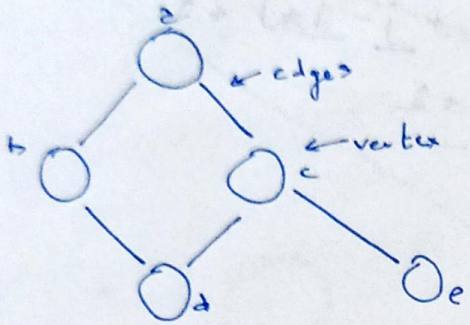
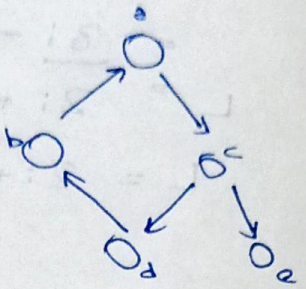
Non-Linear Data Structures

~~Trees~~ Graph



undirected graph



directed graph

→ in undirected graph

$ab = ba$

→ in directed graph

$ab \neq ba$

Connected graph

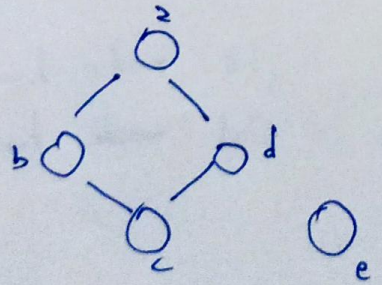
→ any vertex will have at least one path

→ a path between every pair of veritces exist

Disconnected graph

→ There exist a vertex which is not connected to any other vertices

Eg



Mathematical Defination of graph

An undirected graph $G = (V, E)$ consists of a set V (called Vertices) of elements are called vertices and a multiset E of ~~eleman~~ pair of elements of V, called edges

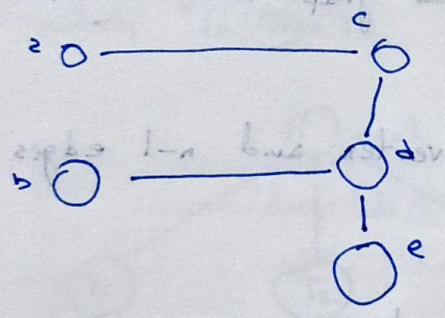
Eg: $V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (b,d), (c,d), (c,e)\}$ ← order doesn't matter as it's undirected

I) Tree

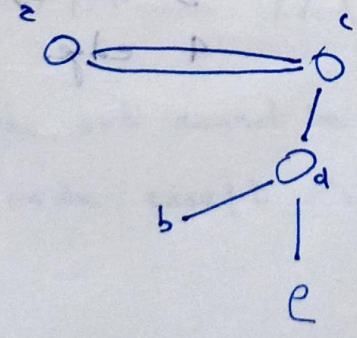
Tree is minimally connected graph.

Eg:



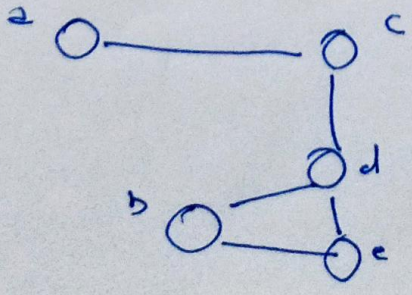
✓ Tree

Eg



X Tree

Eg:



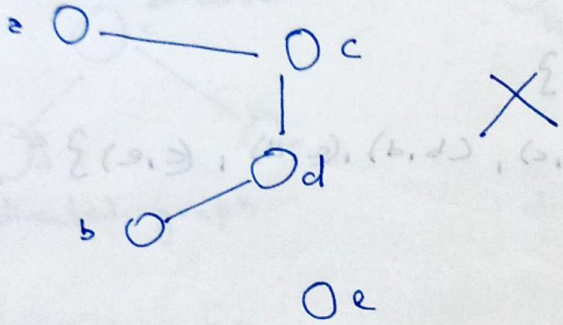
X Tree

→ only one path exist for every vertex

Connected Tree

→ Tree shoud be connected

Eg:



X Tree

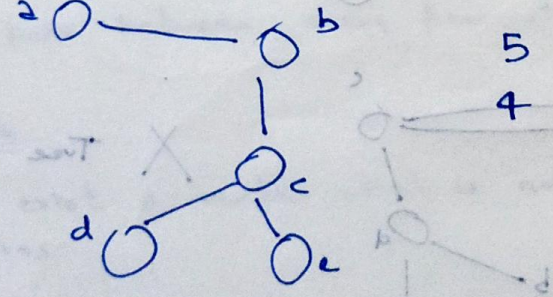
Defination II

Tree is a connected graph with no cycle

Defination III

Tree is A connected graph with n vertex and n-1 edges

Eg:



5 vertices
4 edge.

