Project Report:

Data Structures and Algorithms for Hospital Efficiency

Name: Manh Tuan Duong

Student Number: 22802495

Date: 20/10/2025

Overview

This report proposed a hospital management system to demonstrate the application of data structures and algorithms. The system is designed to address patient data retrieval, pathfinding, treatment scheduling and sorting for analysis, including four modules:

- **Hospital Navigation Module**: A weighted and undirected graph represents the hospital's floor plan with nodes representing departments and edges representing corridor travel times. This module implements Best-First Search, Depth-First Search and the A* pathfinding algorithm.
- **Patient Lookup Module**: A hash table is implemented to store and manage patient records. The design aims for O(1) time complexity for insertion, searching, and deletion.
- **Treatment Scheduling Module**: A max heap is used to manage the treatment schedule. Priority is determined by combining the patient's urgency level and the expected treatment time.
- **Patient Record Sorting Module**: This module implements two sorting algorithms, which are Merge Sort and Quick Sort, to sort patient records by treatment time, with their performance evaluated across datasets of varying sizes.

This report also provides an analysis of each algorithm's time and space complexity.

Data Structures

2.1. Weighted Graph (DSAWeightedGraph)

A weighted and undirected graph is implemented to simulate the hospital's floor plan, where departments are represented as nodes and the pathways between them as edges.

Design and Representation:

- Adjacency List: The graph is implemented by an adjacency list.
- Weighted Edges: Each edge has a weight used as the walking time between two departments.
- Undirected: All pathways are undirected to ensure symmetric travel times between two departments.

Implemented Algorithms:

- Breadth-First Search (BFS): Used to identify reachable departments from a starting point and group them by level.
- Depth-First Search (DFS): Used to detect whether there is a cycle from a starting point, together with the nodes involved.
- A^* Pathfinding Algorithm: Used to find the shortest path from a starting point to a destination department.

Applications:

This module addresses key challenges of the hospital by:

- Optimising Patient Transfers: Always returning the quickest way to move patients between two units.
- Analysing Hospital Layout: Using BFS to have an overview of departments and DFS to identify confusing cyclical pathways.

2.2. Hash Table (PatientHashTable)

A hash table is implemented to provide an O(1) time complexity for patient record management. Each patient record includes Patient ID (unique key), Name, Age, Department, Urgency (rated 1-5) and Treatment Status.

Technical Implementation:

- Hash Function: A hash function is used to store the data, and the table size is maintained as a prime number to help distribute keys and reduce collisions.
- Collision Resolution: This implementation utilises linear probing to resolve hash collisions.
- Deletion: Deletions are handled using a "tombstone" marker to ensure that records can be removed without breaking the probe sequence.
- Load Management: The table is designed to automatically resize into a larger array if the load factor exceeds the threshold (0.7).

Performance and Operations:

The hash table supports the following operations, which are demonstrated to handle collisions correctly:

- insert(record): Adds a new patient. Expected time: O(1).
- search(patientID): Search for a patient record. Expected time: O(1).
- delete(patientID): Removes a patient record. Expected time: O(1).

Applications: This module allows staff to access a patient's status or urgency level, which is used to support the Treatment Scheduler in the following module.

2.3 Priority Heap (TreatmentHeap)

A max heap is implemented to serve as a priority queue, ensuring that the patient with the highest-priority metric is always processed first.

Design and Implementation:

- Max Heap: A Max Heap is implemented. This ensures that the patient with the highest priority is always at the top of the tree.
- Priority Calculation: The priority of each patient record is calculated using a formula: Priority = (6 - U) + 1000 / T, where U is the Urgency (1-5) and T is the expected treatment time (evaluated by each patient’s health situation).

Core Operations and Complexity:

The Treatment Heap supports the following operations:

- insert(): Adds a new patient record. The priority score is calculated, and then it is added to the end of the array. The heap property is then restored using a “trickle-up” operation.
- peek(): Returns the item with highest priority without modifying the heap.
- extract_priority(): Returns the item with highest priority and removes it from the heap. Then the last element is moved to the top, and “trickle-down” operation is performed.
- update_priority(): The treatment priority can be updated. This process involves finding the element in the heap, modifying its priority, and then performing a heapify operation to restore the heap property.

2.4 Sorting Algorithms (DSAsorts)

This module implements two sorting algorithms: MergeSort and QuickSort, to organise the patient records by treatment time and to evaluate their performance under various conditions.

