COMP1002 Final Assignment Report

BY THEJANA KOTTAWATTA HEWAGE

22307822

1) Overview: System Purpose and Modular Design

The purpose of the project is to optimise hospital workflows. Each module of the project is constructed in a way which offer best performance for each module's task.

1. For module 1, a graph-based adjacency list is chosen as the ideal data structure for modelling networks. It represents hospital's each department as vertices and corridors as edges.
2. For module 2, a has table is implemented to provide an average case time complexity for inserting and searching of patient records.
3. For module 3, a max heap is used to assign priority to patients. This module of the program ensures that highest priority patient is identified and treated more efficiently.
4. For module 4, merge sort and quick sort are used to sort large datasets.

2) Data Structures Used: Justification for graph, hash table, heap, sorting.

Module 2: Hash-Based Patient Lookup – Short written justification of hash strategy, table size and hash function

1. Collision Handling Strategy: Chaining

The collision handling strategy I've chosen is chaining using linked lists, over open addressing methods like linear probing.

- Chaining is generally simpler to implement, particularly when it comes to the deletion of entries. In open addressing DELETE often requires complex logic to maintain the integrity of subsequent lookups. Whereas in chaining, you simply remove a node from the linked list.
- Chaining handles high load factors (many entries compared to the table size) more gracefully. In contrast, open addressing methods suffer from clustering, where occupied slots form large contiguous blocks, causing performance to degrade across the entire table.

2. Hash Function & Table Size strategy: Modulo + Prime Number

- The primary goal of the hash function I have chosen is to minimise collisions and apply uniform key distribution.
    - Hash function: The sum of ASCII values of the patient ID's is used for has I believe this was sufficient for the application's needs.
    - Table Size optimisation: The table size is always set to the next prime number greater than the requested capacity.
    - Prime modulo operator: Remainder (%) of when hash is divided by size of the table is used to
- Why Prime? Using a prime number as the table size is crucial when the modulo operator (the remainder of the hash divided by the table size) is used to map the hash value to a bucket index. A prime size helps ensure that the resulting indices are more uniformly distributed, which significantly reduces the number of initial collisions.

Module 3: Heap-Based Emergency Scheduling – Brief written note on update strategy

Update Strategy: Reinsertion is the strategy I have chosen for handling a change in a patient's urgency level. Each update is treated as a new request. A potential new entry with the new, re-calculated priority is inserted into the heap. My approach avoids any alternative approaches such as: a manual search & update O(n) operation. When the old, lower-priority request is eventually found, the system can perform a quick check against the hash table to see if the patient's status has changed or if they have already been treated and discard the old request.

Implementation: 0-Based Array Per the provided example, this heap uses a 0-based array.

Parent of i: (i - 1) // 2

Left Child of i: 2 * i + 1

Right Child of i: 2 * i + 2

Module 4: Sorting Patient Records - Brief write-up on stability and pivot strategy.

Merge Sort: Top-down recursive approach is an intuitive way. Moreover, it uses a temporary array to merge results back into the original array "A".
Quicksort: Quicksort uses *median of* 3 pivot strategy.

- Pivot strategy (implementation):
  - GetMedianPivot function inspects the 1st, middle and final elements.
  - Manually sort the 3 elements in and return the median value as an index.
- My chosen pivot strategy is a good against the worst-case scenario: causing an already sorted list to be resorted.

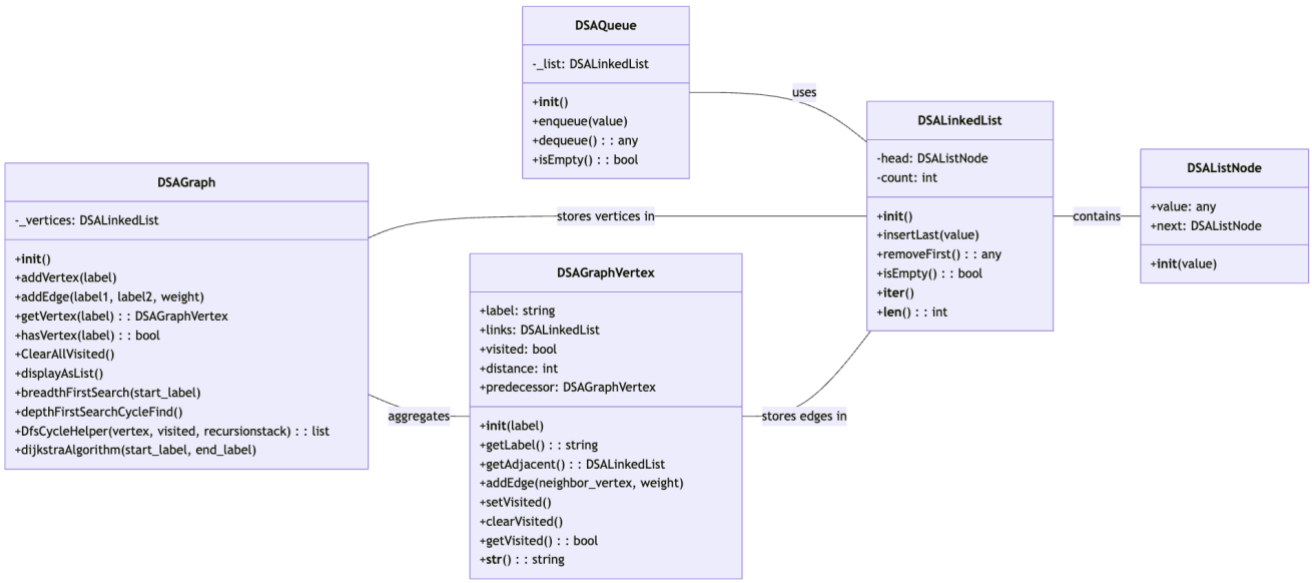
Stability:

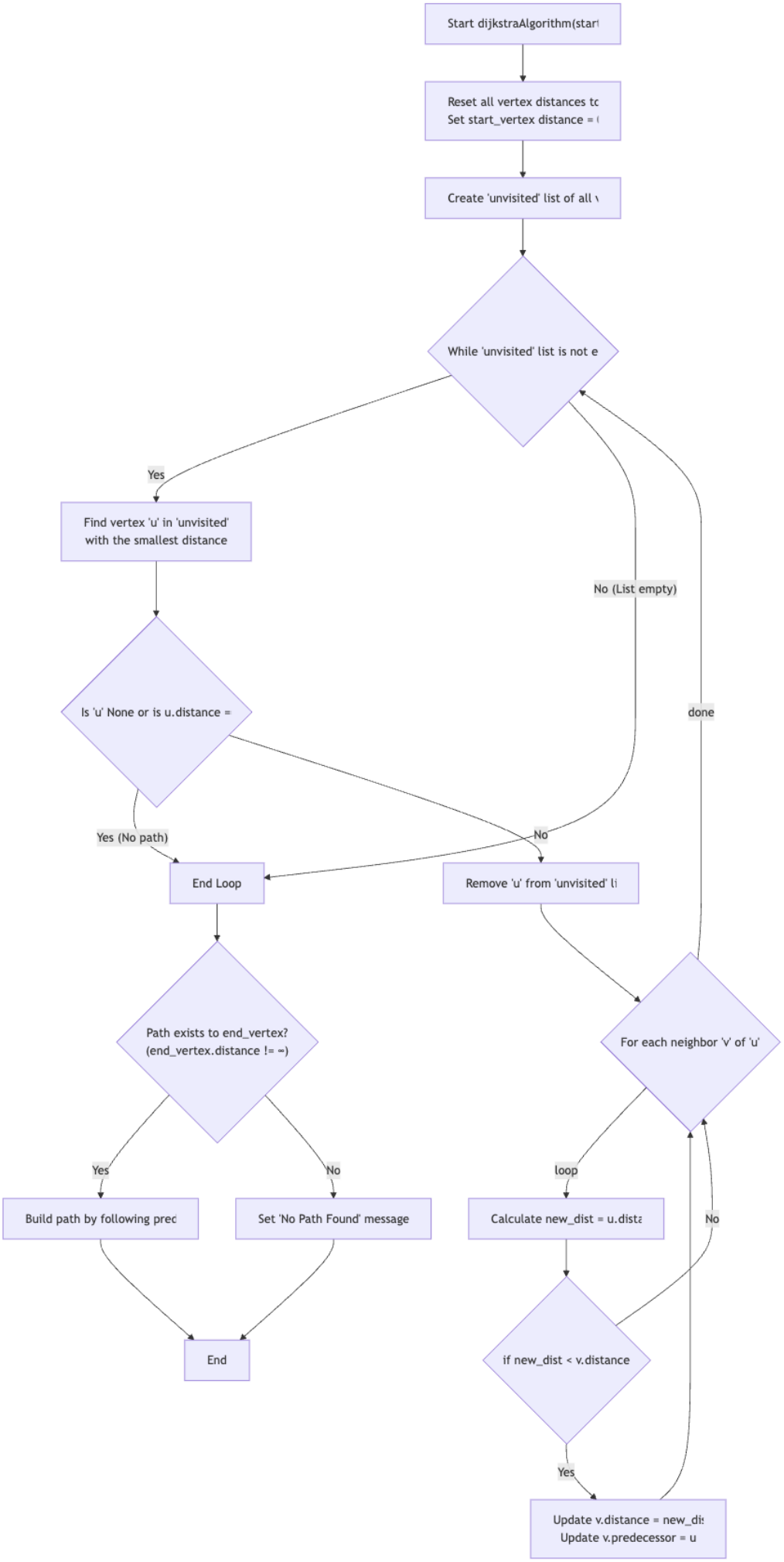
- MergeSort is very stable. Dring merge, if 2 elements have the same duration, element from left array is taken first, preserving the original order of equal elements.
- Quicksort is not stable. Partition involves long distance swaps which don't preserve order of elements.

3) Module Integration: Flowchart/UML describing data flow and interactions.

Flowchart/UML describing data flow and interactions

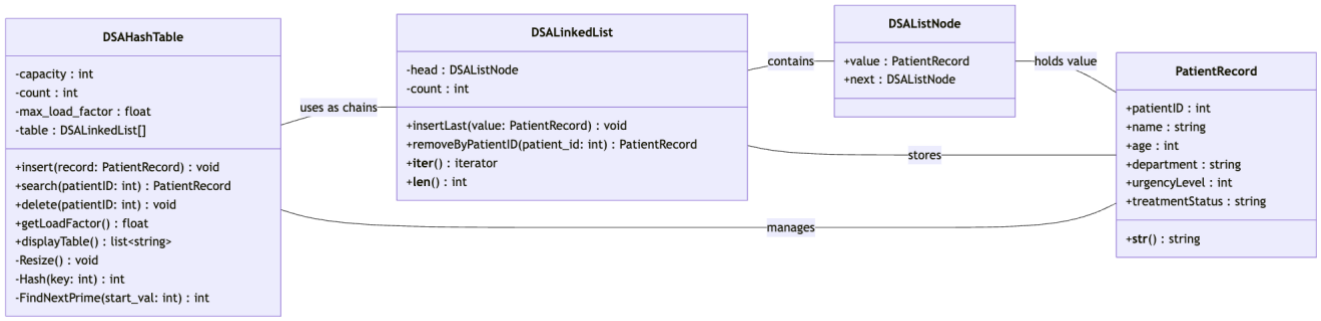
Module 1: Graph Based Hospital Navigation

