COMPUTER SCIENCE With python

Textbook for
Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

PADHAAII

10

Data Structures – II : Stacks and Queues

10.1 INTRODUCTION

Any good programming language course does cover data structures. A *data structure*, in general, refers to a particular way of storing and organizing data in a computer so that it can be used most efficiently to give optimal performance. Different kinds of *data structures* are designed and used for different kinds of applications. The significance of data structures lies in the fact that they help a computer perform tasks in most efficient and productive manner.

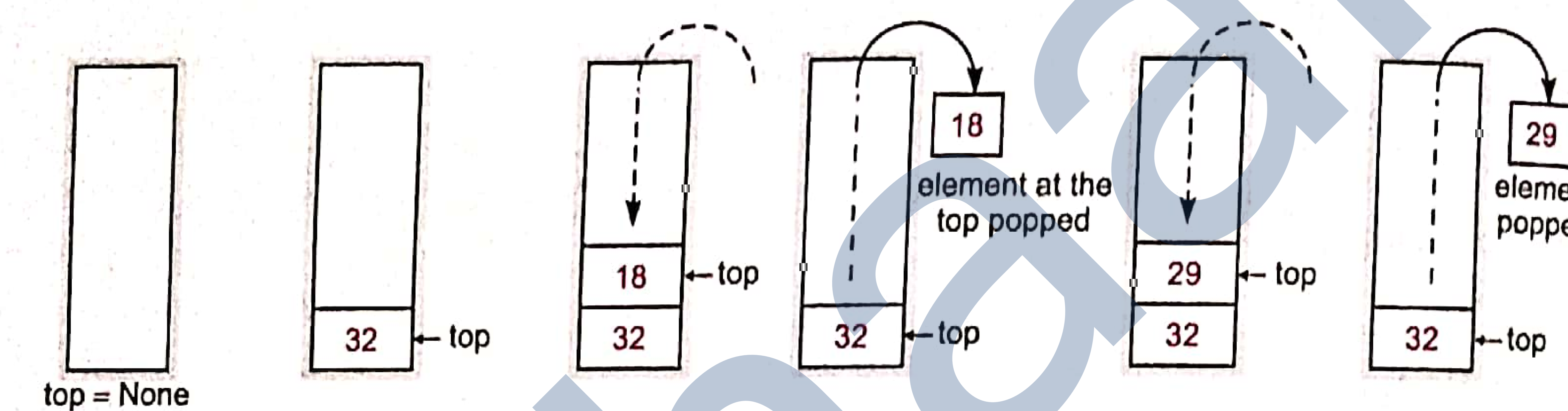
Some data structures are highly specialized to specific tasks. **Stacks** and **Queues** are such data structures. In this chapter, we shall be talking about the basics of these data structures and how these can be implemented in Python.

10.2 STACKS

A stack is a linear structure implemented in LIFO (**Last In First Out**) manner where insertions and deletions are restricted to occur only at one end – ***Stack's top***. LIFO means element last inserted would be the first one to be deleted. Thus, we can say that a stack is a list of data that follows these rules :

1. Data can only be removed from the top (*pop*), *i.e.*, the element at the *top of the stack*. The removal of element from a stack is technically called **POP operation**.
2. A new data element can only be added to the *top of the stack* (*push*). The insertion of element in a stack is technically called **PUSH operation**.

Consider Fig. 10.1 that illustrates the operations (**Push** and **Pop**) on a stack.



(*a*) Stack empty (*b*) Push '32' (*c*) Push '18' (*d*) Pop 1 element (*e*) Push 29 (*f*) Pop 1 element

Notice, all stack operations - **Push** or **Pop-take** place at one end-the ***stack's top***

Figure 10.1 Stack operations – *Push* and *Pop*.

The stack is a *dynamic data structure* as it can grow (with increase in number of elements) or shrink (with decrease in number of elements). A *static data structure*, on the other hand, is the one that has fixed size.

STACK

A Stack is a linear list implemented in LIFO – Last In First Out manner where insertions and deletions are restricted to occur only at one end – *Stack's top*.

Other Stack Terms

There are *some* other terms related to stacks, such as *Peek*, *Overflow* and *Underflow*.

Peek Refers to inspecting the value at the ***stack's top*** without removing it. It is also sometimes referred as **inspection**.

Overflow Refers to situation (ERROR) when one tries to push an item in stack that is full. This situation occurs when the size of the stack is fixed and cannot grow further or there is no memory left to accommodate new item.

Underflow Refers to situation (ERROR) when one tries to pop/delete an item from an empty stack. That is, stack is currently having no item and still one tries to pop an item.

NOTE

The technical terms for insertion-in-a-stack and deletion-from-stack are Push and Pop respectively.

Consider some examples illustrating stack-functioning in limited-size stack. (Please note, we have bound fixed the capacity of the stack for understanding purposes.)

EXAMPLE 10.1 *Given a Bounded Stack of capacity 4 which is initially empty, draw pictures of the stack after each of the following steps. Initially the Stack is empty.*

(*i*) Stack is empty (*ii*) Push 'a' (*iii*) Push 'b' (*iv*) Push 'c'
(*v*) Pop (*vi*) Push 'd' (*vii*) Pop (*viii*) Push 'e'
(*ix*) Push 'f' (*x*) Push (*xi*) Pop (*xii*) Pop
(*xiii*) Pop (*xiv*) Pop (*xv*) Pop

Solution.

top = ↑

(*i*) Stack is empty (top = None)

0	1	2	3

(*ii*) Push 'a' ***top = 0***

0	1	2	3
a			

(*iii*) Push 'b' ***top = 1***

0	1	2	3
a	b		

(*iv*) Push 'c' ***top = 2***

0	1	2	3
a	b	c	

(*v*) Pop ***top = 1***

0	1	2	3
a	b		

c

(*vi*) Push 'd' ***top = 2***

0	1	2	3
a	b	d	

(*vii*) Pop ***top = 1***

0	1	2	3
a	b		

d

(*viii*) Push 'e' ***top = 2***

0	1	2	3
a	b	e	

(*ix*) Push 'f' ***top = 3***

0	1	2	3
a	b	e	f

(*x*) Push 'g' ***top = 3***

0	1	2	3
a	b	e	f

OVERFLOW

[OVERFLOW because the stack is bounded, it cannot grow. If it could grow, then there would have been no OVERFLOW until no memory is left.

In Python, (for stacks implemented through lists) since Lists can grow, OVERFLOW condition does not arise until all the memory is exhausted.]

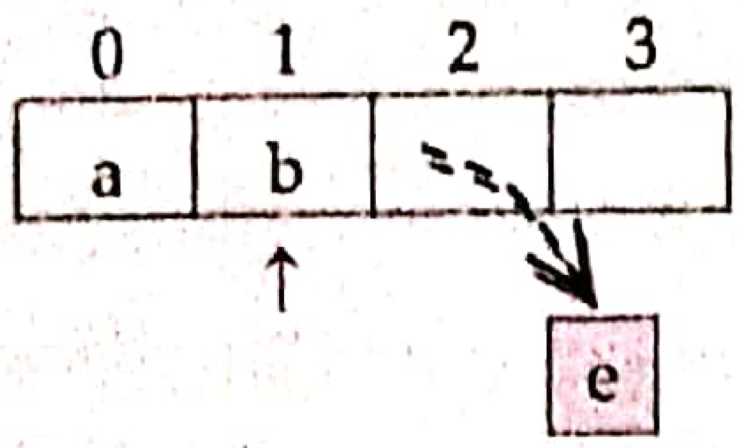
(*xi*) Pop ***top = 2***

0	1	2	3
a	b	e	

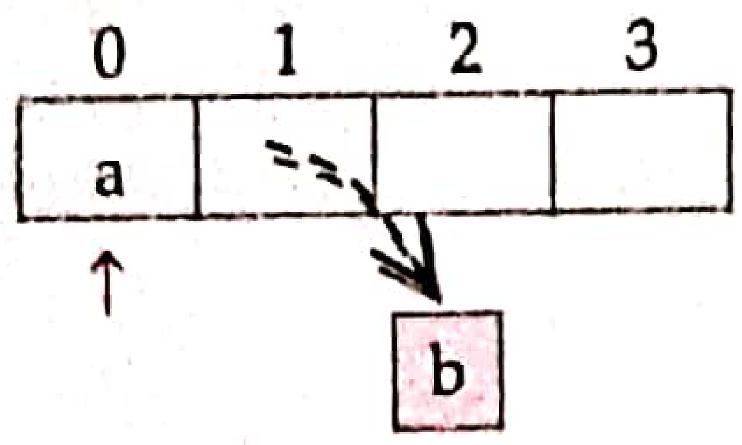
f

top = ↑

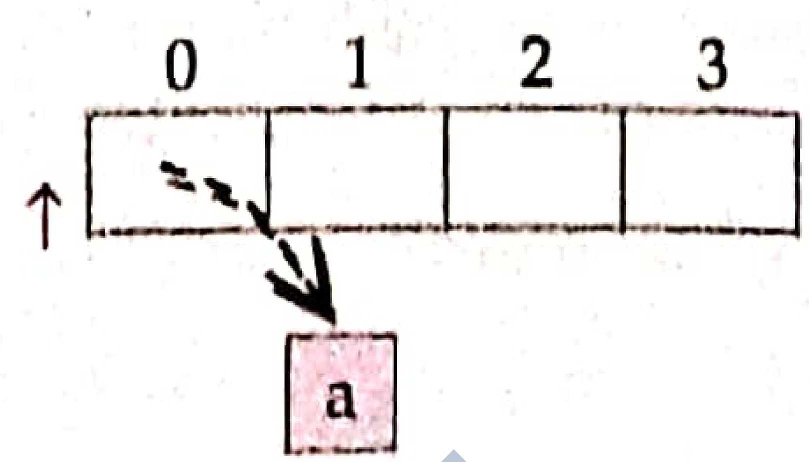
(*xii*) Pop ***top = 1***



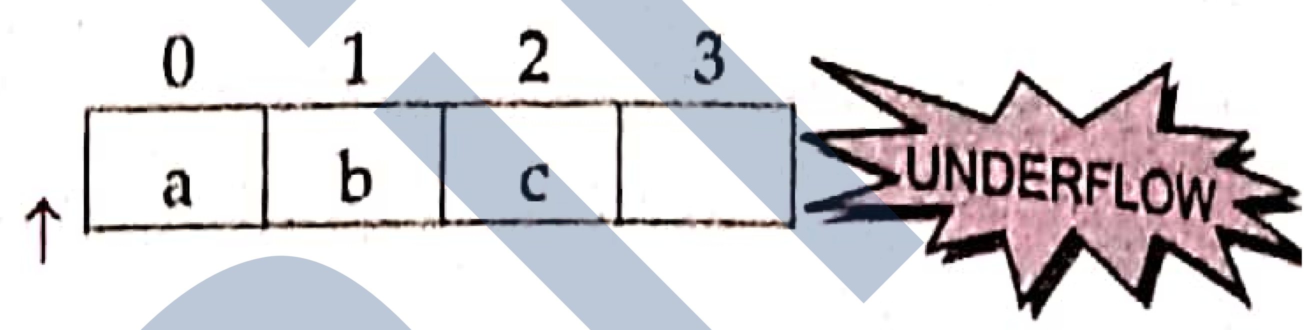
(*xiii*) Pop top = 0



(*xiv*) Pop top = None



(*xv*) Pop top = None



10.2.1 Implementing Stack in Python

In Python, you can use Lists to implement stacks. Python offers us a convenient set of methods to operate lists as stacks.

