Data structures final report

My humble submission for the final report of Data Structures.

Yuman |

Contents

[HospitalManagement 4](#_Toc188905930)

[Register(Patient Info) 5](#_Toc188905931)

[Operations: 5](#_Toc188905932)

[Discharge(Patient Id) 6](#_Toc188905933)

[Operations 6](#_Toc188905934)

[reAdmitPatient 6](#_Toc188905935)

[Operations 6](#_Toc188905936)

[Waiting Queue 6](#_Toc188905937)

[Enqueue 7](#_Toc188905938)

[Operation 7](#_Toc188905939)

[Dequeue 7](#_Toc188905940)

[Operation 7](#_Toc188905941)

[Peek 7](#_Toc188905942)

[PrintRoom 7](#_Toc188905943)

[isEmpty 7](#_Toc188905944)

[Discharge Stack 8](#_Toc188905945)

[Push 8](#_Toc188905946)

[Pop 8](#_Toc188905947)

[Peak 8](#_Toc188905948)

[isFull 8](#_Toc188905949)

[Size 8](#_Toc188905950)

[isEmpty 9](#_Toc188905951)

[showDischarged 9](#_Toc188905952)

[Administration Tree 9](#_Toc188905953)

[Insert 9](#_Toc188905954)

[PostOrder 9](#_Toc188905955)

[Question 2 10](#_Toc188905956)

[Stack ADT for feature 3 10](#_Toc188905957)

[Discharge Stack 10](#_Toc188905958)

[Push 10](#_Toc188905959)

[Pop 11](#_Toc188905960)

[Peak 11](#_Toc188905961)

[isFull 11](#_Toc188905962)

[Size 11](#_Toc188905963)

[isEmpty 11](#_Toc188905964)

[showDischarged 11](#_Toc188905965)

[Question 3 11](#_Toc188905966)

[Question 4 13](#_Toc188905967)

[Merge Sort 13](#_Toc188905968)

[Selection Sort 14](#_Toc188905969)

[Question 5 15](#_Toc188905970)

[Circular Array 15](#_Toc188905971)

[Enqueue 15](#_Toc188905972)

[Dequeue 17](#_Toc188905973)

[Doubly Linked List 18](#_Toc188905974)

[Enqueue 18](#_Toc188905975)

[Dequeue 19](#_Toc188905976)

[Compare and Contrast 19](#_Toc188905977)

[The memory usage 19](#_Toc188905978)

[Scalability 20](#_Toc188905979)

[Implementation 20](#_Toc188905980)

[Applications 20](#_Toc188905981)

[Question 6 20](#_Toc188905982)

[Hospital Management 20](#_Toc188905983)

[Patient Registration Using the Linked List 20](#_Toc188905984)

[Non-Emergency Treatment using a Queue 20](#_Toc188905985)

[Scenario 2: Ali arrives to the hospital with a heart failure (emergency case) 21](#_Toc188905986)

[Emergency Care 21](#_Toc188905987)

[Discharge Stack 21](#_Toc188905988)

[Scenario 3: Emergency Fire Plan using Graph ADT and a Dijkstra Algorithm 21](#_Toc188905989)

[Hospital Ground Plan 21](#_Toc188905990)

[Dijkstra Algorithm 21](#_Toc188905991)

[Scenario 4: Employee management using a Binary Search Tree 21](#_Toc188905992)

[Question 7 22](#_Toc188905993)

[Dijkstra Algorithm 22](#_Toc188905994)

[Initialization 22](#_Toc188905995)

[Main Loop 22](#_Toc188905996)

[Bellman Ford 23](#_Toc188905997)

[Initialization 23](#_Toc188905998)

[Relaxation 23](#_Toc188905999)

[Detection of Negative Cycles 23](#_Toc188906000)

[Then its going to print the output 24](#_Toc188906001)

[Conclusion 25](#_Toc188906002)

[Question 8 25](#_Toc188906003)

[Binary Search Tree for Employee management 25](#_Toc188906004)

[Sorting Algorithms 26](#_Toc188906005)

[Shortest Path Algorithm 26](#_Toc188906006)

[Dijkstra Algorithm 26](#_Toc188906007)

[Question 9 27](#_Toc188906008)

[Time complexity: 27](#_Toc188906009)

[Sorted Arrays 27](#_Toc188906010)

[Binary Search Tree 27](#_Toc188906011)

[Conclusion 28](#_Toc188906012)

[Question 1 28](#_Toc188906013)

[First Call 28](#_Toc188906014)

[Second Call 28](#_Toc188906015)

[Third Call 28](#_Toc188906016)

[Fourth call 28](#_Toc188906017)

[Fifth call 29](#_Toc188906018)

[Question 2 29](#_Toc188906019)

[Example 1 29](#_Toc188906020)

[Question 3 30](#_Toc188906021)

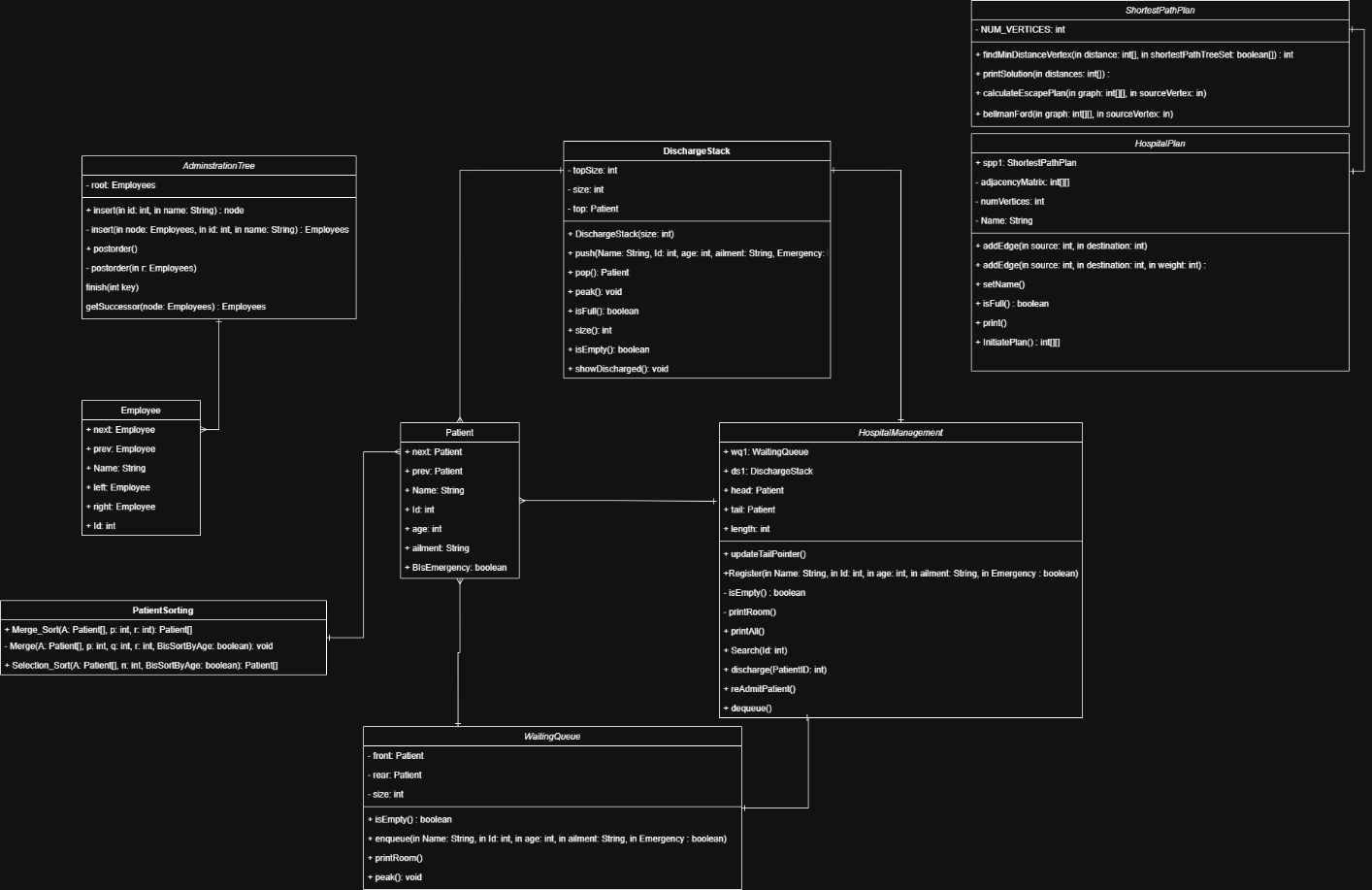
[Question 4 31](#_Toc188906022)

[Question 5 32](#_Toc188906023)

[Question 6 33](#_Toc188906024)

PART 1

Data Structure



# HospitalManagement

This is the Hospital Management class; this class controls everything from admitting patients to discharging them and then re-admitting them if they see fit and printing a current list of all current patients, discharged patients, and patients in the waiting queue. The backbone of the system is built using data structures such as: Linked List (Main backbone of the entire system), a queue to handle patients without emergencies, a stack to handle discharged patients.

