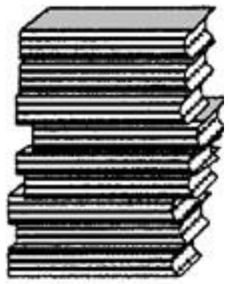
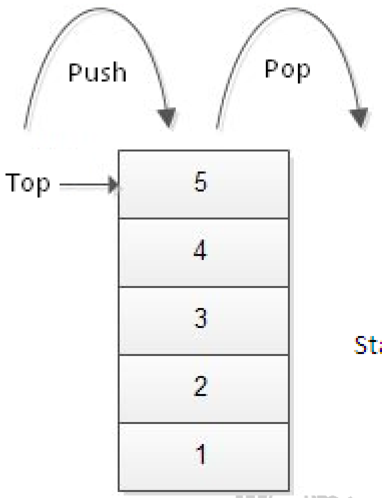
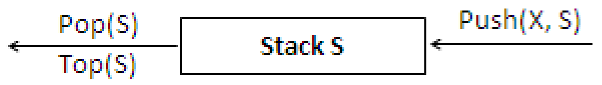
STACK

Definition

Stack is a linear list in which elements are added and removed from **only one end**, called the **top**. It is a "last in, first out" (**LIFO**) data structure. At the logical level, a stack is an **ordered** group of **homogeneous** items or elements. Because items are added and removed only from the top of the stack, the last element to be added is the first to be removed. Stacks are also referred as "piles" and "push-down lists".


Stack of Books

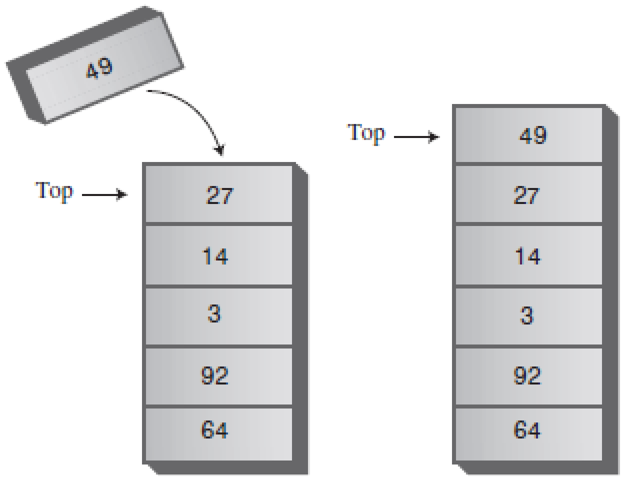




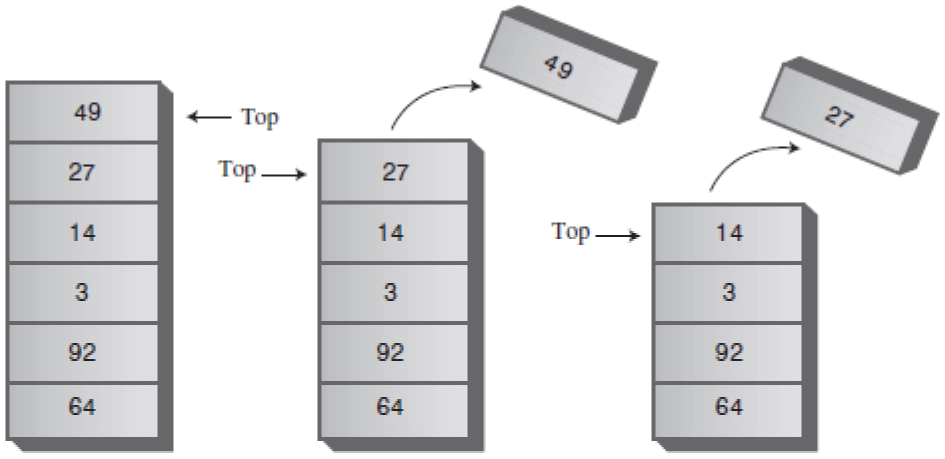
Stack model : Input bto a Stack is by push, output is by pop

- **Operations on stacks**
  - **Push** - Inserts new item to the top of the stack. After the push, the new item becomes the top.
  - **Pop** - Deletes top item from the stack. The next older item in the stack becomes the top.
  - **Top** - Returns a copy of the top item on the stack, but does not delete it.
  - **MakeEmpty** - Sets stack to an empty state.
  - Boolean **IsEmpty** - Determines whether the stack is empty. IsEmpty should compare top with **-1**.
  - Boolean **IsFull** - Determines whether the stack is full. IsFull should compare top with **MAX_ITEMS - 1**.

- **Conditions**
  - **Stack overflow -** The condition resulting from trying to push an element onto a full stack.
  - **Stack underflow -** The condition resulting from trying to pop an element from an empty stack.



