Group 2, Team 4

Inder Barthwal
Harshit Garg
Akash Sharma

Stacks

Introduction

A **stack** is a collection of objects that are inserted and removed according to the **last-in, first-out** (LIFO) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack).

LIFO

LIFO is an abbreviation for **Last in, first out** is same as fist in, last out (FILO).

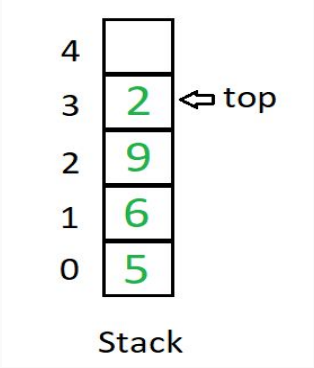
Stack Vs Array

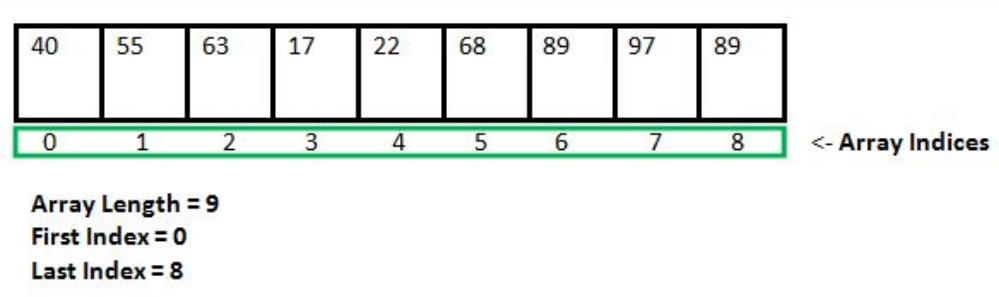
An array is a contiguous block of memory.

- A stack is a first-in-last-out data structure with access only to the top of the data.
- Since many languages does not provide facility for stack, it is backed by either arrays or linked list.

Stack Vs Array

- The values can be added and deleted on any side from an array.
- But in stack, insertion and deletion is possible on only one side of the stack. The other side is sealed.





Implementation

- Stack data structure is not inherently provided by many programming languages.
- Stack is implemented using arrays or linked lists.

  Let S be a stack, n be the capacity, x be the element to be pushed, then push and pop will be given as

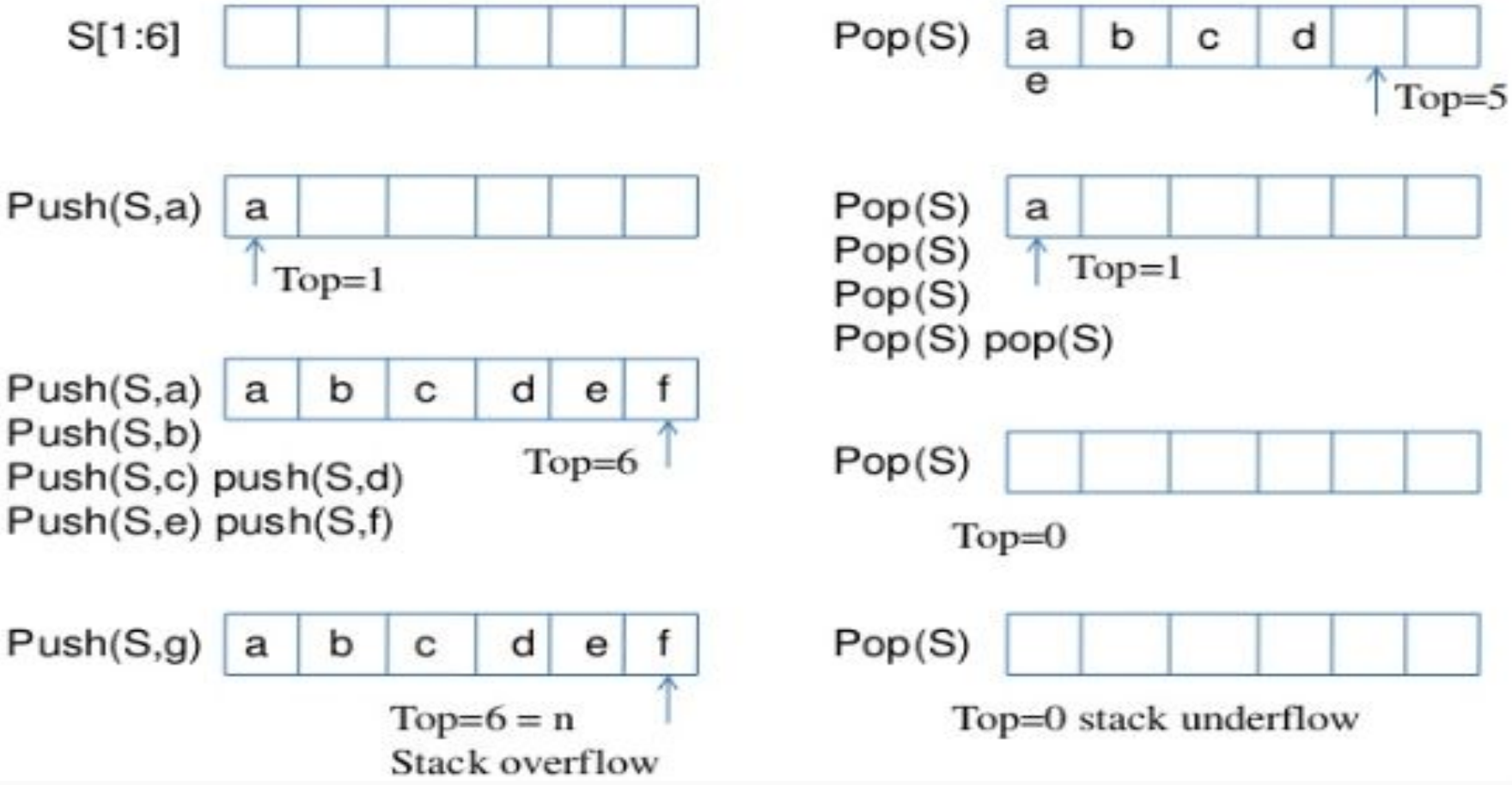
Push(S,x) and Pop(S)

