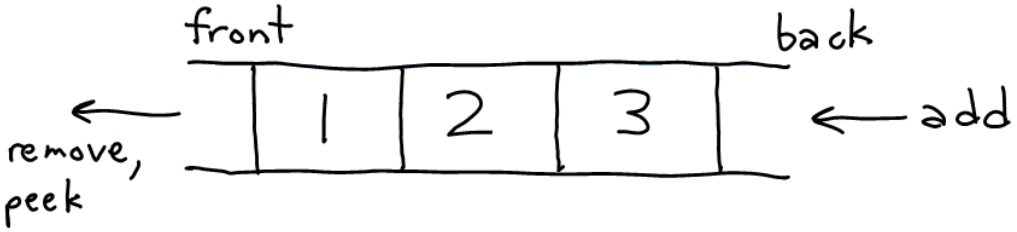
Queues

CS223: Data Structures

Queue

- **Queue**: A list with the restriction that insertions are done at one end and deletions are done at the other.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- Basic queue operations:
  - add (enqueue): Add an element to the back.
  - remove (dequeue): Remove the front element.
  - peek: Examine the element at the front.

Queues - Motivation

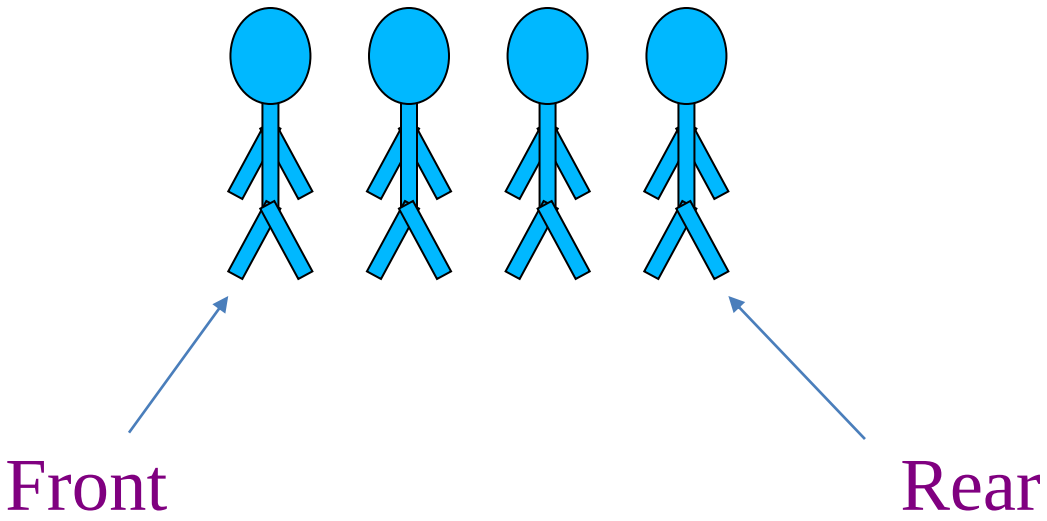
- **Operating systems:**
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send

- **Programming:**
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- **Real world examples:**
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

