The Shirpur Education Society's

R. C. Patel College of Engineering & Polytechnic, Shirpur

COMPUTER ENGINEERING

Unit – IV STACK

Total Marks:12

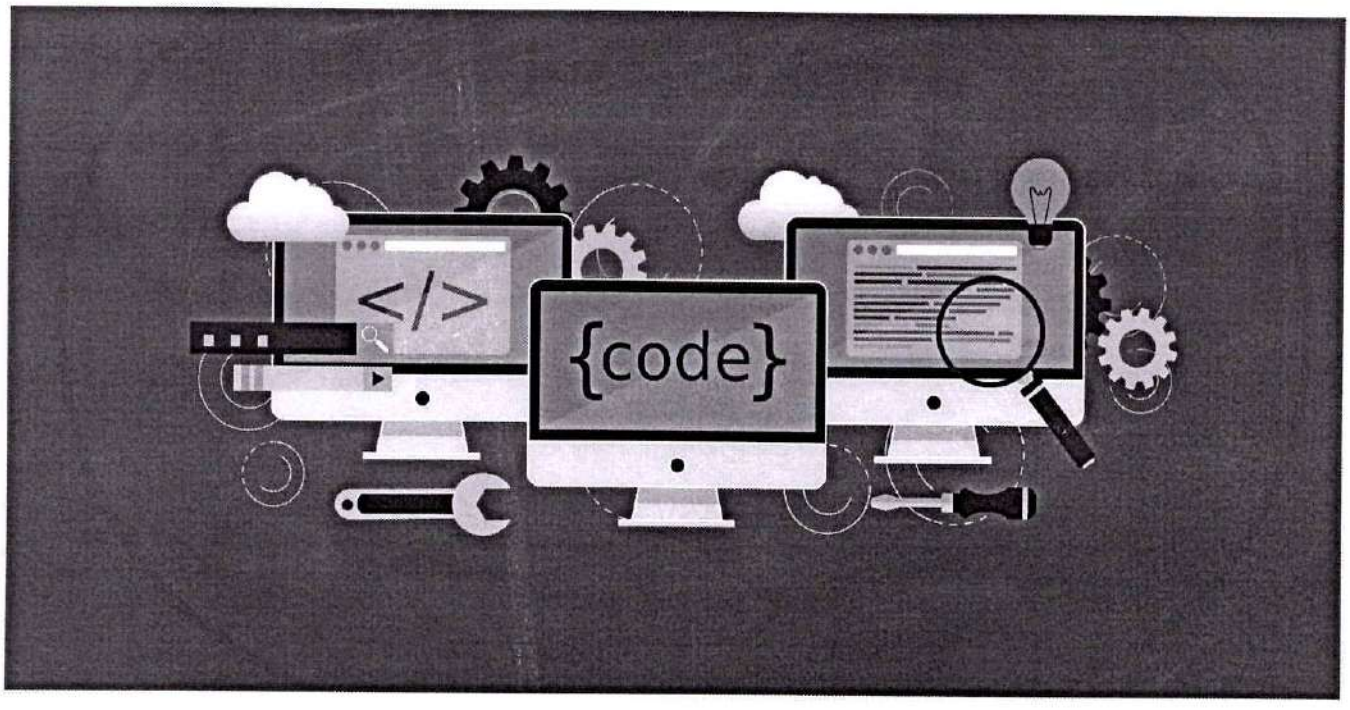
Course Title – Data Structure using C

Course Code - 313301

Programme Name - Computer and Computer Science Engineering

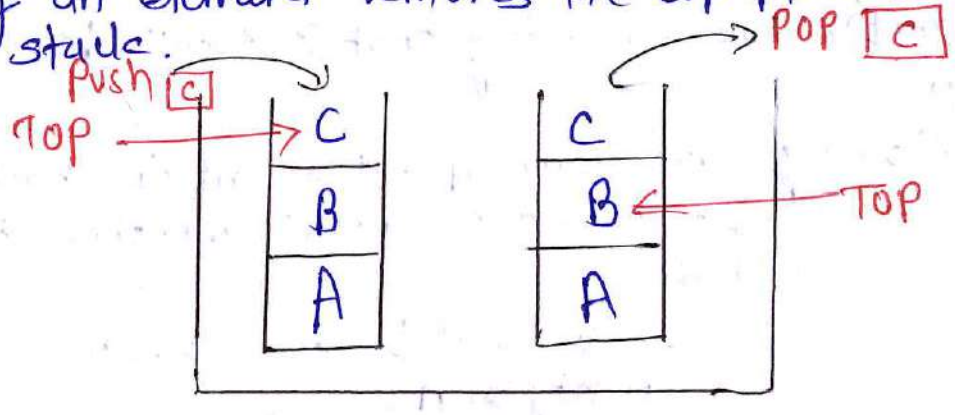
Semester - Third

Prepared By: Mrs.Poonam S.Sonawane



Stack - A stack is a linear data structure that follows the Last in, first out (LIFO) Principle. This means the element added last is the first one to be removed.

