Module 4

Tree and Graph Structure

Content

Part 1: Tree Structure

- Binary Trees
- Tree Traversal
- AVL Trees
- Red-Black Trees

Part 2: Graph Structure

- Definition
- Graph Traversal
- Shortest Path

Objective

Tree vs Graph?

Tree

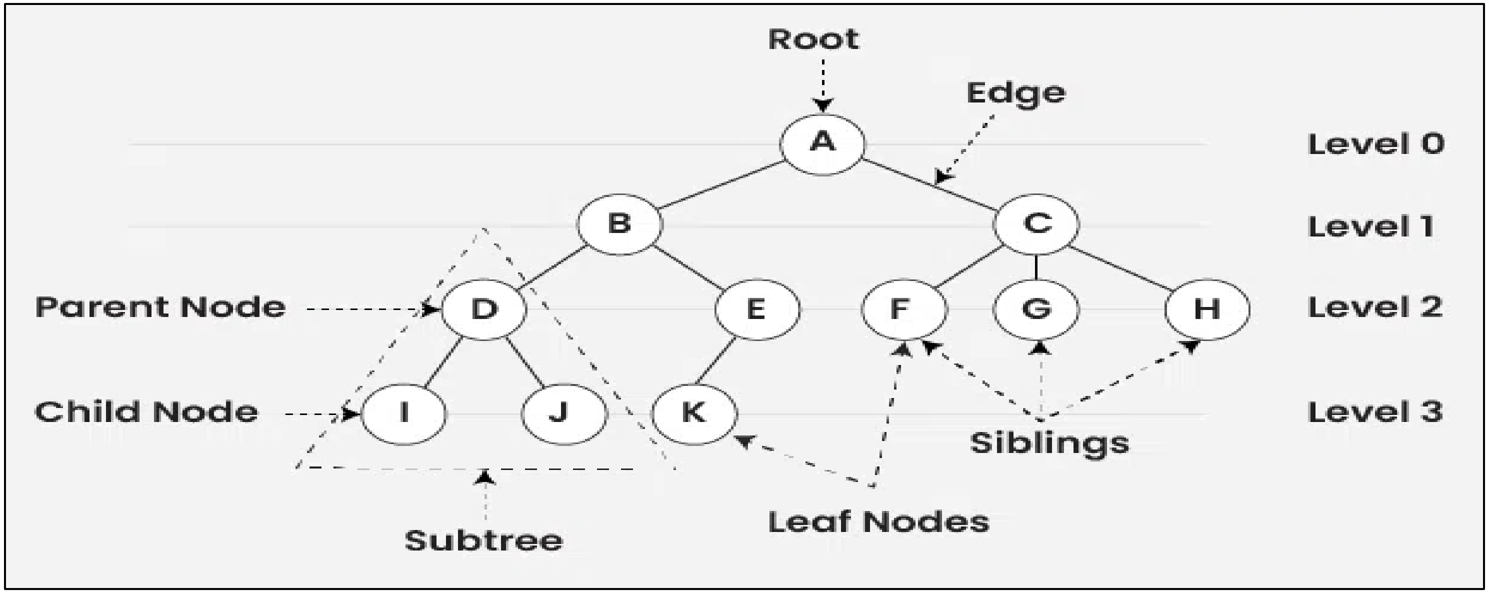
Hierarchical organization

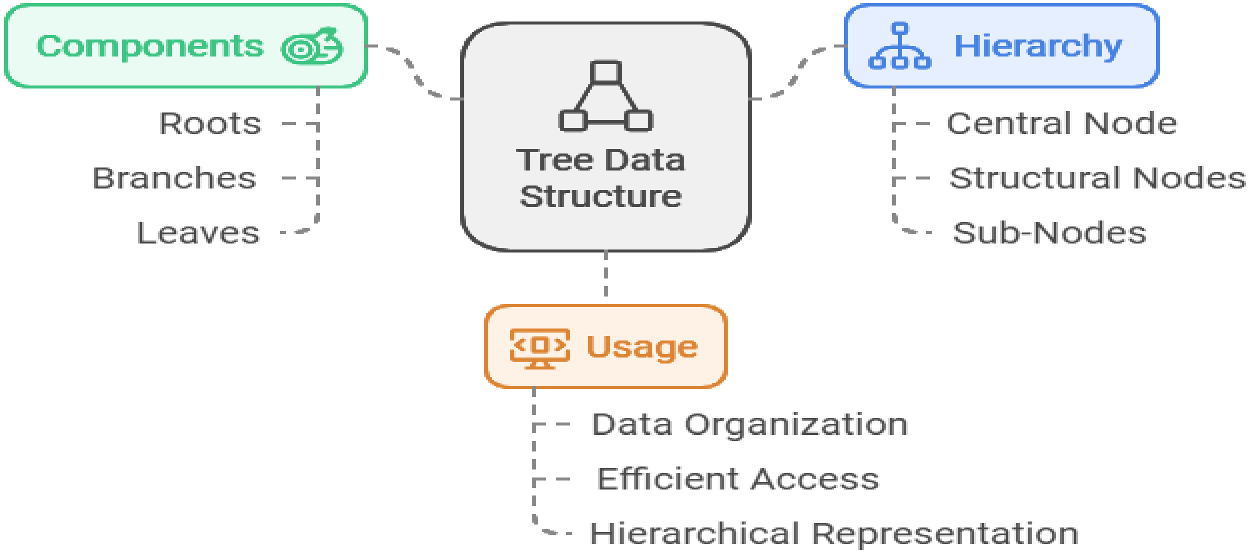
Graph

Flexible connections

Part 1: Tree Structure

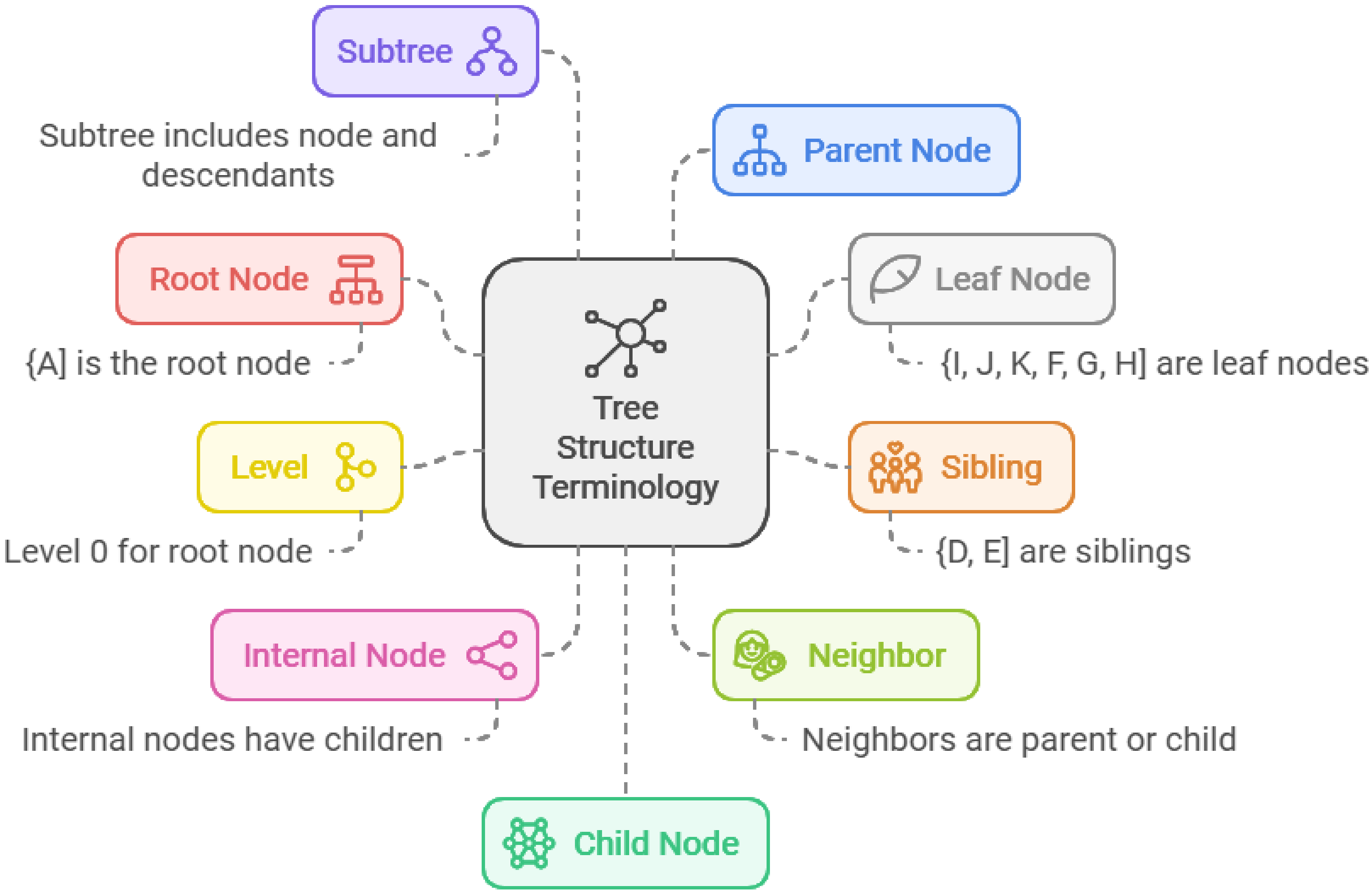
I. Tree Structures



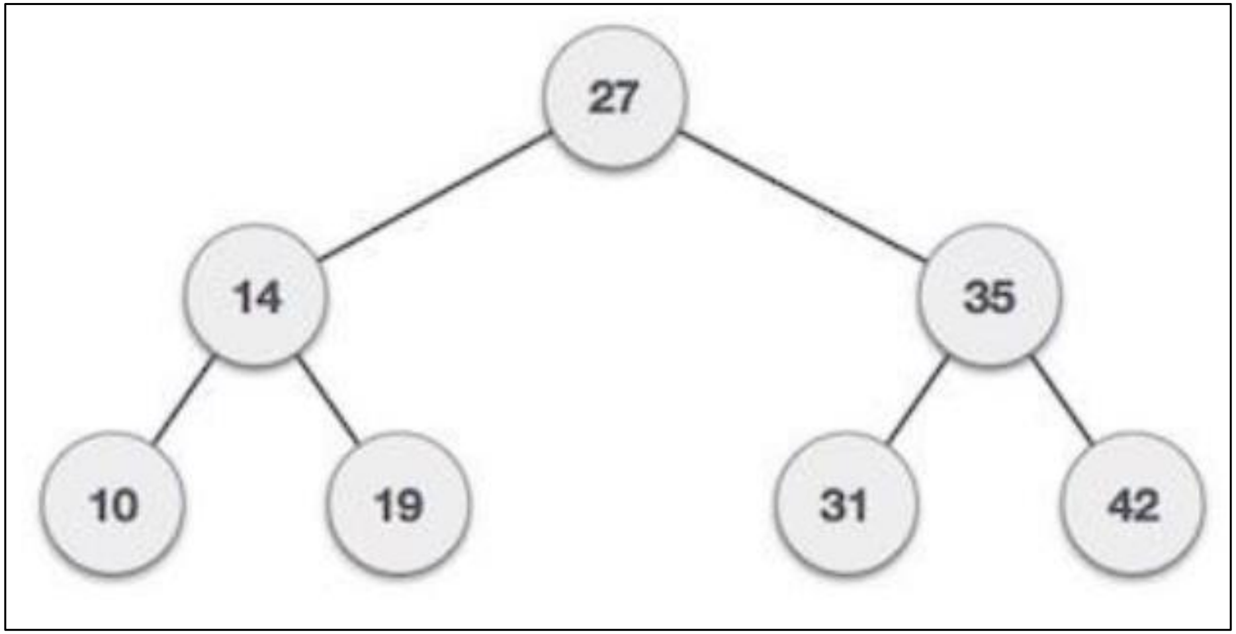


I. Tree Structures

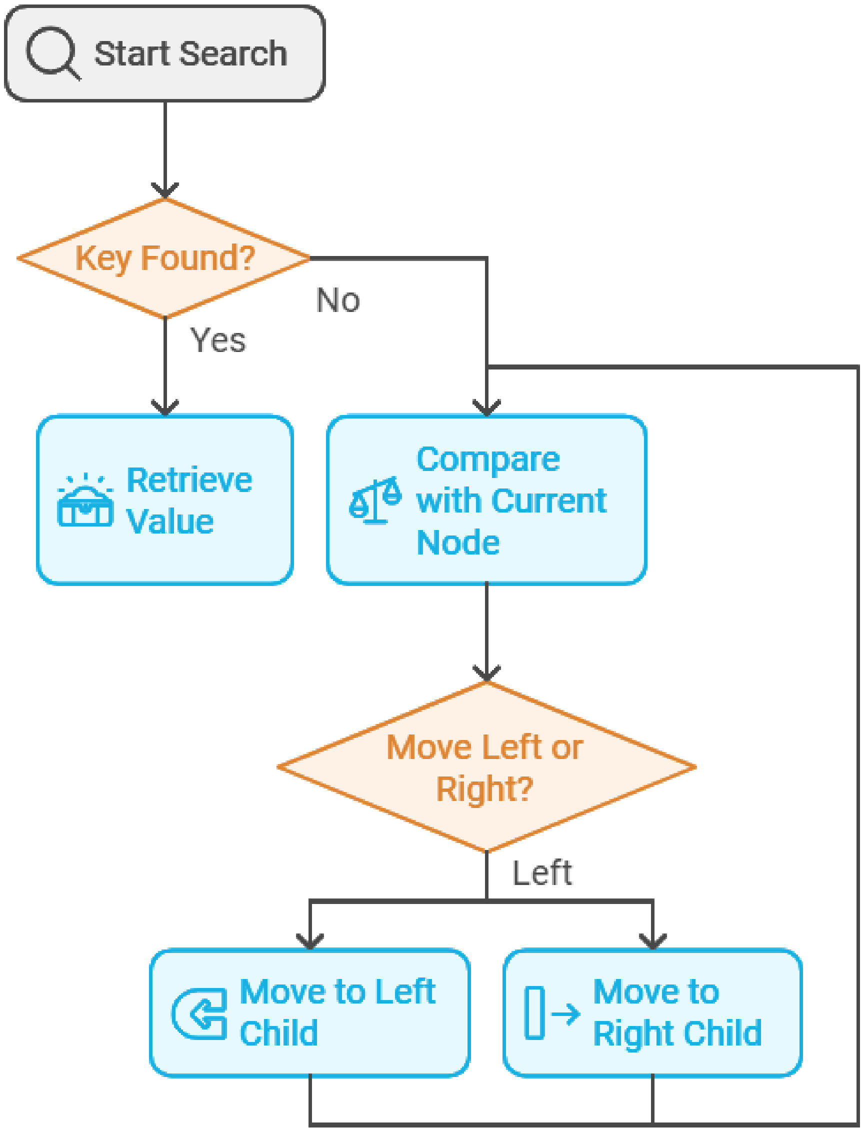
Basic Terminologies in Tree Data Structure:



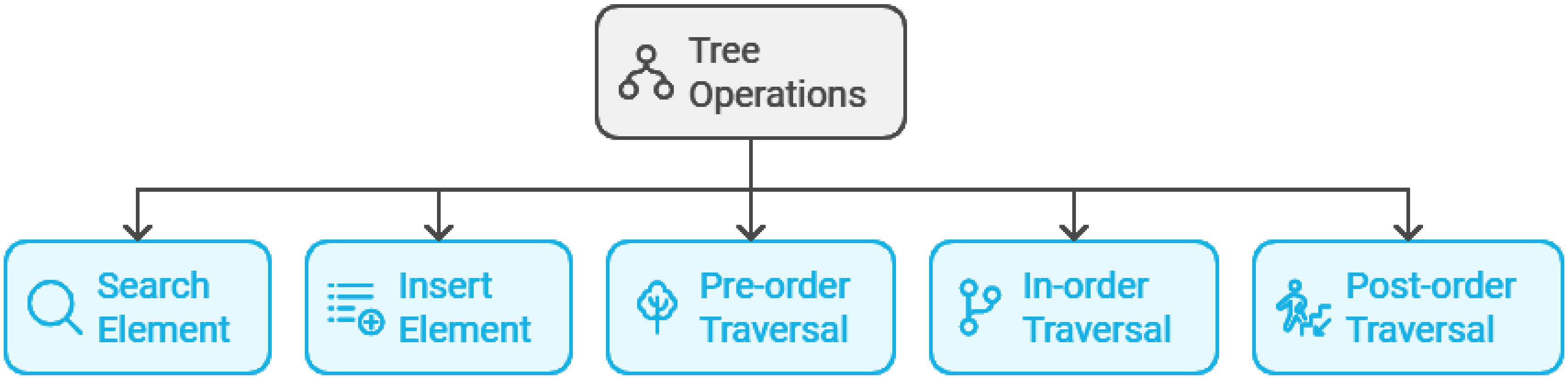
II. Binary Tree Structures



II. Binary Tree Structures



II. Binary Tree Structures




```
class Program
{   static void Main(string[] args)
   {   BinaryTree tree = new BinaryTree();
       tree.Add(5);        tree.Add(3);
       tree.Add(7);        tree.Add(2);
       tree.Add(4);        tree.Add(6);        tree.Add(8);
       Console.WriteLine("In-Order Traversal:");
       tree.InOrderTraversal(tree.Root);    }
}
```

