MediMaint Technical Solution Pseudocode

Repair class holds repair details relevant for queue ordering

```
class Repair{
     int repairID
     int techID
     float timeEstimate
     float repairValue
     datetime repairSubmitted

     boolean bonus
     float repairBonus
     int bonusMaxDays

     boolean penalty
     float repairPenalty
     int penaltyMaxDays
}
```

Priority Queue class orders repairs to maximize profit

```
class RepairQueue{

     DoublyLinkedList <Repair> repairs
     Repair * lastRepair
     int workHoursPerDay = 8

     // Only attempt to find the best position for repairs that
     have bonuses/penalties.
     procedure insert (Repair r)
          if isEmpty()
               insertLast(r)

          if !r.repairBonus && !r.repairPenalty
               insertLast(r)

          else
               insertInOrder(r)

     procedure insertLast (Repair r)
          repairs.add(r)
          lastRepair = r;
```

```
// Add new repair to the front of the queue.  Walk it
backward, evaluating the queue profit each step to find the
max profit. Finalize that location as far back in the queue
as possible.
function insertInOrder (Repair r)
     int [] profits = int [repairs.length]

     for (i = 0; i < repairs.length; i++)
          repairs.add(r, i)
          profits [i] = queueValue(repairs)
          repairs.remove(r)

     float maxProfit = 0
     float maxProfitLocation = repairs.length - 1
     for (i = profits.length - 1, i >= 0; i--)
          if profits [i] > maxProfit
               maxProfit = profits[i]
               maxProfitLocation = i

     repairs.add(r, maxProfitLocation)

     if maxProfitLocation = repairs.length – 1
          lastRepair = r

function queueValue (DoublyLinkedList <Repair> repairs):
float
     float totalTime
     float grossProfit
     Iterator<Repair> r = repairs.Iterator()

     while r.hasNext()
          totalTime += r.timeEstimate
          grossProfit += repairValue(r, totalTime)
     return grossProfit

function queueTime (DoublyLinkedList <Repair> repairs):
float
     float totalTime
     Iterator<Repair> r = repairs.Iterator()

     while r.hasNext()
          totalTime += r.timeEstimate
     return totalTime
```

```
     function repairValue (Repair r, float timeElapsed): float
          float daysElapsed = timeElapsed / workHoursPerDay

          if r.repairBonus && daysElapsed < r.bonuxMaxDays
               return r.repairValue + r.repairBonus

          if r.repairPenalty && daysElapsed >= r.penaltyMaxDays
               return r.repairValue - r.repairPenalty

          return r.repairValue


     function pop (): Repair
          Repair topPriority = repairs
          repairs = repairs.next
          return topPriority

     function isEmpty (): boolean
          return repairs == NULL
}
```

Technician class to contain a queue for one individual

```
class Technician {
     string name
     RepairQueue techQueue

     function getQueueLength (): float
          return techQueue.queueTime()

     function getQueueGrossProfit (): float
          return techQueue.queueValue()


     procedure addRepair (Repair r)
          techQueue.insert(r)
}
```

"Main" class to run the repair scheduling and queue assignment

```
class RepairScheduler {

     RepairQueue allRepairs
     HashMap <int, Technician> techs

     // Add new repairs to the master list (for overall
     priority) and also to the assigned tech's queue
     procedure addRepair (Repair r)
          allRepairs.insert(r)
          tech.get(r.techID).insert(r)

     function getAllRepairGrossProfit (): float
          float grossProfit = 0

          foreach t in techs.values()
               grossProfit = t.getQueueGrossProfit()

          return grossProfit

     function getHoursToRepairQueueCompletion (): float
          float maxHours = 0
          float hours
          foreach t in techs.values()
               hours = t.getQueueLength()

               if hours > maxHours
                    maxHours = hours

          return maxHours
}
```

Notes

- Retrieving the highest priority element is **O(1)**
- Inserting an element is **O(n)** if the repair queue is empty or the repair doesn't have bonus/penalty stipulations. Otherwise, insertion depends on the linked list implementation:
    - A typical linked list with nodes and pointers would insert a new repair in $\mathbf{O(n^2)}$ time because the algorithm iterates over the entire queue (n) and inserts/removes the repair in each possible location (2n with a pointer-based linked list).
    - An ArrayList implementation with array operations can bring it down to amortized **O(n)** time.
- Once the queue is established, the order could be saved to the database (repair table's "priority" column) and reconstructed later to avoid executing the Insert algorithm more than once per repair.
- Priority queue does not account for a tech's specialty – it relies on the foremen to assign the correct time estimate to the repair according to the tech assigned (all techs can accomplish all tasks but not at the same pace per the Q&A).
- This implementation does not actively attempt to keep the queue under 45 days of work. It naively achieves this by adding repairs without bonuses/penalties to the end of the queue (FIFO) but that 45-day turnaround goal could be violated if company management accepted too many repairs, too many repairs with bonus/penalty times, or assigns more than a 45-day queue to one technician
    - Use *RepairScheduler.getHoursToRepairCompletion* to find the longest duration queue. If that is over 45 days, assign repairs from that queue to under-tasked technicians.
- The *Technician* class decouples the priority queue implementation from the main *RepairScheduler* class. If a new priority queue implementation is desired, it only takes a few lines of code for the *Technician* class to change over.
- The website dashboard for this system would display the entire prioritized queue, queues by technician, etc. so management could visualize how their repair assignments are balancing out.