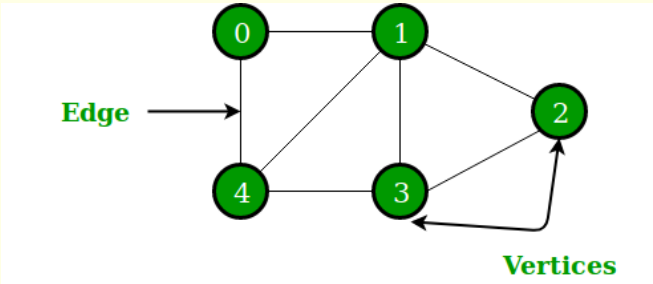
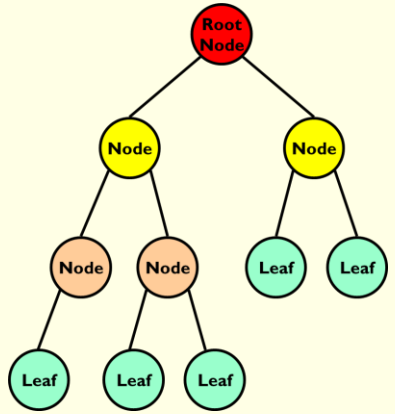
Non-linear Data Structures

GRAPH



Modeling connections or relationships between items

TREE



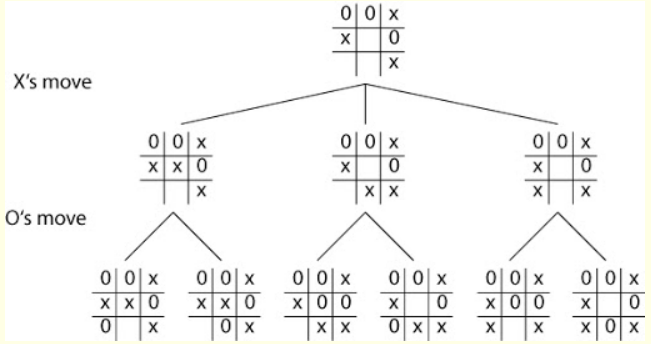
Hierarchical data structure

MAPS

WHY?

Why?

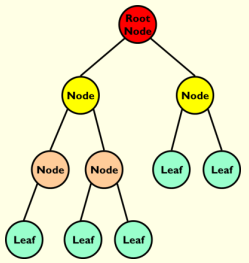
Data can be searched more efficiently
Used to represent a Decision Tree



Tree

TREE

Left Subtree



Right Subtree

Hierarchical data structure

Root Node stores the first data item
Node can have up to 2 descendants
Leaf Nodes have no descendants
Edge/ branches is the line

Type of Tree:
A common kind of **tree** is a **Binary Tree**
Every node can have at the most two child nodes

Depth-First Traversal
- Pre-Order
- In-Order
- Post-Order

Breadth-First

The Applications of Trees
Managing folder structures
Binary Trees are used in routes to **store routing tables**
Binary Search Trees can be built to **speed up searching**
Expression trees can be used to **represent algebraic** and **Boolean expressions** that simplify the processing of the expression

Learning Aims

- Define a binary tree as a rooted tree in which each node has at most two children
- Create and traverse a binary tree
- Create, search and traverse a binary search tree

Traversal - Methods of looking at each node in a tree, in turn

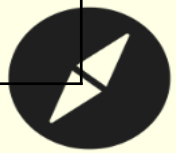
Depth-First

- Pre-Order
- In-Order
- Post-Order

Breadth-First

Key Definitions

Term	Definition
Tree	A hierarchical data structure made up of nodes, with one root node and sub-nodes called children.
Root Node	The topmost node in the tree with no parent.
Leaf Node	A node that has no children.
Parent Node	A node that has one or more children.
Child Node	A node that descends from another node (its parent).
Subtree	A tree formed from a node and all of its descendants.
Depth	The number of edges from the root to a given node.
Height	The number of edges on the longest path from a node to a leaf.
Binary Tree	A tree where each node has at most two children (left and right).
Binary Search Tree (BST)	A binary tree with the property that all nodes in the left subtree contain values less than the parent and all nodes in the right subtree contain values greater than the parent.



