CENG 202 – Data Structures Project Report

Course ID and Name: CENG 202 – Data Structures
Students Name/Number:

Umut Caner 230401035

Deniz Cem Cangöz 220401037

Ahmet Anıl Uçan   230401082

Submitted to:

Prof. Dr. Aytuğ ONAN

Ars. Gör. Cem AYAR

Project Title: ParcelSortX: Smart Package Sorting and Routing Simulation Using Classical Data Structures
Project Type: Application Project
Submission Date: 14.06.2025

Introduction

This Project we make using a parcel sorting center simulation by handwriten data structures in Java. The system models parcel flow from arrival to sorting and dispatching, and includes realistic behaviors such as terminal rotation and parcel misrouting. The simulation demonstrates the practical application of core data structures like stacks, queues, circular linked lists, binary search trees, and hash tables.

Program Interface

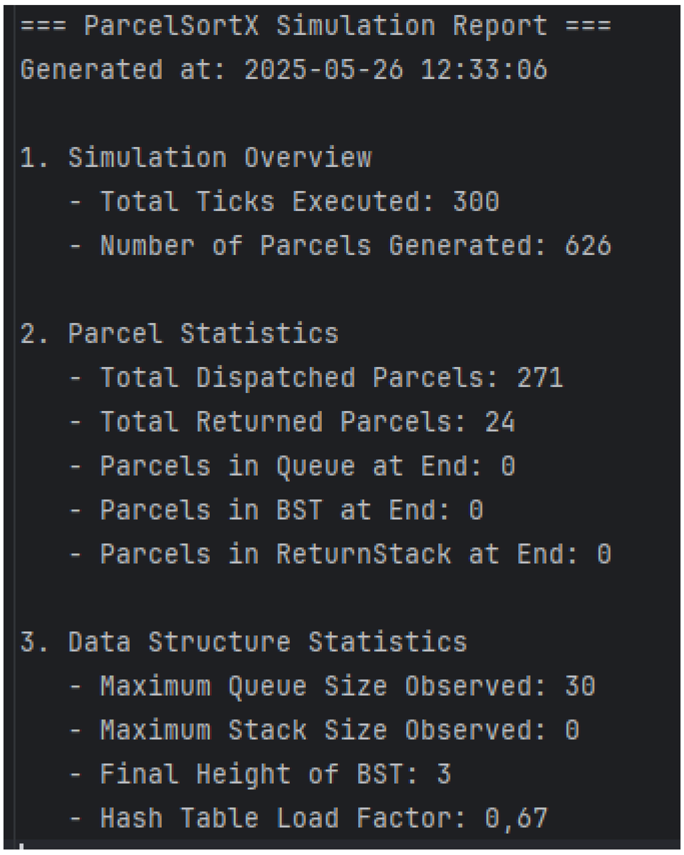
The program runs on console-based interface. System parameters such as queue capacity, number of ticks, terminal rotation interval, and misrouting probability are configured in config.txt. Simulation progress is displayed through console logs and written to log.txt. To run the code, the user run the simulation.java with an IDE such as IntelliJ IDEA or VS2022

Program Execution

This program take parameters such as queue capacity, number of ticks, terminal rotation interval, and misrouting probability are configured from config.txt.

Each simulation tick performs the following actions:

- Generates new parcels and inserts them into the ArrivalBuffer(queue).
- Moves one parcel from the queue to the BST (DestinationSorter).
- Dispatches parcels for the current terminal, with change of misrouting(%10 change).
- Processes ReturnStack every 3 ticks to reintegrate misrouted parcels(take it from config.txt)
- Rotates the terminal every specified interval.
- Logs tick summaries, including queue and stack states.
- Write the report:

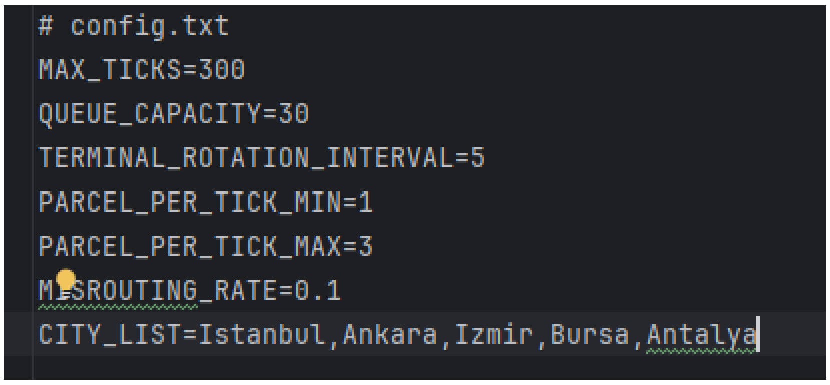


```
=== ParcelSortX Simulation Report ===
Generated at: 2025-05-26 12:33:06

1. Simulation Overview
   - Total Ticks Executed: 300
   - Number of Parcels Generated: 626

2. Parcel Statistics
   - Total Dispatched Parcels: 271
   - Total Returned Parcels: 24
   - Parcels in Queue at End: 0
   - Parcels in BST at End: 0
   - Parcels in ReturnStack at End: 0

3. Data Structure Statistics
   - Maximum Queue Size Observed: 30
   - Maximum Stack Size Observed: 0
   - Final Height of BST: 3
   - Hash Table Load Factor: 0,67
```

Input and Output

Input:

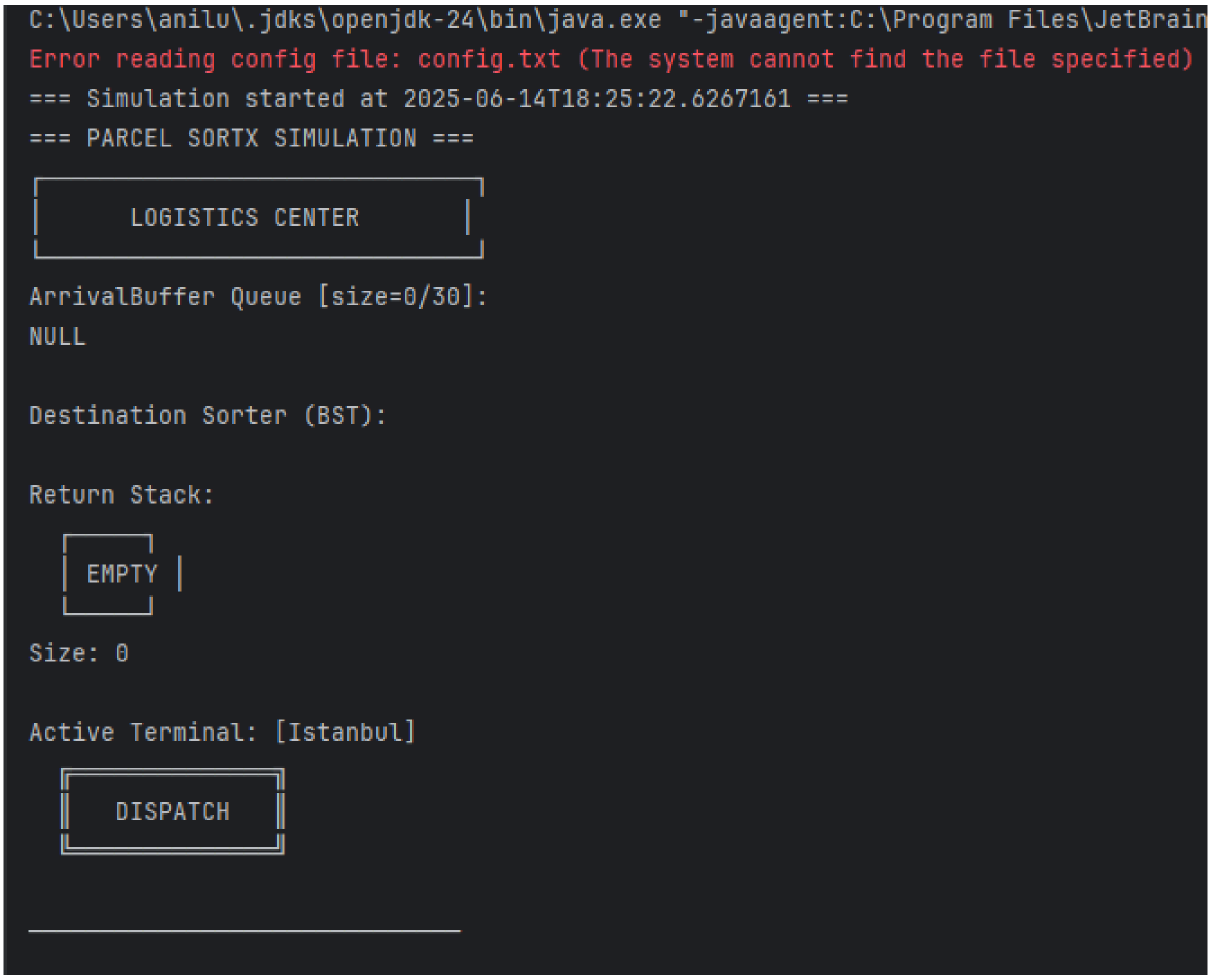
- config.txt file: defines simulation parameters (number of ticks, cities, misrouting rate, etc.). You can add or change the parameters.



```
# config.txt
MAX_TICKS=300
QUEUE_CAPACITY=30
TERMINAL_ROTATION_INTERVAL=5
PARCEL_PER_TICK_MIN=1
PARCEL_PER_TICK_MAX=3
MISROUTING_RATE=0.1
CITY_LIST=Istanbul,Ankara,Izmir,Bursa,Antalya
```

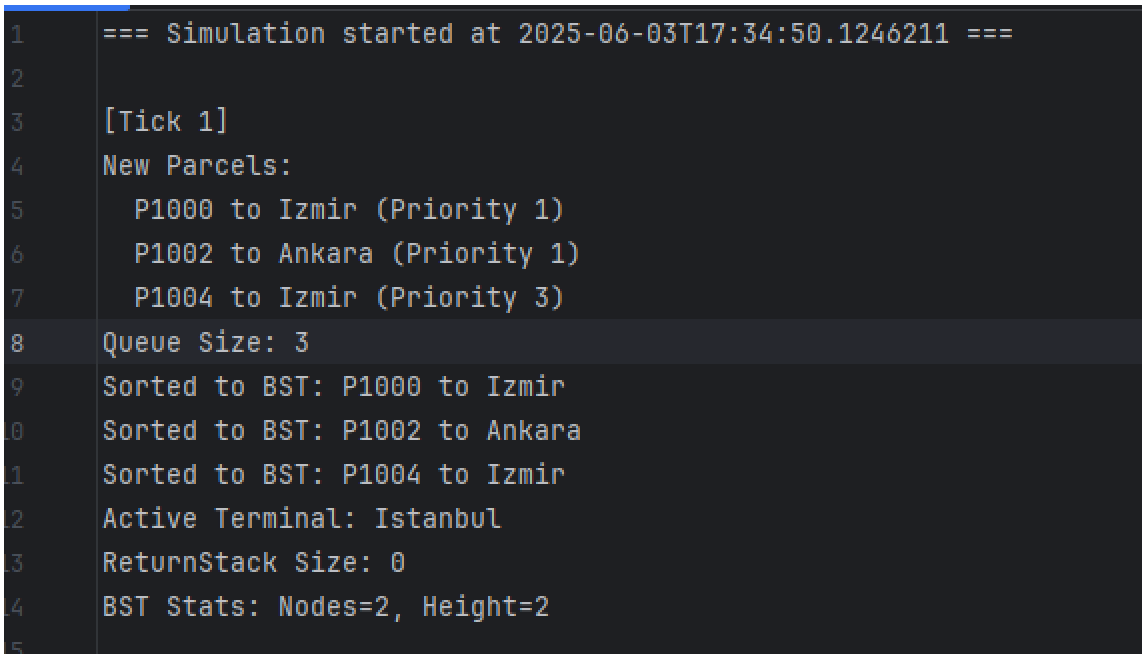
Output:

- Console: visualization of queue, stack, BST state per tick.



