E- Lecture
Data Structures and Algorithms

By
H R Choudhary (Asstt. Professor)
Department of CSE
Engineering College Ajmer

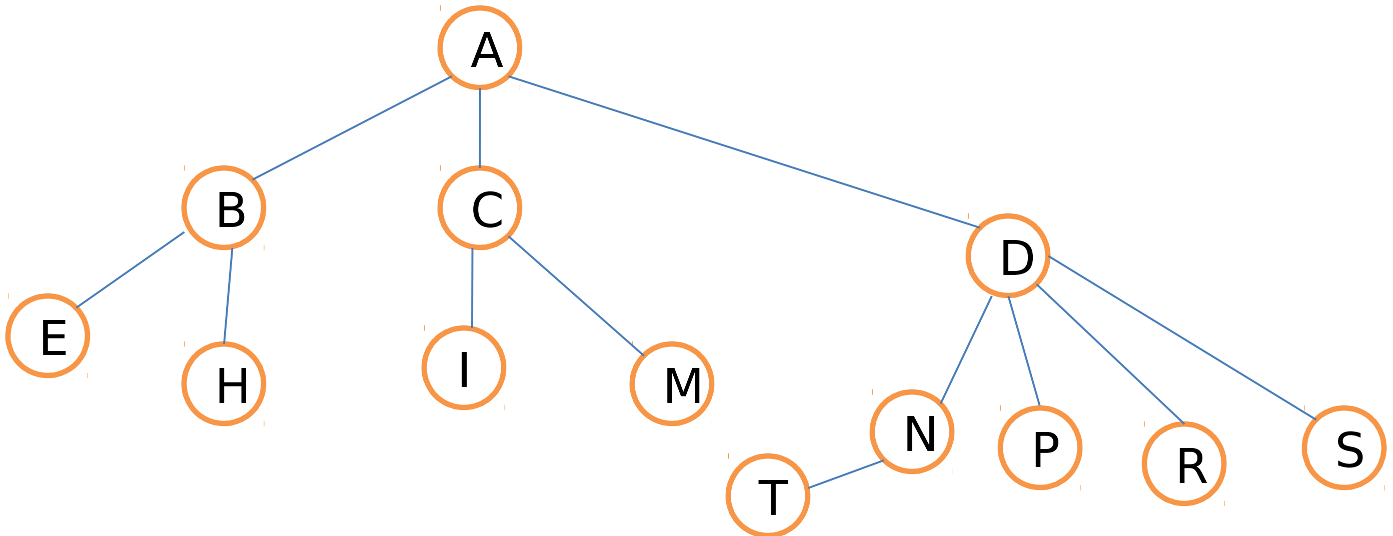
Introduction to trees

Definition

- Non-linear data structure having sorted sequence of items
- A tree is a finite set of one or more data elements called nodes

- One special node is called tree root and others nodes are hierarchically partitioned into disjoint subsets called Subtrees

- **Example**



Introduction to trees cont..

Properties of tree

- **Root:** Top or first element in hierarchy
- **Node :** Basic elements which consists of information and links to other items
- **Degree of a node:** Number of Subtrees of a node e.g. A(3), C(2)
- **Degree of a tree:** maximum of all degree of nodes in a tree e.g. d(4) is the maximum in given tree
- **Leaf nodes:** are nodes having 0 degree e.g. E, H, T, S, I, M, R,

Properties of tree cont..

- **Non- Leaf nodes:** Are nodes not having 0 degree.
- Root node is not included
- Intermediate node from root to leaf. E.g. N, D, C

- **Siblings:** Child nodes of same parent node

- **Levels:** Tree hierarchy is maintained as per following levels

1. Root is at level 0
2. Roots immediate children are at level 1and so on and so forth

- **Depth or height:** of a tree is given by the maximum level of any node

Introduction to tree cont..

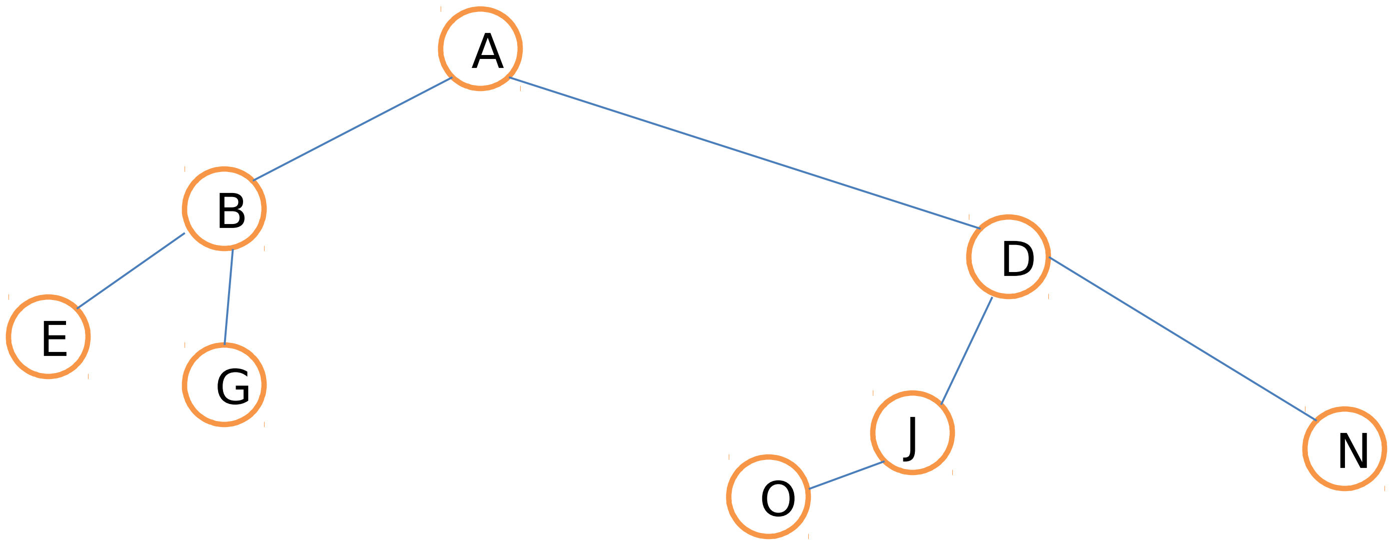
Types of tree

- **Binary tree**

Definition

- A binary tree is a finite set of either empty (represented by a null pointer), or is made of a single node, where the left and right pointers each point to a binary tree.
- If each node of a tree has 0, 1 or 2 Subtrees

