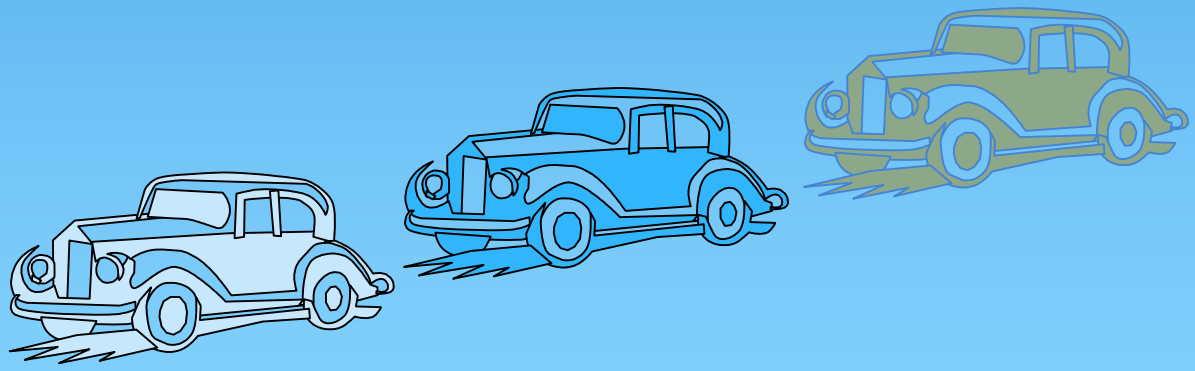
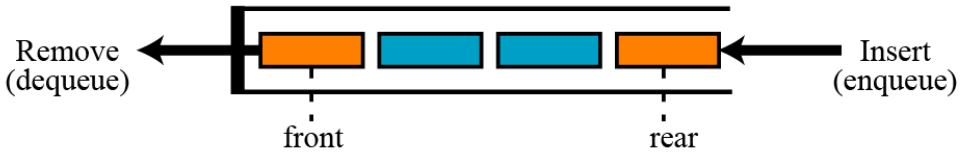
Lecture 5: Queues



Queues

- A data structure of ordered items such that items can be inserted only at one end and removed at the other end.
- A queue is called a FIFO (First in-First out) data structure.
- Insertions are at the rear of the queue and removals are at the front of the queue.

A queue of people



A computer queue

Two representations of queues

Applications of Queues

- Direct applications
  - Waiting lines
  - Round-robin scheduling in processors
  - Input/Output processing
  - Queueing of packets for delivery in networks
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - All types of customer service softwares (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Applications of Queues

Task Scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Main Queue Operations

- Main Queue Operations
  - **enqueue(object):** inserts an element at the end of the queue
  - **object dequeue():** removes and returns the element at the front of the queue

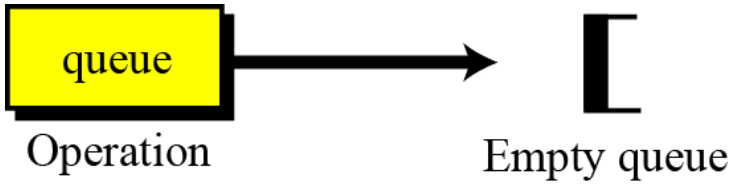
- Auxiliary queue operations:
  - **object front():** returns the element at the front without removing it
  - **integer size():** returns the number of elements stored
  - **boolean isEmpty():** indicates whether no elements are stored

- Exceptions
  - **Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException**.

Main Queue Operations

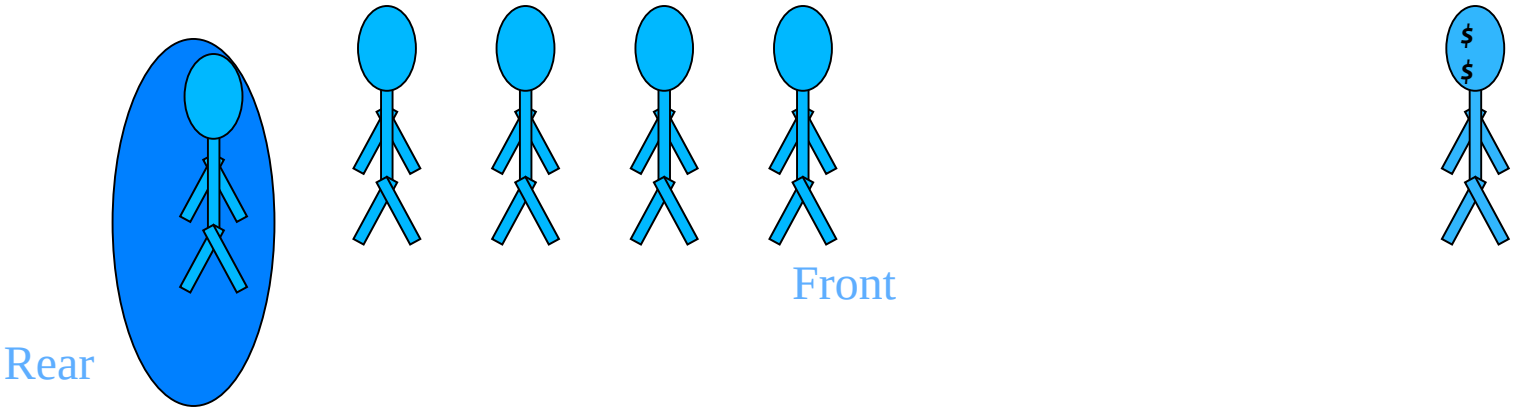
- Creates an empty queue

queue (queueName)