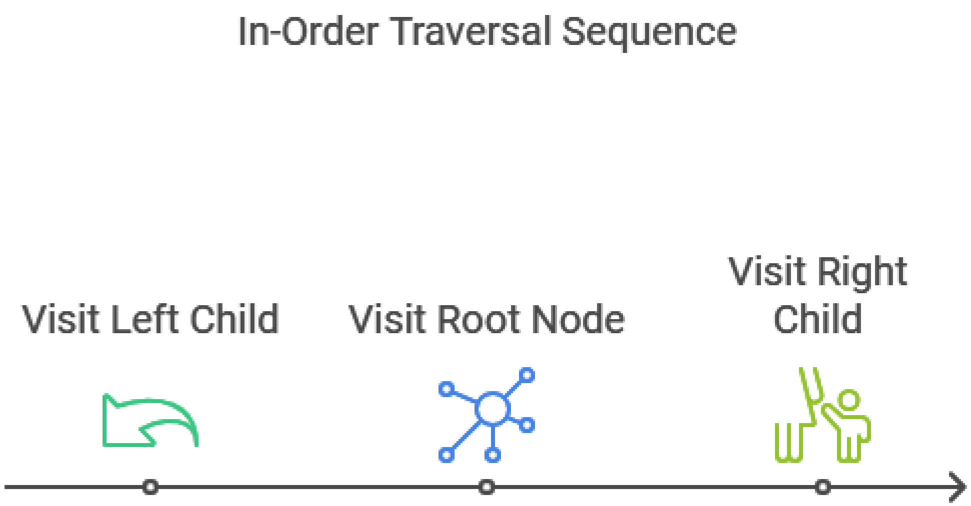
```
public class Node
{   public int Data { get; set; }
    public Node Left { get; set; }
    public Node Right { get; set; }
    public Node(int data)
    {  this.Data = data;
       this.Left = null;
       this.Right = null;    }
}
```

II. Binary Tree Structures

```
public class BinaryTree
{  public Node Root { get; set; }
   public BinaryTree()
   {    this.Root = null;    }
   public void Add(int value)
   {   Node newNode = new Node(value);
      if (Root == null)
      {   Root = newNode;       }
     else { AddRecursively(Root, newNode);       }
   }
   private void AddRecursively(Node current, Node
newNode)
   {   if (newNode.Data < current.Data)
      {
        if (current.Left == null)
        {   current.Left = newNode;           }
else  { AddRecursively(current.Left, newNode); }
        }
        else {  if (current.Right == null)
            {   current.Right = newNode;     }
            else {  AddRecursively(current.Right,
newNode);   }    }
   }
   public void InOrderTraversal(Node node)
   {   if (node != null)
       {   InOrderTraversal(node.Left);
           Console.Write(node.Data + " ");
           InOrderTraversal(node.Right);       }   }
}
```

III. Binary Tree Traversal

a. in-order




```
using System;
class Node {
    public int data;
    public Node left, right;
    public Node(int item) {
        data = item;
        left = right = null;    }
}
```

III. Binary Tree Traversal

```
class GFG {
    public static void InOrderTraversal(Node root) {
        if (root == null) return;
        // Traverse the left subtree
        InOrderTraversal(root.left);
        // Visit the root node
        Console.Write(root.data + " ");
        // Traverse the right subtree
        InOrderTraversal(root.right);
    }
```

```
static void Main(string[] args) {
        Node root = new Node(2);
        root.left = new Node(1);
        root.right = new Node(3);
        root.left.left = new Node(4);
        root.left.right = new Node(5);
        Console.Write("In-Order Traversal: ");
        InOrderTraversal(root);
        Console.WriteLine();
    }
}
```

Output:

In-order traversal of binary tree is: 4 2 5 1 3 6

III. Binary Tree Traversal

b. pre-order


```
using System;
class Node {
   public int data;
   public Node left, right;
   public Node(int item) {
      data = item;
      left = right = null;    }
}
class GFG {
   public static void PreOrderTraversal(Node root) {
      if (root == null) return; // Visit the root node
```

```
// Visit the root node
      Console.Write(root.data + " ");
      // Traverse the left subtree
      PreOrderTraversal(root.left);
      // Traverse the right subtree
      PreOrderTraversal(root.right);
   }
   static void Main(string[] args) { // Create the following binary tree
      Node root = new Node(1);           //      1
      root.left = new Node(2);           //     / \
```

III. Binary Tree Traversal

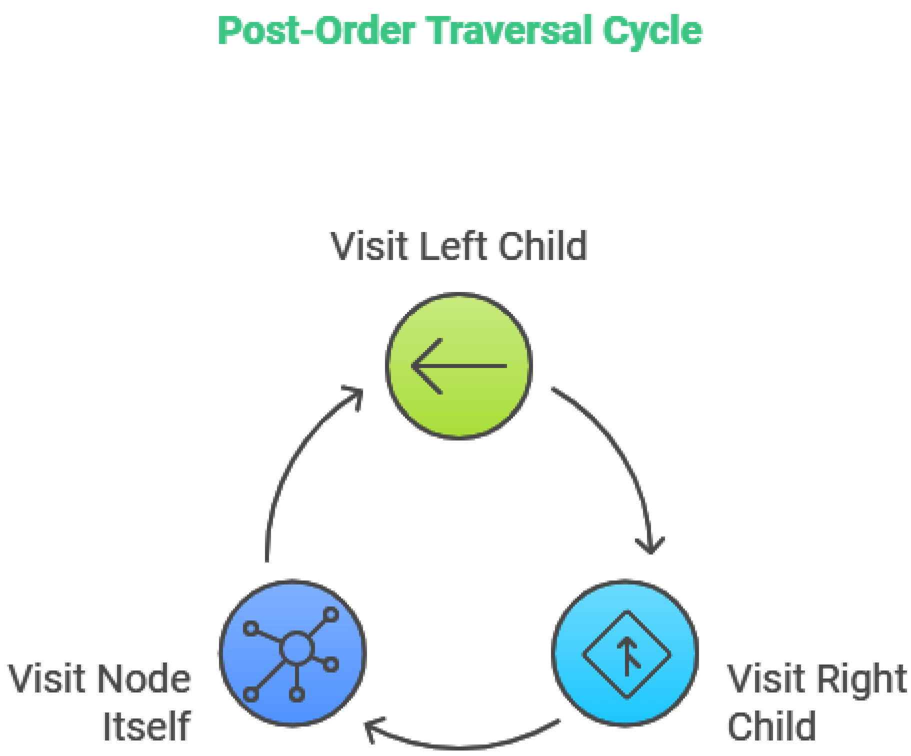
```
        root.right = new Node(3);            //    2   3
        root.left.left = new Node(4);        //   / \
        root.left.right = new Node(5);       //  4   5
        Console.Write("Pre-Order Traversal: ");
        PreOrderTraversal(root);
        Console.WriteLine();    }
}
```

Output:

Pre-Order Traversal: 1 2 4 5 3

III. Binary Tree Traversal

c. **post-order**

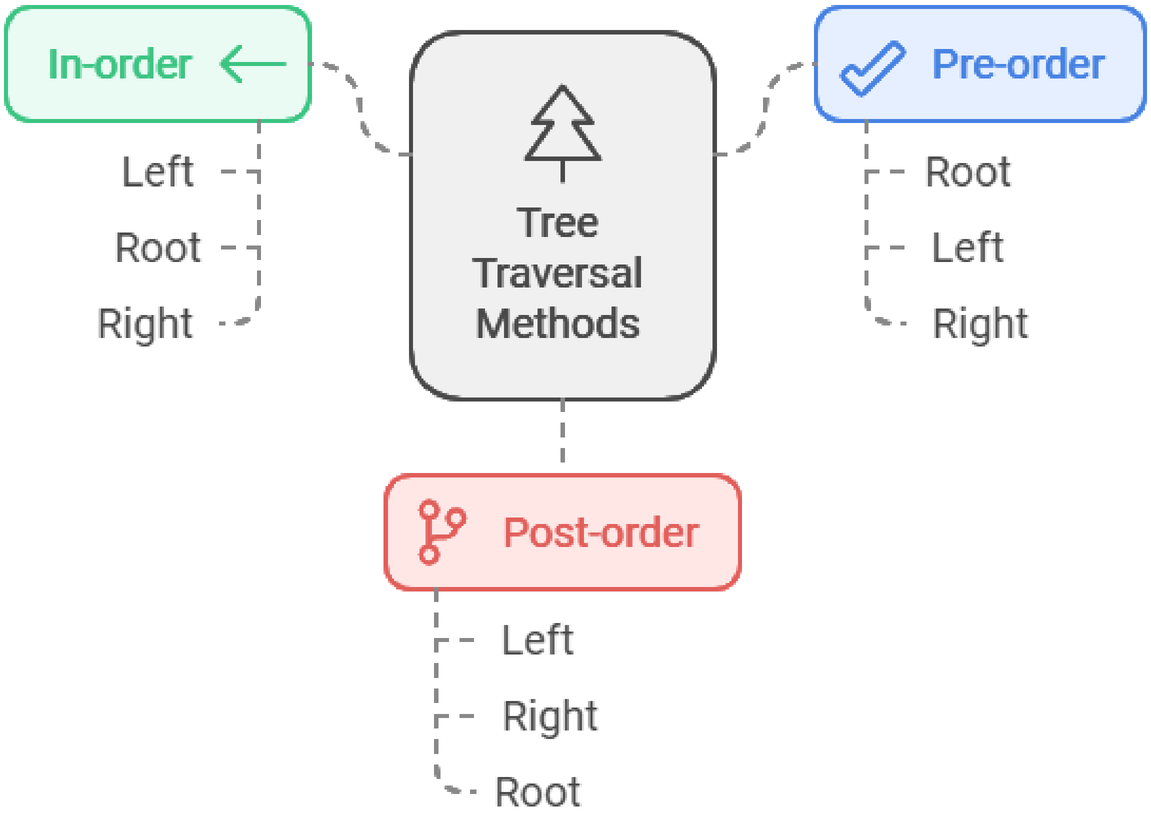


III. Binary Tree Traversal

```
using System;
class Node {
    public int data;
    public Node left, right;
    public Node(int item) {
        data = item;
        left = right = null;    }
}
class GFG {
    public static void PostOrderTraversal(Node root) {
        if (root == null) return;
        // Traverse the left subtree
        PostOrderTraversal(root.left);
      // Traverse the right subtree
        PostOrderTraversal(root.right);
        // Visit the root node
        Console.Write(root.data + " ");
}
static void Main(string[] args) {     // Create the following binary tree
        Node root = new Node(1);              //      1
        root.left = new Node(2);              //     / \
        root.right = new Node(3);             //    2   3
        root.left.left = new Node(4);         //   / \
        root.left.right = new Node(5);        //  4   5
        Console.Write("Post-Order Traversal: ");
        PostOrderTraversal(root);
        Console.WriteLine();    }
}
```

Output: Post-Order Traversal: 4 5 2 3 1

III. Binary Tree Traversal



V. AVL Tree

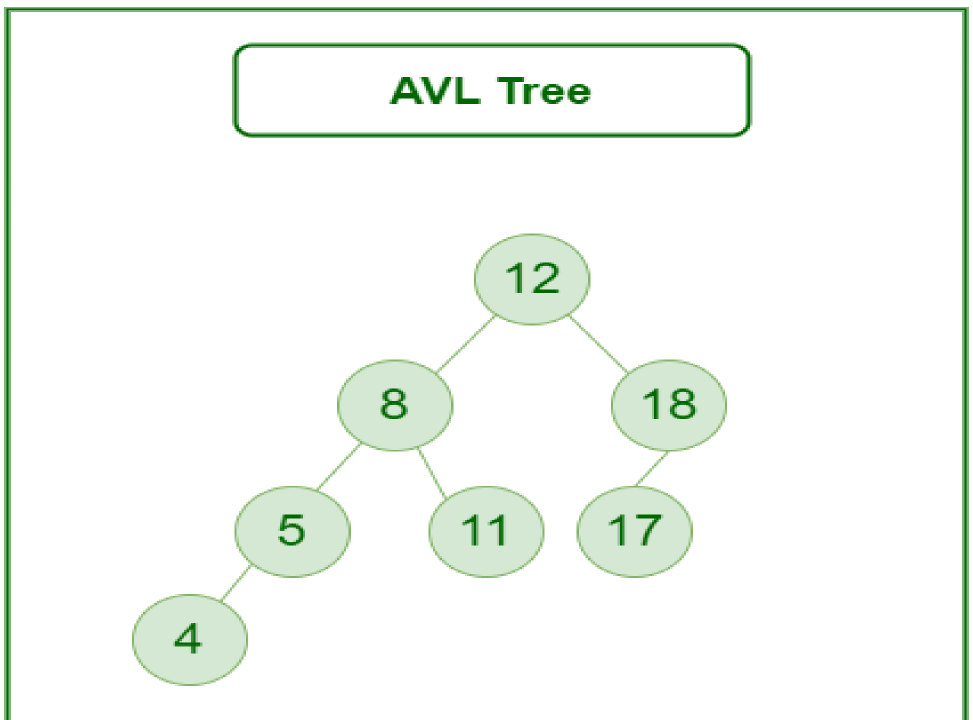
Which self-balancing BST to implement?

AVL Tree

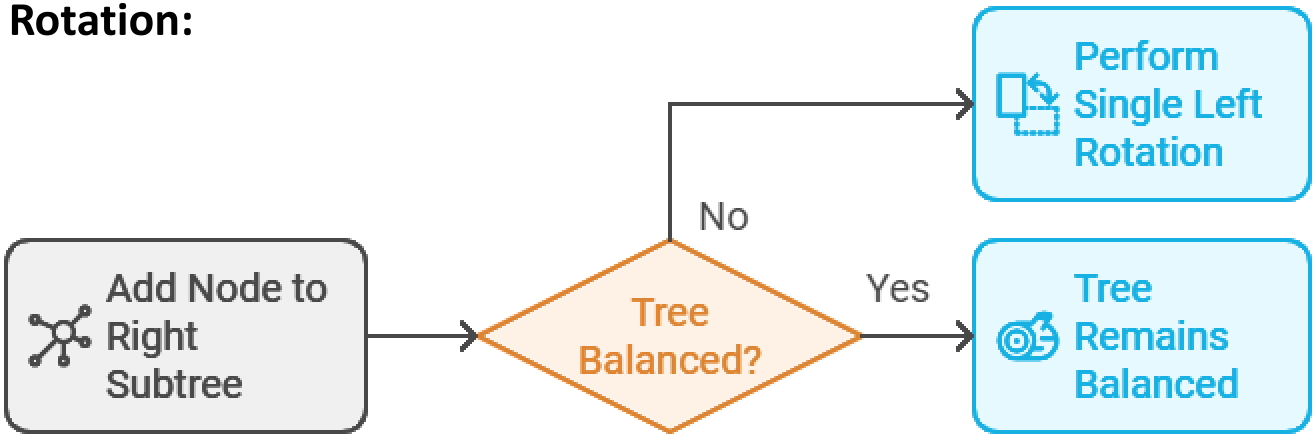
Guarantees O(log n) time for search, insertion, and deletion.

