Data Structures and its Applications

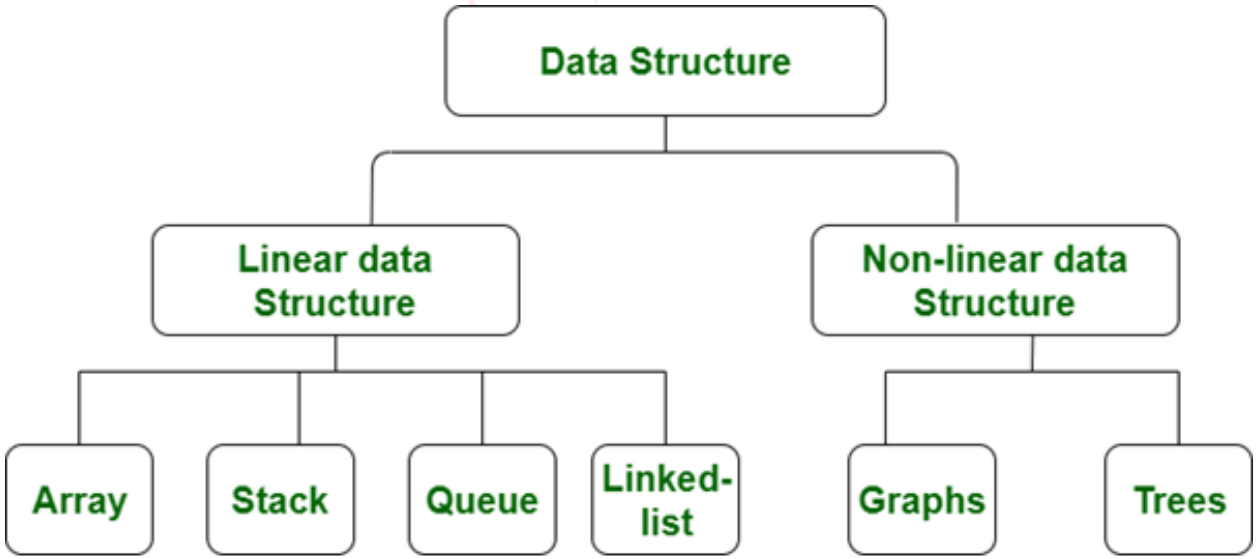
Dr. K. Sita Ramana
Associate Professor
Department of Mathematics,
Malla Reddy University, Hyderabad

June 19, 2023

I must extend my sincere thanks to Dr. P. Praveen Kumar and Dr. Sreenath Itikela for their assistance in gathering data, advice, and ideas during the creation of these slides.

Introduction

- Data structure is a way to store or organize data.
- Data organisation through data structure allows for effective use of the data. For instance, the clothes in our wardrobe can be arranged in a variety of ways depending on their colour, type, intended use, etc. Data can also be organised in multiple ways, hence we have several data structures.
- If data is arranged/accessed linearly then it is called linear data structure otherwise it is called non-linear data structure.

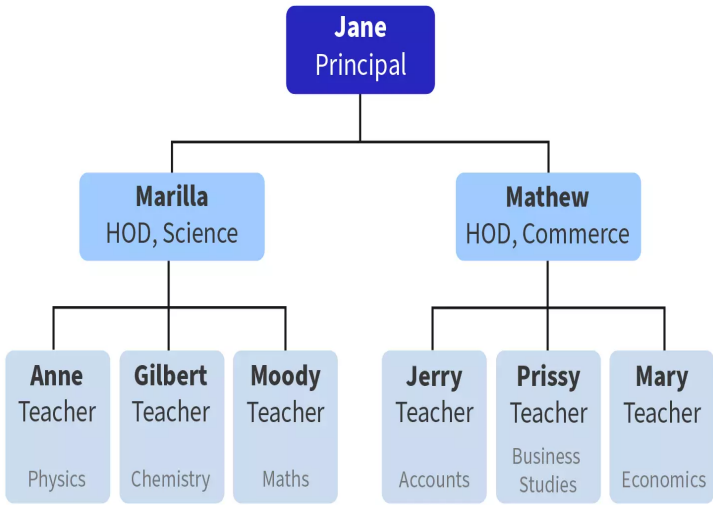


Introduction

- In linear data structure, data items are arranged sequentially and all data items can be traversed in a single run. But in non-linear data structures all data items may not be traversed during a single run as the data items are not organized sequentially.
- Implementation of linear data structures is easier when compared to non-linear data structures.
- In linear data structures, while traversing we attain one element at a time, hence we have only one level in it, but in non-linear data structures we have multiple levels.
- The different non-linear data structures are Tree, Graph.
- Non-linear data structures uses memory more efficiently than linear data structures.

Introduction

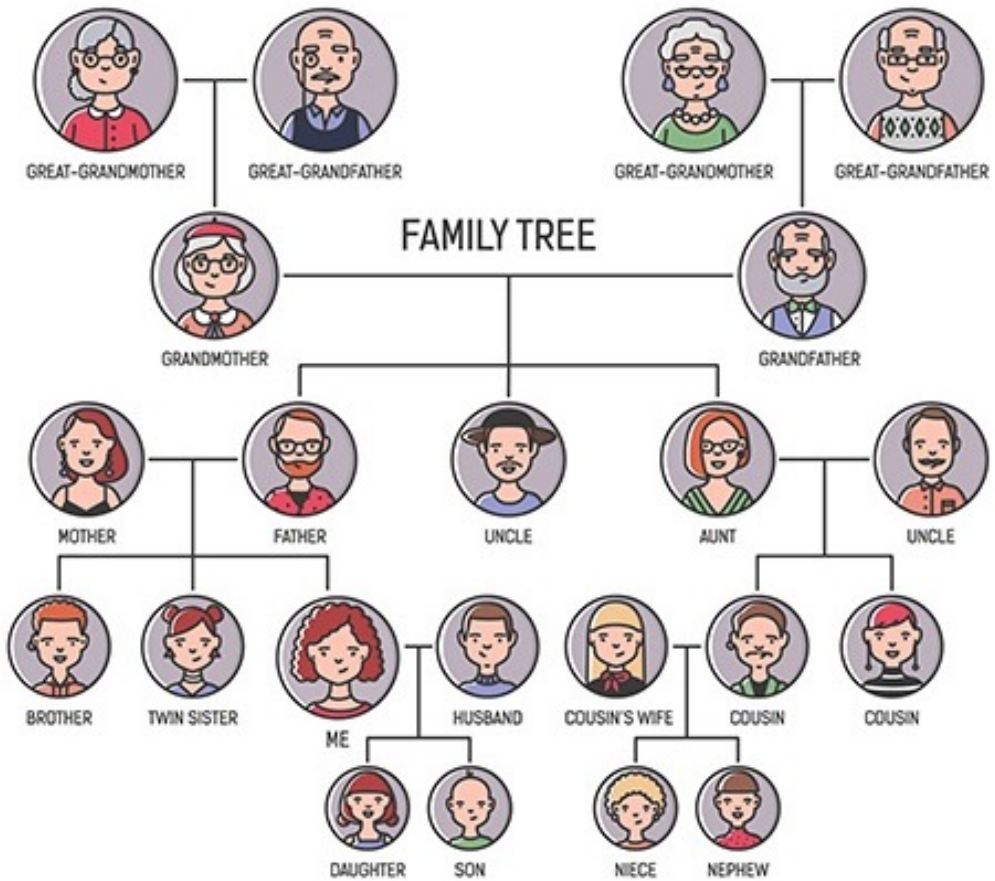
Member Name	Designation
Jane	Principal
Marilla	HOD, Science Dept
Anne	Teacher, Physics
Gilbert	Teacher, Chemistry
Moody	Teacher, Maths
Mathew	HOD, Commerce Dept
Jerry	Teacher, Accountancy
Prissy	Teacher, Business Studies
Mary	Teacher, Economics



An Example data to represent as both linear and non-linear data structures. (a) one-sided list that does not show who works under whom (b) Represented the relationship, so we will know the hierarchy

Introduction

The following picture shows a family tree, which is another Example for a Tree (non-linear data structure).



