||JAI SRI GURUDEV||
Sri AdichunchanagiriShikshana Trust ®
SJB INSTITUTE OF TECHNOLOGY
An Autonomous Institution under Visvesvaraya Technological University, Belagavi
Approved by AICTE, New Delhi, Accredited by NAAC, New Delhi with 'A+' Grade, Accredited by National Board of Accreditation, New Delhi
Recognized by UGC, New Delhi with 2(f) and 12(B), Certified by ISO 9001-2015
No. 67, BGS Health & Education City, Dr. Vishnuvardhan Road, Kengeri, Bengaluru - 560 060

NOTES

Module - 4

Subject Name: **DSA**
Subject Code: **23CDT302**

Prepared By:
Dr. Rekha B
Professor

Department of CSE (Data Science)

Module 4 - Trees

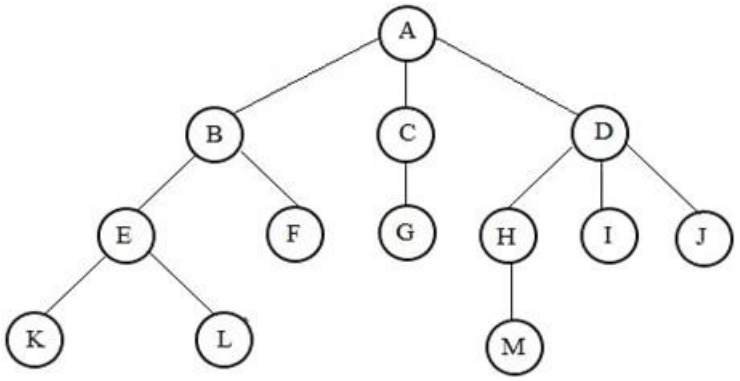
Chapter – Trees

- Terminologies
- Properties of Binary Trees
- Array and Linked Representation
- Binary Tree Traversals
- Threaded Binary Trees -
- Binary Search Trees – Definition, Insertion, Deletion, Traversal, and Searching Operation on Binary Search Tree
- Application of Trees-Evaluation of Expression

Definition

A tree is a finite set of one or more nodes such that

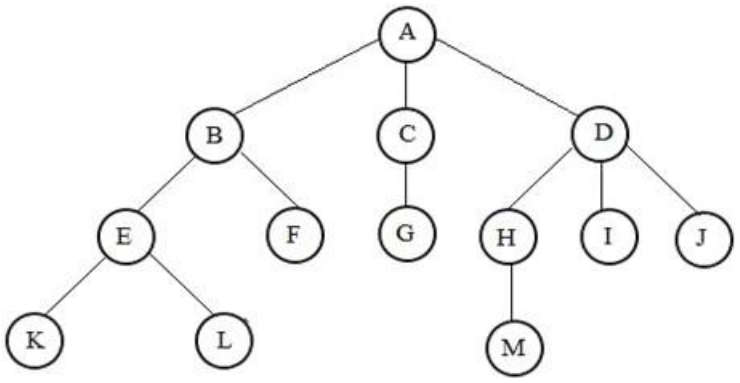
- There is a specially designated node called root.
- The remaining nodes are partitioned into n >= 0 disjoint sets T1,…,Tn, where each of these sets is atree. T1,…,Tn are called the subtrees of the root.



Every node in the tree is the root of some subtree.

Terminologies

In data structures, a tree is a hierarchical structure used to represent data. It consists of nodes connected by edges, forming a parent-child relationship. Understanding the various terminologies associated with trees isessential to grasp their structure and operation.



Node

A fundamental unit of a tree that stores data and may have references (pointers) to other nodes (itschildren).Each node can be a parent to other nodes (child nodes).

---

1. Root Node

- The **root node** is the topmost node of the tree.
- In this case, **A** is the root node.

2. Parent Node

- A **parent node** is a node that has one or more child nodes.
- Examples:
  - **A** is the parent of **B**, **C**, and **D**.
  - **B** is the parent of **E** and **F**.

3. Child Node

- A **child node** is a node that has a parent node.
- Examples:
  - **B**, **C**, and **D** are children of **A**.
  - **K** and **L** are children of **E**.

4. Leaf Node

- A **leaf node** is a node that has no children. It is at the end of a branch.
- Examples:
  - **F**, **G**, **I**, **J**, **K**, **L**, and **M** are leaf nodes.

5. Internal Node

- An **internal node** is a node that has at least one child (not a leaf).
- Examples:
  - **A**, **B**, **D**, **E**, and **H** are internal nodes.

6. Edge

- An **edge** is a connection between two nodes.
- Examples:
  - There is an edge between **A** and **B**, **B** and **E**, etc.

7. Path

- A **path** is a sequence of nodes connected by edges.
- Example:
  - A path from **A** to **M** is **A → D → H → M**.

8. Level

- The **level** of a node represents its distance from the root node.
- Example:
  - **A** is at level 0.
  - **B**, **C**, and **D** are at level 1.
  - **E**, **F**, **G**, **H**, **I**, and **J** are at level 2.
  - **K**, **L**, and **M** are at level 3.

9. Subtree

- A **subtree** is any node and all its descendants.
- Example:
  - The subtree rooted at **D** includes **D**, **H**, **I**, **J**, and **M**.

