

UNIT- III

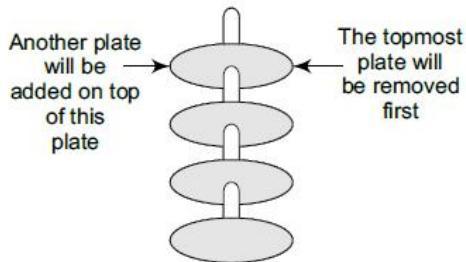
Stacks: Introduction to stacks: properties and operations, implementing stacks using arrays and linked lists, Applications of stacks: Arthematic expression conversions, expression evaluation, backtracking, reversing list and factorial calculation.

INTRODUCTION TO STACKS

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig.

Now, when you want to remove a plate, you remove the top most plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the top most position.



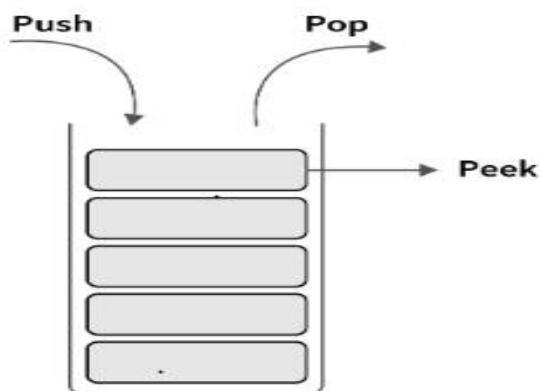
A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.

PROPERTIES AND OPERATIONS ON A STACK

There are following important properties of stack-

- ✓ All insertions and deletions can occur only at the top of stack.
- ✓ The elements that are removed from stack are in reverse order in which they were inserted.
- ✓ Only one element can be pushed or popped from the stack at a time.
- ✓ It works in last-in-first-out or LIFO manner.

A stack supports three basic operations: push, pop, and peek.



The **push** operation adds an element to the top of the stack and the **pop** operation removes the element from the top of the stack. The **peek** operation returns the value of the topmost element of the stack.

Push Operation

- ✓ The push operation is used to insert an element into the stack.
- ✓ The new element is added at the topmost position of the stack.
- ✓ To insert an element with value 6, we first check if $\text{TOP} = \text{MAX} - 1$.
- ✓ If the condition is false, then we increment the value of TOP and store the new element at the position given by $\text{stack}[\text{TOP}]$.

1	2	3	4	5						
0	1	2	3	4	TOP = 4	5	6	7	8	9

```

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

```

1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9

Pop Operation

- ✓ The pop operation is used to delete the topmost element from the stack.
- ✓ However, before deleting the value, we must first check if $\text{TOP} = \text{NULL}$ because if that is the case, then it means the stack is empty and no more deletions can be done.
- ✓ To delete the topmost element, we first check if $\text{TOP} = \text{NULL}$. If the condition is false, then we decrement the value pointed by TOP .

1	2	3	4						
0	1	2	TOP = 3	4	5	6	7	8	9

```

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

```

Peek Operation

- ✓ Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- ✓ However, the Peek operation first checks if the stack is empty, i.e., if $\text{TOP} = \text{NULL}$, then an appropriate message is printed, else the value is returned.
- ✓ Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

```

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

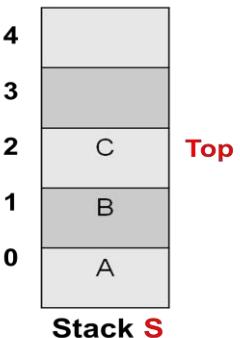
```

