Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 8

Binary Tree

Course Roadmap

Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Binary Tree

2- Traverse Operation

3- BFS vs. DFS for Binary Tree

4- Search Operation

5- Deletion Operation

6- Time Complexity & Space Complexity


Let's
START UP

Lecture Agenda

Section 1: Introduction to Binary Tree

Section 2: Traverse Operation

Section 3: BFS vs. DFS for Binary Tree

Section 4: Search Operation

Section 5: Deletion Operation

Section 6: Time Complexity & Space Complexity

Introduction to Binary Tree

- A Binary Tree is a special data structure used for data storage purposes. It has a special condition that each node can have a maximum of two children, and it has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.
- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. A recursive definition using just set theory notions is that a (non-empty) binary tree is a tuple (Left, Source, Right), where Left and Right are binary trees or the empty set and Source is a singleton set. Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.
- Why Trees?

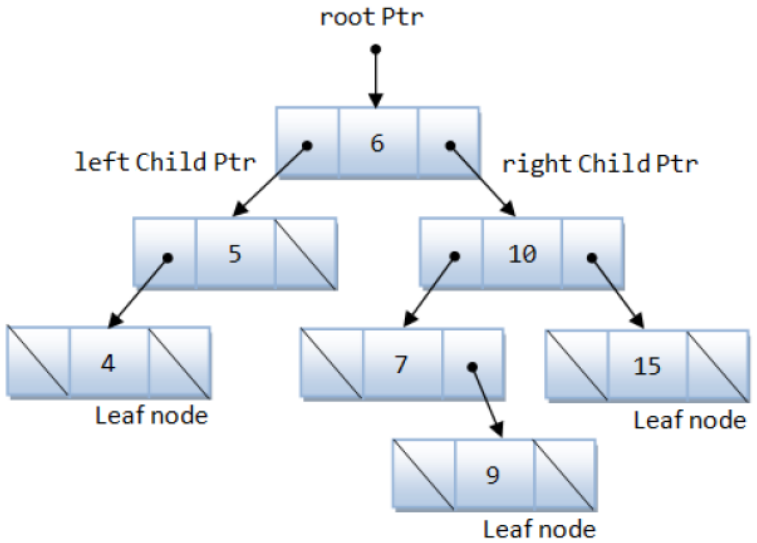
- One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer file system.
- If we organize keys in form of a tree (with some ordering e.g., Binary Search Tree BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays).
- We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self balancing search trees like AVL and Red-Black trees guarantee an upper bound of $\Theta(\log n)$ for insertion/deletion.

Introduction to Binary Tree

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges. Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure. A binary tree is a tree data structure in which each parent node can have at most two children.

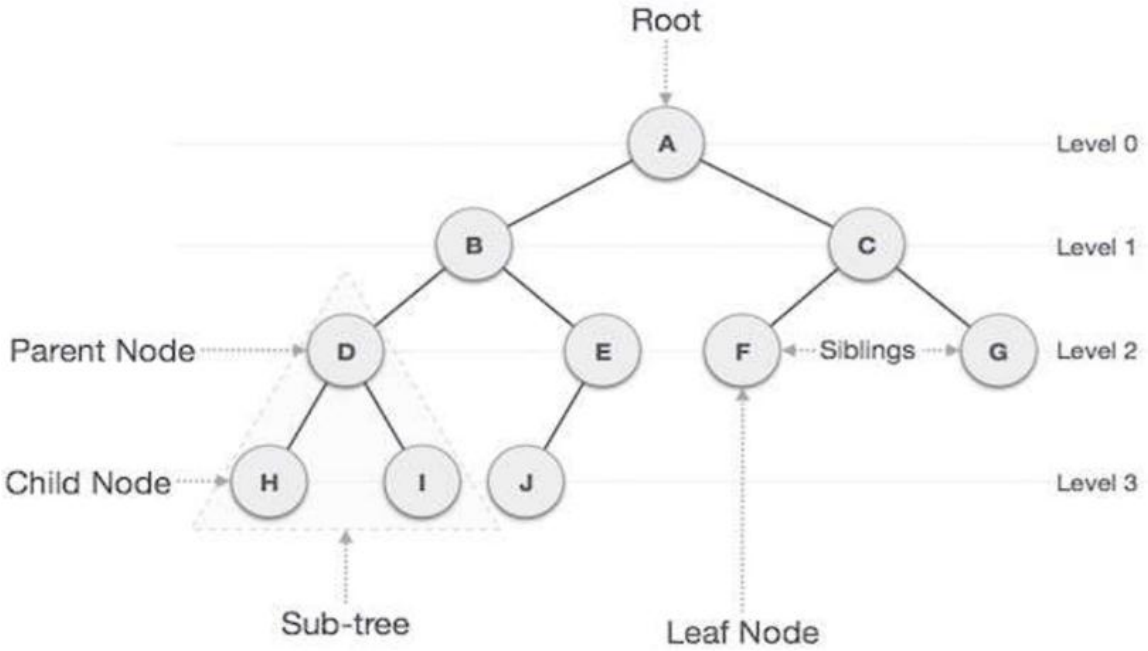
- A binary tree node

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



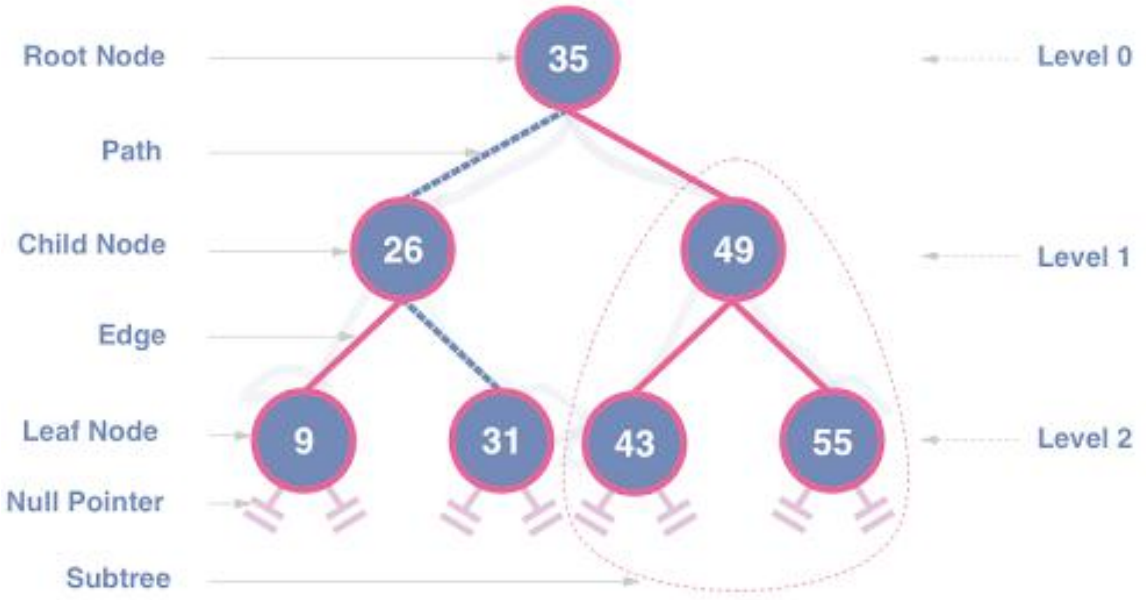
Binary Tree Terminology

- Node: it is an entity that contains a value and pointers to its child nodes.
- Leaf node: The node which does not have any child node.
- Internal node: The node which having at least a child node
- Root: The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent: Any node except the root node has one edge upward to a node called parent.
- Child: The node below a given node connected by its edge downward is called its child node.
- Depth: The distance between a node and the root.



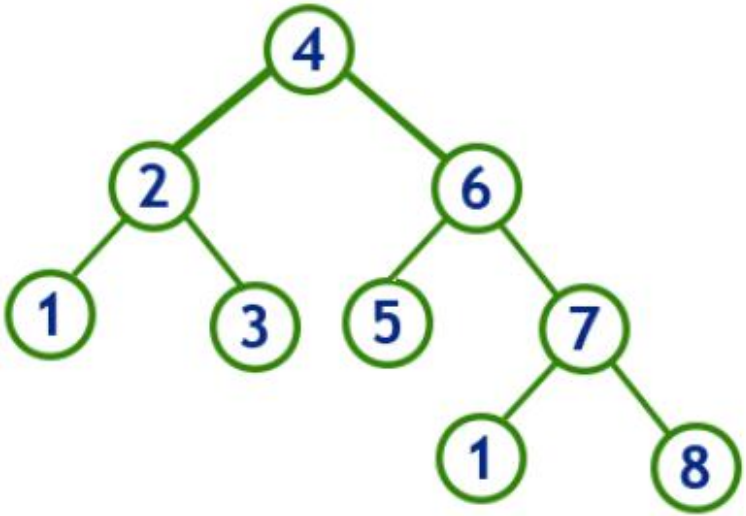
Binary Tree Terminology

- Edge: It is the link between any two nodes.
- Sub-tree: it represents the descendants of a node.
- Visiting: it refers to checking the value of a node when control is on the node.
- Traversing: it means passing through nodes in a specific order.
- Levels: Level of a node represents the generation of a node.
- Path: it refers to the sequence of nodes along the edges of a tree.
- Keys: it represents a value of node based on which a search operation is to be carried out for a node.

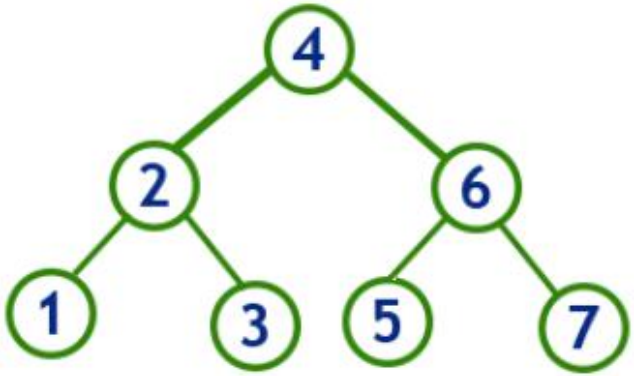


Types of Binary Tree

- **Full Binary Tree:** A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

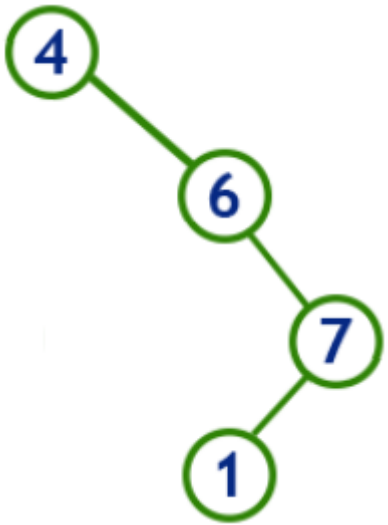


- **Perfect Binary Tree:** A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



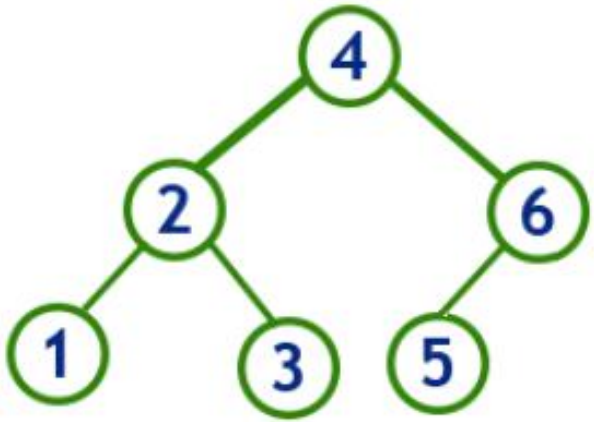
Types of Binary Tree

- **Degenerate or Pathological Tree:** A degenerate or pathological tree is the tree having a single child either left or right.



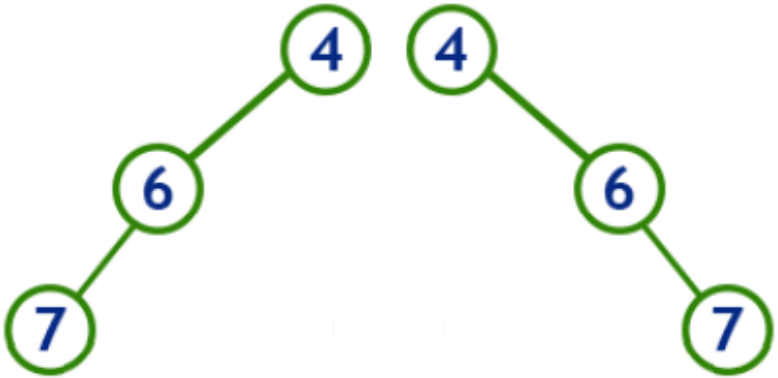
- **Complete Binary Tree:** A complete binary tree is just like a full binary tree, but with two major differences

  - Every level must be completely filled.
  - 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.

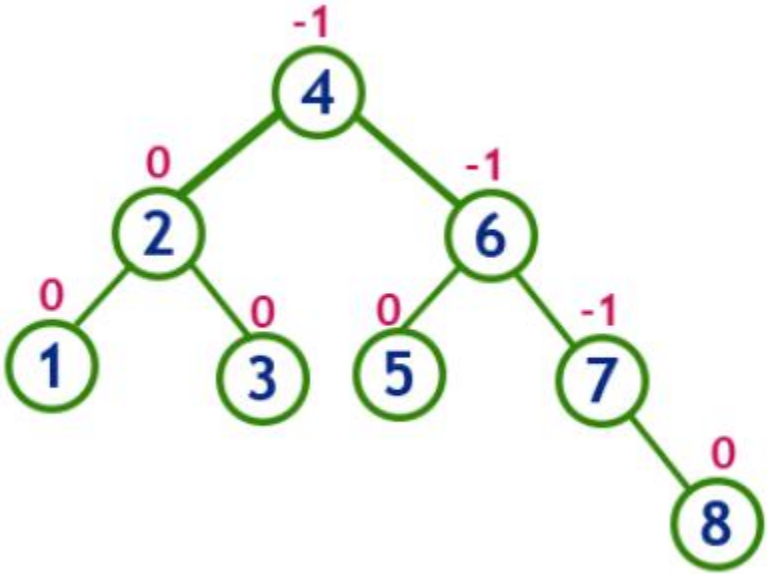


Types of Binary Tree

- **Skewed Binary Tree:** A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



- **Balanced Binary Tree:** It is a type of binary tree in which the difference between the left and the right subtree for each node is either 0 or 1.