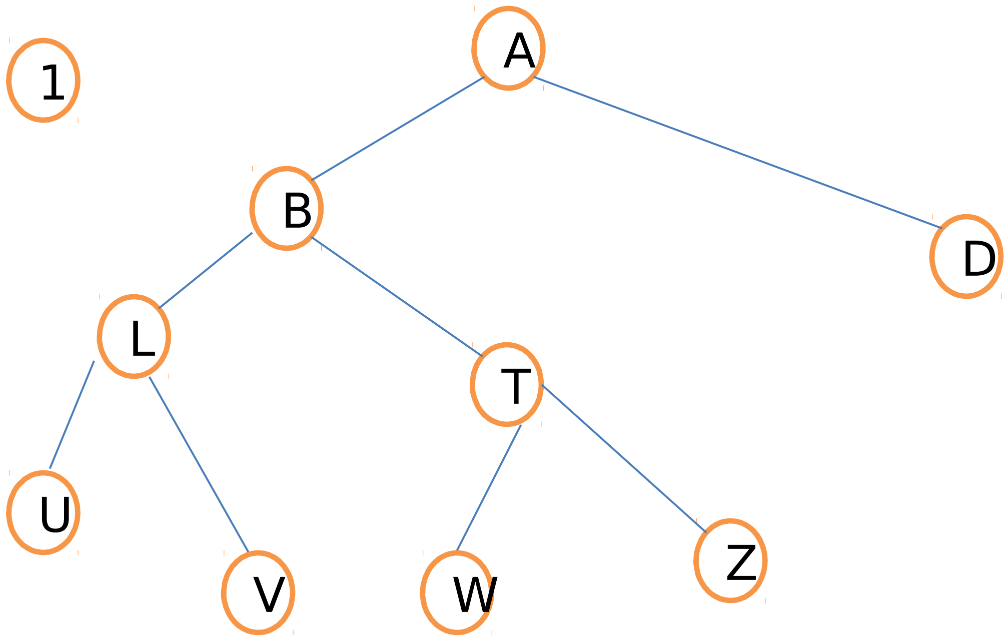
Example

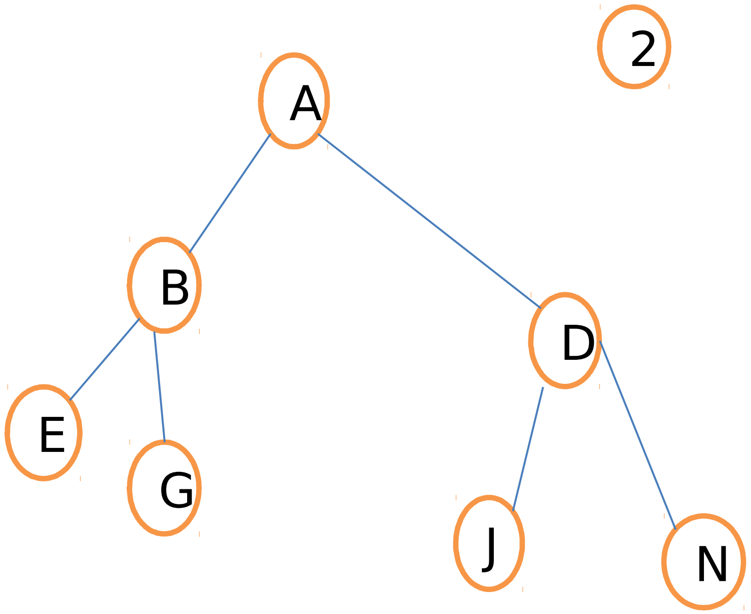


Introduction to tree cont..

- Any node say D in given tree is the ancestor of node J and J is the descendant of D if D is either the father of J or the father of some ancestor of J.

- **Strictly binary trees:** All non –leaf nodes having non empty left and right Subtree.

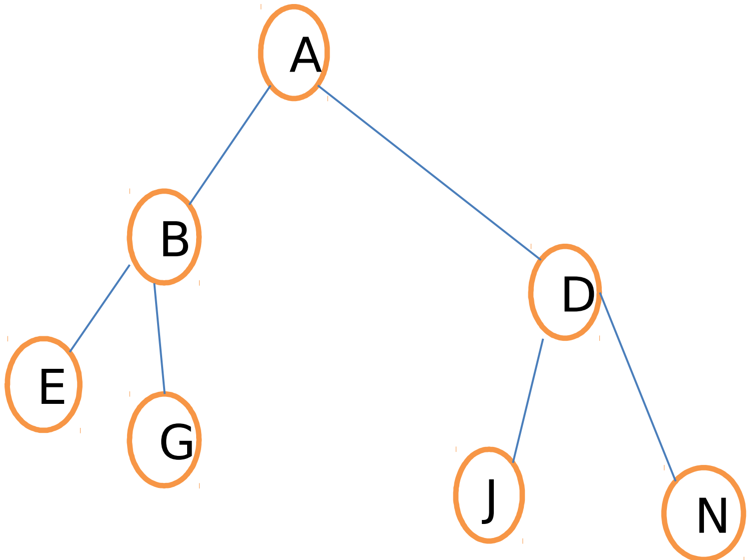




Introduction to tree cont..

- A strictly binary tree with M leaves always contains 2m-1 nodes

- **Complete binary tree** : In a strictly binary tree if all its leaves are at the level of its depth d. E.g. following is the complete binary tree of depth **2**
- **A binary tree can contain at most 2l(2 the power l) nodes at level l**
- **A complete binary tree of depth d contains exactly 2l nodes at each level between 0 and d.**



Introduction to tree cont..

- A complete binary tree of depth **d** contains total number of nodes given by the sum of the number of nodes at each level between 0 and d.
- CBtn=2_0+ 2_1+2_2+…..2_d
- CBtn=2_{d+1} -1

Binary tree representation

- The structure of each node of a binary tree contains three parts
- One **INFO** part, stores the information and 2 pointers which points to the right & left child.
- Each child being a node has also the same structure.

The structure of a binary tree node in C is as follow

```
struct node
{
struct node *lcp ;                  /* points to the left child */
int  info;                          /* info field */
struct node *rcp;                   /* points to the right child */
}
```

Binary tree representation cont..

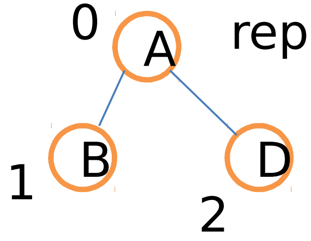
Representation of binary tree.

- Linked list representation
- Array representation

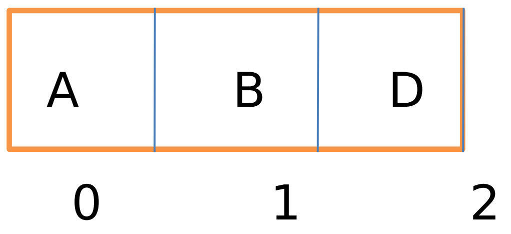
Array representation

- The nodes in array start storing from index 0 to MAXSIZE
- Root node is at index 0.
- Left child node is at next successive memory location then right child node
- E.g.

Binary tree of 3 nodes representation



Array

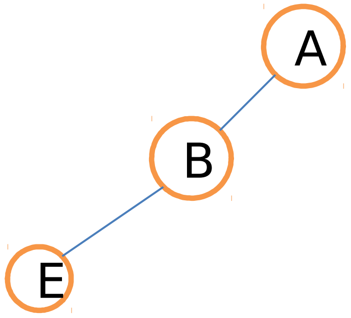


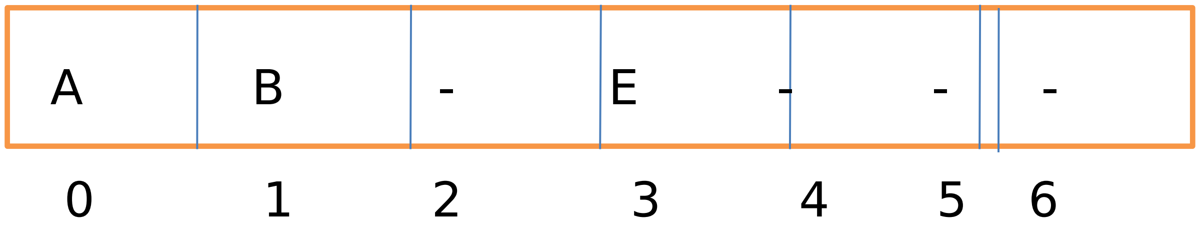
Binary tree representation cont..

