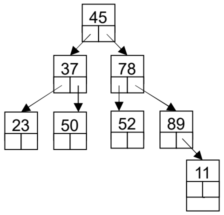
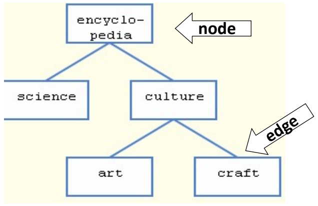
Tree Data Structure

Introduction

Introduction

- A tree is defined as a structure in which a set of *nodes* are connected together by their *edges*, in a parent-child relationship.
- Edges between the nodes represent the way the nodes are related.
- The only way to get from node to node is to follow a path along the edges.
- Linked lists, stacks and queues are linear (sequential) data structures. A tree is different. It is non-linear, or hierarchical.





Tree Data Structure Applications

- The tree structure is naturally suited to many applications:
  - Databases
  - File systems
  - Web sites
- They can often allow algorithms to be significantly more efficient e.g binary search.

Tree Example – File System

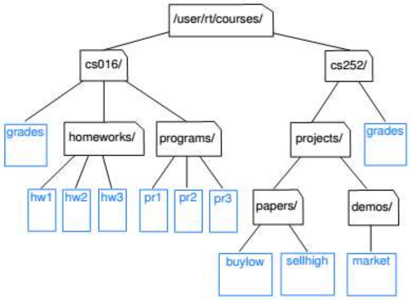
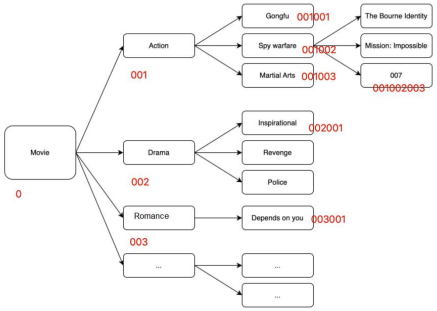


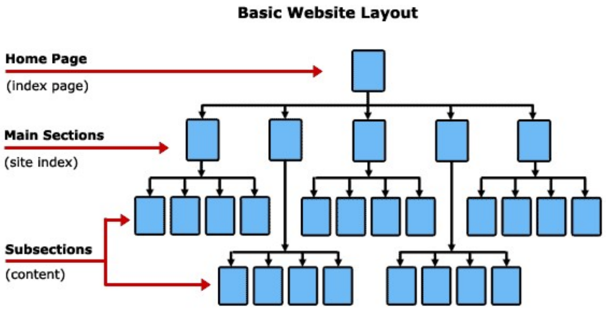
Figure 8.3: Tree representing a portion of a file system.

Tree Example - Database



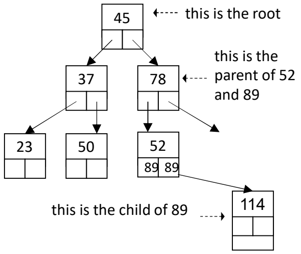
Example: MongoDB allows various ways to use tree data structures to model large hierarchical or nested data relationships.

Tree Example - Website



Tree Terminology

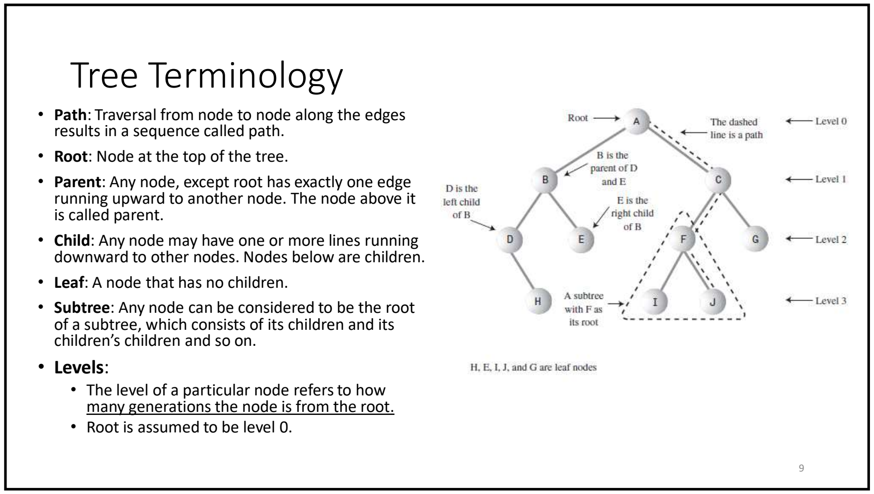
- A **rooted tree** is one where there is one distinguished node called the *root*, and every other node is connected to only one parent.
- **Binary tree**:
  - A **binary tree** is one in which each node has at most two children, each called the *left* and *right* child.
  - The diagram to the right is an example of a rooted binary tree.
  - We will only be concerned with rooted trees in this course.



