o **IT314: Software Engineering**

o **Lab Assignment 7**

o **Aman Mangukiya -202201156**

O **Course instructor: Prof. Saurabh Tiwari**

Armstrong

1. Program Inspection

1. Errors Identified:
   - Remainder calculation: `num / 10` → should be `num % 10`.
   - Incorrect `num` update: `num % 10` → should be `num / 10`.
2. Effective Inspection Categories:
   - Category A: Data Reference Errors
   - Category C: Computation Errors
3. Error Not Easily Identified:
   - Logical intent errors (e.g., Armstrong logic correctness).
4. Applicability of Program Inspection:
   - Yes, it helps catch critical mistakes.

---

2. Code Debugging

1. Errors Identified:
   - 2 errors: Incorrect remainder calculation and incorrect update of `num`.
2. Breakpoints Needed:
   - 2 breakpoints:
     1. First breakpoint in the `while` loop (check `remainder` calculation).
     2. Second breakpoint after `num` update.
3. Steps Taken:
   - Step 1: Set breakpoints in the `while` loop.
   - Step 2: Fix remainder calculation and `num` update logic.
   - Step 3: Re-run to verify output.

```java
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num;
        int check = 0, remainder;

        while (num > 0) {
            remainder = num % 10;  // Fix
            check += (int) Math.pow(remainder, 3);
            num = num / 10;  // Fix
        }

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

GCD and LCM Code

I. Program Inspection:

1. Errors Identified:
   - GCD Error: Incorrect condition in `while(a % b == 0)` → should be `while(a % b != 0)`.
   - LCM Error: Incorrect logic in `if(a % x != 0 && a % y != 0)` → should be `if(a % x == 0 && a % y == 0)`.
2. Effective Inspection Categories:
   - Category A: Data Reference Errors
   - Category C: Computation Errors
3. Error Not Easily Identified:
   - Infinite loop issue in LCM logic due to the wrong condition.
4. Applicability of Program Inspection:
   - Yes, it helps catch logical errors in the loop and condition.

II. Code Debugging

1. Errors Identified:
   - 2 errors:
     1. Incorrect GCD condition `while(a % b == 0)`.
     2. Incorrect LCM condition `if(a % x != 0 && a % y != 0)`.
2. Breakpoints Needed:
   - 2 breakpoints:
     1. First breakpoint in `gcd()` to check loop condition.
     2. Second breakpoint in `lcm()` to check the loop and condition.
3. Steps Taken:

- Step 1: Set breakpoints in `gcd()` and `lcm()`.
- Step 2: Fix the conditions in both methods.
- Step 3: Re-run to verify correct output.

4. Complete 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) ? y : x; // a is smaller number
        b = (x < y) ? x : y; // b is greater number

        while(a % b != 0) {  // Corrected condition
               r = a % b;
               a = b;
            b = r;
        }
           return b;   // Return b instead of r
    }

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

Knapsack Code

I. Program Inspection:

1. Errors Identified:
    - Increment Error in Option 1: `int option1 = opt[n++][w];` should be `opt[n-1][w]`.
    - Array Index Error in Option 2: `profit[n-2]` should be `profit[n]`.
2. Effective Inspection Categories:
    - Category A: Off-by-one and index issues.
    - Category C: Incorrect array reference.
3. Error Not Easily Identified:
    - Logical error in calculating `option1` with incrementing `n++`.
4. Applicability of Program Inspection:
    - Yes, it helps catch common array indexing and logic errors.

II. Code Debugging

1. Errors Identified:
    - 2 errors:
        1. Increment error in `opt[n++][w]`.
        2. Incorrect index `profit[n-2]` instead of `profit[n]`.
2. Breakpoints Needed:
    - 2 breakpoints:
        1. First breakpoint to inspect the loop where `opt` is being calculated.
        2. Second breakpoint to verify `option2` calculation.
3. Steps Taken:
    - Step 1: Set breakpoints to inspect `opt` and `sol` arrays.

- Step 2: Fix the `opt[n-1][w]` and correct the `profit[n]` index.
- Step 3: Re-run to verify correct output.

4. Complete 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
        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-1][w]; // Corrected from n++ to
n-1


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

                // 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]);
        }
    }
}
```

Magic Number Check Code

I. Program Inspection:

1. Errors Identified:
    - Incorrect Condition: In while(sum==0), it should be while(sum>0).
    - Incorrect Calculation: The line s=s*(sum/10); should be s=s+ (sum % 10); to accumulate the digits.
    - Missing Semicolon: After sum=sum%10, a semicolon is required.
2. Effective Inspection Categories:
    - Category A: Logical errors in calculations.
    - Category C: Incorrect flow of control and infinite loops.
3. Error Not Easily Identified:
    - Logic error in calculating the sum of the digits could lead to infinite loops if not caught during inspection.
4. Applicability of Program Inspection:
    - Yes, it can identify common logic errors and control flow issues.

