

CE205 Data Structures

Week-4

Tree Data Structure Types and Applications (Binary Tree, Tree Traversals, Heaps)

Download [PDF](#), [DOCX](#), [SLIDE](#), [PPTX](#)



Outline

- Graph Representation Tools
- Tree Structures and Binary Tree and Traversals (In-Order, Pre-Order, Post-Order)
- Heaps (Max, Min, Binary , Binomial, Fibonacci, Leftist, K-ary) and Priority Queue
- Heap Sort
- Huffman Coding

Graph Representation Tools


- Microsoft Automatic Graph Layout
 - <https://www.microsoft.com/en-us/download/details.aspx?id=52034>
 - <https://github.com/microsoft/automatic-graph-layout>
- Graphviz
 - <https://graphviz.org/resources/>
- Plantuml
 - https://ucoruh.github.io/ce204-object-oriented-programming/week-5/ce204-week-5/#calling-plantuml-from-java_1

Microsoft Automatic Graph Layout

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
class ViewerSample {
    public static void Main() {
        //create a form
        System.Windows.Forms.Form form = new System.Windows.Forms.Form();
        //create a viewer object
        Microsoft.Msagl.GraphViewerGdi.GViewer viewer = new Microsoft.Msagl.GraphViewerGdi.GViewer();
        //create a graph object
        Microsoft.Msagl.Drawing.Graph graph = new Microsoft.Msagl.Drawing.Graph("graph");
        //create the graph content
        graph.AddEdge("A", "B");
        graph.AddEdge("B", "C");
        graph.AddEdge("A", "C").Attr.Color = Microsoft.Msagl.Drawing.Color.Green;
        graph.FindNode("A").Attr.FillColor = Microsoft.Msagl.Drawing.Color.Magenta;
        graph.FindNode("B").Attr.FillColor = Microsoft.Msagl.Drawing.Color.MistyRose;
        Microsoft.Msagl.Drawing.Node c = graph.FindNode("C");
        c.Attr.FillColor = Microsoft.Msagl.Drawing.Color.PaleGreen;
        c.Attr.Shape = Microsoft.Msagl.Drawing.Shape.Diamond;
        //bind the graph to the viewer
        viewer.Graph = graph;
        //associate the viewer with the form
        form.SuspendLayout();
        viewer.Dock = System.Windows.Forms.DockStyle.Fill;
        form.Controls.Add(viewer);
        form.ResumeLayout();
        //show the form
        form.ShowDialog();
    }
}
```

Graph Representation Tools

Microsoft Automatic Graph Layout

 microsoft_graph.png

MSAGL Modules

The Core Layout engine (AutomaticGraphLayout.dll) - [NuGet package](#) This .NET assembly contains the core layout functionality. Use this library if you just want MSAGL to perform the layout only and afterwards you will use a separate tool to perform the rendering and visualization.

MSAGL Modules

The Drawing module (AutomaticGraphLayout.Drawing.dll) - [NuGet package](#) The Definitions of different drawing attributes like colors, line styles, etc. It also contains definitions of a node class, an edge class, and a graph class. By using these classes a user can create a graph object and use it later for layout, and rendering.

MSAGL Modules

A WPF control (`Microsoft.Msagl.WpfGraphControl.dll`) - [NuGet package](#) The viewer control lets you visualize graphs and has and some other rendering functionality. Key features: (1) Pan and Zoom (2) Navigate Forward and Backward (3) tooltips and highlighting on graph entities (4) Search for and focus on graph entities.

MSAGL Modules

A Windows Forms Viewer control (Microsoft.Msagl.GraphViewerGdi.dll) - [NuGet package](#) The viewer control lets you visualize graphs and has and some other rendering functionality. Key features: (1) Pan and Zoom (2) Navigate Forward and Backward (3) tooltips and highlighting on graph entities (4) Search for and focus on graph entities.

Custom MSAGL Demo Project

- Clone and test your self
 - [GitHub - ucoruh/microsoft-graph-layout-cs-demo](https://github.com/ucoruh/microsoft-graph-layout-cs-demo): Example Usage of <https://github.com/microsoft/automatic-graph-layout>
- Also you can find another example in this homework
 - [GitHub - ucoruh/ce205-hw3-template](https://github.com/ucoruh/ce205-hw3-template): CE205-HW3-Template

Custom MSAGL Demo Project

CE205 HW3 2021-2022

File

Algorithm

MST DFS BFS SCC Topological Order

Clear Edges

Random Graph Generation

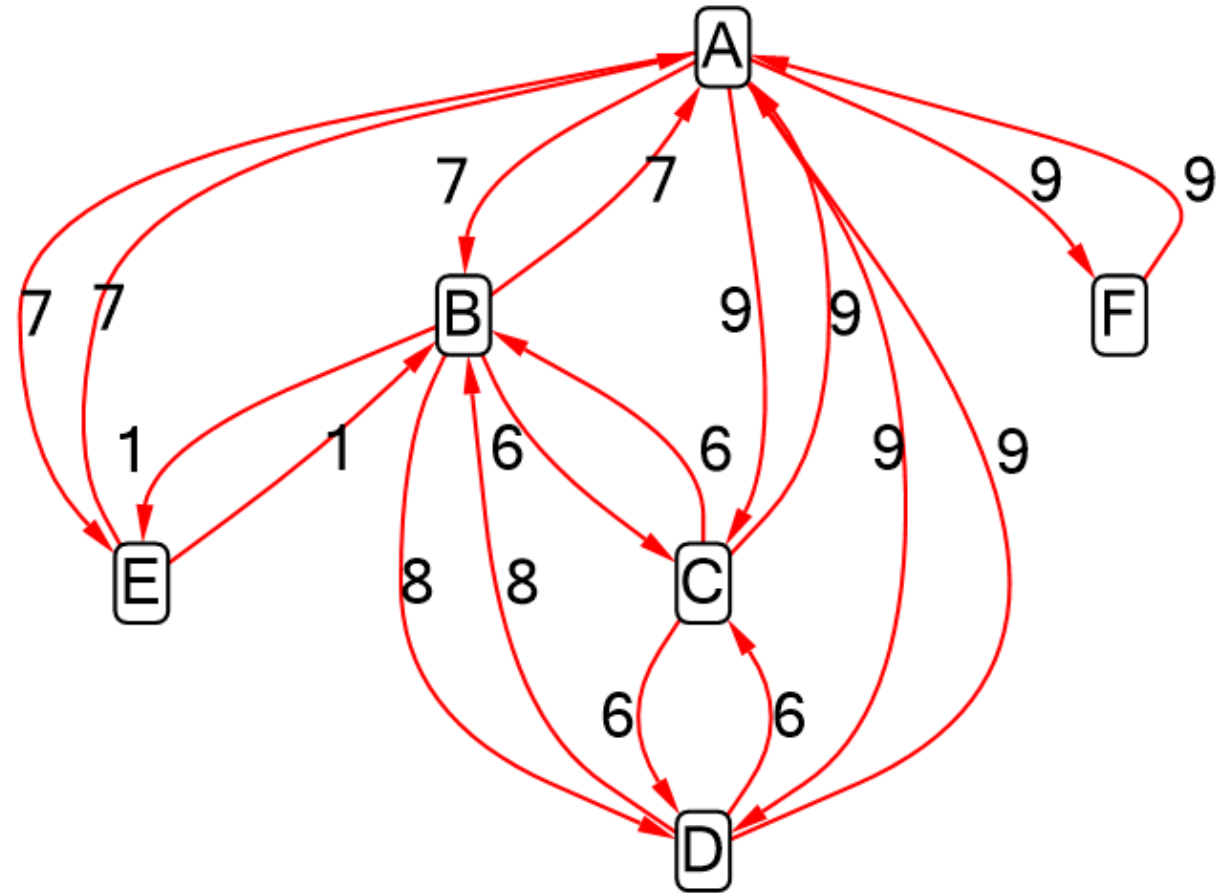
Reset Graph

Solve

<< Previous Step

Next Step >>

NOTE: Select two algorithm and implement in this homework Good Luck. (Step-by-Step Operation is Optional)



Custom MSAGL Demo Project

CE205 Data Structures Week-4

CE205 HW3 2021-2022

File

Algorithm

MST DFS BFS SCC Topological Order

Clear Edges

Random Graph Generation

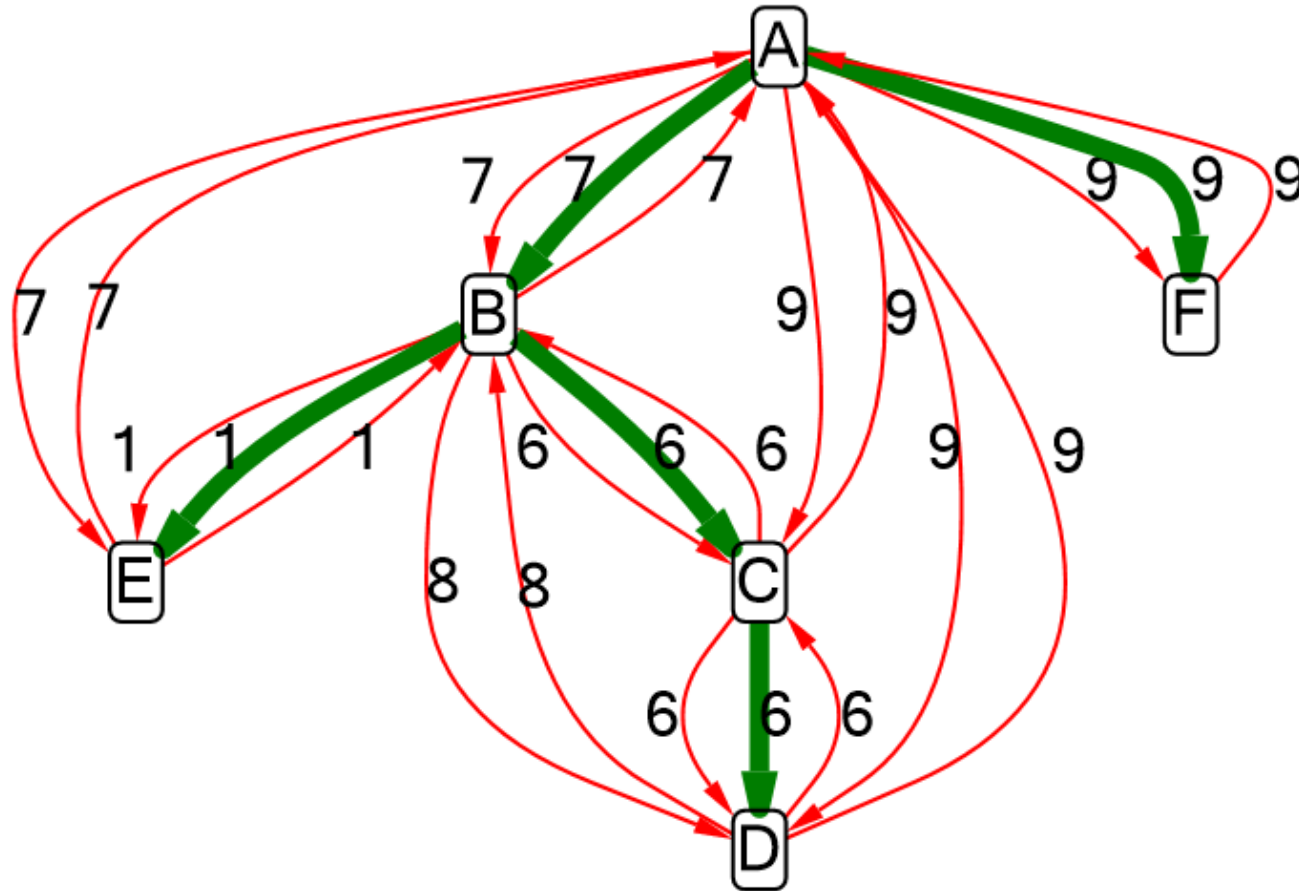
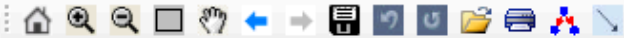
Reset Graph

Solve

<<Previous Step

Next Step>>

NOTE: Select two algorithm and implement in this homework Good Luck. (Step-by-Step Operation is Optional)



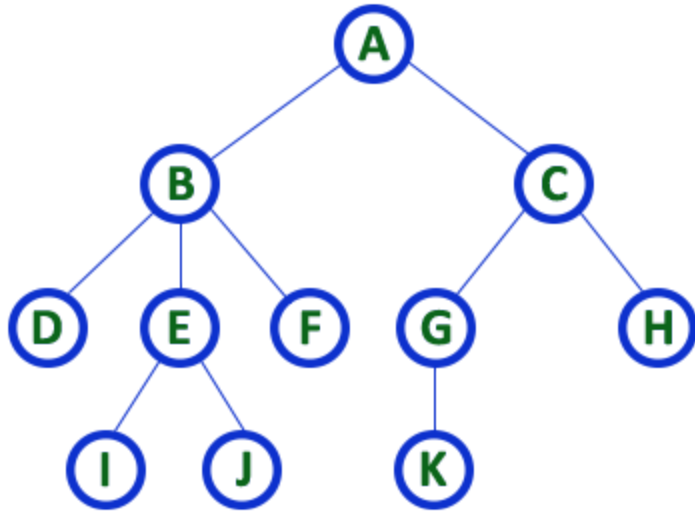
Tree Structures and Binary Tree and Traversals (In-Order, Pre-Order, Post-Order)

Tree - Terminology

- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/tree-terminology.html

- In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order.
- A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows.
- Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.
- A tree data structure can also be defined as follows
 - Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively
 - In tree data structure, every individual element is called as **Node**.
 - Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

Tree Example



TREE with 11 nodes and 10 edges

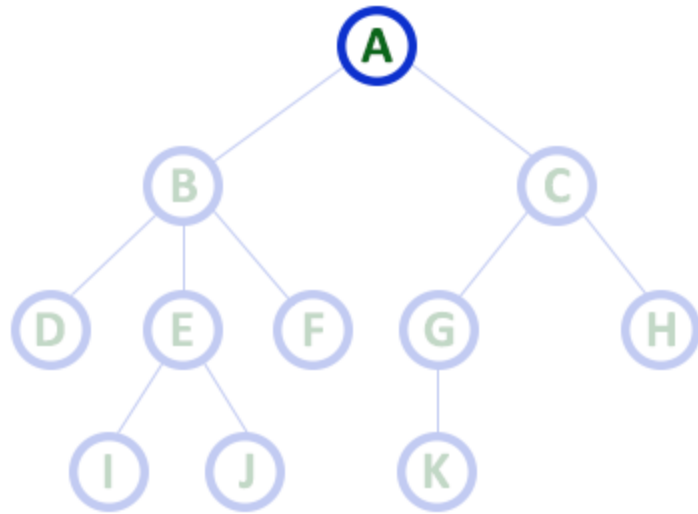
- In any tree with '**N**' nodes there will be maximum of '**N-1**' edges
- In a tree every individual element is called as '**NODE**'

Tree Terminology

- In a tree data structure, we use the following terminology

1. Root

- In a tree data structure, the first node is called as **Root Node**.
- Every tree must have a root node.
- We can say that the root node is the origin of the tree data structure.
- In any tree, there must be only one root node.
- We never have multiple root nodes in a tree.

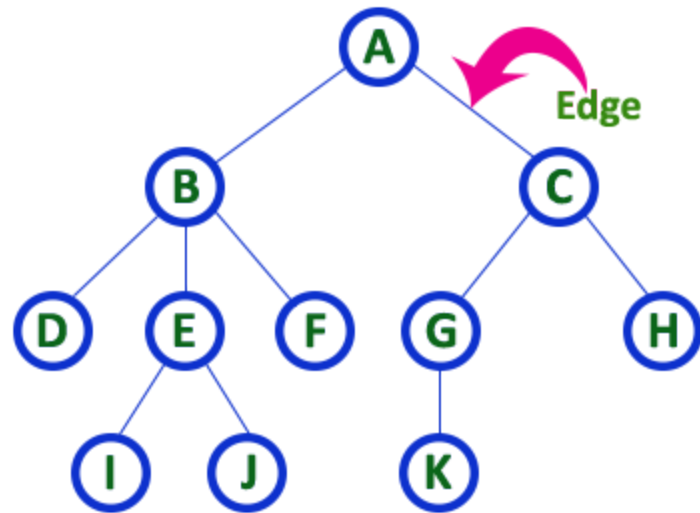


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

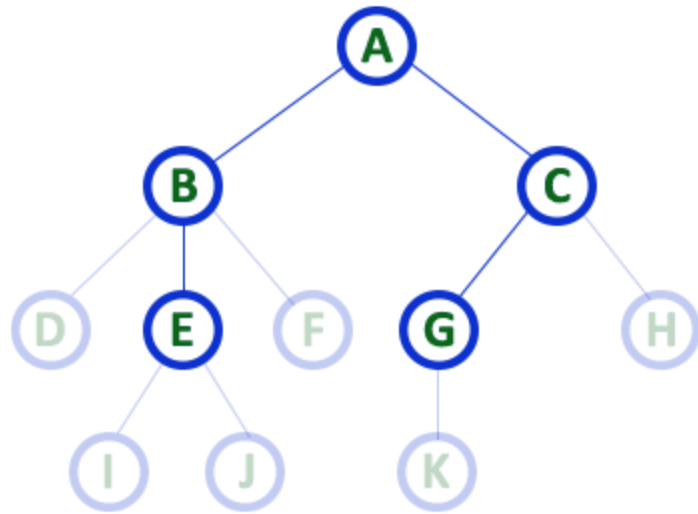
- In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

- In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**.
- In simple words, the node which has a branch from it to any other node is called a parent node.
- Parent node can also be defined as "**The node which has child / children**".

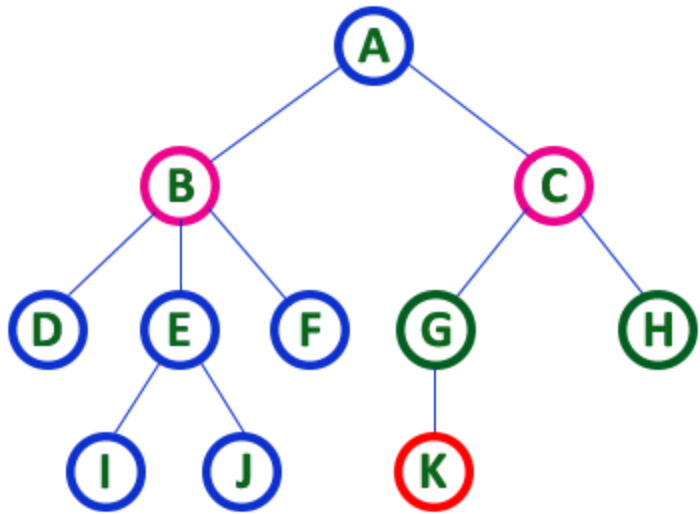


Here A, B, C, E & G are **Parent** nodes

- In any tree the node which has child / children is called '**Parent**'
- A node which is predecessor of any other node is called '**Parent**'

4. Child

- In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node.
- In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

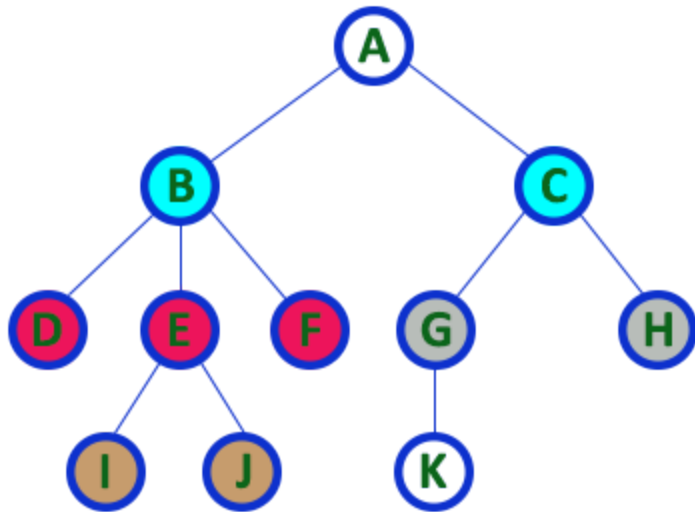


Here **B & C** are **Children of A**
Here **G & H** are **Children of C**
Here **K** is **Child of G**

- descendant of any node is called as **CHILD Node**

5. Siblings

- In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**
- In simple words, the nodes with the same parent are called Sibling nodes.

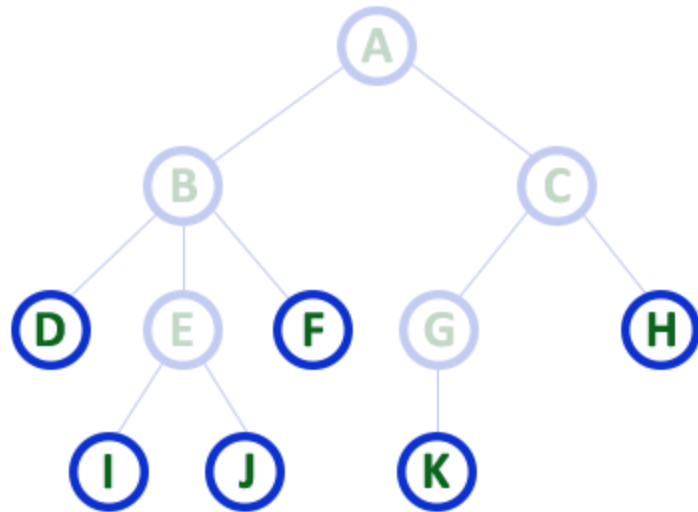


Here **B & C** are **Siblings**
Here **D E & F** are **Siblings**
Here **G & H** are **Siblings**
Here **I & J** are **Siblings**

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

- In a tree data structure, the node which does not have a child is called as **LEAF Node**.
- In simple words, a leaf is a node with no child.
- In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child.
- In a tree, leaf node is also called as 'Terminal' node.

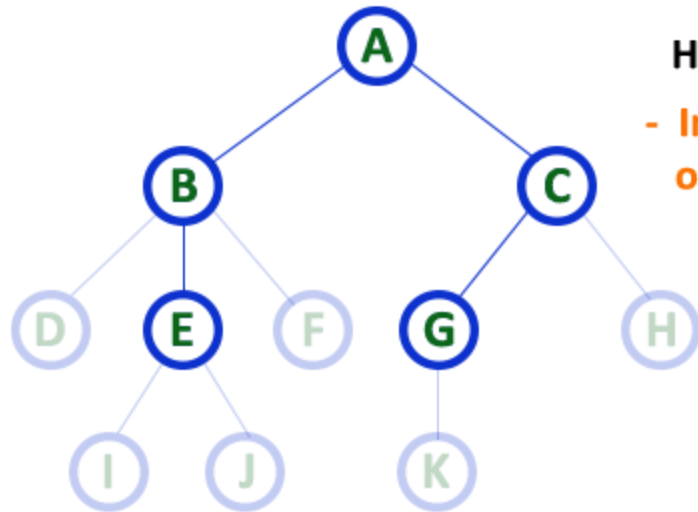


Here D, I, J, F, K & H are **Leaf** nodes

- In any tree the node which does not have children is called '**Leaf**'
- A node without successors is called a '**leaf**' node

7. Internal Nodes

- In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.
- In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The root node is also said to be Internal Node
- if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



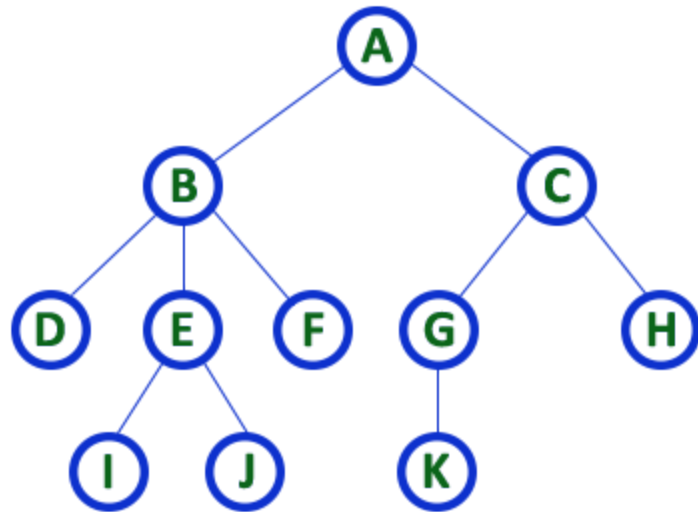
Here A, B, C, E & G are **Internal** nodes

- In any tree the node which has atleast one child is called '**Internal**' node

- Every non-leaf node is called as '**Internal**' node

8. Degree

- In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has.
- The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here **Degree** of B is 3

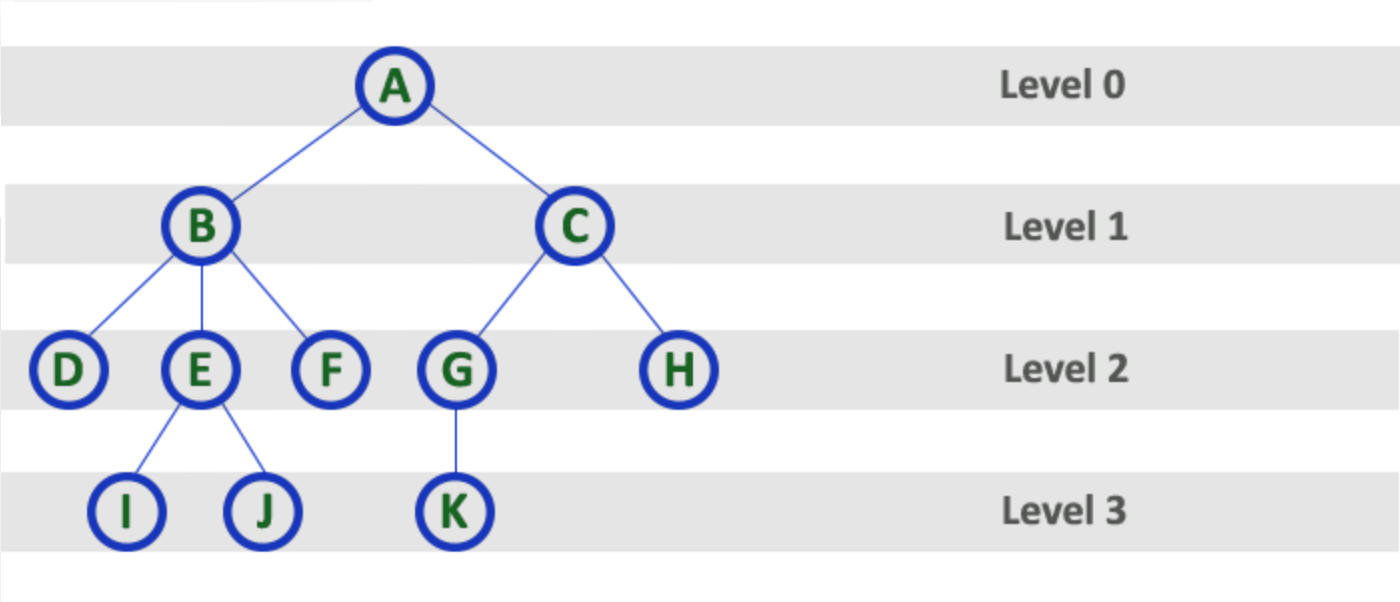
Here **Degree** of A is 2

Here **Degree** of F is 0

- In any tree, '**Degree**' of a node is total number of children it has.

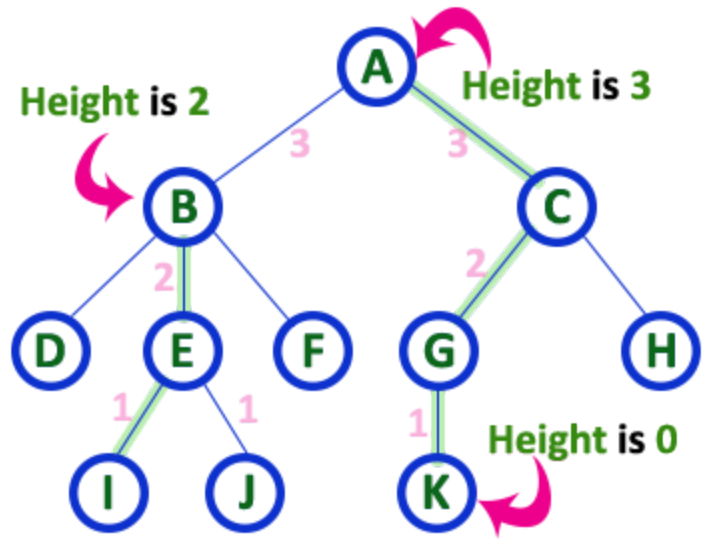
9. Level

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on
- In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

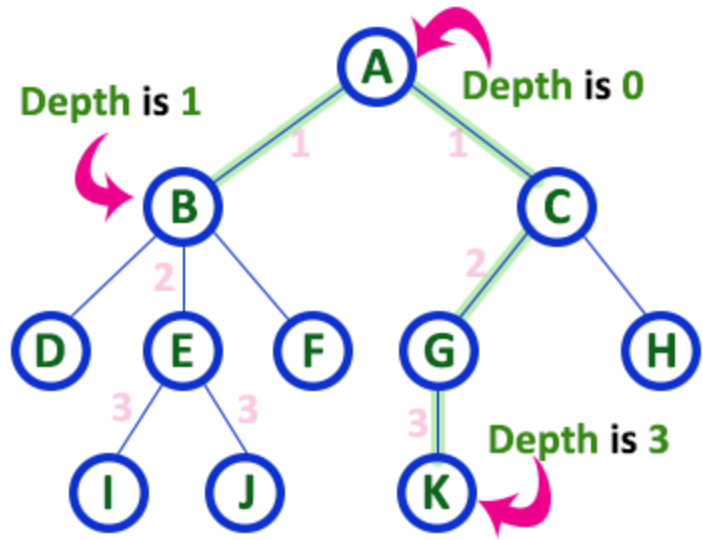
- In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node.
- In a tree, height of the root node is said to be height of the tree.
- In a tree, height of all leaf nodes is '0'.



Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

- In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node.
- In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**.
- In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree.
- In a tree, **depth of the root node is '0'**.

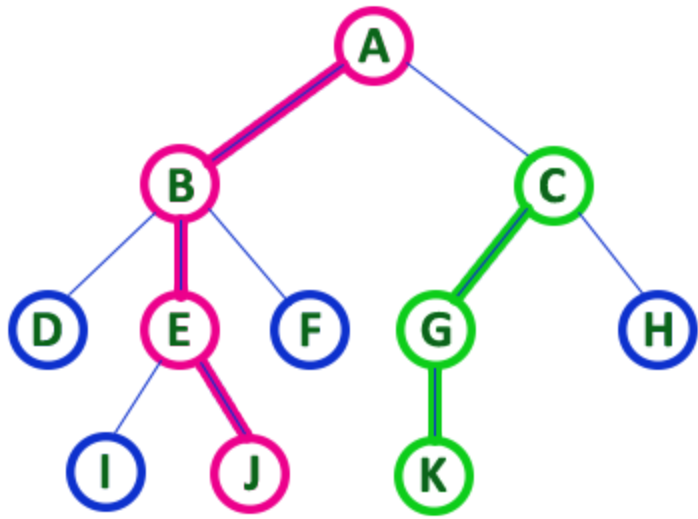


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes.
- Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

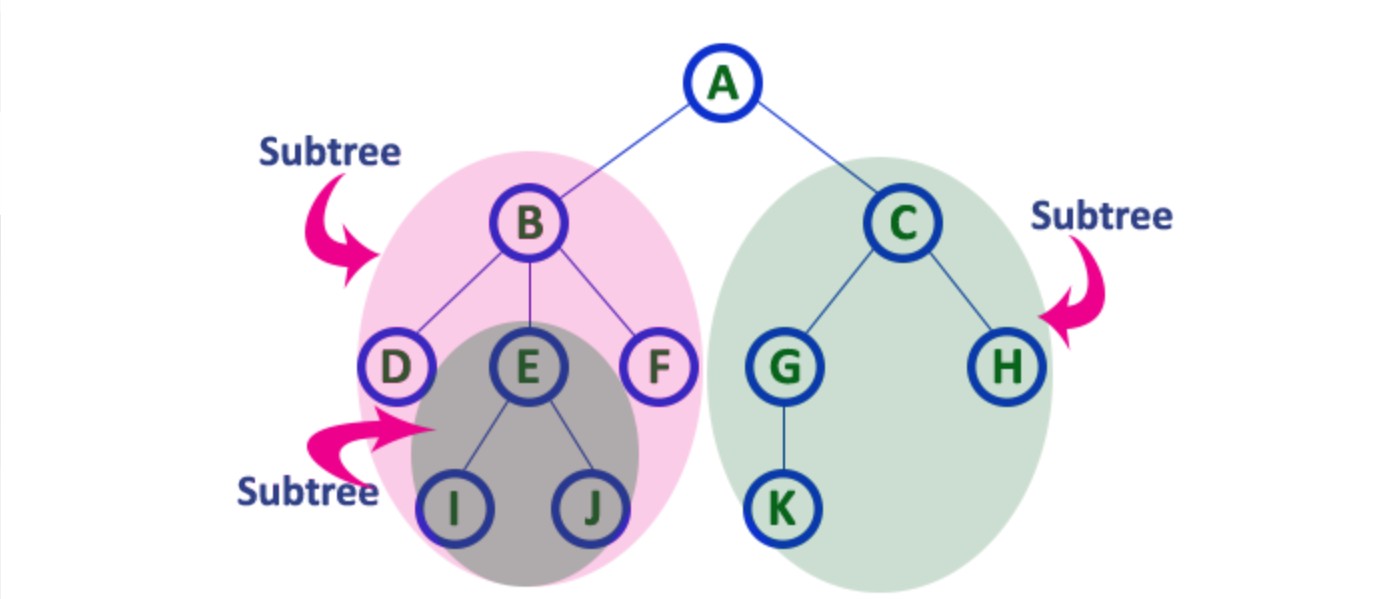
A - B - E - J

Here, 'Path' between C & K is

C - G - K

13. Sub Tree

- In a tree data structure, each child from a node forms a subtree recursively.
- Every child node will form a subtree on its parent node.



Tree Representations

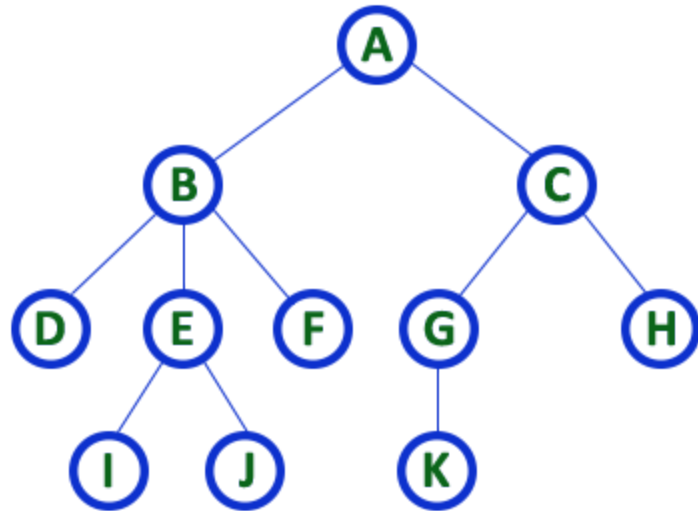
- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/tree-representations.html

Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows.

1. List Representation
2. Left Child - Right Sibling Representation

Consider the following tree.



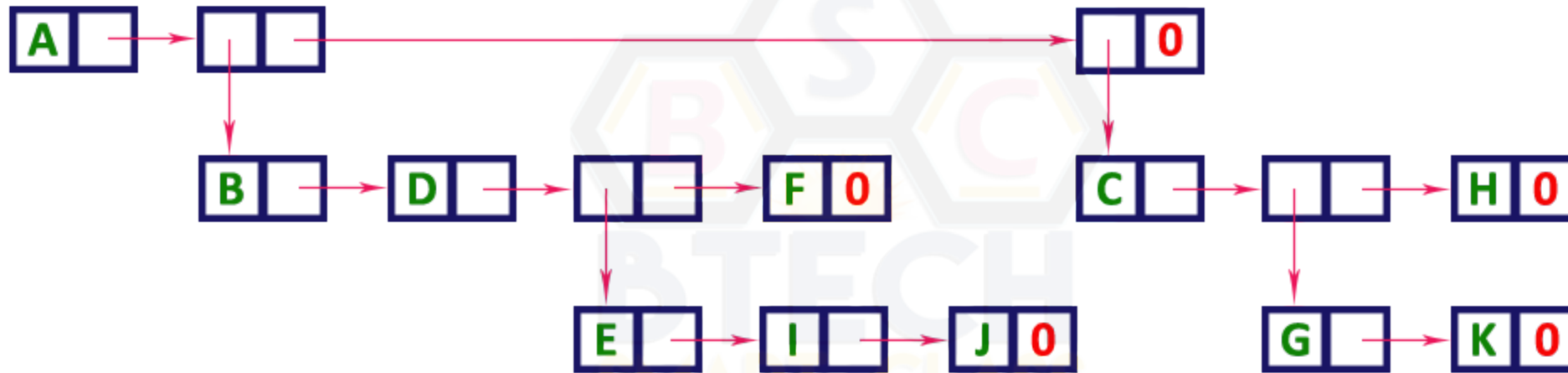
TREE with 11 nodes and 10 edges

- In any tree with '**N**' nodes there will be maximum of '**N-1**' edges
- In a tree every individual element is called as '**NODE**'

1. List Representation

- In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'.
- We start with a 'data node' from the root node in the tree.
- Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly.
- This process repeats for all the nodes in the tree.

The above example tree can be represented using List representation as follows...



2. Left Child - Right Sibling Representation

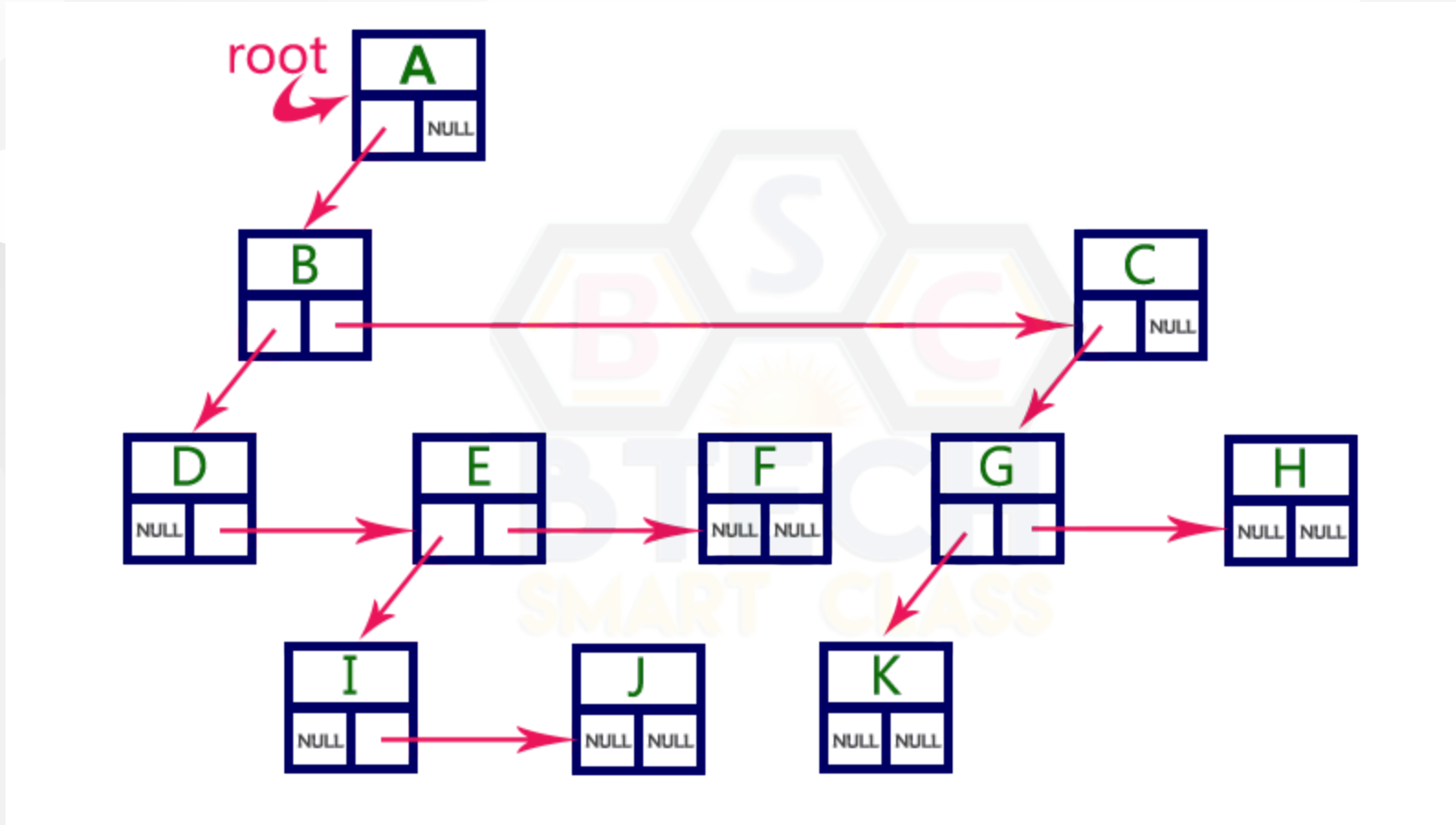
- In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field.
- Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node.

- Graphical representation of that node is as follows.



- In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL.
- If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

- The above example tree can be represented using Left Child - Right Sibling representation as follows.



Binary Tree Datastructure

1. Construction and Conversion
2. Checking and Printing
3. Summation
4. Longest Common Ancestor

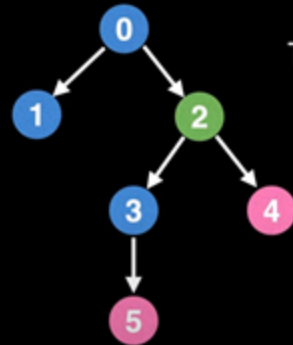
Lowest Common Ancestor in a Binary Tree - GeeksforGeeks

- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/binary-tree.html
- William Fiset
 - https://www.youtube.com/watch?v=sD1loalFomA&ab_channel=WilliamFiset

Longest Common Ancestor

Definition

The **Lowest Common Ancestor** (LCA) of two nodes `a` and `b` in a **rooted tree** is the deepest node `c` that has both `a` and `b` as descendants (where a node can be a descendant of itself)

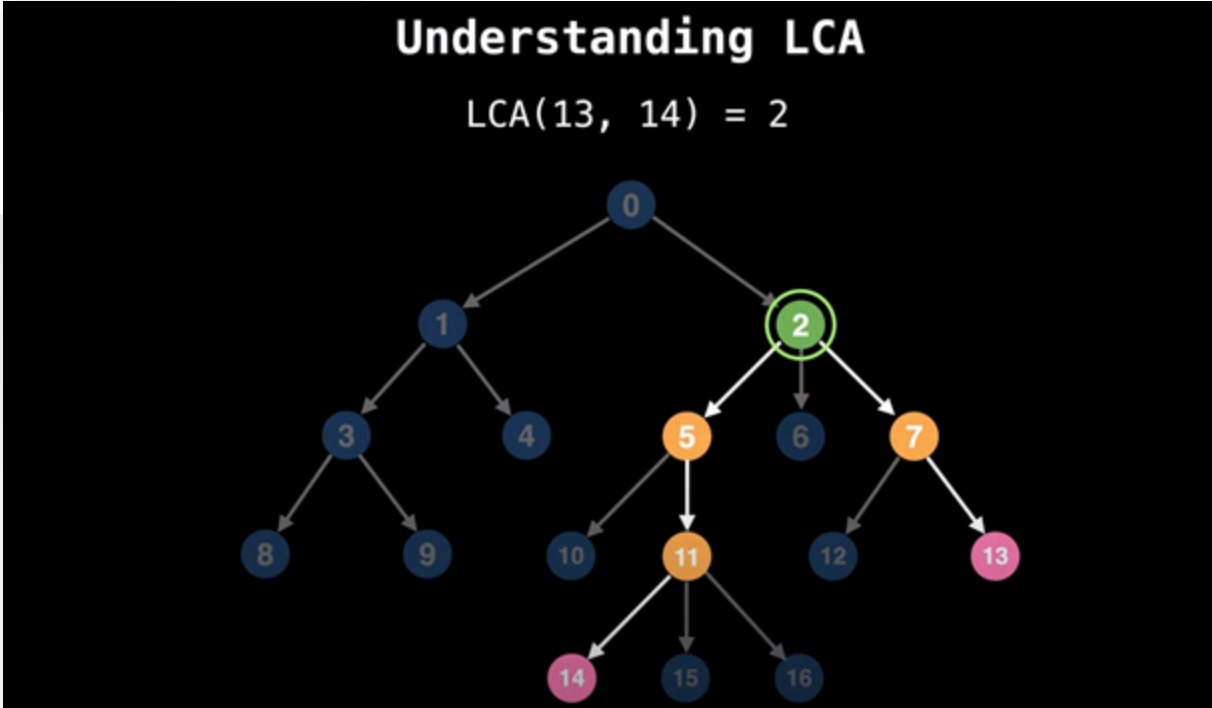


$LCA(5, 4) = 2$

The LCA problem has several applications in Computer Science, notably:

- Finding the distance between two nodes
- Inheritance hierarchies in OOP
- As a subroutine in several advanced algorithms and data structures
- etc...

Longest Common Ancestor



Lowest Common Ancestor in a Binary Tree

The **lowest common ancestor** is the lowest node in the tree that has both $n1$ and $n2$ as **descendants**, where $n1$ and $n2$ are the nodes for which we wish to find the LCA.

Hence, the LCA of a binary tree with nodes $n1$ and $n2$ is the shared ancestor of $n1$ and $n2$ that is located farthest from the root.

Application of Lowest Common Ancestor(LCA)

To determine the distance between pairs of nodes in a tree: the distance from n_1 to n_2 can be computed as the distance from the root to n_1 , plus the distance from the root to n_2 , minus twice the distance from the root to their lowest common ancestor.

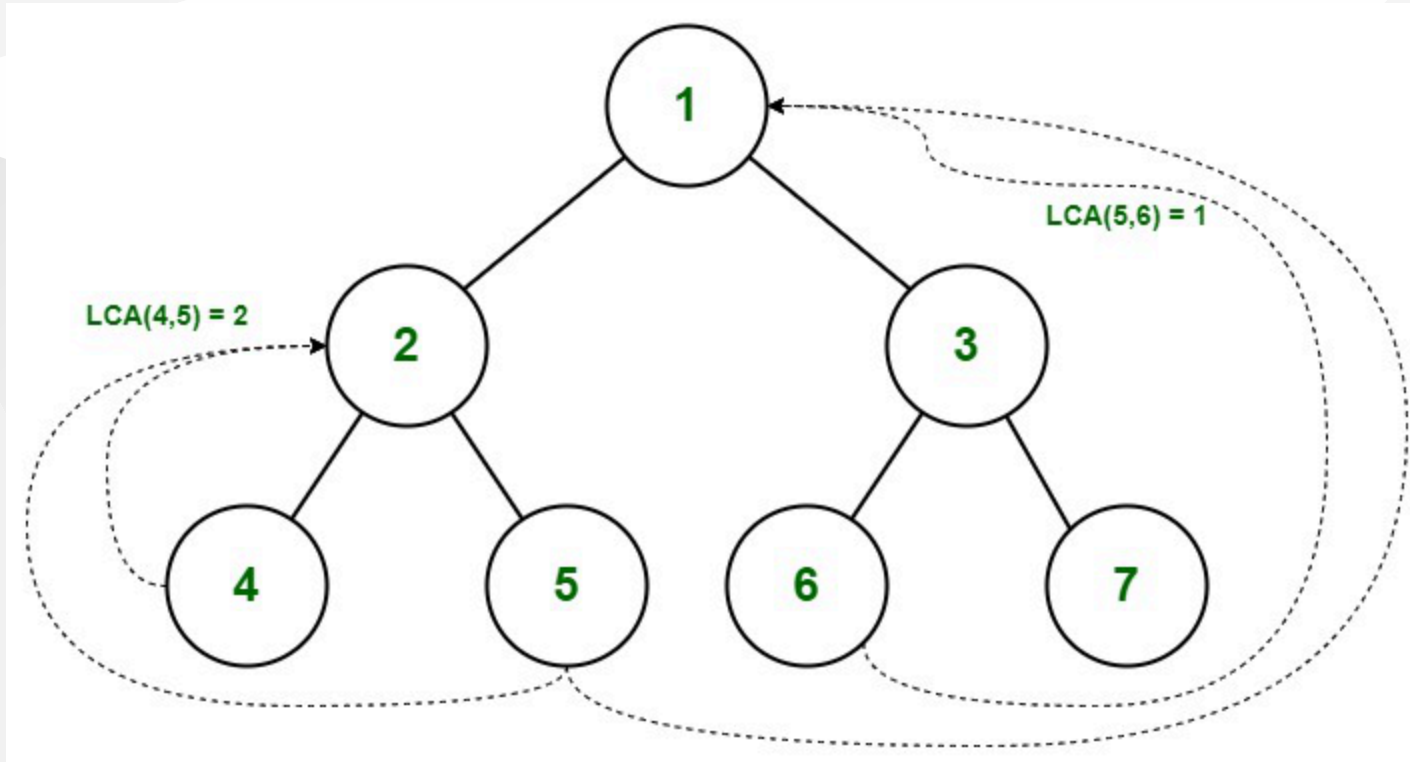
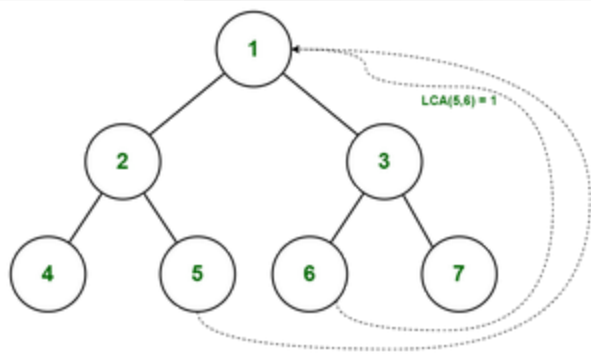


Illustration:

Find the LCA of 5 and 6

Path from root to 5 = { 1, 2, 5 }

Path from root to 6 = { 1, 3, 6 }

- We start checking from 0 index. As both of the value matches($\text{pathA}[0] = \text{pathB}[0]$), we move to the next index.
- $\text{pathA}[1]$ not equals to $\text{pathB}[1]$, there's a mismatch so we consider the previous value.
- Therefore the LCA of (5,6) = 1

LCA in C++

