Software Enginnering (IT314)

LAB 8

Kishan Pansuriya (202201504)

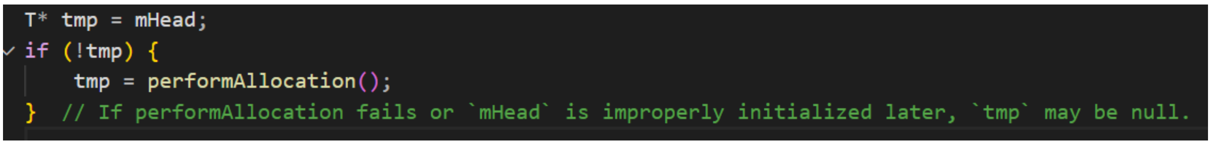
Task 1: PROGRAM INSPECTION

GitHub Code Link:

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

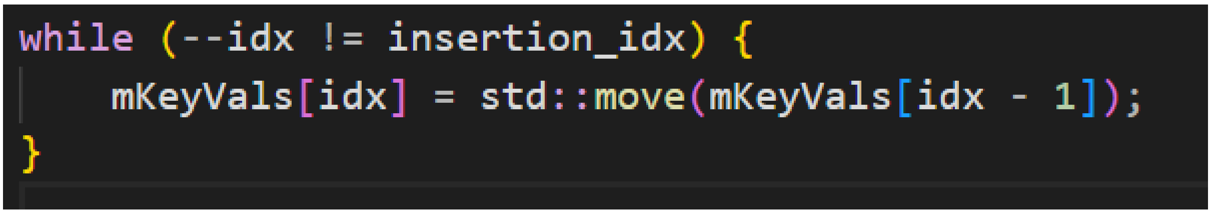
Category A: Data Reference Errors

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



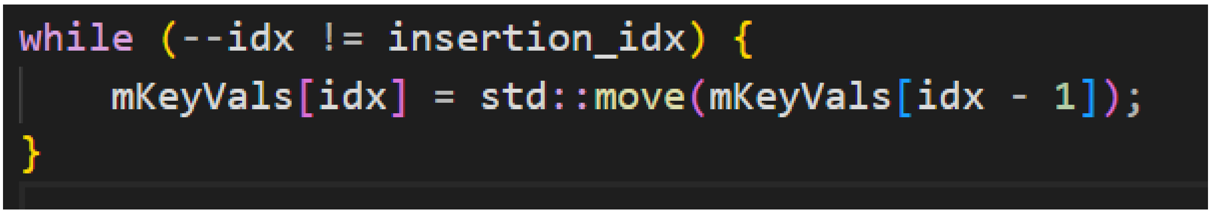
```
T* tmp = mHead;
if (!tmp) {
    tmp = performAllocation();
}  // If performAllocation fails or `mHead` is improperly initialized later, `tmp` may be null.
```

2. Array Bound Violations:
   - shiftUp and shiftDown Operations: No checks to ensure that the indices are within array bounds, risking access to invalid memory.



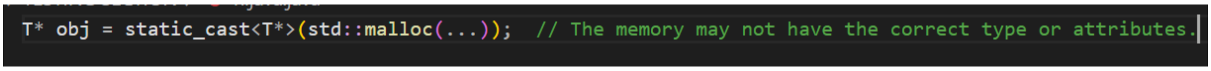
```
while (--idx != insertion_idx) {
    mKeyVals[idx] = std::move(mKeyVals[idx - 1]);
}
```

3. Dangling Pointers:
   - BulkPoolAllocator: The reset() method frees memory but does not reset pointers to nullptr, risking access to deallocated memory.



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

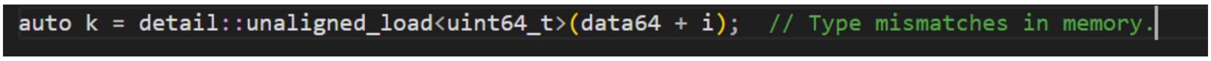
4. Type Mismatches:
   - Incorrect Casts in reinterpret_cast_no_cast_align_warning: Casting memory regions without validating types or attributes may lead to subtle bugs.



```
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: Multiple castings between data types in hashing operations may lead to unexpected behavior due to differing sizes or attributes.

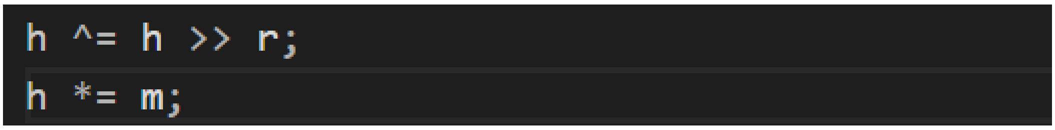


