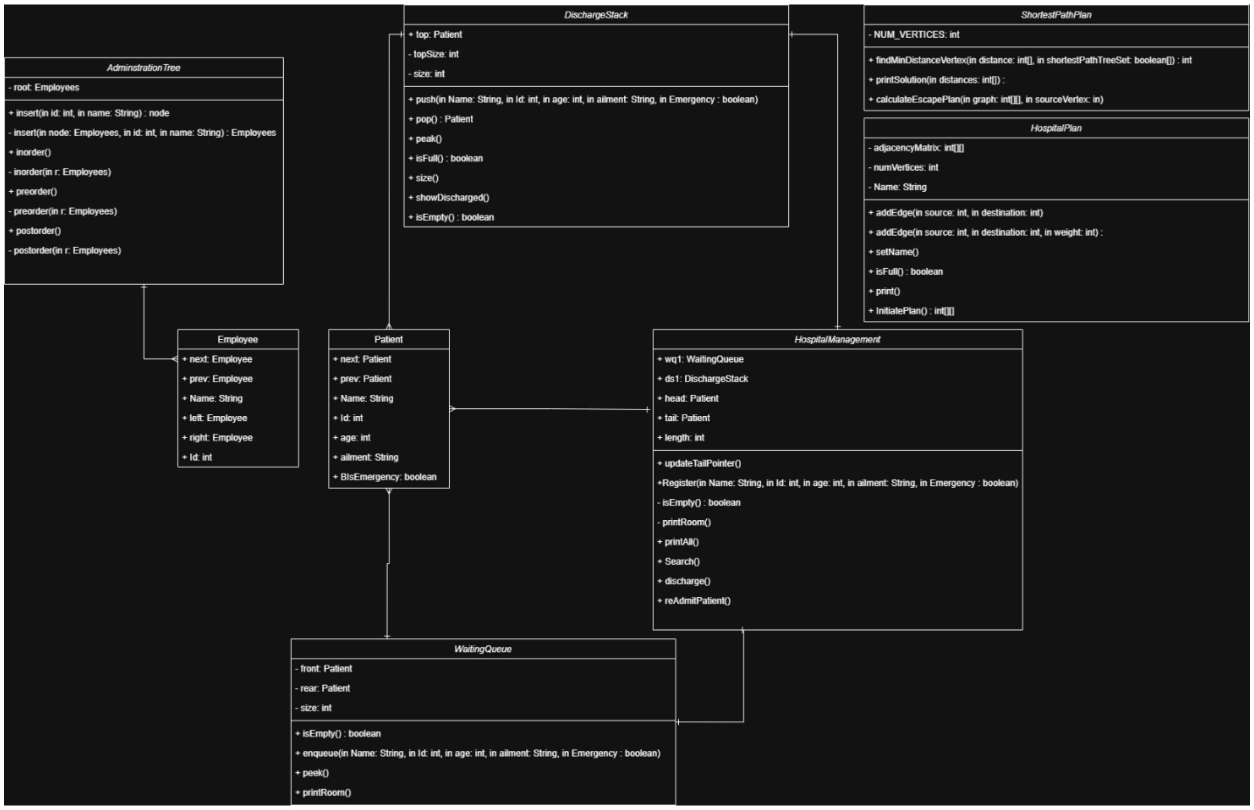
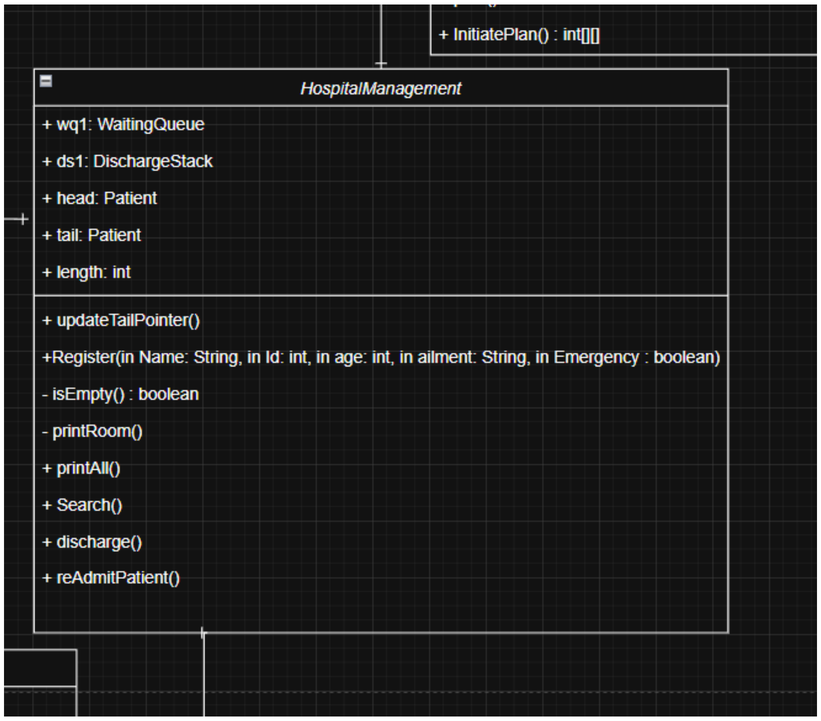
## Data Structure



## 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.

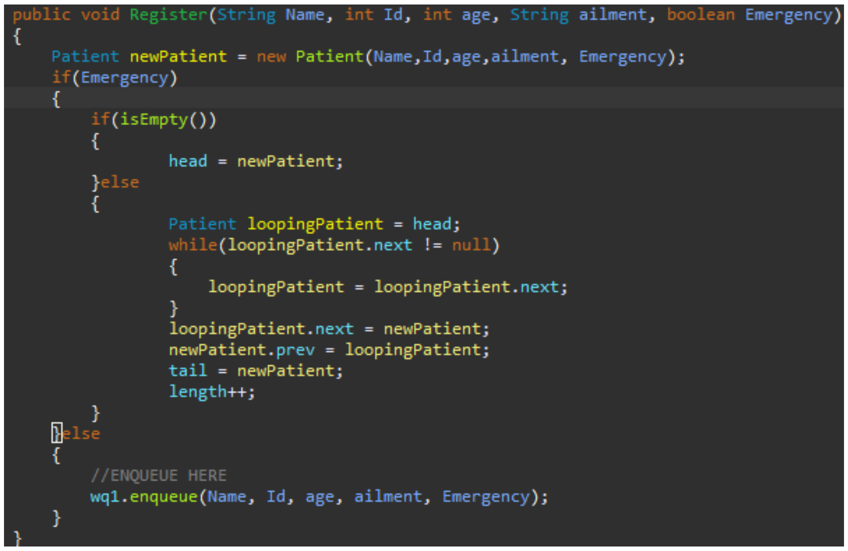
In simpler terms, if the patients require emergency care they are prioritized, if not, they are going to wait.

### 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.



```
public void Register(String Name, int Id, int age, String ailment, boolean Emergency)
{
    Patient newPatient = new Patient(Name,Id,age,ailment, Emergency);
    if(Emergency)
    {
        if(isEmpty())
        {
                head = newPatient;
        }else
        {
                Patient loopingPatient = head;
                while(loopingPatient.next != null)
                {
                    loopingPatient = loopingPatient.next;
                }
                loopingPatient.next = newPatient;
                newPatient.prev = loopingPatient;
                tail = newPatient;
                length++;
        }
    }else
    {
        //ENQUEUE HERE
        wq1.enqueue(Name, Id, age, ailment, Emergency);
    }
}
```

## Discharge(Patient Id)

Parameters =

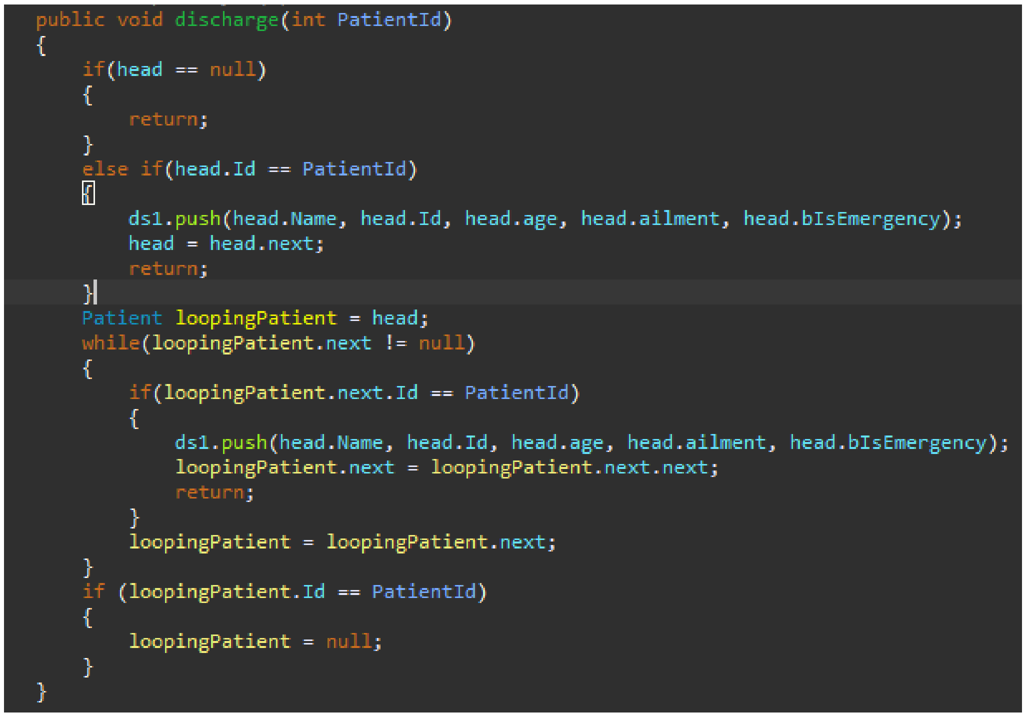
- PatientId

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



```
public void discharge(int PatientId)
{
    if(head == null)
    {
        return;
    }
    else if(head.Id == PatientId)
    {
        ds1.push(head.Name, head.Id, head.age, head.ailment, head.bIsEmergency);
        head = head.next;
        return;
    }
    Patient loopingPatient = head;
    while(loopingPatient.next != null)
    {
        if(loopingPatient.next.Id == PatientId)
        {
            ds1.push(head.Name, head.Id, head.age, head.ailment, head.bIsEmergency);
            loopingPatient.next = loopingPatient.next.next;
            return;
        }
        loopingPatient = loopingPatient.next;
    }
    if (loopingPatient.Id == PatientId)
    {
        loopingPatient = null;
    }
}
```

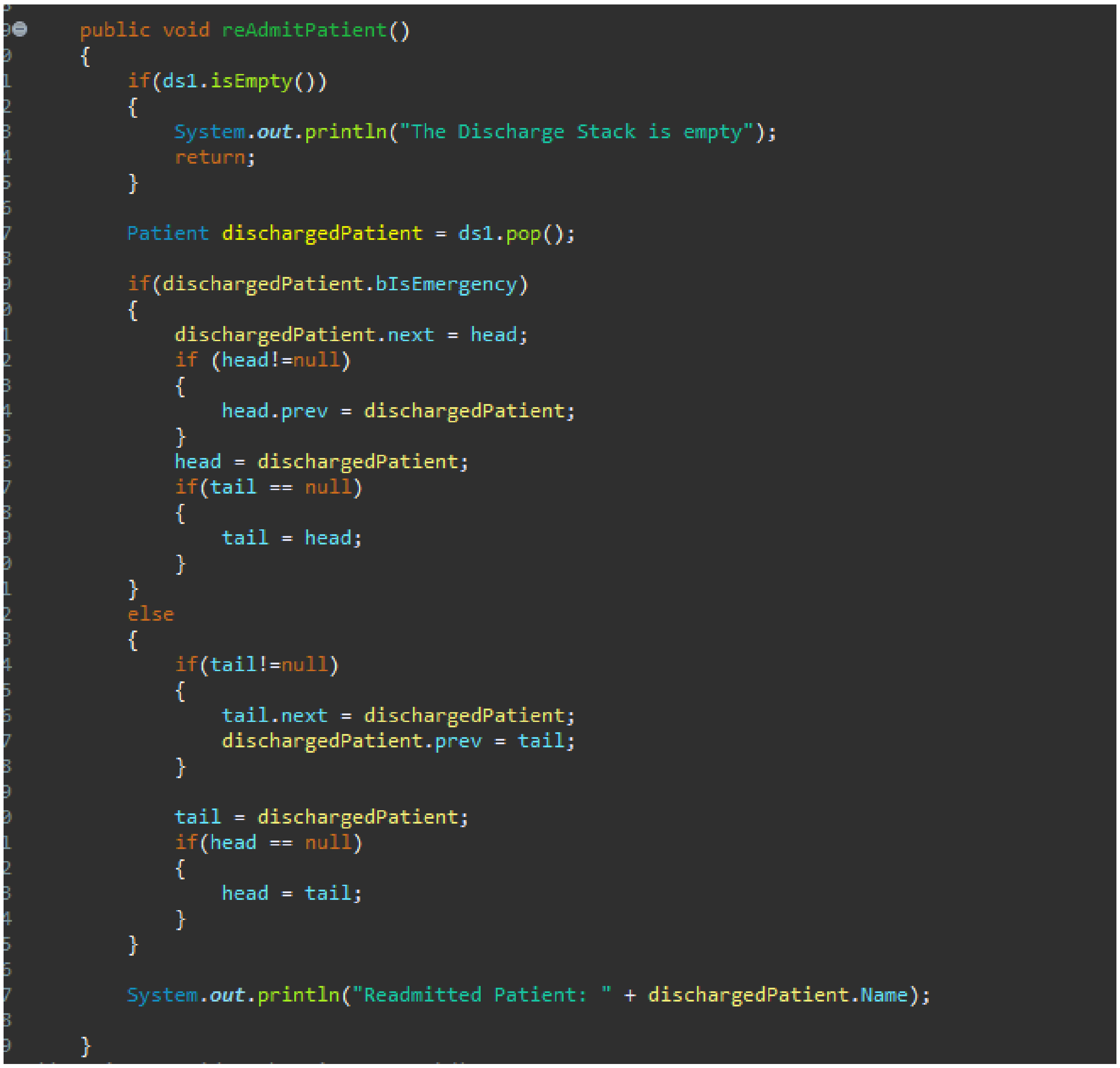
## reAdmitPatient

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.



```
    public void reAdmitPatient()
    {
        if(ds1.isEmpty())
        {
            System.out.println("The Discharge Stack is empty");
            return;
        }

        Patient dischargedPatient = ds1.pop();

        if(dischargedPatient.bIsEmergency)
        {
            dischargedPatient.next = head;
            if (head!=null)
            {
                head.prev = dischargedPatient;
            }
            head = dischargedPatient;
            if(tail == null)
            {
                tail = head;
            }
        }
        else
        {
            if(tail!=null)
            {
                tail.next = dischargedPatient;
                dischargedPatient.prev = tail;
            }

            tail = dischargedPatient;
            if(head == null)
            {
                head = tail;
            }
        }

        System.out.println("Readmitted Patient: " + dischargedPatient.Name);

    }
```