II. Code Debugging

- Errors Identified:
    - 3 errors:
        1. Incorrect loop condition: while(sum==0) should be while(sum>0).
        2. Incorrect calculation of s.
        3. Missing semicolon after sum=sum%10.
- Breakpoints Needed:
    - 2 breakpoints:
        1. Before the while(num>9) loop to inspect initial values.

2. Inside the inner loop to verify calculations of sum and s.

- Steps Taken:
    - **Step 1**: Set breakpoints to monitor the flow of values.
    - **Step 2**: Fix the loop condition and calculation errors.
    - **Step 3**: Re-run the code to verify correct outputs.
- Complete 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) { // Corrected condition
        s = s + (sum % 10); // Corrected calculation
        sum = sum / 10; // Corrected calculation
      }
      num = s;
    }

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

```
    }

}
```

Merge Sort Code

I. Program Inspection: Merge Sort Code

1. Errors Identified:
   - Incorrect Array Slicing: The methods leftHalf and rightHalf are attempting to slice the array incorrectly using array+1 and array-1, which are invalid operations. They should instead use Arrays.copyOfRange or pass the appropriate segments of the array.
   - Incorrect Merge Parameters: In the merge call, left++ and right-- are invalid. They should just be left and right since they are array references, not integers.
   - Merge Method Argument: The merge method is being called with array, but it needs to be called with a new array of the same length as the original array for merging the results.
2. Effective Inspection Categories:
   - Category A: Logic errors in array manipulation.
   - Category C: Incorrect handling of control flow and array bounds.
3. Error Not Easily Identified:
   - The incorrect array slicing and merging logic may not trigger compilation errors, making it harder to detect without running tests.
4. Applicability of Program Inspection:
   - Yes, it is worth applying as it helps in identifying logical errors that may not be evident during initial code writing.

II. Code Debugging

1. Errors Identified:

- 3 errors:
    1. Incorrect array slicing in mergeSort.
    2. Incorrect parameters used in merge method call.
    3. Invalid increment/decrement on array references.
2. Breakpoints Needed:
    - 3 breakpoints:
        1. Before the mergeSort call to check initial array values.
        2. Inside mergeSort to inspect the left and right arrays after slicing.
        3. Before the merge method to confirm correct array contents.
3. Steps Taken:
    - Step 1: Set breakpoints to inspect values at critical points.
    - Step 2: Correct slicing and merge method call.
    - Step 3: Rerun the code to validate proper sorting.
4. Complete 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));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int mid = array.length / 2;
            int[] left = Arrays.copyOfRange(array, 0, mid);
            int[] right = Arrays.copyOfRange(array, mid,
array.length);
```

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

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

    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++;
            }
        }
    }
}
```

Matrix Multiplication Code

I. Program Inspection:

1.Errors Identified:

- Incorrect Array Indexing: The lines first[c-1][c-k] and second[k-1][k-d] use incorrect indexing. This will lead to ArrayIndexOutOfBoundsException. The correct indexing should be first[c][k] and second[k][d].
- Initialization of sum: The variable sum is being reused outside of its intended scope. It should be initialized inside the loop where it's being used.
- Output Formatting: The input and output prompts are not clear, and the second input prompt is incorrectly repeated.

2. Effective Inspection Categories:
   - Category A: Logic errors in matrix operations.
   - Category B: Indexing and boundary errors.
3. Error Not Easily Identified:
   - The incorrect array indexing will not throw an error until runtime, making it hard to detect during code inspection.
4. Applicability of Program Inspection:
   - Yes, program inspection is valuable as it helps identify logical and syntactical issues before running the program.

II. Code Debugging

1. Errors Identified:
   - 3 errors:
     1. Incorrect array indexing in the multiplication logic.

    2. Improper initialization of sum.
    3. Repeated input prompts for the second matrix dimensions.
2. Breakpoints Needed:
    - 3 breakpoints:
        1. Before the matrix multiplication logic to check the contents of both matrices.
        2. Inside the multiplication loop to inspect values of sum, first, and second.
        3. Before printing the final product matrix to verify its correctness.
3. Steps Taken:
    - Step 1: Set breakpoints to inspect values at critical points.
    - Step 2: Correct array indexing and initialization of sum.
    - Step 3: Rerun the code to validate proper multiplication.
4. Complete Executable Code:

```
import java.util.Scanner;


class MatrixMultiplication {

    public static void main(String args[]) {

        int m, n, p, q, sum, c, d, k;


        Scanner in = new Scanner(System.in);

        System.out.println("Enter the number of rows and columns
of first matrix");

        m = in.nextInt();

        n = in.nextInt();
```

