Unit-2

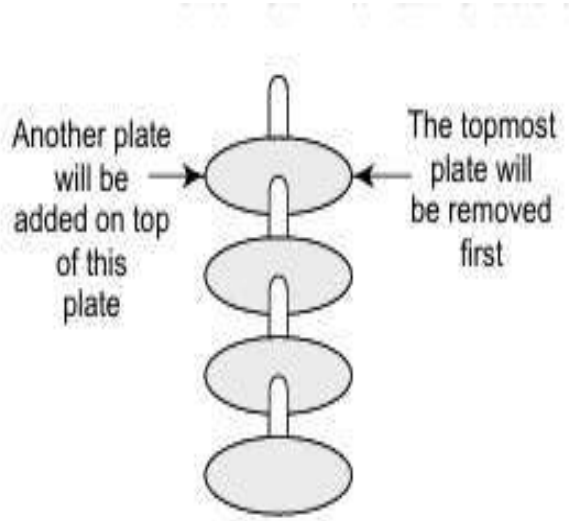
STACK

Contents

- Stacks: Introduction and Definition, Representation
- Operations on Stacks
- Applications of Stacks
- Representation of Arithmetic Expressions: Infix, Postfix, Prefix.

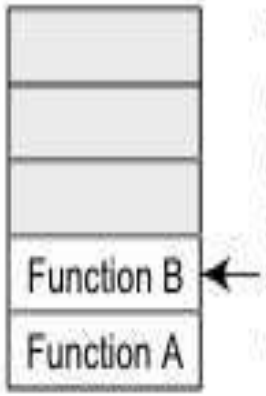
Introduction

- Stores its elements in an ordered manner
- We can add and remove an element (i.e., a plate) only Another plate will be added on top of this plate at/from one position which is the topmost position.
- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.
- Stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

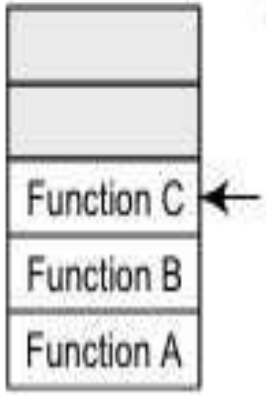


System stack in the case of function calls

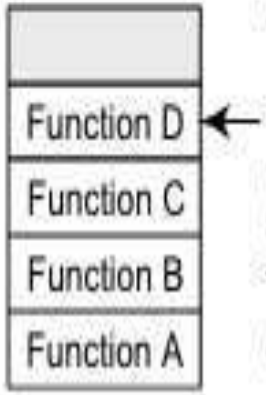
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.



When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.

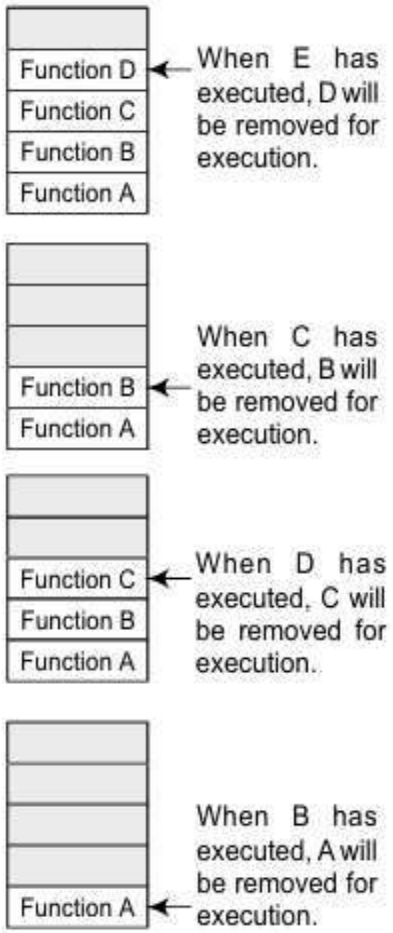


When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.



