Lecture 25–27

- Introduction to Binary Trees

IT205: Data Structures (AY 2023/24 Sem II Sec B) — Dr. Arpit Rana

Motivation

So far we have learned about arrays, stacks, queues and linked lists, which are known as linear data structures.

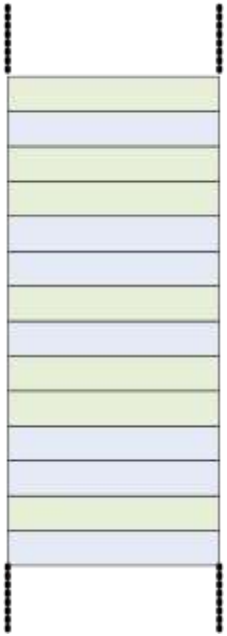
- linear because the elements are arranged in a linear fashion (that is, one-dimensional representation)
  .

Another very useful data structure is tree, where elements appear in a non-linear which fashion, require two-dimensional representation.

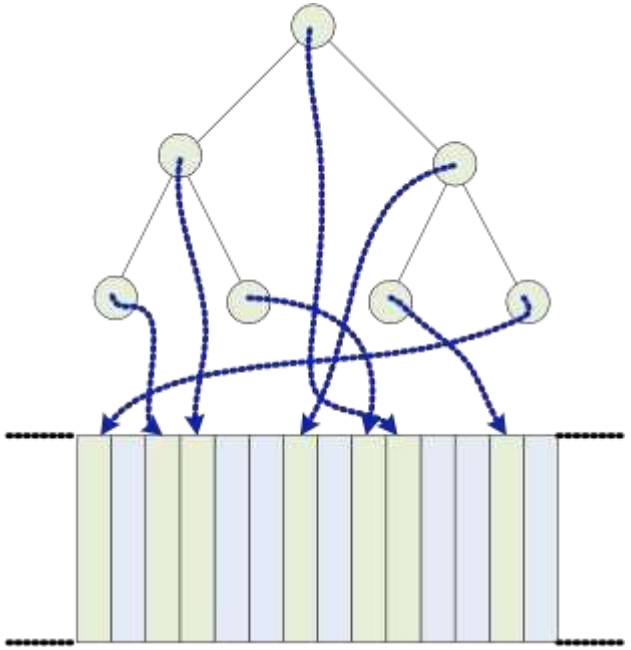
- Maintains information in a hierarchical manner.
- Basic operations such as insertion, deletion, searching, etc., are more efficient in trees than linear data structures. -> decrease time complexity.

Motivation

Contagious
Data Storage



Non-Contagious
Data Storage



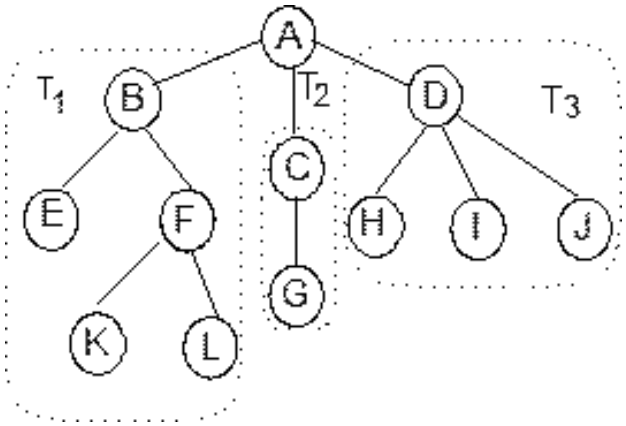
Efficient Retrieval of Data from
Non-Contagious Data Storage

Definition

A Tree is a finite set of elements that is either empty or is partitioned into n (n > 0) disjoint subsets:

- the first subset contains a single element called the *root* of the tree;
- the other subsets are themselves trees, called *subtrees* of the original tree.
    - A subtree can be empty

Each element of a tree is called a *node* of the tree.