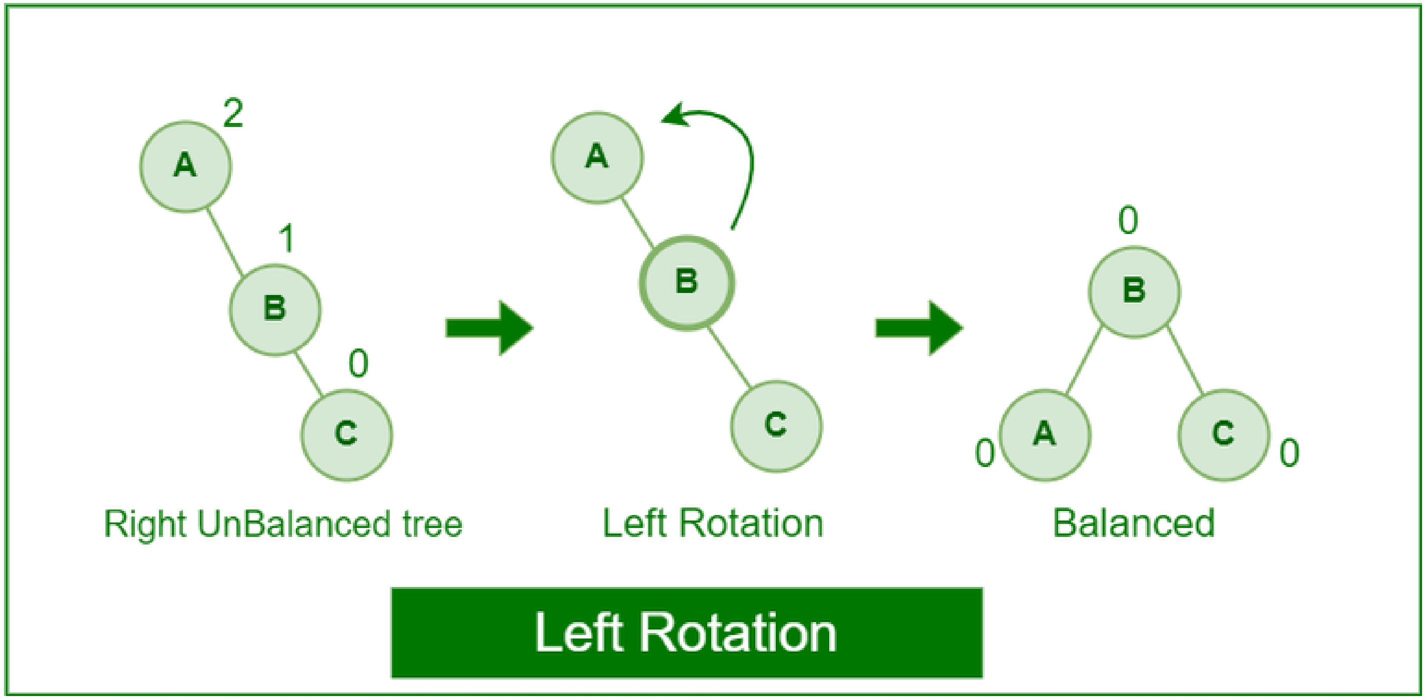
Red-Black Tree

Slightly faster for insertion and deletion, but more complex.



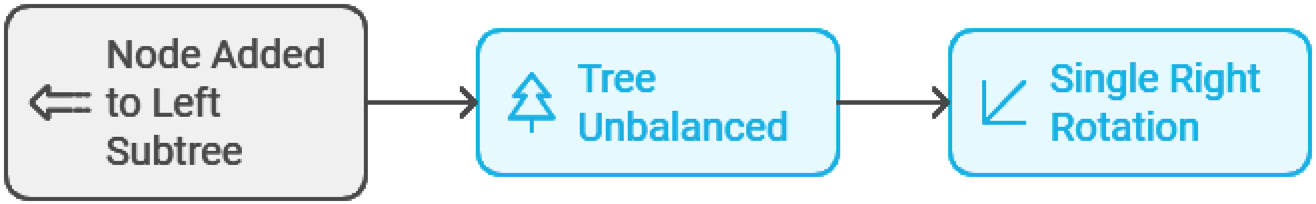
a) **Left Rotation:**

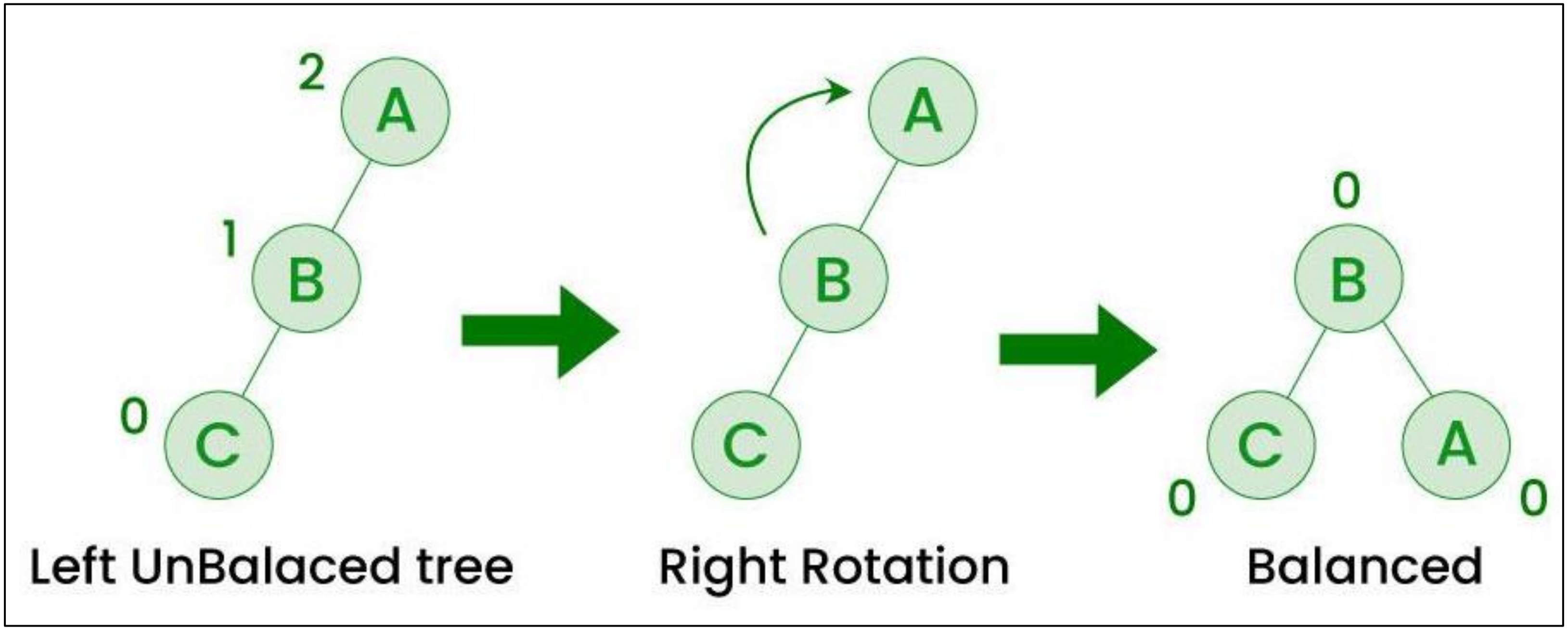




V. AVL Tree

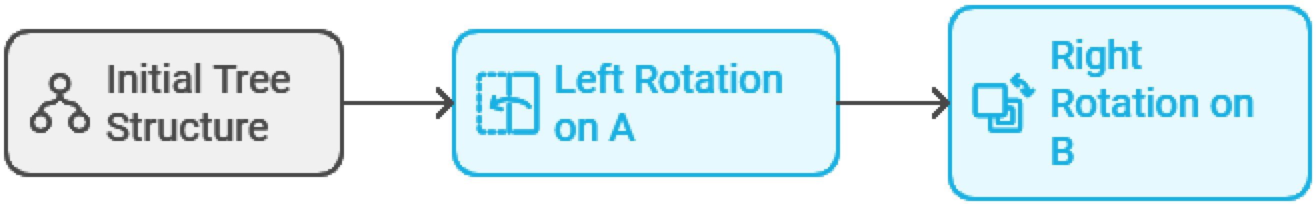
b) Right Rotation:

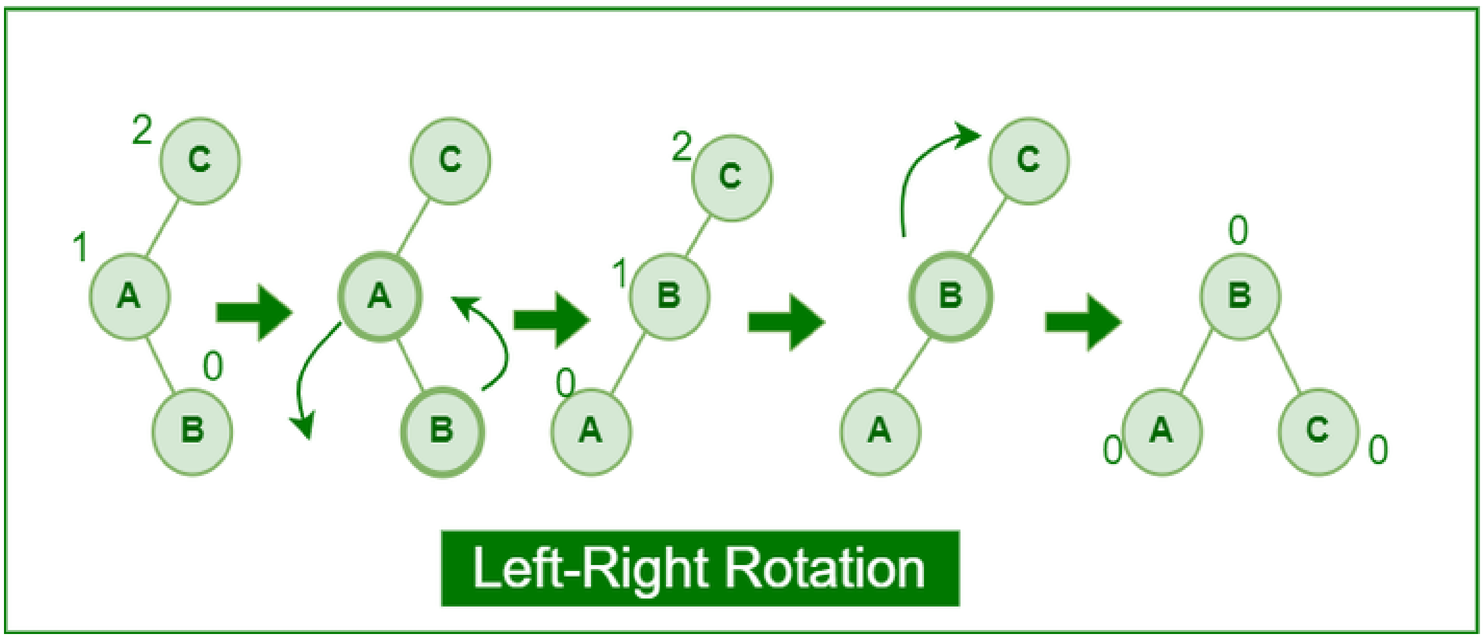




V. AVL Tree

c) Left-Right Rotation:

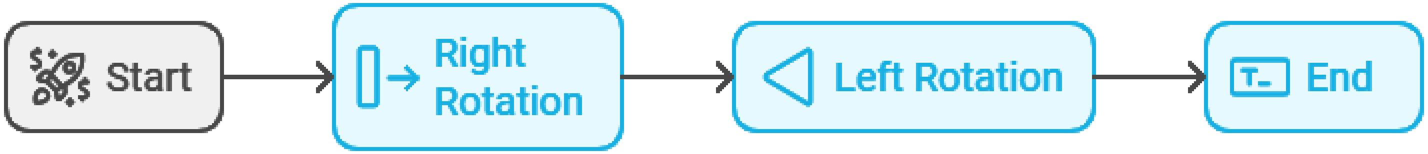


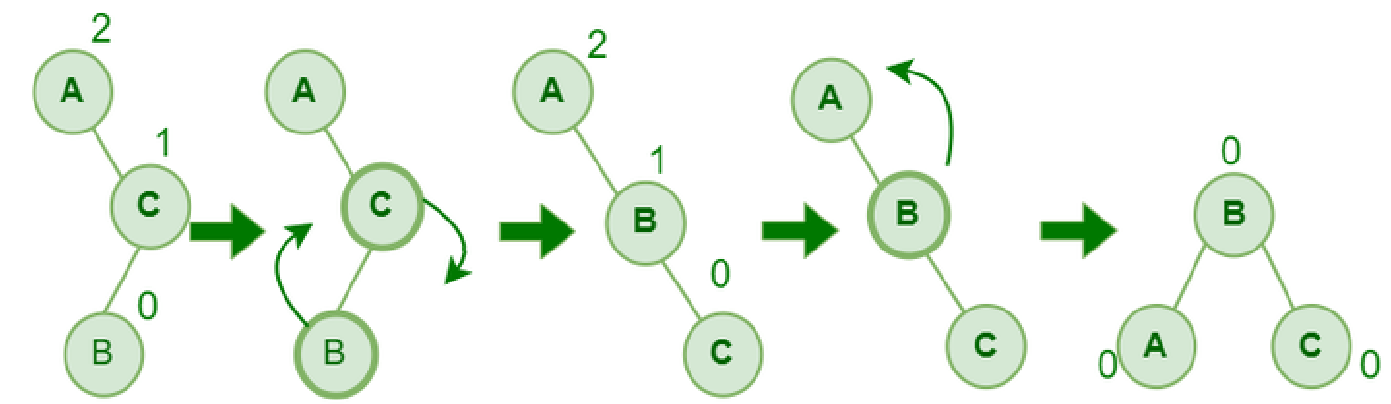


Left-Right Rotation

V. AVL Tree

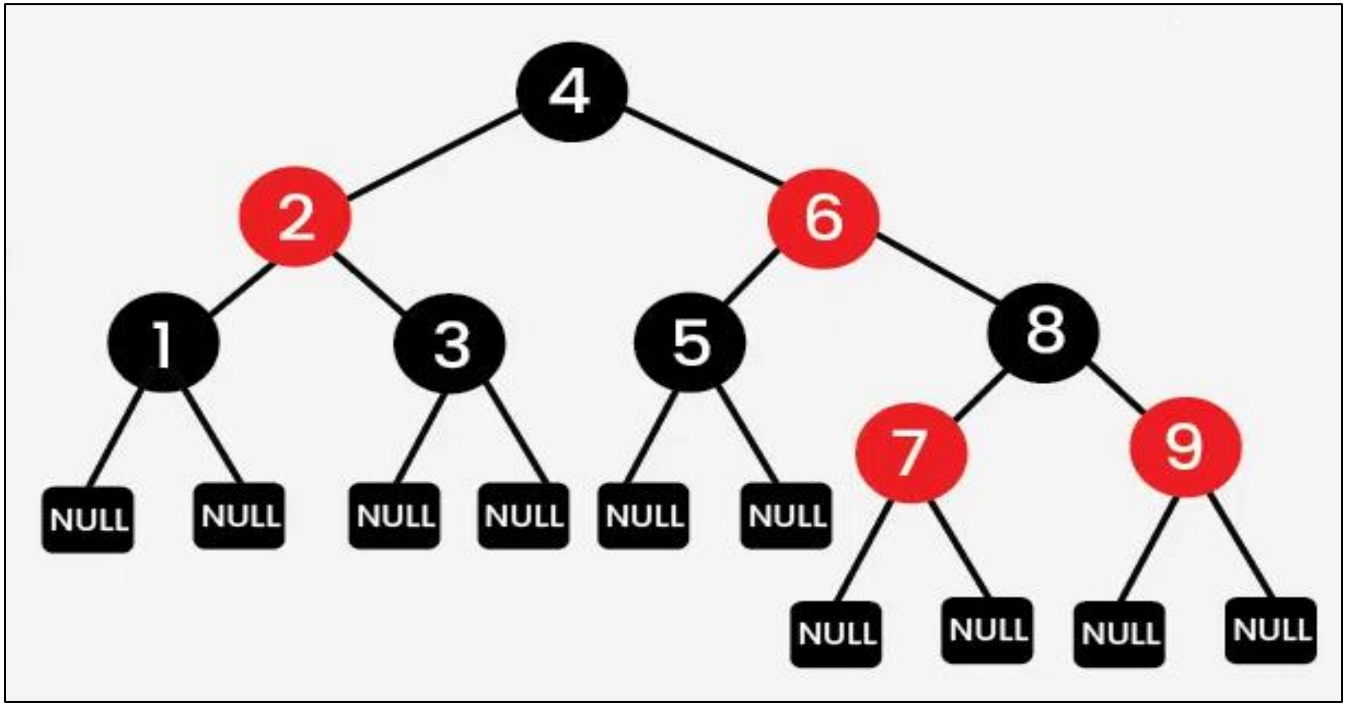
d) Right-Left Rotation:



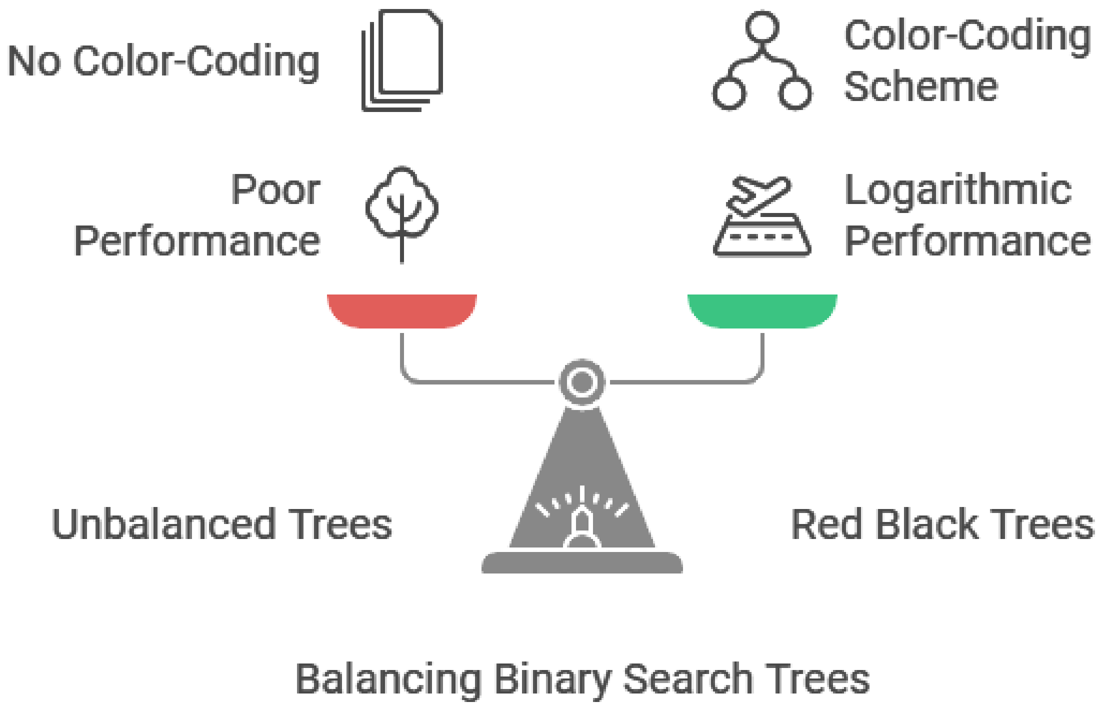


Right-Left Rotation

VI. Red-black Tree

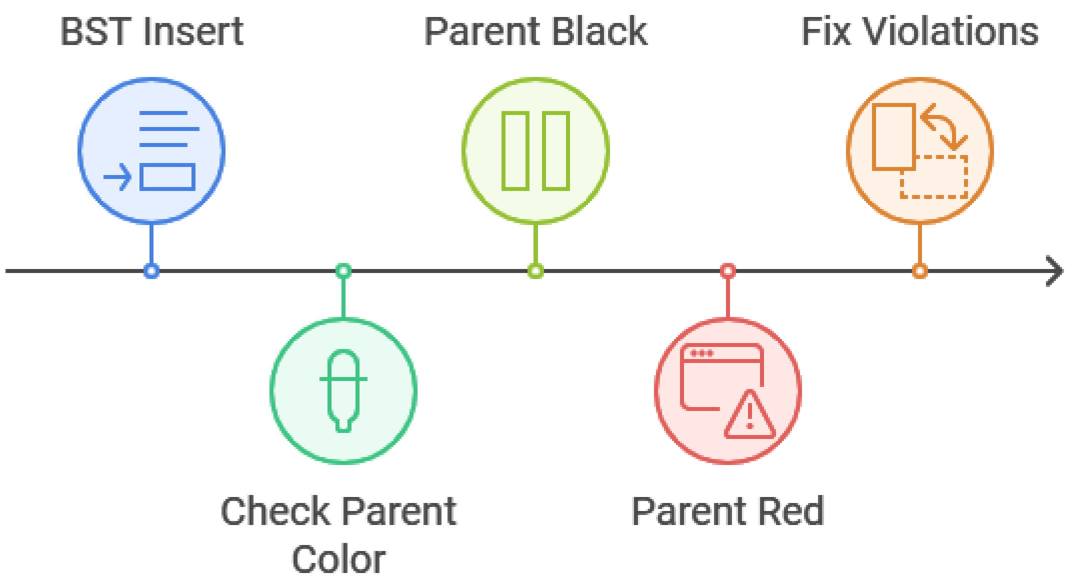


VI. Red-black Tree



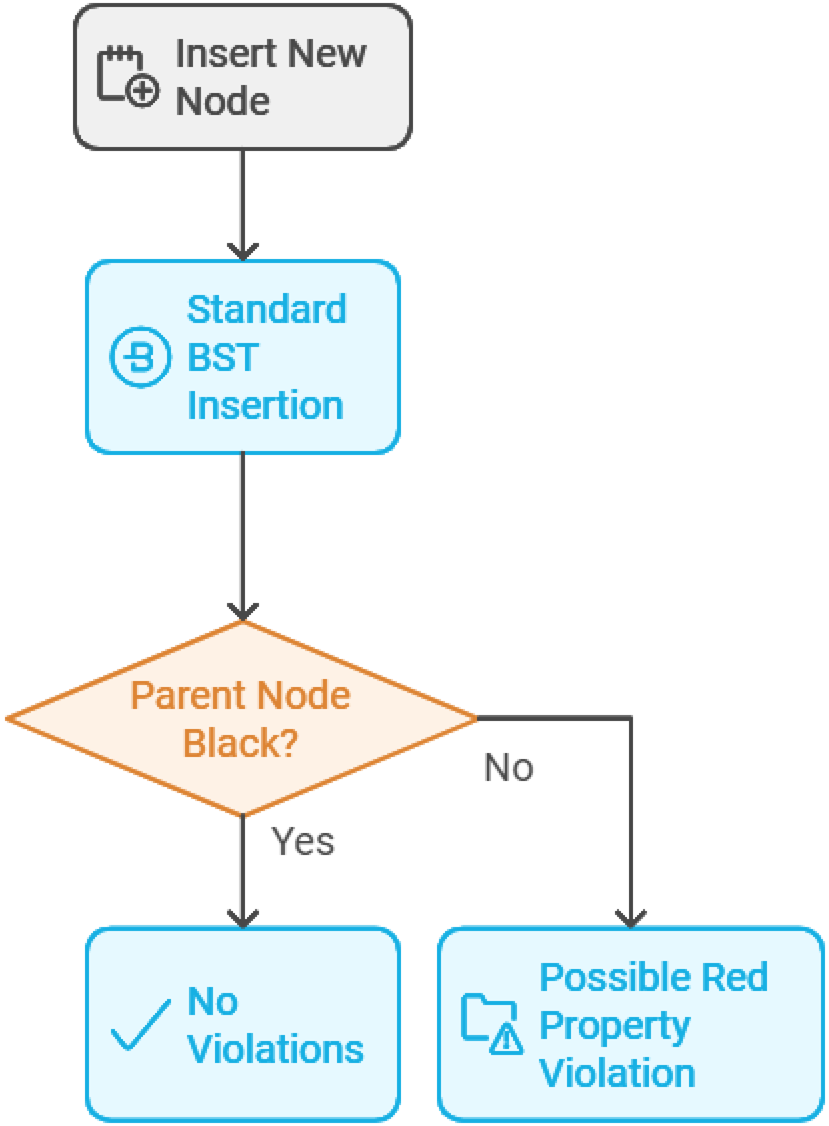
VI. Red-black Tree

a) Insertion



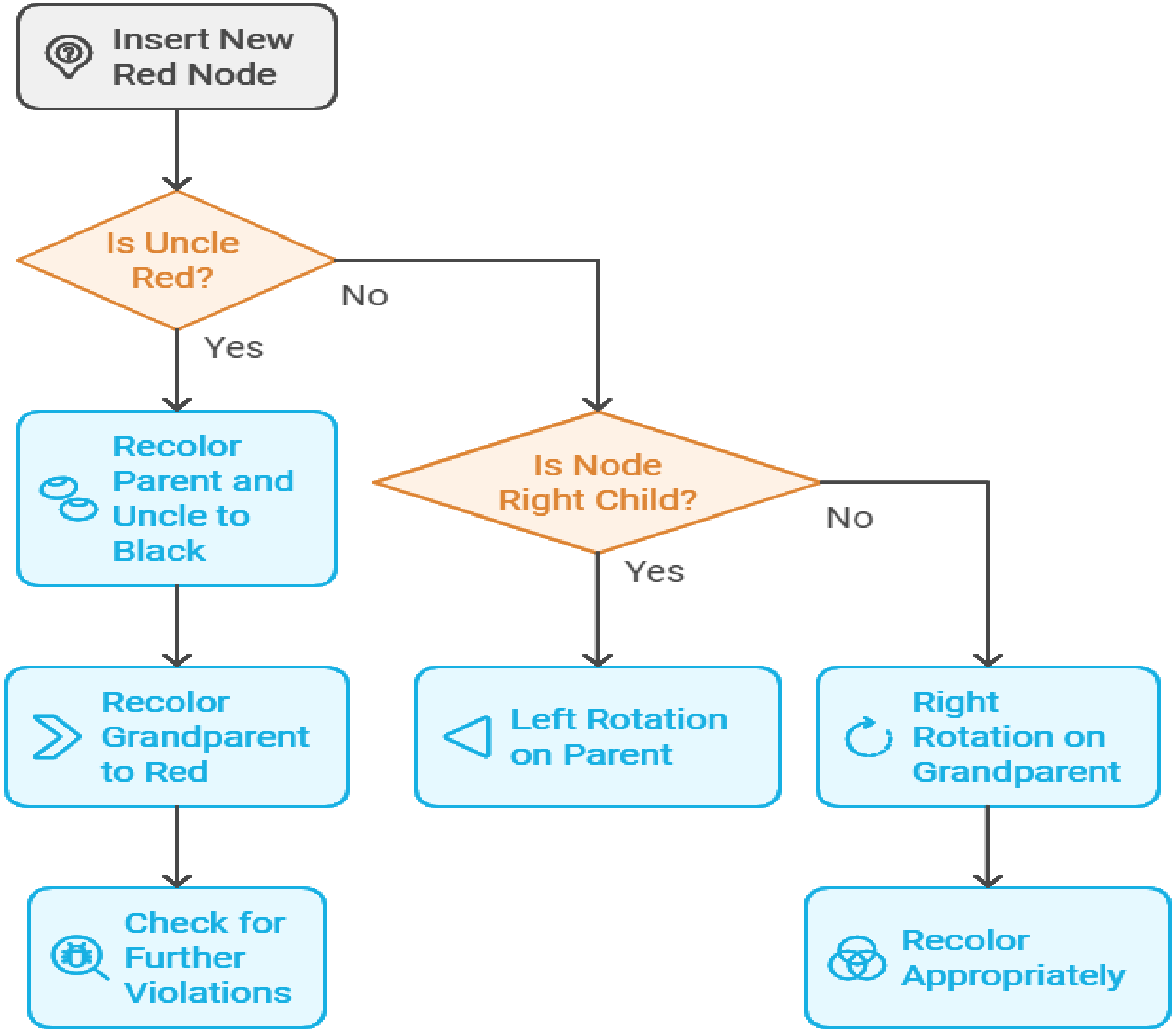
VI. Red-black Tree

a) **Insertion**



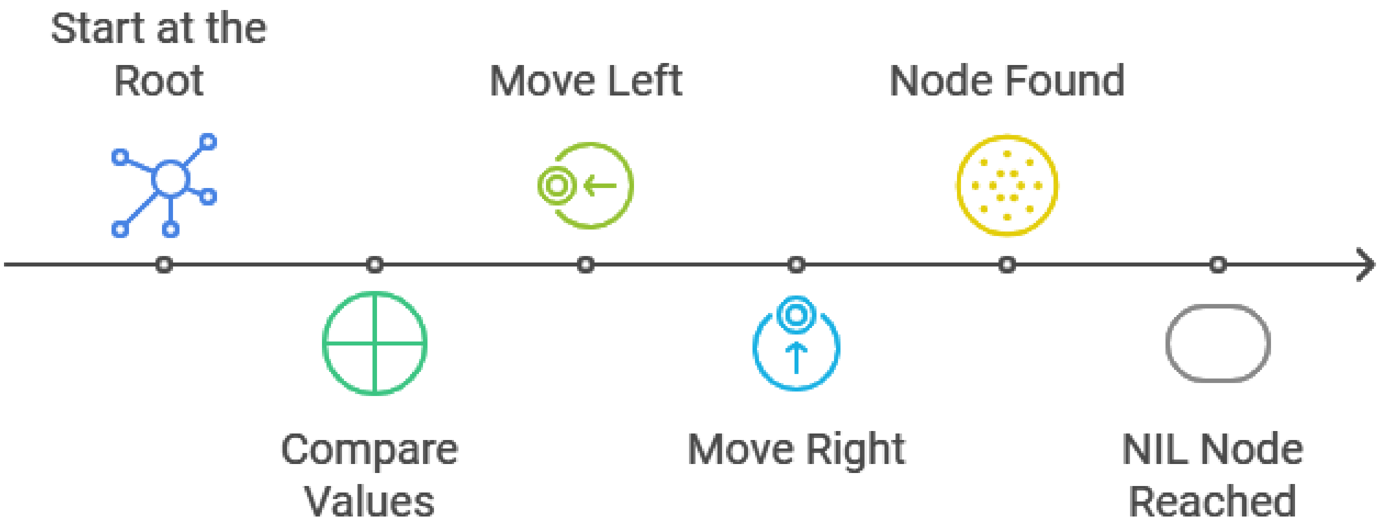
VI. Red-black Tree

Fixing Violations During Insertion



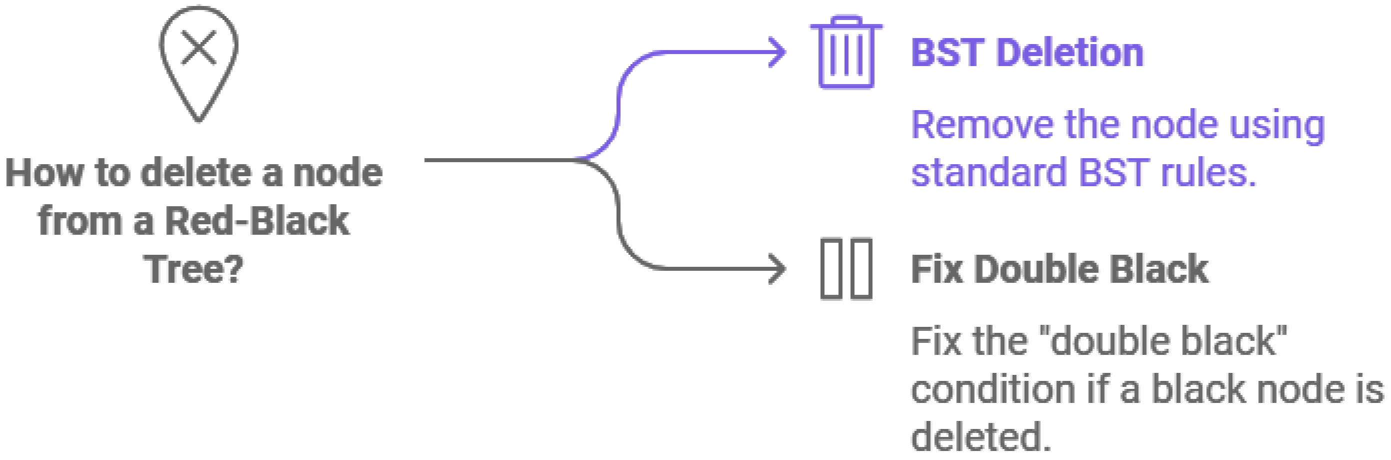
VI. Red-black Tree

b) Searching

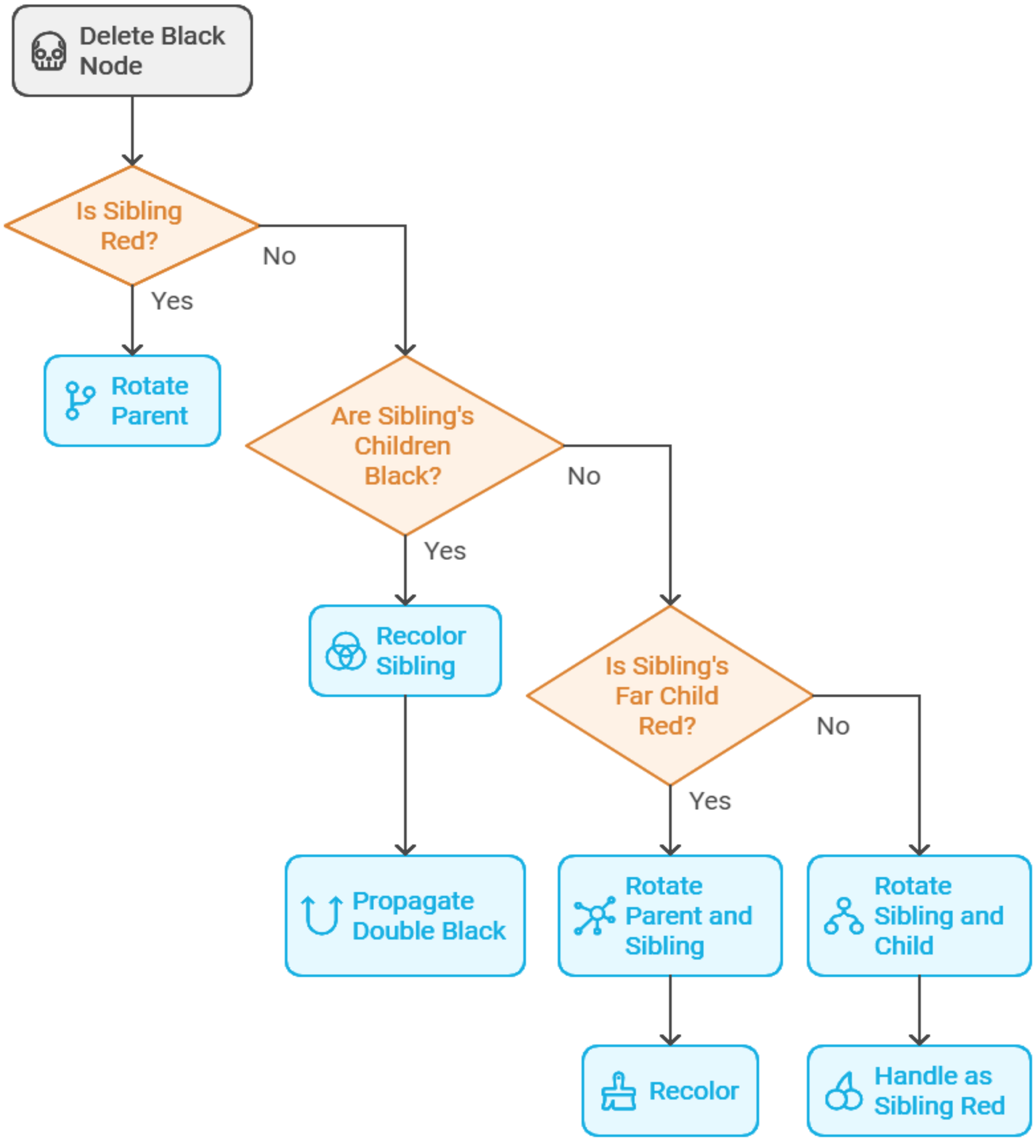


VI. Red-black Tree

c) Deletion



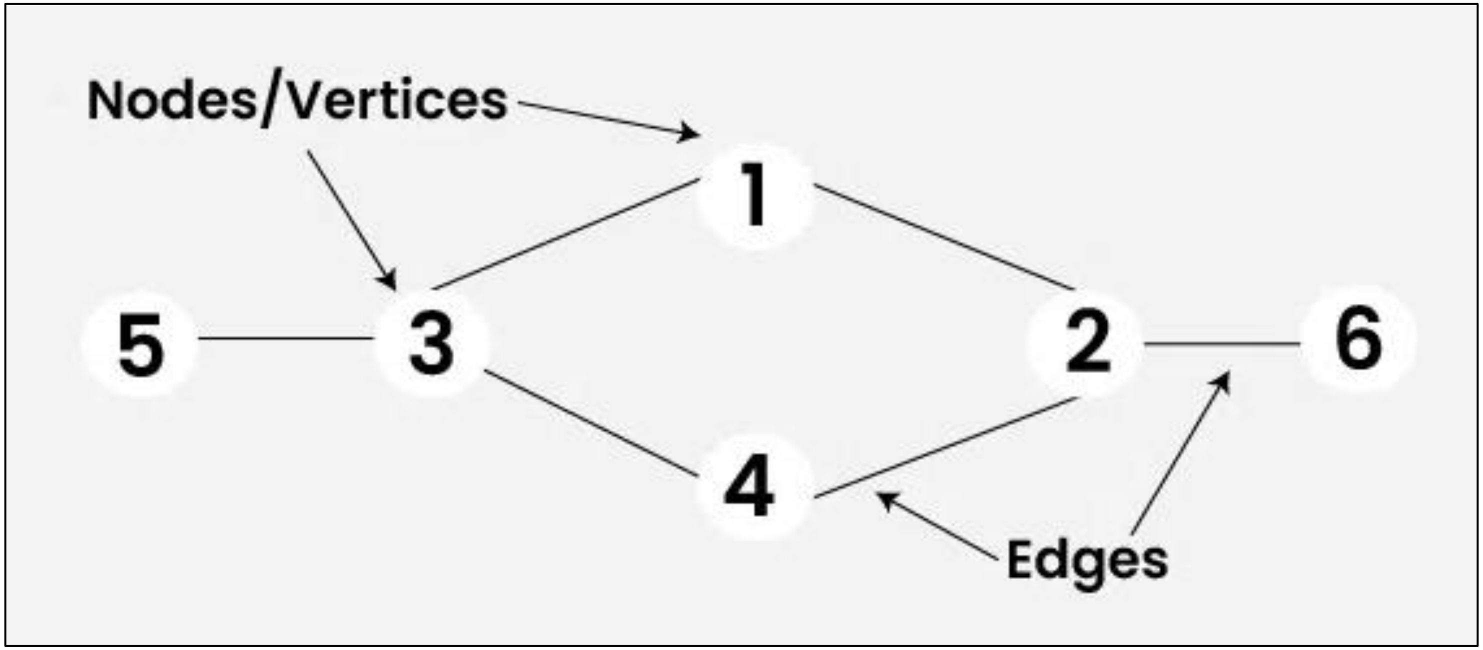
VI. Red-black Tree

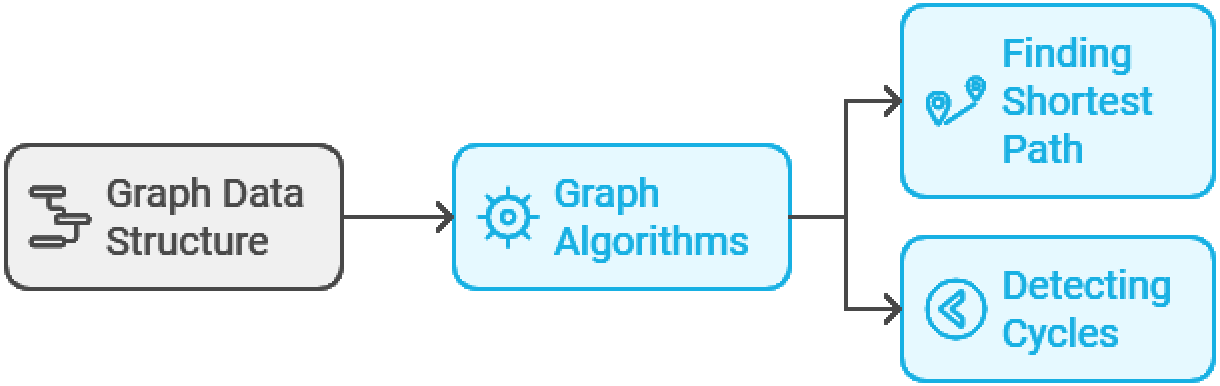


Part 2: Graph Structure

I. Definition and representation

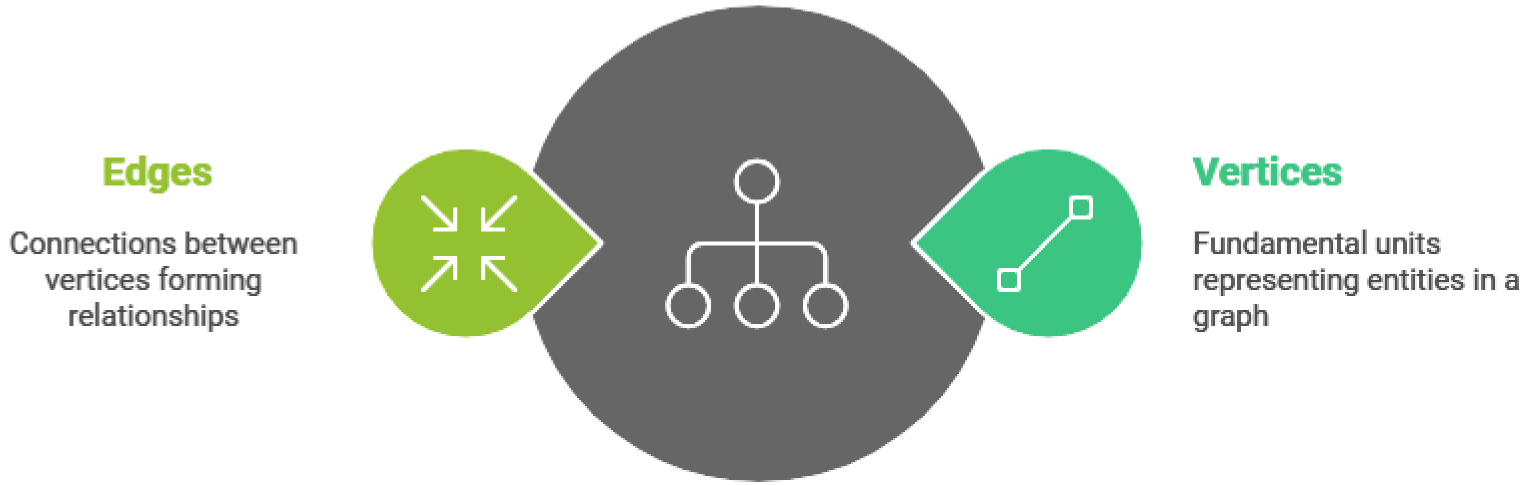