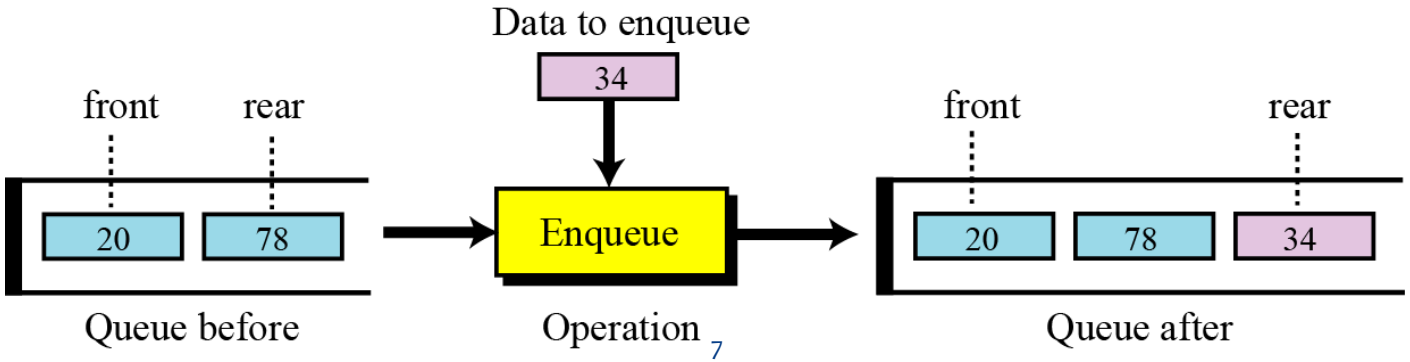


enqueue operation

- Inserts an item at the rear of the queue

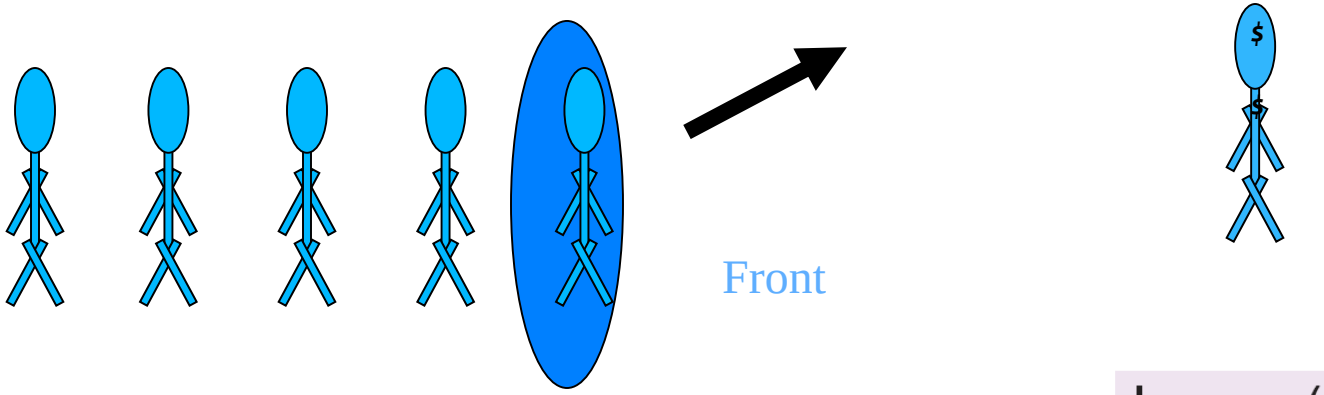


enqueue (queueName, dataItem)

