UNIT –II

Stacks & Queues: Stack – Definitions – Concepts – Operations on Stacks – Infix, postfix & prefix conversions - evaluations of expressions using stack - Applications of stacks – Representation of Queue – Insertion and Deletion Operation – Applications of Queue.

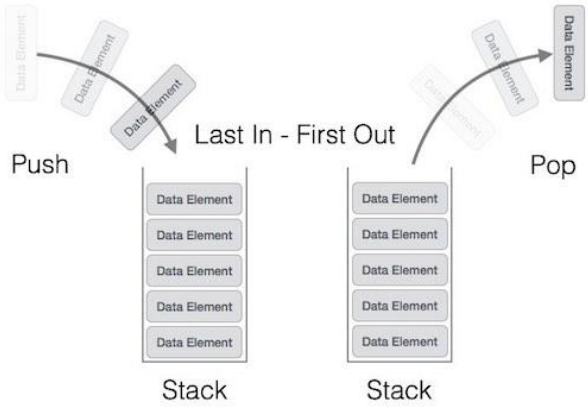
STACK

Definition

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

CONCEPT OF STACK REPRESENTATION

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

OPERATIONS OF STACK

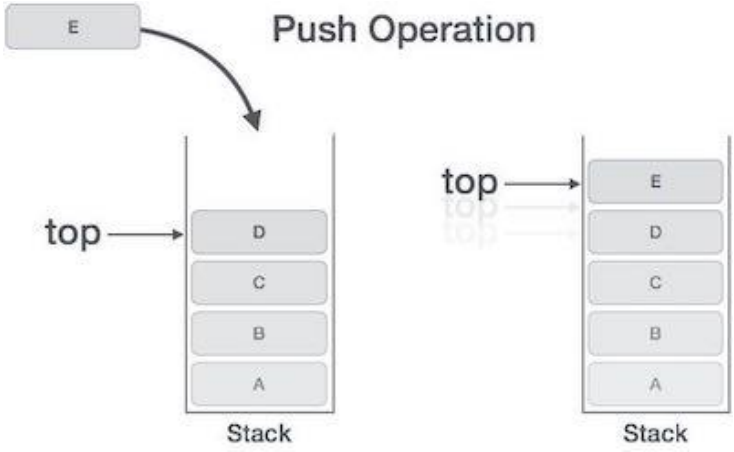
A stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.
- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data
   if stack is full
      return null
   endif
      top ← top + 1
   stack[top] ← data
end procedure
```

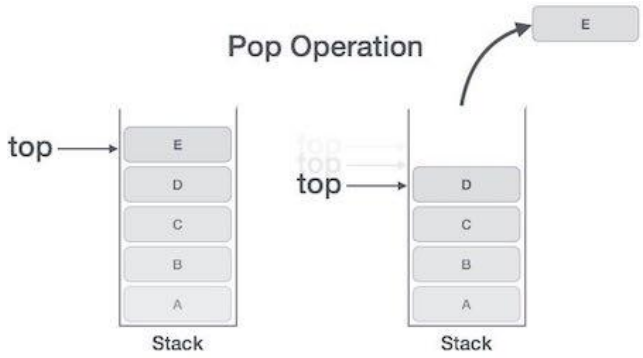
Example

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack
  if stack is empty
    return null
  endif
    data ← stack[top]
  top ← top - 1
  return data end procedure
```

peek( )

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull
   if top equals to MAXSIZE
      return true
   else
      return false
   endif
   end procedure
```

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty
   if top less than 1
      return true
   else
      return false
```

```
  endif
  end procedure
```

Example

```
bool isempty() {
  if(top == -1)
    return true;
  else
    return false;
}
```

INFIX, POSTFIX & PREFIX CONVERSIONS

The way to write arithmetic expression is known as a **notation**.

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

Example: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example: AB +, CD+, etc.

All these expressions are in postfix notation because the operator comes after the operands.

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

EXPRESSION EVALUATION

In expression evaluation problem, we have given a string s of length n representing an expression that may consist of integers, balanced parentheses, and binary operations (+, -, *, /). Evaluate the expression. An expression can be in any one of prefix, infix, or postfix notation.

