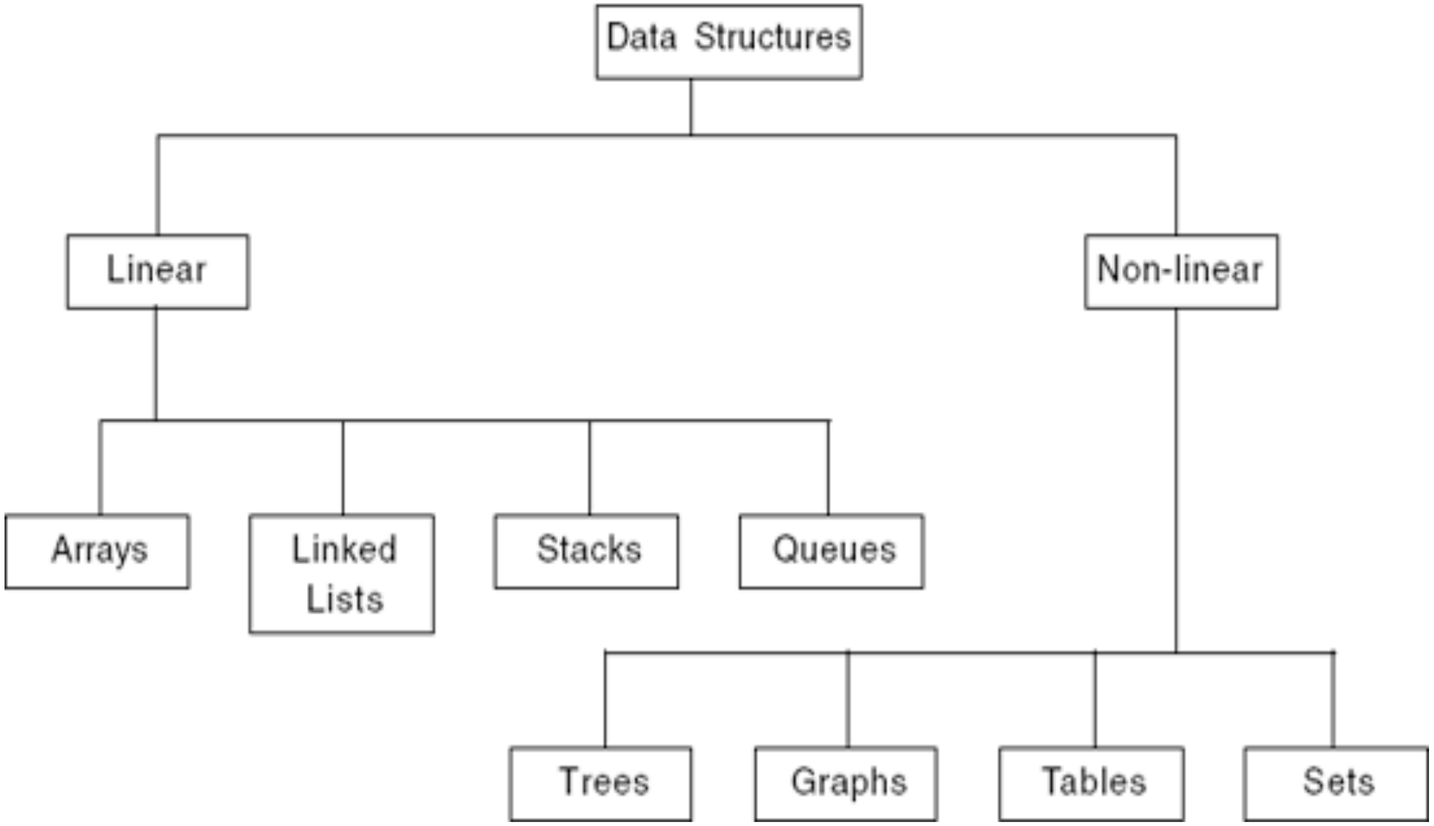
Classification

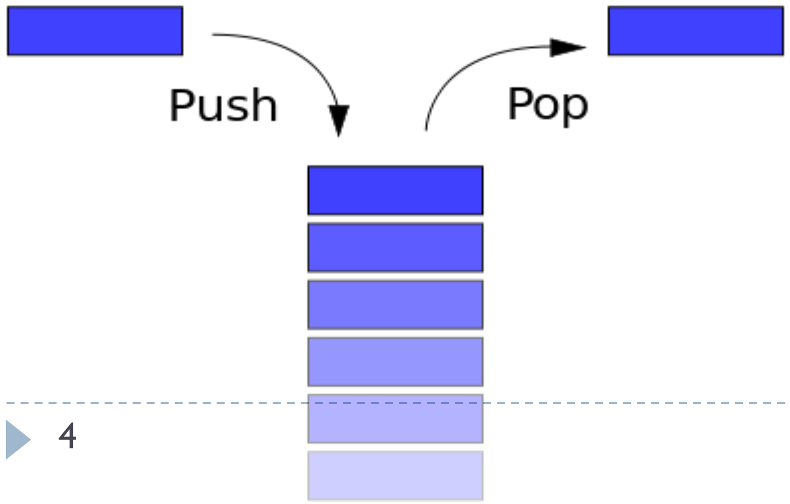


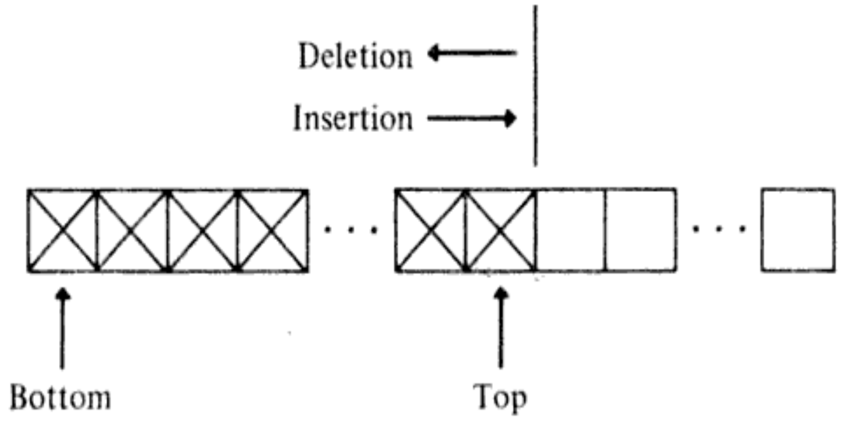
- Stacks
  - Allow insertions and removals only at top of stack
- Queues
  - Allow insertions at the back and removals from the front
- Linked lists
  - Allow insertions and removals anywhere
- Binary trees
  - High-speed searching and sorting of data and efficient elimination of duplicate data items

Stacks

Introduction

- Stack is an important data structure which stores its elements in an ordered manner.
- A stack is a linear data structure which uses the principle, i.e., the elements in a stack are added and removed only from one end, which is called the *top*.
- Hence, a stack is called a LIFO (Last-In, First-Out) data structure as the element that is inserted last is the first one to be taken out.