First of all, I initialize 5 important variables, which are

This class interacts with other important classes and their own methods. This class is responsible for managing every operation related to the patients. The variables inside of this class are:

* Pointer to the Waiting Queue
  + This is a pointer that connects the Hospital Management system (Linked List) to a queue data structure that hold the patients that need medical attention but its not an emergency. This queue follows a First In First Out methodology.
* Pointer to the Discharge Stack
  + This pointer connects the Hospital Management system to a stack data structure that stores temporarily discharged patients from the Main Linked list. This stack follows Last In first Out methodology.
* 2 pointers (head, tails) to the Patient Class
  + These pointers connect the Management system to the patient class, which is used extensively to store the patient’s information which includes His name, age, Id, ailment, and if its an emergency or not.
* And just a length integer variable.
  + This is a normal integer variable used to manage the indexing of the linked list, its largely unused but might still prove useful in the future.

Then I define methods, the most important are:

* Register
* Discharge
* reAdmitPatient
* printAll
* updateTailPointer
* isEmpty
* printRoom

I will begin with explaining the methods

## Register(Patient Info)

Parameters =

* Name
* Id
* Age
* Ailment
* Emergency

This method/function register a new patient in the system, it allows the user to input the patient’s info then insert them into a linked list system. It checks if the patient needs emergency care, if yes, he is immediately registered in the linked list system. If not, he is added to a waiting queue.

In simpler terms, if the patients require emergency care they are prioritized, if not, they are going to wait.

### Operations:

It checks if the patient requires emergency care

It ensures that the patient with emergency is prioritized by being added to the linked list immediately

It ensures that the patient without emergency is added to the waiting queue.

## Discharge(Patient Id)

Parameters =

* PatientId

This method/Function is used to discharge a patient if they are treated. The patient is subsequently removed from the linked list, but his info is still stored, and is then pushed to the discharged stack.

### Operations

It checks the head’s id (the head is the first node in the stack) and compares it to the inputted discharged patient’s id, if they are the same, the head patient is discharged into the stack. This is done to account for a perfect scenario were the time complexity will be a O(1)

If its not the head’s id, its going to loop through all patients until it finds the patient with the same ID. Its then going to push it to the stack. Once pushed it will then make sure to nullify the patient from the linked list

## reAdmitPatient

This method/function is all about readmitting patients from the discharged stack. (Scenario) Its used when the health expert believe that the Patient needs more sessions with them.

### Operations

It checks if the discharge stack is empty, if not, we continue.

We pop the patient from the discharge stack, then we check if the patient needs emergency care, if true then the patient is placed ########### change this 🡨 ############# ontop of the linked list. If not the patient Is placed in the tail of the list.

Basic ADTS used in this class is

* Waiting Queue (Queue ADT)
* Discharge Stack (Stack ADT)
* AdminstrationTree (Binary Tree ADT)
* ShortestPathPlan (Graph ADT)

# Waiting Queue

This is the data structure that is responsible for holding and storing patients without the need for emergency treatment, it follows the first in, first out (FIFO) methodology

Varriables:

* Front – Rear : Pointers to the first and last patient (class) in the queue
* size

Methods:

* Enqueue
* Peek
* Dequeue
* isEmpty
* printRoom

## Enqueue

This method takes in the patient info (Name, id, age, etc) and then insert them into the waiting queue to make sure that any non-emergency patient is thrown into the waiting room in the correct order.

### Operation

This method adds the patient to the end of the queue, by always updating the rear pointer and increases the size of the queue.

## Dequeue

This method removes the next patient from the queue

### Operation

Retrieve the patient from the front of the queue and moves the front pointer to the next patient and then reduces the size of the queue

## Peek

This method allows the system to see the front patient without removing them

## PrintRoom

This method prints the all the patients in the queue by implementing a looping pointer that loops through all the patients inside the queue

## isEmpty

this method checks if there are any patients inside the queue

## Discharge Stack

This class is responsible for handling and storing temporarily discharged patients. It follows the first is last out, or last in first out (LIFO) methodology, making sure that the discharged patients’ information are easily handled and returned to the hospital if the medical expert sees fit

Variables

* topSize: track the amount of patients inside the stack.
* size: implement the maximum number of patients allowed to be discharged.
* top: Pointer to the patient class

Methods:

* push
* pop
* isEmpty
* showDischarged
* peak
* isFull
* size
* isEmpty

## Push

This method adds the discharged patient to the stack

It checks if the stack is full then adds the patient to the top of the stack, updates the top pointer and then increases the stack size

## Pop

This method returns the top patient and removes him from the stack

It checks if the stack is empty, if not then it gets the top of the sack patient and then moves the top to the next patient and reduces the stack size and then returns the previous top patient

## Peak

This method allows the system to see the top patient without removing them

## isFull

it checks if the size reached the top maximum size

## Size

Gets the current amount of discharged patients

## isEmpty

checks if the stack is empty

## 

## showDischarged

this method shows all discharged patients; it creates a looping pointer to loop over all the discharged patients

# Administration Tree

This class organize employees into a hierarchical structure using a binary Tree

Variables

* root: Pointer to the employees class

Methods

* Insert
* Postorder
* Postorder

## Insert

This method adds a new employee to the binary tree

It checks if the the binary tree is empty, if not it inserts the employee, by checking if the employee ID is larger or smaller than the current node, if bigger it goes to the right, if smaller it goes to the list, and keeps looping recursively until it reaches the end of the binary tree.

## PostOrder

Goes to the left subtree, then to the right subtree then to the node.

SortPatients

This sorts the patient via 2 algorithms, which are the merge sort and the selection sort

It sorts the patients by 2 criterions being either the age or the id.

It have 3 methods for merge sort

* Merge\_Sort
* Merge\_Age
* Merge\_ID

The merge sort method works recursively.

And the selection sort method which is only one methods works with a for loop and iterates on all of the dataset to find the lowest number and swap it with the first.

# Question 2

## Stack ADT for feature 3

The Stack ADT is a data structure that collects elements (Patients) with two very important methods being the Push and Pop method.

* Push : Adds elements into the top of the stack
* Pop : Removes the element and retrieve it from the top of the stack

Stack complies with the Last in First Out methodology (LIFO) meaning that the last element to be added to stack is the top one, meaning that its going to be the first to be removed. This is important for temporarily discharging and readmitting patient because in my opinion, the last discharged patient should be the first one to be considered/eligible for a readmission and this is the valid specification:

## Discharge Stack

This class is responsible for handling and storing temporarily discharged patients. It follows the first is last out, or last in first out (LIFO) methodology, making sure that the discharged patients’ information are easily handled and returned to the hospital if the medical expert sees fit

Variables

* topSize: track the amount of patients inside the stack.
* size: implement the maximum number of patients allowed to be discharged.
* top: Pointer to the patient class

Methods:

* push
* pop
* isEmpty
* showDischarged
* peak
* isFull
* size
* isEmpty

## Push

This method adds the discharged patient to the stack

It checks if the stack is full then adds the patient to the top of the stack, updates the top pointer and then increases the stack size

## Pop

This method returns the top patient and removes him from the stack

It checks if the stack is empty, if not then it gets the top of the sack patient and then moves the top to the next patient and reduces the stack size and then returns the previous top patient

## Peak

This method allows the system to see the top patient without removing them

## isFull

it checks if the size reached the top maximum size

## Size

Gets the current amount of discharged patients

## isEmpty

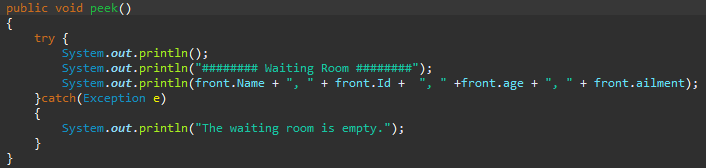
checks if the stack is empty

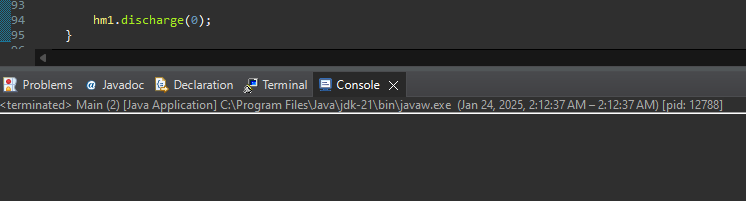
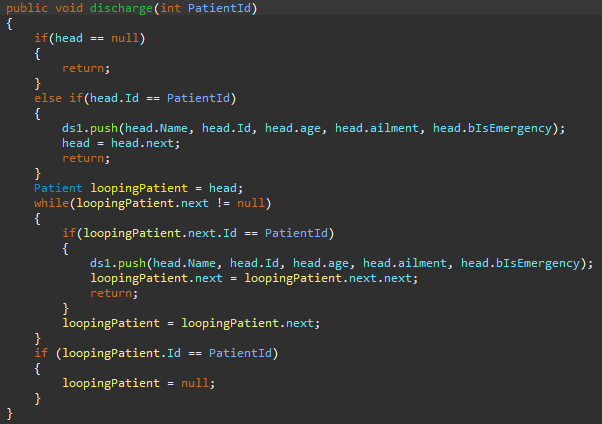
## showDischarged

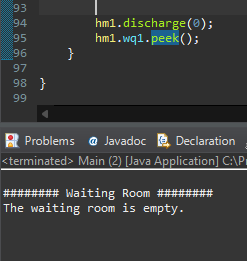
this method shows all discharged patients; it creates a looping pointer to loop over all the discharged patients

In my implementation, the stack is used to temporarily store discharged patients in the hospital system, the patient’s info (being his name, id, ailment, emergency) is pushed into the stack and if the patients needs readmission, their data is popped right back into the main structure (linked list) being the Hospital Management system.

