ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT III : Queues & Stacks

Topic: Stacks

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

Contents

Queues:

- Introduction to Queues
- Representation of Queues-using Arrays
- Implementation of Queues-using Arrays
- Representation of Queues-using Linked list
- Implementation of Queues- using Linked list
- Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.

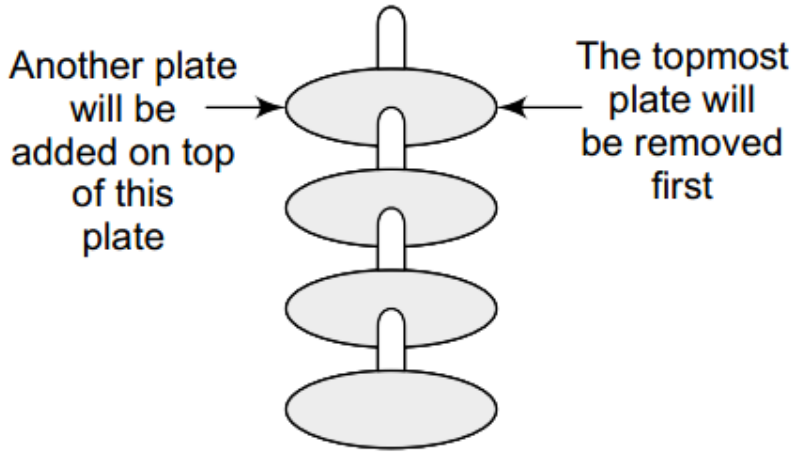
Stacks:

- Introduction to Stacks
- Array Representation of Stacks, Operations on Stacks,
- Linked list Representation of Stacks, Operations on Linked Stack,
- Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

Introduction to Stacks

Introduction to Stacks

- Stack is an important data structure which stores its elements in an ordered manner.
- Take an analogy of a pile of plates where one plate is placed on top of the other.
- A plate can be removed only from the topmost position.
- Hence, you can add and remove the plate only at/from one position, that is, the topmost position.



Introduction to Stacks

- 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**.
- 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.

Definition: A Stack is linear list in which insertions (also called additions or pushes) and removals (also called deletions or pops) takes place at the same end. This end is the **top**. Since the last element inserted into a stack is the first element to remove. A stack is also known as **Last-In First-Out (LIFO)** list.

- Two operations mainly performed in stack.
  1. **Push:** inserting elements into stack.
  2. **Pop:** removing elements from stack.

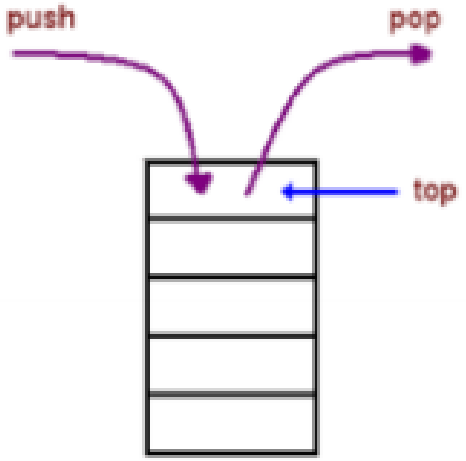
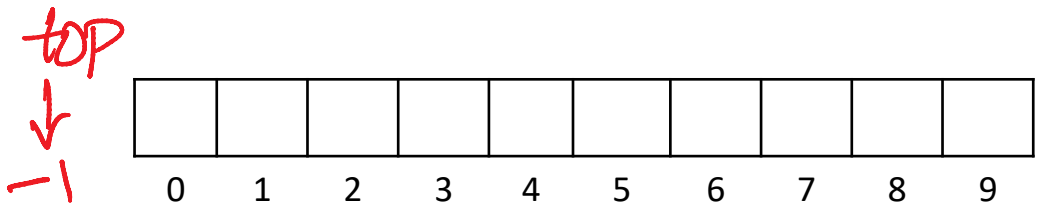


Fig: stack structure

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**.



top == -1 → empty

top == MAX-1 → full

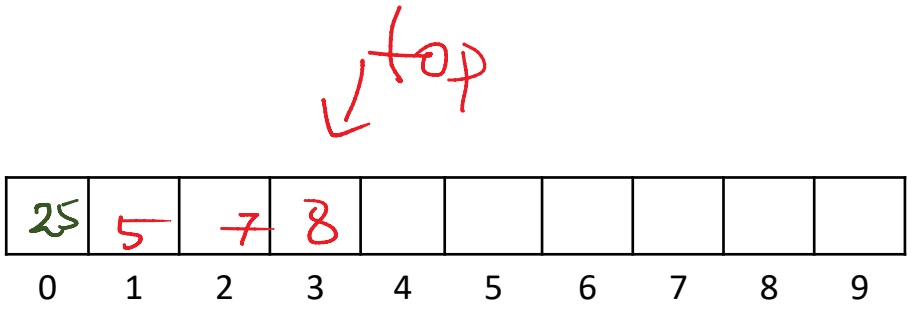
OPERATIONS ON A STACK

- A stack supports three basic operations: **push**, **pop**, and **peek**.
- The **push** operation adds an element to the top of the stack.
- 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.

