**day18\_107856406\_dsdipt\_sudipto\_16july2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 16 July 2025 (Day 18)

### ***Task 1: MCQ Hash Table Collision Resolution Strategy***

Given the following Java implementation:

class Task01 {

LinkedList<Entry>[] data = new LinkedList[10];

public void put(String keyval, int value) {

int index = Math.abs(keyval.hashCode() % data.length);

if (data[index] == null) {

data[index] = new LinkedList<>();

}

for (Entry e : data[index]) {

if (e.keyval.equals(keyval)) {

e.value = value;

return;

}

}

data[index].add(new Entry(keyval, value));

}

static class Entry {

String keyval;

int value;

Entry(String k, int v) {

keyval = k;

value = v;

}

}

}

**What kind of collision resolution strategy is implemented here?**

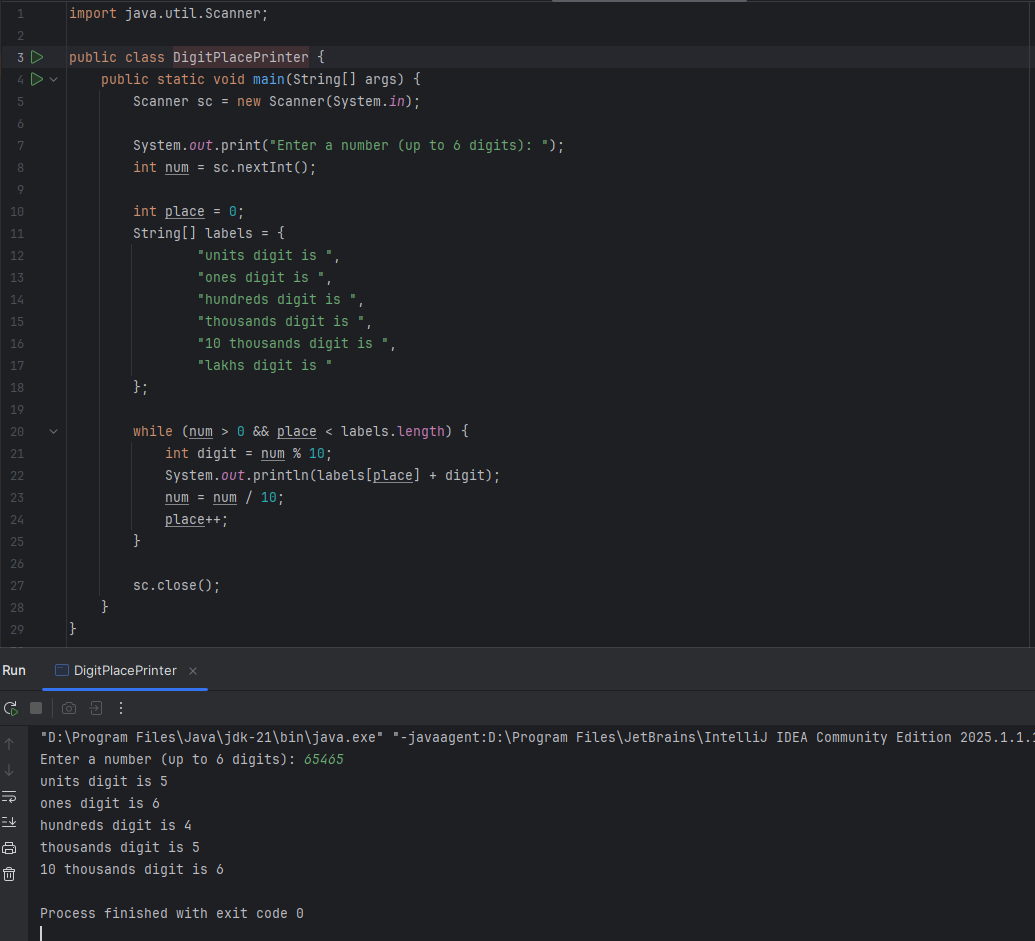
**🔘 A. Linear Probing with Backtracking**

