Trees

Agenda

- Linear data structure
- Non-Linear data structure
- Introduction to trees
- Tree terminologies
- Trees and its types
- Binary Trees and its implementation
- Tree Traversals
- Tree traversal analysis
- Binary search tree
- Operations on BSTs
- Summary

Linear data structure

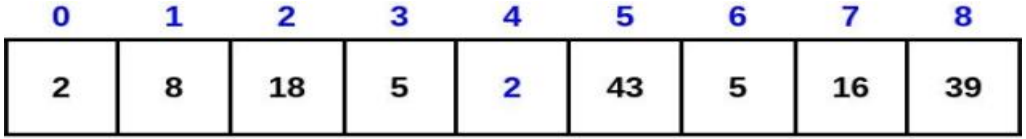
- Linear data structures:
  - Where each element is stored in a linear order.
  - Each element is accessible from its previous or next element
- Linear data structures are used when the data need to be stored in sequential form:
  - List of numbers in a series
  - A list of Students in a class arranged in alphabetical order
  - Street information in Geospatial application managed in sequential order

Example:

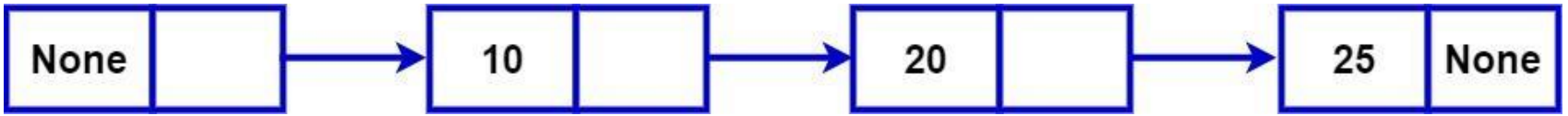
  - Array/list, Linked list, Stacks, Queues
- Is it the only way to store data in Computer?
- What is non-linear data structures?

