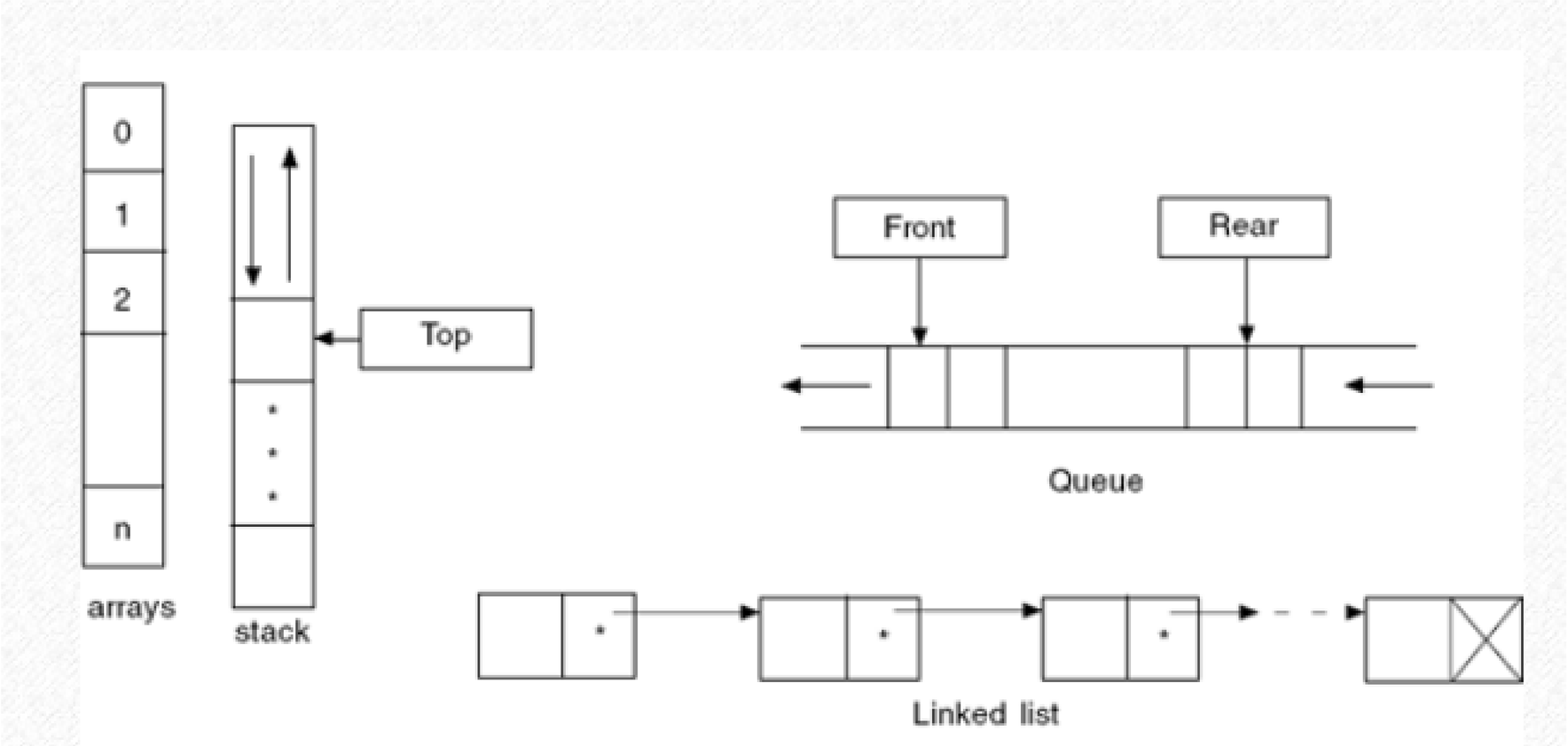
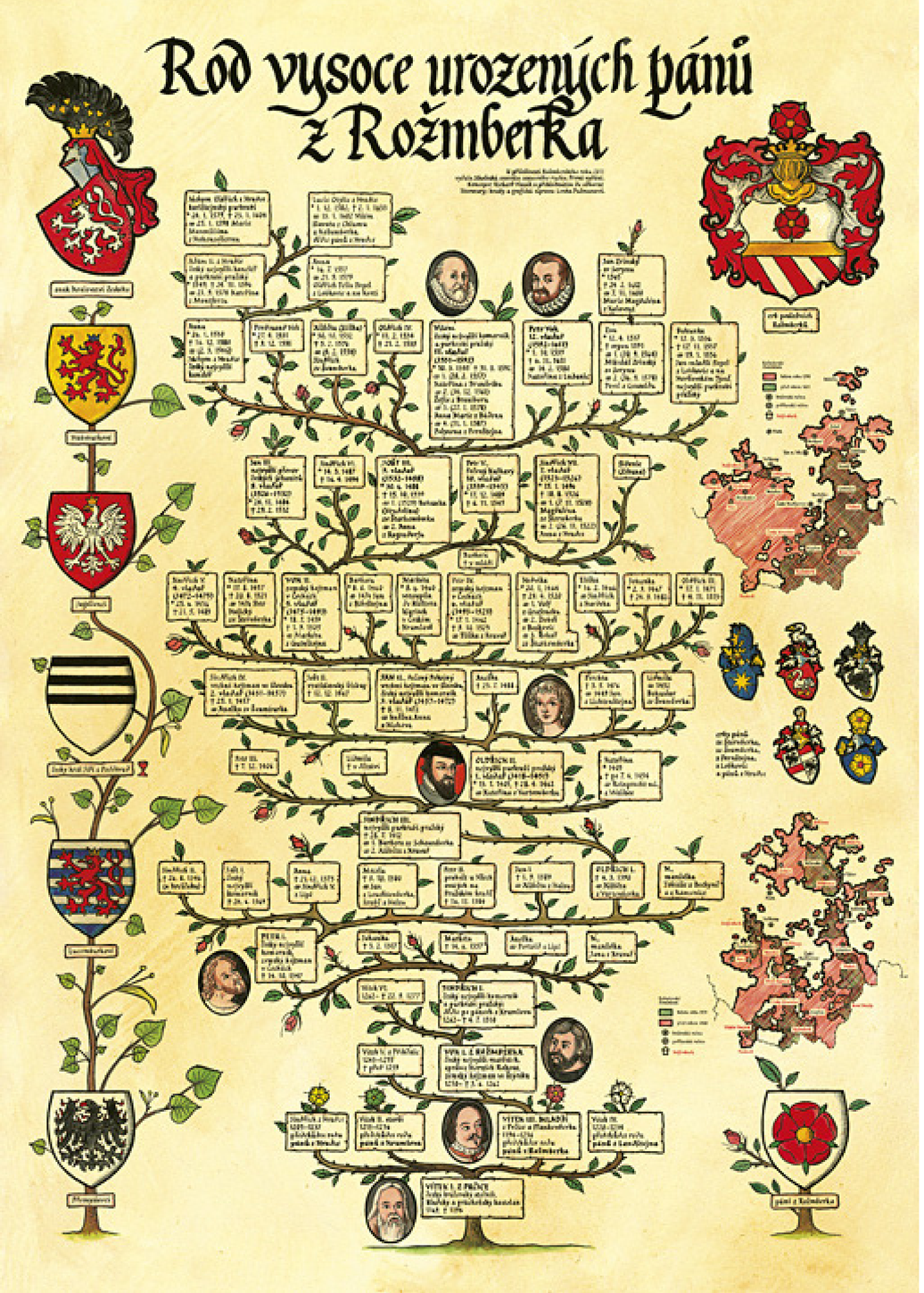
Introduction to binary trees