(adjacency matrix, adjacency list)



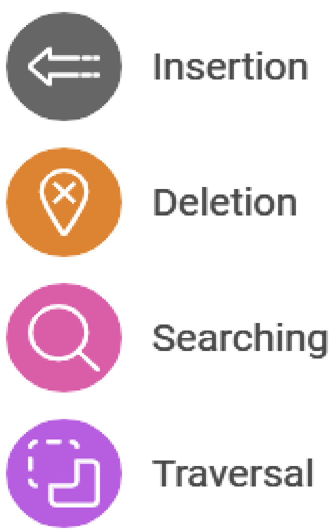


I. Definition and representation

a) Components of a Graph



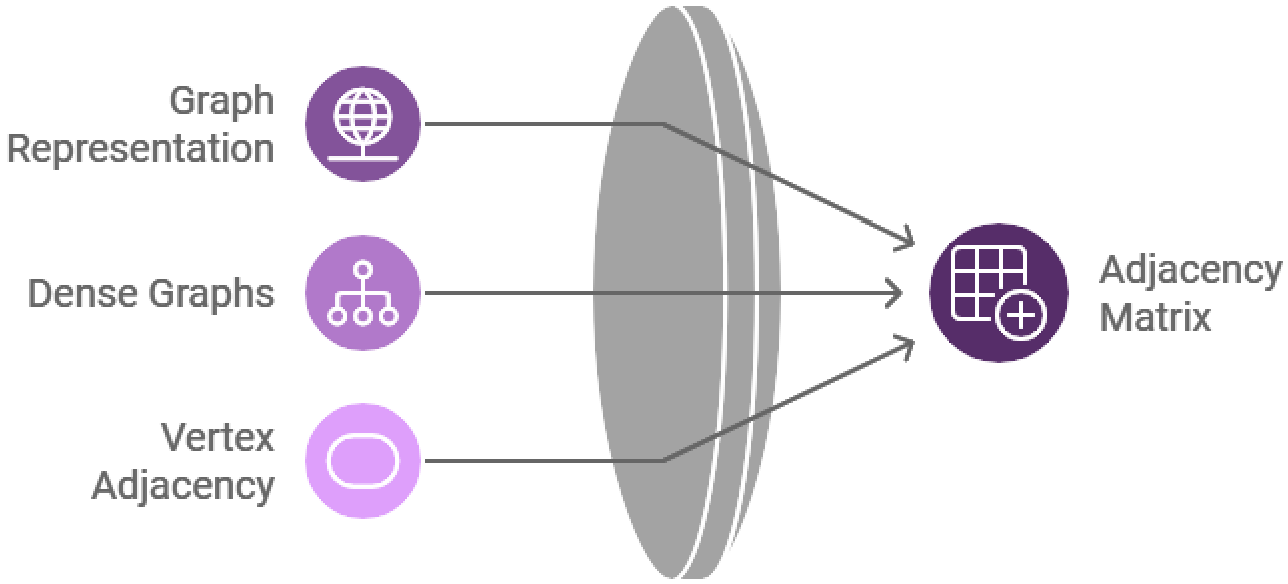
b) Basic Operations



I. Definition and representation

c) Adjacency Matrix

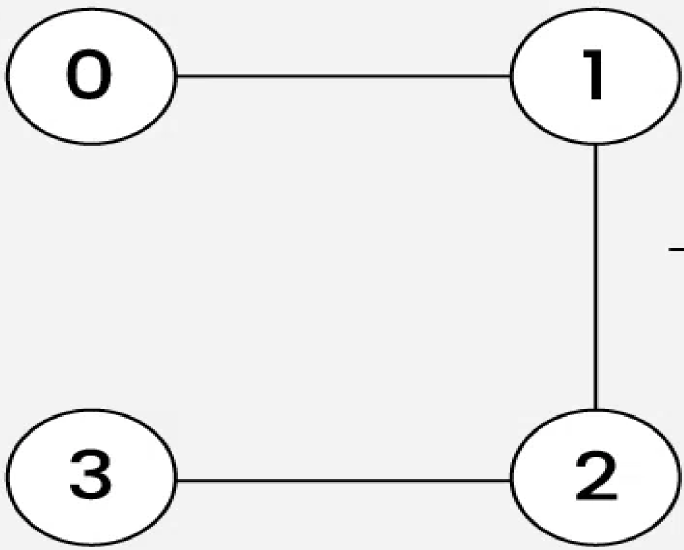
Adjacency Matrix: A Graph Representation



I. Definition and representation

c) Adjacency Matrix

Adjacency Matrix for Undirected and Unweighted graph



	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

Adjacency Matrix A[ ]

I. Definition and representation

c) Adjacency Matrix

A[i][j] = 1, if there is an edge between vertex i and vertex j

A[i][j] = 0, if there is no edge between vertex i and vertex j

A[0][1] = 1, there is an edge between vertex 0 and vertex 1.