Linear data structure

- Array

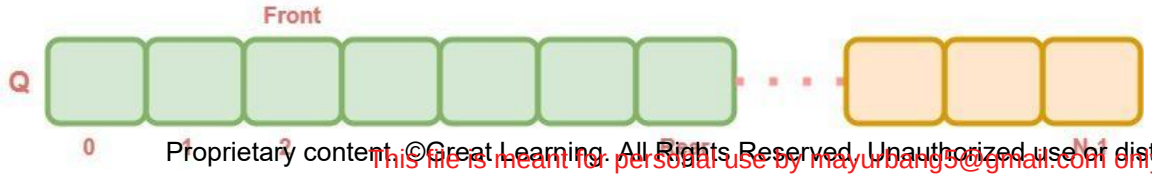


- Linked List



- Stack

- Queue

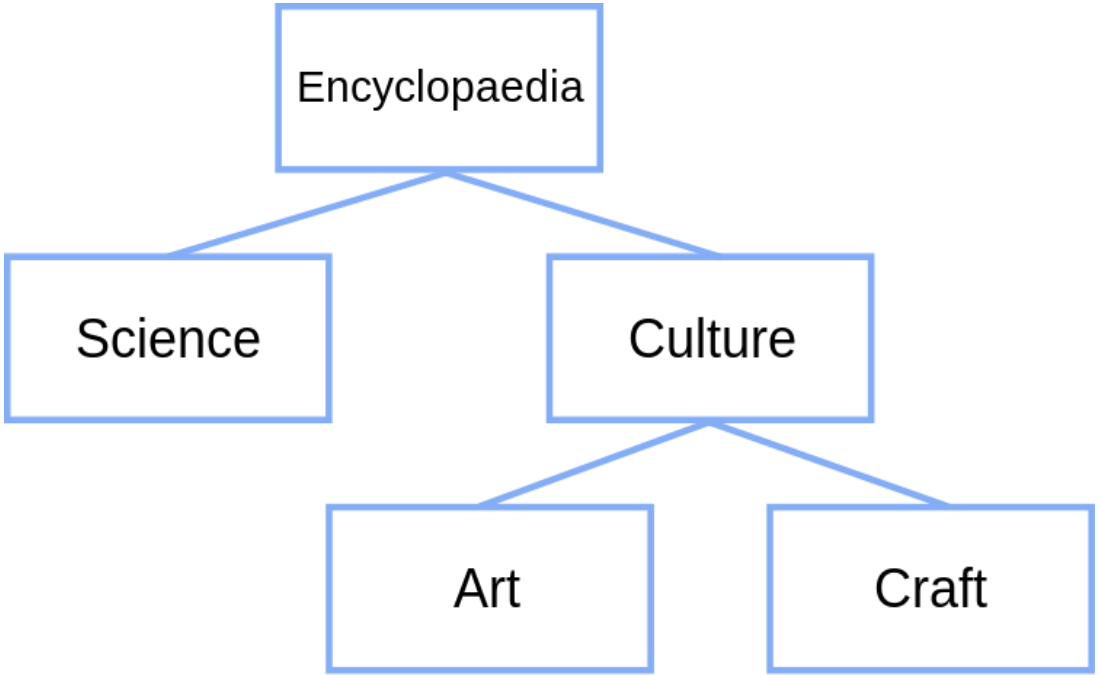


Non-linear data structure

- Non-linear data structures:
  - Where elements are not stored in linear order
  - Data elements are stored at multiple levels
  - Accessing each element is not possible in one traversal or without re-visiting the some data points
- Non-linear data structures are used when the data need to be stored in non-sequential form:
  - Corporate/Government structure
  - File directory in Computer systems
  - Family descendents tree
- Example:
  - Trees and its variations
  - Graphs

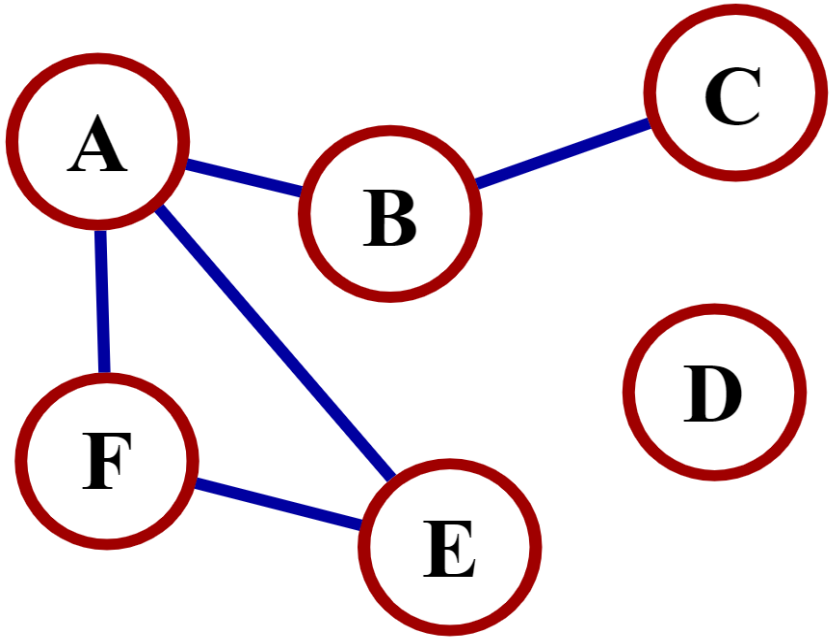
Non-linear data structure

- Tree



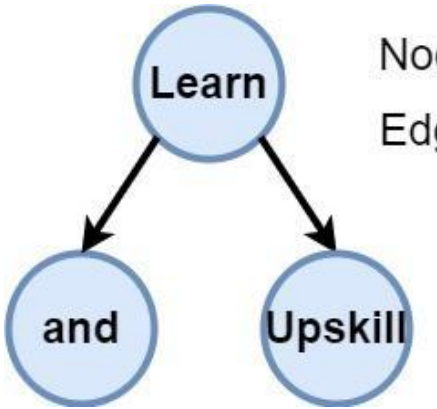
Non-linear data structure

- Graph



Non-linear data structure

- Lets understand some of the terms that are going to be used throughout the non-linear data structure topic.
  - nodes: The structure that holds the information; such as data, information of other nodes
  - edges: This term is used to describe the link between the nodes.
- Look at the following structure and reference;