# Question 3







A test case were I did not add any patient into the system and tried some basic operations to test my error handling and they are working by not pausing my entire system for a single error.

# Question 4

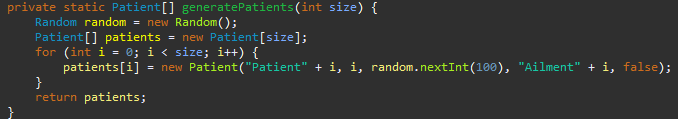
## Merge Sort

First of all, what does merge sort do? Merge sort in simple terms is a divide and conquer algorithm that splits the array into two halves recursively until it reaches 1 element from each array, then recursively sort them and merge them into sorted halves recursively until our array is sorted.

Time Complexity:

* Best Case: O(n log n)
* Worst Case: O(n log n)

First before showing any output, I must show how I populated my arrays



As seen here, we have random ages for each parent, but not random id, so I am going to sort by Age.

Outputs:

<- worst case, best case -> 

<- worst case, best case -> 

<- worst case, best case ->

So when comparing both test cases, and with a different input sizes, I can conclude that even though the time it takes increase between each iteration, and by testing both best and worst cases, the time complexity remains the same O(n log n). And as the number of patients increase, so does the the time it takes, but the time increase logarimically, and not exponentially like selection sort.







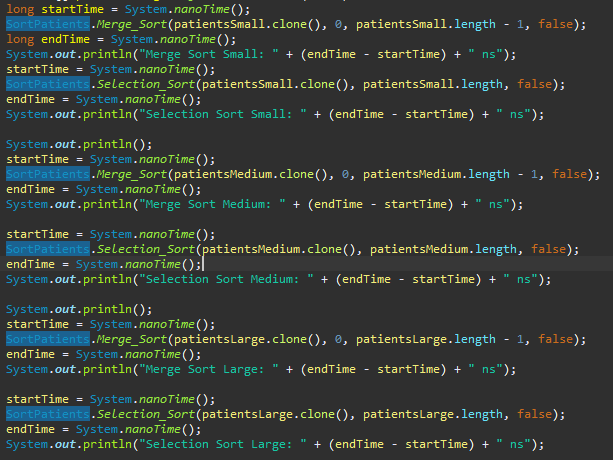
[Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks](https://www.geeksforgeeks.org/merge-sort/)

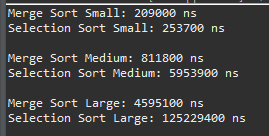
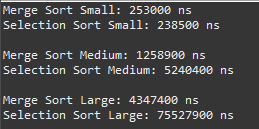
# Selection Sort

Selection Sort: First of all, selection sort algorithm works by repeatedly selecting the lowest/minimum value from the array and moves it to the first element of the array.

* Best case: O(n^2)
* Worst Case: O(n^2)

Output:



<-worst case, best case->

And as observed above, the time complexity of the selection sort is always higher than the merge sort, but I also did not test having an incredibly small dataset, which here is the output





And as seen here, selection sort struggles with having to sort a big dataset, but excells at sorting smaller datasets because it has less number of elements. But as the dataset grows, the selection sort compilation time increases exponential. Which is why selection sort is initially good for sorting smaller datasets but false into complete slowness as the dataset increases.

[Selection Sort - GeeksforGeeks](https://www.geeksforgeeks.org/selection-sort-algorithm-2/)

In conclusion, merge sort is the best for larger datasets as in 100+ entries, whilst the selection sort excels in smaller arrays 10+

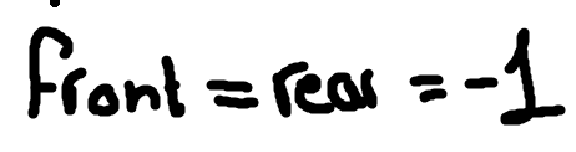
# Question 5

## Circular Array

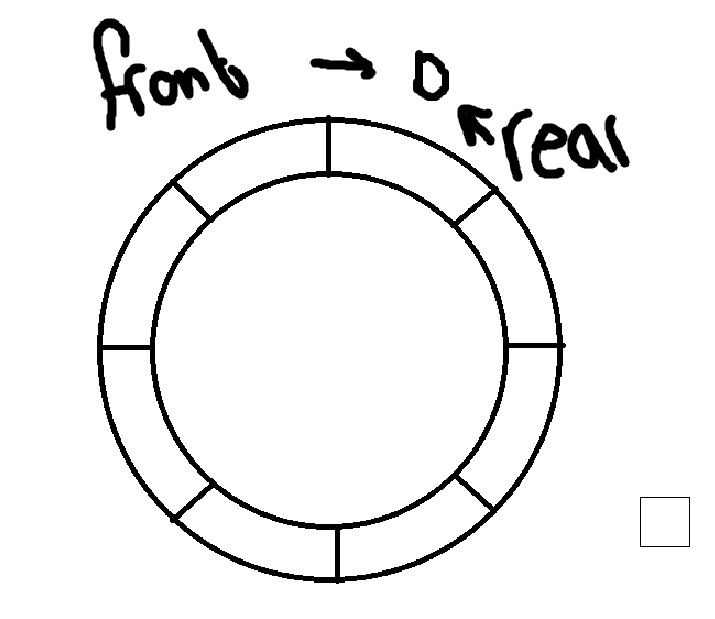
With a circular array we need to make sure that the elements wrap around as in a circle thereby reducing the space consumption seen in a linear queue

### Enqueue

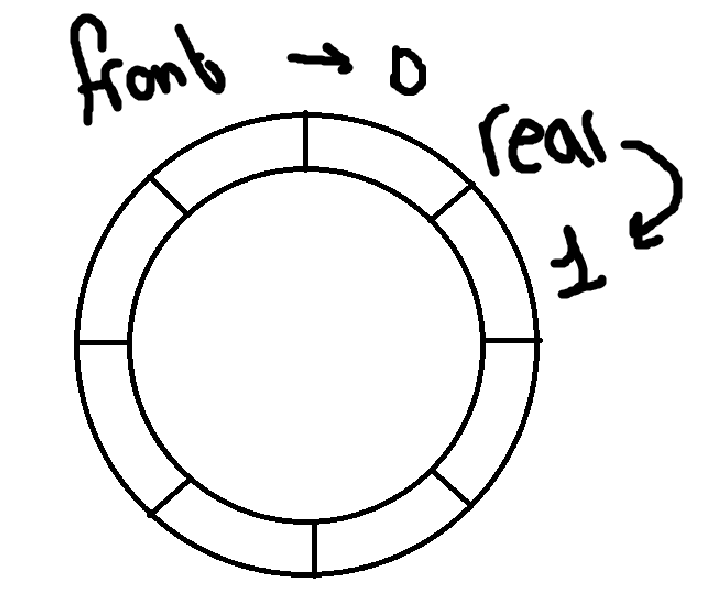
Front and rear is set to -1



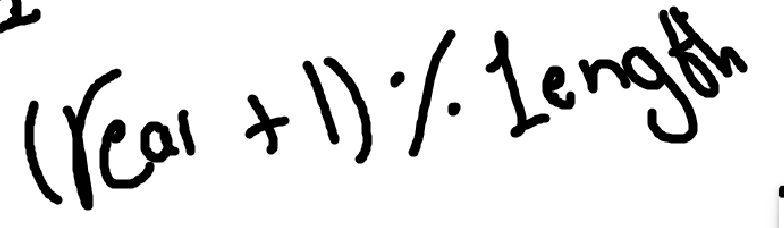
Once adding the first element inside the queue, we are gonna set the both of them to be that element as show in the image below

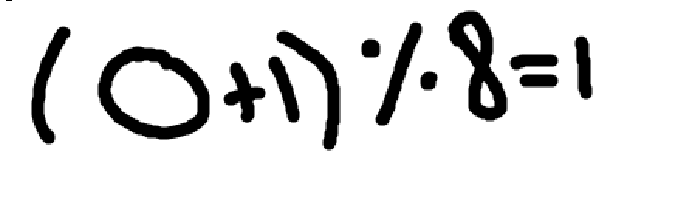


Enqueue once

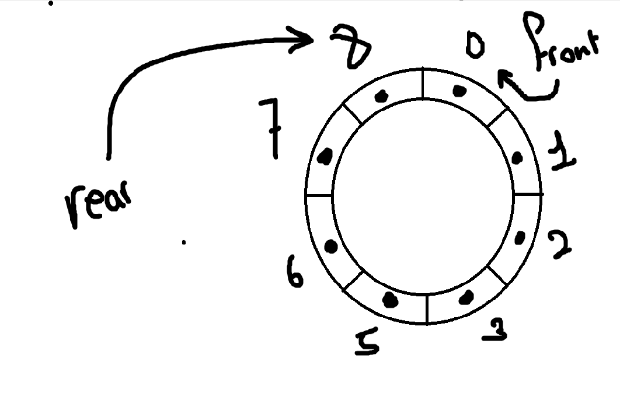


Now to determine the index of the rear, we are gonna use this equation

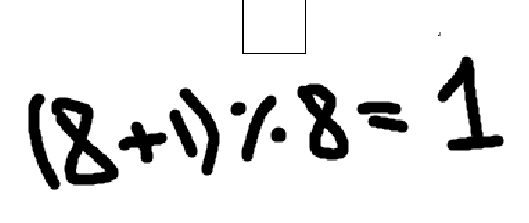


 And as we see with this equation, we determined that the rear’s index should be 1.