For various stack operations, we can use a list say ***Stack*** and use Python code as described below :

Peek We can use : `<Stack>[top]`
where **<Stack>** is a list ; ***top*** is an integer having value equal to **len(<Stack>) - 1**.

Push We can use : `<Stack>.append(<item>)`
where **<item>** is the item being pushed in the Stack.

Pop We can use : `<Stack>.pop()`
it removes the last value from the *Stack* and returns it.

Let us now implement a stack of numbers through a program.

```
def Push(stk, item) :
```

Program 10.1 Python program to implement stack operations.

```
#############    STACK IMPLEMENTATION    ##############
"""
    Stack: implemented as a list
    top : integer having position of topmost element in Stack
"""
def isEmpty( stk ) :
    if stk == [] :
        return True
    else :
        return False
```

```
def Push(stk, item) :
    stk.append(item)
    top = len(stk) - 1
def Pop(stk) :
    if isEmpty(stk) :
        return "Underflow"
    else :
        item = stk.pop()
        if len (stk) == 0:
            top = None
        else :
            top = len(stk) - 1
        return item
def Peek(stk) :
    if is Empty(stk) :
        return "Underflow"
    else :
        top = len(stk) - 1
        return stk[top]
def Display(stk) :
    if isEmpty(stk) :
        print("Stack empty")
    else :
        top = len(stk) - 1
        print(stk[top], "<-top")
        for a in range(top-1, -1 , -1 ) :
            print(stk[a])
# __main__
Stack = []                    # initially stack is empty
top = None
while True :
    print("STACK OPERATIONS")
    print("1. Push")
    print("2. Pop")
    print("3. Peek")
    print("4. Display stack")
    print("5. Exit")
    ch = int(input("Enter your choice (1-5) :"))
    if ch == 1 :
        item = int(input("Enter item :"))
        Push(Stack, item)
    elif ch == 2 :
        item = Pop(Stack)
        if item == "Underflow" :
            print("Underflow! Stack is empty!")
        else :
            print("Popped item is", item)
```

```
    elif ch == 3 :
        item = Peek(Stack)
        if item == "Underflow" :
            print("Underflow! Stack is empty!")
        else :
            print("Topmost item is", item)
    elif ch == 4 :
        Display(Stack)
    elif ch == 5 :
        break
    else :
        print("Invalid choice!")
```

Sample run of the above program is as shown below :

```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :8
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :2
_______________________
```

```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :1
Enter item :4
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :4
4 <- top
2
8
6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice ( 1-5) :3
Topmost item is  4
_______________________
```

```
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :4
4 <- top
2
8
6
_______________________
STACK OPERATIONS
1. Push
2. Pop
3. Peek
4. Display stack
5. Exit
Enter your choice (1-5) :5
```

Types of Stack–Itemnode

An item stored in a stack is also called item-node sometimes. In the above implemented stack, the stack contained item-nodes containing just integers. If you want to create stack that may contain logically group information such as member details like : *member no, member name, age* etc. For such a stack the item-node will be a list containing the member details and then this list will be entered as an item to the stack. (See figure 10.2 below).

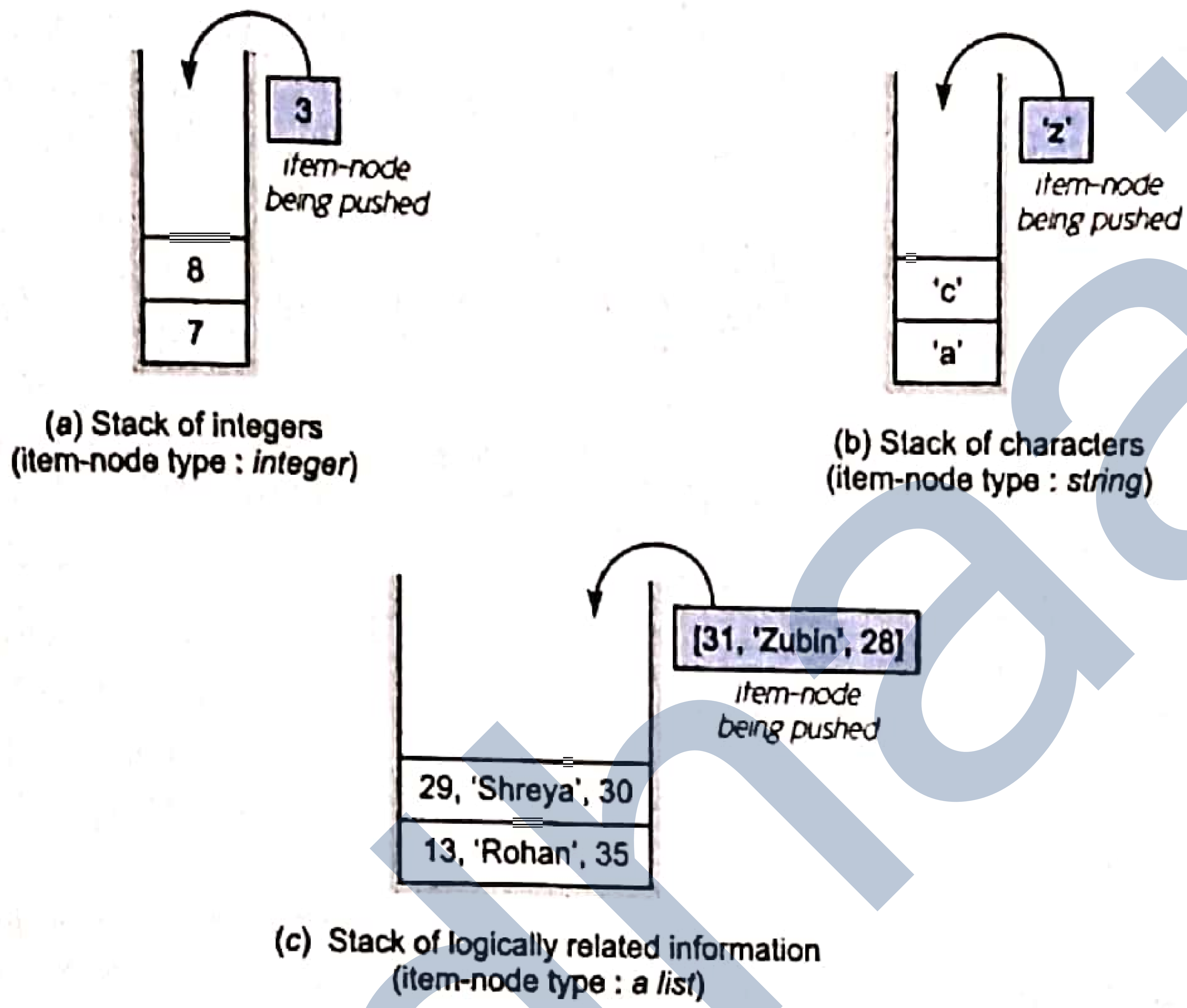


Figure 10.2 Different types of stack item-nodes.

- For stack of Fig. 10.2(*a*), the stack will be implemented as ***Stack of integers*** as *item-node* is of integer type.
- For stack of Fig. 10.2(*b*), the stack will be implemented as ***Stack of strings*** as *item-node* is of string type.
- For stack of Fig. 10.2(*c*), the stack will be implemented as ***Stack of lists*** as *item-node* is of list type. Solved problem 20 implements such a stack.

10.2.2 Stack Applications[1]

There are several applications and uses of stacks. The stacks are basically applied where LIFO (Last In First Out) scheme is required.

10.2.2A Reversing a Line

A simple example of stack application is reversal of a given line. We can accomplish this task by pushing each character on to a stack as it is read. When the line is finished, characters are then

1. **Some other applications of stacks include :**
   (*a*) **The compilers use stacks to store the previous state of a program when a function is called, or during recursion.**
   (*b*) **One of the most important applications of Stacks is** *backtracking*. Backtracking is used in large number of puzzles like *n-Queen* problem, *Sudoku* etc and optimization problems such as *knapsack problem*.

popped off the stack, and they will come off in the reverse order as shown in Fig. 10.3. The given line is : *Stack*

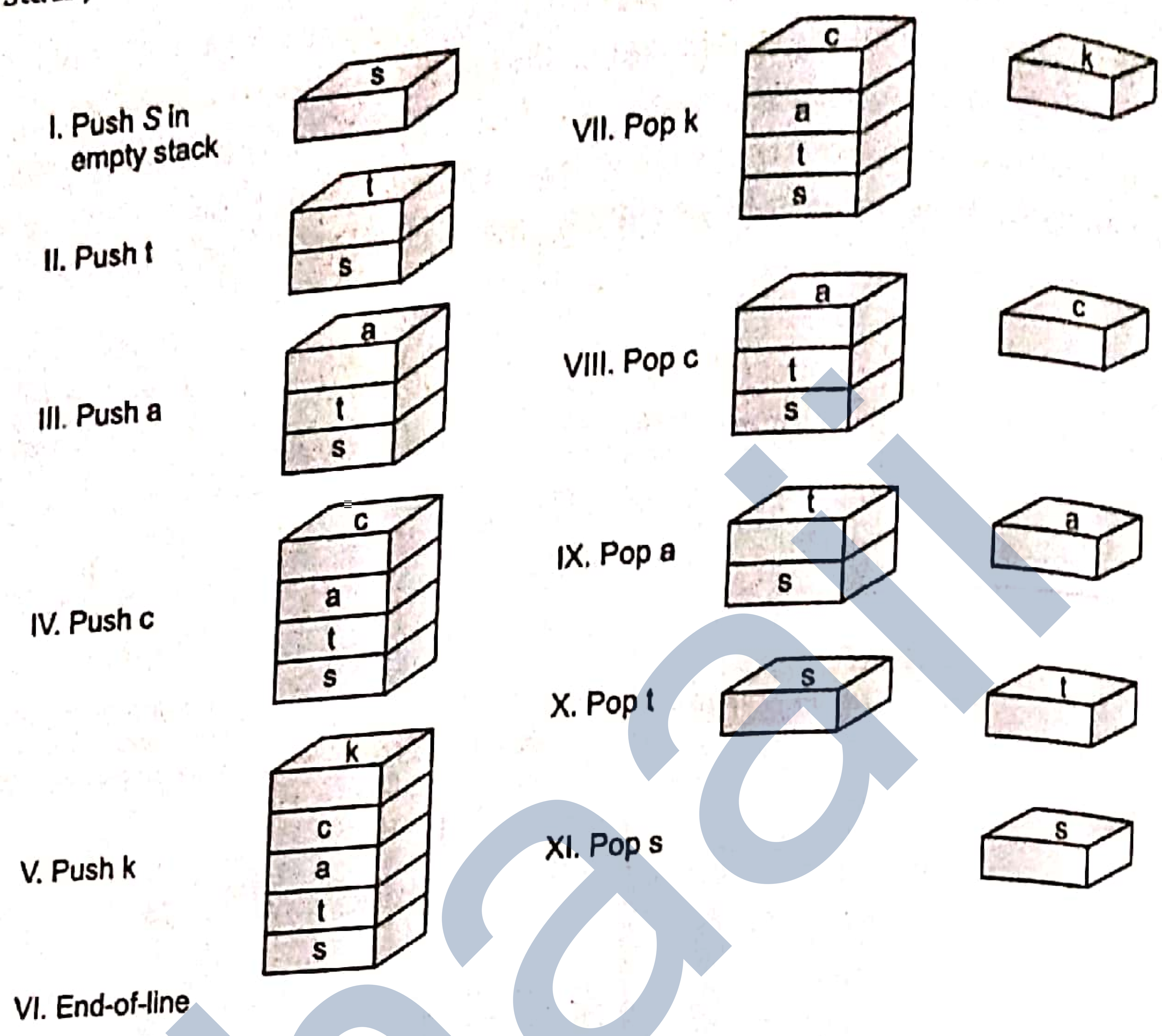


Figure 10.3 Reversal of a line using stack.