When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

System stack when a called function returns to the calling function



Array Representation of Stack

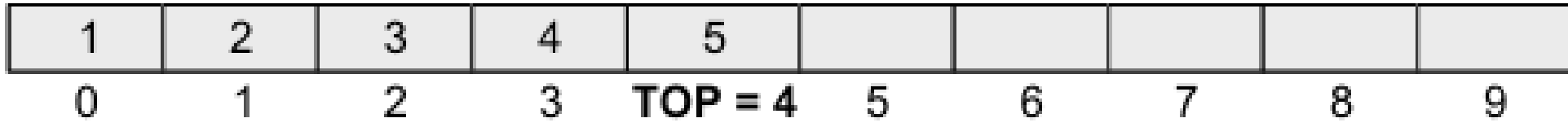
- In the computer's memory, stacks can be represented as a linear array.
- Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack.
- It is this position where the element will be added to or deleted from.
- Another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.

The stack in Fig. shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored

Operations on stack

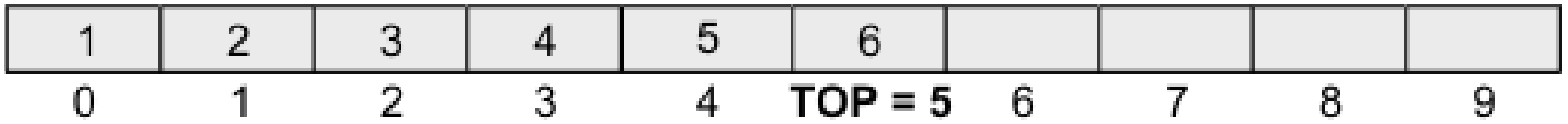
- A stack supports three basic operations: push, pop, and peek.
- Push operation adds an element to the top of the stack
- Pop operation removes the element from the top of the stack
- Peek operation returns the value of the topmost element of the stack