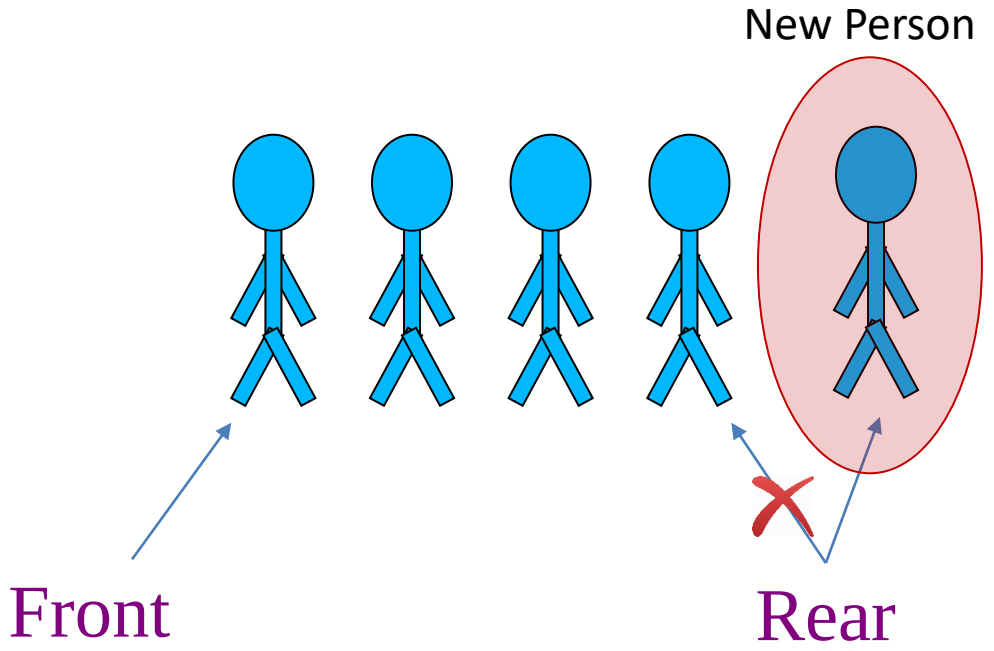
The Queue Operations

A queue is like a line of people waiting for a bank teller. The queue has a front and a rear.



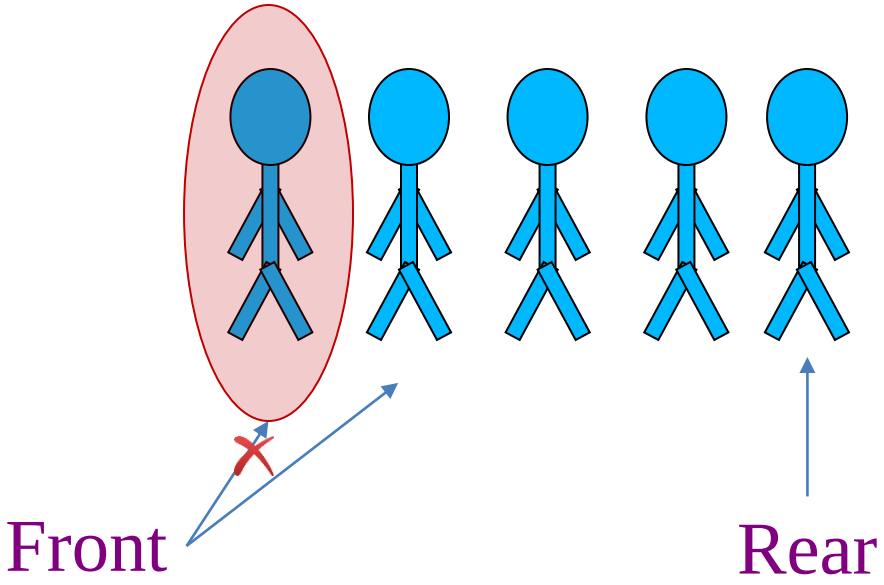
The Queue Operations

New people must enter the queue at the rear. It is usually called an enqueue operation.



The Queue Operations

When an item is taken from the queue, it always comes from the front. It is usually called a dequeue operation.



Queue – Basic Operations

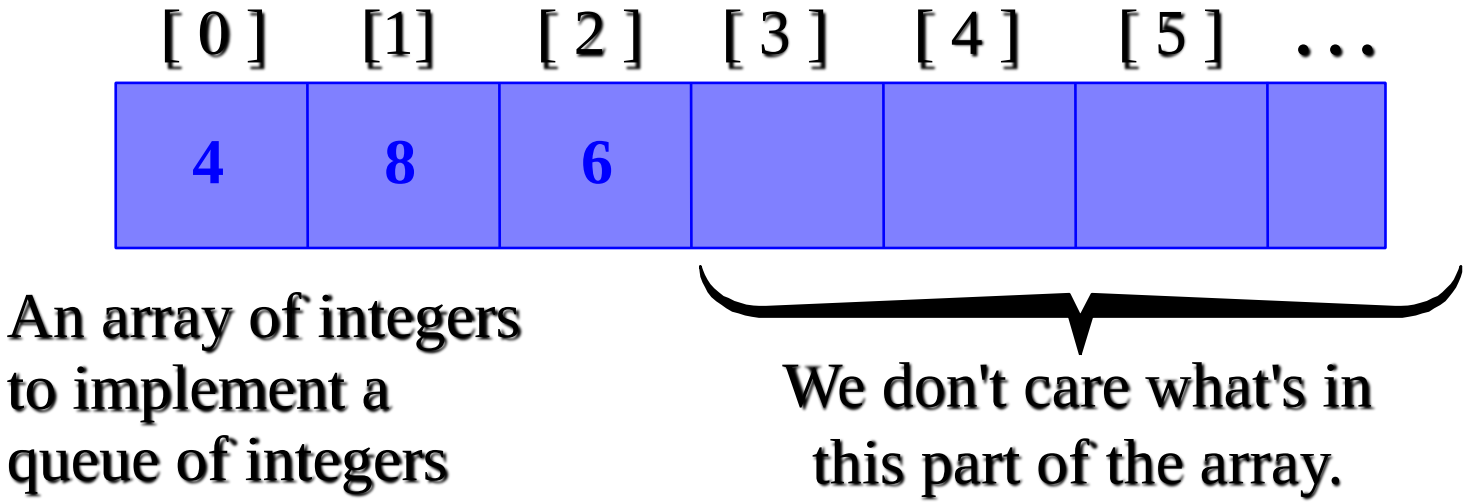
`add(`**value**`)/ enqueue(`**value**`)`	places given value at back of queue
`remove()/dequeue()`	removes value from front of queue and returns it.
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

