Trees

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets T_1, ..., T_n, where each of these sets is a tree.
- We call T_1, ..., T_n the subtrees of the root.

TREE

A tree is a digraph with a nonempty set of nodes such that

(i) there is exactly one node, called the root of the tree, which has indegree o;

(ii) every node other than the root has indegree 1;

(iii) for every node a of the tree, there is a directed path from the root to a.

2 nodes with in degree 0 so not a tree

Bottom node has in degree 2 not a tree

No root

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

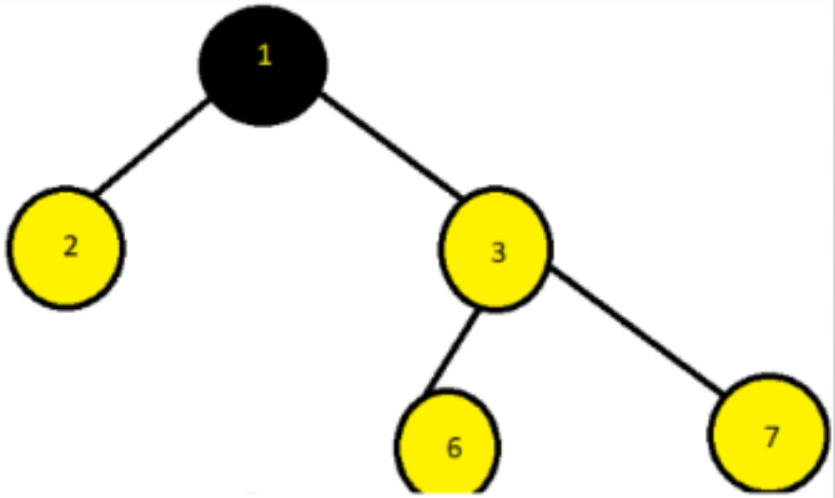
1□**Full binary tree:** Every node has 0 or 2 children.
2□**Complete binary tree:** All levels are filled except possibly the last, and all child nodes are to the left.
3□**Perfect binary tree:** All internal nodes have two children, and all leaf nodes are at the same level.

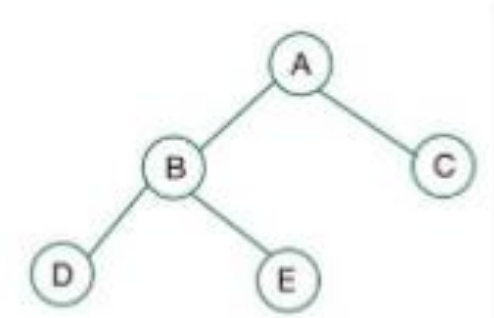
Full Binary Tree

A full binary tree is a type of binary tree in which every node has **0 or 2 children**. We can also say that in a full binary tree, all the nodes have two children except the leaf nodes(nodes at the last level). This also implies that **no particular order** is followed for filling in the nodes in the full binary tree. Also, the leaf nodes **might not be at the same level in a full binary tree.**

Some Important Points

1. If the number of leaf nodes is l, and internal nodes are i, then **i=l-1.**
2. If the number of leaf nodes is l, and the total number of nodes in the tree is n, then **n=2l-1.**
3. **Let us assume that** number of nodes in the tree is n, and the number of leaf nodes is l, then **l=(n+1)/2.**
4. If the number of internal nodes is I and the number of leaf nodes is l, then **l=i+1.**





A Full Binary Tree

Let, ***i*** be the number of internal nodes

n be the total number of nodes

l be number of leaves

λ be number of levels

Then,

The number of leaves is ***(i + 1)***.

The total number of nodes is ***(2i + 1)***.

The number of internal nodes is ***(n – 1) / 2***.

The number of leaves is ***(n + 1) / 2***.

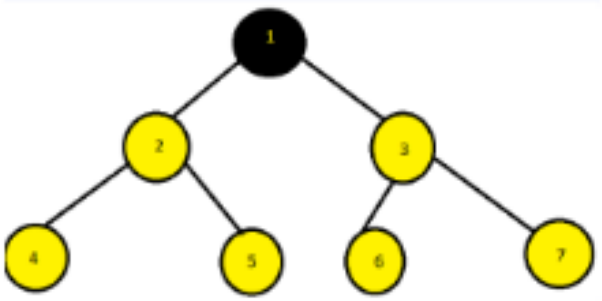
The total number of nodes is ***(2l – 1)***.

The number of internal nodes is ***(l – 1)***.

The number of leaves is at most $(2^{\lambda} - 1)$.

Complete Binary Tree

It is nothing but a type of binary tree in which **all levels are filled** except possibly the last level. The nodes in the last level of a complete binary tree should be **as left as** possible. This indicates that a **particular order** is followed for the filling in nodes in a complete binary tree from left to right. We can draw another inference that the leaf nodes of a complete binary tree **must be at the same level.** Let us understand the complete binary tree clearly, through an example.



In the given binary tree,
1. All levels are filled.
2. All the nodes start from the left.

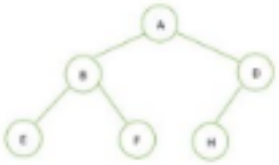
Hence this is a complete binary tree.

Properties of Complete Binary Tree:

- A complete binary tree is said to be a proper binary tree where all leaves have the same depth.
- In a complete binary tree number of nodes at depth **d** is 2^d.
- In a complete binary tree with **n** nodes height of the tree is **log(n+1)**.
- All the levels **except the last level** are completely full.

Relation between number of leaf nodes and nodes of degree 2

For any nonempty binary tree, T,
if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then
n0= = n2 + 1.



A binary tree

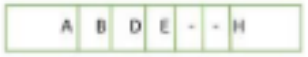
Height of the given binary tree is 2 and the maximum number of nodes that should be there are $2^{h+1} - 1 = 2^{2+1} - 1 = 2^3 - 1 = \mathbf{7}$.
But the number of nodes in the tree is **6**. Hence it is **not a perfect binary tree**.
Now for a complete binary tree, It is full up to height **h-1** i.e.; **1**, and the last level element are stored in left to right order. Hence this is a **complete binary tree**. Store the element in an array and it will be like;

	A	B	D	E	F	H	

Element stored in an array level by level

A binary tree

The height of the binary tree is 2 and the maximum number of nodes that can be there is 7, but there are only 5 nodes hence it is **not a perfect binary tree**. In case of a complete binary tree, we see that in the last level elements are not filled from left to right order. So it is **not a complete binary tree**.



Element stored in an array level by level

The elements in the array are not continuous.

Types of Binary Trees

➢

1- Full Binary Tree

A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.

In a Full Binary, number of leaf nodes is number of internal nodes plus 1

L = I + 1

Where L = Number of leaf nodes, I = Number of internal nodes

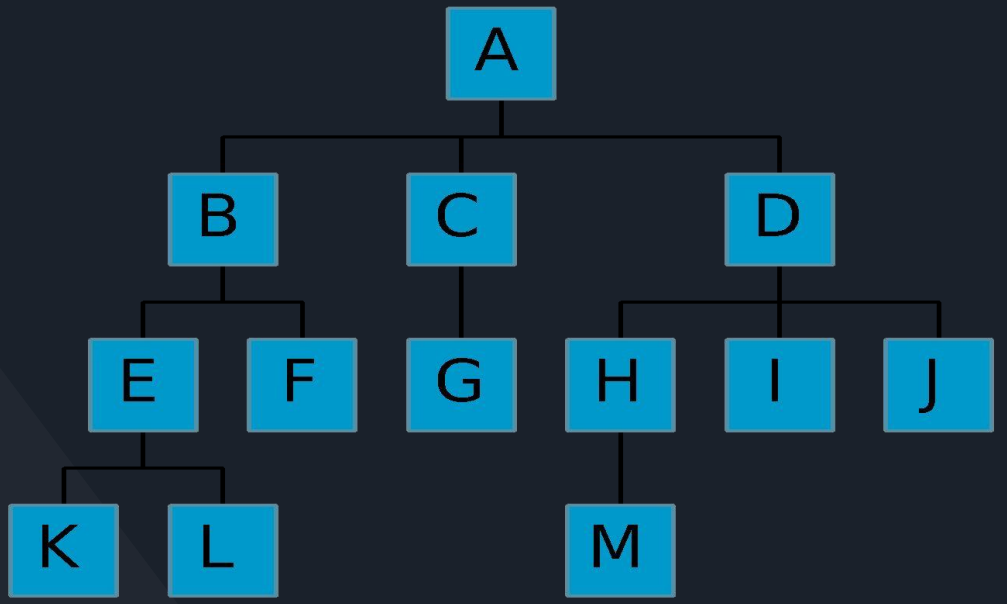
2- Complete Binary Tree:

A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

3- Perfect Binary Tree

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

No.of nodes in a perfect binary tree is: 2^h+1 - 1



Leve

0

1

2

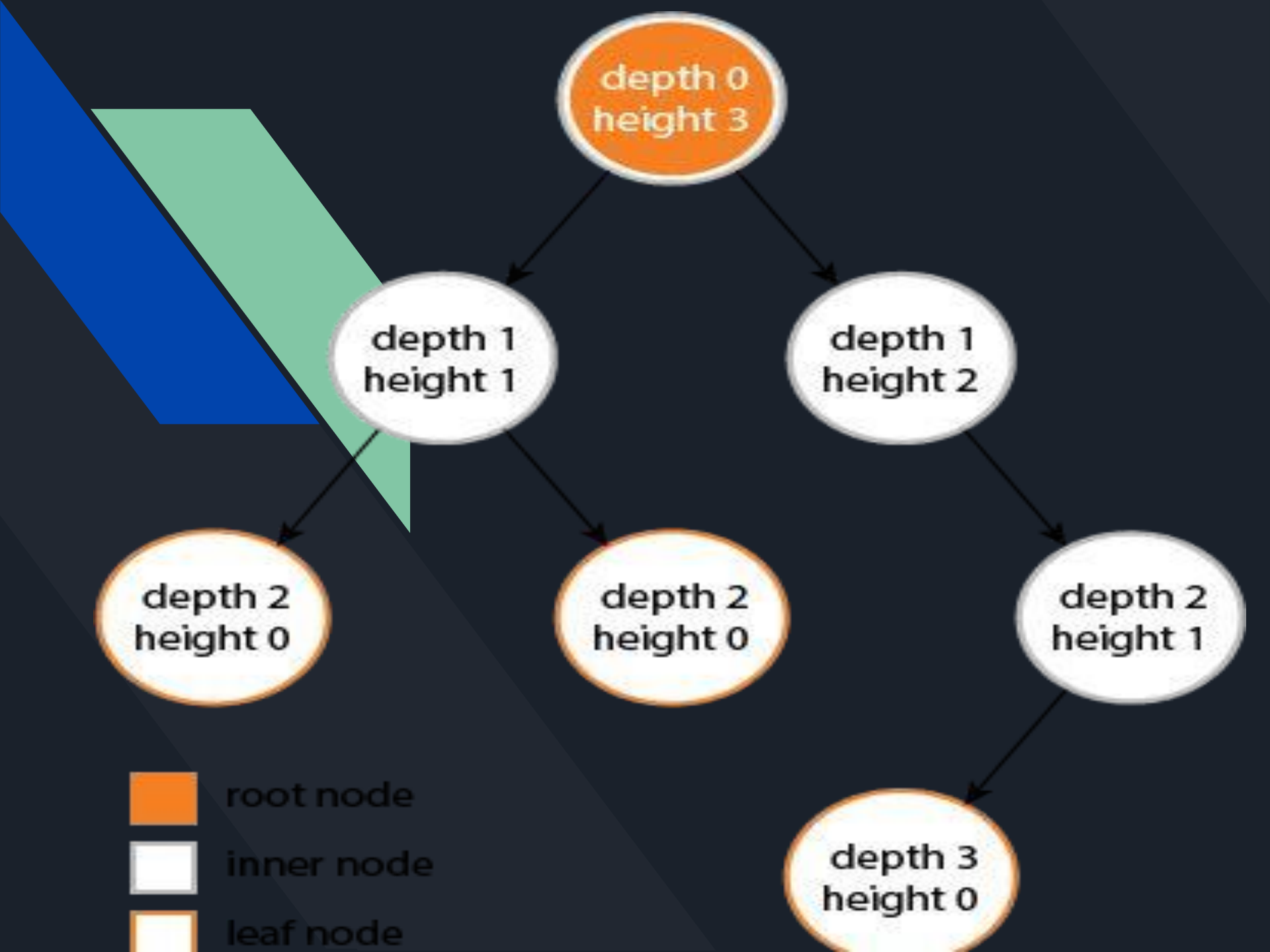
3

- The **depth** of a node is the number of edges from the node to the tree's root node.

  A root node will have a depth of 0.
- The **height** of a node is the number of edges on the *longest path* from the node to a leaf.

  A leaf node will have a height of 0.

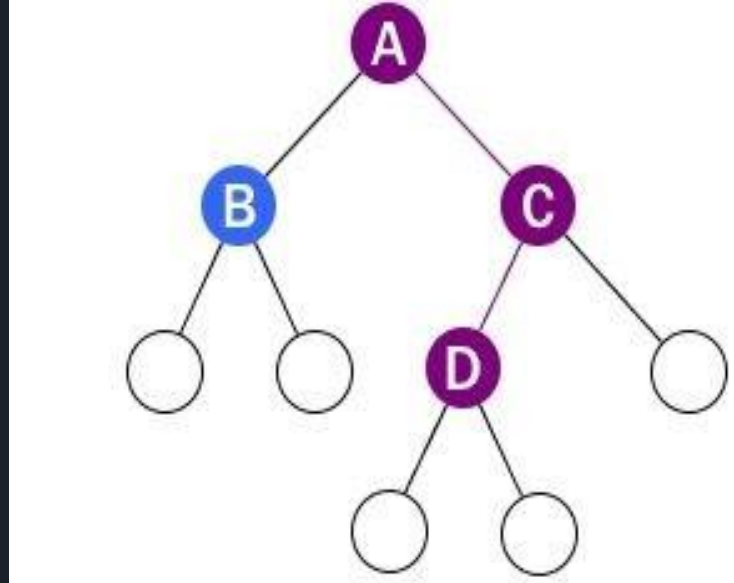
t


depth 0
height 3
depth 1
height 1
depth 1
height 2
depth 2
height 0
depth 2
height 0
depth 2
height 1
depth 3
height 0
root node
inner node
leaf node

The depth of a node X in a tree T is defined as the length of the simple path (number of edges) from the root node of T to X.

The height of a node Y is the number of edges on the **longest** downward simple path from Y to a leaf.

The height of a tree is defined as the height of its root node.



node A:

Height = 3 (no. of edges on longest path from A is

A-> C -> D -> leaf i.e. 3 edges)

depth = 0

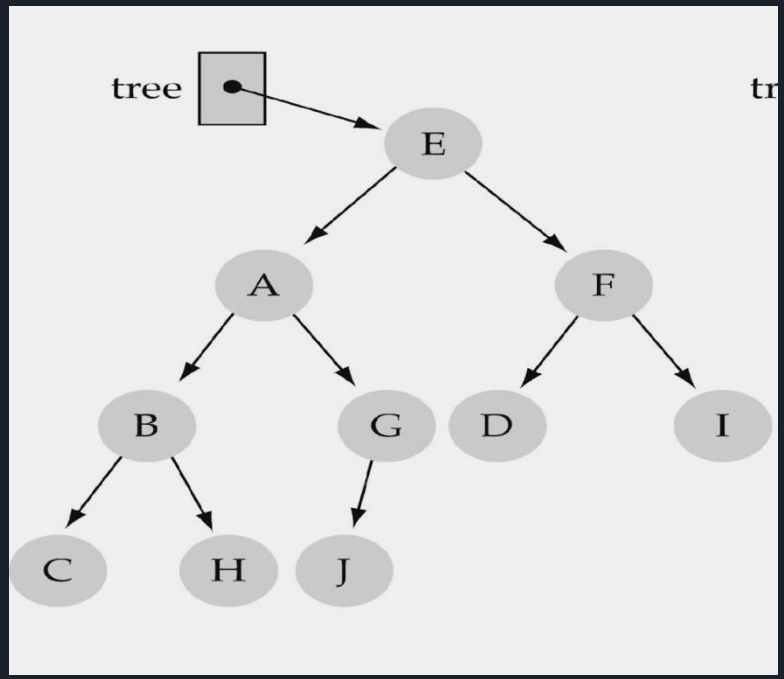
node D:

Height = 1 (D-> leaf i.e., 1 edge)

Depth = 2 (D -> C -> A is the no. of edges from node to root A)

Some terminology

- The successor nodes of a node are called its *children*
- The predecessor node of a node is called its *parent*
- The "beginning" node is called the *root* (has no parent)
- A node without *children* is called a *leaf*



A Taxonomy of Trees

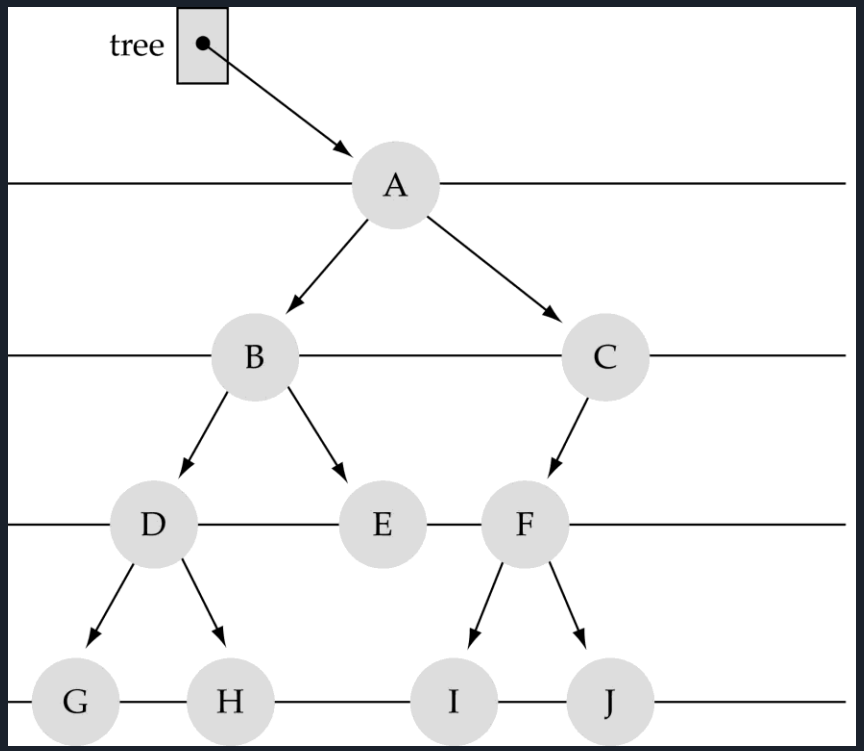
General Trees – any number of children / node

Binary Trees – max 2 children / node

Heaps – parent < (>) childrenBinary Search Trees

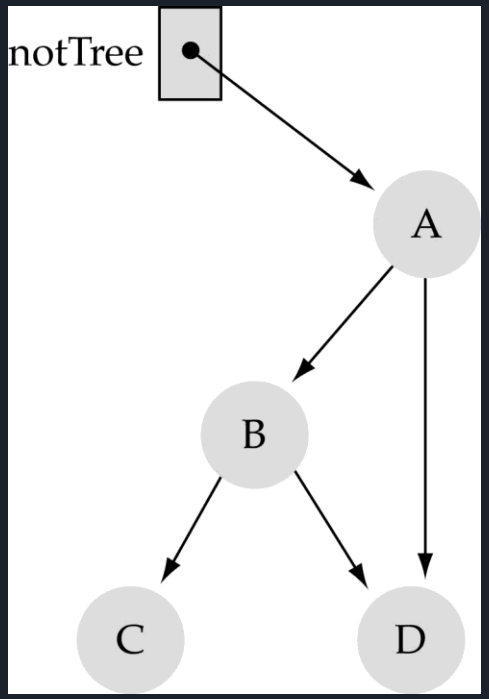
What is a binary tree?

- *Property 1*: each node can have up to two successor nodes.



What is a binary tree? (cont.)

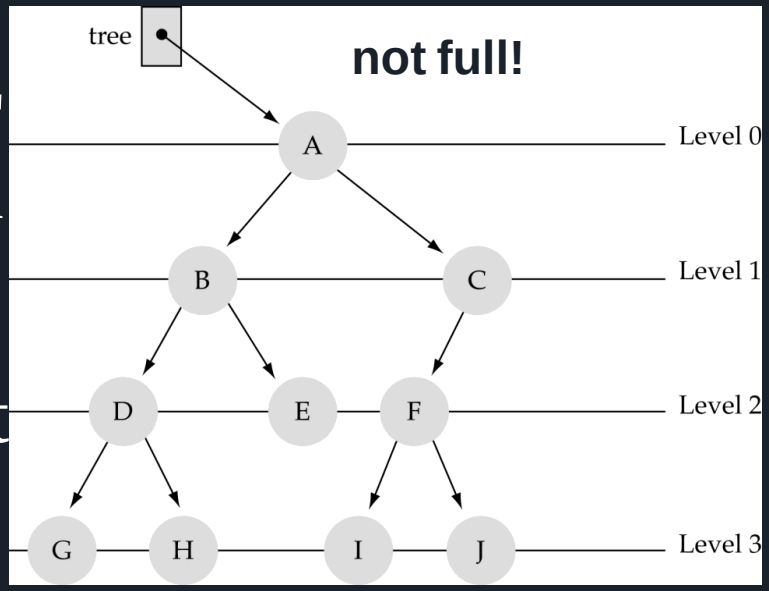
- *Property 2*: a unique path exists from the root to every other node



Not a valid binary tree!

