UNIT-2 STACK

1.1 Introduction to Stack

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. The order may be **LIFO(Last In First Out) or FILO(First In Last Out)**. LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.The deletion and insertion in a stack is done from top of the stack.

For e.g, plates stacked over one another in the canteen

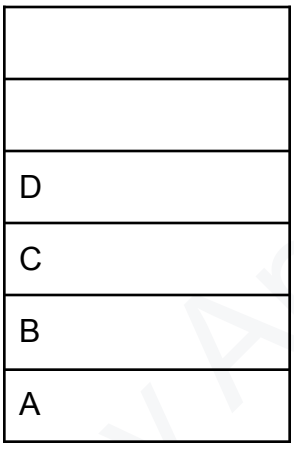
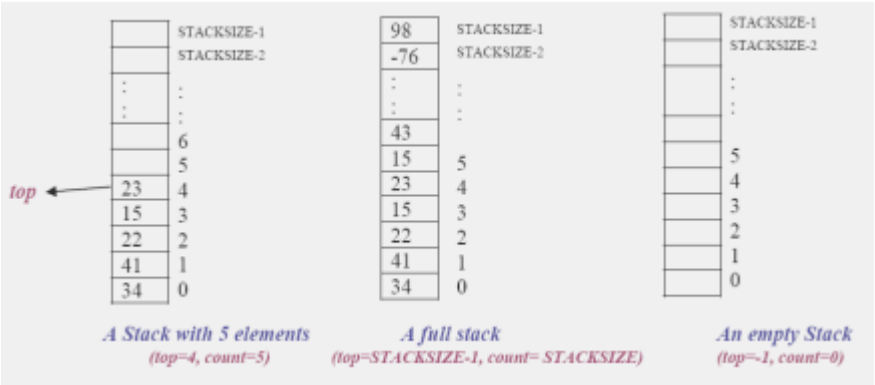


Fig 1. A stack containing elements or items.

In computer science we commonly place numbers on a stack, or perhaps place records on the stack



1.2 Applications of Stack:

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser

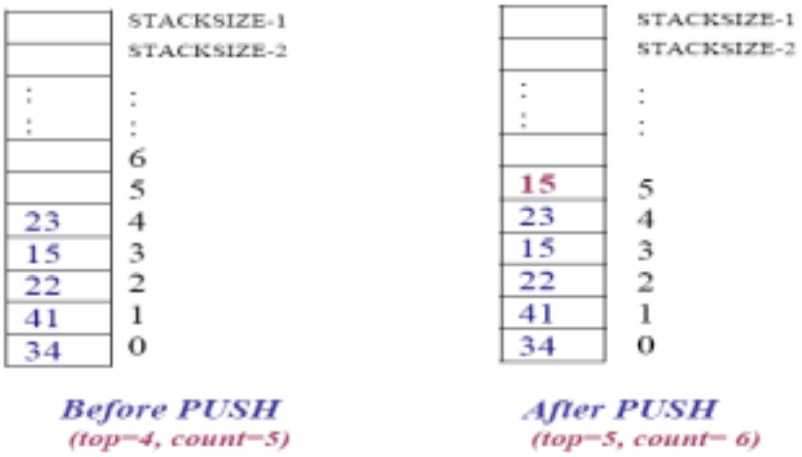
- To perform the undo sequence in a text editor
- Used in recursion
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures.

1.3. Stack Operations:

The following operations can be performed on a stack:

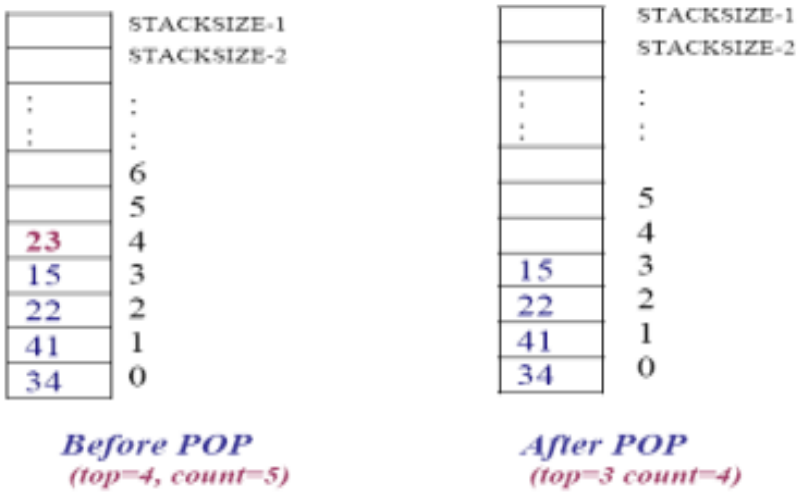
PUSH operation: The push operation is used to add (or push or insert) elements in a stack

- When we add an item to a stack, we say that we push it onto the stack
- The last item put into the stack is at the top.



