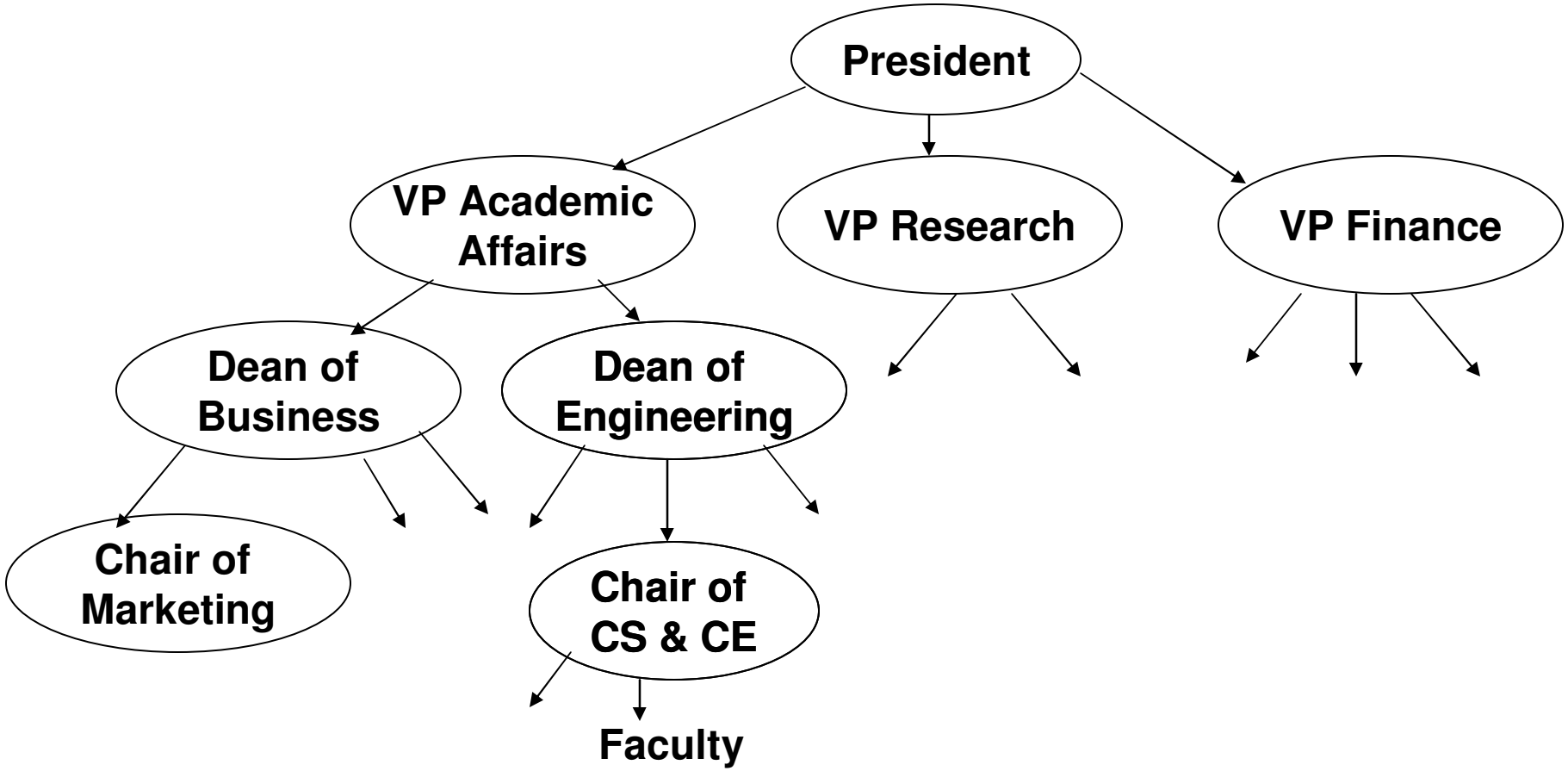
Module 6:
Binary Trees

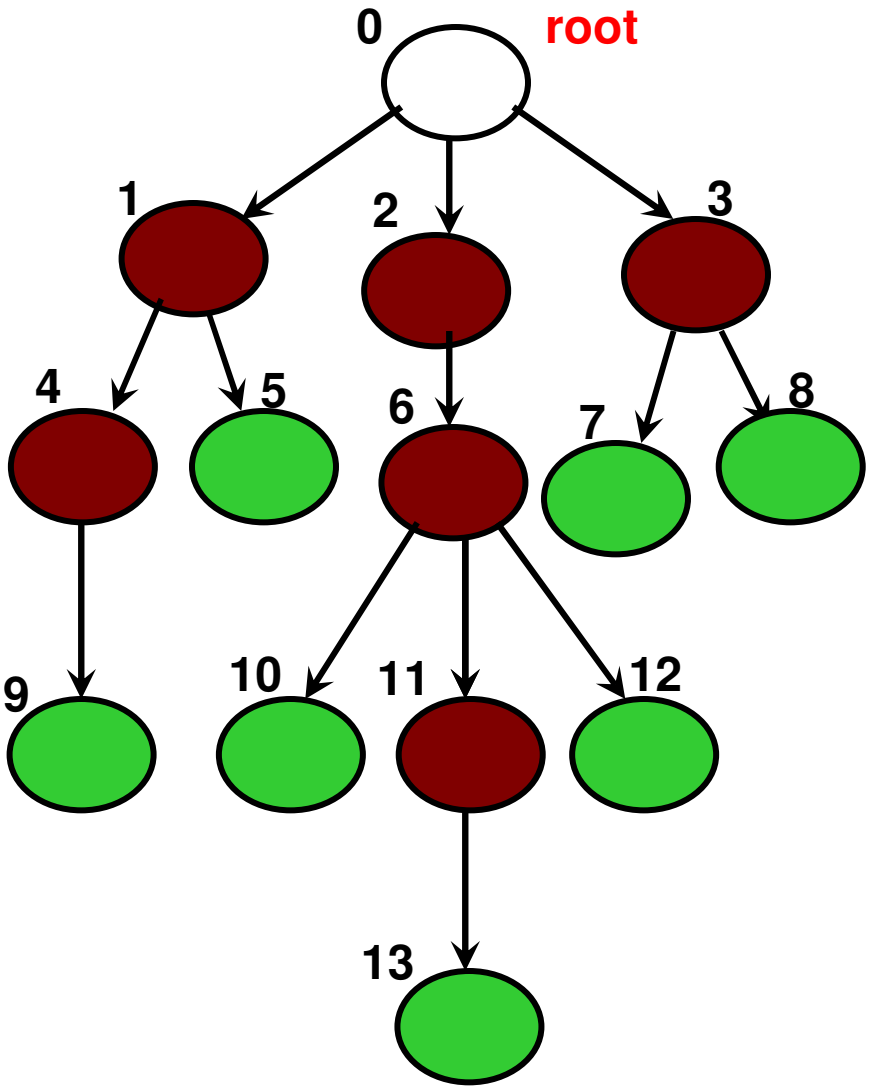
Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Tree

- All the data structures we have seen so far can store only linear data
- Trees are used to store hierarchical data
  - Example: Employees in an organization, file system, network routing



Tree: Nomenclature



Each entity in a tree is called "node or vertex"

The "root" node is at the top of the tree.

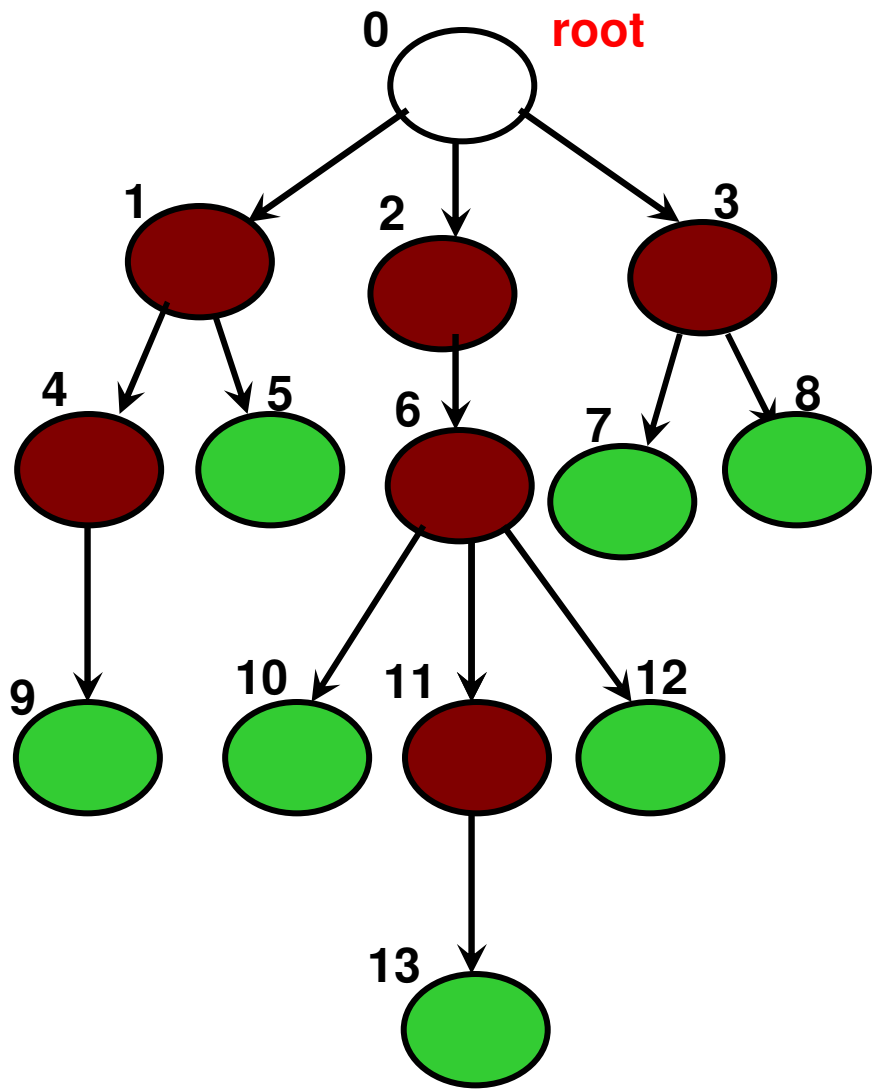
A node could have one or more "child" nodes that are its immediate descendants. A node without any child node is called a "leaf" node. A node with one or more child nodes is called an "internal node".

The "edges or links" are in only direction, from the "parent" to a "child".

Each node (except the root node) in the tree has exactly one parent.

In a tree of 'N' nodes, there will be exactly 'N-1' links.

Tree: Nomenclature



The "depth" (or equivalently the "level number") of a node is the number of edges on the "path" from the root node.

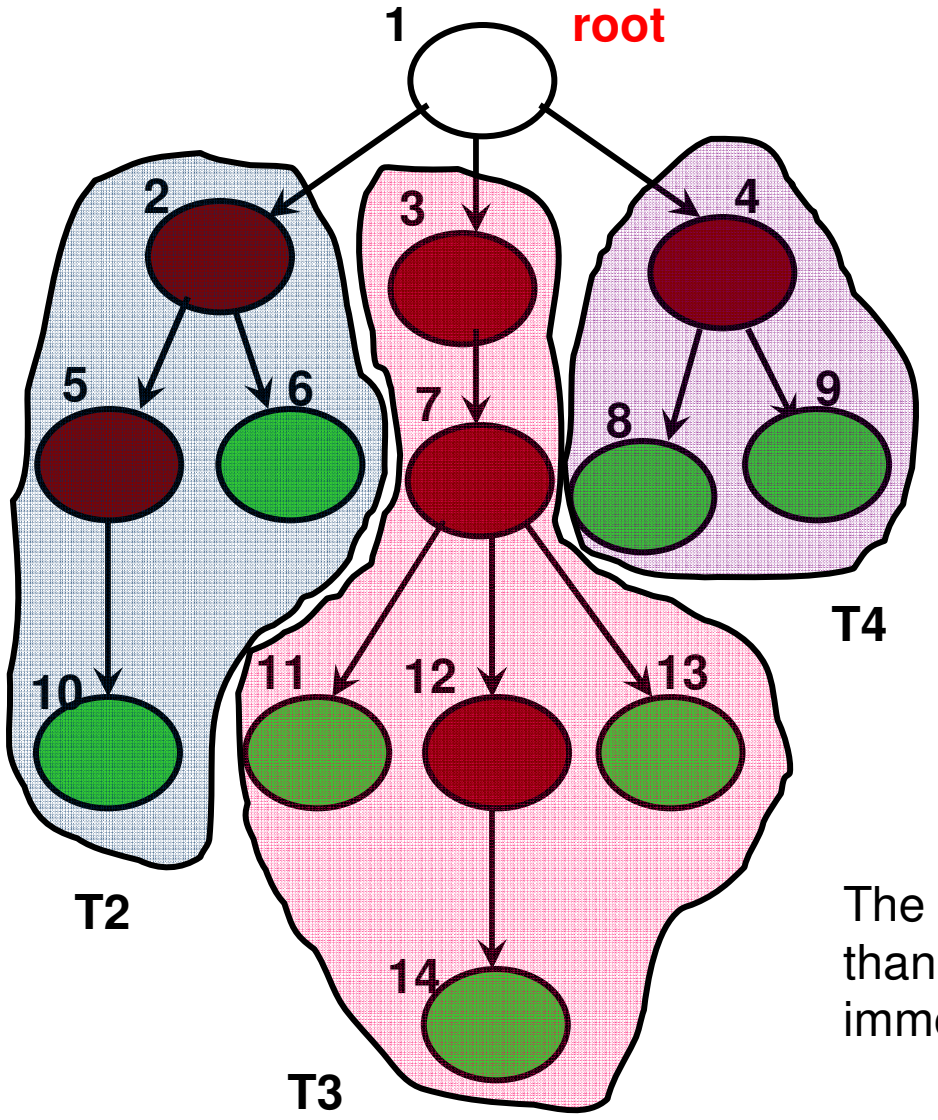
The depth of nodes 10 and 13 are 3 and 4 respectively.

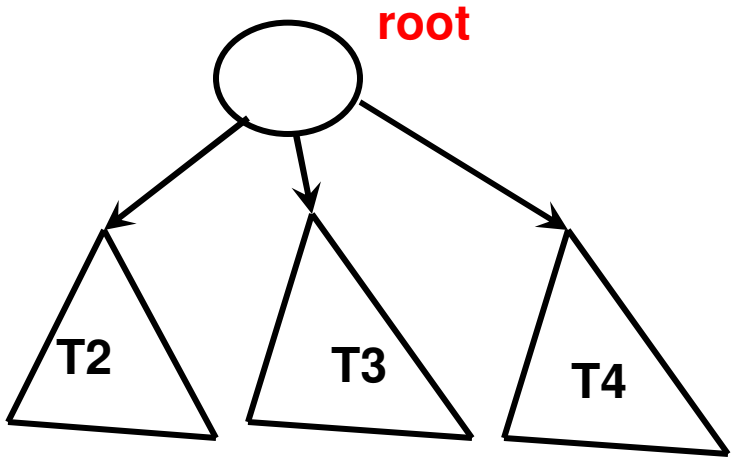
The depth of the root is 0.
The "height" of a node is the number of edges on the longest path from the node to a leaf node.

The height of the root node is 4 as the longest path to a leaf node (node 13) is 4 edges long.
The height of a tree is the height of its root node.

The height of a leaf node is 0.

Tree: Recursive Structure



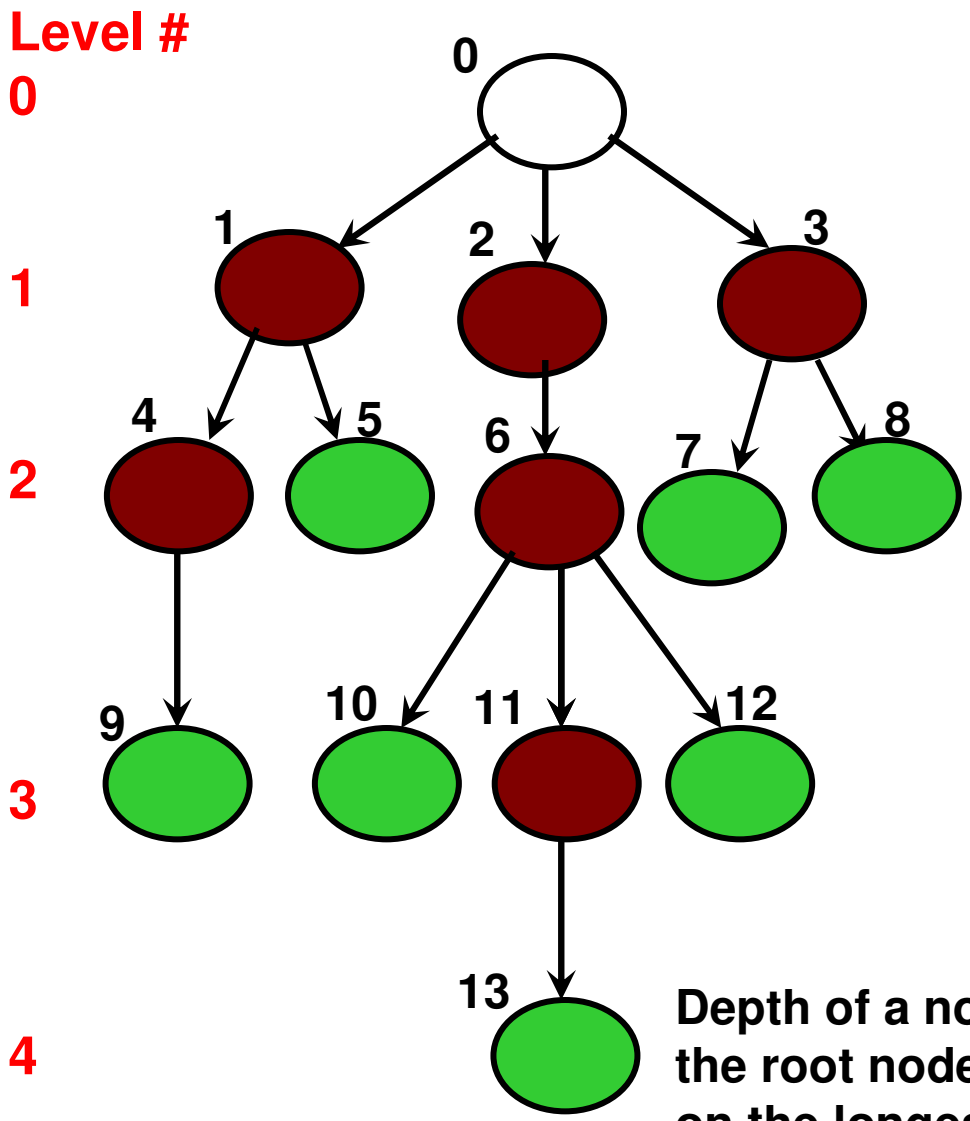


The sub trees have a similar Recursive structure

Node 2 is the root of sub tree T2
Nodes 3 and 4 are the roots of Sub trees T3 and T4.

The height of a node in the tree is one more than the maximum of the heights of its immediate child nodes.

