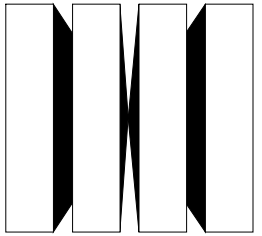


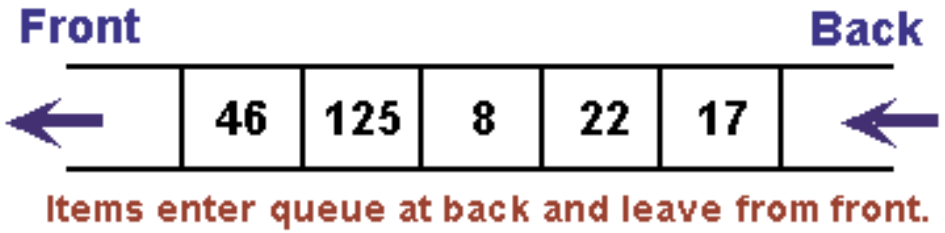
Chapter Five

Queue

Queue

Abstract data type: Queue

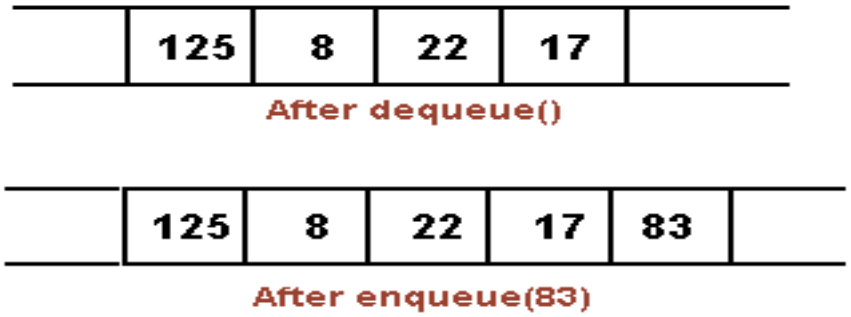
- Many times, we use a list in a way where we always add to the end, and always remove from the front.
- The first element put into the list will be the first element we take out of the list:
- First-In, First-Out ("FIFO")
- **Queue**: a more restricted List with the following constraints:
- Elements are stored by order of insertion from *front* to *back*.
- Items can only be added to the *back* of the queue.
- Only the *front* element can be accessed or removed.
- Goal: every operation on a queue should be **O(1)**.



Queue … (continued)

Operations on a queue

- **Offer** or **enqueue**: add an element to the back.
- **Remove** or **dequeue**: remove and return the element at the front.
- **peek**: return (but not remove) front element:
  - peek on an empty queue returns null.
- Other operations: isEmpty, size.



Queue features

- **ORDERING**: maintains order elements were added (new elements are added to the end by default).

Queue … (continued)

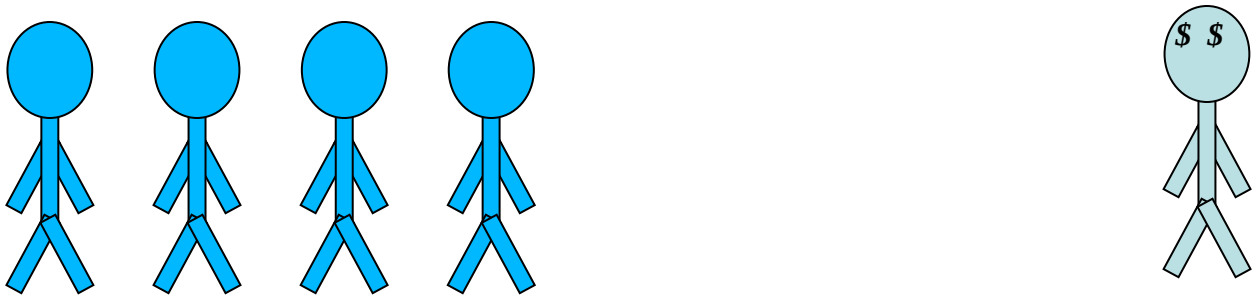
- OPERATIONS:
  - Add element to end of list ('offer' or 'enqueue').
  - Remove element from beginning of list ('remove' or 'dequeue') examine element at beginning of list ('peek').
  - Clear all elements/Initialize()
    - Initialize the queue to be empty
  - isFull():
    - Determine if queue is full or not. A Boolean operation needed for static queues. Returns true if the queue is full. Otherwise, returns false.

Queue … (continued)

- isEmpty():
  - Determine whether queue is empty or not.
    A Boolean operation needed for all queues . Returns true if the queue is empty. Otherwise, returns false.
- display():
  - If the queue is not empty then retrieve all elements.
- getsize().
  - Determine the number of elements in the queue
- All of these operations are efficient! O(1).

