Stack

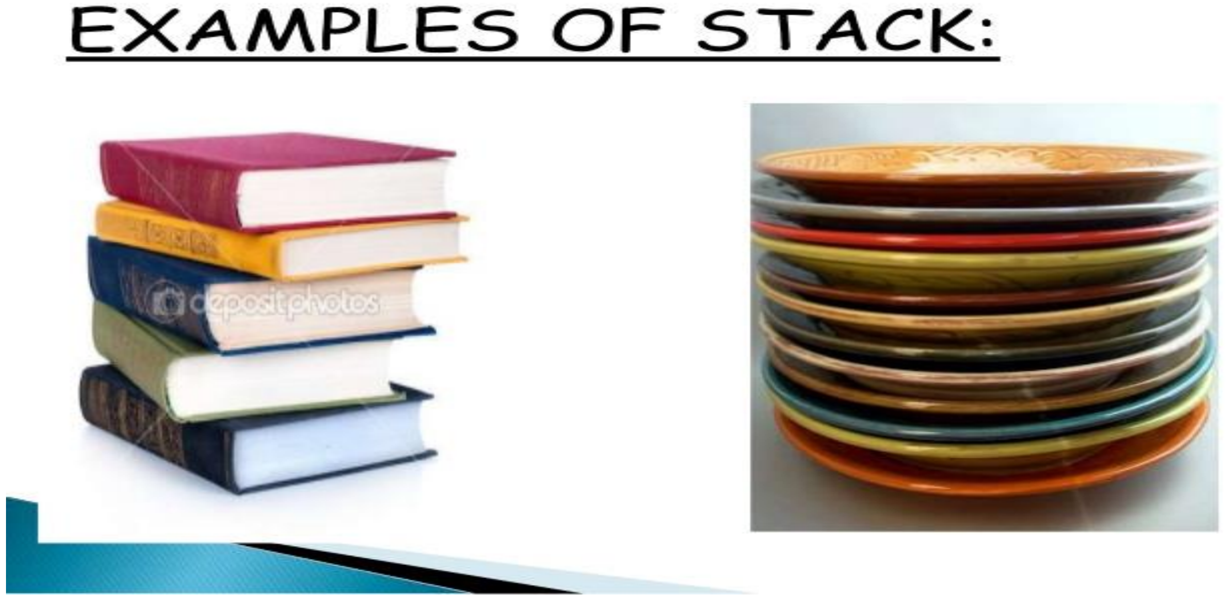
Chapter 2

Some Problems

- Write a menu driven program to determine the following:
    - Check whether a number is prime or not.
    - Check whether the number is palindrome or not
    - Determine the gcd of any two numbers
- WAP to find the largest and smallest number in an array.
- WAP to find the fibonacci series using recursion.
- WAP to find the factorial of a given number using recursion.
- WAP to find the sum of n elements in an array using pass by reference. [Also use calloc or malloc for dynamic memory allocation)
- Create a structure named employee and display the result based on the salary(sorted ascending order)

