Chapter 8.1

Trees

Introduction

- A tree is a **nonlinear** data structure and represents **hierarchical** data. A tree data structure is a non-linear data structure because it does not store data in a sequential manner.
- It is a collection of nodes that are linked together in a hierarchical manner in multiple levels.
- The topmost node in the tree is known as a **root** node. Each node contains some data and the link or reference of other nodes that can be called children.
- Hence, a tree consists of nodes with parent-child relationship.

Introduction

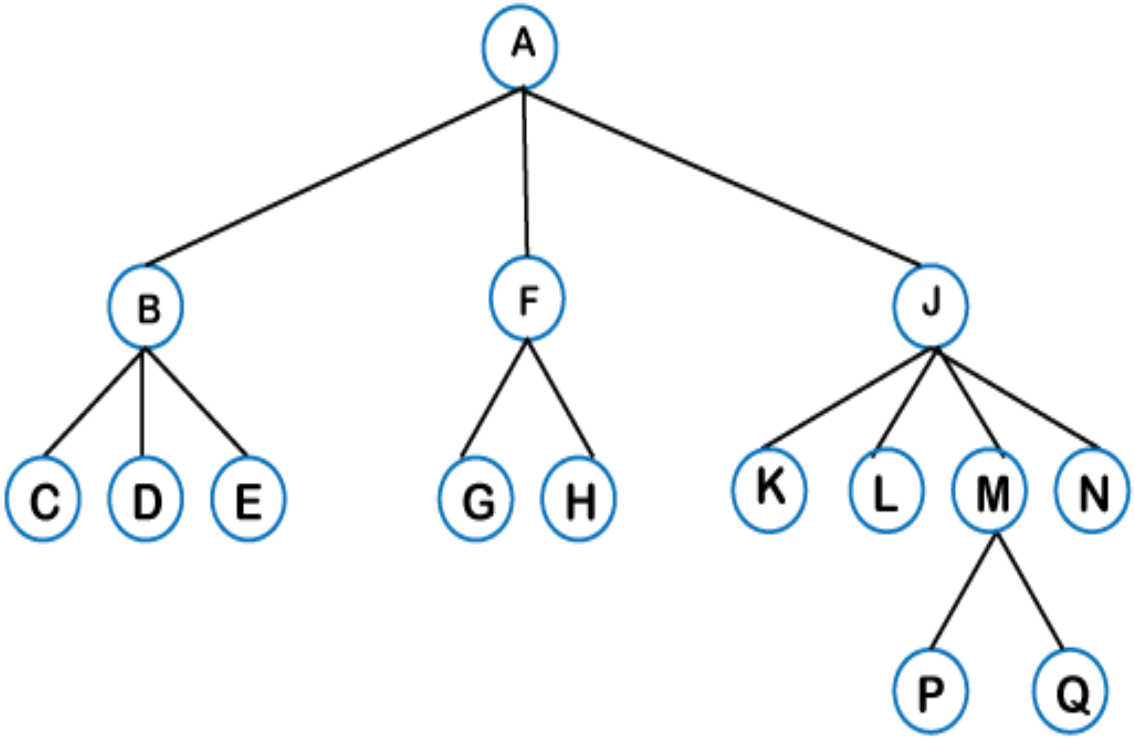


Fig: Tree

Introduction

- **Some Basic Terms:**
  - **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, the node with information **A** is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
  - **Parent (or father) :** The parent of any node $\mathbf{n_2}$ other than the root node is the unique node $\mathbf{n_1}$ such that there is a link from $\mathbf{n_1}$ to $\mathbf{n_2}$. And this parent is unique.
  - **Child (or son):** When node $\mathbf{n_1}$ is parent of node $\mathbf{n_2}$, $\mathbf{n_2}$ is called child of $\mathbf{n_1}$.
  - **Siblings (or brothers):** The nodes that have the same parent are known as siblings.

