SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III -Data Structures – SCSA1203

UNIT 3 STACKS **9 Hrs.**

Basic Stack Operations - Representation of a Stack using Arrays - Algorithm for Stack Operations - Stack Applications: Reversing list - Factorial Calculation - Infix to postfix Transformation - Evaluating Arithmetic Expressions

3. STACK

3.1 INTRODUCTION

Stack is a linear data structure which follows a particular order in which the operations are performed. In stack, insertion and deletion of elements happen only at one end, i.e., the most recently inserted element is the one deleted first from the set.

This could be explained using a simple analogy, a pile of plates, where one plate is placed on the top of another. Now, when a plate is to be removed, the topmost one is removed. Hence insertion/removal of a plate is done at the topmost position. Few real world examples are given as:

- Stack of plates in a buffet table. The plate inserted at last will be the first one to be removed out of stack.

- Stack of Compact Discs

- Stack of Moulded chairs

- Bangles on Women's Hand

- Books piled on top of each other

These above examples state, stack as a linear list where all insertion and deletion are done only at one end of the list called Top. i.e., Stack implements **Last-in, First-out (LIFO)**policy. Stack can be implemented as an Array or Linked list.

3.2 STACK

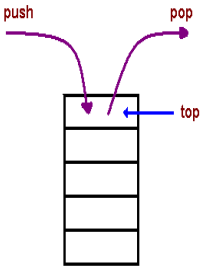


Fig. 3.2.1 Representation of stack as Array

Where does the stack concept implemented in computers? Though there are various applications, simple answer is in function calls. Consider the below example:

main ():	def funA():	def funB():	def funC():
----	----	----	----
funA()	funB()	funC()	----
----	----	----	----

In order to keep track of returning point of each active function, a special stack named System stack is used.

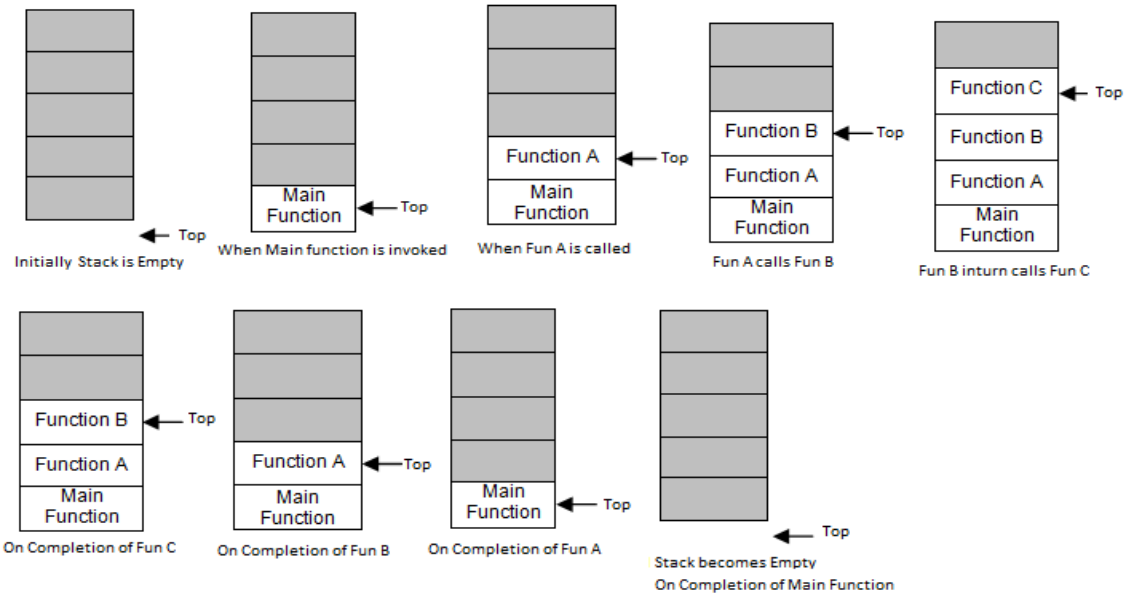


Fig. 3.2.2 Function calls with Stack

when A calls function B, A is pushed onto system stack. Similarly when B calls C,B is pushed onto stack and so on. Once the execution of function C is complete, the control will remove C from the stack. Thus system stack ensures proper execution order of functions.

The below diagram (Figure 3.3) depicts expansion and shrinking of a stack, initially stack is empty

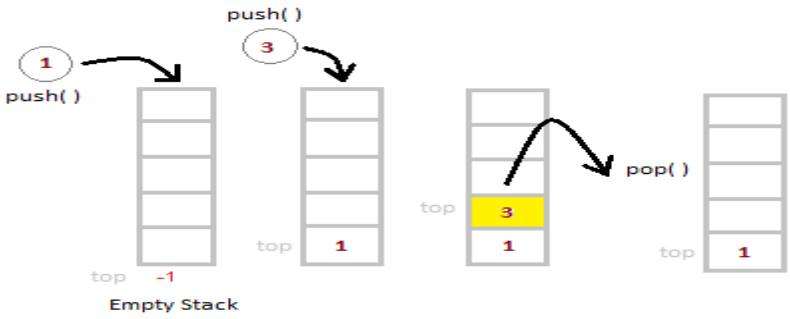


Fig. 3.2.3 Operations on Stack

In Stack, all operations take place at the "top" .Push(x) operation adds an item x at the top of stack and the Pop() operation removes an item from the top of stack.