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:

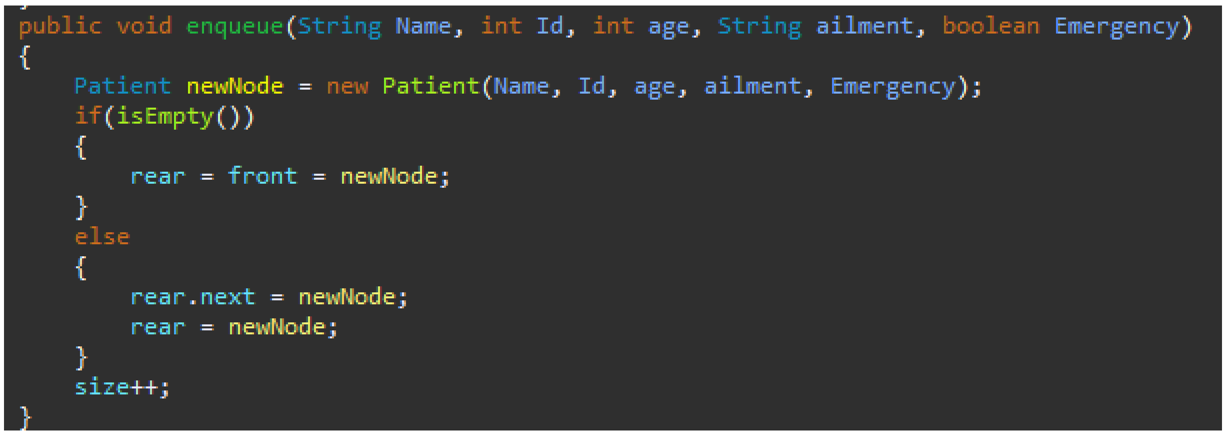
- Enqueue
- Peek
- Dequeue
- isEmpty
- printRoom

## 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.



```
public void enqueue(String Name, int Id, int age, String ailment, boolean Emergency)
{
    Patient newNode = new Patient(Name, Id, age, ailment, Emergency);
    if(isEmpty())
    {
        rear = front = newNode;
    }
    else
    {
        rear.next = newNode;
        rear = newNode;
    }
    size++;
}
```

## Dequeue

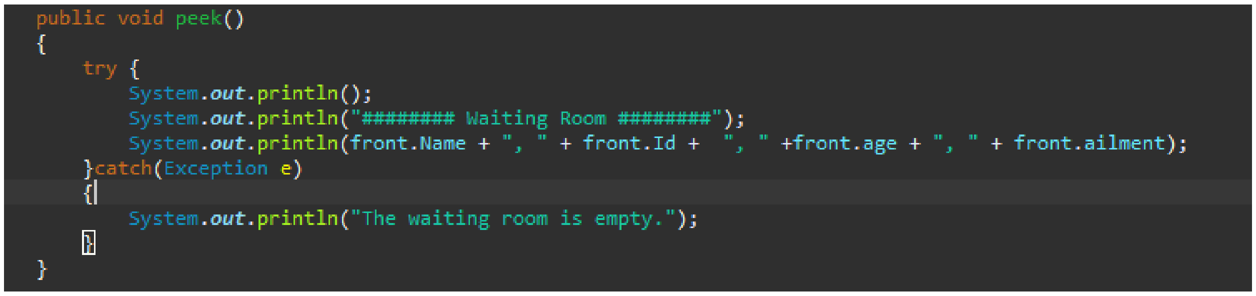
This method removes the next patient from the queue

### 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

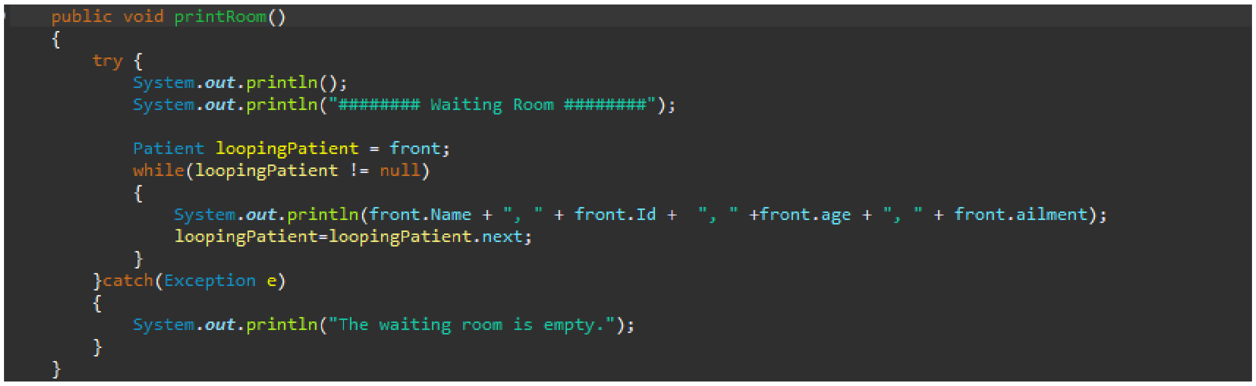
This method allows the system to see the front patient without removing them



```java
public void peek()
{
    try {
        System.out.println();
        System.out.println("######## Waiting Room ########");
        System.out.println(front.Name + ", " + front.Id +  ", " +front.age + ", " + front.ailment);
    }catch(Exception e)
    {
        System.out.println("The waiting room is empty.");
    }
}
```

## PrintRoom

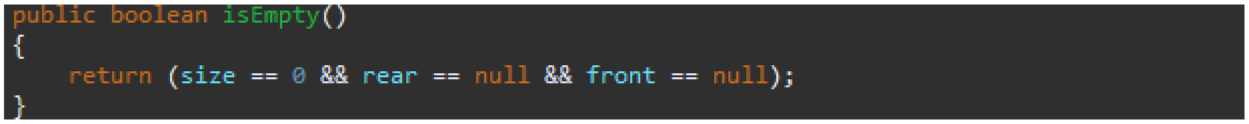
This method prints the all the patients in the queue by implementing a looping pointer that loops through all the patients inside the queue



```java
public void printRoom()
{
    try {
        System.out.println();
        System.out.println("######## Waiting Room ########");

        Patient loopingPatient = front;
        while(loopingPatient != null)
        {
            System.out.println(front.Name + ", " + front.Id +  ", " +front.age + ", " + front.ailment);
            loopingPatient=loopingPatient.next;
        }
    }catch(Exception e)
    {
        System.out.println("The waiting room is empty.");
    }
}
```

## isEmpty

this method checks if there are any patients inside the queue



```java
public boolean isEmpty()
{
    return (size == 0 && rear == null && front == null);
}
```

## 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:

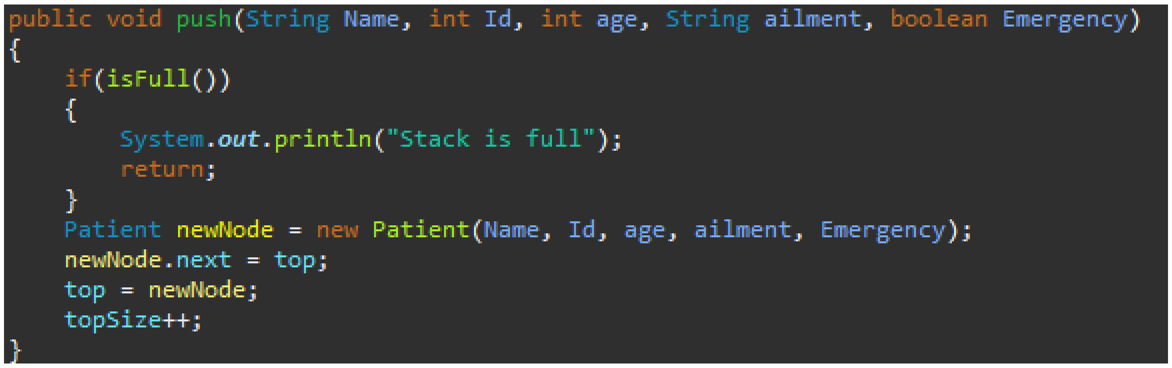
- push

- 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

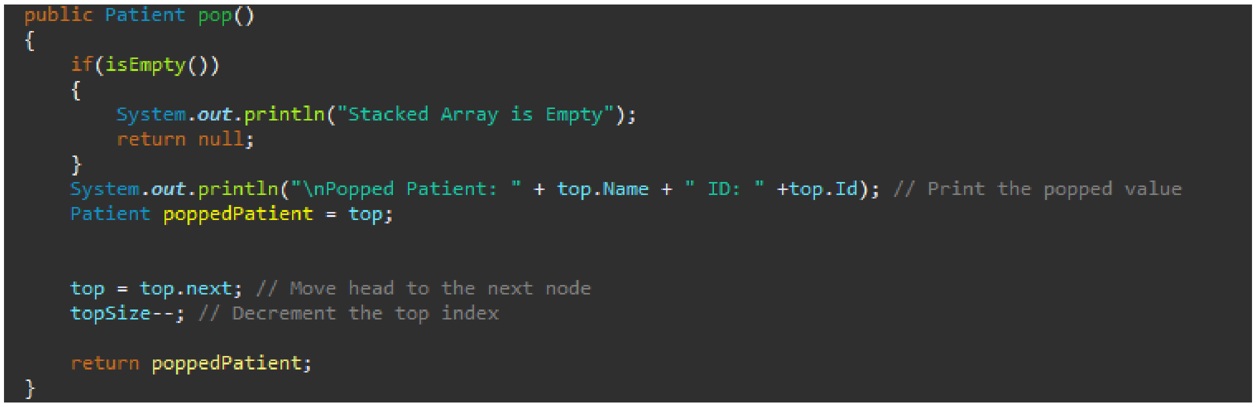


```java
public void push(String Name, int Id, int age, String ailment, boolean Emergency)
{
    if(isFull())
    {
        System.out.println("Stack is full");
        return;
    }
    Patient newNode = new Patient(Name, Id, age, ailment, Emergency);
    newNode.next = top;
    top = newNode;
    topSize++;
}
```

## Pop

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

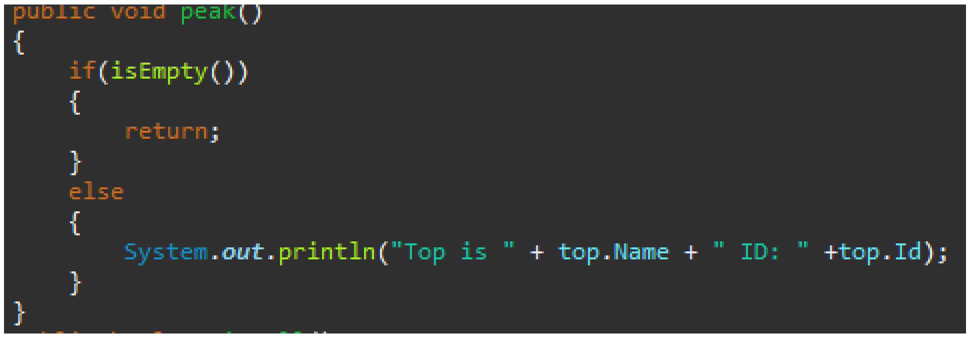
It checks if the stack is empty, if not then it gets the top of the sack patient and then moves the top to the next patient and reduces the stack size and then returns the previous top patient



```java
public Patient pop()
{
    if(isEmpty())
    {
        System.out.println("Stacked Array is Empty");
        return null;
    }
    System.out.println("\nPopped Patient: " + top.Name + " ID: " +top.Id); // Print the popped value
    Patient poppedPatient = top;


    top = top.next; // Move head to the next node
    topSize--; // Decrement the top index

    return poppedPatient;
}
```

