Data Structure (05201151)

Dr. Priya R Swaminarayan, Dean-FITCS
Director, MCA
Parul University

Syllabus

1. Introduction
2. Stack and Queue
3. Linked List
4. Trees
5. Graph
6. Searching, Sorting and Hashing

Books

1. An Introduction to Data Structures with Applications (TextBook) Jean-Paul Tremblay, Paul G. Sorenson; Tata McGraw-Hill; 2nd Edition, (2007)
2. Introduction to Algorithm Cormen, Leiserson, Rivest, Stein; PHI (2003); 2nd Edition
3. Data Structures using C and C++ Tanenbaum; PHI
4. Expert Data Structures with C R. B. Patel
5. Theory and Problems of Data Structures Seymour Lipschutz; Schaums Outline Series
6. Data Structures Through C++ Yashavant Kanetkar; BPB

Course Outcome

- After Learning the course the students shall be able to:

1. describe the significance of various linear and non-linear data structures such as arrays, stack, queue, linked list, trees and graph.
2. identify the appropriate data structure for a given problem.
3. construct most suitable data structure to solve a problem by considering various problem characteristics such as data size and various type of operations.
4. design and implement various techniques for searching, sorting and hashing.

Terms

Data: are facts and statistics collected together for the analysis.

Collection of values, e.g., student's name and its id are the data about the student.

Record: Records can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

Structure : It is a particular way of organising information , so that it will be easier to use.

Data organization refers to the method of classifying and organizing data sets to make them useful in more ways.

What is Data Structure ?

- Data Structures are programmatic way of storing and organizing the data.
- It is a data organization , management and storage format that enables efficient access and modifications.
- It also Allows handling data in an efficient way.
- It is playing a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

- **"It is a way we organize our data"**

- It is a way of organizing data that consider not only the items stored but also the relationship to each other.
- A useful tool for specifying the logical properties of data type.

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

What is Data Structure ?

Image source : Google

Need of Data Structure

- As applications are getting complex and amount of data is increasing day by day, there might arise below mentioned problems:

1. Processor speeds
2. Data Searching
3. Multiple request

Parul® University

DIGITAL LEARNING CONTENT

Advantages of Data Structures

- Efficiency – it is how we access and store data
- Reusability of DS
- Used to manage large amounts of our data
- Specific data structure are used for specific tasks

- Easy to understand the operations.
- Easy to define the relationships of each other.
- Easy to decide programming language.
- More useful for non-numeric applications.

Image source : Google

Applications of DS

- Compiler Designing
- Operating Systems
- Database Management Systems
- Simulation
- Network Analysis
- Artificial Intelligence
- Graph DS
- Numerical Analysis
- Statistical Analysis Package

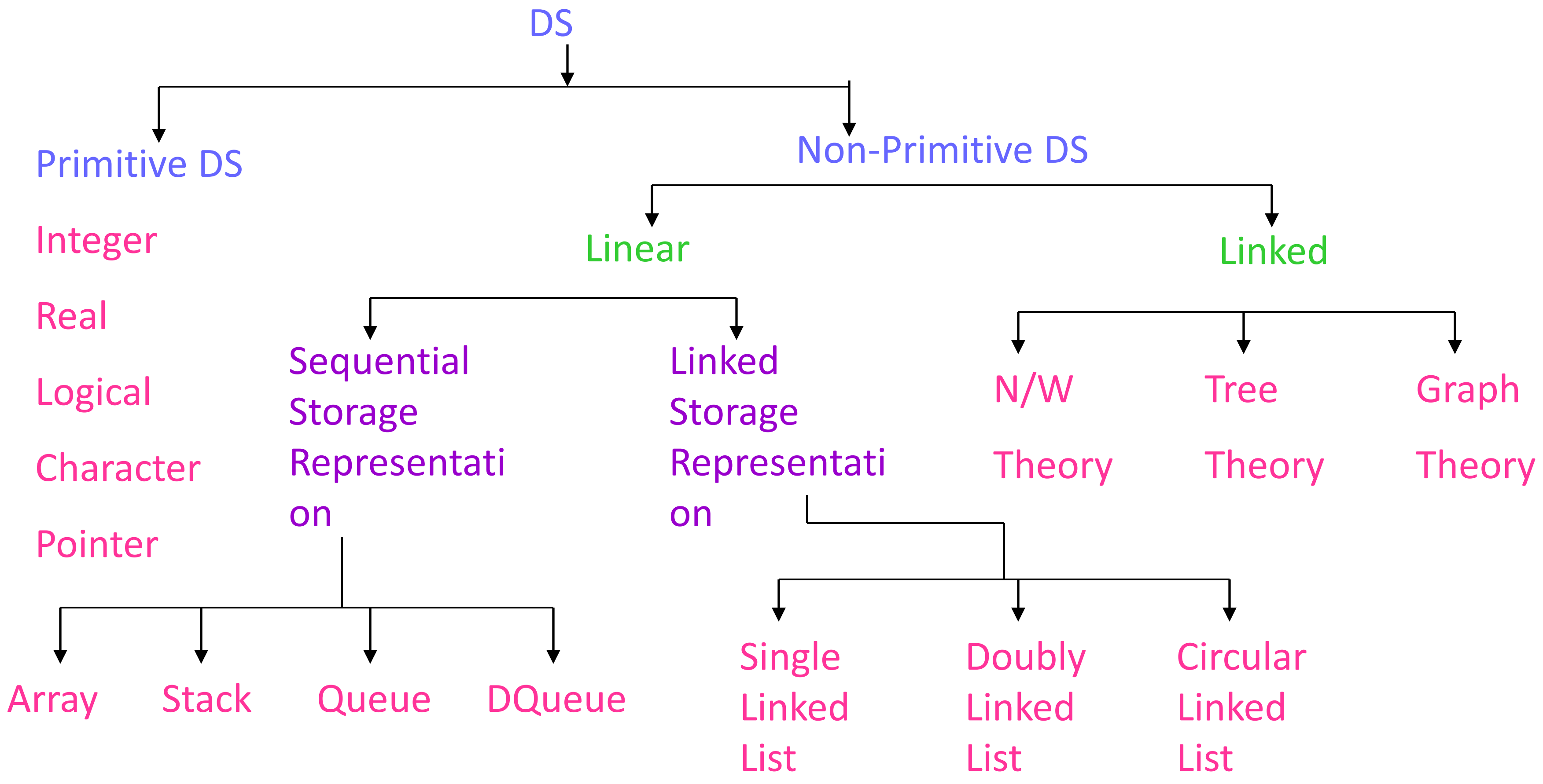
Storage and File Structure

1. **Storage structure**: The representation of a particular DS in the memory is called a Storage Structure.
2. **File Structure**: A storage structure representation in auxiliary memory is called a file structure.

Types of Data Structure

1. **Primitive Data Structure**: The DS that are directly operated by machine instructions is called Primitive DS. For E.g., integer, real, character, logical values, pointer type values etc.
2. **Non-Primitive Data Structure**: The DS that are not directly operated by machine instructions is called Non-primitive DS. For e.g. Array, Stack, Queue etc.

Family of Data Structure



Parul® University

DIGITAL LEARNING CONTENT

CHAPTER-2

Stack and Queue

STACK

1. Introduction to Stack
2. Stack operations
3. Applications of stack –
   - I. recursion
   - II. polish notations - prefix, infix, postfix,
   - III. Algorithms of stack applications,

Stack

- Definition: It's an ordered collection of items into which new items may be inserted from one end and items may be deleted from same end called TOP of the stack.
- It is called as LIFO (Last In First Out) method.
- i.e. the element which is inserted first in the stack, will be deleted last from the stack.
- E.g. of stack is stack of plates. How it works?

- Refer following link for Stack Operations:
- https://yongdanielliang.github.io/animation/web/Stack.html

Representation of Stack with example

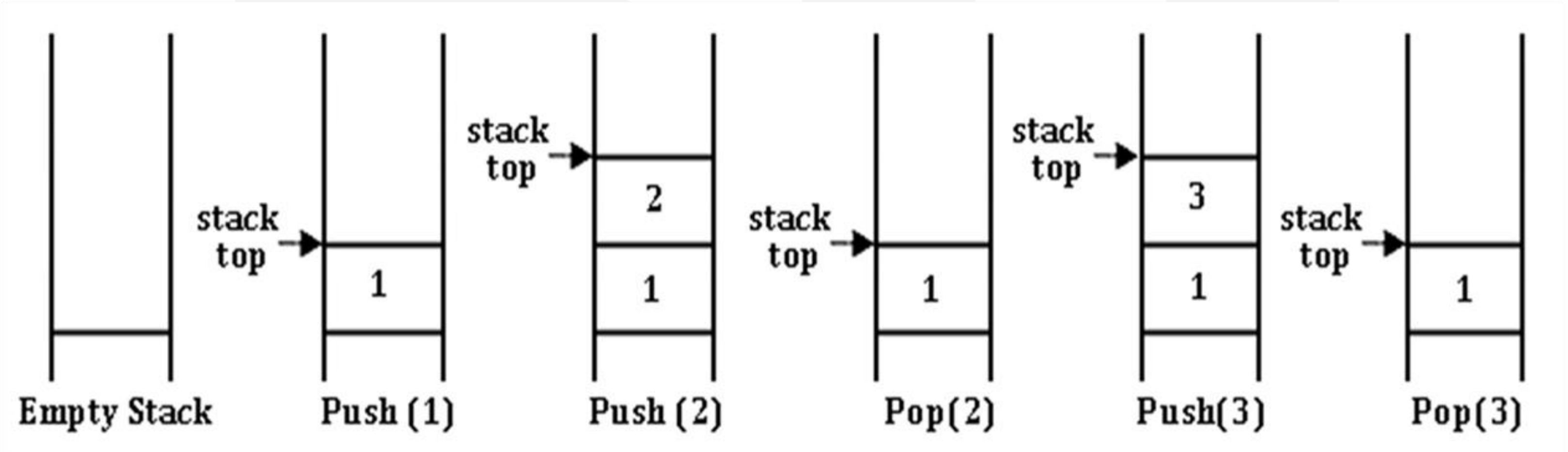
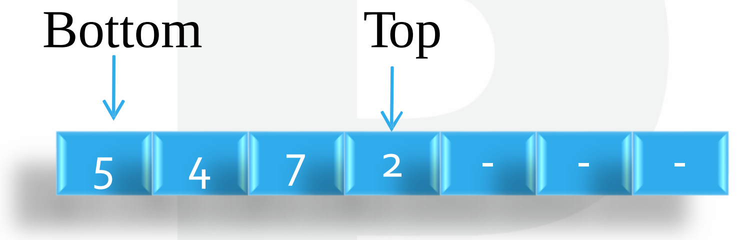
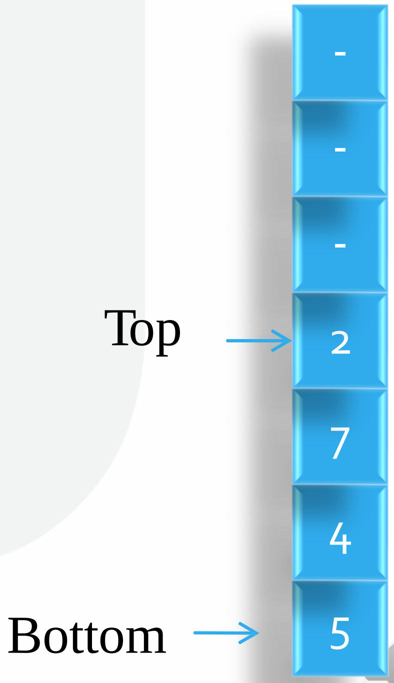


Image source : Google

Representation of stack in memory





Stack- Examples

- Arrangements of disk plates in a rack
- Playing cards is example of stack
- Social media – Instagram posts/ Facebook Post / LinkedIn Post etc.

\

Image source : Google

Terminology in stack

1. TOP:

- The pointer at the top most position of stack is called as TOP.
- It is the more commonly accessible element of stack.
- Insertion and deletion occur at top of the stacks.

2. Stack overflow: If the stack is full and we are trying to insert new element into the stack, then system will give the stack overflow error.

3. Stack underflow: If the stack is empty and we are trying to delete element from the top of the stack then system will give the stack underflow error.

Stack Representation

- A stack can also be implemented by means of Array, Structure, Pointer, and Linked List.
- It can either be a fixed size one or it may have a sense of dynamic resizing.
- Now we are going to implement stack using arrays, which will make it a fixed size stack implementation.

- Theoretically, there is no limit on the size of stack but practically,
- In **dynamic memory allocation**, stack size depends on memory size.
- In **static memory allocation**, stack size depends on array size

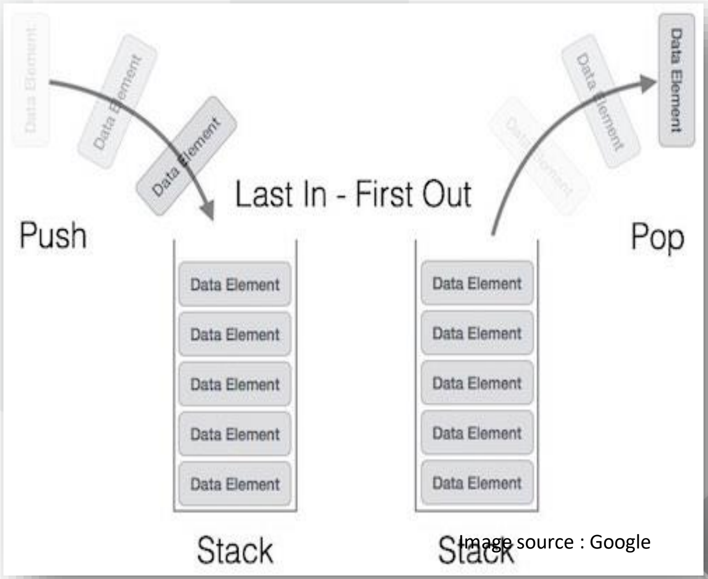


Image source : Google

Operations on Stack

1. **PUSH :** To insert an item into the stack data structure
2. **POP :** To remove an item from a stack data structure.
3. **PEEP :** To find ith element from stack data structure.
4. **CHANGE :** To change ith element from stack data structure.

