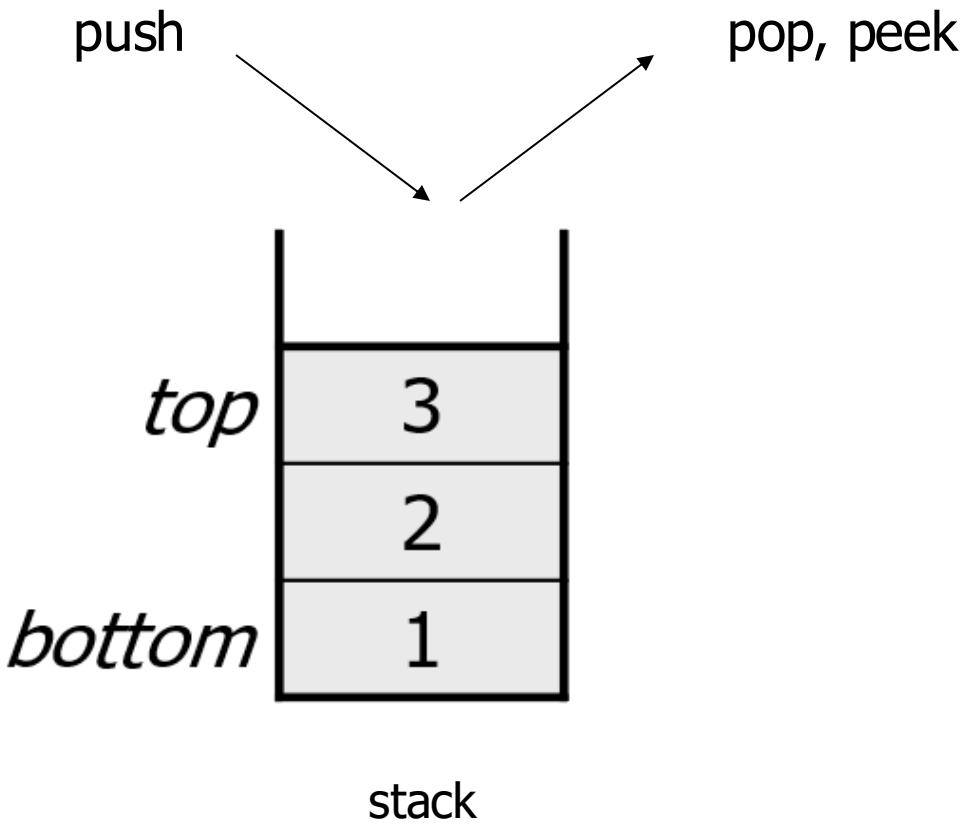
STACKS

STACKS

- **Stack**: It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
  - Client program can only add/remove/examine the last element added (the "top").


push
pop, peek
top
3
2
bottom
1
stack

STACK ADT - SPECS

- **Definitions**: (provided by the user)
  - `MAX_ITEMS`: **Max number of items that might be on the stack**
  - `top` : **Top of the stack**
- **Operations**
  - `makeEmpty()` – **Empty the stack**
  - `boolean IsEmpty()`
  - `boolean IsFull()`
  - `push (ItemType newItem)`
  - `ItType pop ()`
  - `ItemType peek()`

PUSH

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

POP

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and the value is returned to the main program

IMPLEMENTATION OF STACKS

- There are two ways we can implement a stack:
  - Using an array
  - Using a linked list

ARRAY IMPLEMENTATION OF STACK

Implementing a stack using an array is fairly easy.

- Initially `top = -1` , top of the stack. This is also the Empty stack condition.
- Insertion:
  - Increment `top = top+1`
  - Insert new element at `data[top]`
- Deletion:
  - Return the value of `data[top]`
  - Decrement `top =top-1`
- Stack is full when `top == arraysize-1`

OVERFLOW & UNDERFLOW

Stack overflow

- Overflow results from trying to push an element onto a full stack.

```
if(!stack->IsFull())
    Stack->Push(item);
```

Stack underflow

- Underflow results from trying to pop an empty stack.

```
if(!stack->IsEmpty())
    Stack->Pop(item);
```

ARRAY IMPLEMENTATION OF STACK

- **Advantages**
  - Best performance in terms of time, there are no overheads associated with linked lists and the memory requirement is very less.