→ All Trees are graph, but not all graphs are tree.

DS definition of Tree

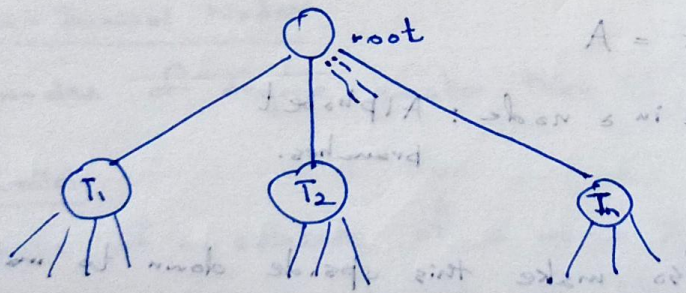
A tree is a finite set of one or more nodes such that

i) there is a specially designated node, called the root

ii) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, \ldots T_n$, where each of these sets are again Tree.

$T_1, T_2, T_3, \ldots T_n$ are called the sub-trees of the Tree

Eg: Directory in Win 10



→ an empty set cannot be a Tree, as it doesn't have "root"

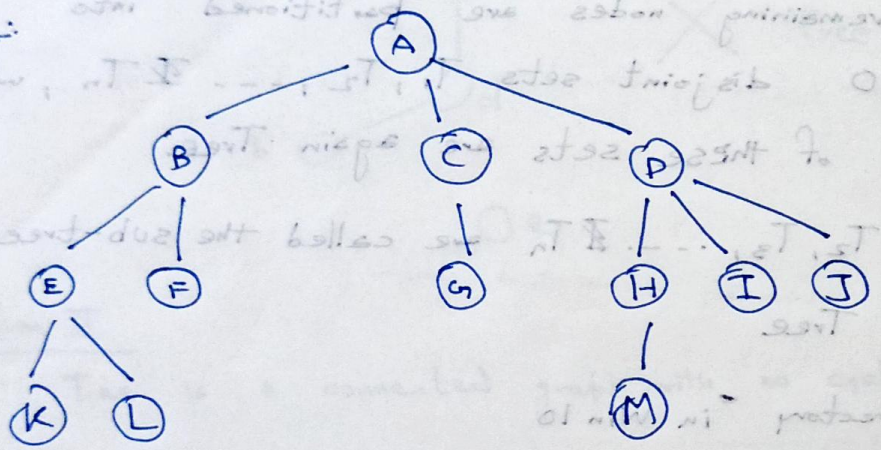
→ every node except "root" will have it's root

node of a tree

~~contain~~

→ A node stands for, the item of information plus branches to the other nodes / items.

Eg:



root = A

data in a node : Alphabet

branches.

→ we can also make this upside down to make it look like actual tree.

Degree of a node

The number of subtrees of a node.

Eg: degree of A = 3
degree of C = 1
degree of G = 0

→ not consider it's upward branches

degree of D = 3, not 4

Leaf node

→ nodes that have degree 0
→ also called terminal nodes

Eg: M, I, J, G, F, K, L are Leaves

Non-Leaf / Non-Terminal Nodes

→ nodes of degree greater than 0

Parent and Child

The roots of a subtree of a node X are the children of X, and X is called the parent of it's children.

Eg: i) A has no parent, but have three childs; B, C, D
ii) parent of B, C, D is A

Siblings

childrens of same parent

Eg i) B, C, D
ii) H, I, J
iii) E, F
iv) K, L

→ E, F, G are not siblings

Degree of a tree

→ max degree of all nodes

Eg: 3 in eg fig

Ancestors

The Ancestors of a node are all the nodes along a path from root to that node

Eg: Ancestor of A = X
Ancestors of C = A
Ancestors of G = A, C
Ancestors of L = A, B, E

Level of a node

→ assume level of root = 1

→ for any arbitrary node in level L, it's children will be in level L+1

Eg Level of A = 1
Level of H = 3
Level of L = 4

Height and/Depth of a Tree

→ height/depth of a tree is defined to be the max level of any node in the tree.

Eg: 4 in eg. fig.

Forest

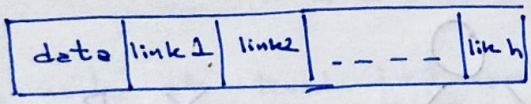
A forest is a set of $n \geq 0$ disjoint trees

Nested Notation of Tree (list representation)

→ Nested Notation of prev Tree;

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

Linked Representation of Tree



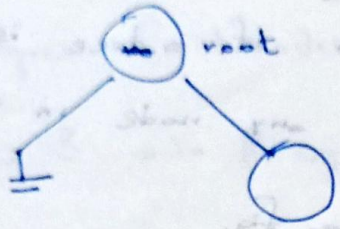
1. Binary Trees

→ every node will have at most two links.

