DATA STRUCTURES

(this is an additional material and not exhaustive study material)

Trees: Introduction to Trees – Basic concepts - Binary Trees – Binary tree representations (Array and list) and Traversals Techniques (Preorder, Inorder, Postorder) – Succinct Data Structures: Overview – Level order representation of Binary Trees – Rank and Select – Subtrees.

Tree Definition

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

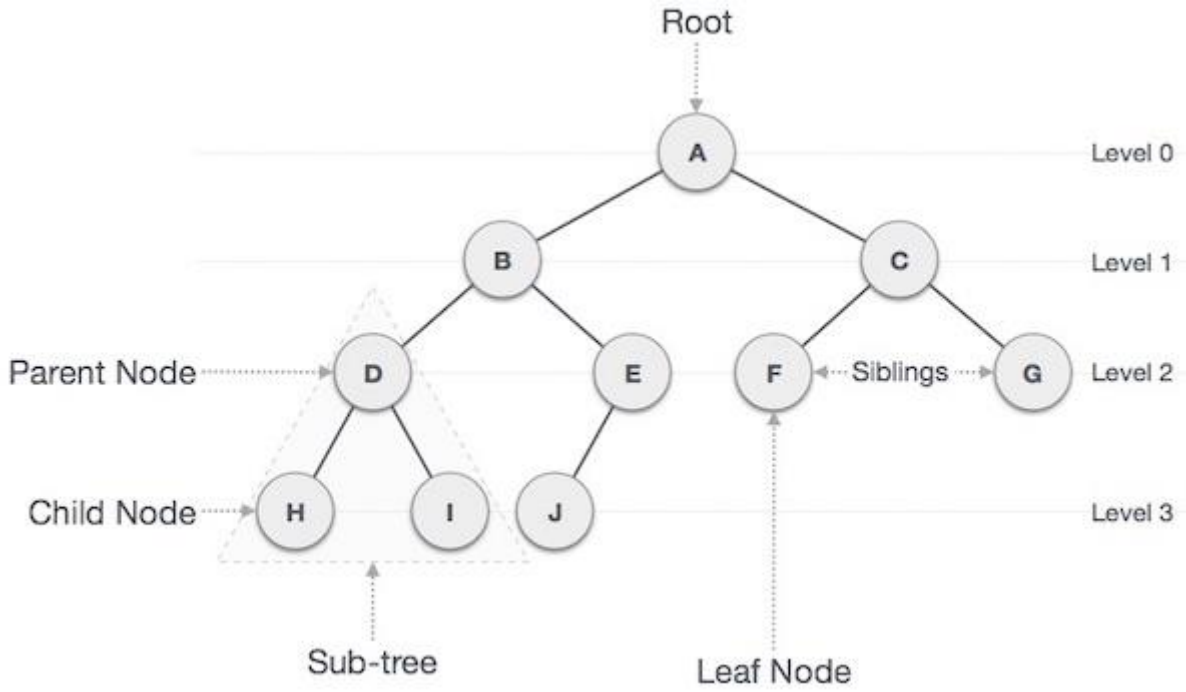
- if T is not empty, T has a special tree called the root that has no parent
- each node v of T different than the root has a unique parent node w; each node with parent w is a child of w
- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- 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.

Applications of trees

- class hierarchy in Java
- file system
- storing hierarchies in organizations
- For explaining Router algorithms
- To represent networks
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Terminologies

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.
- If node has no children, it is called **Leaves** or **External Nodes.**
- Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.
- A tree can be empty with no nodes or a tree consists of one node called the **Root**.



- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Advantages of Tree

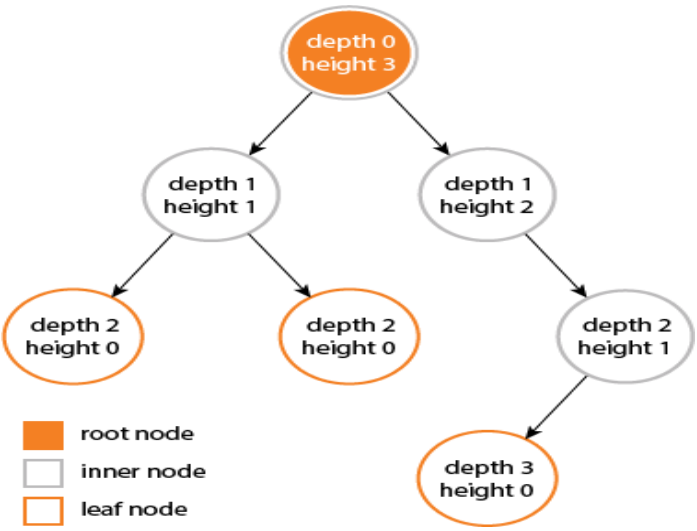
- Tree reflects structural relationships in the data.
- It is used to represent hierarchies.
- It provides an efficient insertion and searching operations.
- Trees are flexible. It allows to move subtrees around with minimum effort.

