Task 01:

What kind of collision resolution strategy is implemented in the below Hash Table ?

import java.util.\*;

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;

        }

    }

}

is it using

1. to fill collisions is it linear probing with backtracking

or

1. Opening address by placing values at next available bucket

or

1. at each index chaining using a linked list

or

1. on each collision resizing hash table

Correct ans: The code you provided uses chaining with a linked list as its collision resolution strategy.

Explanation:

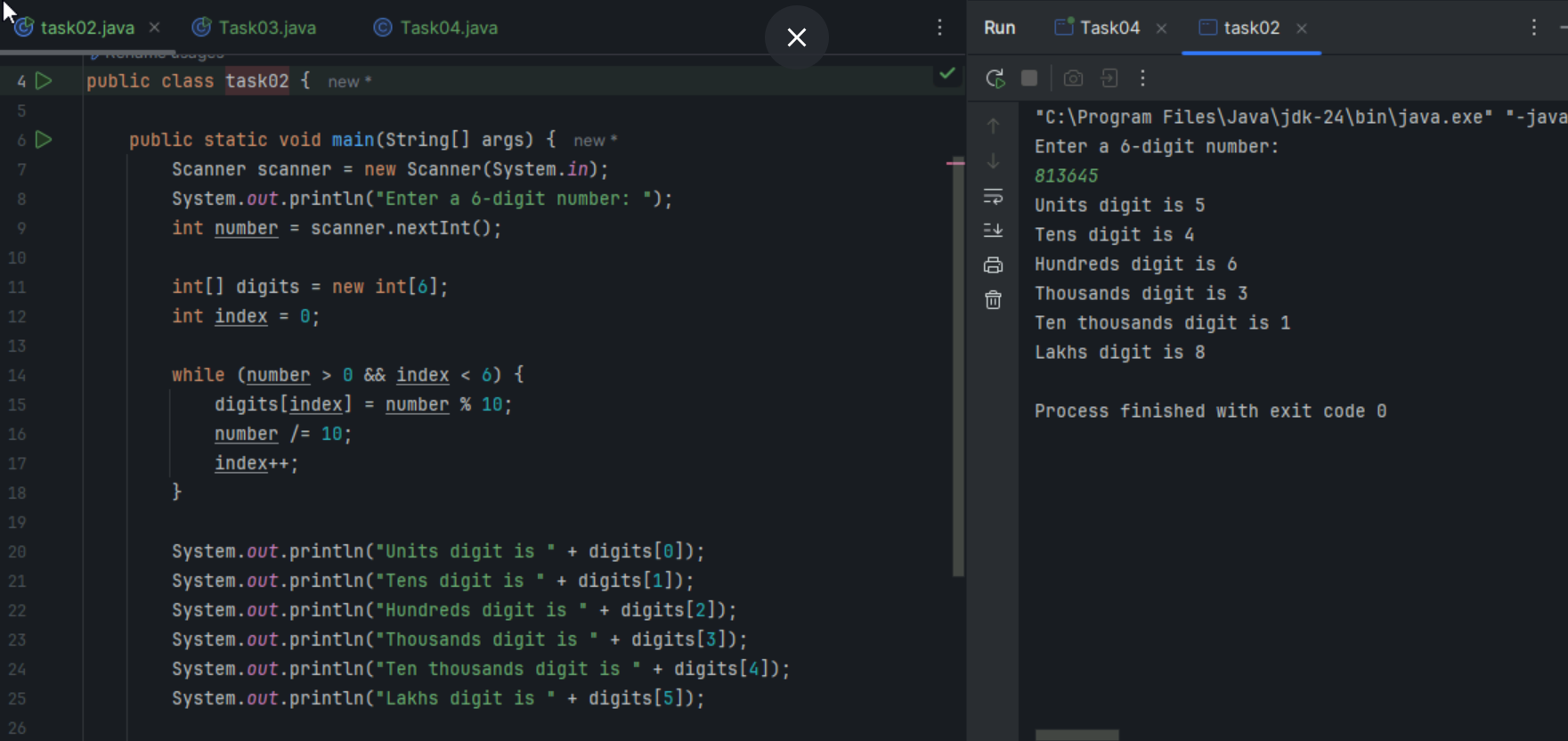
Each index in the data array contains a LinkedList<Entry>.