Example: Height vs. Depth

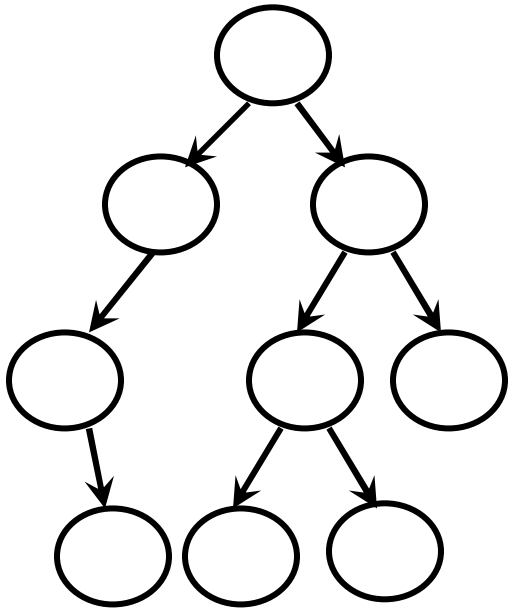


Node	Depth (Level)	Height
0	0	4
1	1	2
2	1	3
3	1	1
4	2	1
5	2	0
6	2	2
7	2	0
8	2	0
9	3	0
10	3	0
11	3	1
12	3	0
13	4	0

Depth of a node is the # edges on the path from the root node. The height of a node is the # edges on the longest path from the node to a leaf node.

Binary Tree

- Binary Tree: A tree with at most two child nodes for each internal node
- Nodes that have the same depth are said to be at the same "level".
  - The root node is at level 0.
  - The leaf node(s) with the largest depth are said to be at the bottommost level.
- Maximum number of nodes at a particular level 'i' ($i \geq 0$) for a binary tree is 2^i.
- The number of nodes at a particular level is at most twice the number of nodes in the previous level.



Level	# Nodes (Max)
0	1 (1)
1	2 (2)
2	3 (4)
3	3 (8)

The "height" of a binary tree (# levels – 1) corresponds to the "max. depth" for any node in the tree.

Maximum number of nodes in a Binary tree of height "h" is

$$2^0 + 2^1 + 2^2 + \ldots + 2^h = (2^{h+1}) - 1$$
$$= (2^{\#\text{levels}}) - 1$$

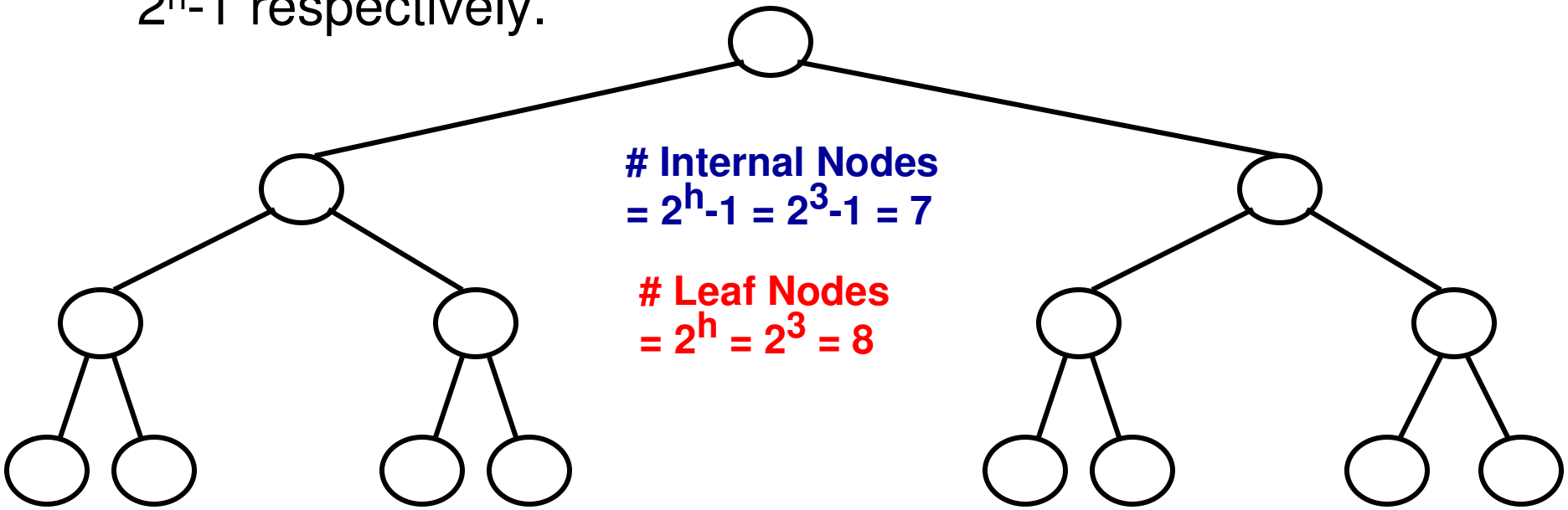
Examples

- Consider a binary tree of height 4. Determine the maximum possible number of nodes in the binary tree.
  - Answer: $2^{h+1} - 1 = 2^{(4+1)} - 1 = 31$.

- Consider a binary tree of 30 nodes. What is the minimum possible height of the binary tree?
  - Min Height (h) $= \lceil \log_2(N+1) - 1 \rceil = \lceil \log_2(30+1) - 1 \rceil = 4$

Complete Binary Tree

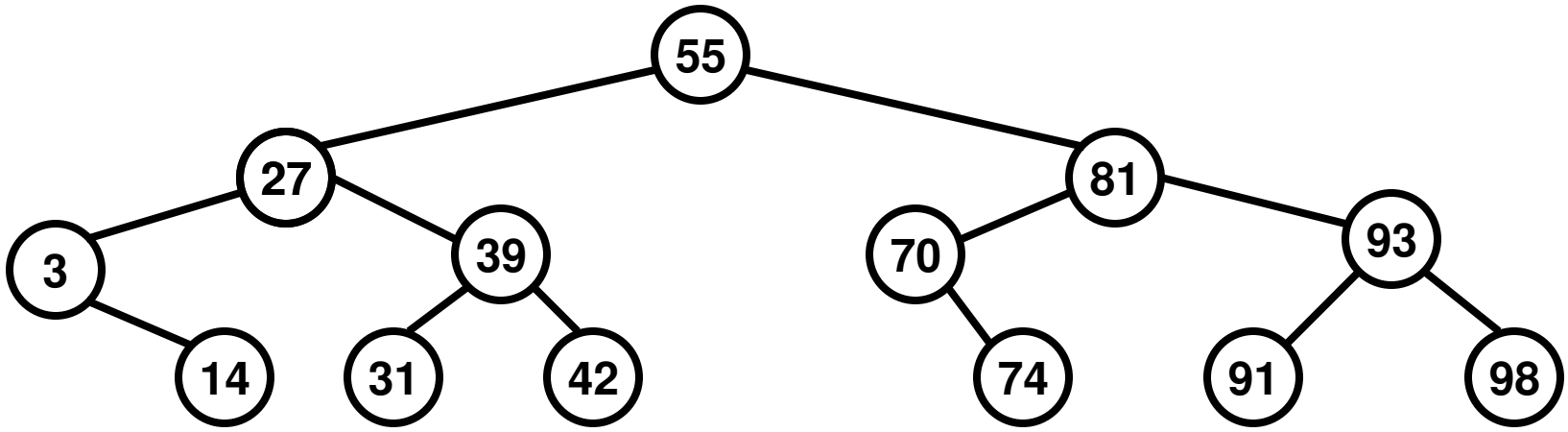
- A binary tree is called a complete binary tree if each internal node has exactly two child nodes.
  - In other words, all the lef nodes of a complete binary tree are at depth (level) ‘h’, which is also the height of the tree.
- The number of leaf nodes and internal nodes (incl. root node) of a complete binary tree of height ‘h’ are 2^h and 2^h-1 respectively.



Total # Nodes in a Complete Binary Tree of height ‘h’ = $2^{h+1}-1 = 2^{3+1}-1 = 15$

Binary Search Tree

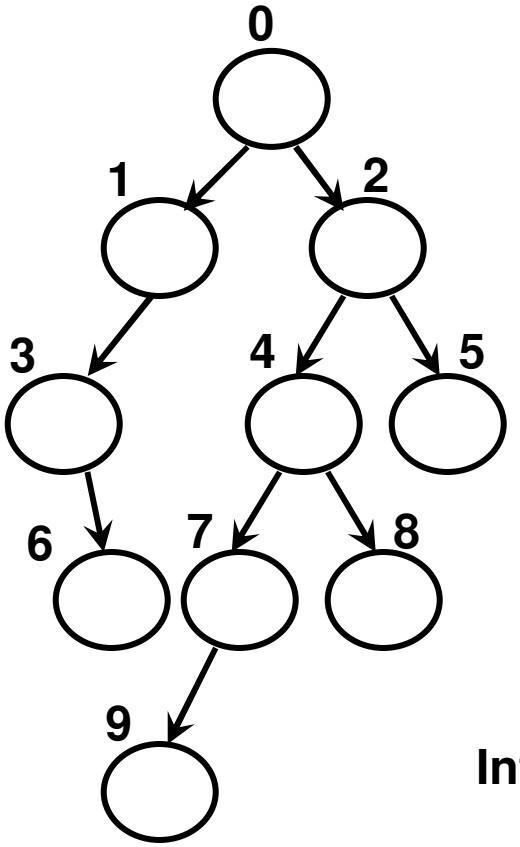
- A binary tree is called a binary search tree if for every internal node: the data at the internal node is greater than or equal to that of its left child node and lower than or equal to that of its right child node.



Example for Reading a File

Code 6.1

Binary Tree



Representation as a Text file (binaryTreeFile_1.txt)

Internal Nodes

```
0: 1, 2
1: 3, -1
2: 4, 5
3: -1, 6
4: 7, 8
7: 9, -1
```

Immediate Downstream Nodes

Format

Internal Node: left child node id, right child node id

If a left (or right) child node is not there, then the id field is represented as -1.
We show a node and its child nodes in the file only if the node has at least one child node (i.e., the node is an internal node)

Reading from a File (C++: Code 6.1)

```
#include <iostream>   // for doing file Input/Output (File I/O)
#include <fstream>
#include <string>
#include <cstring>  // for string tokenizing and c-style string processing
using namespace std;

int main(){

        string filename;
        cout << "Enter a file name: ";
        cin >> filename;

        ifstream fileReader(filename);  // ifstream is the name of the class to declare
                                        // objects for file reading (input file stream)

        if (!fileReader){                       // if fileReader is not pointing to a file
                cout << "File cannot be opened!! ";  // !false
                return 0;
        }

        int numCharsPerLine = 10;

        char *line = new char[numCharsPerLine];
        // '10' is the maximum number of characters per line

        fileReader.getline(line, numCharsPerLine, '\n');
        // '\n' is the delimiting character to stop reading the line
```

```
        while (fileReader){

                cout << "Line Read: " << line << endl;

                // Extracting the integers/node ids using string tokenizer
                char* cptr = strtok(line, ",: ");   // Declare a string tokenizer and
                                                    // declare a character pointer to
                cout << "Extracted node ids: ";     // the token extracted
                while (cptr != 0){

                        string token(cptr);
                        int nodeid = stoi(token);
                        cout << nodeid << " ";

                                                    // Extract the next token
                        cptr = strtok(NULL, ",: "); // until a NULL (end of line) is seen
                }

                cout << endl << endl;

                fileReader.getline(line, numCharsPerLine, '\n');
                                // Read the next file from the file using the fileReader
        }

        return 0;
}
```

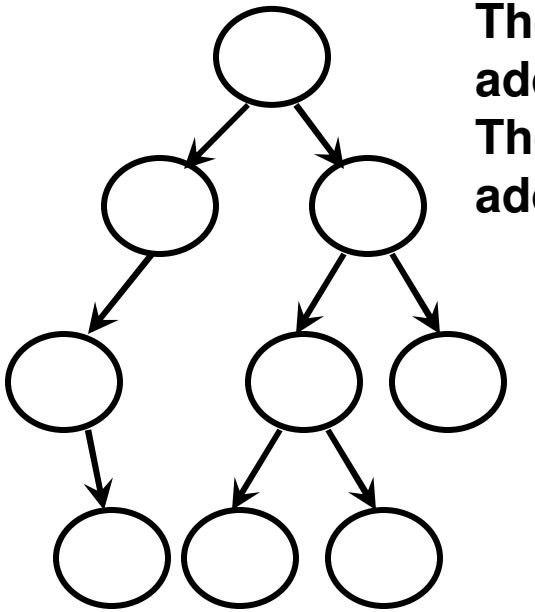
Binary Tree

- A binary tree is a tree wherein each node has at most two child nodes.

Node class (BTNode) for Binary Tree

C++

```
int nodeid;
int data;
int levelNum;
BTNode* leftChildPtr;
BTNode* rightChildPtr;
```



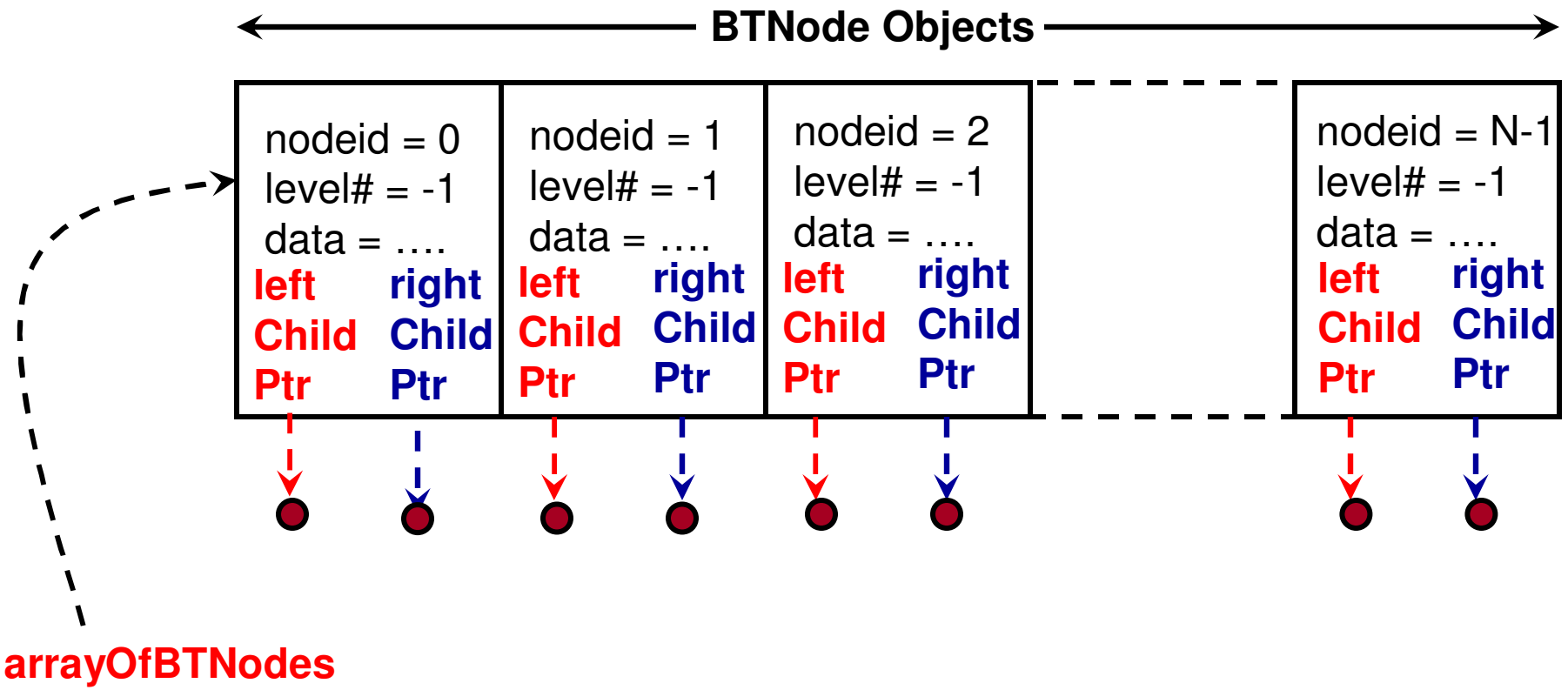
The leftChildPtr has the address of the left node. The rightChildPtr has the address of the right node.

The leftChildPtr or the rightChildPtr are set to null if a node does not have a left child or right child.

Strategy to Create a Binary Tree

- Input the number of nodes (numNodes or N) in the binary tree from the user.
- **Step 1:** Create an array of BTNodes (of size numNodes) and have a pointer/reference of class BTNode (arrayOfBTNodes) to point to it.
  - Initialize the BTNodes in arrayOfBTNodes
    - The id of the BTNode is set to the appropriate value
    - The levelNum of the BTNode is set to -1
    - The leftChildPtr and rightChildPtr of the BTNode are set to null.
    - We do not worry about initializing the data unless an application needs to make use of this variable.
  - **Step 2:** Read the text file containing the info for the binary tree
  - For every internal node read, extract the id of the left child and right child nodes from the line
    - If a child node id is not -1, then call the function to set up the node as the left child or right child, depending on the case.
    - In the setLeftChild(int upstreamnode id, int downstreamnodeid) and the setRightChild(int upstreamnodeid, int downstreamnodeid) functions, pass the pointers to the appropriate downstream node objects and set them as left child or right child, depending on the case.