```

// C++ Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>

using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // Check if k is found in left or right sub-tree
    if ( (root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.pop_back();
    return false;
}

// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n2. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2) )
        return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size(); i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
    return 0;
}

```

LCA in Java

```

// Java Program for Lowest Common Ancestor in a Binary Tree
// A O(n) solution to find LCA of two given values n1 and n2
import java.util.ArrayList;
import java.util.List;

// A Binary Tree node
class Node {
    int data;
    Node left, right;
    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class BT_NoParentPtr_Solution1
{
    Node root;
    private List<Integer> path1 = new ArrayList<>();
    private List<Integer> path2 = new ArrayList<>();

    // Finds the path from root node to given root of the tree.
    int findLCA(int n1, int n2) {
        path1.clear();
        path2.clear();
        return findLCAInternal(root, n1, n2);
    }

    private int findLCAInternal(Node root, int n1, int n2) {
        if (!findPath(root, n1, path1) || !findPath(root, n2, path2)) {
            System.out.println((path1.size() > 0) ? "n1 is missing": "n2 is missing");
            return -1;
        }

        int i;
        for (i = 0; i < path1.size() && i < path2.size(); i++) {
            // System.out.println(path1.get(i) + " " + path2.get(i));
            if (path1.get(i).equals(path2.get(i)))
                break;
        }

        return path1.get(i-1);
    }

    // Finds the path from root node to given root of the tree, Stores the
    // path in a vector path[], returns true if path exists otherwise false
    private boolean findPath(Node root, int n, List<Integer> path)
    {
        // base case
        if (root == null) {
            return false;
        }