Introduction

- **Leaf Node:** A node that doesn't have any child node is called a leaf node. A leaf node is the bottom-most node of the tree. Leaf nodes can also be called **external nodes**.
- **Internal Node:** A node with at least one child node is an internal node. The root is an internal node unless it is the only node in the tree, in which case it is a leaf.
- **Ancestor node:** The **ancestors** of a node other than the root are the vertices in the path from the root to this vertex excluding the vertex itself and including the root.
- **Descendant node:** The **descendants** of a vertex v are those vertices that have v as an ancestor.
- **Level of a node:** The **level** of a node in a tree is the length of the unique path from the root to this node. The level of the root is defined to be zero.

Introduction

- **Depth of a tree:** The **depth** of a tree is the maximum of the levels of vertices. In other words, the depth of a tree is the length of the longest path from the root to any node.
- **Balanced tree:** A tree of depth **d** is **balanced** if all leaves are at level *d* or *d* − 1.
- **Subtree:** The **subtree** with a node **n** in a tree is the tree consisting of node **n** and its descendants.

Binary Tree

- Each node in the binary tree can have utmost two child nodes (left child and right child).
- Each node has both a left subtree and a right subtree (either of which may be empty). The node's **left** (**right**) **subtree** consists of the node's left (right) child together with that child's own children, grandchildren, etc.
- **Strictly Binary Tree:** If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a **strictly binary tree**. A strictly binary tree with n leaves always contains 2n - 1 nodes.
- **Complete Binary Tree:** A complete binary tree of depth d is a strictly binary tree all of whose leaves are at level d.

Binary Tree

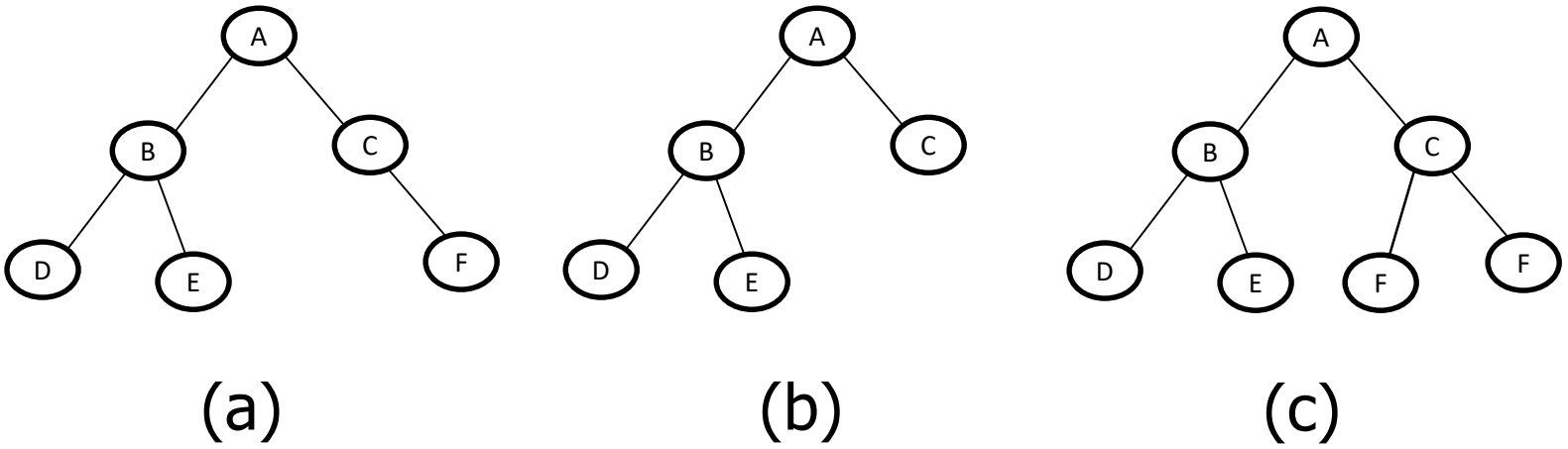


Fig: (a) Binary tree (b) Strictly binary tree (c) Complete binary tree

Binary Tree

- If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l+1.
- A binary tree can contain at most 2^l nodes at level l.
- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l nodes at each level between 0 and d. That is, the binary tree of depth d contains exactly 2^d nodes at level d.
- The total number of nodes in a complete binary tree of depth d, tn, equals the sum of the number of nodes at each level between 0 and d. Thus