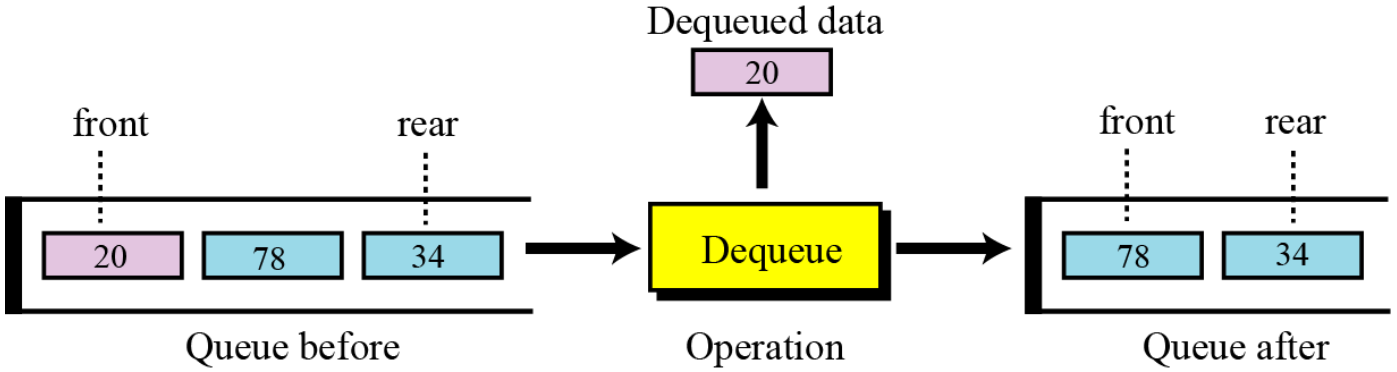


dequeue operation

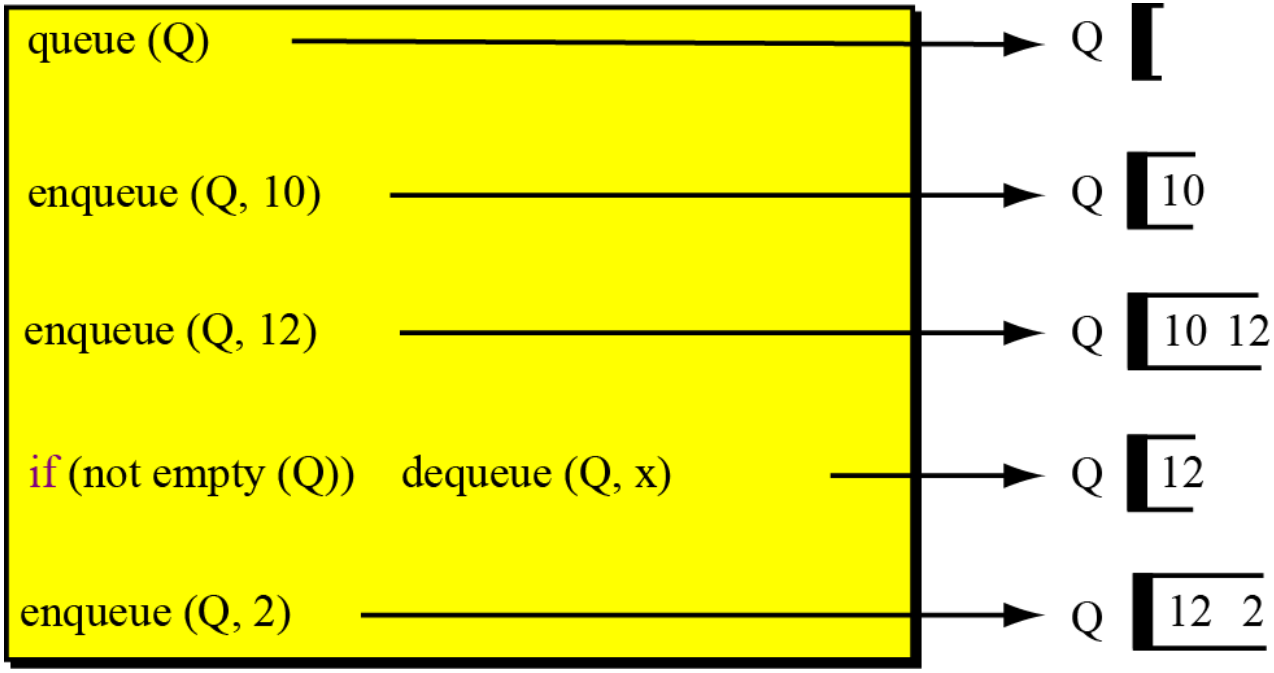
- Deletes the item at the front of the queue



dequeue (queueName, dataItem)

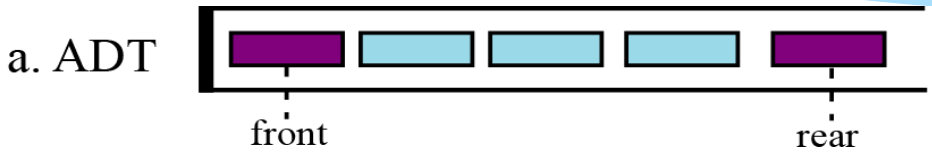


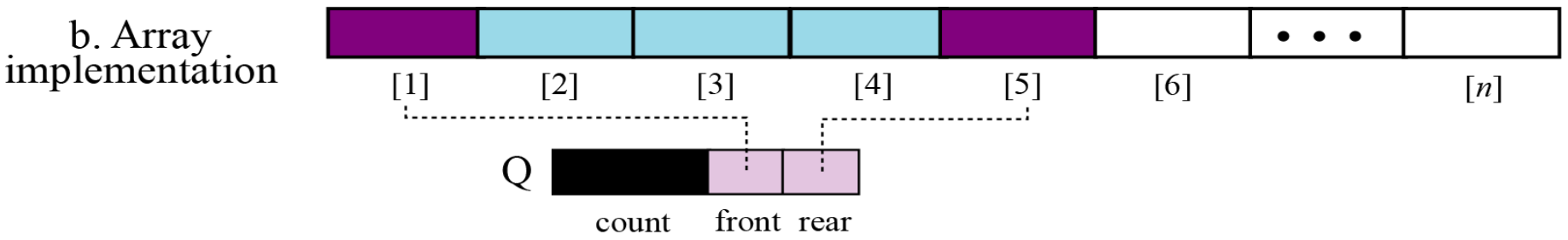
Example

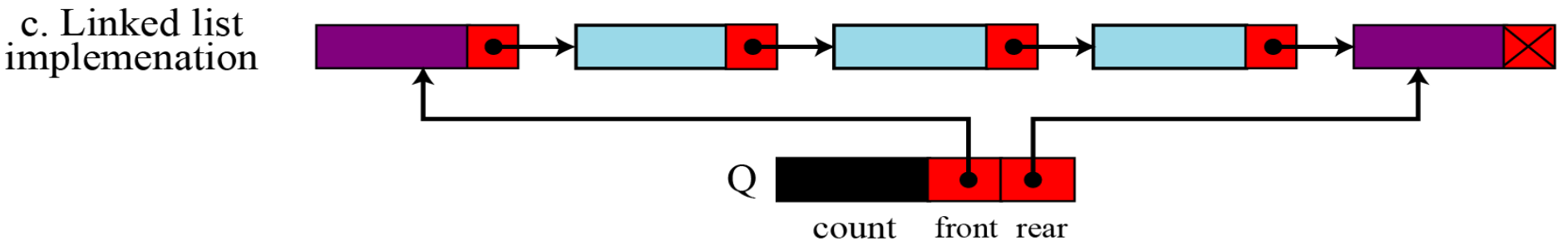


An algorithm segment

Implementing a Queue



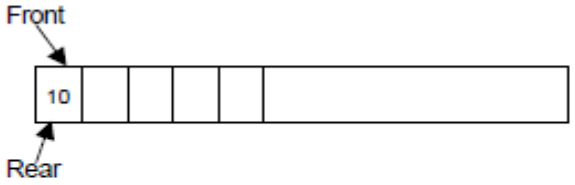




Example(Array Implementation)

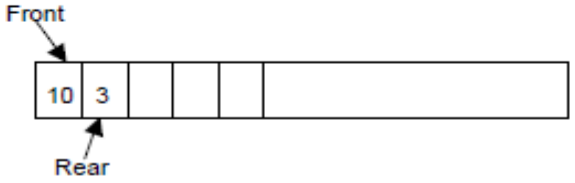
Rear = –1
Front = –1

Fig. 4.1. Queue is empty.