10. Ancestor and Descendant

- An **ancestor** of a node is any node on the path from the node to the root.
- Example:
  - **A** is an ancestor of **E**, **K**, and **M**.
- A **descendant** is any node that is below a given node in the tree.
- Example:
  - **H**, **I**, **J**, and **M** are descendants of **D**.

11. Siblings

- **Siblings** are nodes that share the same parent.
- Examples:
  - **B**, **C**, and **D** are siblings.
  - **E** and **F** are siblings

12. Depth

The depth of the tree is **3** (the number of edges from **A** to the deepest leaf nodes). The depth of the root node is zero.

- The longest path in this tree is from the root node **A** to **K**, **L**, or **M**. The paths are:
  - **A → B → E → K** (3 edges)
  - **A → B → E → L** (3 edges)
  - **A → D → H → M** (3 edges)

Binary Trees

A binary tree is a hierarchical data structure where each node has at most two children, often referred to as the left child and the right child. This structure is recursive, meaning each child node can itself be the root of another binary tree.

Abstract Data Type, skewed tree, complete binary tree

Abstract Data Type

structure *Binary_Tree* (abbreviated *BinTree*) is
objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.
functions:
for all *bt*,*bt*1,*bt*2 ∈ *BinTree*, *item* ∈ *element*

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

Abstract data type *Binary_Tree*

Skewed tree

Skewed tree (Degenerate): Each parent node has only one child. It may arrange the nodes either in left or towards right. The node if it follows towards left means all its child will be in left. Vice versa for right.

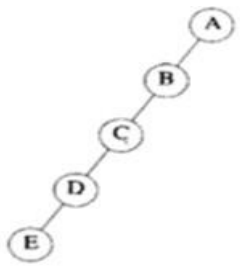


Figure 1(a) Skewed binary tree

-	A	B		C				D		E
0	1	2	3	4	5	6	7	8		16

For the skewed tree less than half the array is utilized.

The node if it follows towards left means all its child will be in left. **Left – 1*2.**

The node if it follows towards right means all its child will be in right. **Right – 1*2+1.**

Complete binary tree

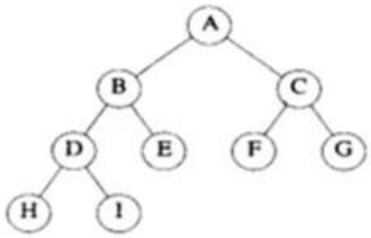


Figure 1(b) Complete binary tree

A binary tree T is said to complete if all its level except possibly the last level, have the maximum number node 2i-1, i>=0 and if all the nodes at the last level appears as far left as possible.

-	A	B		C				D		E
0	1	2	3	4	5	6	7	8		16

COMPLETE(is ideal)
1+1

For complete binary tree the array representation is ideal, as no space is wasted.

Properties of Binary Tree

Node Degree: In a binary tree, each node has a degree of 0, 1, or 2, indicating the number of children it has.

Height of Tree: The height of a binary tree is the length of the longest path from the root to a leaf node.

Level of Node: The level of a node is the number of edges from the root to that node.

Size of Tree: The total number of nodes in the tree.

Types of Binary Trees

Full Binary Tree: Each node has 0 or 2 children.

Complete Binary Tree: All levels are fully filled except possibly the last, which is filled from left to right.

Perfect Binary Tree: All internal nodes have two children, and all leaves are at the same level.

Balanced Binary Tree: The height of the left and right subtrees of any node differ by at most one.

Degenerate (Skewed) Tree: Each parent node has only one child.

Forest: A collection of disjoint trees (i.e., a set of trees).

Array and Linked Representation

The storage representation of tree can be classified as:

Array Representation

Linked Representation

Array representation:

A tree can be represented using an array, which is called sequential representation.

The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes.

Position 0 of this array is left empty and the node numbered i is mapped to position i of the array.

Below figure shows the array representation for both the trees of figure (a).

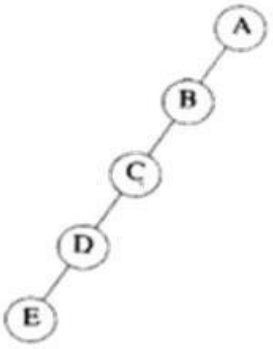


Figure 1(a) Skewed binary tree

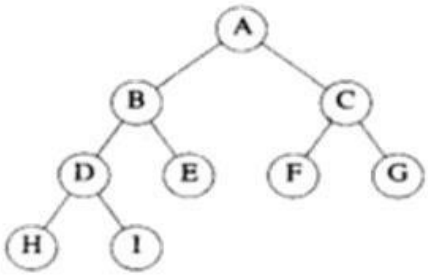
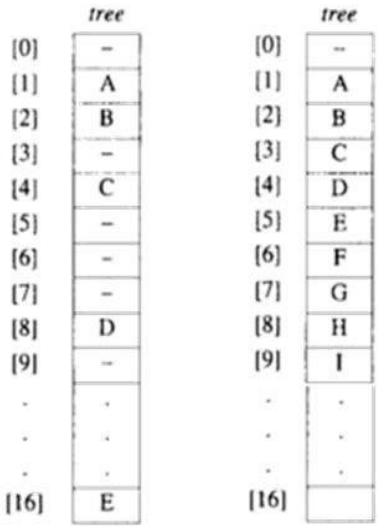