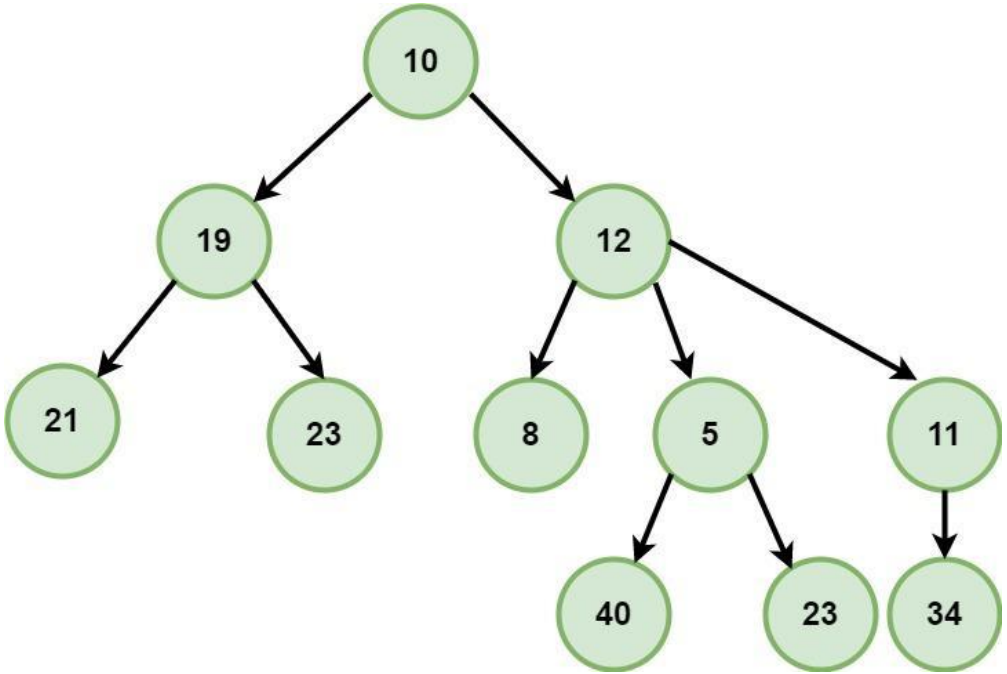
Nodes/Vertices: Learn, and, Upskill

Edges: Connection between nodes

Introduction to Trees

- A tree is a collection of finite number of nodes that are connected to other nodes through edges
- However there are few conditions that need to be satisfied to declare a data structure as a tree data structure:
  - A tree will have a root node which will help in accessing all other nodes.
  - A tree can not have cycle;
  - In a tree every child node will have one parent but parent can have multiple children
  - All nodes in tree should be reachable from root node