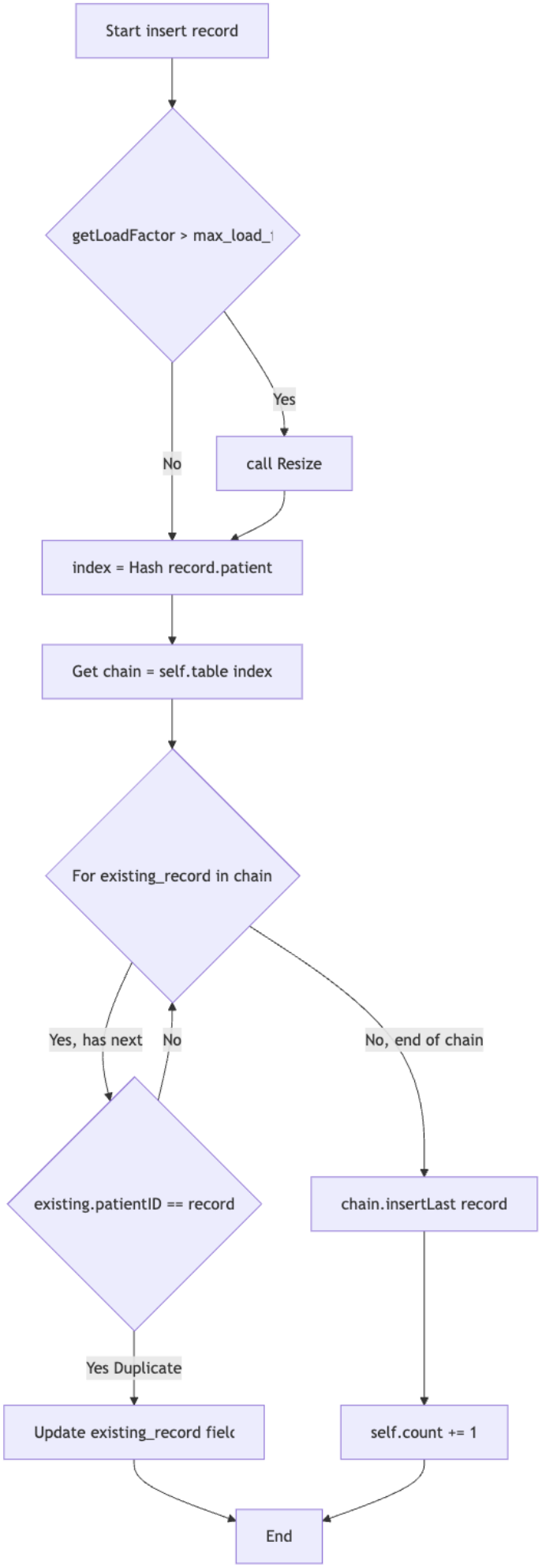




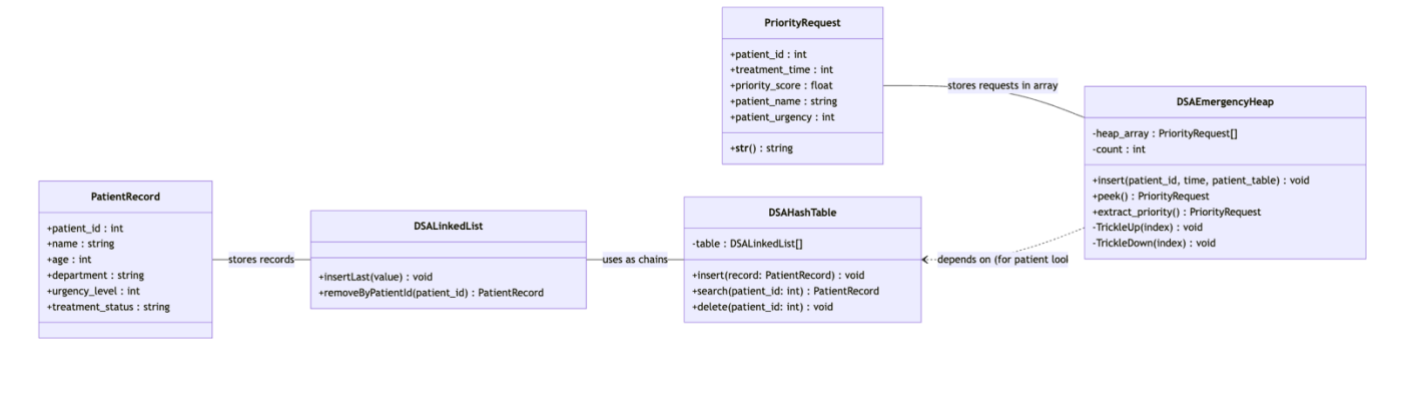
onnected to Discord

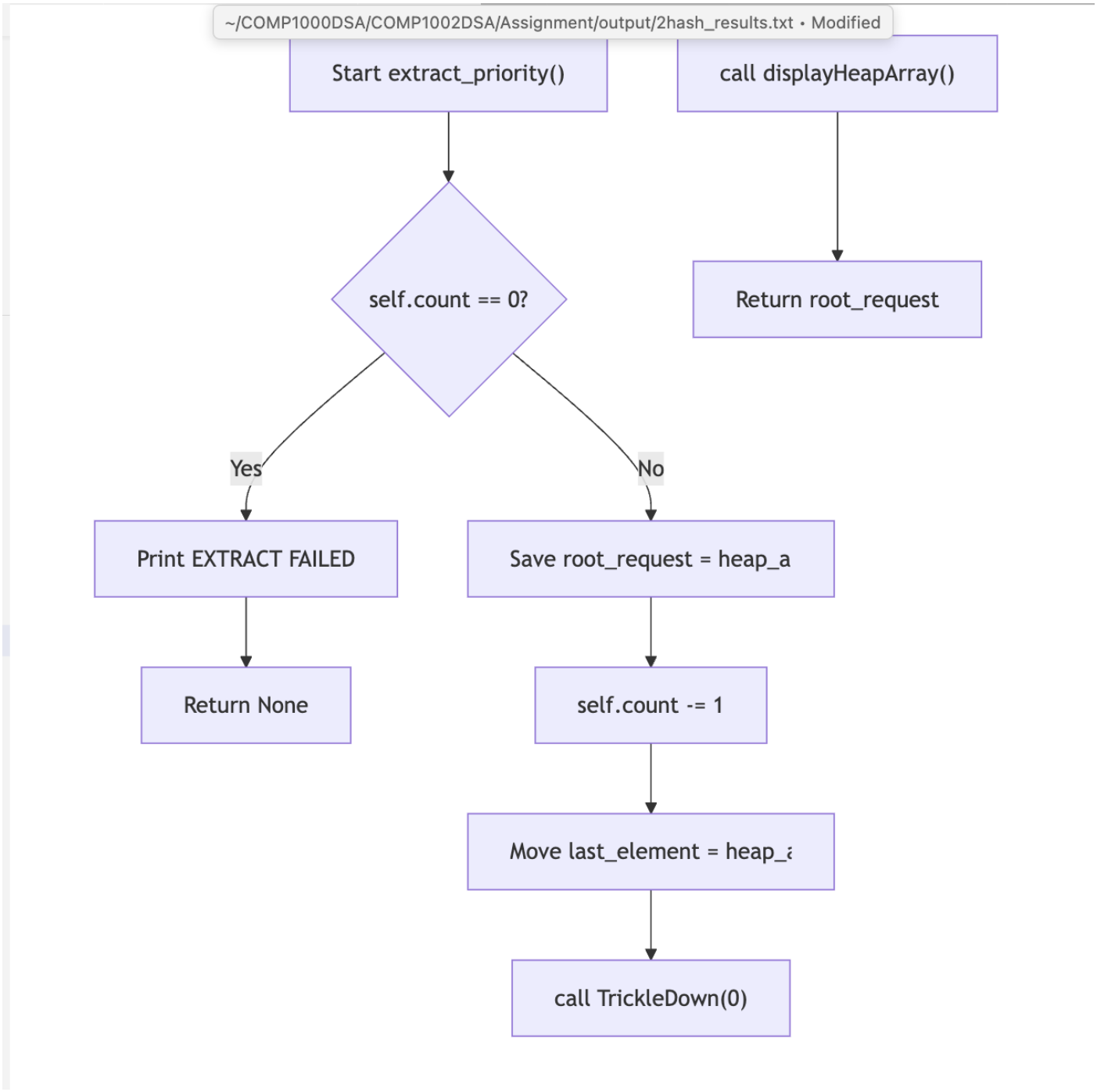
Module 2: Hash-based patient lookup



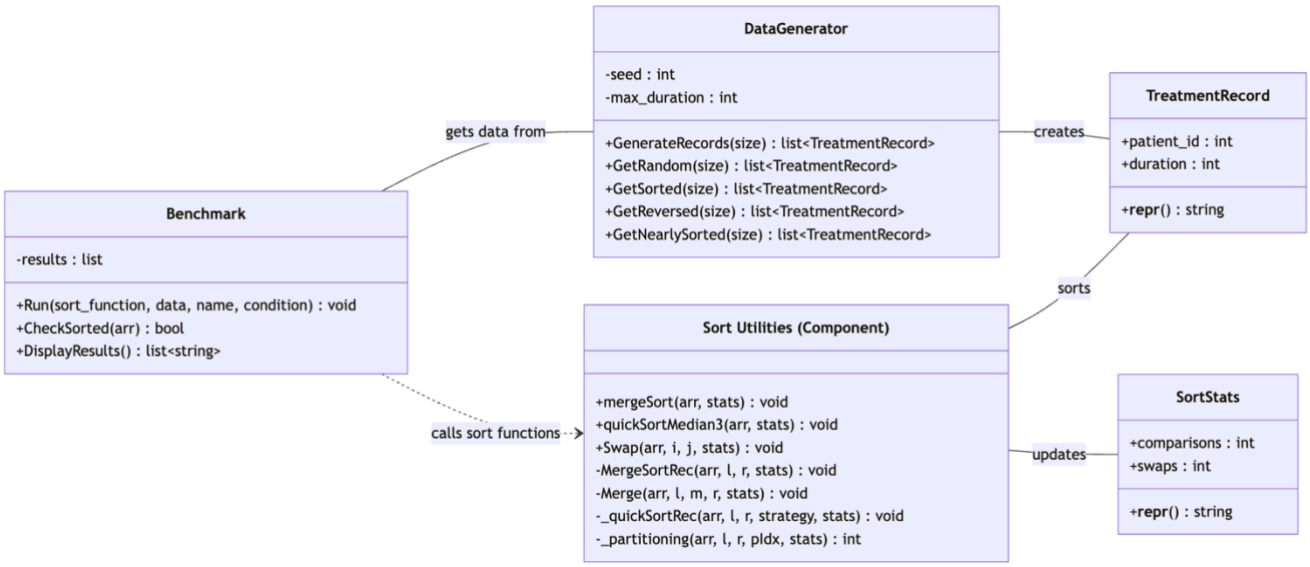

Start insert record
getLoadFactor > max_load_
Yes
No
call Resize
index = Hash record.patient
Get chain = self.table index
For existing_record in chain
Yes, has next
No
No, end of chain
existing.patientID == record
chain.insertLast record
Yes Duplicate
Update existing_record fiel
self.count += 1
End

Module 3: Heap-based emergency scheduling.





Module 4: Sorting Patients Records



4) Algorithm Complexity: Time and space analysis for each module.

Module 1: Graph Based Hospital Navigation

- Performance of module 1 is defined mainly by its 2 main variables V (no. Of departments/vertices) and E (edges/corridors.
- The BFS DFS algorithms have linear time complexity.
- Dijkstra's algorithm has a time complexity of O(n^2). This is a trade off because, as we are unable to use built-in functions, a manual scan of the list of departments is require.

Module 2: Hash-based patient lookup

- Module achieves an average-case time complexity O(1) for insert(0, search() and delete().
- This is achieved with the combination of prime number and good hash function for the distribution of keys.
- The good performance is depicted by the load factor being a healthy figure of 0.45, even after undergoing resizing.
- Primary trade-off: choosing chaining for collision handling.

Module 3: Heap-based emergency scheduling.

- Core operations insert() and extract_priority() achieve a time complexity of O(nlogn).
    - They both use trickle up/down and both need to traverse the height of the heap tree rather than the entire array.
- Peek() finds the highest priority patient with an instantaneous O(1).
- Trade off: a small cost of O(logn) on every insertion.