top++
push = 25, 56, 35

top--
pop = 35, 56

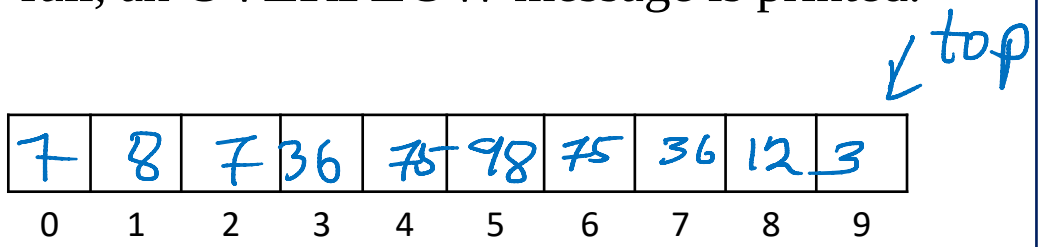
peek = 8

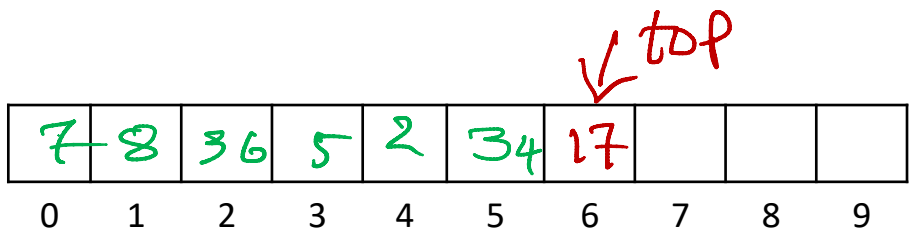


Push Operation:

- The push operation is used to **insert** an element in to the stack.
- The new element is added at the **topmost position** of the stack.
- However, before inserting the value, we must first check if **TOP=MAX-1**, because if this is the case then it means the stack is full and no more insertions can further be done.
- If an attempt is made to insert a value in a stack that is already full, an **OVERFLOW** message is printed.

```
stack[]        val=17
push( )
{ if(top == MAX-1)
      printf("OVERFLOW");
  else
  {
    top++;
    stack[top]=val;
  }
}
```





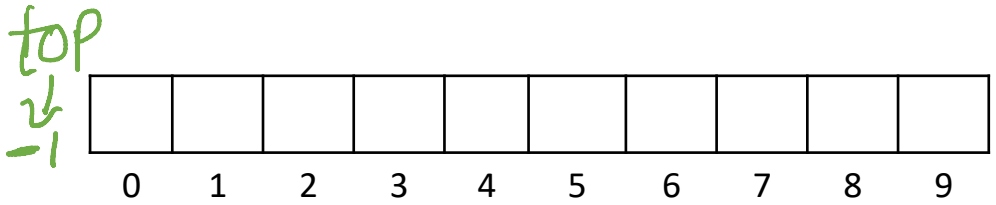
Push Operation Algorithm:

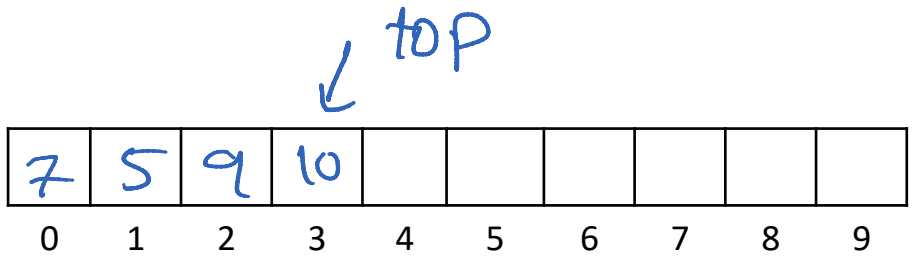
```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

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 **TOP=NULL**, because if this is the case then it means the stack is empty so no more deletions can further be done. •
- If an attempt is made to delete a value from a stack that is already empty, an **UNDERFLOW** message is printed.