this is the root

this is the parent of 52 and 89

this is the child of 89

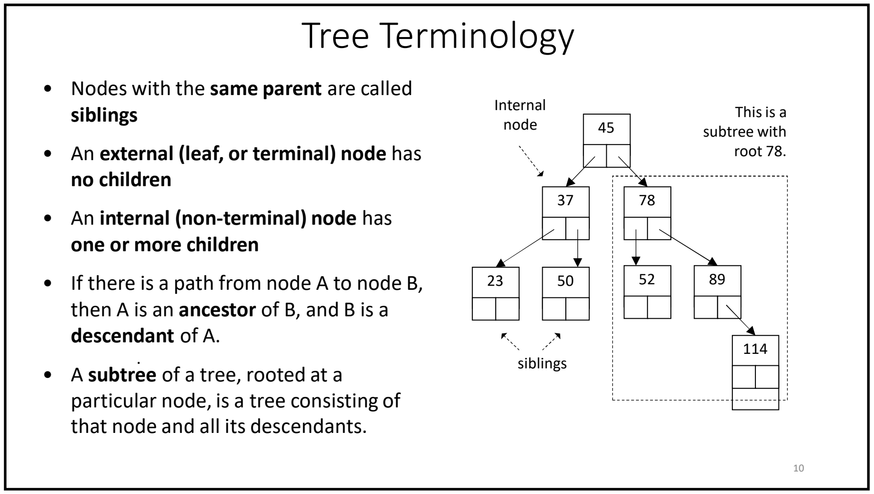


Tree Terminology

- **Path**: Traversal from node to node along the edges results in a sequence called path.
- **Root**: Node at the top of the tree.
- **Parent**: Any node, except root has exactly one edge running upward to another node. The node above it is called parent.
- **Child**: Any node may have one or more lines running downward to other nodes. Nodes below are children.
- **Leaf**: A node that has no children.
- **Subtree**: Any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.
- **Levels**:
  - The level of a particular node refers to how many generations the node is from the root.
  - Root is assumed to be level 0.

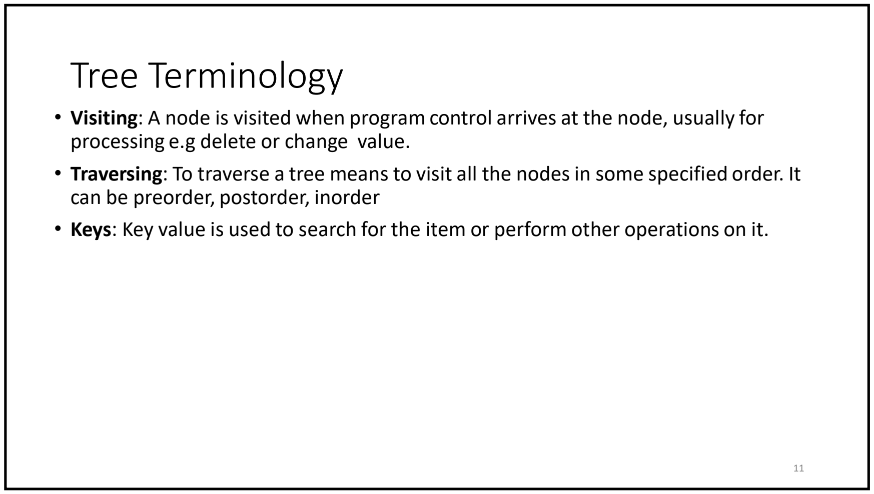
Root
A
The dashed line is a path
Level 0
B is the parent of D and E
B
C
Level 1
D is the left child of B
E is the right child of B
D
E
F
G
Level 2
H
A subtree with F as its root
I
J
Level 3