Here we use “top” which keeps track of the top element in the stack.

When top = = 0 , pop() operation gives stack underflow as result.

When top = = n, push() operation gives stack overflow as result.

The pop() operation just gives an illusion of deletion, but the elements are retained. Only the top is decremented.


S[1:6]
Pop(S)
a b c d
e
Top=5
Push(S,a)
a
Top=1
Pop(S)
Pop(S)
Pop(S)
Pop(S) pop(S)
a
Top=1
Push(S,a)
Push(S,b)
Push(S,c) push(S,d)
Push(S,e) push(S,f)
a b c d e f
Top=6
Pop(S)
Top=0
Push(S,g)
a b c d e f
Top=6 = n
Stack overflow
Pop(S)
Top=0 stack underflow

Basic Stack Operations..

- **Push :** insert an element from the top
- **Pop :** delete an element from the top
- **Peek :** get the top element of the stack, without removing it.
- **isFull :** check if the stack is full.
- **isEmpty :** check if the stack is empty.

Peek Operation..

- It copies the item at the top of the stack and return the item to the user(the application that calls this operation), but does not remove the item.
- Be careful the empty state or *underflow* state of the stack, when implementing **stack top** operation.
- Pseudo Code:

```
int peek() {
return stack[top];
 }
```

isFull Operation..

- This checks if the stack is full or not.
- Pseudo code:

```
bool isfull() {
if(top == MAXSIZE)
return true;
 else
 return false;
 }
```

isEmpty Operation..

- This function checks whether the stack is empty or not.
- Pseudo Code:

```
bool isempty() {
if(top == -1)
return true;
 else
return false;
 }
```

Push Operation..

- **Push** operation adds an item at the top of the stack.
- Before adding the item, the stack space must be checked to ensure that there is enough room to hold the item.
- Push operation involves series of steps −

  Step 1 − Check if stack is full.

