Csci 335 Assignment 2

Mutable Priority Queue

First, modify the quadratic probing HashTable class to utilize (Key, Value) pairs similar to the STL map class. You should implement methods to set, modify and retrieve the Value for an associated Key. You may want to overload `operator[]` for convenience but it is not necessary.

Modify the BinaryHeap implementation given by the book to support the `decreaseKey(x, delta)`, `increaseKey(x, delta)` and `remove(x)` operations, where x is an element of the binary heap.

The BinaryHeap class will need to use your modified HashTable class in order to make these operations logarithmic worst-case time. The BinaryHeap will have a hash table as a private member that must keep track of the index of each element in the binary heap array. This additional hash table needs to be updated during every `buildHeap`, `insert` and `deleteMin` operation on the priority queue. (Note: The hash table will have to be updated at every step of each Percolate Up and Percolate Down operation.) The hash table should hash on the hash Key of objects in the queue and should store as the Value the position index in the binary heap's array where the item is stored.

`decreaseKey` should lower the priority value of `x` by a positive amount `delta` and restore the heap order property with a percolate up operation. `increaseKey` should raise the priority value of `x` by `delta` and restore the heap order property with a percolate down operation. `remove` can be implemented by performing `decreaseKey(key, inf)` and then performing `deleteMin`. Use the limits library to find a good value for inf. Instead of using an inf constant you can also peek at the current min in the heap to make sure you decrease the key to something less than the current min.

Event-Based Simulation

Using the mutable priority queue you will perform an event-based simulation like that described in section 6.4.2. The priority queue will be a queue of events corresponding to the time that each event is scheduled to take place. The priority value for each event is the time the event will take place with respect to the beginning of time at time t = 0. Rather than having a real-time simulation that simulates every time step for every entity from 0 until the end of the simulation, we will instead advance the clock to the next event corresponding to the minimum element in a priority queue. That is, the next event to be simulated is the one closest to the beginning of time. Here we will refer to our units of time as ticks.

You will simulate the (fictional dramatized) Battle of Thermopylae between 300 Spartans and 39,000 Persians. Assign a unique id number to each soldier in the battle, which will be the hash key for events corresponding to that soldier. For each solider there will be an event object in the queue that determines at what time their next action will take place, what is their id, and what side they are on.

At the start of the battle, for each Spartan generate their first action event at some point between time 1 and 50. For each Persian do the same between time 51 and 1000.

For each Spartan event, remove a Persian soldier from the battle and the corresponding queued event associated with that solider. Queue up a new event for the Spartan soldier +1 to 6 ticks from when the current event took place.

For each Persian event, they have a 5% chance of wounding one random Spartan solider and delaying the Spartan soldier's next action by +1-4 ticks (increaseKey). Queue up the Persian soldier's next event at +10-60 ticks after this one. (Note: Efficiently selecting a random Spartan is actually an interesting challenge. Simply using a vector of living Spartans will cause each remove to be O(n) in the worst-case, making this the theoretically slowest part of the simulation. There are several better ways to do this.)

Any Spartan soldier that sustains three wounds dies, but inspires all other remaining Spartans to take their next action +1-2 ticks sooner (decreaseKey). Note: This may cause the priority value for these events to "travel backwards in time", this is okay as long as you keep track of the time of the last event simulated and do not turn back the clock when you dequeue an event with an earlier time. Simply think of this as the Spartans acting faster than expected and not actually time travel.

The battle ends when all the soldiers on one side or the other have been eliminated.

Name your program `sim300`. Your program should run from the command line as follows:

`sim300 i nSpartans nPersians`

Where `i` is the number of times the simulation is run, `nSpartans` is the number of Spartans and `nPersians` is the number of Persians.

For each run print the time for the simulation to complete, who wins and the number of survivors on the winning side.

After all the runs print how many times each side won and the average and standard deviations of: the time to complete the battle, number of surviving Spartan soldiers when Spartans win, and number of surviving Persian soldiers when Persians win.

Include in your readme file the final statistics for `i` = 100, `nSpartans` = 300, `nPersians` = 39,000.