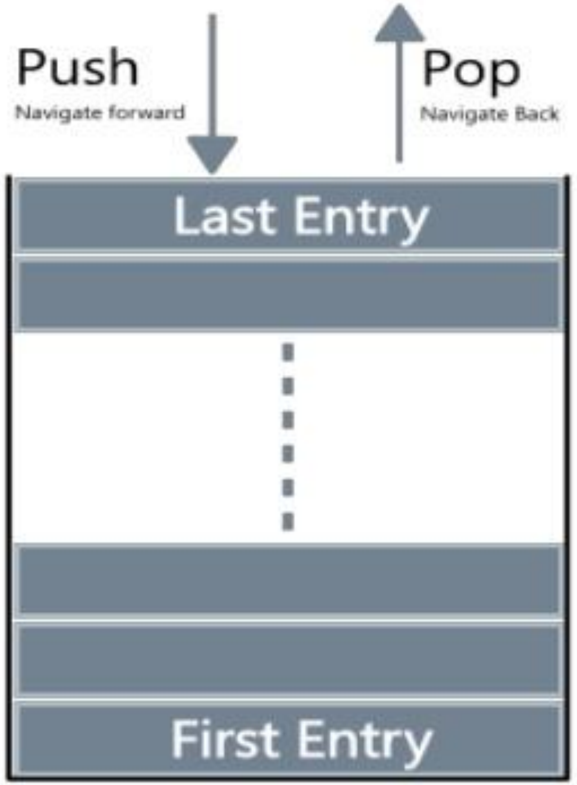
Stack

- A stack is a linear data structure that follows the Last In First Out(LIFO) principle (i.e. the last added elements are removed first).
- Stack is an **ordered collection of items** in to which **new items may be inserted** and from which items may be **deleted at one end** called **top** of the stack



Operations that can be performed on STACK:

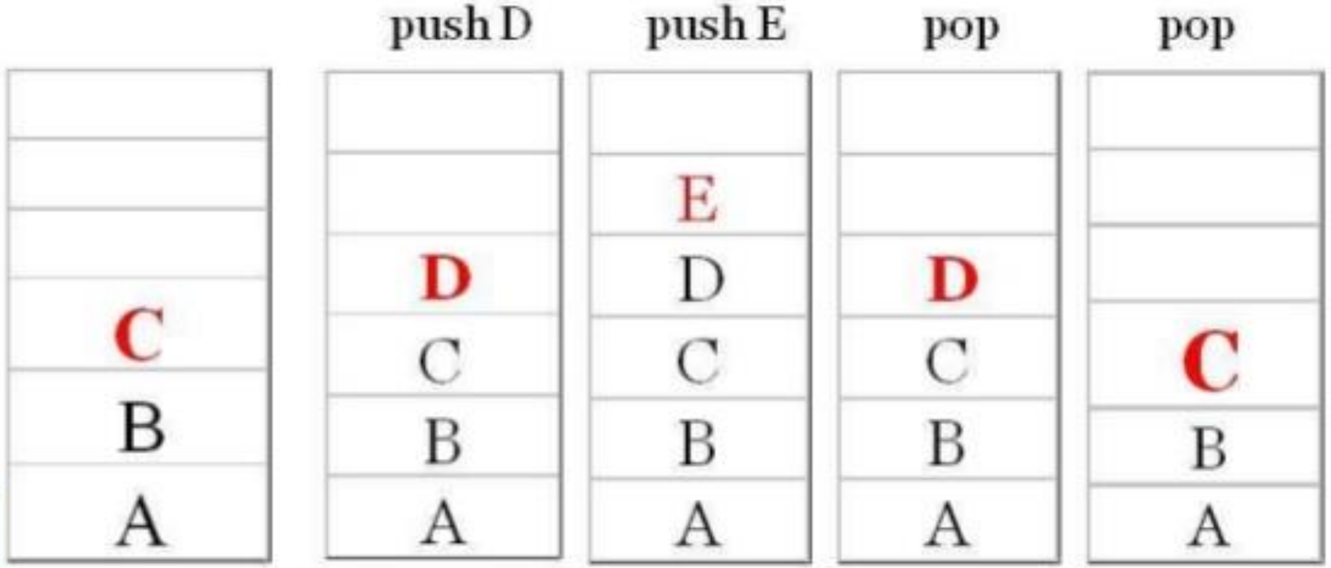
- PUSH.
- POP.



PUSH : It is used to insert items into the stack.

POP: It is used to delete items from stack.

TOP: It represents the current location of data in stack.