H, E, I, J, and G are leaf nodes



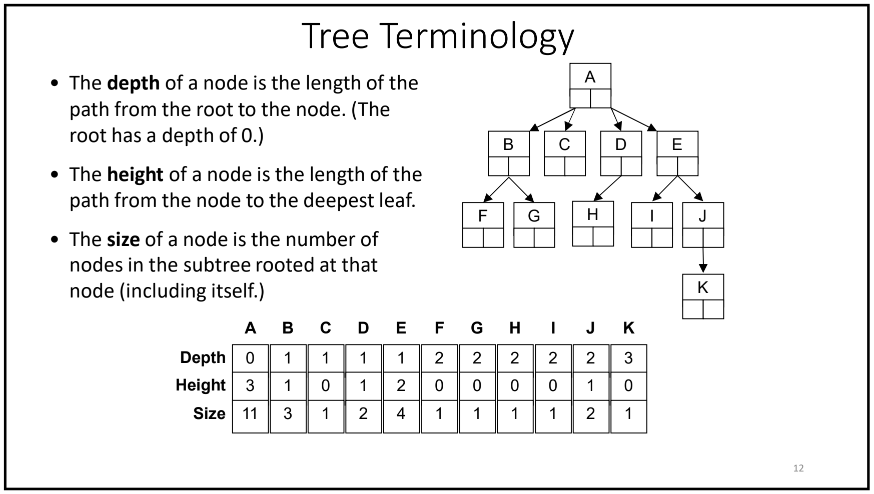
Tree Terminology

- Nodes with the **same parent** are called **siblings**
- An **external (leaf, or terminal) node** has **no children**
- An **internal (non-terminal) node** has **one or more children**
- If there is a path from node A to node B, then A is an **ancestor** of B, and B is a **descendant** of A.
- A **subtree** of a tree, rooted at a particular node, is a tree consisting of that node and all its descendants.



Tree Terminology

- **Visiting**: A node is visited when program control arrives at the node, usually for processing e.g delete or change value.
- **Traversing**: To traverse a tree means to visit all the nodes in some specified order. It can be preorder, postorder, inorder
- **Keys**: Key value is used to search for the item or perform other operations on it.



Tree Terminology

- The **depth** of a node is the length of the path from the root to the node. (The root has a depth of 0.)
- The **height** of a node is the length of the path from the node to the deepest leaf.
- The **size** of a node is the number of nodes in the subtree rooted at that node (including itself.)

	A	B	C	D	E	F	G	H	I	J	K
Depth	0	1	1	1	1	2	2	2	2	2	3
Height	3	1	0	1	2	0	0	0	0	1	0
Size	11	3	1	2	4	1	1	1	1	2	1

General Tree vs Binary Tree

- A general tree is a **data structure** in that each node can have infinite number of children.
- **Binary tree**:
  - Every node in a binary tree can have at most two children.
  - The two children of each node are called the left child and right child corresponding to their positions.
  - A node can have only a left child or only a right child or it can have no children at all.
- We are going to look at binary trees in this course.