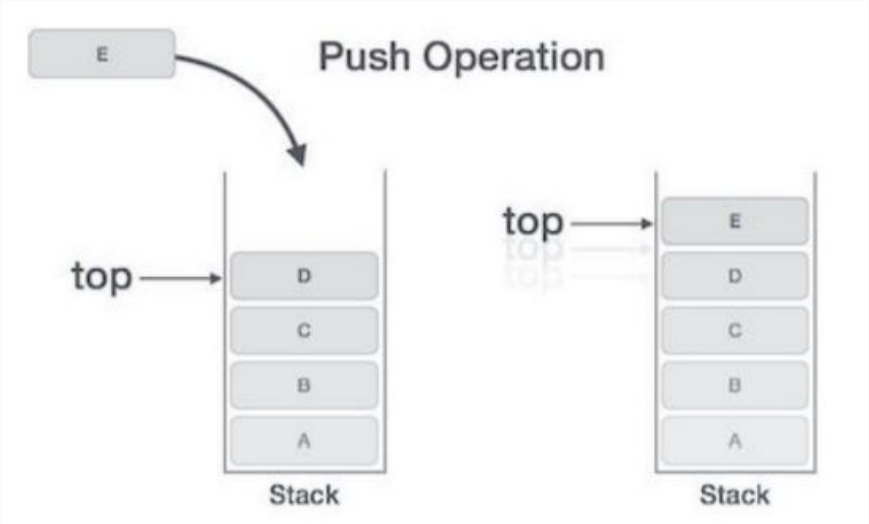
  Step 2 − If stack is full, produce error and exit.

  Step 3 − If stack is not full, increment top to point next empty space.

  Step 4 − Add data element to the stack location, where top is pointing.

  Step 5 − return success.

Push Operation (Contd..)

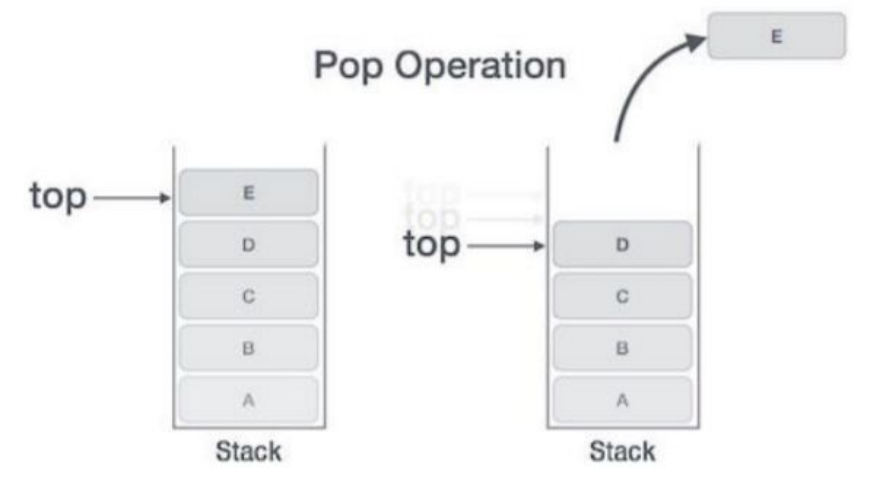



```
void push(int data) {
if(!isFull()) {
 top = top + 1;
 stack[top] = data;
 }
else {
 printf("Could not insert data, Stack is full.\n");
}
 }
```

Pop Operation..

- **POP** removes the item at the top of the stack and return the item to the user(the application that calls this operation).
- Be careful the empty state or *underflow* state of the stack, when implementing **pop** operation.
- A POP operation may involve the following steps −
  Step 1 − Check if stack is empty.
  Step 2 − If stack is empty, produce error and exit.
  Step 3 − If stack is not empty, access the data element at which top is pointing.
  Step 4 − Decrease the value of top by 1.
  Step 5 − return success.

Pop Operation (contd..)