Height and Depth

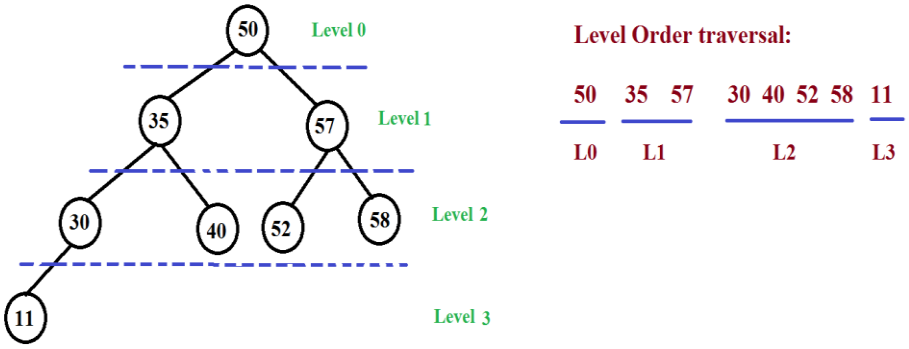
- The **depth** of a node is the number of edges from the node to the tree's root node.
  A root node will have a depth of 0.
- The **height** of a node is the number of edges on the *longest path* from the node to a leaf.
  A leaf node will have a height of 0.

Properties of a *tree*:

- The **height** of a tree would be the height of its root node,
  or equivalently, the depth of its deepest node.
- The **diameter** (or **width**) of a tree is the number of *nodes* on the longest path between any two leaf nodes. The tree below has a diameter of 6 nodes.



Level – The level of a node is defined by 1 + the number of connections between the node and the root.



It **starts from 1** and **the level of the root is 1**.

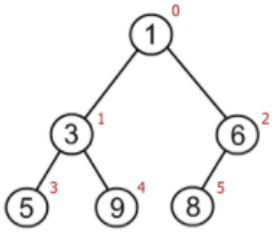
Tree representation (Two Ways)

Array representation

Linked representation

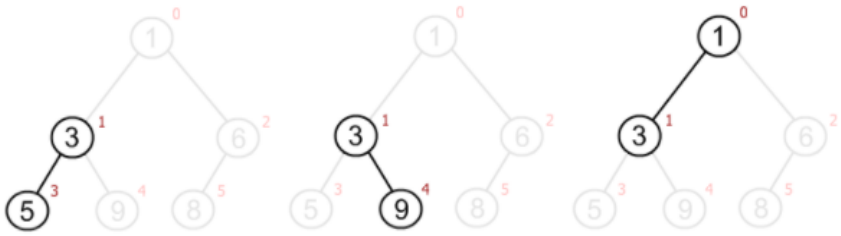
Array Representation

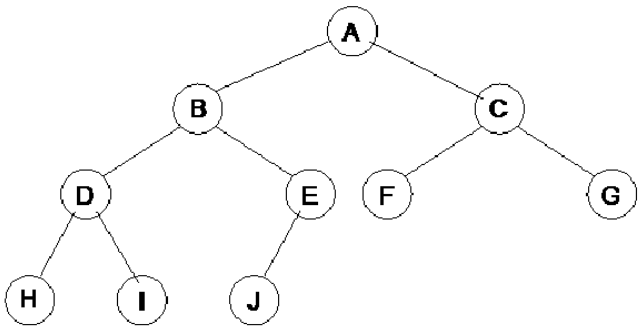
- Root node is at location 0. **(if stored at location 1, the left child at 2*I and right at 2*i+1)**
- If a node is at I, its left child at 2*i+1 and right child at 2*1+2



Such mapping answers to formulas below:

```
Left(i) = 2 * i + 1      Right(i) = 2 * i + 2      Parent(i) = (i - 1) / 2
```