The Queue Operations

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

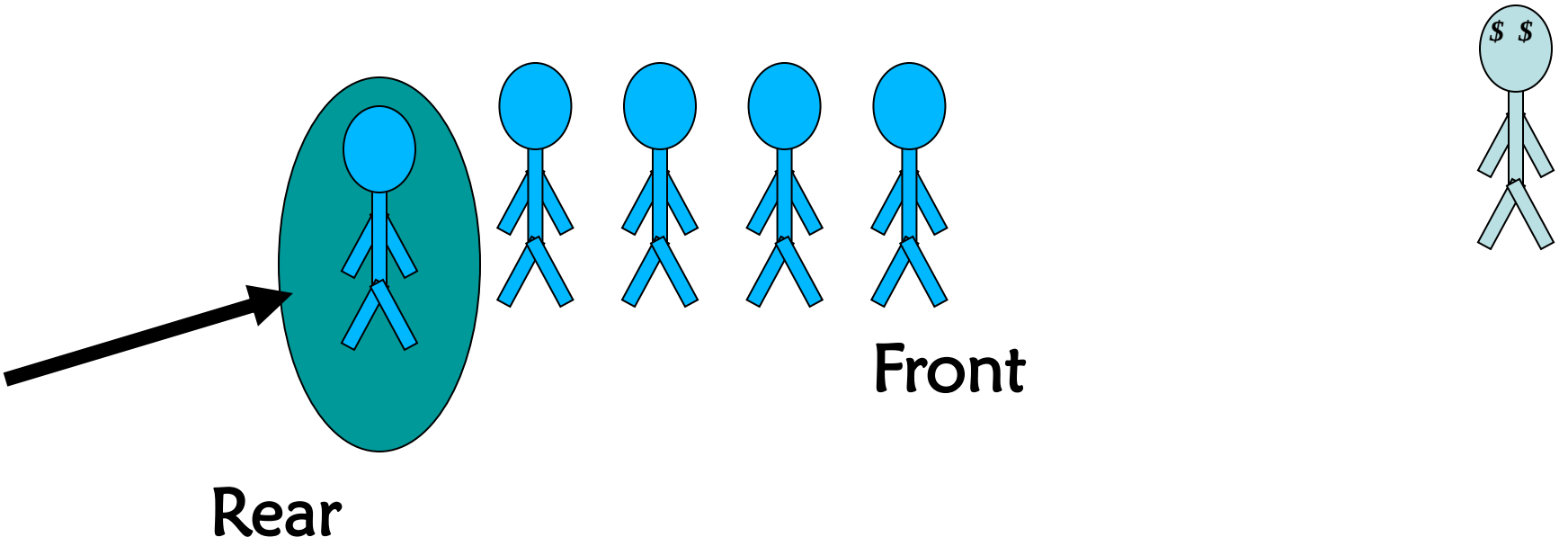


Rear

Front

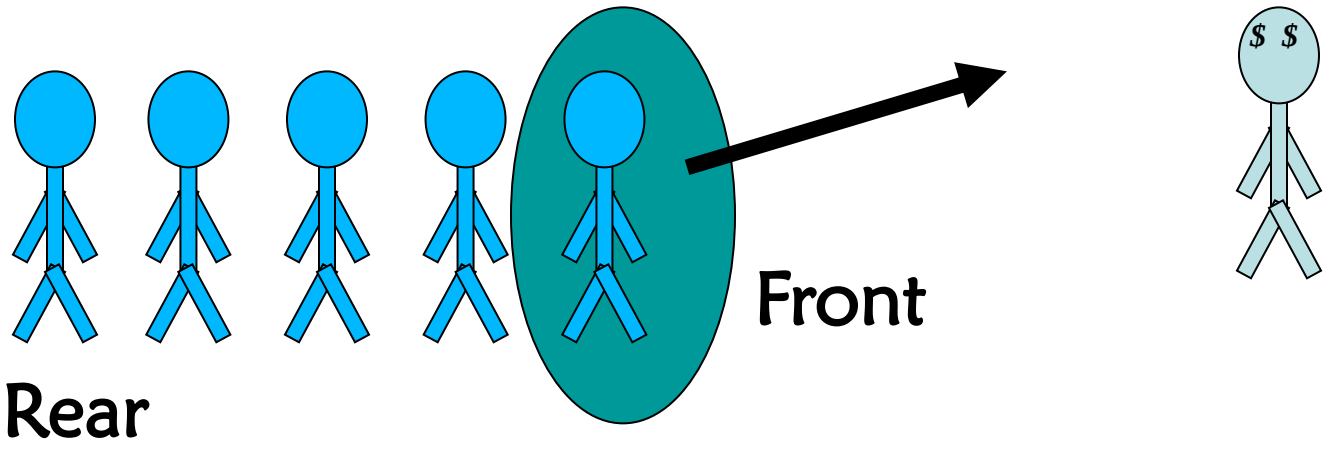
The Queue Operations

- New people must enter the queue at the rear. it is usually called an **enqueue** operation.



