Lab Session VII
Program Inspection, Debugging and Static Analysis

I. PROGRAM INSPECTION:

Q1:- How many errors are there in the program? Mention the errors you have identified.

Data Reference Errors:

- Line 14: The `phone[10]` array may overflow if more than 9 characters are entered for the phone number, as there's no length check.
- Line 115: The variable `flag` in `display()` is not initialized before use. This could lead to incorrect behavior.
- Line 143: Similar to above, `flag` in `rooms()` is used without being initialized.

Data Declaration Errors:

- Line 6: The `#include <dos.h>` is outdated and not supported in modern C++ compilers.
- Line 5: `#include <conio.h>` is a non-standard library, and `getch()` (used throughout the program) may not work in modern C++.

Computation Errors:

- Line 368-374: In the `bill()` function, the room range check for luxury rooms is incorrect. It checks room numbers from 35 to 45, but it should check 31 to 45.
- Line 357-359: The file reading loop in the `bill()` function may skip customer data due to incorrect file reading logic.

Comparison Errors:

- Line 322: In `delete_rec()`, the comparison `if(ch == 'n')` only checks for lowercase `n`. It should also check for uppercase `N`.

Control Flow Errors:

- Line 39: The program loops indefinitely in the `main_menu()` unless the user selects 5, without handling invalid inputs properly.

- Line 122: The file reading in the `display()` function may not stop at the end of the file properly.
- Line 151: Similar issue in `rooms()`, where the file reading may not handle end-of-file correctly.

Interface Errors:

- Line 112: There's no input validation for the room number in `display()`.
- Line 143: Similarly, no input validation for room numbers is done in `rooms()`.
- Line 322: No validation is done for phone number format or length, which may lead to incorrect input storage.

Input/Output Errors:

- Line 97: Room number input is not validated, allowing the user to enter invalid room numbers.
- Line 322: Phone number input is vulnerable to overflow as there is no length limit or check.

Other Checks:

- Line 96: Writing data in binary format using `fout.write((char*)this, sizeof(hotel))` may cause compatibility issues across different platforms and compilers.

Q2:- Which category of program inspection would you find more effective?

The most effective inspection category is Input/Output Errors and Data Reference Errors. These are critical since the program handles user inputs and files, and validation is necessary to avoid errors.

Q3:- Which type of error you are not able to identified using the program inspection?

It’s hard to catch run-time errors like memory corruption or buffer overflow through inspection alone. These need testing with tools like Valgrind.

Q4:- Is the program inspection technique is worth applicable?

Yes, program inspection is helpful for finding logic errors and missing checks before running the program. However, it should be combined with testing, especially for input handling and file operations, to ensure everything works properly.

Github reference:-
https://github.com/The-Young-Programmer/C-CPP-Programming/blob/main/Projects/C%2B%2B%20Projects/Basic/Hotel%20Management%20System/main.cpp

II. CODE DEBUGGING:

1. Armstrong.txt

1. Errors Identified:

1. Incorrect calculation of the remainder: In the line `remainder = num / 10;`, it should be `remainder = num % 10;` because we need the last digit of the number, not the quotient.
2. Incorrect update of `num`: In the line `num = num % 10;`, it should be `num = num / 10;` to remove the last digit of the number.

2. Steps to Fix:

1. Update the calculation of the remainder using the modulus operator (`%`).
2. Correct the update of `num` to divide by 10 instead of using modulus.

3. Corrected Code:

```
class Armstrong {

    public static void main(String args[]) {

        int num = Integer.parseInt(args[0]);

        int n = num; // Use to check at last time

        int check = 0, remainder;

        while (num > 0) {

            remainder = num % 10; // Get the last digit

            check = check + (int)Math.pow(remainder, 3);

            num = num / 10; // Remove the last digit

        }

        if (check == n)

            System.out.println(n + " is an Armstrong Number");
```

```
        else
            System.out.println(n + " is not an Armstrong Number");
    }
}
```

2. GCD and LCM.txt

1. Errors in the Program:

- **GCD Calculation Logic:** In the `gcd` method, there is an error in the condition of the `while` loop. The current condition is `while(a % b == 0)`, which is incorrect. It should be `while(a % b != 0)` because the loop should continue until `a % b` equals 0.
- **LCM Calculation Logic:** The LCM calculation in the `lcm` method has a condition `if(a % x != 0 && a % y != 0)` that is incorrect. It should check for the opposite condition: `if(a % x == 0 && a % y == 0)` to ensure that `a` is divisible by both `x` and `y`.

