3 Stack & Queues

3.1 BASICS OF STACK

- A stack is a linear data structure in which the items can be added or removed only at one end is called top of stack.
- Stacks are also called last in first out (LIFO).
- This means that elements are removed from the stack in reverse order of the way they were inserted.
- Two basic operations associated with stacks:
  - **a)** PUSH or Push: Insert an element into the stack.
  - **b)** POP or Pop: Delete an element from the stack.

Some of the stack representations can be given as:

Grey Matter Alert!

- LIFO (Last in first out) is also referred to as FILO (First In last out)
- Some other names for stacks are "Piles" or "Push-down lists"

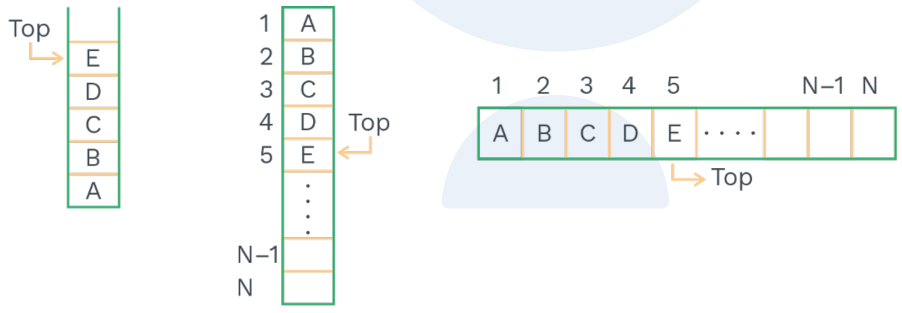


Fig. 3.1

Consider the above shown representation that 5 elements (A, B, C, D, E) are pushed onto an empty stack. The only implication is that rightmost element, i.e. E is the top element. This is emphasised regardless of the way the stack is described since it is the property of stack that insertion and deletion can only occur at the top of stack. This means that D cannot be deleted before E.

Note:

Stacks can either be used to postpone certain decisions or the order of the processing of data can be postponed until certain conditions get true.

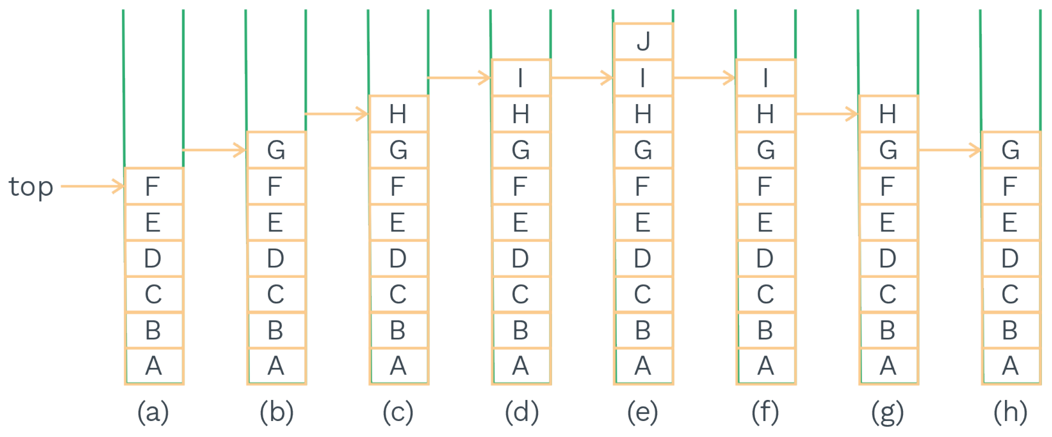


Fig. 3.2

In the above figure, F is the top of the stack. Any elements inserted after F are placed on top of F. The arrow with the word ‘top’ represents the top of stack.

a) represent the stack with F as the top.

b) Push (G), G is pushed on top of F.

c) Push (H), H is pushed on top of G.

d) and (e) similarly, I and J are pushed onto the stack, respectively.

f) POP (J), the element J is deleted from top of the stack.

g) and (h), POP (I), and POP (H) from the stack, respectively.

3.2 OPERATIONS ON STACK

The basic operations used for manipulation of stack data structure is:

a) PUSH: To insert an item into a stack

b) POP: To delete an item from a stack

Another operation used to know the top element of the stack is PEEK().
Therefore when an item is added to a stack, it is pushed onto the stack, and when an item is removed, it is popped out from the stack.
Consider a stack S and an item a then:

Operation	Meaning
PUSH (s, a)	Push the item a onto the top of the stack s
POP(s)	Removes the item from the top of the stack.
x=POP(s)	Assigns the value of the top of the stack to x and removes it from the stack s.
PEEK(s)	Return the top of the stack.

Table 3.1 Stack Operations

3.3 STACK CONDITIONS

- If a stack has no item, this stack is called an empty stack.
- However, PUSH operation can be performed on an empty stack but a POP operation cannot be applied.
- Hence before applying a POP operation, one needs to make sure that the stack is not empty.

Note:

Due to the push operation, the stack is sometimes referred to as push down lists.

- The two stack conditions to be checked before performing an operation on stack are:

a) Underflow condition:

When a POP operation is tried on an empty stack, it is called underflow condition. This condition should be false for successful POP operation and PEEK operation.

Note:

Empty determines whether stack s is empty or not. If the value returned by the operation is true then the stack is empty.

Operation	Return value	Meaning
empty(s)	TRUE	Stack s is empty
empty(s)	FALSE	Stack s is not empty

b) Overflow condition:

An overflow condition checks whether the stack is full or not, i.e. whether memory space is available to push new elements onto the stack. It is an undesirable condition where the program requests more memory space than what is allocated to a particular stack.

Underflow condition: (empty (s) == TRUE)

Or

(TOP == -1)

Overflow condition: TOP == sizeof(s)

Where s is the stack on which PUSH and POP operations are to be performed.
Here TOP is a special pointer which always points to the top of the stack and plays a major role in checking the overflow and underflow conditions of the stack.

Push operation on stack:
Let, n be the maximum size of stack and p be the element to be inserted onto the stack s.

```
Push(s, Top, n, p)
{        if(Top = = n – 1)
                 { printf("stack overflow");
                          exit(1);
                 }
                 Top ++;
                 S[Top] = p; //element p inserted on top of the stack
}
```

Pop operation of stack:
Let n be the maximum size of stack and p be the variable which will contain the value of top of the stack.

```
Pop(s, Top, n)
    {        int p;
                     If(Top = = –1)
                             { printf("under flow condition");
                                      exit(1);
                             }
                     p = s[Top];
                     Top - - ;
                     return(p);
    }
```

Application of stack:
Some of the most popular applications of stacks are:
a) Expressions evaluation
e) Fibonacci series
b) Balanced parenthesis check
f) Permutation
c) Recursion
g) Subroutines
d) Tower of hanoi

Balanced parentheses:
- For checking balanced parentheses for every open brace, there should be a closing brace.

Rack Your Brain

Which data structure does the UNDO function of the test editors use?
How does the compiler work for recursions?

- The types of parenthesis used are:
  - **a)** Simple or common bracket ( )
  - **b)** Square bracket [ ]
  - **c)** Curly bracket { }

Example: {{} [] ()} Balanced parenthesis

((((() ()))) ())

{[]()} {()} Imbalanced parenthesis

[[][]((()

1) Push the open parenthesis onto the stack

2) Whenever a close parenthesis is encountered then pop the top of the stack which should be the matching parenthesis.

Note:

The two conditions that must hold if the parenthesis in an expression forms an admissible pattern:

1) The parenthesis count at the end of the expression should be 0. This implies that there are as many left parentheses as right parentheses.

2) The parenthesis count should be non-negative at each point in the expression.

Also, nesting depth at a particular point in an expression is the number off scopes that have been

Considered, ((()())())

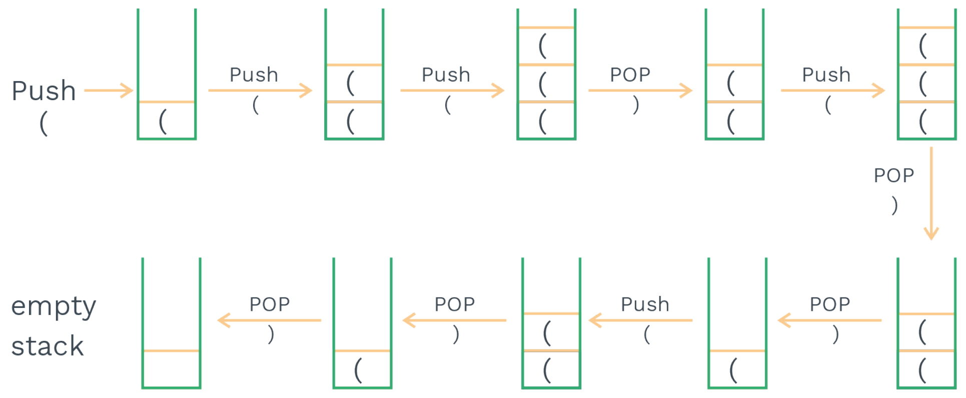


Fig. 3.3

The state of stack at various stages of processing the expression for balanced parenthesis

