COMP 2310 Data Structures and Algorithms

Lecture 5

Queue Structures

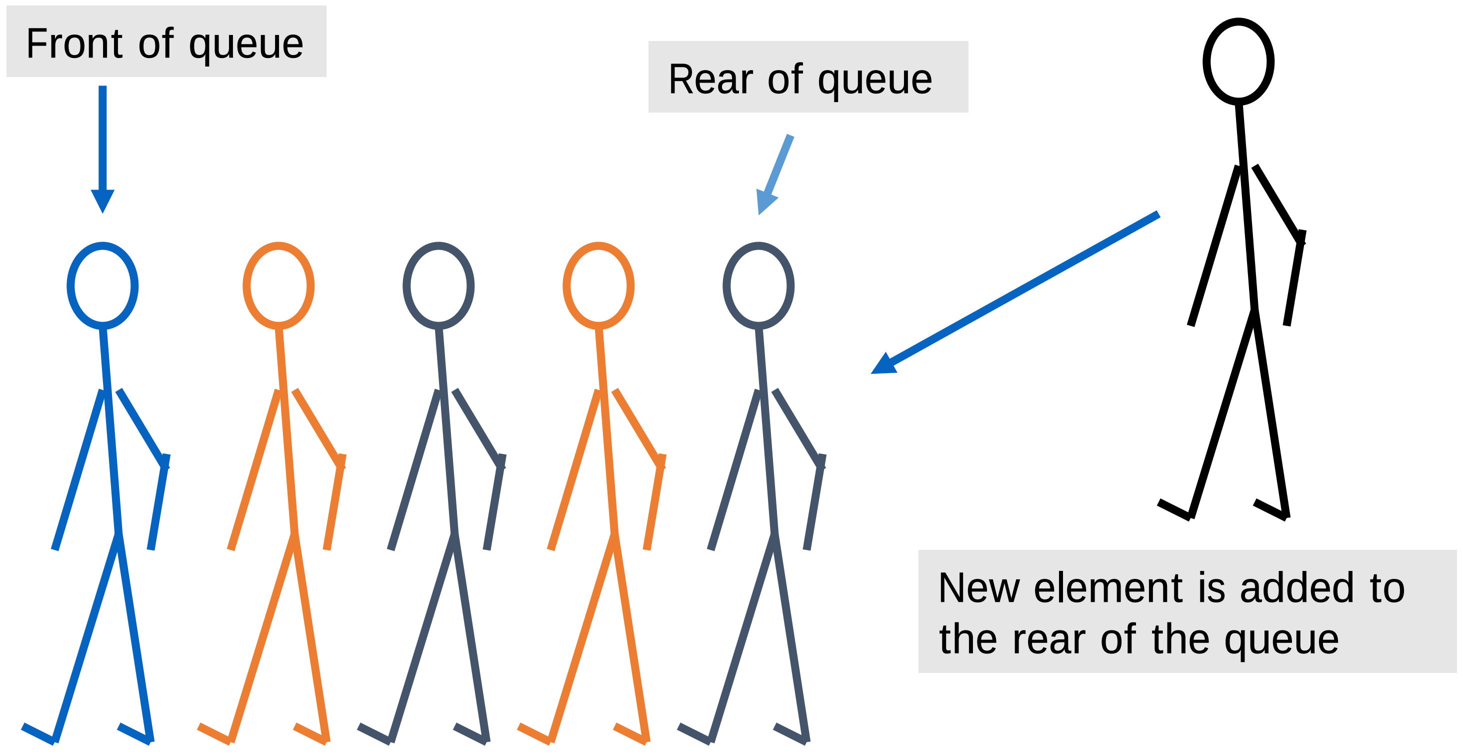
QUEUES

Rear of Queue
Front of Queue
next...
PAY HERE

LAST IN LAST OUT
ONE WAY
FIRST IN FIRST OUT

Conceptual View of a Queue

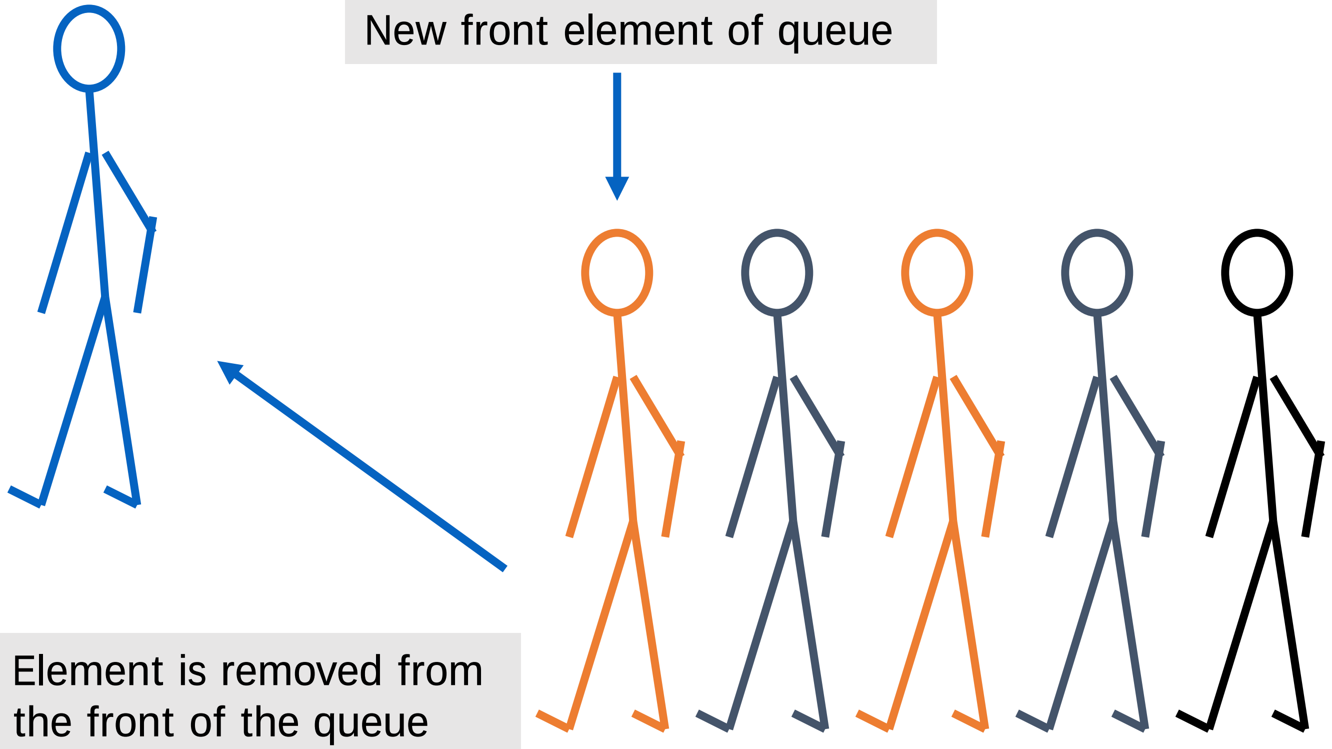
Adding an element



Conceptual View of a Queue

Removing an element

New front element of queue



Element is removed from the front of the queue

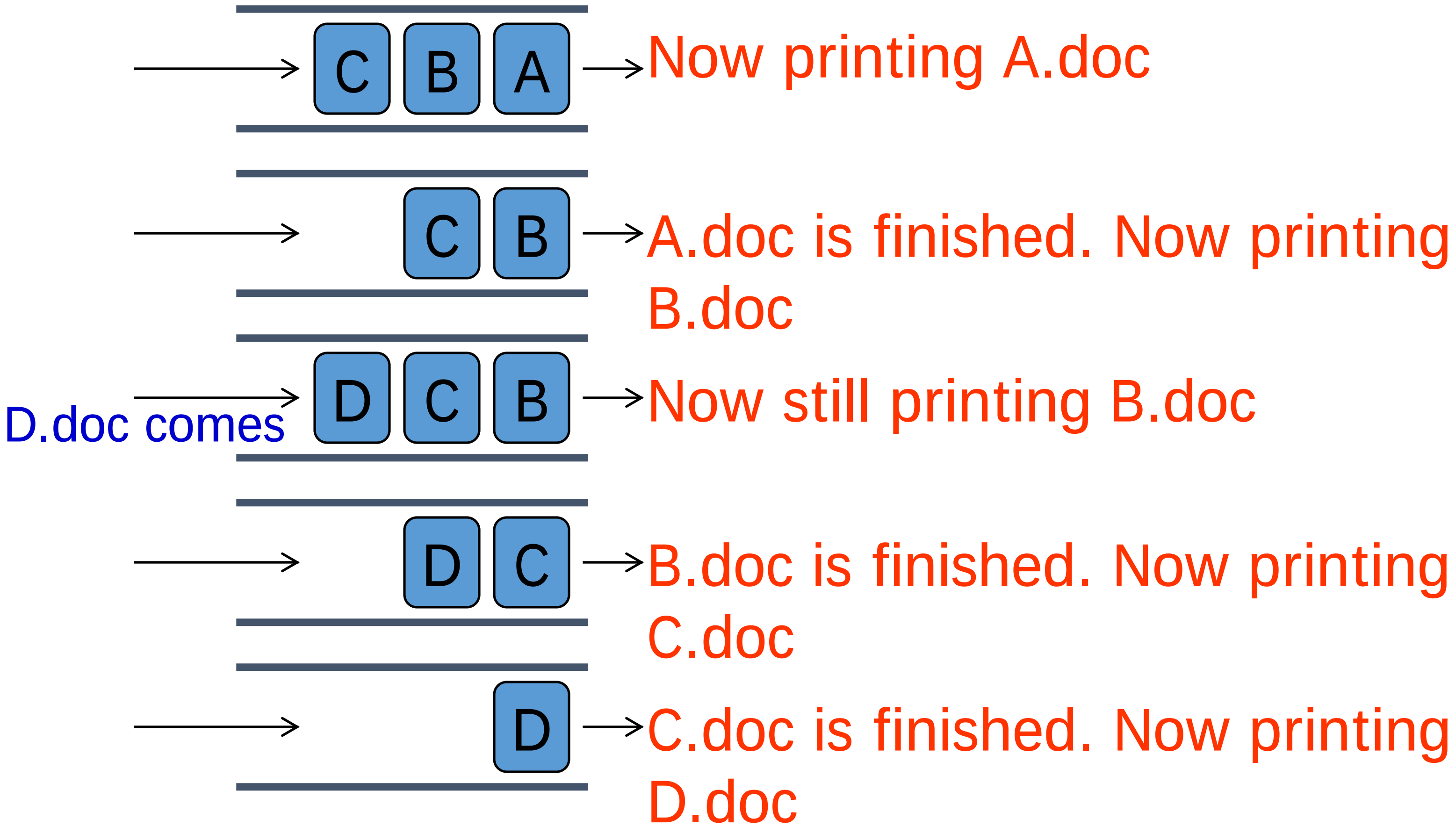
Queue Structures

- Like a stack, a queue is an abstract data type that holds a linear sequence of elements.
- There are two ends of a queue: front (Head), rear (back/tail)
- In a queue, all insertions take place at the *rear*, all deletions take place at the *front*.
