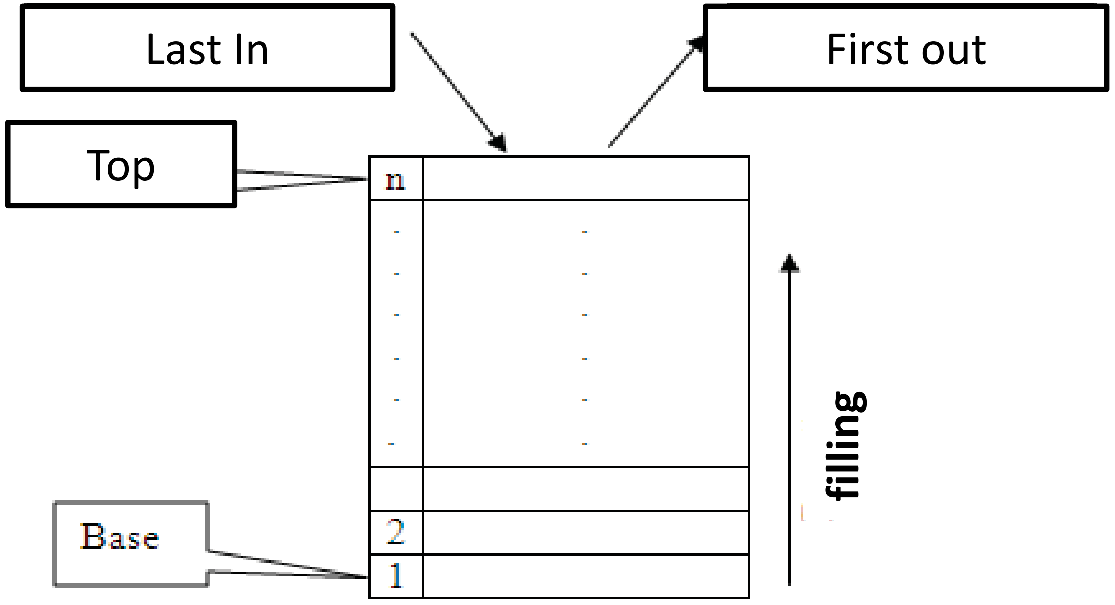
Chapter 3: Stacks

Stacks

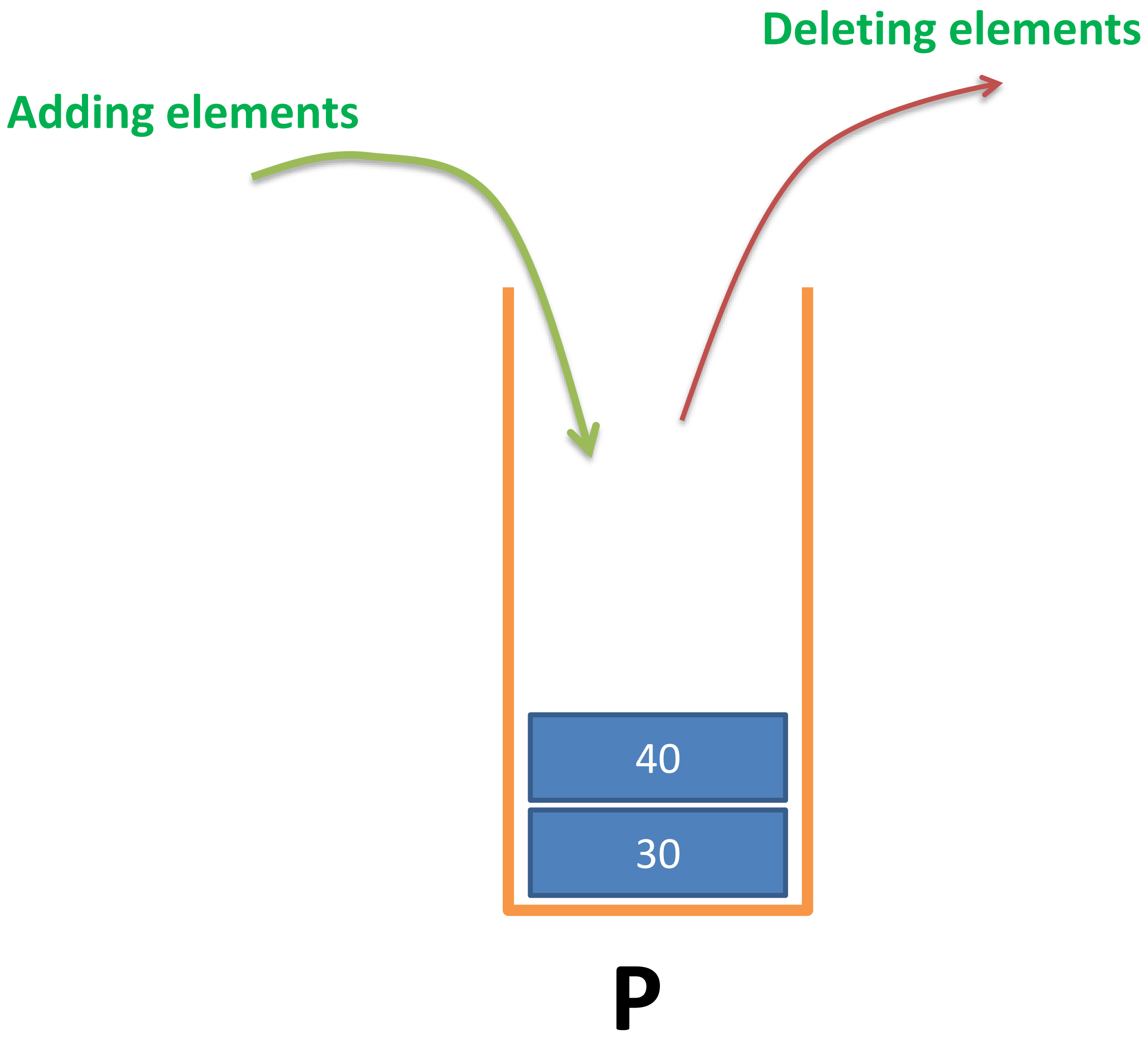
1. **Definition**
2. **Dynamic Stacks**
3. **Static Stacks**