If two keys hash to the same index, their Entry objects are stored in the linked list at that index.

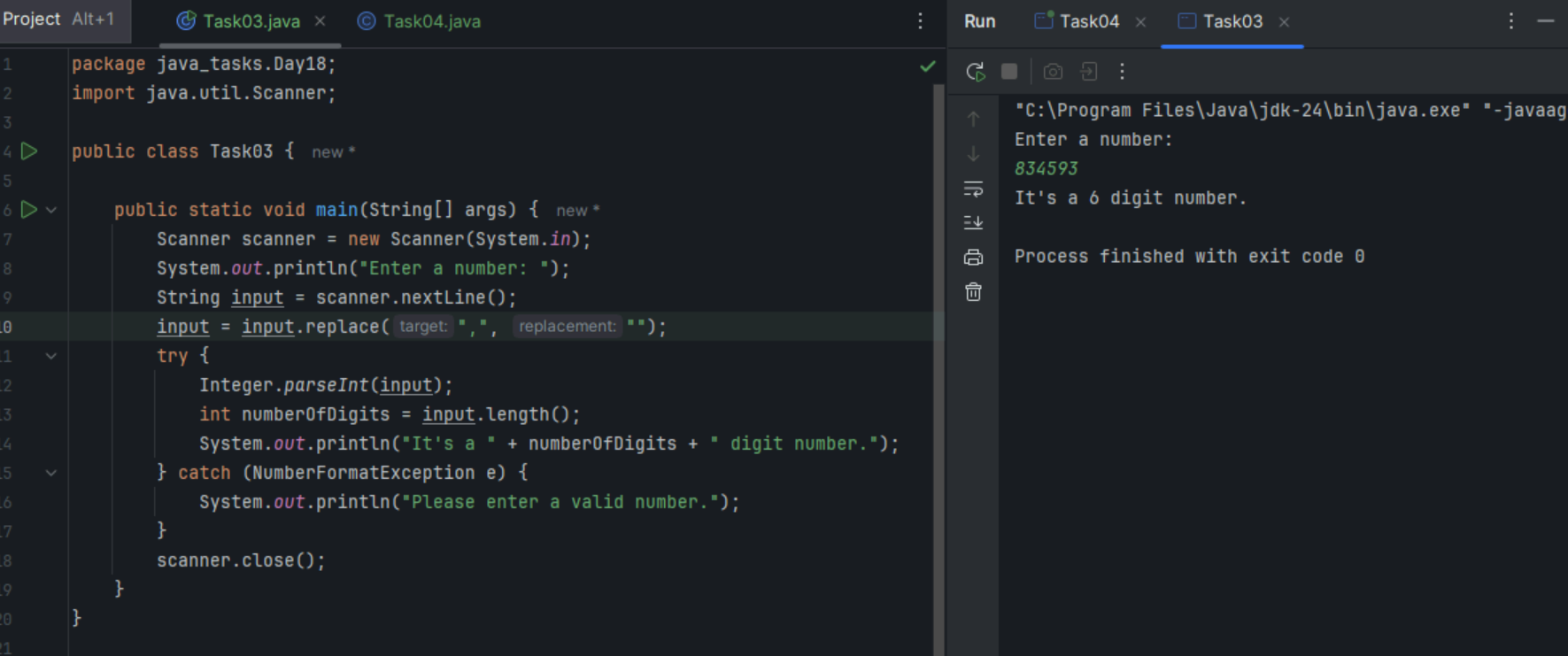
This allows multiple entries to exist at the same index without overwriting each other.