```
auto k = detail::unaligned_load<uint64_t>(data64 + i);  // Type mismatches in memory.
```

2. Similar Variable Names:
   - Confusing Similarities: Variables like mHead, mListForFree, and mKeyVals have similar names, potentially causing confusion during modifications or debugging.

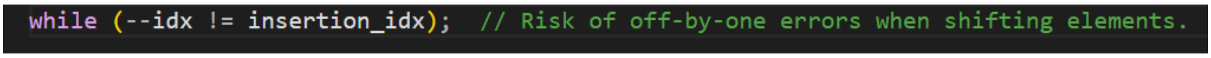
Category C: Computation Errors

1. Integer Overflow:
   - Hash Computations in hash_bytes: The hash function performs shifts and multiplications on large integers, which may cause overflow if the result exceeds the maximum integer size.



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

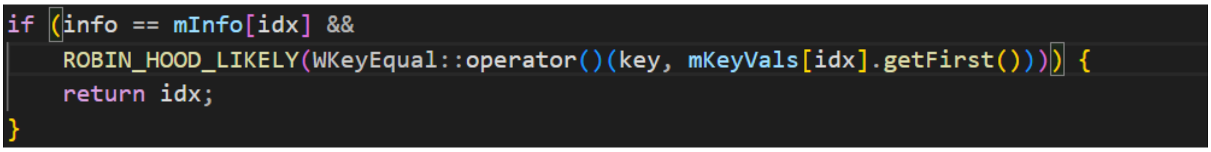
2. Off-by-One Errors:
   - Loop Indexing in shiftUp and shiftDown: The loop conditions might lead to off-by-one errors, particularly if the data structure size is mismanaged.



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

Category D: Comparison Errors

1. Incorrect Boolean Comparisons:
   - Improper Logical Operations: In conditions like those in findIdx, the mishandling of && and || may lead to incorrect evaluations.

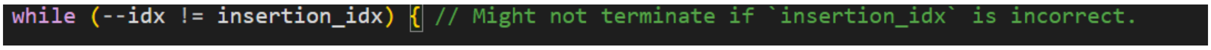


```
if (info == mInfo[idx] &&
    ROBIN_HOOD_LIKELY(WKeyEqual::operator()(key, mKeyVals[idx].getFirst()))) {
    return idx;
}
```

2. Mixed Comparisons:
   - Signed vs. Unsigned Integers: Comparing different types can lead to incorrect outcomes depending on the system or compiler behavior.

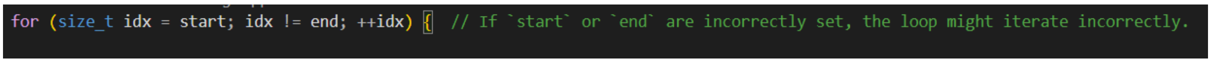
Category E: Control-Flow Errors

1. Potential Infinite Loop:
   - Unterminated Loops: Loops in shiftUp and shiftDown might not terminate correctly if the termination conditions are never satisfied.



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

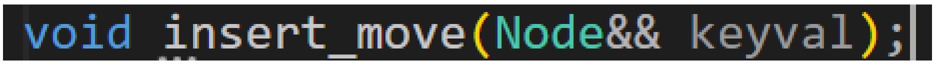
2. Unnecessary Loop Executions:
   - Execution Issues: Some loops may execute one extra time or fail to run due to incorrect initialization or condition checks.



```
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: Functions like insert_move may have mismatched parameters in terms of expected attributes (e.g., data type, size).



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

2. Global Variables:

- Potential Issues with Global State: The use of global variables across different functions necessitates careful initialization and consistency, which may lead to errors if expanded.

Category G: Input/Output Errors

1. Missing File Handling:
   - Potential I/O Errors: Future extensions involving I/O might introduce common file handling errors, including unclosed files or inadequate end-of-file checks.