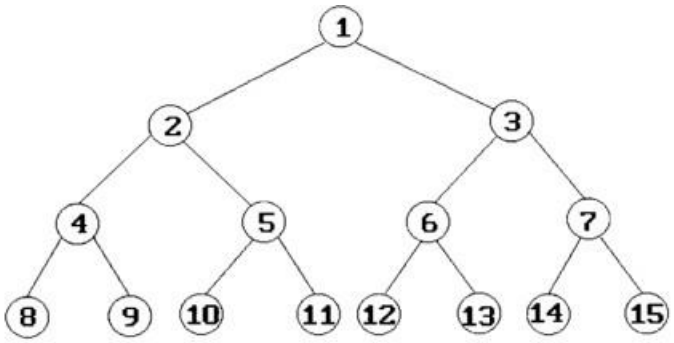
Figure 1(b) Complete binary tree



(a). Tree of figure 1(a) (b). Tree of figure 1(b)

For complete binary tree the array representation is ideal, as no space is wasted.

For the skewed tree less than half the array is utilized.



Full binary tree of depth 4 with sequential node numbers

. **Linked representation:**

The problems in array representation are:

• It is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.

The insertion and deletion of nodes from the middle of a tree require the movement of many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome by linked representation

Each node has three fields,

• LeftChild- which contains the address of left subtree

•RightChild - which contains the address of right subtree.

•Data - which contains the actual information

C Code for node:

```
struct node {
                    int data;
                    struct node  *leftChild,*rightChild;
                };
typedef struct node *treepointer;
```

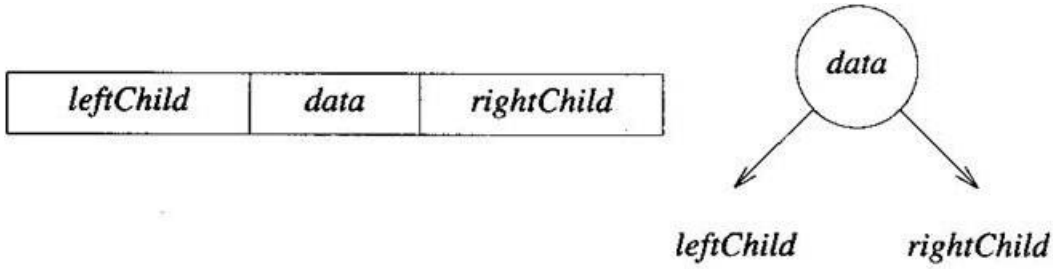


Figure: Node representation

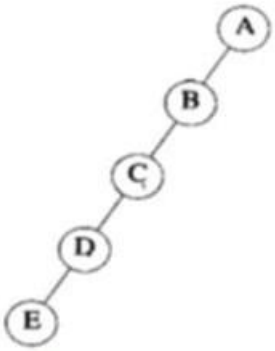


Figure 1(a) Skewed binary tree

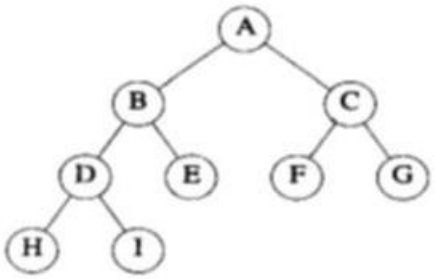
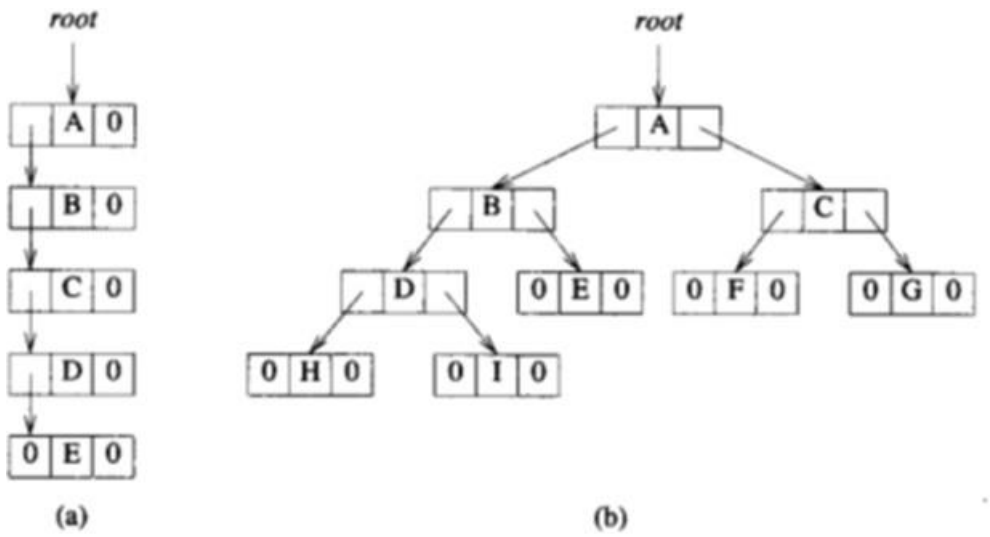


Figure 1(b) Complete binary tree

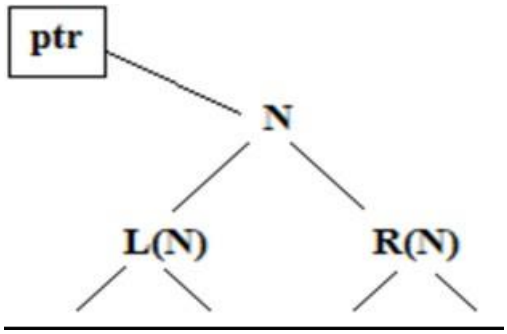


Linked representation of the binary tree

Binary Tree Traversals

Visiting each node in a tree exactly once is called tree traversal.