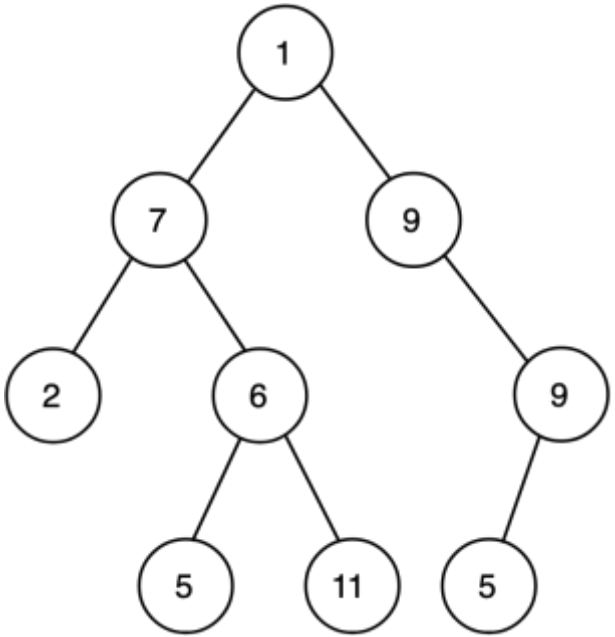


Definition

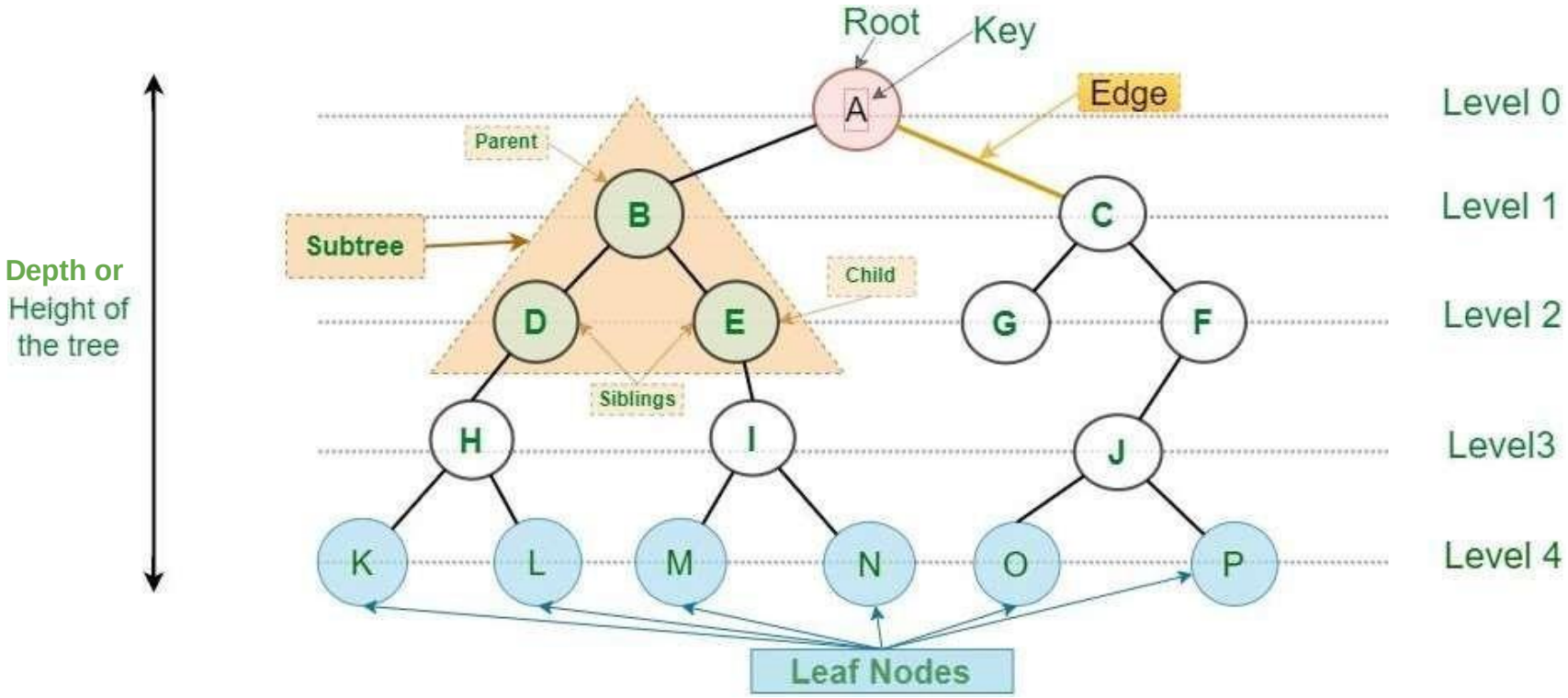
A Binary Tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets:

- the first subset contains a single element called the *root* of the tree;
- the other two subsets are themselves binary trees, called the *left and right subtrees* of the original tree.
    - A left or right subtree can be empty

Each element of a binary tree is called a *node* of the tree.



Basic Terminologies



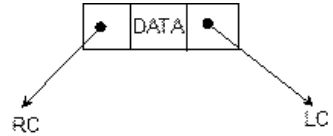
Basic Terminologies

- Node: The main component of a tree structure. It stores the actual data and links to the other nodes.
- Parent/Father: The immediate predecessor of a node.
- Child/Son: The immediate successor of a node. Specifically, a child (son) may be left child (son) or right child (son).
- Link/Edge: This is a pointer to a node in a tree (represented as an edge).
- Leaf/External nodes: The node which has no child (son).
- Level: This is the rank in the hierarchy. The root node has level 0, while, level of any other node is one plus the level of its father (parent).
- Height/Depth: The maximum level of any leaf node is the depth of the tree.

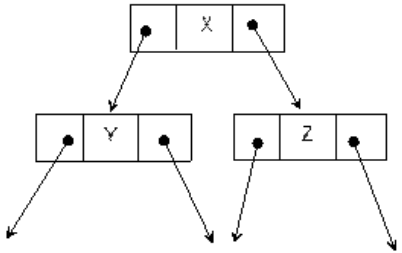
Basic Terminologies

- Degree/Arity: The maximum number of children/sons that is possible for a node is known as the degree of a node.
- Sibling: The nodes at the same level having the same parent/father are known as siblings.

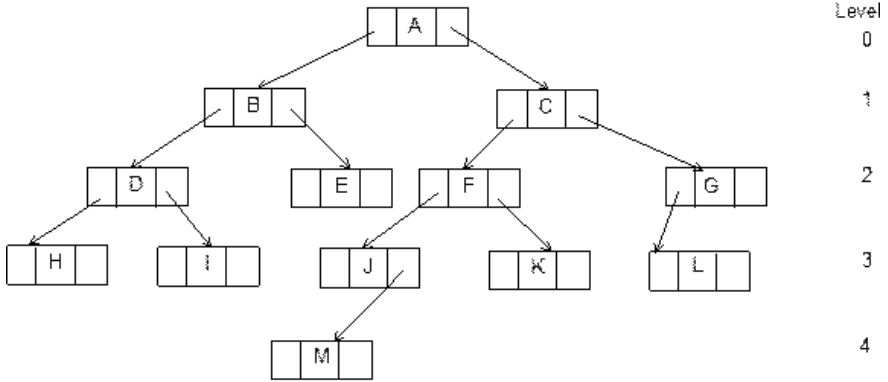
If tree have N node then it have n-1 edges or links.



(a) Structure of a node in a tree



(b) Parent, left child and right child of a node



(c) A simple tree with 13 nodes

BINARY TREE

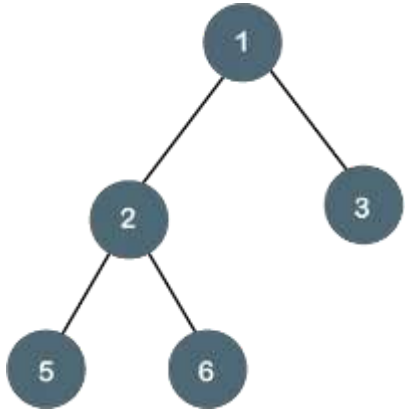
- max degree=2
- if unlabeled tree have n node then total number of possible tree is 2nCn/n+1
- if unlabeled tree have n node then it have 2^n-1 tree which have max height and max height will be n-1
- n labeled tree each shape can be draw in n! ways . So if number of node is n then possible labeled tree 2nCn/n+1 * n!.
- If height is given in binary tree then
- ->min no of nodes H+1(H=height of tree)(skewed binary tree) take root height =0
- ->max no of nodes 2^H+1 – 1(full binary tree)

for strict binary tree

- Max nodes = 2^h+1 -1
- Min nodes = 2h+1
- Min height = log(n+1) -1 {base 2}
- Max height = (n-1)/2
- External nodes = internal nodes + 1 =>> e=n+1