```
pop( )
{ int val;
  if(top == -1)
     printf ("underflow");
  else
  {
     val = stack[top];
      top--;
  } return val;
}
```





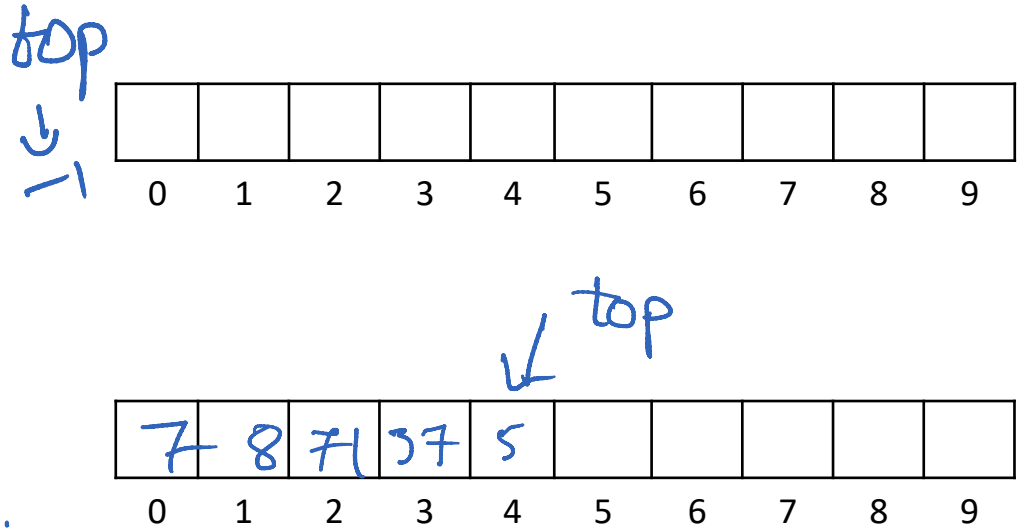
Pop Operation Algorithm:

```
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:

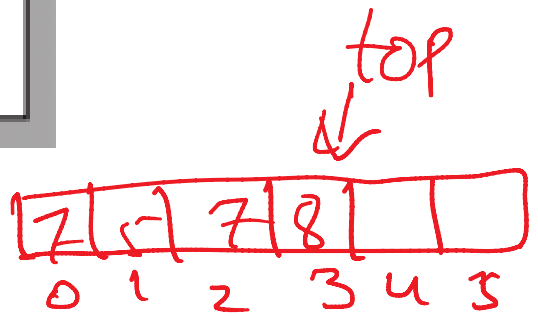
- Peek is an operation that **returns** the value of the **topmost** element of the stack without deleting it from the stack.
- However, the peep operation first checks if the stack is empty or contains some elements.
- If **TOP = NULL**, then an appropriate message is printed else the value is returned.

```
peek ( )
{
  if( top == -1)
   { printf("underflow");
     return 1;
   else
      return stack[top];
}
```



Peek Operation Algorithm:

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

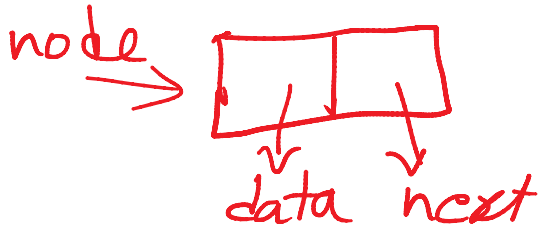


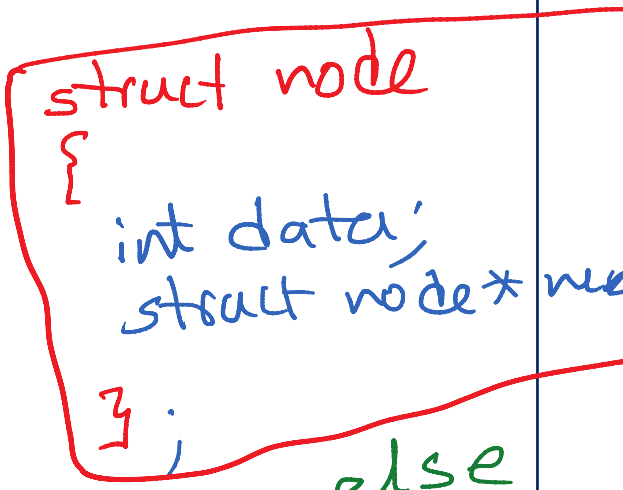
Display :-

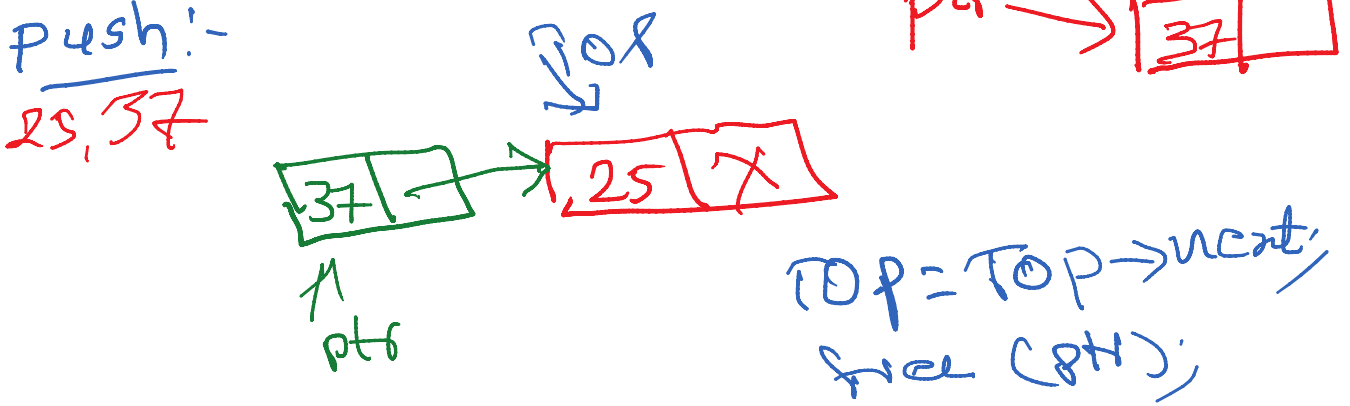
```
display()
{
if(top==-1)
printf("underflow");
}
else
{
for(i=0; i<=top; i++)
    printf("%d", stack[i]);
}
```