In array representation we can identify index number of father , left child and right child node for any node **m, $0 \leq m \leq$ (MAXSIZE)** using

- Left child(lcp) of m: (2m+1)
- Right child(rcp) of m: (2m+2)
- Father of m if m is not equal to 0: floor((m-1)/2)

- The array representation is ideal for the complete binary tree but not suitable for others
- Waste memory space **E.g.** Consider left skew binary tree





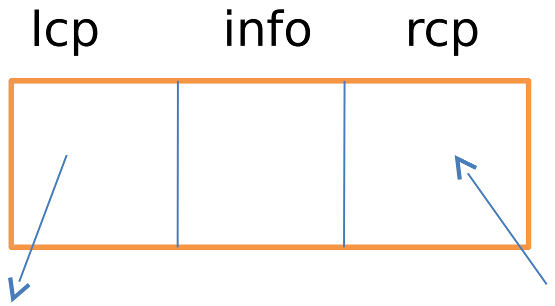
Binary tree representation cont..

Linked list representation

- Each node contains the address of the left child and the right child.
- If any node has its left or right child empty then it will have in its respective link field, a null value.
- A leaf node has null value in both of its links.

- E.g. Representation of a node in C and list representation

```
struct node
{
struct node *lcp ;       /* points to the left child */
char  info;                          /* info field */
struct node *rcp;       /* points to the right child
*/
}
```



Operation of binary tree

Operation on Binary tree

- Creating a binary tree
- Tree traversal
- Insertion of nodes in the tree
- Searching for nodes

Tree traversals: way of visiting each node exactly once and produce linear order of nodes

- All the traversal methods are based on recursive functions

1.Preorder
2. Inorder
3.Postorder

Tree traversals

Inorder traversal:

Algorithm

The algorithm for inorder traversal is as follows.

```
Struct node
{
struct node * lc;
int data;
struct node * rc;
};
void inorder(struct node * root);
{
if(root != NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
inorder(root->rc);
```

To traverse a non empty tree in inorder the following steps are followed recursively.
Traverse the left subtree
Visit the Root
Traverse the right subtree
The inorder traversal of the tree shown below is as follows.

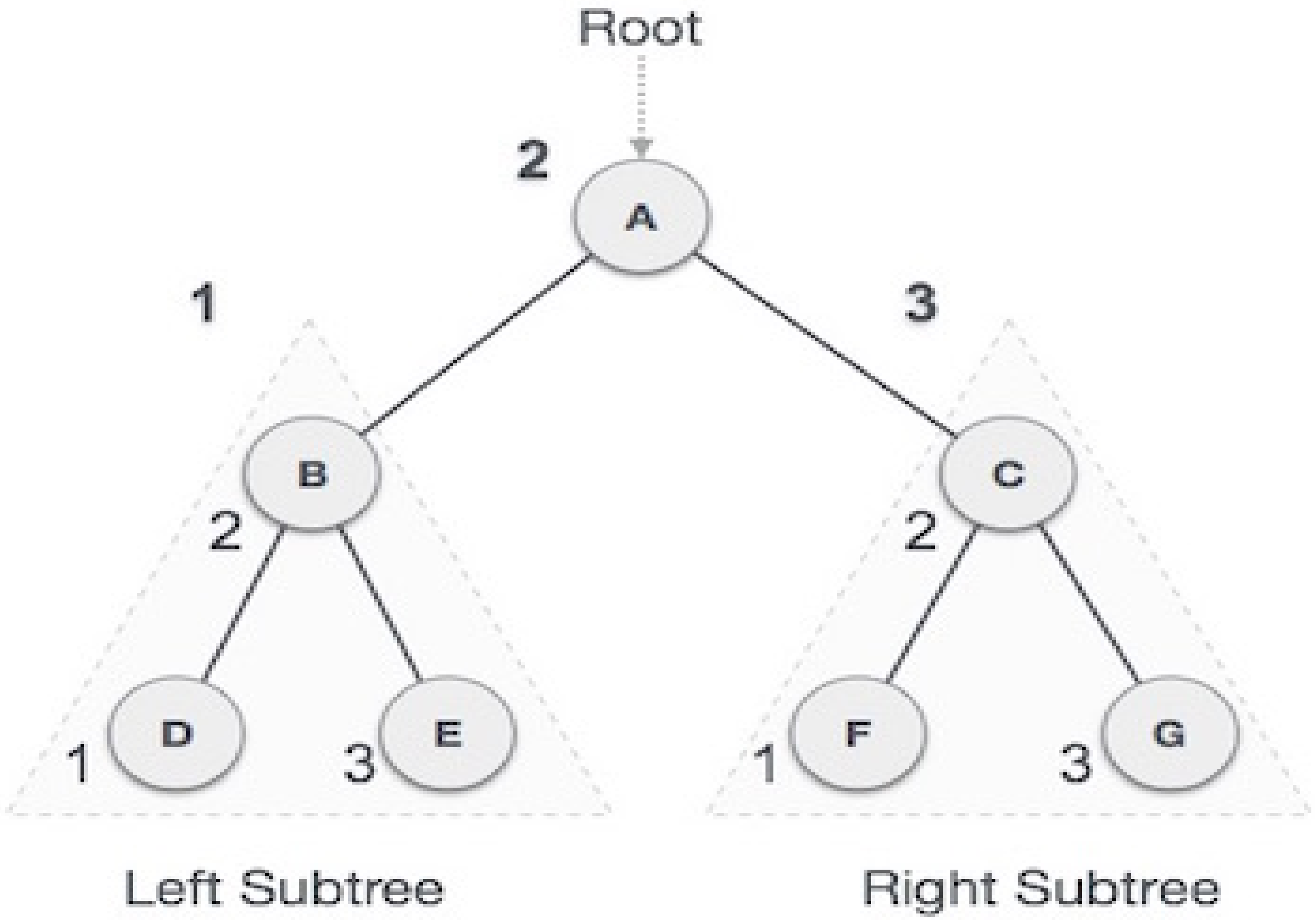
Example of inoreder traversal

In-order Traversal

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.