	A	B	C	D	E	F	G	H	I	J				
	1	2	3	4	5	6	7	8	9	10	11	12	13	1

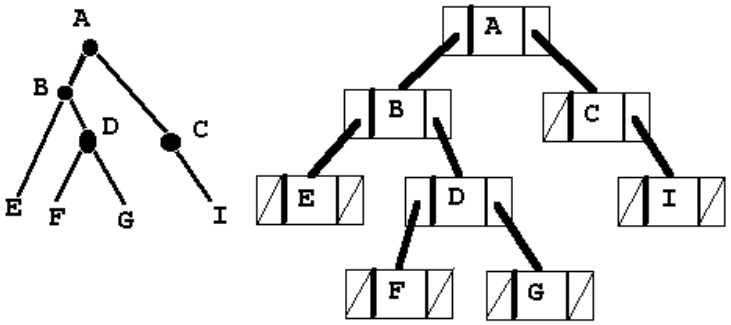
Linked representation

Consider a Binary Tree **T**. **T** will be maintained in memory by means of a linked list representation which uses three parallel arrays; **INFO**, **LEFT**, and **RIGHT** pointer variable **ROOT** as follows. In Binary Tree each node **N** of **T** will correspond to a location **k** such that

1. **LEFT [k]** contains the location of the left child of node **N**.
2. **INFO [k]** contains the data at the node **N**.
3. **RIGHT [k]** contains the location of right child of node **N**.

Representation of a node:

LEFT [k]	INFO [k]	RIGHT [k]



Binary tree:

A binary tree is a non-linear data structure in which each node has maximum of two child nodes.

Complete Binary Tree

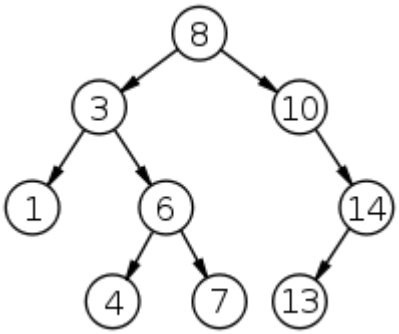
A binary tree **T** is said to be complete binary tree if -

1. All its levels, except possibly except possibly the last, have the maximum number of nodes and
2. All the nodes at the last level appears as far left as possible.

Full Binary Tree

A Binary Tree is full binary tree if and only if -

1. Each non- leaf node has exactly two child nodes.
2. All leaf nodes are at the same level.



The search time is minimal. Best case is O(log N) as in binary search (when the tree is balanced)

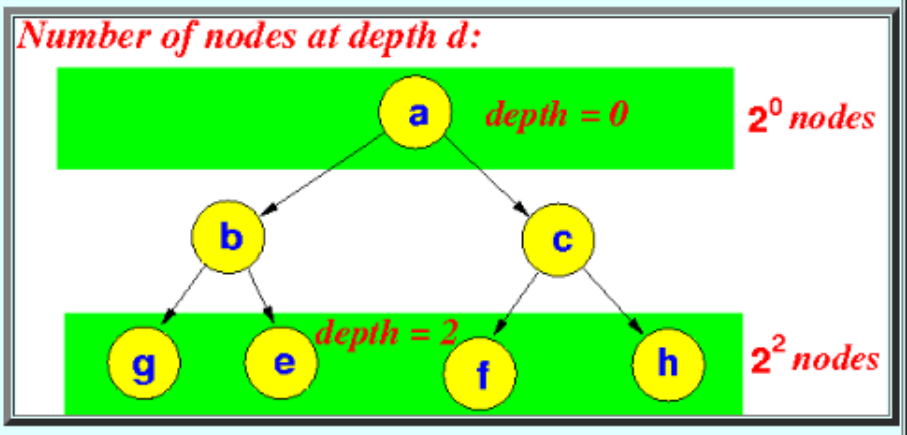
Theorem 1

- The **number of nodes** at **depth** *d* in a ***perfect*** **binary tree** = 2^d

- There is *only* **1 node** (= the **root node**) at **depth** *0*:

  $2^0 = 1$

- In a **perfect binary tree**, **every node** has **2 children nodes**



So:

Depth d	# nodes at depth d	# of child nodes
0	1 = 2^0	2 (each node has 2 children)
1	2 = 2^1	4 (each node has 2 children)
2	4 = 2^0	8 (each node has 2 children)
...		

I.e.:

- The **number** of nodes **doubles** every time the **depth** increases by **1** !