1. Preorder
2. Inorder
3. Postorder



1. Preorder: Preorder is the procedure of visiting a node, traverse left and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.

Recursion function:

The Preorder traversal of a binary tree can be recursively defined as

- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

```
void preorder (treepointer ptr)
{
        if (ptr)
        {
                printf ("%d",ptr→data)
                preorder (ptr→leftchild);
                preorder (ptr→rightchild);
           }
  }
```

.

2. Inorder: Inorder traversal calls for moving down the tree toward the left until you cannot go further. Then visit the node, move one node to the right and continue. If no move can be done, then go back one more node.

Let ptr is the pointer which contains the location of the node N currently being scanned. L(N) denotes the leftchild of node N and R(N) is the right child of node N.

Recursion function:

The inorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

```
void inorder(treepointer ptr)
{
        if (ptr)
        {
                inorder (ptr→leftchild);
                printf ("%d",ptr→data);
                inorder (ptr→rightchild);
          }
   }
```

3. Postorder: Postorder traversal calls for moving down the tree towards the left until you can go no further. Then move to the right node and then visit the node and continue.

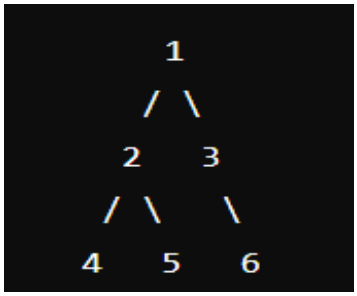
Recursion function:

The Postorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

```
void postorder(treepointerptr)
{
        if (ptr)
        {
                postorder (ptr→leftchild);
                postorder (ptr→rightchild);
                printf ("%d",ptr→data);
         }
   }
```

Examples:



Traversal of the Binary Tree:

Pre-order Traversal (NLR): Visit the root first, then the left subtree, followed by the right subtree.

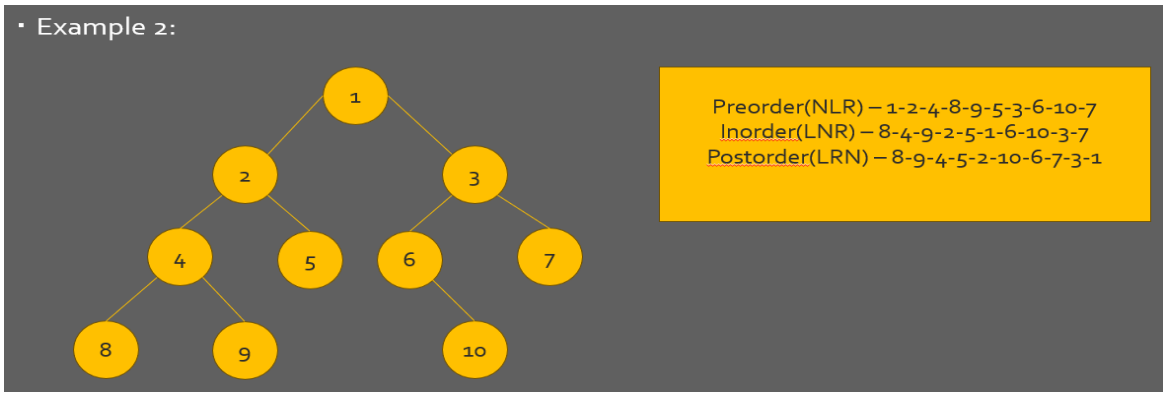
Order: 1, 2, 4, 5, 3, 6

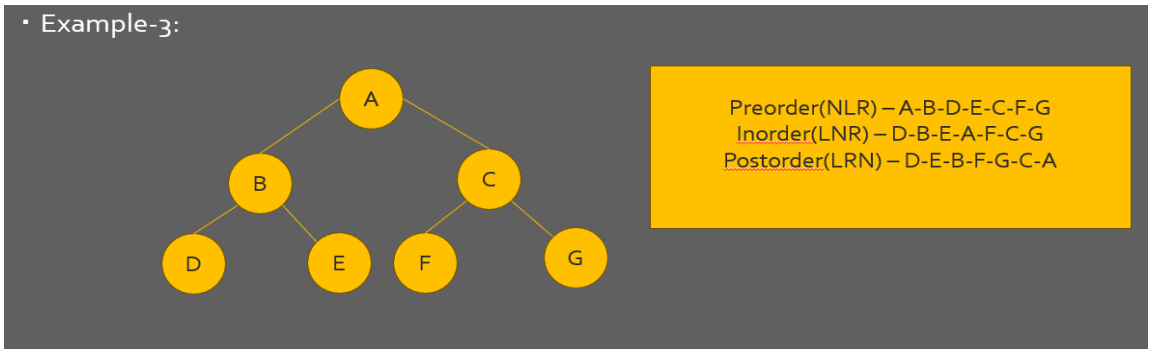
In-order Traversal (LNR): Visit the left subtree, then the root, followed by the right subtree.