→ A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint (binary) trees called the left subtree and right subtree.

→ recursive tree, with base case ~~leaves is~~ of the "childrens of the leaves" [NULL]

Eg

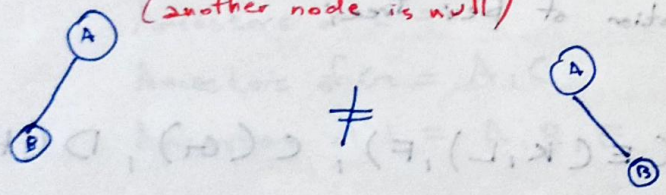


✓ Binary tree

→ you have to allow the possibility of a tree to be empty

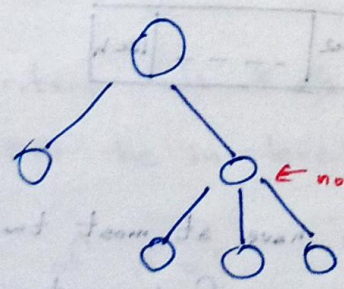
→ here order of subtrees will matter.

Eg:



Importance of "disjoint Binary"

Eg



✗ Binary Tree

← not possibility of Left/Right Position

Lemma 1

The max No. of nodes on level i of a binary tree is 2^{i-1}

$i \geq 1$. ($i-1$ is root)

Proof: Proof by Induction:

Base Case: The root is the only node at level 1.

Hence the max. of node in level 1 is

$$2^{1-1} = 2^0 = 1$$

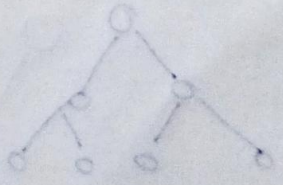
Induction Hypothesis:

Let for all $1 \leq j < i$, the max. no. of nodes in level j in 2^{j-1}.

Induction Step:

By induction hypothesis, the max. no. of nodes at level $i-1$ is $2^{(i-1)-1} = 2^{i-2}$

→ every node at level $i-1$, will have at most 2 children nodes per node., the max. no. of nodes at level i = 2 * (max.no. of node in level $i-1$) = $2^{i-2} * 2 = 2^{i-1}$



Lemma 2

The maximum no. of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$ (root in level 1)

Proof: by lemma 1, the max. no. of nodes in on level i is 2^{i-1}

So the total no. of nodes in binary tree of depth k

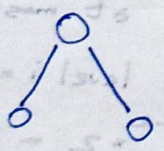
$$N = 2^{1-1} + 2^{2-1} + \cdots + 2^{k-1}$$ (sum of max. no. of nodes in a level i)

$$= 1 + 2 + 4 + \cdots + 2^{k-1}$$

$$= \sum_{i=1}^{k} 2^{i-1}$$

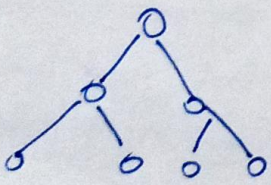
$$\left[\frac{a(1-r^{n+1})}{1-r}\right] = \frac{1(1-2^{k-1+1})}{1-2} = 2^k - 1$$

Eg 1



max. Total $= 2^2 - 1 = 3$

Eg 2



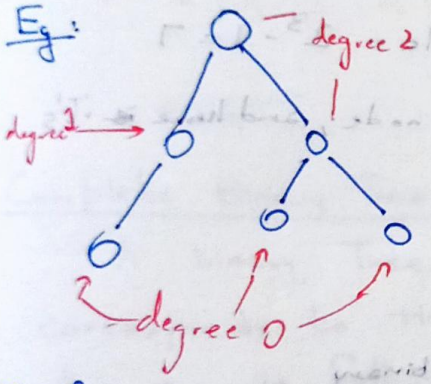
max. Total $= 2^3 - 1 = 7$

Lemma 3

For any non-empty binary Tree T, if n_0 is the ~~no. of~~ nodes terminal nodes and n_2 is the no. of nodes with degree 2, then

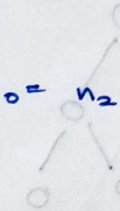
$$n_0 = n_2 + 1$$

Eg:

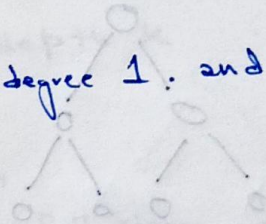


$n_0 = 3$
$n_1 = 1$
$n_2 = 2$

, $n_0 = n_2 + 1$.