Queue Implementation

1. Using Array
2. Using Linked List

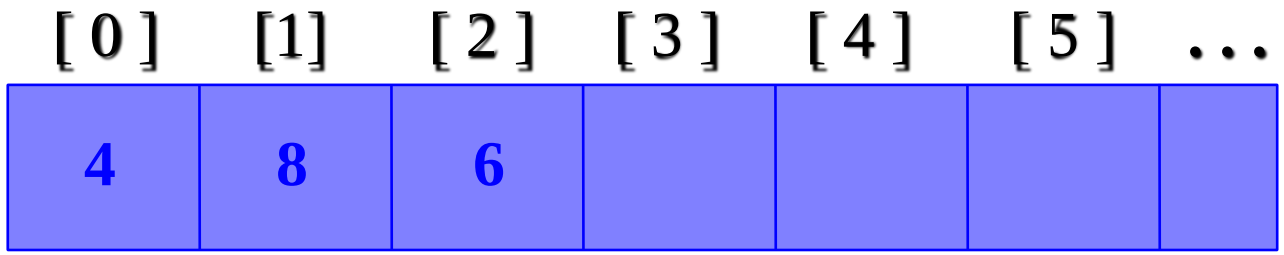
Array Implementation

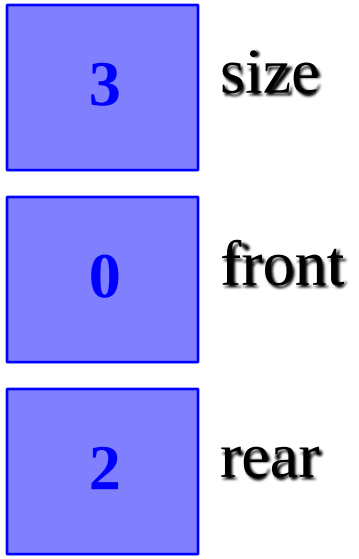
A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



Array Implementation

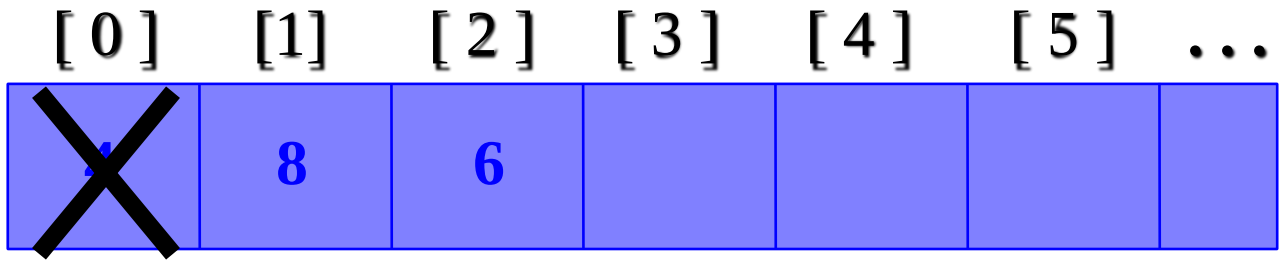
The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