1. Definition

- A stack is a list of elements where insertions and deletions of elements occur at a single end of the list called the top of the stack.
- The principle of adding and removing from the stack is called LIFO (Last In First Out): "the last one in is the first one out". It is therefore impossible to access an element in the middle.

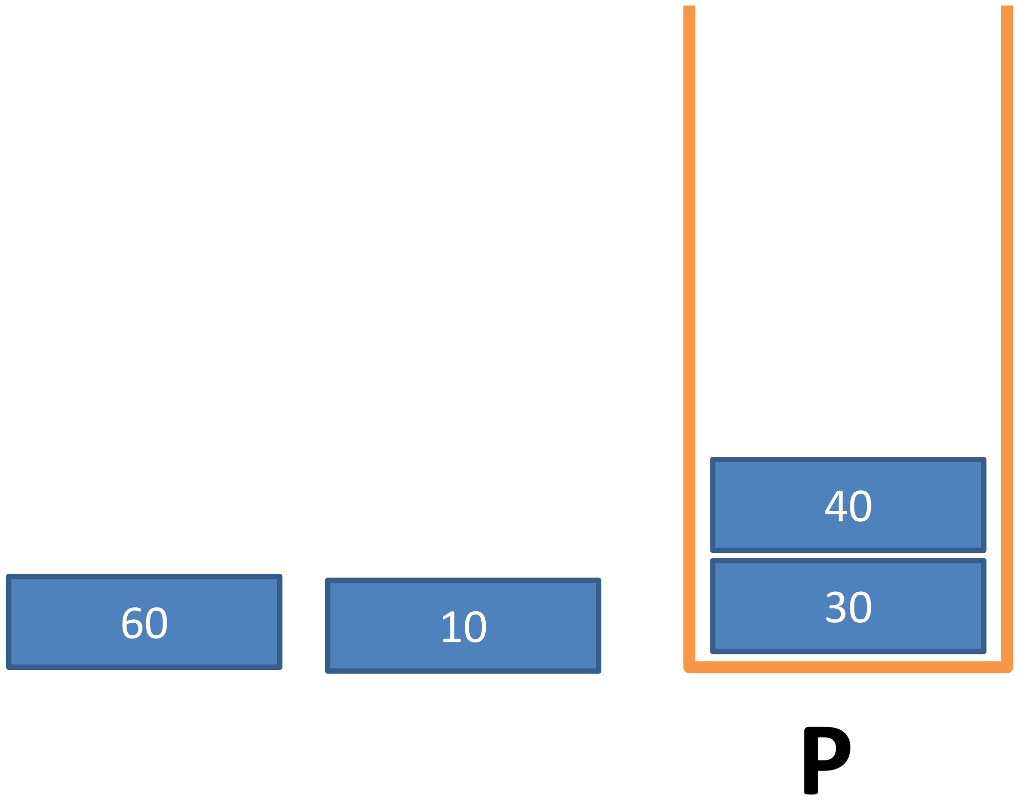


1. Definition



1. Definition

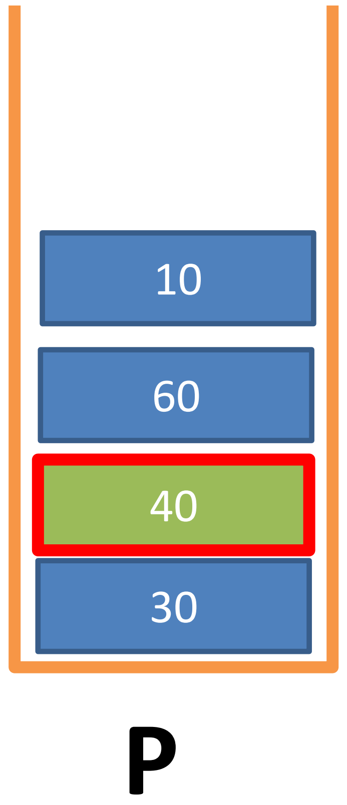
Example: adding two elements 60 and 10 to stack P



1. Definition

Example 2:

- **Remove 40 from the stack P**
- **First you have to remove 10 and 60 and then 40**