**🔘 B. Open Addressing by Placing Values at Next Available Bucket**

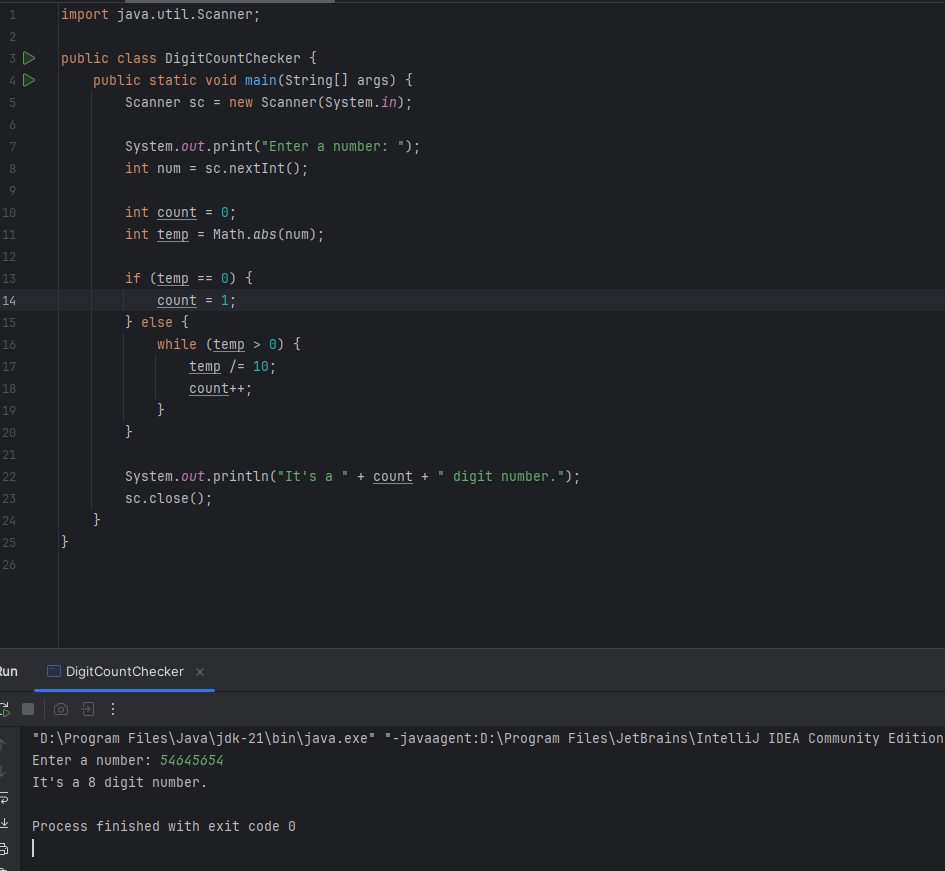
**✅ C. Chaining at Each Index Using a Linked List**

**🔘 D. Resizing the Hash Table on Each Collision**

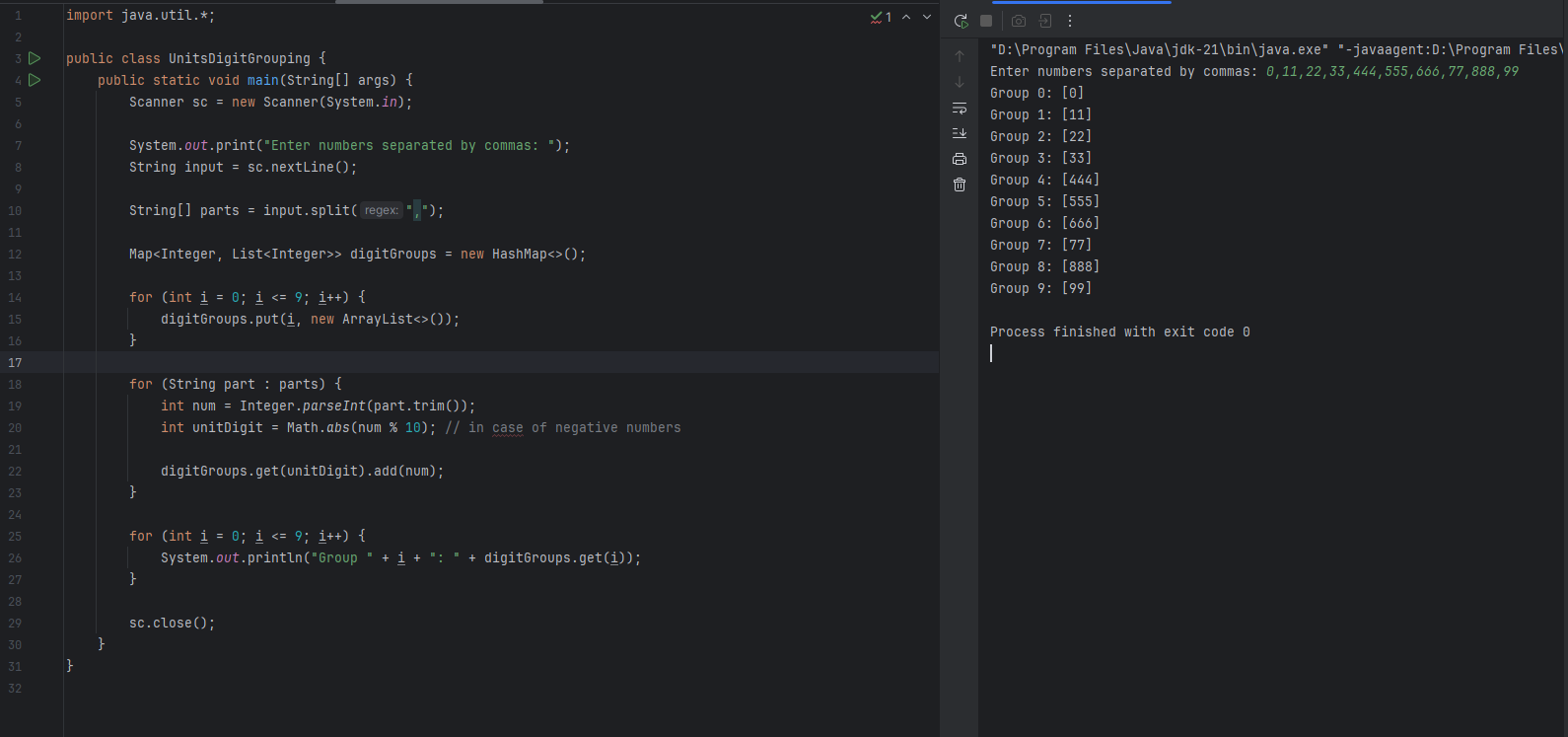
### ***Task 2: Display Place Value of Each Digit in a 5-Digit Number***