Operand1 Operand2 Operator

Example

Using a Stack to Evaluate an Expression

We often deal with arithmetic expressions written in what is called infix notation:

Operand1 op Operand2

We have rules to indicate which operations take precedence over others, and we often use parentheses to override those rules.

It is also quite possible to write arithmetic expressions using postfix notation:

Operand1 Operand2 op

With postfix notation, it is possible to use a stack to find the overall value of an infix expression by first converting it to postfix notation.

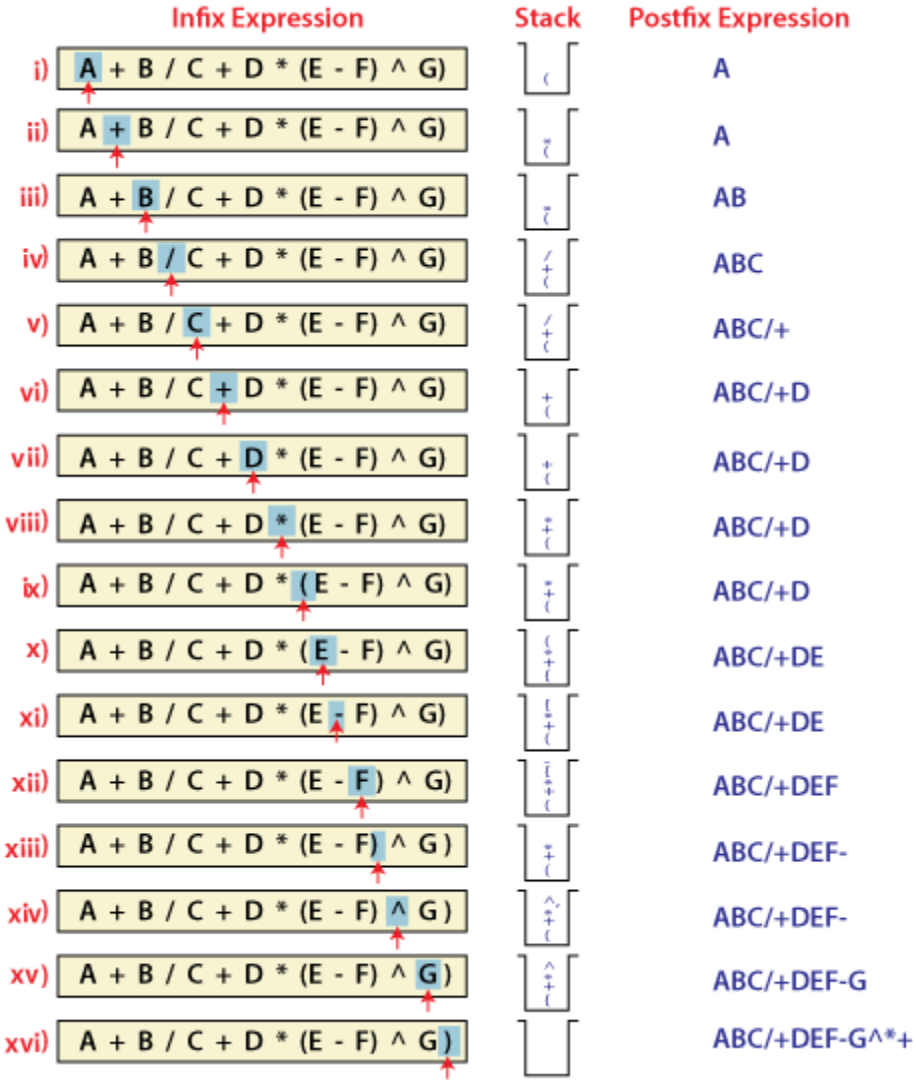
Example: Suppose we have this infix expression Q:

5 * (6 + 2) - 12 / 4

The equivalent postfix expression P is:

5 6 2 + * 12 4 / -

Converting an infix expression into a postfix expression.

