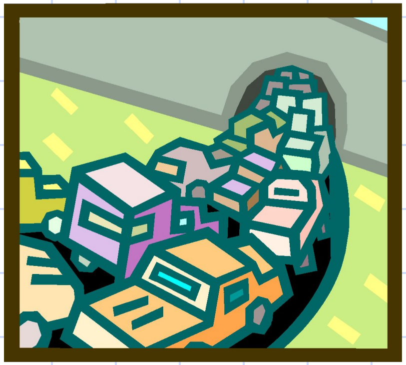
Queue

Queues

- What is a queue?
  - A data structure of ordered items such that items can be inserted only at one end and removed at the other end.
- Example
  - A line at the supermarket

Applications of Queues



- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Applications of Queues

- Processing inputs and outputs to screen (console)
- Server requests
  - Instant messaging servers queue up incoming messages
  - Database requests
- Print queues
  - One printer for dozens of computers
- Operating systems use queues to schedule CPU jobs

Queues

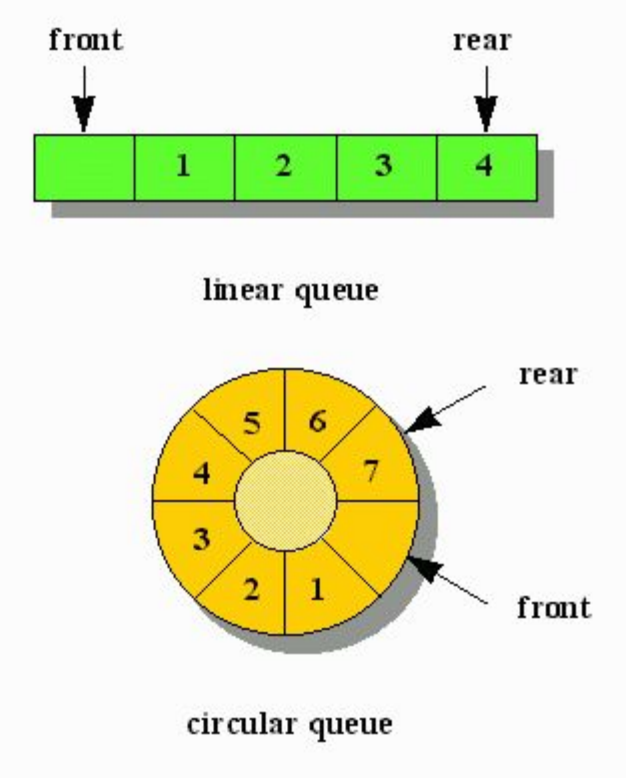
- What can we do with a queue?
  - Enqueue - Add an item to the queue
  - Dequeue - Remove an item from the queue

Queue (FIFO)

- Queue is a linear data structure, in which insertion of a new elements elements takes place from the rear end and deletion takes place from front end.
- It is also called First in First out (FIFO) data structure.
- On insertion of element ----

  rear = rear +1
- On deletion of element

  front = front + 1



Queue Operations

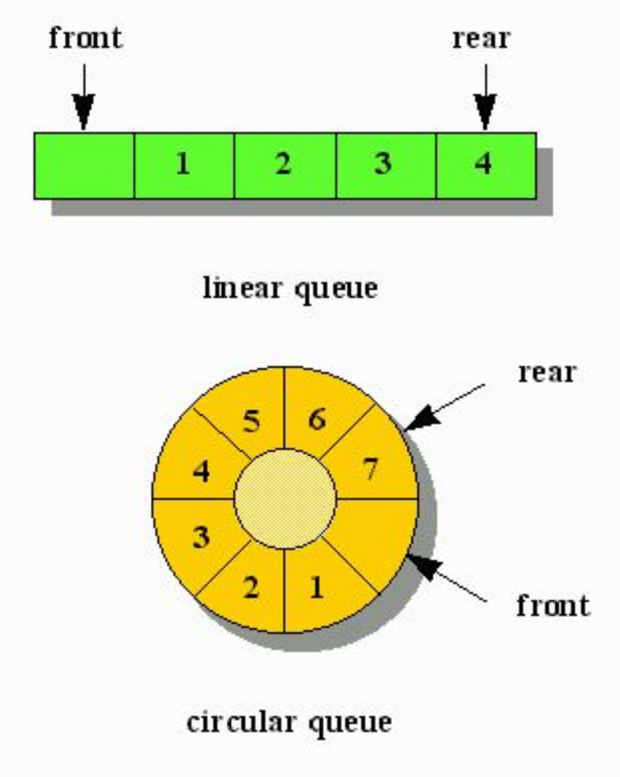
- *Initialize* the queue
- *Insert* to the rear of the queue
- *Remove* (Delete) from the front of the queue
- Is the Queue Empty
- Is the Queue Full
- What is the size of the Queue