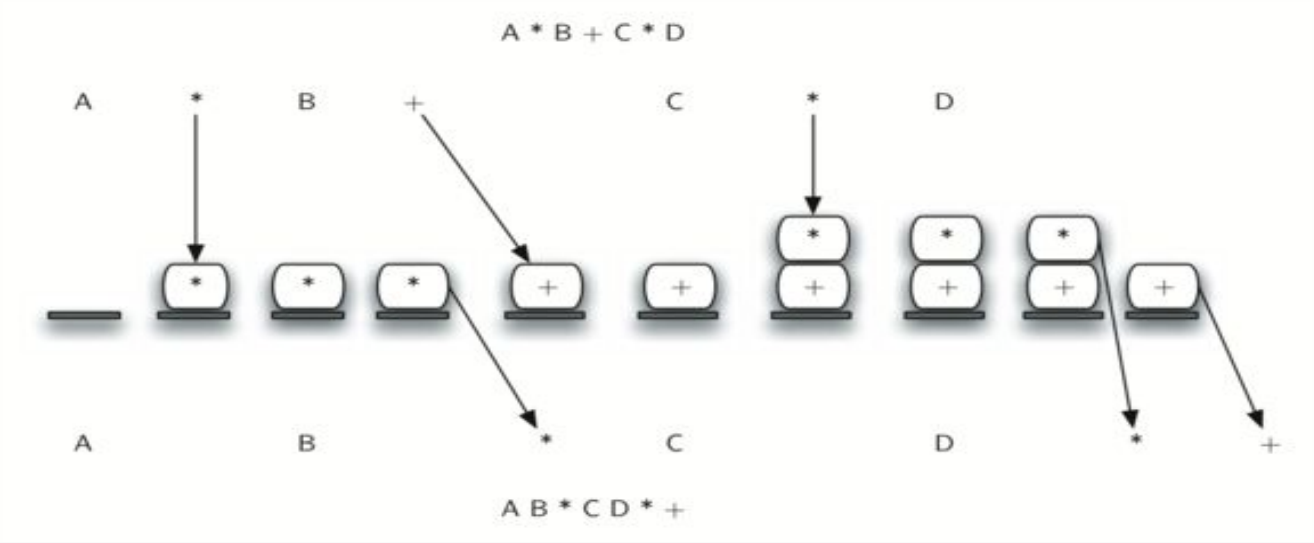

```
int pop(int data) {
if(!isempty()) {
 data = stack[top];
 top = top - 1;
 return data;
 }
else {
 printf("Could not retrieve data, Stack is empty.\n");
}
}
```

APPLICATIONS OF STACK

Some important applications of Stacks are:

- Expression conversion
- Expression evaluation
- Function call
- Backtracking
- Syntax parsing

Expression Conversion



Pushing all the operators in stack until you find a lower precedence operator ,and then pop.

Uses operator stack

Expression conversion

abc*de-/+ = +a/*bc-de

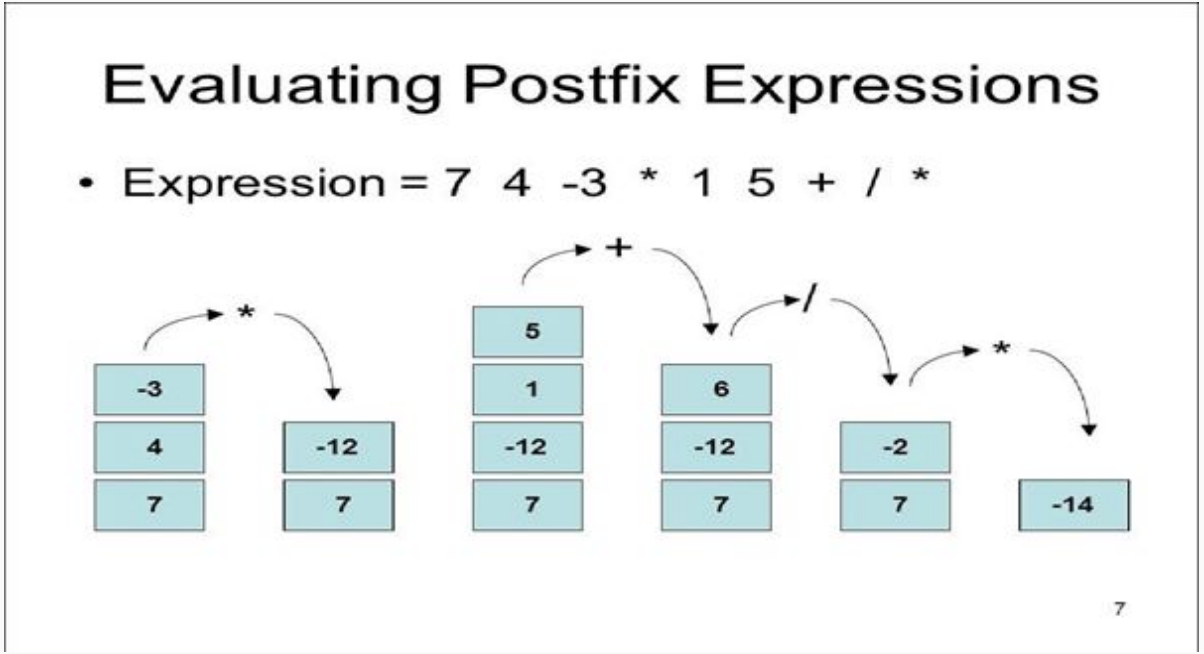
SYMBOL	STACK
a	a
b	a | b
c	a | b | c
*	a | *bc
d	a | *bc | d
e	a | *bc | d | e
-	a | *bc | -de
/	a | /*bc-de
+	+a/*bc-de