Applications of Stack

1. Recursion
2. Expression evaluations and conversions a+b*c
3. Conversion of expression from infix to postfix
4. Conversion of expression from postfix to assembly code
5. Tower of Hanoi
6. Tree Traversals

Pros of Stack

1. Helps managing the data in particular way (LIFO) which will not be possible with Linked list and array.
2. When function has been called the local variables are stored in stack and destroyed once returned. Stack is used when variable will not be used outside the function. [Recursive function]
3. So, it will give control over how memory is allocated and de-allocated
4. Stack frees you from the burden of remembering to cleanup(read delete) the object
5. Not easily corrupted data structure (No one can easily inset data in middle)

Cons of Stack:

1. Stack memory will be limited.
2. Creating too many objects on the stack can increase the chances of stack overflow
3. Random access is not possible/allowed

Features of Stack

- Stack is an ordered list of similar type of data type.
- Stack is called a LIFO(Last in First out) structure or we can say FILO(First in Last out).
- push() function will be used to insert new elements into the Stack and pop() function will be used to remove an element from the stack.
- Both insertion and removal are allowed at only one end of Stack called Top.

Algorithms – PUSH()

- **PUSH (Stack, TOP, Item)**
- This procedure will insert an element Item into the stack which is represented by array containing N elements with the pointer TOP denoting top element in the stack.

```
Step 1: [Check for stack overflow?]
    If TOP >= N then
        write('stack overflow')
    Exit
Step 2: [Increment the TOP pointer]
    TOP <-TOP +1
Step 3: [Insert an element into the stack]
    Stack[TOP] <-Item
Step 4: [Finished]
    Return
```

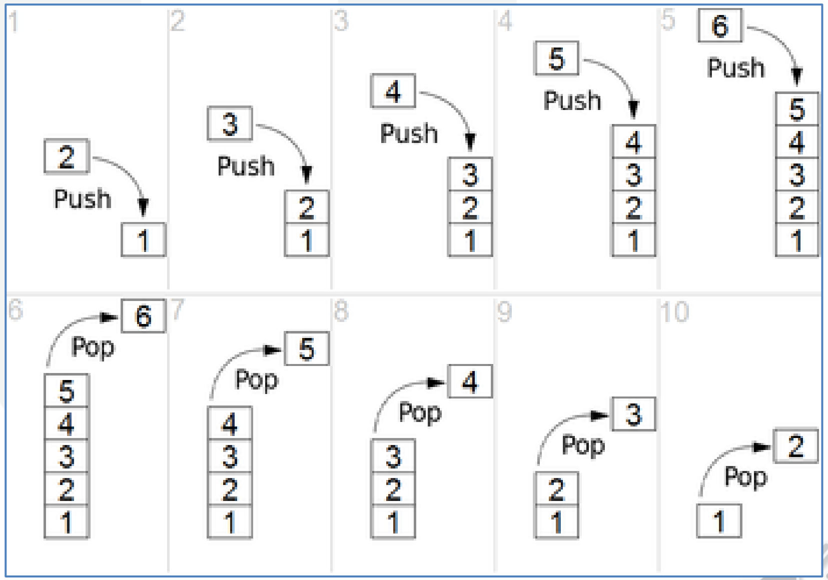


Image source : Google

https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FStack_(abstract_data_type)&psig=AOvVaw0AFgaXALavnxtcYaYwWGUf&ust=1595305832944000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCICA7_v_2uoCFQAAAAAdAAAAABAD

Parul® University

DIGITAL LEARNING CONTENT

Algorithms – POP()

- **POP(Stack, TOP)**
- This algorithm deletes an element Item from the top of a stack S containing N elements.

```
Step 1: [Check for stack underflow?]
        if TOP == 0 then
                write ('Stack Underflow')
        Exit
Step 2: [Accessing the value to be deleted]
        Item <- Stack[TOP]
Step 3: [Decrement the stack pointer]
        TOP<-TOP-1
Step 4: [Return deleted element Item ]
        Return Item
```

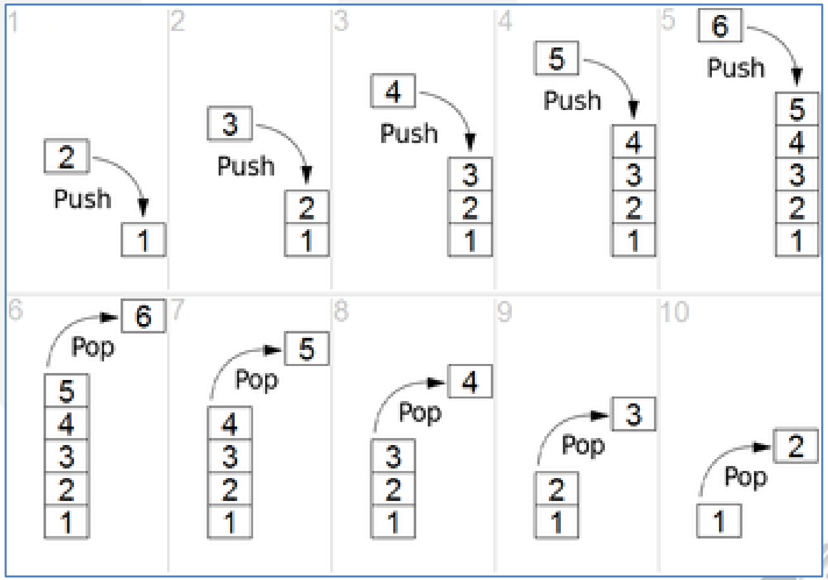


Image source : Google

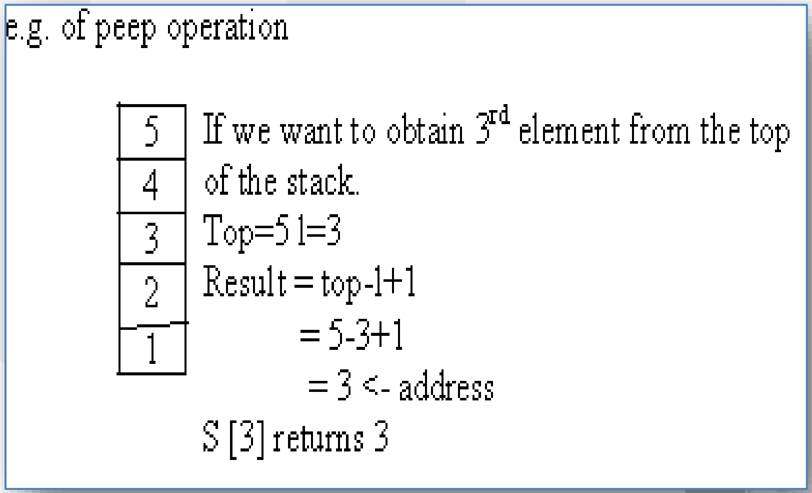
Algorithms – PEEP()

- **PEEP(Stack, TOP, I)**
- Given an array Stack containing N elements. Pointer TOP elements top element of the stack. This algorithm fetched the ith element in the value item.

```
Step 1: [Check for stack underflow?]
        If (TOP-i+1) <= 0 then
                write('Stack underflow')
        Exit
Step 2: [Storing the ith element in item]
        item <- Stack[TOP-i+1]
Step 3: [Finished]
        Return item
```

Top = 5, I = 4   5-4+1 = 2   item = s[top-i+1]
= s[2] = 20

S[5-5+1 = 1] = 1



e.g. of peep operation

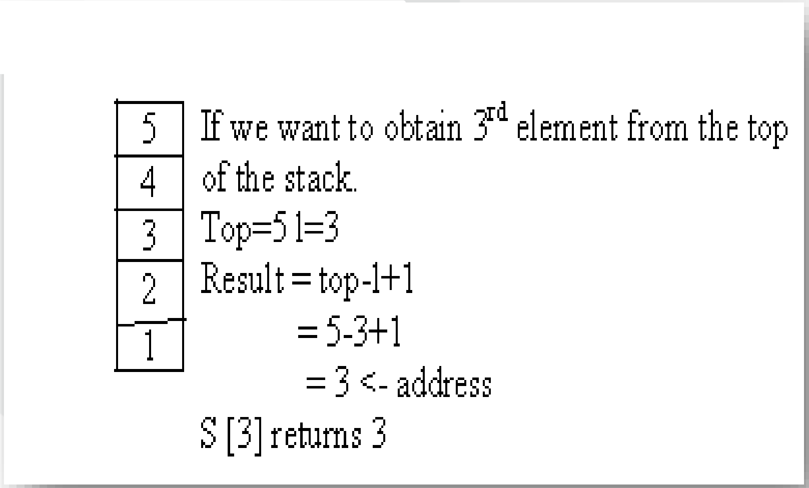
If we want to obtain 3rd element from the top of the stack.
Top=5 I=3
Result = top-I+1
= 5-3+1
= 3 <- address
S [3] returns 3

Image source : Google

Algorithms – CHANGE()

- **CHANGE (S, TOP, I, item):**
- This algorithm updates the ith element with the value item in a stack Stack containing N elements.

```
Step 1: [Check for stack underflow?]
        if (TOP-i+1 <= 0) then
            write('Stack underflow on change')
        Exit
Step 2: [Changing the ith element in stack]
            Stack[TOP-i+1] <- item
Step 3: [Finished]
            Return
```



If we want to obtain 3rd element from the top of the stack.
Top=5 I=3
Result = top-I+1
= 5-3+1
= 3 <- address
S [3] returns 3

Image source : Google

```
#include<stdio.h>
#include<conio.h>
Cost int N=5;
#define N 5
int s[N];
int top=0;     top=-1

int push(int x)
{
  if (top>N-1)
  {
          printf("STACK OVERFLOW ERROR");
          return(0);
  }
  else
  {
          s[top]=x;
          top++;
          return(1);
  }
}
```

```
int pop()
{
        if (top==-1)
        {
                printf("Stack underflow")
        }
        else
        {
        int i;
        i=s[top-1];
        top--;
        return(i);
        }
}
```

A[5] = a[0],a[1].a[2],a[3],a[4]   top = -1

=

```
int peep(int i)
{
        if (top-i<=0)
        {
                printf("Stack Underflow Error");
                return(0);
        }
        else
        {
                return(s[top-i]);
        }
}

int change(int i,int j)
{
        if (top-i<=0)
        {
                printf("Stack Underflow Error");
                return(0);
        }
        else
        {
                s[top-i]=j;
                return(s[top-i]);
        }
}
```

If stack contains 1 element then top = 0, then return I = 1

```
void display()
{
        int i;
        if (top<=N)
        {
        for (i=top-1;i>=0;i--)
        {
                printf("S[%d]->%d\n",i,s[i]);
        }
        }
}
```

```
void main()
{
                int ch=1,x,t;
                clrscr();

                while (ch)
                {

                printf("enter 1 for PUSH\n");
                printf("enter 2 for POP\n");
                printf("enter 3 for PEEP\n");
                printf("enter 4 for CHANGE\n");
                printf("enter 0 for EXIT\n");
                scanf("%d",&ch);
                switch(ch)
                {
                    case 1:printf("enter element value which is to be inserted into the stack\n");
                              scanf("%d",&x);
                              t=push(x);
                               if (t==1)
                                   printf("Inserted Successfully");
                                   break;
                    case 2:  t = pop();
                               printf("Deleted Element --> %d“,t);
                               break;
```

```
    case 3:printf("enter position");
            scanf("%d",&x);
            printf("Returned %dth element from the top of the stack %d\n",x,peep(x));
            break;
    case 4:printf("enter position");
            scanf("%d",&x);
            printf("enter new element");
            scanf("%d",&t);
            printf("Returned %dth element from the top of the stack %d\n",x,change(x,t));
            break;
    case 5:display();
  }
  printf("enter 1 for PUSH\n");
  printf("enter 2 for POP\n");
  printf("enter 3 for PEEP\n");
  printf("enter 4 for CHANGE\n");
  printf("enter 5 for DISPLAY\n");
  printf("enter 0 for EXIT\n");
  scanf("%d",&ch);
 }
}
```

Applications of Stack: 1) Recursion

- A **recursive function is a function** that calls **itself** during its execution.

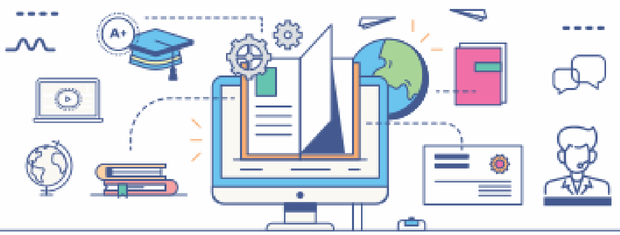
- Calling a procedure or function to itself is recursion.

```
e.g.  proc f(x)
      begin
                  f(x);
      end;
```

-   main()
-     f(x)
- Recursion is not available in all languages like COBOL, BASIC, and FORTRAN. So for that, there is one procedure which is useful to convert the recursion procedure to non-recursion procedure.