- The elements are processed like customers standing in a check-out line: the first customer in line is the first one served.

→ *first-in, first-out: FIFO*

Queue Example: Printing Queue

- A.doc, B.doc, C.doc arrive to printer.


C B A
Now printing A.doc
C B
A.doc is finished. Now printing B.doc
D.doc comes
D C B
Now still printing B.doc
D C
B.doc is finished. Now printing C.doc
D
C.doc is finished. Now printing D.doc

Queue Structure Operations

Common queue operations include:

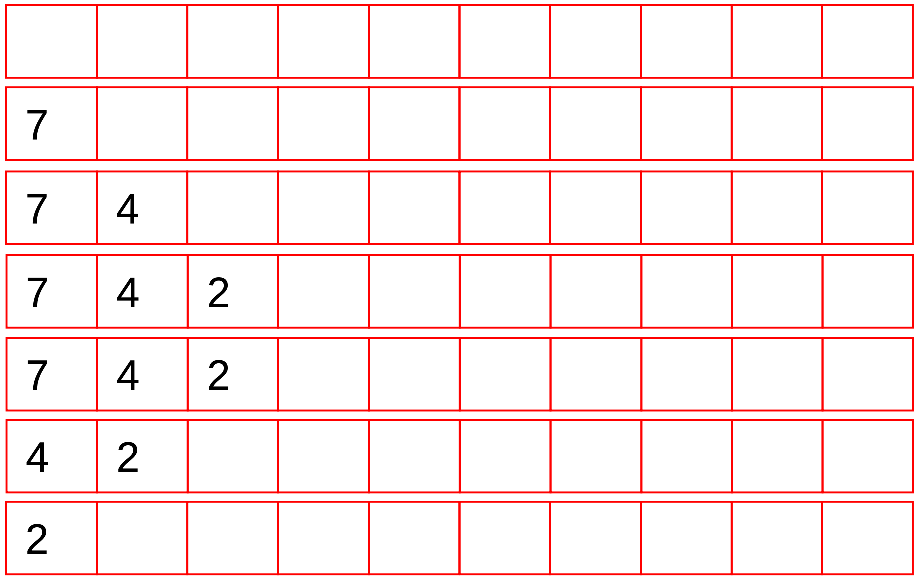
- Create an empty queue
- enqueue(): Add a new item to the queue (Insert)
- dequeue(): Remove an item from the queue (Delete)
- isEmpty(): Check to see if the queue is empty.
- isFull(): Check to see if there is more space for insertion.