The Queue Operations

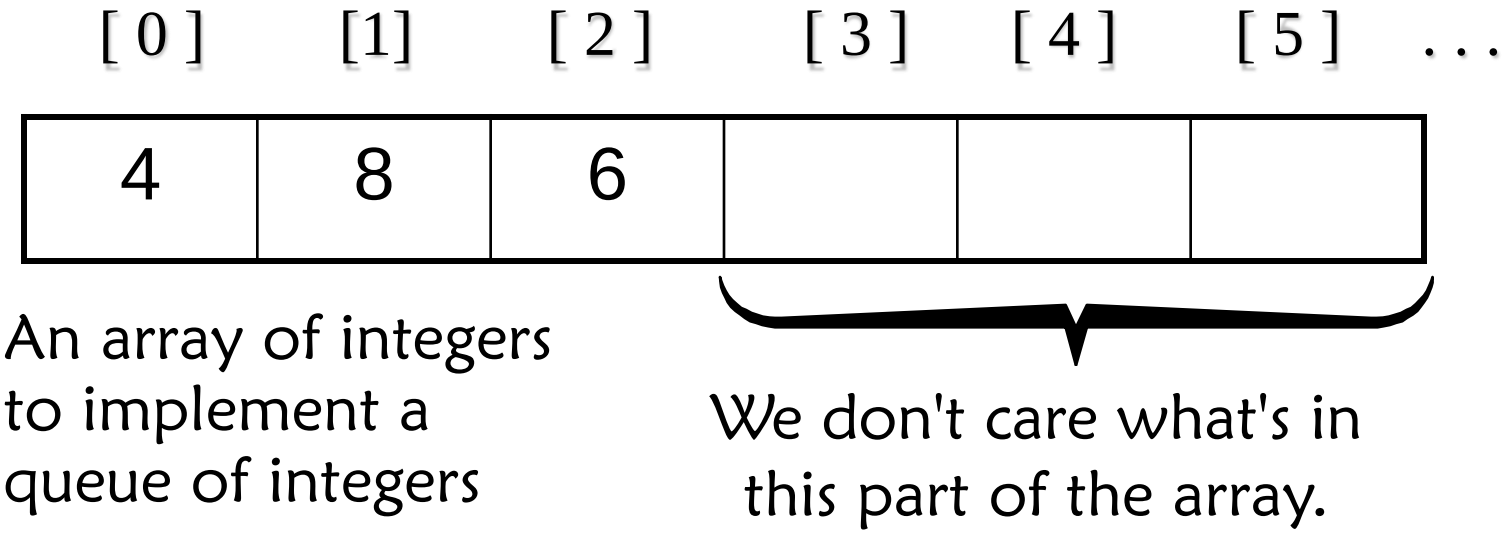
- When an item is taken from the queue, it always comes from the front. it is usually called a dequeue operation.



Array Implementation of Queue

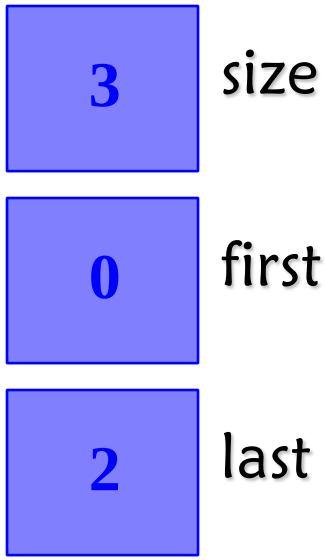
- A queue can be implemented with an array, as shown here.

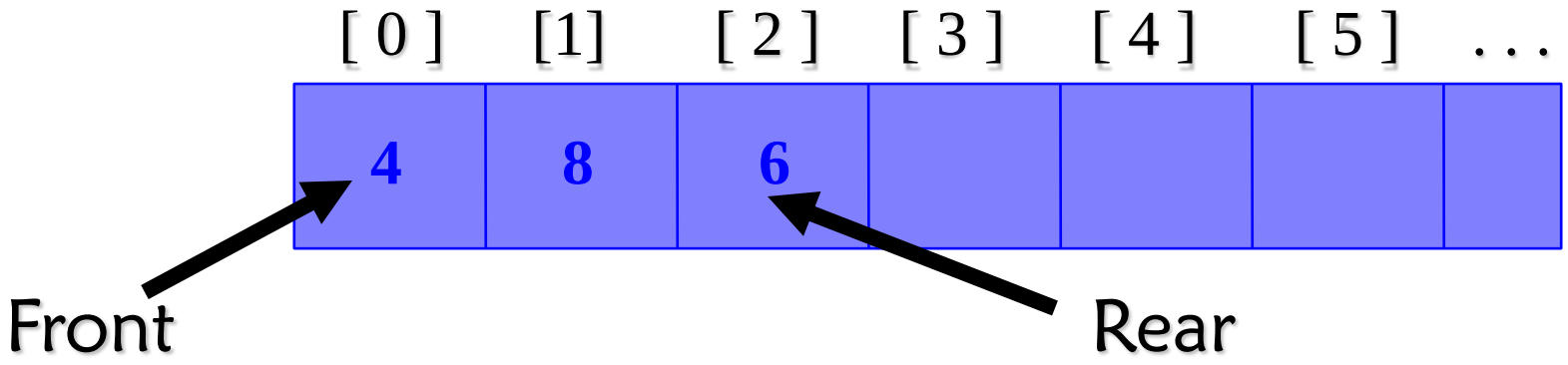
For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



Simple Array Implementation of Queue

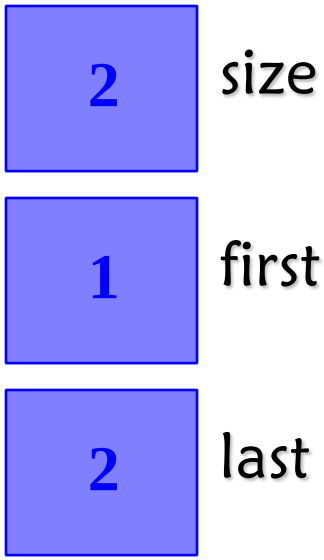
- The easiest implementation also keeps track of the number of items in the queue (Queue Size) and the index of the first element (at the front of the queue), the last element (at the rear).

