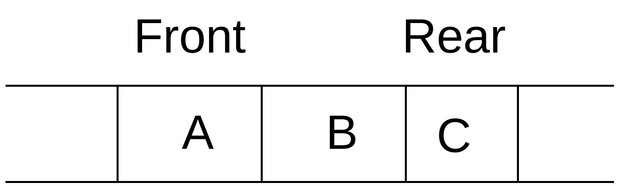
CSE 247
Data Structures

Queues

- Introduction
- Operations on Queue
- Applications of Queue
- Implementation:
  - Array based and Linkedlist based
- Types of Queue
  - Circular Queue
  - Dequeue (double ended queue)
  - Priority Queue

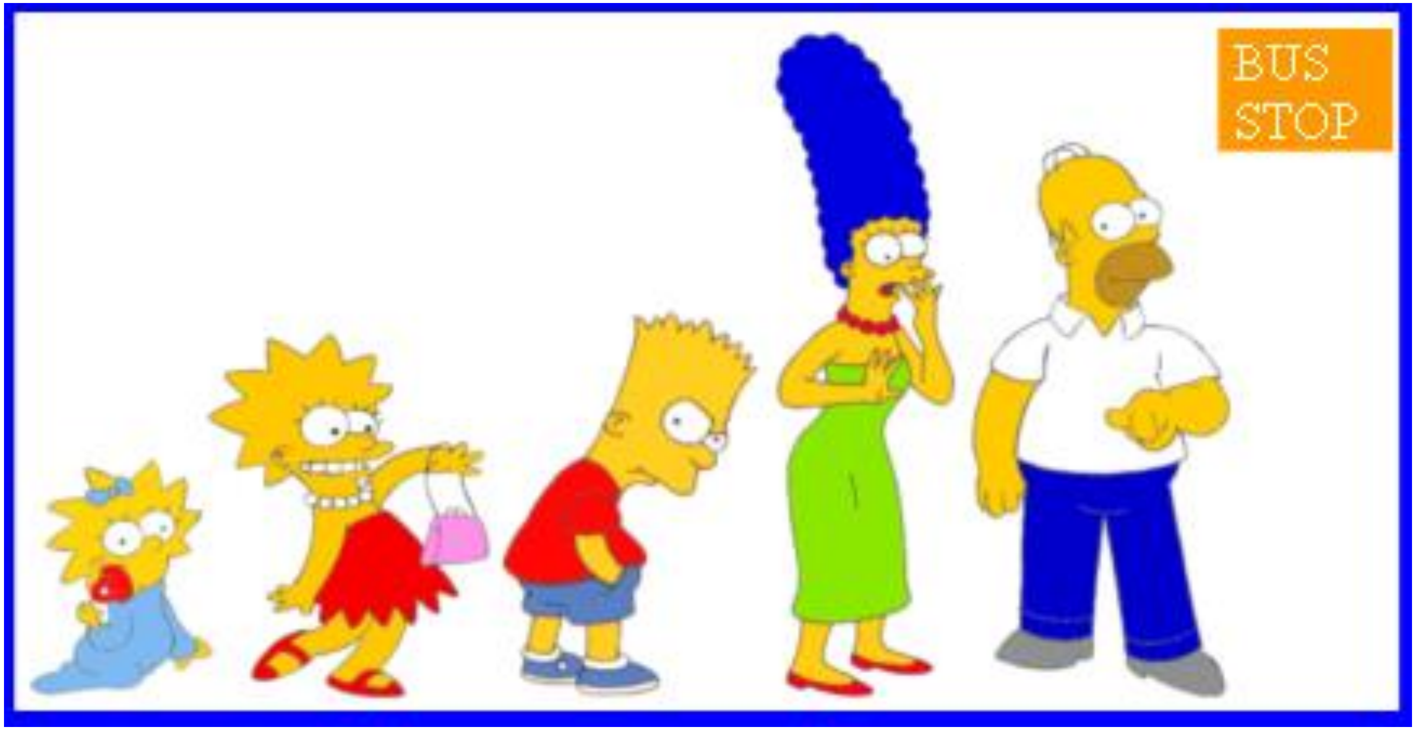
Introduction

- A queue is a first-in-first-out (FIFO) sequential data structure in which elements are added (enqueued) at one end (the back) and elements are removed (dequeued) at the other end (the front).



