UNIT-II

STACKS INTRODUCTION:

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, It contains only one pointer **top pointer** pointing to the topmost element of the stack.

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Stack Operations:

The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation:

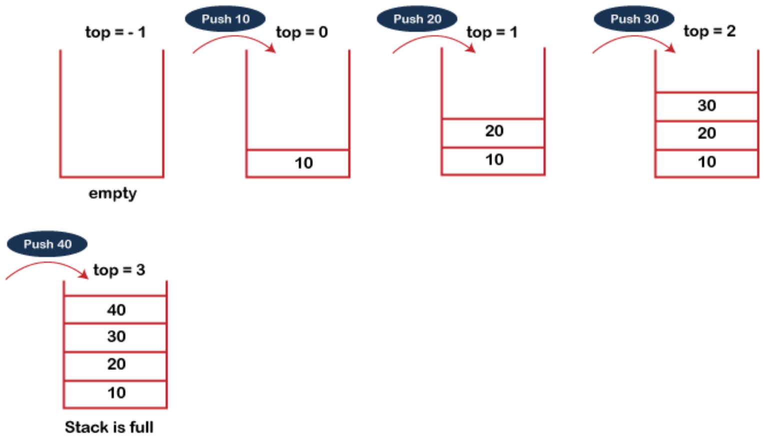
The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.

- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
end procedure
```



<u>POP operation:</u>

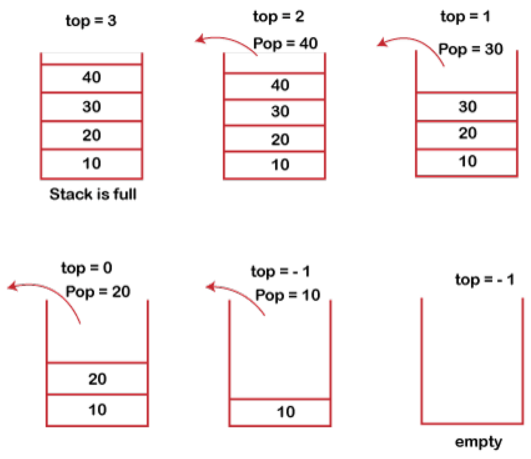
The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
 store value of stack[top]
```

```
        decrement top
        return value
      end else
      end procedure
```



<u>Implementation of Stacks using Arrays:</u>

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let's see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be decremented by 1 whenever an item is deleted from the stack.

The top most element of the stack is stored in another variable and then the top is decremented by 1. The operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an empty stack.

Algorithm :

```
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

```python
# Stack implementation in python


# Creating a stack
def create_stack():
   stack = []
   return stack


# Creating an empty stack
def check_empty(stack):
   return len(stack) == 0


# Adding items into the stack
def push(stack, item):
   stack.append(item)
   print("pushed item: " + item)


# Removing an element from the stack
def pop(stack):
   if (check_empty(stack)):
      return "stack is empty"

   return stack.pop()


stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Output :

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```

Pros: Easy to implement. Memory is saved as pointers are not involved.
Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

APPLICATIONS:

Expression

An expression can be defined as a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

1. **Infix Expression**

In infix expression, operator is used in between the operands. The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Ex: a+b

2. **Postfix Expression**

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**". The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Ex: ab+

3. **Prefix Expression**

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**". The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Ex:+ab

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Conversion from Infix to Postfix notation:

Problem with infix notation:

To evaluate the infix expression, we should know about the **operator precedence** rules, and if the operators have the same precedence, then we should follow the **associativity** rules.

The use of parenthesis is very important in infix notation to control the order in which the operation to be performed. An infix expression is the most common way of writing expression, but it is not easy to parse and evaluate the infix expression without ambiguity.

So, mathematicians and logicians discovered two other ways of writing expressions which are prefix and postfix. Both expressions do not require any parenthesis and can be parsed without ambiguity. It does not require operator precedence and associativity rules.

Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let's understand through an example.