Tree

- **Trees** are non-linear data structures and represents relation between the nodes, which lie in it.
- A tree is used to represent hierarchical relationships such as parent-child relationship.
- Each node is connected to another node by directed edges.
- In tree, the data elements are arranged in the hierarchical form.
- Multiple runs are required to traverse through all the elements completely.
- Each element can have multiple paths to reach another element.
- Here elements are arranged in multiple levels, and each level can be filled completely or partially.
- In general, *It is a collection of entities called nodes that are connected by edges and has a hierarchical relationship between the nodes*.
- It is used to represent and organize data in a way that is easy to navigate and search.

Tree - Terminology

The basic terminology of a Tree is given below.

- **Node**: The individual element of a tree is called node.A node is any structure that holds data.
- **Root**: The topmost node of the tree from which we get all other nodes. A tree with an undistinguishable root node cannot be considered as a tree.
- **Edge**: - Link connecting two nodes.
- **Parent**: The node which has a branch to any other node is called a parent or parent node. In other words it is a node which has a child. Each node in a tree is a parent node.
- **Child**: Each node that is a descendant (successor node) of another node is called a child of that node. We can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.
- **Siblings**: Nodes with same parent

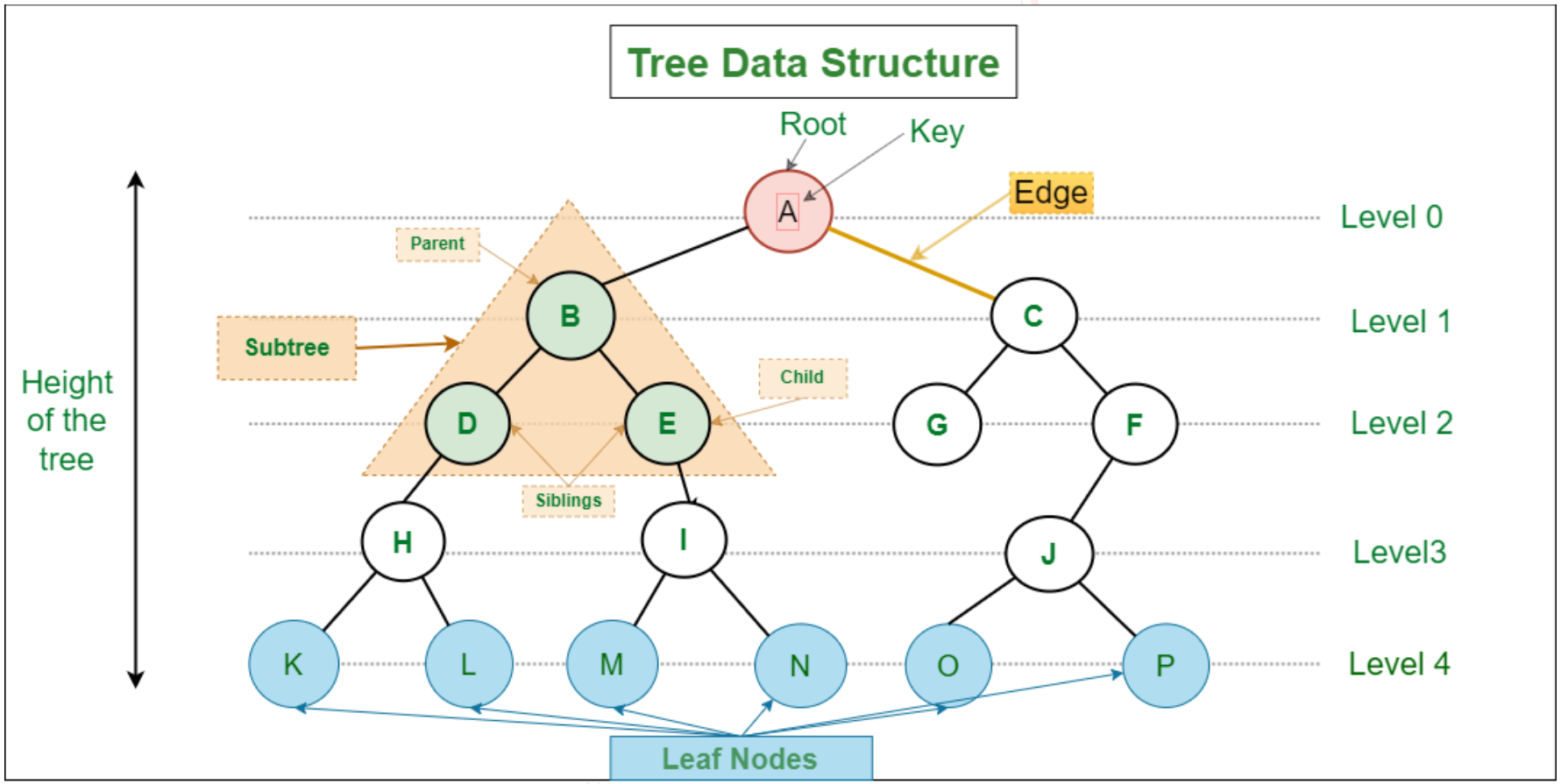
Tree - Terminology

- **Leaf**: The nodes which does't have child is called leaf or leaf node. In other words a node with zero degree is called Leaf or a node in the last/bottom-most level of the tree. It is also called external node or terminal node.
- **Internal Node**: Internal nodes are nodes that have one or more child nodes. They exist between the root and the leaf nodes. In a tree other than leafs, including root node is an internal node.
- **Path**: A path in a tree is a sequence of nodes and edges. It starts from the root and ends at a particular node in the tree.
- **Subtree**: Any node of the tree along with its descendant.
- **Degree of a Node**: The number of sub-trees of a given node or it is the number of its child nodes. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree. A leaf node has a degree of 0, while the maximum degree of a node in a tree is the number of children it can have.

Tree - Terminology

- **Degree of a Tree**: It is the highest degree of a node among all the nodes of a tree.
- **Level of a node**: The number of edges on the path from the root node to that node. The root node has level 0.
- **Height of a tree**: The number of levels in a tree. The height represents the length of the longest path from the root to a leaf node.
- **Height of a Node**: It is the total number of edges lies on a logest path from a leaf node to the node.
- **Depth**: The depth of a node is the number of edges from the root of the tree to that node.

Tree - Terminology



Tree - Characteristics

The hierarchical data structures known as trees are made up of nodes and edges. They have the following characteristics.

- **Root** : The root node of a tree is the node that is at the very top of the hierarchy. It is the starting point of tree's path, and it contain only one root node.
- **Nodes and Edges**: A tree consists of nodes, which hold data, and edges, which represent the connections between nodes. In a tree, if we have N nodes, then we will have exactly (N-1) edges in it.
- **Parent and Child Nodes**: Each node, except the root, has a parent node to which it is directly connected. Nodes that are directly connected to a parent node are called its child nodes. In a tree, every child will have only one parent, but a parent can have multiple child i.e., zero or more child.
- **Recursive data structure** : Tree consists trees i.e., a tree is a recursive data structure. It is composed of smaller trees, called sub-trees.