1. Types

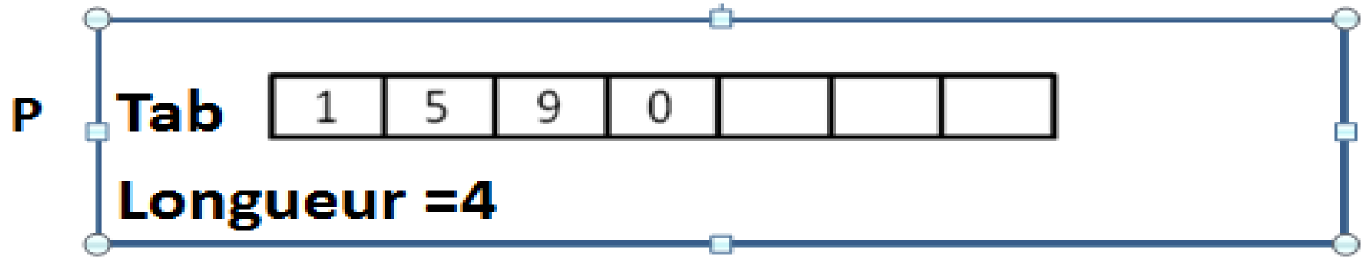
Depending on the type of implementation, we distinguish two types of Stacks:

- **Static (contiguous) stack:** Implementation by an array.
  - The same implementation of a static list
- **Dynamic Stack:** Linked List Implementation
  - The same implementation of a linked list

Static stack

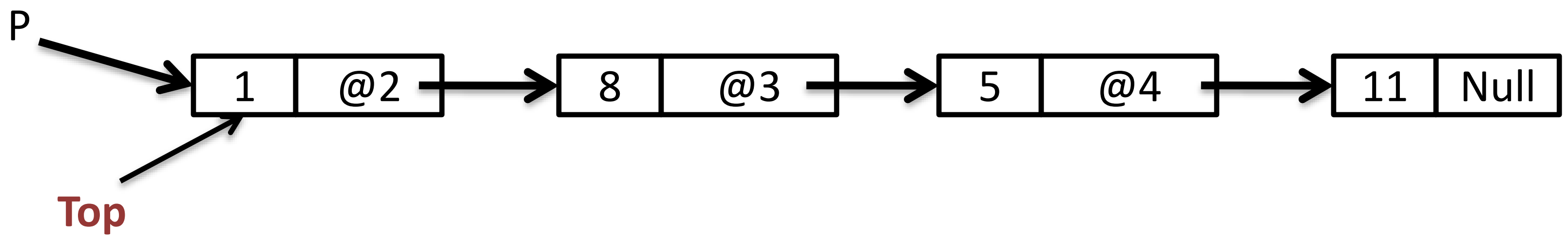
- Implementation : by using an array
- The capacity of the stack is limited by the size of the array.
- Pushing an element onto the stack is done in the increasing order of indices
- Popping is done in the opposite direction.