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	KL+MN* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression **(K + L - M*N + (O^P) * W/U/V * T + Q)** is **KL+MN*-OP^W*U/V/T*+Q+**

#python code to convert from infix to postfix

```
OPERATORS = set(['+', '-', '*', '/', '(', ')', '^'])  # set of operators
PRIORITY = {'+':1, '-':1, '*':2, '/':2, '^':3} # dictionary having priorities
def infix_to_postfix(expression): #input expression
    stack = [] # initially stack empty
    output = '' # initially output empty
       for ch in expression:
        if ch not in OPERATORS:  # if an operand then put it directly in postfix expression
            output+= ch
        elif ch=='(':  # else operators should be put in stack
            stack.append('(')
        elif ch==')':
            while stack and stack[-1]!= '(':
                output+=stack.pop()
            stack.pop()
        else:
            # lesser priority can't be on top on higher or equal priority
             # so pop and put in output
            while stack and stack[-1]!='(' and PRIORITY[ch]<=PRIORITY[stack[-1]]:
                output+=stack.pop()
            stack.append(ch)
    while stack:
        output+=stack.pop()
    return output
expression = input('Enter infix expression')
print('infix expression: ',expression)
print('postfix expression: ',infix_to_postfix(expression))
```

Output:

```
Enter infix expression a+b-c*d^e/f
infix expression:   a+b-c*d^e/f
postfix expression:   ab+cde^*f/-
```

Evaluation of postfix expression:

Postfix expression:

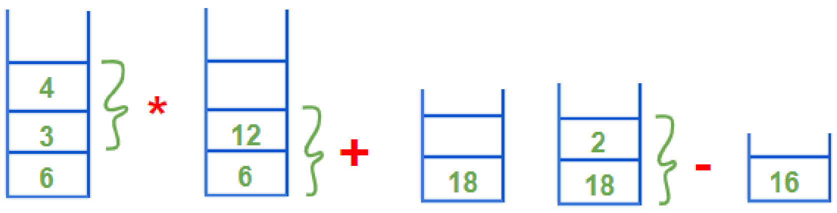
A postfix expression can be represented as: <operand><operand><operator>. Operator is succeeded by operands eg: XY+.

Rules:

1. Scan each value in the expression
2. If character == operand, put in stack
3. If character == operator, pop first 2 elements from stack
4. Use that operator on popped elements and put result in stack
5. Repeat above steps till you get final result

Example:

Consider a postfix expression = 634*+2-



Result = 16

```
OPERATORS=set(['*','-','+','%','/','**']) # set of operators allowed in expression
def evaluate_postfix(expression):
    stack=[] # empty stack for storing numbers
    for i in expression:
        if i not in OPERATORS:
            stack.append(i)  #contains numbers
        else:
            a=stack.pop() # if ch==operator then pop 2 numbers
            b=stack.pop()
            if i=='+':
                res=int(b)+int(a) # old val <operator> recent value
            elif i=='-':
                res=int(b)-int(a)
            elif i=='*':
                res=int(b)*int(a)
            elif i=='%':
                res=int(b)%int(a)
            elif i=='/':
                res=int(b)/int(a)
            elif i=='**':
                res=int(b)**int(a)
            stack.append(res) # final result
    return(''.join(map(str,stack)))
expression = input('Enter the postfix expression ')
print()
print('postfix expression entered: ',expression)
print('Evaluation result: ',evaluate_postfix(expression))
```

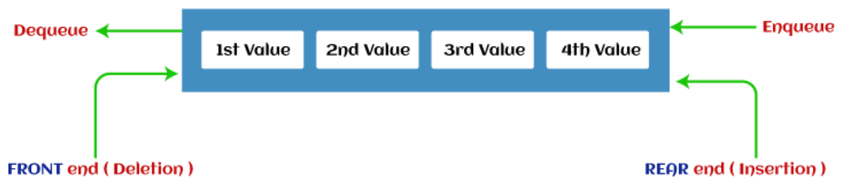
QUEUES:

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy.

Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

The representation of the queue is shown in the below image -



Now, let's move towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows –

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

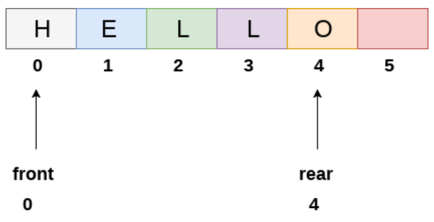
Now, let's see the ways to implement the queue.

Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear that are implemented in the case of every queue.

Front and rear variables point to the position from where insertions and deletions are performed in a queue.

