IT - 314 Software Engineering

Lab - 7 Program Inspection, Debugging and Static Analysis

Name : Harsh Bosamiya

Student ID : 202201243

Lab Group : 3

TASK - 1 PROGRAM INSPECTION

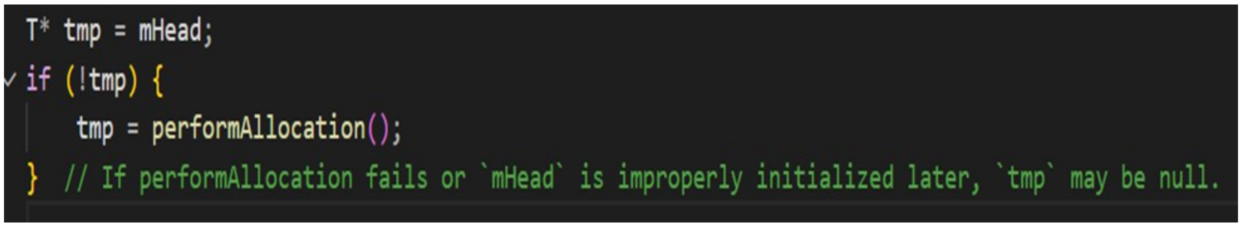
Github Link :

https://github.com/martinus/robin-hood-hashing/blob/master/src/include/robin_hood.h

1.How many errors are there in the program? Mention the errors you have identified.

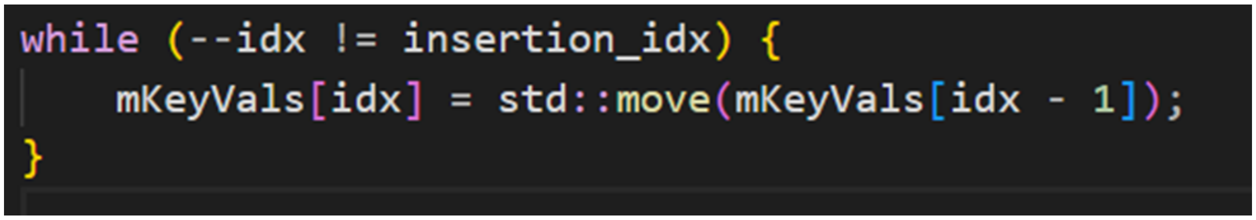
Category A: Data Reference Errors:

1.Uninitialized Variables:
mHead and mListForFree: Initialized to nullptr but not always reset after memory deallocation, leading to potential dangling pointers or uninitialized access.



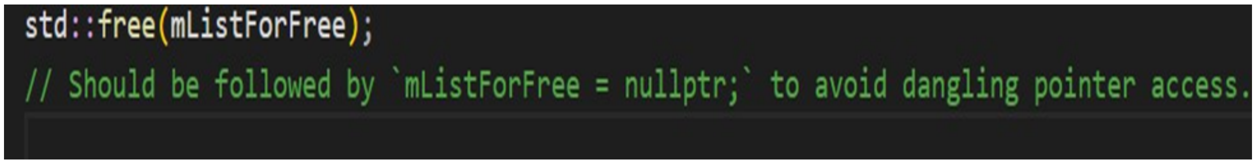
```
T* tmp = mHead;
if (!tmp) {
    tmp = performAllocation();
} // If performAllocation fails or `mHead` is improperly initialized later, `tmp` may be null.
```

2. Array Bound Violations:
shiftUp and shiftDown operations: No checks ensure that the index is within the array bounds.



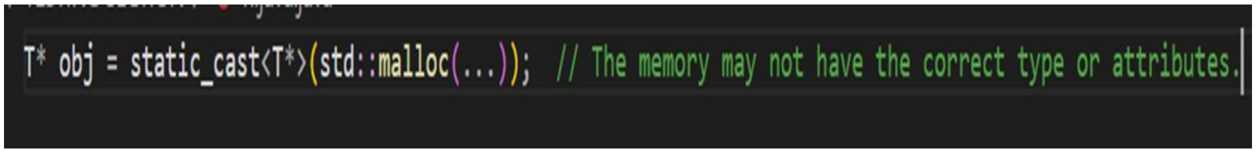
```
while (--idx != insertion_idx) {
    mKeyVals[idx] = std::move(mKeyVals[idx - 1]);
}
```