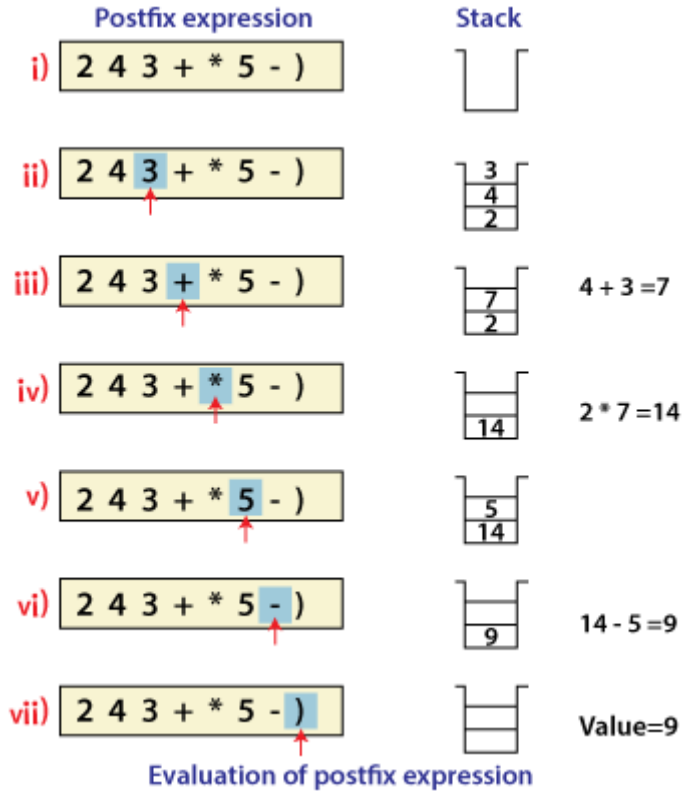


Example:

Now let us consider the following infix expression 2 * (4+3) - 5.

Its equivalent postfix expression is 2 4 3 + * 5.

The following step illustrates how this postfix expression is evaluated.



APPLICATIONS OF STACK

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Call

1. Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: A + (B - C)

2. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking.

Valid Delimiter	Invalid Delimiter
While (i > 0)	While (i >
/* Data Structure */	/* Data Structure
{ (a + b) - c }	{ (a + b) - c

4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

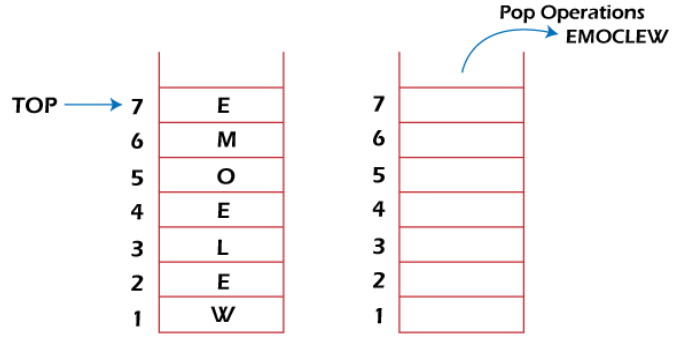
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one.

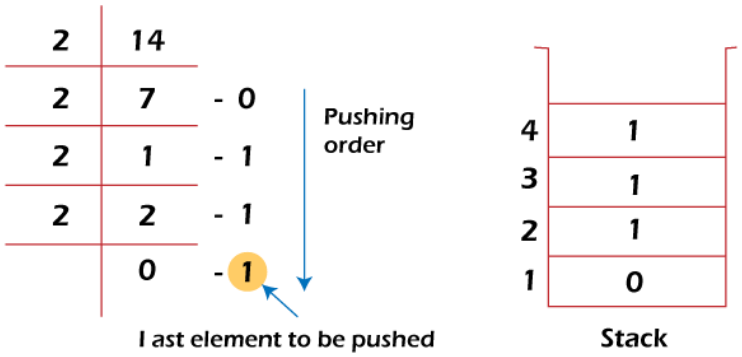


Converting Decimal to Binary:

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be

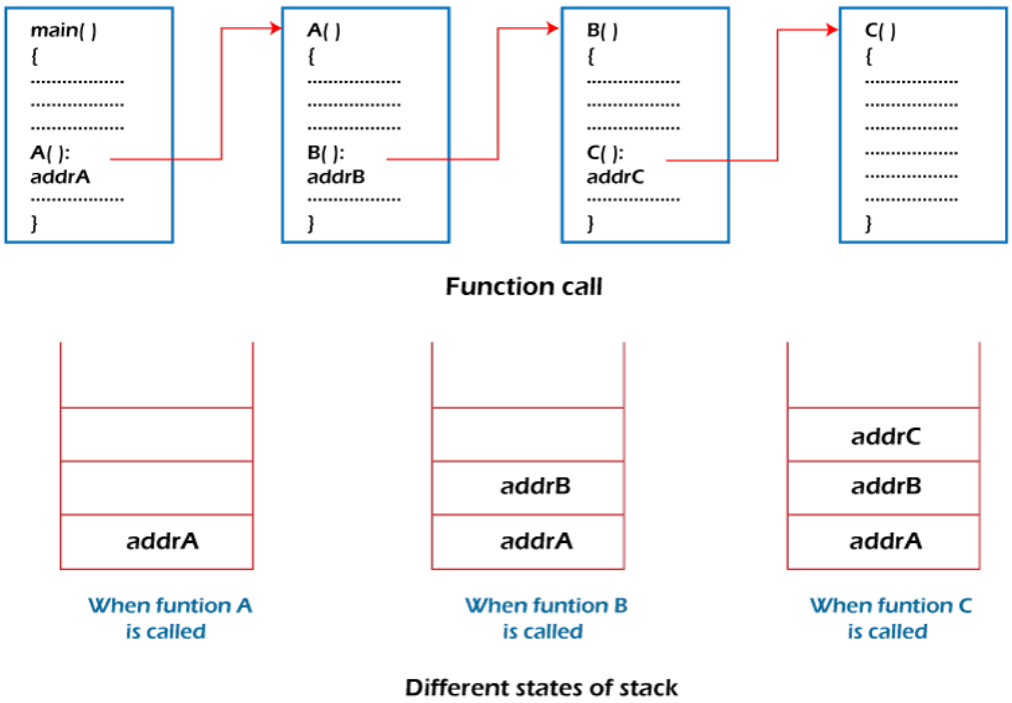
used to convert a number from decimal to binary/octal/hexadecimal form. For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

Example: Converting 14 number Decimal to Binary:



5. Processing Function Calls:

Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



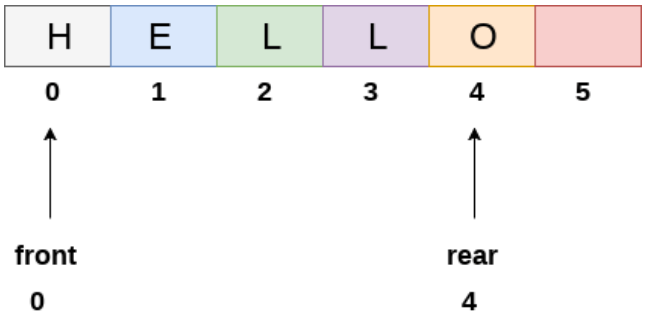
QUEUE

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

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.

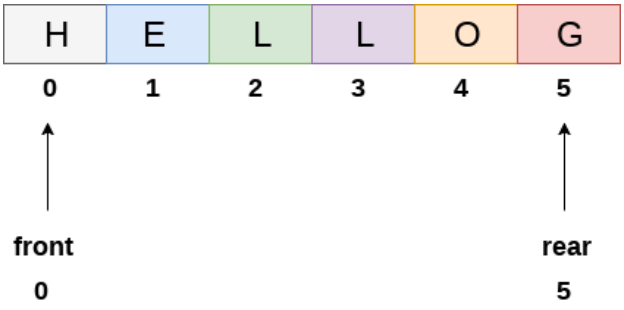
Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue

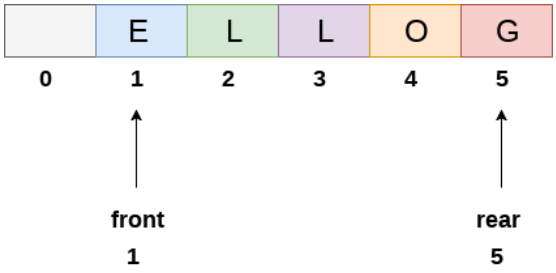
The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However,
the value of rear increases by one every time an insertion is performed in the queue.

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

- **Step 1:** IF REAR = MAX - 1

  Write OVERFLOW

  Go to step

  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1

  SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

- **Step 3:** Set QUEUE[REAR] = NUM
- **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

- **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

APPLICATIONS OF QUEUE

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.