$\{a + (b - [c + d]) * e - [(f + g)]\}$

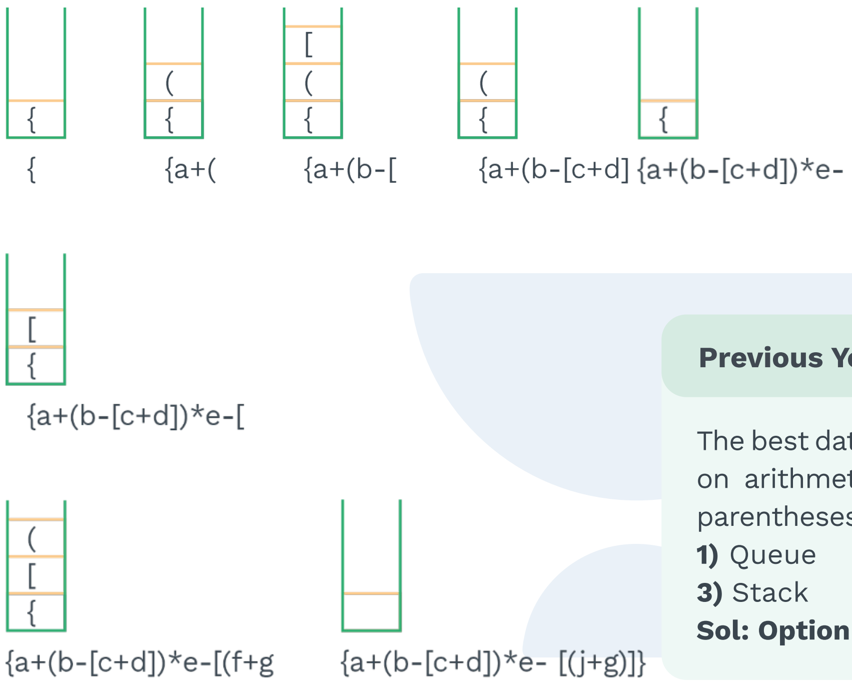


Fig. 3.4

Previous Year's Question

The best data structure to check whether on arithmetic expression has balanced parentheses is a,

1) Queue	**2)** Tree
3) Stack	**4)** List

Sol: Option 3) **(GATE- 2004)**

3.4 EXPRESSION EVALUATION

There are three notations for an expression representation:

a) Infix notation: In this type of notation the operator lies between the operands. Example: sum of A and B can be given as A+B

b) Prefix notation: In this type of Notation the operators come before the operands.
Example: Sum of A and B can be given as +AB

c) Postfix notation: In this type of notation the operator comes after the operands.
Example: Sum of A and B can be given as AB+

There are five binary operations:

a) Addition (+)	**b)** Subtraction (-)
c) Multiplication (*)	**d)** Division (/)
e) Exponentiation ($)	

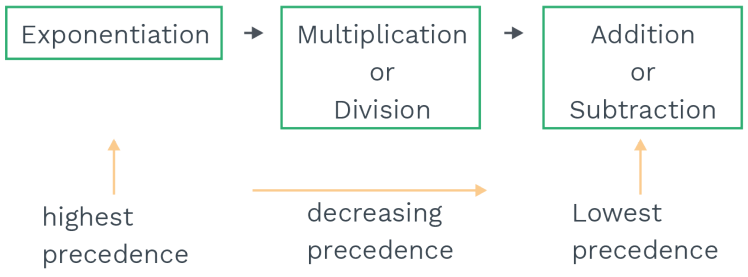


Fig. 3.5

Example:

Infix	Prefix	Postfix
A+B*C	+A*BC	ABC*+
A+B	+AB	AB+
A-B/(C*D$E)	-A/B*C$DE	ABCDE$*/–
(A+B) * (C+D)	*+AB + CD	AB + CD +*

Table 3.2 Infix vs Prefix vs Postfix

Note:

Prefix notation is also known as polish notation postfix notation is also known as reverse polish notation.

The expression sum of two variables A and B can be given as A + B where A and B are the operands, and '+' is the operator.

- While expression evaluation, the precedence of operators plays a vital role.
- Parenthesis are of great help in getting the order of evaluation.

Example:

A + (B * C)	(A + B) * C
• Here, parenthesis emphasises that multiplication is converted first and then addition.	• Here, parenthesis emphasises that addition is converted first and then multiplication.

Table 3.3

When unparenthesized operators having the same precedence are scanned then the order by default is considered from left to right.

- But in the case of exponentiation, the default order of evaluation is considered from right to left.

Example:

A + B + C	A $ B $ C
(Left to Right)	(Right to Left)
(A + B) + C	A $ (B $ C)

Note:

Default precedence can be overridden by using parenthesis.

Expression conversion:

- The prefixes ‘pre-’, “post-”, and “in-” refer to the relative position of the operator w.r.t. the operand.
- The only rules to always remember while expression conversion to any form is that operations with higher precedence are converted first and after a portion of the expression is converted to a form, then it is considered as a single operand.

Example: A + B * C (Infix)

$A + \underbrace{(BC*)}_{\text{Treat as single operand}}$ (Postfix) $A + \underbrace{(*BC)}_{\text{Treat as single operand}}$ (Prefix)

Note:

Rules of expression conversion:

1) The operations of higher precedence are converted first.

2) Once an operation in converted to postfix or prefix form it is considered as a single operand.

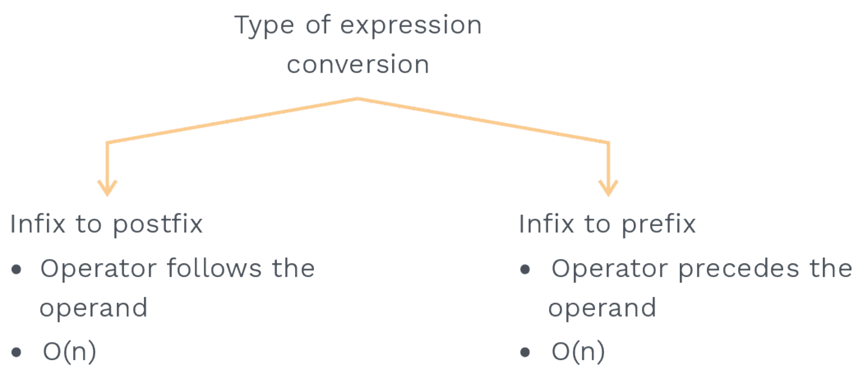

Type of expression conversion
Infix to postfix
• Operator follows the operand
• O(n)
Infix to prefix
• Operator precedes the operand
• O(n)

Fig. 3.6

Note:

The order of the operators in the postfix expression determines the actual order of operations in evaluating the expression, hence making the use of parenthesis in postfix notation is unnecessary.

Infix	Postfix
A + (B * C)	ABC * +
(A + B) * C	AB + C *

Grey Matter Alert!

The prefix form of an expression is not the mirror image of the postfix form.

Example:

$$\begin{tikzcd} & \text{Postfix : AB + C –} \\ \text{Infix : A + B – C} \arrow[ur] \arrow[dr] & \\ & \text{Prefix : – + ABC} \end{tikzcd}$$

Infix to postfix conversion:

1) Check for parenthesis;, the operations enclosed within parenthesis are to be converted first.
2) The other operations in the expression are converted according to the operator precedence.

Example: A + (B * C)

1) Parentheses (B * C) are converted first.

A + (BC *)

2) Now (BC *) is treated as a single operand, and then the entire expression is converted.

$A + \underbrace{(BC\ *)}_{\text{Single operand}}$

3) The postfix expression obtained: ABC * +

Infix to postfix conversion using stack:

An algorithm than can be designed to convert an infix expression to postfix express is given as:

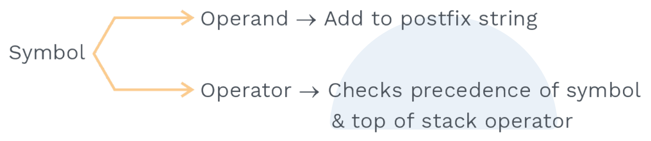
```
Operator_stack = empty stack;
while(!End of input)

    {       symbol = next input character;
                    if (symbol == operand)
                    add symbol to the postfix string;
                    else
                    {  while  (!empty  (operator_stack)  &  precedence  (Top
(operator_stack), symbol))
                    {       temp = pop (operator_stack);
                            add temp to the postfix string;
                    } //end of while
              Push (operator_stack, symbol);
         } //end of else
} //end of while
while(!empty(operator stack))
              {       temp = pop (operator_stack);
                      Add temp to the postfix string;
              }       //output all remaining operators
```

The algorithm given can be understood in step as:

1) The operator-stack is the stack which has been initialized, to push operators from the input string.

2) Symbol is the variable which reads the next input symbol of the infix expression (input string).

3) The if condition checks whether the input symbol is an operand or operator.


Symbol
Operand → Add to postfix string
Operator → Checks precedence of symbol & top of stack operator

4) If the precedence of the operator on top of the stack is greater, then the operator in the input string (symbol) then pops the top of the stack and add to postfix expression, then PUSH the symbol onto the stack. Otherwise, simply push the symbol onto the stack.

5) If the input string is completely scanned, then pop all operators from the operator-stack and add to the postfix string.

Example: A + B * C

S.no	Symbol	Postfix string	Operator-stack
1)	A	A	+
2)	+	A	+
3)	B	AB	+
4)	*	AB	+*
5)	C	ABC	+*

S.no	Symbol	Postfix string	Operator-stack
6)		ABC*	+
7)		ABC*+	

Table 3.5

Postfix string: ABC * +

Note:

The operator-stack has operators that work according to right associativity of operators:

$$+ \; \overset{}{*} \xrightarrow{} \text{Top}$$

Example: ((A – (B + C))/D) $ (E + F)

S. no	Symbol	Postfix string	Operator-stack
1)	(		(
2)	(		((
3)	A	A	((
4)	–	A	((–
5)	(	A	((– (
6)	B	AB	((– (
7)	+	AB	((– (+
8)	C	ABC	((– (+
9)	)	ABC +	((–
10)	)	ABC + –	(
11)	/	ABC + –	(/

S. no	Symbol	Postfix string	Operator-stack
12)	D	ABC + – D	(/
13)	)	ABC + – D/	
14)	$	ABC + – D/	$
15)	(	ABC + – D/	$(
16)	E	ABC + – D/E	$(
17)	+	ABC + – D/E	$(+
18)	F	ABC + – D/EF	$(+
19)	)	ABC + – D/EF +	$
20)		ABC + – D/EF + $	

Table 3.5

Postfix string: ABC + – D/EF + $

Evaluating postfix expression using stack:
The following algorithm evaluates an expression in postfix notation:

```
operand_stack = empty stack;
while (! end of input)
{symbol = next input character;
if (symbol == operand)
            PUSH (operand_stack, symbol);
else {
            operand 2 = pop (operand_stack);
            operand 1 = pop (operand_stack);
            value = result of apply symbol on operand 1 and operand
2.
            /* value = operand 1 symbol operand 2 */
            PUSH (operand_stack, value);
                  }
            } return (pop(operand_stack));
```

Rack Your Brain

Convert these postfix expressions to infix.