2. Breakpoints:

- **Breakpoint 1:** In the `gcd` method at the `while(a % b == 0)` condition to verify the GCD calculation.
- **Breakpoint 2:** In the `lcm` method at the `if(a % x != 0 && a % y != 0)` condition to check the logic for finding LCM.

3. Steps Taken to Fix the Errors:

- **Fix for GCD Calculation:** Change `while(a % b == 0)` to `while(a % b != 0)` to ensure the loop runs while the remainder is non-zero.
- **Fix for LCM Calculation:** Update the condition `if(a % x != 0 && a % y != 0)` to `if(a % x == 0 && a % y == 0)` to properly check for divisibility by both `x` and `y`.

4. Executable Code:

```
import java.util.Scanner;

public class GCD_LCM
{
    static int gcd(int x, int y)
    {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is the greater number
        b = (x < y) ? x : y; // b is the smaller number

        while (b != 0)
        {
            r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    static int lcm(int x, int y)
    {
        int a = (x > y) ? x : y; // a is the greater number
        while (true)
        {
            if (a % x == 0 && a % y == 0)
                return a;
            ++a;
        }
    }

    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

3. Knapsack

1. Errors in the Program:

- Increment in the Knapsack Loop: In the line `int option1 = opt[n++][w];`, the `n++` causes the index to increment unexpectedly. It should be `opt[n][w]` instead of `opt[n++][w]` to avoid altering `n` during the loop.
- Incorrect Profit Calculation: In the line `if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]];`, the condition should check if `weight[n] <= w`, not greater, because we can only consider the item if its weight is less than or equal to the remaining capacity. Additionally, `profit[n-2]` should be `profit[n]` to reference the correct item.

2. Breakpoints:

- Breakpoint 1: In the nested loop at `int option1 = opt[n++][w];` to check the correct behavior of the `opt` array.
- Breakpoint 2: At `if (weight[n] > w)` to ensure the condition is logically valid.

3. Steps Taken to Fix the Errors:

- Fix 1 (Knapsack Loop Increment): Replace `opt[n++][w]` with `opt[n][w]` to prevent unintended index increment.
- Fix 2 (Profit Calculation): Update the condition to `if (weight[n] <= w)` and use `profit[n]` instead of `profit[n-2]` for accurate profit calculation.

4. Executable Code:

```
public class Knapsack {

   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);   // number of items
      int W = Integer.parseInt(args[1]);   // maximum weight of knapsack

      int[] profit = new int[N+1];
      int[] weight = new int[N+1];

      // generate random instance, items 1..N
      for (int n = 1; n <= N; n++) {
```

```
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight limit w
        // sol[n][w] = does opt solution to pack items 1..n with weight limit w include item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n][w];

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) option2 = profit[n] + opt[n-1][w-weight[n]];

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) { take[n] = true;  w = w - weight[n]; }
            else           { take[n] = false;                    }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```

4. Magic number

1. Errors in the Program:

- Sum Calculation Logic: The inner loop condition `while(sum==0)` is incorrect. It should be `while(sum!=0)` to iterate until all digits of the number are processed.
- Sum of Digits: The statement `s = s * (sum / 10)` is incorrect. It should sum the digits, not multiply. The correct logic is `s += sum % 10` to accumulate the digits of the number.
- Syntax Error: The line `sum=sum%10` is missing a semicolon at the end.

2. Breakpoints:

- Breakpoint 1: Before the inner loop to verify the sum calculation logic.
- Breakpoint 2: After checking the updated sum to confirm the correct value of `num`.

3. Steps Taken to Fix the Errors:

- Fix 1: Change `while(sum==0)` to `while(sum!=0)` for proper digit extraction.
- Fix 2: Update `s=s*(sum/10)` to `s += sum % 10` for correct sum of digits logic.
- Fix 3: Add a semicolon at `sum = sum % 10;`.

