Data Structure And Algorithm Mid Term Notes
From week No.7 to Week No.16

All Notes Are Available on Our Website
www.cslearnerr.com

Week No. 07: SORTING

Introduction to sorting: Sorting is the fundamental operation in computer science. It refers to the operation of arranging data in some given order such as increasing or decreasing with numerical data or alphabetically with character data. In real life we come across several examples of sorted information e.g., in the telephone directory, the names of the telephone owners are written in alphabetical order etc.

Example: Consider we have six numbers as 1 3 5 2 6 4 we can arrange it in ascending order as 1 2 3 4 5 6 **or** in descending order as 6 5 4 3 2 1. **Similarly,** if we have alphabets B A D E C F we can arrange it in ascending order (A to Z) as A B C D E F **or** in descending order (Z to A) F E D C B A.

A **sort** can be classified as **Internal Sort** if the elements are sorted in main memory or **External Sort** if some of an element that is sorting in auxiliary memory. Here will be discussed **Internal Sort** only.

Sorting Methods: There are many sorting methods which are used but some of them are the following:

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Quick Sort

1. Selection Sort:

In **selection sort,** select the first element and compare it with the rest of the elements. For **ascending order**, if the next compared element is less than the selected element, then swap the selected element with the compared element. **Otherwise** compare the selected element with the next element of array. This process will find the 1st smallest element and put it on the first position in the **first pass**. In **second pass,** it will find the second smallest element of array and put it on the second position and so on.

Similarly, for **descending order**, if the next compared element is greater than the selected element, then swap the selected element with the compared element. **Otherwise** compare the selected element with the next element of array. This process will find the 1st greatest element and put it on the first position in the **first pass**. In **second pass,** it will find the second greatest element of array and put it on the second position and so on.

The **selection sort** is simple to implement. It is however, insufficient for large lists. It is usually used to sort lists no more than **1000** items. For sorting **N** elements, **N-1** passes are required.

Example [Ascending Order]:

	40	59	36	23	65	46
Pass 1:	40	59	36	23	65	46
	40	59	36	23	65	46
	36	59	40	23	65	46
	23	59	40	36	65	46
	23	59	40	36	65	46
	23	59	40	36	65	46

Pass 2:	23	59	40	36	65	46
	23	40	59	36	65	46
	23	36	59	40	65	46
	23	36	59	40	65	46
	23	36	59	40	65	46
Pass 3:	23	36	59	40	65	46
	23	36	40	59	65	46
	23	36	40	59	65	46
	23	36	40	59	65	46
Pass 4:	23	36	40	59	65	46
	23	36	40	59	65	46
	23	36	40	46	65	59
Pass 5:	23	36	40	46	65	59
	23	36	40	46	59	65

After sorting, the elements of the array are: 23 36 40 46 59 65

Algorithm for Selection Sort: This algorithm is used for s**election sort** of a linear array **A** having **N** filled elements. **S** and **C** are the control variables for loops and **TEMP** is used for swapping process.

ALGORITM [For Ascending Order]: SELECTION SORT (A, N, S, C, TEMP)

Step 1: [Starting Selection Loop]
Repeat step 2 for S = 0 to N - 2 by 1
Step 2: [Starting Comparison Loop]
Repeat step 3 for C = S + 1 to N – 1 by 1
Step 3: [Compare elements]
If (A[S] > A[C]) then:
TEMP = A[S]
A[S] = A[C]
A[C] = TEMP
[End of IF structure]
Step 4: [Finish]
Exit

ALGORITM [For Descending Order]: SELECTION SORT (A, N, S, C, TEMP)

Step 1: [Starting Selection Loop]
Repeat step 2 for S = 0 to N - 2 by 1

Step 2: [Starting Comparison Loop]
Repeat step 3 for C = S + 1 to N – 1 by 1

Step 3: [Compare elements]
If (A[S] < A[C]) then:
TEMP = A[S]
A[S] = A[C]
A[C] = TEMP
[End of IF structure]

Step 4: [Finish]
Exit

2. Bubble Sort: **Bubble sort** bubble up the largest value or smallest value to the end. In **bubble sort**, two adjacent memory cells are compared. If 1st is greater than the 2nd then exchanges them. To arrange an array in **ascending order**, through exchange of elements, the largest value slowly floats or bubbles up to the top or end. **Similarly,** to arrange an array in **descending order**, the smaller value slowly bubbles up to the top or end.

Bubble sort is a slow method therefore it is used for sorting limited amount of data. **Bubble sort** is easily programmable. For sorting **N** number of elements, **N-1** passes are required.

Example [Ascending Order]:

	35	17	49	37	15
Pass 1:	35	17	49	37	15
	17	35	49	37	15
	17	35	49	37	15
	17	35	37	49	15
	17	35	37	15	**49**
Pass 2:	17	35	37	15	
	17	35	37	15	
	17	35	37	15	
	17	35	15	**37**	
Pass 3:	17	35	15		
	17	35	15		
	17	15	**35**		
Pass 4:	17	15			
	15	**17**			

Thus after 4 passes, the elements of the sorted array are: 15 17 35 37 49

Algorithm for Bubble Sort: Algorithm for **bubble sort** of a linear array **A** having **N** filled elements. **P** and **C** are the control variables for loops and **TEMP** is used for swapping process.

ALGORITHM [For Ascending Order]: BUBBLE SORT (A, N, P, C, TEMP)

Step 1: [Starting Passes Loop]
Repeat step 2 for P = 1 to N – 1 by 1
Step 2: [Starting Comparison Loop]
Repeat step 3 for C = 0 to (N – P) - 1 by 1
Step 3: [Compare elements]
If (A[C] > A [C + 1]) then:
    TEMP = A [C]
    A [C] = A [C+1]
    A [C + 1] = TEMP
[End of IF structure]
Step 4: [Finish]
Exit

ALGORITHM [For Descending Order]: BUBBLE SORT (A, N, P, C, TEMP)

Step 1: [Starting Passes Loop]
Repeat step 2 for P = 1 to N – 1 by 1
Step 2: [Starting Comparison Loop]
Repeat step 3 for C = 0 to (N – P) – 1 by 1
Step 3: [Compare elements]
If (A[C] < A [C + 1]) then:
    TEMP = A [C]
    A [C] = A [C+1]
    A [C + 1] = TEMP
[End of IF structure]
Step 4: [Finish]
Exit

3. Insertion Sort: **Insertion sort** is performed by inserting each element at the appropriate position. In **insertion sort**, the **first pass** starts with the comparison of 1st element with itself. In **second pass**, the 2nd element is compared with 1st element. In **3rd pass**, the 3rd element is compared with 1st and 2nd element. In general, in **every pass** elements are compared with all elements to its left. If at any point it is found that element can be inserted at a position then space is created for it by shifting the right elements to the right side and inserting the element at a suitable position. For sorting **N** number of elements, **N** passes are required.

Example for ascending order:

	25	57	48	37	12	92
Pass 1:	25	57	48	37	12	92
	25	57	48	37	12	92
Pass 2:	25	57	48	37	12	92
	25	57	48	37	12	92
Pass 3:	25	57	48	37	12	92
	25	48	57	37	12	92
	25	48	57	37	12	92

Pass 4:	25	48	57	37	12	92
	25	48	37	57	12	92
	25	37	48	57	12	92
	25	37	48	57	12	92
Pass 5:	25	37	48	57	12	92
	25	37	48	12	57	92
	25	37	12	48	57	92
	25	12	37	48	57	92
	12	25	37	48	57	92
Pass 6:	12	25	37	48	57	92
	12	25	37	48	57	92

Thus the sorted array elements are: 12 25 37 48 57 92

Algorithm for Insertion Sort: This algorithm is used for **insertion sort** of a linear array **A** having **N** filled elements. **P** and **I** are control variables for loops and **TEMP** is used to store the inserting element.

ALGORITHM (Ascending Order): INSERTION SORT (A, N, P, I, TEMP)

Step 1: [Starting passes loop]
Repeat step 2 & 3 for P = 0 to N – 1 by 1

Step 2: [Set TEMP and Counter variable I]
TEMP = A [P]
I = P – 1

Step 3: [Starting comparison loop]
Repeat step 4 & 5 while (I > = 0 && TEMP < A [I])

Step 4: [Interchange values]
A [I + 1] = A [I]
A [I] = TEMP

Step 5: [Decrement Counter]
I = I – 1

Step 6: [Finish]
Exit

ALGORITHM (Descending Order): INSERTION SORT (A, N, P, I, TEMP)

Step 1: [Starting passes loop]
Repeat step 2 & 3 for P = 0 to N – 1 by 1

Step 2: [Set TEMP and Counter variable I]
TEMP = A [P]
I = P – 1

Step 3: [Starting comparison loop]
Repeat step 4 & 5 while (I > = 0 && TEMP > A [I])

Step 4: [Interchange values]
A [I + 1] = A [I]
A [I] = TEMP

Step 5: [Decrement counter]
I = I – 1

Step 6: [Finish]
Exit

Quick Sort:

Quick sort uses the idea of **divide and conquer** technique. A **quick sort** first selects a value, which is called the **pivot value** [first item in the list]. The role of the **pivot value** is to assist with splitting the list. The actual position where the **pivot value** commonly called the **split point** will be used to divide the list/array into two **halves** (splits) in such a way that elements in the left half are **smaller** than pivot and elements in the right half are **greater** than pivot.

Three steps are used recursively in quick sort:

1. Find **pivot** that divides the array into two sub arrays.
2. Quick sort the left sub array
3. Quick sort the right sub array

Example for Ascending Order:

Consider an array **arr [6]** having **6** elements:

Arr [6] = {5 2 6 1 3 4} arrange the elements in **ascending order** by using **quick sort**:

Pivot

0	1	2	3	4	5
5	2	6	1	3	4

Left Right

Remember the rules:

For Ascending Order

1. All elements to the **right** of **pivot** must be **greater** than **pivot**.
2. All elements to the **left** of **pivot** must be **smaller** than **pivot**.

For Descending Order:

1. All elements to the **right** of **pivot** must be **smaller** than **pivot**.
2. All elements to the **left** of **pivot** must be **greater** than **pivot**.

Is pivot < right? Pivot=5
NO Right=4
swap pivot and right

Pivot

0	1	2	3	4	5
5	2	6	1	3	4

Left Right

After swapping:

Pivot

0	1	2	3	4	5
4	2	6	1	3	5

Left Right

Is pivot > left? NO

Pivot = 5
Left = 4, 2, 6

Swap pivot and left

pivot

0	1	2	3	4	5
4	2	6	1	3	5

Left Right

After swapping:

pivot

0	1	2	3	4	5
4	2	5	1	3	6

Left Right

Is pivot < right? NO

Pivot = 5
Right = 6, 3

Swap pivot and right

pivot

0	1	2	3	4	5
4	2	5	1	3	6

Left Right

After swapping:

pivot

0	1	2	3	4	5
4	2	3	1	5	6

Left Right

Is pivot > left? NO

Pivot = 5
Left = 3, 1, 5

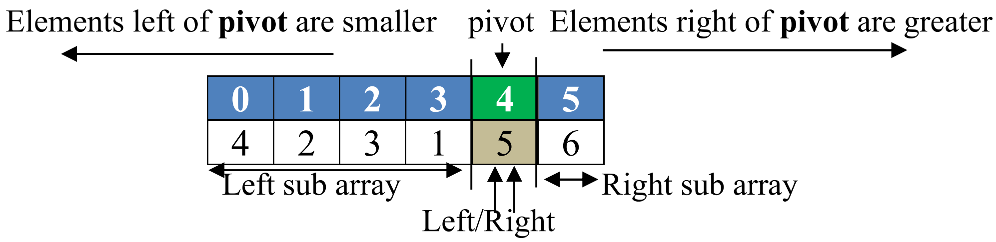
Swap pivot and right

pivot

0	1	2	3	4	5
4	2	3	1	5	6

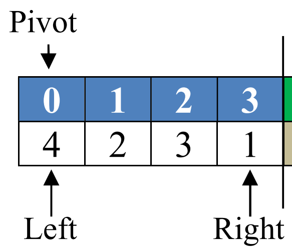
Left/Right

Both left and right are point at the same element of the array
This means **5** is the **pivot** and it is at the sorted position.

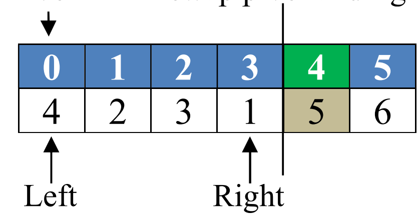


So, **pivot** has divided the array into two sub arrays.

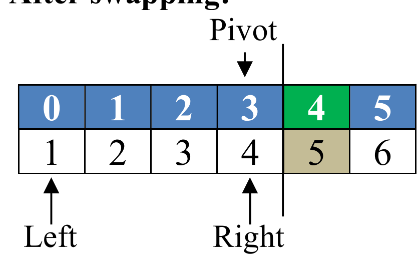
Now quick sort the left sub array:



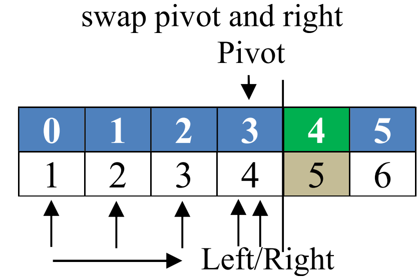
Is pivot < right? Pivot = 4
NO Right = 1
swap pivot and right



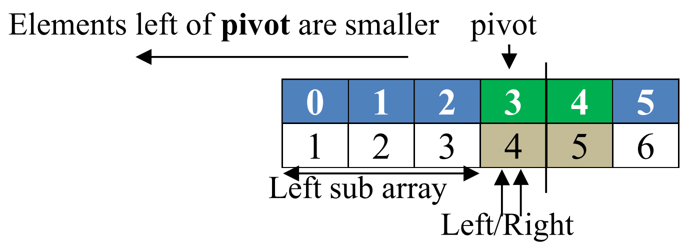
After swapping:



Is pivot > left? Pivot = 4
NO Left = 1, 2, 3, 4
swap pivot and right

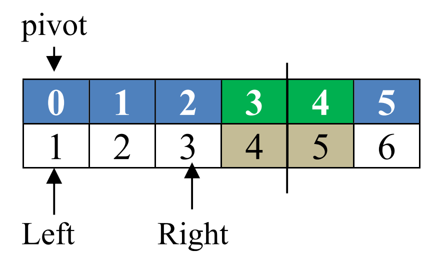


Both left and right are point at the same element of the array
This means **4** is the **pivot** and it is at the sorted position.



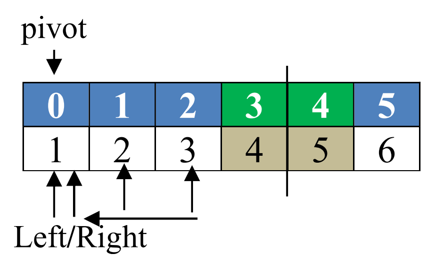
So, **pivot** has divided the array into left sub array and there is a wall at the right side of **4**.

Now quick sort the left sub array:

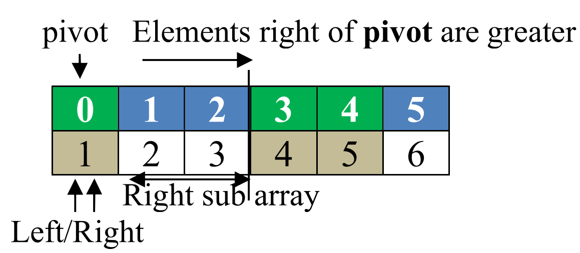


Is pivot < right? NO swap pivot and right

Pivot = 1
Right = 3, 2, 1

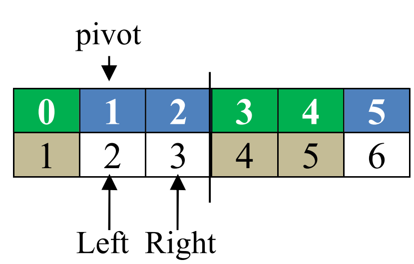


Both left and right are point at the same element of the array
This means **1** is the **pivot** and it is at the sorted position.