- In PASCAL, C, ALGOL-60 recursion is available.

- Main characteristics of recursion.
  1. Call itself
  2. Stopping criteria



Recursion

```
int factorial(int n)
{
        int fact;
        if(n==0)
                return(1);
        else
                fact = n*factorial(n-1);
                //return(fact);
}
void main()
{
        printf("Factorial of 3 = %d",factorial(3));
}
```

1) Actual parameter - 3 , return address (main fun), local variable - fact)

F = 1   3! = 6
N = 3
I = 1, 2, 3, 4
Fact = 1 = 1*1 = 1*2=2*3=6

[How recursion function works???](https://www.youtube.com/watch?v=ozmE8G6YKww)
https://www.youtube.com/watch?v=ozmE8G6YKww

```
int factorial(int n)
{
        int fact=1;
        for (int i = 1; I <= n ; i++)
                fact = fact * I;
        return(fact);
}
```

Parul® University

DIGITAL LEARNING CONTENT

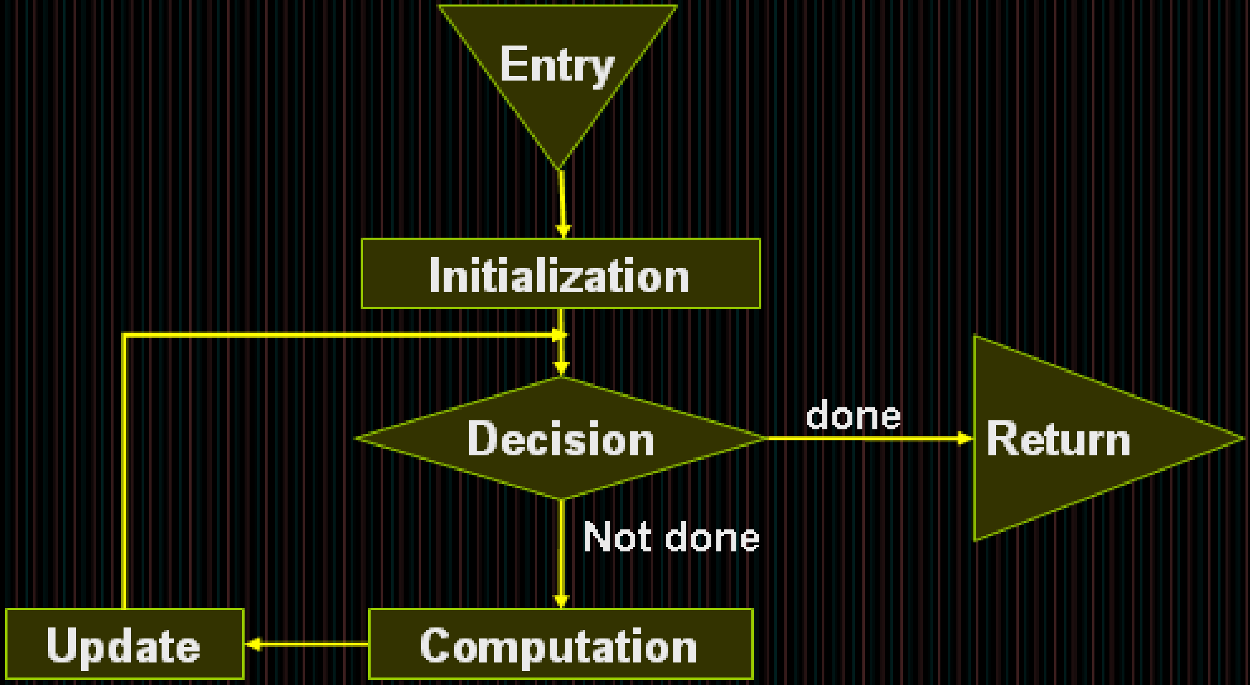
Recursion Vs Non-recursive function

- Many People believe that recursion is an unnecessary luxury in a programming language. This based on the fact that any primitive reclusive function and therefore any function we would normally like to compute, can be solved iteratively.
- There are four parts in the iterative process:

1. **Initialization**: The parameters of the function and a decision parameters in this part are set to their initial values. The decision parameter is used to determine when to exit from the loop
2. **Decision**: The decision parameter is used to determine whether to remain in the loop.
3. **Computation**: The required computation is performed in this part.
4. **Update**: The decision parameter is updated and a transfer to the next iteration results

Recursion

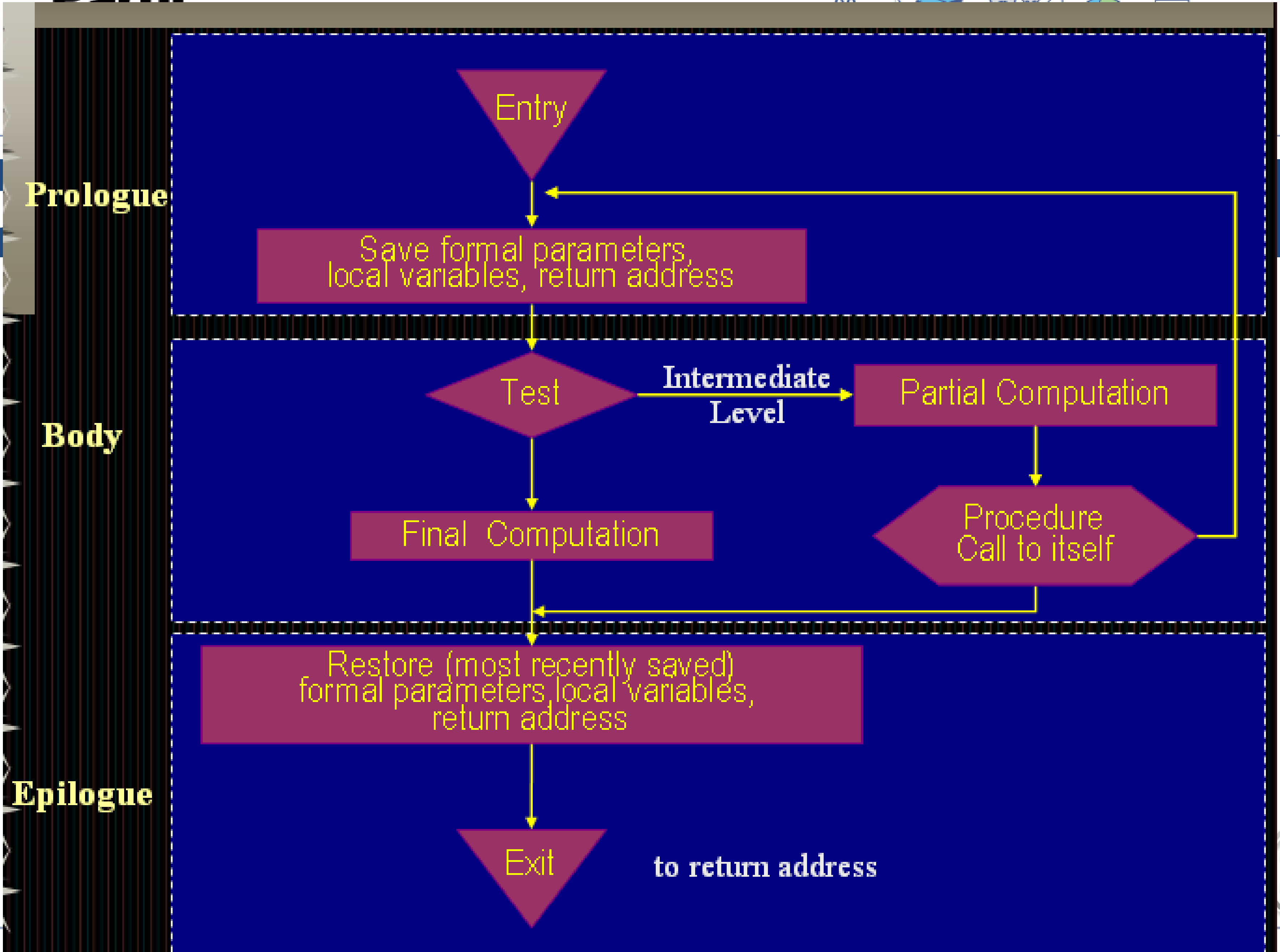
The flowchart for iterative process is show below:



Recursion

- The general algorithm model for any recursive procedure contains the following steps:

1. **Step-1 : [Prologue]** save the parameters, local variables and return address.
2. **Step:2 [Body]** If the base criterion has been reached then perform the final computation and go to step-3 otherwise perform the partial computation and go to step-1 (initiate a recursive call)
3. **Step:3 [Epilogue]** Restore the most recently saved parameter, Local variables and return address. Go to this return address.


Prologue
Entry
Save formal parameters, local variables, return address
Body
Test
Intermediate Level
Partial Computation
Procedure Call to itself
Final Computation
Epilogue
Restore (most recently saved) formal parameters,local variables, return address
Exit
to return address

Recursion Algorithm: Factorial:

- Given an **integer N**, this algorithm computes N!. the **stack A** is used to store an activated record associated with each recursive call. Each **activated record** contains the current value of **N** and current return address **RET_ADDR**.
- **TEMP_REC** is also a record which contains two variables (**PARM** and **ADDRESS**). This temporary record is required to simulate the proper transfer of control from one activated record to another.
- Whenever TEMP_REC is placed onto the stack A, copies of PARM and ADDRESS are pushed onto A and assigned to N and RET_ADDR respectively.
- **TOP** points to the top element of the array A and initially assigns to zero.
- Also, the return address is set to main calling address. PARM is set to the initial value of N.

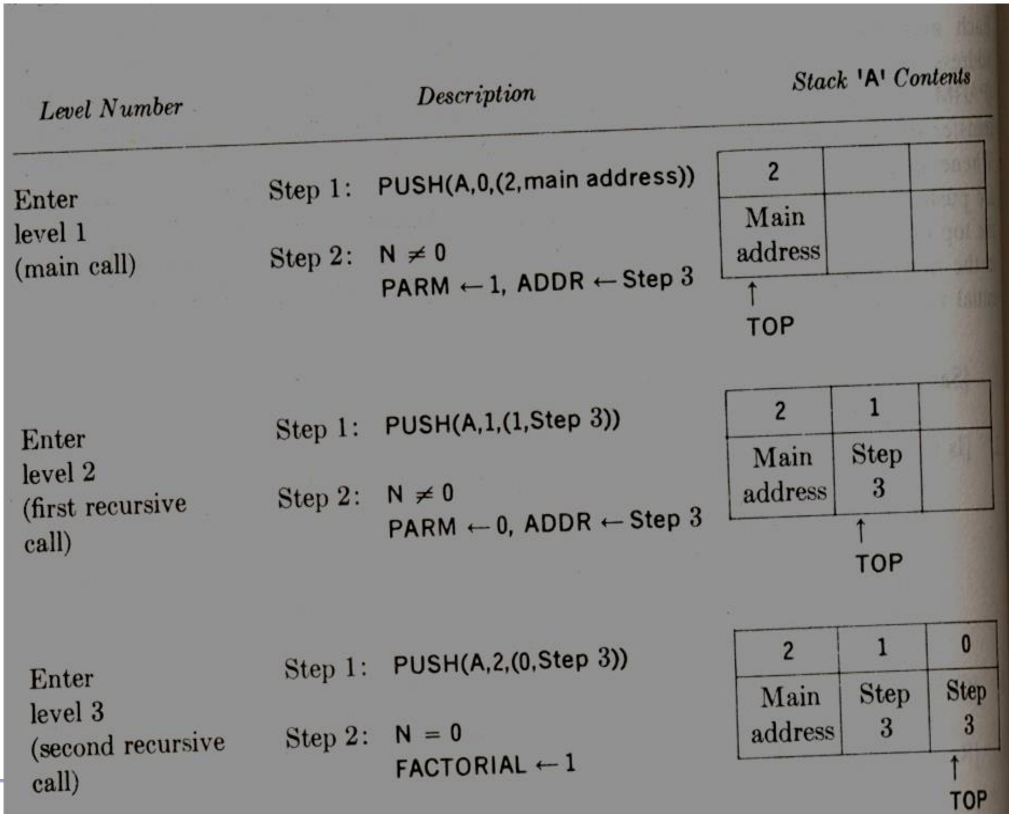
Recursion Algorithm: Factorial:

```
1.    Step-1 [Save N and return address]
      Call PUSH(A,TOP,TEMP_REC)
2.    Step-2 [Is the base criteria found?]
      If n=0 then
            Factorial ←1
            Go to step 4
      Else
            PARM ← N-1
            ADDRESS ← step 3
            Go to step 1
3.    Step-3 [ calculate N!]
      FACTORIAL ← N*FACTORIAL (the factorial of N)
4.    Step-4 [ Restore previous N and return address]
      TEMP_REC ← POP(A,TOP)
      (i.e. PARM=N,ADDRESS=RET_ADDR,popstack)
      GO TO ADDRESS
```

Recursion Algorithm: Factorial:

1. A-stack, N=NO=2, TOP=0
2. Activated_record (current value of N, return address RET_ADDR)
3. TEMP_REC (PARM,ADDR)

```
1.      Step-1 [Save N and return address]
        Call PUSH(A,TOP,TEMP_REC)
2.      Step-2 [Is the base criteria found?]
        If n=0 then
             Factorial ← 1
             Go to step 4
        Else
             PARM ← N-1
             ADDRESS ← step 3
             Go to step 1
3.      Step-3 [ calculate N!]
        FACTORIAL ← N*FACTORIAL (the factorial of N)
4.      Step-4 [ Restore previous N and return address]
        TEMP_REC ← POP(A,TOP)
        (i.e. PARM=N,ADDRESS=RET_ADDR,popstack)
       GO TO ADDRESS
```