3. Dangling Pointers:
In BulkPoolAllocator: The reset() method frees memory but does not reset the pointer to nullptr.



```
std::free(mListForFree);
// Should be followed by `mListForFree = nullptr;` to avoid dangling pointer access.
```

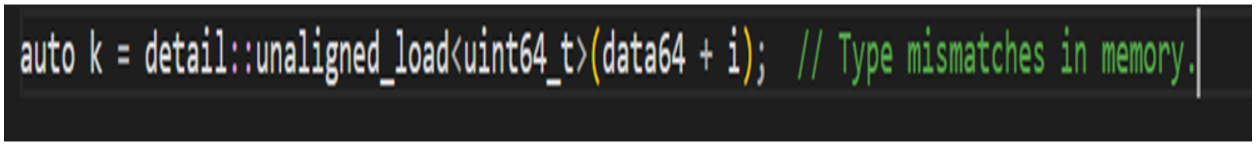
4. Incorrect Casts in reinterpret_cast_no_cast_align_warning:
Casting memory regions without validating types or attributes can lead to subtle bugs.



```
T* obj = static_cast<T*>(std::malloc(...));  // The memory may not have the correct type or attributes.
```

Category B: Data-Declaration Errors:

1.Potential Data Type Mismatches:
Casting in hash_bytes: Hashing operations involve multiple castings between data types. If the size or attributes of the data types differ, unexpected behavior can arise.

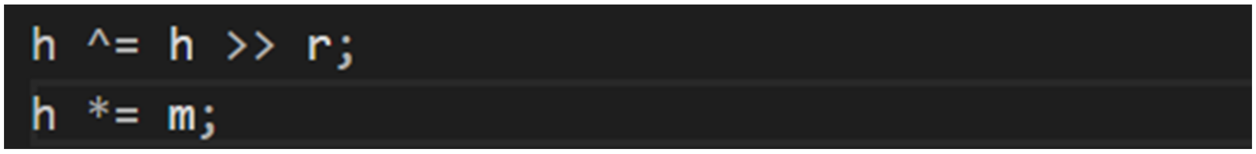


```
auto k = detail::unaligned_load<uint64_t>(data64 + i);  // Type mismatches in memory.
```

2.Similar Variable Names:
Confusion between similarly named variables: Variables like mHead, mListForFree, and mKeyVals are similar in naming, which could cause confusion during modification or debugging.

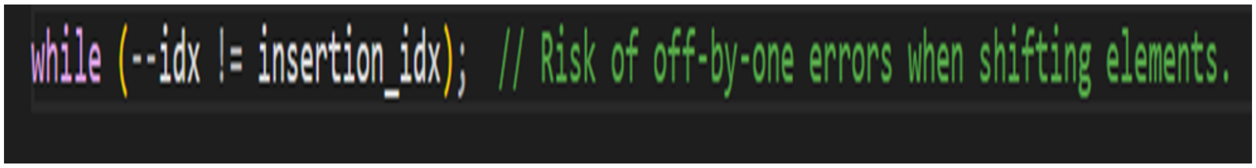
Category C: Computation Errors:

1.Integer Overflow:
Hash Computations in hash_bytes: The hash function performs multiple shifts and multiplications on large integers, potentially leading to overflow if the result exceeds



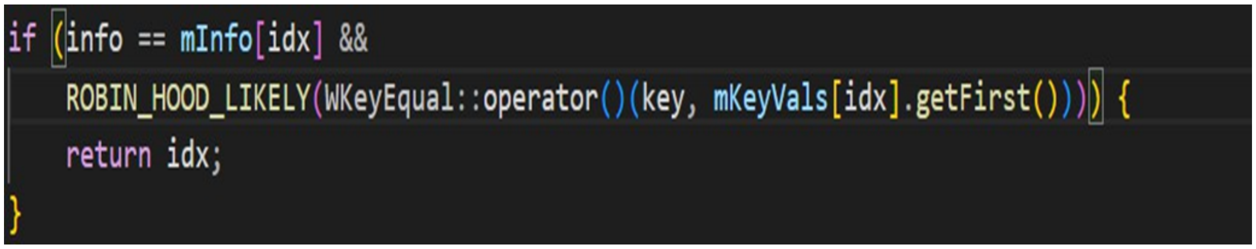
```
h ^= h >> r;
h *= m;
```

2.Off-by-One Errors:
Loop Indexing in shiftUp and shiftDown: The loop conditions may result in off-by-one errors, especially if the size of the data structure is mismanaged



```
while (--idx != insertion_idx);  // Risk of off-by-one errors when shifting elements.
```

Category D: Comparison Errors:
1.Incorrect Boolean Comparisons:
In conditions where multiple logical operations are combined, such as in findIdx, improper handling of && and || could lead to incorrect evaluations.