Task 2: CODE DEBUGGING

1. Armstrong

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

There are **two errors**:

1. Remainder is incorrectly calculated as remainder = num / 10; it should be remainder = num % 10.
2. The update of num is incorrect as num = num % 10; it should be num = num / 10.

2. How many breakpoints do you need to fix those errors?
I need **two breakpoints**:

- One at the line where the remainder is calculated.
- One at the line where num is updated.

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

1. Changed the remainder calculation to use the modulus operator: remainder = num % 10.
2. Corrected the update of num to use division: num = num / 10.

Modified Code

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // store original number
```

```
            int check = 0, remainder;
            while (num > 0) {
                remainder = num % 10;  // Corrected: Get last digit
                check = check + (int) Math.pow(remainder, 3);
                num = num / 10;  // Corrected: Remove last digit
            }
            if (check == n)
                System.out.println(n + " is an Armstrong Number");
            else
                System.out.println(n + " is not an Armstrong Number");
        }
    }
```

2. LCM and GCD

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

There are **two errors**:

1. In the gcd function, the condition while(a % b == 0) is incorrect; it should be while(a % b != 0) to correctly compute the GCD.
2. In the lcm function, the condition if(a % x != 0 && a % y != 0) should be if(a % x == 0 && a % y == 0) to check divisibility for both numbers and calculate the LCM.

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

I need **two breakpoints**:

- One at the while condition in the gcd function.
- One at the if condition in the lcm function.

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

1. Fixed the while condition in gcd to while(a % b != 0).
2. Corrected the if condition in lcm to if(a % x == 0 && a % y == 0).

Modified Code

```
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 larger number

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

3. Knapsack

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

There are **two errors**:

1. In the opt[n++][w] line, n++ should be replaced with n-1 because we want to refer to the previous item.
2. In the condition if (weight[n] > w), it should be if (weight[n] <= w) to correctly handle when the item can fit in the knapsack. Also, in option2, profit[n-2] should be profit[n].

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

I need **two breakpoints**:

- One at the line opt[n++][w] to check the correct index calculation.
- One at the condition if (weight[n] > w) to verify proper weight comparison.

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

1. Changed opt[n++][w] to opt[n-1][w].
2. Modified if (weight[n] > w) to if (weight[n] <= w) and fixed option2 to use profit[n].

Modified 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-1][w];  // Fixed index issue

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w)  // Fixed condition
                    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. How many errors are there in the program? Mention the errors you have identified.
There are **three errors**:

1. The condition in the inner while loop while(sum==0) should be while(sum > 0) to iterate while there are digits left in sum.
2. The calculation in the inner loop s=s*(sum/10); should be s += sum % 10; to sum the digits properly.
3. The line sum=sum%10 is missing a semicolon at the end.

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

I need **three breakpoints**:

- One at the inner while loop condition while(sum == 0).
- One at the digit summation line s=s*(sum/10);.
- One at the sum=sum%10 line to check for the missing semicolon.

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

1. Changed while(sum == 0) to while(sum > 0).
2. Updated s = s * (sum / 10); to s += sum % 10;.
3. Added a semicolon at the end of sum = sum % 10;.

Modified Code:

```
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) {  // Fixed condition
                s += sum % 10;  // Fixed summation
                sum = sum / 10;  // Fixed to update sum correctly
            }
            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. How many errors are there in the program? Mention the errors you have identified.

There are **four errors**:

1. The calls to leftHalf(array + 1) and rightHalf(array - 1) are incorrect; they should pass the actual subarrays, not modify the array pointer. They should be leftHalf(Arrays.copyOfRange(array, 0, mid)) and rightHalf(Arrays.copyOfRange(array, mid, array.length)) respectively.
2. The merging call merge(array, left++, right--); is incorrect; left++ and right-- are not valid for array parameters. It should just be merge(array, left, right);.
3. The midpoint calculation is missing; it should be defined as int mid = array.length / 2;.
4. The merge method needs to accept the result array, which is the original array being sorted.

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

I need **four breakpoints**:

- One at the line where the array is split to check the parameters being passed to leftHalf and rightHalf.
- One at the line where merge is called to check the parameters being passed.
- One at the line where the midpoint should be calculated to ensure it is done correctly.
- One in the merge function to check the merging logic.

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

1. Added the midpoint calculation: int mid = array.length / 2;.
2. Changed leftHalf(array + 1) to leftHalf(Arrays.copyOfRange(array, 0, mid)).
3. Changed rightHalf(array - 1) to rightHalf(Arrays.copyOfRange(array, mid, array.length)).
4. Updated the merge call to merge(array, left, right);.

Modified:

```
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 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);
        }
    }

    // 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. Matrix Multiplication

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

There are **three errors** in the program:

1. **Incorrect Matrix Element Access**: The multiplication loop uses incorrect indices, leading to potential ArrayIndexOutOfBoundsException.
2. **Uninitialized Variables**: The variable sum is not reset appropriately before each product calculation.
3. **Incorrect Output Prompt**: The prompt for the second matrix mistakenly states "first matrix" instead of "second matrix."

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

I need **three breakpoints**:

1. One at the element access in the multiplication loop.
2. One at the output section to verify matrix dimensions.
3. One at the initialization of sum before each product calculation.

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

1. **Correct Element Access**: Change from first[c-1][c-k] to first[c][k] and from second[k-1][k-d] to second[k][d].
2. **Proper Initialization**: Reset sum = 0; at the start of the inner loop for each c and d.
3. **Correct Output Prompt**: Change the prompt for the second matrix to accurately reflect that it is asking for the second matrix.

<u>Modified Code</u>

```
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; // Reset sum for each cell
                    for (k = 0; k < n; k++) {
                        sum = 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.print("\n");
            }
        }
        in.close(); // Close the scanner to avoid resource leak
    }
}
```

7. Quadratic Probing

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

There are **seven errors** in the program:

1. **Syntax Error in Insertion**: In insert, the line i + = (i + h / h--) % maxSize; has an incorrect space in the operator.
2. **Incorrect Formula for Quadratic Probing**: In both insert and get, the formula for updating index i should use h * h instead of h++ (which modifies h after usage).
3. **Logical Error in remove Method**: The while condition should check for keys[i] != null instead of checking for key equality.
4. **Uninitialized Variables**: The hash method should handle negative hash values by adding maxSize to ensure the index is valid.
5. **Hash Function Division by Zero**: The hash function should be adjusted to handle cases where maxSize is 0.
6. **Scanner Not Closed**: The Scanner object should be closed to prevent resource leaks.

7. **Incorrectly Named Comments**: Change comments like /** maxSizeake object of QuadraticProbingHashTable **/ to correctly state the purpose.

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

You need **four breakpoints**:

1. One in the insert method to check the indexing logic.
2. One in the get method to inspect how keys are accessed.
3. One in the remove method to verify the deletion logic.
4. One after initializing QuadraticProbingHashTable to inspect hash calculations.

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

1. **Fix Syntax Error**: Correct the insertion line to i += (i + h) % maxSize;.
2. **Update Quadratic Probing Formula**: Change i = (i + h * h++) % maxSize; to i = (i + h * h) % maxSize; h++;.
3. **Correct Logical Flow in remove**: Update the while condition to while (keys[i] != null).
4. **Adjust Hash Function**: Ensure the hash function handles negative values and cases where maxSize is 0.
5. **Close Scanner**: Add scan.close(); at the end of the main method to prevent resource leaks.
6. **Fix Comment Syntax**: Correct any comments to reflect accurate descriptions.

<u>**Modified Code**</u>

```
/** 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) {
        int hashCode = key.hashCode() % maxSize;
        return (hashCode < 0) ? hashCode + maxSize : hashCode; //
Handle negative hash values
    }

    /** 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; // Corrected formula
            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 formula
            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]))
```

```
            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:");

        /** 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(); // Close scanner
    }
}


1
```

8. Sorting array

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

There are **four errors** in the program:

1. **Class Name Syntax**: The class name Ascending _Order contains a space, which is not allowed. It should be AscendingOrder.
2. **Loop Condition Mistake**: The condition in the first for loop should be i < n instead of i >= n, which prevents the loop from executing.
3. **Incorrect Sorting Logic**: The sorting condition in the inner loop uses <=, which will incorrectly sort the array in descending order instead of ascending order. It should be <.
4. **Output Formatting**: The final output loop prints a comma after every element, even the last one. The formatting should handle this correctly.