3.2.1 Stack as an ADT (Abstract Data Type):

Stack can be represented using an array. A one dimensional array is used to hold the elements of the stack and a variable "top" is used for representing the index of the top most element. Formally defined as:

```
type def struct stack
{
int data[size];          // size is constant, represents maximum
int top;                 // number of elements that can be stored.
} S;
```

3.2.2 Basic Stack Operations

- **isEmpty:** Returns true if stack is empty, else false.
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

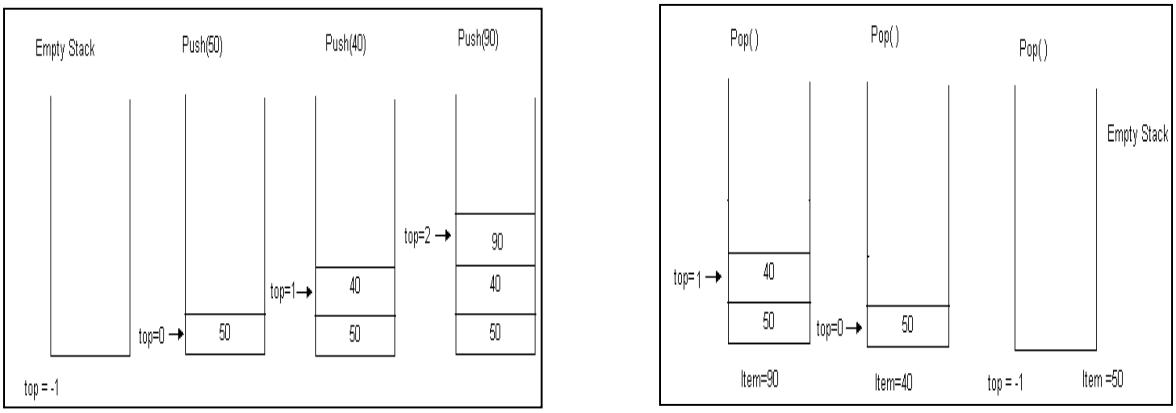


Fig.3.2.4.1 Push and Pop Operation of Stack

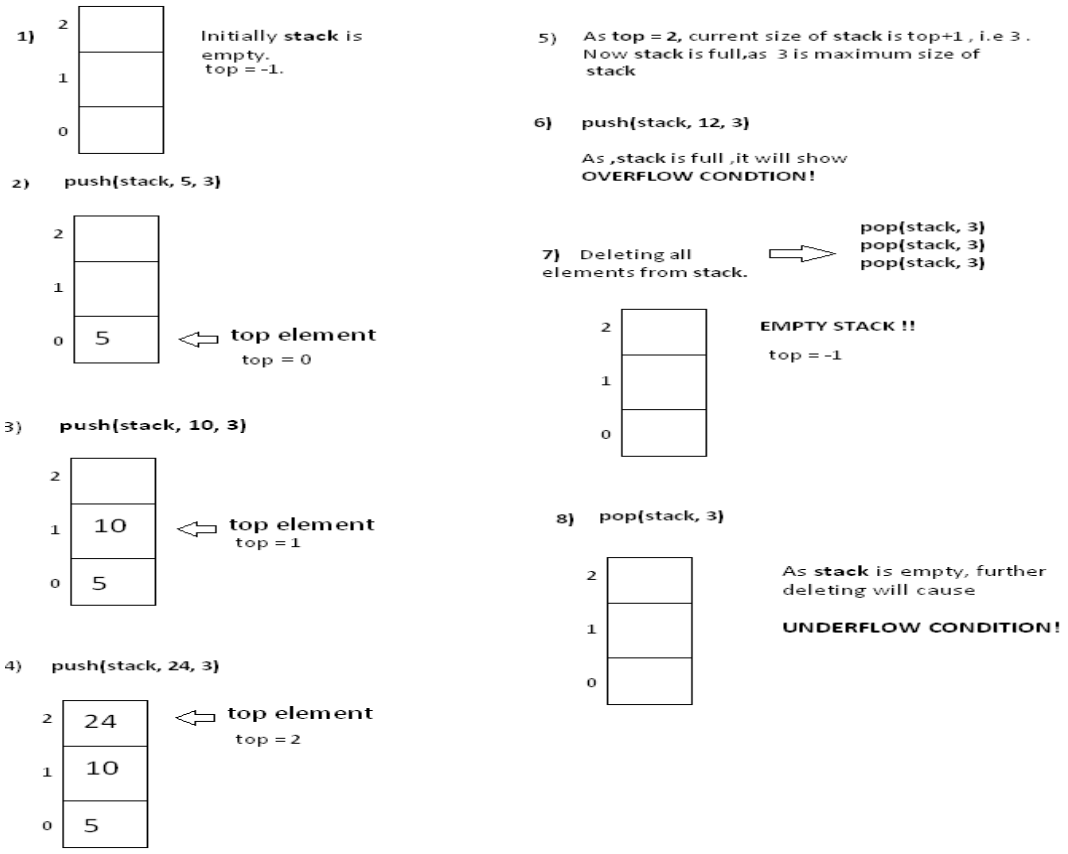


Fig.3.2.4.2 Push and Pop Operation of Stack

The fundamental operations in stack are given as follows:

STACK_EMPTY(S)

```
 if S.top == 0
    return TRUE
 else
    return FALSE

PUSH ( S.x )
   S.top = S.top+1
   S[S.top] = x

POP(S)
  if STACK_EMPTY(S)
      error "underflow"
  else
      x = S[S.top]
      S.top = S.top-1
      return x
```

When S.top = 0, the stack contains no elements and is empty. In that case an attempt to pop an element from empty stack is named stack **underflow**, which is normally reported as error. If S.top exceeds size, then stack **overflows**. In pseudo code implementation stack overflow is not represented. Each of the above three stack operations takes **O(1)** time.

3.2.3 ARRAY REPRESENTATION OF STACK

Stack is represented as a linear array. In an array-based implementation the following fields are maintained: an array S of a default Size (≥ 1), the variable *top* that refers to the top element in the stack and the size that refers to the array size. The variable *top* changes from -1 to Size-1. Stack is called empty when top = -1, and the stack is full when top = Size-1.Consider an example stack with size=10.

10	20	30	40						
0	1	2	Top=3	4	5	6	7	8	9

Fig.3.2.3.1 Array Representation of Stack

The variable Top represents the topmost element of the stack. In the above example six more elements could be stored.

OPERATIONS ON STACK

The two basic operations of stack are push and pop. Let's first discuss about array representation of these operations.

Push Operation

The push operation is used to insert an element into the stack. The element is added always at the topmost position of stack. Since the size of array is fixed at the time of declaration, before inserting the value, check if top=Size-1, if so stack is full and no more insertion is possible. In case of an attempt to insert a value in full stack, an OVERFLOW message gets displayed.

