CSE 230 Intermediate Programming in C and C++

Binary Tree

Fall 2017

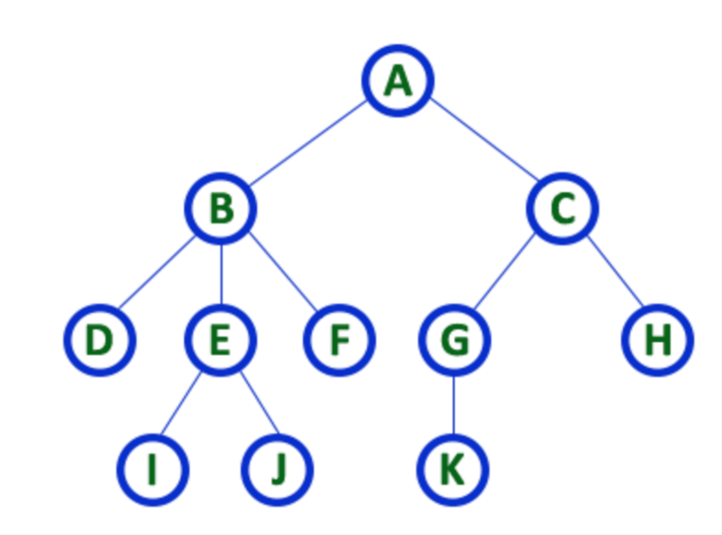
Stony Brook University

Instructor: Shebuti Rayana

shebuti.rayana@stonybrook.edu

Introduction to Tree

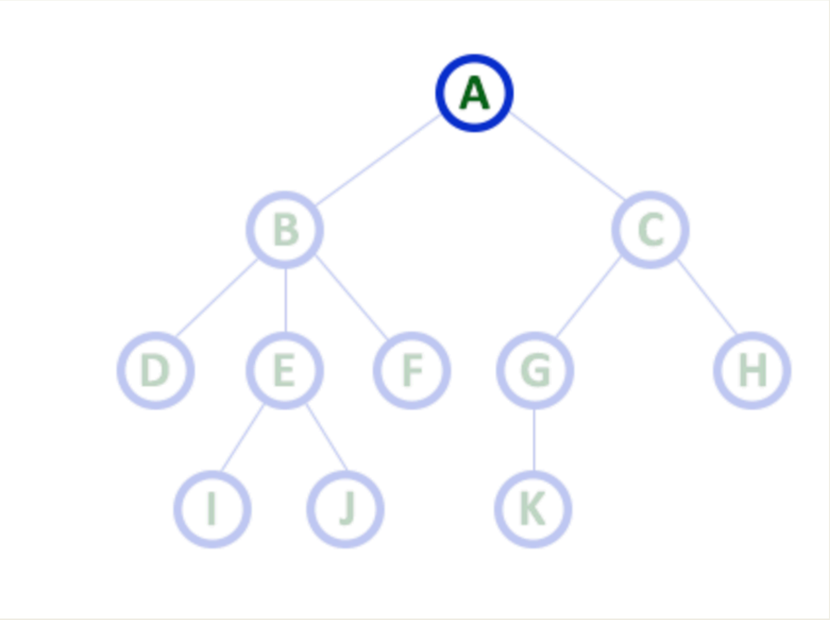
- **Tree is a non-linear data structure which is a collection of data (Node) organized in hierarchical structure.**
- In tree data structure, every individual element is called as **Node**. Node stores
  - *the actual data of that particular element and*
  - *link to next element in hierarchical structure.*



Tree with 11 nodes and 10 edges

Tree Terminology

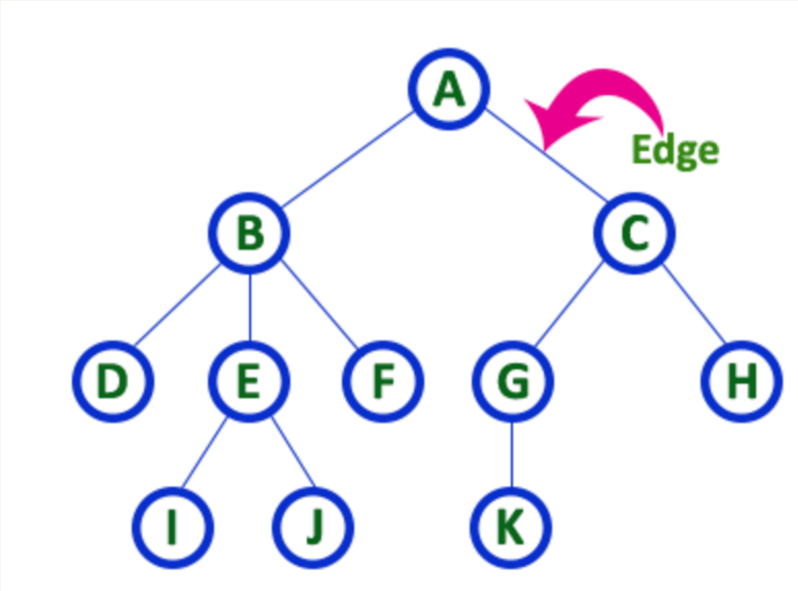
- Root
- In a tree data structure, the first node is called as **Root Node.** Every tree must have root node. In any tree, there must be **only one** root node. Root node does not have any parent. (same as head in a LinkedList).



