

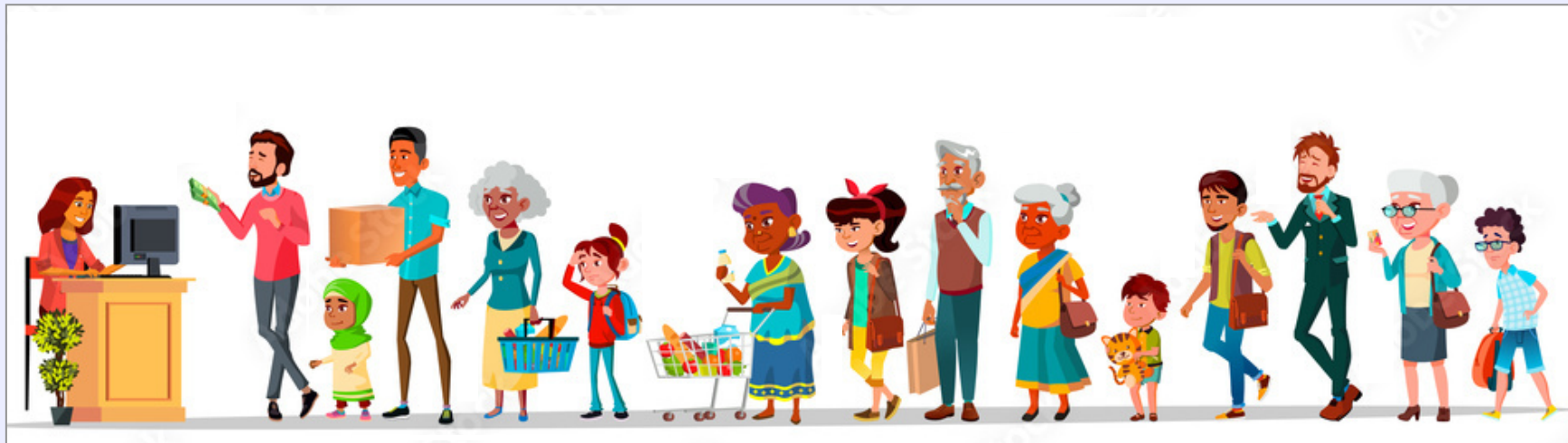
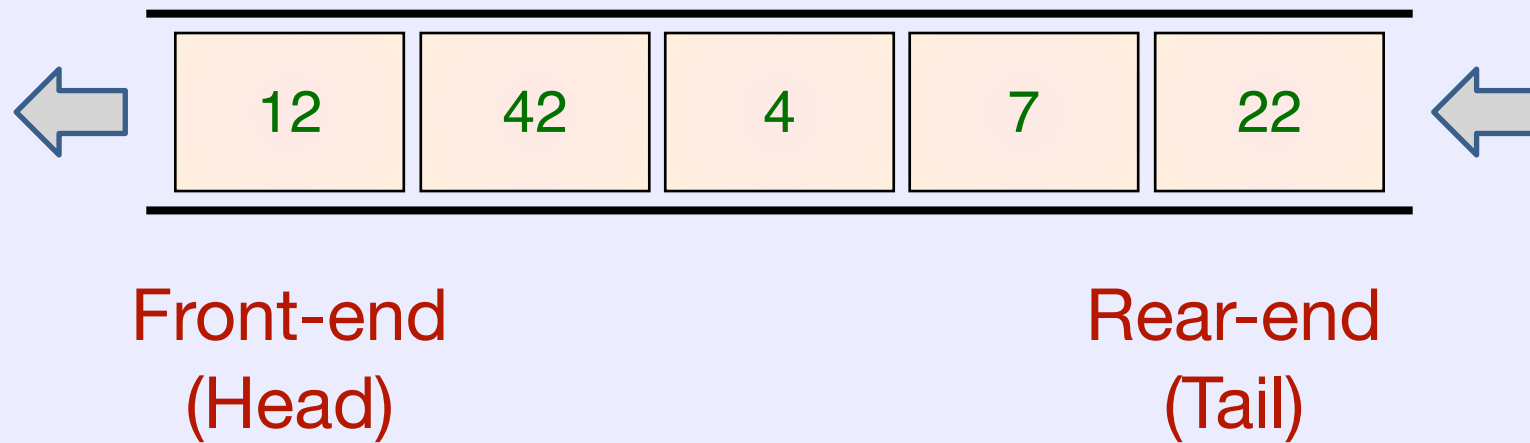
COL106: Data Structures and Algorithms

Lecture 5

Keerti Choudhary
Department of Computer Science, IIT Delhi

Queue : New Data structure

A new data structure with FIFO (First In First Out):



Applications of Queues

Direct Applications

- Waiting lines (Banks, Hospitals, Food courts)
- Access to shared resources (e.g., printer)

Indirect Applications

- Auxiliary data-structure for designing algorithms (eg. shortest route)
- Component of other data-structures

Operations on Queue

Query Operations

- **Front(Q)**: returns the front-most element of Q.
Example: If $Q = (a_1, a_2, \dots, a_n)$, then **Front(Q)** return a_1 .
- **IsEmpty(Q)**: determines if Q is empty.

Update Operations

- **Enqueue(x, Q)**: appends x at the end of Q.
Example: If $Q = (a_1, \dots, a_n)$, then after **Enqueue(x, Q)** the structure is $(a_1, \dots, a_n, x)$.
- **Dequeue(Q)**: deletes the first element of Q.
Example: If $Q = (a_1, a_2, \dots, a_n)$, then after **Dequeue(Q)** the structure is $(a_2, \dots, a_n)$.

Implementing Queue

```
public class Queue{
```

```
    class Node
```

```
    {
```

```
        int data;
```

```
        Node next;
```

```
        public Node(int data)
```

```
        {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public Node head = null;
```

```
    public Node tail = null;
```

```
    public void enqueue(int data)
```

```
    {
```

```
        Node a = new Node(data);
```

```
        if (head == null) { head, tail = a; }
```

```
        else
```

```
        {
```

```
            tail.next = a;
```

```
            tail = a;
```

```
        }
```

```
    }
```

```
    public int front()
```

```
    {
```

```
        return head.data;
```

```
    }
```

```
    public int dequeue()
```

```
    {
```

```
        int value = head.data;
```

```
        head = head.next;
```

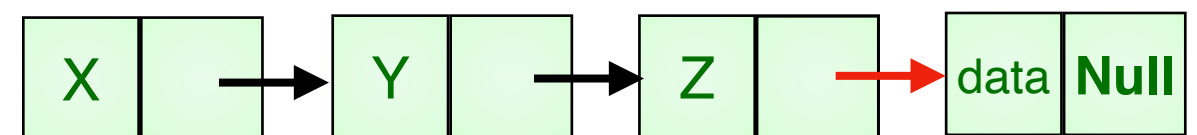
```
        if (head == null)
```

```
            tail = null;
```

```
        return value;
```

```
    }
```

```
}
```



Enqueue(data)

Implementing Queue

```
public class Queue{
```

```
    class Node
```

```
    {
```

```
        int data;
```

```
        Node next;
```

```
        public Node(int data)
```

```
        {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public Node head = null;
```

```
    public Node tail = null;
```

```
    public void enqueue(int data)
```

```
    {
```

```
        Node a = new Node(data);
```

```
        if (head == null) { head, tail = a; }
```

```
        else
```

```
        {
```

```
            tail.next = a;
```

```
            tail = a;
```

```
        }
```

```
    }
```

```
    public int front()
```

```
    {
```

```
        return head.data;
```

```
    }
```

```
    public int dequeue()
```

```
    {
```

```
        int value = head.data;
```

```
        head = head.next;
```