Hence, the equivalent infix expression: +a/*bc-de

Similarly infix ,prefix and postfix expressions can be converted into one another,using a stack

Expression Evaluation

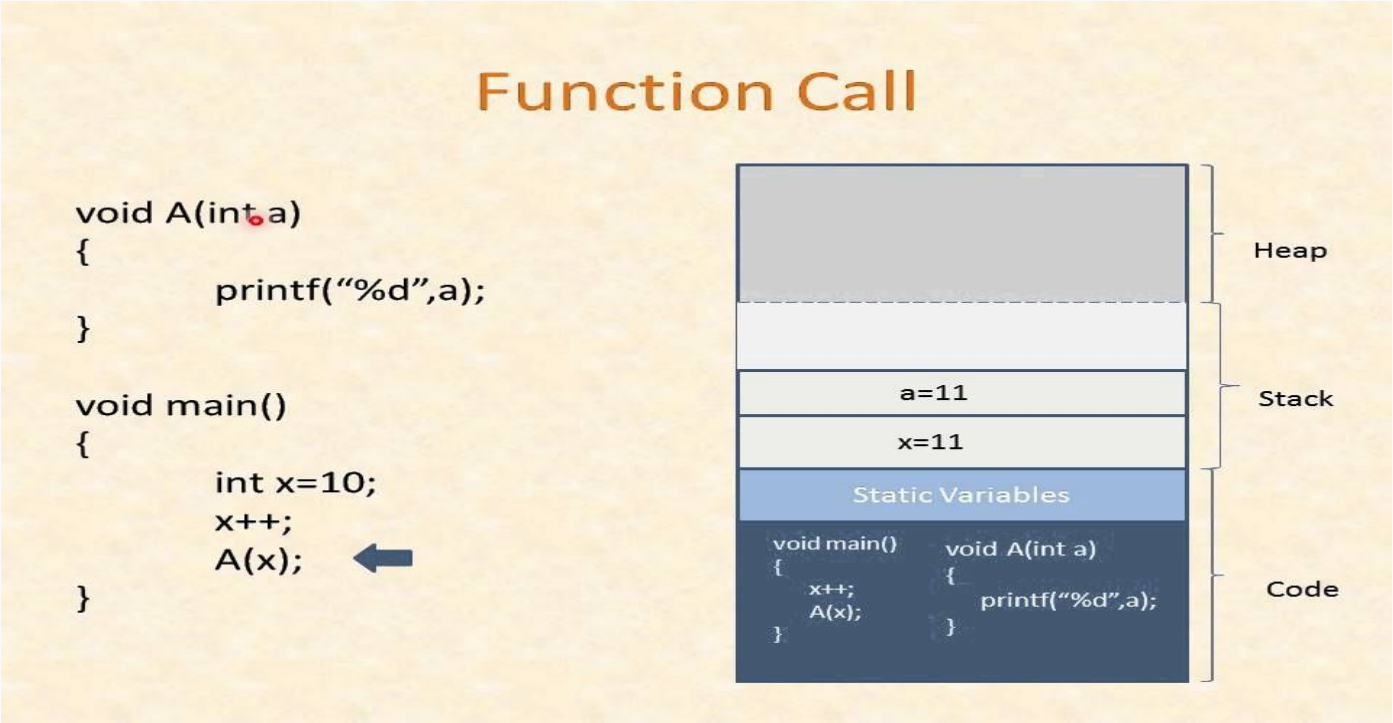
Postfix and prefix expressions can be evaluated with the help of stack.



In expression evaluation stack used to store operand ,and we keep on pushing until we find an operator,where we pop and evaluate.