LINKED REPRESENTATION OF STACKS

- 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.
- A linked stack supports all the three stack operations, that is, **push**, **pop**, and **peek**.





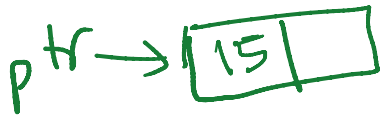


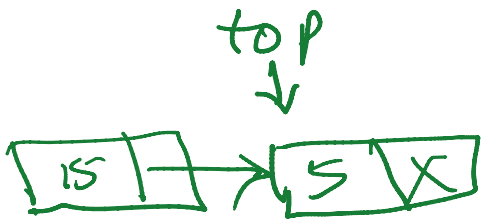
Push Operation on a Linked Stack:

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- 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.

```
push ( top, val)
{
 struct node *ptr;
  ptr = malloc();
  ptr -> data = val;
  if (top == NULL)
  {
    ptr -> next = NULL;
    top = ptr;
  }
  else
  {
    ptr -> next = top;
    top = ptr;
  }
}
```

val = 15





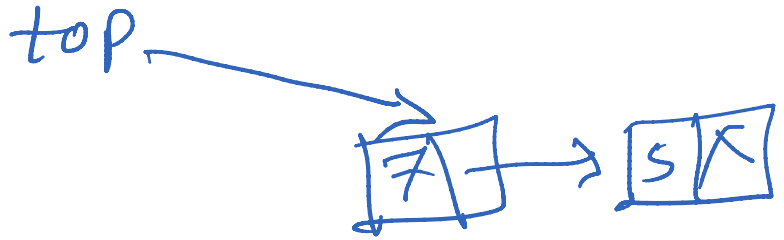
Push Operation Algorithm:

```
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 on a Linked Stack:

- 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. An UNDERFLOW message is printed.
- 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.

```
pop ( top)
{
  struct node * ptr;
   ptr = top;
   if (top= - NULL)
      printf("\n stack empty");
   else
     { top = top -> next;
       pf(" %d", ptr -> data);
     }
   free(ptr);
}
```

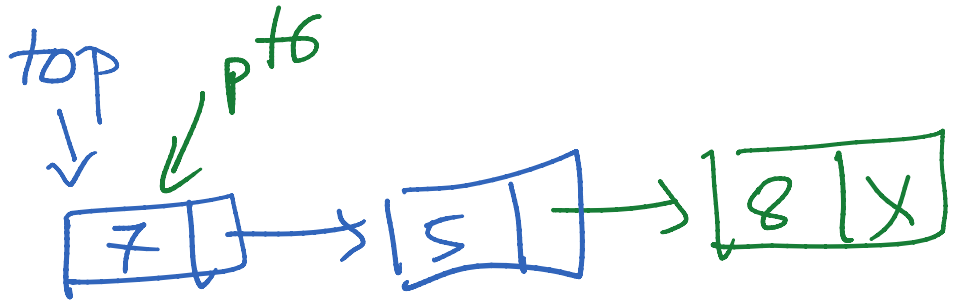


Pop Operation Algorithm:

```
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
```

Peek Operation on a Linked Stack:

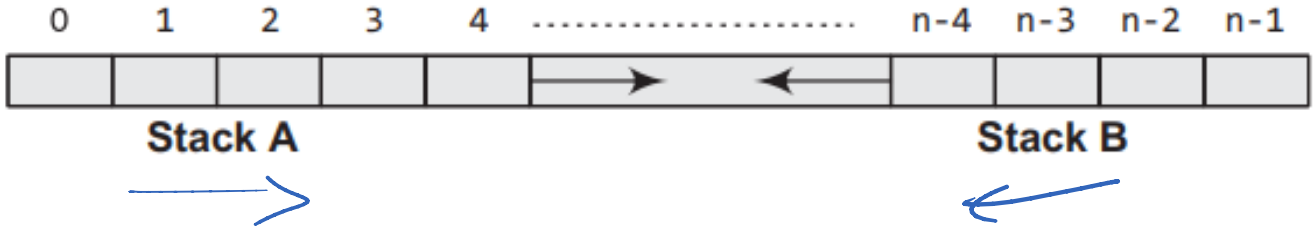
```
Peek(top)
{
   if(top==NULL)
      pf("empty");
   else
      return top->data;
}
```



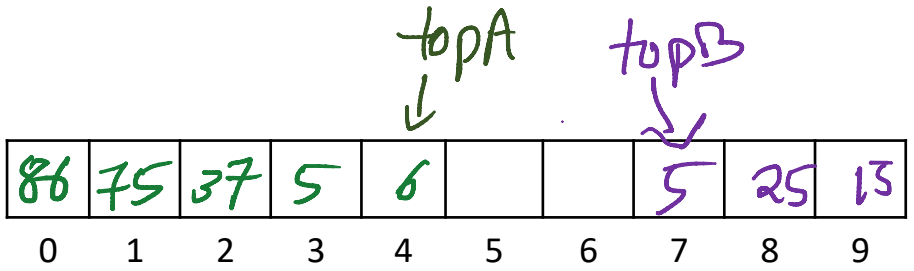