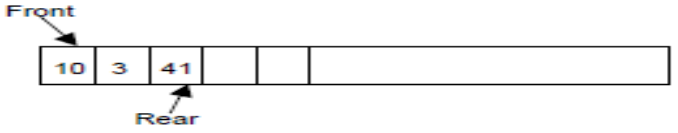
Rear = 0
Front = 0

Fig. 4.2. push(10)



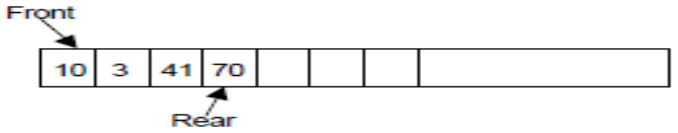
Rear = 1
Front = 0

Fig. 4.3. push(3)



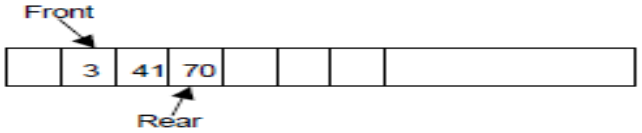
Rear = 2
Front = 0

Fig. 4.4. push(41)



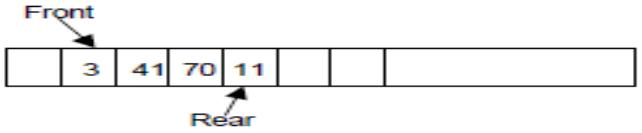
Rear = 3
Front = 0

Fig. 4.5. push(70)



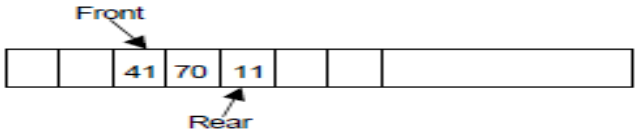
Rear = 3
Front = 1

Fig. 4.6. x = pop() (*i.e.*; x = 10)



Rear = 4
Front = 1

Fig. 4.7. push(11)



Rear = 4
Front = 2

Fig. 4.8. x = pop() (*i.e.*; x = 3)

ENQUEUE()

1. Initialize front=0 & rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)
   (*a) Display “Queue overflow”*
   (*b) Exit*
4. Else
   (*a) Rear = rear +1*
5. Q[rear] = data
6. Exit

ENQUEUE()

```
1.  //This function will insert an element to the queue
2.  void insert ()
3.  {
4.  int added_item;
5.  if (rear==MAX-1)
6.  {
7.  printf("\nQueue Overflow\n");
8.  getch();
9.  return;
10. }
11. else
12. {
13. if (front==–1)                    /*If queue is initially empty */
14. front=0;
15. printf(“\nInput the element for adding in queue: ”);
16. scanf(“%d”, &added_item);
17. rear=rear+1;
18. //Inserting the element
19. queue_arr[rear] = added_item ;
20. }
21. }/*End of insert()*/
```

DEQUEUE()

1. If (rear< front)
   (*a) Front = 0, rear = –1*
   (*b) Display "The queue is empty"*
   (*c) Exit*
2. Else
   (*a) Data = Q[front]*
3. Front = front +1
4. Exit

DEQUEUE()

```
1.  //This function will delete (or pop) an element from the queue
2.  void del()
3.  {
4.  if (front == –1 || front > rear)
5.  {
6.  printf ("\nQueue Underflow\n");
7.  return;
8.  }
9.  else
10. {                                          //deleteing the element
11. printf ("\nElement deleted from queue is : %d\n",
12. queue_arr[front]);
13. front=front+1;
14. }
15. }/*End of del()*/
```

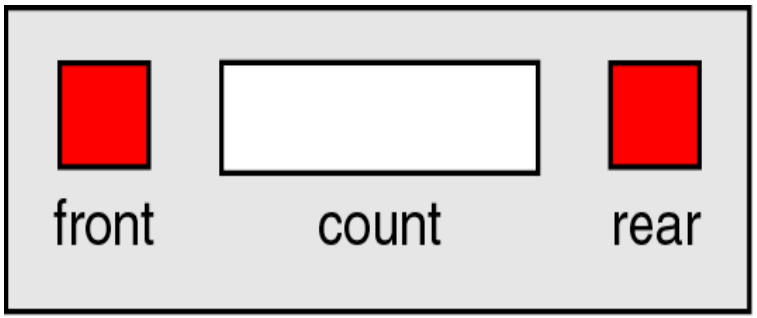
Program to display all queue elements

```
1.  void display()
2.  {
3.  int i;
4.  if (front == –1 || front > rear)          //Checking whether the queue is empty or not
5.  {
6.  printf (“\nQueue is empty\n”);
7.  return;
8.  }
9.  else
10. {
11. printf(“\nQueue is :\n”);
12. for(i=front;i<= rear;i++)
13. printf(“%d ”,queue_arr[i]);
14. printf(“\n”);
15. }
16. }/*End of display() */
```