## Peak

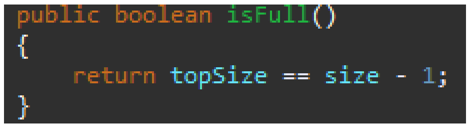
This method allows the system to see the top patient without removing them



```
public void peak()
{
    if(isEmpty())
    {
        return;
    }
    else
    {
        System.out.println("Top is " + top.Name + " ID: " +top.Id);
    }
}
```

## isFull

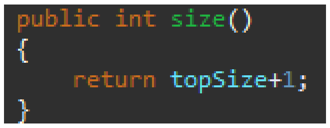
it checks if the size reached the top maximum size



```
public boolean isFull()
{
    return topSize == size - 1;
}
```

## Size

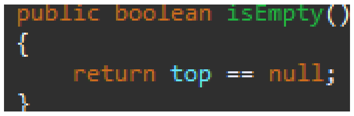
Gets the current amount of discharged patients



```
public int size()
{
    return topSize+1;
}
```

## isEmpty

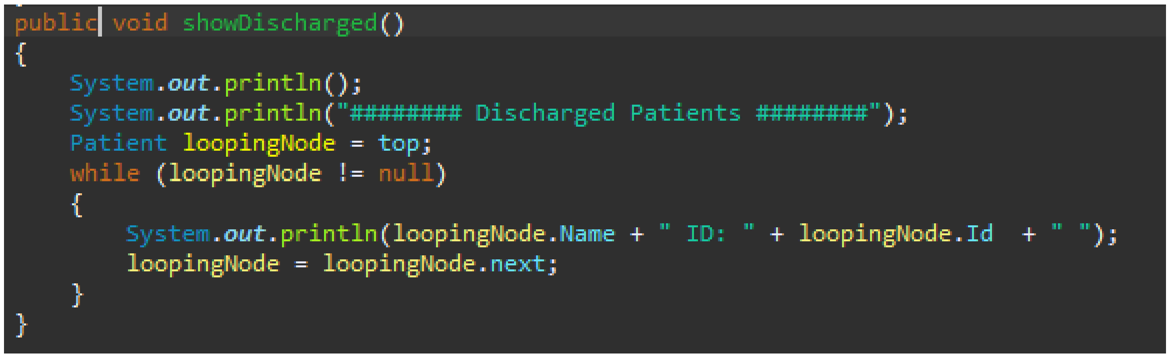
checks if the stack is empty



```
public boolean isEmpty()
{
    return top == null;
}
```

## showDischarged

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



```
public void showDischarged()
{
    System.out.println();
    System.out.println("######## Discharged Patients ########");
    Patient loopingNode = top;
    while (loopingNode != null)
    {
        System.out.println(loopingNode.Name + " ID: " + loopingNode.Id  + " ");
        loopingNode = loopingNode.next;
    }
}
```

## Administration Tree

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

Variables

- root: Pointer to the employees class

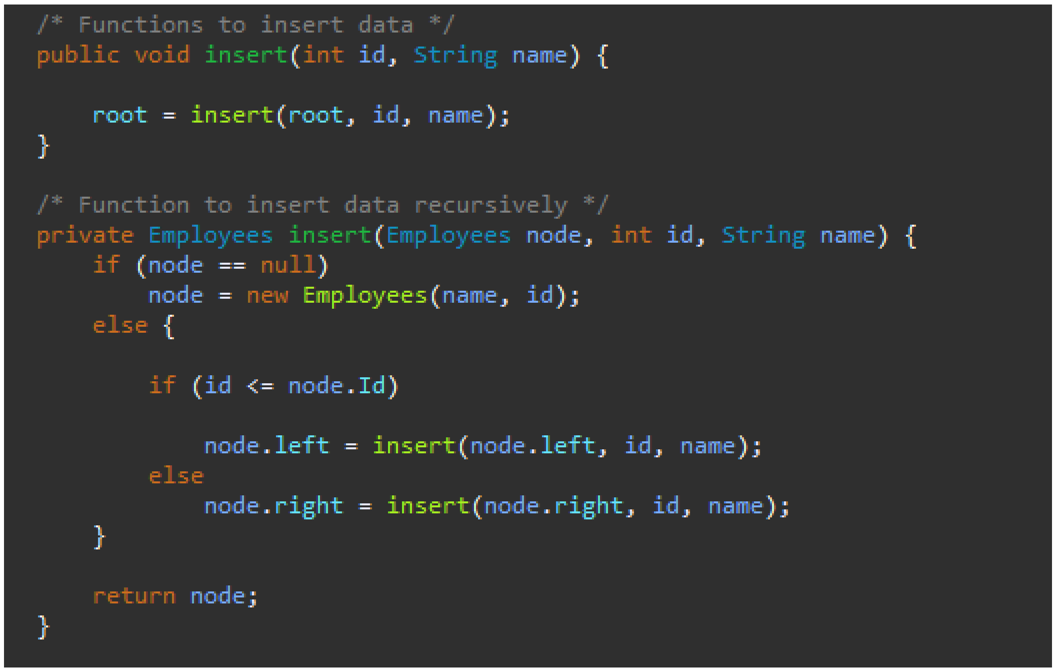
Methods

- Insert
- Inorder
- Preorder
- Preorder
- 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.



```java
/* Functions to insert data */
public void insert(int id, String name) {

    root = insert(root, id, name);
}

/* Function to insert data recursively */
private Employees insert(Employees node, int id, String name) {
    if (node == null)
        node = new Employees(name, id);
    else {

        if (id <= node.Id)

            node.left = insert(node.left, id, name);
        else
            node.right = insert(node.right, id, name);
    }

    return node;
}
```

### InOrder

## PreOrder

## PostOrder

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

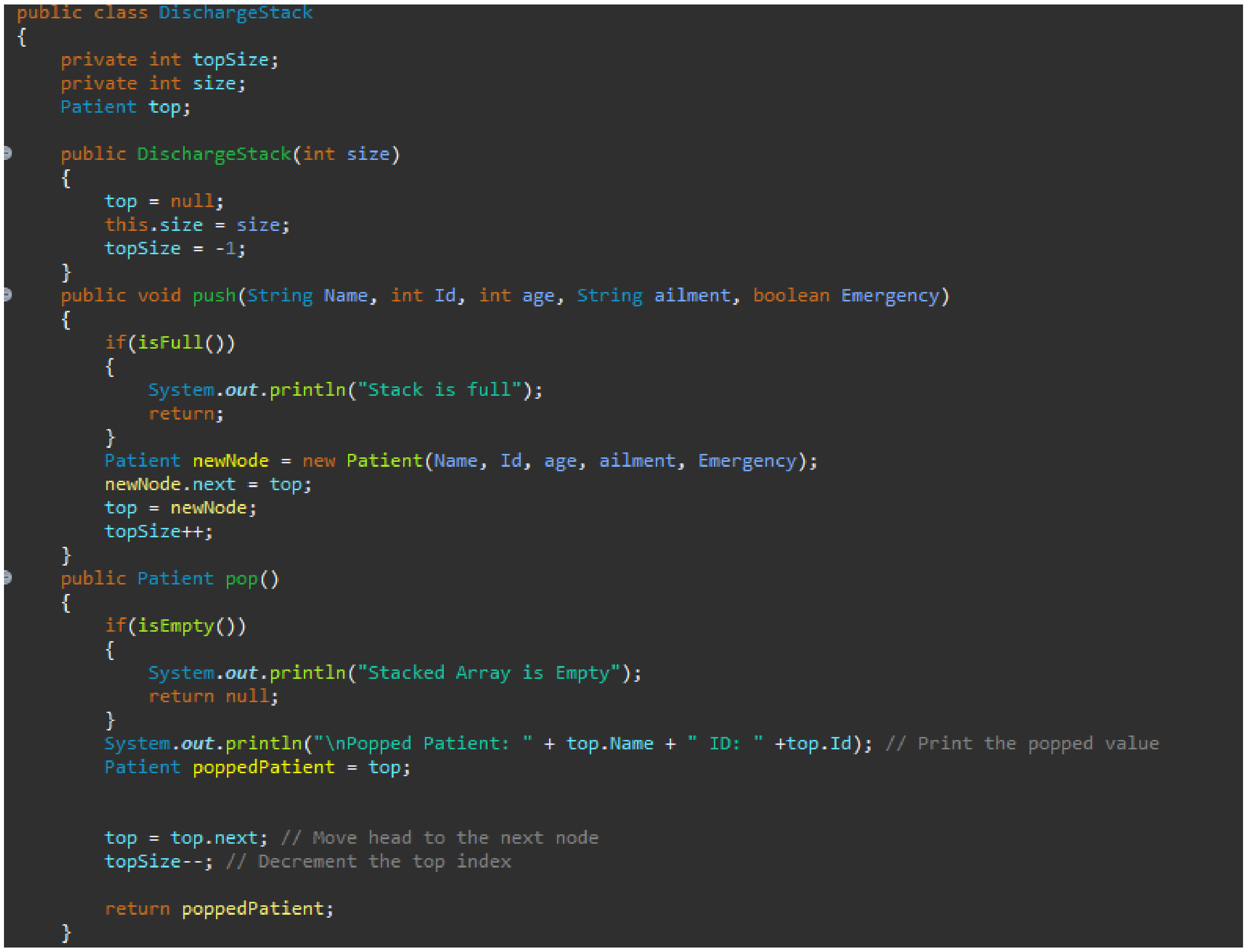
## isEmpty

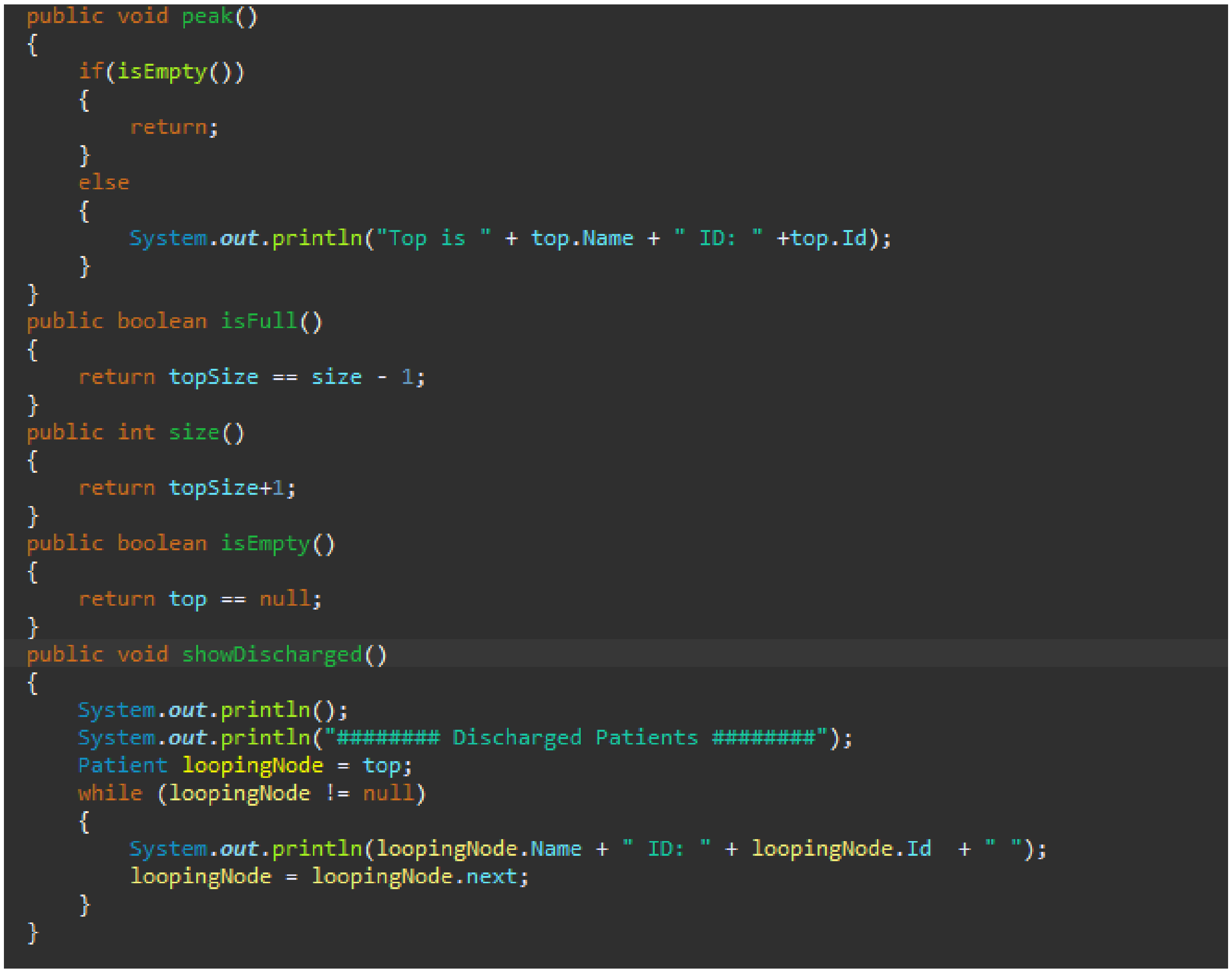
checks if the stack is empty

## showDischarged

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

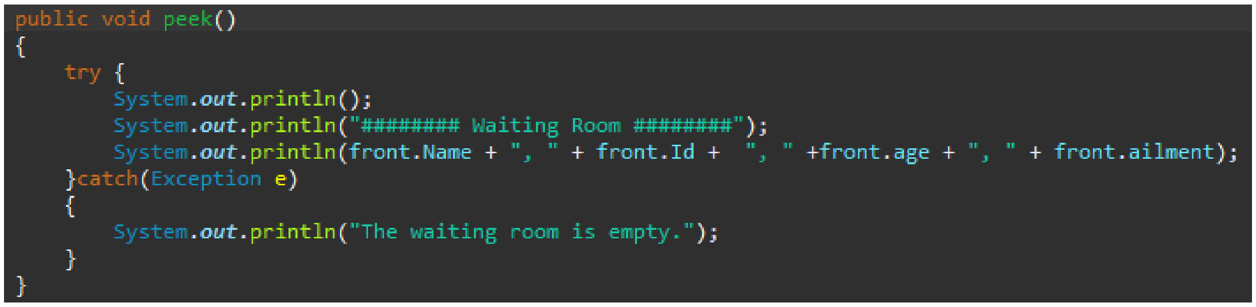
In my implementation, the stack is used to temporarily store discharged patients in the hospital system, the patient's info (being his name, id, ailment, emergency) is pushed into the stack and if the patients needs readmission, their data is popped right back into the main structure (linked list) being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


```
public class DischargeStack
{
    private int topSize;
    private int size;
    Patient top;

    public DischargeStack(int size)
    {
        top = null;
        this.size = size;
        topSize = -1;
    }
    public void push(String Name, int Id, int age, String ailment, boolean Emergency)
    {
        if(isFull())
        {
            System.out.println("Stack is full");
            return;
        }
        Patient newNode = new Patient(Name, Id, age, ailment, Emergency);
        newNode.next = top;
        top = newNode;
        topSize++;
    }
    public Patient pop()
    {
        if(isEmpty())
        {
            System.out.println("Stacked Array is Empty");
            return null;
        }
        System.out.println("\nPopped Patient: " + top.Name + " ID: " +top.Id); // Print the popped value
        Patient poppedPatient = top;


        top = top.next; // Move head to the next node
        topSize--; // Decrement the top index

        return poppedPatient;
    }
```