- **Disadvantage**
  - The size of the stack is fixed.

STACK LIMITATIONS

- You cannot loop over a stack in the usual way.

```
...
for (int i = 0; i < stack_size; i++) {
    do something with stack_get(i);
}
```

- Instead, you pull elements out of the stack one at a time.

```
// process (and destroy) an entire stack
while (!stack.isEmpty()) {
    do something with stack.pop();
}
```

STACK BIG O

- Insertion O(1)
- Deletion O(1)
- Search O(n)
- Access O(n)

EXAMPLES OF STACKS

- Programming languages and compilers:
  - Function calls are placed onto a stack *(call=push, return=pop)*
  - Compilers use stacks to evaluate expressions
  - Syntax Parsing

- Matching up related pairs of things:
  - Find out whether a string is a palindrome
  - Examine a file to see if its braces { } match
  - Convert "infix" expressions to pre/postfix

- Sophisticated algorithms:
  - Searching through a maze with "backtracking"
  - Some programs use an "undo stack" of previous operations

- Graphs Traversal Algorithms
- Undo/Redo Functionality
- Memory Management in VMs

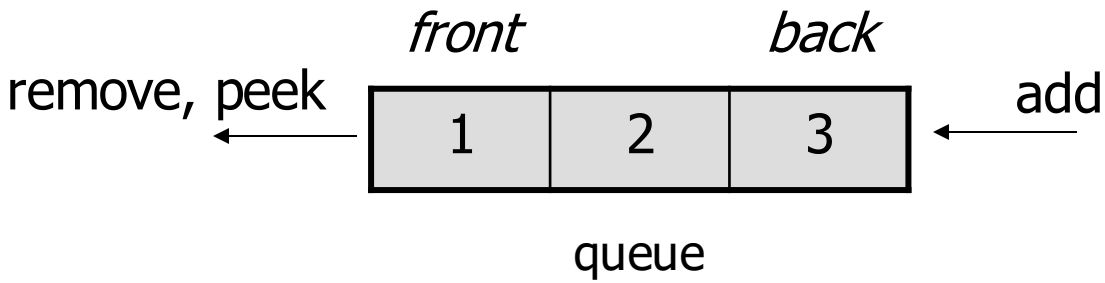
QUEUES

QUEUES

- It is an ordered group of homogeneous items.

- Queues have two ends:
  - Items are added at the rear.
  - Items are removed from the front of the queue

- **FIFO property**: First In, First Out
  - The item added first is also removed first


front
back
remove, peek
1
2
3
add
queue

QUEUE ADT - SPECS

- **Definitions**: (provided by the user)
  - `MAX_ITEMS`: Max number of items that might be on the stack
  - `front` : Front of the Queue
  - `rear` : Rear of the Queue
- **Operations**
  - `makeEmpty`
  - `boolean isEmpty()`
  - `boolean isFull()`
  - `enqueue (ItemType newItem)`
  - `ItemType dequeue()`
  - `ItemType peek()`

ENQUEUE

- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

DEQUEUE

- *Function*: Removes front item from queue and returns it in item.

- *Preconditions*: Queue has been initialized and is not empty.

