STACKS

A stack is a list of elements in which an element may be inserted or deleted only at one end called the top of the stack. also called LIFO list

Basic operations performed on stack

① "Push" is the term used to insert element into a stack.

② "Pop" is the term used to delete an element from a stack.

Note: Stacks are frequently used to indicate the order of processing of data when certain steps of the processing must be postponed until other conditions are fulfilled.

ARRAY REPRESENTATION OF STACKS

→ Each of our stack will be maintained by a linear array STACK

→ A pointer var. TOP which contains the location of the top element

→ A var MAXSTK which gives the max. no. of elements that can be held by the stack.

→ The condition TOP=0 or TOP=NULL indicates that the stack is empty.

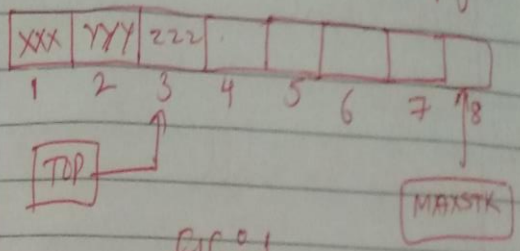


FIG: 1

Procedure 1: PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack

STEP 1: [Stack already filled? OVERFLOW cond.]
If TOP = MAXSTK, then
Print "OVERFLOW"
Return

STEP 2: Set TOP := TOP+1

STEP 3: [Inserts ITEM in new TOP position]
STACK[TOP] := ITEM

STEP 4: Return

Procedure 2:- POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK & assigns it to the variable ITEM

STEP 1: [Stack has an item to delete? UNDERFLOW cond.]
If TOP = 0 then
Print "Underflow" Goto

STEP 2 : SET ITEM := STACK[TOP]
STEP 3 : Set TOP := TOP-1
STEP 4 : Return.

Example

Consider the stack

Case 1 Suppose ITEM = WWW have to pushed onto the stack

1. Since TOP = 3 control is transferred to step 2
2. TOP = 3+1 = 4
3. STACK[TOP] = STACK[4] = WWW
4. Return

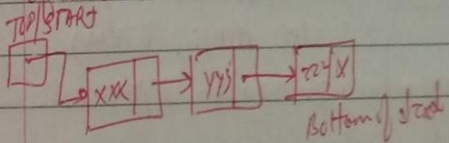
Case 2

Suppose ITEM is to be poped from sta

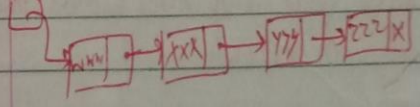
1. Since TOP = 3 control is transferred to step 2
2. ITEM = 222
3. TOP = TOP-1 = 3-1 = 2
4. Return

LINKED REPRESENTATION OF STACKS

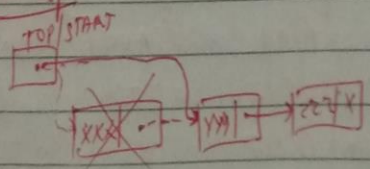
The linked representation of a stack, commonly learned linked stack is a stack ie is implemented using a singly linked list.
→ The START pointer of the linked list behaves as the TOP pointer var. of the stack & the null pointer of the last node in the list signals the bottom of the stack.



PUSH operation push WWW



POP op



In case of array

For Overflow :- (in case of push)
(TOP = MAXSTK)
For underflow :- (in case of pop)
(TOP = 0)

Linked List is free of these requirements

Procedure 3: PUSH_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM)
This procedure pushes an ITEM into a linked stack.

STEP 1: [Available Space?] If AVAIL = NULL then
Write "OVERFLOW"
Exit
STEP 2: [Remove first node from AVAIL list]
Set NEW := AVAIL and AVAIL := LINK[AVAIL]
STEP 3: Set INFO[NEW] := ITEM
STEP 4: Set LINK[NEW] := TOP [new node points to the original new node]
STEP 5: Set TOP = NEW [Reset TOP to point to the new node at the top of the stack]
STEP 6: Exit

Procedure 4: POP_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM)
This procedure deletes the top element of a linked stack and assigns it to the variable ITEM.

STEP 1: [Stack has an item to be removed?]
If TOP = NULL then
Write: UNDERFLOW & Exit
STEP 2: Set ITEM := INFO[TOP]
STEP 3: Set TEMP := TOP and TOP = LINK[TOP]
STEP 4: Set LINK[TEMP] = AVAIL & AVAIL = TEMP
STEP 5: Exit

Applications of STACKS
Stacks are frequently used in evaluation of arithmetic expressions

① Reversing a list
This can be accomplished by pushing each character onto the stack as it is read. When the line is finished, characters are popped off the stack - they come off in reverse order.

② Polish Notation

The process of writing the operators of an expression either before their operands or after them is called the Polish notation.
The notation refers to these arithmetic expressions in three forms:

ⓐ Prefix notation: If the operator symbols are placed before its operands then exp. is in prefix notation
For eg A+B ⟹ +AB

ⓑ Postfix notation: If the operator symbols are placed after its operands, then the exp. is in postfix notation
Eg AB+

ⓒ Infix notation: If the operator symbols are placed b/w the operands the the exp is in infix notation

Examples

Infix	Prefix	Postfix
A+B	+AB	AB+
(A-C)*B	*-ACB	AC-B*
A+(B*C)	+A*BC	ABC*+
(A+B)/(C+D)	/+AB+CD	AB+CD+/
(A+(B*C))/(C-(D*B))	/+A*BC-C*DB	ABC*+CDB*-/

Conversion of Infix to Postfix Expression

Evaluation order acc. to which the operations are executed:
- Brackets or Parentheses
- Exponentiation
- Multiplication or Division
- Addition or Subtraction

The operators with the same priority are evaluated from left to right.