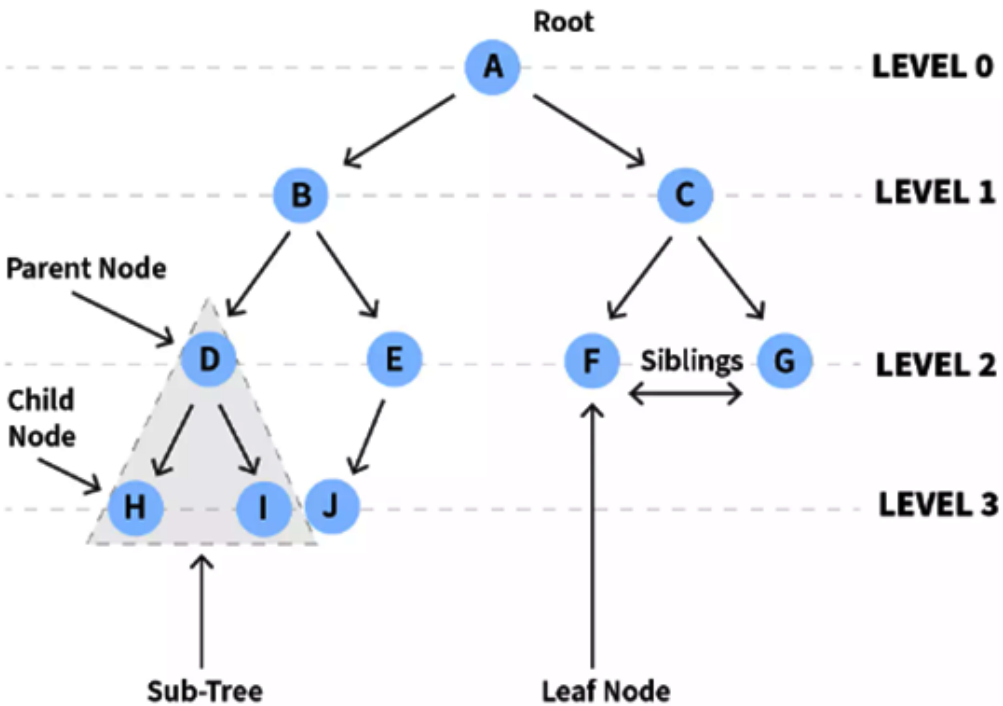
Types of Trees

Different types of trees available.

- **General Tree**:It is a tree, where each node can have any number of childs.
- **Binary Tree**: It is a tree, where each node can have at most two children. i.e., it can have either 0, or 1 or 2 children. In Binary tree there are several types, they are
  - Full Binary Tree
  - Complete Binary Tree
  - Perfect Binary Tree
  - Balanced Binary Tree
  - Pathological/degenerated Binary Tree
  - Skilled Binary Tree etc.,
- **Binary Search Tree**: It is a Binary Tree, with an condition on arrangement of nodes based on their key values.
- **AVL Tree**: It consists the properties of both Binary Tree and Binary Search Tree. This is invented by Adelson Velsky Linds (AVL). These trees are self balalnced i.e., height of Left Sub Tree and Right Sub Tree is balanced.
- **B-Tree**: It is a more generalized form of Binary Search Tree.

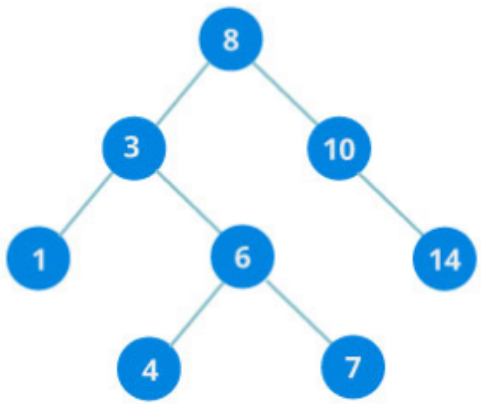
Binary Tree

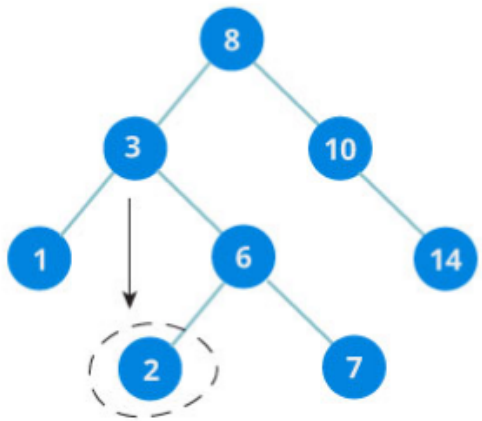
- A binary tree is a tree data structure where each node has at most 2 children. We commonly refer to them as the left and right child as each element in a binary tree can only have two children.
- A Binary tree is represented by a pointer to the topmost node (commonly known as the root) of the tree. If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains – Data, Pointer to the left child, Pointer to the right child.



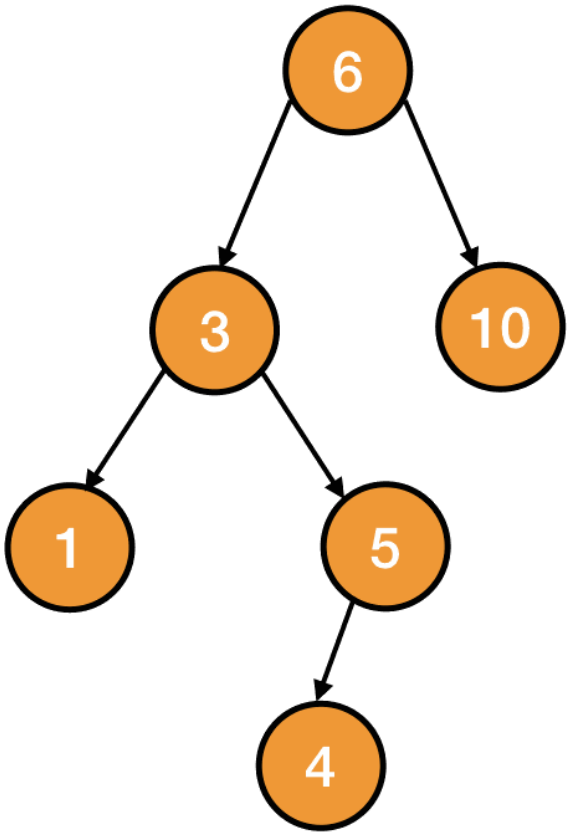
Binary Search Tree

- A binary search tree (BST) is a special kind of binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the nodes key
  - The right subtree of a node contains only nodes with keys greater than the nodes key.
  - The left and right subtree each must also be a binary search tree