- Pushing an element onto the stack is like adding a new plate on top.
- Popping an element removes the top plate from the stack.



Basic Operation on stack

1. Push() - To insert an element into the stack.
2. Pop() - To remove an element from the stack.
3. top() - Returns the top element of the stack.
4. isEmpty() - Returns true if stack is empty else false.
5. isFull() - Returns true if the stack is full else false.

- A stack is a Abstract data type with predefined capacity which means that it can store element of limited size.
- A stack work on LIFO pattern.

Top pointer always point to top element of the first.
A stack is filled form the bottom to top when we perform delete operation on stack there is only one way for entry and exit as other end is closed.

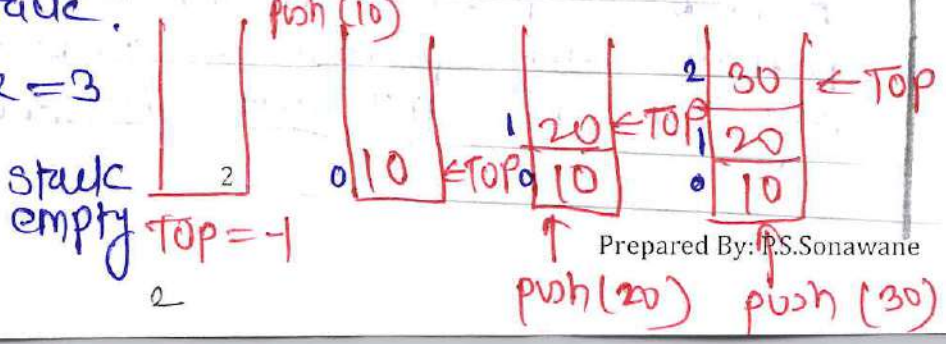
Stack as an ADT

- stack also defined as an Abstract Data type.
- Abstract data type whose behaviour is defined by set of values and set of operation.
- keyword Abstract is used as we can use the data type. we can perform different operation, But how those operation are working that is totally hidden from user.
- following are few operation of the stack ADT:—
  1. isfull() — It is used to check whether stack is full or not.
  2. isEmpty() — It is used to check whether stack is empty or not.
  3. push(x) — It is used to push x into the stack.
  4. pop() — It used to delete one element from the top() of stack.
  5. peek() — It is used to to get top elener on the stack.

* Operation on stack

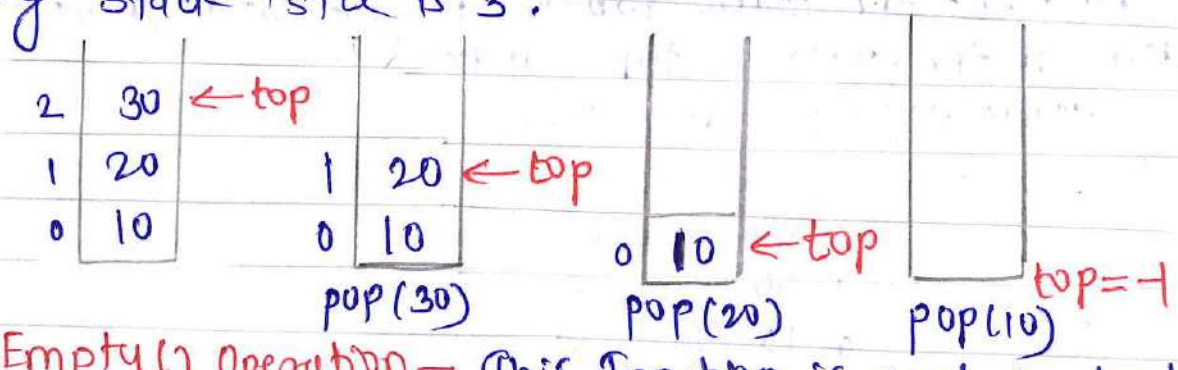
1. Push Operation — the process of adding new element to the top of the stack is called PUSH operation.
- Before inserting an element in a stack we check whether stack is full or not.
- As the new element is added at the top after every push operation the top is incremented by 1.
- the element will be inserted until we reach the capacity of stack.

eg stack size = 3



2. Pop Operation – The process of deleting an element from top of a stack is called as Pop Operation.
- Before deleting the element from the stack we check whether stack is empty or not.
- If the stack is not empty we first access the element which is pointed by the top.
- After every pop operation the top of the stack is decremented by 1.

e.g Stack size is 3.