Consider the below example: Size=10

A	B	C	D	E					
0	1	2	3	Top=4	5	6	7	8	9

Fig.3.2.3.2 Array Representation of Stack, Push(E)

To insert a new element F, first check if Top==Size-1. If the condition fails, then increment the Top and store the value. Thus the updated stack is:

A	B	C	D	E	F				
0	1	2	3	4	Top=5	6	7	8	9

Fig.3.2.3.3 Array Representation of Stack, Push(F)

Algorithm:

Push(X)

```
if Top == Size-1
    Write "Stack Overflow"
```

```
        return
    else
        Top = Top + 1        // St represents an Array with maximum limit as Size
        St[Top] = X          // X element to be inserted
End
```

Pop Operation

The pop operation is used to remove the topmost element from the stack. In this case, first check the presence of element, if top == -1 (indicates no array elements), then it indicates empty stack and thereby deletion not possible. In case of an attempt to delete a value in an empty stack, an UNDERFLOW message gets displayed.

Consider the below example: Size=10

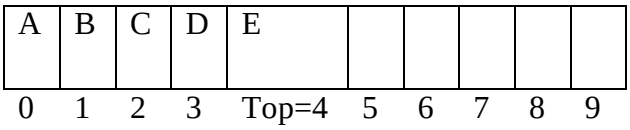


Fig.3.2.3.4 Array Representation of Stack, Pop()

To delete the topmost element, first check if Top== -1. If the condition fails, then decrement the Top.Thus the updated stack is:

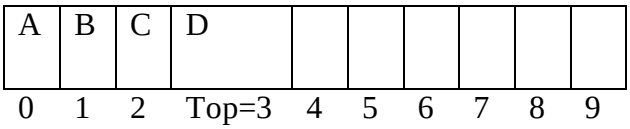


Fig.3.2.3.5 Array Representation of Stack, Pop()

Algorithm:

```
Pop()
  if Top == -1
     Write "Stack Underflow"
     return
  else
    X = St[Top]          // St represents an Array with maximum limit as Size
    Top = Top-1          // X represents element removed
     return X
```

End

Peek Operation

The peek operation returns the topmost element of the stack. Here if stack is not empty, the top element is displayed.

Consider the below example: Size=10

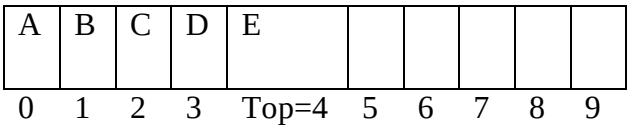


Fig.3.2.3.6 Array Representation of Stack, Peek()

Here, the peek operation will return E, as it's the topmost element.

Algorithm:

```
Peek()
    if Top == -1
        Write "Stack Underflow"
    else
        return St[Top]          // St represents an Array with maximum limit as Size
End
```

Implementation of stack operations using arrays in Python

```
class Stack:
    # Constructor
    def _init__(self):
      self.stack = list()
      self.maxSize = 5
      self.top = 0
```

```
    # Add element to the Stack
    def push(self,data):
        if self.top>=self.maxSize:
            return ("Stack Full!")
        self.stack.append(data)
        self.top += 1
        return "element inserted"

    # Remove element from the stack
    def pop(self):
        if self.top<=0:
            return ("Stack Empty!")
        item = self.stack.pop()
        self.top -= 1
        return item

    # Size of the stack
    def size(self):
        return self.top

s = Stack()
print("Push : ",s.push(1))
print("Push : ",s.push(2))
print("Push : ",s.push(3))
print("Push : ",s.push(4))
print("Size of Array :",s.size())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
print("Element Popped :",s.pop())
```

```
Push :  element inserted
Push :  element inserted
Push :  element inserted
Push :  element inserted
Size of Array : 4
Element Popped : 4
Element Popped : 3
Element Popped : 2
Element Popped : 1
Element Popped : Stack Empty!
>>>
```

Pros:-

- Easier to use.Array elements could be accessed randomly using the array index.
- Less memory allocation, no need to track the next node.
- Data elements are stored in contiguous locations in memory.

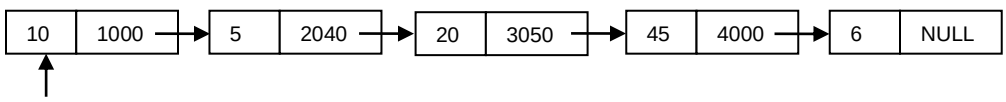
Cons:-

- Fixed size – cannot increase or decrease the total number of elements.

3.2.4 LINKED REPRESENTATION OF STACK

Stack using arrays was discussed in the previous section, the drawback of the above concept is that the array must be pre-declared i.e., size of array is fixed. If array size cannot be determined in advance, i.e., in case of dynamic storage,Linked List representation could be implemented.

In a linked list, every node has two parts, one that stores data and the other represents address of next node. The head pointer is given as Top, all insertions and deletions occur at the node pointed by Top.



Top

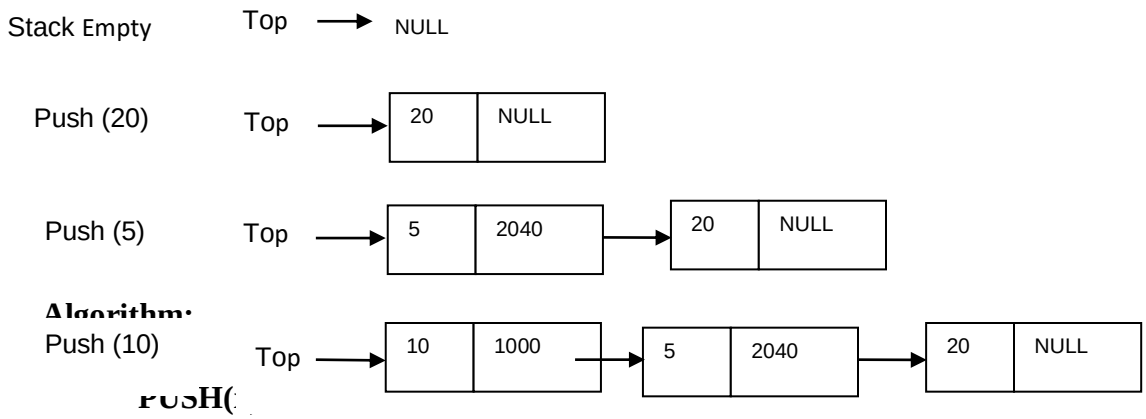
Fig.3.2.4.1 Array Representation of Stack

OPERATIONS ON STACK

Linked List representation supports the two basic operations of stack, push and pop.