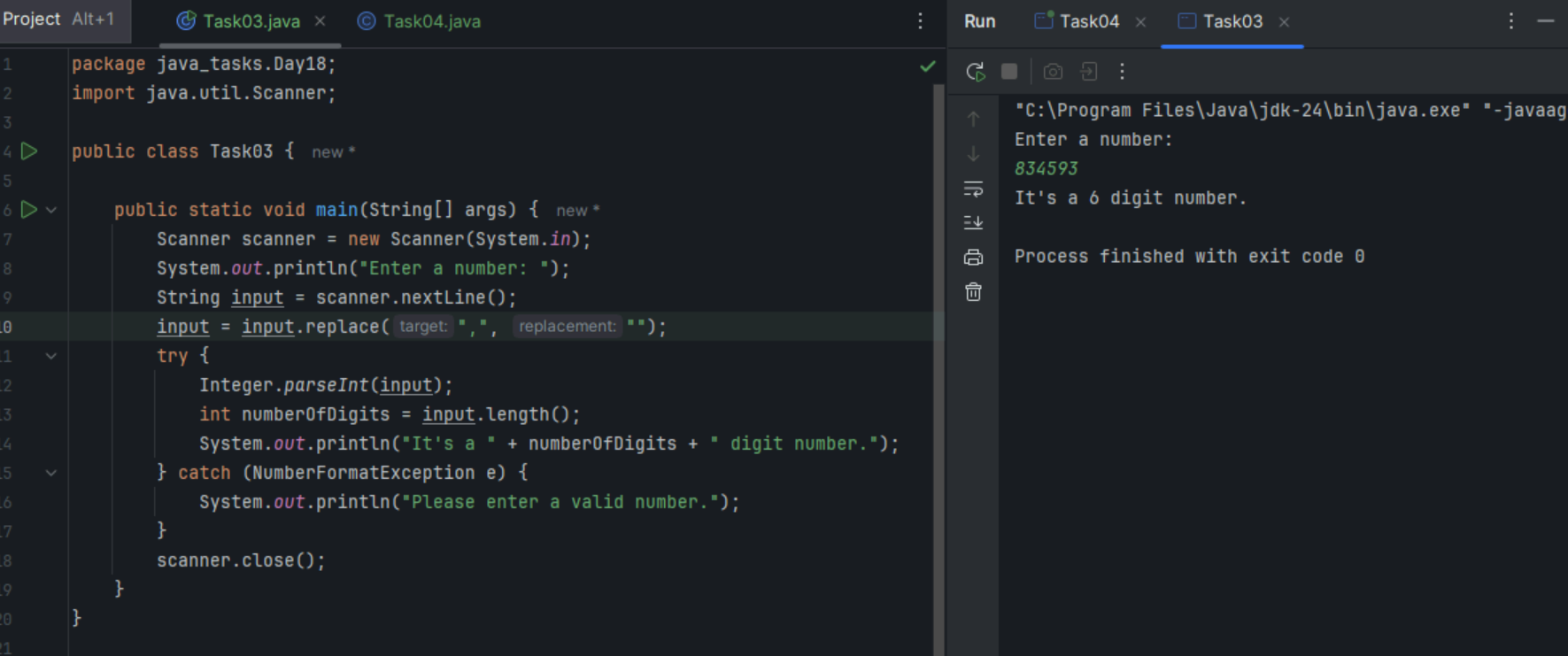
The strategy is chaining rather than open addressing or resizing upon collision.

Task02:



Task03:





Task 04:

What are the applications of heap sort?

Heap sort has several applications, including:

Sorting: Used for sorting large datasets due to its efficiency and in-place sorting capabilities.

Priority Queues: Implements priority queues, where elements with the highest priority are processed first.

Order Statistics: Can be used to find the k-th smallest or largest element efficiently.

Selection Algorithms: Useful in scenarios where selection of top elements is required quickly, such as finding the median.

Real-time Systems: Its predictable performance makes it suitable for real-time system constraints.

Task 05:

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

1.Will it the need for queues entirely by using a stack-based recursion?

2.Will it simplifies implementation by using queues implicitly within recursive function calls?

3.will it achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management?