Proof: let m be the no. of nodes with degree 1. and n be the total no. of nodes.

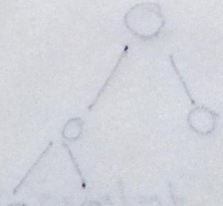


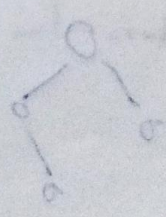
$$n = n_0 + n_1 + n_2$$

B = no. of branches in T = $n_1 \times 1 + n_2 \times 2 = n - 1$

Simplifying $n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$

or $n_2 + 1 = n_0$



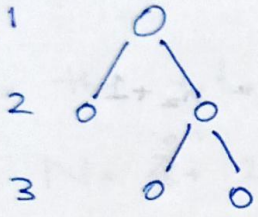


Representation of Binary Trees

Definition

Full Binary Tree: A full binary tree of depth k, is a binary tree of depth k having $2^k - 1$ nodes.

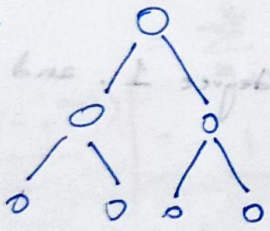
Eg:



Max node possible = $2^3 - 1 = 7$

but this has 5 node, and hence it's not full

Eg:

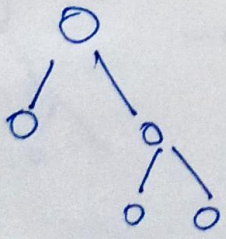


This is full binary

Strictly Binary Tree

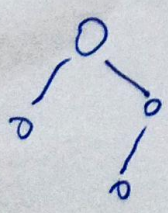
→ each node can have either 0 child or 2 child, but not 1 child

Eg:



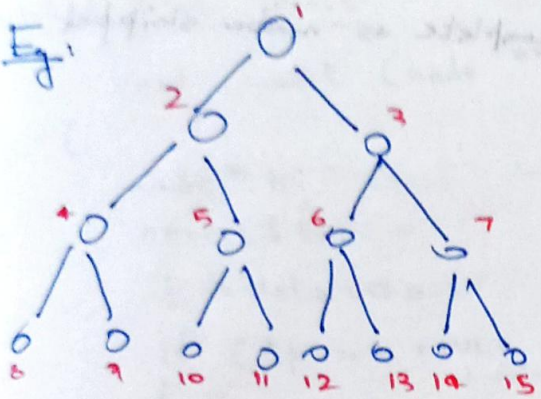
It is a strictly Binary tree.

Eg:



It is not stritctly binary

Eg:



→ we can represent this tree, by sequential numbering of nodes.

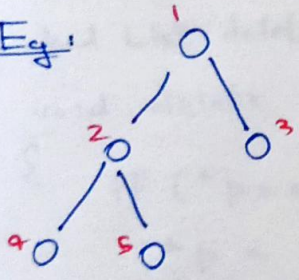
→ so we can use array representation.

~~array I~~

Complete Binary Tree

A binary Tree of depth k is complete iff it's nodes coressponds to the nodes which are numbered from 1 to n, in full binary ~~be~~ tree of depth k

Eg:

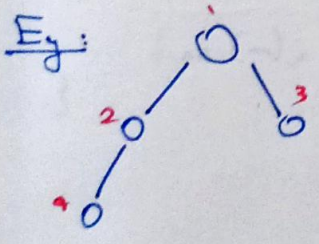


→ This not full Binary

→ Depthe 3, nodes for full binary required = 7 but has 5

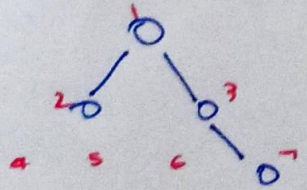
→ doing number as before, as numbering has all the numbers, from 1 to 5, it is complete

Eg:



→ numbering Ok, hence it's full

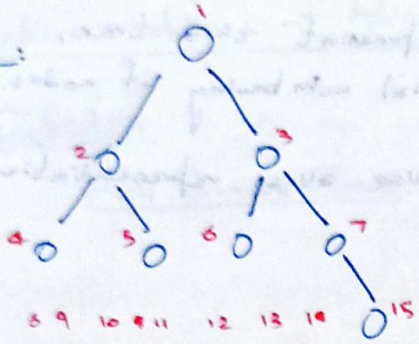
Eg:



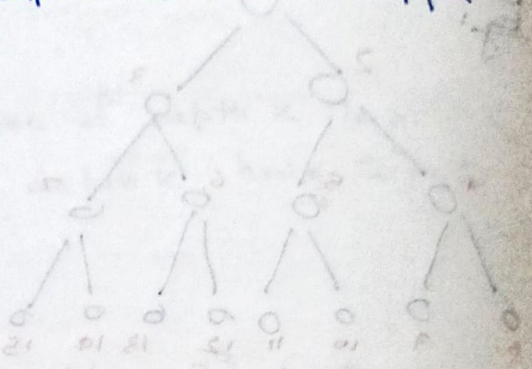
→ here 4,5,6 not existent, ~~here~~ but 7 is present

→ number skipped, hence not complete.