- *Postconditions*: Front element has been removed from queue and the value is returned to the main program

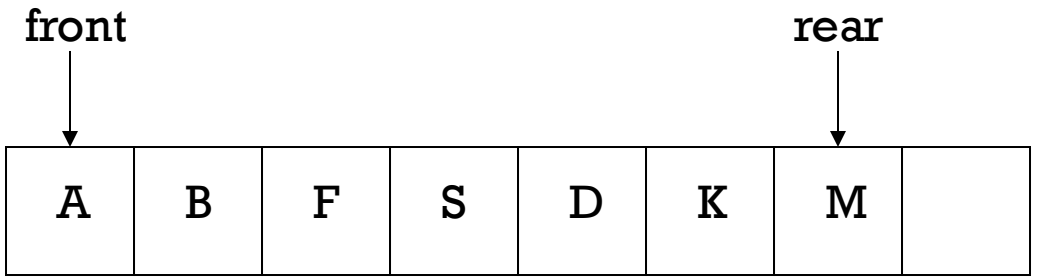
IMPLEMENTATION OF QUEUE

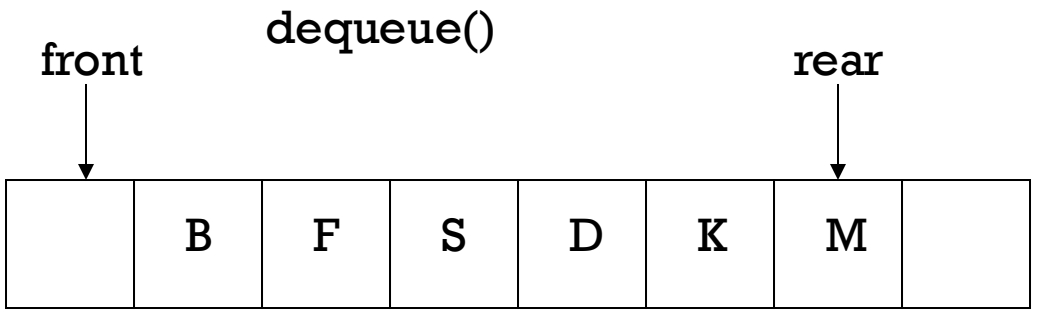
- Using an array to implement a queue is significantly harder than using an array to implement a stack.

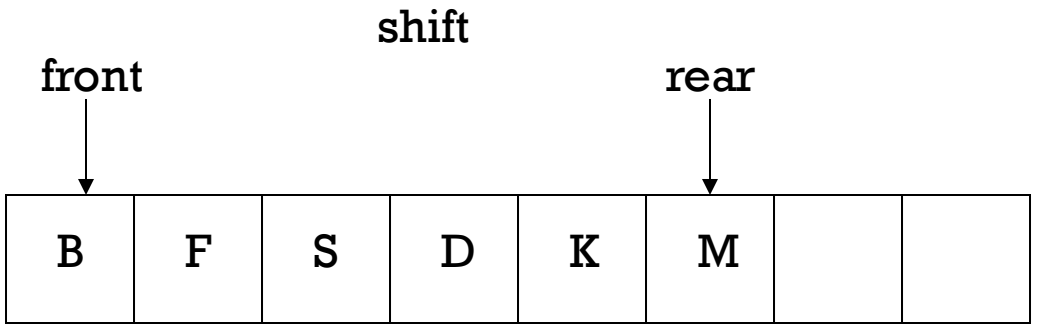
- Why?
  - Unlike a stack, where we add and remove at the same end, in a queue we add to one end and remove from the other.

LINEAR REGULAR ARRAY

- In a standard linear array
  - `front = 0`
  - `rear = MAX_ITEMS- 1`
- All objects will be dequeued from `data[front]`
- All objects will be enqueued after `data[rear]`
- To implement this, when an object is removed, all elements must be shifted down by one spot







front remains unchanged
rear is decremented by 1

DISADVANTAGES OF REGULAR LINEAR ARRAY

- Very expensive structure to use for a queue
  Shifting all the data down by one is very time consuming
  If the data is not shifted, even with empty spaces when `rear = array.size -1` and `front = array.size` no insertion can be done.