Merge Sort

- Implementation: A recursive approach is used to provide a direct and logical implementation of the divide-and-conquer strategy.
- Time Complexity: O(nlogn) in all scenarios.
- Space Complexity: O(n) due to additional storage for the merging process.
- Stability: This implementation ensures stability, meaning that patients with similar treatment times will maintain their original order.

Quick Sort

- Implementation: This report uses a Median-of-Three pivot strategy to mitigate the possibility of the O(n^2) worst-case scenario, which may occur with already-sorted or reversed data.
- Time Complexity: O(nlogn) in the average and best cases. The worst-case is O(n^2).
- Space Complexity: O(logn) on average.

Application: The sorting algorithm is applied in the analytics of patient records at the end of the day.

3. Module Integration

The system is designed as a set of core data structure modules. The following sections detail the data flow within each module and between the core logic and its test harness.

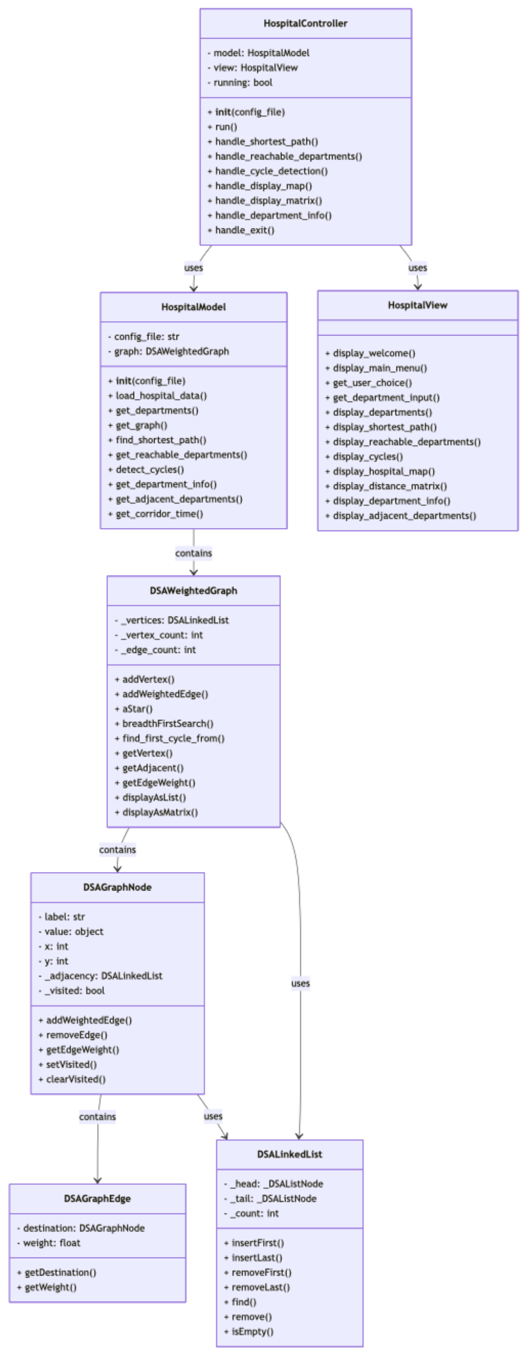
3.1. Hospital Navigation Module

This module's implementation separates the core algorithmic logic from the user-facing test harness. The core logic is encapsulated in the ***DSAWeightedGraph*** class. The ***HospitalModel***, ***HospitalController***, and ***HospitalView*** classes are used to gather user input and generate the textual outputs.

Execution & Data Flow:

The flow for this module is as follows:

1. Input: The ***HospitalView*** captures the inputs from the user.
2. Orchestration: The request is passed to the ***HospitalController***, which calls the methods on the **HospitalModel** respectively.
3. Algorithm Execution: The ***HospitalModel*** invokes the core logic: e.g. DSAWeightedGraph.aStar(). The *aStar()* algorithm executes and returns the raw results (the path and total cost) to the ***HospitalModel***.
4. Output: The ***HospitalModel*** passes the data back to the ***HospitalController***, which instructs the ***HospitalView*** to format and display the final textual output.



3.2. Patient Lookup Module

The core of this module is the ***PatientHashTable*** class (which inherits from a base ***DSAHashTable***), built to provide O(1) expected time for record operations . The ***PatientController*** and ***PatientView*** classes function as the interactive test driver, responsible for demonstrating the core logic and generating the "Log of inserts, searches, and deletes" and "collision example" outputs.