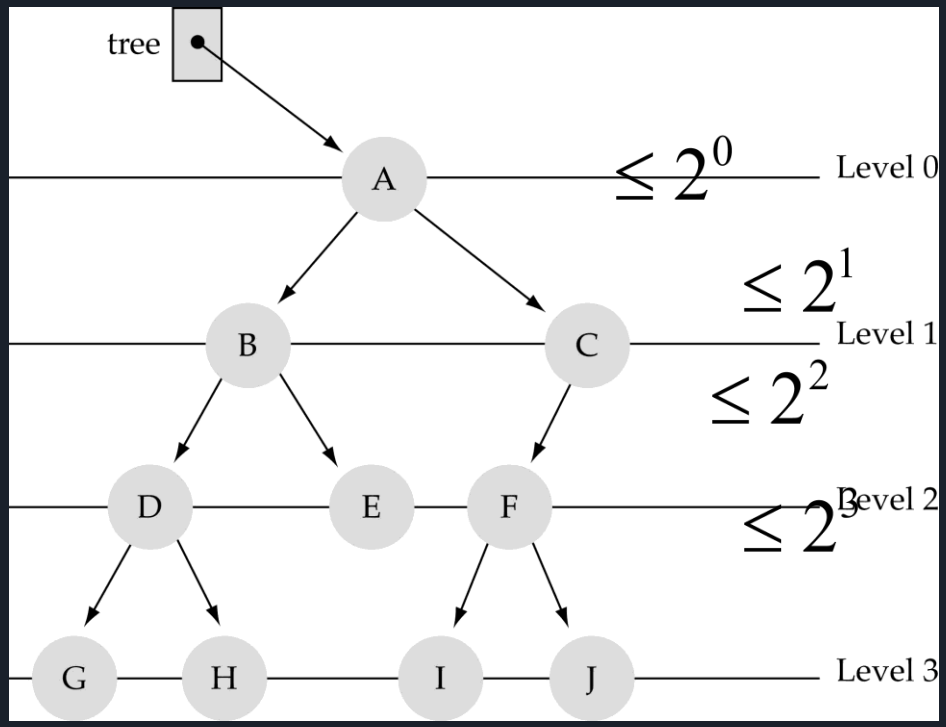
Some terminology (cont'd)

- Nodes are organized in levels (indexed from 0).
- **Level (or depth) of a node**: number of edges in the path from the root to that node.
- **Height of a tree h**: #levels = L
  (Warning: some books define h as #levels-1).
- **Full tree:** every node has exact two children ***and*** all the leaves are on the same level.



What is the max #nodes at some level *l*?

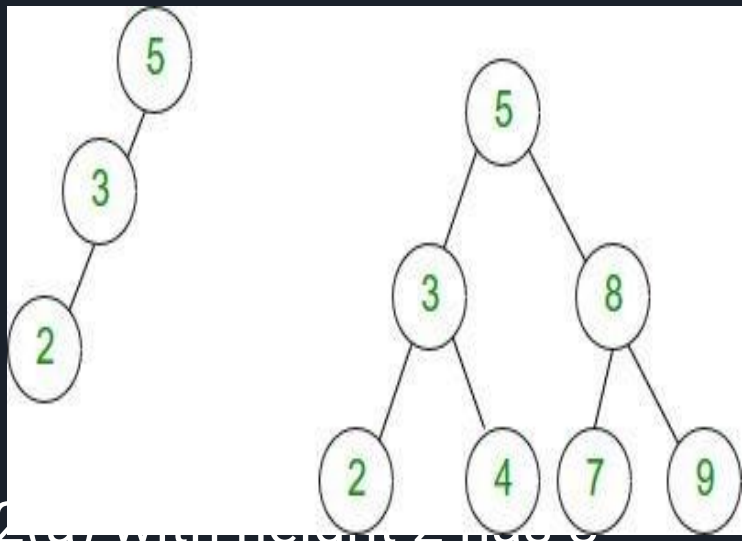
The max #nodes at level l is 2^l where *l=0,1,2, ...,L-1*



What is the total #nodes **N** of a full tree with height **h**?

Total number of nodes will be 2^0 + 2^1 + …. 2^h = 2^(h+1)-1.

Calculating minimum and maximum number of nodes from height



If binary tree has **height h**, minimum number of **nodes is h+1** (in case of left skewed and right skewed binary tree).

For example, the binary tree shown in Figure 2(a) with height 2 has 3 nodes.

If binary tree has height h, maximum number of nodes will be when all levels are completely full.

Total number of nodes will be 2^0 + 2^1 + …. 2^h = 2^(h+1)-1.

For example, the binary tree shown in Figure 2(b) with height 2 has 2^(2+1)-1 = 7 nodes.

Que-1. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and the minimum number of nodes in a binary tree of height 5 are:

(A) 63 and 6, respectively

(B) 64 and 5, respectively

(C) 32 and 6, respectively

(D) 31 and 5, respectively

Solution: According to formula discussed,

max number of nodes = 2^(h+1)-1 = 2^6-1 =63.

min number of nodes = h+1 = 5+1 = 6.

Que-2. Which of the following height is not possible for a binary tree with 50 nodes?

(A) 4

(B) 5

(C) 6

(D) None

Solution: According to formula discussed,

Minimum height with 50 nodes = floor(log250) = 5. Therefore, height 4 is not possible.

What is the height **h** of a full tree with **N** nodes?

$$2^h - 1 = N$$

$$\Rightarrow 2^h = N + 1$$

$$\Rightarrow h = \log(N+1) \rightarrow O(\log N)$$

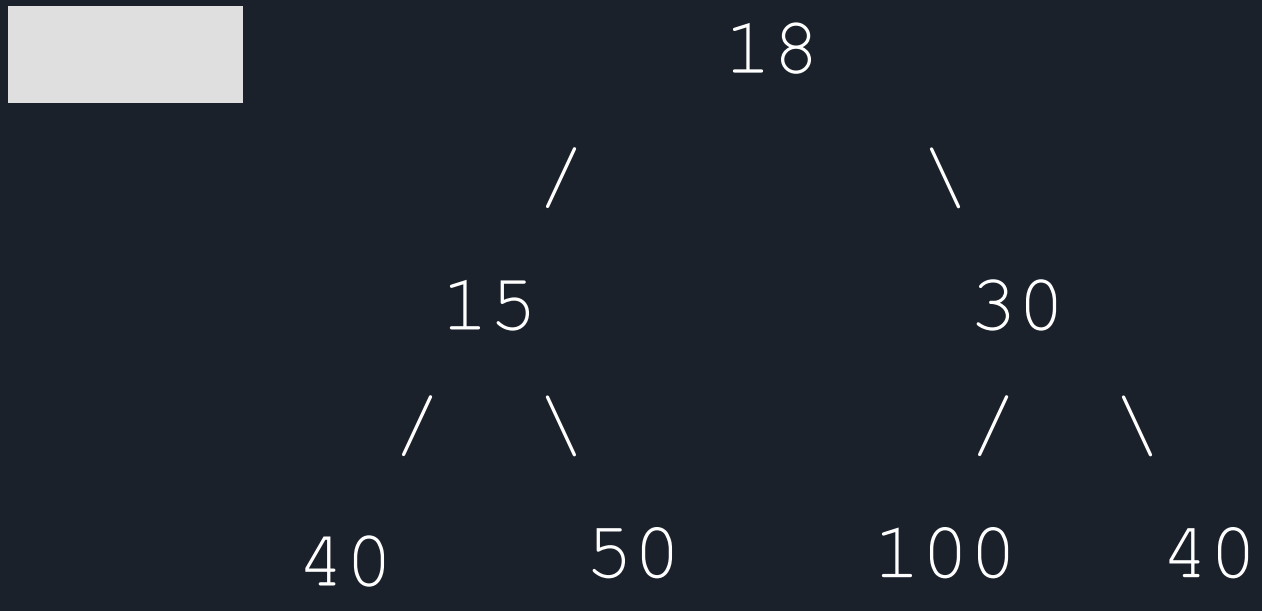
Why is h important?

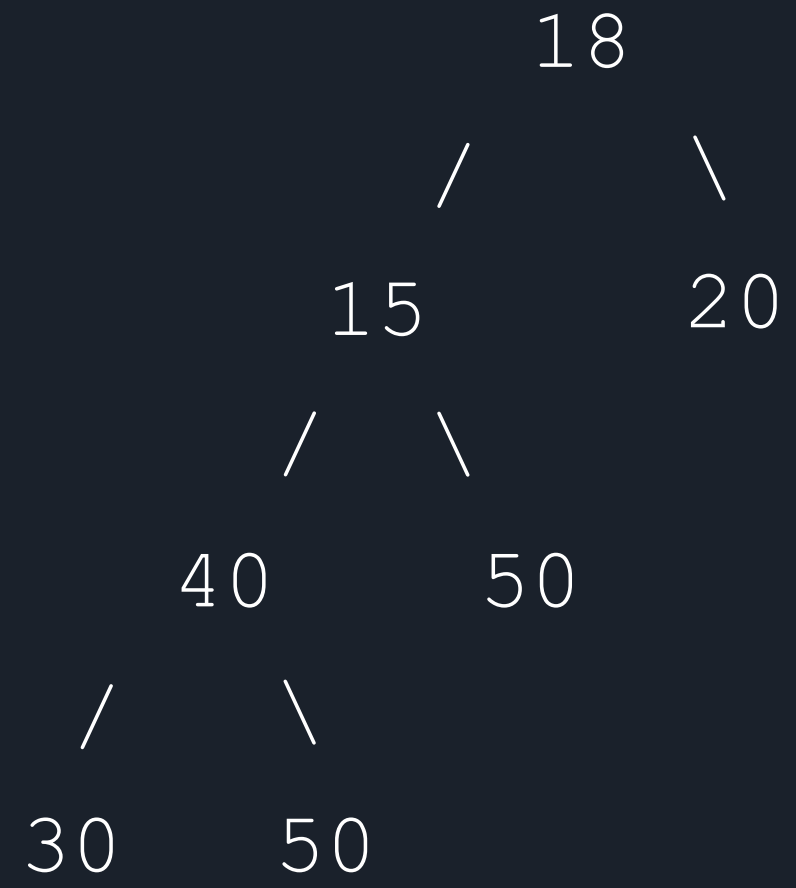
- Tree operations (e.g., insert, delete, retrieve etc.) are typically expressed in terms of h.

