**The Stack**

1. **Concept and Definition**

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. Stacks are dynamic sets in which element deleted from the set is the one most recently inserted: the stacks implements a **last-in, first-out**, or **LIFO,** policy. We can represent stack digramatically as follows:



**Fig:** Stack containing five items

The insert operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP.The stack name is allusion to physical stacks, such as stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack , since only the top plate is accessible.

* + **Primitive Operations**

The value of stack are changed by two operation PUSH and POP which are called primitive operations of stacks. Given a stack s, and an item i, performing the operation push (s, i) adds the item i to the top of stack s. Similarly, the operation pop(s) removes the top element and returns it as a function value. Thus the assignment operations:

i = pop(s);

removes the element at the top of s and assigns the value to i.



**Fig:** PUSH operation

Here, PUSH(s,44) adds item 44 on top of stack s as shown in above figure.



**Fig:** POP operation

Here, POP(s) delete the item 44 which is stored on top of the stack. We don't need to pass any parameter for delete operation.

The primitive operations performed on the stack are summarized as follows:

**PUSH:** The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

**POP:** The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs. To avoid underflow, we need to check whether the stack is empty or not before each POP operation. ***ISEMPTY*** is used to check empty condition of stack. ***ISEMPTY*** returns *TRUE* if there is no item on stack otherwise it will return *FALSE*. ***ISFULL*** is used to check overflow condition. The illegal attempt to insert an item on a full stack is called overflow. In static implementation, we should check ***ISFULL*** condition before each PUSH operation.

Another operation that can be performed on stack is TOPOFSTACK. This operation is used to find the topmost element of stack. This is not a new operation and is equivalent to PUSH and POP operation. And it is achieved as

i=POP(s);

PUSH (s, i);

Here i will return the topmost element of the stack

**STACK IMPLEMENTATION**

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (i.e., memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity. The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

* + **Stack as an ADT**

Stack can be expressed as ADT as:

Here, eltype is used to denote the type of the stack element .

**abstract typedef** << eltype >> STACK (eltype);

**abstract** empty (s)

STACK (eltype) s;

**postcondition**  empty = = (len(s) == 0);

**abstract** eltype pop(s)

STACK (eltype) s;

***precondition***  4***ion***  pop == first(s,);

s == sub(s, , 1, len(s,) - 1);

***abstract*** push(s, elt)

STACK (eltype) s;

eltype elt;

***postcondition***  s == <elt> + s, ;

* + **Implementing PUSH and POP operation**

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

**Algorithm for push**

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let ITEM is the data item to be pushed.

1. If TOP = SIZE – 1, then:
   1. Display “The stack is in overflow condition”
   2. Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

**Algorithm for pop**

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then
   1. Display “The Stack is empty”
   2. Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit.

FULL PROGRAM

#include <stdio.h>

#include <ctype.h>

#include<stdlib.h>

#define size 100

struct stack{

int top;

int array[size];

};

int isempty(struct stack \*ps)

{

if(ps->top==-1)

printf("\n\tThe stack is empty");

else

printf("\n\tThe stack is not empty");

}

int isfull(struct stack \*ps)

{

if(ps->top==size-1)

printf("\n\t\tThe stack is full");

else

printf("\n\t\tThe stak is not full");

}

int PUSH(struct stack \*ps,int elt)

{

ps->top=ps->top+1;

ps->array[ps->top]=elt;

return elt;

}

int POP(struct stack \*ps)

{

int elt=ps->array[ps->top];

ps->top=ps->top-1;

return elt;

}

int TOS(struct stack \*ps)

{

int ele=ps->array[ps->top];

return ele;

}

int main()

{

struct stack ps;

int elt,choice;

char c;

ps.top=-1;

do{

printf("\n\tEnter\n\t1:PUSH\n\t2:POP\n\t3:TOS\n\t4:ISEMPTY\n\t5:ISFULL\n\t\t");

printf("\n\tmake your choice: ");

scanf("%d",&choice);

switch(choice)

{

case 1:

if(ps.top==size-1)

{

printf("\n\tThe stack is FULL");

}

else

{

printf("\n\tEnter the element to push: ");

scanf("%d",&elt);

printf("\n\t %d is pushed", PUSH(&ps,elt));

}

break;

case 2:

if(ps.top==-1)

{

printf("\n\tThe stack is empty");

exit(0);

}

else

printf("\n\t %d is popped",POP(&ps));

break;

case 3:

if(ps.top==-1)

{

printf("\n\tThe stack is empty");

exit(0);

}

else

{

printf("\n\t%d is tossed",TOS(&ps));

}

break;

case 4:

isempty(&ps);

break;

case 5:

isfull(&ps);

break;

}

printf("\n\tD you want to continue??");

printf("\n\tEnter Y to continue N to exit: ");

scanf("%s",&c);

}

while(toupper(c)=='Y');

return 0;

}

* + **Testing for overflow and underflow conditions**

If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs. To avoid underflow, we need to check whether the stack is empty or not before each POP operation. ***ISEMPTY*** is used to check empty condition of stack. ***ISEMPTY*** returns *TRUE* if there is no item on stack otherwise it will return *FALSE*. ***ISFULL*** is used to check overflow condition. The illegal attempt to insert an item on a full stack is called overflow. In static implementation, we should check ***ISFULL*** condition before each PUSH operation.