4. Executable Code:

```
import java.util.*;

public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while (num > 9) {
            sum = num;
            int s = 0;
            while (sum != 0) {
                s += sum % 10;
                sum = sum / 10;
            }
            num = s;
        }
```

```
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

5. Merge sort

1. Errors in the Program:

- Incorrect Array Splitting Logic:
    - In `mergeSort`, the array splitting uses incorrect operations: `int[] left = leftHalf(array + 1)` and `int[] right = rightHalf(array - 1)`. These expressions attempt to manipulate arrays like numbers, which causes an error.
    - In the merge call `merge(array, left++, right--);`, increment and decrement operators (`++` and `--`) are incorrectly used on arrays, causing a compilation error.

2. Breakpoints:

- Breakpoint 1: Before splitting the array into two halves in `mergeSort`.
- Breakpoint 2: During the merge phase to ensure proper merging of sorted sub-arrays.

3. Steps Taken to Fix the Errors:

- Fix 1: Remove the incorrect operations `+1` and `-1` from `leftHalf(array)` and `rightHalf(array)`. Simply pass the `array` to these functions as-is.
- Fix 2: Remove the `++` and `--` operators in `merge(array, left++, right--)` and pass `left` and `right` directly.

4. Executable Code:

```
import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before:  " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:   " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // post: array is in sorted (nondecreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);

            // recursively sort the two halves
            mergeSort(left);
            mergeSort(right);

            // merge the sorted halves into a sorted whole
            merge(array, left, right);
        }
    }

    // Returns the first half of the given array.
    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

    // Returns the second half of the given array.
    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
```

```
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    // Merges the given left and right arrays into the given
    // result array.  Second, working version.
    // pre : result is empty; left/right are sorted
    // post: result contains result of merging sorted lists;
    public static void merge(int[] result,
                             int[] left, int[] right) {
        int i1 = 0;  // index into left array
        int i2 = 0;  // index into right array

        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length &&
                    left[i1] <= right[i2])) {
                result[i] = left[i1];    // take from left
                i1++;
            } else {
                result[i] = right[i2];   // take from right
                i2++;
            }
        }
    }
}
```

6. Multiply two matrices

1. Errors in the Program:

- Incorrect Indexing in Matrix Multiplication:
  - In the nested `for` loops, the matrix element multiplication and summation are using incorrect indices: `first[c-1][c-k]` and `second[k-1][k-d]`. The indices with `-1` are invalid and cause an `ArrayIndexOutOfBoundsException`.
- Improper Input Parsing:
  - The input parsing for the matrix elements should read row by row, but the input provided reads the entire row at once.
  -

2. Breakpoints:

- Breakpoint 1: During the multiplication process, when matrices are being accessed, to monitor array indices and prevent index errors.

3. Steps Taken to Fix the Errors:

- Fix 1: Correct the matrix multiplication indexing by replacing `first[c-1][c-k]` with `first[c][k]` and `second[k-1][k-d]` with `second[k][d]`.
- Fix 2: Ensure that the matrix elements are correctly parsed for each row.

4. Executable Code:

```
import java.util.Scanner;

class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q, sum = 0, c, d, k;

      Scanner in = new Scanner(System.in);
      System.out.println("Enter  the number of rows and columns of the first matrix:");
      m = in.nextInt();
      n = in.nextInt();

      int first[][] = new int[m][n];

      System.out.println("Enter  the elements of the first matrix:");
      for (c = 0; c < m; c++)
         for (d = 0; d < n; d++)
            first[c][d] = in.nextInt();

      System.out.println("Enter  the number of rows and columns of the second matrix:");
      p = in.nextInt();
      q = in.nextInt();

      if (n != p) {
         System.out.println("Matrices  with entered orders can't be multiplied with each other.");
      } else {
         int second[][] = new int[p][q];
         int multiply[][] = new int[m][q];

         System.out.println("Enter  the elements of the second matrix:");
         for (c = 0; c < p; c++)
            for (d = 0; d < q; d++)
               second[c][d] = in.nextInt();
```

```
        // Matrix multiplication
        for (c = 0; c < m; c++) {
            for (d = 0; d < q; d++) {
                for (k = 0; k < n; k++) {
                    sum += first[c][k] * second[k][d];
                }
                multiply[c][d] = sum;
                sum = 0;
            }
        }