Function call

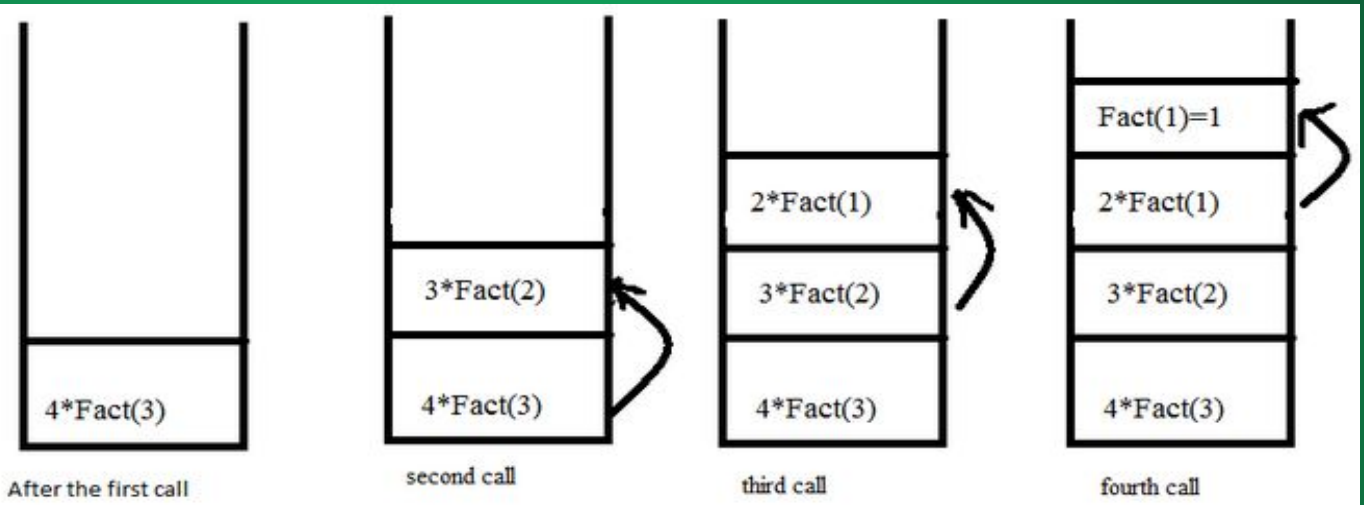
Memory can be of many types like stack,heap,static etc.
Function call uses stack memory.



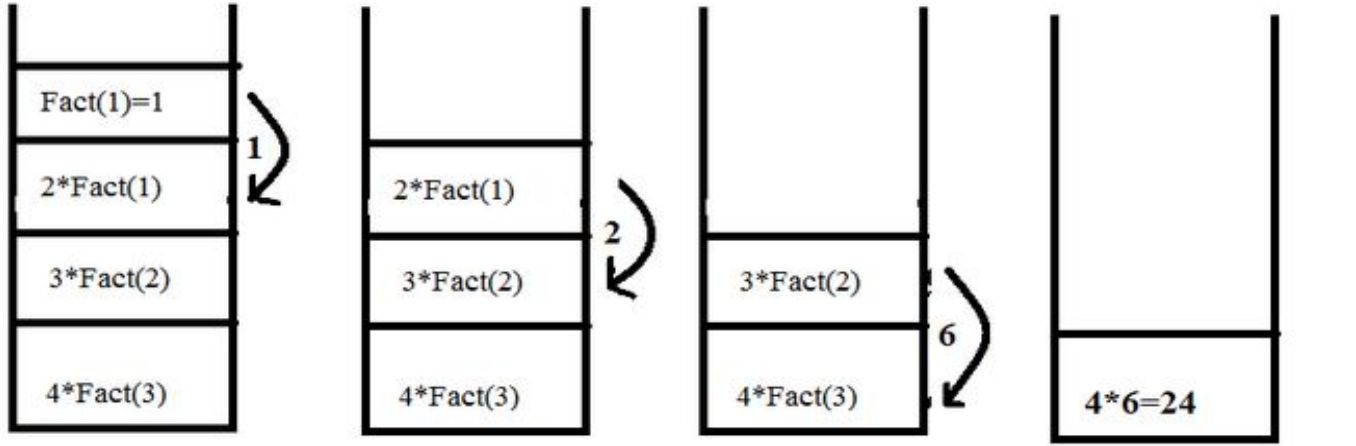
Stack memory is holding the values of the local variables like a,x .

```
int fact(int n)
{
if(n==1)
   return 1;
return fact(n-1)*n;
}
main()
{
int n=4;
printf("%d",fact(n))
return 0;
}
```

For the factorial program the expression evaluates using stack.