```
Display(top)
{ struct node *ptr;
  ptr = top;
  if(top==NULL)
    pf("empty");
  else
    while(ptr!=NULL)
    { printf("\n%d", ptr->data);
      ptr = ptr->next;
    }
}
```

MULTIPLE STACKS

- When we implemented a stack using an array, we had seen that the size of the array must be known in advance.
- If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered.
- In case, we allocate a large amount of space for the stack, it will result in sheer wastage of memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size.



topA++
① pushA 86,75,37,5,6,7
② popA(topA--) 3
③ display A



topB--
① push B
② pop B
③ display B
MAX

if(topA == -1)
↘ underflow

overflow
if(topA == topB-1)
if(topB == topA+1)

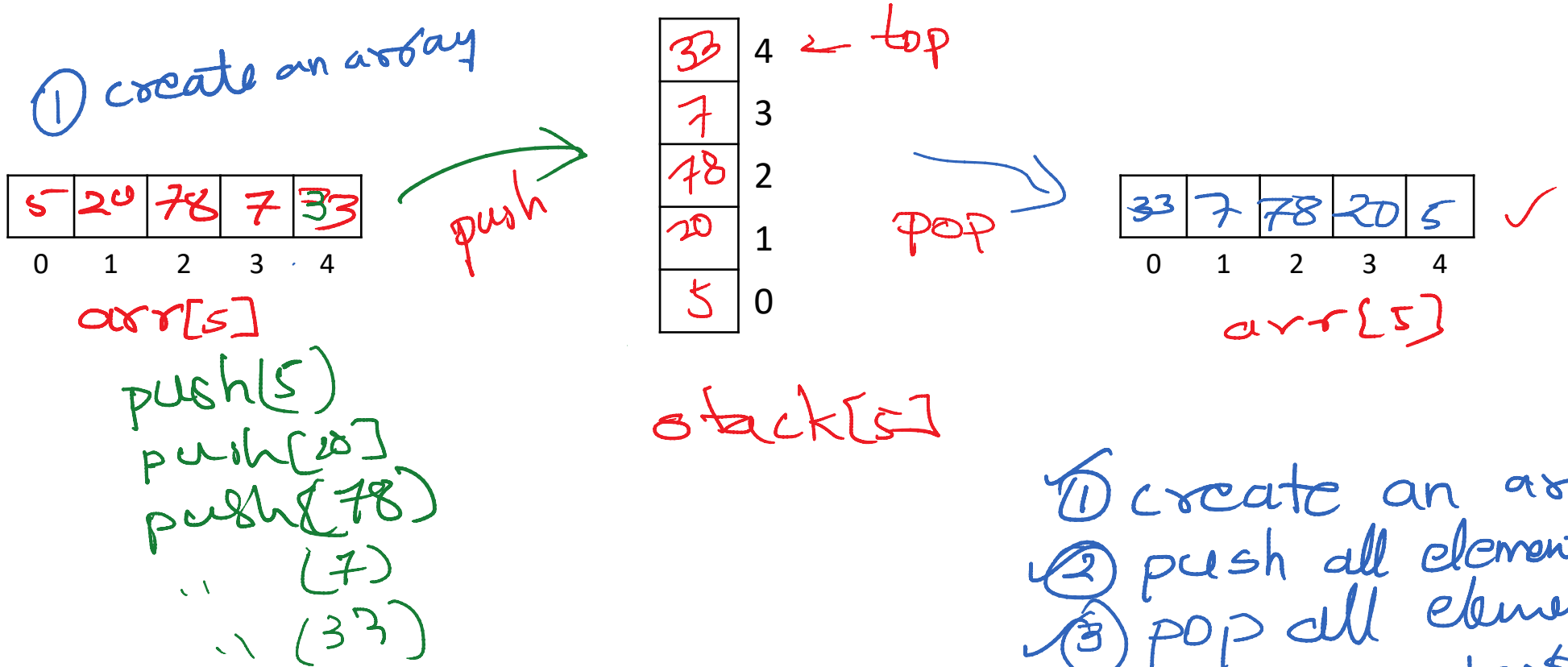
if(topB == MAX)
↘ underflow

APPLICATIONS OF STACKS

- 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

Reversing a List

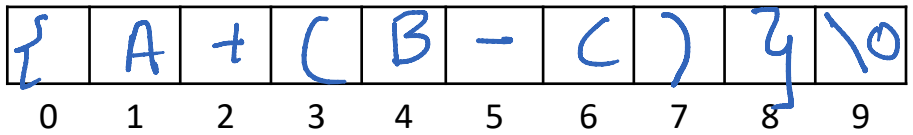
- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

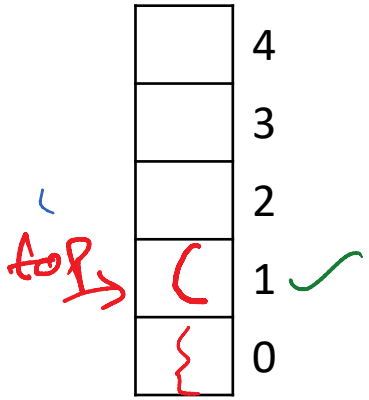


Implementing Parentheses Checker

- Stacks can be used to check the validity of parentheses in any algebraic expression.
- An algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.

char arr[20];





{ [(→ push into stack

}]) → pop from stack

compare

temp = pop( );

temp = '(';

if(arr[i] ==) && Val

flag = 1;

if(arr[i] == '(' || arr[i] == '{' || arr[i] ==

push(arr[i]);

Evaluation of Arithmetic Expressions

Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, A+B; here, plus operator is placed between the two operands A and B. A+B*C
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation (RPN)

- Postfix notation was given by Jan Lukasiewicz who was a Polish mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, 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.
- The expression (A + B) * C is written as: 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.

Prefix Notation (polish notation)

- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression (A + B) * C is written as: *+ABC in the prefix notation

	Infix		Postfix		Prefix
1.	X+Y	→	XY+	→	+XY
2.	X+Y*Z	→	XYZ*+	→	+X*YZ
3.	(X+Y)*Z	→	XY+Z*	→	*+XYZ
4.	(X+Y)*(P+Q)	→	XY+PQ+*	→	*+XY+PQ

Conversion of an Infix Expression into 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
           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 X 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 X.
               b.  Push the operator X 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: 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