$$tn = 2^0 + 2^1 + \ldots + 2^d = \sum_{j=0}^{d} 2^j = 2^{d+1} - 1$$

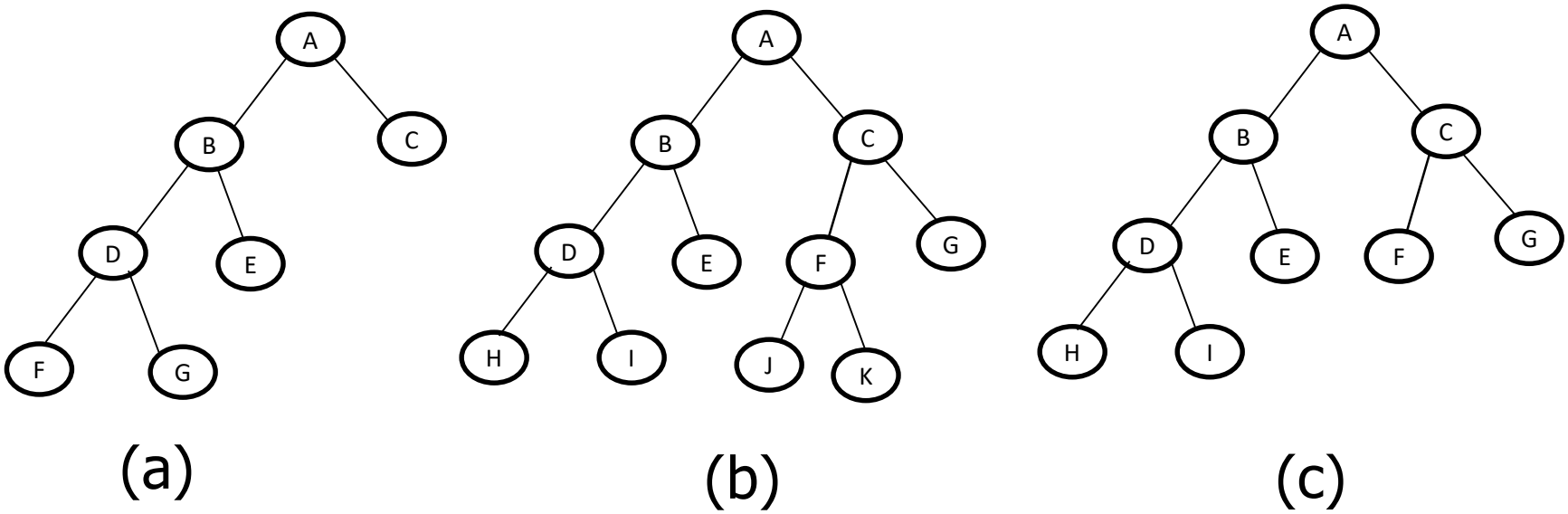
Binary Tree

- Since, all leaves in a complete binary tree are at level d, the tree contains 2^d leaves, and therefore, $2^d - 1$ nonleaf nodes.
- If the number of nodes, tn, in a complete binary tree is known, we can compute its depth

$$d = \log_2(tn + 1) - 1$$

- A **almost complete binary tree** is a binary tree in which (a) every level of the tree is completely filled except the last level. (b) Also, in the last level, nodes should be attached starting from left-most position.