Keyword	Definition
Node	An item in a tree
Edge	Connects two nodes together and is also known as a branch or pointer
Root	A single node which does not have any incoming nodes
Child	A node with incoming edges
Parent	A node with outgoing edges
Subtree	A subsection of a tree consisting of a parent and all the children of a parent
Leaf	A node with no children
Traversing	The process of visiting each node in a tree data structure, exactly once

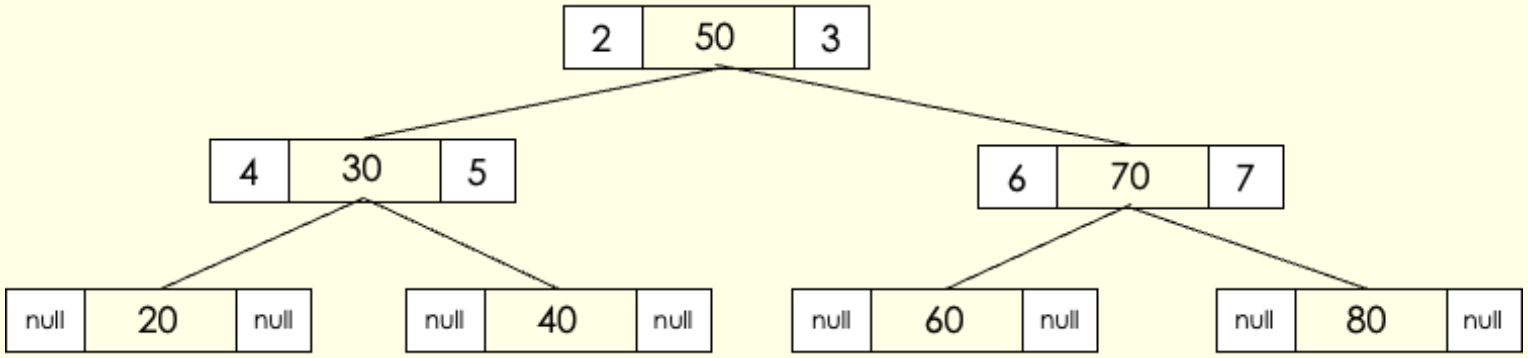
Concept of a tree

Trees are a very common data structure in many areas of computer science and other contexts.

Like a tree in nature, a **rooted tree** has a root, branches and leaves, the difference being that a tree in computer science has its root at the top and its leaves at the bottom.

Typical uses for rooted trees include:

- manipulating hierarchical data, such as folder structures or moves in a game
- making information easy to search
- manipulating sorted lists of data



Tree

- A tree is a data structure used to represent **non-linear** data.
- A tree consists of **nodes** (also called vertices) that are connected by **edges** (also called arcs).
- Each node contains some data, and each edge indicates a relationship between the nodes it connects.

Binary Tree

- A binary tree is a **rooted tree** where every node has **at most two child nodes**. This means that a tree where every nodes has either two, one or none child nodes is a binary tree.

Binary Search Tree

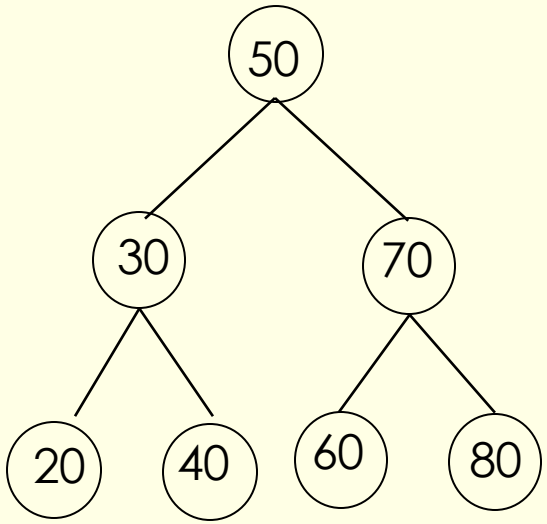
- A binary search tree (BST) is a **rooted tree** where the nodes of the tree are **ordered.**
- Binary Search Trees can have a **maximum of 2 children** for each parent node, but it is not a requirement that they have 2.
- An advantage of storing data in a binary search tree instead of a 1-dimensional array: searching is faster (O(log n), inserting new tasks is faster and do not need to sort the structure (each time a new task is inserted)

