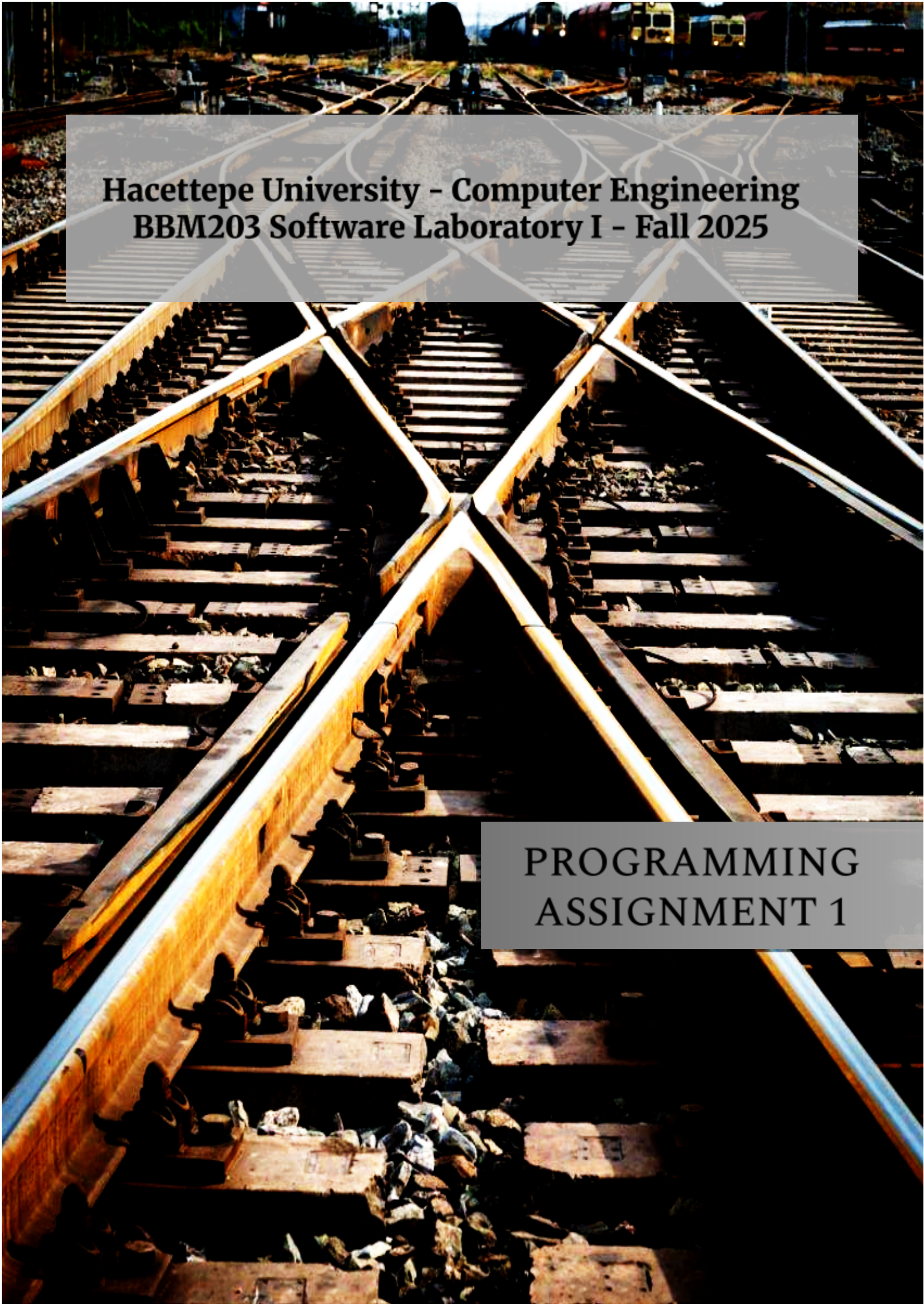


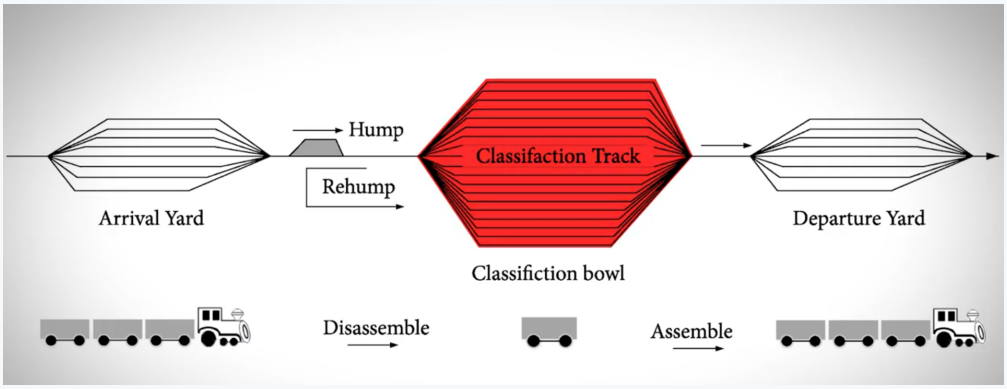
Hacettepe University - Computer Engineering
BBM203 Software Laboratory I - Fall 2025

PROGRAMMING ASSIGNMENT 1

PROGRAMMING ASSIGNMENT 1

Topics: Linked Lists, Doubly Linked Lists, Matrices
Dynamic Memory Allocation, File I/O

Programming Language: C++11
Due Date: **Friday, 07/11/2025 (23:59:59)** over **https://submit.cs.hacettepe.edu.tr**



Introduction and Background

H.U. Transportation Technologies (H.U.T.T.), has launched an ambitious modernization project for its national freight network. Their goal is to develop an intelligent Rail Marshal System that can automatically classify, assemble, and dispatch trains more efficiently across Turkey's major railway lines.

To achieve this, H.U.T.T. needs a working simulation that models how freight wagons are sorted and assembled in a marshaling yard, i.e. a prototype. And who better to build it than the talented interns from Hacettepe University?

In this simulation, wagons carrying different types of cargo (coal, livestock, hazardous materials, and more) will arrive at the classification yard. Each wagon must be sorted by its destination and cargo type, then grouped into temporary blocks before being assembled into complete trains. Once assembled, trains will move to the departure yard, where they wait in line for dispatch.

The executives have given you, the interns, two weeks to deliver this working prototype. Your program will model the core mechanisms of a real-world rail yard, including:

- Wagon sorting and organization by type and destination,
- Formation of trains from classified wagons,
- Management of departure tracks, and