Therefore:

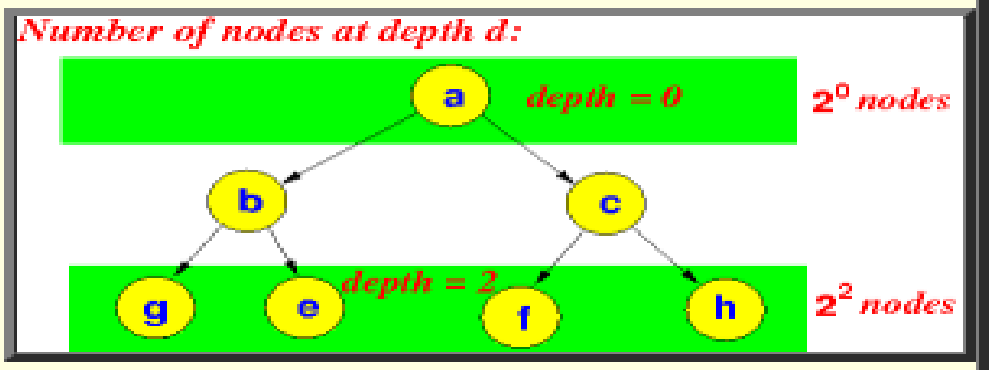
nodes at depth d = 2^d

Theorem 2

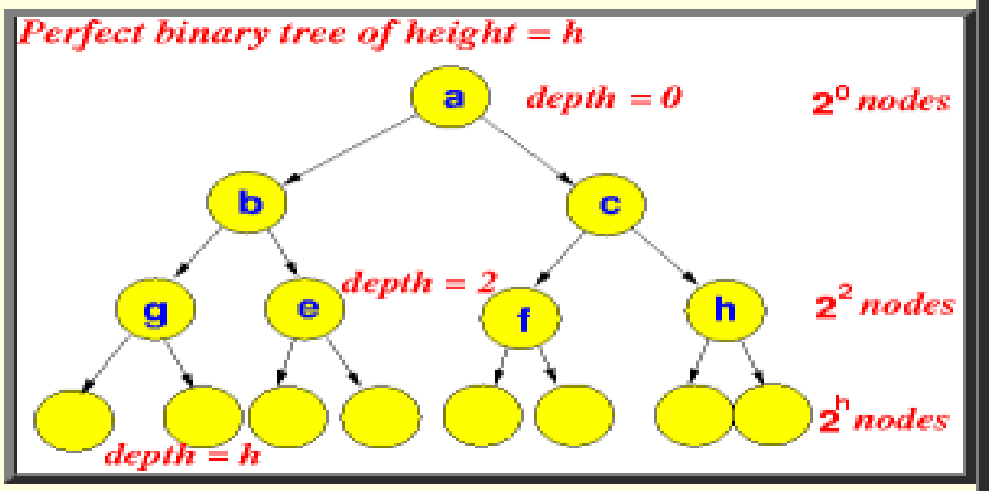
- A ***perfect*** **binary tree** of **height** *h* has:
  - $2^{h+1} - 1$ nodes

- **Previously**, we have shown that:
  - # nodes at *depth* d = 2^d

See:



- So the **total number** of **nodes** in a **perfect binary tree** of **height** *h*:



nodes = $2^0 + 2^1 + \_\_ 2^h = 2^{h+1} - 1$

- **Proof:**

$$S = 1 + 2 + 2^2 + 2^3 + \ldots + 2^h$$
$$2xS = \quad 2 + 2^2 + 2^3 + \ldots + 2^h + 2^{h+1} \quad \text{- (subtract)}$$
$$2xS - S = 2^{h+1} - 1$$
$$\Longleftrightarrow \quad S = 2^{h+1} - 1$$

