

A ■ P ■ I ■ I ■ T

ASIA PACIFIC INSTITUTE OF
INFORMATION TECHNOLOGY

STACK

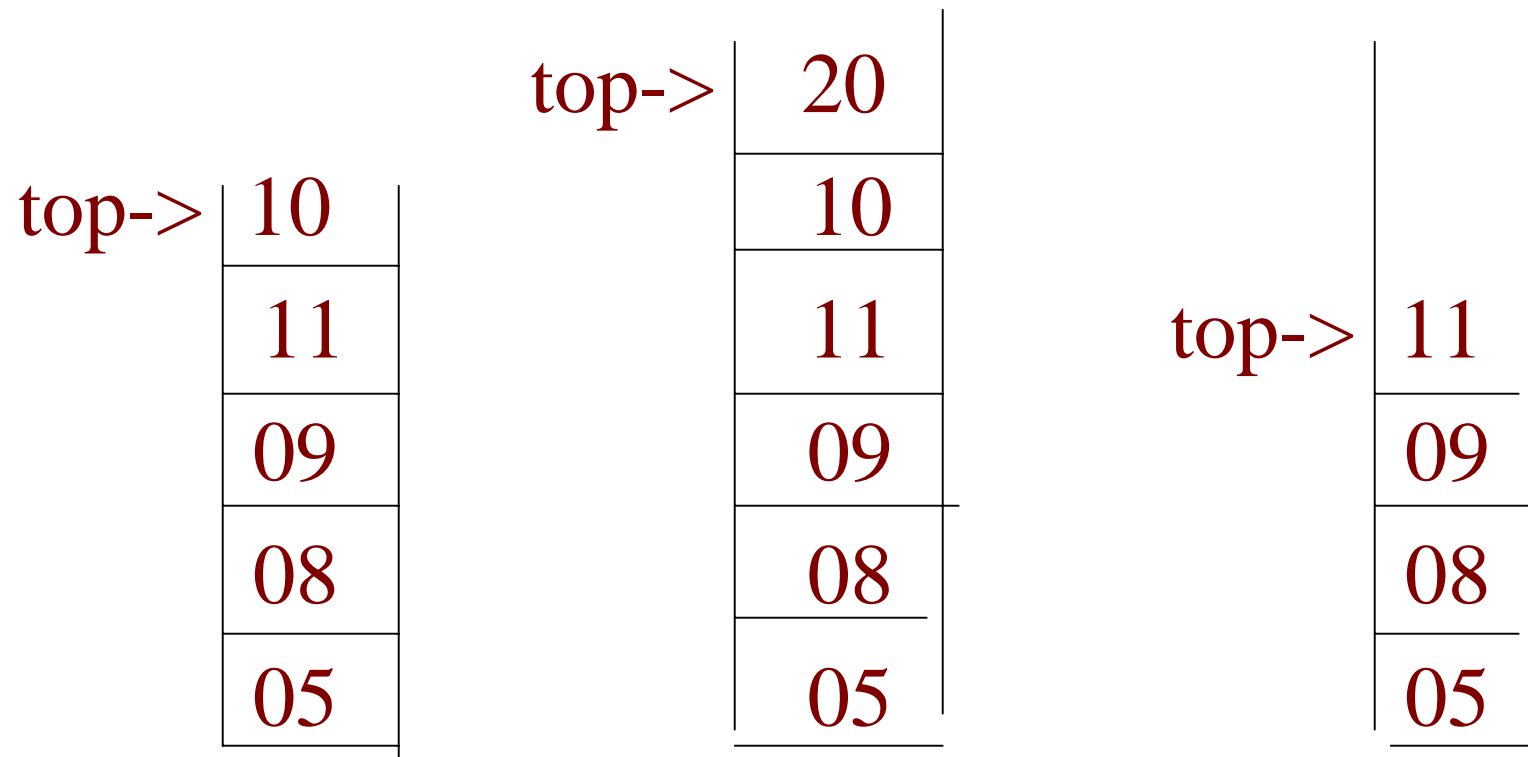
DEFINITION

- Stack is an ordered collection of items
- A new item is entered on top of the stack an item is removed from the top of the stack.
- Always current TOP element is removed and a new item is placed on top of the current top element.
- A stack grows/ shrinks as items are added / deleted