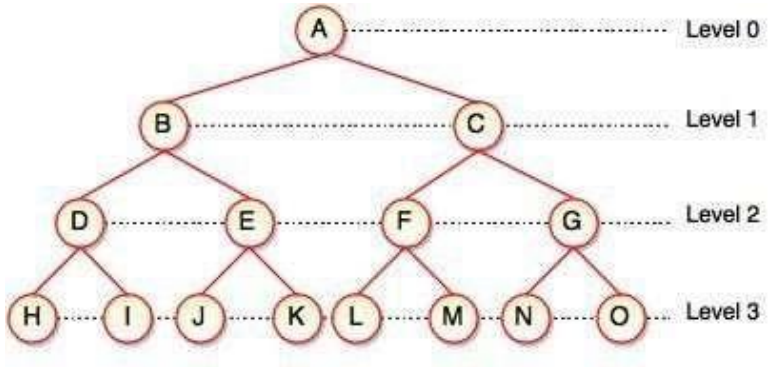
Basic Terminologies

- Strictly Binary Tree: If every non-leaf node in a binary tree has non-empty left and right subtrees, the tree is termed as StrictlyBinary Tree. degree of node must be zero or two-> one child not possible



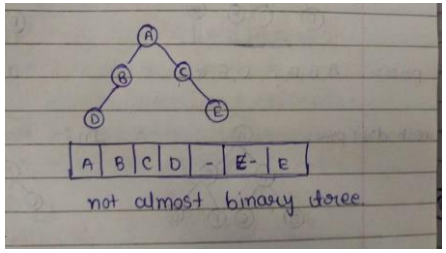
- Complete (Full) Binary Tree: A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d. degree of every non- leaf node must be 2.

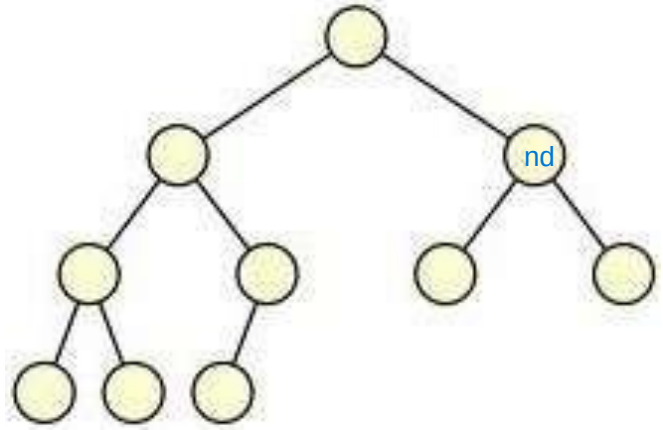
  - Total number of nodes will be m = 2^{d+1}-1, such that 2^d leaves and 2^d-1 non-leafnodes. take height of root node 0
  - The depth will be d = $\log_2(m+1) - 1$.



Basic Terminologies

- Almost Complete Binary Tree: A complete binary tree of depth d is an almost complete binary tree if:
  - Any node at level less than d - 1 has two sons. means till d-1 level it is full binary tree.
  - For any node *nd* in the tree with a right descendant at level d, *nd* must have a left son and every left descendant of the *nd* is either a leaf at level d or has two sons.



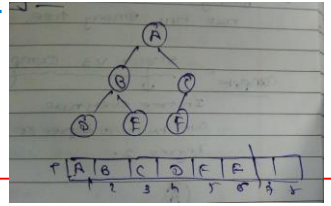


->in almost complete binary tree of height h , will be full binary tree up to h-1 height and at last level or at height h element should fill up from left to right without skipping any element .

->so a full binary tree is always almost complete binary tree but almost complete binary tree may or may not full binary tree.

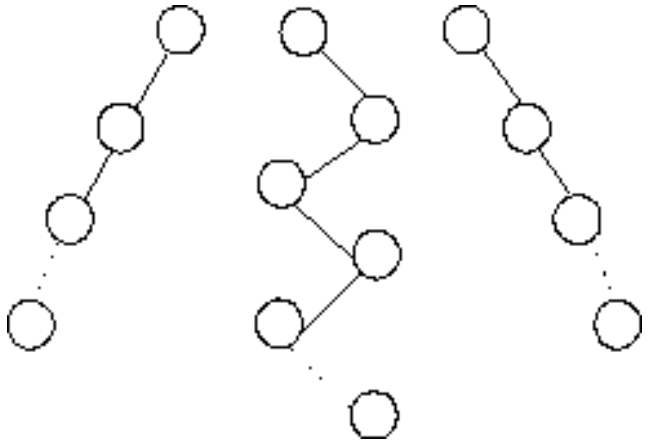
->in array representation ,if in array no blank spaces in between, then it is almost binary tree.

almost binary tree - ->>



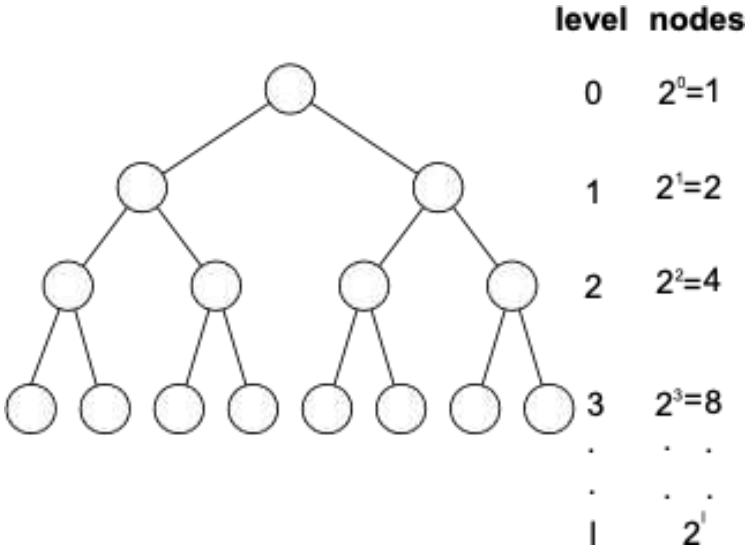
Basic Terminologies

- Skewed Binary Tree: A binary tree in which all the nodes have only either one child or no child.
    - Skewed binary tree of depth d has only one path which contains d number of nodes.



