Discrete Mathematics (ITPC-309)

Trees : Part II

Mrs. Sanga G. Chaki

Department of Information Technology

Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

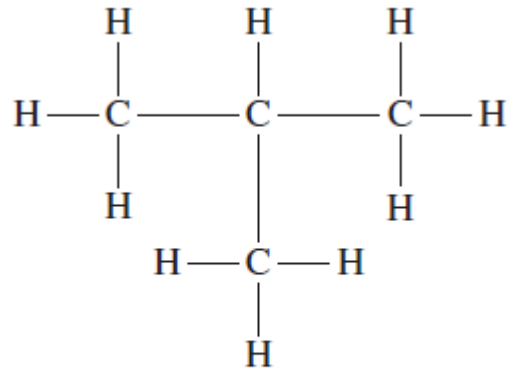
- Trees as Models
- Tree traversal
- Binary search tree
- Binary search tree Traversals

Trees as Models

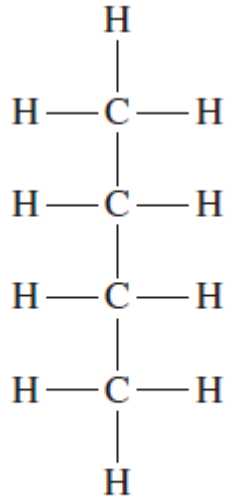
1. Applications of trees
2. Trees are used as models in diverse areas of study as computer science, chemistry, geology, botany, and psychology
3. We will go through some examples

Trees as Models - Chemistry

1. Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges
2. In graph models of saturated hydrocarbons, each carbon atom is represented by a vertex of degree 4, and each hydrogen atom is represented by a vertex of degree 1.
3. There are 3n + 2 vertices in a graph representing a compound of the form C_nH_{2n+2}.
4. There are 3n + 1 edges in these graphs.
5. Because the graph is connected and the number of edges is one less than the number of vertices, it must be a tree.
6. Using tree models, chemists find isomers of compounds