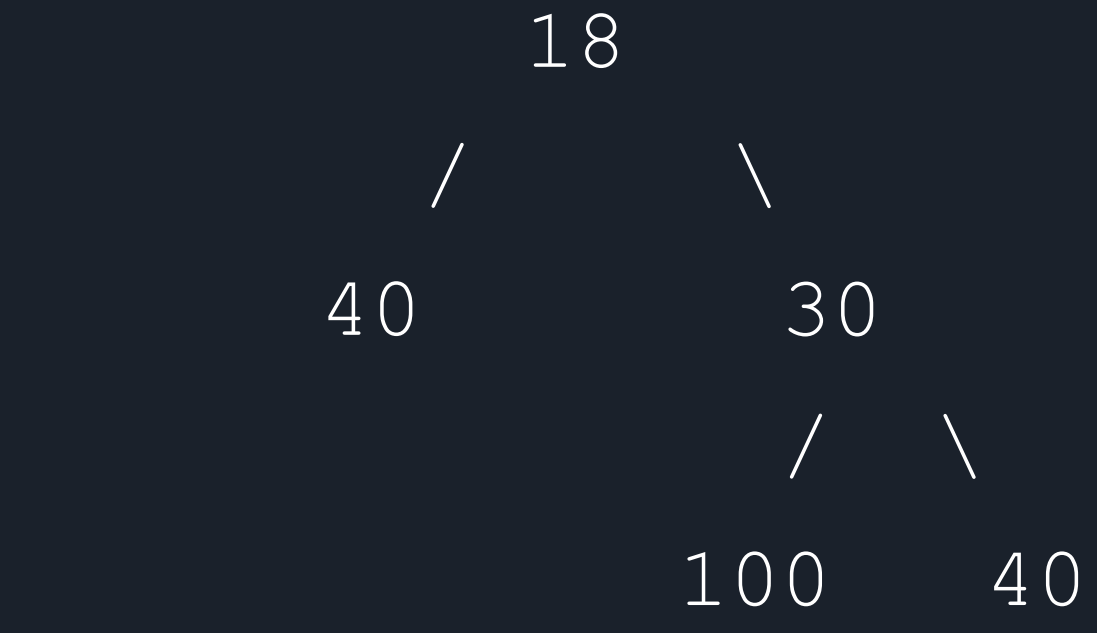
- So, h determines running time!

Types of Binary trees

1- **Full Binary Tree** : A Binary Tree is a full binary tree if every node has 0 or 2 children.




18
15
20
40
50
30
50



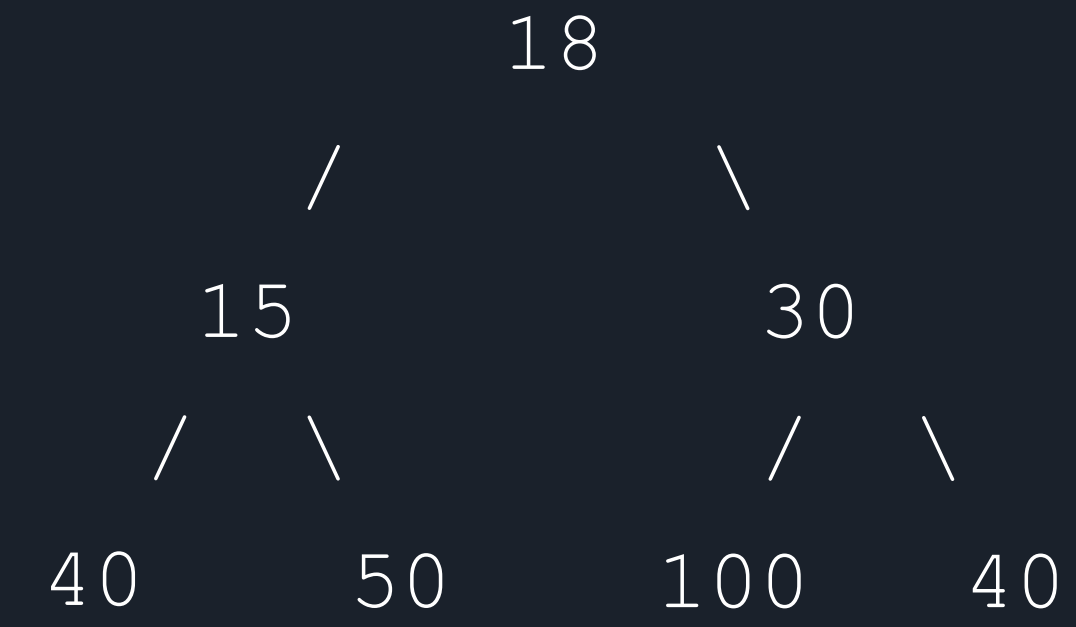
In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1

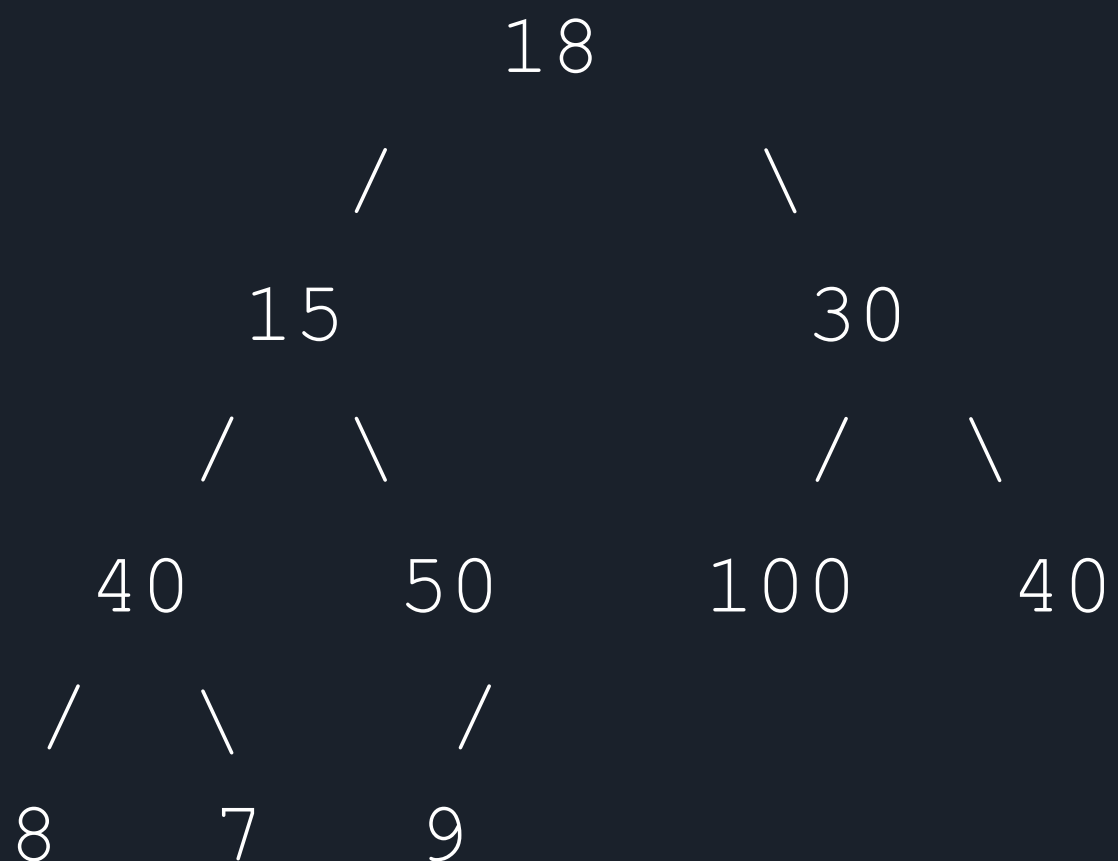
$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree

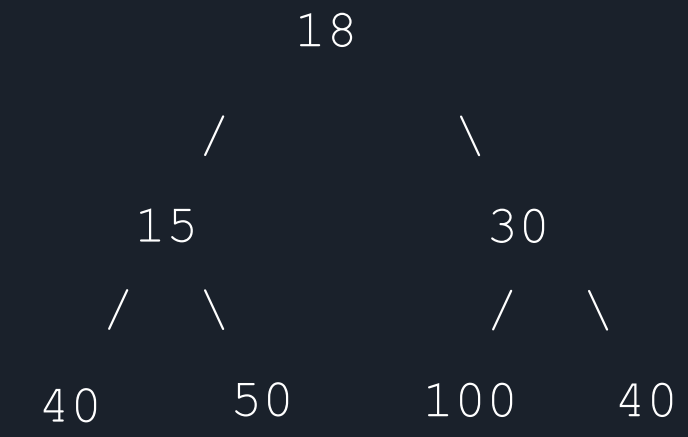
A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible


18
15
30
40
50
100
40


18
15
30
40
50
100
40
8
7
9

Perfect Binary Tree A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

The following are the examples of Perfect Binary Trees.

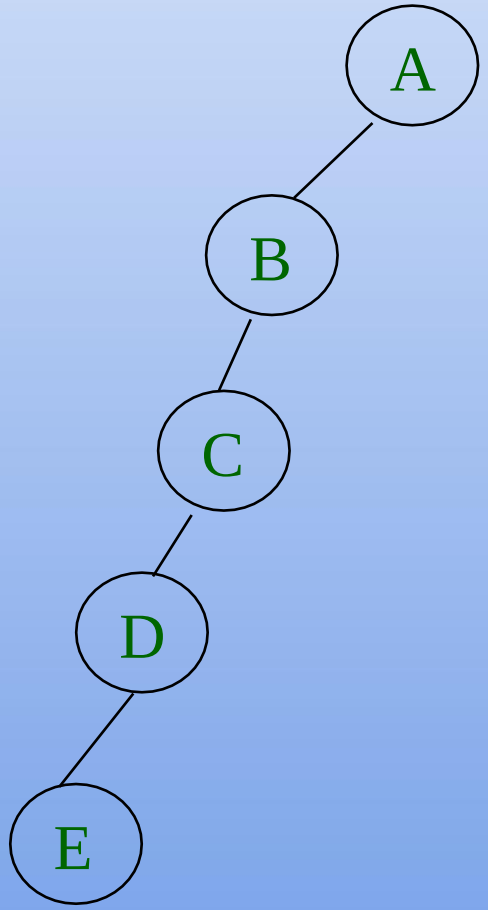


A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has 2h – 1 node.