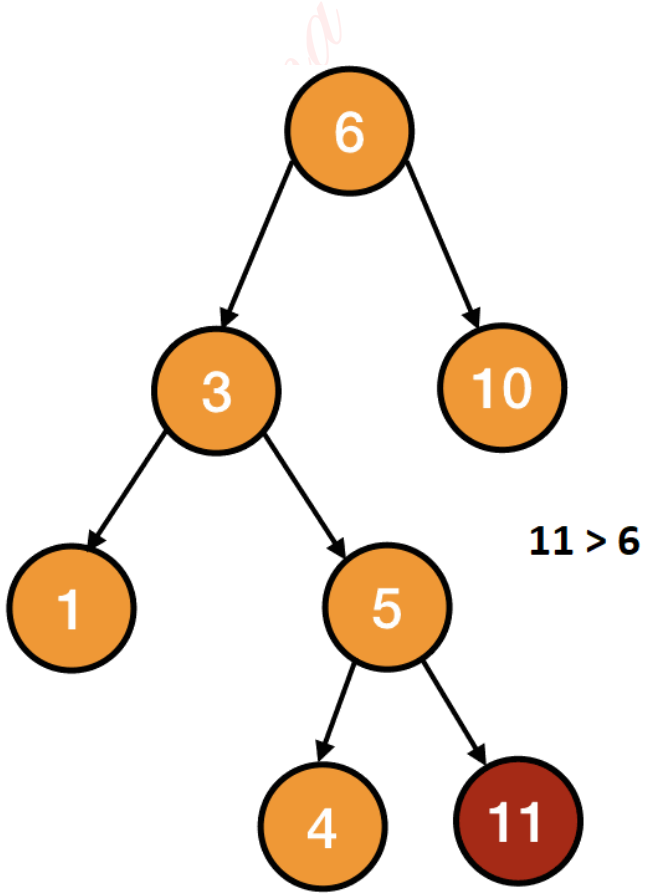




Binary Search Tree .. More Examples



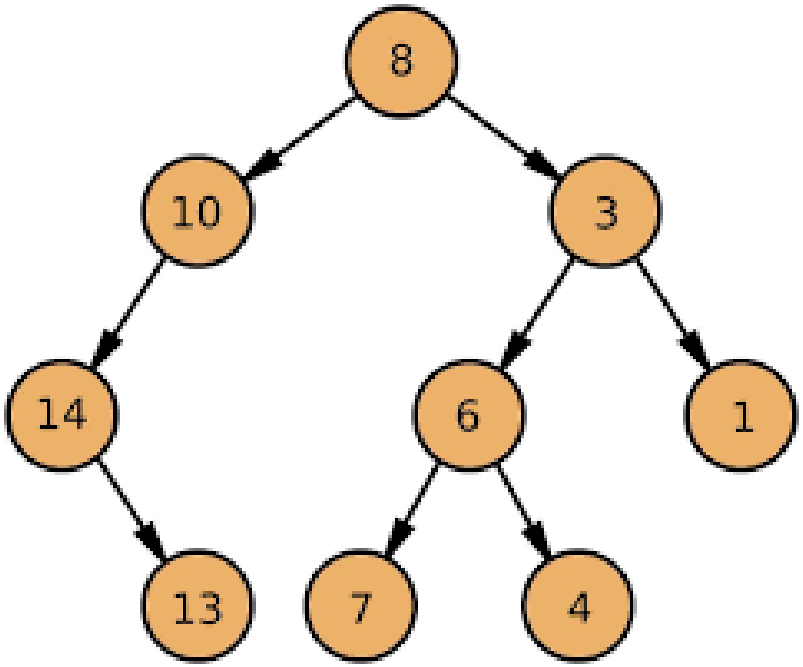
Binary Search Tree



Not a Binary Search Tree

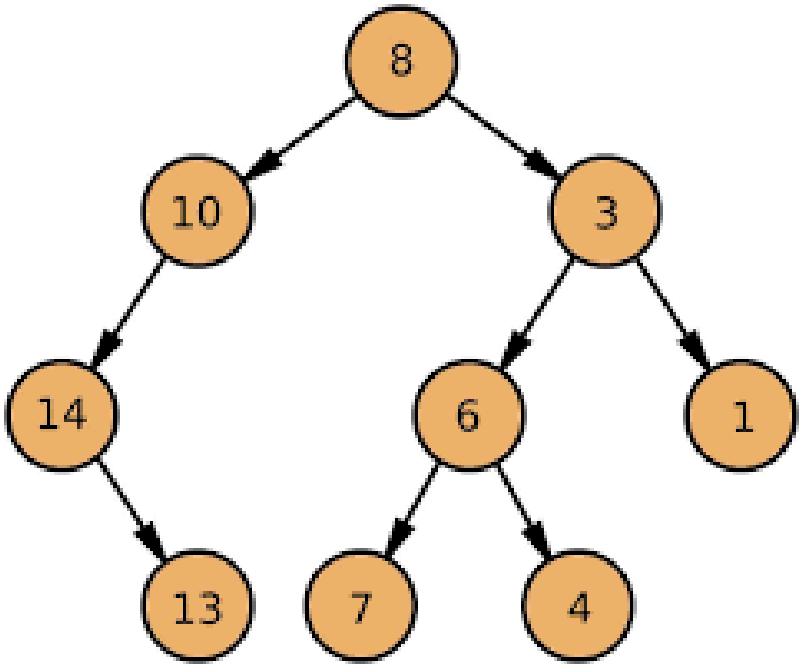
Binary Search Tree

Is the following tree, a Binary Search Tree.



Binary Search Tree

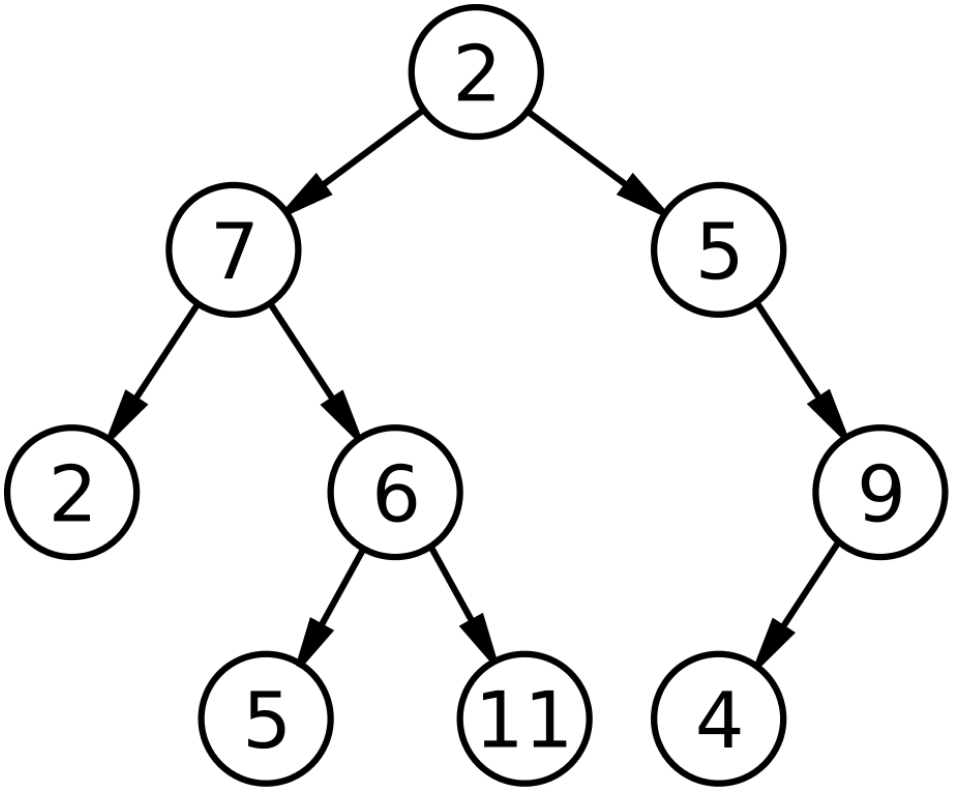
Is the following tree, a Binary Search Tree.



- ✓ No (because, $10 > 8$ and $3 < 8$ etc)