In-order Traversal



In-order Traversal

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 −

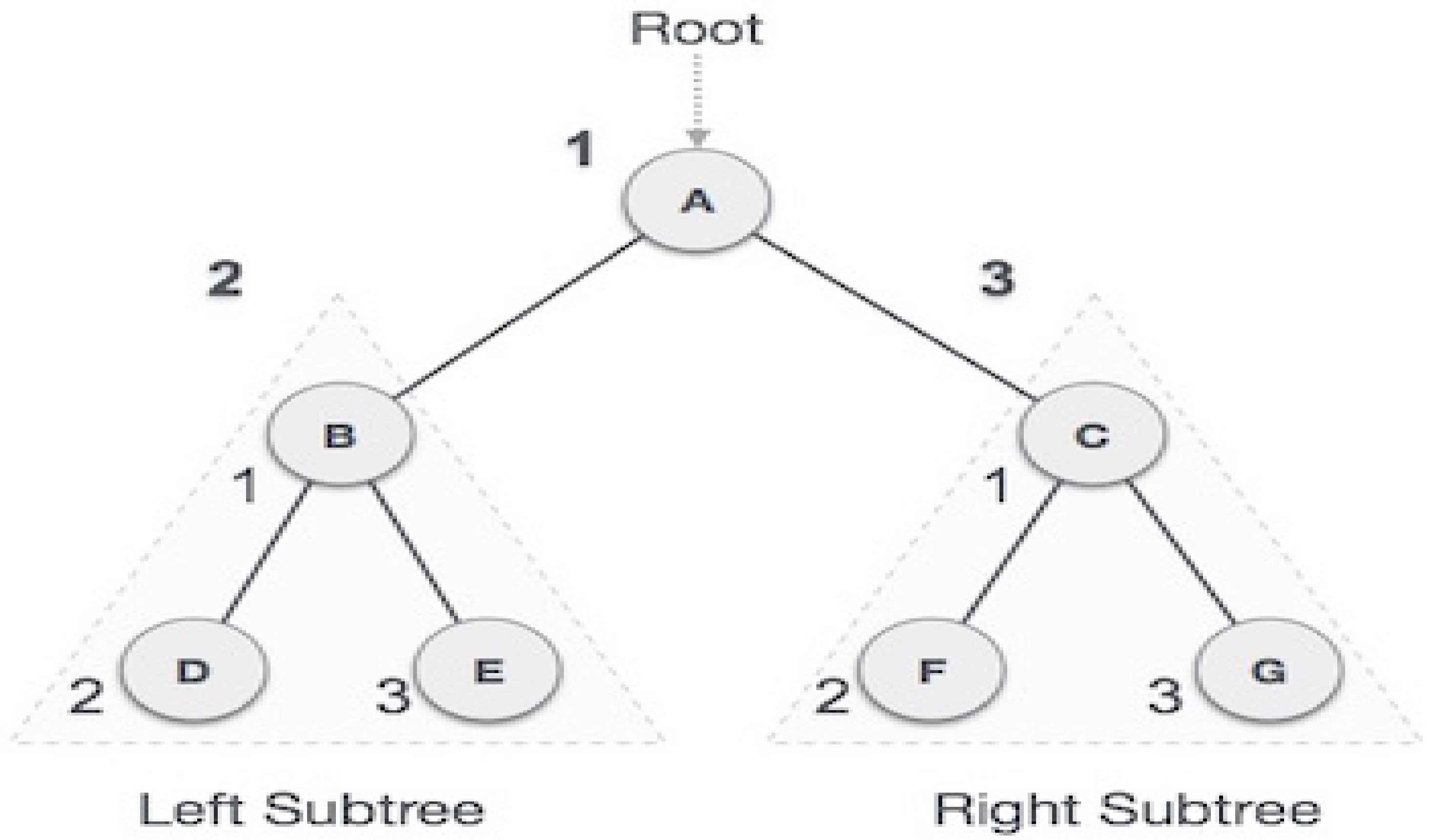
D → B → E → A → F → C → G

Algorithm:

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

Pre-order Traversal

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



Pre-order Traversal

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 −

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

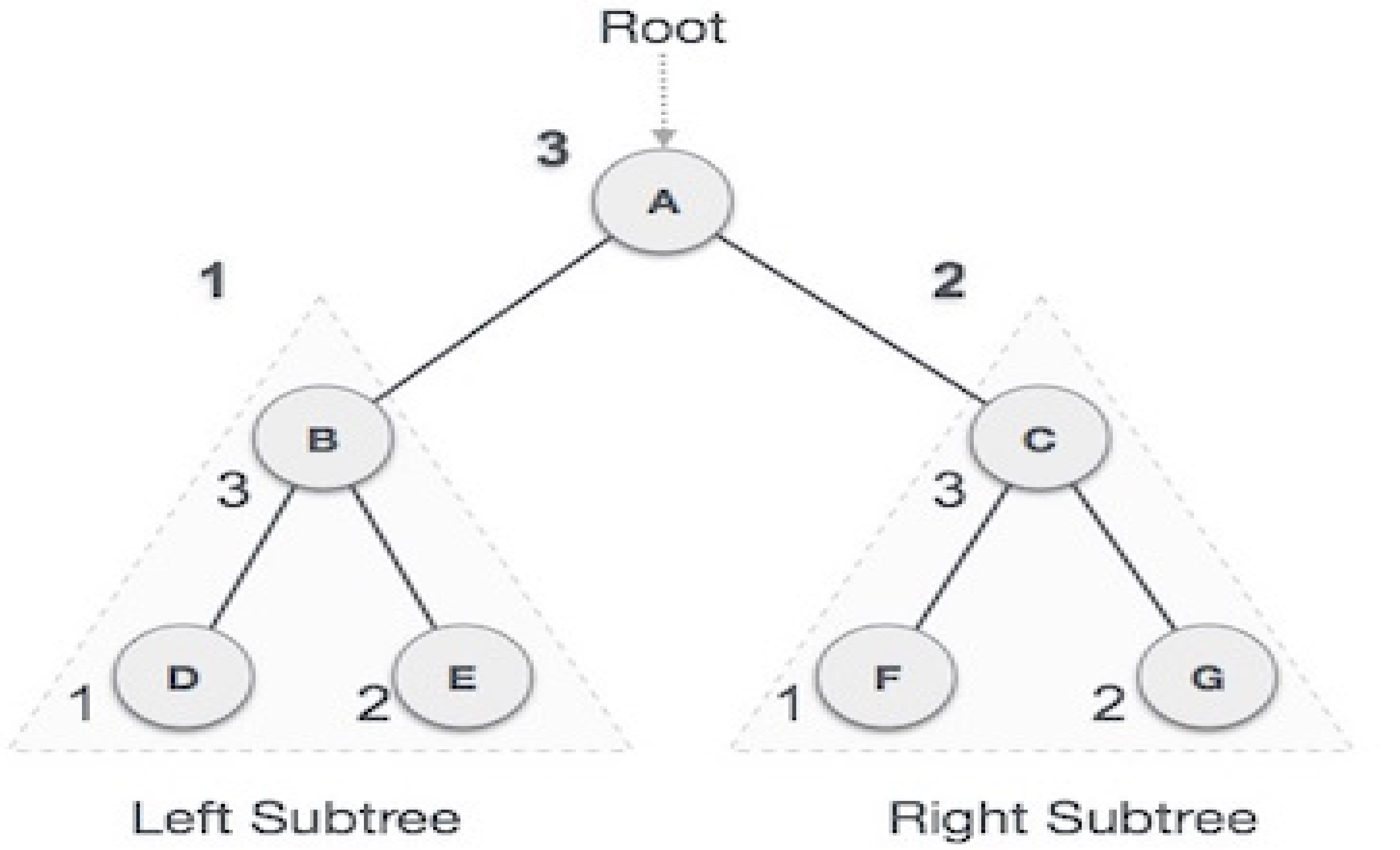
Algorithm:

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

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.

Post-order Traversal



Post-order Traversal

We start from A, and following Post-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 → E → B → F → G → C → A

Post-order Traversal

Algorithm:

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

Binary Search Tree(BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −**left_subtree (keys) < node (key) ≤ right_subtree (keys)**

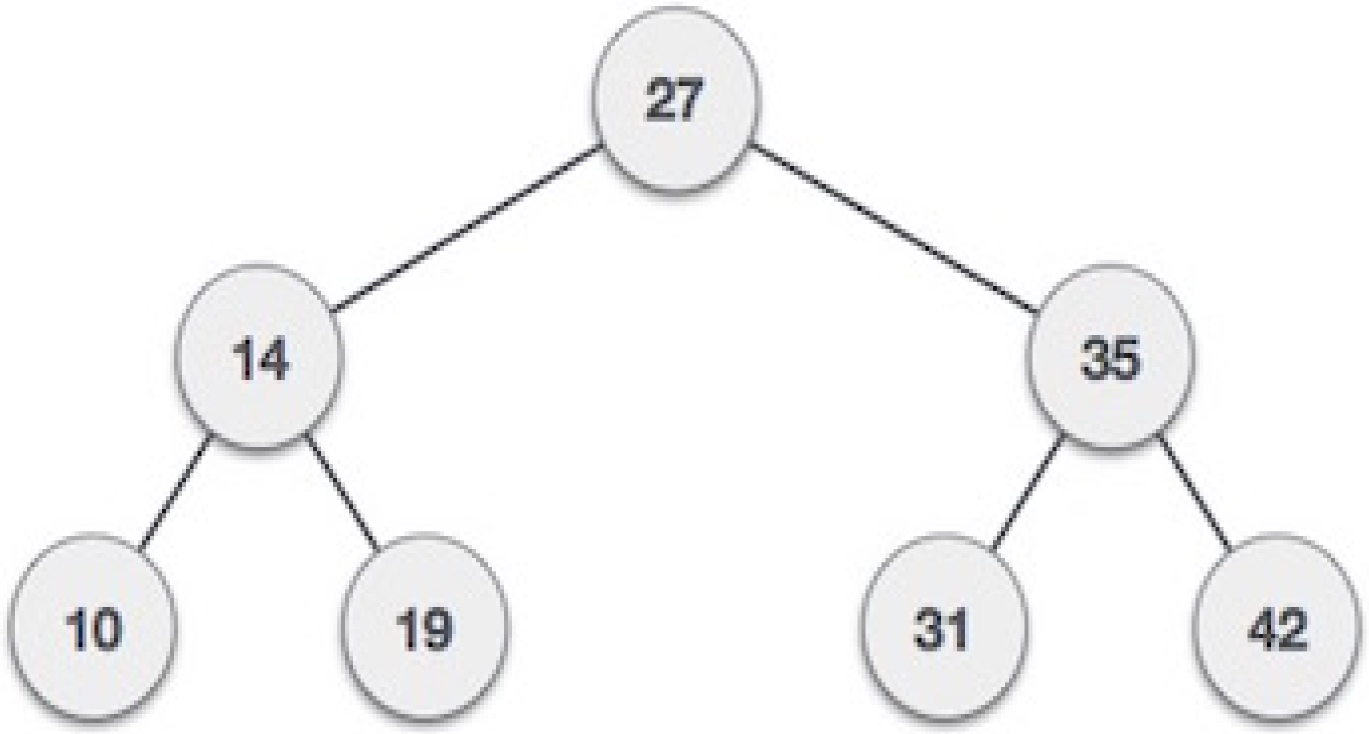
Binary Search Tree(BST)..