        // Store this node . The node will be removed if
        // not in path from root to n.
        path.add(root.data);

        if (root.data == n) {
            return true;
        }

        if (root.left != null && findPath(root.left, n, path)) {
            return true;
        }

        if (root.right != null && findPath(root.right, n, path)) {
            return true;
        }

        // If not present in subtree rooted with root, remove root from
        // path[] and return false
        path.remove(path.size()-1);

        return false;
    }

    // Driver code
    public static void main(String[] args)
    {
        BT_NoParentPtr_Solution1 tree = new BT_NoParentPtr_Solution1();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);

        System.out.println("LCA(4, 5): " + tree.findLCA(4,5));
        System.out.println("LCA(4, 6): " + tree.findLCA(4,6));
        System.out.println("LCA(3, 4): " + tree.findLCA(3,4));
        System.out.println("LCA(2, 4): " + tree.findLCA(2,4));
    }
}
// This code is contributed by Sreenivasulu Rayanki.

```

LCA in C#

```

// C# Program for Lowest Common
// Ancestor in a Binary Tree
// A O(n) solution to find LCA
// of two given values n1 and n2
using System.Collections;
using System;

// A Binary tree node
class Node
{
public int data;
public Node left, right;
public Node(int value)
{
    data = value;
    left = right = null;
}
}

public class BT_NPParentPr_Solution1
{
Node root;
private ArrayList path1 =
    new ArrayList();
private ArrayList path2 =
    new ArrayList();

// Finds the path from root node
// to given root of the tree
// tree:
int FindLCA(int n1, int n2)
{
    path1.Clear();
    path2.Clear();
    return FindLCAInternal(root,
        n1, n2);
}

private int FindLCAInternal(Node root,
    int n1, int n2)
{
if ((FindPath(root, n1, path1) &&
    FindPath(root, n2, path2)) {
    Console.WriteLine(path1.Count > 0)
        ? "n1 is present" :
        "n1 is missing");
    Console.WriteLine(path2.Count > 0)
        ? "n2 is present" :
        "n2 is missing");
    return -1;
}

int i = 0; i < path1.Count &&
i < path2.Count; i++)
{
// System.out.println(path1.get(i))
// /- /- /- path2.get(i));
if ((int)path1[i] ==
    (int)path2[i])
        break;
}

return (int)path1[i - 1];
}

// Finds the path from root node
// to given root of the tree.
// Stores the path in a vector
// path1, returns true if path
// exists otherwise false
private bool FindPath(Node root,
    int n,
    ArrayList path)
{
// base case
if (root == null)
{
    return false;
}

// Store this node - The node
// will be removed if not in
// path from root to n.
path.Add(root.data);

if (root.data == n)
{
    return true;
}

if (root.left != null &&
    FindPath(root.left,
        n, path))
{
    return true;
}

if (root.right != null &&
    FindPath(root.right,
        n, path))
{
    return true;
}

// If not present in subtree
// (rooted with root, remove root
// from path) and return false
path.Remove(path.Count - 1);
return false;
}

// Driver code
public static void Main(String[] args)
{
BT_NPParentPr_Solution1 tree =
    new BT_NPParentPr_Solution1();

tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);
tree.root.right.left = new Node(6);
tree.root.right.right = new Node(7);

Console.WriteLine(LCA(4, 5));
Console.WriteLine(LCA(4, 6));
Console.WriteLine(LCA(4, 7));
Console.WriteLine(LCA(2, 4));
Console.WriteLine(LCA(2, 5));
Console.WriteLine(LCA(2, 6));
Console.WriteLine(LCA(2, 7));
}

// This code is contributed by Ravi01_36

```

Output

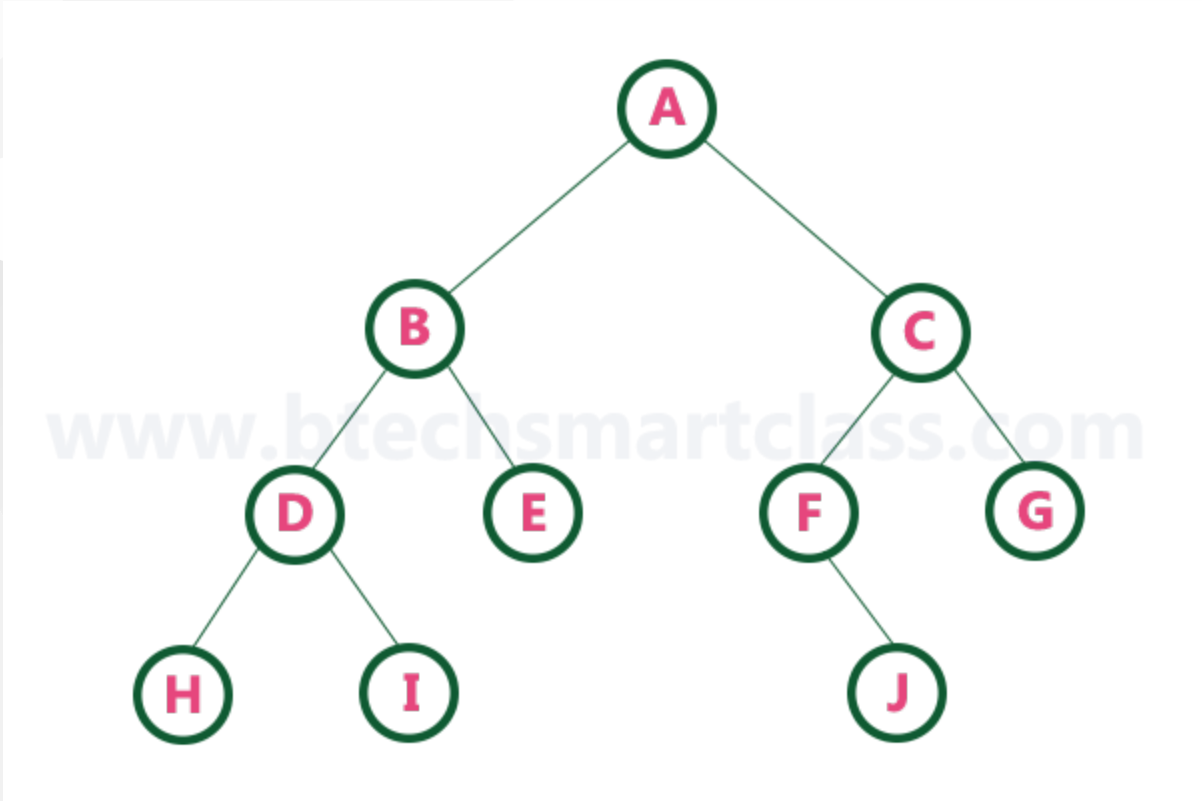
```
LCA(4, 5) = 2  
LCA(4, 6) = 1  
LCA(3, 4) = 1  
LCA(2, 4) = 2
```

- **Time Complexity:** $O(n)$. The tree is traversed twice, and then path arrays are compared.
- **Auxiliary Space:** $O(n)$. Extra Space for path1 and path2.

Binary Tree Datastructure

- In a normal tree, every node can have any number of children.
- A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**.
- One is known as a left child and the other is known as right child.
- A tree in which every node can have a maximum of two children is called Binary Tree.
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

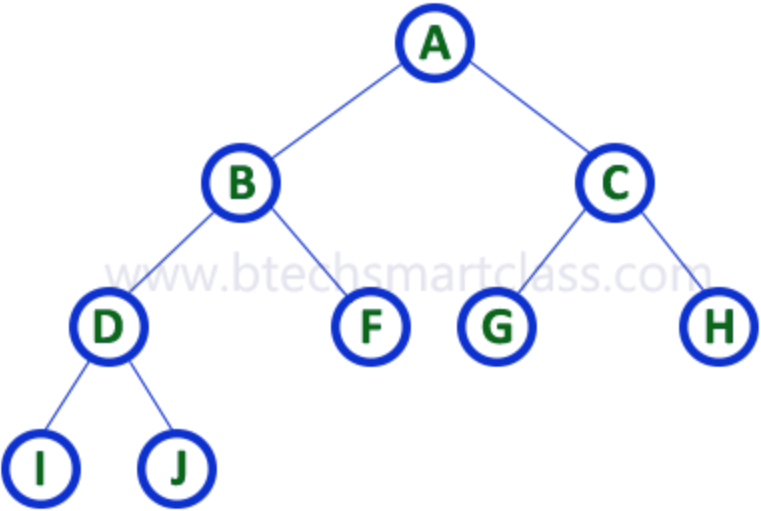
Example



There are different types of binary trees and they are

1. Strictly Binary Tree (Full Binary Tree / Proper Binary Tree or 2-Tree)

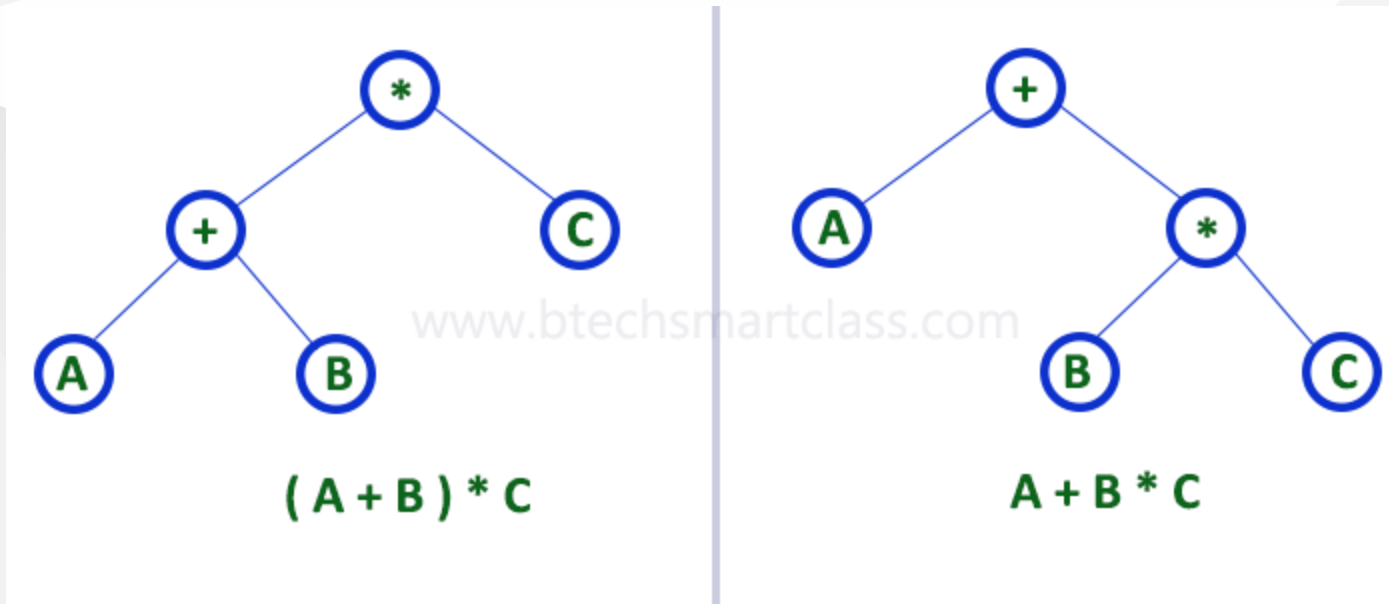
- In a binary tree, every node can have a maximum of two children.
- But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children.
- A strictly Binary Tree can be defined as follows.
- A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree
- Strictly binary tree is also called as **Full Binary Tree or Proper Binary Tree or 2-Tree**



www.btechsmartclass.com

Example

- Strictly binary tree data structure is used to represent mathematical expressions.



Full Binary Tree Theorems

Let, i = the number of internal nodes
 n = be the total number of nodes
 l = number of leaves
 λ = number of levels

Full Binary Tree Theorems

1. The number of leaves is $i + 1$.
2. The total number of nodes is $2i + 1$.
3. The number of internal nodes is $(n - 1) / 2$.
4. The number of leaves is $(n + 1) / 2$.
5. The total number of nodes is $2l - 1$.
6. The number of internal nodes is $l - 1$.
7. The number of leaves is at most $2\lambda - 1$.

Full Binary Tree in C

```

// Checking if a binary tree is a full binary tree in C

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int item;
    struct Node *left, *right;
};

// Creation of new Node
struct Node *createNewNode(char k) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->item = k;
    node->right = node->left = NULL;
    return node;
}

bool isFullBinaryTree(struct Node *root) {
    // Checking tree emptiness
    if (root == NULL)
        return true;

    // Checking the presence of children
    if (root->left == NULL && root->right == NULL)
        return true;

    if ((root->left) && (root->right))
        return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

    return false;
}

int main() {
    struct Node *root = NULL;
    root = createNewNode(1);
    root->left = createNewNode(2);
    root->right = createNewNode(3);

    root->left->left = createNewNode(4);
    root->left->right = createNewNode(5);
    root->left->right->left = createNewNode(6);
    root->left->right->right = createNewNode(7);

    if (isFullBinaryTree(root))
        printf("The tree is a full binary tree\n");
    else
        printf("The tree is not a full binary tree\n");
}

```

Full Binary Tree in C++

```
// Checking if a binary tree is a full binary tree in C++

#include <iostream>
using namespace std;

struct Node {
    int key;
    struct Node *left, *right;
};

// New node creation
struct Node *newNode(char k) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

bool isFullBinaryTree(struct Node *root) {

    // Checking for emptiness
    if (root == NULL)
        return true;

    // Checking for the presence of children
    if (root->left == NULL && root->right == NULL)
        return true;

    if ((root->left) && (root->right))
        return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

    return false;
}

int main() {
    struct Node *root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->left = newNode(6);
    root->left->right->right = newNode(7);

    if (isFullBinaryTree(root))
        cout << "The tree is a full binary tree\n";
    else
        cout << "The tree is not a full binary tree\n";
}
```

Full Binary in Java

```
// Checking if a binary tree is a full binary tree in Java

class Node {
    int data;
    Node leftChild, rightChild;

    Node(int item) {
        data = item;
        leftChild = rightChild = null;
    }
}

class BinaryTree {
    Node root;

    // Check for Full Binary Tree
    boolean isFullBinaryTree(Node node) {

        // Checking tree emptiness
        if (node == null)
            return true;

        // Checking the children
        if (node.leftChild == null && node.rightChild == null)
            return true;

        if ((node.leftChild != null) && (node.rightChild != null))
            return (isFullBinaryTree(node.leftChild) && isFullBinaryTree(node.rightChild));

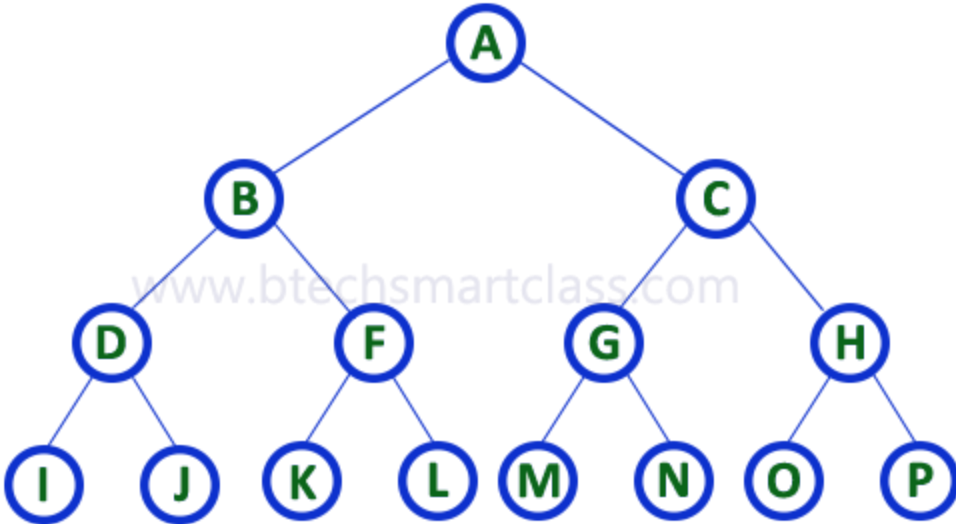
        return false;
    }

    public static void main(String args[]) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.leftChild = new Node(2);
        tree.root.rightChild = new Node(3);
        tree.root.leftChild.leftChild = new Node(4);
        tree.root.leftChild.rightChild = new Node(5);
        tree.root.rightChild.leftChild = new Node(6);
        tree.root.rightChild.rightChild = new Node(7);

        if (tree.isFullBinaryTree(tree.root))
            System.out.print("The tree is a full binary tree");
        else
            System.out.print("The tree is not a full binary tree");
    }
}
```

2. Complete Binary Tree (Perfect Binary Tree)

- In a binary tree, every node can have a maximum of two children.
- But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes.
- For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.
- A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.
- Complete binary tree is also called as **Perfect Binary Tree**



Perfect Binary Tree Theorems

1. A perfect binary tree of height h has $2^{h+1} - 1$ nodes.
2. A perfect binary tree with n nodes has height $\log_2(n + 1) - 1 = \Theta(\ln(n))$.
3. A perfect binary tree of height h has 2^h leaf nodes.
4. The average depth of a node in a perfect binary tree is $\Theta(\ln(n))$.

Perfect Binary Tree in C

```

// Checking if a binary tree is a perfect binary tree in C
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *left;
    struct node *right;
};

// Creating a new node
struct node *newnode(int data) {
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

// Calculate the depth
int depth(struct node *node) {
    int d = 0;
    while (node != NULL) {
        d++;
        node = node->left;
    }
    return d;
}

// Check if the tree is perfect
bool is_perfect(struct node *root, int d, int level) {
    // Check if the tree is empty
    if (root == NULL)
        return true;

    // Check the presence of children
    if (root->left == NULL && root->right == NULL)
        return (d == level + 1);

    if (root->left == NULL || root->right == NULL)
        return false;

    return is_perfect(root->left, d, level + 1) &&
        is_perfect(root->right, d, level + 1);
}

// Wrapper function
bool is_Perfect(struct node *root) {
    int d = depth(root);
    return is_perfect(root, d, 0);
}

int main() {
    struct node *root = NULL;
    root = newnode(1);
    root->left = newnode(2);
    root->right = newnode(3);
    root->left->left = newnode(4);
    root->left->right = newnode(5);
    root->right->left = newnode(6);

    if (is_Perfect(root))
        printf("The tree is a perfect binary tree\n");
    else
        printf("The tree is not a perfect binary tree\n");
}

```

Perfect Binary Tree in C++

```
// Checking if a binary tree is a perfect binary tree in C++
#include <iostream>
using namespace std;

struct Node {
    int key;
    struct Node *left, *right;
};

int depth(Node *node) {
    int d = 0;
    while (node != NULL) {
        d++;
        node = node->left;
    }
    return d;
}

bool isPerfectR(struct Node *root, int d, int level = 0) {
    if (root == NULL)
        return true;

    if (root->left == NULL && root->right == NULL)
        return (d == level + 1);

    if (root->left == NULL || root->right == NULL)
        return false;

    return isPerfectR(root->left, d, level + 1) &&
        isPerfectR(root->right, d, level + 1);
}

bool isPerfect(Node *root) {
    int d = depth(root);
    return isPerfectR(root, d);
}

struct Node *newNode(int k) {
    struct Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

int main() {
    struct Node *root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);

    if (isPerfect(root))
        cout << "The tree is a perfect binary tree\n";
    else
        cout << "The tree is not a perfect binary tree\n";
}
```

Perfect Binary Tree in Java

```
// Checking if a binary tree is a perfect binary tree in Java
class PerfectBinaryTree {
    static class Node {
        int key;
        Node left, right;
    }

    // Calculate the depth
    static int depth(Node node) {
        int d = 0;
        while (node != null) {
            d++;
            node = node.left;
        }
        return d;
    }

    // Check if the tree is perfect binary tree
    static boolean is_perfect(Node root, int d, int level) {

        // Check if the tree is empty
        if (root == null)
            return true;

        // If for children
        if (root.left == null && root.right == null)
            return (d == level + 1);

        if (root.left == null || root.right == null)
            return false;

        return is_perfect(root.left, d, level + 1) && is_perfect(root.right, d, level + 1);
    }

    // Wrapper function
    static boolean is_Perfect(Node root) {
        int d = depth(root);
        return is_perfect(root, d, 0);
    }

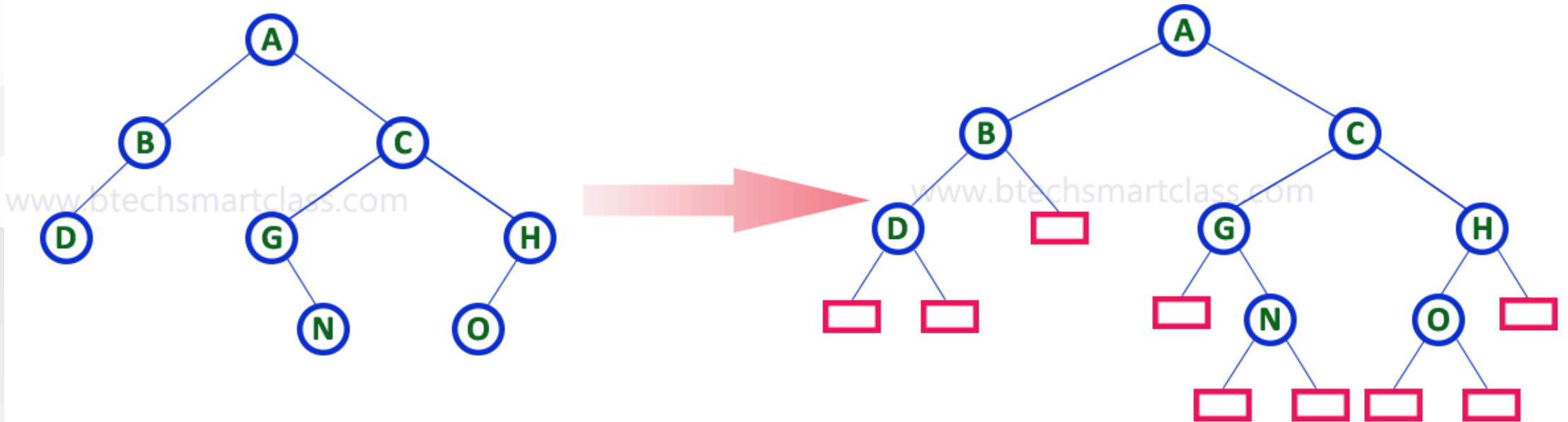
    // Create a new node
    static Node newNode(int k) {
        Node node = new Node();
        node.key = k;
        node.right = null;
        node.left = null;
        return node;
    }

    public static void main(String args[]) {
        Node root = null;
        root = newNode(1);
        root.left = newNode(2);
        root.right = newNode(3);
        root.left.left = newNode(4);
        root.left.right = newNode(5);