Are linear data structures enough?

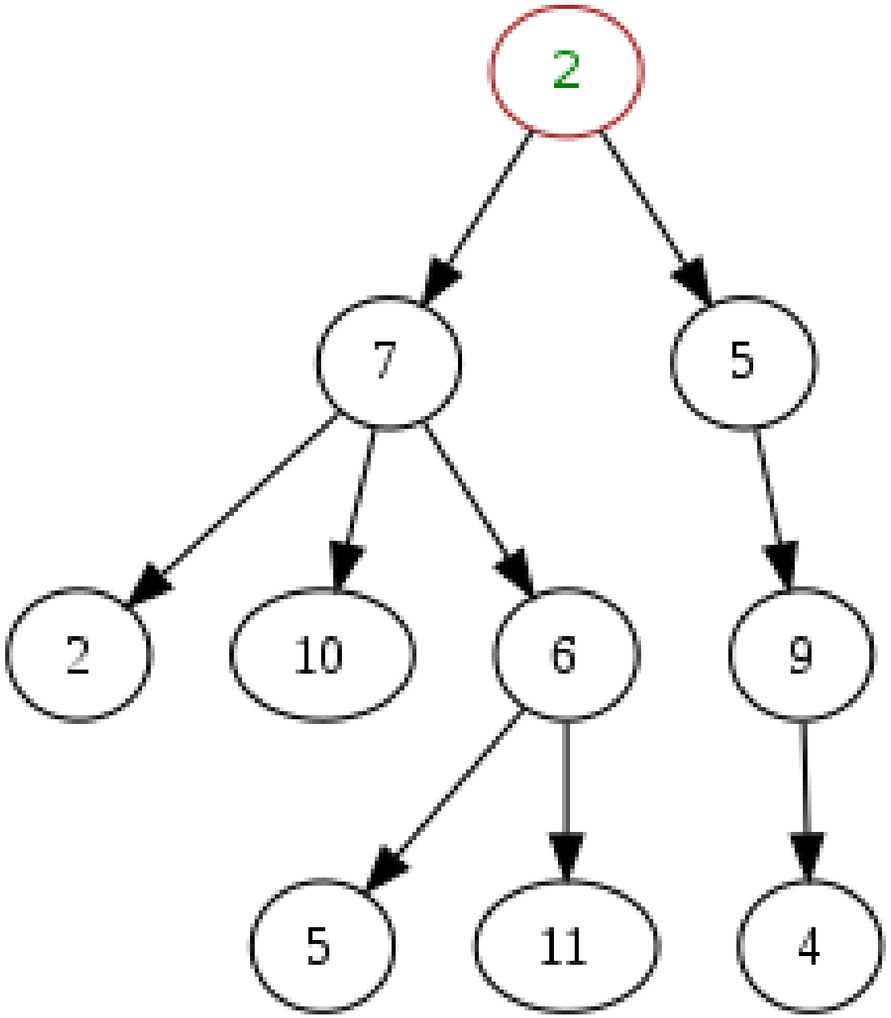


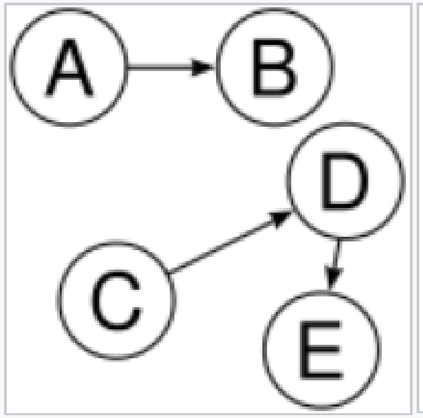


The Rosenberg family (Czech: Rožmberkové, sg. z Rožmberka) was a prominent Bohemian noble family that played an important role in Czech medieval history from the 13th century until 1611.

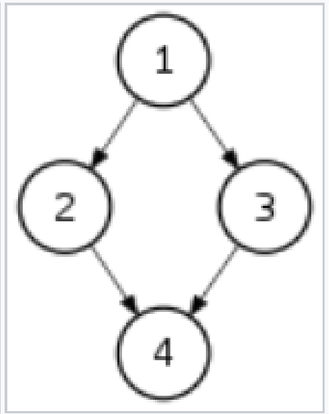
Tree data structure

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the “children”).