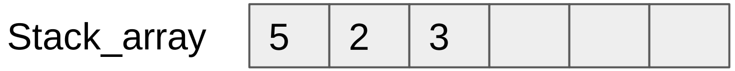


Implementation of Stack

- Array Implementation
- Linked List

Array Implementation

- In array we can push elements one by one from 0th position 1th position ……… (n-1)th position. Any element can be added or deleted at any place.
- We can push or pop elements from the top of the stack only.
    - When there is no place for adding element in the array, then this is called stack overflow. So first we check the value of top with the size of array.
    - When there is no element in the stack, then value of top will be -1. So we check the value of top before deleting the element from the stack.



Here stack is implemented using array and top of the stack value is 2

Algorithm for inserting elements into stack

Push()

```
1. If top=size-1
    1. Then write “Stack is full”
2. Else
    1. Read item or data
    2. top= top+1
    3. stack[top]=item
3. Stop
```

Operation of Stack:- PUSH()

Push Operation:-

```
If (top == (max-1))
        printf(“stack over flow”);
else {
        Top == top + 1;
        Stack_arr[top] = pushed-item;
        }
```

Algorithm for deleting element from stack

Pop()

```
1. If top=-1
    1. Then write Stack is empty
2. Else
    1. Item = stack[top]
    2. Top =top - 1
3. Stop
```

Pop Operation:- POP()

Pop Operation:-

```
If (top ==-1)
            printf(“stack underflow”);
else
{           item =stack[top];
            printf(“Poped element is %d”,stack_arr[top]);
            Top= top--;
}
```

Algorithm: peek()

Peek()

- If
    - top=-1, then print Stack is Empty
- Else
    - Display the top element.

peek()

```
Peek()
If (top ==-1)
            printf(“stack empty”);
else
{
            printf(“Top is =%d”,stack_arr[top]);


}
```

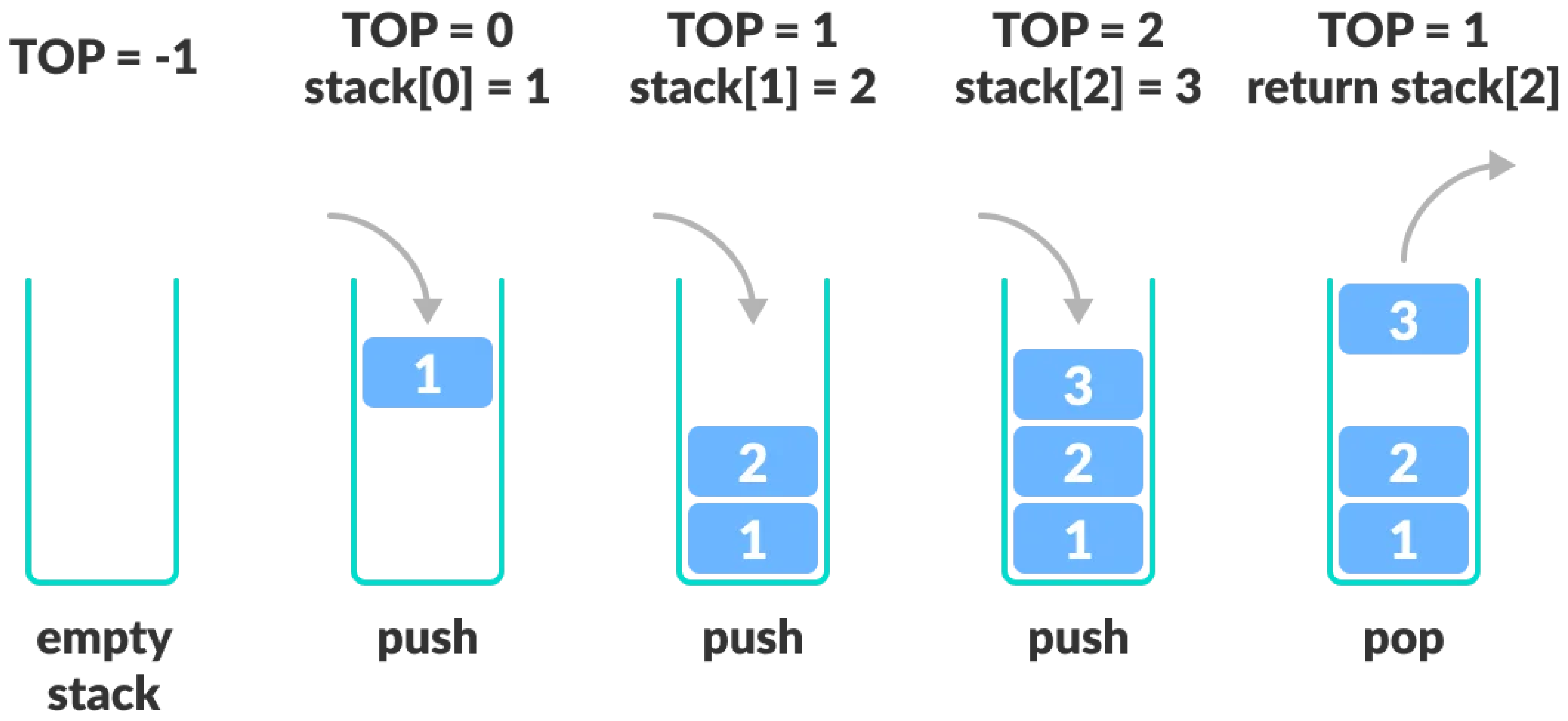
Algorithm:display()