        if (is_Perfect(root) == true)
            System.out.println("The tree is a perfect binary tree");
        else
            System.out.println("The tree is not a perfect binary tree");
    }
}
```

3. Extended Binary Tree

- A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.
- The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

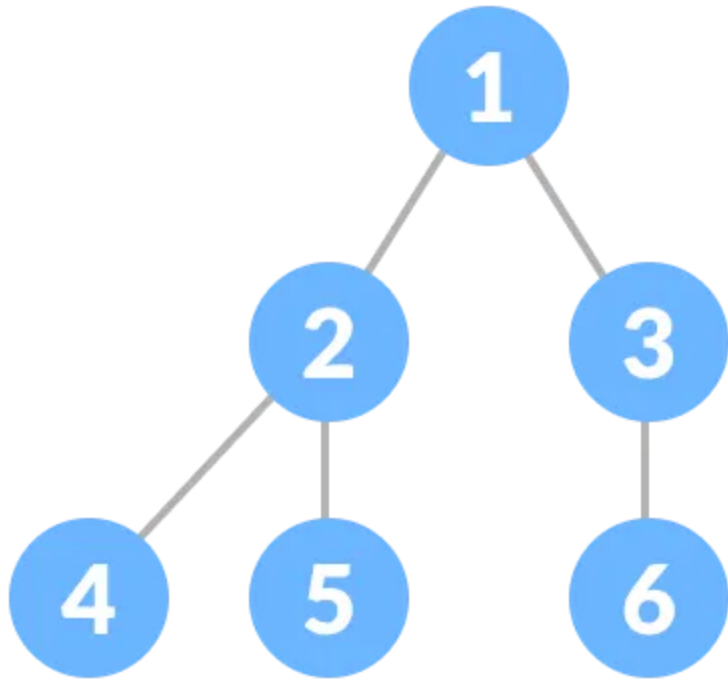


- In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

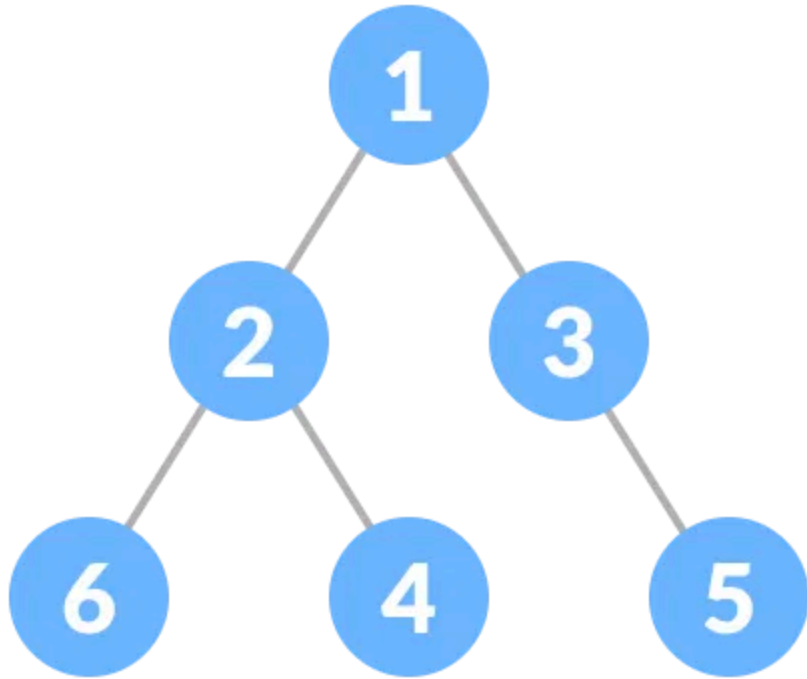
Complete Binary Tree

- A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.
- A complete binary tree is just like a full binary tree, but with two major differences
 - All the leaf elements must lean towards the left.
 - The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

Complete Binary Tree



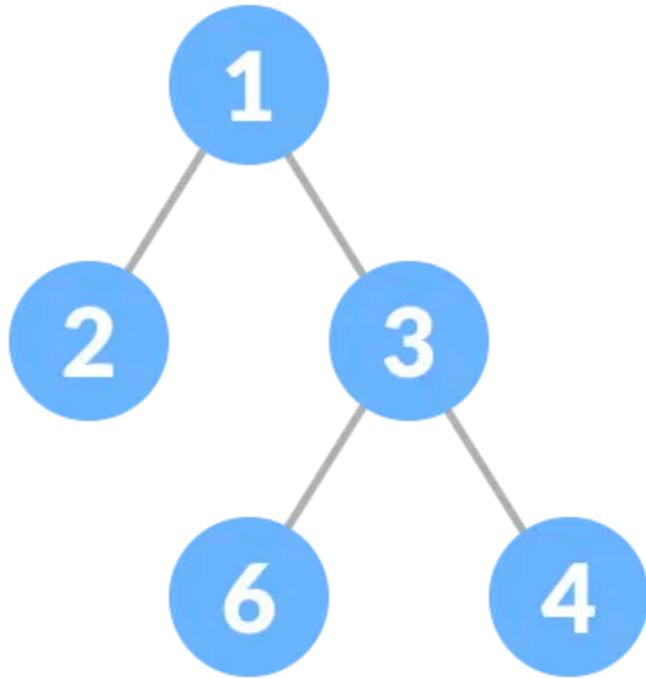
Comparison between full binary tree and complete binary tree



✘ Full Binary Tree

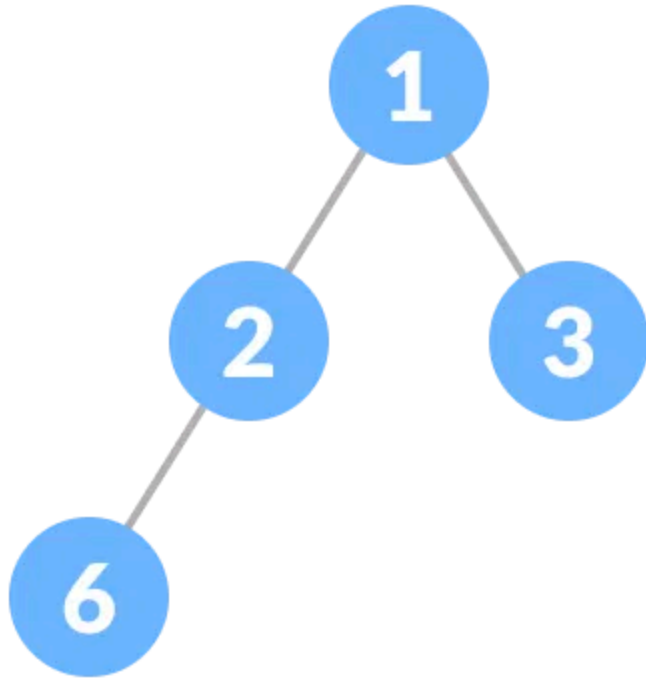
✘ Complete Binary Tree

Comparison between full binary tree and complete binary tree



- ✓ Full Binary Tree
- ✗ Complete Binary Tree

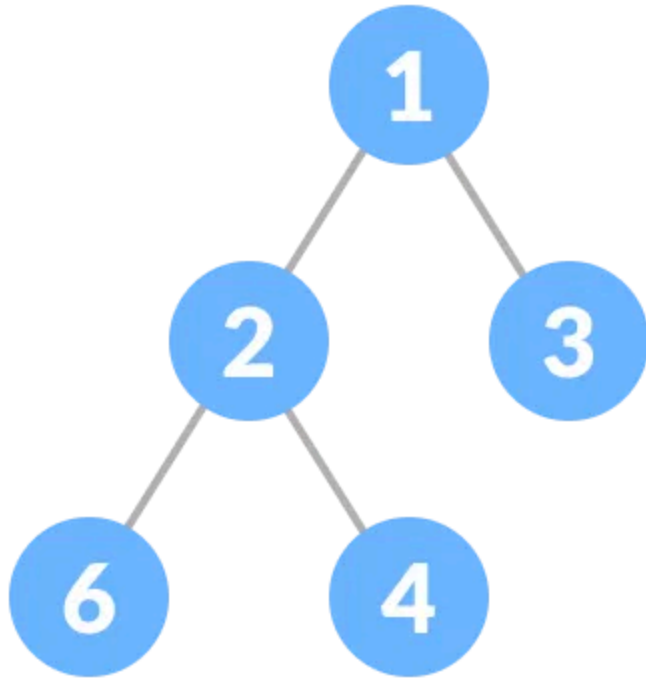
Comparison between full binary tree and complete binary tree



✘ Full Binary Tree

✔ Complete Binary Tree

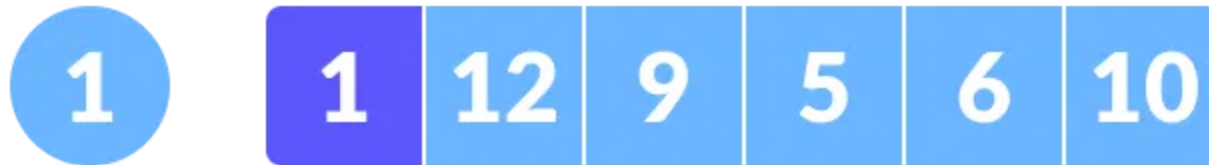
Comparison between full binary tree and complete binary tree



- ✓ Full Binary Tree
- ✓ Complete Binary Tree

How a Complete Binary Tree is Created?

1. Select the first element of the list to be the root node. (no. of elements on level-l: 1)



Select the first element as root

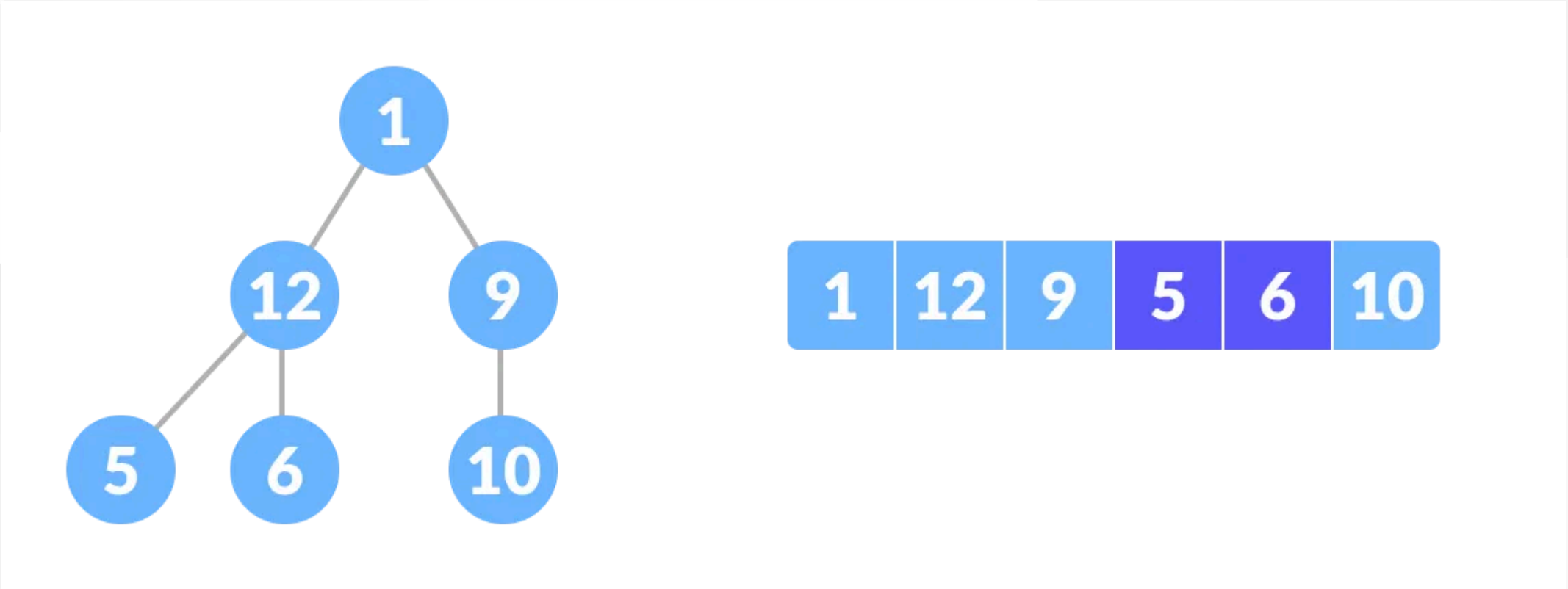
1. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



12 as a left child and 9 as a right child

1. Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4) elements).

1. Keep repeating until you reach the last element.



5 as a left child and 6 as a right child

Relationship between array indexes and tree element

- 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 $2i+1$ will become the left child and element in $2i+2$ index will become the right child.
- Also, the parent of any element at index i is given by the lower bound of $(i-1)/2$.

Complete Binary Tree in C

```
// Checking if a binary tree is a complete binary tree in C
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    struct Node *left, *right;
};

// Node creation
struct Node *newNode(char k) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// Count the number of nodes
int countNumNodes(struct Node *root) {
    if (root == NULL)
        return (0);
    return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}

// Check if the tree is a complete binary tree
bool checkComplete(struct Node *root, int index, int numberNodes) {
    // Check if the tree is complete
    if (root == NULL)
        return true;

    if (index >= numberNodes)
        return false;

    return (checkComplete(root->left, 2 * index + 1, numberNodes) && checkComplete(root->right, 2 * index + 2, numberNodes));
}

int main() {
    struct Node *root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);

    int node_count = countNumNodes(root);
    int index = 0;

    if (checkComplete(root, index, node_count))
        printf("The tree is a complete binary tree\n");
    else
        printf("The tree is not a complete binary tree\n");
}
```

Complete Binary Tree in C++

```

// Checking if a binary tree is a complete binary tree in C++
#include <iostream>
using namespace std;

struct Node {
    int key;
    struct Node *left, *right;
};

// Create node
struct Node *newNode(char k) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// Count the number of nodes
int countNumNodes(struct Node *root) {
    if (root == NULL)
        return (0);
    return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}

// Check if the tree is a complete binary tree
bool checkComplete(struct Node *root, int index, int numberNodes) {

    // Check if the tree is empty
    if (root == NULL)
        return true;

    if (index >= numberNodes)
        return false;

    return (checkComplete(root->left, 2 * index + 1, numberNodes) && checkComplete(root->right, 2 * index + 2, numberNodes));
}

int main() {
    struct Node *root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);

    int node_count = countNumNodes(root);
    int index = 0;

    if (checkComplete(root, index, node_count))
        cout << "The tree is a complete binary tree\n";
    else
        cout << "The tree is not a complete binary tree\n";
}

```

Complete Binary Tree in Java

```
// Checking if a binary tree is a complete binary tree in Java

// Node creation
class Node {
    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;

    // Count the number of nodes
    int countNumNodes(Node root) {
        if (root == null)
            return 0;
        return 1 + countNumNodes(root.left) + countNumNodes(root.right);
    }

    // Check for complete binary tree
    boolean checkComplete(Node root, int index, int numberNodes) {

        // Check if the tree is empty
        if (root == null)
            return true;

        if (index >= numberNodes)
            return false;

        return (checkComplete(root.left, 2 * index + 1, numberNodes)
            && checkComplete(root.right, 2 * index + 2, numberNodes));
    }

    public static void main(String args[]) {
        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.right = new Node(5);
        tree.root.left.left = new Node(4);
        tree.root.right.left = new Node(6);

        int node_count = tree.countNumNodes(tree.root);
        int index = 0;

        if (tree.checkComplete(tree.root, index, node_count))
            System.out.println("The tree is a complete binary tree");
        else
            System.out.println("The tree is not a complete binary tree");
    }
}
```

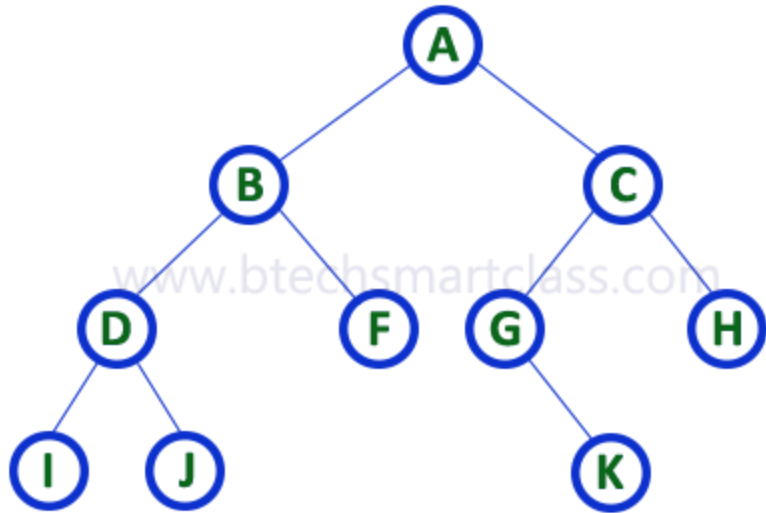
Binary Tree Representations

- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/binary-tree-representations.html

Binary Tree Representations

- A binary tree data structure is represented using two methods. Those methods are as follows.
 - Array Representation
 - Linked List Representation

- Consider the following binary tree



1. Array Representation of Binary Tree

- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
- Consider the above example of a binary tree and it is represented as follows.



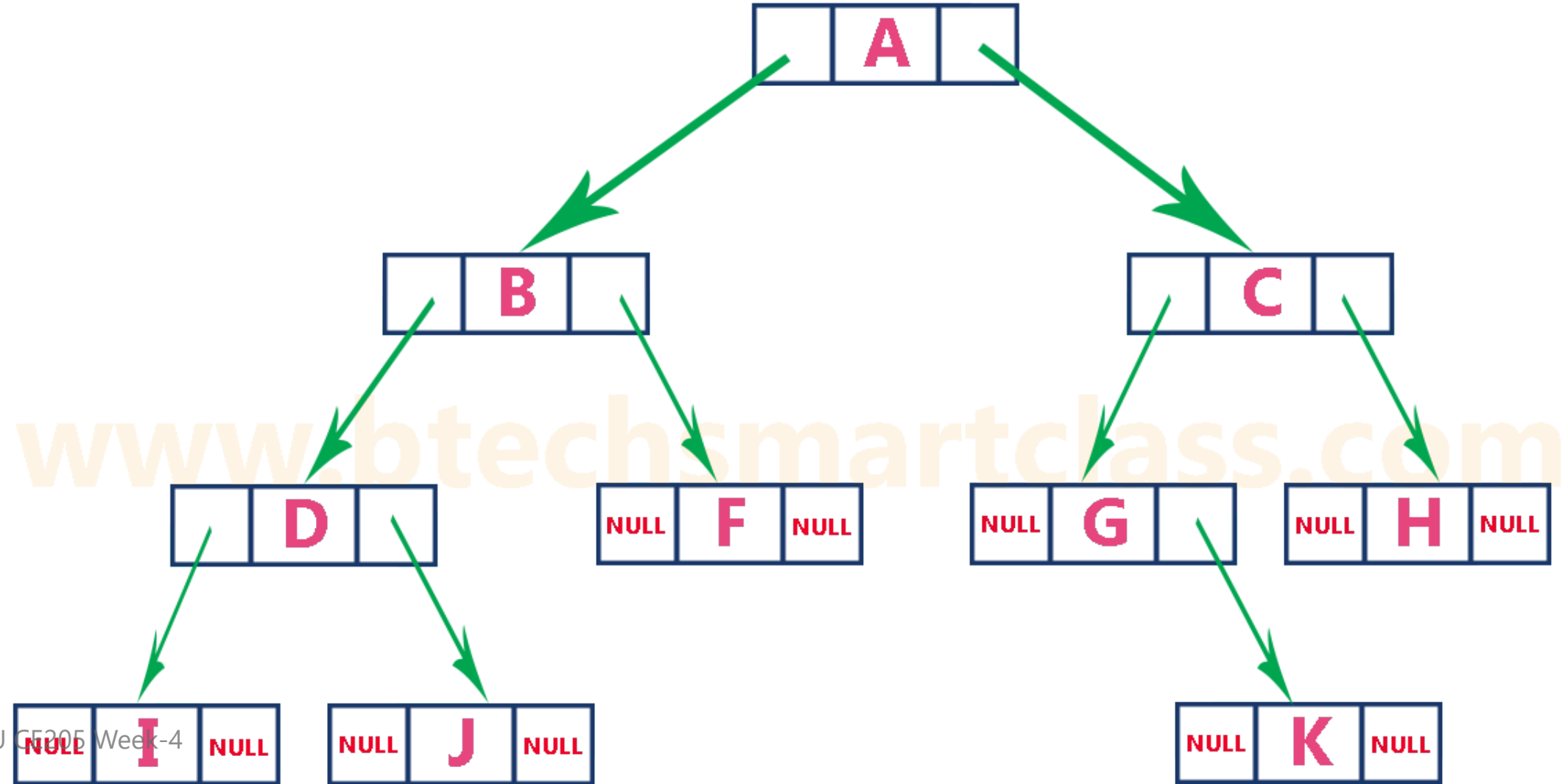
- To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2n + 1$.

2. Linked List Representation of Binary Tree

- We use a double linked list to represent a binary tree.
- In a double linked list, every node consists of three fields.
- First field for storing left child address, second for storing actual data and third for storing right child address.
- In this linked list representation, a node has the following structure.



rootNode



www.btechsmartclass.com



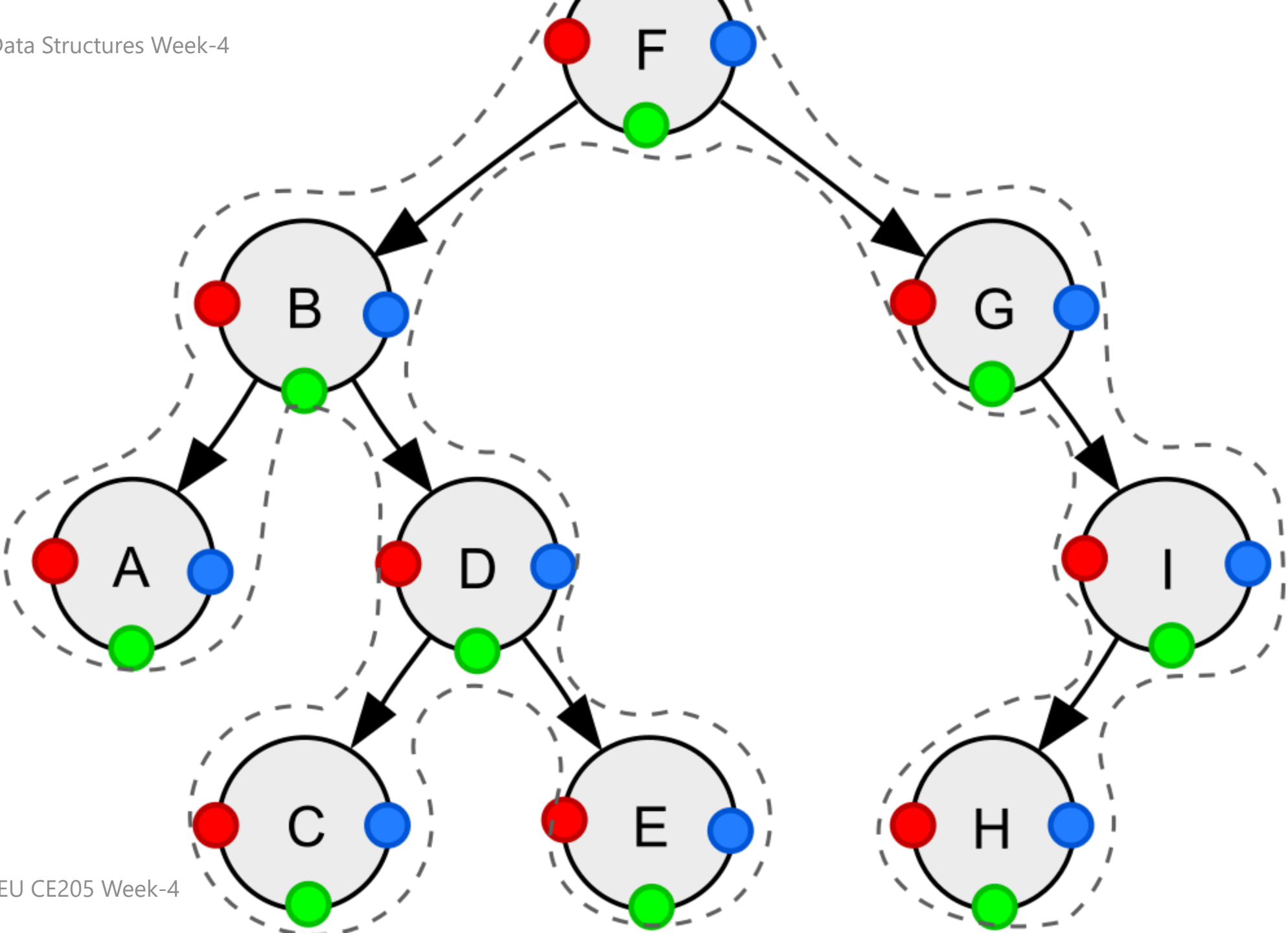
Binary Tree Traversals

- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/binary-tree-traversals.html
 - In-Order
 - Pre-Order
 - Post-Order

Binary Tree Traversals

- When we wanted to display a binary tree,
- we need to follow some order in which all the nodes of that binary tree must be displayed.
- In any binary tree, displaying order of nodes depends on the traversal method.
- Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

- There are three types of binary tree traversals.
 - In - Order Traversal
 - Pre - Order Traversal
 - Post - Order Traversal



Notations We Will Use For Orders

NLR : Node Left Right

LRN : Left Right Node

LNR : Left Node Right

Pre-order, NLR

1. Visit the current node (in the figure: position red).
2. Recursively traverse the current node's left subtree.
3. Recursively traverse the current node's right subtree.

The pre-order traversal is a **topologically sorted** one, because a parent node is processed before any of its child nodes is done.

Post-order, LRN

1. Recursively traverse the current node's left subtree.
2. Recursively traverse the current node's right subtree.
3. Visit the current node (in the figure: position blue).

Post-order traversal can be useful to get postfix expression of a **binary expression tree**.

In-order, LNR

1. Recursively traverse the current node's left subtree.
2. Visit the current node (in the figure: position green).
3. Recursively traverse the current node's right subtree.

- In a **binary search tree** ordered such that in each node the key is greater than all keys in its left subtree and less than all keys in its right subtree,
- in-order traversal retrieves the keys in *ascending* sorted order.[7]

Reverse pre-order, NRL

1. Visit the current node.
2. Recursively traverse the current node's right subtree.
3. Recursively traverse the current node's left subtree.

Reverse post-order, RLN

1. Recursively traverse the current node's right subtree.
2. Recursively traverse the current node's left subtree.
3. Visit the current node.

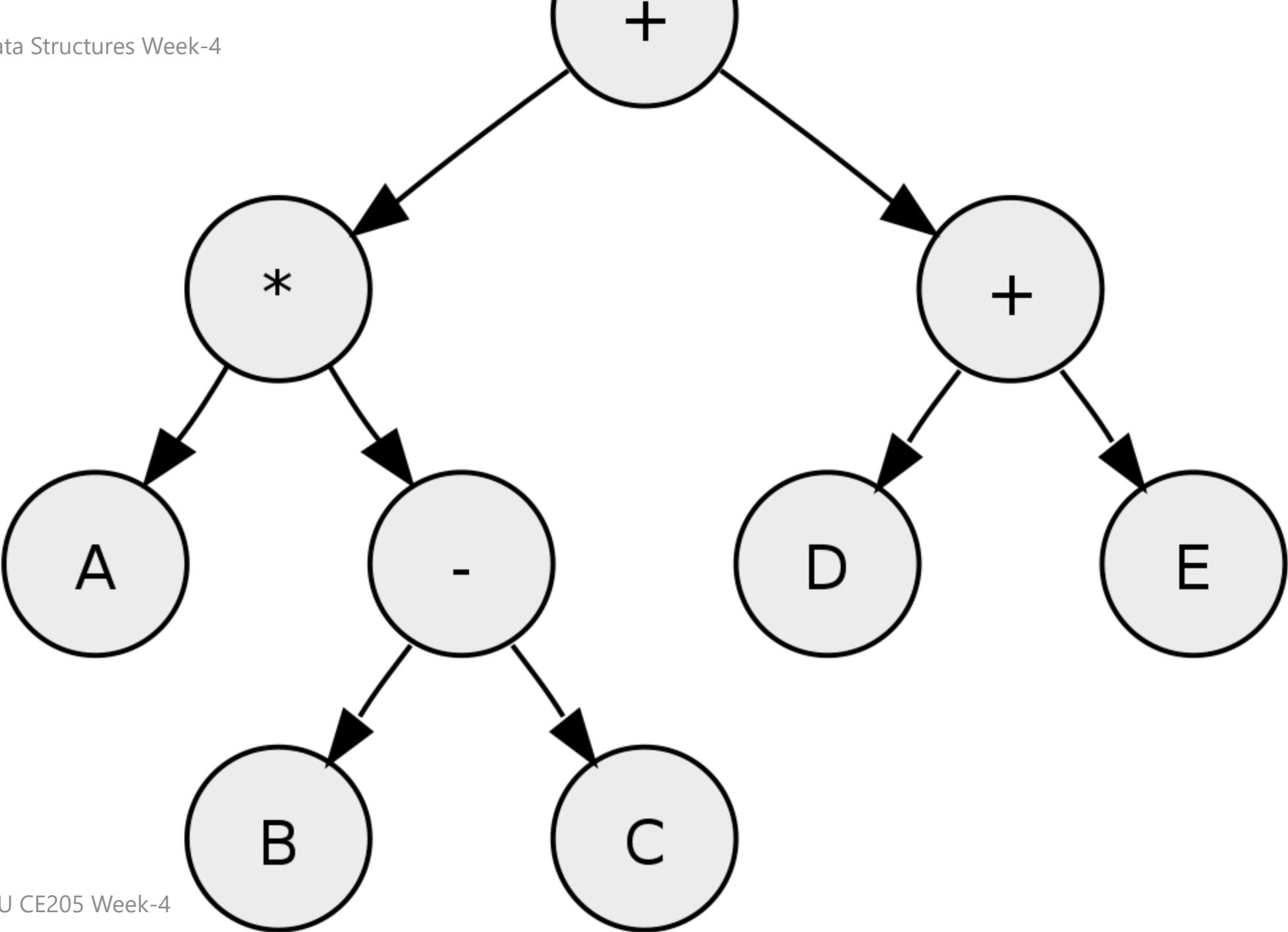
Reverse in-order, RNL

1. Recursively traverse the current node's right subtree.
2. Visit the current node.
3. Recursively traverse the current node's left subtree.

- In a **binary search tree** ordered such that in each node the key is greater than all keys in its left subtree and less than all keys in its right subtree,
- reverse in-order traversal retrieves the keys in *descending* sorted order.

Applications for Pre-Order

- Pre-order traversal can be used to make a prefix expression ([Polish notation](#)) from [expression trees](#): traverse the expression tree pre-orderly.
- For example, traversing the depicted arithmetic expression in pre-order yields "+ * A - B C + D E".
- In prefix notation, no need any parentheses as long as each operator has a fixed number of operands.
- Preorder traversal is also used to create a copy of the tree.



- Post-order traversal can generate a postfix representation ([Reverse Polish notation](#)) of a binary tree.
- Traversing the depicted arithmetic expression in post-order yields " $A B C - * D E + +$ "; the latter can easily be transformed into [machine code](#) to evaluate the expression by a [stack machine](#).
- Postorder traversal is also used to delete the tree.
- Each node is freed after freeing its children.

Pre-order implementation Recursive