- **A good example of a queue we encounter queues all the time in every day life.**
- **What makes a queue a queue? What is the essence of queueness?**

Queue Concept



Queue applications

- Print server
  - maintains a queue of print jobs.
- Disk driver
  - maintains a queue of disk input/output requests.
- Scheduler (e.g., in an operating system)
  - maintains a queue of processes awaiting a slice of machine time.
- Handling requests and reservations systems

Operation on Queue using array

- Insert
- Delete
- Empty
- full

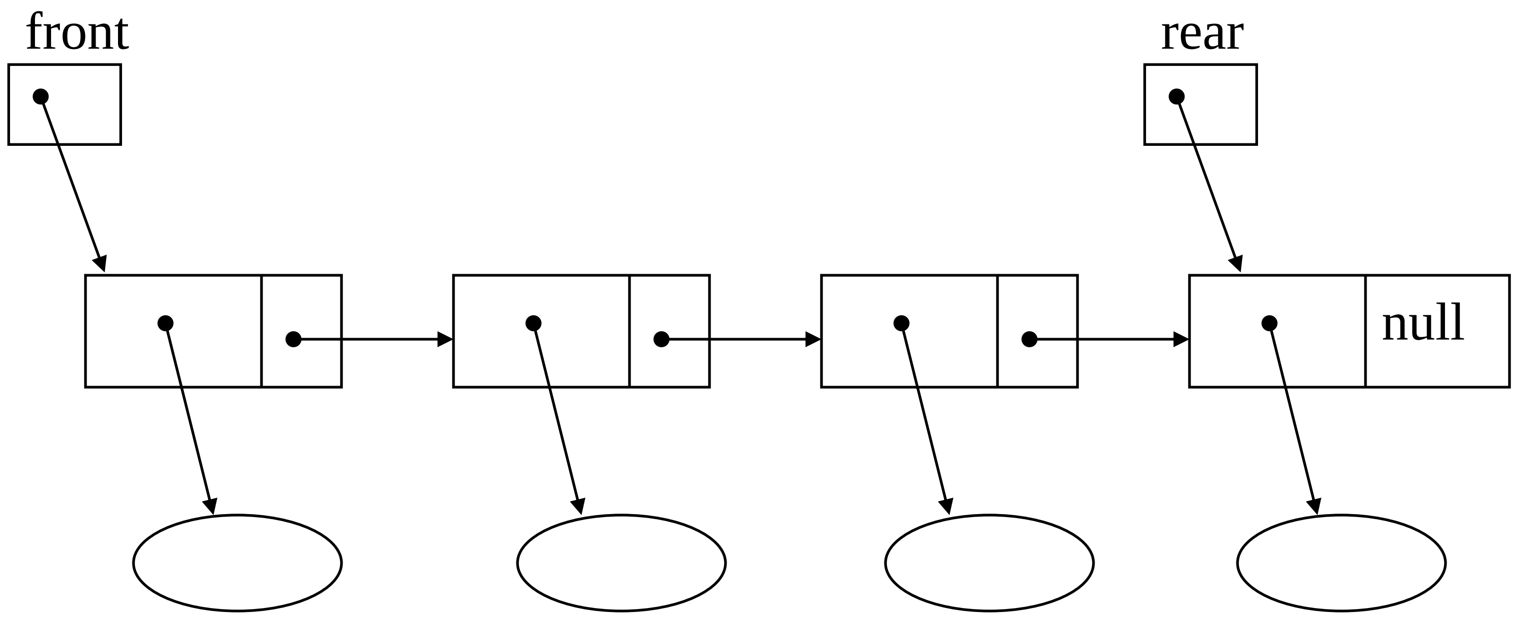
Queue

- First-in, First-out (FIFO) structure
- Operations
  - **enqueue**: insert element at rear
  - **dequeue**: remove & return front element
  - **empty**: check if the queue has no elements
  - Full: check if the queue is full

Implementation of queue

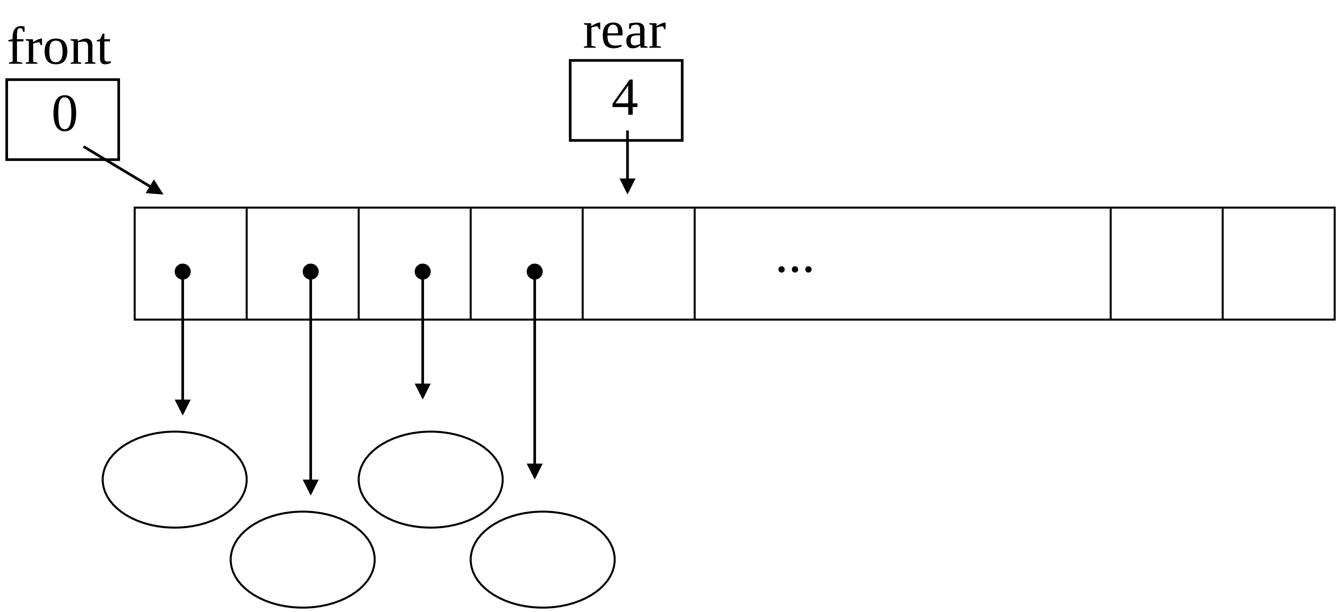
- Linked list based implementation
- Array based implementation