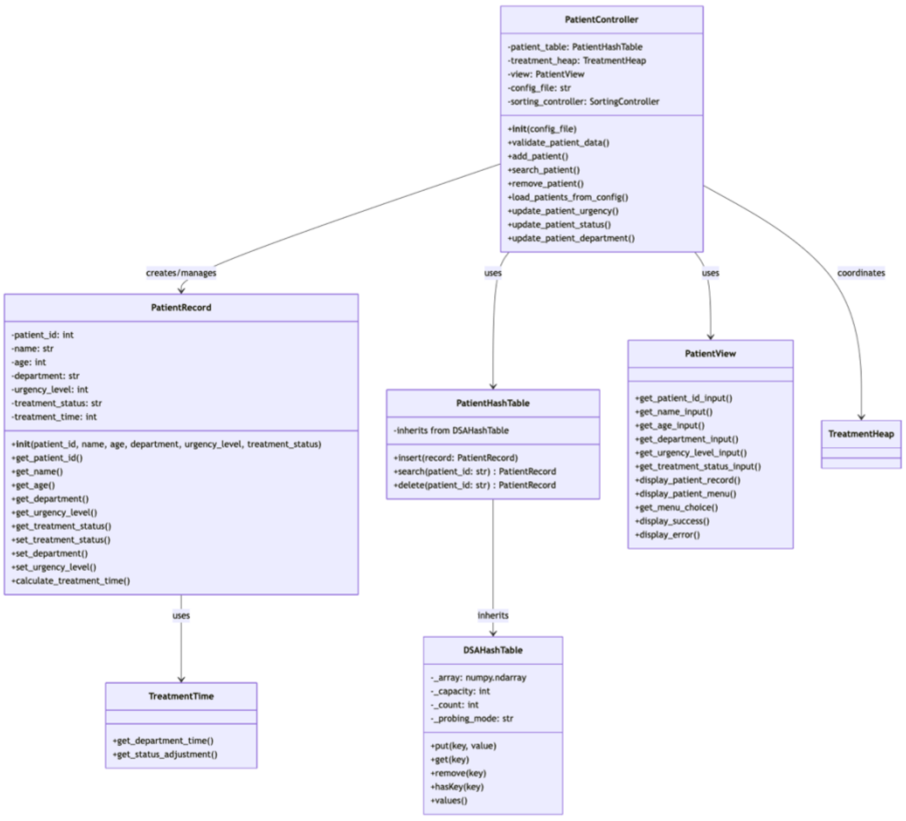
Execution & Data Flow:

The flow for demonstrating the hash table's core operations is as follows:

- Insertion (insert):
    1. The ***PatientView*** receives a patient record, including ID, Name, Urgency, etc.
    2. The ***PatientController*** validates the inputs and creates a new PatientRecord object.

3. The controller calls *PatientHashTable.insert(record).* The underlying ***DSAHashTable*** computes the hash index, resolves any collisions using linear probing, and places the record.
4. The PatientView logs the successful insertion and provides a trace of the collision and probe sequence if one occurred.

- Search (search):
    1. The ***PatientView*** receives a patientID.
    2. The ***PatientController*** calls *PatientHashTable.search(patientID),* which executes the search logic.
    3. The ***PatientView*** displays either the patient record or a not found message.
- Deletion (delete):
    1. Input: The ***PatientView*** captures the patientID to be removed.
    2. Execution: The ***PatientController*** calls *PatientHashTable.delete(patientID).*
    3. Output: The ***PatientView*** confirms the deletion.



3.3. Treatment Scheduling Module

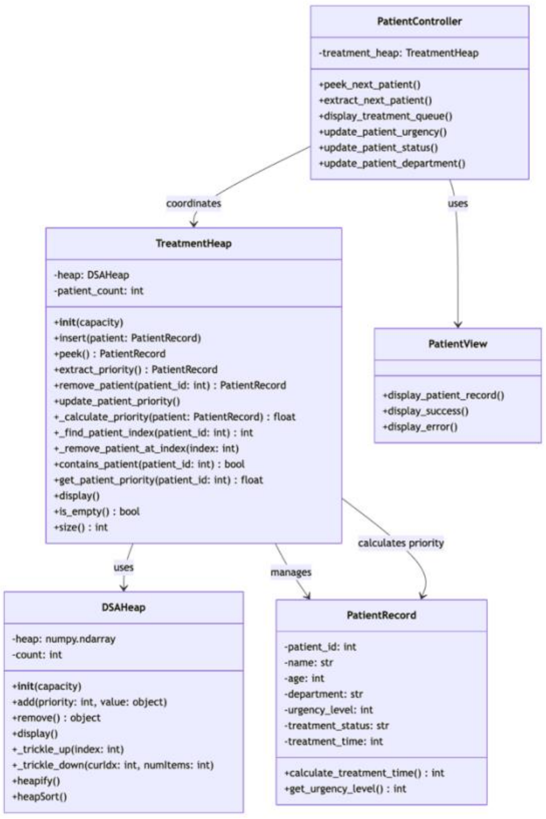
The core logic of this module is encapsulated in ***TreatmentHeap***. The ***PatientController*** and ***PatientView*** act as the test driver.