Lecture Agenda

- ✓ Section 1: Introduction to Binary Tree
- **Section 2: Traverse Operation**
- Section 3: BFS vs. DFS for Binary Tree
- Section 4: Search Operation
- Section 5: Deletion Operation
- Section 6: Time Complexity & Space Complexity

Traverse Operation - Binary Tree

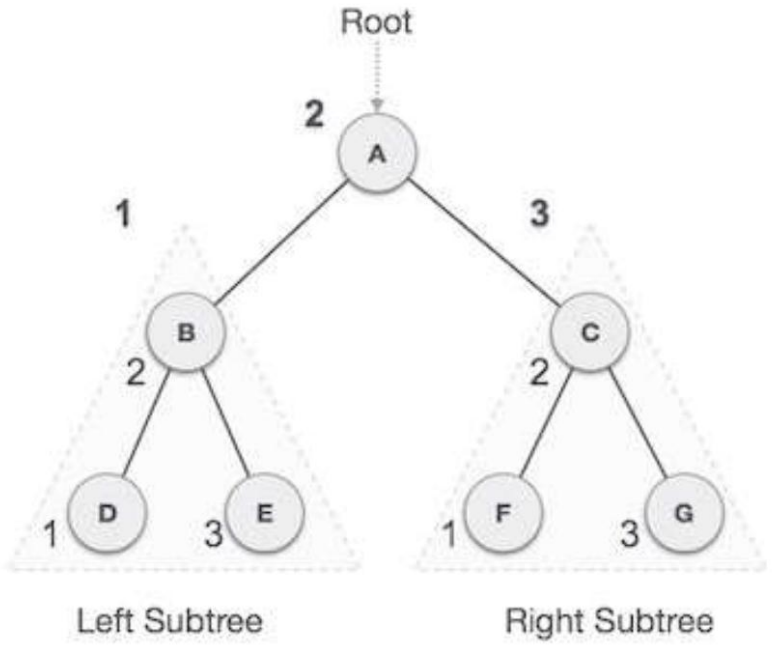
- Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree. Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.
- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in a tree. A Tree is typically traversed in two ways, BFS & DFS.
- Depth-First Search (DFS) Algorithm: It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.
- Depth First Traversals Types: there are three types of Depth First Traversal for binary trees, In-order Traversal, Pre-order Traversal, and Post-order Traversal.

Traverse Operation - Binary Tree (in-order)

- In-order Traversal is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will visit that current node and go to the left-most node of its right subtree (if exists). [Left - Root - Right]

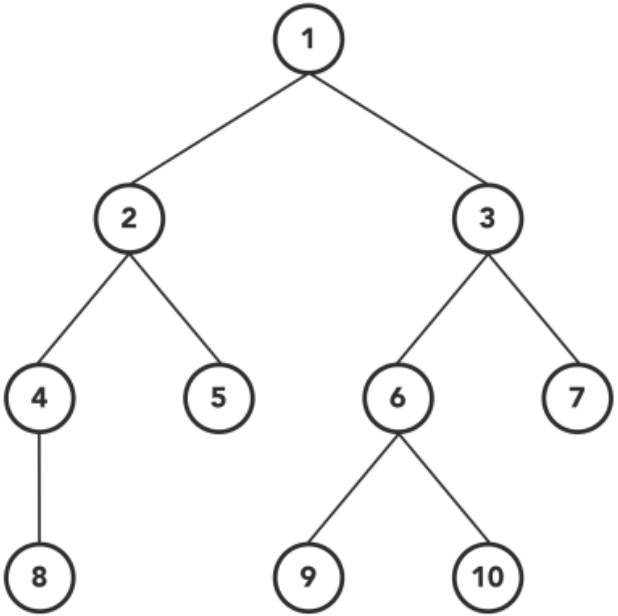
- ***Traverse Algorithm***:

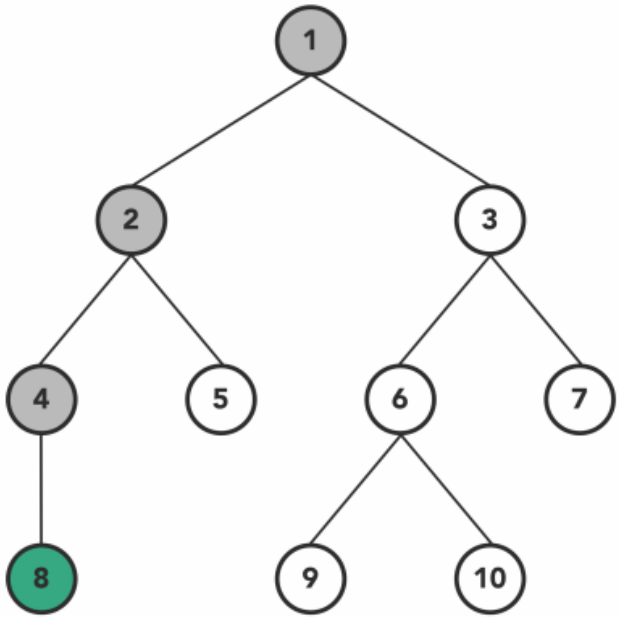
1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***print curr node data***
4. ***go to step 1 with right node of curr***



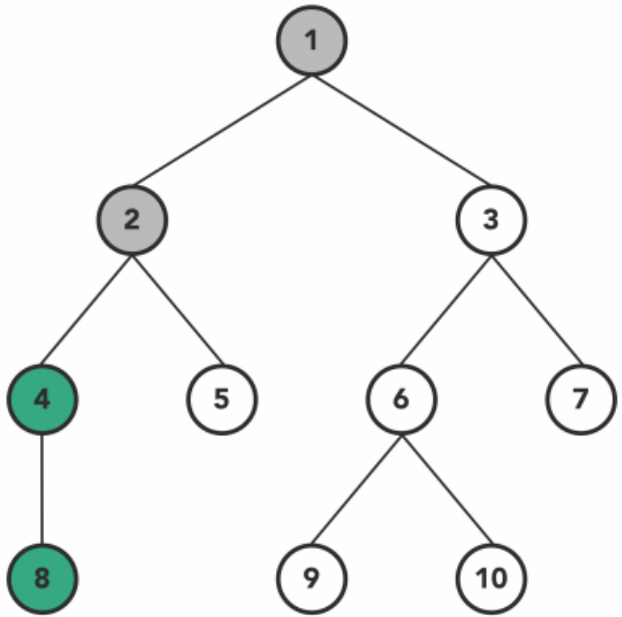
Traverse Operation - Binary Tree (in-order)

- Inorder Traversal: [Left - Root - Right]





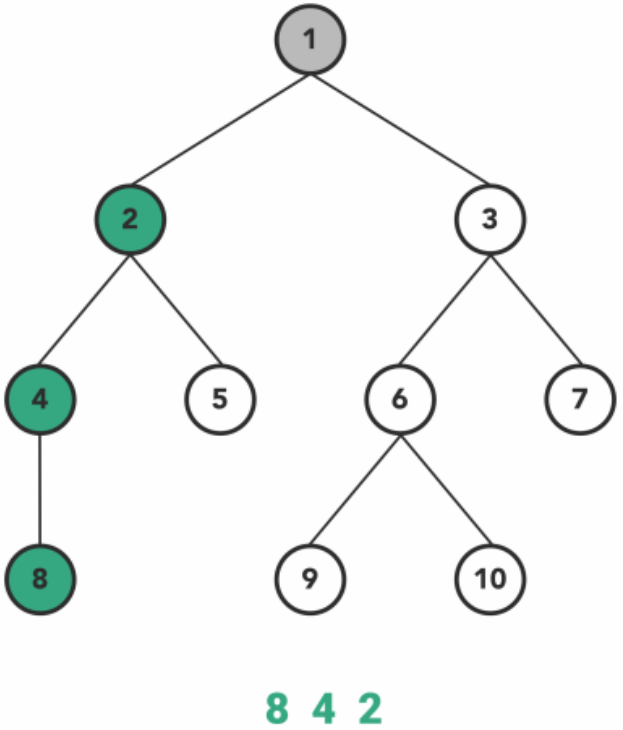
8

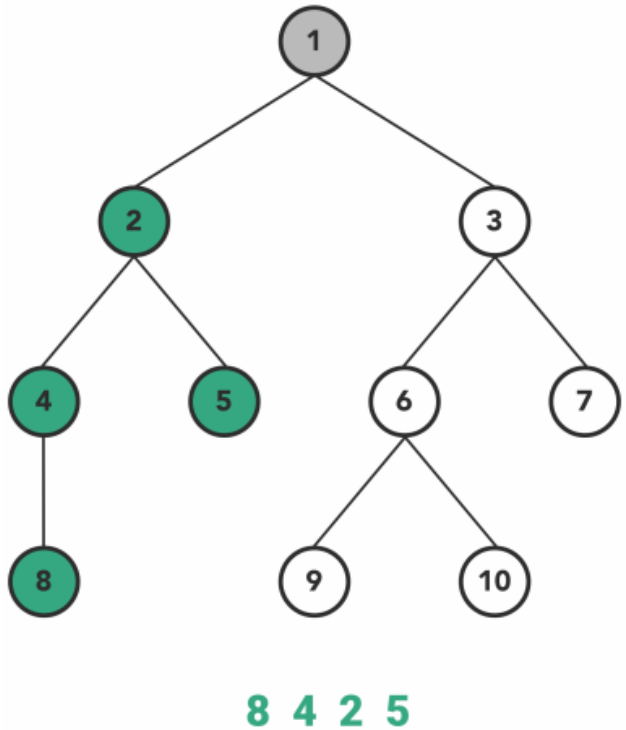


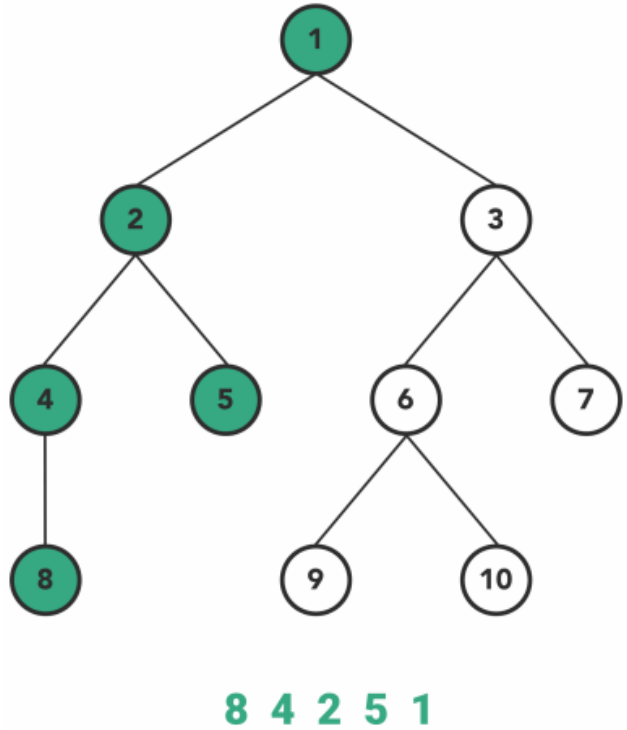
8 4

Traverse Operation - Binary Tree (in-order)

- Inorder Traversal: [Left - Root - Right]

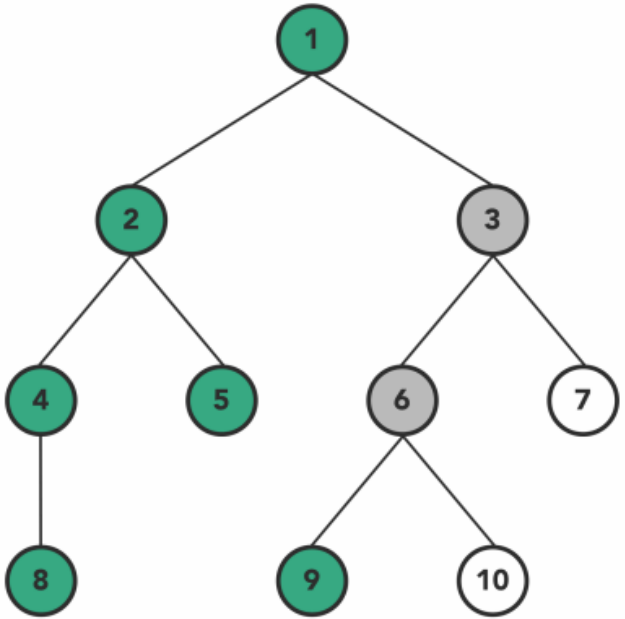




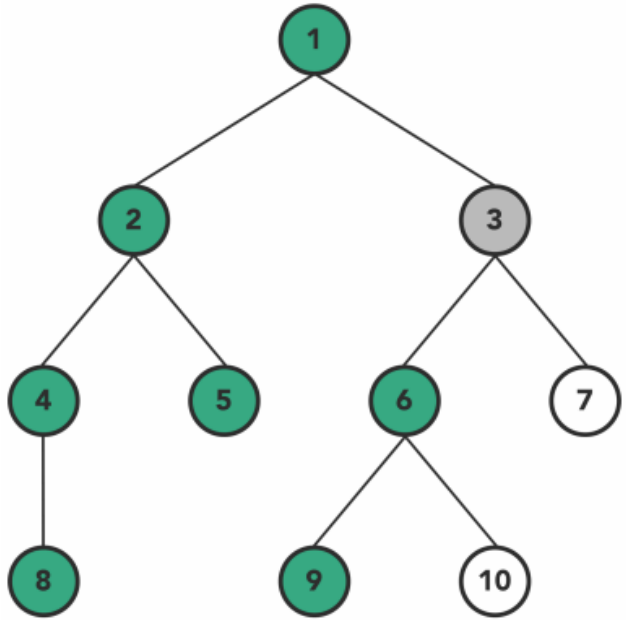


Traverse Operation - Binary Tree (in-order)

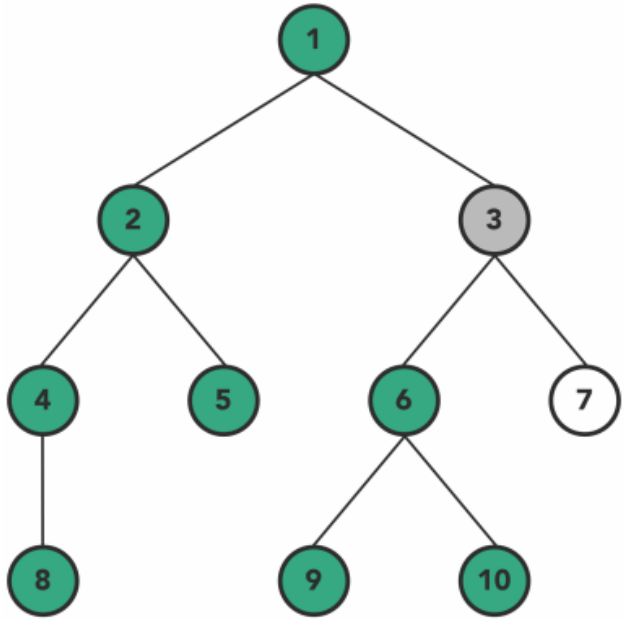
- Inorder Traversal: [Left - Root - Right]



