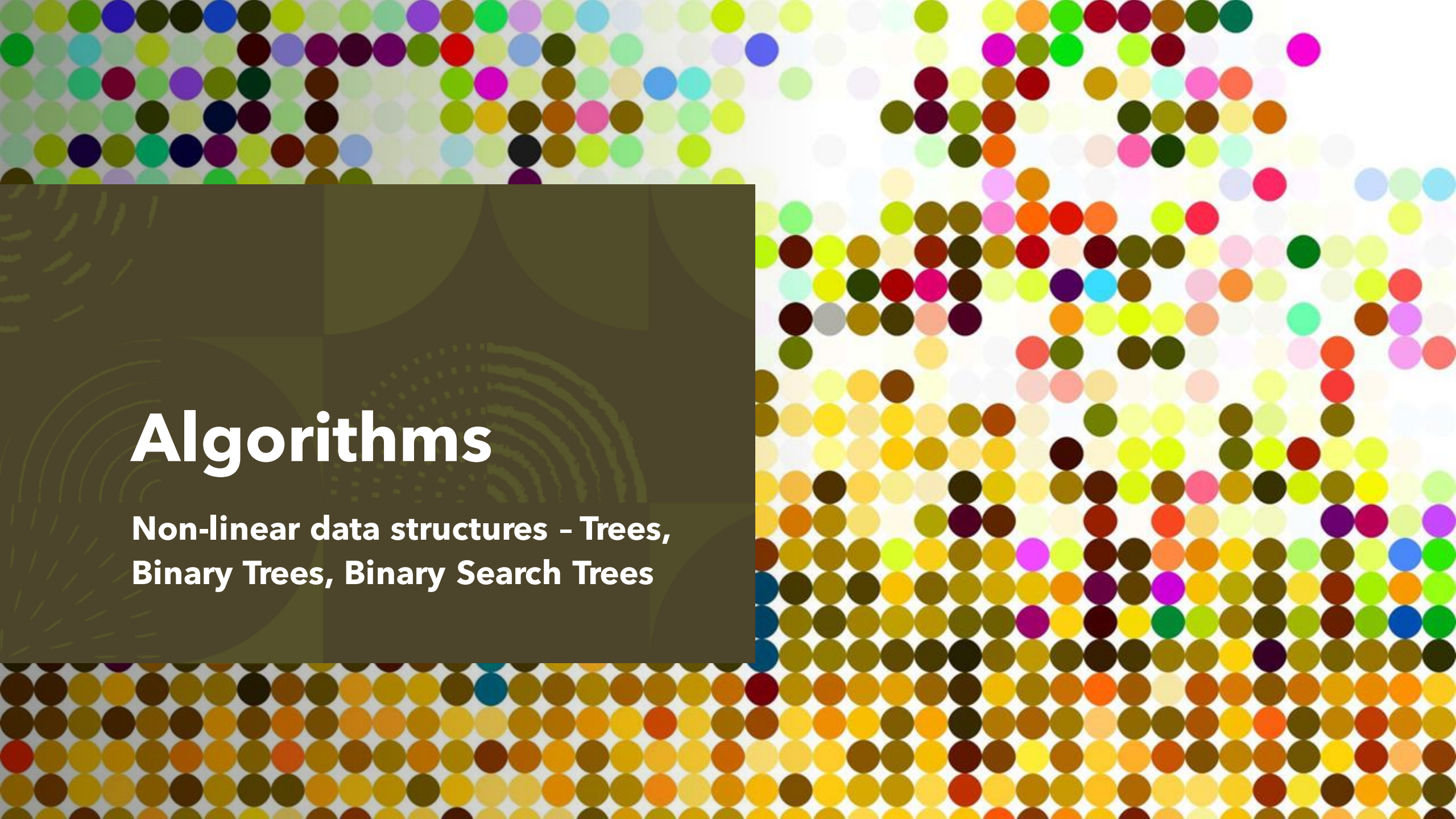
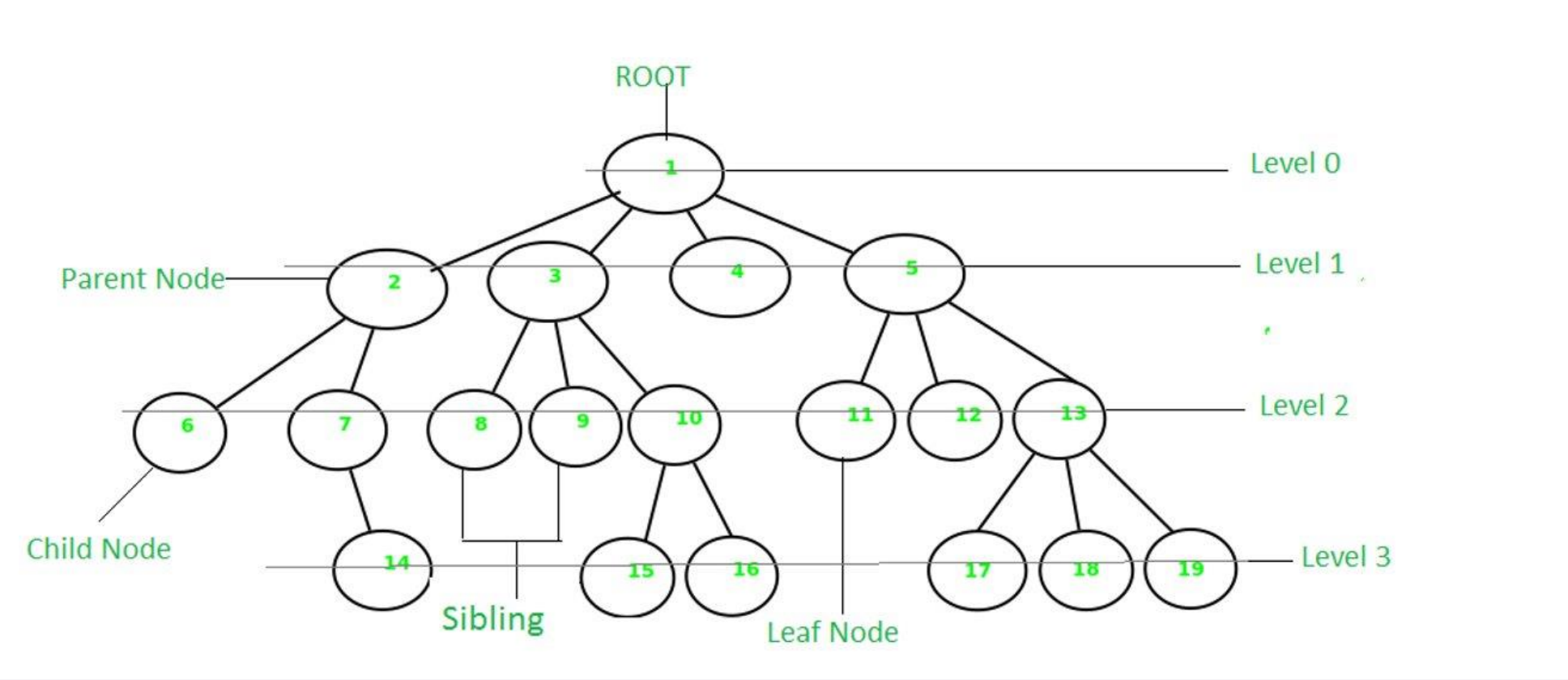

Algorithms
Non-linear data structures – Trees,
Binary Trees, Binary Search Trees

Introduction

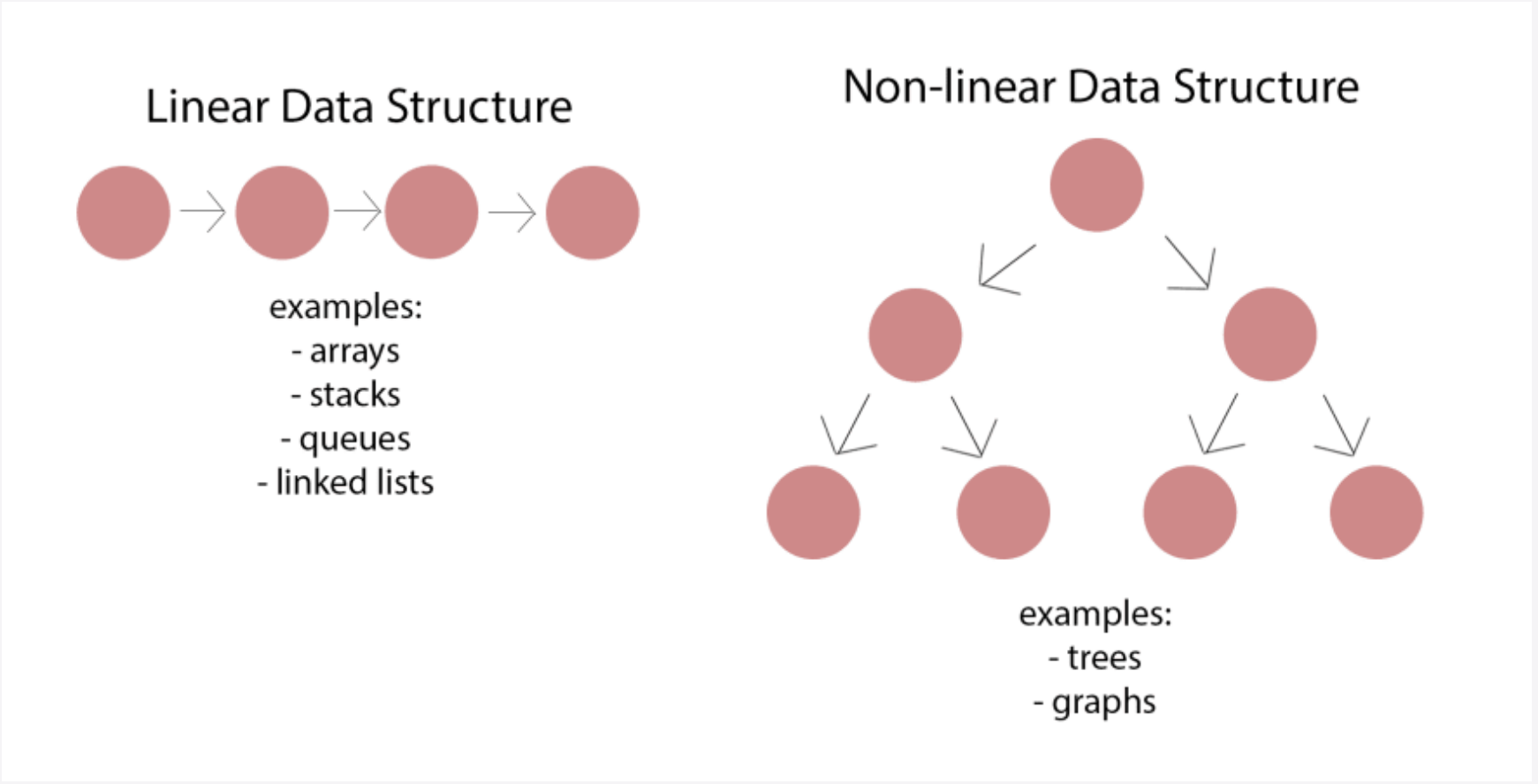
It is a widely used **abstract data type** (**ADT**) that represents a **hierarchical tree structure** with a set of connected **nodes**.

Each **node** in the tree can be connected to many **children**, but connected to exactly one **parent**, except for the **root node**, which has **no parent**.



There are constraints: **no cycles** or **"loops"** (no node can be its own ancestor), and that each **node** can be treated like the **root node** of its own subtree, making **recursion** a useful technique for tree traversal.

In contrast to **linear data structures**, many **trees cannot be represented by relationships between nodes in a single straight line**.



Terminology (1/2)

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node.
- **Root Node:** The top-most node of a tree or the node which does not have any parent node is called the root node.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes.
- **Subtree:** Any node of the tree along with its descendants
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node.

Terminology (2/2)

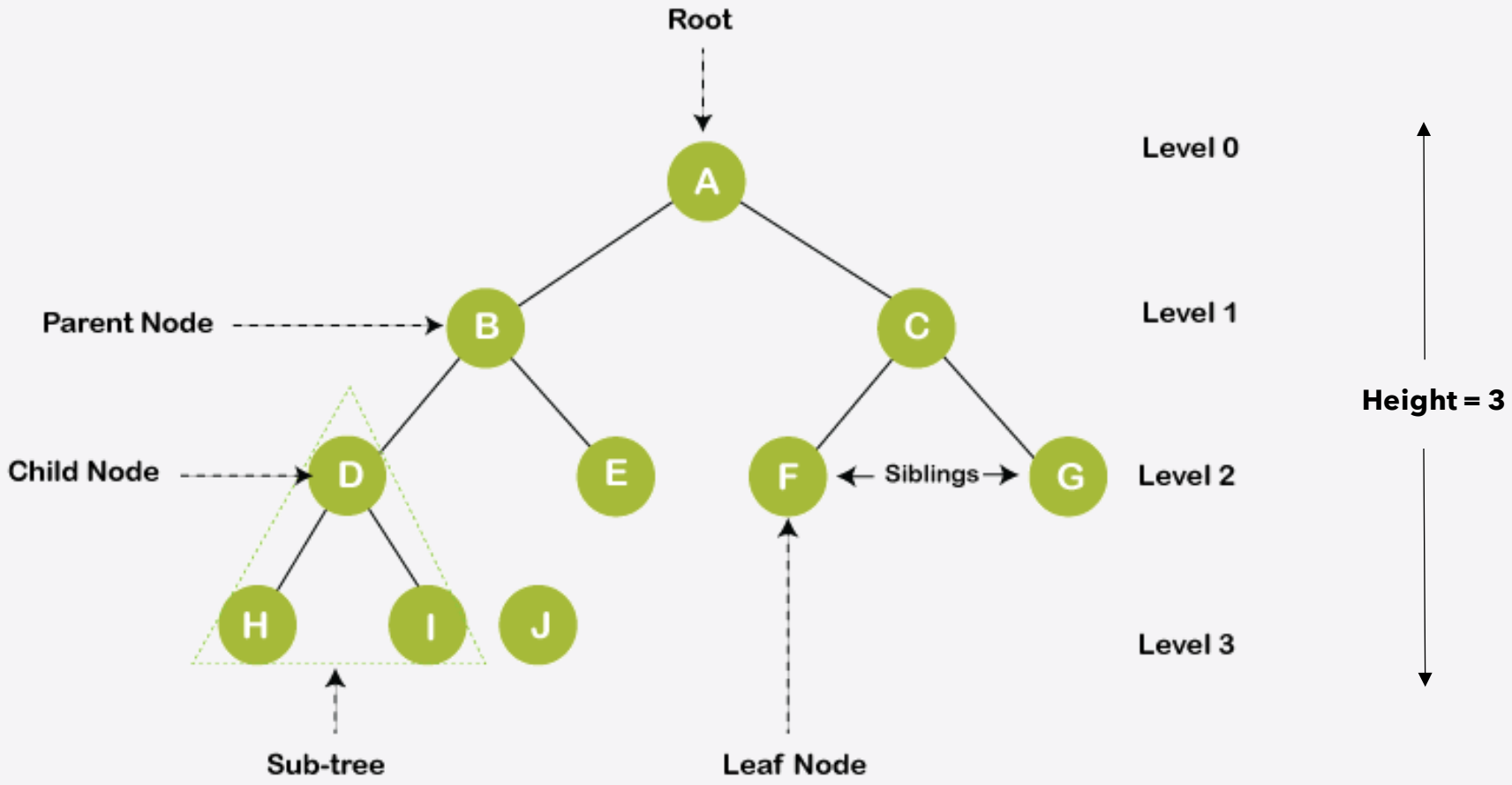
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node.
- **Descendant:** Any successor node on the path from the leaf node to that node.
- **Sibling:** Children of the same parent node are called siblings.
- **Depth (height/level) of a node:** The count of edges from the root to the node.
- **Height of a tree:** The count of edges from the root to the deepest node.
- **Internal node:** A node with at least one child.

Binary trees

Binary trees

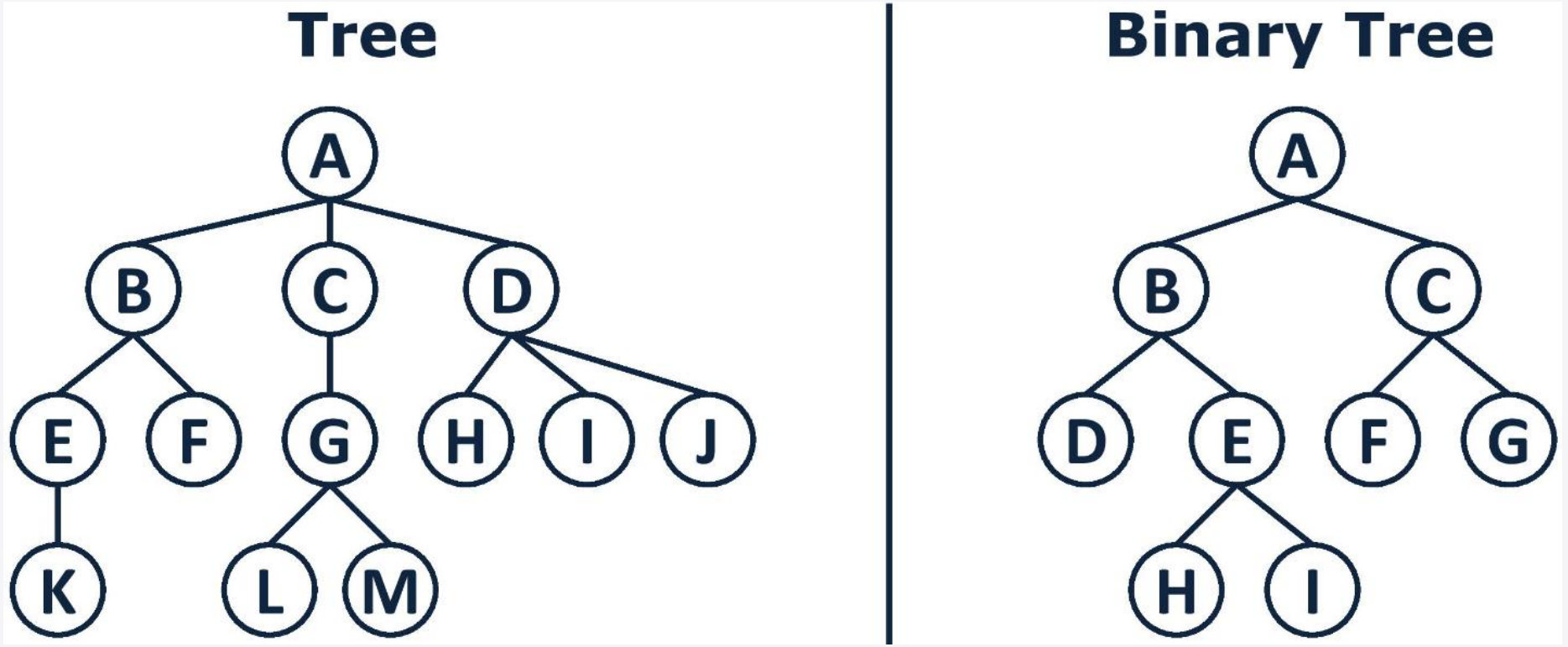
In a binary tree, **each node has at most two children**, which are referred to as the **left child** and the **right child**.

A **value** or **pointer** to other data may be associated with every **node** in the tree.



Trees vs. Binary trees

Remember that a binary tree **each node has up to two children**. **No all trees are binary trees**. For example, the displayed tree is not a binary tree. You could call it a **ternary tree**.

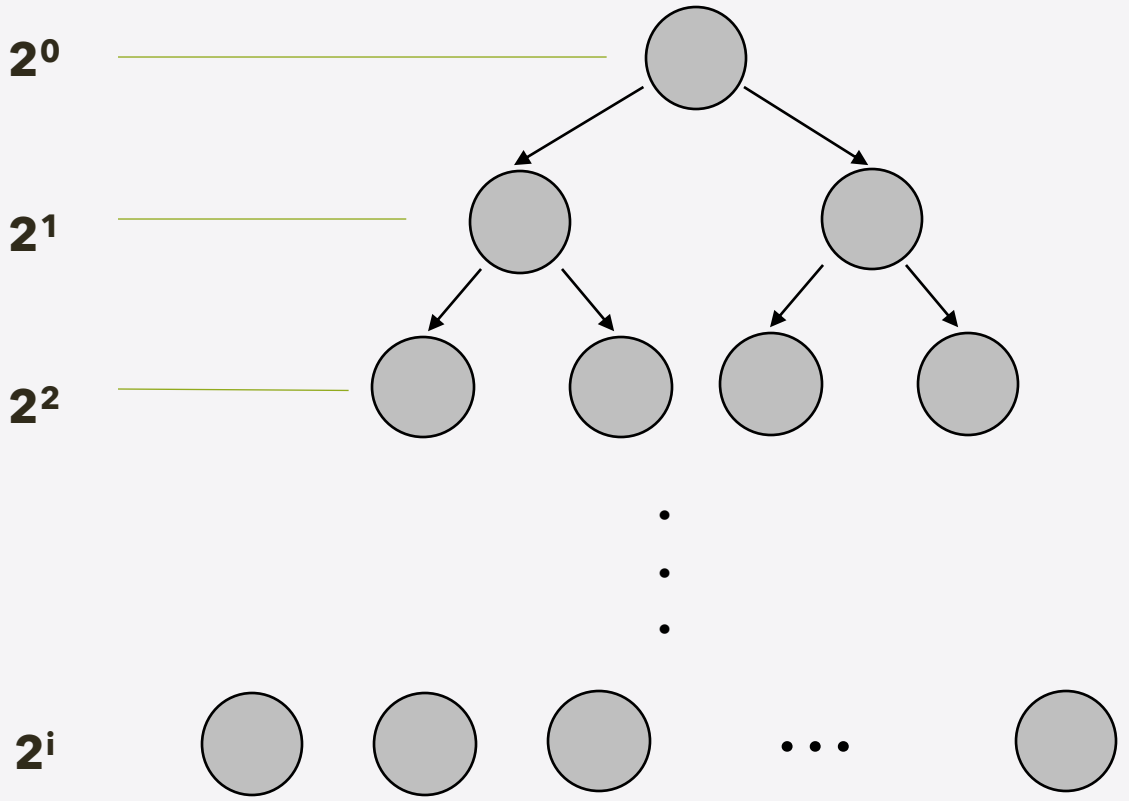


Properties Binary trees (1/3)

Given the fact that each node can have up to two child nodes, it makes us wonder about the properties of binary trees.

- What is the Maximum number of nodes at **level i**?

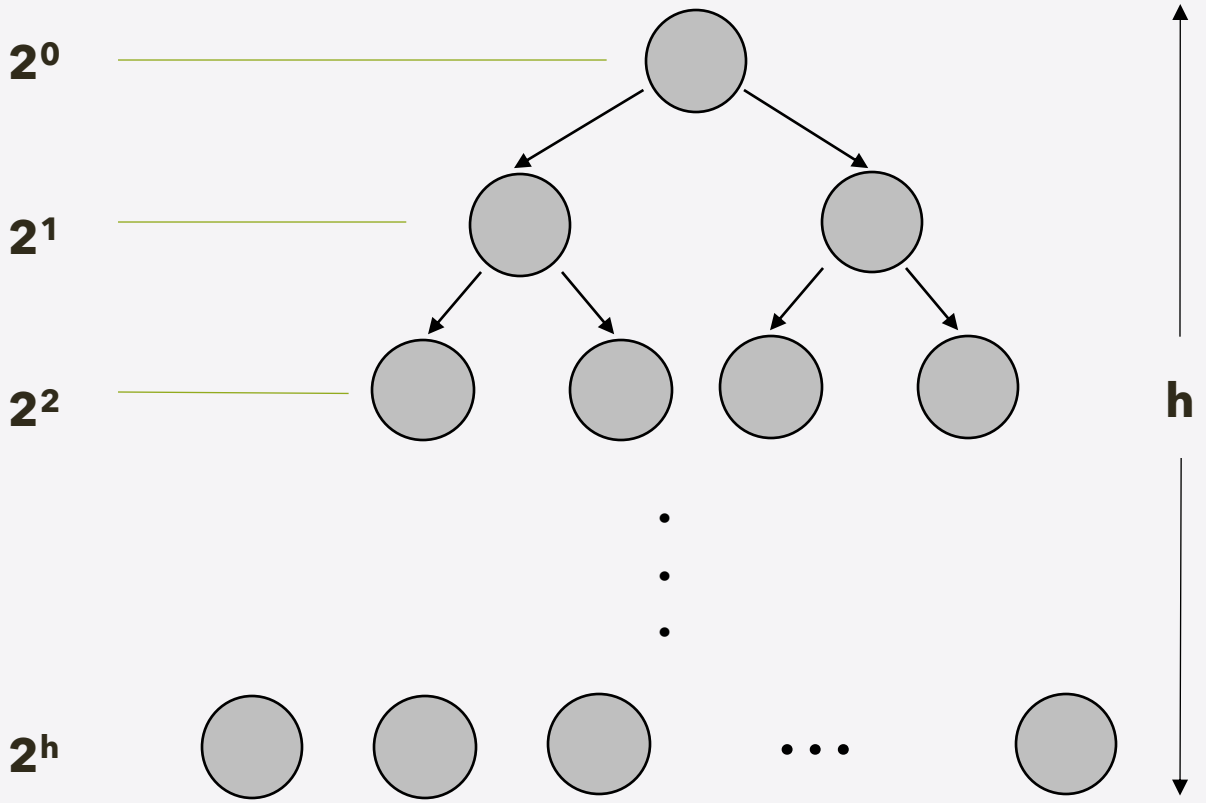
 # Nodes for level:



Properties Binary trees (2/3)

- What is the **Maximum number of nodes** in a binary tree of **height "h"**?

 # Nodes for level:



Using the **Geometric series**:

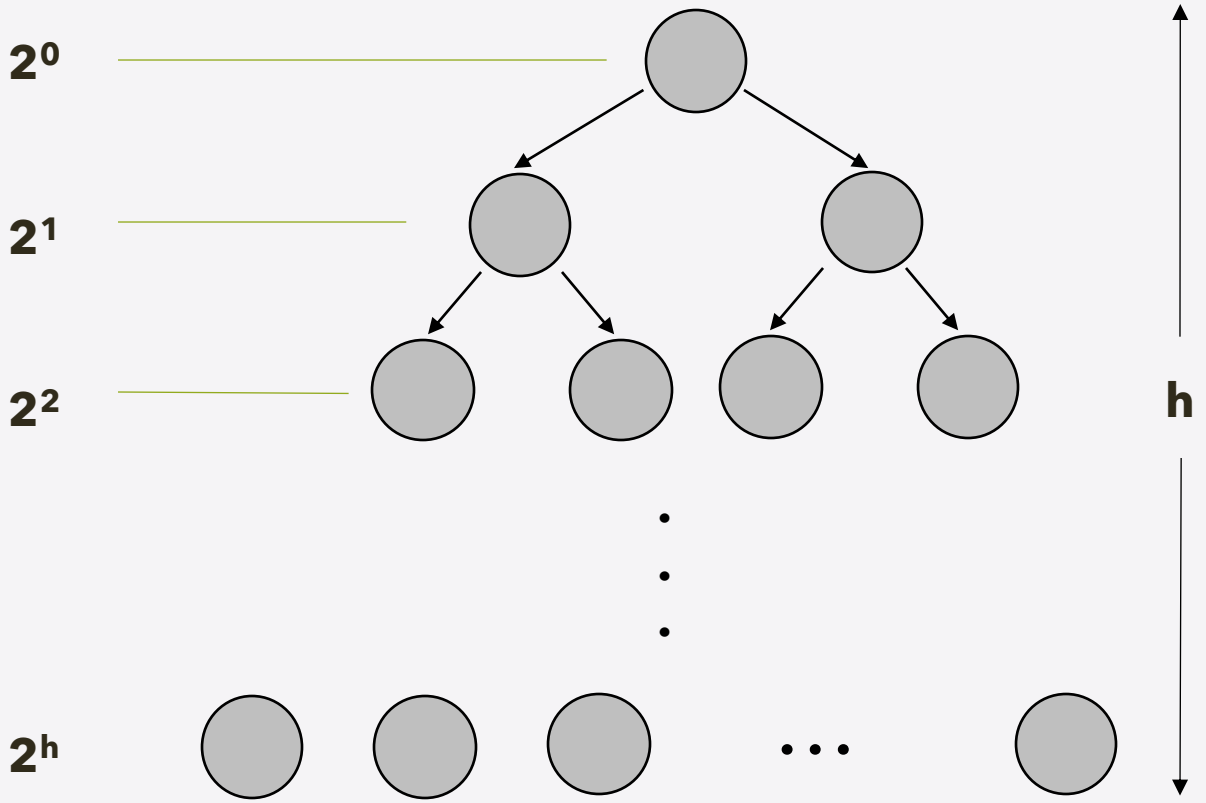
$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

Where **n = h**, so the maximum number of nodes **= $2^{h+1} - 1$**

Properties Binary trees (3/3)

- In a Binary Tree with **N** nodes, minimum possible **height** or the **minimum number of levels** is?