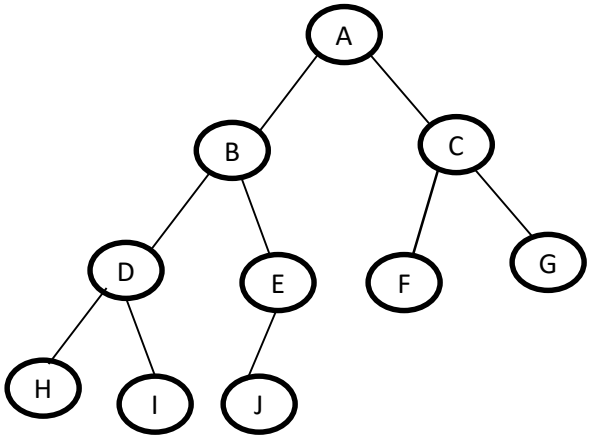
Binary Tree



(a) (b) (c)

The strictly binary tree in figure (a) is not almost complete, since it violates condition a. The strictly binary tree of (b) is not almost complete since it satisfies condition a but violates condition b. The strictly binary tree of (c) satisfies both conditions a and b and is therefore an almost complete binary tree.

Binary Tree



(d)

The binary tree of Figure (d) is also an almost complete binary tree but is not strictly binary

Binary Tree

- An almost complete strictly binary tree with n leaves has 2n-1 nodes, as does any other strictly binary tree with n leaves.
- An almost complete binary tree with n leaves that is not strictly binary has 2n nodes.
- There is only a single almost complete binary tree with n nodes. This tree is strictly binary if and only if n is odd. This tree is only almost complete binary tree but not strictly if an only if n is even.

Operations on Binary Trees

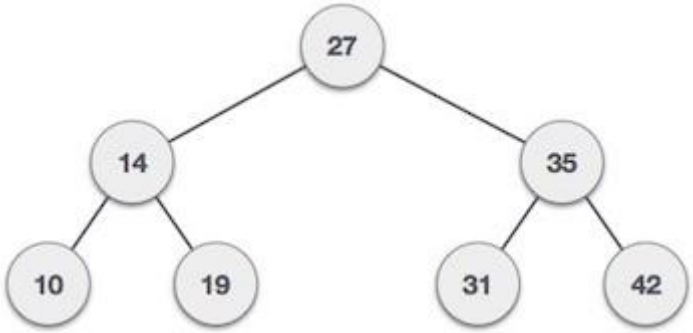
- There are a number of primitive operations on binary trees. If p is a pointer to a node nd of a binary tree, the function **info(p)** returns the content of nd.
- The functions left(p), right(p), father(p), and brother(p) return pointers to the left son of nd, the right son of nd, the father of nd, and the brother of nd, respectively. These functions return the null pointer if nd has no left son, right son, father, or brother.
- Finally, the logical functions isleft(p) and isright(p) return the value true if nd is a left or right son, respectively, of some other node in the tree, and false otherwise.

Operations on Binary Trees

- In constructing a binary tree, the operations maketree, setleft, and setright are useful.
- The **maketree(x)** creates a new binary tree consisting of a single node with information field x and returns a pointer to that node.
- The **setleft(p, x)** accepts a pointer p to a binary tree node with no left son. It creates a new left son of node(p) with information field x.
- The **setright(p, x)** is analogous to setleft except that it creates a right son of node(p).

Binary Search Tree

- A **binary search tree** (or **BST**) is a binary tree with the following property. For any node in the binary tree, if that node contains information *info*:
  - Its left subtree (if nonempty) contains only nodes with information less than *info*.
  - Its right subtree (if nonempty) contains only nodes with information greater than *info*.



Binary Search Tree

- **Search:** To search for a given target value in a binary search tree
  - Compare the target value with the element in the root node.
  - If the target value is **equal**, the search is successful.
  - If target value is **less**, search the left subtree.
  - If target value is **greater**, search the right subtree.
  - If the subtree is **empty**, the search is unsuccessful.
- **Insertion:** To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.