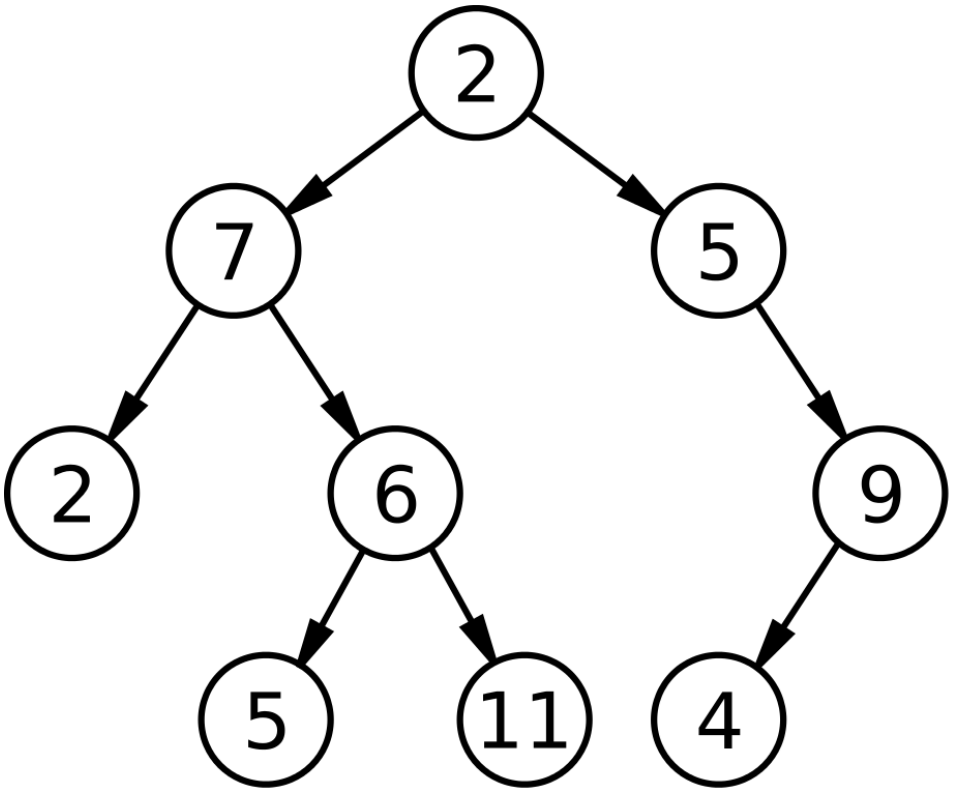
Binary Search Tree

Is the following tree, a Binary Search Tree.



Binary Search Tree

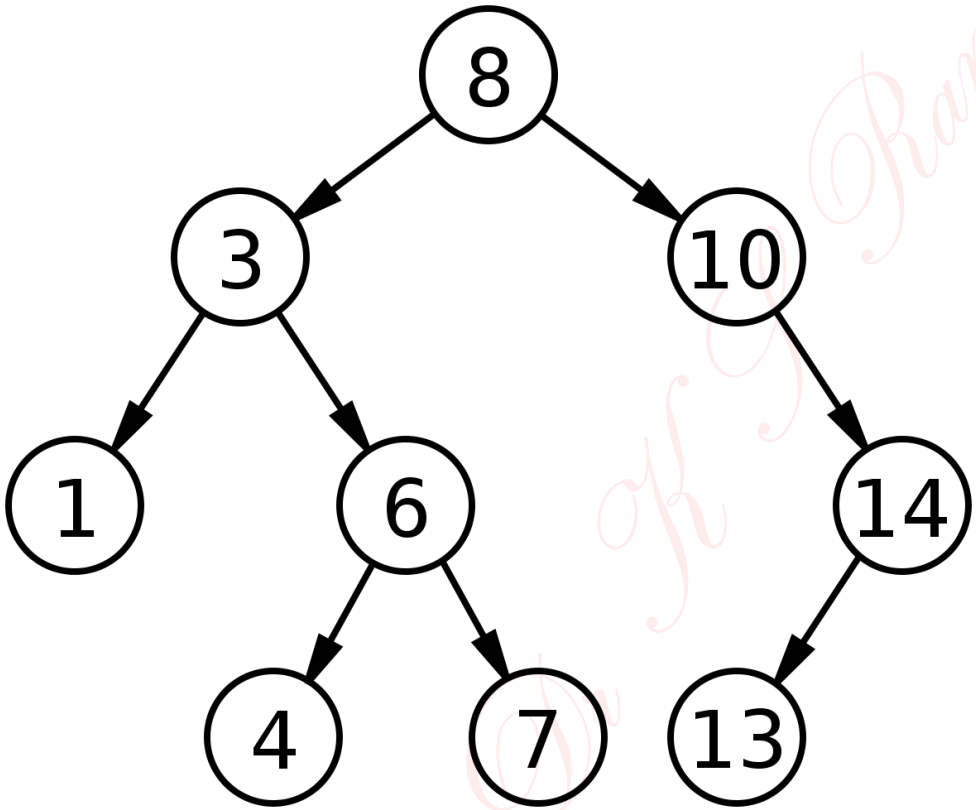
Is the following tree, a Binary Search Tree.



- ✓ No (because, $7, 5 > 2$ etc)

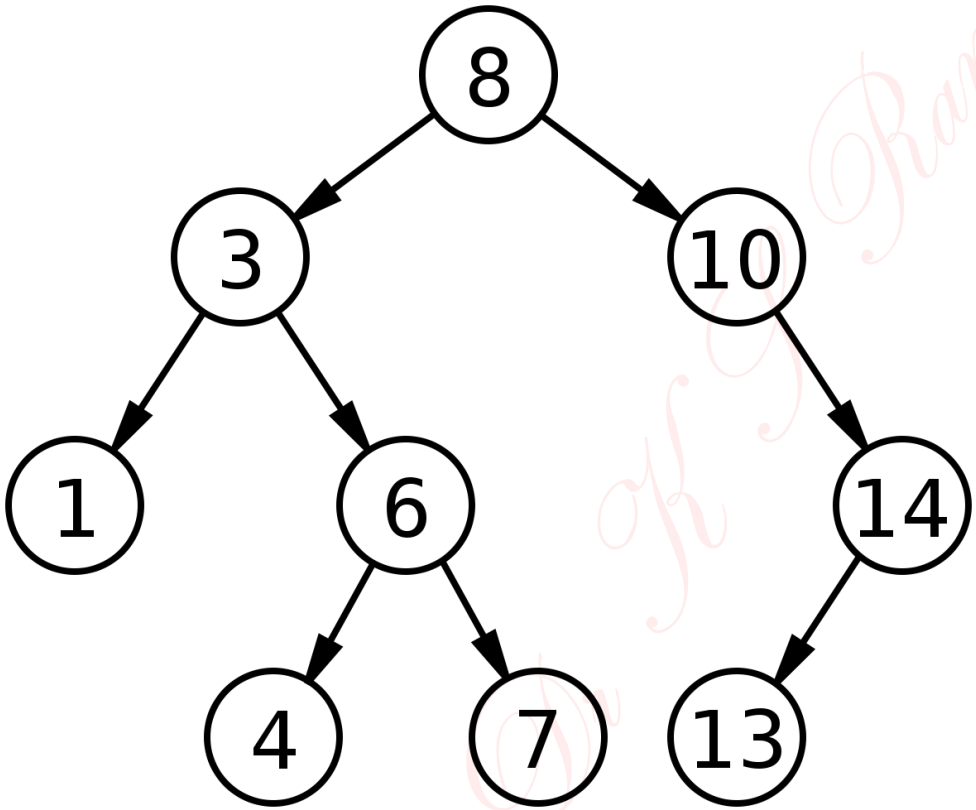
Binary Search Tree

Is the following tree, a Binary Search Tree.