Level Number	*Description*	*Stack 'A' Contents*
Enter level 1 (main call)	Step 1: PUSH(A,0,(2,main address)) Step 2: $N \neq 0$ PARM ← 1, ADDR ← Step 3	2 | | Main address | | ↑ TOP (under column 1)
Enter level 2 (first recursive call)	Step 1: PUSH(A,1,(1,Step 3)) Step 2: $N \neq 0$ PARM ← 0, ADDR ← Step 3	2 | 1 | Main address | Step 3 | ↑ TOP (under column 2)
Enter level 3 (second recursive call)	Step 1: PUSH(A,2,(0,Step 3)) Step 2: $N = 0$ FACTORIAL ← 1	2 | 1 | 0 Main address | Step 3 | Step 3 ↑ TOP (under column 3)

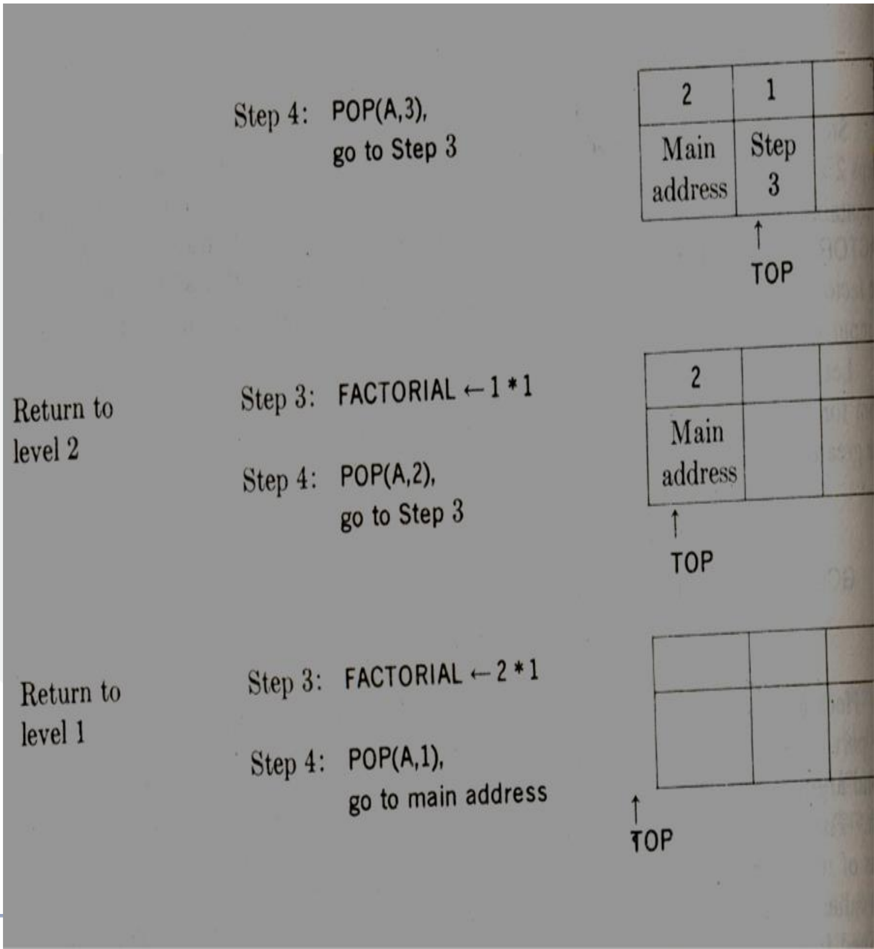
Recursion Algorithm: Factorial:

```
1.    Step-1 [Save N and return address]
      Call PUSH(A,TOP,TEMP_REC)
2.    Step-2 [Is the base criteria found?]
      If n=0 then
            Factorial ← 1
            Go to step 4
      Else
            PARM ← N-1
            ADDRESS ← step 3
            Go to step 1
3.    Step-3 [ calculate N!]
      FACTORIAL ← N*FACTORIAL (the factorial of N)
4.    Step-4 [ Restore previous N and return address]
      TEMP_REC ← POP(A,TOP)
      (i.e. PARM=N,ADDRESS=RET_ADDR,popstack)
      GO TO ADDRESS
```

Parm = 1
Add = step3

Parm = 2
Add = step3

Parm = Garbage value
Add = main



Step 4: POP(A,3),
go to Step 3

2	1	
Main address	Step 3	

↑ TOP

Return to level 2

Step 3: FACTORIAL ← 1 * 1

Step 4: POP(A,2),
go to Step 3

2		
Main address		

↑ TOP

Return to level 1

Step 3: FACTORIAL ← 2 * 1

Step 4: POP(A,1),
go to main address

↑ TOP

Polish expressions and their compilations

❖ **Infix Expression:**

1. We write expression in **infix** notation, e.g. a - b,
2. where operators are used **in**-between operands.
3. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.
4. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.
5. Ope Operator Ope

❖ **Prefix Expression :**

1. In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands.
2. For example, **+ab**.
3. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Polish expressions and their compilations

❖ **Postfix Expression :**

1. This notation style is known as **Reversed Polish Notation**.
2. In this notation style, the operator is **postfix**ed to the operands
3. i.e., the operator is written after the operands.
4. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Polish expressions and their compilations

1. Infix notation:

- The arithmetic operators appears between the operands.
- E.g. A+B*C

2. Suffix (postfix) notation or Reverse Polish Notation:

- The arithmetic operators appears after the operands
- E.g. A+B*C can be represented as ABC*+

3. Prefix notation or Polish notation:

- The arithmetic operators appears before the operands
- E.g. A+B*C can be represented as +*ABC

Polish expressions and their compilations

- Rank (operand) = +1
- Rank (operator) = -1
- Valid Rank (expression) = 1
- Rank of a +b = +1 -1 +1 = +1 (i.e. it's a valid expression)
- Rank of a+b+ = +1 -1 +1 -1 = 0 (i.e. it's an invalid expression)

Polish expressions and their compilations

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Polish expressions and their compilations

❖ **Precedence of Operators**

1. When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

   a + b * c ⇨ a + (b * c)

2. As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later

Polish expressions and their compilations

❖ **Associativity**

1. Associativity describes the rule where operators with the same precedence appear in an expression.
2. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.
3. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.
4. Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Polish expressions and their compilations

❖**Associativity**

1. The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –
2. In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.
3. 1 * 3 * 2 / 1

SN.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

Polish expressions and their compilations

Algorithms of expressions

1. Conversion of infix to postfix (un-parenthesized)
2. Conversion of infix to postfix (parenthesized)
3. Conversion of Polish (postfix) expression to code

Conversion of infix to postfix (Parenthesized)

Algorithm : REVPOL.

1. Given an input string **INFIX** containing an infix expression **which has been padded on the right with ‘)’** and whose symbols have precedence values given by Table ,
2. a **vector S**, used as a stack, and
3. a function **NEXTCHAR** which , when invoked , returns the next character of its argument
4. this algorithm converts INFIX into reverse Polish and places the result in the string POLISH.
5. The integer variable **TOP** denotes the top of the stack.
6. Algorithms **PUSH** and **POP** are used for stack manipulation.
7. The integer variable **RANK** accumulates the rank of the expression.
8. Finally , the string variable **TEMP** is used for temporary storage purposes.

Conversion of infix to postfix (Parenthesized)

Algorithm : REVPOL. Given an input string **INFIX** containing an infix expression which has been padded on the right with ')' and whose symbols have precedence values given by Table , a vector **S**, used as a stack, and a function **NEXTCHAR** which , when invoked , returns the next character of its argument , this algorithm converts INFIX into reverse Polish and places the result in the string **POLISH.** The integer variable **TOP** denotes the top of the stack. Algorithms PUSH and POP are used for stack manipulation. The integer variable **RANK** accumulates the rank of the expression. Finally , the string variable **TEMP** is used for temporary storage purposes.

```
1.[Initialize stack]
          TOP <- 1
          S[TOP] <- '('
2. [Initialize output string and rank count]
          POLISH <- "
          RANK <- 0
```

Conversion of infix to postfix (Parenthesized)

```
3. [Get first input symbol]
          NEXT <- NEXTCHAR(INFIX)
4. [Translate the infix expression]
          Repeat thru step 7 while  NEXT ≠  ""
5. [Remove symbols with greater precedence from stack]
          If TOP < 1
          then Write('INVALID')
                    Exit
          Repeat while f(NEXT) < g(S[TOP])
                    TEMP <- POP(S,TOP)
                    POLISH <- POLISH  O TEMP
                    RANK <- RANK + r(TEMP)
                    If RANK < 1 then
                              Write('INVALID')
                              Exit
```

Conversion of infix to postfix (Parenthesized)

```
6. [Are there matching parentheses?]
        If f(NEXT) ≠ g(S[TOP])
        then Call PUSH(S,TOP,NEXT)
        else POP(S,TOP)
7. [Get next input symbol]
        NEXT <- NEXTCHAR(INFIX)
8. [Is the expression valid?]
        If TOP ≠ 0 or RANK ≠ 1
        then Write('INVALID')
        else Write('VALID')
        Exit
```

Polish expressions and their compilations

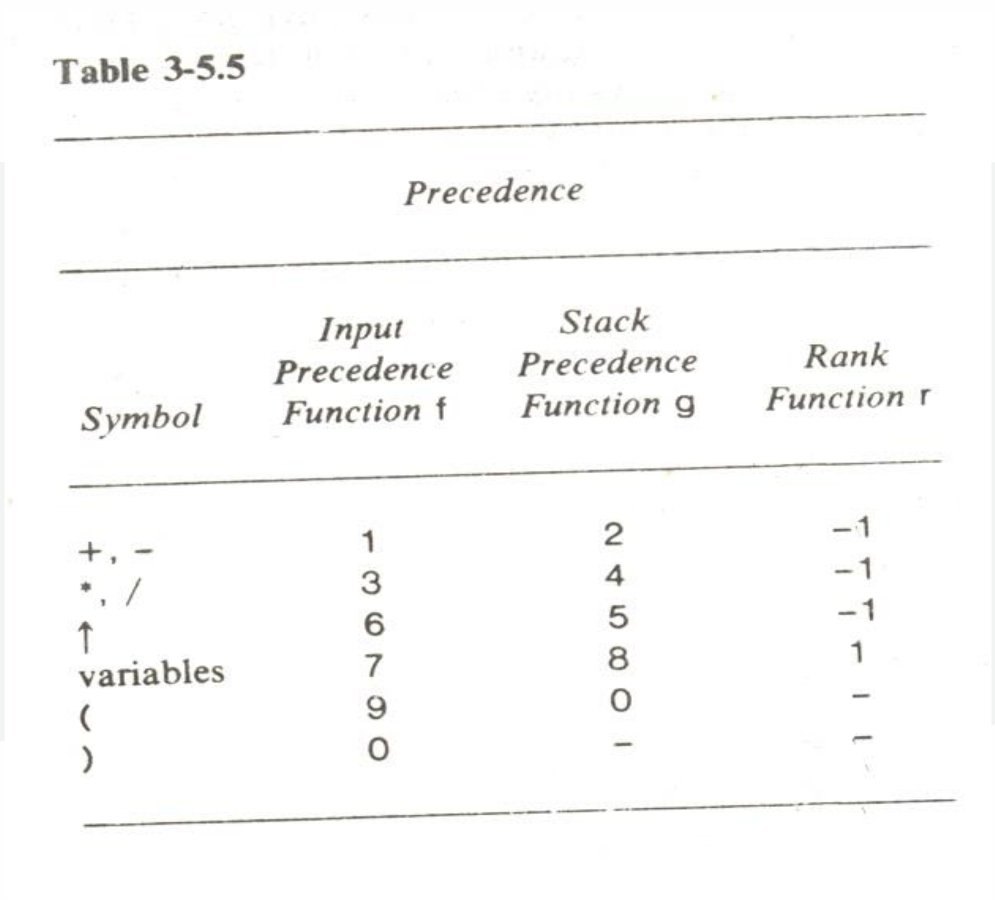
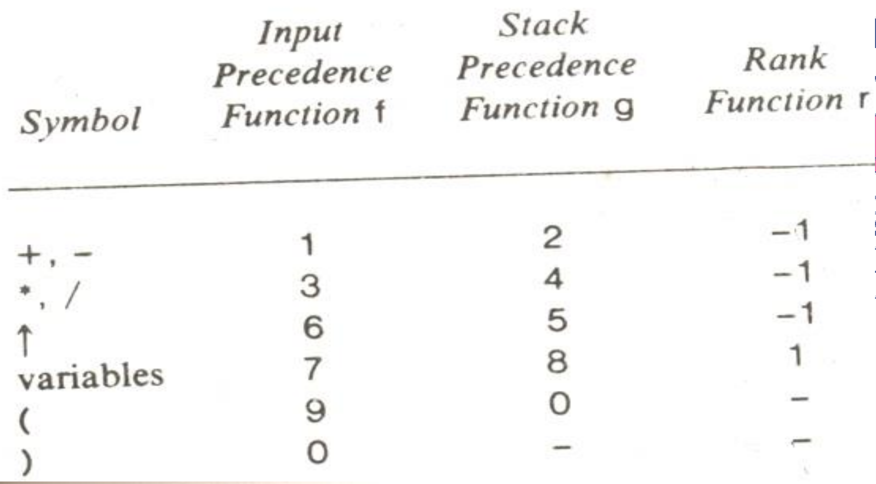