### ***Task 3: Count and Display the Number of Digits in an Input Number***



### ***Task 4: Group Numbers Based on Their Units Digit***



### ***Task 5: Write Algo for Radix sort***

1. **Find the maximum number in the array.**
2. **Set digit place = 1 (for units digit).**
3. **Repeat while (max / digit place) > 0:**

* Use counting sort to sort the array based on the current digit place.
* Move to the next digit place (multiply digit place by 10).

### ***Task 6: Write pseudo Code for Radix sort***

Algorithm RadixSort(arr)

1. Find the maximum number in arr

2. Get the number of digits in the maximum number → call it maxDigits

3. Set place = 1 // 1 for units, then 10 for tens, etc.

4. While (place <= maxDigits):

a. Create 10 empty buckets (lists) for digits 0 to 9

b. For each number in arr:

i. Get the digit at 'place' → (number / place) % 10

ii. Put the number into the corresponding bucket

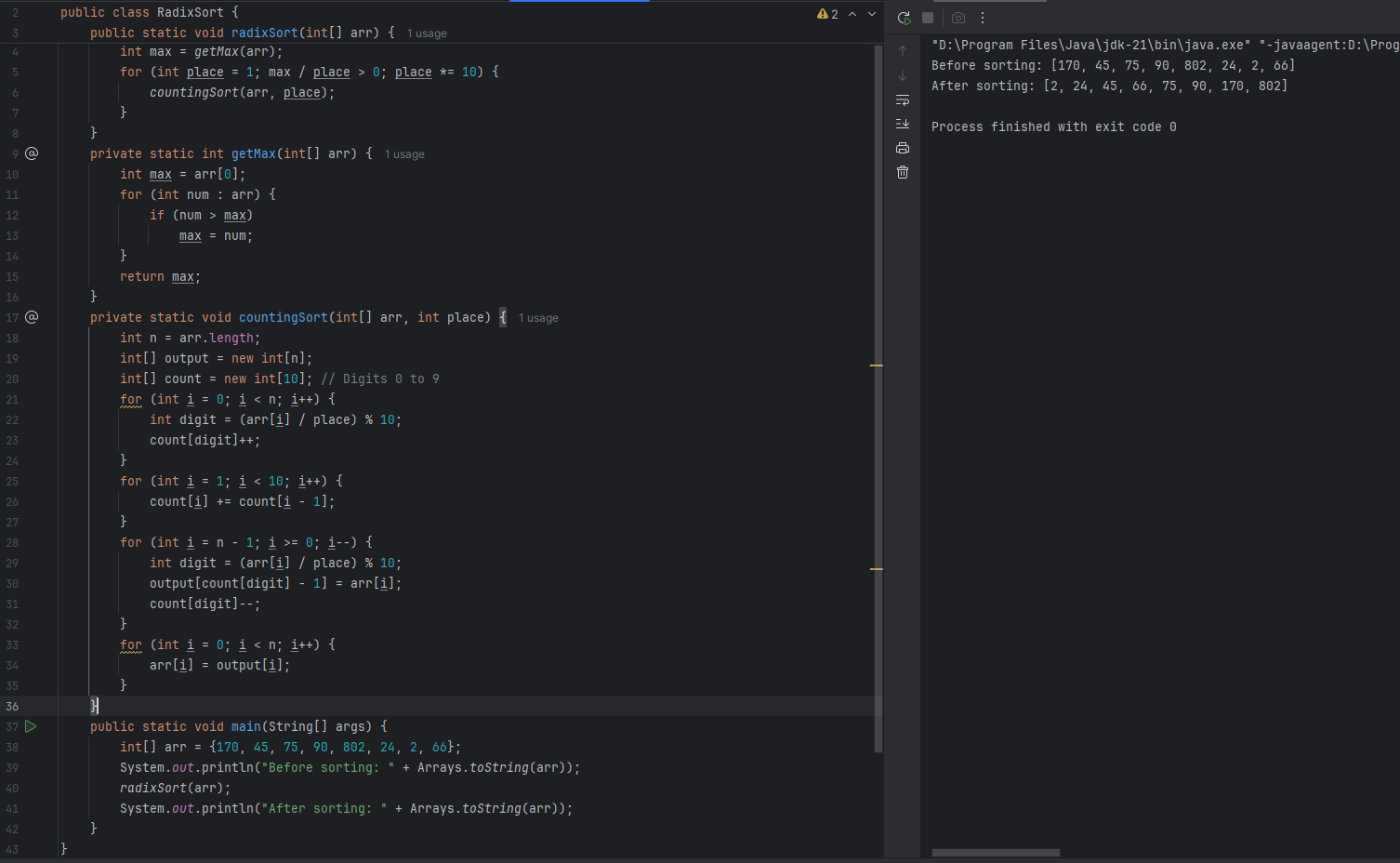
c. Reconstruct arr by collecting numbers from all buckets in order

