**Possible solutions to the database of customers:**

* An ArrayList driven by a class so that we could easily search for each patient, and we wouldn’t be limited in terms of space.
* A binary search tree where the first right and left roots divide the two different types of patients, then it will store them normally by using the patient id.
* A hash table holding linked lists so that the adding and searching of patients would be efficient, and the linked list to give us more space to work with.
* A .txt file that will store all the existing patients (with all the relevant information). When a patient is going to be registered it will search if he already exists in the document.

**Solution to the differing priorities within the patients**

* After some consideration we have concluded that the best way to implement the waiting mechanic that the patients are to be submitted to is a single priority queue that modifies the priority of each patient according to their waiting time. The reason of this is that single queue would mean that priority patients are to be attended as any other patient, leading to an increased risk upon their lives, and while two queues would be a certainly fitting solution, it still has the problem of patients being attended only by their access time, and further risk patients would still be left behind less severe patients. As such, a single priority queue would solve this problem to a reasonable degree; because the priority queue works in around a priority system, higher priority patients would be naturally higher in the list than non-priority patients, also, to avoid normal patients staying indefinitely in queue, a point of priority may be added each time a patient is called.

**Problems with the implementation of hash table**

* After some initial testing, our implementation of hash table seamed to be in working order, until we came to the realization that we skipped over something: our key had to be a string while in theory it had to be able to be anything. This all led to some quick changes being done to our implementation, where the numeric value of the kay was no longer calculated by our method, but rather the included method to do such thing in java: .hashCode(). This meant that we no longer had complete knowledge of what exactly was being calculated, nor what conditions had to be met for it to be deterministic. After making the changes, initial testing proved optimistic, as everything worked as intended, but when the time came to test the deletion method, it all came crumbling down, as not only did it not work, but it was also unclear as to why. The behavior presented by the deletion methos was unusual to say the least, giving different results depending on how it was run; as running it as usual would net a failure, while running it by the debugger net a success, this behavior is still not fully understood by the team. After some changes and extra debugging the possible cause of the error was reached: .hashCode() gave a different integer depending on which method was running it; an unexpected computational error had led to the deletion of the wrong element, sometimes none. The solution to this error was apparently to make a common method that converted a key to an int value, and this for whatever reason worked flawlessly. In conclusion it is still unknown what caused the behavior presented by the program before, but at the very least the mistake seems to be solved.