**Underflow Condition**

*if(s.top == -1)*

*/\* Stack is Empty\*/*

*else*

*/\* Stack is not Empty\*/*

**Overflow Condition**

*If TOP = SIZE – 1, then:*

*/\*Display “The stack is in overflow condition” \*/*

*else*

*/\*Display “The stack is not in overflow condition” \*/*

* **Explain, What are the Application of Stack ?**
  + Reversing a string
  + Expression Evaluation
  + Undo application in word processor
  + page visited history in web

1. **The Infix, Postfix and Prefix**

* **Concept and Definition**

**Arithmetic expression:** An expression is defined as a number of operands or data items combined using several operators. We can represent the expression in following three ways:

1. Infix
2. Postfix
3. Prefix
4. **Infix Notation: X + Y**

Infix notation is easy to read for *humans*. It is an ordinary mathematical notation of expression that we used in our daily life in which operator is written in between operands. For example: A + B Here A and B are operands and + is operator which comes in between operand.

1. **Postfix notation (also known as "Reverse Polish notation"): X Y +**

In postfix notation, operators are written after their operands. The infix expression A \* ( B + C ) / D is equivalent to A B C + \* D /

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "\*" in the example above, the addition must be performed before the multiplication.

1. **Prefix notation (also known as "Polish notation"): + X Y**

In postfix notation, operators are written before their operands. The expressions given above are equivalent to / \* A + B C D

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

( / ( \* A ( + B C) ) D )

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication). Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

**Notation Conversion:**

Let us take an example of following infix operation of 2 + 3 \* 3 . If we compute this we will get following result 5 \* 3 = 15. Is it right? No, So what is wrong? We should have knowledge about precedence rule to get correct result. Here multiplication have higher precedence than addition, so it should perform before addition.

**Operator Precedence:**

**Operator Symbol Precedence**

Exponential $ Highest

Multiplication /Division, \*, / Next Highest

Addition/Subtraction +, - Lowest

* + **Evaluating the postfix operation**

The reason to convert infix to postfix expression is that we can compute the answer of postfix expression easier by using a stack.We can evaluate a postfix expression using a stack. Each operator in a postfix string corresponds to the previous two operands . Each time we read an operand we push it onto a stack. When we reach an operator its associated operands (the top two elements on the stack ) are popped out from the stack. We then perform the indicated operation on them and push the result on top of the stack so that it will be available for use as one of the operands for the next operator .

The following algorithm is used to evaluate the postfix notation.

opndstk = the empty stack;

/\* Scan the input string reading one element at a time into symb \*/

while (not end of input) {

symb = next input character ;

if ( symb is an operand )

push(opndstk, symb);

else {

/\* symb is an operator\*/

opnd2 = pop (opndstk);

opnd1 = pop (opndstk);

value = result of applying symb to opnd1 and opnd2 ;

push(opndstk,value);

} /\* end else\*/

} /\* end while \*/

return (pop(opndstk));

The following example shows how a postfix expression can be evaluated using a stack.

Lets evaluate the following expression:

6 2 3 + - 382 / + \* 2 $ 3 +

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| symb | opnd1 | opnd2 | value | opndstk |
| 6 |  |  |  | 6 |
| 2 |  |  |  | 6, 2 |
| 3 |  |  |  | 6, 2, 3 |
| + | 2 | 3 | 5 | 6, 5 |
| - | 6 | 5 | 1 | 1 |
| 3 |  |  |  | 1, 3 |
| 8 |  |  |  | 1, 3, 8 |
| 2 |  |  |  | 1, 3, 8,2 |
| / | 8 | 2 | 4 | 1, 3, 4 |
| + | 3 | 4 | 7 | 1, 7 |
| \* | 1 | 7 | 7 | 7 |
| 2 |  |  |  | 7, 2 |
| $ | 7 | 2 |  | 49 |
| 3 |  |  |  | 49, 3 |
| + | 49 | 3 |  | 52 |

**C program to evaluate a postfix expression**

**# include<stdio.h>**

**# include<conio.h>**

**# include<math.h>**

**# define MAXCOLS 80**

**# define TRUE 1**

**# define FALSE 0**

**double eval(char[]);**

**double pop(struct stack \*);**

**double push(Struct stack\*, double);**

**int empty(struct stack \*);**

**int isdigit(char);**

**double oper(int, double, double);**

void main()