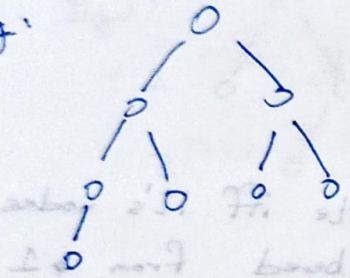
Eg:



→ Not complete as number skipped

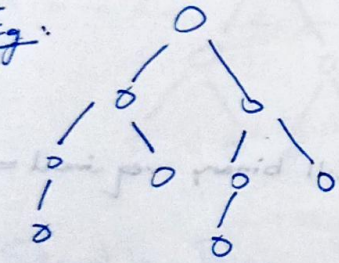


Eg:

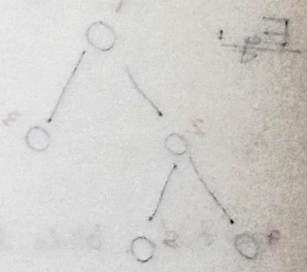


complete

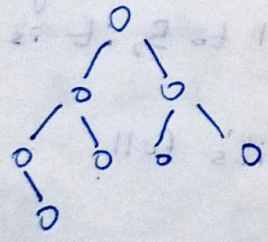
Eg:



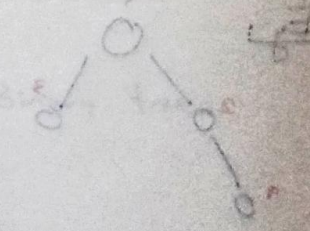
Not complete

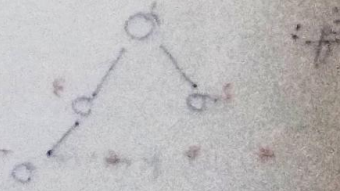


Eg



Not complete





Linked List Insert

```
void insert (node **p, node *y)
{
    node *t;
    new (&t);
    t->data = 40
    if (*p == NULL
    {
        *p = t;
        t->next = NULL;
    }
    else
    { t->next = y->next;
      y->next = t;
    }
}
```

Linked List delete.

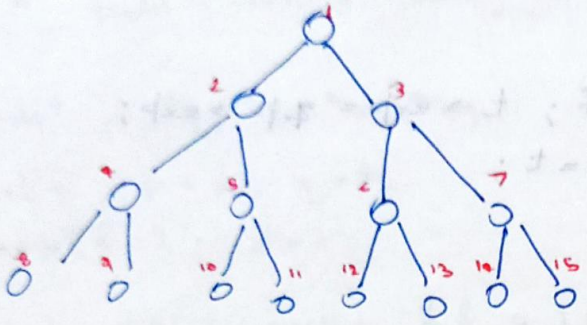
```
void delete (node **p, node *m, node *y)
{
    if (*p == m)
        *p = (*p)->next;
    else
        y->next = m->next;
    free (m);
}
```

Polynomial Addition

```
node * padd (node *a, node *b)
{
   node *p, *q, *c, *d, *t;
   int m; p = a; q = b;
   new(&c); d = c;
   while (p != NULL && q != NULL
   {
      if (p->expo == q->expo)
      {
         m = p->coeff + q->coeff;
         if(m != 0)
         {
            new (&t);
            t->coeff = m; t->expo = p->expo;
            d->next = t ; d = t;
         }
         p = p->next; q = q->next
      }
      else if (p->expo < q->expo)
      {
         new(&t) ;
         t->coeff = q->coeff ; t->expo = q->expo;
         d->next = t ; d = t;
         q = q->next;
      }
      else if (p->expo > q->expo)
      {
         new (&t);
         t->coeff = p->coeff ; t->expo = p->expo;
         d->next = t ; d = t;
         p = p->next;
      }
}
```

```
while (p != NULL)
{
    new (&t);
    t->coeff = p->coeff ; t->exp = p->exp;
    d->next = t ; d = t;
    p = p -> next;
}
while (q != NULL)
{
    new(&t);
    t->coeff = q-> coeff; t->exp = q->exp;
    d->next = t ; d=t;
    q = q->next;
}
d->next = NULL;
t = c; c = c->next ; free(t);
return c;
}
```