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

You need **three breakpoints**:

1. One before the first for loop to check the loop condition.
2. One inside the inner for loop to inspect the sorting logic.
3. One before printing the sorted array to verify the content of the array.

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

1. **Fix Class Name**: Rename Ascending _Order to AscendingOrder.
2. **Correct Loop Condition**: Change the first loop condition from i >= n to i < n.

3. **Update Sorting Condition**: Change the inner loop condition from if (a[i] <= a[j]) to if (a[i] > a[j]).
4. **Improve Output Formatting**: Update the printing logic to correctly format the output without trailing commas.

Modified Code

```
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();
        }

        // Sorting the array in ascending order
        for (int i = 0; i < n; i++) {  // Corrected loop condition
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Corrected sorting condition
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        System.out.print("Ascending Order: ");
        for (int i = 0; i < n; i++) { // Corrected to include all 
elements
            System.out.print(a[i]);
            if (i < n - 1) {
                System.out.print(","); // Print comma for all except 
last element
            }
        }
    }
}
```

9. Stack Implementation

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

There are **four errors** in the program:

1. **Push Method Logic**: In the push method, the line top--; should be top++; to correctly increment the top index before adding the new value to the stack.
2. **Pop Method Logic**: In the pop method, the line top++; should be top--; to decrement the top index when popping an element from the stack.
3. **Display Method Loop Condition**: In the display method, the loop condition i > top should be i <= top to correctly iterate through the elements in the stack.
4. **Incorrect Output Formatting**: The output of the display method will not print anything if the stack is empty or if the top index is -1.

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

You need **three breakpoints**:

1. One in the push method to inspect the value of top before and after the value is added.
2. One in the pop method to inspect the value of top before and after popping.
3. One in the display method to check the loop iteration and printed values.

3. What are the steps you have taken to fix the errors you identified in the code fragment?

1. **Correct the Push Logic**: Change top-- to top++ in the push method.
2. **Fix the Pop Logic**: Change top++ to top-- in the pop method.
3. **Update the Display Method Loop**: Change the loop condition from i > top to i <= top.
4. **Ensure Correct Output**: Make sure the stack displays correctly by iterating over the stack based on the top index.

Modified 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; // Stack is initially empty
    }

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

    public void pop() {
        if (!isEmpty()) {
            top--; // Corrected to decrement the top index
        } 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; // Return if the stack is empty
        }
        for (int i = 0; i <= top; i++) { // Corrected loop 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(); // Displays current stack elements
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop(); // Pops all elements

        newStack.display(); // Displays stack after pops
    }
}
```

10. Tower of Hanoi

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

There are **three errors** in the program:

1. **Increment/Decrement Operators**: The expressions topN ++ and inter-- should not be using increment and decrement operators in the recursive calls. Instead, they should be simply passing the unchanged topN and inter.
2. **Incorrect Arguments in Recursive Calls**: The arguments in the recursive call doTowers(topN ++, inter--, from+1, to+1) should correctly represent the parameters for the Tower of Hanoi algorithm. Specifically, from + 1 and to + 1 do not make sense because they should be passed as characters and not integers.
3. **Missing Case for topN == 1**: In the first if statement, while it correctly prints the move for the first disk, it does not return afterwards, leading to potential unexpected behavior.

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

1. One in the doTowers method to inspect the values of topN, from, inter, and to before and after the recursive calls.
2. One after the first if statement to check the flow of control when topN == 1.

3. What are the steps you have taken to fix the errors you identified in the code fragment?

1. **Remove Increment/Decrement Operators**: Change topN ++ and inter-- to simply use topN - 1 and inter as arguments.
2. **Correct Recursive Call Arguments**: Change the call from from + 1 and to + 1 to just from and to, as they should remain characters.
3. **Ensure Correct Flow Control**: Add a return; statement after printing the move for

the first disk to prevent further unnecessary processing.

Modified Code

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

   public static void doTowers(int topN, char from, char inter, char to)
 {
      if (topN == 1) {
         System.out.println("Disk 1 from " + from + " to " + to);
         return; // Added return to prevent further processing
      } else {
         doTowers(topN - 1, from, to, inter); // Move topN - 1 disks
         System.out.println("Disk " + topN + " from " + from + " to " +
 to); // Move the last disk
         doTowers(topN - 1, inter, from, to); // Move the disks from
 intermediate to destination
      }
   }
}
```