POP Operation: The pop operation is used to remove or delete the top element from the stack.

- When we remove an element, we say that we pop it from the stack.
- When an item popped, it is always the top item which is removed.



The PUSH and POP operations are the basic or primitive operations on a stack. Some other operations are:

- **createEmptyStack** operation: This operation is used to create an empty stack.
- **isFull** operation: This isFull operation is used to check whether the stack is full or not (i.e, stack overflow)
- **isEmpty** operation: This isEmpty operation is used to check whether the stack is empty or not. (i.e, stack underflow)
- **Top** operation: This operation returns the current item at the top of stack, it doesn't remove it.

1.4 Stack ADT(Abstract Data Type):

A stack **S** of elements of type **T** is a finite sequence of elements of T together with the operations

- **CreateEmptyStack(S)**: Create or make stack S be an empty stack
- **Push(S, x)**: Insert x at one end of the stack, called its top
- **Top(S)**: If stack S is not empty; then retrieve the element at its top
- **Pop(S)**: If stack S is not empty; then delete the element at its top
- **IsFull(S)**: Determine if S is full or not. Return true if S is full stack; return false otherwise
- **IsEmpty(S)**: Determine if S is empty or not. Return true if S is an empty stack; return false otherwise.

1.5 Implementation of Stack:

Stack can be implemented in two ways:

1. Array implementation of stack(or static implementation)
2. Linked list implementation of stack(or dynamic)

Array implementation of a stack:

It is one of the ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value(an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of the current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1 (top=-1).

```
#define MAX 10
struct stack{
    int item[MAX];          //declaring an array to store item
    int top;                //Top of a stack
};
```

typedef struct *stack st;*

➢ Creating Empty Stack:

The value of top=-1 indicates the empty stack in C implementation.

```
/* function to create an empty stack */
    void createEmptyStack(st *s){
        s->top=-1;  //pointer s is reference to the top which is assigned by value -1
    }
```

➢ Stack Empty or Underflow

This is the situation when the stack contains no element. At this point the top of the stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function returns 1 if the stack is empty, 0 otherwise.

```
int isEmpty(st *s){
     if(s->top==-1)
          return 1;
     else
          return  0;
}
```

➢ **Stack Full or overflow:**

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location(MAXSIZE-1) of the stack. The following function returns true(1) if stack is full false(0) otherwise.

```
int isFull(st *s){
     if(s->top==MAX-1)
          return 1;
     else
          return 0;
}
```

PUSH and POP operations on Stack

Let stack[MAXSIZE] be an array to implement the stack. The variable top denotes the top of the stack.

i) Algorithm for PUSH (inserting an item into the stack) operation:

This algorithms adds or inserts an item at the top of the stack

```
1.[CHECK FOR STACK OVERFLOW ?]
  If top = MAXSIZE-1 then
       Print "Stack overflow" and Exit
```

```
    Else
        Set top = top+1 [increase top by 1]
        Set stack[top] = item (inserts item in new top
    position)
  2. Exit
```

ii) Algorithm for POP (removing an item from the stack) operation:

This algorithm deleted the top element of the stack and assign it to a variable item

```
  1. [CHECK FOR STACK UNDERFLOW?]
    If top<0 then
        Print "stack underflow" and exit
    else
        [Remove the top element]
        Set item = stack[top]
        Decrement top by 1
        Set top = top-1
        Return the deleted item from the stack
  2. Exit.
```

The PUSH and POP functions

The C function for **PUSH** operation

```
void push(st *s, int element)
{
    if(isFull(s))
        printf("\n The stack is overflow \n");
    else
        s->items[++(s->top)]=element;
}
```

The C function for **POP** operation

```
void pop(stack *s){
    if(isEmpty(s))
        printf("\n Stack Underflow: Empty Stack ");
    else
        printf("\n The deleted item is %d:\t", s->item[s->top--]);
}
```

1.6 Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix
- Prefix
- Postfix

- **Infix Expression:** It is an ordinary mathematical notation of expression where the operator is written in between the operands. For e.g, **A+B** Here "+" is an **operator** and A , B are called **operands**.
- **Prefix Expression**: In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notations. Example: **+AB**
- **Postfix Expression**: in this expression the operators are written after the operands so it is called the postfix notation(post mean after). In this notation the operator follows the two operands. For **AB+**
  Example:
  A+b (infix)
  +AB(prefix)
  AB+ (postfix)
  Prefix and postfix are parenthesis free expressions. For e.g,
  (A+B)*C Infix form
  *+ABC Prefix form
  AB+C* Postfix form

1.7 Converting an infix expression to postfix

First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.