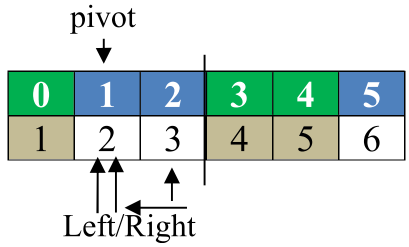
So, **pivot** has divided the array into right sub array and there is no element to the left of **1**.

Now quick sort the right sub array:

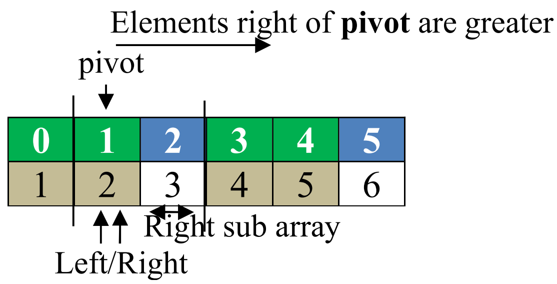


Is pivot < right? NO
swap pivot and right

Pivot = 2
Right = 3, 2

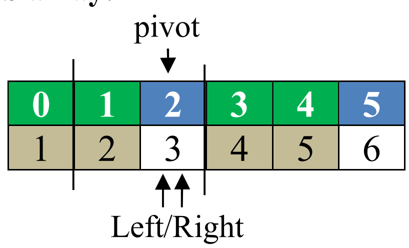


Both left and right are point at the same element of the array
This means **2** is the **pivot** and it is at the sorted position.



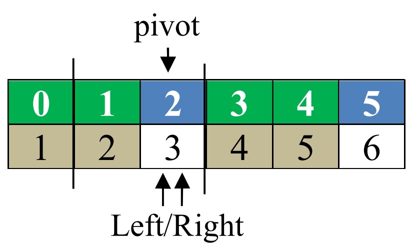
So, **pivot** has divided the array into right sub array and there is a wall at the left side of **2**.

Now quick sort the right sub array:

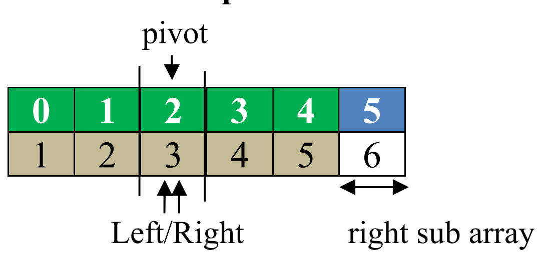


Is pivot < right? NO
swap pivot and right

Pivot = 3
Right = 3



Both left and right are point at the same element of the array
This means **3** is the **pivot** and it is at the sorted position.



Pivot has walls on both the sides so it is done with **left sub array**.

Remember: 5 was the first **pivot** and it divided the array into two sub arrays.

Now quick sort the right sub array:

pivot

0	1	2	3	4	5
1	2	3	4	5	6

Left/Right

Is pivot < right? Pivot = 6
NO Right = 6
swap pivot and right

pivot

0	1	2	3	4	5
1	2	3	4	5	6

Left/Right

Both left and right are point at the same element of the array
This means **6** is the **pivot** and it is at the sorted position.

pivot

0	1	2	3	4	5
1	2	3	4	5	6

Left/Right

So, there is no element to the right side of **6** and also there is a wall at left side of **6**.

Thus, the array is sorted!

0	1	2	3	4	5
1	2	3	4	5	6

Week No. 08: STACK

Introduction to Stack:

Stack is a Linear Data Structure. By **Linear** means that **elements** are stored in **continuous memory location** in computer.

A **Stack** is a list of elements in which an element may be **inserted** or **deleted** only at the **one end**, which is called **TOP** of the **Stack**. It means that elements of **Stack** can be removed in **reverse order**, in which they are inserted into **stack**.

Insertion of **data/element** into the **Stack TOP** is called **PUSH** or "**Stacking**". And **Deletion** of **data/element** from the **Stack TOP** is called **POP** or "**Un Stacking**".

The **items** in the **Stack** are stored and retrieved in **LIFO** and **FILO** manner. **LIFO** means **L**ast **I**n **F**irst **O**ut and **FILO** means **F**irst **I**n **L**ast **O**ut. Other names used for **Stack** are "**PILES**" and "**Push Down List**".

For example: Suppose 11 12 13 14 15 are elements and **PUSH** these elements to the **empty Stack**. They are shown as follow:

15	4	→ **Top of Stack**
14	3	
13	2	
12	1	
11	0	

Stack Implementation:
Stack can be **implemented** in the following **two ways**:

1. Static Implementation
2. Dynamic Implementation

1. Static Implementation:
Static implementation of **Stack** is done through the **linear array** e.g. "**Stack**". A pointer variable **TOP** (which contains the location of the TOP element of the **Stack)** and a variable **MAXSTK** (which gives us maximum number of elements that can be **inserted** or **Pushed** on the **Stack)** are used e.g.

Stack

10	20	30	40				
0	1	2	3	4	5	6	7

TOP $\boxed{3}$ → 3 **MAXSTK** $\boxed{7}$ → 7

There are some **limitations** in the **static implementation** of **stack** using **array** such as:

1. When a **size of the stack** is declared, its size cannot be modified during program execution.

2. It is also inefficient for utilization of memory i.e. when a **stack** is declared then memory is allocated which is equal to the **stack size**. But when need more space in memory at the execution time, the **stack** does not provide the facility to reserve the memory at the execution time. And also, when the **stack** is declared with maximum size but the elements stored in **stack** is less than the maximum size then the remaining memory will be wasted.

2. Dynamic Implementation:

Pointers can also be used for **dynamic implementation** of **stack**. The **link list** is an example of **dynamic implementation**. Using **dynamic implementation**, at run time there is no restriction on the number of elements. The **stack** may be expendable. The **memory** is efficiently utilized with **pointers**. **Memory** is allocated only when element is **pushed** to the **stack**.

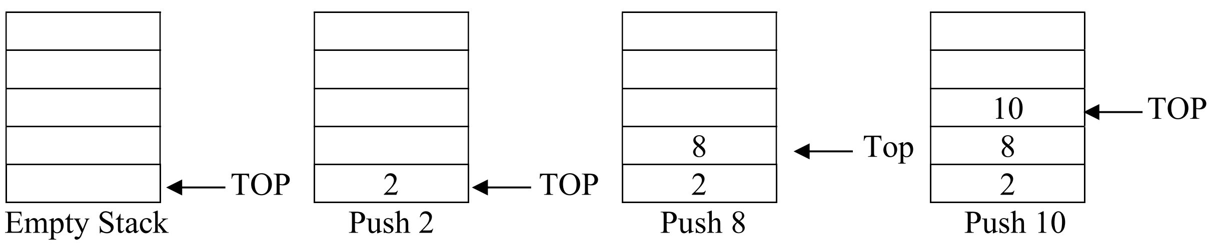
Basic Operations on Stack:

The following **operations** can be performed on the **Stack**.

1. **Create Stack:** This operation creates an **Empty Stack**.
2. **PUSH:** When a **new item** is inserted into the **Top of the Stack** is called **PUSH**.
3. **POP:** When an **item** or **element** is removed from the **Top of the Stack** is called **POP**.
4. **Empty:** Return **true** if **Stack** contains no elements otherwise **false**.

PUSH Operation:

PUSH operation is used to insert an element to the **Stack**. To insert an element first of all the **Stack Top** is monitored if it is equal to the **maximum size** of **Stack** then it shows the message of **Overflow** of **Stack**. Otherwise, the **Top pointer** is incremented: **TOP = TOP + 1** and the **item** is **pushed** in the Top of the **Stack** e.g.



Algorithm for PUSH Operation:

PUSH algorithm is used to push an **ITEM** into the Stack. **TOP** is the pointer pointing to the Top of the **Stack**. **MAXSTK** is the maximum numbers of elements.

ALGORITHM: PUSH (STACK, TOP, MAXSTK, ITEM, N)

```
Step 1:    [Initially TOP and MAXSTK pointers positions]
           TOP = -1 or 0 or 1 or 2…….... & MAXSTK = N-1
Step 2:    [Check overflow condition]
           If (TOP = = MAXSTK) then:
                Write (“OVERFLOW”) and return
           [End of IF structure]
Step 3:    [Set TOP Pointer]
           If (TOP = = -1) then:
                TOP = 0
           Else:
                TOP = TOP + 1
           [End of IF structure]
Step 4:    [Insert value]
           Set STACK [TOP] = ITEM
Step 5:    [Finish]
           Exit
```

Algorithm to PUSH multiple elements at a time:
This algorithm is used to **PUSH** multiple elements at a time in a **stack**.

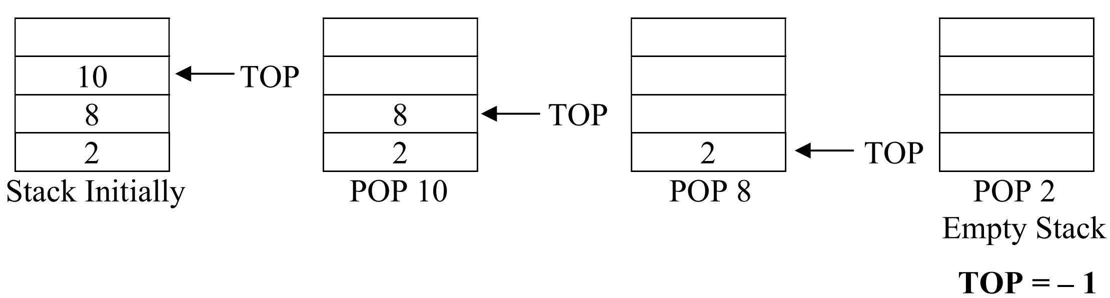
ALGORITHM: PUSH (STACK, TOP, MAXSTK, N, ITEM)

Step 1: [Initially TOP and MAXSTK pointers positions]
TOP = -1 or 0 or 1 or 2…….. & MAXSTK = N-1
Step 2: [Check overflow condition]
If (TOP = = MAXSTK) then:
Write ("OVERFLOW") and return
[End of IF structure]
Step 3: [Start loop to enter multiple elements]
Repeat step 4 & 5 while (TOP < MAXSTK)
Step 4: [Set TOP Pointer]
If (TOP = = -1) then:
TOP = 0
Else:
TOP = TOP + 1
[End of IF structure]
Step 5: [Insert value]
Set STACK [TOP] = ITEM
[End of Loop]
Step 6: [Finish]
Exit

POP Operation:

POP operation is used to delete an element from the **Stack**. To delete an element first of all the S**tack TOP** is monitored if it is equal to the empty stack (**-1** in **C**) then it shows the message of "**Underflow**" or "**Empty Stack**". Otherwise, the **TOP pointer** is decremented: **TOP = TOP - 1** and the **item** is returned.

Example:



Algorithm for POP Operation:
POP algorithm deletes the **TOP** element of **STACK** and assigns it to the variable **ITEM**.

ALGORITHM: POP (STACK, TOP, ITEM)

Step 1: [Initially TOP pointer position]
TOP = – 1 or 0 or 1 or 2……………...
Step 2: [Check underflow condition]
If TOP = = – 1 then:
Write ("UNDERFLOW") and return
[End of IF Structure]
Step 3: [Assign TOP element to ITEM]
ITEM = STACK [TOP]
STACK [TOP] = -1 //Assign Garbage value
Step 4: [Set TOP Pointer]
If (TOP = = 0) then:
TOP = -1
Else:
TOP = TOP - 1
[End of IF Structure]
Step 5: [Finish]
Exit

Algorithm to POP multiple elements at a time:
This algorithm is used to delete **multiple elements** at a time from a **STACK**.

ALGORITHM: POP (STACK, TOP, ITEM)

Step 1: [Initially TOP pointer position]
TOP = – 1 or 0 or 1 or 2………………
Step 2: [Check underflow condition]
If TOP = = -1 then:
Write ("UNDERFLOW") and return
[End of IF Structure]
Step 3: [Start loop to delete multiple elements]
Repeat step 4 & 5 while (TOP > = 0)
Step 4: [Assign TOP element to ITEM]
ITEM = STACK [TOP]
STACK [TOP] = - 1 //Assign Garbage value
Step 5: [Set TOP pointer]
If (TOP = = 0) then:
TOP = -1
Else:
TOP = TOP - 1
[End of IF Structure]
[End of Loop]
Step 6: [Finish]
Exit

Week No. 09: QUEUE

Introduction to Queue:

A **Queue** is a Linear List of elements in which **Deletion** can take place only at **one end**, called **FRONT**, and **Insertion** can take place at **other end**, called **REAR**. The term ***FRONT*** and ***REAR*** are used in describing a Linear List only when it is implemented as a **Queue**.

The **Queue** is also called **First In First Out** (FIFO) Lists, since the first element in the **Queue** will be the first element that can get out of the **Queue**. In **other words**, the order in which the elements enter into **Queue** is the order in which they will leave.

The **purpose of Queue** is to provide some form of **buffering**. In a computer system, **Queue** is used for:

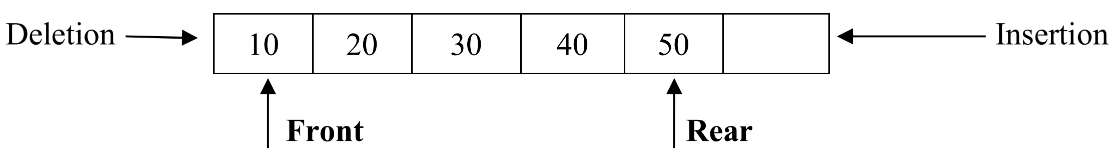
Process Management: For example, in a timesharing system in computer, programs are added to a **queue** and are executed one after the other.

Buffer between the fast computer and a slow printer: Documents sent to the printer for printing is added to a **queue**. The document sent first is printed first and document sent last is printed last.

An important **example** of **queue** in computer science occurs in a **timesharing system** in which programs with different **priorities** form a **queue** which waiting to be executed (**priority queue**).

Queues abound in **everyday life**. The **automobiles** are waiting to pass through an intersection from a **queue** in which the first car in line is the first can through; the **people** are waiting in line at a bank from a **queue**, where the first person in line is the first person to be waited on and so on.

For example:



Operation on Queue:

Four primitive's operations applied to a **Queue**.

1. **Insertion:** The insertion operation [***insert (QUEUE, ITEM)***] inserts an **ITEM** at the ***REAR*** of a **QUEUE**.

2. **Deletion:** The deletion operation [**ITEM =** ***remove (QUEUE)***] removes the element from the ***FRONT*** of a **QUEUE**.

3. **Empty:** The third operation [***empty (QUEUE)***] return **FALSE** when the **QUEUE** is not empty **otherwise** it returns **TRUE** if the **QUEUE** is **empty**.

4. **Full:** There is another operation performed on a **QUEUE** when it is implemented in linear arrays. The full operation [***full (QUEUE)***] return **TRUE** value if the **QUEUE** is **full** otherwise it returns **FALSE**.

Representation of Queue:
The **queue** is represented in the computer in various ways, usually by means of **one-way link list** or **linear array**. Let takes the **example** of **linear array**.

Consider a linear array **QUEUE** and two pointers' variables **FRONT** and **REAR**. **FRONT** will contain the location number of the **front element** of the **queue** and **REAR** will contain the location number of the **rear element**. The condition **FRONT = REAR = – 1** will indicate that **queue** is **empty**. If the array **QUEUE** has **N** elements, then whenever a **new element** is added, the value of the **REAR** will be incremented by **1** e.g.

REAR = REAR + 1

Similarly, when an item is deleted the value of the **FRONT** will be incremented by **1** e.g.

FRONT = FRONT + 1

Types of Queue:

There are two types of **Queue**:

1. Non-Circular Queue
2. Circular Queue

1. Non-Circular Queue:

Simple example for non-Circular QUEUE: To explain the above **operations** we have a simple **example** as follow:

FRONT = REAR = -1 or NULL
Empty (QUEUE) =? It will return **TRUE** because **QUEUE** is empty.

0	1	2	3	4

FRONT = REAR = 0
Insert (QUEUE, A)

0	1	2	3	4
A				

FRONT = 0, REAR = 1
Insert (QUEUE, B)

0	1	2	3	4
A	B			

FRONT = 0, REAR = 2
Insert (QUEUE, C)

0	1	2	3	4
A	B	C		

full (QUEUE) =? It will return **FALSE** because the **QUEUE** is not **full**.