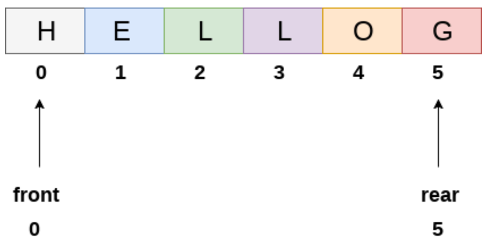
Initially, the value of front and rear is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue

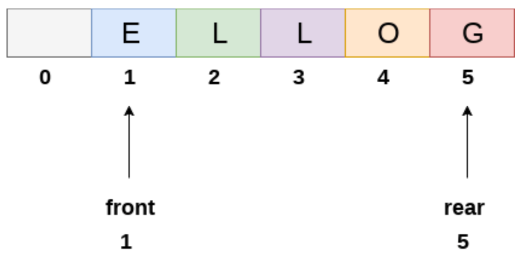
The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 .

However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
o Step1: IF REAR = MAX - 1
    Write OVERFLOW
    Go to step
    [END OF IF]
o Step 2: IF FRONT = -1 and REAR = -1
    SET FRONT = REAR = 0
    ELSE
    SET REAR = REAR + 1
    [END OF IF]
o Step 3: Set QUEUE[REAR] = NUM
o Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear, write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

```
o Step 1: IF FRONT = -1 or FRONT > REAR
    Write UNDERFLOW
    ELSE
```

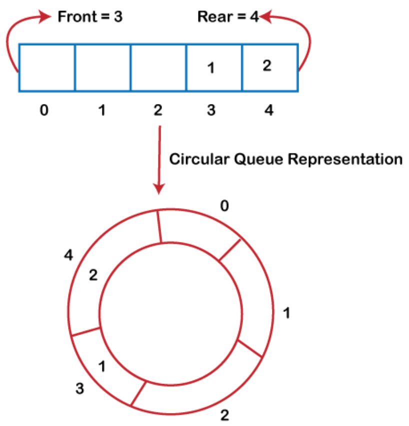
```
        SET VAL = QUEUE[FRONT]
        SET FRONT = FRONT + 1
        [END OF IF]
```

- **Step 2:** EXIT

CIRCULAR QUEUE:

There was one limitation in the array implementation of Queue

If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position.

In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue.

There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly.

It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.

- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

- front== rear + 1;

Algorithm to insert an element in a circular queue

```
Step 1: IF (REAR+1)%MAX = FRONT
    Write " OVERFLOW "
    Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
    SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
    SET REAR = 0
ELSE
    SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT
```

Dequeue Operation

The steps of dequeue operation are given below:

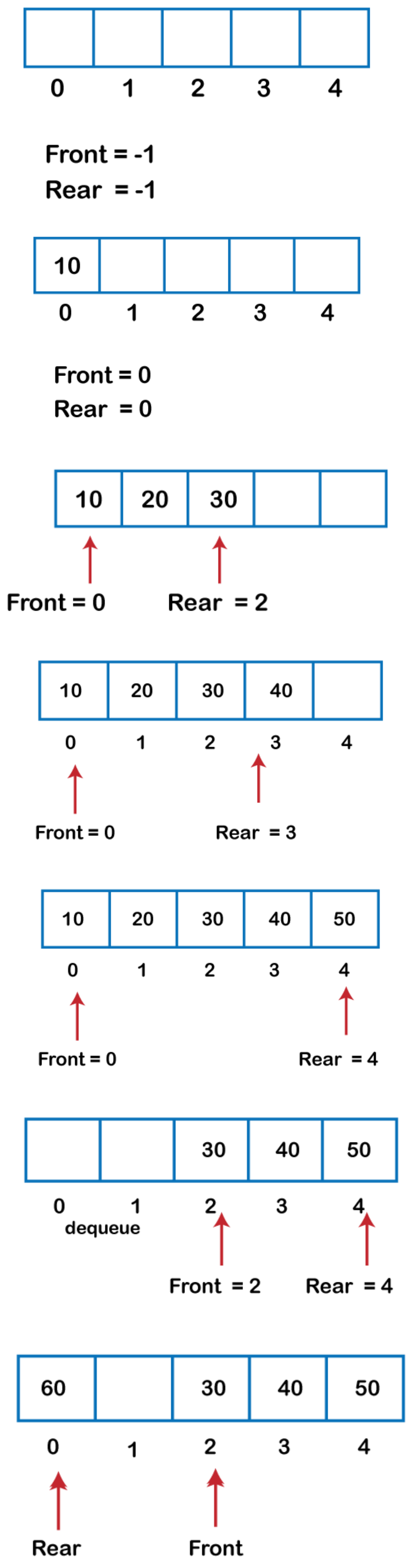
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