Initial Stage

Add[20]

Del [20,10]



Real Life Application using Stack :

- * Dinner Plates stacked one over the other
- * Boxes kept in a godown
- * Shuttling the carriages in a Railway Station

PRIMITIVE OPERATIONS ON A STACK

PUSH

Takes two values. It places an item on top of a stack.

Eg. `push(s,H)`

`Push(s,I)`

`push(s,J)`

were “s”

POP

Takes two values. It places an item on top of a stack.

Eg. `pop(s1)`

`pop(s2)`

`pop(s3)`

The assignment statement `x = pop(s)` , removes the TOP element of stack “s” and assign it to the variable x.

stacktop :

Examines the topmost element of a stack
without removing the element

E.g. $x = \text{stacktop}(s) \implies x = \text{pop}(s)$
 $\text{push}(s, x)$

At the time of creating a stack, it will not have any element and hence will be empty - Empty stack
pop / stacktop operation cannot be applied on an empty stack. The stack should be checked to ensure it is non-empty before performing any of these operations.

empty :-

empty(s) returns TRUE if the stack s is empty. It returns FALSE otherwise.

An application of STACK :

PARENTHESES CHECKING

$$7 - ((x * ((x + y) / (j - 3) + y) / (4 - 2.5))$$

- Every right parenthesis should be preceded by a matching left parentheses
- # right parenthesis should be equal to the # left parentheses.

Invalid Expressions :

$((A+B)$

$A+BC$

$)A+BC-C$

$(A+C))-(C+D$

Parenthesis count :-

The parenthesis count is a particular point in an expression is the # left parenthesis & # right parenthesis that have been encountered in scanning the expression from its left end up to that point .

- 1) parenthesis count is at each point in the expression should be non -ve.
- 2) parenthesis count is at the end of the expression should be ϕ .

Delimiter Checking - an example

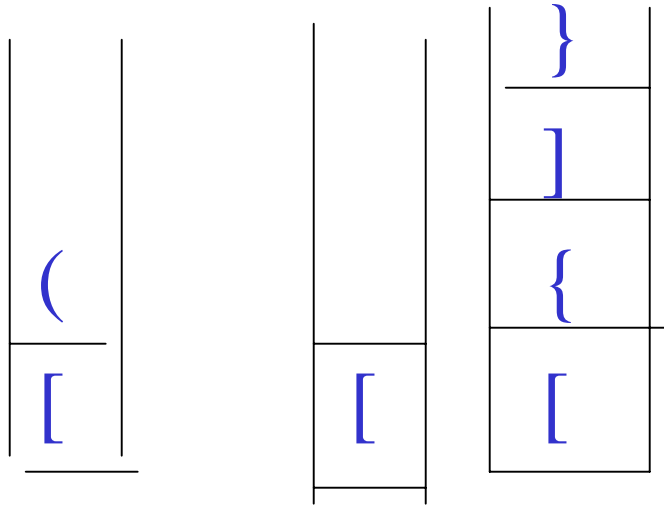
Instead of parenthesis, consider three delimiters ‘()’, ‘[]’ and ‘{ }’