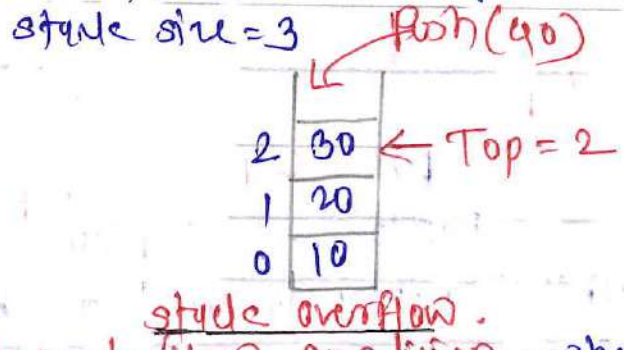
3. isEmpty() Operation – This function is used to check whether stack is empty or not.
- This function returns boolean value true if stack is empty otherwise false.
- This function is useful while returning the values from stack because retriving the values from empty stack is not possible.

4. isFull() operation – This function is used to check whether a stack become full or not.
- It returns true if stack is full else false.
- It is useful when inserting the element into stack, no more elements can be added the stack it is full.

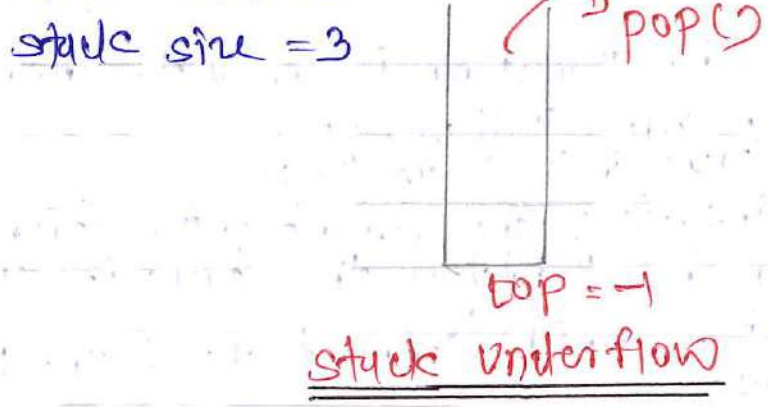
*Top – It is pointer, which keeps track of the top element in the stack.
- If an array of size N is to be a stack.
- then top will be -1 when stack is empty.
- & top will be N-1 when stack is full.

* Stack Operation Condition –

1. Stack Overflow condition – when stack is completely full and we try to insert more element onto stack then this condition is called as stack overflow condition.
- no element will be inserted until any element is deleted.
- This is situation where stack become full and no more element pushed onto stack.
- At this point the stack top is present at highest location of the stack ie top = max - 1



2. Stack underflow condition – when stack is empty and we try to delete more element from it. then this condition is called as underflow condition.
- This is situation when the stack contains no ele. At this point the top of stack is present at the bottom of the stack i.e top = -1.



* Stack implementation using array:-

Array can be used to hold element of a stack. i.e size of array is fixed.
- while in a stack there is no fixed size, since size of stack changed with no of elements inserted or deleted from it.