8 4 2 5 1 9



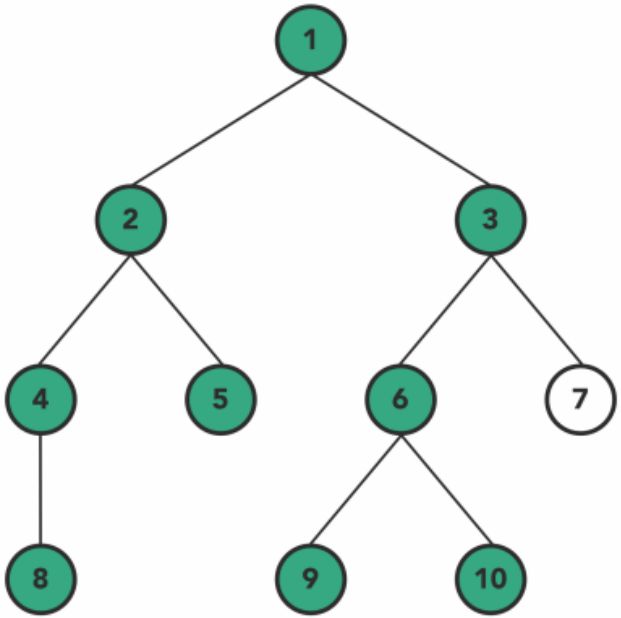
8 4 2 5 1 9 6



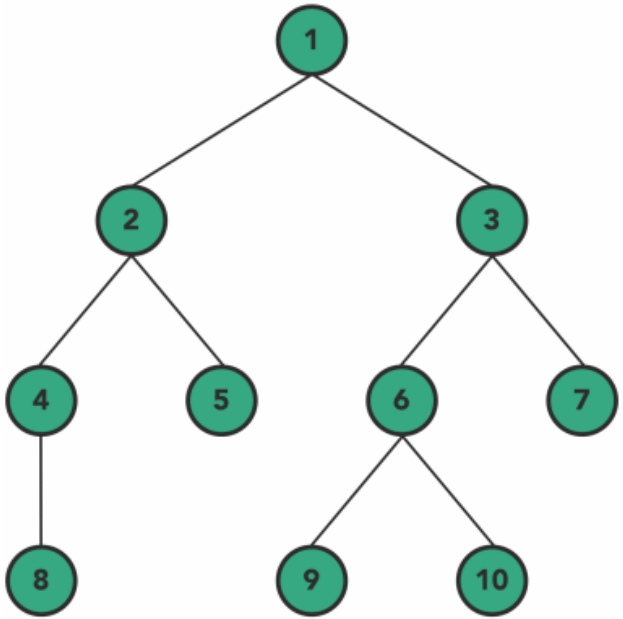
8 4 2 5 1 9 6 10

Traverse Operation - Binary Tree (in-order)

- Inorder Traversal: [Left - Root - Right]



8 4 2 5 1 9 6 10 3



8 4 2 5 1 9 6 10 3 7

Traverse Operation - Binary Tree (in-order)

```cpp
// This function do inorder traversal of the binary tree
void inOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition of inorder traversal
    inOrder(curr->left);
    cout << curr->data << ' ';
    inOrder(curr->right);
}
```

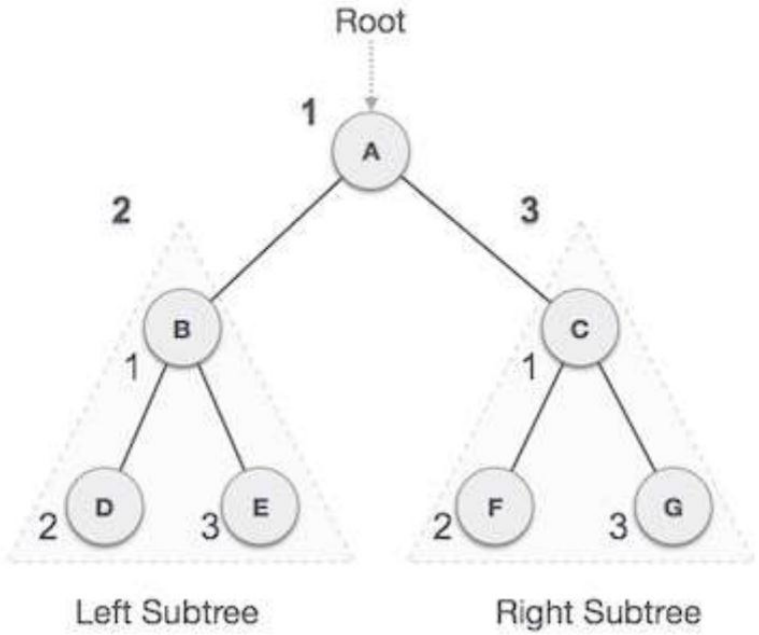
Binary-Tree.cpp

Traverse Operation - Binary Tree (pre-order)

- Pre-order Traversal: is the one of DFS Traversal of the tree. Therefore, we visit the current node first and then goes to the left sub-tree. After covering every node of the left sub-tree, we will move towards the right sub-tree and visit in a similar fashion. [Root - Left - Right]

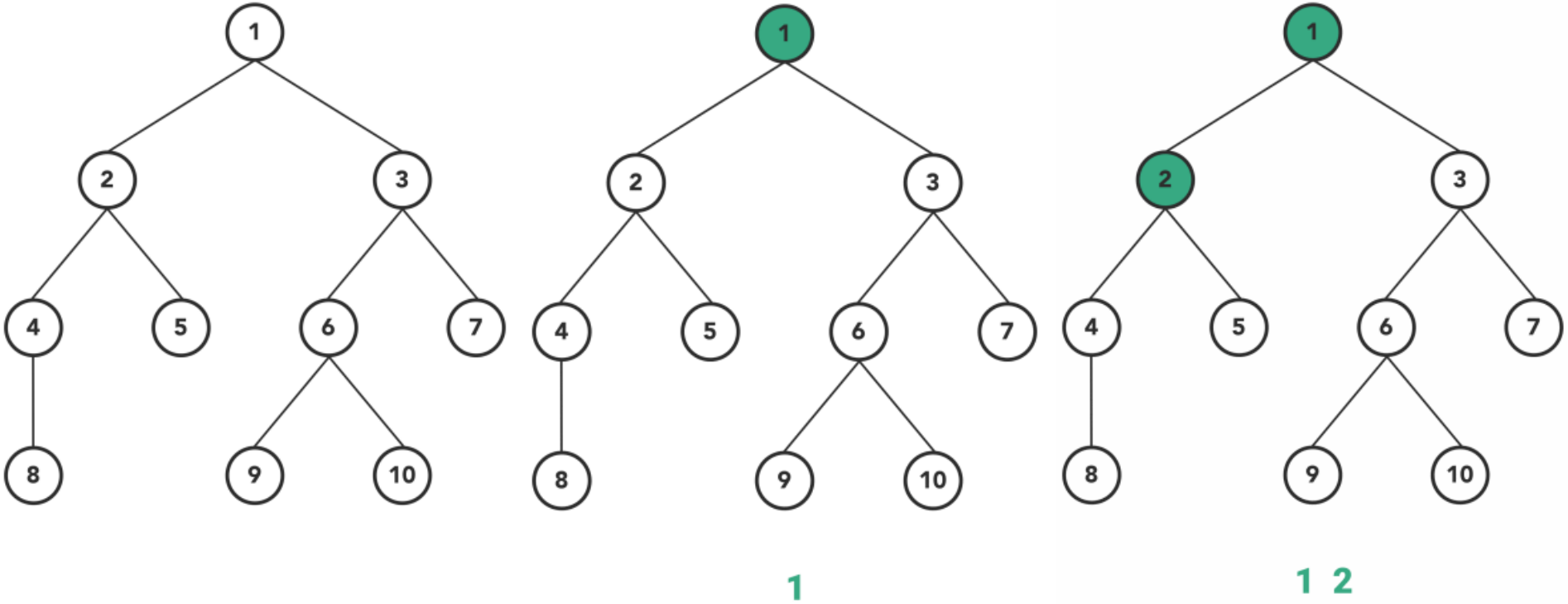
- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***print curr node data***
3. ***go to step 1 with left node of curr***
4. ***go to step 1 with right node of curr***



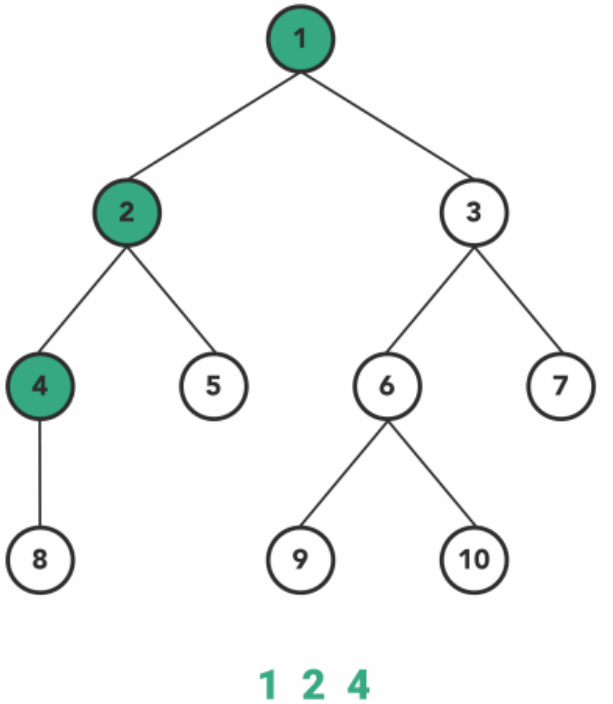
Traverse Operation - Binary Tree (pre-order)

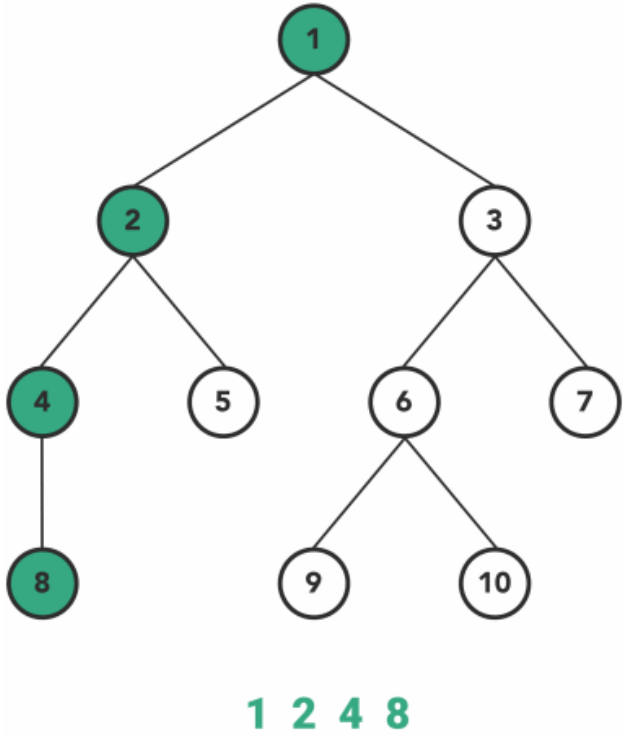
- Preorder Traversal: [Root - Left - Right]

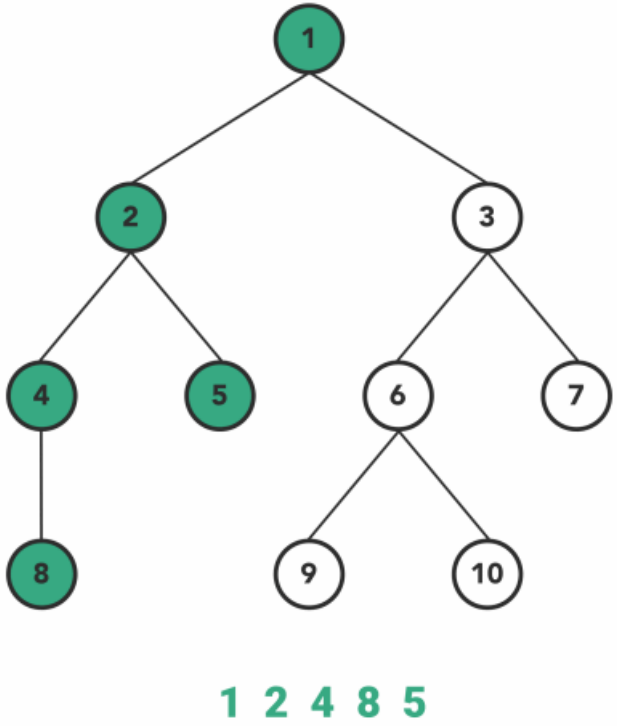


Traverse Operation - Binary Tree (pre-order)

- Preorder Traversal: [Root - Left - Right]