Push Operation



Insertion before Push operation

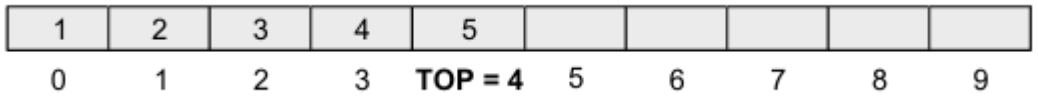
```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```



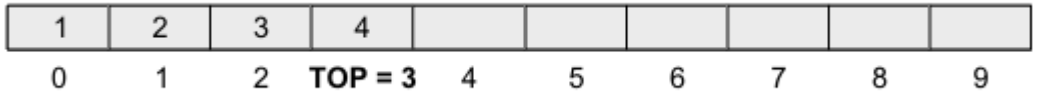
Insertion after Push operation

Pop Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```



Deletion before Pop Operation

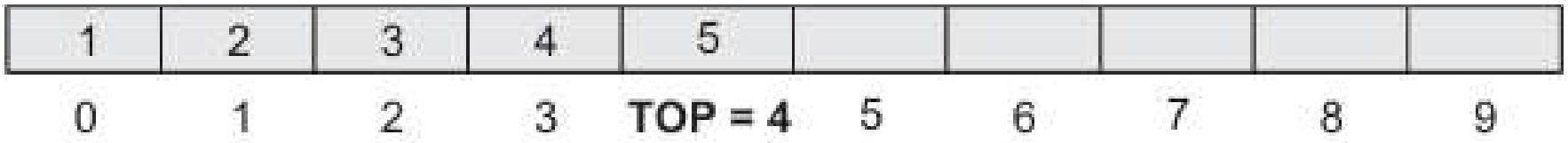


Deletion after Pop Operation

Peek Operation

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.



Application of stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Evaluation of arithmetic expressions :Polish Notation

- Infix to Postfix:
- To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...
- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.
- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

(A + B) * C

= [AB+]*C

= AB+C* (Postfix notation)

Infix to Postfix

1.(A + B) * C

(+AB)*C

*+ABC

2.(A–B) * (C+D)

[–AB] * [+CD]

*–AB+CD

3.(A + B) / (C + D) – (D * E)

[+AB] / [+CD] – [*DE]

[/+AB+CD] – [*DE]–/+AB+CD*DE

Expression: A+B*C/D-E

Step	Symbol Read	Action Taken	Stack	Postfix Expression
1	(	Push (onto stack	(	
2	A	Operand → Add to postfix	(	A
3	+	Operator → Push to stack	(+	A
4	B	Operand → Add to postfix	(+	A B
5	*	Higher precedence than + → Push to stack	(+ *	A B
6	C	Operand → Add to postfix	(+ *	A B C
7	/	Same precedence as * → Pop * → Add to postfix, then push /	(+ /	A B C *
8	D	Operand → Add to postfix	(+ /	A B C * D
9	−	Lower precedence than / → Pop /, then + → Add both to postfix, push −	(−	A B C * D / +
10	E	Operand → Add to postfix	(−	A B C * D / + E
11	)	Pop until (→ Pop − → Add to postfix, discard (		A B C * D / + E −

Step	Symbol Read	Action Taken	Stack	Postfix Expression
1	(	Push (onto stack	(	
2	X	Operand → Add to postfix	(	X
3	−	Operator → Push to stack	(−	X
4	Y	Operand → Add to postfix	(−	X Y
5	/	Higher precedence than − → Push to stack	(− /	X Y
6	(	Push (onto stack	(− / (	X Y
7	Z	Operand → Add to postfix	(− / (	X Y Z
8	+	Operator → Push to stack	(− / (+	X Y Z
9	U	Operand → Add to postfix	(− / (+	X Y Z U
10	)	Pop until (→ Pop + → Add to postfix, discard (	(− /	X Y Z U +
11	*	Same precedence as / → Pop / → Add to postfix, then push *	(− *	X Y Z U + /
12	V	Operand → Add to postfix	(− *	X Y Z U + / V
13	)	Pop until (→ Pop *, then − → Add both to postfix, discard (		X Y Z U + / V * −

Input String	Output Stack	Operator Stack
A+(B*C-(D/E^F)*G)*H		
A+(B*C-(D/E^F)*G)*H	A	
A+(B*C-(D/E^F)*G)*H	A	+
A+(B*C-(D/E^F)*G)*H	A	+(
A+(B*C-(D/E^F)*G)*H	AB	+(
A+(B*C-(D/E^F)*G)*H	AB	+(*
A+(B*C-(D/E^F)*G)*H	ABC	+(*
A+(B*C-(D/E^F)*G)*H	ABC*	+(-
A+(B*C-(D/E^F)*G)*H	ABC*	+(-(