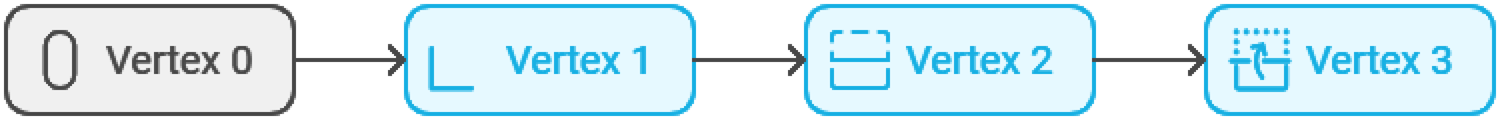
A[1][0] = 1, there is an edge between vertex 1 and vertex 0.

A[1][2] = 1, there is an edge between vertex 1 and vertex 2.

A[2][1] = 1, there is an edge between vertex 2 and vertex 1.

A[2][3] = 1, there is an edge between vertex 2 and vertex 3.

A[3][2] = 1, there is an edge between vertex 3 and vertex 2.

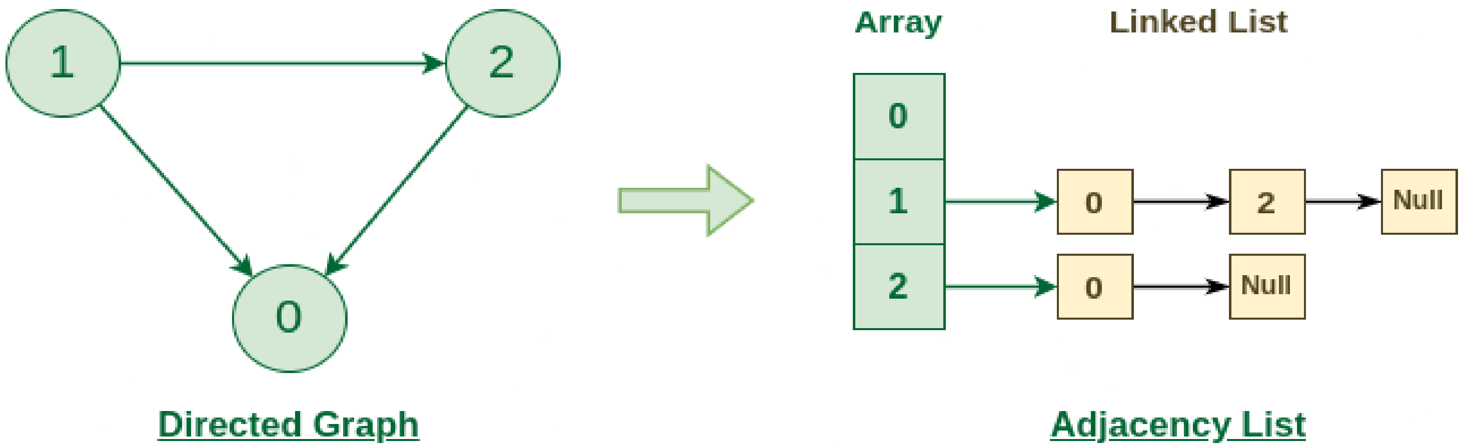


I. Definition and representation

d) Adjacency List

I. Definition and representation

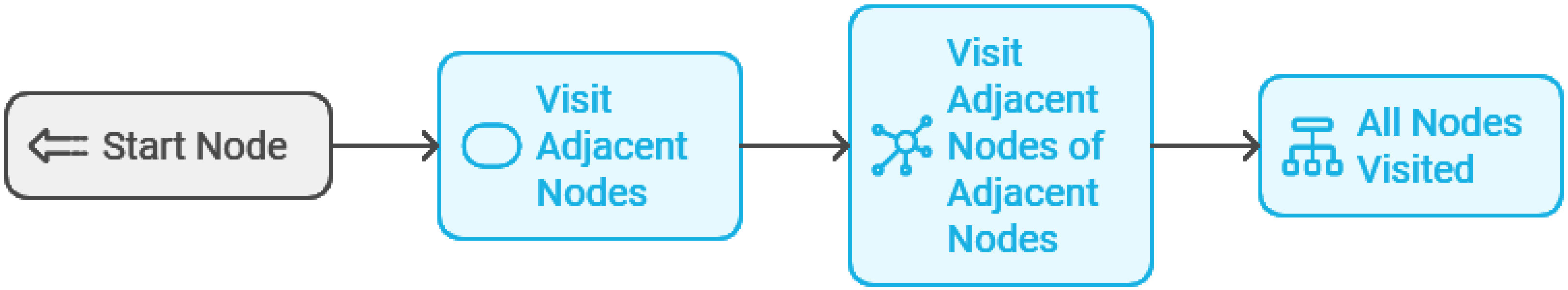
d) Adjacency List



Graph Representation of Directed graph to Adjacency List

II. Graph Traversal Algorithm

a) Breadth First Search or BFS for a Graph

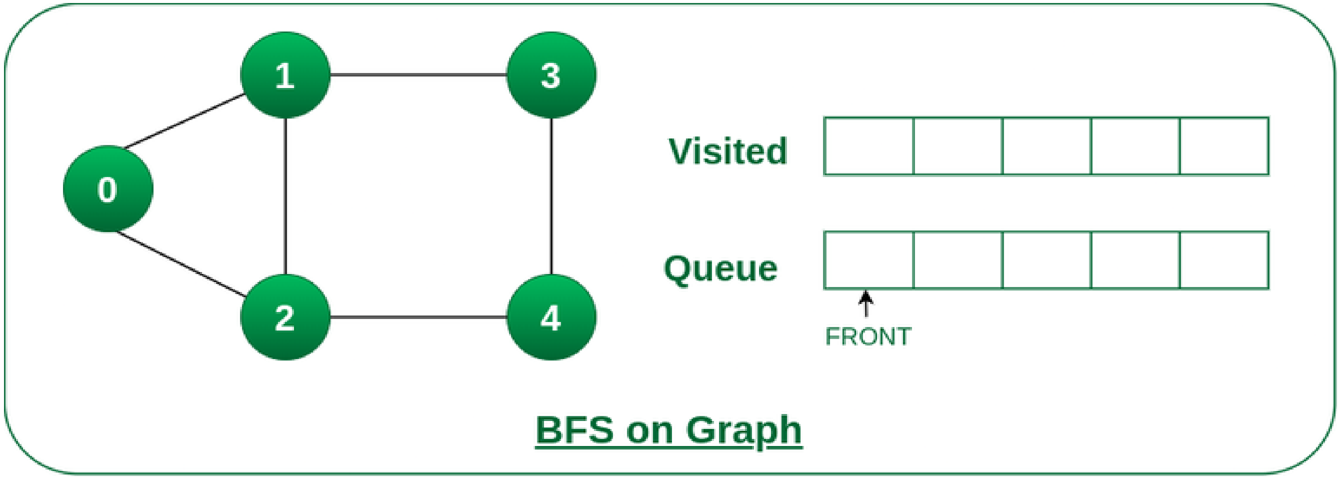


b) How Does the BFS Algorithm Work?

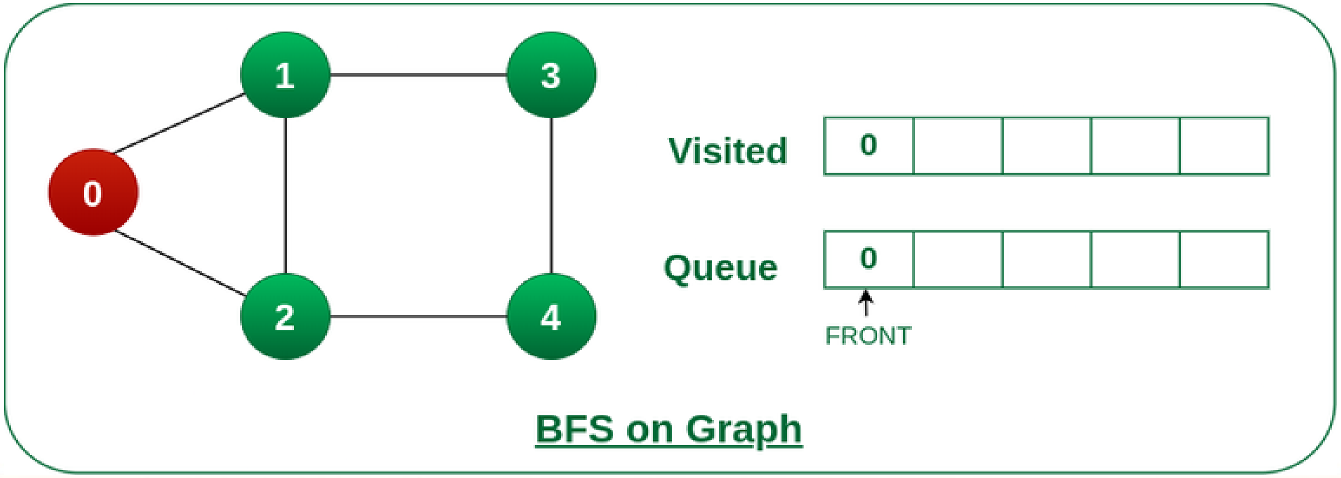
Start at Source Vertex 0 → Visit Adjacent Vertices of 0 → Move to Vertex 1 → Visit Adjacent Vertices of 1 → Move to Vertex 2 → Visit Adjacent Vertices of 2

II. Graph Traversal Algorithm

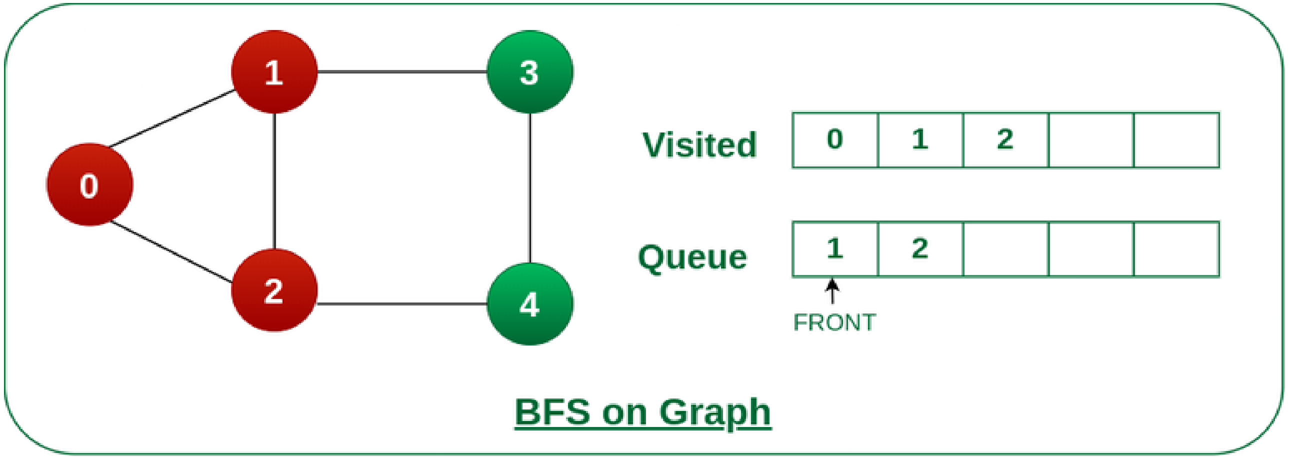
Step 1: Initially queue and visited arrays are empty.



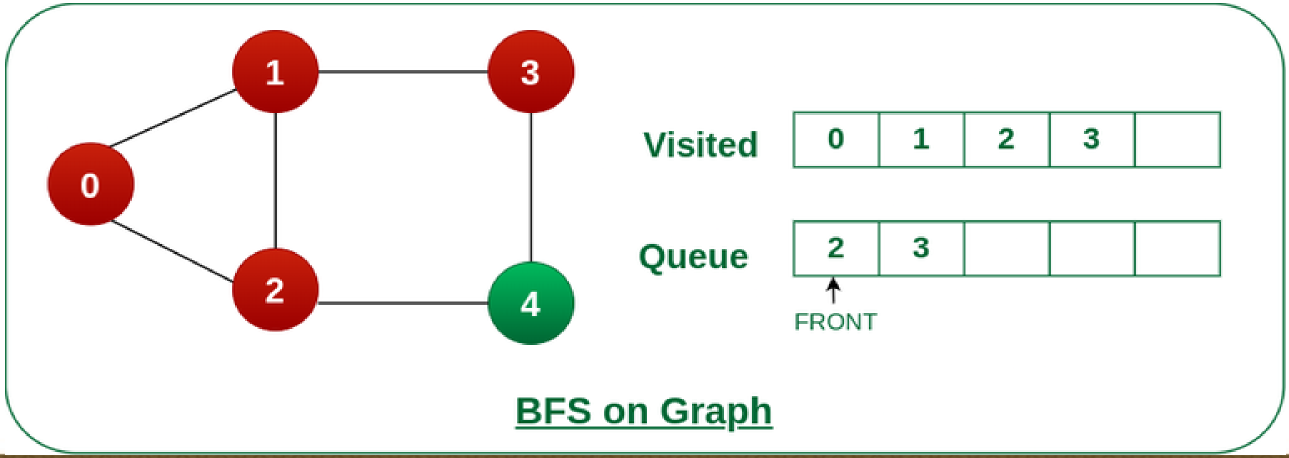
Step 2: Push 0 into queue and mark it visited.



Step 3: Remove 0 from the front of queue and visit the unvisited neighbors and push them into queue.

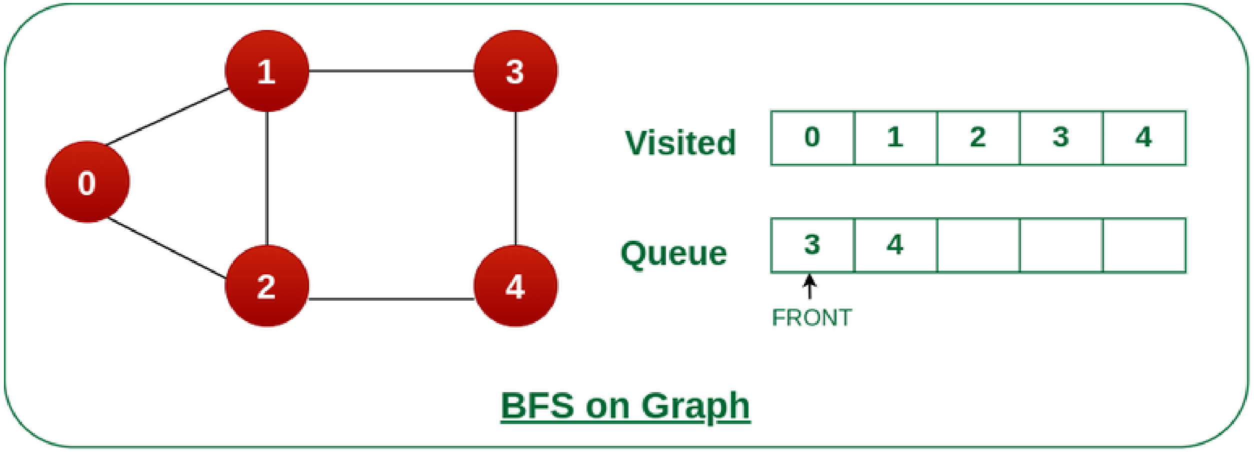


Step 4: Remove node 1 from the front of queue and visit the unvisited neighbors and push them into queue.