{

char expr[MAXCOLS];

int position= 0;

while((expr[position++] = getchar()) ! = ‘\n’);

expr[--position] = ‘\0’;

printf(“%s%s”, “the original postfix expression is”,expr);

printf(“\n%f”,eval(expr));

}//End main

struct stack{

int top;

double items [MAXCOLS];

};

double eval(char expr[])

{

int c,position;

double opnd1,opnd2,value;

struct stack opndstk;

opndstk.top=-1;

for(position=0; ( c= expr[position]) !=’\0’; position++)

{

if(isdigit(c))

/\* operand -- convert the character representation\*/

/\* of the digit into double and push it onto the stack \*/

push(&opndstk, (double) (c- ‘0’));

else{

/\* operator \*/

opnd2 = pop(&opndstk);

opnd1= pop(&opndstk);

value= oper ( c ,opnd1, opnd2);

push(&opndstk, value);

}//end val

int isdigit(char symb)

{

return(symb>=’0’ && symb < = ‘9’;

}

double oper(int symb, double op1, double op2)

{

switch(symb){

case ’+’: return(op1 + op2);

case ‘-’ : return ( op1-op2);

case ‘\*’ : return ( op1 \* op2 ) ;

case ‘/’ : return ( op1 / op2 );

case ‘$’ : return (pow(op1, op2));

default : printf(“%s”, “illegal operation”);

exit(1);

} //end switch

}// end oper

* + **Converting from infix to postfix**

**Algorithm for converting an expression from infix to postfix**

1. *opstk = the empty stack;*
2. *while ( not end of input ) {*
3. *symb = next input character*
4. *if ( symb is an operand )*

*add symb to the postfix string*

1. *else {*
2. *while (!empty(opstk) && prcd(stacktop(opstk), symb)){*
3. *topsymb = pop(opstk);*
4. *add topsymb to the postfix string;*

*}/\* End while\*/*

1. *push(opstk, symb)*

*} /\* End else\*/*

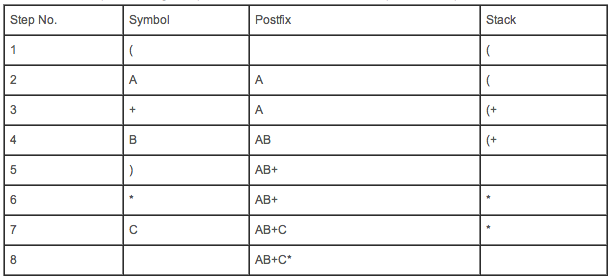
*} /\* End while\*/*

*/\* Output any remaining operators \*/*

1. *while(!empty(opstk)) {*
2. *topsymb = pop (opstk) ;*
3. *add topsymb to the postfix string;*

*Example:*

*The infix expression ( A + B ) \* C is converted into postfix expression as follows:*

*:*

1. **Recursion**
   * **Concept and Definition**

Recursion is a process, call or subroutine by which a function calls itself repeatedly, until some specified condition has been satisfied. Recursive algorithms can be done iteratively. Recursion is one of the most powerful tools in a programming language.

*Recurrence* From the Latin, re=back + currere = to run to happen again.

The process is used for repetitive computations in which each action is stated in terms of a previous result. problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

The recursion process has two parts;

**1) The Base Case**

In order for recursion to work correctly, every recursive method must have a base case. The base case is an input for which the method does not make a recursive call. The base case prevents infinite recursion.

**2) Recursive Case :**

Recursively divide the problem into subproblems that are similar in behaviour and smaller in size . Finally combine each solved subproblem to get the final result. These are similar to divide and conquer paradigm.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

A recursive Example

**int** sum(**int** n){  
 **if**(n==0)  
 **return** n;  
 **else**  
 **return** n+sum(n-1); /\*self call to function sum() \*/  
}

Here, the sum of 0 number is 0 which is the base case. It recursively calculate the sum by calling itself.

* + **Implementation of:**
* **Multiplication of Natural Numbers**

The counting number from 0 upward is called natural numbers. For ex. 1, 2, 3….. In an iterative way , the product a \* b, where a and b are positive integers, may be defined as a added to itself b times. An equivalent recursive definition is :

a \* b = a if b =1; // Base condition

a \* b = a \* (b-1) + a if b>1 //Recursive definition

To evaluate 5 \* 4 , we first evaluate 5 \* (4-1) i.e 5 \* 3,

To evaluate 5 \* 3 , we first evaluate 5 \* (3-1) i.e 5 \* 2,

To evaluate 5 \* 2 , we evaluate at first 5 \* (2-1) i.e 5 \* 1, it reaches base condition and can not be divided further.

