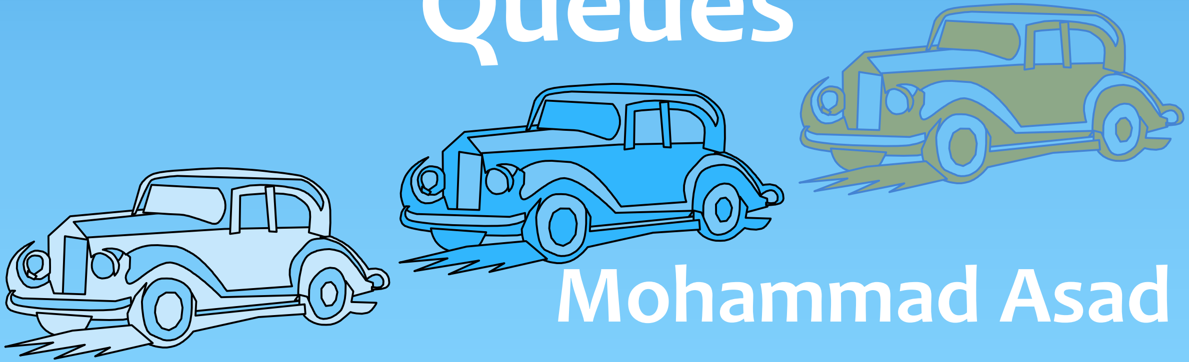
Queues



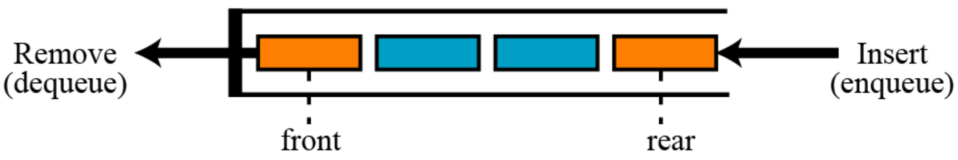
Mohammad Asad Abbasi

Lecture 5

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

Application: Job 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

Insertion and deletion from a sequential queue

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 software (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

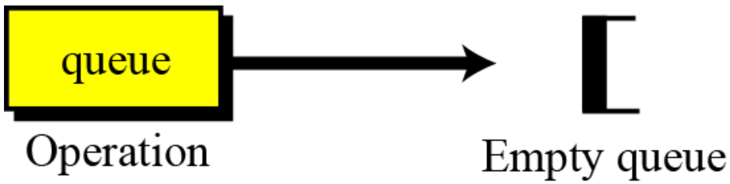
Main Queue Operations

- 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.**

Operations on queues

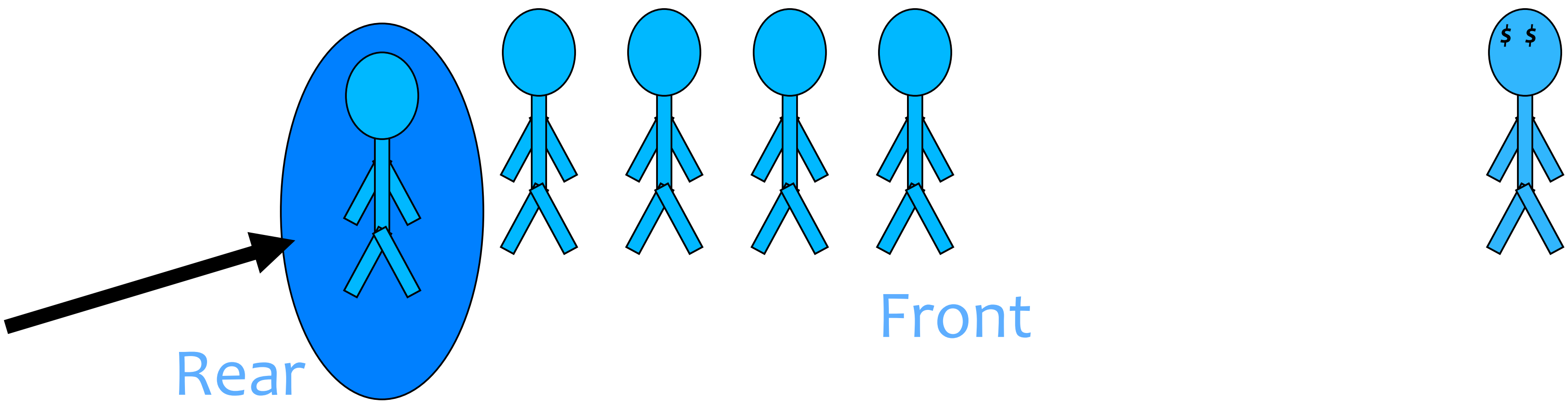
- The queue operation creates an empty queue.

queue (queueName)

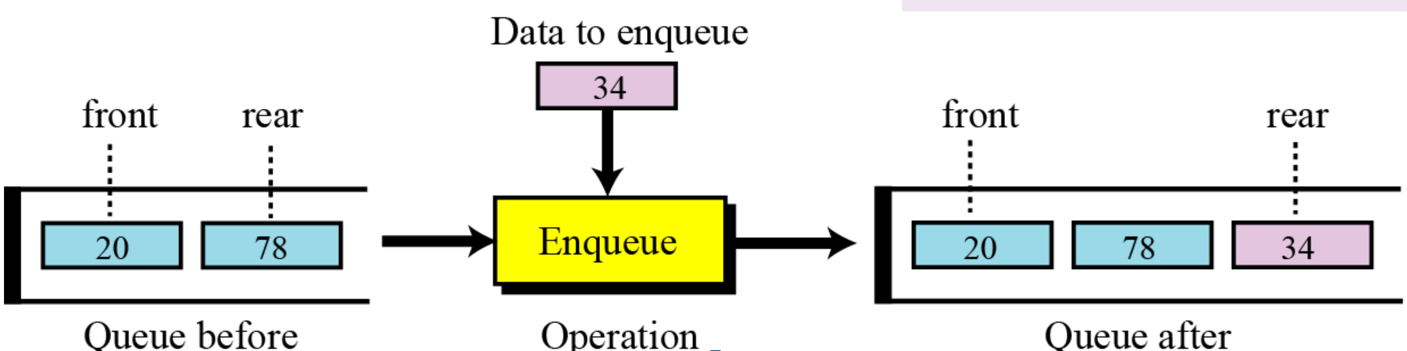


The *enqueue* operation

- The enqueue operation inserts an item at the rear of the queue.