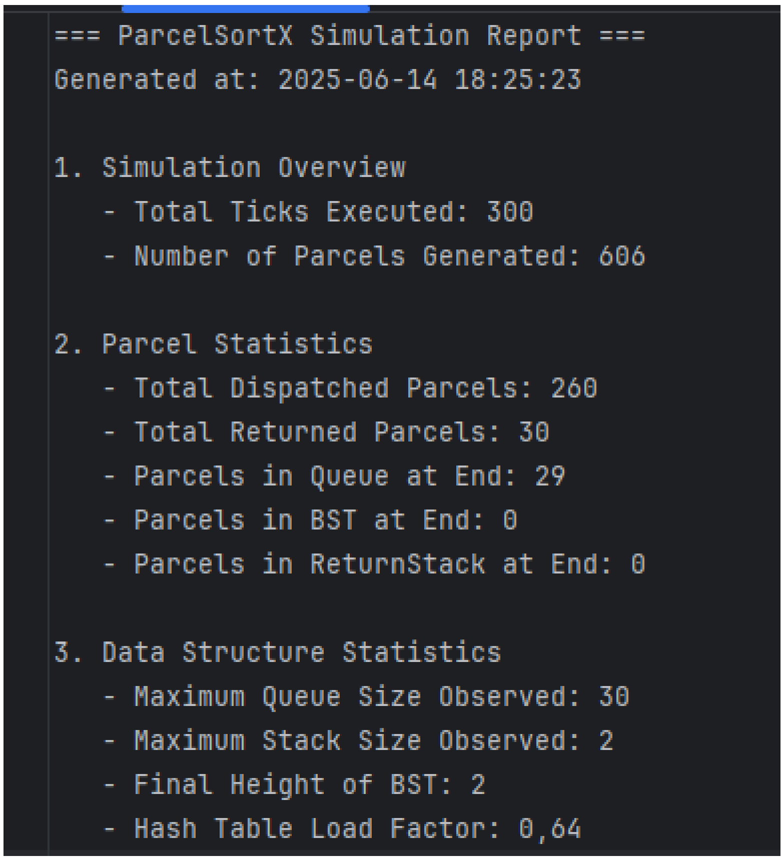
```
C:\Users\anilu\.jdks\openjdk-24\bin\java.exe "-javaagent:C:\Program Files\JetBrain
Error reading config file: config.txt (The system cannot find the file specified)
=== Simulation started at 2025-06-14T18:25:22.6267161 ===
=== PARCEL SORTX SIMULATION ===
┌─────────────────────────────┐
│      LOGISTICS CENTER       │
└─────────────────────────────┘
ArrivalBuffer Queue [size=0/30]:
NULL

Destination Sorter (BST):

Return Stack:
  ┌─────┐
  │ EMPTY │
  └─────┘
Size: 0

Active Terminal: [Istanbul]
  ╔═════════════╗
  ║  DISPATCH   ║
  ╚═════════════╝

______________________________
```

- log.txt: detailed record of all events and parcel movements.



```
=== Simulation started at 2025-06-03T17:34:50.1246211 ===

[Tick 1]
New Parcels:
  P1000 to Izmir (Priority 1)
  P1002 to Ankara (Priority 1)
  P1004 to Izmir (Priority 3)
Queue Size: 3
Sorted to BST: P1000 to Izmir
Sorted to BST: P1002 to Ankara
Sorted to BST: P1004 to Izmir
Active Terminal: Istanbul
ReturnStack Size: 0
BST Stats: Nodes=2, Height=2
```

- Report.txt: final record of the output like overview, parcel statistic and data structure statistics:



```
=== ParcelSortX Simulation Report ===
Generated at: 2025-06-14 18:25:23

1. Simulation Overview
   - Total Ticks Executed: 300
   - Number of Parcels Generated: 606

2. Parcel Statistics
   - Total Dispatched Parcels: 260
   - Total Returned Parcels: 30
   - Parcels in Queue at End: 29
   - Parcels in BST at End: 0
   - Parcels in ReturnStack at End: 0

3. Data Structure Statistics
   - Maximum Queue Size Observed: 30
   - Maximum Stack Size Observed: 2
   - Final Height of BST: 2
   - Hash Table Load Factor: 0,64
```

- Final summary including parcel counts, max stack size, BST height, and hash table load factor.

Program Structure

UML of the program: mermaid code

(In case of link doesn’t work, we also added the png as zip file)

Key Classes:

- Simulation: Manages overall program flow.
- ArrivalBuffer: Handmade queue for parcels.
- DestinationSorter: AVL-balanced BST with linked lists for city-based parcel storage.
- ReturnStack: Hand written stack for holding misrouted parcels.
- ParcelTracker: Hand written hash table for tracking parcel statuses.
- TerminalRotator: Circular linked list managing terminal rotation.
- Config: Loads and parses config.txt.
- Logger: Handles console and file logging.

Data Structures:

- Linked List: ArrivalBuffer, city parcel lists, DestinationSorter.
- Queue: ArrivalBuffer
- Stack: ReturnStack
- Circular Linked List: TerminalRotator.
- AVL Binary Search Tree: DestinationSorter.
- Hash Table: ParcelTracker.

Examples

At each tick, the program generates 1-3 new parcels. One parcel is moved from the ArrivalBuffer to the BST per tick (bottleneck). The currently active terminal is used to dispatch available parcels. Misrouted parcels(10% chance) are pushed to the ReturnStack and reuse later. Every few ticks, the active terminal rotates, enabling dispatch from different cities. The program logs each turn, and outputs a final report showing simulation results.

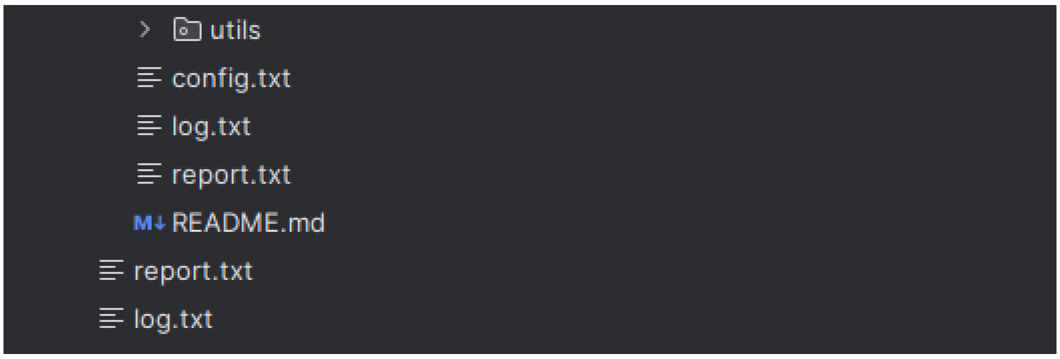
Improvements and Extensions

- Can be added better and interactive interface.
- Add a more better parcel prioritization mechanism in the queue.
- Provide visualization of the terminal rotation and BTS as a map.

- Implement parcel aging (time in system).
- Support batch dispatching of multiple parcels per tick.

Difficulties Encountered

- Biggest difficulty we had encountered is the fact we couldn't use Github as we want to so we just send zip file of code each other every time.
- We forgot to add bottleneck part so queue always seen as 0 so fix it much later by Changing queue to BTS flow by 1 parcel per tick.
- When we run the code the log.txt and report.txt duplicate but didn't affect our program running so we keep it in case:



```
    > utils
    config.txt
    log.txt
    report.txt
    README.md
  report.txt
  log.txt
```

Conclusion

This project successfully demonstrates the use of handmade data structures queue (ArrivalBuffer), AVL tree (DestinationSorter), stack (ReturnStack), hash table (ParcelTracker), and circular linked list (TerminalRotator) to model a realistic system. The modular and object-oriented design ensures extensibility. The system can be further improved by enhancing performance and visualization features.