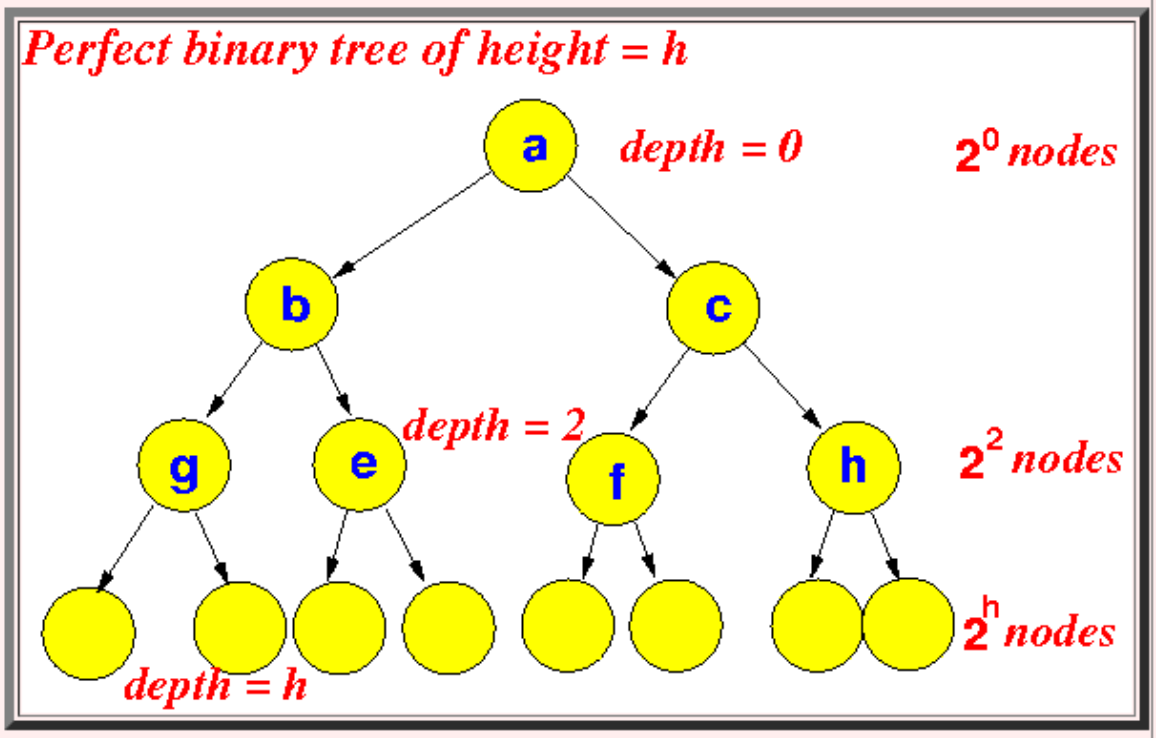
Theorem 3

- **Number** of *leaf* **nodes** in a **perfect binary tree** of **height** $h = 2^h$

- # nodes at **depth** *d* in a **perfect binary tree** = 2^d

---

- **All** the ***leaf* nodes** in a **perfect binary tree** of **height** *h* has a depth equal to *h*:



- # nodes at **depth** *h* in a **perfect binary tree** = 2^h

Therefore:

- **Number** of ***leaf* nodes** in a **perfect binary tree** of **height** *h* = 2^h

Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Delete** – To delete a node in a tree
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
  int data;
```

```
  struct node *left;
  struct node *right;
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data){

    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->left;
      }//else go to right tree
      else {
        current = current->right;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }
  return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm – Non Recursive

```
// to insert a new node with the data value key
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct   tempNode->data = key;
  tempNode->left = NULL;
```

```
  tempNode->right = NULL;

  //if tree is empty
  if(root == NULL) {
                root = tempNode; }
  else {
       current = root;
       parent = NULL;
       while(1) {
                parent = current;

                //go to left of the tree
                if(key < parent->data)
                  {
                   current = current->left;
                   //insert to the left
                              if(current == NULL)
                         {  parent->left = tempNode;
                            return;    }
                   }
              //go to right of the tree
              else {
                    current = current->right;
                    //insert to the right
                    if(current == NULL)
                         { parent->right = tempNode;
                           return;      }
                    }
             }   // while loop ends
  }   // else ends
```

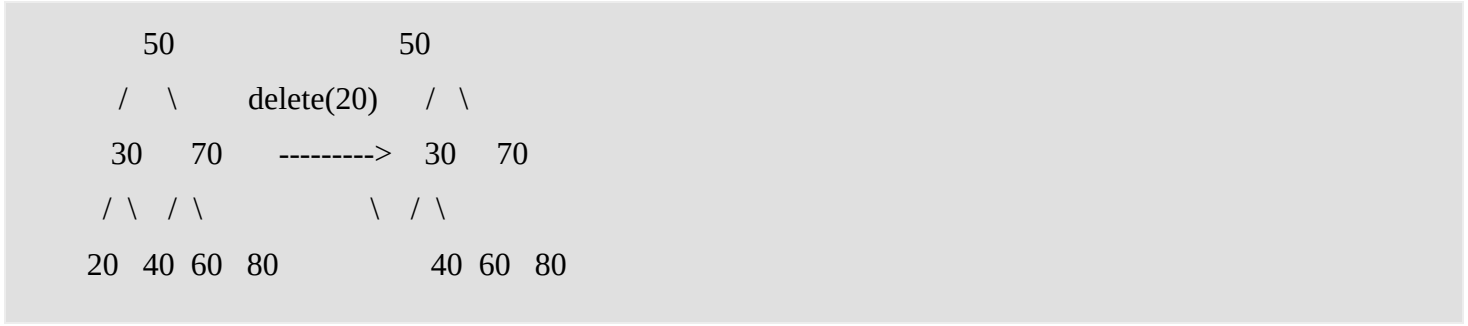
Algorithm – Recursive

```
struct node* insert(struct node* node, int key)
{
   /* If the tree is empty, return a new node */
   if (node == NULL) return newNode(key);

   /* Otherwise, recur down the tree */
   if (key < node->key)
      node->left  = insert(node->left, key);
   else
      node->right = insert(node->right, key);

   /* return the (unchanged) node pointer */
   return node;
}
```