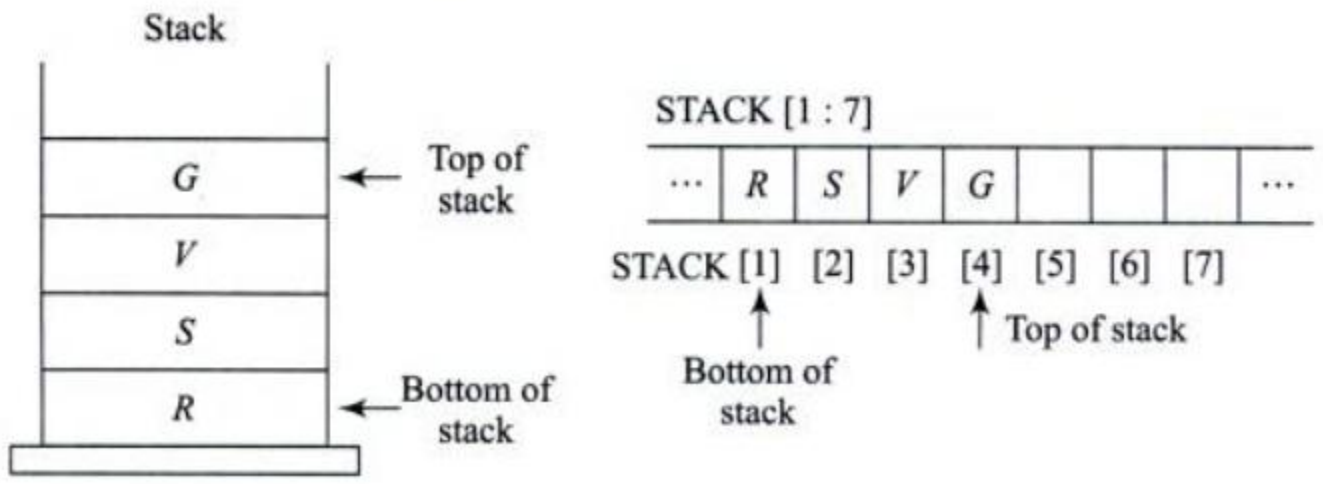
Introduction

- Real life examples of stack:
  - Suppose we have created stack of the book
  - How books are arranged in the stack?
    - Books are kept one above the other
    - Books which are inserted first is taken out last. (brown)
    - Book which is inserted lastly is served first. (light green)
  - Suppose at your home you have multiple chairs then you put them together to form a vertical pile. From that vertical pile, the chair which is placed last is always removed first.
  - Chair which was placed first will be removed last.

Array representation of stacks

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it.
- TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- There is another variable MAX which will be 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.

Array representation of stacks

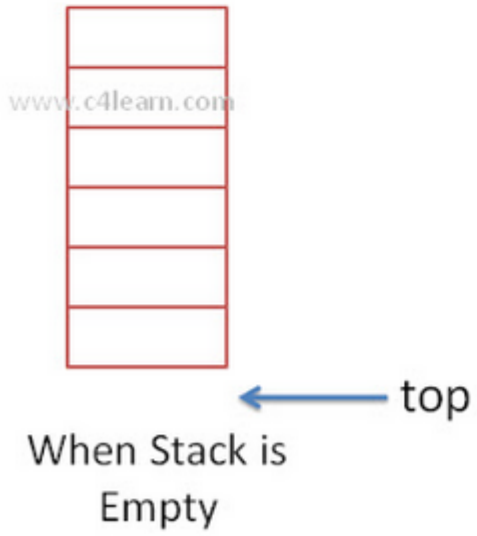


Operations of Stack

- Push : inserting element onto stack.
- Pop : removing element from stack.
- Peep : returns the ith element from top element of the stack.
- Change : changes the ith element from top of stack to the mentioned element.

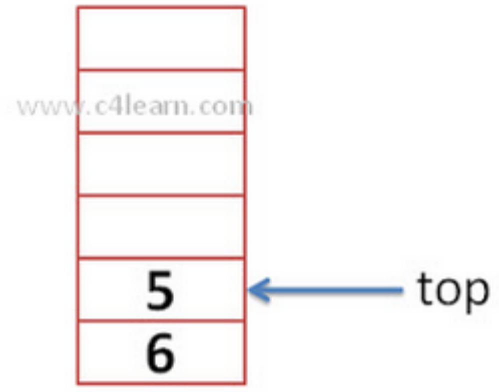
Visual Representation of Stack

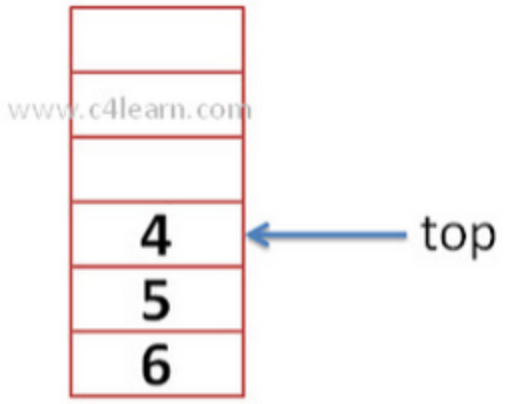
- View 1: When stack is empty
  - When stack is empty then it does not contain any element inside it. Whenever stack is empty, the position of topmost element is -1.



Visual Representation of Stack

- View 2: When stack is not empty
  - Whenever we add very first element then topmost position will be increment by 1. After adding first element, top = 0.





- View 3: After deletion of 1 element top will be decremented by 1.

Visual Representation of Stack

- Position of top and its value:

Position of Top	Status of Stack
-1	Stack is Empty
0	First Element is Just Added into Stack
N-1	Stack is said to Full
N	Stack is said to be Overflow

- Values of stack and top:

Operation	Explanation
top = -1	-1 indicated Empty Stack
top = top + 1	After push operation value of top is incremented by integer 1
top = top – 1	After pop operation value of top is decremented by 1

Push Operation

- Procedure PUSH(S, TOP, X): This procedure inserts an element X on the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

```
Algorithm to PUSH an element in a stack

Step 1: IF TOP = N-1, then
             PRINT "OVERFLOW"
             Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET S[TOP] = X
Step 4: END
```

Pop Operation

- Procedure POP(S, TOP): This procedure removes the element from the stack which is represented by a vector S and returns the element.

```
Algorithm to POP an element from a stack

Step 1: IF TOP = -1, then
             PRINT "UNDERFLOW"
             Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP - 1
Step 3: Return S(TOP+1)
Step 4: END
```

