Stacks

Outline I

Stacks I

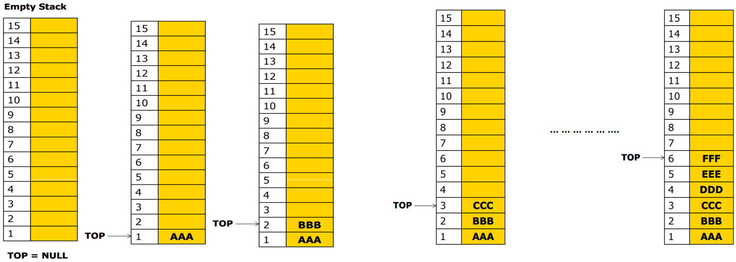
- A stack is a list of elements in which an element can be inserted or deleted only at one end.
- The end is referred to as the “top of stack”.
- So elements are removed from the stack in the reverse order of that in which they were inserted into the stack.
- This way a stack is a LIFO (Last in First Out) or FILO (First in Last Out)data structure.

Stack Operations I

- Special terminology is used to refer to the two basic operations associated with stack:
  - **PUSH:** is the term used to insert an element into the stack.
  - **POP:** is the term used to delete an element from the stack.

Stack Operations: Example I

- Say following six elements are pushed in order onto an empty stack: AAA, BBB, CCC, DDD, EEE, FFF
- Following figures depict the above operations:

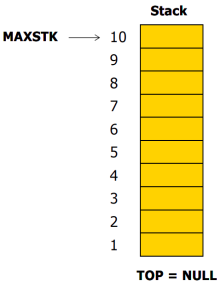


Array Representation of Stacks I

- Can be maintained using arrays or linked list.
- We shall discuss discuss array representation representation of stack.
- Array representation requires following:
  - A linear array named as **STACK**.
  - A pointer variable **TOP** which contains the location of the top element of the stack.
  - A variable **MAXSTK** which gives the maximum number of elements that can be held by the stack.

Array Representation of Stacks II

- The condition **TOP == 0** or **TOP == NULL** will indicate that the stack is empty.



Operations on Stack

- Operations: PUSH and POP.
- We have already already looked into them. So, its time to discuss them formally.

Algorithm for PUSH

- Following algorithm pushes (inserts)ITEM into STACK.
  PUSH(STACK, TOP, MAXSTK, ITEM)
  This procedure pushes an ITEM onto a stack.
  1. [Stack already filled?]
     If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
  2. Set TOP := TOP + 1. [Increases TOP by 1.]
  3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
  4. Return.

Algorithm for POP

- Following algorithm pops (deletes) ITEM from the STACK.

  POP(STACK, TOP, ITEM)

  This procedure deletes the top element of STACK and assigns it to the variable ITEM.

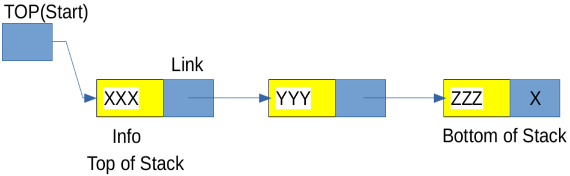
  1. [Stack has an item to be removed?]
     If TOP = 0, then: Print: UNDERFLOW, and Return.
  2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
  3. Set TOP := TOP - 1. [Decreases TOP by 1.]
  4. Return.

Linked Representation of Stacks I

- Can be maintained using one-way list or singly linked list.
- Linked representation requires following:
  - The **INFO** fields of the nodes hold the elements of the stack.
  - The **LINK** fields hold pointers to the neighboring element in the stack.
  - The **START** pointer of the linked list behaves as the **TOP** pointer variable of the stack.
  - The null pointer of the last node in the list signals the bottom of stack.

Linked Representation of Stacks II

- The condition **TOP == NULL** will indicate that the stack is empty.



Algorithm for PUSH

- Following algorithm pushes (inserts)ITEM into STACK.
- PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM): This procedure pushes an ITEM into a linked stack

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

Algorithm for POP

- Following algorithm pops (deletes) ITEM from the STACK.
- POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM): This procedure deletes the top element of a linked stack and assigns it to the variable ITEM

1. [Stack has an item to be removed?]
   IF TOP = NULL then Write: UNDERFLOW and Exit.
2. Set ITEM := INFO[TOP] [Copies the top element of stack into ITEM]
3. Set TEMP := TOP and TOP = LINK[TOP]
   [Remember the old value of the TOP pointer in TEMP and reset TOP to point to the next element in the stack]
4. [Return deleted node to the AVAIL list]
   Set LINK[TEMP] = AVAIL and AVAIL = TEMP.
5. Exit.

Applications of Stacks

- Stacks are widely used computer science.
- Their specific applications are:
  - Management of Function Calls
  - Evaluation of Expressions
  - Implementation of certain algorithms (e.g., Quick Sort)

Evaluation of Expressions I

- Operators Precedence:
  - In arithmetic expressions operators precedence is observed:
    - Highest: Exponentiation ($\uparrow$)
    - Next highest: Multiplication ($*$) and division ($/$)
    - Lowest: Addition ($+$) and subtraction ($-$)
- An Example:
  - Evaluate: $2 \uparrow 3 + 5 * 2 \uparrow 2 - 12/6$
  - Answer: **26**

Evaluation of Expressions II

- An Important Fact:
  - Parentheses' alter the precedence of operators.
- An Example:
  - $(A + B) * C \neq A + (B * C)$
  - $(2 + 3) * 7 = 35$ while $2 + (3 * 7) = 23$
- How computer evaluates the arithmetic expressions? – is the question we want to seek answer for.

