Unit 1: Stack

Structure

1.0 Introduction

This unit is introducing the concept of another linear data structure i.e. Stack. It provides the definition of stack, its representation in memory, implementation procedure and different common and important operations those can perform on the elements of stack. This unit also includes the method for evaluation of arithmetic expressions using stack. In the end it highlights about the multiple stack concept and the different applications of the stack.

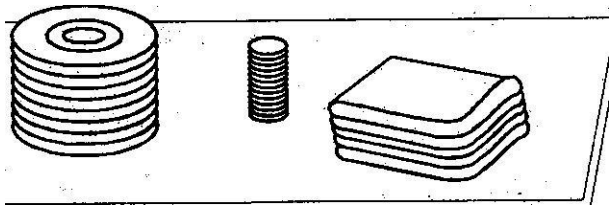
1.1 Objectives

After going through this unit, you should be able to:

- Understand for the concept of stack
- Implementation of the stack using array.
- Implementation for the various operations on stack (Push, Pop).
- Understanding for the method of evaluation of arithmetic expressions using stack (infix, prefix and postfix representation).
- Understanding the concept of multiple stacks and its application.

1.2 Stack

A ***stack*** is a linear data structure where all the elements in the stack can insert and delete from one side only rather than at the middle or from both the side. Thus, from the stack the elements may be added or removed only from one side. The following figure shows the three everyday common use examples of such structure i.e. a stack of dishes, a stack of coins and a stack of folded towels.



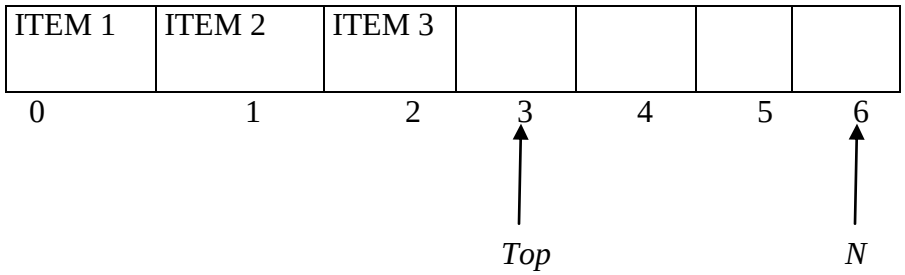
As the items in this type of data structure can be added or removed from the top, it means the last item to be added to the stack is the first item to be removed. Therefore stacks are also called Last –in-First-out lists. Thus, the stack follows the principle of Last-in-First-out (LIFO) or First-in-Last-Out (FILO) type working system. Thus, the element which is inserted first in the stack will remove or delete last from the stack. The other names used for the stacks are *"piles"* and "*push-down list"*. Therefore we can understand that a stack is a list of elements in which an element may be inserted or deleted only at the end, called the ***Top*** of the stack. A stack of elements of any particular type is a finite sequence of elements of that together with the following operations:

- **Initialize** the stack to be empty.
- Determine whether stack is **empty** or not.
- Determine if stack is **full** or not.
- If stack is not full, then add or insert a new node at the *Top*. This operation is called **push.**
- If the stack is not empty, then retrieve the node at its *Top.*
- If the stack is not empty, then delete the node at its *Top*. This is called **pop.**

The above definition of stack produces the concept of stack as an abstract data type.

1.3 Array Representation of Stack

Generally a stack may be represented with a linear array or one way list. In the stack, a pointer variable *TOP* contains the location of the top element of the stack. The variable *N* provides the maximum number of elements that can be stored or held in the stack or size of the linear array. Sometimes it is also called as the size of the Stack. The condition where *TOP* is *NULL*, indicate that the stack is empty and when the *TOP* is *N*, the stack is full. The following figure shows the array representation of a stack. The *TOP* is pointing to 3 which says that stack has three items and as the $N = 6$, there is still space for accommodating four items.