New item pushed on Stack



Two items popped from Stack

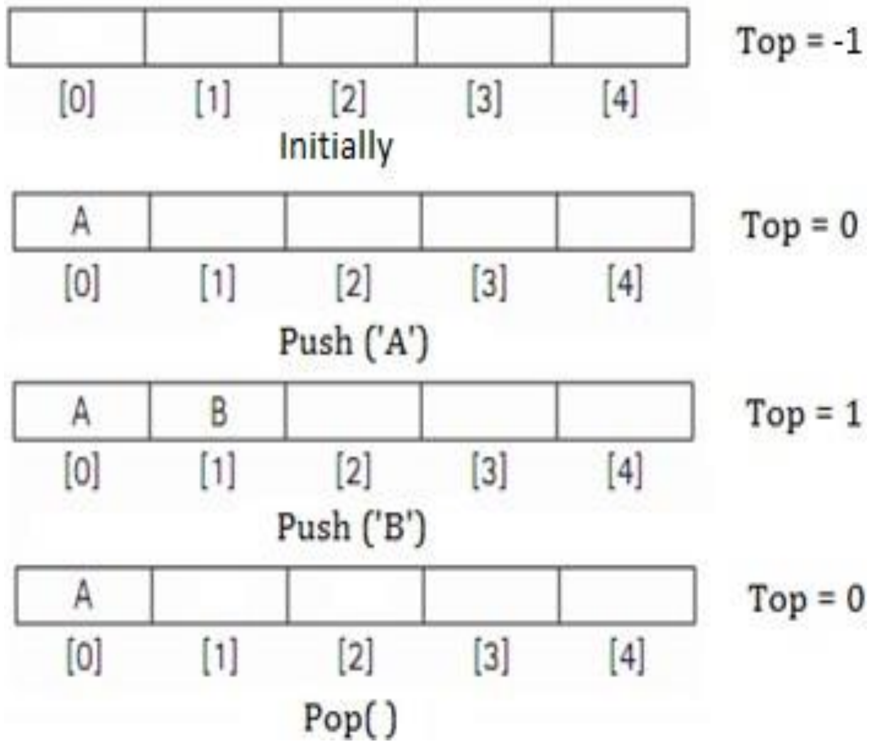
APPLICATIONS OF STACKS

- **Recursion -** Example, Factorial, Tower of Hanoi.
- **Balancing Symbols**, i.e., finding the unmatched/missing parenthesis. For example, ((A+B)/C and (A+B)/C). Compilers often use stacks **to perform syntax analysis of language statements**.
- C**onversion** of infix expression to postfix expression and decimal number to binary number.
- **Evaluation** of postfix expression.
- **Backtracking**- For example, 8-Queens problem.
- **Function calls -** When a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Similarly the current location in the routine must be saved so that the new function knows where to go after it is done. For example, the main program calls operation A, which in turn calls operation B, which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A; and so on. The call-and-return sequence is essentially a LIFO sequence, so a stack is the perfect structure for tracking it.

- **Implementations of stack**
  1. Array implementation of stack
  2. Linked list implementation of stack

- **Array implementation of stack**

Stack can be represented using one dimensional array and it is probably the **more popular** solution.Here the stack is of fixed size. That is maximum limit for storing elements is specified. Once the maximum limit is reached, it is not possible to store the elements into it. So array implementation is not flexible and not an efficient method when resource optimization is concerned.


Top = -1
[0] [1] [2] [3] [4]
Initially
A
Top = 0
[0] [1] [2] [3] [4]
Push ('A')
A B
Top = 1
[0] [1] [2] [3] [4]
Push ('B')
A
Top = 0
[0] [1] [2] [3] [4]
Pop( )

PUSH AND POP OPERATION

```
#include<stdio.h>
#include<conio.h
>#define MAX 5

void push();
void pop();
void display();
int stack[MAX], top=-1, item;
void push()
{
        if(top == MAX-1)
                printf("Stack is full");
        else
        {
                printf("Enter item:
                ");
                scanf("%d",&item);
                top++;
                stack[top] = item;
                printf("Item pushed = %d", item);
        }
}

void pop()
{
        if(top == -1)
        printf("Stack is
        empty");else
        {
                item = stack[top];
                top--;
                printf("Item popped = %d", item);
        }
}

void display()
{
        int i;
        if(top == -1)
                printf("Stack is empty");
        else
        {
                for(i=top; i>=0; i--)
                        printf("\n %d", stack[i]);
        }
}
```

ARRAY IMPLEMENTATION STACK

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.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

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
```

Implementation of this algorithm in C, is very easy. See the following code −

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. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

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
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

  if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

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 −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in-**between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation

1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operation
Step 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumed
Step 6 − pop the stack and perform operation