Peek Operation

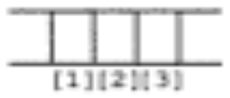
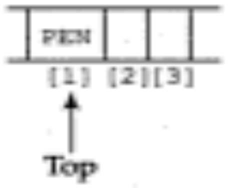
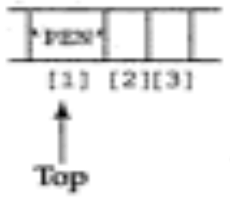
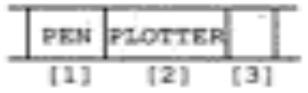
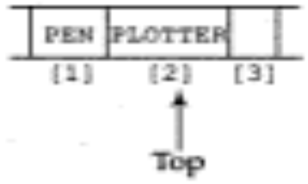
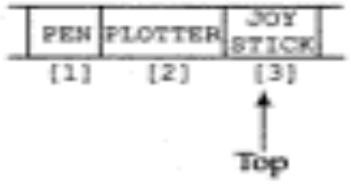
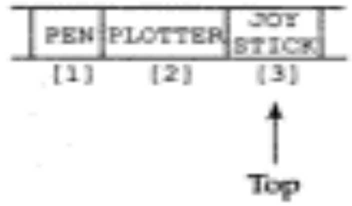
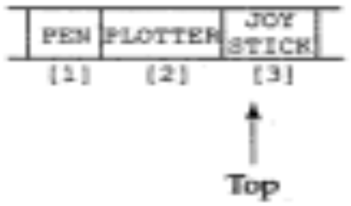
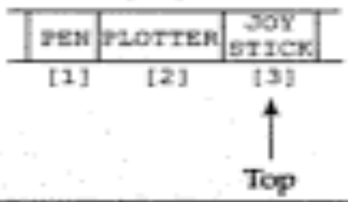
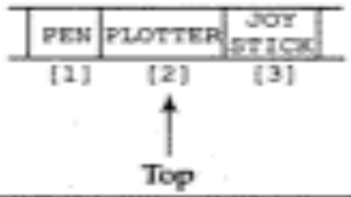
- Procedure PEEP(S, TOP, I): Given the vector S (consisting of N elements) representing a sequentially allocated stack, and a variable TOP denoting the top element of the stack, this function returns the value of the ith element from the top of the stack. The element is not deleted by this function.

```
Algorithm to PEEP an element from a stack

Step 1: IF TOP – I < 0, then
           PRINT "UNDERFLOW"
           Goto Step 3
        [END OF IF]
Step 2: Return S(TOP – I)
Step 3: END
```

Change Operation

- Procedure CHANGE(S, TOP, X, I): Given the vector S (consisting of N elements) representing a sequentially allocated stack, and a pointer TOP denoting the top element of the stack, this procedure changes the value of the ith element from the top of the stack to the value contained in X.

```
Algorithm to CHANGE an element from a stack

Step 1: IF TOP – I < 0, then
           PRINT "UNDERFLOW"
           Goto Step 3
        [END OF IF]
Step 2: S(TOP – I) = X
Step 3: END
```

Stack operation	Stack before operation	Algorithm invocation	Stack after operation	Remarks
1. Push 'PEN' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3,0,'PEN')	DEVICE[1:3] 	Push 'PEN' Successful
2. Push 'PLOTTER' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE,3, 1, 'PLOTTER')	DEVICE[1:3] 	Push 'PLOTTER' successful
3. Push 'JOY STICK' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 2, 'JOY STICK')	DEVICE[1:3] 	Push 'JOY STICK' successful
4. Push 'PRINTER' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 3, 'PRINTER')	DEVICE[1:3] 	Push 'PRINTER' failure! STACK-FULL condition invoked
5. Pop from DEVICE[1:3]	DEVICE[1:3] 	POP(DEVICE, 3, ITEM)	DEVICE[1:3] 	ITEM = 'JOY STICK' Pop operation successful

Applications of Stack

- Expression conversion
- Expression evaluation
- Recursion

Expression conversion

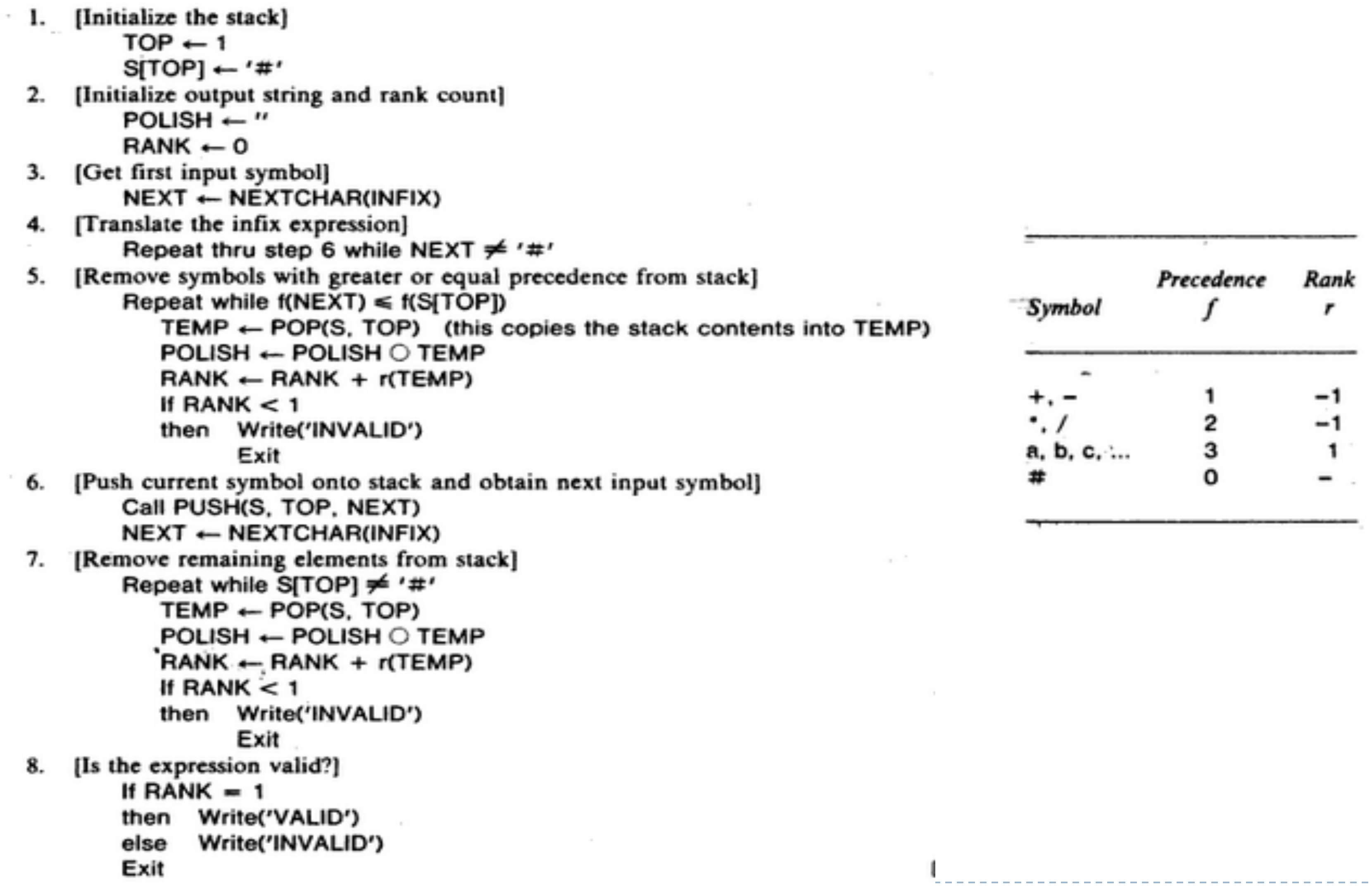
- Conversion from infix to postfix expression
- Conversion from infix to prefix expression
- Conversion from prefix to infix expression
- Conversion from prefix to postfix expression
- Conversion from postfix to prefix expression
- Conversion from postfix to infix expression