        // Display the result
        System.out.println("Product  of entered matrices:");
        for (c = 0; c < m; c++) {
            for (d = 0; d < q; d++)
                System.out.print(multiply[c][d]  + "\t");
            System.out.println();
        }
    }
  }
}
```

7. Quadratic Probing

1. Errors in the Program:

- Incorrect Increment Operation in `insert` Method:
  - In the `insert` method, the line `i + = (i + h / h--) % maxSize;` has a typo (`i + =` should be `i =`).
  - The current quadratic probing approach is incorrect. Instead of incrementing by `(i + h / h--)`, the correct increment should be `i = (i + h * h) % maxSize`.
- Infinite Loop Risk in `insert` Method:
  - The program may enter an infinite loop if the table is full or if a probing cycle forms. There should be a condition to stop after checking all indices.
- Incorrect Deletion Logic in `remove` Method:
  - After deleting a key, the decrement of `currentSize` happens twice, once within the loop and once at the end, which is unnecessary.

2. Steps Taken to Fix the Errors:

- Fix 1: Correct the increment operation in the `insert` method.
- Fix 2: Ensure the correct rehashing in the `remove` method without redundant decrements.
- Fix 3: Handle quadratic probing by ensuring proper usage of the probing formula.

3. Corrected Code:

```
import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to clear hash table **/
    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to get size of hash table **/
    public int getSize() {
        return currentSize;
    }

    /** Function to check if hash table is full **/
    public boolean isFull() {
        return currentSize == maxSize;
    }

    /** Function to check if hash table is empty **/
    public boolean isEmpty() {
        return getSize() == 0;
    }
```

```
/** Function to check if hash table contains a key **/
public boolean contains(String key) {
    return get(key) != null;
}

/** Function to get hash code of a given key **/
private int hash(String key) {
    return key.hashCode() % maxSize;
}

/** Function to insert key-value pair **/
public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;
    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i = (i + h * h) % maxSize; // Correct quadratic probing
        h++;
    } while (i != tmp && currentSize < maxSize);
}

/** Function to get value for a given key **/
public String get(String key) {
    int i = hash(key), h = 1;
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h) % maxSize;
        h++;
    }
    return null;
}

/** Function to remove key and its value **/
```

```
    public void remove(String key) {
        if (!contains(key))
            return;

        /** Find position of key and delete **/
        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h) % maxSize;
        keys[i] = vals[i] = null;

        /** Rehash all keys **/
        for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h) % maxSize) {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }

    /** Function to print HashTable **/
    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++)
            if (keys[i] != null)
                System.out.println(keys[i] + " " + vals[i]);
        System.out.println();
    }
}

/** Class QuadraticProbingHashTableTest **/
public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Hash Table Test\n\n");
        System.out.println("Enter size:");
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

        char ch;
        /** Perform QuadraticProbingHashTable operations **/
        do {
            System.out.println("\nHash Table Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. remove");
```

```
            System.out.println("3.  get");
            System.out.println("4.  clear");
            System.out.println("5.  size");

            int choice = scan.nextInt();
            switch (choice) {
               case 1:
                  System.out.println("Enter  key and value");
                  qpht.insert(scan.next(),  scan.next());
                  break;
               case 2:
                  System.out.println("Enter  key");
                  qpht.remove(scan.next());
                  break;
               case 3:
                  System.out.println("Enter  key");
                  System.out.println("Value  = " + qpht.get(scan.next()));
                  break;
               case 4:
                  qpht.makeEmpty();
                  System.out.println("Hash Table Cleared\n");
                  break;
               case 5:
                  System.out.println("Size  = " + qpht.getSize());
                  break;
               default:
                  System.out.println("Wrong  Entry\n");
                  break;
            }
            /** Display hash table **/
            qpht.printHashTable();

            System.out.println("\nDo  you want to continue (Type y or n)\n");
            ch = scan.next().charAt(0);
         } while (ch == 'Y' || ch == 'y');
      }
   }
```

8. Sorting Array

1. Errors in the Program

- Class Name: The class name `Ascending _Order` contains a space, which is not allowed in Java. Class names should not have spaces.
- Loop Condition: The outer `for` loop's condition `i >= n` is incorrect. It should be `i < n`.
- Unnecessary Semicolon: There is a semicolon at the end of the first `for` loop (`for (int i = 0; i >= n; i++);`), which makes it an empty loop.
- Sorting Logic: The sorting logic inside the nested loop is incorrect. It should sort the elements in ascending order, which means swapping should occur when `a[i] > a[j]`.
- Printing Elements: The final print statement prints an additional comma after the last element, which is not desirable.

2. Breakpoints

- Correct the Class Name: Change `Ascending _Order` to `AscendingOrder`.
- Fix Loop Condition: Change `for (int i = 0; i >= n; i++)` to `for (int i = 0; i < n; i++)`.
- Remove Semicolon: Remove the semicolon after the outer loop declaration.
- Correct Sorting Logic: Update the condition in the nested loop from `if (a[i] <= a[j])` to `if (a[i] > a[j])`.
- Adjust Printing Logic: Modify the final print statements to prevent an extra comma after the last element.

3. Corrected Code