10.2.2B Polish Strings

Another application of stacks is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only *e.g.*, $A + B$, $C \times D$, D/A etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands *e.g.*, $(A + B) \times C\ (D/(J + D))$. These complex arithmetic operations can be converted into *polish strings* using stacks which then can be executed in *two operands and a operator* form.

Polish string, named after a polish mathematician, *Jan Lukasiewicz*, refers to the notation in which the operator symbol is placed either before its operands (*prefix notation*) or after its operands (*postfix notation*) in contrast to usual form where operator is placed in between the operands (*infix notation*).

Following table shows the three *types* of notations :

Table 10.1 *Expressions in infix, prefix, postfix notations*

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

Conversion of Infix Expression to Postfix (Suffix) Expression

While evaluating an infix expression, there is an evaluation order according to which

I Brackets or Parenthesis,

II Exponentiation,

III Multiplication or Division,

IV Addition or Subtraction

take place in the above specified order. The operators with the same priority (*e.g.*, × and/) are evaluated from left to right.

To convert an infix expression into a postfix expression, this evaluation order is taken into consideration.

An infix expression may be converted into postfix form either manually or using a stack. The manual conversion requires two passes : one for inserting braces and another for conversion. However, the conversion through stack requires single pass only.

The steps to convert an infix expression into a postfix expression manually are given below :

(*i*) Determine the actual evaluation order by inserting braces.

(*ii*) Convert the expression in the innermost braces into postfix notation by putting *the operator after the operands.*

(*iii*) Repeat step (*ii*) until entire expression is converted into postfix notation.

EXAMPLE 10.2 *Convert* $(A+B)\times C/D$ *into postfix notation.*

Solution.

Step I : Determine the actual evaluation order by putting braces

$$=((A+B)\times C)/D$$

Step II : Converting expressions into innermost braces

$$=((AB+)\times C)/D=(AB+C\times)/D=AB+C\times D/$$

EXAMPLE 10.3 *Convert* $((A+B)*C/D+E\wedge F)/G$ *into postfix notation.*

Solution. The evaluation order of given expression will be

$$=((((A+B)*C)/D)+(E\wedge F))/G$$

Converting expressions in the braces, we get

$$=((((AB+)*C)/D)+(EF\wedge))/G$$

$$=(((AB+C*)/D)+EF\wedge)/G$$

$$=((AB+C*D/)+EF\wedge)/G=(AB+C*D/EF\wedge+)/G$$

$$=AB+C*D/EF\wedge\ +G/$$

EXAMPLE 10.4 *Give postfix form of the following expression*

$$A*(B+(C+D)*(E+F)/G)*H$$

Solution. Evaluation order is

$$(A*(B+((C+D)*(E+F))/G))*H$$

Converting expressions in the braces, we get

$$=(A*(B+[(CD+)*(EF+)]/G))*H = A*(B+(CD+EF+*)/G)*H$$
$$= A*(B+(CD+EF+*G/))*H = (A*(BCD+EF+*G/+))*H$$
$$=(ABCD+EF+*G/+*)*H = \mathbf{ABCD+EF+*G/+*H*}$$

Example 10.5 *Give postfix form for $A+[(B+C)+(D+E)*F]/G$.*

Solution. Evaluation order is : $A+[\{(B+C)+((D+E)*F)\}/G]$

Converting expressions in braces, we get

$$= A+[\{(BC+)+(DE+)*F\}/G] = A+[\{(BC+)+(DE+F*)\}/G]$$
$$= A+[\{BC+DE+F*+\}/G] = A+[BC+DE+F*+G/]$$
$$= ABC+DE+F*+G/+$$

Example 10.6 *Give postfix form of expression for the following : NOT A OR NOT B NOT C*

Solution. The order of evaluation will be

((NOT *A*) OR ((NOT *B*) AND (NOT *C*)))

(As priority order is NOT, AND, OR)

=((*A* NOT) OR ((*B* NOT) AND (*C* NOT)))

=((*A* NOT) OR ((*B* NOT *C* NOT AND))

= *A* NOT *B* NOT *C* NOT AND OR

While converting from *infix to prefix form*, operators are put before the operands. Rest of the conversion procedure is similar to that of *infix to postfix* conversion.

Algorithm to Convert Infix Expression to Postfix Form

The following algorithm transforms the infix expression *X* into its equivalent postfix expression *Y*. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression *Y* will be constructed from left to right using the operands from *X* 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 *X*. The algorithm is completed when STACK is empty.

Algorithm ***Infix to Postfix Conversion using Stack***

Suppose X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto STACK, and add ")" to the end of X.
2. Scan X from left to right and REPEAT Steps 3 to 6 for each element of X UNTIL the STACK is empty :
3. If an operand is encountered, add it to Y.
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 Y 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 structure*'''

6. If a right parenthesis is encountered, then :

(a) Repeatedly pop from STACK and add to Y 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 Y].
'''*End of If structure*'''
'''*End of Step 2 Loop*'''

7. END.

EXAMPLE 10.7 *Convert X: A+(B*C–(D/E^F)*G)*H into postfix form showing stack status after every step in tabular form.*

Solution.

Symbol Scanned		Stack	Expression Y
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 * +

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence). Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Evaluation of a postfix Expression using stacks

As postfix expression is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle. Evaluation rule of a postfix expression states :

- While reading the expression from left to right, push the element in the stack if it is an operand ;
- pop the two operands from the stack, if the element is a binary operator. In case of NOT operator, pop one operand from the stack and then evaluate it (two operands and an operator).
- Push back the result of the evaluation. Repeat it till the end of the expression.

For a binary operator, two operands one popped from stack and for a unary operator, one operand is popped. Then, the result is calculated using operand(s) and the operator, and pushed back into the stack.

Algorithm *Evaluation of Postfix Expression*

```
'''Reading of expression takes place from left to right'''
1.  Read the next element      '''first element for the first time'''
2.  If element is operand then
        Push the element in the stack
3.  If element is operator then
    {
4.      Pop two operands from the stack
          '''POP one operand in case of unary operator'''
5.      Evaluate the expression formed by the two operands and the operator
6.      Push the result of the expression in the stack end
    }
7.  If no-more-elements then
        POP the result
    else
        go to step 1.
8.  END.
```

EXAMPLE 10.8 *Evaluate the postfix expression* $AB + C \times D /$ *if* $A = 2$, $B = 3$, $C = 4$ *and* $D = 5$.

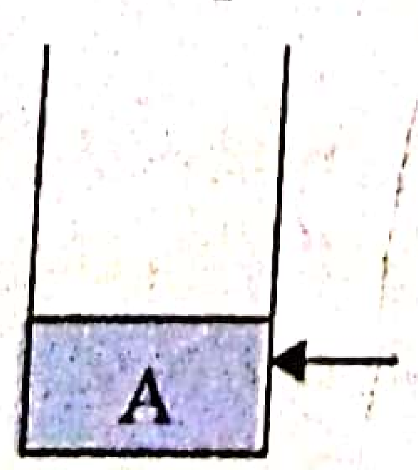
Solution. The expression given is $AB + C \times D /$

Starting from left to right

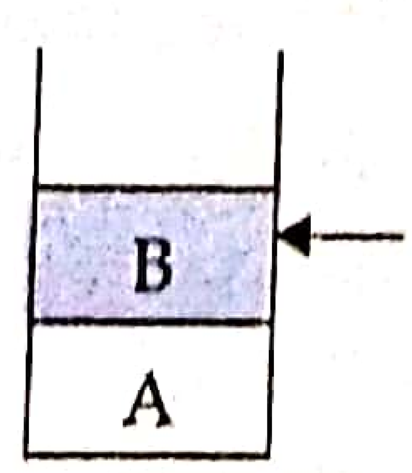
I First element is operand A, push A into the stack, (see *Stack I*)

II Second element is also operand B, push B also into the stack, (see *Stack II*)

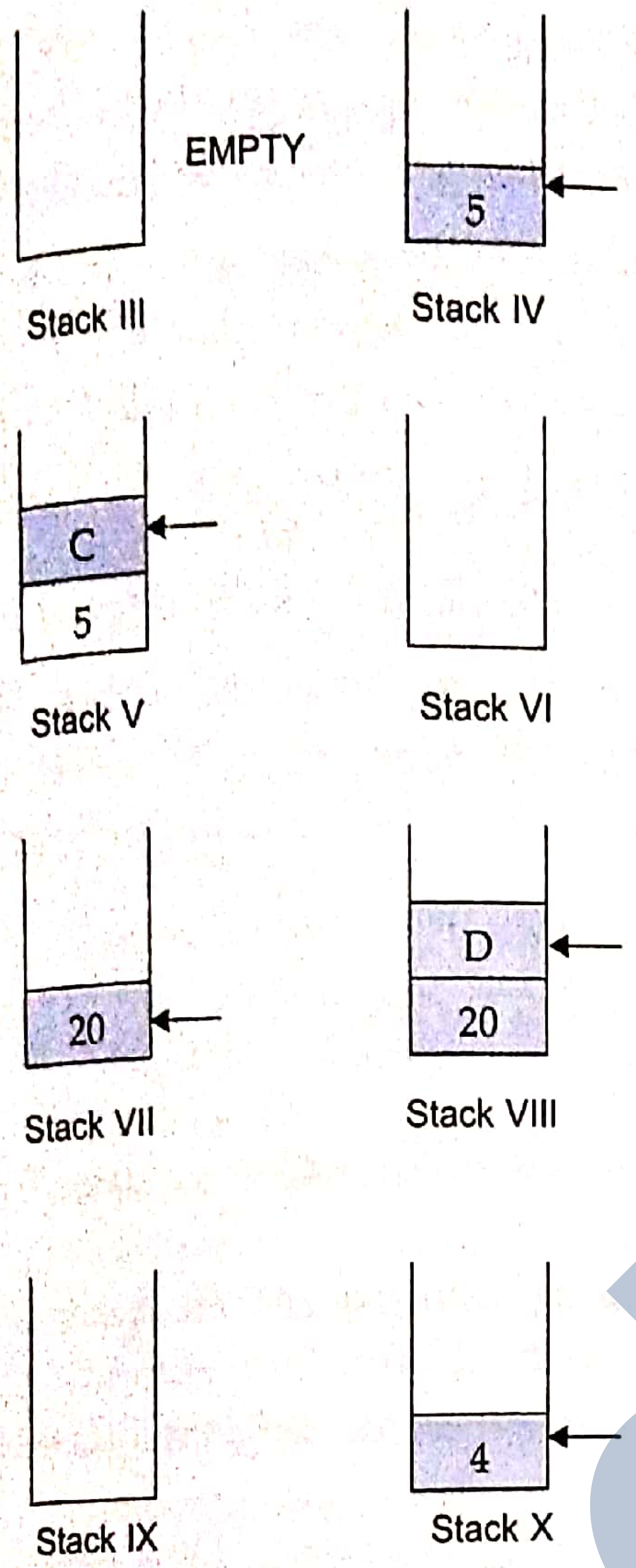
(← signifies top)