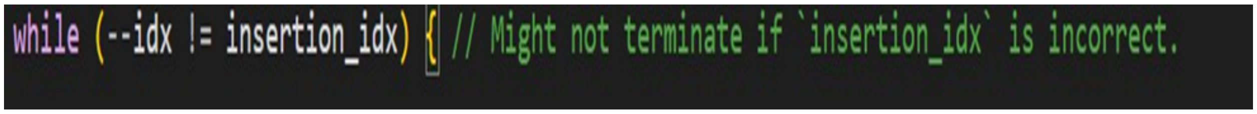


```
if (info == mInfo[idx] &&
    ROBIN_HOOD_LIKELY(WKeyEqual::operator()(key, mKeyVals[idx].getFirst()))) {
    return idx;
}
```

2.Mixed Comparisons:
In some cases, different types (e.g., signed and unsigned integers) are compared, which could lead to incorrect outcomes depending on the system/compiler.

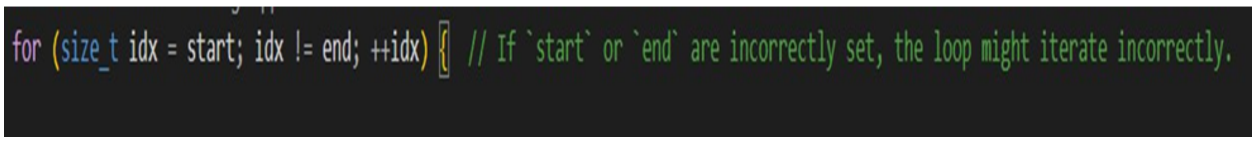
Category E: Control-Flow Errors:

1.Potential Infinite Loop:
Unterminated Loops: In loops like shiftUp and shiftDown, there is a risk of the loop not terminating correctly if the termination condition is never met.



```
while (--idx != insertion_idx) { // Might not terminate if `insertion_idx` is incorrect.
```

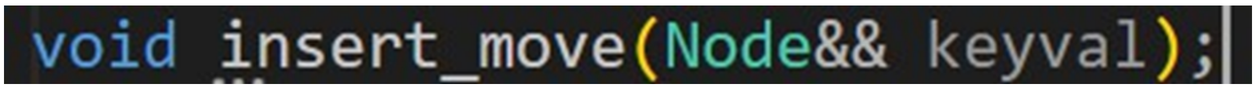
2.Unnecessary Loop Executions:
In some cases, loops might execute one extra time or fail to execute due to incorrect initialization or condition checks.



```
for (size_t idx = start; idx != end; ++idx) { // If `start` or `end` are incorrectly set, the loop might iterate incorrectly.
```

Category F: Interface Errors:

1.Mismatched Parameter Attributes:
Function Calls: There is potential for parameter mismatch in functions like insert_move. The arguments passed to these functions might not match the expected attributes (e.g., data type, size).



```
void insert_move(Node&& keyval);
```

2.Global Variables:
Global variables in different functions: If the same global variable is referenced across different functions or procedures, care must be taken that they are used consistently and initialized properly. This is not explicitly seen but could be a potential error source in expansions of the code

Category G: Input/Output Errors:

1.Missing File Handling:

While the code doesn't deal with files directly, any extension that includes I/O might introduce typical file handling errors such as unclosed files, failure to check for end-of-file conditions, or improper error handling

2.Which category of program inspection would you find more effective?

Category A: Data Reference Errors is the most effective in this case because of the use of manual memory management, pointers, and dynamic data structures. Since errors in pointer dereferencing and memory allocation/deallocation can easily lead to critical issues like crashes, segmentation faults, or memory leaks, focusing on this category is vital. Other important categories are Computation Errors and Control-
Flow Errors, especially for large projects.

3.Which type of error are you not able to identify using the program inspection?