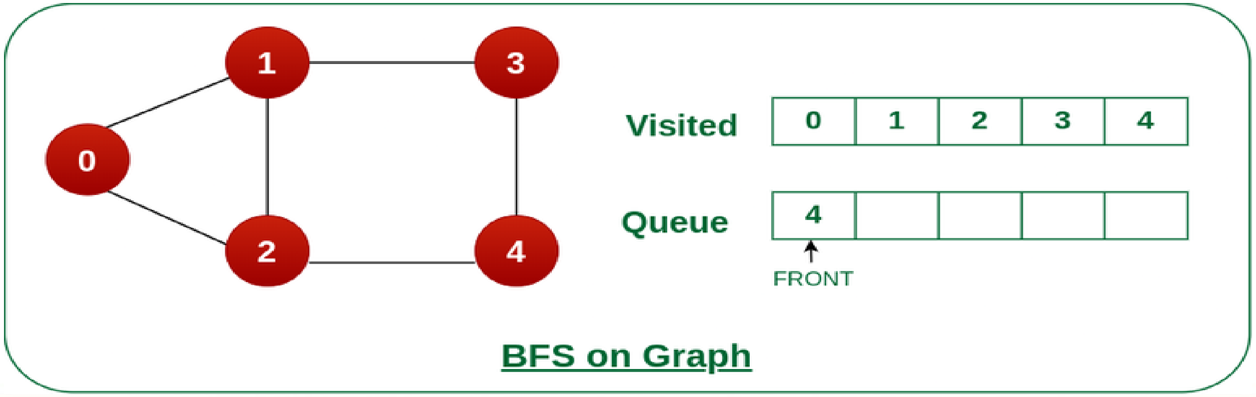


II. Graph Traversal Algorithm

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbors and push them into queue.

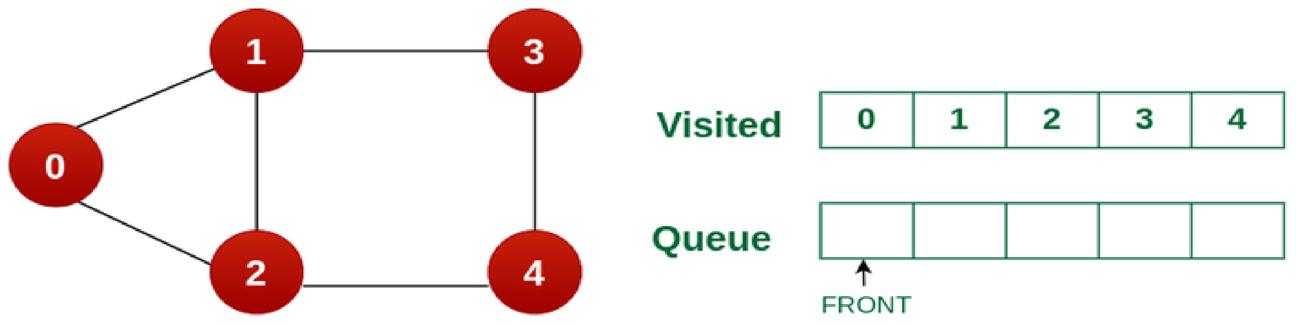


Step 6: Remove node 3 from the front of queue and visit the unvisited neighbors and push them into queue. As we can see that every neighbors of node 3 is visited, so move to the next node that are in the front of the queue.



II. Graph Traversal Algorithm

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbors and push them into queue. As we can see that every neighbor of node 4 are visited, move to the next.



BFS on Graph

II. Graph Traversal Algorithm

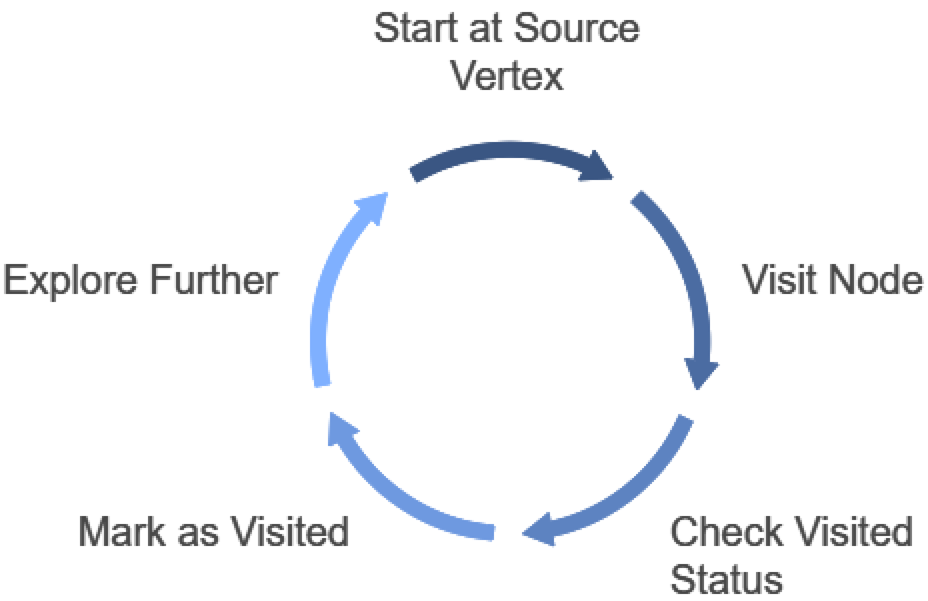
```
using System;
using System.Collections.Generic;
class GfG {
   // BFS from given source s
   static void Bfs(List<List<int>> adj, int s, bool[]
visited)
   {  // Create a queue for BFS
      Queue<int> q = new Queue<int>();
   // Mark the source node as visited and enqueue it
      visited[s] = true;
      q.Enqueue(s);
      // Iterate over the queue
      while (q.Count > 0) {
         // Dequeue a vertex from queue and print it
         int curr = q.Dequeue();
         Console.Write(curr + " ");
  // Get all adjacent vertices of the dequeued vertex
         // If an adjacent has not been visited,
         // mark it visited and enqueue it
         foreach (int x in adj[curr]) {
            if (!visited[x]) {
               visited[x] = true;
               q.Enqueue(x); }   }  } }
static void AddEdge(List<List<int>> adj, int u, int v)
   {   adj[u].Add(v);
       adj[v].Add(u);
   }
 public static void Main(string[] args) {
       // Number of vertices in the graph
       int V = 5;
```

II. Graph Traversal Algorithm

```
// Adjacency list representation of the graph
    List<List<int>> adj = new List<List<int>>(V);
    for (int i = 0; i < V; i++) {
      adj.Add(new List<int>());
    } // Add edges to the graph
    AddEdge(adj, 0, 1);     AddEdge(adj, 0, 2);
    AddEdge(adj, 1, 3);     AddEdge(adj, 1, 4);
    AddEdge(adj, 2, 4);
    // Mark all the vertices as not visited
    bool[] visited = new bool[V];
    // Perform BFS traversal starting from vertex 0
    Console.WriteLine("BFS starting from 0 : ");
    Bfs(adj, 0, visited);   }
}
```

II. Graph Traversal Algorithm

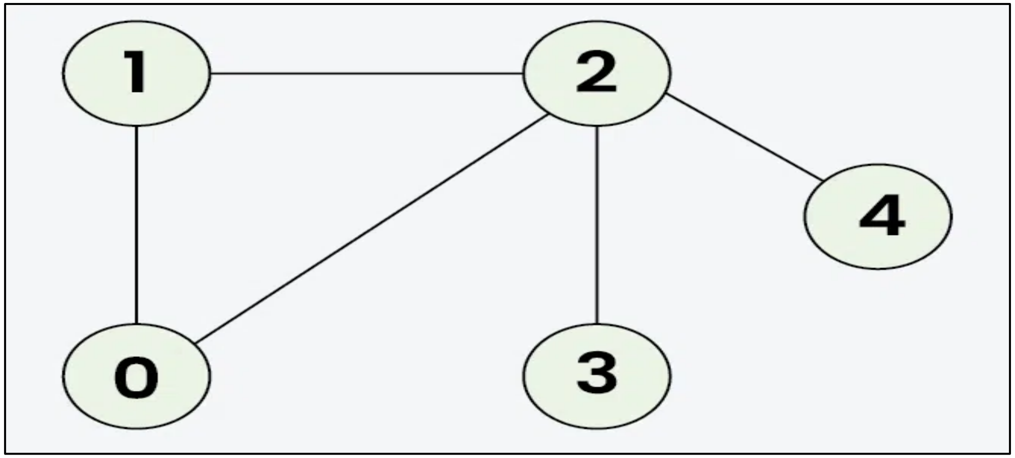
b) Depth First Search or DFS for a Graph



II. Graph Traversal Algorithm

b) Depth First Search or DFS for a Graph

Input: V = 5, E = 5, edges = {{1, 2}, {1, 0}, {0, 2}, {2, 3}, {2, 4}}, source = 1



Output: 1 2 0 3 4

II. Graph Traversal Algorithm

```
using System;
using System.Collections.Generic;
class GfG{
   static void AddEdge(List<List<int>> adj, int s, int t){
      adj[s].Add(t);
      adj[t].Add(s);   }
   // Recursive function for DFS traversal
   static void DFSRec(List<List<int>> adj, bool[]
visited, int s){
      // Mark the current vertex as visited
        visited[s] = true;
       // Print the current vertex
       Console.Write(s + " ");
       // Recursively visit all adjacent vertices
        // that are not visited yet
        foreach (int i in adj[s]){
         if (!visited[i]){
            DFSRec(adj, visited, i); }  }
   }
// Main DFS function that initializes the visited array
   static void PerformDFS(List<List<int>> adj, int s){
      bool[] visited = new bool[adj.Count];
      // Call the recursive DFS function
      DFSRec(adj, visited, s);
   }
```

II. Graph Traversal Algorithm

```
static void Main(){
        int V = 5;
        // Create an adjacency list for the graph
        List<List<int>> adj = new List<List<int>>(V);
        for (int i = 0; i < V; i++){
            adj.Add(new List<int>());
        }
        // Define the edges of the graph
        int[,] edges = {
            { 1, 2 }, { 1, 0 }, { 2, 0 }, { 2, 3 }, { 2, 4 }
        };
```

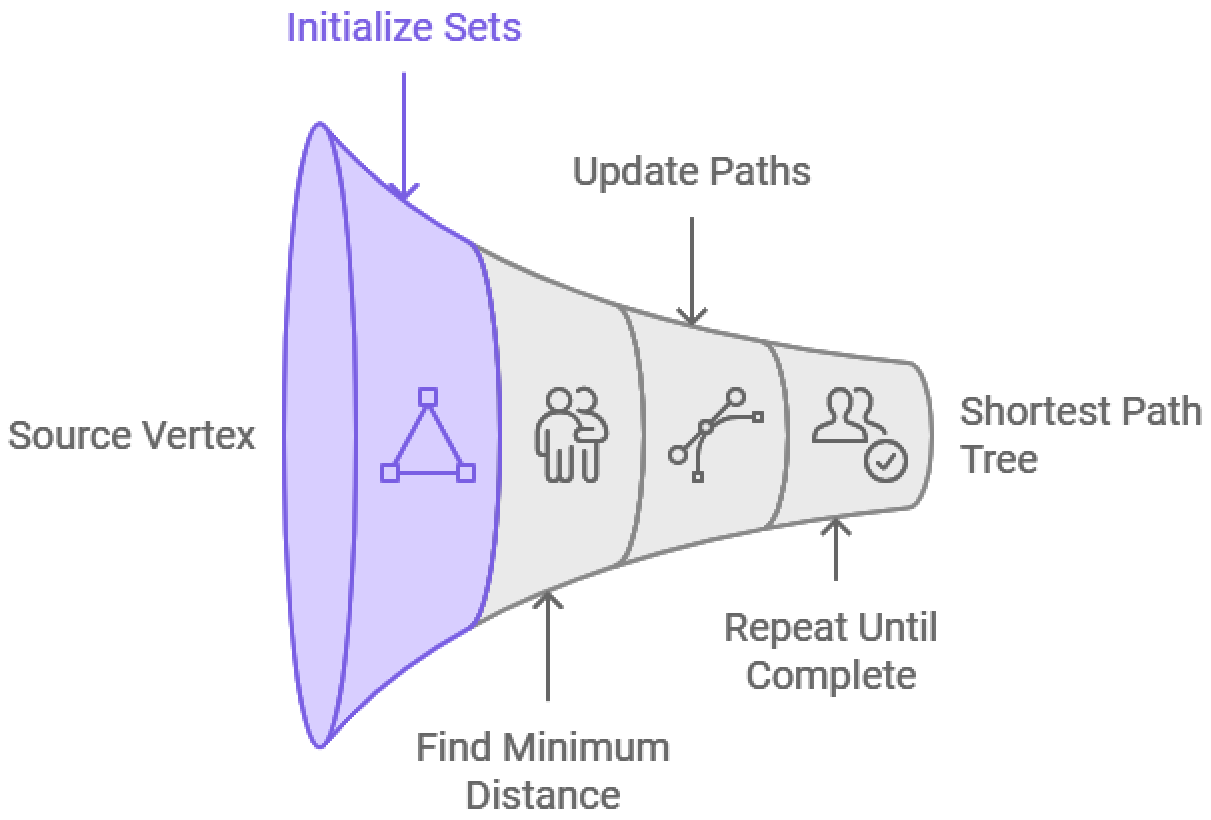
```
// Populate the adjacency list with edges
        for (int i = 0; i < edges.GetLength(0); i++){
            AddEdge(adj, edges[i, 0], edges[i, 1]);
        }
        int source = 1; // Starting vertex for DFS
        Console.WriteLine("DFS from source: " +
source);
        PerformDFS(adj, source);  }
}
```

Output

DFS from source: 1 1 2 0 3 4

III. Shortest Path Algorithm

a) Dijkstra Algorithm



III. Shortest Path Algorithm

a) Dijkstra Algorithm

Mark Source Node

Initialize the source node with a distance of 0 and others with infinity.

Select Current Node

Choose the non-visited node with the smallest current distance as the current node.

Update Neighbor Distances

For each neighbor, update the distance if a shorter path is found.

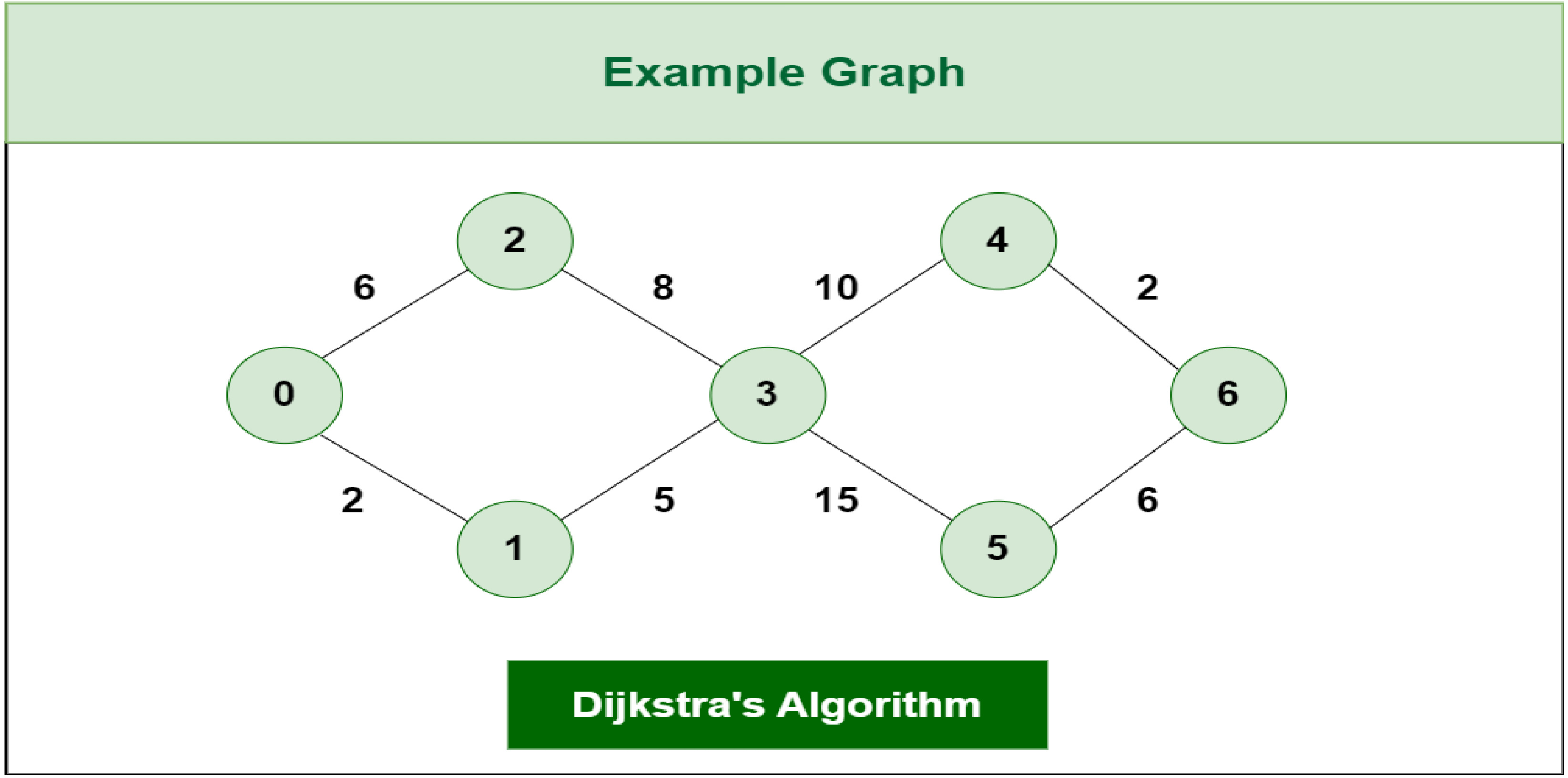
Mark Node as Visited

Mark the current node as visited to avoid revisiting.