Here, A is the Root node

Tree Terminology

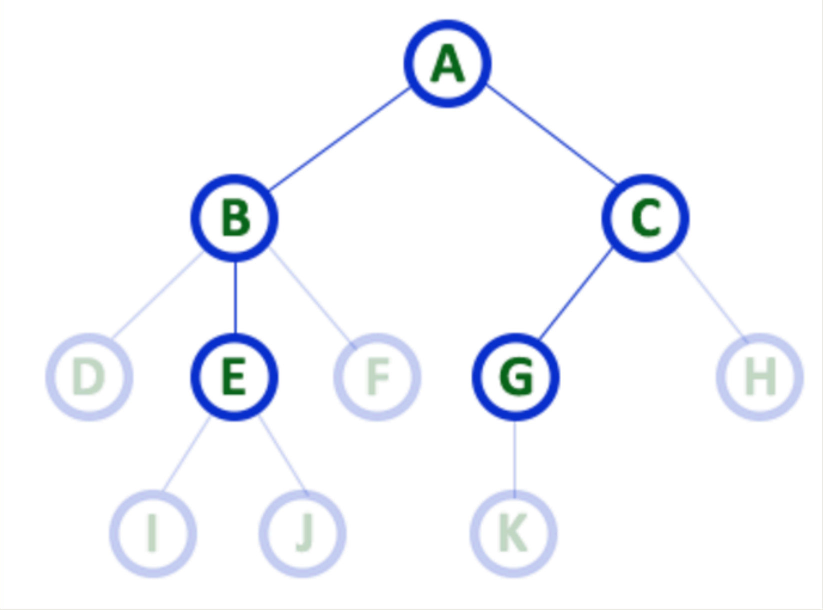
- Edge
- The connecting link between any two nodes is called an **Edge.** In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



Edge is the connecting link between the two nodes

Tree Terminology

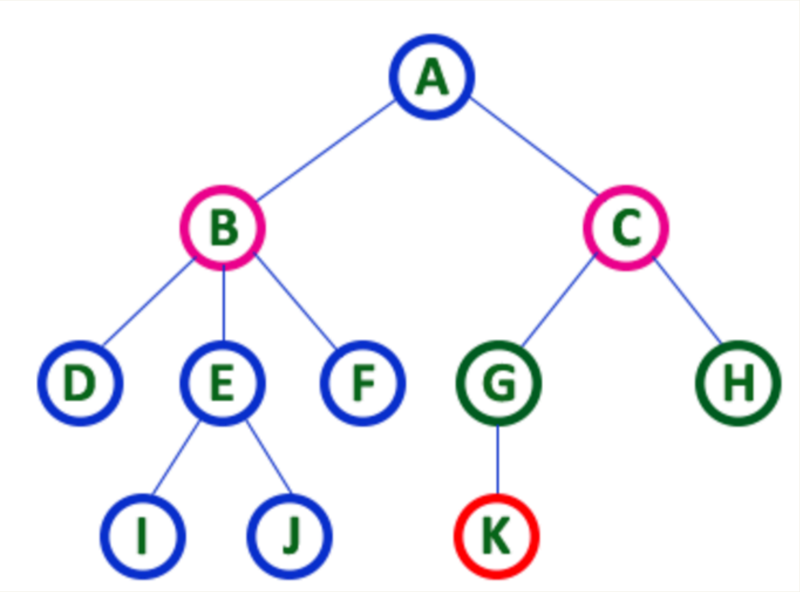
- Parent
- The node which is predecessor of any node is called as **Parent Node.** The node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".



Here,
A is Parent of B and C
B is Parent of D, E and F
C is the Parent of G and H

Tree Terminology

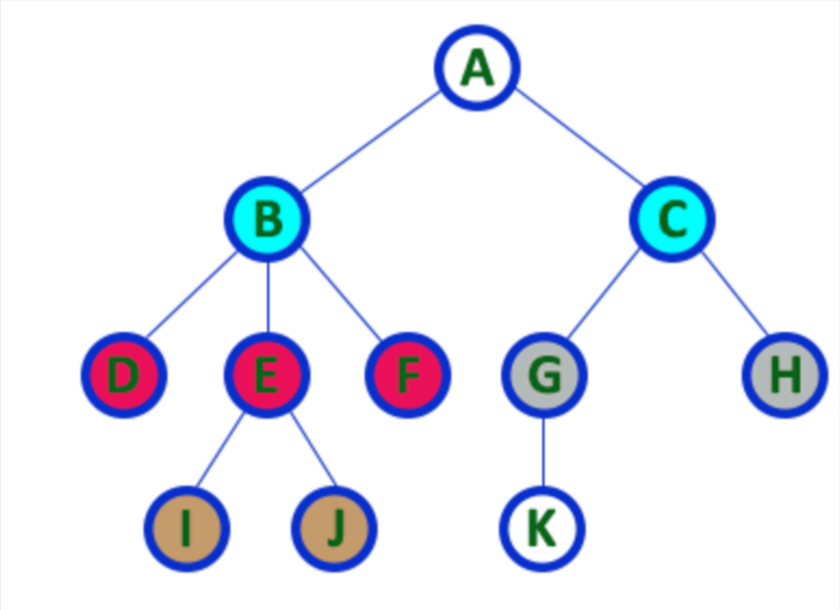
- Child
- The node which is descendant of any node is called as **CHILD Node**. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here,
B and C are Children of A
G and H are Children of C
K is a Child of G

Tree Terminology

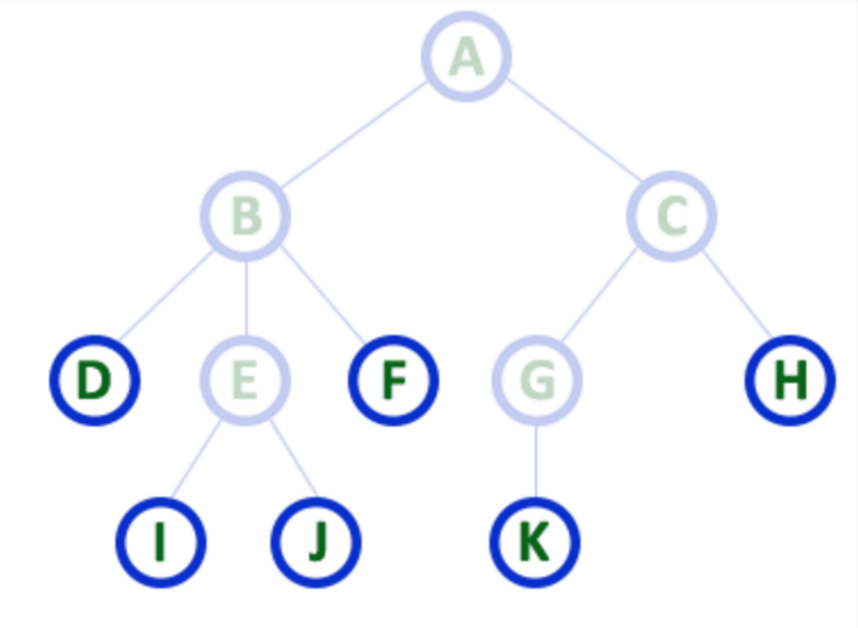
- Siblings
- In a tree data structure, nodes which belong to same Parent are called as **Siblings**. In simple words, the nodes with same parent are called as Sibling nodes.