Expressions

- Expressions is a string of operands and operators. Operands are some numeric values and operators are of two types: Unary and binary operators. Unary operators are '+' and '-' and binary operators are '+', '-', '*', '/' and exponential. In general, there are three types of expressions:
  - Infix expression : operand1 operator operand2
  - Postfix expression : operand1 operand2 operator
  - Prefix expression : operator operand1 operand2

Infix	Postfix	Prefix
(a + b)	ab +	+ ab
(a + b) * (c - d)	ab + cd - *	* + ab - cd
(a + b / e) * (d + f)	abe /+ df + *	* + a/be + df

Conversion from infix to postfix (without parenthesis)



```
1.  [Initialize the stack]
        TOP ← 1
        S[TOP] ← '#'
2.  [Initialize output string and rank count]
        POLISH ← ''
        RANK ← 0
3.  [Get first input symbol]
        NEXT ← NEXTCHAR(INFIX)
4.  [Translate the infix expression]
        Repeat thru step 6 while NEXT ≠ '#'
5.  [Remove symbols with greater or equal precedence from stack]
        Repeat while f(NEXT) ≤ f(S[TOP])
            TEMP ← POP(S, TOP)   (this copies the stack contents into TEMP)
            POLISH ← POLISH ○ TEMP
            RANK ← RANK + r(TEMP)
            If RANK < 1
            then   Write('INVALID')
                   Exit
6.  [Push current symbol onto stack and obtain next input symbol]
        Call PUSH(S, TOP, NEXT)
        NEXT ← NEXTCHAR(INFIX)
7.  [Remove remaining elements from stack]
        Repeat while S[TOP] ≠ '#'
            TEMP ← POP(S, TOP)
            POLISH ← POLISH ○ TEMP
            RANK ← RANK + r(TEMP)
            If RANK < 1
            then   Write('INVALID')
                   Exit
8.  [Is the expression valid?]
        If RANK = 1
        then   Write('VALID')
        else   Write('INVALID')
        Exit
```

Symbol	*Precedence* *f*	*Rank* *r*
+, −	1	−1
*, /	2	−1
a, b, c, ...	3	1
#	0	−

Conversion from infix to postfix (with parenthesis)

```
1.  [Initialize stack]
        TOP ← 1
        S[TOP] ← '('
2.  [Initialize output string and rank count]
        POLISH ← ''
        RANK ← 0
3.  [Get first input symbol]
        NEXT ← NEXTCHAR(INFIX)
4.  [Translate the infix expression]
        Repeat thru step 7 while NEXT ≠ ''
5.  [Remove symbols with greater precedence from stack]
        If TOP < 1
        then    Write('INVALID')
                Exit
        Repeat while f(NEXT) < g(S[TOP])
            TEMP ← POP(S, TOP)
            POLISH ← POLISH ○ TEMP
            RANK ← RANK + r(TEMP)
            If RANK < 1
            then    Write('INVALID')
                    Exit
6.  [Are there matching parentheses?]
        If f(NEXT) ≠ g(S[TOP])
        then    Call PUSH(S, TOP, NEXT)
        else    POP(S, TOP)
7.  [Get next input symbol]
        NEXT ← NEXTCHAR(INFIX)
8.  [Is the expression valid?]
        If TOP ≠ 0 or RANK ≠ 1
        then    Write('INVALID')
        else    Write('VALID')
        Exit
```

Precedence

Symbol	*Input Precedence Function* f	*Stack Precedence Function* g	*Rank Function* r
+, −	1	2	−1
*, /	3	4	−1
↑	6	5	−1
variables	7	8	1
(	9	0	−
)	0	−	−

Conversion from infix to prefix (with parenthesis)

- Step 1: Reverse the infix expression and convert '(' to ')' and ')' to '('.
- Step 2: Read this reversed expression from left to right one character at a time.
- Step 3: Rest of the steps remains same as in case of "conversion from infix to postfix (with parenthesis)".
- Step 4: After all the elements are popped, reverse the expression obtained in prefix expression.

