Data Structure

Prof. Vijya Tulsani, Assistant Professor
IT and Computer Science

Parul® University

DIGITAL LEARNING CONTENT

CHAPTER-2

Linear Data Structure

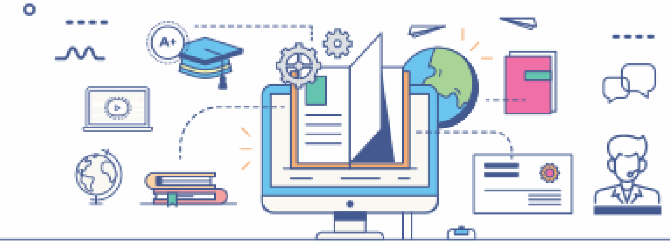
Arrays

- It is a most commonly used data structure
- If you want to store group of data together in one place , than array is data structure that is most commonly used.
- Arrays enable us to arrange more than one element, and so it is known as composite data structure.

DIGITAL LEARNING CONTENT

Arrays

- "Arrays are finite and ordered collection of Homogeneous data elements."
- Arrays are finite because it contains only limited number of element.
- Arrays are ordered because all the elements are stored one by one in the computer memory.
- Arrays are homogeneous because all the elements of an array are of the same type only.



Derived Data Type

- Simply, declaration of array is as follows:

  int a[5]
- Where int specifies the data type or type of elements arrays stores.
- "a" is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

1st ele	2nd ele	3rd ele	4th ele	5th ele	
4	8	7	9	10	Value of element
a[0]	a[1]	a[2]	a[3]	a[4]	

a[0] = 1026 a[1] = 1028

Arrays

- • An individual element of the array can be accessed by specifying name of the array, following by index value inside square brackets.
- • The first element of the array has index value [0]. It means the first element and last element will be specified as: my_arr [0] & my_arr [19] = 20 Respectively.
- • Following equation will give the number of elements that can be stored in an array, that is the size of array or its length.

(Ubound – lbound) +1

- For the above array it would be
- (19-0)+1=20, where 0 is lower bound of array and 19 is upper bound of array.

Image source : Google

Array - limitations

- It is wastage of memory sometimes
- It cannot work with elements of different data type.
- In arrays task of insertion and deletion is not easy
- It is also shortage of Memory- if we don't know the size of memory in advance

Image source : Youtube video

Operations – Arrays

- Insertion of new array element
- Deletion of required array element
- Modification of an existing array element
- Merging of arrays in a single array

Image source : Google

Array - Applications

- Arrays are used to implement matrices, as well as other kinds of rectangular tables.
- Databases, small as well as large, consisting of one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.

Image source : Google

Types of Array

1. **One Dimensional**

- One dimensional array is having only one subscript also called an index value to access and store any element in the array.
- e.g. int s5[100]

2. **Two Dimensional**

- One dimensional array is having two subscripts also called an index value to access and store any element in the array.
- e.g. int s5[30][40]

Image source : Google

Address Calculation in single (one) Dimension Array:

int a[5] = {10,20,30,40,50};

The memory representation of this array is

Actual address in memory =>	1000	1004	1008	1012	1016
Array Elements =>	10	20	30	40	50
Array Subscript/index =>	0	1	2	3	4

Image source : Google

Address Calculation in single (one) Dimension Array:

- Array of an element of an array say “A[I]” is calculated using the following formula:

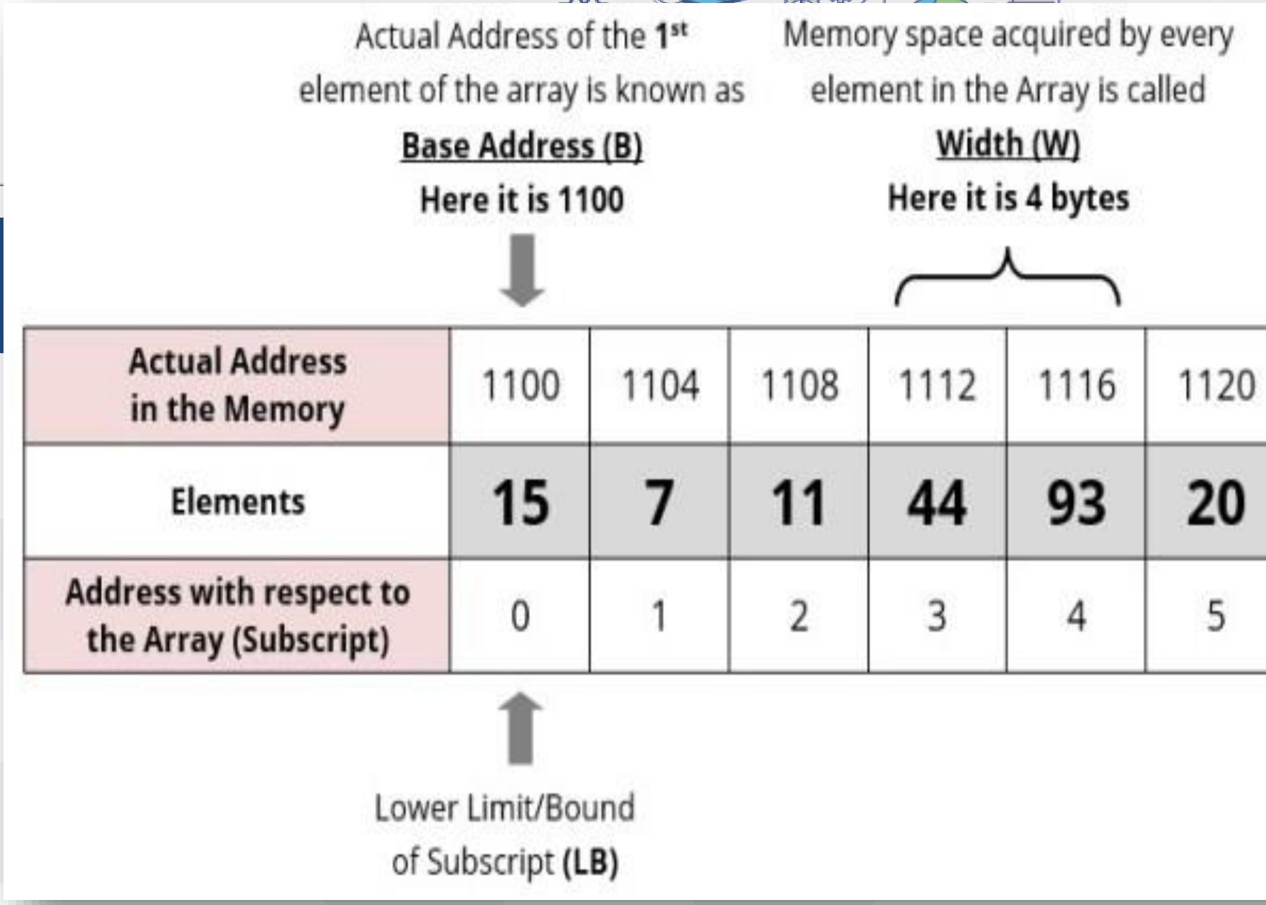
Address of A [I] = B + [W * (I – LB)]

This is the Important formula to calculate address of array.

Where,
B = Base address.
W = Storage Size of one element stored in the array (in byte).
I = Subscript of an element whose address is to be found.
LB = Lower limit or Lower Bound of subscript, if not specified assume 0 (zero).

Image source : Google

Example



Actual Address of the 1st element of the array is known as **Base Address (B)** **Here it is 1100**

Memory space acquired by every element in the Array is called **Width (W)** **Here it is 4 bytes**

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

- A [4] = B + W * (I – LB)

= 1100 + [4*(4-0)]

=1100 + 16

=1116

Image source : Google

2-D array

- Also defined as an array of arrays.
- This type of 2D arrays is organized as matrices which can be represented as the collection of rows and columns.
- These 2D arrays can be created to implement a relational database lookalike data structure.
- 2D arrays are providing ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

2-D array

- Just like 1D array, we must have tospecify the data type, the name, and the size of the array.
- In this array size of the array is described as the number of rows and number of columns.
- Syntax:

```
datatype array_name_2D[rows][columns];
```

- Example:

```
int twodimen_2D[6][6];
```

```
a: array_2D [ 1 .. 6, 1 .. 5] of integer
              lb1   ub1lb2   ub2
```

Image source : Google

2-D array

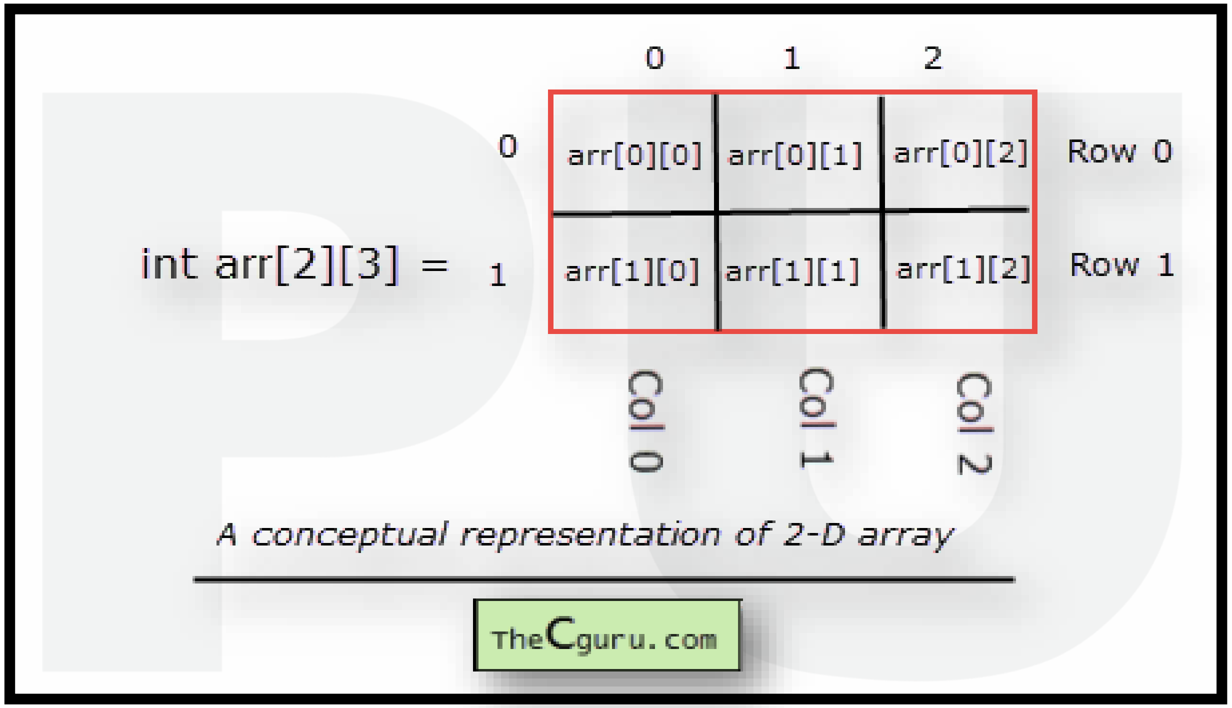


Image source : Google

2-D array

a: array [1 .. 5, 1 .. 4] of integer

lb1 ub1 lb2 ub2

- **Two type of representation:**

1. **Row major** – Representation of array is row by row
2. **Column major** – Representation of array is column by column

