Name-Abhi Mehta
Reg No.-221080001

# Experiment-09

**Aim**: To implement a Hash Map and develop two implementations of a hash table, one using separate chaining and the other using open addressing with linear probing

**Theory**: The Abstract Hashmap class provides a blueprint for implementing hash maps. It contains abstract methods that need to be implemented by concrete subclasses.

**createTable():** This method is responsible for creating an initially empty table with a size equal to a designated capacity instance variable. The initial capacity is crucial for the performance of the hash map.

**bucketGet(h, k):** This method mimics the semantics of the public get method but is specific to a key k that is known to hash to a particular bucket h. In separate chaining, this might involve searching a linked list or another data structure within the bucket. In open addressing, it involves probing linearly through the array until the key is found.

**bucketPut(h, k, v):** This method mimics the semantics of the public put method but is specific to a key k that is known to hash to a particular bucket h. In both separate chaining and open addressing, this involves inserting the key-value pair into the appropriate bucket.

**bucketRemove(h, k):** This method mimics the semantics of the public remove method but is specific to a key k known to hash to a particular bucket h. This involves removing the key-value pair associated with the given key from the corresponding bucket.

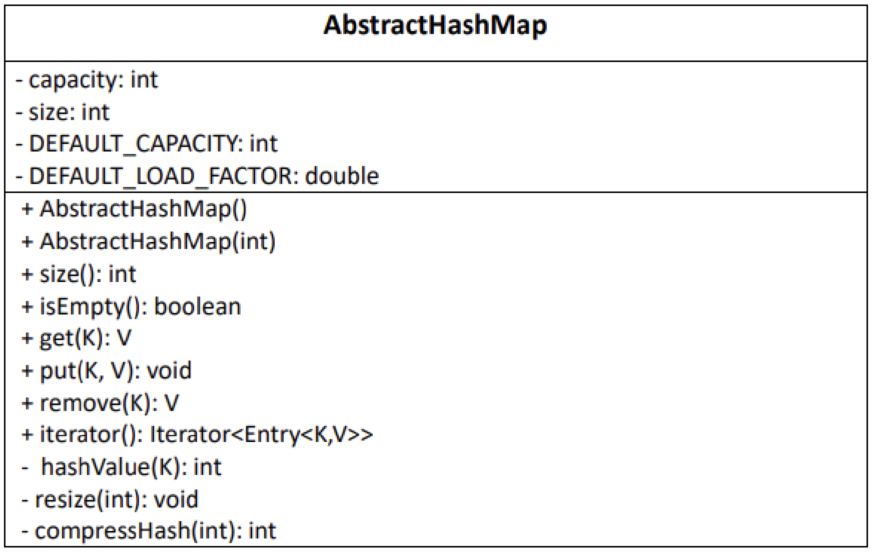
**entrySet():** This method is a standard map method that iterates through all entries of the map. In the case of separate chaining, it might involve iterating through each linked list in each bucket. For open addressing, it requires iterating through the entire array.

## Concrete Utility Methods

**hashValue(K key)**: This method is responsible for generating a hash code for a given key. It typically involves a hash compression function using a randomized MultiplyAdd-and-Divide (MAD) formula. The goal is to distribute keys uniformly across the array to minimize collisions.

**resize(int newCap)**: This method is responsible for automatically resizing the underlying hash table when the load factor (the ratio of the number of elements to the table size) reaches a certain threshold. Resizing involves creating a new table with a larger capacity and rehashing all the existing key-value pairs into the new table. This helps maintain a balanced trade-off between space and time complexity.

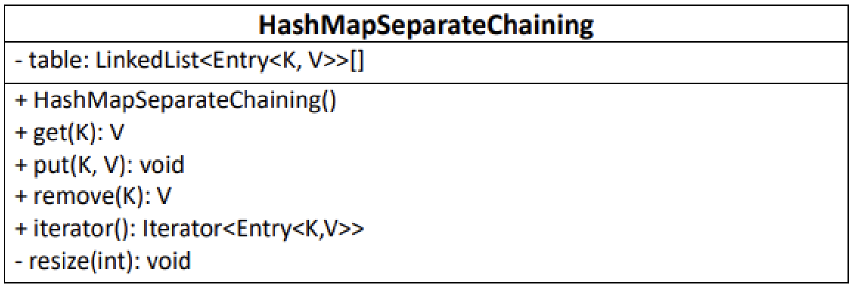
## Class Diagram:



**AbstractHashMap**

- capacity: int
- size: int
- DEFAULT_CAPACITY: int
- DEFAULT_LOAD_FACTOR: double

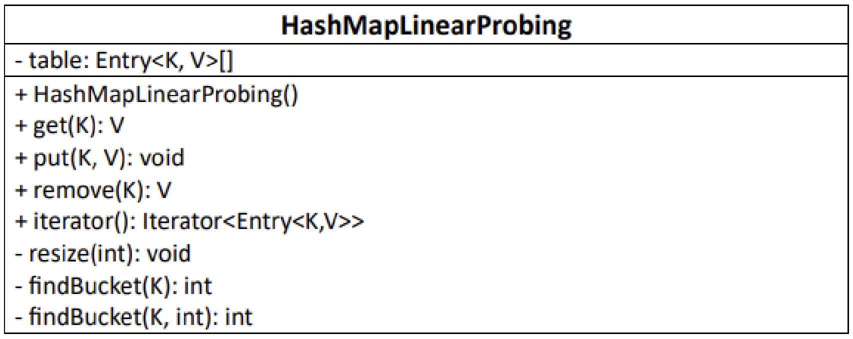
```
+ AbstractHashMap()
+ AbstractHashMap(int)
+ size(): int
+ isEmpty(): boolean
+ get(K): V
+ put(K, V): void
+ remove(K): V
+ iterator(): Iterator<Entry<K,V>>
-  hashValue(K): int
- resize(int): void
- compressHash(int): int
```



**HashMapSeparateChaining**

```
- table: LinkedList<Entry<K, V>>[]
```

```
+ HashMapSeparateChaining()
+ get(K): V
+ put(K, V): void
+ remove(K): V
+ iterator(): Iterator<Entry<K,V>>
- resize(int): void
```



**HashMapLinearProbing**

```
- table: Entry<K, V>[]
```

```
+ HashMapLinearProbing()
+ get(K): V
+ put(K, V): void
+ remove(K): V
+ iterator(): Iterator<Entry<K,V>>
- resize(int): void
- findBucket(K): int
- findBucket(K, int): int
```