Display()

- If
  - top=-1, then print Stack is Empty
- Else
  - Display all the elements in the array

display()

```
display)()
{
if(top==-1)
        {
        printf(“stack empty);
Else
        {
        printf(“top=%d”,stack[top]);
        }
}
```


TOP = -1
TOP = 0
stack[0] = 1
TOP = 1
stack[1] = 2
TOP = 2
stack[2] = 3
TOP = 1
return stack[2]
1
2
1
3
2
1
3
2
1
empty stack
push
push
push
pop

Stack as an ADT

- In computer science, a stack is an abstract data type that serves as a collection of elements, with two main operations: **Push, which adds an element to the collection and Pop, which removes the most recently added element that was not yet removed**.

```
Struct stack{

        Int stack[5];

        Int top;

}

Void push();

Void pop();

Void display();
```

Application of Stack

1. Conversion of an expression from infix to post fix.
2. Evaluation of an arithmetic expression from post fix expression.

Application of Stack: precedence

Precedence:-

1. $ or ^ (Power), %(remainder) (Right to left associativity)
2. *(mul), /(div) (Left to right associativity)
3. + (add), -(sub) (Left to Right associvity)
4. (
5.)

Application of Stack: Expression

- An expression is find as the number of operands or data items combined with several operators.
- An application of stack is calculation of postfix expression.
- There are basically three types of notation for an expression (Mathematical expression).
  1. **Infix notation**
  2. **Prefix notation**
  3. **Postfix notation**

Application of Stack: Expression

Infix Notation:

- ❖ The **infix notation** is what we come across in our general mathematics, where the **operator is written in-between the operands**.
- ❖ For example: The expression to add two numbers A and B is written in **infix** notation as: **A + B** ; here the **operator '+'** is written in between the **operands A and B**.

Application of Stack: Expression

Prefix Notation:

- The prefix notation is a notation in which the operator(s) is written before the operands.
- The same expression when written in prefix notation looks like: + A B ;As the operator ‘**+**’ is written before the operands A and B, this notation is called prefix (pre means before).

Application of Stack: Expression

Prefix Notation:

- ❖ The prefix notation is a notation in which the operator(s) is written before the operands.
- ❖ The same expression when written in prefix notation looks like: + A B ;As the operator ‘**+**’ is written before the operands A and B, this notation is called prefix (pre means before).

Application of Stack: Expression

Postfix Notation:

- ❖ In the postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as suffix notation or reverse polish notation.
- ❖ The above expression if written in postfix expression looks like: A B +

Application of Stack

Reversing Strings

Conversion of an expression from infix to postfix.