Binary Tree Representations

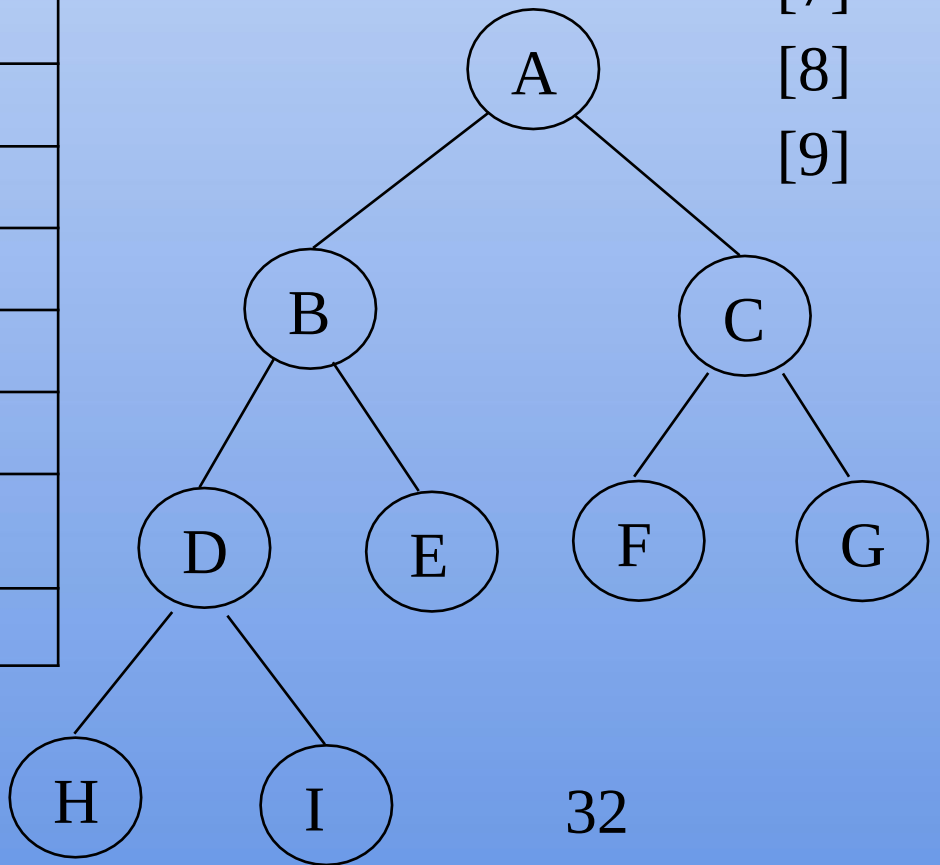
- If a complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:
  - *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
  - *left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.
  - *right_child*(*i*) ia at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

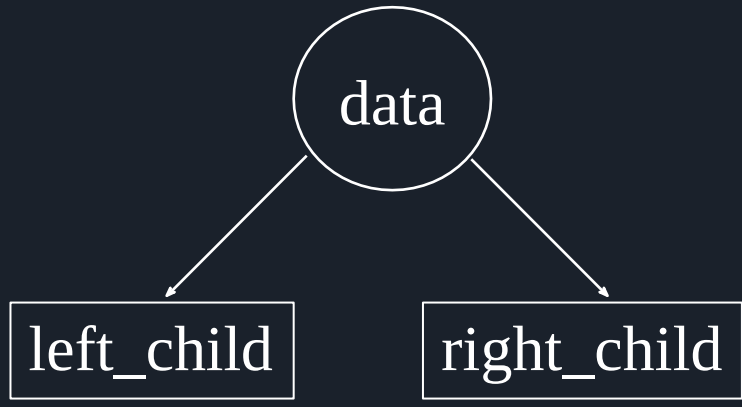
(1) waste space
(2) insertion/deletion problem



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Linked Representation

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```



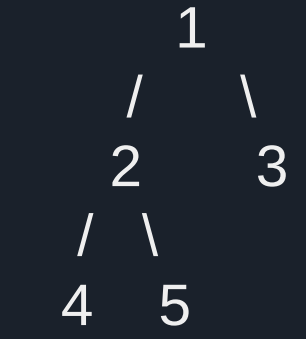
```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

```
/* newNode() allocates a new node with
the given data and NULL left and
   right pointers. */
struct node* newNode(int data)
{
  // Allocate memory for new node
struct node* node = (struct
node*)malloc(sizeof(struct node));

  // Assign data to this node
  node->data = data;

  // Initialize left and right children as NULL
  node->left = NULL;
  node->right = NULL;
  return(node);
}
```

```
int main()
{
  /*create root*/
  struct node *root = newNode(1);
  /* following is the tree after above
statement
        1
      /   \
 NULL  NULL
  */

 root->left        = newNode(2);
  root->right       = newNode(3);
```

```
/* 2 and 3 become left and
right children of 1
          1
        /   \
       2      3
     /    \    /  \
 NULL NULL NULL NULL
  */
root->left->left  =
newNode(4);
  /* 4 becomes left child of 2
           1
       /       \
      2          3
    /   \       /  \
   4    NULL  NULL  NULL
  /  \
NULL NULL
*/
```

Tree Traversals (Inorder, Preorder and Postorder)



Following are the generally used ways for traversing trees.

Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Inorder Traversal:

Algorithm Inorder(tree)
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

```
/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}
```

Preorder Traversal

Algorithm Preorder(tree)

1. Visit the root.

2.Traverse the left subtree, i.e., call Preorder (left-subtree)

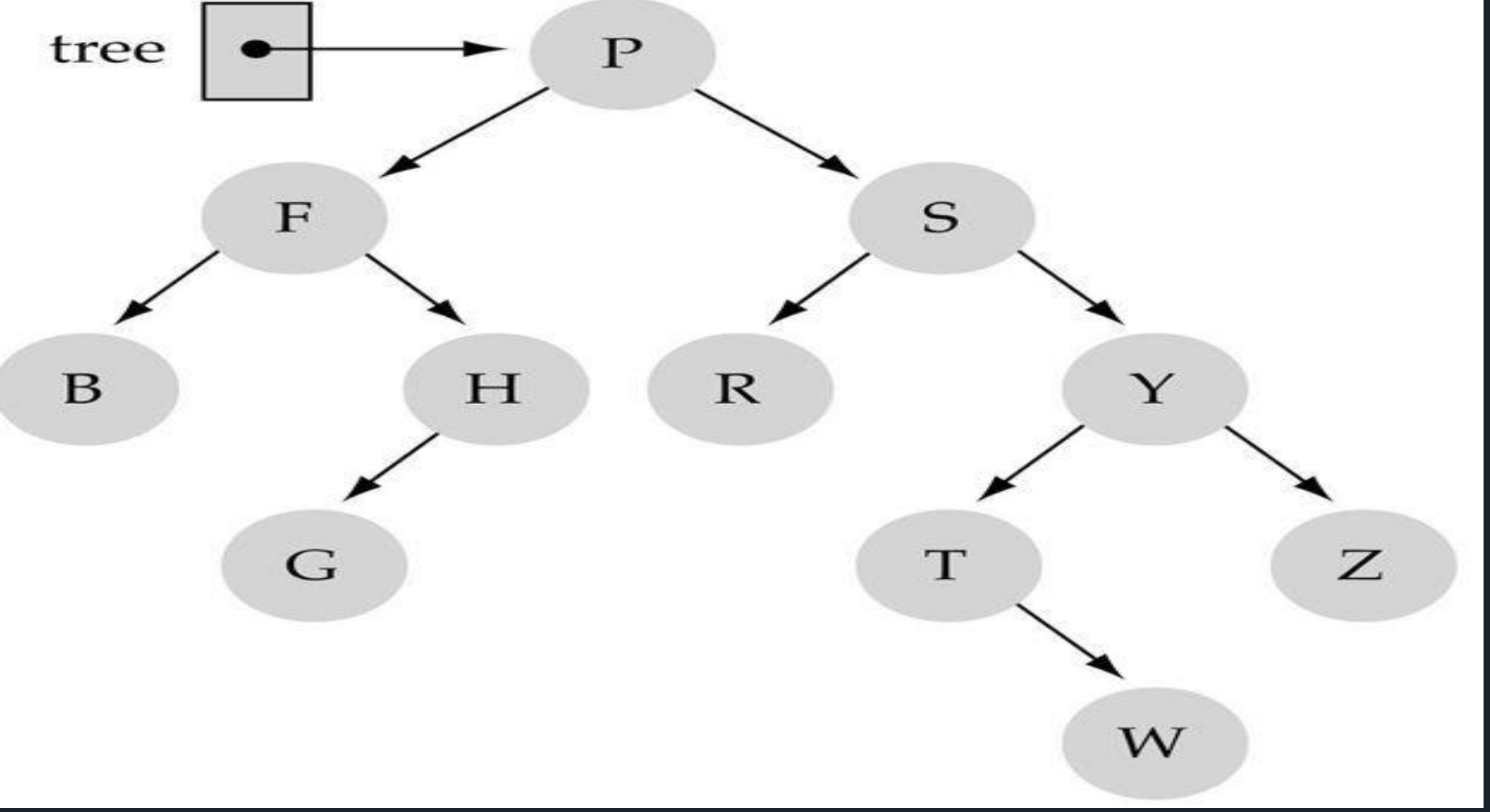
3.Traverse the right subtree, i.e., call Preorder (right-subtree)

```
/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}
```

Postorder Traversal

Algorithm Postorder(tree)
1.Traverse the left subtree, i.e., call Postorder(left-subtree)
2.Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