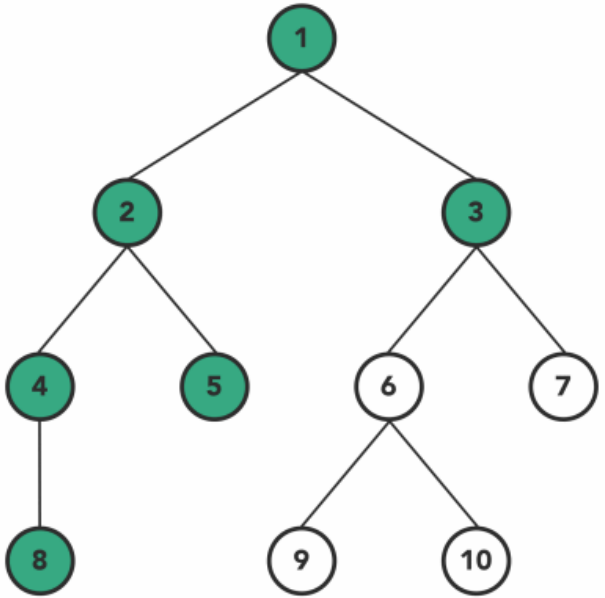




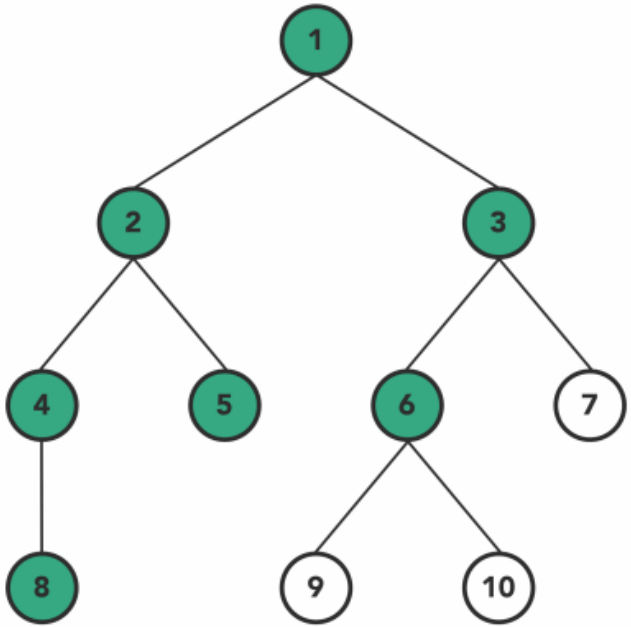


Traverse Operation - Binary Tree (pre-order)

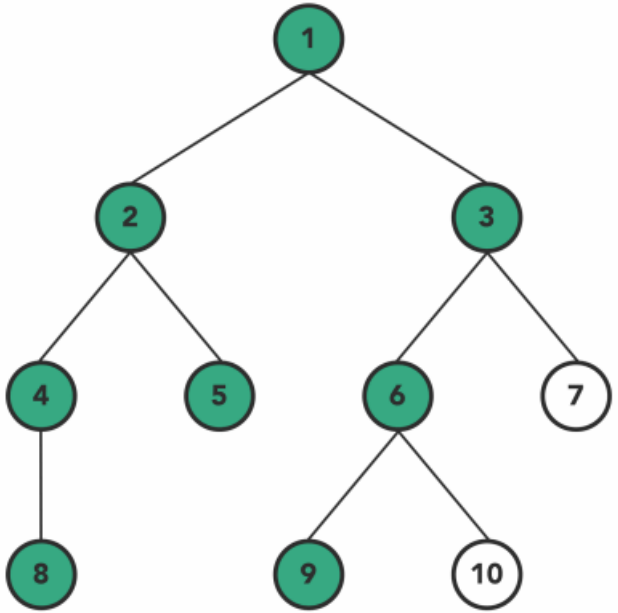
- Preorder Traversal: [Root - Left - Right]



1 2 4 8 5 3



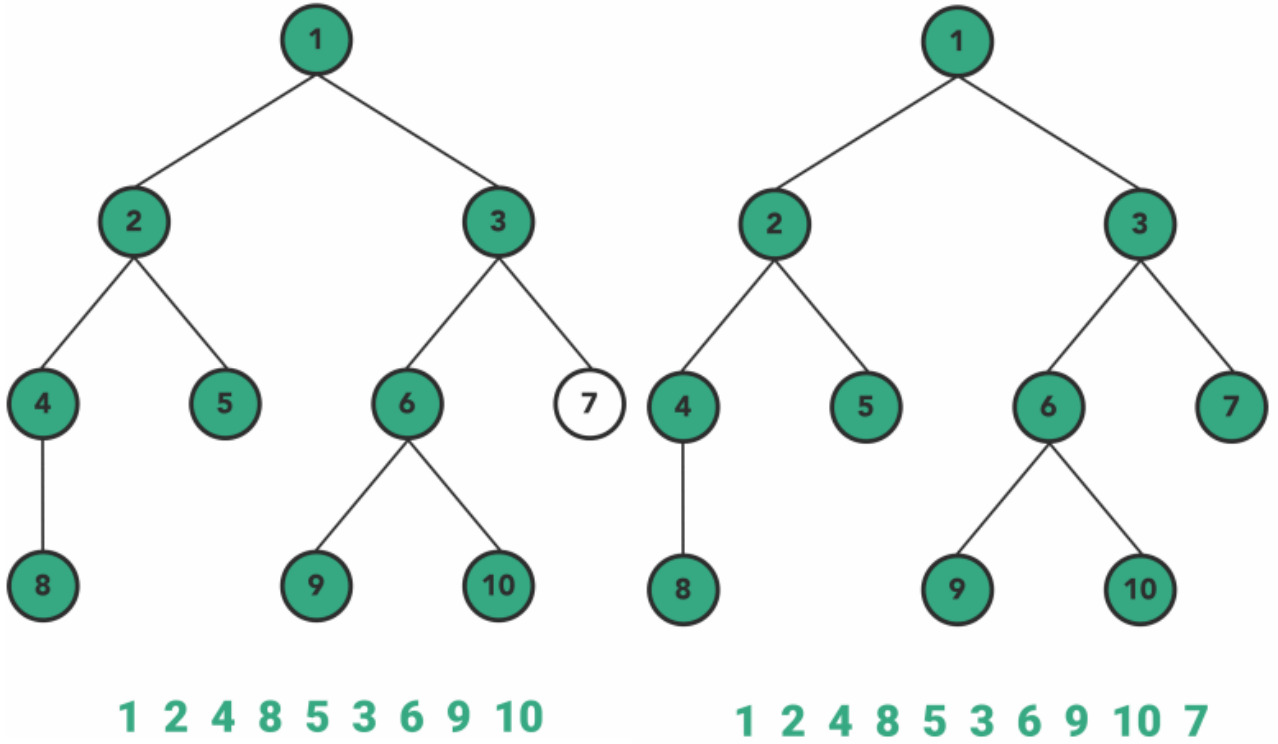
1 2 4 8 5 3 6



1 2 4 8 5 3 6 9

Traverse Operation - Binary Tree (pre-order)

- Preorder Traversal: [Root - Left - Right]



Traverse Operation - Binary Tree (pre-order)

```
// This function do preorder traversal of the binary tree
void preOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition of preOrder traversal
    cout << curr->data << ' ';
    preOrder(curr->left);
    preOrder(curr->right);
}
```

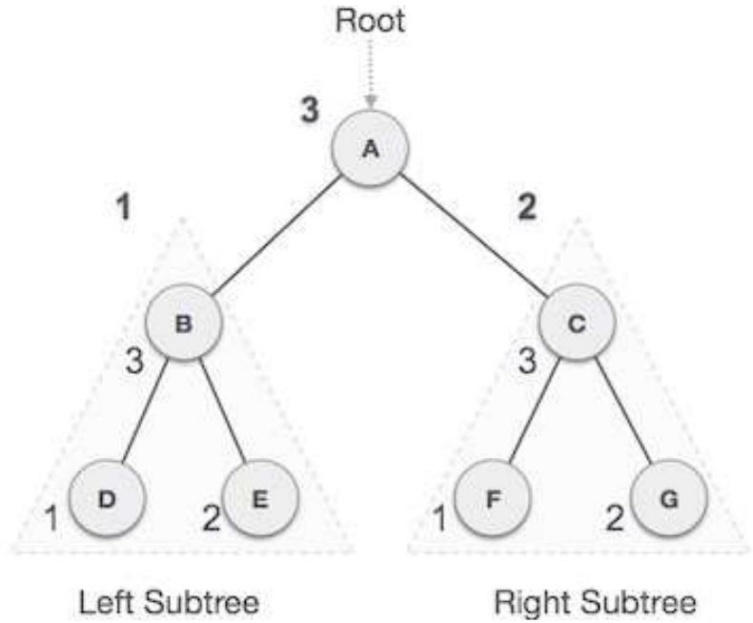
Binary-Tree.cpp

Traverse Operation - Binary Tree (post-order)

- Post-order Traversal: is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will go to the left-most node of its right subtree (if exists) and finally we will visit that current node. [Left - Right - Root]

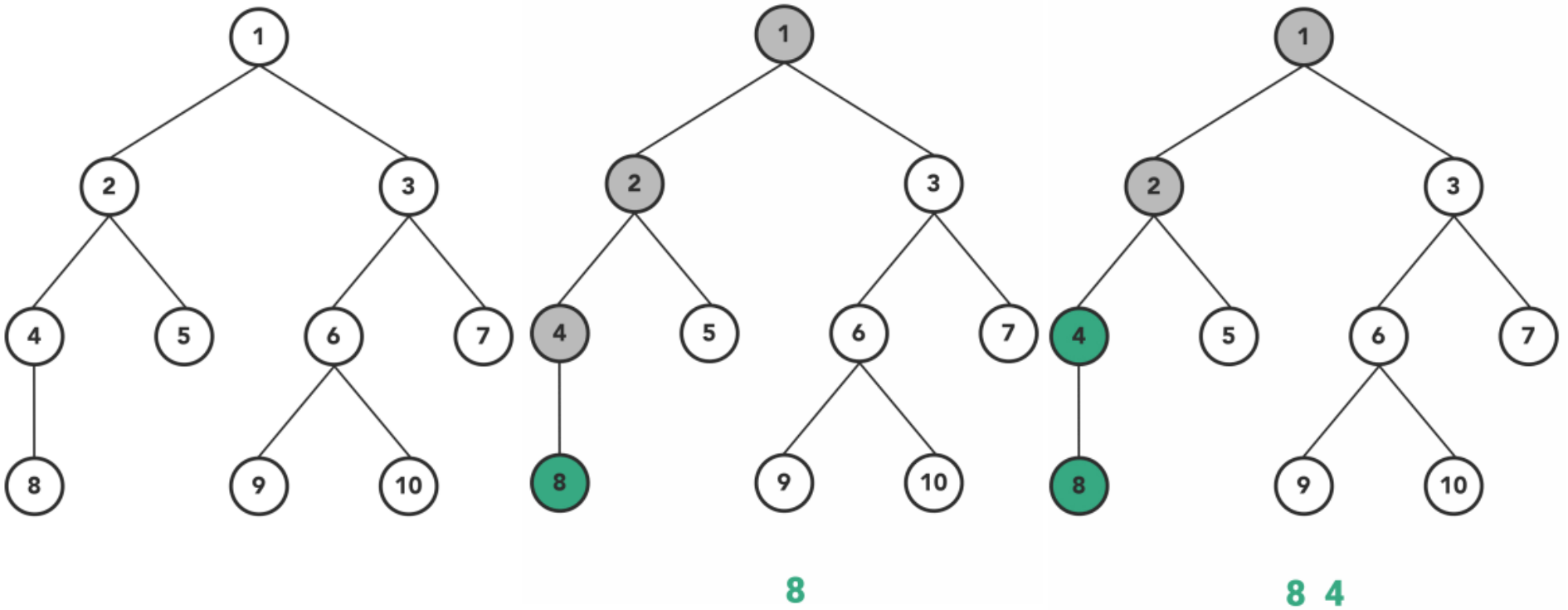
- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***go to step 1 with right node of curr***
4. ***print curr node data***



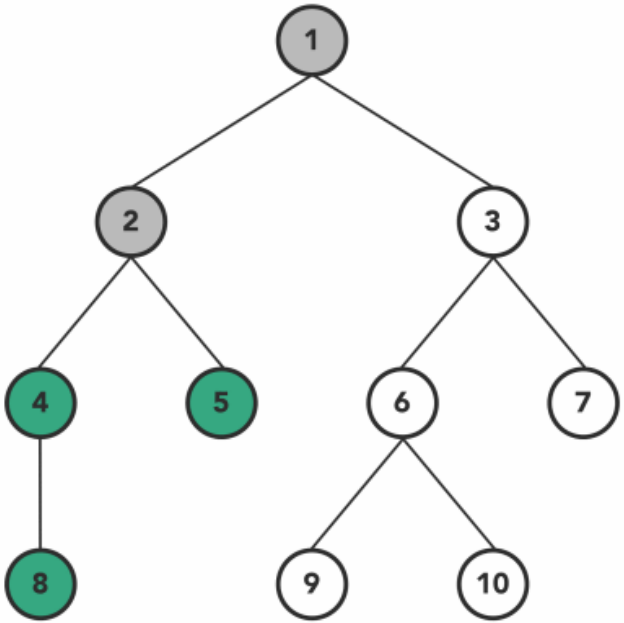
Traverse Operation - Binary Tree (post-order)

- Postorder Traversal: [Left - Right - Root]

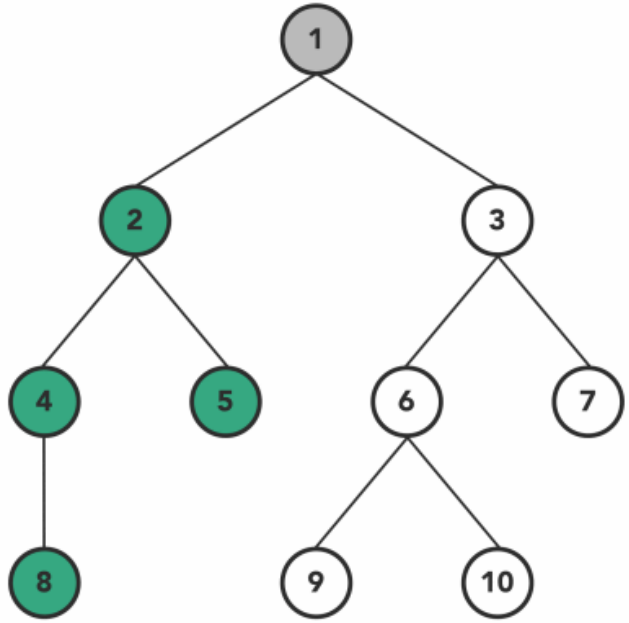


Traverse Operation - Binary Tree (post-order)

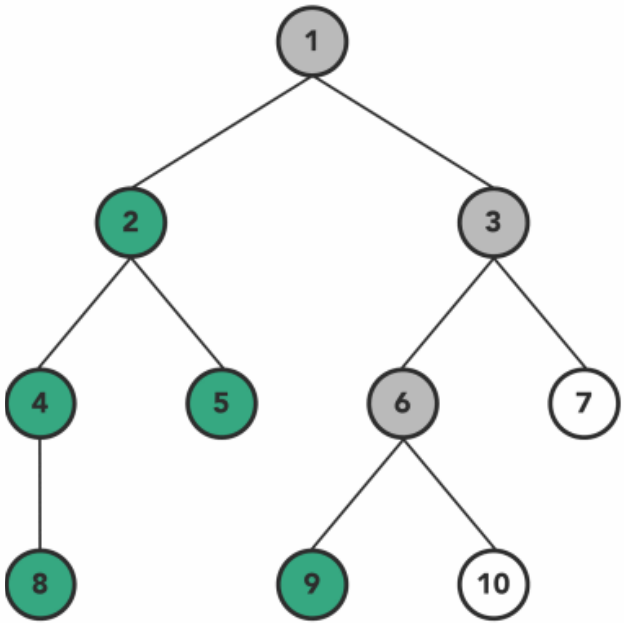
- Postorder Traversal: [Left - Right - Root]