Example:



3. Dynamic Stacks

It is a linked list where stacking and popping is done only at the head of the list.



<u>Stack type definition</u>

Type Structure Node

  Ele : typeq ; // typeq designates any type (int, float , person, student...etc).

  next: * Node ;

END

Type stack: * Node; // The Stack type designates all pointers to a Node

Primitive operations

Primitive operations: Common operations on dynamic stacks are :

- **Create_Node (X: element): Stack**
- **Is_empty (P: stack): Boolean**
- **Size (P: stack): integer**
- **Push (var P: stack, x: Typeq):**
- **Pop (Var P: stack)**
- **Top (P: stack): Typeq**

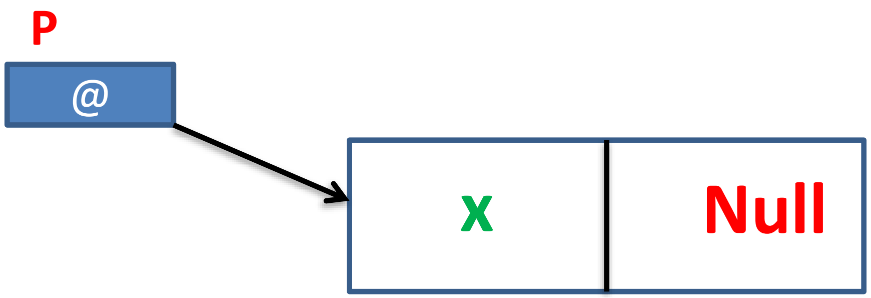
Dynamic Stack (Primitive Operations)

1. **Create_node** : creates a new Node, with dynamic memory reservation (Allocate), containing the value **x** in the field element and **Null** in the Next field. This function returns the address of the created Node.

```
Function Create_Node (x: typeq ) : Stack

    P: Stack// or P: * Node

Begin
     P ← Allocate (Node)
     P - > Ele ←x;
     P -> next ← Null;
     Return (P);
END
```



3. Dynamic Stacks

Is_empty : returns true if the Stack is empty (P = NULL) otherwise returns false.

```
Function Is_empty(P: Stack): Boolean
Begin
    If (P=Null) then
        Returns (true);
    Else
        Return(false);
    End if
END
```

3. Dynamic Stacks

Top (P: Stack): Returns the element that exists in the Stack header (The last stacked element).

```
Function Top (P: Stack): type
Begin
    Return (P -> ele );
END
```

3. Dynamic Stacks

Push: Pushes an element on top of the stack. This function is equivalent to Add for a linked list.

```
Procedure Push (var P: Stack, x: typeq )
    Q: Stack
Begin
    Q ← create_node (x);
    Q -> next ← P;
    P ← Q ;
END
```

3. Dynamic Stacks

Pop : Returns the stack without its top.

```
Procedure Pop (var P: Stack)
    Q: stack
begin
   If ( not is_empty (p)) then
          Q ← P ;
          P ← p - >next;
          Disallocate(Q);
    End If
END
```

1. Dynamic stacks

The size function : returns the number of elements stacked in the stack. Equivalent to Length for Linked Lists.

```
Function Size (P: Stack): integer
        Current: stack; Nb : integer;
Begin
   Current ← P ; Nb ← 0 ;
   While (current ≠ NULL) do
        Nb ← Nb+1 ;
        Current ← Current -> next ;
   Endwhile
        Return Nb ;
END
```

4. Examples of stack usage

1. Evaluation of post-fixed expressions

- For the evaluation of arithmetic or logical expressions, programming languages typically use **prefix** and **postfix** representations .
- In the postfix representation, the expression is represented by a new one, where the operations always come after the operands.
- <u>**Example**</u>
  - ✓ The expression **(2 + 3 *)* * *6*** *is expressed, in postfix, as follows* ***2 3+ 6****
  - ✓ The expression **(** ***a* + *(b* ∗ c))/(c - d)** *is expressed, in postfix, as follows:* ***a bc* ∗ + *cd* – */***

4. Examples of stack usage

- For the evaluation of postfix expressions, programming languages use the Stack type and its primitives by traversing the expression from left to right.

