PROGRAMMING ASSIGNMENT 2

Topics: Stacks, Queues Dynamic Memory Allocation, File I/O

Programming Language: C++11
Due Date: **Friday, 05/12/2025 (23:59:59)** over **https://submit.cs.hacettepe.edu.tr**

Introduction and Background

The Turkish Disaster Response Agency (TDRA) is building an emergency coordination prototype called **QuakeAssist** to manage crisis operations after major earthquakes.

Immediately after a quake, two different types of requests flood the central system:

- **Supply Requests**: food, water, tents, medicine, fuel, etc.
- **Rescue Requests**: trapped people, collapsed buildings, critical evacuations.

TDRA has a limited number of field **teams**. Each team can work on both supply and rescue missions, but at any given moment, they must be dispatched to handle the **most emergent** request in the entire system, not just one queue.

To evaluate their strategies, TDRA needs a simulation that models:

- How incoming requests are managed in **two queues** (supply and rescue),
- How teams are assigned missions using **stack-based** mission tracking,
- How **emergency level** is computed and used to decide which queue is served,
- How overload situations trigger **rollback**, returning requests safely back to the queues.

You, the talented interns from Hacettepe University, are asked to build this prototype. Your implementation will demonstrate how stacks and queues can power an emergency coordination system that mimics essential aspects of real-world disaster management. The reliability of Quake-Assist—and the success of the simulation—rests on your code!

1 The QuakeAssist Emergency Coordination System

1.1 The Request Queues

QuakeAssist maintains two global queues of requests:

- A **supply queue** for logistics tasks (water, food, tents, fuel, etc.),
- A **rescue queue** for search-and-rescue tasks (trapped people, risky buildings).

Both queues are implemented using dynamically allocated **circular arrays**:

```
RequestQueue supplyQueue;
RequestQueue rescueQueue;
```

Each `RequestQueue` internally stores:

- A pointer to a dynamic array of `Request` objects,
- The current `front` and `rear` indices,
- The current `count` of elements, and
- The underlying `capacity`.

Your queue implementation must support:

- `enqueue()` — insert at the rear (with circular wrap-around),
- `dequeue()` — remove from the front,
- `peek()` — read the front element without removing it,
- Automatic **resizing** when the queue becomes full (e.g. doubling capacity).

No STL containers (such as `std::queue` or `std::vector`) are allowed; all behavior must be implemented manually.

1.2 Requests: Supply and Rescue

Each `Request` represents one emergency request in the system and contains:

- A **unique ID** (e.g., `S101`, `R205`) stored as `std::string`,
- A **type** field (string) that is either `"SUPPLY"` or `"RESCUE"`,
- A **city** field (string) such as `"ANKARA"`, `"IZMIR"`, `"VAN"`, . . . ,
- An integer `emergencyLevel` in the range [1, 5], where 5 is the most urgent.

In addition, supply and rescue requests have different fields.

For **SUPPLY** requests:

- A `supplyType` string (e.g., `"WATER"`, `"FOOD"`, `"MEDICINE"`, `"TENT"`, `"FUEL"`, `"OTHER"`),
- `amount` (integer; units or pallets of supply).

For **RESCUE** requests:

- `numPeople` (integer; people to be rescued),
- A `risk` string (e.g., `"LOW"`, `"MEDIUM"`, `"HIGH"`).

These fields will be used when computing emergency priority and team workload. In the starter code, these textual properties are stored as `std::string` and must be compared using string comparisons (e.g. `if (risk == "HIGH") { ... }`).

1.3 Teams and Mission Stacks

TDRA uses a configurable number of response **teams**. The number of teams and their capacities are determined at runtime by commands in the input file (see Section 2).

Each `Team` has:

- A `teamID`,
- A `maxLoadCapacity` (maximum workload the team can handle in a single mission),
- A `MissionStack` that stores the requests assigned to the team in the current mission.

The **mission stack** is implemented using a dynamically allocated array and a top index:

```
class MissionStack {
    Request* data;
    int capacity;
    int top;  // -1 when empty
    ...
};
```

The stack must support:

- `push()` — pushes a new `Request` on top,
- `pop()` — removes and returns the top request,
- `peek()` — returns the top request without removing it,
- Automatic **resizing** when full,
- `clear()` — empties the stack.

This stack represents the currently loaded mission for a team and is used to support rollback when overload occurs.

1.4 Emergency Decision Engine

Teams do not choose which queue to serve manually. Instead, QuakeAssist has an **emergency decision engine** that compares the front of each queue and selects the **globally most emergent** request.