8 4 5



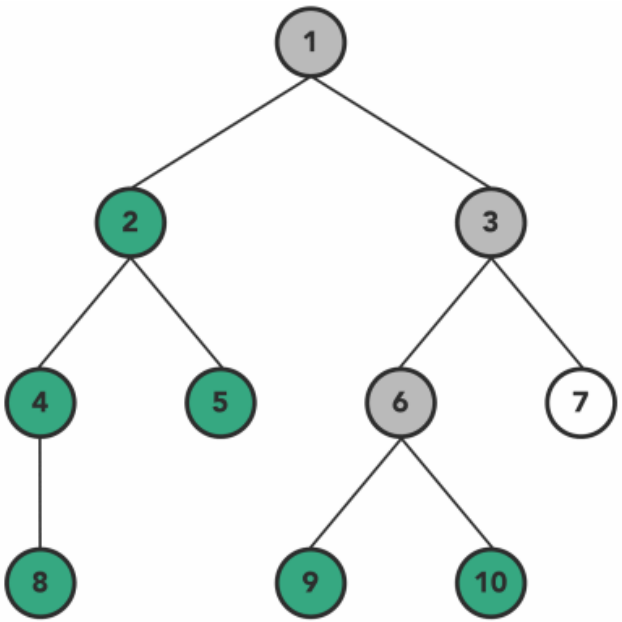
8 4 5 2



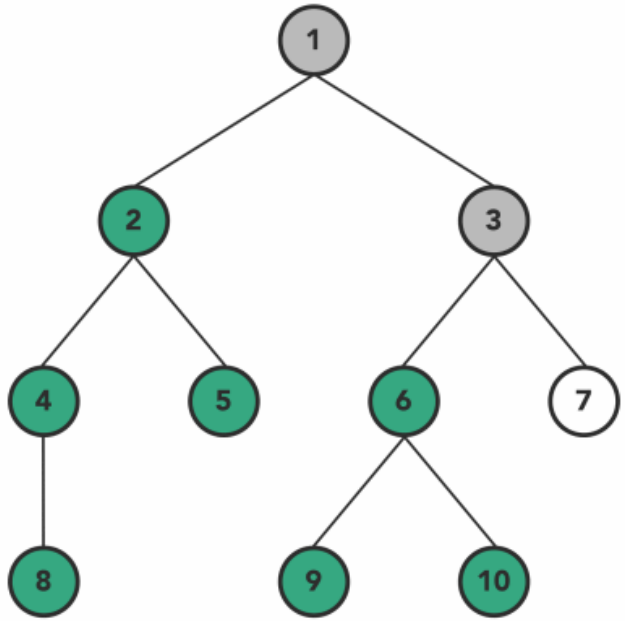
8 4 5 2 9

Traverse Operation - Binary Tree (post-order)

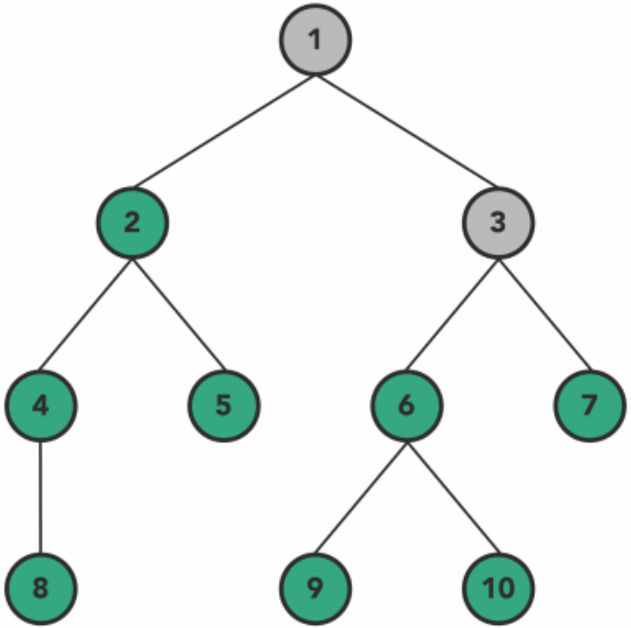
- Postorder Traversal: [Left - Right - Root]



8 4 5 2 9 10



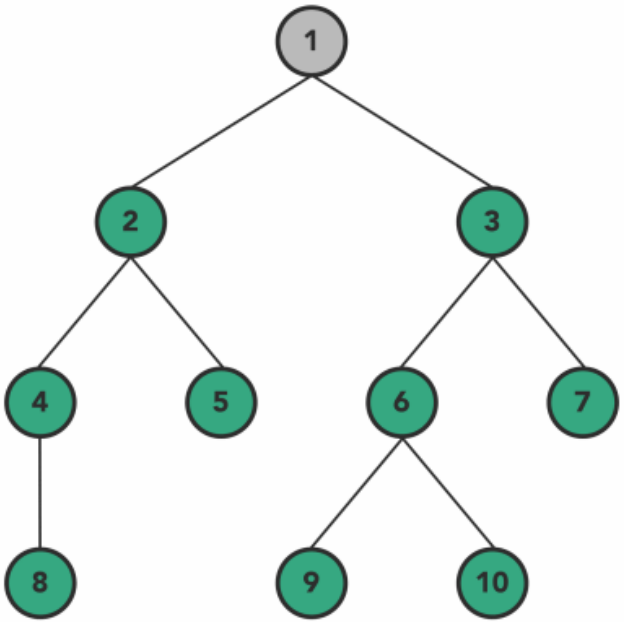
8 4 5 2 9 10 6



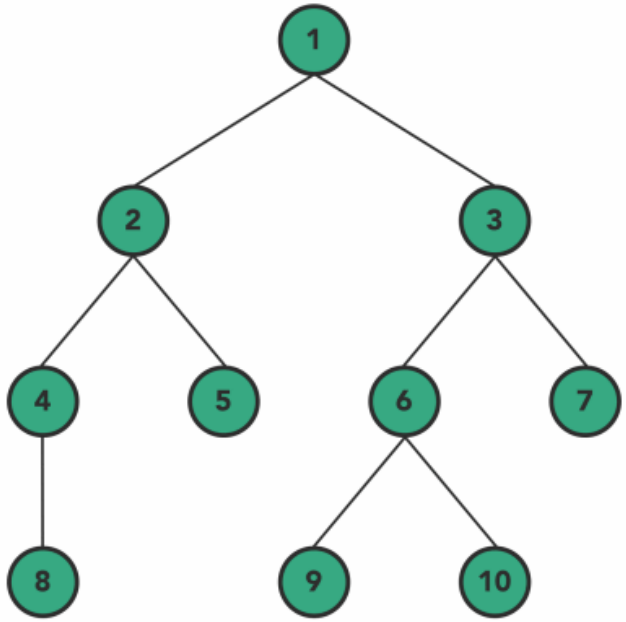
8 4 5 2 9 10 6 7

Traverse Operation - Binary Tree (post-order)

- Postorder Traversal: [Left - Right - Root]



8 4 5 2 9 10 6 7 3



8 4 5 2 9 10 6 7 3 1

Traverse Operation - Binary Tree (post-order)

```cpp
// This function do postorder traversal of the binary tree
void postOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition of postorder traversal
    postOrder(curr->left);
    postOrder(curr->right);
    cout << curr->data << ' ';
}
```

Binary-Tree.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Binary Tree
- ✔ Section 2: Traverse Operation
- **Section 3: BFS vs. DFS for Binary Tree**
- Section 4: Search Operation
- Section 5: Deletion Operation
- Section 6: Time Complexity & Space Complexity

BFS vs. DFS for Binary Tree

- Depth-First Search (DFS) Algorithm: It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.

  Breadth-First Search (BFS) Algorithm: It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.

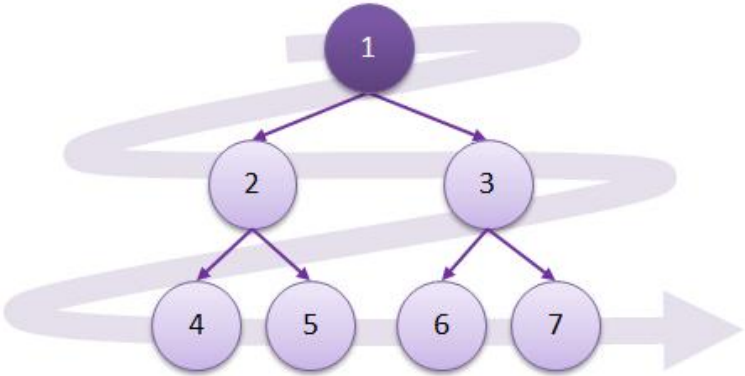
- Level Order Traversal: This is a different traversal than what we have covered above. Level order traversal follows BFS(Breadth-First Search) to visit/modify every node of the tree. As BFS suggests, the breadth of the tree takes priority first and then move to depth. In simple words, we will visit all the nodes present at the same level one-by-one from left to right and then move to the next level to visit all the nodes of that level.

Traverse Operation - Binary Tree (BFS)

- BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

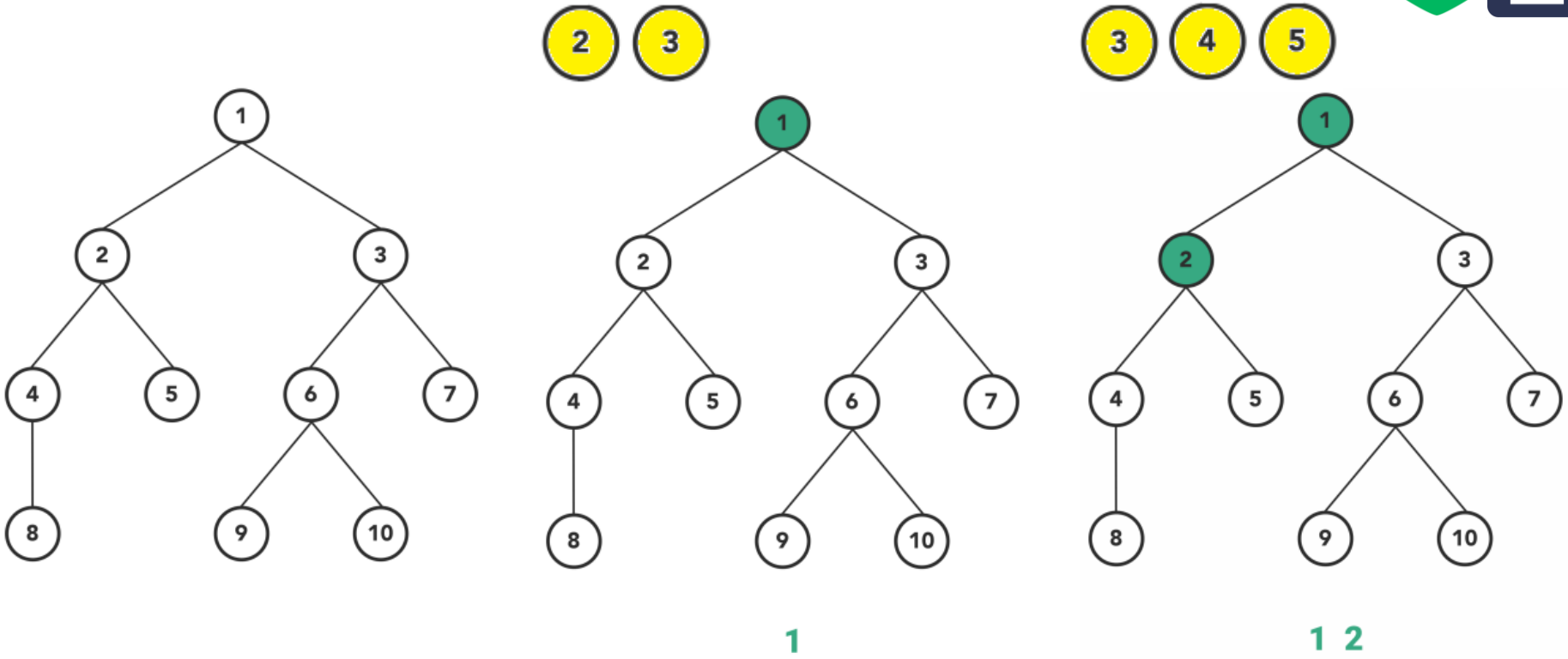
- ***Traverse Algorithm***:

1. $curr = root$
2. $print\ curr\ node\ data$
3. $push\ curr\ left\ (if\ exists)$
4. $push\ curr\ right\ (if\ exists)$
5. $curr = front()$
6. $pop()$
7. $repeat\ step\ 2\ while\ curr\ not\ NULL$



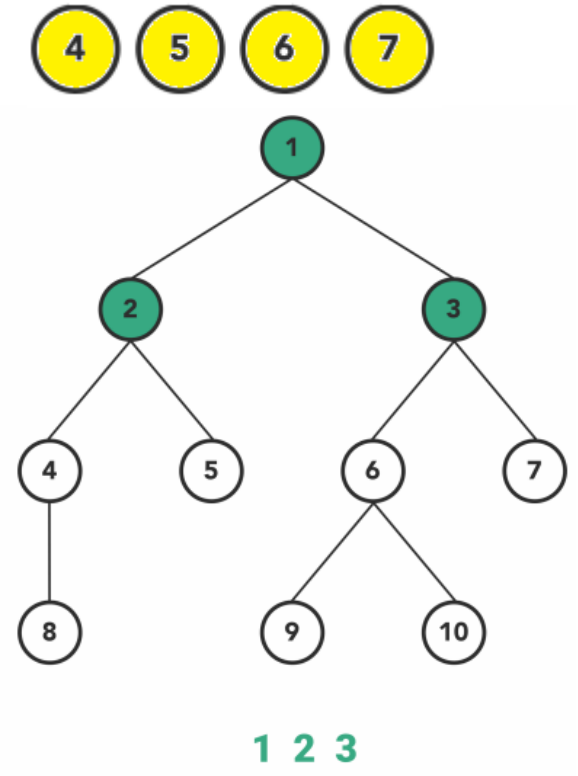
Traverse Operation - Binary Tree (BFS)

