12|12|2023

Unit : 3

Graph : ★ Set of Vertices and edges.
★ Cyclic in Nature.

★ Can be implemented by Matrix (2D Array) or linked list

Tree : ★ Set of nodes and edges.
★ Non-Cyclic in Nature.

NOTE : Every tree is a graph, but not every graph is a tree.

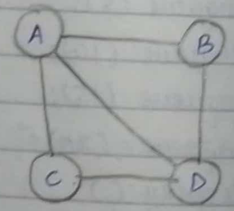
13|12|23

Representation of graph in memory :-

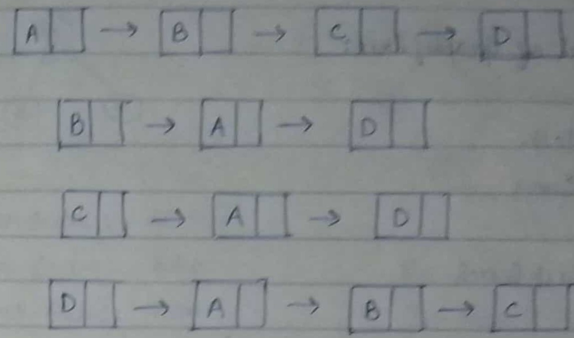
→ Matrix (2D Array) [Used for Dense Graph]
→ Linked List [Used for Sparse Graph]

X	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

Matrix Representation



Graph



Linked List Representation

Traversal of Graph / tree :

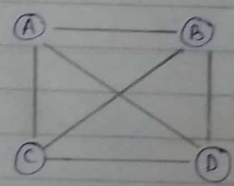
1) DFS : Stack data structure (Depth first Search)

2) BFS : Queue data structure (Breadth first Search)

9.1.24

Representation of Graph in Memory:

- Matrix (2D Array)
- Linked List

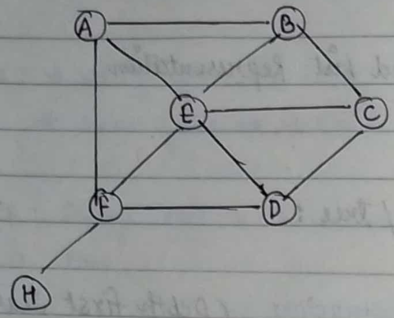


Traversal of Graph / Tree :

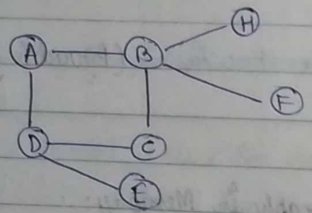
→ DFS = Stack

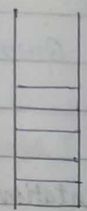
→ BFS = Queue

★ Depth First Search.



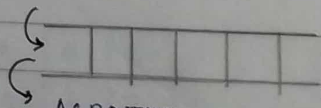
Eg:





ADECBHI

BFS :



ACBDIHE

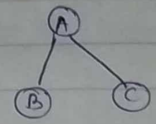
(followed queue concept)

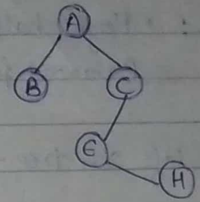
Tree Traversal :

★ Pre Order : Root L R

★ In Order : L Root R

★ Post Order : LR Root





Pre-Order : ABC

In-Order : BAC

Post-Order : BCA

Pre-Order : ABCEH

In-Order : BAEHC

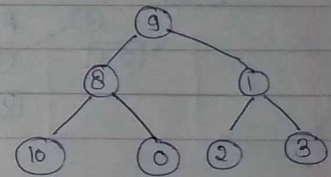
Post-Order : BHECA

BST (Binary Search Tree) i) (0-2) childs.

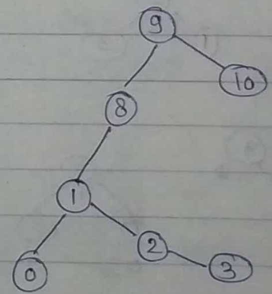
ii) R< Left

R> Right

Binary Tree



Eg: 9,8,1,2,3,10,0.



Pre: 9, 8, 1, 0, 2, 3, 10.