4.will it processes nodes in post-order sequence to avoid memory allocation?

Ans : A recursive approach could still manage a queue explicitly to maintain the BFS logic. This approach maintains level-order processing while framing the logic recursively, but it still relies on a queue to manage the nodes at each level.

Task 06:

How does heap sort work ? explain the technique in 5

Build a Max Heap: Convert the array into a max heap, where each parent node is greater than its children. Start from the last non-leaf node and apply heapify up to the root.

Max Element Extraction: Swap the first element (maximum) with the last element in the heap. Reduce the heap size by one.

Heapify: Restore the max heap property by heapifying the root element. This involves comparing the root with its children and swapping with the largest if necessary.

Iterate:

Repeat the extraction and heapify steps for the reduced heap size until all elements are sorted.

Result:

The array is now sorted in ascending order as the maximum elements are repeatedly moved to the end.

Task 07:

how can you say recursive functions maintain the state of each call during execution?

1. Each recursive call creates a new thread, and context switching maintains state.

2. Recursive functions store state in global variables accessible across calls.

3. The system call stack tracks local variables and return addresses for each recursive invocation.

4. Recursive functions replicate the heap structure to keep values between calls.

Ans : 3

Each recursive call creates a new frame on the call stack.

This frame stores local variables and the return address, effectively maintaining the state for that specific call.

When a function call returns, its frame is removed, and the context switches back to the calling function's frame.

Task 08: Priority Queue Property

The correct option is:

Elements are dequeued based on their priority, not their insertion order, often implemented using a binary heap.

Task 09: Purpose of Using a Binary Heap

The correct option is:

To ensure that the highest-priority element always bubbles to the root efficiently.

Task 10:

Can you print the corner nodes of a binary search tree?

Task 15:

Write algo for radix sort

RADIX-SORT Algorithm:

1. Find the maximum number in the array to determine the number of digits

2. Initialize digit position (exp) = 1

3. While (maximum number / exp) > 0:

. Perform counting sort for current digit position

. Multiply exp by 10 to move to next digit position

4. Array is now sorted

COUNTING-SORT-FOR-RADIX (for a specific digit position):

1. Create count array of size 10 (for digits 0-9)

2. Count occurrences of each digit at current position

3. Modify count array to contain actual positions

4. Build output array using the count array

5. Copy output array back to original array

Task 16:

Write pseudo code for radix sort

PROCEDURE RadixSort(array[], n)

// Step 1: Find maximum number

max ← array[0]

FOR i ← 1 TO n-1 DO

IF array[i] > max THEN

max ← array[i]

END IF

END FOR

// Step 2: Process each digit position

exp ← 1 // Represents current digit position (1s, 10s, 100s, etc.)

WHILE (max / exp) > 0 DO

CountingSortByDigit(array, n, exp)

exp ← exp × 10

END WHILE

END PROCEDURE

PROCEDURE CountingSortByDigit(array[], n, exp)

// Step 1: Initialize

output[0...n-1] ← new array

count[0...9] ← {0} // Initialize count array with zeros

// Step 2: Count occurrences of each digit

FOR i ← 0 TO n-1 DO

digit ← (array[i] / exp) MOD 10

count[digit] ← count[digit] + 1

END FOR

// Step 3: Update count array to contain actual positions

FOR i ← 1 TO 9 DO

count[i] ← count[i] + count[i-1]

END FOR

// Step 4: Build output array (traverse from right to left for stability)