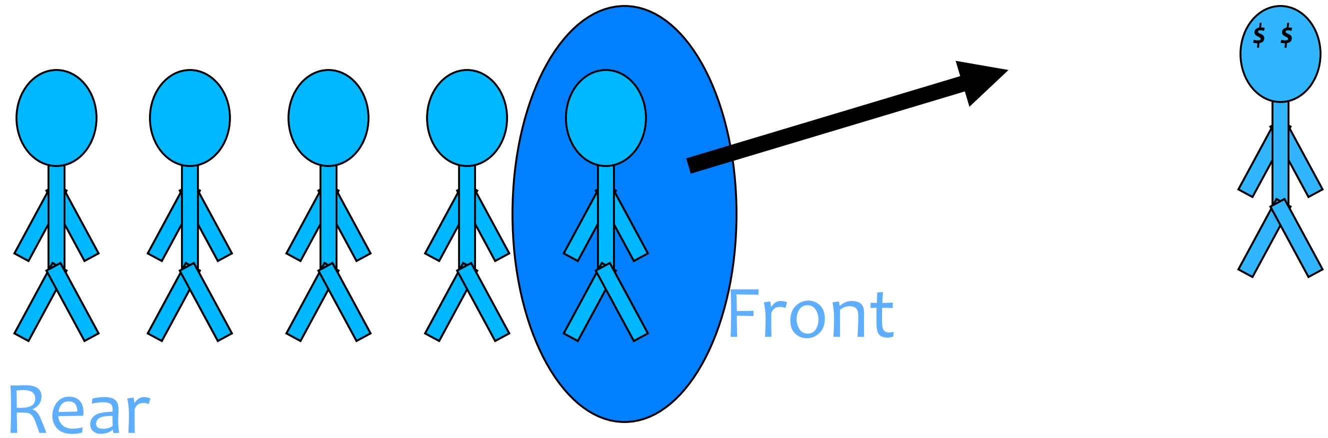


enqueue (queueName, dataItem)

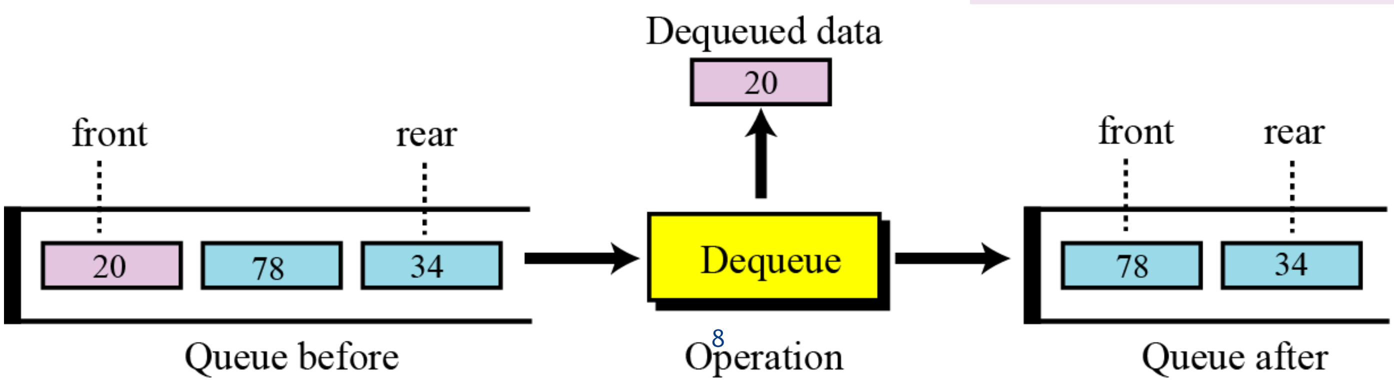


The *dequeue* operation

- The dequeue operation deletes the item at the front of the queue.