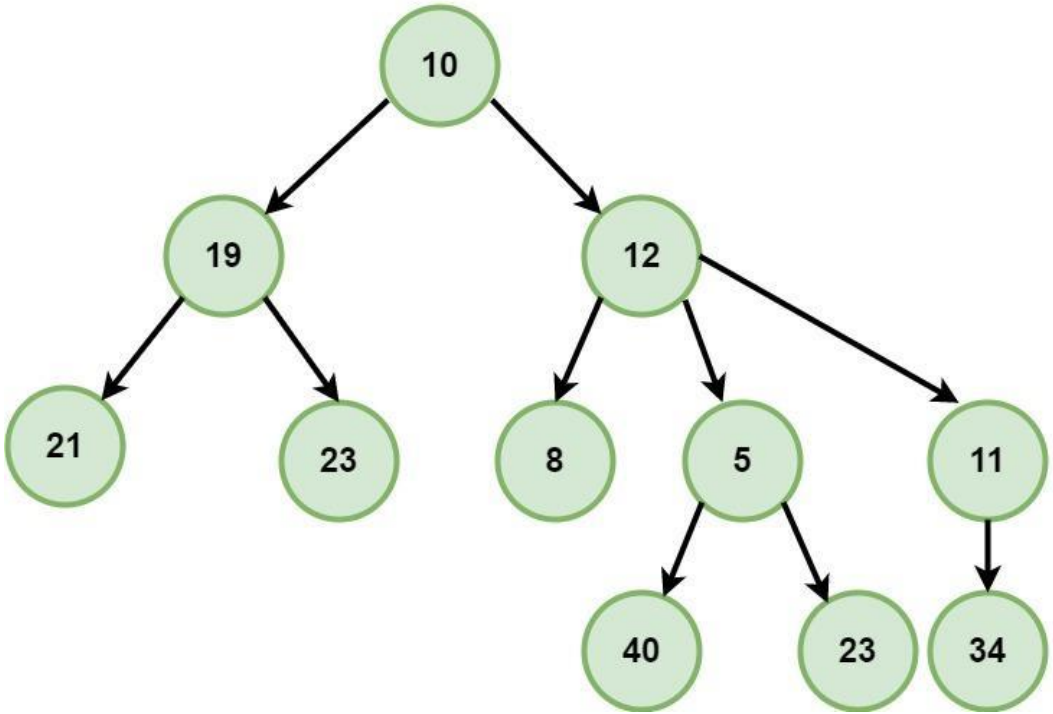
Introduction to Trees



Tree terminologies

- Let’s explore other terms that are used in tree data structure
  - root: top most node of tree, only node that has no incoming edges
  - edge: link between the nodes
  - child node: a node that has an incoming edge from some node is child node of that node
  - parent node: a node that has edge connected to a child node
  - subtree: subset of nodes and connections of the original tree
  - leaf node: nodes that do not have any child node
  - height: length of the longest path to leaf node from root node
  - level: number of edges from root to a particular node is considered as level of node
  - node degree: Number of child to a particular node

Tree terminologies



(10): root node
(19) & (12): child of (10)
(21), (23), (8), (40), (23), (34): leaf nodes
(19) & (12): example of subtrees
height of tree: 3
degree(12): 3

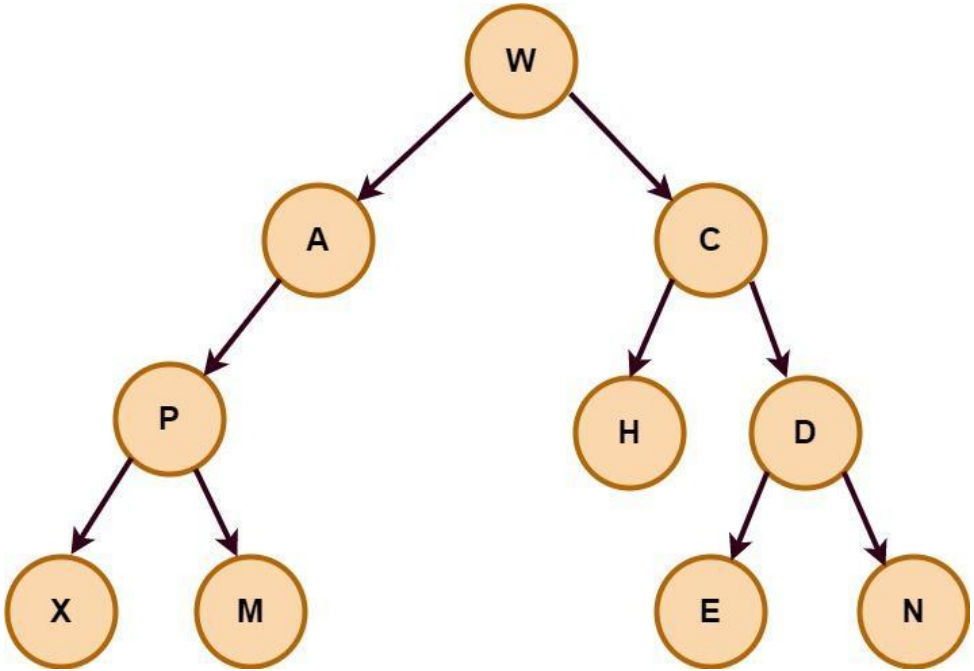
Tree and its types

- Based on the number of child and the values being stored, there could be multiple types of trees.
- Let’s have a look at some of the types of trees data structures:
  - General tree
  - Binary Tree
  - Binary Search trees
  - B-trees

Binary Trees Implementation

- Let’s have a look at some of the features of Binary tree data structure
  - It will have same terms and properties of a general tree
  - A node can have at most two children
  - These children are generally named as left and right child of the parent.
  - Every node will have three data points
    - Value: payload or data points
    - Reference to left child
    - Reference to right child