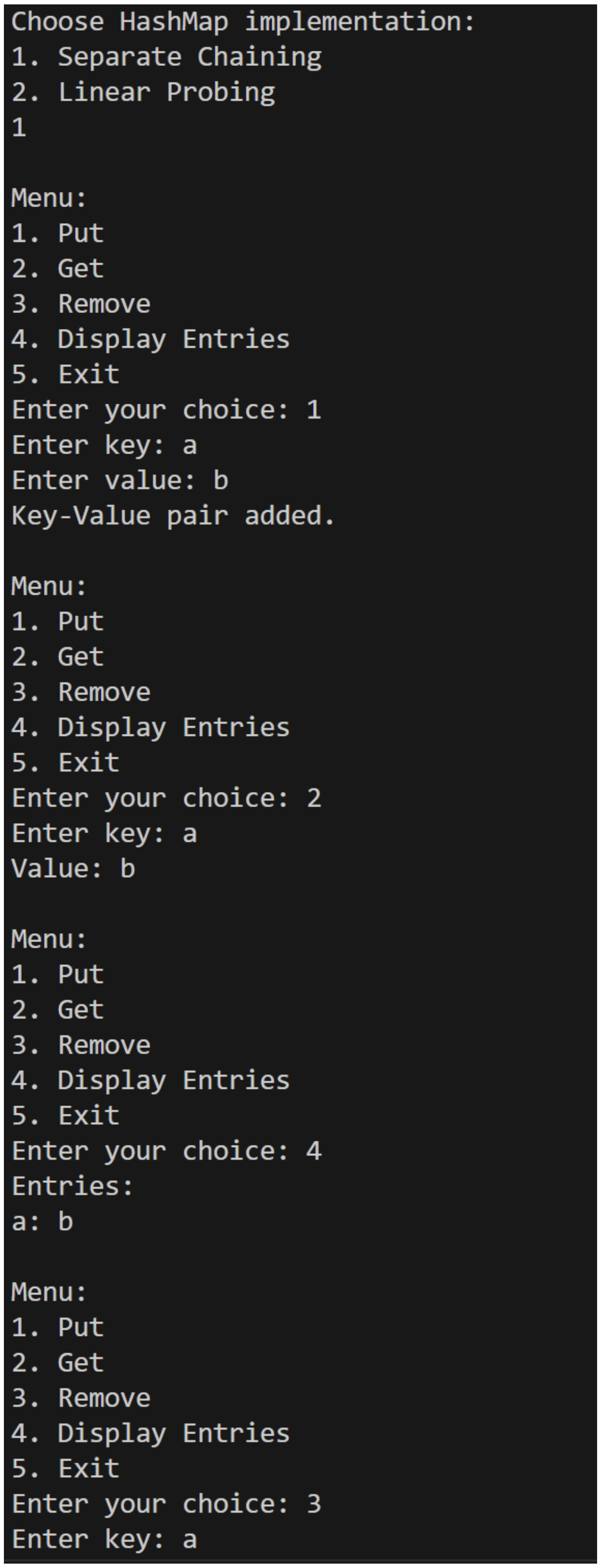
**Code:**

```
import java.util.*;
// Abstract base class for hash map implementation
abstract class AbstractHashMap<K, V> {
    protected int capacity; // Capacity of the hash map
    protected int size; // Number of elements in the hash map
    protected final double loadFactor; // Load factor for resizing
    // Constructor for initializing capacity and load factor
    public AbstractHashMap(int capacity, double loadFactor) {
        this.capacity = capacity;
        this.loadFactor = loadFactor;
        this.size = 0;
        createTable(); // Create the underlying data structure
    }

    // Abstract methods to be implemented by subclasses
    abstract void createTable(); // Create the underlying data structure
    abstract V bucketGet(int h, K k); // Get value from a specific bucket
    abstract void bucketPut(int h, K k, V v); // Put key-value pair in a specific bucket
    abstract V bucketRemove(int h, K k); // Remove key-value pair from a specific bucket
    abstract Set<Map.Entry<K, V>> entrySet(); // Get all entries in the hash map
    abstract int hashValue(K key); // Compute hash value for a key
    abstract void resize(int newCap); // Resize the hash map
    abstract void put(K key, V value); // Put key-value pair in the hash map
    abstract V get(K key); // Get value for a key
    abstract V remove(K key); // Remove key-value pair from the hash map
    // Calculate the load factor of the hash map
    protected double getLoadFactor() {
        return (double) size / capacity;
    }
}

// Separate Chaining Hash Map implementation
class SeparateChainingHashMap<K, V> extends AbstractHashMap<K, V> {
    private ArrayList<Map.Entry<K, V>>[] table; // Array of array lists for separate chaining
    // Constructor calling the superclass constructor
    public SeparateChainingHashMap(int capacity, double loadFactor) {
        super(capacity, loadFactor);
    }
    // Create the underlying data structure
    @Override
    void createTable() {
        table = (ArrayList<Map.Entry<K, V>>[]) new ArrayList[capacity];
        for (int i = 0; i < capacity; i++) {
            table[i] = new ArrayList<>();
        }
    }
    // Get value from a specific bucket
    @Override
    V bucketGet(int h, K k) {
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        for (Map.Entry<K, V> entry : bucket) {
            if (entry.getKey().equals(k)) {
                return entry.getValue();
            }
        }
        return null;
    }
// Put key-value pair in a specific bucket
    @Override
    void bucketPut(int h, K k, V v) {
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        bucket.add(new java.util.AbstractMap.SimpleEntry<>(k, v));
        size++;
        // Resize if load factor exceeds the threshold
        if (getLoadFactor() > loadFactor) {
            resize(2 * capacity - 1);
        }
    }
// Remove key-value pair from a specific bucket
    @Override
    V bucketRemove(int h, K k) {
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        Iterator<Map.Entry<K, V>> iterator = bucket.iterator();