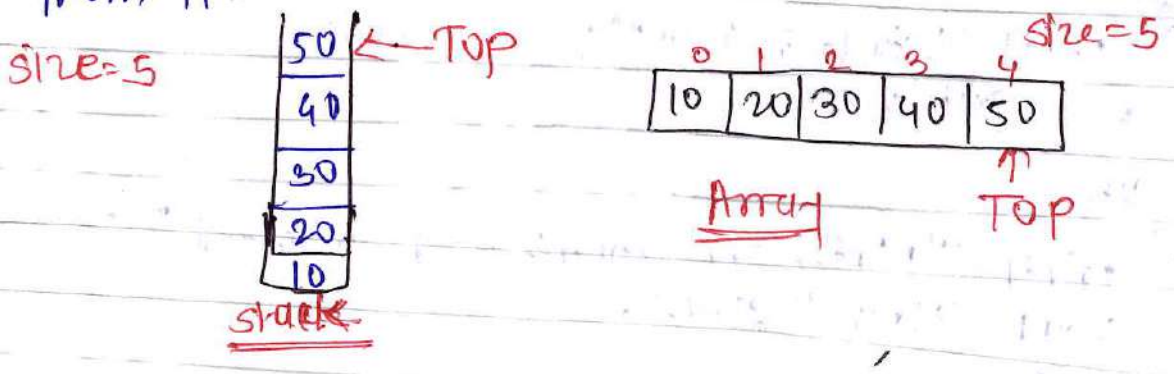


Fig: Representation of stack using array

- Top variable is used to keep track of the top most element.
- Initially Top is set to -1.
- A stack with top as -1 is an empty stack.
- when the value of top become max-1 after series of insertion, It is full.
- After push operation Top is incremented by 1 i.e top = top+1
- After pop operation Top is decremented by 1 i.e top = top-1
- All the operation regarding the stack implementation

1. **Push operation** — Adding an element on the top of stack is termed push operation.

push operation has following steps:-
1. check if stack is full.
2. if ~~check~~ stack is full produces error and exit.
3. if stack is not full, increment the top variable of the stack so that it can refer to the next memory Location.
4. Add a data element at the increment top, position.

* Implementation of push operation:-

```
void push()
{
  int ele;
if(top==max-1)
{ printf("stack is full");
    exit(0);
}
else
{ printf("\n Enter the element to be inserted");
  scanf("\n%d",&ele);
    top++;
  st[top] = ele;
}
}
```

2. Pop operation :- Removing a data element from the stack is called pop operation.
pop operation has following steps:-
1. check wheather stack empty or not.
2. if stack is empty produces error and exit.
   i.e we cannot delete element.
3. If stack is not empty, we must delete the element by decrementing the position of top.

Implementation of pop operation:-

```
void pop()
{ int ele;
  if(top==-1)
  { pf("stack is empty");
  }else
  { ele = st[top];
    printf("Deleted element is :%d", ele);
    top--;
}
```

3.* Implemention for display operation:-

To display the stack either in the same way they arrive or the reverse.

* display from top to bottom —

```
void display()
{
    while(top>=0)
    {
        printf("%d", st[top]);
        top--;
    }
}
```

* display from bottom to top —

```
void display()
{
    int i;
    for(i=0; i<=top; i++)
    {
        printf("\n%d", st[i]);
        top--;
    }
}
```

* stack Representation using Linked List —

- stack is implemented using dynamic data structure Linked List.
- Using Linked list for Application of stack make a dynamic stack.
- It means that size is change automatically according to the element present.

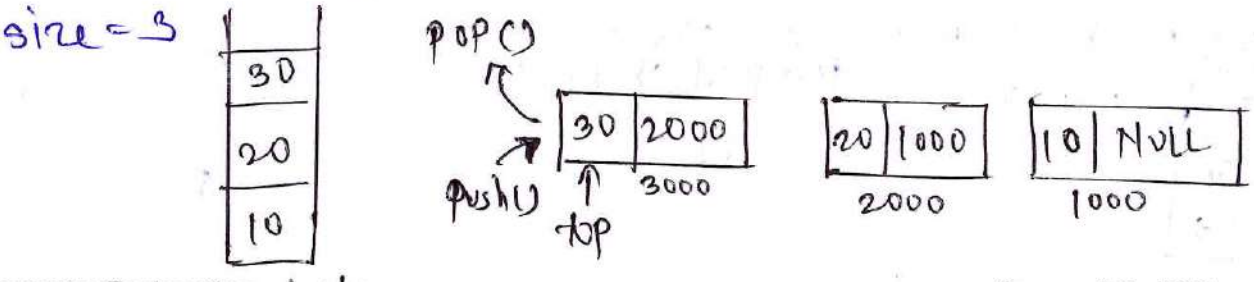


fig :- representation of stack using Linked List.

- stack is implemented using linked list can be represented by the pointer to the top node.
- each node is that linked list represents the element of the stack.
The type of linked list here is singly linked list.