Table 3-5.5

	Precedence		
Symbol	Input Precedence Function f	Stack Precedence Function g	Rank Function r
+, −	1	2	−1
*, /	3	4	−1
↑	6	5	−1
variables	7	8	1
(	9	0	−
)	0	−	−

Parul® University

```
1.[Initialize stack]
        TOP <- 1
        S[TOP] <- '('
2. [Initialize output string and rank count]
        POLISH <- "
        RANK <- 0
3. [Get first input symbol]
        NEXT <- NEXTCHAR(INFIX)
4. [Translate the infix expression]
        Repeat thru step 7 while NEXT != "#"
5. [Remove symbols with greater precedence from stack]
        If TOP < 1
        then Write('INVALID')
                    Exit
        Repeat while f(NEXT) < g(S[TOP])
        TEMP <- POP(S,TOP)
        POLISH <- POLISH  O TEMP
        RANK <- RANK + r(TEMP)
        If RANK < 1
        then Write('INVALID')
            Exit
6. [Are there matching parentheses?]
        If f(NEXT)   !=    g(S[TOP])
        then Call PUSH(S,TOP,NEXT)
        else POP(S,TOP)
7. [Get next input symbol]
        NEXT <- NEXTCHAR(INFIX)
8. [Is the expression valid?]
        If TOP   !=   0 or RANK  !=    1
        then Write('INVALID')
        else Write('VALID')
```



Symbol	Input Precedence Function f	Stack Precedence Function g	Rank Function r
+, −	1	2	−1
*, /	3	4	−1
↑	6	5	−1
variables	7	8	1
(	9	0	−
)	0	−	−

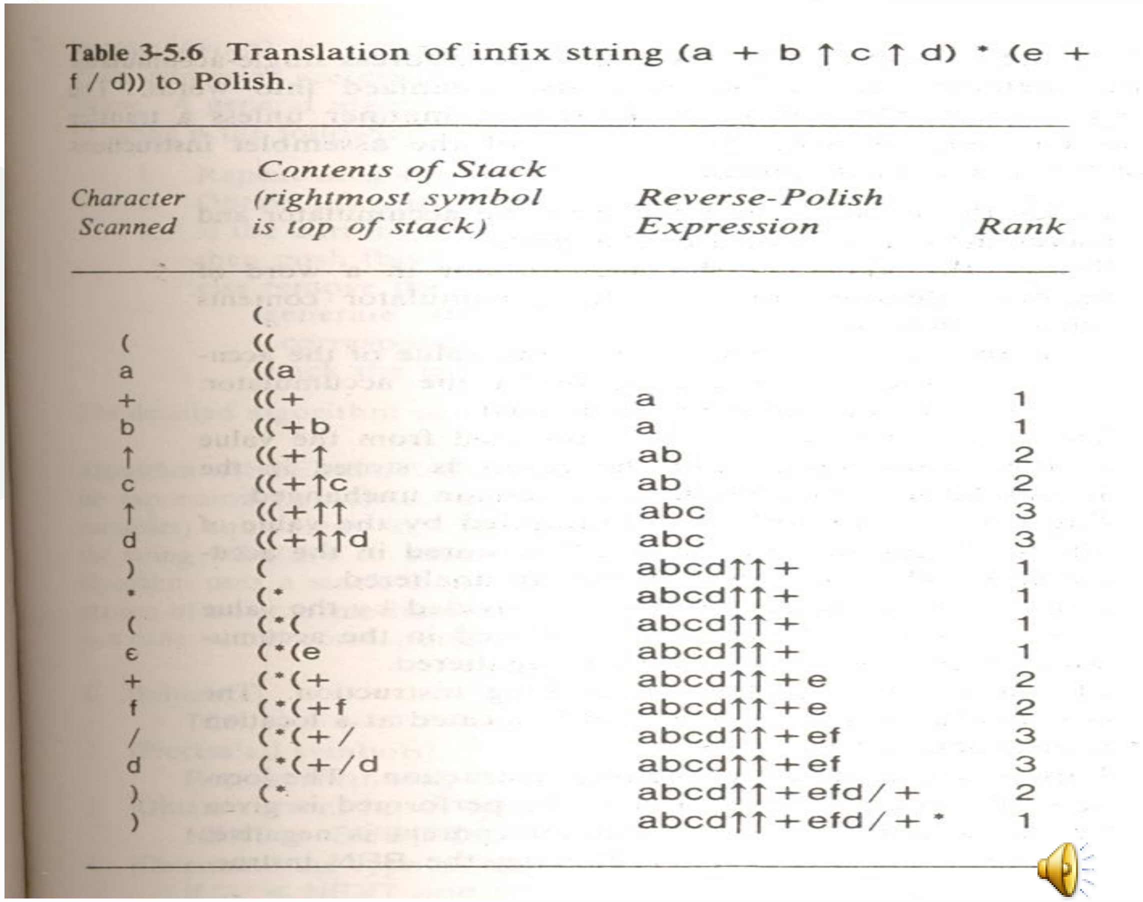


Table 3-5.6 Translation of infix string (a + b ↑ c ↑ d) * (e + f / d)) to Polish.

Character Scanned	Contents of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	(		
(	((		
a	((a		
+	((+	a	1
b	((+b	a	1
↑	((+↑	ab	2
c	((+↑c	ab	2
↑	((+↑↑	abc	3
d	((+↑↑d	abc	3
)	(	abcd↑↑+	1
*	(*	abcd↑↑+	1
(	(*(	abcd↑↑+	1
e	(*(e	abcd↑↑+	1
+	(*(+	abcd↑↑+e	2
f	(*(+f	abcd↑↑+e	2
/	(*(+/	abcd↑↑+ef	3
d	(*(+/d	abcd↑↑+ef	3
)	(*	abcd↑↑+efd/+	2
)		abcd↑↑+efd/+*	1

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

①

(S1, S2)

Scanned char	Stack Content	Postfix	Rank
	G	~~b~~	∅
NEXT = G	f(G) g(G)		
	$9 \not< 0$		
	GG		
NEXT = a	$f(a) < g(G)$		
	$7 \not< 0$		
	GGa		
NEXT = +	f(+) g(a)		
	$1 < 8$		
	GG~~a~~	a	1
	TEMP = a		
	f(+) g(G)		
	$1 \not< 0$		
	GG+		
NEXT = b	f(b) g(+)		
	$7 \not< 2$		
	GG+b		
NEXT = ↑	f(↑) g(b)		
	$6 < 8$		
	GG+~~b~~	ab	2
	f(↑) g(+)		
	$6 \not< 2$		
	GG+↑		

Character scanned	Stack content	Postfix	Rank
NEXT = C	GG + ↑	ab	2
	f(c) g(↑)		
	7 ≮ 5		
	GG +↑ c		
NEXT = ↑	f(↑) g(c)		
	6 < 8		
	GG + ↑ ~~c~~	abc	3
	f(↑) g(↑)		
	7 ≮ 8	abc	
	~~GG +↑~~		
	GG + ↑ ↑		
NEXT = d	f(d) g(↑)		
	~~7~~ ≮ 5		
	GG + ↑↑ d		
NEXT =)	f()) g(d)		
	0 < 8		
	GG + ↑↑ ~~d~~	abcd	4
	f()) g(↑)		
	0 < 5		3
	GG + ↑ ~~↑~~	abcd ↑	
	GG + ↑		
	f()) g(↑)		
	0 < 5	abcd ↑↑	2
	GG + ~~↑~~		
	f()) g(+)		
	0 < 2	abcd ↑↑ +	1
	GG ~~+~~		

Rank ③

Character | stack | " " Postfix | Rank

$\supset$ GG
f($\supset$) g(G)
0 ≮ 0
0 ≠ 0
G ~~G~~
G → NEXT = *
f(*) < g(G)
3 < 0

NEXT = G f(G) g(G)
9 ≮ 0
GG

NEXT = e f(e) g(G)
7 ≮ 0
GGe

NEXT = + f(+) g(e)
1 < 8 abcd↑↑ + e 2
GG
f(+) g(G)
1 ≮ 0
GG+

NEXT = f f(f) g(+)
7 ≮ 2
GG+f

NEXT = / f(/) g(f)
3 < 8 abcd↑↑+ef 3
GG+
f(/) g(+)
3 ≮ 2
GG+/

NEXT = d f(d) g(/)
7 ≮ 4
GG + / d

② ④

```
NEXT = ;      ( * ( + / d          abcd↑↑+ef          3
             f(;)   g(d)
               0 < 8
              ( * ( + / ~~d~~     abcd↑↑+ef d         4
            f(;)    g(/)
              0 < 4
             ( * ( + ~~/~~       abcd↑↑+efd /        3
            f(;)  g(+)
              0 < 2
            ( * ( ~~+~~         abcd↑↑+efd/+        2
            f(;)  g(()
             0 ≮ 0
             ( *
NEXT = ;    f(;)   g(*)
             0 < 4
             ( ~~*~~            abcd↑↑+efd/+*       1
            f(;)  g(()
             0 ≮ 0
            pop(()
```

Conversion of infix to postfix (un-parenthesized)

Algorithm : UNPARENTHESIZED_SUFFIX. Given an **input string** INFIX representing an infix expression whose single character symbols have precedence value and ranks , a vector S representing a stack , and a string function NEXTCHAR which , when invoked , returns the next character of the input string , the algorithm converts the string INFIX to its reverse Polish string equivalent , POLISH, RANK contains the value of each head of the reverse Polish string, NEXT contains The symbol being examined and TEMP is a temporary variable which contains the unstacked element. We assume that the given input string is padded on the right with the special symbol '#'

Symbols	*Precedence* f	*Rank* r
+ , -	1	-1
*,/	2	-1
a ,b , c,....	3	1
#	0	-

Conversion of infix to postfix (un parenthesized)

```
1.  [Initialize the stack]
        TOP <- 1
        S[TOP] <- '#'
2. [Initialize output string and rank count]
        POLISH <- "
        RANK <- 0
3. [Get first input symbol]
        NEXT <- NEXTCHAR(INFIX)
4. [Translate the infix expression]
        Repeat thru step 6 while NEXT ≠ '#'
```

Conversion of infix to postfix (un parenthesized)

```
5. [Remove symbols with greater or equal precedence from stack]
        Repeat while f(NEXT) <= f(S[TOP])
                TEMP <- POP(S,TOP) (this copies the stack contents into TEMP)
                POLISH <- POLISH O TEMP
                RANK <- RANK + r(TEMP)
                If RANK < 1
                then        Write('INVALID')
                            Exit
6. [Push current symbol onto stack and obtain next input symbol]
        Call PUSH(S,TOP,NEXT)
        NEXT <- NEXTCHAR(INFIX)
```

Conversion of infix to postfix (un parenthesized)

```
7. [Remove remaining elements from stack]
      Repeat while S[TOP] ≠ '#'
                  TEMP <- POP(S,TOP)
                  POLISH <- POLISH ○ TEMP
                  RANK <- RANK + r(TEMP)
                  If RANK < 1
                  then Write('INVALID')
                            Exit
8. [Is the expression valid?]
      If RANK = 1
      then        Write('VALID')
      else        Write('INVALID')
      Exit
```

```
3. [Get first input symbol]
•        NEXT <- NEXTCHAR(INFIX)
4. [Translate the infix expression]
•        Repeat thru step 6 while NEXT  !=   '#'
5. [Remove symbols with greater or equal precedence from stack]
•        Repeat while f(NEXT) <= f(S[TOP])
•                TEMP <- POP(S,TOP) (this copies the stack contents into TEMP)
•                POLISH <- POLISH O TEMP
•                RANK <- RANK + r(TEMP)
•                If RANK < 1
•                then        Write('INVALID')
•                            Exit
6. [Push current symbol onto stack and obtain next input symbol]
•        Call PUSH(S,TOP,NEXT)
•        NEXT <- NEXTCHAR(INFIX)
7. [Remove remaining elements from stack]
•        Repeat while S[TOP] !=    '#'
•                TEMP <- POP(S,TOP)
•                POLISH <- POLISH ○ TEMP
•                RANK <- RANK + r(TEMP)
•                If RANK < 1
•                then Write('INVALID')
•                            Exit
8. [Is the expression valid?]
•        If RANK = 1
•        then        Write('VALID')
•        else        Write('INVALID')
•        Exit
```

Symbols	*Precedence* f
+ , -	1
*,/	2
a ,b , c,….	3
#	0

A+b*c-d/e*h

A*b-c+d/e/f

Parul® University

DIGITAL LEARNING CONTENT

Conversion of infix to postfix (un parenthesized)