Binary Search Tree

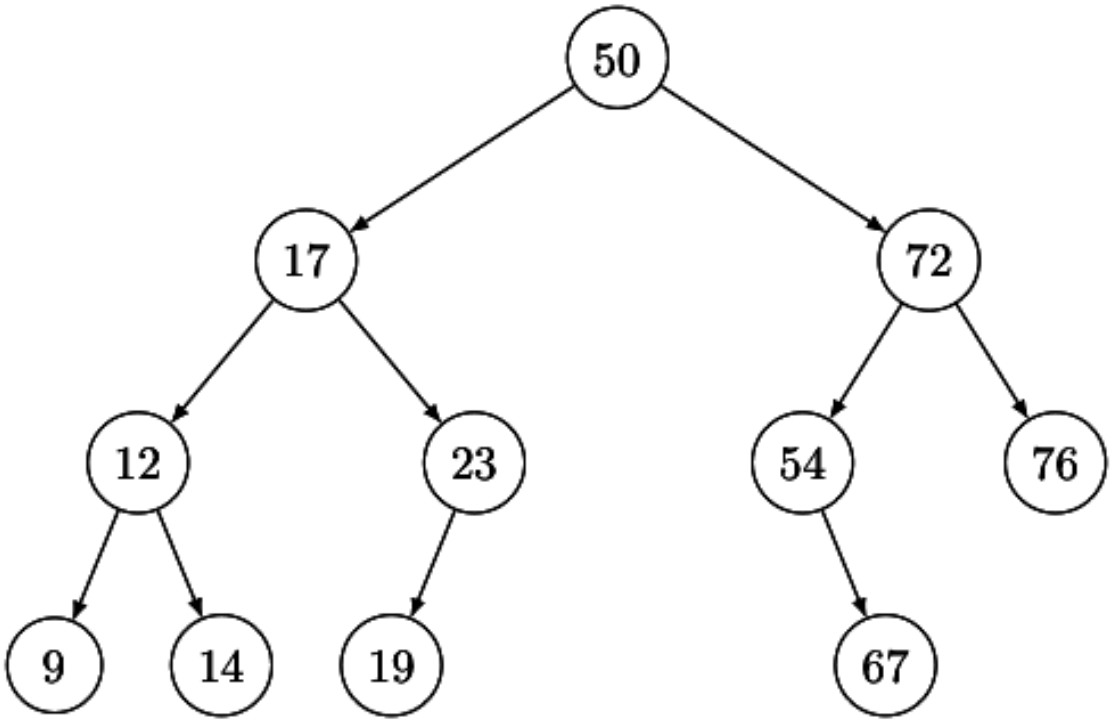
Is the following tree, a Binary Search Tree.



✓ Yes

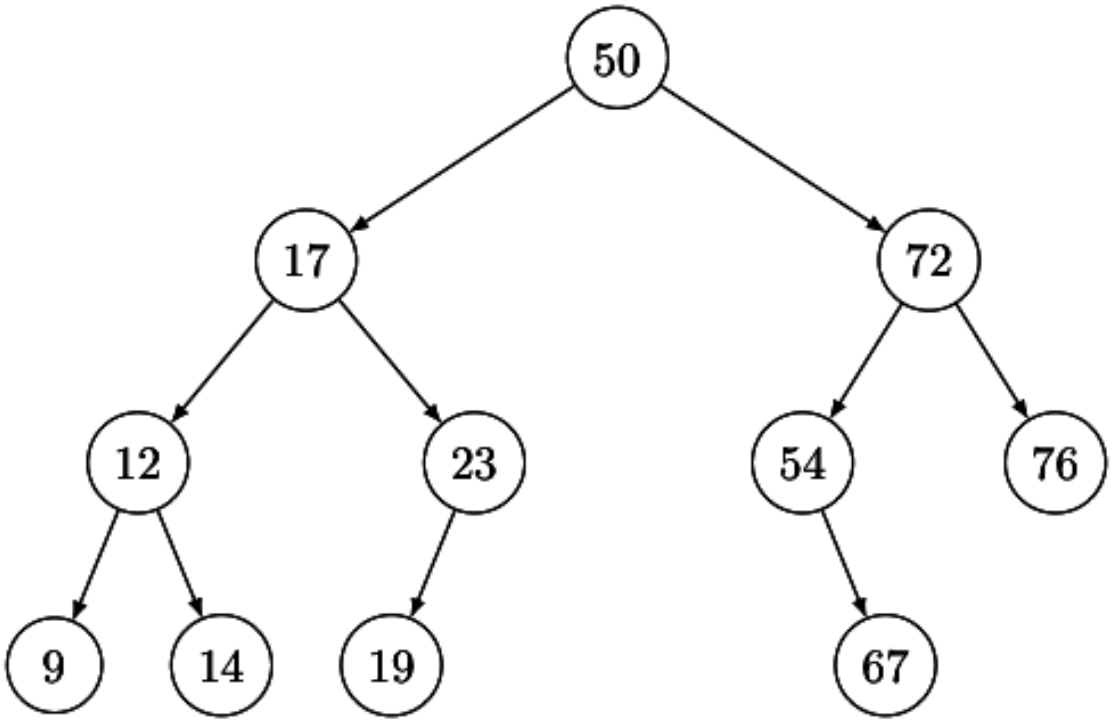
Binary Search Tree

Is the following tree, a Binary Search Tree.



Binary Search Tree

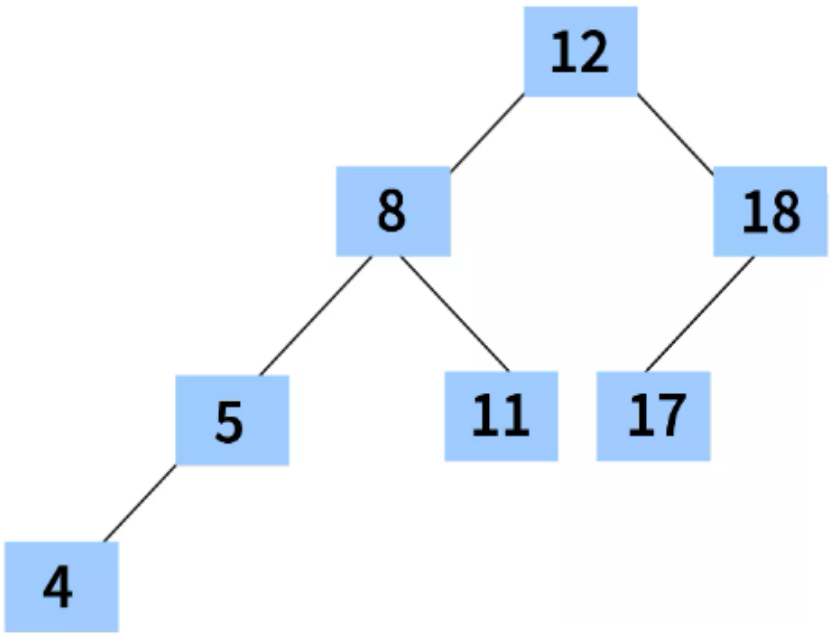
Is the following tree, a Binary Search Tree.



✓ Yes

AVL Tree

- AVL Treeis a special kind of Binary Search Tree where the height difference of the left and right subtrees is less than or equal to one.



Parent Node of Subtree	Height of left subtree	Height of right subtree	Height difference
12	3	2	1
8	2	1	1
18	1	0	1
5	1	0	1
11	0	0	0
17	0	0	0
4	0	0	0

As, we can see, the maximum height difference between any left and right subtree is one. Thus this tree is an AVL Tree

Operations

The basic operations to manipulate the data in a tree are given below.

- **Insertion**: Adding a new node to the binary tree. The insertion process typically involves finding the appropriate position based on the binary tree's ordering property and adding the node as a leaf node.
- **Deletion**: Removing a node from the binary tree. The deletion process can vary depending on the specific case, such as deleting a leaf node, a node with only one child, or a node with two children. It often requires rearranging the tree structure to maintain its properties.
- **Search**: Searching for a specific value or node in the binary tree. This operation involves traversing the tree, comparing values, and potentially moving left or right based on the comparison until the desired value or node is found or determined to be absent.

Operations

- **Traversal**: Visiting all nodes in the binary tree in a specific order. There are different traversal techniques, including
  - **Inorder Traversal**: Visit the left subtree, visit the root node, visit the right subtree.
  - **Preorder Traversal**: Visit the root node, visit the left subtree, visit the right subtree.
  - **Postorder Traversal**: Visit the left subtree, visit the right subtree, visit the root node.
  - **Level-order Traversal**: Visit nodes in each level from left to right, starting from the root and moving down the levels.
- **Height and Depth Calculation**: Determining the height or depth of the binary tree. The height of a tree is the maximum number of edges from the root to a leaf node, while the depth of a node is the number of edges from the root to that node.