* Application of stack

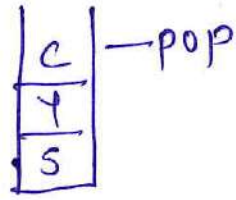
1. **Reversing a list:-** one of the simplest appln of stack is reversal a list.
- which can simply be done by putting the invidual character and when complete line is pushed on the stack then individual element of stack are popped off.
- This can be achived very easily by reading the input string character by character and push that onto stack till end of the string is reached.
- now, keep on popping the character from stack plate them in a string till stack become empty.

e.g Reverse a string "stco"

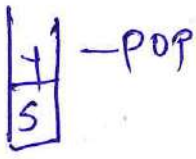
Input string	character Read	stack
stco ↑	s	s
stco ↑	t	t s
stco ↑	c	c t s
stco ↑	o	o c t s

Now perform pop operation —

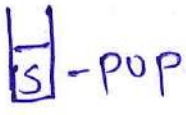
stack	pop()	Popped character	Reverse string
o c t s		o	o ↑



C → OC



Y → OCY



S → OCYS

Reverse string = OCYS

2. Polish Notation :- polish notation in data structure is a method of expressing mathematical, logical, algebric equation.
- This notation is used by compiler to evaluate mathematical equations based on their order of operation.
- There are 3 types of notation that commonly used-

1. Infix Notation :-
- If the operator symbols are placed between its operands then the expression is in infix notation.

e.g 1. A + B ← operands
↑ operator.

2. (A - C) * B

2. Prefix Notation :- It is also known as 'polish notation'
- If the operator symbol are placed before its operands then the expression is prefix notation

e.g 1. +AB
2. * - ACD.
3. /+AB - CD.

3. Postfix Notation - It also known as suffix Notation or reverse polish Notation.
If the operator symbols are placed after its operands then the expression is in postfix Notation

eg 1. AB+
2. AC-*D
3. AB+CD-/

* Converting of infix to postfix Expression -

- To convert infix to postfix expression, we will use stack.
- To convert infix to postfix, we assign priority to each of the operator present in the expression.
- each operator has its priority for an expression

Highest priority operators = *, /, %
lower priority operators = +, -

Algorithm

step 1- Initialize the empty stack and postfix string.
2. scan the infix expression from left to right.
3. If the scanned character is an operand, add it to postfix string.
4. else if scanned character is operator then -
* If precedence of scanned operator is greater then the precedene of the operator in the stack or stack is empty or stack contains '(' then push it in stack.
* else pop all the operators from stack which are greater than or equal to in precedence than that scanned operator.
* After doing this push the scanned operator to the stack.
* If you encounter parenthesis while popping then stop there and push scanned operator in the stack.

5. If scanned character is '(' push it to the stack.
6. If scanned character is ')' pop the stack and add it into postfix string until '(' encountered and discard both parenthesis.
7. Repeat step 3 to 6 until infix expression is scanned
8. print the output.
9. pop and output from the stack until it is not empty.

e.g Infix expression — (A+B* C-D)/(E*F)

Input	stack	postfix string
(A+B* C- D)/(E*F)	(	
A+B*C-D)/(E*F)	(	A
+B*C-D)/(E*F)	(+	A
B* C-D)/(E*F)	(+	AB
* C-D)/(E*F)	(+*	AB
C-D)/(E*F)	(+*	ABC
-D)/(E*F)	(+	ABC*
	(	ABC*+
	(-	ABC*+
D)/(E*F)	(-	ABC*+D
)/(E*F)	(	ABC*+D-
/(E*F)	/	ABC*+D-

(E*F) ↑	/C	ABC*+D-
E*F) ↑	/C	ABC*+D-E
*F) ↑	/C*	ABC*+D-E
F) ↑	/C*	ABC*+D-EF
) ↑	/C	ABC*+D-EF*
	/	ABC*+D-EF*

postfix string- ABC*+D-EF*

*Evaluation of postfix Expression -

In postfix Expression the operator is placed after the operands.

e.g AB+ ← operator
↑
operands

- To evaluate postfix expression using stack, we can use following steps:-

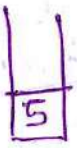
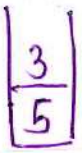
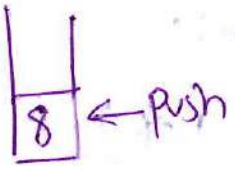
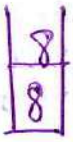
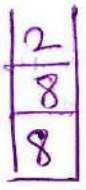
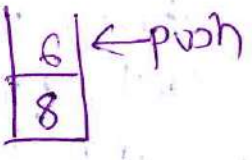
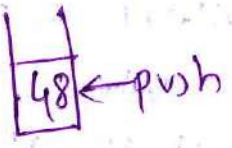
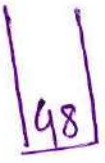
step 1- Read all the symbol one by one from left to right in given expression.