Linked List Implementation



Array Implementation of Queues

- An Object array and two integers
  - **front:** index of first element in queue
  - **rear:** index of first FREE element in queue



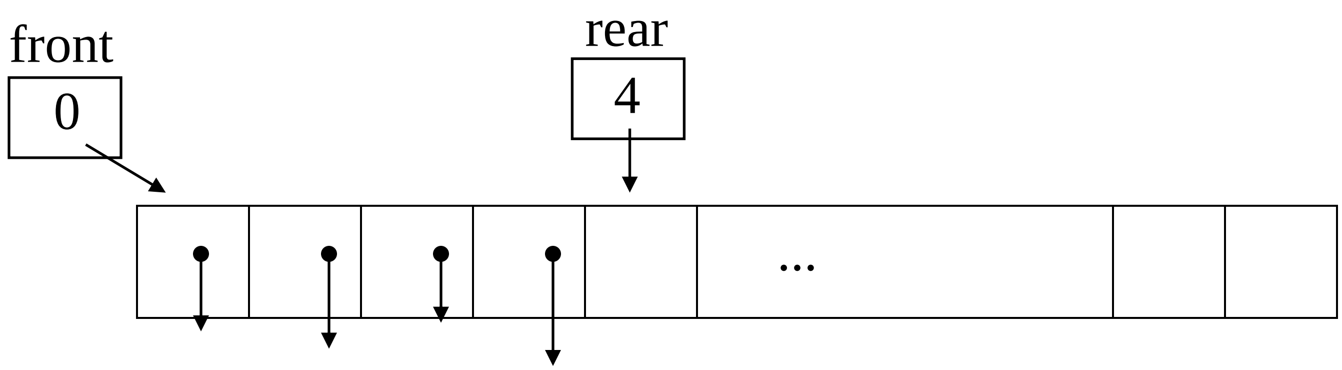
ArrayQueue

```
public class ArrayQueue
{
    int store[];
    int front, rear;
    static final int MAX = 100;

}
```

Check for Empty and Enqueue

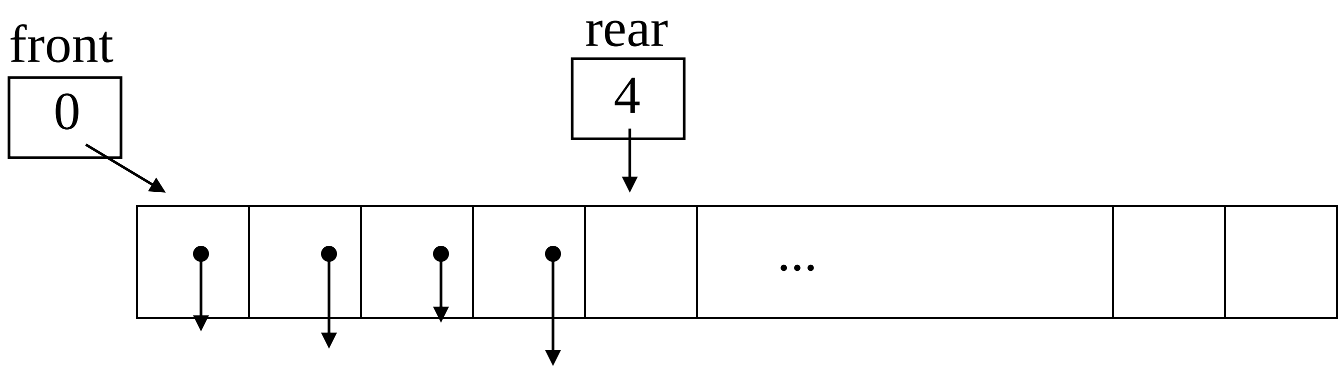
```
public class ArrayQueue
{
   public boolean empty()
    {
          return ( front == rear );
    }
    public void enqueue( int d )
    {
          if ( rear < MAX )
          store[rear++] = d;
    }
}
```