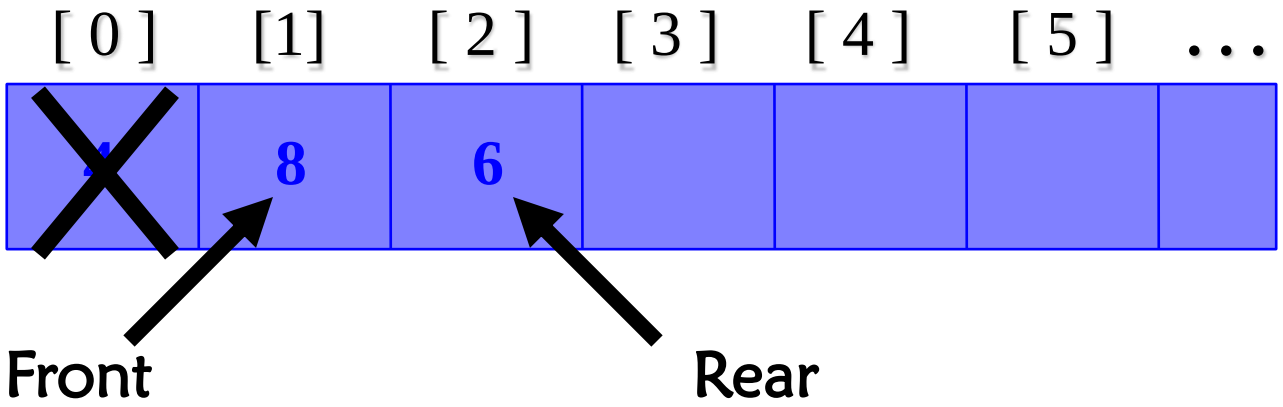




A Dequeue Operation

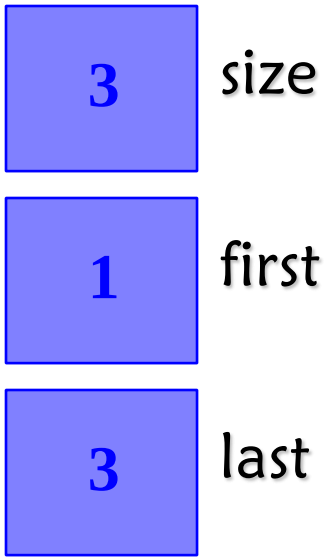
- When an element leaves the queue, size is decremented, and first changes, too