d. Multiply place by 10 (move to next digit place)

5. Return arr

End Algorithm

### ***Task 7: Write Code for Radix sort***



### ***Task 8: Which of the following statements is true?***

**Do you find any significant change between the breadthFirstSearchRecursive() approach compared to the standard BFS?**

A. ⬜ It removes the need for queues entirely by using stack-based recursion.  
B. ⬜ It simplifies implementation by using queues implicitly within recursive function calls.  
C. ✅ It achieves the same result but emphasizes a recursive style using the same level-order logic with explicit queue management.  
D. ⬜ It processes nodes in post-order sequence to avoid memory allocation.

### ***Task 9: What is Memoization?***

Memoization is like giving your function a **memory**.

When we solve a problem once, you **store the result** so the next time you get the same input, you can **skip the work** and just reuse the answer.

It’s used to **optimize recursive solutions**, especially in problems where the same subproblem happens **over and over again** (aka overlapping subproblems).

**Without Memoization:**

Let’s say we’re solving Fibonacci recursively:

int fib(int n) {

if (n <= 1) return n;

return fib(n - 1) + fib(n - 2);

}

* Calling fib(5) will trigger fib(4) and fib(3)
* But fib(3) gets called **again** inside fib(4)
* It’s like solving the same math multiple times 😩

**With Memoization:**

We use a cache (like a HashMap or array) to remember:

Map<Integer, Integer> memo = new HashMap<>();

int fib(int n) {

if (n <= 1) return n;

if (memo.containsKey(n)) return memo.get(n); // already solved

int result = fib(n - 1) + fib(n - 2);

memo.put(n, result); // save it for next time

return result;

}

* Now fib(3) only gets solved **once**
* Every other time, we just **fetch** from memory
* Much faster!

### ***Task 10: What is Dynamic Programming?***

Dynamic Programming is like solving a problem by **breaking it down into smaller pieces**, solving each piece **only once**, and then **reusing** those answers to build up to the final solution.

It’s perfect for problems that have:

* **Overlapping subproblems** (same problem gets solved multiple times)
* **Optimal substructure** (big solution depends on small ones)

**Real Example: Climbing Stairs**

You can climb either 1 or 2 steps at a time.  
 How many ways can you reach the top if there are n stairs?