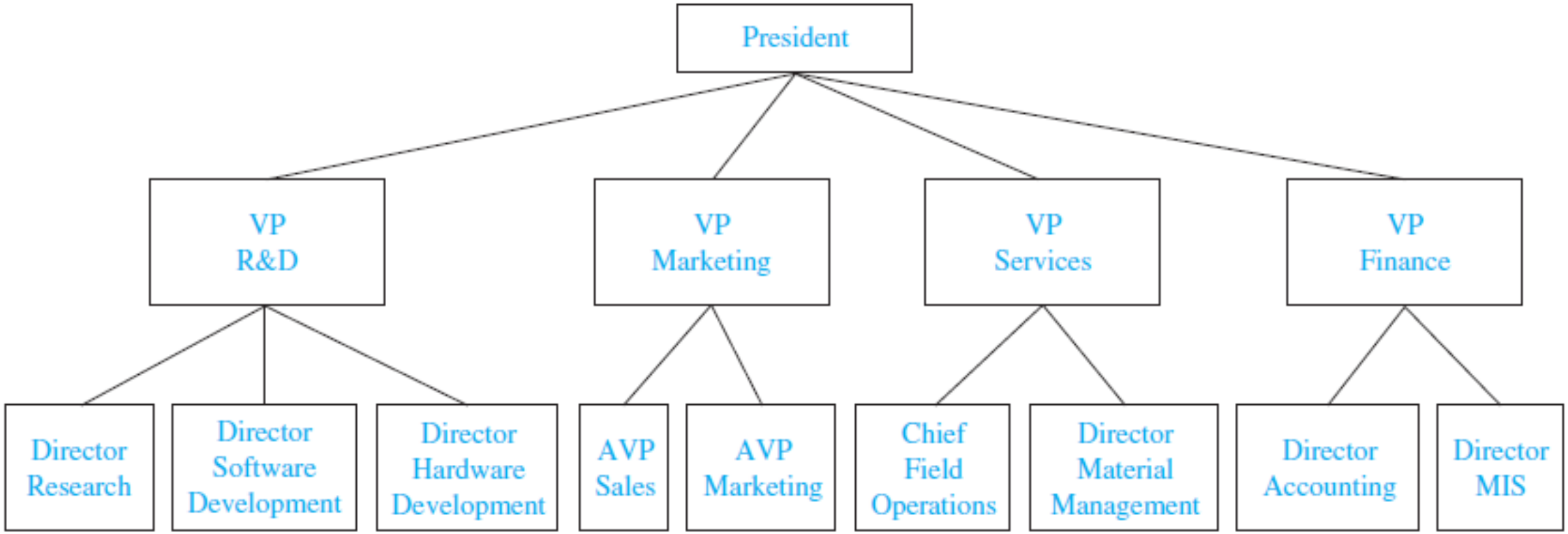
Isobutane



Butane

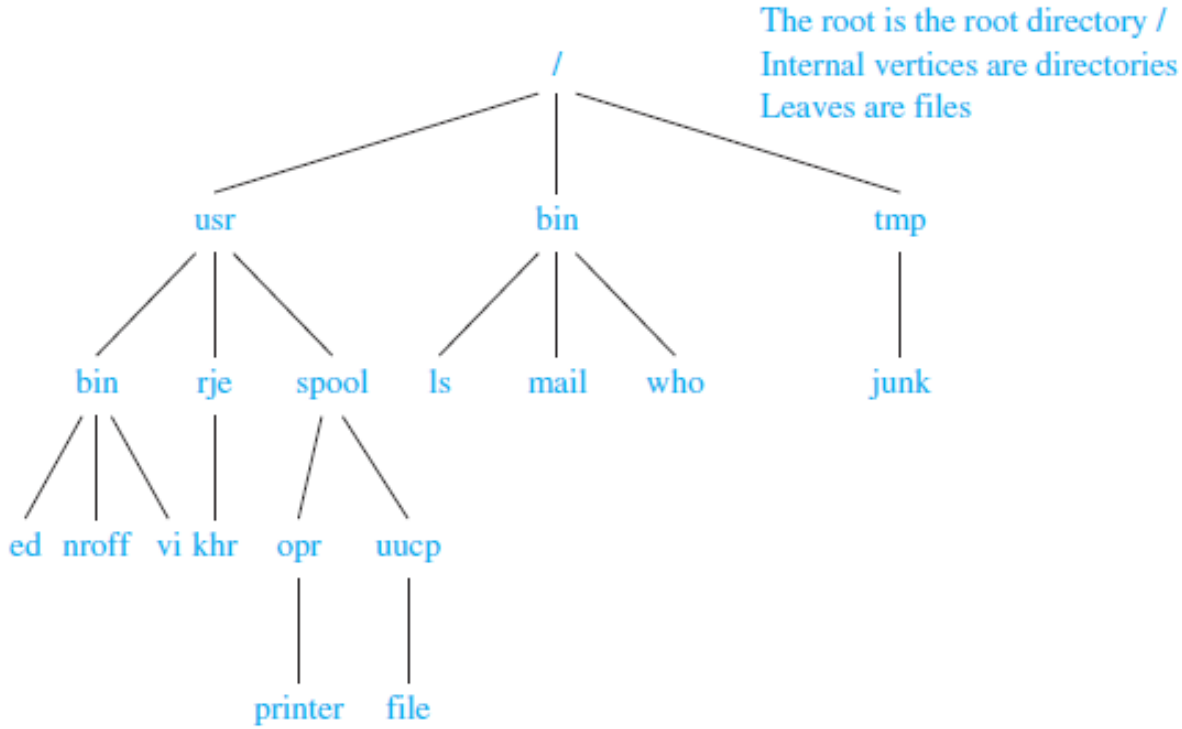
Trees as Models – To Represent Organizations

1. The structure of a large organization can be modeled using a rooted tree.
2. Each vertex in this tree represents a position in the organization.
3. An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex.
4. Example: In the organization represented by the given tree, the Director of Hardware Development works directly for the Vice President of R&D.