1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

ARRAY REPRESENTATION OF STACKS

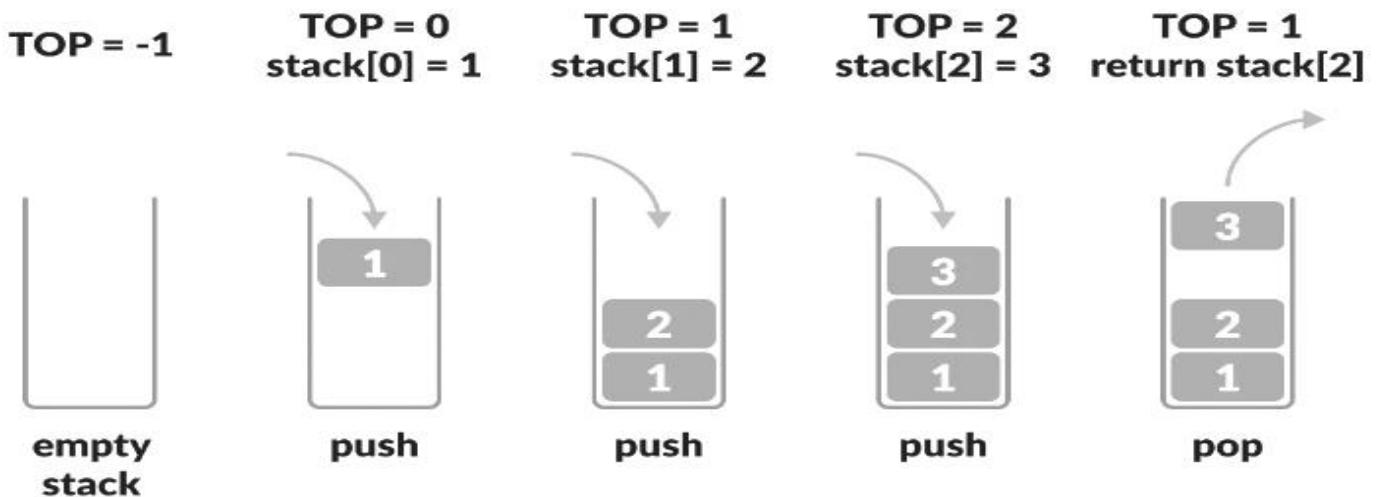
In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.

There is 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 operations work as follows:

- ✓ A pointer called TOP is used to keep track of the top element in the stack.
- ✓ When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
- ✓ On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
- ✓ On popping an element, we return the element pointed to by TOP and reduce its value.
- ✓ Before pushing, we check if the stack is already full
- ✓ Before popping, we check if the stack is already empty

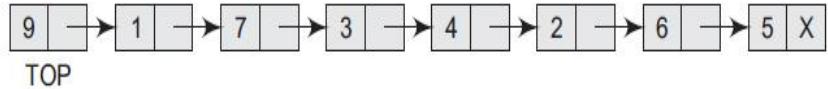


LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty. The linked representation of a stack is shown in below figure.

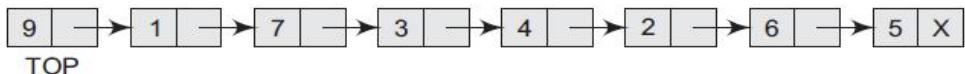


Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in below figure.



To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step

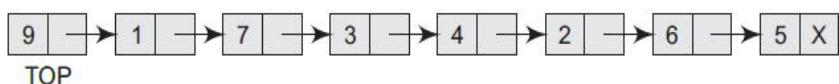
2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

```

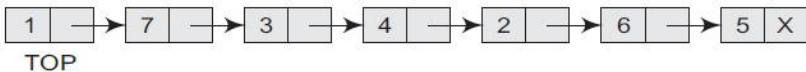
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
    
```

Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in below figure.



In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in below figure.



```

Step 1: IF TOP = NULL
        PRINT "UNDERFLOW"
        Goto Step 5
    [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
  
```

The algorithm to delete an element from a stack.

In Step 1, we first check for the UNDERFLOW condition.

In Step 2, we use a pointer PTR that points to TOP.

In Step 3, TOP is made to point to the next node in sequence.

In Step 4, the memory occupied by PTR is given back to the Free pool.

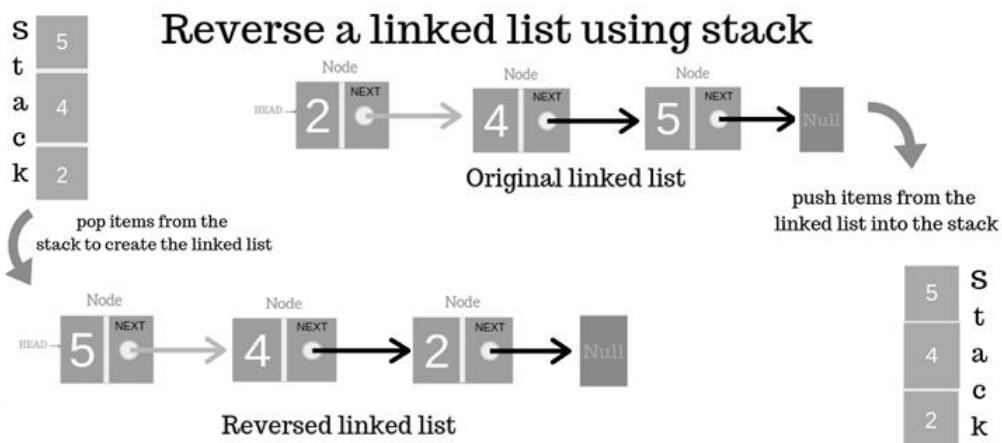
APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- ✓ Reversing a list
- ✓ 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

REVERSING LIST

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.



Evaluation of Arithmetic Expressions

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.