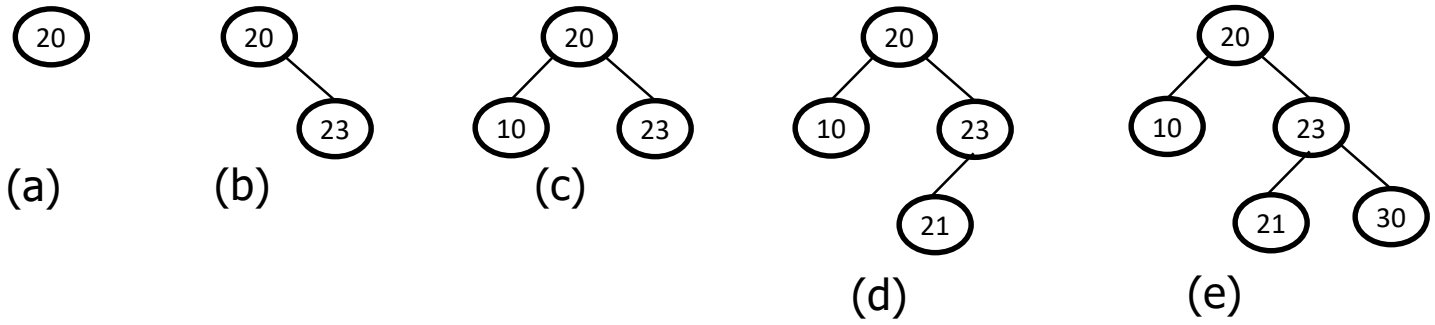
Binary Search Tree

- **Deletion:** To delete an element from a BST, proceed as if searching for that element.
  - If the node containing the element has no children (i.e., leaf node), replace the link to this node with NULL and remove the allocated space.
  - If the node containing the element has only one child, replace the link to this node with the address of its child node and remove the allocated space.
  - If the node containing the element has two children, copy the element of the leftmost node of its right subtree into the node, then delete the right subtree's leftmost node.

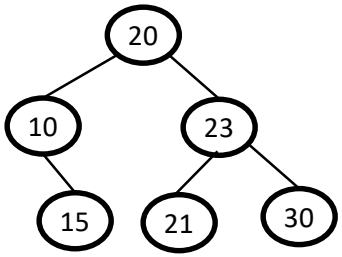
Binary Search Tree

Example: Starting with empty BST, show the effect of successively adding the following numbers as keys: 20, 23, 10, 21, 30, 15, 5, 22 and 40. Also, show the effect of successively deleting 10, 30, 20 from the resulting BST.

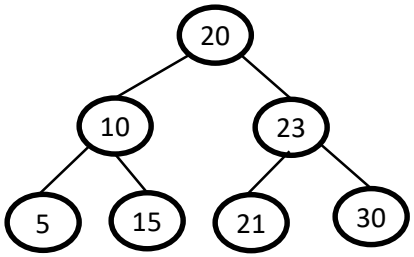
After successively adding 20, 23, 10, 21, 30, 15, 5, 22 and 40. 10.



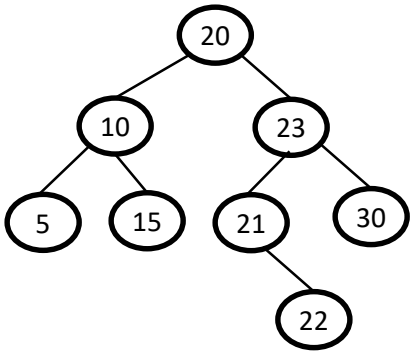
Binary Search Tree



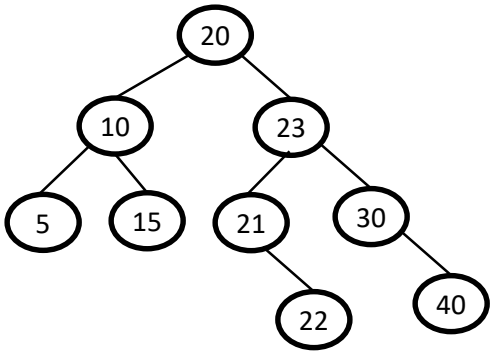
(f)



(g)

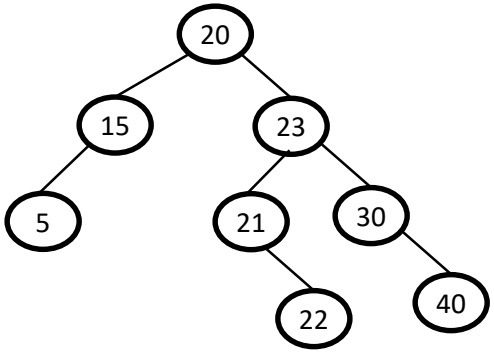


(h)