XY – P + QRS – + $

PQRST – + $ * AB * –

The algorithm given for postfix expression evaluation can be understood as:

1) The operand_stack is an empty stack where the operands are pushed into.

2) Symbol is the variable which reads the next input symbol of the postfix expression.

3) The if condition checks whether the input symbol is an operand or an operator.

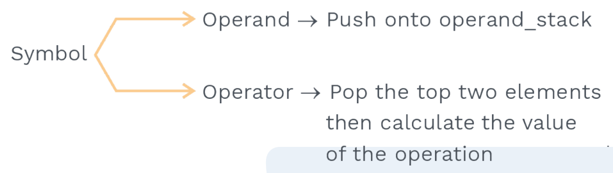

Symbol
Operand → Push onto operand_stack
Operator → Pop the top two elements then calculate the value of the operation

Fig. 3.8

- Keep the value of the top of stack in operand 2 and the next value in the operand 1 then apply operator:
  Value = operand 1 operator operand 2

Example:

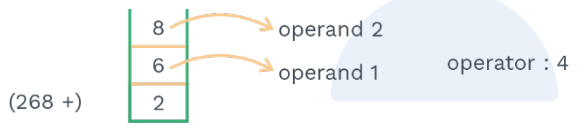


Fig. 3.9

Value = 6 + 8

Value = 14

4) Then, push the value onto the operand stack.

5) Once the entire string is traversed or read, then pop the top of the stack which returns the value of the expression.

Example: 26 12 13 + – 3 18 2 / + * 2 $ 3 +

S. no	Symbol	Operand 1	Operand 2	Value	Operand_stack
1)	26				26
2)	12				26 12
3)	13				26 12 13
4)	+	12	13	25	26 25

S. no	Symbol	Operand 1	Operand 2	Value	Operand_stack
5)	–	26	25	1	1
6)	3	26	25	1	1 3
7)	18	26	25	1	1 3 18
8)	2	26	25	1	1 3 18 2
9)	/	18	2	9	1 3 9
10)	+	3	9	12	1 12
11)	*	1	12	12	12
12)	2	1	12	12	12 2
13)	$	12	2	144	144
14)	3	12	2	144	144 3
15)	+	144	3	147	147

Table 3.6

The value of the expression (26 12 13 + – 3 18 2 / + * 2 $ 3 +) is 147.

Infix to prefix conversion:

Some of the steps to follow to convert a complex infix expression into prefix form:

1) Read the expression in reverse from right to left. Convex the open braces as closed and vice versa.

2) Now, the reversed expression obtained is converted into postfix notation using stack.

3) Now, for the final step, reverse the postfix notation i.e. the postfix expression read from right to left results in a prefix notation expression.

Rack Your Brain

Why infix to postfix conversion is required?

Example: (4 + (6 * 2) * 8)

Step I: Reverse the infix expression

(4 + (6 * 2) * 8) ← Reading from right to left

(8 * (2 * 6) + 4)

Step II: Convert (8 * (2 * 6) + 4) into postfix.

Postfix: 826 ** 4 +

Step III: Reverse the postfix notation.
826 ** 4 + ← Read from right to left.
Prefix: +4 ** 628

Evaluation of prefix expression:

- The prefix expression evaluation uses the same steps as the postfix expression evaluation just that the input expression is read from right to left.

Rack Your Brain

Convert the following prefix expression to infix + – $ ABC * D **EFG

Example: +4 * * 628 ← Read

S. no	Symbol	Operand 1	Operand 2	Value	Operand stack
1)	8				8
2)	2				8 2
3)	6				8 2 6
4)	*	2	6	12	8 12
5)	*	8	12	96	96
6)	4	8	12	96	96 4
7)	+	96	4	100	100

Table 3.7

The value of prefix expression (+4 * * 628) is 100.

Previous Years' Question

The following postfix expression with single digit operands are evaluated using a stack: 8 2 3 ^ / 2 3 * + 5 1 * - Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

1) 6, 1 **2)** 5, 7 **3)** 3, 2 **4)** 1, 5

Sol: 1) **(GATE CSE- 2007)**

Previous Years' Question

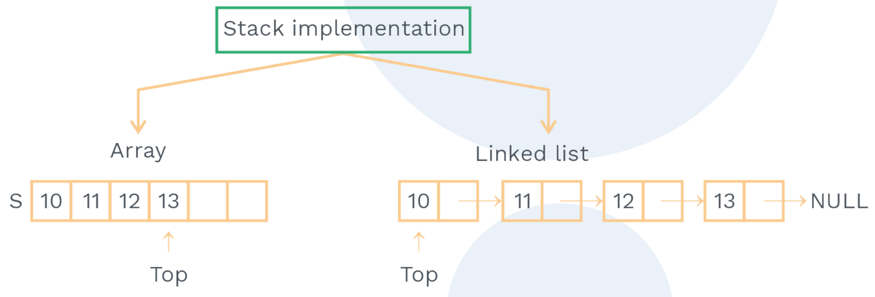
Assume that the operators +, –, ×, are left associative, and ^ is right associative. The order of precedence (from highest to lowest) is ^, ×, +, –. The postfix expression corresponding to the infix expression a + b×c–d^e^f is

1) abc×+def^^–   **2)** abc×+de^f^–

3) ab+c×d–e^f^   **4)** – + a×bc^^def

Sol: 1)   **(GATE CSE- 2004)**

3.5 IMPLEMENTATION OF STACK



PUSH (S, Top, x)

1) Top = Top + 1;

2) S[Top] = x;

1) Temp = S[Top];

2) Top = Top – 1;

3) Return temp;

- Push & pop both take 0(1) time.

Overflow condition

⇒ (Top == size of array)

Underflow condition

⇒ (Top == –1)

PUSH (Top, x)

1) Temp = x;

2) Temp → next = Top;

3) Top = temp;

POP(S)   POP (Top)

1) Temp = Top;

2) Top = Top → Next;

3) a = temp → info;

4) Free (temp);

5) Return (a);

- Push & pop both take 0(1) time.

Overflow condition

⇒ (AVAIL == NULL) linked list have dynamic memory allocation

∴ No limit till memory is available.

Underflow condition

⇒ (Top == NULL)

3.6 APPLICATION OF STACK: TOWER OF HANOI

The tower of Hanoi problem is described as follows:

Suppose there are three pegs labeled A, B and C. On peg A a finite number of n disks are placed in order of decreasing diameter. The objective of the problem is to move the disks from peg A to peg C using peg B as an auxiliary. The rules are as follows:

1) Only one disk can be moved at a time.

2) Only the top disk on any peg can be moved to any other peg.

3) At no time can a larger disk be placed on a smaller disk.

The solution to a Tower of Hanoi problem for n = 3.

n = 3: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.

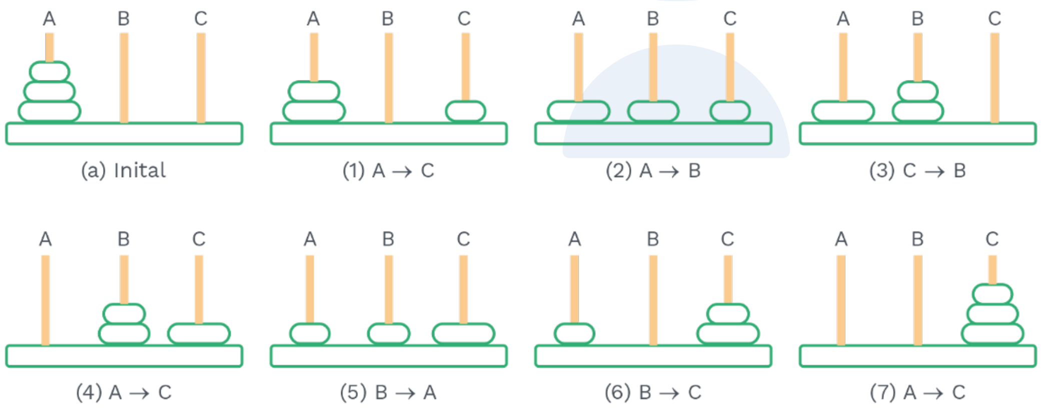


Fig. 3.11 The Steps of Tower's of Hanoi

Tower of hanoi solution:

n	Steps
1)	A → C
2)	A → B, A → C, B → C
3)	A → C, A → B, C → B, A → C, B → A, B → C, A → C,

Generalized Solution for n

1) Move the top (n – 1) disks from peg A to peg B
2) Move the top disk from peg A to peg C
3) Move the top (n – 1) disks from peg B to peg C

Let Peg A referred to as BEG, Peg C as END and Peg B as AUX then:

1) TOH (N – 1, BEG, END, AUX) **2)** TOH (1, BEG, AUX, END)
3) TOH (N – 1, AUX, BEG, END)

The tower of Hanoi (TOH) can be divided into three sub problems and can be solved recursively as:

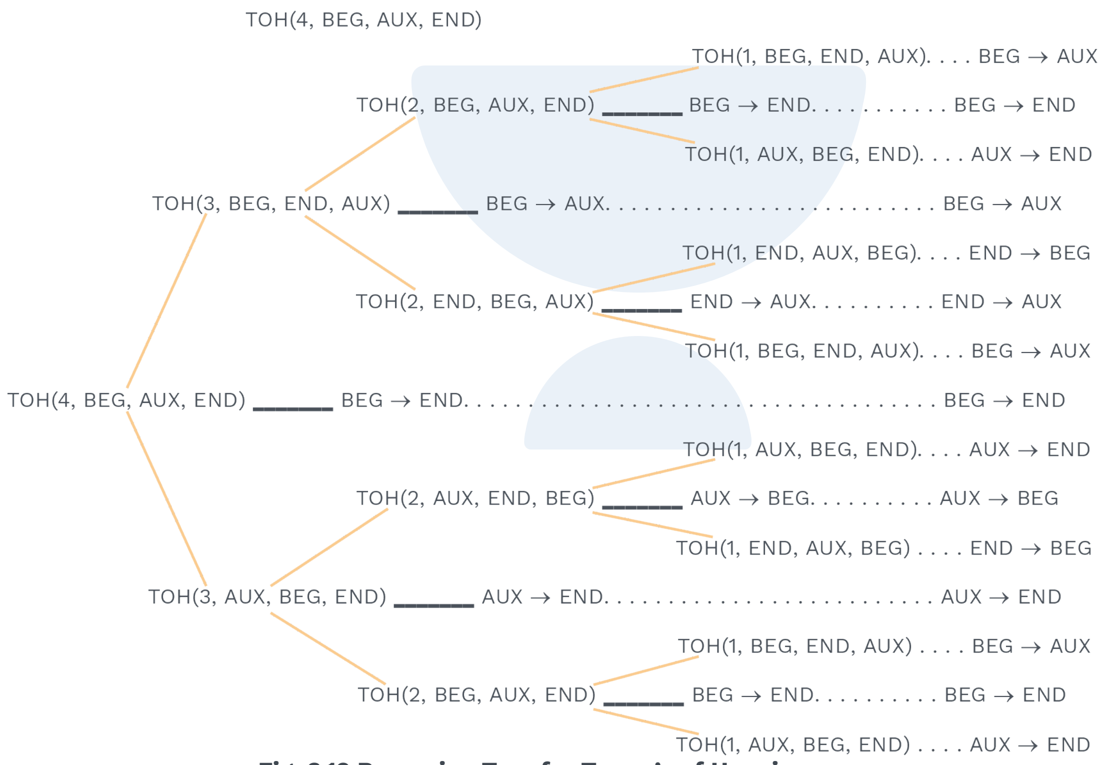
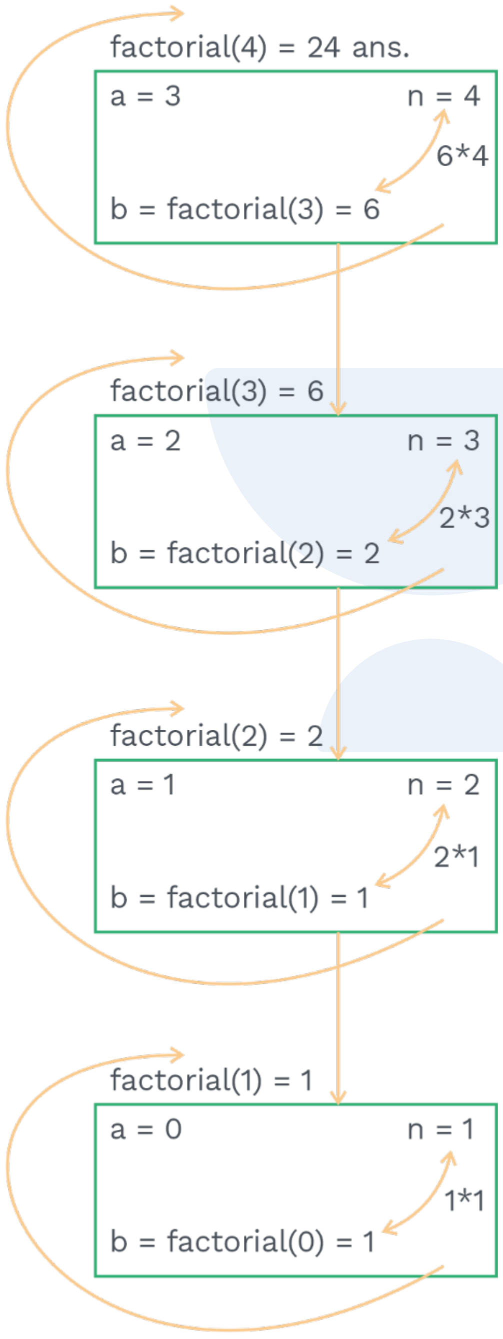

TOH(4, BEG, AUX, END)
TOH(1, BEG, END, AUX). . . . BEG → AUX
TOH(2, BEG, AUX, END) BEG → END. . . . . BEG → END
TOH(1, AUX, BEG, END). . . . AUX → END
TOH(3, BEG, END, AUX) BEG → AUX. . . . . BEG → AUX
TOH(1, END, AUX, BEG). . . . END → BEG
TOH(2, END, BEG, AUX) END → AUX. . . . . END → AUX
TOH(1, BEG, END, AUX). . . . BEG → AUX
TOH(4, BEG, AUX, END) BEG → END. . . . . BEG → END
TOH(1, AUX, BEG, END). . . . AUX → END
TOH(2, AUX, END, BEG) AUX → BEG. . . . . AUX → BEG
TOH(1, END, AUX, BEG) . . . . END → BEG
TOH(3, AUX, BEG, END) AUX → END. . . . . AUX → END
TOH(1, BEG, END, AUX) . . . . BEG → AUX
TOH(2, BEG, AUX, END) BEG → END. . . . . BEG → END
TOH(1, AUX, BEG, END) . . . . AUX → END

Fig. 3.12 Recursion Tree for Tower's of Hanoi

Factorial:

```
int n;
    factorial(n)
    {   int a, b;
        if(n == 0)
        return(1);
        a = n – 1;
        b = factorial(a);
        return(n * b);
    }
```

Evaluating the above function by taking n = 4, by using recursive tree evaluation of the function code:



Ans. 24

Fig. 3.13

Fibonacci number:

```
int n;
fibonacci(n)

{         int a, b;
                    if(n < = 1)
                    return(n);
                    a = fibonacci(n – 1);
                    b = fibonacci(n – 2);
                    return(a + b);
}
```

Evaluating the above function by taking n = 4 by using recursive tree evaluation of function code:

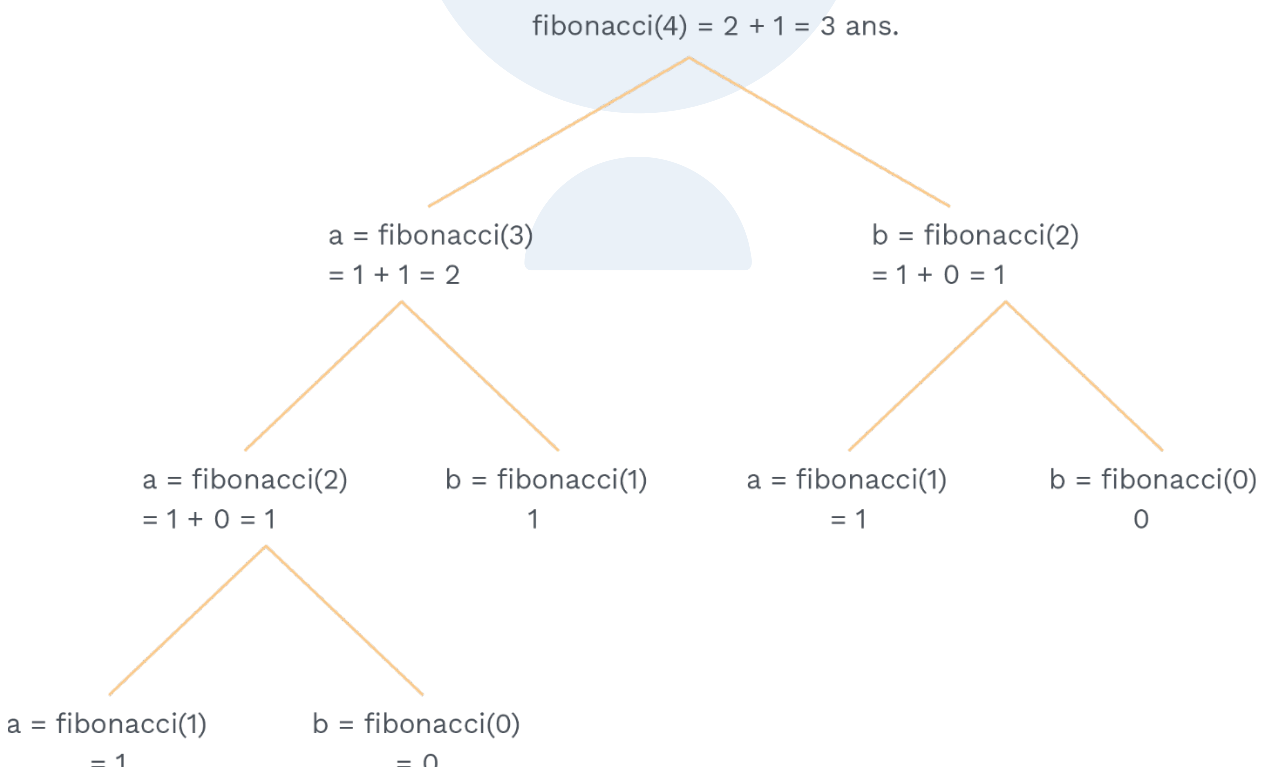


Fig. 3.14 Recursion Tree for Fibonacci(N)

Sol: 3)

3.7 BASICS OF QUEUE

- Queue is a data structure in which we insert and delete data items.
- The fashion in which we insert/delete data items is: ‘One side insertion and other side deletion’.
- Queue is represented as:

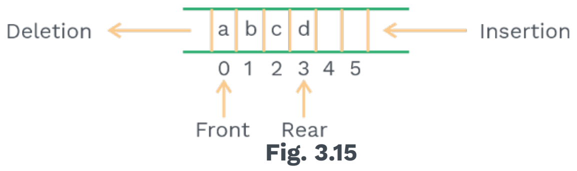


Fig. 3.15

- **Front:** It is a variable that contains position of the element to be deleted.
- **Rear:** It is a variable that contains position of the newly inserted element.
- For the above representation, Let’s find the front and rear:
  1) Front is 0 **2)** Rear is 3
- **Property of queue:**

FIFO: First In First Out
OR
LILO: Last In Last Out

In the given queue below, Let’s understand FIFO or LILO.
Consider a queue of size ‘6’.

