Unit-2
STACK

⊛ Concept & Example of Stack:

→ A stack is a linear data structure in which an element may be inserted or deleted only at one end. i.e, The elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

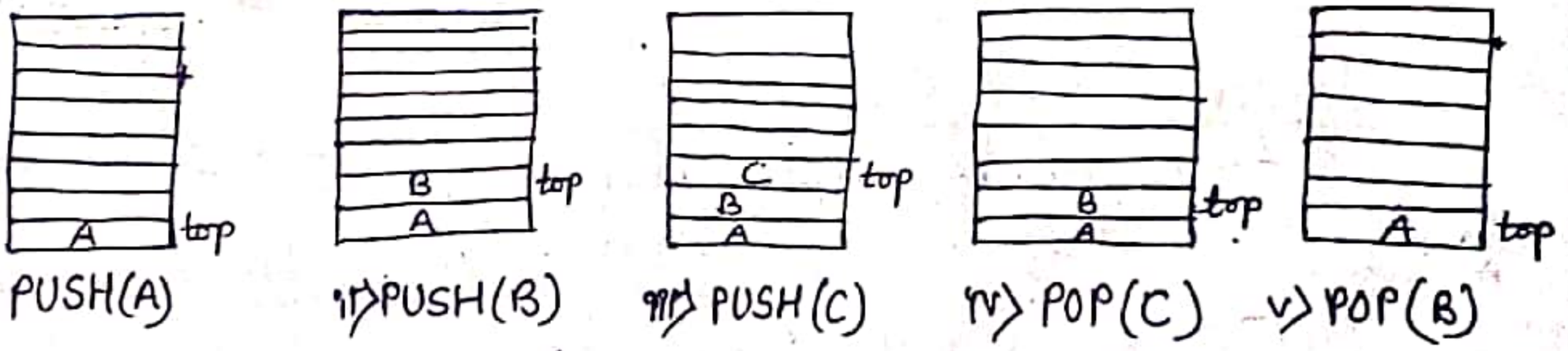
→ Stack uses a variable called top which points topmost element in the stack.

→ Top is incremented while pushing (inserting) an element in to the stack and decremented while popping (deleting) an element from the stack.

→ A stack follows the principle of last-in-first-out (LIFO) system.

→ PUSH and POP are two terms used for insertion and deletion operations in stack respectively.

Example:



i) PUSH(A) ii) PUSH(B) iii) PUSH(C) iv) POP(C) v) POP(B)

fig. push, pop operations in stack

Applications of stack:

→ To evaluate the postfix and prefix expressions.
→ To keep the page-visited history in web browser.
→ To perform the undo sequence in a text editor.
→ Used in recursion.
→ To check the correctness of parentheses sequence.
→ To pass the parameters between the functions in C program.

⊛ Stack ADT:

In Stack ADT (Abstract Data Type) we discuss about the operations that can be performed on stack. Following are such operations:

i) Create Empty Stack(S) → Create stack S which is initially an empty stack.

ii) Push (S, x) → Insert x at one end of stack, called its top.

iii) Top(S) → If stack S is not empty; then retrive the element at its top.

iv) Pop(S) → If stack S is not empty; then delete the element at its top.

v) IsFull(S) → Determine whether the stack S is full or not. Return true if S is full; return false otherwise.

vi) IsEmpty(S) → Determine whether the stack S is empty or not. Return true if S is an empty; return false otherwise.

⊛ Implementation of Stack:

Stack can be implemented in following two ways:-

1) Array implementation of stack (or static implementation):

This implementation method uses one dimensional array to store the data. In this implementation top is an integer value 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 current top of the stack.