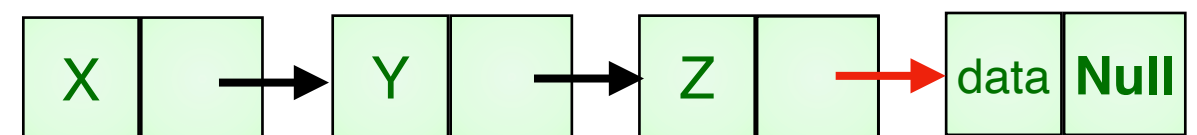
```
        if (head == null)
```

```
            tail = null;
```

```
        return value;
```

```
    }
```

```
}
```



Dequeue()

Implementing Queue

```
public class Queue{
```

```
    class Node
```

```
    {
```

```
        int data;
```

```
        Node next;
```

```
        public Node(int data)
```

```
        {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public Node head = null;
```

```
    public Node tail = null;
```

```
    public void enqueue(int data)
```

```
    {
```

```
        Node a = new Node(data);
```

```
        if (head == null) { head, tail = a; }
```

```
        else
```

```
        {
```

```
            tail.next = a;
```

```
            tail = a;
```

```
        }
```

```
    }
```

```
    public int front()
```

```
    {
```

```
        return head.data;
```

```
    }
```

```
    public int dequeue()
```

```
    {
```

```
        int value = head.data;
```

```
        head = head.next;
```

```
        if (head == null)
```

```
            tail = null;
```

```
        return value;
```

```
    }
```

```
}
```



Dequeue()

Implementing Queue

```
public class Queue{
```

```
    class Node
```

```
    {
```

```
        int data;
```

```
        Node next;
```

```
        public Node(int data)
```

```
        {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public Node head = null;
```

```
    public Node tail = null;
```

```
    public void enqueue(int data)
```

```
    {
```

```
        Node a = new Node(data);
```

```
        if (head == null) { head, tail = a; }
```

```
        else
```

```
        {
```

```
            tail.next = a;
```

```
            tail = a;
```

```
        }
```

```
    }
```

```
    public int front()
```

```
    {
```

```
        return head.data;
```

```
    }
```

```
    public int dequeue()
```

```
    {
```

```
        int value = head.data;
```

```
        head = head.next;
```

```
        if (head == null)
```

```
            tail = null;
```

```
        return value;
```

```
    }
```

```
}
```

```
Queue Q = new Queue();
```

```
for( int i=1; i<=5; i++) Q.enqueue(i);
```

```
for( int i=1; i<=5; i++)
```

```
    System.out.println(Q.dequeue());
```

```
/* Output is : 1 2 3 4 5 */
```

Implementing Queues using Arrays

```
public class Stack{
```

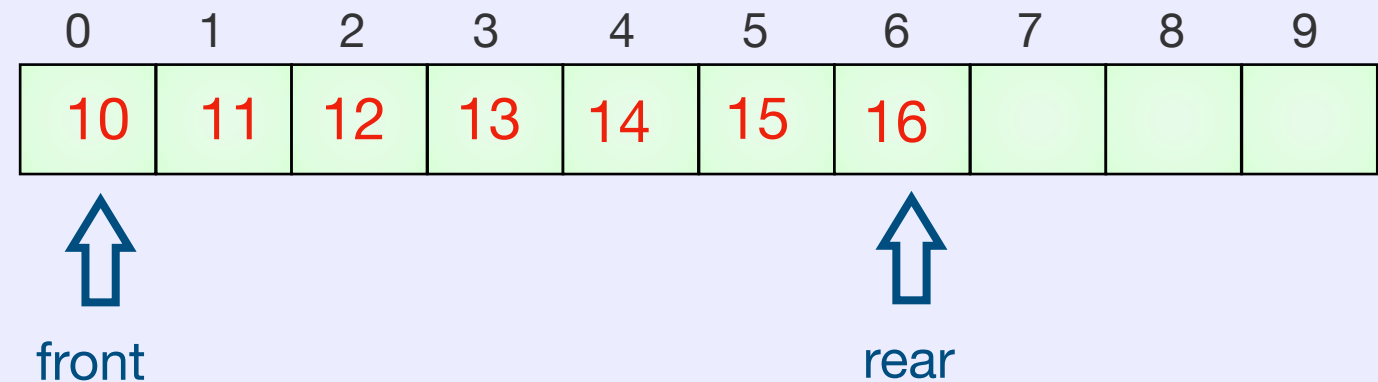
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Implementing Queues using Arrays

```
public class Stack{
```

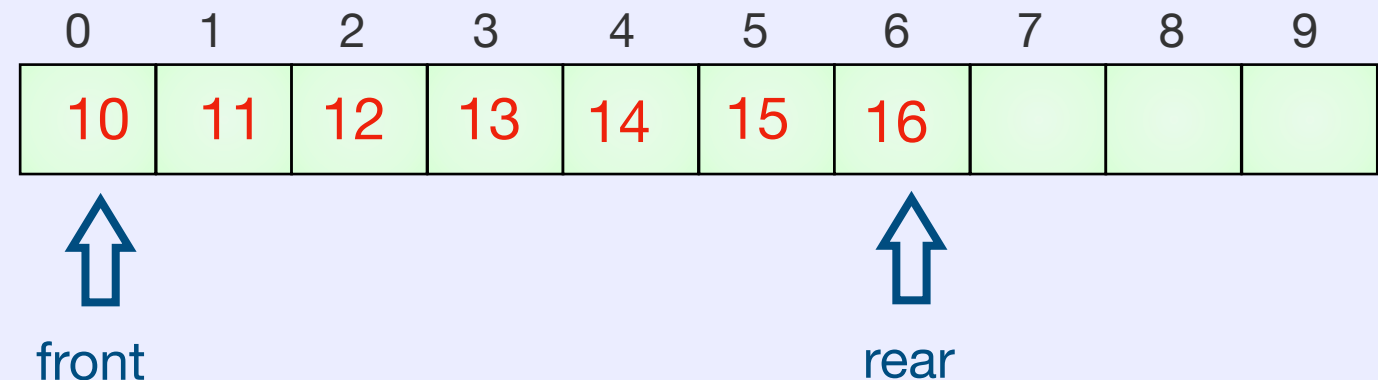
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)

Implementing Queues using Arrays

```
public class Stack{
```

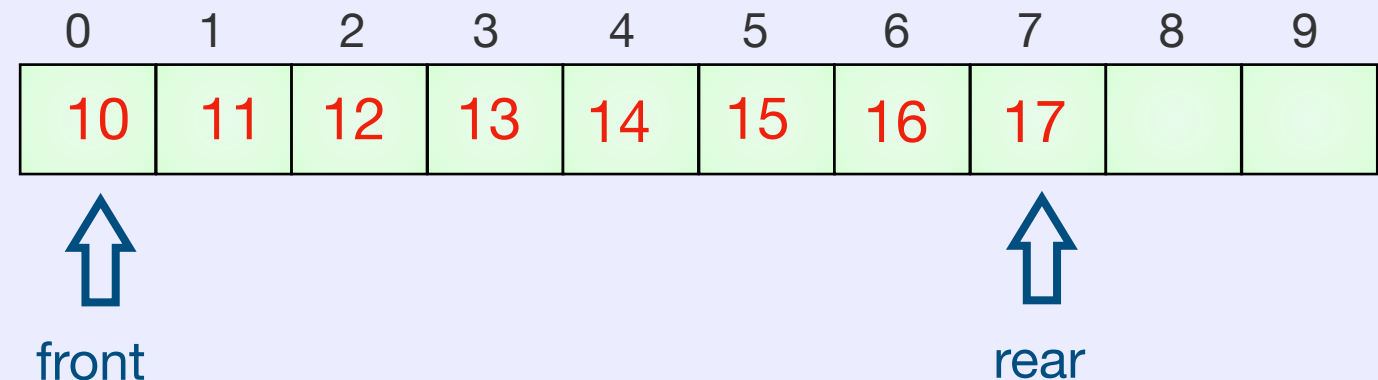
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)