Stack I



Stack II



III Third element '+' is an operand, pop 2 operands from the stack *i.e.*, *A* and *B* (see *Stack III*) and evaluate the expression, *i.e.*,

$$A + B = 2 + 3 = 5$$

IV Push the result (5) into the stack, (see *Stack IV*)

V Next element C is operand ; pushed into the stack, (see *Stack V*)

VI Next X is an operator, pop 2 operands from the stack *i.e.*, 5 and C (see *Stack VI*) and evaluate :

$$5 \times C = 5 \times 4 = 20$$

VII Push the result (20) into the stack, (*Stack VII*)

VIII Next '*D*' is operand ; pushed into the stack, (*Stack VIII*)

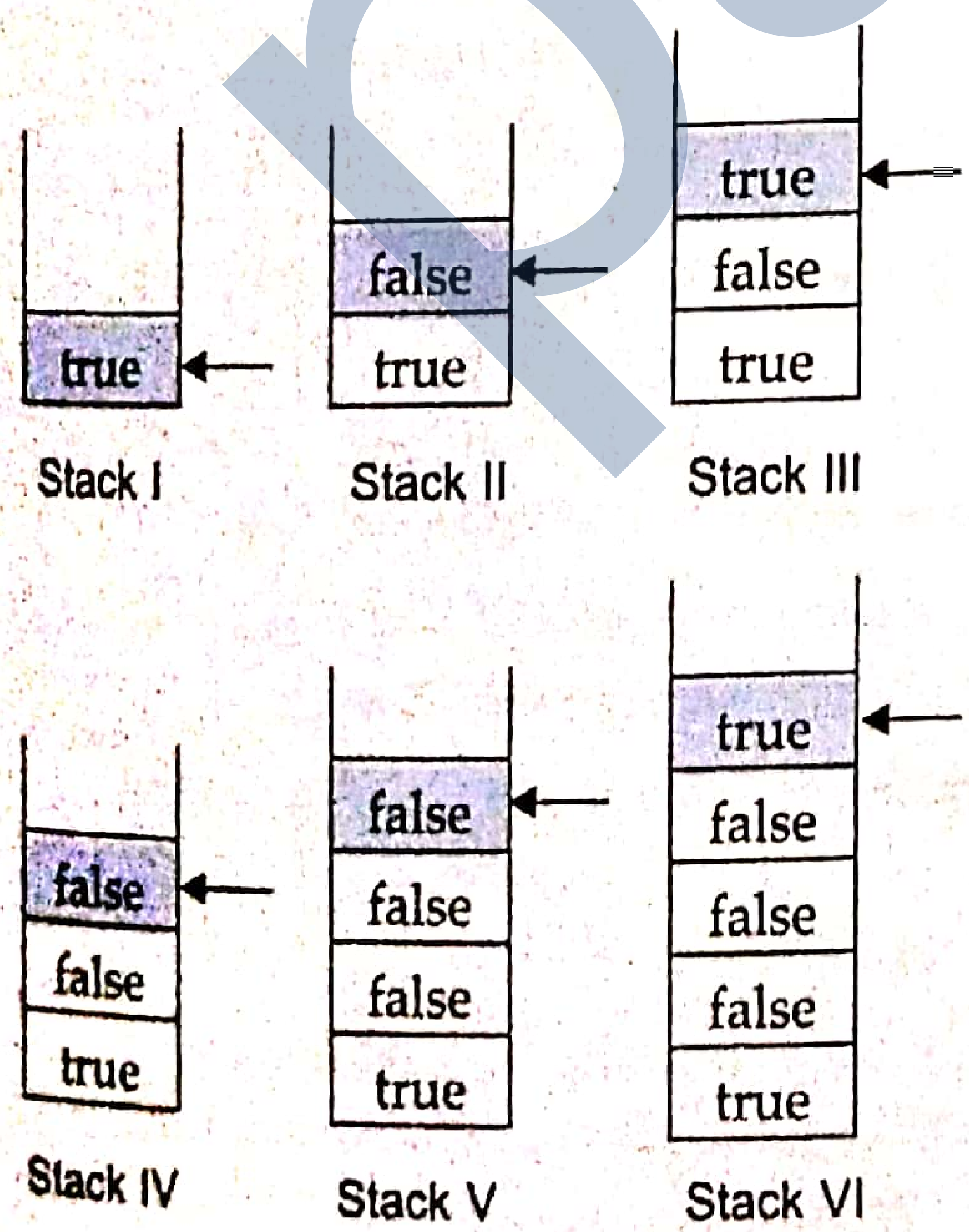
IX Next '/' is operator ; 2 operands popped *i.e.*, *D* and 20 ; (*Stack IX*)

Expression evaluated as follows : $\frac{20}{D} = \frac{20}{5} = 4$, pushed back. (*Stack X*)

X End of the expression, Pop from stack. Thus **the result is 4.**

EXAMPLE 10.9 *Evaluate the following expression in postfix form using a stack and show the contents of the stack after execution of each operation : true false true NOT false true OR NOT AND OR AND.*

Solution. Reading from left to right



I *'true'*, is operand ; pushed into stack (*Stack I*)

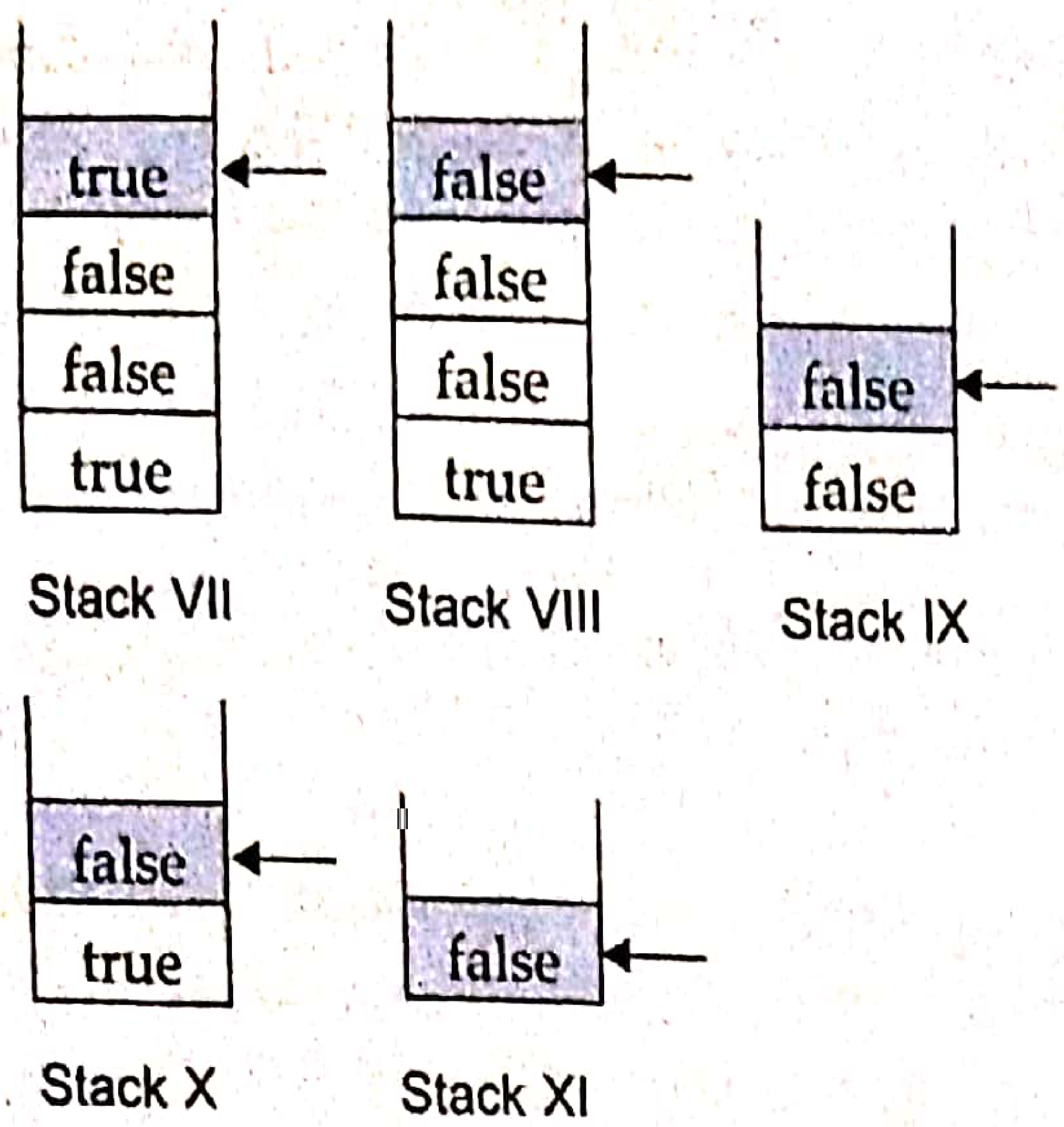
II *'false'* is operand ; pushed into stack (*Stack II*)

III 'true' is operand ; pushed into stack (*Stack III*)

IV 'NOT' is a unary operator. Thus, one operand is popped *i.e.*, *true* ;
Evaluating NOT *true* = *false* ; Pushing the result *'false'* into the stack (*Stack IV*)

V *'false'* is operand ; pushed (*Stack V*)

VI *'true'* is operand, pushed (*Stack VI*)



VII 'OR' is operator, Pop two operands *i.e., true, false.* Evaluating the expression, we get : *false* OR *true* = *true* ; Push the result *'true'*, into the stack (*Stack VII*)

VIII 'NOT' is operator ; One operand is popped, *i.e., true.* Evaluating : NOT *true* = *false* Push the result *'false'*, into the stack (*Stack VIII*)

IX 'AND' is operator ; Pop two operands, *i.e., false, false.* Evaluating : *false* AND *false* = *false* ; Push the result *'false'*, into the stack (*Stack IX*)

X 'OR' is operator ; Pop two operands, *i.e., false, false.* Evaluating : *false* OR *false* = *false* ; Push the result *'false'* (*Stack X*)

XI 'AND' is operator ; Pop *two* operands *i.e., false, true.* Evaluate and Push back : *true* AND *false* = *false* (*Stack XI*)

XII No-more-elements Pop from stack, therefore, the **result is 'false'.**

Evaluation of Prefix expression using stacks

The prefix expression is also without parentheses. The prefix expressions are evaluated by the compiler by using *two* stacks :

- one stack (**Symbol Stack**) for holding the *symbols of the expression* (All the *operators* and *operands/values* of the expression are considered ***symbols***) and
- another stack (**Number Stack** or **Operand-Value Stack**) for holding the numbers or values (*i.e.*, the *operands*).

EXAMPLE 10.10 *Evaluate the expression* 5 6 2 + * 12 4 /- *in tabular form showing stack status after every step.*

Solution.

Step	**Input Symbol/Element**		**Stack**	**Intermediate Calculations Output**
1.	5	Push	5	
2.	6	Push	5, 6	
3.	2	Push	5, 6, **2**	
4.	+	Pop (2 elements) & evaluate	5	6 + 2 = 8
5.		Push result (8)	5, **8**	
6.	*	Pop (2 elements) & evaluate	# empty	5 × 8 = 40
7.		Push result (40)	**40**	
8.	12	Push	40, **12**	
9.	4	Push	40, 12, **4**	
10.	/	Pop (2 elements) & evaluate	**40**	12/4 = 3
11.		Push result (3)	40, **3**	
12.	–	Pop (2 elements) & evaluate	#	40 – 3 = 37
13.		Push result (37)	**37**	
14.		No-more-elements		**37** (result)

(top most element is shown in colour)

Check Point 10.1

1. What are the technical names of insertion and deletion in a stack ?
2. What is the situation called, when an insertion/push is attempted in a full stack ?
3. What is the situation called, when a deletion/pop is attempted in an empty stack ?
4. Give some examples of stack applications.
5. What are infix expressions ? What are postfix expressions ?
6. Write equivalent postfix expressions for the following infix expressions :
   (*a*) a + b * d (*b*) p/(q – r)
   (*c*) i**2 + 3
7. Evaluate following postfix expression using a stack : 28, 8, 4, /, +.