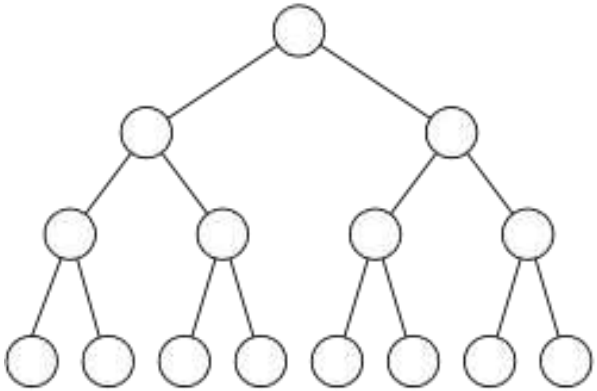
Properties of Binary Tree

- In any binary tree, maximum number of nodes on level *l* is 2^l, where *l* ≥ 0.



Properties of Binary Tree

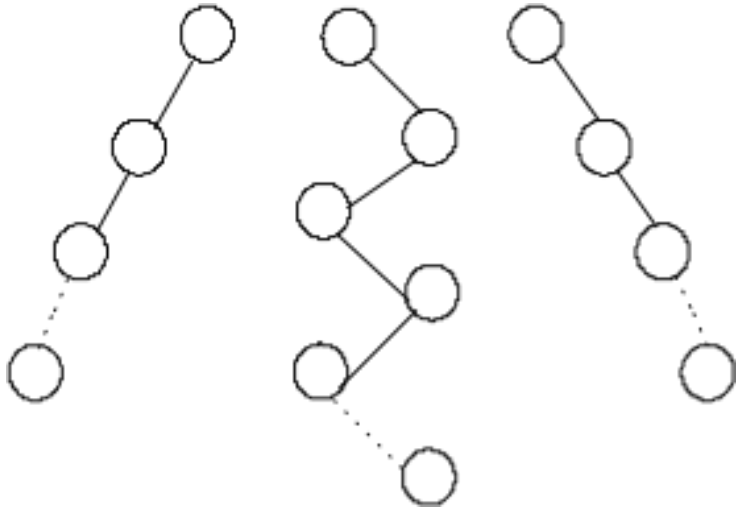
- Maximum number of nodes possible in a binary tree of height (depth) h is $2^{h+1} - 1$, for h >= 0.



level	nodes
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
.	. .
.	. .
l	2^l

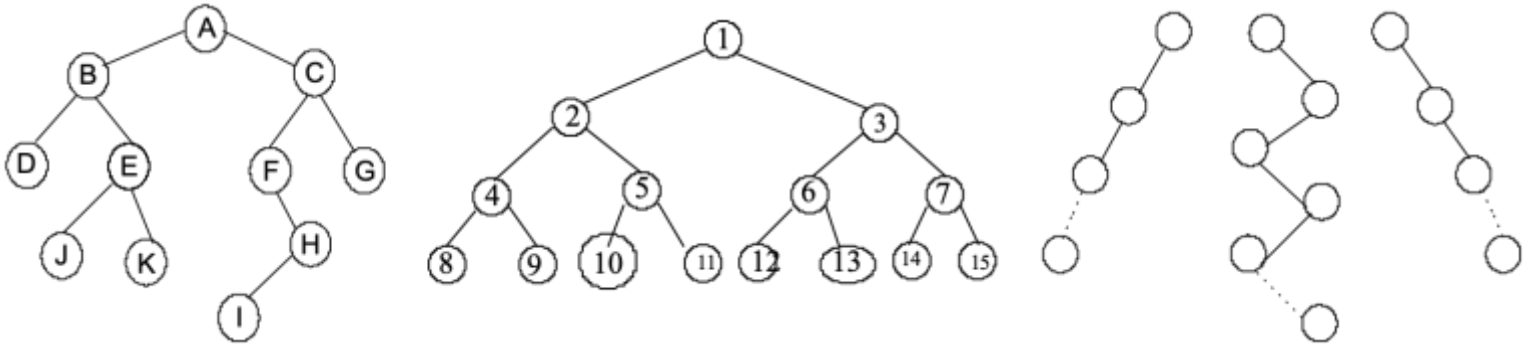
Properties of Binary Tree

- Minimum number of nodes possible in a binary tree of height (depth) h is h.



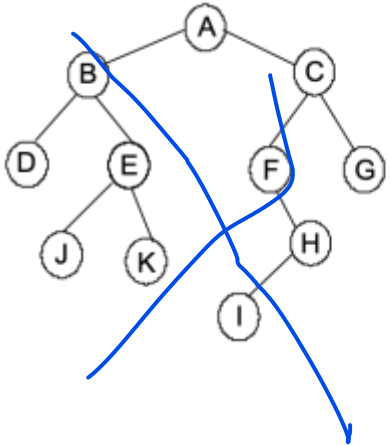
Properties of Binary Tree

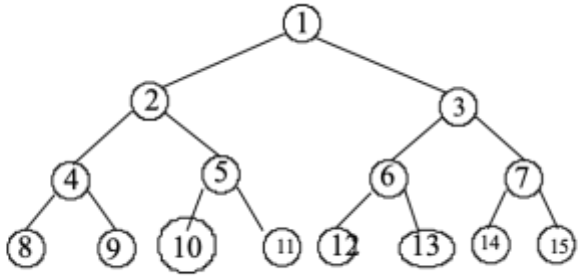
- For any non-empty binary tree, if n is the number of nodes and e is the number of edges, then $n = e + 1$.

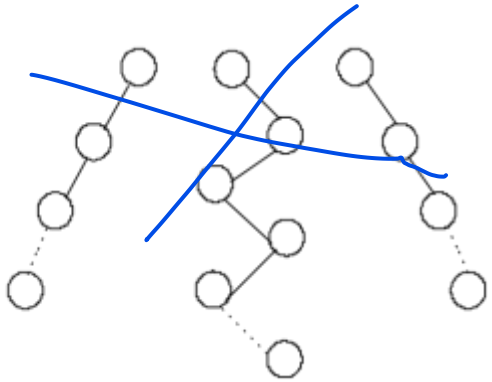


Properties of Binary Tree

- For any non-empty binary tree T, if n_0 is the number of leaf nodes (or external nodes, degree = 0) and n_2 is the number of internal node (degree = 2), then $n_0 = n_2 + 1$.