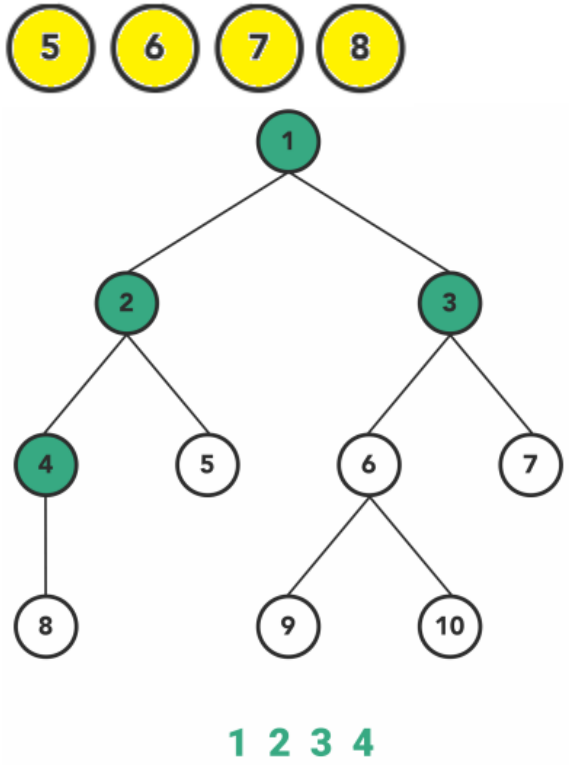
- Breadth First Search: Level Order

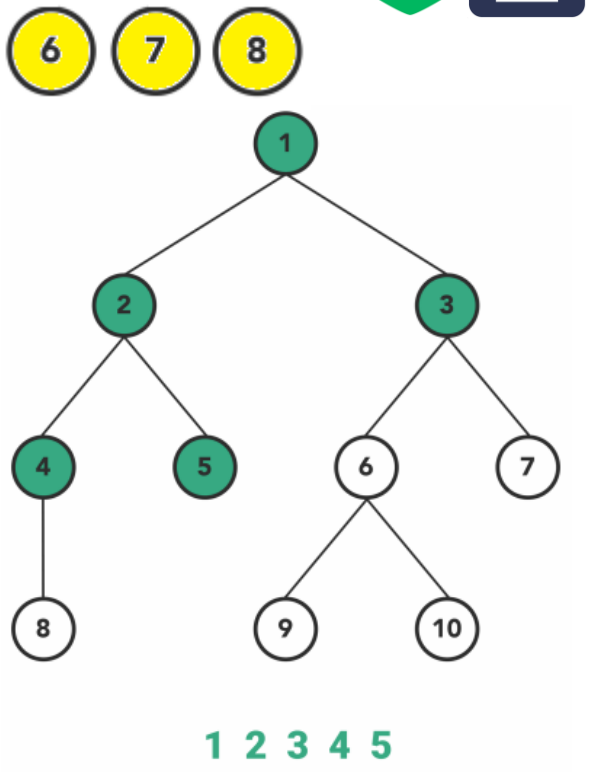


Traverse Operation - Binary Tree (BFS)

- Breadth First Search: Level Order

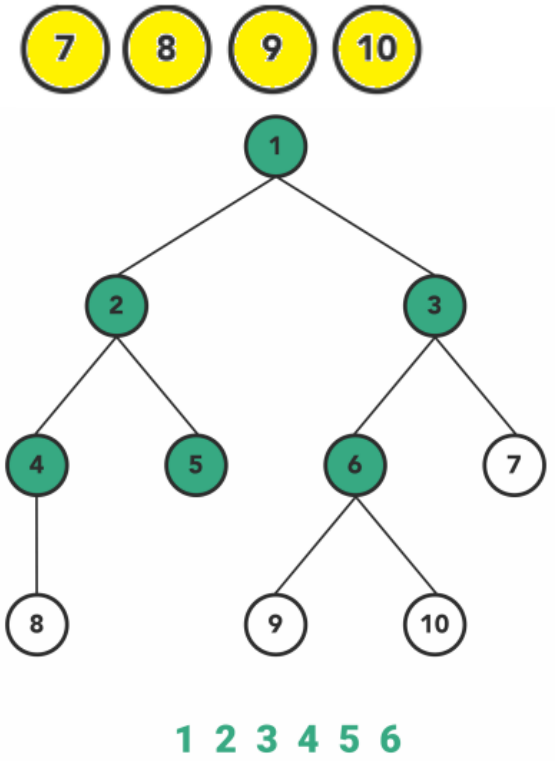


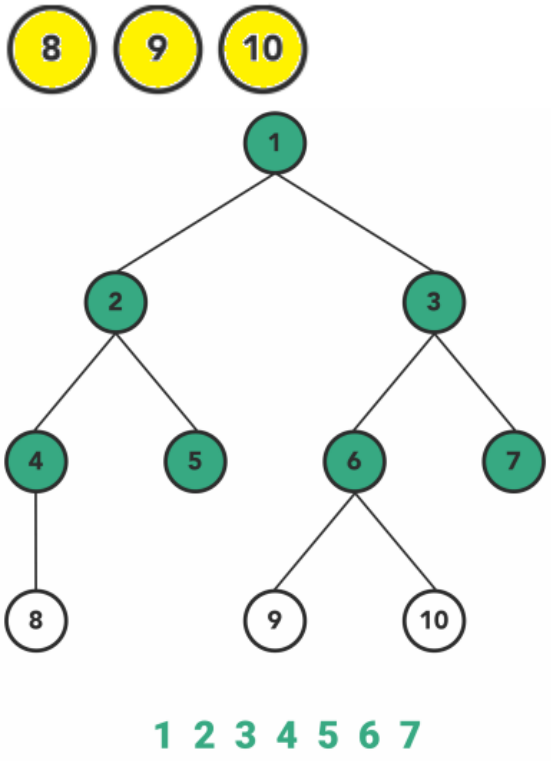


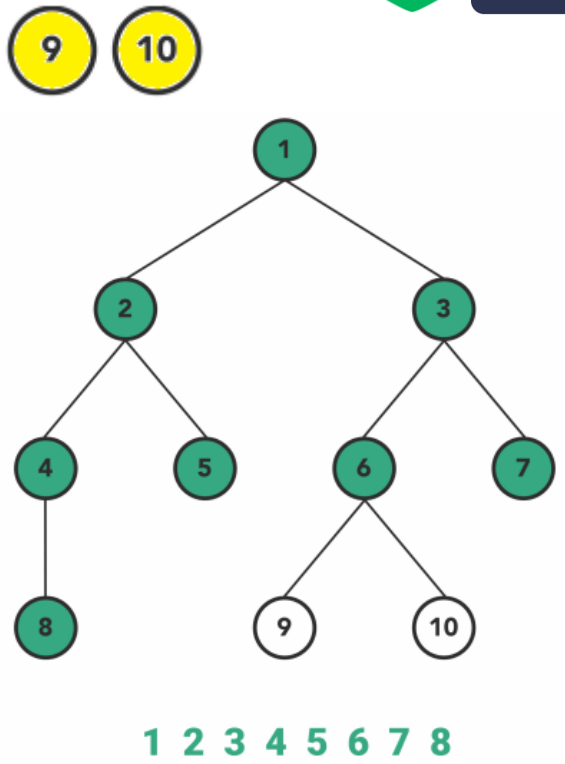


Traverse Operation - Binary Tree (BFS)

- Breadth First Search: Level Order

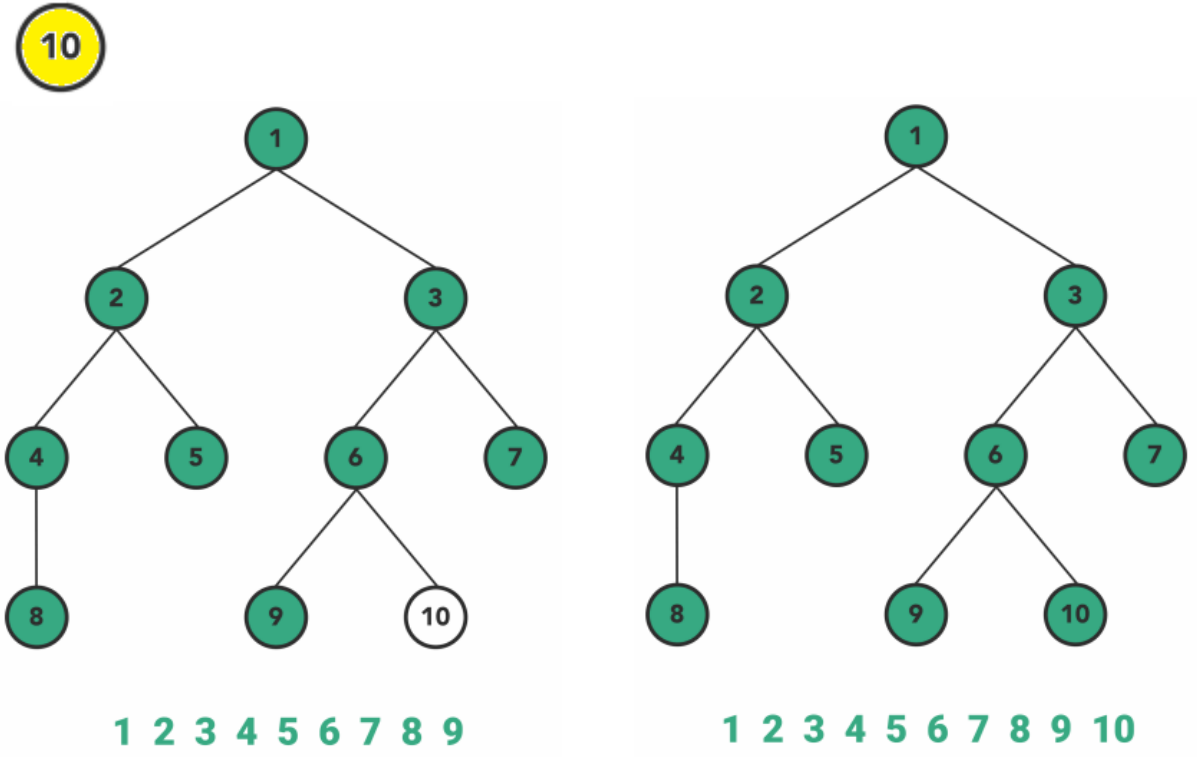






Traverse Operation - Binary Tree (BFS)

- Breadth First Search: Level Order



BFS Method - Queue in Binary Tree BFS

```
// This function do breadth first traversal of the binary tree
void bfs() {
    if (root == NULL)
        return;
    // starting with the root of the binary tree
    push(root);
    // iterate on the nodes till reach all node in the binary tree
    while (head != NULL) {
        // dequeue the front node and make it curr
        node* curr = front();
        pop();
        cout << curr->data << ' ';
        // enqueue the left child
        if (curr->left != NULL)
            push(curr->left);
        // enqueue the right child
        if (curr->right != NULL)
            push(curr->right);
    }
}
```

Binary-Tree-BFS.cpp

Initialize Queue in Binary Tree - BFS

```cpp
// A queue node
struct q_node {
    node* data;
    q_node* prev;
    q_node* next;
};

// Initialize a global pointers for head and tail
q_node* head;
q_node* tail;
```

Binary-Tree-BFS.cpp

Push Operations - Queue in Binary Tree BFS

```
// This function adds a node at the begin of the queue
void push(node* new_data) {
    // allocate new node and put it's data
    q_node* new_node = new q_node();
    new_node->data = new_data;
    // check if the queue is empty
    if (head == NULL) {
        head = new_node;
        tail = new_node;
    }
    // otherwise reach the end of the queue
    else {
        // set the next of the last node to be the new node and vice versa
        tail->next = new_node;
        new_node->prev = tail;
        // set the new node as a tail
        tail = new_node;
    }
}
```

Binary-Tree-BFS.cpp

Pop Operations - Queue in Binary Tree BFS

```
// This function deletes the first node in the queue
void pop() {
    // check if the queue is empty
    if (head == NULL)
        return;
    // get the node which it will be deleted
    q_node* temp_node = head;
    // check if the queue has only one node
    if (head == tail) {
        delete(temp_node); // delete the temp node
        head = tail = NULL;
    }
    // otherwise the queue has nodes more than one
    else {
        // shift the head to be the next node
        head = head->next;
        head->prev = NULL;
        delete(temp_node); // delete the temp node
    }
}
```

Binary-Tree-BFS.cpp

Front & Back Operations - Queue in Binary Tree BFS

```
// This function returns the value of the first node in the queue
node* front() {
    // check if the queue is empty
    // to return the biggest integer value as an invalid value
    if (head == NULL)
        return NULL;
    // otherwise return the real value
    else
        return head->data;
}
// This function returns the value of the last node in the queue
node* back() {
    // check if the queue is empty
    // to return the biggest integer value as an invalid value
    if (tail == NULL)
        return NULL;
    // otherwise return the real value
    else
        return tail->data;
}
```

Binary-Tree-BFS.cpp

Lecture Agenda

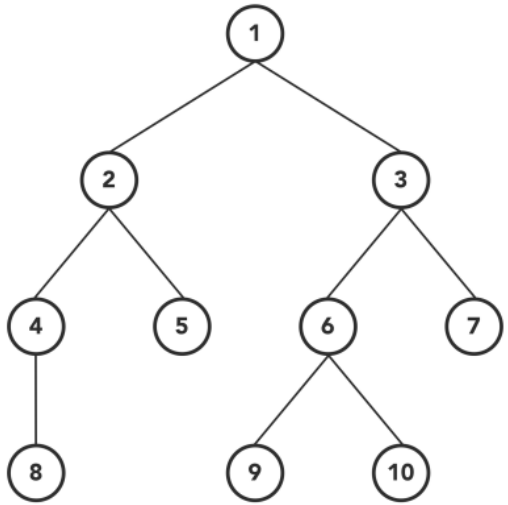
- ✔ Section 1: Introduction to Binary Tree
- ✔ Section 2: Traverse Operation
- ✔ Section 3: BFS vs. DFS for Binary Tree
- **Section 4: Search Operation**
- Section 5: Deletion Operation
- Section 6: Time Complexity & Space Complexity

Search Operation in Binary Tree

- The idea is to use any of the tree traversals to traverse the tree and while traversing check if the current node matches with the given node. Return True if any node matches with the given node and stop traversing further and if the tree is completely traversed and none of the node matches with the given node then Return False.
- We can search in the tree with other traversals also with extra space complexity but why should we go for the other traversals if we have the preorder one available which does the work without storing anything in the same time complexity.