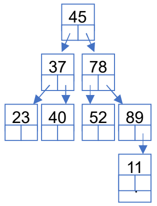
Binary Trees

Types of Binary Trees

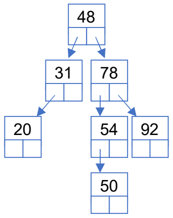
- The following are the types of binary trees:
  - Balanced binary tree
  - Unbalanced binary tree
  - Binary search tree

Balanced Binary Tree

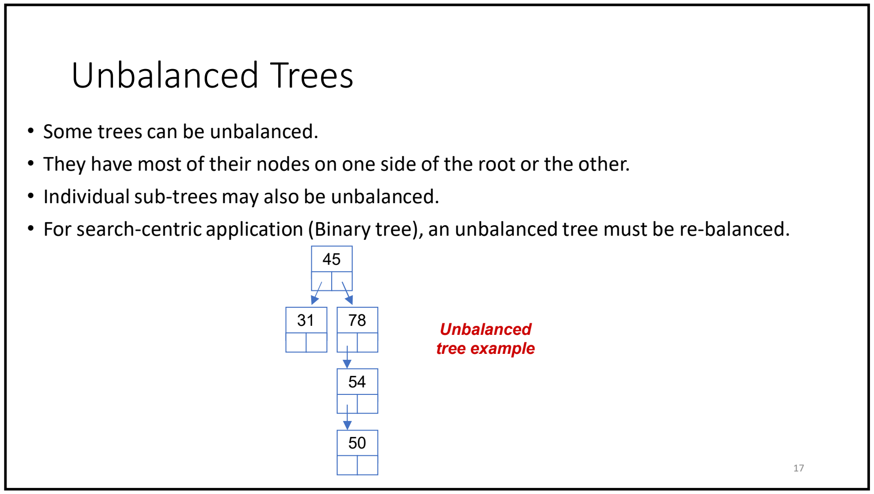
- A binary tree is balanced if and only if, for every node, the height of its left and right subtree differ by at most 1.
- A balanced binary tree is also known as an AVL tree, after its inventors Adelson, Velskii, and Landis.



Balanced tree



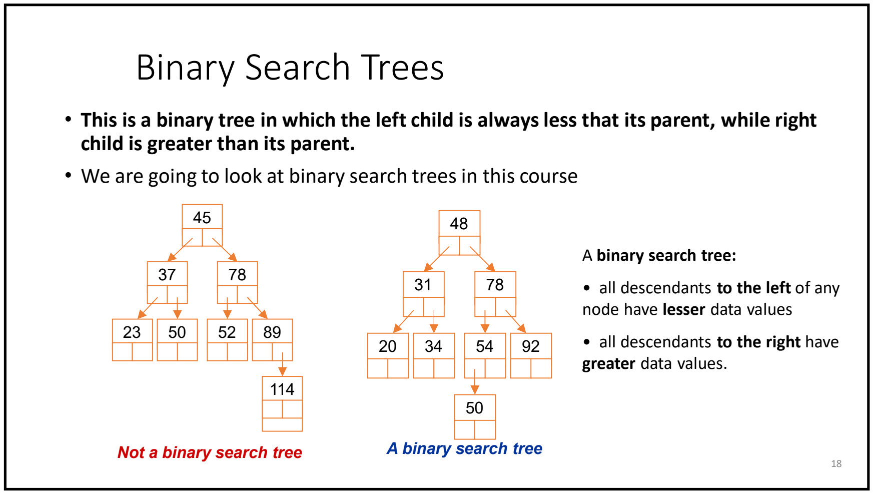
Balanced tree



Unbalanced Trees

- Some trees can be unbalanced.
- They have most of their nodes on one side of the root or the other.
- Individual sub-trees may also be unbalanced.
- For search-centric application (Binary tree), an unbalanced tree must be re-balanced.

Unbalanced tree example



Binary Search Trees

- **This is a binary tree in which the left child is always less that its parent, while right child is greater than its parent.**
- We are going to look at binary search trees in this course