Representation:

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −

Binary Search Tree(BST)..



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Binary Search Tree(BST)..

Basic Operations:
Following are the basic operations of a tree −
Search − Searches an element in a tree.
Insert − Inserts an element 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.

Binary Search Tree(BST)..

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

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Binary Search Tree(BST)..

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.

Binary Search Tree(BST)..

Algorithm:

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

   while(current->data != data){

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

Binary Search Tree(BST)..

```
//go to left tree
      if(current->data > data){
        current = current->leftChild;
      }  //else go to right tree
      else {
        current = current->rightChild;
      }
    //not found
      if(current == NULL){
        return NULL;
      }
    }
  }
  return current;
}
```

Binary Search Tree(BST)..

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.

Binary Search Tree(BST)..

Algorithm:

```
void insert(int data) {
        struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;
```

Binary Search Tree(BST)..

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

Binary Search Tree(BST)..

```
while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
```

Binary Search Tree(BST)..

```
}  //go to right of the tree
        else {
           current = current->rightChild;

           //insert to the right
           if(current == NULL) {
              parent->rightChild = tempNode;
              return;
           }
        }
     }
  }
}
```

Thank You

B -Trees

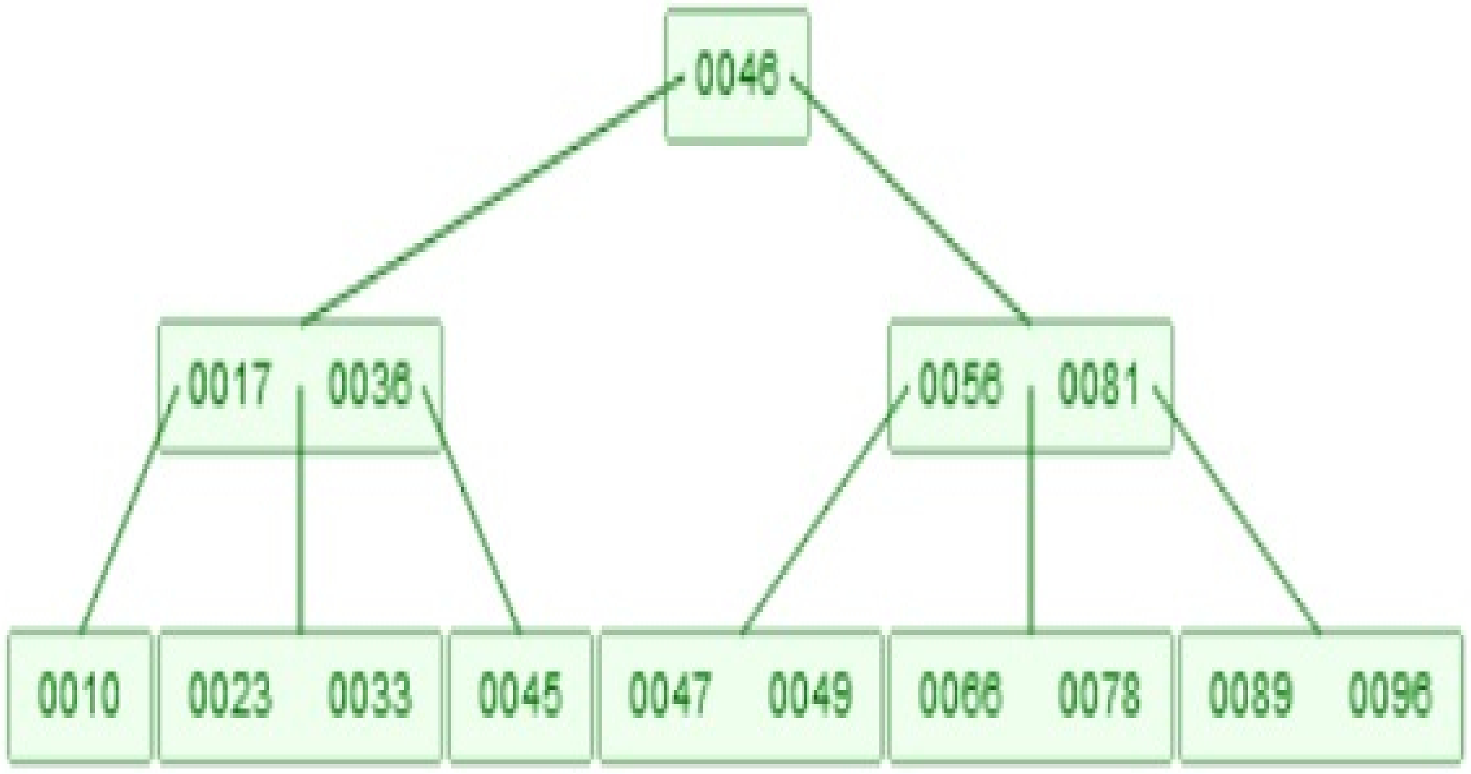
The B-Trees are specialized m-way search tree. This can be widely used for disc access. A B-tree of order m, can have maximum m-1 keys and m children. This can store large number of elements in a single node. So the height is relatively small. This is one great advantage of B-Trees.

B-Tree has all of the properties of one m-way tree. It has some other properties.

- Every node in B-Tree will hold maximum m children
- Every node except root and leaves, can hold at least m/2 children
- The root nodes must have at least two children.
- All leaf nodes must have at same level

B-Trees..