References

Data structures: Course Slides and lab solutions CENG202

Hashmap usage: https://www.w3schools.com/java/java_hashmap.asp

How AVL Trees Work (with Rotations): https://medium.com/@agarwalharshita23/avl-tree-in-java-1cdd5d25d3bd

Build a Parcel Delivery System using Core Java: https://medium.com/@vishalsingh.dev/logistics-system-in-java-3eab3f7de7d8

Appendices

- Source Code for: Simulation.java, Parcel.java, ParcelPriority.java, ParcelSize.java, ParcelStatus.java, ArrivalBuffer.java, ReturnStack.java, DestinationSorter.java, ParcelTracker.java, TerminalRotator.java, HashTable.java, LinkedList.java, Config.java, Logger.java, TerminalUI.java

- Config.txt, log.txt, report.txt
- Parcel projects report
- PNG of UML of classes

Source code:

```
package main;

import model.Parcel;
import model.enums.ParcelPriority;
import model.enums.ParcelSize;
import model.enums.ParcelStatus;
import structures.*;
import utils.Config;
import utils.Logger;
import utils.TerminalUI;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Random;

public class Simulation {
    private final Config config;
    private final Logger logger;
    private final Random random;

    private int currentTick;
    private int totalParcelsGenerated;
    private int totalParcelsDispatched;
    private int totalParcelsReturned;

    private final ArrivalBuffer arrivalBuffer;
    private final ReturnStack returnStack;
    private final DestinationSorter destinationSorter;
    private final ParcelTracker parcelTracker;
    private final TerminalRotator terminalRotator;

    public Simulation(String configFile) throws IOException {
        this.config = new Config(configFile);
        this.logger = Logger.getInstance("log.txt", true);
        this.random = new Random();

        int queueCapacity = config.getInt("QUEUE_CAPACITY", 30);
        this.arrivalBuffer = new ArrivalBuffer(queueCapacity);
        this.returnStack = new ReturnStack();
        this.destinationSorter = new DestinationSorter();
        this.parcelTracker = new ParcelTracker(100);

        String[] cities = config.getStringArray("CITY_LIST", new String[]{"Istanbul", "Ankara", "Izmir"});
        this.terminalRotator = new TerminalRotator(cities);

        this.currentTick = 0;
        this.totalParcelsGenerated = 0;
        this.totalParcelsDispatched = 0;
        this.totalParcelsReturned = 0;
    }
```

```
public void run() {
    int maxTicks = config.getInt("MAX_TICKS", 300);
    Boolean useTerminalUI = true;

    while (currentTick < maxTicks) {
        currentTick++;
        if (useTerminalUI) {
        TerminalUI.displayDashboard(arrivalBuffer, destinationSorter, returnStack, terminalRotator);
    }

        logger.log("\n[Tick " + currentTick + "]");

        generateParcels();

        processArrivalBuffer();

        dispatchParcels();

        if (currentTick % 3 == 0) {
            processReturnStack();
        }

        if (currentTick % config.getInt("TERMINAL_ROTATION_INTERVAL", 5) == 0) {
            terminalRotator.advanceTerminal();
            logger.log("Rotated to: " + terminalRotator.getActiveTerminal());
        }

        logTickSummary();
        try {
            Thread.sleep(0);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    generateFinalReport();
    logger.close();
}

private void generateParcels() {
    int minParcels = config.getInt("PARCEL_PER_TICK_MIN", 1);
    int maxParcels = config.getInt("PARCEL_PER_TICK_MAX", 3);
    int numParcels = minParcels + random.nextInt(maxParcels - minParcels + 1);

    if (numParcels > 0) {
        logger.log("New Parcels:");
    }

    for (int i = 0; i < numParcels; i++) {
        String parcelID = "P" + (1000 + totalParcelsGenerated + i);
        String[] cities = config.getStringArray("CITY_LIST", new String[]{"Istanbul", "Ankara", "Izmir"});
        String destination = cities[random.nextInt(cities.length)];
        ParcelPriority priority = ParcelPriority.values()[random.nextInt(3)];
        ParcelSize size = ParcelSize.values()[random.nextInt(3)];

        Parcel parcel = new Parcel(parcelID, destination, priority, size, currentTick);
        arrivalBuffer.enqueue(parcel);
        parcelTracker.insert(parcelID, parcel);

        logger.log("  " + parcelID + " to " + destination + " (Priority " + priority.getValue() + ")");
        totalParcelsGenerated++;
    }

    logger.log("Queue Size: " + arrivalBuffer.size());
}
```

```java
private void processArrivalBuffer() {
    if (!arrivalBuffer.isEmpty()) {
        Parcel parcel = arrivalBuffer.dequeue();
        if (parcel != null) {
            destinationSorter.insertParcel(parcel);
            parcel.setStatus(ParcelStatus.SORTED);
            parcelTracker.updateStatus(parcel.getParcelID(), ParcelStatus.SORTED);
            logger.log("Sorted to BST: " + parcel.getParcelID() + " to " + parcel.getDestinationCity());
        }
    }
}

private void dispatchParcels() {
    String activeTerminal = terminalRotator.getActiveTerminal();
    LinkedList cityParcels = destinationSorter.getCityParcels(activeTerminal);

    if (cityParcels != null && cityParcels.size() > 0) {
        // Priority-based selection
        Parcel parcelToDispatch = selectByPriority(cityParcels);

        double misrouteRate = config.getDouble("MISROUTING_RATE", 0.1);
        boolean misrouted = random.nextDouble() < misrouteRate;

        if (misrouted) {
            returnStack.push(parcelToDispatch);
            parcelToDispatch.setStatus(ParcelStatus.RETURNED);
            parcelTracker.updateStatus(parcelToDispatch.getParcelID(), ParcelStatus.RETURNED);
            parcelTracker.incrementReturnCount(parcelToDispatch.getParcelID());
            logger.log("Returned: " + parcelToDispatch.getParcelID() + " misrouted -> Pushed to ReturnStack");
            totalParcelsReturned++;
        } else {
            destinationSorter.removeParcel(activeTerminal, parcelToDispatch.getParcelID());
            parcelToDispatch.setStatus(ParcelStatus.DISPATCHED);
            parcelTracker.updateStatus(parcelToDispatch.getParcelID(), ParcelStatus.DISPATCHED);
            logger.log("Dispatched: " + parcelToDispatch.getParcelID() + " from BST to " + activeTerminal + " -> Success");
            totalParcelsDispatched++;
        }
    }
}

private Parcel selectByPriority(LinkedList parcels) {
    LinkedList.Node current = parcels.getHead();
    while (current != null) {
        Parcel p = current.getValue();
        if (p.getPriority() == ParcelPriority.HIGH) {
            return p;
        }
        current = current.getNext();
    }

    current = parcels.getHead();
    while (current != null) {
        Parcel p = current.getValue();
        if (p.getPriority() == ParcelPriority.MEDIUM) {
            return p;
        }
        current = current.getNext();
    }

    if (parcels.getHead() != null) {
        return parcels.getHead().getValue();
    }

    return null;
```

```java
}



private void processReturnStack() {
    if (!returnStack.isEmpty()) {
        Parcel parcel = returnStack.pop();
        if (parcel != null) {
            destinationSorter.insertParcel(parcel);
            parcel.setStatus(ParcelStatus.SORTED);
            parcelTracker.updateStatus(parcel.getParcelID(), ParcelStatus.SORTED);
            logger.log("Reprocessed from ReturnStack: " + parcel.getParcelID());