Binary Trees Implementation



Binary Trees Implementation

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


# create root
root = Node(W)
# adding some nodes in the tree
root.left      = Node(A);
root.right     = Node(C);
root.left.left  = Node(P);
```

Binary Trees Implementation

- Let’s understand the process completed in the previous step:
  - A structure is defined to hold the necessary information for every node
  - Then we created the root node that will be used as entry point to add more nodes in the tree
  - Created different nodes and established a link/edge with different nodes using root node.

Binary Tree Traversal

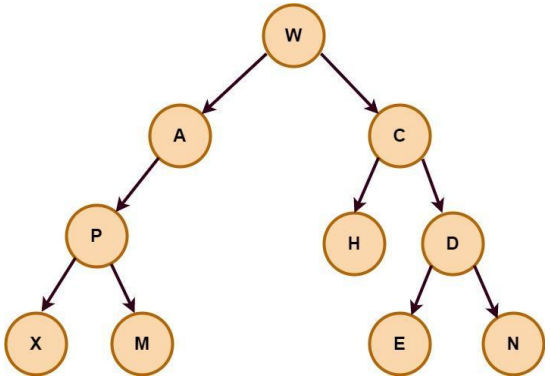
- Once we have created and inserted the nodes in the tree,
- Next step should be to ensure that we are able to access the elements available in the tree
- Based on the way we are accessing the nodes; there are three general approaches available
  - **Preorder:** Visit the root node, recursivley call preorder method on the left subtree, followed by a recursive call to preorder method for the right subtree
  - **Postorder:** recursively call postorder method to the left subtree and then recursively call postorder method on right subtree, followed by visit to the root node.
  - **Inorder:** recursively call inorder method on the left subtree, Visit the root node, and finally do a recursive inorder method call of the right subtree.

Binary Tree Traversal

- **Preorder: We will be using the same image that was shown earlier to**

```
#Preorder Traversal
Preorder(node)
  if node is not None
      visit the node
      print the data
  Preorder(node-->left)
  Preorder(node-->right)
```

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



Output: W A P X M C H D E N

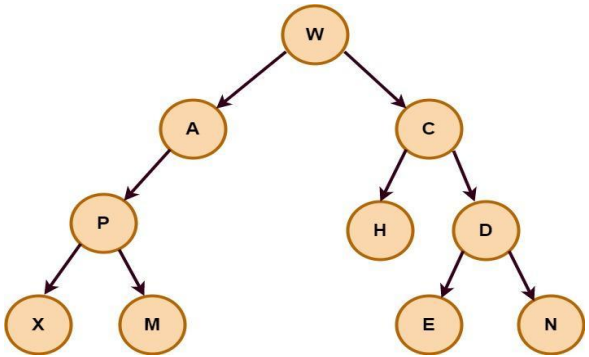
Binary Tree Traversal

- **Postorder**

```
#Postorder Traversal

Postorder(node)
  if node is not None
      Postorder(node-->left)
      Postorder(node-->right)
      visit the node
      print the data
```

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



Output: X M P A H E N D C W

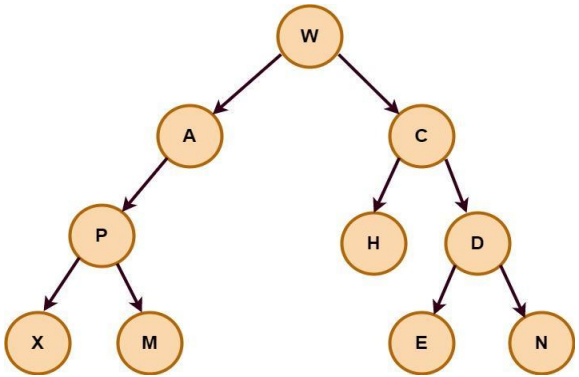
Binary Tree Traversal

- **Inorder**