```
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}
```



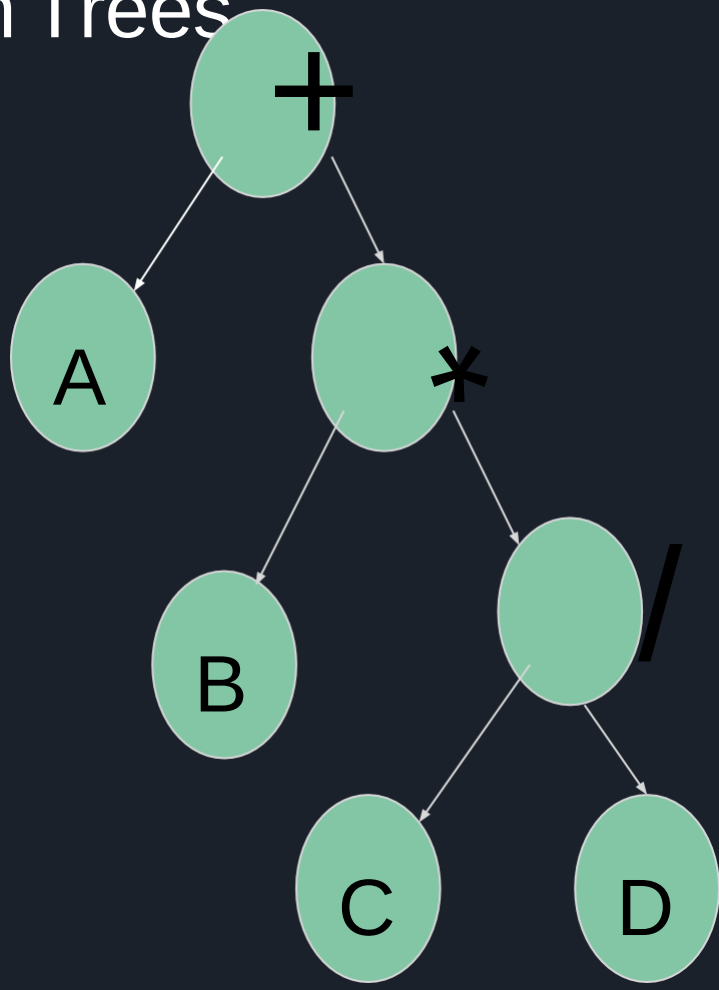
Inorder:	B	F	G	H	P	R	S	T	W	Y	Z
Preorder:	P	F	B	H	G	S	R	Y	T	W	Z
Postorder:	B	G	H	F	R	W	T	Z	Y	S	P

Application: Arithmetic Expression Trees

Example Arithmetic Expression:

A + (B * (C / D))

Tree for the above expression:



Used in most compilers

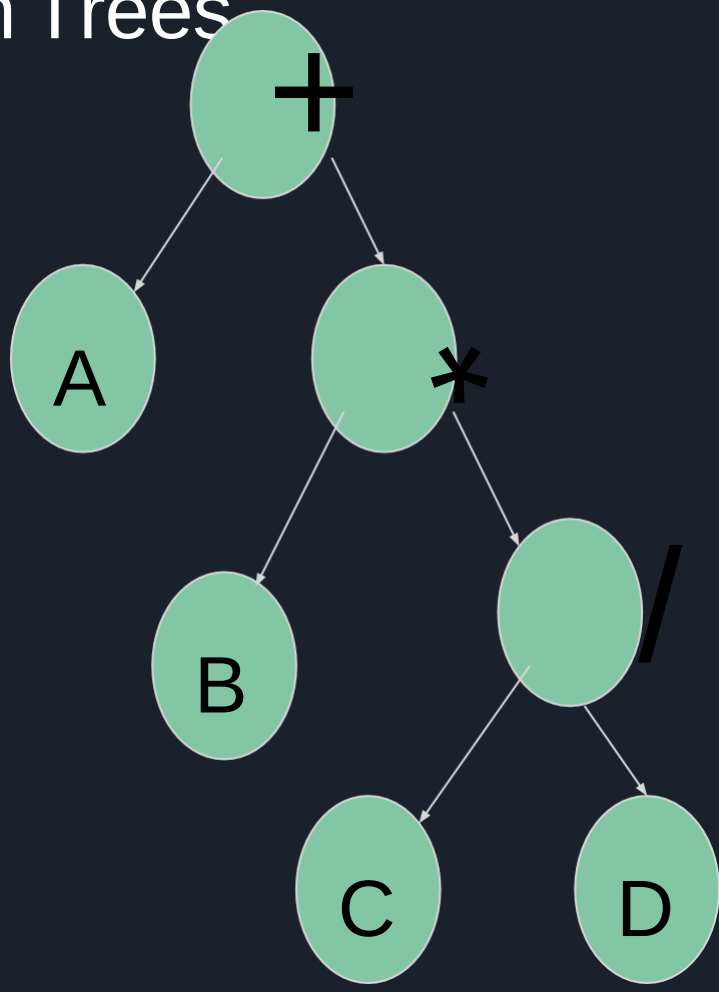
- No parenthesis need – use tree structure
- Can speed up calculations e.g. replace
  - / node with C/D if C and D are known
- Calculate by traversing tree (how?)

Application: Arithmetic Expression Trees

Example Arithmetic Expression:

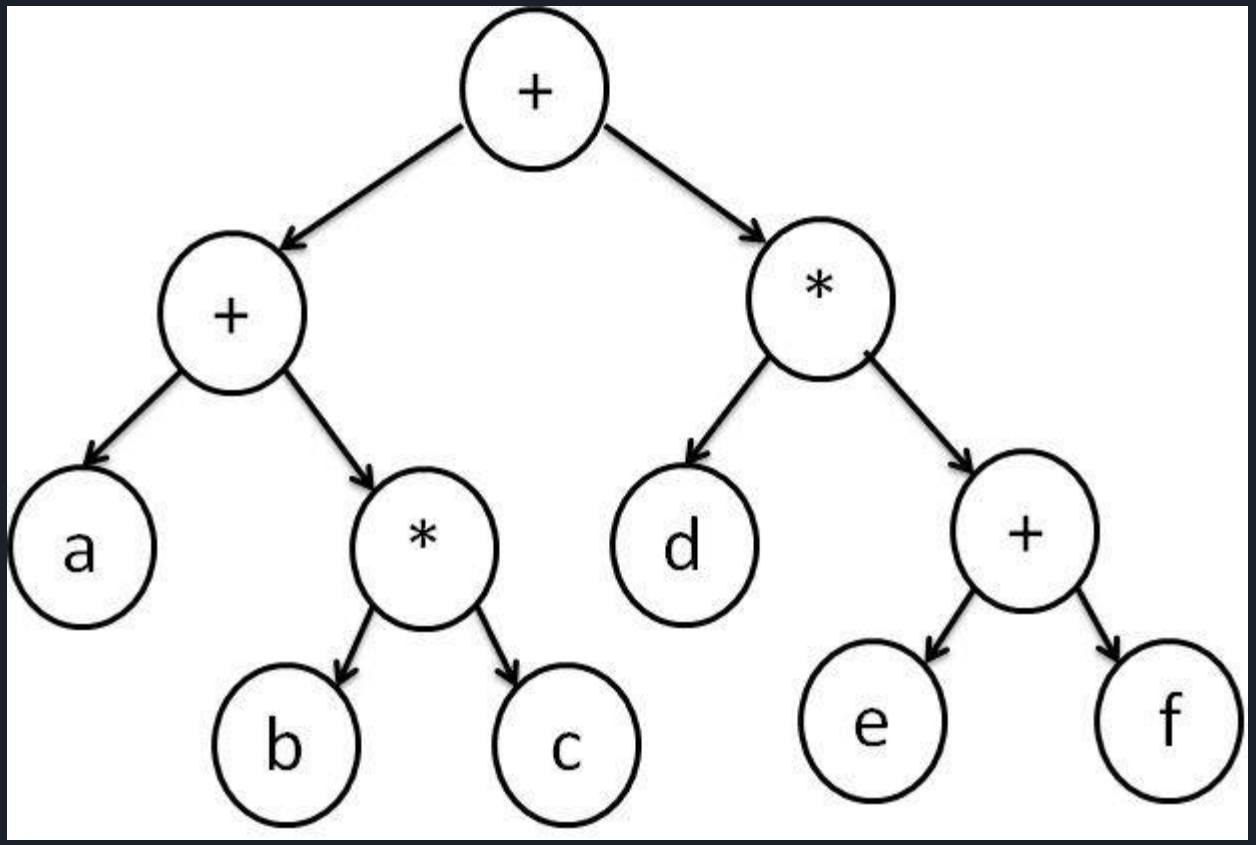
A + (B * (C / D))

Tree for the above expression:



Preorder: Root, then Children
+ A * B / C D
Postorder: Children, then Root
A B C D / * +
Inorder: Left child, Root, Right child
A + B * C / D

Expression: a + (b * c) + d * (e + f)



Postfix Expression:

```
a b c * + d e f + * +
```

Prefix Expression:

```
+ + a * b c * d + e f
```

Construction of Expression Tree

Input is postfix expression

Following are the step to construct an expression tree:

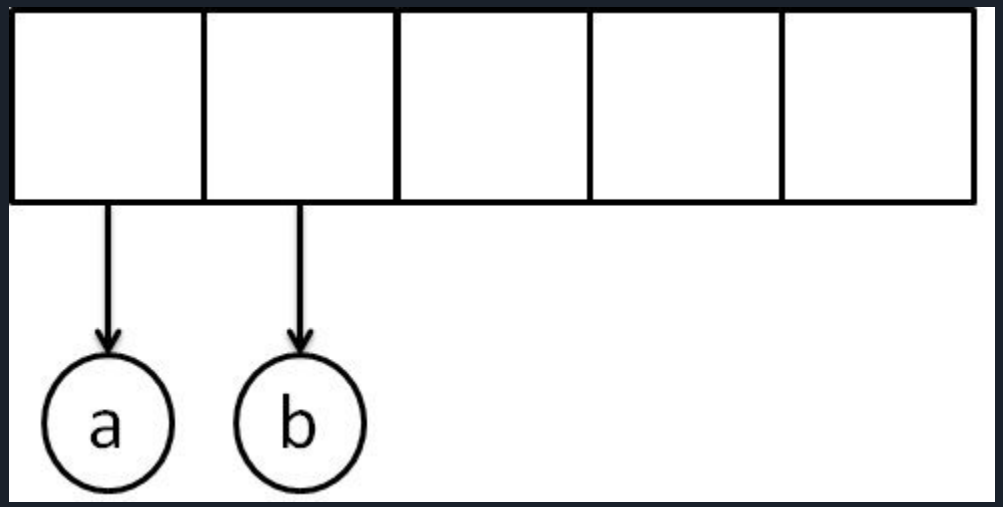
1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and pushed a pointer onto a stack
4. If the symbol is an operator, pop two pointer from the stack namely T1 & T2 and form a new tree with root as the operator, T1 & T2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

Example

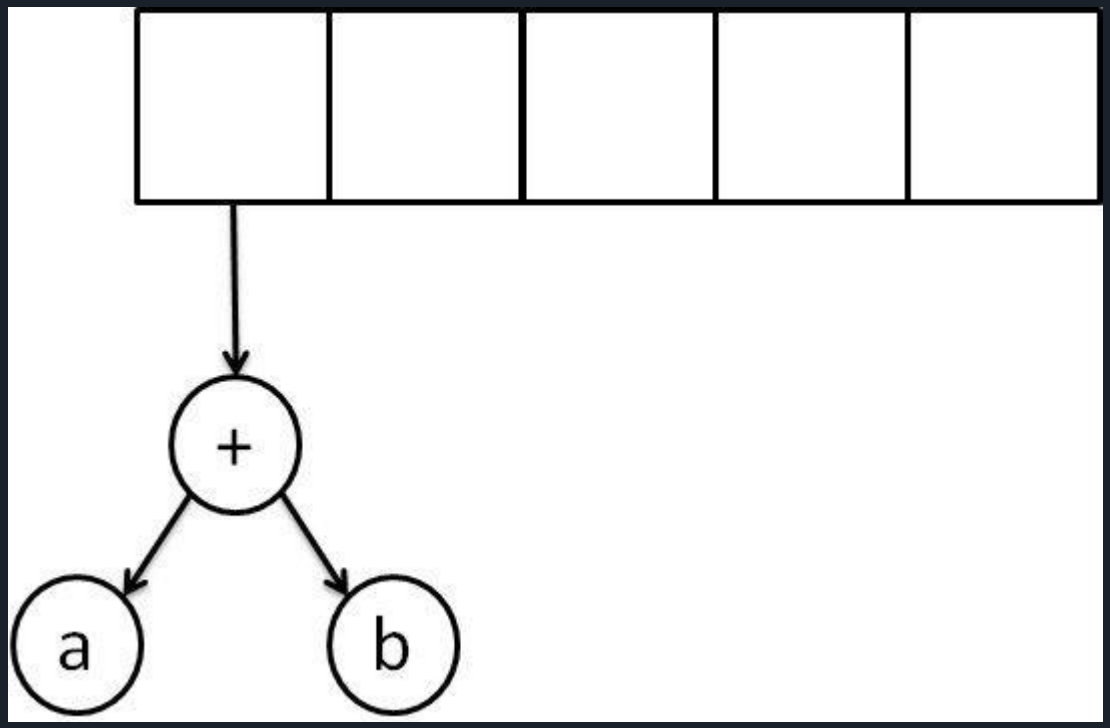
The input is:

```
a b + c *
```

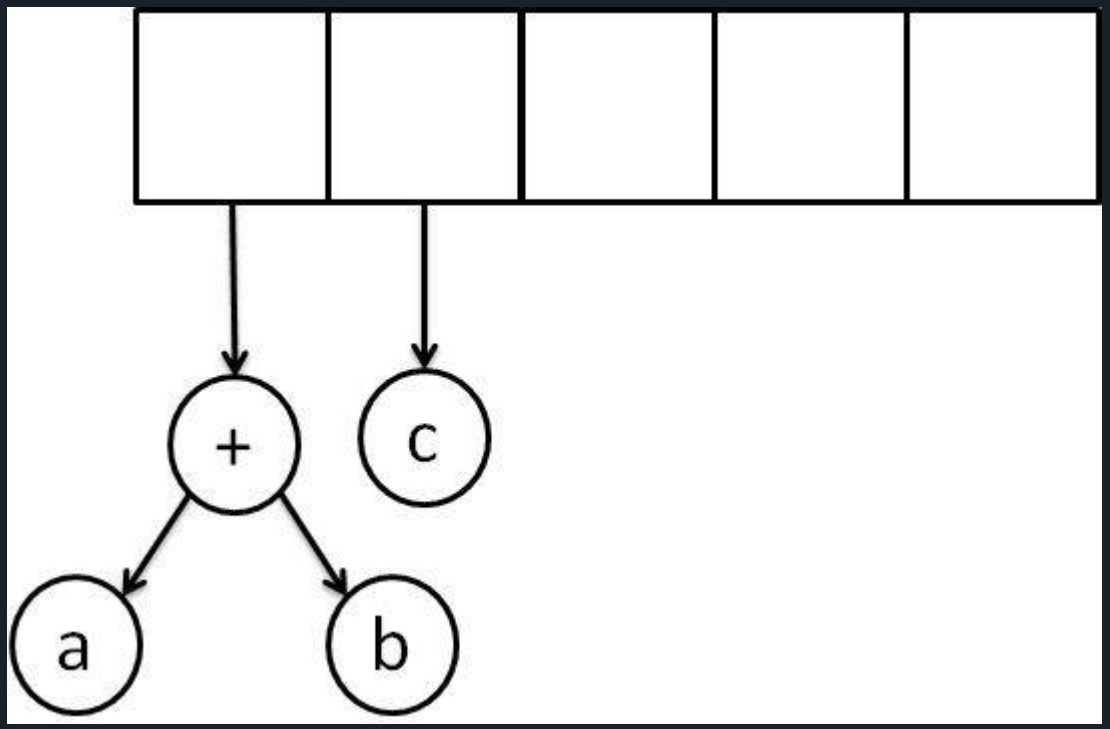
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



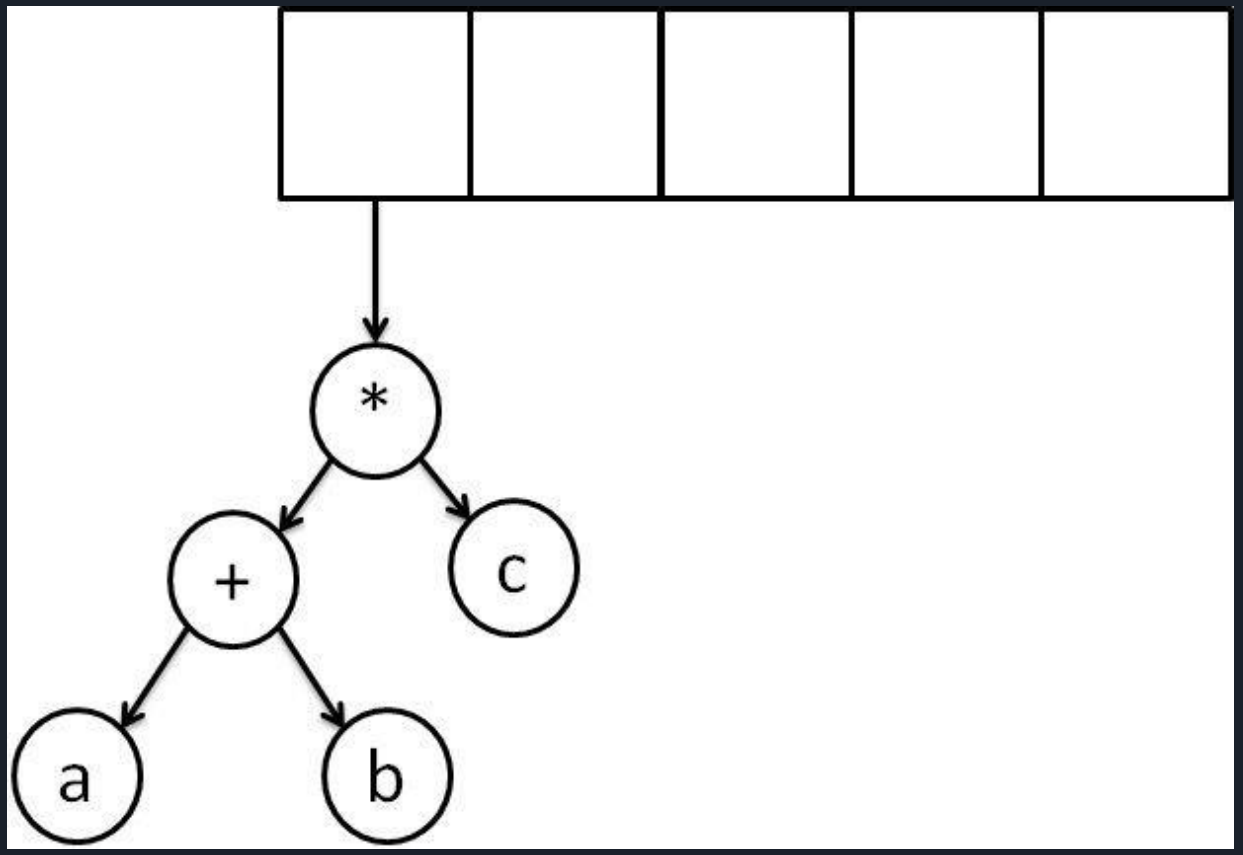
Next, read a'+' symbol, so two pointers to tree are popped,a new tree is formed and push a pointer to it onto the stack.



Next, '**c**' is read, we create one node tree and push a pointer to it onto the stack.

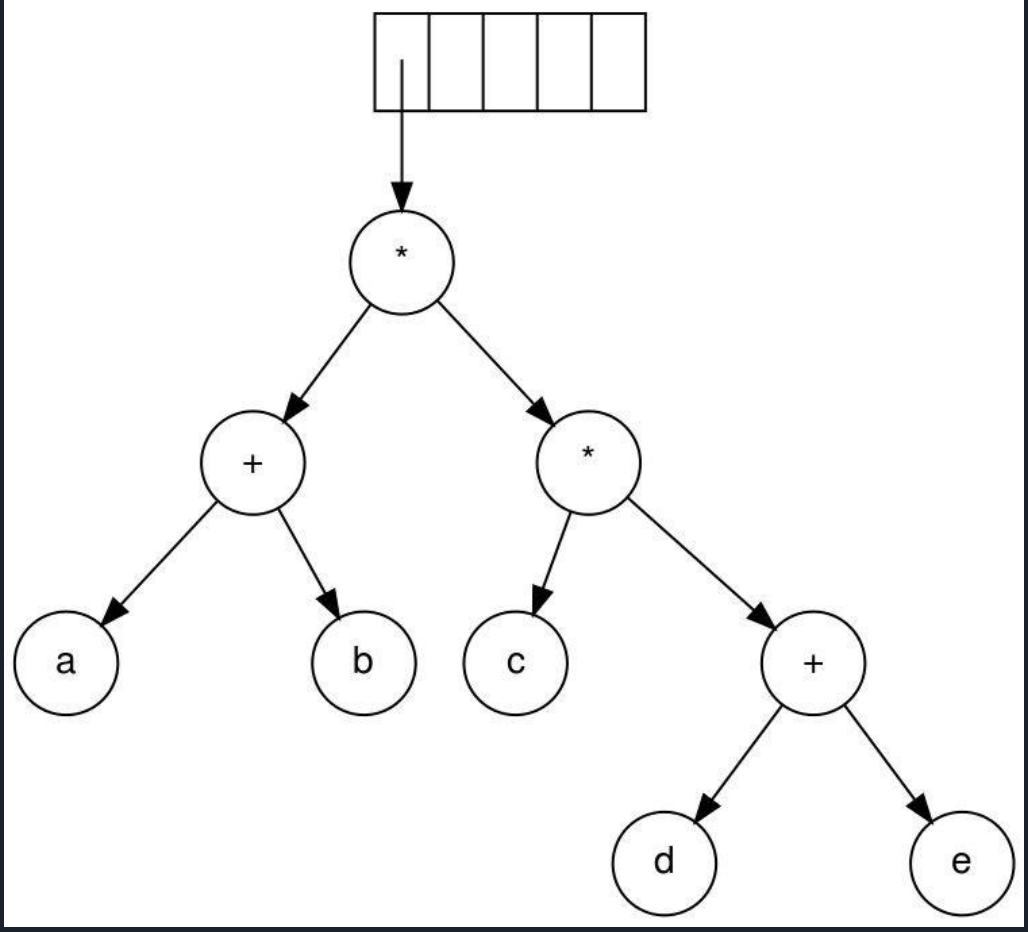


Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.



Homework:

The input in postfix notation is: a b + c d e + * *



constructing Binary tree from given Inorder and postorder sequence

in[] = {4, 8, 2, 5, 1, 6, 3, 7}

post[] = {8, 4, 5, 2, 6, 7, 3, 1}

1)We first find the last node in post[]. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.

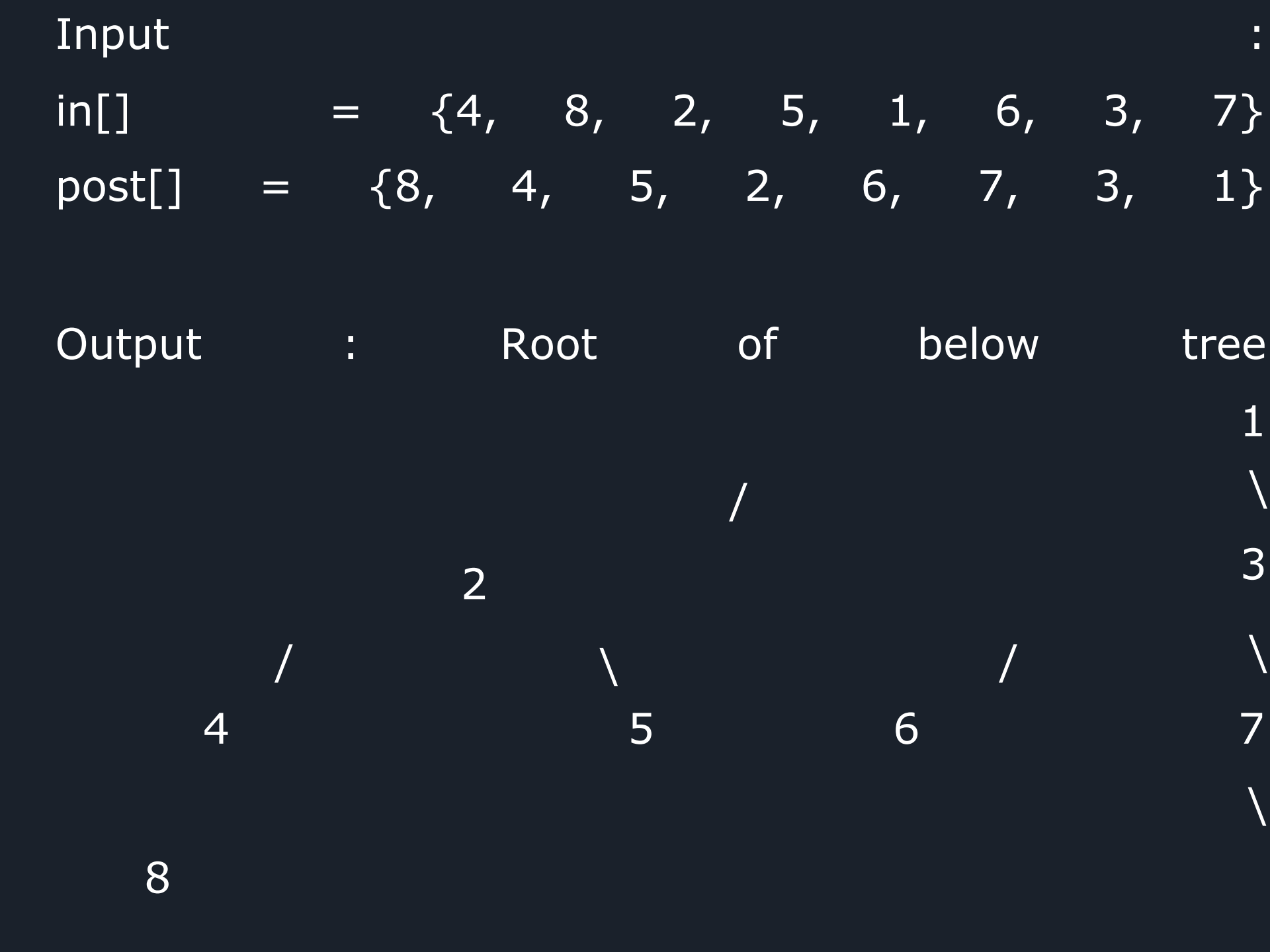
2)We search “1” in in[] to find left and right subtrees of root. Everything on left of “1” in in[] is in left subtree and everything on right is in right subtree.