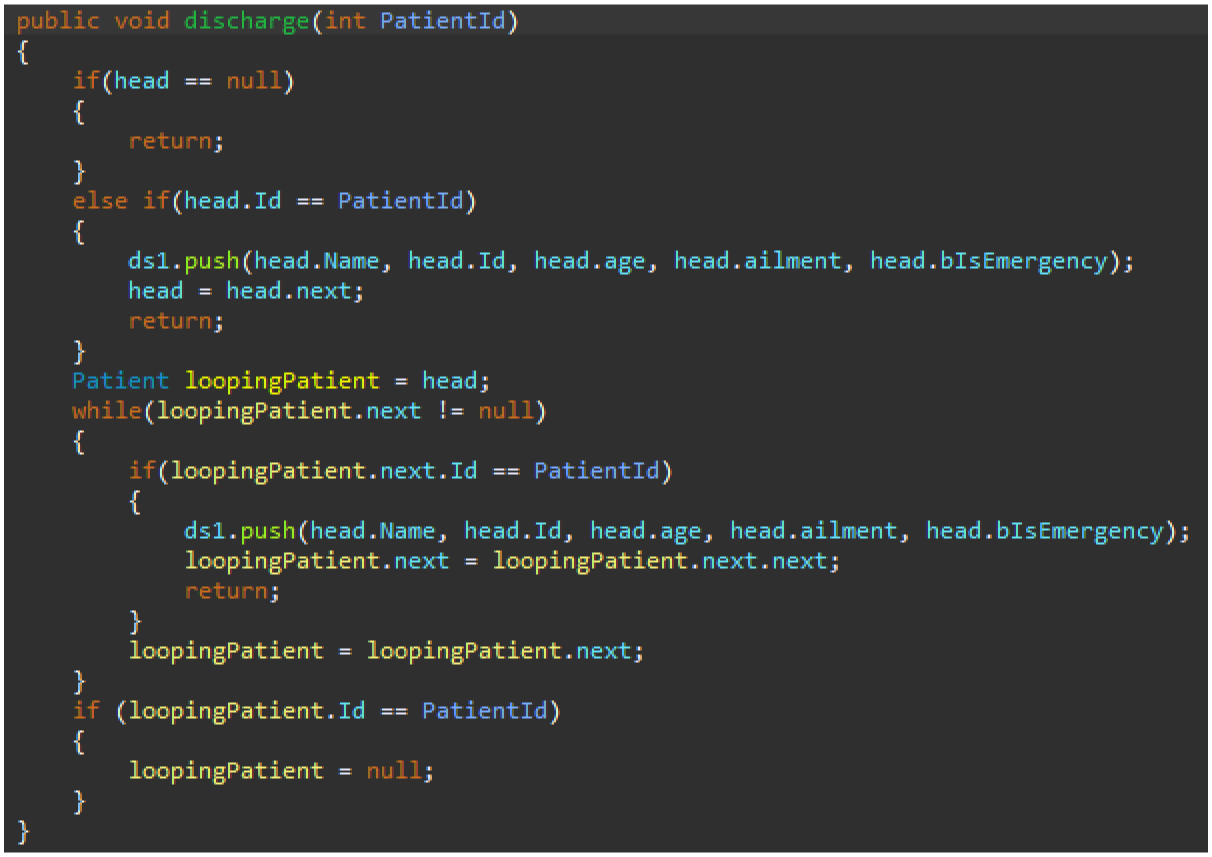


```java
public void peak()
{
    if(isEmpty())
    {
        return;
    }
    else
    {
        System.out.println("Top is " + top.Name + " ID: " +top.Id);
    }
}
public boolean isFull()
{
    return topSize == size - 1;
}
public int size()
{
    return topSize+1;
}
public boolean isEmpty()
{
    return top == null;
}
public void showDischarged()
{
    System.out.println();
    System.out.println("######## Discharged Patients ########");
    Patient loopingNode = top;
    while (loopingNode != null)
    {
        System.out.println(loopingNode.Name + " ID: " + loopingNode.Id  + " ");
        loopingNode = loopingNode.next;
    }
}
```

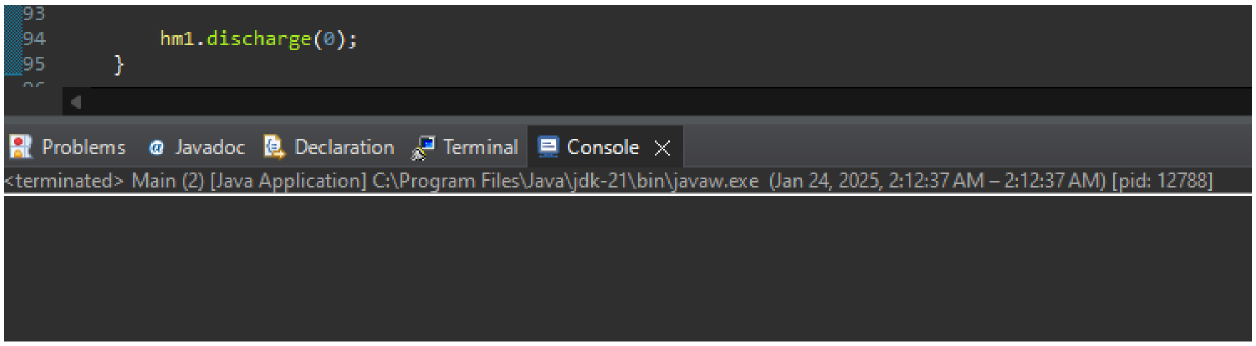
Question 3



```java
public void peek()
{
    try {
        System.out.println();
        System.out.println("######## Waiting Room ########");
        System.out.println(front.Name + ", " + front.Id +  ", " +front.age + ", " + front.ailment);
    }catch(Exception e)
    {
        System.out.println("The waiting room is empty.");
    }
}
```

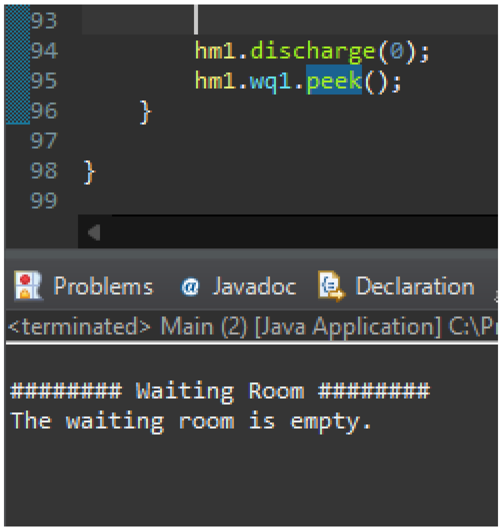


```
public void discharge(int PatientId)
{
    if(head == null)
    {
        return;
    }
    else if(head.Id == PatientId)
    {
        ds1.push(head.Name, head.Id, head.age, head.ailment, head.bIsEmergency);
        head = head.next;
        return;
    }
    Patient loopingPatient = head;
    while(loopingPatient.next != null)
    {
        if(loopingPatient.next.Id == PatientId)
        {
            ds1.push(head.Name, head.Id, head.age, head.ailment, head.bIsEmergency);
            loopingPatient.next = loopingPatient.next.next;
            return;
        }
        loopingPatient = loopingPatient.next;
    }
    if (loopingPatient.Id == PatientId)
    {
        loopingPatient = null;
    }
}
```



```
        hm1.discharge(0);
    }
```

```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (Jan 24, 2025, 2:12:37 AM – 2:12:37 AM) [pid: 12788]
```



```java

        hm1.discharge(0);
        hm1.wq1.peek();
    }

}

```

```
######## Waiting Room ########
The waiting room is empty.
```

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.

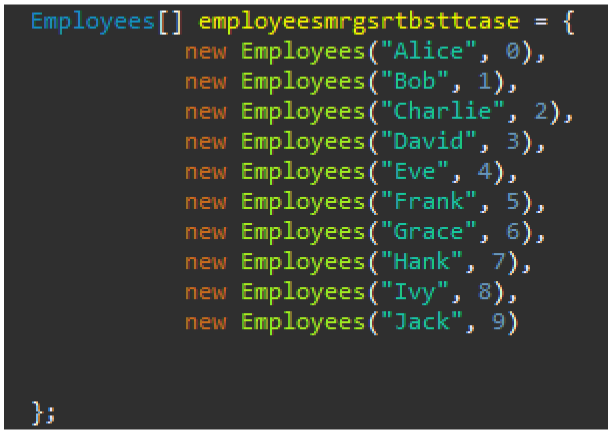
Question 4

Merge Sort

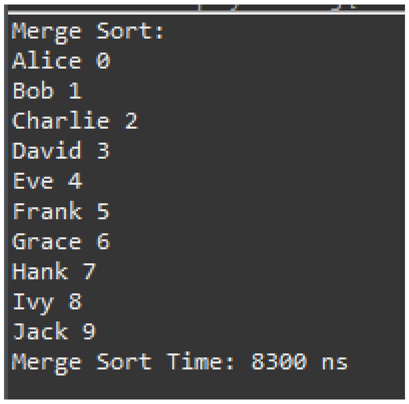
Merge Sort: first of all what does merge sort do? Merge sort in simple terms is a divide and conquer algorithm that splits the array into two halves recursively until it reaches 1 element from each array, then recursively sort them and merge them into sorted halves recursively until our array is sorted.

The time complexity in the best case scenario is : O(n log n) and here is the output for our best case scenario

Best Case

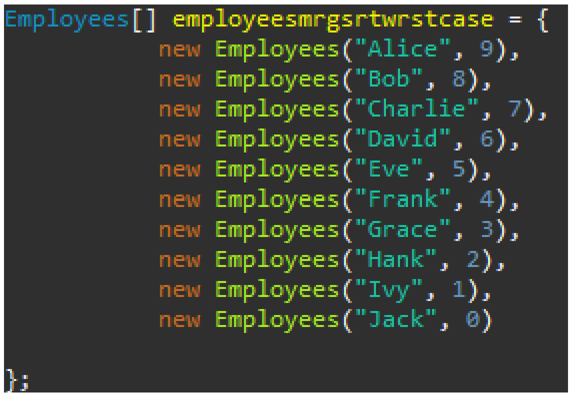


```java
Employees[] employeesmrgsrtbsttcase = {
        new Employees("Alice", 0),
        new Employees("Bob", 1),
        new Employees("Charlie", 2),
        new Employees("David", 3),
        new Employees("Eve", 4),
        new Employees("Frank", 5),
        new Employees("Grace", 6),
        new Employees("Hank", 7),
        new Employees("Ivy", 8),
        new Employees("Jack", 9)


};
```

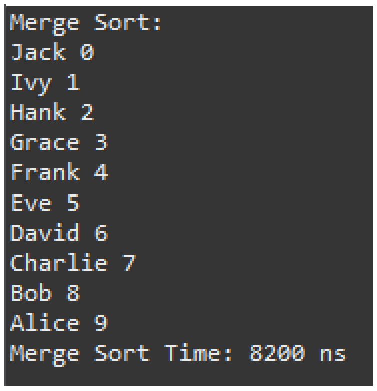


```
Merge Sort:
Alice 0
Bob 1
Charlie 2
David 3
Eve 4
Frank 5
Grace 6
Hank 7
Ivy 8
Jack 9
Merge Sort Time: 8300 ns
```

Worst Case



```
Employees[] employeesmrgsrtwrstcase = {
           new Employees("Alice", 9),
           new Employees("Bob", 8),
           new Employees("Charlie", 7),
           new Employees("David", 6),
           new Employees("Eve", 5),
           new Employees("Frank", 4),
           new Employees("Grace", 3),
           new Employees("Hank", 2),
           new Employees("Ivy", 1),
           new Employees("Jack", 0)
};
```