Implementing Queues using Arrays

```
public class Stack{
```

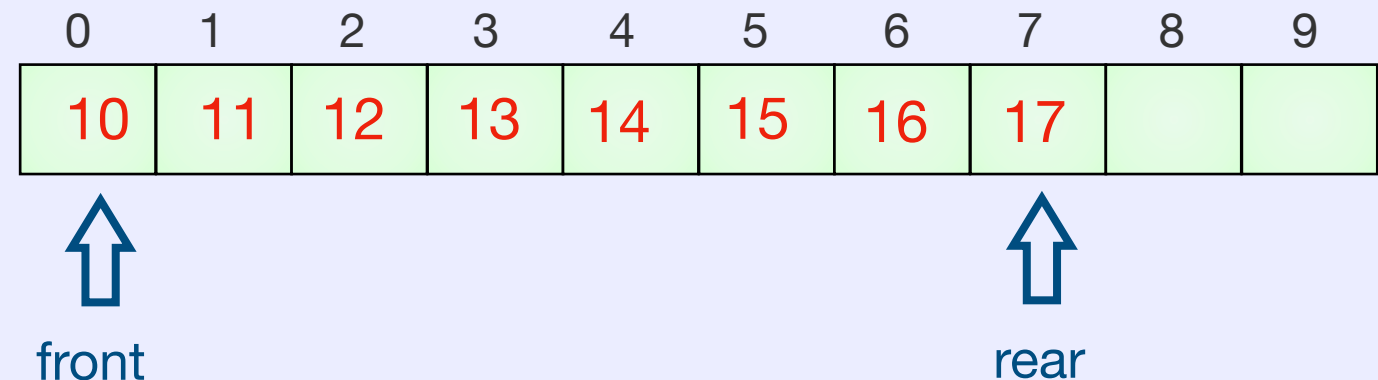
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)

Implementing Queues using Arrays

```
public class Stack{
```

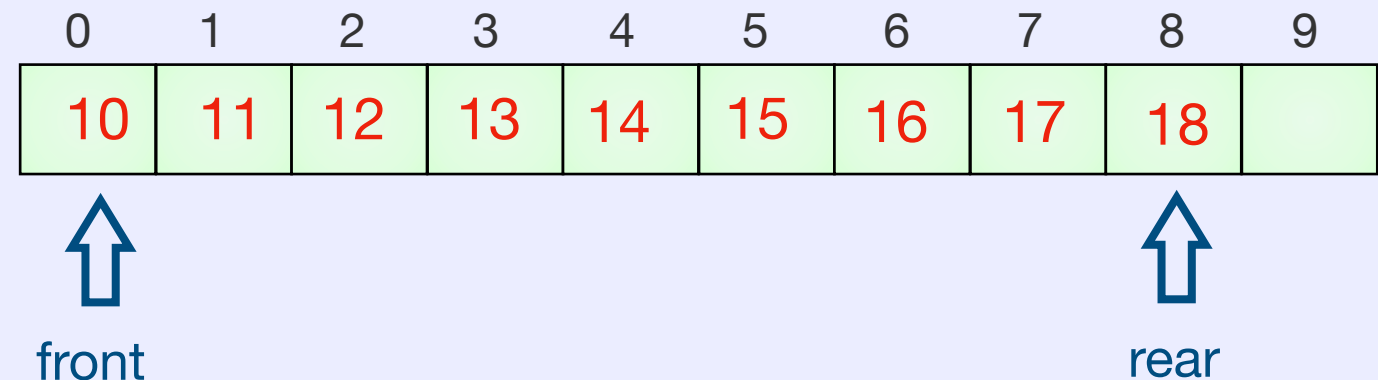
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)

Implementing Queues using Arrays

```
public class Stack{
```

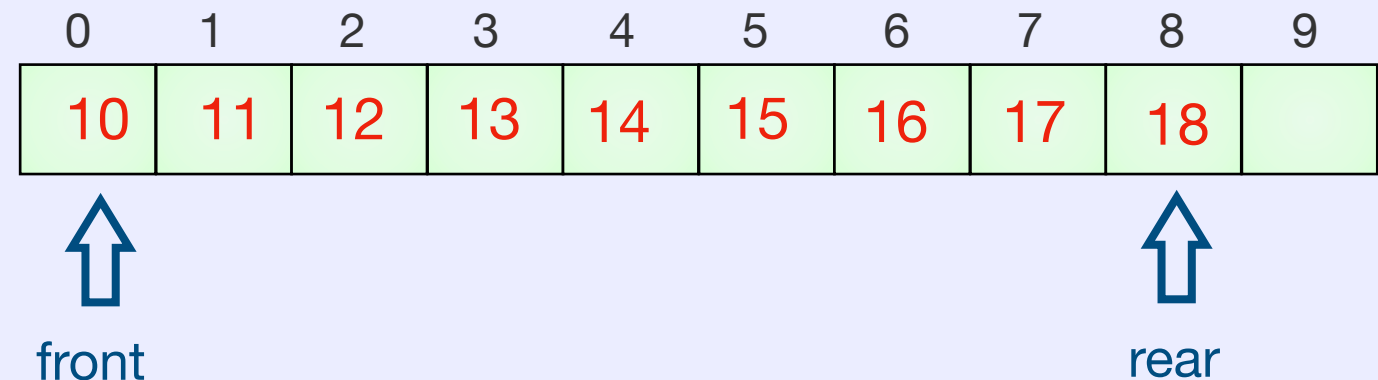
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (...) /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (...) /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()

Implementing Queues using Arrays

```
public class Stack{
```

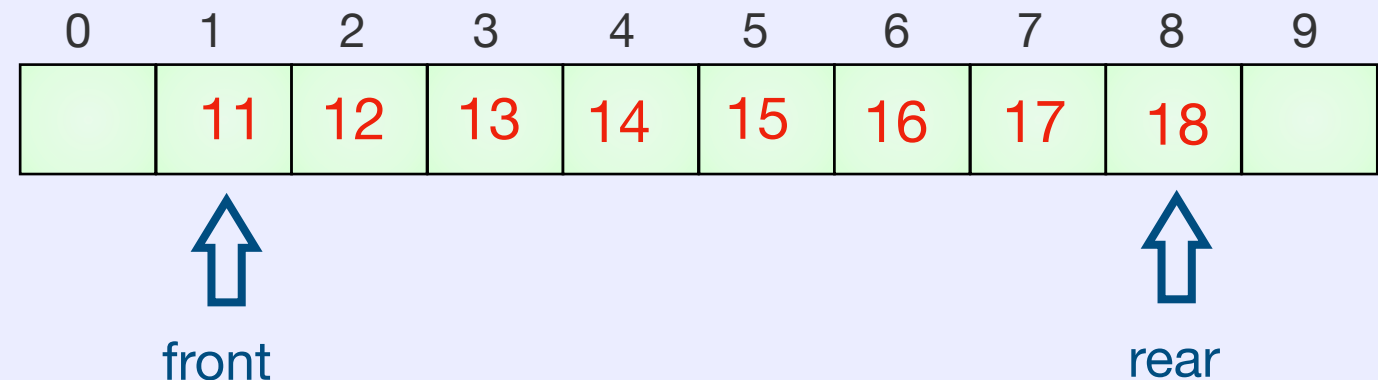
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()