A+(B*C-(D/E^F)*G)*H	ABC*D	+(-(
A+(B*C-(D/E^F)*G)*H	ABC*D	+(-(/
A+(B*C-(D/E^F)*G)*H	ABC*DE	+(-(/
A+(B*C-(D/E^F)*G)*H	ABC*DE	+(-(/^
A+(B*C-(D/E^F)*G)*H	ABC*DEF	+(-(/^
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/	+(-
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/	+(-*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G	+(-*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-	+
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-	+*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-H	+*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-H*+	

Token	Action	Stack	Postfix
A	Operand → Add to postfix	(	A
+	Operator → Push	(+	A
(	Push	(+ (	A
B	Operand → Add to postfix	(+ (	A B
*	Operator → Push	(+ (*	A B
C	Operand → Add to postfix	(+ (*	A B C
-	Operator → Pop * (higher), then push -	(+ (-	A B C *
(	Push	(+ (- (	A B C *
D	Operand → Add to postfix	(+ (- (	A B C * D

/	Operator → Push	(+ (- (/	A B C * D
E	Operand → Add to postfix	(+ (- (/	A B C * D E
^	Operator → Push (higher precedence)	(+ (- (/ ^	A B C * D E
F	Operand → Add to postfix	(+ (- (/ ^	A B C * D E F
)	Pop until (→ Pop ^, then /, discard (	(+ (-	A B C * D E F ^ /
*	Operator → Push	(+ (- *	A B C * D E F ^ /
G	Operand → Add to postfix	(+ (- *	A B C * D E F ^ / G
)	Pop until (→ Pop *, then -, discard (	(+	A B C * D E F ^ / G * -

`*`	Operator → Push	(+ *	A B C * D E F ^ / G * -
H	Operand → Add to postfix	(+ *	A B C * D E F ^ / G * - H
)	Pop until (→ Pop *, then +, discard (	(empty)	A B C * D E F ^ / G * - H * +

Algo:

1. Add) to the end of the infix expression.
2. Push (onto the stack to mark the beginning.
2. Scan each character in the expression:
    - Operand (letter/number) → Add directly to postfix output.
    - Operator (+, -, *, /, ^) →
        - Pop operators from the stack with equal or higher precedence, add them to postfix.
        - Then push the current operator.
    - Left parenthesis (→ Push onto the stack.
    - Right parenthesis) →
    - Pop and add to postfix until a (is found.
    - Discard the (.
3. After scanning, pop all remaining operators from the stack and add to postfix.
4. Done! You now have a postfix expression.

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

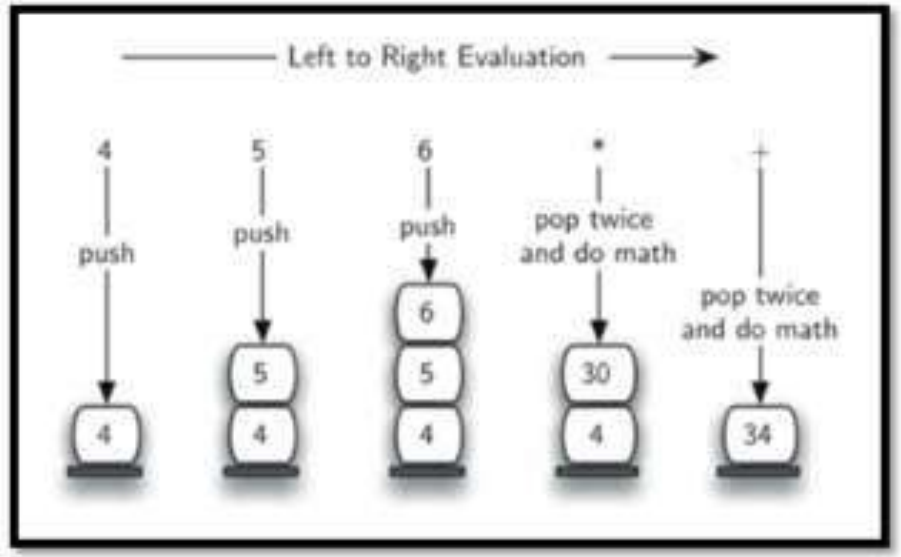
(a) A – (B / C + (D % E * F) / G)* H

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Evaluation of a Postfix Expression

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

6 2 3 + – 3 8 2 / + * 2 ^ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
^	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Practice

Expression: 5 1 2 + 4 * + 3 –