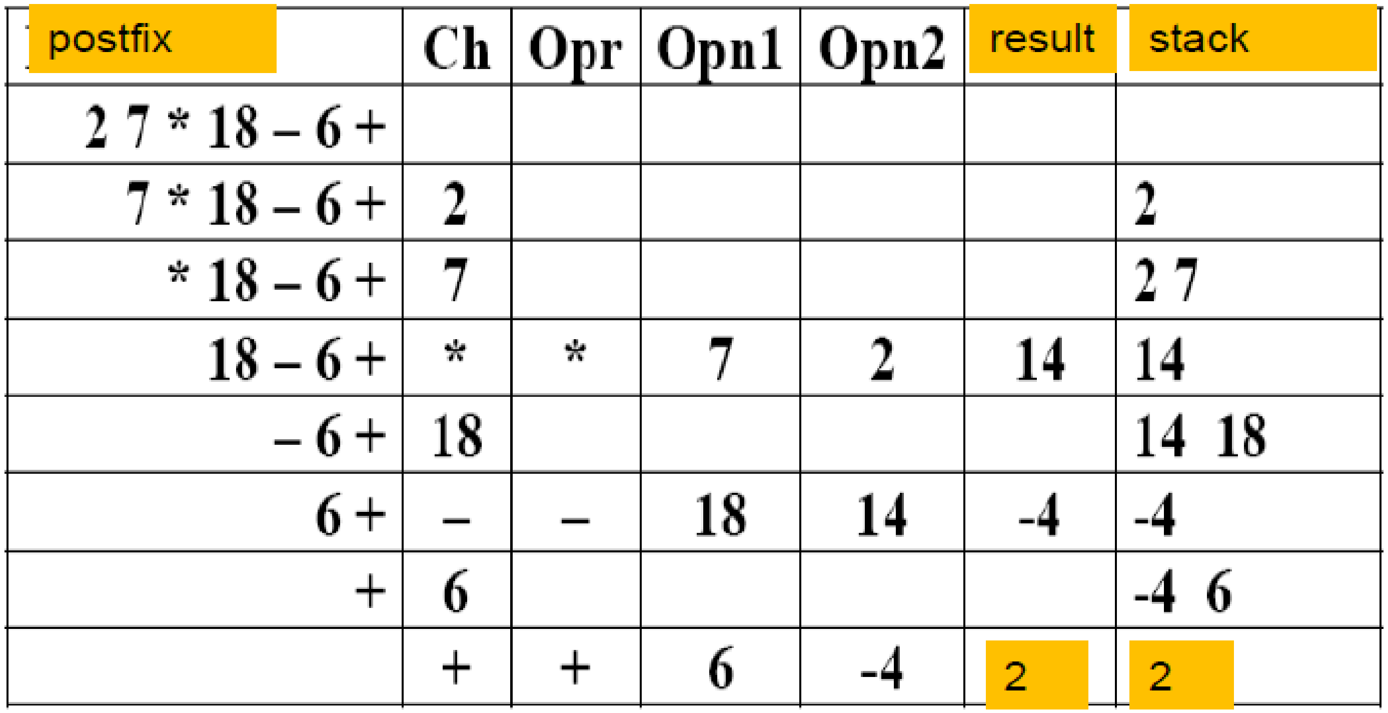
Expression Evaluation

- Evaluation of postfix expression
- Evaluation of prefix expression
- Evaluation of infix expression

Evaluation of postfix expression

- Step 1: If char read from postfix expression is an operand, push operand to stack.
- Step 2: If char read from postfix expression is an operator, pop the first 2 operand in stack and implement the expression using the following operations:
    - **pop(opr1) then pop(opr2)**
    - **result = opr2 operator opr1**
- Step 3: Push the result of the evaluation to stack.
- Step 4: Repeat steps 1 to steps 3 until end of postfix expression
- Finally, At the end of the operation, only one value left in the stack. The value is the result of postfix evaluation.

Evaluation of postfix expression

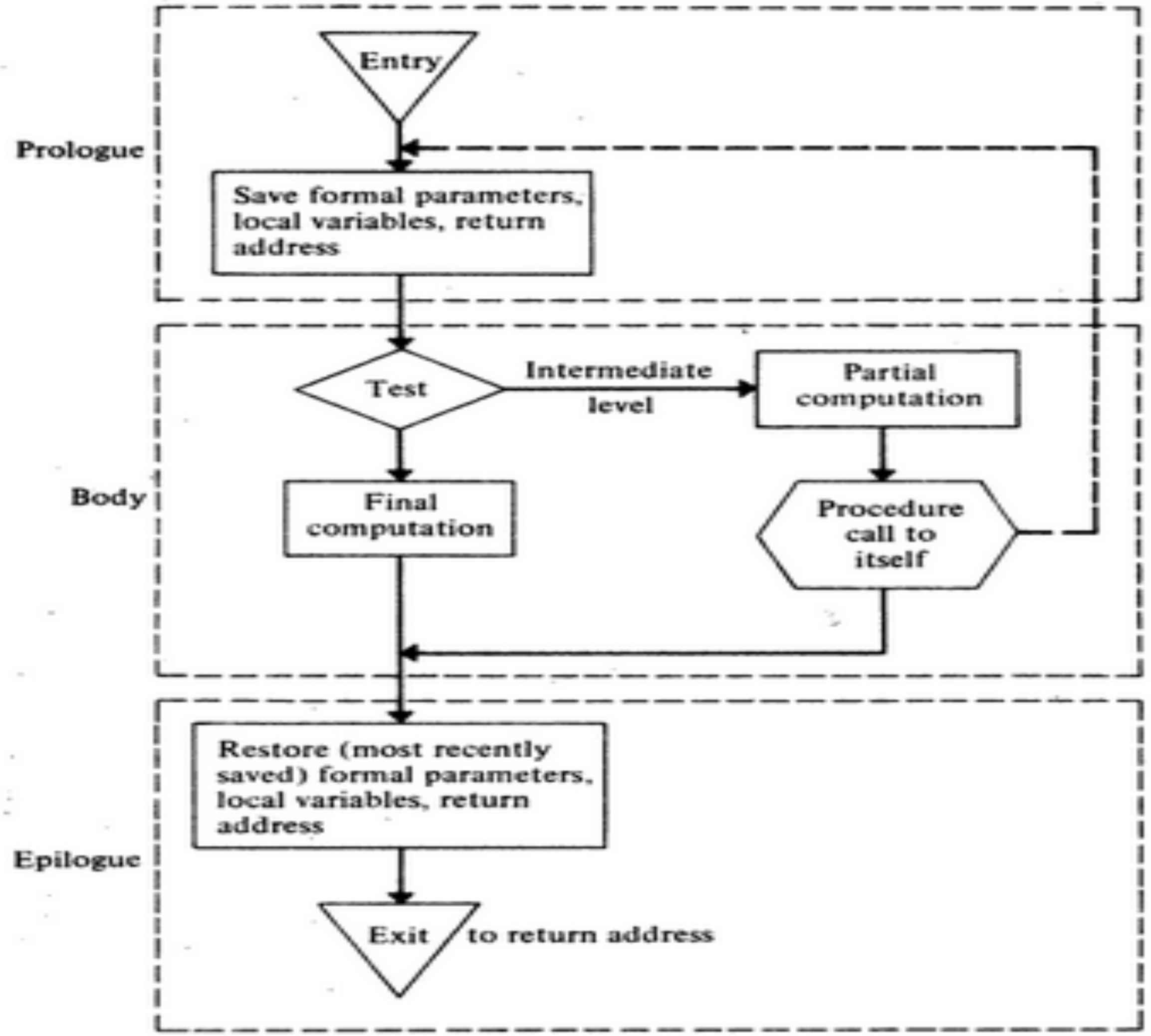


postfix	Ch	Opr	Opn1	Opn2	result	stack
2 7 * 18 – 6 +						
7 * 18 – 6 +	2					2
* 18 – 6 +	7					2 7
18 – 6 +	*	*	7	2	14	14
– 6 +	18					14 18
6 +	–	–	18	14	-4	-4
+	6					-4 6
	+	+	6	-4	2	2