Execution & Data Flow:

- Insertion and Priority Calculation:
    1. Whenever a new patient record is inserted into the Hash Table, the ***PatientController*** inserts it into the ***TreatmentHeap***.
    2. The controller computes the priority using the formula: Priority = (6 - U) + 1000 / T.

3. The PatientRecord and the priority are passed to the *TreatmentHeap.insert()* method, which executes the insertion logic.
4. The ***PatientController*** then notifies the ***PatientView*** to print the successful state and current state of the heap array.

- Extraction:
    1. The ***PatientController*** requests the next patient.
    2. The controller calls *TreatmentHeap.extract_priority()*, which performs removing a record and trickling-down.
    3. The PatientRecord with the highest priority is returned to ***PatientView***.

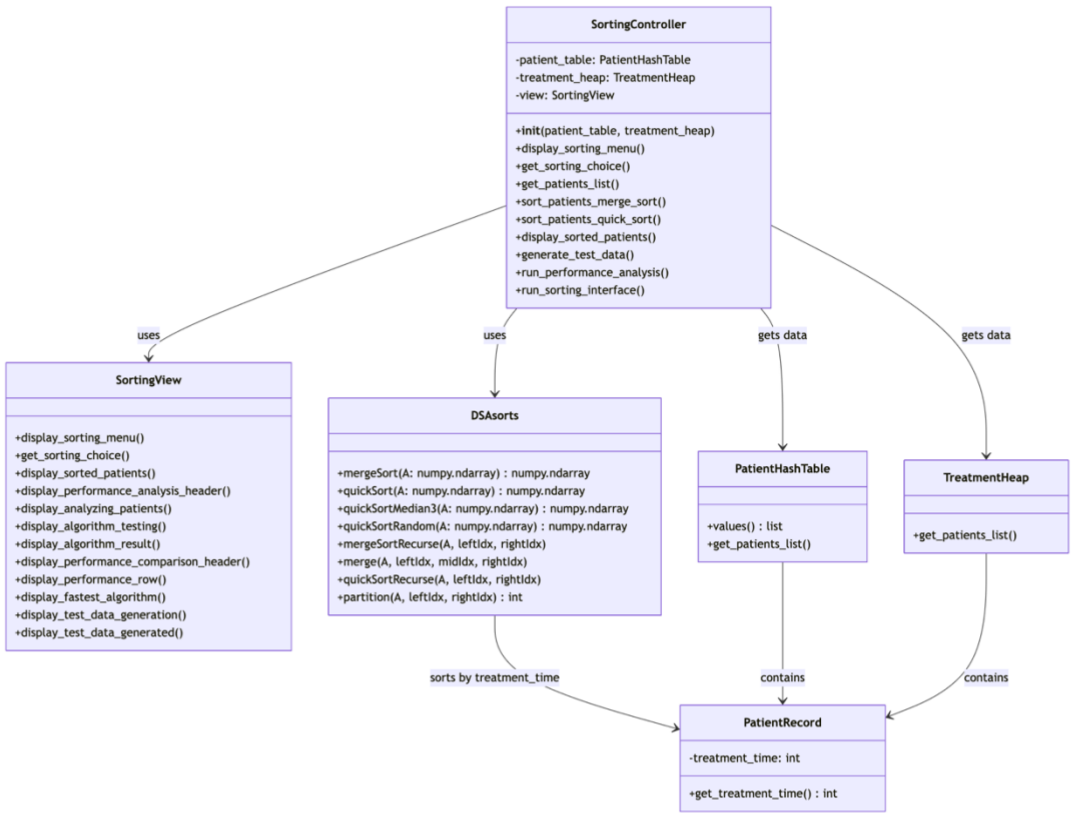

PatientController
-treatment_heap: TreatmentHeap
+peek_next_patient()
+extract_next_patient()
+display_treatment_queue()
+update_patient_urgency()
+update_patient_status()
+update_patient_department()
coordinates
uses
TreatmentHeap
-heap: DSAHeap
-patient_count: int
+init(capacity)
+insert(patient: PatientRecord)
+peek() : PatientRecord
+extract_priority() : PatientRecord
+remove_patient(patient_id: int) : PatientRecord
+update_patient_priority()
+_calculate_priority(patient: PatientRecord) : float
+_find_patient_index(patient_id: int) : int
+_remove_patient_at_index(index: int)
+contains_patient(patient_id: int) : bool
+get_patient_priority(patient_id: int) : float
+display()
+is_empty() : bool
+size() : int
PatientView
+display_patient_record()
+display_success()
+display_error()
uses
manages
calculates priority
DSAHeap
-heap: numpy.ndarray
-count: int
+init(capacity)
+add(priority: int, value: object)
+remove() : object
+display()
+_trickle_up(index: int)
+_trickle_down(curIdx: int, numItems: int)
+heapify()
+heapSort()
PatientRecord
-patient_id: int
-name: str
-age: int
-department: str
-urgency_level: int
-treatment_status: str
-treatment_time: int
+calculate_treatment_time() : int
+get_urgency_level() : int

3.4. Patient Record Sorting and Benchmarking

This module implements and analyses the performance of Merge Sort and Quick Sort. The core logic is encapsulated in the ***DSAsorts*** class. The ***SortingController*** and ***SortingView*** serve as the test harness.

Execution & Data Flow:

- Test Data Generation: the ***SortingController*** creates lists of ***PatientRecord*** for sizes 100, 500, and 1,000 according to user input.
- Algorithm Execution & Timing: The ***SortingController*** passes the data to the *DSAsorts.mergeSort()* and *DSAsorts.quickSort()*.
- The ***SortingController*** aggregates the results, notifies the ***SortingView*** to display the performance comparison.


SortingController
-patient_table: PatientHashTable
-treatment_heap: TreatmentHeap
-view: SortingView
+init(patient_table, treatment_heap)
+display_sorting_menu()
+get_sorting_choice()
+get_patients_list()
+sort_patients_merge_sort()
+sort_patients_quick_sort()
+display_sorted_patients()
+generate_test_data()
+run_performance_analysis()
+run_sorting_interface()
uses
uses
gets data
gets data
SortingView
+display_sorting_menu()
+get_sorting_choice()
+display_sorted_patients()
+display_performance_analysis_header()
+display_analyzing_patients()
+display_algorithm_testing()
+display_algorithm_result()
+display_performance_comparison_header()
+display_performance_row()
+display_fastest_algorithm()
+display_test_data_generation()
+display_test_data_generated()
DSAsorts
+mergeSort(A: numpy.ndarray) : numpy.ndarray
+quickSort(A: numpy.ndarray) : numpy.ndarray
+quickSortMedian3(A: numpy.ndarray) : numpy.ndarray
+quickSortRandom(A: numpy.ndarray) : numpy.ndarray
+mergeSortRecurse(A, leftIdx, rightIdx)
+merge(A, leftIdx, midIdx, rightIdx)
+quickSortRecurse(A, leftIdx, rightIdx)
+partition(A, leftIdx, rightIdx) : int
PatientHashTable
+values() : list
+get_patients_list()
TreatmentHeap
+get_patients_list()
sorts by treatment_time
contains
contains
PatientRecord
-treatment_time: int
+get_treatment_time() : int

4. Algorithm Complexity Analysis

This section provides the time and space complexity analysis for the core algorithms implemented in each module.

4.1. Hospital Navigation Analysis

The navigation module relies on three graph algorithms. In this analysis, V represents the number of vertices (departments) and E represents the number of edges (corridors).

Breadth-First Search (BFS):

- Purpose: Implemented to find all reachable departments from a source and group them by level.
- Time Complexity: O(V+E). Each department and corridor is visited exactly once in the traversal.
- Space Complexity: O(V). This is determined by the maximum size of the queue used to store nodes, which in the worst case could hold nearly all vertices.