Step 1: Create an Array of BTNodes and Initialize each BTNode

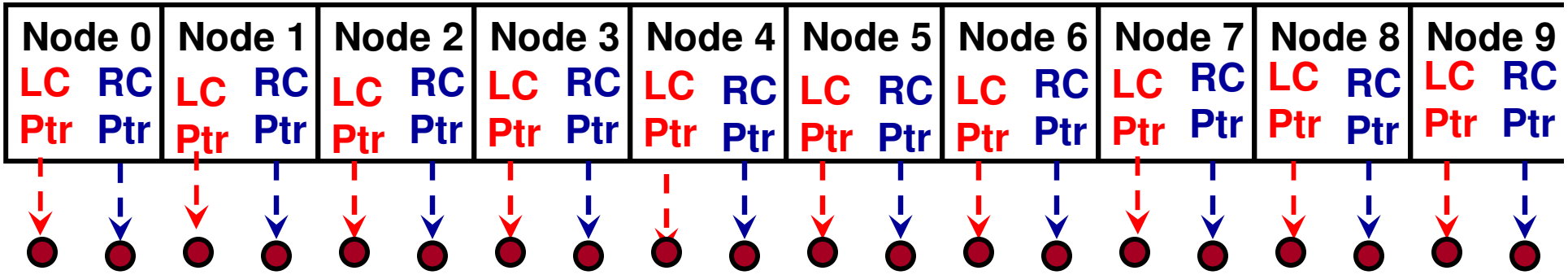


Step 2: Read the File and Set up the Pointers to the Child Nodes

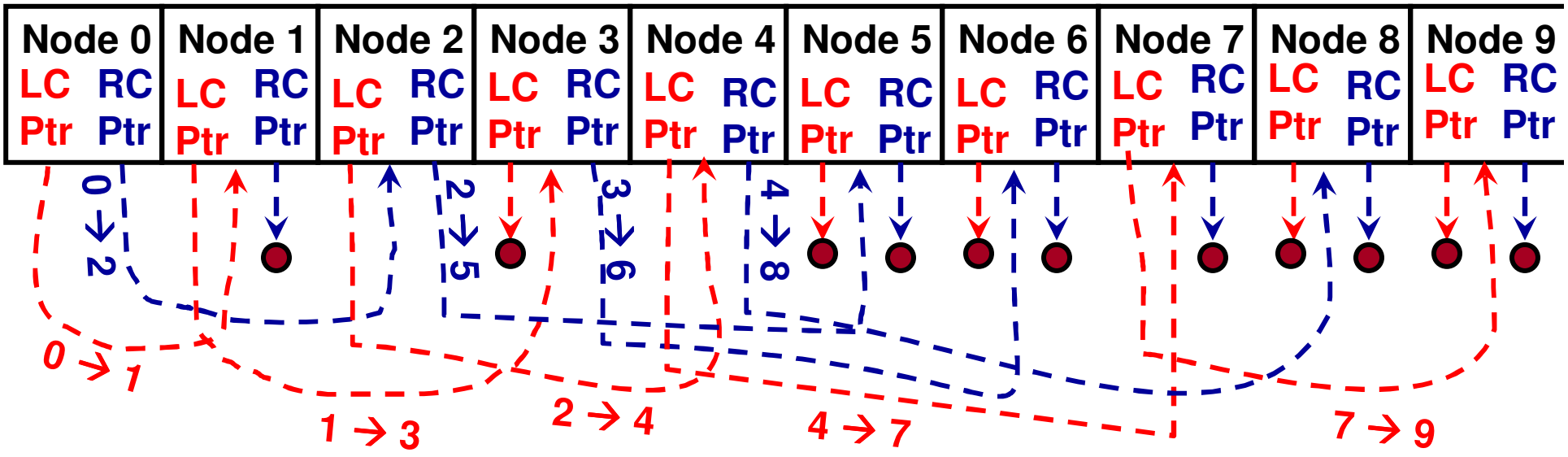
binaryTreeFile_1.txt

```
0: 1, 2
1: 3, -1
2: 4, 5
3: -1, 6
4: 7, 8
7: 9, -1
```

After Step 1; but Before Step 2

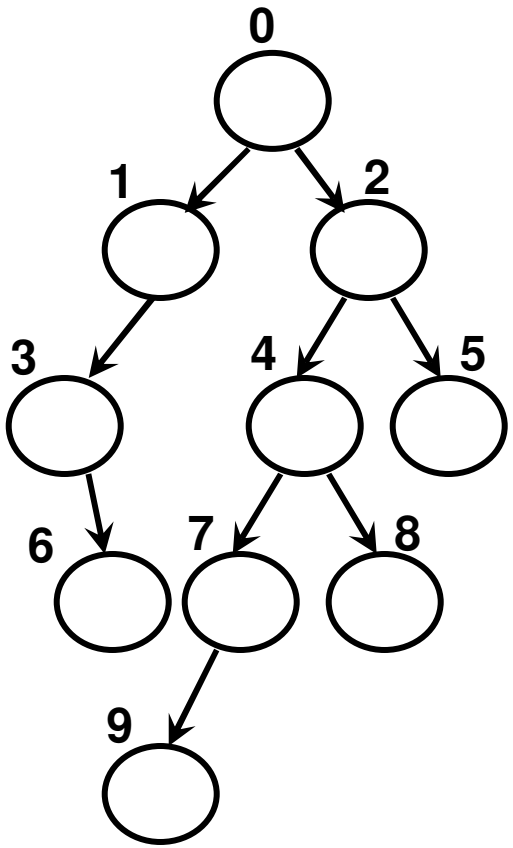


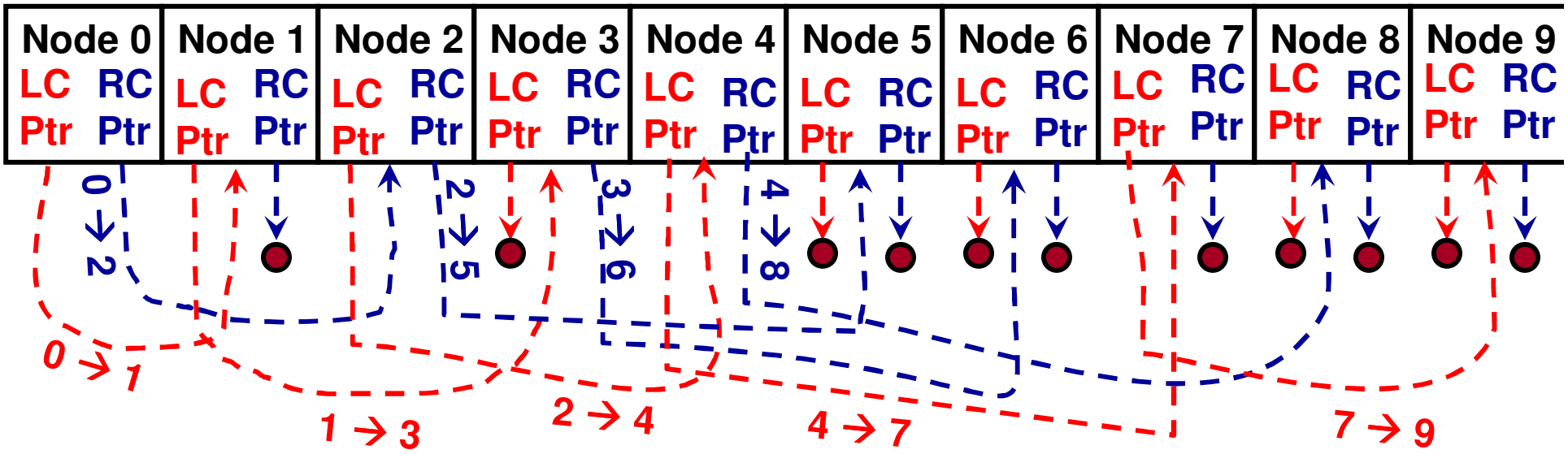
After Step 2



Example Binary Tree

binaryTreeFile_1.txt





Binary Tree Implementation (C++: Code 6.2): Class BTNode

Private member variables

```
int nodeid;
int data;
int levelNum;
BTNode* leftChildPtr;
BTNode* rightChildPtr;
```

Public Member Functions/ Empty Constructor

```
BTNode(){}

void setNodeId(int id){
        nodeid = id;
}

int getNodeId(){
        return nodeid;
}
```

```
BTNode* getLeftChildPtr(){
        return leftChildPtr;
}

BTNode* getRightChildPtr(){
        return rightChildPtr;
}

int getLeftChildID(){
        if (leftChildPtr == 0)
                return -1;

        return leftChildPtr->getNodeId();
}

int getRightChildID(){
        if (rightChildPtr == 0)
                return -1;

        return rightChildPtr->getNodeId();
}
```

```
void setLeftChildPtr(BTNode* ptr){
        leftChildPtr = ptr;
}

void setRightChildPtr(BTNode* ptr){
        rightChildPtr = ptr;
}

BTNode* getLeftChildPtr(){
        return leftChildPtr;
}

BTNode* getRightChildPtr(){
        return rightChildPtr;
}
```

```
void setLevelNum(int level){
        levelNum = level;
}

int getLevelNum(){
        return levelNum;
}
```

```
void setData(int d){
        data = d;
}

int getData(){
        return data;
}
```

Binary Tree Implementation (C++ Code 6.2): Class BinaryTree

Private member variables

```
int numNodes;
BTNode* arrayOfBTNodes;
```

Public Constructor

```
BinaryTree(int n){
        numNodes = n;
        arrayOfBTNodes = new BTNode[numNodes];

        for (int id = 0; id < numNodes; id++){
                arrayOfBTNodes[id].setNodeId(id);
                arrayOfBTNodes[id].setLevelNum(-1);
                arrayOfBTNodes[id].setLeftChildPtr(0);
                arrayOfBTNodes[id].setRightChildPtr(0);
        }
}
```

Public Member Functions to set the Left Link (left child node ptr) and Right Link (right child node ptr) for a BTNode

```
void setLeftLink(int upstreamNodeID, int downstreamNodeID){
      arrayOfBTNodes[upstreamNodeID].setLeftChildPtr(&arrayOfBTNodes[downstreamNodeID]);
}
```

```
void setRightLink(int upstreamNodeID, int downstreamNodeID){
      arrayOfBTNodes[upstreamNodeID].setRightChildPtr(&arrayOfBTNodes[downstreamNodeID]);
}
```

Binary Tree Implementation (C++ Code 6.2): Class BinaryTree

Regarding Leaf Nodes:

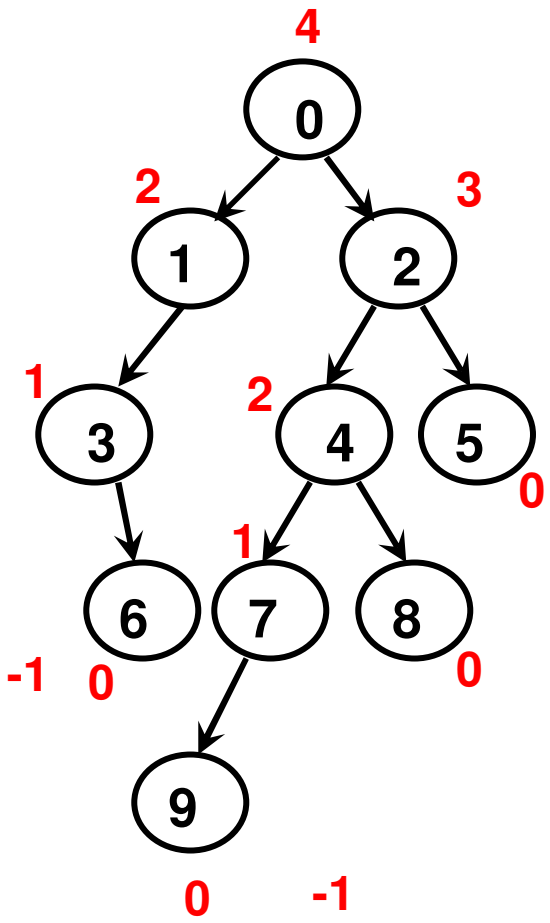
A node is a leaf node if both its left and child node ptrs point to null

```
bool isLeafNode(int nodeid){

        if (arrayOfBTNodes[nodeid].getLeftChildPtr() == 0 &&
                arrayOfBTNodes[nodeid].getRightChildPtr() == 0)
                        return true;

        return false;
}
```

```
void printLeafNodes(){
    for (int id = 0; id < numNodes; id++){

        if (arrayOfBTNodes[id].getLeftChildPtr() == 0 &&
                arrayOfBTNodes[id].getRightChildPtr() == 0)
                        cout << id << " ";
    }

    cout << endl;
}
```

Height of a Binary Tree

- The height of an internal node in a binary tree is one plus the maximum of the height of its child node(s).
- In other words, the height of a node is one plus the maximum of the heights of its two sub trees.
- The height of a leaf node is 0.
- The height of a non-existing child node (i.e., a non-existing sub tree) is -1.
- The height of a binary tree is calculated in a recursive manner.



Ht(0) = 1 + Max(Ht(1), Ht(2))

Ht(1) = 1 + Max(Ht(3), -1)
Ht(3) = 1 + Max(-1, Ht(6))
Ht(6) = 0
➔Ht(3) = 1 + Max(-1, 0) = 1
➔Ht(1) = 1 + Max(1, -1) = 2

Ht(2) = 1 + Max(Ht(4), Ht(5))
Ht(4) = 1 + Max(Ht(7), Ht(8))
Ht(5) = 0
Ht(8) = 0
Ht(7) = 1 + Max(Ht(9), -1)
Ht(9) = 0
➔Ht(7) = 1 + Max(0, -1) = 1
➔Ht(4) = 1 + Max(1, 0) = 2
➔Ht(2) = 1 + Max(2, 0) = 3

Ht(0) = 1 + Max(2, 3) = 4

Binary Tree Implementation (C++ Code 6.2): Class BinaryTree

```
int getNodeHeight(int nodeid){
        if (nodeid == -1)                  // The height of a non-existing node
                return -1                  // (or sub tree) is -1

        if (isLeafNode(nodeid) )           // The height of a leaf node is 0
                return 0;

        int leftChildID = arrayOfBTNodes[nodeid].getLeftChildID(); // -1 if not exist
        int rightChildID = arrayOfBTNodes[nodeid].getRightChildID(); // -1 if not exist

        return max(getNodeHeight(leftChildID), getNodeHeight(rightChildID)) + 1;

}
```

```
int getTreeHeight(){
        return getNodeHeight(0);
}
```

// The height of a binary tree
// is the height of node 0
// node 0 – the root node

C++ Code 6.2: Main Function

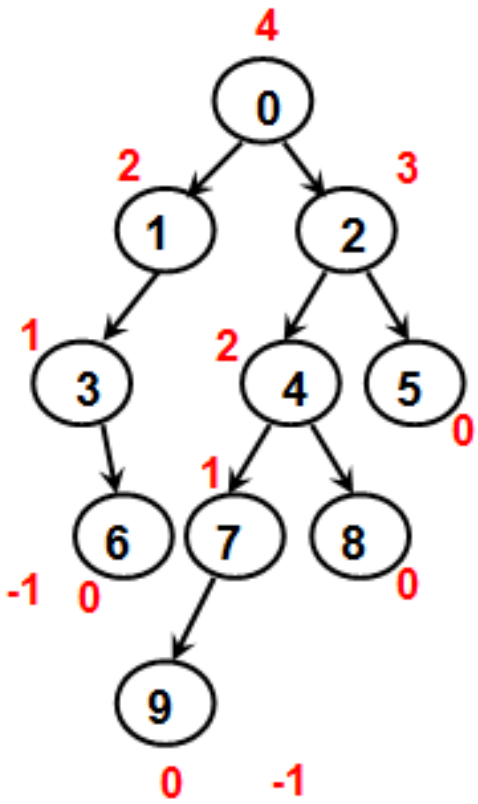
```
string filename;
cout << "Enter a file name: ";      // Enter the name of the text file
cin >> filename;                    // that stores the tree info

int numNodes;
cout << "Enter number of nodes: ";
cin >> numNodes;

BinaryTree binaryTree(numNodes);  // Instantiate an object of class BinaryTree

ifstream fileReader(filename);  // Instantiate an object of class ifstream

if (!fileReader){
        cout << "File cannot be opened!! ";
        return 0;                   // We need to know the maximum number of characters
}                                   // that would be there per line in the file in order to read
int numCharsPerLine = 10;           // the line/string as a character array and do string
                                    // tokenization
char *line = new char[numCharsPerLine];
// '10' is the maximum number of characters per line

fileReader.getline(line, numCharsPerLine, '\n');      // Get the first line in the file
// '\n' is the delimiting character to stop reading the line
```

C++ Code 6.2: Main Function

```
while (fileReader){    // Run the loop until the last line is read (i.e., fileReader != null)
        char* cptr = strtok(line, ",: "); // Read the first token as the upstreamNodeID
        string upstreamNodeToken(cptr);
        int upstreamNodeID = stoi(upstreamNodeToken);

        cptr = strtok(NULL, ",: ");

        int childIndex = 0; // 0 for left child; 1 for right child

        while (cptr != 0){    // Loop through to extract the next two tokens

                string downstreamNodeToken(cptr);
                int downstreamNodeID = stoi(downstreamNodeToken);
        // The second token (i.e., when childIndex = 0) is the left child node id.
                if (childIndex == 0 && downstreamNodeID != -1)
                        binaryTree.setLeftLink(upstreamNodeID, downstreamNodeID);
        // The third token (i.e., when childIndex = 1) is the right child node id.
                if (childIndex == 1 && downstreamNodeID != -1)
                        binaryTree.setRightLink(upstreamNodeID, downstreamNodeID);

                cptr = strtok(NULL, ",: ");
                childIndex++;
        }

        fileReader.getline(line, numCharsPerLine, '\n');

}

binaryTree.printLeafNodes();
cout << "Tree Height: " << binaryTree.getTreeHeight() << endl;
cout << "Height of node 1: " << binaryTree.getNodeHeight(1) << endl;
```

Height-Balanced Binary Tree: Ex. 1

- An internal node in a binary tree is said to be height-balanced if the absolute difference of the heights of its left sub tree and right sub tree is no greater than 1.
- A binary tree is said to be height-balanced if all its internal nodes are height-balanced.
  - Even if one internal node is not height-balanced, then the binary tree is not height-balanced.

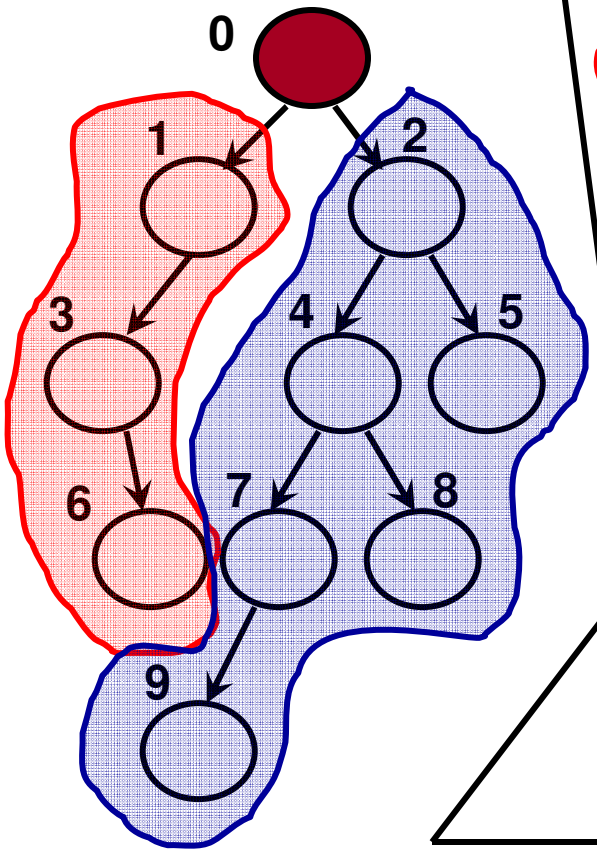


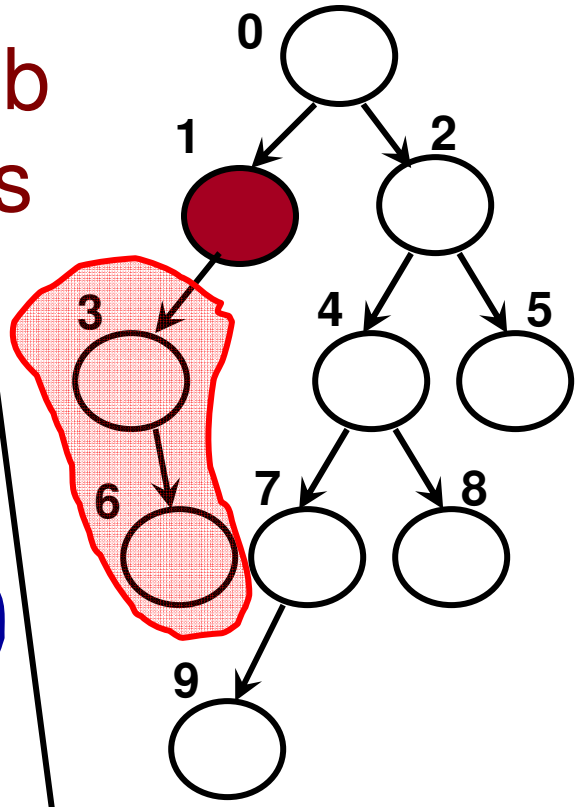
Node	Height of Left subtree	Height of Right subtree	Abs. Diff.	Height Balanced
0	2	3	1	YES
1	1	-1	2	NO
2	2	0	2	NO
3	-1	0	1	YES
4	1	0	1	YES
7	0	-1	1	YES

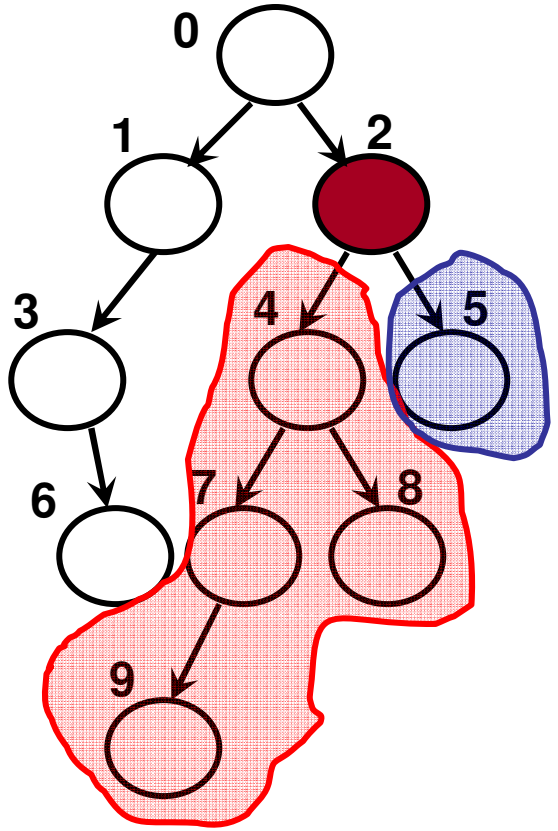
Since nodes 1 and 2 are not height-balanced, the whole Binary tree is considered to be not height-balanced.