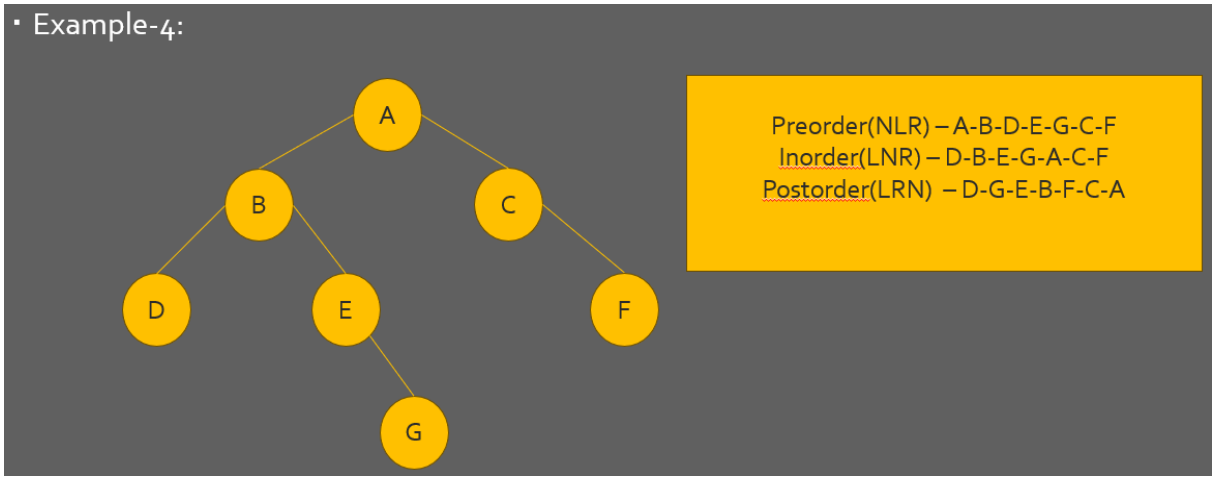
Order: 4, 2, 5, 1, 3, 6

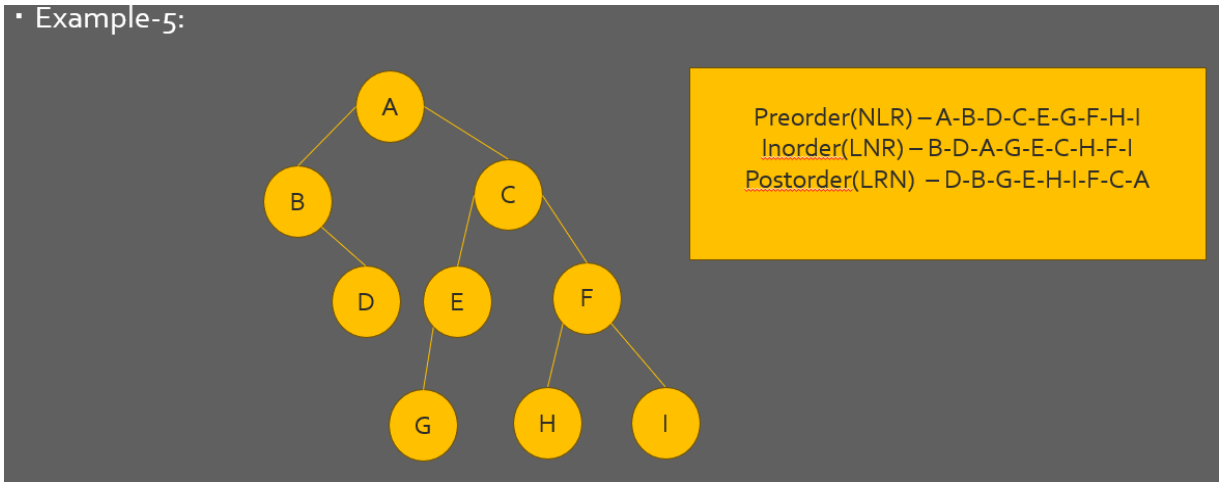
Post-order Traversal (LRN): Visit the left subtree, then the right subtree, followed by the root.

Order: 4, 5, 2, 6, 3, 1









Threaded Binary Tree

The threaded binary tree, the NULL links can be replaced by pointers called threads, to other nodes in the tree(which facilitate upward movement in the tree).

```
struct ThreadedNode
{
   int data;
   ThreadedNode* left;
   ThreadedNode* right;
   bool isLeftThread; // True if left pointer is a thread bool
   bool isRightThread; // True if right pointer is a thread
};
```

Types of Threaded Binary Trees

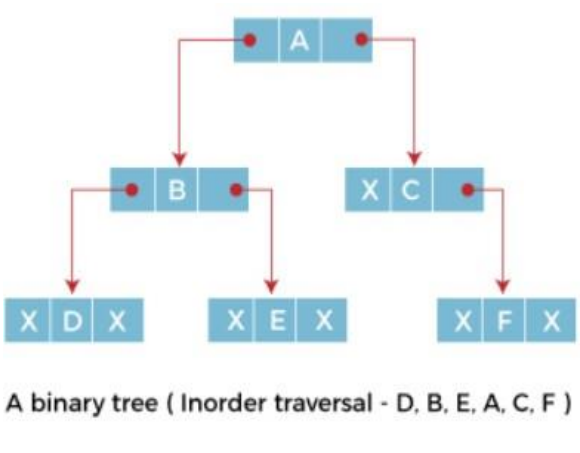
1. Single Threaded Binary Tree:

Each node's null pointer is replaced with a thread pointing to its inorder successor (right thread) or inorder predecessor (left thread), but not both.