Module 4: Sorting Patients Records

- As n increases from 100 -> 1000, runtime increases from 0.08ms to 1.3ms. (nlogn growth).

- Merge sort is independent of order of input. Time on reversed data is the same as random input (1.106ms == 1.319ms).
- Conversely, quick sort operates using O(logn). With O(n^2) on worst caps input.

5) Sample Output: Screenshots/log excerpts per module.

Module 1: Graph Based Hospital Navigation

~/COMP1000DSA/COMP1002DSA/Assignment/module1.py • Deleted

```
1  ##############################
2  ###   Hospital Navigation   ###
3  ##############################
4
5
6  --- Hospital Adjacency List ---
7  Emergency          | ICU(4) -> Radiology(2) -> Wards(7)
8  ICU                | Emergency(4) -> Operating Theatres(2) -> Pharmacy(3)
9  Pharmacy           | ICU(3) -> Laboratories(2)
10 Radiology          | Emergency(2) -> Laboratories(3)
11 Laboratories       | Radiology(3) -> Pharmacy(2)
12 Operating Theatres | ICU(2) -> Wards(2)
13 Wards              | Emergency(7) -> Operating Theatres(2) -> Outpatient Units(5) -> Cafeteria(3)
14 Outpatient Units   | Wards(5) -> Cafeteria(2)
15 Cafeteria          | Wards(3) -> Outpatient Units(2)
16 Morgue             |
17
18 --- BFS starting from 'Emergency' ---
19 Level 0: Emergency, ICU, Radiology, Wards
20 Level 1: Operating Theatres, Pharmacy, Laboratories, Outpatient Units, Cafeteria
21 Level 2:
22 Cycle Detected: Emergency -> ICU -> Emergency
23
24
25 --- Shortest Path from 'Emergency' to 'Outpatient Units' ---
26 Path: Emergency -> Wards -> Outpatient Units
27 Total walking time: 12 minutes.
28
29
30
31 --- Shortest Path from 'Emergency' to 'Morgue' ---
32 No path found from 'Emergency' to 'Morgue'.
33
```

Module 2: Hash-based patient lookup

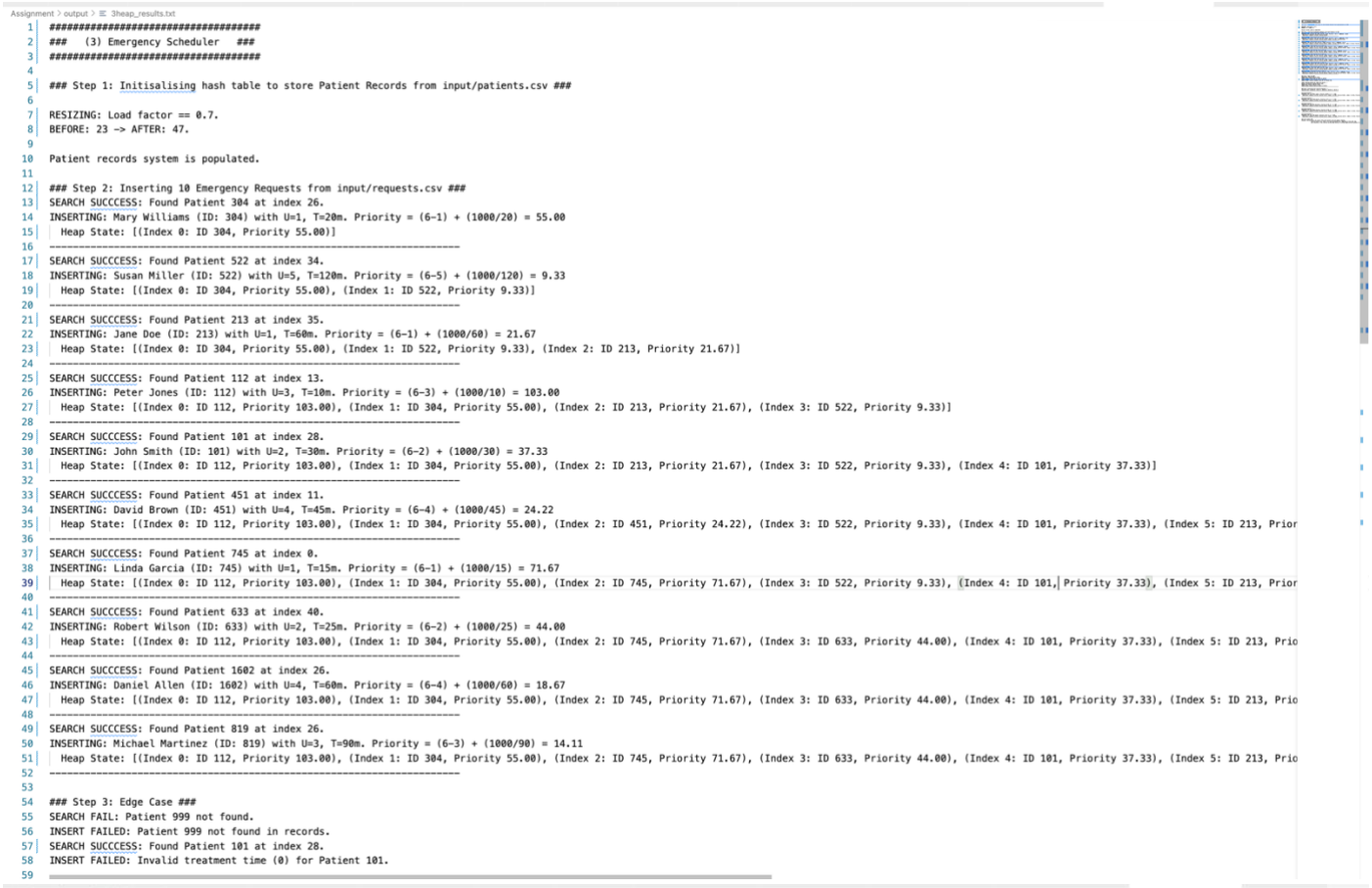
Assignment > output > 2hash_results.txt

```
##########################################
###   MODULE 2: Hash-Based Patient Lookup   ###
##########################################

### Step 1: Demonstrating Inserts ###
Demonstration: Hash(112) = 5, Hash(213) = 10
Patient records system is populated.


COLLISION DEMO: Patient 112 and 213 both hash to the same initial index.
    Hash Table will resolve collsions via the colllsion strategy I've implemented; Chaining.

======================== HASH TABLE ========================
Count: 21, Capacity: 47, Load Factor: 0.45
Index 00: [ID: 745] -> None
Index 01: [ID: 1734] -> None
Index 06: [ID: 2088] -> None
Index 08: [ID: 901] -> None
Index 09: [ID: 1599] -> None
Index 11: [ID: 451] -> None
Index 12: [ID: 1366] -> None
Index 13: [ID: 112] -> None
Index 22: [ID: 1245] -> None
Index 23: [ID: 1010] -> None
Index 26: [ID: 819] -> [ID: 304] -> [ID: 1602] -> None
Index 28: [ID: 101] -> None
Index 29: [ID: 1977] -> None
Index 31: [ID: 1123] -> None
Index 34: [ID: 522] -> [ID: 1851] -> None
Index 35: [ID: 213] -> None
Index 40: [ID: 633] -> None
Index 44: [ID: 1482] -> None
============================================================


### Step 2: Searching for Patients ###
Searching for an existing patient: (patient 101)
   Record found: ID: 101, Name: John Smith, Age: 45, Dept: Cardiology, Urgency: 2, Status: Admitted

Searching for a non-existent patient: (patient 999)
    Record not found (SUCCESSFUL).

### Step 3: DELETE Demonstration ###
Deleting Patient 451...
Attempting to search for the deleted patient:
    Record not found (SUCCESSFULLY DELETED.)

Attempting to delete a non-existent patient:
    Delete attempt finished (GRACEFUL ERROR).

======================== HASH TABLE ========================
Count: 20, Capacity: 47, Load Factor: 0.43
Index 00: [ID: 745] -> None
Index 01: [ID: 1734] -> None
Index 06: [ID: 2088] -> None
Index 08: [ID: 901] -> None
Index 09: [ID: 1599] -> None
Index 12: [ID: 1366] -> None
Index 13: [ID: 112] -> None
```