```
Step 1:    Add ")" to the end of the postfix expression.
Step 2:    Scan every character of the postfix expression and repeat 3 and 4
           until ")" is encountered.
Step 3:    If an operand is encountered, push into stack.
           If an operator X is encountered, then
           a.  Pop the top two elements from the stack as A and B.
           b.  Evaluate B X 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
```

Example:

- 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.

Infix

9 – ((3 * 4) + 8) / 4

= 9 – (12 + 8) / 4

= 9 – 20 / 4

= 9 – 5

= 4

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

pop → 4 * 3 = 12 → push

pop → 8 + 12 = 20 → push

pop → 20 / 4 = 5 → push

pop → 9 - 5 = 4 → push

pop → 4

Conversion of an Infix Expression into a Prefix Expression

```
Let, X is an arithmetic expression written in infix notation. This algorithm finds the
equivalent prefix expression Y.
Step 1:    Reverse the infix expression.
Step 2:    Make Every " ( " as " ) " and every " ) " as " ( ".
Step 3:    Push " ( " onto Stack, and add " ) " to the end of X.
Step 4:    Scan X from left to right and repeat Step 3 to 6 for each element of X until the
           Stack is empty.
Step 5:    If an operand is encountered, add it to Y.
Step 6:    If a left parenthesis is encountered, push it onto Stack.
Step 7:    If an operator is encountered ,then :
           a.  Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
               which has the same precedence as or higher precedence than operator.
           b.  Add operator to Stack.
           [ End of If ]
Step 8:    If a right parenthesis is encountered ,then :
           a.  Repeatedly pop from Stack and add to Y each operator (on the top of Stack)
               until a left parenthesis is encountered.
           b.  Remove the left Parenthesis.
           [ End of If ]
Step 9:    Reverse the prefix expression.
Step 10:   Exit.
```

Example: Infix Expression : A + (B * C - (D / E ^ F) * G) * H

- Reverse the infix expression :
  H *) G *) F ^ E / D (- C * B (+ A
- Make Every " (" as ") " and every ") " as " ("
  H * (G * (F ^ E / D) - C * B) + A