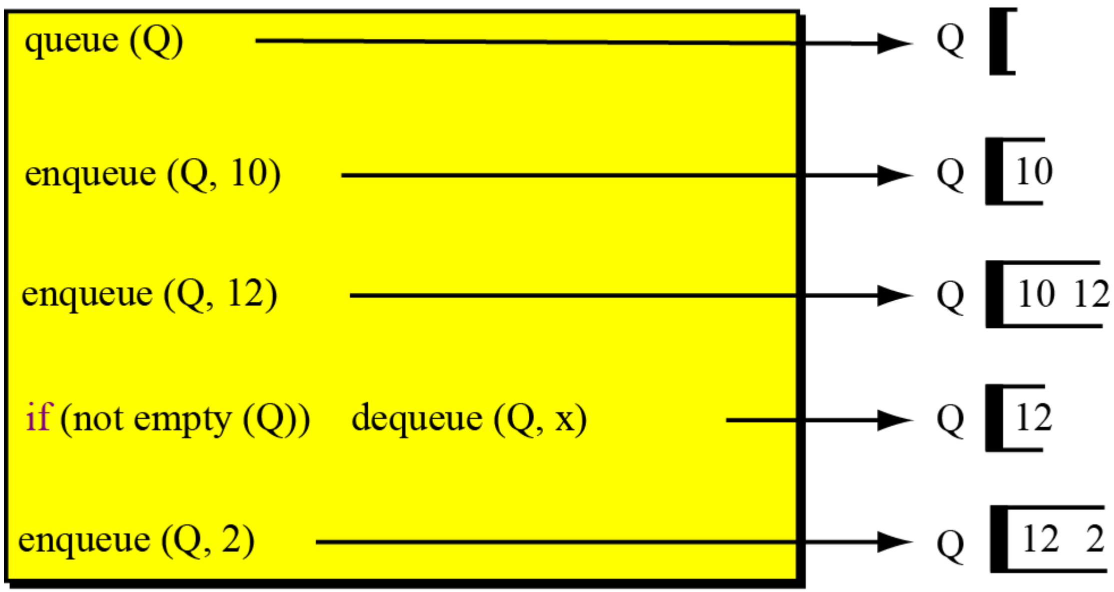


dequeue (queueName, dataItem)



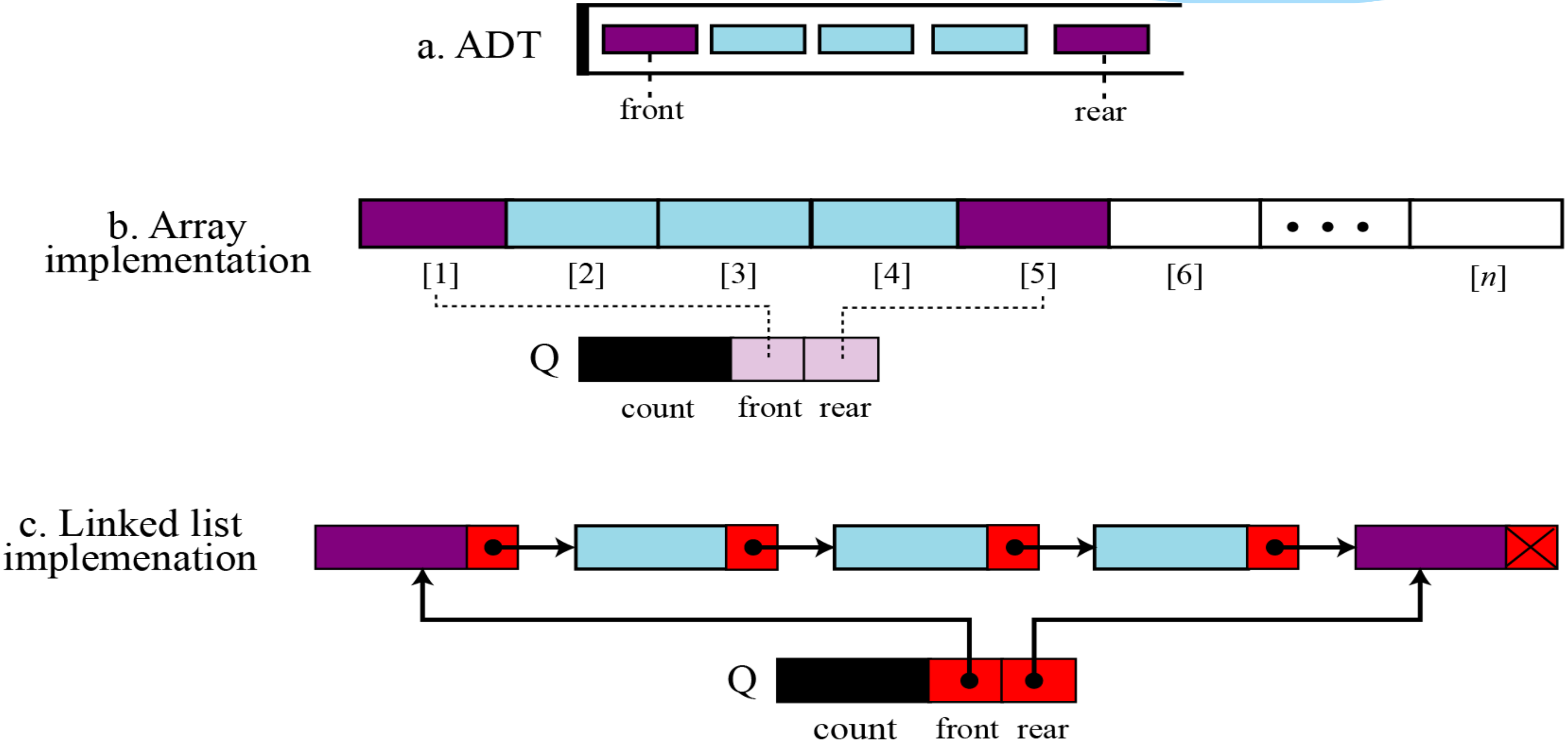
Example

- A segment of an algorithm that applies the previously defined operations on a queue Q.



An algorithm segment

Implementing a Queue



Example(Array Implementation)

Fig. 4.1. Queue is empty.