Queue Operations

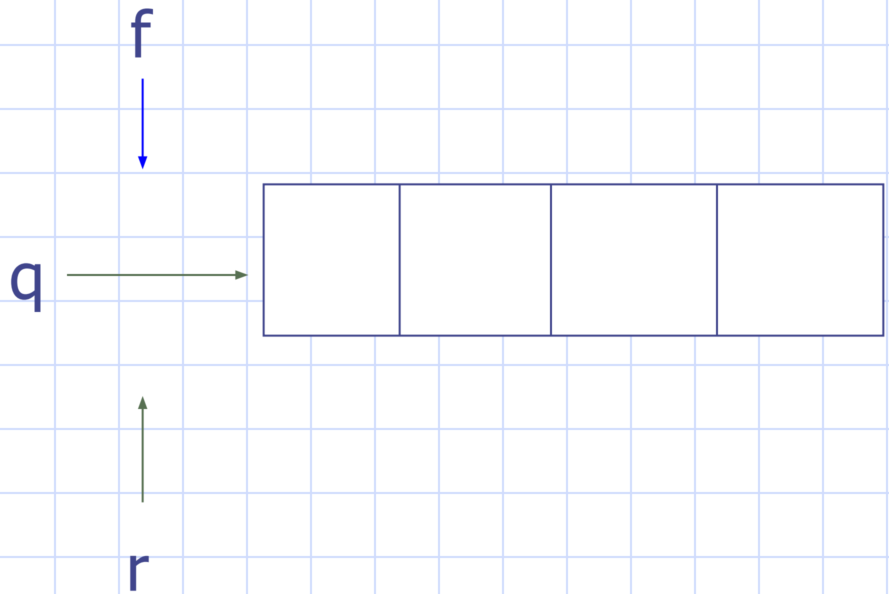
- The abstract operations on a queue include —
  - Enqueue(x,q) — Insert item x at the back of queue q.
  - Dequeue(q) — Return (and remove) the front item from queue q
  - Initialize(q), Full(q), Empty(q) — Analogous to these operation on stacks.

Queues

- The simplest implementation uses an array, inserting new elements at one end and moving all remaining elements to fill the empty space created on each dequeue.
- However, it is very wasteful to move all the elements on each dequeue.
- Circular queues let us reuse this empty space.



Implementing a queue q using an array



(a)

The method

- q.queue_init()

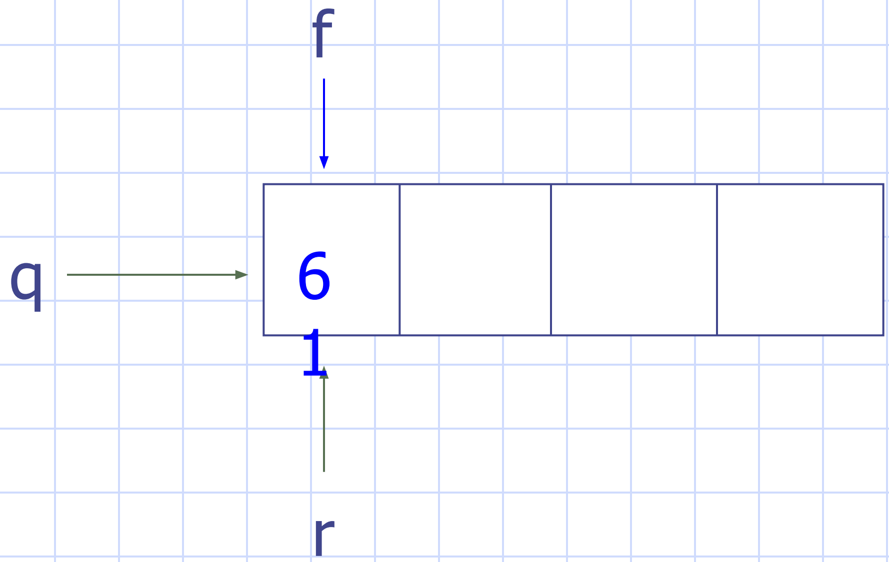
initializes an emty queue.

The rear of the queue is at index r, and the front of the queue is at index f .

An empty queue has

r = f = -1.

Implementing a queue q using an array of size 4



(b)

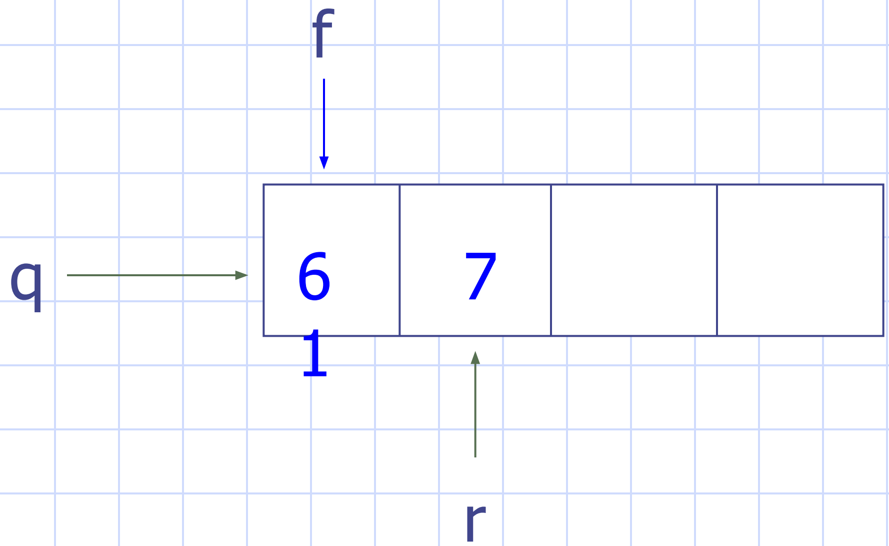
After

- q.enqueue(61)

We have the situation shown in (b).

Since the first index is 0, we have $r = f = 0$.

