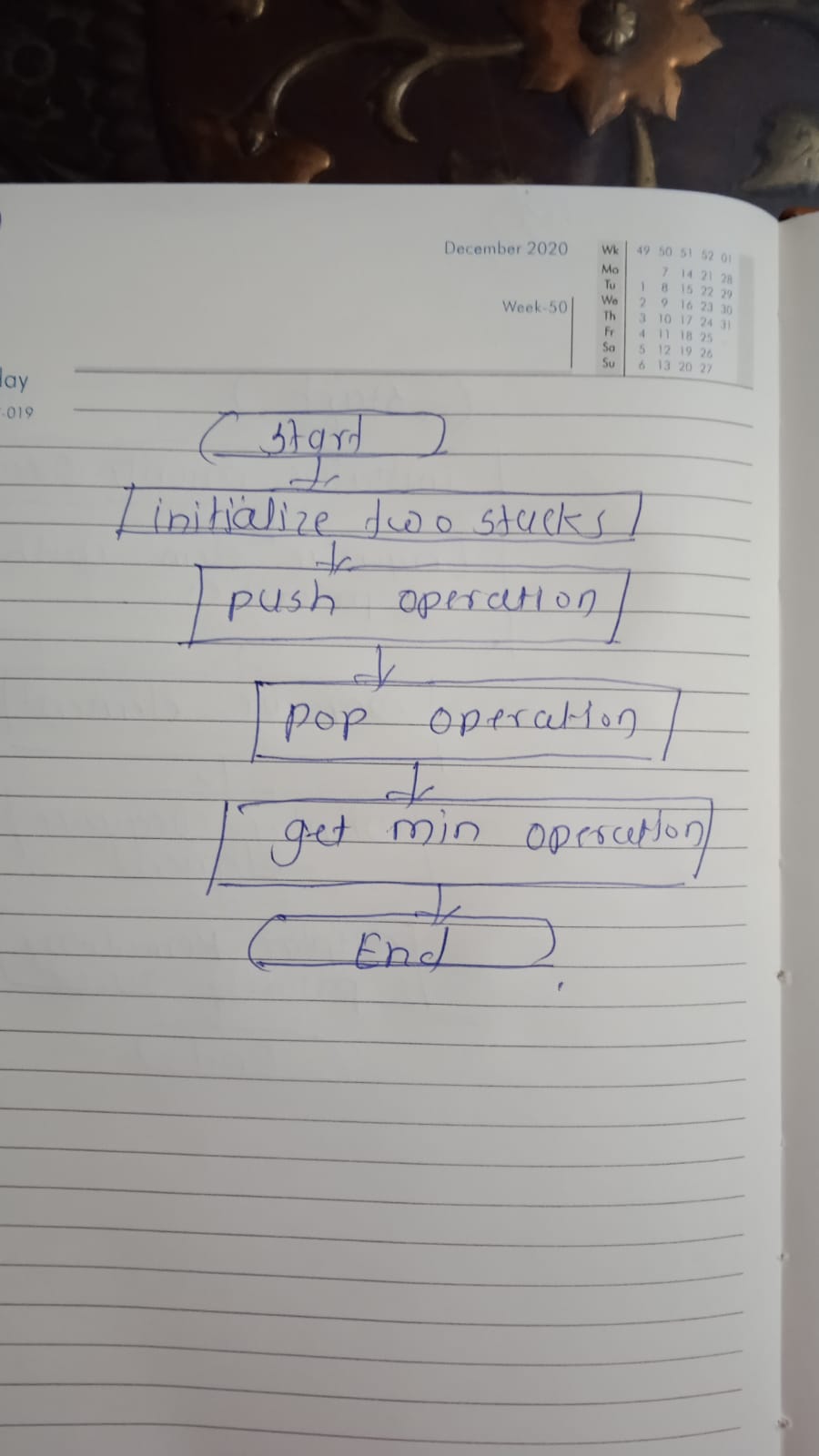
* **MinStack Class: Contains two stacks, one for normal stack operations (stack) and one to keep track of the minimum values (minStack).**
* **Push: When a value is pushed, it is added to stack. If it is less than or equal to the current minimum, it is also added to minStack.**
* **Pop: When an element is popped, if it is the current minimum (i.e., it equals the top of minStack), it is also removed from minStack.**
* **Get Min: This function simply returns the top of minStack, which holds the minimum value.**