FRONT = 1, REAR = 2
ITEM = remove (QUEUE, A)

0	1	2	3	4
	B	C		

FRONT = 1, REAR = 3
Insert (QUEUE, D)

0	1	2	3	4
	B	C	D	

FRONT = 1, REAR = 4
Insert (QUEUE, E)

0	1	2	3	4
	B	C	D	E

FRONT = 2, REAR = 4
ITEM = remove (QUEUE, B)

0	1	2	3	4
		C	D	E

full (QUEUE) =? It will return **TRUE** because the **QUEUE** is **full** (means REAR = N-1).

2. **Circular Queue:**

Example: To explain the above **operations** we have a simple **example** as follow:

0	1	2	3	4

FRONT = REAR = -1 or NULL

Empty (QUEUE) =? It will return **TRUE** because **QUEUE** is **empty**.

FRONT = REAR = 0
Insert (QUEUE, A)

0	1	2	3	4
A				

FRONT = 0, REAR = 1
Insert (QUEUE, B)

0	1	2	3	4
A	B			

FRONT = 0, REAR = 2
Insert (QUEUE, C)

0	1	2	3	4
A	B	C		

FRONT = 1, REAR = 2
ITEM = remove (QUEUE, A)

0	1	2	3	4
	B	C		

FRONT = 1, REAR = 3
Insert (QUEUE, D)

0	1	2	3	4
	B	C	D	

FRONT = 1, REAR = 4
Insert (QUEUE, E)

0	1	2	3	4
	B	C	D	E

FRONT = 2, REAR = 4
ITEM = remove (QUEUE, B)

0	1	2	3	4
		C	D	E

full (QUEUE) =? It will return **FALSE** because the **QUEUE** is not **full**.

FRONT = 2, REAR = 0
Insert (QUEUE, F)

0	1	2	3	4
F		C	D	E

FRONT = 2, REAR = 1
Insert (QUEUE, G)

0	1	2	3	4
F	G	C	D	E

full (QUEUE) =? It will return **TRUE** because the **QUEUE** is **full** (means REAR = N-1 && FRONT = 0 || REAR + 1 = FRONT).

In **Circular Queue,** the **QUEUE** [**0**] comes after the **QUEUE** [**N – 1**] in **linear array** i.e. the first element stored after the last element of the queue if the space is available. With this assumption an **ITEM** is inserted into the **QUEUE** by assigning **ITEM** to **QUEUE** [**0**]. Instead of incrementing **REAR** to **N,** reset **REAR = 0** and then assign the **ITEM**.

QUEUE [REAR] = ITEM

Similarly, if **FRONT = N – 1** and the element of **QUEUE** is deleted then reset **FRONT = 0** instead of increasing **FRONT** to **N**.

Suppose that **QUEUE (**Circular/non-Circular**)** contains only one element i.e. **FRONT = REAR** and the element is deleted then reset **FRONT = REAR = NULL** (or **-1**) indicating that the **queue** is **empty**. If **FRONT = REAR = -1** (means queue is empty) and the element is inserted, then reset **FRONT = REAR = 0**.

Insertion Algorithm for Non-Circular QUEUE: This algorithm is used to **insert** an **ITEM** to **Non-Circular QUEUE**.

```
ALGORITHM:    INSERTION (FRONT, REAR, ITEM, N, QUEUE)
              Step 1:  [Initially FRONT & REAR pointers positions]
                       FRONT = -1 or 0 or 1 or 2.......& REAR = -1 or 0 or 1 or 2……
              Step 2:  [Check overflow condition]
                       If (REAR = = N - 1) Then:
                              Write (“Overflow”)
                              Return
                       [End of IF Structure]
              Step 3:  [Set REAR Pointer]
                       If (REAR = = -1) Then:
                              REAR = 0
                       Else:
                              REAR = REAR +1
                       [End of IF Structure]
              Step 4:  [Insert the Item]
                       QUEUE [REAR] = ITEM
              Step 5:  [Set the FRONT Pointer]
                       If (FRONT = = -1) Then:
                              FRONT = 0
                       [End of IF Structure]
              Step 6:  [Finish]
                       Exit
```

Insertion Algorithm for Circular QUEUE: This algorithm is used to **insert** an **ITEM** to **Circular QUEUE**.

```
ALGORITHM:    INSERTION (FRONT, REAR, ITEM, N, QUEUE)
              Step 1: [Initially FRONT & REAR pointers positions]
                      FRONT = -1 or 0 or 1 or 2.......& REAR = -1 or 0 or 1 or 2………
              Step 2: [Check overflow condition]
                      If (FRONT = 0 && REAR = N - 1 || REAR + 1 = FRONT) Then:
                              Write (“Overflow”)
                              Return
                      [End of IF Structure]
              Step 2: [Set REAR Pointer]
                      If (REAR = = - 1 || REAR = = N - 1) Then:
                              REAR = 0
                      Else:
                              REAR = REAR +1
                      [End of IF Structure]
              Step 3: [Insert the Item]
                      QUEUE [REAR] = ITEM
              Step 4: [Set the FRONT Pointer]
                      If (FRONT = = -1) Then
                              FRONT = 0
                      [End of IF Structure]
              Step 5: [Finish]
                      Exit
```

Deletion Algorithm for Non-Circular QUEUE: This algorithm is used to **delete** an **ITEM** from **non-Circular QUEUE**.

ALGORITHM: DELETION (FRONT, REAR, QUEUE, ITEM)

```
Step 1: [Initially FRONT & REAR pointers positions]
        FRONT = -1 or 0 or 1 or 2.......& REAR = -1 or 0 or 1 or 2………
Step 2: [Check Underflow Condition]
        If (FRONT = = -1) Then:
                Write ("Underflow")
                Return
        [End of IF Structure]
Step 3: [Delete Item]
        ITEM = QUEUE [FRONT]
        QUEUE [FRONT] = -1          //Assign Garbage value
Step 4: [Set FRONT and REAR Pointers]
        If (FRONT = = REAR) Then://its mean queue has only one item
                FRONT = REAR = NULL     // or   FRONT = REAR = - 1
        Else:
                FRONT = FRONT + 1
        [End of IF Structure]
Step 5: [Finish]
        Exit
```

Deletion Algorithm for Circular QUEUE: This algorithm is used to **delete** an **ITEM** from **Circular QUEUE**.

ALGORITHM: DELETION (FRONT, REAR, QUEUE, N, ITEM)

```
Step 1: [Initially FRONT & REAR pointers positions]
        FRONT = -1 or 0 or 1 or 2.......& REAR = -1 or 0 or 1 or 2………
Step 2: [Check Underflow Condition]
        If (FRONT = = -1) Then:
                Write ("Underflow")
                Return
        [End of IF Structure]
Step 3: [Delete Item]
        ITEM = QUEUE [FRONT]
        QUEUE [FRONT] = -1             //Assign Garbage value
Step 4: [Set FRONT and REAR Pointers]
        If (FRONT = = REAR) Then:    //its mean queue has only one item
                FRONT = REAR = NULL     //or FRONT = REAR = - 1
        Else if (FRONT = = N - 1) Then:
                FRONT = 0
        Else:
                FRONT = FRONT + 1
        [End of IF Structure]
Step 5: [Finish]
        Exit
```

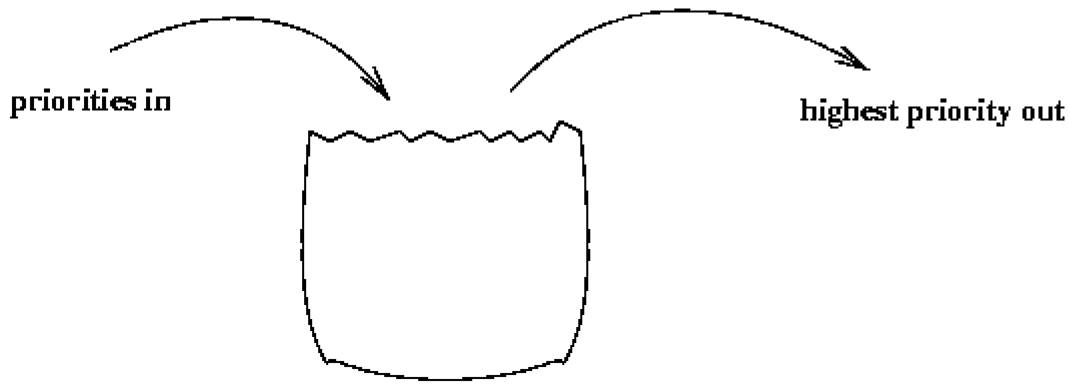
Priority Queue:

A **Priority Queue** is different from a "**Normal Queue**", because instead of being a "**FIFO** (**F**irst-**I**n-**F**irst-**O**ut)" technique, data structure values come **out** in order by **Priority**.

Priority Queue stores multiple tasks using a partial ordering based on **Priority** and ensure **Highest Priority** task at the **Head** of **Queue**.

Priority Queue is a variant (alternative/modified) of **Queue** in which **Insertion** is performed in the order of arrival and **Deletion** is performed based on the **Priority** means each element is deleted on the basis of their **Priority** [Higher Priority > Lower Priority]. If there is **Same Priority,** then will base on **FCFS** (**F**irst **C**ome **F**irst **S**erve) technique.

Here is a conceptual picture of a **Priority Queue**:



Think of a **Priority Queue** as a kind of **Bag** that holds **Priorities**. One can put **In** and the current **Highest Priority** can take **Out**. (**Priorities** can be any Comparable values e.g. use numbers etc.)

Priority Queue is an extension of **Queue** with the following **Properties:**

1. Every item has a **Priority** associated with it.
2. An element with **High Priority** is **Dequeued** before an element with **Low Priority**.
3. If two elements have the **Same Priority**, they are served according to their **Order** in the **Queue**.

A typical **Priority Queue** supports following **Operations:**

1. **Insert (Item, Priority):** Inserts an item with given **Priority**.
2. **GetHighestPriority ():** Returns the **Highest Priority** item means find/search **Highest Priority** item.
3. **DeleteHighestPriority ():** Removes the **Highest Priority** item.

Types of Priority Queue:

1. **Ascending Order Priority Queue/Min Priority Queue:**

   Lower number is given to a **High Priority** e.g. **1 2 3 4**……………………...**n**

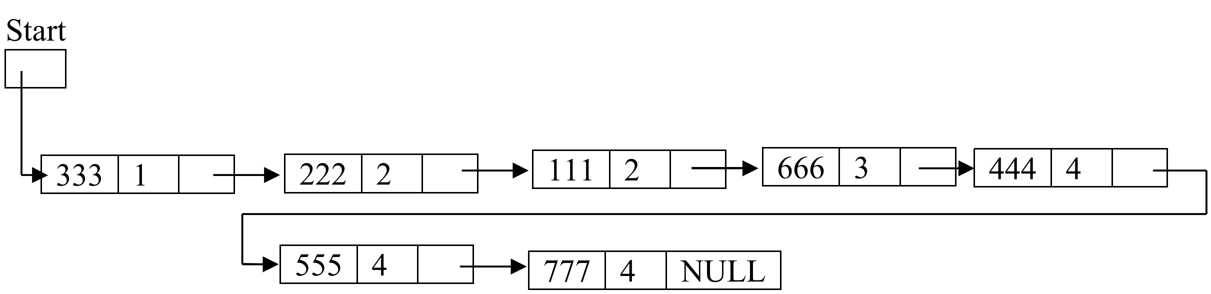
   Example: A **Priority Queue** might be used, for **example**, to handle the jobs sent to the **Computer Science Department's printer**: Jobs sent by the department **chairman** should be printed **first**, then jobs sent by **professors**, then those sent by **graduate students**, and finally those sent by **undergraduates**. The values put into the **priority queue** would be the **priority** of the **sender** (e.g. using **1** for the chairman, **2** for professors, **3** for graduate students, and **4** for undergraduates), and the associated information would be the document to print. Each time the **printer** is free; the job with the **highest priority** would be removed from the **print queue** and printed. (**Note** that it is **OK** to have multiple jobs with the **same priority**; if there is more than one job with the **same highest priority** and when the **printer** is free, then any one of them can be selected).

2. **Descending Order Priority Queue/Max Priority Queue:**

   Higher number is given to **Higher Priority** e.g. **n**…………**4 3 2 1**.

   Example: Same **example** as **above**. The values put into the **priority queue** would be the **priority** of the **sender** (e.g. using **4** for the chairman, **3** for professors, **2** for graduate students, and **1** for undergraduates), and the associated information would be the document to print. Each time the **printer** is free; the job with the **highest priority** would be removed from the **print queue** and printed. (**Note** that it is **OK** to have multiple jobs with the **same priority**; if there is more than one job with the **same highest priority** and when the **printer** is free, then any one of them can be selected).

Representation of Priority Queue (Min Priority Queue) as One-Way Linked List:



Loc.	List	Priority	Link
0	222	2	4
1	444	4	2
2	555	4	6
3	333	1	0
4	111	2	5
5	666	3	1
6	777	4	NULL
-	-	-	-
-	-	-	-
-	-	-	-

Week No. 10: ONE WAY LINKED LIST

Introduction:
A **list** is an ordered collection of data. One way to use the **list** is **sequential array**. In this type of **lists**, it is easy to compute the **address** of an **element** for **storage** and **retrieval** purposes. On the other hand, these have certain **limitations**. There are many situations in which there is a need for a **data structure** in which **data** can be **updated**, **inserted** and **deleted** continuously and the **data** should be in **sorted format** at run time. That is, **insertion** and **deletion** of **data items** frequently occurs. But it is relatively **expensive** to **insert** or **delete** elements from **sequential list** e.g. array.

The **problems** with **arrays** are:

- When a **number** of **users** share **main memory**, there may not be enough adjacent memory locations left to hold an array. But there could be **enough memory** in the shape of small free blocks.
- The **second major problem** with **array** is when we have a large list of data elements and exact number of elements cannot be known in advance while an **array** has a **fixed size** and we cannot increase the **size of array** on run time when **additional memory** is required; therefore, **arrays** are called **static data structure**.
- The **data access speed** becomes **slow** when the **size** of the **array** becomes **large**.

To **overcome** these **limitations Linked Lists** are used. In a **link list** the **elements** are **logically adjacent** needs not to be **physically adjacent** in the **memory,** but they should be **linked** or **connected** through a **pointer**.

Link List:
The **link list** is a **dynamic data structure** i.e. the **size** is not **fixed** and it will expand during program execution. Also, a **link list** is a **linear data structure** having **unlimited** elements, each **element** of a **link list** is called a "**node**".

Types of Link List: There are two types of link list:

- Single Linked List/One Way Linked List
- Double Linked List/Two Way Linked List

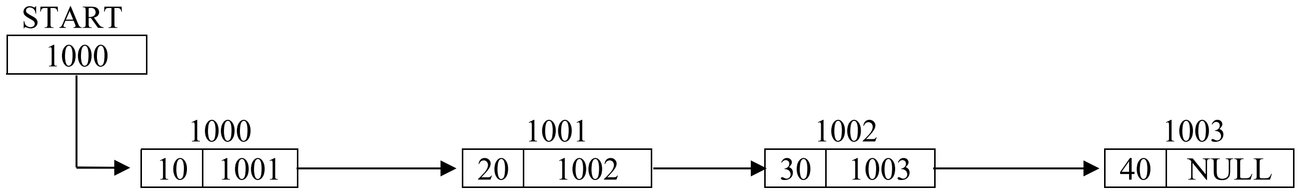
Single Linked List/One Way Linked List:
In **one-way link list**, a node (each element of a link list is called a "**node**") have at least **two fields**, the **first one** is the **data** or **information field** (more than one data fields can be used in a node to store information) which contain the **actual data** of a **list** and the **other field** is called **link field** or **next-pointer field** which contain the **address** of the **next node** of a **link list**.

Data/Information	Pointer

Example:

In the following **diagram** of a **one-way link list** have **four nodes**. **Each node** has **two parts**. The **left part** represents the **information part** of the **node** and the **right part** represents **pointer field** or **link field**, which contains the **memory address** of the **next node**. In the **list** we have a **start node** whose **address** is stored in a pointer **START**. We need a **START** pointer to trace the **list**. A special case is the **list** that when the **list** has **no nodes**. Such a **list** is called a **null list** or **empty list** and denoted by **null pointer** in the **START** pointer.