Here,
B and C are siblings
D, E and F are siblings
G and H are siblings

Tree Terminology

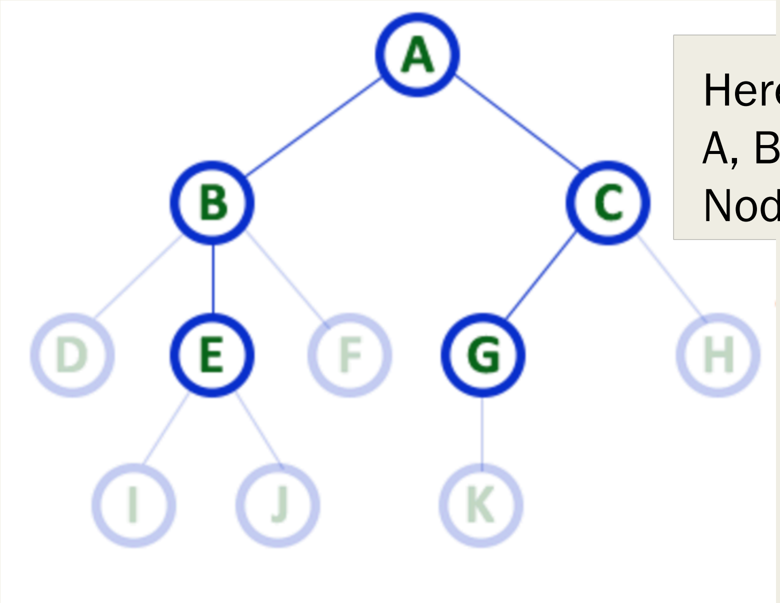
- Leaf
- The node which does not have a child is called as **Leaf Node**.
- leaf node is also called as '**Terminal**' node.



Here,
D, I, J, F. K and H are leaf nodes

Tree Terminology

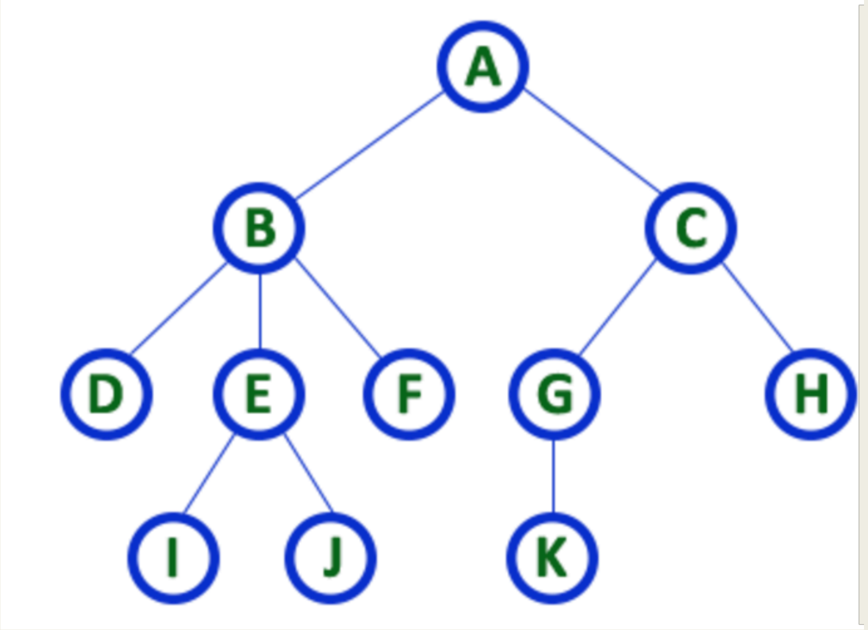
- Internal Nodes
- In a tree data structure, the node which has at least one child is called as **Internal Node.**



Here,
A, B, E, C, G are Internal Nodes

Tree Terminology

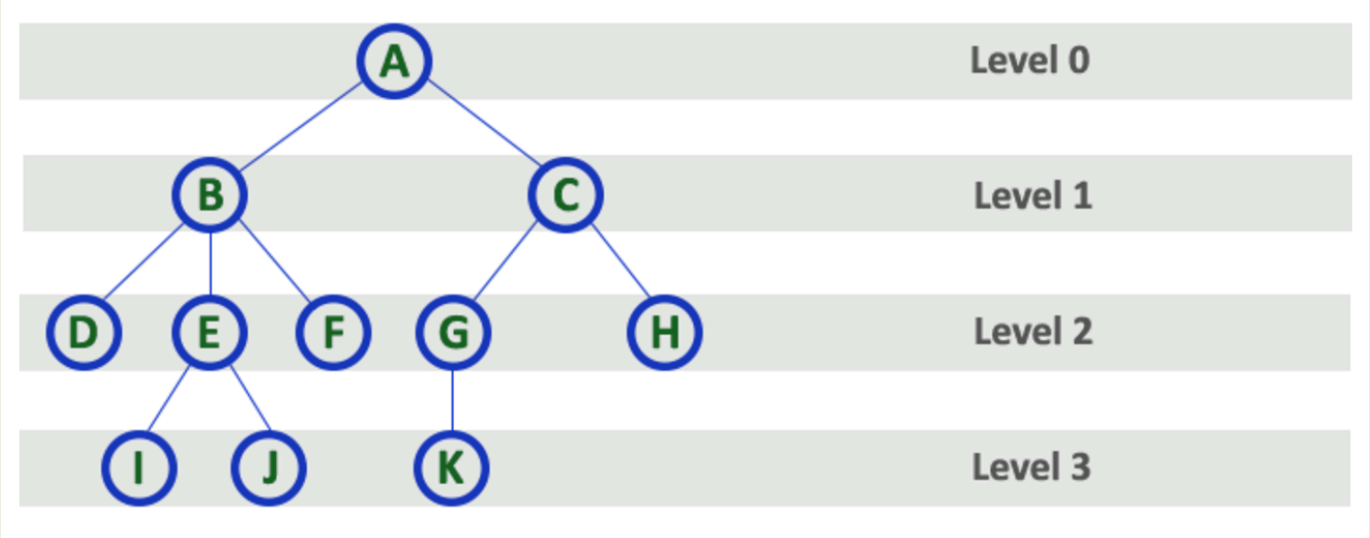
- Degree
- the total number of children of a node is called as **Degree** of that Node. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here,
Degree of A is 2
Degree of B is 3
Degree of F is 0