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


        System.out.println("Enter the elements of 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 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 second
matrix");


            for (c = 0; c < p; c++)

                for (d = 0; d < q; d++)
```

```
                    second[c][d] = in.nextInt();


            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    sum = 0; // Initialize sum inside the loop
                    for (k = 0; k < n; k++) { // Changed from p
to n
                        sum += first[c][k] * second[k][d]; //
Corrected indexing
                    }
                    multiply[c][d] = sum;
                }
            }


            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.print("\n");
            }
        }
        in.close(); // Close the scanner
    }
}
```

Quadratic Probing Hash Table Code

I. Program Inspection:

1. Errors Identified:
    - Syntax Errors in Insertion Logic:
        - The line i + = (i + h / h--) % maxSize; contains a space that should be removed. It should be i += (i + h * h) % maxSize;.
        - In the get and remove methods, i = (i + h * h++) % maxSize; has the same issue. The increment operator (++) should be used correctly.
    - Incorrect use of hash in remove method:
        - The condition while (!key.equals(keys[i])) could potentially lead to an infinite loop if the key does not exist in the table. It should include a check for keys[i] == null.
2. Effective Inspection Categories:
    - Category A: Logic errors in insertion, removal, and retrieval processes.
    - Category B: Syntax and boundary errors.
3. Error Not Easily Identified:
    - Logical errors that might not produce immediate exceptions or errors (like infinite loops in the removal process).
4. Applicability of Program Inspection:

    - Yes, this technique is useful for identifying logical flaws, particularly in complex data structures like hash tables.

II. Code Debugging

1. Errors Identified:
    - 3 errors:
        1. Incorrect syntax in incrementing i.
        2. Potential infinite loop in the remove method.
        3. Incorrect hash logic in insert.
2. Breakpoints Needed:
    - 3 breakpoints:
        1. Before the insertion logic to check hash, keys, and vals.
        2. In the get and remove methods to inspect i values and conditions.
        3. Before returning the size to validate currentSize.
3. Steps Taken:
    - Step 1: Set breakpoints to observe the flow of data and identify where logic fails.
    - Step 2: Correct syntax errors and logical errors based on observations.
    - Step 3: Rerun the code to validate that it works as intended.
4. Complete Executable 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 Math.abs(key.hashCode() % maxSize); // Use Math.abs to
avoid negative index

    }



    /** Function to insert key-value pair **/

    public void insert(String key, String val) {

        int tmp = hash(key);

        int i = tmp, h = 1;
```

```java
        do {
            if (keys[i] == null) {
                keys[i] = key;
                vals[i] = val;
                currentSize++;
                return;
            }
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
            i += (h * h) % maxSize; // Corrected the increment syntax
            h++;
        } while (i != tmp);
    }

    /** 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; // Corrected increment
            h++;
```

```
        }

        return null;
    }

    /** Function to remove key and its value **/
    public void remove(String key) {
        if (!contains(key))
            return;

        /** find position key and delete **/
        int i = hash(key), h = 1;

        while (!key.equals(keys[i])) {
            if (keys[i] == null) // Check to prevent infinite loop
                return;
            i = (i + h * h) % maxSize;
            h++;
        }

        keys[i] = vals[i] = null;

        /** rehash all keys **/
        for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h *
h) % maxSize) {
```

```java
            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");

        /** Create object of QuadraticProbingHashTable **/

        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');

        scan.close();
```

```
        }


}
```

Ascending Order Array Sorting

I. Program Inspection:

1. Errors Identified:
   - Class Name Issue: Invalid class name Ascending _Order (space present).
   - Loop Condition Error: Outer loop condition for (int i = 0; i >= n; i++) should be for (int i = 0; i < n; i++).
   - Sorting Logic Error: Condition in sorting logic if (a[i] <= a[j]) should be if (a[i] > a[j]) to swap for ascending order.
   - Extra Semicolon: An unnecessary semicolon after the outer loop declaration terminated the loop prematurely.
   - Print Statement Logic: Can be simplified by using Arrays.toString() for better output formatting.
2. Effective Program Inspection Category:
   - Static Analysis: Reviewing code structure and logic without executing it.
3. Error Types Not Identified Using Program Inspection:
   - Logical Errors: Some logical errors may not be apparent without executing the code.
4. Applicability of Program Inspection Technique:
   - Worthwhile: Program inspection is valuable for identifying syntax and structural errors before runtime.