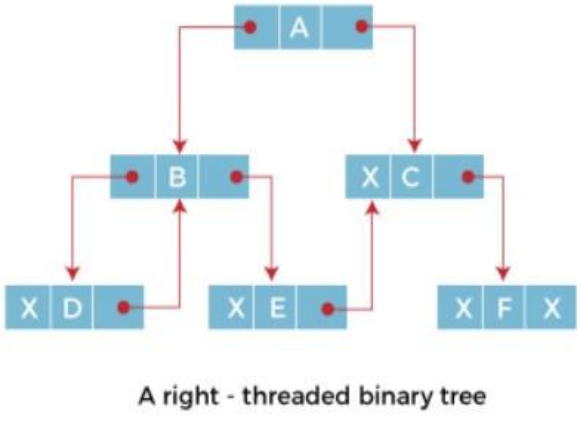
2. Fully Threaded (or Double Threaded) Binary Tree:

Each node has two threads, one pointing to its inorder predecessor and the other to its inorder successor.

One-way threaded Binary trees / Single threaded

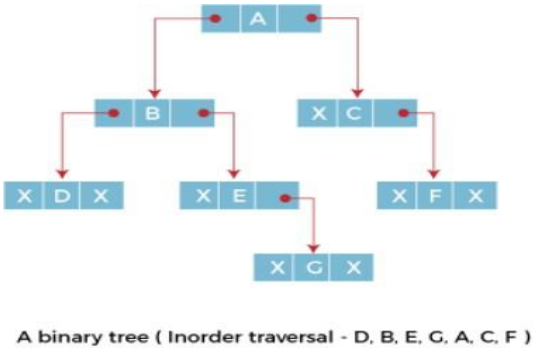


A binary tree (Inorder traversal - D, B, E, A, C, F)

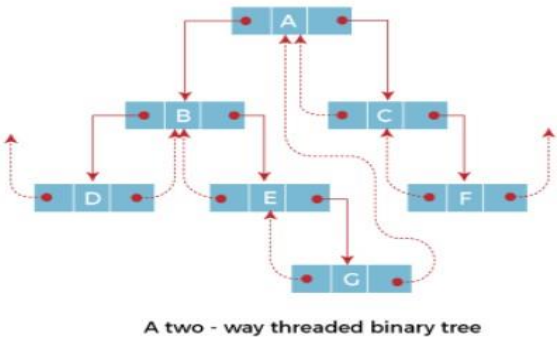


A right - threaded binary tree

The figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

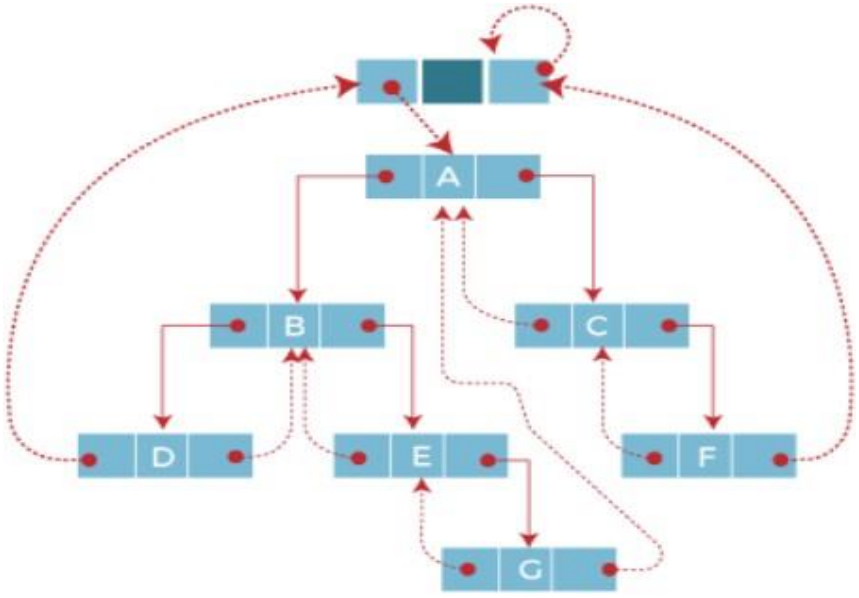


A binary tree (Inorder traversal - D, B, E, G, A, C, F)



A two - way threaded binary tree

The figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



Two-way threaded - tree with header node

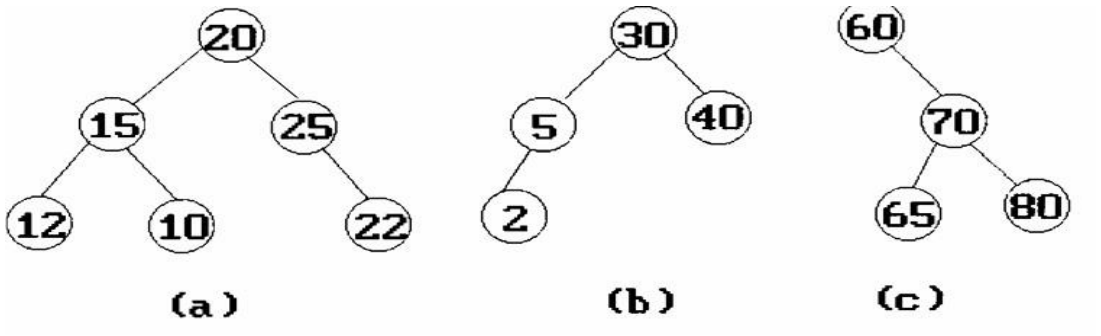
In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the header node.