Two index variables front and rear (back) specify the two ends of a queue.

Queue Operations

Initially: front=rear=-1

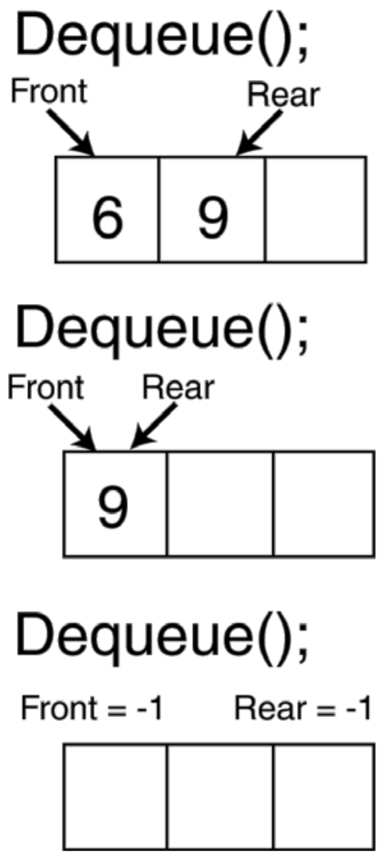
```
createQueue()
   enqueue(7)
   enqueue(4)
   enqueue(2)
front=0,rear=2
     dequeue()
     dequeue()
front=0,rear=0
```



Queue Operations

- After each insert/delete, the front or rear index has to be updated.

Example: Assume the queue contains: 4,6,9. front=0, rear=2.

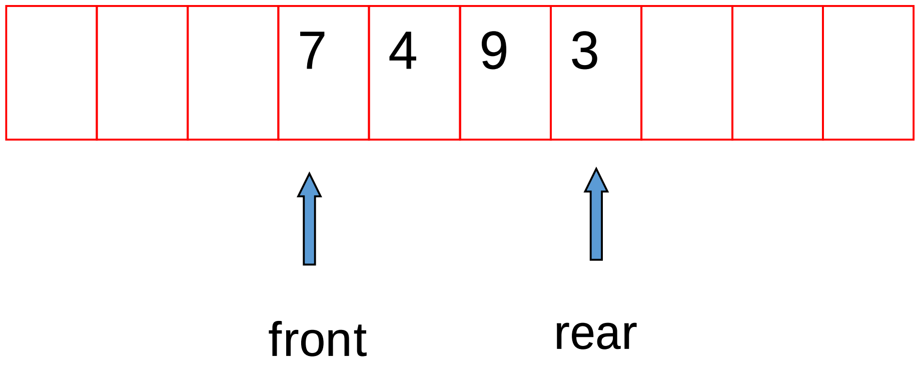


Array Implementation Of Queues

- Like stacks, queue structures can be implemented using an array.
- We need to keep the positions of front and rear in two index variables.

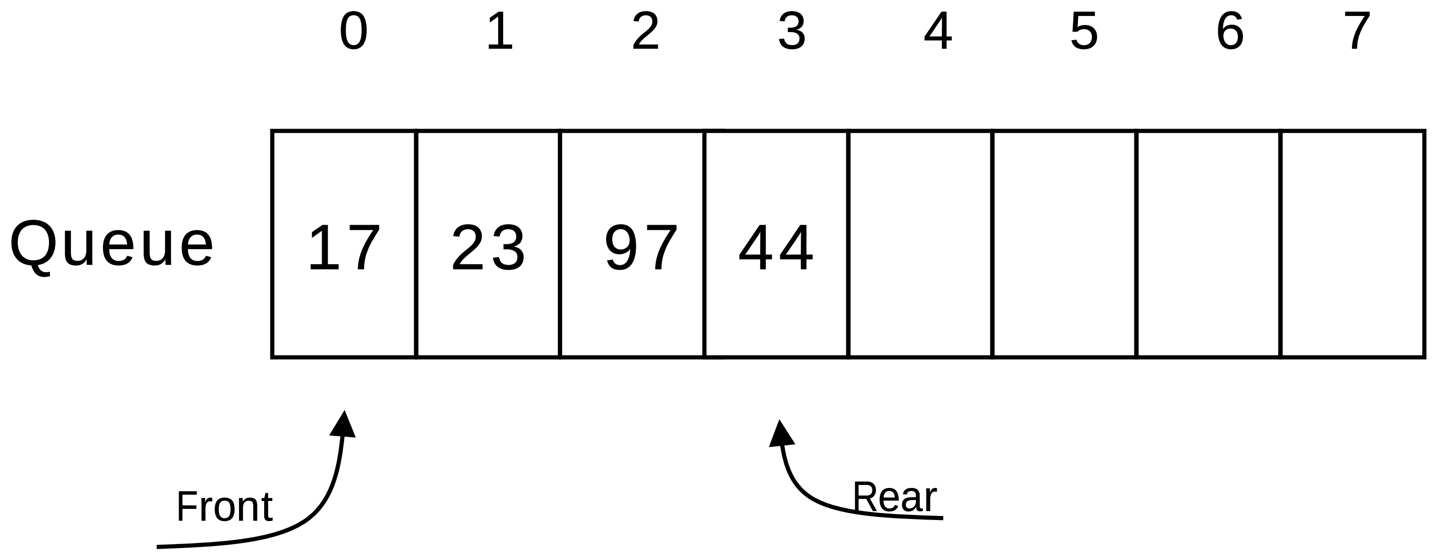
Empty queue: front = rear = -1

- After some operations the queue may look like this:



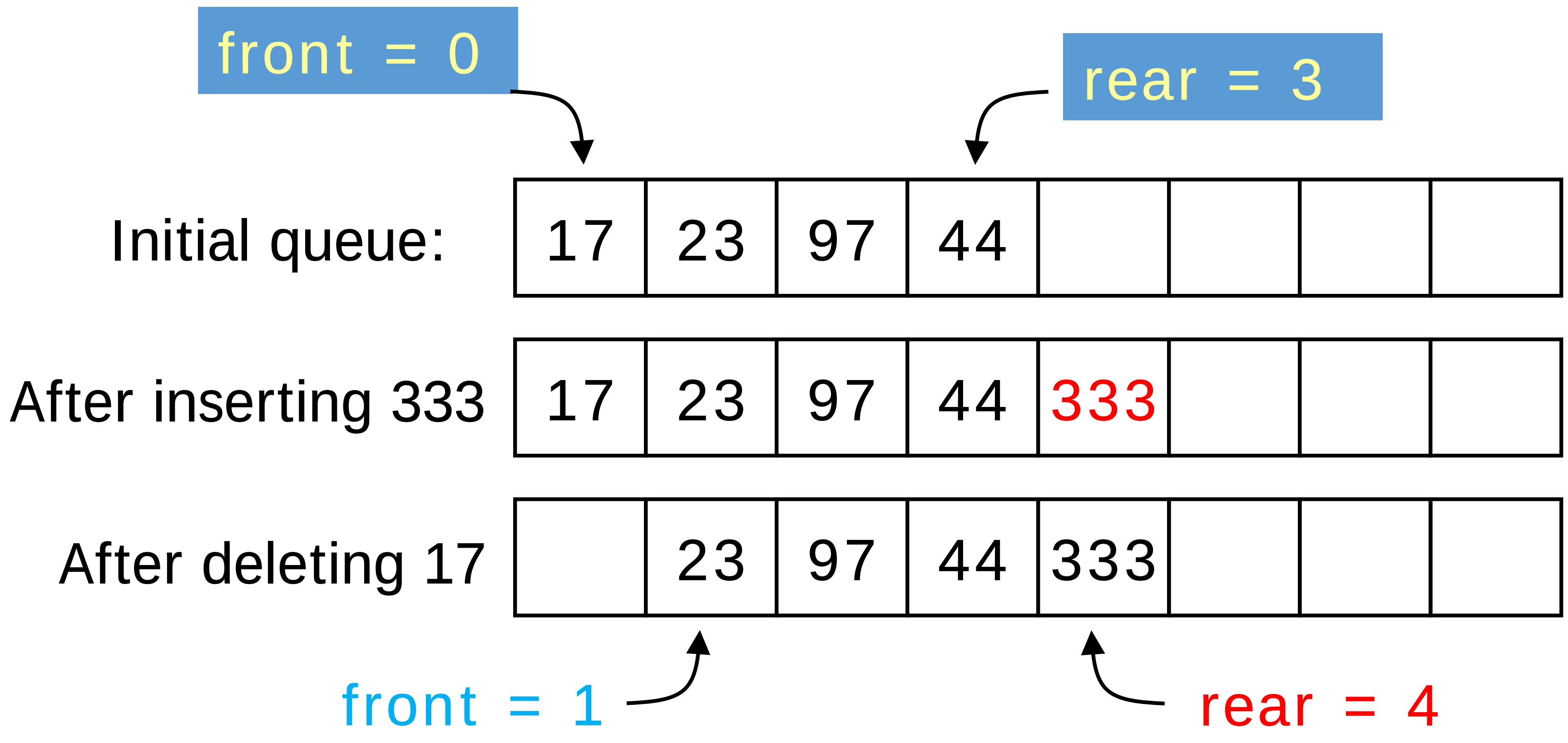
Array implementation of queues

- FIFO is accomplished by inserting at one end and deleting from the other end



- To insert: put new element in location 4, and set rear to 4
- To delete: take element from location 0, and set front to 1

Array implementation of queues: Example



- Notice how the array contents move to the right as elements are inserted and deleted
- This will be a problem after a while! Why?

Problems of Array Implementation

- A static array has limited size, once rear index is at the end of this array, and there are new items coming in, what can we do?

Two Solutions:

1. Shifting all items to front in the array to empty positions (Too Costly…)
2. Moving the rear index to the start of the queue

   $\rightarrow$ circular queue

   A circular queue is implemented on a circular array.

Circular Queues

- When a new item is inserted or deleted, the corresponding index has to be moved upwards.

- After many insert and delete operations, the rear might reach the end of the queue and then no more items can be inserted.

- However, the items from the front have been deleted and there is space there.

$\rightarrow$Continue insertions from the front!

Circular Queues

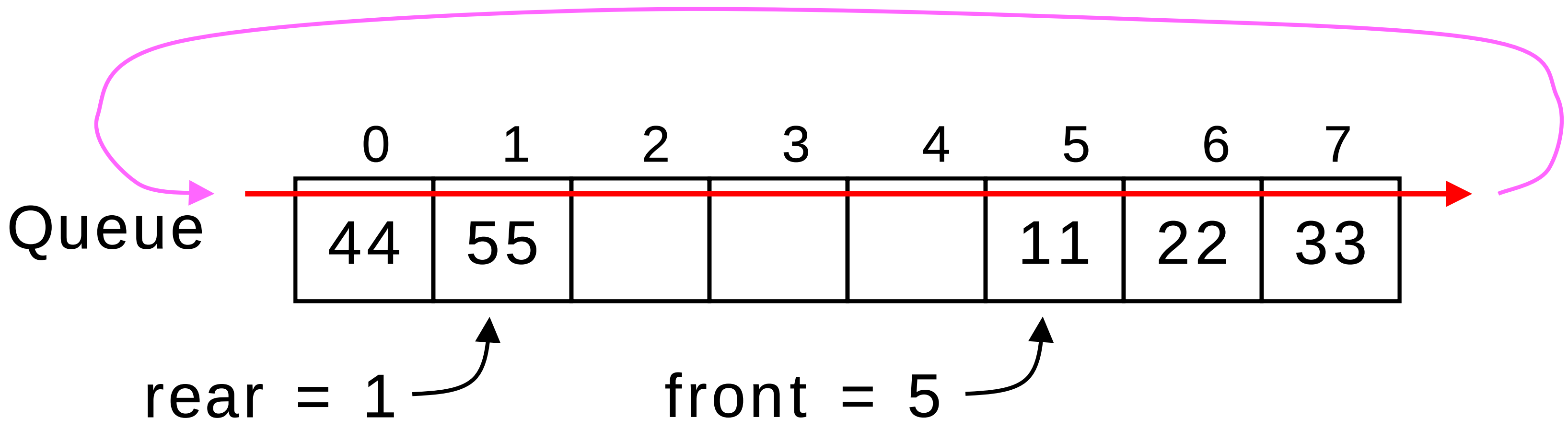
- Circular arrays are used to implement circular queue structures.

- In circular queues, whenever *front* or *back* reaches the end of the array it is wrapped around the beginning.

  →Both the front and the rear indexes wrap around to the beginning of the array.