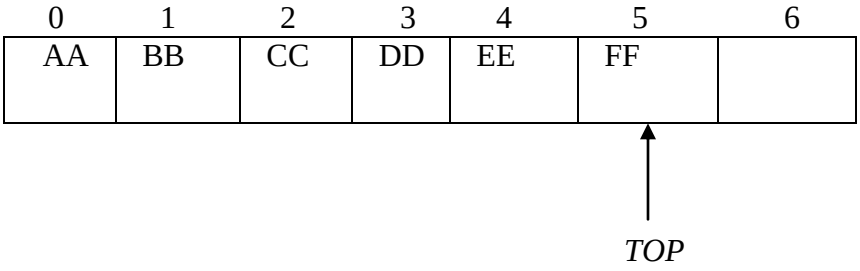
Array Representation of a Stack

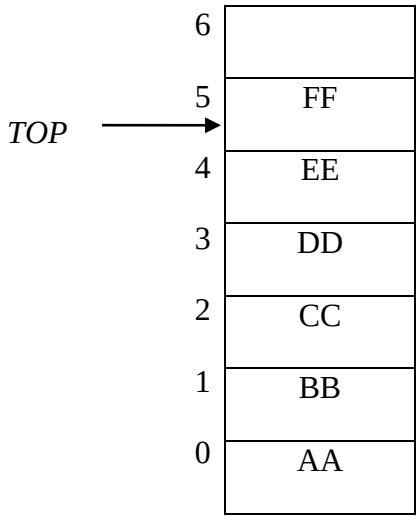
Example:

Suppose the following elements are inserted in order in an empty stack of size 7:

AA, BB, CC, DD, EE, FF

This stack can represent in following ways using a linear array:





1.4 Operation on Stack

The operation to add an element into the stack (*push*) and the operation to remove an element from the stack (*pop*) can be implemented using the ***PUSH*** and ***POP*** functions. When we add a new element into the stack, first we check that whether there is a free space in the stack for the new element or not. If there is no free space available for the new element then we have the condition of overflow. In the same way for the function ***POP*** we must first check the condition that whether there is an element in the stack to be deleted or not. If there is no element in the stack to delete then we have the condition of *underflow*. These functions are defined as follows:

Function *PUSH (STACK, TOP, N, ITEM)*

[This function adds or pushes an ITEM in the stack]

```
/* check the condition of overflow*/
If (TOP == N)
Printf ("stack is overflow"); exit;
Else {
    TOP = TOP+1; /* Increase TOP by 1 */
    STACK [TOP] = ITEM; } /* Insert ITEM in new TOP position */
    RETURN
```

Function *POP (STACK, TOP, ITEM)*

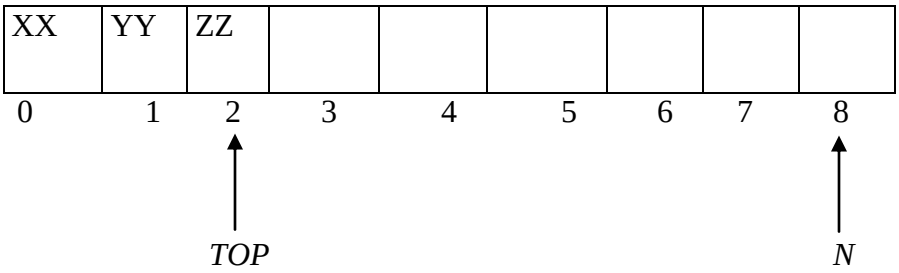
[This function deletes or pops the *TOP* element of *STACK* and assigns it to the variable *ITEM*]

```
/* check the condition of underflow*/
If (TOP == -1)
Printf ("stack is underflow"); exit;
Else {
    ITEM = STACK [TOP];  /* assign TOP element to ITEM */
    TOP = TOP – 1; }  /* Decrease TOP by 1]
    RETURN
```

We can see from the above defined functions that the value of *TOP* is changed before adding the element in the stack in function ***PUSH*** but the value of *TOP* is changed after removing the element from stack in function ***POP***.

Example:

Consider the following stack with size of 9.

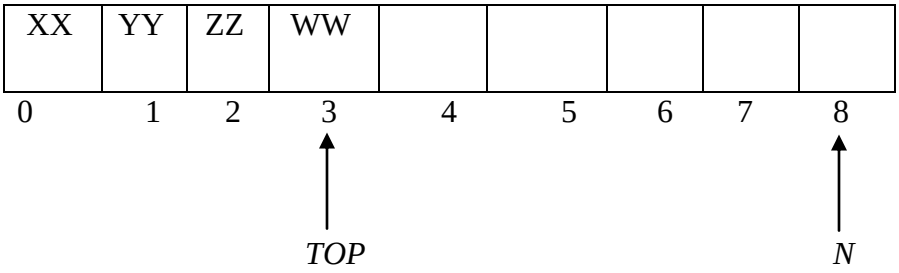


If we perform the operation ***PUSH (STACK, WW);*** then the status of the stack can determine as:

Since *TOP = 2*, so *TOP = 2 + 1 = 3*.