Note: The height of a leaf node is 0.
The height of a (non-existing) subtree with no nodes is -1

Height of the sub trees: Examples







Node '0'

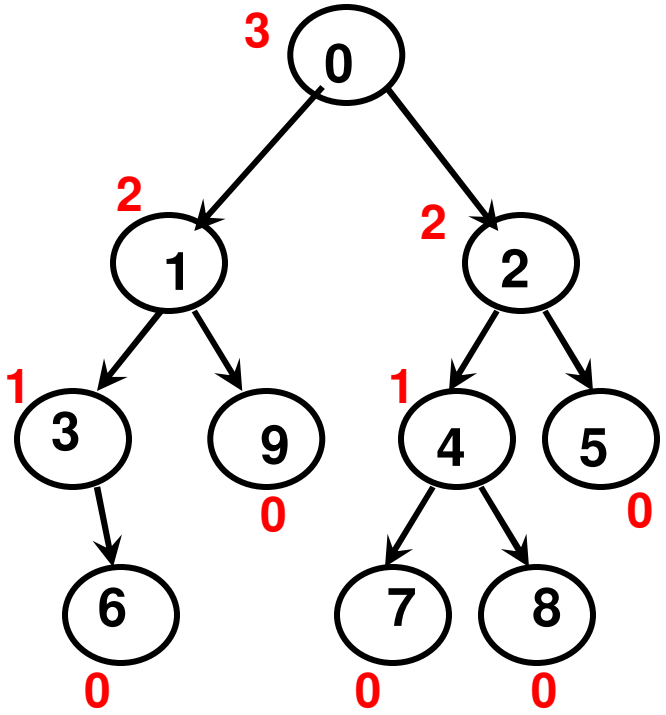
Sub trees	Root	Height
Left sub tree	1	2
Right sub tree	2	3

Node '1'

Sub trees	Root	Height
Left sub tree	3	1
Right sub tree	None	-1

Node '2'

Sub trees	Root	Height
Left sub tree	4	2
Right sub tree	5	0



Height-Balanced Binary Tree: Ex. 2

Node	Height of Left subtree	Height of Right subtree	Abs. Diff.	Height Balanced
0	2	2	0	YES
1	1	0	1	YES
2	1	0	1	YES
3	-1	0	1	YES
4	0	0	0	YES

Since all the internal nodes are height-balanced, the entire binary tree is height-balanced.

Breadth First Search (BFS) Algorithm

- BFS is a graph traversal algorithm (also applicable for binary trees) that could be used to determine the shortest paths from a vertex to the rest of the vertices in the graph
- In the context of binary trees, BFS could be used to determine the number of edges (level # or the depth) on the shortest paths from the root to the rest of the vertices in the graph.
- BFS proceeds in iterations. We keep track of the vertices visited in a queue.
- We start the algorithm by enqueuing the root node to the queue.
- In each iteration, we dequeue the vertex that is currently the first node (i.e., in the front) of the queue and visit its the child node(s) (i.e., enqueue them).
  - In case of a tie, the child node with the smallest ID is enqueued first.
- The level number of a node is one more than the level number of its immediate parent node.
  - The level number of a child node is set when its parent is dequeued and the child node and its sibling node, if any, are enqueued.
- The algorithm is run until there is at least one node in the queue.

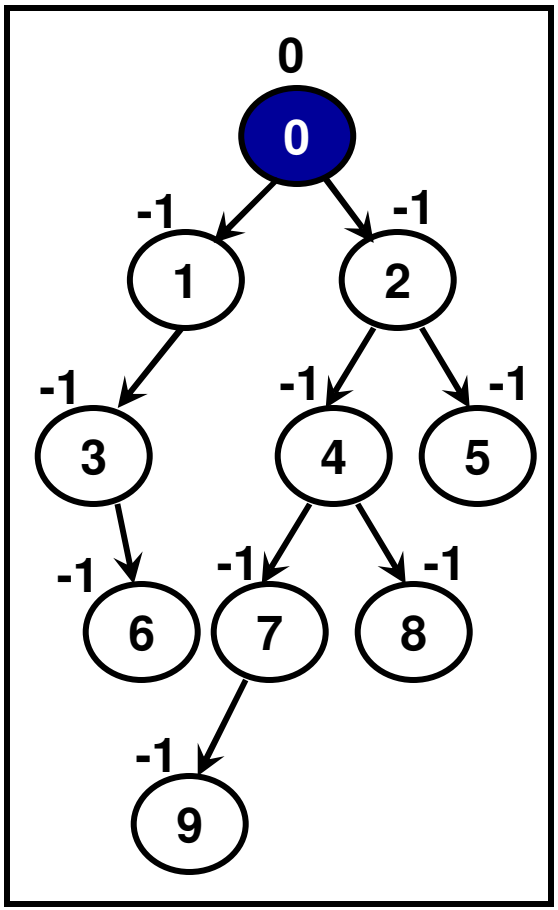
Breadth First Search (BFS) Algorithm

```
Queue queue
queue.enqueue(root node id 0)
Level[root node 0] = 0

Begin BFS_BinaryTree
while (!queue.isEmpty()) do
   FirstNodeID = queue.dequeue();
   if (FirstNode.leftChildNodeID != -1) then
      Level[FirstNode.leftChildNodeID] = Level[FirstNode.getNodeID()] + 1
      queue.enqueue(FirstNode.leftChildNodeID)
   end if
   if (FirstNode.rightChildNodeID != -1) then
      Level[FirstNode.rightChildNodeID] = Level[FirstNode.getNodeID()] + 1
      queue.enqueue(FirstNode.rightChildNodeID)
   end if
end while
End BFS_BinaryTree
```

BFS is a traversal algorithm that returns the level number of the vertices (i.e., the number of edges on the path from the root node to any other vertex).

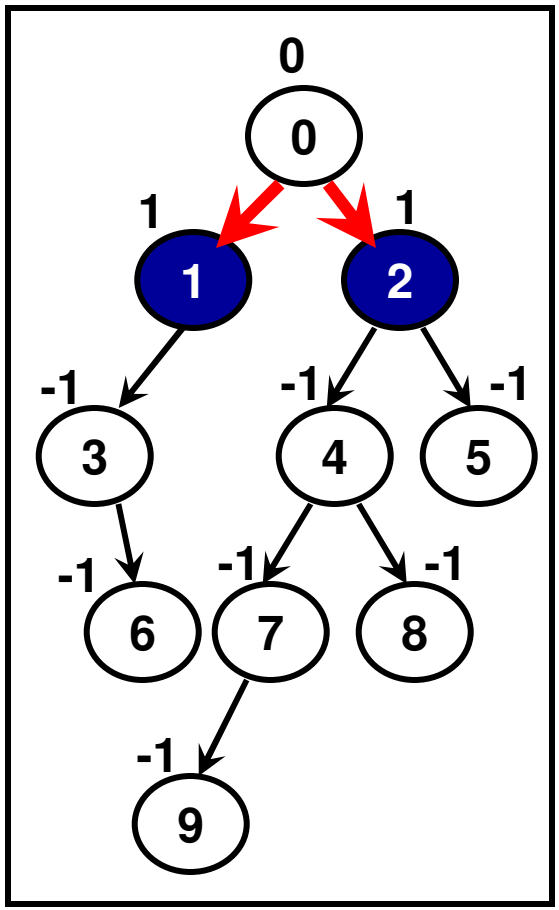
For a binary tree of V vertices, we need to run V iterations and enqueue at most two of its Child nodes in each iteration (traverse E edges). Hence, the time complexity of BFS is Θ(V+E).

