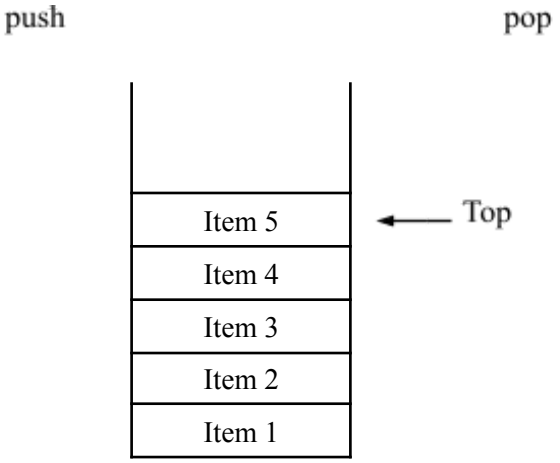
Unit - III

Stacks and Queues

Learning Material

Stacks

- Stack is a linear data structure.
- Stack is an ordered collection of homogeneous data elements, where insertion and deletion operations takes place at only one end called **TOP**.
- The insertion operation is termed as **PUSH** and deletion operation is termed as **POP** operation.
- The **PUSH** and **POP** operations are performed at **TOP** of the stack.
- An element in a stack is termed as **ITEM**.



Schematic diagram of a stack

- The maximum number of elements that stack can accommodate is termed as **SIZE** of the stack.
- Stack follows **LIFO** principle. i.e. **L**ast **I**n **F**irst **O**ut.

Representation of stack

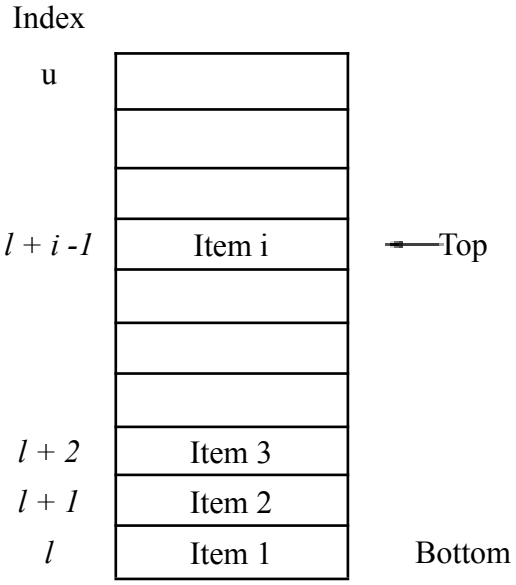
There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

1. Array representation of a stack

- First we have to allocate memory for array.

- Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



Array representation of stack

In the above figure item i denotes the i^{th} item in stack.

l and u denotes the index ranges.

Usually *l* value is **0** and *u* value is **size-1**.

From the above representation the following two statuses can be stated.

Empty Stack: top < *l* i.e. **top < 0**

Stack is full: top >=*u* i.e. **top >= size-1**

Stack overflow

Trying to PUSH an item into full stack is known as stack overflow.

Stack overflow condition is top >= size-1

Stack underflow

Trying to POP an item from empty stack is known as Stack underflow.

Stack underflow condition is top < 0 or top = = -1

Operations on Stack

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

Algorithm Stack_PUSH(item)

Input: *item* is new item to push into stack.

Output: pushing new item into stack at top whenever stack is not full.

1. if(top >= size-1)
    a) print "stack is full, not possible to perform push operation"
2. else
    a) top = top+1
    b) s[top] = item
3. end if

End Stack_PUSH

Algorithm Stack_POP( )

Input: Stack with some elements.

Output: item deleted at top most end.

1. if(top < 0)
    a) print "stack is empty not possible to pop"
2. else
    a) item = s[top]
    b) top = top - 1
    c) print(item)
3. end if

End Stack_POP

Algorithm Stack_Status( )

Input: Stack with some elements.

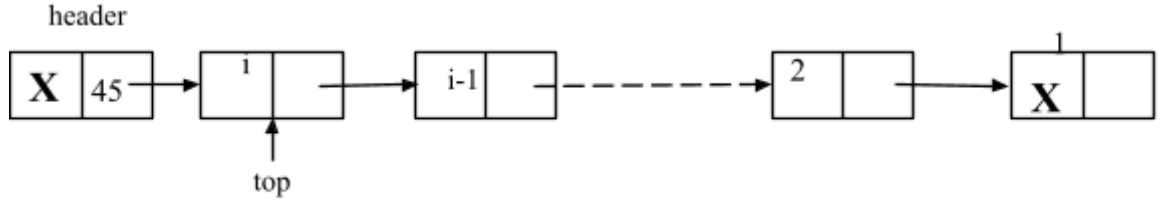
Output: Status of stack. i.e. Stack is empty or not, full or not, top most element in Stack.

1. if(top > = size-1)
    a) print "stack is full"
2. else if(top < 0)
    a) print "stack is empty"
3. else
    a) print "top most item in stack is" s[top]
4. end if

End Stack_Status

2. Linked List representation of a stack

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for the item and link field points to next node.