 # Nodes for level:



Maximum number of nodes = $2^{h+1} - 1$

The minimum possible height / minimum number of levels is

$h = \log_2(n+1) - 1$

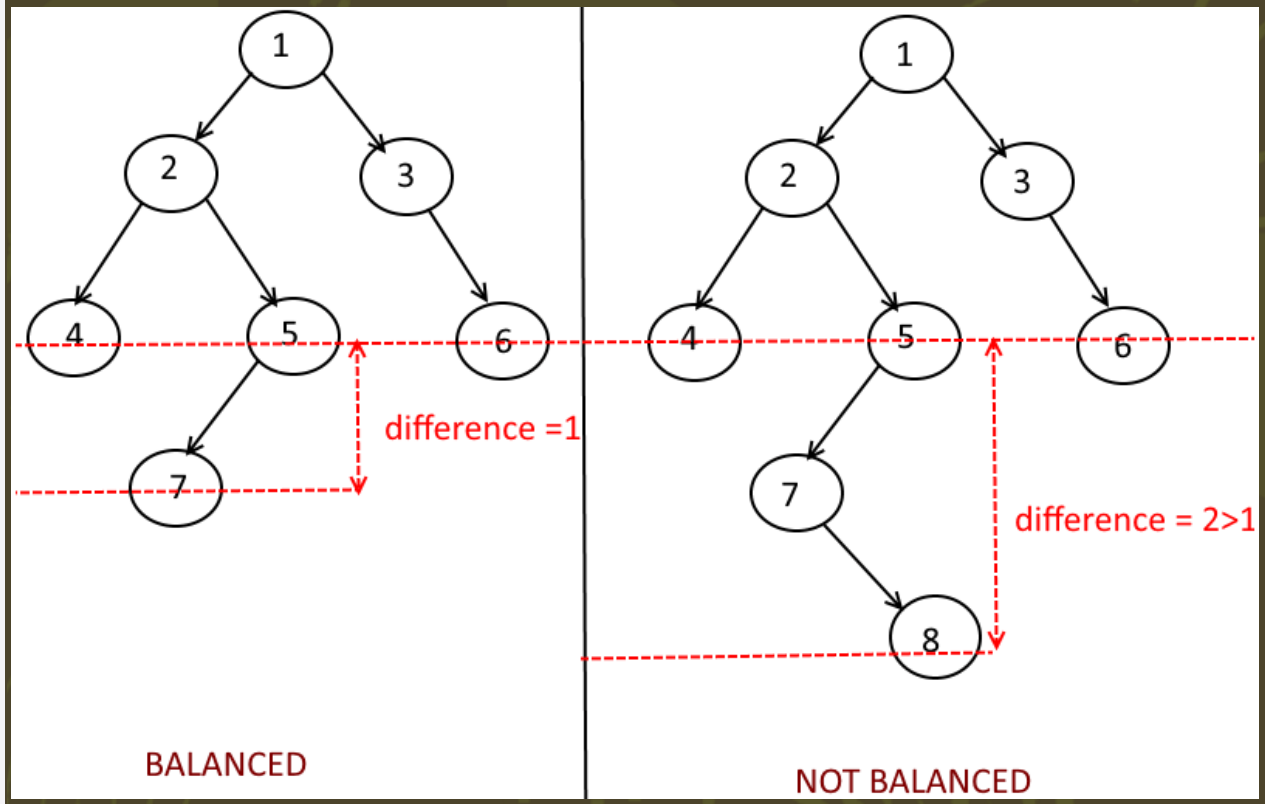
What is the maximum possible height?

Types of Binary trees

Balanced vs. Unbalance trees

Height is an aspect that influences the **complexity of the operations**, we often want to keep the height as low / small as possible, we say we want to keep the tree **balanced**.

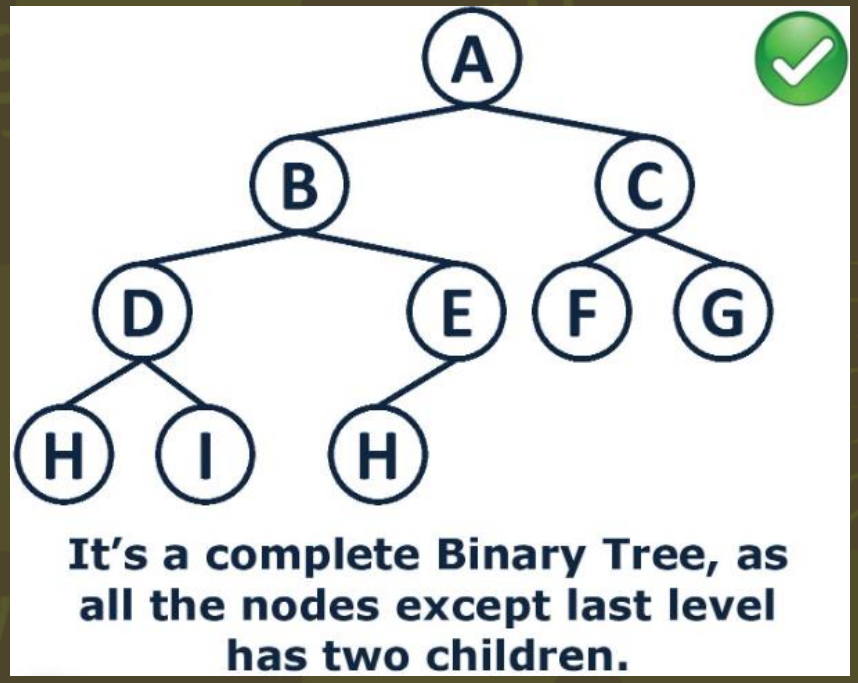
A **balanced binary tree** is a type of binary tree in which the difference between the **height** of the **left** and the **right subtree** for each node is either **0** or **1**.

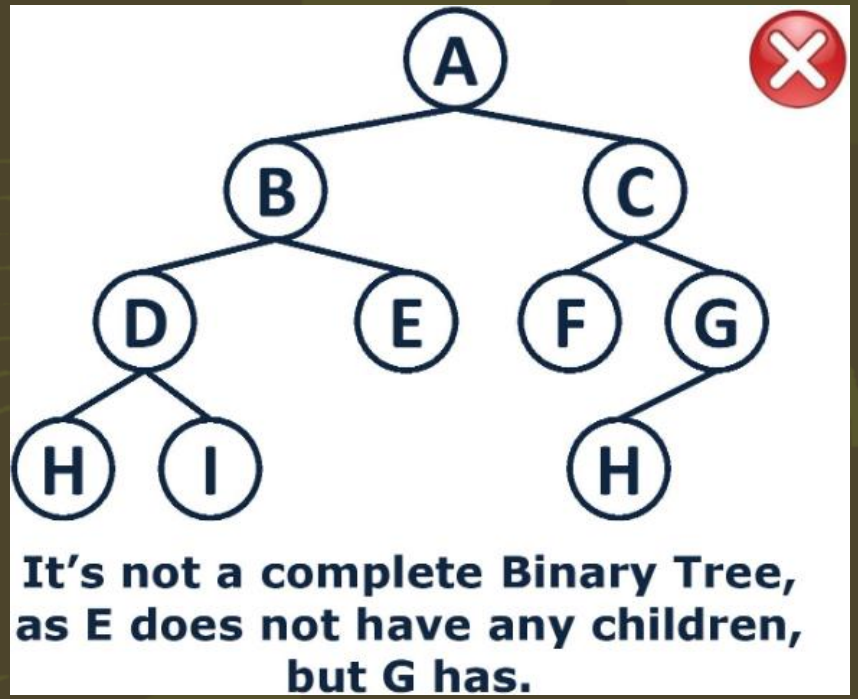


Complete Binary Trees

A **complete binary tree** is a binary tree in which **every level of the tree is fully filled**, except perhaps the **last level**.

If the **last level is not fully filled**, it must be from left to right.

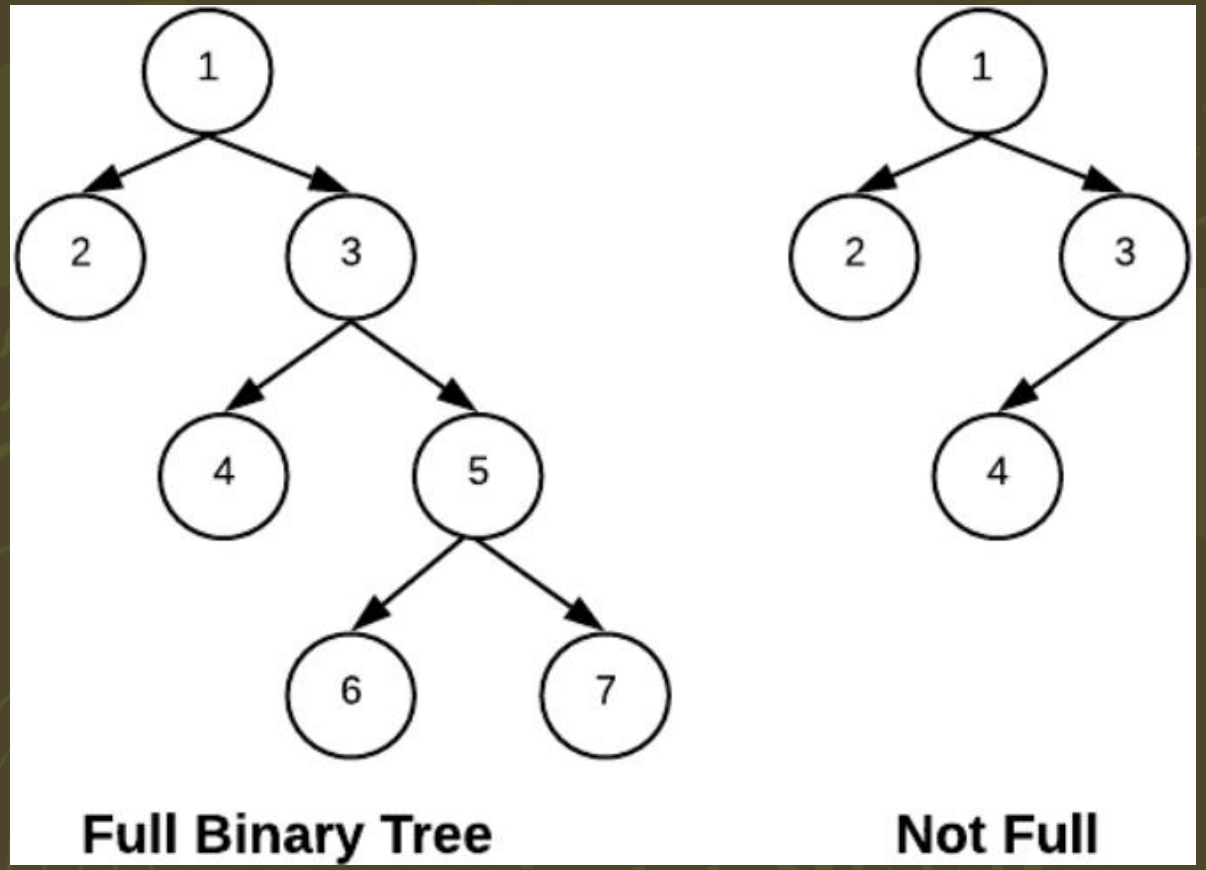




Full Binary Trees

A **full binary tree** is a binary tree in which **every node** has either **zero** or **two children**.

That is, no node has only one child.

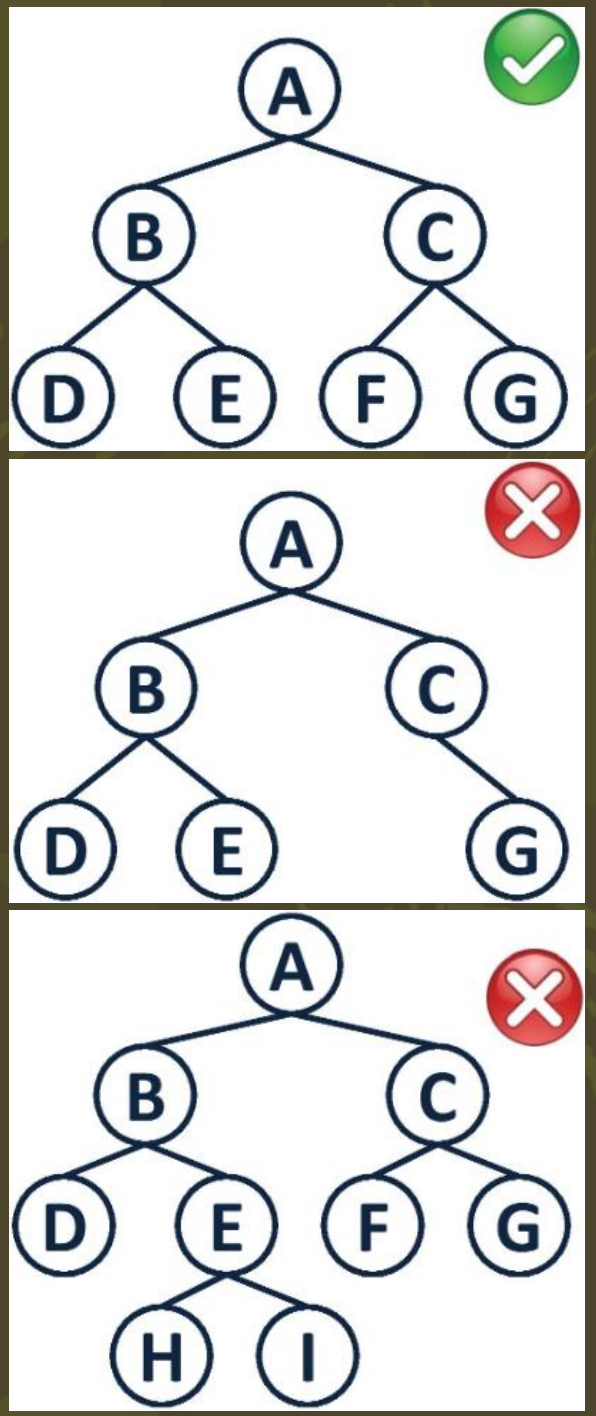


Perfect Binary Trees

A **perfect binary tree** is one that is **full** and **complete**. All **leaf nodes** will be at the same level, and this level has the maximum number of nodes.

A **perfect tree** must have exactly $2^{k+1}-1$ nodes (where **k** is the number of levels).

Complete trees guarantees the best time complexities.



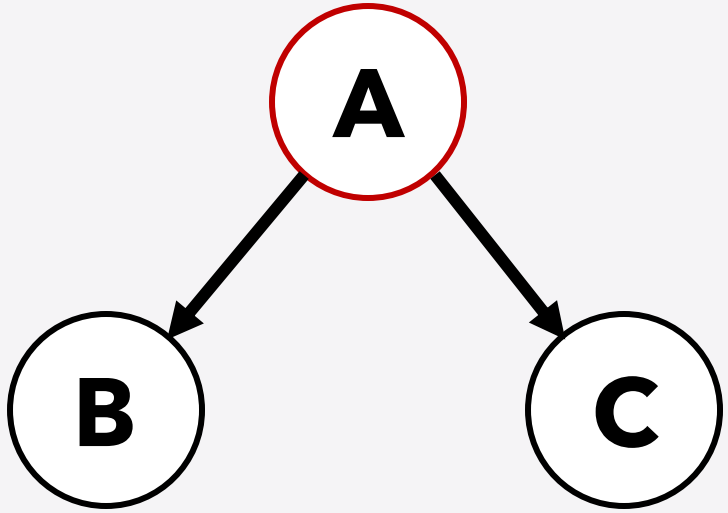
Binary Trees Traversal

Unlike **linear data structures** (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, **trees** can be traversed in different ways:

Pre-Order Traversal

It means to "*visit*" (print) the **current node**, then solve the **left** branch, and finally, the **right** branch.

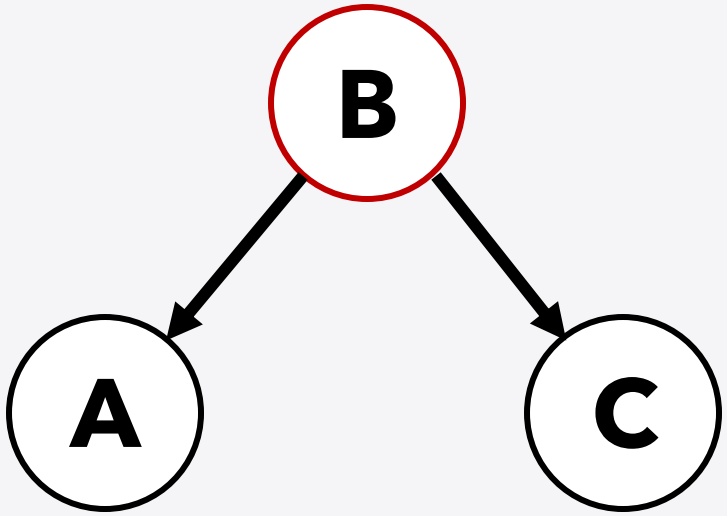
- Current node
- Left branch
- Right branch



In-Order Traversal

It means to "*visit*" (print) the **left** branch, then the **current** node, and finally, the **right** branch.

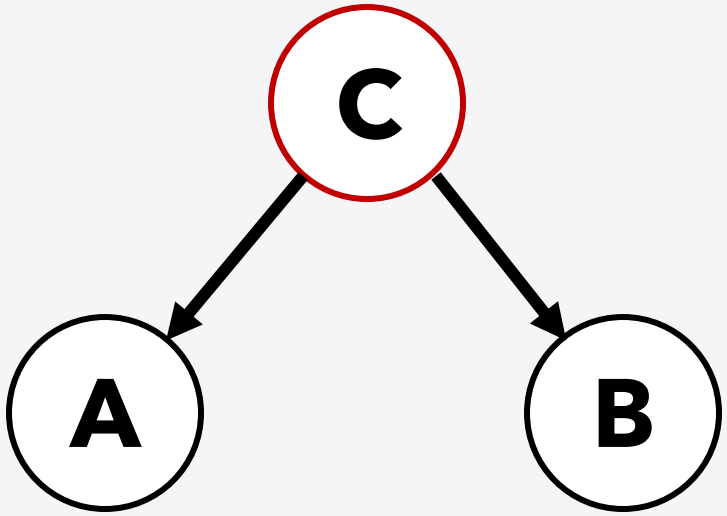
- Left branch
- Current node
- Right branch



Post-Order Traversal

It means to "*visit*" (print) the **left** branch, then the **right** branch, and finally, the **current** node.

- Left branch
- Right branch
- Current node

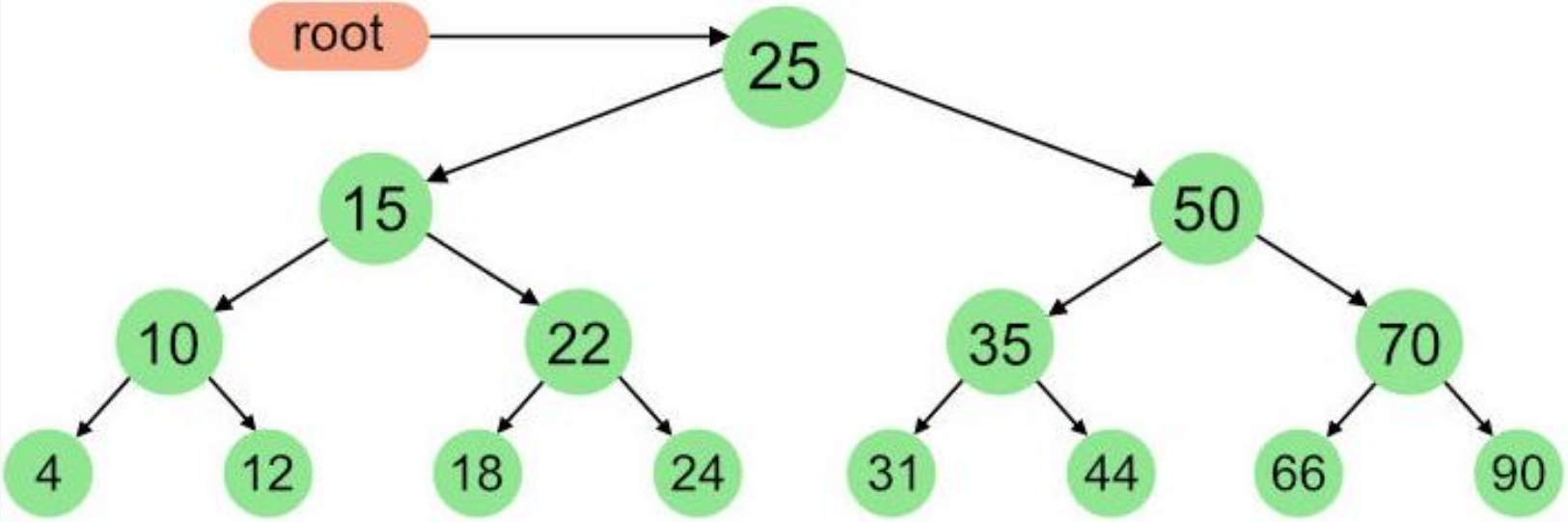


Example Binary Trees Traversal

- **Pre-Order** traversal visits the nodes in the following order:

- **In-Order** traversal visits the nodes in the following order:

- **Post-Order** traversal visits the nodes in the following order:

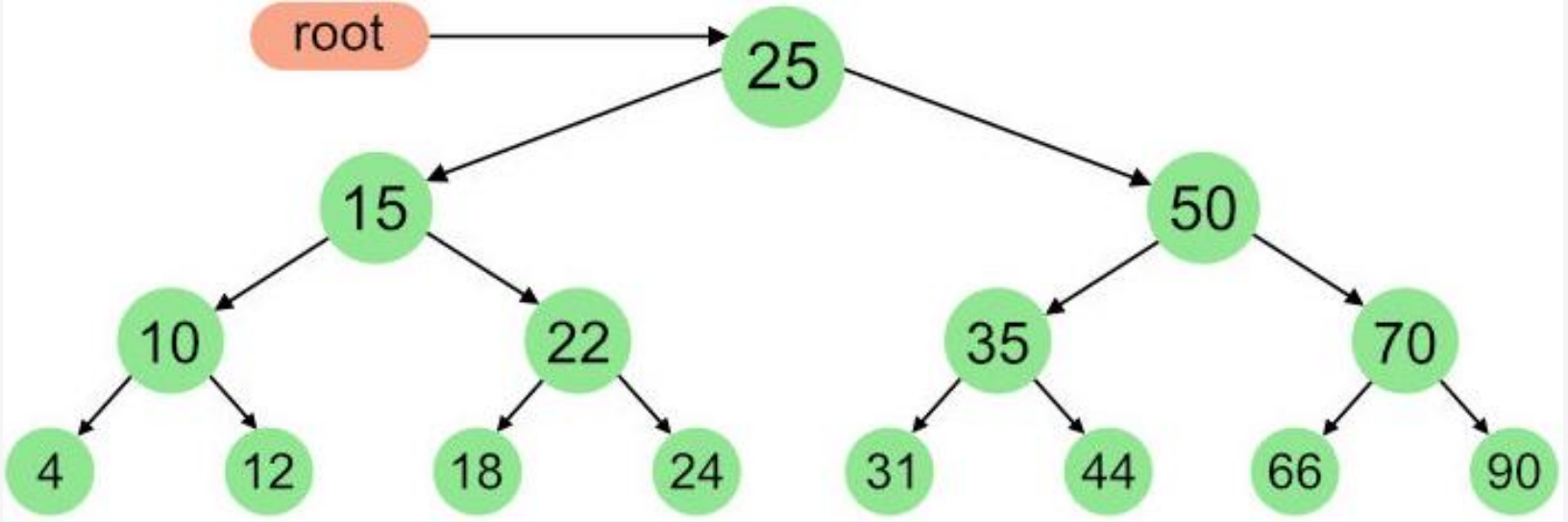


Example Binary Trees Traversal

- **Pre-Order** traversal visits the nodes in the following order:
  25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

- **In-Order** traversal visits the nodes in the following order:

- **Post-Order** traversal visits the nodes in the following order:

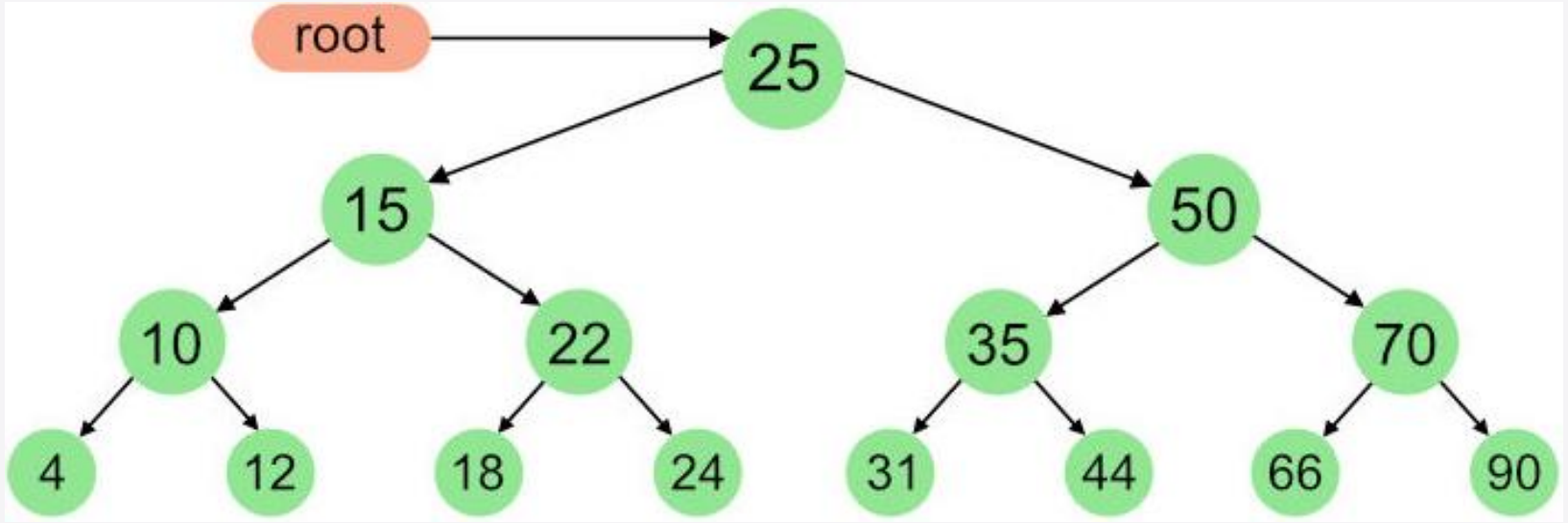


Example Binary Trees Traversal

- **Pre-Order** traversal visits the nodes in the following order:
  25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