II. Code Debugging

Errors Identified:

- Class Name Issue: Invalid class name Ascending _Order.
- Loop Condition Error: Incorrect outer loop condition.

- Sorting Logic Error: Incorrect condition in the sorting logic.
- Extra Semicolon: Improper termination of the loop.
- Print Statement Logic: Could be improved for better output formatting.

2. Breakpoints Needed:
   - 3 Breakpoints:
     - Before the outer loop to check the initialization of the array.
     - Inside the sorting logic to observe the values before and after the swap.
     - Before the print statement to verify the final sorted array.

3. Steps to Fix Errors:

   - Rename the class to AscendingOrder.
   - Change the outer loop condition to i < n.
   - Update the sorting condition to if (a[i] > a[j]).
   - Remove the extra semicolon after the outer loop.
   - Use Arrays.toString(a) for printing the sorted array.

3. Complete Executable Code:

```
import java.util.Arrays;

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();
```

```java
        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]) { // Changed <= to >

                    // Swap a[i] and a[j]

                    temp = a[i];

                    a[i] = a[j];

                    a[j] = temp;

                }

            }

        }



        // Print sorted array

        System.out.print("Ascending Order: ");

        System.out.println(Arrays.toString(a)); // Simplified print
statement

    }

}
```

Stack Implementation

Program Inspection:

1. Errors Identified:
   - Push Method:
     - Incorrectly decrements top before assignment: top-- should be top++.
   - Display Method:
     - Loop condition is incorrect: for(int i=0;i>top;i++) should be for(int i=0;i<=top;i++).
2. Effective Inspection Category:
   - Data Reference Errors:
     - Incorrect handling of stack operations (pushing and popping).
3. Unidentified Errors:
   - Logic errors regarding stack operations might not be caught without testing.
   - No handling for underflow in pop method (popping from an empty stack).
4. Applicability of Inspection Technique:
   - Yes, it's worth it to identify logic errors and ensure proper data handling.

Code Debugging

1. Errors Identified:
   - Same as above for the push and display methods.
2. Breakpoints Needed:
   - One breakpoint at the start of each method to trace execution flow (especially in push and display).
3. Steps to Fix Errors:
   - Correct the push method to increment top after adding an element.
   - Fix the loop condition in the display method to properly iterate through the stack.

Complete Executable Code (Fixed)

```
// Stack implementation in Java
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++;
            stack[top] = value; // Corrected to top++
        }
    }

    public void pop() {
        if (!isEmpty())
            top--; // Corrected to top-- for proper pop operation
        else {
            System.out.println("Can't pop...stack is empty");
        }
    }

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

    public void display() {
        for (int i = 0; i <= top; i++) { // Corrected loop condition
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
```

```java
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.display();
    }
}
```

Tower of Hanoi Implementation

Program Inspection: Tower of Hanoi Implementation

1. Errors Identified:
   - In doTowers Method:
     - Incorrectly using ++ and -- operators: doTowers(topN ++, inter--, from+1, to+1) should not increment or decrement the variables. Instead, it should be doTowers(topN - 1, inter, from, to) for proper recursion.
     - The parameters from and to should be passed as is without arithmetic operations (i.e., no from + 1, to + 1).
2. Effective Inspection Category:
   - Data Reference Errors:
     - Incorrect handling of parameters for recursive calls.
3. Unidentified Errors:
   - Logical errors in the recursive structure might not be identified without thorough testing.
4. Applicability of Inspection Technique:
   - Yes, it is useful to catch recursion errors and ensure the logical flow of the algorithm.

Code Debugging

1. Errors Identified:
   - Same issues as above regarding the recursive calls.
2. Breakpoints Needed:
   - Set breakpoints at the start of the doTowers method to trace the recursive calls and parameter values.
3. Steps to Fix Errors:
   - Remove increment and decrement operations in the recursive calls and ensure parameters are passed correctly.