Complete Binary Tree



Theorem

If a complete binary tree with n ~~do~~ nodes, $[\text{depth} = \lfloor \log_2 n \rfloor + 1]$ is represented sequentially, then for any node with index ~~i, 1≤i≤n~~ $1 \le i \le n$, we have

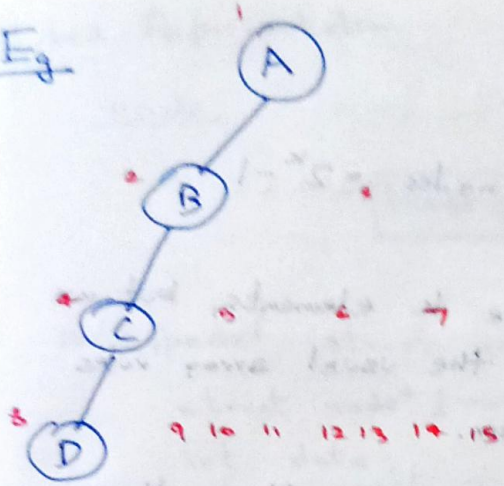
I) parent (i) is $\lfloor i/2 \rfloor$, $i \neq 1$; when $i = 1$, i is the root of the tree

II) left-child $(i) = 2i$, $2i < n$; if $2i > n$, then i doesn't have a left-child

III) right-child $(i) = 2i + 1$; ~~2i<n~~ $2i+1 < n$; if $2i+1 > n$, right child doesn't exist

→ using these rules, we can represent complete binary tree using array. ; using array ~~makes~~ ~~less~~ uses less space. but can only be used in tree, whose structure is always known.

Eg



arr:

1	2	3	...	9	...
A	B	C	.....	D	...

(Space wasted a lot)
(using links better)

→ go for array, if tree is complete ; else go with links

Proofs

→ (III) is a consequence of (II)
→ we can prove (I), using (II) and (III)

∴ Proving (II)

→ using induction.

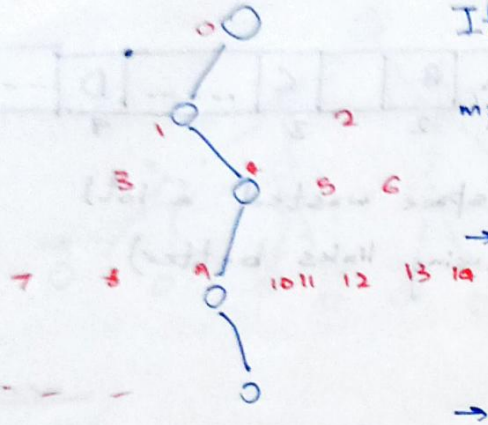
Base case : for i=1, left child of root → index = 2
if 2 > n, root doesn't have left child.

Induction hypothesis :
assume for all j, $1 \le j \le i$; left-child(j) = 2j

Induction Step :
The two nodes immediately preceeding l-child (i+1) are left and right child of i :

left-child (i) = 2i (assume true as hypothesis)
right-child (i) = 2i+1 (by definition of complete B Tree)
∴ left-child (i+1) = 2i+2 = 2(i+1)

Skewed Binary Tree.



If depth = k

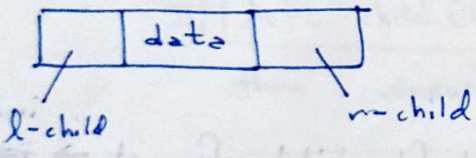
max. no. of nodes = $2^k - 1$

→ we can use k elements, but we can't use the usual array rules and Theorem.

→ so for making sure, those theorem remains valid, we have to make ~~2^k~~ $(2^k - 1)$ size array

→ but $2^k - 1 - k$ places will be wasted. (very inefficient)

→ we can ~~make it size array~~ ; or use links.

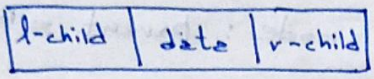


Disadvantage of ~~Linked~~ Sequential Representation

→ Deleting a node means deleting all the child / grand child, etc. of the node, (lot of data movement)