** Time Complexity:**

* **Push: O(1) – Adding to the stack and possibly to the min stack takes constant time.**
* **Pop: O(1) – Removing from the stack and possibly from the min stack also takes constant time.**
* **Get Min: O(1) – Accessing the top of the min stack is a constant time operation.**

** Space Complexity: O(n)**

**Flowchart:**



**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**

**Program code:**

**package assignment;**

**class CircularQueue1 {**

**private int[] queue;**

**private int front, rear, size, capacity;**

**public CircularQueue1(int capacity) {**

**this.capacity = capacity;**

**queue = new int[capacity];**

**front = 0;**

**rear = 0;**

**size = 0;**

**}**

**public void enqueue(int value) {**

**if (isFull()) {**

**System.*out*.println("Queue is full. Cannot enqueue " + value);**

**return;**

**}**

**queue[rear] = value;**

**rear = (rear + 1) % capacity;**

**size++;**

**}**

**public int dequeue() {**

**if (isEmpty()) {**

**System.*out*.println("Queue is empty. Cannot dequeue.");**

**return -1;**

**}**

**int dequeuedValue = queue[front];**

**front = (front + 1) % capacity; // Circular increment**

**size--;**

**return dequeuedValue;**

**}**

**public boolean isFull() {**

**return size == capacity;**

**}**

**public boolean isEmpty() {**

**return size == 0;**

**}**

**}**

**public class CircularQueueExample {**

**public static void main(String[] args) {**

**System.*out*.println("Test Case 1:");**

**CircularQueue cq1 = new CircularQueue(4);**

**cq1.enqueue(1);**

**cq1.enqueue(2);**

**cq1.enqueue(3);**

**cq1.enqueue(4);**

**System.*out*.println("isFull() = " + cq1.isFull());**

**System.*out*.println("\nTest Case 2:");**

**CircularQueue cq2 = new CircularQueue(3);**

**cq2.enqueue(5);**

**cq2.enqueue(6);**

**cq2.dequeue();**

**cq2.enqueue(7);**

**System.*out*.println("isEmpty() = " + cq2.isEmpty());**

**}**

**}**

**Output code:**

**Test Case 1:**

**1 enqueued to the queue**

**2 enqueued to the queue**

**3 enqueued to the queue**

**4 enqueued to the queue**

**isFull() = true**

**Test Case 2:**

**5 enqueued to the queue**

**6 enqueued to the queue**

**7 enqueued to the queue**

**isEmpty() = false**

**Explanation**

* **Circular Queue Class: Maintains an array to store elements, along with pointers (front, rear) and variables (size, capacity) to manage the queue.**
* **Enqueue: Adds an element to the rear of the queue if it is not full. It wraps around using the modulo operator.**
* **Dequeue: Removes an element from the front of the queue if it is not empty. It also wraps around using the modulo operator.**
* **isFull: Checks if the current size of the queue equals its capacity.**
* **isEmpty: Checks if the current size of the queue is zero.**

**Time and Space Complexity**

* **Time Complexity:**
  + **Enqueue: O(1) – Adding an element takes constant time.**
  + **Dequeue: O(1) – Removing an element takes constant time.**
  + **isFull: O(1) – Checking if the queue is full takes constant time.**
  + **isEmpty: O(1) – Checking if the queue is empty takes constant time.**
* **Space Complexity: O(n) – The space used is proportional to the capacity of the queue.**

**Flowchart:**