Structure for stack:

```
#define MAX 10
struct stack {
    int items[MAX]; //Declaring array to store items
    int top;
};
typedef struct stack st;
```

keyword | struct name | object of type struct

Some Operations for static implementation of stacks

i) **Creating Empty stack →**

The value of top = −1 indicates the empty stack. (In C implementation)

```
void create_empty_stack(struct stack e)  //function to create an empty stack.
{
    e.top=-1;
}
```

ii) **Stack Empty or Underflow →** This is the situation when the stack contains no element. At this point top (i.e, variable that holds address of last inserted element) is present at the bottom of the stack. Top = −1 indicates the stack is empty.

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

```
int IsEmpty() {
    if(top==-1)
        return 1;
    else
        return 0;
}
```

iii) **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 of the stack.

The following function will return true (i.e 1) if the stack is full, false (i.e 0) otherwise.

```
int IsFull() {
    if(top==MAXSIZE-1)
        return 1;
    else
        return 0;
}
```

#Algorithms for PUSH and POP operations (For Static implementation):

Let Stack[MAXSIZE] is an array to implement the stack and top variable denotes the top of the stack.

Algorithm for PUSH operation (adds or inserts an item on top of stack)

Overflow भए element add गर्ने अघिनै यसैले overflow condition check गरेको

Step1: Check for stack overflow as
If top == MAXSIZE-1 then
print "Stack Overflow" and Exit the program.
else, increase top by 1 as
Set, top = top+1

Step2: Read element to be inserted (say element)

Step3: Set, stack[top] = element //Inserts item on new top position.

Step4: Stop

Algorithm for POP operation (deletes top element of stack):

element नभएको condition मा element नै नभए के delete गर्ने यसैले Underflow condition check गरेको

Step1: Check for the stack Underflow as
If top < 0 then
Print "Stack Underflow" and Exit the program.
else,
Remove the top element and set this element to the variable as;
Set element = Stack[top]
Now decrement top by 1 as;
Set top = top-1

Step2: Print 'element' as a deleted item from the stack.

Step3: Stop.

2) Linked List implementation of stack (or dynamic implementation):

The limitation, in case of an array (or static implementation) is that we need to define the size at the beginning of the implementation. This makes stack static (i.e, of fixed size). So, It may result in "stack overflow" if we try to add elements after array is full. So, to eliminate this limitation we use this implementation method, so that stack can grow in real time.

Structure:

```
struct linked_stack {
    int info;
    linked_stack *next;
};
```

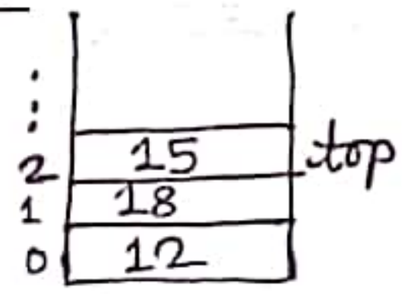
```
typedef struct linked_stack NodeType;
linked_stack *top;
top=NULL;
```

⊛ Stack Operations:

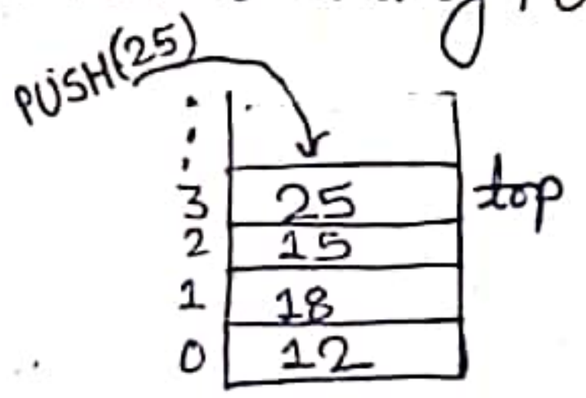
The PUSH and POP operations are performed on a stack which are described as follows:-

1) **PUSH operation** → The push operation is used to add or insert elements in the stack.
   → When we add an item to stack, we say that we push it into the stack.
   → The last item put into the stack is at top.
   → Top is incremented when PUSH operation occurs.
   → PUSH operation increases size of stack.
   → We check Overflow condition during PUSH operation.