- Proper dispatching according to rail yard rules.

The H.U.T.T. team is counting on your implementation to demonstrate how linked data structures can power an efficient, rule-based logistics system that mirrors real operations in railway networks. The fate of this modernization project rests on your code!

1 The Rail Marshal System

1.1 The Classification Yard

The classification yard is the heart of the marshaling process. This area is where incoming wagons are temporarily stored and organized based on their **destination** and **cargo type**. To represent this, the yard is implemented as a 2D matrix of wagon lists:

```
WagonList blockTrains[Destination][CargoType];
```

Each cell in the matrix holds a doubly linked list of wagons (`WagonList`) sharing both destination and cargo type. Within each list, wagons are kept **sorted by weight (descending)**, so heavier wagons are placed toward the front to ensure proper balance when the train is assembled.

When a train is ready to be formed, wagon lists corresponding to the same destination are merged into a single train, simulating how block trains are joined together in real marshaling yards.

1.2 The Train

A `Train` represents an assembled convoy of wagons that share a common destination. Each train internally holds a doubly linked list of wagons i.e. `WagonList`, allowing wagons to be easily added, rearranged, or detached when necessary.

Every train stores:

- A `name` (e.g., `Train_ANKARA_1`),
- Its `destination`,
- Its `total weight`, and
- A pointer to the `next train` in the departure track.

Trains are dynamically created when the classification yard combines wagons for a destination and can hold several cargo types. They are deleted once dispatched.