While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here, plus operator is placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer

needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

In postfix notation(also known as Polish notation), as the name suggests, the operator is placed after the operands. For example, if an expression is written as $A+B$ in infix notation, the same expression can be written as $AB+$ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

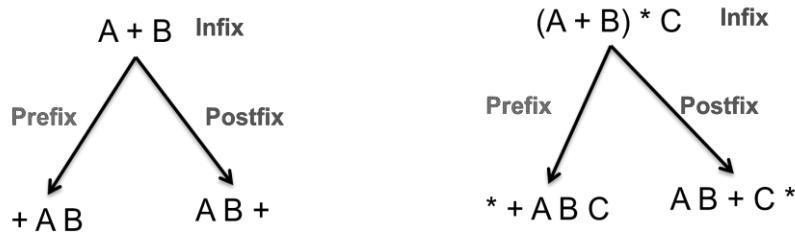
The expression $(A + B) * C$ can be written as:

$[AB+] * C$

$AB+C*$ in the postfix notation

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.

For example, given a postfix notation $AB+C*$. While evaluation, addition will be performed prior to multiplication. Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, $AB+C*$, $+$ is applied on A and B , then $*$ is applied on the result of addition and C .



Conversion of an Infix Expression into a Postfix Expression:

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only $+, -, *, /, \%$ operators.

The precedence of these operators can be given as follows:

- ✓ Higher priority $*, /, \%$
- ✓ Lower priority $+, -$

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression $A + B * C$, then first $B * C$ will be done and the result will be added to A . But the same expression if written as, $(A + B) * C$, will evaluate $A + B$ first and then the result will be multiplied with C .

Example: Convert the following infix expressions into postfix expressions.

Solution:

(a) $(A-B) * (C+D)$

$[AB-] * [CD+]$

$AB-CD+*$

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

$[AB+] / [CD+] - [DE^*]$

$[AB+CD+/] - [DE^*]$

$AB+CD+/DE^*-$

The algorithm given below transforms an infix expression into postfix expression. The algorithm accepts an infix expression that may contain operators, operands, and parentheses.

For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

```
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 "(".
    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
```

Example: Convert the following infix expression into postfix expression using the algorithm

$A - (B / C + (D \% E * F) / G)^*H$

$A - (B / C + (D \% E * F) / G)^* H)$

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 * -

Evaluation of a Postfix Expression:

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

```

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
    
```

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Algorithm to evaluate a postfix expression

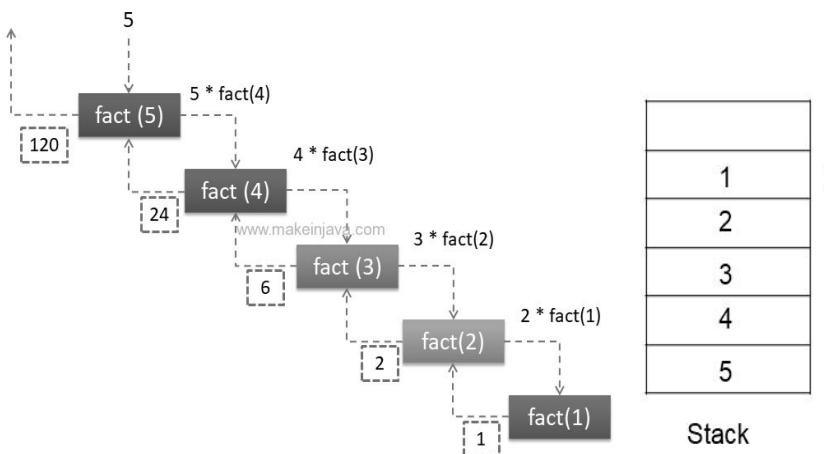
Evaluation of a postfix expression

Let us now take an example that makes use of this algorithm.

Consider the infix expression given as $9 - ((3 * 4) + 8) / 4$. Evaluate the expression.

The infix expression $9 - ((3 * 4) + 8) / 4$ can be written as $9\ 3\ 4\ * \ 8\ + \ 4\ / \ -$ using postfix notation.