Notations for Expressions I

- Infix Notations:
  - Expressions in which operator lies between the operands are referred to as infix notations.
  - A+B, C-D, P*F, $\cdots$ all are infix notations.
  - A+(B*C) and (A+B)*C are distinguished by parentheses or by applying the operators precedence discussed above.

Notations for Expressions II

- Prefix or Polish Notations:
  - Named in honour of Polish mathematician, Jan Lukasiewiez, refer to the expressions in which the operator symbol is placed before its two operands.
  - +AB, -CD, *PF, $\cdots$ all are examples of prefix or polish expressions.
  - Simple infix expressions can be converted to polish expressions as follows:
    - $(A+B)*C = [+AB]*C = *+ABC$
    - $A+(B*C) = A+[*BC] = +A*BC$
    - $(A+B)/(C-D) = [+AB]/[-CD]/+AB-CD$
  - An important property of these notations is that they are parentheses free.

Notations for Expressions III

- Postfix or Reverse Polish Notations
  - Refer to the expressions in which operator is placed after its two operands.
  - AB+, CD-, PF*... all are examples of postfix or reverse polish notations.
  - Like prefix notations, they are also parentheses' free.

How Computer Evaluates Expressions? I

- Expressions are represented in infix notations and use of parentheses is very common.
- Computer may apply the operators precedence and parentheses' rules and evaluate the expression.
- But, this process is not feasible in terms of computer timing (timing complexity)as computer takes a lot of time to resolve parentheses'.
- So, the computer first converts an infix expression into an equivalent postfix expression and then evaluates it.

How Computer Evaluates Expressions? II

- Following figure depicts the process:

How Computer Evaluates Expressions? III

- Clearly following two procedures (algorithms) are required:
  - Algorithm 1: Converting an infix expression to an equivalent postfix expression.
  - Algorithm 2: Evaluating the postfix expression.
- For each algorithm, Stack is the main tool to be utilized.

Algorithm 1 I

- Conversion of an infix expression to an equivalent postfix expression.

Algorithm 1 II

- POLISH(Q, P): Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the 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, push it onto STACK.
5. If an operator $\otimes$ is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than $\otimes$.
   (b) Add $\otimes$ 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 (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to P.]

   [End of If structure.]

   [End of Step 2 loop.]
7. Exit.

Algorithm 1-Example

- arithmetic infix expression $Q : A + (B * C - (D/E \uparrow F) * G) * H$

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	−	(+ (−	A B C *
(8)	(	(+ (− (	A B C *
(9)	D	(+ (− (	A B C * D
(10)	/	(+ (− (/	A B C * D
(11)	E	(+ (− (/	A B C * D E
(12)	↑	(+ (− (/ ↑	A B C * D E
(13)	F	(+ (− (/ ↑	A B C * D E F
(14)	)	(+ (−	A B C * D E F ↑ /
(15)	*	(+ (− *	A B C * D E F ↑ /
(16)	G	(+ (− *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * −
(18)	*	(+ *	A B C * D E F ↑ / G * −
(19)	H	(+ *	A B C * D E F ↑ / G * − H
(20)	)		A B C * D E F ↑ / G * − H * +

Algorithm 2

- Evaluating the postfix expression.
- This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B $\otimes$ A.
   (c) Place the result of (b) back on STACK.

   [End of If structure.]

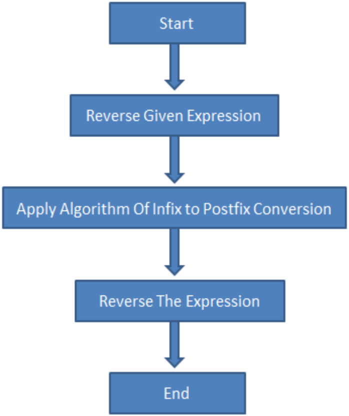
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

Algorithm 2-Example

- Arithmetic expression P written in postfix notation: P: 5, 6, 2, +, *, 12, 4, /, –

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)	)	

Infix to Prefix I



Infix to Prefix II

Algorithm of Infix to Prefix

1. Push “)” onto STACK, and add “(“ to end of the A
2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. If an operand is encountered add it to B
4. If a right parenthesis is encountered push it onto STACK
5. If an operator is encountered then:
    a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
    b. Add operator to STACK
6. If left parenthesis is encontered then
    a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
    b. Remove the left parenthesis
7. Exit

Example Infix to Prefix I

Expression: (A + B $\wedge$ C) * D + E $\wedge$ 5

1. **Reverse the infix expression:** 5 $\wedge$ E + D *) C $\wedge$ B + A (
2. **Make every '(' as ')' and every ')' as '(':** 5 $\wedge$ E + D * (C $\wedge$ B + A)

Example Infix to Prefix II

3. **Convert expression to postfix form:**

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Example Infix to Prefix III

4. **Reverse the expression:** + * + A ∧ B C D ∧ E 5

Result: + * + A ∧ B C D ∧ E 5

Evaluation of Prefix Expressions I

Algorithm for evaluating a prefix expression

1. Put a pointer P at the end of the end.
2. If character at P is an operand push it to Stack.
3. If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack.
4. Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.
5. The Result is stored at the top of the Stack, return it.
6. End.

Example- Evaluation of Prefix Expressions I

Expression: + 9 * 2 6

Character Scanned	Stack (Front to Back)	Explanation
6	6	6 is an operand, push to Stack
2	6 2	2 is an operand, push to Stack
*	12 (6 * 2)	* is an operator, pop 6 and 2, multiply them and push result to Stack
9	12 9	9 is an operand, push to Stack
+	21 (12+9)	+ is an operator, pop 12 and 9 add them and push result to Stack

Result: 21