**Brute Force (Recursion):**

int climb(int n) {

if (n <= 1) return 1;

return climb(n - 1) + climb(n - 2);

}

* Recomputes the same stuff over and over 😵
* Exponential time = bad

**With Dynamic Programming (Bottom-Up):**

int climbStairs(int n) {

int[] dp = new int[n + 1];

dp[0] = dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

* We solve from the bottom up
* Each subproblem is solved once
* 🚀 Much faster: O(n) time, O(n) space

### ***Task 11: Fibonacci using Dynamic Programming***

public class FibonacciDP {

public static int fib(int n) {

if (n <= 1) return n;

int[] dp = new int[n + 1]; // Store results

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

public static void main(String[] args) {

int n = 10;

System.out.println("Fibonacci of " + n + " is: " + fib(n));

}

}

### ***Task 12: Explanation of Heap Sort Technique***

* **Build a max heap** from the input array.
* The **largest element** is now at the root (index 0).
* **Swap** the root with the last element in the heap.
* **Reduce heap size** by 1 and **heapify** the root to restore heap property.
* **Repeat** the swap and heapify process until the heap size becomes 1.
* Result: Array is sorted in **ascending order**, in-place, with **O(n log n)** time complexity.

### ***Task 13: Do recursive functions maintain the state of each call during execution?***

**A.** Each recursive call creates a new thread, and context switching maintains state.  
**B.** Recursive functions store state in global variables accessible across calls.  
**C. The system call stack tracks local variables and return addresses for each recursive invocation.**  
**D.** Recursive functions replicate the heap structure to keep values between calls.

### ***Task 14: Iterative uses less memory — no stack frames.***

**TRUE**

### ***Task 15: Missing base case or deep recursion → stack overflow.***

**TRUE**

### ***Task 16: Missing base case or deep recursion → stack overflow.***

import java.util.\*;

public class HashCollision {

static class Entry {

String key;

int value;

Entry(String key, int value) {

this.key = key;

this.value = value;

}

}

// Hash table with chaining

List<Entry>[] table = new ArrayList[10];

// Insert or update key-value pair

public void put(String key, int val) {

int index = Math.abs(key.hashCode() % table.length);

if (table[index] == null) {

table[index] = new ArrayList<>();

}

for (Entry entry : table[index]) {

if (entry.key.equals(key)) {

entry.value = val; // Update existing key

return;

}

}

table[index].add(new Entry(key, val)); // Add new key

}

// Get value by key

public Integer get(String key) {

int index = Math.abs(key.hashCode() % table.length);

if (table[index] != null) {

for (Entry entry : table[index]) {

if (entry.key.equals(key)) {

return entry.value;

}

}

}

return null; // Key not found

}

}

public class Main {

public static void main(String[] args) {

HashCollision map = new HashCollision();

map.put("apple", 1);

map.put("orange", 2);

map.put("apple", 99); // update

System.out.println(map.get("apple")); // Output: 99

System.out.println(map.get("orange")); // Output: 2

}

}

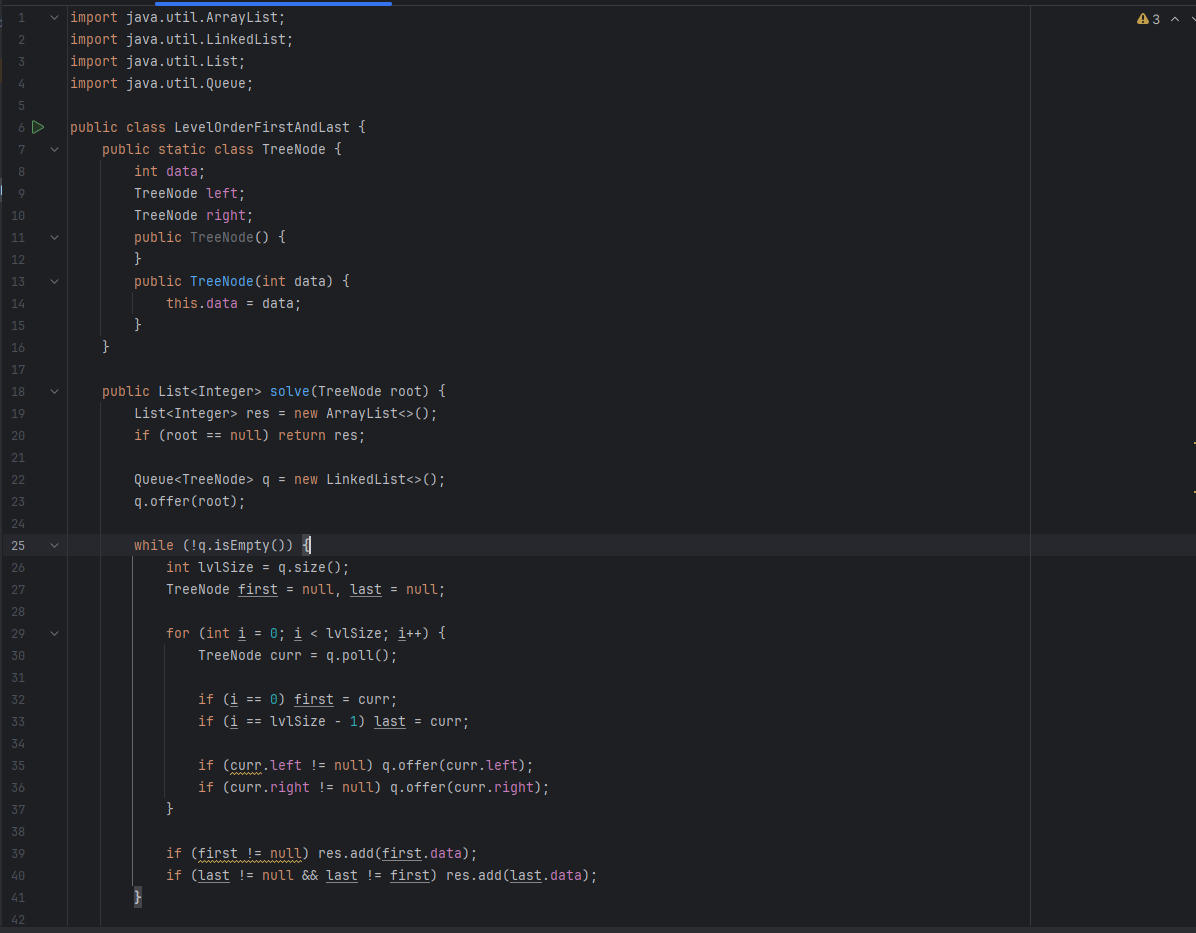
**1.** Insertion will fail due to duplicate key exception