Example of B-Tree:



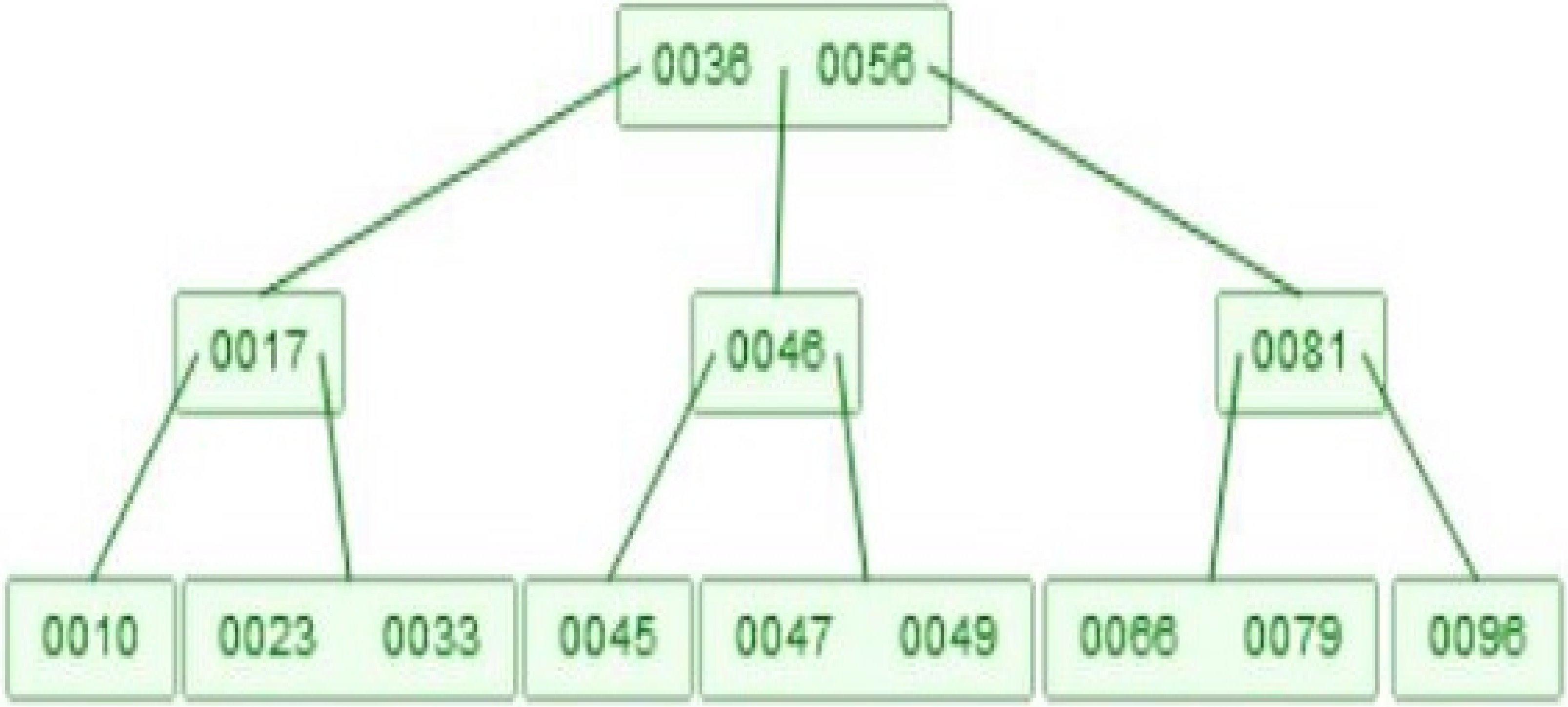
B-Trees..

This supports basic operations like searching, insertion, deletion. In each node, the item will be sorted. The element at position i has child before and after it. So children sored before will hold smaller values, and children present at right will hold bigger values.

Deletion in B-Trees

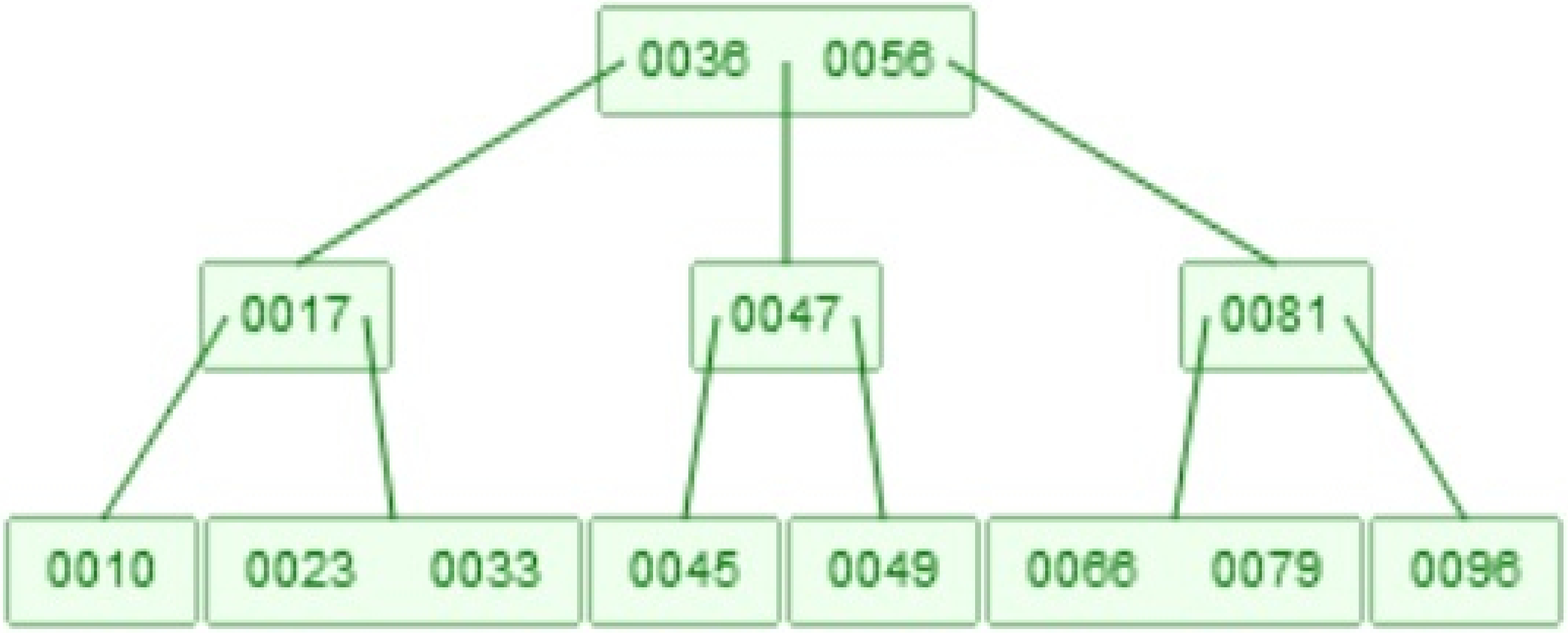
Deletion has two parts. At first we have to find the element. That strategy is like the querying. Now for deletion, we have to care about some rules. One node must have at-least m/2 elements. So if we delete, one element, and it has less than m-1 elements remaining, then it will adjust itself. If the entire node is deleted, then its children will be merged, and if their size issame as m, then split them into two parts, and again the median value will go up.

B-Trees


0036 0056
0017
0046
0081
0010
0023 0033
0045
0047 0049
0066 0079
0096

B-Trees

Suppose we want to delete 46. Now there are two children. [45], and [47, 49], then they will be merged, it will be [45, 47, 49], now 47 will go up.