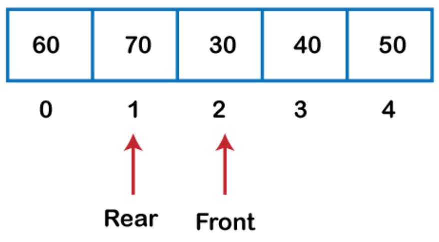
Algorithm to delete an element from the circular queue

```
Step 1: IF FRONT = -1
    Write " UNDERFLOW "
    Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
    SET FRONT = REAR = -1
ELSE
    IF FRONT = MAX -1
        SET FRONT = 0
    ELSE
        SET FRONT = FRONT + 1
    [END of IF]
[END OF IF]
```

Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.

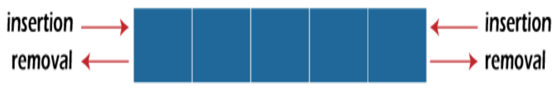




DEQUE (OR DOUBLE-ENDED QUEUE)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -



Representation of deque

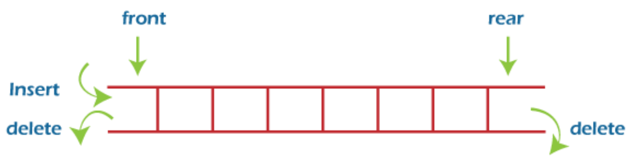
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

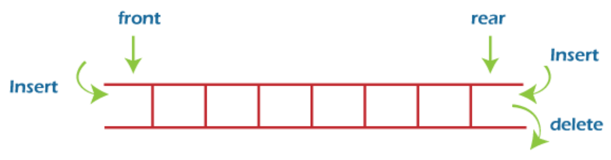
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque –

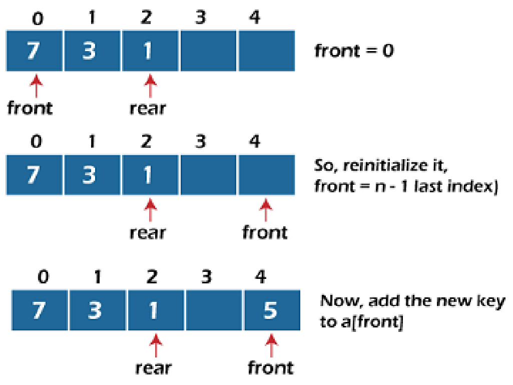
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

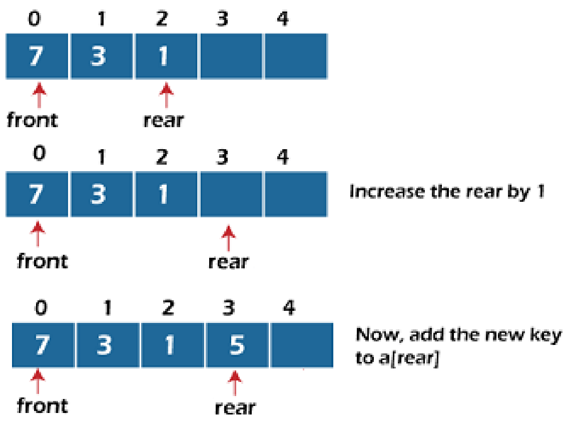
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions –

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

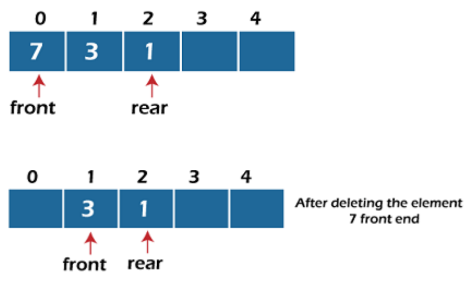
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

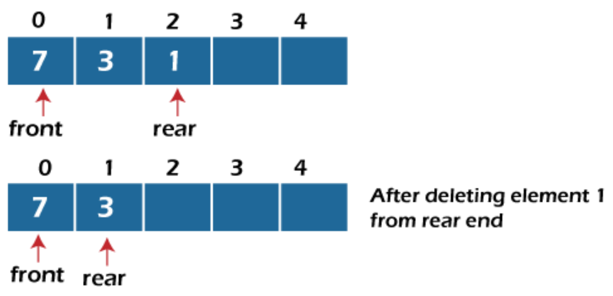
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

PRIORITY QUEUE:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue.