Implementing a queue q using an array of size 4



(c)

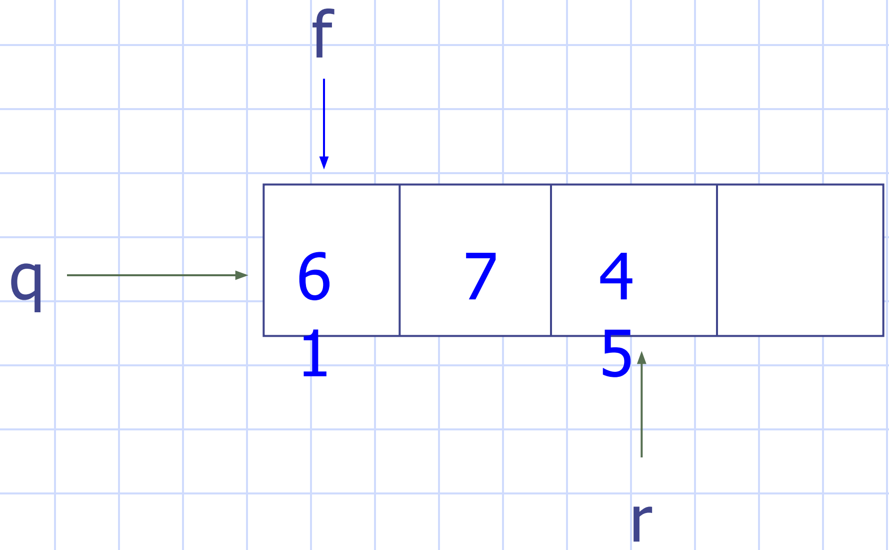
After

- q.enqueue(7)

We have the situation shown in (c).

Since the first index is 0, we have r=1, f=0.

Implementing a queue q using an array of size 4



(d)

After

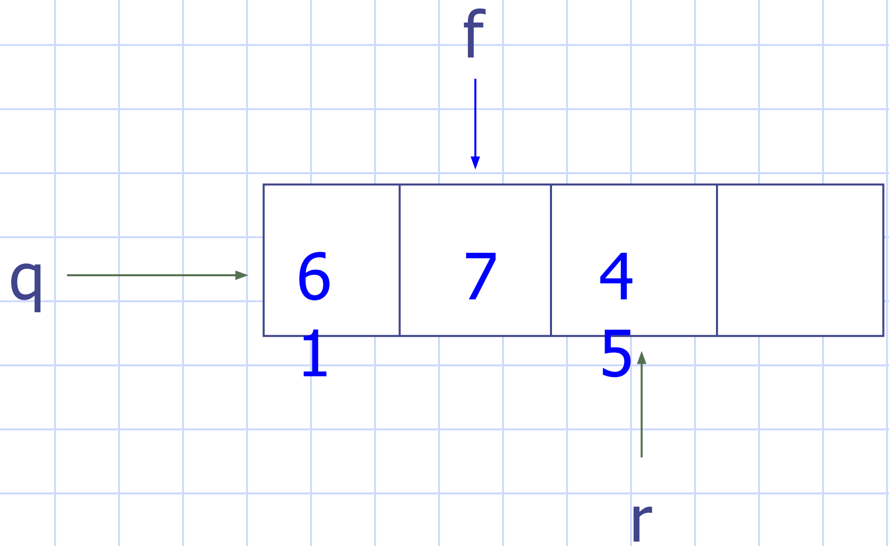
- q.enqueue(45)

We have the situation shown in (d).

Since the first index is 0, we have

r=2, f=0.

Implementing a queue q using an array of size 4



(e)

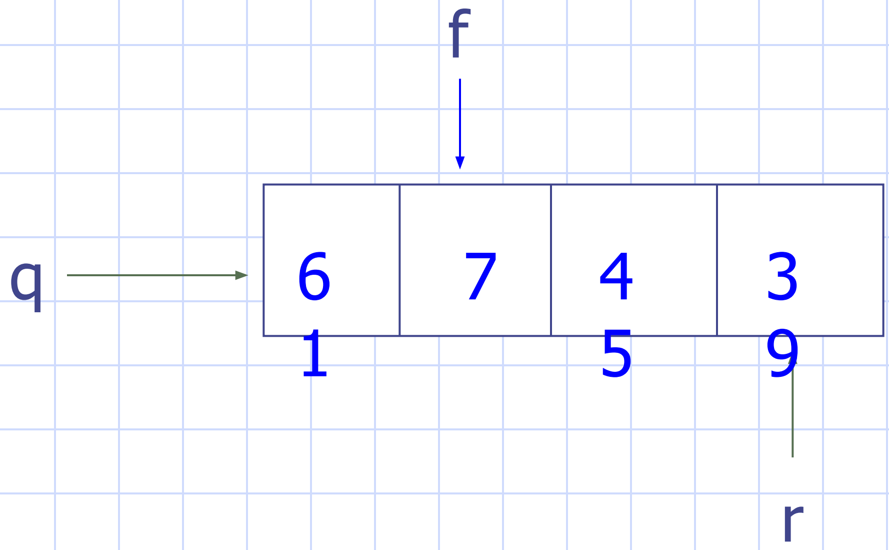
After

- q.dequeue()

We have the situation shown in (e).

Since the first index is 0, we have r=2, f=1.