- **Row major:** Not Much Important

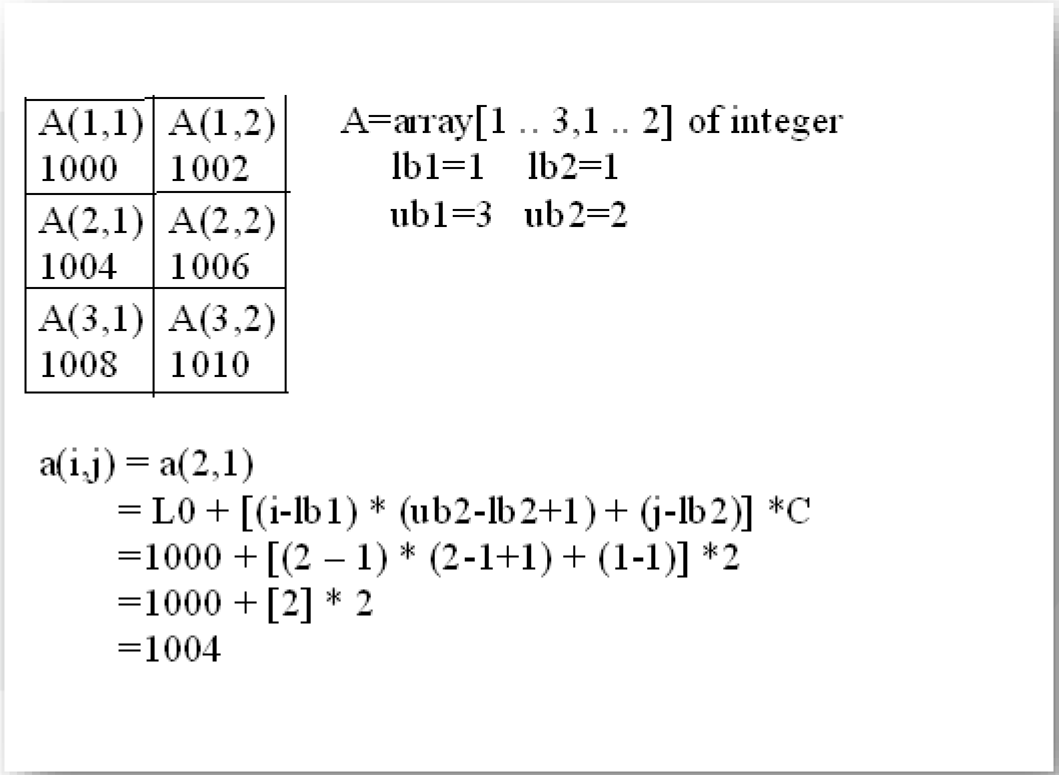
  a [i,j] = SA + [(i-lb1) * (ub2-lb2+1) + (j-lb2)] * C

- **column major:**

  a [i,j] = SA + [(j-lb1) * (ub2-lb2+1) + (i-lb2)] * C

Image source : Google

Example – 2D-array



A(1,1) 1000	A(1,2) 1002
A(2,1) 1004	A(2,2) 1006
A(3,1) 1008	A(3,2) 1010

A=array[1 .. 3,1 .. 2] of integer
lb1=1 lb2=1
ub1=3 ub2=2

a(i,j) = a(2,1)
= L0 + [(i-lb1) * (ub2-lb2+1) + (j-lb2)] *C
=1000 + [(2 – 1) * (2-1+1) + (1-1)] *2
=1000 + [2] * 2
=1004

Image source : Google

Sparse Matrix

- A matrix is called a two-dimensional data object made up of m rows and n columns, therefore having total m x n values.
- If in my matrix most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- Here instead of storing zeroes with non-zero elements, we only store non-zero elements.
- That means we will store non-zero elements with triples- (Row, Column, value).

Sparse Matrix

Why to use Sparse Matrix instead of simple matrix ?

- Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Image source : Google

Sparse Matrix

- Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

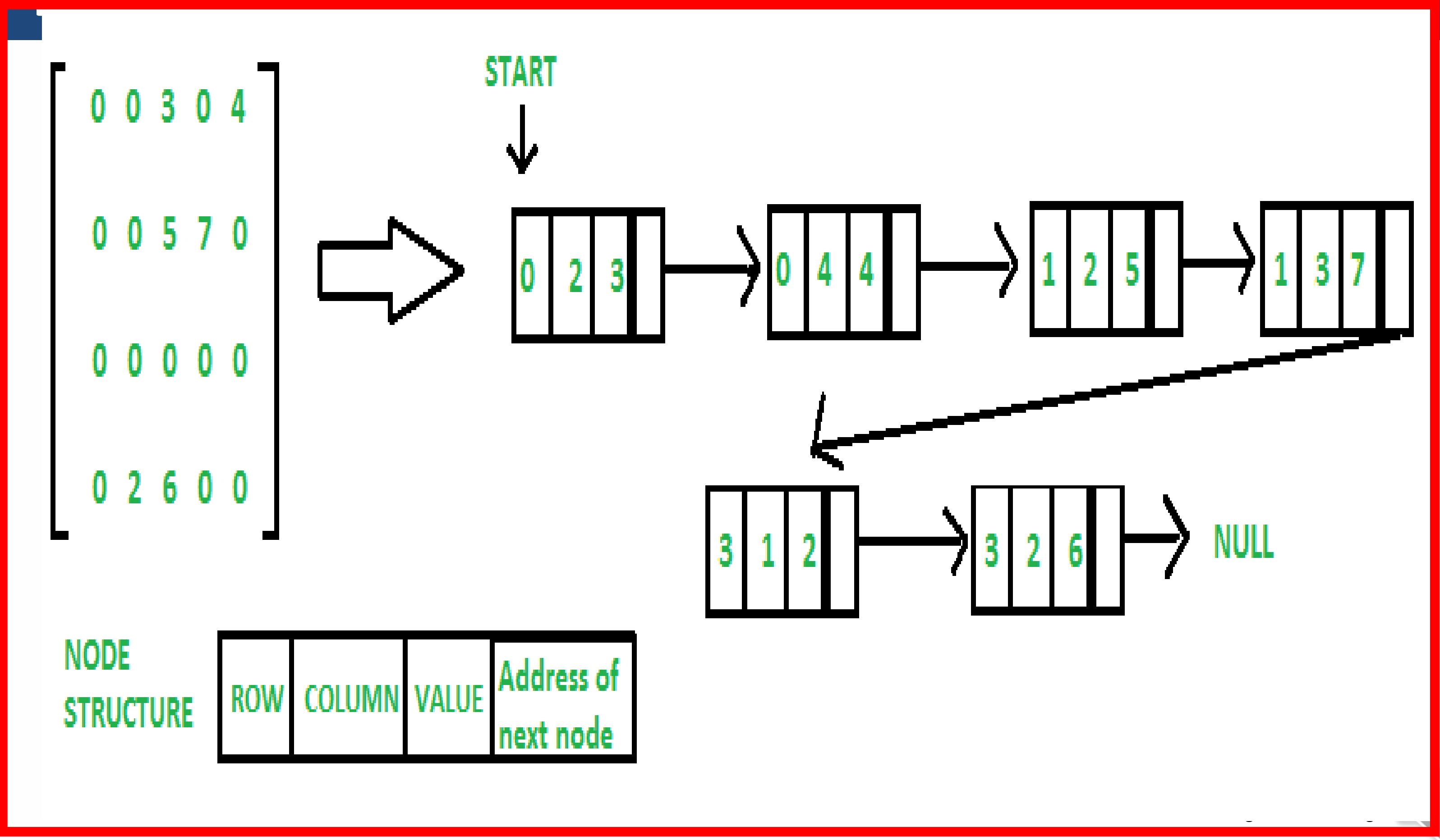
Image source : Google

Example

- **Array representation**
- 2D array will used to represent a sparse matrix in which there are three rows named as
- Row: Index of row, in which non-zero element is located
- Column: Index of column, in which non-zero element is located
- Value: Value of the non zero element located at index position – (row,column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Example


0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
START
0 2 3
0 4 4
1 2 5
1 3 7
3 1 2
3 2 6
NULL
NODE STRUCTURE
ROW
COLUMN
VALUE
Address of next node

Example

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

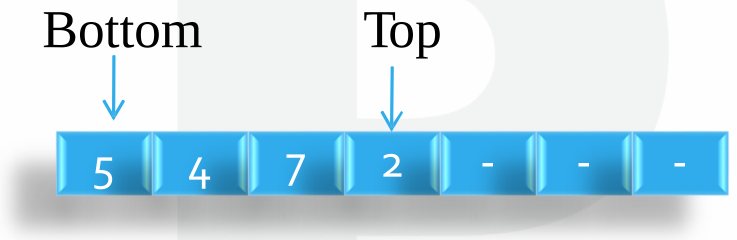
Next node: Address of the next node

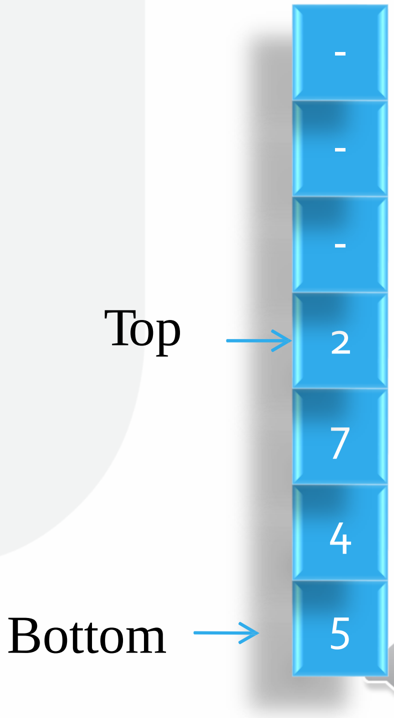
Image source : Google

Stack

- Stacks are also an ordered collection of elements like array, but having a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Stack is sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Representation of stack in memory





Parul® University

DIGITAL LEARNING CONTENT

Stack- Examples

- Arrangements of disk plates in a rack
- Playing cards is example of stack
- Social media – Instagram posts

\

\

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. BOTTOM

- The pointer at the last position of stack is called as BOTTOM.
- It is least commonly accessible element of the stack.