        }
    }
}

private void logTickSummary() {
    logger.log("Active Terminal: " + terminalRotator.getActiveTerminal());
    logger.log("ReturnStack Size: " + returnStack.size());
    logger.log("BST Stats: Nodes=" + destinationSorter.getNodeCount() + ", Height=" + destinationSorter.getHeight());

}

private void generateFinalReport() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter("report.txt"))) {
        writer.write("=== ParcelSortX Simulation Report ===\n");
        writer.write("Generated at: " + LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")) + "\n\n");

        writer.write("1. Simulation Overview\n");
        writer.write("   - Total Ticks Executed: " + currentTick + "\n");
        writer.write("   - Number of Parcels Generated: " + totalParcelsGenerated + "\n\n");

        writer.write("2. Parcel Statistics\n");
        writer.write("   - Total Dispatched Parcels: " + totalParcelsDispatched + "\n");
        writer.write("   - Total Returned Parcels: " + totalParcelsReturned + "\n");
        writer.write("   - Parcels in Queue at End: " + arrivalBuffer.size() + "\n");
        writer.write("   - Parcels in BST at End: " + getTotalParcelsInBST() + "\n");
        writer.write("   - Parcels in ReturnStack at End: " + returnStack.size() + "\n\n");

        writer.write("3. Data Structure Statistics\n");
        writer.write("   - Maximum Queue Size Observed: " + arrivalBuffer.getCapacity() + "\n");
        writer.write("   - Maximum Stack Size Observed: " + returnStack.getMaxSize() + "\n");
        writer.write("   - Final Height of BST: " + destinationSorter.getHeight() + "\n");
        writer.write("   - Hash Table Load Factor: " + String.format("%.2f", parcelTracker.getLoadFactor()) + "\n");

    } catch (IOException e) {
        System.err.println("Error writing final report: " + e.getMessage());
    }
}

private int getTotalParcelsInBST() {
    return 0;
}

private int getMaxStackSize() {
    return returnStack.size();
}

public static void main(String[] args) {
    try {
        Simulation simulation = new Simulation("config.txt");
        simulation.run();
    } catch (IOException e) {
        System.err.println("Failed to initialize simulation: " + e.getMessage());
        }
    }
}

// ParcelStatus.java
package model.enums;

public enum ParcelStatus {
    IN_QUEUE("InQueue"),
    SORTED("Sorted"),
    DISPATCHED("Dispatched"),
    RETURNED("Returned");

    private final String status;

    private ParcelStatus(String status) {
        this.status = status;
    }

    @Override
    public String toString() {
        return status;
    }
}

// ParcelPriority.java
package model.enums;

public enum ParcelPriority {
    LOW(1),
    MEDIUM(2),
    HIGH(3);

    private final int value;

    ParcelPriority(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public static ParcelPriority fromValue(int value) {
        for (ParcelPriority priority : ParcelPriority.values()) {
            if (priority.getValue() == value) {
                return priority;
            }
        }
        throw new IllegalArgumentException("Invalid priority value: " + value);
    }
}

// ParcelSize.java
package model.enums;

public enum ParcelSize {
    SMALL("Small"),
    MEDIUM("Medium"),
    LARGE("Large");

    private final String displayName;

    private ParcelSize(String displayName) {
        this.displayName = displayName;
    }
    @Override
    public String toString() {
        return displayName;
    }
}



package model;

import model.enums.ParcelPriority;
import model.enums.ParcelSize;
import model.enums.ParcelStatus;

public class Parcel {
    private String parcelID;
    private String destinationCity;
    private ParcelPriority priority;
    private ParcelSize size;
    private int arrivalTick;
    private ParcelStatus status;

    public Parcel(String parcelID, String destinationCity, ParcelPriority priority,
                  ParcelSize size, int arrivalTick) {
        this.parcelID = parcelID;
        this.destinationCity = destinationCity;
        this.priority = priority;
        this.size = size;
        this.arrivalTick = arrivalTick;
        this.status = ParcelStatus.IN_QUEUE;
    }

    public String getParcelID() { return parcelID; }
    public String getDestinationCity() { return destinationCity; }
    public ParcelPriority getPriority() { return priority; }
    public ParcelSize getSize() { return size; }
    public int getArrivalTick() { return arrivalTick; }
    public ParcelStatus getStatus() { return status; }
    public void setStatus(ParcelStatus status) { this.status = status; }

    @Override
    public String toString() {
        return String.format("ParcelID: %s, Destination: %s, Priority: %s, Size: %s, ArrivalTick: %d",
                parcelID, destinationCity, priority, size, arrivalTick);
    }

}

package structures;

import model.Parcel;

public class LinkedList {
    private Node head;

    public LinkedList() {
        head = null;
    }

    public void add(Parcel parcel) {
        Node newNode = new Node(parcel);
        if (head == null) {
            head = newNode;
        } else {
            Node curr = head;
            while (curr.getNext() != null) {
                curr = curr.getNext();
            }
            curr.setNext(newNode);
        }
    }
    public Node getHead() {
        return head;
    }


    public void remove(Parcel parcel) {
        Node prev = null;
        Node curr = head;
        while (curr != null && !curr.getValue().getParcelID().equals(parcel.getParcelID())) {
            prev = curr;
            curr = curr.getNext();
        }
        if (curr != null) {
            if (prev == null) {
                head = curr.getNext();
            } else {
                prev.setNext(curr.getNext());
            }
        }
    }

    public Node search(String parcelID) {
        Node curr = head;
        while (curr != null && !curr.getValue().getParcelID().equals(parcelID)) {
            curr = curr.getNext();
        }
        return curr;
    }

    public void insert(Parcel parcel, int pos) {
        Node newNode = new Node(parcel);
        if (pos == 0) {
            newNode.setNext(head);
            head = newNode;
        } else {
            Node prev = null;
            Node curr = head;
            for (int i = 0; i < pos && curr != null; i++) {
                prev = curr;
                curr = curr.getNext();
            }
            if (curr != null) {
                prev.setNext(newNode);
                newNode.setNext(curr);
            } else {
                prev.setNext(newNode);
            }
        }
    }

    public void reverse() {
        Node prev = null;
        Node curr = head;
        while (curr != null) {
            Node next = curr.getNext();
            curr.setNext(prev);
            prev = curr;
            curr = next;
        }
        head = prev;
    }

    public int size() {
        int count = 0;
        Node curr = head;
        while (curr != null) {
            count++;
            curr = curr.getNext();
        }
        return count;
    }

    public static class Node {
        private Parcel value;
        private Node next;

        public Node(Parcel value) {
            this.value = value;
            this.next = null;
        }