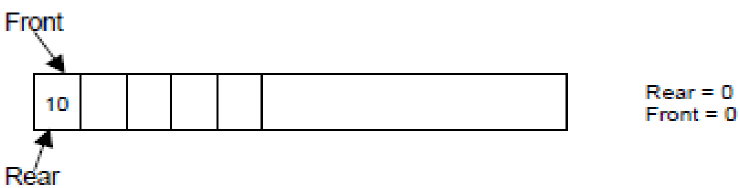


Fig. 4.2. push(10)

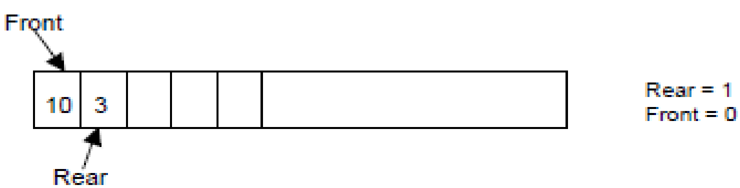


Fig. 4.3. push(3)

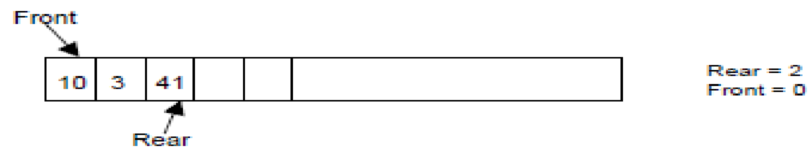


Fig. 4.4. push(41)

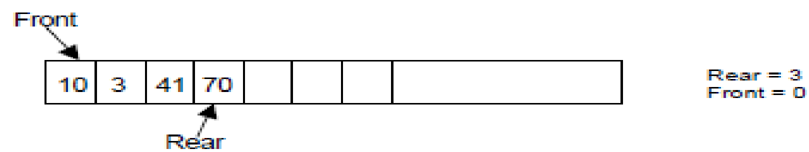


Fig. 4.5. push(70)

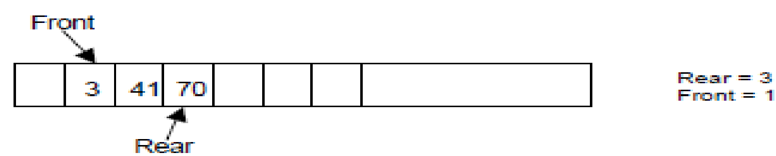


Fig. 4.6. x = pop() (*i.e.*; x = 10)

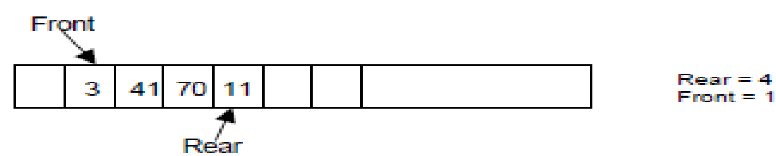


Fig. 4.7. push(11)

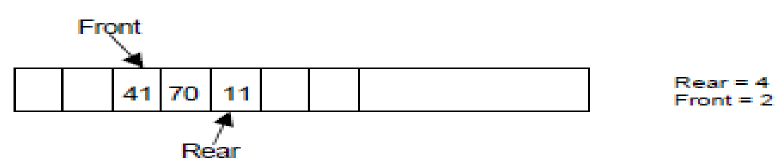
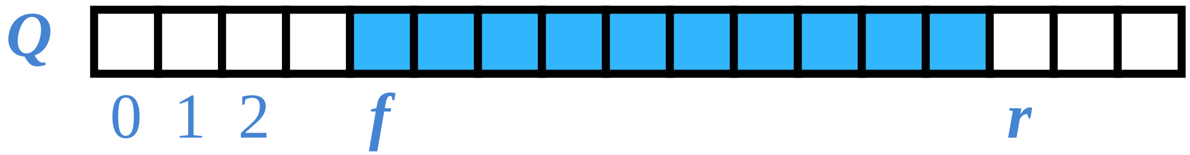


Fig. 4.8. x = pop() (*i.e.*; x = 3)

Implementation
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

normal configuration



INSERTING AN ELEMENT INTO THE QUEUE

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()

```
//This function will insert an element to the queue
void insert ()
{
int added_item;
if (rear==MAX-1)
{
printf("\nQueue Overflow\n");
getch();
return;
}
else
{
if (front==–1)                         /*If queue is initially empty */
front=0;
printf(“\nInput the element for adding in queue: ”);
scanf(“%d”, &added_item);
rear=rear+1;
//Inserting the element
queue_arr[rear] = added_item ;
}
}/*End of insert()*/
```

ALGORITHM TO DELETE AN ELEMENT FROM QUEUE

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()

```
//This function will delete (or pop) an element from the queue
void del()
{
if (front == –1 || front > rear)
{
printf ("\nQueue Underflow\n");
return;
}
else
{                                          //deleteing the element
printf ("\nElement deleted from queue is : %d\n",
queue_arr[front]);
front=front+1;
}
}/*End of del()*/
```

PROGRAM TO DISPLAY ALL QUEUE ELEMENTS

```
//Displaying all the elements of the queue
void display()
{
int i;
if (front == –1 || front > rear)                //Checking whether the queue is empty or not
{
printf (“\nQueue is empty\n”);
return;
}
else
{
printf(“\nQueue is :\n”);
for(i=front;i<= rear;i++)
printf(“%d ”,queue_arr[i]);
printf(“\n”);
}
}/*End of display() */
```

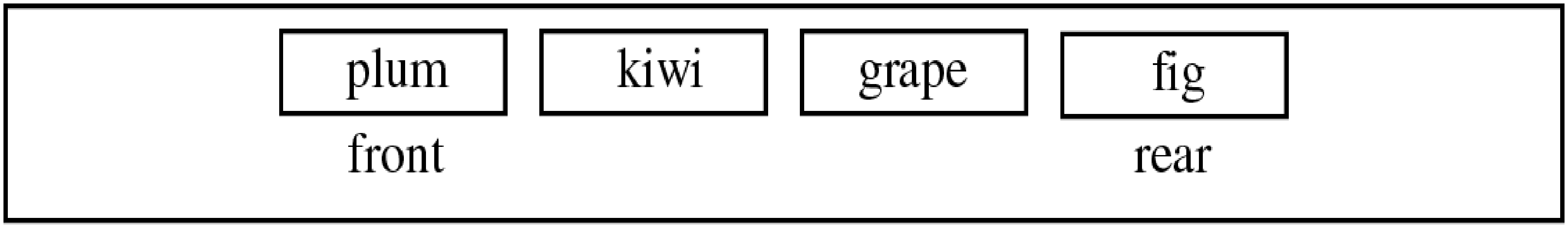
Linked-list implementation of queues

- 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 they are 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