- If the order is ascending (low to high), the nodes of the left subtree have values that are lower than the root, and the nodes of the right subtree have values that are higher than the root.
- This property is true for any node of the tree; the nodes of its left subtree will have values that are lower, and the nodes of its right subtree will have values that are higher.

- **Imagine that you are searching for the number 40:**

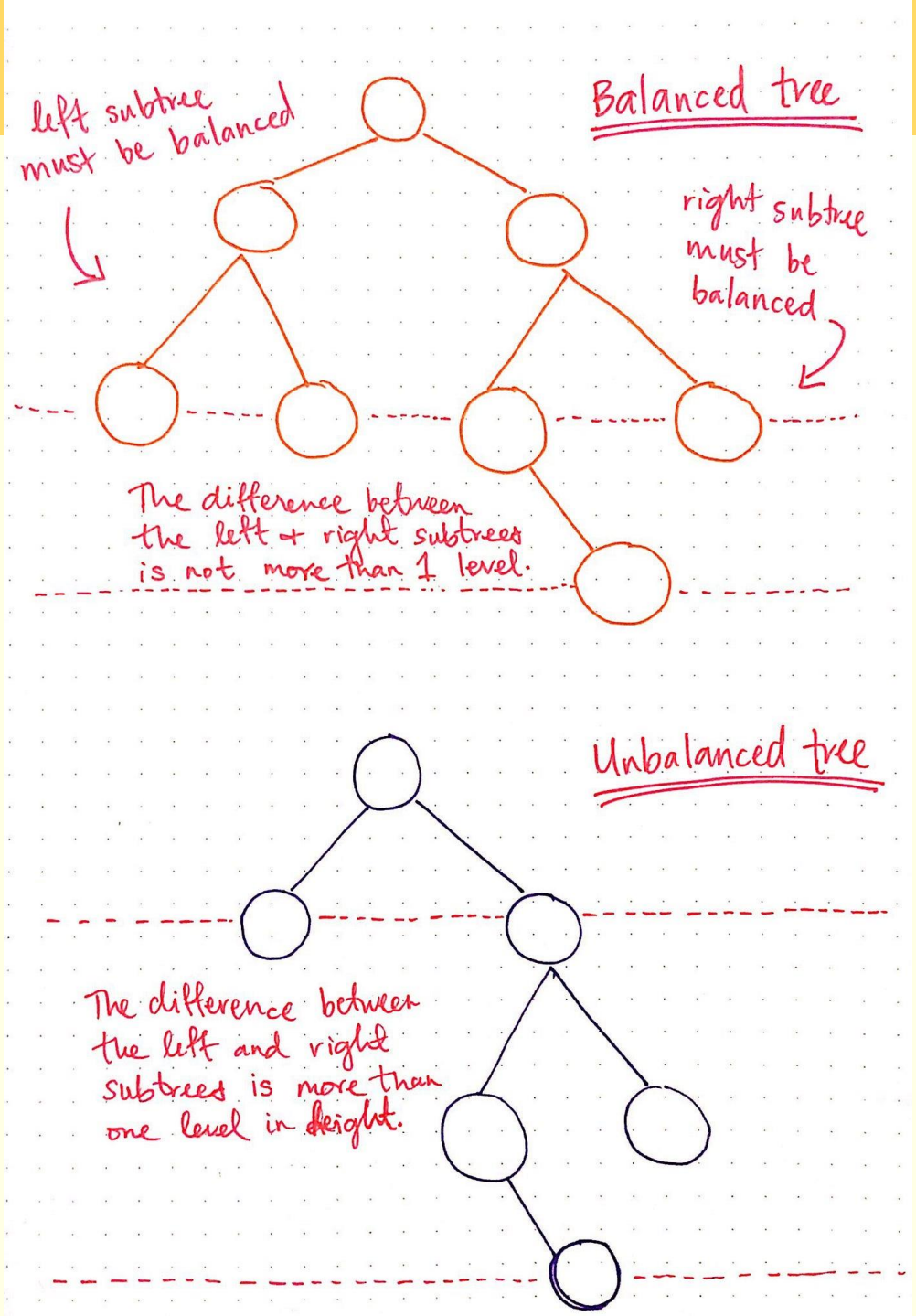
  - Start at the root 50
  - Compare 40 with 50 - 40 is lower, so check the left child
  - Compare 40 with 30 - 40 is higher, so check the right child
  - Compare 40 with 40 - success!