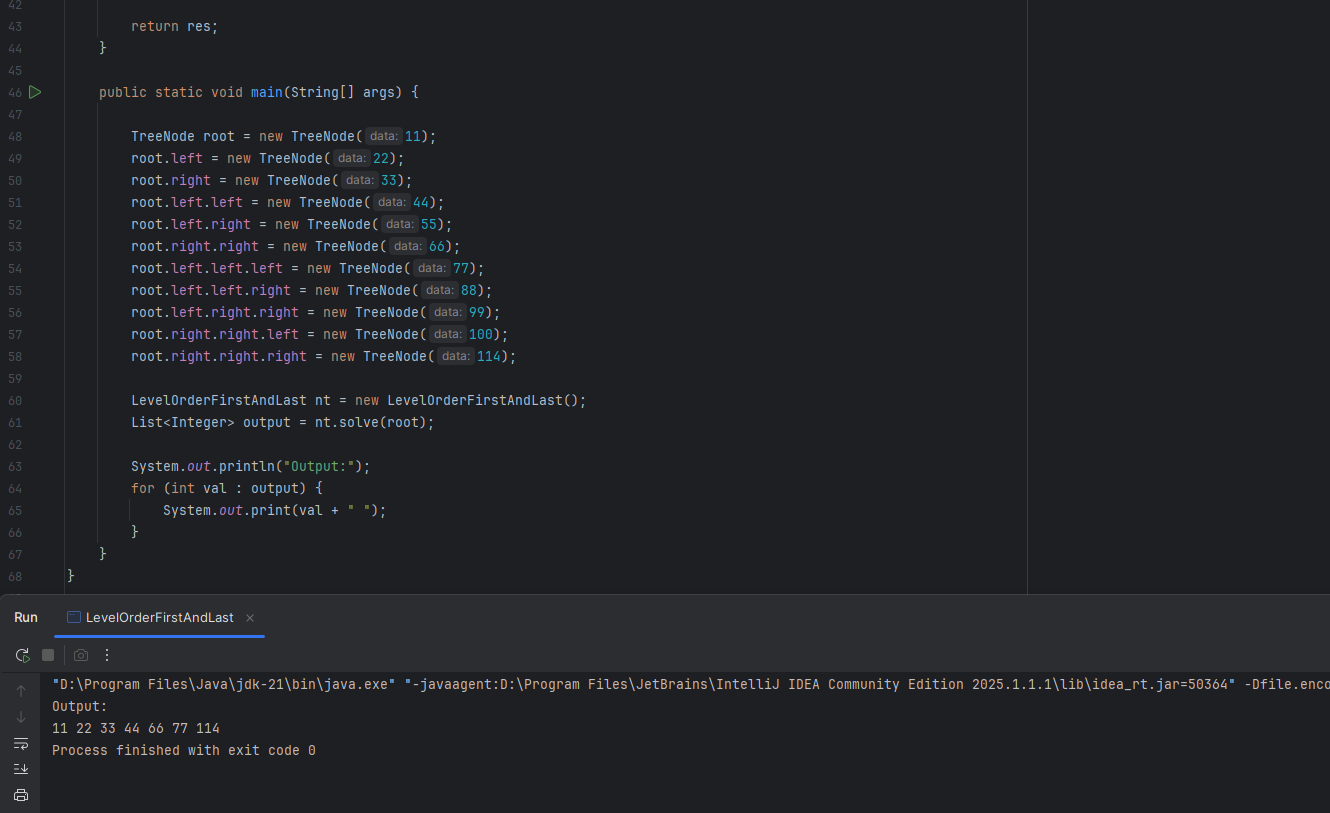
**2.** Values are distributed across different buckets using linear probing