And *STACK [TOP] = STACK [3] = WW*.

Therefore the item *WW* is now top element of the stack. This can represent as:



Now, on the same stack we perform the operations ***POP (STACK, ITEM)***; ***POP (STACK, ITEM)***; then the status of the stack can determine as:

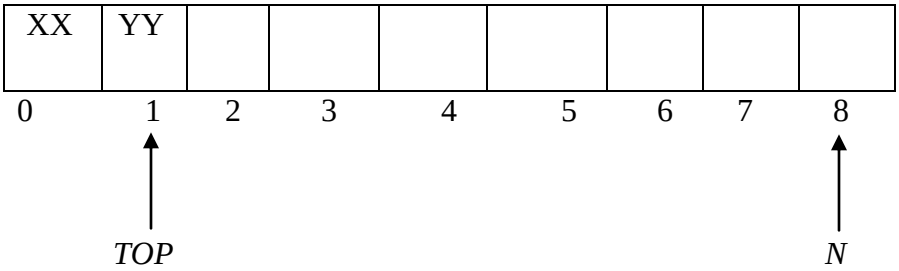
Right now the *TOP* = 3 and the operation *POP* is performed two times. So, after the execution of first *POP* operation:

ITEM = WW and *TOP = 3-1 = 2*

Now after the execution of second *POP* operation:

ITEM = ZZ and *TOP = 2-1 = 1*

Therefore *STACK [TOP] = STACK [1] = YY* is now the top element in the stack.



1.5 Evaluation of Arithmetic expression (infix, postfix and prefix notions) using stack

This is well known that the computer system can understand and work only on binary paradigm. In which an arithmetic operation can take place between two operands only like *A + B, C * D, D / A*. generally an arithmetic expression may consist of more than one operator and two operands, for example *(A + B) * (D / (J + D)).* Such form of the arithmetic expression is commonly known as the *infix* expression. Normally the evaluation of the any arithmetic expression does not take place in its *infix* form. Here we are introducing the method for evaluating the *infix* expression form computation point of view. The ***stack*** is found to be more efficient to evaluate an infix arithmetical expression by first converting to a *prefix* or *postfix* expression and then evaluating these converted expressions. This approach will eliminate the repeated scanning of an infix expression in order to obtain its value. Therefore there are three basic notations are referred for the representation of any complex arithmetic expression. These forms are as follows:

- If the operator symbols are placed before its operands, then the expression is in ***prefix form***.
- If the operator symbols are placed after its operands, then the expression is in ***postfix form***.

- If the operator symbols are placed between the operands then the expression is in ***infix form***.

Hence, a normal arithmetic expression is normally called as infix expression i.e. A+B. A Polish mathematician found a way to represent the same expression called **polish notation** or prefix expression by keeping operators as prefix i.e. +AB. We use the reverse way of the above expression for our evaluation. The representation is called **Reverse Polish Notation** (RPN) or postfix expression i.e. AB+.

Let ***Q*** be an infix arithmetic expression involving constants and operations. This expression will be evaluated with the common rule of arithmetic evaluation with the following level of operator precedence:

Highest:	Exponentiation (↑)
Next highest:	Multiplication (*) and division (/)
Lowest:	Addition (+) and Subtraction (-)

Now let we evaluate the following parenthesis free arithmetic expression:

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

First we evaluate the exponentiations to obtain:

$$8 + 5 * 4 - 12 / 6$$

Then we evaluate the multiplication and division to obtain $8 + 20 - 2$. Last, we evaluate the addition and subtraction to obtain the final result i.e. 26. We can observe that in this evaluation the whole expression is traversed three times, each time corresponding to a level of precedence of the operations.

Second important issue about the *infix notation* is that these expressions use parentheses for clarity and to make the evaluation convenient like $(A + (B - C)) * D) / E$. On the other hand the polish notation expression i.e. *prefix* and reverse polish notation expression i.e. *postfix* do not include any parentheses for clarity.