Trees as Models – Computer File Systems

1. Files in computer memory can be organized into directories.
2. A directory can contain both files and subdirectories.
3. The root directory contains the entire file system.
4. Thus, a file system may be represented by a rooted tree, where the root represents the root directory, internal vertices represent subdirectories, and leaves represent ordinary files or empty directories.
5. Example: One such file system is shown below. In this system, the file khr is in the directory rje.



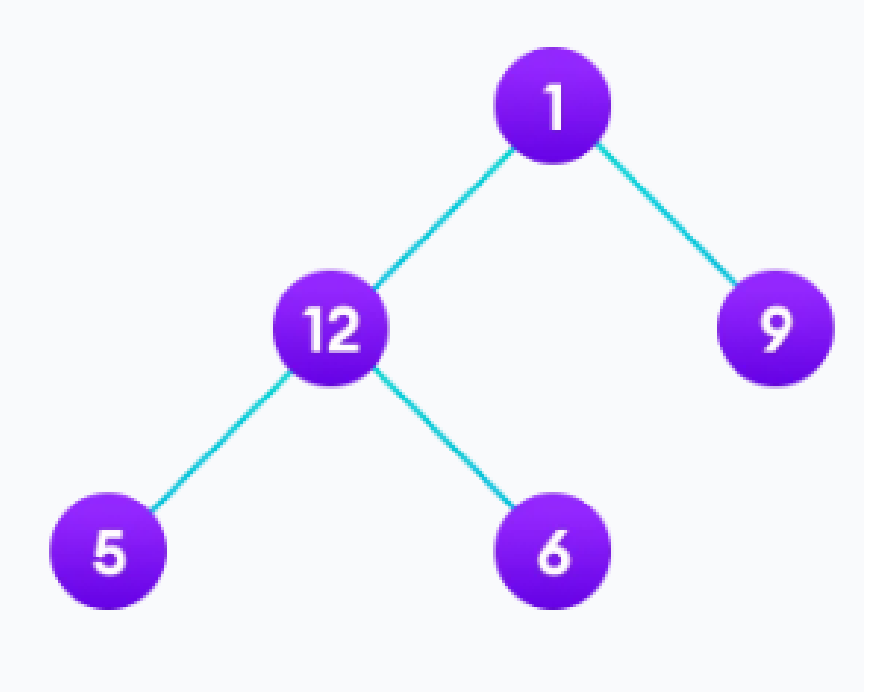
Tree Traversals

Tree Traversals

1. Traversing a tree means visiting every node in the tree.
2. Why needed?
   - To solve problems like add all the values in the tree
   - find the largest value
   - For all these operations, we need to visit each node of the tree.
3. Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data.
4. But a hierarchical data structure like a tree can be traversed in different ways.

Tree Traversals

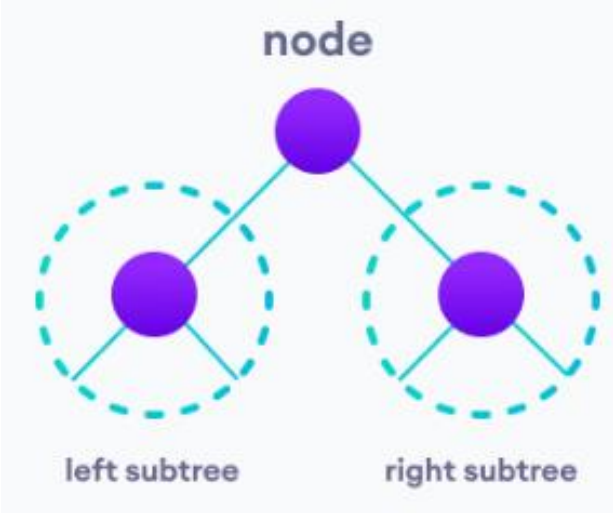
1. Let's think about how we can read the elements of the given tree
2. 1st Method: Starting from top, Left to right
   - 1 -> 12 -> 5 -> 6 -> 9
3. 2nd Method: Starting from bottom, Left to right
   - 5 -> 6 -> 12 -> 9 -> 1
4. Methods are easy
5. But these methods do not respect the hierarchy of the tree, only the depth of the nodes.