3. 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.

4. 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.

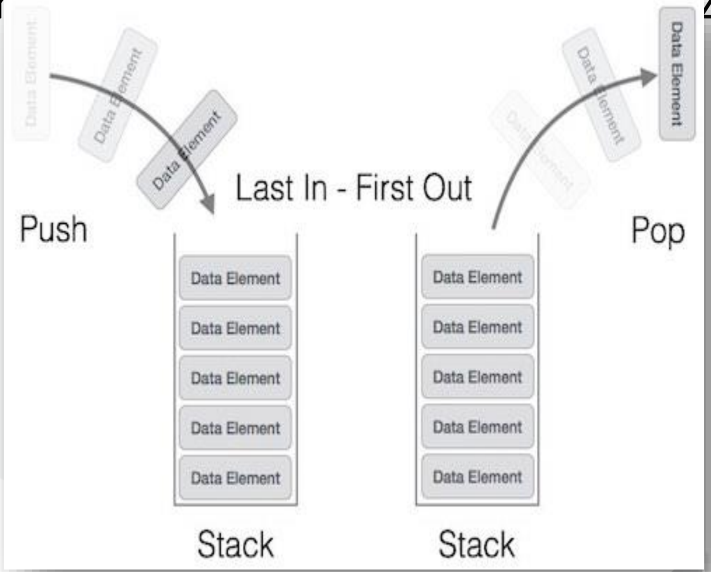
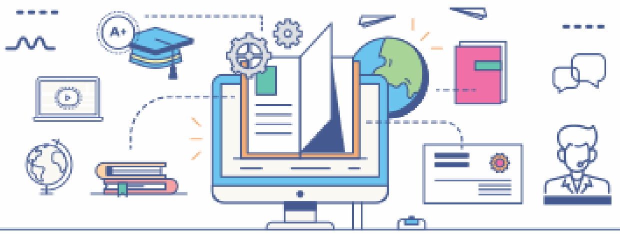


Image source : Google



Applications of Stack

- Recursion
- Expression evaluations and conversions
- Parsing
- Browsers
- Editors
- Tree Traversals

Pros of Stack

Just read these

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.
So, it will give control over how memory is allocated and deallocated
3. Stack frees you from the burden of remembering to cleanup(read delete) the object
4. 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.

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 I th element from stack data structure.
4. **CHANGE :** To change I th element from stack data structure.

Algorithms – PUSH() Practice as much as you can

- 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

Algorithms – POP()

- POP(Stack, TOP, Item)
- 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

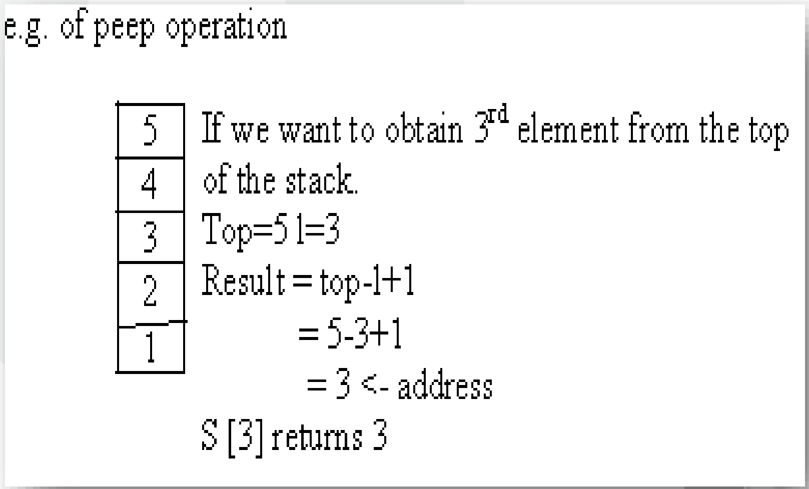
Parul® University

DIGITAL LEARNING CONTENT

Algorithms – PEEP()

- PEEP(Stack, TOP, I,item)
- 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
```



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

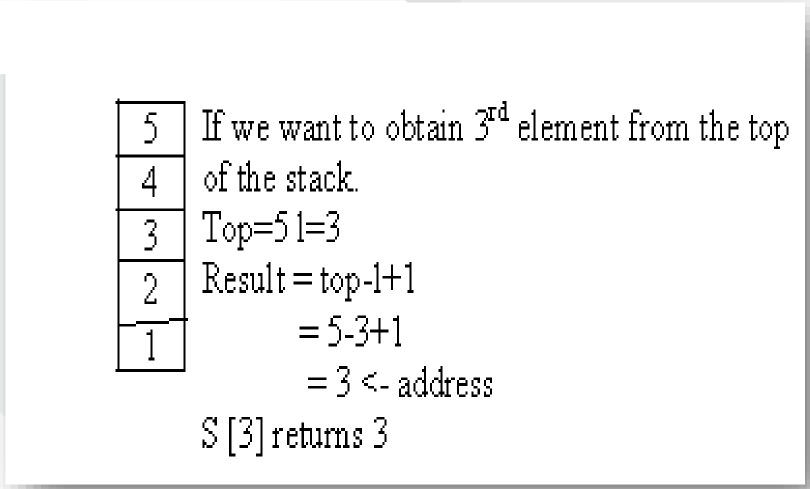
Parul® University

DIGITAL LEARNING CONTENT

Algorithms – CHANGE()

- CHANGE (S, TOP, l, X):
- 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 l=3

Result = top-l+1

= 5-3+1

= 3 <- address

S [3] returns 3

Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Representation of Stack with example

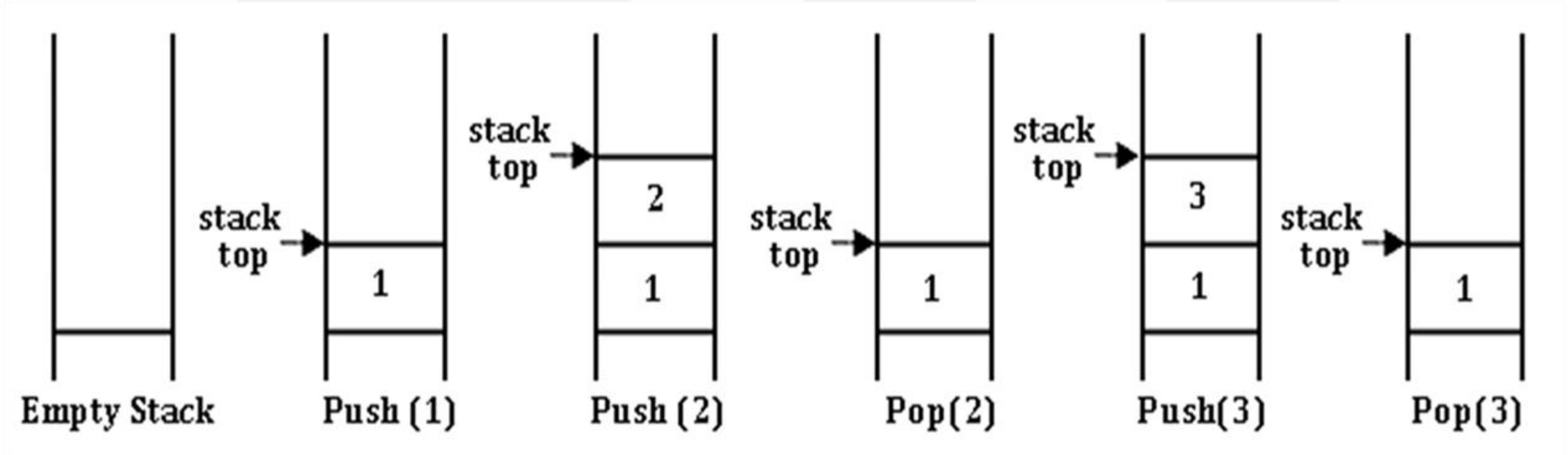


Image source : Google

Recursion

- Recursion is the technique of defining a set or process in term of itself.
- E.g. The factorial function can be recursively defined as follows:

```
int factorial(int n)
{
        int fact;
        if(n==1)
                return(1);
        else
                fact = n*factorial(n-1);
                return(fact);
}
```

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

- Types of expressions:
- Infix expression, for e.g. a+b
- Postfix expression, for e.g. ab+
- Prefix expression, for e.g. +ab
- Rank (operand) = +1
- Rank (operator) = -1
- Valid Rank (expression) = 1

Polish expressions and their compilations

Algorithms of expressions

- Conversion of infix to postfix (un-parenthesized)
- Conversion of infix to postfix (parenthesized)
- Conversion of Polish (postfix) expression to code

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 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

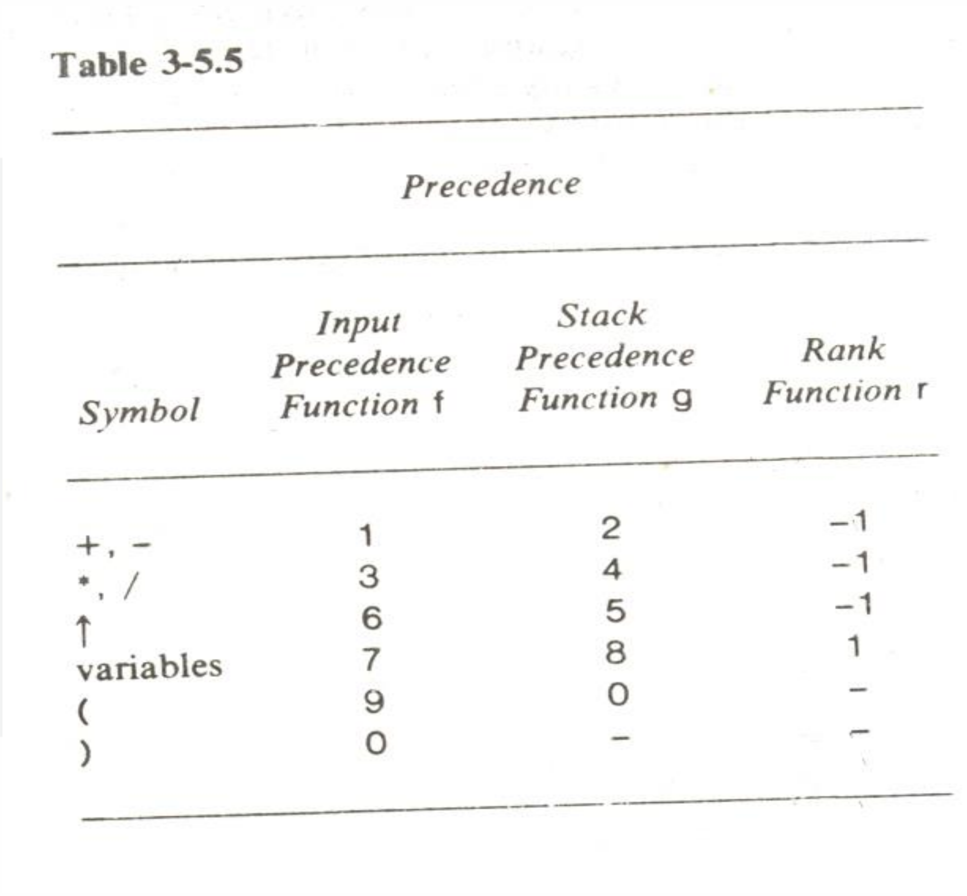


Table 3-5.5

	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	−	−

Translation of infix to polish

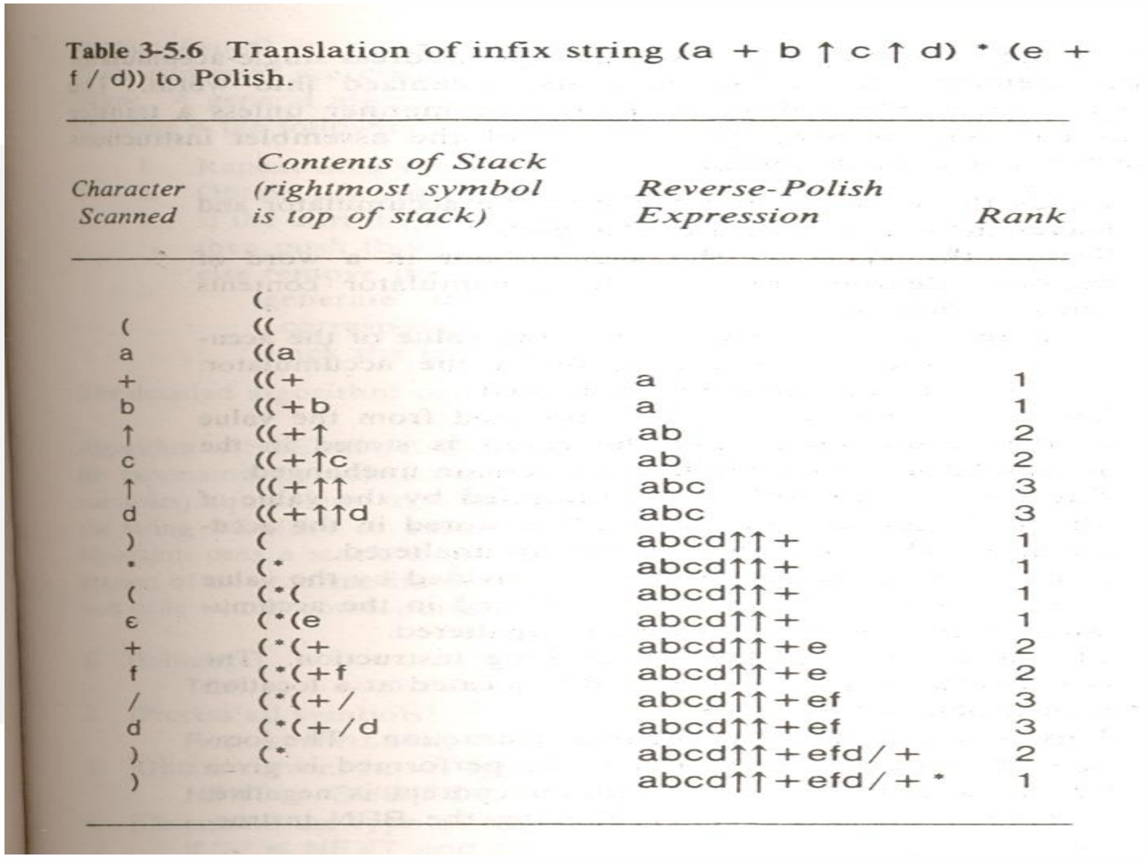


Table 3-5.6 Translation of infix string (a + b ↑ c ↑ d) * (e + f / d)) to Polish.

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
ε	(*(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

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

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

Conversion of infix to postfix (un parenthesized)

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 ‘#’

```
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)
```

Parul® University

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) 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.

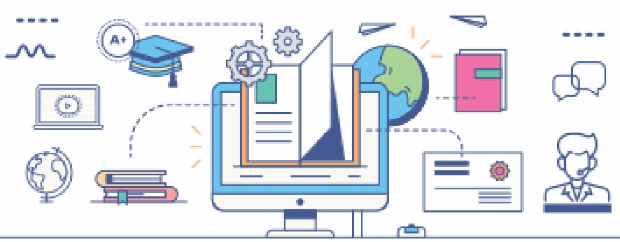
(3) Conversion of Polish expression to code

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.

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
```

ASSEMBLY_CODE

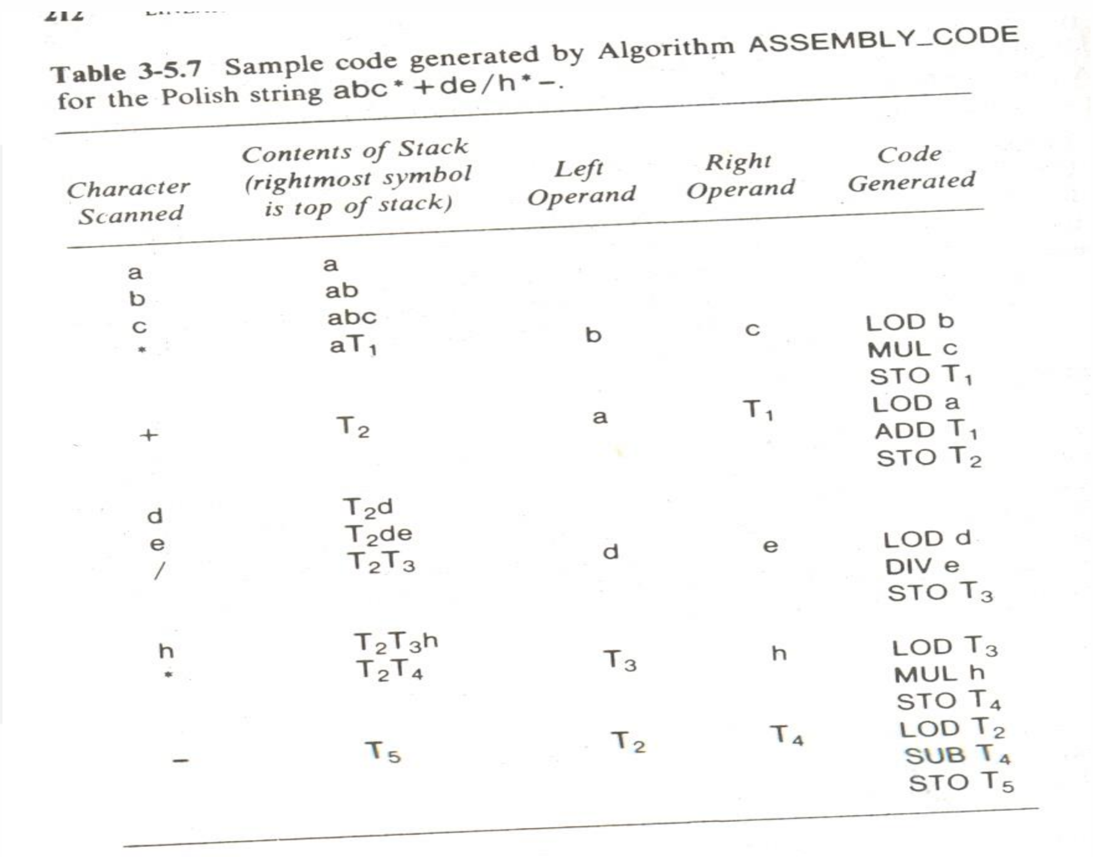
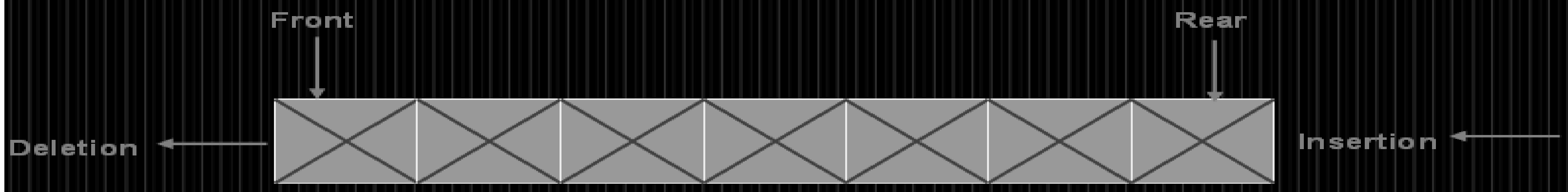


Table 3-5.7 Sample code generated by Algorithm ASSEMBLY_CODE for the Polish string abc * + de / h * −.

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

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

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

\

\

Real examples of Queue:

- Line of people at ticket window
- Line of vehicles at railway crossing

Types of Queue:

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

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,Y)
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

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.