Now we consider for example the step by step translation of following infix expression into polish notation using square brackets i.e. [ ] to indicate a partial translation:

$$(A + B) * C = [+AB] * C = * + ABC$$

$$A + (B * C) = A + [*BC] = + A * BC$$

$$(A + B) / (C - D) = [+AB] / [-CD] = / + AB - CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. There is no need of parentheses when writing expressions in Polish notation. The computer usually evaluates an arithmetic expression written in infix notation in following two steps.

- Coverts the expression in Reverse Polish notation form (*post fix notation).*
- Evaluate the *post fix* expression using stack.

Algorithm for transforming Infix Expression into Postfix Expression

Let ***Q*** be an arithmetic expression written in infix notation. The following algorithm transforms the given infix expression ***Q*** into its equivalent postfix expression ***P***. This algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression ***P*** will be constructed from left to right using the operands from ***Q*** and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of ***Q***. The algorithm is completed when STACK is empty:

Algorithm: Conv_POLISH (*Q, P)*

[Suppose ***Q*** is an arithmetic expression written in *infix* notation. The 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 step3 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 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 operator.*
    b. *Add operator to STACK.*
    [end of If]
6. *If a right parenthesis is encountered, then:*
    a. *Repeatedly pop from STACK and add to* ***P*** *each operator (on 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]
[End of step 2 loop]

7. *Exit.*

Example:

Consider the following arithmetic *infix* expression:

***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

The elements of ***Q*** have now been labeled from left to right for easy reference. Following table shows the status of STACK and of the Postfix string ***P*** as each element of ***Q*** is scanned to follow the following steps of algorithm.

1. Each element is simply added to ***P*** and does not change STACK from A to C; operators +. (, * are pushed to stack and A, B, C are added to P.
2. The subtraction operator (- in row 7 sends * from STACK to ***P*** before it (-) is pushed onto the STACK.
3. The right parenthesis in row 14 sends ↑ and then / from STACK to ***P*** and then removes the left parenthesis from the STACK.
4. The right parenthesis in row 20 sends * then + from STACK to ***P*** and then removes the left parenthesis from the top of STACK.
   After step 20 is executed, the stack is empty.

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)	)	(+(-	ABC*DEF↑ /
(15)	*	(+(-*	ABC*DEF↑ /
(16)	G	(+(-*	ABC*DEF↑ /G
(17)	)	(+	ABC*DEF↑ /G*-
(18)	*	(+*	ABC*DEF↑ G*-
(19)	H	(+*	ABC*DEF↑ G*-H
(20)	)		ABC*DEF↑ G*-H*+

Evaluation of Post Fix Expression

As we know that in the *infix* expression it is difficult for the computer to keep track of precedence of operators. On the other hand, a *postfix* expression itself determines the precedence of operators due to the placement of the operator. Hence it is easier for the computer to perform the evaluation of a postfix expression. The evaluation rule for the *postfix* expression is stated as:

1. *Read the expression from left to right.*
2. *If it is an operand then push the element into the stack.*

3. *If the element is an operator except NOT operator, pop the two operands from the stack and evaluate them with the read operator and push back the result of the evaluation into the stack.*
4. *If it is the NOT operator then pop one operand from the stack and then evaluate it and push back the result of the evaluation into the stack.*
5. *Repeat it till the end of stack.*

Now we define the algorithm for evaluation of postfix expression using STACK. Let ***P*** be a arithmetic expression written in postfix notation. The following algorithm uses the STACK to hold the operands and evaluate expression ***P.***

[This algorithm finds VALUE of an arithmetic expression ***P*** written in *postfix* notation.]

1. *Add a right parenthesis ")" at the end of* ***P.***
2. *Scan* ***P*** *from left to right and repeat step 3 and 4 for each element until ")" is not encountered.*
3. *If an operand is encountered , put it on STACK*
4. *If an operator 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 and A for the encountered operator.*
    c. *Place the result of evaluation on step b back on STACK*

    [End of if]

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

Example

Evaluate the following arithmetic expression ***Q*** written in *infix* notation:

Q: *10 * (8 + 4) – 6 / 3*

The equivalent *postfix* notation for the given *infix* notation is:

P: *10, 8, 4, +, *, 6, 3, /,* - [Here Commas are used to separate the elements of ***P***]

Now we apply the algorithm to evaluate the *postfix notation:*

First we add ')' at the end of right side in expression ***P***.