The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

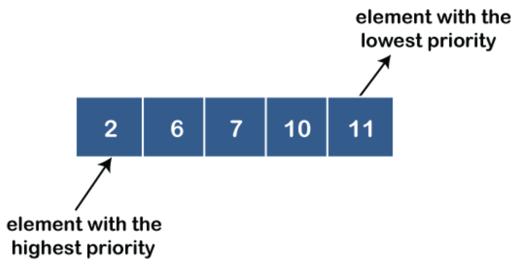
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

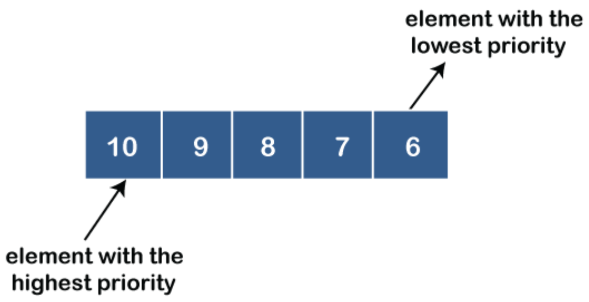
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e.,1 is given as the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

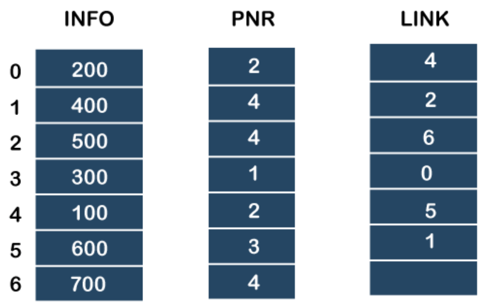
- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the the **INFO** list, and LINK basically contains the address of the next node.



	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

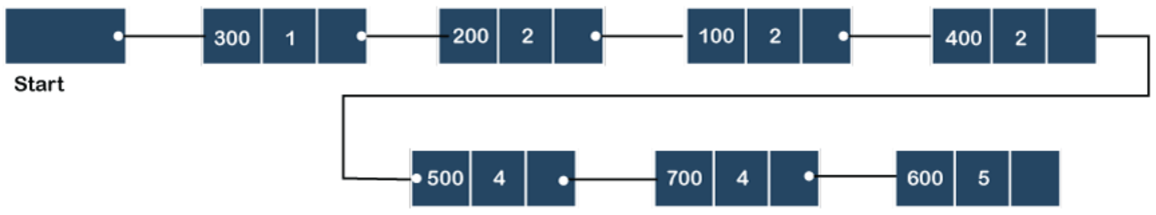
In the case of priority queue, lower value number is considered the higher priority, i.e., lower number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 300, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.



Implementation of Priority Queue

How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

- Arrays
- Linked list
- Heap data structure
- Binary search tree

Let's discuss all these in detail.

1) Using Array: A simple implementation is to use an array of the following structure.

```
struct item {
  int item;
  int priority;
}
```

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.

- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

2) Using Linked List:

In a Linked List implementation, the entries are sorted in descending order based on their priority. The highest priority element is always added to the front of the priority queue, which is formed using linked lists.

The functions like **push()**, **pop()**, and **peek()** are used to implement priority queue using a linked list and are explained as follows:

- **push():** This function is used to insert new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

3) Using Heaps:

Binary Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or LinkedList. Operation on Binary Heap are as follows:

- **insert(p):** Inserts a new element with priority p.
- **extractMax():** Extracts an element with maximum priority.
- **remove(i):** Removes an element pointed by an iterator i.
- **getMax():** Returns an element with maximum priority.
- **changePriority(i, p):** Changes the priority of an element pointed by i to p.

4) Using Binary Search Tree:

A Self-Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc. can also be used to implement a priority queue. Operations like peek(), insert() and delete() can be performed using BST.

Binary Search Tree	peek()	insert()	delete()
Time Complexity	O(1)	O(log n)	O(log n)

What is the difference between Priority Queue and Normal Queue?

- There is no priority attached to elements in a queue, the rule of first-in-first-out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupts handling.

Application of a linear queue

Figure shows a basic diagram of a time-sharing system. A CPU (processor) endowed with memory resources, is to be shared by n number of computer users.

The sharing of the processor and memory resources is done by allotting a definite time slice of the processors attention on the users and in a round-robin fashion.