Lets enqueue 7 more times, rear is now index #8

****

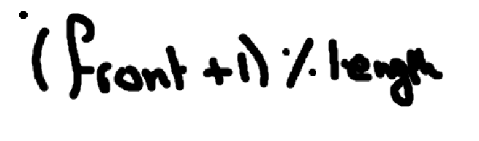
If we enqueue one more time, what will the rear’s index be? Lets see



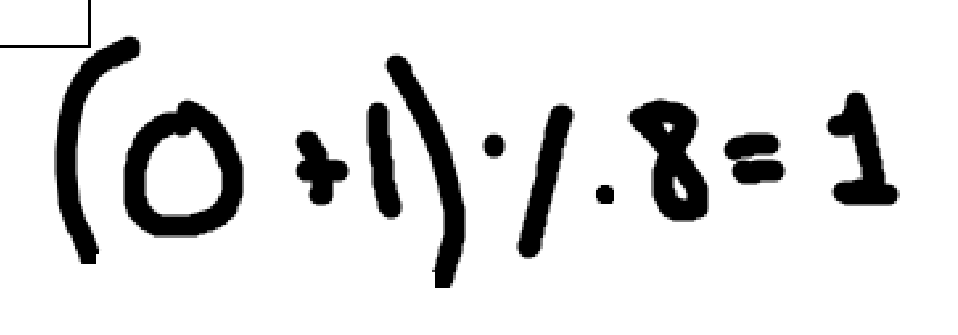
Interesting, when reaching the limit, and enqueue one more time, we wrap around the 0 as in skipping it and going straight to 1, and if we continue it we are gonna loop again.

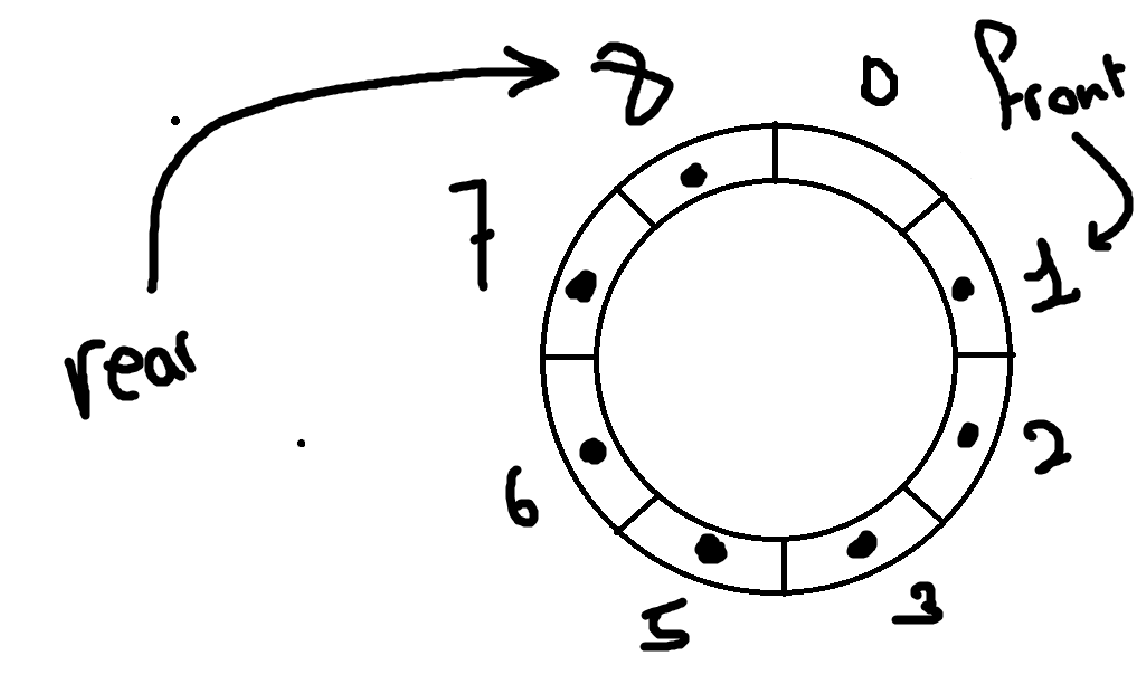
### Dequeue

Same concept for the dequeue

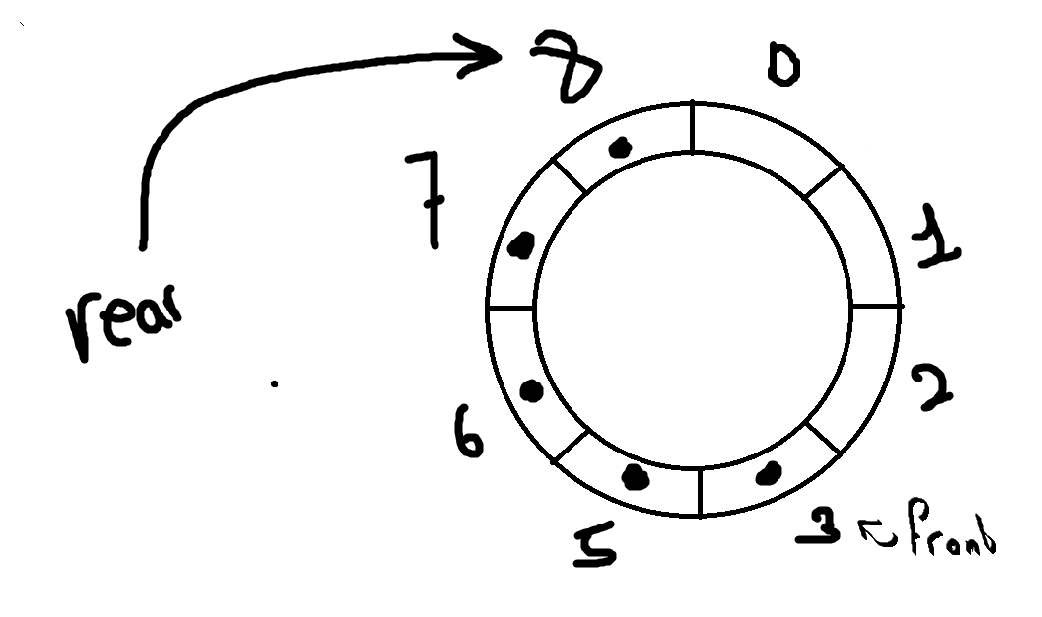
Now once we dequeue once, we are gonna use this equation to determine the index of the front after dequeuing

Dequeue once:

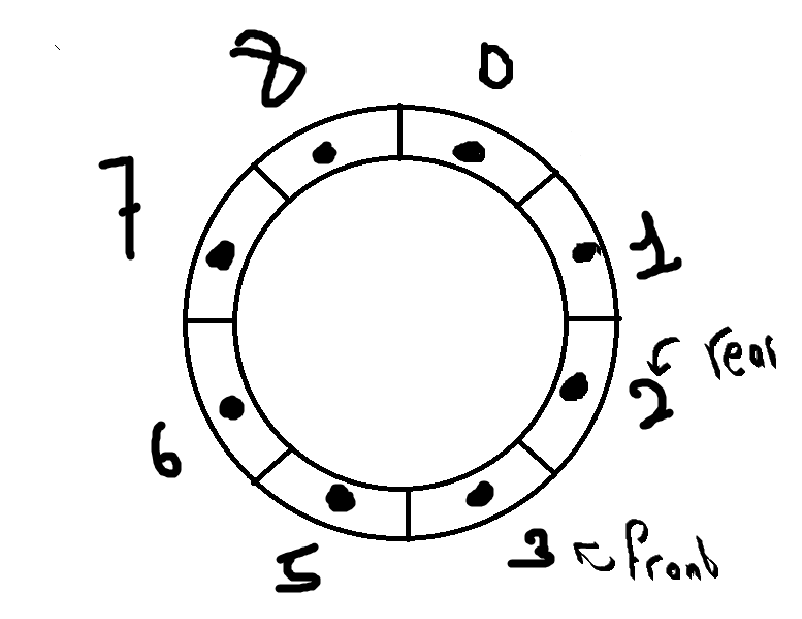




Lets dequeue twice



Enqueue thrice now

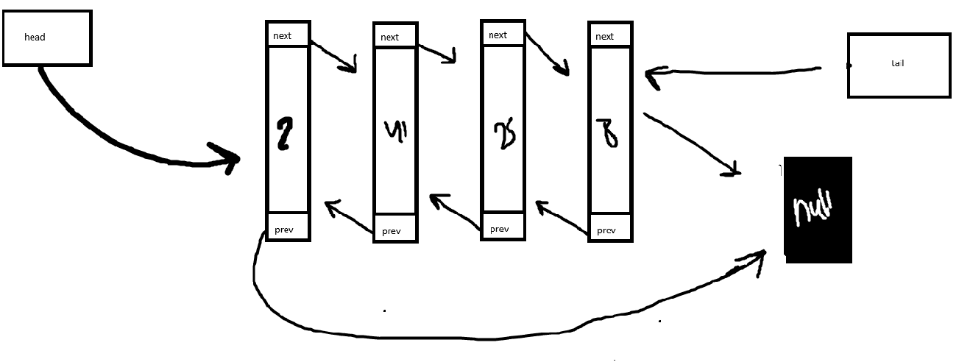


And voila, the front and the rear has completely changed, no longer is the rear’s index the larger number but the front’s index!

[Circular Queue Implementation - Array](https://www.youtube.com/watch?v=8sjFA-IX-Ww)

## Doubly Linked List

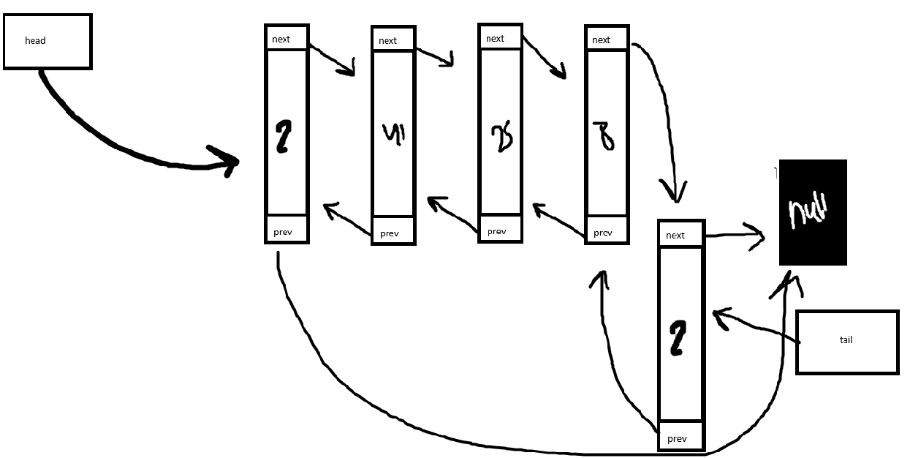
For the doubly linked list this is how its gonna look



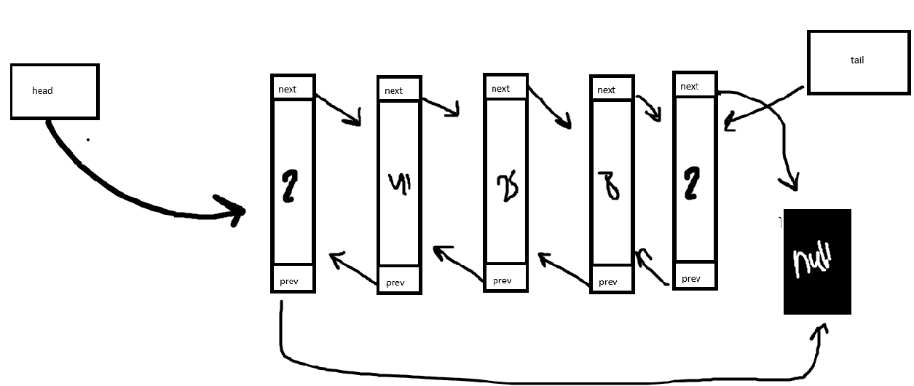
### Enqueue

Lets enqueue one element, as we know the queue works by adding elements to the last index not the first index like the stack

Here is out its gonna look like



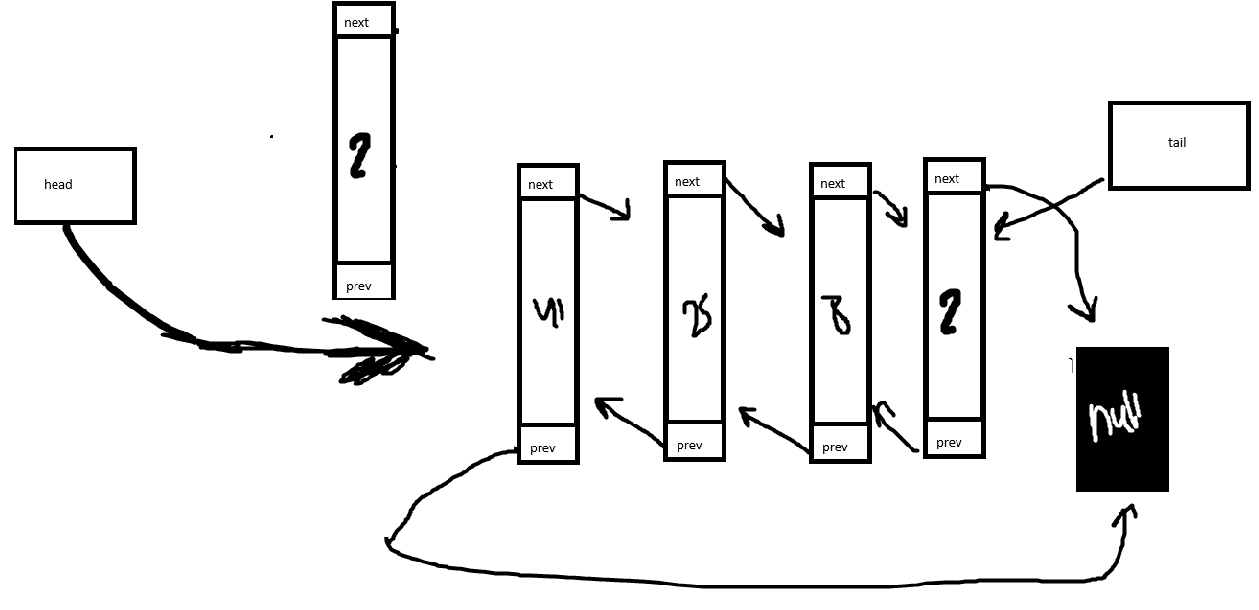
And as shown in the image above, we add the node in the last index, we link the next of the previous node to the new node. And the link prev to the previous node, and the next of the new node to the null and make sure that you save a reference to it by assigning it to the tail variable.



[Stacks, Queues, and Double Ended Queues (Deques) - YouTube](https://www.youtube.com/watch?v=IITnvmnfi_Y)

### Dequeue

And here is the dequeue



## Compare and Contrast

Now comparing them together

### The memory usage

in the circular queue is much better when looking at it from a conservative point of view as in you create a fixed size and then you will have the enqueue and dequeue loop around it. So one point to Circular Array. +1

In the doubly linked list, if there are no fixed size, then the system is forced to dynamically allocate memory to make sure that the data can be stored, no points to doubly linked list +0

### Scalability

In the circular queue, we must specify a size in order to get it to work, this means that we can’t scale it dynamically unless we ourself hard code it. +0

In the doubly linked list, we can add more nodes inside the queue without having any issue except for the memory, so one point to doubly linked list +1

### Implementation

Circular Queue, easy to implement and straight forward, doesn’t need that much oversight or future sight in order to implement it.+1 points

For the doubly linked list we need to make sure that our pointers are always being assigned correctly automatically by the code, +0 points, but it can support more complex methods like traversal in both direction + 1 points

### Applications

The best/perfect for scenarios where queue sizes are easily predicted like for example machine idling scheduling or something like that +1

Best for scenarios were queue size cant be predicted like for example customer support cases. +1

Lets count the points

4 for circular queue and 3 for doubly linked list, take it as you may.

# Question 6

## Hospital Management

Scenario: Ahmad arrives to hospital with a toenail infection (non-emergency case) The hospital Management system is going to handle the following

### Patient Registration Using the Linked List

The System uses a linked list ADT to register patients’ info like his name, age, id, and ailment. When Ahmad arrives, a new node is added to the linked list with his details. The system is going to check if needs emergency treatment, if yes, he is gonna be added to the emergency room, if not he is gonna be thrown to the waiting queue.