The **pointer field** of the **last node** will contain the **null pointer** to show the **end** of the **list**.



One Way Link List

Operations on One Way Linked List with algorithms:

1. Insertion Operation:

Let us suppose **LIST** be a **one-way link list** with **N** successive nodes and a **node insert** is to be inserted in a **link list**. The **given node** can be inserted in a **link list** in the following three locations.

I. Front Insertion
II. Middle Insertion
III. End Insertion

I. Front Insertion:

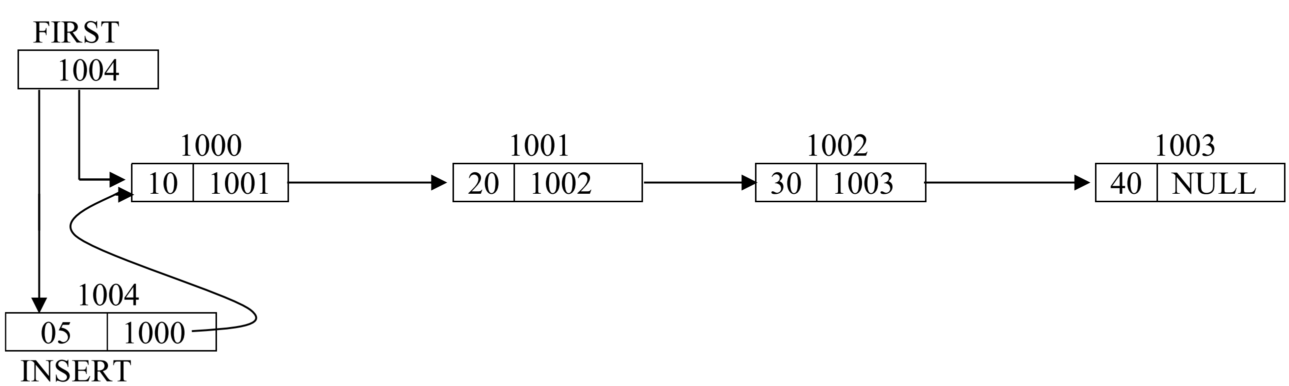
It is the easiest way to **insert** a **node** in the **link list**. Following **algorithm** is used to **insert** a **node** in the **front** of the **LIST**.

ALGORITHM: FRONT INSERTION (Insert, Node, First, Data, Info)

This **algorithm** is used to **insert** a **node** insert at the **start** of a **one-way link list**.

Step 1: [Create a node and assign it to insert]
Insert = (struct node *) malloc (size of (struct node))
Step 2: [Store Information]
Insert -> data = info
Step 3: [assign first to the insert link]
Insert -> link = First
Step 4: [Set first as insert]
First = Insert
Step 5: [Finish]
Exit

In **step 1**, **2** & **3** we create a **node insert**, store **information** in it and assign to the **first node address**. In **step 4**, we assign insert to **first pointer** to make **newly inserted node** the first node of the **LIST**. The following **figure** shows the above mechanism:



One Way Link List Front Insertion

II. Middle Insertion: There are **two methods**, which can be used to **insert node** at the **middle** if a **one-way link list**.

Method 1: If the **nodes** of the **given list** are **unsorted** then we insert **node** in a **list** after a **given node**.

Method 2: If the **nodes** of the **given list** are **sorted** in some particular order then the **node** is inserted at a **particular position** so that the **sorted order** could be maintained.

Here is discussed the **second method**. The following **algorithm** inserts a **node** at a **proper place**.

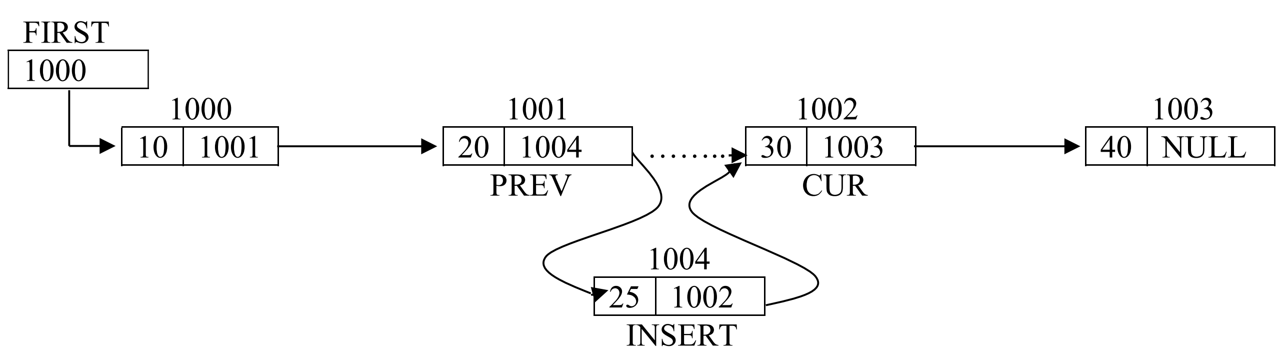
ALGORITHM: MIDDLE INSERTION (Prev, First, Cur, X, Insert)

This algorithm is used to insert a node at the proper location in as ordered **LIST**.

Step 1: [Set Prev to First]
Prev = First
Step 2: [Set Cur to Prev Link]
Cur = Prev -> link
Step 3: [Read value for insertion]
Read (X)
Step 4: [Search for a proper location]
Repeat step 5 & 6 ------ while (Cur -> data < X)
Step 5: [Set Cur as Prev]
Prev = Cur
Step 6: [Set Cur to Cur Link]
Cur = Cur -> link
Step 7: [Create a Node and set it as Insert]
Insert = (struct node *) malloc (sizeof (struct node))
Step 8: [Store information]
Insert -> data = X
Step 9: [Link with Cur]
Insert -> link = Cur
Step 10: [Link Prev with Insert]
Prev -> link = Insert
Step 11: [Finish]
Exit

In above **algorithm step 4**, **5** & **6** are used to find out a **proper position** for a **new node**. In **step 7** & **8**, we create a **new node**, assign its **address** to insert **pointer** and store **information**. In **step 9**, **10** & **11**, the **new node** is **linked** to the **current** and **previous nodes**.

The following **figure** shows the above mechanism:



One Way Link List Middle Insertion

III. End Insertion:

By **end insertion,** means to insert a node at the **end/last** of a **link list**. The algorithm is given as follow:

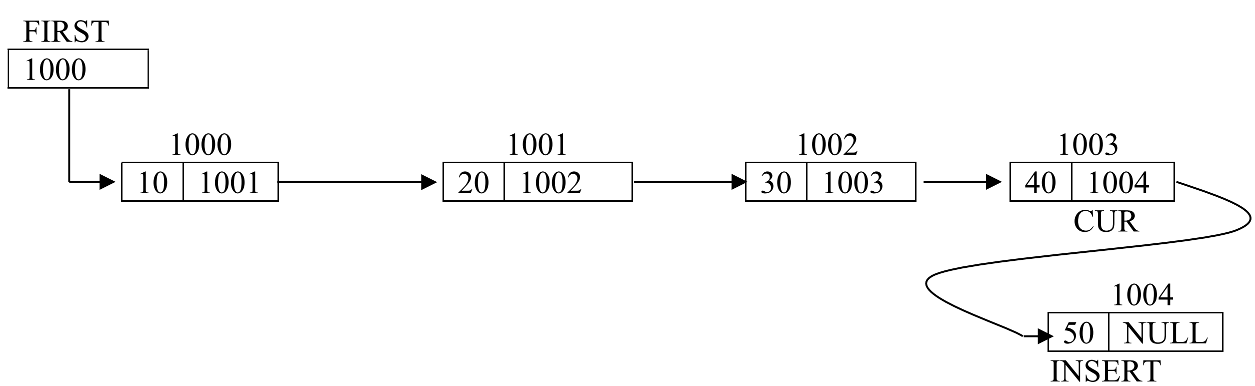
ALGORITHM: END INSERTION (Prev, First, Cur, X, Insert)

This algorithm is used to insert a node insert at the end of a **one-way link list**.

Step 1: [Set Prev pointer to First Node]
Prev = First
Step 2: [Set Cur to prev link]
Cur = Prev -> link
Step 3: [Read value for insertion]
Read (X)
Step 4: [Start loop to reach to the end of list]
Repeat step 5 & 6 ------- while (Cur -> link != Null)
Step 5: [Set Cur as Prev]
Prev = Cur
Step 6: [Set Cur to Cur -> link]
Cur = Cur -> link
Step 7: [Create a new node and set it to insert]
Insert = (struct node *) malloc (size of (struct node))
Step 8: [Store Information]
Insert -> data = X
Step 9: [Set Insert link as Null]
Insert -> link = null or Insert -> link = Cur -> link
Step 10: [Set Prev link as Insert]
Cur -> link = Insert
Step 11: [Finish]
Exit

In the above **algorithm step 2** & **3** are used to reach to the **end of link list**. When **end** of **list** is found then in **step 4**, **5** & **6**, we create a **new node**; assign its **address** to insert **pointer**, store **information** and set **insert pointer** as **NULL**, because **null link pointer** shows the **end** of the **list**. In **step 7**, the **new node** is **linked** to **previous node**.

The following **figure** shows the above mechanism.



One Way Link List End Insertion

2. Deletion Operation:

Let **LIST** be a given **link list** with **N** successive nodes and we want to **delete** a node **X** from a **link list**. Then we can **delete** that **node** from three different location of a **link list**, which is given as:

I. Front Deletion
II. Middle Deletion
III. End Deletion

I. Front Deletion: Front deletion is the easiest way to delete a node from a **link list**. Following is the algorithm is used to delete a node from **one-way link list** from the **front** or **start**.

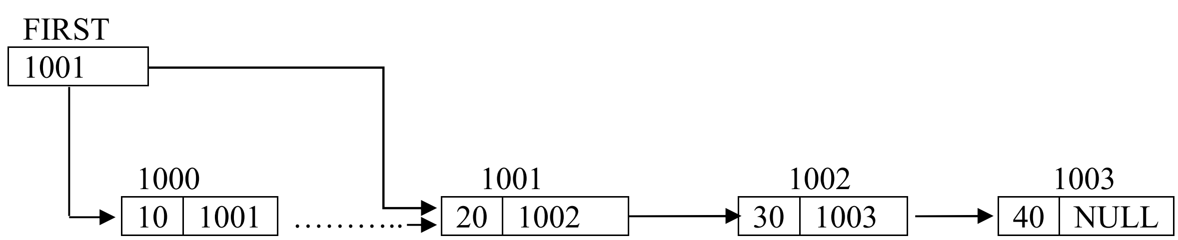
ALGORITHM: FRONT DELETION (Cur, First)

This algorithm is used to delete a node from the **front** of a **one-way link list**.

Step 1:	[Set Cur to First] Cur = First		Step 1:	[Set Prev to First] Prev = First
Step 2:	[Update First] First = Cur -> link	**OR**	Step 2:	[Set Cur to Prev link] Cur = Prev -> link
Step 3:	[Delete Node] Free (Cur)		Step 3:	[Update Prev] Prev -> link = Cur -> link
Step 4:	[Finish] Exit		Step 4:	[Delete Node] Free (Cur)
			Step 5:	[Finish] Exit

In the above **algorithm** in **step 1**, we store the **address** of **first node** in **current pointer**. In **step 2** we update the **first** to the **next node** to make the **second node** as **first**. In **step 3,** we **delete** the **node** by **free ()** function provided in **C** language.

The following figure shows the above mechanism.



One Way Link List Front Deletion

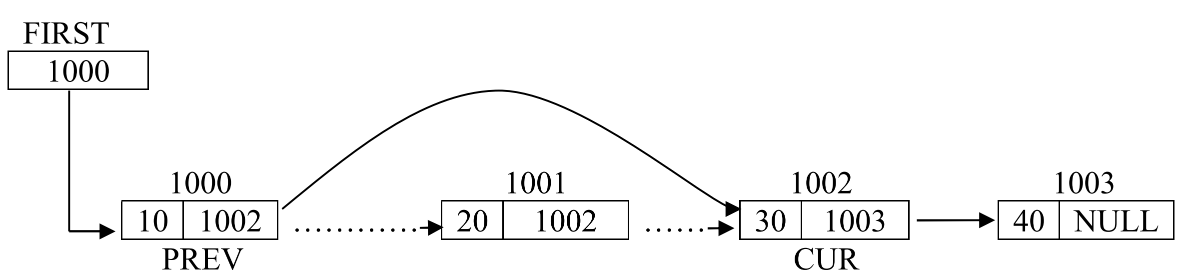
II. Middle Deletion: In **middle deletion** if we want to delete node **X** from a linked list which is not at the first or last location. For this we start searching for that node in a list. If found then we should update successor and predecessor and then delete the node. The following algorithm is used to delete a node from middle form **one-way link list**.

ALGORITHM: MIDDLE DELETION (Prev, Cur, First, X)
This algorithm is used to delete a node from middle form **one-way link list**.

Step 1: [Set Prev to First]
Prev = First
Step 2: [Set Cur to Prev link]
Cur = Prev -> link
Step 3: [Read the value of Node to delete]
Read (X)
Step 4: [Starting loop to search a node for deletion]
Repeat step 5, 6 & 7 While (Cur -> link != NULL)
Step 5: [Check nodes, if found delete and return]
If (X == Cur -> data) then
    Prev -> link = Cur -> link
    Free (Cur)
    Return
[End of If structure]
Step 6: [Set Cur as Prev]
Prev = Cur
Step 7: [Update Cur pointer]
Cur = Cur -> link
Step 8: [Not deleted]
Write ("Element not found to delete")
Step 9: [Finish]
Exit

In the above algorithm in **step 1**, **2** we assign **previous pointer** to **first node** and **cur** to the **next** of **first node**. In **step 4**, **5**, **6** & **7** we start **loop** for **searching** for the **node**, if it is found it is deleted and exited otherwise the **prev** is **updated** to **cur** and **cur** is **updated** to the **next node**.

The following **figure** shows the above mechanism.



One Way Link List Middle Deletion

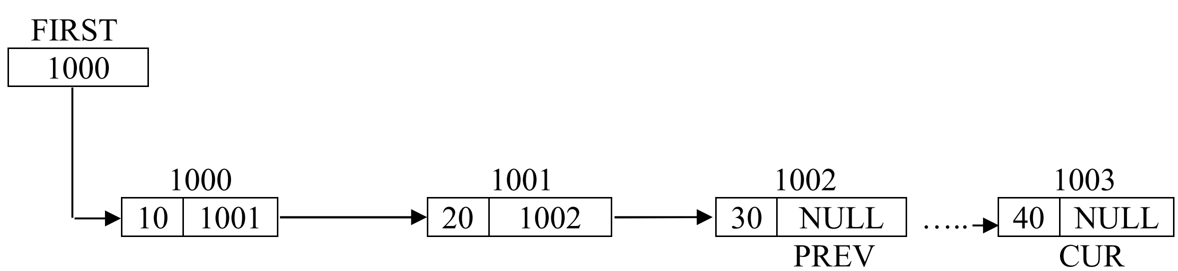
III. End Deletion: To delete the end node of a one-way link list first we have to reach to the end of the list.

ALGORITHM: END DELETION (Prev, First, Cur)
This algorithm is used to delete the end node of a one-way link list.

Step 1: [Set Prev to First]
Prev = First
Step 2: [Set Cur to the link of Prev]
Cur = Prev -> link
Step 3: [Start loop to reach to the end of the list]
Repeat step 4 & 5 ----- While (Cur -> link != NULL)
Step 4: [Set Cur as Prev]
Prev = Cur
Step 5: [Update Cur]
Cur = Cur -> link
Step 6: [Assign NULL to the Prev link]
Prev -> link = NULL or Prev -> link = Cur -> link
Step 7: [Delete Cur node]
Free (Cur)
Step 8: [Finish]
Exit

In the above **algorithm** in **step 1** & **2** we assign **prev** to **first node** and **cur** to the **next node** to the **prev**. In **step 3**, **4** & **5**, we reach to the **end** of the **list**. In **step 6** & **7**, we assign **Null** to **prev** to make it the **end node** and free **cur pointer** to delete the **end node**.

The following **figure** shows the above mechanism.