In a system such as this, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU.

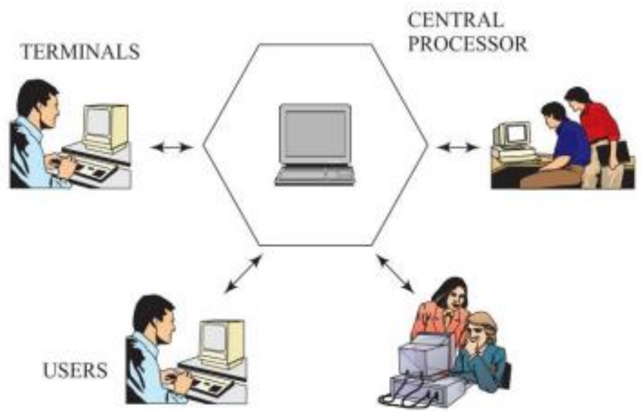


Fig. A basic diagram of a time-sharing system

However, to keep track of the jobs initiated by the users, the processor relies on a queue data structure recording the active user ids.

Example below demonstrates the application of a queue data structure for this job-scheduling problem.

The following is a table of three users A,B,C with their job requests J_i (k) where i is the job number and k is the time required to execute the job.

User	Job requests and the execution time in μ secs
A	J_1 (4), J_2 (3)
B	J_3 (2), J_4 (1), J_5 (1)
C	J_6 (6)

Thus J1 (4), a job request initiated by A needs 4 μ secs for its execution before the user initiates the next request of J2(3).

Throughout the simulation, we assume a uniform user delay period of 5 μ secs between any two sequential job requests initiated by a user. Thus B initiates J4 (1), 5 μ secs after the completion of J3 (2) and so on.

To initiate the simulation, we assume that A logged in at time 0, B at time 1 and C at time 2. Figure below shows a graphical illustration of the simulation.

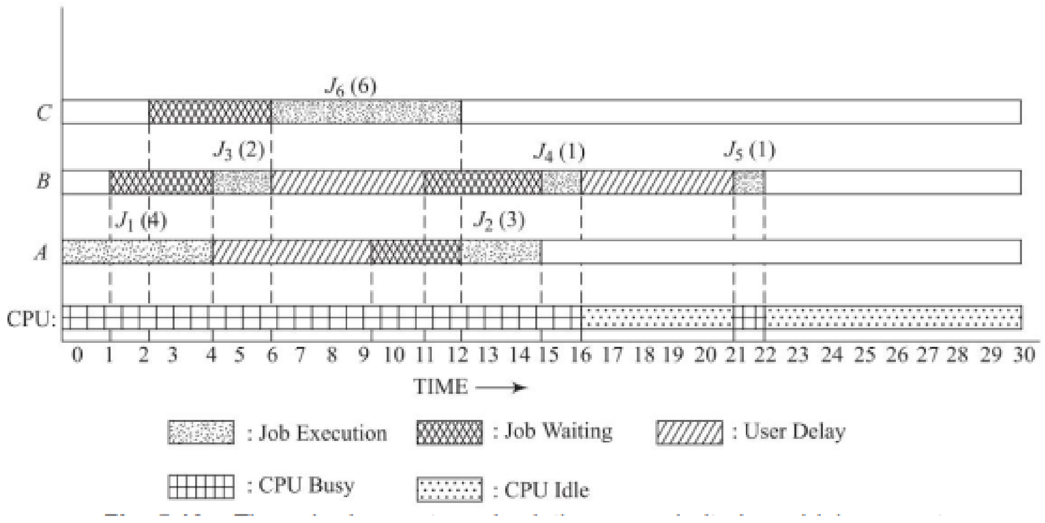
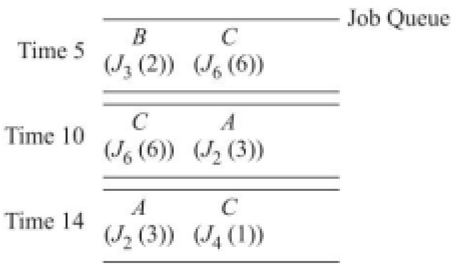


Fig: Time sharing system simulation non-priority based job requests



	Job Queue	
Time 5	B (J_3 (2))	C (J_6 (6))
Time 10	C (J_6 (6))	A (J_2 (3))
Time 14	A (J_2 (3))	C (J_4 (1))

Fig: Snapshot of the queue at times 5, 10 and 14

Application of priority queues:

Assume a time-sharing system in which job requests by users are of different categories. For example, some requests may be real time, the others online and the last may be batch processing requests.