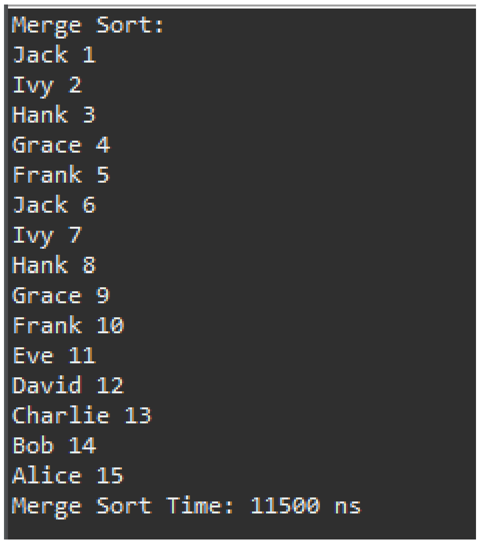


```
Merge Sort:
Jack 0
Ivy 1
Hank 2
Grace 3
Frank 4
Eve 5
David 6
Charlie 7
Bob 8
Alice 9
Merge Sort Time: 8200 ns
```

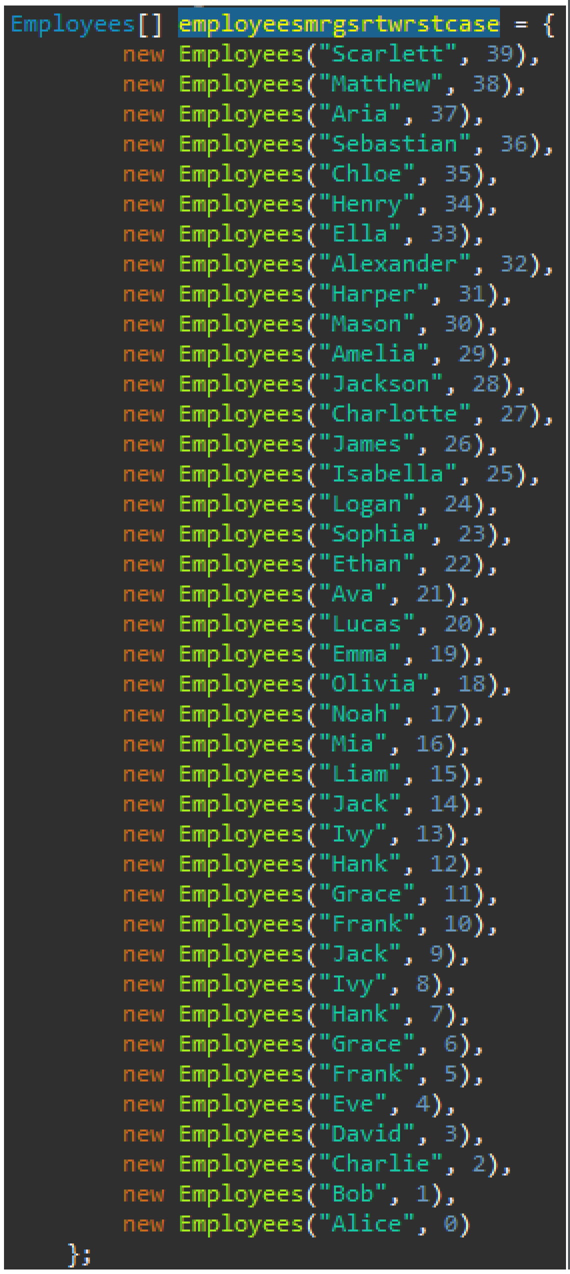
Larger size

And when comparing both cases and the time it took to compile, I can confidently conclude that the time complexity for best case scenario is the same as the worst case scenario as proved by the output

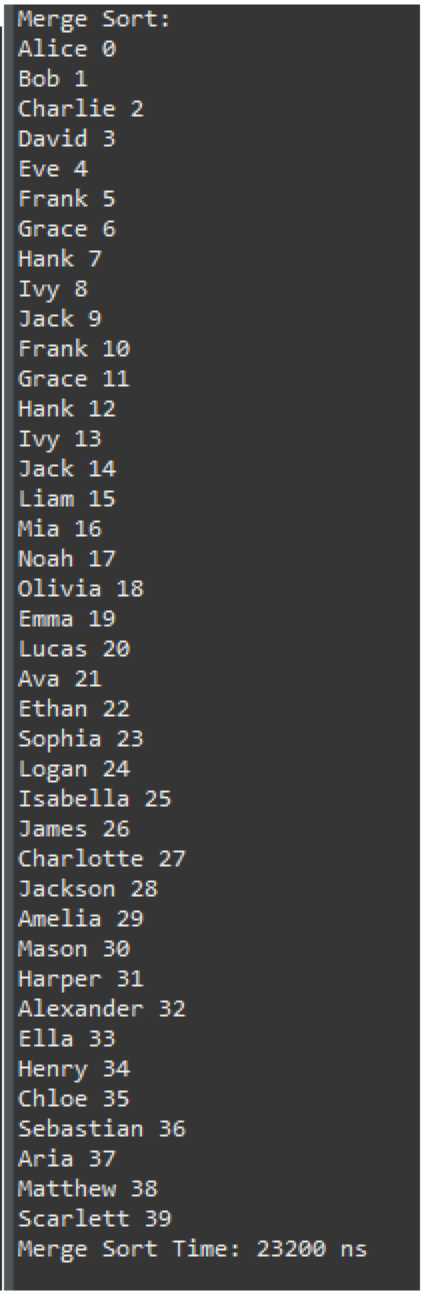
So when increasing the number of employees to be sorted we can witness that the time it takes also increase, even if we sort it ourselves or not, its going to take the same time when compiling



```
Merge Sort:
Jack 1
Ivy 2
Hank 3
Grace 4
Frank 5
Jack 6
Ivy 7
Hank 8
Grace 9
Frank 10
Eve 11
David 12
Charlie 13
Bob 14
Alice 15
Merge Sort Time: 11500 ns
```



```
Employees[] employeesmrgsrtwrstcase = {
        new Employees("Scarlett", 39),
        new Employees("Matthew", 38),
        new Employees("Aria", 37),
        new Employees("Sebastian", 36),
        new Employees("Chloe", 35),
        new Employees("Henry", 34),
        new Employees("Ella", 33),
        new Employees("Alexander", 32),
        new Employees("Harper", 31),
        new Employees("Mason", 30),
        new Employees("Amelia", 29),
        new Employees("Jackson", 28),
        new Employees("Charlotte", 27),
        new Employees("James", 26),
        new Employees("Isabella", 25),
        new Employees("Logan", 24),
        new Employees("Sophia", 23),
        new Employees("Ethan", 22),
        new Employees("Ava", 21),
        new Employees("Lucas", 20),
        new Employees("Emma", 19),
        new Employees("Olivia", 18),
        new Employees("Noah", 17),
        new Employees("Mia", 16),
        new Employees("Liam", 15),
        new Employees("Jack", 14),
        new Employees("Ivy", 13),
        new Employees("Hank", 12),
        new Employees("Grace", 11),
        new Employees("Frank", 10),
        new Employees("Jack", 9),
        new Employees("Ivy", 8),
        new Employees("Hank", 7),
        new Employees("Grace", 6),
        new Employees("Frank", 5),
        new Employees("Eve", 4),
        new Employees("David", 3),
        new Employees("Charlie", 2),
        new Employees("Bob", 1),
        new Employees("Alice", 0)
    };
```



```
Merge Sort:
Alice 0
Bob 1
Charlie 2
David 3
Eve 4
Frank 5
Grace 6
Hank 7
Ivy 8
Jack 9
Frank 10
Grace 11
Hank 12
Ivy 13
Jack 14
Liam 15
Mia 16
Noah 17
Olivia 18
Emma 19
Lucas 20
Ava 21
Ethan 22
Sophia 23
Logan 24
Isabella 25
James 26
Charlotte 27
Jackson 28
Amelia 29
Mason 30
Harper 31
Alexander 32
Ella 33
Henry 34
Chloe 35
Sebastian 36
Aria 37
Matthew 38
Scarlett 39
Merge Sort Time: 23200 ns
```

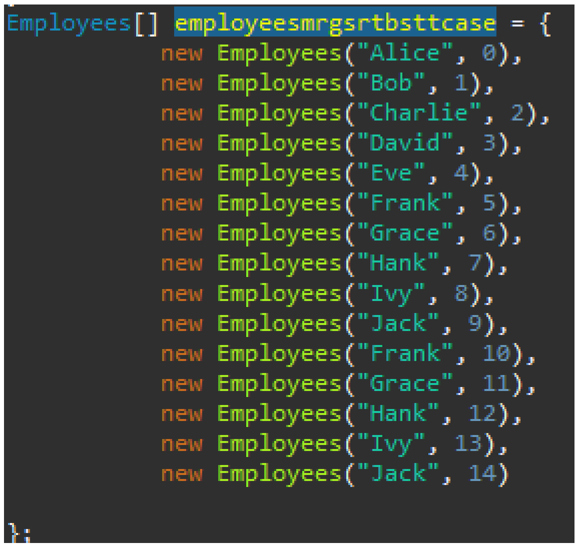
We observe this dramatic increase because we increased the number of entries in our array, and when comparing it to the selection sort we can see that selection sort is faster that is because merge sort is suitable for larger datasets, I am talking about 100 entries and more not 40,

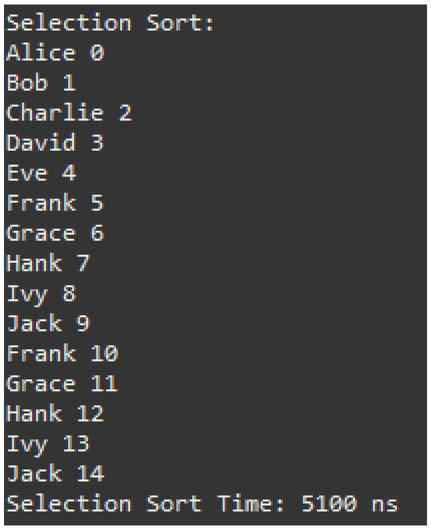
[Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks]

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: Time complexity for the best case scenario is O(n^2)

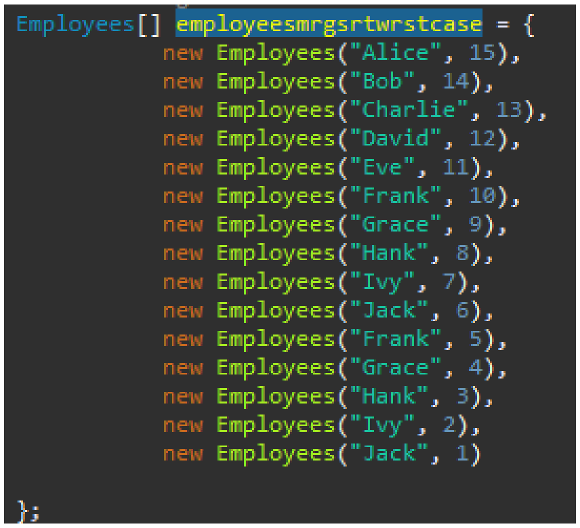
Output:

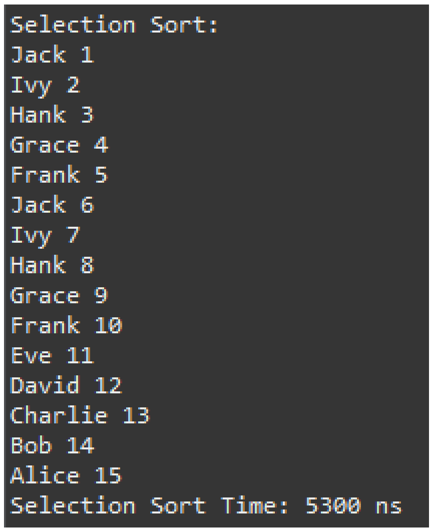

```
Employees[] employeesmrgsrtbsttcase = {
            new Employees("Alice", 0),
            new Employees("Bob", 1),
            new Employees("Charlie", 2),
            new Employees("David", 3),
            new Employees("Eve", 4),
            new Employees("Frank", 5),
            new Employees("Grace", 6),
            new Employees("Hank", 7),
            new Employees("Ivy", 8),
            new Employees("Jack", 9),
            new Employees("Frank", 10),
            new Employees("Grace", 11),
            new Employees("Hank", 12),
            new Employees("Ivy", 13),
            new Employees("Jack", 14)

};
```


```
Selection Sort:
Alice 0
Bob 1
Charlie 2
David 3
Eve 4
Frank 5
Grace 6
Hank 7
Ivy 8
Jack 9
Frank 10
Grace 11
Hank 12
Ivy 13
Jack 14
Selection Sort Time: 5100 ns
```