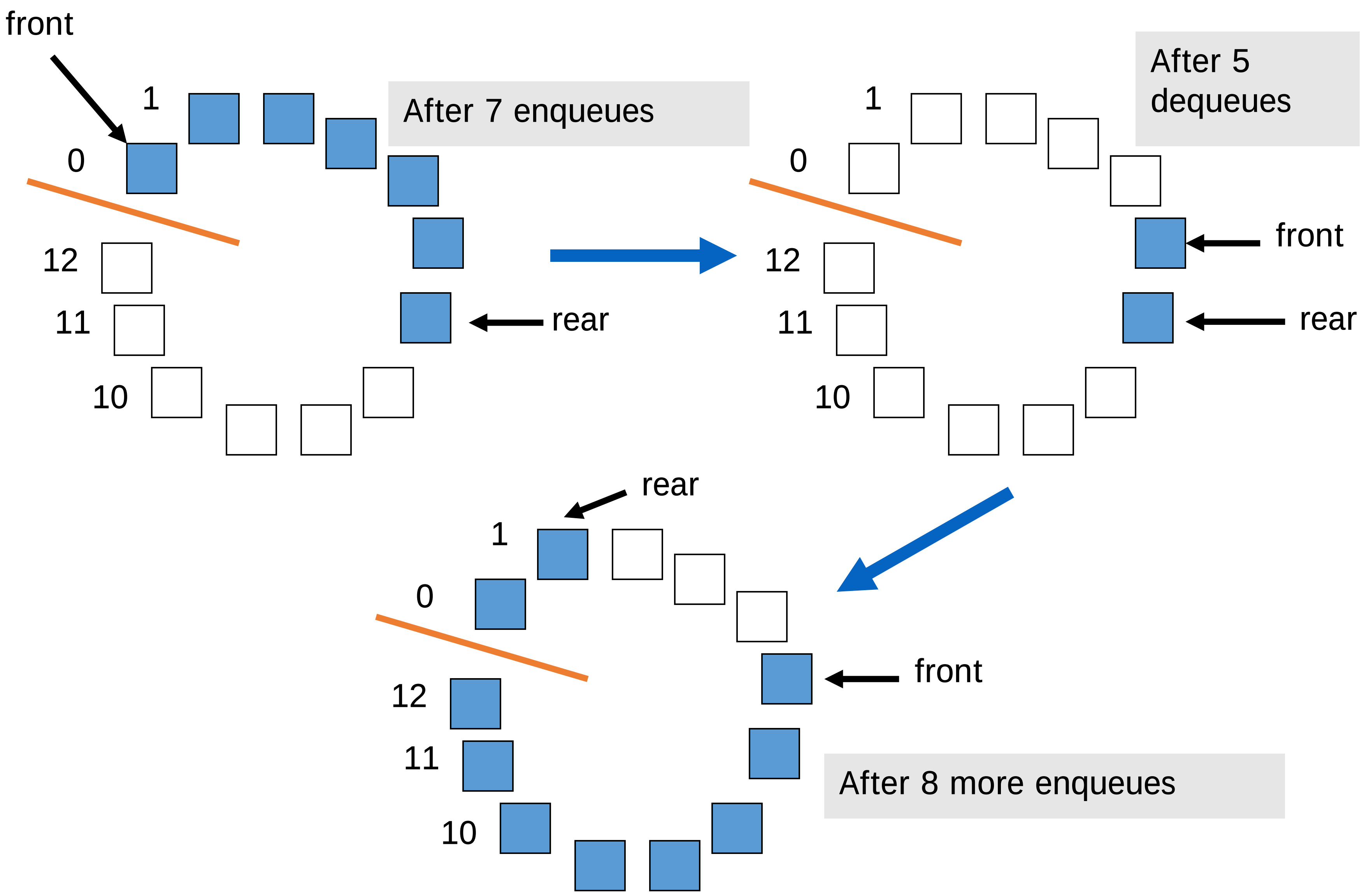
Circular arrays

- We use the array holding queue elements in circular form: The two ends are joined!



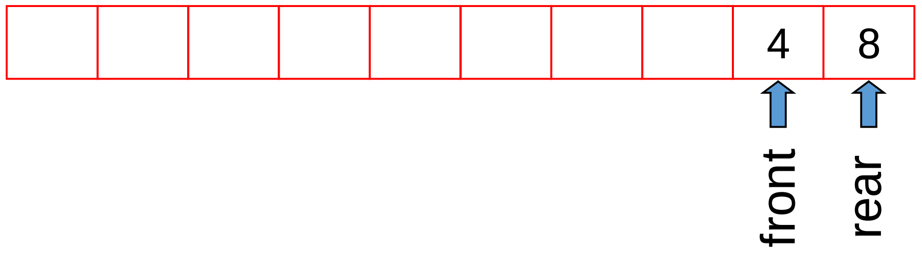
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order

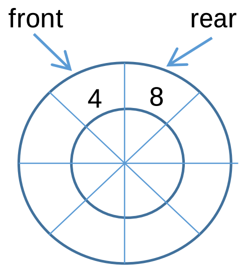
Example of a Circular Queue



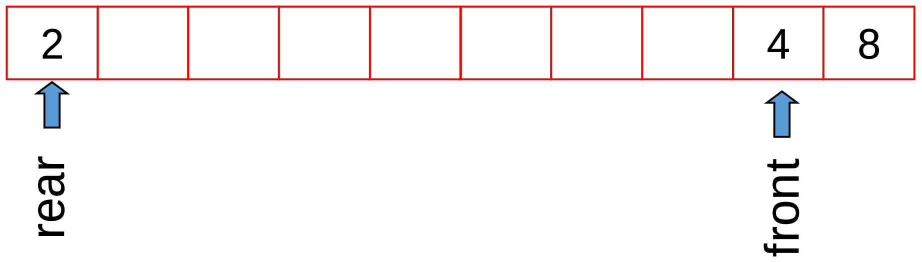
Circular Queue Operations

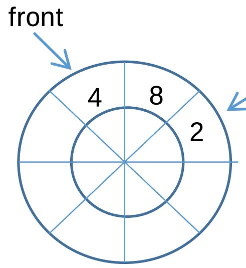
Initial state:



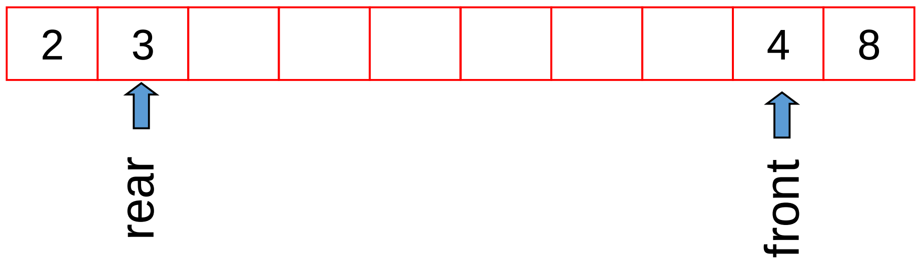


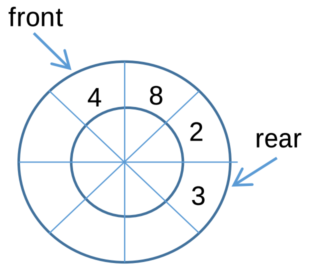
enqueue(2)





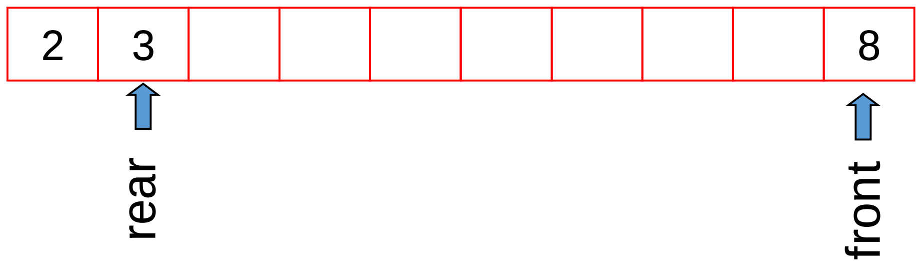
enqueue(3)