WORKING WITH STACKS

PriP — Progress In Python 10.1

This PriP session is dedicated to the practice of data structure Stack's concepts and provides practical assignment for the same.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 10.1 under Chapter 10 after practically doing it on the computer.

>>>❖<<<

10.3 QUEUES

Queues are similar to stacks in that a queue also consists of a sequence of items (a linear list), and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the **front-end** and the **back-end** or **rear-end** of the queue. Items are always added to the queue at the ***rear-end*** and removed from the queue at the *front-end*. The operations of adding and removing items are called **enqueue** and **dequeue**. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed.

> **QUEUE**
>
> A Queue is a linear list implemented in FIFO – First In First Out manner where insertions take place at the rear-end and deletions are restricted to occur only at front end of the queue.

A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue. Thus a queue is also called a **FIFO list** *i.e.*, **First In First Out** list where the item first inserted is the first to get removed. So, we can say that a queue is a list of data that follows these rules :

1. Data can only be removed from the front end, *i.e.*, the element at the *front-end of the queue*. The removal of element from a queue is technically called **DEQUEUE operation.**
2. A new data element can only be added to the *rear of the queue*. The insertion of element in a stack is technically called **ENQUEUE operation**.

> **NOTE**
>
> The Enqueue operation adds item at the back of the queue (at rear-end) and the Dequeue operation removes the item from the front of the queue (at front-end) and returns it.

Other Queue Terms

There are *some* other terms related to queues, such as *Overflow* and *Underflow*.

Peek Refers to inspecting the value at the *queue's front* without removing it. It is also sometimes referred as **inspection**.

Overflow Refers to situation (ERROR) when one tries to enqueue an item in a queue that is full. This situation occurs when the size of the queue is fixed and cannot grow further or there is no memory left to accommodate new item.

Underflow Refers to situation (ERROR) when one tries to Dequeue/delete an item from an empty queue. That is, queue is currently having no item and still one tries to dequeue an item.

EXAMPLE 10.12 *Given a Bounded Queue of capacity 4 which is initially empty, draw pictures of the queue after each of the following steps :*

(i) Queue empty (ii) enqueue 'a' (iii) enqueue 'b'
(iv) enqueue 'c' (v) dequeue (vi) enqueue 'd'
(vii) enqueue 'e' (viii) dequeue (ix) dequeue
(x) dequeue (xi) dequeue

Solution.

Front = ↑ f ; Rear = ↑ r

(i) Queue is empty **(front = rear = None)**

0	1	2	3

↑f ↑r

(ii) enqueue 'a' **(front = 0, rear = 0)**

0	1	2	3
a			

↑f ↑r

(iii) enqueue 'b' **(front = 0, rear = 1)**

0	1	2	3
a	b		

↑f ↑r

(iv) enqueue 'c' **(front = 0, rear = 2)**

0	1	2	3
a	b	c	

↑f ↑r

(v) dequeue **(front = 1, rear = 2)**

0	1	2	3
	b	c	

a

↑f ↑r

(vi) enqueue 'd' **(front = 1, rear = 3)**

0	1	2	3
	b	c	d

↑f ↑r

(vii) enqueue 'e' **(front = 1, rear = 3)**

0	1	2	3
	b	c	d

OVERFLOW

↑f ↑r

[OVERFLOW because the queue is bounded, it cannot grow. If it could grow, then there would have been no OVERFLOW until no memory is left. In Python, (for queues implemented through lists) since Lists can grow, OVERFLOW condition does not arise until all the memory is exhausted.]

(viii) dequeue **(front = 2, rear = 3)**

0	1	2	3
		c	d

b

↑f ↑r

(ix) dequeue **(front = 3, rear = 3)**

0	1	2	3
			d

c

↑f ↑r

(x) dequeue **(front = None, rear = None)**

0	1	2	3

d

↑f ↑r

(xi) dequeue **(front = None, rear = None)**

0	1	2	3

UNDERFLOW

↑f ↑r

10.3.1 Implementing Queues in Python

In Python, you can use ***Lists*** to implement queues. Python offers us a convenient set of methods to operate lists as queues. For various Queue operations, we can use a list say ***Queue*** and use Python code as described below :

Peek We can use : `<Queue>[front]`

where **<Queue>** is a list ; ***front*** is an integer storing the position of first value in the queue.

Enqueue We can use : `<Queue>.append(<item>)`

where **<item>** is the item being pushed in the *Queue*. The item will be added at the rear-end of the queue.

Dequeue We can use : `<Queue>.pop(0)`

it removes the first value from the *Queue* (*i.e.*, the item at the *front-end*) and returns it.

Number of Elements in Queue

We can determine the size of a queue using the formula:

`Number of elements in queue = rear – front + 1`

In Python queues, implemented through lists, the *front* and *rear* are :

`front = 0 and rear = len(<queue>) - 1`

You can also use Python function **len(<queue>)** to get the size of the queue.

Let us now implement a Queue of numbers through a program.

Program 10.2 Program to implement Queue Operations

```
############# queue IMPLEMENTATION ##############
"""
queue: implemented as a list
front : integer having position of first (frontmost) element in queue
rear : integer having position of last element in queue
"""
def cls():
    print("\n" * 100)

def isEmpty( Qu ) :
    if Qu == [] :
        return True
    else :
        return False

def Enqueue(Qu, item) :
    Qu.append(item)
    if len(Qu) == 1 :
        front = rear = 0
    else :
        rear = len(Qu) - 1
```

```
def Dequeue(Qu) :
    if isEmpty(Qu) :
        return "Underflow"
    else :
        item = Qu.pop(0)
    if len(Qu) == 0 :                  #if it was single-element queue
        front = rear = None
    return item

def Peek(Qu) :
    if isEmpty(Qu) :
        return "Underflow"
    else :
        front = 0
    return Qu[front]

def Display(Qu) :
    if isEmpty(Qu) :
        print("Queue Empty!")
    elif len(Qu) == 1:
        print(Qu[0], "<== front, rear")
    else :
        front = 0
        rear = len(Qu) - 1
        print(Qu[front], "<- front")
        for a in range(1, rear ) :
            print(Qu[a])
        print(Qu[rear], "<- rear")

# __main__ program
queue = []                   # initially queue is empty
front = None
while True :
    cls()
    print("QUEUE OPERATIONS")
    print("1. Enqueue")
    print("2. Dequeue")
    print("3. Peek")
    print("4. Display queue")
    print("5. Exit")
    ch = int(input("Enter your choice ( 1-5) : "))
    if ch == 1 :
        item = int(input("Enter item :"))
        Enqueue(queue, item)
        input("Press Enter to continue...")
    elif ch == 2 :
        item = Dequeue(queue)
        if item == "Underflow" :
            print("Underflow! Queue is empty!")
        else :
            print("Dequeue-ed item is", item)
        input("Press Enter to continue...")
```

```
        elif ch == 3 :
            item = Peek(queue)
            if item == "Underflow" :
                print("Queue is empty!")
            else :
                print("Frontmost item is", item)
            input("Press Enter to continue...")
        elif ch == 4 :
            Display(queue)
            input("Press Enter to continue...")
        elif ch == 5 :
            break
        else :
            print("Invalid choice!")
            input("Press Enter to continue...")
```

```
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :5
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 4
5 <== front, rear
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :7
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 3
Frontmost item is  5
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 1
Enter item :9
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 4
5 <- front
7
9 <- rear
Press Enter to continue...
__________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  5
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  7
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Dequeue-ed item is  9
Press Enter to continue...
_______________________
QUEUE OPERATIONS
1. Enqueue
2. Dequeue
3. Peek
4. Display queue
5. Exit
Enter your choice (1-5) : 2
Underflow! Queue is empty!
Press Enter to continue...
```

Queue Item-nodes

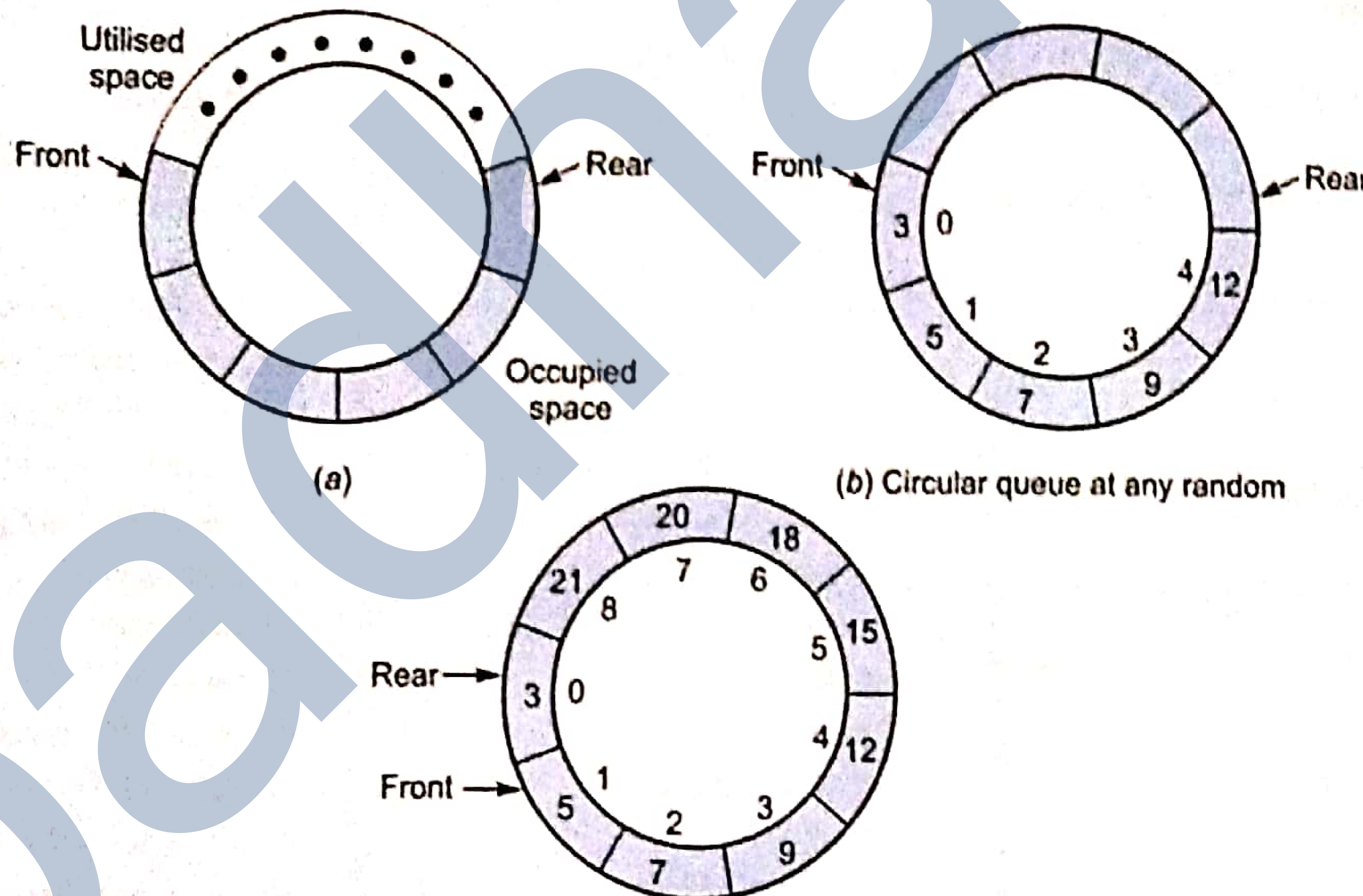
Like stacks, queues can also store information of various types in its item-nodes and if information is logically related then item-node is first create as a list (or any other sequence as per your requirements) and then inserted in the queue. Solved problem 21 implements such a queue.