Implementing Queues using Arrays

```
public class Stack{
```

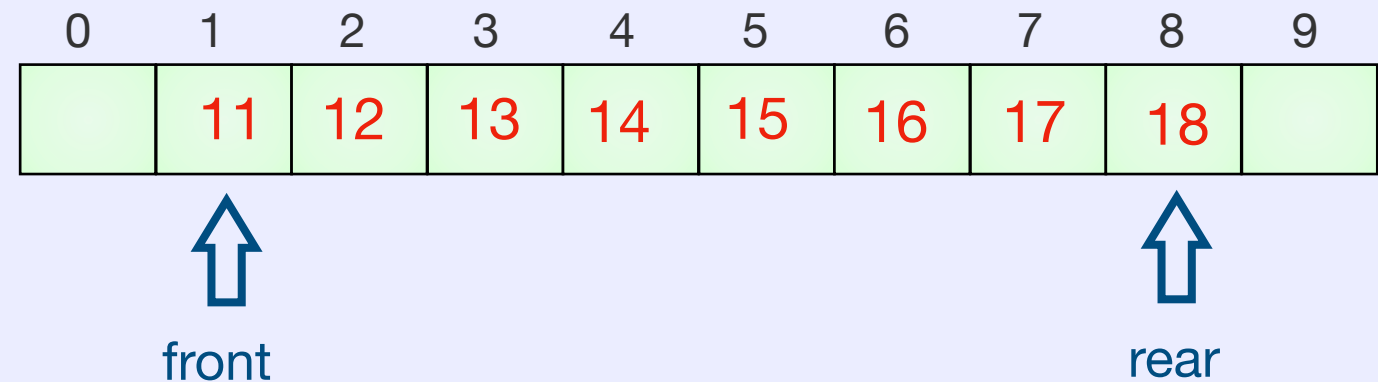
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()

Implementing Queues using Arrays

```
public class Stack{
```

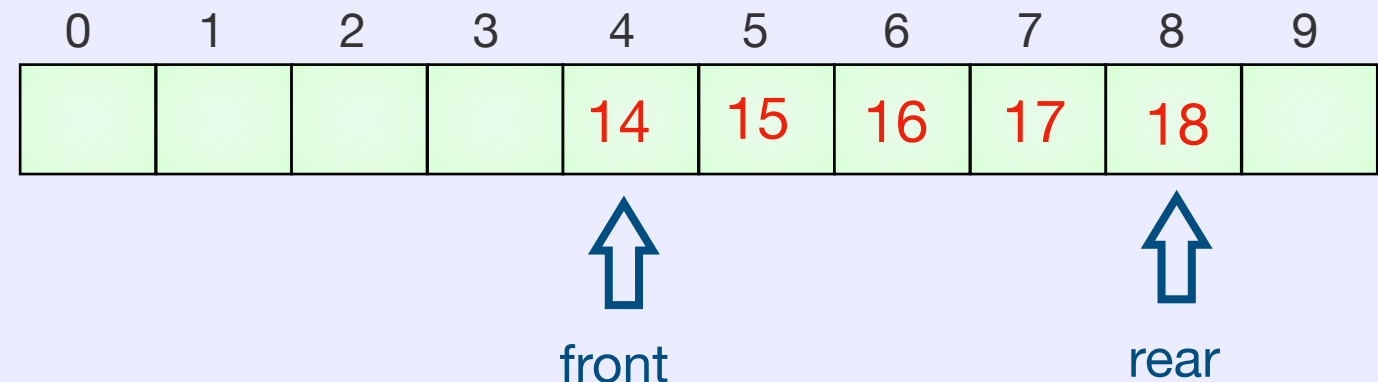
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()

Implementing Queues using Arrays

```
public class Stack{
```

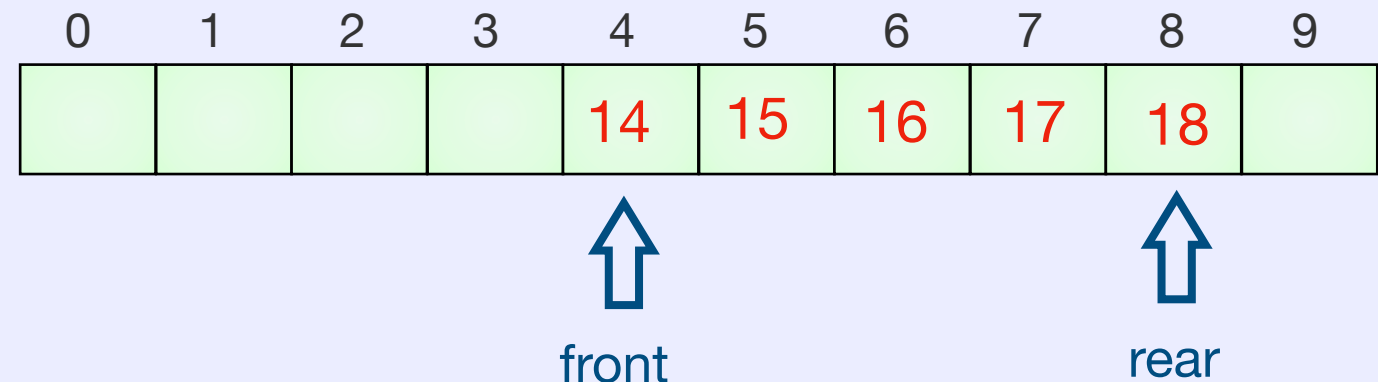
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)

Implementing Queues using Arrays

```
public class Stack{
```

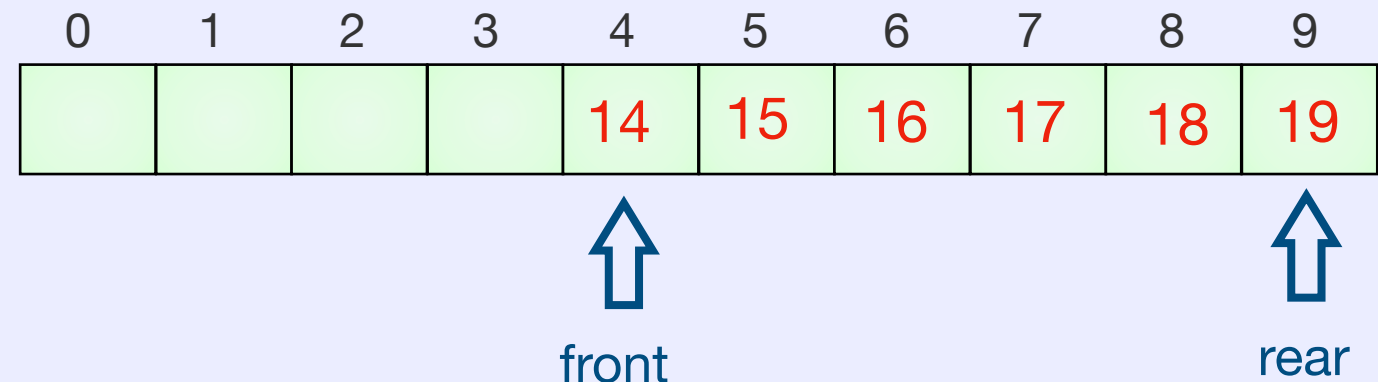
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)