Algorithm to Convert Infix Expression to Postfix expression

Q → Infix expression
P → equivalent postfix expression

The algo. uses a stack to temporarily hold operators & left parentheses

Page No.
Date: / /

Algorithm 6: POLISH(Q, P)

Q → arithmetic exp in infix notation.
P → equivalent postfix exp.

1. Push "(" onto STACK and add ")" to the ~~left parenthesis~~ end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the stack is empty.
3. If an operand is encountered, add it to P
4. If a left parenthesis is encountered, put it onto STACK.
5. If an operator (X) is encountered then:
   (a) Repeatedly pop from STACK and add to P each operator which has the same precedence as or higher precedence than (X).
   (b) Add (X) to STACK.
   [End of if structure]
6. If a right parenthesis is encountered then
   (a) Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered.
   (b) Remove the left parenthesis.
   [End of If structure]
   [End of Step 2 loop]
7. Exit

Page No.
Date: / /

Example

Transform Q into its equivalent postfix exp P.

Q: A + (B * C - (D / E ↑ F) * G) * H)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Sol: The elements of Q have been labelled from left to right for easy reference.

→ Each operand is simply added to P & does not change STACK
→ The subtraction (-) op in row 7 sends * from STACK to P before it (-) is pushed onto STACK
→ The right parenthesis in row 13 sends ↑ & then / from STACK to P, and then removes the left parenthesis from top of STACK
→ The right parenthesis in row 20 sends * & then + from STACK to P & then removes the left parenthesis from top of STACK.

After step 20 is executed the STACK is empty & we get P

	Symbol Scanned	Stack	Expression P
1	A	(	A
2	+	(+	A
3	(	(+(	A
4	B	(+(	AB
5	*	(+(*	AB
6	C	(+(*	ABC
7	-	(+(-	ABC*
8	(	(+(-(	ABC*
9	D	(+(-(	ABC*D
10	/	(+(-(/	ABC*D
11	E	(+(-(/	ABC*DE
12	↑	(+(-(/↑	ABC*DE
13	F	(+(-(/↑	ABC*DEF
14	)	(+(- Step 6	ABC*DEF↑/
15	*	(+(-*	ABC*DEF↑/
16	G	(+(-*	ABC*DEF↑/G
17	)	(+ Step 6	ABC*DEF↑/G*-
18	*	(+*	ABC*DEF↑/G*-
19	H	(+*	ABC*DEF↑/G*-H
20	)	—	ABC*DEF↑/G*-H*+

Evaluation of a Postfix Expression

Algo 5.9: This algo finds the VALUE of an arithmetic exp P written in Postfix notation

1. Add a right parenthesis ')' at the end of P.
2. Scan P from left to right & repeat 3 & 4 for each element of P until the ')' is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator (X) is encountered then
   (a) Remove the top two elements of STACK where A is the top element & B is the next-to-top element.
   (b) Evaluate B X A
   (c) Place the result of (b) back on STACK
   [End of If structure]
   [End of Step 2 Loop]
5. Set VALUE equal to top element on STACK
6. Exit

Example:

P: 5 6 2 + * 12 4 / -)

1 2 3 4 5 6 7 8 9 10

	Symbol Scanned	STACK
1	5	5
2	6	5, 6
3	2	5, 6, 2
4	+	5, 8
5	*	40
6	12	40, 12
7	4	40, 12, 4
8	/	40, 3
9	-	37
10	)	

VALUE = 37

③ Recursion

Recursive Procedures: If a procedure contains either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure. Then such a procedure is called a recursive procedure.

Ex: the problem of factorial can be solved using a recursive procedure.

Properties of recursive procedures:

(i) There must be certain criteria called the base criteria for which the procedure does not call itself.

(ii) Each time the procedure does call itself it must be closer to the base criteria.

A recursive fn is the one that refers to itself again & it is said to well-defined when it has the following properties:

(i) There must be certain arguments called base values for which the fn does not refer to itself.

(ii) Each time the fn does refer to itself the arg. of the fn must be closer to a base value.

Example Find 4! using recursion

(I) $4! = 4 \cdot 3!$

(II) $3! = 3 \cdot 2!$

(III) $2! = 2 \cdot 1!$

(IV) $1! = 1 \cdot 0!$

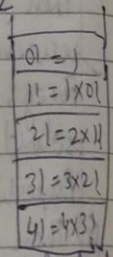
(V) $0! = 1$

(VI) $1! = 1 \cdot 1 = 1$

(VII) $2! = 2 \cdot 1 = 2$

(VIII) $3! = 3 \cdot 2 = 6$

(IX) $4! = 4 \cdot 6 = 24$



From step 6-9 We backtrack

This type of postponed processing leads itself to the use of stacks

Recursion

Example (6.15)

Let a and b denote +ve integers. Suppose a function Q is defined recursively as follows:

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b,b)+1 & \text{if } b \le a \end{cases}$$

(a) Find the value of $Q(2,3)$ & $Q(14,3)$

(b) What does this function do? Find $Q(5861,7)$

Sol:

(a) $Q(2,3) = 0$ since $2 < 3$

$Q(14,3) = Q(11,3)+1$

$= [Q(8,3)+1]+1 = Q(8,3)+2$

$= [Q(5,3)+1]+2 = Q(5,3)+3$

$= [Q(2,3)+1]+3 = Q(2,3)+4$

$= 0+4 = 4$

(b) Each time b is subtracted from a, the value of Q is increased by 1. Hence $Q(a,b)$ finds the quotient when a is divisible by b. Thus,

$Q(5861,7) = 837$