- **In-Order** traversal visits the nodes in the following order:
  4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

- **Post-Order** traversal visits the nodes in the following order:

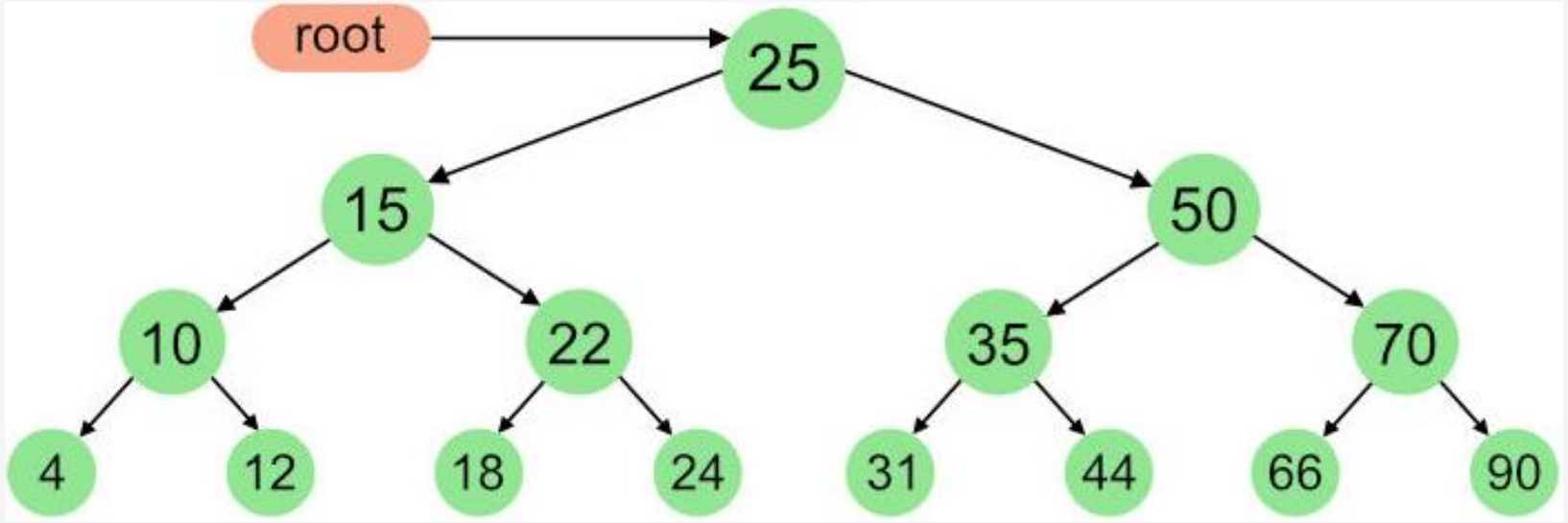


Example Binary Trees Traversal

- **Pre-Order** traversal visits the nodes in the following order:
  25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

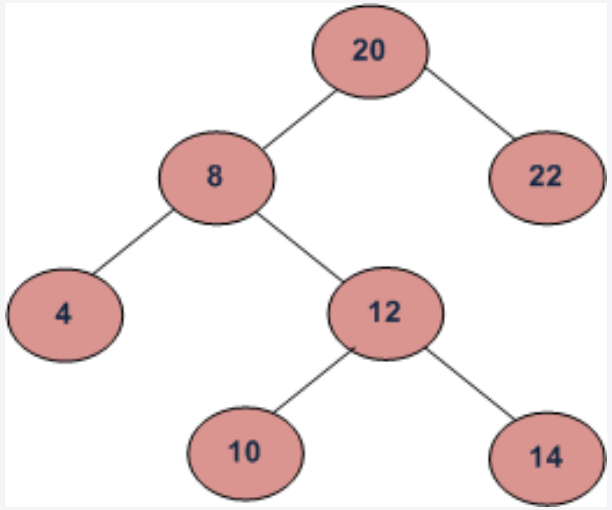
- **In-Order** traversal visits the nodes in the following order:
  4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

- **Post-Order** traversal visits the nodes in the following order:
  4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

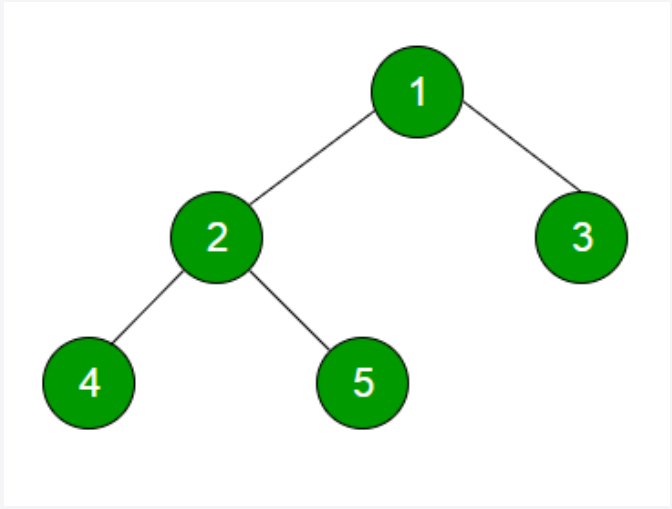


Level Order Binary trees Traversal

Given the **root** of the Binary Tree. The task is to print the **Level order traversal** of a tree is **breadth first traversal** for the tree.



Output:
20
8 22
4 12
10 14



Output:
1
2 3
4 5

Binary Search Trees

Binary Tree vs. Binary Search Tree

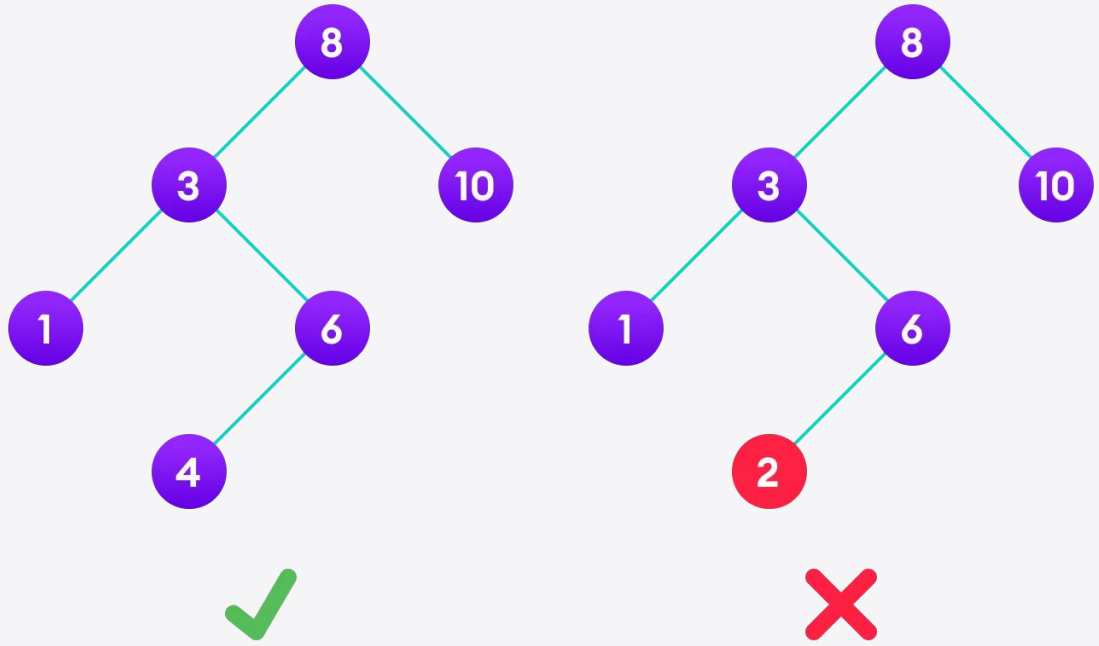
A Binary Search Tree (BST) is a binary tree in which every node fits a specific ordering property:

Left descents <= n < right descents.

This must be true for each node **n**.

Note. The definition may vary respect to equality. In some cases, the tree cannot have duplicate values.

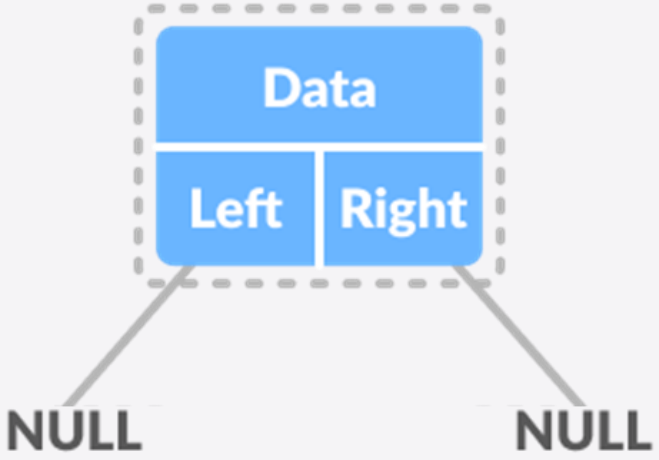
Note that this inequality must be true for all the node's descendents, not just its immediate children.



Implementation of BST - Base

First, define a **Node structure**, which represent each element in the BST.

- Here, each **node** will store the **data** and pointers to **left** and **right** subtree.




```
struct Node
{
    int data;
    Node *left,*right;

    Node(int d):data(d),left(NULL),right(NULL){}
};
```

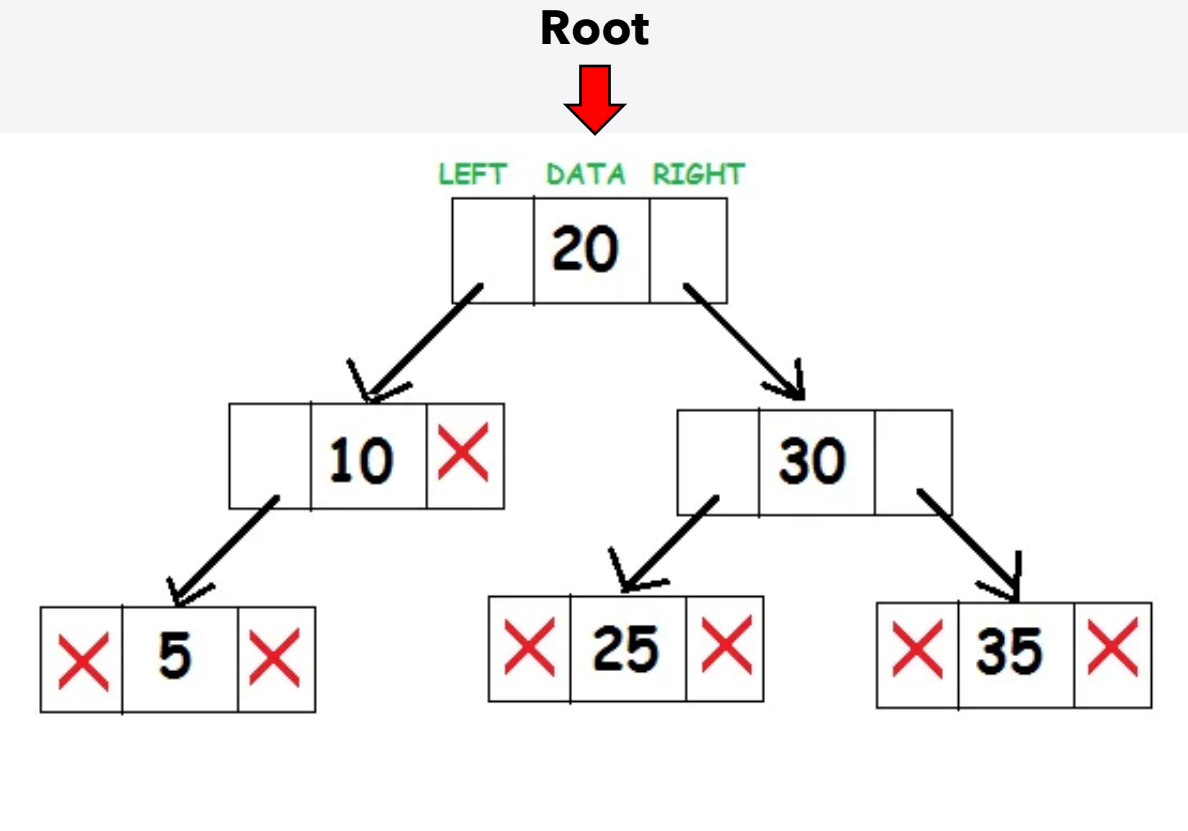
Implementation of BST - Base

Now we will recreate a **simple BST** through a **BST class** where we implement all the functionalities of this data structure.

```
class BST
{
    private:
        Node *Root;

        //Functions
        void Insert(int&, Node*&);
        void InOrder(Node*);
        void PreOrder(Node*);
        void PostOrder(Node*);
        void DeleteNode(int&, Node*&);
    public:
        BST():Root(NULL){}
        ~BST()
        {
            DeleteBST(Root);
            cout << "\nDestructor: BST deleted.\n";
        }

        //Functions
        void Insert(int &value) { Insert(value, Root); }

        void InOrder() { InOrder(Root); }
        void PreOrder() { PreOrder(Root); }
        void PostOrder() { PostOrder(Root); }
        void BFT();

        void SubstituteToMin(Node*&, Node*&);
        void DeleteNode(int &value) { DeleteNode(value, Root); }

        void DeleteBST(Node*&);
};
```

Implementation of BST - Base

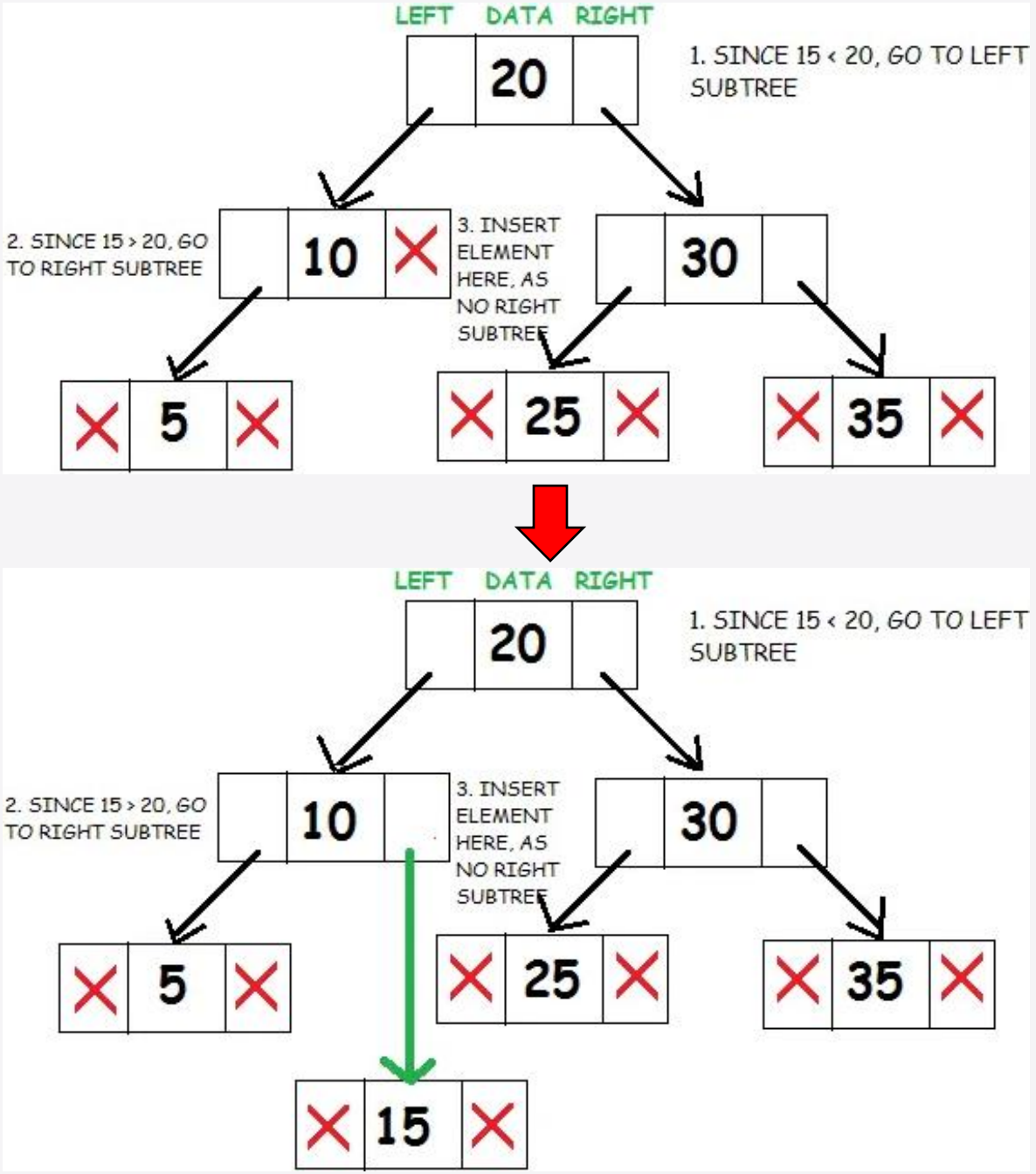
Let's assume we have the next tree:




```
struct Node
{
    int data;
    Node *left,*right;

    Node(int d):data(d),left(NULL),right(NULL){}
};

class BST
{
    private:
        Node *Root;

        //Functions
        void Insert(int&, Node*&);
        void InOrder(Node*);
        void PreOrder(Node*);
        void PostOrder(Node*);
        void DeleteNode(int&, Node*&);
    public:
        BST():Root(NULL){}
        ~BST()
        {
            DeleteBST(Root);
            cout << "\nDestructor: BST deleted.\n";
        }

        //Functions
        void Insert(int &value) { Insert(value, Root); }

        void InOrder() { InOrder(Root); }
        void PreOrder() { PreOrder(Root); }
        void PostOrder() { PostOrder(Root); }
        void BFT();

        void SubstituteToMin(Node*&, Node*&);
        void DeleteNode(int &value) { DeleteNode(value, Root); }

        void DeleteBST(Node*&);
};
```

Implementation of BST - Insert

To **insert data** into a binary tree involves a **function searching** for a space in the proper position in the tree in which to insert the **key value**.



Implementation of BST - Insert

A **recursive function** that continues moving down the BST comparing the elements:

- If the data of the **current node** is **greater**, look at the **left subtree** (if exist). Else,
- If the data of the **current node** is **smaller**, look at the **right subtree** (if exist). Else,
- Else, if is **equal**, the element is repeated.
- If the **space is available** (**NULL**) insert the new node.

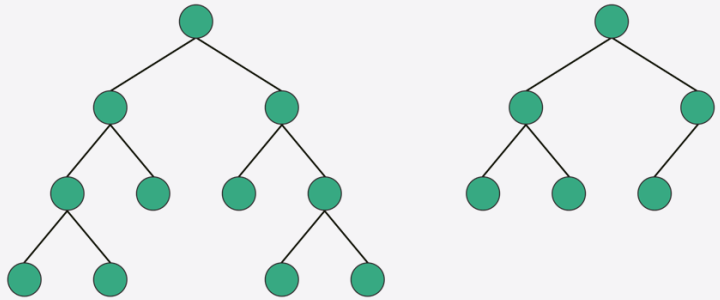
```
void Insert(int &value) { Insert(value, Root); }
```

```
//*& means receive the pointer by reference.
//An int* is a pointer to an int,
//so int*& must be a reference to a pointer to an int.
void BST::Insert(int &value, Node *&currentNode)
{
    if(currentNode == NULL)
    {
        currentNode = new Node(value);
    }
    else
    {
        if(value < currentNode->data)
            Insert(value, currentNode->left);
        else if(value > currentNode->data)
            Insert(value, currentNode->right);
        else
            cout << "Repeated element.\n";
    }
}
```

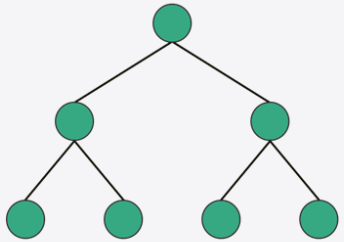
Implementation of BST - Insert

What is the time complexity?

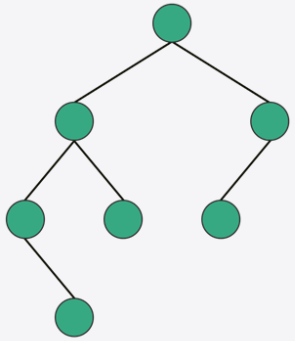
- **Best case?**



Full



Complete



Perfect

Balanced

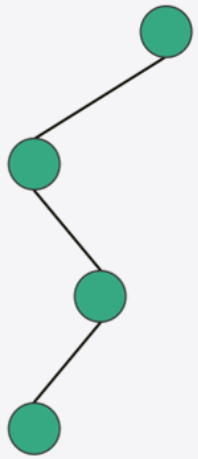
```
void Insert(int &value) { Insert(value, Root); }
```

```
//*& means receive the pointer by reference.
//An int* is a pointer to an int,
//so int*& must be a reference to a pointer to an int.
void BST::Insert(int &value, Node *&currentNode)
{
    if(currentNode == NULL)
    {
        currentNode = new Node(value);
    }
    else
    {
        if(value < currentNode->data)
            Insert(value, currentNode->left);
        else if(value > currentNode->data)
            Insert(value, currentNode->right);
        else
            cout << "Repeated element.\n";
    }
}
```

Implementation of BST - Insert

What is the time complexity?

- **Worst case?**



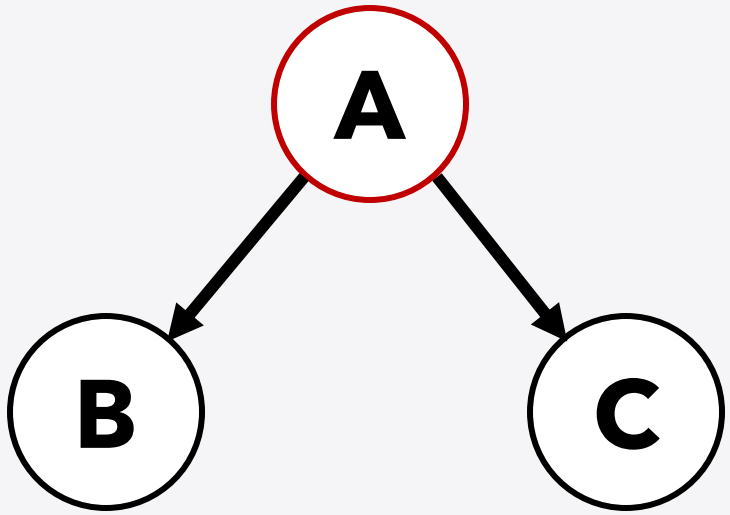
Degenerate

```
void Insert(int &value) { Insert(value, Root); }
```

```
//*& means receive the pointer by reference.
//An int* is a pointer to an int,
//so int*& must be a reference to a pointer to an int.
void BST::Insert(int &value, Node *&currentNode)
{
    if(currentNode == NULL)
    {
        currentNode = new Node(value);
    }
    else
    {
        if(value < currentNode->data)
            Insert(value, currentNode->left);
        else if(value > currentNode->data)
            Insert(value, currentNode->right);
        else
            cout << "Repeated element.\n";
    }
}
```

Implementation of BST - PreOrder

The **Pre-Order function** is generally a **recursive function** that moves through the levels of the binary tree following the designated order:

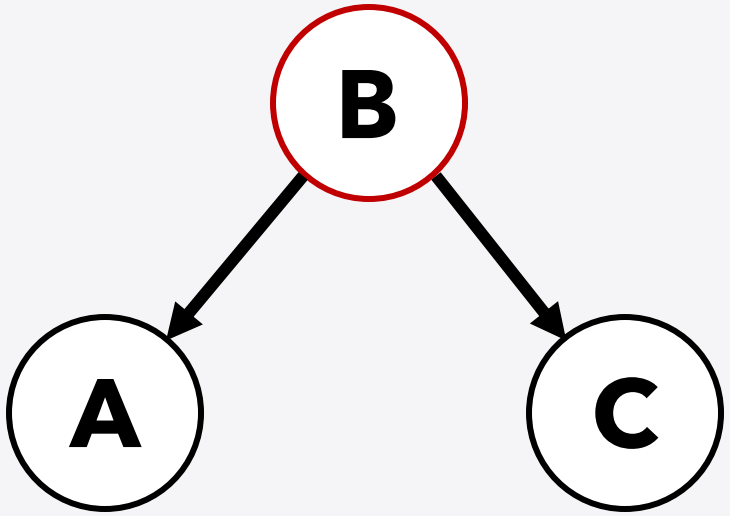



```
void PreOrder() { PreOrder(Root); }
```

```
void BST::PreOrder(Node *currentNode)
{
    if(currentNode == NULL)
        return;

    cout << currentNode->data << " ";
    PreOrder(currentNode->left);
    PreOrder(currentNode->right);
}
```

Implementation of BST - InOrder

The **In-Order function** is generally a **recursive function** that moves through the levels of the binary tree following the designated order:

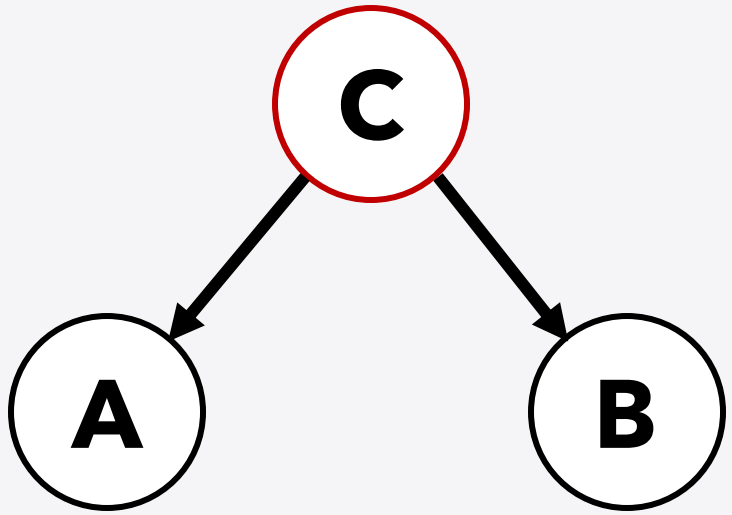



```
void InOrder() { InOrder(Root); }
```

```
void BST::InOrder(Node *currentNode)
{
    if(currentNode == NULL)
        return;

    InOrder(currentNode->left);
    cout << currentNode->data << " ";
    InOrder(currentNode->right);
}
```

Implementation of BST - PostOrder

The **Post-Order function** is generally a **recursive function** that moves through the levels of the binary tree following the designated order:



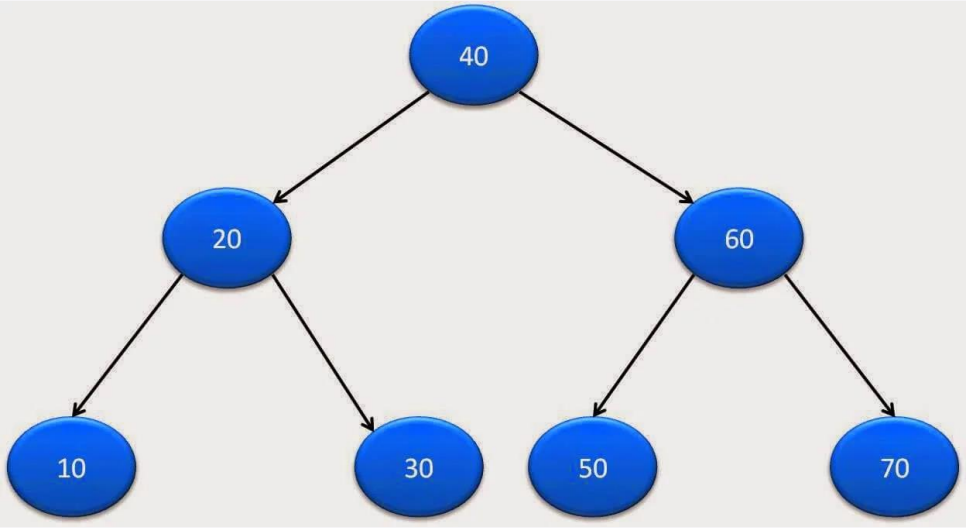

```
void PostOrder() { PostOrder(Root); }
```

```
void BST::PostOrder(Node *currentNode)
{
    if(currentNode == NULL)
        return;

    PostOrder(currentNode->left);
    PostOrder(currentNode->right);
    cout << currentNode->data << " ";
}
```

What is the time complexity?