10.3.2 Variations in Queues

Queues can be used in several forms and ways, depending upon the requirements of the program. Two popular variations of queues are **Circular Queues** and **Deque** (Double-Ended queues).

10.3.2A Circular Queues

Circular Queues are the queues implemented in circular form rather than a straight line. These are used in programming languages that allow the use of fixed-size linear structures (such as arrays of C/C++ etc.) as queues. In such queues, after some insertions and deletions, some unutilized space lies in the beginning of the queue. To overcome such problems, circular queues are used that overcome the problem of unutilized spaces in fixed-size linear queues. Figure 10.4 shows a circular queue.



(*a*) (*b*) Circular queue at any random

(*c*) Circular queue (full to the capacity) at any random time

Figure 10.4

Python lists are dynamic structures that can grow and shrink when needed, thus, you don't need circular queues generally when you are implementing queues through Python lists.

10.3.2B Deque (Double-ended Queues)

Deques (double-ended queues) are the refined queues in which elements can be added or removed at either end but not in the middle. There are *two* variations of a deque – an **input restricted deque** and an **output restricted deque**.

- **An Input Restricted Deque** is a deque which allows insertions at only one end but allows deletions at both ends of the list. [see Fig 10.5(*a*)]
- **An Output Restricted Deque** is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list. [see Fig. 10.5(*b*)]

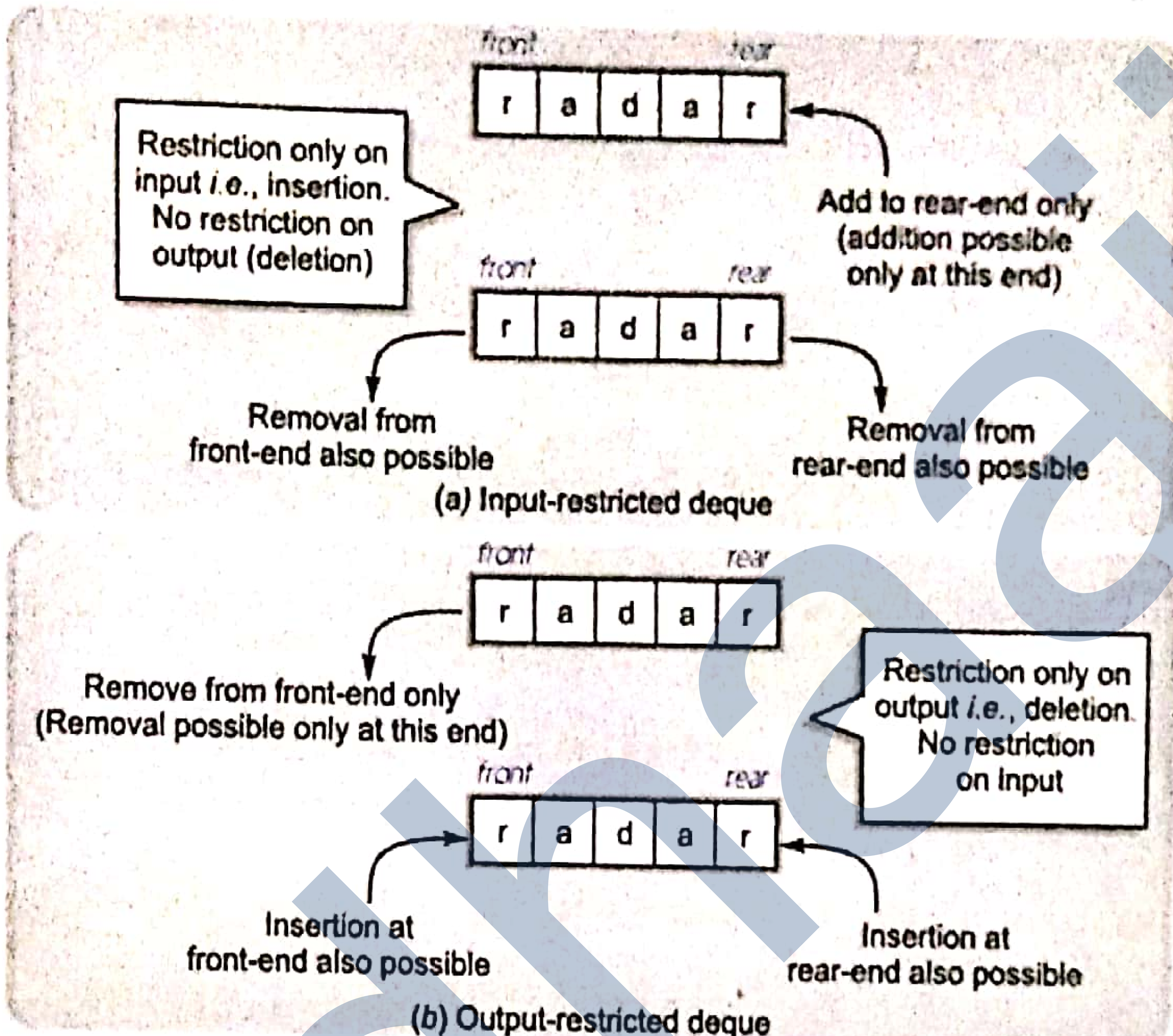


Figure 10.5 Dequeues

10.3.3 Queue Applications

Queue data structure has found application in many diverse areas. We are discussing here only a few of them, for understanding purposes.

Direct Applications

- *Any flow* of ***first-in-first-out*** or ***first-come-first-served*** may be implemented as a queue. For example, the passengers leave a bus as the order (sequence) they hopped on. (*first-in-first-out, last-in-last-out*), or a single-lane vehicles into a tunnel, and come out from the other end – all these are real-life applications of queues.
- When a resource is shared among multiple consumers, queues are the best way to implement this type of sharing. For example, in a computer lab, there can be printer being shared among a number of computers. All the computers can send their printing requests to the shared printer. For all the printing requests, printer will maintain a queue and will print on FIFO basis.
- When you make calls to a call center, the call center phone systems will use a queue to hold people in line until a service representative is free.
- Airport authorities make use of queues in situation of sharing a single runway of airport for both landing and take-off of flights.

Check Point 10.2

1. What are the technical names of insertion and deletion in a queue?
2. What is the situation called, when an insertion/push is attempted in a full queue ?
3. What is the situation called, when a deletion/pop is attempted in an empty queue ?
4. Give some examples of queue applications.
5. What is circular queue ?
6. What is a deque ? What are its types ?

- When you run multiple programs/applications on one CPU, and the CPU has to treat each application equally, then CPU might use queues to process these applications in phased-manner[2]. (Job scheduling)

Indirect Applications

- Queues have found indirect usage in many computer applications as auxiliary data structures and components of other data structures.[3]

LET US REVISE

- *A stack is a linear structure implemented in LIFO (Last In First Out) manner where insertions and deletions take place only at one end i.e., the stack's top.*
- *An insertion in a stack is called pushing and a 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 an OVERFLOW.*
- *When a stack is empty and an element's deletion is tried from the stack, it is called an UNDERFLOW.*
- *Polish string refers to the notation in which the operator symbol is placed either before its operands (this form is called prefix notation) or after its operands (this form is called postfix notation). The usual form, in which the operator is placed in between the operands, is called infix notation.*
- *A queue is a linear structure implemented in FIFO (First In First Out) manner where insertions can occur only at the " rear" end and deletions can occur only at the " front" end.*
- *Circular Queues are the queues implemented in circle form rather than a straight line.*
- *Dequeues (double-ended queues) are the refined queues in which elements can be added or removed at either end but not in the middle.*
- *An input restricted deque is a deque which allows insertions at only one end but allows deletions at both ends of the list.*
- *An output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.*

WORKING WITH QUEUES

PriP — Progress In Python 10.2

This PriP session is dedicated to the practice of data structure Stack's concepts and provides practical assignment for the same.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 10.2 under Chapter 10 after practically doing it on the computer.

2. This is known as **Round-Robin Scheduling.**
3. Queues also play a very essential role in graph algorithms like *BFS* and *DFS* but these are way ahead of the scope of this book.

Solved Problems

1. *What is a Data-Structure ?*

Solution. A data-structure is a logical way for organizing data in memory that considers not only the items stored but also the relationship among the items so as to give efficient and optimal performance. In other words, it is a way of organizing data such that it can be used efficiently.

2. *What is a stack ? What basic operations can be performed on them ?*

Solution. Stack is a basic data-structure where insertion and deletion of data takes place at one end called *the top of the stack i.e.,* it follows the **Last In-First Out(LIFO)** principle.

In Python, a stack is generally implemented with lists.

Following basic operations can be performed on stacks:

(*a*) ***Push*** *i.e.,* Insertion of element in the stack
(*b*) ***Pop*** *i.e.,* Deletion of an element from the stack
(*c*) ***Peek*** *i.e.,* viewing topmost element without removing it
(*d*) ***Display*** or view all the elements in the stack.

3. *Enlist some applications of stacks.*

Solution. Because of LIFO property of stacks, these are used in various applications like :

- reversal of a sequence,
- Infix to Postfix conversion,
- Postfix and prefix expression evaluation,

and many more.

4. *What are queues? What all operations can be performed on queues?*

Solution. Queues are the data structures where data is entered into the queue at one end – *the rear end* and deleted from the other end – *the front end, i.e.,* these follow **First In-First Out (FIFO)** principle. Following basic operations can be performed on queues :

(*i*) ***Enqueue*** *i.e.,* Insertion of an element in the queue
(*ii*) ***Dequeue*** *i.e.,* Deletion of an element from the queue
(*iii*) ***Peek*** *i.e.,* viewing *frontmost* element without removing it
(*iv*) ***Display*** or view all the elements in the queue.

5. *Enlist some applications of queues.*

Solution. Applications of queues include the situations where FIFO property is exploited. Some common applications of queues include :

(*i*) Sharing of one resource among multiple users or seekers such as shared printer among multiple computers; Call center executive's response among waiting callers etc.
(*ii*) Airport authorities make use of queues in situation of sharing a single runway of airport for both landing and take-off of flights.
(*iii*) CPU uses queues to implement round-robin scheduling among waiting processes.
(*iv*) Queues are used in many computer algorithms also.

6. *Write the equivalent infix expression for :* 10, 3, *, 7, 1, –, *, 23, +

Solution. $10 * 3 * (7-1) + 23$

7. *Consider the following stack of characters implemented as an array of 8 elements :*

STACK : A, J, P, N
↑ top

Describe the stack as the following operations take place :

(a) POP (STACK, ITEM) (b) PUSH (STACK, K) (c) PUSH (STACK, S)
(d) POP (STACK, ITEM) (e) PUSH (STACK, G)

Solution.