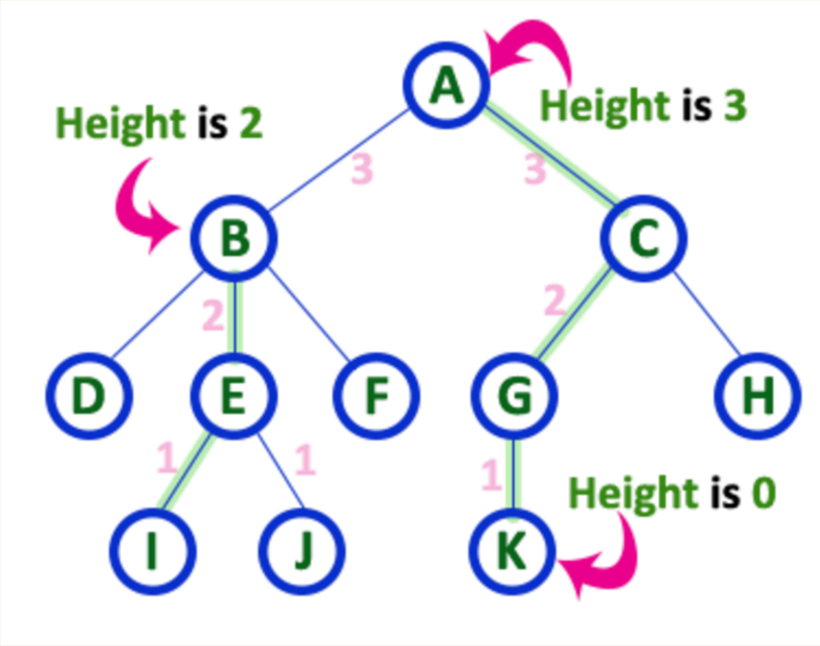
Tree Terminology

- Level
- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on. In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



Tree Terminology

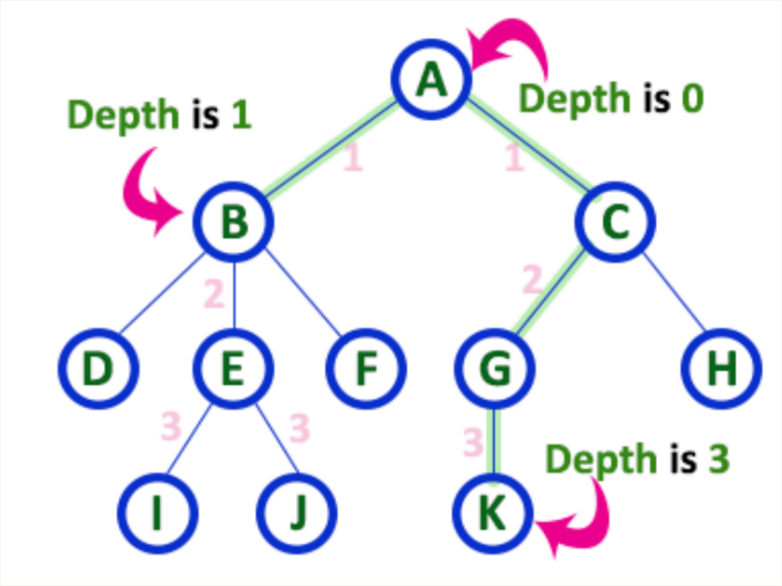
- Height
- the total number of edges from leaf node to a particular node in the longest path is called the Height of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



Here,
Height of the tree is 3

Tree Terminology

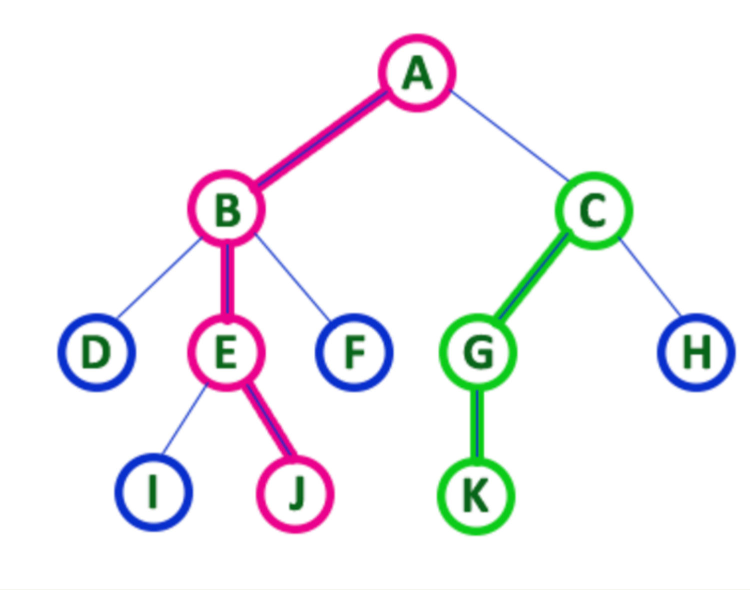
- Depth
- The total number of edges from root node to a particular node is called as **Depth** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree.**