One Way Link List End Deletion

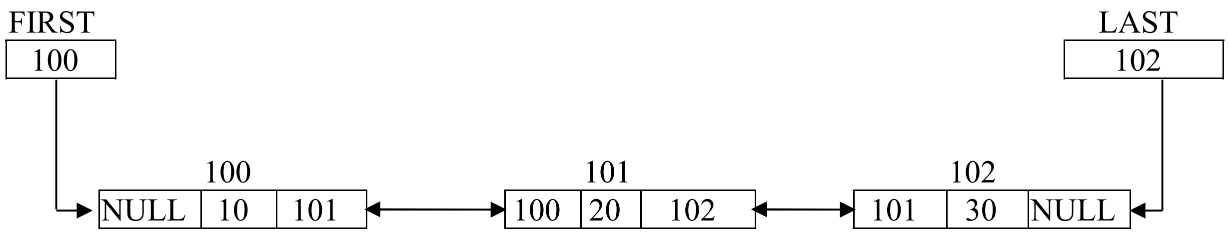
Week No. 11: TWO WAY LINKED LIST

Introduction to Two Way Linked List/ Double Linked List:
One of the big **disadvantages of one-way link list** is that only possible way to traverse the data is in the **list** is the **forward traversing**. There is no **backward traversing** of a **list** in one-way link list. The problem is handled by the **double** or **two-way link list**. In **two-way link list** each node is linked to both its **successor** and **predecessor**. The **two-way link list** is traversable from either direction i.e. **forward** and **backward**. On **two-way link list** a node is divided into three parts.

Information part: It contains the data of a node.
Right or next pointer: It points to the successor node.
Left or previous pointer: Pointer to the predecessor node.

Left	Information	Right

In the following diagram of **two-way link list** with three **nodes**, each **node** has **three parts**. The **left part** contains the **address** of **predecessor node** while the **right part** contains the **address** of the **successor node** and the **information part** contains the **information** about the **element**. There are **two pointers** also used i.e. **FIRST** and **LAST**. The **FIRST pointer** points to the **first** or **start node** of the **two-way link list** and the **LAST pointer** points to the **last** or **end node** of the **two-way link list**.



The following **example** explains the concepts of declaring **two-way link list** node using a **C** structure.

```
Struct node
{
        int data;
        Struct node *left;
        Struct node *right;
};
Struct node *start;
```

In the above **example**, **structure node** is defined with **three fields**:

- **Data** is the **int type** and is used to store **integer values** in the **node**.
- **Left** as a **pointer** to the **left node**. It is used to store the **memory addresses**. It contains the **memory address** of the **predecessor node** of **list**.
- **Right** as a **pointer** to the **right node**. It is used to store the **memory addresses**. It contains the **memory address** of the **successor node** of **list**.

Operations on Two Way Linked List with algorithms:

1. Insertion Operation:
Let us suppose **LIST** be a **Two-way link list** with **N** successive **nodes** and a **node insert** is to be inserted in a **link list**. The given node can be inserted in a link list in the following three locations.

I. Front Insertion
II. Middle Insertion
III. End Insertion

I. Front Insertion:

Following **algorithm** is used to **insert** a **node** in the **front/start** of the **Two-way link list**.

ALGORITHM: FRONT INSERTION (Cur, First, Insert)

Step 1: [Set cur as First]
Cur = first

Step 2: [Create a node and assign it to insert]
Insert = (struct node *) malloc (sizeof (struct node))

Step 3: [Store information]
Insert -> data = info

Step 4: [Assign cur to the insert right]
Insert -> right = cur

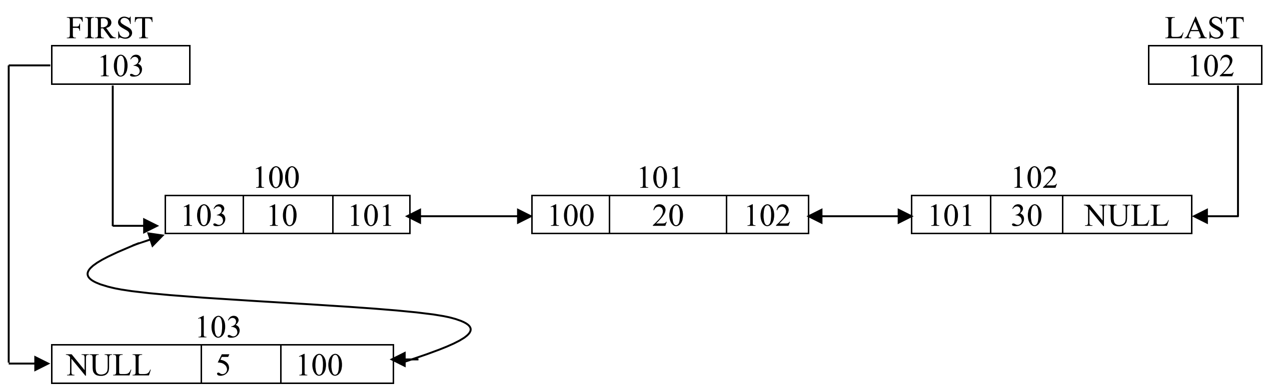
Step 5: [Assign Null to insert left]
Insert -> left = NULL

Step 6: [Assign insert to cur left]
Cur -> left = insert

Step 7: [Set first as insert]
First = insert

Step 8: [Finish]
Exit

The following **figure** shows the above mechanism.



II. Middle Insertion:

There are **two methods** that can be used to **insert node** at the **middle** of a **Two-way link list**.

Method 1: If the **nodes** of the **given list** are **unsorted** then we **insert node** in a **list** after a **given node**.

Method 2: If the **nodes** of the **given list** are **sorted** in some particular order then the **node** is inserted at a particular position so that the **sorted order** could be maintained.

We will discuss the **second method**. The following **algorithm** inserts a **node** at a proper place in **Two-way link list**.

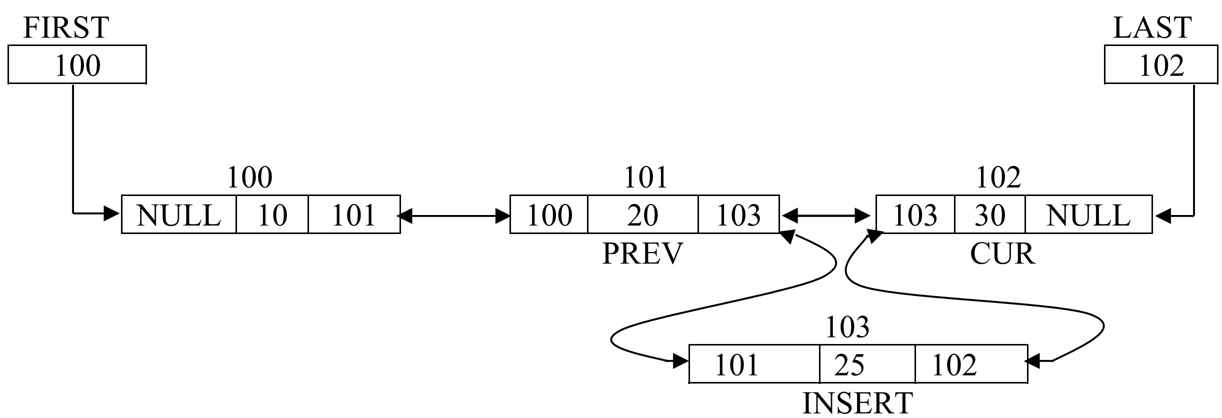
ALGORITHM: MIDDLE INSERTION (Prev, Cur, First, Insert)

This **algorithm** is used to insert a **node** at the **proper location** in an ordered **LIST** in **Two-way link list**.

Step 1:	[Set prev to first] Prev = first
Step 2:	[Set cur to prev right] Cur = prev -> right
Step 3:	[Read value for insertion] Read (X)
Step 4:	[Search for a proper location] Repeat step 5 & 6 ------- while (cur -> data<X)
Step 5:	[Set prev as cur] Prev = cur
Step 6:	[Set cur to cur right pointer] Cur = cur -> right
Step 7:	[Create a node and set it as insert] Insert = (struct node *) malloc (sizeof (struct node))
Step 8:	[Store information] Insert -> data = X
Step 9:	[Link with cur] Insert -> right = cur
Step 10:	[Link insert left pointer with prev] Insert -> left = prev
Step 11:	[Link cur & prev with insert] Cur -> left = insert Prev -> right = insert
Step 12:	[Finish] Exit

In the above **algorithm step 4**, **5** & **6** are used to find out a **proper position** for a **new node**. In **step 7** & **8**, we create a **new node**, assign its **address** to **insert pointer** and **store information**. In **step 9**, **10** & **11**, we link the **new node** to the **current** and **previous nodes** respectively.

The following **figure** shows the above mechanism.



III. End Insertion:

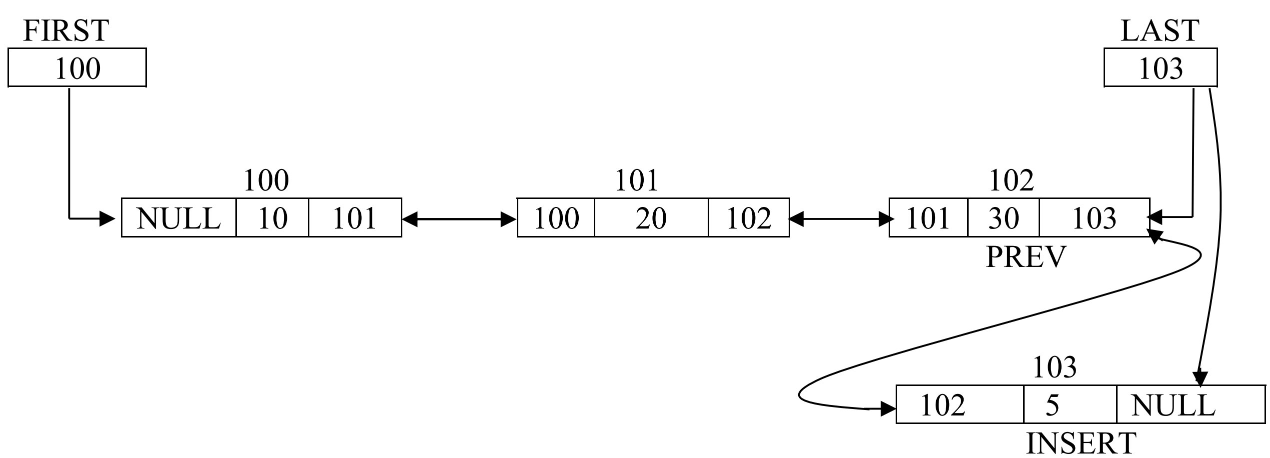
End insertion in the **two-way link list** is much easier than the **one –way link list** because we have a direct access to the **end node** of the **two-way link list** node.

Following is the **algorithm** for **end insertion** in **Two-way link list**.

ALGORITHM: END INSERTION (Prev, Last, Insert)

This **algorithm** is used to insert a node at the **end** of the **Two-way link list**.

Step 1: [Set prev as Last]
Prev = Last
Step 2: [Create a node and assign it to insert]
Insert = (struct node *) malloc (sizeof (struct node))
Step 3: [Store information]
Insert -> data = X
Step 4: [Assign prev to insert left]
Insert -> left = prev
Step 5: [Assign NULL to insert right]
Insert -> right = NULL
Step 6: [Assign insert to prev right]
Prev -> right = insert
Step 7: [Set Last as insert]
Last = insert
Step 8: [Finish]
Exit



2. Deletion Operation:

Let **LIST** be a given **Two-way link list** with **N** successive nodes and we want to delete a node **X** from **LIST**. Then we can delete that node from three different location of a **Two-way link list** which is given as:

I. Front Deletion
II. Middle Deletion
III. End Deletion

I. Front Deletion:

Front deletion is the easiest way to delete a node from a **link list**. Following is the **algorithm** which is used to delete a node from **Two-way link list** from the **front** or **start**.

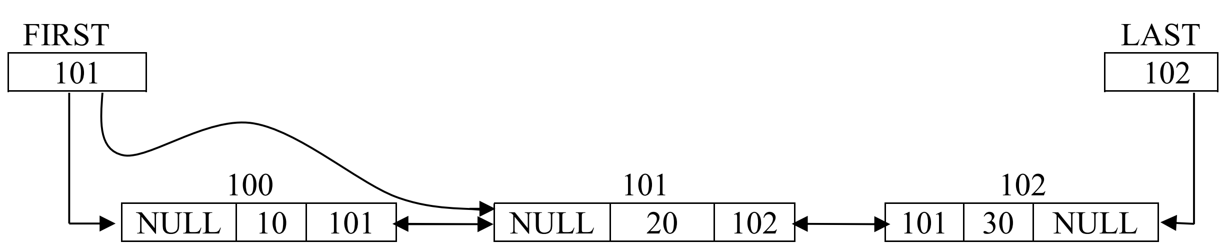
ALGORITHM: FRONT DELETION (Cur, First)

This algorithm is used to delete a node from the **front** of a **Two-way link list**.

```
Step 1:     [Set cur to first]
            Cur = first
Step 2:     [Update first]
            First = cur -> right
Step 3:     [Delete node]
            Free (cur)
Step 4:     [Set cur again as first]
            Cur = first
Step 5:     [Assign Null to cur left]
            Cur -> left = NULL
Step 6:     [Finish]
            Exit
```

In the above **algorithm** in **step 1**, we store the **address** of **first node** in **current pointer**. In **step 2**, we update the **FIRST** to the **next node** to make the **second node** as **first**. **In step 3**, we delete the node by **free ()** function provided in **C** language. In **step 4**, again we set the **current pointer** to point to **first node** and now **current node** is the **first node** in the **list**. In **step 5**, we assign **NULL** to **current left pointer** to make it **first node** of the **list**.

The following **figure** shows the above mechanism.



Two-way link list Front Deletion

II. Middle Deletion:

Let **LIST** be a **Two-way link list** with **N** successive nodes and we want to delete a node **X** from a **linked list** which is not at the **FIRST** or **LAST** location. For this we start searching for that node in the list. If found then one should update the **successor** and **predecessor node pointers**.

The following **algorithm** inserts a **node** at a **proper place** in **Two-way link list**.

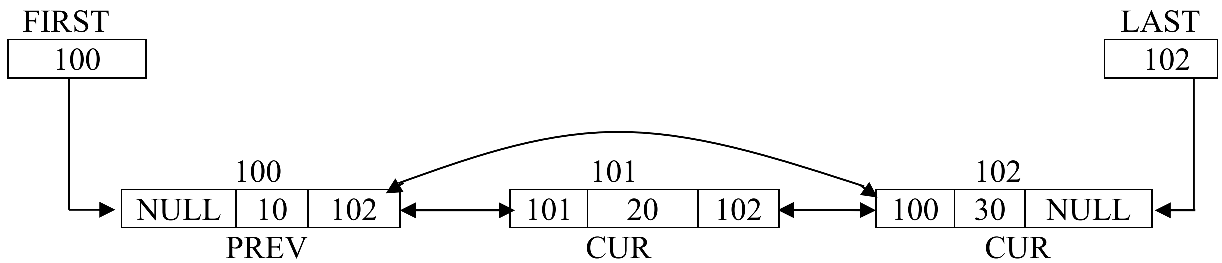
ALGORITHM: MIDDLE DELETION (Prev, Cur, First)

This **algorithm** is used to delete a **node** at the **proper location** in **Two-way link list**.

Step 1:	[Set prev to first]
	Prev = first
Step 2:	[Set cur to prev right]
	Cur = prev -> right
Step 3:	[Read the value of node to delete]
	Read (X)
Step 4:	[Starting loop for search a node deletion]
	Repeat step 5, 6 & 7 ----- while (cur -> right != Null)
Step 5:	[Check a node, if found, then delete and return]
	If (cur -> data ==X) then
	Prev -> right = cur -> right
	Free (cur)
	Cur = prev -> right
	Cur -> left = prev
	Return
Step 6:	[Set prev as cur]
	Prev = cur
Step 7:	[Update cur to right]
	Cur = cur -> right
Step 8:	[Finish]
	Exit

In the above algorithm in **step 1, 2**, we assign **previous pointer** to **first node** and **cur** to the **next** of **first node**. In **step 4**, **5**, **6** & **7**, we start loop for searching for the node, if it is found it is deleted and exited otherwise the **prev** is updated to **cur** and **cur** is updated to the **next node** at **right side**.