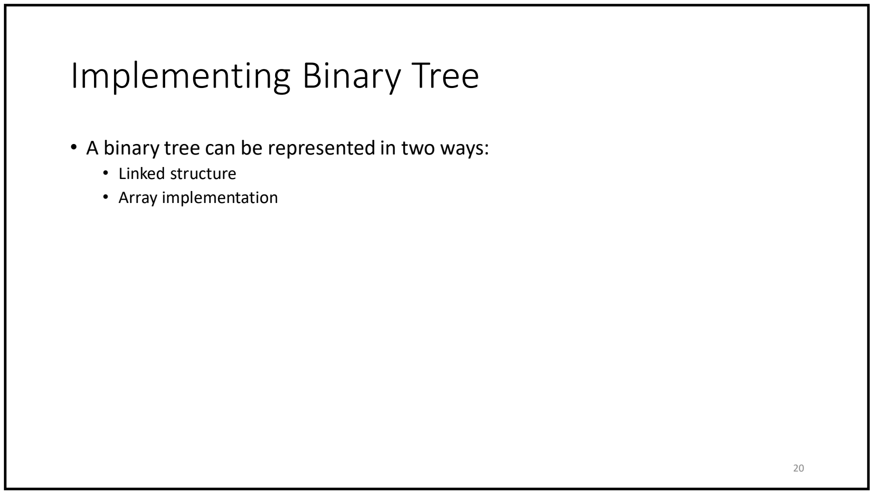
Not a binary search tree

A binary search tree

A **binary search tree:**

- all descendants **to the left** of any node have **lesser** data values
- all descendants **to the right** have **greater** data values.

Implementing a Binary Tree

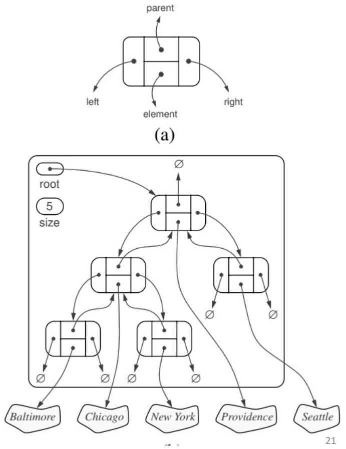


Implementing Binary Tree

- A binary tree can be represented in two ways:
  - Linked structure
  - Array implementation

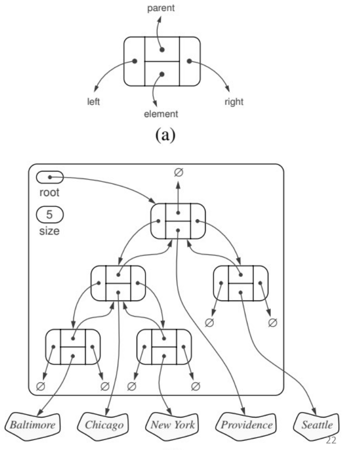
Linked Structure

- A natural way to realize a binary tree T is to use a linked structure
- Each node that maintains references children and parent of p.
- If p is the root of T, then the parent field of p is None.
- Likewise, if p does not have a left child (respectively, right child), the associated field is None.

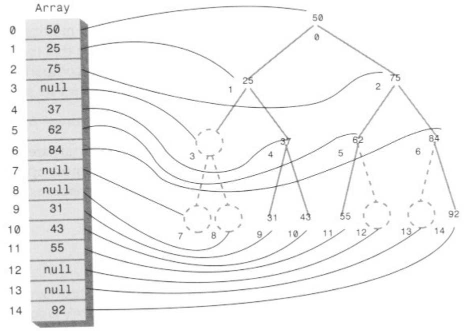


Linked Structure

- The itself tree maintains:
  - An instance variable storing a reference to the root node (if any),
  - A variable, called size, that represents the overall number of nodes of T .



Array Implementation of a Binary Tree



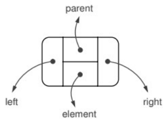
	Array
0	50
1	25
2	75
3	null
4	37
5	62
6	84
7	null
8	null
9	31
10	43
11	55
12	null
13	null
14	92

Binary Search Tree Implementation in Python Using a Linked Structure

Definition of a Node

- A node that stores some data, and references to its left and right child nodes.

```
def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
```



(a)

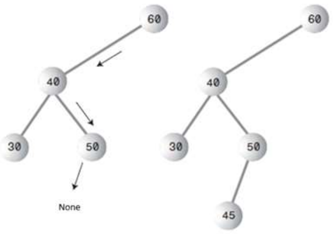
Basic Operations

- 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.

Insert Operation - Pseudocode

- 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.