Here,
Depth of the tree is 3

Tree Terminology

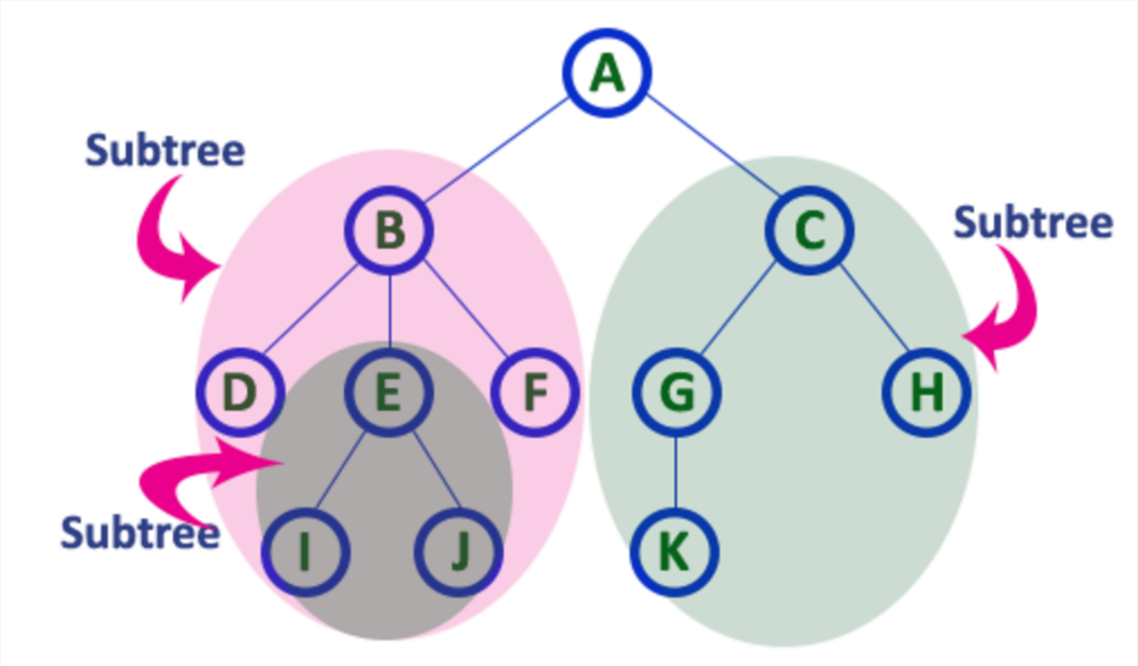
- Path
- The sequence of Nodes and Edges from one node to another node is called a **Path** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4.**



Here,
Path between A and J:
A-B-E-J
Path between C and K:
C-G-K

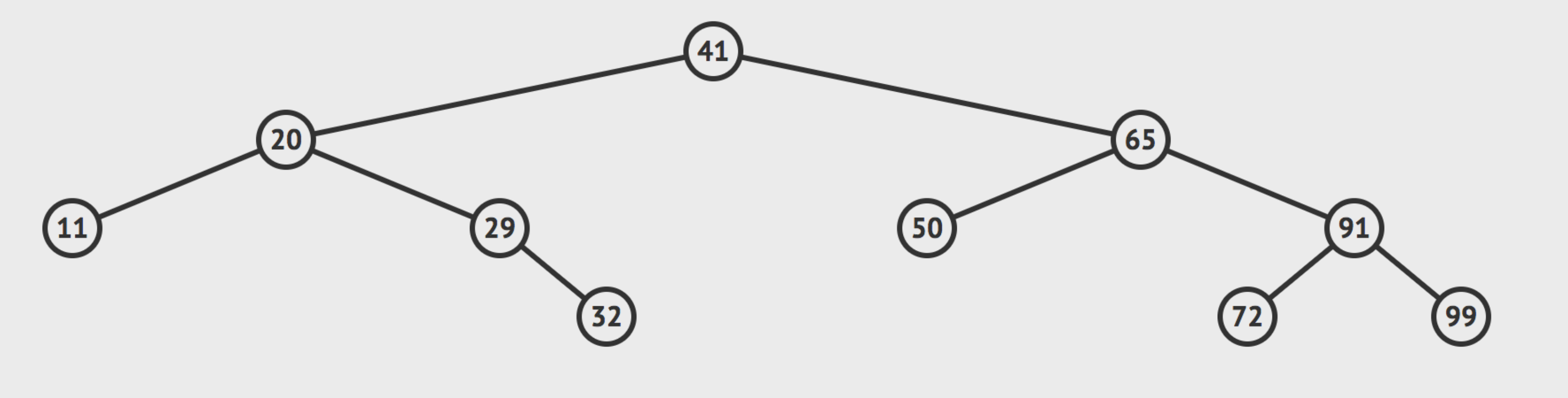
Tree Terminology

- Sub-tree
- Each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



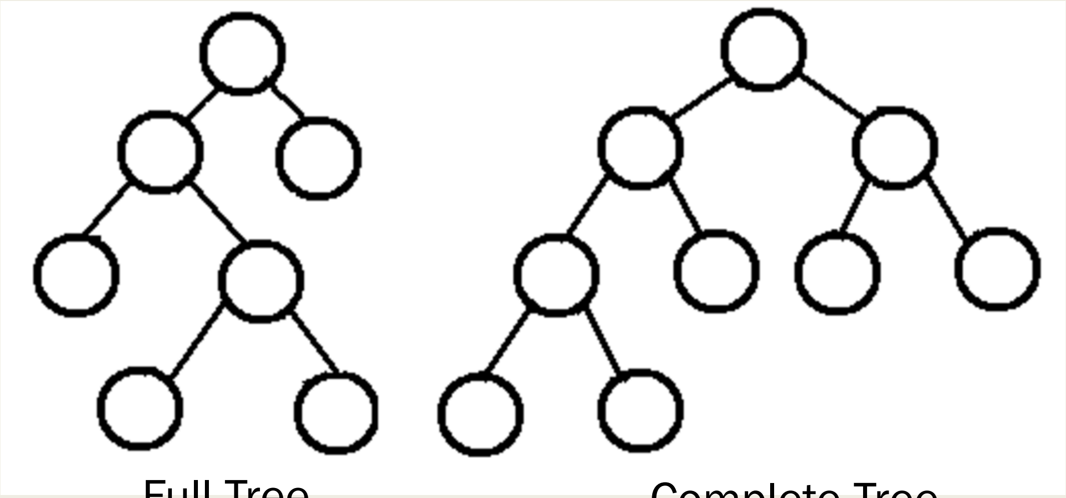
Binary Tree

- Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children.**
  - *One is known as left child and*
  - *the other is known as right child.*
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.