P: 10, 8, 4, +, *, 6, 3, /, -,)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

The evaluation procedure with contents of STACK can consider from the following table:

Symbol Scanned	Stack
(1) 10	10
(2) 8	10, 8
(3) 4	10, 8, 4
(4) +	10, 12
(5) *	120
(6) 6	120, 6
(7) 3	120, 6, 3
(8) /	120,2
(9) –	118
(10))	

The final number in STACK is 118, which will assign to the VALUE when ')' encounters. Thus the evaluation of *postfix notation* ***P*** is 118.

Algorithm to convert *infix* into *prefix* expression form

Suppose ***Q*** is an arithmetic expression written in infix form. The following steps find its equivalent *prefix* expression ***P.***

1. *Push ')' onto STACK and add '(' to the begin of* ***Q.***
2. *Scan* ***Q*** *from right to left and repeat steps 3 to 6 for each element of* ***P*** *until the STACK is empty.*
3. *If an operand is encountered add it to* ***P.***
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* ***P*** *each operator (on the top of STACK) which has same or higher precedence than the operator.*
    b. *Add operator to STACK.*
6. *If a left parenthesis is encountered then*
    a. *Repeatedly pop from the STACK and add to* ***P*** *until a right parenthesis is encountered.*
    b. *Remove the Right parenthesis*
7. *Exit*

Example:

Convert the following given *infix expression* in its equivalent *prefix notation form*.

Q: ***(A + B * C – D + E / (F + G))***

The procedure for converting the given *infix expression* into its equivalent *prefix notation* form by applying the algorithm can represent in following table:

Symbol Scanned	**Stack**	***Prefix* expression**
)		
)	))	
G	))	G
+	)) +	G
F	))+	FG
(	Empty	+FG
/	)/	+FG
E	)/	E+FG
+	)+	/ E+FG
D	)+	D/E+FG
-	)+-	D/E+FG
C	)+-	CD/E+FG
*	)+-*	CD/E+FG
B	)+-*	BCD/E+FG
+	)+-+	*BCD/E+FG
A	)+-+	A*BCD/E+FG
(	Empty	+-+A*BCD/E+FG

Hence the resultant equivalent *prefix notation* for the given *infix notation* is:

P: *+, -, *, +, A, B, C, D, / E + F G*

Algorithm for evaluation of Prefix Expression

[This algorithm performs the evaluation for the *infix notation* expression. Here the expression will read from right to left]

1. *Read the next element.*
2. *If element is operand then*

a. *Push the element in the stack*

3. *If element is operator then*
    a. *Pop two operands from the stack*
    b. *Evaluate the expression formed by two operands and the operator*
    c. *Push the results of the expression in the stack*
4. *If no more elements then*
    a. *Pop the result*

    Else

    Go to step 1.

Example:

Evaluate the following *prefix notation* expression.

P: *+ 2 * 3 + 4 5*

Now we start to read it from right to left: *+2 * 3 + 45*

Symbol Scanned	**Stack**
5	5 (push)
4	4, 5 (push)
+	9 (pop, push)
3	3, 9 (push)
*	27 (pop, push)
2	2, 27 (push)
+	29 (pop, push)

The final number in STACK is 29. Thus the evaluation of pre*fix notation* ***P*** is 29.

1.6 Application of Stacks

The stack has various applications in computer science. It includes the application of the level of computer organization and programming level. Generally being a linear data structure the following applications of stacks are highlighted:

- The stack is used for reversal of a given list. We can accomplish this task by pushing each element onto the stack as it is read. When the line is finished, elements are then popped off the stack, so they come off in reverse order.
- The important application of the stack in computer organization is for the evaluation of arithmetic expressions. It accomplishes by converting first the given expression in reverse polish notation and then evaluates the expression.
- It is also used for the zero address instruction implementation in computer organization.
- Stacks are used for the address holding in function calling procedure of programming.
- The stacks are used to implement recursive procedures. Recursion is useful in developing algorithm for specific problems. Suppose a function contains either a call statement to itself or a call statement to a second function that may eventually result in a call statement back to the original function. Then such a function is called recursive function. The stacks are used generally for the implementation of such type of recursive functions.