        while (iterator.hasNext()) {
            Map.Entry<K, V> entry = iterator.next();
            if (entry.getKey().equals(k)) {
                iterator.remove();
                size--;
                return entry.getValue();
            }
        }
        return null;
    }
// Get all entries in the hash map
    @Override
    Set<Map.Entry<K, V>> entrySet() {
        Set<Map.Entry<K, V>> entries = new java.util.HashSet<>();
        for (ArrayList<Map.Entry<K, V>> bucket : table) {
            entries.addAll(bucket);
        }
        return entries;
    }
// Compute hash value for a key
    @Override
    int hashValue(K key) {
        int h = key.hashCode();
        return (h & 0x7FFFFFFF) % capacity;
    }
// Resize the hash map
    @Override
    void resize(int newCap) {
        ArrayList<Map.Entry<K, V>>[] newTable = (ArrayList<Map.Entry<K, V>>[])
        new ArrayList[newCap];
        for (int i = 0; i < newCap; i++) {
            newTable[i] = new ArrayList<>();
        }
    // Rehash and redistribute entries to the new table
        for (Map.Entry<K, V> entry : entrySet()) {
            int h = hashValue(entry.getKey());
            newTable[h].add(entry);
        }
        table = newTable;
        capacity = newCap;
    }
// Duplicate createTable method (commenting one occurrence to avoid redundancy)
//@Override
//void createTable() {
// table = (ArrayList<Map.Entry<K, V>>[]) new ArrayList[capacity];
// for (int i = 0; i < capacity; i++) {
// table[i] = new ArrayList<>();
// }
//}
// Put key-value pair in the hash map
    @Override
    void put(K key, V value) {
        int h = hashValue(key);
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        for (Map.Entry<K, V> entry : bucket) {
            if (entry.getKey().equals(key)) {
                entry.setValue(value);
                return;
            }
        }
        bucket.add(new java.util.AbstractMap.SimpleEntry<>(key, value));
        size++;
        // Resize if load factor exceeds the threshold
        if (getLoadFactor() > loadFactor) {
            resize(2 * capacity - 1);
        }
    }
// Get value for a key
    @Override
    V get(K key) {
        int h = hashValue(key);
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        for (Map.Entry<K, V> entry : bucket) {
            if (entry.getKey().equals(key)) {
                return entry.getValue();
            }
        }
        return null;
    }
// Remove key-value pair from the hash map
    @Override
    V remove(K key) {
        int h = hashValue(key);
        ArrayList<Map.Entry<K, V>> bucket = table[h];
        Iterator<Map.Entry<K, V>> iterator = bucket.iterator();
        while (iterator.hasNext()) {
            Map.Entry<K, V> entry = iterator.next();
            if (entry.getKey().equals(key)) {
                iterator.remove();
                size--;
                return entry.getValue();
            }
        }
        return null;
    }
}
        // Linear Probing Hash Map implementation
```