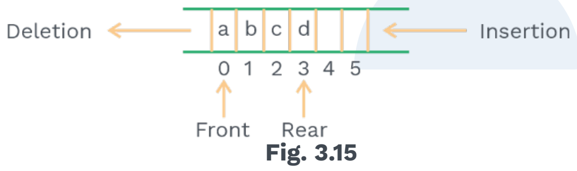


Fig. 3.15

Steps for insertion:

1) First, a is inserted in the queue at position ‘0’.
2) b is inserted at position 1.
3) c is inserted at position 2.
4) d is inserted at position 3.
Positions 4 and 5 are vacant.

Steps for deletion:

1) The first deletion from queue would be at position 0
2) Second deletion be at position 1
3) Third deletion be at position 2
4) Fourth deletion be at position 3
Clearly, ‘a’ was inserted first and deleted first and so on.

Note:

If we want to delete b at first from above queue, then it is not possible. Hence, queue works according to the property FIFO or LILO.

Note:

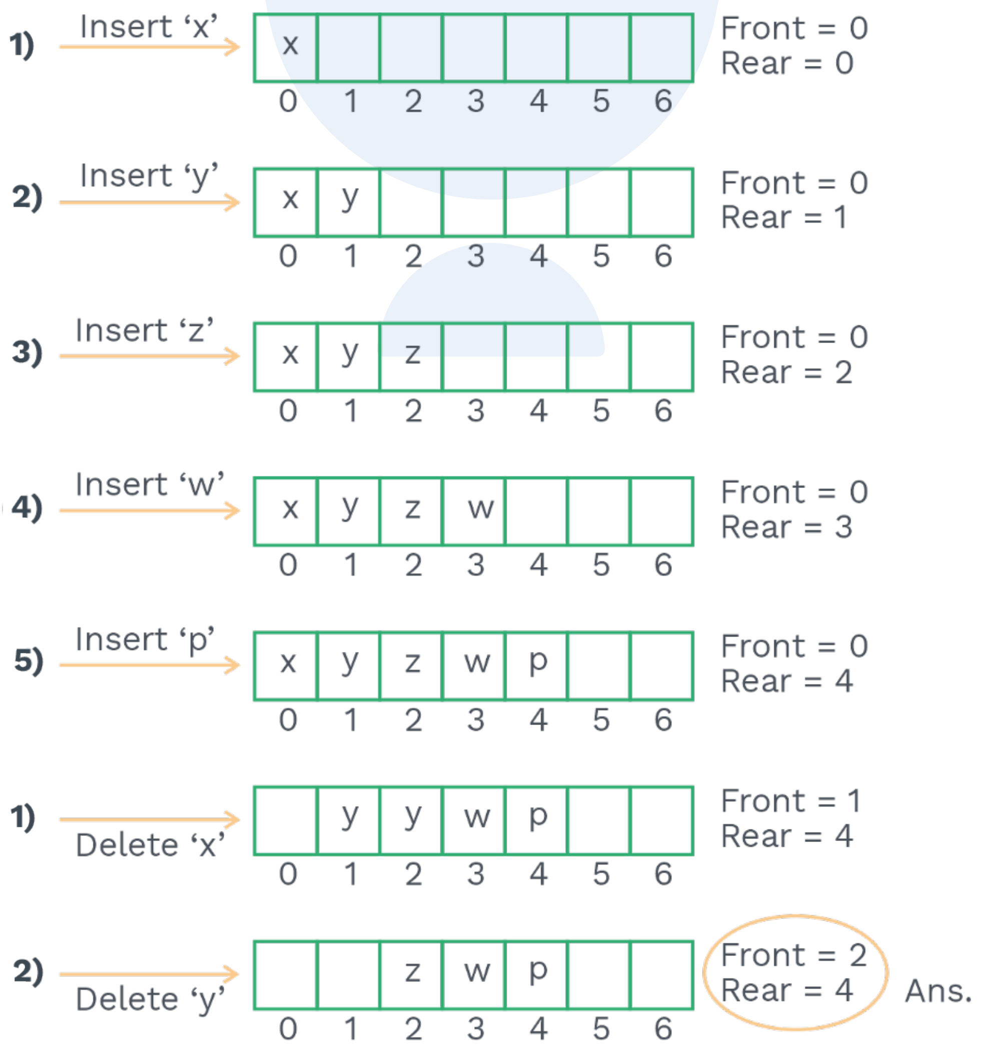
Queue–data structure is used in breadth first search traversal of graphs.

SOLVED EXAMPLES

Q1 **Consider a queue of size ‘7’. In this queue, five insertions and two deletions have been performed. Find the values of ‘Front’ and ‘Rear’.**

Sol: **Queue size is given as ‘7’.**

Let’s plot the queue and perform 5 insertions and 2 deletions ;



3.8 OPERATIONS ON QUEUE

The operations we can perform on queue is called ADT of queue.

(ADT: Abstract data type)

Rack Your Brain

You are given a queue of size 'n'. It's needed to perform 'n+3' insertions. What would be the result?

- **ADT of queue:**
  1) Enqueue ( )
  2) Dequeue ( )
  We perform two operations on queue, and they are enqueue and dequeue.

1) Enqueue ( ):
- Using enqueue, an element is inserted into the queue.
- To insert 'n' elements there are 'n' enqueue() operations needed.

2) Dequeue ( ):
- Using dequeue an element is deleted from the queue.
- To delete 'n' elements from the queue, 'n' dequeue() operations needed.

For Example:
Consider an empty queue of size 6,

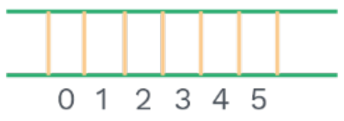
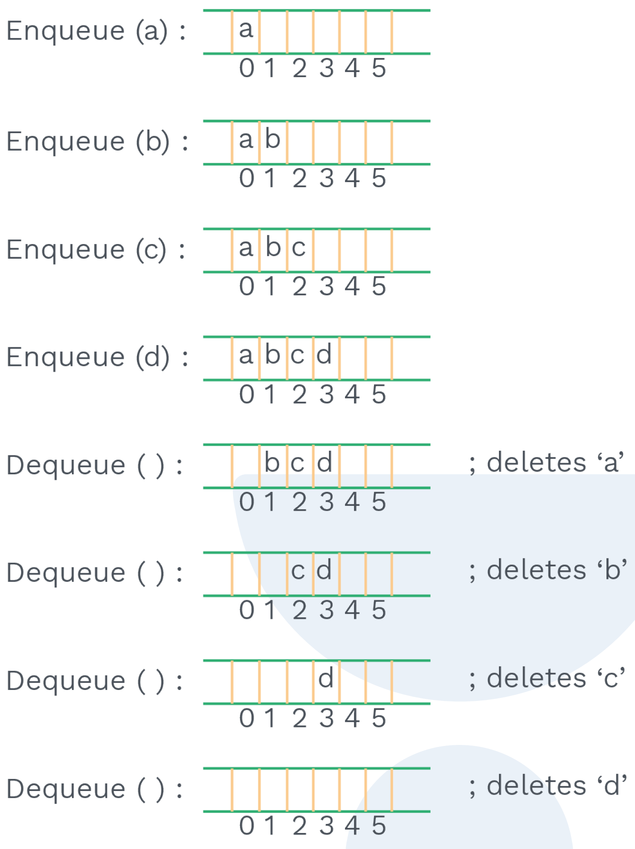


Fig. 3.17

Enqueue **a)**: Will insert 'a' at 0.

Enqueue **b)**: Will insert b at 1 and so on.

Dequeue ( ): Will delete a by default because it was inserted first and so on.



Enqueue (a) : a | 0 1 2 3 4 5
Enqueue (b) : a b | 0 1 2 3 4 5
Enqueue (c) : a b c | 0 1 2 3 4 5
Enqueue (d) : a b c d | 0 1 2 3 4 5
Dequeue ( ) : b c d | 0 1 2 3 4 5 ; deletes 'a'
Dequeue ( ) : c d | 0 1 2 3 4 5 ; deletes 'b'
Dequeue ( ) : d | 0 1 2 3 4 5 ; deletes 'c'
Dequeue ( ) : | 0 1 2 3 4 5 ; deletes 'd'

Fig. 3.18

3.9 TYPES OF QUEUE

There are different types of queue; Let's understand them one by one.

Circular queue:

- Linear queue is not that efficient because the rear can't go back once it reaches end of queue.

For example: Suppose a situation as shown in diagram

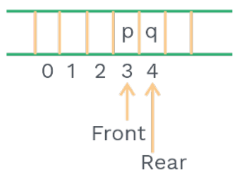


Fig. 3.19

and we need to insert an element to the above queue, but we can't. Even though lot of space is free in the queue. We can't insert any new element in queue because rear is at the last position.