### Non-Emergency Treatment using a Queue

Since Ahmad is not an emergency case, he is thrown to the waiting queue which operates in a First in First Out basis. Ahmad is enqueued to the waiting queue at the back of line, there are many patients in front of him but he waits like a man. One by one all of the other patients got dequeued and treated until it was his turn, The doctor uses the peak() method to confirm that its Ahmad next turn, after confirming the doctor dequeues Ahmad, treats him, and Ahmad goes back home.

### Scenario 2: Ali arrives to the hospital with a heart failure (emergency case)

### Emergency Care

After the hospital management system registered the patient in, and it analyzed that Ali needs emergency care so they did **not** send him to the waiting queue. Instead, he is still at the Linked List, which is the emergency room and getting treated by medical officials. After treatment, Ali is the Discharged into the Discharged Stack

### Discharge Stack

When Ali is discharged after his treatment, his Patient ID is pushed into the stack ADT

If medical officials believe that Ali require more treatment, they can use the reAdmitPatient function in the linked list class to pop his ID back into the emergency room for further treatment. The stack operates on a Last In First Out basis meaning that if Ali is still the last guy to get discharged, he is going to be the first guy to get re-assessed if the medical officials decide to call the reAdmitPatients Function.

### Scenario 3: Emergency Fire Plan using Graph ADT and a Dijkstra Algorithm

### Hospital Ground Plan

In order to efficiently map out the hospital, we created a system that maps out the ground plan of the hospital really efficiently using the Graph ADT. This data structure has features like adding edges to connect any rooms together. I also created another function to immediately initiate the plan and shows the rooms, hallways, and exits as nodes and the paths are the edges.

### Dijkstra Algorithm

This system uses the Dijkstra algorithm paired with the Hospital Plan (graph ADT) in order to find the shortest distance to the exits from the source.

### Scenario 4: Employee management using a Binary Search Tree

This system which I like to call the Administration System uses a binary search tree in order to efficiently index employees that are working currently inside of the hospital, It works based on the Employees ID, it insert the first employee to start working inside of the tree, then the second employee its going to check if the employee’s ID is larger or smaller than the root id, if its larger, its gonna go to the right, if its smaller its going to go to the left, and its going to repeat that recursively until no more employees are registering in to work for the day. And in order to find and search for the employees we apply many methods such as inorder, preorder, and postorder methods, depending on your situation you can use anyone of them. For the inorder traversal it works like the following, it follows a left-root-right pattern

Meaning that the entire left tree is traversed to first, then it goes to the root, and traverses the right subtrees one by one until reaching the original root node and then goes to traverse the right subtree.

Preorder follows a Root-Left-Right traversal policy where the root node of the tree subtree is visited first, then the left subtree is traversed, and then the right subtree is traversed

Postorder follows the left-right-root traversal policy where that for each node the left subtree is traversed to first, then the right subtree is traversed to, then finally the root node of the subtree is traversed.

All of these work recursively and indefinitely until finding the employee that we are searching for.

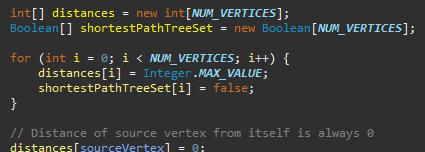
[Preorder vs Inorder vs Postorder - GeeksforGeeks](https://www.geeksforgeeks.org/preorder-vs-inorder-vs-postorder/)

# Question 7

## Dijkstra Algorithm

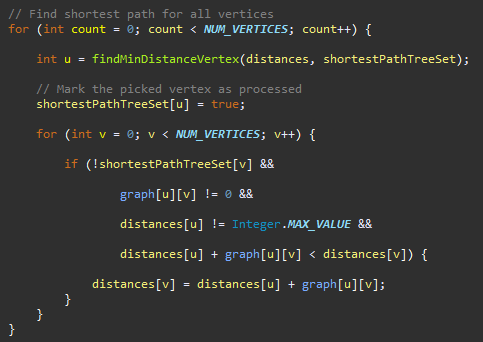
### Initialization

Set the distance of the source vertex as 0 and the rest of the vertices are infinite, it will then make sure that all of the vertices are marked unprocessed as in “false”



### Main Loop

It will select the vertex with the least distance that hasn’t been checked yet and then its going to update the distance of its adjacent vertices if a shorter path is found. Its going to mark the select vertex as checked or “true”



Its then going to repeat until every vertex is processed and is then going to print the shortest distance from the source

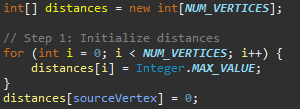
Then we are going to print the distances from source.



## Bellman Ford

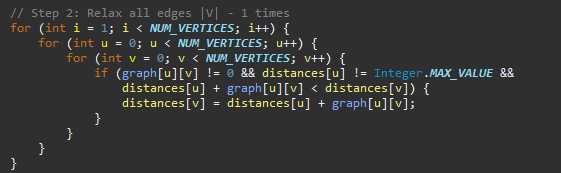
### Initialization

Its going to set the source vertex to 0 and every other vertex to infinity

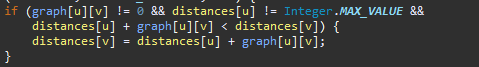


### Relaxation

Its then going to relax the edges meaning updating the shortest distance to a node if a shorter path is found through another node.



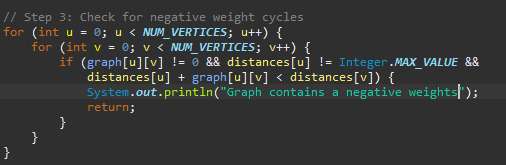
Its going to relax the distance for each edge being (u,v) and update the distance of v if a shorter path is found by this if statement



Meaning if a distance[u] + graph[u][v] is less than distance[v], then make the distance[v] = distance[u] + graph[u][v]

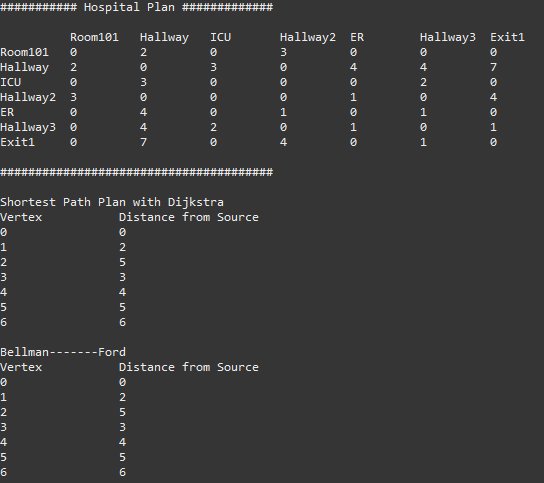
### Detection of Negative Cycles

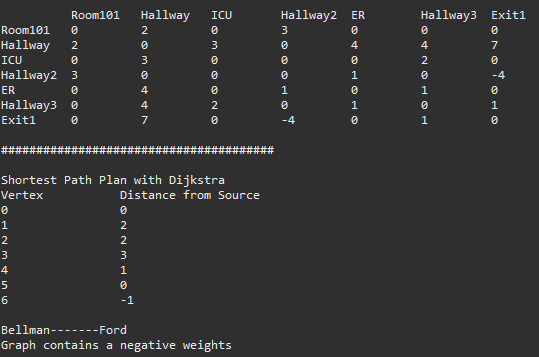
After the relaxation step, check for any negative weights. If any distances can get improved more, the graph has included negative weights.



### Then its going to print the output



 here is a graph without a negative cycle. Here is it again but with a negative cycle at hallway 2 connected to exit 1



As seen here, we have an error value for the Dijkstra algorithm but for the bellman ford it successfully recognized the negative cycle.

## Conclusion

In conclusion the Dijkstra Algorithm is good for graphs that do not incorporate negative weights

And the Bellman ford algorithm is flexible and can handle graphs with negative weights and can detect negative cycles

Only reason we should not deal with negative weights is because useless because you can repeatedly travel into the negative cycle and it will in-consequence shorten the path significantly and illogically meaning there is no shortest path as the distance decreases and decreases more.

[Bellman Ford Algorithm in Java - Sanfoundry](https://www.sanfoundry.com/java-program-implement-bellmanford-algorithm/?utm_source=chatgpt.com)

# Question 8

The hospital management system uses many algorithms for many different functions and operations including but not limited to searching for employees in a binary search tree, merge sorting or selection sorting, and shortest path algorithm using either Bellman ford or Dijkstra algorithm.

## Binary Search Tree for Employee management

The binary search tree enables us to efficiently search, insert and “delete” the currently active employees based on their IDs, this works best for random employees that begin their work day in random intervals, but when the employees are sorted and come in sorted intervals as in:

Employee with ID# 1 is the first employee of the day, then ID#2 is the second, etc. The Tree is going to behave like a linked list rather than a BST, which is the worst case scenario which will have a time complexity of O(N).