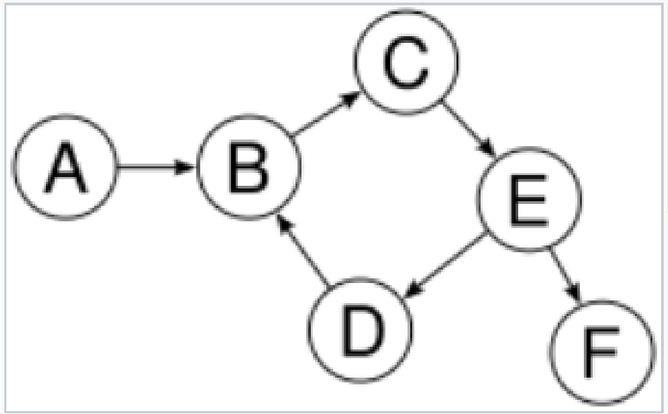




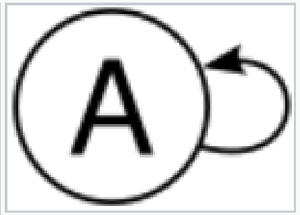
Not a tree: two non-connected parts, A→B and C→D→E. There is more than one root.



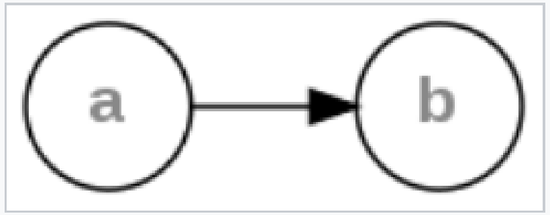
Not a tree: undirected cycle 1-2-4-3. 4 has more than one parent (inbound edge).



Not a tree: cycle B→C→E→D→B. B has more than one parent (inbound edge).

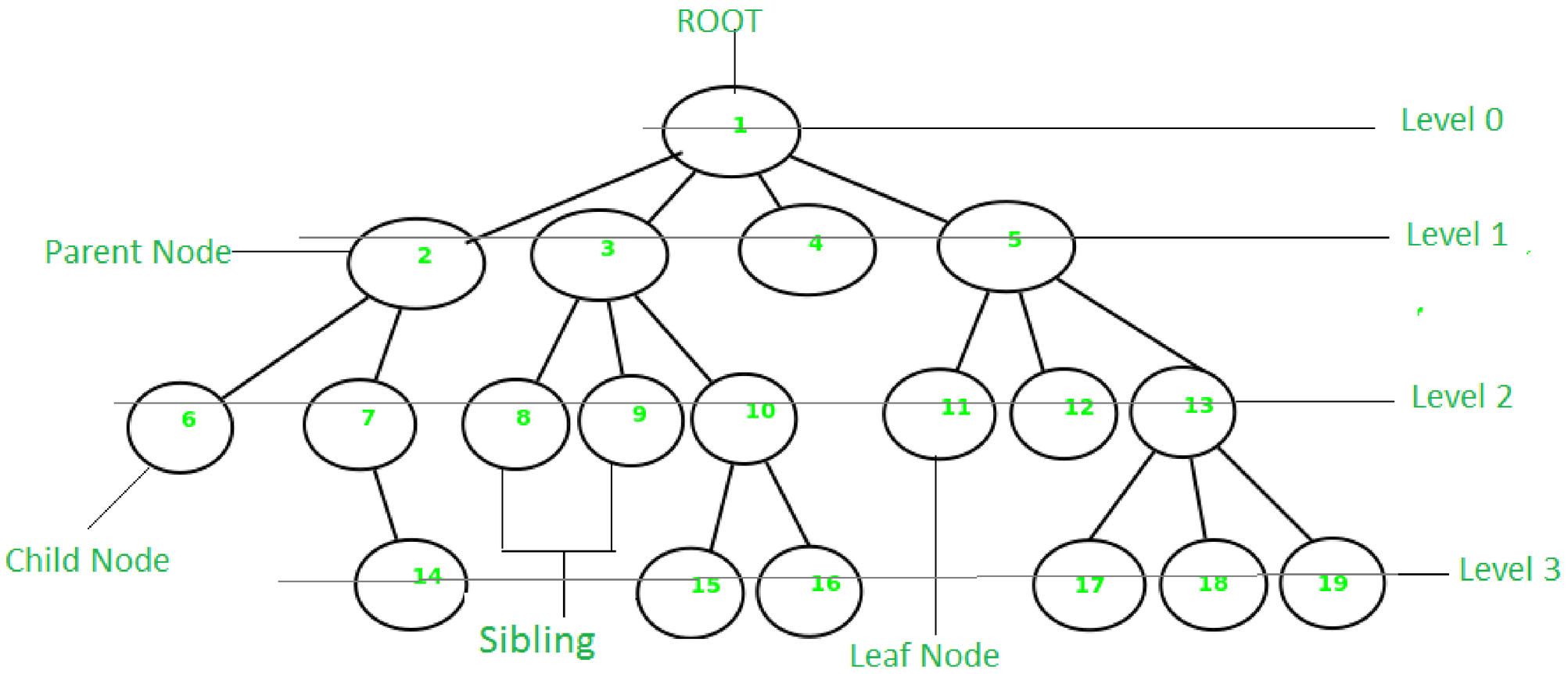


Not a tree: cycle A→A. A is the root but it also has a parent.