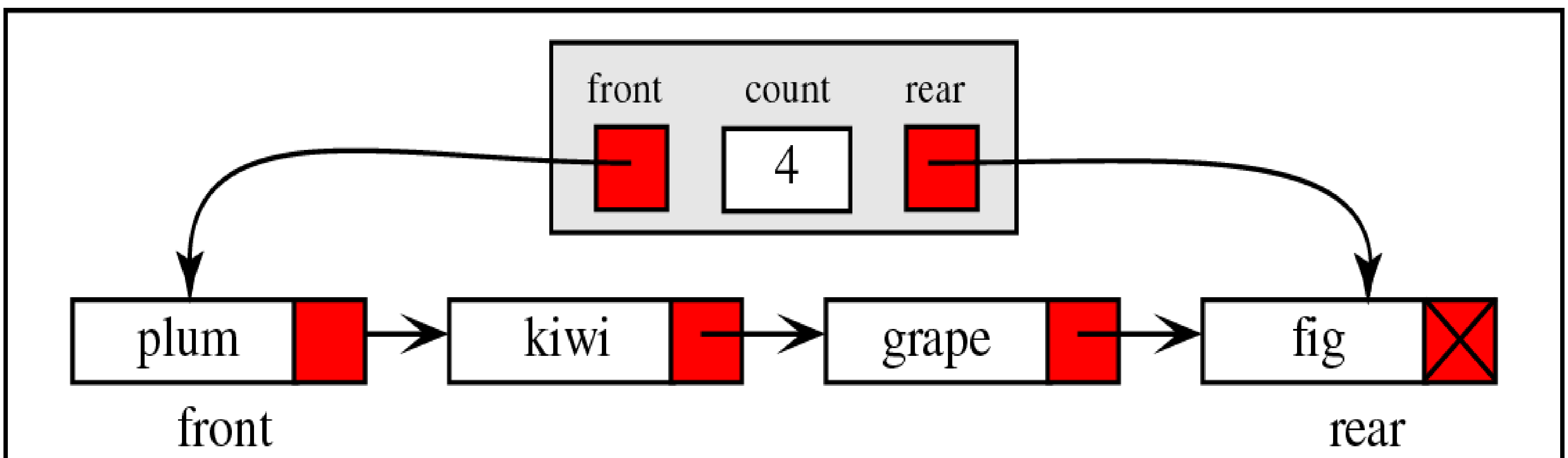
SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- 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 Linked List Design