1.3 The Departure Yard

The departure yard holds trains waiting to be dispatched. It consists of an array of departure tracks, one per destination:

```
TrainTrack departureTracks[NUM_DESTINATIONS];
```

Each track behaves like a waiting line and trains are only added from the rear and only the first train in a track can depart, protecting the wait line.

Each `TrainTrack` maintains:

- Pointers to the `first` and `last` trains,
- The `total weight` of all trains on the track,
- A unique `index` (for destination identification).

When the total weight of a track exceeds a set threshold, or when a `DISPATCH` command is given, the first train in the track departs.

1.4 The Wagon and WagonList

Each `Wagon` object represents a single freight wagon in the system. Wagons contain:

- A `unique ID`,
- `Cargo type` (e.g., `COAL`, `MAIL`, `LIVESTOCK`, `HAZARDOUS`, `OTHER`),
- `Destination`, and
- `Weight` (in tons).
- `Maximum Coupler Load` (in tons).

Wagons are linked together via a doubly linked list managed by `WagonList`. Each `WagonList` provides essential operations such as:

- `insertSorted()` — inserts wagons in descending order of weight,
- `appendList()` — merges another list (used during train assembly),
- `clear()` and `detach()` — manage memory and selective removals.

During merging (i.e., `appendList()`), wagon lists that carry the same cargo type are kept together (protected) but are ordered according to the weight of their heaviest wagon. Except for Hazardous cargo which is always placed furthest away from the locomotive. See Section 2.2. for an example.

1.5 The Rail Marshal (Controller)

The `RailMarshal` class acts as the **main controller** of the entire simulation. It manages:

- A `ClassificationYard`,
- An array of `TrainTracks` (the departure yard),
- Command parsing and simulation flow.

The `RailMarshal` reads commands from an input file and executes operations such as:

- Adding new wagons to the yard,
- Forming trains,
- Dispatching trains from departure tracks, and
- Printing the current yard status.

It ensures all operational rules are respected, such as:

- Trains cannot be modified once in the departure yard, and
- Dispatching only happens from the front of each track.

1.6 Summary of System Flow

1. **Wagons arrive** → Added to the classification yard based on destination and cargo type.
2. **Sorting** → Wagons automatically sorted by weight within each category.
3. **Assembly** → Wagon groups for a destination are merged into a train.
4. **Waiting Line** → Formed trains are sent to their departure tracks.
5. **Dispatch** → Trains leave the yard following operational rules and capacity limits.

2 Commands and Simulation Rules

This section describes the available commands, their expected formats, and the rules governing the behavior of the system. All commands are provided through an **input file**, which the program reads and processes sequentially.

2.1 Command List

Command	Description
`ADD_WAGON <id> <cargoType> <destination> <weight> <maxCouplerLoad>`	Creates a new wagon and places it into the classification yard at the correct cell (`Destination`, `CargoType`). Wagons in each list are sorted in descending order of weight.
`REMOVE_WAGON <id>`	Removes the wagon with the given ID from the classification yard.
`ASSEMBLE_TRAIN <destination>`	Collects all wagons with the given destination, merges them into a new train, and moves it to the corresponding departure track.
`DISPATCH_TRAIN <destination>`	Dispatches the first train from the specified track in the departure yard.
`PRINT_YARD`	Prints the contents of both the classification yard and the departure yard.
`PRINT_TRACK <destination>`	Prints the trains waiting in the departure track of the given destination.
`AUTO_DISPATCH <on/off>`	Enables or disables automatic dispatching when a track exceeds a given weight threshold.
`CLEAR`	Clears all data and resets the simulation.

2.2 Classification Yard Rules

1. The classification yard is a matrix of wagon lists indexed by destination and cargo type. Each wagon is placed according to its properties.
2. Inside each list, wagons are ordered by **weight (descending)**. Heavier wagons appear at the front, lighter ones at the end.

3. When a train is assembled for a destination, cargo-type lists in that destination's row are merged into one train,
   - During merge weight and coupler load limits must be respected (see Section 2.3.)
   - Also during merge, wagon lists that carry the same cargo type must be kept together, but are ordered according to the weight of their heaviest wagon.
   - However, Hazardous cargo must always be placed furthest away from the locomotive, regardless of its weight.