Complete Executable Code (Fixed)

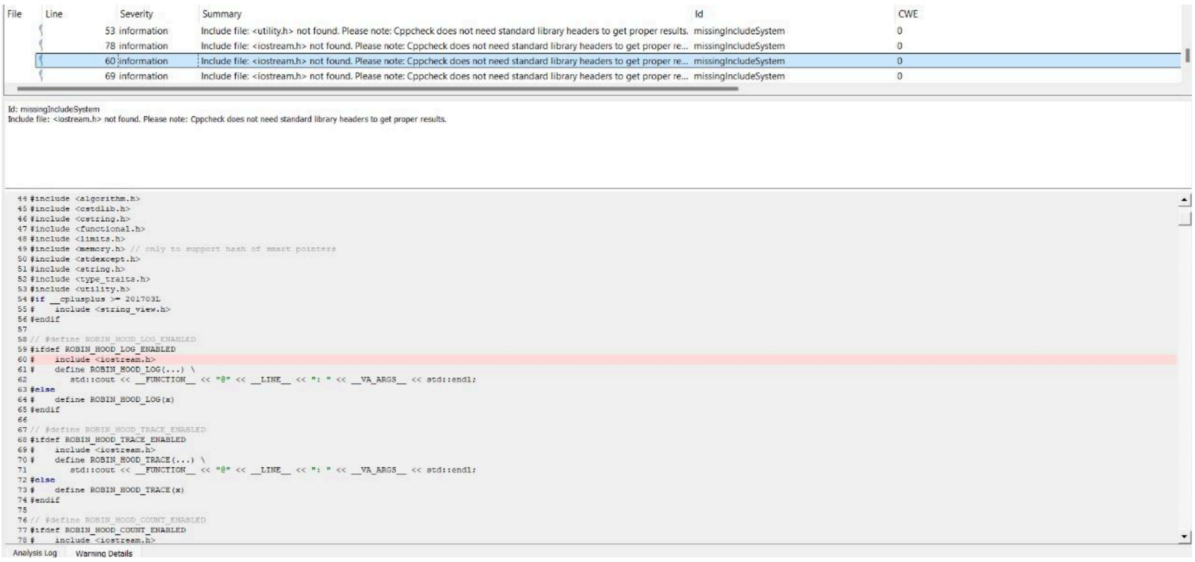
```
// Tower of Hanoi
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3;
        doTowers(nDisks, 'A', 'B', 'C');
    }

    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); // Fixed recursion
            System.out.println("Disk " + topN + " from " + from + " to
" + to);
            doTowers(topN - 1, inter, from, to); // Fixed recursion
        }
    }
}
```

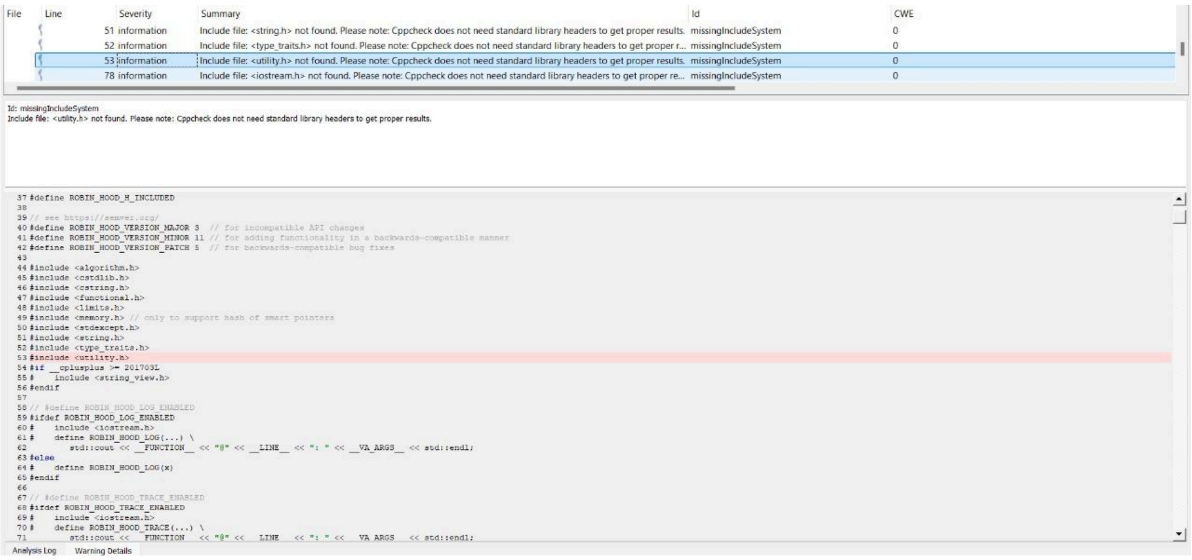
Static Analysis Tools

Code Link:

[robin-hood-hashing/src/include/robin_hood.h at master · martinus/robin-hood-hashing · GitHub]

JPEG Results:





File	Line	Severity	Summary	Id	CWE
	49	information	Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	50	information	Include file: <stdexcept.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
33 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
34 // SOFTWARE.
35
36 #ifndef ROBIN_HOOD_H_INCLUDED
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
```