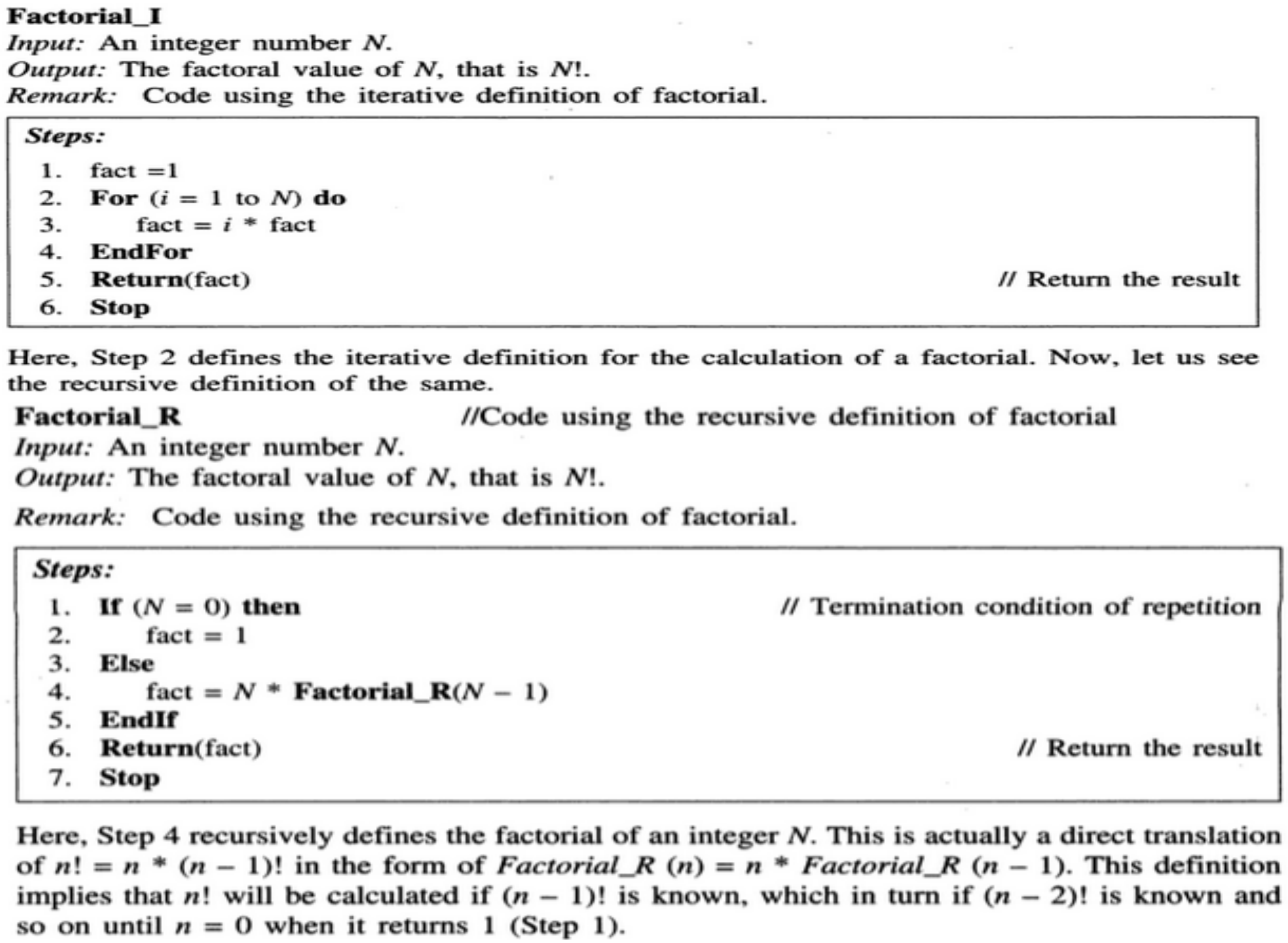
Evaluation of prefix expression

- Step 1: Reverse the prefix expression.
- Step 2: Read this reversed prefix expression from left to right one character at a time.
- Step 3: If char read from reversed prefix expression is an operand, push operand to stack.
- Step 4: If char read from reversed prefix expression is an operator, pop the first 2 operand in stack and implement the expression using the following operations:
  - **pop(opr1) then pop(opr2)**
  - **result = opr1 operator opr2**
- Step 5: Push the result of the evaluation to stack.
- Step 6: Repeat steps 3 to steps 5 until end of reversed prefix expression
- Finally, At the end of the operation, only one value left in the stack. The value is the result of prefix evaluation.

Recursion


Prologue
Entry
Save formal parameters, local variables, return address
Body
Test
Intermediate level
Partial computation
Final computation
Procedure call to itself
Epilogue
Restore (most recently saved) formal parameters, local variables, return address
Exit
to return address

Recursion



Factorial_I
Input: An integer number N.
Output: The factoral value of N, that is $N!$.
Remark: Code using the iterative definition of factorial.

```
Steps:
1.  fact =1
2.  For (i = 1 to N) do
3.      fact = i * fact
4.  EndFor
5.  Return(fact)                                   // Return the result
6.  Stop
```

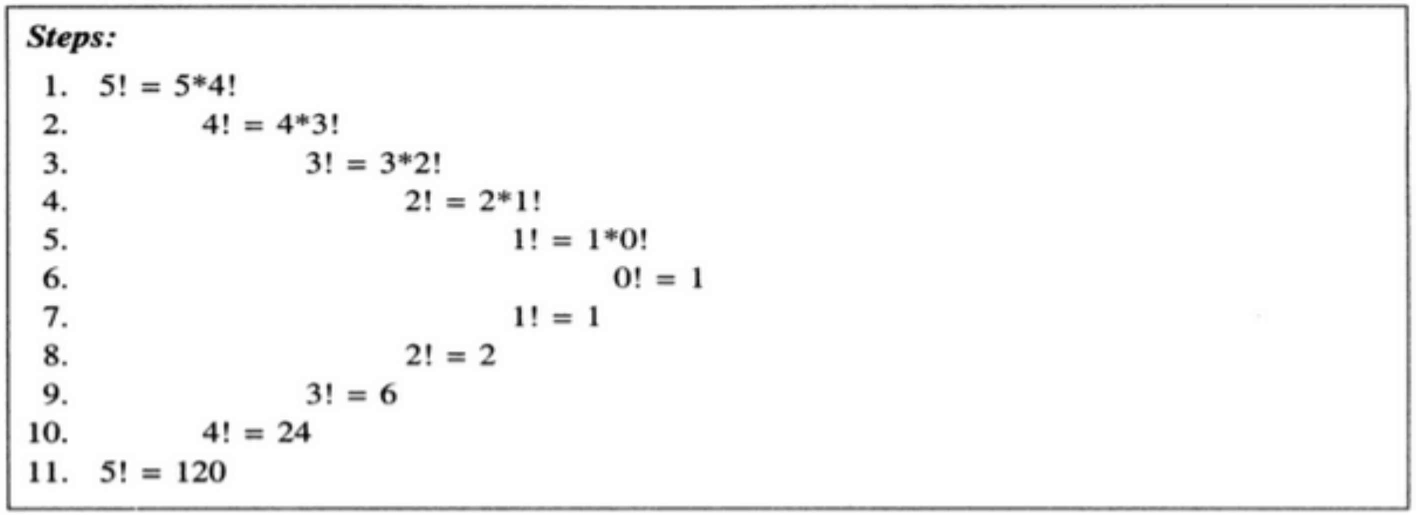
Here, Step 2 defines the iterative definition for the calculation of a factorial. Now, let us see the recursive definition of the same.