Representation of Binary Trees

- Linear Representation using Arrays
- LInked Representation using Linked Lists

Representation of Binary Trees using Arrays

For any node with index i, $0 \le i \le n-1$, (for some n):

- PARENT(i) = $\lfloor i/2 \rfloor$ index start with 1

  [(i-1)/2] for index start with 0

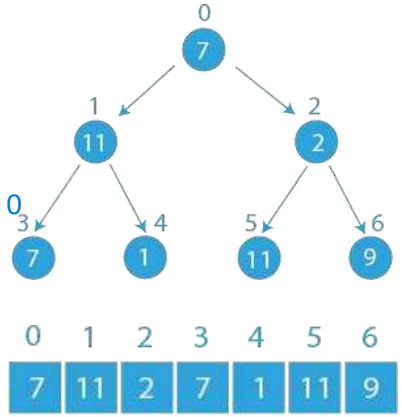
  For the node when i = 0, there is no parent.

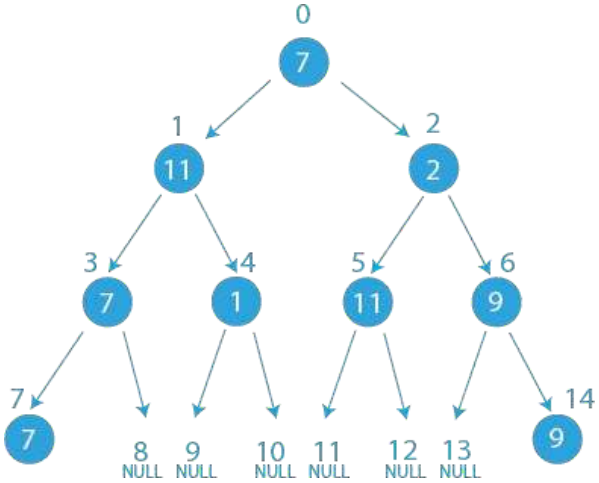
- LCHILD(i) = 2 * i +1 (2*i, when index starts at

  1) If 2 * i + 1 > n-1, then i has no left child.

- RCHILD(i) = 2 * i + 2 (2*i+1, when index starts at

  1) If 2 * i + 2 > n-1, then i has no right child.

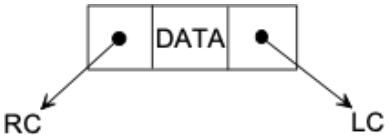


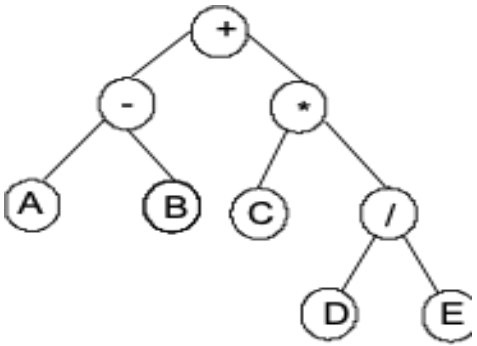


What will be the maximum and minimum size of the array to store a binary tree with *n* number of nodes?

min = n
max = 2^h+1 - 1

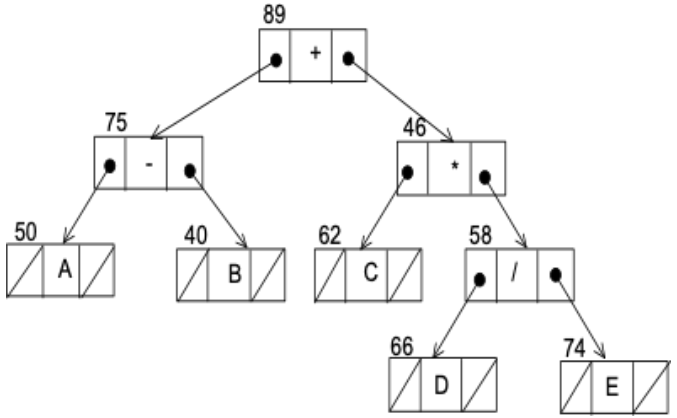
Representation of Binary Trees using Linked Lists





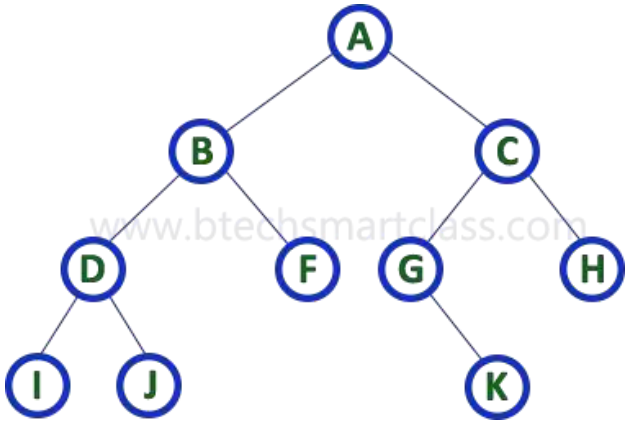
Address	RCHILD	DATA	LCHILD
50		A	
75	50	-	40
40		B	
89	75	+	46
62		C	
46	62	*	58
66		D	
58	66	/	74
74		E	

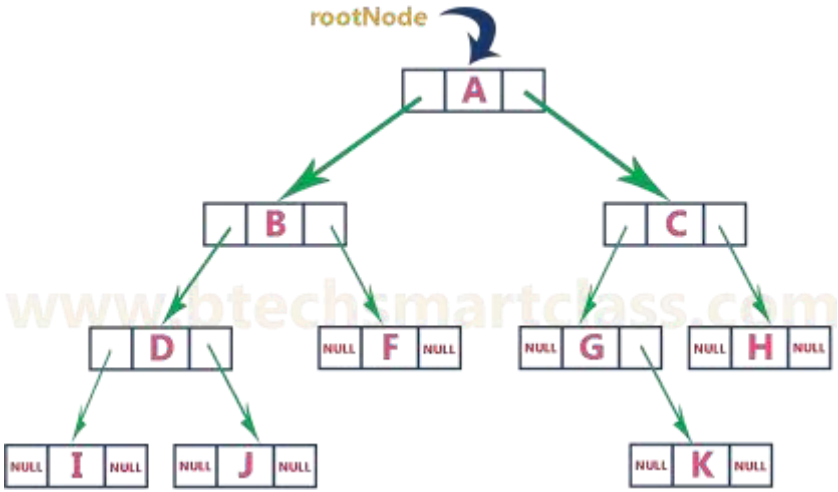
Physical View



Logical View

Array vs. Linked Representation of Binary Trees





A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

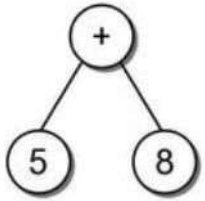
so because of memory wastage in array , linked representation is more useful.