(a) **STACK** : *A, J, P* (top) as *N* is Popped

(b) **STACK** : *A, J, P, K* (top) as *K* is Pushed

(c) **STACK** : *A, J, P, K, S* (top) as *S* is Pushed

(d) **STACK** : *A, J, P, K* (top) as *S* is Popped

(e) **STACK** : *A, J, P, K, G* (top) as *G* is Pushed

8. *Given a stack as an array of 7 elements STACK : K, P, S, -, -, -,*

(a) *When will overflow and underflow occur ?*

(b) *Can K be deleted before S ? Why ?*

Solution.

(a) *Overflow* will occur when stack will be having 7 elements and there would be no space to insert a new element.

Underflow will occur when all the elements will have been deleted from the stack and no more element could be deleted.

(b) *K* can not be deleted before *S*. As S has been inserted after *K* and stack follows the LIFO rule i.e., *Last In First Out*, therefore *S* will be deleted before *K*.

9. *Consider the following infix expression* $P = -a + b * c \uparrow (d * e) / f$ *where* ↑ *denotes exponentiation, and / integer division. Translate P into its equivalent postfix expression using the following set of priorities :*

(	0
+ –)	1
unary – × / ÷	2
↑	3

Solution. $P = -a + b * c \uparrow (d * e) / f$

The order of evaluation will be (according to given priorities) as given below

$$= ((-a) + ((b * (c \uparrow (d * e))) / f)) = (a - + ((b * (c \uparrow (de*))) / f))$$

$$=((a-)+((b*(cde*\uparrow))/f))$$

$$=((a-)+((bcde*\uparrow*/f))=(a-)+((bcde*\uparrow*f/))$$

$$=(a-)+(bcde*\uparrow*f/)=a-bcde*\uparrow*f/+$$

10. *Give the algorithm for converting an infix arithmetic expression into a postfix arithmetic expression. Use the algorithm for the following expression, showing in a tabular form the changing status of the stack :*

Q : (A – B) * (C / D) + E

Solution. The simulation of this algorithm for the above expression is shown in the following table :

*Conversion of (A – B) * (C / D) + E into postfix expression*

Step	Input element / symbol taken	Action	Stack status	Output to be printed
			# (empty)	
1.	'('	PUSH	'('	
2.	'A'	Print	'('	A
3.	'–'	PUSH	'('-	A
4.	B	Print	'(' -	A
5.	')'	POP & Print POP	'(' # #	AB- AB-
6.	*	PUSH	#	AB-
7.	'('	PUSH	* '('	AB-
8.	C	Print	*'('	AB-C
9.	/	PUSH	*'('/	AB-C
10.	D	Print	*'('/	AB-CD
11.	')'	POP & Print POP	*'(' *	AB-CD/ AB-CD/
12.	+	POP & Print PUSH	* + +	AB-CD/*
13.	E	Print	+	AB-CD/*E
14.	;	POP & Print STOP	# (empty)	AB-CD/*E+

The resultant postfix expression is AB – CD/ *E + .

11. *Evaluate following arithmetic expression A which is in postfix notation. Show the contents of the stack during the execution of the algorithm using the following :*

A = 30, 5, 2, ^ 12, 6, /, +, – .

Solution. *Push '(' to the stack and add ')' at the end of expression i.e.,* the expression becomes

30, 5, 2, **, 12, 6, /, +, –)

Step	Input Element/Symbol	Action taken	Stack Status	Output
1.			(	
2.	30	Push	(, 30	
3.	5	Push	(, 30, 5	
4.	2	Push	(, 30, 5, 2	
5.	**	POP (2 elements)	(, 30	5 ** 2 =25
6.		Push the result (25)	(, 30, 25	
7.	12	Push	(, 30, 25, 12	
8.	6	Push	(, 30, 25, 12, 6	
9.	/	POP (2 elements)	(, 30, 25	12/6 = 2
10.		Push result (2)	(, 30, 25, 2	
11.	+	POP (2 elements)	(, 30	25 + 2 =27
12.		Push the result (27)	(, 30, 27	
13.	–	POP (2 elements)	(	30 – 27 =3
14.		Push the result (3)	(3	
15.	)	POP everything	#	

Result = 3.

12. *Convert the expression (TRUE and FALSE) or not (FALSE or TRUE) to postfix expression. Show the contents of the stack at every step.*

Solution. (TRUE and FALSE) or not (FALSE or TRUE)]

Adding] to the end of expression and inserting [to the beginning of stack.

Scanning from Left to Right

S.No	Symbol	Stack	Postfix Expression Y
0.		[	
1.	(	[(	
2.	TRUE	—	TRUE
3.	and	[(and	—
4.	FALSE	—	TRUE FALSE
5.	)	[	TRUE FALSE and
6.	or	[or	—
7.	not	[or not	—
8.	(	[or not (	—
9.	FALSE	—	TRUE FALSE and FALSE
10.	or	[or not (or	—
11.	TRUE	—	TRUE FALSE and FALSE TRUE
12.	)	[or not	TRUE FALSE and FALSE TRUE or
13.	]	End of Expression	TRUE FALSE and FALSE TRUE or not or

13. *Write the equivalent infix expression for given postfix expression : a, b, AND, a, c, AND, OR.*

Solution. a AND b OR a AND c.

14. *Evaluate the following postfix notation of expression (Show status of stack after execution of each operation) :*

4, 10, 5, +, *, 15, 3, /, –

Solution. Adding] in the end of given postfix expression and pushing [in the stack.

4, 10, 5, +, *, 15, 3, 1, –]

S. No.	Symbol	Action taken	Stack	Intermediate output
0			[	
1	4	Operand, Push	[4 ↑ (↑– top)	
2	10	Operand, Push	[4, 10 ↑	
3	5	Operand, Push	[4, 10, 5 ↑	
4	+	Operator, Pop twice	[4 ↑	10 + 5 = 15
		Evaluate and Push back	[4, 15 ↑	
5	*	Operator ; Pop twice	[	4 * 15 = 60
		Evaluate and push back	[60 ↑	
6	15	Operand ; Push	[60, 15 ↑	
7	3	Operand ; Push	[60, 15, 3 ↑	
8	/	Operator ; Pop twice	[60 ↑	15/3 = 5
		Evaluate and Push back	[60, 5 ↑	
9	–	Operator ; Pop twice	[	60 – 5 = 55
		Evaluate and Push back	[55 ↑	
10	]	Pop All	55 Ans.	

15. *Evaluate the following postfix notation of expression : 15 3 2 + / 7 + 2. **

Solution. Adding] in the end of expression and pushing [in the stack initially.

Symbol	Action	Stack	Intermediate Output
		[↑ signifies top ↑	
15	Operand : Push	[15 ↑	
3	Operand : Push	[15,3 ↑	

Symbol	Action	Stack	Intermediate Output
2	Operand : Push	[15, 3, 2 ↑	
+	Operator : Pop twice	[15 ↑	
	Calculate and push back the result	[15, 5 ↑	3 + 2 = 5
/	Operator : Pop twice	[↑	
	Calculate result and Push back the result	[3 ↑	15 / 5 = 3
7	Operand : Push	[3, 7 ↑	
+	Operator : Pop twice	[↑	
	Calculate and Push back the result	[10 ↑	3 + 7 = 10
2	Operand : Push	[10, 2 ↑	
*	Operator : Pop twice	[↑	
	Calculate and Push back the result	[20 ↑	10 * 2 = 20
]	End of expression : Pop All ;	# 20 : Answer	

16. *Evaluate the following postfix notation of expression : 50, 60, +, 20, 10, –, **

Solution.

Step	Input Symbol	Action Taken	Stack Status	Output
1.	50	Push	50 ↑ (↑– top)	
2.	60	Push	50, 60 ↑	
3.	+	Pop (2 elements)	↑ (*empty stack*)	50 + 60 = 110
		Push result (110)	110 ↑	
4.	20	Push	110, 20 ↑	
5.	10	Push	110, 20, 10 ↑	
6.	–	Pop (2 elements)	110 ↑	20 – 10 = 10
		Push result (10)	110, 10 ↑	
7.	*	Pop (2 elements)	↑ (*empty stack*)	110 * 10 = 1100
		Push result (1100)	1100 ↑	
8.	End of Expression Pop (result)		**Ans = 1100**	

17. *Evaluate the following postfix notation of expression :*
(Show status of Stack after each operation)

False, True, NOT, OR, True, False, AND, OR

Solution.

Step	Input Symbol	Action taken	Stack Status	Output
1.	False	Push	False	
2.	True	Push	False, True	
3.	NOT	Pop (1 *element*) Push result (*False*)	False, False	NOT True = False
4.	OR	Pop (2 *elements*) Push result (*False*)	False	False OR False = False
5.	True	Push	False, True	
6.	False	Push	False, True, False	
7.	AND	Pop (2 *elements*) Push result (*False*)	False, False	True AND False = False
8.	OR	Pop (2 *elements*) Push result (*False*)	False	False OR False = **False**
				Ans. = False

18. *Write a program to implement a stack for these book-details (bookno, book name). That is, now each item node of the stack contains two types of information – a bookno and its name. Just implement Push and display operations.*

Solution.

```
"""
    Stack: implemented as a list
    top : integer having position of topmost element in Stack
"""

def cls():
    print("\n" * 100)

def isEmpty( stk ) :
    if stk == [] :
        return True
    else :
        return False

def Push(stk, item) :
    stk.append(item)
    top = len(stk) - 1
```

```
def Display(stk) :
    if isEmpty(stk) :
        print("Stack empty")
    else :
        top = len(stk) - 1
        print(stk[top], "<- top")
        for a in range(top-1, -1, -1 ) :
            print(stk[a])

# __main__
Stack = []                         # initially stack is empty
top = None
while True :
    cls()
    print("STACK OPERATIONS")
    print("1. Push")
    print("2. Display stack")
    print("3. Exit")
    ch = int(input("Enter your choice (1-5) :"))
    if ch == 1 :
        bno = int(input("Enter Book no. to be inserted :"))
        bname = input("Enter Book name to be inserted :")
        item = [bno, bname]        # creating a list from the input items.
        Push(Stack, item)
        input()
    elif ch == 2 :
        Display(Stack)
        input()
    elif ch == 3 :
        break
    else :
        print("Invalid choice!")
        input()
```

19. *Write a program to perform insert and delete operations on a Queue containing Members details as given in the following definition of itemnode:*

MemberNo	integer
MemberName	String
Age	integer

Solution.

```
"""
queue: implemented as a list
front : integer having position of first (frontmost) element in queue
rear : integer having position of last element in queue
"""
def cls():
    print("\n" * 100)
```

```
def isEmpty( Qu ) :
    if Qu == [] :
        return True
    else :
        return False

def Enqueue(Qu, item) :
    Qu.append(item)
    if len(Qu) == 1 :
        front = rear  = 0
    else :
        rear = len(Qu) - 1

def Dequeue(Qu) :
    if isEmpty(Qu) :
        return "Underflow"
    else :
        item = Qu.pop(0)
        if len(Qu) == 0 :          # if it was single-element queue
            front = rear = None
        return item

def Display(Qu) :
    if isEmpty(Qu) :
        print("Queue Empty!")
    elif len(Qu) == 1:
        print(Qu[0], "<== front, rear")
    else :
        front = 0
        rear = len(Qu) - 1
        print(Qu[front], "<- front")
        for a in range(1, rear ) :
            print(Qu[a])
        print(Qu[rear], "<- rear")

# __main__
queue = []                         # initially queue is empty
front = None

while True :
    cls()
    print("QUEUE OPERATIONS")
    print("1. Enqueue)
    print("2. Dequeue")
    print("3. Display queue")
    print("4. Exit")
    ch = int(input("Enter your choice (1-5) : "))
```

```
        if ch == 1 :
            print("For the new member, enter details below:")
            memberNo = int(input("Enter member no :"))
            memberName = input("Enter member name :")
            age = int(input("Enter member's age :"))
            item = [memberNo, memberName, age]
            Enqueue(queue, item)
            input("Press Enter to continue...")
        elif ch == 2 :
            item = Dequeue(queue)
            if item == "Underflow" :
                print("Underflow! Queue is empty!")
            else :
                print("Dequeue-ed item is", item)
            input("Press Enter to continue...")
        elif ch == 3 :
            Display(queue)
            input("Press Enter to continue...")
        elif ch == 4 :
            break
        else :
            print("Invalid choice!")
            input("Press Enter to continue...")
```

20. *Describe the similarities and differences between queues and stacks.*

Solution.