  You have found the item with 3 comparisons.



Unbalanced vs balanced

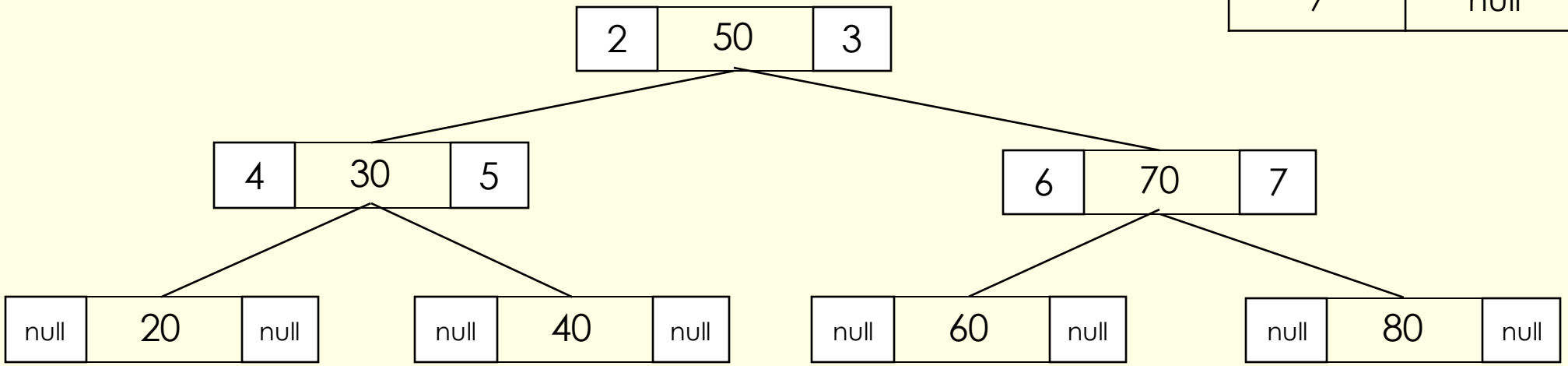
- Drawback of an unbalanced binary search tree when searching for a value
- If the tree is unbalanced, nodes may form a long chain.
- Searching may require visiting almost every node in the tree.
- **Time complexity** worsens from **O(log n)** (balanced) to **O(n)** (unbalanced), meaning more comparisons are needed to find a value.



OCR Exam Abstract Representation of a Binary Tree using an array representation

Data to insert: 50, 30, 70, 20, 40, 60, 80

Position	Left	Value	Right
1	2	50	3
2	4	30	5
3	6	70	7
4	null	20	null
5	null	40	null
6	null	60	null
7	null	80	null