Even though the binary tree worst case time complexity is not that bad, its still highly inefficient to use a binary search tree if its going to act like a linked list. By implementing a self-balancing trees like AVL its going to mitigate for worst case scenario.

Binary Search Tree also has many different kinds of traversal methods, which are : Inorder, PostOrder, PreOrder. These methods are important to process the binary search tree, they are efficient but they do not improve the time complexity, but they are still important for operations like insertion and searching.

For searching, inserting, and deleting the time complexity is going to be

Best Case: O(1), the root is the employee we are searching for

Mid case: O(LogN) the employee is some where in the middle of the binary tree

Worst Case: O(n) employee is the furthest entry from the root.

[Complexity of different operations in Binary tree, Binary Search Tree and AVL tree - GeeksforGeeks](https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/)

[AVL Tree Data Structure - GeeksforGeeks](https://www.geeksforgeeks.org/introduction-to-avl-tree/)

[Time and Space complexity of Binary Search Tree (BST)](https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/)

## Sorting Algorithms

Merge Sort: its an important sorting algorithm that is really efficient on large datasets, making it one of the best sorting algorithms to use if we are going to have a huge database of patient records, its methodology of diving and conquering makes sure that the performance is consistent but after dividing, we have the merging step, this sorting algorithm requires additional space for merging, which means that its going to be a big step back for systems with memory issues or lets just say low memory.

Its time complexity is

Best Case: O(n log n)

Worst Case: O(n log n)

Selection Sort: Selection sort is a simple algorithm that linearly search for the smallest number in the array and swaps it with the first element pushing it back. Its really good for smaller datasets and arrays, but its really bad when trying to scale the system because its highly inefficient for larger datasets/arrays.

Its time complexity is

Best case: O(N^2)

Worst Case: O(N^2)

And this hear shows that its highly inefficient, imagine it trying to sort a dataset with 1000 entries, its going to repeat it 1000\*1000 times.

## Shortest Path Algorithm

### Dijkstra Algorithm

Dijkstra Algorithm is really efficient for finding shortest path in a weighted or unweighted graph(every edge is weighted 1). For a weighted hospital layout it makes sure that the shortest path to the exit is calculated really efficiently and taking every path into account to find the shortest one. But, it does not handle negative weights, which as stated earlier will create an illogical infinite distance reduction and this is bad if we want to apply it in a real life scenario, so to combat this we can add a check for negative weights or cycles like how the bellman ford algorithm operates, which is also a shortest path seeker algorithm.

# Question 9

For feature 6 I am going to discuss its trade off considering sorted array and BST ADT. When considering Sorted Arrays and BST both of them have their pros and cons

## Time complexity:

|  |  |  |
| --- | --- | --- |
|  | Best Case | Worst Case |
| Sorted Array | Insertion: 0(1)  Searching  O(1)  Deletion  O(1) | Insertion: 0(n)  Searching  O(Log n)  ^ binary search  Deletion  O(n) |
| BST | Insertion: O(log n)  Searching  O(log n)  Deletion  O(log n) | Insertion: 0(n)  Searching  O(n)  Deletion  O(n) |

## Sorted Arrays

Sorted Arrays Evaluation: Sorted Arrays are really efficient for the best-case scenarios, but for worst case scenarios they quickly degenerate into inefficiency in all of its operations, Insertion and deletion is going to be slow especially as the array grows. They suffer from poor performance on the entire board for inserting and deletion, but they are space efficient and, if your dataset is small, or you are using a really good searching algorithm such as binary search. But the need to shift elements whenever inserting or deleting is highly inefficient.

## Binary Search Tree

When the tree is balanced, every operation is efficient but when the tree is unbalanced, performance degenerates and degrades significantly. BST allows for highly efficient data management with really efficient insertion and deletions; however, you must maintain balance, or else your worst case could be that the binary search tree becomes a linked list. It must maintain balance to avoid performance issues. The solution to this could be implementing self-balancing trees, but will add to the complexity of the implementation.

## Conclusion

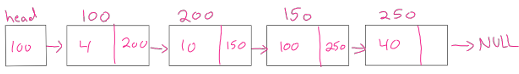
In conclusion, the choice between choosing a sorted Array and a binary search tree for managing employees in the hospital management system highly depends on the requirements of the application, forget about the assignment brief I am talking about a real-world scenario. If we need to have so many insertions and deletions, we are going to look at the BST more favorably. But if the dataset is going to be small and generally static and requires really frequent searches, we are going to look at the sorted arrays paired with a good sorting algorithm like binary search. So in real world scenario, don’t limit yourself to what you think is best, you must critically evaluate your uses and find the best ADT for it.

PART 2

# Question 1

We are going to break down the execution of the print function step by step. As seen in this code provided the function iterates recursively.

The print function takes in a parameter of type Node, if the node is null the function returns immediately. Else its going to recursively call itself with the next node in the list. (node.next) and the prints the data of the current node.



## First Call

Head = Node 100

Function calls itself with print(node.next) = 200

Current step is (data 100) is being pushed onto stack

## Second Call

Print is called node = 200, function calls itself against with print(node.next) = 150

The current step (data 200) being printed pushed onto stack

## Third Call

Node = 250, function calls itself again with print(node.next) = 250

Current step is (data 150) printed pushed onto stack

## Fourth call

Function calls itself again with print(node.next) = null

Current step is (data 250) being printed pushed onto stack

## Fifth call

Function checks if node != null, which is true, its going to return immediately. Base case is met so the function stops.

Output =

250 150 200 100

# Question 2

Asymptotic analysis is critical to assess the effectiveness by allowing us to evaluate performance of algorithms in relation to the input size, which is important for scalability.

## Example 1

You have two algorithms, Algorithm A is going to be a linear search, whilst Algorithm B is going to be a selection Sort.

* Algorithm A: It has a linear time complexity as in O(n); say you have an array size of 10, if you want to run it through this algorithm, at worst case scenario you are going to end up with 10 operations
* Algorithm B: it has a quadratic time complexity as in O(n\*n); say you have an array of size 10, if you want to run it through this algorithm, at worst case scenario you are going to end up with 10\*10 operations.

The difference is absolutely staggering and is going to lead someone to believe that the Algorithm B is highly inefficient, which leads us to Asymptotic analysis, because it helps with understanding how an algorithm behaves as input sizes scales.

Without considering time complexity, the input size can impact the performance significantly. When analyzing performance , it is important to recognize that input size can impact the efficiency of the algorithm beyond what is captured through big O notation. For example you might have factors like input size, implementation, machine performance, etc.

For example you have a merge sort algorithm, as seen in Part 1 question 4, it has a time complexity of O(n log n) which is really good, but when I tried to test it with a small test case it was worse than selection sort (O(n\*n), which excelled with the smaller array, but when you increase the size, the selection sort is going to increase quadratically/exponentially whilst the merge sort is going increase logarithmically, which is more effective and efficient.

In practice its crucial to examine many test cases, and test input sizes because any algorithm can be affected by many different factors. Asymptotic analysis is also good in preparing the devs with how their system is gonna be used which will hopefully lead to a better performance and optimization but also, Asymptotic Analysis is not always perfect but it’s the best way to analyze algorithms.

[Asymptotic Analysis - GeeksforGeeks](https://www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/)

# Question 3

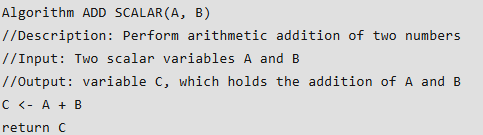
You have two types of measuring algorithm efficiency

* Time Complexity
* Space Complexity

Time Complexity

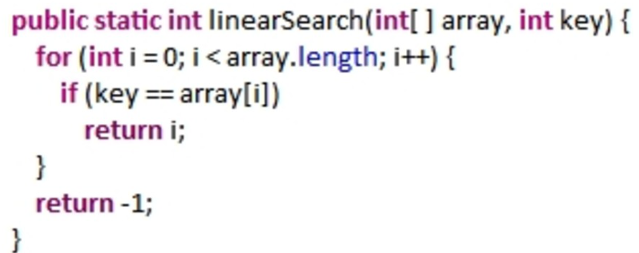
Time complexity measures the amount of time an algorithm takes to complete as a function of the length of the input, not the actual execution time of the machine, by expressing time complexity as Big O notation.

Common Time Complexities

* **O(1)**: Constant time: Simple arithmetics like a + b = c, no increase
* **O(log n)**: Logarithmic time: increases slowly with the input size
* **O(n)**: Linear time, increases with the input size directly, occur in loops.
* **O(n^2)**: Quadratic time: increases exponentially as input size increase, occurs in nested loops

Example:

Linear Search



Here is a linear search, worst case scenario here is if the target is in the back of the array which results in O(n) where the number n is the number of elements in the array.

Space Complexity

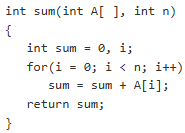
Space complexity measures the amount of space/memory an algorithm uses relative to the input size. Including but not limited to the space needed to hold the input and any more space needed.