Implementing Queues using Arrays

```
public class Stack{
```

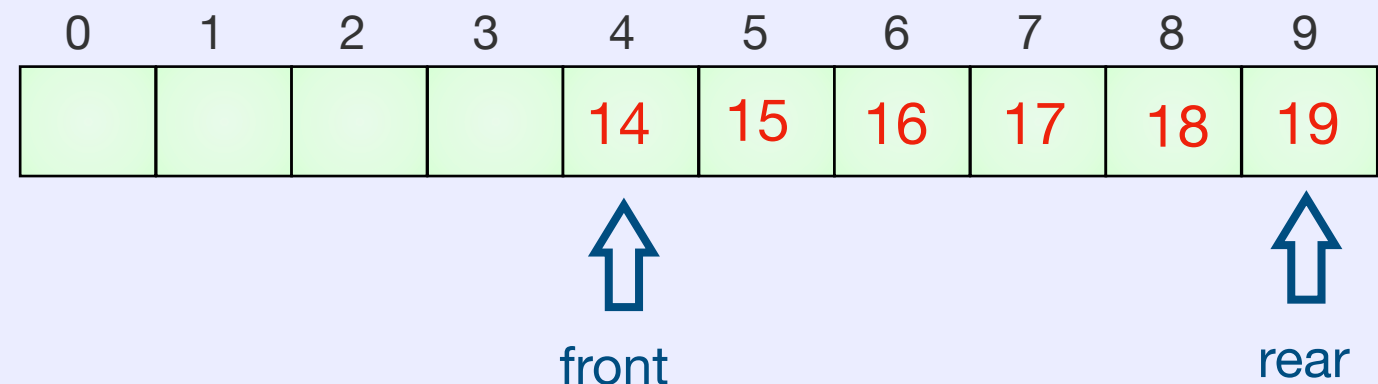
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)
Enqueue(20)

Implementing Queues using Arrays

```
public class Stack{
```

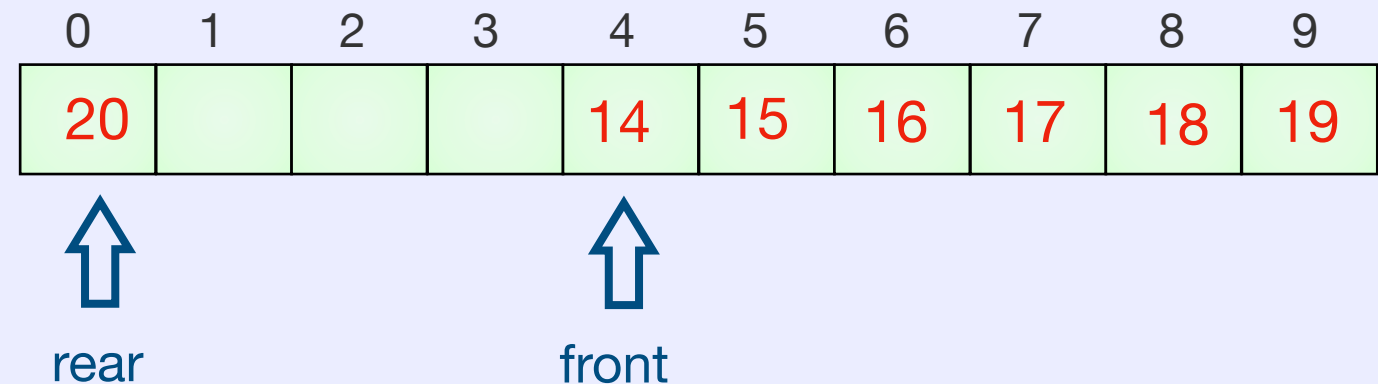
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)
Enqueue(20)

Implementing Queues using Arrays

```
public class Stack{
```

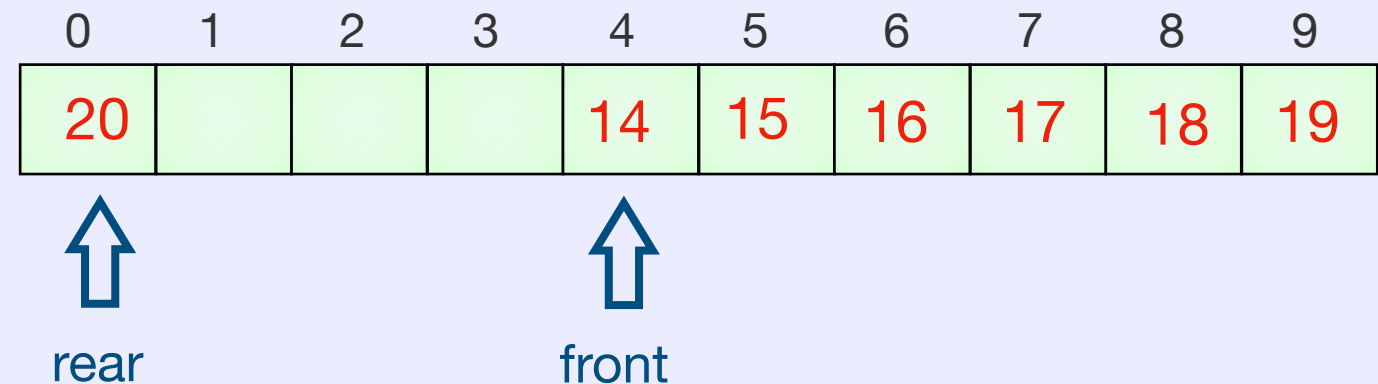
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)
Enqueue(20)
Enqueue(21)

Implementing Queues using Arrays

```
public class Stack{
```

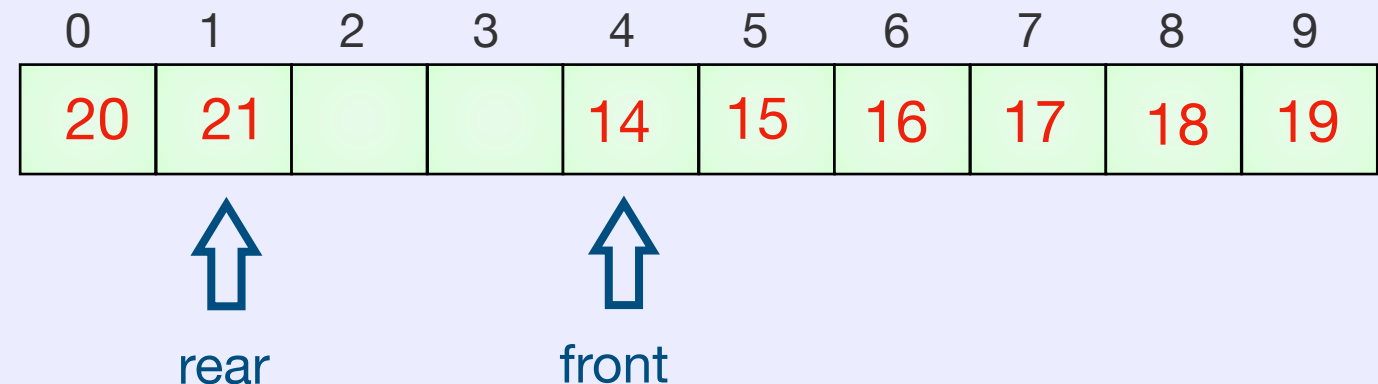
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)
Enqueue(20)
Enqueue(21)