Algorithm	Scenario	Time Complexity	Space Complexity	Notes
BFS	Best Case	0(1)	0(1)	Start = Goal
BFS	Average Case	O(V + E)	O(V)	Systematic exploration
BFS	Worst Case	O(V + E)	O(V)	Complete graph traversal

Depth-First Search:

- Purpose: Implemented to detect the presence of cycle from a starting point and report the nodes involved.
- Logic: The algorithm traverses the graph by exploring as deeply as possible along each branch before backtracking. To detect a cycle, the algorithm maintains a "visited" set and tracks the "parent" of the current node. A cycle is detected if the traversal encounters an adjacent node that is already in the "visited" set but is not the immediate parent of the current node.
- Time Complexity: O(V+E). Each department (vertex) and corridor (edge) is visited exactly once.
- Space Complexity: O(V). In the worst case, the stack could grow to the height of V if the graph is a single long path.

Algorithm	Scenario	Time Complexity	Space Complexity	Notes
DFS	Best Case	0(1)	0(1)	Start = Goal
DFS	Average Case	O(V + E)	0(V)	Recursion stack depth
DFS	Worst Case	O(V + E)	O(V)	Deep recursion path

A* Shortest Path Algorithm

- Purpose: Implemented to find the shortest path in a weighted graph, according to Hart et al. (1968).
- Core Concept: A* finds the shortest path by combining the actual cost to reach a node with an estimated cost to the goal. It prioritizes nodes using the cost function *f(n)=g(n)+h(n),* where:

  g(n): The actual known cost from the start node to node n.

  h(n): The heuristic estimate of the cost from node n to the goal.

- Heuristic Function: This report implements a heuristic function that returns the direct edge weight if a node is adjacent to the goal, and returns 0 in all other cases. This ensures admissibility, which means it doesn’t overestimate the actual travel cost.
- Time Complexity: O((E+V)logV). In the worst case, the algorithm functions like Dijkstra's. Every edge may need to be processed, and for each edge, a node's priority might be updated in the min-priority queue.
- Space Complexity: O(V+E). This space is required to store the Open and Closed sets, which in the worst case could hold all vertices, as well as the data structures used to reconstruct the final path.

Algorithm	Scenario	Time Complexity	Space Complexity	Notes
A*	Best Case	O(d)	O(d)	Perfect heuristic, direct path
A*	Average Case	O(bAd)	O(bAd)	Good heuristic guidance
A*	Worst Case	O((V + E) log V)	O(V)	Degenerates to Dijkstra

4.2. Patient Management Module

4.2.1. Patient Lookup

Implementation Strategy:

- The system implements a hash table using open addressing with linear probing to resolve collisions. This strategy was chosen for its implementation simplicity and favorable cache performance (compared to chaining), as all data is stored contiguously in a single array.
- Collision Handling: When an insert operation hashes to an occupied slot, the algorithm probes the next slot (index i+1, i+2,… modulo table size) until an empty slot is found.
- Deletion: Deletions are managed using "tombstones". A deleted slot is marked but not left empty, ensuring that subsequent search operations can still probe past it to find keys that were inserted later in the same cluster.
- Load Factor Management: The table is designed to resize into a larger table when the load factor (α) exceeds the threshold (0.7).

Complexity:

- insert(record) and search(patientID):
    - Best Case Time: O(1). This occurs when the hash index is empty or already contains the correct key.
    - Average Case Time: O(1). This assumes a reasonably low load factor. More precisely, the expected number of probes is $O(1/(1-\alpha))$.
    - Worst Case Time: O(n). This occurs when all keys hash to the same cluster, forcing the algorithm to scan all n elements in the table.
    - Space Complexity: O(k), where k is the capacity of the array. k is O(n) as the load factor is maintained below a constant threshold (e.g., 0.7).
- delete(patientID):
    - Best Case Time: O(1). The element is found at its initial hash index.
    - Average Case Time: O(1). Identical to the search operation.
    - Worst Case Time: O(n). The element is at the end of a single large cluster.
- Resizing: Time Complexity: O(n). It requires allocating a new, larger array and re-hashing all n existing elements into their new positions.

4.2.2. Treatment Priority Scheduler

The scheduler is implemented as an array-based binary max-heap. This structure is chosen to ensure that the patient with the highest priority score is always at the root.

- insert(request)
    - Logic: The new request, with its priority calculated using the specified formula, is added to the end of the array. A "trickle-up" operation restores the max-heap property.
    - Best Case Time: O(1). This occurs if the new element does not violate the heap property (e.g., it is inserted as a leaf and has a lower priority than its parent), requiring no percolation.
    - Average/Worst Case Time: O(logn). The new element is added to the end, and the "trickle-up" operation may travel the full height of the tree, which is logn.