Linked Representation

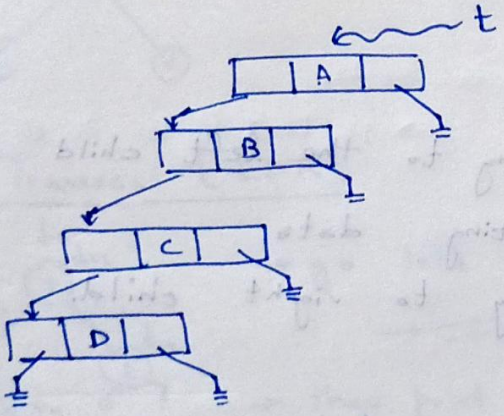
node



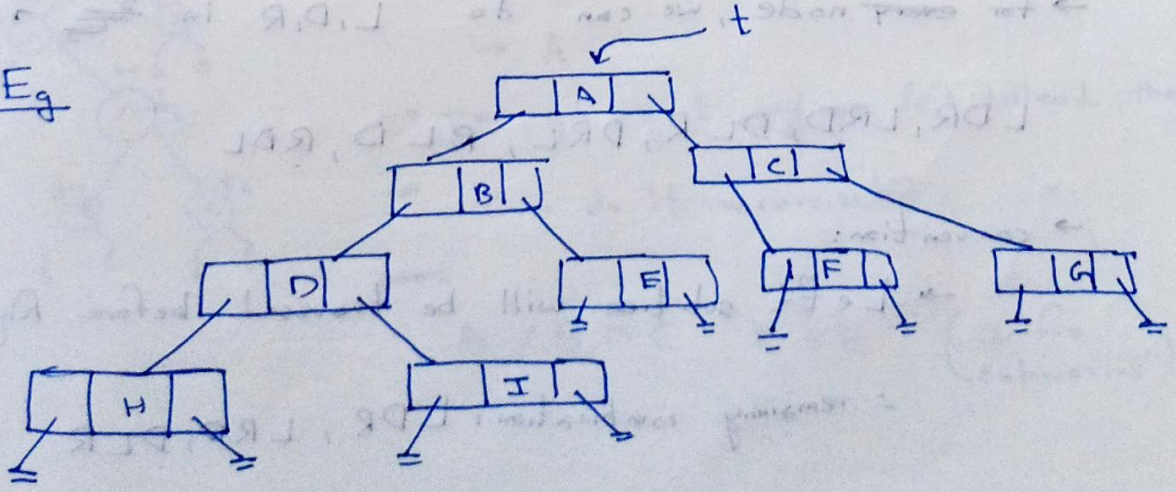
→
```
typedef struct node {
    struct node* l-child;
    int data;
    struct node* r-child;
} ~~node;~~ tree_node;
```

→ `tree_node *t;` → pointer to a tree_node structure.

Eg:



Eg



(complete Binary: 4 levels)

→ we can add one more field in node so as to make it possible for a node to go to it's parent

→ For root node : parent = NULL

→ We go for making balanced tree, ~~as it~~ as it has lesser height, it means

→ less comparisions while traversing or searching

Operations on Trees

→ Traversal: visiting every nodes; one at a time.

Traversal

→ Notation

→ L : moving to the left child

→ D : Printing data

→ R : moving to right child.

→ for ~~every~~ node, we can do L, D, R in ~~3!~~ → 6 ways:

LDR, LRD, DLR, DRL, RLD, RDL

→ convention:

→ Left subtree will be traversed before Right subtree

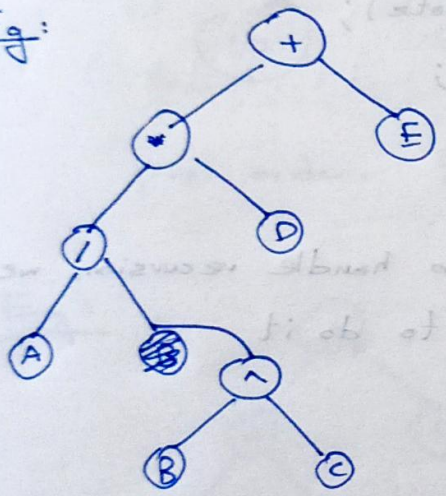
∴ remaining combination: LDR, LRD, DLR

→ depending on the position of D, we see

- → LDR : in-order
- → LRD : post-order
- → DLR : pre-order

[similar to prefix, infix, postfix]

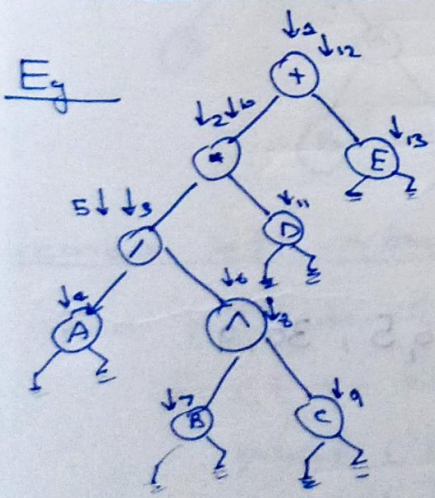
Eg:



→ This is called syntax tree
(made during compiling)
(it's making is discussed in Compiler design)