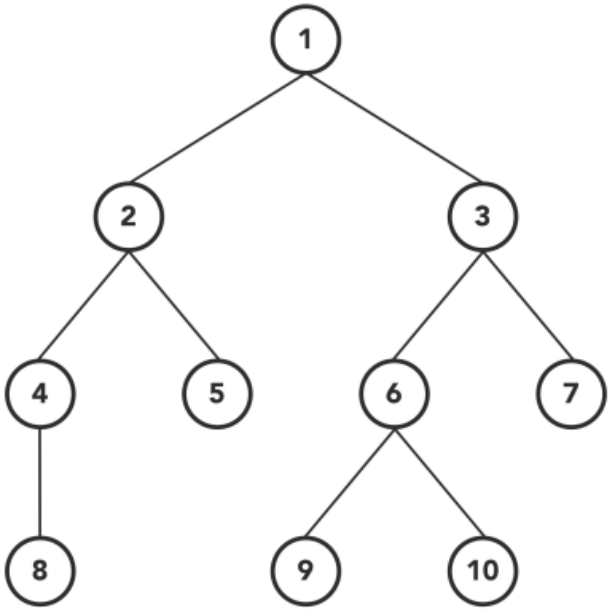
- ***Search Algorithm***:

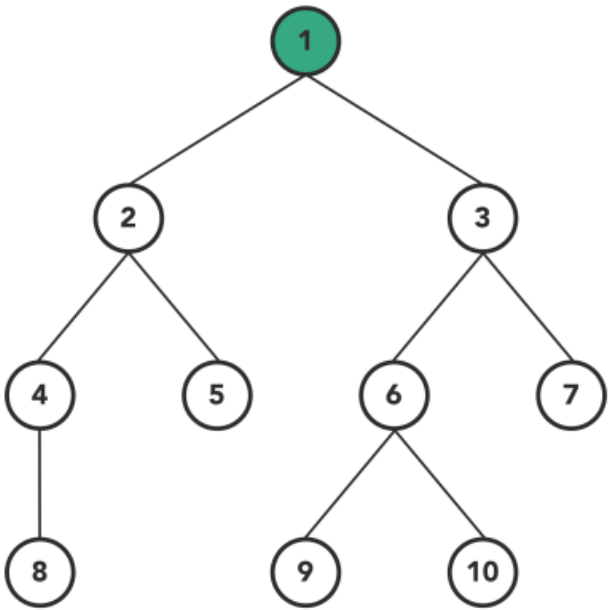
1. ***if curr == NULL then return false***
2. ***if curr node data == key then return true***
3. ***go to step 1 with left node of curr***
4. ***go to step 1 with right node of curr***



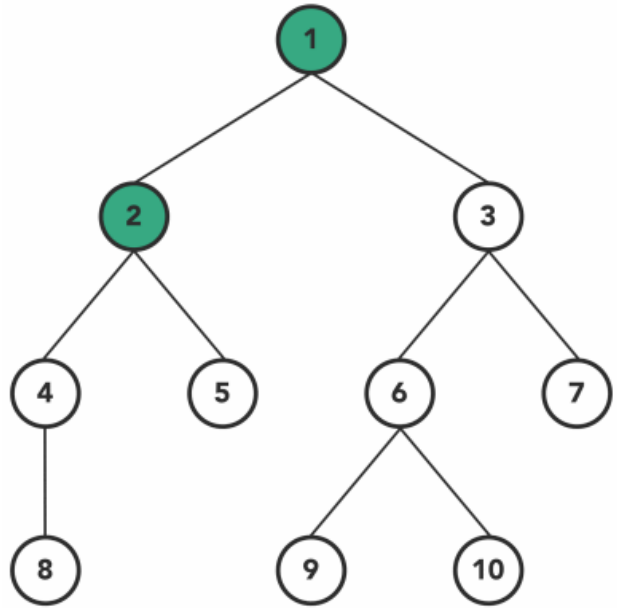
Search Operation in Binary Tree

- Search for 5





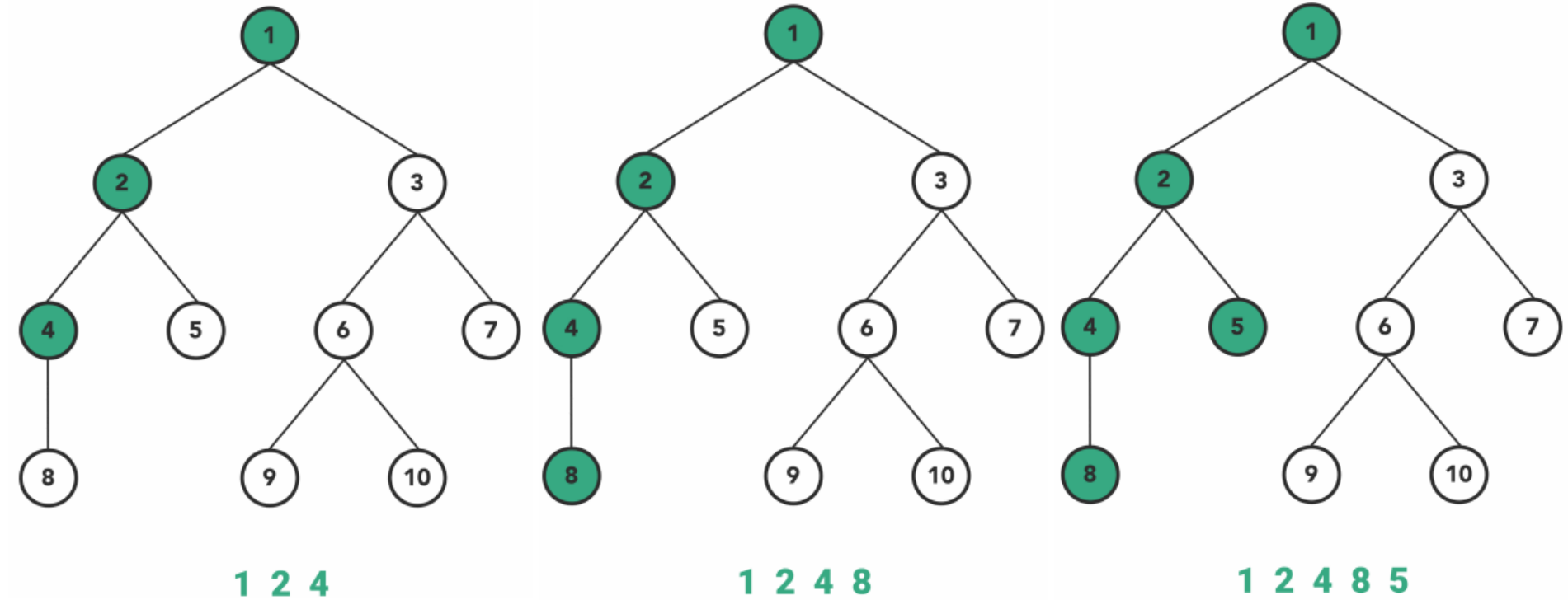
1



1 2

Search Operation in Binary Tree

- Search for 5



Search Operation in Binary Tree

```
// This function searches if the given data in the binary tree
bool search(node* curr, int data) {
    // base case we reach a null node
    if (curr == NULL)
        return false;
    // check if we find the data at the curr node
    if (curr->data == data)
        return true;
    // repeat the same definition of search at left and right subtrees
    bool left_search  = search(curr->left, data);
    bool right_search = search(curr->right, data);
    return left_search || right_search;
}
```

Binary-Tree.cpp

Lecture Agenda

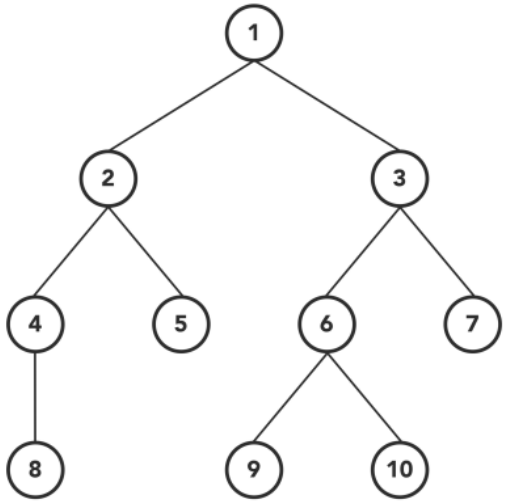
- ✔ Section 1: Introduction to Binary Tree
- ✔ Section 2: Traverse Operation
- ✔ Section 3: BFS vs. DFS for Binary Tree
- ✔ Section 4: Search Operation
- **Section 5: Deletion Operation**
- Section 6: Time Complexity & Space Complexity

Deletion Operation in Binary Tree

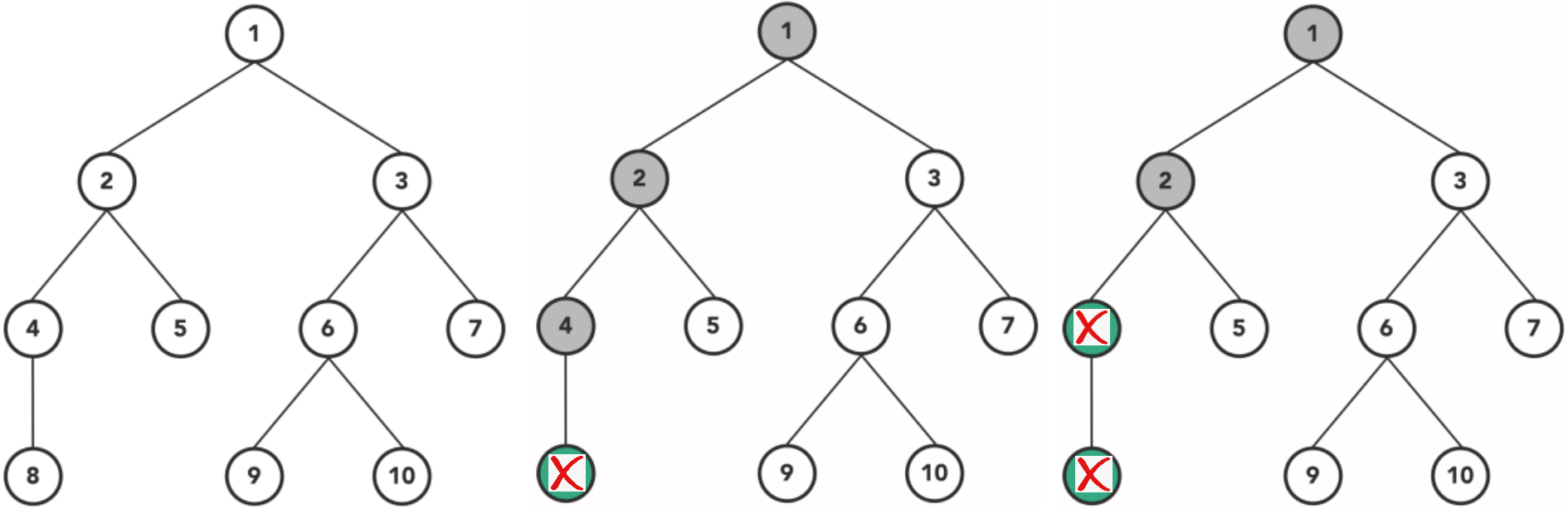
- To delete a tree, we must traverse all the nodes of the tree and delete them one by one. So, which traversal we should use - inorder transversal, preorder transversal, or the postorder transversal? The answer is simple. We should use the postorder transversal because before deleting the parent node, we should delete its child nodes first.
- We can delete the tree with other traversals also with extra space complexity but why should we go for the other traversals if we have the postorder one available which does the work without storing anything in the same time complexity.

- ***Delete Algorithm***:

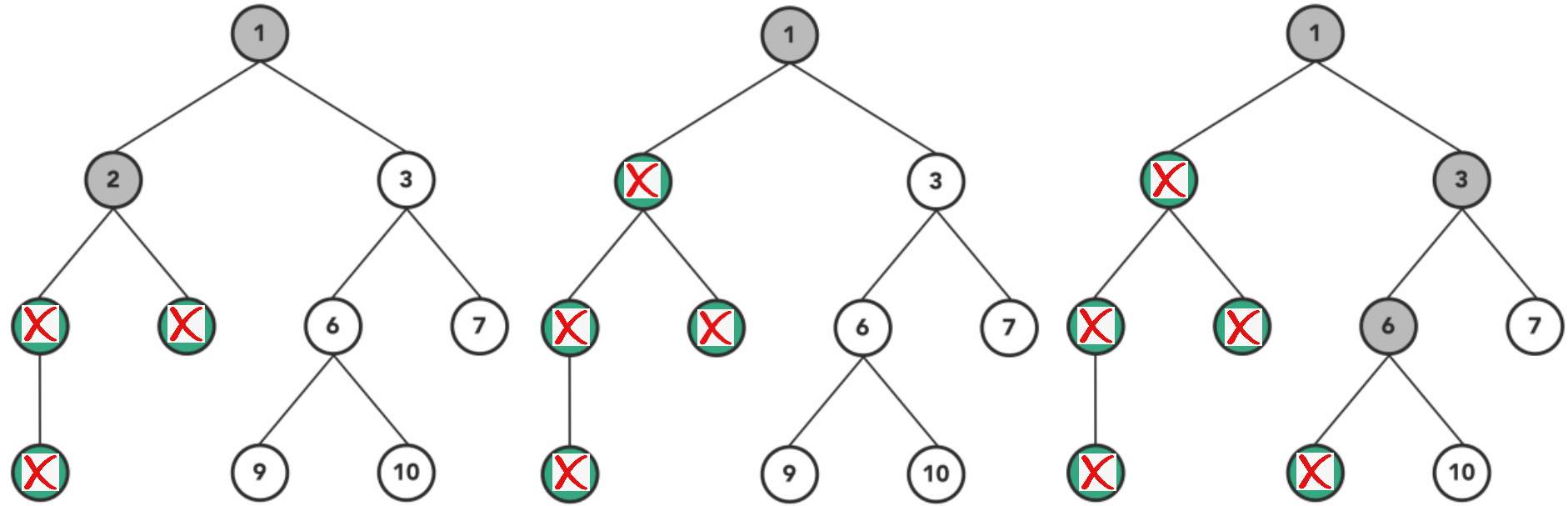
1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***go to step 1 with right node of curr***
4. ***delete curr node***