2. If the reading symbol is operands then push it on to the stack.

3. If the reading symbol is operator then, pop two operands from the stack then perform reading symbol operation using two operands and push result back on to the stack.

4. If no more symbols for scan then display popped value as final result.

e.g Postfix expression - 53+82-*

Reading symbol	stack	Evaluated.
53+82-* ↑		—
3+82-* ↑		—
+82-* ↑	 ← push	pop value 1 = 3 Pop value 2 = 5 Value2 + value1 = Reasult 5 + 3 = 8
82-*		—
2-* ↑		—
-* ↑	 ← push	pop value1 = 2 Pop value2 = 8 Value2 - value1 = reasult 8 - 2 = 6
* ↑	 ← push	pop value1 = 6 pop value2 = 8 Value2 * value1 = 8 * 6 Pop & Display Reasult 48
End		

***Recursion :-** Recursion is the process of calling function by itself.

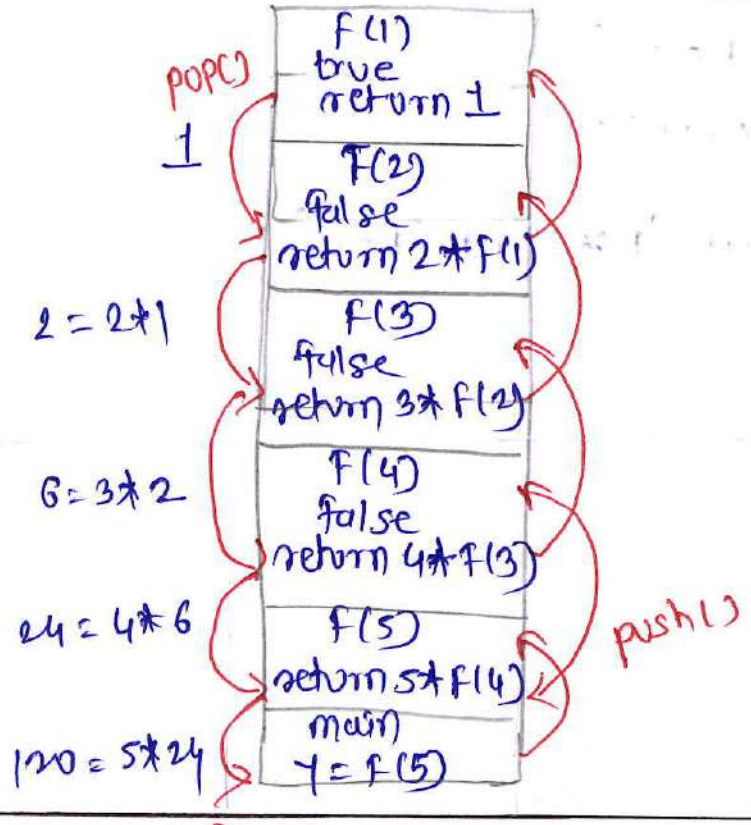
- Recursion function body contains function call statement that's calls itself repeatedly.
- when the recursive function calls itself from body, stack is used to store temporary data handled by the function in every iteration.

e.g consider n=5 and find factorial using recursion.

function call from main() = fact(n)

```
int fact(int n)
{
    if(n==1)
    return 1;
    else
    return (n* fact(n-1));
}
```

- Each time when function makes a call itself, it saves its current status in stack and executes next function call.
- when fact() function calls from main function. It initalize n=5 Return statement inside function body executes a recursive function call.
- first value of n is stored using push in stack i.e n=5 and function call again with value 4(n-1)
- In every call value of n is push and then reduce by 1.
- When function is called with n=1 recursive process stops.
- At the end all the value from stack are retrived using pop() to perform multiplication to calculate factorial of number.



* Program to find factorial of given number using Recursion.

```
#include <stdio.h>
#include <conio.h>
int fact (int n)
void main()
{
    int n, result;
    printf ("\n Enter the number");
    scanf ("%d", &n);
    result = fact(n);
printf ("\n factorial is=%d", result);
getch();
}
```

```
int fact (int n)
{
    if (n==1)
     return 1;
    else
    return (n * fact (n-1));
}
```

* Difference between stack and Queue.