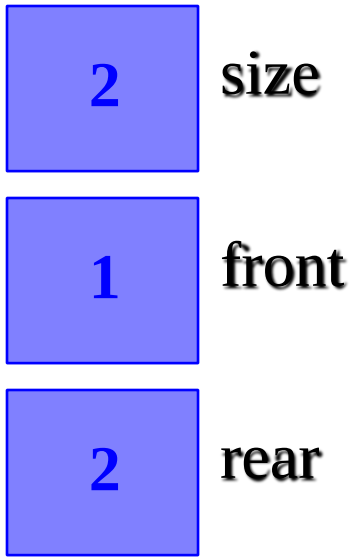




A Dequeue Operation

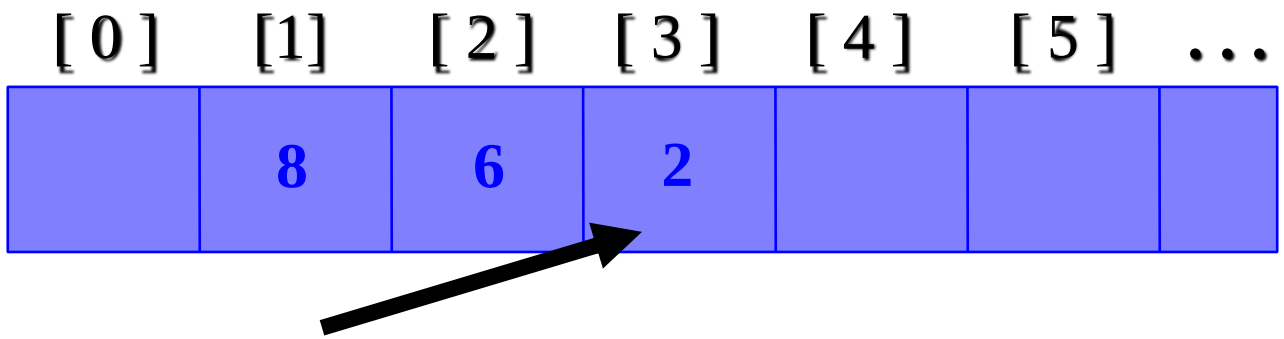
When an element leaves the queue, size is decremented, and first changes, too.

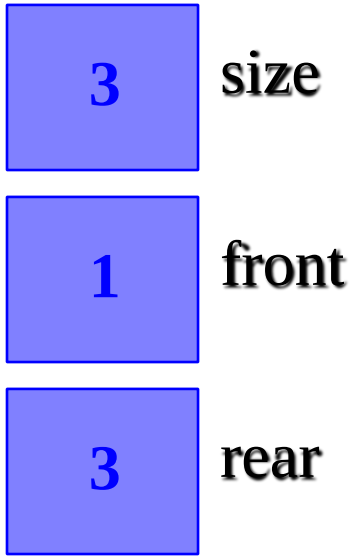




An Enqueue Operation

When an element enters the queue, size is incremented, and last changes, too.





Queue – Array

Initialize front and rear to -1
MAXSIZE is array size

Queue – Array: Peek

```
procedure peek
  return queue[front]
```

Queue – Array: enQueue Operation

```
procedure enqueue(data)

  if queue is full
     return overflow

 if front is equal to -1
     front = 0
     rear = 0
else
     rear = rear + 1

queue[rear] = data
```

Queue – Array: IsFull

```
procedure isfull

  if rear equals to MAXSIZE - 1
    return true
  else
    return false
```

Queue – Array: deQueue operation

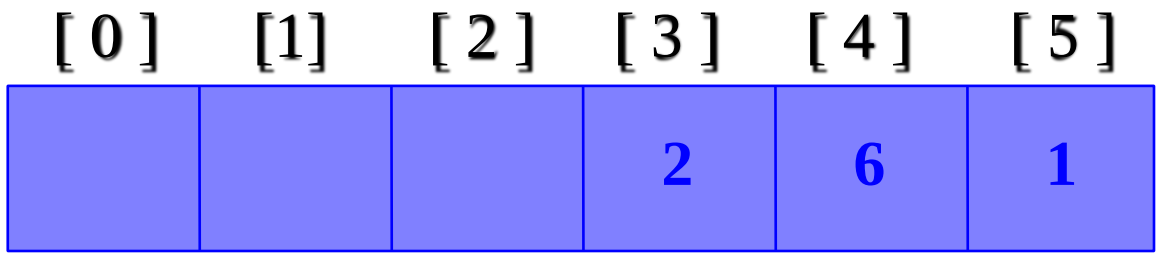
```
procedure dequeue

   if queue is empty
      return underflow
   data = queue[front]

   if front is equal to rear
       rear = -1
       front = -1
  else
       front = front + 1

return data
```