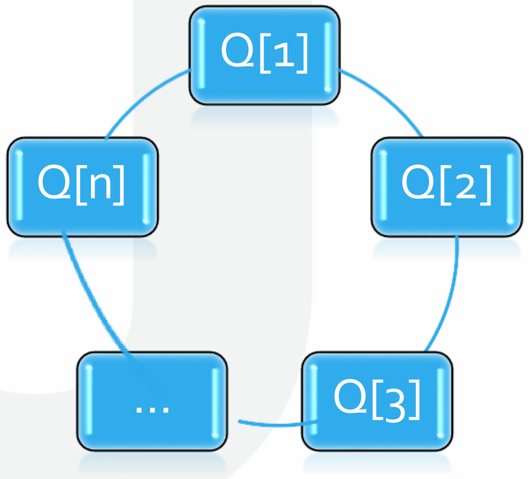


Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

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

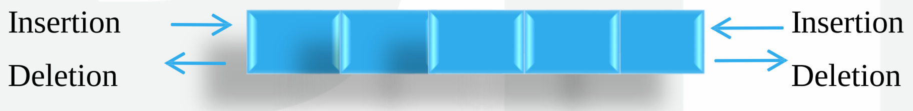
```
QDELETE(F,R,Q,Y)
Step 1 : [Underflow ?]
        if F=0
                then write('Underflow ... No
                 elements to delete')
        Return
Step 2 : [Delete an element]
        Y<-Q[F]
Step 3 : [Queue empty]
        if R=F
                then R<-F<-0
        return (Y)
Step 4 : [Increment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

Image source : Google

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.

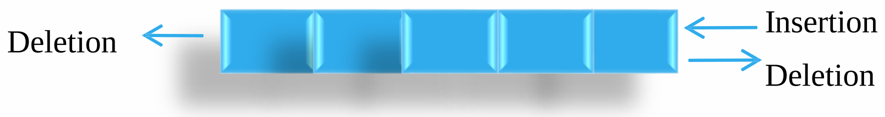


Image source : Google

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.

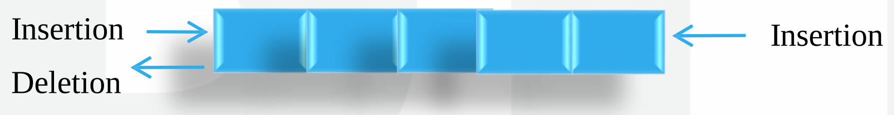


Image source : Google

Double Ended queue (D-queue)

- here are four basic operations in usage of Deque that we will explore:

1. Insertion at rear end
2. Insertion at front end
3. Deletion at front end
4. Deletion at rear end

Algorithm for Insertion at rear end

```
Step-1: [Check for overflow]
            if(rear==MAX)
                  Print("Queue is Overflow");
                  return;
Step-2:  [Insert Element]
            else
                  rear=rear+1;
                  q[rear]=no;
            [Set rear and front pointer]
                  if front=0
                  front=1;
Step-3: return
```

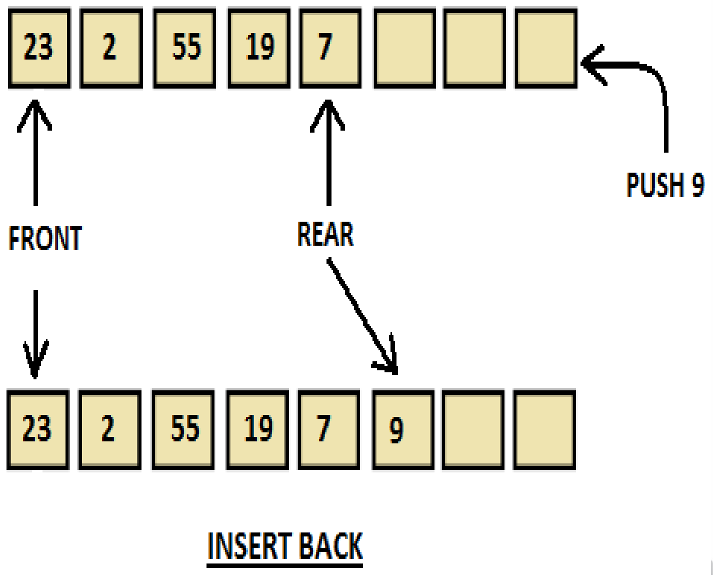


Image source : Google

Parul® University

DIGITAL LEARNING CONTENT

Algorithm for Insertion at front end

```
Step-1 :  [Check for the front position]
                    if(front<=1)
                              Print("Cannot add item at the front");
                              return;
Step-2 :  [Insert at front]
                    else
                              front=front-1;
                              q[front]=no;
Step-3  : Return
```

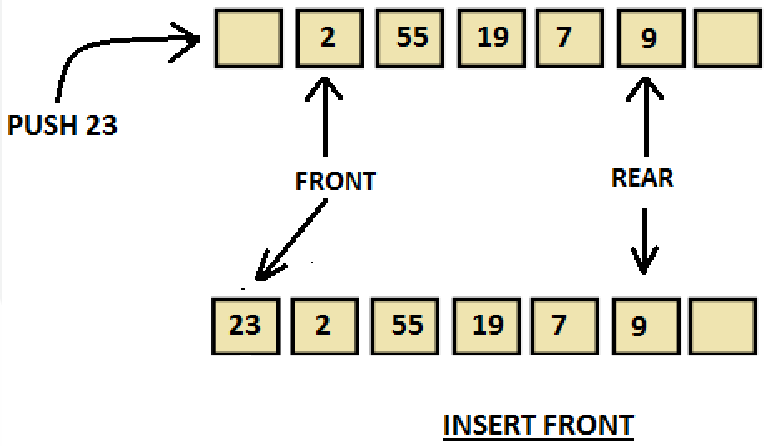


Image source : Google

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
            if front=0
                  print(" Queue is Underflow”);
                  return;
Step-2 [Perform deletion]
            else
                  no=q[front];
                  print(“Deleted element is”,no);
            [Set front and rear pointer]
            if front=rear
                  front=0;
                  rear=0;
            else
                  front=front+1;
Step-3 : Return
```

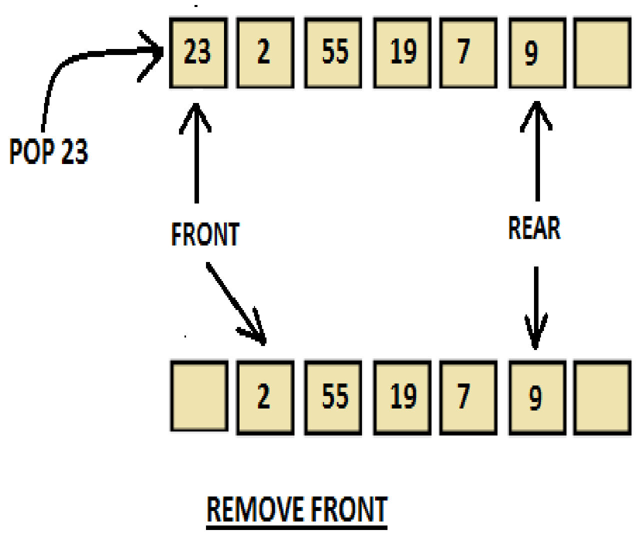
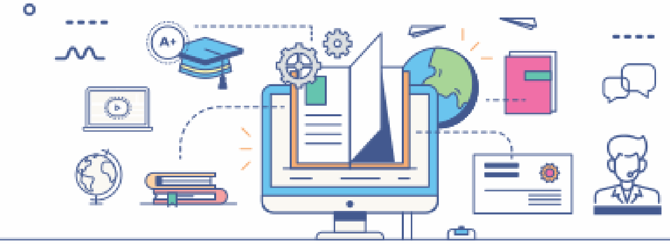


Image source : Google



Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
            if rear=0
                  print("Cannot delete value at rear end");
                  return;
Step-2:  [ perform deletion]
            else
                  no=q[rear];
                  [Check for the front and rear pointer]
            if front= rear
                  front=0;
                  rear=0;
            else
                  rear=rear-1;
                  print("Deleted element is",no);
Step-3 : Return
```

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.

Image source : Google

Linear data structure and their linked representation.

1. Unpredictable storage requirements. The exact amount of data storage required by a program in these areas often depends on the particular data being processed, and consequently, this requirement cannot be easily determined at the time the program is written.
2. Extensively manipulation of the stored data. Programs in these areas typically required that operations such as and insertions and deletion be performed frequently on the data.
3. The linked allocation method of storage can result in both the efficient use of computer storage and computer time.

\

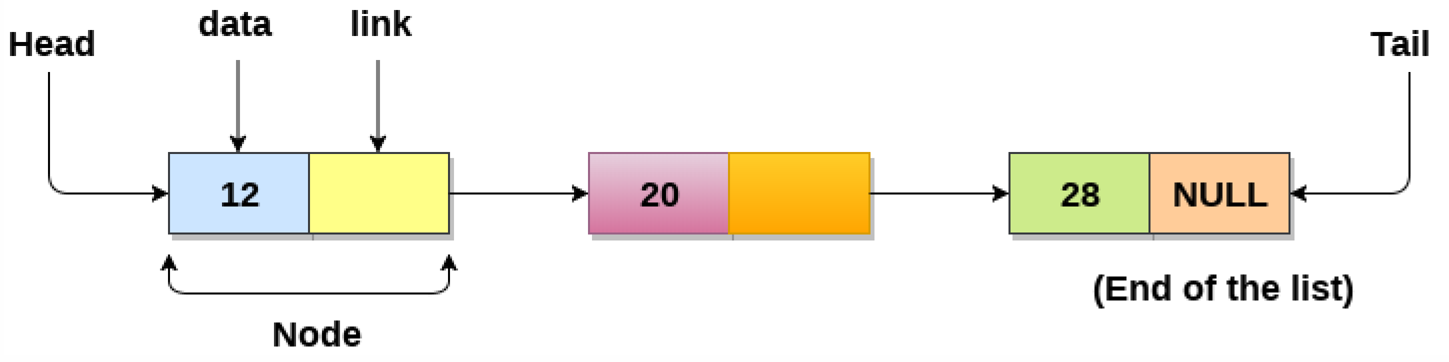
\

Linked List

Static Memory Allocation	Dynamic Memory Allocation
Eg. Array, Stack, Queue etc.	Linked List, Graph, Tree etc.
Linear memory allocation	Linked memory allocation
Storage requirement must be known at the time of creation	Not required
Insertion and deletion operations require large no. of movement of data	Less movement of date is needed
We can directly compute the address of a particular element with help of address calculating formula	It's difficult to calculate the address of a particular node by using formula.
Size of each element in a data structure is same	Size may be different
All the elements are logically as well as physically adjacent to each other in the memory.	All the elements are logically adjacent but need not be physically in the memory.

What is linked list

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null



\

Singly linked list or One way chain

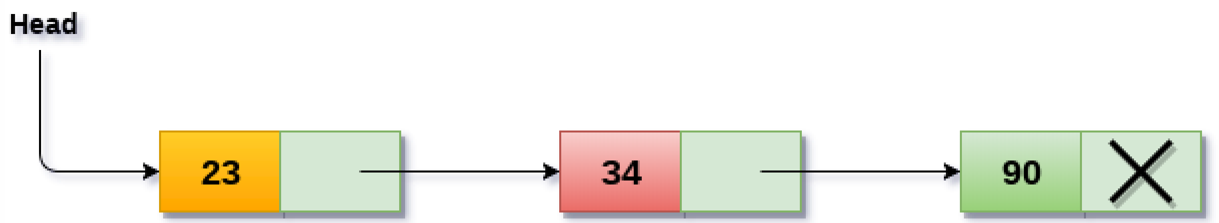
- Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program.
- A node in the singly linked list consist of two parts: **data part and link part.**
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction.
- In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

\

\

Singly linked list or One way chain

- Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



- In the above figure, the arrow represents the links.
- The data part of every node contains the marks obtained by the student in the different subject.
- The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Parul® University

DIGITAL LEARNING CONTENT

Linked List

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

struct link
{
		int info;
		struct link *next;
};
```