Input :

in[] = {4, 8, 2, 5, 1, 6, 3, 7}

post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Output : Root of below tree



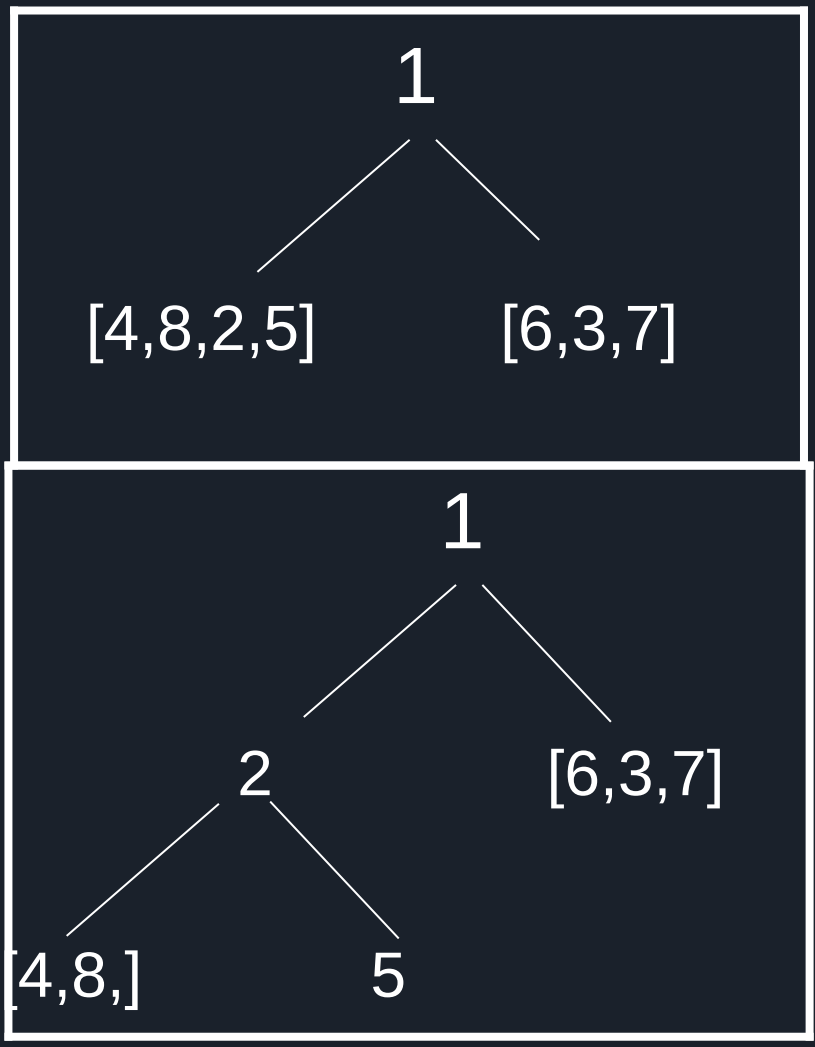
3) We recur the above process for following two.

….**b)** Recur for in[] = {6,3,7} and post[] = {6,7,3}

…….Make the created tree as right child of root.

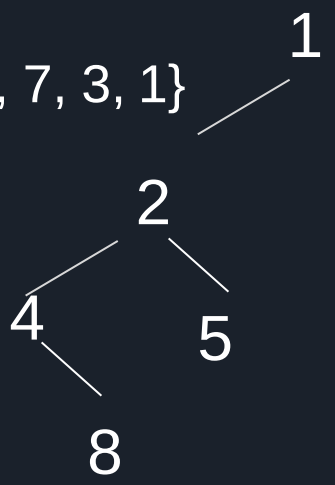
….**a)** Recur for in[] = {4,8,2,5} and post[] = {8,4,5,2}.

…….Make the created tree as left child of root.

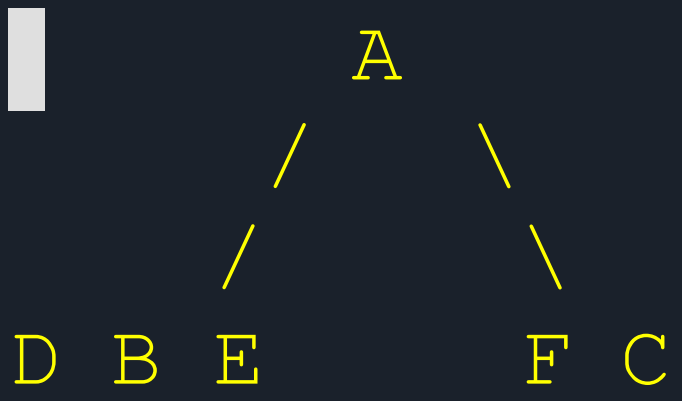


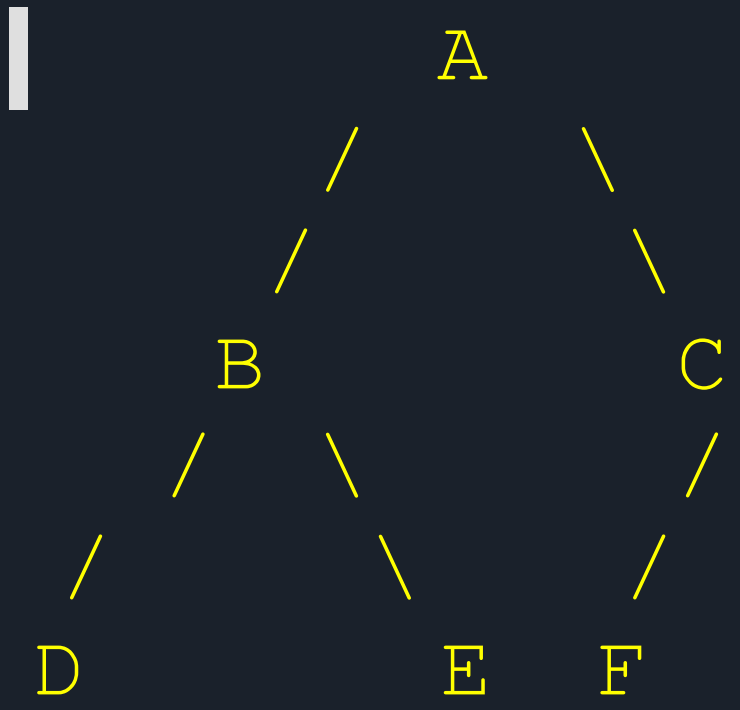
In[] = {4, 8, 2, 5, 1, 6, 3, 7}

post[] = {8, 4, 5, 2, 6, 7, 3, 1}



Inorder sequence: D B E A F C
Preorder sequence: A B D E C F





```
in[] = { 9, 8, 4, 2, 10, 5, 10, 1, 6, 3, 13, 12, 7 };
pre[] ={ 1, 2, 4, 8, 9, 5, 10, 10, 3, 6, 7, 12, 13 };
```

9 8 4 2 10 5 10 1 6 3 13 12 7

```
in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
```

Inorder traversal of the constructed tree is

D B E A F C