Each linear list is trivially a tree

Tree Vocabulary

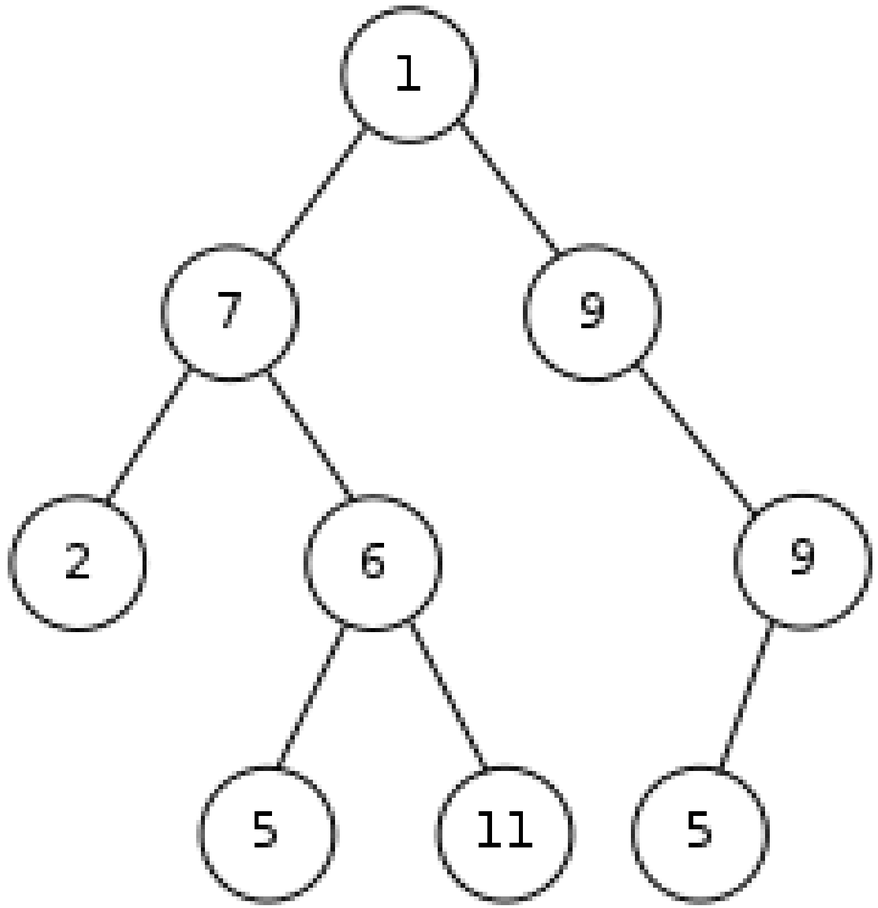


Why tree?

- 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.
- Trees (*with some ordering* e.g., BST) provide **moderate access/search** (quicker than Linked List and slower than arrays).
- Trees provide **moderate insertion/deletion** (quicker than Arrays and slower than Linked Lists).
- Like Linked Lists and unlike Arrays, Trees **don't have an upper limit on the number of nodes** as nodes are linked using pointers.

Binary tree

- A tree whose elements have at most **2** children is called a **binary tree**.
- Since each element in a binary tree can have only 2 children, we typically name them the **left** and **right** children.

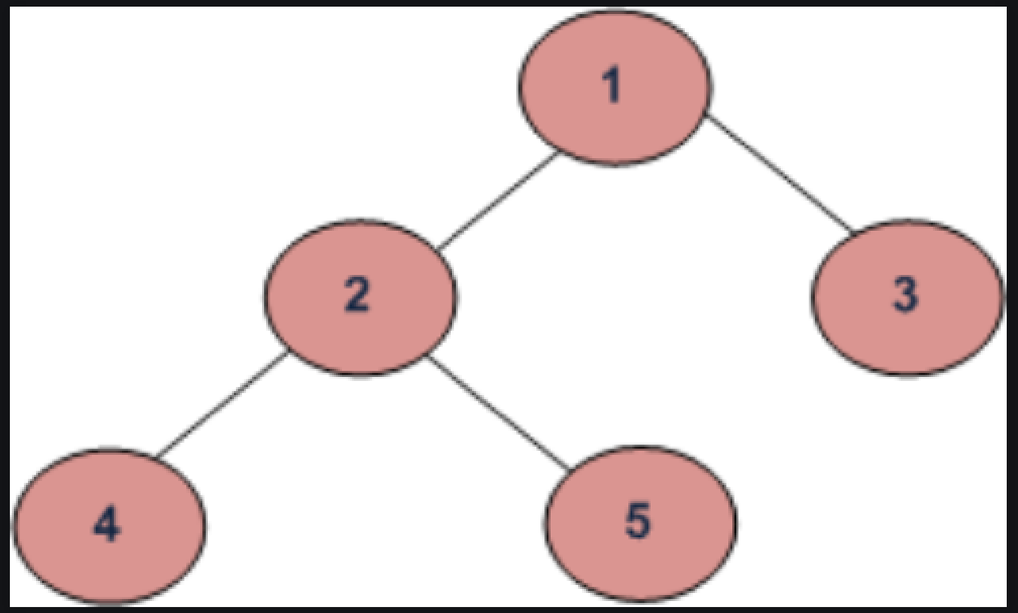


Traversals

A Tree is typically traversed in two ways:

- Breadth First Traversal (Or Level Order Traversal)
- Depth First Traversals
  - Inorder Traversal (Left-Root-Right)
  - Preorder Traversal (Root-Left-Right)
  - Postorder Traversal (Left-Right-Root)

Example




```
BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:
      Preorder Traversal : 1 2 4 5 3
      Inorder Traversal  :  4 2 5 1 3
      Postorder Traversal : 4 5 2 3 1
```

Implementations

Breadth First Traversal
(Or Level Order Traversal)

Method 1 (Use a function to print a current level)

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
   printCurrentLevel(tree, d);

/*Function to print all nodes at a current level*/
printCurrentLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printCurrentLevel(tree->left, level-1);
    printCurrentLevel(tree->right, level-1);
```

Method 2 (Using queue)

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children
       (first left then right children) to q
    c) Dequeue a node from q.
```

Depth First Traversals

Inorder Traversal

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Preorder Traversal

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Postorder Traversal

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

One more example

InOrder(root) visits nodes in the following order:

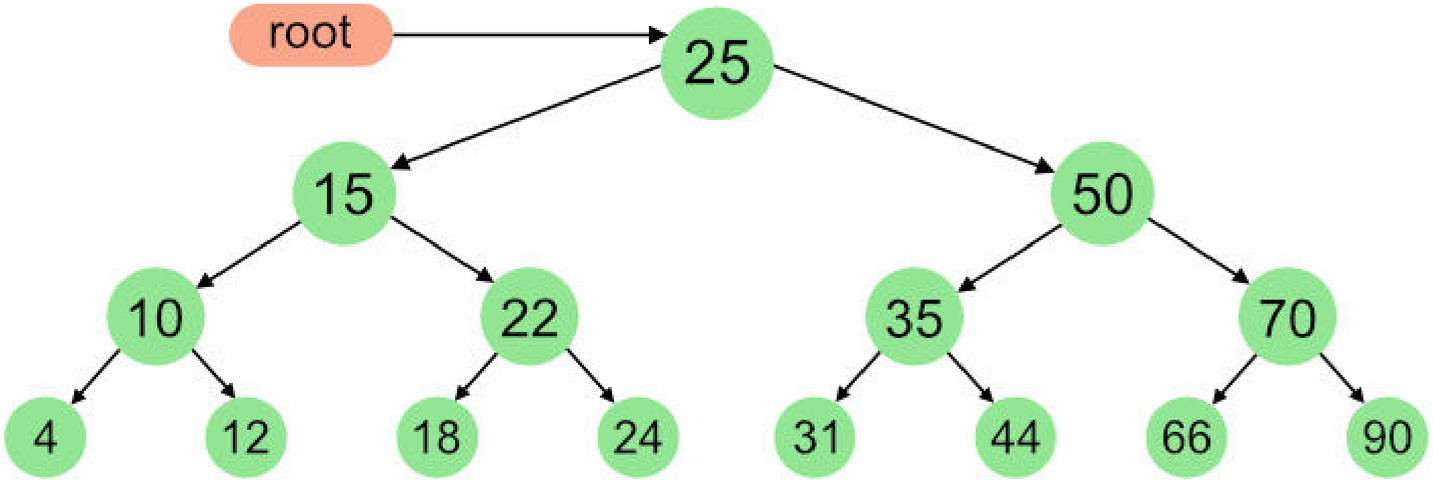
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Is there any difference in terms of Time Complexity?

- All four traversals require O(n) time as they visit every node exactly once.

Is there any difference in terms of Extra Space?

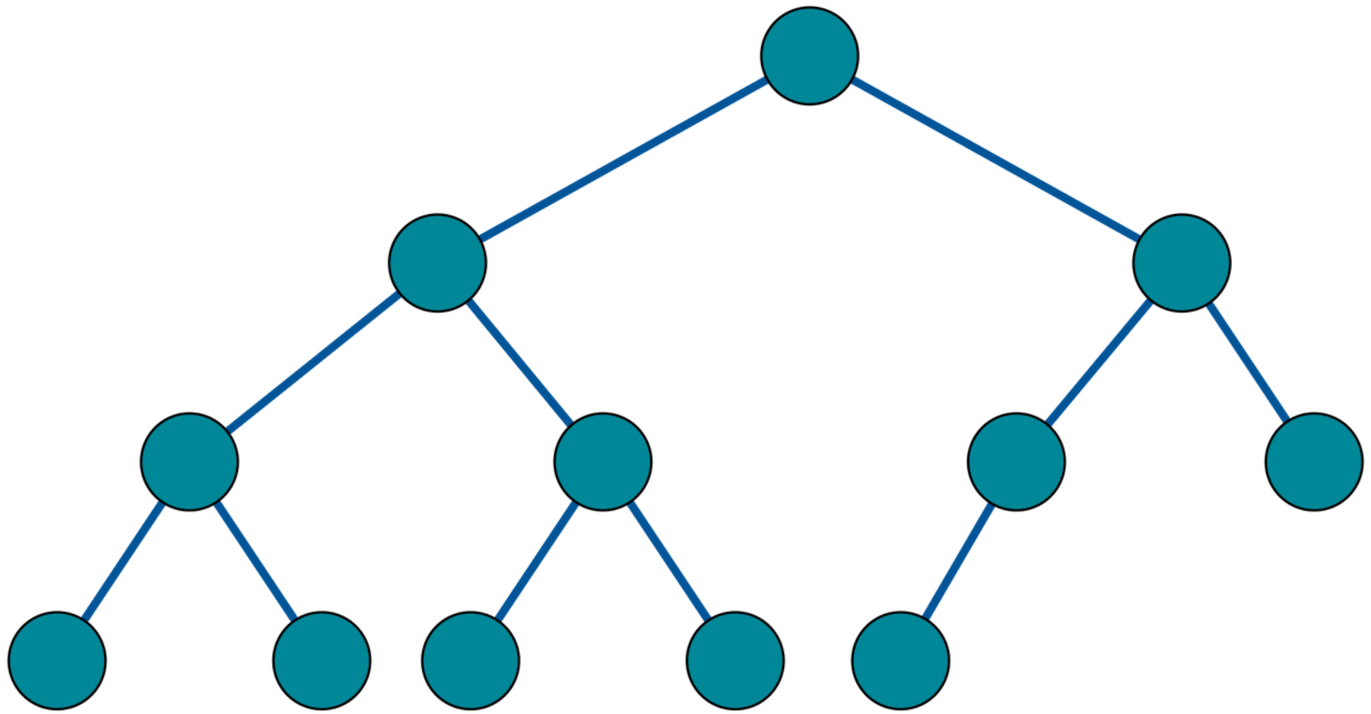
There is difference in terms of extra space required.