Operations on Binary Trees

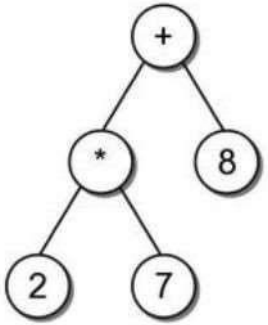
- Traversal: To visit all the nodes in a binary tree.
- Insertion: To include a node into an existing (may be empty) binary tree.
- Deletion: To delete a node from a non-empty binary tree.
- Merging: To merge two binary trees into a larger one.

Traversing a Binary Tree

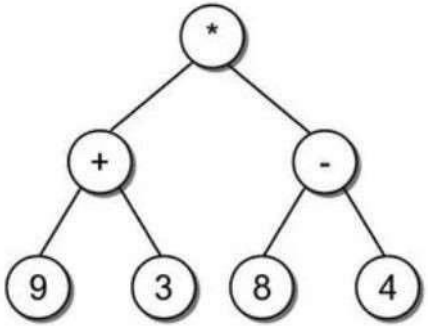
- Traversal operation is a frequently used operation on a binary tree.
- This operation is used to visit each node in the tree exactly once.
- A full traversal on a binary tree gives a linear ordering of the data in the tree.
    - For example, if the binary tree contains an arithmetic expression, then its traversal may give us the expression in infix notation, postfix notation or prefix notation.



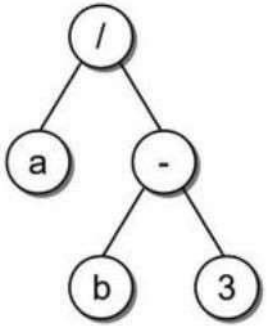
5 + 8



2 * 7 + 8



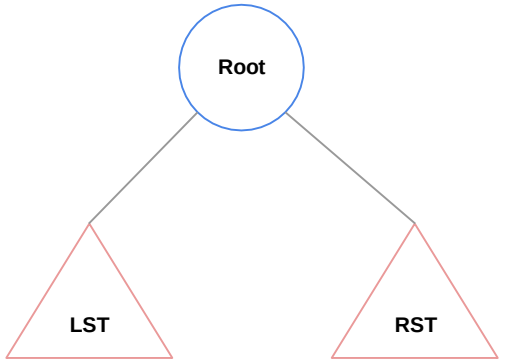
(9 + 3) * (8 - 4)



a / (b - 3)

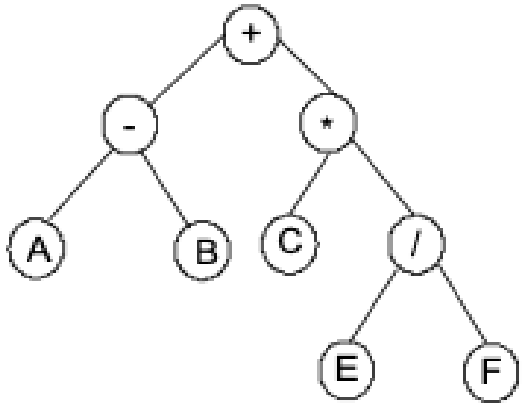
Traversing a Binary Tree

- For a systematic traversal, it is better to visit each node (starting from the root) and its sub trees in the same fashion.
- A tree can be traversed in six possible ways:
  - Root → LST → RST
  - LST → Root → RST
  - LST → RST → Root
  - RST → LST → Root
  - RST → Root → LST
  - Root → RST → LST



Traversing a Binary Tree

- Root → LST → RST: +-A B *C /E F
- LST → Root → RST:
- LST → RST → Root:
- RST → LST → Root:
- RST → Root → LST:
- Root → RST → LST:



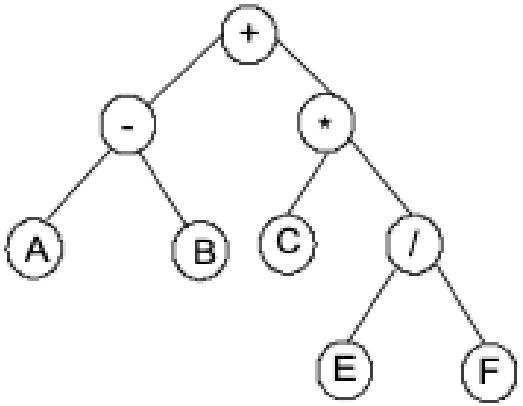
Traversing a Binary Tree

- Root → LST → RST:
- LST → Root → RST:
- LST → RST → Root:
- RST → LST → Root:
- RST → Root → LST:
- Root → RST → LST:

+ - A B * C / E

F A – B + C * E

/ F



Traversing a Binary Tree

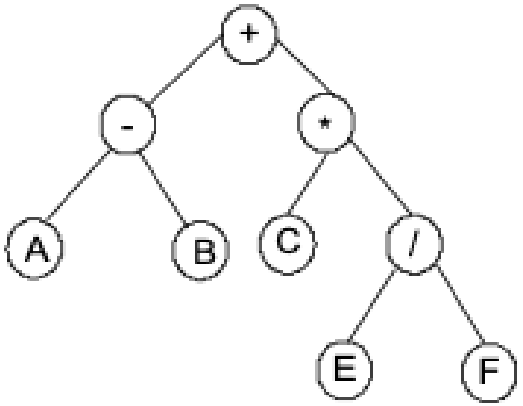
- Root → LST → RST:
- LST → Root → RST:
- LST → RST → Root:
- RST → LST → Root:
- RST → Root → LST:
- Root → RST → LST:

+ - A B * C / E

F A – B + C * E

/ F A B – C E F /

* +



Traversing a Binary Tree

- Root → LST → RST:
- LST → Root → RST:
- LST → RST Root:
- RST → LST Root:
- RST → Root → LST:
- Root → RST → LST:

+ - A B * C / E

F A – B + C * E

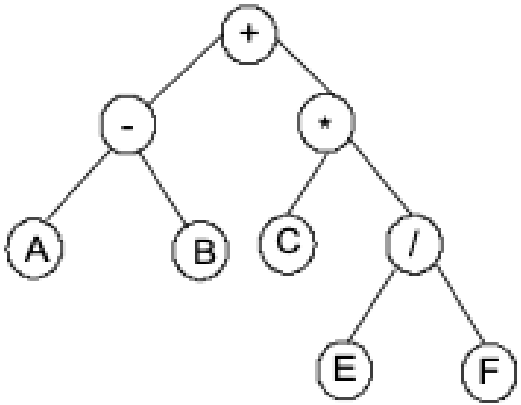
/ F A B – C E F /

* + F E / C * B A

– + F / E * C + B

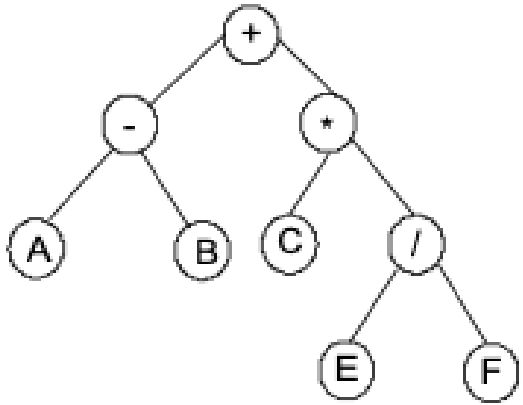
–
A

+ * / F E C – B
A



Traversing a Binary Tree

- Root → LST → RST: + - A B * C / E
- LST → Root → RST: F A – B + C * E
- LST → RST → Root: / F A B – C E F /
- RST → LST → Root: * + F E / C * B A
- RST → Root → LST: – + F / E * C + B
- Root → RST → LST: – A
- RST → LST → Root: + * / F E C – B A



Observation:

The last three expressions are mirror images of first three expressions respectively.

Therefore, out of six possible traversals, only the first three are fundamental.

Traversing a Binary Tree

- Pre-order Traversal: Root → LST → RST
- In-order Traversal: LST → Root → RST
- Post-order Traversal: LST → RST → Root

Pre-order Traversal

```
void pretrav(tree)
{
  If (tree != NULL)
     print(data(tree))
     pretrav(left(tree))
     pretrav(right(tree))
  EndIf
}
```

In-order Traversal

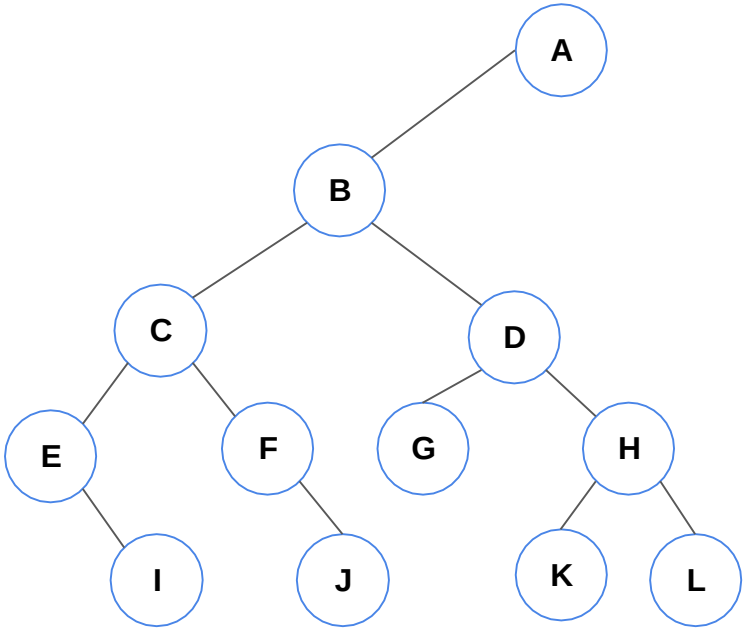
```
void intrav(tree)
{
  If (tree != NULL)
     intrav(left(tree))
     print(data(tree))
     intrav(right(tree))
  EndIf
}
```

Post-order Traversal

```
void posttrav(tree)
{
  If (tree != NULL)
     posttrav(left(tree))
     posttrav(right(tree))
     print(data(tree))
  EndIf
}
```

Traversing a Binary Tree

- Pre-order Traversal: Root → LST → RST
- In-order Traversal: LST → Root → RST
- Post-order Traversal: LST → RST → Root



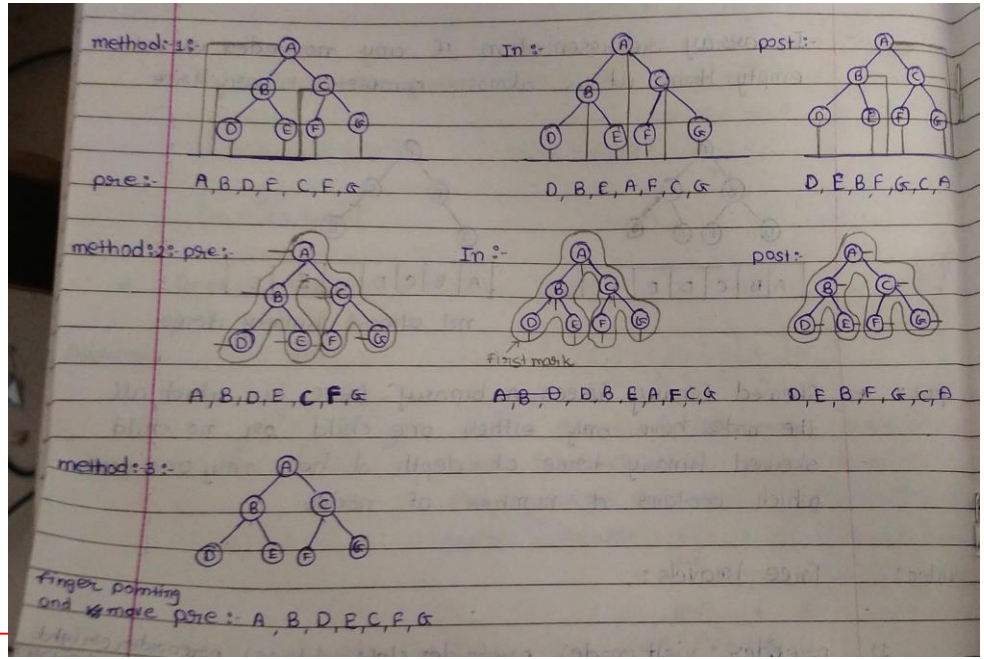
- Pre-order: **ABCEIFJDGHKL**
- In-order: **EICFJBGDKHLA**
- Post-order: **IEJFCGKLHDBA**