**3.** Only one key-value pair will be stored due to overwriting

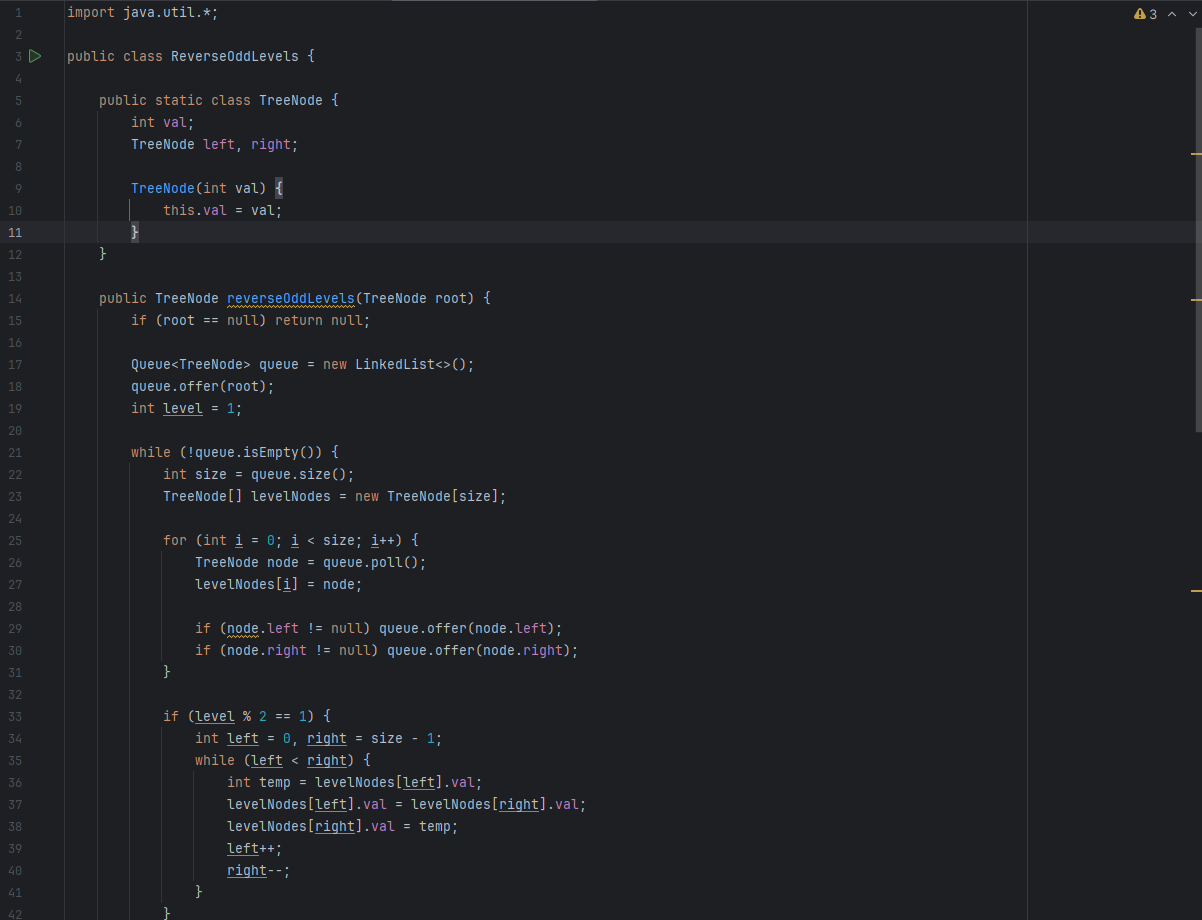
**4. Multiple values are stored in the same bucket via chaining**