Option 2:

- Data "wrap-around" is done inside the array
  - Front would not always be zero
    - it would be the first occupied cell
  - The last item may appear in the array before the first item
  - This is called a ***circular array***

CIRCULAR ARRAY

- Initially `front = -1` and `rear = -1`
- **Insertion**:
  - rear is incremented by 1
  - If `rear == arraysize-1` the make `rear = 0` (this is the wrap around condition)
  - If `front==-1` change `front=0` (initially empty queue)
- **Deletion**:
  - Element at `data[front]` is deleted
  - front is increment by 1
  - If `front == arraysize-1` , make `front = 0` (this is the wrap around condition)

CIRCULAR ARRAY

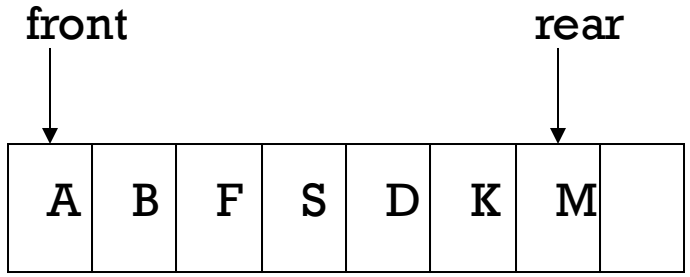
- If `front == rear`, there is only one element in the queue

  (except when front and rear = -1)

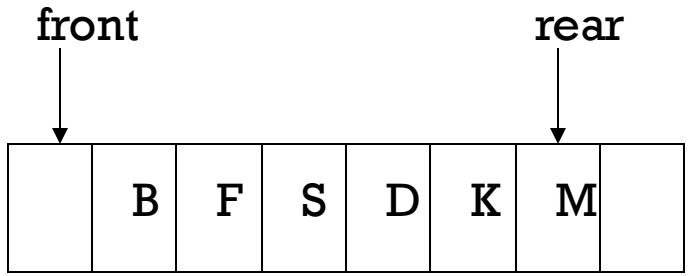
  Set `front = -1` and `rear = -1` when `dequeue()` is performed with this condition

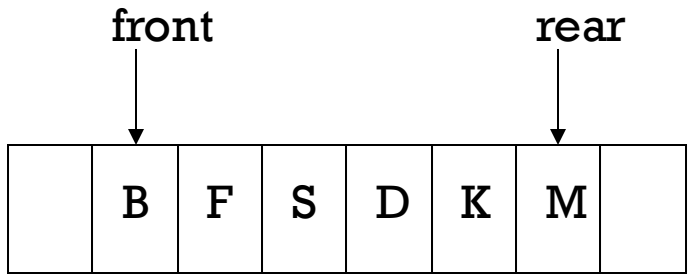
- **Queue Empty cases:** `front == -1`
- **Queue is full cases:**
  - `front ==rear+1`
  - `front =0` and `rear=arraysize-1`
- **Display cases:**
  - `If (front <=rear)` display `data[front]` to `data[rear]`
  - `If (front>rear)` display `data[front]` to `data[arraysize-1]` then `data[0]` to `data[rear]`

Dequeue from data[front], increment front by 1.

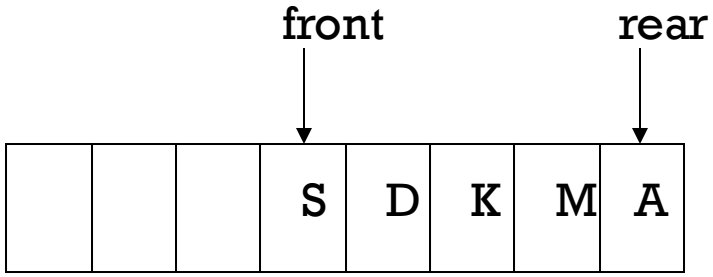


dequeue()