BST – Level Order

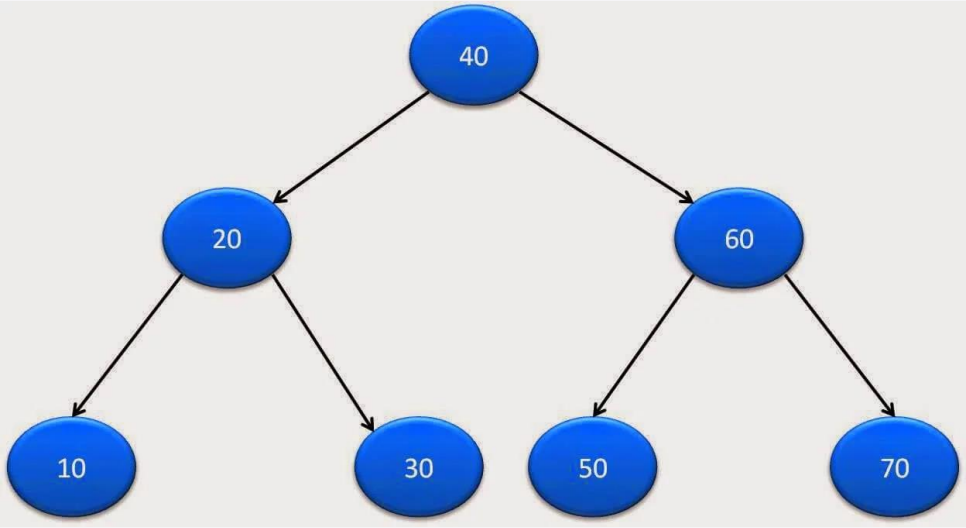
Let's say your binary tree is:

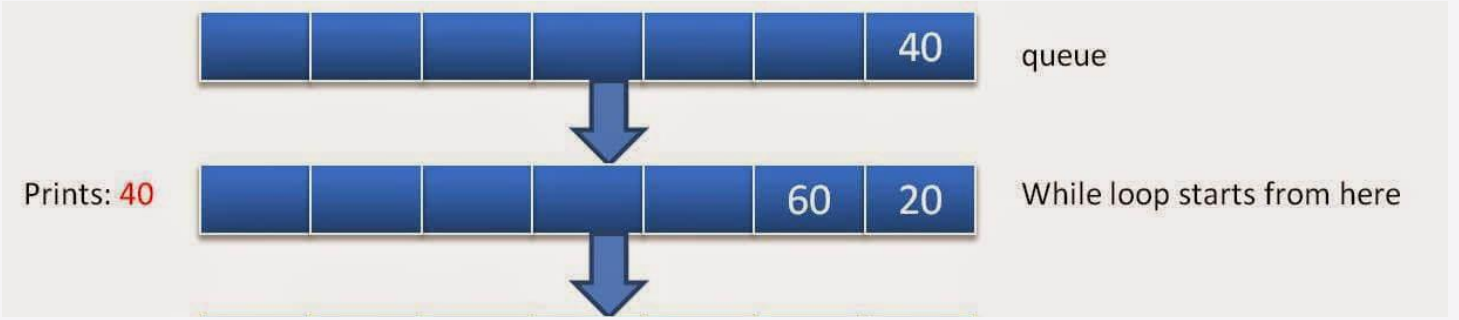


Create **empty queue** and
push **root node** to it.

BST – Level Order

Let's say your binary tree is:



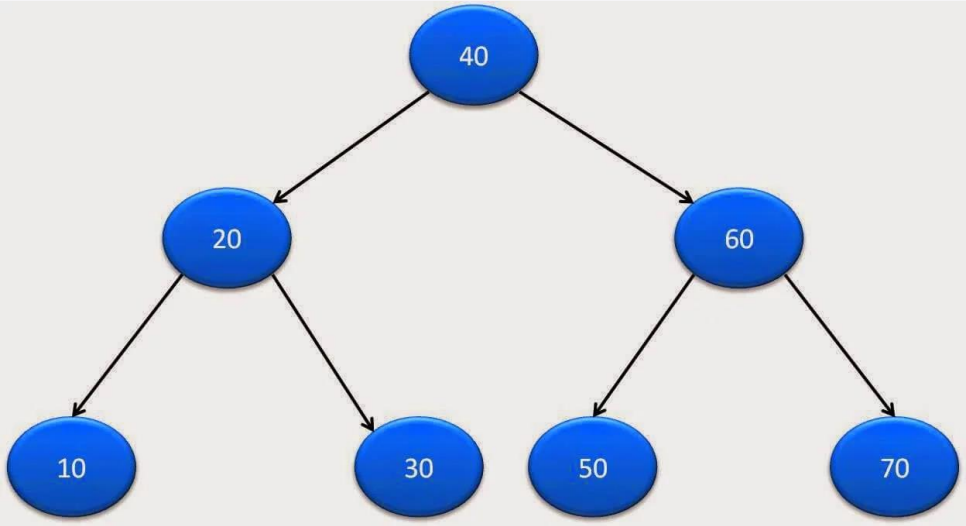


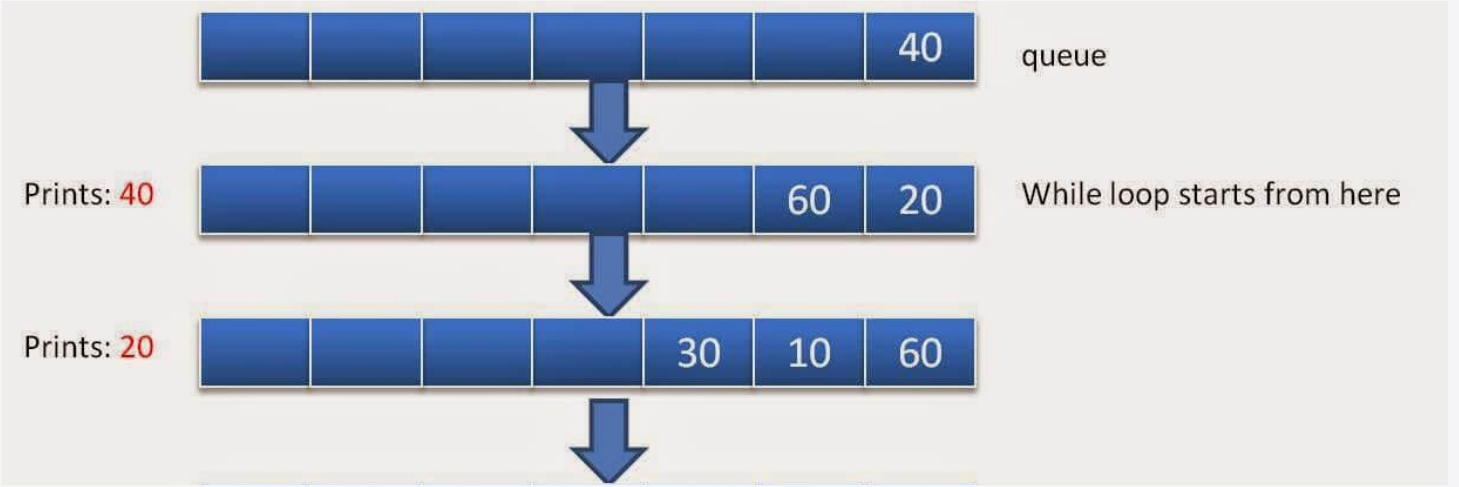
If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.

BST – Level Order

Let's say your binary tree is:



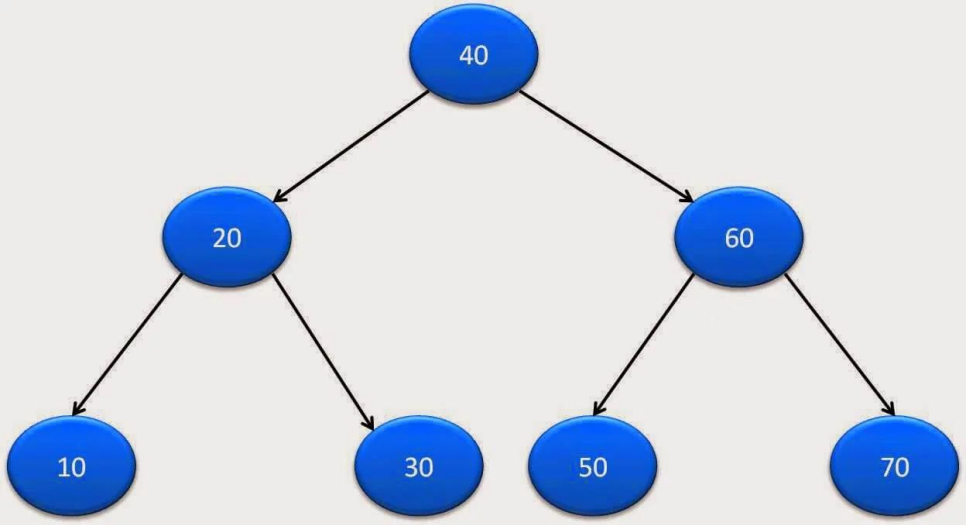


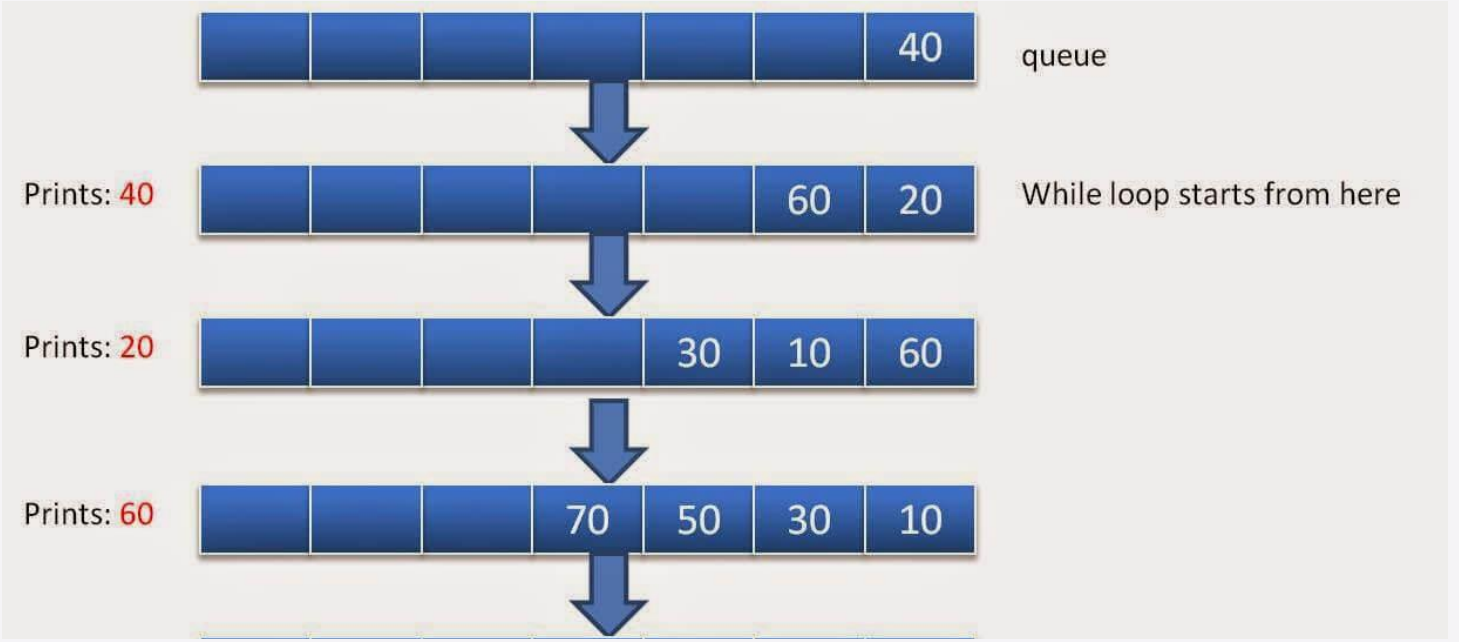
If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.

BST – Level Order

Let's say your binary tree is:



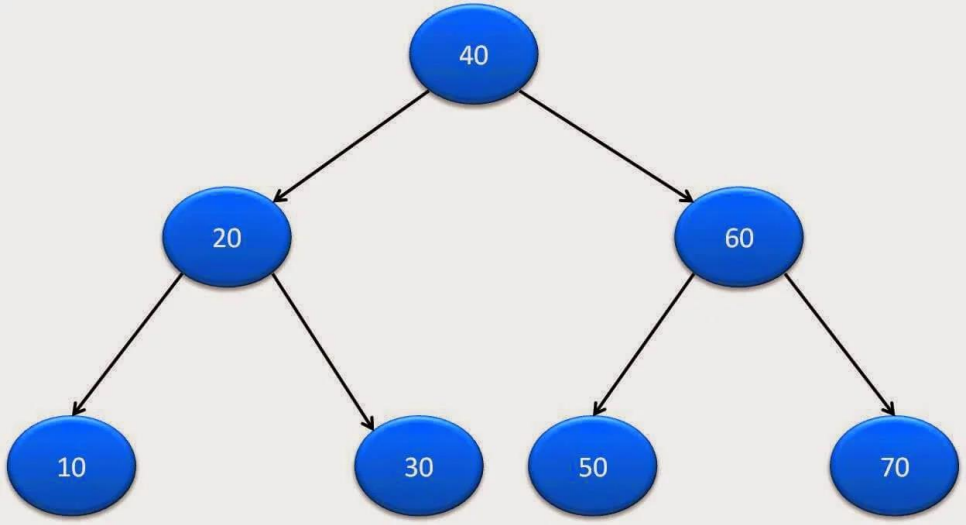


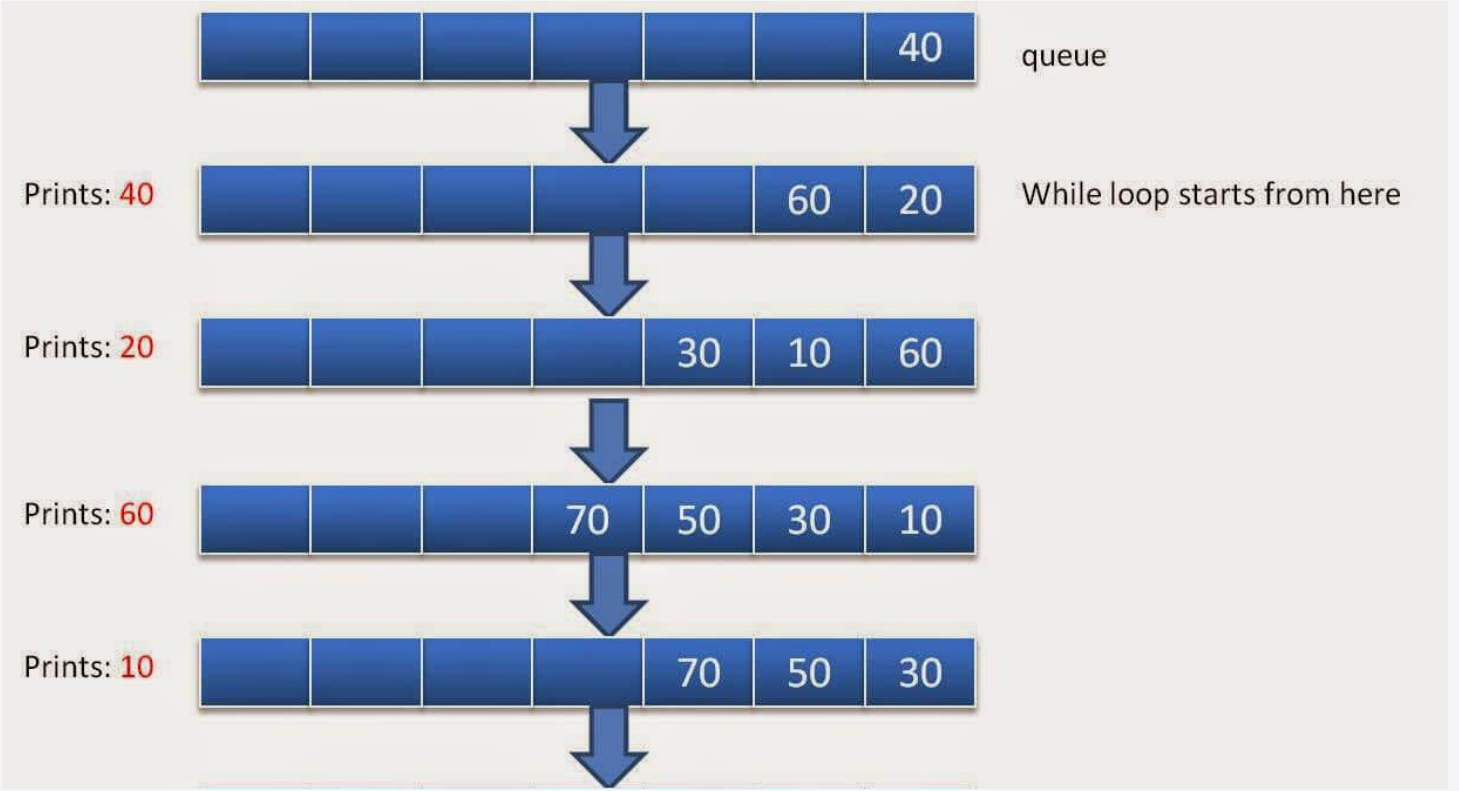
If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.

BST – Level Order

Let's say your binary tree is:



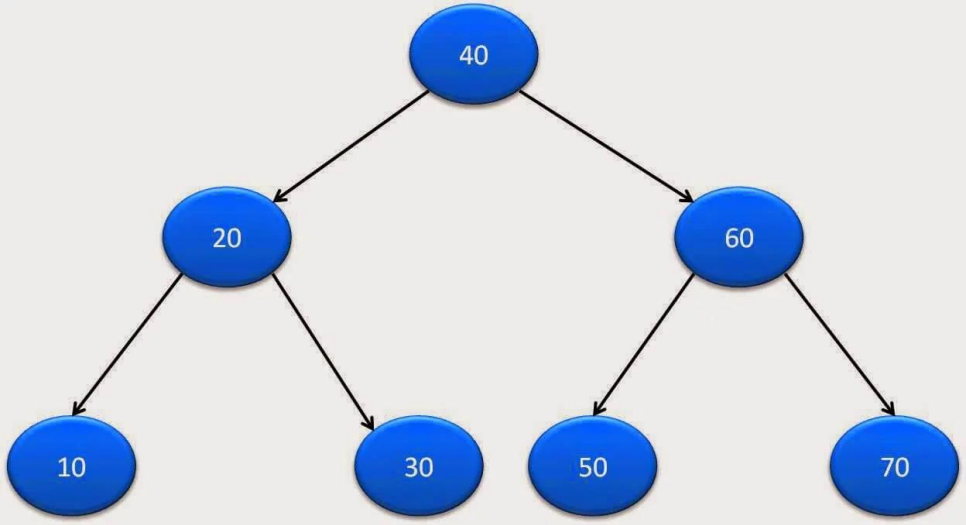


If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push** **left child** of popped node to queue if not **null**.
- **Push** **right child** of popped node to queue if not **null**.

BST – Level Order

Let's say your binary tree is:

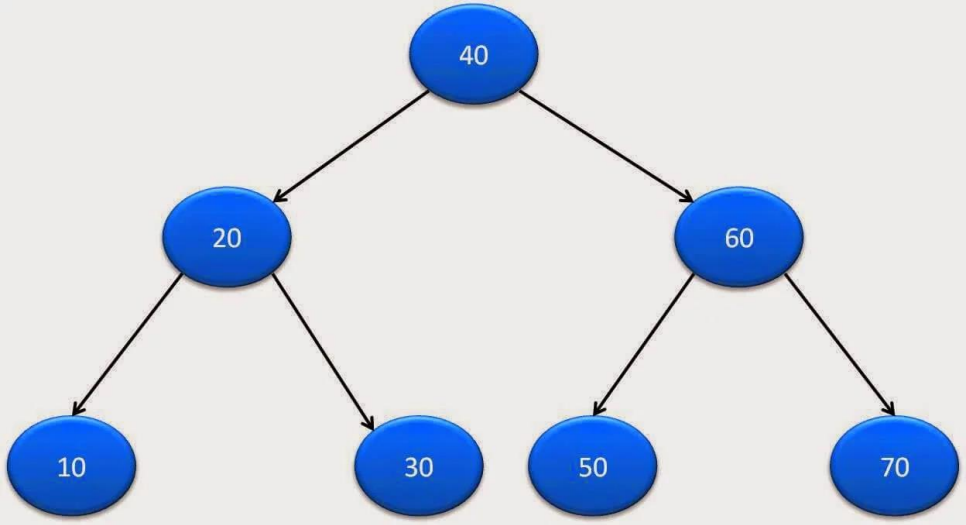


If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.

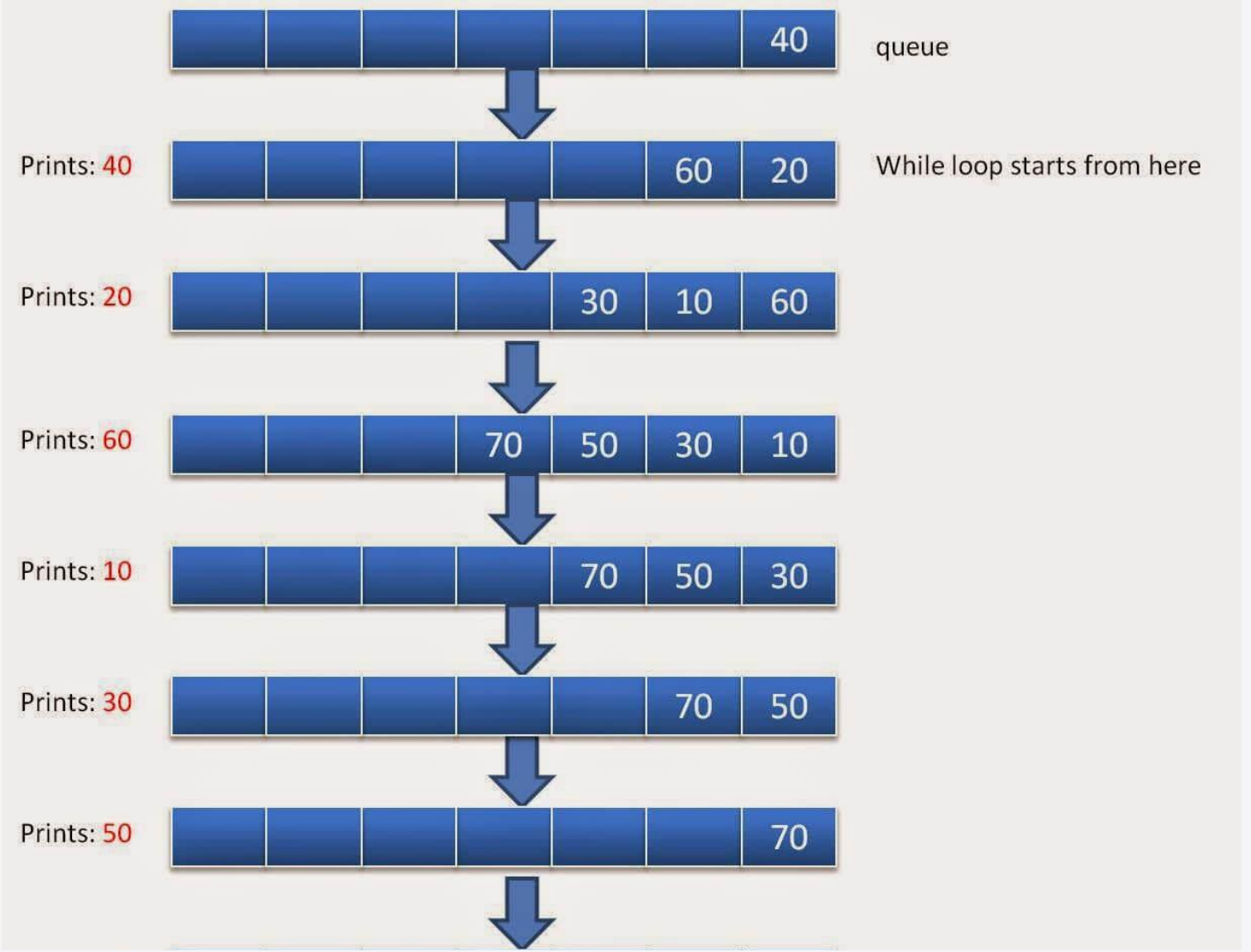
BST – Level Order

Let's say your binary tree is:



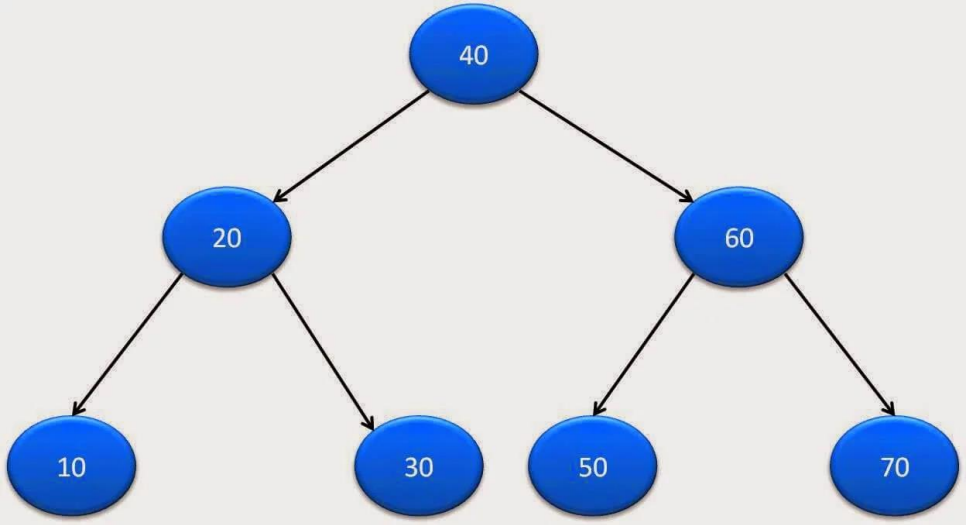
If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.