Push operation

Initially, when the stack is empty the pointer top points NULL. When an element is added using the push operation, top is made to point to the latest element whichever is added.



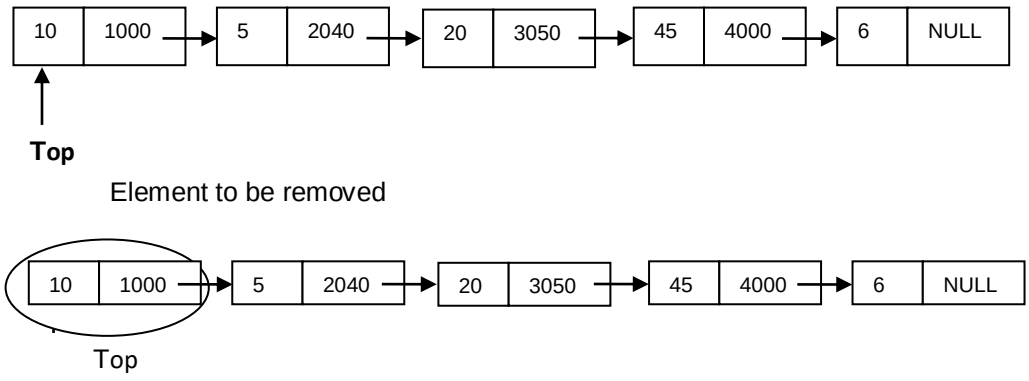
Algorithm:

PUSH(

Allocate memory for a new node and name it as Temp
Temp->data = x
Temp->link = Top
Top = Temp
End

Pop operation

The data in the topmost node of the stack, which is to be removed, is stored in a variable called item. Then a temporary pointer is created to point to top. The top is now safely moved to the next node below it in the stack. Temp node is then deleted and the item is returned.



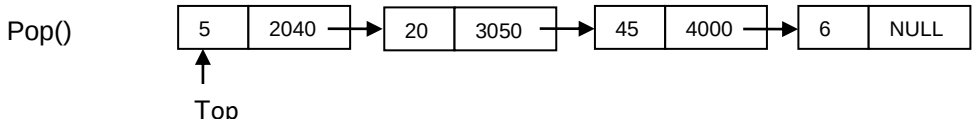


Fig.3.2.4.1 Linked List Representation of Stack

Algorithm:

```
POP()
  if Top == NULL
      print "Underflow"
      return
  else
      Temp = Top
      Item = Temp->data
      Top = Top->link
      delete Temp
      return Item
End
```

Implementation of stack operations using linked lists in Python

```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None
class Stack:                    # default value of head is NULL
    def __init__(self):
        self.head = None
    def isempty(self):          # Checks if stack is empty
```

```
        if self.head == None:
            return True
        else:
            return False

    # Method to add data to the stack
    def push(self,data):
        if self.head == None:
            self.head=Node(data)
        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode

# Remove element that is the current head (start of the stack)
    def pop(self):
        if self.isempty():
            return None
        else:
            temp = self.head
            self.head = self.head.next
            temp.next = None
            return temp.data
# Returns the head node data
    def peek(self):
        if self.isempty():
            return None
        else:
            return self.head.data
# Prints out the stack
    def display(self):
        temp = self.head
        if self.isempty():
            print("Stack Underflow")
        else:
            while(temp != None):
                print(temp.data,"->",end = " ")
                temp = temp.next
            print("NULL")
            return
```

```
# Driver code
St = Stack()
St.push(11)
St.push(22)
St.push(33)
St.push(44)

# Display stack elements
print("\nElements in Stack : ")
St.display()

# Print top elementof stack
print("\nTop element is ",St.peek())

# Delete top elements of stack
St.pop()
St.pop()

# Display stack elements
print("\nElements in Stack : ")
St.display()

# Print top elementof stack
print("\nTop element is ",St.peek())
```

Output:

```
Elements in Stack :
44 -> 33 -> 22 -> 11 -> NULL

Top element is  44

Elements in Stack :
22 -> 11 -> NULL

Top element is  22
>>>
```

Pros:-

- No fixed size declaration.
- New elements can be stored anywhere and a reference is created for the new element using pointers.

Cons:-

- Requires extra memory to keep details about next node.
- Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.

Difference between an Array and Stack ADT

Stack:

- Size of the stack keeps on changing with insertion/deletionoperation.
- Stack can store elements of different data types [Heterogeneous].

Array:

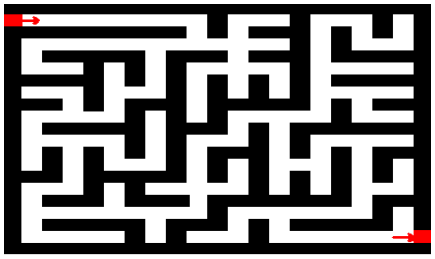
- Size of the array is fixed at the time of declaration itself
- Array stores elements of similar data type [Homogeneous].

3.3 APPLICATIONS OF STACK

In this section, simple applications of stack that are extensively used in computer applications are discussed. Few applications of Stack are listed as below:

- Stack - undo\redo operation in word processors.
- Expression evaluation and syntax parsing.
- Many virtual machines like JVM are stack oriented**.**
- DFS Algorithm - Depth First Search.
- Graph Connectivity.
- LIFO scheduling policy of CPU.
- When a processor receives an interrupt, it completes its execution of the current instruction, then pushes the process's PCB to the stack and then perform the ISR (Interrupt Service Routine)

- When a process calls a function inside a function - like recursion, it pushes the current data like local variables onto the stack to be retrieved once the control is returned.
- Reverse polish AKA postfix notations.
- Used in IDEs to check for proper parenthesis matching.
- Browser back-forth button.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack
- Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



Once you reach a dead end, you must backtrack. But backtrack to where?,the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

3.3.1 Reversing a list

A list of numbers are reversed with a stack by just pushing all elements one by one into the stack and then elements are popped one by one and stored back starting from the first index of the list.