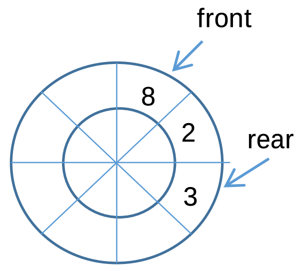




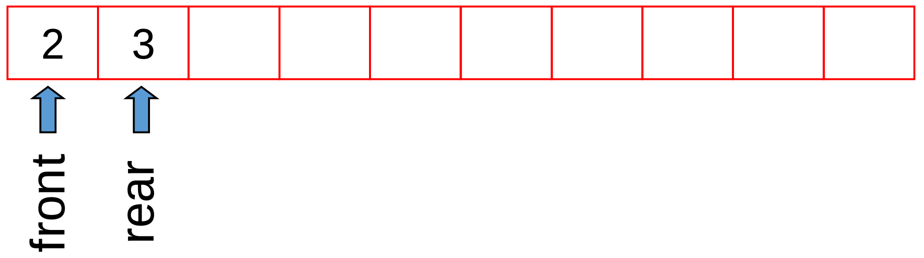
Circular Queue Operations

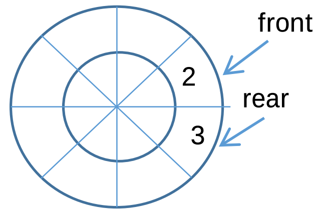
dequeue()



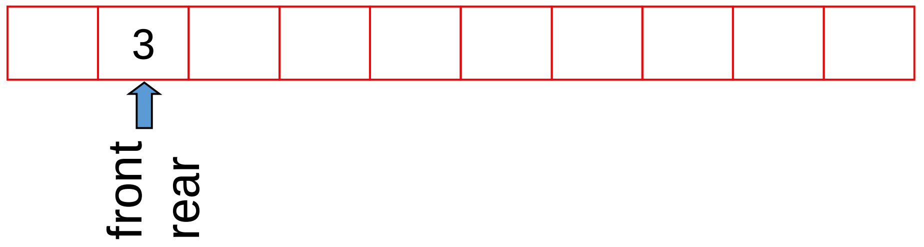


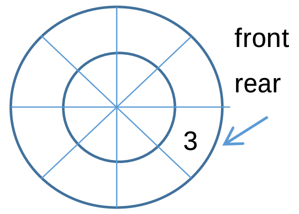
dequeue()





dequeue()





Queue Operations: isEmpty and isFull

// Assume circular array implementation and queue capacity = n

```
bool IS_EMPTY()
    if front == -1 and rear == -1
        return true
    else
        return false


bool IS_FULL()
    if front==0 and rear==n-1
        return true
    if ( front == rear + 1 )
        return true
    return false
```

Queue Operations: Enqueue

```
// Q size is n, Insert x.
// Assume Circular queue
Enqueue(x)
if (isFull())
    print ``Queue is full``
    return
if (front==-1 and rear==-1)
    front  ← 0
    rear ← 0
else
    rear ← (rear+1)%n
Q[rear] ← x
```

Queue Operations: Dequeue

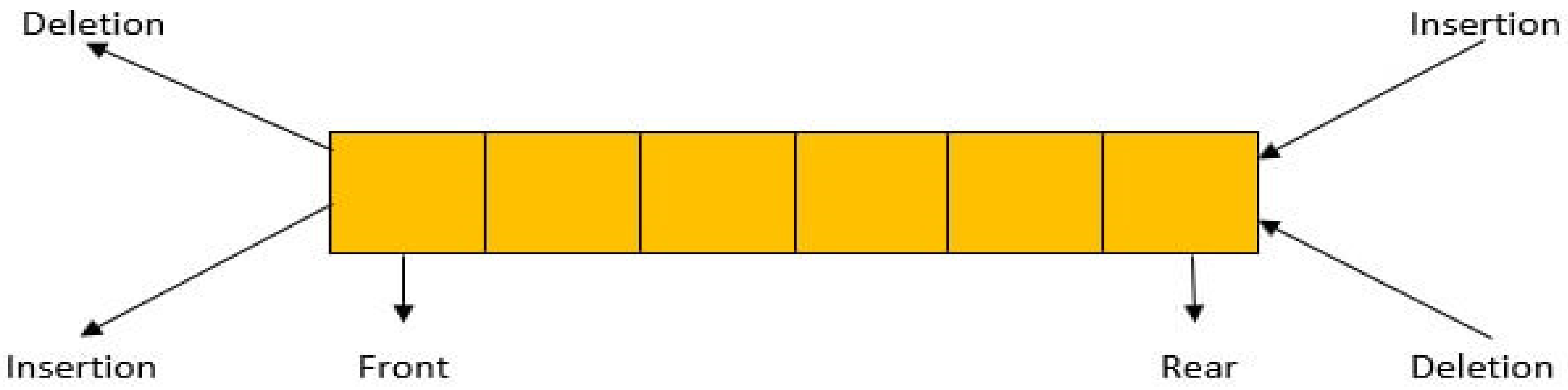
```
// Remove x from the queue, size=n
Dequeue()
if isEmpty()
   print ``Queue is empty``
   then return
x ← Q[front]
if (front==rear)
    front ← -1
    rear ← -1
else
    front ←(front+1)%n
return x
```

Queue Operations: Display

```
Display()
if isEmpty()
    print ``queue is empty``
else
    for(i=front ; i!=rear ; i=(i+1)%n)
        print Q[i]
   print Q[i]
```

Double Ended Queues

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both ends.
- That means, we can insert at both front and rear positions and can delete from both front and rear positions.
- Often written as Deque and pronounced like "deck" to avoid confusing it with Dequeue Operation