Binary Search Tree

Definition: Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node. This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree.

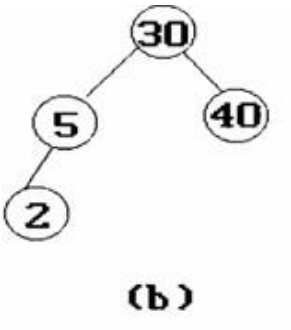
A binary search tree is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

1. Every element has a key, and no two elements have the same key, that is, the keys are unique.
2. The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
3. The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
4. The left and right subtrees are also binary search trees.



The tree of Figure (a) is not a binary search tree since the right subtree fails to satisfy property (4). This subtree has a root with a key value of 25 and a right child with a smaller key value (22). Figure (b) and Figure (c) are binary search trees.

Inserting Into A Binary Search Tree



To insert a new element, key, we must first verify that the key is different from those of existing elements. To do this we search the tree. If the search is unsuccessful, then we insert the element at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure (b), we first search the tree for 80. This search terminates unsuccessfully, and the last node examined has value 40. We insert the new element as the right child of this node. The resulting search tree is shown in Figure 5.31(a). Figure 5.31(b) shows the result of inserting the key 35 into the search tree of Figure 5.31(a).

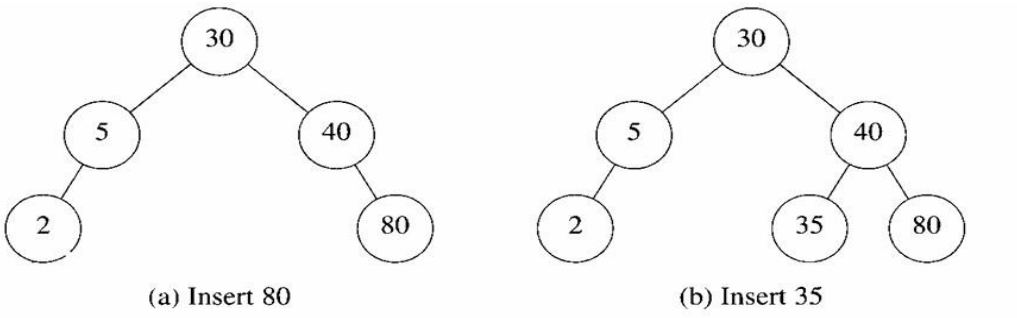


Figure 5.31: Inserting into a binary search tree

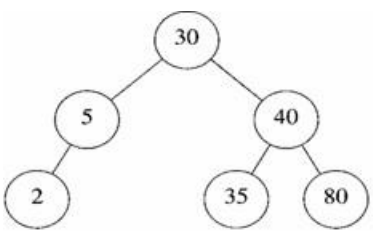
```
void insert_node(tree_pointer *node, int num)
/* If num is in the tree pointed at by node do nothing;
otherwise add a new node with data = num */
{
   tree_pointer ptr, temp = modified_search(*node, num);
   if (temp || !(*node)) {
      /* num is not in the tree */
      ptr = (tree_pointer)malloc(sizeof(node));
      if (IS_FULL(ptr)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      ptr->data = num;
      ptr->left_child = ptr->right_child = NULL;
      if (*node) /* insert as child of temp */
         if (num < temp->data) temp->left_child = ptr;
         else temp->right_child = ptr;
      else *node = ptr;
   }
}
```

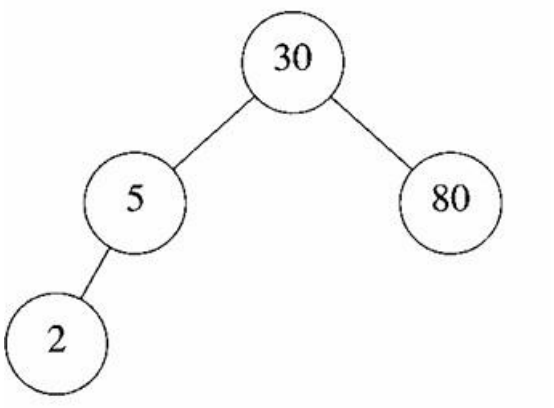
Analysis of insert-node: The time required to search the tree for num is $\Theta(h)$ where h is its height. Theremainder of the algorithm takes 0(1) time. So, the overall time needed by insert-node is O(h).

Deletion From a Binary Search Tree

Deletion of a leaf node is easy. For example, to delete 35 from the tree we set the left child field of its parent to NULL and free the node. The deletion of a nonleaf node that has only a single child is also easy. We erase the node and then place the single child in the place of the erased node.

For example, if we delete 40 from the tree we obtain the tree





suppose that we wish to delete 60 from the tree of Figure 5.33(a). We may replace 60 with either the largest element (55) in its left subtree or the smallest element (70) in its right subtree. Suppose we opt to replace it with the largest element in the left subtree. We move the 55 into the root of the subtree. We then make the left child of the node that previously contained the 55 the right child of the node containing 50, and we free the old node containing 55. Figure 5.33(b) shows the final result. One may verify that the largest and smallest elements in a subtree are always in a node of degree zero or one. This observation simplifies the code for the deletion function.