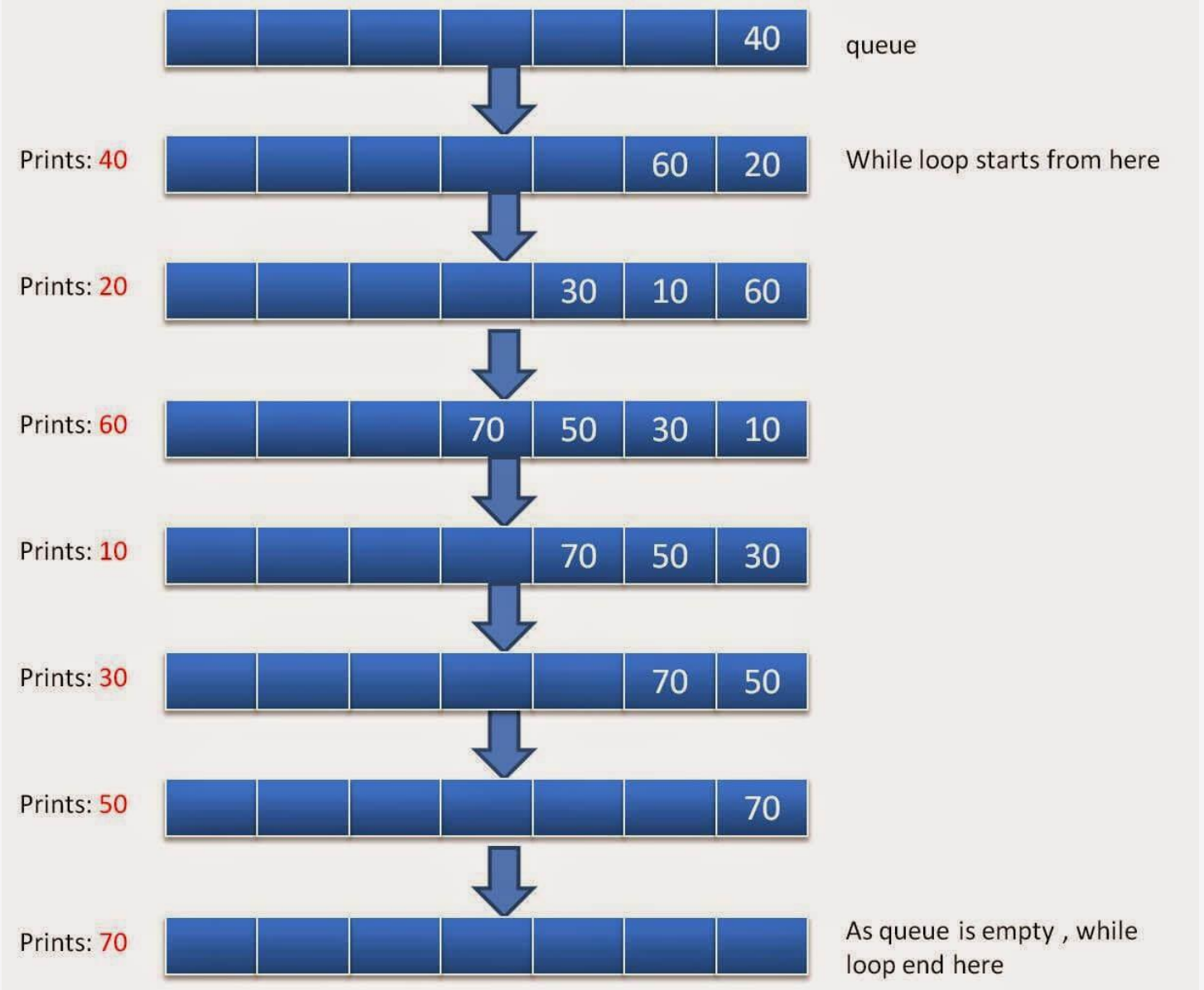
BST – Level Order

Let's say your binary tree is:



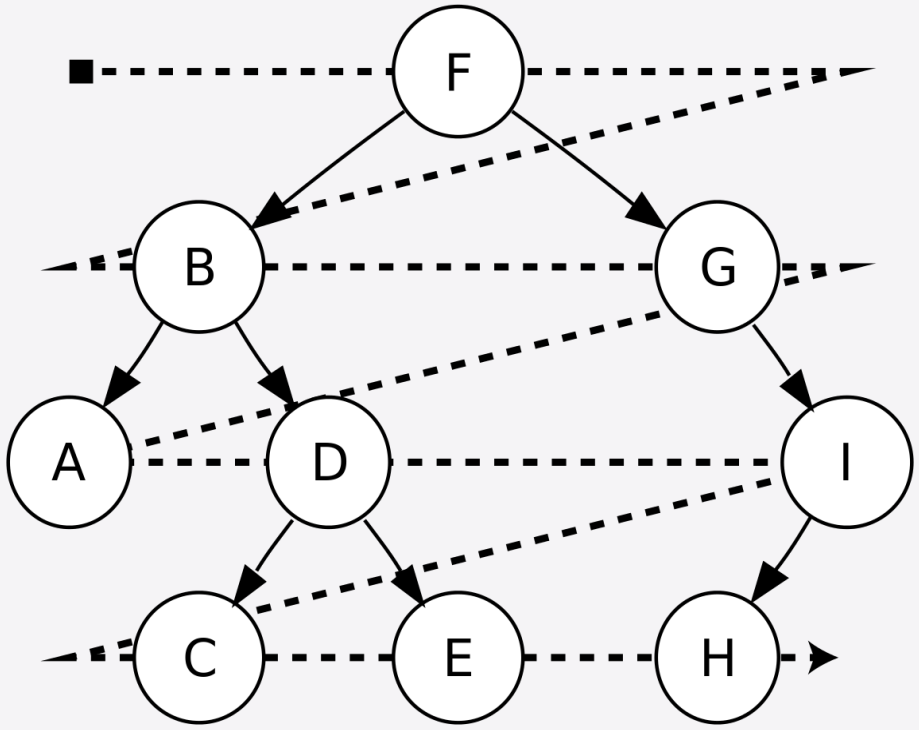
If the queue is not empty:

- **Pop** a node from **queue** and print it.
- **Push left child** of popped node to queue if not **null**.
- **Push right child** of popped node to queue if not **null**.



Implementation of BST – Level Order

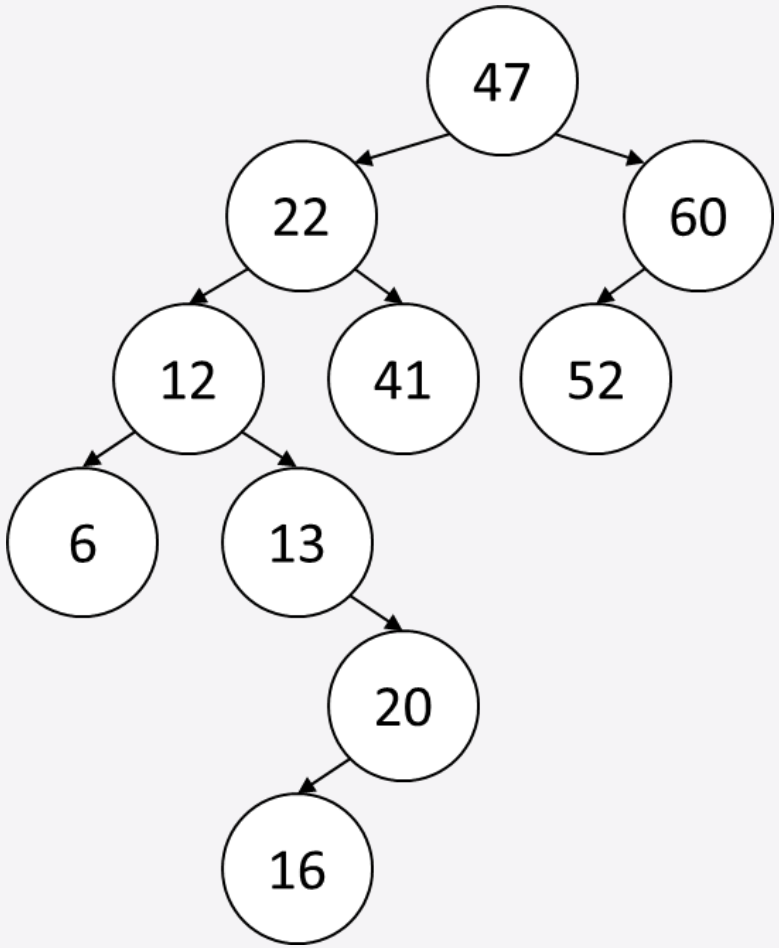
Breadth-first search (**BFS**) is a search algorithm. It starts at the **Root node** and explores all nodes at each **level** prior to moving on to the next level.




```
void BST::BFT()
{
    if(Root == NULL)
    {
        cout << "The BST is empty" << endl;
        return;
    }

    queue<Node*> Q;
    Q.push(Root);

    Node *Aux;
    while (!Q.empty())
    {
        Q.push(NULL);

        Aux = Q.front();
        while(Aux != NULL)
        {
            cout << Aux->data << " ";

            if (Aux->left != NULL)
                Q.push(Aux->left);
            if (Aux->right != NULL)
                Q.push(Aux->right);

            Q.pop();
            Aux = Q.front();
        }
        Q.pop();

        cout << endl;
    }
}
```

Display Tree - BST alternative

Input: {47,60,22,12,6,13,41,20,52,16}

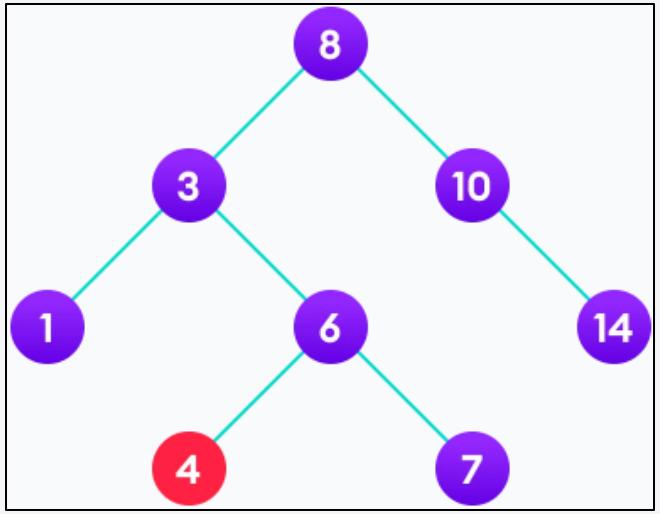



```
47(^-1)
22(<47) 60(>47)
12(<22) 41(>22) 52(<60)
6(<12) 13(>12)
20(>13)
16(<20)
```

```
void BST::Display_tree()
{
    if(Root == NULL){
        cout << "The BST is empty" << endl;
        return;
    }

    queue<Node*> Q;
    Q.push(Root);

    queue<int> parents;
    parents.push(-1);

    queue<char> dir;
    dir.push('^');

    Node *Aux;
    while (!Q.empty())
    {
        Q.push(NULL);

        Aux = Q.front();
        while(Aux != NULL)
        {
            cout << Aux->data << "(" << dir.front() << parents.front() << ") ";
            parents.pop();
            dir.pop();

            if (Aux->left != NULL)
            {
                Q.push(Aux->left);
                parents.push(Aux->data);
                dir.push('<');
            }
            if (Aux->right != NULL)
            {
                Q.push(Aux->right);
                parents.push(Aux->data);
                dir.push('>');
            }

            Q.pop();
            Aux = Q.front();
        }
        Q.pop();
        cout << endl;
    }
}
```

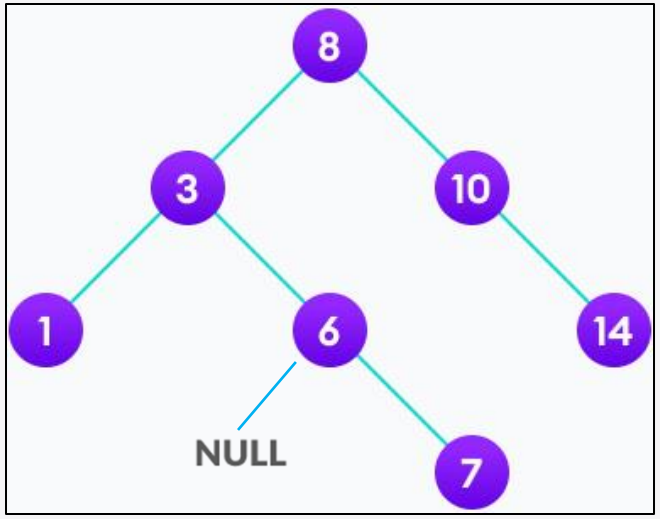
Implementation of BST - Delete

There are three cases for deleting a node from a binary search tree:



Case I

In the first case, **the node to be deleted is the leaf node** (no child nodes). In such a case, simply delete the node from the tree.

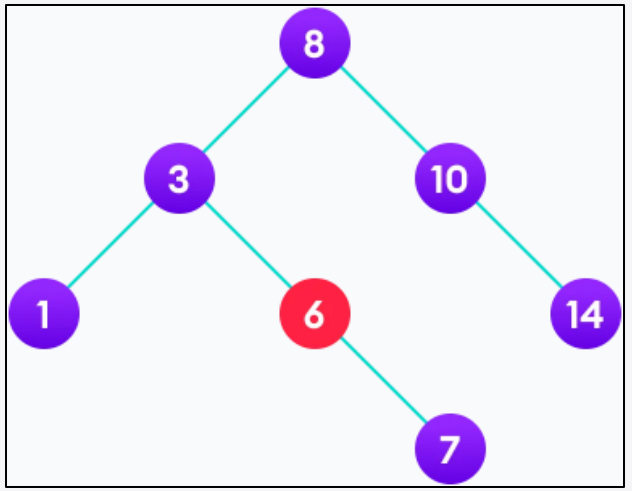


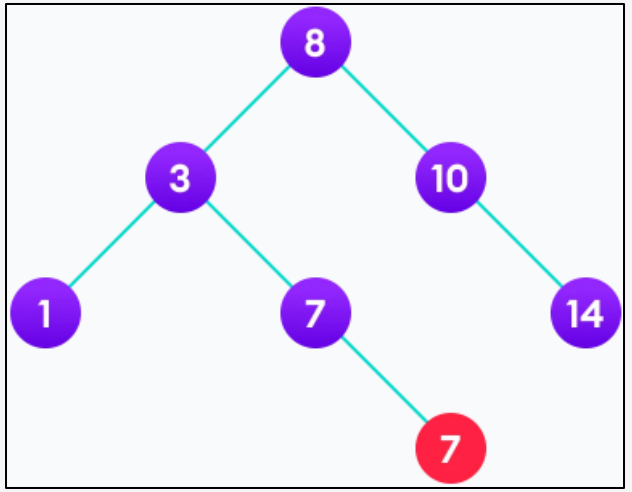
Implementation of BST - Delete

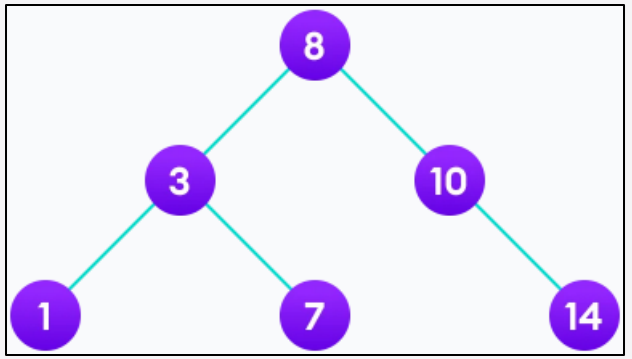
Case II

the node to be deleted lies has a single child node. In such a case follow the steps below:

- Replace that node with its **child node**.
- Remove the **child node** from its original position.







Implementation of BST - Delete

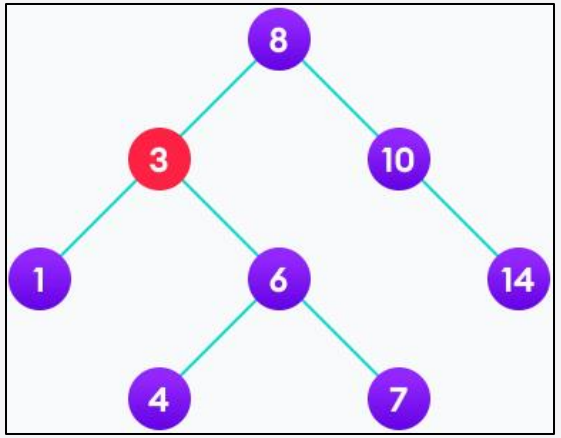
Case III

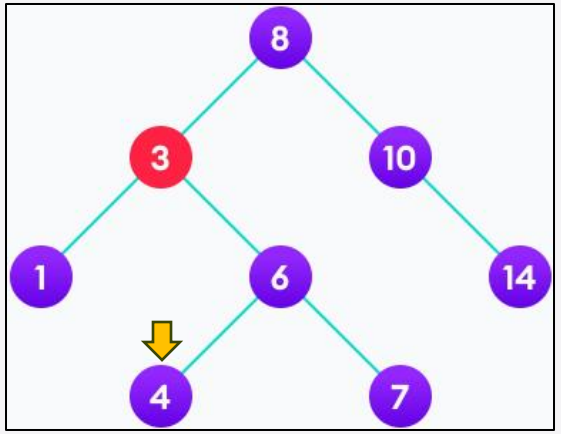
The node to be deleted has two children. In such a case follow the steps below:

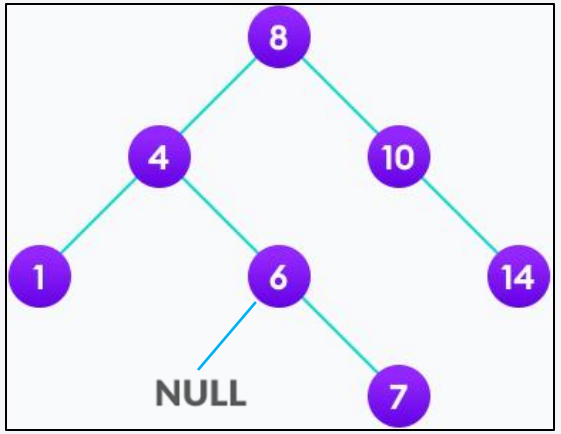
- Get the **in-order successor** of that node's right sub-tree. **The leftmost node of the right sub-tree**.
  (*also, it's possible to use the rightmost node of the left sub-tree*)

- Replace the node with the successor value.

- Remove the successor: Replace it with its **right child node** (**Node** or **NULL**).







Implementation of BST - Delete

In the deletion implementation, **search for the node** and the three possibilities of the node to be deleted are considered:

```
void BST::SubstituteToMin(Node *&aptAux, Node *&aptNode)
{
    if (aptAux->left != NULL)
        SubstituteToMin(aptAux->left, aptNode);
    else
    {
        //aptNode: Node to be deleted
        //aptAux: MinValue

        //transfer data from the substitude Node
        aptNode->data = aptAux->data;
        //save the minValue Node (To delete)
        aptNode = aptAux;
        //swap place minValue-right (Node or Null)
        aptAux = aptAux->right;
    }
}
```

```
void BST::DeleteNode(int &value, Node *&currentNode)
{
    if(currentNode == NULL)
    {
        cout << "The BST is empty" << endl<< endl;
    }
    else
    {
        if (value < currentNode->data)
            DeleteNode(value, currentNode->left);
        else if (value > currentNode->data)
            DeleteNode(value, currentNode->right);
        else
        {
            Node *apAux = currentNode;

            if (apAux->right == NULL) //only left child Node
                currentNode = apAux->left;
            if (apAux->left == NULL) //only right child Node
                currentNode = apAux->right;
            if (apAux->left != NULL && apAux->right != NULL) //two child Nodes
                SubstituteToMin(currentNode->right, apAux);

            cout << "\nThe key to delete: "<< value << endl;
            cout << "The element was deleted with the key: "<< apAux->data << endl;
            delete apAux;
        }
    }
}
```

Implementation of BST - Delete BST

The implementation to delete the entire BST (called in the **destructor** of the class) use the **Post-Order Traversal**, which **visits every leaf node first and deletes them**.

This function also can be used to delete **subtrees**, beginning from the input node.

```
void BST::DeleteBST(Node *&currentNode)
{
    // Post-Order Traversal
    if(!currentNode)
        return;

    DeleteBST(currentNode->left);
    DeleteBST(currentNode->right);
    delete currentNode;
}
```

Program body

```
int main()
{
    system("cls");

    BST A;

    vector<int> v = {47,60,22,12,6,13,41,20,52,16};
    for(int i: v)
    {
        A.Insert(i);
    }

    cout << "In-Order: ";
    A.InOrder();
    cout << "\nPre-Order: ";
    A.PreOrder();
    cout << "\nPost-Order: ";
    A.PostOrder();
    cout << "\nBreadth-First Traversal:" << endl;
    A.BFT();

    int a = 16; //Leaf Node
    A.DeleteNode(a);
    A.BFT();

    a = 13; //Only one child Node
    A.DeleteNode(a);
    A.BFT();

    a = 47; //two child Nodes (Root)
    A.DeleteNode(a);
    A.BFT();

    return 0;
}
```

Result:

```
In-Order: 6 12 13 16 20 22 41 47 52 60
Pre-Order: 47 22 12 6 13 20 16 41 60 52
Post-Order: 6 16 20 13 12 41 22 52 60 47
Breadth-First Traversal:
47
22 60
12 41 52
6 13
20
16

The key to delete: 16
The element was deleted with the key: 16
47
22 60
12 41 52
6 13
20

The key to delete: 13
The element was deleted with the key: 13
47
22 60
12 41 52
6 20

The key to delete: 47
The element was deleted with the key: 52
52
22 60
12 41
6 20

Destructor: BST deleted.
```

Binary Search Tree Complexities

Time Complexity

Here, **n** is the number of nodes in the tree.

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Space Complexity

The space complexity for all the operations is **O(n)**.

Binary Search Tree Applications

Some of the applications of a BST are:

- In multilevel indexing in the database
- For dynamic sorting
- For managing virtual memory areas in Unix kernel

Act 3.1
- Advanced operations in BST

What do I have to do?

Individually, implement an **Binary Search Tree (BST)** with the fundamental functionalities:

- **Visit** (PreOrder, InOrder, PostOrder, Level-by-level)
- **Height** (return the max height of the BST)
- **Ancestors** (display the ancestors of a specific node)
- **WhatLevelAmI**

(return level number of a specific node, **-1** if not in the BST)

- **Delete All** (Free up all the dynamic memory space used)

ACT 3.4 - Comprehensive BST activity (competence evidence)

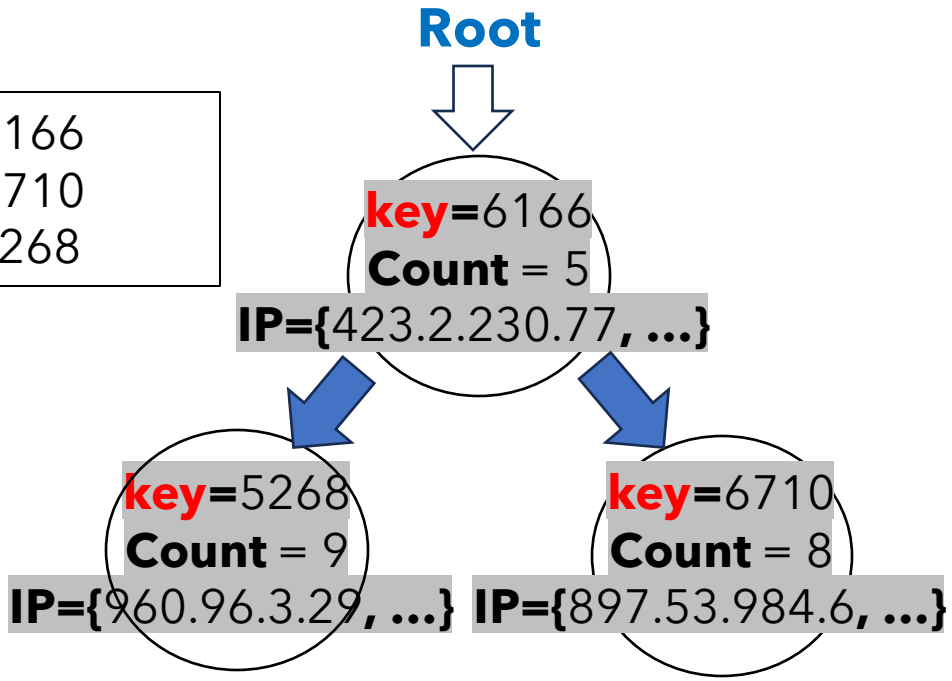
What do We have to do?

In teams of three, make an application that:

1. Open the input file and add all the entries in a BST type structure where the **key** will be the **port number** and the **values** are the **IPs addresses** that access in that port and the **total number of accesses**.
2. Do a search to find the **five Ports with most accesses** and display the information of the node.

```
Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest
Aug 28 23:07:49 897.53.984.6:6710 Failed password for root
Aug 4 03:18:56 960.96.3.29:5268 Failed password for admin...
```

IP: 423.2.230.77 **Port:** 6166
IP: 897.53.984.6 **Port:** 6710
IP: 960.96.3.29 **Port:** 5268

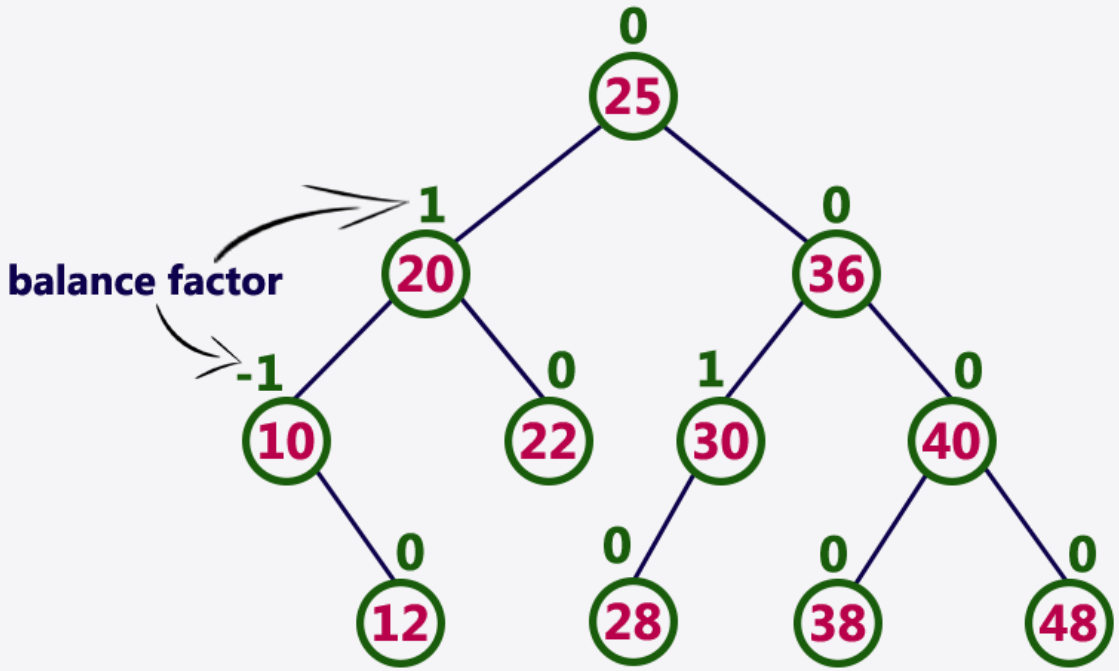


AVL Trees

AVL Tree

AVL tree is a **self-balancing binary search tree** in which each node maintains extra information called a **balance factor** whose value is either **-1**, **0** or **+1**.

AVL tree got its name after its inventor **Georgy Adelson-Velsky** and **Landis**.