Factorial Calculation:



We know $n! = n * n - 1! = n * n - 1 * n - 2! \text{ and so on}$

Recursion: $\text{factorial}(n) = n * \text{factorial}(n-1)$

$\text{fact}(5)$	$5 * \text{factorial}(4) \rightarrow \text{Push}(5)$
$\text{fact}(4)$	$4 * \text{factorial}(3) \rightarrow \text{Push}(4)$
$\text{fact}(3)$	$3 * \text{factorial}(2) \rightarrow \text{Push}(3)$
$\text{fact}(2)$	$2 * \text{factorial}(1) \rightarrow \text{Push}(2)$
$\text{fact}(1)$	returns 1 $\rightarrow \text{Push}(1)$



- Pop(1) Factorial value = 1
- Pop(2) Factorial value = $1 * 2 = 2$
- Pop(3) Factorial value = $2 * 3 = 6$
- Pop(4) Factorial value = $6 * 4 = 24$
- Pop(5) Factorial value = $24 * 5 = 120$

Factorial value = 120

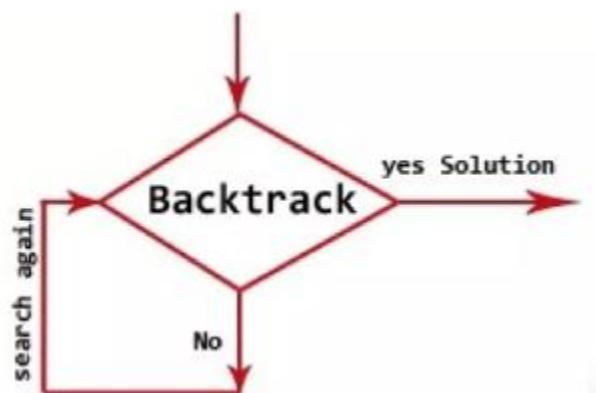
BACKTRACKING:

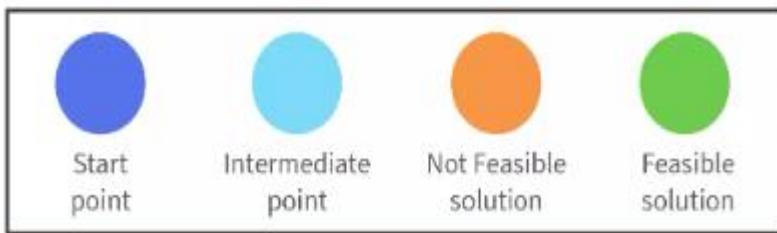
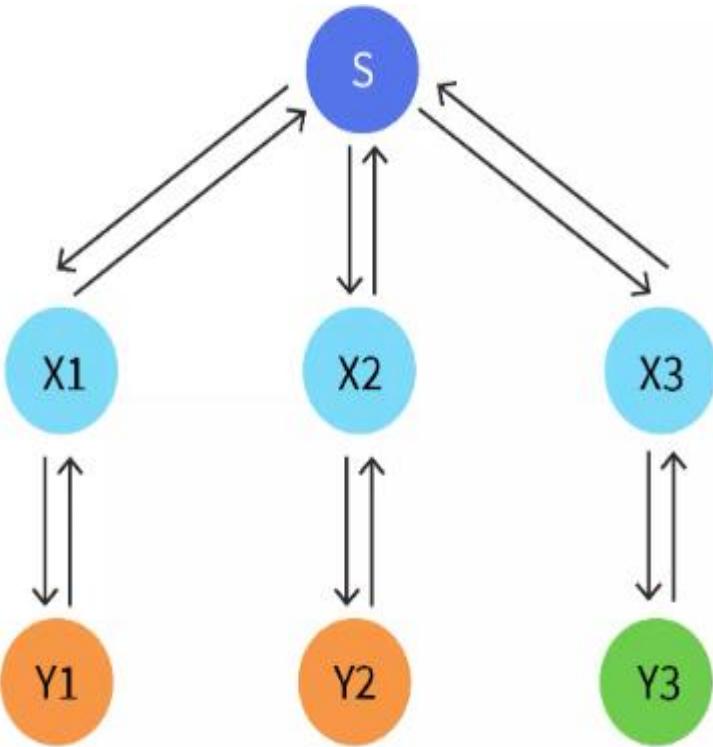
Backtracking is the technique to solve programs recursively. In backtracking problems, you will have a number of options and you must choose one of these options.

After you make your choice you will explore a new set of options, if you reach the feasible solution through those choices you will print the solution otherwise you will backtrack and return to explore the other paths and choices that can possibly lead to the solution.

For Instance: Consider the following space state tree. With the help of the below space state tree let's understand how the backtracking algorithm actually works.

We start from S which is the starting point of our problem. We go to find a solution to Y1 from intermediate point X1 and then we find that Y1 is not a feasible solution to our problem therefore we backtrack and go back to X1 via Y1 then go back to S and then again try to find out the feasible solution by following another path S->X2->Y2 and again we will find that Y2 is not a feasible solution so we will again return to S by following path Y2->X2->S and then we will ultimately reach out to feasible solution Y3 by following the path S->X3->Y3.





So we can summarize the backtracking algorithm as follows:

- ✓ We select one possible path and try to move forward towards finding a feasible solution.
- ✓ If we will reach a point from where we can't move towards the solution then we will backtrack.
- ✓ Again we try to find out the feasible solution by following other possible paths.