Implementing a queue q using an array of size 4



After

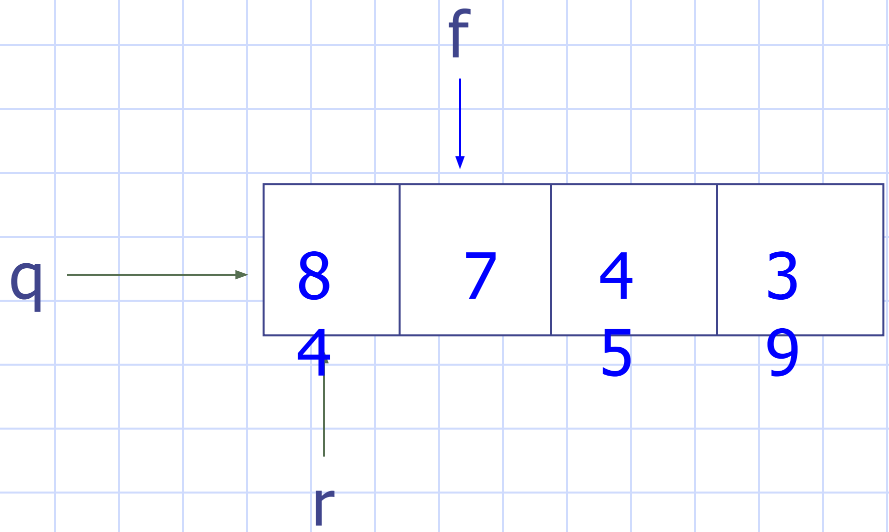
- q.enqueue(39)

We have the situation shown in (f).

Since the first index is 0, we have r=3, f=1.

(f)

Implementing a queue q using an array of size 4



(g)

After

- q.enqueue(84)

We have the situation shown in (g).

Since the first index is 0, we have

r=0, f=1.

Queue Operations: Insert

- QINSERT(Q, R, F, N, Y)

```
1.    [Overflow?]
   if R >= N
          then Write ('Overflow')
          Return
 2. [Increment rear Pointer]
      R □ R + 1
 3. [Insert Element]
      Q[R] □ Y
 4. [Is front pointer properly set?]
      If F=0
          then F □ 1
      Return
```

Queue Operations: Delete

- QDELETE(Q, F, R)

1. [Underflow]
   if F = 0
      then Write ('Underflow')
         Return (0)   (0 Denotes an empty queue)
2. [Delete Element]
   Y ☐ Q[F]
3. [Queue Empty?]
   If F = R
      then F ☐ R ☐ 0
      else F ☐ F + 1
3.[Return element]
   Return (Y)

Declaration of a Queue

```
# define MAXQUEUE 50 /* size of the queue items*/
typedef struct {
    int front;
    int rear;
    int items[MAXQUEUE];
}QUEUE;
```

queue_arr

Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- There is a simple way to use a singly-linked list to implement both insertions and deletions
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list
  - Linked list structure for queue

```
struct Queue
 {  int item;
     struct Queue *next
 }
```

SLL implementation of queues

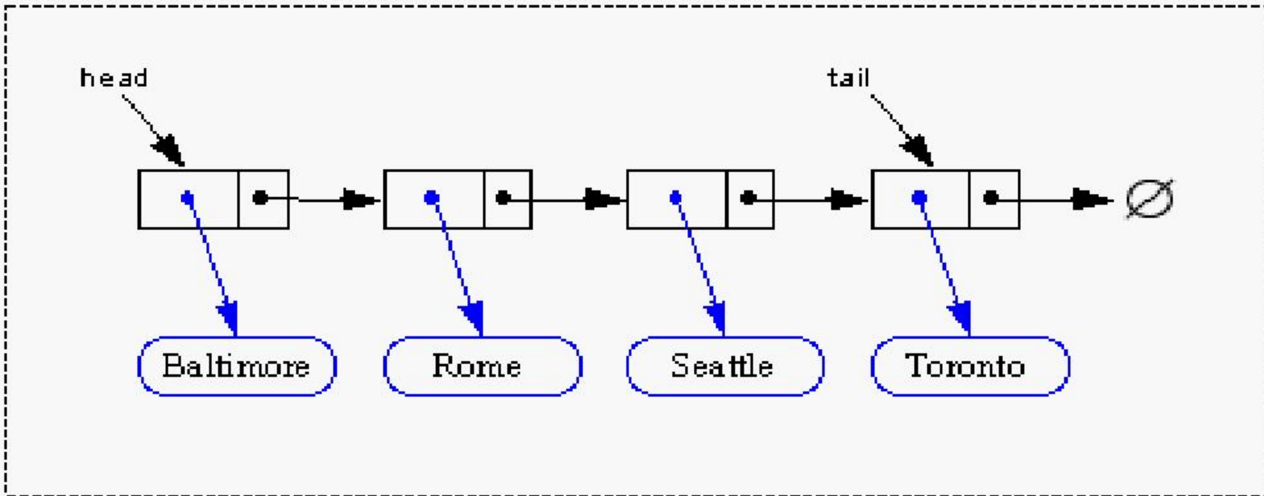
- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to null
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to null