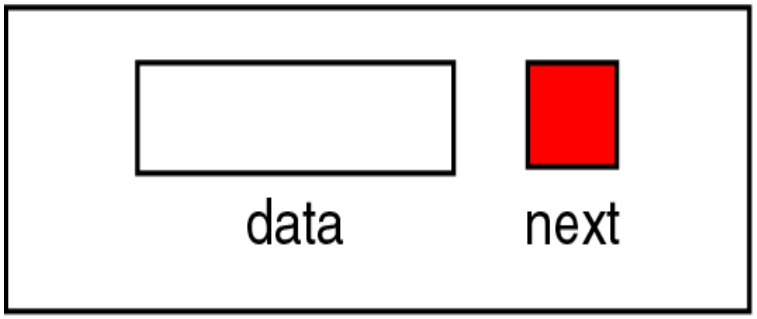
Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in O(1) time
  - You always need a pointer to the first element in the list
  - You can keep an additional pointer to the *last* element in the list
- 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

SLL implementation of queues



Head structure

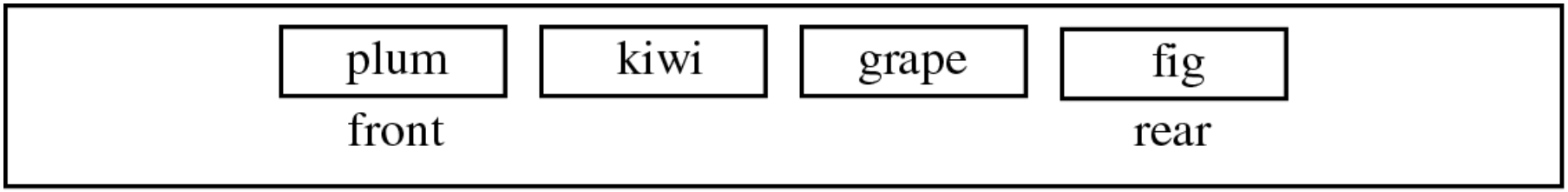


Node structure

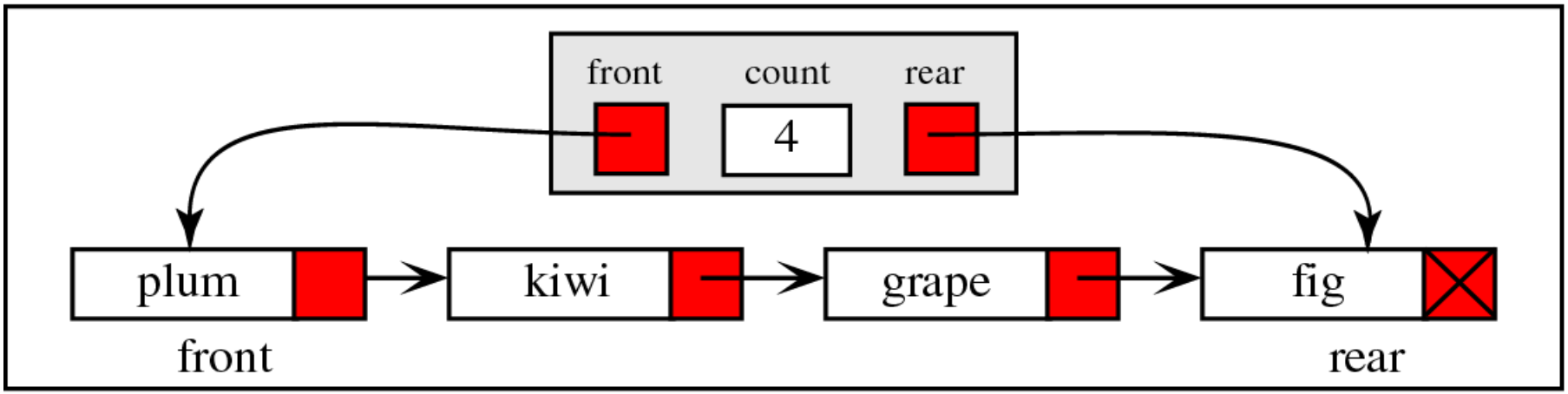
```
queueHead
  front     <node pointer>
  count     <integer>
  rear      <node pointer>
end queueHead


node
   data     <dataType>
   next     <node pointer>
end node
```