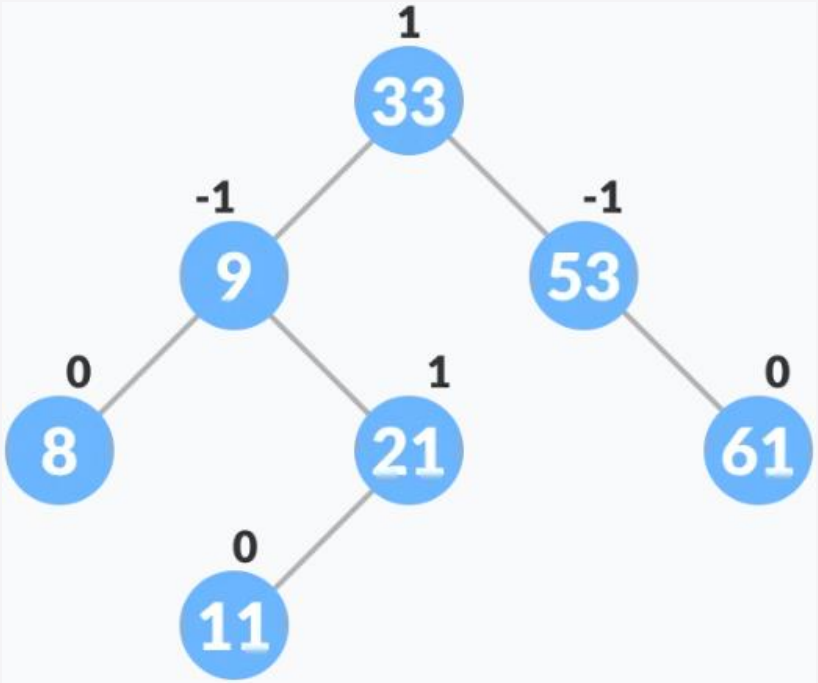
Balance factor of a node in an AVL tree is the difference between **the height of the left subtree and that of the right subtree** of that node.

Balance Factor = (**Height of Left Subtree** - **Height of Right Subtree**)

or

Balance Factor = (**Height of Right Subtree** - **Height of Left Subtree**)

The value of balance factor should always be **-1**, **0** or **+1**. An example of a balanced avl tree is:



Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

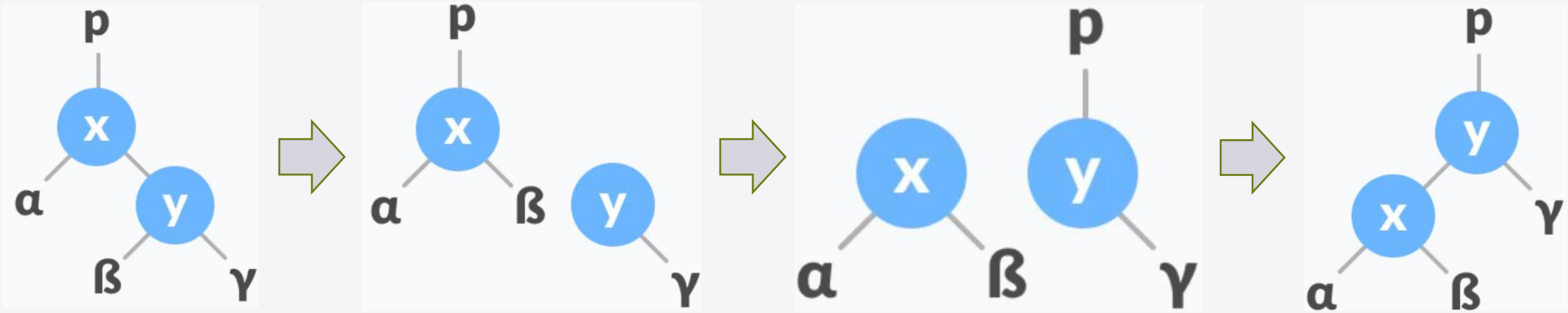
In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

- **Left Rotate**
- **Right Rotate**
- **Left-Right & Right-Left Rotate**

- Left Rotate

In left-rotation, **the arrangement of the nodes on the right** is transformed into **the arrangements on the left node**.

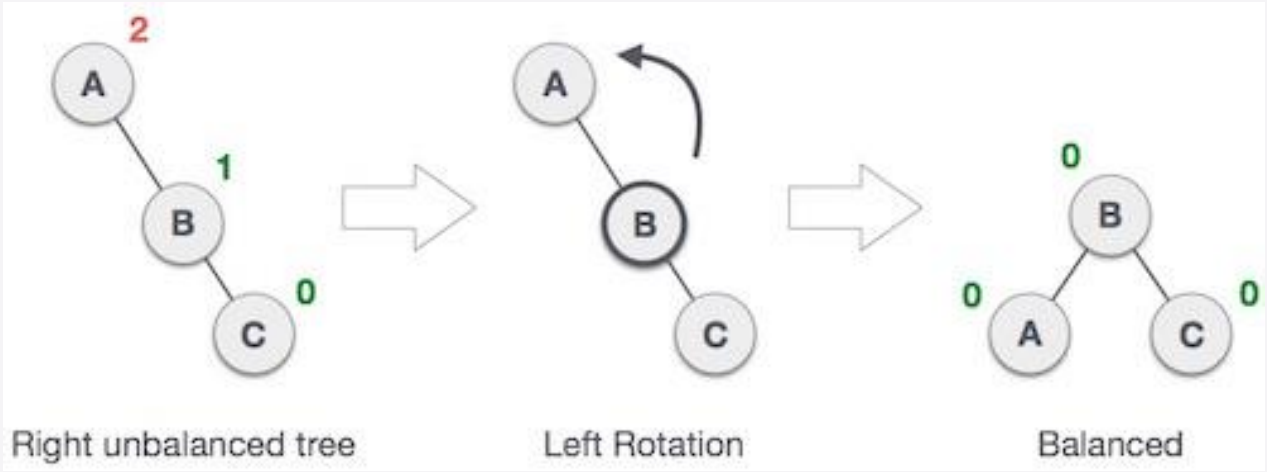


Implementation of AVL – Left Rotate

If a **tree becomes unbalanced**, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.

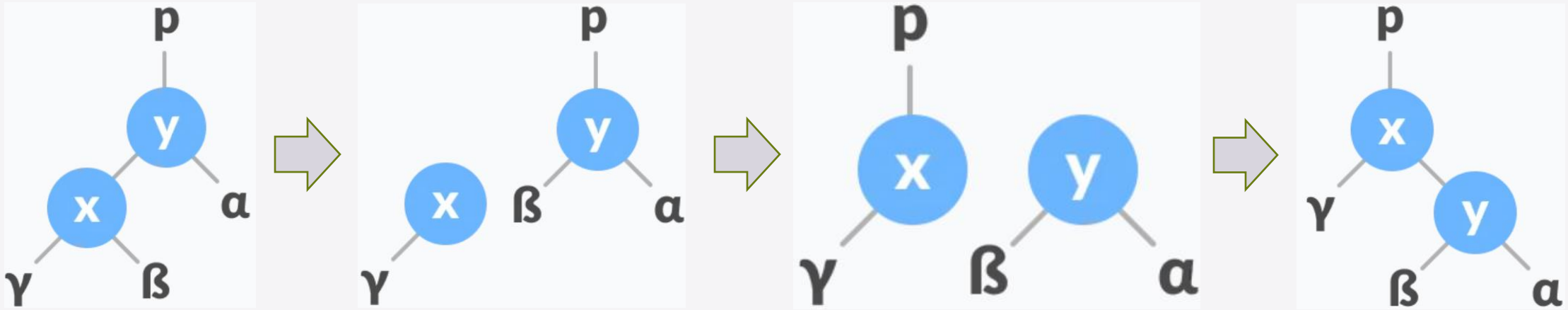
In our example, **node A** has become unbalanced as a node is inserted in the right subtree of **A's right subtree**. We perform the left rotation by making **A** the **left-subtree of B**.

```
Node *AVL::LeftRotate(Node *&node)
{
    //Save right child Node
    Node *apAux = node->right;
    //swap right child for child's left subtree
    node->right = apAux->left;
    //swap child's left subtree for node
    apAux->left = node;

    //Update node and left child's height (Only those change their height)
    node->height = max(Height(node->left), Height(node->right)) + 1;
    apAux->height = max(Height(apAux->left), Height(apAux->right)) + 1;

    return apAux;
}
```



- Right Rotate

In Right-rotation, **the arrangement of the nodes on the left** is transformed into **the arrangements on the right node**.

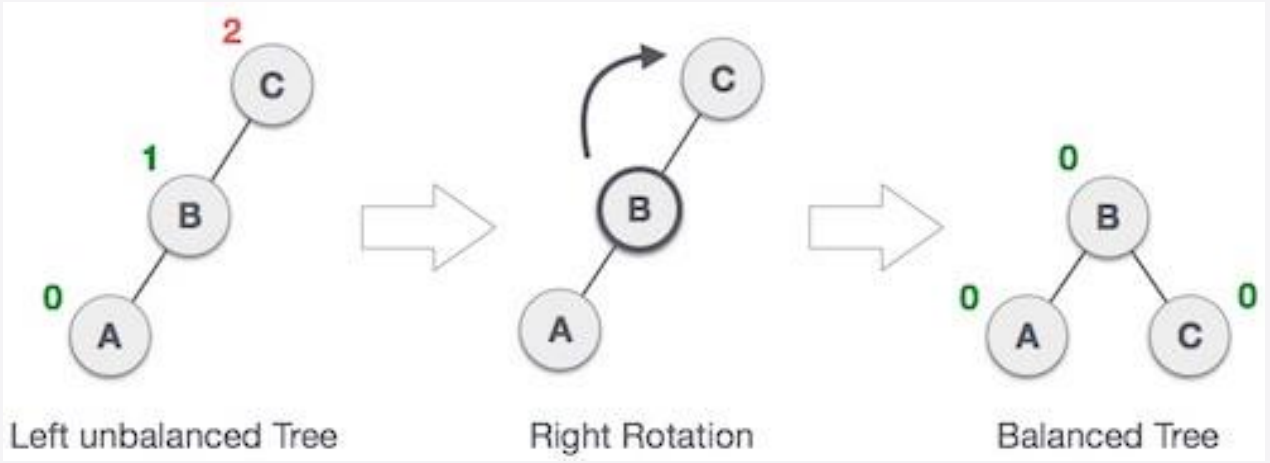


Implementation of AVL – Right Rotate

AVL tree may become unbalanced, if a node is inserted in the **left subtree** of the **left subtree**. The tree then needs a right rotation.

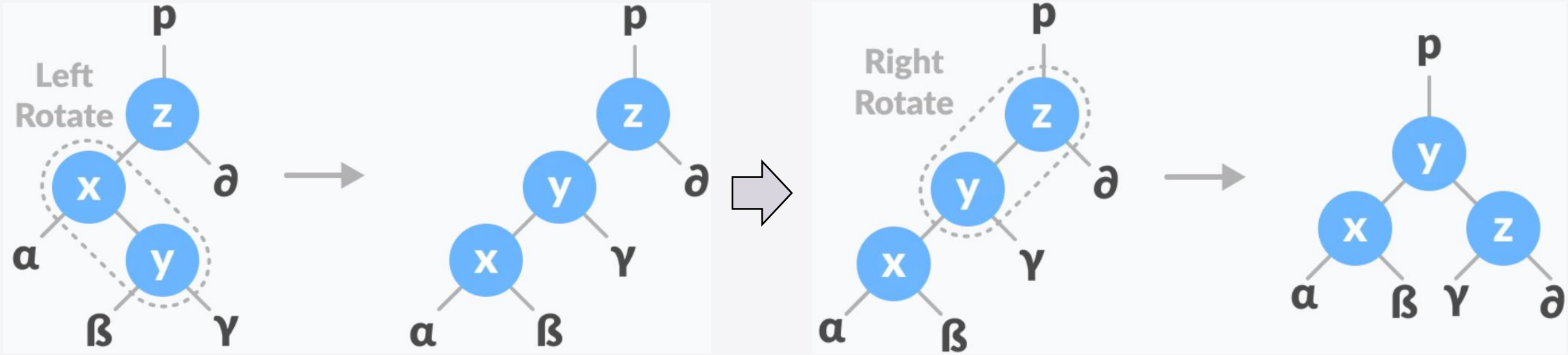
As depicted, the unbalanced node becomes the **right child** of its **left child** by performing a right rotation.

```
Node *AVL::RightRotate(Node *&node)
{
    //Save left child Node
    Node *apAux = node->left;
    //swap left child for child's right subtree
    node->left = apAux->right;
    //swap child's right subtree for node
    apAux->right = node;

    //Update node and left child's height (Only those change their height)
    node->height = max(Height(node->left), Height(node->right)) + 1;
    apAux->height = max(Height(apAux->left), Height(apAux->right)) + 1;

    return apAux;
}
```

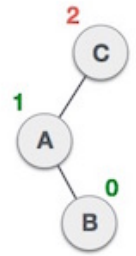
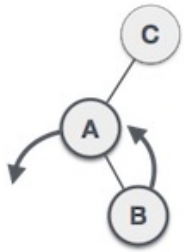
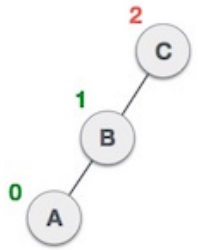
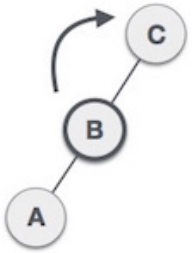
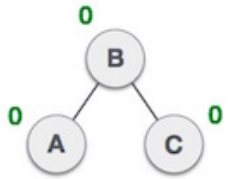


- Left-Right Rotate

In left-right rotation, the arrangements are first shifted to the **left** and then to the **right**.

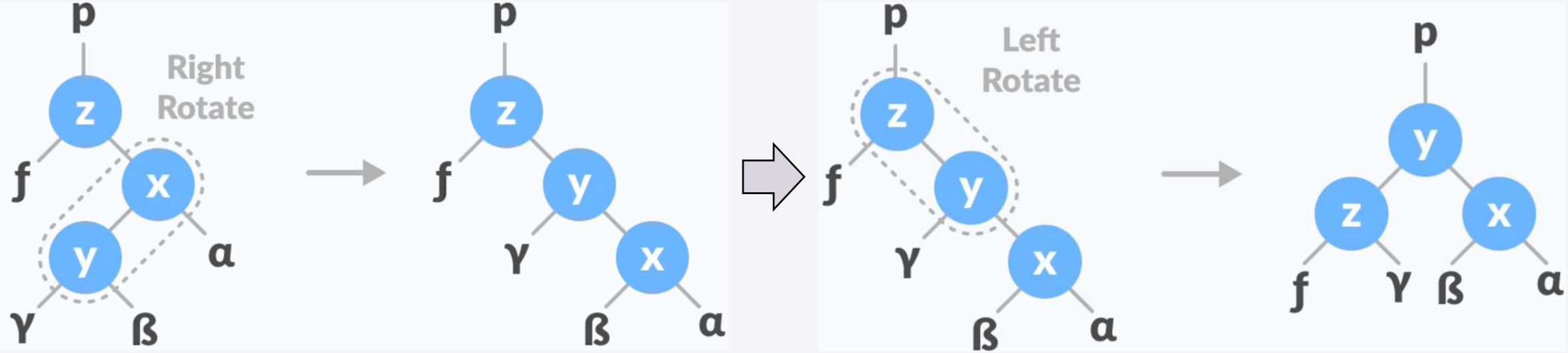


Implementation of AVL – Left-Right Rotate

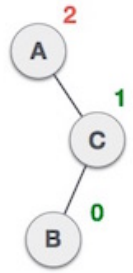
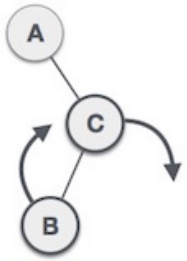
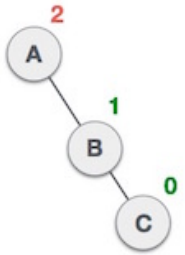
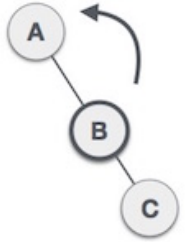
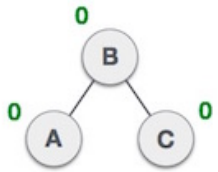
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

- Right-Left Rotate

In right-left rotation, the arrangements are first shifted to the **right** and then to the **left**.

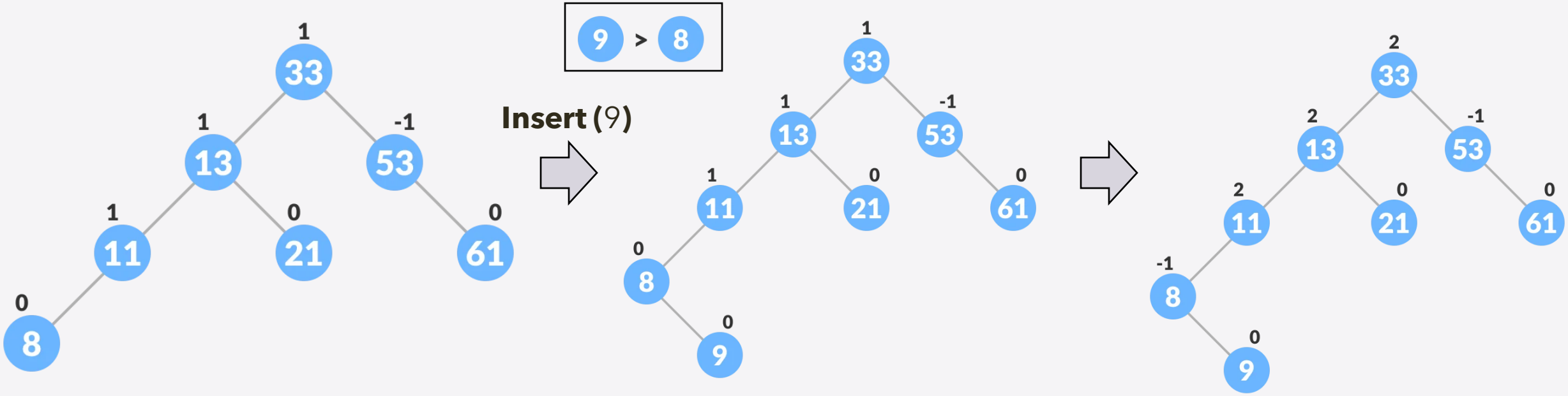


Implementation of AVL – Right-Left Rotate

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Insert a Node (1/3)

A new Node is always inserted as a **leaf node** with balance factor equal to **0**. Remember to Update **balanceFactor** of the nodes in the path you followed to insert the new node.

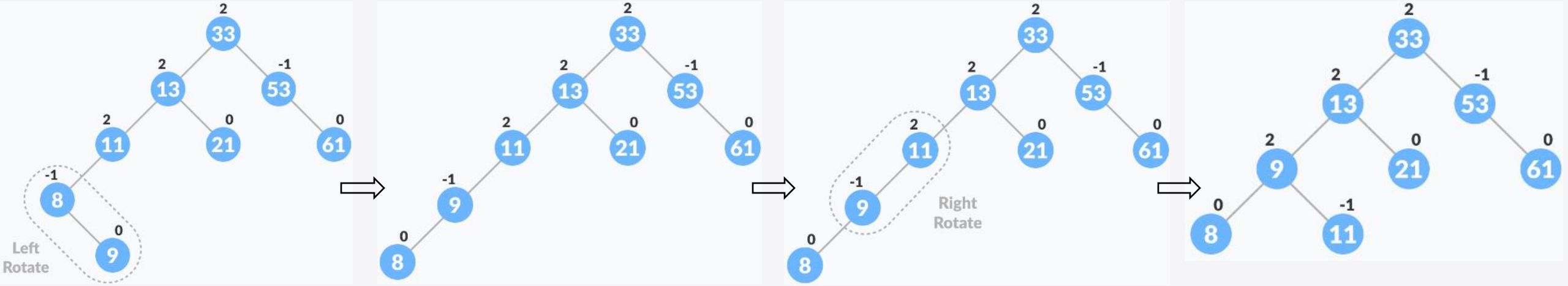


Insert a Node (2/3)

If the nodes are unbalanced, then rebalance the node.

If **balanceFactor > 1**, it means the height of the **left subtree is greater than that of the right subtree**.

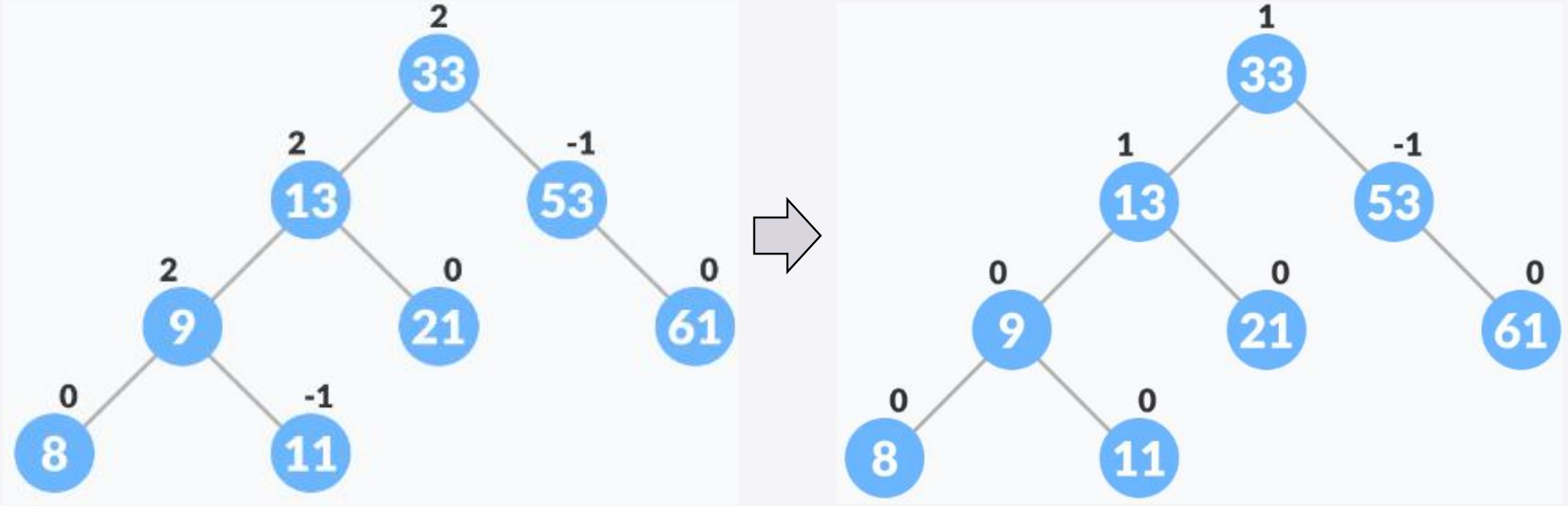
- If **newNodeKey** < **leftChildKey** do **right rotation**.
- Else, do **left-right rotation***.



Insert a Node (3/3)

If **balanceFactor < -1**, it means the height of **the right subtree is greater than that of the left subtree**.

- If **newNodeKey > rightChildKey** do **left rotation**.
- Else, do **right-left rotation**.



Delete a Node (1/4)

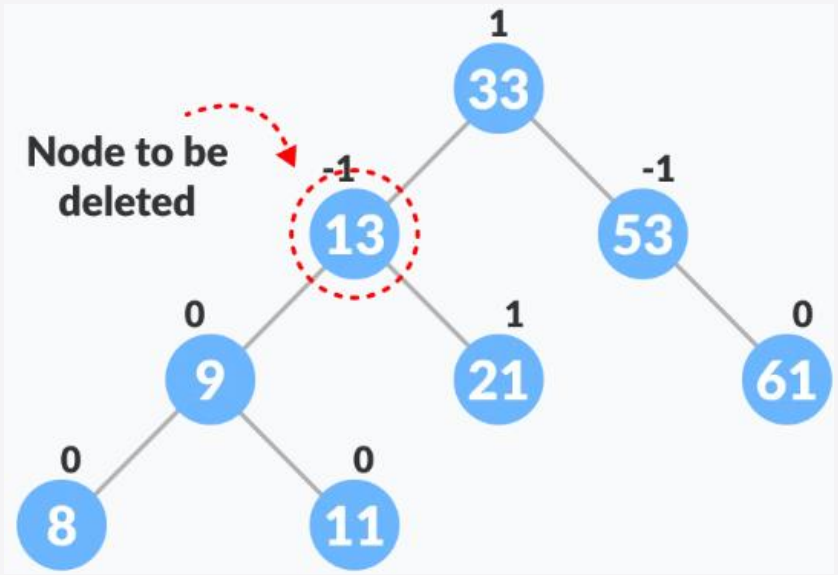
A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

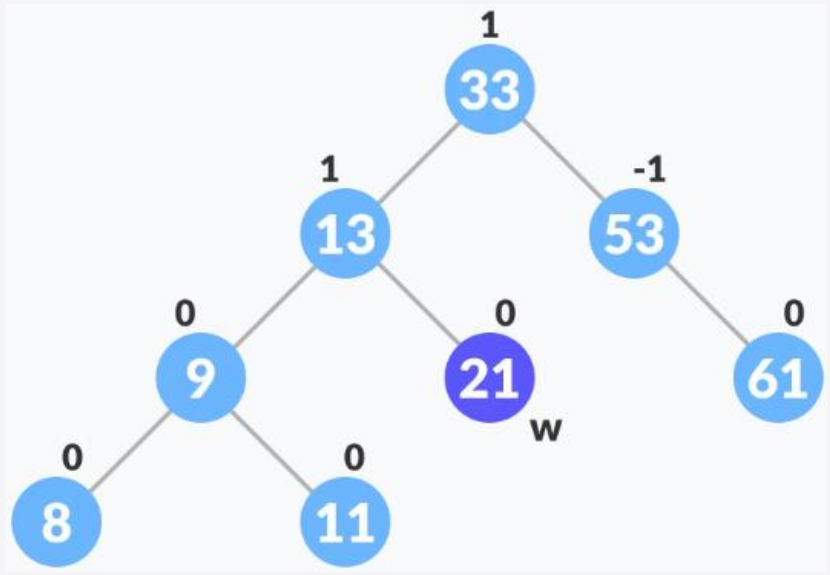
There are three cases for deleting a node:

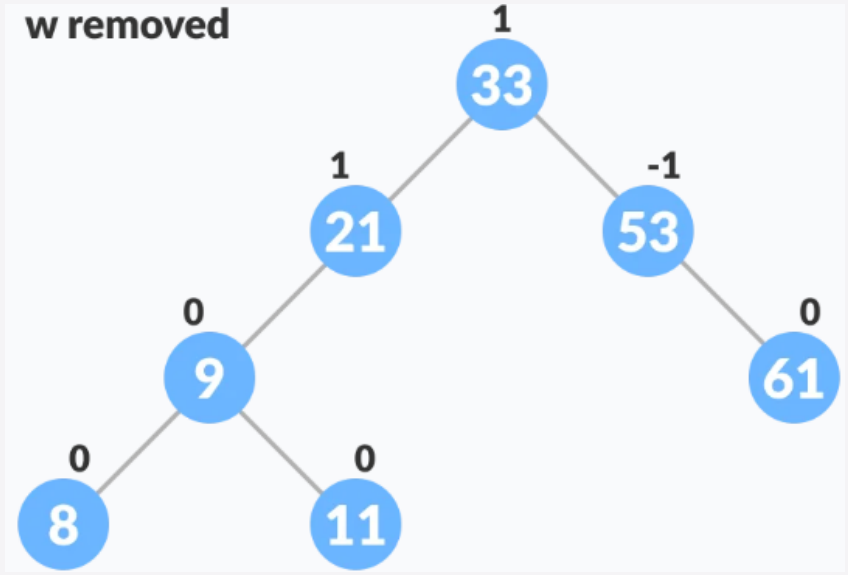
- If **nodeToBeDeleted** is the **leaf node** (ie. does not have any child), then remove nodeToBeDeleted.
- If **nodeToBeDeleted** has **one child**, then **substitute the contents of nodeToBeDeleted with that of the child**. Remove the child.
- If **nodeToBeDeleted** has **two children**, find the **inorder successor w** of **nodeToBeDeleted** (ie. node with a minimum value of key in the right subtree).