```
procedure preorder(node)
  if node = null
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

Pre-order implementation Iterative

```
procedure iterativePreorder(node)
  if node = null
    return
  stack ← empty stack
  stack.push(node)
  while not stack.isEmpty()
    node ← stack.pop()
    visit(node)
    // right child is pushed first so that left is processed first
    if node.right ≠ null
      stack.push(node.right)
    if node.left ≠ null
      stack.push(node.left)
```

Post-order implementation Recursive

```
procedure postorder(node)
  if node = null
    return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```

Post-order implementation Iterative

```
procedure iterativePostorder(node)
  stack ← empty stack
  lastNodeVisited ← null
  while not stack.isEmpty() or node ≠ null
    if node ≠ null
      stack.push(node)
      node ← node.left
    else
      peekNode ← stack.peek()
      // if right child exists and traversing node
      // from left child, then move right
      if peekNode.right ≠ null and lastNodeVisited ≠ peekNode.right
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← stack.pop()
```

In-order implementation Recursive

```
procedure inorder(node)
  if node = null
    return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```

In-order implementation Iterative

```
procedure iterativeInorder(node)
  stack ← empty stack
  while not stack.isEmpty() or node ≠ null
    if node ≠ null
      stack.push(node)
      node ← node.left
    else
      node ← stack.pop()
      visit(node)
      node ← node.right
```

Binary Tree Traversal in C

```

// Tree traversal in C
#include <stdio.h>
#include <stdlib.h>

struct node {
    int item;
    struct node* left;
    struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ->", root->item);
    inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
    if (root == NULL) return;
    printf("%d ->", root->item);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(int value) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->item = value;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
    root->left = createNode(value);
    return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
    root->right = createNode(value);
    return root->right;
}

int main() {
    struct node* root = createNode(1);
    insertLeft(root, 12);
    insertRight(root, 9);

    insertLeft(root->left, 5);
    insertRight(root->left, 6);

    printf("Inorder traversal \n");
    inorderTraversal(root);

    printf("\nPreorder traversal \n");
    preorderTraversal(root);

    printf("\nPostorder traversal \n");
    postorderTraversal(root);
}

```


Binary Tree Traversal in C++

```

// Tree traversal in C++
#include <iostream>
using namespace std;

struct Node {
    int data;
    struct Node *left, *right;
    Node(int data) {
        this->data = data;
        left = right = NULL;
    }
};

// Preorder traversal
void preorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    cout << node->data << "->";
    preorderTraversal(node->left);
    preorderTraversal(node->right);
}

// Postorder traversal
void postorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    postorderTraversal(node->left);
    postorderTraversal(node->right);
    cout << node->data << "->";
}

// Inorder traversal
void inorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    inorderTraversal(node->left);
    cout << node->data << "->";
    inorderTraversal(node->right);
}

int main() {
    struct Node* root = new Node(1);
    root->left = new Node(12);
    root->right = new Node(9);
    root->left->left = new Node(5);
    root->left->right = new Node(6);

    cout << "Inorder traversal ";
    inorderTraversal(root);

    cout << "\nPreorder traversal ";
    preorderTraversal(root);

    cout << "\nPostorder traversal ";
    postorderTraversal(root);
}

```

Binary Tree Traversal in Java

```
// Tree traversal in Java

class Node {
    int item;
    Node left, right;

    public Node(int key) {
        item = key;
        left = right = null;
    }
}

class BinaryTree {
    // Root of Binary Tree
    Node root;

    BinaryTree() {
        root = null;
    }

    void postorder(Node node) {
        if (node == null)
            return;

        // Traverse left
        postorder(node.left);
        // Traverse right
        postorder(node.right);
        // Traverse root
        System.out.print(node.item + "->");
    }

    void inorder(Node node) {
        if (node == null)
            return;

        // Traverse left
        inorder(node.left);
        // Traverse root
        System.out.print(node.item + "->");
        // Traverse right
        inorder(node.right);
    }

    void preorder(Node node) {
        if (node == null)
            return;

        // Traverse root
        System.out.print(node.item + "->");
        // Traverse left
        preorder(node.left);
        // Traverse right
        preorder(node.right);
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(12);
        tree.root.right = new Node(9);
        tree.root.left.left = new Node(5);
        tree.root.left.right = new Node(6);

        System.out.println("Inorder traversal");
        tree.inorder(tree.root);

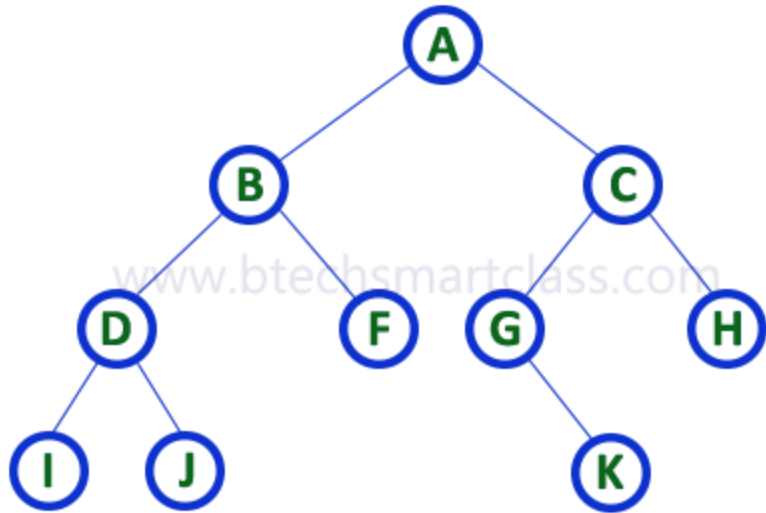
        System.out.println("\nPreorder traversal ");
        tree.preorder(tree.root);

        System.out.println("\nPostorder traversal");
        tree.postorder(tree.root);
    }
}
```

Review



Consider the following binary tree.



1. In - Order Traversal (leftChild - root - rightChild)

- In In-Order traversal,
- the root node is visited between the left child and right child.
- In this traversal,
- the left child node is visited first,
- then the root node is visited and
- later we go for visiting the right child node.
- This in-order traversal is applicable for every root node of all subtrees in the tree.
This is performed recursively for all nodes in the tree.

- In the above example of a binary tree,
- first we try to visit left child of root node 'A',
- but A's left child 'B' is a root node for left subtree.
- so we try to visit its (B's) left child 'D' and
- again D is a root for subtree with nodes D, I and J.
- So we try to visit its left child 'I' and it is the leftmost child.

- So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'.
- With this we have completed the left part of node B.
- Then visit 'B' and next B's right child 'F' is visited.
- With this we have completed left part of node A.
- Then visit root node 'A'. With this we have completed left and root parts of node A.

- Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G.
- But G does not have left part so we visit '**G**' and then visit G's right child K.
- With this we have completed the left part of node C.
- Then visit root node '**C**' and next visit C's right child '**H**' which is the rightmost child in the tree. So we stop the process.

- That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

- In Pre-Order traversal,
- the root node is visited before the left child and right child nodes.
- In this traversal,
- the root node is visited first,
- then its left child and
- later its right child.
- This pre-order traversal is applicable for every root node of all subtrees in the tree.

- In the above example of binary tree,
- first we visit root node 'A' then visit its left child 'B' which is a root for D and F.
- So we visit B's left child 'D' and again D is a root for I and J.
- So we visit D's left child 'I' which is the leftmost child.
- So next we go for visiting D's right child 'J'.
- With this we have completed root,

- left and right parts of node D and root,
- left parts of node B.
- Next visit B's right child 'F'.
- With this we have completed root and left parts of node A.
- So we go for A's right child 'C' which is a root node for G and H.

- After visiting C, we go for its left child '**G**' which is a root for node K.
- So next we visit left of G, but it does not have left child so we go for G's right child '**K**'. With this, we have completed node C's root and left parts.
- Next visit C's right child '**H**' which is the rightmost child in the tree. So we stop the process.

- That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.
- **Pre-Order Traversal for above example binary tree is**
- **A - B - D - I - J - F - C - G - K - H**

3. Post - Order Traversal (leftChild - rightChild - root)

- In Post-Order traversal,
- the root node is visited after left child and right child.
- In this traversal,
- left child node is visited first,
- then its right child and
- then its root node.
- This is recursively performed until the right most node is visited.

- Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.
- **Post-Order Traversal for above example binary tree is**
- **I - J - D - F - B - K - G - H - C - A**

Program to Create Binary Tree and display using In-Order Traversal - C Programming

```

#include<stdio.h>
#include<conio.h>

struct Node{
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *root = NULL;
int count = 0;

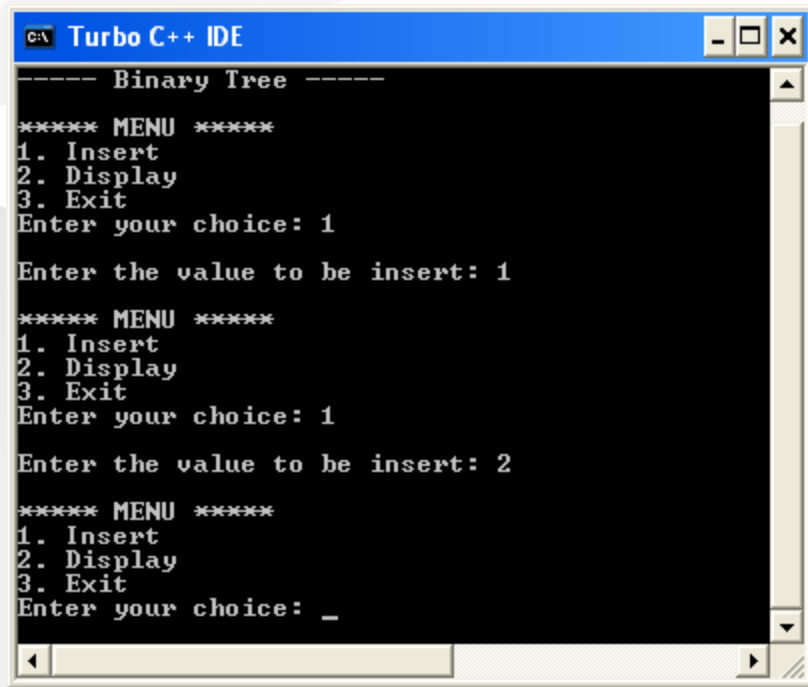
struct Node* insert(struct Node*, int);
void display(struct Node*);

void main(){
    int choice, value;
    clrscr();
    printf("\n---- Binary Tree ----\n");
    while(1){
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Display\n3. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("\nEnter the value to be insert: ");
                    scanf("%d", &value);
                    root = insert(root,value);
                    break;
            case 2: display(root); break;
            case 3: exit(0);
            default: printf("\nPlease select correct operations!!!\n");
        }
    }
}

struct Node* insert(struct Node *root,int value){
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(root == NULL){
        newNode->left = newNode->right = NULL;
        root = newNode;
        count++;
    }
    else{
        if(count%2 != 0)
            root->left = insert(root->left,value);
        else
            root->right = insert(root->right,value);
    }
    return root;
}
// display is performed by using Inorder Traversal
void display(struct Node *root)
{
    if(root != NULL){
        display(root->left);
        printf("%d\t",root->data);
        display(root->right);
    }
}

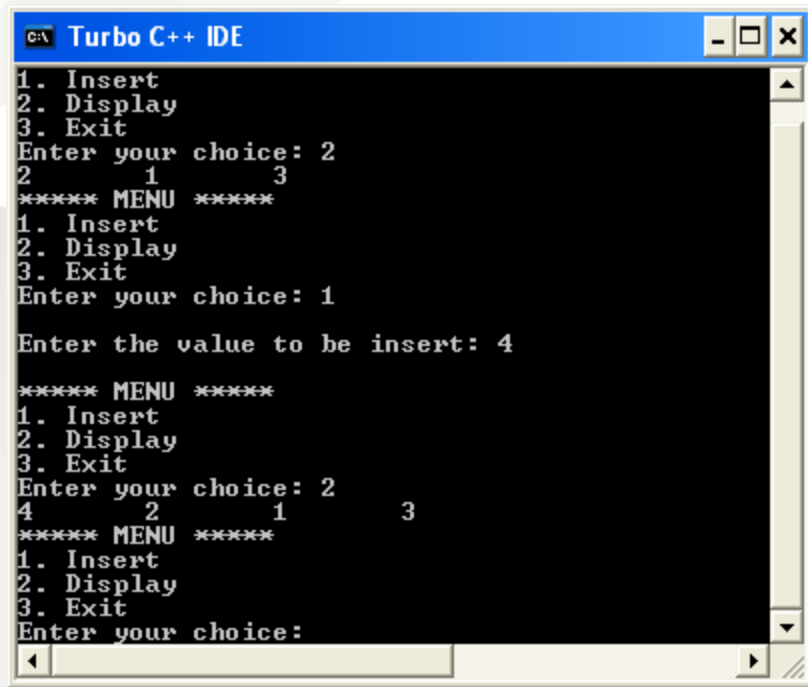
```

Output



```
C:\ Turbo C++ IDE
----- Binary Tree -----
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter the value to be insert: 1
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter the value to be insert: 2
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: _
```

Output



```
C:\ Turbo C++ IDE
1. Insert
2. Display
3. Exit
Enter your choice: 2
2 1 3
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter the value to be insert: 4
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 2
4 2 1 3
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice:
```

Threaded Binary Trees

- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/threaded-binary-trees.html

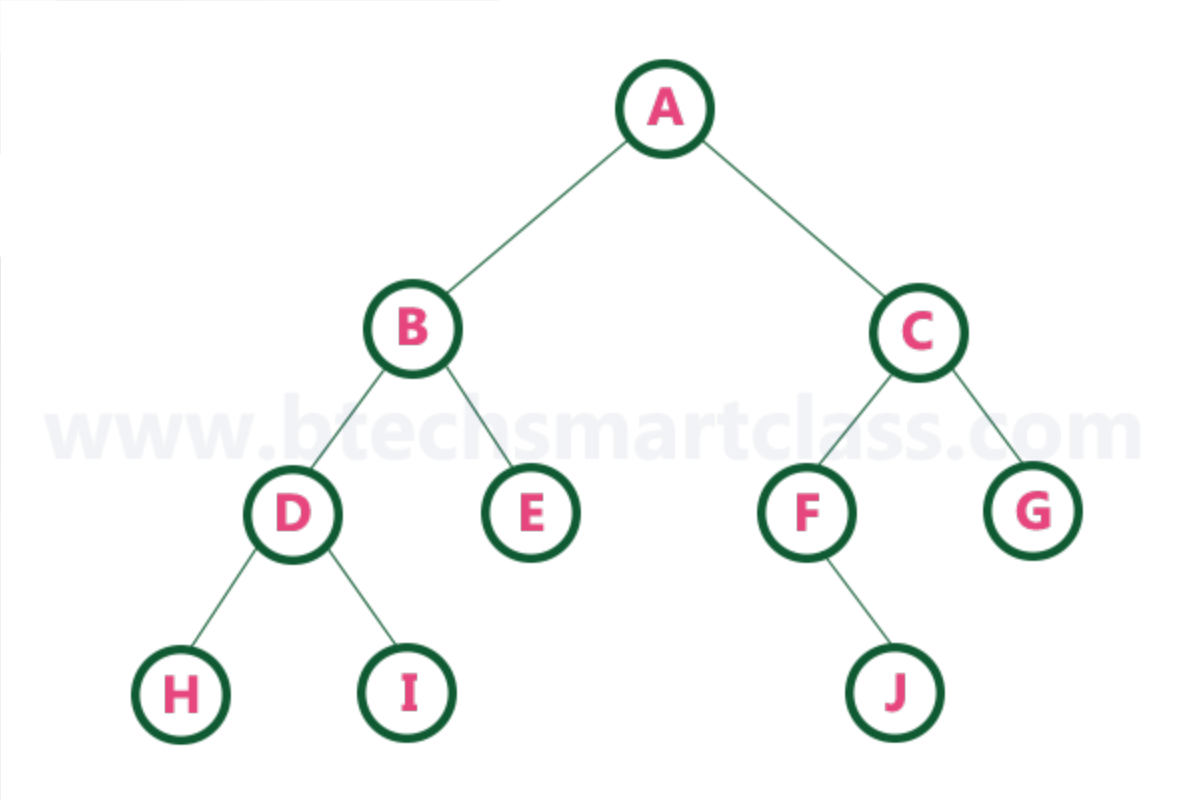
Threaded Binary Trees

- A binary tree can be represented using array representation or linked list representation.
- When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer.
- In any binary tree linked list representation, there is a number of NULL pointers than actual pointers.
- Generally, in any binary tree linked list representation, if there are $2N$ number of reference fields, then $N+1$ number of reference fields are filled with NULL ($N+1$ are NULL out of $2N$).
- This NULL pointer does not play any role except indicating that there is no link (no child).

- A. J. Perlis and C. Thornton have proposed new binary tree called "***Threaded Binary Tree***", which makes use of NULL pointers to improve its traversal process.
- In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as ***threads***.
- Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

- If there is no in-order predecessor or in-order successor, then it points to the root node.

Consider the following binary tree.



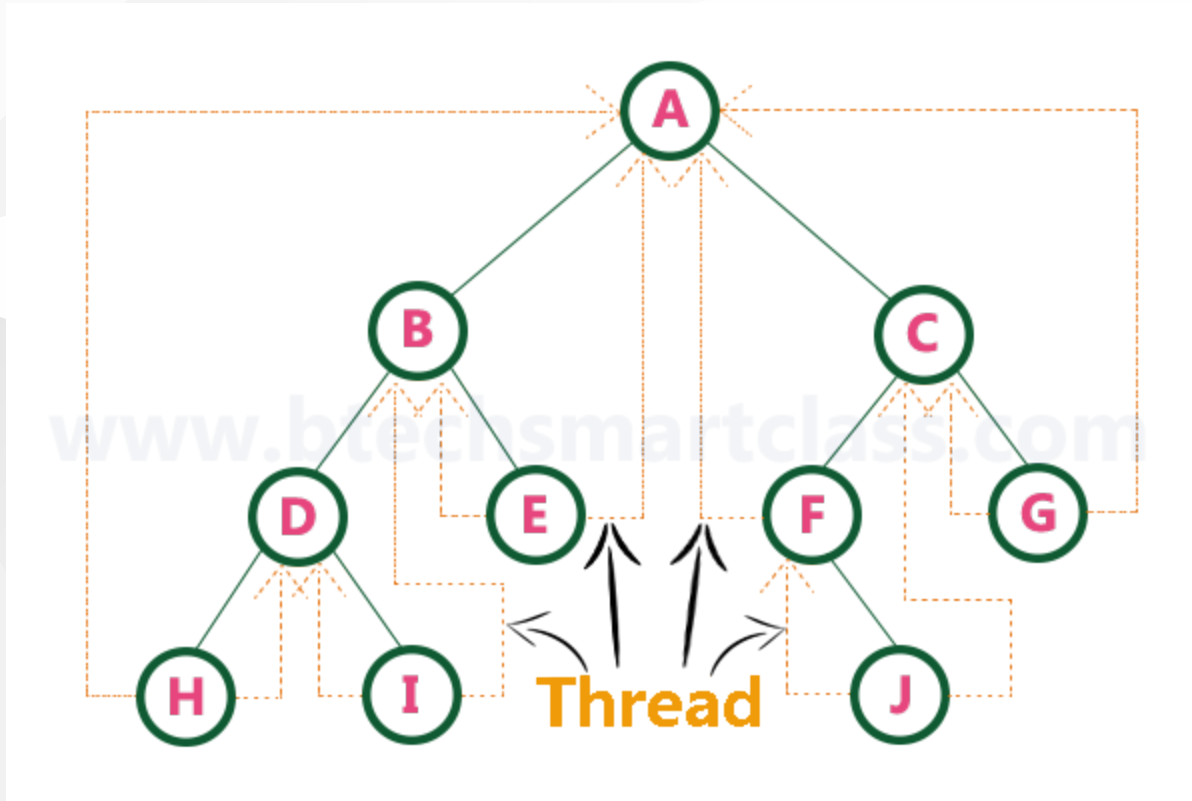
- To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

- When we represent the above binary tree using linked list representation, nodes **H**, **I**, **E**, **F**, **J** and **G** left child pointers are NULL.
- This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.
- And nodes **H**, **I**, **E**, **J** and **G** right child pointers are NULL.
- These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

- Above example binary tree is converted into threaded binary tree as follows.



- In the above figure, threads are indicated with dotted links.

Heaps (Max, Min, Binary , Binomial, Fibonacci, Leftist, K-ary) and Priority Queue

Heap Data Structure

- <https://ucoruh.github.io/ce100-algorithms-and-programming-II/week-4/ce100-week-4-heap/>
- Programiz
 - <https://www.programiz.com/dsa/heap-data-structure>

Max-Heap

- [Data Structures Tutorials - Max Heap with an exaple](#)
- [CE100 Algorithms and Programming II - RTEU CE100 Algorithms and Programming-II Course Notes](#)

Max Priority Queue

- Course Notes
 - [CE100 Algorithms and Programming II - RTEU CE100 Algorithms and Programming-II Course Notes](#)
- Btech Smart Class
 - http://www.btechsmartclass.com/data_structures/max-priority-queue.html
- William Fiset
 - https://www.youtube.com/watch?v=wptevk0bshY&t=0s&ab_channel=WilliamFiset
 - <https://github.com/williamfiset/Algorithms/tree/master/src/main/java/com/williamfiset/algorithms/datastructures/priorityqueue>

Max Priority Queue with Heap

- Please follow the link below for Heap and Max-Priority
 - [CE100 Algorithms and Programming II - RTEU CE100 Algorithms and Programming-II Course Notes](#)

Max Priority Queue in C

```

// Priority Queue Implementation in C
#include <stdio.h>
int size = 0;
void swap(int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}

// Function to heapify the tree
void heapify(int array[], int size, int i) {
    if (size == 1) {
        printf("Single element in the heap");
    } else {
        // 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 && array[l] > array[largest])
            largest = l;
        if (r < size && array[r] > array[largest])
            largest = r;

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

// Function to insert an element into the tree
void insert(int array[], int newNum) {
    if (size == 0) {
        array[0] = newNum;
        size += 1;
    } else {
        array[size] = newNum;
        size += 1;
        for (int i = size / 2 - 1; i >= 0; i--) {
            heapify(array, size, i);
        }
    }
}

// Function to delete an element from the tree
void deleteRoot(int array[], int num) {
    int i;
    for (i = 0; i < size; i++) {
        if (num == array[i])
            break;
    }

    swap(&array[i], &array[size - 1]);
    size -= 1;
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(array, size, i);
    }
}

// Print the array
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i)
        printf("%d ", array[i]);
    printf("\n");
}

// Driver code
int main() {
    int array[10];

    insert(array, 3);
    insert(array, 4);
    insert(array, 9);
    insert(array, 5);
    insert(array, 2);

    printf("Max-Heap array: ");
    printArray(array, size);

    deleteRoot(array, 4);

    printf("After deleting an element: ");
    printArray(array, size);
}

```

Max Priority Queue in C++

```

// Priority Queue implementation in C++
#include <iostream>
#include <vector>
using namespace std;

// Function to swap position of two elements
void swap(int *a, int *b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}

// Function to heapify the tree
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);
    }
}

// Function to insert an element into the tree
void insert(vector<int> &ht, int newNum) {
    int size = ht.size();
    if (size == 0) {
        ht.push_back(newNum);
    } else {
        ht.push_back(newNum);
        for (int i = size / 2 - 1; i >= 0; i--) {
            heapify(ht, i);
        }
    }
}

// Function to delete an element from the tree
void deleteNode(vector<int> &ht, int num) {
    int size = ht.size();
    int i;
    for (i = 0; i < size; i++) {
        if (num == ht[i])
            break;
    }
    swap(&ht[i], &ht[size - 1]);
    ht.pop_back();
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(ht, i);
    }
}

// Print the tree
void printArray(vector<int> &ht) {
    for (int i = 0; i < ht.size(); ++i)
        cout << ht[i] << " ";
    cout << "\n";
}

// Driver code
int main() {
    vector<int> heapTree;

    insert(heapTree, 3);
    insert(heapTree, 4);
    insert(heapTree, 9);
    insert(heapTree, 5);
    insert(heapTree, 2);

    cout << "Max-Heap array: ";
    printArray(heapTree);

    deleteNode(heapTree, 4);

    cout << "After deleting an element: ";
    printArray(heapTree);
}

```

Max Priority Queue in Java

```
// Priority Queue implementation in Java
import java.util.ArrayList;

class Heap {
    // Function to heapify the tree
    void heapify(ArrayList<Integer> 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.get(l) > ht.get(largest))
            largest = l;
        if (r < size && ht.get(r) > ht.get(largest))
            largest = r;

        // Swap and continue heapifying if root is not largest
        if (largest != i) {
            int temp = ht.get(largest);
            ht.set(largest, ht.get(i));
            ht.set(i, temp);
        }
        heapify(ht, largest);
    }

    // Function to insert an element into the tree
    void insert(ArrayList<Integer> ht, int newNum) {
        int size = ht.size();
        if (size == 0) {
            ht.add(newNum);
        } else {
            ht.add(newNum);
            for (int i = size / 2 - 1; i >= 0; i--) {
                heapify(ht, i);
            }
        }
    }

    // Function to delete an element from the tree
    void deleteNode(ArrayList<Integer> ht, int num) {
        int size = ht.size();
        int i;
        for (i = 0; i < size; i++) {
            if (num == ht.get(i))
                break;
        }

        int temp = ht.get(i);
        ht.set(i, ht.get(size - 1));
        ht.set(size - 1, temp);

        ht.remove(size - 1);
        for (int j = size / 2 - 1; j >= 0; j--) {
            heapify(ht, j);
        }
    }

    // Print the tree
    void printArray(ArrayList<Integer> array, int size) {
        for (Integer i : array) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    // Driver code
    public static void main(String args[]) {
        ArrayList<Integer> array = new ArrayList<Integer>();
        int size = array.size();

        Heap h = new Heap();
        h.insert(array, 3);
        h.insert(array, 4);
        h.insert(array, 0);
        h.insert(array, 5);
        h.insert(array, 2);

        System.out.println("Max-Heap array: ");
        h.printArray(array, size);

        h.deleteNode(array, 4);
        System.out.println("After deleting an element: ");
        h.printArray(array, size);
    }
}
```

Max Priority Queue with Array

- In the normal queue data structure,
- insertion is performed at the end of the queue and deletion is performed based on the FIFO principle.
- This queue implementation may not be suitable for all applications.
- Consider a networking application where the server has to respond for requests from multiple clients using queue data structure.