Result: 14

Expression: 7 8 + 3 2 + /

Result: 3

Expression: 2 3 1 * + 9 –

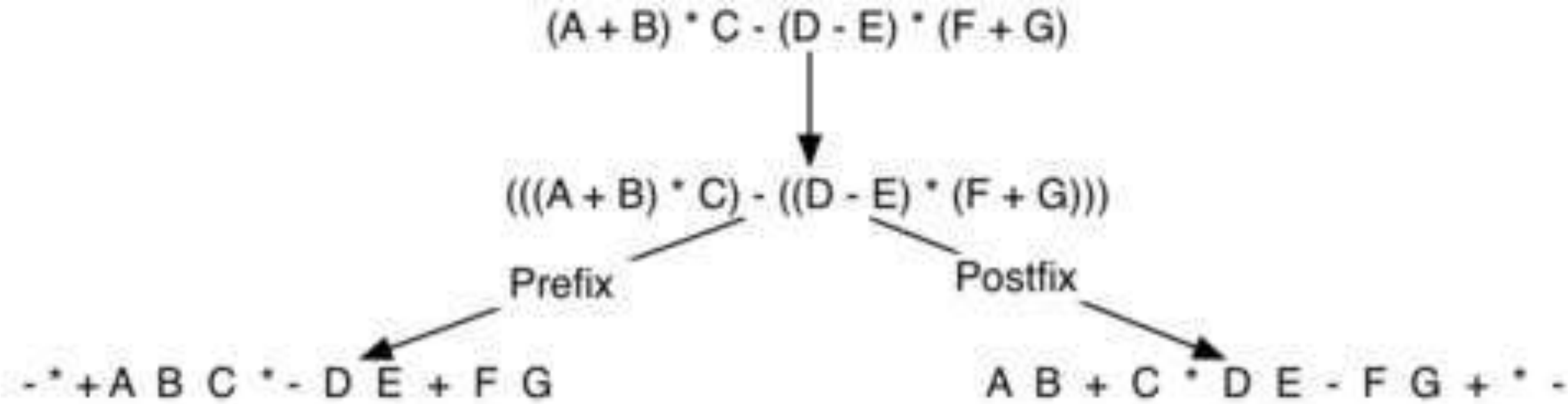
Result: –4

Expression: 4 2 ^ 3 5 1 – * +

Result: 28

Expression: 10 2 8 * + 3 –

Result: 23


(A + B) * C - (D - E) * (F + G)
(((A + B) * C) - ((D - E) * (F + G)))
Prefix
Postfix
- * + A B C * - D E + F G
A B + C * D E - F G + * -

Infix to Prefix

Step 1: Reverse the Infix Expression

- Reverse the entire expression.
- Swap every (with) and vice versa.

Example:

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

Reversed:) E - D ^ C (* B + A

After bracket swap: (E - D ^ C) * B + A

Step 2: Convert the Reversed Expression to Postfix

Use the standard infix to postfix conversion rules:

- Operands go directly to the result.
- Operators go to the stack based on precedence.
- Pop from stack when encountering (or lower precedence.

Step 3: Reverse the Postfix Result

The final step is to reverse the postfix result to get the prefix expression.

Example Continued:

Postfix of reversed infix: E D C ^ - B * A +

Reverse it: + A * B - ^ C D E

Prefix: + A * B - ^ C D E

Mnemonic to Remember:

"Reverse → Postfix → Reverse"

Just think: RPR — Reverse, Postfix, Reverse

Infix Expression : A+B*(C^D-E)				
Reverse Infix expression:)E-D^C(*B+A				
Reverse brackets: (E-D^C)*B+A				
Token	**Action**	**Result**	**Stack**	**Notes**
(	Push **(** to stack		(	
E	Add **E** to the result	E	(	
-	Push **-** to stack	E	(-	
D	Add **D** to the result	ED	(-	
^	Push **^** to stack	ED	(- ^	
C	Add **C** to the result	EDC	(- ^	
)	Pop **^** from stack and add to result	EDC^	(-	Do process until (is popped from stack
	Pop **-** from stack and add to result	EDC^-	(	
	Pop **(** from stack	EDC^-		
*	Push ***** to stack	EDC^-	*	
B	Add **B** to the result	EDC^-B	*	
+	Pop ***** from stack and add to result	EDC^-B		- has lower precedence than ^
	Push **+** to stack	EDC^-B*	+	
A	Add **A** to the result	EDC^-B*A	+	
	Pop **+** from stack and add to result	EDC^-B*A+		Given expression is iterated, do Process till stack is not Empty, It will give the final result
Prefix Expression (Reverse Result): +A*B-^CDE				

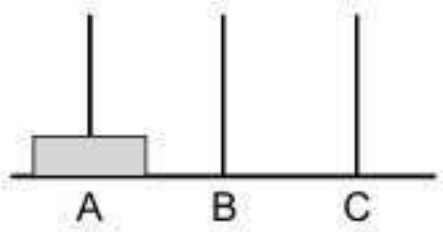
Infix to prefix

Infix Expression	Prefix Expression
A +B-C	-+ABC
(A+B) *(C+D)	*+AB+CD
A/B*C-D+E/F/(G+H)	+-*/ABCD//EF+GH
((A + B) * C - (D-E))*(F+G)	*-*+ABC-DE +FG
A-B/(C *D/E)	-A/B/*CDE

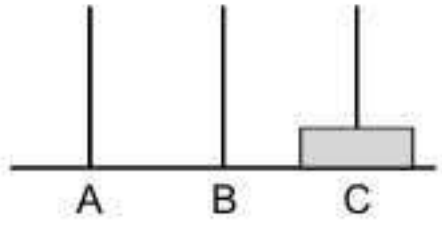
Recursion: Tower of hanoi

- Only one disk can be moved at a time
- Only the disk at the top of a stack can be moved
- A disk cannot be placed on top of another disk with smaller diameter