Concurrency Issues: The inspection does not account for multi-threading or concurrency-related issues, such as race conditions or deadlocks. If this program were expanded to handle multiple threads, issues related to shared resources, locks, and thread safety would need to be addressed.
Dynamic Errors: Some errors, such as those related to memory overflow, underflow, or runtime environment behaviour, may not be caught until the code is executed in a real-world scenario.

4.Is the program inspection technique worth applying?

Yes, the program inspection technique is valuable, particularly for detecting static errors that might not be caught by compilers, such as pointer mismanagement, array bound violations, and improper control flow. Although it may not catch every dynamic issue or concurrency-related bug, it's an essential step to ensure code quality, especially in memory-critical applications like this C++ implementation of hash tables. This approach improves the code's reliability and helps maintain best practices in memory handling, control flow, and computational logic.

TASK – 2 CODE DEBUGGING

· Code debugging for the given text files(in Java)

1. Armstrong

1) **How many errors are there in the program? Mention the errors you have identified**.

· **Logic error in calculating remainder:** In line remainder = num / 10;, it incorrectly calculates the remainder by dividing the number by 10 instead of finding the last digit of the number.

· **Logic error in reducing the number:** In line num = num % 10;, it should be updating num by removing the last digit using integer division by 10 instead of modulus operation.

· **Incorrect input number handling in the loop:** The current logic of the loop doesn't work as expected since it doesn't correctly traverse the digits of the number.

2) How many breakpoints you need to fix those errors?

· Fix the logic to correctly calculate the remainder (remainder = num % 10).

· Fix the logic to reduce the number by removing the last digit (num = num / 10).

a) What are the steps you have taken to fix the error you identified in the code fragment?

· Step 1: Change remainder = num / 10; to remainder = num % 10;. This finds the correct last digit of the number.

· Step 2: Change num = num % 10; to num = num / 10;. This properly removes the last digit from the number after processing it.

3) The corrected executable of the code is as follows:

```
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;

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

            num = num / 10;

        }


        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

1. How many errors are there in the program? Mention the errors you have identified.

Logical error in the GCD method:

- The condition in the while loop is incorrect. The condition should be while(a % b != 0) instead of while(a % b == 0). This is required for computing the greatest common divisor using the Euclidean algorithm.

Logical error in the LCM method:

- The condition if(a % x != 0 && a % y != 0) is incorrect. It should be if(a % x == 0 && a % y == 0) since you are looking for a number that is divisible by both x and y to find the least common multiple.

Inefficiency in the LCM calculation:

- The LCM can be calculated using the formula lcm(x, y) = (x * y) / gcd(x, y). The current implementation with a loop is unnecessarily complex and inefficient.

2. How many breakpoints do you need to fix those errors?

Place a breakpoint inside the gcd() function to check if the while condition and values of a and b are being updated correctly.

Place another breakpoint inside the lcm() function to ensure the correct condition is being checked in the if statement.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Fix in the GCD method: Change the while(a % b == 0) condition to while(a % b != 0) to correctly implement the Euclidean algorithm.

Fix in the LCM method: Change the condition in the if statement from if(a % x != 0 && a % y != 0) to if(a % x == 0 && a % y == 0).

Improve LCM calculation: Replace the loop with the formula lcm = (x * y) / gcd(x, y) to make it more efficient.

3. The corrected executable of the code is as follows:

```java
import java.util.Scanner;


public class GCD_LCM

{

    static int gcd(int x, int y)

    {

        int r = 0;

        int a = (x > y) ? x : y;

        int b = (x > y) ? y : x;


        while (b != 0)

        {

            r = a % b;

            a = b;

            b = r;

        }

        return a;

    }


    static int lcm(int x, int y)

    {

        return (x * y) / gcd(x, y);

    }
```

```
  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 the two numbers is: " + gcd(x, y));

    System.out.println("The LCM of the two numbers is: " + lcm(x, y));

    input.close();

  }

}
```

3. Knapsack

1. How many errors are there in the program? Mention the errors you have identified.

Increment Error in Line int option1 = opt[n++][w];:

- The n++ is incorrect. It increments n before accessing the value, which changes the logic of the loop. It should be opt[n][w] instead of opt[n++][w].

Wrong Indexing in Line option2 = profit[n-2] + opt[n-1][w-weight[n]];:

- The profit[n-2] is incorrect. It should be profit[n], and the check if (weight[n] > w) should be if (weight[n] <= w) because it is testing whether the item can fit.

2. How many breakpoints do you need to fix those errors?

Breakpoint 1: On line int option1 = opt[n++][w]; to identify the incorrect use of n++.

Breakpoint 2: On line if (weight[n] > w) to check the condition on weights and indexing in option2.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Replace opt[n++][w] with opt[n][w].

Change if (weight[n] > w) to if (weight[n] <= w).

Update option2 to correctly reference the profit[n].

3. The corrected executable of the code is as follows:

```
public class Knapsack {


  public static void main(String[] args) {

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

    int W = Integer.parseInt(args[1]);


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

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


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

      profit[n] = (int) (Math.random() * 1000);

      weight[n] = (int) (Math.random() * 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++) {

    int option1 = opt[n][w];

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

    opt[n][w] = Math.max(option1, option2);
    sol[n][w] = (option2 > option1);
  }
}


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;                   }
}


\
        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. How many errors are there in the program? Mention the errors you have identified.

In the inner while loop condition, sum == 0 is incorrect. It should be sum != 0.

The calculation inside the loop s = s * (sum / 10) is wrong. The operation should be a summation, not multiplication. The correct operation is s += sum % 10 to sum the digits.

sum = sum % 10; is incorrect. It should be sum /= 10; to divide the number by 10, reducing it in each iteration.

Missing semicolon after sum = sum % 10.

The logic to sum the digits is incorrect; instead of multiplying and dividing the digits, the sum of the digits should be calculated using s += sum % 10.

2. How many breakpoints do you need to fix those errors?

One at the start of the outer while loop to check the value of num and ensure that the loop works correctly when num > 9.

Another inside the inner while loop to inspect how s and sum are changing in each iteration to ensure the summation of digits is correct.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Corrected the while(sum != 0) condition to properly sum the digits.

Replaced the incorrect calculation s = s * (sum / 10) with the correct summation logic s += sum % 10.

Corrected the division logic with sum /= 10 to reduce the sum correctly in each loop iteration.

Fixed the missing semicolon.

3. The corrected executable of the code is as follows:

```
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 num = n;


        while(num > 9) {

            int sum = 0;

            while(num != 0) {

                sum += num % 10;

                num /= 10;

            }

            num = sum;

        }
    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. How many errors are there in the program? Mention the errors you have identified.

Error 1: The line int option1 = opt[n++][w]; incorrectly increments n. It should simply reference opt[n][w] without modifying n. The correct statement should be int option1 = opt[n-1][w];.

Error 2: In the line if (weight[n] > w), the condition logic is incorrect. It should be if (weight[n] <= w) to ensure the item is considered if its weight is less than or equal to the remaining capacity.

Error 3: In the line option2 = profit[n-2] + opt[n-1][w-weight[n]], the array index profit[n-2] is incorrect because it should be profit[n] (using the current item's profit, not a prior one). The corrected line should be option2 = profit[n] + opt[n-1][w-weight[n]].

2. How many breakpoints do you need to fix those errors?

- A breakpoint before the start of the nested loop (on the line for (int n = 1; n <= N; n++) {), to verify if the profit and weight arrays are initialized correctly.

- A breakpoint inside the nested loop, particularly before and after the statement opt[n][w] = Math.max(option1, option2);, to check the correctness of option1 and option2.
- A breakpoint in the section where the selected items are determined (at the line for (int n = N, w = W; n > 0; n--) {), to verify the item selection logic.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Step 1: Corrected the option1 calculation by changing int option1 = opt[n++][w]; to int option1 = opt[n-1][w]; to avoid the incorrect increment of n.

Step 2: Modified the condition if (weight[n] > w) to if (weight[n] <= w) to ensure that the item can only be included if its weight is less than or equal to the current capacity.

Step 3: Corrected the option2 calculation by replacing profit[n-2] with profit[n].

3. The corrected executable of the code is as follows:

```
public class Knapsack {


  public static void main(String[] args) {

    int N = Integer.parseInt(args[0]);         int W = Integer.parseInt(args[1]);


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

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


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

      profit[n] = (int) (Math.random() * 1000);

      weight[n] = (int) (Math.random() * W);

    }

    int[][] opt = new int[N+1][W+1];

    boolean[][] sol = new boolean[N+1][W+1];
```

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

        int option1 = opt[n-1][w];

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

        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}

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;                    }
}

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]);
}
```

```
  }
}
```

6. Multiply matrices

1. How many errors are there in the program? Mention the errors you have identified.

Array index out of bounds in multiplication logic: In the inner loop (line sum = sum + first[c-1][c-k]*second[k-1][k-d];), the array indices use c-1 and k-1, which could cause an out-of-bounds error for arrays. These indices need to be corrected.