FOR i ← n-1 DOWNTO 0 DO

digit ← (array[i] / exp) MOD 10

output[count[digit] - 1] ← array[i]

count[digit] ← count[digit] - 1

END FOR

// Step 5: Copy output array back to original array

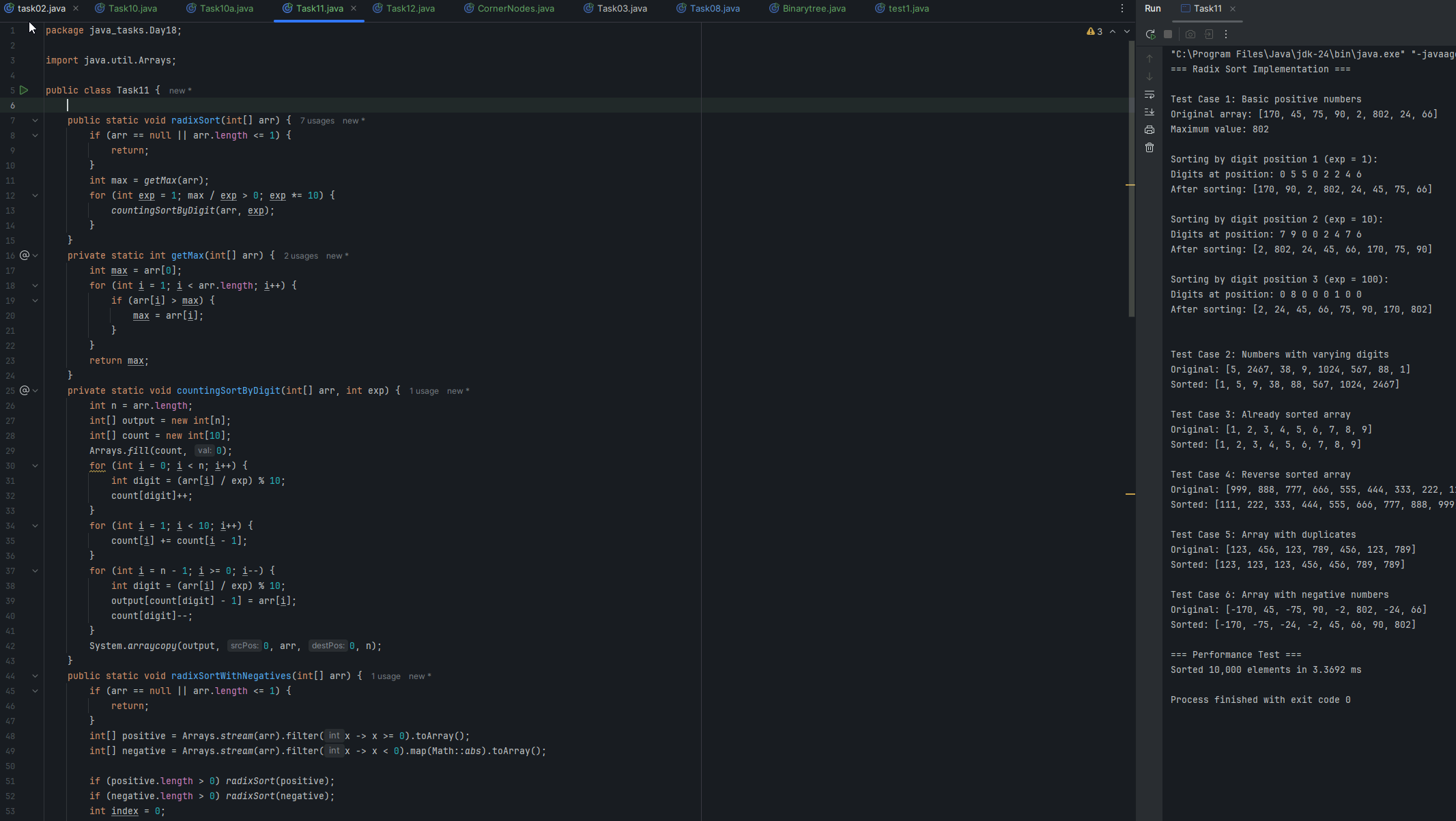
FOR i ← 0 TO n-1 DO

array[i] ← output[i]