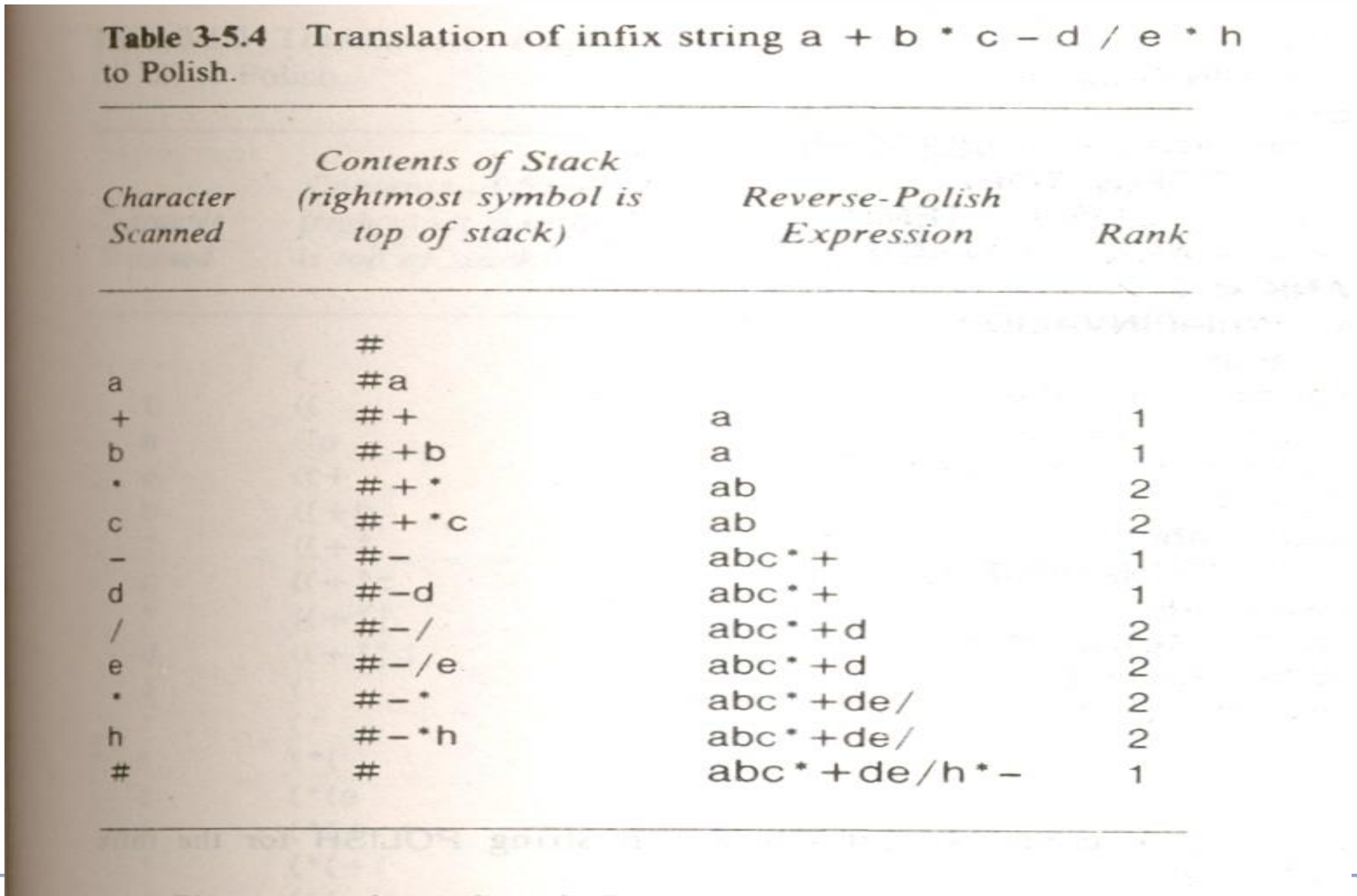


Table 3-5.4 Translation of infix string a + b * c − d / e * h to Polish.

Character Scanned	Contents of Stack (rightmost symbol is top of stack)	Reverse-Polish Expression	Rank
	#		
a	#a		
+	#+	a	1
b	#+b	a	1
*	#+*	ab	2
c	#+*c	ab	2
−	#−	abc*+	1
d	#−d	abc*+	1
/	#−/	abc*+d	2
e	#−/e	abc*+d	2
*	#−*	abc*+de/	2
h	#−*h	abc*+de/	2
#	#	abc*+de/h*−	1

a + b * c - d / e * h

Character Scanned	stack	o/p string	Rank
	#	$\cancel{b}$	0
NEXT = a	$f(a) = 3 \not<= f(\#) = 0$		
	#a		
+	$f(+) = 1 <= f(a) = 3$		
	# ~~a~~	~~ab~~ a	1
	$f(+) = 1 \not<= f(\#) = 0$		
	# +		
b	$f(b) = 3 \not<= f(+) = 1$		
	# + b		
*	$f(*) = 2 <= f(b) = 3$		
	# + ~~b~~	ab	2
	$f(*) = 2 \not<= f(+) = 1$		
	# + *		
c	$f(c) = 3 \not<= f(*) = 2$		
	# + * c		
–	$f(-) = 1 <= f(c) = 3$	abc	3
	# + * ~~c~~		
	$f(-) = 1 <= f(*) = 2$	abc *	2
	# + ~~*~~		
	$f(-) = 1 <= f(+) = 1$	abc * +	1
	# ~~+~~		
	$f(-) = 1 \not<= f(\#) = 0$		
	# ~~o~~ –		
d	$f(d) = 3 \not<= f(-) = 1$		
	# – d		
/	$f(/) = 2 <= f(d) = 3$	abc * + d	2
	# – ~~d~~		
	$f(/) = 2 \not<= f(-) = 1$		
	# – /		

a + b * c - d / e * h #

-/ abc* +d 2

e f(e) = 3 ≮= f(/) = 2

- / e

* f(*) = 2 <= f(e) = 3

- / ~~e~~ abc * + d e 3

f(*) = 2 <= f(/) = 2 abc * + de/ 2

- ~~/~~

f(*) = 2 ≮= f(-) = 1

~~ab~~

~~-~~ *

h f(h) = 3 ≮= f(*) = 2

- * h

# # - * h abc* +de/ 2 (step 7)

- * h abc * + de/h 3

- * ~~h~~ abc * + de/h* 2

- ~~*~~ abc * + de/h* - 1

~~-~~

o/p string: abc * + de/h* -

Rank: ~~0~~1

(3) Conversion of Polish expression to code

The following are some of the assembler instructions which are available in Assembly language.

1. **LOD a** – Load the value of a variable a in the accumulator content leaves the contents of a unchanged.
2. **STO a** – Stores the value of the accumulator in a word of memory denoted by a. The contents of accumulator remain unchanged.
3. **ADD a** – Adds the value of variable a to the value of the accumulator and leaves the result in the accumulator. The contents of a remain unchanged.

(3) Conversion of Polish expression to code

4. **SUB a** – The value of variable a is subtracted from the value of the accumulator and leaves the result in the accumulator. The contents of a remain unchanged.
5. **MUL a** - The value of variable a is multiply with the value of the accumulator and result is stored in the accumulator. The contents of a remain unchanged.
6. **DIV a** - The value of the accumulator is divided by the value of variable a and result is stored in the accumulator. The contents of a remain unchanged.
7. **JMP b** – This is an unconditional branching instruction. The next instruction to be executed is located at a location denoted by label b.
8. **BRN b** – This is a conditional branching instruction. The location of the next instruction to be performed is given **by label b if the accumulator content is negative**; otherwise, the instruction following the BRN instruction is next.

(3) Conversion of Polish expression to code

```
X = 10
Y = 20
RES = 10 + 20 = 30

LOD X      (A = 10)
ADD Y      (A = A + Y = 10 + 20 = 30)
STO RES    (A to RES)
```

```
X = 10
Y = 20
RES = 10 - 20 = -10

LOD X
SUB Y
STO RES
```

```
S1
S2   BRN S5
S3
S4
S5
S6
```

```
X = 10
Y = 20
RES = 10 * 20 = 200

LOD X
MUL Y
STO RES
```

```
X = 20
Y = 10
RES = 20 / 10 = 2

LOD X
DIV Y
STO RES
```

ASSEMBLY_CODE

Algorithm : ASSEMBLY_CODE. Given a string POLISH representing a reverse Polish expression (which contains the four basic arithmetic operators and single-letter variables) equivalent to some well-formed infix expressions, this algorithm translates the string POLISH to assembly language instructions as previously specified. The algorithm uses a **stack S** as usual. The integer **variable I** is associated with the generation of an intermediate result. The string variable **OPCODE** contains the operation code which corresponds to the current operation being processed.

```
1.  [Initialize]
            TOP <- I <- 0
2.  [Process all symbols]
            Repeat thru step 4 for J = 1,2,..... LENGTH(POLISH)
3.  [Obtain current input symbol]
            NEXT <- SUB( POLISH, J, 1)
4.  [Determine the type of NEXT]
            If 'A' <= NEXT and NEXT <= 'Z'
            then      Call PUSH(S, TOP , NEXT) (Push variable on stack)
```

ASSEMBLY_CODE

```
        else        Select case(NEXT) (process current operator)
                    Case '+':
                    OPCODE <- 'ADD□'
                    Case '-':
                    OPCODE <- 'SUB□'
                    Case '*':
                    OPCODE <- 'MUL□'
                    Case '/':
                    OPCODE <- 'DIV□'
        RIGHT <- POP(S,TOP) (unstack two operands)
        LEFT <- POP(S,TOP)
        Write('LOD□' ○ LEFT)         (output load instruction)
        Write(OPCODE ○ RIGHT)        (output arithmetic instruction)
        I <- I + 1            (obtain temporary storage index)
        TEMP <- 'T' ○ I
        Write('STO□' ○ TEMP) (output temporary store instructions)
        Call PUSH(S,TOP,TEMP) (stack intermediate result)
5. [Finished]
        Exit
```

DIGITAL LEARNING CONTENT

ASSEMBLY_CODE

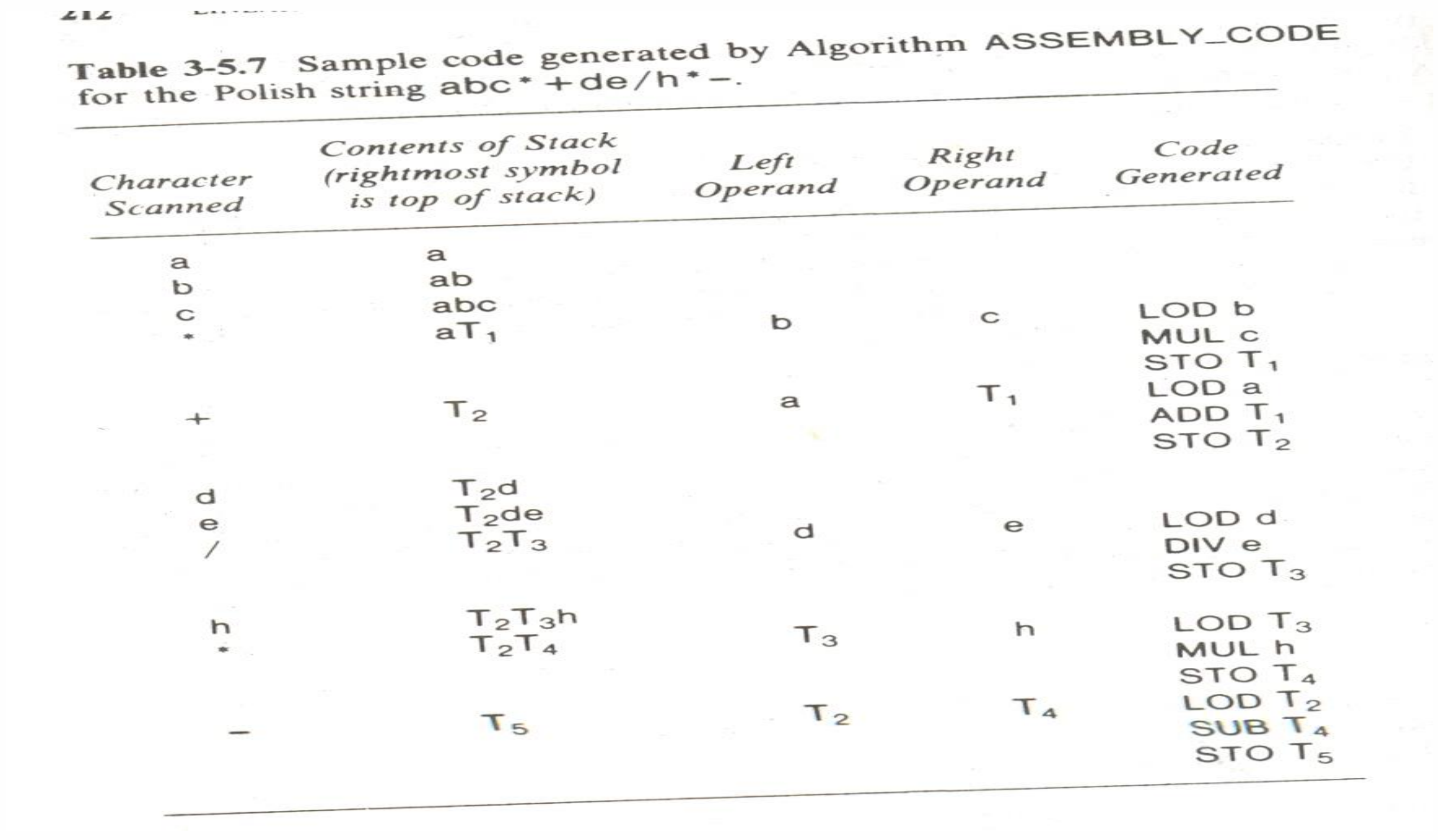


Table 3-5.7 Sample code generated by Algorithm ASSEMBLY_CODE for the Polish string abc* + de/h* −.