- Remember, to convert an infix expression to its postfix expression , we first convert the innermost parentheses to postfix, resulting as a new operand.
- In this fashion parentheses can be successively eliminated until the entire expression is converted.
- The last pair of parentheses to be opened within a group of parentheses encloses the first expression within the group to be transformed.
- This last in, fist out behavior suggests the use of a stack.

Precedence Rule:

While converting infix to prefix you have to consider the precedence rule, and the precedence rules are as follows

1. Exponentiation(the expression A$B is A raised to the B power, so that 4$2=16)
2. Multiplication/Division
3. Addition/subtraction

When un-parenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be formed right to left.

A+B+C means (A+B)+C

ABC means A$(B$C)

By using parentheses we can override the default precedence.

Consider an example that illustrate the converting of infix to postfix expression,

A+(B*C). Use the following rule to convert it in prefix:

1. Parentheses for emphasis
2. Convert the multiplication
3. Convert the addition
4. Postfix form

Illustration:

A+(B*C)	Infix form
A+(B*C)	parentheses for emphasis
A+(BC*)	convert the multiplication
A(BC*)+	Convert the addition
ABC*+	Postfix form

Consider an example:

(A + B) * ((C - D) + E) / F Infix form

(AB+) * ((C – D) + E) / F

(AB+) * ((CD-) + E) / F

(AB+) * (CD-E+) / F

(AB+CD-E+*) / F

AB+CD-E+*F/ Postfix form

1.7.1 Algorithm to convert infix to postfix notation

Let here two stacks **opstack** and **poststack** are used and **otop** & **ptop** represents the opstack top and poststack top respectively.

1. Scan one character at a time of an infix expression from left to right

2. opstack=the empty stack

3. Repeat till there is data in infix expression

3.1 if scanned character is '(' then push it to opstack

3.2 if scanned character is operand then push it to poststack

3.3 if scanned character is operator then

```
if(opstack!=-1)
```

```
            while(precedence
            (opstack[otos])>precedence(scan
            character)) then
            pop and push it into poststack
        Otherwise
            push into opstack
    3.4 if scanned character is ')' then
        pop and push into poststack until '(' is not
    found and ignore both symbols
4. pop and push into poststack until opstack is not
empty.
5. return
```

1.8 Conversion of infix expression to prefix expression

The precedence rule for converting an expression from infix to prefix is identical to that of infix to postfix. Only changes from postfix conversion are that the operator is placed before the operands rather than after them. The prefix of A+B-C is -+ABC.

For e.g, consider an infix expression:

A $ B * C - D + E / F / (G + H) infix form

= A $ B * C - D + E / F / (+GH)

= $AB * C - D + E / F / (+GH)

= *$ABC - D + (E/F)/ (+GH)

= *$ABC-D + //(EF+GH)

= *$ABC-D + //EF+GH

= (-*$ABCD) + (//EF+GH)

= **+-*$ABCD//EF+GH** which is in **prefix** form.

1.8.1 Algorithm to convert infix to prefix expression by using stack.

Let the two stacks **opstack** and **prestack** are used and **otop** and **ptop** represents the opstack top and prestack top respectively.

1. *Start*
2. *Scan one character at a time of an infix expression from right to left*
3. *Opstack = the empty stack*
4. *Repeat til there is data in infix expression*

4.1 if scanned character is ')' then push it to opstack

4.2 if scanned character is operand then push it to prestack

4.3 if scanned character is operator then,

if (opstack != -1)

while (precedence(opstack[otop])> precedence(scan character)) then

Pop and push it into prestack

Otherwise

Push scanned character into opstack

4.4 if scanned character is '(' then

Pop and push into prestack until ')' is not found and ignore both symbols

5. *Pop and push into prestack until opstack is not empty*
6. *Pop and display prestack which gives the required prefix expression.*

1.9 Evaluation of Infix, Postfix and Prefix Expressions.

1.9.1. Evaluation of Infix expression: For evaluation of infix expressions, we use two stacks. One stack is operand stack which is used to keep operands and another is

operator stack which is used to keep operators (+,-,*,/ and ^). Let us consider a "process" means,

a. Pop operand stack once (value1)
b. Pop operand stack once (operator)
c. Pop operand stack again (value2)
d. Compute value2 operator value1
e. Push the value obtained in operand stack

Algorithms:

1. Given an infix expression, scan one character from the expression at a time from left to right. Until the end of the expression is reached, perform only one of the following steps (a) through (f):

a. If the character is an operand, push it onto the operand stack.

b. If the character is an operator, and the operator stack is empty then push it onto the operator stack

c. if the character is an operator and the operator stack is not empty, and the character precedence is greater than the precedence of the stack top of the operator stack, then push the character onto the operator stack.

d. If the character is "(", then push it onto the operator stack.

e. If the character is ")", then "process" as explained above until the corresponding "(n is encountered in the operator stack.