It is known that real time job requests carry the highest priority, followed by online processing and batch processing in that order.

In such a situation the job scheduler needs to maintain a priority queue to execute the job requests based on their priorities.

If the priority queue were to be implemented using a cluster of queues of varying priorities, the scheduler has to maintain one queue for real time jobs (R), one for online processing jobs (O) and the third for batch processing jobs (B). The CPU proceeds to execute a job request in O only when R is empty.

Example: The following is a table of three users A,B,C with their job requests. Ri (k) indicates a real time job Ri whose execution time is k µ secs. Similarly Bi (k) and Oi (k) indicate batch processing and online processing jobs respectively.

User	Job requests and their execution time in μ secs
A	R_1 (4) B_1 (1)
B	O_1(2) O_2(3) B_2 (3)
C	R_2 (1) B_3 (2) O_3(3)

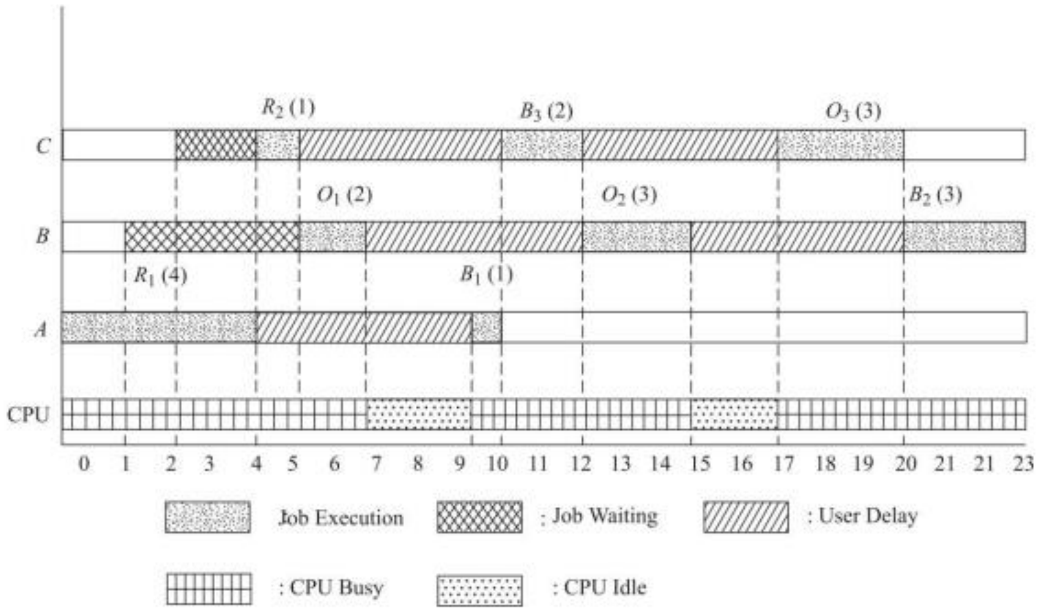


Fig. Simulation of the time sharing system for priority based jobs

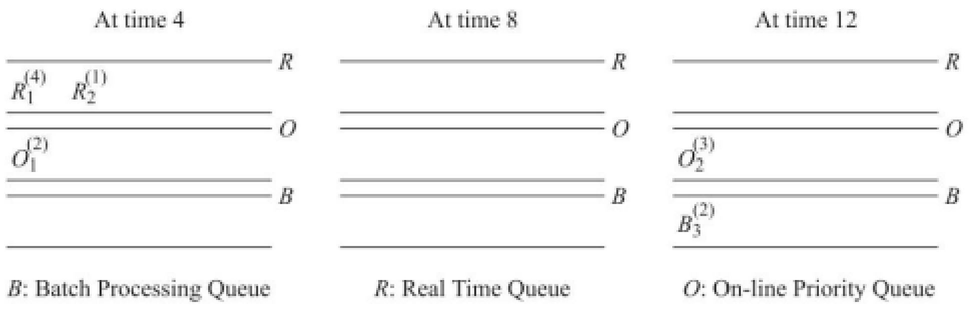


Fig. Snapshots of the priority queue at time 4, 8 and 12