ASSIGNMENT NO: 3

Group No: 5

1) Problem Statement:

Develop a program Token based algorithm or ring topology.

2) Code With Comment-Line:

Main.java

```
package Assignment_3;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class main {
	static String current_token_holder;//variable for token holder
	static ArrayList<String> req_queue = new ArrayList<>();//queue data structure for request
list
	//Method for processing request through several hop
	public static void sendRequest(String nd,ArrayList<String> rq,CircularQueue nodeList) {
		String temp_node=nd;
		while(true) {
			temp_node=nodeList.check(temp_node);//method to search for
neighbouring node
			if(temp_node.equals(rq.get(0))) {
				grantRequest(rq.get(0));// finally granting request for the next
requesting node
				break;
			}else {
				System.out.println(temp_node+" got message and
passed");//passing request from node to node
			}
		}
	}
	private static void grantRequest(String cs_node) {
		// TODO Auto-generated method stub
		current_token_holder=cs_node;//token delivered to the next node
		System.out.println("Token delivered to "+cs_node);
		 if(req_queue.size()>1) {
			 System.out.println("in queue-->"+req_queue.get(1));//queue description
after token delivery
```

```
		}
	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		//Necessary variable initialisation
		 int no_of_node=0;
		 CircularQueue nodeList = null;//circular queue DAta structure for token ring
		 char[] cnnt;
		 //reading data from file input
		 try {
			FileReader fileReader = new FileReader("test12.txt");
			BufferedReader bufferedReader = new BufferedReader(fileReader);
			no_of_node= Integer.parseInt(bufferedReader.readLine());//reading total
no of nodes
			System.out.println("no of nodes:"+no_of_node);
			nodeList=new CircularQueue(no_of_node);
			cnnt = bufferedReader.readLine().toCharArray();

			for(int i=0;i<no_of_node;i++) {
				nodeList.enQueue(Character.toString(cnnt[i]));//reading order of
nodes
			}
			cnnt=null;
			cnnt = bufferedReader.readLine().toCharArray();
			for(int i=0;i<cnnt.length;i++) {
				req_queue.add(Character.toString(cnnt[i]));//reading order of node
request
			}
			current_token_holder=bufferedReader.readLine();//reading first token
holder
		 }catch (IOException e) {
		    e.printStackTrace();
		}

		 System.out.println("Currently token holder is : "+current_token_holder);
		 System.out.println(req_queue.get(0)+"&"+req_queue.get(1)+" sending request for
token ");
		 System.out.println("in queue-->"+req_queue.get(0)+","+req_queue.get(1));
		 //sending requset for the first time
		 sendRequest(current_token_holder,req_queue,nodeList);
		 if(current_token_holder.equalsIgnoreCase(req_queue.get(0))) {
			req_queue.remove(0);
		 }
		 //a loop for generating continous request generation for access token
		 while(! req_queue.isEmpty()) {
			System.out.println("Currently token holder is :
"+current_token_holder);//current token holder
			if(req_queue.size()>1) {
```

```
                                        System.out.println(req_queue.get(1)+" sending request for token
");//generating new request
                                        System.out.println("in queue--
>"+req_queue.get(0)+","+req_queue.get(1));//current queue description
                                }

                                //Accessing token reqest from requesting node to current token holder
                                sendRequest(current_token_holder,req_queue,nodeList);
                                if(current_token_holder.equalsIgnoreCase(req_queue.get(0))) {
                                        req_queue.remove(0);//remove current token holder from request
queue
                                }

                        }
                }

}
```

CircularQueue.java

```
package Assignment_3;
//Java program for insertion and
//deletion in Circular Queue
import java.util.ArrayList;

class CircularQueue{

//Declaring the class variables.
private int size, front, rear;

//Declaring array list of integer type.
private ArrayList<String> queue = new ArrayList<String>();

//Constructor
CircularQueue(int size)
{
        this.size = size;
        this.front = this.rear = -1;
}

//Method to insert a new element in the queue.
public void enQueue(String data)
{

        // Condition if queue is full.
        if((front == 0 && rear == size - 1) ||
        (rear == (front - 1) % (size - 1)))
        {
```

```
                System.out.print("Queue is Full");
        }

        // condition for empty queue.
        else if(front == -1)
        {
                front = 0;
                rear = 0;
                queue.add(rear, data);
        }

        else if(rear == size - 1 && front != 0)
        {
                rear = 0;
                queue.set(rear, data);
        }

        else
        {
                rear = (rear + 1);

                // Adding a new element if
                if(front <= rear)
                {
                        queue.add(rear, data);
                }

                // Else updating old value
                else
                {
                        queue.set(rear, data);
                }
        }
}

//Function to dequeue an element
//form th queue.
public String deQueue()
{
        String temp;

        // Condition for empty queue.
        if(front == -1)
        {
                System.out.print("Queue is Empty");

                // Return -1 in case of empty queue
                return "-1";
```

```
        }

        temp = queue.get(front);

        // Condition for only one element
        if(front == rear)
        {
                front = -1;
                rear = -1;
        }

        else if(front == size - 1)
        {
                front = 0;
        }
        else
        {
                front = front + 1;
        }

        // Returns the dequeued element
        return temp;
}

//Method to display the elements of queue
public void displayQueue()
{

        // Condition for empty queue.
        if(front == -1)
        {
                System.out.print("Queue is Empty");
                return;
        }

        // If rear has not crossed the max size
        // or queue rear is still greater then
        // front.
        System.out.print("Elements in the " +
                                        "circular queue are: ");

        if(rear >= front)
        {

                // Loop to print elements from
                // front to rear.
                for(int i = front; i <= rear; i++)
                {
```

```
                System.out.print(queue.get(i));
                System.out.print(" ");
            }
            System.out.println();
        }

        // If rear crossed the max index and
        // indexing has started in loop
        else
        {

            // Loop for printing elements from
            // front to max size or last index
            for(int i = front; i < size; i++)
            {
                System.out.print(queue.get(i));
                System.out.print(" ");
            }

            // Loop for printing elements from
            // 0th index till rear position
            for(int i = 0; i <= rear; i++)
            {
                System.out.print(queue.get(i));
                System.out.print(" ");
            }
            System.out.println();
        }
}

public String check(String temp_node) {
        // TODO Auto-generated method stub
        String ret_node=null;
        int i=front;
        while(i<=rear) {
            if(queue.get(i).equalsIgnoreCase(temp_node)) {
                ret_node=queue.get((i+1)%size);
                break;
            }
            i=(i+1)%size;
        }
        return ret_node;
}
}
```