Character Scanned	*Contents of Stack (rightmost symbol is top of stack)*	*Left Operand*	*Right Operand*	*Code Generated*
a	a			
b	ab			
c	abc			
*	aT_1	b	c	LOD b MUL c STO T_1
+	T_2	a	T_1	LOD a ADD T_1 STO T_2
d	T_2d			
e	T_2de			
/	T_2T_3	d	e	LOD d DIV e STO T_3
h	T_2T_3h			
*	T_2T_4	T_3	h	LOD T_3 MUL h STO T_4
−	T_5	T_2	T_4	LOD T_2 SUB T_4 STO T_5

```
2.  [Process all symbols]
•          Repeat thru step 4 for J = 1,2,….. LENGTH(POLISH)
3.  [Obtain current input symbol]
•          NEXT <- SUB( POLISH, J, 1)
•   4.   [Determine the type of NEXT]
•          If ‘A’ <= NEXT and NEXT <= ‘Z’
•          then      Call PUSH(S, TOP , NEXT) (Push variable on stack)
•   else   Select case(NEXT) (process current operator)
•                    Case ‘+’:
•                    OPCODE <- ‘ADD□’
•                    Case ‘-’:
•                    OPCODE <- ‘SUB□’
•                    Case ‘*’:
•                    OPCODE <- ‘MUL□’
•                    Case ‘/’:
•                    OPCODE <- ‘DIV□’
•          RIGHT <- POP(S,TOP) (unstack two operands)
•          LEFT <- POP(S,TOP)
•          Write(‘LOD□’ ○ LEFT)      (output load instruction)
•          Write(OPCODE ○ RIGHT)   (output arithmetic instruction)
•          I <- I + 1          (obtain temporary storage index)
•          TEMP <- ‘T’ ○ I
•          Write(‘STO□’ ○ TEMP) (output temporary store instructions)
•          Call PUSH(S,TOP,TEMP) (stack intermediate result)
•   5. [Finished]
•          Exit
```

Assembly-Code Algorithm

a b c * + d e / h * −

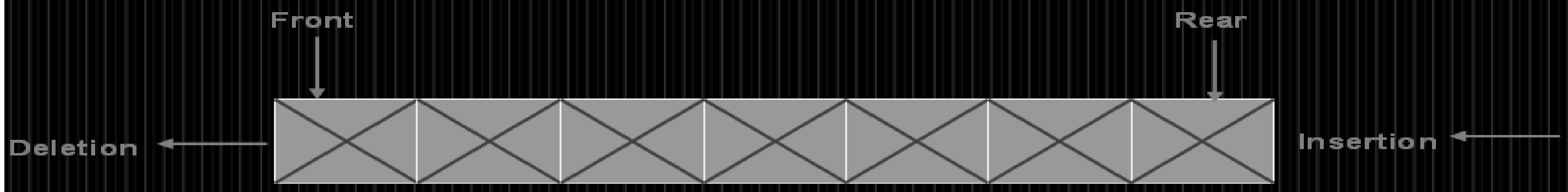
Character Scanned	Stack Content	Left operand	Right operand	Code Generated
a	a			
b	ab			
c	abc			
*	OPCode= MUL			
	ab~~c~~		c	
	a~~b~~	b		LOD b MUL C STO T1
	I=1 TEMP=T1			
	a T1			
+	opcode = ADD			
	a ~~T1~~		T1	
	~~a~~	a		LOD a ADD T1 STO T2
	I=2 TEMP=T2			
	T2			
d	T2 d			
e	T2 d e			
/	opcode = DIV	d	e	LOD d
	T2 ~~d~~ ~~e~~			DIV e
	TEMP=T3			STO T3
	T2 T3			
h	T2 T3 h			
*	Opcode = MUL	T3	h	Lod T3
	T2 ~~T3~~ ~~h~~			MUL h
	TEMP=T4			STO T4
	T2 T4			
−	opcode = SUB	T2	T4	Lod T2
	~~T2~~ ~~T4~~			SUB T2
	T5			STO T5

Queue

1. Introduction to queue,
2. Algorithms and implementation of
3. simple queue,
4. Circular queue,
5. Double ended queue,
6. Priority queue.

Queue

- A queue is a linear list of elements in which deletions can take place only at one end, called the front and insertions can take place only at the other end, called the rear.
- Queues are also called first-in-first-out (FIFO) lists OR First-Come-First-Served (FCFS).



Representation of a queue

Terminologies in Queue

- The difference between stacks and queues is in removing.
- In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

1. REAR : Pointer representing the tail end of queue
2. FRONT : Pointer representing the front end of queue

\

\

Real examples of Queue:

- Line of people at ticket window
- Line of vehicles at railway crossing
- Queue of vehicles at Toll Booth, Voting Booth etc.

Types of Queue:

1. Simple Queue
2. Circular Queue
3. Double ended Queue
4. Priority Queue

How Queue works

Example: Consider the following queue (linear queue).

Rear = 4 and Front = 1 and N = 7

10	50	30	40			
1	2	3	4	5	6	7

(1) Insert 20. Now Rear = 5 and Front = 1

10	50	30	40	20		
1	2	3	4	5	6	7

(2) Delete Front Element. Now Rear = 5 and Front = 2

	50	30	40	20		
1	2	3	4	5	6	7

(3) Delete Front Element. Now Rear = 5 and Front = 3

		30	40	20		
1	2	3	4	5	6	7

(4) Insert 60. Now Rear = 6 and Front = 3

		30	40	20	60	
1	2	3	4	5	6	7

Algorithm to insert an element in a simple queue

```
QINSERT(Q,F,N,R,Y)
Step 1 : [Check for an overflow]
          if R>=N then
                    write('Overflow')
          return
Step 2 : [increment rear pointer]
          R<-R+1
Step 3 : [insert an element at rear position]
          Q[R]<-Y
Step 4 : [is front pointer properly set ? ]
          if F=0 then
                    F<-1
          return
```

Image source : Youtube video

Algorithm to delete an element in a simple queue

QDELETE(Q,F,R,N)

```
Step 1 : [Check for an underflow]
        if F=0then
                write('Underflow')
        return
Step 2 : [delete an element]
        Y<-Q[F]
Step 3 : [is the queue empty ?]
        if F=R then
                F<-R<-0
        else
                F<-F+1
Step 4 : [Return deleted element ]
        return (Y)
```

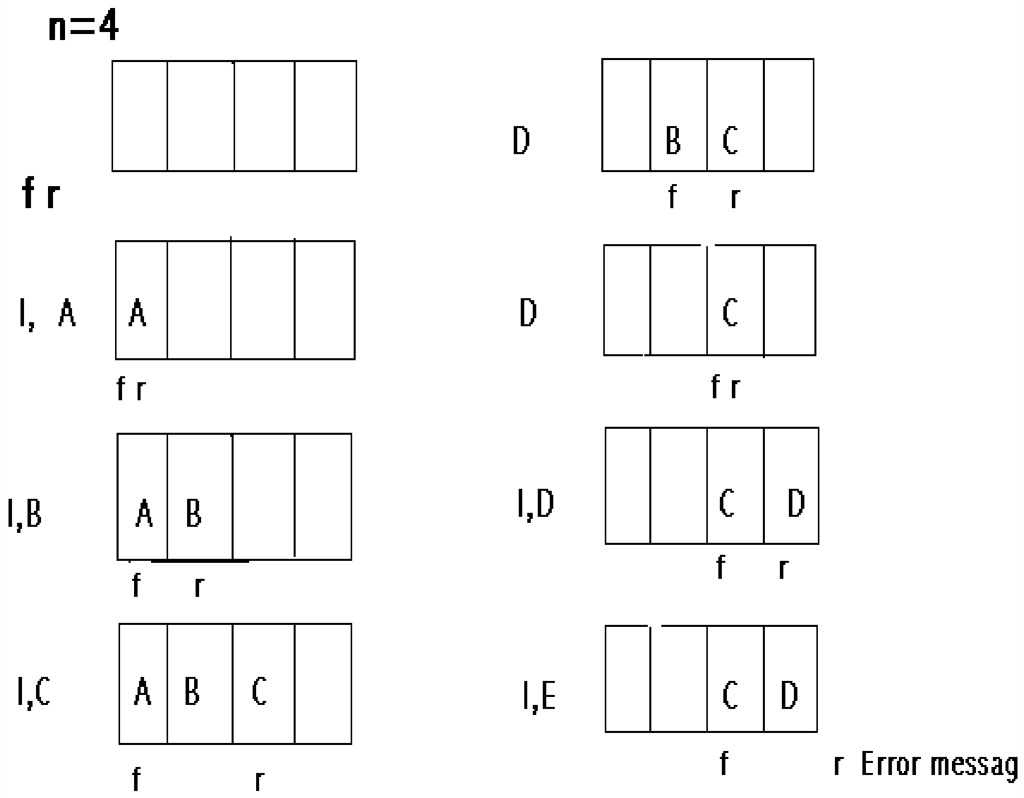


Image source : Google

Drawback of Linear Queue

- Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the Queue’s rear most position.

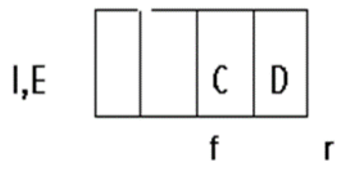


Image source : Youtube video

```
#include <stdio.h>
# define SIZE 10
void InsertQueue();
void DeleteQueue();
void show();
int queue[SIZE];
int Rear = - 1;
int Front = - 1;
```

```
main()
{   int ch;
while (1)
{
            printf("1.Insert Operation\n");
            printf("2.Deletion Operation\n");
            printf("3.Display the Queue\n");
            printf("4.Exit\n");
            printf("Enter your choice of operations : ");
            scanf("%d", &ch);
            switch (ch)
            {       case 1:
                          InsertQueue();
                          break;
                    case 2:
                          DeleteQueue();
                          break;
                     case 3:
                          show();
                                     break;
                    case 4:        exit(0);
                   default:
                          printf("Incorrect choice \n");
            }
}
}
```

```
void InsertQueue()
{    int item;
     if (Rear == SIZE - 1)
               printf("Overflow \n");
    else
   {
               if (Front == - 1)
                  Front = 0;
               printf("Element to be inserted in the Queue\n : ");
               scanf("%d", &item);
               Rear = Rear + 1;
               queue[Rear] = item;
   }
}
```

```
QINSERT(Q,F,N,R,Y)
Step 1 : [Check for an overflow]
          if R>=N then
                    write('Overflow')
          return
Step 2 : [increment rear pointer]
          R<-R+1
Step 3 : [insert an element at rear position]
          Q[R]<-Y
Step 4 : [is front pointer properly set ? ]
          if F=0 then
                    F<-1
          return
```

```
void DeleteQueue()
{
        if (Front == - 1 || Front >
Rear)
        {
                printf("Underflow
\n");      return ;
        }
        else
        {      printf("Element
deleted from the Queue: %d\n",
queue[Front]);
        Front = Front + 1;
        }
}
```

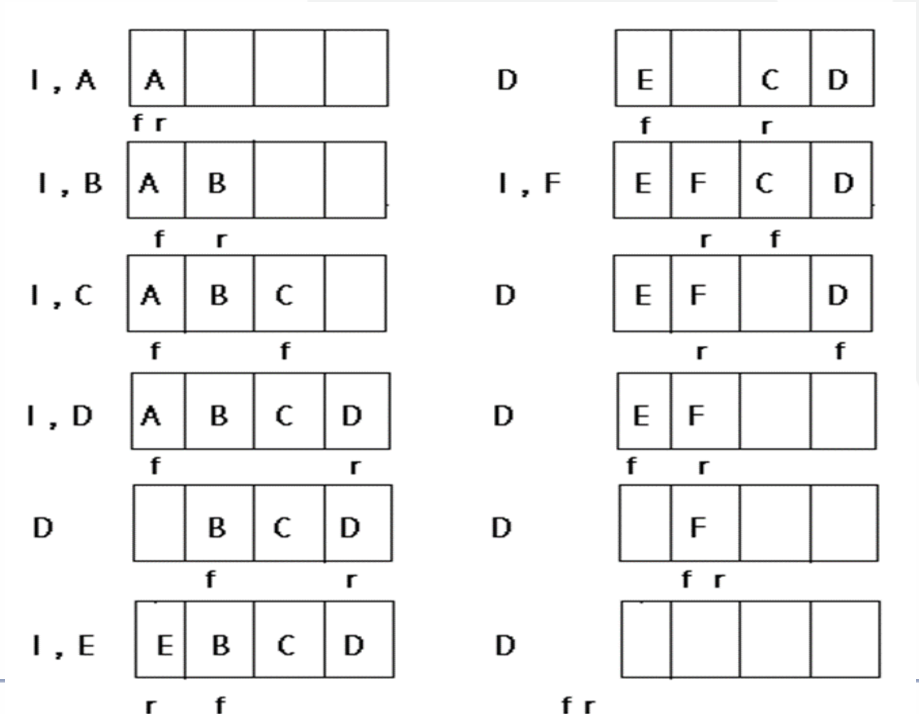
QDELETE(Q,F,R,N)

```
Step 1 : [Check for an underflow]
        if F=0then
                write('Underflow')
        return
Step 2 : [delete an element]
        Y<-Q[F]
Step 3 : [is the queue empty ?]
        if F=R then
                F<-R<-0
        else
                F<-F+1
Step 4 : [Return deleted element ]
        return (Y)
```

```
void show()
{
        if (Front == - 1)
                printf("Empty Queue \n");
        else
        {
                printf("Queue: \n");
                for (int i = Front; i <= Rear; i++)
        printf("%d ", queue[i]);
                printf("\n");
        }
}
```

Circular Queue

- The queue in which elements are arranged in a circular fashion
- In a circular queue any element is accessible from any position but only in a forward manner.