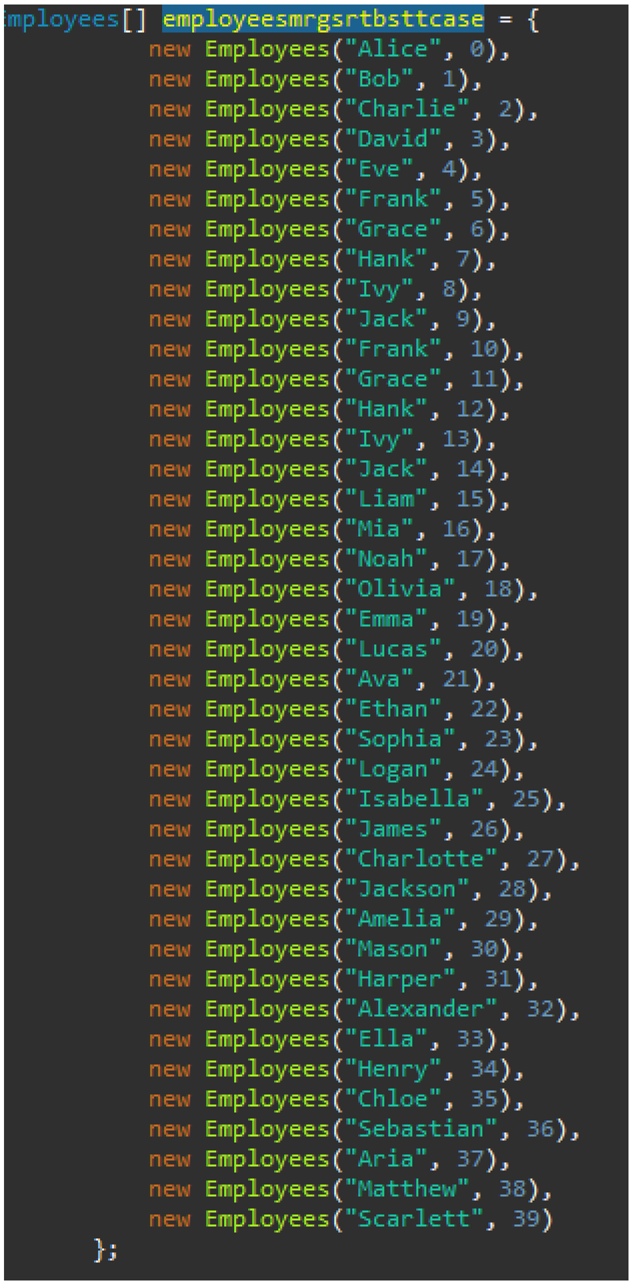
As observed above, the output for the selection sort is much lower than the output for the selection sort than the merge sort, even though in the selection sort we have a 1.5x bigger array.

Worst Case: Time complexity of also an O(n^2)

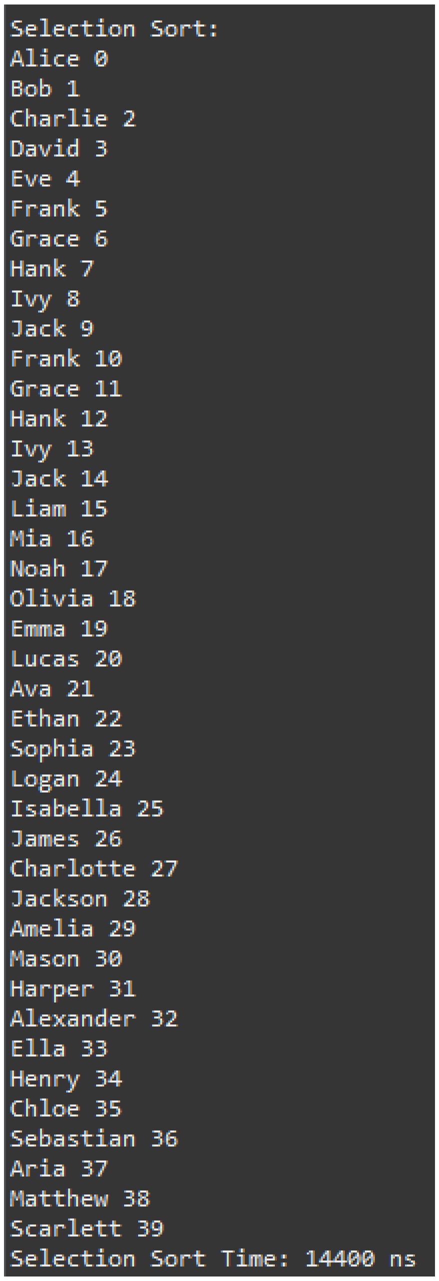

```
Employees[] employeesmrgsrtwrstcase = {
            new Employees("Alice", 15),
            new Employees("Bob", 14),
            new Employees("Charlie", 13),
            new Employees("David", 12),
            new Employees("Eve", 11),
            new Employees("Frank", 10),
            new Employees("Grace", 9),
            new Employees("Hank", 8),
            new Employees("Ivy", 7),
            new Employees("Jack", 6),
            new Employees("Frank", 5),
            new Employees("Grace", 4),
            new Employees("Hank", 3),
            new Employees("Ivy", 2),
            new Employees("Jack", 1)

};
```


```
Selection Sort:
Jack 1
Ivy 2
Hank 3
Grace 4
Frank 5
Jack 6
Ivy 7
Hank 8
Grace 9
Frank 10
Eve 11
David 12
Charlie 13
Bob 14
Alice 15
Selection Sort Time: 5300 ns
```

Almost the same amount of time taken for the best case scenario, but when increasing the array and making it bigger, we are going to observe that the selection sort is much slower than merge sort.

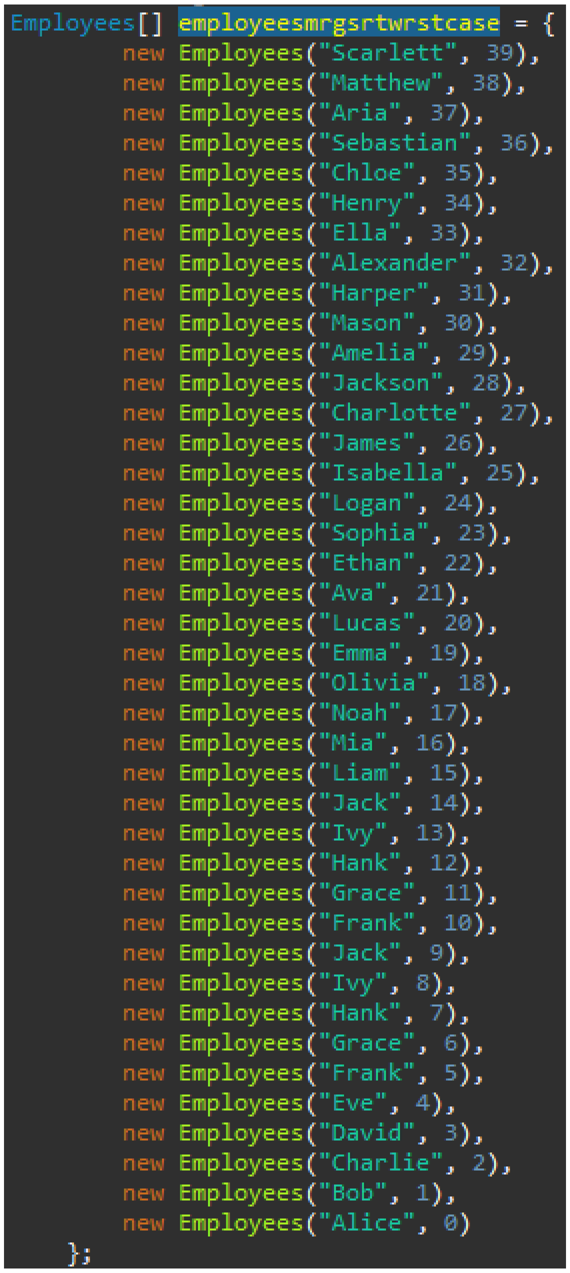


```
mployees[] employeesmrgsrtbsttcase = {
            new Employees("Alice", 0),
            new Employees("Bob", 1),
            new Employees("Charlie", 2),
            new Employees("David", 3),
            new Employees("Eve", 4),
            new Employees("Frank", 5),
            new Employees("Grace", 6),
            new Employees("Hank", 7),
            new Employees("Ivy", 8),
            new Employees("Jack", 9),
            new Employees("Frank", 10),
            new Employees("Grace", 11),
            new Employees("Hank", 12),
            new Employees("Ivy", 13),
            new Employees("Jack", 14),
            new Employees("Liam", 15),
            new Employees("Mia", 16),
            new Employees("Noah", 17),
            new Employees("Olivia", 18),
            new Employees("Emma", 19),
            new Employees("Lucas", 20),
            new Employees("Ava", 21),
            new Employees("Ethan", 22),
            new Employees("Sophia", 23),
            new Employees("Logan", 24),
            new Employees("Isabella", 25),
            new Employees("James", 26),
            new Employees("Charlotte", 27),
            new Employees("Jackson", 28),
            new Employees("Amelia", 29),
            new Employees("Mason", 30),
            new Employees("Harper", 31),
            new Employees("Alexander", 32),
            new Employees("Ella", 33),
            new Employees("Henry", 34),
            new Employees("Chloe", 35),
            new Employees("Sebastian", 36),
            new Employees("Aria", 37),
            new Employees("Matthew", 38),
            new Employees("Scarlett", 39)
        };
```

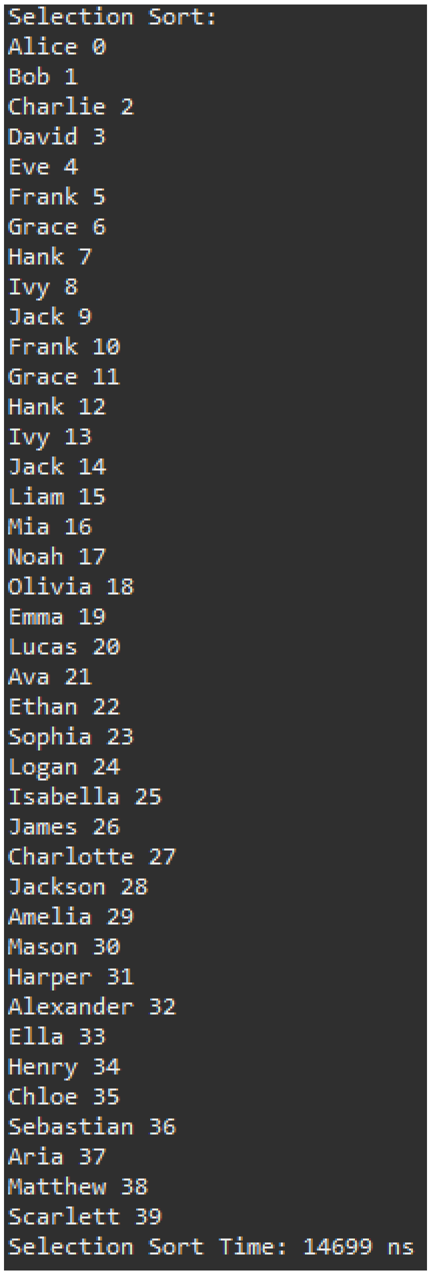


```
Selection Sort:
Alice 0
Bob 1
Charlie 2
David 3
Eve 4
Frank 5
Grace 6
Hank 7
Ivy 8
Jack 9
Frank 10
Grace 11
Hank 12
Ivy 13
Jack 14
Liam 15
Mia 16
Noah 17
Olivia 18
Emma 19
Lucas 20
Ava 21
Ethan 22
Sophia 23
Logan 24
Isabella 25
James 26
Charlotte 27
Jackson 28
Amelia 29
Mason 30
Harper 31
Alexander 32
Ella 33
Henry 34
Chloe 35
Sebastian 36
Aria 37
Matthew 38
Scarlett 39
Selection Sort Time: 14400 ns
```

Worst case



```
Employees[] employeesmrgsrtwrstcase = {
        new Employees("Scarlett", 39),
        new Employees("Matthew", 38),
        new Employees("Aria", 37),
        new Employees("Sebastian", 36),
        new Employees("Chloe", 35),
        new Employees("Henry", 34),
        new Employees("Ella", 33),
        new Employees("Alexander", 32),
        new Employees("Harper", 31),
        new Employees("Mason", 30),
        new Employees("Amelia", 29),
        new Employees("Jackson", 28),
        new Employees("Charlotte", 27),
        new Employees("James", 26),
        new Employees("Isabella", 25),
        new Employees("Logan", 24),
        new Employees("Sophia", 23),
        new Employees("Ethan", 22),
        new Employees("Ava", 21),
        new Employees("Lucas", 20),
        new Employees("Emma", 19),
        new Employees("Olivia", 18),
        new Employees("Noah", 17),
        new Employees("Mia", 16),
        new Employees("Liam", 15),
        new Employees("Jack", 14),
        new Employees("Ivy", 13),
        new Employees("Hank", 12),
        new Employees("Grace", 11),
        new Employees("Frank", 10),
        new Employees("Jack", 9),
        new Employees("Ivy", 8),
        new Employees("Hank", 7),
        new Employees("Grace", 6),
        new Employees("Frank", 5),
        new Employees("Eve", 4),
        new Employees("David", 3),
        new Employees("Charlie", 2),
        new Employees("Bob", 1),
        new Employees("Alice", 0)
    };
```



```
Selection Sort:
Alice 0
Bob 1
Charlie 2
David 3
Eve 4
Frank 5
Grace 6
Hank 7
Ivy 8
Jack 9
Frank 10
Grace 11
Hank 12
Ivy 13
Jack 14
Liam 15
Mia 16
Noah 17
Olivia 18
Emma 19
Lucas 20
Ava 21
Ethan 22
Sophia 23
Logan 24
Isabella 25
James 26
Charlotte 27
Jackson 28
Amelia 29
Mason 30
Harper 31
Alexander 32
Ella 33
Henry 34
Chloe 35
Sebastian 36
Aria 37
Matthew 38
Scarlett 39
Selection Sort Time: 14699 ns
```