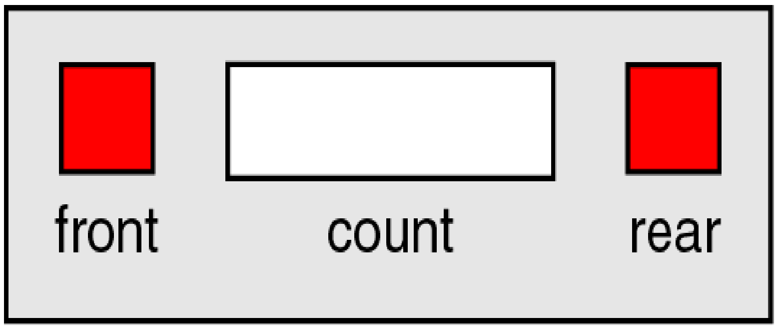


(a) Conceptual queue

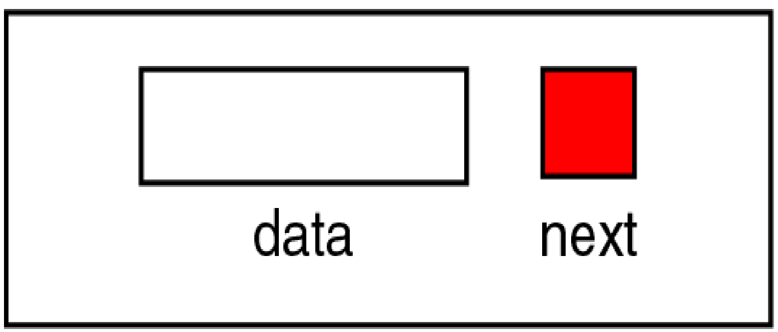


(b) Physical queue

Queue Data Structure



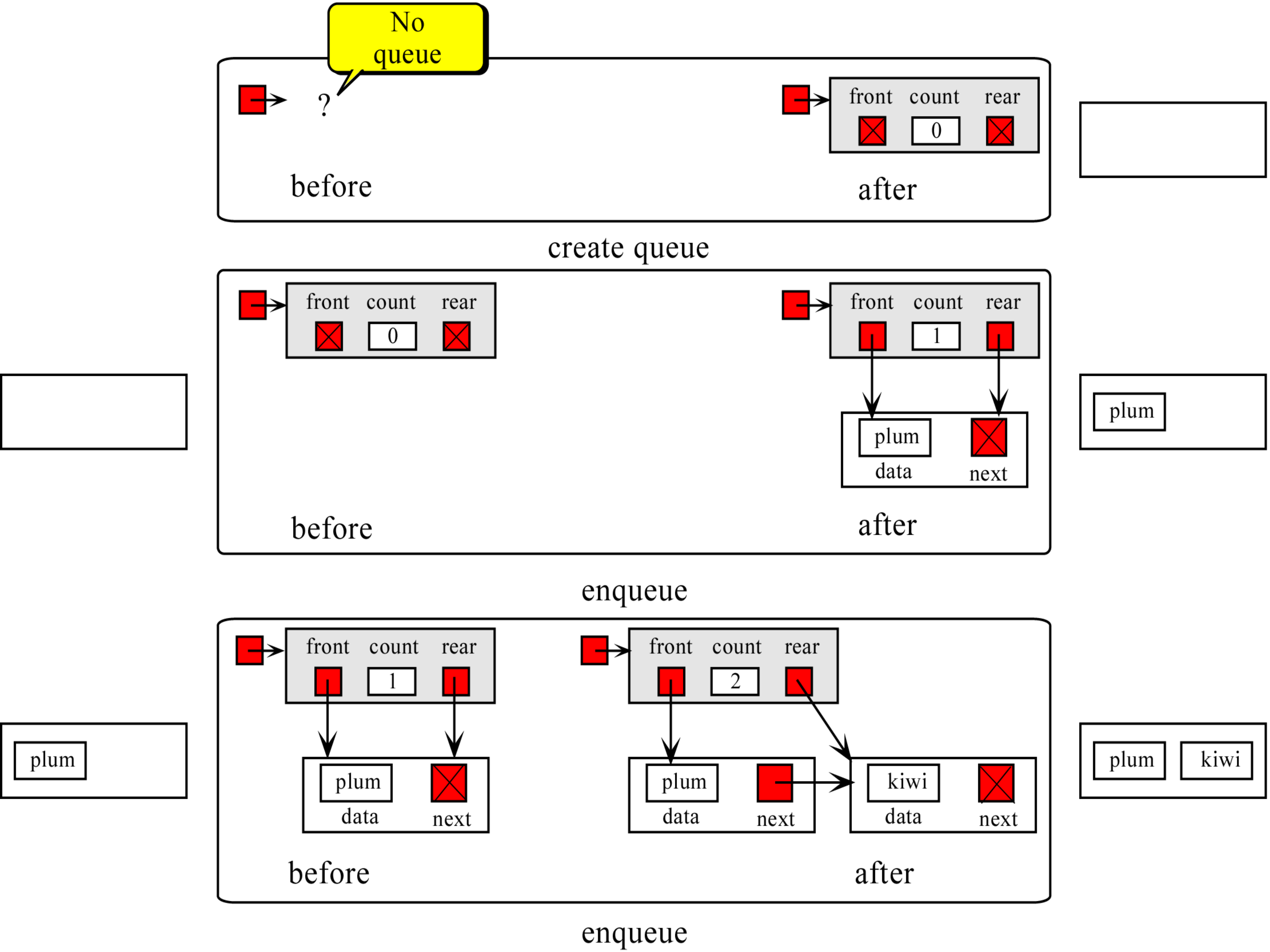
Head structure



Node structure

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


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



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

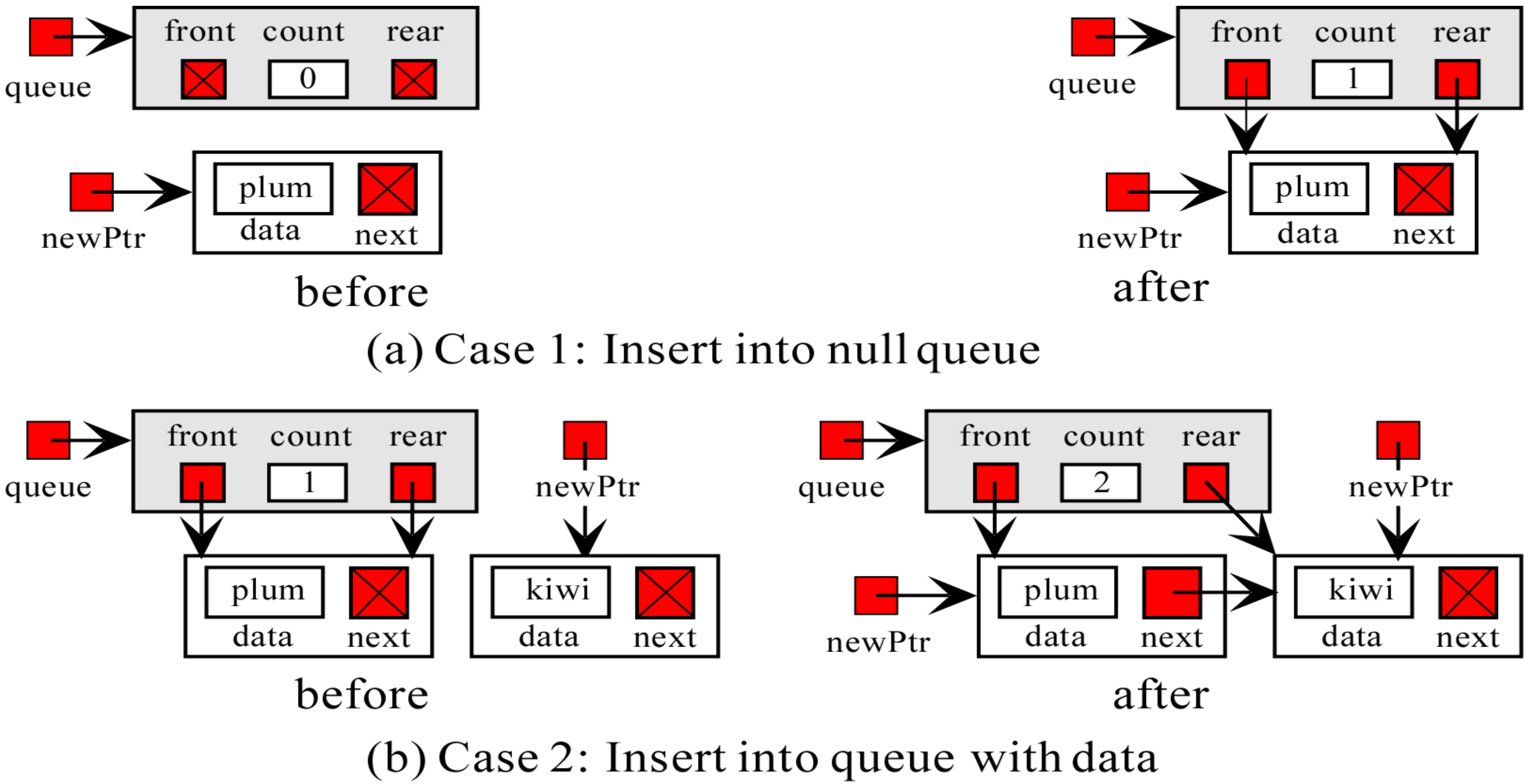
Queue Algorithms - Create Queue

- 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.

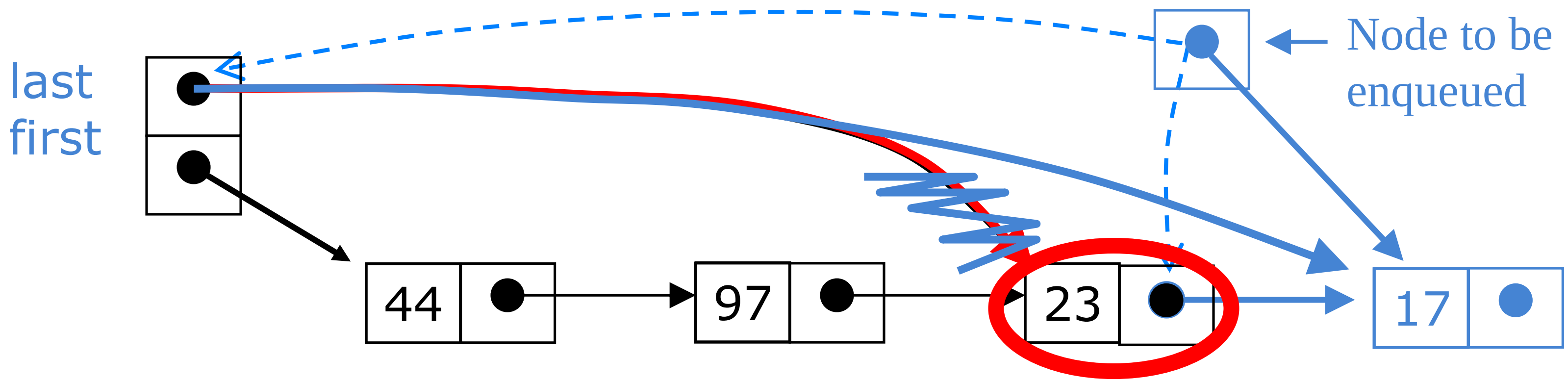