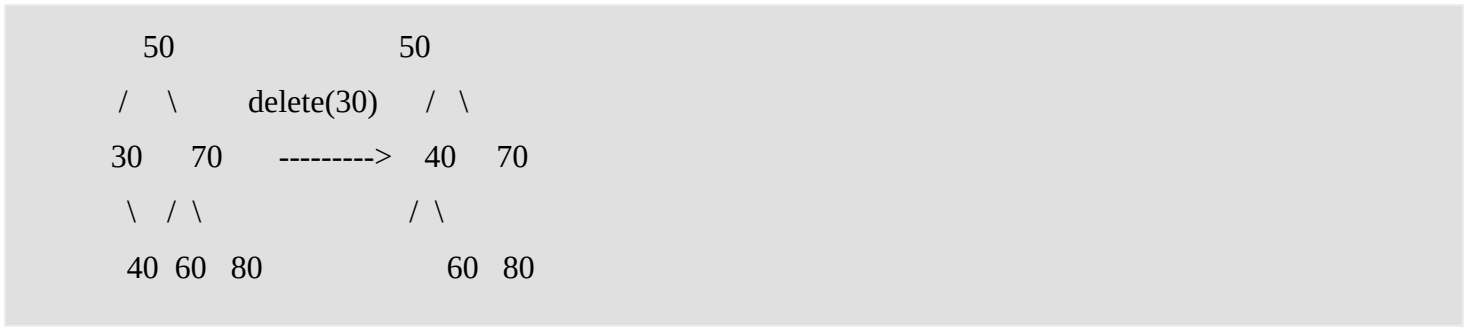
Binary Search Tree – Deletion

When we delete a node, there possibilities arise.

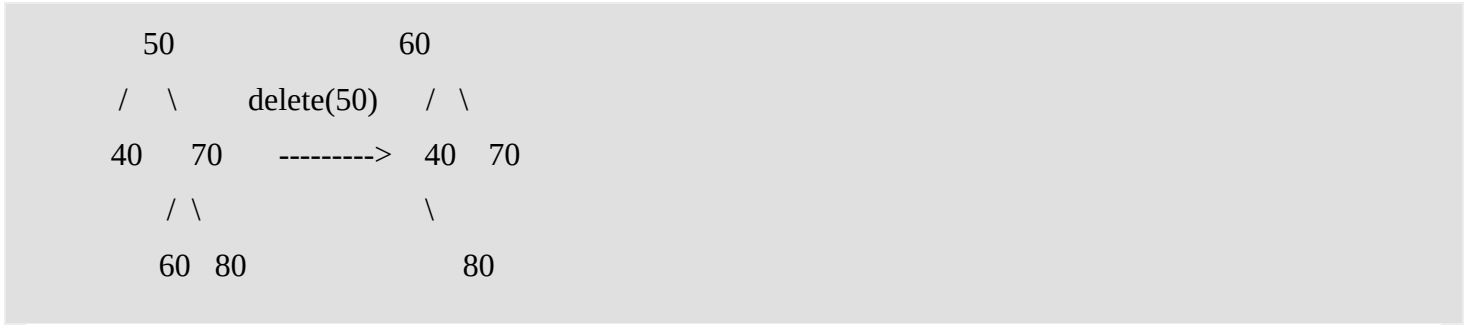
1) ***Node to be deleted is leaf:*** Simply remove from the tree.