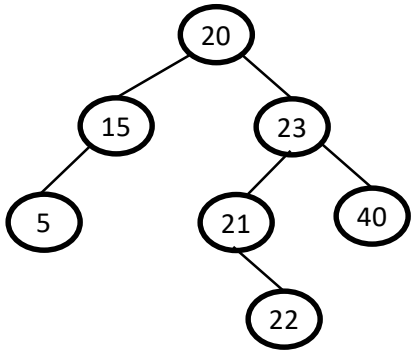
(h)

After deleting 10.

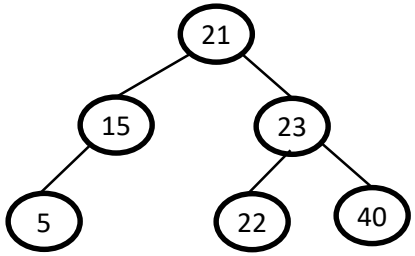


Binary Search Tree

After deleting 30.



After deleting 20.



Exercise: Consider a binary search tree (BST) whose elements are abbreviated names of chemical elements.

- Starting with an empty BST, show the effect of successively adding the following elements: H, C, N, O, Al, Si, Fe, Na, P, S, Ni, Ca.
- Show the effect of successively deleting Si, N, O, H from the resulting BST.

Binary Search Tree

- **Analysis:** Let number of nodes in the binary search tree be n.
  - If the tree is balanced, the maximum number of comparisons for search, insert, and delete operations is $\lfloor \log_2 n \rfloor + 1$. Hence, best-case time complexity is $O(\log_2 n)$.
  - If the tree is not balanced, maximum number of comparisons search, insert, and delete operations is n. Hence, worst-case time complexity is O(n).

Binary Tree Traversal

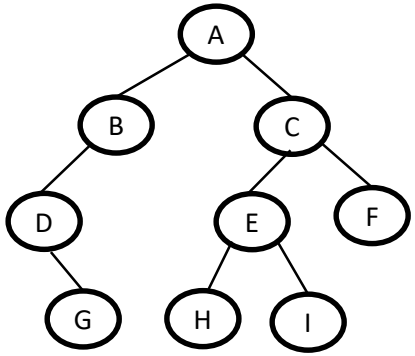
- Visiting each node of the binary tree is called binary tree traversal.
- Three traversal techniques: **preorder**, **inorder**, and **postorder**.
- **Preorder (or depth-first order) Traversal:** To traverse a non-empty binary tree in preorder, we perform following three operations.
  1. Visit the root.
  2. Traverse the left subtree in preorder.
  3. Traverse the right subtree in preorder.

Binary Tree Traversal

- **Inorder (or symmetric order) Traversal:** To traverse a non-empty binary tree in inorder, we perform following three operations.
  1. Traverse the left subtree in inorder.
  2. Visit the root.
  3. Traverse the right subtree in inorder.
- **Postorder Traversal:** To traverse a non-empty binary tree in postorder, we perform following three operations.
  1. Traverse the left subtree in postorder.
  2. Traverse the right subtree in postorder.
  3. Visit the root.

Binary Tree Traversal

Example: Traverse the binary tree below in preorder, inorder, and postorder.



Preorder (VLR): ABDGCEHIF

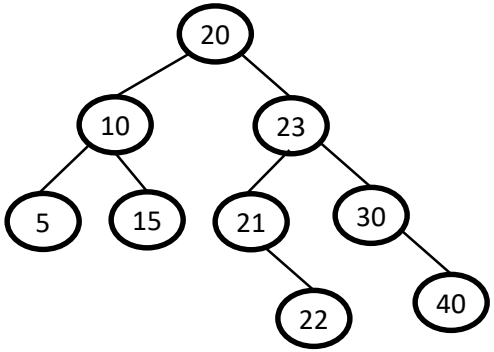
Inorder (LVR): DGBAHEICF

Postorder (LRV): GDBHIEFCA

Remember: The inorder traversal traverses a binary search tree in ascending order

Binary Tree Traversal

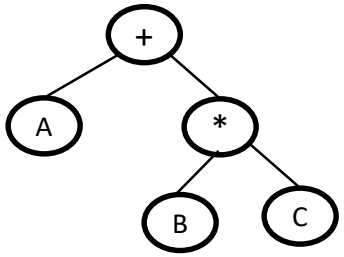
Exercise: Traverse the binary tree below in preorder, inorder, and postorder.