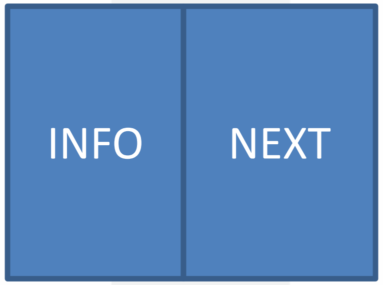
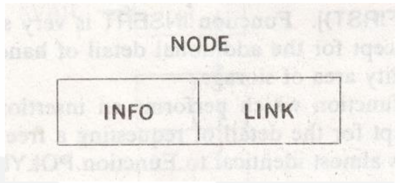


Image source : Google

Linked liner list : Operations on linear list



- It is assumed that an available area of storage for these node structure consist of a linked stack of available nodes.
- The task of obtaining a node from the availability stack can now be formulated. Assume that the address of the next available node is to be stored in the variable NEW. **For practical reasons the availability stack contains only a finite number of nodes: therefore, it must be checked for an overflow condition. This condition signalled by the value of AVAIL being NULL**. If a node is available, then the new top most element of the stack is a denoted by LINKED(AVAIL).the fields of the nodes corresponding to the pointer value of NEW can now be filled in and the field LINK(NEW) is set to a value which designates the successor node of this new node. The availability stack before and after a free has being obtained is given in fig.

Parul® University

DIGITAL LEARNING CONTENT

Linked List

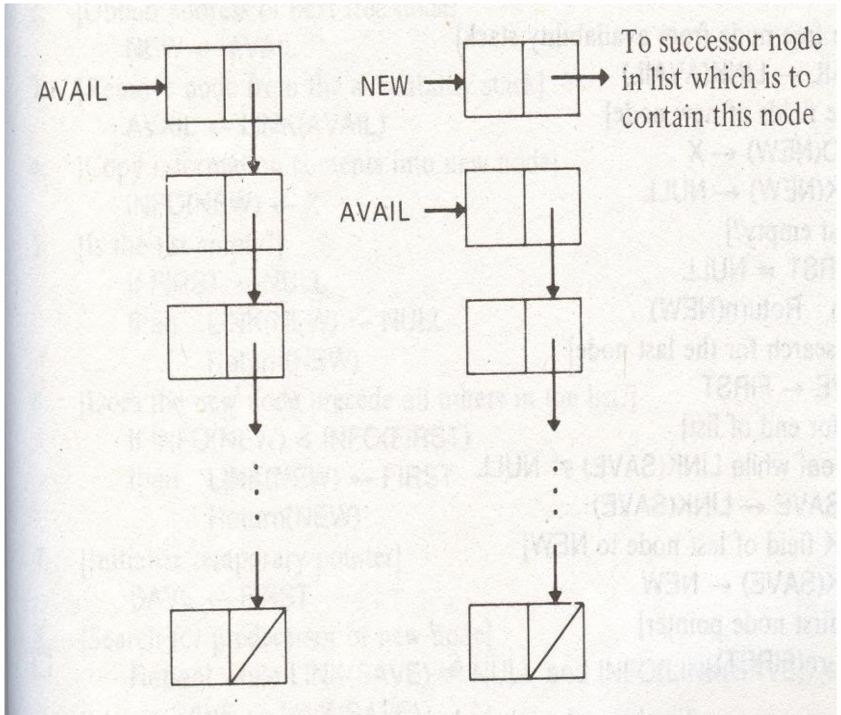


FIGURE 4-2.1 Availability stack before and after obtaining a node from it.

[Obtain address of next free node]
NEW <- AVAIL
[Remove free node from availability stack]
AVAIL <- LINK(AVAIL)

Image source : Youtube video

Linked List

```
[Return node to availability area]
LINK(X) <- AVAIL
AVAIL <- X
Return
```

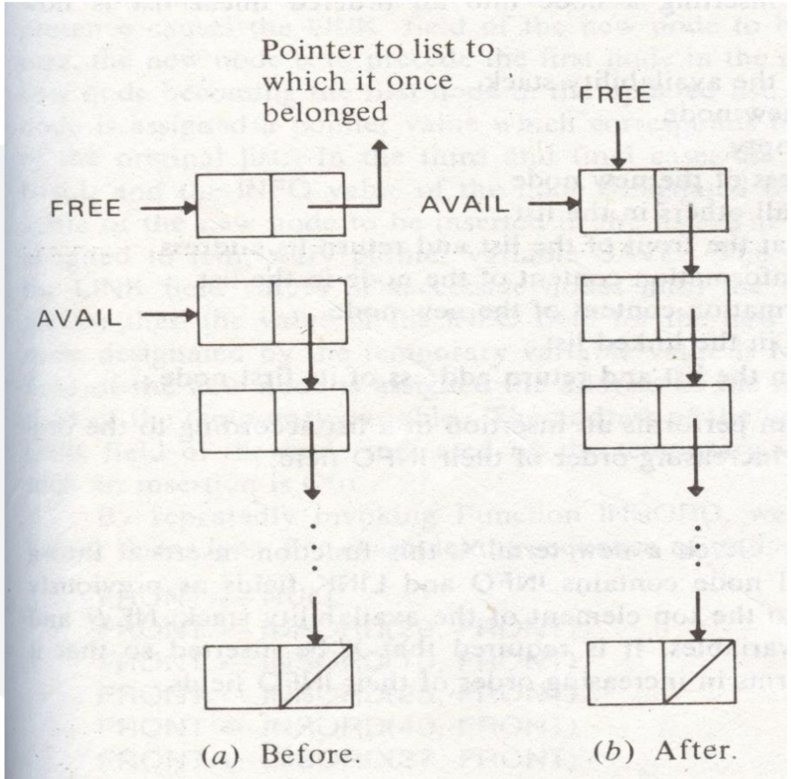


FIGURE 4-2.2 Availability stack before and after a free node has been returned.

Algorithm for insertion of a node at the first position

Function : INSERT(X,FIRST). Given X, a new element , and FIRST , a pointer to the first element of a linked list whose typical node contains INFO and LINK fields as previously described , this function inserts X.AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. It is required that X precede the node whose address is given by FIRST. [1]

```
1. [OVERFLOW?]
        If AVAIL = NULL
        then Write('AVAILABILITY STACK OVERFLOW')
                        Return(FIRST)
2. [Obtain address of next free node]
        NEW <- AVAIL
```

Image source : Google

Algorithm for insertion of a node at the first position

```
3. [Remove free node from availability stack]
        AVAIL <- LINK(AVAIL)
4. [Initialise fields of new node and its link to the list]
        INFO(NEW) <- X
        LINK(NEW) <- FIRST
5. [Set NEW as FIRST]
        FIRST<-NEW
6.[Return address of new node]
        Return(NEW)
```

[1]

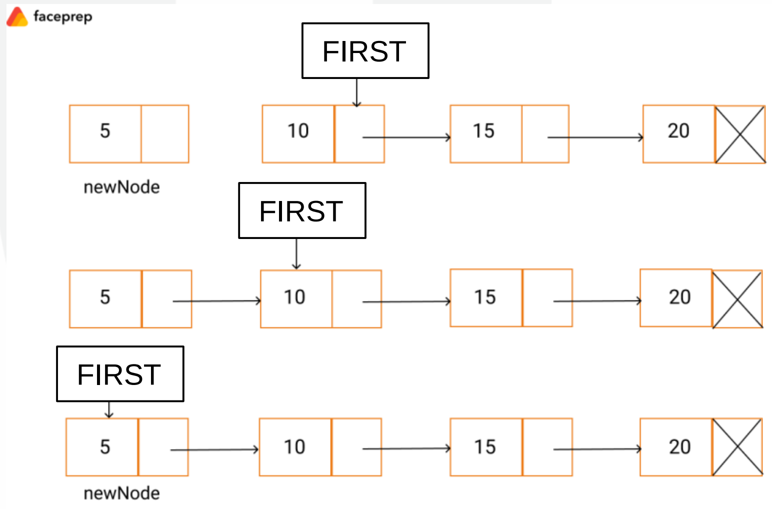


Image source : Google

Algorithm for insertion of a node at the first position

- Let us assume a newNode as shown above. The newNode with data=5 has to be inserted at the beginning of the linked list.
- For this to happen, the address part of the newNode should point to the address of the head node. Once it points to the head node, then the newNode is made as the head node as shown above. [1]

Image source : Google

Algorithm for insertion of node at the end

Function: INSEND(X,FIRST).Given X, a new element , and FIRST , a pointer to the first element of a linked linear list whose typical node contains INFO and LINK fields as previously described , this function inserts X.AVAIL is a pointer to the top element of the availability stack; NEW and SAVE are temporary pointer variables. It is required that X be inserted at the end of the list.

```
1.[Overflow?]
        If AVAIL = NULL
        then Write('AVAILABILITY STACK OVERFLOW')
                Return(FIRST)
2. [Obtain address of next free node]
        NEW <- AVAIL
3. [Remove free node from availability stack]
        AVAIL <- LINK(AVAIL)
```