Delete a Node (2/4)

- If **nodeToBeDeleted** has **two children**, find the **inorder successor w** of **nodeToBeDeleted** (ie. node with a **minimum value of key in the right subtree**).







Delete a Node (3/4)

Update **balanceFactor** of the nodes. Rebalance the tree if the **balance factor** of any of the nodes is not equal to **-1**, **0** or **1**.

If **balanceFactor of currentNode > 1**,

- If **balanceFactor of leftChild >= 0**, do **right** rotation.
- Else do **left-right** rotation.

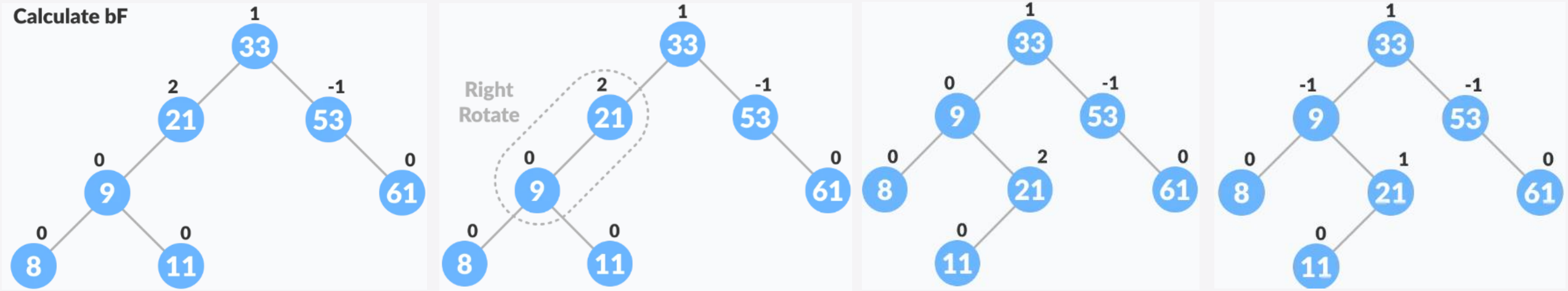
If **balanceFactor of currentNode < -1**,

- If **balanceFactor of rightChild <= 0**, do **left** rotation.
- Else do **right-left** rotation.

Delete a Node (4/4)

If **balanceFactor of currentNode > 1**,

- If **balanceFactor of leftChild >= 0**, do **right** rotation.
- Else do **left-right** rotation.



Binary Search Tree Complexities

Time Complexity

Here, **n** is the number of nodes in the tree.

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

Space Complexity

The space complexity for all the operations is **O(n)**.

AVL Tree Applications

Some of the applications of a AVL Tree are:

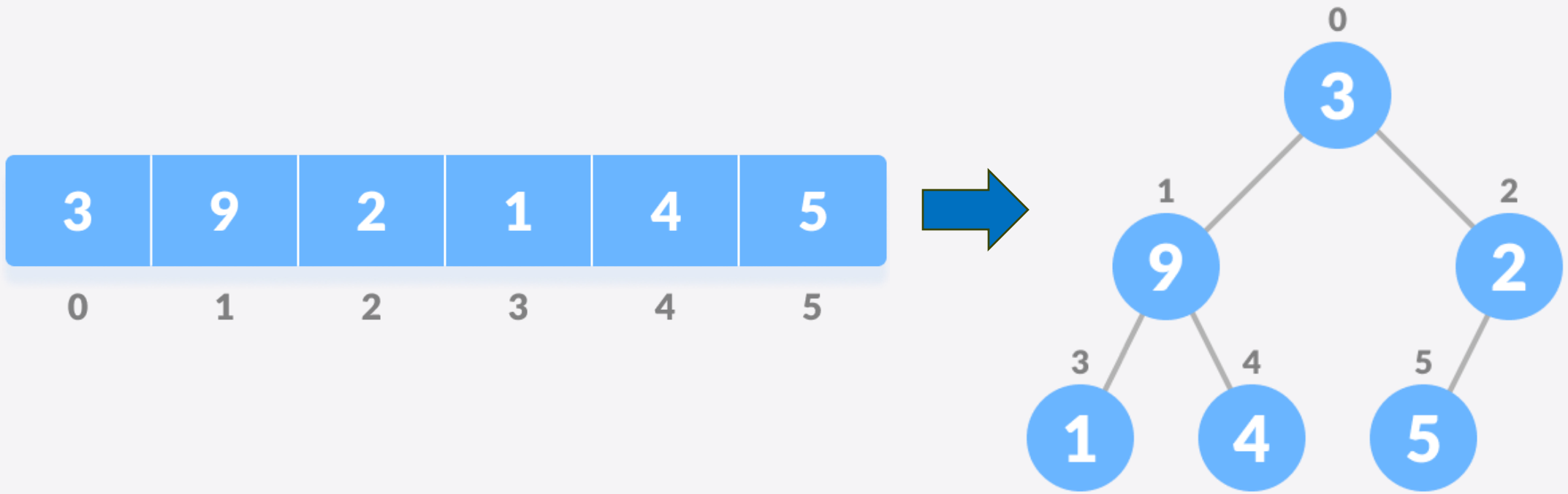
- For indexing large records in databases
- For searching in large databases

Heap Data Structure

Relationship between Array Indexes and Tree Elements

A **complete** **binary tree** has an interesting property that we can use to find the **children** and **parents** of any node.

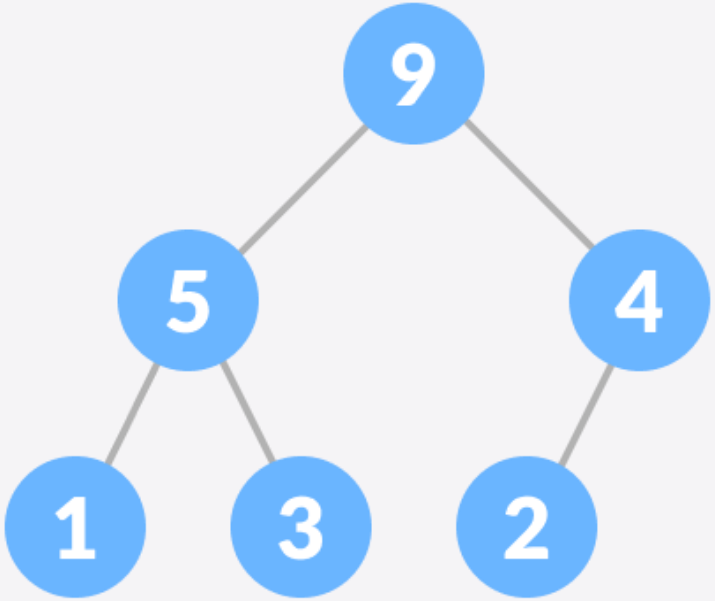
If the index of **any element in the array** is **i**, the element in the index **2*i+1** will become the **left child**, and element in **2*i+2** index will become the **right child**. Also, the **parent** of any element at index **i** is given by **floor[(i-1)/2]**.



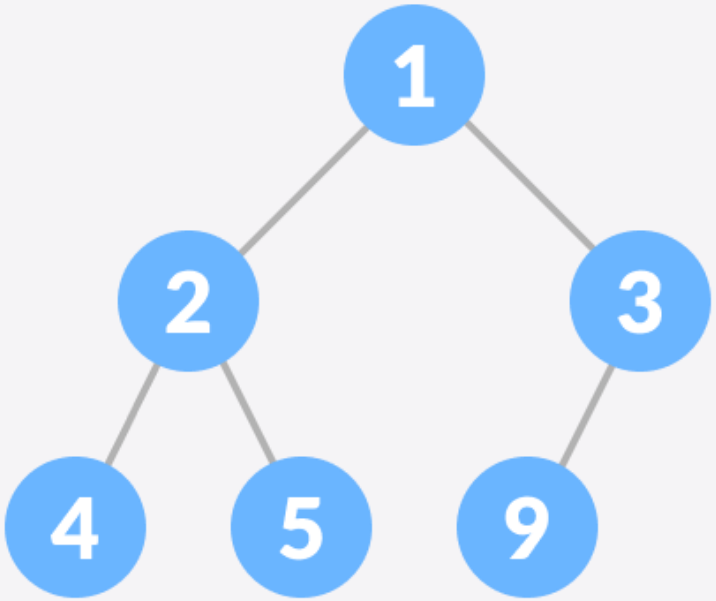
What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a **heap data structure** if it is a **complete binary tree** and follow one of this properties:

- **Max heap property:** All nodes are **greater** than its child node/s, i.e., **the largest element is at the root, both its children are smaller, and so on.**
- **Min heap property:** All nodes are **smaller** than the child node/s, i.e., **the smallest element is at the root, both its children are greater, etc.**



Max-Heap



Min-Heap

Heap Operations

Some of the important operations performed on a heap are:

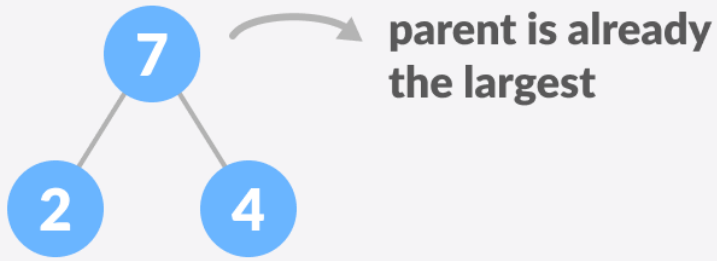
- **Heapify**
- **Insert element into Heap**
- **Delete element from Heap**
- **Peek (Find max/min)**
- **Extract-Max/Min**

How to "heapify" a tree?

Starting from a **complete binary tree**, we can modify it to become a **Max/Min-Heap** by running the function **heapify** on all the **non-leaf elements**.

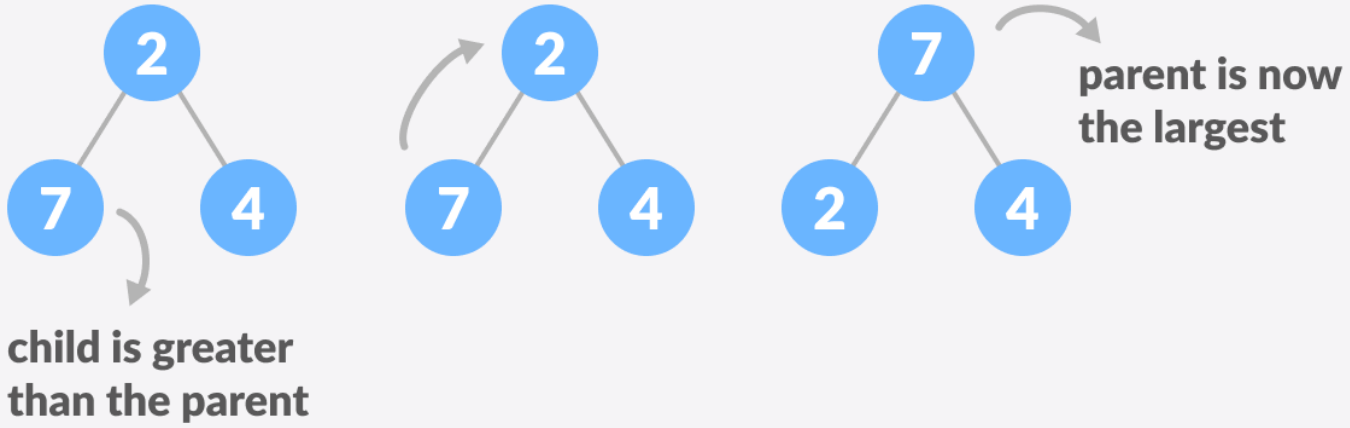
Heapify uses **recursion**, let's first think in a scenario of just three elements.

Scenario-1



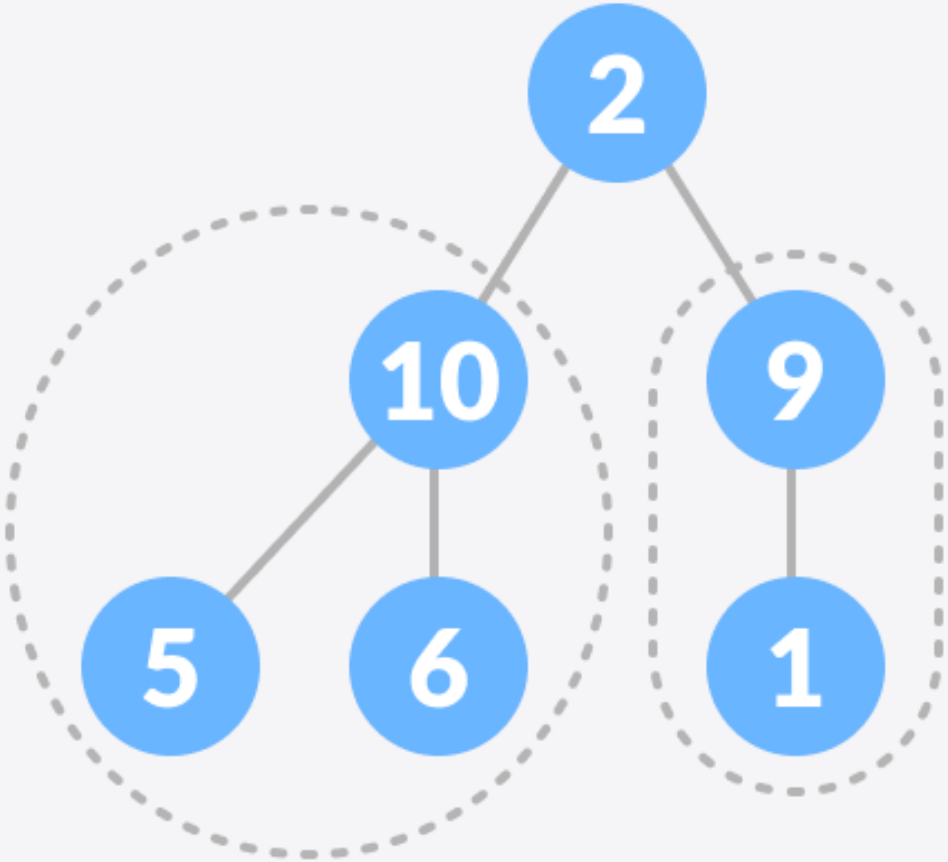

```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1], array[2*0+2])
    if(Root != Largest)
          Swap(Root, Largest)
```

Scenario-2



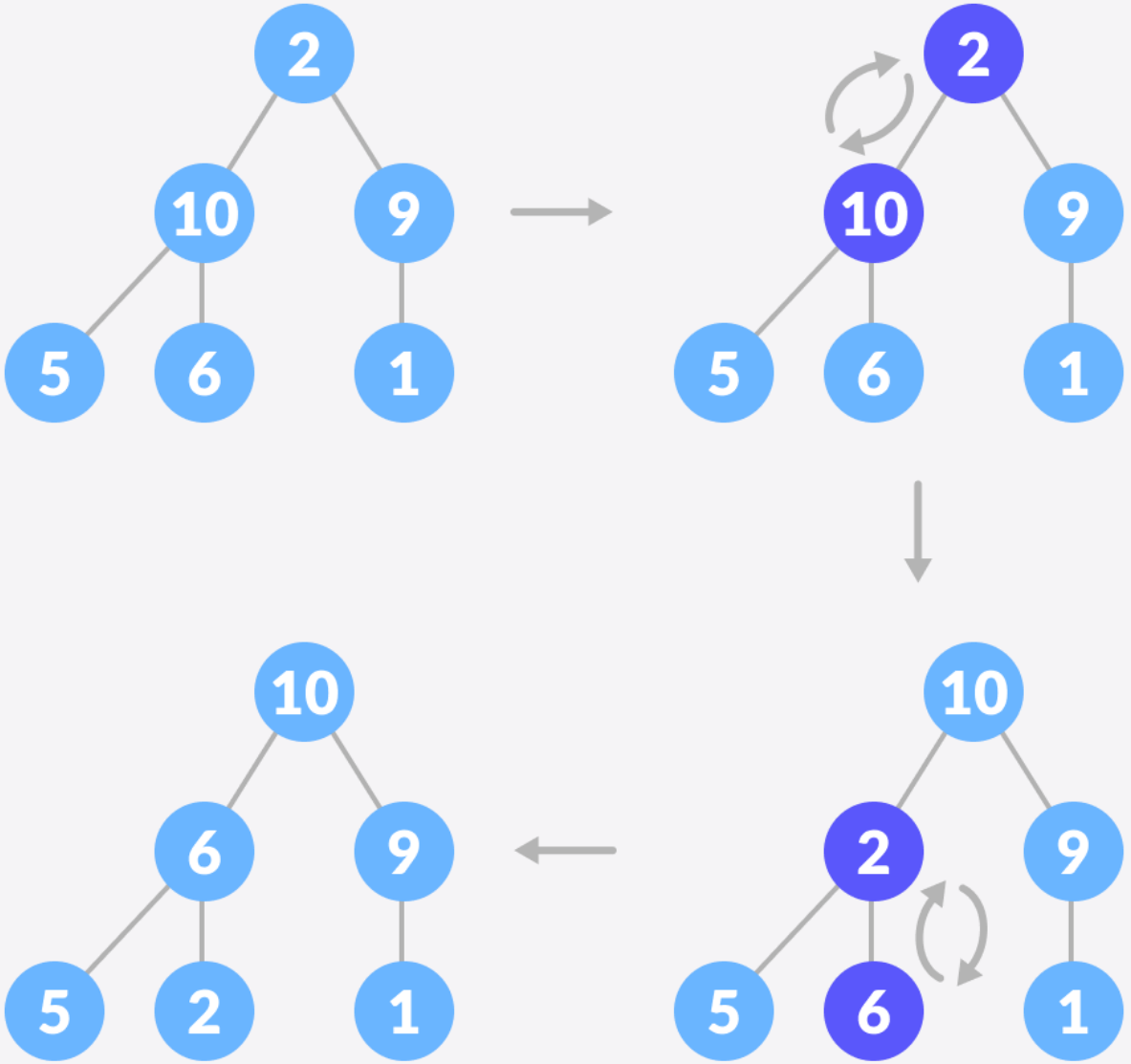
You've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.



The **top element** isn't a **max-heap** but all the **sub-trees** are **max-heaps**.

Normally, we need to run **heapify** on the **root** element **repeatedly** until **its on the right position**. In this case, **we will have to keep pushing 2 downwards until it reaches its correct position**.



Note we assumed that the subtrees were already Max-Heap.

What happen if not?

We need a strategy.

Build Max/Min-Heap

To build a **max-heap** from any tree, we can thus start **heapifying** **each sub-tree from the bottom up to the root element**.

The **first index of a non-leaf node** is given by **n/2 - 1**. All other nodes after that are **leaf-nodes** and thus don't need to be **heapified**.

```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
```

If we have

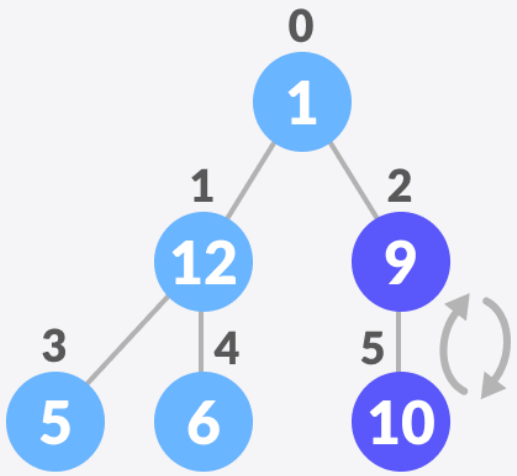
	0	1	2	3	4	5
arr	1	12	9	5	6	10

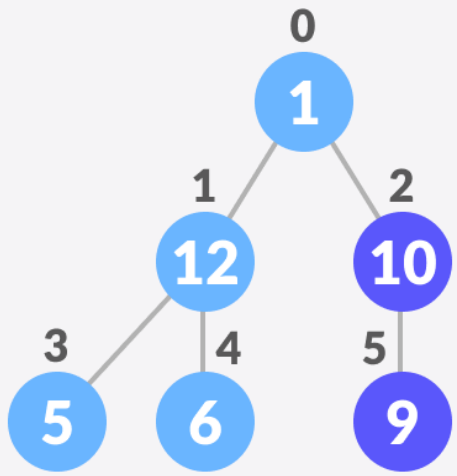
n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0

we can build a maximum heap as

i = 2 → heapify(arr, 6, 2)



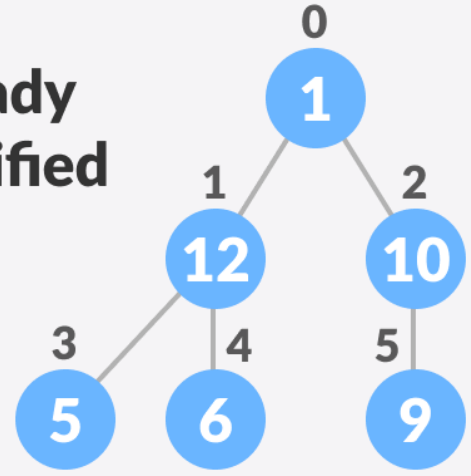


	0	1	2	3	4	5
	1	12	9	5	6	10

	0	1	2	3	4	5
	1	12	10	5	6	9

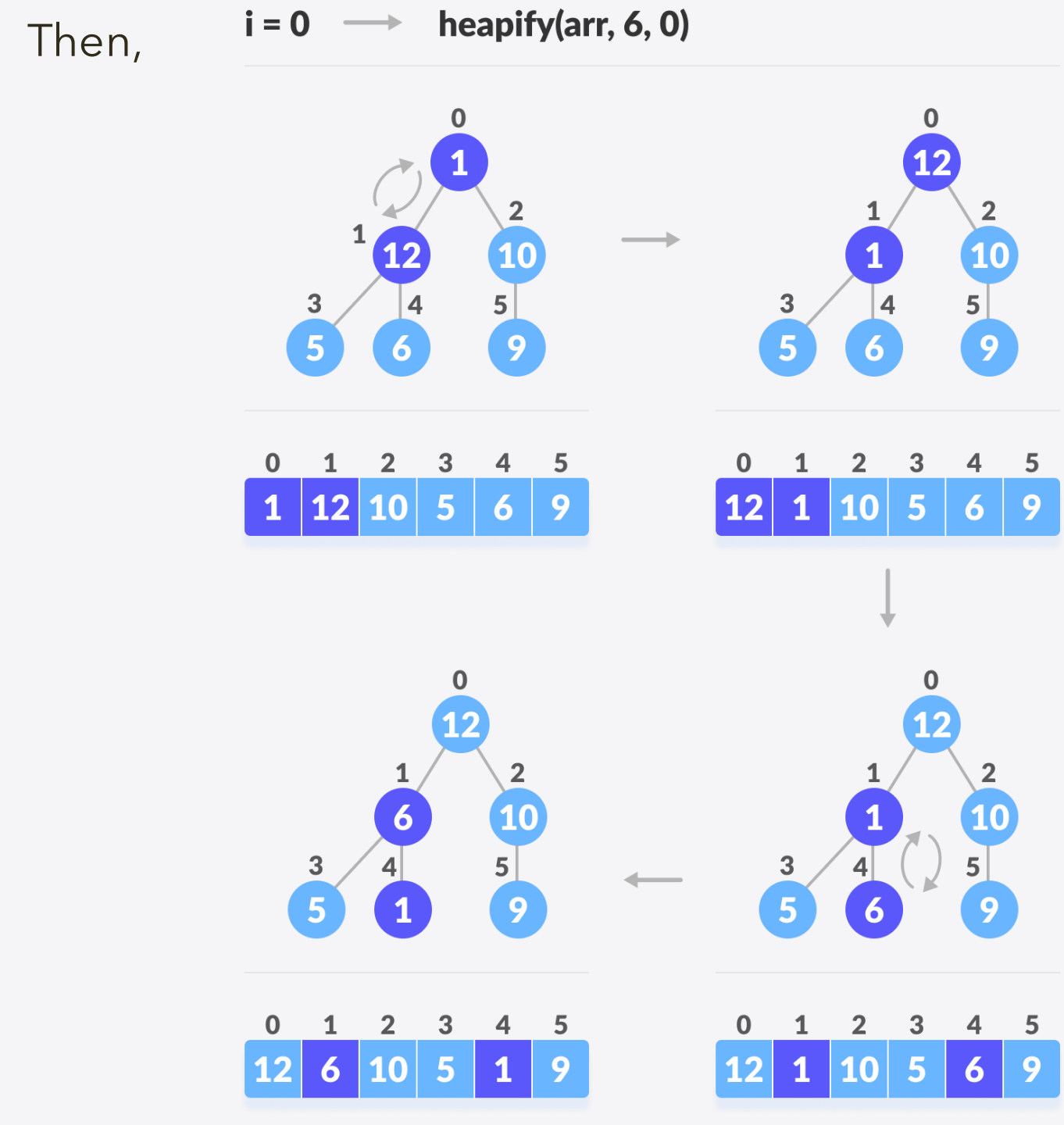
i = 1 → heapify(arr, 6, 1)

already heapified



	0	1	2	3	4	5
	1	12	10	5	6	9

Then,



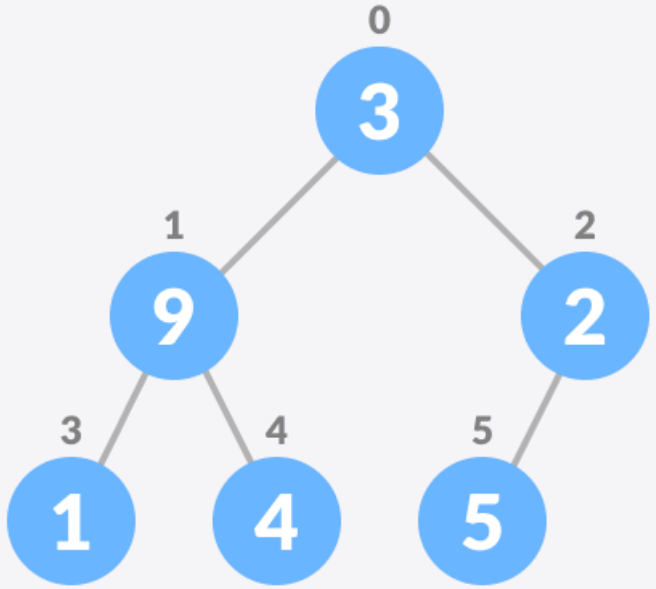
As shown, we start by **heapifying** the **lowest smallest trees** and gradually move up until we reach the **root element**.

Heapify

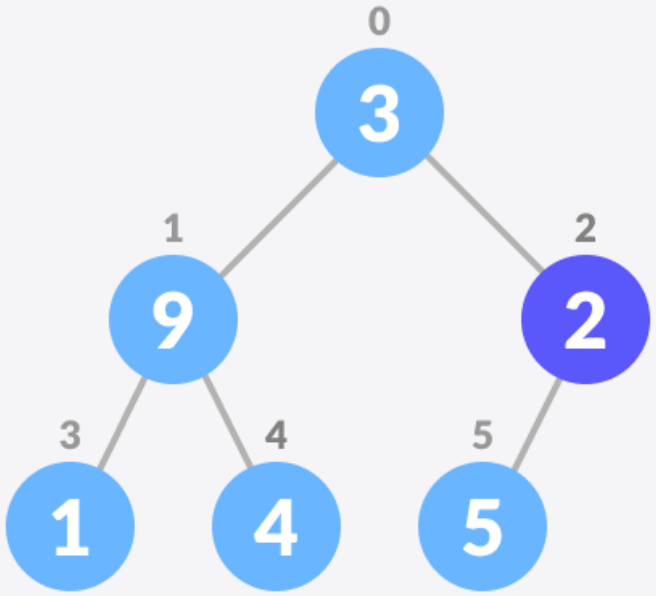
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a **Min-Heap** or a **Max-Heap**.