Example:



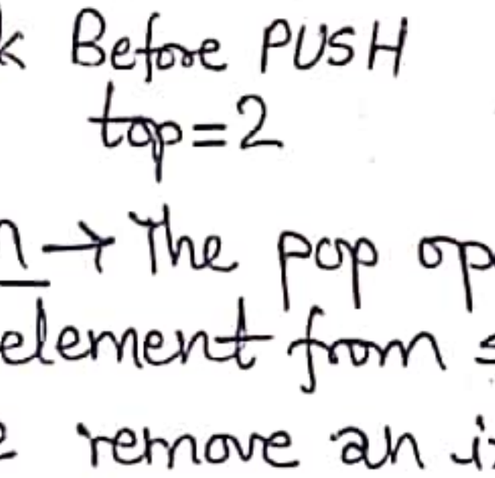
Stack Before PUSH
top=2



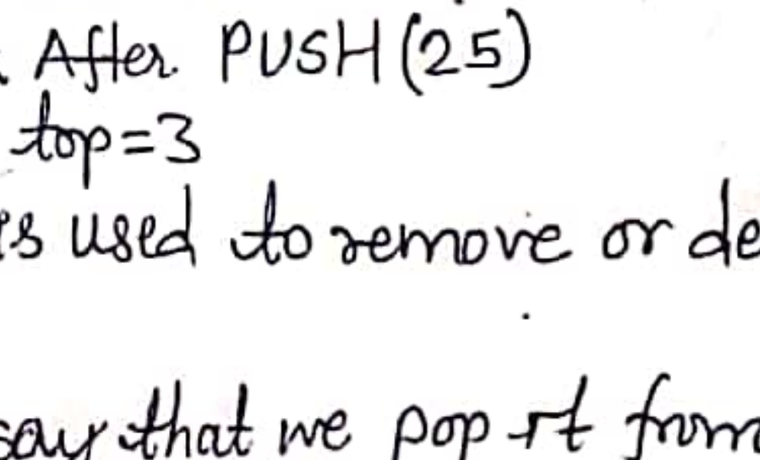
Stack After PUSH(25)
top=3

2) **POP operation** → The pop operation is used to remove or delete the top element from stack.
   → When we remove an item, we say that we pop it from stack.
   → When an item is poped, it is always the top item which is removed
   → Top is decremented when POP operation occurs.
   → POP operation decreases size of stack.
   → We check Under-flow condition during POP operation.

Example.



Stack Before POP
top=2.



Stack after POP(15)
top=1

⊛ Bracket Matching (OR Delimiter Matching) Application of stack:

One application of the stack is in matching delimeters in a program. No program is considered correct if the delimiters are mismatched. In C program we have following delimeters:

- Paranthesis "(" and ")".
- Square brackets "[" and "]".
- Curly brackets "{" and "}".
- Comment delimeters "/*" and "*/"

Example: The opening of each delimeters (, {, [, /* should be matched by a closing or righ delimeters), },], */ respectively.

Infix, Prefix and Postfix Notation:

One of the application of the stack is to evaluate the expression. We can represent the expression in following three types of notations:

- Infix
- Prefix
- Postfix

Infix expression → If the operator is placed between its two operands then it is called infix notation.
for e.g. A+B, C/D, C-D, A*E etc.
where, +, -, /, * are operators
& A, B, C, D, E on which operation is done are operands.

Prefix expression / Prefix notation → If the operator symbol is placed before its two operands then it is known as prefix expression
for eg. +AB, *EF etc.

Postfix expression → If the operator symbol is placed after its two operands then it is known as postfix expression.
for eg. AB+, CD- etc.

Note: Both prefix and postfix are parenthesis free expressions.

For example → (A+B)*C [Infix form]
*+ABC [Prefix form]
AB+C* [Postfix form]

Similarly following are some examples for understanding:

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

⊛ Conversion of one type of expression to another:

(Lesser imp concept of precedence rule understand only)

1) Converting an Infix expression to Postfix:

Precedence Rule → Before any type of conversion first we should know three levels of precedence as follows:-

i) Exponent (Symbol ↑ or ^ i.e, Power operator must be performed first).