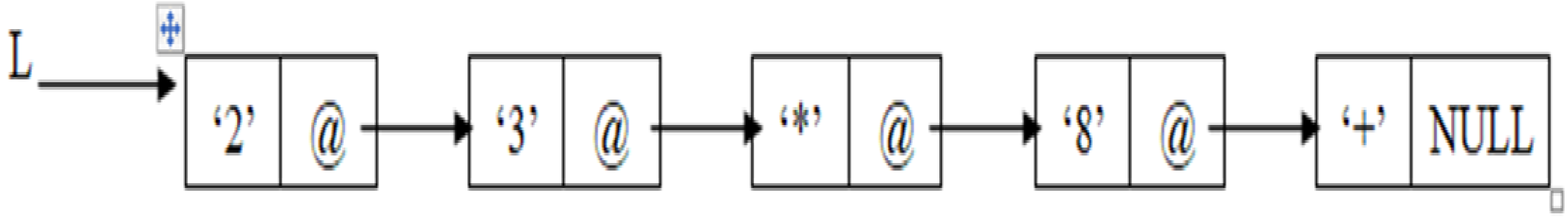
4. Examples of stack usage

Example : The following algorithm evaluates the arithmetic expression **(2 + 3** ***) * 6*** *or 2 3 + 6 * in postfix* using a stack.

```
P ←NiL ;
Push(P,2); Push(P,3);
b ← Top (P); Pop (P);
a ← Top (P); Pop (P);
Push(P,a+b);
Push (P, 6);
b ← Top (p); Pop (p);
a ← Top (p); Pop (p);
Push (p,a*b);
Write (Top (p));
```

4. Examples of stack usage

Exercise: Using a stack, write the function to evaluate a post-fixed arithmetic expression, represented by a linked list, containing only digits (0,1,…9) as operands and the following operators: addition '+' and multiplication '*'.



4. Examples of stack usage

2. Recursive calls

- To execute recursive functions programming languages use stacks to make them iterative.
- The following example represents an iterative version for the factorial function using a stack and its primitives.

4. Examples of stack usage

Example: **Fact (4)**

=4*Fact(3)

=4* **3*Fact(2)**

=4*3* **2*Fact(1)**

=4*3*2* **1*Fact(0)**

= 4*3*2*1* **1**

=4*3*2* **1**

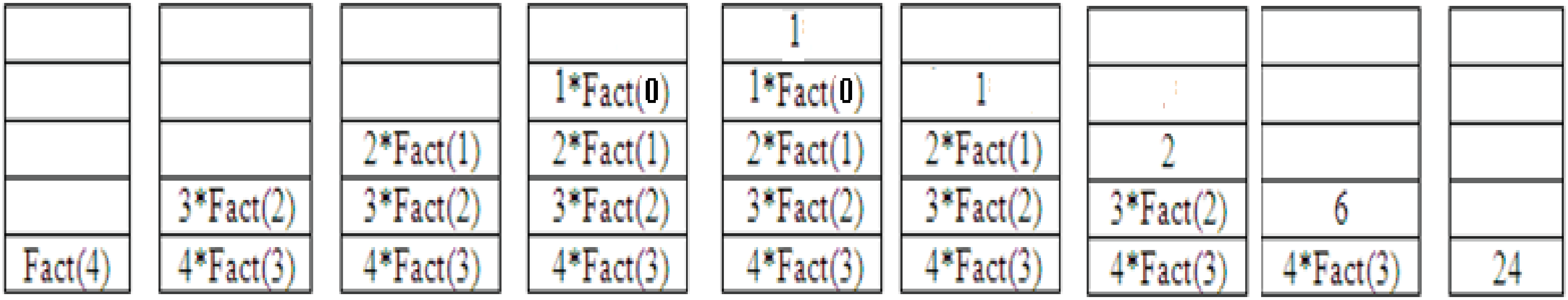
=4*3* **2**

=4* **6**

= **24**

Recursive

Calculation phase
(return and calculation)



4. Examples of stack usage

```
int factorial ( int n)
{   stack p=NULL ;
   for( int i = N, i > =0 ; i --)
     {
             if ( i ==0)
                 push (P, 1);
             else
               push (P,i);
    }

    while ( size (P) > 1 )

     {
             b = top(P); Pop(P);
             a = top(P); Pop(P);
             push (P, a*b);
     }
             return ( Top(p));
}
```