5 \* 4 = 5 \* 3 + 5 // 5 \* ( 4 - 1 ) = 5 \* 3,

= 5 \* 2 + 5 + 5 // 5 \* ( 3 - 1 ) = 5\*2

= 5 \* 1 + 5 + 5 +5

= 20

**Q: Write a menu driven program to find product of natural numbers by using recursive method and iterative method.**

* **Factorial**

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), the factorial of a [non-negative integer](http://en.wikipedia.org/wiki/Non-negative_integer) *n*, denoted by *n*!, is the [product](http://en.wikipedia.org/wiki/Product_(mathematics)) of all positive integers less than or equal to *n*. For example,

5! = 5  \times  4  \times  3  \times  2  \times  1 = 120.  \ 

Factorial of any number can be calculated by both iterative and recursive method.

**Iterative method**

int fact( int n) {

int prod = 1; // 0!= 1 Base Condition

for (int k = 1; k <= n; k++)

prod = prod \* k; return prod;

}

**Recursive Method**

int fact( int n ) {

// precondition: n >= 0

if ( n = = 0 )

return 1;

return n \* fact( n-1 );

}

* **Fibonacci Sequences**

In mathematics, the Fibonacci numbers, or Fibonacci series, are the numbers that are in the following sequence:

0,1,1,2,3,5,6,13,21,34,55,89,…

The first number in the Fibonacci sequence is 0, the second number is 1. The subsequent number is the result of the sum of the previous two e.g., third number 1 = 1+0, fourth number 2=1+1, fifth number 3 = 2+1, etc.

The Fn number is defined as follows:

Fn = Fn-1 + Fn-2,

with the base values:

F0 = 0, F1 = 1.

Fibonacci problem can be solved by both iterative and recursive technique. Here we are going to analyze both technique as follows.

1. **Iterative Method**

The repetition of same process again and again is called iteration. Iteration solves a problem by repeatedly working on successive parts of the problem until some specified condition is met. Here in fibonacci series, we start generating next values by adding two previous values as shown in following iterative function

int IterativeFibonacci( int n)

{

int i;

int f1 = 0;

int f2 = 1;

int fi ;

if ( n = = 0 )

return 0 ;

if( n = = 1)

return 1 ;

for ( i = 2 ; i < = n ; i++)

{

fi = fi + f2 ;

f1 = f2;

f2 = fi;

}

return fi;

}

1. **Recursive Method**

In the fibonacci recursive method, the base cases are n = 0 and n = 1. for other input of n, it recursively call itself until base condition is arises. It is summarized as

fibo ( 0 )

int RecursiveFibonacci(int n)

{

/\* base case \*/

if ( n == 0 )

return 0;

if ( n == 1 )

return 1;

/\*recursive definition \*/

return ( RecursiveFibonacci(n-1) +

RecursiveFibonacci(n-2));

}

* **The Tower of Hanoi**

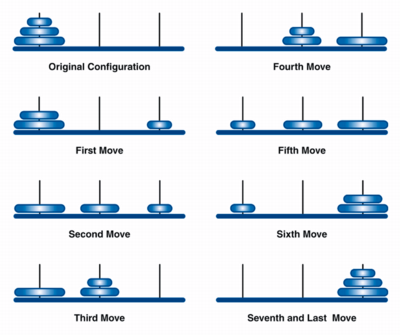
The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is 2*n* - 1, where *n* is the number of disks, as shown below.

A B C A B C



Suppose that we could move three disc from peg A to peg C. Then we could use peg B as auxiliary to make a move n-1 disks from peg A to peg C. We could then move the largest disk from A to C. Again, we could use peg A as auxiliary to make a move n-2 disks from peg B to A. We could then move n-2, second largest, disk to peg C. We may state a recursive solution to the towers of Hanoi problem as follows

**Algorithm for Towers of Hanoi Problems**

To move n disks from A to C, using B as auxiliary:

1. if n = = 1, move the single disk from A to C and stop
2. Move the top n - 1 disks from A to B , using C as auxiliary.
3. Move the remaining disk from A to C
4. Move the n - 1 disks from B to C , Using A as an Auxiliary

**A C program to implement Tower of Hanoi**

#include<stdio.h>  
#include<conio.h>  
void TowerOfHanoi(int n,char x,char y,char z);  
void main()  
{  
 int n;  
 printf("nEnter number of plates:");  
 scanf("%d",&n);  
 TowerOfHanoi(n-1,'A','B','C');  
 getch();  
}

void TowerOfHanoi(int n,char x,char y,char z)  
{  
 if(n>0)  
 {  
 TOH(n-1,x,z,y);  
 printf("n%c -> %c",x,y);  
 TOH(n-1,z,y,x);  
 }  
}