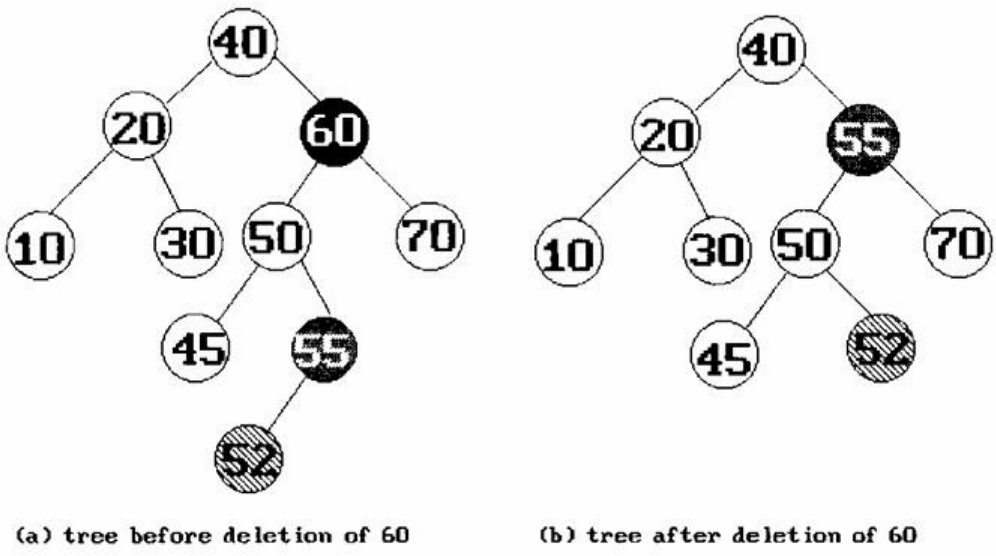


Figure 5.33: Deletion of a node with two children

Deletion can be performed in O(h) time where h is the height of the tree.

Searching A Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for an element with a key. We begin at the root. If the root is NULL, the search tree contains no elements and the search is unsuccessful. Otherwise, we compare key with the key value in root. If key equals root's key value, then the search terminates successfully. If key is less than root's key value, then no element in the right subtree can have a key value equal to key. Therefore, we search the left subtree of root. If key is larger than root's key value, we search the right subtree of root.

```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
      return search(root->left_child, key);
   return search(root->right_child,key);
}
```

We can easily replace the recursive search function with a comparable iterative one. The function search2 accomplishes this by replacing the recursion with a while loop.

```
tree_pointer search2(tree_pointer tree, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   while (tree) {
      if (key == tree->data) return tree;
      if (key < tree->data)
         tree = tree->left_child;
      else
         tree = tree->right_child;
      }
   return NULL;
}
```

Analysis of search and search!'. If h is the height of the binary search tree, then we can perform the search using either search or search! in 0(h). However, search has an additional stack space requirement which is O(h).

Traversal Binary Search Tree

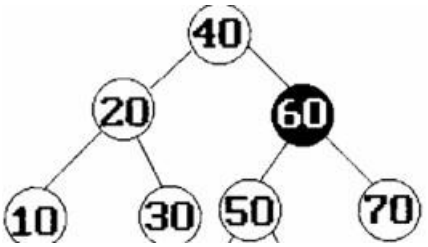
Binary Search Tree (BST) traversal involves visiting all nodes in the tree systematically. The main traversal methods for a BST are:

Inorder Traversal (Left, Root, Right)

Preorder Traversal (Root, Left, Right)

Postorder Traversal (Left, Right, Root)

Level Order Traversal (Breadth-First Traversal)



1. Inorder Traversal

In Inorder traversal, the nodes are visited in ascending order (left subtree, root, right subtree).

Steps:

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

Result: 10, 20, 30, 40, 50, 60, 70

2. Preorder Traversal

In Preorder traversal, the nodes are visited in this order: root, left subtree, right subtree.

Steps:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

Result: 40, 20, 10, 30, 60, 50, 70

3. Postorder Traversal

In Postorder traversal, the nodes are visited in this order: left subtree, right subtree, root.

Steps:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.

Result: 10, 30, 20, 50, 70, 60, 40

4. Level Order Traversal (Breadth-First)

In Level Order traversal, nodes are visited level by level from top to bottom, left to right.

Result: 40, 20, 60, 10, 30, 50, 70

Application of trees – Evaluation of Expression

What is an Expression Tree?

An Expression Tree is a binary tree where:

Internal nodes represent operators (like +, -, *, /).

Leaf nodes represent operands (like constants or variables).

Example Expression Consider the mathematical expression: (3 + 5) * (2 - 4)

Evaluation of the Expression Tree

The expression tree is evaluated by recursively evaluating the left and right subtrees and then applying the operator at the root.

Steps:

Start at the root *.

Evaluate the left subtree +:

Evaluate 3 and 5 and add them: 3 + 5 = 8.

Evaluate the right subtree -:

Evaluate 2 and 4 and subtract: 2 - 4 = -2.Finally,

multiply the results of the left and right subtrees: 8 * -2 = -16.

Infix, Prefix, and Postfix Notation

Expression trees can also be used to convert between different notations:

Infix (Original Expression): (3 + 5) * (2 - 4)

Prefix (Polish Notation): * + 3 5 - 2 4

Postfix (Reverse Polish Notation): 3 5 + 2 4 - *