- Circular queue is upgraded form of linear queue in which 'Front' and 'Rear' can be represented on a circular structure keeping first position and last position connected.
- Circular queue is represented as:

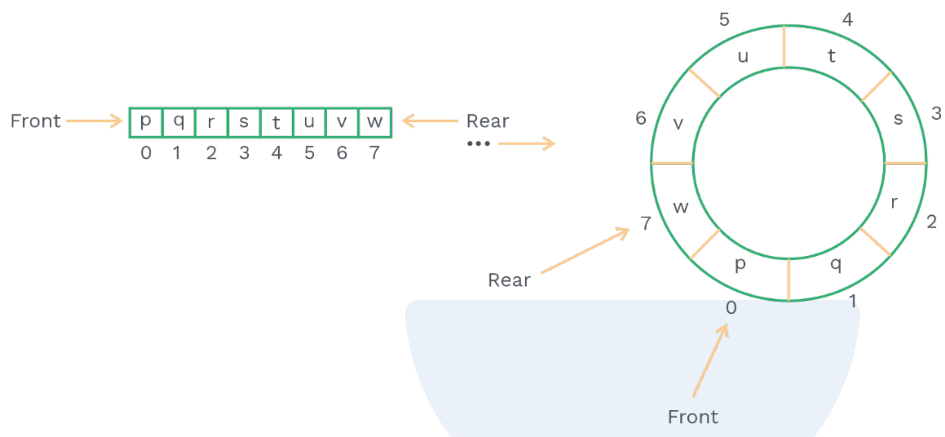


Fig. 3.20 Circular Queue

Priority queue:

In Priority queue, every element is associated with some priority and according to priority an element in the queue is processed, especially on deletion time.

So, while inserting elements on the queue, we can follow any order but, while deleting, we consider the requirement.

Rack Your Brain

Consider a queue of size '5'. Find the value of 'Front' and 'Rear' after – 5 insertions, 2 deletions and 2 insertions respectively.

Types of priority queue:

1) Ascending priority queue

2) Descending priority queue

1) Ascending priority queue:

We insert items arbitrarily in the queue.

While deleting, on the first deletion, it gives the smallest item of all, and so on.

For example: items are 3, 9, 2, 0, 11, 10, 20

- Insertion and elements follow as:

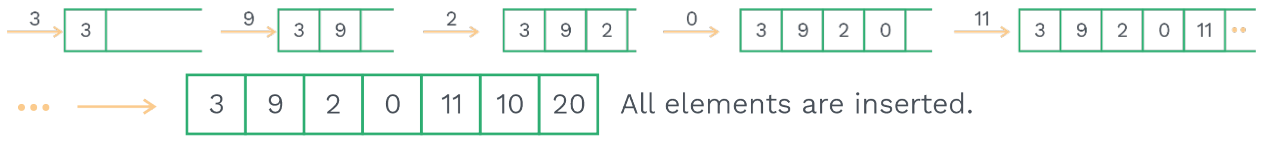


Fig. 3.21

Deletion of elements follow as:

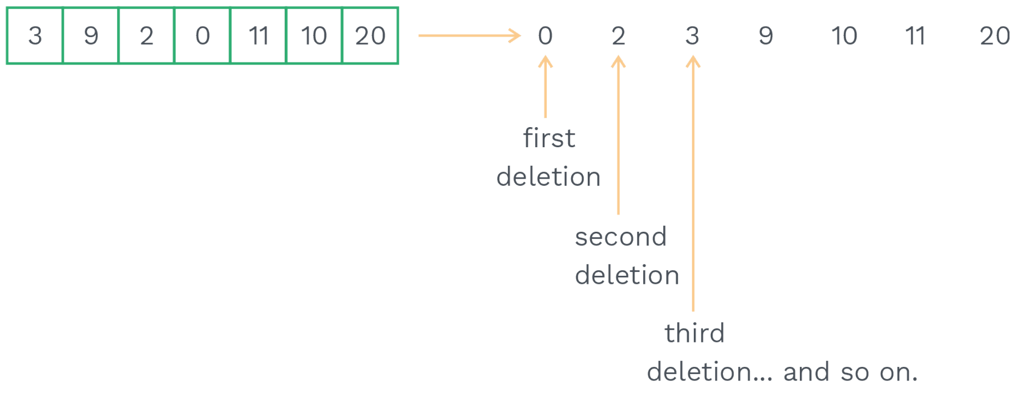


Fig. 3.22

Thus, by ascending the priority queue, we get items from the queue in ascending order.

2) Descending priority queue:

- Insertion of items on queue follows an arbitrary order.
- Deletion follows as: On first deletion, it gives the largest item of all and so on.

  For example: Items are 3, 9, 2, 0, 11, 10, 20.
- Insertion of items follows as:

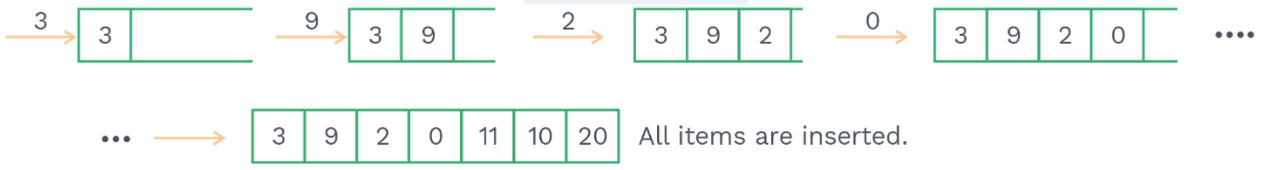


Fig. 3.23

Deletion of elements/items follow as:

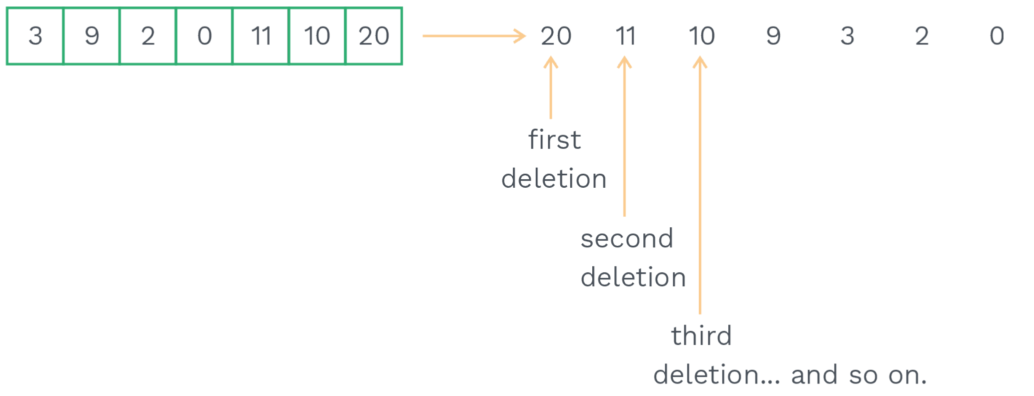


Fig. 3.24

Thus, by descending the priority queue, we get items from the queue in descending order.

Rack Your Brain

There is ascending priority queue of size 10. We are given ten numerals to insert into the queue. After insertion, we performed deletion till 5th position. What is the result? (Take positions of items, starting from 0, 1 and so on).

Previous Years' Question

A priority queue Q is used to implement a stack that stores characters. PUSH (C) is implemented INSERT (Q, C, K) where K is an appropriate interger key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in

a) non–increasing order **b)** non–decreasing order
c) strictly increasing order **d)** strictly decreasing order

Sol: d) **(GATE 1997 - 2 Marks)**

Doubly ended queue (Dequeue):
A doubly ended queue is a kind of queue in which insertion, and deletion operations are performed at both places of 'Front' and 'Rear'.

- Let's understand it with a diagrammatic view:

Consider a queue of size '7', where 'F' represents 'Front' and 'R' represents 'Rear'.

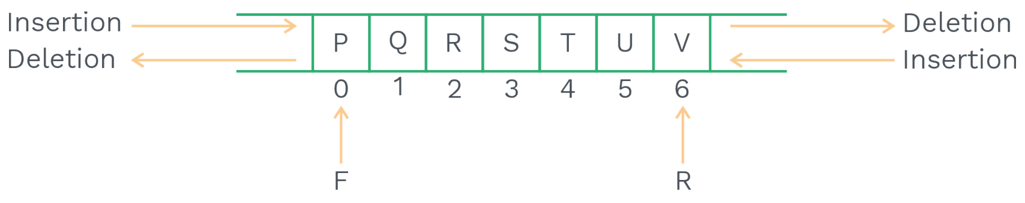


Fig. 3.25

As shown in the diagram, clearly, we can insert or delete items on both 'F' and 'R'.

Rack Your Brain

A doubly ended queue is given below. Calculate the value of 'Front' and 'Rear' after the following sequence of operations :

1) Two–deletion at 'Front'

2) Three–deletions at 'Rear'

3) Two–insertions at Rear'.

Input restricted queue:

Input restricted queue is a kind of queue in which insertion operations are performed only at 'Rear' as in regular queue, but deletion operations are performed at 'Front' and 'Rear' both the positions.

Let's understand it with a diagrammatic view:

Consider the queue, in which 'F' represents 'Front' and 'R' represents 'Rear'.

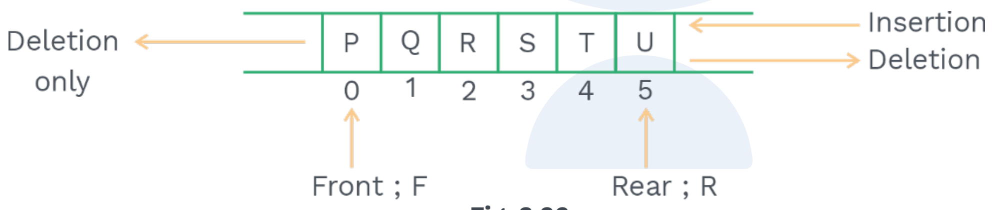


Fig. 3.26

In the above diagram, it can be seen that items can be deleted from 'Rear' also.

Output restricted queue:

Output restricted queue is a kind of queue; deletion operations are performed only at 'Front' as in regular queues, but insertion operations can be performed at 'Front' and 'Rear' both the positions.

A diagrammatic representation can be shown as:

Consider the queue, in which 'F' represents 'Front' and 'R' represents 'Rear'.