For each non-empty queue, you consider its front request:

- `frontSupply` for the supply queue,
- `frontRescue` for the rescue queue.

An `emergencyScore` is computed as follows:

- For **SUPPLY** requests:

$$\text{emergencyScore} = \text{emergencyLevel} \times 10 + \min(\text{amount}, 50)$$

- For **RESCUE** requests:

$$\text{emergencyScore} = \text{emergencyLevel} \times 10 + \text{numPeople} \times \text{riskMultiplier}$$

where

- `riskMultiplier` is 1 for risk `"LOW"`, 2 for `"MEDIUM"`, and 3 for `"HIGH"`.

Decision rules:

- The request with the higher `emergencyScore` is chosen.
- If scores are equal, **RESCUE** requests take priority over SUPPLY.
- If one queue is empty, the request must be taken from the other queue.

This engine is used by the `HANDLE_EMERGENCY` command when assigning missions to teams.

1.5 The QuakeAssist Controller

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

- The global `supplyQueue` and `rescueQueue`,
- The dynamic array of `Teams` and their mission stacks,
- Command parsing and simulation flow.

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

- Initializing the number of teams and configuring their capacities,
- Adding new supply and rescue requests,
- Removing existing requests,
- Assigning teams to emergency missions using both queues,
- Dispatching teams and printing current system status.

It is responsible for enforcing all operational rules described in this document.

1.6 Summary of System Flow

1. **Requests arrive** $\rightarrow$ Added to the appropriate queue (supply or rescue).
2. **Emergency handling** $\rightarrow$ Teams are assigned using the emergency decision engine that compares both queues.

3. **Stack loading** → Selected requests are pushed onto the team's mission stack and workload is accumulated.
4. **Overload check** → If the team capacity is exceeded, the mission is rolled back and requests are returned to the queues.
5. **Dispatch** → When a mission is accepted and completed, the team is dispatched and its stack is cleared.

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**
`INIT_TEAMS <numTeams>`	Initializes the system with `numTeams` teams. Any previously existing teams are cleared. This command must be called before assigning missions.
`SET_TEAM_CAPACITY <teamID> <maxLoadCapacity>`	Sets the maximum load capacity for the specified team. This value is used during `HANDLE_EMERGENCY` to detect overload situations.
`ADD_SUPPLY <id> <city> <supplyType> <amount> <emergencyLevel>`	Creates a new SUPPLY request and enqueues it into the global `supplyQueue`. All fields are read as strings or integers (no enum types are used).
`ADD_RESCUE <id> <city> <numPeople> <buildingRisk> <emergencyLevel>`	Creates a new RESCUE request and enqueues it into the global `rescueQueue`. `buildingRisk` is given as a string: `"LOW"`, `"MEDIUM"`, or `"HIGH"`.
`REMOVE_REQUEST <id>`	Searches both queues for the given request ID and removes it if found. If not found, prints an error.
`HANDLE_EMERGENCY <teamID> <k>`	Assigns up to `k` requests to the given team by repeatedly selecting the most emergent request from the fronts of both queues, pushing them onto the team's mission stack, and updating workload. If overload occurs, a rollback is performed.
`DISPATCH_TEAM <teamID>`	If the specified team has an active mission (non-empty stack), prints a summary and clears the stack. Otherwise, prints an error.
`PRINT_QUEUES`	Prints the current contents of both the supply and rescue queues from front to rear, one request per line.
`PRINT_TEAM <teamID>`	Prints the current mission stack contents for the specified team from top to bottom.
`CLEAR`	Clears both queues and all teams' mission stacks, resetting the simulation.

2.2 Queue and Stack Rules

1. The system maintains two global queues:
   - `supplyQueue` for SUPPLY requests,
   - `rescueQueue` for RESCUE requests.

   Both must be implemented using dynamic circular arrays without STL containers.
2. All queue operations must behave as FIFO:
   - New requests are always enqueued at the rear,
   - Requests are always dequeued from the front.
3. Each team's mission stack must behave as LIFO:
   - The last request pushed is the first one popped,
   - Stack resizing must preserve existing elements.
4. When `HANDLE_EMERGENCY` is executed, the decision which queue to take from is made by the **emergency decision engine** at each step, always using the front elements of non-empty queues.
5. The implementation must ensure there are no out-of-bounds accesses when resizing arrays for queues or stacks.

