Project Report
Data Structures course, Zewail City of Science and Technology
Abdelrahman Mahmoud – Mohamed Ahmed – Abdelrahman Bedier – Amr Mohammed

I. Fibonacci heaps:

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a lazy manner, postponing the work for later operations. For example, merging heaps is done simply by concatenating the two lists of trees, and operation decrease key sometimes cuts a node from its parent and forms a new tree.

A Fibonacci heap has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap. They are named Fibonacci heaps after the Fibonacci numbers, which are used in their running time analysis. The find-minimum operation takes constant O(1) amortized time. The insert and decrease key operations also work in constant amortized time. Deleting an element (most often used in the special case of deleting the minimum element) works in O(log n) amortized time, where n is the size of the heap. This means that starting from an empty data structure, any sequence of insert and decrease key operations and b delete operations would take O(a + b log n) worst case time, where n is the maximum heap size. In a binary or binomial heap such a sequence of operations would take O((a + b) log n) time. A Fibonacci heap is thus better than a binary or binomial heap when b is smaller than a by a non-constant factor. It is also possible to merge two Fibonacci heaps in constant amortized time, improving on the logarithmic merge time of a binomial heap, and improving on binary heaps which cannot handle merges efficiently.

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of children and whether the node is marked. Moreover, we maintain a pointer to the root containing the minimum key. Operation find minimum is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost are constant. Operation insert works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant. Operation extract minimum operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time O(d) to process all new roots and the potential increases by d−1. Therefore, the amortized running time of this phase is O(d) = O(log n).

II. Project description:

In this project, we are required to make a game of a castle that is guarded by a number of towers and soldiers within, and attacked by different types of enemies. The input file for this game program contains the following:

- **Time stamp:** When the enemy will appear (enemy arrival time).
- **Health:** The start health of the enemy.
- **Fire Power:** The shot hit power of the enemy.
- **Hit Period:** The delay between two successive shots from the enemy.
- **Type:** Three types of enemies (paver, fighter and shielded fighter)
- **Region:** The attack region of the enemy.

And each tower will have a starting health and can shoot at most N enemies at each time step. Each
tower guards one region and can only shoot enemies in its region.

An active enemy is defined as an enemy who arrives at time less than or equals to current time step and its health is greater than zero. At each time step, each tower should choose N active enemies to fight. Other enemies are either killed or inactive. The enemy distance is the horizontal distance between the enemy and the tower.

Damage to the tower by certain enemy is calculated by the following formula:

$$\text{Damage (Enemy} \rightarrow \text{Tower)} = \frac{1}{Enemy\_distance} * Enemy\_fire\_power$$

And the damage caused by a tower to the enemy is calculated by:

$$\text{Damage (Tower} \rightarrow \text{Enemy)} = \frac{1}{Enemy\_distance} * Tower\_fire\_power * \frac{1}{k}$$

Where k=2 for shielded enemies and k=1 for other enemies.

Thus, based on the enemy attack and defense abilities and its type and power, every enemy should be given his "priority" by the towers' soldiers to attack the most prior one first. The priority is calculated by:

$$\text{Priority (Enemy)} = \left(\frac{1}{Enemy\_distance} * Enemy\_fire\_power * \frac{1}{Enemy\_remaming\_time\_to\_shoot+1} * C1\right) + Enemy\_health * C2 + Enemy\_Type * C3$$

Where c1, c2 and c3 are constants given in the input file.

There is a special enemy type which is the "Paver enemy". All enemies can approach to the castle one meter at every time step only if the next meter is paved. At the start of the simulation, the last 30 meters in the road to the castle are not paved and only the paver enemies can enter the unpaved distance to pave it so that enemies of other types can enter this paved distance in the next time steps. A paver does not shoot the towers and its "Fire Power" represents the number of meters it can pave.

III. Project implementation:

First some assumptions are made:

• Every tower can only attack the enemies in its region.
• Every enemy can only attack the tower in its region.
• All enemies start at 60 meters distance from the castle.
• Every tower can attack at most N enemies at each time step. N is read from the input file.
• The minimum possible distance for any enemy to approach is 2 meters.
• The enemies can approach to the castle one paved meter at every time step.
• The game is "win" if all enemies are killed
• The game is "loss" if the all towers are destroyed.
• If a tower in a region is destroyed all enemies (current and incoming enemies) in that
region should be transported to the next region. The next region means the adjacent region moving in the clockwise direction. (A → B → C → D → A)

The project is implemented using fibonacci heap, that is a heap is constructed for each tower and every enemy is assigned to the tower's heap in which it lies in its scope. The tower's soldiers hence perform three attacks to the most prior enemies given in the heap.

The simulation is time-driven, meaning that the game "clock" is incremented and every time step the data is processed and the resulting scenario is shown. Also the project was done to be as modular as possible.

IV. **Output**:

An output file is generated at the end of the game that states the ID of every enemy, whether it died or not, the time of death if died, the fight delay and the kill delay. Also it shows whether you win or lose and some statistics such as the total number of enemies and the average fight and kill delays. Following is a sample output file:

5	1	0	5	5
10	2	4	4	8
15	3	5	2	7

33.5	12.5	55	200
30	30	25	2

WIN
50
4.5
12.36