Dequeue Operation

```
public class ArrayQueue
{
  public int dequeue() throws Exception
   {
      if ( empty() )
        throw new Exception();
      else
        return store[front++];


   }
}
```



Array based queue

- Suppose many enqueue operations followed by many dequeue operations
- Result: rear approaches MAX but the queue is not really full

How to handle this problem?

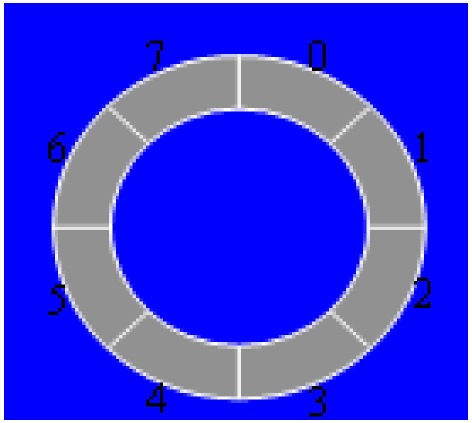
- Solution1: fixed the front position
  - Fixed front at 0^{th} index through backward movement.
  - Drawback: lots of movement in worst case
- Solution 2: Circular Array
  - allow rear (and front) to "wrap around" the array (if rear = MAX-1, incrementing rear means resetting it to 0)
  - Drawback: one slot remain unused

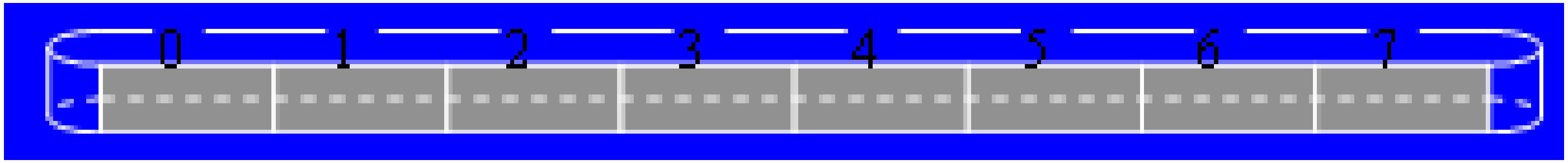
Array based queue

- Solution1: fixed the front position
  - Fixed front at 0^{th} index through backward movement.
  - Drawback: lots of movement in worst case
- Solution 2: Place indicator
  - Rather than backward movement place indicator to show it as empty cell.
  - Drawback: insertion operation change to O(n) instead of O(1).
- Solution 3: Circular Array
  - allow rear (and front) to “wrap around” the array (if rear = MAX-1, incrementing rear means resetting it to 0)
  - Drawback: one slot remain unused

Alternative

- Alternative ways of visualizing a cyclic array (length 8)





Empty operation

```
if (front == rear)
        return true
else
        return false
```

Remove operation

```
If (empty(queue)
  {
   Print("underflow); exit;
  }
front=(front+1)% (queue.length-1))

Return (queue_items[front]);
```

Insert operation

```
Public void insert(int x){
If (isFull()){
Print("queue overflow"); }
else{
rear=(rear+1)% (queue.length-1))
queue_items[rear]=x;
}
}
```

Circular Array, continued

- When is the array full?
  - Simple answer: when (rear == front)
  - Problem: this is the same condition as empty

- Solution to handle Full and empty conditions: Reserve a slot unused.
  - full: when ((rear+1) % MAX == front) (one free slot left)
  - empty: when (rear == front)

- Note: “wastes” a slot

Time complexity of queue

- O(contant)

Queue Performance

- the time needed to add or delete an item is constant and *independent of the number of items in the queue*.
- both addition and deletion operation takes constant time i-e ***O*(constant)**.
- For any given real machine+operating system+language combination, addition may take ***c1*** seconds and deletion ***c2*** seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, *etc*.
- The key point is that in Queue the time is not dependent on ***data size***
- ***O*(1)** methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

Practice Question

- Suppose a circular queue of capacity $(n - 1)$ elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are

A. Full: (REAR+1) mod n == FRONT, empty: REAR == FRONT
B. Full: (REAR+1) mod n == FRONT, empty: (FRONT+1) mod n == REAR
C. Full: REAR == FRONT, empty: (REAR+1) mod n == FRONT
D. Full: (FRONT+1) mod n == REAR, empty: REAR == FRONT

Answer is option A.