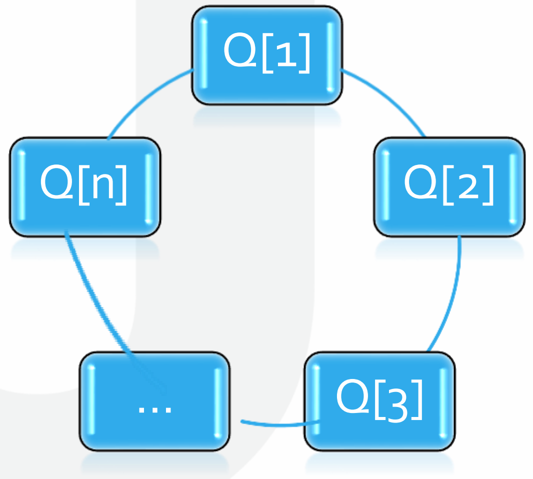
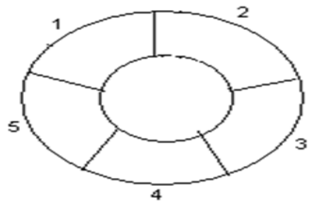


Image source : Google

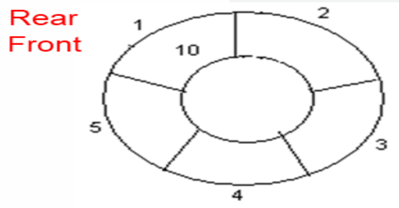
Algorithm to insert an element in a circular queue

Example: Consider the following circular queue with N = 5.

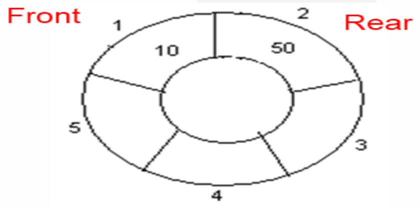
1. Initially, Rear = 0, Front = 0.



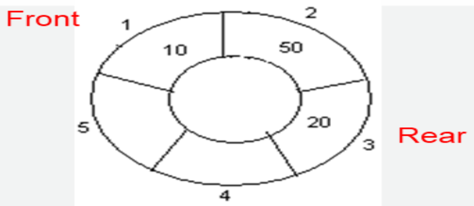
2. Insert 10, Rear = 1, Front = 1.



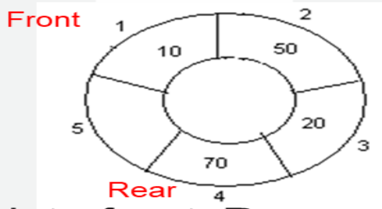
3. Insert 50, Rear = 2, Front = 1.



4. Insert 20, Rear = 3, Front = 1.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.

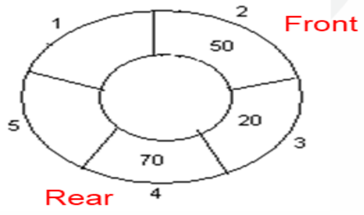
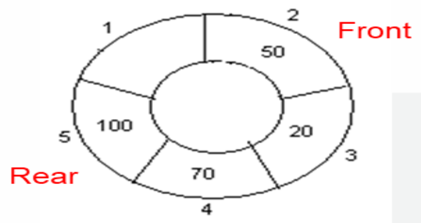


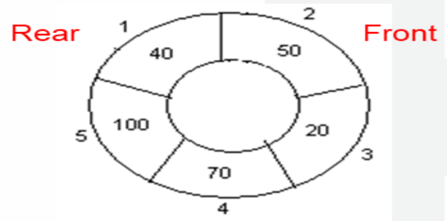
Image source : Google

Algorithm to insert an element in a circular queue

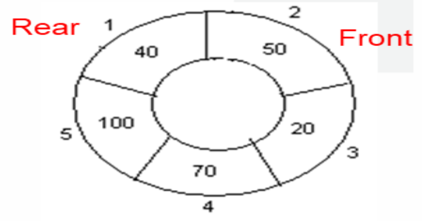
7. Insert 100, Rear = 5, Front = 2.



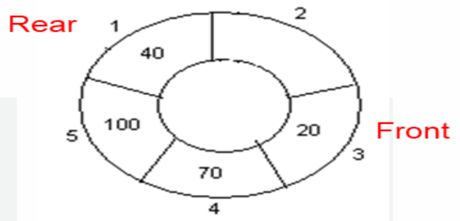
8. Insert 40, Rear = 1, Front = 2.



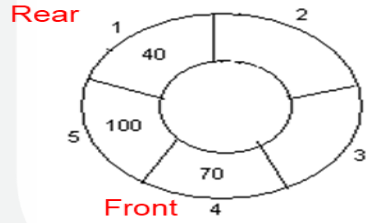
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear, so Queue overflow.



10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.

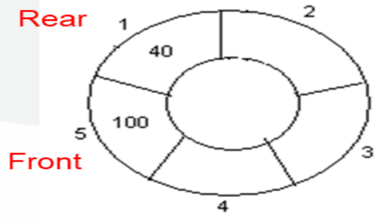


Image source : Google

Algorithm to insert an element in a circular queue

```
QINSERT(F,R,N,Q,Y)
Step 1: [Reset rear pointer ?]
        if R=N
                then R<-1
        Else
                R<-R+1
Step 2 : [Check for overflow]
        if R=F
                then write('OVERFLOW')
        Return
Step 3 : [Insert element]
        Q[R]<-Y
Step 4 : [Is front pointer properly set?]
        if F=0
                then F<-1
        Return
```

Image source : Google

Algorithm to delete an element in a circular queue

```
QDELETE(F,R,Q,N)
Step 1 : [Underflow ?]
        if F=0
                then write('Underflow ... No
                 elements to delete')
        Return
Step 2 : [Delete an element]
        Y<-Q[F]
Step 3 : [Queue empty]
        if R=F
                then R<-F<-0
        return (Y)
Step 4 : [Increment front pointer]
        if F=N
                then f<-1
        else
                F<-F+1
        Return (Y)
```

Image source : Google

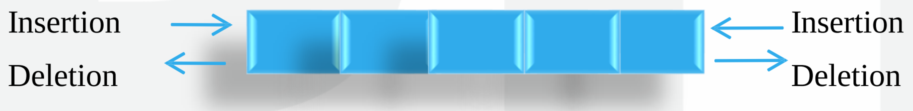
Parul® University

DIGITAL LEARNING CONTENT

Double Ended queue (D-queue)

- A D-queue is a linear list in which elements can be inserted or deleted from either end of a queue.
- **There are two variations in D-queue:**

1. An input restricted D-queue
2. An output restricted D-queue



- An **Input restricted D-queue** allows insertion only at one end of the queue but deletion at both the ends of a queue.

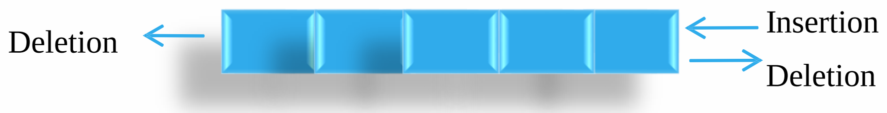


Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Double Ended queue (D-queue)

- An **output restricted D-queue** allows deletion only at one end of the queue but insertion at both the ends of a queue.

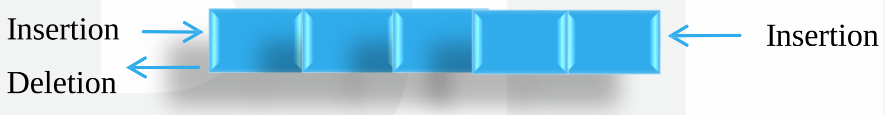


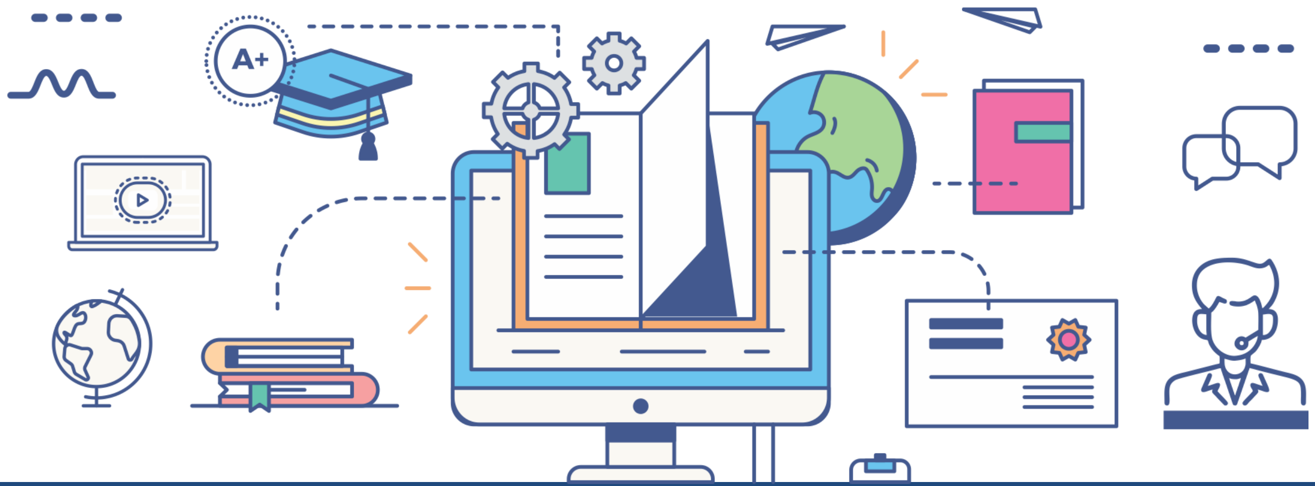
Image source : Google

Priority Queue

- A queue in which we are able to insert items or remove items from any position based on someproperty is (based on the priority assigned to the tasks) is knows as Priority Queue.
- A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

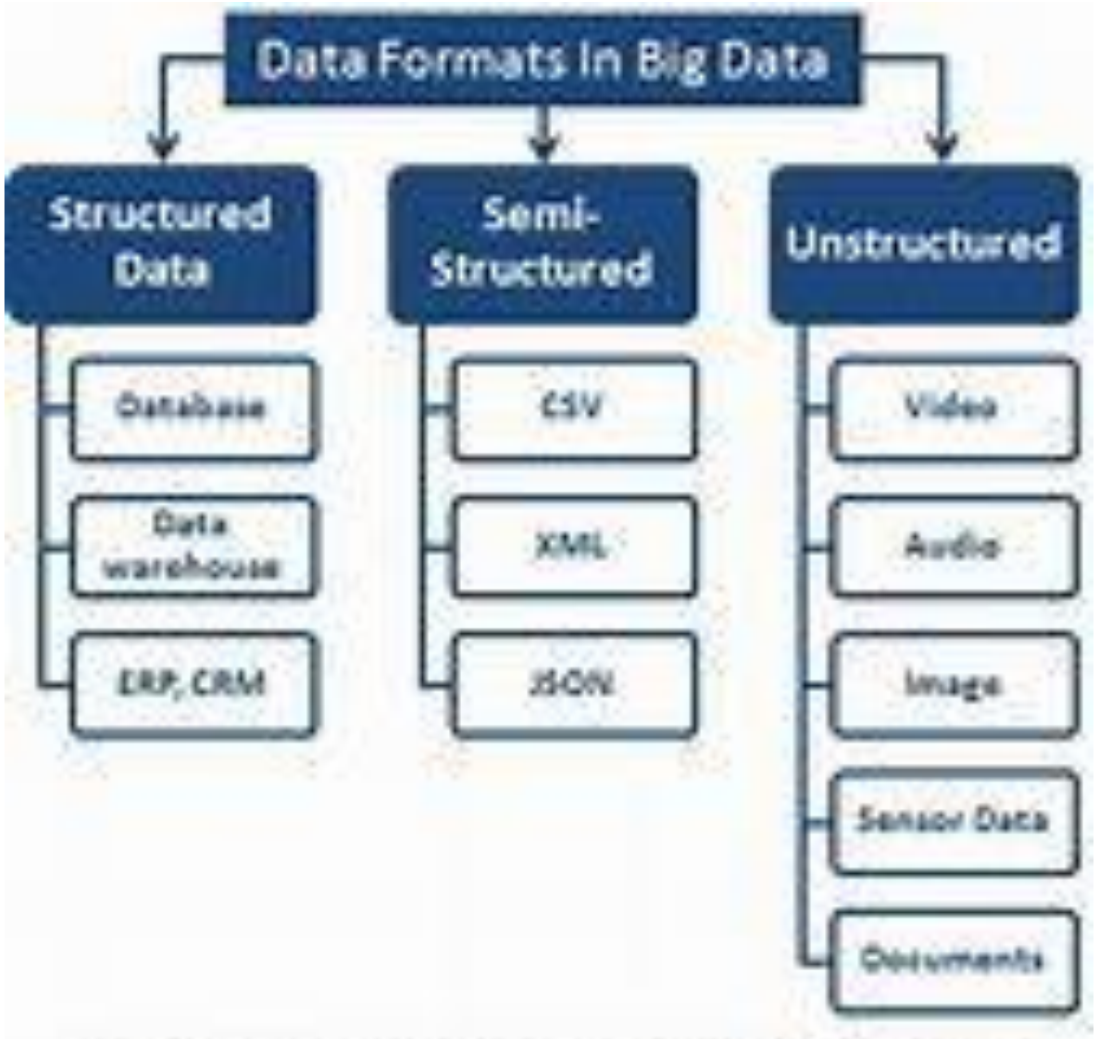
1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

DIGITAL LEARNING CONTENT



Parul® University

www.paruluniversity.ac.in


Data Formats In Big Data
Structured Data
Semi-Structured
Unstructured
Database
Data warehouse
ERP, CRM
CSV
XML
JSON
Video
Audio
Image
Sensor Data
Documents