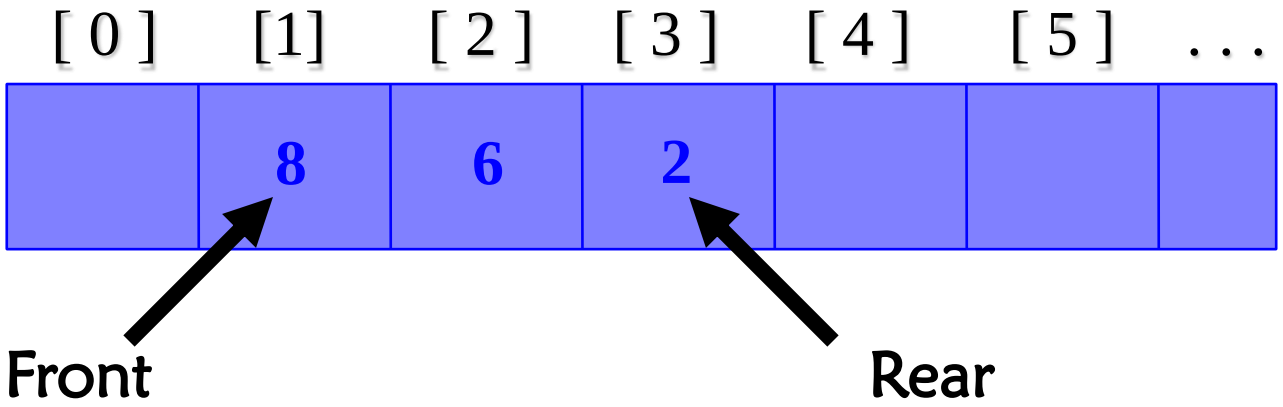




An Enqueue Operation

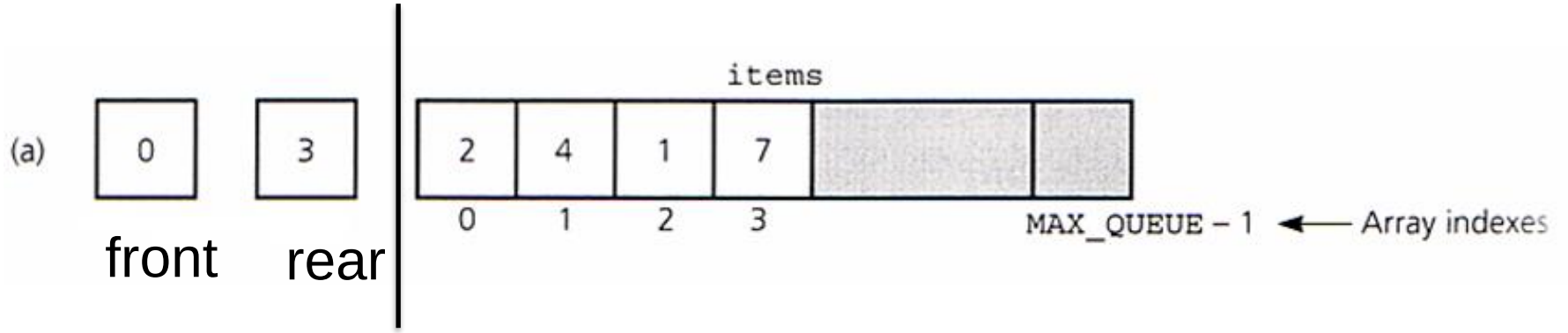
- When an element enters the queue, size is incremented, and last changes, too





Array Based Implementation

- Array: items[]
- Variables:
  - front
  - rear
  - MAX_QUEUE



Array Based Implementation

```
.....
int NumArray[];
int MAX_SIZE;
int FRONT= -1;
int REAR= -1;
int QUEUESIZE=0;
```

```
//*****************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.            *
//*****************************************
bool isFull()
{
  if (REAR<MAX_SIZE-1)
     return false;
  else
     return true;
}
```

```
//*****************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.            *
//*****************************************
bool isEmpty()
{
  if (QUEUESIZE==0)
     return true;
  else
     return false;
}
```

Array Based Implementation: Enqueue

Algorithm

To enqueue data to the queue

- check if there is space in the queue
  REAR<MAX_SIZE-1 ?
  Yes:
  - Increment REAR
  - Store the data in Num[REAR]
  - Increment QUEUESIZE
  - FRONT = = -1?
    Yes: - Increment FRONT

  No: - Queue Overflow

Implementation

```
void enqueue(int x)
{
        if(Rear<MAX_SIZE-1)
        {
            REAR++;
            Num[REAR]=x;
            QUEUESIZE++;
            if(FRONT = = -1)
                    FRONT++;
        }
        else
            cout<<"Queue Overflow";
}
```

Array Based Implementation: Dequeue

Algorithm

To dequeue data from the queue

- check if there is data in the queue

  QUEUESIZE > 0 ?

  Yes:
  - Copy the data in Num[FRONT]
  - Increment FRONT
  - Decrement QUEUESIZE

  No: - Queue Underflow

Implementation

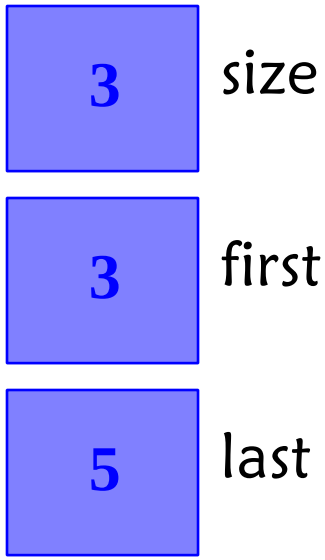
```
int dequeue()
{
        int x;
        if(QUEUESIZE>0)
        {
            x=Num[FRONT];
            FRONT++;
            QUEUESIZE--;

        }
        else
            cout<<"Queue Underflow";
        return(x);
}
```

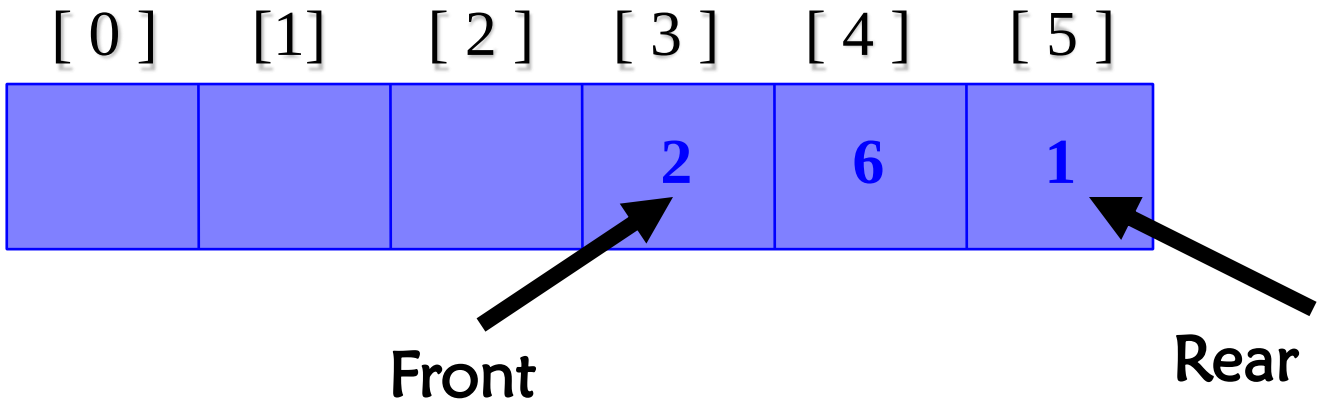
Circular Array Implementation of Queues

- A problem with simple arrays is we run out of space even if the queue never reaches the size of the array.
- Thus, circular arrays can be used to solve this problem.
    - Freed spaces are re-used to store data