Types of Binary Tree

- A full binary tree is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.



Full Tree Complete Tree

Binary Tree (tree.h)

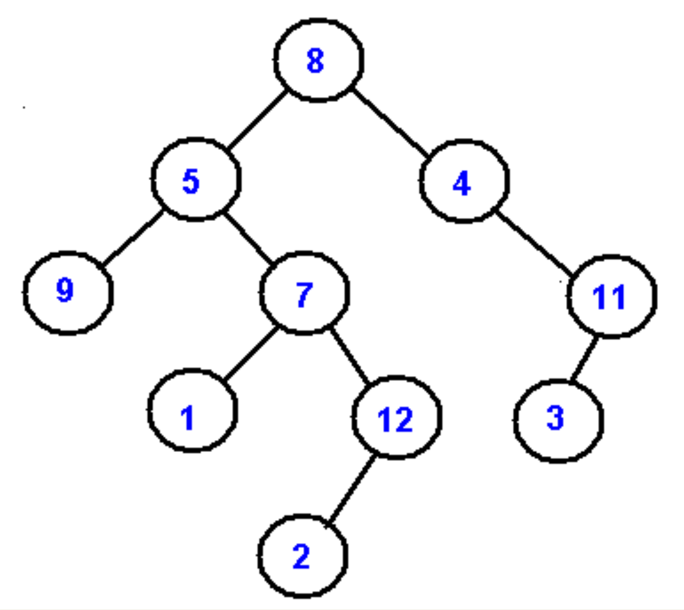
```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

typedef char DATA;

struct node {
        DATA d;
        struct node *left;
        struct node *right;
};

typedef struct node NODE;
typedef NODE *BTREE;
```

Binary Tree Traversal

- A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds
  - *depth-first traversal*
  - *breadth-first traversal*
- There are three different types of depth-first traversals:
  - *Pre Order traversal*
  - *In Order traversal*
  - *Post Order traversal*

Binary Tree Traversal

- Pre Order Traversal:
  - *Visit the root.*
  - *Perform a preorder traversal of the left subtree.*
  - *Perform a preorder traversal of the right subtree.*



Pre Order - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

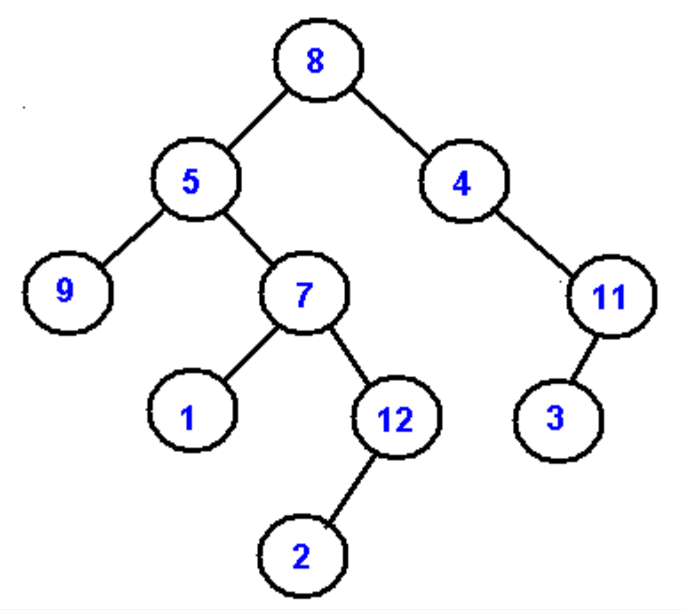
Binary Tree Traversal

- Preorder traversal

```c
/* Preorder binary tree traversal. */
void preorder(BTREE root)
{
        if (root != NULL) {
                printf("%c ", root -> d);
                preorder(root -> left);   /* recur left */
                preorder(root -> right); /* recur right */
        }
}
```

Binary Tree Traversal

- In Order Traversal:
  - *Perform an in order traversal of the left subtree.*
  - *Visit the root.*
  - *Perform an in order traversal of the right subtree.*



In Order - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

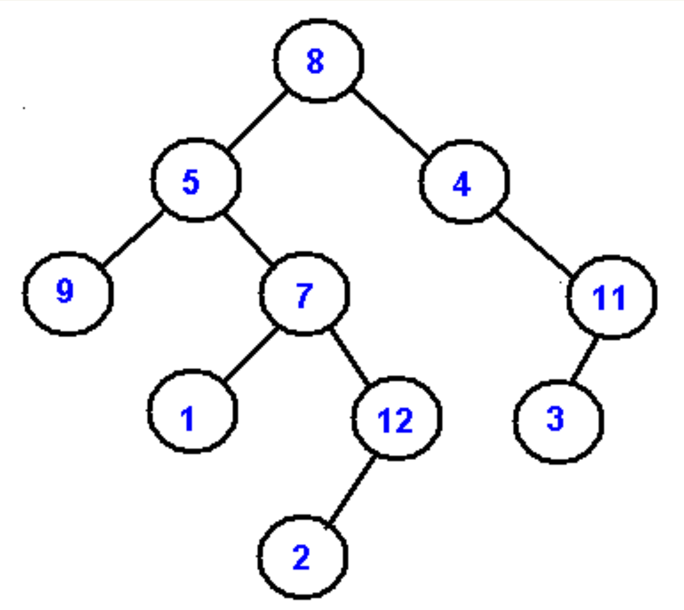
Binary Tree Traversal

- Inorder traversal

```
/* Inorder binary tree traversal. */
void inorder(BTREE root)
{
        if (root != NULL) {
                inorder(root -> left);   /* recur left */
                printf("%c ", root -> d);
                inorder(root -> right); /* recur right */
        }
}
```