        public Parcel getValue() {
            return value;
        }

        public Node getNext() {
            return next;
        }

        public void setNext(Node next) {
            this.next = next;
        }
    }
}


package structures;

import model.Parcel;
import utils.Logger;

import java.io.IOException;

public class ArrivalBuffer {

    private static class Node {
        Parcel data;
        Node next;

        Node(Parcel data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node front;
    private Node rear;
    private int size;
    private final int capacity;
    private final Logger logger;

    public ArrivalBuffer(int capacity) {
        this.capacity = capacity;
        this.front = null;
        this.rear = null;
        this.size = 0;

        Logger tempLogger;
        try {
            tempLogger = Logger.getInstance("log.txt", true);
        } catch (IOException e) {
            throw new RuntimeException("Logger init failed", e);
        }
        this.logger = tempLogger;
    }

    public void enqueue(Parcel parcel) {
        if (isFull()) {
            logger.log("WARNING: Queue Overflow! Parcel " + parcel.getParcelID() + " to " + parcel.getDestinationCity() + " discarded.");
            return;
        }

        Node newNode = new Node(parcel);
        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        size++;
    }

    public Parcel dequeue() {
        if (isEmpty()) return null;
        Parcel parcel = front.data;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        size--;
        return parcel;
    }

    public Parcel peek() {
        return (front != null) ? front.data : null;
    }

    public boolean isFull() {
        return size >= capacity;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }

    public int getCapacity() {
        return capacity;
    }

    public void visualizeQueue() {
        System.out.println("ArrivalBuffer Queue [size=" + size + "/" + capacity + "]:");
        Node current = front;
        while (current != null) {
            Parcel parcel = current.data;
            System.out.print("[" + parcel.getParcelID() + " to " + parcel.getDestinationCity() + "] -> ");
            current = current.next;
        }
        System.out.println("NULL");
    }
}

package structures;

import model.Parcel;

public class DestinationSorter {

    private class CityNode {
        String cityName;
        LinkedList parcelList;
        CityNode left, right;
        int height;

        CityNode(String cityName) {
            this.cityName = cityName;
            this.parcelList = new LinkedList();
            this.left = this.right = null;
            this.height = 1;
        }
    }

    private CityNode root;
    private int nodeCount;

    private int height(CityNode node) {
        return node == null ? 0 : node.height;
    }

    private int getBalance(CityNode node) {
        return node == null ? 0 : height(node.left) - height(node.right);
    }

    private CityNode rightRotate(CityNode y) {
        CityNode x = y.left;
        CityNode T2 = x.right;

        x.right = y;
        y.left = T2;

        y.height = Math.max(height(y.left), height(y.right)) + 1;
        x.height = Math.max(height(x.left), height(x.right)) + 1;

        return x;
    }

    private CityNode leftRotate(CityNode x) {
        CityNode y = x.right;
        CityNode T2 = y.left;

        y.left = x;
        x.right = T2;

        x.height = Math.max(height(x.left), height(x.right)) + 1;
        y.height = Math.max(height(y.left), height(y.right)) + 1;

        return y;
    }

    public void insertParcel(Parcel parcel) {
```

```
    root = insertParcel(root, parcel);
}

private CityNode insertParcel(CityNode node, Parcel parcel) {
    if (node == null) {
        nodeCount++;
        CityNode newNode = new CityNode(parcel.getDestinationCity());
        newNode.parcelList.add(parcel);
        return newNode;
    }

    int cmp = parcel.getDestinationCity().compareTo(node.cityName);
    if (cmp < 0) {
        node.left = insertParcel(node.left, parcel);
    } else if (cmp > 0) {
        node.right = insertParcel(node.right, parcel);
    } else {
        node.parcelList.add(parcel);
        return node;
    }

    node.height = 1 + Math.max(height(node.left), height(node.right));

    int balance = getBalance(node);

    if (balance > 1 && parcel.getDestinationCity().compareTo(node.left.cityName) < 0) {
        return rightRotate(node);
    }

    if (balance < -1 && parcel.getDestinationCity().compareTo(node.right.cityName) > 0) {
        return leftRotate(node);
    }

    if (balance > 1 && parcel.getDestinationCity().compareTo(node.left.cityName) > 0) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    if (balance < -1 && parcel.getDestinationCity().compareTo(node.right.cityName) < 0) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

public LinkedList getCityParcels(String city) {
    CityNode node = findNode(root, city);
    return node != null ? node.parcelList : null;
}

private CityNode findNode(CityNode node, String city) {
    if (node == null) {
        return null;
    }

    int cmp = city.compareTo(node.cityName);
    if (cmp < 0) {
        return findNode(node.left, city);
    } else if (cmp > 0) {
        return findNode(node.right, city);
    } else {
        return node;
    }
}
```

```
    public Parcel removeParcel(String city, String parcelID) {
        CityNode node = findNode(root, city);
        if (node == null) {
            return null;
        }

        LinkedList.Node parcelNode = node.parcelList.search(parcelID);
        if (parcelNode != null) {
            Parcel parcel = parcelNode.getValue();
            node.parcelList.remove(parcel);
            return parcel;
        }
        return null;
    }

    public int countCityParcels(String city) {
        CityNode node = findNode(root, city);
        return node != null ? node.parcelList.size() : 0;
    }

    public int getNodeCount() {
        return nodeCount;
    }

    public int getHeight() {
        return calculateHeight(root);
    }

    private int calculateHeight(CityNode node) {
        if (node == null) {
            return 0;
        }
        return 1 + Math.max(calculateHeight(node.left), calculateHeight(node.right));
    }

    public void inOrderTraversal() {
        inOrderTraversal(root);
    }

    private void inOrderTraversal(CityNode node) {
        if (node != null) {
            inOrderTraversal(node.left);
            System.out.println("City: " + node.cityName + " - Parcels: " + node.parcelList.size());
            inOrderTraversal(node.right);
        }
    }

    public void visualizeBST() {
        System.out.println("\nDestination Sorter (BST):");
        visualizeBST(root, 0);
    }

    private void visualizeBST(CityNode node, int level) {
        if (node != null) {
            visualizeBST(node.right, level + 1);
            for (int i = 0; i < level; i++) System.out.print("    ");
            System.out.printf("%s (%d parcels)\n", node.cityName, node.parcelList.size());
            visualizeBST(node.left, level + 1);
        }
    }


}
```

```
// ParcelTracker.java
//hash table
// ParcelTracker.java
package structures;

import model.Parcel;
import model.enums.ParcelStatus;
import java.util.ArrayList;
import java.util.List;