Operations

- **Counting Nodes**: Counting the total number of nodes in the binary tree, including both internal nodes and leaf nodes.
- **Finding Minimum and Maximum**: Finding the minimum or maximum value/node in the binary tree. This operation typically involves traversing the tree in a specific order (e.g., inorder traversal for finding the minimum value in a binary search tree).
- **Checking Balanced Tree**: Verifying if the binary tree is balanced, meaning that the heights of the left and right subtrees of each node differ by at most 1.
- **Constructing a Binary Tree**: Creating a binary tree from a given set of values or data. This operation involves building the tree structure based on a specific algorithm or input sequence.

These are just a few common operations on binary trees. The choice of which operations to perform depends on the requirements of the application and the specific problem being solved using the binary tree data structure.

Tree Traversal-Inorder,Preorder,Postorder

There are different methods for visiting the nodes of a tree, specially Binary Tree or Binary Search Tree. These are named as Inorder, Pre-order, Postorder and Levelorder. Among them first three are popular.

Inorder Traversal:

- In inorder traversal, the left subtree is visited first, followed by the root node, and then the right subtree (Left-Root-Right).
- The order of visiting nodes is: left subtree $\rightarrow$ root node $\rightarrow$ right subtree.
- In a binary search tree, the nodes visited in inorder traversal would be in ascending order.
- In terms of code execution, inorder traversal is useful for retrieving the elements of a binary tree in sorted order.

Tree Traversal-Inorder,Preorder,Postorder

Preorder Traversal:

- In preorder traversal, the root node is visited first, followed by the left subtree, and then the right subtree (Root-Left-Right).
- The order of visiting nodes is: root node $\rightarrow$ left subtree $\rightarrow$ right subtree.
- Preorder traversal is commonly used to create a copy of a binary tree, as the root node is visited first.

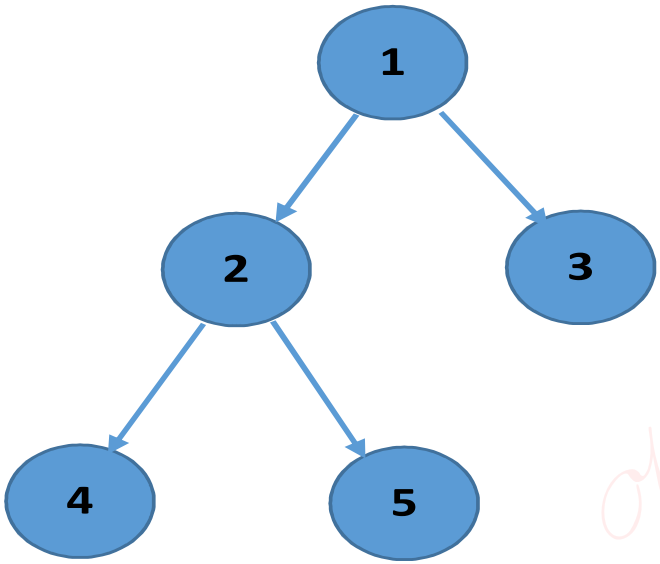
Postorder Traversal:

- In postorder traversal, the left subtree is visited first, followed by the right subtree, and then the root node (Left-Right-Root).
- The order of visiting nodes is: left subtree $\rightarrow$ right subtree $\rightarrow$ root node.
- Postorder traversal is often used in deleting or freeing nodes of a binary tree, as the root node is visited last.

Levelorder Traversal:

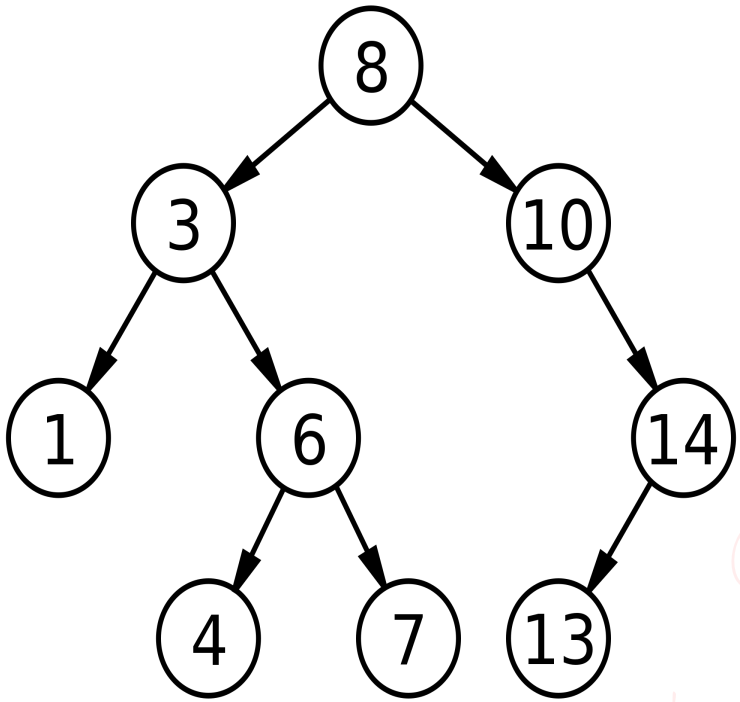
- In level-order traversal, we start from the root node (level 0), then visit the nodes at level 1, followed by the nodes at level 2 and soon.