Evaluation of arithmetic expression from postfix expression

Conversion of Infix to post fix:-

Algorithm:-

1. Add a unique symbol # in to stack and at the end of array infix.
2. Scan symbol of array infix fro left to right.
3. If the symbol is left parenthesis '('then add in to the stack.
4. If symbol is operand then add it to array post fix.
5. (i) If symbol is operator then pop the operator which have same precedence or higher precedence then the operator which occurred.

   (ii) add the popped operator to array post fix.

   (iii) Add the scanned symbol operator in to stack.
6. (i) If symbol is right parenthesis ')' then pop all the operators from stack until left parenthesis '(' in stack.

   (ii) Remove left parenthesis '(' from stack.
7. If symbol is # then pop all the symbol form stack and add them to array post fix except #.
8. Do the same process until # comes in scanning array infix.

Example of conversion infix to postfix: A+B-C+D

Input Expression	Stack	Postfix expression
A	#	A
+	#+	A
B	#+	AB
-	#-	AB+
C	#-	AB+C
+	#+	AB+C-
D	#+	AB+C-D+

Example of conversion infix to postfix :K+L-M*N+(O^P)*w/u/v*T+Q

Input	Stack	Output
K	#	K
+	#+	K
L	#+	KL
-	#-	KL+
M	#-	KL+M
*	#*-	KL+M
N	#*-	KL+MN
+	#+	KL+MN*-
(	#(+	KL+MN*-
O	#(+	KL+MN*-O

Input	Stack	Output
^	#^(+	KL+MN*-O
P	#^(+	KL+MN*-OP
)	#+	KL+MN*-OP^
*	#*+	KL+MN*-OP^
w	#*+	KL+MN*-OP^W*
/	#/+	KL+MN*-OP^W*
u	#/+	KL+MN*-OP^W*U
/	#/+	KL+MN*-OP^W*U/
v	#/+	KL+MN*-OP^W*U/V
*	#*+	KL+MN*-OP^W*U/V/
T	#*+	KL+MN*-OP^W*U/V/T
+	#+	KL+MN*-OP^W*U/V/T*+
Q	#+	KL+MN*-OP^W/U/V/T*+Q+

CONVERSION OF INFIX INTO POSTFIX

2+(4-1)*3 into 241-3*+

CURRENT SYMBOL	ACTION PERFORMED	STACK STATUS	POSTFIX EXPRESSION
(	PUSH C	C	2
2			2
+	PUSH +	(+	2
(	PUSH (	(+(	24
4			24
–	PUSH –	(+(–	241
1	POP		241–
)		**(+**	**241–**
*	PUSH *	(+*	241–
3			241–3
	POP *		241–3*
	POP +		241–3*+
)			

Evaluation of postfix expression

In this case the stack contains the operands instead of operator

Whenever any operator occurs on scanning we evaluate with last two element of the stack.

Algorithm

1. Scan the symbol of array post fix one by one from left to right.
2. If symbol is operand, two push into stack.
3. If symbol is operator then pop last two element of the stack and evaluate as [top-1] operator [top] and push it to stack.
4. Do the same process while scanning from left to right.
5. Pop the element of the stack which will be value of evaluation of postfix arithmetic expression.

Post fix expression ABCD ^+*EF^GH/*-

Evaluate postfix expression where A=4, B=5, C=4 D=2, E=2, F=2, G=9, H=3

Step	Symbol	Operand in stack
1	4	4
2	5	4,5
3	4	4,5,4
4	2	4,5,4,2
5	^	4,5,16
6	+	4,21
7	*	84

Step	Symbol	Operand in stack
8	2	84,2
9	2	84,2,2
10	^	84,4
11	9	84,4,9
12	3	84,4,9,3
13	/	84,4,3
14	*	84,12
15	-	72

Convert from infix to postfix expression

1. Convert from infix to postfix expression((A+B)*C – (D-E) 4F+G

2. Postfix expression ABCD $+* EF$GHI * - Evaluate postfix expression where A = 5, B = 5, C = 4, D = 2 E = 2, F= 2 , G = 9, H= 3 ,I=2

3. Convert from the infix to postfix expression A* (B+C $D) – E $F * (G/H)

4. Evaluate the postfix expression: AB+C*DE - - FG +$ =>1,2, +3, *4, 3, -, -, 2, 1, +, $, #

Conversion of Infix to Prefix

Iterate the given expression from left to right, one character at a time

Step 1: First reverse the given expression

Step 2: If the scanned character is an operand, put it into prefix expression.