The following **figure** shows the above mechanism:



Two-way link list Middle Deletion

III. End Deletion:

End deletion is also very easy as **front deletion** in **Two-way link list**.

Following **algorithm** is used to **delete end node** of **Two-way link list**.

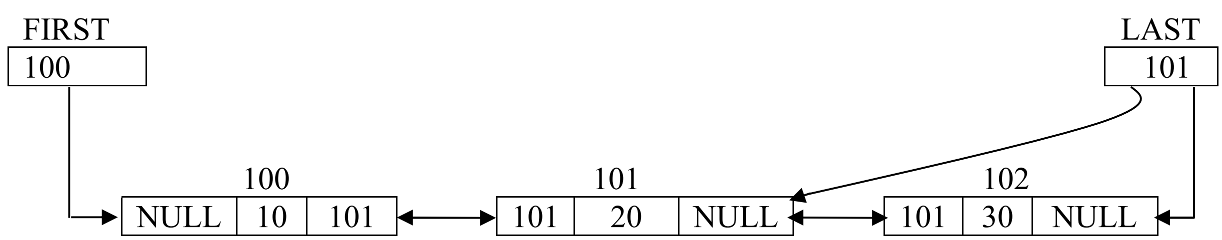
ALGORITHM: END DELETION (Cur, Last)

This **algorithm** is used to delete the **end node** of a **Two-way link list**.

Step 1: [Set cur to last]
Cur = last
Step 2: [Update last]
Last = cur -> left
Step 3: [Delete node]
Free (cur)
Step 4: [Set cur again as last/end]
Cur = last
Step 5: [Assign NULL to the cur right]
Cur -> right = NULL
Step 6: [Finish]
Exit

In **step 2** of the above **algorithm** we updated the **LAST** pointer to point the **2^{nd} last node** in the **list**. In **step 3**, memory is free occupied by the **last node**. In **step 4**, again set **current** pointer to **last node**, and now **current node** is the **last node** in the **list**. Therefore, assign **NULL** to **current right pointer**.

The following **figure** shows the above mechanism:



Two-way link list End Deletion

Week No. 12: TREES

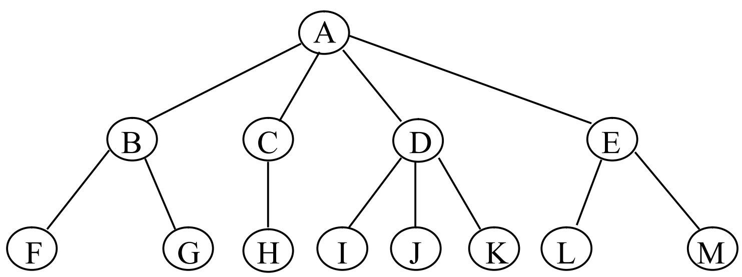
Introduction to Tree:
A **Tree** is a **non-linear** data structure. Each object of a **Tree** starts with a **root** and extends into several **branches**. Each **branch** may extend into other **branches**. **Tree** is mainly used to represent the data containing a **hierarchal relationship** between elements e.g. family tree, table of contents, organization chart of a company etc.

General Tree:
A **General Tree** (sometimes called a **tree**) is defined to be a **non-empty** finite set of elements called **nodes**, such that:

i. **Tree** contains a distinguished element called the **root** of the **tree**.
ii. The remaining elements of a **Tree** is an ordered collection of zero or more **Disjoint** (separate/disconnect) **Trees**.
iii. In a **Tree**, a **node** can have any number of children.

A typical **Tree** is shown below:

Figure 1:



Tree Terminology:
Family Relationships Terminology is frequently used to describe **relationship** between the **nodes** of a **tree**.

Node: An entry in a **Tree**.

Root Node: The node at the top of **Tree** e.g., in the above **figure 1: A** is the **root node**.

Parent: Those nodes that have either the child nodes or the child nodes along with one parent node is called **parent node** e.g., in the above **figure 1: B** is a **parent node**.

Children: The node that is directly connected to a parent node is called the **child node** e.g., in the above **figure 1: F** is a child of **B**.

Sibling: The nodes having same parent is called **sibling** or **brother nodes** e.g., in the above **figure 1: I, J** and **K** are sibling nodes because they have same parent **D**.

Sub tree: The **child node** of the **root** that has its own **child nodes** is called **sub tree** e.g., in the above **figure 1: B, C, D & E** are **sub trees**.

Level: The root of a **tree** has a **level 0** and the **level** of any other nodes in the **tree** is one more than the **level** of its **father** e.g., in the above **figure 1:** node **J** is on **level 2**.

Edge: The **line** drowns from a node of **tree** to a **successor** is called **edge** or **connection** between one node to another.

Path: A **sequence** of connected **edges** is called **Path**.

Leaf: The **node** with no **successor** is called **leaf node** or **terminal node** e.g., in the above **figure 1: M** is a **leaf node** because it has no **successor** (child nodes) **or** a node with no children.

Branch: A **path** ending on a **leaf** is called **Branch**.

Depth or Height: The maximum numbers of a node in a **branch** is called **depth** or **height** of the **tree** e.g., **Depth** or **Height** of the **tree** in the above **figure 1:** is **three**.

Degree: Maximum number of children possible for a **node** or number of **sub trees** of a **node** e.g. in the above **figure 1: degree** of **A** is **4**, **degree** of **D** is **3**, **degree** of **C** is **1** and **degree** of **F** is **0** etc.

Similar Trees: -
Two or more than two trees are said to be **Similar**, if they have the same shape. Consider the following two figures: 2 & 3:

Figure 2:

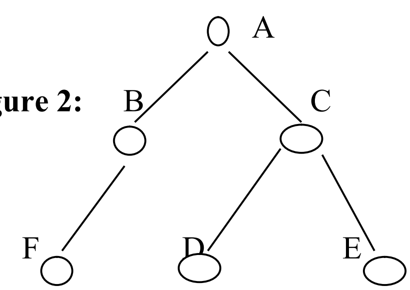
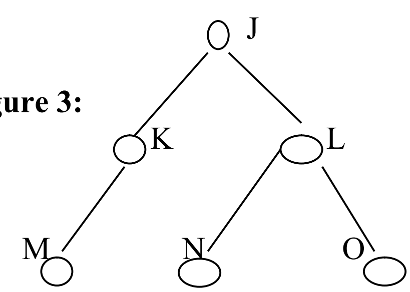


Figure 3:



In the above two **trees**, each **root node** has two children's nodes. The **rightmost child** has again two children's nodes in each **tree**. The **leftmost node** of each **tree B** and **K** has one child node each. It should be noted that **B** and **K** has one child node but these child nodes appear to the **left** of each node.

Hence the **shapes** of **each tree** are the **same** and thus they are known to be **Similar**.

Copies of Trees: -

Two or more than two **trees** are said to be **Copies** of **each other**, if they are **similar** as well as the **contents** of **each node** are also **same**.

Consider the following two **figures: 4 & 5:**

Figure 4:

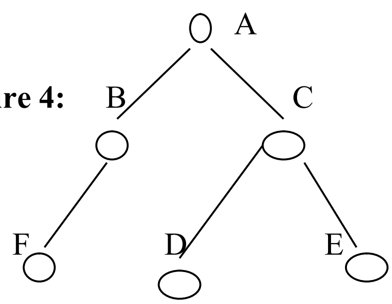
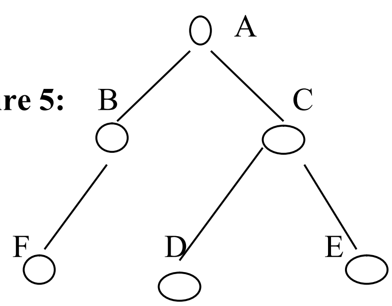


Figure 5:

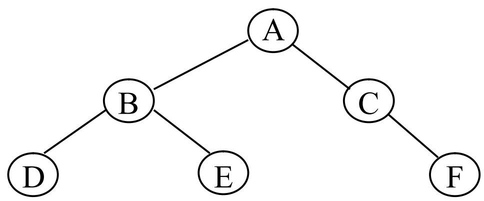


In the above **trees**, both the **trees** have **same shapes**. So, they are **similar trees**. But all the contents of the two **trees** are also the **same**, i.e. **A** is the root node in each **tree**, **B** and **C** lies at the **same level** of each **tree** and so on. So, they are also called **copies** of **each other**. It should be noted that the two **similar trees** cannot be **copies** of **each other**, but it is necessary that the two **copied trees** are always **similar** to each other.

Week No. 13: BINARY TREE AND BINARY SEARCH TREE

Binary Tree:

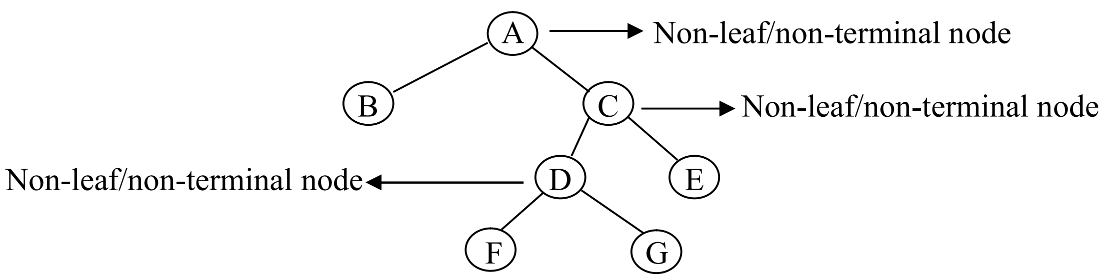
A **binary tree** is a **non-linear** data structure in which **each node** has only **0**, **1**, or **2** children (mostly has two children). **Binary Tree** can be **empty** or contains **one node** that is called **root** of the **tree**. Typically, the **child nodes** are called **left** & **right** nodes (Child).



Types of Binary Tree: A **Binary Tree** has the following types:

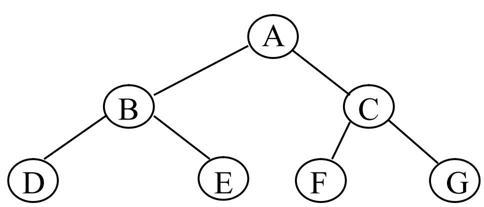
1. Strictly Binary Tree:

If every **non-leaf/non-terminal** node in a **Binary Tree** has non-empty left and right subtrees/children then such a **tree** is called **Strictly Binary Tree** e.g.



2. Full Binary Tree:

A **binary tree** is said to be a **Full Binary Tree**, if its leaf nodes are at same level & every node have two children **OR** a **Full Binary Tree** is a **tree** in which each **level ‘L’** has $\mathbf{2^n}$ elements including **last level** as ‘**n**’ represent number of **levels** e.g.



Level 0 $2^n = 2^0 = 1$ nodes

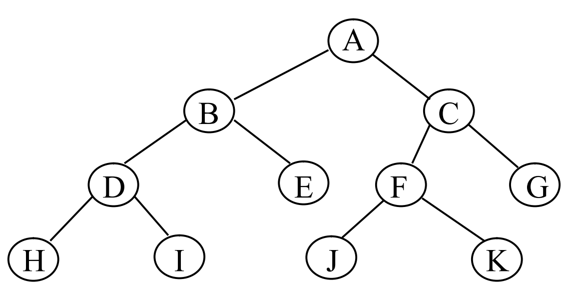
Level 1 $2^n = 2^1 = 2$ nodes

Level 2 $2^n = 2^2 = 4$ nodes

3. Complete Binary Tree:

A **Complete Binary Tree** is a **tree** in which each **level ‘L’** has $\mathbf{2^n}$ elements except the **last level**.

Example:



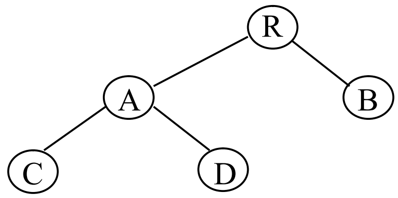
Level 0 $2^n = 2^0 = 1$ nodes

Level 1 $2^n = 2^1 = 2$ nodes

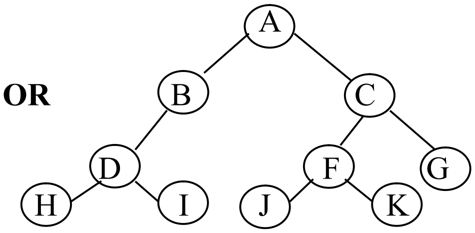
Level 2 $2^n = 2^2 = 4$ nodes

4. Extended Binary Tree:

A **binary tree** in which each child node of the root has either one or two children is called an **extended binary tree**. It is also called **2 – Tree**. The **node** that has children is called **internal node** and the **node** that have no children is called **external node** e.g. **C**, **D** & **B** are **external nodes** & **A** is **internal node**.

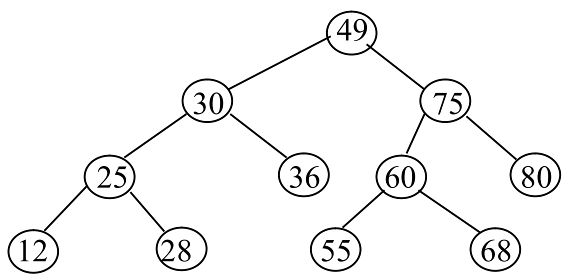


OR



Binary Search Tree:
A **Binary Search Tree** in which the **left child node** value of a **tree** is **less** than the value of its **root node** and the **right child node** value is **greater** than its **root node** value, is called **Binary Search Tree**.
Due to these properties, the elements traversed using **in order** will yield a **sorted list** of the elements of a **tree**. Main **advantage** of **Binary Search Tree** is that it is easy to create a **new tree**, and also some **operations** like insertion, searching and deletion are performed easily.

The following **tree** is an **example** of **binary search tree**: {49 30 36 25 75 60 55 68 80 28 12}



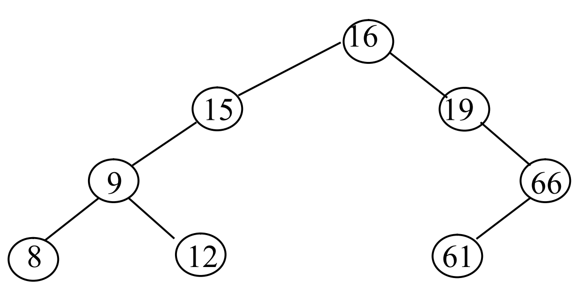
Operations on Binary Search Tree:
Following **operations** can be performed on a **binary search tree**:

1. Making/Constructing a Binary Search Tree:
If there is more than one element in a list then a **binary search tree** construct using the following steps:

i. Select **1st** element as root **R**.
ii. Compare **2nd** element with the root **R**, if it is less than the **root**, place it at **left node** of **root** otherwise place it at **right node** of **root**.
iii. Repeat **step 2** until all elements are placed.

For example: To construct a **Binary Search Tree** for the following data:

16 19 15 9 8 66 12 61



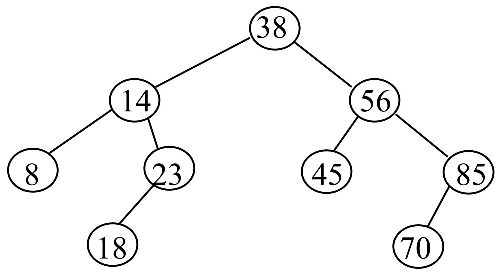
2. Insertion:

A **new node** is inserted after searching other nodes if the **new value** exists in any node then **insertion** operation fails otherwise the **new node** is inserted when searching terminates.

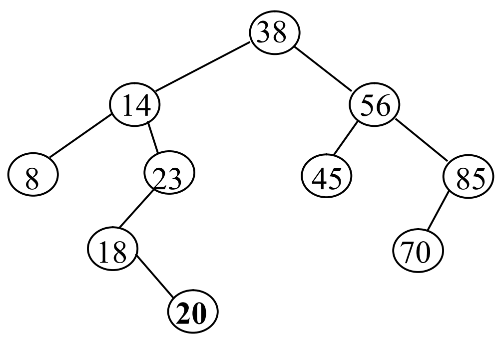
The following **algorithm** finds the location of an item in the **Binary Search Tree**:

- Compare **item** with **Root node** of **Tree**
  - If item < Root then: move to **Left child**
  - If item > Root then: move to **right child**
- Repeat the **above** until the **item** inserted into correct place.