Linked List representation of stack

- Empty stack condition is
    top = = NULL (or) header link = = NULL
- Full condition is *not applicable* for Linked List representation of stack. Because here memory is dynamically allocated**.**
- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

Operations on Stack with linked list representation

PUSH : To insert element in to stack

POP : To delete element from stack

Status : To know present status of the stack

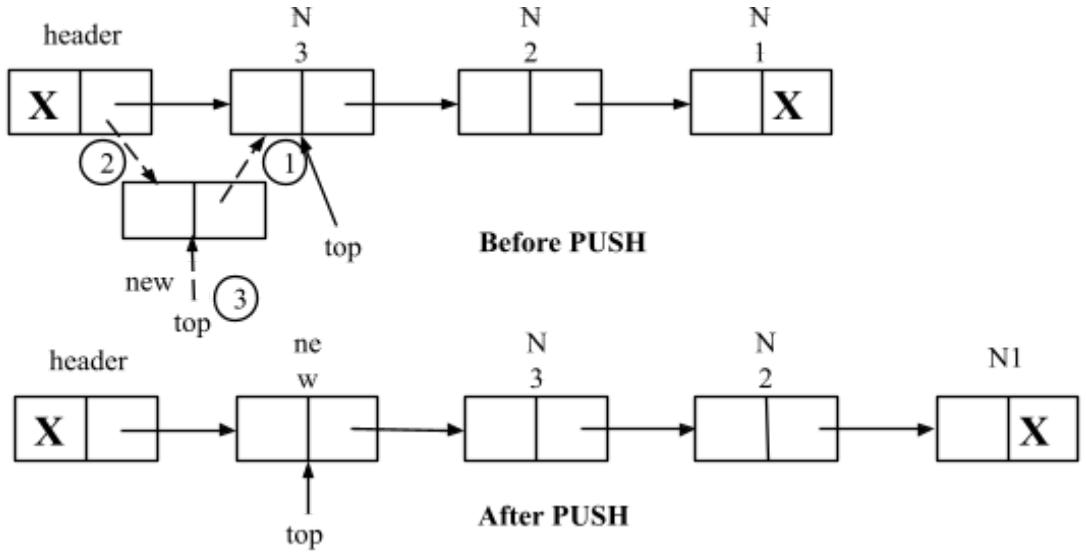
Algorithm Stack_PUSH_LL(item)

Input: *item* is new item to push into stack.

Output: pushing new item into stack at top.

1. new = getnewnode( )
2. if(new = = NULL)
    a) print "Required node is not available in memory"
3. else
    a) new link = header link
    b) header link = new
    c) top = new
    d) new data = item

End Stack_PUSH_LL



1. The link part of the new node is re placed with address of the previous top most node.
2. The link part of the header node is replaced with address of the new node.
3. Now the new node becomes top most node. So top is points to new node.

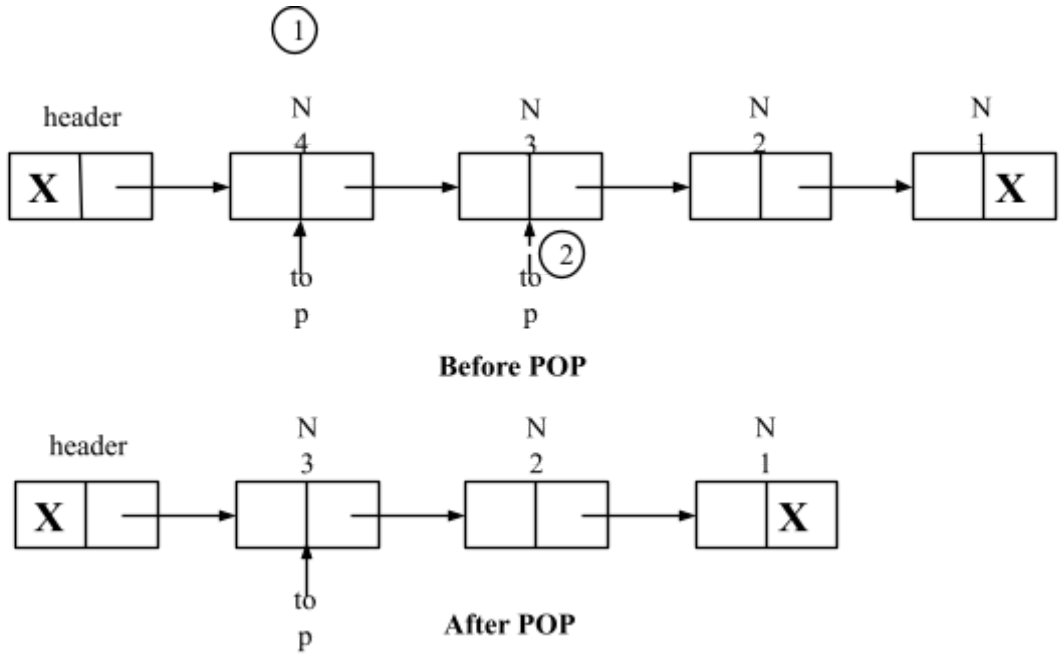
Algorithm Stack_POP_LL( )

Input: Stack with some elements.