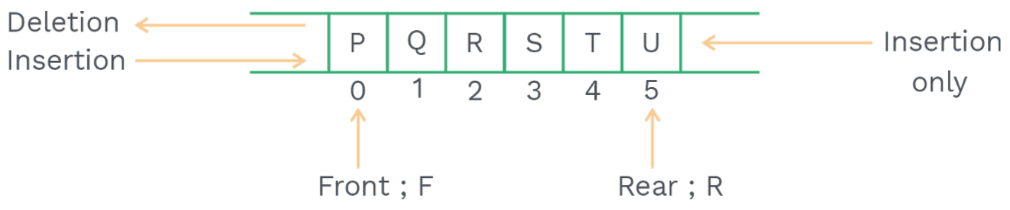


Fig. 3.27

In above diagram, It can be seen that items can be inserted at 'Front' also.

Note:

Priority queue, input restricted queue, output restricted queue, doubly ended queue, all these kinds of queue are not guaranteed to satisfy FIFO property.

3.10 IMPLEMENTATION OF QUEUE

We can implement queue by using other data structures like using array, stack or linked list etc.

Implementation of queue using array:

- Implementing queue means implementing basic operations of queue by using array.
- So, let's take an array first ;
- Array is denoted with 'q', which represents a queue.
- 'S' is the size of array, 'q'.
- Front is represented with 'F'.
- Rear is represented with 'R'.

Enqueue operation:

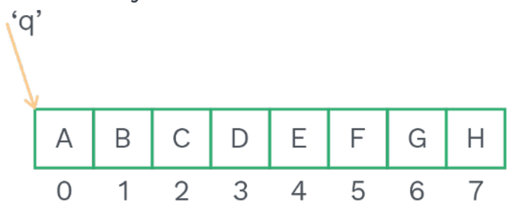
```
Void insert(x)          // An item 'x' is be inserted.
 {
        If (R+1 == S)       // Checking overflow condition.
          {
               printf ("q is overflow") ;
               exit(1) ;
          }
        else
          {
               if (R == –1)      // Checking if queue 'q' is
               F ++ ;               empty, if yes–performs
               R ++ ;               first insertion of an item.
               else

               R = R+1 ;         // Normally, if there is space,
                                    rear increments one by one.

               q[R] = x ;        // item 'x' is stored at location R.

          }
 }
```

Dequeue operation:

```
int remove ( )
   {
      int I ;
      If (F == –1)        // Checking, if ‘q’ has no element.
        {
           printf (“q is underflow”) ;
           exit (1) ;
        }
      else
        {
         I = q[F] ;          // Storing elemented from position ‘F’
                                to integer variable I.

         If (F==R)           // It performs last element removal.
         F = R =–1 ;
         else

         F = F + 1 ;          // Normally, if there are some items  in queue, front
         return (I) ;            increments one by one after deletion of elements sequentially.

        }
   }
```

Assumptions:

- Array positions start from zero i.e. $\underbrace{0, 1, 2 \ldots\ldots S-1}_{\text{Total} \quad S}$
- Initially, when there is no item in queue, F = –1 and R = –1.
- If at any stage F = R, there is only one element in the queue.
- In general, ‘R’ increments by one for each insertion.
- In general, ‘F’ increments by one for each deletion.
- So, queue can be represented by using an array as:



Here, S = 8 i.e. size of queue.

Rack Your Brain

There is queue ‘Q’ implemented with an array. In this queue front is represented with ‘F’, and rear is represented with ‘R’. size of ‘Q’ is S. Write the condition for (i) overflow of queue ‘Q’.
(ii) Underflow of queue ‘Q’.

Previous Years' Question

A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is correct (n refers to the number of items in the queue)?

a) Both operations can be performed in O(1) time

b) At most one operation can be performed in O(1) time but the worst case time for the other operation will be Ω(n)

c) The worst case time complexity for both operations will be Ω(n)

d) Worst case time complexity for both operations will be Ω(logn)

Sol: a) **(GATE CSE 2016 - SET 1)**

Previous Years' Question

Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are :

i) isEmpty (Q) – returns true if the queue is empty, false otherwise.

ii) delete (Q) – deletes the element at the front of the queue and returns its value.

iii) insert (Q, i) – insert the integer i at the rear of the queue.

Consider the following function :

```
void f (queue Q) {
     int i ;
     if (! isEmpty(Q) ) {
         i = delete(Q) ;
         f (Q) ;
         insert (Q, i) ;
     }
}
```

What operation is performed by the above function f?

a) Leaves the queue Q unchanged

b) Reverses the order of the elements in the queue Q

c) Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order

d) Empties the queue Q

Sol: b) **(GATE CSE - 2007)**

Implementation of circular queue using array:

- Implementing a circular queue means implementing basic operations of the circular queue by using array.
- Let's take an array 'q'.
- 'S' is the size of a circular queue i.e., 'q'.
- Front is represented with 'F'.
- Rear is represented with 'R'.

1) Enqueue operation: (To insert items)

```
void C_insert(x)        // x is an item to be inserted.
{
        if ((R+1) mod S == F)
            {
                printf ("CQ is full") ;      // circular queue is full.

                exit(1) ;
             }
        else
             {
                if (R == –1)
                   {
                       F++ ;
                       R++ ;
                   }
                else
                       R = (R+1) mod S ;  // This condition will use the
                                             array in a circular manner, to
                                             insert an item on the queue.
             }
q [R] = x ;
}
```

The Above code will insert items in a circular manner by the testing status of a queue in circular manner.

2) Dequeue operation: (To remove items)

```
int C_remove( )
    {
        int I ;
        if (F == –1)
          {
              printf (“Underflow”) ;
              exit(1) ;
          }
        else
          {
              I = q [F] ;
              if (F == R)
                  F = R = –1 ;
              else
                  F = (F+1) mod S ;    // This condition will use the array to
                                          remove items in a circular manner.
              return I ;
          }
    }
```

Assumptions:

- Array position starts from zero i.e. 0, 1, 2 .... S–1.
- Initially when there is no any item in C_queue F = –1, R = –1.
- If at any stage F = R, there is only one item in the C_queue.

Previous Years' Question

Suppose a circular queue of capacity (n–1) elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are

a) full : (REAR+1)
mod n == FRONT
empty : REAR == FRONT

b) full : (REAR+1)
mod n == FRONT
empty : (FRONT+1)

c) full : REAR == FRONT
empty : (REAR+1)
mod n == FRONT

d) full : (FRONT+1)
mod n == REAR
empty : REAR == FRONT

Sol: a) **(GATE CSE - 2012)**

Implementation of queue using stack:

- **A brief note on stack:**

A stack is a data structure which functions according to property as last in first out. Items are pushed onto stack and can be popped out of it.

Stack has two basic operations:

1) Push( ): To push items on stack
2) Pop( ): To pop items from stack.

- Implementation of queue using stack means, performing basic operations of queues by using basic operations of a stack and satisfying the property of queue.

- **Basic operations of queue are:**

  1) Enqueue (to insert)
  2) Dequeue (to remove)

- Property that should be satisfied for queue is first in first out (FIFO).
- So, we have to satisfy the property FIFO by using a push–pop operation.

For example:

- We want a queue whose items are P, Q, R, S, T, U, V.
  - * Let’s implement this queue by using stack.

1) Enqueue operation: (To insert items on queue)

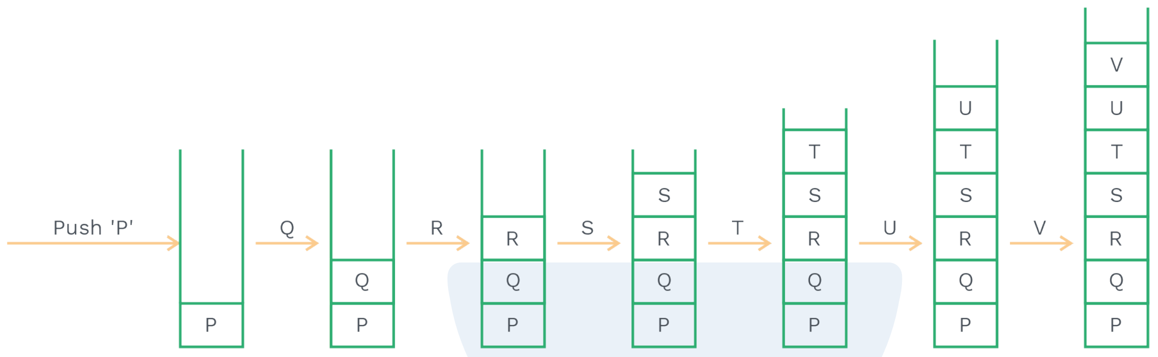


Fig. 3.29

By performing above operations, we enqueued all the items successfully.
So, enqueue operation can be given as:

```
Insert (x)
{
   Push(x, ST) ;
}
```

x : is an item to be inserted into queue

ST : is the stack.

Push(x, ST) : performs push item x on stack ST.

2) Dequeue operation: (To delete items from queue)

Till now, we have the stack :
(means queue) where we
have inserted all the items.

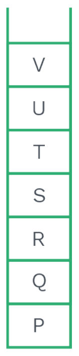


Fig. 3.30

- We need to delete first item, second item and so on from stack which we had pushed. For that, we need two stacks. The first stack is ST, and second stack is S’T.

- We need to pop all items from ST and pushed onto S'T. Thus we will get the first item inserted onto ST as top of stack_S'T, and we can get the first item deleted easily.