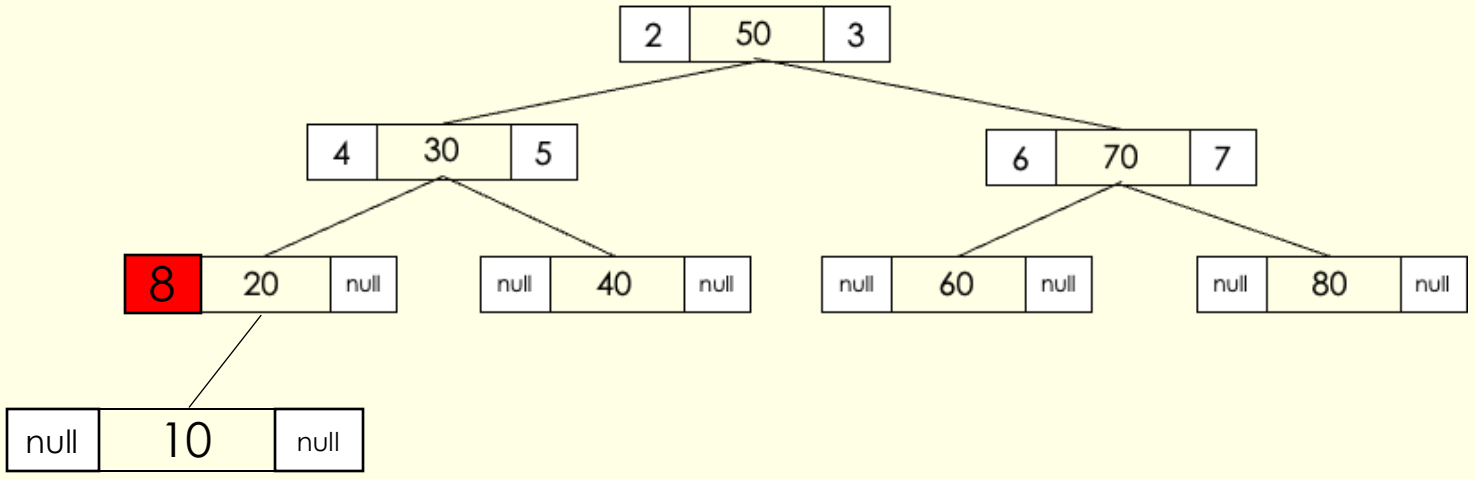


If a node does not have child node then this is indicated with a null or -1

Insert new node

1. Create a new node
2. Start at the root node
3. Search the tree to find the location in of the new node
4. Once a null pointer is found insert the new node at that position by updating the appropriate left or right of the parent node to point to the new node.

Add new node: 10



Position	Left	Value	Right
1	2	50	3
2	4	30	5
3	6	70	7
4	8	20	null
5	null	40	null
6	null	60	null
7	null	80	null
8	null	10	null

Abstract representation of a binary tree

- A binary tree can be stored as objects
- Each node has a left pointer, a right pointer and the data being stored.
- This is the node class and this is used to create a new_node object.

Node object

- **data** which in this case is a simple value attached to the node
- **right** which is a reference to the node to the right of the node
- **left** which is a reference to the node to the left of the node

Class node

class: node
attributes: private data private left: Integer private right: Integer
methods: new (new_data) getData() setData() getLeft () getRigh () setLeft() setRight ()

class Node

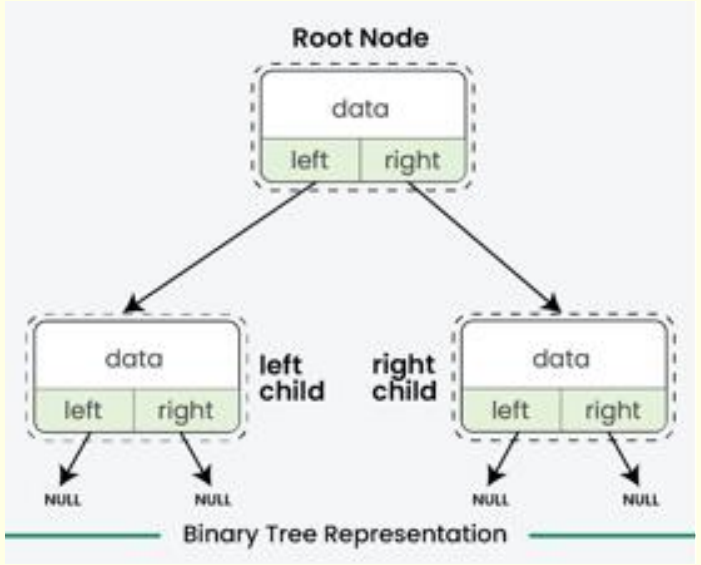
```
    private data
    private left
    private right

    public procedure new (new_data)
        data = new_data # instance variable to store the data
        left = null
        right = null
    end procedure
end class
```