BFS Execution: Example



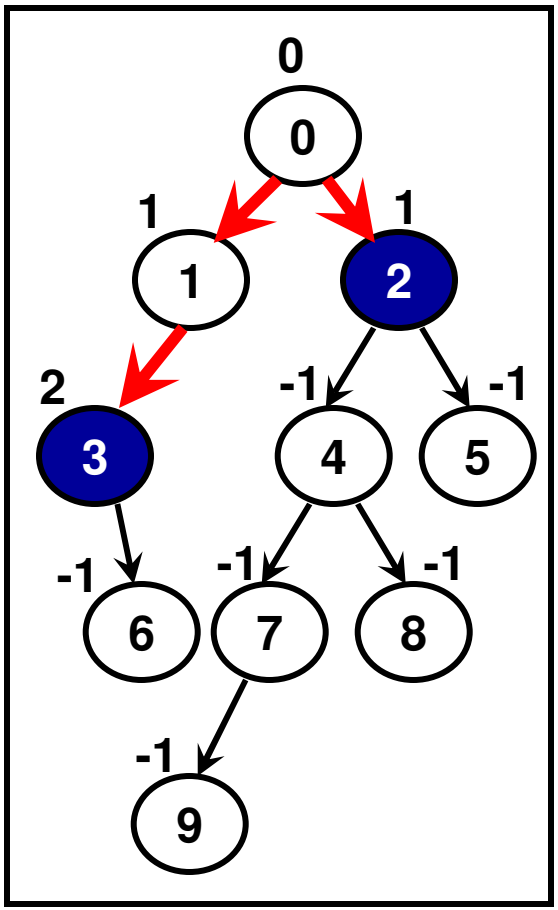
Queue Initialization

{0}



Iteration 1

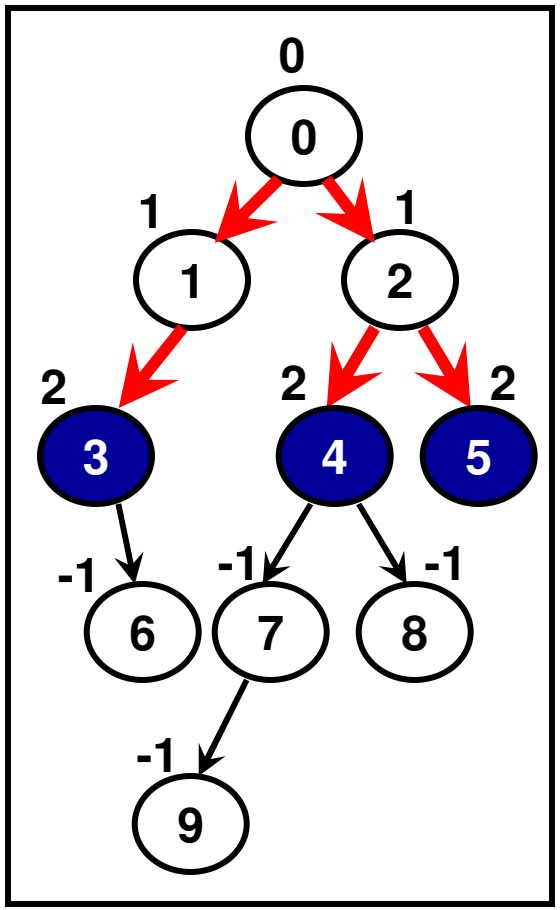
{1, 2}



Iteration 2

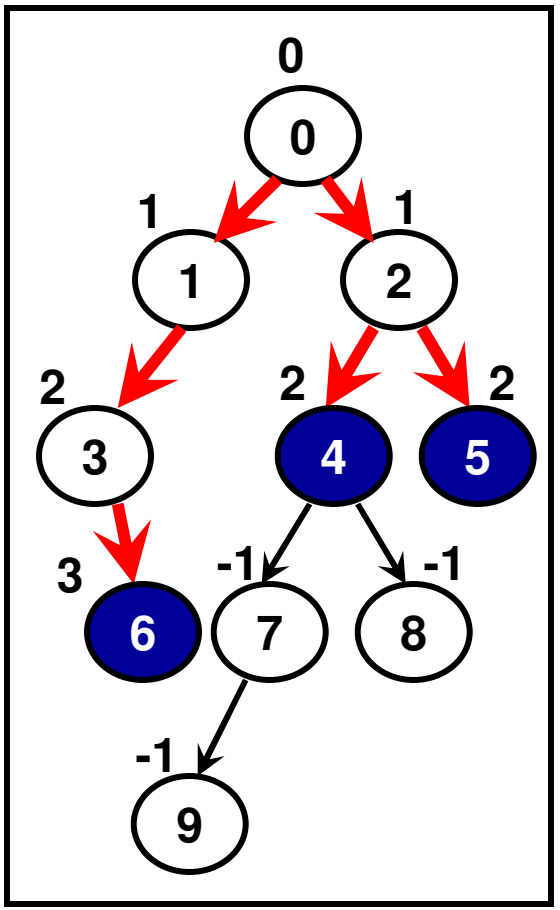
{2, 3}

BFS Execution: Example



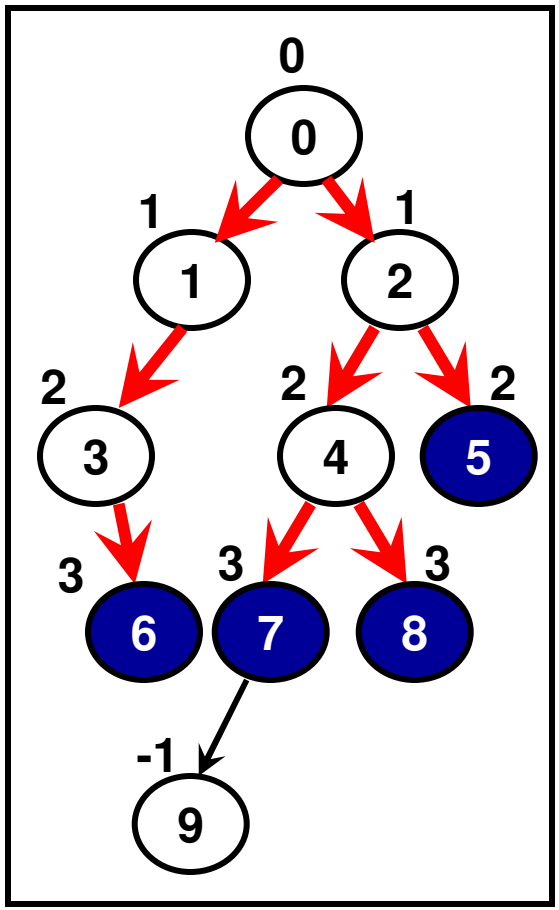
Iteration 3

{3, 4, 5}



Iteration 4

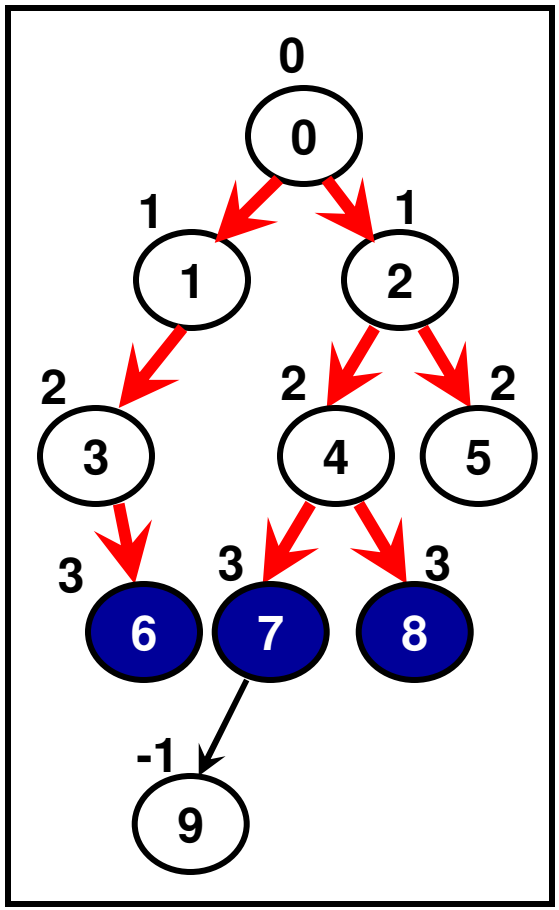
{4, 5, 6}



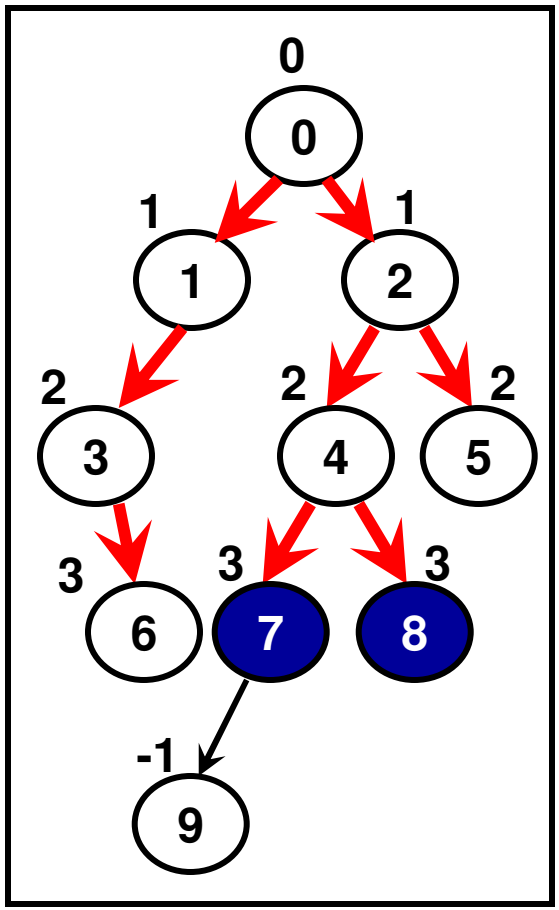
Iteration 5

{5, 6, 7, 8}

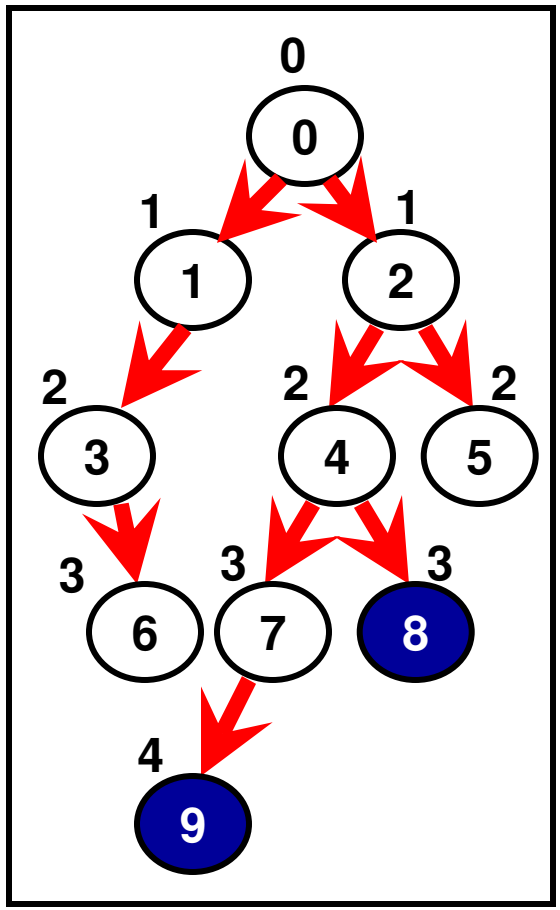
BFS Execution: Example



Iteration 6
{6, 7, 8}

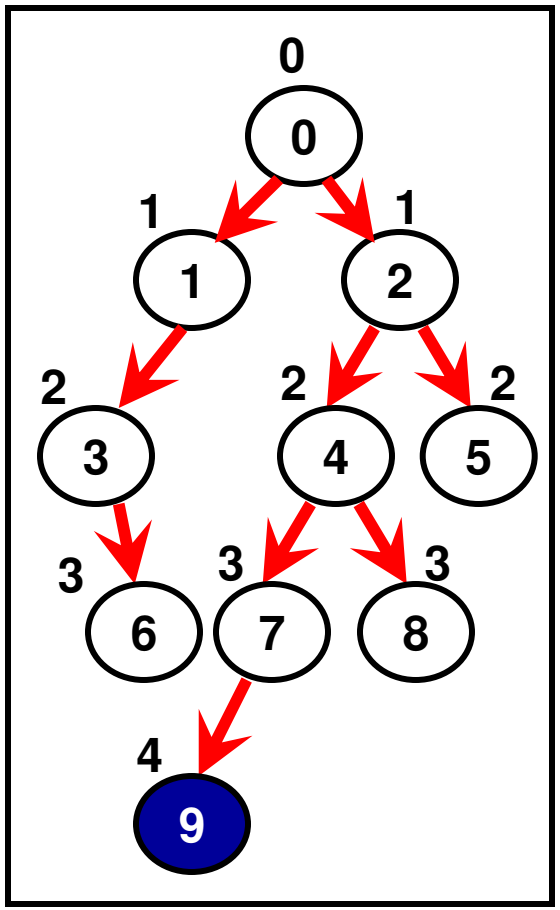


Iteration 7
{7, 8}



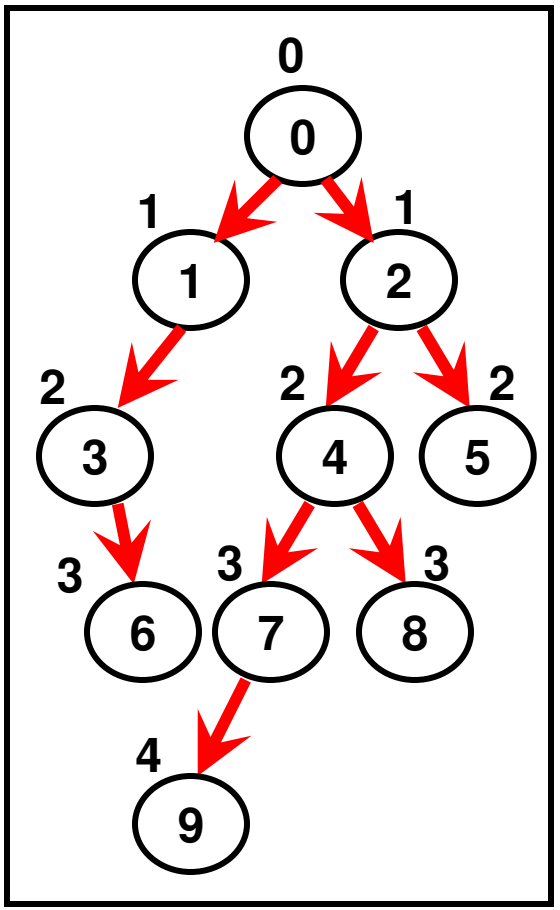
Iteration 8
{8, 9}

BFS Execution: Example



Iteration 9

{9}



Iteration 10

{}

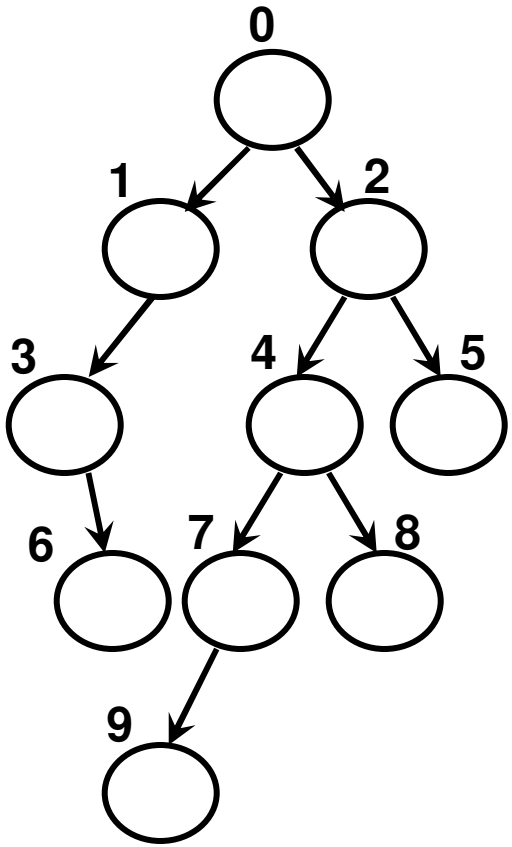
Depth of a Binary Tree (C++ Code: 6.3)

```
void assignLevelNumbers(){

        Queue queue;          // Instantiate an object of class Queue and enqueue the root
        queue.enqueue(0);
        arrayOfBTNodes[0].setLevelNum(0);          // Set the level number of the root to 0

        while (!queue.isEmpty()){

                int firstNodeInQueue = queue.dequeue();          // Dequeue the first node in the
                                                                 // queue
                int leftChildID = arrayOfBTNodes[firstNodeInQueue].getLeftChildID();
                if (leftChildID != -1){
                        queue.enqueue(leftChildID);          // Enqueue the left child node in the queue
                        arrayOfBTNodes[leftChildID].setLevelNum(
                                arrayOfBTNodes[firstNodeInQueue].getLevelNum()+1);
                }       // The level number of the left child is one more than that of its
                        // immediate parent node
                int rightChildID = arrayOfBTNodes[firstNodeInQueue].getRightChildID();
                if (rightChildID != -1){
                        queue.enqueue(rightChildID);          // Enqueue the right child node in the queue
                        arrayOfBTNodes[rightChildID].setLevelNum(
                                arrayOfBTNodes[firstNodeInQueue].getLevelNum()+1);
                }
                        // The level number of right child one more than that of its parent
        }
}
```

```
int getDepth(int nodeid){
        return arrayOfBTNodes[nodeid].getLevelNum();
}
```

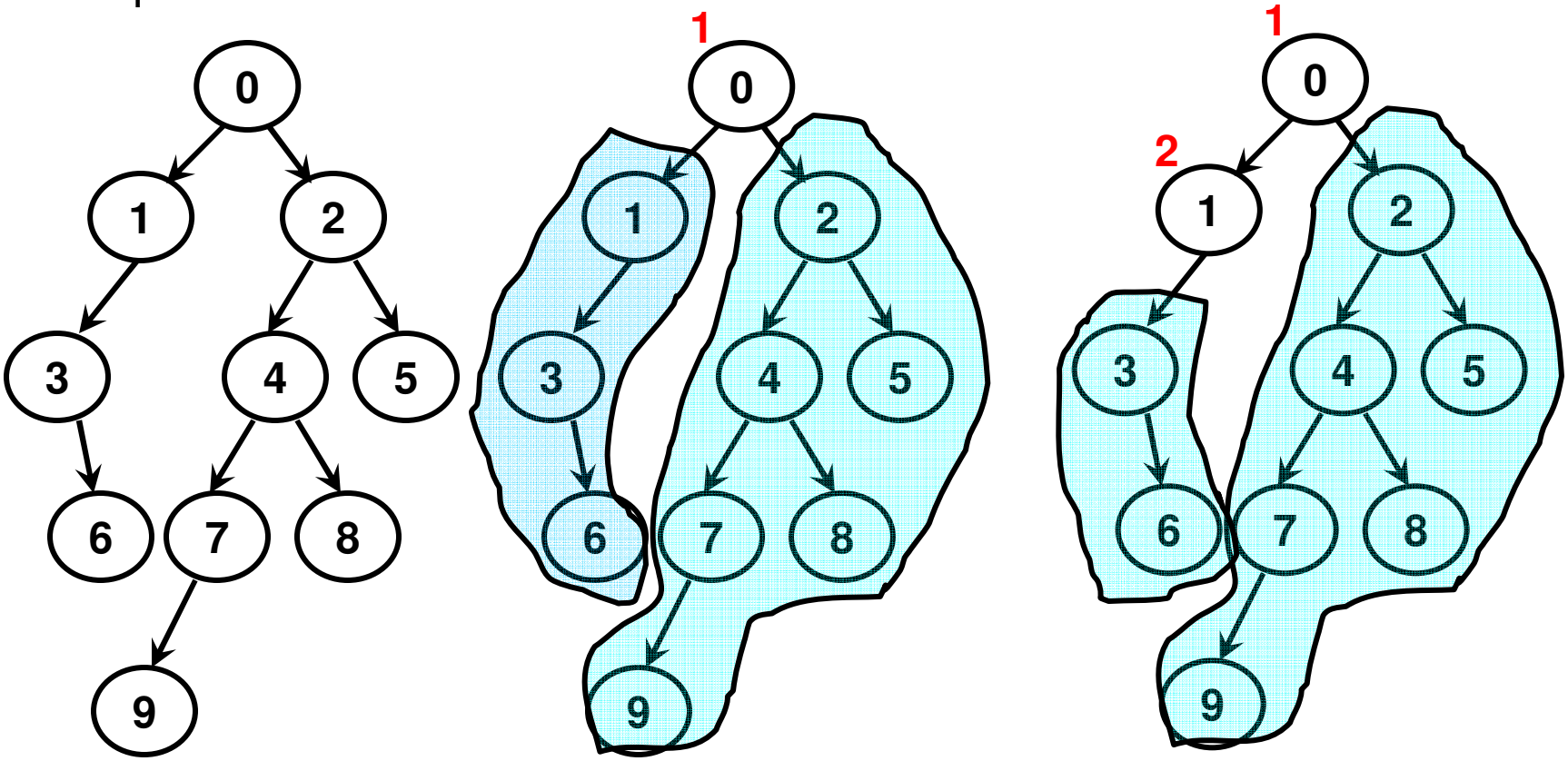
Pre, Post and In Order Traversals

- We will now look at three tree traversal algorithms that could be used to print the vertices of the tree in a certain order.
- **Pre Order:**
- Root, Left sub tree, Right sub tree
  - 0, 1, 3, 6, 2, 4, 7, 9, 8, 5
- **In Order:**
- Left sub tree, Root, Right sub tree
  - 3, 6, 1, 0, 9, 7, 4, 8, 2, 5
- **Post Order:**
- Left sub tree, Right sub tree, Root
  - 6, 3, 1, 9, 7, 8, 4, 5, 2, 0
- To implement each of the three traversal algorithms, we go through a recursive approach

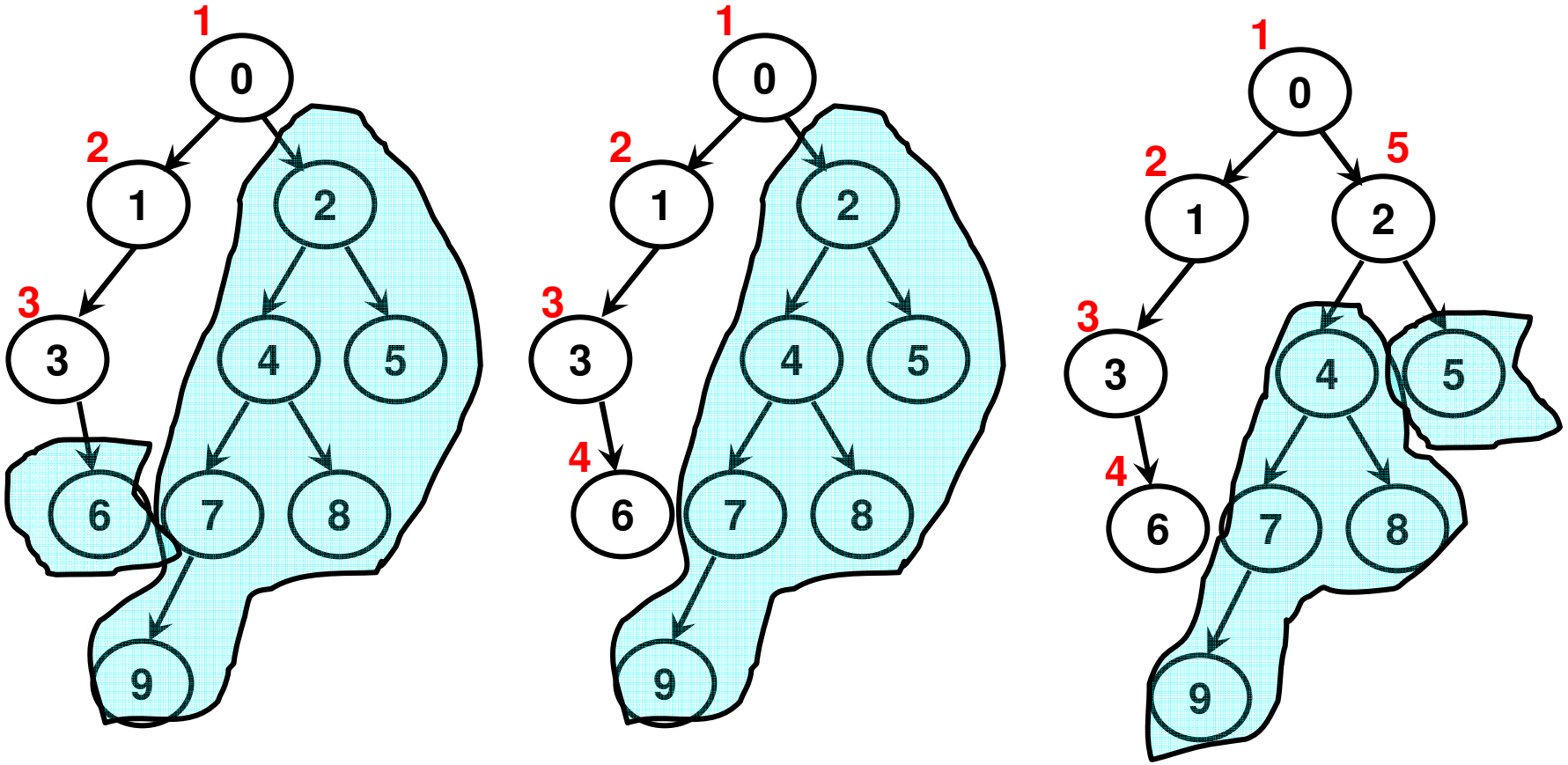


Pre Order Traversal: Example

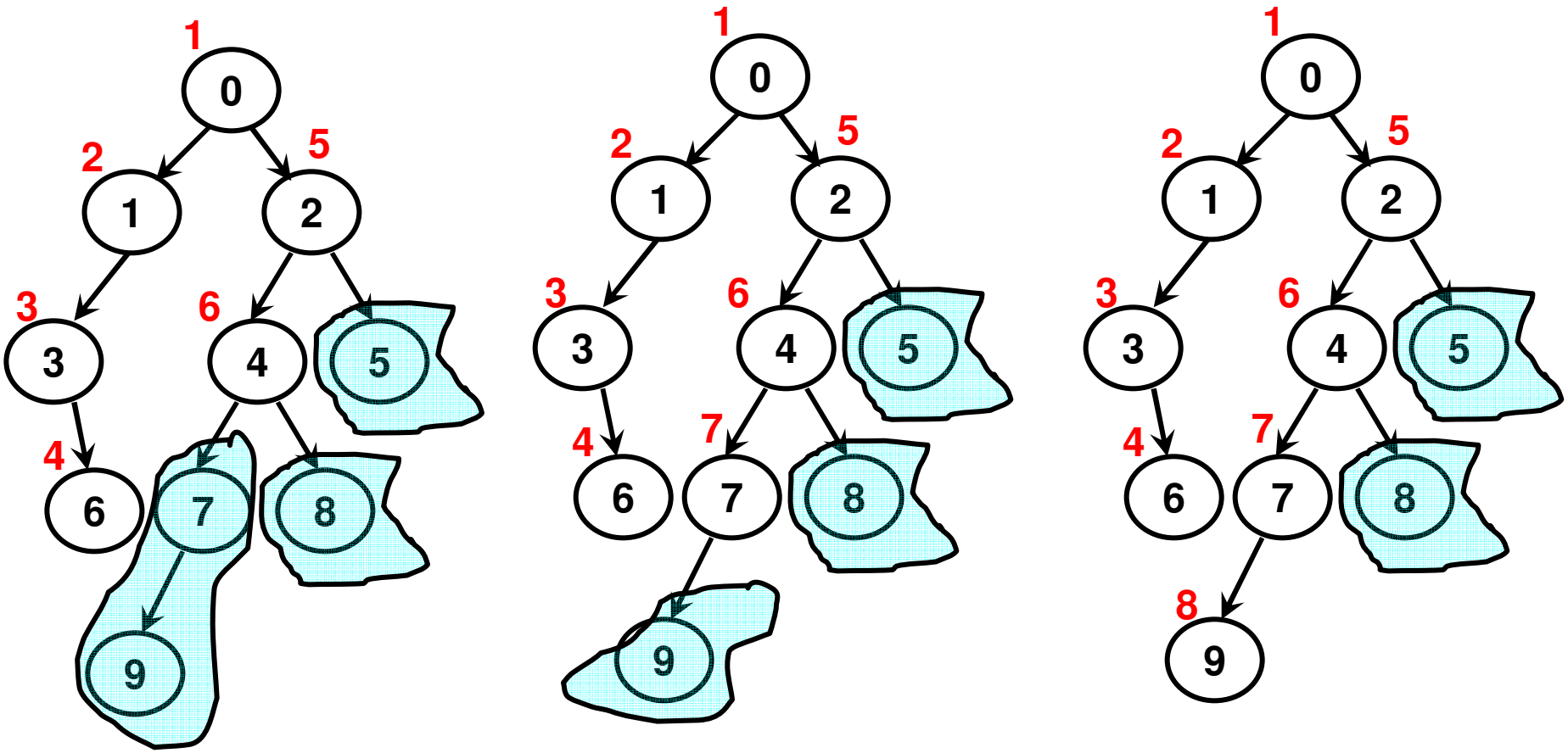
- We first print the root of the sub tree, followed by all the nodes of the left sub tree of the root, and followed by all the nodes of the right sub tree of the root
- To print the nodes of the left and right sub tree, we follow the above procedure.