Image source : Google

Algorithm for insertion of node at the end

```
4. [Initialise fields of new node]
        INFO(NEW) <- X
        LINK(NEW) <- NULL
5. [Is the list empty?]
        If FIRST = NULL
        then Return(NEW)
6. [Initiate search for the last node]
        SAVE <- FIRST
7. [Search for end of list]
        Repeat while LINK(SAVE) ≠ NULL
        SAVE <- LINK(SAVE)
8.[Set LINK field of last node to NEW]
        LINK(SAVE) <- NEW
9.[Return first node pointer]
        Return(FIRST)
```

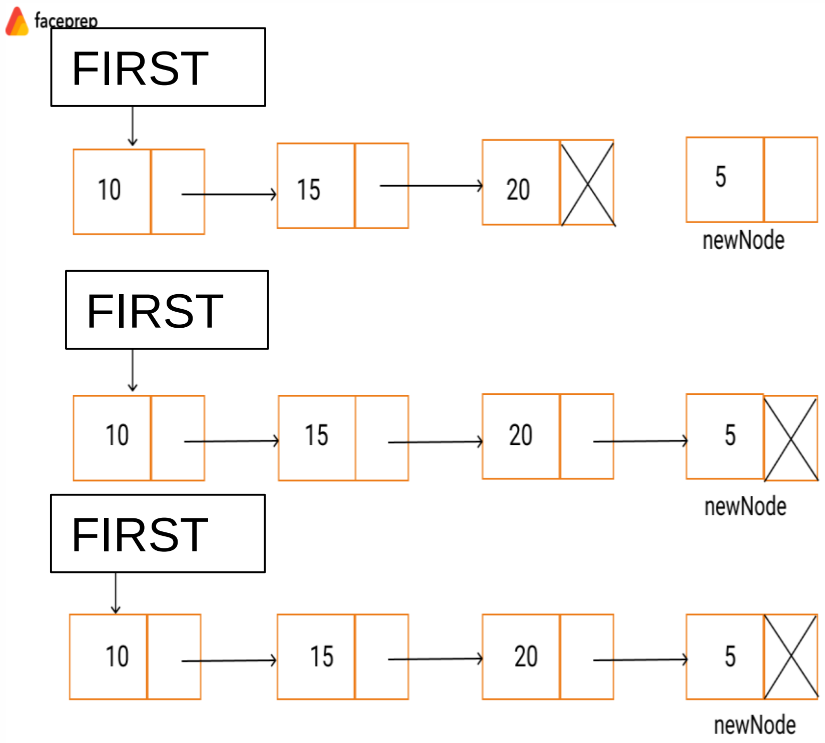


Image source : Google

Linked List

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

struct link
{
        int info;
        struct link *next;
};
```

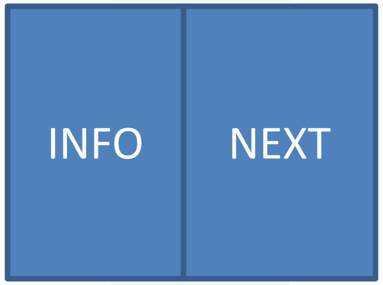


Image source : Google

Linked List

```
void main()
{
        link *start=NULL;
        char ch;
        int x;
        do
        {
                printf("C for Creation\n");
                printf("D for display\n");
                printf("Q for Quit\n");
                flushall();
                scanf("%c",&ch);
    \
switch(ch)
{
        case 'c':
        case 'C':
        start=createList(start);
        break;
        case 'd':
        case 'D':display(start);
        break;
}                                  \
} while (ch !='Q');
}
```

Linked List

```
void display(link *start)
{
        link *tmp;
        tmp=start;
        while (tmp != NULL)
        {
                printf("Info = %d\n",tmp->info);
                tmp = tmp->next;
        }
}
```

\

\

Linked List

```
link *insertAtFirst(link *start,int x)
{
        link *new1;
        new1=(link *)malloc(sizeof(link));
        new1->info=x;
        new1->next=start;
        return(new1);
}
```

\

\

Linked List

```
link *insertAtLast(link *start,int x)
{
        link *new1,save;
        new1=(link *)malloc(sizeof(link));
        new1->info=x;
        new1->next=NULL;
        if (start==NULL)
                return(new1);
        save=start;
        while (save->next != NULL)
                save=save->next;
        save->next=new1;
        return(start);
}
```

\

Linear List

```
link *createList(link *start)
{
        link *tmp,*prv;
        int no;
        tmp=prv=NULL;
        printf("enter any no - 0 for exit");
        scanf("%d",&no);
        while (no!=0)
        {
                tmp=(link *)malloc(sizeof(link));
                tmp->info=no;
                tmp->next=NULL;
                if (start==NULL)
                        start=tmp;
                else
                        prv->next=tmp;
                prv=tmp;
                printf("enter any no - 0 for exit");
                scanf("%d",&no);
        }
        return(start);
}
```

\

\

Algorithm for deletion of element from a linear list

Procedure : DELETE(X,FIRST). Given X and FIRST, pointer variables whose values denote the address of a node in a linked list and the address of the first node in the linked list, respectively , this procedure deletes the node whose address is given by X.TEMP is used to find the desired node , and PRED keeps track of the predecessor of TEMP. Note that FIRST is changed only when X is the first element of the list.

\

\

Algorithm for deletion of element from a linear list

```
1.[Empty list?]
        If FIRST = NULL
        then Write('UNDERFLOW')
        Return
2. [Initialise search for X]
        TEMP <- FIRST
3. [FIND X]
        Repeat through step 5 while TEMP $\neq$ X and LINK(TEMP) $\neq$ NULL
4. [Update predecessor/previous marker]
        PRED <- TEMP
5. [Move to next node]
        TEMP <- LINK(TEMP)
```

\

Algorithm for deletion of element from a linear list

```
6. [End of the list?]
        If TEMP ≠ X
        then Write('NODE NOT FOUND')
        Return
7. [Delete X]
        If X = FIRST (Is X the first node?)
        then FIRST <- LINK(FIRST)
        else LINK(PRED) <- LINK(X)
8. [Return node to availability area]
        LINK(X) <- AVAIL
        AVAIL <- X
        Return
```

\

Copy Linked List

Function: COPY(FIRST). Given FIRST , a pointer to the first node in a linked list , this function makes a copy of this list. A typical node in the given list consists of INFO and LINK fields, the new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR , respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW,SAVE and PRED are pointer variables.

```
1.[Empty list?]
        If FIRST = NULL
        then Return(NULL)
2. [Copy first node]
        If AVAIL = NULL
        then Write('AVAILABILITY STACK OVERFLOW')                    \
                Return(0)
        else NEW <- AVAIL
            AVAIL <- LINK(AVAIL)
           \FIELD(NEW) <- INFO(FIRST)
            BEGIN <- NEW
```

Copy Linked List

3.

```
3.[Initialise traversal]
        SAVE <- FIRST
4. [Move to next node if not at end of list ]
        Repeat through step 6 while LINK(SAVE) ≠ NULL
5. [Update predecessor and save pointers]
        PRED <- NEW
        SAVE <- LINK(SAVE)
6. [Copy node]
        If AVAIL = NULL
        then Write('AVAILABILITY STACK UNDERFLOW')
                Return(0)
        else   NEW <- AVAIL
               AVAIL <- LINK(AVAIL)
               FIELD(NEW) <- INFO(SAVE)
               PTR(PRED) <- NEW
7.[Set link of last node and return]
        PTR(NEW) <- NULL
        Return(BEGIN)
```

Parul® University

DIGITAL LEARNING CONTENT

Doubly linked lists

1. A singly linked list and circular list always traverse only in one direction.
2. But doubly linked list can be traverse either in forward manner or in backward manner.
3. Doubly list is also known as a two-way chain.

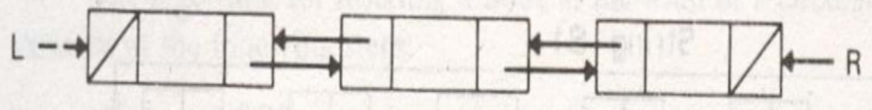


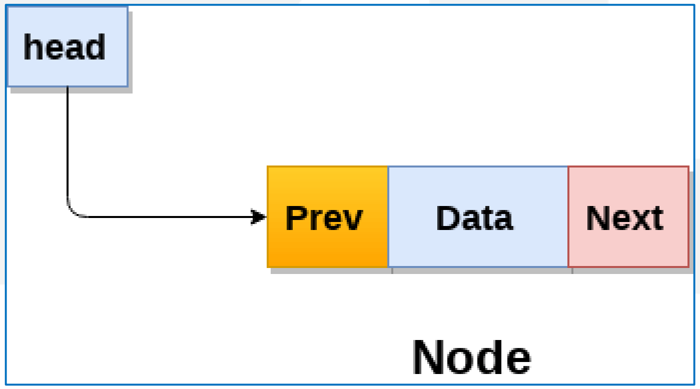
FIGURE 4-2.22 A doubly linked linear list.

\

\

Doubly linked lists

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).
- A sample node in a doubly linked list is shown in the figure.

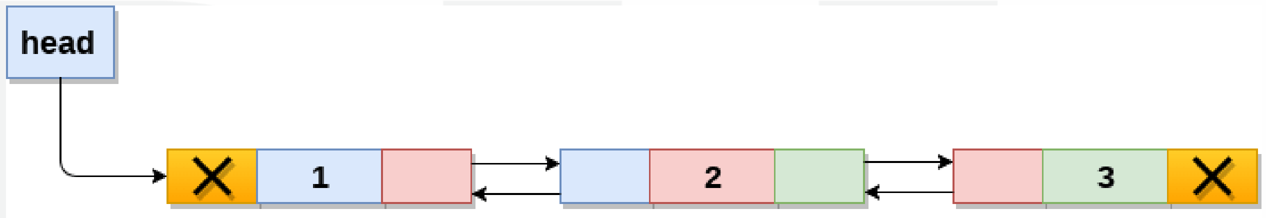


\

\

Doubly linked lists

- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

An algorithm for inserting a node to the left of a given node “M” in a doubly linked linear list.

Procedure : DOUBLEINS(L,R,M,X). Given a doubly linked linear list whose left-most and right-most node addresses are given by the pointer variable. L and R respectively, it is required to insert a node whose address is given by the pointer variable NEW. The left and right links of a node are denoted by LPTR and RPTR, respectively he information field of a node is denoted by the variable INFO.The name of an element of the list is NODE. The insertion is to be performed to the left of a specified node with its address given by the pointer variable M. The information to be entered in the node is contained in X.

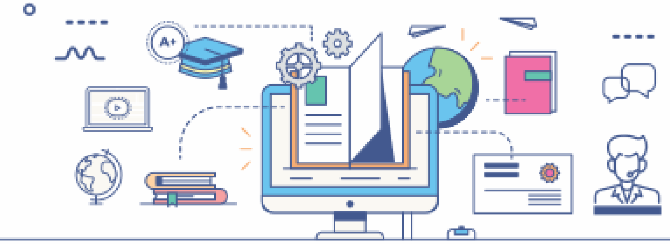
\

An algorithm for inserting a node to the left of a given node “M” in a doubly linked linear list.