A scope ender must be of the same type as its scope opener

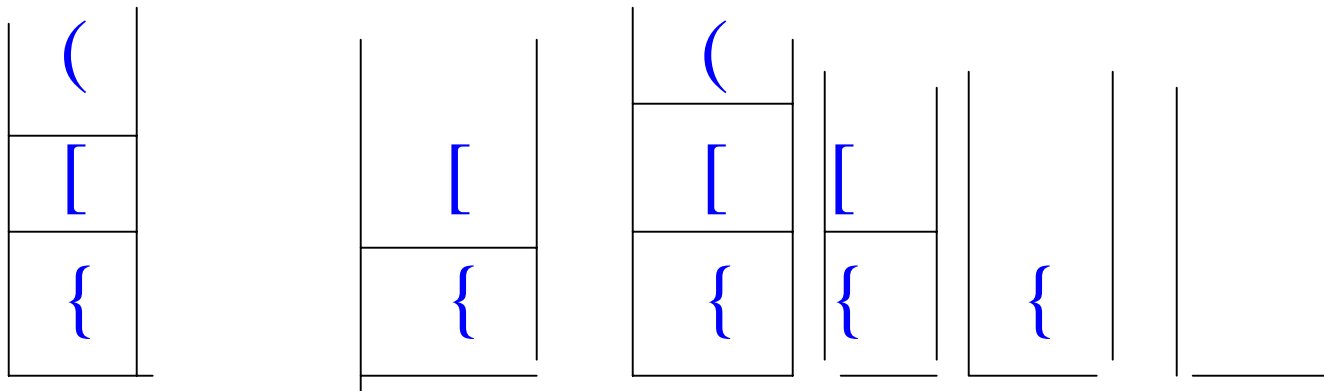
Invalid Cases : $(A+B) = [(A+B), \{A-CB\}]$

Whenever a scope opener is encountered, it is pushed onto the stack. Whenever a scope ender is encountered, the stack is examined.

$[(A+B), \{A-CB\}]$



$\{[(A+B)+(C/D)]\}$



If the stack is empty, the scope ender does not have a matching opener and hence the string is invalid.

If the stack is nonempty, we pop the stack and check whether the current top elements matches with the scope ender.

If it matches, we continue
otherwise the string is invalid.

When the end of the string is reached, the stack must be empty. Otherwise the string is invalid.

Algorithm :

valid = true;

s=the empty stack;

while(we have not read the entire string)

{ read the next symbol(symb) of the string;

push(s,symb);

if(symb== '(' || '[' || '{')

push(s,symb);

if (symb== ')' || ']' || '}')

if(empty(s)) valid=false;

```
else{ I=pop(s);  
    if ( I is not the matching opener of symb)  
        valid = false;  
    }  
} /* End while  
if(!empty(s))  
    valid = false;  
if(valid)  
    printf(“%s”,”The string is valid”);  
else  
    printf(“%s”,”The string is invalid”);
```


Implementation of Stack Using C

array - Holds fixed # elements

stack- Holds variable # elements

grow and shrink

size changes after each push or pop operation

a dynamic object

Stack is implemented using two object

- An array to store the elements in the stack
- An integer variables to include current top position of the stack [Both these can be included in a struct.



```
#define MAX_STACK_SIZE 100
```

```
struct stack{  
    int top_of_stack;  
    int stack_elements[MAX_STACK_SIZE ];  
};  
stack s1;
```

For empty stack, top_of_stack = -1;

Operation to test whether a stack is Empty or Not:

```
empty(ps)
```

```
struct    stack *ps;
```

```
{    if(ps -> top_of_stack == -1)
```

```
        return (TRUE);
```

```
    else return (FALSE);
```

```
};
```

```
main()    {
```

```
    if(empty(&s1))
```

```
        :
```

```
    else
```

```
    }
```

IMPLEMENTATION OF POP OPERATION

pop pops out the current - top element from a Stack

- cases like attempting operation on empty stack need to be checked

Pop can be implemented as a function with the following operation

* Generate a warning message and halts execution if the stack is empty

Otherwise

- * Returns the current top element of the stack
- * Removes the current top element of the stack

```

pop(ps);
struct    stack *ps;
{   if(empty(ps) ) {
        printf(“%s”,”UnderFlow”);
        exit(1);  }
return (ps -> stack_element[ps -> top_of_stack --]);
}
:
main() {
    :
    x=pop(&s1);
    :
    }