Stack	Queue
1. Stack is linear data structure that follows LIFO (Last in first out)	1. Queue is linear data structure that follows FIFO (First in first out)
2. Insertion operation called as push.	2. Insertion operation called as Enqueue.
3. Deletion operation is called pop.	3. Deletion operation is called Dequeue.
4. use single pointer called Top.	4. use two pointer called front and Rear.
5. The last inserted element is removed first.	5. The first inserted ele. is removed first.
6. Ex: stack of plates, Browser history, undo/Redo operation	6. ex: cpu scheduling, printer queue, Ticket booking system.
7. Insertion and Deletion occur at the same end (Top)	7. Insertion occur at the Rear and deletion occurs at the front.
8. e.g push 10, 20, 30 - POP returns 30	8. Enqueue 10, 20, 30 Dequeue returns 10.

* Application of stack.

1. Function calls - stack manage the "active" function in program. When function is called, its execution state is pushed onto the stack. When it finishes, it is popped to return control to the caller.

2. Recursion:- since recursion is essentially a function calling itself, then stack stores a "snapshot" of each call (including local variables) so the program doesn't lose its place. e.g factorial(n!)

3. Expression Evaluation:- stacks are used by compilers and calculators to handle the order of operation without needing complex parentheses. e.g 3+4*5 finally result total 35.

4. Syntax parsing:- stack are perfect for "balancing" symbols. They ensure that every opening bracket has a corresponding closing bracket in the correct order. e.g A stack it pushes '(' and '{' onto the stack and also when it encounter '}' and ')' it pop the top element

5. Memory Mangement: The stack is a specific region of RAM used for automatic variable allocation. It is incredibly fast because it only allocates and deallocates memory in strict Last-In, FIRST-out (LIFO) order.

6. Time & memory efficiency - push and pop operation on stack can be performed in constant time-O(1) enable efficient data access.

7. Last In, first out (LIFO) - stack follow LIFO principle, ensure last ele. added the first one removed. This behaviour is useful scenario like function calls and expression evaluation.

* Convert the following expression infix to postfix.

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

→

Input	stack	postfix string
[(A+B)-C*(D/E)]+F	[	-
(A+B)-C*(D/E)]+F	[(	
A+B)-C*(D/E)]+F	[(	A
+B)-C*(D/E)]+F	[(+	A
B)-C*(D/E)]+F	[(+	AB
)-C*(D/E)]+F	[	AB+
-C*(D/E)]+F	[-	AB+
C*(D/E)]+F	[-	AB+C
*(D/E)]+F	[-*	AB+C
(D/E)]+F	[-*(	AB+C
D/E)]+F	[-*(	AB+CD
/E)]+F	[-*(/	AB+CD
E)]+F	[-*(/	AB+CDE

] + F	[− *	AB + CDE /
] + F		AB + CDE / * −
+ F	+	AB + CDE / * −
F	+	AB + CDE / * − F
End	−	AB + CDE / * − F +

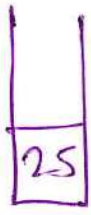
postfix expression = AB + CDE / * − F +

* Evaluate the following postfix expression with suitable digram using stack.

10, 2, *, 15, 3, /, +, 12, 3, +, +.

Reading Symbol	stack	Evaluated
10, 2, *, 15, 3, /, +, 12, 3, +, + ↑	10	—
2, *, 15, 3, /, +, 12, 3, +, + ↑	2 / 10	—
*, 15, 3, /, +, 12, 3, +, + ↑	20	10 * 2 = 20
15, 3, /, +, 12, 3, +, + ↑	15 / 20	
3, /, +, 12, 3, +, + ↑	3 / 15 / 20	
/, +, 12, 3, +, + ↑	5 / 20	15 / 3 = 5

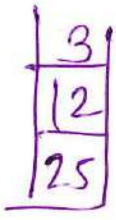
+, 12, 3, +, +
↑



20 + 5 = 25

12, 3, +, +
↑

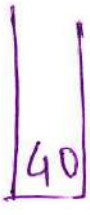
3, +, +
↑



+, +
↑

12 + 3 = 15

+
↑



25 + 15 = 40

Valve of postfix expression = 40

*To convert an infix to prefix notation

① [(A+B)+C]*D

→ [(A+B)+C]*D

((A+B)+C)*D equivalent with parenthesis

D*(C+(B+A)) Reversed expression.

infix to postfix conversion table.