Common Space Complexities

* **O(1):** Constant space, uses a constant space regardless of the input size
* **O(n):** Linear Space: space use grows directly with input size growth.
* **O(n\*n)**: Quadratic Space: space use exponential grow as input size grows.

Example:

Linear Summation



Here in the linear summation algorithm, space needed is going to directly grow as the for loop runs., so the space complexity is O(n)

In conclusion to measure algorithm efficiency you have many ways, but the two most known ways are the Space and Time complexities, in which the space complexity focuses on analyzing how the memory/space is used as input size increase. Time complexity is used to analyze how the execution time grows as input size grows.

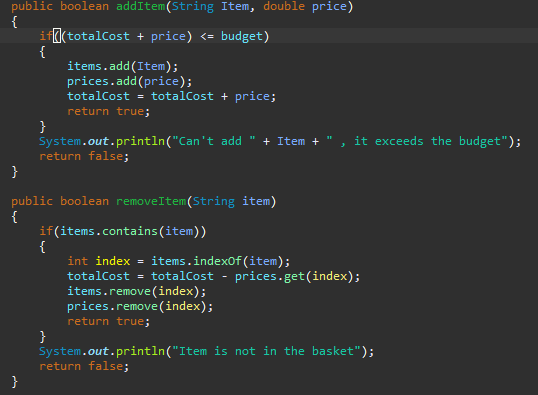
# Question 4

What is encapsulation? Encapsulation is the process of collecting variables and methods into a single unit which is typically a class.

Why is it important? Because it allows for data hiding which means that it does not allow everybody to access the internal variables outside of the class. They are only accessed by public methods which adds a layer of security and makes it easy to add or update the functionality of the code without affecting other parts of the code, like the client-side code.



This way it protects the integrity of the basketClass by only allowing manipulation to pass through addItem and removeItem. And not the variables themselves (items, prices).



# Question 5

Cores of Object Oriented Programming

* Encapsulation
  + This is the bundling of variables and methods in a single unit commonly called a class, and also involves adding a layer of restriction which won’t allow users the direct access to one’s object’s components. This adds security and prevents any unintended manipulation.
* Abstraction
  + Hiding the complex implementation and showing only the necessary features is what abstraction is all about. It also reduces the complexity of the class by modelling it to the appropriate thing you want to do.
* Inheritance
  + Inheritance is where a child class extends the parent class and thus inherits all the methods and variables from the parent class.
* Polymorphism
  + This allows the object to be seen as an instance of the parent class rather than their own.

Encapsulation is an important part of developing an ADT, it allows the ADT to encapsulate the variables and methods into a single class which hides the way data is handled from the user and only provides access through a well defined functions and interfaces which adds a layer of restriction and security that wont allow the user to directly manipulate the data. For example a stack ADT will encapsulate a linked list but will only show methods like push and pop this is to ensure that the internal handling of the stack ADT cannot be directly manipulated by users and thus maintaining its integrity.

For the abstraction, its important to know what operation is going to be handled instead of how to implement those operation for example the Stack operates in a LIFO methodology which means last in first out, so we need to make methods like push, pop. The user doesn’t need to know if the stack is operating on a linked list or an array list, he just need to be able to use it.

Inheritance allows the ADTs to extend a parent class, this parent class can define how the ADT is going to behave, and any child adt can add or remove functions of its own. For Example now that I think about it I should have done a base linked list class and extended the hospital management, discharge stack and waiting queue from it. This would have shortened the amount of time I took to create the linked list for every class.

For polymorphism they will allow the ADTs to be used together if they share a common interface this is excellent because it will be so useful when designing systems that work with multiple implementations of the same ADT.

These OOP principles help make the ADTs better by making them Modular by making them contained which makes it easier to design and test, and also make them reusable as we could make one base ADT and derive many adts from it and will also increase the security by encapsulation which allows the user to be able to interact with only what the dev wants him to interact with.

# Question 6

Why queues and stacks are used over basic arrays

Using queues and stacks instead of just using arrays or any other basic data structure offer us more pros than cons for example:

* Simplicity: Queues and stacks operate on easy to read and understand methodologies (LIFO for the stacks and FIFO for the queue) this makes it so that the code is easier to understand and maintain, and this also makes it so that the users can use these structures without knowing how they are implemented in the current solution.
* Efficiency: Queues and stacks are designed to offer you the most efficient functions and methods for their specific cases be it enqueue/dequeue for the queue adt, push/pop for the stack adt. This decreases the need for brainstorming techniques like shifting arrays or reducing size or adding last that are the direct root cause to a lot of my headaches.
* Security: by encapsulating the variables and methods in these classes, we make sure that the user can only manipulate the data the way we want him to, which reduces security risks and the risk of misuse. For example, the waiting queue doctor needs to only access the patient next in line, why provide him with more leniency by showing him every patient in the queue if we can just show him the next patient in line and give him the ability to pop him to be operated on.

Difference between queues and stacks

* Queue operates in first in first out methodology (FIFO) which means that the first patient to arrive in the waiting queue is the first patient to be operated on. I would use it specifically when I need to store waiting patients from the order the register in and operate on them one by one, ensuring fairness with not allowing patients that came last to be operated on before the already waiting patients.
* Stacks operate on a last in first out methodology (LIFO) which is important in my use case which is temporarily manage discharged patients. Its effective because I want to prioritize the most recent discharged patient over the ones that were already discharged in the past and never re-admissioned again.

Maintability and Scalability with Independent data structures

* Modularity
  + By separating the handling of vastly different logics like the queues and stacks which are the polar opposite of each other, we can easily modify them to our needs without messing up the whole system. For example if the method for handling queues is changed, only that data structure’s implementations needs to be updated, not one giant blop of a data structure.
* Simplicity
  + Independent data structures can be tested individually which will lead to a better understanding of the system, how it works, and a better reliability in the overall system. Test cases can be focused on the specific functions inside this specific class without affecting other classes.
* Scalability
  + As the number of patients increase, so does the need for an efficient management system. Queues and stacks can both be scaled separately in accordance to the demand. Which will allow me, the developer, to be able to handle a large amount of data very efficiently and not get poinked in the head by a giant spaghetti code.

Writing Tests Cases for Queue Implementation:

To make sure that my queue is in tip top conditions for my medical system I am going to make sure that I adhere to the following criteria

* Basic Tests:
  + I am going to enqueue first to make sure that the item is added correctly to the back of the queue
  + I am going to dequeue to make sure that the item in front is removed from the queue
* Extreme cases tests
  + I am going to empty the queue, then I am going to try to dequeue and observe the output to make sure that the behavior is consistent.
  + I am then going to do a series of enqueues and dequeues in random behavior and then view the data to make sure that its in correct order.
  + I am then going to fill the queue, and then enqueue once more to make sure that the current implementation correctly handles my attempt of overfilling it.
* Stress Tests
  + I am going to enqueue a huge number of times, and dequeue a huge number of times, and document the performance of each iteration.

# References

**Works Cited**

Algorithms with Attitude. “Stacks, Queues, and Double Ended Queues (Deques).” *YouTube*, 16 Jan. 2016, www.youtube.com/watch?v=IITnvmnfi\_Y. Accessed 27 Jan. 2025.

“Bellman Ford Algorithm in Java - Sanfoundry.” *Sanfoundry*, 25 July 2013, www.sanfoundry.com/java-program-implement-bellmanford-algorithm/?utm\_source=chatgpt.com. Accessed 27 Jan. 2025.

Blue Tree Code. “Circular Queue Implementation - Array.” *YouTube*, 12 June 2019, www.youtube.com/watch?v=8sjFA-IX-Ww. Accessed 27 Jan. 2025.

GeeksForGeeks. “Asymptotic Notation and Analysis (Based on Input Size) in Complexity Analysis of Algorithms.” *GeeksforGeeks*, 3 Jan. 2012, www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/.

---. “Complexity of Different Operations in Binary Tree, Binary Search Tree and AVL Tree - GeeksforGeeks.” *GeeksforGeeks*, 19 Jan. 2018, www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/.

GeeksforGeeks. “Difference between Stack and Queue Data Structures.” *GeeksforGeeks*, 23 Jan. 2019, www.geeksforgeeks.org/difference-between-stack-and-queue-data-structures/.

---. “Merge Sort - GeeksforGeeks.” *GeeksforGeeks*, 31 Oct. 2018, www.geeksforgeeks.org/merge-sort/.

---. “Preorder vs Inorder vs Postorder.” *GeeksforGeeks*, GeeksforGeeks, 17 Apr. 2024, www.geeksforgeeks.org/preorder-vs-inorder-vs-postorder/.

geeksforgeeks. “Introduction of Object Oriented Programming.” *GeeksforGeeks*, 5 Sept. 2020, www.geeksforgeeks.org/introduction-of-object-oriented-programming/.

“Introduction to AVL Tree.” *GeeksforGeeks*, 2 Jan. 2023, www.geeksforgeeks.org/introduction-to-avl-tree/.

“Time and Space Complexity of Binary Search Tree (BST).” *OpenGenus IQ: Computing Expertise & Legacy*, 29 Dec. 2021, iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/.