Binary Tree Class

A binary tree has a root node

```
class BinaryTree
    private root
    public procedure new ()
        root = null
    …
end class
```

- This implementation is using **linked-node representation** of a binary tree instead of using an array representation.



Linked-node representation	**Array representation**
Each node stores: • Data value • Pointer (reference) to left child • Pointer (reference) to right child • The tree is connected via pointers rather than	• The tree's nodes are stored in a sequential array. • Each node's position is determined by its index in the array.

Binary tree search algorithm.

The algorithm, returns True if the value is found and False if the value is not found. It is a recursive algorithm and it takes one argument:

- searchValue which is the item being searched for

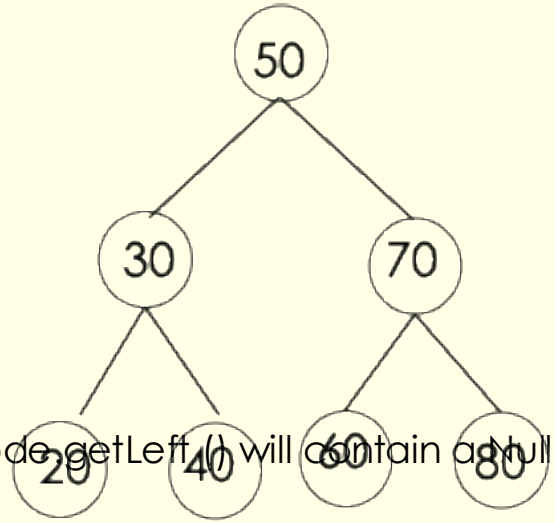
As the data is stored in a tree, we refer to each element as a node rather than an item (as in a list). The node is a structure that contains:

- data which in this case is a simple value attached to the node
- right which is a reference to the node to the right of the node
- left which is a reference to the node to the left of the node

For example, if we examined the root node:

- node.getData() would be 50
- node.getRight() would return a reference to the node that contained the data value 70
- node.getLeft() would return a reference to the node that contained the data value 30

If there is no node to the right, node.getRight() will contain a Null value; if there is no node to the left, node.getLeft() will contain a Null value.



Search for an item – Complete the algorithm

```
function searchforitem(searchValue):
        currentNode = root

        if searchValue == currentNode.getData() then
          return True

        elif searchValue < currentNode.getData() then

          if currentNode.getLeft() != None then
              searchforitem(currentNode.getLeft(), searchValue)

          else:
            return False

        else:
            if currentNode.getRight() != None then
              searchforitem(currentNode.getRight(), searchValue)

            else:
                return False
```

Insert Algorithm

Inserting a node into binary tree

Step 1: Create a New Node

- Assign the new data to the node.
- Set the left and right of the node to null.

Step 2: Check if the Tree is Empty

- If the root is null:
  - Set the root to the new node.

Step 3: Traverse the Tree to find insertion point

- If the tree is not empty:
  - Start from the **root node.**
  - Repeat until you find **null:**
    - Compare the new data with the current node's data:

      If new_data < current_node.data

      - Follow the **left**

      Else

      - Follow the **right**

    - Keep track of the **parent node** during traversal.