3) Prerequisite:

NOTE:

In this algorithm the input file format will be:

- Total number of nodes
- Actual order of nodes in token ring (without white space)
- Order of nodes requesting for token except initiator (without white space)
- Initiator node

Example:

```
7
abcdefg
caed
f
```

4) Code Run:

INPUT 1:

```
10
abcdefghij
caedigb
f
```

OUTPUT 1:

```
no of nodes:10
Currently token holder is : f
c&a sending request for token
in queue-->c,a
g got message and passed
h got message and passed
i got message and passed
j got message and passed
a got message and passed
b got message and passed
Token delivered to c
in queue-->a
Currently token holder is : c
e sending request for token
in queue-->a,e
d got message and passed
e got message and passed
f got message and passed
g got message and passed
h got message and passed
i got message and passed
j got message and passed
Token delivered to a
in queue-->e
Currently token holder is : a
d sending request for token
```

```
in queue-->e,d
b got message and passed
c got message and passed
d got message and passed
Token delivered to e
in queue-->d
Currently token holder is : e
i sending request for token
in queue-->d,i
f got message and passed
g got message and passed
h got message and passed
i got message and passed
j got message and passed
a got message and passed
b got message and passed
c got message and passed
Token delivered to d
in queue-->i
Currently token holder is : d
g sending request for token
in queue-->i,g
e got message and passed
f got message and passed
g got message and passed
```

```
h got message and passed
Token delivered to i
in queue-->g
Currently token holder is : i
b sending request for token
in queue-->g,b
j got message and passed
a got message and passed
b got message and passed
c got message and passed
d got message and passed
e got message and passed
f got message and passed
Token delivered to g
in queue-->b
Currently token holder is : g
h got message and passed
i got message and passed
j got message and passed
a got message and passed
Token delivered to b
```

INPUT 1:

```
8
ABCDEFGH
CAEG
F
```

OUTPUT 2:

```
no of nodes:8
Currently token holder is : F
C&A sending request for token
in queue-->C,A
G got message and passed
H got message and passed
A got message and passed
B got message and passed
Token delivered to C
in queue-->A
Currently token holder is : C
E sending request for token
in queue-->A,E
```

```
D got message and passed
E got message and passed
F got message and passed
G got message and passed
H got message and passed
Token delivered to A
in queue-->E
Currently token holder is : A
G sending request for token
in queue-->E,G
B got message and passed
C got message and passed
D got message and passed
Token delivered to E
in queue-->G
Currently token holder is : E
F got message and passed
Token delivered to G
```

5) Remarks:

Complexity:

1. **Reading input from the file**:
   - The code reads the number of nodes, the order of nodes, the order of node requests, and the first token holder from a file.
   - The time complexity of reading from a file is generally considered O(n), where n is the size of the input file.

2. **Initializing the CircularQueue**:
   - The enQueue operation in the CircularQueue class is likely to have a time complexity of O(1) since it's an array-based implementation.
   - The initialization of the CircularQueue involves enqueuing all the nodes, which takes O(n) time, where n is the number of nodes.

3. **Adding node requests to the req_queue**:
   - The code adds all the node requests to the req_queue ArrayList.
   - The time complexity of adding elements to an ArrayList is amortized O(1), assuming there's enough capacity in the underlying array.

4. **sendRequest method**:
   - The sendRequest method uses a while loop to iterate through the nodes until it finds the requested node.
   - In the worst case, where the requested node is the last node in the CircularQueue, the time complexity of this method would be O(n), where n is the number of nodes.

5. **grantRequest method**:
   - The grantRequest method has a constant time complexity of O(1) since it performs simple operations like assignment and printing.

6. **Main method**:
   - The main method contains several loops and operations:
     - The first loop for sending the initial request has a time complexity of O(n), where n is the number of nodes (due to the sendRequest method).
     - The second loop for generating continuous requests has a time complexity of O(m * n), where m is the number of requests and n is the number of nodes. This is because the sendRequest method is called for each request, and each call has a time complexity of O(n).
     - The removal of elements from the req_queue ArrayList has an amortized time complexity of O(1).

Considering all the above parts, the overall time complexity of the code is **O(m * n)**, where m is the number of requests, and n is the number of nodes. This is because the dominant factor is the

nested loop in the main method, which involves calling the sendRequest method for each request, and the sendRequest method itself has a time complexity of O(n).