Similarities :

1. Both queues and stacks are special cases of linear lists.
2. Both can be implemented as arrays or lists.

Differences :

1. A stack is a LIFO list, a queue is a FIFO list.
2. There are no variations of stack, a queue, however, may be circular or deque.

21. *What is the difference between an array and a stack housed in an array ? Why is stack called a LIFO data structure ? Explain how push and pop operations are implemented on a stack.*

Solution. An array is a group of homogeneous elements stored in contiguous memory locations. The elements in an array can be processed from anywhere in the array.

A stack implemented as an array also has elements stored in contiguous memory locations. But a stack is always processed in a LIFO manner *i.e., Last In First Out* manner wherein the elements can be added or removed from the top end of the stack. That is why a stack is also called a LIFO data structure.

An addition to the stack is known as PUSH. The new element is added at the top and the top (variable or pointer) is made to refer to the new element.

A removal of an element from a stack is known as POP. The elements (being pointed to by top) is removed and the top is made to point to the next element in the row.

22. *An algorithm requires two stacks of sizes M and N that are to be maintained in the memory. Illustrate with an example how will you adjust two stacks in one dimensional array with M + N memory locations so that the overflow condition is minimised.*

Solution. Let us say that stack *A* is with size *M* and stack *B* is with size *N*.

If the stack *A* is stored in locations 0 to $M-1$ and the stack *B* is stored in locations *M* to $M+N-1$, the separate areas for the two are ensured. In the beginning the *TOPs* of both stacks are at opposite ends. When TOP.A reaches at $M-1$, any further insertion will lead to overflow in stack *A* and when TOP.B reaches at *M*, any further insertion in stack *B* will lead to overflow.

23. *Propose a data structure which supports the stack Push and Pop operations and a third operation FindMin, which returns the smallest element in the data structure.*

Solution. Use two stacks say *S1* and *S2* – *S1* in to which the elements are pushed and *S2* in to which only the current minimum is pushed.

1. When first element is pushed in stack *S1*, push it to stack *S2* too, considering that this is the minimum element.
2. When one needs to insert an element, say *Ele* :
   (*a*) we first push *Ele* on to *S1* and
   (*b*) then access the top element, say *T* of S2 which is the minimum before the insertion of *Ele*.
   (*c*) Compare *Ele* with *T* ; if only *Ele* is less than *T*, we push *Ele* on to S2 . Hence the current minimum will always be on top of S2.
3. When one needs to pop an element ,
   (*a*) pop the top element of S1 and
   (*b*) if this element is also equal to the one on top of S2, then pop from S2 as well.

24. *Consider an empty stack of integers. Let the numbers 1, 2 , 3, 4, 5, 6 be pushed on to this stack only in the order they appeared from left to right. Let S indicates a push and X indicates a pop operation. Can they be outputted in to the order 325641 and order 154623 ?.*

(Hint. *SSSSSSXXXXXX outputs 654321*)

Solution. SSSXXSSXSXXX outputs 325641.

154623 cannot be output as 2 is pushed much before 3 so can appear only after 3 in output.

GLOSSARY

Circular Queue	Queues implemented in circle form rather than a straight line.
Dequeue	Double-ended queue where elements can be added or removed from either end.
Infix Expression	An expression where operators are placed in between the operands.
Popping	Deletion of an element in a stack.
Postfix expression	(Also called *reverse polish notation* of an expression). An expression where operators are placed after the operands.
Prefix expression	(Also called *polish notation* of an expression). It is an expression where operators are placed before the operands.
Pushing	Inserting a new element in a stack.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is a stack ?
2. What is a queue ?
3. Which data structure will you use for simulating a blanket donation camp ?
4. Differentiate between :
   (*i*) Linear Queue and Circular Queue
   (*ii*) A Queue and a Deque.
5. What are input/output restricted queues ?
6. What are enqueue and dequeue operations ?
7. What are push and pop operations ?
8. Give some examples of stack applications.
9. Give some examples of queue applications.
10. Can you suggest a real life application for input/output restricted queues ?

Type B : Application Based Questions

1. Translate following infix expression into its equivalent postfix expression :
   $((A - B) * (D / E)) / (F * G * H)$ [Ans. $A\ B -\ D\ E\ / *\ F\ G * H * /$]
2. Translate following infix expression into its equivalent postfix expression :
   $(A + B \uparrow D) / (E - F) + G$ [Ans. $A\ B\ D \uparrow +\ E\ F - / G +$]
3. Translate following infix expression into its equivalent postfix expression :
   $A * (B + D) / E - F - (G + H / K)$ [Ans. $A\ B\ D + * E / F - G\ H\ K / + -$]
4. Write the equivalent infix expression for
   10, 3, *, 7, 1, –, *, 23, + [Ans. 10 * 3 * (7 – 1) + 23]
5. Write the equivalent infix expression for *a*, *b*, AND, a, c, AND, OR. [Ans. a AND b OR a AND c]
6. Consider the arithmetic expression *P*, written in postfix notation :
   12, 7, 3, –, /, 2, 1, 5, +, *, +
   (*a*) Translate *P*, into its equivalent infix expression.
   (*b*) Evaluate the infix expression. [Ans. 15]
7. Convert the following infix notation of expression to an equivalent postfix notation of expression (Show status of Stack after execution of each operation) : (A + B) * C - D / E [Ans. AB + C * DE/–]
8. Consider each of the following postfix expressions :
   P_1 : 5, 3, +, 2, *, 6, 9, 7, –, /, – [Ans. 13]
   P_2 : 3, 5, +, 6, 4, –, *, 4, 1, –, 2, ↑, + [Ans. 25]
   P_3 :, 3, 1, +, 2, ↑, 7, 4, –, 2, *, +, 5, – [Ans. 17]
   Translate each expression into infix notation and then evaluate.
9. Give postfix form of the following expression :
   $A * (B + (C + D) * (E + F) / G) * H$ [Ans. $A\ B\ C\ D + E\ F + * G / + * H *$]

10. Give postfix form for $A + [(B + C) + (D + E) * F] / G$. [Ans. $A\,B\,C + D\,E + F * + G / +$]

11. Give postfix form of expression for the following :

    (*i*) *NOT A OR NOT B AND NOT C*

    (*ii*) *NOT (A OR B) AND C*

    [Ans. (*i*) A NOT B NOT C NOT AND OR
    (*ii*) A B OR NOT C AND]

12. Evaluate the following postfix notation of expression :

    True, False, NOT, AND, True, True, AND, OR

13. Evaluate the following postfix expression using a stack and show the contents of the stack after execution of each operation 5, 11, –, 6, 8, +, 12, *, / [Ans. – 1/ 28]

14. Let P be the postfix arithmetic expression : 7, 2, –, 1, 14, –, 1, 2, * Evaluate P using stack and showing the status of the stack at every step. [Ans. 2]

15. Consider the infix expression

    $$Q : A + B * C \uparrow (D / E) / F.$$

    Translate Q into P, where P is the postfix equivalent expression of Q. What will be the result of Q if this expression is evaluated for A, B, C, D, E, F as 2, 3, 2, 7, 2, 2 respectively.

    [Ans. $A\,B\,C\,D\,E / \uparrow * F / +$; 14]

16. Write equivalent Postfix expressions for the infix expressions given below :

    (*i*) $A + B - D / X$ (*ii*) $(X + Y) / (Z * Y) - R$ [Ans. Postfix : (*i*) $A\,B + D\,X / -$ (*ii*) $X\,Y + Z\,Y * / R -$]

17. Evaluate the following postfix notation of expression : 10 20 + 25 15 – * 30 / [Ans. 10]

18. Evaluate the following postfix notation of expression : 20 10 + 5 2 * – 10 / [Ans. 2]

19. Evaluate the following postfix expression using a stack and show the contents of stack after execution of each operation :

    120, 45, 20, +, 25, 15, –, +, * [Ans. 9000]

20. Evaluate the following postfix expression using a stack and show the contents of stack after execution of each operation :

    20, 45, +, 20, 10, –, 15, +, * [Ans. 1625]

21. Use a stack to evaluate the following postfix expression and show the content of the stack after execution of each operation. Don't write any code. Assume as if you are using push and pop member functions of the stack AB - CD + E* +

    (where A = 5, B = 3, C = 5, D = 4, and E = 2) [Ans. 20]

22. Change the following infix expression into postfix expression. $(A + B) * C + D / E - F$

    [Ans. $A\,B + C * D\,E / + F -$]

23. Evaluate the following postfix notation of expression :

    (Show status of Stack after each operation)

    True, False, NOT, OR, False, True, OR, AND

24. Convert the following infix expression to its equivalent postfix expression showing stack contents for the conversion :

    $$X - Y / (Z + U) * V$$

25. Convert the following infix expression to its equivalent postfix expression showing stack contents for the conversion :

    A + B * (C - D) / E

Type C : Programming Practice/Knowledge based Questions

1. Write a program that depending upon user's choice, either pushes or pops an element in a stack.
2. A line of text is read from the input terminal into a stack. Write an program to output the string in the reverse order, each character appearing twice.
   (*e.g.*, the string *a b c d e* should be changed to *ee dd cc bb aa*)
3. Write a program that depending upon user's choice, either pushes or pops an element in a stack. The elements are shifted towards right so that *top* always remains at 0^{th} ($zero^{th}$) index.
4. Write a program to insert or delete an element from a queue depending upon user's choice. The elements are not shifted after insertion or deletion.
5. Write a program to insert or delete an element from a queue depending upon user's choice.
6. Write a program to implement input-restricted deque. (Both the operations *i.e.*, insertions and deletions should be taken care of.)
7. Each node of a STACK contains the following information :
   (*i*) Pin code of a city,
   (*ii*) Name of city.
   Write a program to implement following operations in above stack
   (*a*) PUSH( ) To push a node in to the stack.
   (*b*) POP( ) To remove a node from the stack.
8. Write a program that implements *three* queues namely **HighestPr**, **NormalPr** and **LowestPr**. The program accepts an element alongwith its priority from the user, *e.g.*,

```
Enter element : KP213
Priority (Highest/ Normal / Lowest (H/N/L): H
```

As per priority entered, the element is added in the corresponding queue.
A menu offers following options :

```
1. Insert Element
2. Search for Element
3. Change Priority
```

For **option 1,** element is inserted in one queue as described above.

For **option 2,** ask for an element and search for it in all three queues. If found, display its queue name, otherwise display *"Element not found"*.

For **option 3,** ask for an element, if it exists in a queue, ask for its new priority as :

```
Want to Increase/Decrease its priority (I/D) ? I
```

And then remove the element from its existing queue and add it to the new queue as per the changed priority.