Output: item deleted at top most end

1. if(header link = = NULL)
    a) print "Stack is empty, unable to perform POP operation"
2. else
    a) header link = top link
    b) item = top data
    c) top = header link

End Stack_POP_LL



1. Link part of the header node is replaced with the address of the second node in the list.
2. After deletion of top most node from list, the second node becomes the top most node in the list. So top points to the second node.

Algorithm Stack_Status_LL( )

Input: Stack with some elements.

Output: Status of stack. i.e. Stack is empty or not, top most element in Stack.

1. if(header link = = NULL || top = = NULL)
    a) print "Stack is empty"
2. else
    a) print "Element present at top of stack is" top data
3. end if

End Stack_Status_LL

Applications of stack

1. Infix to postfix conversion
2. Evaluation of postfix expression
3. Reversing list

1. Infix to postfix conversion

- An expression is a combination of operands and operators.

  Eg: c = a + b

- In the above expression a, b, c are operands and +, = are called as operators.
- We have 3 notations for the expressions.
  i. Infix notation
  ii. Prefix notation
  iii. Postfix notation

Infix notation: Here operator is present between two operands.

eg. a + b

The format for Infix notation as follows:

<operand> ***<operator>*** <operand>

Prefix notation: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows:

<operator> <operand> <operand>

Postfix notation: Here operator is present after two operands.

eg. a b +

The format for Prefix notation as follows:

<operand> <operand> ***<operator>***

- While conversion of infix expression to postfix expression, we must follow the precedence and associativity of the operators.

Operator	**Precedence**	**Associativity**
^ or $ or ↑ (exponential)	3	Right to Left
* / %	2	Left to Right
+ -	1	Left to Right

- In the above table *, / and % have same precedence. So then go for associativity rule, i.e. from Left to Right.
- Similarly + and - same precedence. So then go for associativity rule, i.e. from Left to Right.

Eg: 1(Conversion of infix expression to postfix expression without using STACK)

(A + B) * (C – D)

A B + * (C – D)

A B + * C D –

A B + C D - *

Eg: 2 (Conversion of infix expression to postfix expression without using STACK)

([(A - { B + C }) * D] $ E + F)

([(A - BC+) * D] $ E + F)

([ABC+- * D] $ E + F)

(ABC+-D* $ E + F)

(ABC+-D*E$ + F)

ABC=-D*E$F+

- To convert an infix expression to postfix expression, we can use one stack.
- Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

Ex: Convert the given infix expression to postfix expression using STACK

Input Character	**Operations on Stack**	**Operator Stack**	**Postfix Expression**
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till Empty		**AB*CD+-E+**

A * B- (C + D) + E

Ex: Convert the given infix expression to postfix expression using STACK

(A + B * (C - D)) / E

Input Character	**Operator Stack**	**Postfix Expression**
(	(	
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
(	(+*(	AB
C	(+*(	ABC
-	(+*(-	ABC
D	(+*(-	ABCD
)	(+*	ABCD-
)		ABCD-*+
/	/	ABCD-*+
E	/	ABCD-*+E
End		ABCD-*+E/

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

1. Perform the following steps while reading of infix expression is not over
    a) if symbol is left parenthesis then push symbol into stack.
    b) if symbol is operand then add symbol to postfix expression.
    c) if symbol is operator then check stack is empty or not.
        i) if stack is empty then push the operator onto stack.
        ii) if stack is not empty then check priority of the operators.
            (I) if priority of current operator > priority of operator present at top of stack then push operator into stack.
            (II) else if priority of operator present at top of stack >= priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step i)
    d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.
2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

End conversion of infix to postfix

2. Evaluation of postfix expression

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression

1. Repeat the following steps while reading the postfix expression.
    a) if the read symbol is operand, then push the value onto stack.
    b) if the read symbol is operator then pop the top most two items of the stack and apply the operator on them, and then push back the result to the stack.
2. Finally stack has only one item, after completion of reading the postfix expression. That item is the result of expression.

End PostfixExpressionEvaluation

Ex: Evaluate the given postfix epression using stack: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

3. Reversing List of elements

- A list of numbers can be reversed by reading each number from an array starting from 1st index and pushing into stack.
- Once all the numbers have been push into stack, the numbers can be poped one by one from stack and store into array from the 1st index.

Algorithm Reverse_List_Stack(a <array>, n <intetger>)

Input : Array a with n elements

Output: Reversed List of elements

1. i=1 , top =0

2. while (i <=n)

    a) top = top + 1

    b) s[top] = a[i]

    c) i = i + 1

3. end while loop

4. i = 1

5. while(i <= n)

    a) a[i] = s[top]

    b) top = top – 1

    c) i = i + 1

6. end while loop

End Reverse_List_Stack