Example 01: To insert the item **20** in the following **binary search tree**:

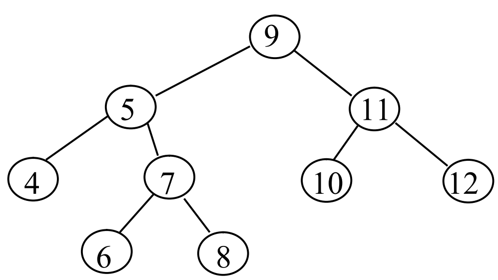


Before insertion

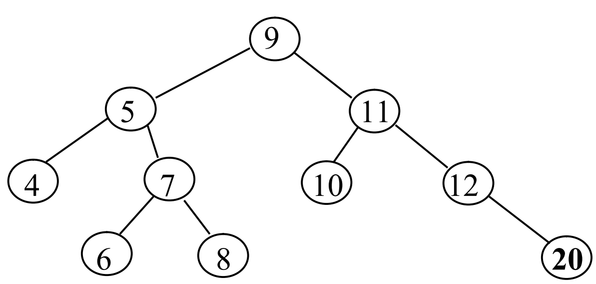


After insertion

Example 02: To insert the item **20** in the following **binary search tree**:



Before insertion



After insertion

3. **Deletion:**

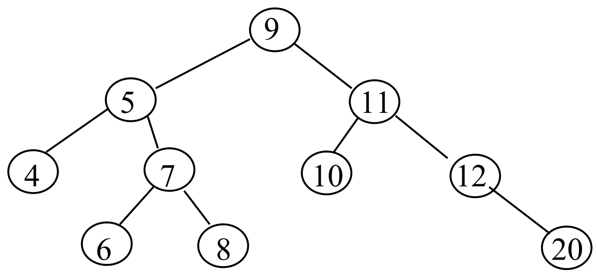
To **delete** a node involves **three conditions**:

I. A **node** with **no Children** i.e. leaf node
II. A **node** with **one Child**
III. A **node** with **two Children**

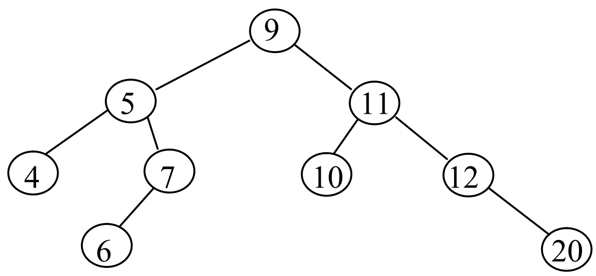
I. Leaf Node Deletion:

To delete a **leaf node** first of all, search the **tree** if the element to delete is found then check the position of **deleting node** and if it is the **leaf node** then just delete that **leaf node**.

For example: Delete item **8** from the following **binary search tree**:



Before deleting

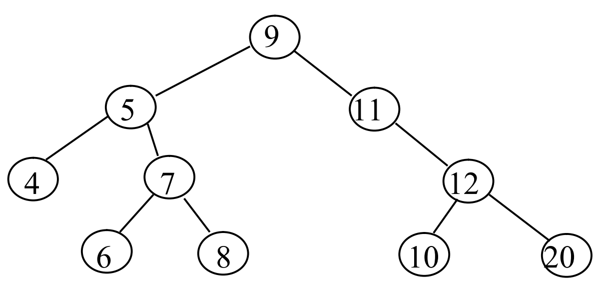


After deleting

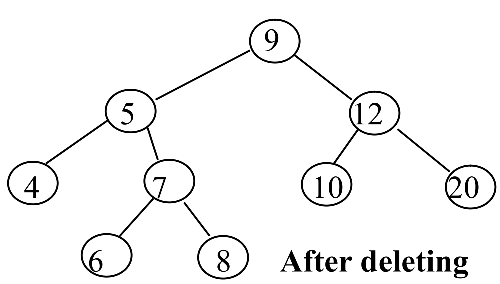
II. Deleting Node with one Child:

First of all, search the node if search is successful then check the position of **deleting node** and the position of its child if the **deleting node** is **left** of its parent node then child node of **deleting node** become left child of the parent of **deleting node** or vice versa (in **other words**, **deletion** of **non-leaf node** with one child, child node takes place of disposed node).

For example: Delete item **11** from the following **binary search tree**:



Before deleting



After deleting

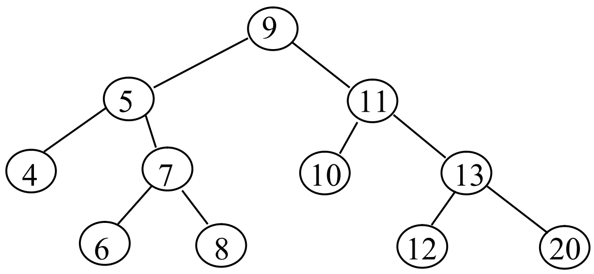
III. Deleting Node with two Children:

- Replace the **deleting node** with **largest element** of its **left sub tree**

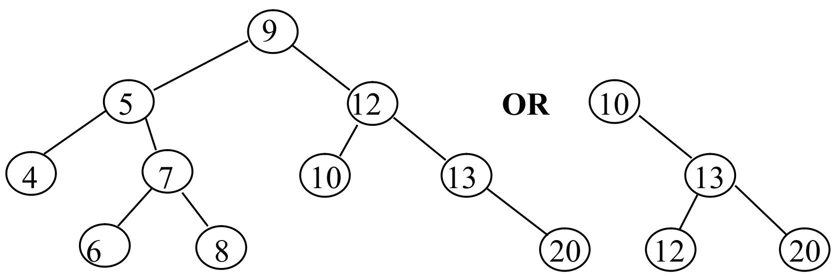
OR

- Replace the **deleting node** with **smallest element** of its **right sub tree**

For example: Delete item **11** from the following **binary search tree**:



Before deleting



After deleting

Week No. 14: TRAVERSING OF GENERAL & BINARY TREES

Traversing of General Tree:

Accessing the nodes of a **General Tree** exactly once is called **Traversing/Visiting** of a **Tree**.

There are **three basic ways** of accessing the nodes of a **General Tree**:

1. Level by Level Traversing
2. Pre-Order/Prefix Traversing
3. Post Order/Postfix/Suffix Walk Traversing

The **order** in which the nodes or elements of a linear list are visited in a **traversal** is clearly from first node to last node, however, there is no such natural linear order for the nodes of a **tree** so different orderings are used for **traversal** in different cases.

1. Level by Level Traversing:

In **level by level traversing**, first visit **level 0** / **root** then visit **level 1**, **level 2** and so on, from **left** to **right**. In **level by level traversing**, the following criteria have to follow:

i. Visit the **root**/**level 0**
ii. Visit the **first level** from **left** to **right**
iii. Visit the **next level** from **left** to **right** and **so on**.

2. Pre-Order/Prefix Traversing:

In **pre-order traversing**, **parent nodes** are always accessed before their **children nodes**. So, a **pre-order traversing** involves first processing the **root** then **traverses** the **siblings**/**children** from **left** to **right**. In **pre-order traversing**, the following criteria have to follow:

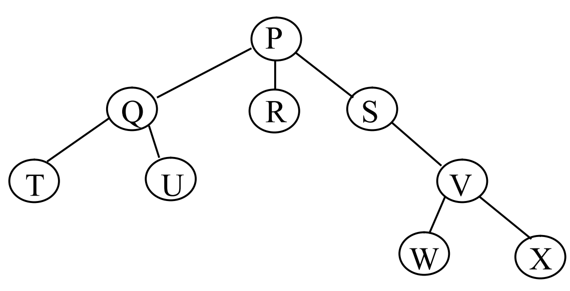
i. Visit the **root**
ii. Traverse **subtrees** from **left** to **right** in **pre-order** (means traverse **left subtree**, **middle subtree** and then **right subtree** in **pre-order**).

3. Post Order/ Postfix/Suffix Walk Traversing:

In **postfix traversing**, first traverses the **siblings**/**children** from **left** to **right** and then traverse the **root**. In **postfix traversing**, the following criteria have to follow:

i. Traverse the **left most terminal node/leaf node**.
ii. Traverse across its **siblings**.
iii. Traverse the **parent node**.

For example: Consider the following given **general tree**:



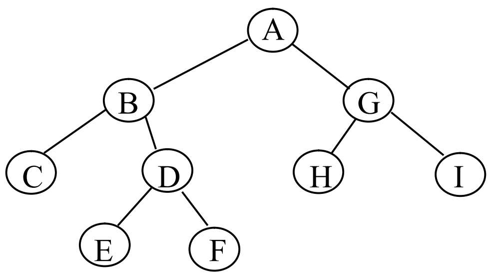
1. **Level by Level Traversing**: P Q R S T U V W X
2. **Pre Order Traversing**: P Q T U R S V W X
3. **Postfix Traversing**: T U Q R W X V S P

Traversing of Binary Tree:
In **traversing**, each node of a **binary tree** is accessed for processing exactly once. It is also called **visiting;** the following different methods are used to visit a **binary tree** i.e.

1. Pre-Order Traversal
2. In Order Traversal
3. Post Order Traversal

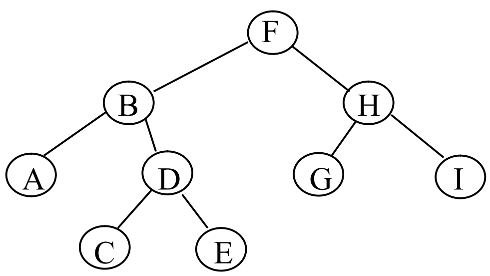
Note:
- If a **tree T** is **null** then the **empty list** is the **Pre-Order**, **In Order** and **Post Order** listing of **T**.
- If a **tree T** consist a **single node** then that node by **itself** is the **Pre-Order**, **In Order** and **Post Order** listing of **T**.

1. Pre-Order Traversal:
In **Pre-Order Traversal**, first **root** is processed then **left child** and then **right child**.
Example:



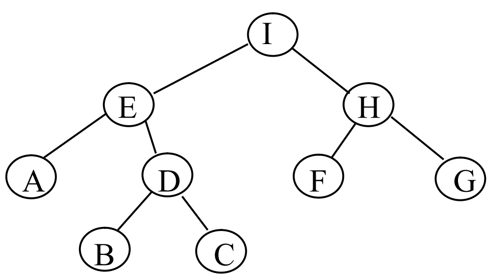
Pre-Order Traversal: A B C D E F G H I

2. In Order Traversal:
In **In-Order Traversal**, the **left child** is traversal first then the **root** and after it the **right child** is accessed.
Example:



In Order Traversal: A B C D E F G H I

3. Post Order Traversal:
In **post order traversal**, first **left node** is processed then **right node** and at the end **parent node** is processed.
Example:



Post Order Traversal: A B C D E F G H I

Notations and Expressions:

Polish Notations:

Named after **Polish** mathematician **Jan Lukasicwicz**.

There are three **Polish Notations**:

i. Polish Infix Notation
ii. Polish Prefix Notation
iii. Polish Postfix Notation

i. Polish Infix Notation:

Polish Infix Notation refers to the **notation** in which the **operator symbol** is placed **between** its two **operands**.

For example: i. A + B, ii. C – D * E, iii. A * (B + C) etc.

ii. Polish Prefix Notations:

Polish Prefix Notation refers to the **notation** in which the **operator symbol** is placed **before** its two **operands**.

For example: i. +AB, ii. – C*DE, iii. *A+BC etc.

iii. Polish Postfix Notation:

Polish Postfix Notation refers to the **notation** in which the **operator symbol** is placed **after** its two **operands**.

For example: i. AB+, ii. CDE*–, iii. ABC+* etc.

Inter Conversion of Notations:

1. Infix to Prefix:

Convert the following **Infix Notations** to the **Prefix Notations**:

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

ii. A + (B * C)
= A + [*BC]
= +A*BC

iii. (A + B)/(C – D)
= [+AB]/[– CD]
= /+AB–CD

2. Infix to Postfix:

Convert the following **Infix Notations** to the **Postfix Notation**:

i. (A + B) * C
= [AB+] * C
= AB+C*

ii. A + (B * C)
= A + [BC*]
= ABC*+

iii. (A + B)/(C – D)
= [AB+]/[CD–]
= AB+CD–/

3. Infix to Prefix & Postfix:

Convert the expression ((a + b) + c * (d + e) + f) * (g +h) to a **Prefix** expression & **Postfix** expression:

To Prefix:

((a + b) + c * (d + e) + f) * (g +h)
= ([+ab] + c * [+de] + f) * [+gh]
= ([+ab] + [*c+de] + f) * [+gh]
= ([++ab*c+de] + f) * [+gh]
= [+++ab*c+def] * [+gh]
= *+++ab*c+def+gh

To Postfix:

((a + b) + c * (d + e) + f) * (g +h)
= ([ab+] + c * [de+] + f) * [gh+]
= ([ab+] + [cde+*] + f) * [gh+]
= ([ab+cde+*+] + f) * [gh+]
= [ab+cde+*+f+] * [gh+]
= ab+cde+*+f+gh+*

Week No. 15: GRAPH

Introduction to Graph:

Tree data structure is used to represent the **one to many** relationships. In **real life**, we frequently come across the problem can be best described by **many to many** relationships. Such a problem cannot be solved using **tree** or other **data structures**. To solve this problem, use a non-linear data structure, called **graph**.

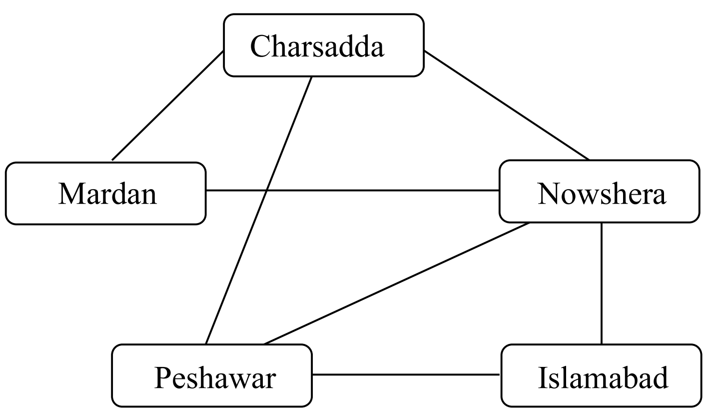
A **graph** is a non-linear data structure which is made up of sets of **nodes** and **lines**. **Nodes** are called **Vertices** or **Points** and **lines** are called **Edges** of **arcs**. **Lines** are used to connect **vertices** with each other. An **edge** of a **graph** is represented as follow:

e = [u, v]

'**u**' and '**v**' denote the **start** and **end nodes** of an **edge** '**e**'. They are also called **head** and **tail nodes** of **edge** '**e**'.

Graphs are used to represents essentially any relationship. **Graphs** are used to study the problems in a wide variety of areas including computer science, electrical engineering, chemistry etc. for **example** it is used to represent and study transport networks, communication networks and electrical circuits.

In **transportation networks**, **graph vertices** represent the **location** between which people or goods can be moved. **Location** may be **cities**, **airports**, **terminals** etc. The **edge** represents **path** between the **vertices** which may be **roads**, **railway tracks** etc., through which the communication between **cities** takes place. Following **graph** represents a **transport links** between **cities**. The **vertices** represent the **city** and the **edge** represents the **roads** between these **cities**.

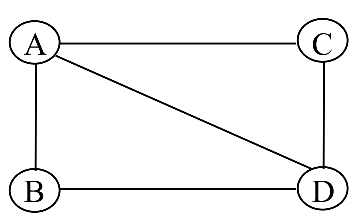


The above **graph** has **5 vertices** and **7 edges**.

Graph Terminologies:

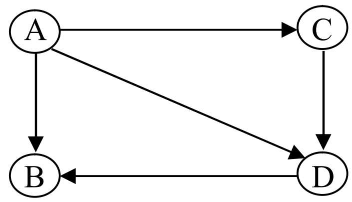
1. Degree of a Node:

The number of **edges** a **node** contains is called the **degree of the node**. For **example**, in the following figure **A** has a **degree 3**, **B** has a **degree 2**, **C** has a **degree 2** and **D** has a **degree 3**.