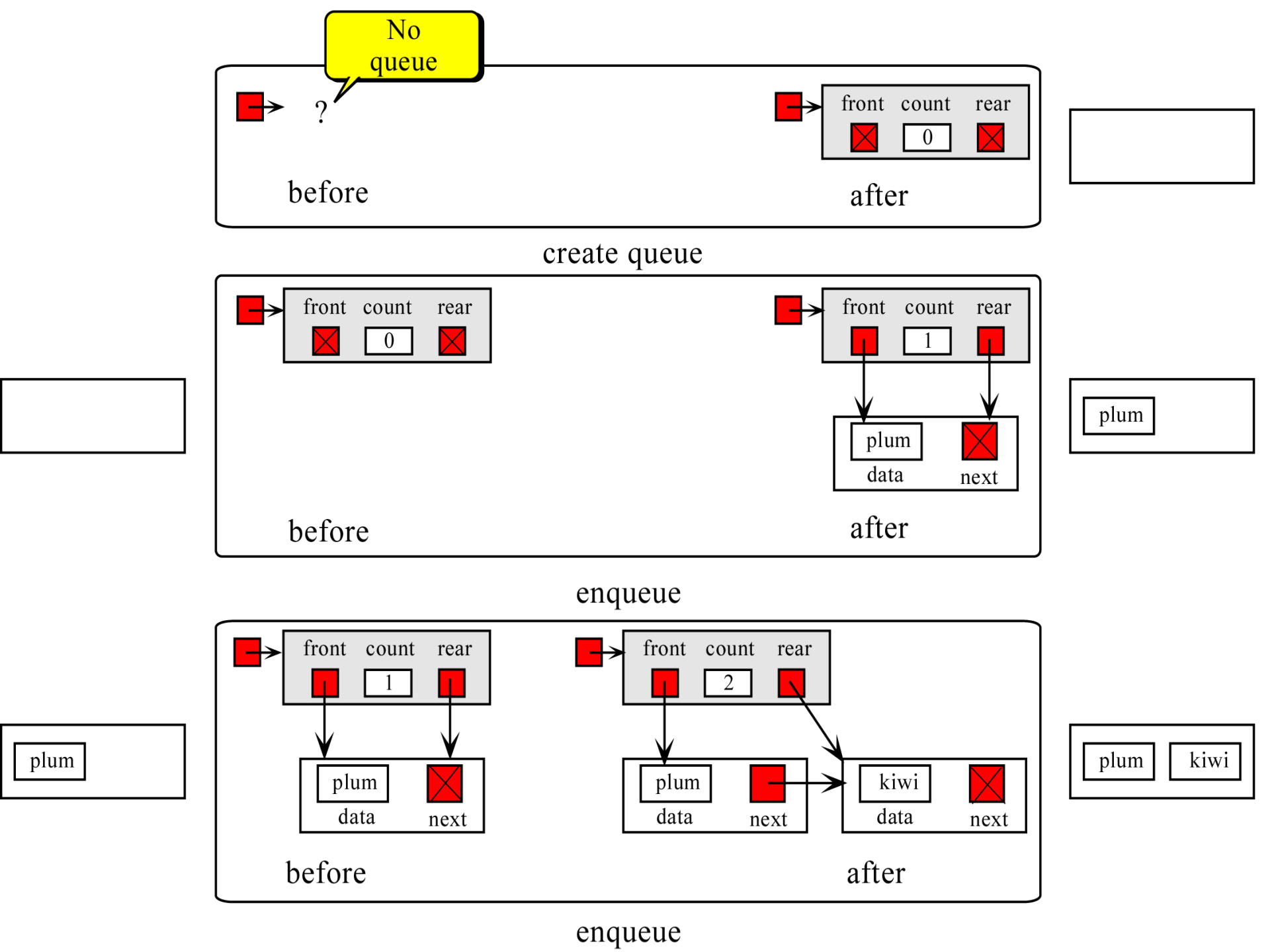
SLL implementation of queues



(a) Conceptual queue



(b) Physical queue

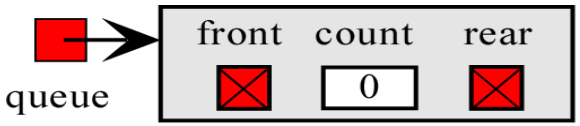

No queue
?
front count rear
0
before
after
create queue
front count rear
0
front count rear
1
plum
data
next
plum
before
after
enqueue
front count rear
1
front count rear
2
plum
plum
data
next
plum
data
next
kiwi
data
next
plum
plum
kiwi
before
after
enqueue

Queue Algorithms - Create Queue

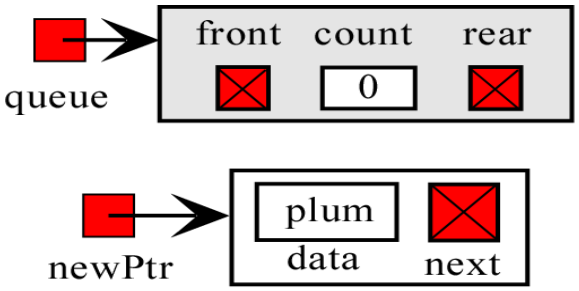
- Algorithm createQueue
- Allocates memory for a queue head node from dynamic memory and returns its address to the caller.
- Pre Nothing
- Post head has been allocated and initialized
- Return head's address if successful, null if memory owerflow.

1. if (memory available)
   - allocate (newPtr)
   - newPtr□front = null pointer
   - newPtr□rear = null pointer
   - newPtr□count = 0
   - return newPtr
2. else
   - return null pointer