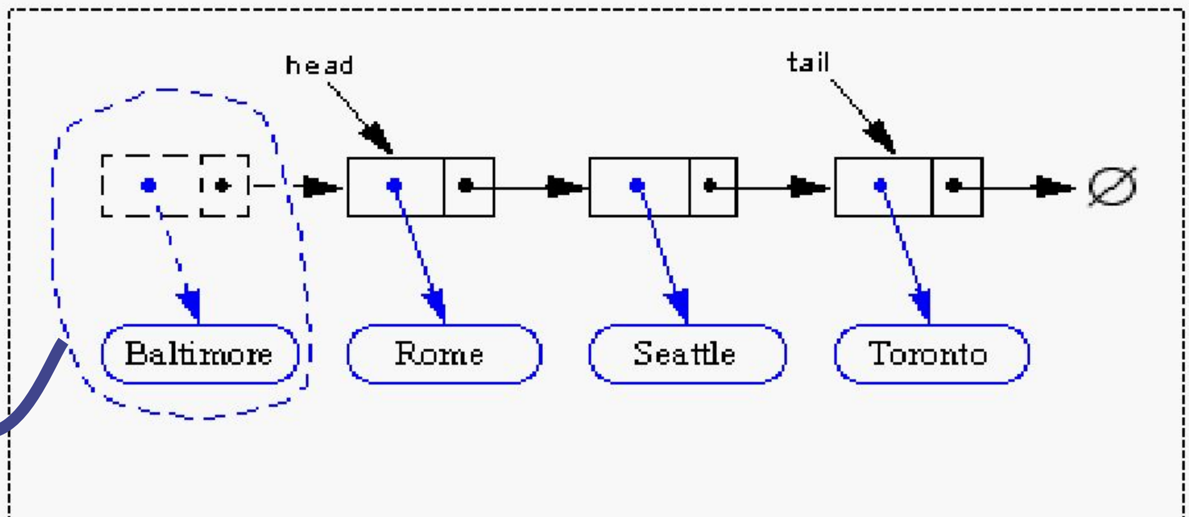
Removing at the Head

1.



- advance head reference

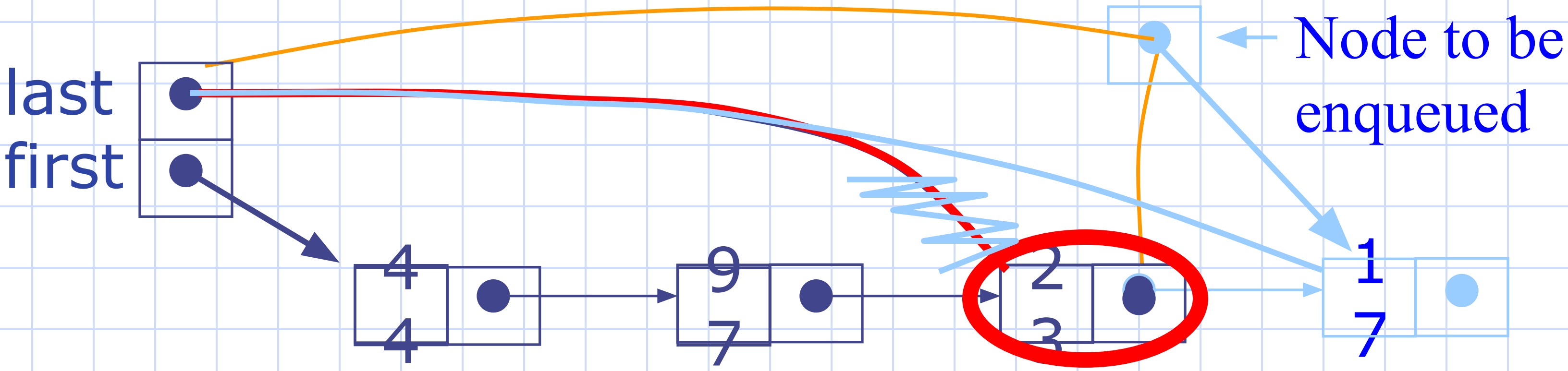
2.



removed item

- inserting at the head is just as easy

Enqueueing a node



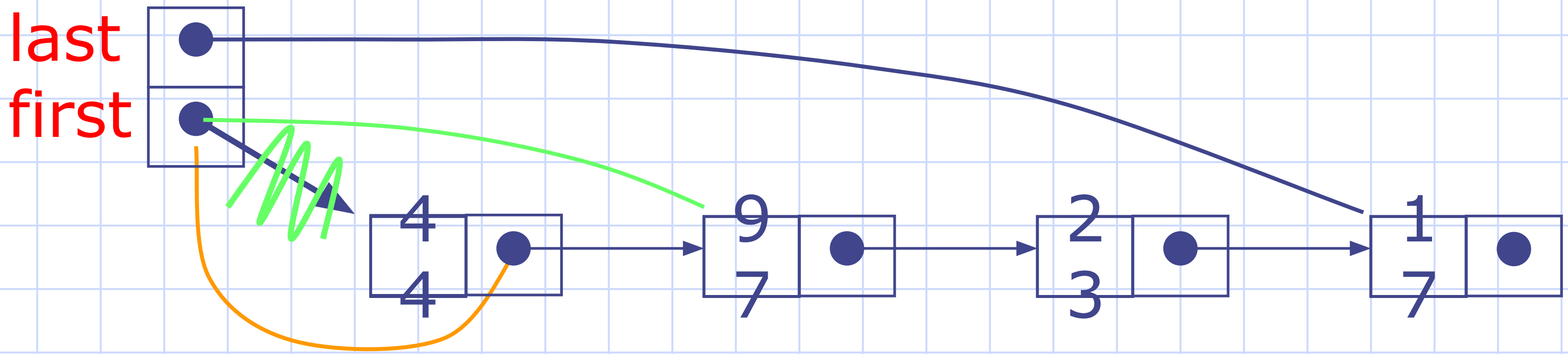
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the last pointer in the list header

Dequeueing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

[Link queue]()