Insertion in a Binary Tree

Insertion is a two-step process:

- To search for the existence of a node in the given binary tree after which an insertion has to be made, and
- To establish a link for the new node.

pre ,post ,inorder traversal methods



base case -> O(1)
worst case ->O(n) in skewed binary tree
avg case -> O(logn) balanced binary tree

Deletion in a Binary Tree

Like insertion, deletion is also a two-step process:

- To search for the existence of a node in the given binary tree which has to be deleted
- To adjust the links among parent and child nodes of the deleted node.

avg case O(logn)

Merging Two Binary Trees

This operation is applicable for trees that are represented using linked structure.

There are two ways that this operation can be carried out.

- Suppose, T1 and T2 are two binary trees. T2 can be merged with T1 if all the nodes from T2, one by one, is inserted into the binary tree T1 (insertion may be as internal node when it has to maintain certain property or may be as external nodes).

- Alternatively, when the entire tree T2 (or T1) is included as a subtree of T1 (or T2). For this, obviously we need that in either (or both) tree there must be at least one NULL subtree. We will consider this second case of merging.in our subsequent discussions.

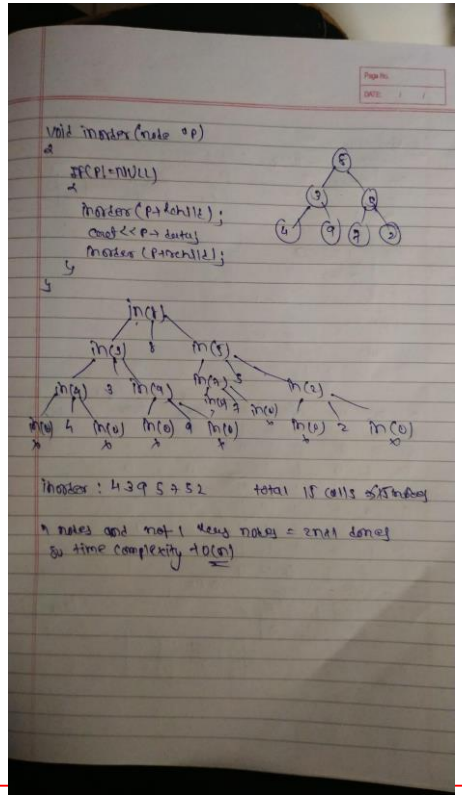
O(nlogn)

Different Types of Binary Trees

There are several types of binary trees, each with its own properties. Few important and frequently used trees are listed as below.

- Expression tree
- Threaded binary tree
- Binary search tree
- Height balanced tree (a.k.a. AVL tree)
- Red black tree
- Heap tree
- Huffman tree
- Splay tree
- Decision tree

inorder recursive tree

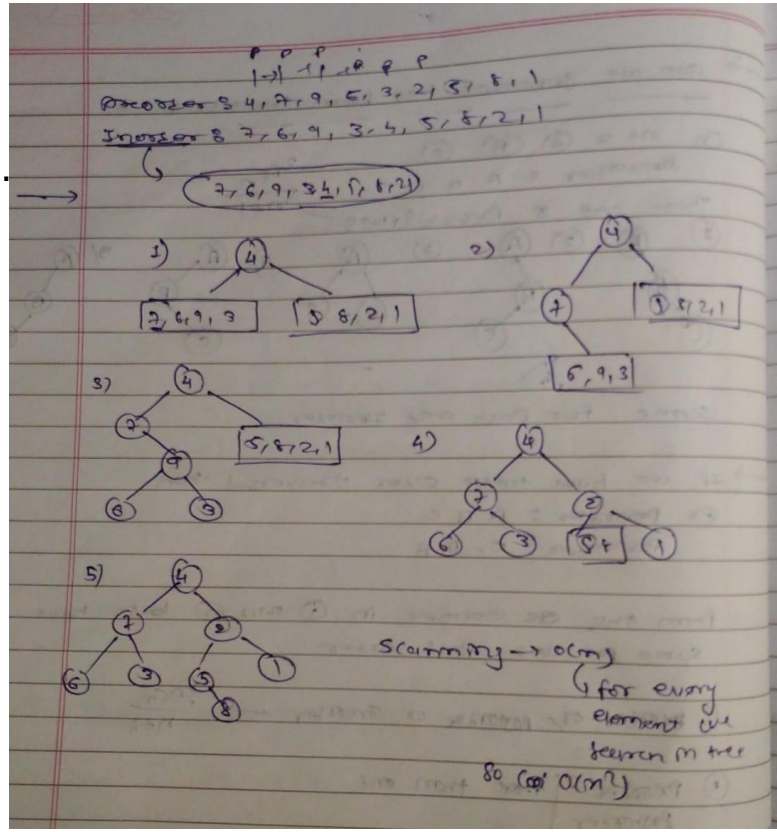


Can we generate tree from traversal ?

If we use only pretraversal or post then we can not generate .

For generating Binary tree we must need inorder traversal.

Preorder + inorder **OR** post order + inorder.
example:



time complexity in all traversal O(n) we can get this from recursive tree.

Next Lecture

- Expression, Threaded Binary, and Binary Search Tree

creating binary tree - >>> tree1.cpp
inorder ,preorder,postorder recursive ---> tree1.cpp for iterative method ---> tree3.cpp
level order->tree4.cpp

counting height,total data,total nodes , one degree nodes , 2 degree nodes , leaf nodes ->tree5.cpp

number of internal node --->tree5.cpp