```
1. [Obtain new node from availability stack]
NEW <- NODE
2. [Copy information field]
        INFO(NEW) <- X
3. [Insertion into an empty list?]
        If  R = NULL
        then    LPTR(NEW) <- RPTR(NEW) <- NULL
                L <- R <- NEW
                Return
```

\

\



An algorithm for inserting a node to the left of a given node “M” in a doubly linked linear list.

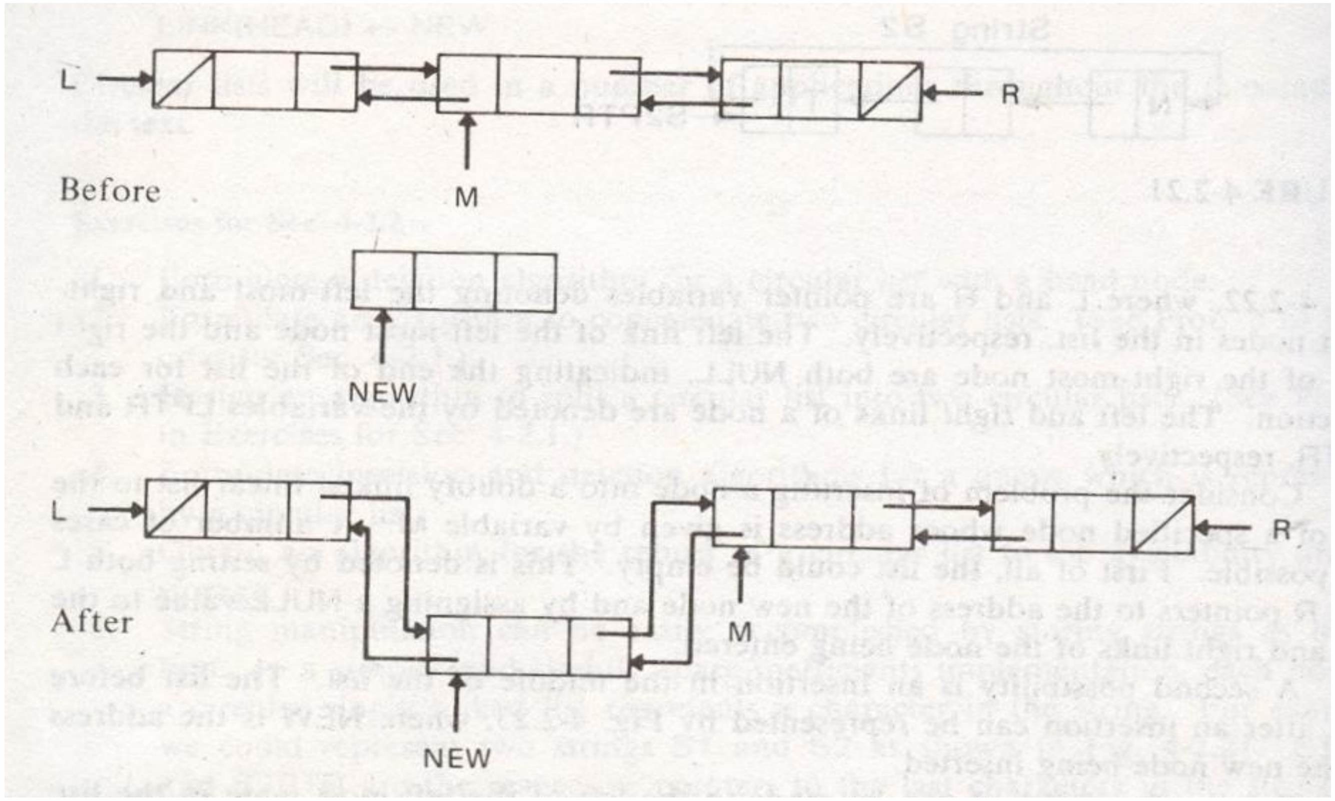
```
4. [Left-most insertion]
        If M=L
        then LPTR(NEW) <- NULL
        RPTR(NEW) <- M
        LPTR(M) <- NEW
        L <- NEW
        Return
5. [Insert in middle]
        LPTR(NEW) <- LPTR(M)
        RPTR(NEW) <- M
        LPTR(M) <- NEW
        RPTR(LPTR(NEW)) <- NEW
        Return
```

\

Parul® University

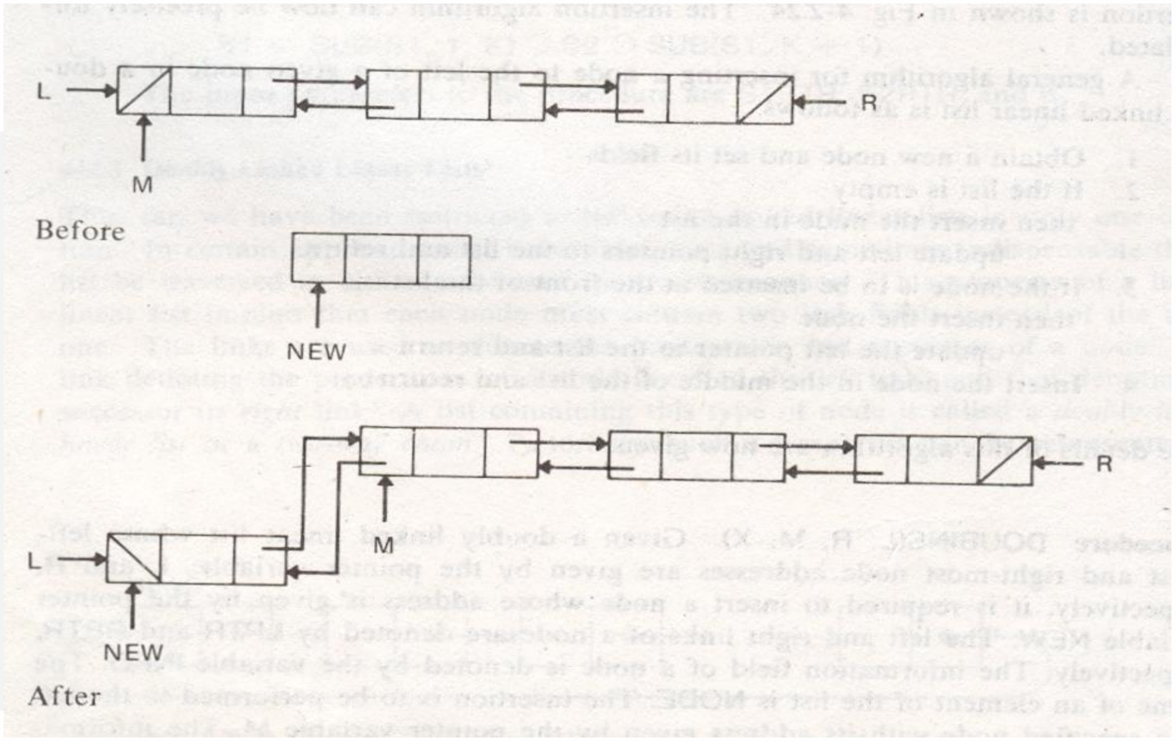
DIGITAL LEARNING CONTENT

Inserting a node in the middle of a given node “M” in a doubly linked linear list.



\

A left most insertion in a doubly linked linear list



\

\

An algorithm for deletion of a node from a doubly linked linear list

Procedure : DOUBDEL(L,R,OLD).Given a doubly linked list with the addressed of the left-most and right-most nodes given by the pointer variables L and R, respectively, it is required to delete the node whose address is contained in the variable OLD. Nodes contain left and right links with names LPTR and RPTR , respectively.

\

An algorithm for deletion of a node from a doubly linked linear list

```
1. [Underflow?]
        If R=NULL
        then Write('Underflow')
                Return
2. [Delete node]
        If L=R (Single node in list)
        then L <- R <- NULL
        else If OLD = L (Left-most node being deleted)
                then L <- RPTR(L)
                    LPTR(L) <- NULL
        else If OLD = R (Right-most node being deleted)
                then R <- LPTR(R)
                    RPTR(R) <- NULL
                else RPTR(LPTR(OLD)) <-  RPTR(OLD)
                    LPTR(RPTR(OLD)) <- LPTR(OLD)
3. [Return deleted node]
        Restore(OLD)
        Return
```

\

Concept of Multi-Linked Lists

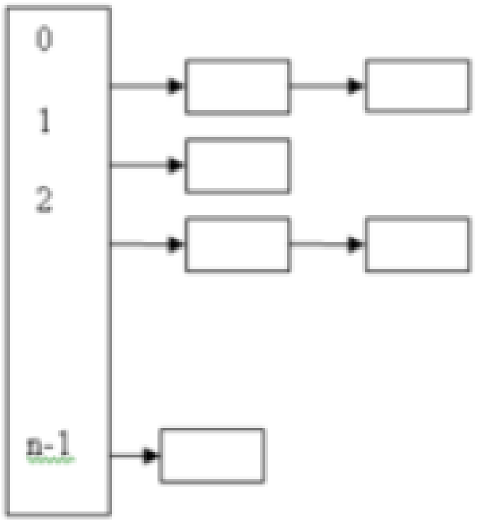
- In a general multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.
- The standard use of multi-linked lists is to organize a collection of elements in two different ways.
- For example, suppose my elements include the name of a person and his/her age. e.g.(FRED,19) (MARY,16) (JACK,21) (JILL,18)

\

\

Applications of Linked List

- Suppose you need to program an application that has a pre-defined number of categories, but the exact items in each category is unknown.



\

Implementation of Stack Operations

Implementation of Queue Operations

Implementation of Sparse\ Matrix

Graph Implementation

Parul® University

DIGITAL LEARNING CONTENT

Linked List

```
link *insertAtFirst(link *start,int x)
{
		link *new1;
		new1=(link *)malloc(sizeof(link));
		new1->info=x;
		new1->next=start;
		return(new1);
}
```

\

\

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in