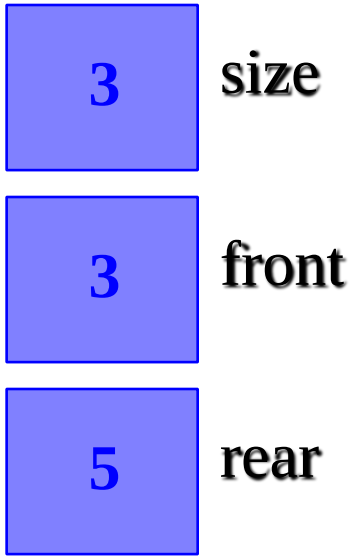
Queue – Array: IsEmpty

```
procedure isempty

  if front is less than 0
    return true
  else
    return false
```

At the End of the Array

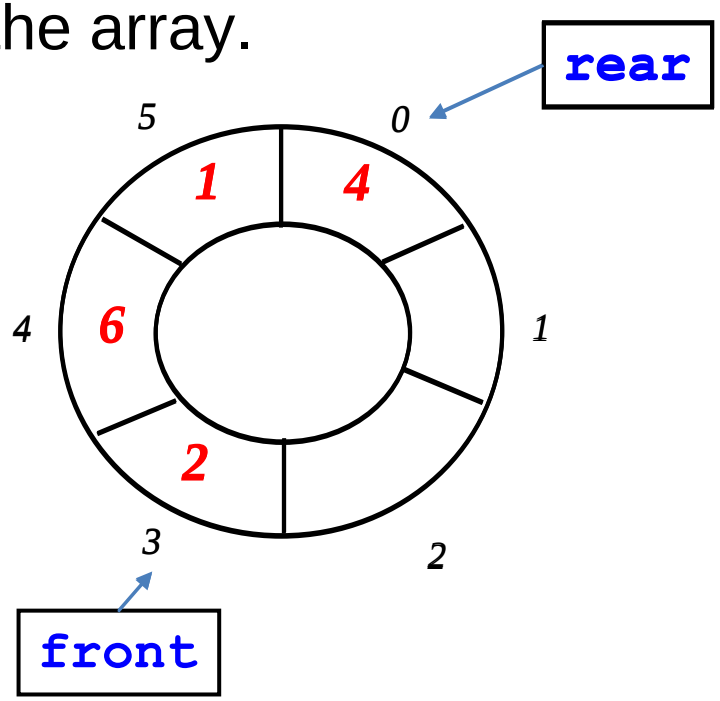
There is special behaviour at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