$\left.\vphantom{\begin{matrix}a\\b\\c\\d\\e\\f\end{matrix}}\right\}$ Filling phase
(Recursive calls)

$\left.\vphantom{\begin{matrix}a\\b\\c\\d\\e\\f\end{matrix}}\right\}$ Calculation phase
(return and calculation)

4. Examples of stack usage

Exercise : Using a stack, write an iterative version for the recursive function power(x, n) respecting the algorithm for its execution by programming languages.

```
int ( int x, int n)
{
    if ( n ==0)
        return 1;
    else
        return x * power(x, n-1);
}
```

Algorithm Power_Iterative_Using_Stack(x, n : integer) : integer

```
Var
   S : Stack         // stack of integers
   result : integer
Begin
   Create_Empty(S)
   // Simulate recursive calls: push x exactly n times
   While (n > 0) do
      Push(S, x)
      n ← n - 1
   End while
   // Base case: x^0 = 1
   result ← 1

   // Simulate return of recursive calls
   While (not Is_Empty(S)) do
      result ← result * top(S)
      pop(S)
   End while

   Return(result)
End
```

4. Examples of stack usage

Undo Mechanism in Text Editors: Text editors often use a stack to implement the undo feature. Every change you make (such as typing, deleting, or formatting) is pushed onto the stack. When you hit "undo," the last change is popped off and reversed. This makes it easy to implement multiple undos.

Browser History Management: Browsers use stacks to store the history of visited pages. When you navigate to a new page, it is pushed onto the stack. Pressing the "back" button pops the current page, taking you to the previous one.

Syntax Parsing: Compilers and interpreters use stacks to parse syntax, especially for languages with nested structures like parentheses or curly braces. This stack structure helps ensure that every opening symbol has a corresponding closing symbol.

1. Static Stack

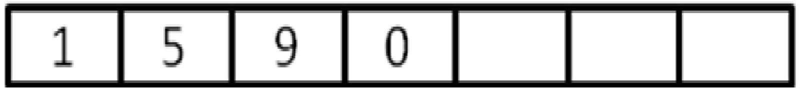
Implementation by an array and in this case:

- the stack capacity is limited by the size of the array.
- **Addition to the stack is done in the ascending direction of the indices, while removal is done in the opposite direction.**

Example :

Let the following static stack P be with a 7 element array (fixed size) and a length variable containing the value 4 which is the number of elements existing in the list.

Tab



Length =4

4. Static stack

Stack Type Definition

```
Type Structure Stack
Begin
Tab: array[MAX] of Elements;
length: Integer ; // keeps the number of elements stored in the Stack
END
```

Example: Stack of integers

```
    Type Structure Stack
    Begin
    Tab: Array[1000] of integers;
    length: Integer;
    END
```

1. Static Stack : common operations

Common operations on static Stacks

The initialize procedure : initializes the length variable of a stack to the value 0. It therefore creates an empty stack.

```
Procedure Initialize(var P: Stack)
Begin
    P. length ← 0;
END
```

The function is_empty : a boolean function that receives a stack as input and returns true if the stack is empty (Length = 0) and false otherwise.

```
    Is_empty function (P:stack): Boolean
    Begin
    Returns ( P.Length == 0);
    END
```

1. Static Stack : is-full function

The function isfull: tests whether a stack is full or not. It returns true if the length of the stack equals its maximum size (length = max);

```
Is_full function (P: Stack): Boolean
Begin
Returns ( P.length = Max);
END
```

The Top function : returns the top element of the stack (the last element stacked)

```
Function Top (P: Stack) :  integer
Begin
return( P.Tab [ P.length ]);
END
```

1. Static stack : Push procedure

The stack procedure : If the Stack is not full, adds an element to the top of the stack and increments the length variable by 1, otherwise it does nothing.

```
Procedure Push (P: Stack, x: Element )
Begin
If (! is_full (P) ) then
    P.Tab [ P.Length +1] ← x;
    P.Length ← P.Length +1;
End if
END
```

1. Static Stack : pop procedure

The pop procedure : if the stack is not empty, it decrements the length variable by 1, otherwise it does nothing.

```
Procedure Pop (var P: stack)
begin
If (! is_empty (P) )then
        P.Length ← P.Length -1;
End if
END
```

1. Static Stack : size function

The size function : returns the number of elements already stacked in the stack.

```
size function (P: Stack): integer
begin
return (P.Length) ;
END;
```