```
#Inorder Traversal

Inorder(node)
  if node is not None
      Inorder(node-->left)
      visit the node
      print the data
      Inorder(node-->right)
```

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



Output: X P M A W H C E D N

Binary Tree traversal analysis

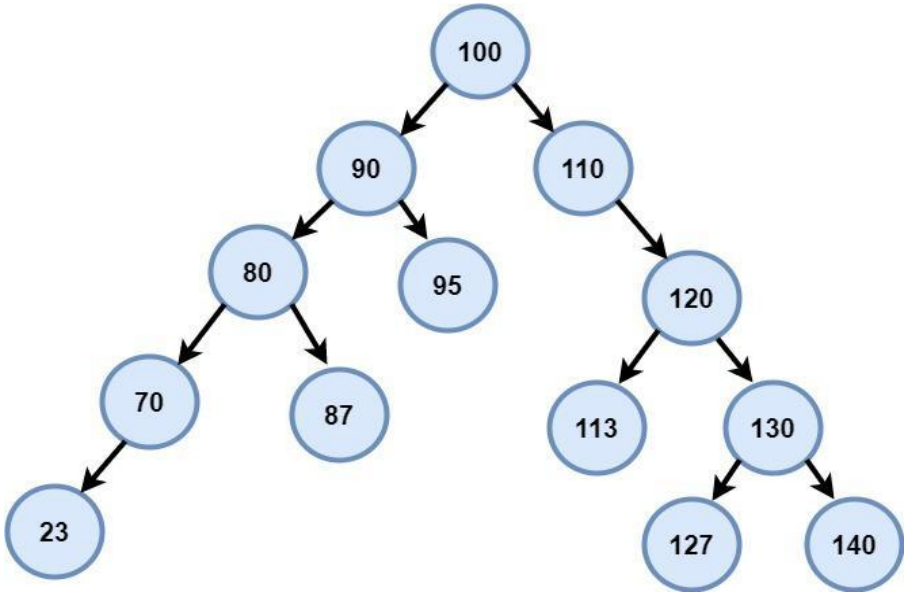
- Let’s do the analysis of operations on tree
  - All the traversal are supposed to access every node once hence the execution time for:
    - Preorder: number of nodes
    - Postorder: number of nodes
    - Inorder: number of nodes
  - Because the algorithms are calling each function recursively, total space consumption for the mentioned methods are:
    - Preorder: height of tree
    - Postorder: height of tree
    - Inorder: height of tree

Binary search tree

- Let’s discuss one of the variation of binary tree known as binary search tree
- A binary tree can be classified as binary search tree iff:
  - Values that are less than the parent are found in the left subtree,
  - Values that are greater than the parent are found in the right subtree.
  - Every parent can have at most two children
  - Every subtree will follow the BST property

Binary search tree

- Example



Binary search tree

- As discussed; every node in in BST will have place holder to store one value and address of left and right children
  - In case if a node has left children; left will point to left children
  - In case if a node has right children; right will point to right children
  - In case if a node does not have children, respective left and right will point to NULL

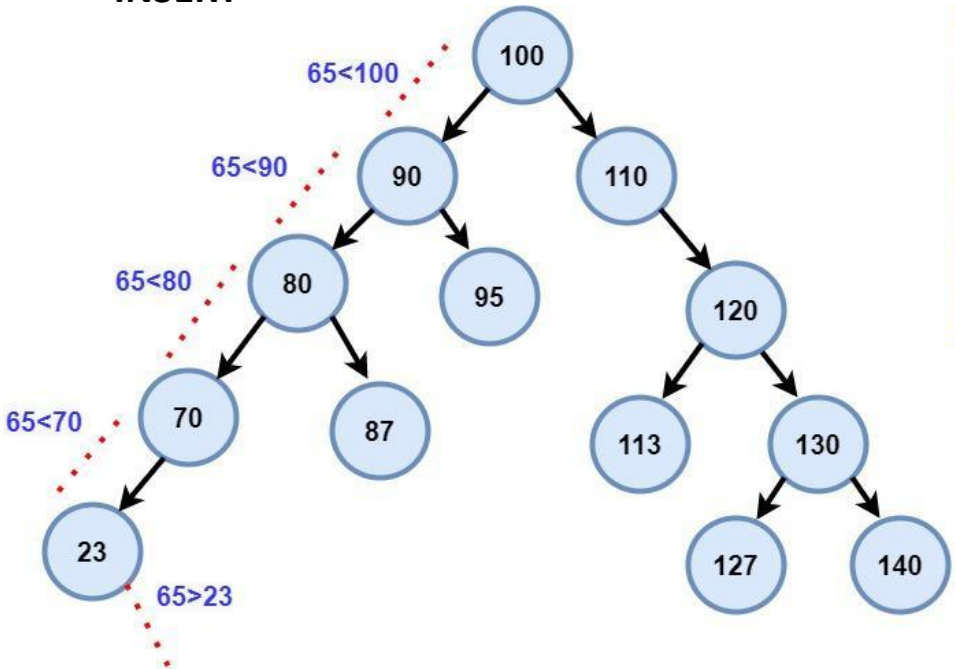
```
class Node:
    def __init__(self, data):
        self.data = data
        self.right = None
        self.left = None
        self.parent = None
```