Example:

```
Before merging (Classification Yard):

Destination: ANKARA
COAL:          [W12:90] - [W03:70]
LIVESTOCK:     [W07:120] - [W11:90]
MAIL:          [W05:75] - [W08:60] - [W18:40]
HAZARDOUS:     [W17:100] - [W01:90]

After merging into train (appendList()):
 LOCOMATIVE
     <-> [W07:120] - [W11:90](LIVESTOCK) <-> [W12:90] - [W03:70](COAL)
            <-> [W05:75] - [W08:60] - [W18:40](MAIL) <-> [W17:100](HAZARDOUS)

Explanation:
Wagon lists of the same cargo type remain grouped,
but their order in the final train is determined by
the heaviest wagon in each list (LIVESTOCK:120 > COAL:90 > MAIL:75).
HAZARDOUS is always placed to the end of the train and only a one can be carried
per train. Other hazardous wagon [W01] will wait for another train.
```

2.3 Train Formation Rules

1. Each formed train consists of wagons that share the **same destination**.
2. The train's name is generated automatically (e.g., `Train_ANKARA_1`, `Train_ANKARA_2`, ...).
3. Once a train is formed and moved to the departure yard, it **cannot be modified**.

Train Limits and Safety Rules

- A train can only hold one Hazardous cargo at a time, this must always be placed furthest from the locomotive (i.e. driver), regardless of its weight.

- Each wagon has a **maxCouplerLoad** value.
  - After a train is assembled, verify coupler safety **from tail to head**.
  - Maintain a cumulative total of trailing weight while moving backward through the train.
  - If the trailing weight exceeds a wagon's `maxCouplerLoad`, a **coupler overload** occurs.
  - When the first overload is found:
    - Split the train after that wagon.
    - The detached wagons form a **new train**, appended to the same track.
    - Print: `Train <name> split due to coupler overload at Wagon <id>`
  - The verification should be done all throughout the train and train should be split as many times as necessary.

2.4 Departure Yard Rules

1. The departure yard has one track per destination. Trains are added in the order they are formed.
2. Each track operates as a waiting line:
   - New trains are added to the rear,
   - Only the train at the front can be dispatched.
3. When a `DISPATCH <destination>` command is executed:
   - If the track is empty: `Error: No trains to dispatch from track <destination>.`
   - Otherwise, the train is removed and deleted:

     `Dispatching Train_ANKARA_1 (355 tons).`
4. If AUTO_DISPATCH is used
   When the total weight on a track exceeds a threshold (e.g., 2000 tons), the first train is automatically dispatched.

2.5 System-Wide Rules

1. All dynamic memory must be properly managed.
   - Wagons are deleted when trains are cleared.
   - Trains are deleted when dispatched.
2. `PRINT_YARD` and `PRINT_TRACK` must reflect the current system state accurately.
3. Any invalid command should be reported as:

   `Error: Unknown command '<command>'.`
4. Execution stops when the end of the input file is reached.

Implementation Notes

- You are provided with header and source templates defining all classes, structures, and enumerations required for the simulation (i.e. Starter Code).
- All linked data structures (`WagonList`, `TrainTrack`, and internal train connections) **must be implemented manually**. Usage of STL containers such as `std::list`, `std::vector`, or `std::map` **is not allowed**.
- You are expected to complete all functions marked with `// TODO` comments in the provided template files without modifying class or function names. Likewise, refrain from renaming or changing the types of the specified member variables. Other than that, you're free to introduce any additional functions or variables as needed.

3 Files: Input/Output

3.0.1 Input Format

The program will read commands from an input file provided as a command-line argument. Each line in the file represents a single command. Commands may include parameters such as wagon identifiers, destinations, cargo types, or train names. All keywords are written in uppercase and parameters are separated by spaces.