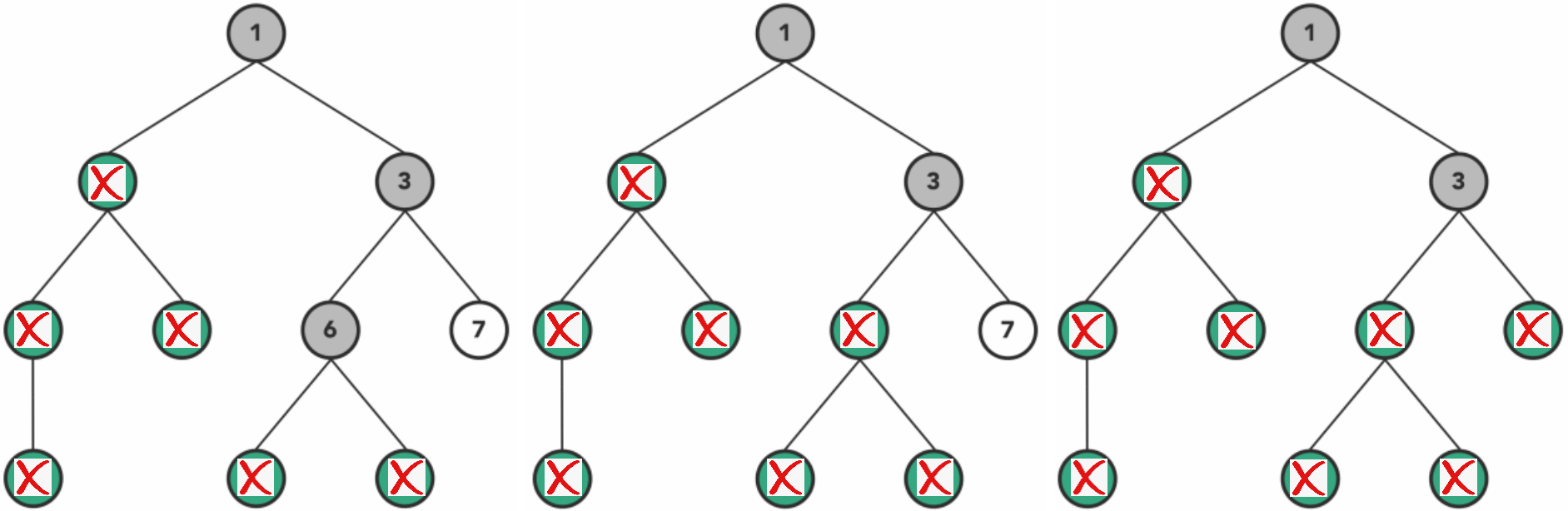
Deletion Operation in Binary Tree



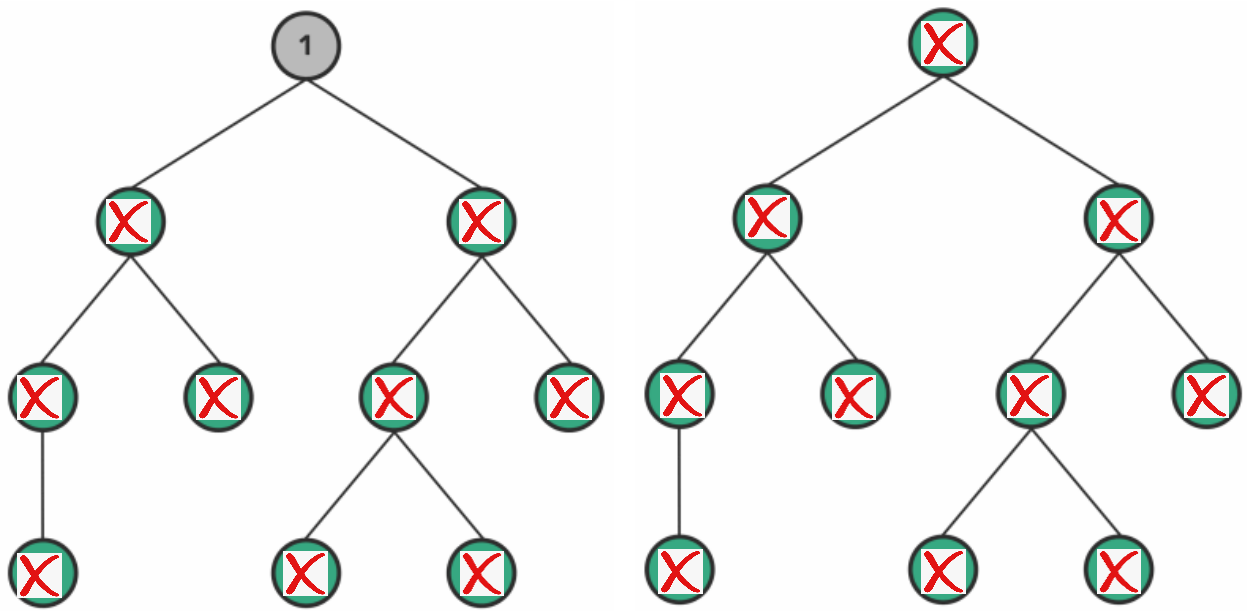
Deletion Operation in Binary Tree



Deletion Operation in Binary Tree



Deletion Operation in Binary Tree



Deletion Operation in Binary Tree

```
// This function traverses the binary tree in postorder to delete all nodes
void delete_tree(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition of postorder traversal
    delete_tree(curr->left);
    delete_tree(curr->right);
    delete(curr);
}
```

Binary-Tree.cpp

Functionality Testing - Binary Tree

- Initialize a global struct

```
#include <bits/stdc++.h>
using namespace std;

// A binary tree node
struct node {
    int data;
    node* left;
    node* right;
};
// Initialize a global pointer for root
node* root;
```

Binary-Tree.cpp

Functionality Testing - Binary Tree

- In the Main function:

```
// first level of the binary tree (root only)
root = new node();
root->data = 5;

cout << "Binary Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "Binary Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "Binary Tree post-order traversal : ";
postOrder(root);
```

- Expected Output:

```
Binary Tree in-order traversal   : 5
Binary Tree pre-order traversal  : 5
Binary Tree post-order traversal : 5
```

- Tree Diagram

5

Binary-Tree.cpp

Functionality Testing - Binary Tree

- In the Main function:

```
// second level of the binary tree
root->left = new node();
root->left->data = 7;
root->right = new node();
root->right->data = 11;

cout << "Binary Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "Binary Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "Binary Tree post-order traversal : ";
postOrder(root);
```

- Expected Output:

```
Binary Tree in-order traversal   : 7 5 11
Binary Tree pre-order traversal  : 5 7 11
Binary Tree post-order traversal : 7 11 5
```

- Tree Diagram

Binary-Tree.cpp

Functionality Testing - Binary Tree

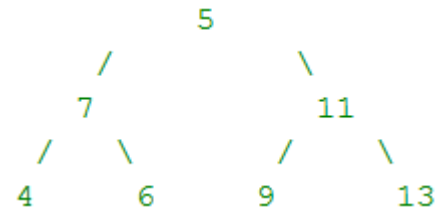
➢ In the Main function:

```
// third level of the binary tree
root->left->left = new node();
root->left->left->data = 4;
root->left->right = new node();
root->left->right->data = 6;
root->right->left = new node();
root->right->left->data = 9;
root->right->right = new node();
root->right->right->data = 13;

cout << "Binary Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "Binary Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "Binary Tree post-order traversal : ";
postOrder(root);
```

➢ Expected Output:

```
Binary Tree in-order traversal   : 4 7 6 5 9 11 13
Binary Tree pre-order traversal  : 5 7 4 6 11 9 13
Binary Tree post-order traversal : 4 6 7 9 13 11 5
```

➢ Tree Diagram



Functionality Testing - Binary Tree

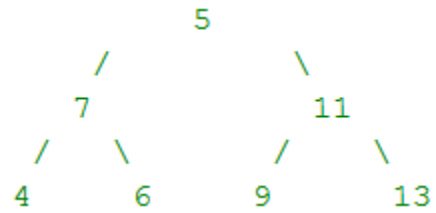
- In the Main function:

```
cout << "Binary Tree breadth first traversal : ";
bfs();
```

- Expected Output:

```
Binary Tree breadth first traversal : 5 7 11 4 6 9 13
```

- Tree Diagram



Binary-Tree-BFS.cpp

Functionality Testing - Binary Tree

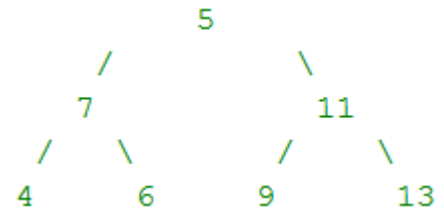
- In the Main function:

```
if (search(root, 9))
    cout << "element " << 9 << " in the tree\n";
else
    cout << "element " << 9 << " not in the tree\n";

if (search(root, 4))
    cout << "element " << 4 << " in the tree\n";
else
    cout << "element " << 4 << " not in the tree\n";

if (search(root, 8))
    cout << "element " << 8 << " in the tree\n";
else
    cout << "element " << 8 << " not in the tree\n";

if (search(root, 3))
    cout << "element " << 3 << " in the tree\n";
else
    cout << "element " << 3 << " not in the tree\n";
```

- Expected Output:

```
element 9 in the tree
element 4 in the tree
element 8 not in the tree
element 3 not in the tree
```

- Tree Diagram



Binary-Tree.cpp

Functionality Testing - Binary Tree

- In the Main function:

```
delete_tree(root);
root = NULL;

cout << "Binary Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "Binary Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "Binary Tree post-order traversal : ";
postOrder(root);
```

- Expected Output:

```
Binary Tree in-order traversal   :
Binary Tree pre-order traversal  :
Binary Tree post-order traversal :
```

Lecture Agenda

- ✔ Section 1: Introduction to Binary Tree
- ✔ Section 2: Traverse Operation
- ✔ Section 3: BFS vs. DFS for Binary Tree
- ✔ Section 4: Search Operation
- ✔ Section 5: Deletion Operation

Section 6: Time Complexity & Space Complexity

Time Complexity & Space Complexity

➢ Time Analysis

	Worst Case	Average Case
• Traverse	Θ(n)	θ(n)
• Search	Θ(n)	θ(n)
• Delete Tree	Θ(n)	θ(n)

➢ Space Analysis

All traversals require Θ(n) time as they visit every node exactly once. There is difference in terms of extra space required.

- Extra Space required for Level Order Traversal is Θ(w) where w is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.
- Extra Space required for Depth First Traversals is Θ(h) where h is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Lecture Agenda

- ✓ Section 1: Introduction to Binary Tree
- ✓ Section 2: Traverse Operation
- ✓ Section 3: BFS vs. DFS for Binary Tree
- ✓ Section 4: Search Operation
- ✓ Section 5: Deletion Operation
- ✓ Section 6: Time Complexity & Space Complexity

Practice

Practice

1- Traversal a binary tree in-order in (recursive/iterative) ways
2- Traversal a binary tree pre-order in (recursive/iterative) ways
3- Traversal a binary tree post-order in (recursive/iterative) ways
4- Traversal a binary tree level Order (BFS) one per line
5- Traversal a binary tree all root-to-leaf paths one per line
6- Count number of nodes in binary tree in (recursive/iterative) ways
7- Count number of leaf nodes in binary tree in (recursive/iterative) ways
8- Find the maximum value of all nodes in a binary tree in (recursive/iterative) ways
9- Find the minimum value of all nodes in a binary tree in (recursive/iterative) ways
10- Calculate the maximum depth in a binary tree in (recursive/iterative) ways
11- Calculate the minimum depth in a binary tree in (recursive/iterative) ways
12- Calculate height of each node in a binary tree
13- Calculate diameter of a binary tree
14- Calculate the maximum path sum in a binary tree in (recursive/iterative) ways
15- Calculate the minimum path sum in a binary tree in (recursive/iterative) ways

Practice

16- Evaluation of expression tree

17- Print all full nodes in a binary tree

18- Convert a tree into it's mirror tree

19- Sum of all nodes in a binary tree

20- Check for children sum property in a binary tree

21- Check if a given binary tree is sum tree

22- Check if a given number equal root to leaf path sum

23- Replace each node in binary tree with the sum of its in-order traversal

24- Find n-th node of in-order traversal of a binary tree

25- Find n-th node in post-order traversal of a binary tree

26- Find n-th node in pre-order traversal of a binary tree

27- Check if there is a root-to-leaf path with given sequence

28- Check if leaf traversal of two binary trees is the same

29- Check if a binary tree is subtree of another binary tree

30- Check if a binary tree has duplicate values

Practice

31- Check if a given binary tree is perfect or not
32- Check if a given binary tree is full or not
33- Check if a given binary tree is complete or not
34- Check if a given binary tree is balanced or not
35- Check if a given binary tree is skewed or not
36- Check if a given binary tree is pathological or not
37- Check if a binary tree contains duplicate subtrees
38- Check if two trees are mirror of each other
39- Check if all leaves are at same level in a binary tree
40- Check if two nodes are cousins in a binary tree
41- Check if given pre-order, in-order and post-order are of the same tree
42- Check if two trees are identical
43- Check for symmetric binary tree by iterative way
44- Check if removing an edge can divide a binary tree in two halves
45- Print middle level of a perfect binary tree without finding height

Practice

46- Print cousins of a given node in a binary tree
47- Print the longest leaf to leaf path in a binary tree
48- Print path from root to a given node in a binary tree
49- Print root-to-leaf paths in a binary tree
50- Print the nodes at odd levels of a binary tree
51- Find lowest common ancestor LCA in a binary tree
52- Find distance between two nodes of a binary tree
53- Print common nodes on path from root or common ancestors
54- Maximum difference between node and its ancestor in binary tree
55- Print the path common to the two paths from the root to the two given nodes
56- Print all k-sum paths in a binary tree
57- Sum of all left leaves in a given binary tree
58- Sum of all right leaves in a given binary tree
59- Sum of nodes on the longest path from root to leaf node in a binary tree
60- Find largest subtree sum in a binary tree

Practice

61- Find the maximum path sum between two leaves of a binary tree
62- Find the maximum sum leaf to root path in a binary tree
63- Find the maximum sum from a tree with adjacent levels not allowed
64- Find the subtree with given sum in a binary tree
65- Count subtrees that sum up to a given value in a binary tree
66- Difference between sums of odd level and even level nodes of a binary tree
67- Find the maximum level sum in a binary tree
68- Sum of all leaf nodes of a binary tree
69- Sum of leaf nodes at minimum level
70- Construct a binary tree from pre-order and level order traversals
71- Construct a binary tree from in-order and level order traversals
72- Construct a binary tree from post-order and level order traversals
73- Construct a binary tree from pre-order and post-order traversals
74- Construct a binary tree from post-order and in-order traversals
75- Construct a binary tree from in-order and pre-order traversals


DO
MORE.