- Assume four requests arrived at the queue in the order of R1, R2, R3 & R4 where R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. A queue is as follows.



Now, check to wait time of each request that to be completed.

1. R1 : 20 units of time
2. R2 : 22 units of time (R2 must wait until R1 completes 20 units and R2 itself requires 2 units. Total 22 units)
3. R3 : 32 units of time (R3 must wait until R2 completes 22 units and R3 itself requires 10 units. Total 32 units)
4. R4 : 37 units of time (R4 must wait until R3 completes 35 units and R4 itself requires 5 units. Total 37 units)

****Here, the average waiting time for all requests (R1, R2, R3 and R4) is $(20+22+32+37)/4 \approx 27$ units of time.**

- That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.
- Now, consider another way of serving these requests.
- If we serve according to their required amount of time, first we serve R2 which has minimum time (2 units) requirement.
- Then serve R4 which has second minimum time (5 units) requirement and then serve R3 which has third minimum time (10 units) requirement and finally R1 is served which has maximum time (20 units) requirement.

Now, check to wait time of each request that to be completed.

1. R2 : 2 units of time
2. R4 : 7 units of time (R4 must wait until R2 completes 2 units and R4 itself requires 5 units. Total 7 units)
3. R3 : 17 units of time (R3 must wait until R4 completes 7 units and R3 itself requires 10 units. Total 17 units)
4. R1 : 37 units of time (R1 must wait until R3 completes 17 units and R1 itself requires 20 units. Total 37 units)

****Here, the average waiting time for all requests (R1, R2, R3 and R4) is $(2+7+17+37)/4 \approx 15$ units of time.**

- From the above two situations, it is very clear that the second method server can complete all four requests with very less time compared to the first method.
- This is what exactly done by the priority queue.
- Priority queue is a variant of a queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

- There are two types of priority queues they are as follows.
 - Max Priority Queue
 - Min Priority Queue

1. Max Priority Queue

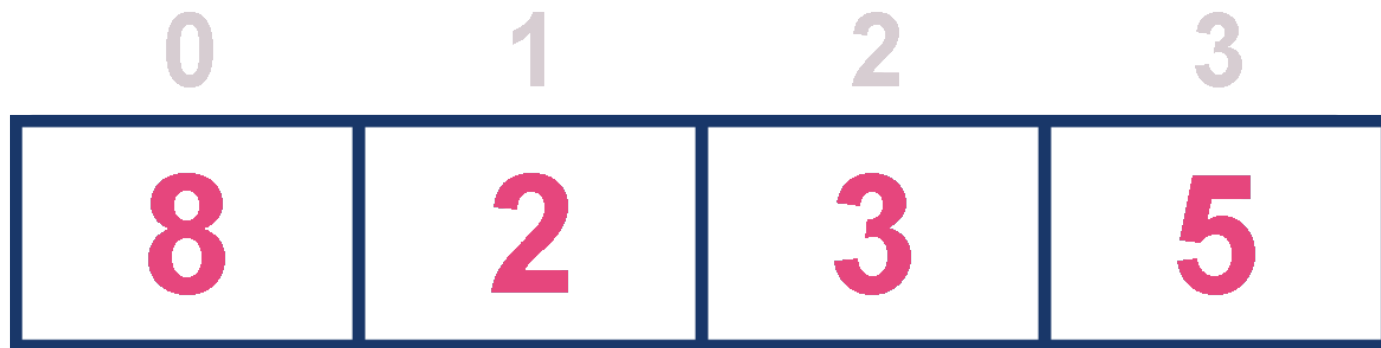
- In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue.
- For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.
- The following are the operations performed in a Max priority queue...
 - **isEmpty()** - Check whether queue is Empty.
 - **insert()** - Inserts a new value into the queue.
 - **findMax()** - Find maximum value in the queue.
 - **remove()** - Delete maximum value from the queue.

Max Priority Queue Representations

- There are 6 representations of max priority queue.
 - Using an Unordered Array (Dynamic Array)
 - Using an Unordered Array (Dynamic Array) with the index of the maximum value
 - Using an Array (Dynamic Array) in Decreasing Order
 - Using an Array (Dynamic Array) in Increasing Order
 - Using Linked List in Increasing Order
 - Using Unordered Linked List with reference to node with the maximum value

1. Using an Unordered Array (Dynamic Array)

- In this representation, elements are inserted according to their arrival order and the largest element is deleted first from the max priority queue.
- For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



- Now, let us analyze each operation according to this representation.
- **isEmpty()** - If 'front == -1' queue is Empty. This operation requires $O(1)$ time complexity which means constant time complexity.
- **insert()** - New element is added at the end of the queue. This operation requires $O(1)$ time complexity which means constant time complexity.
- **findMax()** - To find the maximum element in the queue, we need to compare it with all the elements in the queue. This operation requires $O(n)$ time complexity.
- **remove()** - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** which requires $O(n)$ time complexity, then that element is deleted with constant time complexity $O(1)$. The remove() operation requires $O(n) + O(1) \approx O(n)$ time complexity.

2. Using an Unordered Array (Dynamic Array) with the index of the maximum value

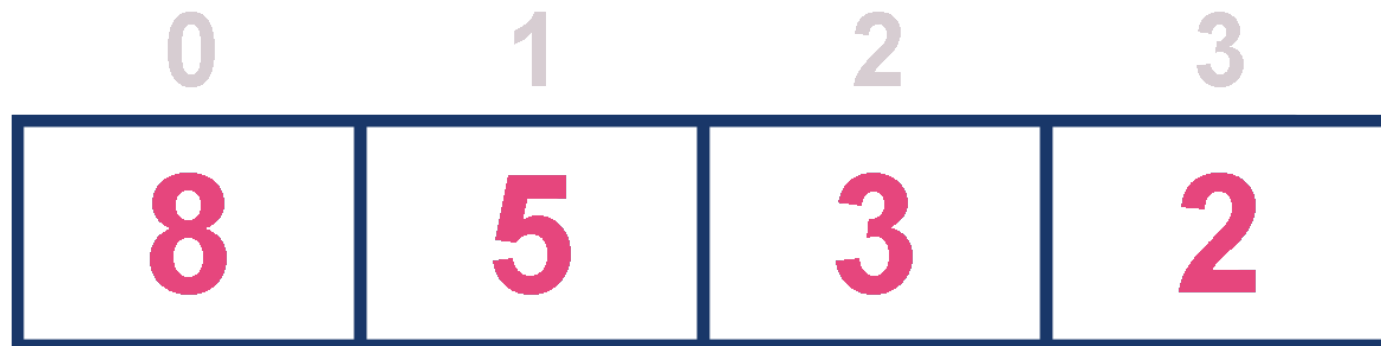
- In this representation, elements are inserted according to their arrival order and the largest element is deleted first from max priority queue.
- For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



- Now, let us analyze each operation according to this representation.
- **isEmpty()** - If 'front == -1' queue is Empty. This operation requires $O(1)$ time complexity which means constant time complexity.
- **insert()** - New element is added at the end of the queue with $O(1)$ time complexity and for each insertion we need to update maxIndex with $O(1)$ time complexity. This operation requires $O(1)$ time complexity which means constant time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because index of the maximum element is stored in maxIndex. This operation requires $O(1)$ time complexity.
- **remove()** - To remove an element from the queue, first we need to find the largest element using **findMax()** which requires $O(1)$ time complexity, then that element is deleted with constant time complexity $O(1)$ and finally we need to update the next largest element index value in maxIndex which requires $O(n)$ time complexity. The **remove()** operation requires $O(1)+O(1)+O(n) \approx O(n)$ time complexity.

3. Using an Array (Dynamic Array) in Decreasing Order

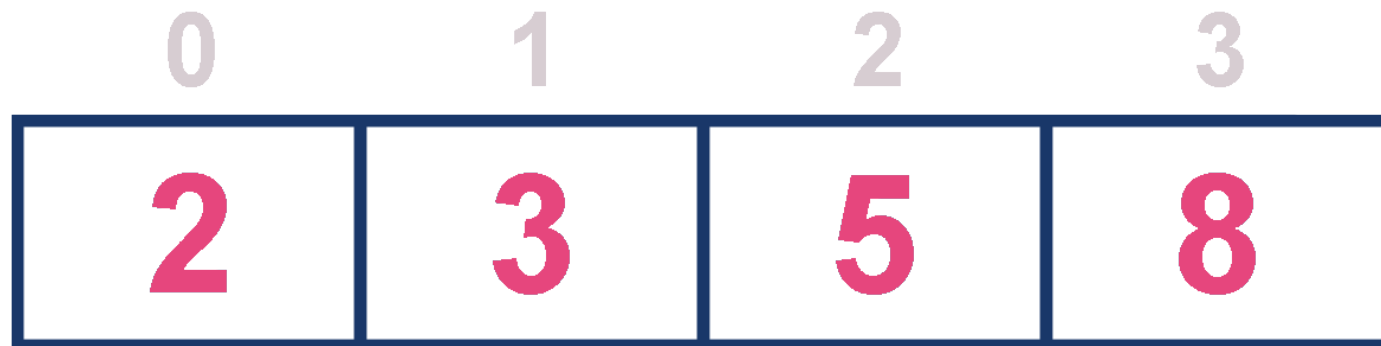
- In this representation, elements are inserted according to their value in decreasing order and largest element is deleted first from max priority queue.
- For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



- **isEmpty()** - If 'front == -1' queue is Empty. This operation requires $O(1)$ time complexity which means constant time complexity.
- **insert()** - New element is added at a particular position based on the decreasing order of elements which requires $O(n)$ time complexity as it needs to shift existing elements in order to insert new element in decreasing order. This insert() operation requires $O(n)$ time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the beginning of the queue. This findMax() operation requires $O(1)$ time complexity.
- **remove()** - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** operation which requires $O(1)$ time complexity, then that element is deleted with constant time complexity $O(1)$ and finally we need to rearrange the remaining elements in the list which requires $O(n)$ time complexity. This remove() operation requires $O(1) + O(1) + O(n) \approx O(n)$ time complexity.

4. Using an Array (Dynamic Array) in Increasing Order

- In this representation, elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.
- For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



- **isEmpty()** - If 'front == -1' queue is Empty. This operation requires $O(1)$ time complexity which means constant time complexity.
- **insert()** - New element is added at a particular position in the increasing order of elements into the queue which requires $O(n)$ time complexity as it needs to shift existing elements to maintain increasing order of elements. This insert() operation requires $O(n)$ time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the end of the queue. This findMax() operation requires $O(1)$ time complexity.
- **remove()** - To remove an element from the queue first we need to find the largest element using **findMax()** which requires $O(1)$ time complexity, then that element is deleted with constant time complexity $O(1)$. Finally, we need to rearrange the remaining elements to maintain increasing order of elements which requires $O(n)$ time complexity. This remove() operation requires $O(1) + O(1) + O(n) \approx O(n)$ time complexity.

5. Using Linked List in Increasing Order

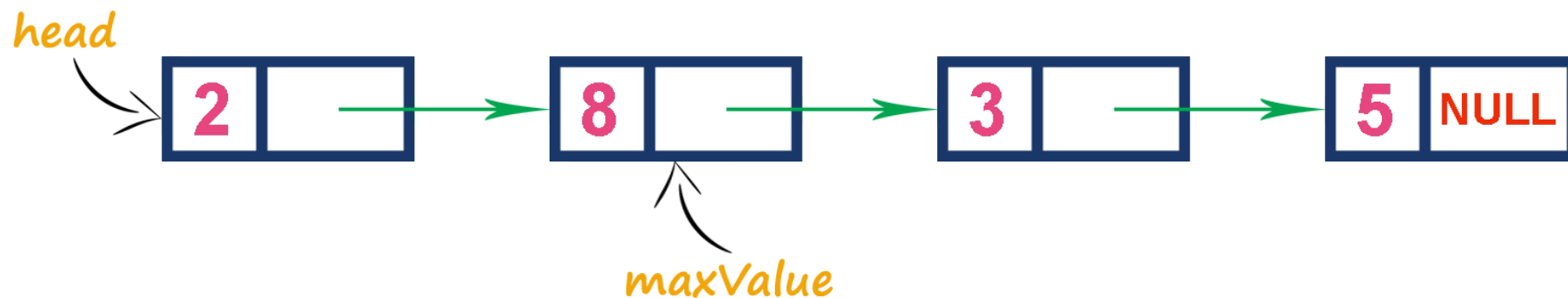
- In this representation, we use a single linked list to represent max priority queue. In this representation, elements are inserted according to their value in increasing order and a node with the maximum value is deleted first from the max priority queue.
- For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order of 8, 5, 3 and 2.



- Now, let us analyze each operation according to this representation...
- **isEmpty()** - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.
- **insert()** - New element is added at a particular position in the increasing order of elements which requires **O(n)** time complexity. This insert() operation requires **O(n)** time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.
- **remove()** - Removing an element from the queue is simple because the largest element is last node in the queue. This remove() operation requires **O(1)** time complexity.

6. Using Unordered Linked List with reference to node with the maximum value

- In this representation, we use a single linked list to represent max priority queue. We always maintain a reference (`maxValue`) to the node with the maximum value in the queue. In this representation, elements are inserted according to their arrival and the node with the maximum value is deleted first from the max priority queue.
- For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order of 8, 5, 3 and 2.



- Now let us analyze each operation according to this representation..
- **isEmpty()** - If 'head == NULL' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.
- **insert()** - New element is added at end of the queue which requires **O(1)** time complexity. And we need to update max**Value** reference with address of largest element in the queue which requires **O(1)** time complexity. This insert() operation requires **O(1)** time complexity.
- **findMax()** - Finding the maximum element in the queue is very simple because the address of largest element is stored at max**Value**. This findMax() operation requires **O(1)** time complexity.
- **remove()** - Removing an element from the queue is deleting the node which is referenced by max**Value** which requires **O(1)** time complexity. And then we need to update max**Value** reference to new node with maximum value in the queue which requires **O(n)** time complexity. This remove() operation requires **O(n)** time complexity.

2. Min Priority Queue Representations

- Min Priority Queue is similar to max priority queue except for the removal of maximum element first. We remove minimum element first in the min-priority queue.
- The following operations are performed in Min Priority Queue...
 - **isEmpty()** - Check whether queue is Empty.
 - **insert()** - Inserts a new value into the queue.
 - **findMin()** - Find minimum value in the queue.
 - **remove()** - Delete minimum value from the queue.

Binary Heap

- Geeks for Geeks
 - Binary Heap
 - <https://www.geeksforgeeks.org/binary-heap/?ref=lbp>
 - <https://www.geeksforgeeks.org/difference-between-binary-heap-binomial-heap-and-fibonacci-heap/?ref=rp>

1. Structure of Fibonacci Heaps
2. Mergeable-heap operations

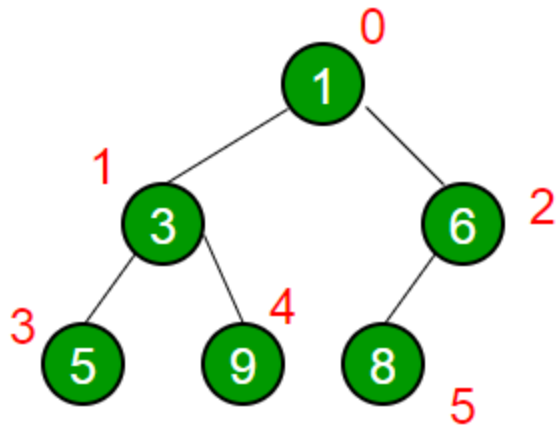
A Binary Heap is a Binary Tree with following properties.

1. It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

- The root element will be at $Arr[0]$.
- Below table shows indexes of other nodes for the i th node, i.e., $Arr[i]$:

$Arr[(i-1)/2]$	Returns the parent node
$Arr[(2*i)+1]$	Returns the left child node
$Arr[(2*i)+2]$	Returns the right child node

The traversal method use to achieve Array representation is **Level Order**



Applications of Heaps:

- 1) **Heap Sort**: Heap Sort uses Binary Heap to sort an array in $O(n \log n)$ time.
- 2) **Priority Queue**: Priority queues can be efficiently implemented using Binary Heap because it supports `insert()`, `delete()` and `extractmax()`, `decreaseKey()` operations in $O(\log n)$ time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
- 3) **Graph Algorithms**: The priority queues are especially used in Graph Algorithms like **Dijkstra's Shortest Path** and **Prim's Minimum Spanning Tree**.
- 4) Many problems can be efficiently solved using Heaps. See following for example.
 - a) **K'th Largest Element in an array**.
 - b) **Sort an almost sorted array/**
 - c) **Merge K Sorted Arrays**.

1) `getMini()`: It returns the root element of Min Heap. Time Complexity of this operation is $O(1)$.

2) `extractMin()`: Removes the minimum element from MinHeap. Time Complexity of this Operation is $O(\text{Log}n)$ as this operation needs to maintain the heap property (by calling `heapify()`) after removing root.

3) `decreaseKey()`: Decreases value of key. The time complexity of this operation is $O(\text{Log}n)$. If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) `insert()`: Inserting a new key takes $O(\text{Log}n)$ time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) `delete()`: Deleting a key also takes $O(\text{Log}n)$ time. We replace the key to be deleted with minus infinite by calling `decreaseKey()`. After `decreaseKey()`, the minus infinite

Basic Heap operations in C++

```

// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<limits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *a, int *b);

// A class for min heap
class MinHeap {
int heap; // pointer to array of elements in heap
int capacity; // maximum possible size of min heap
int heap_size; // Current number of elements in min heap
public:
// Constructor
MinHeap(int capacity);
// to heapify a subtree with the root at given index
void MinHeapify(int i);
int parent(int i) { return (i-1)/2; }
// to get index of left child of node at index i
int left(int i) { return (2*i + 1); }
// to get index of right child of node at index i
int right(int i) { return (2*i + 2); }
// to extract the root which is the minimum element
int extractMin();
// Decreases key value of key at index i to new_val
void decreaseKey(int i, int new_val);
// returns the minimum key (or root) from min heap
int getMin() { return heap[0]; }
// Deletes a key stored at index i
void deleteKey(int i);
// Inserts a new key 'k'
void insertKey(int k);
};

// Decreases key value of a heap from a given array a[] of given size
MinHeap::MinHeap(int cap) {
heap_size = 0;
capacity = cap;
heap = new int[cap];
// Inserts a new key 'k'
void MinHeap::insertKey(int k) {
if (heap_size == capacity) {
cout << "Overflow: Could not insertKey\n";
return;
}
// First insert the new key at the end
heap[heap_size] = k;
heap_size++;
// Fix the min heap property if it is violated
while (i > 0 && heap[parent(i)] > heap[i]) {
swap(heap[i], heap[parent(i)]);
i = parent(i);
}
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than heap[i]
void MinHeap::decreaseKey(int i, int new_val) {
heap[i] = new_val;
while (i > 0 && heap[parent(i)] > heap[i]) {
swap(heap[i], heap[parent(i)]);
i = parent(i);
}
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin() {
if (heap_size == 0)
return INT_MAX;
if (heap_size == 1) {
heap_size--;
return heap[0];
}
// Store the minimum value, and remove it from heap
int root = heap[0];
heap[0] = heap[heap_size-1];
heap_size--;
MinHeapify(0);
return root;
}

// This function deletes key at index i. It first reduced value to minus
// infinity, then calls extractMin()
void MinHeap::deleteKey(int i) {
decreaseKey(i, INT_MIN);
extractMin();
}

// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i) {
int l = left(i);
int r = right(i);
int smallest = i;
if (l < heap_size && heap[l] < heap[i])
smallest = l;
if (r < heap_size && heap[r] < heap[smallest])
smallest = r;
if (smallest != i) {
swap(heap[i], heap[smallest]);
MinHeapify(smallest);
}
}

// A utility function to swap two elements
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}

// Driver program to test above functions
int main() {
MinHeap h(11);
h.insertKey(12);
h.insertKey(1);
h.insertKey(9);
h.insertKey(5);
h.insertKey(3);
h.insertKey(7);
h.insertKey(4);
cout << h.extractMin() << "\n";
cout << h.getMin() << "\n";
h.deleteKey(1);
cout << h.getMin();
return 0;
}

```


Basic Heap operations in Java (Built-in Functions)

PriorityQueue in Java - GeeksforGeeks

```
// Java program to demonstrate the
// working of PriorityQueue
import java.util.*;

class PriorityQueueDemo {

    // Main Method
    public static void main(String args[])
    {
        // Creating empty priority queue
        PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();

        // Adding items to the pQueue using add()
        pQueue.add(10);
        pQueue.add(20);
        pQueue.add(15);

        // Printing the top element of PriorityQueue
        System.out.println(pQueue.peek());

        // Printing the top element and removing it
        // from the PriorityQueue container
        System.out.println(pQueue.poll());

        // Printing the top element again
        System.out.println(pQueue.peek());
    }
}
```

Output

```
10  
10  
15
```

K-Arr Heap

- Geeks for Geeks
 - <https://www.geeksforgeeks.org/k-ary-heap/?ref=lbp>

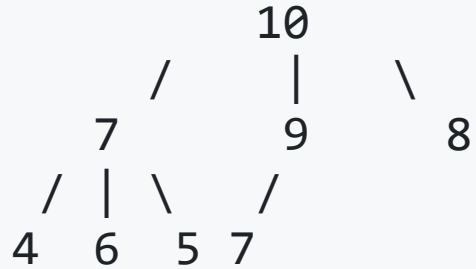
K-Arr Heap

K-ary heaps are a generalization of binary heap($K=2$) in which each node have K children instead of 2. Just like binary heap, it follows two properties:

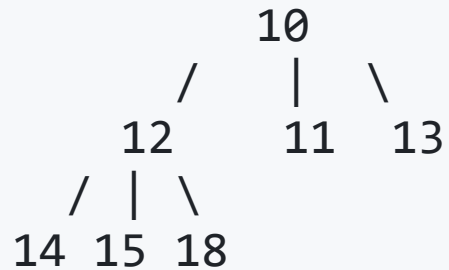
1. Nearly complete binary tree, with all levels having maximum number of nodes except the last, which is filled in left to right manner.
2. Like Binary Heap, it can be divided into two categories:
 - i. Max k-ary heap (key at root is greater than all descendants and same is recursively true for all nodes).
 - ii. Min k-ary heap (key at root is lesser than all descendants and same is recursively true for all nodes)

K-Arr Heap Example

3-ary max heap - root node is maximum
of all nodes



3-ary min heap - root node is minimum
of all nodes



The height of a complete k-ary tree with n-nodes is given by $\log_k n$.

Applications of K-ary Heap:

- K-ary heap when used in the implementation of **priority queue** allows faster decrease key operation as compared to binary heap ($O(\log_2 n)$) for binary heap vs $O(\log_k n)$ for K-ary heap).
- Nevertheless, it causes the complexity of extractMin() operation to increase to $O(k \log_k n)$ as compared to the complexity of $O(\log_2 n)$ when using binary heaps for priority queue.
- This allows K-ary heap to be more efficient in algorithms where decrease priority operations are more common than extractMin() operation.
- Example: **Dijkstra's** algorithm for single source shortest path and **Prim's** algorithm for minimum spanning tree
- K-ary heap has better memory cache behaviour than a binary heap which allows them to run more quickly in practice, although it has a larger worst case running time of both extractMin() and delete() operation (both being $O(k \log_k n)$).

Implementation:

Assuming 0 based indexing of array, an array represents a K-ary heap such that for any node we consider:

- Parent of the node at index i (except root node) is located at index $(i - 1)/k$
- Children of the node at index i are at indices $(k * i) + 1, (k * i) + 2 \dots (k * i) + k$
- The last non-leaf node of a heap of size n is located at index $(n - 2)/k$

buildHeap() : Builds a heap from an input array.

This function runs a loop starting from the last non-leaf node all the way upto the root node, calling a function `restoreDown`(also known as `maxHeapify`) for each index that restores the passed index at the correct position of the heap by shifting the node down in the K-ary heap building it in a bottom up manner.

Why do we start the loop from the last non-leaf node ?