Double Ended Queues: Operations

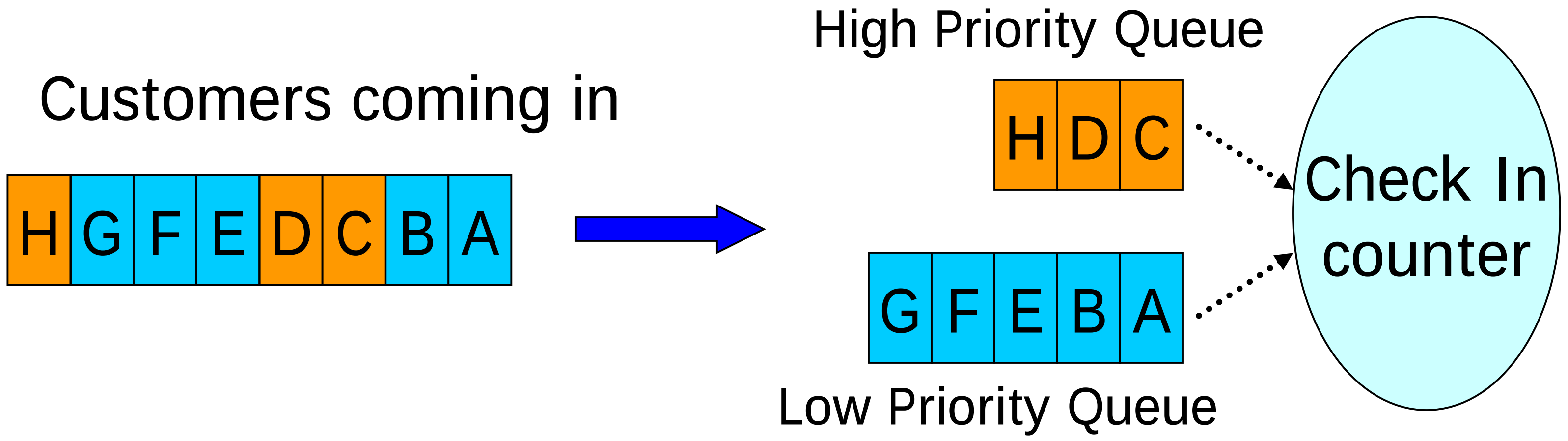
- **insertFront:** Insert or add an item at the front of the deque.
- **insertLast:** Insert or add an item at the rear of the deque.
- **deleteFront:** Delete or remove the item from the front of the queue.
- **deleteLast:** Delete or remove the item from the rear of the queue.
- **getFront:** Retrieves the front item in the deque.
- **getLast:** Retrieves the last item in the queue.
- **isEmpty:** Checks if the deque is empty.
- **isFull:** Checks if the deque is full.

Priority Queues

- The queues that we have considered so far are “Honest” queues: All operations are FIFO.
- In real life this may not be realistic or practical and some elements or operations may have priority over the others.
- Consider two queues:
  - one is high priority queue
  - the other is low priority queue
- Service rules:
  - Start service from the high priority queue
  - If there are no elements in the high priority queue, then begin serving the low priority queue

Two Queues with Different Priorites

- High Priority Queue, will come in *hpQue* 🟧
- Low Priority Queue, will come in *lpQue* 🟦



Check-in begins with hpQue, when it becomes empty, service in lpQue starts.

Priority Queue Implementation

- Priority queues are more specialized data structures than a stack or a simple queue.

- In a priority queue, items are ordered by a key value, in such a way that, the items with the lowest key (or the highest) are always at the front.

  → The items with minimum key have the highest priority.

- Items are inserted/deleted in the proper position according to the key values to maintain the order.

Priority Queue Example: Air Travel Check-In

- Assume there is only one check-in service in THY counter at some airport.

- Two waiting lines (Queues) for passengers are formed:
  - One is First class service
  - The other is Economy class service

- Passengers in the first-class waiting line have higher priority to check-in than those in the economy-class waiting line.

Pseudocode For Arrival

Priority Queue Passenger Arrival:

```
if (new Passenger comes)  {
        if(is FirstClass)
            hpQue.enqueue(new Passenger);
        else
            lpQue.enqueue(new Passenger);
 }
```

Pseudocode for Priority Queue Service

Priority Queue Check-In Service:

```
if( !isEmpty(hpQue) ) {
        serve the passenger from high priority queue
        hpQue.dequeue();
 }
 else  {
        serve the passenger from low priority queue
        lpQue.dequeue();
  }
```

Computer Applications of Queues

- In a multi-user system, a queue holds print jobs submitted by users. The printer services those jobs one at a time (FIFO).
- Communications software uses queues to hold information received over networks:
  - If information is transmitted to a system faster than it can be processed, it is placed in a queue.
- Job scheduling in operating systems.
- Simulation of real-world situations.

Computer Applications of Queues

- Priority queues are used in various ways in certain computer systems.
- Multitasking operating systems: Schedule of jobs on a shared computer. Some jobs will be more urgent than others. A priority queue keeps track of the jobs to be performed and their priorities. If a job is finished or interrupted, then one with highest priority will be performed next.
- Networking: Managing limited bandwidth on a transmission line. In the event of outgoing traffic queuing due to insufficient bandwidth, all other queues can be halted to send the traffic from the highest priority queue upon arrival.
- Priority queues are best implemented with a data structure called a heap.

Queue Appplications: Recognizing Palindromes

- A palindrom is a string of characters which reads the same backwards and forwards.

  Examples: eye, madam, racecar, no lemon, no melon

  Was it a car or a cat I saw?

  NIPSON ANOMIMATA MI MONANOS PIN

  (Means: wash your sins not only your face.

  Written on a fountain in Hagia Sofia (Ayasofya Mosque) in Istanbul)

Queue Appplications: Recognizing Palindromes

- We can use both a stack and a queue to find out whether a string is a palindrome or not.
- Starting from the first character of a string and travesing from left to right we insert each character into both a stack and a queue.
- Characters removed (popped) from the stack will be in the opposite order.
- Characters removed (dequed) from the queue will be in the original order.

Appplications: Recognizing Palindromes

Based on this information;

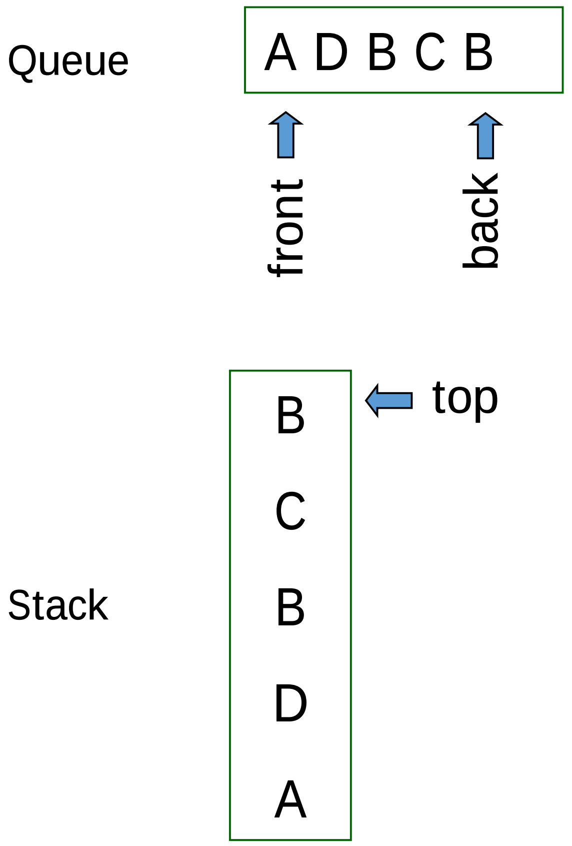
- If the characters at the top of the stack and at the front of the queue are compared and found to be the same then we can delete both of them.
- This process is continued until the end of the string.
- Finally if
  - The stack and the queue both are empty then the original string is a palindrome.
  - Otherwise the string is not a palindrome.