Incorrect prompt messages: After entering the dimensions of the first matrix, the program asks for "Enter the number of rows and columns of the first matrix" again, when it should ask for the second matrix.

Logical issue in matrix multiplication loop: The matrix multiplication should involve iterating over the number of columns of the second matrix, not the rows. This affects how the product matrix is filled.

2. How many breakpoints do you need to fix those errors?

Breakpoint 1: Right before the matrix multiplication logic, to check how indices are accessed.

Breakpoint 2: After each iteration of the multiplication, to ensure the sum is being computed correctly.

a. What are the steps you have taken to fix the error you identified in the code fragment?

- **Fix the indexing error:** Modify the line to sum = sum + first[c][k] * second[k][d]; so that the indices are within bounds and correct.
- **Correct the input prompt:** Change the prompt to ask for the second matrix's dimensions properly.

- **Fix logic in multiplication:** Ensure that the loop iterates correctly over the columns and rows, ensuring the result is stored properly in the multiply matrix.

3. The corrected executable of the code is as follows:

```java
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 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;

          for (k = 0; k < n; k++) {

            sum += first[c][k] * second[k][d];

          }

          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.println();

        }

      }

    }

}
```

7. Quadratic Probing

1. How many errors are there in the program? Mention the errors you have identified.

Syntax Error in the insert method:
In the line i + = (i + h / h--) % maxSize;, the syntax i + = is incorrect. It should be i = (i + h * h++) % maxSize; to correctly calculate the next probing index with quadratic probing.

Logical Error in remove method rehash loop:
When rehashing in the remove method, currentSize--; is executed twice—once inside the loop and once after. This will incorrectly decrement the currentSize multiple times.

Print Message Error in QuadraticProbingHashTableTest:
The comment /** maxSizeake object of QuadraticProbingHashTable **/ contains a typo. It should be something like /** Make object of QuadraticProbingHashTable **/.

Improper exit from the loop in the get method:
The print statement System.out.println("i " + i); should be removed or adjusted, as it might be unnecessary for the final version of the code and can cause excessive output in the loop.

2. How many breakpoints do you need to fix those errors?

- A breakpoint at the insert method when calculating the quadratic probing index to ensure proper key insertion.
- A breakpoint at the rehashing loop in the remove method to check for the correct rehashing of elements.
- A breakpoint at the get method to ensure proper key lookup without looping indefinitely.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Fix the syntax error in the insert method by changing i + = to i =.

Correct the logical error in the remove method by removing the extra currentSize-- in the rehashing loop.

Fix the typo in the comment in the test class.

Remove the unnecessary print statement in the get method or modify it for debugging purposes only.

3. The corrected executable of the code is as follows:

```
import java.util.Scanner;



class QuadraticProbingHashTable {

   private int currentSize, maxSize;

   private String[] keys;

   private String[] vals;



   public QuadraticProbingHashTable(int capacity) {
       currentSize = 0;

       maxSize = capacity;

       keys = new String[maxSize];

       vals = new String[maxSize];

   }


   public void makeEmpty() {

       currentSize = 0;

       keys = new String[maxSize];

       vals = new String[maxSize];

   }


   public int getSize() {

       return currentSize;

   }


   public boolean isFull() {

       return currentSize == maxSize;

   }


   public boolean isEmpty() {

       return getSize() == 0;

   }
```

```
public boolean contains(String key) {

    return get(key) != null;

}



private int hash(String key) {

    return key.hashCode() % maxSize;

}



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;

        } while (i != tmp);

    }


    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;

        }

        return null;

    }



    public void remove(String key) {

        if (!contains(key))

            return;



        int i = hash(key), h = 1;

        while (!key.equals(keys[i]))

            i = (i + h * h++) % maxSize;

        keys[i] = vals[i] = null;
        i = (i + h * h++) % maxSize;

        while (keys[i] != null) {

            String tmp1 = keys[i], tmp2 = vals[i];

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

            currentSize--;

            insert(tmp1, tmp2);

        }

        currentSize--;      }



    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();

    }

}


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;

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;

        }

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

- The class name Ascending _Order contains an underscore between Ascending and Order, which is not allowed in Java for class names. It should be corrected to AscendingOrder.
- In the outer for loop, the condition i >= n is incorrect. It should be i < n.
- The semicolon ; after the for loop for (int i = 0; i >= n; i++); ends the loop prematurely, causing incorrect logic execution.
- In the nested if condition, the comparison a[i] <= a[j] is wrong for ascending order. It should be a[i] > a[j].
- There's a potential issue with the array printing logic. The loop for (int i = 0; i < n - 1; i++) correctly avoids printing a trailing comma, but it's not very readable.

2. How many breakpoints do you need to fix those errors?

One at the class definition.

One at the outer for loop.

One at the inner if statement for comparison logic.

One for ensuring proper output formatting

a. What are the steps you have taken to fix the error you identified in the code fragment?

Renamed the class to AscendingOrder.

Corrected the outer loop condition from i >= n to i < n.

Removed the semicolon after the outer for loop to ensure proper block execution.

Changed the comparison in the if statement to a[i] > a[j] for sorting in ascending order.

3. The corrected executable of the code is as follows:

```
import java.util.Scanner;
```

```java
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();
        }
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    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]);

  }

}
```

9. Stack Implementation

1. How many errors are there in the program? Mention the errors you have identified.

Push Method Logic Error: The push method decreases top before assigning the value, which causes an ArrayIndexOutOfBoundsException. It should increment top instead.

top++;

stack[top] = value;

Display Method Logic Error: The loop condition in the display method is incorrect. It should iterate from 0 to top inclusive, using < instead of >.

for (int i = 0; i <= top; i++) {

Pop Method Logic Error: The pop method should only increase top if it is not empty, which is correct, but it should return the popped value. Also, if the stack is empty, it should ideally throw an exception or handle it appropriately.

Stack Overflow Check: The push method does check for a full stack, which is correct, but it could throw an exception instead of just printing a message.

2. How many breakpoints do you need to fix those errors?

To fix the identified errors, you would need to set breakpoints at the following locations:

- Inside the push method.
- Inside the pop method.
- Inside the display method.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Fix the push method to correctly increment top before pushing a value.

Fix the display method loop condition to correctly iterate through the stack elements.

Fix the pop method to return the popped value and manage the empty stack scenario better.

3. The corrected executable of the code is as follows:

```
import java.util.Arrays;


public class StackMethods {

    private int top;

    private int size;

    private 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;

    }

}


public int pop() {

    if (!isEmpty()) {

        int poppedValue = stack[top];

        top--;

        return poppedValue;

    } else {

        System.out.println("Can't pop...stack is empty");

        return -1;

    }

}


public boolean isEmpty() {

    return top == -1;

}


public void display() {

    if (isEmpty()) {

        System.out.println("Stack is empty");

        return;
        }
        for (int i = 0; i <= top; i++) {
            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.display();
    }
```

```
}
```

10. Tower of Hanoi

1. How many errors are there in the program? Mention the errors you have identified.

Incorrect Increment/Decrement in Recursive Call:

- doTowers(topN ++, inter--, from+1, to+1) is incorrect.
- The ++ and -- operators are used incorrectly. They should not be applied directly in this context. The parameters should simply be topN - 1, inter, from, and to.

Missing Base Case for Recursive Calls:

- The doTowers function should use the same parameters for the recursive calls but with correct values. Specifically, you should not alter from and to directly as you are doing with from + 1 and to + 1.

Incorrect Printing Logic:

- The logic for printing the disk movements does not align properly when the function is called recursively.

2. How many breakpoints do you need to fix those errors?

You will need at least **two breakpoints** to effectively debug and fix the identified errors:

1. **Breakpoint before the recursive call in doTowers** to inspect the parameters being passed.
2. **Breakpoint at the base case** to ensure that the logic for moving disks is correct.

a. What are the steps you have taken to fix the error you identified in the code fragment?

Correct the recursive call in doTowers:

- Change doTowers(topN ++, inter--, from+1, to+1) to doTowers(topN - 1, inter, to, from) for the second recursive call.

Ensure that the parameters passed are correct and that they reflect the intended state of the Towers of Hanoi problem.

Adjust the print statements if necessary to ensure clarity.

3. The corrected executable of the code is as follows:

```
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);

         System.out.println("Disk " + topN + " from " + from + " to " + to);

         doTowers(topN - 1, inter, from, to);

      }

   }

}
```