[Selection Sort - GeeksforGeeks]

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

Question 5

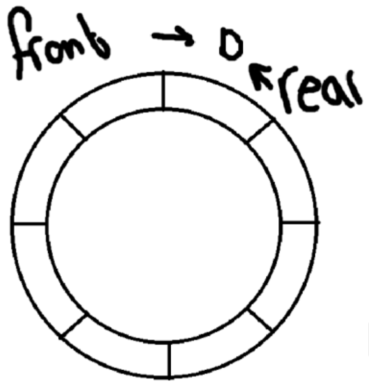
Circular queue, with a circular queue we need to make sure that the elements wrap around as in a circle thereby reducing the space consumption seen in a linear queue

Enqueue

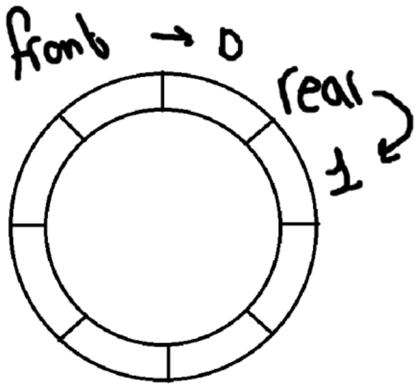
Front and rear is set to -1

$$\text{front} = \text{rear} = -1$$

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



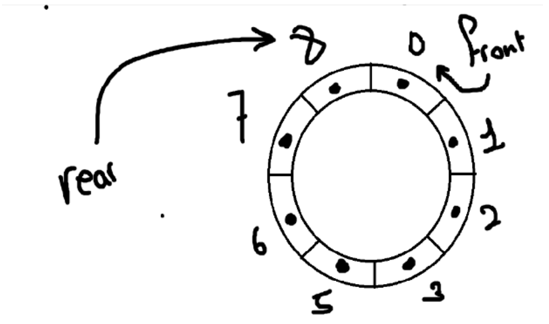
Now to determine the index of the rear, we are gonna use this equation

$$(\text{rear} + 1) \% \text{length}$$

$$(0+1) \% 8 = 1$$

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

$$(8+1)\%8=1$$

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.

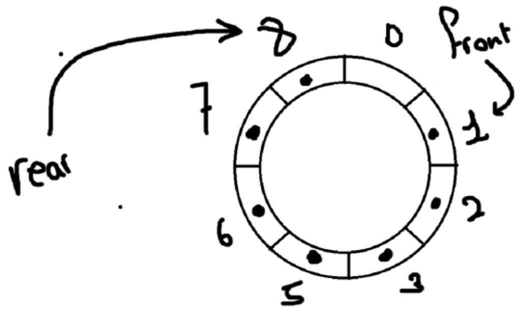
Same concept for the dequeue

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

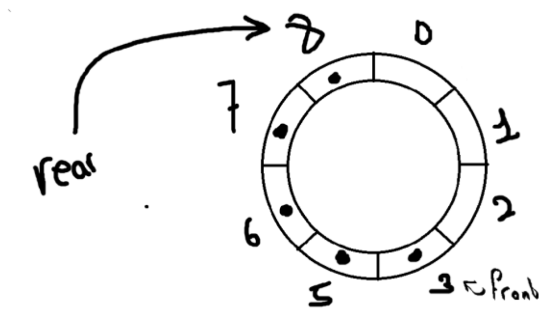
$$(front+1)\%length$$

Dequeue once:

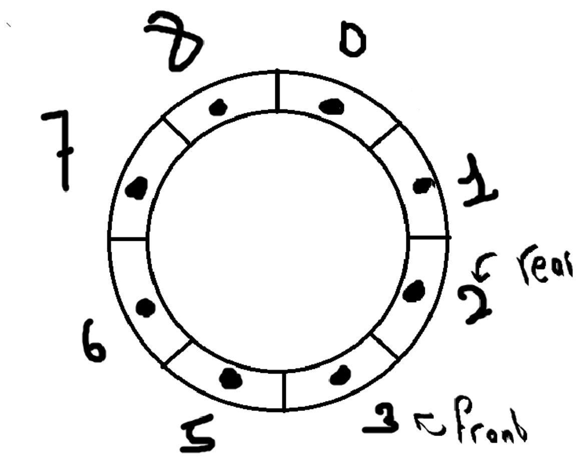
$$(0+1)\%8=1$$



Lets dequeue twice



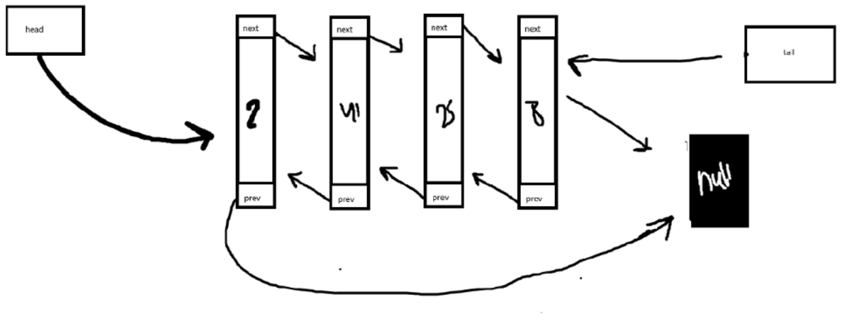
Enqueue thrice now



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!

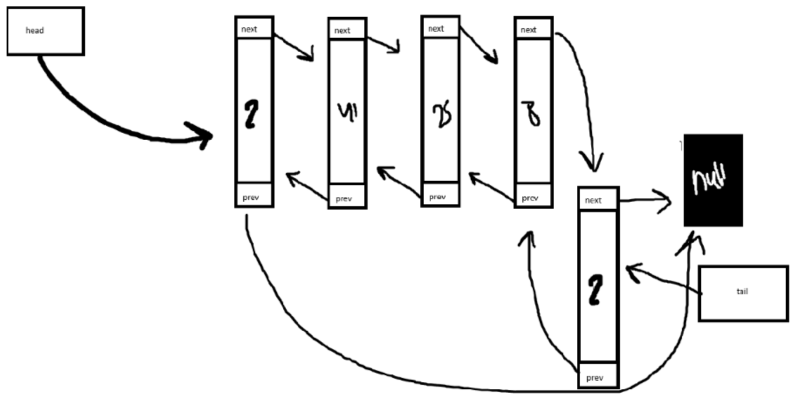
Circular Queue Implementation - Array

For the doubly linked list this is how its gonna look

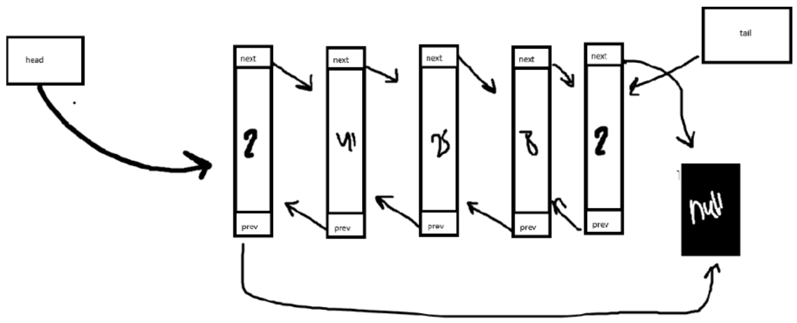


Lets enqueue one element, as we know the queue works by adding elements to the last index not the first index like the stack

Here is out its gonna look like

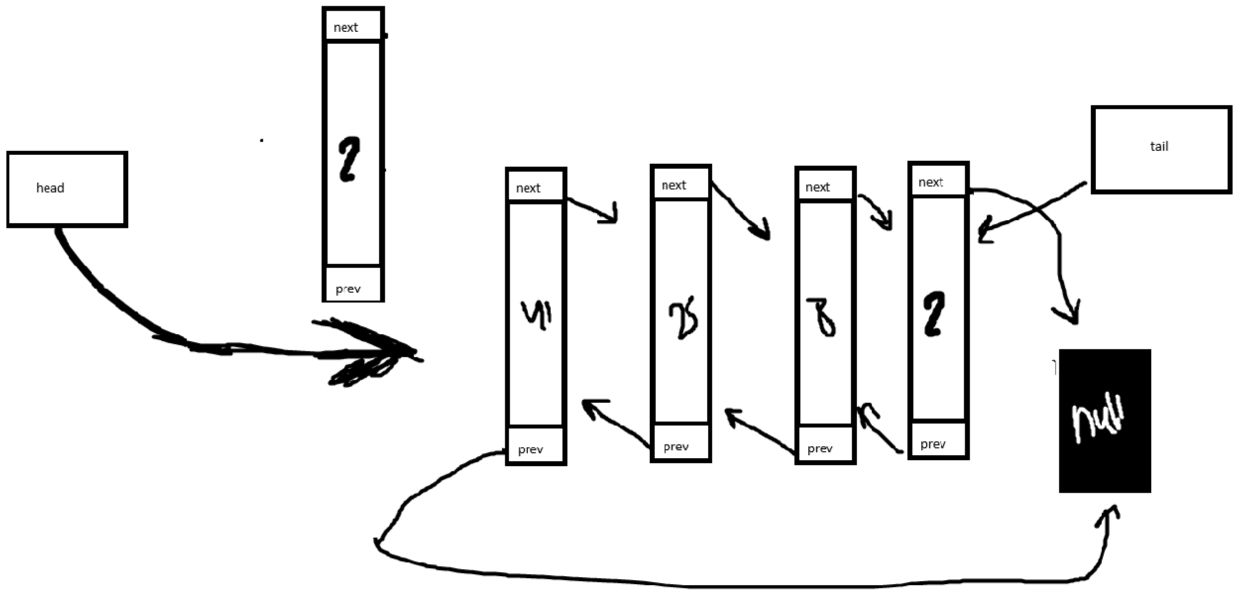


And as shown in the image above, we add the node in the last index, we link the next of the previous node to the new node. And the link prev to the previous node, and the next of the new node to the null and make sure that you save a reference to it by assigning it to the tail variable.



Stacks, Queues, and Double Ended Queues (Deques) - YouTube

And here is the dequeue



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

Time Complexity

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.

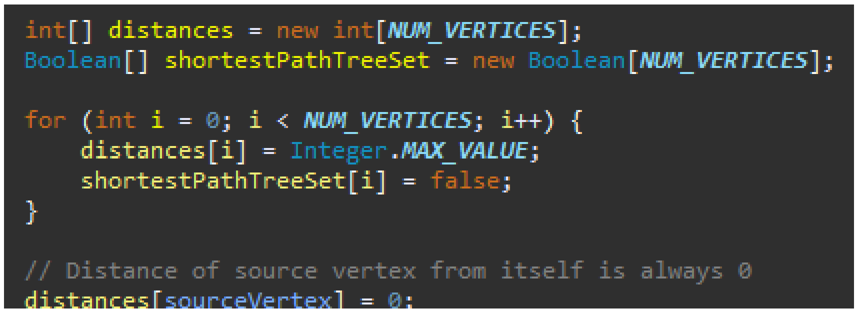
[Preorder vs Inorder vs Postorder - GeeksforGeeks]

Question 7

Dijkstra Algorithm

Initialization

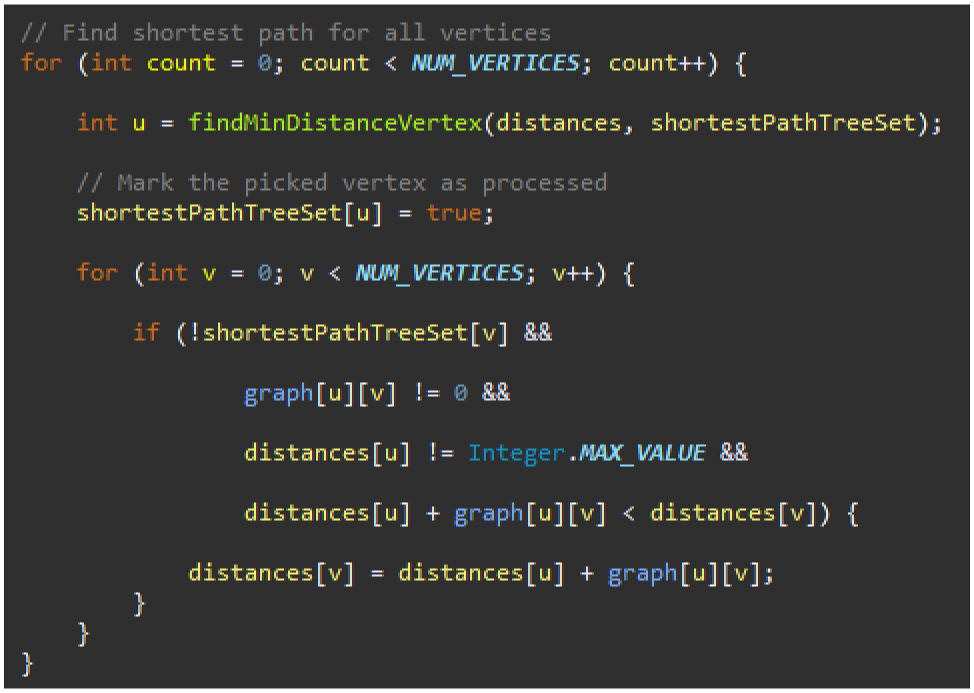
Set the distance of the source vertex as 0 and the rest of the vertices are infinite, it will then make sure that all of the vertices are marked unprocessed as in "false"



```
int[] distances = new int[NUM_VERTICES];
Boolean[] shortestPathTreeSet = new Boolean[NUM_VERTICES];

for (int i = 0; i < NUM_VERTICES; i++) {
    distances[i] = Integer.MAX_VALUE;
    shortestPathTreeSet[i] = false;
}

// Distance of source vertex from itself is always 0
distances[sourceVertex] = 0;
```