($ this sign is also exponent i.e, power)

ii) Multiplication and Division will be second priority (i.e, * or /).

iii) Addition and Subtraction will be last priority. (i.e, + or −)

Note:- Braces (i.e, ( )) will be in first priority, to solve then exponent and others. i.e, +, −, *, / will have priority according to our normal mathematics simplification priority. i.e, / first, then * and + and − finally.

Example 1:- Convert Infix form (A+B)*((C-D)+E)/F to Postfix form.

Soln.

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

= (AB+)*((C-D)+E)/F //Computer performs only one operation at a time so we will do only one at a time.

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

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

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

= AB+CD-E+*F/ [Postfix form]

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

i) First we convert the innermost parenthesis to postfix, resulting as a new operand.

i) Parenthesis should be successively eliminated until the entire expression is converted.

ii) The last pair of parenthesis to be opened within a group of parenthesis, encloses the first expression within the group to be transformed.

iv) This last in, first-out behaviour suggests to use of stack.

Rule

i) Parenthesis for emphasis
ii) Convert the multiplication
iii) Convert the addition
iv) Post-fix form.

Let eg. A+(B*C)

A+(B*C)

A+(BC*)

A(BC*)+

ABC*+

Now, We solve example 1 by different method: (Kaiser Method but somewhat lengthy)

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

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

= P*((C-D)+E)/F (let AB+=P)

= P*((CD-)+E)/F

= P*(Q+E)/F (Let CD-=Q)

= P*(QE+)/F

= P*H/F (Let QE+=H)

= P*HF/

= P*I (let HF/=I)

= PI*

Now substituting each value as in rough we get postfix as;

= AB+CD-E+*F/. [Postfix form].

Rough

PI*

= P*HF/ (substituting I=HF/)

= PQE+*F/ (Putting H=QE

= PCD-E+*F/ (Q=CD-

= AB+CD-E+*F/.

2) Converting an Infix expression to Prefix expression:

The precedence rule and process for converting from an infix expression to postfix are same as we wrote before. Only change is that the operator is placed before the operands rather than after them.

Method to convert

There are several methods one of them easy method is as;

i) First reverse the given expression.
ii) Then convert it into postfix as we did before.
iii) Again, finally reverse the expression to get prefix form.

Example 1;-

$x\hat{}y/(5*z)+2$ [Infix form]

step1: reverse

$2+(z*5)/y\hat{}x$

Step2: convert into postfix

$2+(z*5)/y\hat{}x$
$= 2+(z5*)/y\hat{}x$
$= 2+z5*/yx\hat{}$
$= 2+z5*yx\hat{}/$
$= 2z5*yx\hat{}/+$

Step3: again reverse to get Prefix form

$= +/\hat{}xy*5z2$. [required Prefix form]

⊛. Conversion from one type to another using Stack: (OR tracing algorithms)

1) Converting Infix to postfix using Stack:

We should remember following some important notes while converting infix to postfix using stack:-

i) We will always scan the given expression from left to right, one symbol at a time.
ii) Scanned braces, operands and operators and first placed in symbol column.

iii) Stack part in below table will only contain operators and postfix part will contain operands with poped out operators from stack.

iv) @ If higher priority (i.e, precedence) operator comes to stack than the operator that is stored just before in stack then, it can remain on stack.

ⓑ But if same or **lower** priority operator comes to stack then in this case the last added operator in stack is poped out to postfix. The precedence is as follows:-

1 → ^ or $
2 → /, *
3 → +, −

v) . Braces will not be compared as higher or lower precedence but opening of brace will be compared with closing of brace. While opened brace is closed in stack the braces are cancelled/eliminated by poping all the operators between brace to postfix.

Example:- Convert infix (A+B/C *(D+E)−F) to postfix using stack.