```

If we wish to continue the program without halting then

`x= pop(&s1)` should be replaced by

`if(!empty(&s1))`

`x=pop(&s1);`

PUSH Operation

```
push(ps,x)
```

```
struct stack *ps;
```

```
int x;
```

```
{
```

```
    if(ps -> top_of_stack == MAX_STACK_SIZE-1
```

```
        { printf(“%s “, “OverFlow”);
```

```
            exit(1);
```

```
    else
```

```
        ps -> stack_element[++(ps->top_of_stack)]=x;
```

```
    return;    }
```

Notations

Infix

$A + B$

Postfix

$AB+$

prefix

$+AB$

INFIX

$A+B-C$

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

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

$((A+B)*C-(D-E)) \$ (F+G)$

$A-B/(C*D\$E)$

PREFIX

$- +ABC$

$*+AB-CD$

$+-* \$ABCD//EF+GH$

$\$-*+ABC-DE+FG$

$-A/B*C\$DE$

INFIX

$A+B$

$A+B-C$

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

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

$((A+B)*C-(D-E)) \$(F+G)$

$A-B/(C*D\$E)$

POSTFIX

$AB+$

$AB+C-$

$AB+CD-*$

$AB\$C*D-EF/GH+/+$

$AB+C*DE -- FG+\$$

$ABCDE\$*/-$

Note : Infix notation is NOT the mirror image of
Postfix notation

Procedure for Evaluating Postfix Expression :

```
opstk=empty stack;  
while ( not end of input)  
{ symb = next input character;  
  if(symb is an operand)  
    push(opstk, symb)  
  else {    opnd2 = pop(opstk);  
          opnd1 = pop(opstk);  
          value  = result of applying symb to opnd1  
                  and opnd2;  
          push(opstk,value);  
        }  
}  
return (pop(opstk));
```

Example: 623 + - 382 / + *2 \$ 3+

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

Evaluation of Postfix Expression :

Assumptions :

- operands are single non -ve digits
- input string ends with the end_of_line char '\n'

The operands read as characters need to be converted to numbers

Eg.

Characters '8' need to be converted to be number 8

This can be done by using the property of collating sequence.

Eg. In C, the expression I-'0' gives the value stored in variable i

An operation in the expression can be implemented using a function `oper` with the char representation of an operator and two operands as arguments and returns the value of the sub-expression.

[Click Here to see an Algorithm :](#)

Converting an Expn. from INFIX into POSTFIX

Example :

Infix

$A+B*C$

$(A+B)*C$

Postfix

$ABC*+$

$AB+C*$

Note : Order of operands in both expressions are same.

How to maintain the precedence?

Define a function $\text{prcd}(\text{op1}, \text{op2})$ where op1 and op2 are operators.

$\text{prcd}(\text{op1}, \text{op2})$ returns TRUE if op1 has precedence over op2 . _____

Example :

`prcd('*', '+')` returns TRUE

`prcd('+', '+')` returns TRUE

`prcd('+', '*')` returns FALSE

Two cases

1) without parentheses

2) With parentheses

[Click Here to see an Algorithm :](#)

At any time, an operator on the stack has a higher precedence than all elements below it:

How to accommodate parenthesis ?

Push all opening parenthesis on to the stack and pop out when the closing parenthesis are encountered.

How is this done?

For any operator 'op' other than ')' make
`prcd(op,'(')` return **FALSE** and `prcd('(' ,op)` return
FALSE

How to pop out ‘(‘ and all elts within its scope upon reaching a ‘)’?

Define **prcd(op, ‘)’)** return **TRUE** for any ‘op’ other than ‘(‘

Let **prcd(‘(‘, ‘)’)** return **FALSE**

How to skip ‘(‘ ?

Replace **push(opstk, symb)** by

if (**symb != ‘(‘**)

push(opstk, symb);

else

topsymb = pop(opstk);

Precedence rules for ')' and '('

$\text{prcd}('(', \text{op})$ FALSE

$\text{prcd}(\text{op}, '(')$ FALSE for any operator other than ')'

$\text{prcd}(\text{op}, ')')$ TRUE for any operator other than '('

$\text{prcd}(')', \text{op})$????

THE END OF II LESSON