- Convert expression to postfix form :
- Reverse the postfix expression :
  +A*-*BC*/D^EFGH

Scanned	Stack	Postfix Expression
infix	(	
H	(	H
*	(*	H
(	(*(	H
G	(*(	HG
*	(*(*	HG
(	(*(*(	HG
F	(*(*(	HGF
^	(*(*(^	HGF
E	(*(*(^	HGFE
/	(*(*(/	HGFE^
D	(*(*(/	HGFE^D
)	(*(*	HGFE^D/
-	(*(-	HGFE^D/*
C	(*(-	HGFE^D/*C
*	(*(-*	HGFE^D/*C
B	(*(-*	HGFE^D/*CB
)	(*	HGFE^D/*CB*-
+	(+	HGFE^D/*CB*-*
A	(+	HGFE^D/*CB*-*A
)	Empty	HGFE^D/*CB*-*A+

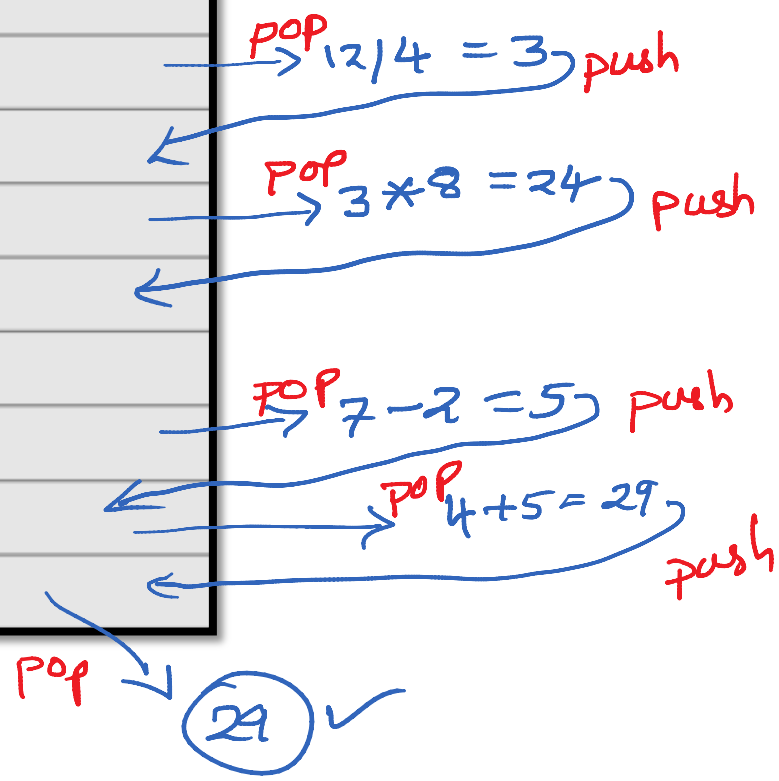
Evaluation of a Prefix Expression

```
Step 1:     Accept the prefix expression
Step 2:     Repeat until all the characters in the prefix expression have been
            scanned.
            a.  Scan the prefix expression from right, one character at a time.
            b.  If the scanned character is an operand, push it on the operand
                stack.
            c.  If the scanned character is an operator, then
                I.  pop two values from the operand stack.
                II. Apply the operator on the popped operands.
                III. Push the result on the operand stack.
Step 3:     EXIT
```

Example:

consider the prefix expression + – 9 2 7 * 8 / 4 12.

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29



Recursion

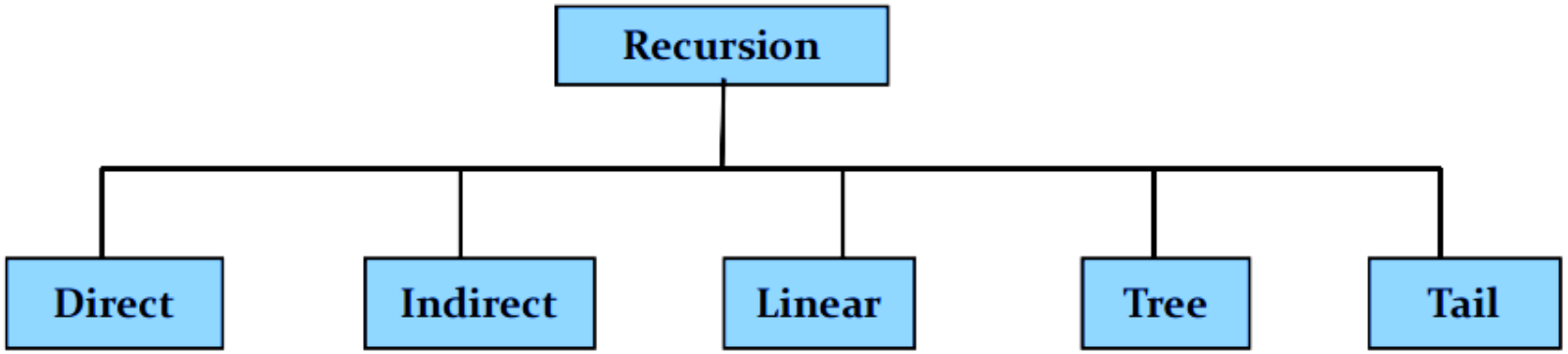
- A **recursive function** is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has **two major cases**. They are
  1. **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  2. **Recursive case**, in which first the problem at hand is divided into simpler **sub-parts**. Second the function **calls itself** but with sub-parts of the problem obtained in the first step. Third, the result is obtained by **combining the solutions of simpler sub-parts**.
- To understand recursive functions, let us take an example of calculating factorial of a number. To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.
- In other words, n! = nX(n–1)!
- Let us say we need to find the value of 5!

  5! = 5 X 4 X 3 X 2 X 1 = 120
- For the factorial function,
  - **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
  - **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)