1.7 Summary

In this unit we presented another important linear data structure i.e. Stack. A stack is a linear data structure where all the elements in the stack can insert and delete from one side only rather than at the middle or from both the side. We have explored the implementation of stack in static manner using array. The two basic operations of PUSH and POP are discussed with example for the stack. The contents can be summarized as follows:

- A stack is a linear structure implemented in LIFO (Last In First Out) manner where insertions and deletions take place at the same end.
- An insertion is a stack is called pushing and deletion from a stack is called popping.
- When a stack implemented as an array, is full and no new element can be accommodated, it is called OVERFLOW.
- When a stack is empty and an attempt is made to delete an element from the stack, it is called UNDERFLOW.
- The main application of stack can be implementation of Polish notation which refers to a notation in which operator symbol is placed either before its operands (prefix notation) or

after its operands (postfix notation). The usual form, in which operator is placed in between the operands, is called infix notation.

- The other application of stack can be reversing a list and providing recursion in various programs.
- It is also used for the zero address instruction implementation in computer organization.
- Stacks are used for the address holding in function calling procedure of programming.

Bibliography

- Horowitz, E., S. Sahni: "Fundamental of computer Algorithms", Computer Science Press, 1978
- J. P. Tremblay, P. G. Sorenson "An Introduction to Data Structures with Applications", Tata McGraw-Hill, 1984
- M. Allen Weiss: "Data structures and Problem solving using C++", Pearson Addison Wesley, 2003
- Ulrich Klehmet: "Introduction to Data Structures and Algorithms", URL: http://www7 . Informatik.uni-erlangen.de/~klehmet/teaching/SoSem/dsa
- Markus Blaner: "Introduction to Algorithms and Data Structures", Saarland University, 2011
- M. E. D'imperio, "Data structures and their representation in storage", Annual Review in Automatic programming, Vol. 5 pp. 1-75, Pergammon Press, Oxford, 1969.
- B. Flaming, "Practical Data Strcture in C++", Jhon Wiley & Sons, New York, 1983
- D. E. Knuth, "The art of Computer programming", Vol. 2: Seminumerical Algorithms, 3rd edition, Addison-Wesley, 1997.

Self Evaluation

Multiple Choice Questions:

1. ..... form of access is used to add and remove nodes from a stack
   (a) LIFO (b) FIFO
   (c) Both (a) and (b) (d) None of these
2. A data structure in which elements are added and removed only at one end is know as:
   (a) Queue (b) Stack
   (c) Array (d) None of these
3. Underflow is a condition where you:
   (a) Insert a new node when there is no free space for it
   (b) Delete a non-existent node in the list
   (c) Delete a node from the empty list
   (d) None of the above
4. Stack is:
   (a) Static data structure
   (b) Dynamic data structure
   (c) A built in data structure
   (d) None of these
5. Which operation in the stack is used for getting value of most recent node and delete the node:
   (a) PUSH
   (b) POP
   (c) Empty
   (d) None of these
6. If A, B, C are inserted into a stack in the lexicographic order, the order of removal will be:
   (a) A, B, C
   (b) C, B, A
   (c) B, C, A
   (d) None of these.

Fill in the blank:

1. A stack may be represented by a …… linked list. (linear / non-linear)
2. Push operation in stack may result in…… (overflow / underflow)
3. If TOP points to the top of stack, then TOP is……. (increased / decreased)

State whether True or False

1. Push operation in stack is performed at the rear end.
2. PUSH operation in stack may result in underflow
3. For a stack implemented with linear array arbitrary amount of memory can be allocated.

Descriptive Questions

1. Consider the following stack, where ***STACK*** is allocated ***N = 6*** memory cells.

   STACK: ***AA, DD, EE, FF, GG……...***

   Describe the stack as the following operations take place and also consider the overflow condition.

   a. PUSH (***STACK, KK***)
   b. POP (***STACK, ITEM***)
   c. PUSH (***STACK, LL***)
   d. PUSH (***STACK, SS***)
   e. POP (***STACK, ITEM***)
   f. PUSH (***STACK, TT***)
2. Write an algorithm which upon user's choice, either pushes or Pops an element from the stack implemented as an array (the element should not shifted after the push or pop).
3. Write a program to convert an infix arithmetic expression into a prefix arithmetic expression. The algorithm for your program should use the following expression:

   *Q: (A – B) * (C / D) + E*

   Show in tabular form the changing status of stack.

4. Convert the expression *(A + B) / (C – D)* into postfix expression and then evaluate it for A = 10, B = 20, C = 15, D = 5. Display the stack status after each operation.

5. Write a program to read a string (one line of characters) and push any vowels in the string to a stack. Then pop your stack repeatedly and count the number of vowels in the string.