Operations:

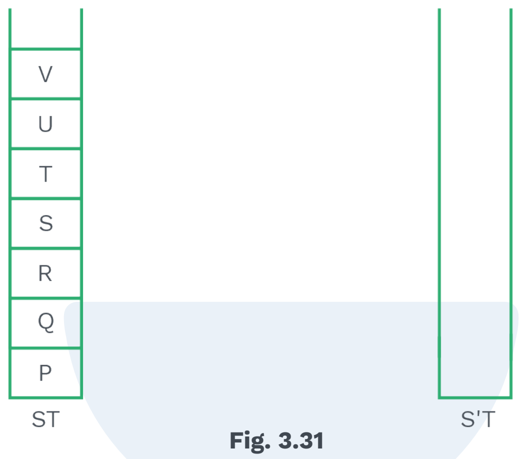


Fig. 3.31

1. Pop (V, ST)	8. Push (S, S'T)
2. Push (V, S'T)	9. Pop (R, ST)
3. Pop (U, ST)	10. Push (R, S'T)
4. Push (U, S'T)	11. Pop (Q, ST)
5. Pop (T, ST)	12. Push (Q, S'T)
6. Push (T, S'T)	13. Pop (P, ST)
7. Pop (S, ST)	14. Push (P, S'T)

- So,

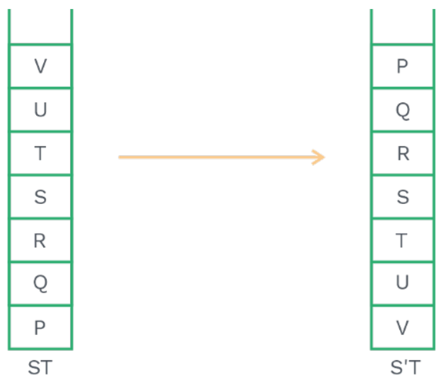


Fig. 3.32

Now, pop (P,S'T) 1st dequeue

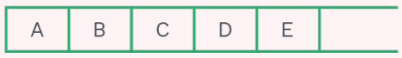
- **So, the dequeue operation can be given as:**

```
Remove( )
{
            if (S'T has some elements/items)
                return (pop (x, S'T))
            else
                {
                    while (ST has some elements/items)   // While loop
used for
                          {                            continuously
popping an item
                                I = Pop (x, ST);              from   ST
which are supposed
                                                          to be pushed
onto S'T.
                          }
                    return (Pop (x, S'T)) ;
                 }
}
```

Thus, it's successfully done using stack. i.e. queue is implemented by using stack.

Rack Your Brain

A queue is considered to be implemented by using stack. No. of items is 5 as shown in the diagram



Element–D has to be removed. How many push–pop operations are needed?

Previous Years' Question

An implementation of a queue Q, using two stacks S1 and S2, is given below :

```
void insert (Q, X) {
    push (S1, X) ;
}
void delete (Q) {
    if(stack_empty(S1)) then {
    print ("Q is empty") ;
    return ;
} else while (!(stack_empty(S1))) {
    X = pop(S1) ;
    push(S2, X);
}
X = pop(S2) ;
}
```

Let, n insert and m($\leq$ n) delete operations to be performed in an arbitrary order on an empty queue Q.
Let, x and y be the number of push and pop operations performed respectively in the process.
Which one of the following is true for all m and n?
a) $n + m \leq x \leq 2n$ and $2m \leq y \leq n + m$
b) $n + m \leq x \leq 2n$ and $2m \leq y \leq 2n$
c) $2m \leq x \leq 2n$ and $2m \leq y \leq n + m$
d) $2m \leq x \leq 2n$ and $2m \leq y \leq 2n$
Sol: a) **(GATE CSE - 2006)**

Implementation of queue using linked list

- **A brief note on linked list:**

Linked list is a data structure which is in the form of list of nodes as shown in the diagram below:

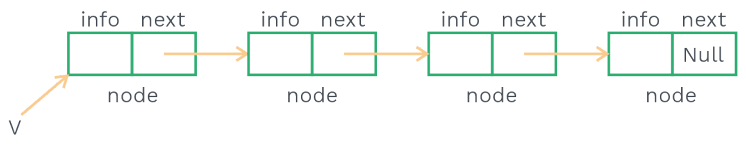


Fig. 3.33

- A node contains two fields:
  1) An information field (we can use an integer data or char data etc.)
  2) Next address field
- An external pointer 'V' that points to the first node, and we can access the entire list by this external pointer. 'V' is a pointer variable which contains the address of the first node of the linked list.
- Implementing queue using linked list means performing enqueue and dequeue operations on the linked list to satisfy the condition of first in first out, which is of queue's property.
- **Definition of two operations of queue using linked list:**
  1) Enqueue/insert operation:
  It means we need to add a node at the end of a given linked list.
  2) Dequeue/remove operation:
  It means we need to delete a node from starting of a given linked list.
- Let's understand with an example:

Consider a linked list:

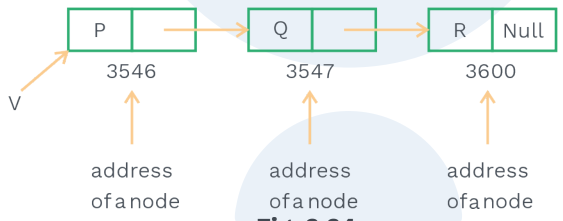


Fig. 3.34

1) Enqueue/insert operation:

Steps: **I)** – Create a node

II) – Attach it at the end of the linked list and so on

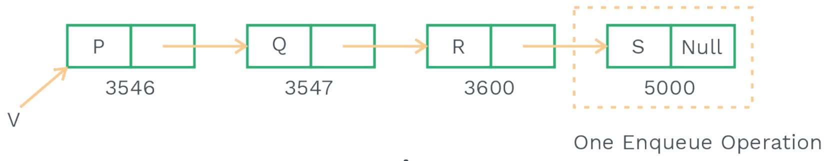


Fig. 3.35

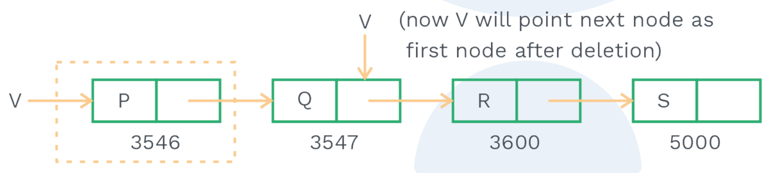
- **Code for enqueue operation:**

```
Insert(x)          // x is an item/data to be inserted.
{
   New_node = malloc( ) ;     // Creation of a dynamic memory space
                                for a new node.
   if (New_node == Null)     // When memory is over, New_node
                                won't get any space.
```

```
            {
            printf ("overflow") ;
            exit(1) ;
            }
            S = V ;        // Taking another pointer variable 'S'.

            While (S → next ! = Null)     // 'S' traverses to the
            {                                end of the linked list.
            S = S → next ; }
            S → next = New_node ;
            S = new_node ;

            }
   }
```

2) Dequeue/remove operation:

Steps: I – Just remove the first node, second node and so on.



One Dequeue Operation

Fig. 3.36

- **Code for dequeue operation:**

```
Remove( )
{
            int I ;

Removed_node = V ;    // assigning the value of pointer variable 'V' to a
                         new pointer variable which is Removed_node.

I = Removed_node → data ;    // Taking Removed_node data out to integer
                                variable–I.

V = V → next ;

free (Removed_node) ;    // deallocating the node pointed by Removed_node.

Removed_node = Null ;
return (I) ;
}
```

So, now list will appear as:

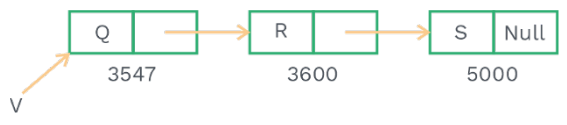
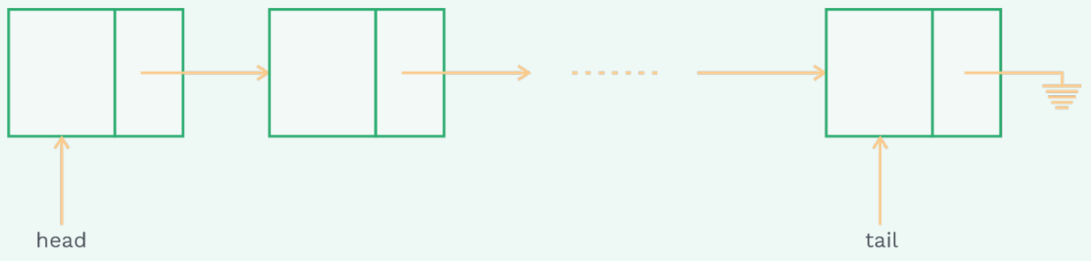


Fig. 3.37

Thus, we can perform insert/remove operation of the queue using linked list.

Previous Years' Question

A queue is implemented using a non–circular singly linked list. The queue has a head pointer and tail pointer, as shown in the figure. Let n denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head and dequeue be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time–efficient implementation of enqueue and dequeue, respectively, for this data structure?

a) θ(1), θ(1) **b)** θ(1), θ(n)
c) θ(n), θ(1) **d)** θ(n), θ(n)

Sol: b) **(GATE CSE - 2018)**

Chapter Summary

- Stack is a linear data structure in which the items can be added or removed only at one end called top of stack. Stack satisfied LIFO property.
- Basic operations of stack : **1)** PUSH
  2) POP
- Stack conditions: **a)** Underflow condition
  b) Overflow condition
- Expression evaluation:
  We need to convert infix expression to postfix expression for evaluation.
  → Stack is only used in both conversion and evaluation.
- Stack can be implemented with array as well as linked list.
- Tower of Hanoi is an application of stack.
- Queue is a data structure which follows the first in first out property.
- Operations of queue: **1)** Enqueue/insert operation
  2) Dequeue/remove/delete operation
- Types of queue:
  1) Circular queue: A circular queue in which the first position and last position is connected.
  2) Priority queue: While inserting elements into the queue we can follow any order, but while deleting, we consider the priority.
  Types: Ascending priority queue and
  Descending priority queue.
  3) Doubly ended queue:
  It is a kind of queue in which insertion and deletion operations are performed at front and rear both places.
  4) Input restricted queue:
  Insertion operations are performed only at the rear.
  5) Output restricted queue:
  Deletion operations are performed only at the front.
- Implementation of queue:
  1) Using array:
  It means implementing basic operations of queue to satisfy FIFO using array.
  2) Using stack:
  This means implementing queue using Push–Pop operations to satisfy FIFO.
  3) Using linked list:
  It means implementing queue by adding or deleting nodes to satisfy FIFO.