Because all the nodes after that are leaf nodes which will trivially satisfy the heap property as they don't have any children and hence, are already roots of a K-ary max heap.

restoreDown() (or maxHeapify) : Used to maintain heap property.

It runs a loop where it finds the maximum of all the node's children, compares it with its own value and swaps if the $\max(\text{value of all children}) > (\text{value at node})$. It repeats this step until the node is restored into its original position in the heap.

extractMax() : Extracting the root node.

A k-ary max heap stores the largest element in its root. It returns the root node, copies last node to the first, calls restore down on the first node thus maintaining the heap property.

insert() : Inserting a node into the heap

This can be achieved by inserting the node at the last position and calling `restoreUp()` on the given index to restore the node at its proper position in the heap. `restoreUp()` iteratively compares a given node with its parent, since in a max heap the parent is always greater than or equal to its children nodes, the node is swapped with its parent only when its key is greater than the parent.

K-Arr Heap in C++

```

// C++ program to demonstrate all operations of
// a heap
#include<bits/stdc++.h>
using namespace std;

// Function to modify (or restore the max-heap
// property). This is used to build a k-ary heap
// and its extraction
// arr[] - Array that stores heap
// size - Size of array
// k - Value of k. If k is equal to be restored
// for heapify()
void restoreHeap(int arr[], int size, int k)
{
    // child array to store indexes of all
    // the children of given node
    int child[2];

    while (1)
    {
        // child[0]-1 if the node is a leaf
        // children the children
        for (int i=0; i<k; i++)
            child[i] = (2*(index + 1) + i) - 1;

        // max_child stores the maximum child and
        // its index
        int max_child = -1, max_child_index;

        // Loop to find the maximum of all
        // the children of a given node
        for (int i=0; i<k; i++)
        {
            if (child[i] != -1 &&
                arr[child[i]] > max_child)
            {
                max_child = child[i];
                max_child_index = i;
            }
        }

        // Leaf node
        if (max_child == -1)
            break;

        // Swap only if the key of max_child_index
        // is greater than the key of node
        if (arr[max_child_index] > arr[index])
            swap(arr[max_child_index], arr[index]);

        index = max_child_index;
    }
}

// Restore a given node up in the heap. This is used
// in decreaseKey() and insert()
void restoreNode(int arr[], int index, int k)
{
    // parent stores the index of the parent variable
    // of the node
    int parent = (index-1)/k;

    // Loop should only run till root node in case the
    // node is not the root node
    while (parent > 0)
    {
        if (arr[index] > arr[parent])
        {
            swap(arr[index], arr[parent]);
            index = parent;
            parent = (index-1)/k;
        }

        // Node has been restored at the correct position
        break;
    }
}

// Function to build a heap of arr[0..n-1] and value of k.
void buildHeap(int arr[], int n, int k)
{
    // Heapify all internal nodes starting from last
    // non-leaf node all the way upto the root node
    // and calling restore heap on each
    for (int i = (n-1)/k; i >= 0; i--)
        restoreNode(arr, i, k);
}

// Function to insert a value in a heap. Parameters are
// the array, size of heap, value k and the element to
// be inserted arr[], int n, int k, int elem
{
    // Put the new element in the last position
    arr[n] = elem;

    // Increase heap size by 1
    n = n+1;

    // Call restoreNode on the last index
    restoreNode(arr, n, k);
}

// Function that returns the key of root node of
// the heap and then restores the heap property
// of the remaining nodes
int extractMax(int arr[], int n, int k)
{
    // Stores the key of root node to be returned
    int max = arr[0];

    // Copy the last node's key to the root node
    arr[0] = arr[n-1];

    // Decrease heap size by 1
    n = n-1;

    // Call restoreNode on the root node to restore
    // it to the correct position in the heap
    restoreNode(arr, 0, k);

    return max;
}

// Driver program
int main()
{
    const int capacity = 100;
    int arr(capacity) = {4, 5, 6, 7, 8, 9, 10};
    int n = 7;

    buildHeap(arr, n, 3);

    printf("Build Heap : %d", n);
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    int element = 5;
    insert(arr, n, element);

    printf("Heap after insertion of %d: %d",
        element);
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    printf("Extracted max is %d",
        extractMax(arr, n, 3));

    printf("Heap after extract max: %d", n);
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

Output

```
Built Heap :  
10 9 6 7 8 4 5
```

```
Heap after insertion of 3:  
10 9 6 7 8 4 5 3
```

```
Extracted max is 10
```

```
Heap after extract max:  
9 8 6 7 3 4 5
```

K-Arr Heap Time Complexity Analysis

- For a k-ary heap, with n nodes the maximum height of the given heap will be $\log_k n$. So `restoreUp()` run for maximum of $\log_k n$ times (as at every iteration the node is shifted one level up in case of `restoreUp()` or one level down in case of `restoreDown()`).
- `restoreDown()` calls itself recursively for k children. So time complexity of this function is $O(k \log_k n)$.
- Insert and `decreaseKey()` operations call `restoreUp()` once. So complexity is $O(\log_k n)$.
- Since `extractMax()` calls `restoreDown()` once, its complexity $O(k \log_k n)$
- Time complexity of build heap is $O(n)$ (Analysis is similar to binary heap)

Leftist Heap

- Geeks for Geeks
 - <https://www.geeksforgeeks.org/leftist-tree-leftist-heap/>
 - <https://www.geeksforgeeks.org/leftist-tree-leftist-heap/?ref=lbp>
- Toronto
 - [CSC378: Leftist Trees](#)

Leftist Tree / Leftist Heap

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a **complete binary tree**), a leftist tree may be very unbalanced. Below are **time complexities** of **Leftist Tree / Heap**.

Leftist Tree / Leftist Heap

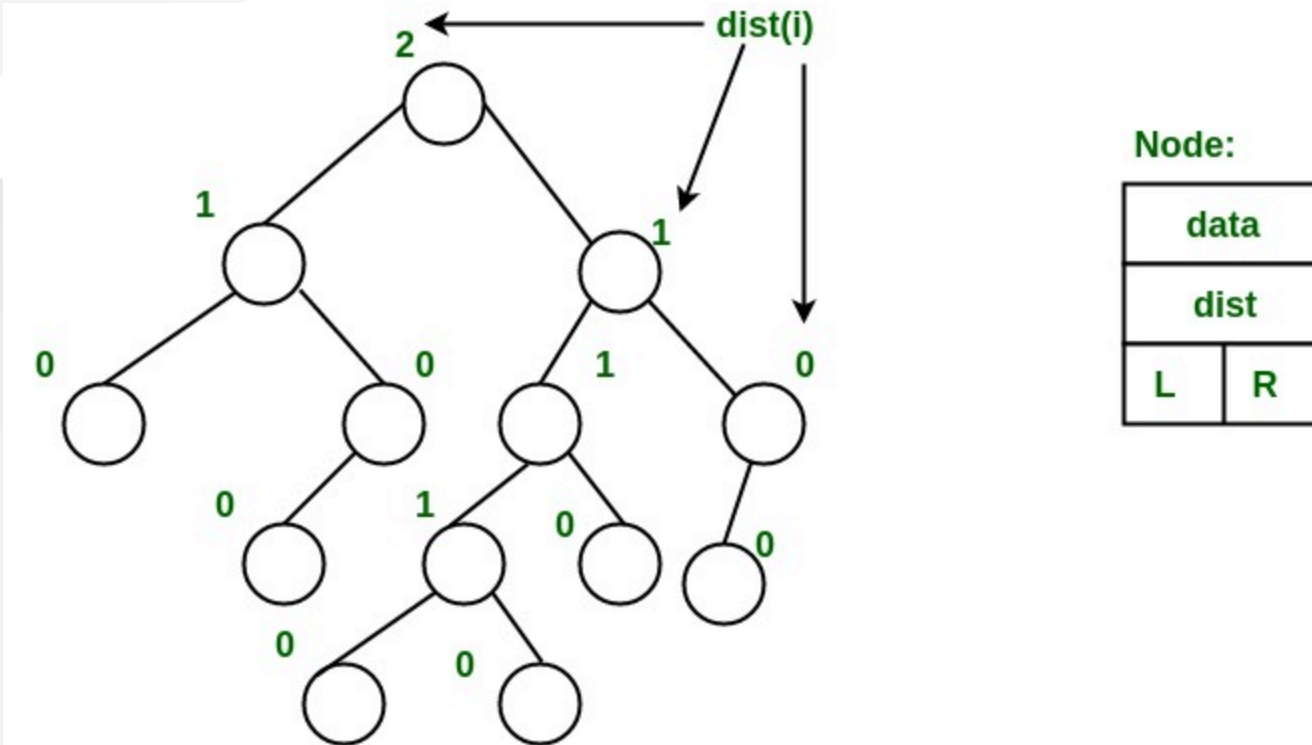
Function	Complexity	Comparison
Get Min	$O(1)$	same as both Binary and Binomial
Delete Min	$O(\log n)$	same as both Binary and Binomial
Insert	$O(\log n)$	$O(\log n)$ in Binary and $O(1)$ in Binomial and $O(\log n)$ for worst case
Merge	$O(\log n)$	$O(\log n)$ in Binomial

Leftist Tree / Leftist Heap

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property** : $\text{key}(i) \geq \text{key}(\text{parent}(i))$
2. **Heavier on left side** : $\text{dist}(\text{right}(i)) \leq \text{dist}(\text{left}(i))$. Here, $\text{dist}(i)$ is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$.

- The rightmost node has a rank of 0 as the right subtree of this node is null and
- its parent has a distance of 1 by $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$.
- The same is followed for each node and their s-value(or rank) is calculated.



From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.

Leftist Tree / Leftist Heap Operations

1. The main operation is merge().
2. deleteMin() (or extractMin()) can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done by create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

Leftist Tree / Leftist Heap Operations

Idea behind Merging : Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

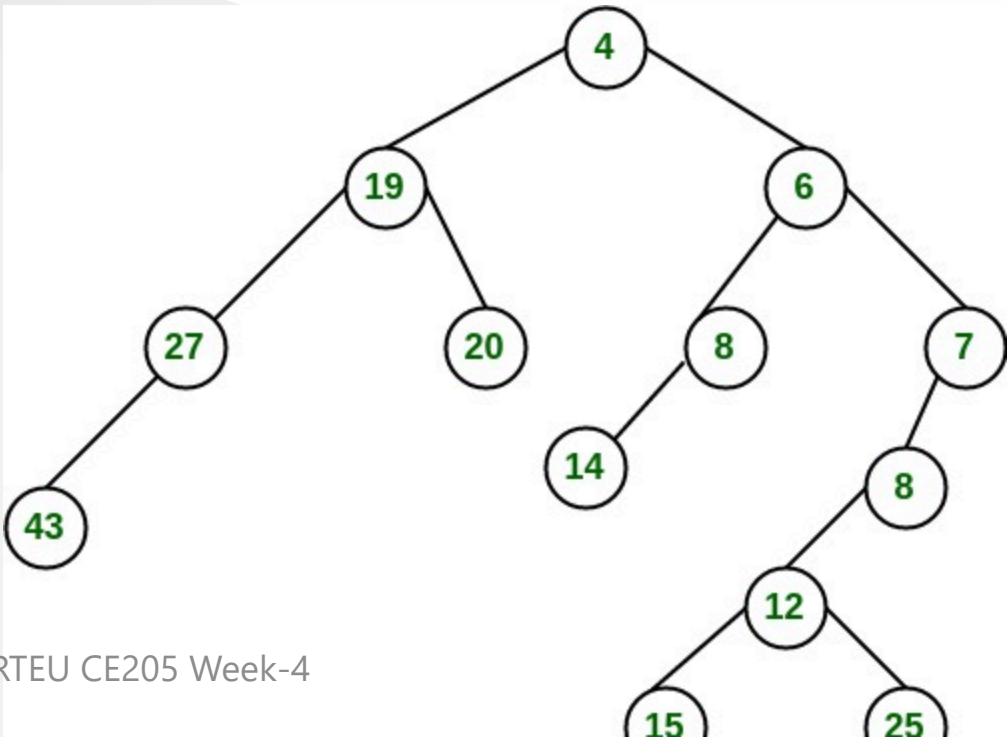
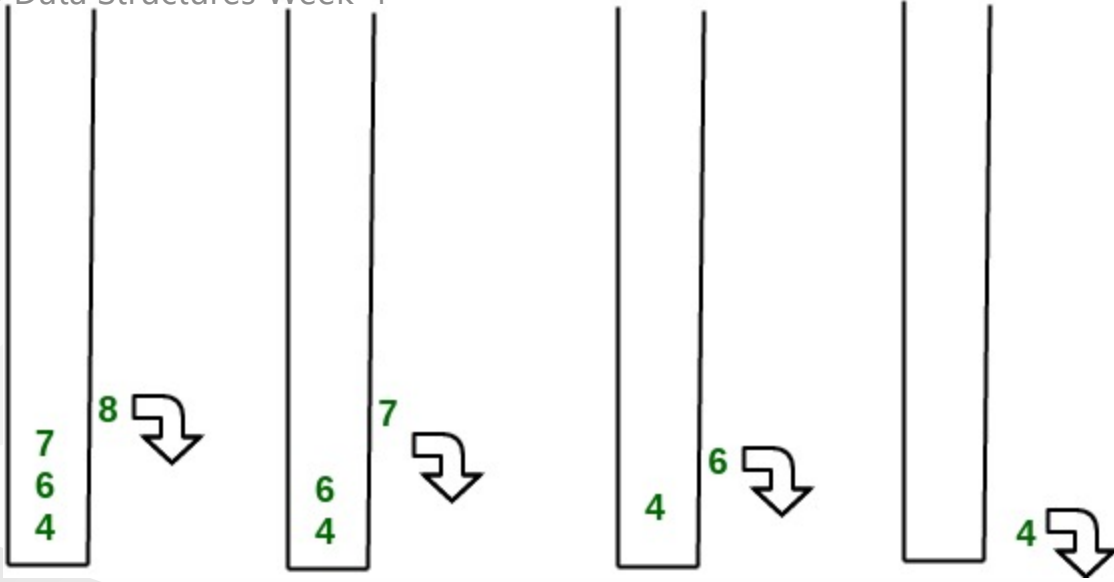
1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion: – Update `dist()` of merged root. – Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

Leftist Tree / Leftist Heap Operations

Detailed Steps for Merge:

1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Convert to leftist heap. Repeat the process



Output

```
1  
22  
5
```

Binomial Heap

- Geeks for Geeks
 - <https://www.geeksforgeeks.org/binomial-heap-2/>

The main application of **Binary Heap** is as implement a priority queue. Binomial Heap is an extension of **Binary Heap** that provides faster union or merge operation with other operations provided by Binary Heap.

A Binomial Heap is a collection of Binomial Trees

What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order $k-1$ and making one the leftmost child or the other.

A Binomial Tree of order k the has following properties.

- It has exactly 2^k nodes.
- It has depth as k .
- There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \dots, k$.
- The root has degree k and children of the root are themselves Binomial Trees with order $k-1, k-2, \dots, 0$ from left to right.

$k = 0$ (Single Node)

```
o
```

$k = 1$ (2 nodes)

[We take two $k = 0$ order Binomial Trees, and make one as a child of other]

```
o
 /
o
```

$k = 2$ (4 nodes)

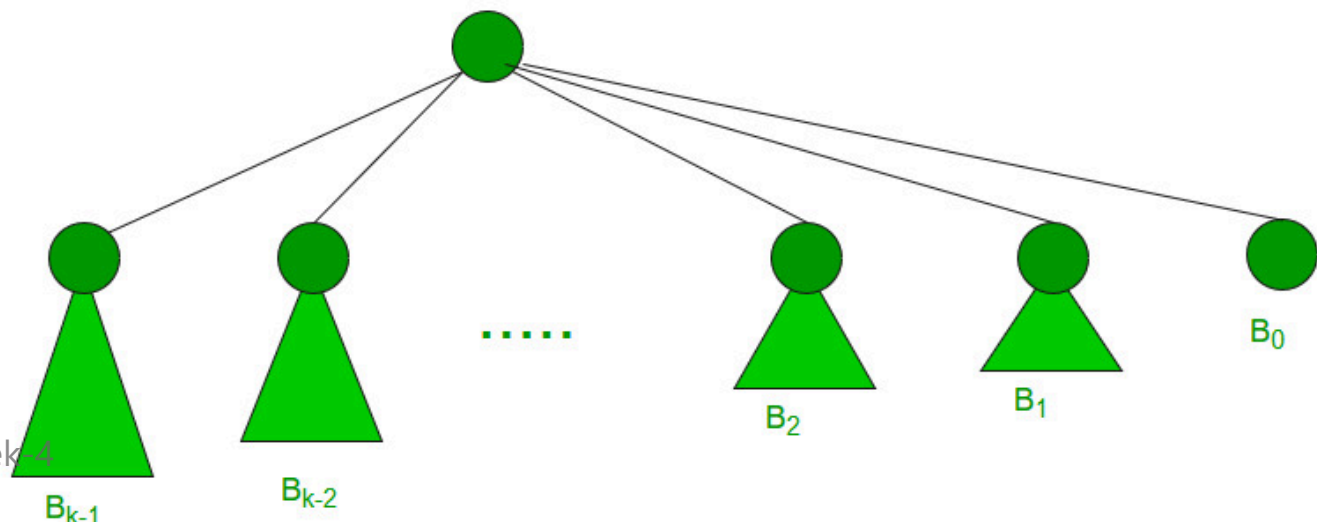
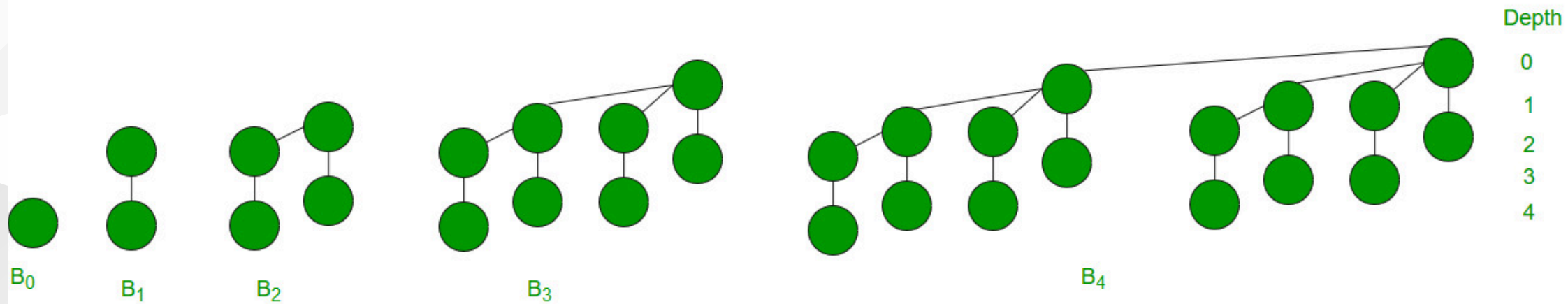
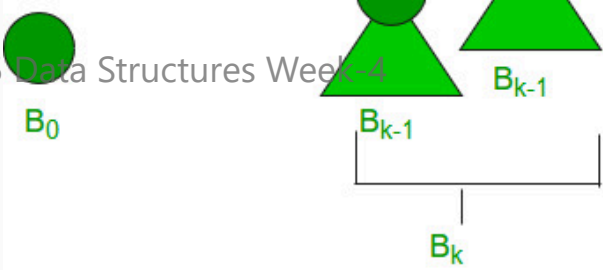
[We take two $k = 1$ order Binomial Trees, and make one as a child of other]

```
o
 / \
o   o
 /
o
```

$k = 3$ (8 nodes)

[We take two $k = 2$ order Binomial Trees, and make one as a child of other]

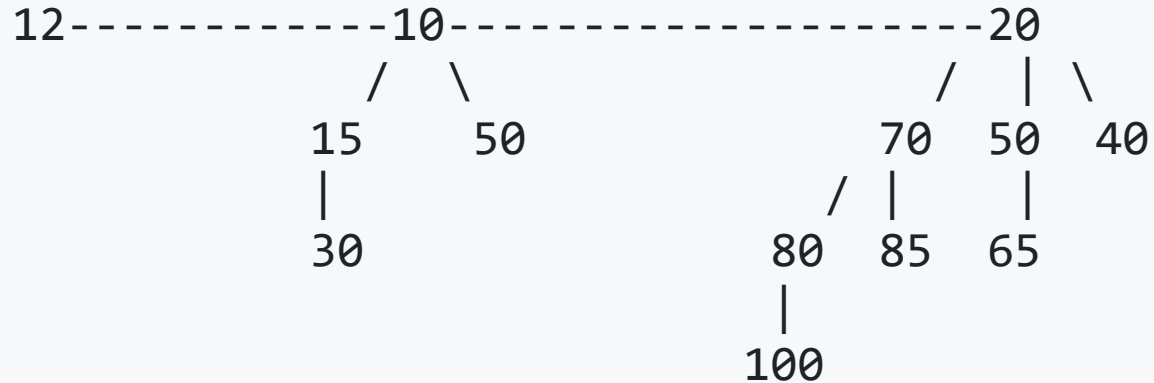
```
o
 / | \
o  o o
 / \ |
o  o o
     \
     o
```



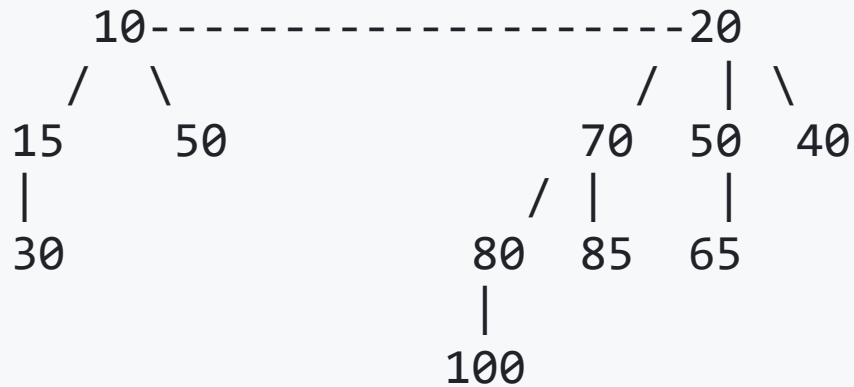
Binomial Heap:

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows the Min Heap property. And there can be at most one Binomial Tree of any degree.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2, and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has the number of Binomial Trees equal to the number of set bits in the binary representation of n . For example, let n be 13, there are 3 set bits in the binary representation of n (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are $O(\text{Log}n)$ Binomial Trees in a Binomial Heap with ' n ' nodes.

Union operation in Binomial Heap:

Given two Binomial Heaps H1 and H2, $\text{union}(H1, H2)$ creates a single Binomial Heap.

6. The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.
7. After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev , x , and next-x . There can be the following 4 cases when we traverse the list of roots.
 - Case 1: Orders of x and next-x are not the same, we simply move ahead.
 - In the following 3 cases, orders of x and next-x are the same.
 - Case 2: If the order of next-next-x is also the same, move ahead.
 - Case 3: If the key of x is smaller than or equal to the key of next-x , then make next-x a child of x by linking it with x .
 - Case 4: If the key of x is greater, then make x the child of next .

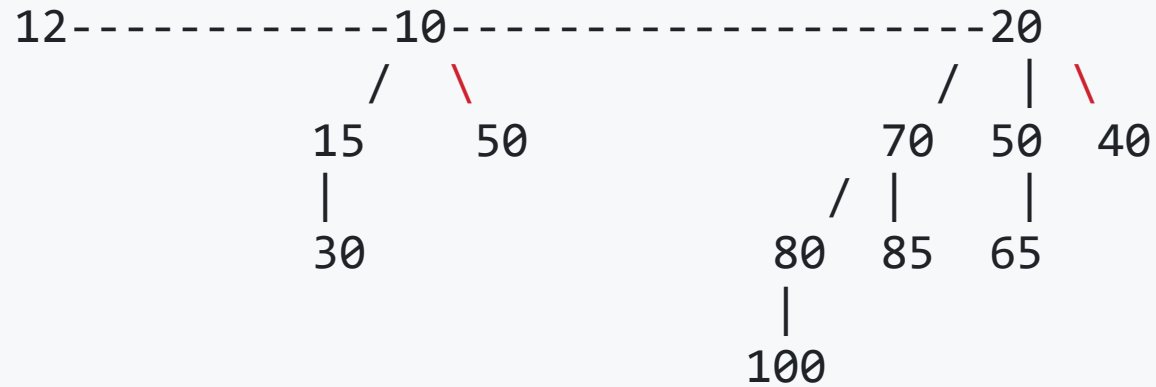
Time Complexity Analysis:

Operations	Binary Heap	Binomial Heap	Fibonacci Heap
Procedure	Worst-case	Worst-case	Amortized
Making Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Inserting a node	$\Theta(\log(n))$	$O(\log(n))$	$\Theta(1)$
Finding Minimum key	$\Theta(1)$	$O(\log(n))$	$O(1)$
Extract-Minimum key	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$
Union or merging	$\Theta(n)$	$O(\log(n))$	$\Theta(1)$
Decreasing a Key	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(1)$
Deleting a node	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$

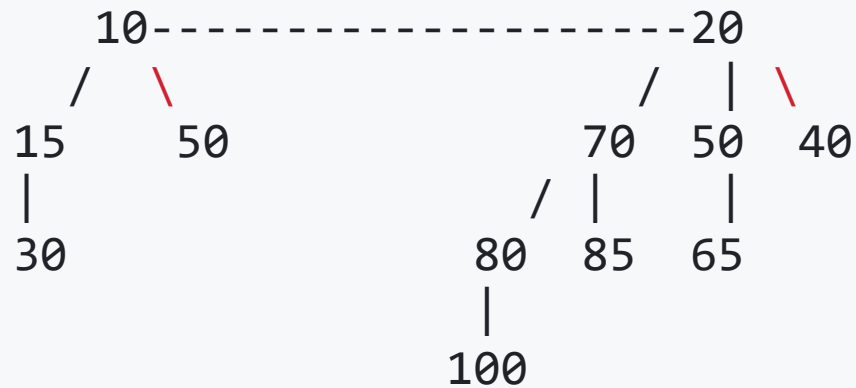
How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in and extracting() and delete()). The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and the other to the right sibling.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.



1. **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
2. **getMin(H):** A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires $O(\text{Log}n)$ time. It can be optimized to $O(1)$ by maintaining a pointer to minimum key root.
3. **extractMin(H):** This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires $O(\text{Log}n)$ time.

1. **delete(H)**: Like Binary Heap, delete operation first reduces the key to minus infinite, then calls `extractMin()`.
2. **decreaseKey(H)**: `decreaseKey()` is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of `decreaseKey()` is $O(\text{Log}n)$

Output