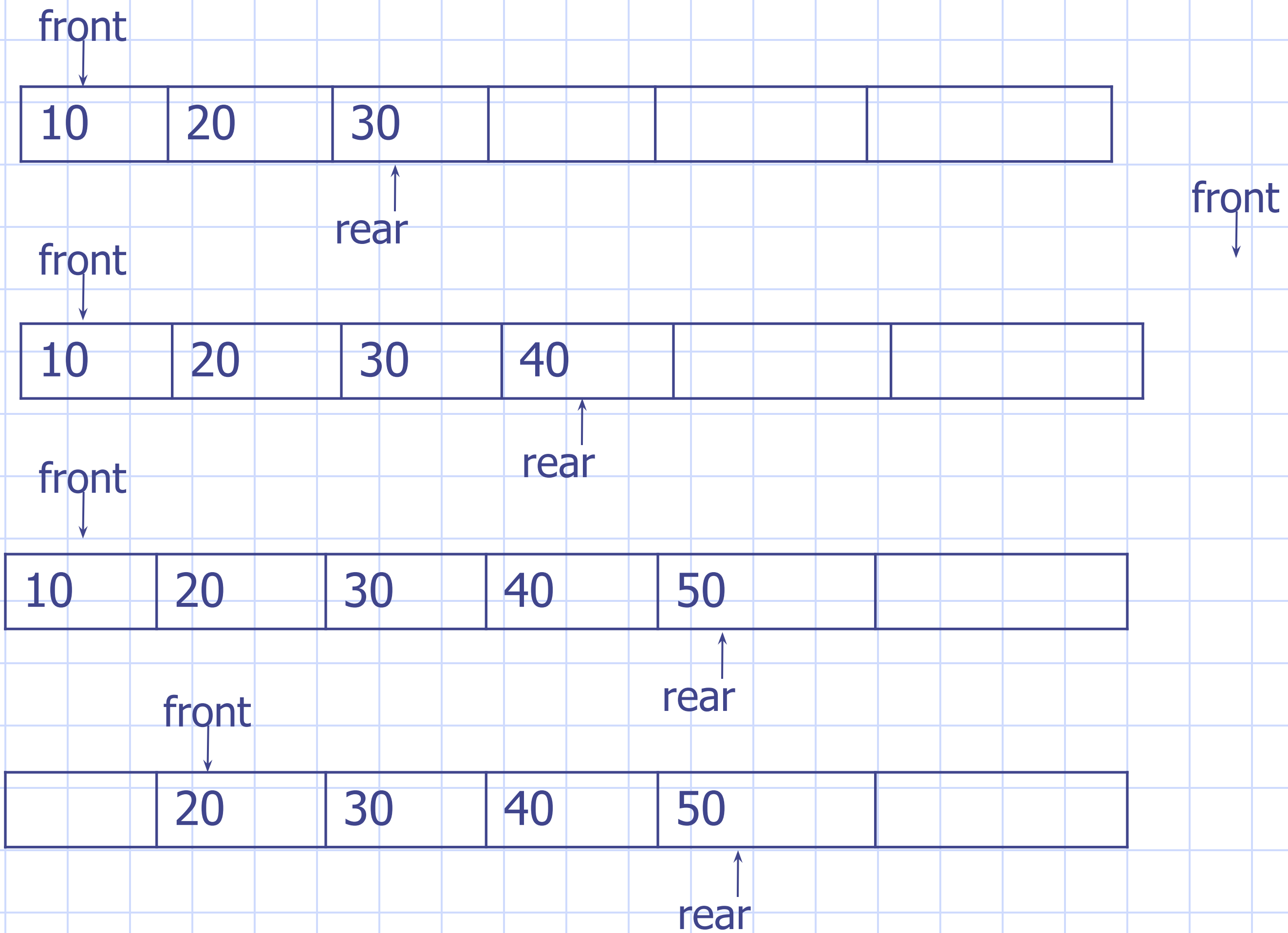
Circular queue :Conceptual form

Insert 30 at rear

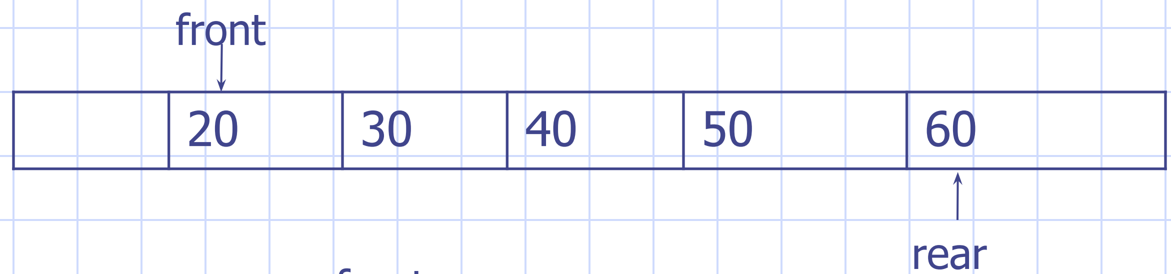
Insert 40 at rear

Insert 50 at rear

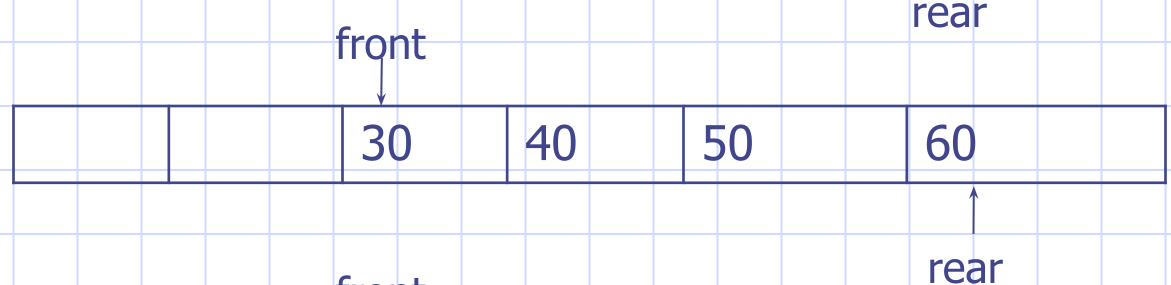
Delete from front



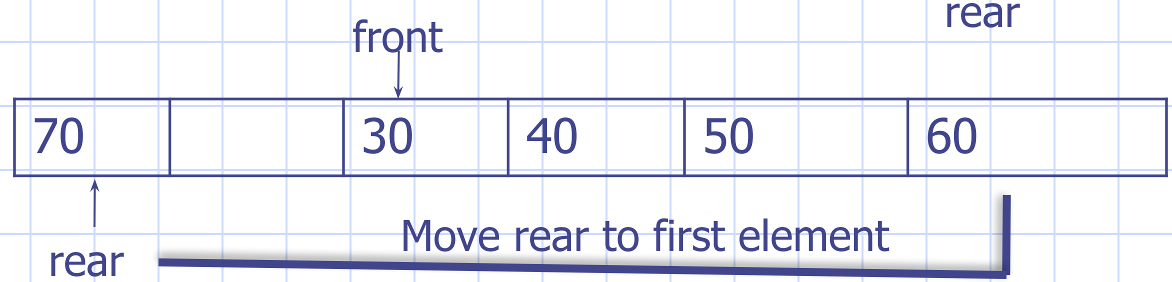
Insert 60 at rear



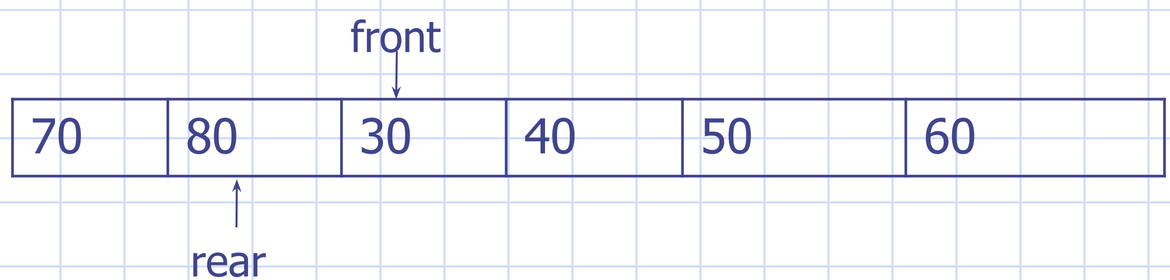
Delete from front

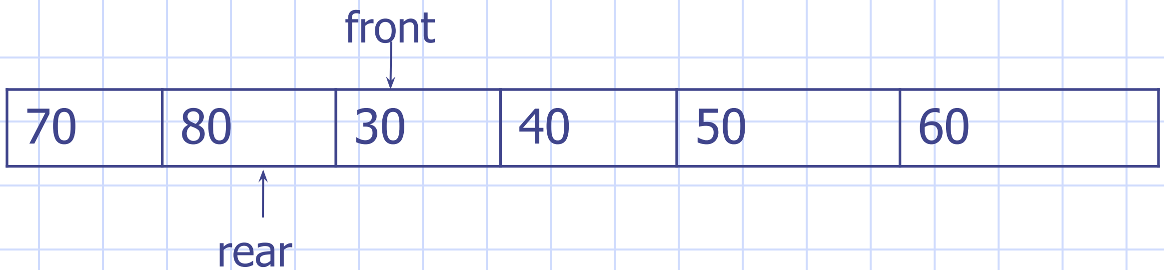


Insert 70 at rear



Insert 80 at rear





- rear =front -1   i.e front = 2 rear = 1 .. Then queue is over flow during insertion
- If front==rear that is single element into array delete and set front=rear=-1

Circular Queue Operations: Insert

- CQINSERT(Q, R, F, N, Y)

1. [Reset rear pointer]
   If R=N
   then R ☐ 1
   else R ☐ R +1
2. [Overflow?]
   if R = F-1
   then Write ('Overflow')
   Return
3. [Insert Element]
   Q[R] ☐ Y
4. [Is front pointer properly set?]
   If F=0
   then F ☐ 1
   Return

1. [Overflow?]
   if R >= N
   then Write ('Overflow')
   Return
2. [Increment rear Pointer]
   R ☐ R + 1
3. [Insert Element]
   Q[R] ☐ Y
4. [Is front pointer properly set?]
   If F=0
   then F ☐ 1
   Return

Circular Queue Operations: Delete

- CQDELETE(Q, F, R, N)