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

Complete binary trees

Complete Binary Tree

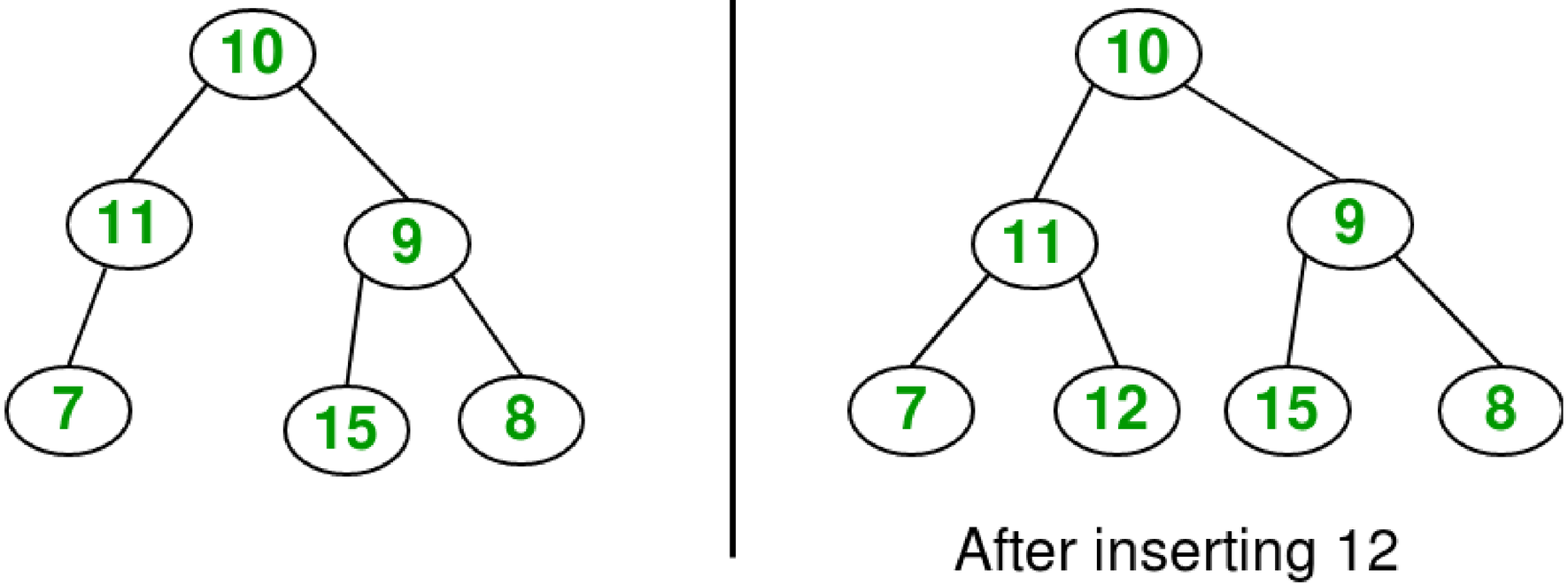
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.



Insertion in a Binary Tree

Insertion in a Binary Tree in level order

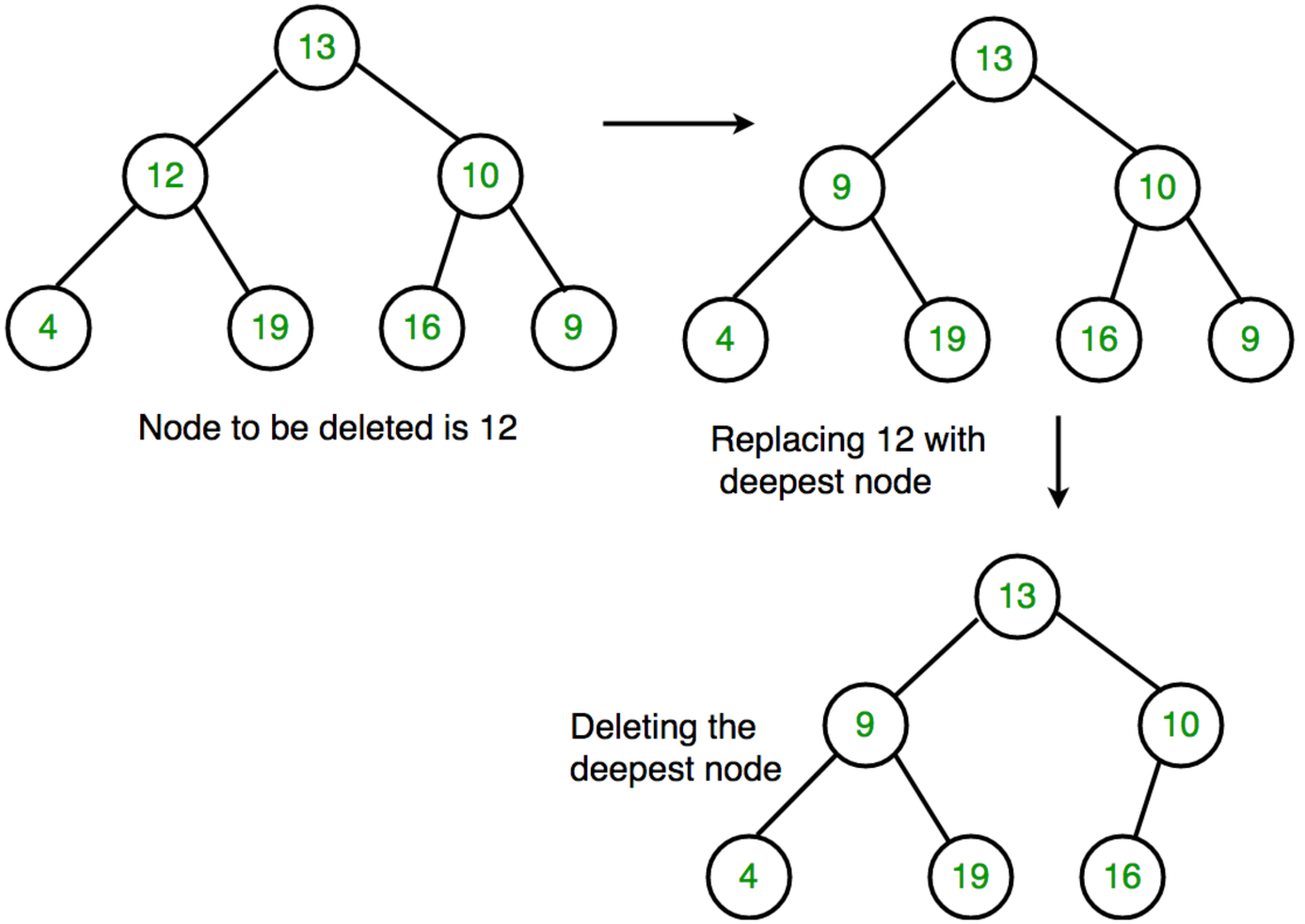
Given a binary tree and a key, insert the key into the binary tree at the first position available in level order.



Deletion in a Binary Tree

Algorithm:

1. Starting at the root, find the deepest and rightmost node in the binary tree and node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.