2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



ONE STEP LEFT DEEP DOWN RIGHT or ONE STEP RIGHT DEEP DOWN LEFT

```
/* Given a binary search tree and a key, this function deletes the key
  and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
```

```
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

Tree Traversals

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

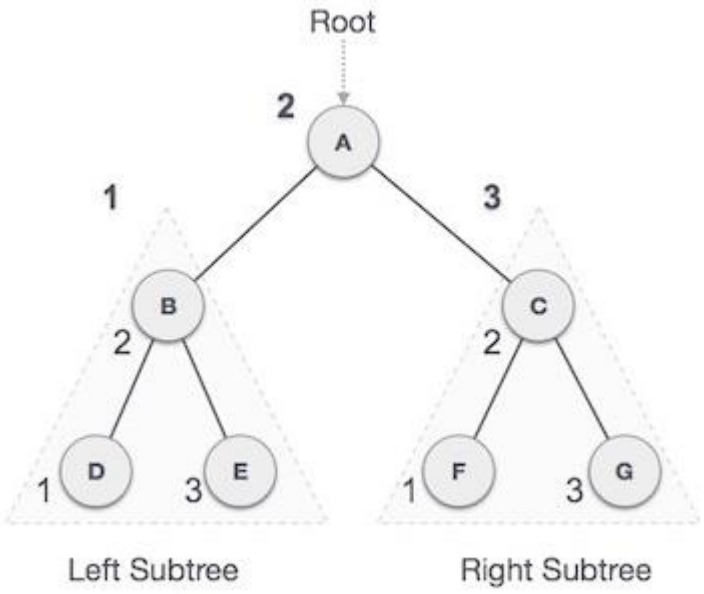
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal (L Rt R)

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

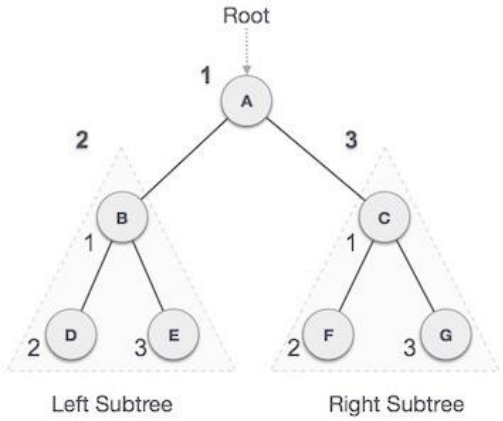
Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

```
// A function to do inorder traversal of BST
void inorder(struct node *root)
{
   if (root != NULL)
   {
      inorder(root->left);
      printf("%d ", root->key);
      inorder(root->right);
   }
}
```

Pre-order Traversal (Rt L R)

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

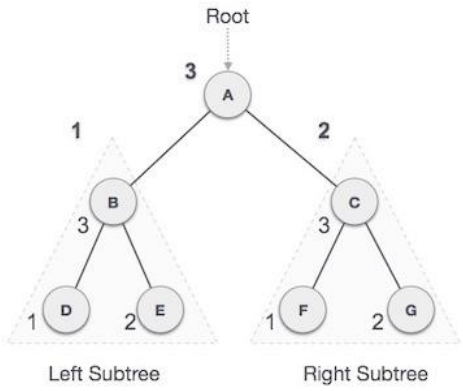
Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

```
// A function to do preorder traversal of BST
void inorder(struct node *root)
{
   if (root != NULL)
   {
      printf("%d ", root->key);
      inorder(root->left);
      inorder(root->right);
   }
}
```

Post-order Traversal (L R Rt)

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

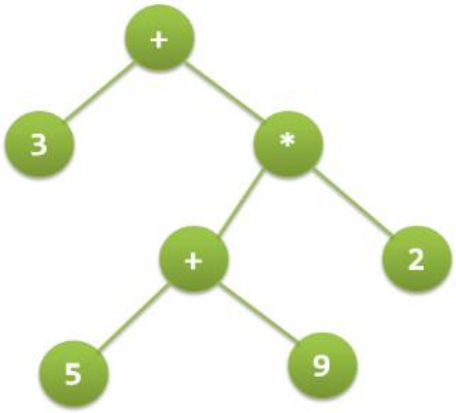
Step 2 − Recursively traverse right subtree.

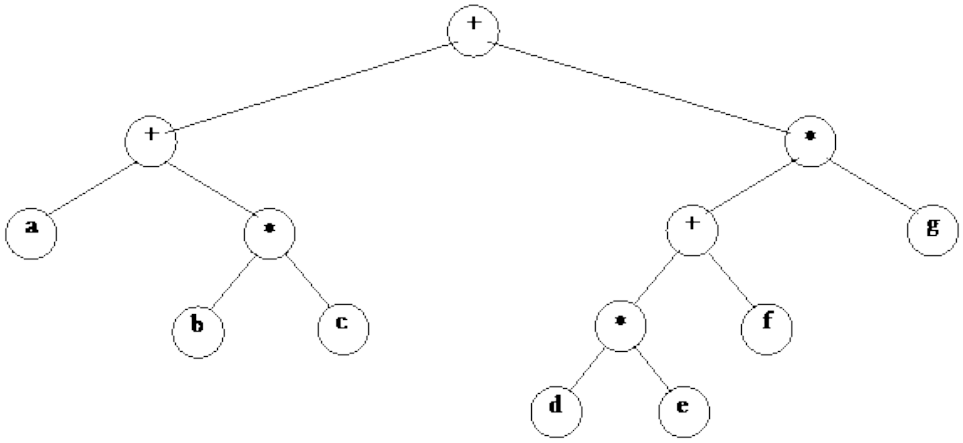
Step 3 − Visit root node.