  - Once a null pointer is found:

Insert the new node at that position by updating the appropriate left or right of the parent node to point to the new node.

```
Procedure insert(new_value)
    # Step 1: Create a New Node
    new_node = new Node(new_value)
    # Step 2: Check if the Tree is Empty
    if root == null then
      root = new_node
      return
    # Step 3: Traverse the Tree to Find Insertion Point
    current = root
    parent = null
    while current != null:
       parent = current
       if new_value < current.getData()
          current = current.getLeft()
       else
          current = current.getRight()
    # Step 4: Insert the Node
    if new_value < parent.getData() then
       parent.setLeft(new_node)
    else
       parent.setRight(new_node)
```

Deleting a node

Simple method:

Set data Item of deleted node to Null
Leave node within structure

Complicated method

Update left or right pointers of the previous node and if required replace node with a leaf node

Removing an item from a binary tree

Step 1: Find the Node to Delete

- **Start at the root.**
- While the current node does not match the target value
  - Keep track of the **parent** node.
  - If the value to delete is less than the current node's value, go left.
  - If greater, go right.

Step 2: Handle Deletion Cases

Case 1: Node has No Children (Leaf Node) - simply remove it from the tree

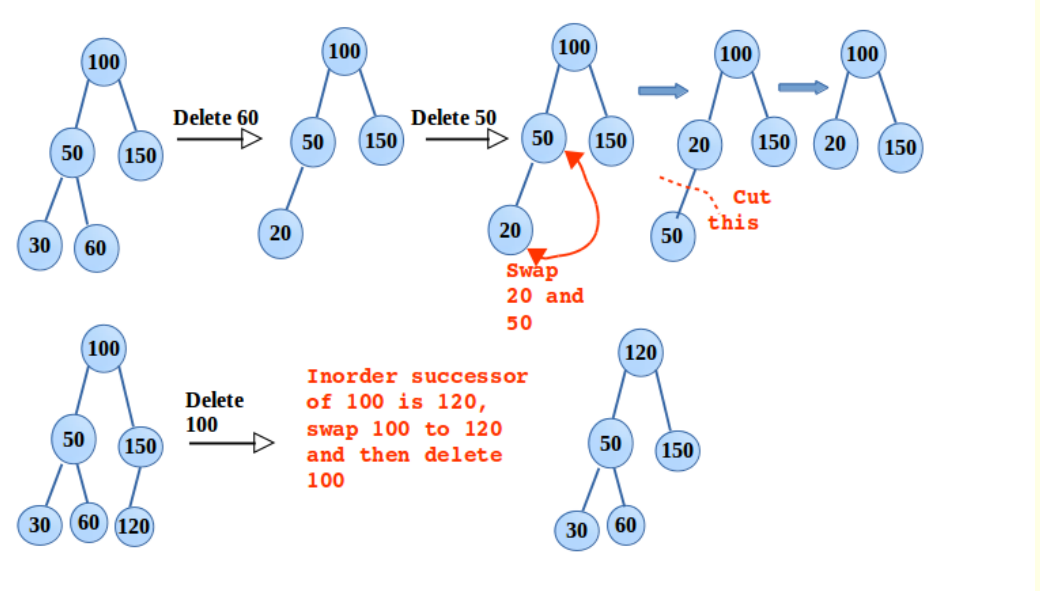
- If the node to delete is a leaf
  - Update the parent's pointer to null.

Case 2: Node has One Child – replace the node with its child

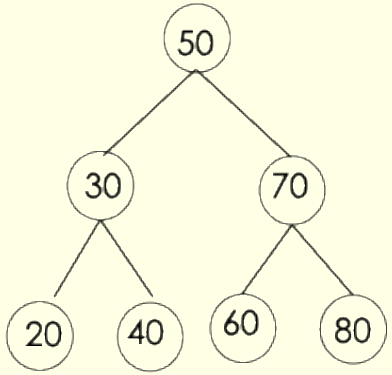
Case 3: Two Children

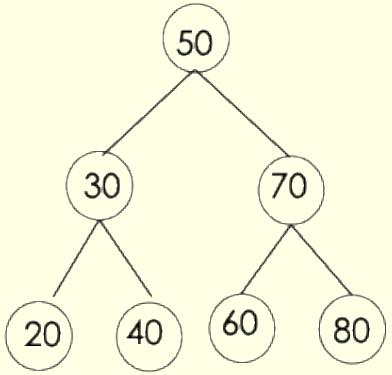
- Find the **smallest value** in the node's **right side** (this is called the in-order successor).
- Copy that smallest value into the node you want to delete.
- Then delete that smallest node (which will now be easy — it will have 0 or 1 child).