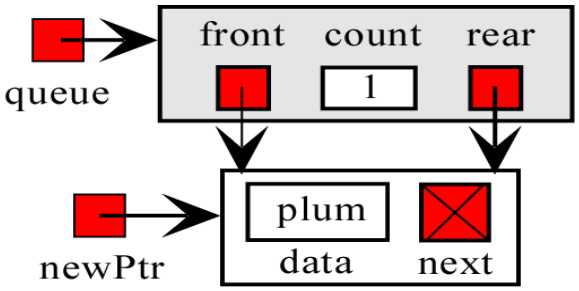
end createQueue



Queue Algorithms - Enqueue

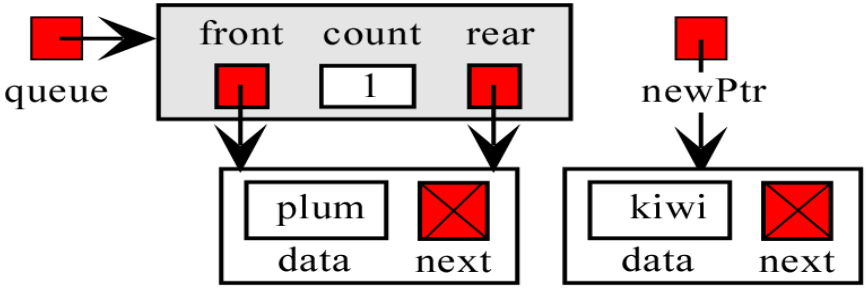


before

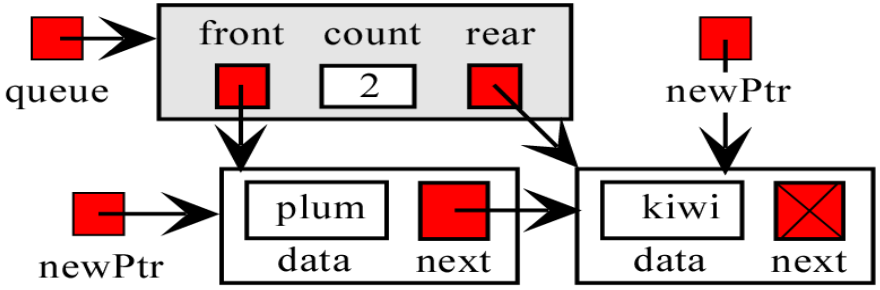


after

(a) Case 1: Insert into null queue



before



after

(b) Case 2: Insert into queue with data

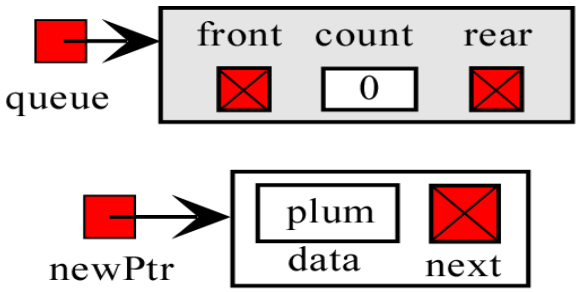
ENQUEUE()-Algorithm

- REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(REAR not equal to NULL)

(*a*) REAR → next = NewNode;

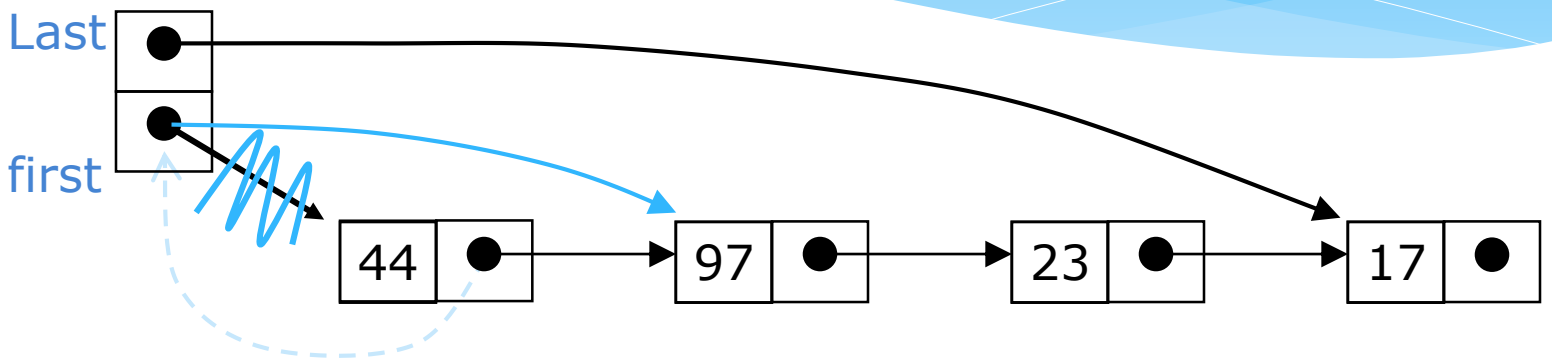
6. REAR =NewNode;
7. Exit



ENQUEUE()

```
        //This function will push an element into the queue
1.  NODE push(NODE rear)
2.  {
3.  NODE NewNode;                       //New node is created to push the data
4.  printf ("\nEnter the no to be pushed = ");
5.  scanf ("%d",&NewNode->info);
6.  NewNode->next=NULL;
7.  if (rear != NULL)                   //setting the rear pointer
        rear->next=NewNode;
8.  rear=NewNode;
9.  return(rear);
10. }
```

Dequeueing a node



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

Algorithm-pop

- REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT is equal to NULL)
   (*a*) Display “The Queue is empty”
2. Else
   (*a*) Display “The popped element is FRONT → DATA”
   (*b*) If(FRONT is not equal to REAR)
   (*i*) FRONT = FRONT → Next
   (*c*) Else
   (*d*) FRONT = NULL;
3. Exit

QUEUES

- There are three major variations in a simple queue. They are
  1. Circular queue
  2. Double ended queue (de-queue)
  3. Priority queue

Circular queue

- Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

- Rear is at last index
- New elements cannot be pushed

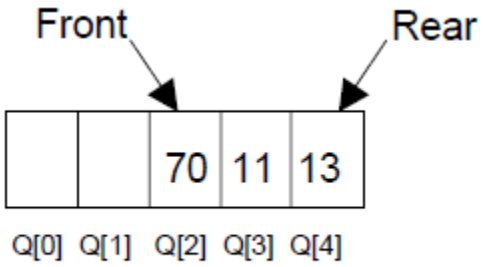


Fig. 4.10

Circular queue

- In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n].
- A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.
- Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular.