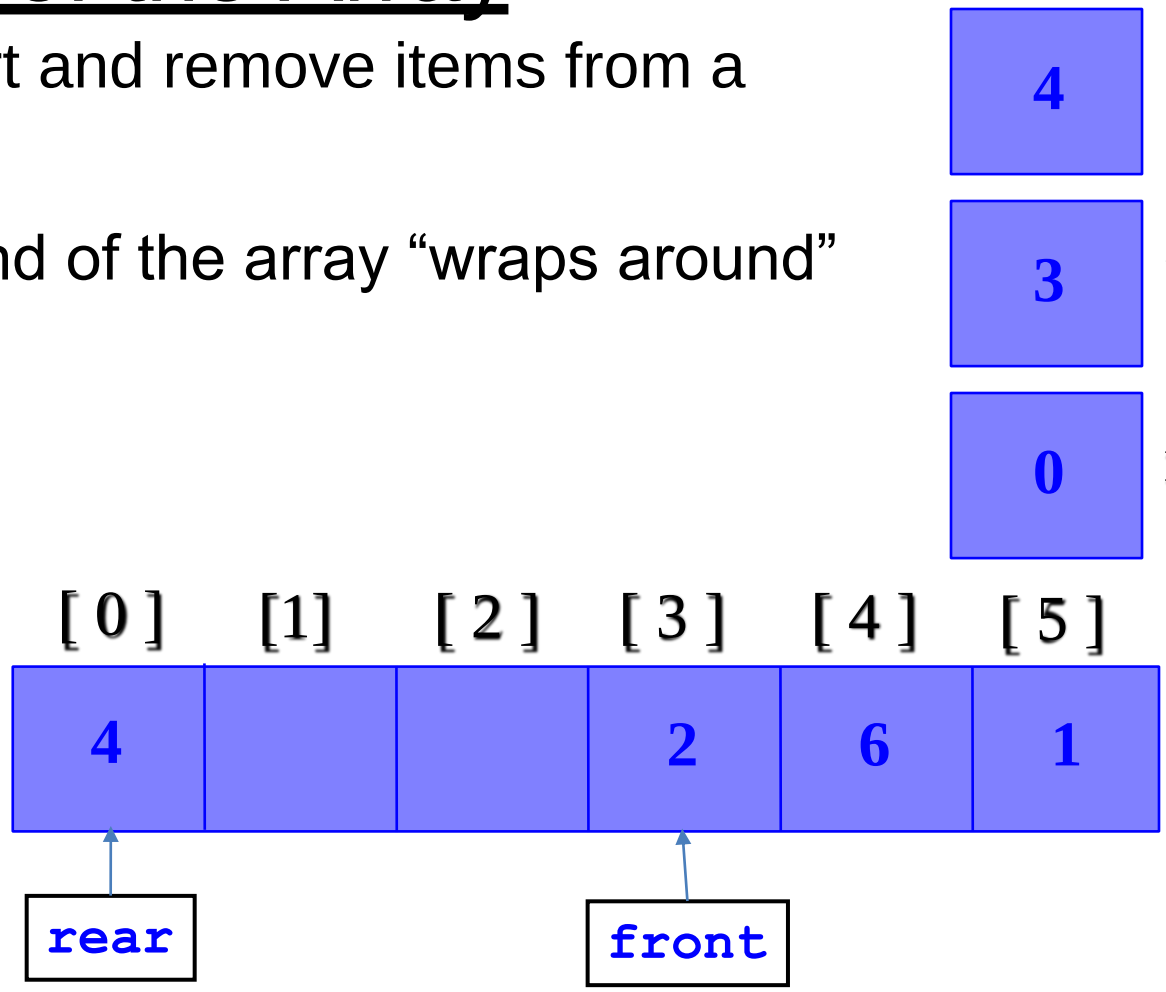




At the End of the Array

- **Neat trick**: use a ***circular array*** to insert and remove items from a queue in constant time.
- The idea of a circular array is that the end of the array "wraps around" to the start of the array.

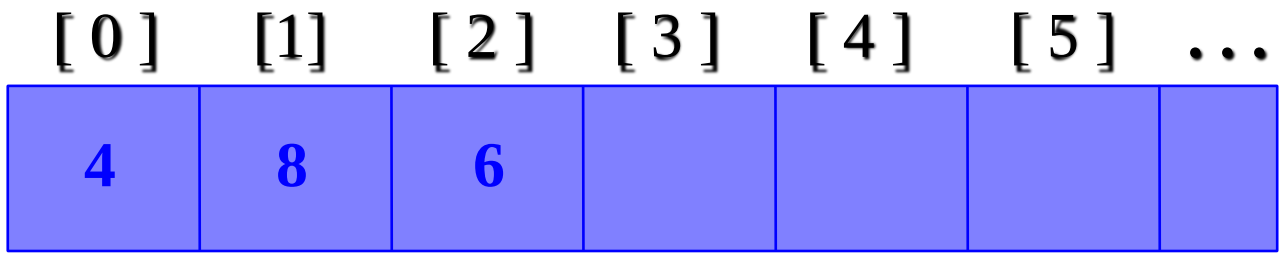


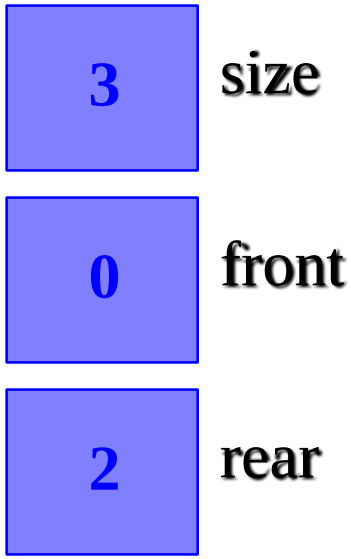


Circular view of arrays.

Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Special behaviour is needed when the rear reaches the end of the array.





Queue – Array: enQueue Operation

```
procedure enqueue(data)

  if queue is full
       return overflow

  if front is equal to -1
     front = 0
     rear = 0
  else
     rear = (rear + 1) % MAXSIZE

  queue[rear] = data
```

Queue – Array: IsFull

```
procedure isfull

  if (rear+1)%MAXSIZE equal to front
    return true
  else
    return false
```

Queue – Array: deQueue operation

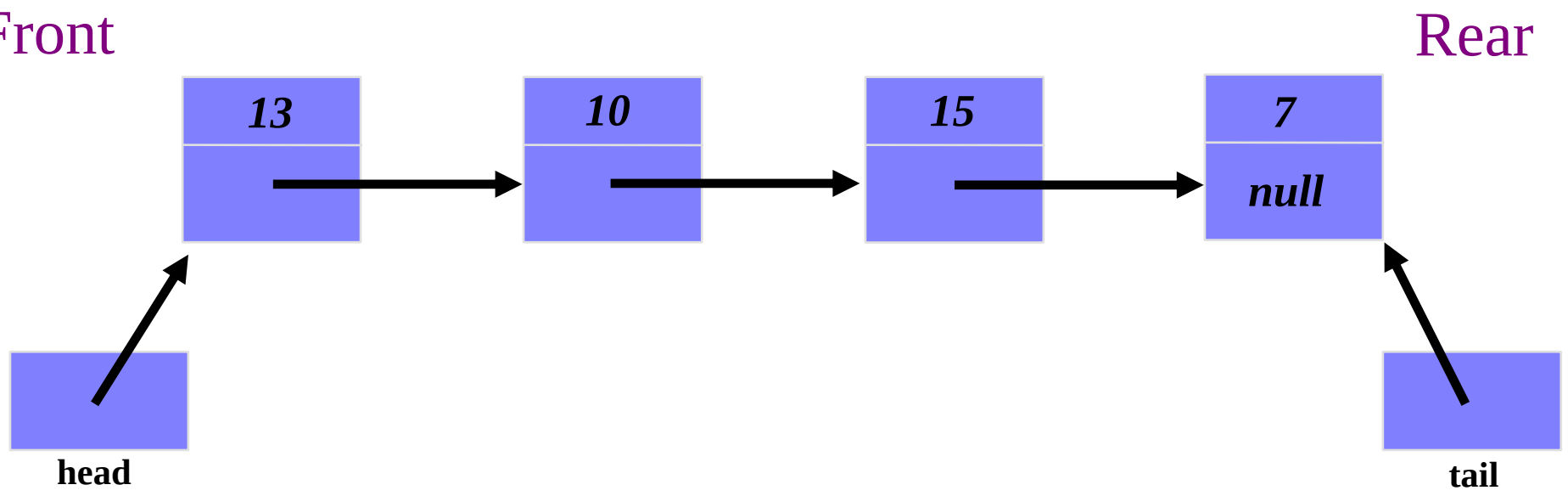
```
procedure dequeue

  if queue is empty
        return underflow

  data = queue[front]

   if front is equal to rear
      rear = -1
      front = -1
  else
     front = (front + 1)  %  MAXSIZE

  return data
```

Queue Implementation

1. Using Array
2. Using Linked List

Linked List Implementation

A queue can also be implemented with a linked list with both a head and a tail pointer.



Queue – Linked List Implementation: enQueue Operation

```
procedure enqueue(data)

    Allocate the space for the new node PTR
    SET PTR -> DATA = data
    SET PTR-> NEXT = NULL

    if FRONT equal to NULL
       SET FRONT = REAR = PTR
    ELSE
       SET REAR -> NEXT = PTR
       SET REAR = PTR
```

Queue – Linked List Implementation: deQueue Operation

```
procedure deQueue

    if FRONT = NULL
      Write " Underflow "
    else
      SET PTR = FRONT
      SET FRONT = FRONT -> NEXT
      delete PTR
```

Queue: Circular Array vs. Linked List

Circular Array

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast

Linked List

- Always just enough space
- But more space per element
- Operations very simple / fast