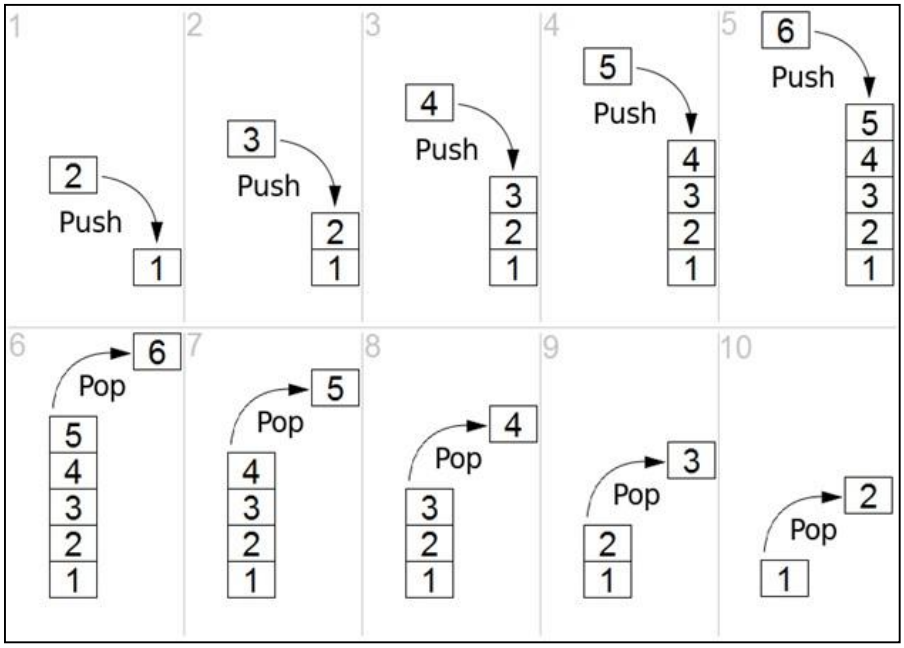


Fig.3.3.1 Reversing a list using Stack

The logic behind for implementing reverse of a string:

- Read a string.
- Push all characters until NULL is not found - Characters will be stored in stack variable.
- Pop all characters until NULL is not found - Stack is a LIFO technique, so last character will be pushed first and finally will get reversed string in a variable in which input string is stored.

Implementation in Python

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
```

```
        return self.items == []
    def push(self, data):
        self.items.append(data)
    def pop(self):
        return self.items.pop()
    def display(self):
        for data in reversed(self.items):
            print(data,end=" ")
    def insert_at_bottom(s, data):
        if s.is_empty():
            s.push(data)
        else:
            popped = s.pop()
            insert_at_bottom(s, data)
            s.push(popped)
    def reverse_stack(s):
        if not s.is_empty():
            popped = s.pop()
            reverse_stack(s)
              insert_at_bottom(s, popped)
s = Stack()
data_list = input('Enter the elements to push : \n').split()
for data in data_list:
   s.push(int(data))
print('The elements in stack :')
s.display()
reverse_stack(s)
print('\nAfter reversing : ')
s.display()
```

Output:

```
Enter the elements to push :
10 15 20 25 30 35
```

The elements in stack :
35 30 25 20 15 10
After reversing :
10 15 20 25 30 35

3.3.2 Recursion

All recursive functions are examples of implicit application of Stack ADT.A recursive function is defined as a function that calls itself again to complete a smaller version of task until a final call, which does not require a call to itself is met. Therefore recursion is implemented for solving complex problems in terms of smaller and easily solvable problems.

To understand recursive functions, let's consider the simplest example of calculating factorial of a number.

Factorial Calculation using Recursive Approach

To calculate n!, multiply the given number with factorial of that number less than one. i.e., n! = n ^{x} (n-1) !

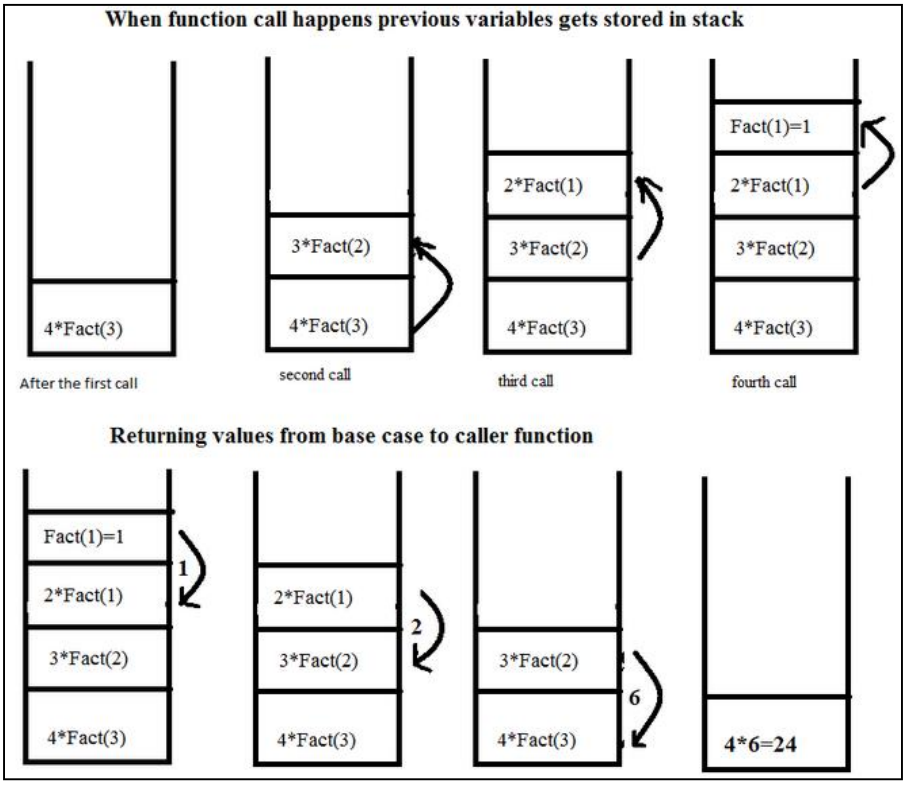


Fig.3.3.2 Factorial Calculation using Stack

Consider an example 5!

5! = 5 x (4)! , where 4! = 4 x (3)! , where 3! = 3 x (2)! ,where 2! = 2 x(1)!,where 1!=1;

i.e., 5! = 5 x 4x 3x 2x 1

= 120

Python Code:

```
def factorial(n):
    if(n <= 1):
        return 1
    else:
        return(n*factorial(n-1))
n = int(input("Enter number :"))
print("Factorial :",factorial(n))
```

Output:

```
Enter number : 5
5
Factorial : 120
>>>
```

3.3.3 Conversion of Infix Expression to Postfix Expression

The stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. The operands can be numeric values or numeric variables. The operators used in an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentiation.

The arithmetic expression expressed in its normal form is said to be Infix notation, as shown: A * B

The above expression in prefix form would be represented as follows:* AB

The same expression in postfix form would be represented as follows:AB *

Hence the given expression in infix form is first converted to postfix form and then evaluated to obtain the results. Postfix expressions are represented as Reverse Polish notation. To understand the conversion of Infix to Postfix expression, knowledge about operator precedence is important.

Operator Precedence:

The Precedence of the operators takes a crucial place while evaluating expressions. The top operator in the table has the highest precedence. As per the precedence, the operators will be pushed to the stack.

Table.3.3.3 Operator Precedence

S.No	Operator	Associativity	S.No	Operator	Associativity	S.No	Operator	Associativity
1	( )	Left to Right	11	&	Right to Left	21	|	Left to Right
2	[ ]	Left to Right	12	sizeof	Right to Left	22	&&	Left to Right
3	.	Left to Right	13	* / %	Left to Right	23	||	Left to Right
4	->	Left to Right	14	+ -	Left to Right	24	? :	Right to Left
5	++ --	Left to Right	15	<< >>	Left to Right	25	=	Right to Left
6	++ --	Right to Left	16	< <=	Left to Right	26	+= -=	Right to Left
7	+ -	Right to Left	17	> >=	Left to Right	27	*= /=	Right to Left
8	! ~	Right to Left	18	== !=	Left to Right	28	%= &=	Right to Left
9	(type)	Right to Left	19	&	Left to Right	29	^= |=	Right to Left
10	*	Right to Left	20	^	Left to Right	30	<<= >>=	Right to Left

The function to convert an expression from infix to postfix consists following steps:

1. Every character of the expression string is scanned in a while loop until the end of the expression is reached.
2. Following steps are performed depending on the type of character scanned.
    (a) If the character scanned happens to be a space then that character is skipped.
    (b) If the character scanned is a digit or an alphabet, it is added to the target string pointed to by t.
    (c) If the character scanned is a closing parenthesis then it is added to the stack by calling push( ) function.
    (d) If the character scanned happens to be an operator, then firstly, the topmost element from the stack is retrieved. Through a while loop, the priorities of the character scanned and the character popped 'opr' are compared. Then following steps are performed as per the precedence rule.
        i. If 'opr' has higher or same priority as the character scanned, then opr is added to the target string.
        ii. If opr has lower precedence than the character scanned, then the loop is terminated. Opr is pushed back to the stack. Then, the character scanned is also added to the stack.
    (e) If the character scanned happens to be an opening parenthesis, then the operators present in the stack are retrieved through a loop. The loop continues till it does not encounter a closing parenthesis. The operators popped, are added to the target string pointed to by t.
3. Now the string pointed by t is the required postfix expression.

Example: Convert A * (B + C) * D to postfix notation.

Step No	Input	Stack	Output
1	A	Empty	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	END	Empty	A B C + * D *

Notes:

- In this table, the stack grows toward the left. Thus, the top of the stack is the leftmost symbol.
- Step No.3, the left parenthesis was pushed without popping the * because * had a lower priority then "(".
- Step No.5, "+" was pushed without popping "(" because you never pop a left parenthesis until you get an incoming right parenthesis. In other words, there is a distinction between incoming priority, which is very high for a "(", and instack priority, which is lower than any operator.
- Step No.7, the incoming right parenthesis caused the "+" and "(" to be popped but only the "+" as written out.
- Step.No.8, the current "*" had equal priority to the "*" on the stack. So the "*" on the stack was popped, and the incoming "*" was pushed onto the stack.

Pseudo-code:

```
CONVERSION(INFIX)
Begin
Read infix
```

```
L=Length(infix)
J=1
For i=1 to L
Do
        C=infix(i)
        If  C is a  number OR Alphabet Then
                Postfix[j]= c
                j=j+1
        if C= ‘( ‘ Then
                Push ( C )
        if C= ‘*’ ||’ /’|| ‘+ ‘||’ –‘ ||’ % ‘
                If top = 0 then
                        Push( C )
                Else If Priority ( C )< Priority (Stack[top]) then
                        Postfix[j]= pop()
                        J=j+1
                        Push( C )
                Else
                        Push ( C )

if  C= ‘) ‘
                Repeat
postfix[j]=pop()
                  j=j+1
                 Until (stk[top]<>'(')
pop()
        End if
End For

Repeat
        postfix[j]=pop()
j=j+1
Until (top > = 0)
Write postfix        # Print the resultant postfix string
END CONVERSION(INFIX)


FUNCTION PRIORITY(CHAR CH)
Begin
if (ch=='*'||ch=='/'||ch=='%') then
return 2
else if(ch=='+'||ch=='-') then
return 1
else
return 0
END FUNCTION PRIORITY
```

EXAMPLE:

A+(B*C-(D/E – F)*G)*H

Step No	Input	Stack	Output
1	A+(B*C-(D/E-F)*G)*H	Empty	-
2	+(B*C-(D/E-F)*G)*H	Empty	A
3	(B*C-(D/E-F)*G)*H	+	A
4	B*C-(D/E-F)*G)*H	+(	A
5	*C-(D/E-F)*G)*H	+(	AB
6	C-(D/E-F)*G)*H	+(*	AB
7	-(D/E-F)*G)*H	+(*	ABC
8	(D/E-F)*G)*H	+(-	ABC*
9	D/E-F)*G)*H	+(-(	ABC*
10	/E-F)*G)*H	+(-(	ABC*D
11	E-F)*G)*H	+(-(/	ABC*D
12	-F)*G)*H	+(-(/	ABC*DE
13	F)*G)*H	+(-(-	ABC*DE/
14	F)*G)*H	+(-(-	ABC*DE/
15	)*G)*H	+(-(-	ABC*DE/F
16	*G)*H	+(-	ABC*DE/F-
17	G)*H	+(-*	ABC*DE/F-
18	)*H	+(-*	ABC*DE/F-G
19	*H	+	ABC*DE/F-G*-
20	H	+*	ABC*DE/F-G*-
21	End	+*	ABC*DE/F-G*-H
22	End	Empty	ABC*DE/F-G*-H*+

Python code for converting a given expression from Infix to Postfix notation:

```
class Conversion:
```

```
 # Constructor to initialize the class variables
def __init__(self, capacity):
    self.top = -1
    self.capacity = capacity
    self.array = []
    self.output = []
    self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

# check if the stack is empty
def isEmpty(self):
    if self.top == -1:
        return True
  else :
      return False

# Return the value of the top of the stack
def peek(self):
    return self.array[-1]

# Pop the element from the stack
def pop(self):
    if not self.isEmpty():
        self.top -= 1
        return self.array.pop()
    else:
        return "$"

# Push the element to the stack
def push(self, op):
    self.top += 1
    self.array.append(op)

# A utility function to check is the given character is operand
def isOperand(self, ch):
    return ch.isalpha()

# Check if the precedence of operator is strictly less than top of stack or not
def notGreater(self, i):
    try:
        a = self.precedence[i]
        b = self.precedence[self.peek()]
        if a  <= b :
            return True
```

```
            else :
                return False
        except KeyError:
            return False


    # Function that converts given infix expression to postfix expression
    def infixToPostfix(self, exp):
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

            # If the character is an '(', push it to stack
            elif i  == '(':
                self.push(i)

            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

        # pop all the operator from the stack
        print("\nPostfix Notation : ")
        while not self.isEmpty():
            self.output.append(self.pop())
        print("".join(self.output))

# Program to test above function
exp = "a+b*(c^d-e)^(f+g*h)-i"
print("\n\nInfix Notation :",exp)
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

Output :

Infix Notation : a+b*(c^d-e)^(f+g*h)-i
Postfix Notation :
abcd^e-fgh*+^*+i-

3.3.4 Evaluation of Expression given in Postfix form:

The program takes the input expression in postfix form. This expression is scanned character by character. If the character scanned is an operand, then first it is converted to a digit form and then it is pushed onto the stack. If the character scanned is a blank space, then it is skipped. If the character scanned is an operator, then the top two elements from the stack are retrieved. An arithmetic operation is performed between the two operands. The type of arithmetic operation depends on the operator scanned from the string s. The result is then pushed back onto the stack. These steps are repeated as long as the string s is not exhausted. Finally the value in the stack is the required result.

In the first example, the conversion of infix to postfix notation is discussed; nextherethe evaluationof postfix notation to obtain the resultant value is discussed.

Example:

Infix Notation :A * (B + C) * DPostfix Notation :A B C + * D *

Let's assume the values of A,B,C,D as 2,3,4,5 respectively.

Evaluate the postfix expression 2 3 4 + * 5 * to obtain the resultant value.

Step No	Input	Stack (grows toward left)
1	2	2
2	3	3 2
3	4	4 3 2

4	+	7 2
5	*	14
6	5	5 14
7	*	70

Notes:

- Step No 4: an operator is encountered, so 4 and 3 are popped, summed, then pushed back onto stack.
- Step No 5: operator * is current token, so 7 and 2 are popped, multiplied, pushed back onto stack.
- Step No 7: stack top holds correct value.
- Notice that the postfix notation has been created to properly reflect operator precedence. Thus, postfix expressions never need parentheses.

Pseudo-code:

```
EVALUATION ( POSTFIX )
Begin
Read postfix
L=Length(postfix)
For i=1 to L
Do
        C=infix(i)
        If C is a number or Alphabet then
                # Get the value of C and store it in the stack
                Read n
                Push(n)
        Else if C= '*' ||' /'|| '+ '||' –' ||' % '
        Then
                Call EVAL(c)
        End if
End for
Result = pop()     # Print the  result
Write Result
End
```

```
EVAL(CHAR C)
Begin
        x=pop()
        y=pop()
Switch(C)
Begin
case '+': z=x+y
case '-': z=x-y
case '*': z=x*y
case '/': z=x/y
case '%': z=x%y
End Switch
push(z)
End EVAL
```

Python code for evaluating Postfix notation:

```
class Evaluate:
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        self.array = []

    def isEmpty(self):
        if self.top == -1:
            return True
        else:
            return False

    def peek(self):
        return self.array[-1]

    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    def push(self, op):
```

```
        self.top += 1
        self.array.append(op)

    def evaluatePostfix(self, exp):
        for i in exp:
            if i.isdigit():
                self.push(i)
            else:
                val1 = self.pop()
                val2 = self.pop()
                self.push(str(eval(val2 + i + val1)))
            return int(self.pop())

# Program to test above function
exp = "231*+9-"
print ("\nPostfix Notation : ",exp)
obj = Evaluate(len(exp))
print ("Postfix Evaluation : ",obj.evaluatePostfix(exp))
```

Output:

```
Postfix Notation : 231*+9-
Postfix Evaluation : 2
```

Examples

Infix Expression	**Prefix Expression**	**Postfix Expression**
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Points to Remember

- *A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop. Stack is also called as Last-In-First-Out (LIFO) list.*
- *Stacks are implemented using Arrays or Linked Lists.*
- *The storage requirements of linked representation of stack with n elements is O(n) and the time required for all stack operations is given as O(1).*
- *All Recursive function calls are implemented using System Stack.*

Exercises

1. Check if given expression is balanced expression or not. For example, Input : ((()))Output: 1Input :)((Output : -1
2. Find duplicate parenthesis in an expression : ((a+b)+((c+d)))
3. Evaluate given postfix expression : A B C * + D +
4. Decode the given sequence to construct minimum number without repeated digits.
5. Explain how to implement two stacks in one array A[1…n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.
   Stack Implementation using Linked List.
6. Design a stack which returns minimum element without using auxiliary stack.
7. Reverse a string without using recursion.
8. Reverse a string using stack data structure.
9. In order Tree Traversal | Iterative & Recursive.
10. Preorder Tree Traversal | Iterative & Recursive.
11. Post order Tree Traversal | Iterative & Recursive.
12. Check if two given binary trees are identical or not | Iterative & Recursive.
13. Reverse Level Order Traversal of Binary Tree.
14. Reverse given text without reversing the individual words.
15. Find all binary strings that can be formed from given wildcard pattern.