In: 0, 1, 2, 3, 8, 9, 10 (Ascending Order Always)

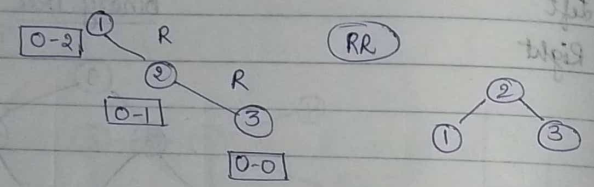
AVL Tree : • Height balanced.
Balanced factor: 0, +1, -1.

★ Height of left sub tree − Height of right sub tree.

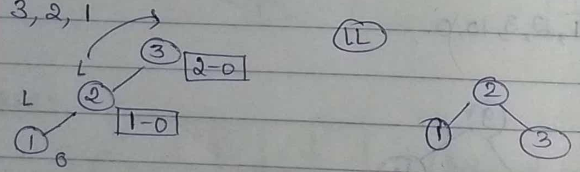
Rotations :

→ R,R
→ R,L
→ L,L
→ L,R

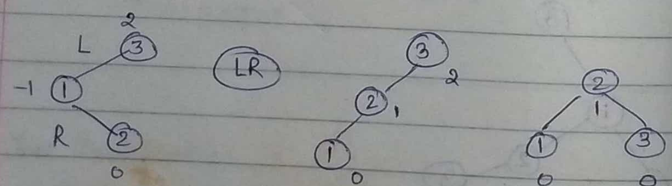
Example 1, 2, 3



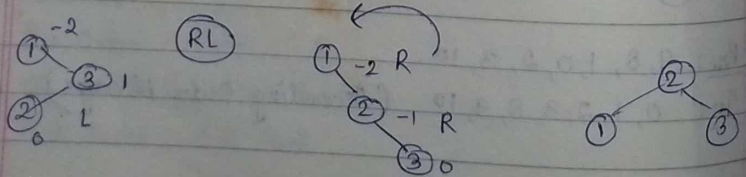
Example: 3, 2, 1



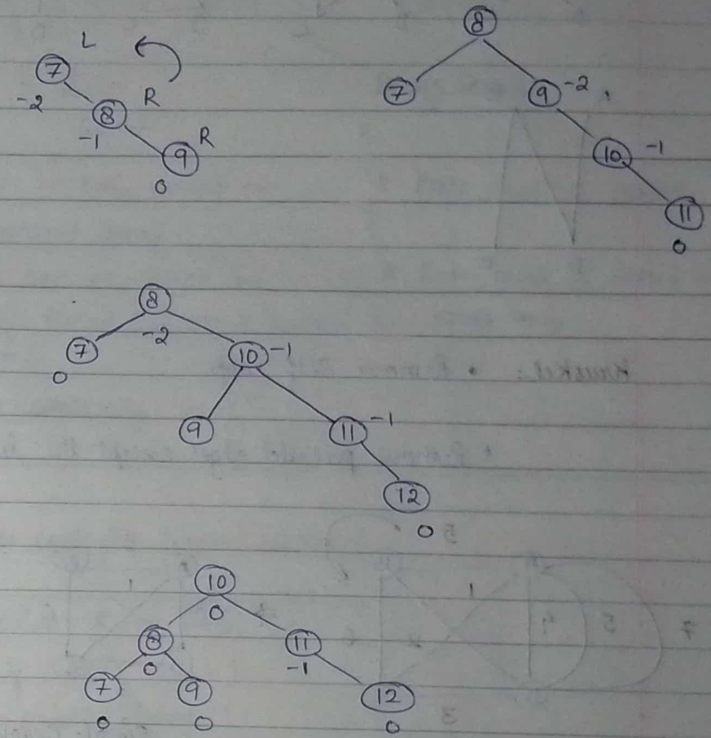
Example:



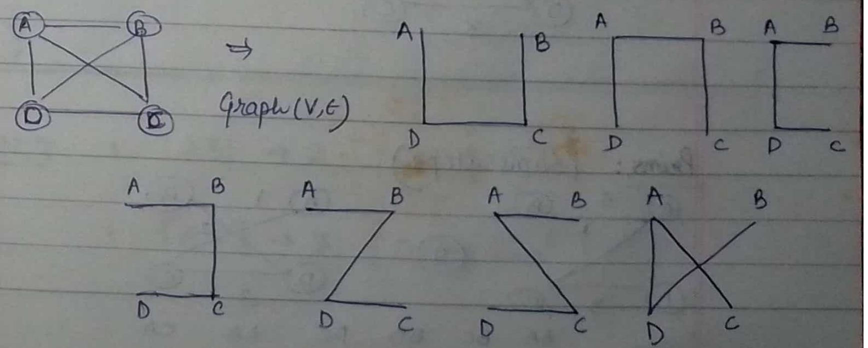
Example:

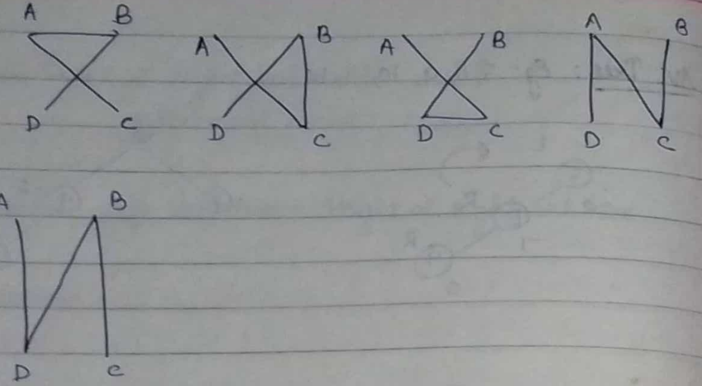


AVL Tree : Eg: 7, 8, 9, 10, 11, 12.



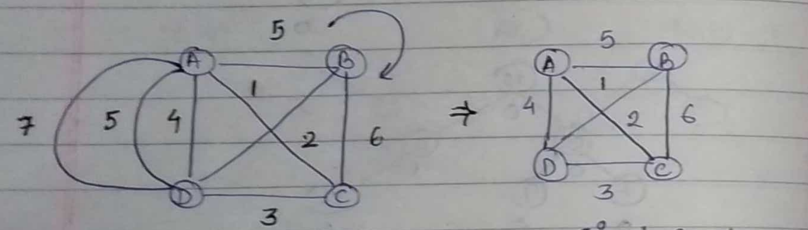
Spanning Tree : Subgraph of Graph G(V, E) that contains all the vertices with minimum no. of edges.



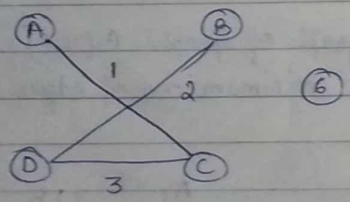


Kruskal: • Remove Self loop.

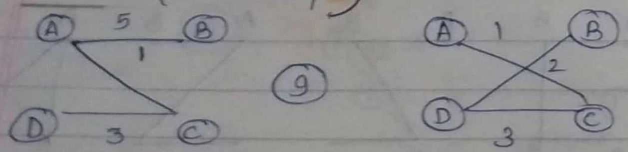
• Remove parallel edges except the least one.



Simple Graph
(min. edge choose)



Prims: (Same steps)



BA	BC	BD	DC	DA	CA
5	6	2	3	4	1

10|1|24

Searching

Linear Search | Binary Search

Linear Search	Binary Search
★ (Works on both sorted or unsorted data)	★ Data must be sorted.
★ Data can be stored in array, linked list, stack, queues, etc.	★ Data must be stored in array only.
★ Time complexity: $O(n)$.	

Pseudo code for linear search:

```
for( i=0; i<n; i++)
{ if (a[i] = = x).
  { printf ("found");
    exit();
  }
else
{ printf ("not found");
  }
}
```

NOTE: • $\lfloor 2.5 \rfloor \rightarrow 2$ [Floor funcⁿ]

• $\lceil 2.5 \rceil \rightarrow 3$ [Ceiling funcⁿ]

PAGE NO.:
DATE: / /20

Time Complexity for Binary Search :

n elements
↓
$(n/2)$
↓
$(n/2)/2 = n/2^2$
↓
$(n/2^2)/2 = n/2^3$
|
|
|
$(n/2^K = 1)$. Only last index is left.

So, $n = 2^K$

Applying log.