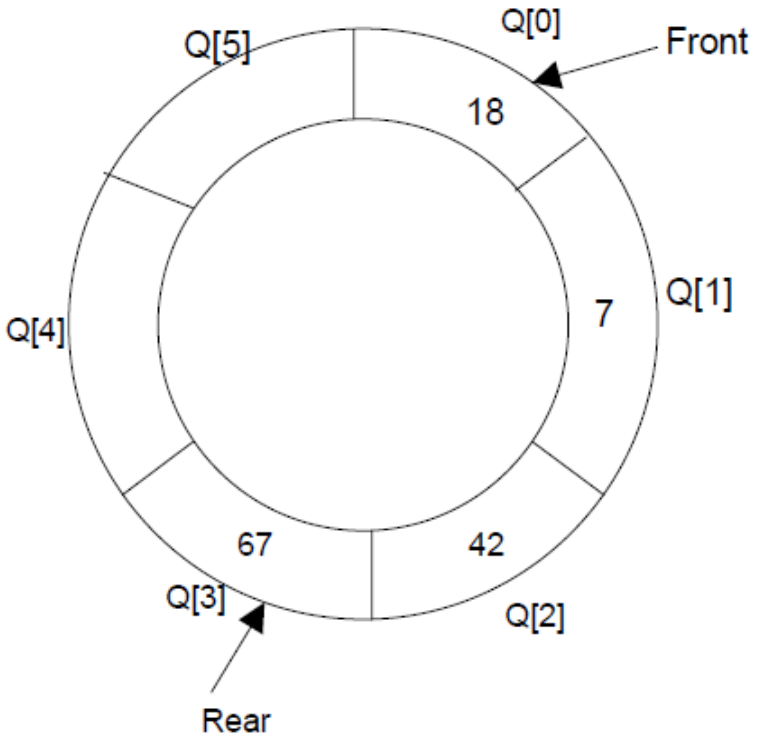


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

Circular queue

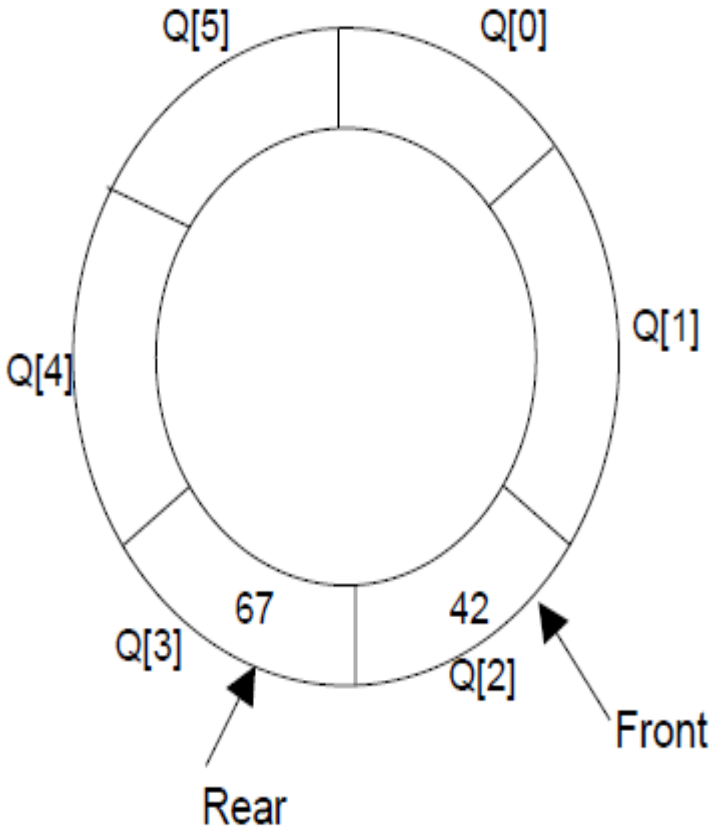


Fig. 4.12. A circular queue after popping 18, 7

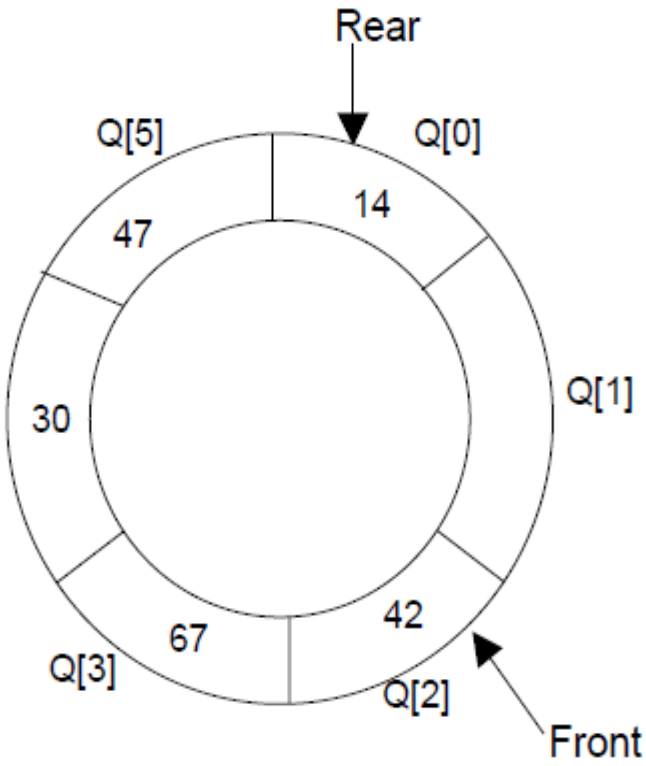


Fig. 4.13. A circular queue after pushing 30, 47, 14

Circular queue

- At any time the position of the element to be inserted will be calculated by the relation **Rear = (Rear + 1) % SIZE.**
- After deleting an element from circular queue the position of the front end is calculated by the relation **Front= (Front + 1) % SIZE.**
- After locating the position of the new element to be inserted, *rear*, compare it with *front*. If **(rear = front),** the queue is full and cannot be inserted anymore.