Soln

Symbol Scanned	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
−	(−	ABC/DE+*+
F	(−	ABC/DE+*+F
)		ABC/DE+*+F−

according to iv) @

/ poped out according to iv) ⓑ

according to v), (+*(+) =(+*

Since − is of lower precedence than * so * is poped out and again + & − are of same precedence so + is poped out.

Empty this part since opening brace is cancelled by closing brace poping out minus [i.e (−)]

required postfix form

2) Converting Infix to Prefix using Stack:

One of the easiest way to convert infix to prefix is as follows:-

i) First reverse the given expression.
ii) Then we convert it into postfix using stack as we did before.
iii) Again, we reverse the postfix expression to get prefix form.

Example:- Convert infix $(A+B/C*(D+E)-F)$ to prefix using stack.

Soln: Given, $(A+B/C*(D+E)-F)$

Step1: First we reverse given expression.

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

Step2: Now we convert this reversed expression to postfix.

Symbol Scanned	Stack	Postfix
(	(	
F	(	F
−	(−	F
(	(−(	F
E	(−(	FE
+	(−(+	FE
D	(−(+	FED
)	(−	FED+
*	(−*	FED+
C	(−*	FED+C
/	(−/	FED+C*
B	(−/	FED+C*B
+	(−/+	FED+C*B
A	(−/+	FED+C*BA
)		FED+C*BA−/+

Postfix = FED+C*BA−/+

Step3: Finally we reverse this postfix to get prefix form:-

+/−AB*C+DEF [required Prefix form]

⊛. To evaluate postfix / prefix expression using stack:

1) Evaluation of postfix expression [V.V.Imp] ✓

Algorithm to evaluate the postfix expression:

Let the stack be vstack (value stack).

Step1: Scan one character at a time from left to right of given postfix expression.

Step2: If scanned symbol is operand then read its corresponding value and push it into vstack.

If scanned symbol is operator then
- pop and place into op2.
- pop and place into op1.
- Compute result according to given operator and push result into vstack.

(i.e. vstack मा last value लाई op2 मा राख्ने; i.e. vstack मा second last value लाई op1 मा राख्ने)

(दिइएको operator को आधारमा operation गरि op2 र op1 मा लेखि vstack मा store गर्ने)

Step3: pop and display which is required value of given postfix expression.

Step4: return.

Trace of Evaluation: Consider an example: A,B,C,+,*,C,B,A,-,+,* ; 1,2,3,+,*,3,2,1,-,+,*

Scanned character	value	op2	op1	Result	vstack
A	1				1
B	2				1,2
C	3				1,2,3
+		3	2	5	1,5
*		5	1	5	5
3	3				5,3
2	2				5,3,2
1	1				5,3,2,1
-		1	2	1	5,3,1
+		1	3	4	5,4
*		4	5	20	20

∴ It's final value = 20.

2) Evaluation of prefix expression:-

The algorithm and evaluation procedure will be same as in postfix, the only difference is that first we reverse the given expression (i.e, instead of scanning from left to right we scan from right to left) and we convert in similar way that we did for postfix.

For e.g: ~~ABC + * CBA - + *~~
123 + * 321 - + *

First we reverse it as:

* + - ABC * + CBA

For e.g: +, −, *, 2, 2, /, 16, 8, 5

Now we reverse this expression since we can not perform operation if the operands are after operators. So, we reverse as follows;

5, 8, 16, /, 2, 2, *, −, +

Let: A; B; C, /, D, D, *, −, +.

We can also solve directly without considering characters like this

Now we proceed similarly as we did for postfix.

Scanned character	value	op2	op1	Result	vstack
A	5				5
B	8				5,8,
C	16				5,8,16
/		16	8	2	5,2
D	2				5,2,2
D	2				5,2,2,2
*		2	2	4	5,2,4
−		4	2	2	5,2
+		2	5	7	7

∴ Its final value = 7.