Factorial_R //Code using the recursive definition of factorial
Input: An integer number N.
Output: The factoral value of N, that is $N!$.
Remark: Code using the recursive definition of factorial.

```
Steps:
1.  If (N = 0) then                  // Termination condition of repetition
2.      fact = 1
3.  Else
4.      fact = N * Factorial_R(N – 1)
5.  EndIf
6.  Return(fact)                                   // Return the result
7.  Stop
```

Here, Step 4 recursively defines the factorial of an integer N. This is actually a direct translation of $n! = n * (n - 1)!$ in the form of *Factorial_R* $(n) = n$ * *Factorial_R* $(n - 1)$. This definition implies that $n!$ will be calculated if $(n - 1)!$ is known, which in turn if $(n - 2)!$ is known and so on until $n = 0$ when it returns 1 (Step 1).

Recursion

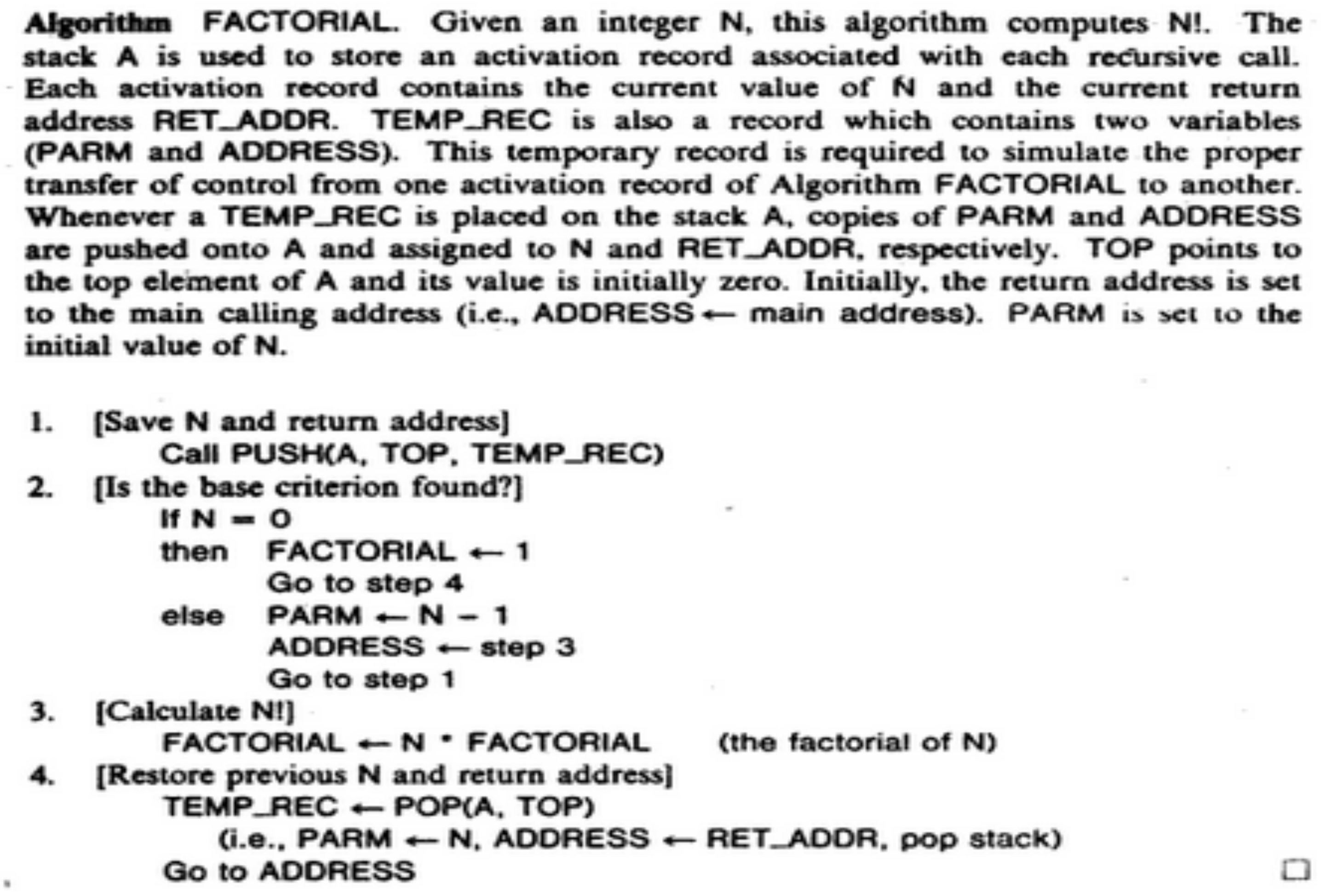


Steps:

```
 1.  5! = 5*4!
 2.          4! = 4*3!
 3.                   3! = 3*2!
 4.                            2! = 2*1!
 5.                                     1! = 1*0!
 6.                                              0! = 1
 7.                                     1! = 1
 8.                            2! = 2
 9.                   3! = 6
10.          4! = 24
11.  5! = 120
```

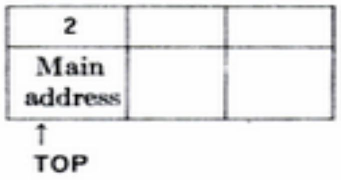
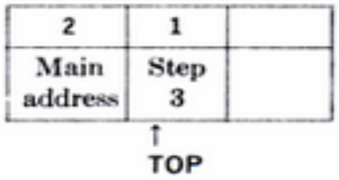
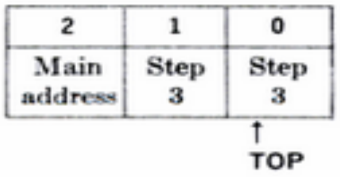
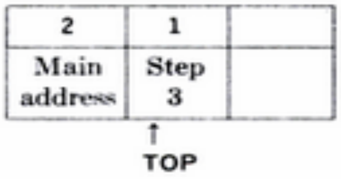
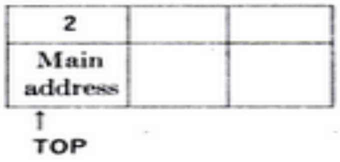
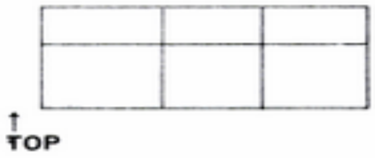
- Here, it is required to push the intermediate calculations till the terminal condition is reached. In the above calculation for 5!, Steps 1 to 6 are the push operations. The subsequent pop operations will evaluate the value of intermediate calculations till the stack is exhausted.
- Stack for parameter(s): To store the parameter with which the recursion is defined.
- Stack for local variable(s): To hold the local variable that are used within the definition.
- Stack to store the return address.