In-order Traversal [LDR]

Eg



→ go left till you can

→ then print
→ A

→ then go right, and go left till end, then print
(we do it recursively)

ans:

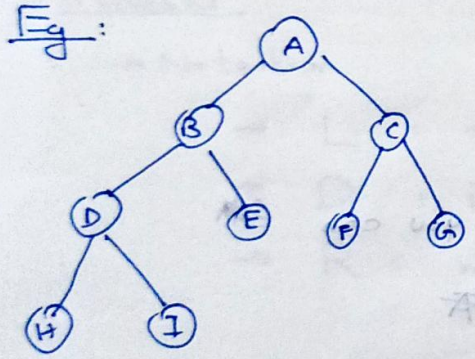
A / B ^ C * D + E (in-fix expression)

Recursive In-order Traversal

```
void in_order (tree_node* t) {
    if (t != NULL) {
        in_order (&t->l_child);
        printf ("%d ", t->data);
        in_order (t->r_child);
    }
}
```

→ as compiler uses stack to handle recursion, we can also implement stack to do it

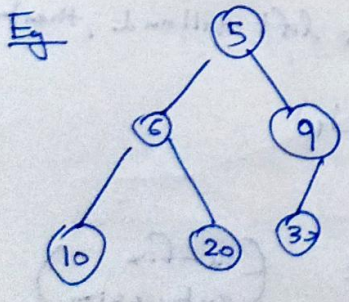
Eg:



in-order

→ H D I B E A F C G

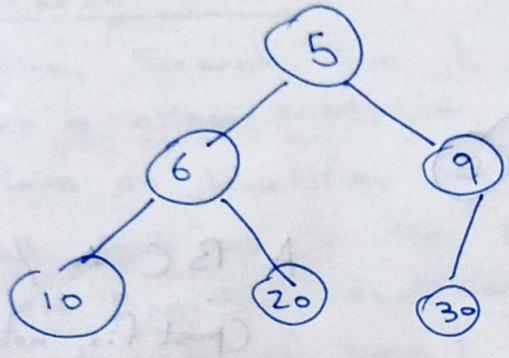
Eg



in-order → 10, 6, 20, 5, 30, 9

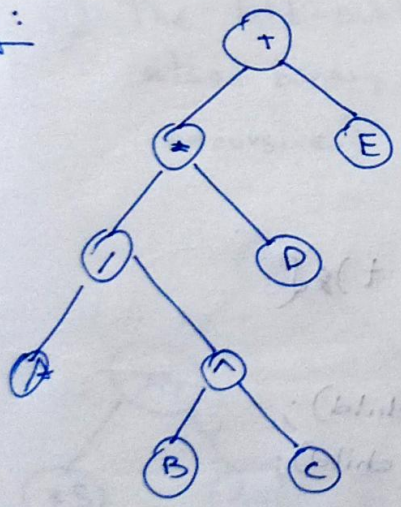
Pre-Order Traversal [~~RR~~ DLR]

Eg:



pre-order: 5, 6, 10, 20, 9, 30

Eg:



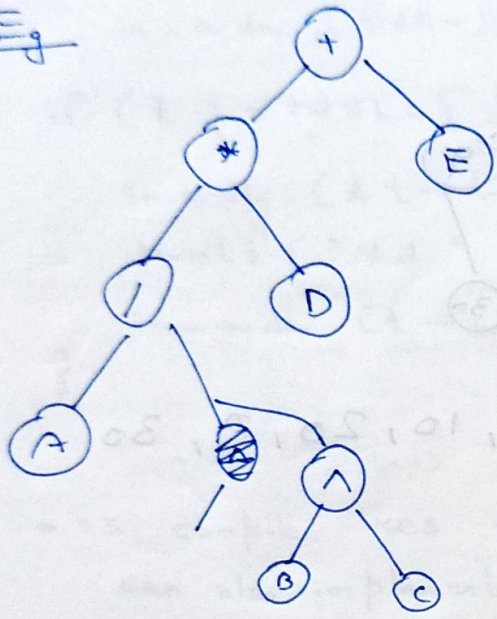
pre-order
→ + * / A ^ B C D E
(pre-fix operation)

recursive in pre-order Traversal

```
void pre-order (tree-node* t) {
    if (t != NULL) {
        printf ("%d ", t→data);
        pre-order (t→l_child);
        pre-order (t→r_child);
    }
}
```

Post-order Traversal [LRD]

Eg



A B C ^ / D * E +

(post fix notation)

recursive post order traversal

```
void post_order (tree_node* t){
    if (t != NULL){
        post_order (t->l_child);
        post_order (t->r_child);
        printf ("%d", t->data);
    }
}
```