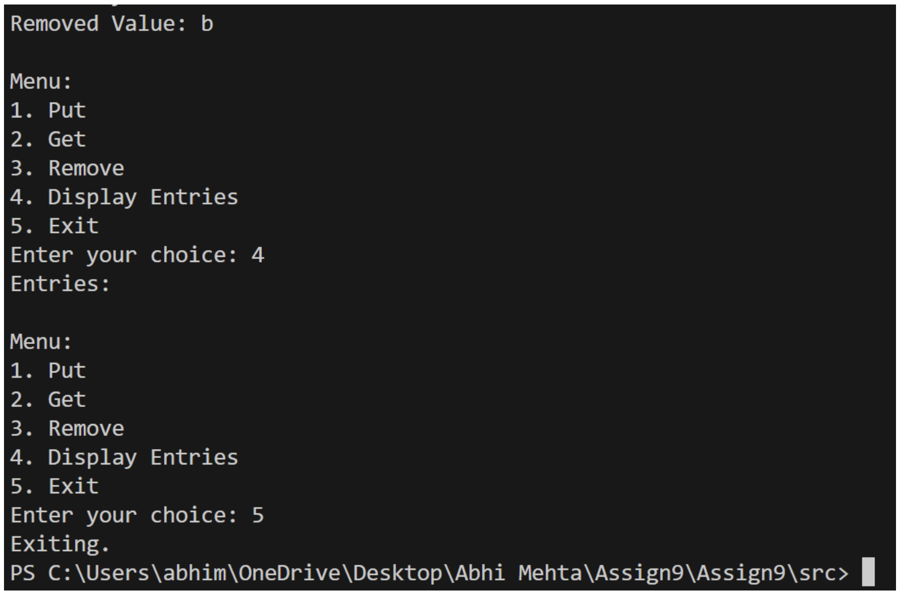
```
class LinearProbingHashMap<K, V> extends AbstractHashMap<K, V> {
    private K[] keys; // Array for keys
    private V[] values; // Array for values
        // Constructor calling the superclass constructor
    public LinearProbingHashMap(int capacity, double loadFactor) {
        super(capacity, loadFactor);
    }
// Create the underlying data structure
    @Override
    void createTable() {
        keys = (K[]) new Object[capacity];
        values = (V[]) new Object[capacity];
    }
        // Get value from a specific bucket using linear probing
    @Override
    V bucketGet(int h, K k) {
    int originalHash = h;
    do {
        K key = keys[h];
        if (key == null) return null;
        if (key.equals(k)) return values[h];
        h = (h + 1) % capacity;
    }
    while (h != originalHash);
        return null;
}
// Put key-value pair in a specific bucket using linear probing
    @Override
    void bucketPut(int h, K k, V v) {
        int originalHash = h;
        do {
            K key = keys[h];
            if (key == null || key.equals(k)) {
            keys[h] = k;
            values[h] = v;
            size++;
        // Resize if load factor exceeds the threshold
                if (getLoadFactor() > loadFactor) {
                    resize(2 * capacity - 1);
                }
                return;
            }
            h = (h + 1) % capacity;
        }
        while (h != originalHash);
    }
// Remove key-value pair from a specific bucket using linear probing
    @Override
    V bucketRemove(int h, K k) {
        int originalHash = h;
        do {
            K key = keys[h];
            if (key == null) return null;
            if (key.equals(k)) {
                V value = values[h];
                keys[h] = null;
                values[h] = null;
                size--;
                return value;
            }
            h = (h + 1) % capacity;
        }
        while (h != originalHash);
        return null;
    }
// Get all entries in the hash map
    @Override
    Set<Map.Entry<K, V>> entrySet() {
        Set<Map.Entry<K, V>> entries = new java.util.HashSet<>();
        for (int i = 0; i < capacity; i++) {
            K key = keys[i];
            if (key != null) {
                entries.add(new java.util.AbstractMap.SimpleEntry<>(key,
                values[i]));
            }
        }
        return entries;
    }
// Compute hash value for a key
    @Override
    int hashValue(K key) {
        int h = key.hashCode();
        return (h & 0x7FFFFFFF) % capacity;
    }
// Resize the hash map
    @Override
    void resize(int newCap) {
        LinearProbingHashMap<K, V> newMap = new
LinearProbingHashMap<>(newCap,loadFactor);
        for (Map.Entry<K, V> entry : entrySet()) {
            newMap.put(entry.getKey(), entry.getValue());
        }
        keys = newMap.keys;
        values = newMap.values;
        capacity = newMap.capacity;
    }
// Put key-value pair in the hash map
    @Override
    void put(K key, V value) {
        int h = findSlot(key);
        if (h < 0) {
            // Key not found, insert at the first available slot
            h = -h - 1;
            keys[h] = key;
            values[h] = value;
            size++;
            // Resize if load factor exceeds the threshold
            if (getLoadFactor() > loadFactor) {
                resize(2 * capacity - 1);
            }
        }
        else {
            // Key already exists, update the value
            values[h] = value;
        }
    }
// Get value for a key
    @Override
    V get(K key) {
        int h = findSlot(key);
        if (h < 0) return null;
        return values[h];
    }
// Remove key-value pair from the hash map
    @Override
    V remove(K key) {
        int h = findSlot(key);
        if (h < 0) return null;
        V removed = values[h];
        keys[h] = null;
        values[h] = null;
        size--;
        return removed;
    }
// Find the slot for a key (used in put, get, and remove methods)
    private int findSlot(K key) {
        int originalHash = hashValue(key);
        int h = originalHash;
        do {
            if (keys[h] == null) {
                return -1; // Key not found
            }
            if (keys[h].equals(key)) {
                return h; // Key found
            }
            h = (h + 1) % capacity;
        }
        while (h != originalHash);
        return -1; // Full loop, key not found
    }
}

//Menu.java


import java.util.Map;
import java.util.Scanner;

public class Menu<K, V> {

    public static void main(String[] args) {
        new Menu<String, Object>().menuDrivenProgram();
    }

    private void menuDrivenProgram() {
        Scanner scanner = new Scanner(System.in);
        AbstractHashMap<K, V> hashMap = null;

        System.out.println("Choose HashMap implementation:");
        System.out.println("1. Separate Chaining");
        System.out.println("2. Linear Probing");

        int choice = scanner.nextInt();

        switch (choice) {
```

```
    case 1:
        hashMap = new SeparateChainingHashMap<>(10, 0.75);
        break;
    case 2:
        hashMap = new LinearProbingHashMap<>(10, 0.75);
        break;
    default:
        System.out.println("Invalid choice. Exiting.");
        return;
}

while (true) {
    System.out.println("\nMenu:");
    System.out.println("1. Put");
    System.out.println("2. Get");
    System.out.println("3. Remove");
    System.out.println("4. Display Entries");
    System.out.println("5. Exit");
    System.out.print("Enter your choice: ");

    int menuChoice = scanner.nextInt();

    switch (menuChoice) {
        case 1:
            System.out.print("Enter key: ");
            K putKey = (K) scanner.next();
            System.out.print("Enter value: ");
            V putValue = (V) castToType(scanner.next());
            hashMap.put(putKey, putValue);
            System.out.println("Key-Value pair added.");
            break;
        case 2:
            System.out.print("Enter key: ");
            K getKey = (K) scanner.next();
            V getValue = hashMap.get(getKey);
            System.out.println("Value: " + getValue);
            break;
        case 3:
            System.out.print("Enter key: ");
            K removeKey = (K) scanner.next();
            V removedValue = hashMap.remove(removeKey);
            if (removedValue != null) {
                System.out.println("Removed Value: " + removedValue);
            } else {
                    System.out.println("Key not found.");
                }
                break;
            case 4:
                System.out.println("Entries:");
                for (Map.Entry<K, V> entry : hashMap.entrySet()) {
                    System.out.println(entry.getKey() + ": " + entry.getValue());
                }
                break;
            case 5:
                System.out.println("Exiting.");
                scanner.close();
                System.exit(0);
                break;
            default:
                System.out.println("Invalid choice. Try again.");
        }
    }
}

// Utility method to handle type casting
private Object castToType(String value) {
    try {
        return Integer.parseInt(value);
    } catch (NumberFormatException e) {
        // If it's not an integer, return as String
        return value;
    }
}
}
```