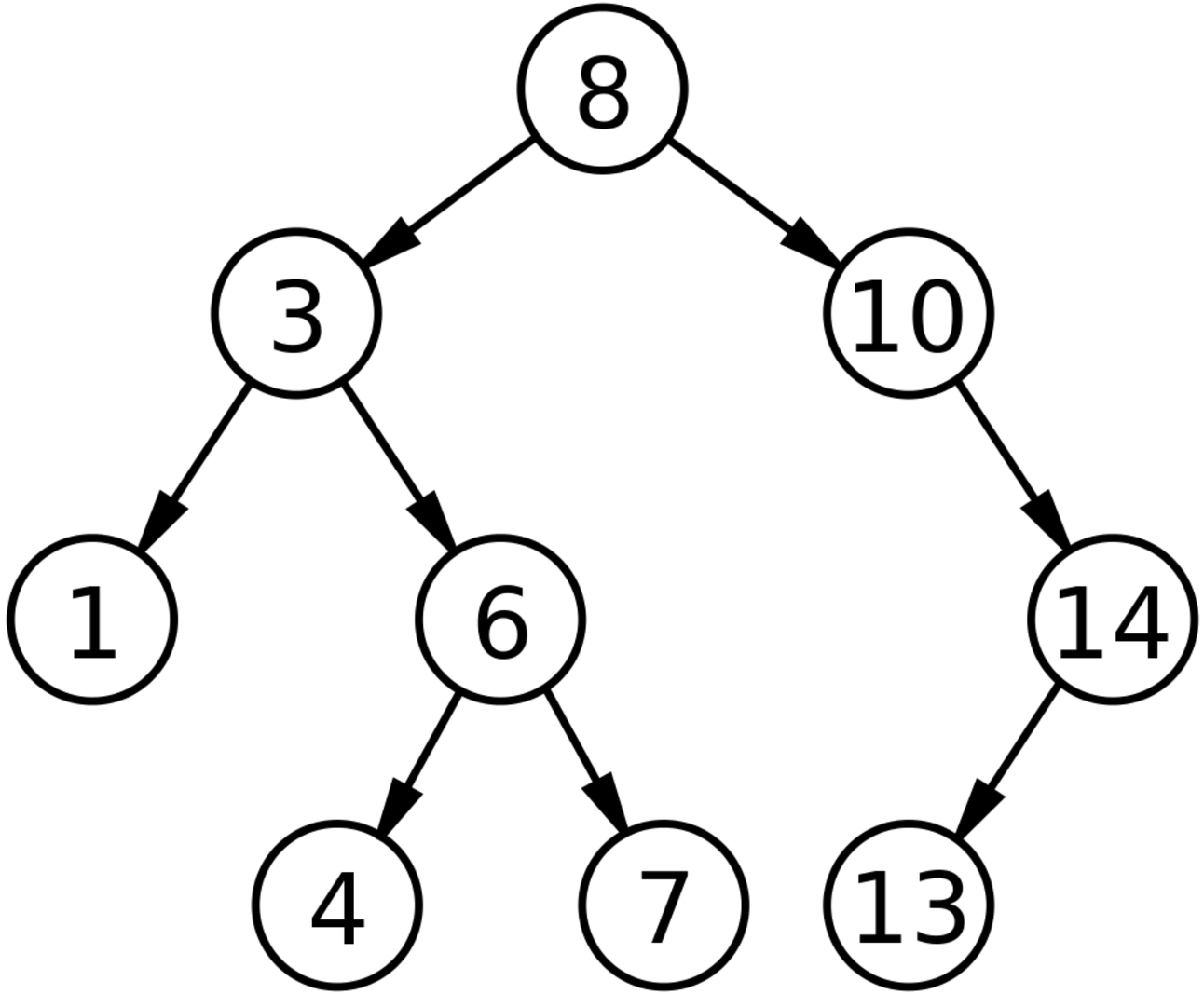


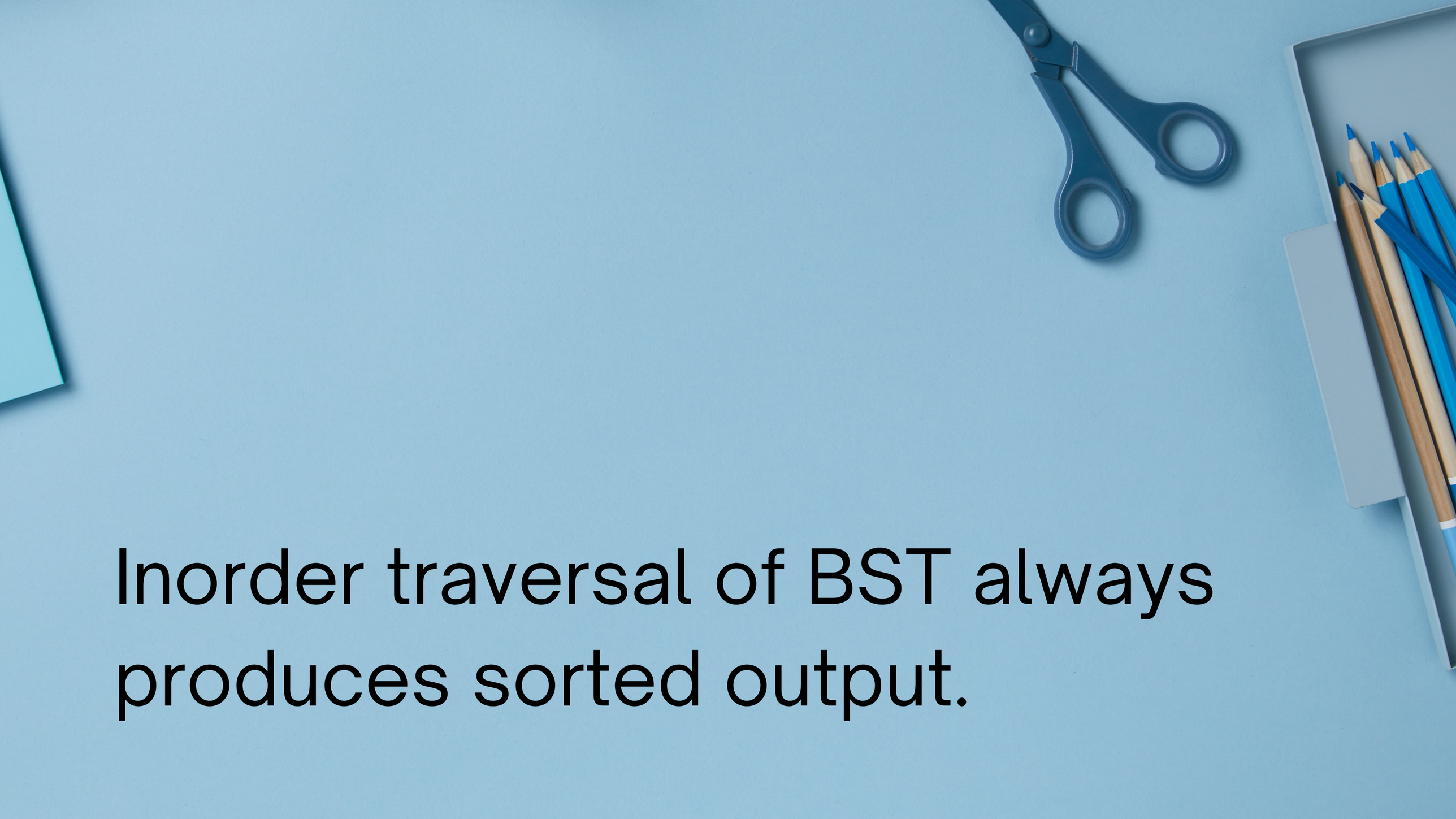
A complete binary tree will remain complete after the introduced insertion and deletion