```
import java.util.Scanner;

public class AscendingOrder
{
   public static void main(String[] args)
   {
      int n, temp;
      Scanner s = new Scanner(System.in);
      System.out.print("Enter  no. of elements you want in array: ");
      n = s.nextInt();
      int a[] = new int[n];
      System.out.println("Enter  all the elements:");
```

```
        for (int i = 0; i < n; i++)
        {
            a[i] = s.nextInt();
        }

        // Corrected sorting logic
        for (int i = 0; i < n; i++)
        {
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] > a[j])
                {
                    // Swap elements
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++)
        {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]); // Print last element without trailing comma

        s.close(); // Close the scanner
    }
}
```

9. Stack implementation

1. Errors in the Program

- Incorrect Indexing in `push` Method: The line `top--;` should be `top++;` when adding a value to the stack. This will correctly increment the `top` index before placing the value.
- Incorrect Indexing in `display` Method: The loop condition in the `display` method `for(int i=0;i>top;i++)` should be `i<=top`. This change ensures it loops through all valid indices of the stack.

- Top Initialization: The initial value of `top` is `-1`, which is correct. However, when you pop an element, you should check the condition properly and adjust the `top` variable accordingly. This is already done but ensure that the logic is correct in the `pop` method.
- Error Handling: It is a good practice to handle situations where push and pop operations are called when the stack is full or empty, respectively.

2. Steps Taken to Fix the Errors:

- Fix the `push` Method: Change `top--;` to `top++;` before assigning `value` to `stack[top]`.
- Fix the `display` Method: Update the loop condition to `i <= top` to print all elements in the stack.
- Add `System.out.println()` in `pop`: It can be helpful to print the value being popped for clarity.

3. Corrected Code

```
import java.util.Arrays;

public class StackMethods {
    private int top;
    int size;
    int[] stack;

    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack  is full, can't push a value");
        } else {
            top++; // Increment top to add the value
            stack[top] = value; // Place the value in the stack
        }
    }
```

```
    public void pop() {
        if (!isEmpty()) {
            System.out.println("Popped value: " + stack[top]); // Print the value being popped
            top--; // Decrement top to remove the value
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        System.out.print("Stack elements: ");
        for (int i = 0; i <= top; i++) { // Corrected condition
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop(); // Attempt to pop from an empty stack
        newStack.display();
    }
}
```

10. Tower of Hanoi

1. Errors in the Program

- Incorrect Increment and Decrement: In the recursive call `doTowers(topN ++, inter--, from + 1, to + 1)`, the `++` and `--` operators are misused. The arguments should simply pass the values without incrementing or decrementing them incorrectly.
- Use `topN - 1` for the recursive call.
- Use `inter` as it is, not decremented.
- Use `from` and `to` directly as they represent the character rods, so no addition is necessary.
- Base Case Missing: In the base case, you need to call the `doTowers` method recursively for the case when `topN` is greater than 1 and correctly return the disks. The structure of the method is slightly off.

2. Steps Taken to Fix the Errors:

- Correct the Recursive Calls: Make sure to pass the right parameters to the recursive calls without modifying them unnecessarily.
- Ensure Correct Disk Movement: Ensure that you are correctly tracking the disks and their movements according to the rules of the Tower of Hanoi.

3. Corrected Code

```
public class MainClass {
  public static void main(String[] args) {
    int nDisks = 3; // Number of disks
    doTowers(nDisks, 'A', 'B', 'C'); // A = source, B = auxiliary, C = destination
  }

  public static void doTowers(int topN, char from, char inter, char to) {
    if (topN == 1) {
      System.out.println("Disk  1 from " + from + " to " + to);
    } else {
      doTowers(topN - 1, from, to, inter); // Move topN-1 disks from source to auxiliary
      System.out.println("Disk  " + topN + " from " + from + " to " + to); // Move the largest disk
      doTowers(topN - 1, inter, from, to); // Move the disks from auxiliary to destination
    }
  }
}
```