Binary search tree - operation

- As we know that every data structure should accept operation on insertion, deletion and searching of the element in the designed data structure.
- BST also supports these operations.
  - Insertion (root, new_node): Inserting the new node in the BST. The insertion of node will always happen at the leaf level.
  - Deletion(root, value): In this operation, we will be deleting the element from the tree. This operation can have three variations
    - Deletion of leaf node
    - Deletion of a node that have one child
    - Deletion of node that have two children
  - Searching(root, value):
    - Traversing the tree to find the element in the BST

Binary search tree - operation

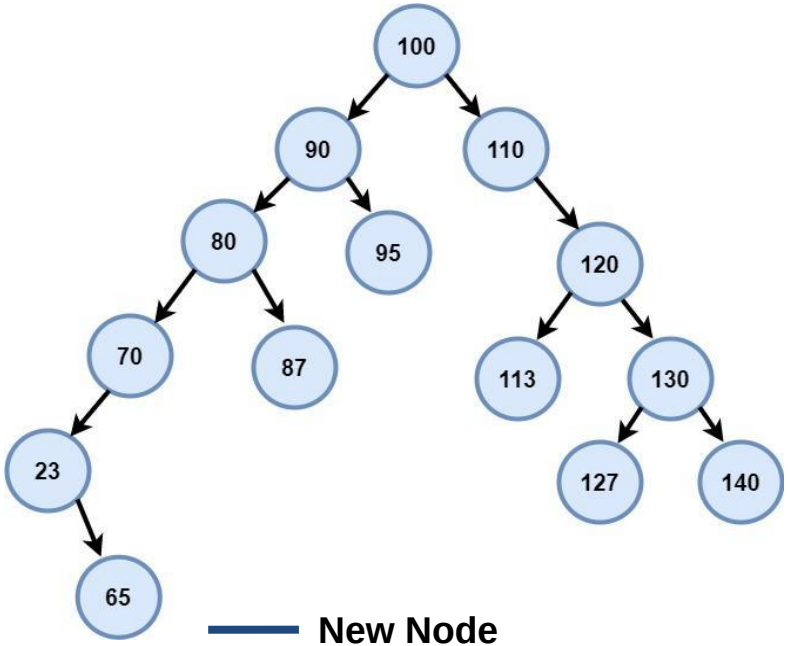
- **INSERT**



INSERT(root, 65)
Comparison 1: 65<100 --> Go left
Comparison 2: 65<90 --> Go left
Comparison 3: 65<80 --> Go left
Comparison 4: 65<70 --> Go left
Comparison 5: 65>23 --> Go right
The node and place is empty
Add 65 as right child of 23.

Binary search tree - operation

- **INSERT**



Binary search tree - operation

- **INSERT**

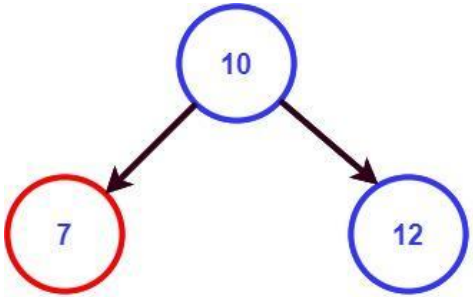
```
INSERT(root, new_node)
    temp = root
    p = NULL
    while temp != NULL
        p = temp
        if new_node.data < temp.data
            temp = temp.left
        else
            temp = temp.right
    new_node.parent = p
    if p==NULL
        root = new_node
    else if new_node.data < p.data
        p.left = new_node
    else
        p.right = new_node
```

Binary search tree - operation

- **Deletion**
  - In BST deletion is possible based in the number of children available to the target node
    - No child
    - Single child
    - Both children

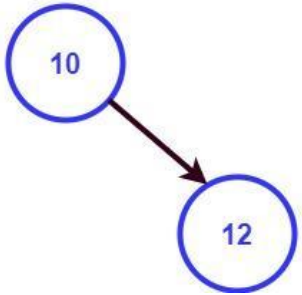
Binary search tree - operation