Binary Search Tree

Binary Search Tree is a binary tree which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.


8
3
10
1
6
14
4
7
13



Inorder traversal of BST always produces sorted output.

Search:

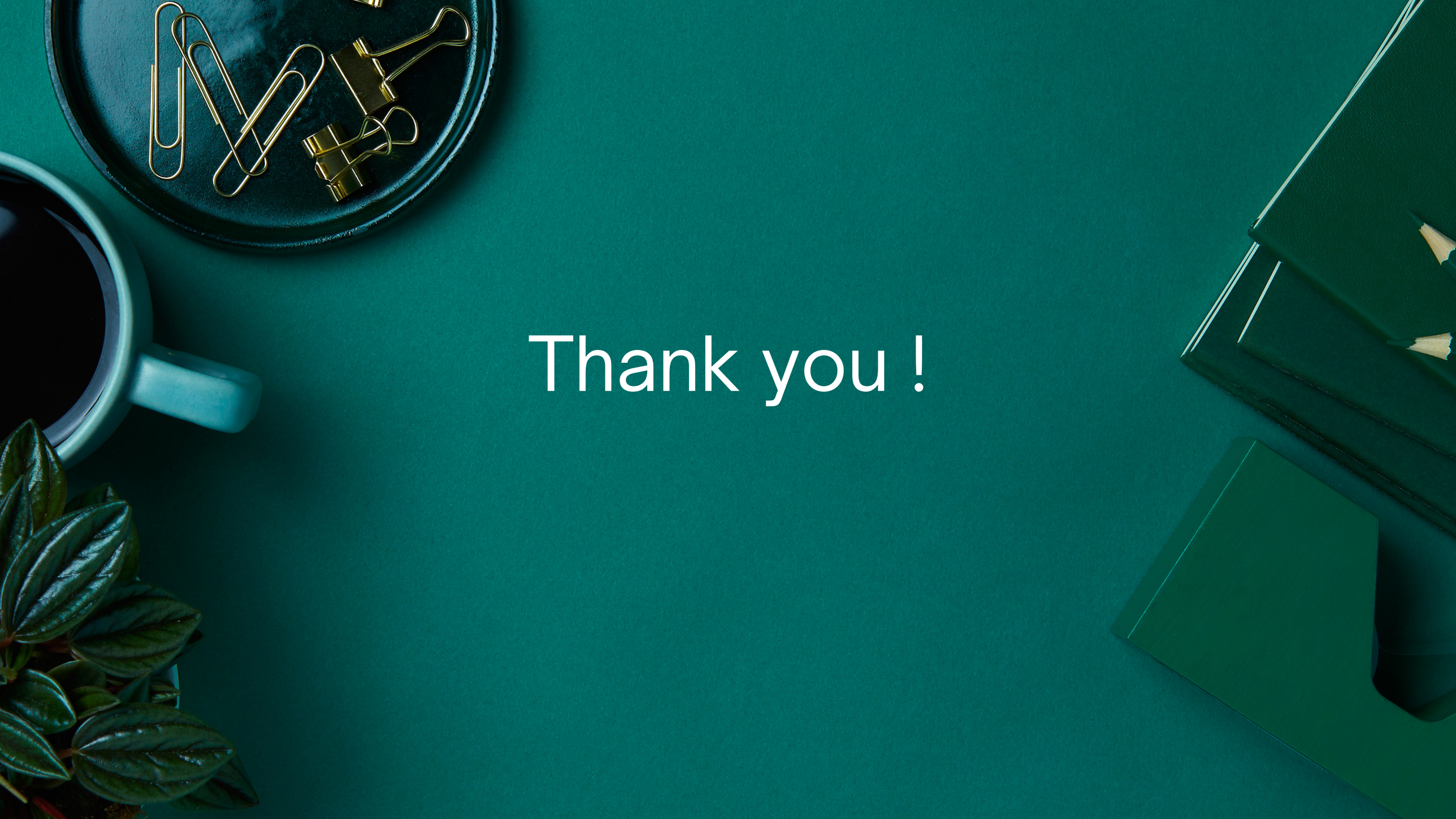
1. Start from the root.
2. Compare the searching element with root, if less than root, then recursively call left subtree, else recursively call right subtree.
3. If the element to search is found anywhere, return true, else return false.

Insertion:

- A new key is always inserted at the leaf. We start searching a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

Insertion (Recursive algorithm):

1. Start from the root.
2. Compare the inserting element with root, if less than root, then recursively call left subtree, else recursively call right subtree.
3. After reaching the end, just insert that node at left(if less than current) else right.


Thank you !