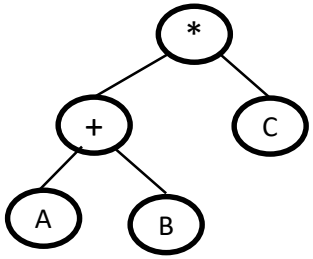
Applications of Binary Tree

- **Binary Search Tree:**
  - A binary search tree is a useful data structure when two-way decisions must be made at each point in a process. Such tree is called, **binary search tree**.
- **Binary Expression Tree:**
  - An expression containing operands and binary operators can be represented by a strictly binary tree called **binary expression tree**.
  - The root of this strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees. A node representing an operator is a nonleaf whereas a node representing an operand is a leaf.

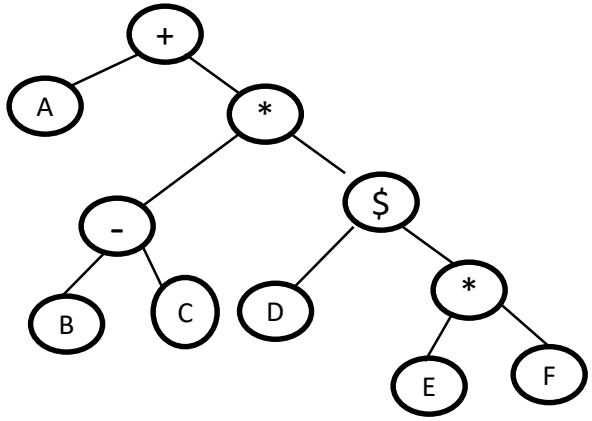
Applications of Binary Tree



A + B * C



(A + B) * C



A + (B – C) * D $ (E * F)

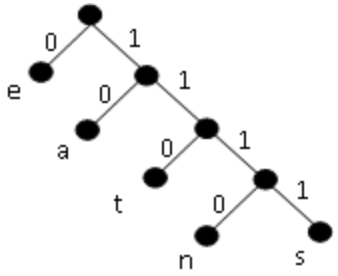
Remember: Traversing binary expression tree in preorder yields prefix form of the expression. Traversing such in postorder yields postfix form of the expression. Traversing such tree in inorder yields infix form of the expression except parenthesis.

Applications of Binary Tree

- **Data Compression:**
  - A binary tree is a useful data structure for **data compression**.
  - **Prefix codes** are the codes where no bit string corresponds to more than one sequence of letters. For example, the encoding of e as 0, a as 10, and t as 11 is a prefix code. A word can be recovered from the unique bit string that encodes its letters. For example, the string 10110 is the encoding of ate.
  - A prefix code can be represented using a binary tree, where the characters are the labels of the leaves in the tree.

Applications of Binary Tree

- The edges of the tree are labeled so that an edge leading to a left child is assigned to a 0 and an edge leading to a right child is assigned to a 1. The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label. For example, the tree below represents the encoding of e by 0, a by 10, t by 110, n by 1110, and s by 1111.



Applications of Binary Tree

- We can use this tree to decode a bit string. For example, the bit string 11111011100 is decoded to the word sane.
- We can also use this tree to encode characters with the bit string using the labels of the edges in the unique path from the root to the leaves. For example, the word eat is encoded the bit string 010110.
- Prefix codes are used to save memory and reduce transmission time because it can be used to represent data with fewer bits. The concept of representing data with fewer bits is called data compression.
- **Note:** ***Huffman coding***, a fundamental algorithm in *data compression*, uses the concept of prefix codes to compress data.