**Output:**



```
Choose HashMap implementation:
1. Separate Chaining
2. Linear Probing
1

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 1
Enter key: a
Enter value: b
Key-Value pair added.

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 2
Enter key: a
Value: b

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 4
Entries:
a: b

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 3
Enter key: a
```



```
Removed Value: b

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 4
Entries:

Menu:
1. Put
2. Get
3. Remove
4. Display Entries
5. Exit
Enter your choice: 5
Exiting.
PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\Assign9\Assign9\src>
```

## **<u>Errors Encountered:</u>**

Compilation Errors:

- Description: Compilation errors occur during the compilation phase when the source code is translated into bytecode. These errors indicate syntax or semantic issues in the code.
- Solution: To address compilation errors, carefully examine the error messages provided by the compiler. Focus on fixing syntax errors, addressing missing semicolons, and resolving other issues highlighted by the compiler. Pay close attention to the line numbers mentioned in the error messages, as they pinpoint the location of the problems.

Logic Errors:

- Description: Logic errors manifest when a program produces incorrect results due to flaws in the algorithm or logical structure of the code.
- Solution: To rectify logic errors, employ debugging techniques such as inserting print statements, utilizing debugging tools, or implementing logging to trace the flow of execution. Review the logic and algorithm in the code to identify and rectify errors that lead to incorrect outcomes. This process involves a careful examination of the program's logical flow and structure.

## Learnings from the Code:

Type Inference Issues:

- The code encountered type inference issues, particularly when trying to create instances of SeparateChainingHashMap and LinearProbingHashMap. This occurred because the generic types K and V were not explicitly specified when creating instances of these classes.
- Learning: When creating instances of generic classes, ensure that the generic types are explicitly provided, especially when the compiler cannot infer them.

Type Casting and Generics:

- The code involved type casting from Object to generic types (K and V) and vice versa. This led to unchecked type safety warnings.
- Learning: Type casting should be used cautiously, and efforts should be made to handle generics more safely to avoid unchecked type warnings.

Static Reference to Non-static Type:

- The code encountered an error stating, "Cannot make a static reference to the non-static type V." This was due to trying to use a non-static type (V) in a static context.
- Learning: Understand the distinction between static and non-static contexts. Non-static types cannot be directly referenced in static contexts, and appropriate measures should be taken to address this.

User Input Handling:

- User input was taken using the Scanner class, and the code attempted to handle input for both keys and values in a generic way.
- Learning: When handling user input in a generic context, appropriate casting and validation mechanisms are crucial to ensure correct data types are used.

## Conclusion:

The experiment provided valuable insights into Java programming concepts, including generics, user input handling, and the implementation of data structures. It showcased the adaptability of HashMaps by incorporating different collision resolution strategies, allowing for a deeper understanding of their impact on performance and efficiency. The experience reinforced the importance of modular design and effective error resolution in creating robust and flexible software.