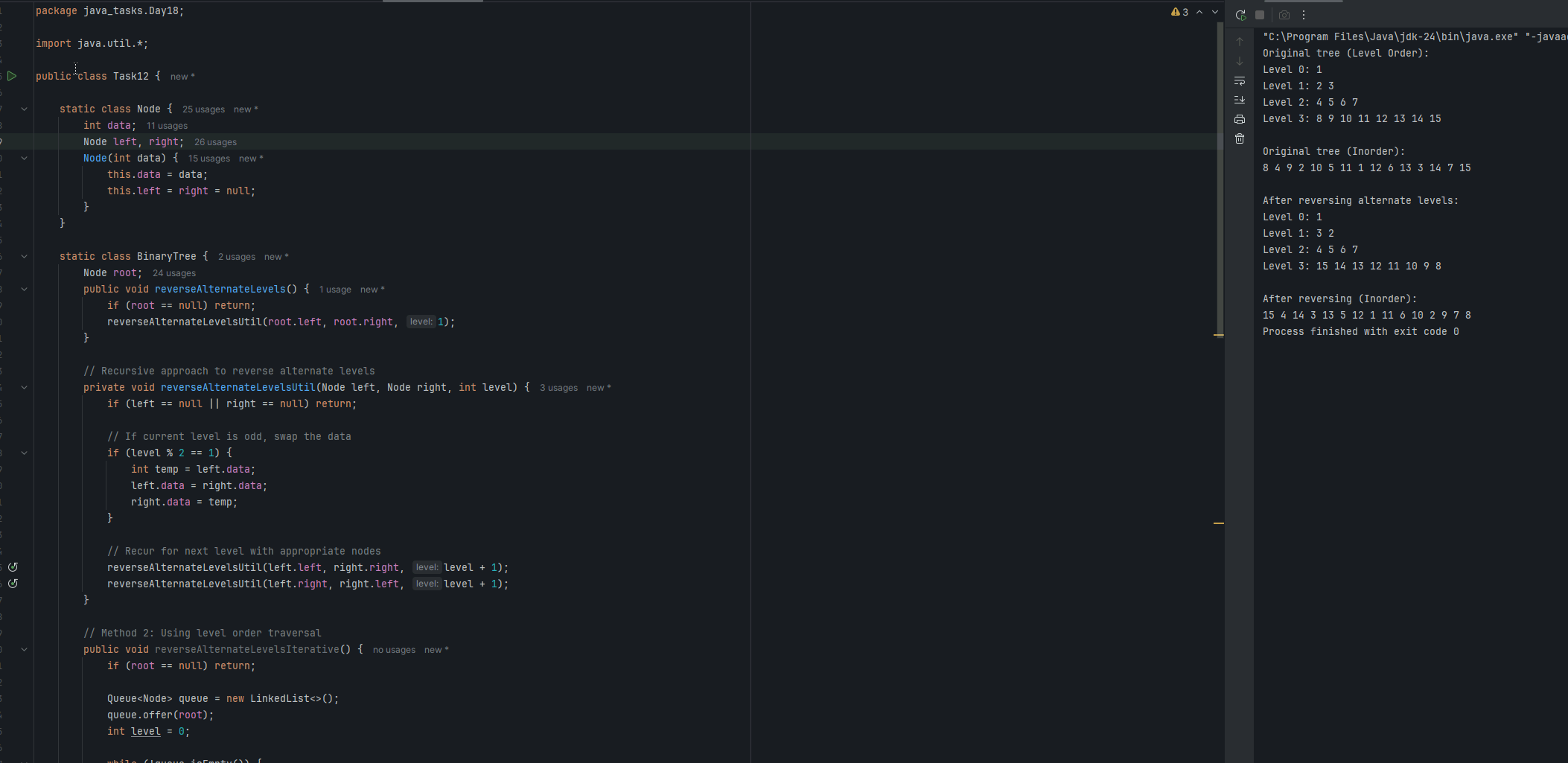
END FOR

END PROCEDURE

Task 17: display all corner elements of a tree



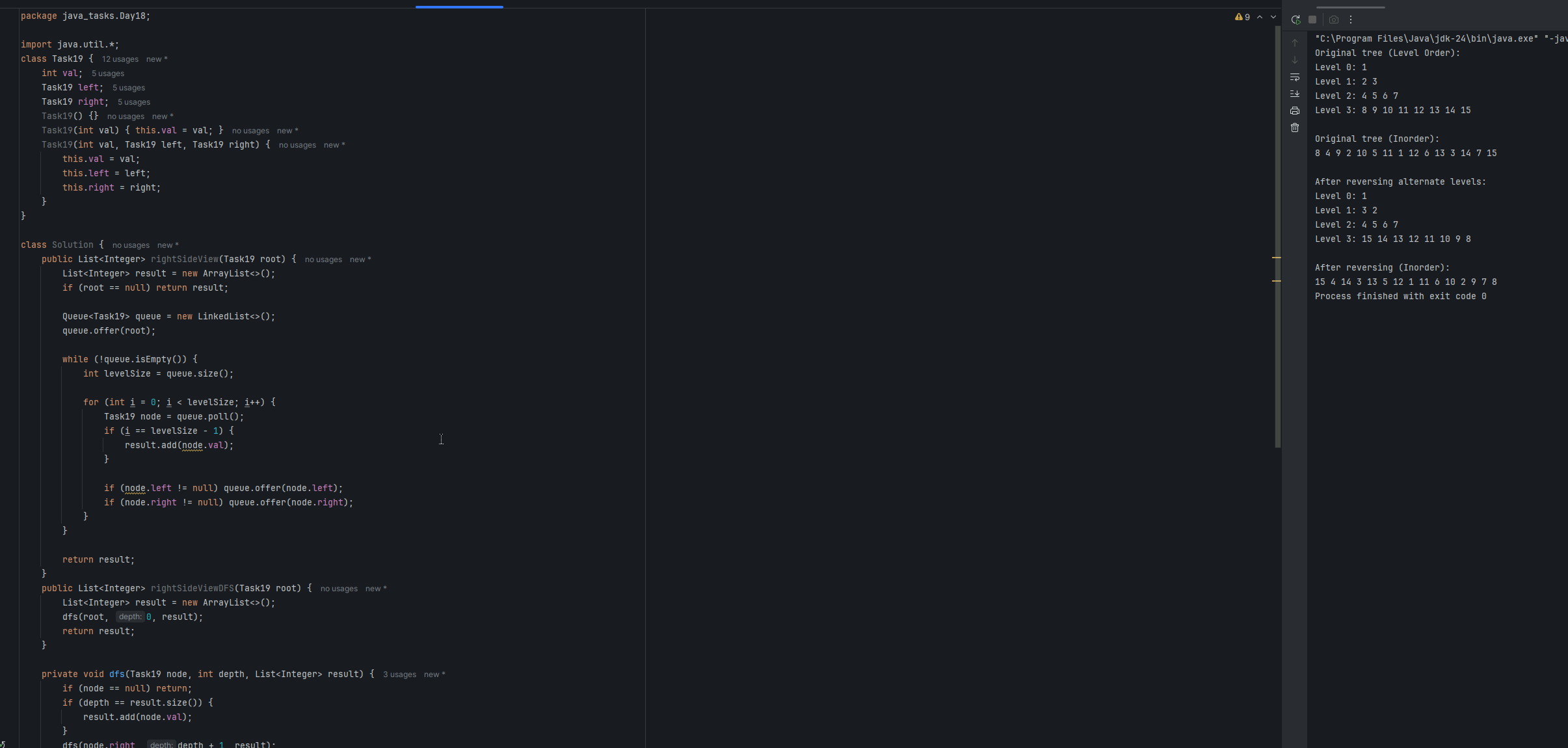
Task 18  Reverse Alternate levels:



Task, 19 (leet code problem statement) right side view

<https://leetcode.com/problems/binary-tree-right-side-view/description/>

Plz solve this Problem statement



Addon :

Wap to display the groups of digits depending upon the unit digits

Hint:

If input is 45,81, 85,100,20. 95,60,10,21

Output:

Array 1 has : 100,20,60,10

Array 2 has : 81, 21

Array 3 has : 45 , 85 ,95