```
// recursive program to implementation of  factorial of a number
#include<stdio.h>
 int Fact(int); // FUNCTION DECLARATION
int main()
 {
       int num, val;
       printf("\n Enter the number: ");
       scanf("%d", &num);
       val = Fact(num);
       printf("\n Factorial of %d = %d", num, val);
       return 0;
}
 int Fact(int n)
{
        if(n==1)
              return 1;
       else
             return (n * Fact(n–1));
}
```

Types of Recursion

- Any recursive function can be characterized based on:
  - Whether the function calls itself directly or indirectly (direct or indirect recursion).
  - Whether any operation is pending at each recursive call (tail-recursive or not).
  - The structure of the calling pattern (linear or tree-recursive).



Direct Recursion

- A function is said to be directly recursive if it explicitly calls itself.
- For example, consider the function given below.

```
int Func( int n)
{
        if(n==0)
              retrun n;
        return (Func(n-1));
}
```

Indirect Recursion

- A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it.
- Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
```

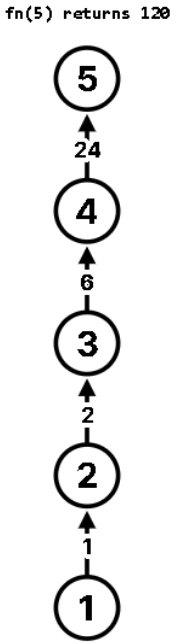
```
int Func2(int x)
{
        return Func1(x-1);
}
```

Linear Recursion

- Recursive functions can also be characterized depending on the way in which the recursion grows: in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be linearly recursive when no pending operation involves another recursive call to the function.
- For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to fact() function.

```
Int fun(n)
{
        if (n == 1)
                return 1;
        else
                return  n * fun(n-1);
}
```

recursion-visualization

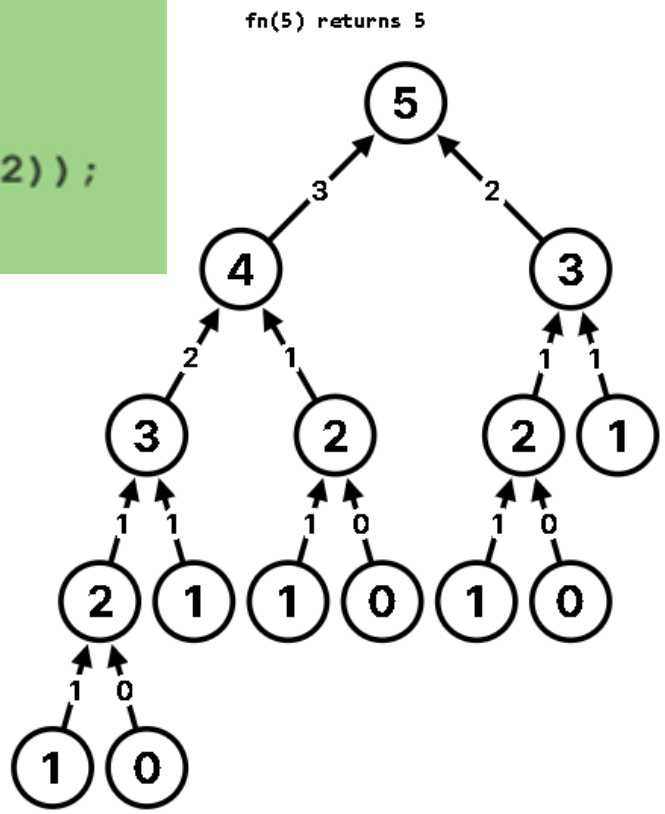


Tree Recursion

- A recursive function is said to be tree recursive (or non-linearly recursive) if the pending operation makes another recursive call to the function.
- For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

```
int Fibonacci(int num)
{
        if(num <= 2)
                return 1;
        return ( Fibonacci (num - 1) + Fibonacci(num - 2));
}
```

recursion-visualization

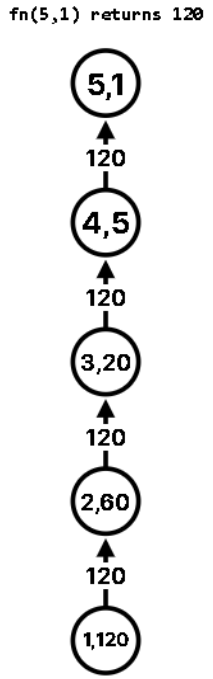


Tail Recursion

- A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

recursion-visualization



Pros and Cons of Recursion

Pros

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Cons

- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly when using global variables.