Circular Array Implementation of Queues…

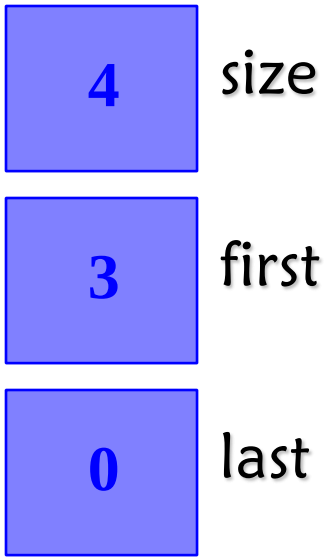
- There is special behaviour at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

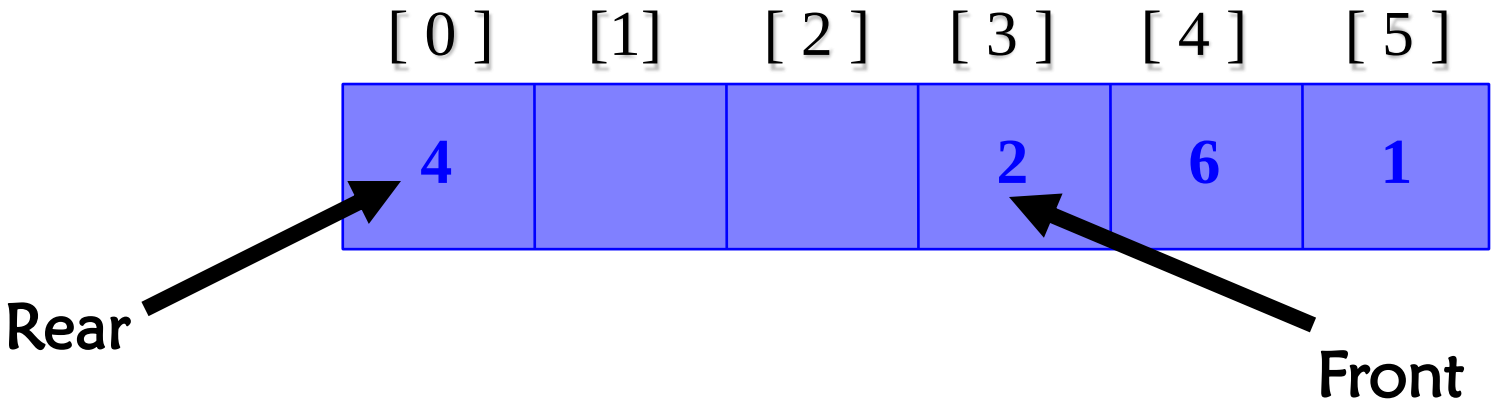




An Enqueue Operation

- The new element goes at the front of the array (if that spot isn't already used):





Example: Consider a queue with MAX_SIZE = 4

Operation	Simple array				Circular array			
	Content of the array	Content of the Queue	QUEUE SIZE	Message	Content of the array	Content of the queue	QUEUE SIZE	Message
Enqueue(B)	B	B	1		B	B	1	
Enqueue(C)	B C	BC	2		B C	BC	2	
Dequeue()	C	C	1		C	C	1	
Enqueue(G)	C G	CG	2		C G	CG	2	
Enqueue (F)	C G F	CGF	3		C G F	CGF	3	
Dequeue()	G F	GF	2		G F	GF	2	
Enqueue(A)	G F	GF	2	Overflow	A G F	GFA	3	
Enqueue(D)	G F	GF	2	Overflow	A D G F	GFAD	4	
Enqueue(C)	G F	GF	2	Overflow	A D G F	GFAD	4	Overflow
Dequeue()	F	F	1		A D F	FAD	3	
Enqueue(H)	F	F	1	Overflow	A D H F	FADH	4	
Dequeue ()		Empty	0		A D H	ADH	3	
Dequeue()		Empty	0	Underflow	D H	DH	2	
Dequeue()		Empty	0	Underflow	H	H	1	
Dequeue()		Empty	0	Underflow		Empty	0	
Dequeue()		Empty	0	Underflow		Empty	0	Underflow

Algorithm and Implementation: enqueue

To enqueue data to the queue

- check if there is space in the queue
  QUEUESIZE<MAX_SIZE ?
  - Yes:
    - Increment REAR
      REAR = = MAX_SIZE ?
      Yes: REAR = 0
    - Store the data in Num[REAR]
    - Increment QUEUESIZE
      FRONT = = -1?
      Yes: - Increment FRONT
  - No: - Queue Overflow