Implementing Queues using Arrays

```
public class Stack{
```

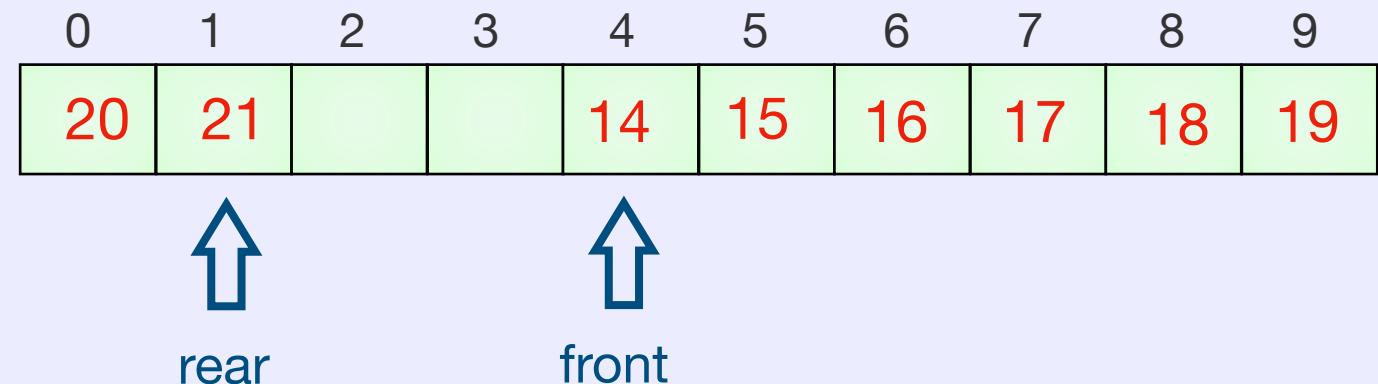
```
    private int N=10;  
    private int[] X=new int[N];  
    private int front= 0, rear = -1;
```

```
    public void enqueue(int data)  
    {  
        if (....)    /* size <= N*/  
        {  
            rear=(rear+1)%N;  
            X[rear]=data;  
        }  
    }
```

```
    public int front()  
    { return X[front]; }
```

```
    public int dequeue()  
    {  
        if (....)    /* size >= 0*/  
        {  
            int value = X[front];  
            front = (front+1)%N;  
            return value;  
        }  
    }
```

```
}
```



Enqueue(17)
Enqueue(18)
Dequeue()
Dequeue()
Dequeue()
Dequeue()
Enqueue(19)
Enqueue(20)
Enqueue(21)

Limitation: There is a bound (N) on the size of queue.

Queue from Dynamic Arrays

Doubling Approach:

Make a new array of **double** size (and duplicate the contents) whenever the array is full.

Halving Approach:

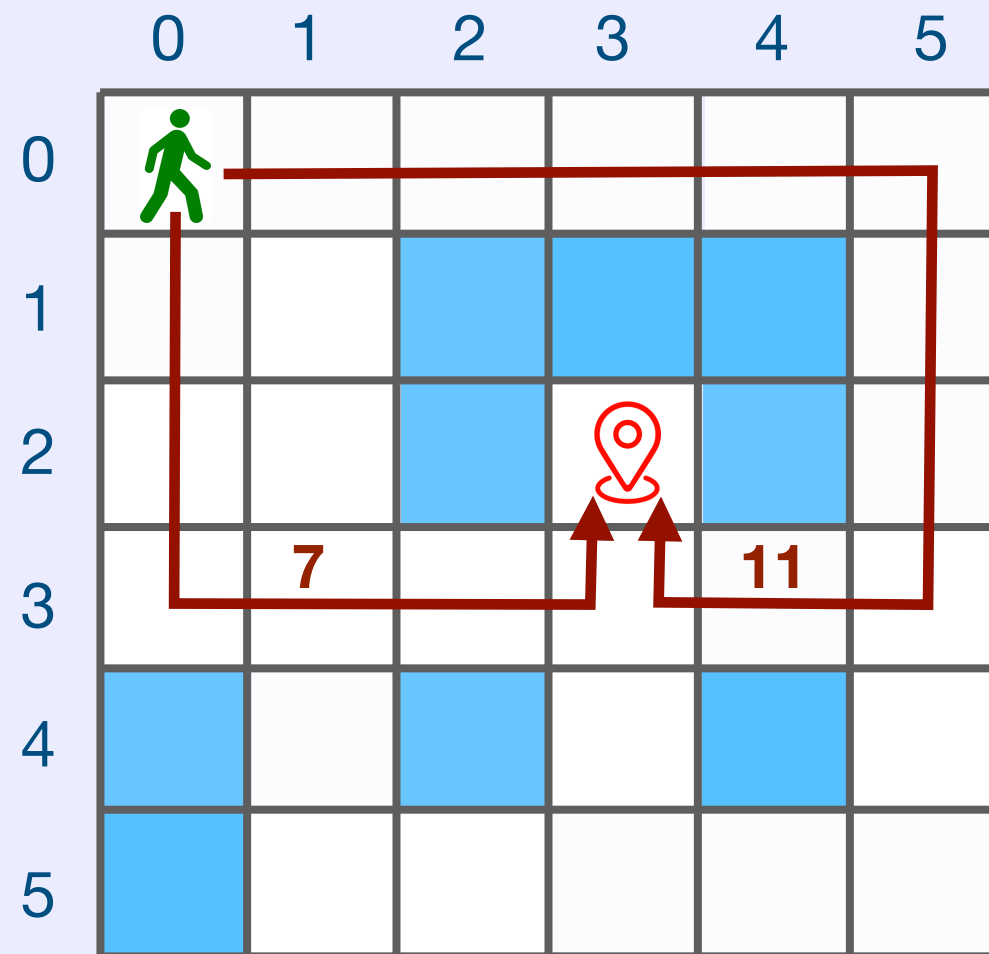
Make a new array of **half** size (and duplicate the contents) whenever the array is 3/4th empty.

Claim (Homework): The amortized (average) time-complexity of “enqueue” and “dequeue” operations is $O(1)$.

Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

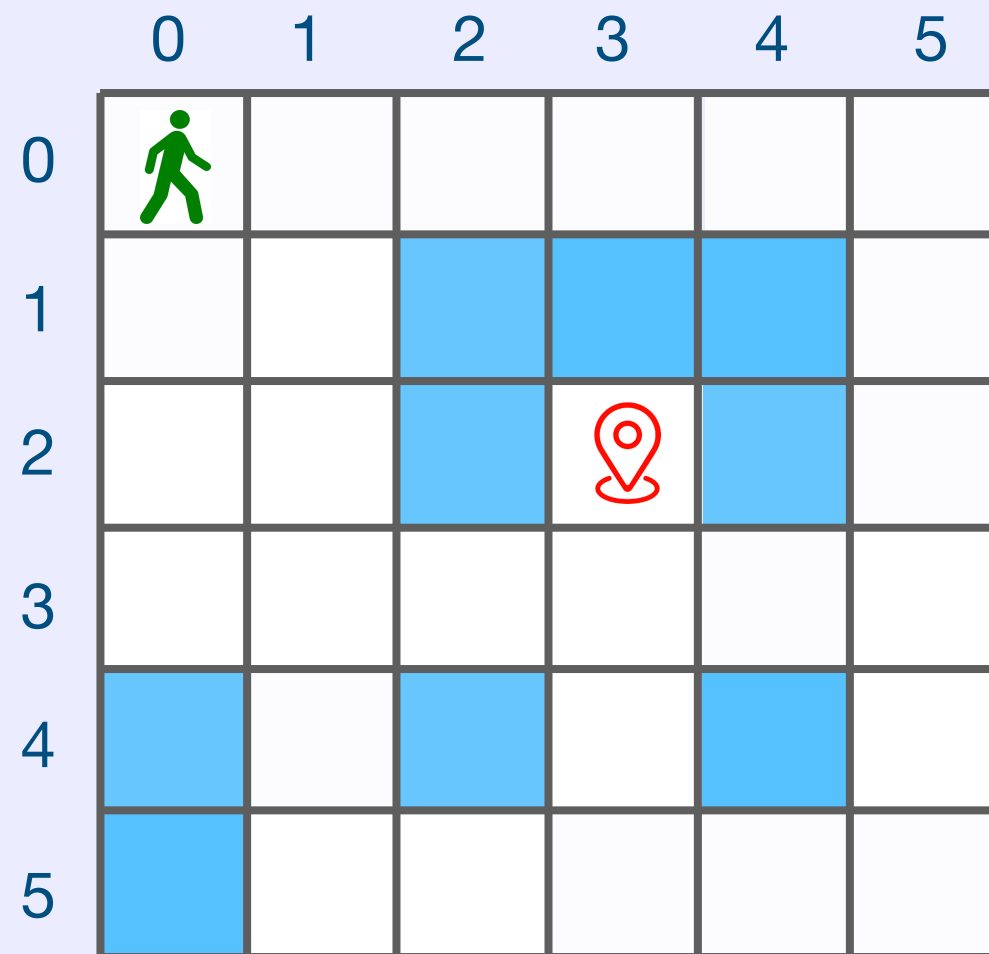
Goal: Compute shortest path to destination.



Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

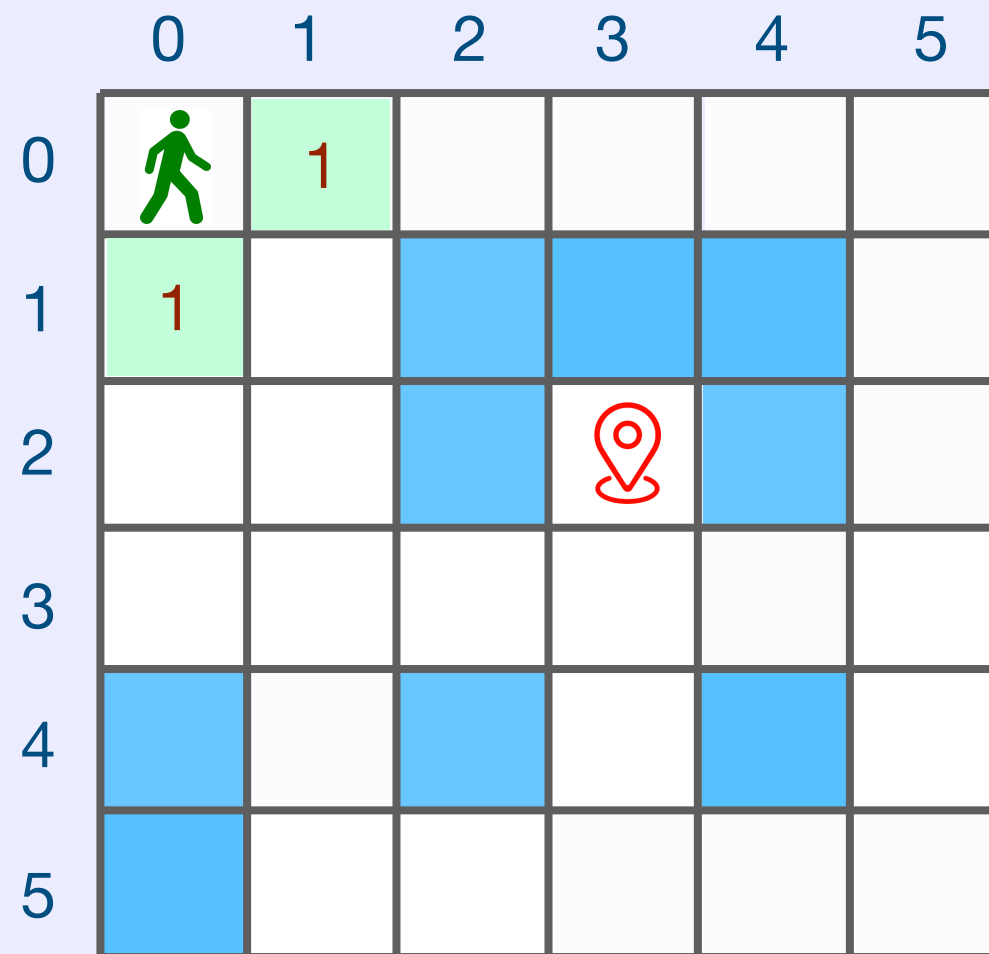
Goal: Compute shortest path to destination.



Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

Goal: Compute shortest path to destination.