Example: Insert 45



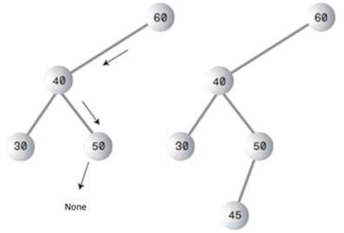
Insert Operation - Algorithm

1. If the tree is empty, insert the first element as the root node of the tree. The following elements are added as the leaf nodes.
2. If an element is less than the root value, it is added into the left subtree as a leaf node.
3. If an element is greater than the root value, it is added into the right subtree as a leaf node.
4. The final leaf nodes of the tree point to NULL values as their child nodes.

Insert Operation – Python Code

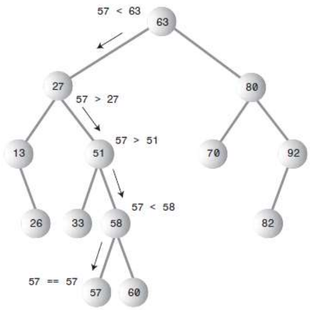
```
# Insert method to create nodes
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
         elif data > self.data:
            if self.right is None:
               self.right = Node(data)
            else:
               self.right.insert(data)
         else:
            self.data = data
```

Example: Insert 45



Search Operation - Pseudocode

- 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.

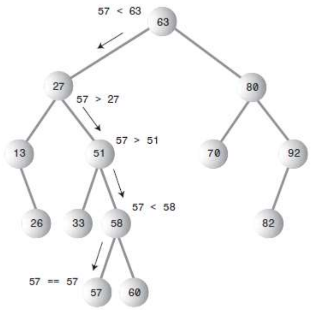


Search Operation - Algorithm

1. Check whether the tree is empty or not
2. If the tree is empty, search is not possible
3. Otherwise, first search the root of the tree.
4. If the key does not match with the value in the root, search its subtrees.
5. If the value of the key is less than the root value, search the left subtree
6. If the value of the key is greater than the root value, search the right subtree.
7. If the key is not found in the tree, return unsuccessful search.

Search Operation – Python Code

```
# search method to compare the value with nodes
   def search(self, key):
      if key < self.data:
         if self.left is None:
            return str(key)+" Not Found"
         return self.left.search(key)
      elif key > self.data:
         if self.right is None:
            return str(key)+" Not Found"
         return self.right.search(key)
      else:
         print(str(self.data) + ' is found')
```



Tree Traversal

- **Traversal**: Visiting all nodes in a specific order.
- **Inorder**: 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.
- **Preorder**: In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.
- **Postorder**: 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.

Inorder Traversal- Pseudocode

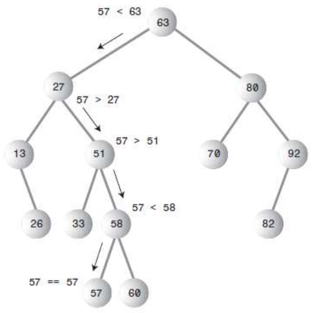
- Firstly, we traverse the left child of the root node/current node, if any.
- Next, traverse the current node.
- Lastly, traverse the right child of the current node, if any.

Inorder Traversal- Algorithm

1. Traverse the left subtree, recursively
2. Then, traverse the root node
3. Traverse the right subtree, recursively.

Inorder Traversal- Python Code

```
def Inorder(self):
    if self.left:
      self.left.Inorder()
    print(self.data, "->", end = " ")
    if self.right:
      self.right.Inorder()
```



Preorder Traversal

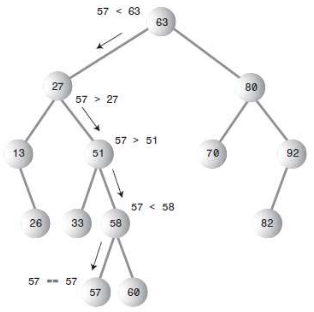
- The preorder traversal operation in a Binary Search Tree visits all its nodes. However, the root node in it is first printed, followed by its left subtree and then its right subtree.