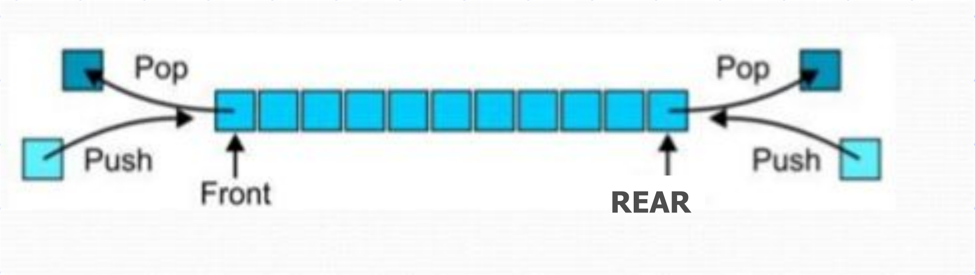
1. [Underflow?]
if F = 0
then Write (‘Underflow’)
Return (0)
2. [Delete Element]
Y □ Q[F]
3. [Queue Empty?]
If F = R
then F □ R □ 0
Return Y
3.[Increment Front Pointer]
If F = N
then F □ 1
else F □ F +1
Return (Y)

1. [Underflow]
if F = 0
then Write (‘Underflow’)
Return (0)
2. [Delete Element]
Y □ Q[F]
3. [Queue Empty?]
If F = R
then F □ R □ 0
else F □ F + 1
3.[Return element]
Return (Y)

Circular queue

Deques

- A deque is double-ended queue, in which elements can be added or removed at either end but not in middle



- There are two variations of a deque:

  1) Input-restricted deque

  - Elements can be inserted in one ends.

    Element can be remove from both the end.

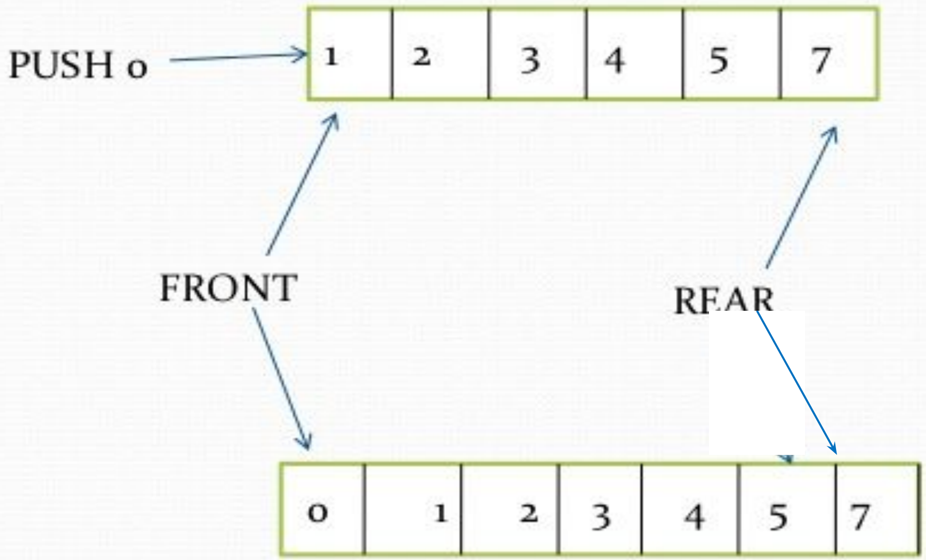
2) Output-restricted deque

- Element can be removed from only one end.
- Element can be inserted from both the end.

- Operations in Deque
  - Insert element at back
  - Insert element at front
  - Delete element from back
  - Delete element from front

Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



Implementing insertion operation

- Algo for insert_front

1. Check queue is full from front or not(front==0)
2. If not then front=front-1
3. Insert the element at dq[front] position

- Algo for insert_back

1. Check queue is full from back or not (rear=size-1)
2. If not then update rear=rear+1;
3. Insert the element at position dq[rear];

Implementing insertion operation

- Algo for delete_front

1. Check queue is empty or not(front == -1)
2. If not then Perform deletion y = dq[front]

   2.1 if front == rear then front=rear = -1

   else

   Front = front + 1

- Algo for delete_back

1. Check queue is empty or not (rear==-1)
2. If not then perform deletion y = dq[rear]

   2.1 if front==rear then front=rear=-1;

   else

   rear = rear-1;

Priority Queue

- A priority queue is a collection of elements such that each element is assigned a priority in that order elements are deleted or inserted on priority basis.
  - The highest priority entry is removed first
  - Entries with equal priority can be removed according some criterion e.g. FIFO as an queue.

Priority Queues

- Queue:
  - First in, first out
  - First to be removed: First to enter

- Priority queue:
  - First in, Largest out
  - First to be removed: Highest priority

  - Operations: *Insert(), Remove-top()*

Priority Queue Implementation Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMax and max take $O(n)$ time since we have to traverse the entire sequence to find the largest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMax and max take $O(1)$ time, since the largest key is at the end

Priority Queue Applications

- Process scheduling
  - Give CPU resources to most urgent task
- Communications
  - Send most urgent message first

Types of priority queues

- Ascending priority queue
  - Removal of minimum-priority element
  - *Remove-top():* Removes element with min priority
- Descending priority queue
  - Removal of maximum-priority element
  - *Remove-top():* Removes element with max priority

Representing a priority queue (I)

<u>linked-list, with head pointer</u>

- Insert()
  - Search for appropriate place to insert
  - O(n)

- Remove()
  - Remove first in list
  - O(1)

Representing a priority queue (II)

Heap: Almost-full binary tree with heap property

- Almost full:
  - Balanced (all leaves at max height *h* or at *h-1*)
  - All leaves to the left
- Heap property: Parent >= children (descending)
  - True for all nodes in the tree
  - Note this is very different from binary search tree (BST)