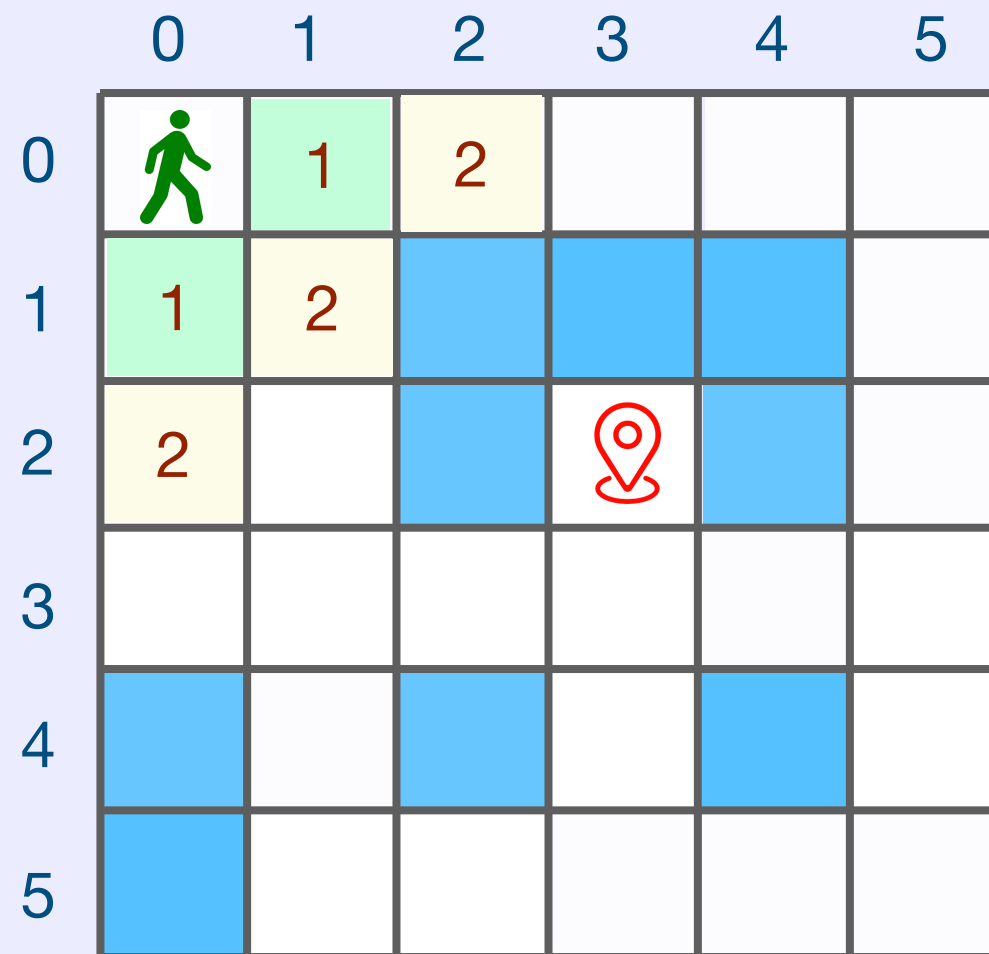


Q [0,0]

Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

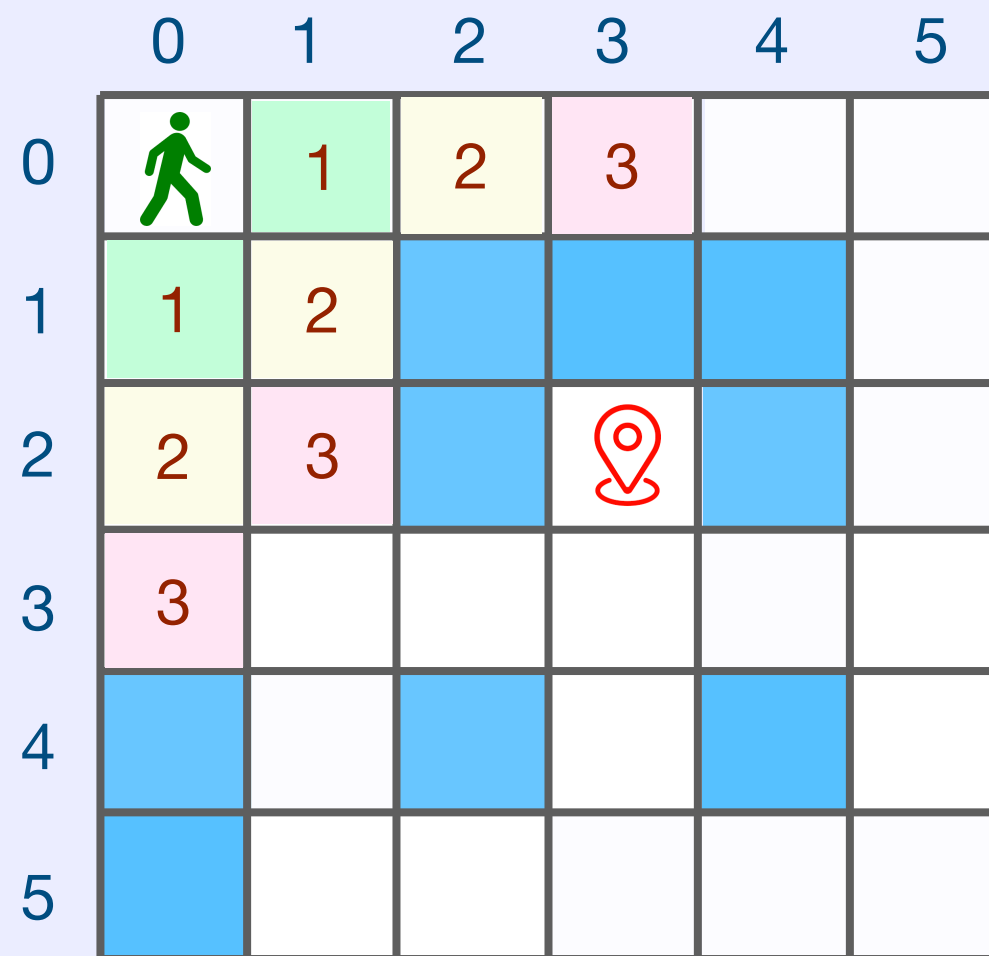
Goal: Compute shortest path to destination.



Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

Goal: Compute shortest path to destination.



Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

Goal: Compute shortest path to destination.

	0	1	2	3	4	5
0		1	2	3	4	5
1	1	2				6
2	2	3				7
3	3	4		10	9	8
4		5		9		9
5		6	7	8	9	10

```
int M[6][6];
```

```
int Queue Q;
```

```
Q.push( [0,0] );
```

```
While(Q is non)
```

Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

Goal: Compute shortest path to destination.



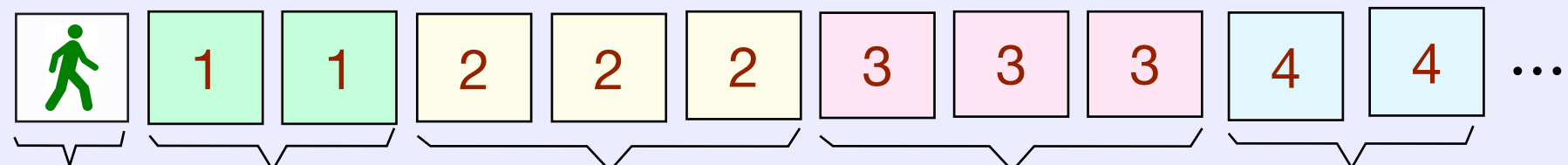
Application: Shortest Route in Maze

Given: A maze represented by a binary matrix M , starting cell, and destination cell.

Goal: Compute shortest path to destination.



Distances are
assigned in order:



Shortest Route in Maze: Approach 1

Define S_i : The set of cells having distance exactly i from $(0,0)$.

Compute($i + 1$):

for each c in S_i :

for each neighbour b of c whose distance is unassigned

 Distance(b) = Distance(c) + 1

 Add b to S_{i+1}

Compute-All-Distances(M, n):

 Distance $\leftarrow$ A 2d-array of size $n \times n$

$S_0 = \{(0,0)\}, S_1, \dots, S_{n-1} = \emptyset$

for $i = 0$ to $n - 1$: **Compute**($i + 1$)

$$\text{Neighbours}(x, y) = \{(x', y') \mid |x - x'| + |y - y'| = 1, \text{ and } M[x', y'] = 0\}$$

Shortest Route in Maze: Approach 2

Algorithm sketch:

1. Create a queue and enqueue the cell (0,0); also set its distance as zero.
2. For each cell **c** removed from queue, enqueue those neighbours **b** of **c** whose distance is not yet assigned. Set $\text{Distance}(\mathbf{b}) = \text{Distance}(\mathbf{c}) + 1$.

Distance $\leftarrow$ A 2d-array of size $n \times n$

Q $\leftarrow$ an empty queue

Distance(0,0) = 0

Enqueue((0,0))

while **Q** is *non-empty* **do**

c = Dequeue()

for each neighbour **b** of **c** whose distance is unassigned

 Distance(**b**) = Distance(**c**) + 1

 Enqueue(**b**)

return Distance(*destination-cell*)

Homework

Write complete pseudocode for computing distances in a given maze using a queue.

Also provide a correctness proof for your pseudocode.