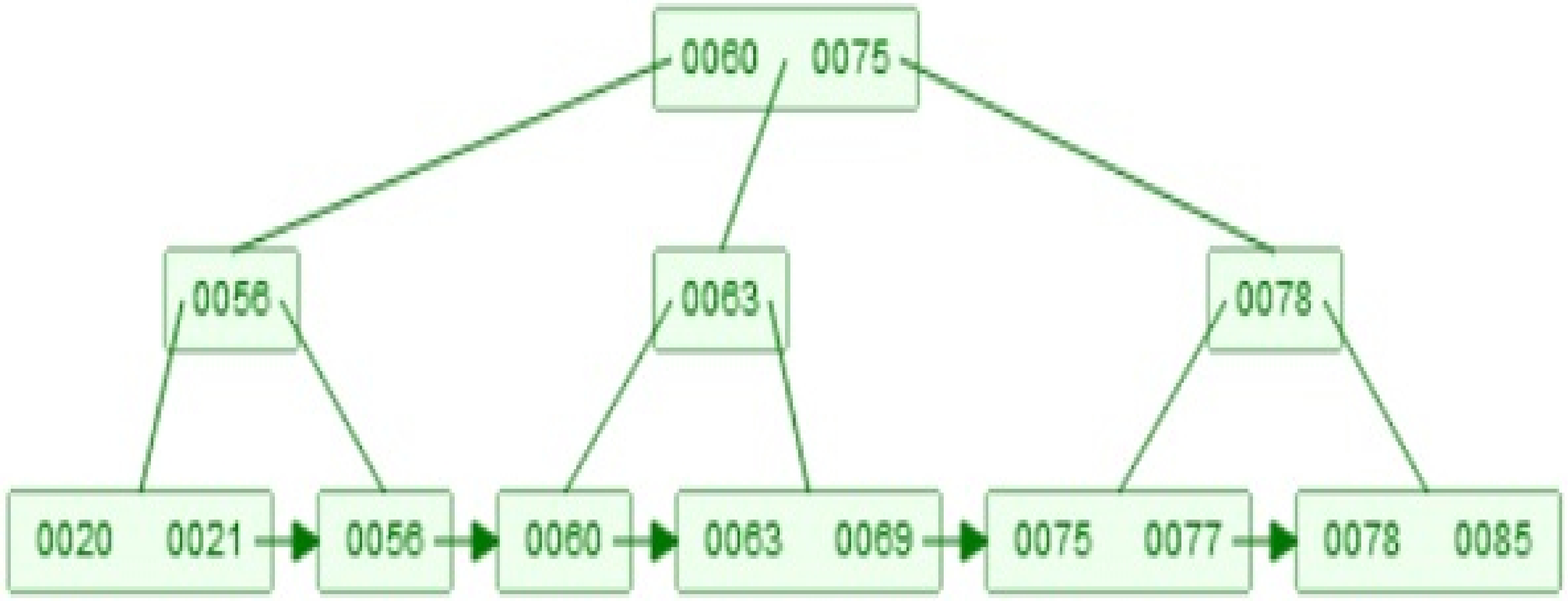
B+Trees

The B+ Trees are extended version of B-Trees. This tree supports better insertion, deletion and searching over B-Tree.

B-trees, the keys and the record values are stored in the internal as well as leaf nodes. In B+ tree records, can be stored at the leaf node, internal nodes will store the key values only. The leaf nodes of the B+ trees are also linked like linked list

B+Trees

This supports basic operations like searching, insertion, deletion. In each node, the item will be sorted. The element at position i has child before and after it. So children sored before will hold smaller values, and children present at right will hold bigger values.



B+Trees

Advantages over B-Tree

- Records can be fetched in equal number of disk accesses
- Height of the tree remains balanced, and less as compared to B-Trees
- As the leafs are connected like linked list, we can search elements in sequential manner also
- Keys are used for indexing
- The searching is faster, as data are stored at leaf level only.

Threaded Binary Trees

We know that the binary tree nodes may have at most two children. But if they have only one children, or no children, the link part in the linked list representation remains null. Using threaded binary tree representation, we can reuse that empty links by making some threads.

If one node has some vacant left or right child area, that will be used as thread. There are two types of threaded binary tree. The single threaded tree or fully threaded binary tree. In single threaded mode, there are another two variations. Left threaded and right threaded.

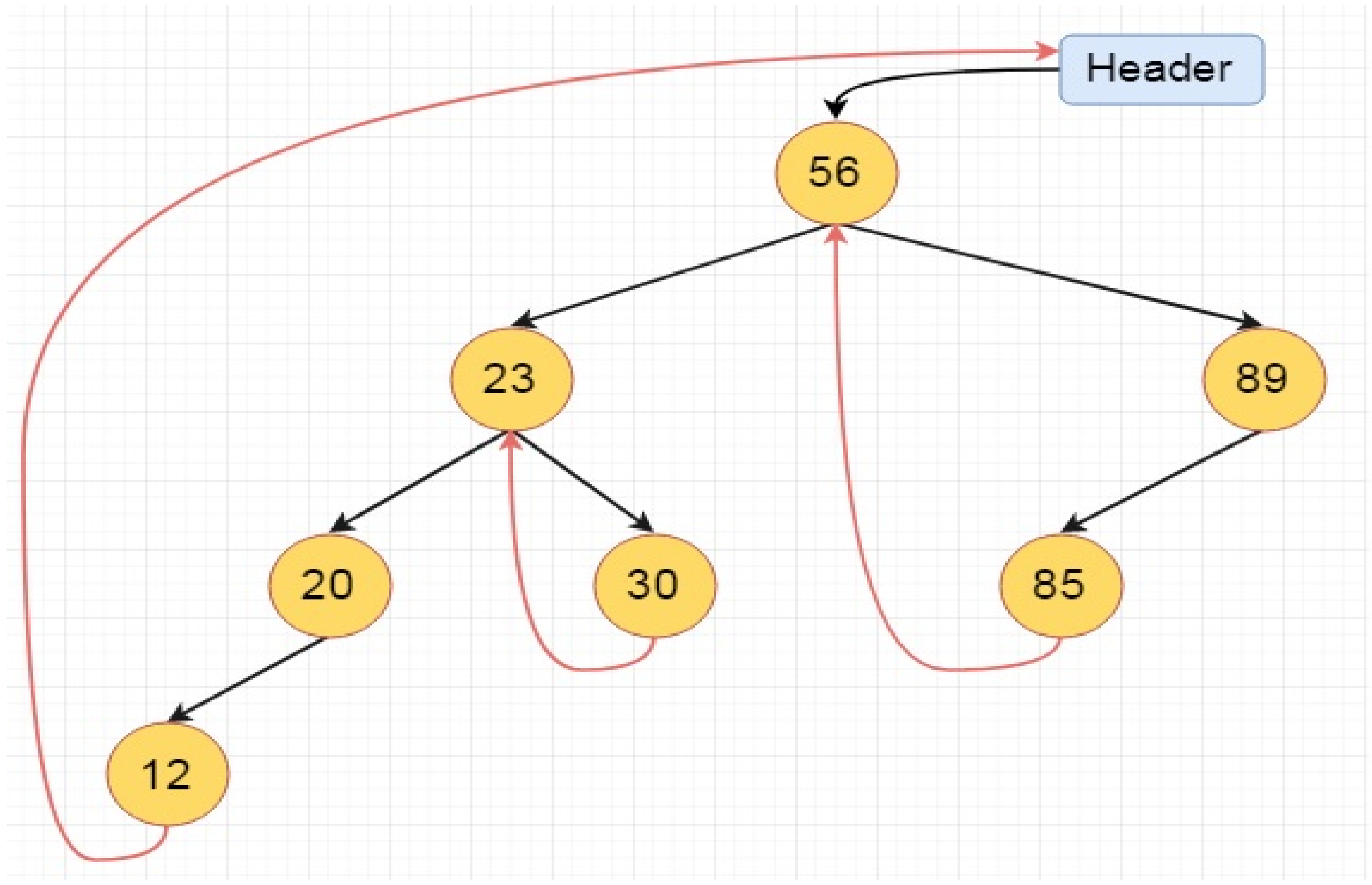
Threaded Binary Trees

In the left threaded mode if some node has no left child, then the left pointer will point to its inorder predecessor, similarly in the right threaded mode if some node has no right child, then the right pointer will point to its inorder successor. In both cases, if no successor or predecessor is present, then it will point to header node.

For fully threaded binary tree, each node has five fields. Three fields like normal binary tree node, another two fields to store Boolean value to denote whether link of that side is actual link or thread.