import model.enums.ParcelPriority;
import model.enums.ParcelSize;

public class ParcelTracker {
    private static class HashNode {
        String parcelID;
        ParcelStatus status;
        int arrivalTick;
        Integer dispatchTick;
        int returnCount;
        String destinationCity;
        int priority;
        String size;
        HashNode next;

        HashNode(String parcelID, Parcel parcel) {
            this.parcelID = parcelID;
            this.status = parcel.getStatus();
            this.arrivalTick = parcel.getArrivalTick();
            this.dispatchTick = null;
            this.returnCount = 0;
            this.destinationCity = parcel.getDestinationCity();
            this.priority = parcel.getPriority().getValue();
            this.size = parcel.getSize().toString();
            this.next = null;
        }
    }

    private int capacity;
    private final List<HashNode> buckets;
    private int size;
    private final double loadFactorThreshold = 0.75;

    public ParcelTracker(int initialCapacity) {
        this.capacity = initialCapacity;
        this.buckets = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            buckets.add(null);
        }
        this.size = 0;
    }

    private int hash(String key) {
        return Math.abs(key.hashCode()) % capacity;
    }

    public void insert(String parcelID, Parcel parcel) {
        if (exists(parcelID)) {
            return;
        }

        int bucketIndex = hash(parcelID);
        HashNode newNode = new HashNode(parcelID, parcel);
        HashNode head = buckets.get(bucketIndex);

        if (head == null) {
            buckets.set(bucketIndex, newNode);
        } else {
            newNode.next = head;
            buckets.set(bucketIndex, newNode);
        }
        size++;

        if ((1.0 * size) / capacity >= loadFactorThreshold) {
            resize();
        }
    }

    private void resize() {
        List<HashNode> temp = buckets;
        buckets.clear();
        capacity *= 2;
        for (int i = 0; i < capacity; i++) {
            buckets.add(null);
        }
        size = 0;

        for (HashNode head : temp) {
            while (head != null) {
                insert(head.parcelID, createParcelFromNode(head));
                head = head.next;
            }
        }
    }

    private Parcel createParcelFromNode(HashNode node) {
        return new Parcel(node.parcelID, node.destinationCity,
                    ParcelPriority.values()[node.priority - 1],
                    ParcelSize.valueOf(node.size.toUpperCase()),
                    node.arrivalTick);
    }

    public void updateStatus(String parcelID, ParcelStatus newStatus) {
        HashNode node = getNode(parcelID);
        if (node != null) {
            node.status = newStatus;
        }
    }

    public HashNode get(String parcelID) {
        return getNode(parcelID);
    }

    private HashNode getNode(String parcelID) {
        int bucketIndex = hash(parcelID);
        HashNode head = buckets.get(bucketIndex);

        while (head != null) {
            if (head.parcelID.equals(parcelID)) {
                return head;
            }
            head = head.next;
        }
        return null;
    }

    public void incrementReturnCount(String parcelID) {
        HashNode node = getNode(parcelID);
        if (node != null) {
            node.returnCount++;
        }
    }

    public boolean exists(String parcelID) {
        return getNode(parcelID) != null;
    }

    public int size() {
        return size;
    }

    public double getLoadFactor() {
        return (double) size / capacity;
    }

}

// TerminalRotator.java
//circular linked list
package structures;

public class TerminalRotator {
    private static class TerminalNode {
        String cityName;
        TerminalNode next;

        TerminalNode(String cityName) {
            this.cityName = cityName;
            this.next = null;
        }
    }

    private TerminalNode head;
    private TerminalNode currentActiveTerminal;
    private int size;

    public TerminalRotator(String[] cities) {
        if (cities == null || cities.length == 0) {
            throw new IllegalArgumentException("City list cannot be empty");
        }

        // Create the circular linked list
        head = new TerminalNode(cities[0]);
        TerminalNode current = head;
        size = 1;

        for (int i = 1; i < cities.length; i++) {
            current.next = new TerminalNode(cities[i]);
            current = current.next;
            size++;
        }

        current.next = head;
        currentActiveTerminal = head;
    }

    public void advanceTerminal() {
        if (currentActiveTerminal != null) {
            currentActiveTerminal = currentActiveTerminal.next;
        }
    }

    public String getActiveTerminal() {
        return currentActiveTerminal != null ? currentActiveTerminal.cityName : null;
    }
    public void printTerminalOrder() {
        if (head == null) {
            return;
        }

        TerminalNode current = head;
        do {
            System.out.print(current.cityName);
            if (current == currentActiveTerminal) {
                System.out.print(" (Active)");
            }
            System.out.print(" -> ");
            current = current.next;
        } while (current != head);
        System.out.println("...");
    }

    public int size() {
        return size;
    }
}

// ReturnStack.java
package structures;

import model.Parcel;

public class ReturnStack {
    private static class Node {
        Parcel parcel;
        Node next;

        Node(Parcel parcel) {
            this.parcel = parcel;
            this.next = null;
        }
    }

    private Node top;
    private int size;
    private int maxSize = 0;// sonradan eklendi

    public ReturnStack() {
        this.top = null;
        this.size = 0;
    }

    public void push(Parcel parcel) {
        Node newNode = new Node(parcel);
        newNode.next = top;
        top = newNode;
        size++;

        // Update maxSize
        if (size > maxSize) {
            maxSize = size;
        }
    }

    public Parcel pop() {
        if (isEmpty()) {
            return null;
        }

        Parcel parcel = top.parcel;
        top = top.next;
        size--;
        return parcel;
    }

    public Parcel peek() {
        if (isEmpty()) {
            return null;
        }
        return top.parcel;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public int size() {
        return size;
    }
    public int getMaxSize() {
        return maxSize;
    }

}

package structures;

public class HashTable<K, V> {
    private int capacity;
    private LinkedList<Entry<K, V>>[] table;
    private int size;

    public HashTable(int capacity) {
        this.capacity = capacity;
        this.table = new LinkedList[capacity];
        for (int i = 0; i < capacity; i++) {
            table[i] = new LinkedList<>();
        }
        this.size = 0;
    }

    private int hash(K key) {
        return Math.abs(key.hashCode()) % capacity;
    }

    public void put(K key, V value) {
        int index = hash(key);
        LinkedList<Entry<K, V>> bucket = table[index];
        for (int i = 0; i < bucket.size(); i++) {
            Entry<K, V> entry = bucket.get(i);
            if (entry.getKey().equals(key)) {