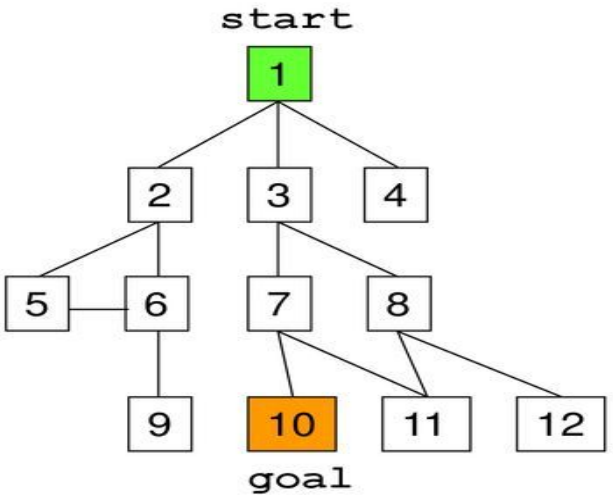


Returning values from base case to caller function



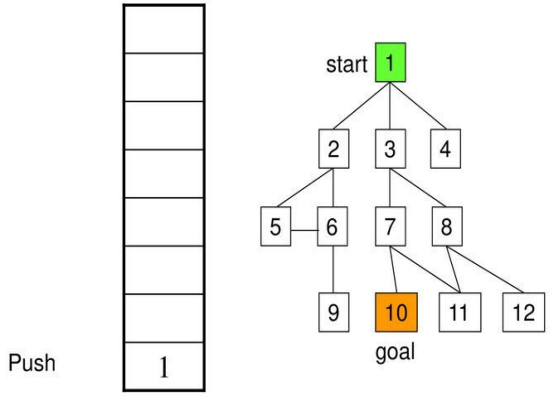
Backtracking

- Problem
  - Discover a path from `start` to `goal`
- Solution
  - Go deep
    - If there is an unvisited neighbor, go there
  - Backtrack
    - Retreat along the path to find an unvisited neighbor
- Outcome
  - If there is a path from `start` to `goal`, DFS finds one such path

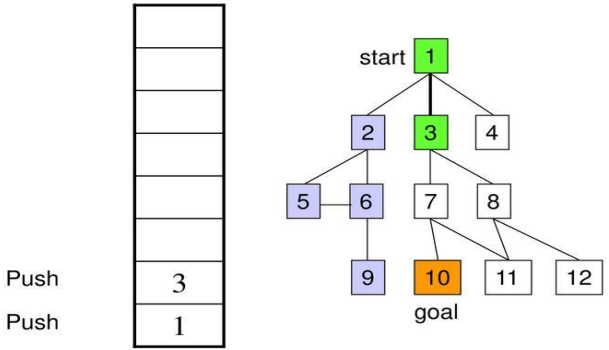


- Stack

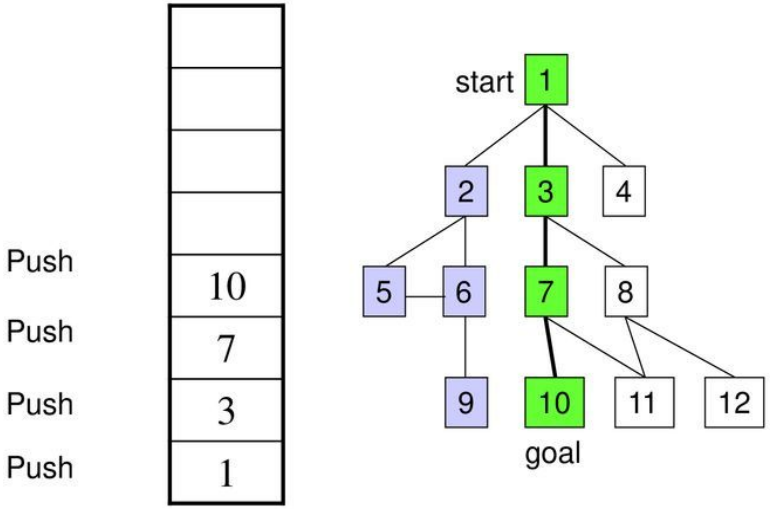
1



- Stack



After some number of steps we could reach to the goal,and if we make some wrong path we could pop the paths and move to the next directions.
DFS uses stack.



Syntax parsing

For grammar
S->S+S
S->S*S
S->id

Perform parsing on string:
Id+id+id
Predictive parsing uses stack.

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce by S --> id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce by S --> id
$S+S	+id$	Shift
$S+S+	id$	Shift
$S+S+id	$	Reduce by S --> id
$S+S+S	$	Reduce by S --> S+S
$S+S	$	Reduce by S --> S+S
$S	$	Accept


THANK YOU