A **node** that has **0 degree** is called **Isolated Node**. And a **graph** having only **one isolated node** is called **Null Graph**.

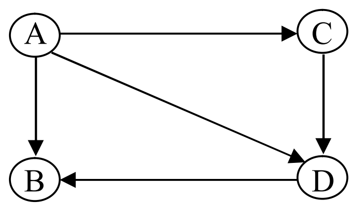
Out – Degree & In – Degree:
The **number** of **edges** beginning from a **node** is called **out – degree** of the **node**. For **example** in the following figure: **A** has **3 out – degree**, **C** has **1 out – degree**, **D** has **1 out – degree** and **B** has **0 out – degree**. A **node** having **0 out – degree** is called **terminal node or leaf node** and other nodes are called **Branch Nodes**.



The **number** of **edges** ending at a **node** is called **in – degree** of the **node**. In the above **graph** shown **in – degree** of **A** is **0**, **B** has **in – degree 2**.

The **sum** of the **out – degree** and **in – degree** is the **Total Degree**. The **total degree** of a **loop node** is **2** and that of **isolated node** is **0**.

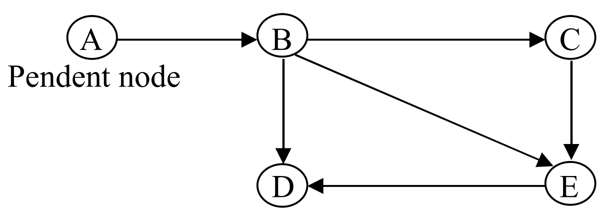
2. Source & Sink Nodes:
The **node** that has a **positive out – degree** but **0 in – degree** is called **Source Node**. In the following figure **A** is a **source node** because it has **positive out – degree 3** and **0 in – degree**.



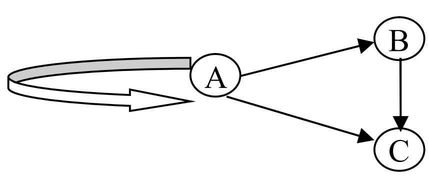
The **node** that has **0 out – degree** but have **positive in – degree** is called **Sink Node**.

For **example,** in above figure **B** is a **sink node** because it has **0 out – degree** and **positive in – degree 2**.

3. Pendent Node:
A **node** is said to be a **pendent node** if it has **total degree** equal to **1**. In the below figure **A** is a **pendent node** because it has **out – degree 1** and **in – degree 0**, so its **total degree** becomes **1**. All the other **nodes** have more then **1 total degree** so they are not **pendent nodes**.

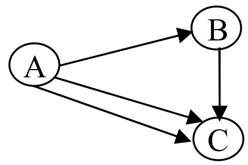


4. Loop Edge:
An **edge 'e'** is said to be a **loop edge** if the **same node** is its **tail** and **head**. A **loop edge** is shown in the following figure:



5. Multiple Edges:

A **graph** is said to have **multiple edges**, if it has **more** than **one edge** have the **same tail** and **head nodes**. An **example** of **multiple edges** is given below:



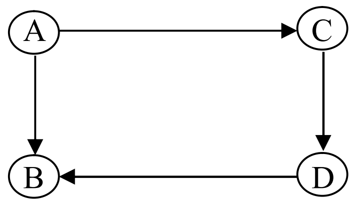
The **edges** that have the **same tail** and **head nodes** are known as **Parallel Edges**.

6. Path & Length of Graph:

A **list** of **nodes** of a **graph** where **each node** has an **edge** from it to the **next node** is called **Path**. It is written as a **sequence** of **nodes** u_1, u_2, u_3, u_4……….u_n.

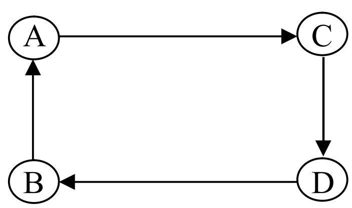
A **path** which repeats **no node** is known as the **Simple Path**. A **path** is usually assumed to be a **simple path** unless otherwise defined. For **example**, in the following **figure**, a **path** from **A** to **B** is a **Simple Path**.

The maximum number of **edges** in a **path** of a **graph** is called **Length** of the **Graph**. The **length** of a **path** which consisting of '**n**' number of **nodes**, is **n – 1**. For **example**, in the following **figure**, a **path** from **A** to **B** has **two nodes** so the **length** of that **path** is **n – 1** = 2 – 1 **= 1** and a **path** from **A** to **C**, **D**, **B** has **four nodes** so the **length** of that **path** is **n – 1 =** 4 – 1 **= 3**. So, the **Length** of the following **Graph** is **3**.

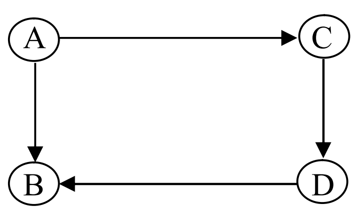


7. Cyclic & Acyclic Path:

A **path** which **starts** and **ends** at the **same node** is called **Cyclic Path**. In other words, a **path** from a **node** to **itself** is called **Cyclic Path**. It is also known as **circuit**. The **length of a cycle** must be at least **1**. Following figure is an **example** of **Cyclic Path**:



A **path** in which **start node** & **end node** are different is called **Acyclic Path**. The following **figure** is an **example** of **Acyclic Path**:

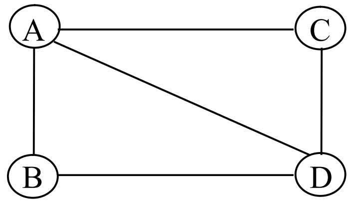


The **directed graph** that has **no cycles** is called **Acyclic Graph**. A **directed acyclic graph** is also referred to as **DAG** (**Directed Acyclic Graph**).

Types of Graph:

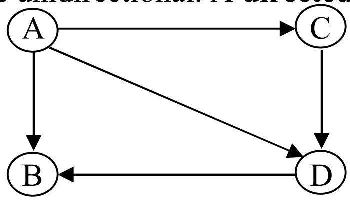
1. Undirected Graph:

An **undirected graph** has edges that have no directions. An **undirected graph** is also called **undigraph** e.g.



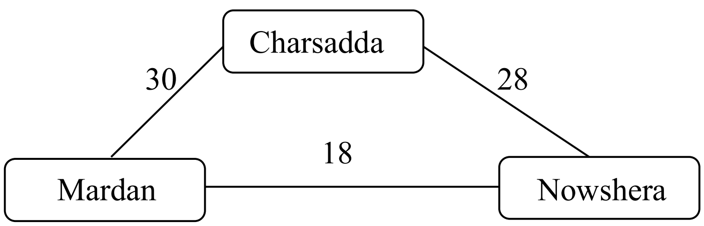
2. Directed Graph:

A **directed graph** has edges that are unidirectional. A **directed graph** is also called **digraph** e.g.



3. Weighted Graph:

A **graph** which has a **weight** or **number** associated with **each edge** is called a **weighted graph**. **Weight** of an edge is sometimes called its **cast**. The **weight** of edge usually represents some conditions or situations. For **example**, in the following **weighted graph**, the **weights** represent the distance between the cities.

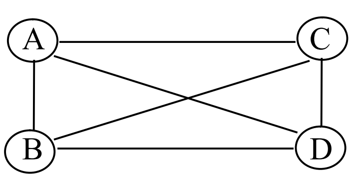


An **edge** of a **weighted graph** is represented as: **e** = [u, v, w]
'**u**' and '**v**' represents the **start** and **end node** of an **edge** where '**w**' represents the **weight** of an **edge** '**e**'.

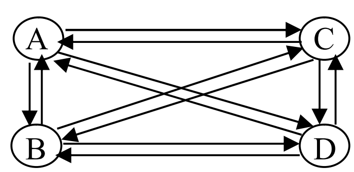
4. Complete Graph:

A **graph** is said to be a **complete graph** in which every **vertices** or **node** is connected to each other or a **graph** in which there is an **edge** between every **pair of vertices**. For **example,** if a **graph** has '**n**' nodes, then each **node** has (**n - 1**) **total degree** i.e. it is connected with **n – 1** nodes, so the **number** of **edges** can be calculated by the **formula**: **e = n (n – 1)/2.** Where **n** is the **total numbers** of **nodes** in a **graph** and **e** is the **number** of **edges**. For **example**, in the following figure **4 nodes** are connected, so the **numbers** of **edges** are:

e = n (n -1)/2		e = n (n -1)
= 4 (4 - 1)/2	**OR**	= 4 (4 - 1)
= 4 (3)/2		= 4 (3)
= 12/2		= 12
So e = 6		So e = 12



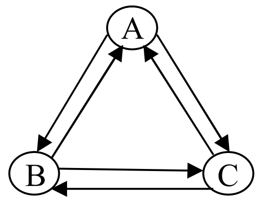
- Undirected Complete Graph



- Directed Complete Graph

5. Regular Graph:

A **graph** in which **each node** has equal **total degree** is called **regular graph**. Consider the following **figure** in which there are three nodes. **Each node** of them has equal **in – degree** & **out – degree** and **total degree** of the nodes is also **equal**. Hence the **graph** is said to be a **regular graph**.

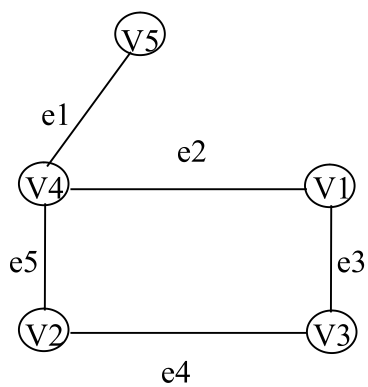


6. Isomorphic Graphs:

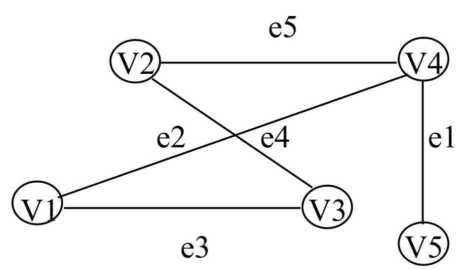
Two graphs are said to be **isomorphic** if they have **same** behavior in terms of **graphical property**. All the **edges** of the **two graphs** must be incident at their corresponding nodes. The conditions for **isomorphism** are:

1. The **number** of **nodes** in **two graphs** must be **same**.
2. There must be the **same number** of **edges** in the **two graphs**.
3. All the corresponding nodes of the **two graphs** must have **same** in – degree and out – degrees.

For **example,** in the following **two graphs** there are **5 nodes**: **v1**, **v2**, **v3**, **v4**, **v5** and **5 edges** in the **first graph** and in the **second one** there also **5 nodes** and **5 edges**. The **behavior** of the **two graphs** is **same**, because **e1** lies in **v4** and **v5**. Similarly, **e1** lies in **v4** and **v5**. So, in **graph 1** and **2 numbers** of **nodes** are **equal**, the **numbers** of **in – degree** and **out – degree** of all corresponding nodes are **same**. So, they are **isomorphic graphs**.

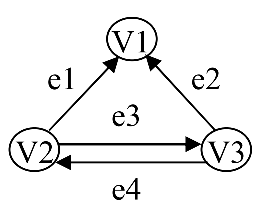


Graph 1

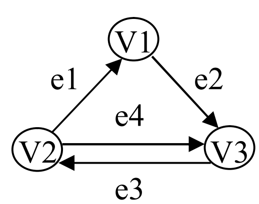


Graph 2

In **graph 3** & **4**, the **number** of **edges** and **nodes** are **equal**. Also, the **numbers** of **in – degree** and **out – degree** of **v2** are **same** in **both graphs**. But **v1** and **v3** have not **same in – degree** and **out – degree** in **both graphs**. Hence **graph 3** & **4** are not **isomorphic**.



Graph 3



Graph 4

Graph Representation:
Graphs are **unstructured**. One **vertex** in a **graph** might be **adjacent** to every other **vertex**. Similarly, a **vertex** might be **adjacent** to **just** one **vertex**. This **property** of **adjacency** is used to represent **graph** in **computer memory**.

Link representation of graph or Adjacency List:
Let **figure G** be a **directed graph** with **5 nodes**. The following **table** shows **each node** in **figure G** followed by its **adjacency list** which is its **list** of **adjacent nodes**, also called its **successors** or **neighbors**.

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

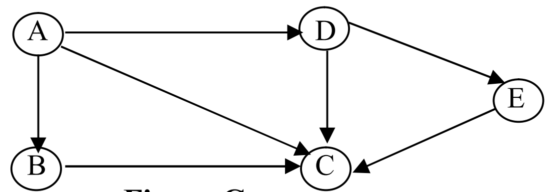
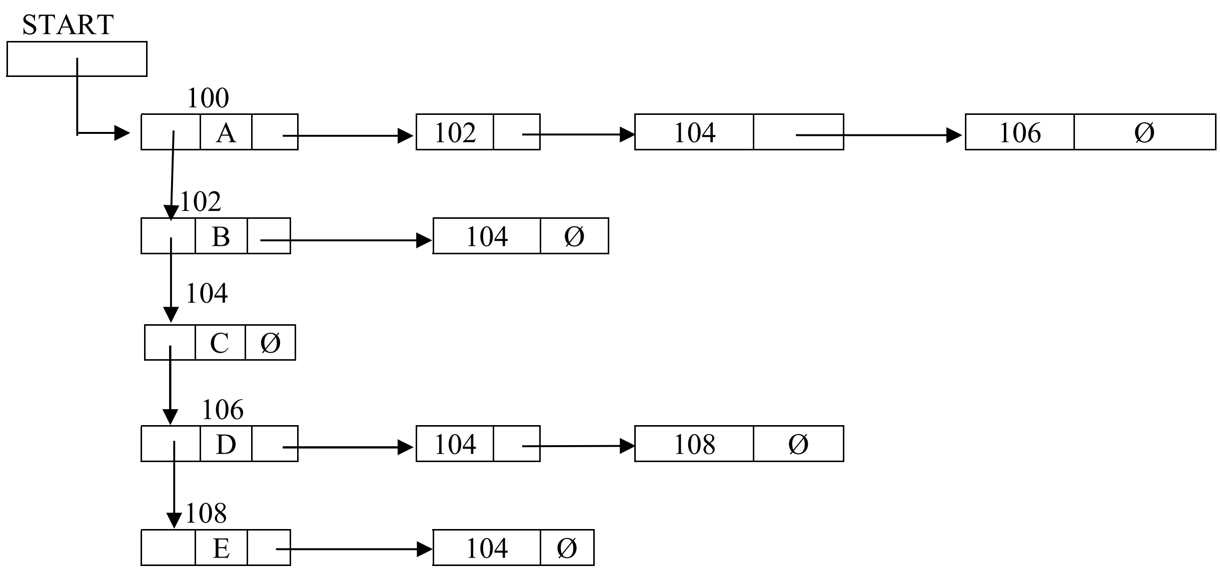


Figure G

Following **figure** shows a **diagram** of a **linked representation** of **figure G** in **memory**. Specifically, the **linked representation** will contain **two lists**, a **node list** and **edge list**.



A. **Node List:** Each **element** in the **list NODE** will correspond to a **node** in **graph**, and it will be a **record** of the form.

NODE	NEXT	ADJ		

Here **NODE** will be the name of **key value** of the **node**, **NEXT** will be a **pointer** to the **next node** in the **list NODE** and **ADJ** will be a **pointer** to the **first element** in **adjacency list** of the **node**, which is maintained in the **list EDGE**. The **shaded area** indicates that there may be **other information** in the **record** like **in – degree**, **out – degree** etc.

B. **Edge List:** Each **element** in the **list EDGE** will correspond to an **edge** of **graph** and will be a **record** of the form.

DEST	LINK		

The field **DEST** will **point** to the **location** in the **list NODE** of the **destination** or **terminal node** of **edge**. The field **LINK** will **link** together the **edges** with the **same initial node**, that is, the **nodes** in the **same adjacency list**. The **shaded area** indicates that there may be **other information** in the **record** corresponding to the **edge**, such as **weight** etc.

Week No. 16: REVISION

Revision of the complete course.

→ THE END ←