`Example:`

```
ADD_WAGON 101 COAL ANKARA 85 200
ADD_WAGON 102 HAZARDOUS ADANA 60 150
ASSEMBLE_TRAIN ANKARA
DISPATCH_TRAIN ANKARA
PRINT_YARD
```

3.0.2 Output Format

All outputs must be printed to the standard output (`stdout`) via **`std::cout`**. Do not use C-style output functions such as printf() or fprintf(). Each operation produces a specific message, which will be verified through automated testing. Deviations in text, spacing, or punctuation may result in lost points.

`Example Outputs (Not connected to example inputs above):`

```
Auto-dispatch: departing Train_ANKARA_2_split_2 to make room.
Train Train_ANKARA_2 assembled with W404(950ton) wagons.
[Track 0] Train_ANKARA_2(950ton)-W404(950ton) ->
Auto dispatch disabled
Train_ANKARA_1 departed from Track 0 (ANKARA).
Train Train_ESKISEHIR_1 departed from Track ESKISEHIR.
Dispatching Train_ESKISEHIR_1 (560 tons).
[Track 2] Train_ESKISEHIR_2(370ton)-W504(190ton) - W503(180ton) ->
Error: Unknown command 'FLY_TRAIN'
```

Must-Use Starter Code

You MUST use the starter code. All classes should be placed directly inside your **zip** archive.

Build and Run Configuration

Here is an example of how your code will be compiled (note that instead of `main.cpp` we will use our test files):

```
$ g++ -std=c++11  *.cpp, *.h -o HUTrain
```

Or, you can use the provided `Makefile` or `CMakeLists.txt` within the sample input to compile your code:

```
$ make all
```

or

```
$ mkdir HUTrain_build
$ cmake -S . -B HUTrain_build/
$ make -C HUTrain_build/
```

After compilation, you can run the program as follows:

```
$ ./bin/HUTrain input.txt
```

Grading Policy

- **No memory leaks and runtime errors: 10%**
  - No memory leaks: 5%
  - No memory errors: 5%
- **Correct implementation of core data structures: 45%**
  - Correct implementation of the `WagonList` (doubly linked list) 15%
  - Correct implementation of the `Train` class and wagon management 10%
  - Correct implementation of the `TrainTrack` wait line behavior 10%
  - Correct implementation of the `ClassificationYard` and sorting mechanism 10%
- **Correct implementation of commands and system actions: 35%**
  - Correct execution of `ADD_WAGON` 5%
  - Correct execution of `ASSEMBLE_TRAIN` 10%
  - Correct execution of `DISPATCH_TRAIN` 5%
  - Correct execution of `AUTO_DISPATCH` 7%
  - Enforcement of system constraints (weight limits, hazardous wagon restrictions, invalid inputs) – 8%
- **Output tests: 10%**

Note that you need to get a NON-ZERO grade from the assignment in order to get the submission accepted. Submissions with grade 0 will be counted as NO SUBMISSION!

Important Notes

- Do not miss the deadline: **Friday, 07/11/2025 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (`https://piazza.com/hacettepe.edu.tr/fall2025/bbm203`), and you are supposed to be aware of everything discussed on Piazza.
- You must test your code via **Tur^{3}Bo Grader** `https://test-grader.cs.hacettepe.edu.tr/` (**does not count as submission!**).
- You must submit your work via `https://submit.cs.hacettepe.edu.tr/` with the file hierarchy given below:
  - **b<studentID>.zip**
    - ClassificationYard.cpp <FILE>
    - ClassificationYard.h <FILE>
    - Train.cpp <FILE>
    - Train.h <FILE>
    - TrainTrack.cpp <FILE>
    - TrainTrack.h <FILE>
    - Wagon.cpp <FILE>
    - Wagon.h <FILE>
    - WagonList.cpp <FILE>
    - Wagonlist.h <FILE>
    - RailMarshal.cpp <FILE>
    - RailMarshal.h <FILE>
    - Enums.cpp <FILE>
    - Enums.h <FILE>
- **You MUST use the starter code**
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).
- Do not submit any object or executable files. Only header and C++ files are allowed.

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in the suspension of the involved students. Do not copy any code produced by LLM tools (e.g., ChatGPT, Gemini, Copilot) or any online source.