- extract_priority()
    - Logic: The root element (highest priority) is removed and returned. The last element in the array is moved to the root, and a "trickle-down" operation restores the heap property.
    - Best/Average/Worst Case Time: O(logn). This operation always removes the root, moves the last element, and then performs a "trickle-down," which takes logarithmic time.
- peek()
    - Best/Average/Worst Case Time: O(1). This operation simply reads and returns the element at the first index of the array (the root).
- update_priority() (Increase/Decrease-Key Strategy)
    - Logic: To support patient priority updates, the algorithm first locates the patient in the heap array via a linear search (O(n)). The priority is then recalculated and updated in-place. To restore the heap invariant, the algorithm compares the new priority with its parent/children, executing a trickle-up (if priority increased) or trickle-down (if priority decreased) as needed (O(logn)).
    - Best Case Time: O(1). This occurs if the patient is found at the beginning of the array (e.g., at the root) and the priority update does not require any percolation.
    - Average/Worst Case Time: O(n). The dominant cost is the linear search (O(n)) required to find the patient in the heap array. The subsequent O(logn) heap-fix operation is masked by the search cost.

Space Complexity (Overall): Best/Average/Worst Case: O(n). The heap requires an array to store all n patient requests.

4.2.3. Patient Sorting Algorithm

This module implements and analyzes two sorting algorithms, Merge Sort and Quick Sort, to organize records by treatment duration.

- Merge Sort: Best/Average/Worst Case: O(nlogn): The algorithm must always divide the array to the base case and then merge all sub-arrays, regardless of the initial order. This implementation is stable, which means patients with similar treatment times will maintain their original order.
- Quick Sort: This version employs a Median-of-Three pivot strategy.
    - Best/Average Case: O(nlogn). This occurs when the pivot index divides the array into two equal halves.
    - Worst Case: $O(n^2)$. This occurs if the pivot index creates unbalanced partitions.

5. Sample Output

5.1. Main Application Entry Point

```
HOSPITAL MANAGEMENT SYSTEM
============================================================
1. Hospital Navigation (Module 1)
2. Patient Management System (Module 2)
3. Run Test Suite
4. Exit
============================================================
```

5.2. Hospital Navigation Module

5.2.1. Main Navigation Module

```
MAIN MENU
------------------------------
1. Find Shortest Path
2. Get Reachable Departments
3. Detect Cycles
4. Display Hospital Map
5. Display Distance Matrix
6. Department Information
7. Exit
------------------------------
```

5.2.2. Shortest Path Finding

```
FIND SHORTEST PATH
------------------------------

Available Departments:
----------------------------------------
 1. Emergency
 2. Reception
 3. ICU
 4. Outpatient
 5. Pharmacy
 6. Laboratory
 7. Radiology
 8. Operating Theatre
 9. Wards
10. Isolated Department
----------------------------------------
Enter starting department: Emergency
Enter destination department: Laboratory

SHORTEST PATH: Emergency → Laboratory
==================================================
Path: Emergency → Reception → Outpatient → Laboratory
Total Walking Time: 24 minutes
Number of Corridors: 3

Step-by-Step Directions:
  1. From Emergency to Reception
  2. From Reception to Outpatient
  3. From Outpatient to Laboratory

============================================================
```

5.2.3. Reachable Departments

```
GET REACHABLE DEPARTMENTS
------------------------------

Available Departments:
----------------------------------------
 1. Emergency
 2. Reception
 3. ICU
 4. Outpatient
 5. Pharmacy
 6. Laboratory
 7. Radiology
 8. Operating Theatre
 9. Wards
```

```
10. Isolated Department
----------------------------------------
Enter starting department: Emergency

REACHABLE DEPARTMENTS FROM: Emergency
==================================================

Level 0 (Walking time: 0 corridors away):
  Departments: Emergency

Level 1 (Walking time: 1 corridors away):
  Departments: Reception, ICU

Level 2 (Walking time: 2 corridors away):
  Departments: Outpatient, Pharmacy, Operating Theatre, Radiology

Level 3 (Walking time: 3 corridors away):
  Departments: Laboratory, Wards

============================================================
```

5.2.4. Cycle detection

```
CYCLE DETECTION
------------------------------

Available Departments:
----------------------------------------
 1. Emergency
 2. Reception
 3. ICU
 4. Outpatient
 5. Pharmacy
 6. Laboratory
 7. Radiology
 8. Operating Theatre
 9. Wards
10. Isolated Department
----------------------------------------
Enter starting department: Emergency

CYCLE DETECTION FROM: Emergency
==================================================
1 cycle(s) found that include 'Emergency':
  Cycle 1: Emergency → Reception → Outpatient → Laboratory → Radiology → ICU → Emergency

============================================================
```