```
// A function to do postorder traversal of BST
void inorder(struct node *root)
{
   if (root != NULL)
   {
     inorder(root->left);
     inorder(root->right);
     printf("%d ", root->key);
   }
}
```

Expression Trees

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:

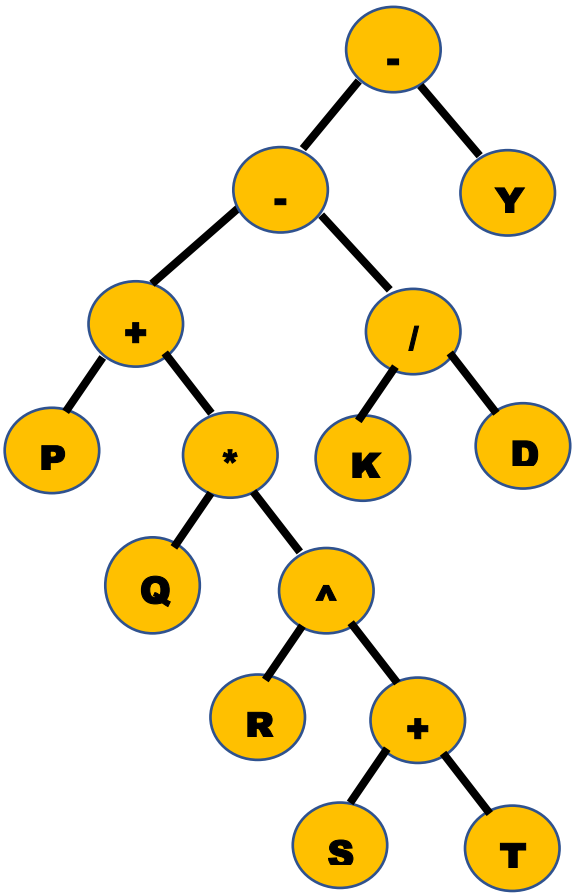




Inorder: *(a + (b * c)) + (((d * e) + f) * g)*

Traversals in an Expression Tree :

P+Q*R^(S+T)-K/D-Y



Inorder : P + Q * R ^ S + T – K / D - Y

Postorder : P Q R S T + ^ * + K D + - Y -

Preorder : - - + P * Q ^ R + S T / K D Y

Succinct Data Structures

In Many Computations storage costs of Pointers and other structures dominate that of Real Data. Succinct Data Structure is a space efficient data structure

The objective is to

- Encode a combinatorial object (e.g. a tree) of specialized information
- ain a small amount of space
- and still perform queries in constant time

Representation of a combinatorial object:

- Space requirement of representation "close to" information theoretic lower bound
- Time for operations required of the data type comparable to that of representation without such space constraints (O(1))

Applications of Succinct Data Structures

There exist a range of real-world applications that can utilize this space-efficient storage method, most centered around information retrieval:

Search Engines- Search engines can index billions of web pages and respond to queries about those pages in real-time. Therefore, it is crucial to decrease the space used by their index while still allowing efficient queries.

Mobile applications - Mobile applications have a limited amount of storage available on the device and efficient storage allows for added functionality.

DNA representation - Medical databases are massive and contain sequences that contain patterns and need to be queried quickly. One can successfully represent DNA in a succinct manner.

Streaming environments - In a streaming environment, the next frame(s) need to be accessed quickly and efficiently. The smaller the size of the content, the faster it can be processed, and at a smaller cost.