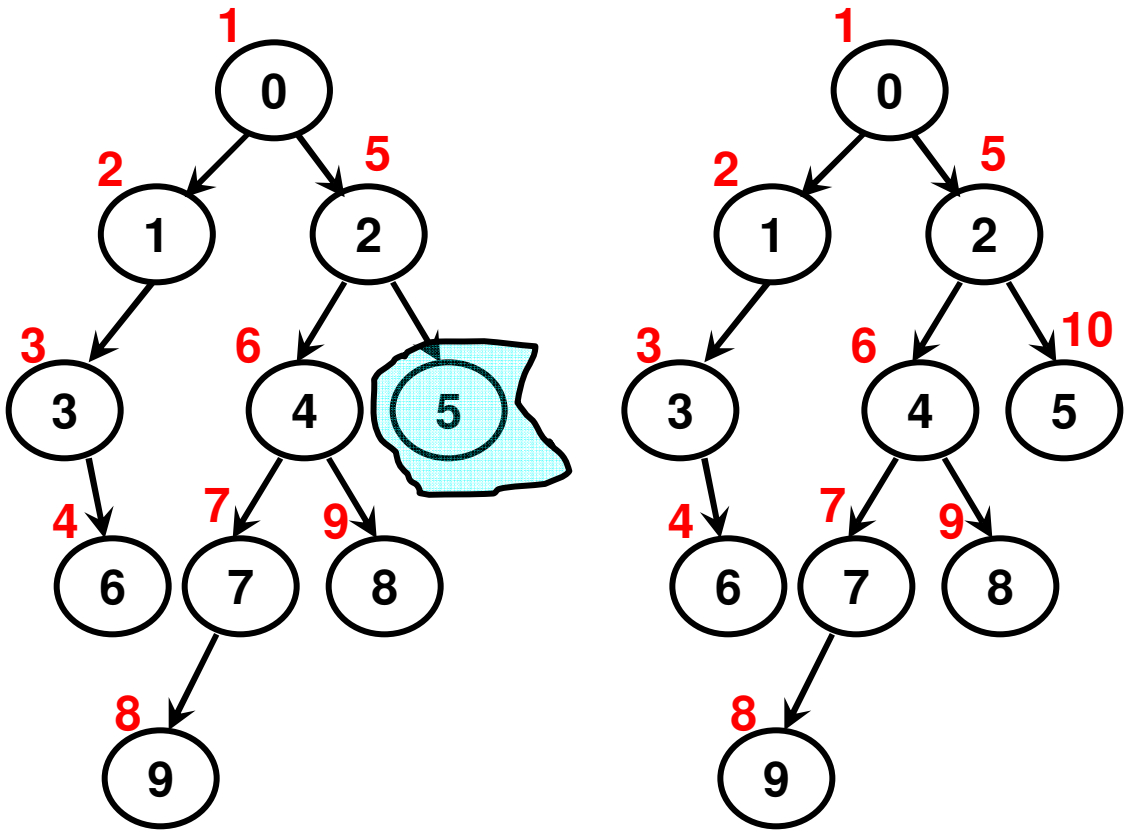
Pre Order Traversal: Example



Pre Order Traversal: Example



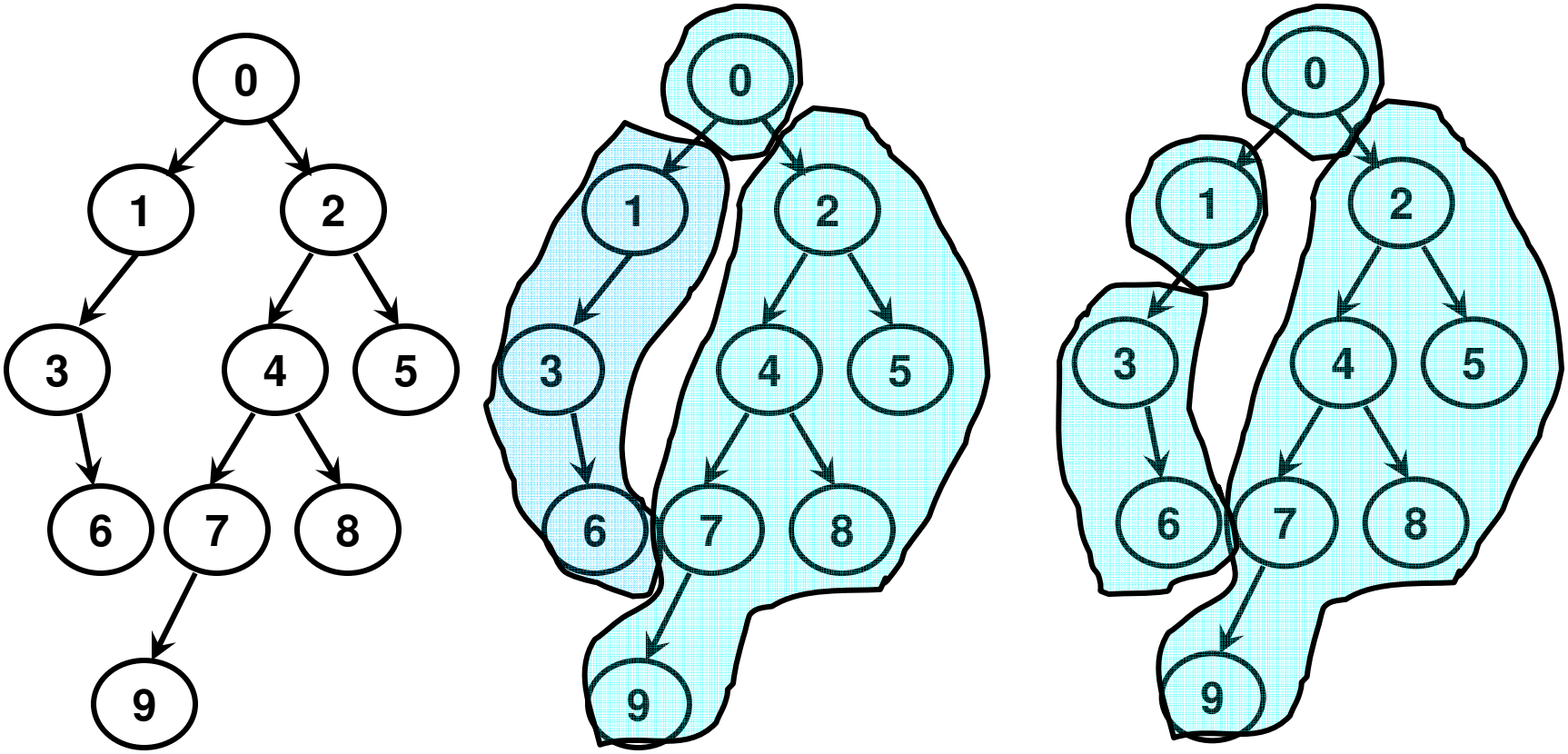
Pre Order Traversal: Example



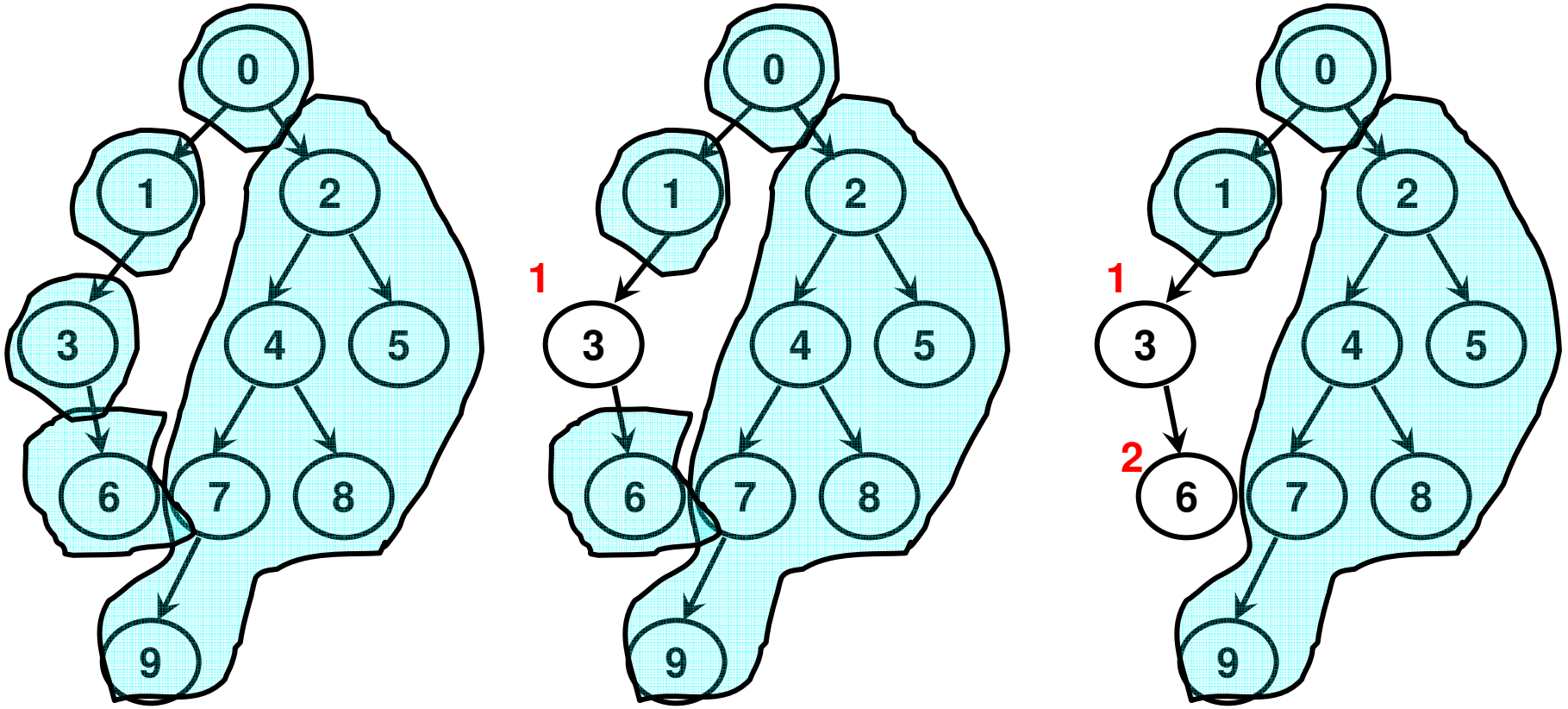
0, 1, 3, 6, 2, 4, 7, 9, 8, 5

In Order Traversal: Example

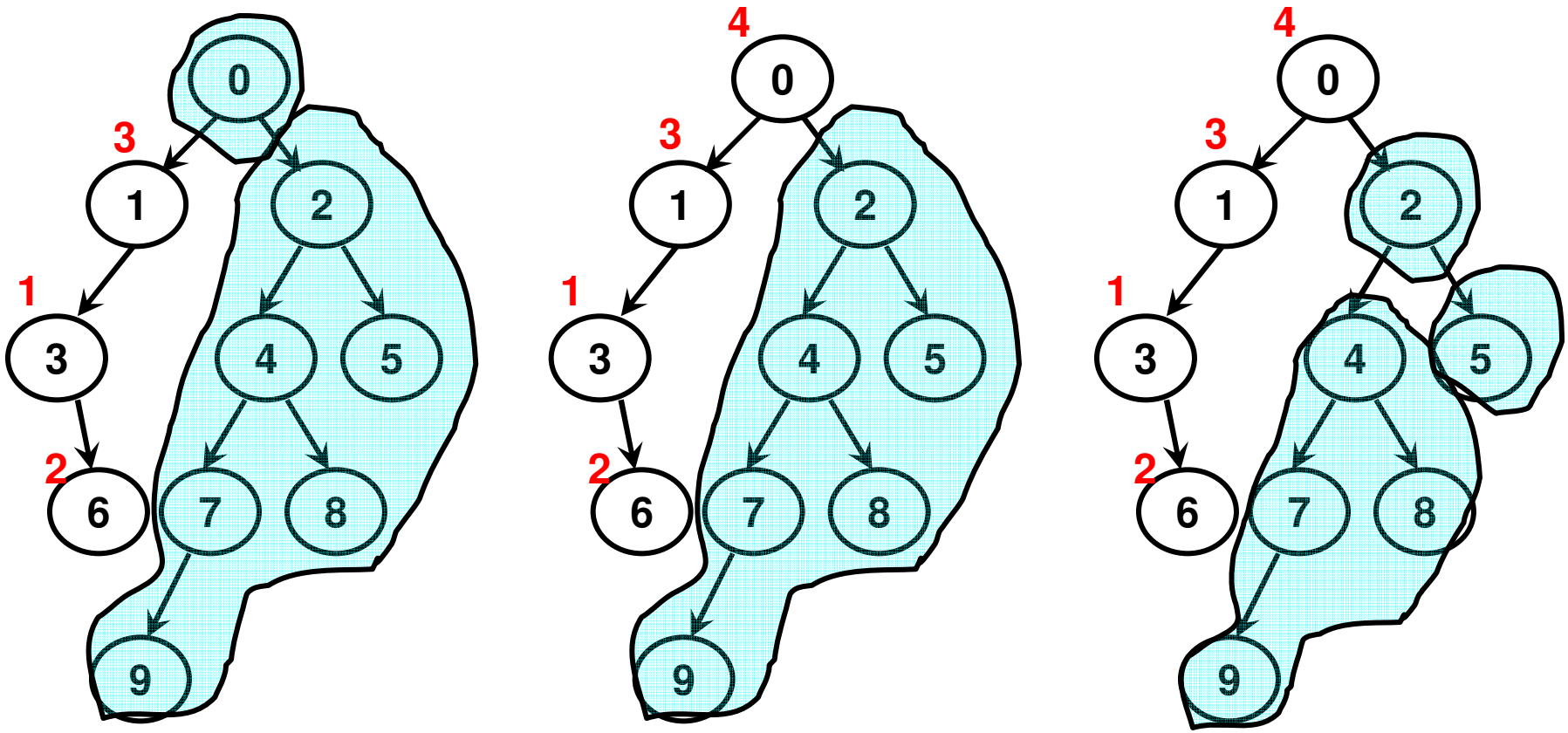
- We first print all the nodes in the left sub tree of the root, followed by the root, followed by the nodes in the right sub tree of the root, if any.
- To print the nodes of the left and right sub tree, we follow the above procedure.