Implementation:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
{
    if(QUEUESIZE<MAX_SIZE)
    {
        REAR++;
        if(REAR = = MAX_SIZE)
            REAR=0;
        Num[REAR]=x;
        QUEUESIZE++;
        if(FRONT = = -1)
            FRONT++;
    }
    else
        cout<<"Queue Overflow";
}
```

Algorithm and Implementation: denqueue

To dequeue data from the queue

- check if there is data in the queue

  QUEUESIZE > 0 ?

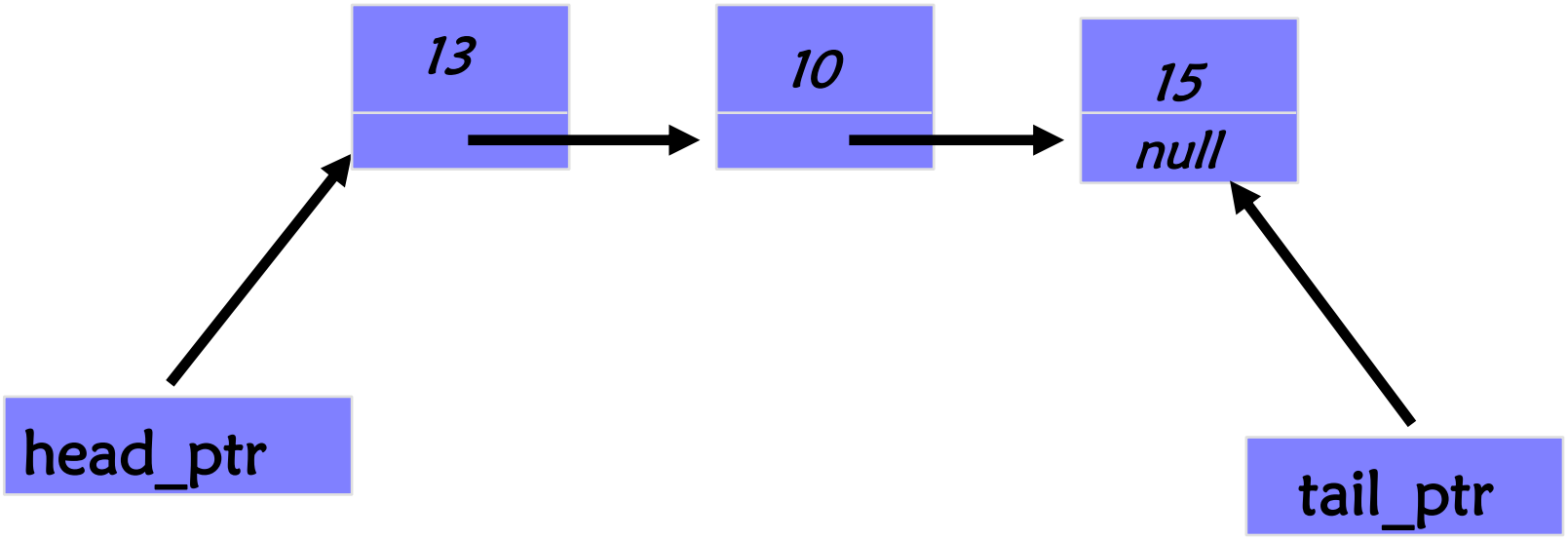
  Yes:
  - Copy the data in Num[FRONT]
  - Increment FRONT
    - FRONT = = MAX_SIZE ?
      - Yes: FRONT = 0
  - Decrement QUEUESIZE

  No:
  - Queue Underflow

```
int dequeue()
{
    int x;
    if(QUEUESIZE>0)
    {
        x=Num[FRONT];
        FRONT++;
        if(FRONT = = MAX_SIZE)
            FRONT = 0;
        QUEUESIZE--;