```
The heap is:  
50 10 30 40 20  
After deleting 10, the heap is:  
20 30 40 50
```

Fibonacci Heap

- William Fiset
 - [Algorithms/src/main/java/com/williamfiset/algorithms/datastructures/fibonacciheap at master · williamfiset/Algorithms · GitHub](https://github.com/williamfiset/Algorithms/tree/master/src/main/java/com/williamfiset/algorithms/datastructures/fibonacciheap)
- Geeks for Geeks
 - [Fibonacci Heap | Set 1 \(Introduction\) - GeeksforGeeks](#)

Fibonacci Heap

- <https://www.geeksforgeeks.org/fibonacci-heap-insertion-and-union/?ref=lbp>
 - <https://www.geeksforgeeks.org/fibonacci-heap-deletion-extract-min-and-decrease-key/?ref=lbp>
1. Decreasing a key and deleting a node
 2. Bounding the maximum degree

<https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf>

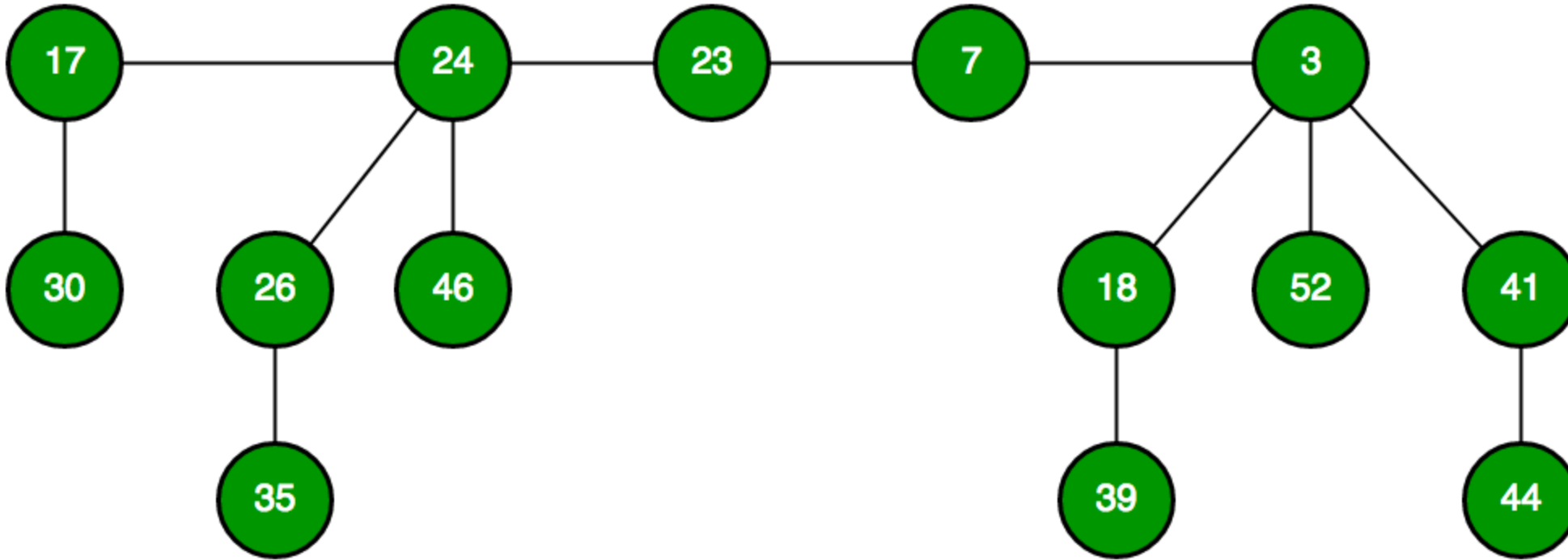
Fibonacci Heap

In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are **amortized time complexities** of Fibonacci Heap.

1) Find Min:	$\Theta(1)$	[Same as both Binary and Binomial]
2) Delete Min:	$O(\log n)$	[$\Theta(\log n)$ in both Binary and Binomial]
3) Insert:	$\Theta(1)$	[$\Theta(\log n)$ in Binary and $\Theta(1)$ in Binomial]
4) Decrease-Key:	$\Theta(1)$	[$\Theta(\log n)$ in both Binary and Binomial]
5) Merge:	$\Theta(1)$	[$\Theta(m \log n)$ or $\Theta(m+n)$ in Binary and $\Theta(\log n)$ in Binomial]

Like [Binomial Heap](#), Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of

Below are some interesting facts about Fibonacci Heap

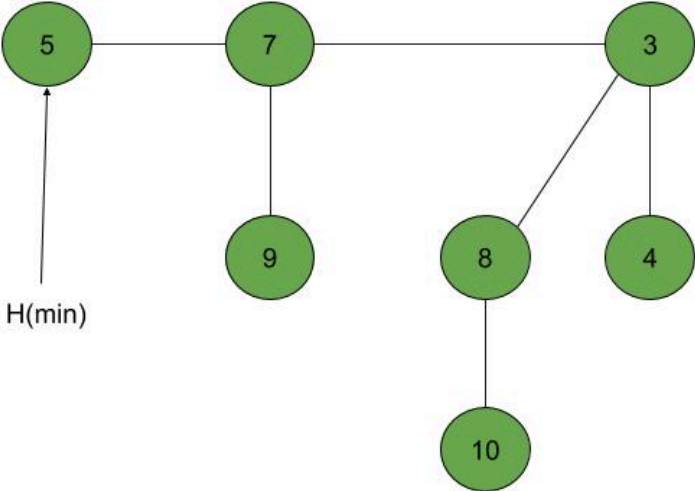
1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is $O(V\log V + E\log V)$. If Fibonacci Heap is used, then time complexity is improved to $O(V\log V + E)$
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source [Wiki](#)).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2} , where F_k is the k th Fibonacci number.

Fibonacci Heap – Insertion and Union

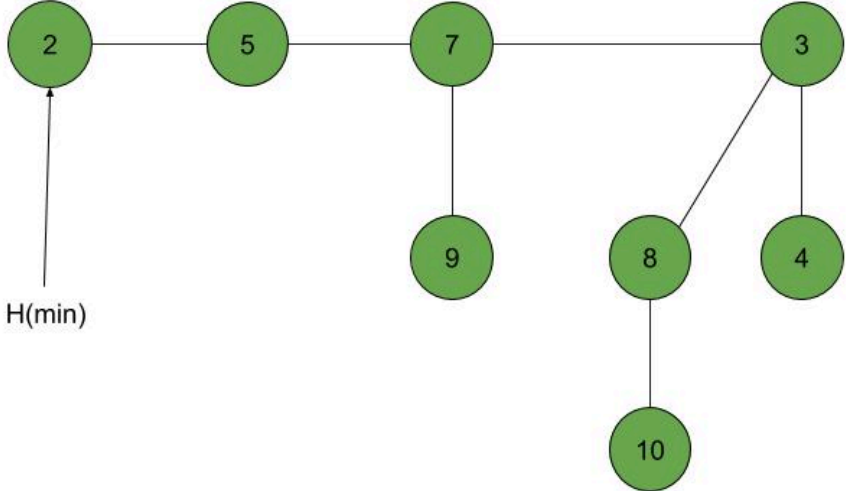
Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree). In this article, we will discuss Insertion and Union operation on Fibonacci Heap.

Insertion: To insert a node in a Fibonacci heap H , the following algorithm is followed:

```
Create a new node 'x'.
Check whether heap H is empty or not.
If H is empty then:
    Make x as the only node in the root list.
    Set H(min) pointer to x.
Else:
    Insert x into root list and update H(min).
```



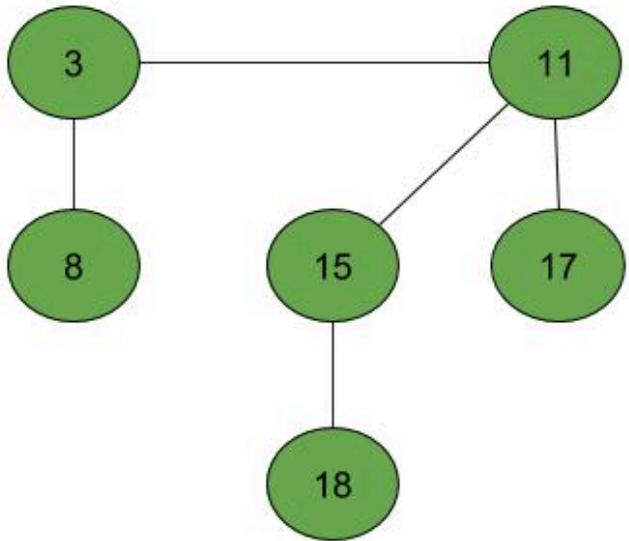
After inserting (2),



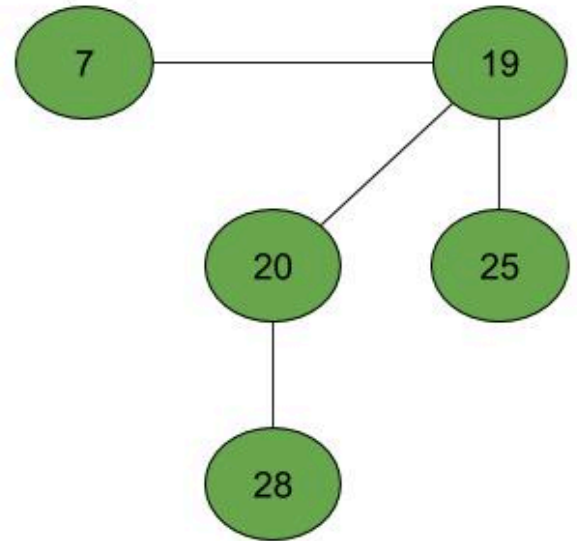
Union: Union of two Fibonacci heaps H1 and H2 can be accomplished as follows:

```
Join root lists of Fibonacci heaps H1 and H2 and make a single Fibonacci heap H.  
If  $H1(\min) < H2(\min)$  then:  
     $H(\min) = H1(\min)$ .  
Else:  
     $H(\min) = H2(\min)$ .
```

H1.(min)

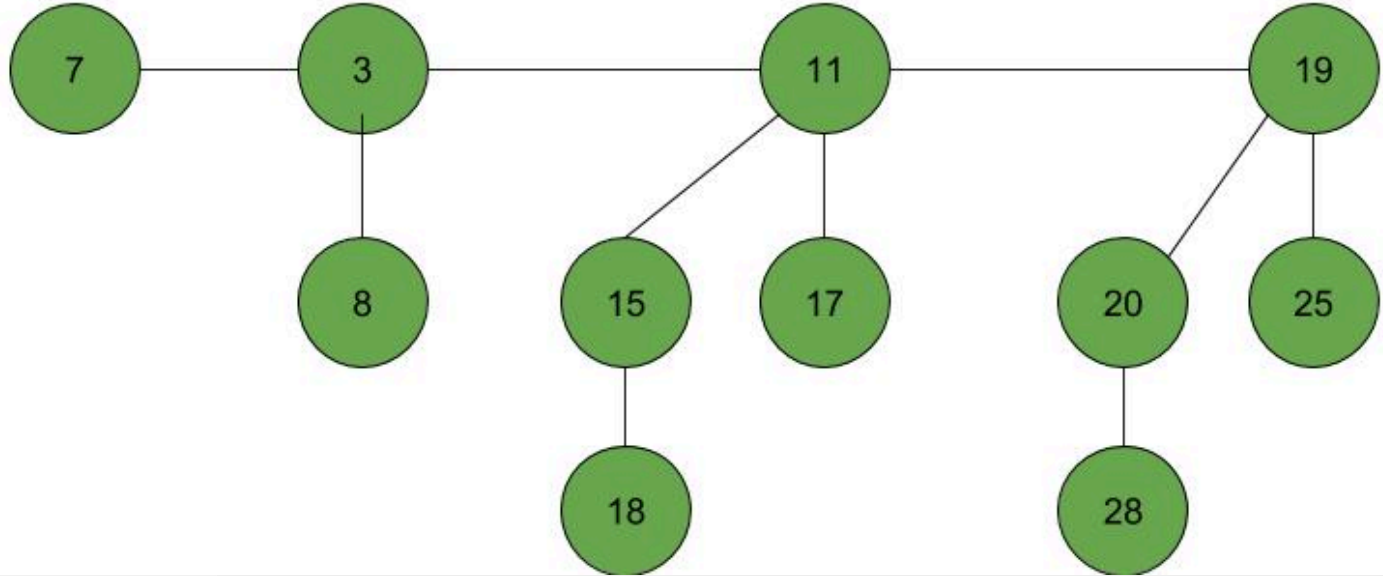


H2.(min)



union

H.(min)



Fibonacci Heap – Insertion and Union in C++

```

// C++ program to demonstrate building
// and inserting in a Fibonacci heap
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

struct node {
    node* parent;
    node* child;
    node* left;
    node* right;
    int key;
};

// Creating min pointer as "mini"
struct node* mini = NULL;

// Declare an integer for number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in heap
void insertion(int val)
{
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->key = val;
    new_node->parent = NULL;
    new_node->child = NULL;
    new_node->left = NULL;
    new_node->right = NULL;
    if (mini != NULL) {
        (mini->left)->right = new_node;
        new_node->right = mini;
        new_node->left = mini->left;
        mini->left = new_node;
        if (new_node->key < mini->key)
            mini = new_node;
    }
    else {
        mini = new_node;
    }
}

// Function to display the heap
void display(struct node* mini)
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << "The Heap is Empty" << endl;

    else {
        cout << "The root nodes of Heap are: " << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << "-->";
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl;
        cout << "The heap has " << no_of_nodes << " nodes" << endl;
    }
}

// Function to find min node in the heap
void find_min(struct node* mini)
{
    cout << "min of heap is: " << mini->key << endl;
}

// Driver code
int main()
{
    no_of_nodes = 7;
    insertion(4);
    insertion(3);
    insertion(7);
    insertion(5);
    insertion(2);
    insertion(1);
    insertion(10);

    display(mini);

    find_min(mini);

    return 0;
}

```

Fibonacci Heap – Deletion, Extract min and Decrease key

Extract_min():

We create a function **for** deleting the minimum node and setting the min pointer to the minimum value in the remaining heap. The following algorithm is followed:

Delete the min node.

Set head to the next min node and add all the trees of the deleted node in the root **list**.

Create an **array** of degree pointers of the size of the deleted node.

Set degree pointer to the current node.

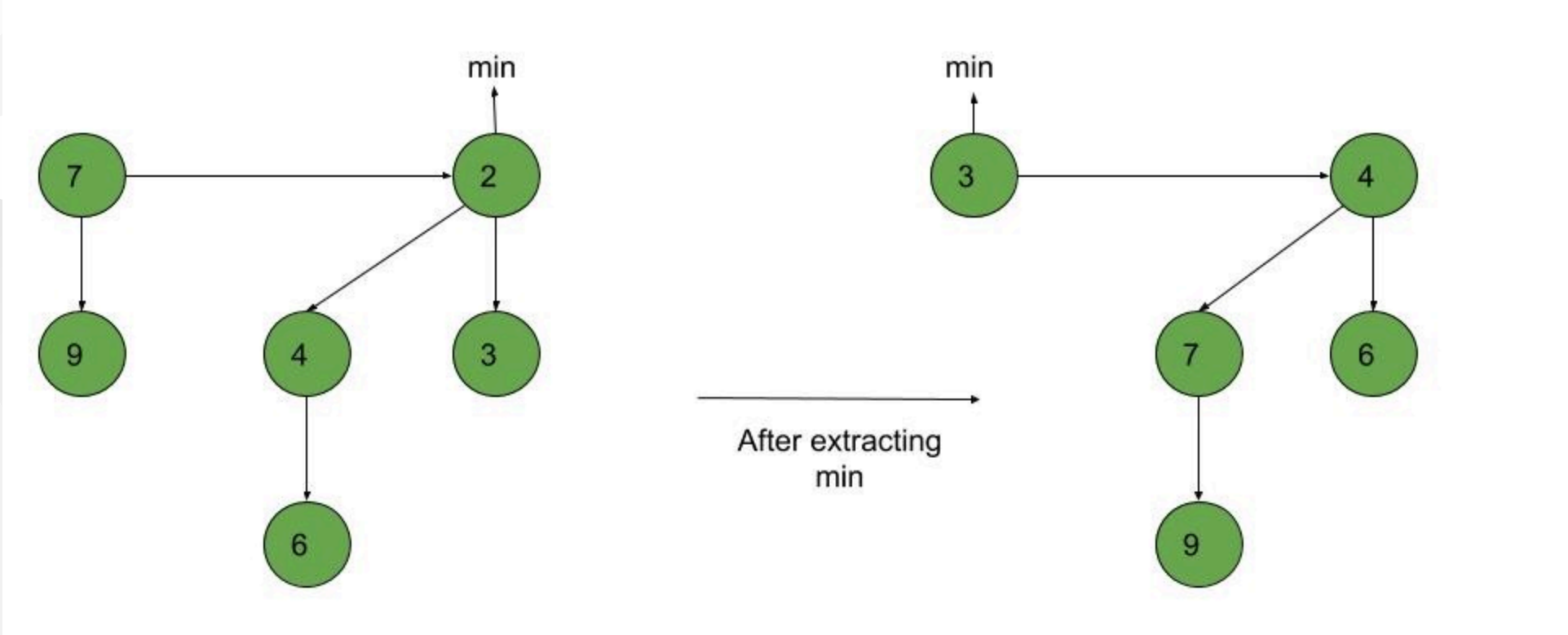
Move to the next node.

If degrees are different then **set** degree pointer to next node.

If degrees are the same then join the Fibonacci trees by **union operation**.

Repeat steps 4 and 5 until the heap is completed.

Example:



Decrease_key():

To decrease the value of any element in the heap, we follow the following algorithm:

Decrease the value of the node 'x' to the new chosen value.

CASE 1) If min-heap property is not violated,

Update min pointer if necessary.

CASE 2) If min-heap property is violated and parent of 'x' is unmarked,

Cut off the link between 'x' and its parent.

Mark the parent of 'x'.

Add tree rooted at 'x' to the root list and update min pointer if necessary.

CASE 3) If min-heap property is violated and parent of 'x' is marked,

Cut off the link between 'x' and its parent p[x].

Add 'x' to the root list, updating min pointer if necessary.

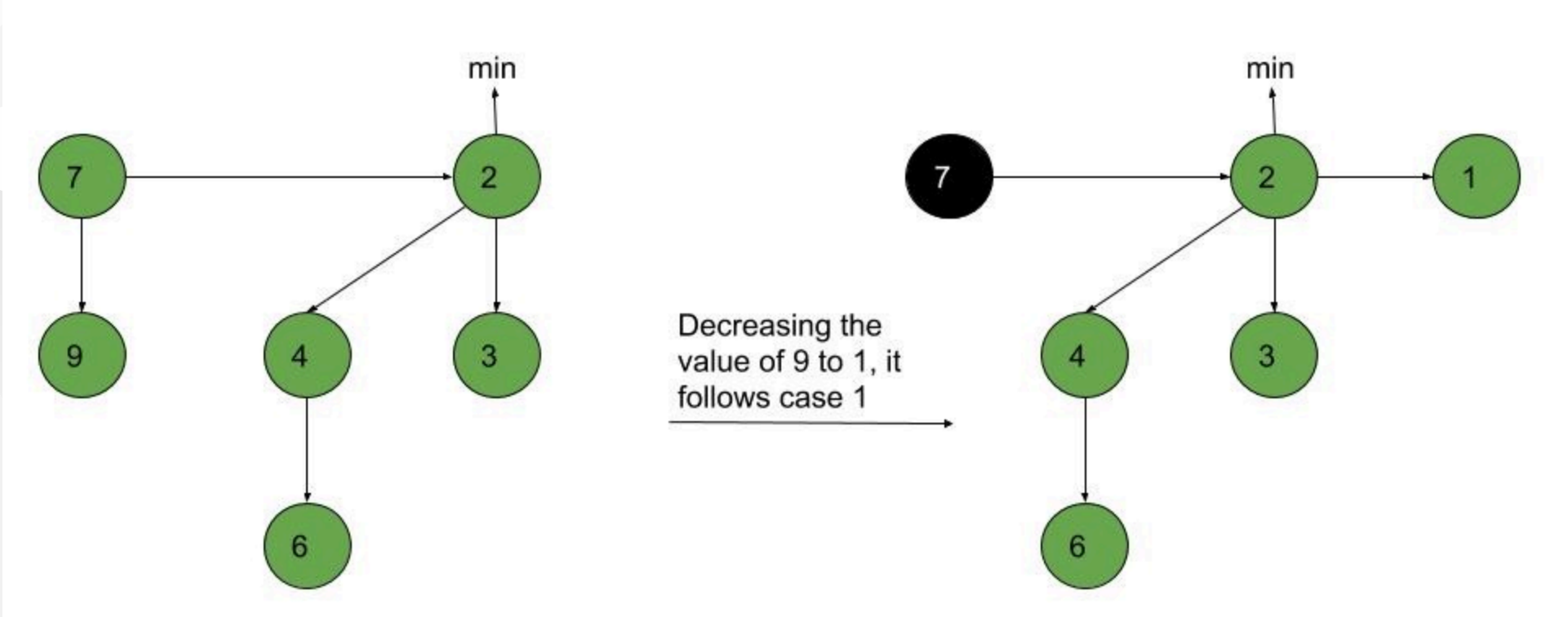
Cut off link between p[x] and p[p[x]].

Add p[x] to the root list, updating min pointer if necessary.

If p[p[x]] is unmarked, mark it.

Else, cut off p[p[x]] and repeat steps 4.2 to 4.5, taking p[p[x]] as 'x'.

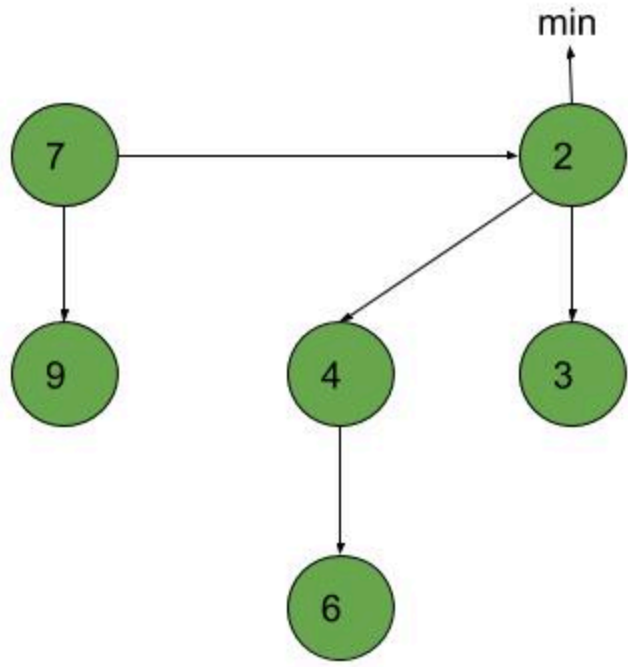
Example:



Deletion():

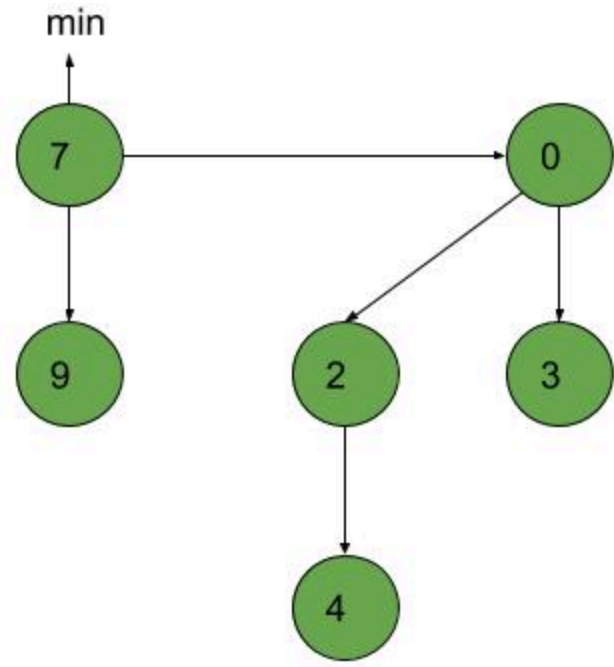
To delete any element in a Fibonacci heap, the following algorithm is followed:

Decrease the value of the node to be deleted 'x' to a minimum by `Decrease_key()` function. By using min-heap property, heapify the heap containing 'x', bringing 'x' to the root list. Apply `Extract_min()` algorithm to the Fibonacci heap.

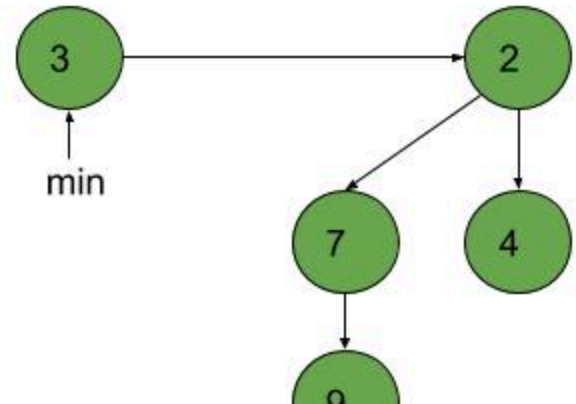


Delete 6

Decrease value of 6 to 0



Apply extract_min()



Heap Sort

- <https://ucoruh.github.io/ce100-algorithms-and-programming-II/week-4/ce100-week-4-heap/>

Huffman Coding

- Huffman Coding
 - <https://ucoruh.github.io/ce100-algorithms-and-programming-II/week-9/ce100-week-9-huffman/>
 - Geeks for Geeks
 - <https://www.geeksforgeeks.org/difference-between-binary-heap-binomial-heap-and-fibonacci-heap/?ref=rp>

End – Of – Week – 4