Recognizing Palindromes: Pseudocode

```
//Str contains the string to be checked
IsPalindrome ($Str$)
CreateQueue($Q$) , CreateStack($S$)
$n \leftarrow length[Str]$
for $i \leftarrow 1$ to $n$
   do $nextchar \leftarrow Str[i]$
      Enqueue ($nextchar$)
      Push($nextchar$)
// Compare the queue characters with stack characters
$charactersSame \leftarrow TRUE$
while $Q$ is not empty and $charactersSame$
        $x \leftarrow$ Dequeue()    // x=$Q[front]$
        $y \leftarrow$ Pop()        // y= $S[top]$
        if $x$ != $y$
            $charactersSame \leftarrow FALSE$
 return $charactersSame$
```

Appplications: Recognizing Palindromes

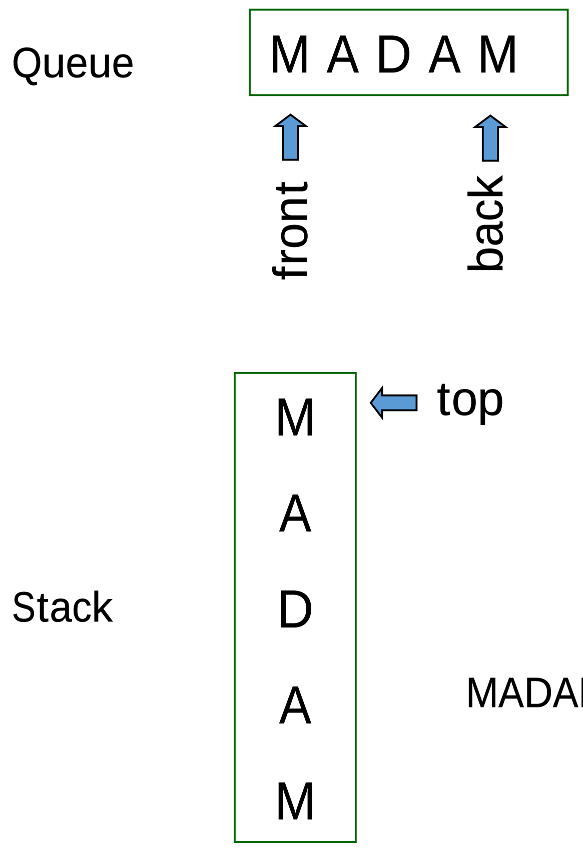
String: ADBCB



Dequeue : X=A
Pop stack: Y=B
X != Y
Not a palindrome !

Appplications: Recognizing Palindromes

String: MADAM

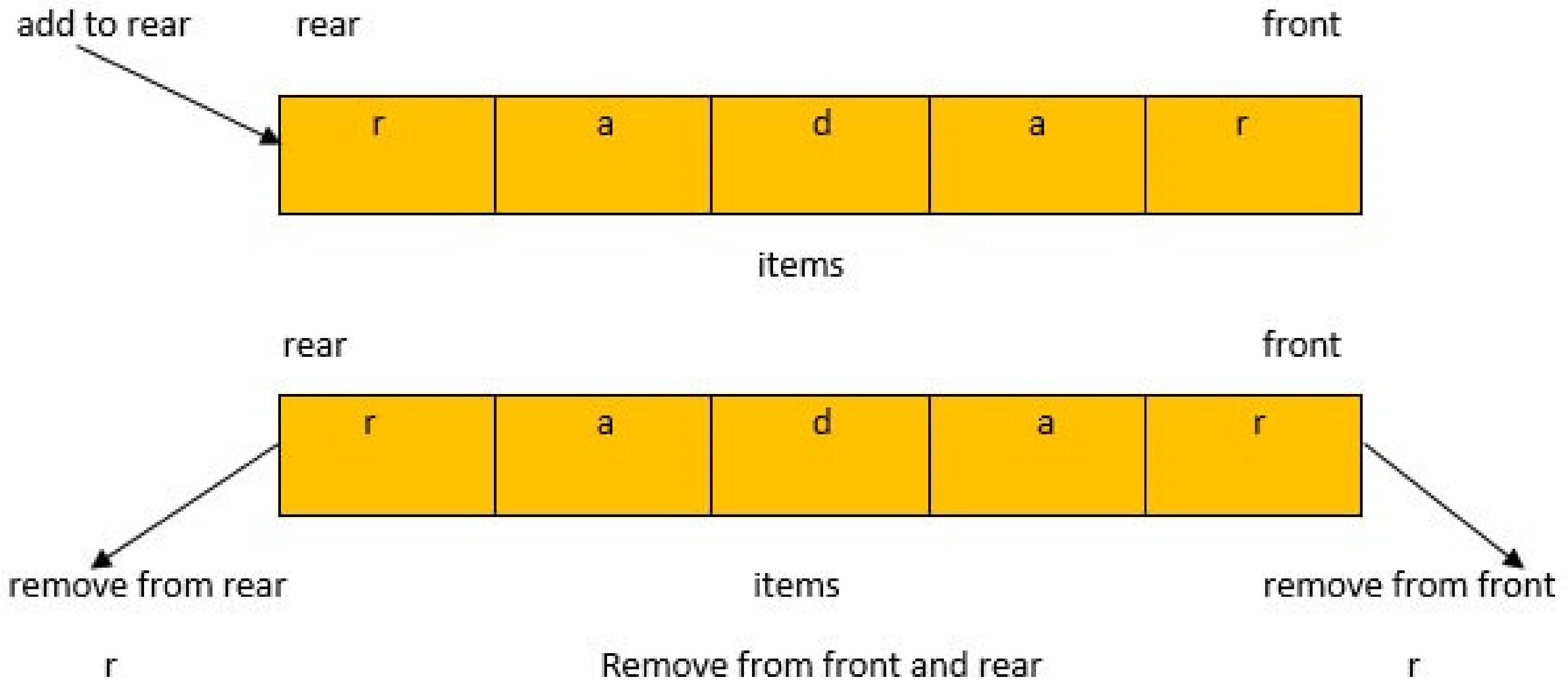


X=M
Y=M
X=Y
....

MADAM is a palindrome !

Using Deque (Dec) for Palindrom Checking

The front of the deque will hold the first character of the string and the rear of the deque will hold the last character. Since we can remove both of them directly, we can compare them and continue only if they match. Example:



Running Time of Queue Operations

Consider simple array implementation:

- A single item can be inserted and deleted from a queue in O(1) time.
- Inserting or deleting n elements in a queue is O(n).

Dynamic Data Structures

- Storage for all elements of a static array is allocated together as one fixed block of memory.
- If insertions and deletions are frequent, array implementation is not a good choice because of the need for several shift operations.
- Shifts at run time are costly (Take time).
- What can be done to overcome disadvantages of the static array implementations?
- The answer is dynamic data structures whose sizes can change at run time.