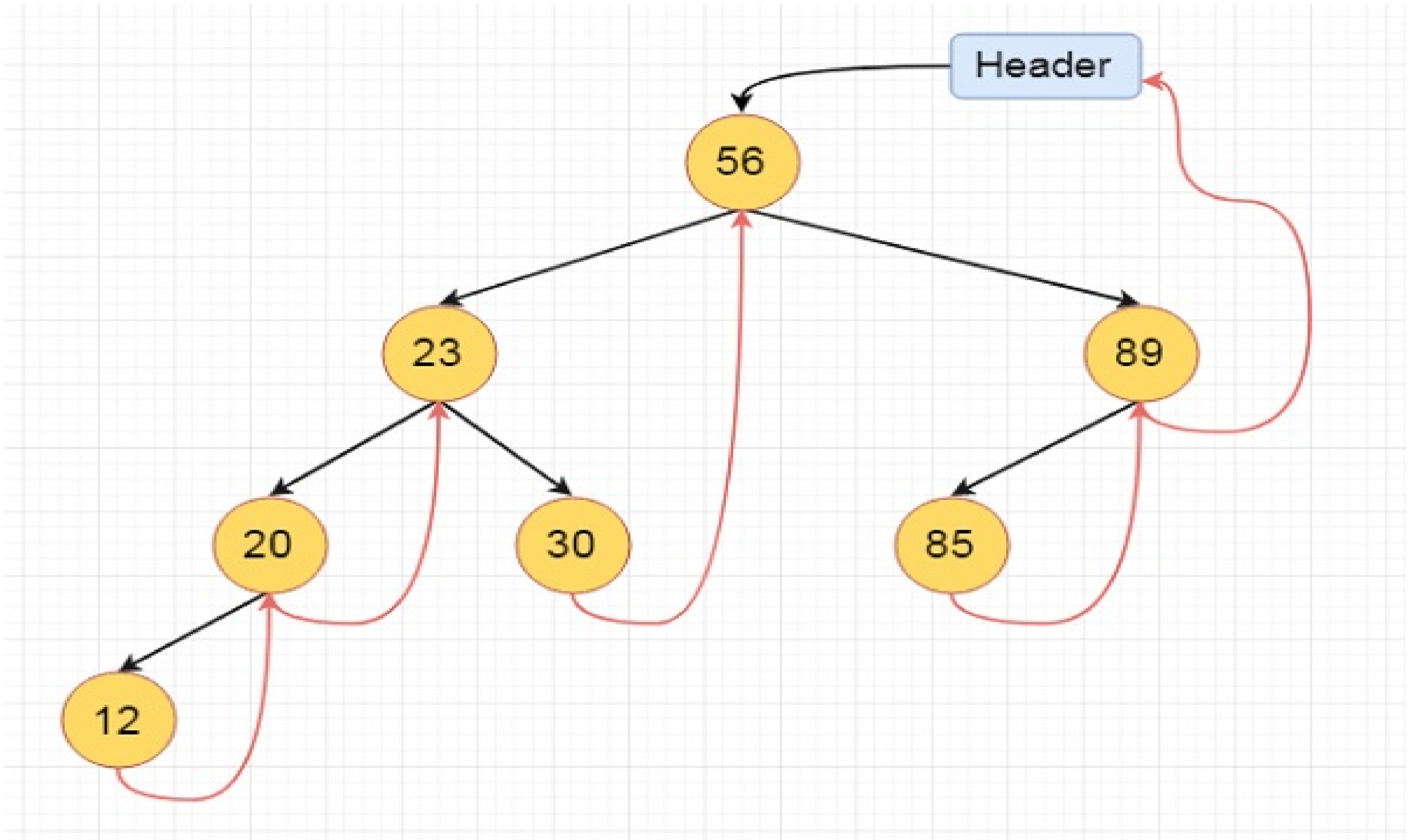
Threaded Binary Trees

Left threaded binary Trees



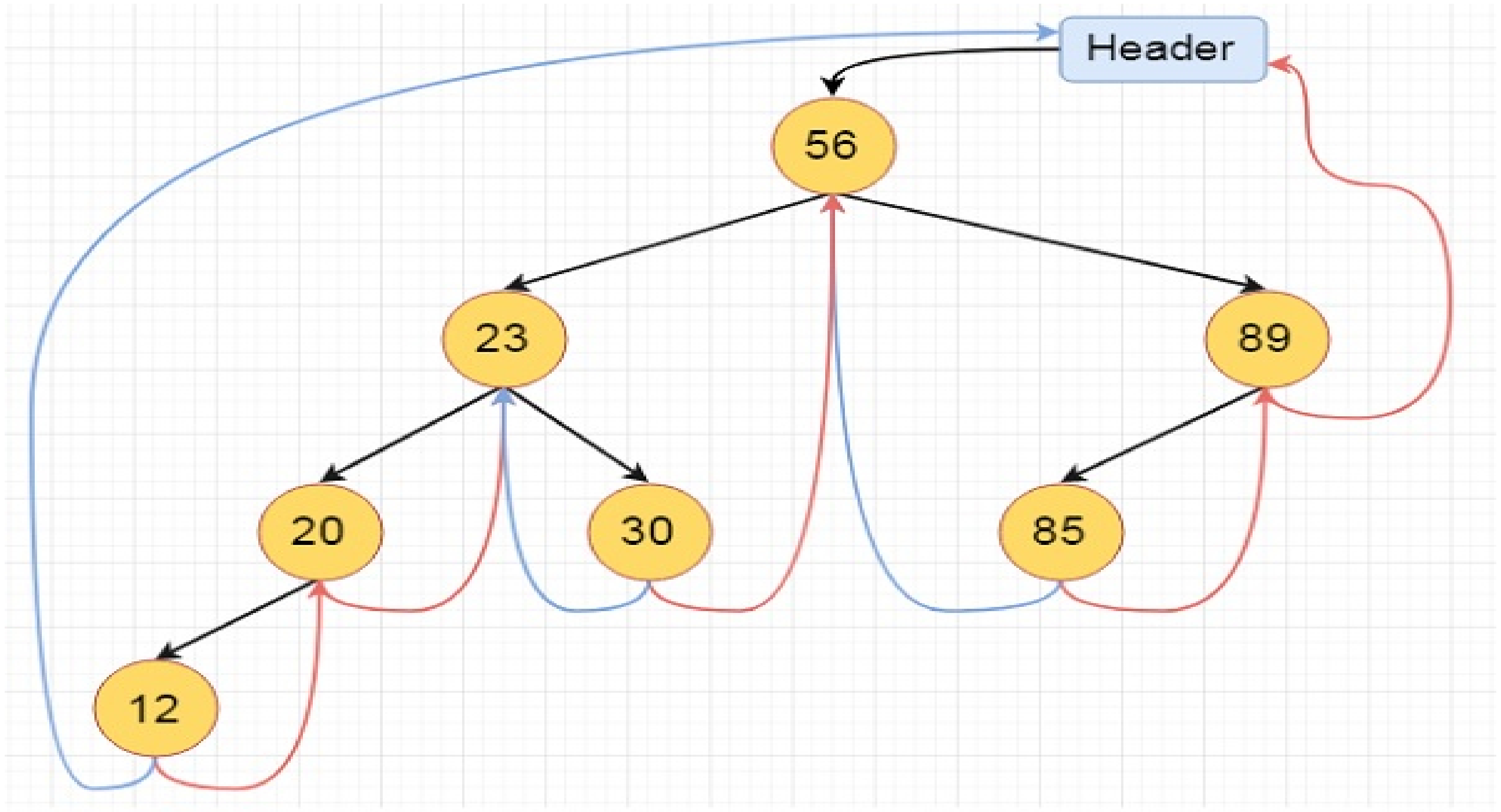
Threaded Binary Trees

Right threaded binary Trees



Threaded Binary Trees

Fully threaded binary Trees



Thank You