- **Deletion - leaf node**



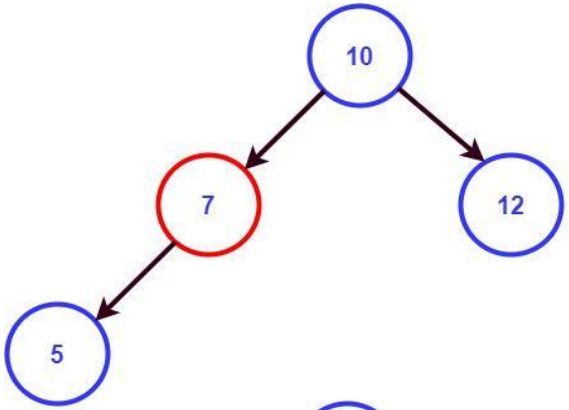
Delete(7)
No children node for 7
left_child of 10 will become
NULL

Output



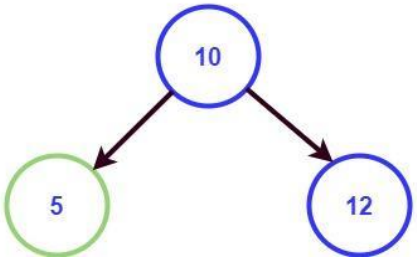
Binary search tree - operation

- **Deletion - single child**



Delete(7)
One child node for 7
left_child of 10 will become 5

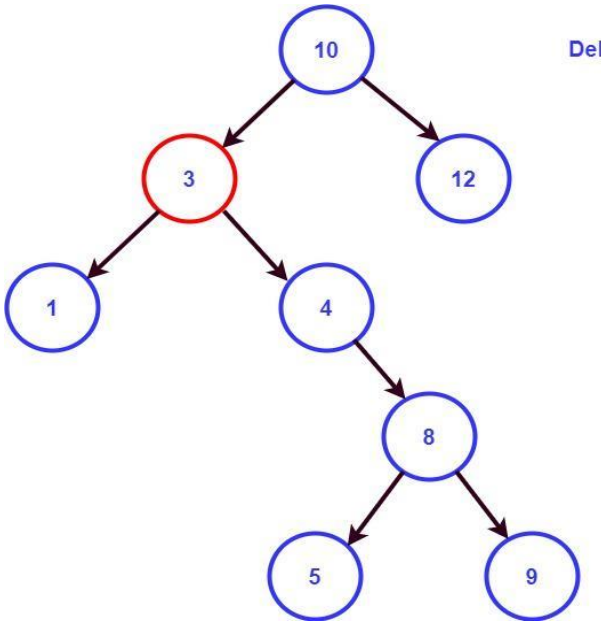
Output



Updated Tree

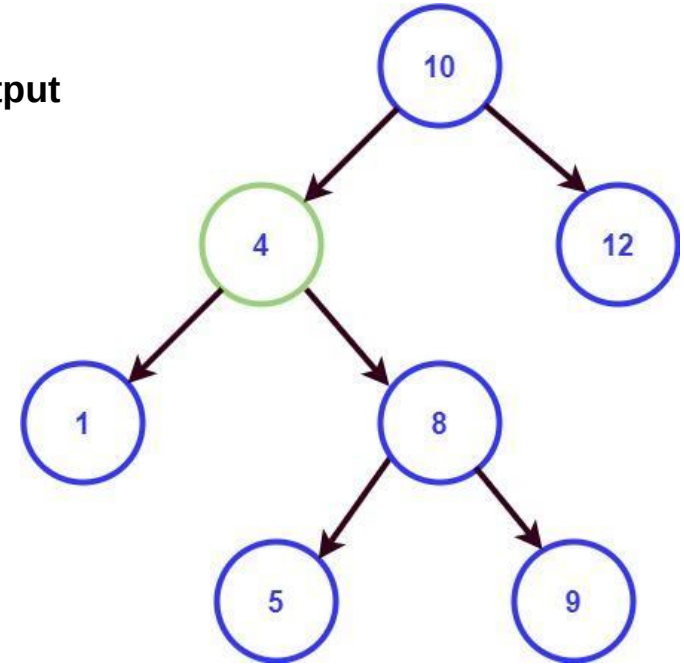
Binary search tree - operation

- **Deletion - both children**



Delete(3)
two child node for 3
replace the node with the
smallest node in right subtree

Output



Binary search tree - operation

- **Deletion**

```
DELETE(root, target_node)
    if target_node.left == NULL
        DELTEHELPER(root, target_node, target_node.right)
    elseif target_node.right == NULL
        DELTEHELPER(root, target_node, target_node.left)
    else
        // minimum value node from target node
        y = MINIMUM(target_node.right)
        // target node is not direct child
        if y.parent != target_node
            DELTEHELPER(root, y, y.right)
            y.right = target_node.right
            y.right.parent = y
        DELTEHELPER(root, target_node, y)
        y.left = target_node.left
        y.left.parent = y
```

```
DELETEHELPER(root, u, v)
    //u is root
    if u.parent == NULL
        root = v
    //u is left child
    elseif u == u.parent.left
        u.parent.left = v
    //u is right child
    else
        u.parent.right = v
    if v != NULL
        v.parent = u.parent
```

Summary

- We understand that linear and non-linear data structure are
- We understood Binary trees and different operation on B-tree
- We learn about tree traversals such as preorder, postorder and inorder.
- We understood the applications of Binary tree in real time.