Binary Tree Traversal

- Post Order Traversal:
  - *Perform a postorder traversal of the left subtree.*
  - *Perform a postorder traversal of the right subtree.*
  - *Visit the root.*



PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

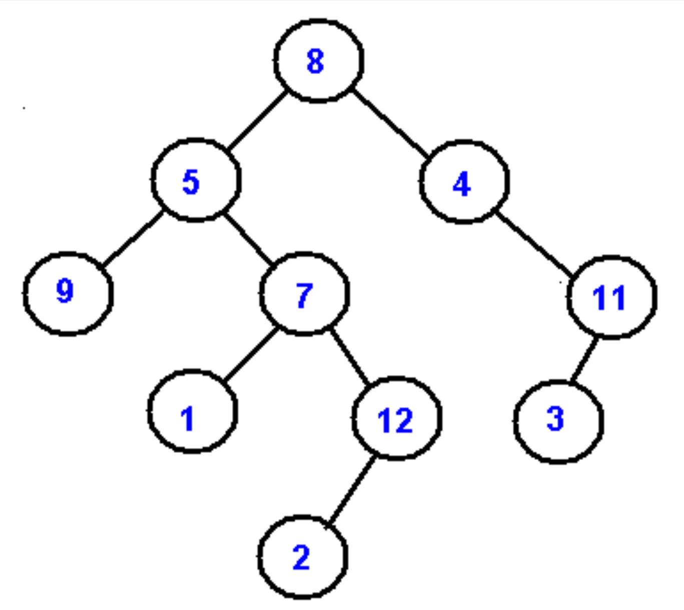
Binary Tree Traversal

- Postorder traversal

```
/* Postorder binary tree traversal. */
void postorder(BTREE root)
{
        if (root != NULL) {
                postorder(root -> left);   /* recur left */
                postorder(root -> right); /* recur right */
                printf("%c ", root -> d);
        }
}
```

Binary Tree Traversal

- There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.



LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Binary Tree Traversal

- Level order traversal

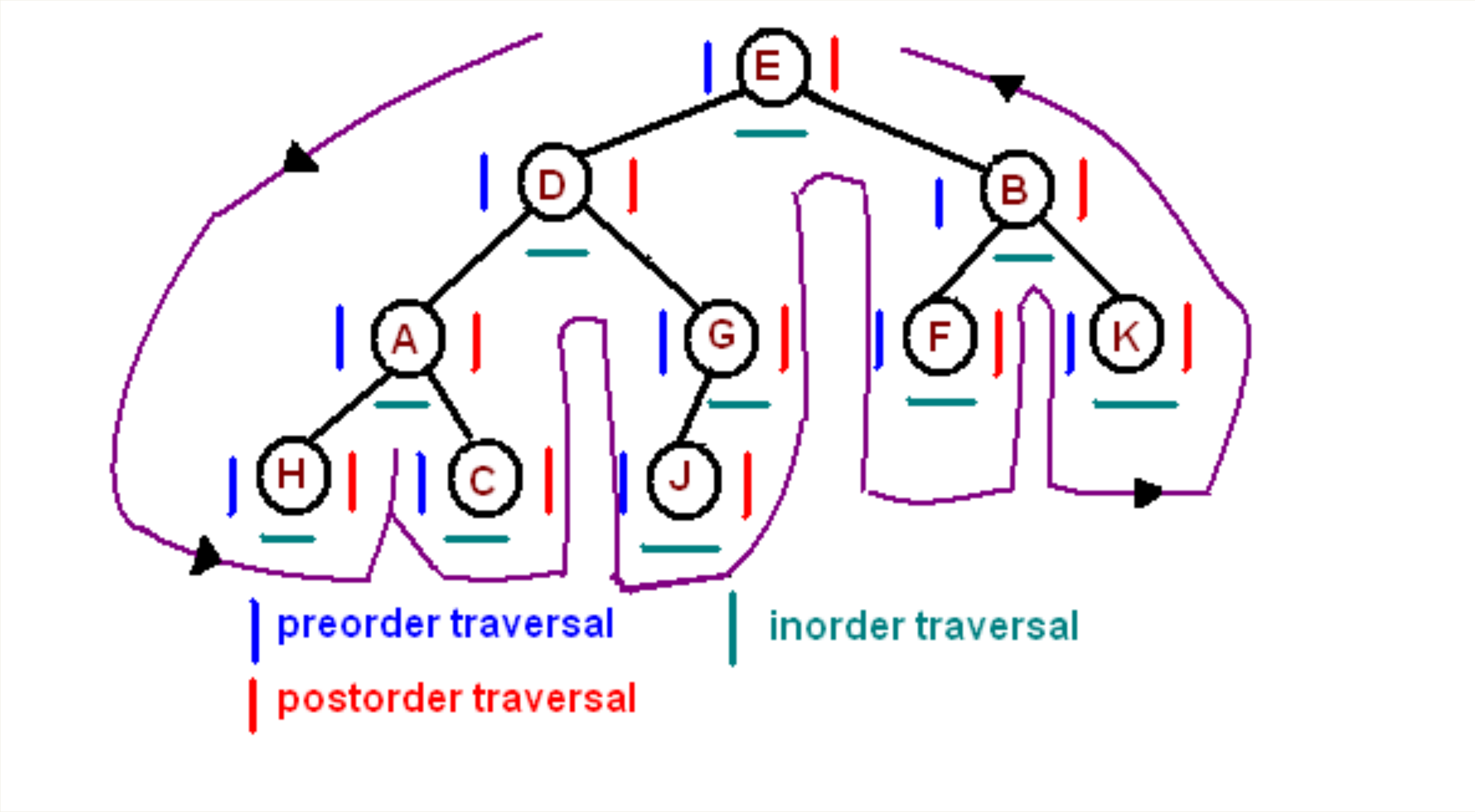
```c
/* Print nodes using level order traversal */
void printLevelOrder(BTREE root)
{
    int h = height(root);
    int i;
    for (i=0; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(BTREE root, int level)
{
    if (root == NULL)
        return;
    if (level == 0)
        printf("%c ", root->d);
    else if (level > 0)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}
```

```c
/* Compute the "height" of a tree */
int height(BTREE node)
{
    if (node==NULL)
        return -1;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}
```