Example (Conceptual):

```
Supply queue front:
    S101 ANKARA WATER amount=40 emergencyLevel=4

Rescue queue front:
    R205 IZMIR numPeople=7 buildingRisk=HIGH emergencyLevel=5

Emergency scores:
    S101: 4*10 + min(40, 50)  = 80
    R205: 5*10 + 7*3          = 50 + 21 = 71

Result:
    S101 is more emergent, so HANDLE_EMERGENCY will take from the
    supply queue first for this step.
```

2.3 Mission Overload and Rollback Rules

Each team has a `maxLoadCapacity` representing the maximum workload it can handle in a single mission. During `HANDLE_EMERGENCY`, each assigned request contributes to the team's workload as follows:

- For SUPPLY requests: `workload += amount`
- For RESCUE requests: `workload += numPeople * riskMultiplier`

where `riskMultiplier` is 1 for `"LOW"`, 2 for `"MEDIUM"`, and 3 for `"HIGH"`.

- If adding a new request causes the workload to exceed `maxLoadCapacity`, a mission overload occurs:
  - Print:

    ```
    Overload on Team <teamID>: rolling back mission.
    ```
  - All requests currently stored in the team's mission stack must be popped and returned back to their **original queues** (supply or rescue), preserving their relative order within each type.
  - The team's mission stack becomes empty and the mission is cancelled.
- If no overload occurs, the assigned requests remain in the mission stack until `DISPATCH_TEAM` is called.

Worked Example: Team Capacity and Overload

Assume:

- There is 1 team, `Team 0`, with `maxLoadCapacity = 60` (configured in the input file),
- The risk multipliers are: `"LOW" = 1`, `"MEDIUM" = 2`, `"HIGH" = 3`.

Input commands:

```
INIT_TEAMS 1
# Initialize the system with 1 team (Team 0)

SET_TEAM_CAPACITY 0 60
# Set maxLoadCapacity for Team 0 to 60

# --- First mission: will fit into Team 0's capacity (60) ---
ADD_SUPPLY S101 ANKARA WATER 20 3
# S101 (SUPPLY): emergencyScore = 3*10 + min(20,50) = 30 + 20 = 50
# workload contribution if assigned = amount = 20

ADD_RESCUE R201 IZMIR 2 MEDIUM 3
# R201 (RESCUE): emergencyScore = 3*10 + (2 people * MEDIUM(2))
#                                = 30 + 4 = 34
# workload contribution if assigned = numPeople * riskMultiplier = 2 * 2 = 4

HANDLE_EMERGENCY 0 2
# Step 1: frontSupply = S101 (score 50), frontRescue = R201 (score 34)
#         => pick S101 (SUPPLY)
#         workload(T0) = 0 + 20 = 20
#
# Step 2: supplyQueue is now empty, so only R201 is available
#         => pick R201 (RESCUE)
#         workload(T0) = 20 + (2*2) = 24 <= maxLoadCapacity(60)
#         => mission accepted (no overload)

DISPATCH_TEAM 0
# Team 0 completes mission with 2 assigned requests (S101, R201)

# --- Second mission: will cause overload and rollback for Team 0 ---

ADD_SUPPLY S102 ANKARA FOOD 40 4
# S102 (SUPPLY): emergencyScore = 4*10 + min(40,50) = 40 + 40 = 80
# workload contribution if assigned = amount = 40

ADD_RESCUE R202 IZMIR 5 HIGH 4
# R202 (RESCUE): emergencyScore = 4*10 + (5 people * HIGH(3))
#                                = 40 + 15 = 55
# workload contribution if assigned = 5 * 3 = 15

ADD_RESCUE R203 IZMIR 8 HIGH 4
# R203 (RESCUE): emergencyScore = 4*10 + (8 people * HIGH(3))
#                                = 40 + 24 = 64
# workload contribution if assigned = 8 * 3 = 24

HANDLE_EMERGENCY 0 3
# Step 1: frontSupply = S102 (score 80), frontRescue = R202 (score 55)
#         => pick S102
#         workload(T0) = 0 + 40 = 40
#
# Step 2: supplyQueue is now empty, so only rescueQueue is used.
#         frontRescue = R202 (score 55)
#         => pick R202
#         workload(T0) = 40 + (5*3) = 40 + 15 = 55  <= 60  (still safe)
```

Corresponding program output (without comments):

```
Initialized 1 teams.
Team 0 capacity set to 60.
Request S101 added to SUPPLY queue.
Request R201 added to RESCUE queue.
Team 0 assigned 2 requests (1 SUPPLY, 1 RESCUE), total workload 24.
Team 0 dispatched with workload 24.

Request S102 added to SUPPLY queue.
Request R202 added to RESCUE queue.
Request R203 added to RESCUE queue.
Overload on Team 0: rolling back mission.

SUPPLY QUEUE:
S102 ANKARA FOOD 40 4

RESCUE QUEUE:
R202 IZMIR 5 HIGH 4
R203 IZMIR 8 HIGH 4
```

In this example:

- The **first** `HANDLE_EMERGENCY 0 2` call assigns two requests to Team 0, with workload

  $$20 \text{ (from S101)} \ + \ 2 \times 2 \text{ (from R201)} = 24,$$

  which is within the capacity 60, so the mission is kept and later dispatched.