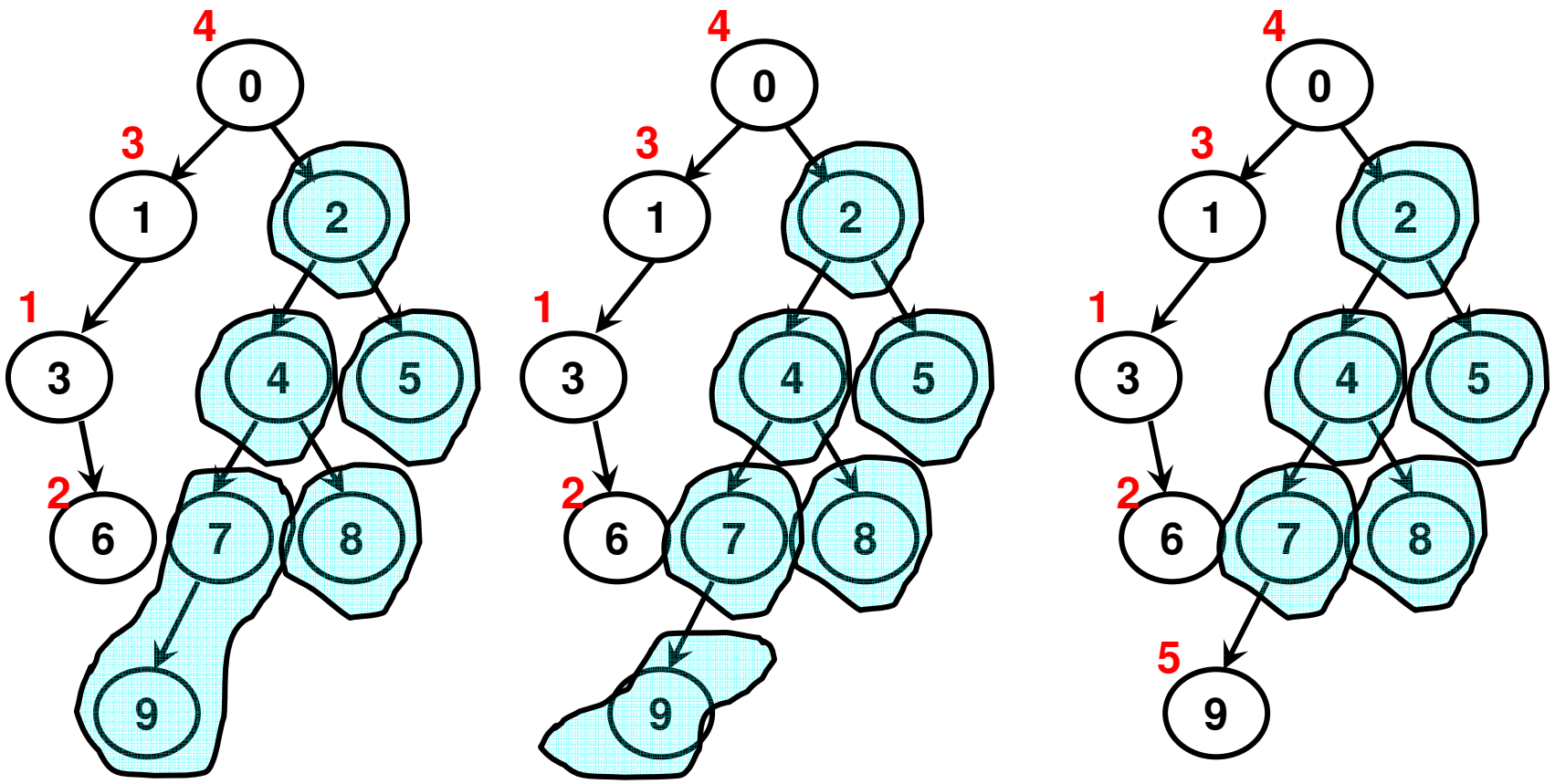
In Order Traversal: Example



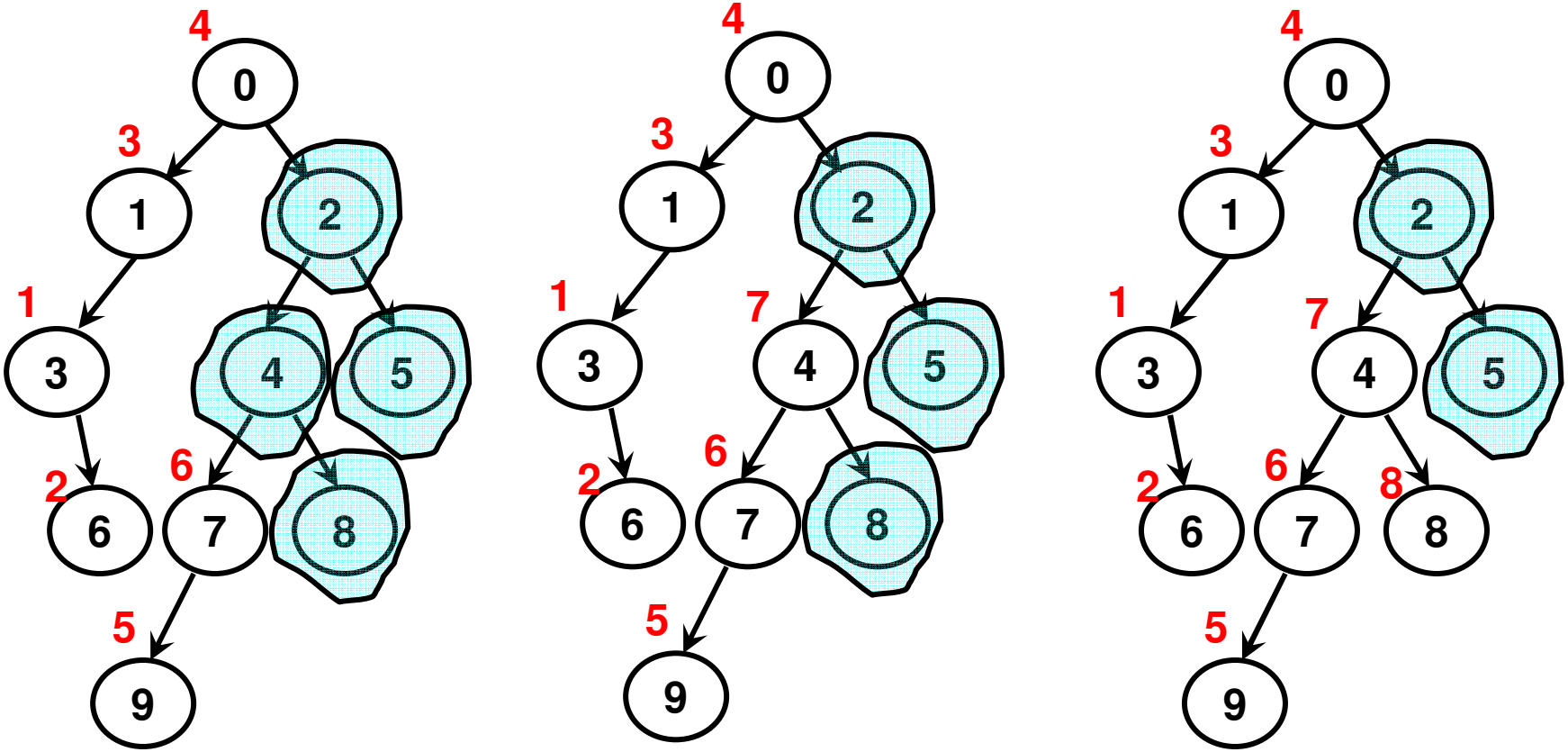
In Order Traversal: Example



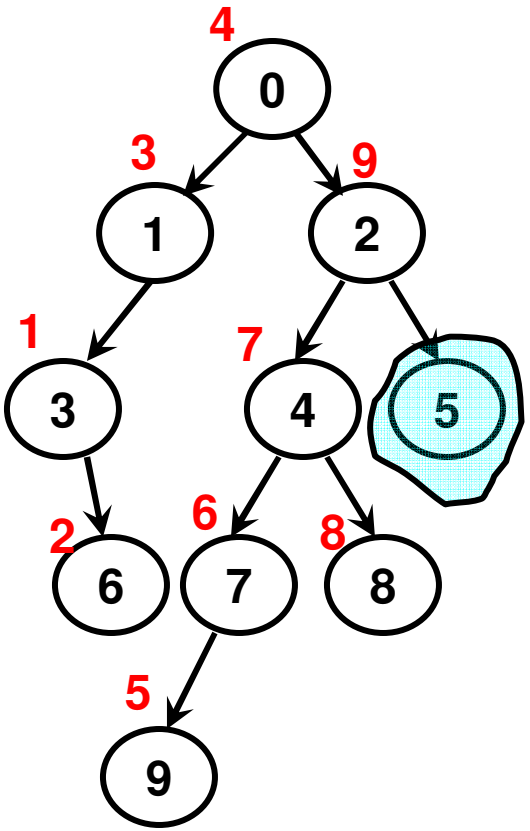
In Order Traversal: Example

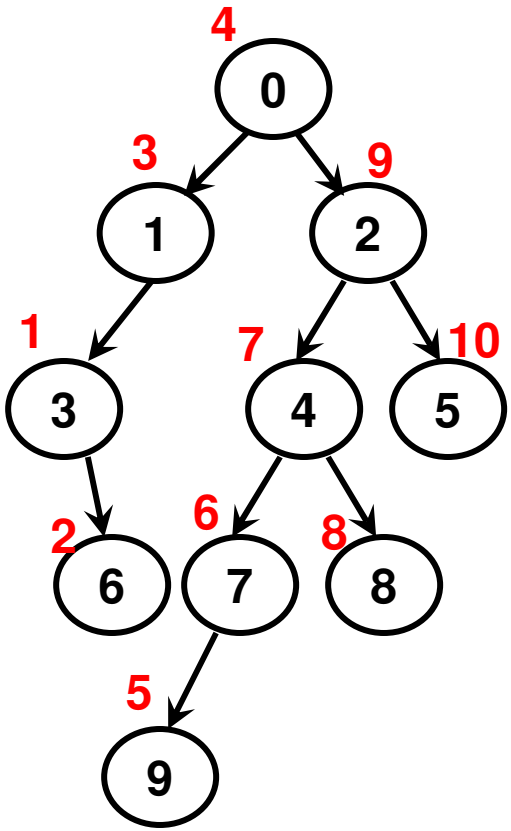


In Order Traversal: Example



In Order Traversal: Example

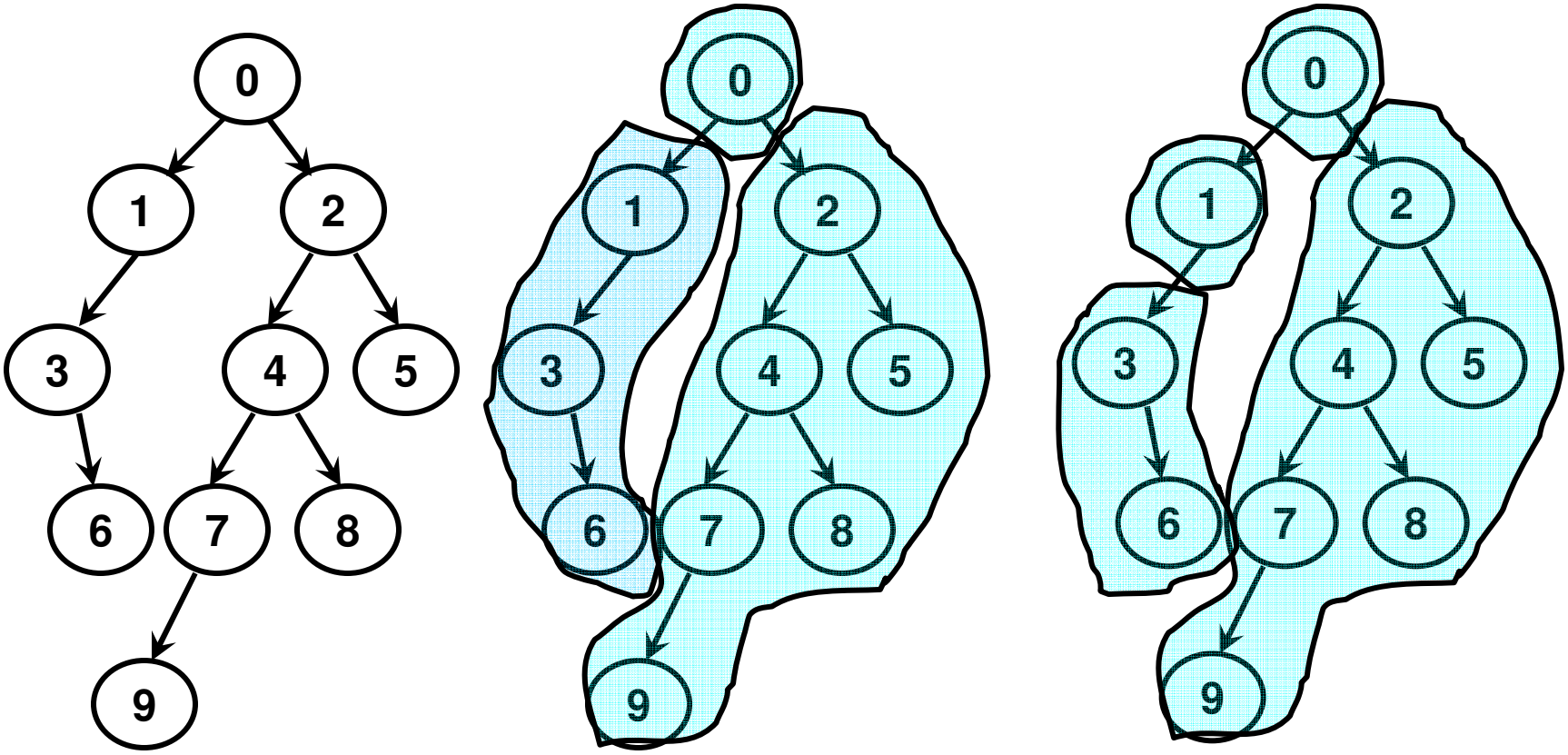




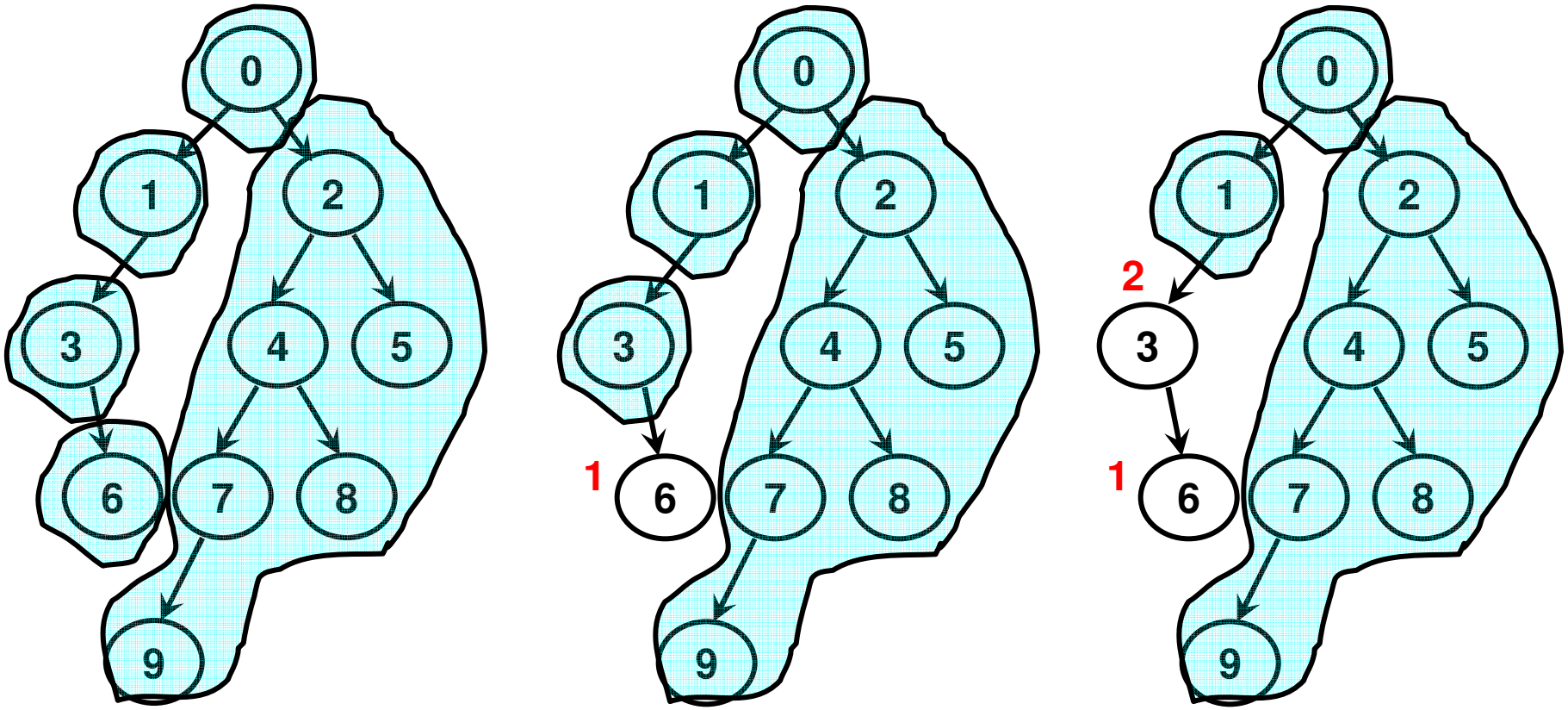
3, 6, 1, 0, 9, 7, 4, 8, 2, 5

Post Order Traversal: Example

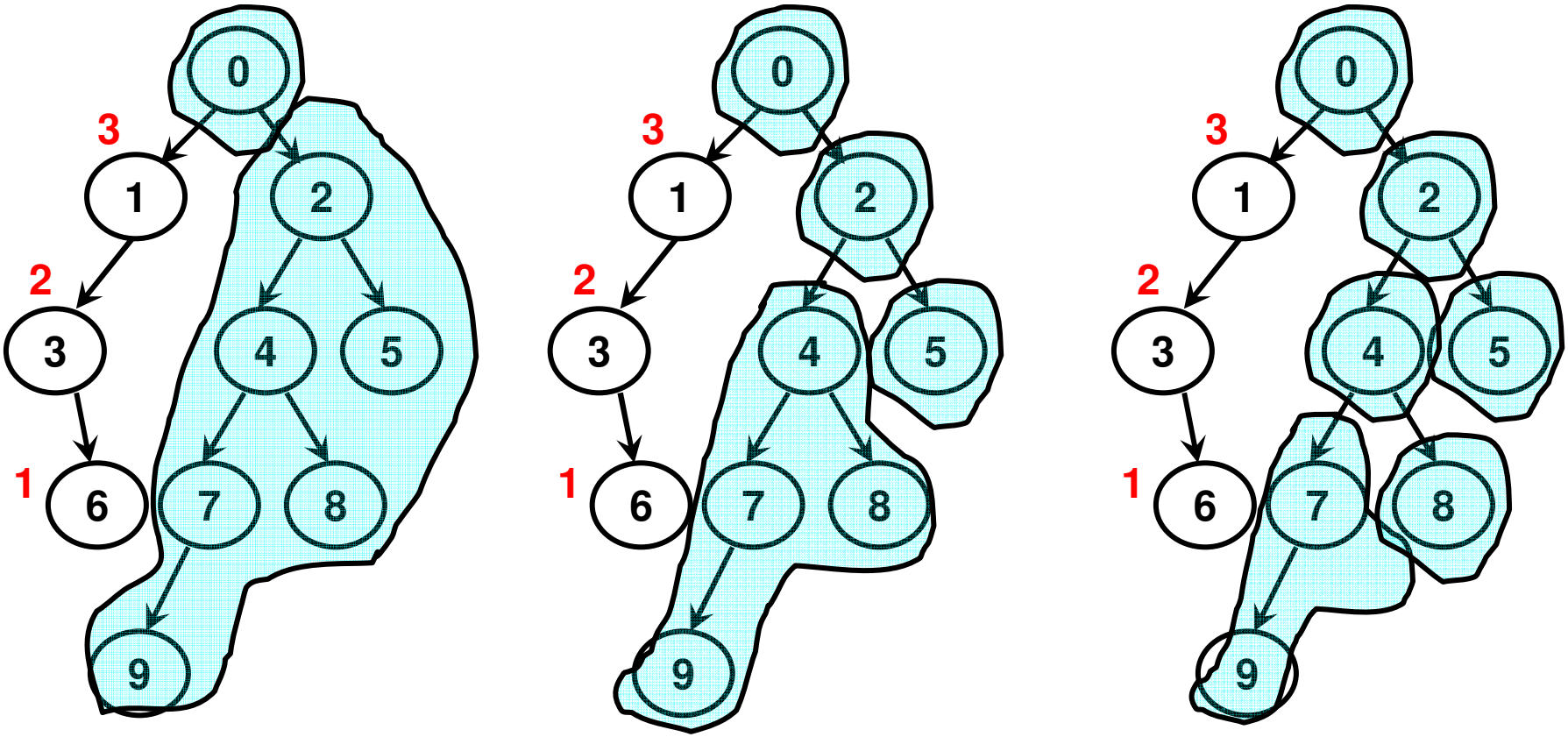
- We first print all the nodes in the left sub tree, followed by all the nodes in the right sub tree and then the root.
- To print the nodes of the left and right sub tree, we follow the above procedure.