```
#include <stdio.h>
int main()
{
printf("\n Enter the number of rings: ");
scanf("%d", &n);
move(n,'A', 'C', 'B');
return 0;
}
void move(int n, char source, char dest, char spare)
{
if (n==1)
printf("\n Move from %c to %c",source,dest);
else
{
}
}
move(n–1,source,spare,dest);
move(1,source,dest,spare);
move(n–1,spare,dest,source);
```

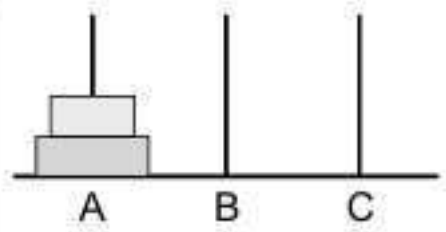


(Step 1)

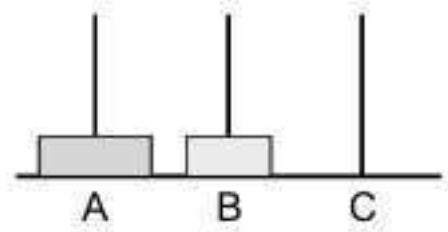


(Step 2)

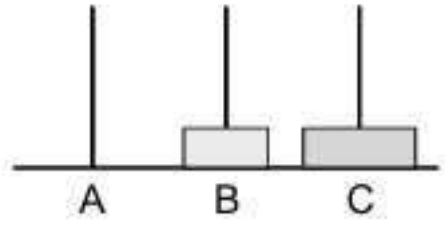
(If there is only one ring, then simply move the ring from source to the destination.)



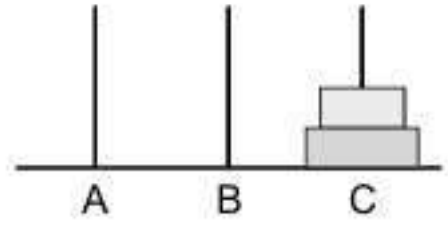
(Step 1)



(Step 2)

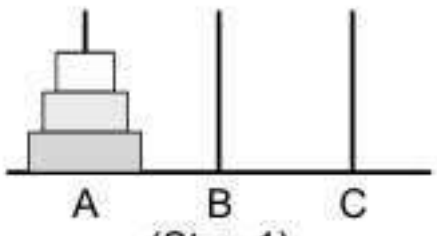


(Step 3)

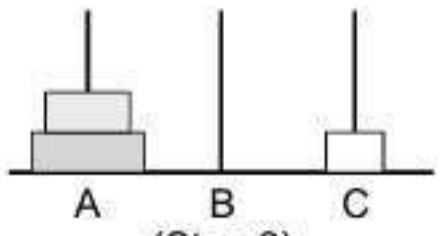


(Step 4)

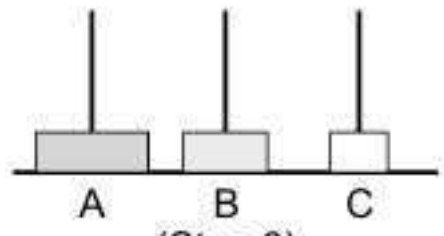
(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)



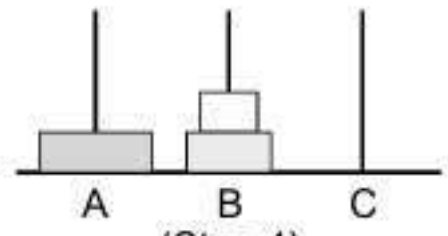
(Step 1)



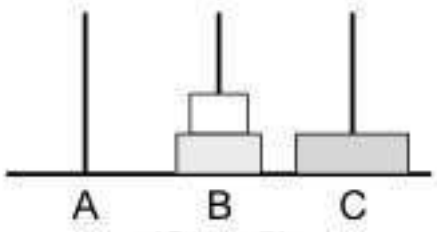
(Step 2)



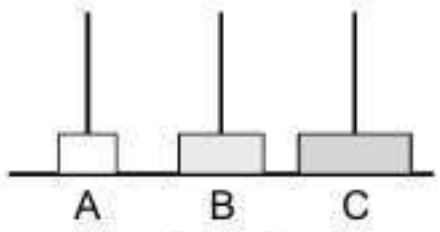
(Step 3)



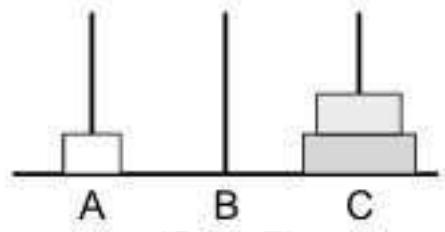
(Step 4)



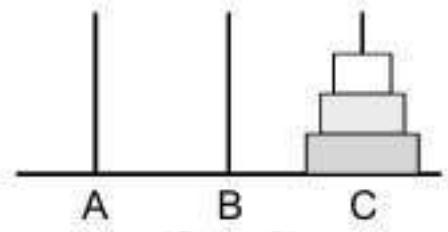
(Step 5)



(Step 6)



(Step 7)



(Step 8)

(Consider the working with three rings.)