- The **second** `HANDLE_EMERGENCY 0 3` call tries to assign three requests, but the predicted workload

  $$40 \text{ (S102)} \ + \ 5 \times 3 \text{ (R202)} \ + \ 8 \times 3 \text{ (R203)} = 40 + 15 + 24 = 79$$

  exceeds the capacity 60, so overload is detected *before* R203 is finally assigned. The system prints the overload message and rolls back all assigned requests for this mission, returning them to the queues.

Configuring Team Limits

Each `Team` object has a `maxLoadCapacity` field that determines when overload occurs.

In this assignment, the number of teams and their capacities are configured **via the input file** using:

- `INIT_TEAMS <numTeams>` to create the teams,
- `SET_TEAM_CAPACITY <teamID> <maxLoadCapacity>` to set each team's limit.

You must ensure that:

- `INIT_TEAMS` is called before any other team-related command,

- Every team used in `HANDLE_EMERGENCY` or `DISPATCH_TEAM` has its capacity set with `SET_TEAM_CAPACITY`.

All overload checks during `HANDLE_EMERGENCY` will use these `maxLoadCapacity` values.

2.4 System-Wide Rules

1. All dynamic memory must be properly managed.
   - Queues and stacks must deallocate their dynamic arrays when cleared or destroyed.
   - No memory leaks or invalid memory accesses should occur.
2. `PRINT_QUEUES` and `PRINT_TEAM` must reflect the current system state accurately.
3. Any invalid command should be reported as:

   ```
   Error: Unknown command '<command>'.
   ```
4. If `REMOVE_REQUEST` cannot find a given ID in either queue, the following message must be printed:

   ```
   Error: Request <id> not found.
   ```
5. Execution stops when the end of the input file is reached.

Implementation Notes

- You are provided with header and source templates defining all classes and structures required for the simulation (i.e., Starter Code).
- Request properties such as `type`, `city`, `supplyType`, and `risk` are represented as `std::string` fields. You must use string comparison (e.g. `if (risk == "HIGH")`) instead of enum types.
- All queue and stack data structures (`RequestQueue`, `MissionStack`, and any internal arrays) **must be implemented manually**. Usage of STL containers such as `std::queue`, `std::stack`, `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 are 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 request identifiers, cities, types, team IDs, or integer values. All keywords are written in uppercase and parameters are separated by spaces.

`Example:`

```
INIT_TEAMS 2
SET_TEAM_CAPACITY 0 80
SET_TEAM_CAPACITY 1 120
ADD_SUPPLY S101 ANKARA WATER 40 4
ADD_RESCUE R205 IZMIR 7 HIGH 5
HANDLE_EMERGENCY 1 2
DISPATCH_TEAM 1
PRINT_QUEUES
```

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.

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 QuakeAssist
```

After compilation, you can run the program as follows:

```
$ ./QuakeAssist example_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 `RequestQueue` (supply and rescue) 20%
  - Correct implementation of the `MissionStack` and team workload handling 15%
  - Correct management of teams and global system state 10%
- **Correct implementation of commands and system actions: 35%**
  - Correct execution of `INIT_TEAMS` and `SET_TEAM_CAPACITY` 5%
  - Correct execution of `ADD_SUPPLY` and `ADD_RESCUE` 8%
  - Correct execution of `HANDLE_EMERGENCY` and emergency selection logic 15%
  - Correct execution of `DISPATCH_TEAM` 5%
  - Correct execution of `REMOVE_REQUEST`, `PRINT_QUEUES`, and `PRINT_TEAM` 7%
- **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, 05/12/2025 (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**
    - main.cpp <FILE>
    - Request.cpp <FILE>
    - Request.h <FILE>
    - RequestQueue.cpp <FILE>
    - RequestQueue.h <FILE>
    - MissionStack.cpp <FILE>
    - MissionStack.h <FILE>
    - Team.cpp <FILE>
    - Team.h <FILE>
    - QuakeAssistController.cpp <FILE>
    - QuakeAssistController.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.