1. if (memory available)
   - allocate (newPtr)
   - newPtr→front = null pointer
   - newPtr→rear = null pointer
   - newPtr→count = 0
   - return newPtr
2. else
   - return null pointer

end createQueue

Queue Algorithms - Enqueue



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

ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

- 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 $\rightarrow$ DATA = DATA
4. NewNode $\rightarrow$ Next = NULL
5. If(REAR not equal to NULL)

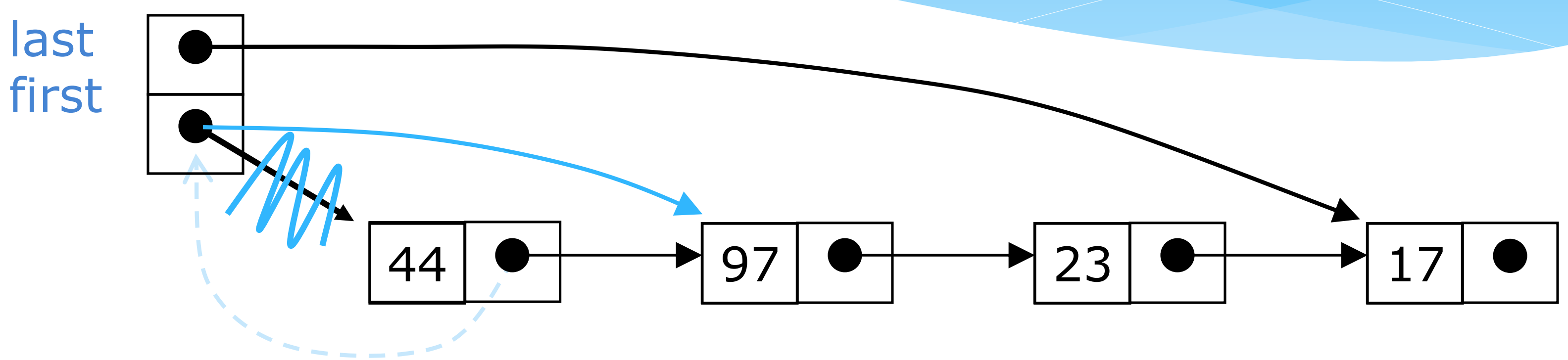
(*a*) REAR $\rightarrow$ next = NewNode;