Preorder Traversal Algorithm

1. Traverse the root node first.
2. Then traverse the left subtree, recursively
3. Later, traverse the right subtree, recursively.

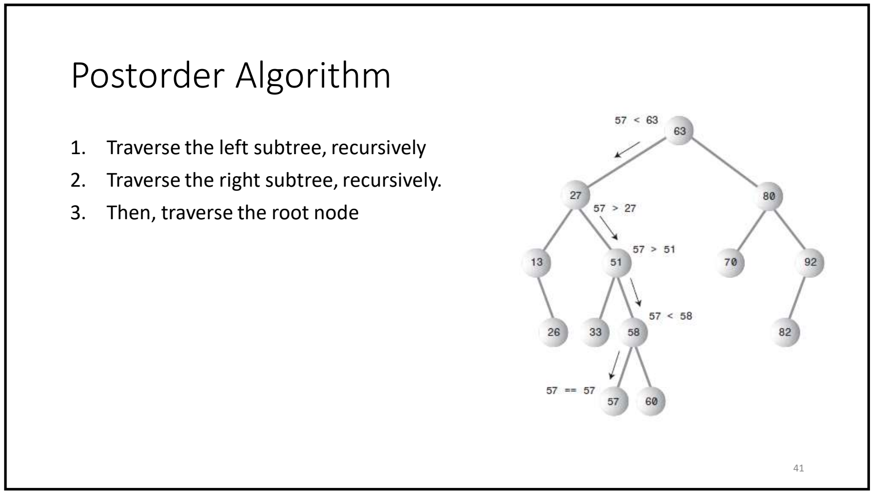
Preorder Traversal Python Code

```
def Preorder(self):
    print(self.data, "->", end = "")
    if self.left:
      self.left.Preorder()
    if self.right:
      self.right.Preorder()
```



Postorder Traversal

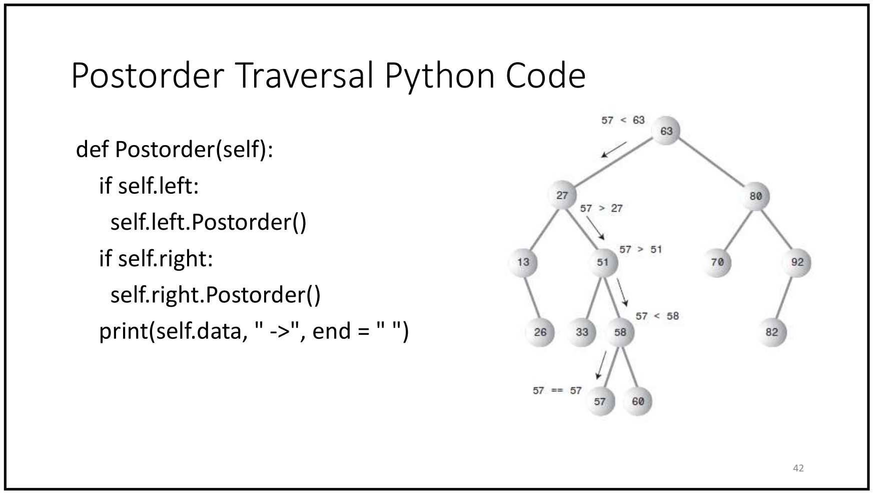
- Like the other traversals, postorder traversal also visits all the nodes in a Binary Search Tree and displays them. However, the left subtree is printed first, followed by the right subtree and lastly, the root node.



Postorder Algorithm

1. Traverse the left subtree, recursively
2. Traverse the right subtree, recursively.
3. Then, traverse the root node

57 < 63
63
27
80
57 > 27
57 > 51
13
51
70
92
57 < 58
26
33
58
82
57 == 57
57
60



Postorder Traversal Python Code

```
def Postorder(self):
    if self.left:
      self.left.Postorder()
    if self.right:
      self.right.Postorder()
    print(self.data, " ->", end = " ")
```

57 < 63
63
27
80
57 > 27
57 > 51
13
51
70
92
57 < 58
26
33
58
82
57 == 57
57
60