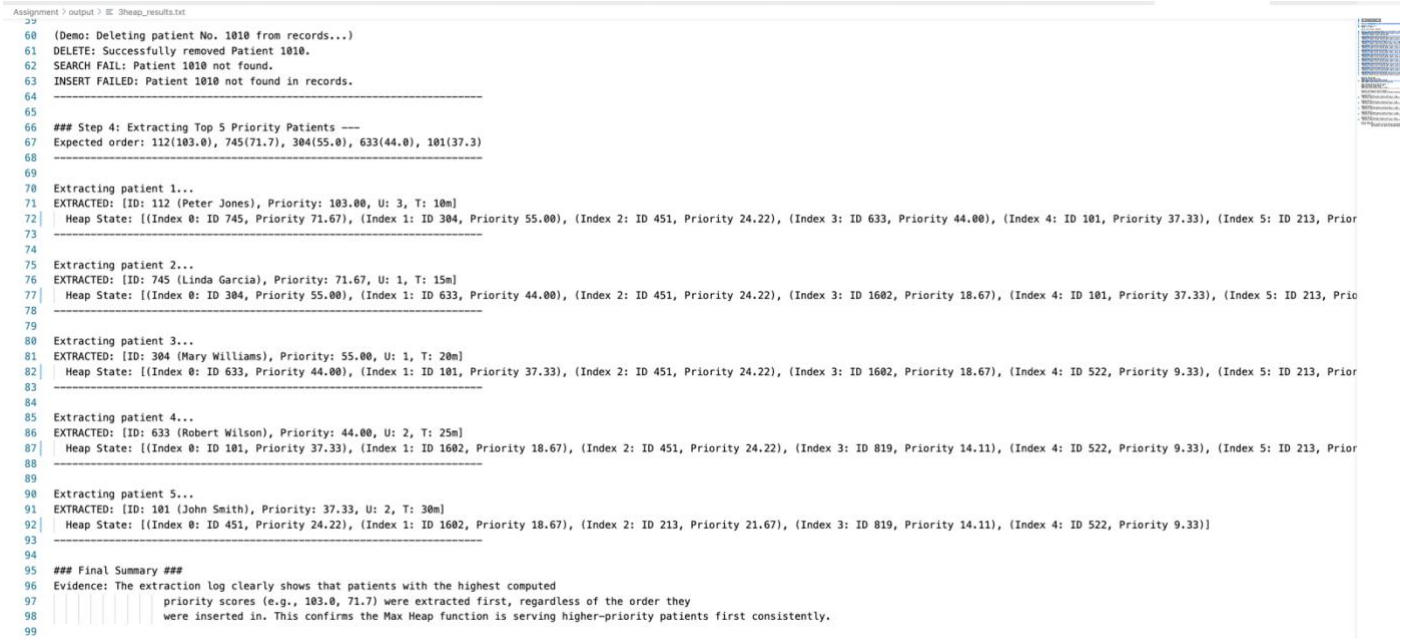
Assignment > output > 2hash_results.txt

```
Index 13: [ID: 112] -> None
Index 22: [ID: 1245] -> None
Index 23: [ID: 1010] -> None
Index 26: [ID: 819] -> [ID: 304] -> [ID: 1602] -> None
Index 28: [ID: 101] -> None
Index 29: [ID: 1977] -> None
Index 31: [ID: 1123] -> None
Index 34: [ID: 522] -> [ID: 1851] -> None
Index 35: [ID: 213] -> None
Index 40: [ID: 633] -> None
Index 44: [ID: 1482] -> None
======================================================================


--- Final Summary ---
Complexity: All operations (insert, search, delete) demonstrated low 'hop' counts,
            supporting the expected O(1) average time complexity. The worst-case for a single
            operation would be O(n) if all keys hashed to the same index.

Load Factor Behaviour: The table began with a capacity of 11 and was resized
            to 23 when the load factor exceeded the threshold of 0.7,
            ensuring that chains. remain short and performance remains high.
```