$\log n = K \log 2$.

$\boxed{\log_2 n = K}$

Hence, Time complexity for Binary Search is $O(\log_2 n)$

PAGE NO.:
DATE: / /20

• Algorithm of Binary Search:

```
int binarysearch( int arr[], int left, int right, int Key)
{
  if ( right >= left) {

  int mid = left + ((right - left)/2);
  }

  if (arr[mid] == Key)
  { return mid;
  }

  if ( arr[mid] > Key)

  { return binarySearch (arr, left, mid-1, Key); }

else
  { return binarySearch( arr, mid+1, right, Key); }

}
return -1;

}
```

Another approach :

```
int binarysearch(struct list list, int key){
  int l, mid, h;
  l=0;
  h= list.length-1;
  while ( l<=h)
  {
    mid = (l+h)/2;

    if (Key == list.B[mid])
      { return mid;
      }

    else if (key< list.B[mid])
      {
        h=mid-1;
      }

    else {
        l= mid+1;
      }
  return -1;
}
```

Sorting :

1) **Bubble Sorting.**

Time Complexity : $O(n^2)$

	0,	7,	10,	0,	3,	9,	2	c + s
1st Pass	5,	7,	0,	3,	9,	2,	10	6 + 6.
2nd Pass	5,	0,	3,	7,	2,	9,	10	5 + 5
3rd Pass	0,	3,	5,	2,	7,	9,	10	4 + 4
4th Pass	0,	3,	2,	5,	7,	9,	10	3 + 3
5th Pass	0,	2,	3,	5,	7,	9,	10	2 + 2
6th Pass								1 + 1

$\frac{n(n+1)}{2}, n\frac{(n+1)}{2}$

⇒ Time complexity is of the $O(n^2)$

Psuedo Code of Bubble Sorting:

```
for(i=0; i<n ; i++)
{
 for (j=1; j<n-i; j++)
 {
   if (a[j] > a[j+1])
   { temp = a[j];
     a[j] = a[j+1];
     a[j+1] = temp;
   }
 }
}
```

Selection Sort

* Comparisions can be multiple, but there exists only a single swap in each pass.

* Time Complexity : $O(n^2)$.

	9	2	10	0	5	3
"Pass 1"	0	2	10	9	5	3
----	0	2	10	9	5	3.

Psuedo Code for Selection Sort :-

```
for( i=0; i<n ; i++)
{ pos = i;
  for ( j= i+1; j<n ; j++)
{
  if ( a[j] < a[pos])
   {
     pos = j;
   }
}
 if (pos != i)
{ temp = a[i];
  a[i]= a[pos];
  a[pos] = temp;
 }
}
```

Insertion Sort

* Time Complexity : $O(n^2)$.

	9	2	10	0	5	3
Pass 1	2	9	10	0	5	3
Pass 2	0	2	9	10	5	3
Pass 3	0	2	3	9	10	5
Pass 4	0	2	3	5	9	10

Psuedo code for Insertion Sort:

```
for (i=1; i<n; i++)
{ Key = a[i];
   j = i-1;

while( j>=0 && a[j] > key)
{
  a[j+1] = a[j];
  j=j-1;
 }

}
```

12.1.24

Hashing: • Worst case [Time complexity : $O(1)$]

- DAT : Direct Data Translation.
- Hash function : $29999 \% 10 = 9$

$$\left\{\begin{array}{l} 1 \% 10 = 1 \\ 11 \% 10 = 1 \end{array}\right\} \rightarrow \text{Hash Values.}$$

Collision

Linear Probability — Quantitative Probability

0	
1	1
2	2
3	3
4	

1, 2, 3, 4, 5, 10, - - - 2999

	0
1	1
2	2
3	3
	4
	5
	|
	|
	|
	|
	|
	|
	2999

(Use) Can easily access data only by indices.

(Drawback) Large Memory required for large data.

Hash Table

Advantages :-

→ Insertion is easy, $I = O(1)$.

→ Infinity collision resolve.

Disadvantages :-

→ Searching & Deletion is tough.
$S = O(n)$; $D = O(n)$.

Collision :-

★ Seperate Chain [Open Hashing]

★ Open Addressing [Closed Hashing]

→ Linear Probing

→ Quadratic Probing

→ Double Hashing.