Examples



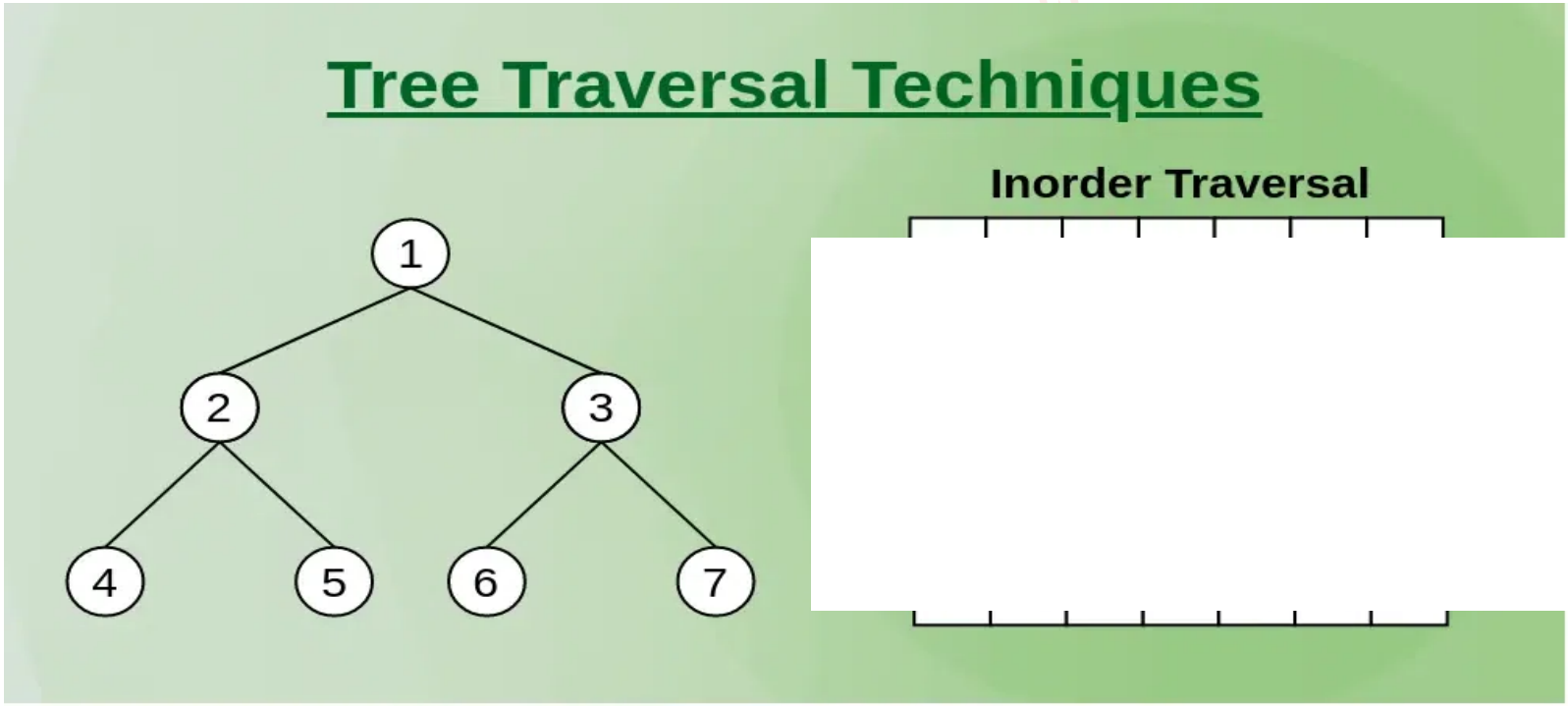
Binary Tree and its traversals

Examples



Binary Tree and its traversals

Examples



Binary Tree and its traversals

Binary Search Tree - Insertion

1. If the tree is empty, create a new node with the given element and make it the root of the tree.
2. If the tree is not empty, traverse the tree starting from the root.
3. At each node, compare the element to be inserted with the current node's value.
   - If the element is less than the current node's value, move to the left subtree.
   - If the element is greater than the current node's value, move to the right subtree.
4. Repeat step 3 until reaching a leaf node (a node with no children) or a node with a vacant child (left or right).
5. Create a new node with the given element and insert it as a child of the leaf node or the node with a vacant child.
   - If the element is less than the parent node's value, insert it as the left child.
   - If the element is greater than the parent node's value, insert it as the right child.

Binary Search Tree - Deletion

Deleting an element from a binary search tree involves several steps. The deletion process is a bit more complex than insertion due to various cases that need to be considered.

- **Case 1: Deleting a Leaf Node**: If the node to be deleted has no children (leaf node), simply remove the node from the tree
- **Case 2: Deleting a Node with One Child**: If the node to be deleted has only one child, replace the node with its child. Update the parent of the node to be deleted to point to its child.
- **Case 3: Deleting a Node with Two Children**: If the node to be deleted has two children, find the node's successor or predecessor (either the minimum node in the right subtree or the maximum node in the left subtree). Replace the node to be deleted with its successor/predecessor. Then, delete the successor/predecessor node from its original position.

Binary Search Tree - Deletion

- Start at the root and traverse the tree to find the node containing the element to be deleted. Keep track of the parent node as well.
- If the node is not found, the element does't exist. Return the tree as it is.
- If the node is found, consider the following cases:
  - If the node is a leaf node, simply remove the node from the tree by updating the parent node's reference to it (set it to None).
  - If the node has only one child, bypass the node by updating the parent node's reference to point directly to the child.
  - If the node has two children, then:
    - Find the in-order successor (or in-order predecessor) of the node to be deleted. This will be the node with the next smallest (or largest) value in the tree.
    - Replace the node's value with the in-order successor (or predecessor) value.
    - Delete the in-order successor (or predecessor) node from its original position using the above steps (recursive deletion).
- Return the updated tree.

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        # If the root is empty, create a new node and assign it as the root
        return Node(key)
    else:
        if key < root.key:
            # If the data is smaller than the current node, move to the
            # left subtree
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root
```

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```
def delete_node(root, key):
    if root is None:
        # Base case: empty tree
        return root
    if key < root.key:
        # If the key is smaller, move to the left subtree
        root.left = delete_node(root.left, key)
    elif key > root.key:
        # If the key is larger, move to the right subtree
        root.right = delete_node(root.right, key)
    else:
        # Found the node to be deleted
        # Case 1: Node with no children or only one child
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        # Case 2: Node with two children
        temp = find_min_value_node(root.right)
        root.key = temp.key
        root.right = delete_node(root.right, temp.key)
    return root
```

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```
def find_min_value_node(node):
    # Find the minimum value node in the subtree rooted at 'node'
    current = node
    while current.left is not None:
        current = current.left
    return current

def search(root, key):
    if root is None or root.key == key:
        # Base case: empty tree or target value found
        return root
    if key < root.key:
        # If the target value is smaller, move to the left subtree
        return search(root.left, key)
    else:
        # If the target value is larger, move to the right subtree
        return search(root.right, key)
```

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def preorder(root):
    if root:
        print(root.key, end=" ")
        preorder(root.left)
        preorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.key, end=" ")
```

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```
# Create an empty binary search tree
root = None

# Insert elements into the binary search tree
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

# Print the inorder traversal of the binary search tree
print("Inorder Traversal:")
inorder(root)
print()

# Print the preorder traversal of the binary search tree
print("Preorder Traversal:")
preorder(root)
print()

# Print the postorder traversal of the binary search tree
print("Postorder Traversal:")
postorder(root)
print()
```

Binary Search Tree - Insertion, Deletion, Searching and Inorder traversing operations - Complete Code

```python
# Search for an element in the binary search tree
search_key = 60
result = search(root, search_key)
if result:
    print(f"Element {search_key} found in the tree")
else:
    print(f"Element {search_key} not found in the tree")

# Delete an element from the binary search tree
delete_key = 30
root = delete_node(root, delete_key)
print(f"Deleted node with key {delete_key}")

# Print the tree after deletion
print("Tree after deletion:")
inorder(root)

#OUT PUT
Inorder Traversal:
20 30 40 50 60 70 80
Preorder Traversal:
50 30 20 40 70 60 80
Postorder Traversal:
20 40 30 60 80 70 50
Element 60 found in the tree
Deleted node with key 30
Tree after deletion:
20 40 50 60 70 80
```

Tree - Applications

- **Data Structures**: Trees are fundamental data structures used in computer science and programming. Examples include binary trees, binary search trees, AVL trees, B-trees, and heaps. Trees provide efficient storage and retrieval of data, enabling operations such as searching, insertion, deletion, and sorting.
- **File Systems**: File systems often use a tree structure to organize and manage files on a computer's storage devices. Directories and subdirectories form a hierarchical tree, allowing efficient navigation and storage of files.
- **Database Systems**: In database systems, trees are employed for indexing and searching data. B-trees and B+ trees are commonly used as index structures to facilitate fast access and retrieval of data in databases.
- **Compiler Design**: Trees play a crucial role in compiler design and parsing. Abstract Syntax Trees (ASTs) represent the structure of source code, aiding in syntax analysis, semantic analysis, optimization, and code generation.

Tree - Applications

- **Artificial Intelligence and Machine Learning**: Decision trees are popular in machine learning for classification and regression tasks. They provide interpretable models and can be combined to form more complex ensemble models like random forests and gradient boosting.
- **Graph Theory**: Trees are a special case of graphs without cycles. Graph algorithms often utilize trees as a foundation for solving problems such as spanning trees, shortest paths, and network optimization.
- **Hierarchical Representations**: Trees are used to represent hierarchical relationships in various applications. For instance, in organizational structures, family trees, XML/HTML document structures, and file dependency trees.
- **Network Routing**: Routing algorithms in computer networks use tree-based structures, such as spanning trees and multicast trees, to efficiently route data packets from a source to a destination.

Tree - Applications

- **User Interface Design**: Tree structures are used in the design of user interfaces, particularly in menu systems. Hierarchical menus and tree-like navigation structures enable users to explore and interact with software systems.
- **Evolutionary Biology**: Phylogenetic trees represent the evolutionary relationships between species, showcasing the branching patterns of species divergence over time. These trees aid in understanding evolutionary history, biodiversity, and genetic relationships.

These are just a few examples of the wide range of applications where trees are employed. The versatility and efficiency of tree structures make them invaluable in various fields of computer science, mathematics, and beyond.


thank you