6. REAR =NewNode;
7. Exit

PROGRAM FOR PUSHING AN ELEMENT TO A QUEUE

```
        //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 FOR POPPING AN ELEMENT FROM A QUEUE

- 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

OTHER 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.

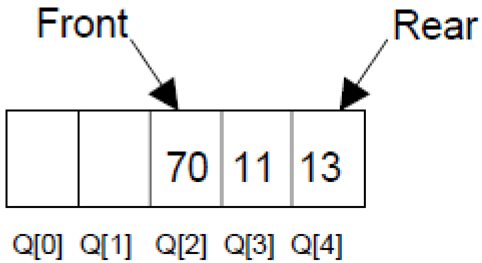


Fig. 4.10

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

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.

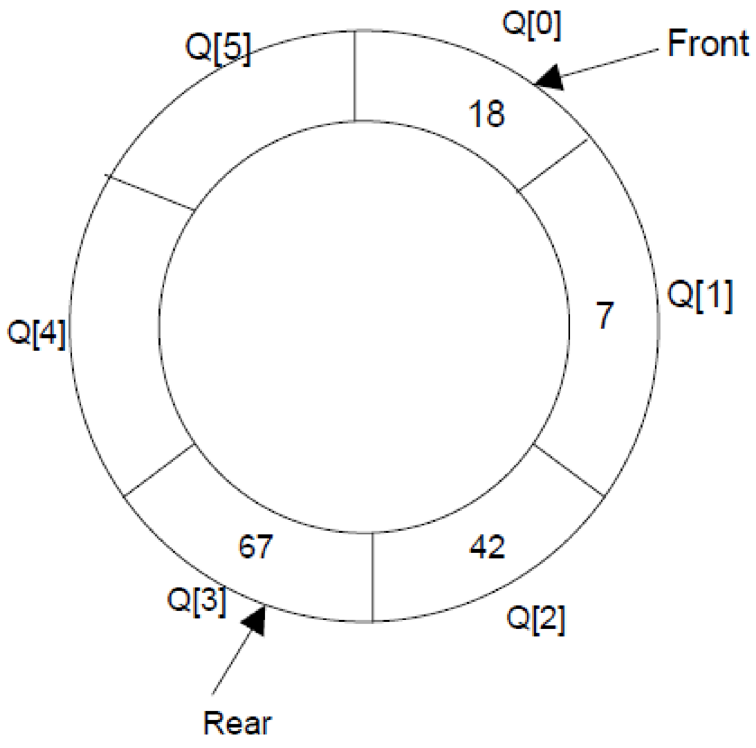


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

CIRCULAR QUEUE

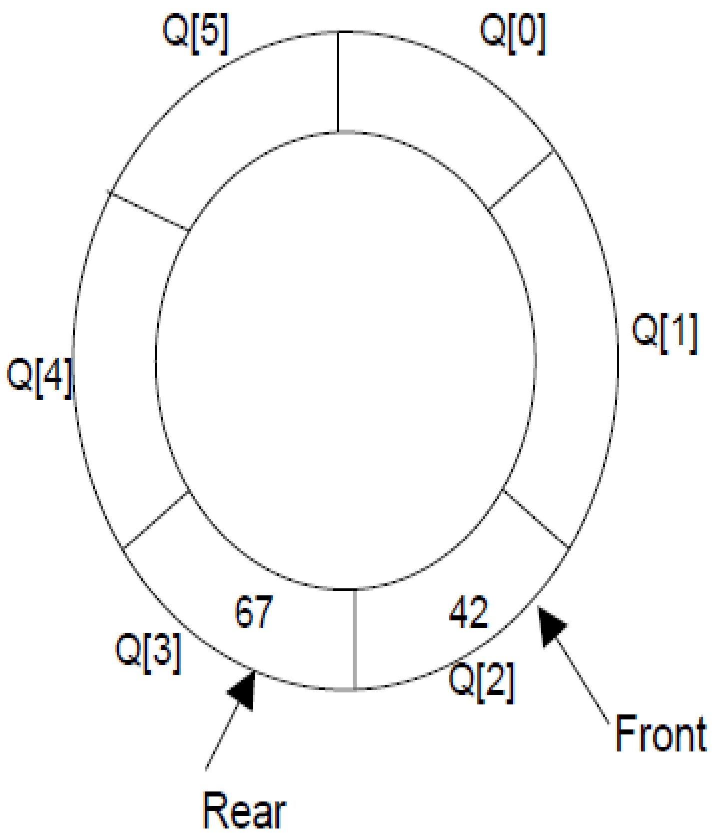


Fig. 4.12. A circular queue after popping 18, 7

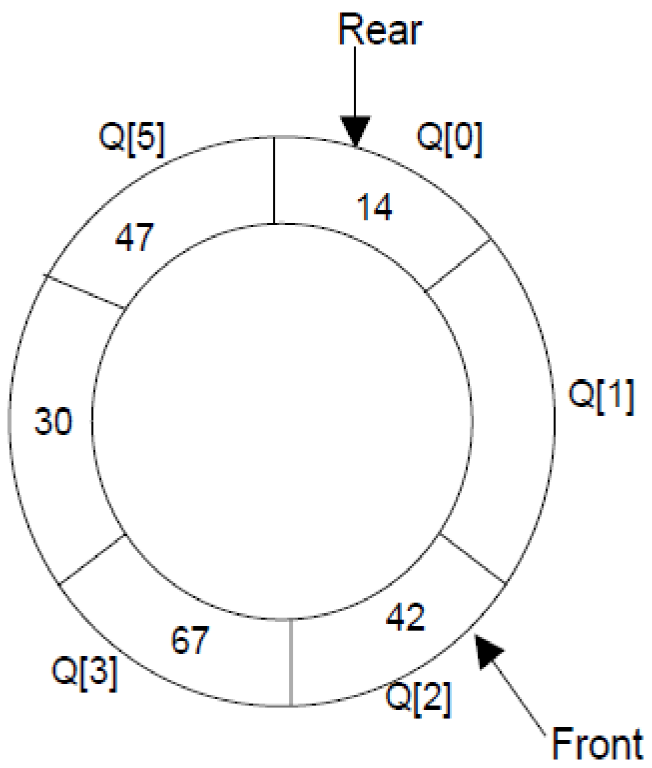


Fig. 4.13. A circular queue after pushing 30, 47, 14

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.