16. Depth First Search (DFS) | Iterative & Recursive Implementation.
17. Invert given Binary Tree | Recursive and Iterative solution.
18. Longest Increasing Subsequence.
19. Implement Queue using Stacks.
20. Design a stack with operations on middle element.
21. The Stock Span Problem.
22. Check for balanced parentheses in an expression.
23. Next Greater Frequency Element.
24. Number of NGEs to the right.
25. Maximum product of indexes of next greater on left and right.
26. The Celebrity Problem.
27. Iterative Tower of Hanoi.
28. Sorting array using Stacks.

Check your Understanding:

1. What is the best case time complexity of deleting a node in Singly Linked list?
   a) O (n)
   b) O (n^2)
   c) O (nlogn)
   d) O (1)

2. Recursion follows divide and conquer technique to solve problems. State True or False.

3. Consider you have a stack whose elements in it are as follows.
   5 4 3 2 << top
   Where the top element is 2.
   You need to get the following stack
   6 5 4 3 2 << top
   The operations that needed to be performed are (You can perform only push and pop):
   a) Push(pop()), push(6), push(pop())
   b) Push(pop()), push(6)

c) Push(pop()), push(pop()), push(6)

d) Push(6)

4. What does ‘stack overflow’ refer to?

   a) accessing item from an undefined stack

   b) adding items to a full stack

   c) removing items from an empty stack

   d) index out of bounds exception

5. Which of the following statement is incorrect with respect to evaluation of infix expression algorithm?

   a) Operand is pushed on to the stack

   b) If the precedence of operator is higher, pop two operands and evaluate

   c) If the precedence of operator is lower, pop two operands and evaluate

   d) The result is pushed on to the operand stack

6. The process of accessing data stored in a serial access memory is similar to manipulating data on a :

   a) n-ary tree

   b) queue

   c) Stack

   d) Array

7. Representation of data structure in memory is known as :

   a) Storage Structure

   b) File Structure

   c) Record Structure

   d) Abstract Data Type

8. The data structure required to check whether an expression contains balanced parenthesis is :

a) n-ary tree

b) queue

c) Stack

d) Array

9. Which of the following data structures can be used for parentheses matching?

a) n-ary tree

b) queue

c) priority queue

d) stack

10. What will be the output after performing these sequence of operations : push(20); push(4);
pop();pop();push(5);top();

a) 20

b) 4

c) stack underflow

d) 5

11. Which of the following real world scenarios would you associate with a stack data structure?

a) piling up of chairs one above the other

b) people standing in a line to be serviced at a counter

c) offer services based on the priority of the customer

d) tatkal Ticket Booking in IRCTC

12. What is the time complexity of pop() operation when the stack is implemented using an array?

a) O(1)

b) O(n)

c) O(logn)

d) O(nlogn)

13. Array implementation of Stack is not dynamic, which of the following statements supports this argument?

a) space allocation for array is fixed and cannot be changed during run-time

b) user unable to give the input for stack operations

c) a runtime exception halts execution

d) improper program compilation

14. Evaluation of infix expression is done based on precedence of operators.

a) True

b) False

15. Of the following choices, which operator has the lowest precedence?

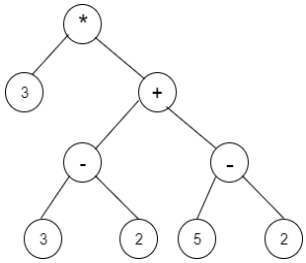
a) ^

b) +

c) /

d) #

16. From the given expression tree, identify the infix expression, evaluate it and choose the correct result.



a) 5

b) 10

c) 12

d) 16

17. What is the result of the following operation : Top (Push (S, X))

a. X
b. Null
c. S
d. None of the above

18. Evaluate the following statement using infix evaluation algorithm and choose the correct answer. 1+2*3-2
    a) 3
    b) 6
    c) 5
    d) 4

19. Convert the Expression ((a + B) * C - (d - E) ^ (f + G)) To Equivalent Prefix and Postfix Notations?

20. Evaluate the following infix expression using algorithm and choose the correct answer. a+b*c-d/e^f where a=1, b=2, c=3, d=4, e=2, f=2.
    a) 6 b) 8 c) 9 d) 7

21. Convert the infix to postfix for A-(B+C)*(D/E)
    a. ABC+DE/*-
    b. ABC-DE/*-
    c. ABC-DE*/-
    d. None of the above

Answers:

1. D
2. True
3. A
4. B

5. B
6. C
7. D
8. C
9. D
10. D
11. A
12. A
13. A
14. A
15. B
16. C
17. A
18. C
19. Prefix Notation: ^ – * +ABC – DE + FG

    Postfix Notation: AB + C * DE – – FG + ^

20. A
21. A

Case Study:

I. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. push(x) -- Push element x onto stack.• pop() -- Removes the element on top of the stack. top() -- Get the top element. getMin() -- Retrieve the minimum element in the stack.

II. Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.An input string is valid if:
    1. Open brackets must be closed by the same type of brackets.
    2. Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

1. Input: "()"Output: true
2. Input: "()[]{}" Output: true
3. Input: "(]" Output: false
4. Input: "([)]" Output: false
5. Input: "{[]}" Output: true

III. Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

a. For any array, rightmost element always has next greater element as -1.
b. For an array which is sorted in decreasing order, all elements have next greater element as -1.
c. For the input array [4, 5, 2, 25], the next greater elements for each element are as follows :

Element	Next Greater
4	5
5	25
2	25
25	-1

IV. Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit.

For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 1, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red).

V. Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.

For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8} }. The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}. Similarly {5, 7} and {6, 8} should be merged and become {5, 8}.

VI. Length of the longest valid substring

Given a string consisting of opening and closing parenthesis, find length of the longest valid parenthesis substring.

Examples:

Input : (((()
Output : 2
Explanation : ()

Input:)()())
Output : 4
Explanation: ()()

Input: ()((())))
Output: 6
Explanation: ()(())

VII. Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example: Input: [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6.

VIII. The Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days. The

span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day. For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}

IX. The Celebrity Problem

In a party of N people, only one person is known to everyone. Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "does A know B? ". Find the stranger (celebrity) in minimum number of questions. We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function HaveAcquaintance(A, B) which returns true if A knows B, false otherwise. How can we solve the problem?

MATRIX = [[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 1, 0]]