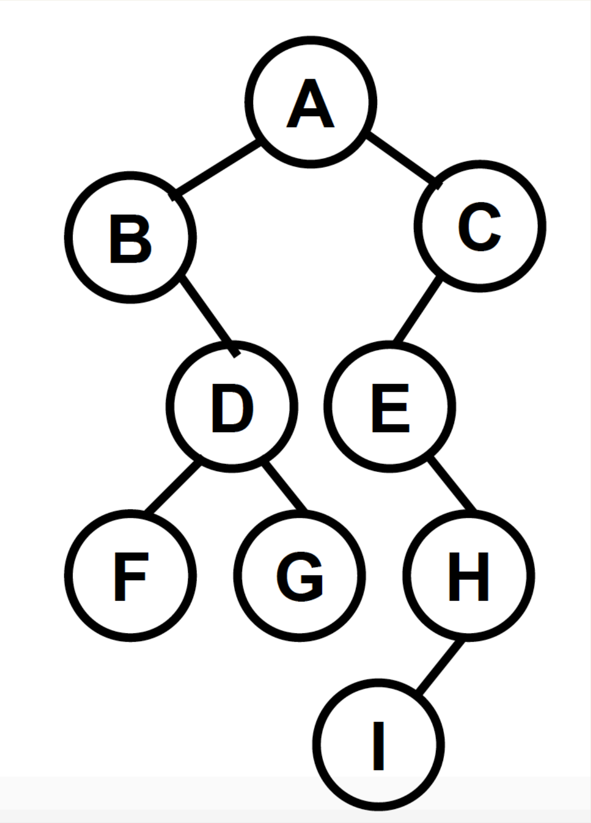
Binary Tree Traversal

- These common traversals can be represented as a single algorithm by assuming that we visit each node three times.
- An *Euler tour* is a walk around the binary tree where each edge is treated as a wall, which you cannot cross. In this walk each node will be visited either on the left, or from the below, or on the right.
- The Euler tour in which
  - *When we visit nodes on the left produces a preorder traversal*
  - *When we visit nodes from the below, we get an inorder traversal. And*
  - *when we visit nodes on the right, we get a postorder traversal.*

Binary Tree Traversal



Binary Tree Traversal: Example

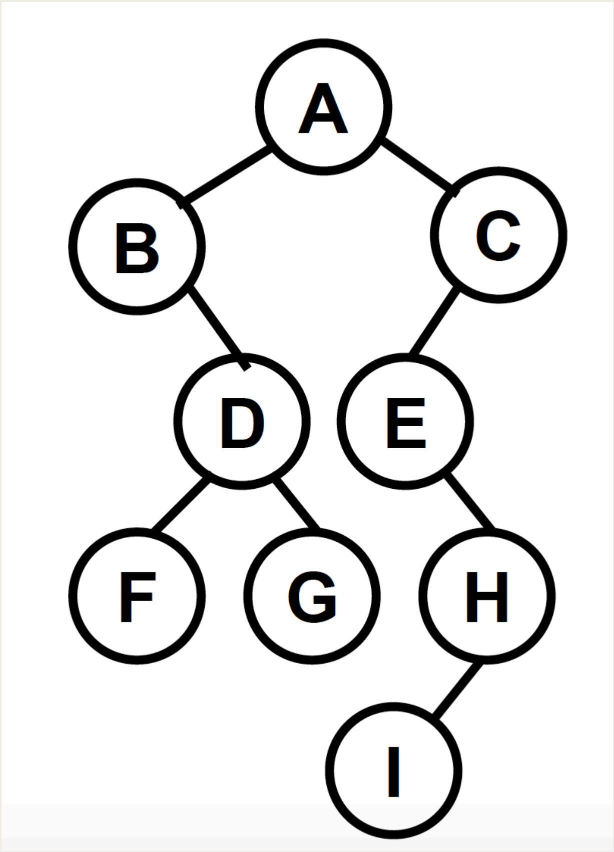


Pre Order

In Order

Post Order

Binary Tree Traversal: Example



Pre Order
ABDFGCEHI

In Order
BFDGAEIHC

Post Order
FGDBIHECA

Creating Binary Trees

```
/* Creating a binary tree. */
BTREE new_node(void){
        return (malloc(sizeof(NODE)));
}

BTREE init_node(DATA dl, BTREE pl, BTREE p2)
{
        BTREE t;
        t = new_node() ;
        t -> d = dl;
        t -> left = p1;
        t -> right = p2;
        return t;
}
```

Creating Binary Trees

- We will use these routines as primitives to create a binary tree from data values in an array.
  - *There is a very nice mapping from the indices of a linear array into nodes of a binary tree. We do this by taking the value a[i] and letting it have as child a[2*i+1] and a[2*i+2].*
  - *Then we map a[0] into the unique root node of the resulting binary tree. Its left child will be a [1], and its right child will be a [2].*
  - *The function create_tree() embodies this mapping.*
  - *The formal parameter size is the number of nodes in the binary tree.*

Creating Binary Trees

```
/* Create a linked binary tree from an array. */
BTREE create_tree(DATA a[], int i, int size)
{
        if (i >= size)
                return NULL;
        else
                return (init_node(a[i],
        create_tree(a, 2 * i + 1, size),
        create_tree(a, 2 * i + 2, size)));
}
```