    }
    else
        cout<<"Queue Underflow";
    return(x);
}
```

Linked List Implementation

- A queue can also be implemented with a linked list with both a **head (start)** and a **tail (end)** pointer.
- **Enqueue**:- is inserting a node at the end of a linked list.
- **Dequeue**:- is deleting the first node in a linked list.



Implementation – see the linked list adding at the end and deleting at the start example from the previous lectures.

Different Types of Queue

Deque (pronounced as Deck)

- Is a Double Ended Queue.
- Insertion and deletion can occur at either end.
- Has the following basic operations:

  EnqueueFront:– inserts data at the front of a list.

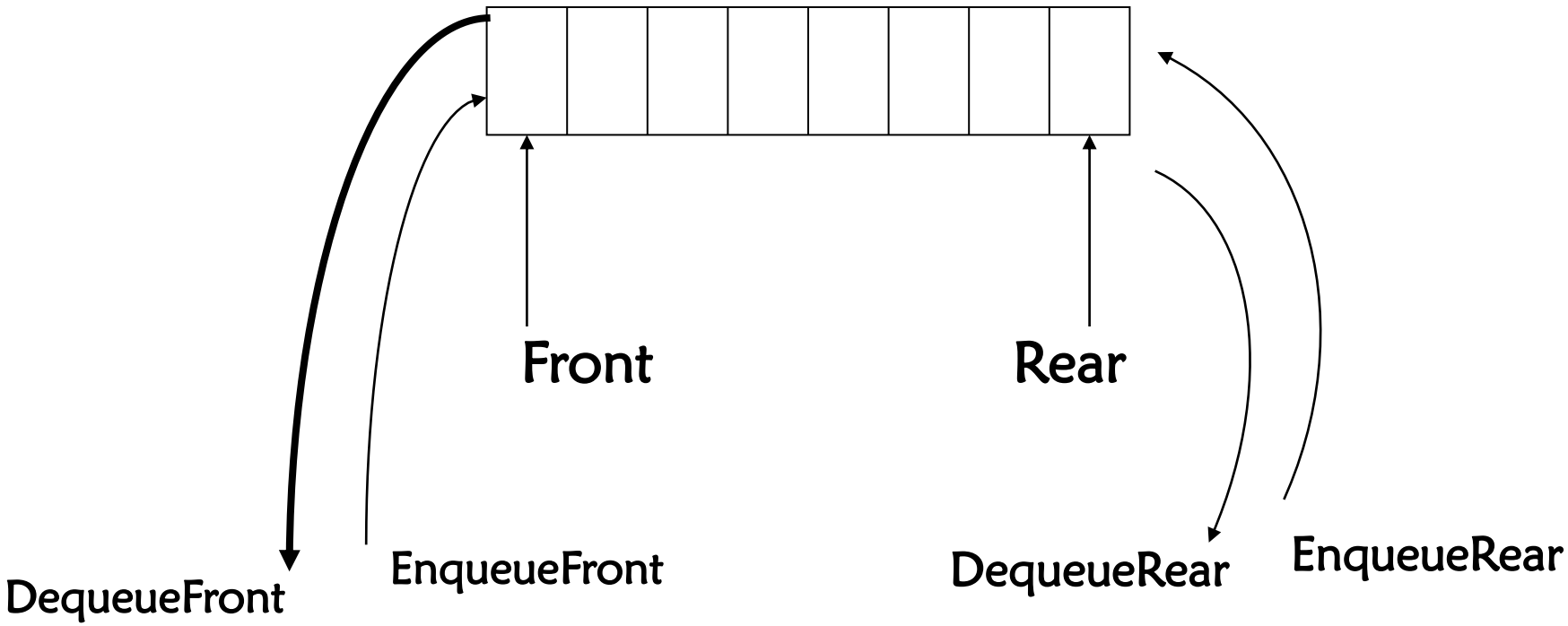
  DequeueFront:– deletes data at the front of a list.

  EnqueueRear:– inserts data at the end of a list.

  DequeueRear:– deletes data at the end of a list.

Deque (pronounced as Deck).....

- Implementation is similar to that of queue.
- Is best implemented using doubly linked list.



Priority Queue

- Is a queue where each data has an associated key that is provided at the time of insertion.
- Dequeue operation deletes data having highest priority in the list.
- One of the previously used dequeue or enqueue operations has to be modified.

Example:

Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Abebe	**Alemu**	**Aster**	**Belay**	**Kedir**	Meron	**Yonas**
Male	**Male**	**Female**	**Male**	**Male**	**Female**	**Male**

- Dequeue()- deletes Aster.

Abe**be**	Alemu	Belay	Kedir	Meron	Yonas
Male	Male	Male	Male	Female	Male

- **Dequeue()- deletes Meron**

Abebe	Alemu	Belay	Kedir	Yonas
Male	**Male**	Male	Male	Male

- Now the queue has data having equal priority and **dequeue** operation deletes the front element like in the case of ordinary queues.

- **Dequeue()**:- deletes Abebe

- **Dequeue():-** deletes Alemu

Belay	Kedir	Yonas
Male	Male	Male

- Thus, in the above example the implementation of the **dequeue** operation need to be modified.

Demerging Queues:

- Is the process of creating two or more queues from a single queue.
- Used to give priority for some groups of data

Example: The following two queues can be created from the above priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Algorithm:

```
create empty females and males queue
while (PriorityQueue is not empty)
{
 Data=DequeuePriorityQueue();//delete data at the
  front
 if(gender of Data is Female)
     EnqueueFemale(Data);
  else
          EnqueueMale(Data);
  }
```

Merging Queues:

- Is the process of creating a priority queue from two or more queues.
- The ordinary dequeue implementation can be used to delete data in the newly created priority queue.

Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Aster	Meron	Abebe	Alemu	Belay	Kedir	Yonas
Female	Female	Male	Male	Male	Male	Male

Algorithm:

```
create an empty priority queue
while(FemalesQueue is not empty)
  EnqueuePriorityQueue(DequeueFemalesQueue());
while(MalesQueue is not empty)
     EnqueuePriorityQueue(DequeueMalesQueue());
```

Example 2

- The data with the largest element with higher priority.

A	B	C	F	E	D	H
40	30	25	15	18	20	12

- Dequeue() A will be selected
- Dequeue() B >>
- Dequeue() C >>
- Dequeue() D >>

Application of Queues

I. **Access to shared resources** (Example: printer)

```
Print()
{
        EnqueuePrintQueue(Document)
}
EndOfPrint()
{
    DequeuePrintQueue()
}
```

II. **Disk Driver:-**maintains a queue of disk input/output requests.

Application of Queues….

III. **Task scheduler in multiprocessing system** maintains priority queues of processes.

IV. **Telephone calls in a busy environment** maintains a queue of telephone calls.

V. **Simulation of waiting line** - maintains a queue of persons.