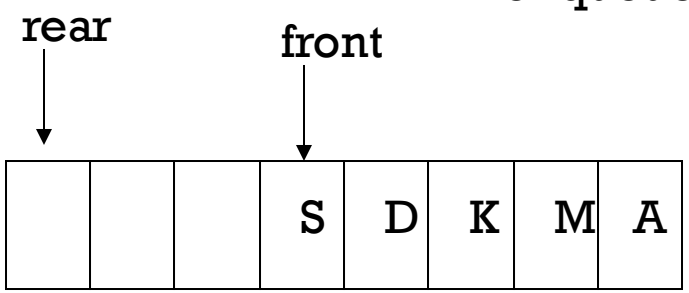


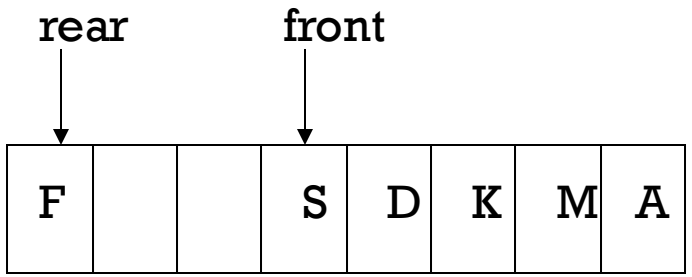


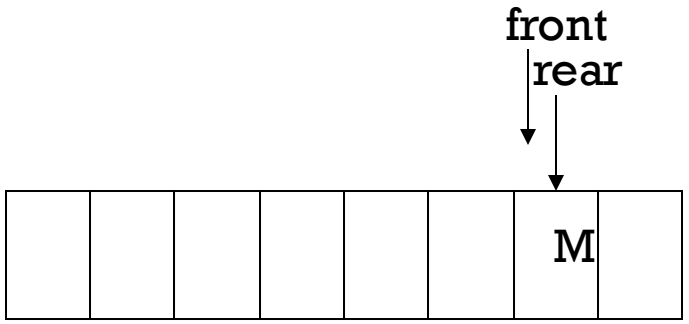
When rear is arraysize-1, make rear=0. Warp Around condition.



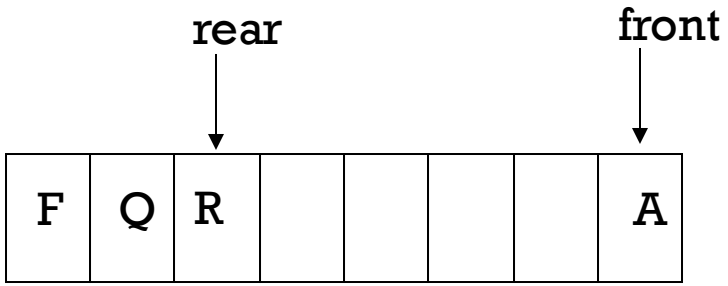
enqueue(F)



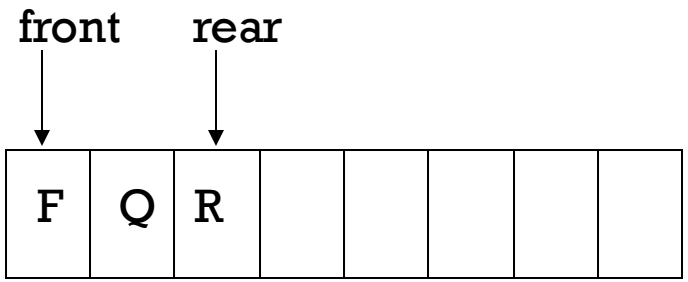




When front == rear and both are not equal to -1, dequeue empties the queue.



dequeue()



When front is arraysize-1, make front=0. Warp Around condition.

OVERFLOW & UNDERFLOW

- **Queue Overflow**:

  This results from trying to add an element onto a full queue.

```
if(!q.IsFull())
      q.Enqueue(item);
```

- **Queue Underflow**:

  This results from trying to remove an element from an empty queue.

```
if(!q.IsEmpty())
   q.Dequeue(item);
```

QUEUE BIG O

- Insertion O(1)
- Deletion O(1)
- Search O(n)
- Access O(n)

EXAMPLES OF QUEUES

- **Operating systems:**
  - Print jobs sent to the printer
  - Programs / processes to be run
  - 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 at a service counter