Check Unvisited Nodes

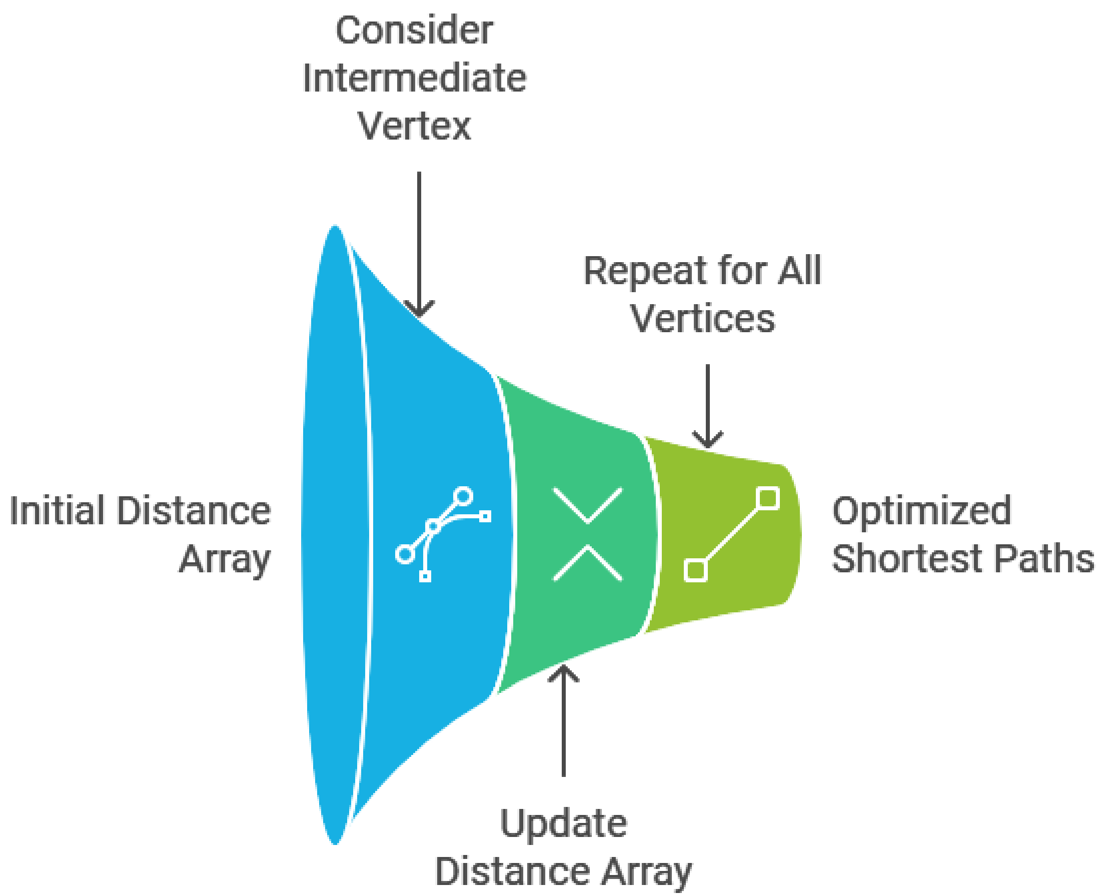
Check if there are any unvisited nodes remaining.

III. Shortest Path Algorithm



III. Shortest Path Algorithm

b) Floyd Warshall Algorithm

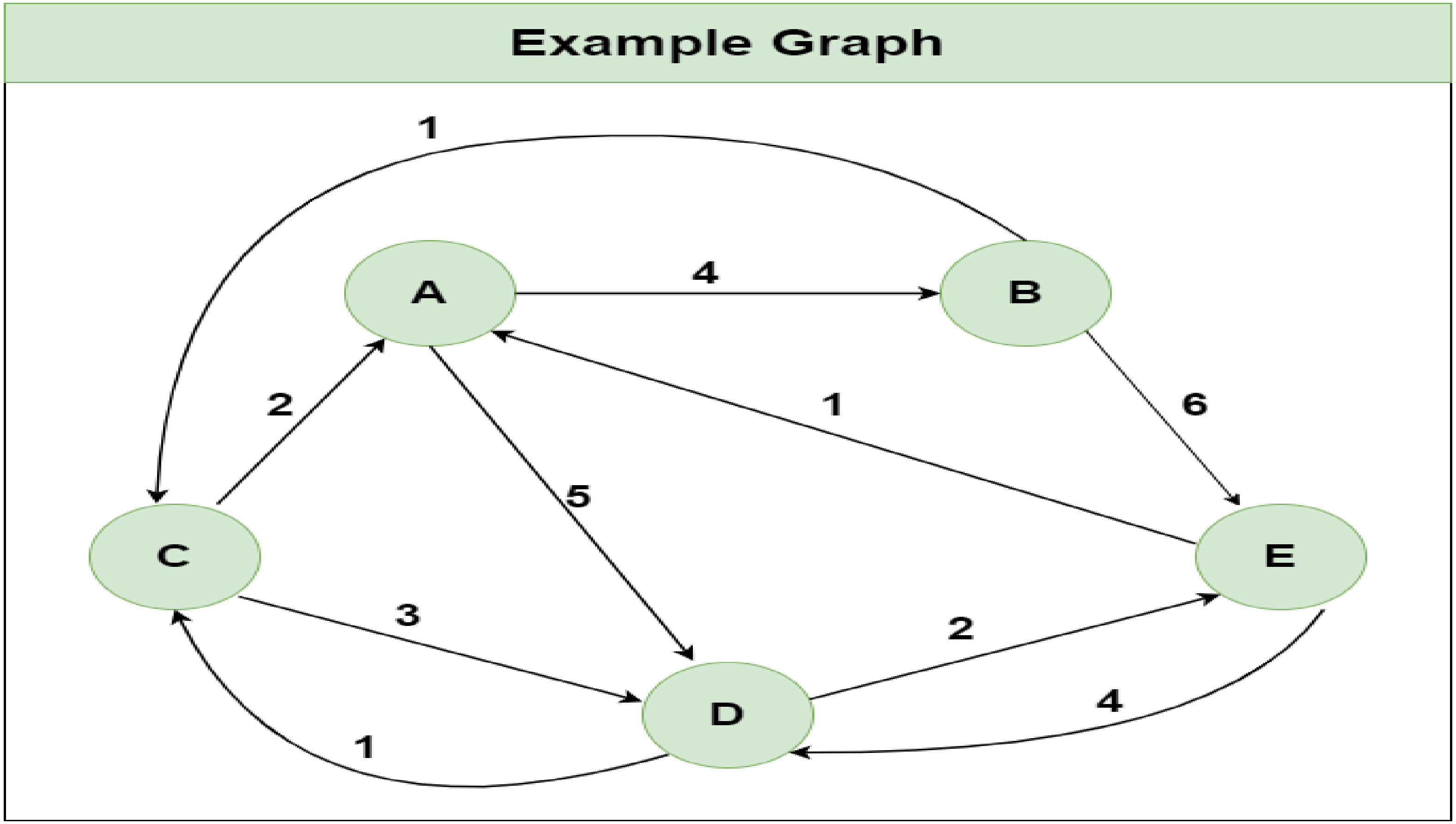


III. Shortest Path Algorithm

b) Floyd Warshall Algorithm

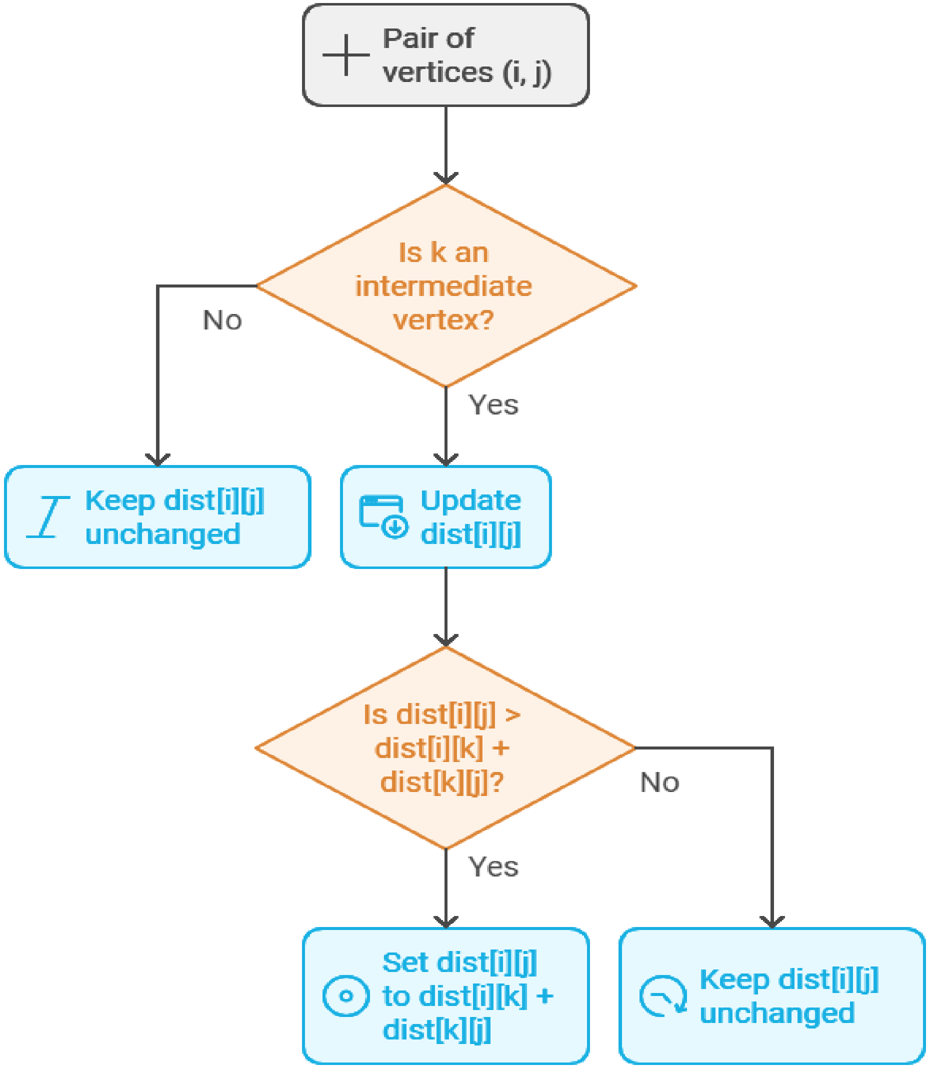
III. Shortest Path Algorithm

b) Floyd Warshall Algorithm



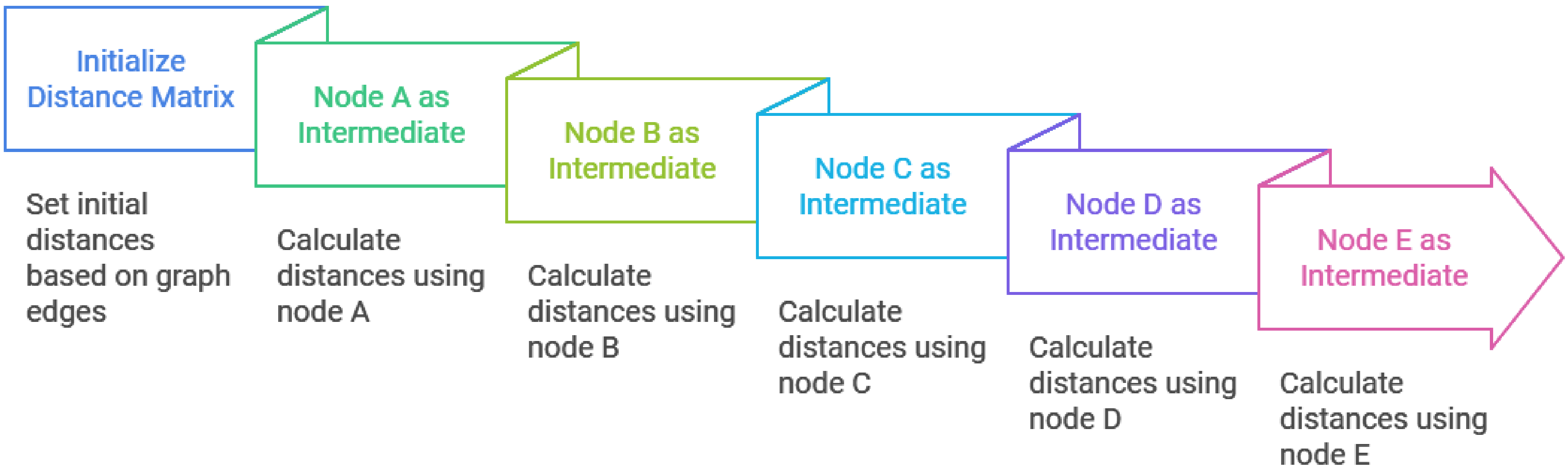
III. Shortest Path Algorithm

b) Floyd Warshall Algorithm



III. Shortest Path Algorithm

b) Floyd Warshall Algorithm



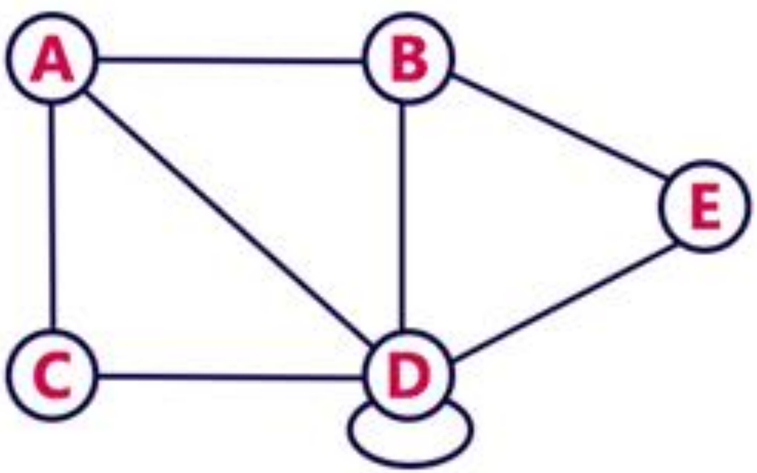
Quizzes

1. What is the maximum number of children that a binary tree node can have?
2. What is a leaf node? What is a root node?
3. How do you find the lowest common ancestor (LCA) of a binary tree?
4. How do you check if a given binary tree is a subtree of another binary tree?
5. How do you find the distance between two nodes in a binary tree?
6. The following given tree is an example for?

```
/* Construct the following tree
          1
        /   \
       2     3
      /     / \
     4     5   6
          / \
         7   8
```

Quizzes

7. A binary tree is a rooted tree but not an ordered tree.
8. What is the traversal strategy used in the binary tree?
9. How many common operations are performed in a binary tree?
10. What is a Graph?
11. What are some common Types and Categories of Graphs?



$$
\begin{array}{c|ccccc|}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 1 \\
C & 1 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0
\end{array}
$$

Quizzes

8. What is the difference between a Tree and a Graph?
9. Compare the adjacency List and adjacency matrixes
10. How can you determine the Min number of edges for a graph to remain connected?
11. Explain the Breadth-First Search (BFS) and Breadth-First Search (DFS) traversing
12. What are the differences between BFS and DFS?


THANK
YOU!