DATA STRUCTURES PROJECT

TOWER DEFENSE SIMULATION GAME
SUBMITTED TO: ENG.\ NESMA ABD EL-HAKIM

TEAM MEMBERS : TEAM 5

1. Mayar Mohsin Mostafa	1133070
2. Alaa Hazem Radwan	11332024
3. Assem Amr Mohamed	1132461
4. Ahmed Hamada Mohamed Kamel El-Hinidy	1132118

DATA STRUCTURES USED

1. **Queue:**
   Used for *"Inactive Enemies"* as they are sorted by team in the input file. We enqueue all inactive enemies at the end, and dequeue the enemies when they become active. We only check on the head of the queue to check if the GameTime==Arrival Time of enemy.
   The complexity for finding each enemy that becomes active is O(1)

2. **Priority Queue using Heap Tree:**
   Used for the *"Active Enemies"* as they need to be sorted by their priority at all times. We used the Heap Tree to implement the priority queue as it is a balanced tree that sorts itself with complexity O(logn).
   We used array representation of the Heap Tree by creating an "Array of Pointers to Enemies" in order to be able to send them to the DrawEnemies function directly.
   The complexity of finding each highest priority enemy to hit is O(1). The complexity of adding each enemy to the ADT and sorting it is O(logn).
   Note: In order to determine the length of arrays we read the input file 2 times to count the enemy numbers for each region then read the enemy data in the second time.

   Alternative Solution: We could read the file only once because we enqueue inactive enemies and we don't need to initialize the Arrays of the Heap tree at this point of the program flow, but for the sake if modularity, we preferred to keep them as separate modules “Module for Reading Enemy Data, and another Module for determining Enemy Numbers”

Innovative Solution: We initially allocate 4 Arrays according the number of enemies for each region in the input file. When a tower Dies and we need to transfer enemies from a region to another one, we merge the 2 Arrays of both regions.
We suffer from N + M iterations to copy enemies to a new array, but we find this pretty good to save memory. The bad solution was to allocate the 4 arrays with the Sum of the Enemy Numbers for all regions from the beginning.

3. **Double Linked List:**
We Used a DLL for the dead enemies in order to handle the requirement of printing the enemies ordered firstly by Kill Time, then by the Fight Delay.
We wanted to separate the logic of printing the enemies sorted from the core logic of the game, as we consider it an extra requirement that might be changed at any time, therefore it was better to keep it as a separate module.
We add the enemies to the end of the DLL as they die “sorted by kill time”. For the sake of printing we designed a recursive function to recursively determine the enemy that should be printed and returns a pointer to it. We then print its required data then remove it from the DLL.
We Chose DLL instead of normal Single Linked List in order to be able to remove the enemy and connect its next and previous nodes by only having a pointer to the node itself “which contains pointers to its next and previous nodes”. We don’t need another pointer to its previous node.

Alternative Solution: We could have inserted the enemies that die in each Time Step into a temp array or list then sort it by Fight Delay then insert them sorted by Kill Time and Fight Delay into the original list of Dead Enemies. We preferred our solution to separate the Game Logic from the printing logic, as we previously mentioned that we consider the printing logic a separate requirement and we wanted to keep it as a separate module instead of embedding it into the game’s core logic. We also wanted to practice designing a recursive function and use it in a real project.

Innovative Solution: The design of the recursive function to determine the next enemy that should be printed.

ASSUMPTIONS

1. If a tower has a certain enemies. We consider N the number of hits he can do at a certain time step. The tower chooses the Enemy with highest priority and hits it, then this enemy's stats are updated. Then the tower repeats the process again. This happens N times.

2. The paver moves normally with his own speed. When his hit period passes he should pave, he distance equal to his fire power and moves it at the same time.

   Example: Move 1, Move 1, Move 1, Pave 3 and Move 3, Move 1, Move 1

   This only happens when the distance in front of him is unpaved, otherwise he moves with his own speed until he finds unpaved land.

3. The Enemies begin hitting the tower first in each time step.

4. If an enemy has the ability to move a distance greater than the present paved distance, he will only move the allowed distance.

   Example: If an enemy wants to move 5 meters according to his speed, but the paved distance in front of him is only 2 meters, he will only move the allowed 2 meters.

BONUS REQUIREMENTS

1. The Modularity:
   We divided our project into the following .cpp files "each with the corresponding .h header file except the main.cpp":
   a) Read: To handle getting the enemy numbers and reading the enemy data.
   b) Queue: Queue Data Structure Interface.
   c) PriorityQueueHeap: Priority Queue using Heap Tree Interface.
   d) DLL: Double Linked List Interface + The printing logic as we consider it associated to the DLL.
   e) Utility: To handle the main structures and the drawing logic.

f) GameCore: To handle the logic of the simulation; e.g. hitting the enemies, updating the regions, etc…
g) main: To handle the closest logic to the user; e.g. determining the game mode, the main game loop, etc…