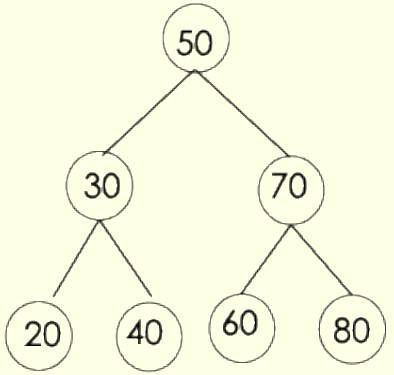
Step 3: Add the deleted node to the free storage list (leave for garbage clear up)



Binary Tree Traversal (Depth-First)



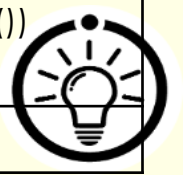




Pre-order Create a copy of a tree. 1. Visit the root. 2. Traverse the left subtree, 3. Traverse the right subtree	In-order traversal 1. Traverse the left subtree, 2. Visit the root. 3. Traverse the right subtree	Post-order Delete a tree 1. Traverse the left subtree 2. Traverse the right subtree 3. Visit the root.
50, 30, 20, 40, 70, 60, 80	20, 30, 40, 50, 60, 70, 80	20, 40, 30, 60, 80, 70, 50

Algorithm Preorder
 	
procedure preorder(current_node)
if current_node != None:
#Visit each node: NLR
print(current_node.getData())
if current_node.getLeft() != None:
preorder(current_node.getLeft())
if current_node.getRight() != None:
preorder(current_node.getRight ())
	Algorithm Inorder
	
	procedure inorder(current_node)
	if current_node != None:
	#Visit each node:
	if current_node.getLeft() != None:
	inorder(current_node.getLeft())
	print(current_node.getData())
	if current_node.getRight() != None:
	inorder(current_node. getRight())
		Algorithm Postorder
		
		procedure postorder(current_node):
		if current_node != None:
		#Visit each node: LRN
		if current_node.getLeft() != None:
		postorder(current_node.getLeft())
		if current_node.getRight() != None:
		postorder(current_node. getRight())
		print(current_node.getData())

Useful for saving a tree.	Outputs values in sorted order for a BST	Useful for deleting nodes.



Breadth-first Traversal:

Traverse through one level of children noes, then traverse thought the level of grand children nodes (and so on…_

First add the root node into the **queue**

while the queue is not empty.

Get the first node in the queue, and then print its value

Add both left and right children into the queue (if the current node has children)

We will print the value of each node, level by level.

BFS Algorithm

```
procedure breadthFirst(currentnode)

    if (currentnode ! = None)
       q = new Queue()
       q.enqueue(currentnode)
       while NOT q.isEmpty()

          currentnode = q.dequeue()

          print (currentnode.getData())


          if(currentnode.getLeft() !=null)
             q.enqueue(currentnode.getLeft())
          if(currentnode.getRight()!=null)
            q.enqueue(currentnode.getRight())
       endwhile
end procedure
```

Check root node is not null

Add root node to queue

While the queue is not empty

print data

Add the left and right child nodes to the queue

When to use BFS vs. DFS?

- Both algorithms can come in handy when traversing through a tree to look for a value, but which one is better?
- It all depends on the structure of the tree and what you are looking for.
- If you know the value that you are looking for is closer to the top, a BFS approach might be a superior choice, but if a tree is very wide and not too deep, a DFS approach might be faster and more efficient.

A comparison

Breadth is more efficient when the data searched for is closer to the root.

Depth is more efficient when data to be search for is further down.

Depth memory requirement stores only the current path

Breadth uses additional memory because stores all nodes at the current level

If the item you're searching for is close to the root, BFS will find it early without exploring unnecessary deeper branches.

DFS might go very deep into a branch that has nothing to do with the target before backtracking.