Factorial with recursion using stack



Algorithm FACTORIAL. Given an integer N, this algorithm computes N!. The stack A is used to store an activation record associated with each recursive call. Each activation record contains the current value of N and the current return address RET_ADDR. TEMP_REC is also a record which contains two variables (PARM and ADDRESS). This temporary record is required to simulate the proper transfer of control from one activation record of Algorithm FACTORIAL to another. Whenever a TEMP_REC is placed on the stack A, copies of PARM and ADDRESS are pushed onto A and assigned to N and RET_ADDR, respectively. TOP points to the top element of A and its value is initially zero. Initially, the return address is set to the main calling address (i.e., ADDRESS ← main address). PARM is set to the initial value of N.

```
1.  [Save N and return address]
        Call PUSH(A, TOP, TEMP_REC)
2.  [Is the base criterion found?]
        If N = 0
        then    FACTORIAL ← 1
                Go to step 4
        else    PARM ← N - 1
                ADDRESS ← step 3
                Go to step 1
3.  [Calculate N!]
        FACTORIAL ← N * FACTORIAL        (the factorial of N)
4.  [Restore previous N and return address]
        TEMP_REC ← POP(A, TOP)
            (i.e., PARM ← N, ADDRESS ← RET_ADDR, pop stack)
        Go to ADDRESS                                          □
```

Level Number	*Description*	*Stack 'A' Contents*
Enter level 1 (main call)	Step 1: PUSH(A,0,(2,main address)) Step 2: N ≠ 0 PARM ← 1, ADDR ← Step 3	
Enter level 2 (first recursive call)	Step 1: PUSH(A,1,(1,Step 3)) Step 2: N ≠ 0 PARM ← 0, ADDR ← Step 3	
Enter level 3 (second recursive call)	Step 1: PUSH(A,2,(0,Step 3)) Step 2: N = 0 FACTORIAL ← 1	
	Step 4: POP(A,3), go to Step 3	
Return to level 2	Step 3: FACTORIAL ← 1 * 1 Step 4: POP(A,2), go to Step 3	
Return to level 1	Step 3: FACTORIAL ← 2 * 1 Step 4: POP(A,1), go to main address	

Tower of Hanoi Problem

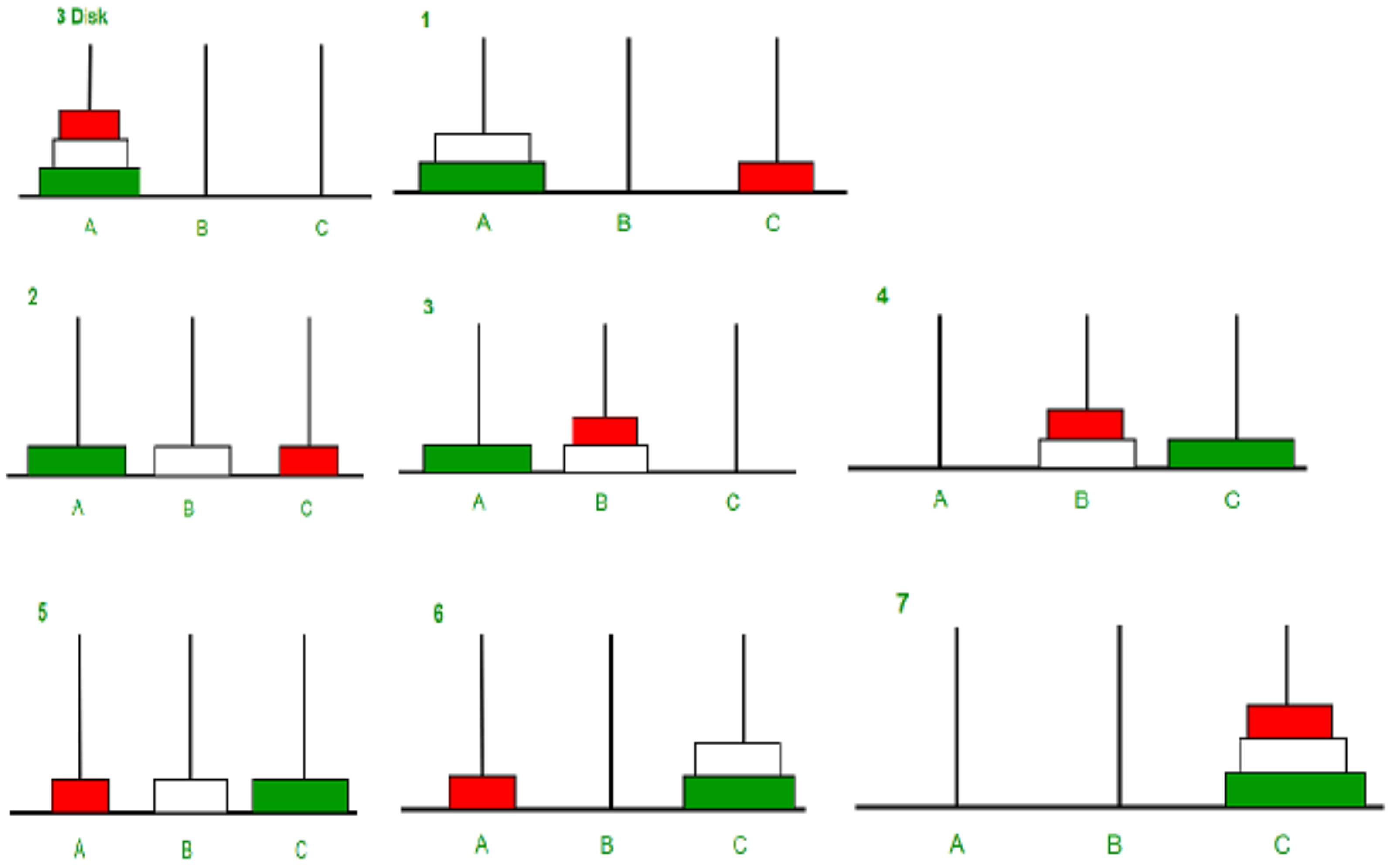
- The Tower of Hanoi puzzle is solved by moving all the disks to another tower by not violating the sequence of the arrangements.

The rules to be followed by the Tower of Hanoi are -

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

- **Algorithm**
- Step 1 – Move n-1 disks from source to aux
- Step 2 – Move nth disk from source to dest
- Step 3 – Move n-1 disks from aux to dest

Example



THANK YOU!!

ANY QUESTIONS??