At this stage POP the operator stack and ignore "("

f. If cases (a), (b), (c), (d) and (e) do not apply, then process as explained above.

2. *When there are no more input characters, keep processing until the operator stack becomes empty.*
3. *The values left in the operand stack is the final result of the expression.*

Example:

Infix expression: **3+4*5*(4+3)-1 / 2 +1**

S.N	Character Scanned	Operand stack	Operator stack	Operation performed
	3	3	-	
	+	3	+	
	4	3, 4	+	
	*	3, 4	*	
	5	3, 4, 5	+	
	*	3, 20	+ *	Pop 5 , pop * pop 4 performed 4 *5 and push result (20) into the operand stack.
	(	3 20	+ * (	
	4	3 20 4	+ * (	
	+	3 20 4	+ * (+	
	3	3 20 4 3	+ * (+	
	)	3 20 7	+ *	Pop 3, pop + pop 4 performed 4+3=7 and push result into stack , ignore).

	-	3, 140	+	Pop 7 and pop 20 and pop * performed 20 * 7 and push result into stack
		143	-	Pop 140, pop 3 and pop + performed 140+3 and push result into stack
	1	143 1	-	
	/	143 1	- /	
	2	143 1 2	- /	
	+	143 0.5	-	Pop 1, pop 2 and pop / performed ½ and push result into stack
		142.5	+	Pop 0.5 and pop 143 and pop - performed 143-0.5 push result into stack
	1	142.5 1	+	
		143.5		Pop 1 pop + pop 143.5 compute 142.5+1 , push 143.5 into operand stack

The final value at operand stack is 143.5 . result is 143.5

1.9.2 Evaluation of postfix expressions

The postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parentheses are not required in postfix

Algorithm:

1. ***Create an empty stack called operandStack.***
2. ***Scan the string from left to right.***
    - ***If the token is an operand, convert it from a string to an integer and push the value onto the operand stack.***
    - ***If the token is an operator, *, /, +, or -, it will need two operands. Pop the operandStack twice. The first pop is the second operand and the second pop is***

the first operand. Perform the arithmetic operation. Push the result back on the operand stack.

3. *When the input expression has been completely processed, the result is on the stack. Pop the operand stack and return the value.*

Example:

Postfix expression: **231*+23*+9-1+**

S.N	Character scanned	Operand stack	Operator stack	Operation Performed
	2	2	-	
	3	2 3	-	
	1	2 3 1	-	
	*	2 3	*	Pop 1 and pop 3 performed 1 *3 push the result into the stack.
	+	5	+	Pop 3 and pop 2 performed 2+ 3 push result into stack
	2	5 2		
	3	5 2 3		
	*	5 6	*	Pop 3 and pop 2 performed 3 *2 push result 6 into stack
	+	11	+	Pop 6, pop 5 and performed 6 +5 push result 11 into stack
	9	11 9		
	-	-2		Pop 9 , pop 11 and performed 9-11 push result -2 into stack
	1	-2 1		
	+	-1		Pop -2, pop 1 and performed -2+1 push result -1 into stack

The Final result is **-1**

1.9.3 Evaluation of Prefix Expressions

Prefix and postfix expressions can be evaluated faster than an infix expression, this is because we don't need to process any brackets or follow operator precedence rules. In postfix and prefix expressions whichever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions.

Algorithms

1. Scan the prefix expression from right to left(that is from the end of the expression)
2. If character is an operand, push it to stack
3. If the character is an operator, pop two elements from the stack. Operand that is popped at first is the first operand(say operand1) and the operator that is popped at second is the second operator(say operand 2). Compute operand1 operator operand2, and push the result back to the stack
4. Repeat the above steps until there are no characters left to be scanned in the expression.
5. The result is stored at the top of the stack, return it

Example-1

Prefix expression: **+9*28**

In prefix the expression should evaluate from **right** to **left**

Character Scanned	**Operand Stack**	**Processing**
8	8	Push into stack (8)
2	8 2	Push into stack (2)
*	16	Pop 2 and 8. Multiply 2 and 8 push result into the stack
9	16 9	Push into stack (9)
+	25	Pop 9 and 16 add them and push the result into the stack.