Step 3: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 4: If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack untill we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.

If the scanned character is opening round bracket ('('), push it into operator's stack.

If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Repeat Step 2,3 and 4 till expression has character

Step 5: Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Example: A+B*C+D into prefix

First of all reverse the infix expression: D+C*B+A

Input	Stack	Expression	Action
D		D	Add D into expression string
+	+	D	Push + into stack
C	+	DC	Add C into expression string
*	+*	DC	Precedence of * is higher so push * into stack
B	+*	DCB	Add B into expression string
+	++	DCB*	Precedence of + is lower. So pop * from stack
A	++	DCB*A	Add A into expression string
		DCB*A++	Pop all operators one by one as the end of expression is reached

Reverse the expression to get prefix expression: ++A*BCD

Example: Infix to prefix

Infix Expression: (A+B)*C

Reverse the expression: C*)B+A(

Input	Stack	Expression	Action
C		C	Add C into expression string
*	*	C	Push * into stack
)	*)	C	Push) into stack
B	*)	CB	Add B into expression string
+	*)+	CB	Push + into stack

Example: Infix to prefix

Input	Stack	Expression	Action
A	*)+	CBA	Add A into expression string
(	*	CBA+	(pair matched so pop + from stack
		CBA+*	Pop all operators one by one as we have reached end of the expression

Reverse the expression to get prefix expression *+ABC

Infix to Prefix conversion example

Infix Expression: (A+B)+C-(D-E)^F

First reverse the given infix expression: **After Reversing:** F^)E-D(-C+)B+A(

Input	**Stack**	**Expression**	**Action**
F		F	Add F into expression string
^	^	F	Push ^ into stack
)	^)	F	Push) into stack
E	^)	FE	Add E into expression string
-	^)-	FE	Push - into stack

Input	Stack	Expression	Action
D	^)-	FED	Add D into expression string
(	^)	FED-	`(` Pair matched, so pop operator `-`
-	-	FED-^	- Has less precedence than ^. So pop from stack
C	-	FED-^C	Add C into expression string
+	+	FED-^C-	Same precedence but associativity from right to left so pop from stack
)	+)	FED-^C-	Push) into stack
B	+)	FED-^C-B	Add B into expression string

Input	**Stack**	**Expression**	**Action**
+	+)+	FED-^C-B	Push + into stack
A	+)+	FED-^C-BA	Add A into expression string
(	+	FED-^C-BA+	‘(‘ Pair matched, so pop operator ‘+’
		FED-^C-BA++	Pop all operators one by one as we have reached end of the expression

Now **reverse** the expression to get prefix expression ++AB-C^-DEF

Convert (A+B)*(C+D) into prefix

Reverse infix expression:)D+C(*)B+A(

Input	Stack	Expression	Action
)	)		Push) into stack
D	)	D	Add D into expression string
+	)+	D	Push + into stack
C	)+	DC	Add C into expression string
(		DC+	(Pair matched so pop + from stack
*	*	DC+	Push * into stack
)	*)	DC+	Push) into stack
B	*)	DC+B	Add B into expression string
+	*)+	DC+B	Push + into stack
A	*)+	DC+BA	Add A into expression string
(	*	DC+BA+	(Pair matched so pop + from stack
		DC+BA+*	Pop elements one by one from stack

Reverse the expression to get prefix expression: *+AB+CD

Thank you