1. Let the input array be

2. Use an **array** as a **complete binary tree**.



3. Start from the first index of non-leaf node whose index is given by **n/2 - 1**.



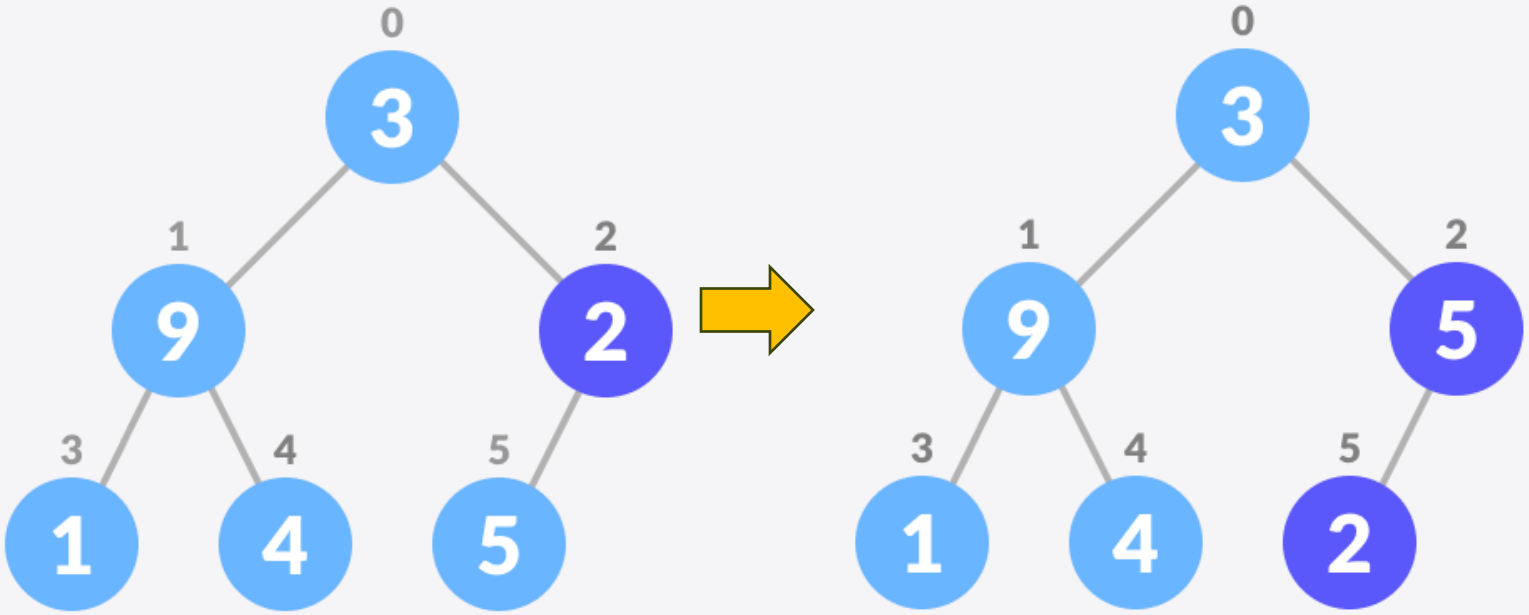
n/2 - 1 = 6/2 - 1 = 2

4. Set element **i** as **current largest element**.

5. The index of **left child** is given by **2i + 1** and the **right child** is given by **2i + 2**.
If **leftChild** is **greater** than largest element **i**, set **leftChildIndex** as largest.

If **rightChild** is **greater** than largest element **i**, set **rightChildIndex** as largest.

6. Swap **largest** with largest element **i**.



7. Repeat steps **3**-**6** until the subtrees are also heapified.

Implementation – Heapify

```
void heapify(vector<int> &hT, int i)
{
  int size = hT.size();

  // Find the largest among root, left child and right child
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;

  // Swap and continue heapifying if root is not largest
  if (largest != i) {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}
```

To create a **Max-Heap**:

```
for (int i = size / 2 - 1; i >= 0; i--)
{
  heapify(hT, i);
}
```

For **Min-Heap**, both left child **L** and right child **R** must be **larger** than the parent for all nodes.

What is the time complexity?

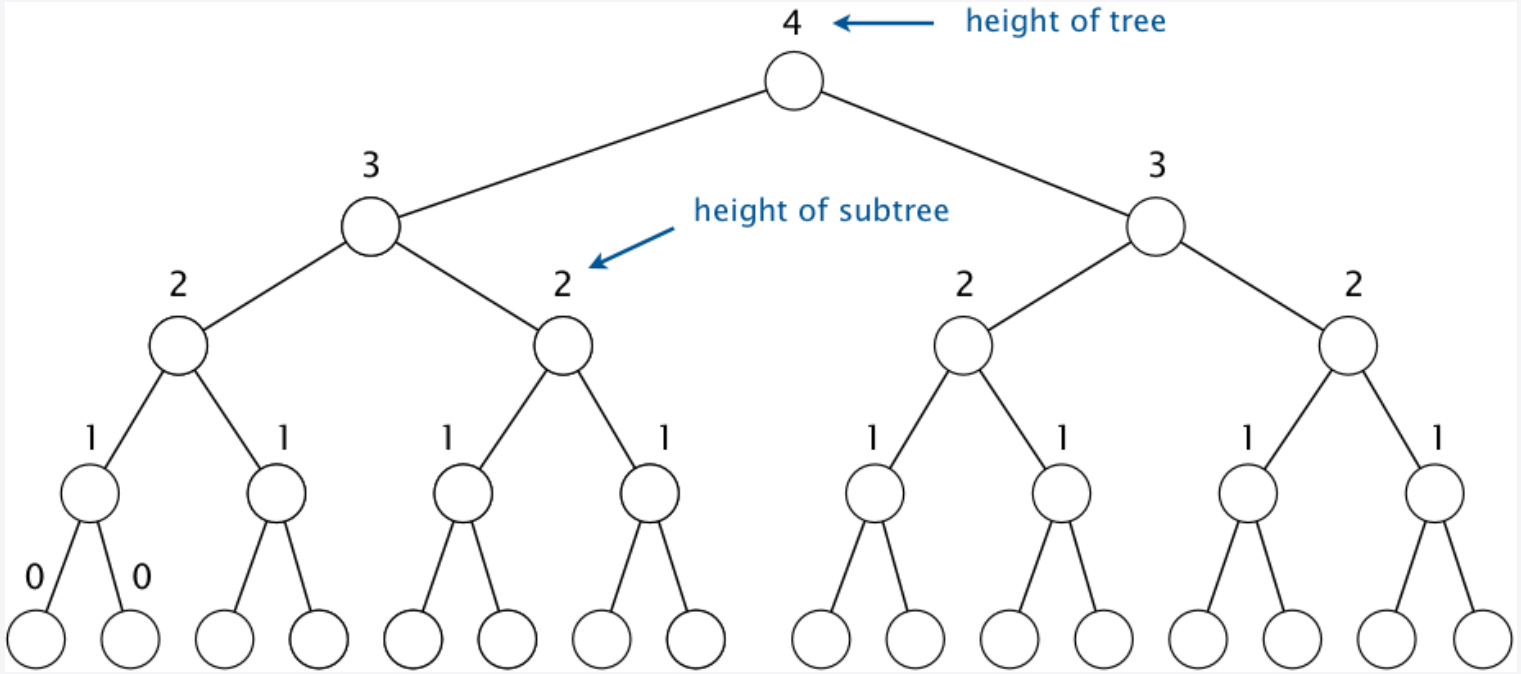
A quick look over the above algorithm suggests that the running time is **O(**n*log(n)**)** since each call to **Heapify** costs **O(**log(n)**)** and **Create Max -Heap** makes **O(**n**)** such calls.

Time Complexity of building a heap

This upper bound **O(n*log(n))**, though correct, is **not asymptotically tight**.

We can derive a **tighter bound** by observing that the running time of **Heapify** depends on the height of the tree **'h'** (equal to **log(n)** max) and it is measure from the **leaf Nodes**.

In this case, a binary heap has at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height **h**.



n = 32

Time Complexity of building a heap

This upper bound **O(n*log(n))**, though correct, is **not asymptotically tight**.

We can derive a **tighter bound** by observing that the running time of **Heapify** depends on the height of the tree **'h'** (equal to **log(n)** max) and it is measure from the **leaf Nodes**.

In this case, a binary heap has at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height **h**.

To obtain a **Max-Heap**, we need to run a loop from the index of the **last internal node** (**n/2**) with **h=1**, to the index of **root** (**1**) with **h = log(n)**.

```
for (int i = size / 2 - 1; i >= 0; i--)
{
  heapify(hT, i);
}
```

Time Complexity of building a heap

This upper bound **O(n*log(n))**, though correct, is **not asymptotically tight**.

We can derive a **tighter bound** by observing that the running time of **Heapify** depends on the height of the tree **'h'** (equal to **log(n)** max) and it is measure from the **leaf Nodes**.

In this case, a binary heap has at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes at height **h**.

To obtain a **Max-Heap**, we need to run a loop from the index of the **last internal node** (**n/2**) with **h=1**, to the index of **root** (**1**) with **h = log(n)**.

To find the Time Complexity, we must know the **number of nodes having height h** and the **complexity for a node at that height**.

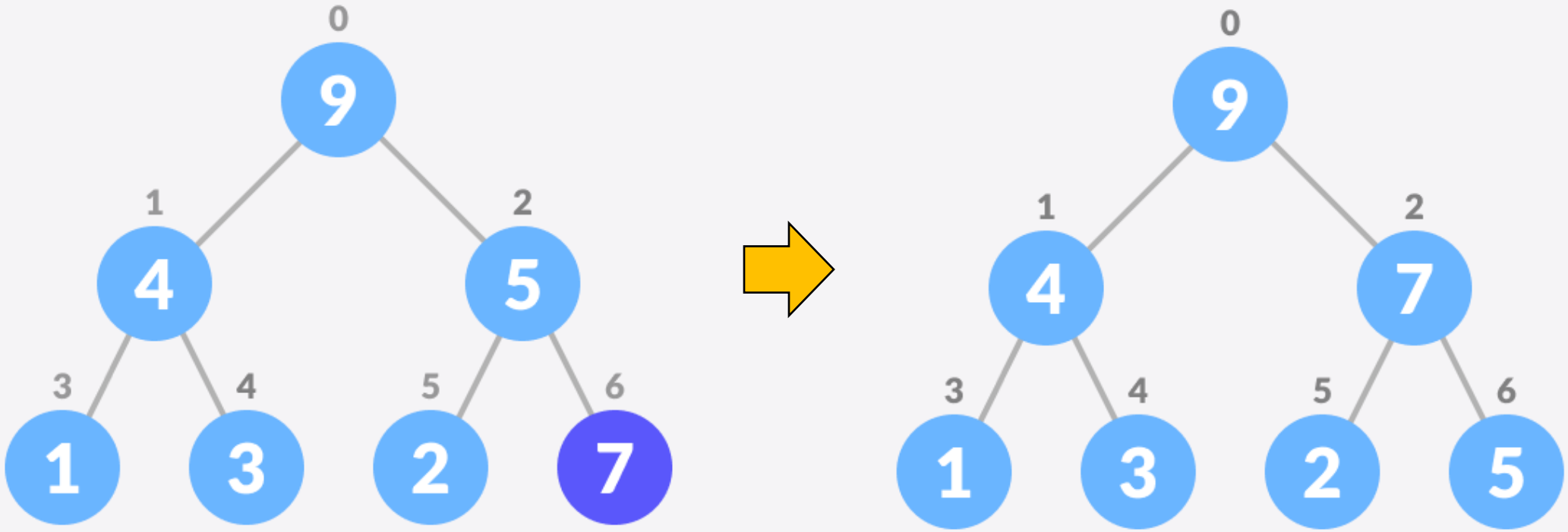
- Giving the properties of the Big-Oh notation, we ignore the **ceiling function** and the constant **2**.
- The **upper limit of the summation** can be increased to **infinity** since we are using Big-Oh notation to find a limit.

$$\begin{aligned}
\sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) &= O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^{h+1}} \right) \\
&= O\left(\frac{n}{2} \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^{h}} \right) \\
&= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right) \\
&= O(2n) \\
&= O(n)
\end{aligned}$$

Insert Element into Heap

Algorithm for insertion in Max Heap.

1. Insert the new element at the end of the tree.
2. **Heapify** the tree.

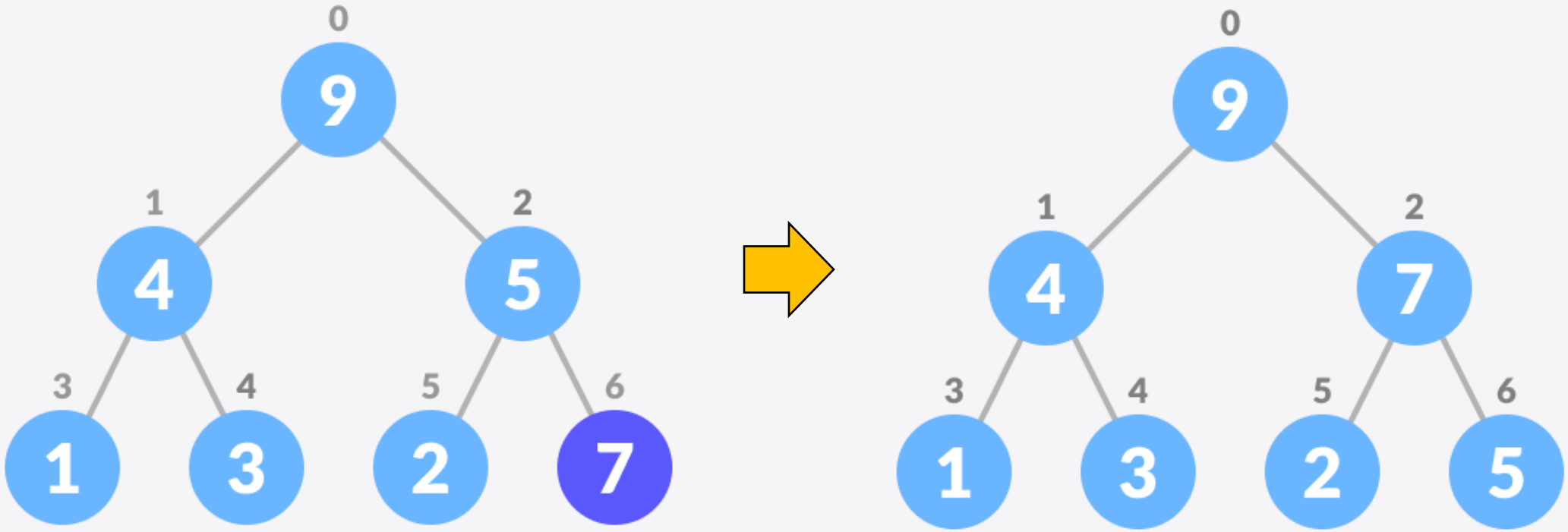


For **Min Heap**, the above algorithm is modified so that **parentNode** is always smaller than **newNode**.

Insert Element into Heap

Algorithm for insertion in **Max Heap**.

1. Insert the new element at the end of the tree.
2. **Heapify** the tree.



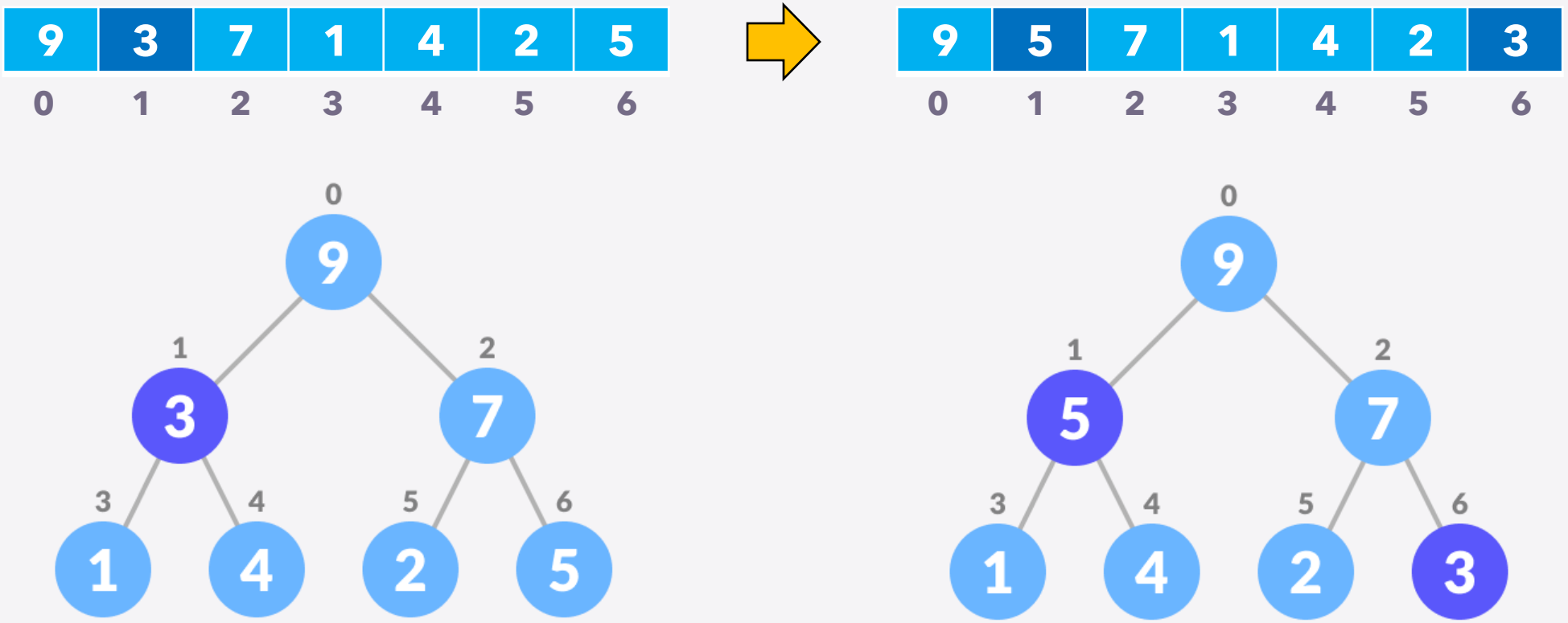
For **Min Heap**, the above algorithm is modified so that **parentNode** is always smaller than **newNode**.

Delete Element from Heap

Algorithm for deletion in **Max Heap**.

1. Select the element to be deleted.

2. Swap it with the last element.

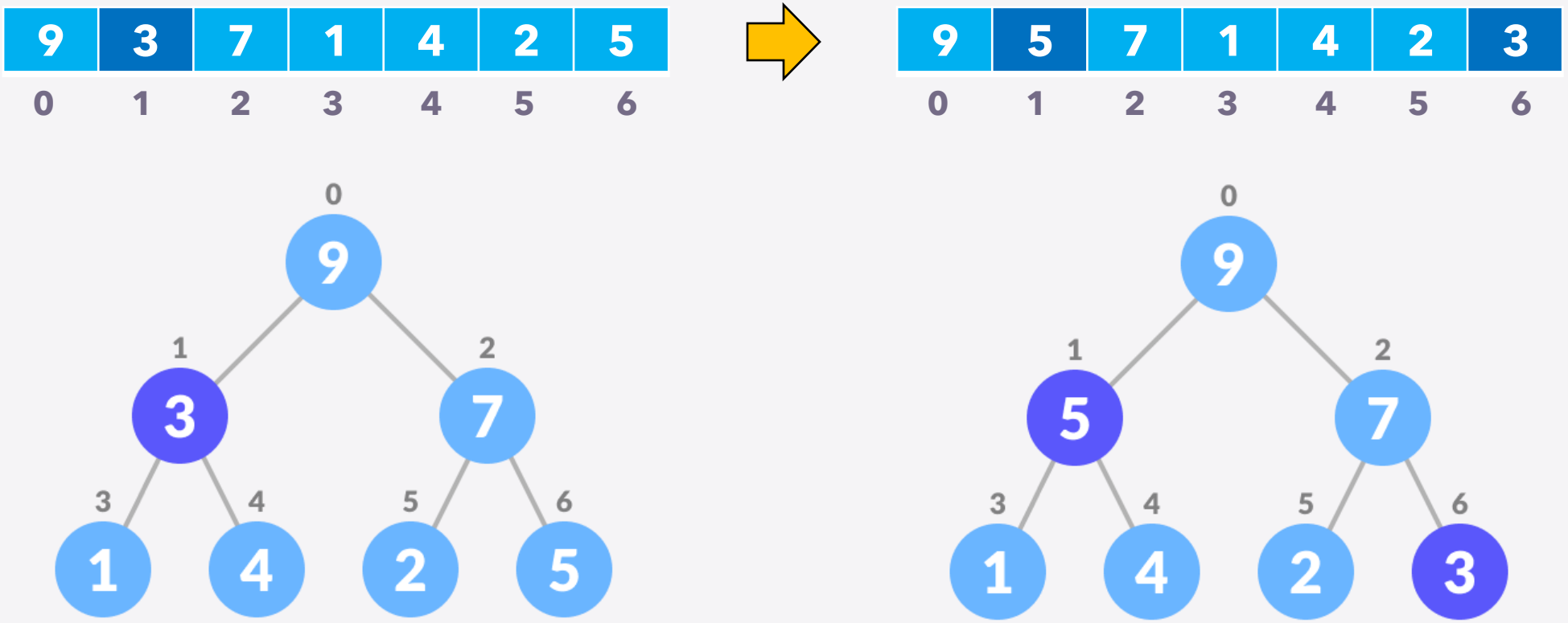


Delete Element from Heap

Algorithm for deletion in **Max Heap**.

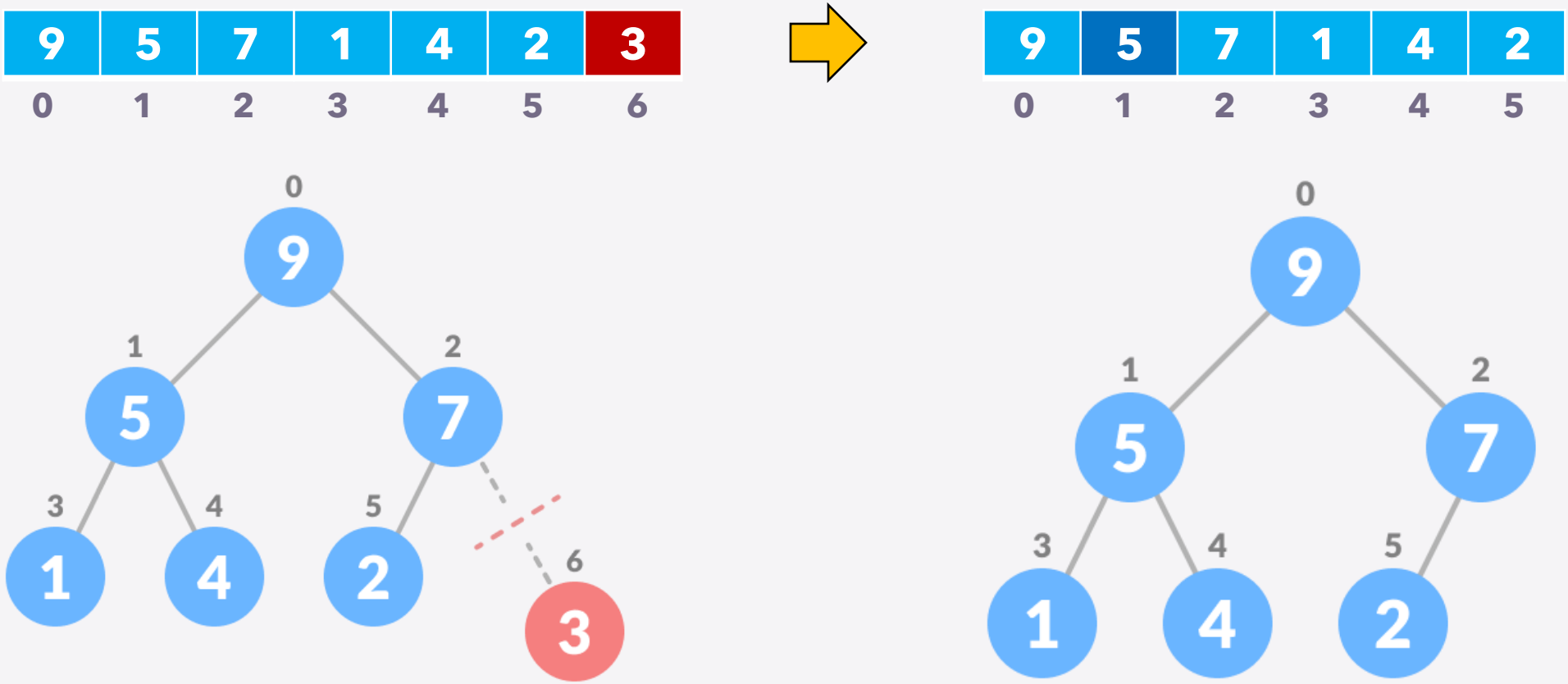
1. Select the element to be deleted.

2. Swap it with the last element.



3. Remove the last element.

4. **Heapify** the tree.



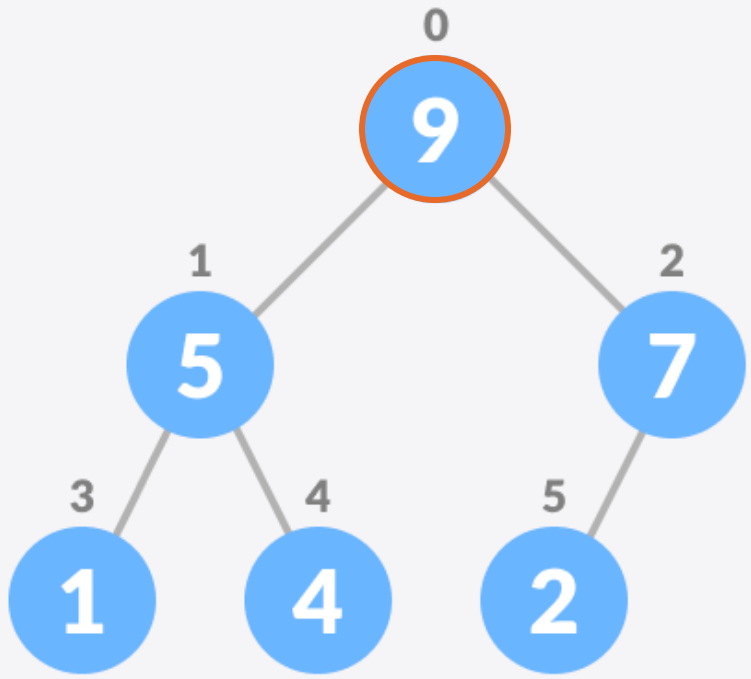
For **Min Heap**, the above algorithm is modified so that both **childNodes** are greater smaller than **currentNode**.

Peek (Find Max/Min Element)

Peek operation **returns** the **maximum** element from **Max Heap** or **minimum** element from **Min Heap** **without deleting the node**.

For both Max heap and Min Heap

```
return rootNode
```



Heap Applications

Some of the applications of a Heap data structure are:

- Heap is used while implementing a **priority queue**.
- Dijkstra's Algorithm
- Heap Sort

Act 3.2
- Heap Tree. Implementing a priority Queue

What do I have to do?

Individually, implement an **Priority Queue (PQ)** using **Heap** data structured, where:

- You can use **vector<**node**>** data type.
- Every node have to have **(data**, **priority)** values.

Implementing the fundamental functionalities:

- **Push/Insert** (Add data)
- **Pop/Delete** (remove data with highest priority)
- **Top/Peek**

(return the data with highest priority without removing it)

- **Empty** (return True/False if the PQ is empty)
- **Size** (return the current amount of data in the PQ)
- **Delete All** (Free up all the dynamic memory space used)