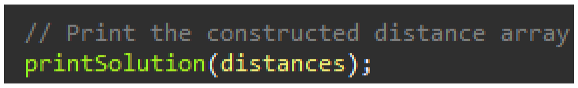
Main Loop

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"



```
// Find shortest path for all vertices
for (int count = 0; count < NUM_VERTICES; count++) {

    int u = findMinDistanceVertex(distances, shortestPathTreeSet);

    // Mark the picked vertex as processed
    shortestPathTreeSet[u] = true;

    for (int v = 0; v < NUM_VERTICES; v++) {

        if (!shortestPathTreeSet[v] &&

                graph[u][v] != 0 &&

                distances[u] != Integer.MAX_VALUE &&

                distances[u] + graph[u][v] < distances[v]) {

            distances[v] = distances[u] + graph[u][v];
        }
    }
}
```

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

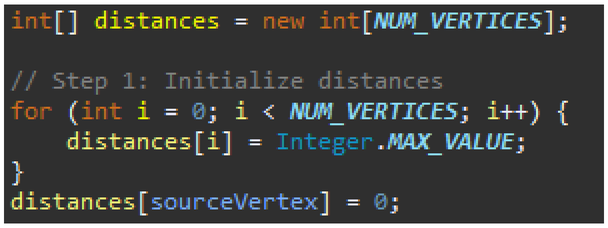


```
// Print the constructed distance array
printSolution(distances);
```

Bellman Ford

Initialization

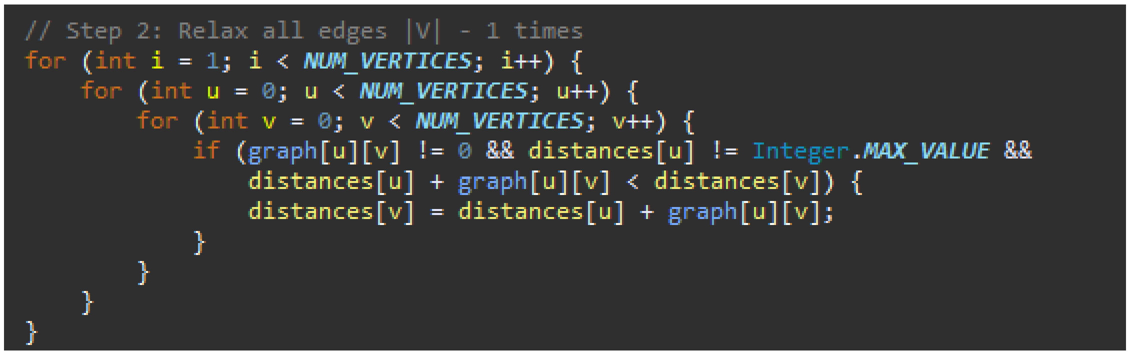
Its going to set the source vertex to 0 and every other vertex to infinity



```
int[] distances = new int[NUM_VERTICES];

// Step 1: Initialize distances
for (int i = 0; i < NUM_VERTICES; i++) {
    distances[i] = Integer.MAX_VALUE;
}
distances[sourceVertex] = 0;
```

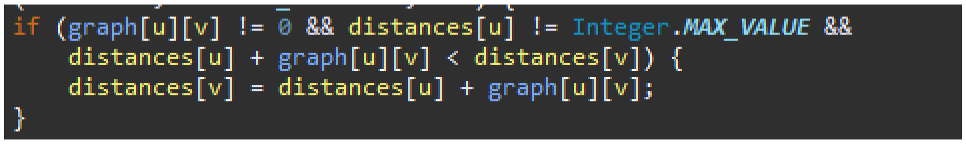
Relaxation

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



```
// Step 2: Relax all edges |V| - 1 times
for (int i = 1; i < NUM_VERTICES; i++) {
    for (int u = 0; u < NUM_VERTICES; u++) {
        for (int v = 0; v < NUM_VERTICES; v++) {
            if (graph[u][v] != 0 && distances[u] != Integer.MAX_VALUE &&
                distances[u] + graph[u][v] < distances[v]) {
                distances[v] = distances[u] + graph[u][v];
            }
        }
    }
}
```

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

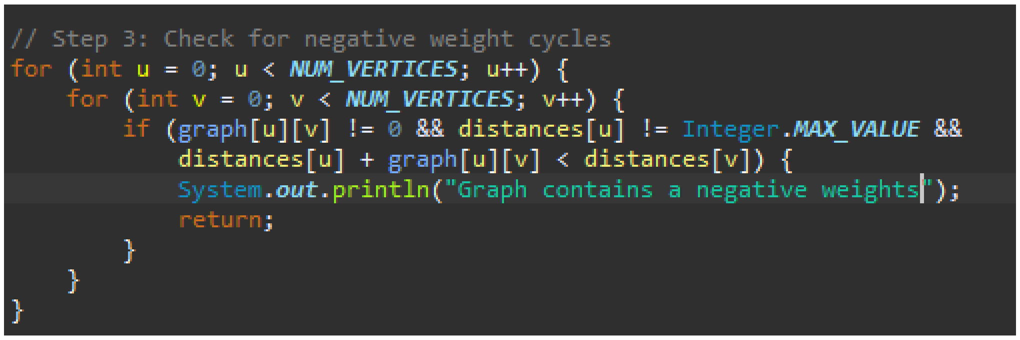


```
if (graph[u][v] != 0 && distances[u] != Integer.MAX_VALUE &&
    distances[u] + graph[u][v] < distances[v]) {
    distances[v] = distances[u] + graph[u][v];
}
```

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

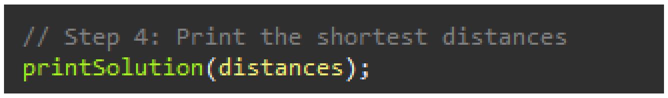
Detection of Negative Cycles

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

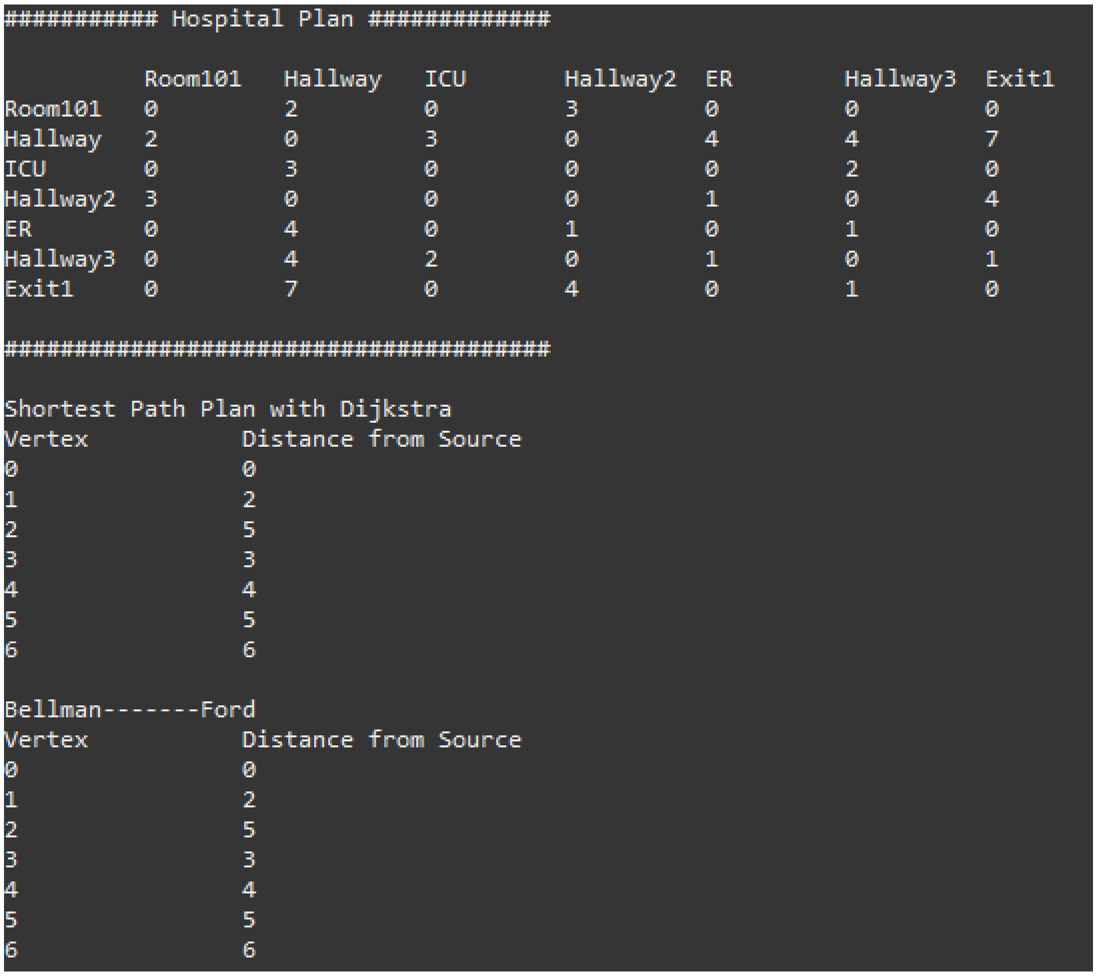


```
// Step 3: Check for negative weight cycles
for (int u = 0; u < NUM_VERTICES; u++) {
    for (int v = 0; v < NUM_VERTICES; v++) {
        if (graph[u][v] != 0 && distances[u] != Integer.MAX_VALUE &&
            distances[u] + graph[u][v] < distances[v]) {
            System.out.println("Graph contains a negative weights");
            return;
        }
    }
}
```

Then its going to print the output

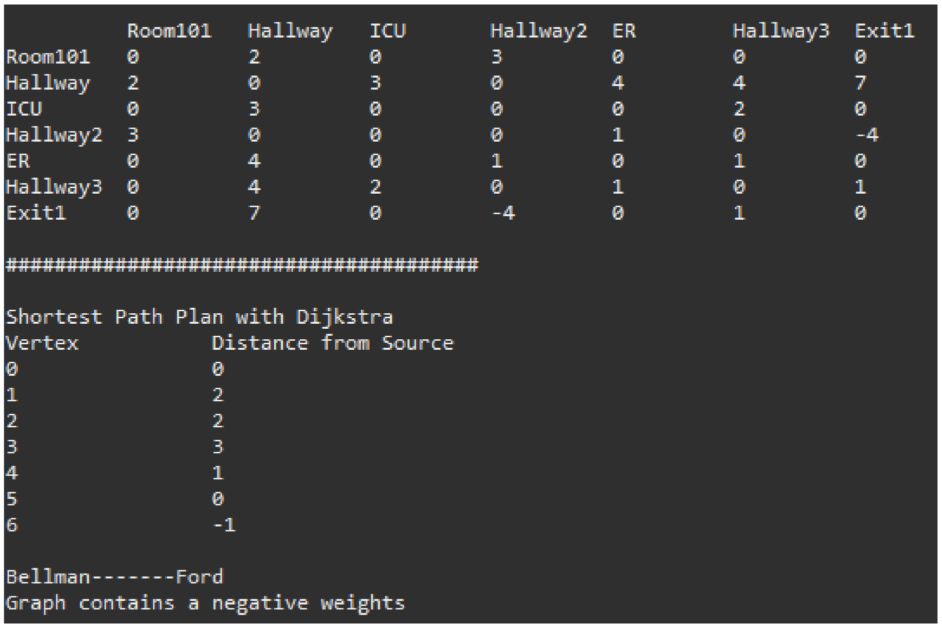


```
// Step 4: Print the shortest distances
printSolution(distances);
```



```
########### Hospital Plan #############

          Room101   Hallway   ICU       Hallway2  ER        Hallway3  Exit1
Room101   0         2         0         3         0         0         0
Hallway   2         0         3         0         4         4         7
ICU       0         3         0         0         0         2         0
Hallway2  3         0         0         0         1         0         4
ER        0         4         0         1         0         1         0
Hallway3  0         4         2         0         1         0         1
Exit1     0         7         0         4         0         1         0

#######################################

Shortest Path Plan with Dijkstra
Vertex           Distance from Source
0                0
1                2
2                5
3                3
4                4
5                5
6                6

Bellman-------Ford
Vertex           Distance from Source
0                0
1                2
2                5
3                3
4                4
5                5
6                6
```

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



```
          Room101   Hallway   ICU       Hallway2  ER        Hallway3  Exit1
Room101   0         2         0         3         0         0         0
Hallway   2         0         3         0         4         4         7
ICU       0         3         0         0         0         2         0
Hallway2  3         0         0         0         1         0         -4
ER        0         4         0         1         0         1         0
Hallway3  0         4         2         0         1         0         1
Exit1     0         7         0         -4        0         1         0

#######################################

Shortest Path Plan with Dijkstra
Vertex           Distance from Source
0                0
1                2
2                2
3                3
4                1
5                0
6                -1

Bellman-------Ford
Graph contains a negative weights
```

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

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 its because useless because you can repeatedly travel into the negative cycle and it will inconsequence 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