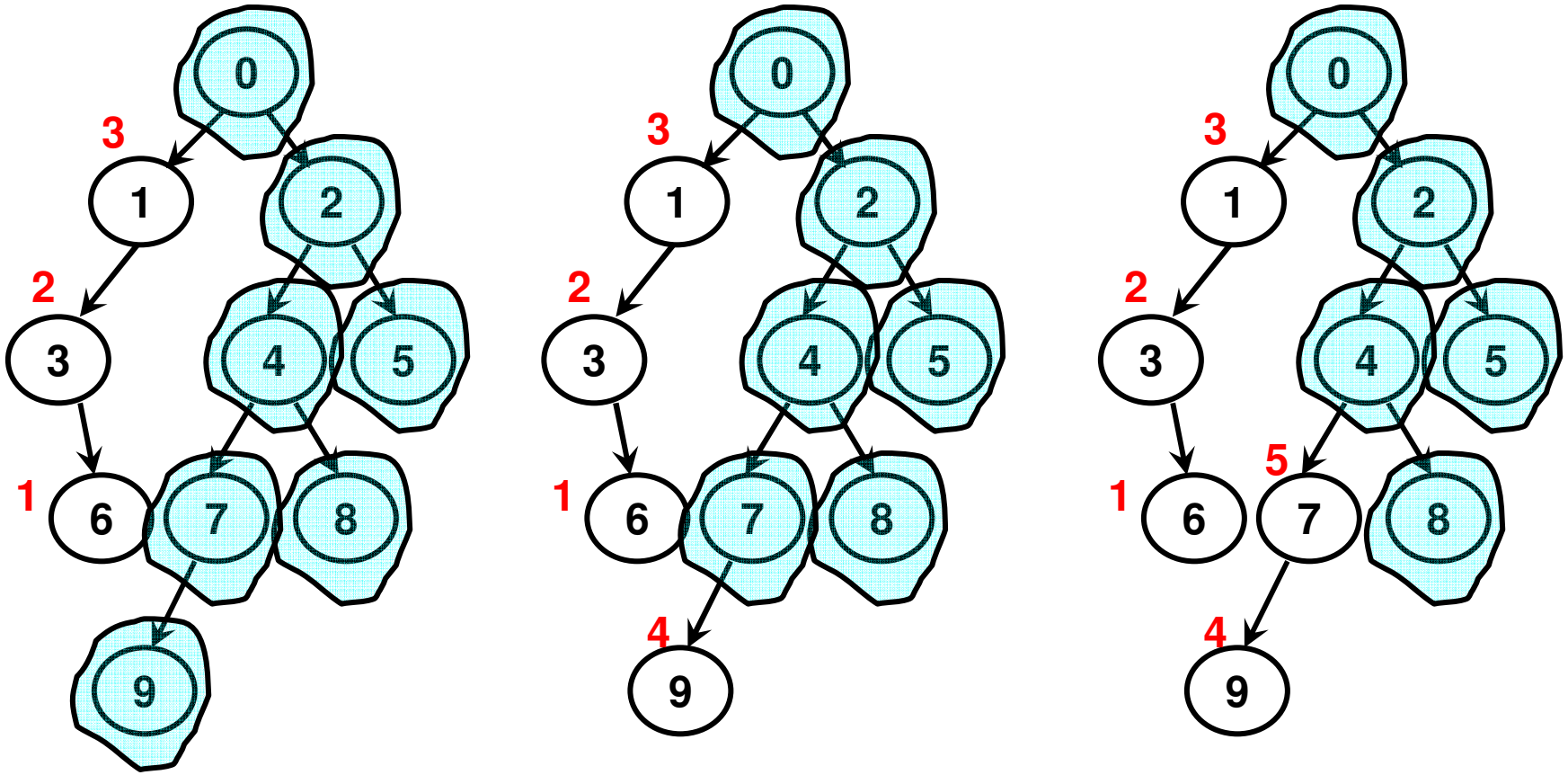
Post Order Traversal: Example



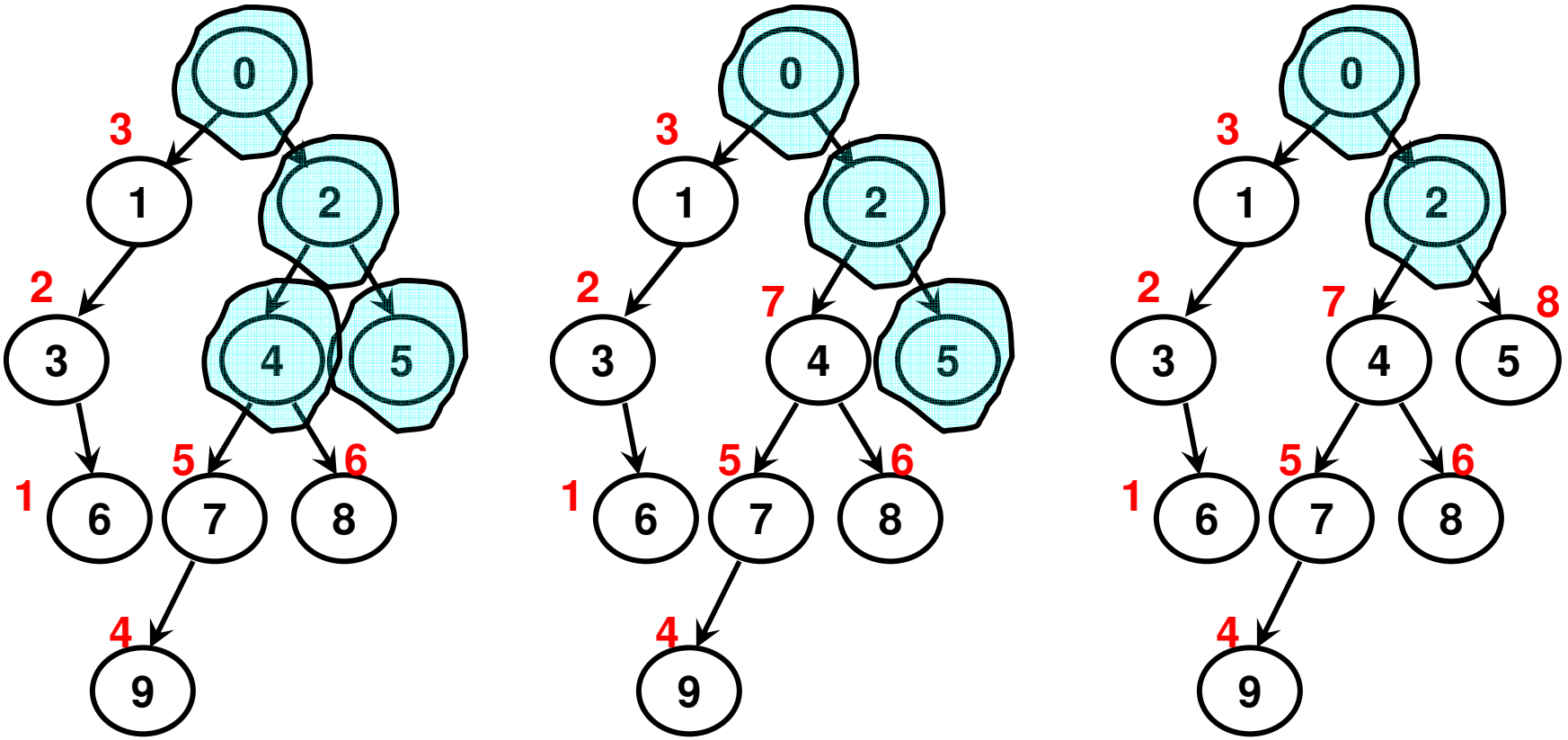
Post Order Traversal: Example



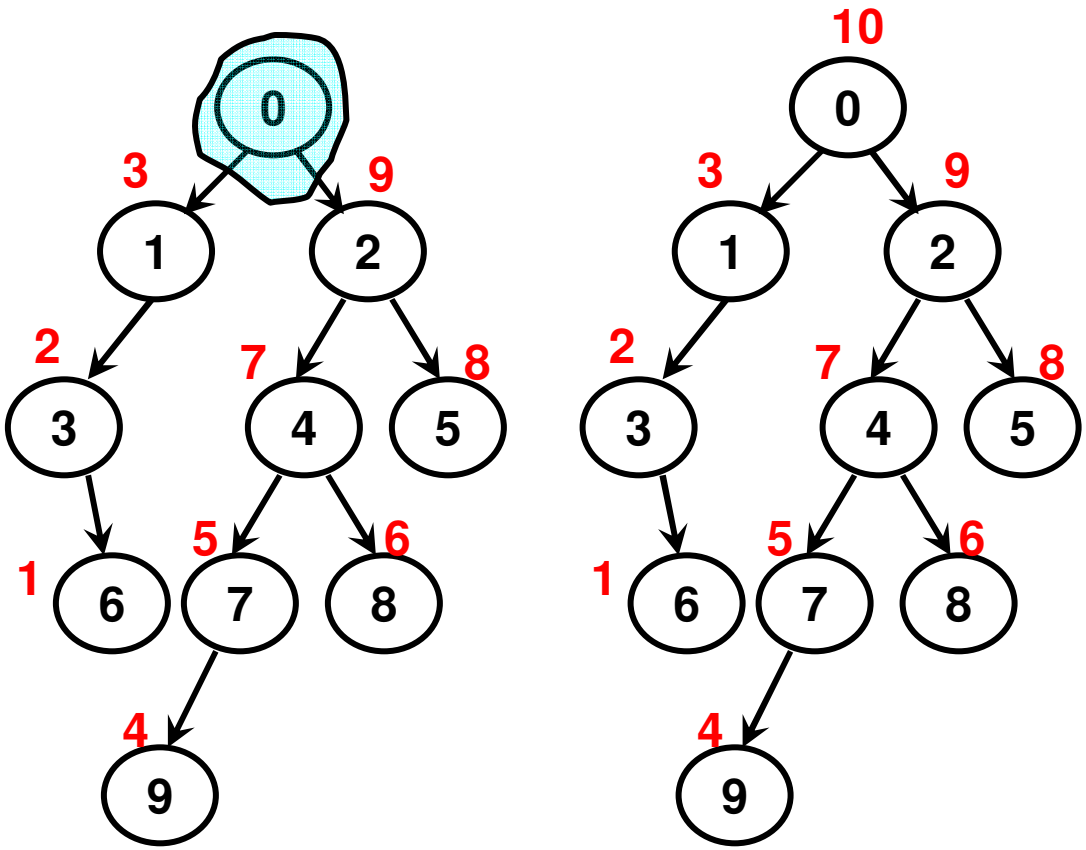
Post Order Traversal: Example



Post Order Traversal: Example



Post Order Traversal: Example



6, 3, 1, 9, 7, 8, 4, 5, 2, 0

Pre Order Traversal (Code: 6.4)

```
void PreOrderTraversal(int nodeid){
        if (nodeid == -1)
                return;

        cout << nodeid << " ";

        PreOrderTraversal(arrayOfBTNodes[nodeid].getLeftChildID());
        PreOrderTraversal(arrayOfBTNodes[nodeid].getRightChildID());
  }
```

```
void PrintPreOrderTraversal(){
        PreOrderTraversal(0);
        cout << endl;
}
```

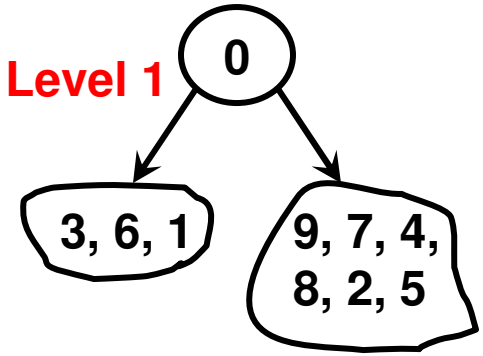
Constructing a Binary Tree

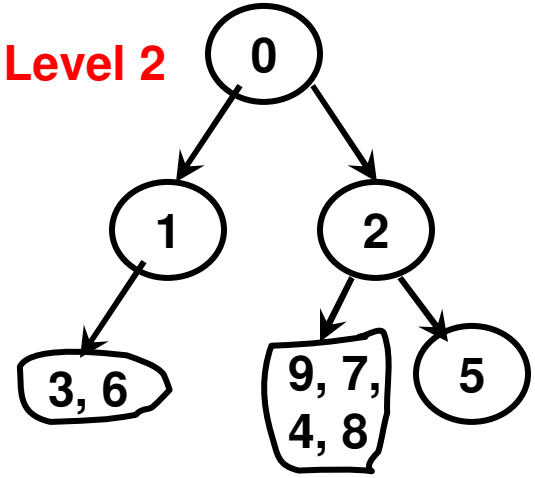
- Given the in order traversal sequence and the pre or post order traversal sequence for a binary tree, we can construct the binary tree.
- If given the pre order traversal, the first node in the sequence represents the overall root node. The nodes that appear before (after) the root node in the in order sequence are the nodes of the left (right) sub tree.
- If given the post order traversal, the last node in the sequence represents the overall root node. The nodes that appear before (after) the root node in the in order sequence are the nodes of the left (right) sub tree.
- We follow the above procedure in a recursive fashion.

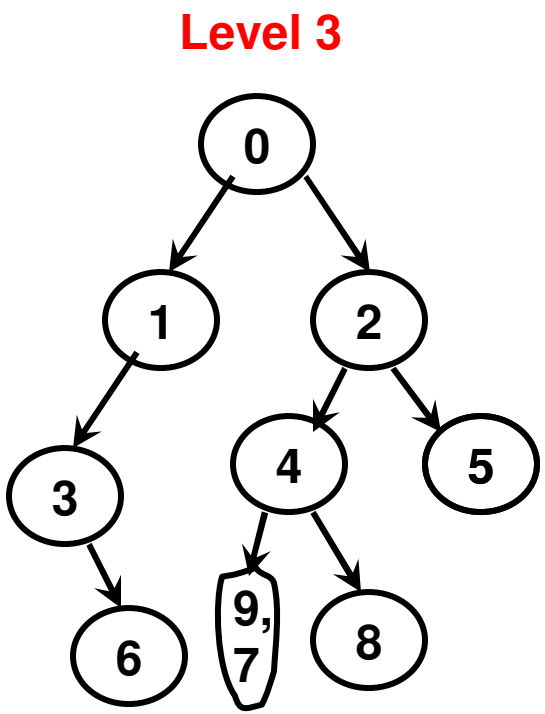
Constructing a Binary Tree: Example

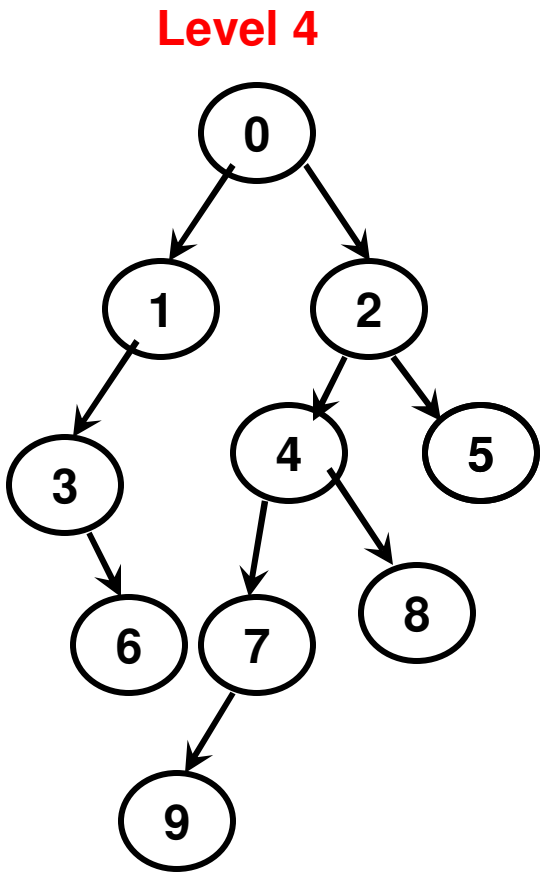
Post-order: 6, 3, 1, 9, 7, 8, 4, 5, 2, 0

In-order: 3, 6, 1, 0, 9, 7, 4, 8, 2, 5







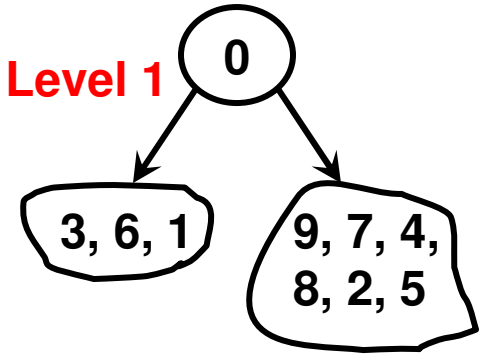


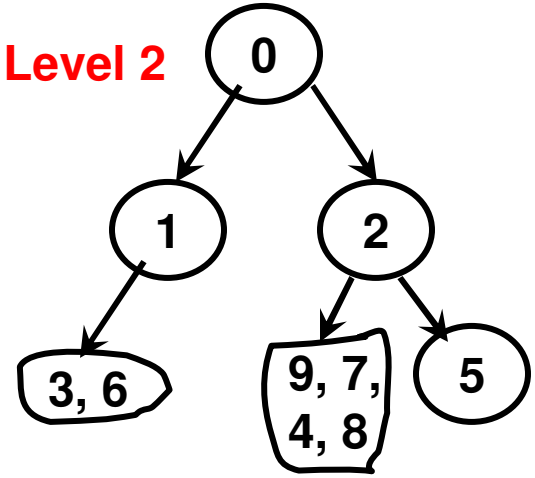
Final Binary Tree

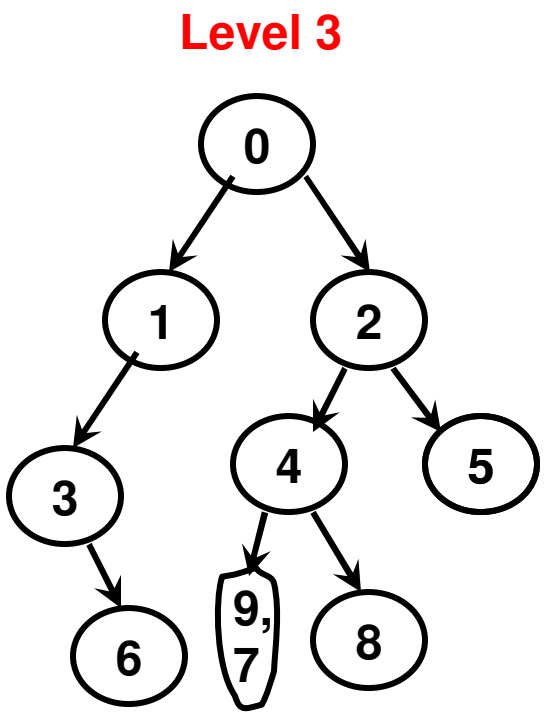
Constructing a Binary Tree: Example

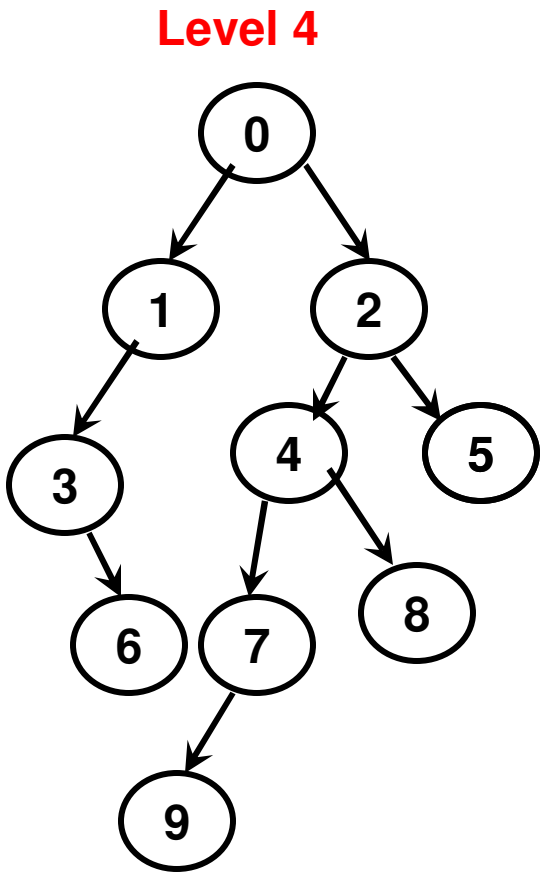
Pre-order: 0, 1, 3, 6, 2, 4, 7, 9, 8, 5

In-order: 3, 6, 1, 0, 9, 7, 4, 8, 2, 5









Final Binary Tree