                entry.setValue(value);
                return;
            }
        }
        bucket.add(new Entry<>(key, value));
        size++;
    }

    public V get(K key) {
        int index = hash(key);
        LinkedList<Entry<K, V>> bucket = table[index];
        for (int i = 0; i < bucket.size(); i++) {
            Entry<K, V> entry = bucket.get(i);
            if (entry.getKey().equals(key)) {
```

```
            return entry.getValue();
        }
    }
    return null;
}

public void remove(K key) {
    int index = hash(key);
    LinkedList<Entry<K, V>> bucket = table[index];
    for (int i = 0; i < bucket.size(); i++) {
        if (bucket.get(i).getKey().equals(key)) {
            bucket.remove(i);
            size--;
            return;
        }
    }
}

public boolean containsKey(K key) {
    return get(key) != null;
}

public int size() {
    return size;
}

private static class Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }
}

private static class LinkedList<T> {
    private Node<T> head;
    private int size;

    public void add(T data) {
        if (head == null) {
            head = new Node<>(data);
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new Node<>(data);
        }
        size++;
    }
```

```
        public T get(int index) {
            Node<T> current = head;
            for (int i = 0; i < index; i++) {
                current = current.next;
            }
            return current.data;
        }

        public void remove(int index) {
            if (index == 0) {
                head = head.next;
            } else {
                Node<T> current = head;
                for (int i = 0; i < index - 1; i++) {
                    current = current.next;
                }
                current.next = current.next.next;
            }
            size--;
        }

        public int size() {
            return size;
        }
    }

    private static class Node<T> {
        T data;
        Node<T> next;

        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }
    public double getLoadFactor() {
        return (double) size / capacity;
    }

}

// Logger.java
package utils;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Logger {
    private static Logger instance;
    private BufferedWriter logWriter;
    private final boolean consoleOutput;

    private Logger(String logFile, boolean consoleOutput) throws IOException {
        this.logWriter = new BufferedWriter(new FileWriter(logFile, true));
        this.consoleOutput = consoleOutput;
        log("=== Simulation started at " + LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME) + " ===");
    }

    public static synchronized Logger getInstance(String logFile, boolean consoleOutput) throws IOException {
        if (instance == null) {
            instance = new Logger(logFile, consoleOutput);
        }
        return instance;
    }

    public void log(String message) {
        try {
            logWriter.write(message);
            logWriter.newLine();
            logWriter.flush();
        } catch (IOException e) {
            System.err.println("Error writing to log file: " + e.getMessage());
        }

        if (consoleOutput) {
            System.out.println(message);
        }
    }

    public void close() {
        try {
            logWriter.close();
        } catch (IOException e) {
            System.err.println("Error closing log file: " + e.getMessage());
        }
    }
}

// Config.java
package utils;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import structures.HashTable; // bizim elle yazdığımız HashTable'ı import ediyoruz

public class Config {
    private final HashTable<String, String> configMap;

    public Config(String configFile) {
        this.configMap = new HashTable<>(100); // kapasiteyi 100 verdim, değiştirebilirsin
        loadConfig(configFile);
    }

    private void loadConfig(String configFile) {
        try (BufferedReader reader = new BufferedReader(new FileReader(configFile))) {
            String line;
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty() || line.startsWith("#")) {
                    continue;
                }

                String[] parts = line.split("=", 2);
                if (parts.length == 2) {
                    configMap.put(parts[0].trim(), parts[1].trim());
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading config file: " + e.getMessage());
        }
    }

    public int getInt(String key, int defaultValue) {
        String value = configMap.get(key);
        if (value == null) {
            return defaultValue;
        }
        try {
            return Integer.parseInt(value);
        } catch (NumberFormatException e) {
            return defaultValue;
        }
    }

    public double getDouble(String key, double defaultValue) {
        String value = configMap.get(key);
        if (value == null) {
            return defaultValue;
        }
        try {
            return Double.parseDouble(value);
        } catch (NumberFormatException e) {
            return defaultValue;
        }
    }

    public String getString(String key, String defaultValue) {
        String value = configMap.get(key);
        if (value == null) {
            return defaultValue;
        }
        return value;
    }

    public String[] getStringArray(String key, String[] defaultValue) {
        String value = configMap.get(key);
        if (value == null) {
            return defaultValue;
        }
        return value.split("\\s*,\\s*");
    }
}

// TerminalUI.java
package utils;

import structures.*;

public class TerminalUI {
    public static void displayDashboard(ArrivalBuffer buffer, DestinationSorter sorter,
                                        ReturnStack stack, TerminalRotator rotator) {
        clearScreen();
        displayHeader();
        buffer.visualizeQueue();
        sorter.visualizeBST();
        displayStack(stack);
        displayTerminal(rotator);
        displaySeparator();
    }

    private static void clearScreen() {
        System.out.print("\033[H\033[2J");
        System.out.flush();
    }

    private static void displayHeader() {
        System.out.println("=== PARCEL SORTX SIMULATION ===");
        System.out.println("┌────────────────────────────────┐");
        System.out.println("│      LOGISTICS CENTER      │");
        System.out.println("└────────────────────────────────┘");
    }
    private static void displayStack(ReturnStack stack) {
        System.out.println("\nReturn Stack:");
        System.out.println("  ┌────────┐ ");
        if (stack.size() > 0) {
            System.out.println("  │ " + stack.peek().getParcelID() + " │");
        } else {
            System.out.println("  │ EMPTY │");
        }
        System.out.println("  └────────┘ ");
        System.out.println("Size: " + stack.size());
    }

    private static void displayTerminal(TerminalRotator rotator) {
        System.out.println("\nActive Terminal: [" + rotator.getActiveTerminal() + "]");
        System.out.println("  ╔════════════════╗ ");
        System.out.println("  ║   DISPATCH   ║");
        System.out.println("  ╚════════════════╝ ");
    }

    private static void displaySeparator() {
        System.out.println("\n────────────────────────────────────────");
    }
}
```