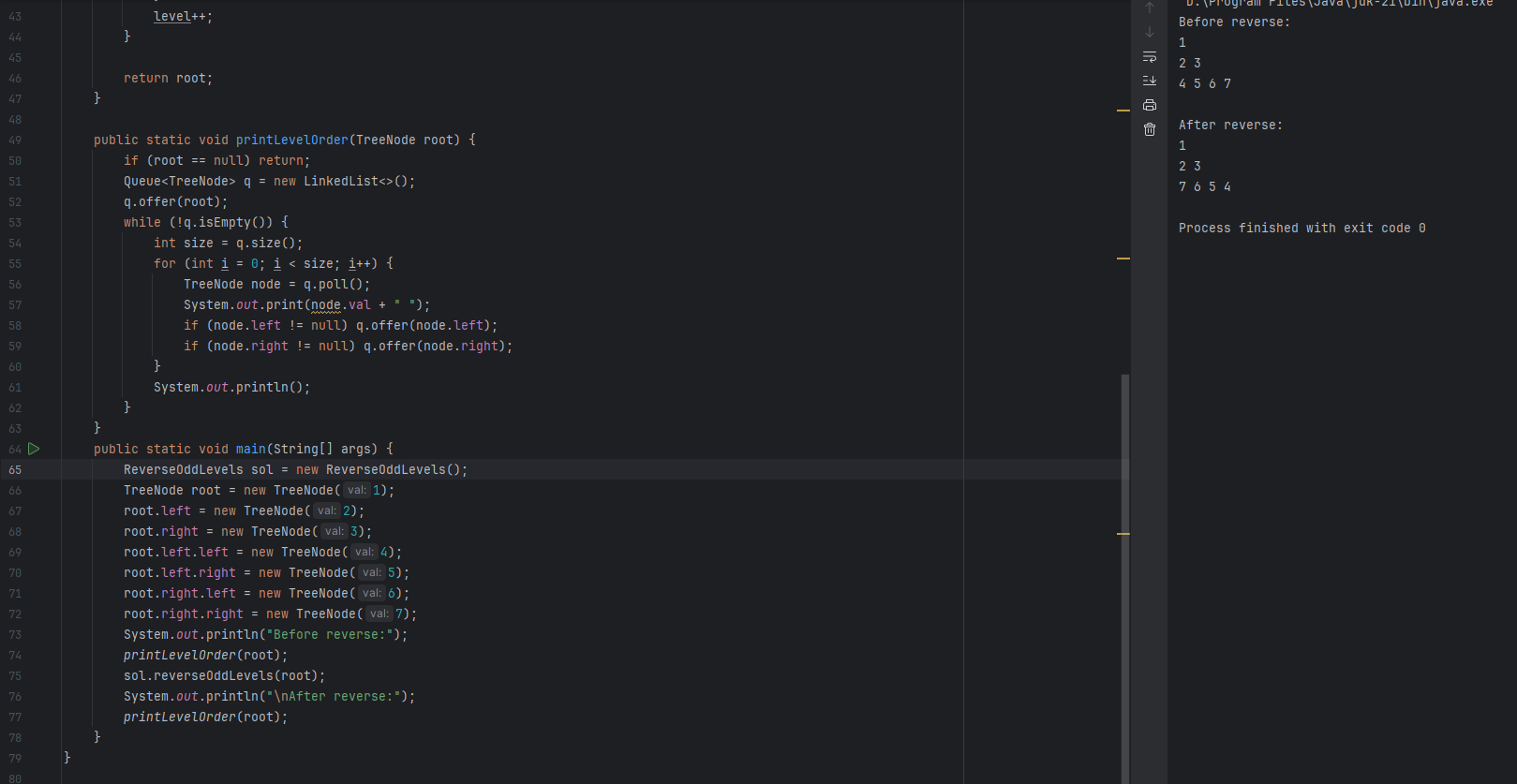
### ***Task 17: Level Order Traversal – First and Last Node at Each Level***





### ***Task 18: Reverse Node Values at Odd Levels in a Binary Tree***





### ***Task 19: LeetCode Problem 199 : Binary Tree Right Side View***

class Solution {

public List<Integer> rightSideView(TreeNode root) {

List<Integer> res = new ArrayList<>();

if(root == null) {

return res;

}

Queue<TreeNode> q = new LinkedList<>();

q.add(root);

while(!q.isEmpty()) {

int lvlSize = q.size();

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

TreeNode curr = q.poll();

if(curr.left!=null) {q.add(curr.left);}

if(curr.right!=null) {q.add(curr.right);}

if(i==lvlSize-1) {

res.add(curr.val);

}

}

}

return res;

}

}