Input	stack	postfix o/p
D*(C+(B+A)) ↑		D
*(C+(B+A)) ↑	*	D
(C+(B+A)) ↑	*(	D
C+(B+A)) ↑	*(	DC
+(B+A)) ↑	*(+	DC
(B+A)) ↑	*(+(	DC
B+A)) ↑	*(+(	DCB
+A)) ↑	*(+(+	DCB

A)) ↑	*(+(+	DCBA
)) ↑	*(+	DCBA+
) ↑	*	DCBA++
End	empty	DCBA++*

postfix of reversed expression = DCBA++*

. Reverse the postfix = DCBA++* ⇒ *++ABCD

Prefix Expression

infix [(A+B)+C]*D = prefix *++ABCD .

② A[(B*C)+D]

→ A*((B*C)+D)

Reversed — (D+(C*B))*A

input	stack	postfix string
(D+(C*B))*A ↑	(	empty
D+(C*B))*A ↑	(	D
+(C*B))*A ↑	(+	D
(C*B))*A ↑	(+(	D

(*B))*A ↑	(+(	DC
*B))*A ↑	(+(*	DC
B))*A ↑	(+(*	DCB
))*A ↑	(+	DCB*
)*A ↑		DCB*+
*A ↑	*	DCB*+
A ↑	*	DCB*+A
End	empty	DCB*+A*

postfix exp = DCB*+A*

DCB*+A* = *A+*BCD

Final Ans.

A[(B*C)+D] ⟹ *A+*BCD.

* Question Bank *

Q.1) write any two operations performed on the stack. (2m) (W-25).

Q.2) state the LIFO principle of stack - 2m (S-25)

Q.3) List any four application of stack data structure. (2m S-25)

Q.4) List any two application of stack (2m W-24)

Q.5) Difference betn stack and queue (2m W-24)

Q.6) Define stack with suitable example (2m S-24)

Q.7) Convert the following infix expression to its postfix from using stack : A+B-C*D/E+F (2m W23)

Q.8) write any 2 operation performed on the stack (2m-S23)

Q.9) Explain stack overflow and stack underflow with example. (4m Winter-25)

Q.10) Convert the following Infix expression to its postfix from using stack. show the details of stack at each step of conversion Expression : [[(A+B) - C*(D/E)]+F (4m W25)

Q.11) Evalvate the following postfix expression —
10, 2, *, 15, 3, 1, +, 12, 3, +, +.
show digrammatically each step of evalvation using stack (4m winter-25) (summer-25)

Q.12) write down step by step conversion of the following infix expression to postfix expression using stack
((A+B)*C) ^ (D-E) (4m summer-25) (summer-24)

Q.13) Describe stack overflow and stack underflow condition with help of example (4m - summer 25) (S-23)

Q.14) Convert following expression into postfix form. give stepwise procedure.
A*(B+C)/D-G (4m. W-24)

Q.15) show the effect of PUSH and POP operation on to the stack of size 10. The stack contains 10, 20, 30, 40, 50 and 60 with 60 begins at top of the stack. show diagrammatically the effect of—

i) Push 55

ii) Push 70

iii) pop

iv) Pop. sketch the final structure of stack after performing the above said operations. 4m (Winter-24) (Summer-23)

Q.16) convert the following infix expression into prefix expression using stack in tabular form:

(A+B) * C^D − (E/F). 4m (Winter-24)

Q.17) Define the term recursion. write a program in C to perform multiplication of two numbers using recursion. 4m (winter-24)

Q.18) Explain push and pop operation on stack with suitable example. 4m (Summer-24)

Q.19) Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion:

Expression: A*B^C-D/E+[F/G] 4m winter-23)( )

Q.20) show the effect of PUSH and pop operation on the stack of size[10]. The stack contains 10, 20, 25, 15, 30 and 40 with 40 begins at top of stack. show diagrammatically the effect of:

i) push (45) ii) push (50)

iii) pop iv) push (55) 4m (winter-23) (3)

Q.21) find out prefix equivalent of the expression—

i) [(A+B)+C]*D

ii) A[(B*C)+D] 4m (winter-23)

Q.22) Convert the infix expression to its postfix expression using stack ((A+B)*D)^(E-F). Show diagrammatically each step of conversion. (4m summer-23)

Q.23) Evaluate the following postfix expression

4 6 2 + * 6 3 / -

Show diagrammatically each step of evaluation using stack. (4m summer-23)