Module 3: Heap-based emergency scheduling.



```
Assignment > output > ≡ 3heap_results.txt
1  ####################################
2  ###   (3) Emergency Scheduler   ###
3  ####################################
4
5  ### Step 1: Initisalising hash table to store Patient Records from input/patients.csv ###
6
7  RESIZING: Load factor == 0.7.
8  BEFORE: 23 -> AFTER: 47.
9
10 Patient records system is populated.
11
12 ### Step 2: Inserting 10 Emergency Requests from input/requests.csv ###
13 SEARCH SUCCCESS: Found Patient 304 at index 26.
14 INSERTING: Mary Williams (ID: 304) with U=1, T=20m. Priority = (6-1) + (1000/20) = 55.00
15   Heap State: [(Index 0: ID 304, Priority 55.00)]
16 ----------------------------------------------------------------------
17 SEARCH SUCCCESS: Found Patient 522 at index 34.
18 INSERTING: Susan Miller (ID: 522) with U=5, T=120m. Priority = (6-5) + (1000/120) = 9.33
19   Heap State: [(Index 0: ID 304, Priority 55.00), (Index 1: ID 522, Priority 9.33)]
20 ----------------------------------------------------------------------
21 SEARCH SUCCCESS: Found Patient 213 at index 35.
22 INSERTING: Jane Doe (ID: 213) with U=1, T=60m. Priority = (6-1) + (1000/60) = 21.67
23   Heap State: [(Index 0: ID 304, Priority 55.00), (Index 1: ID 522, Priority 9.33), (Index 2: ID 213, Priority 21.67)]
24 ----------------------------------------------------------------------
25 SEARCH SUCCCESS: Found Patient 112 at index 13.
26 INSERTING: Peter Jones (ID: 112) with U=3, T=10m. Priority = (6-3) + (1000/10) = 103.00
27   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 213, Priority 21.67), (Index 3: ID 522, Priority 9.33)]
28 ----------------------------------------------------------------------
29 SEARCH SUCCCESS: Found Patient 101 at index 28.
30 INSERTING: John Smith (ID: 101) with U=2, T=30m. Priority = (6-2) + (1000/30) = 37.33
31   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 213, Priority 21.67), (Index 3: ID 522, Priority 9.33), (Index 4: ID 101, Priority 37.33)]
32 ----------------------------------------------------------------------
33 SEARCH SUCCCESS: Found Patient 451 at index 11.
34 INSERTING: David Brown (ID: 451) with U=4, T=45m. Priority = (6-4) + (1000/45) = 24.22
35   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 451, Priority 24.22), (Index 3: ID 522, Priority 9.33), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prior
36 ----------------------------------------------------------------------
37 SEARCH SUCCCESS: Found Patient 745 at index 0.
38 INSERTING: Linda Garcia (ID: 745) with U=1, T=15m. Priority = (6-1) + (1000/15) = 71.67
39   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 745, Priority 71.67), (Index 3: ID 522, Priority 9.33), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prior
40 ----------------------------------------------------------------------
41 SEARCH SUCCCESS: Found Patient 633 at index 40.
42 INSERTING: Robert Wilson (ID: 633) with U=2, T=25m. Priority = (6-2) + (1000/25) = 44.00
43   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 745, Priority 71.67), (Index 3: ID 633, Priority 44.00), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prio
44 ----------------------------------------------------------------------
45 SEARCH SUCCCESS: Found Patient 1602 at index 26.
46 INSERTING: Daniel Allen (ID: 1602) with U=4, T=60m. Priority = (6-4) + (1000/60) = 18.67
47   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 745, Priority 71.67), (Index 3: ID 633, Priority 44.00), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prio
48 ----------------------------------------------------------------------
49 SEARCH SUCCCESS: Found Patient 819 at index 26.
50 INSERTING: Michael Martinez (ID: 819) with U=3, T=90m. Priority = (6-3) + (1000/90) = 14.11
51   Heap State: [(Index 0: ID 112, Priority 103.00), (Index 1: ID 304, Priority 55.00), (Index 2: ID 745, Priority 71.67), (Index 3: ID 633, Priority 44.00), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prio
52 ----------------------------------------------------------------------
53
54 ### Step 3: Edge Case ###
55 SEARCH FAIL: Patient 999 not found.
56 INSERT FAILED: Patient 999 not found in records.
57 SEARCH SUCCCESS: Found Patient 101 at index 28.
58 INSERT FAILED: Invalid treatment time (0) for Patient 101.
59
```



```
Assignment > output > ≡ 3heap_results.txt
60 (Demo: Deleting patient No. 1010 from records...)
61 DELETE: Successfully removed Patient 1010.
62 SEARCH FAIL: Patient 1010 not found.
63 INSERT FAILED: Patient 1010 not found in records.
64 ----------------------------------------------------------------------
65
66 ### Step 4: Extracting Top 5 Priority Patients ---
67 Expected order: 112(103.0), 745(71.7), 304(55.0), 633(44.0), 101(37.3)
68 ----------------------------------------------------------------------
69
70 Extracting patient 1...
71 EXTRACTED: [ID: 112 (Peter Jones), Priority: 103.00, U: 3, T: 10m]
72   Heap State: [(Index 0: ID 745, Priority 71.67), (Index 1: ID 304, Priority 55.00), (Index 2: ID 451, Priority 24.22), (Index 3: ID 633, Priority 44.00), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prior
73 ----------------------------------------------------------------------
74
75 Extracting patient 2...
76 EXTRACTED: [ID: 745 (Linda Garcia), Priority: 71.67, U: 1, T: 15m]
77   Heap State: [(Index 0: ID 304, Priority 55.00), (Index 1: ID 633, Priority 44.00), (Index 2: ID 451, Priority 24.22), (Index 3: ID 1602, Priority 18.67), (Index 4: ID 101, Priority 37.33), (Index 5: ID 213, Prio
78 ----------------------------------------------------------------------
79
80 Extracting patient 3...
81 EXTRACTED: [ID: 304 (Mary Williams), Priority: 55.00, U: 1, T: 20m]
82   Heap State: [(Index 0: ID 633, Priority 44.00), (Index 1: ID 101, Priority 37.33), (Index 2: ID 451, Priority 24.22), (Index 3: ID 1602, Priority 18.67), (Index 4: ID 522, Priority 9.33), (Index 5: ID 213, Prior
83 ----------------------------------------------------------------------
84
85 Extracting patient 4...
86 EXTRACTED: [ID: 633 (Robert Wilson), Priority: 44.00, U: 2, T: 25m]
87   Heap State: [(Index 0: ID 101, Priority 37.33), (Index 1: ID 1602, Priority 18.67), (Index 2: ID 451, Priority 24.22), (Index 3: ID 819, Priority 14.11), (Index 4: ID 522, Priority 9.33), (Index 5: ID 213, Prior
88 ----------------------------------------------------------------------
89
90 Extracting patient 5...
91 EXTRACTED: [ID: 101 (John Smith), Priority: 37.33, U: 2, T: 30m]
92   Heap State: [(Index 0: ID 451, Priority 24.22), (Index 1: ID 1602, Priority 18.67), (Index 2: ID 213, Priority 21.67), (Index 3: ID 819, Priority 14.11), (Index 4: ID 522, Priority 9.33)]
93 ----------------------------------------------------------------------
94
95 ### Final Summary ###
96 Evidence: The extraction log clearly shows that patients with the highest computed
97           priority scores (e.g., 103.0, 71.7) were extracted first, regardless of the order they
98           were inserted in. This confirms the Max Heap function is serving higher-priority patients first consistently.
99
```