5.3. Patient Management Module

5.3.1. Patient Lookup Module

```
PATIENT LOOKUP SYSTEM
============================================================
Sub-module 2.1: Patient Lookup
============================================================
1. Add Patient
2. Search Patient
3. Remove Patient
4. Update Patient Information
5. Display All Patients
6. Back to Module 2 Menu
============================================================
```

5.3.2. Treatment Scheduler Module

```
TREATMENT SCHEDULER SYSTEM
============================================================
Sub-module 2.2: Treatment Scheduler
============================================================
1. Peek Next Patient (View Next in Queue)
2. Extract Next Patient (Remove from Queue)
3. Display Treatment Queue
4. Update Patient Priority
5. Back to Module 2 Menu
============================================================
```

5.3.3. Patient Sorting Module

```
PATIENT RECORD SORTING SYSTEM
============================================================
Sub-module 2.3: Patient Record Sorting
============================================================
1. Merge Sort (by Treatment Duration)
2. Quick Sort - Leftmost Pivot (by Treatment Duration)
3. Quick Sort - Median of Three (by Treatment Duration)
4. Quick Sort - Random Pivot (by Treatment Duration)
5. Performance Analysis
6. Generate Test Data
7. Back to Module 2 Menu
============================================================
```

6. Reflection

This assignment provided a practical insight into the application of core data structures contributing to a functional and efficient system.

Key learnings:

The first key learning was the implementation of A* heuristic function. An admissible heuristic is required to get the quickest pathways.

The second key learning was the integration of data structures within a system, where no module was standalone. This project demonstrated that the efficiency of a system is not just about one optimal algorithm, but about the design of the data flow between them.

Key Design Trade-offs:

Module 2 (Hash Table): Linear probing brings simplicity, as all data is stored in a single array. However, as the load factor increases, performance will degrade rapidly. A chaining strategy using linked lists is less likely to lead to performance degradation.

Module 3 (Heap): Update-Key Complexity. Heaps provide optimal O(logn) insert and extract_priority operations but are not designed for efficient updating. To implement the update_priority function, a linear O(n) search was required to find the patient within the heap.

Module 4 (Sorting): Merge Sort proved to be the most reliable, with O(nlogn) time complexity in all cases (worst, average, and best). However, this reliability came at the cost of O(n) space. Quick Sort offered O(logn) space (in-place) and is often faster in practice. However, it carried the significant risk of O(n^2) worst-case performance. The choice to implement a Median-of-Three pivot was a trade-off: it adds a small overhead to each partitioning step but mitigates the risk of the worst-case scenario.

7. Limitations and Assumptions

This report introduces a system to demonstrate core data structures and algorithms, which still has some simplifications and known limitations.

Assumptions:

- Static Hospital Layout: The graph model of the hospital is assumed to be static . The system does not account for real-time changes in "walking times", such as corridor congestion, elevator delays, or temporary closures. The shortest path is based purely on pre-defined edge weights.
- In-Memory Data Storage: All data structures (graph, hash table, and heap) operate entirely in-memory, therefore all patient records and scheduler data are lost on restart.
- Single-threaded Operation: The system is designed as a single-user, single-threaded application. It does not implement any concurrency controls which would be critical in a real-world (e.g., multiple nurses accessing the patient list simultaneously).

Known limitations:

- Inefficient Heap Updates (Module 3): The update_priority operation in the heap scheduler has a O(n) time complexity. This is a limitation of using a simple array-based heap, as locating a specific patient to update requires a linear search.
- Quick Sort Worst-Case (Module 4): While a Median-of-Three pivot strategy was implemented to mitigate risk, the Quick Sort algorithm still retains a theoretical $O(n^2)$ worst-case time complexity. This makes it less reliable than Merge Sort for mission-critical reporting where a performance guarantee is required.

Future work:

- O(logn) Heap Priority Updates (Module 3): Future work involves storing a PatientID as the key and its current index in the heap array as the value. This would allow the update_priority operation to find any patient in O(1) expected time, reducing the total update complexity from O(n) to O(logn).
- Optimized Hash Table Collision Strategy (Module 2): Future work implements a more robust collision strategy such as separate chaining using linked lists. This ensures more consistent O(1) expected time operations.

Reference

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136