Module 4: Sorting Patients Records

Assignment > output > 4benchmark_results.txt

```
##############################################
###   MODULE 4: Sorting Patients Records   ###
##############################################

--- 1. Sample Sort Output (Size=100, Random) ---
Original (first 10): ['205m', '180m', '128m', '95m', '207m', '155m', '293m', '151m', '206m', '14m']
Sorted (first 10):   ['14m', '17m', '19m', '19m', '21m', '21m', '23m', '24m', '26m', '30m']
Sorted (last 10):    ['247m', '253m', '262m', '280m', '282m', '286m', '287m', '292m', '293m', '294m']

--- 2. Running Full Benchmark ---
Generating and testing datasets for size 100...
Generating and testing datasets for size 500...
Generating and testing datasets for size 1000...

================================================================================
--- Sorting Algorithm Benchmark Results ---
================================================================================
Algorithm    | Condition          | Size   | Time (ms)  | Comparisons  | Swaps        | Passed
--------------------------------------------------------------------------------
Merge Sort   | Random             | 100    | 0.100      | 542          | 0            | Yes
Quick Sort   | Random             | 100    | 0.102      | 707          | 482          | Yes
Merge Sort   | Reversed           | 100    | 0.085      | 322          | 0            | Yes
Quick Sort   | Reversed           | 100    | 0.103      | 839          | 531          | Yes
Merge Sort   | Nearly Sorted      | 100    | 0.095      | 502          | 0            | Yes
Quick Sort   | Nearly Sorted      | 100    | 0.091      | 798          | 419          | Yes
Merge Sort   | Random             | 500    | 0.656      | 3852         | 0            | Yes
Quick Sort   | Random             | 500    | 0.830      | 5446         | 3233         | Yes
Merge Sort   | Reversed           | 500    | 0.558      | 2371         | 0            | Yes
Quick Sort   | Reversed           | 500    | 0.872      | 6702         | 4162         | Yes
Merge Sort   | Nearly Sorted      | 500    | 0.693      | 3524         | 0            | Yes
Quick Sort   | Nearly Sorted      | 500    | 1.855      | 5257         | 2850         | Yes
Merge Sort   | Random             | 1000   | 1.428      | 8680         | 0            | Yes
Quick Sort   | Random             | 1000   | 8.544      | 11793        | 6866         | Yes
Merge Sort   | Reversed           | 1000   | 5.879      | 5469         | 0            | Yes
Quick Sort   | Reversed           | 1000   | 18.934     | 15279        | 9714         | Yes
Merge Sort   | Nearly Sorted      | 1000   | 9.196      | 7988         | 0            | Yes
Quick Sort   | Nearly Sorted      | 1000   | 4.297      | 12119        | 7083         | Yes
================================================================================

--- 3. Benchmark Analysis ---
Merge Sort (In-Place):
 - Consistent O(nlogn) performance. Time scales predictably with size.
 - Performance is almost identical for Random, Reversed, and Nearly Sorted data,
   proving it is not vulnerable to input order. Swaps are 0 as it copies.

Quick Sort (Median-of-Three):
 - Fastest on average (especially for Random and Nearly Sorted data).
 - The Mo3 pivot strategy successfully prevented $O(n^2)$ behavior on Reversed
   and Nearly Sorted lists, keeping performance at $O(n log n).
 - Shows the highest number of swaps, as it works in-place.

Conclusion:
 - For general-purpose sorting, QuickSort (with Mo3) is fastest.
 - If stability is required (preserving order of 30m, 30m), Merge Sort is the only choice.
 - If predictable, worst-case performance is critical, Merge Sort is safer.
```

6) Reflection: Key learnings and design trade-offs.

The project demonstrated that there is no single best algorithm or data structure. Every design choice made will have a trade-off. For example, average time for the hash table implemented in module 2 is a good example of trade-offs. Its great performance is only achieved because of computationally expensive, but rare resize() operation. Another example can be in module 3, ability to be fully sorted is is traded for ability to insert in O(logn) time. Moreover, in module 4, merge sort guarantees a time complexity of O(nlogn) at the cost of O(n) space complexity.

7) Limitations & Assumptions: Known issues and simplifications.

1. Assuming all input files exist in the correct directory and contain the exact headers as outlined in the python code.
2. Module 1 cannot account for real time changes like for example a corridor being inaccessible.
3. Reinsertion assumes the older lower priority request will eventually be deleted.