Tree Traversals

1. So, we need to use traversal methods that take into account the basic structure of a tree
2. The nodes pointed to by left and right pointers might have other left and right children so we should think of them as sub-trees instead of sub-nodes.
3. According to this structure, every tree is a combination of
   - A node carrying data
   - Two subtrees

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

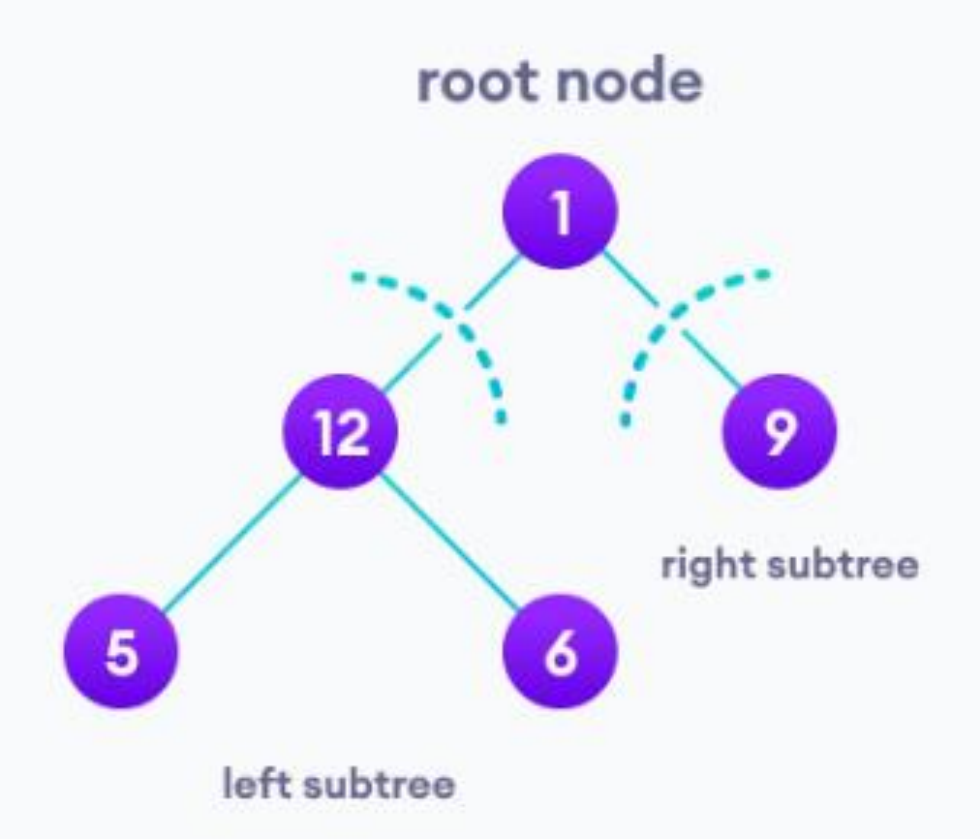


Tree Traversals

1. The goal: to visit each node
2. So we need to visit
   - all the nodes in the left subtree,
   - visit the root node and
   - visit all the nodes in the right subtree as well.
3. Depending on the order in which we do this, there can be three types of traversals.
   - Inorder traversal
   - Preorder traversal and
   - Postorder traversal

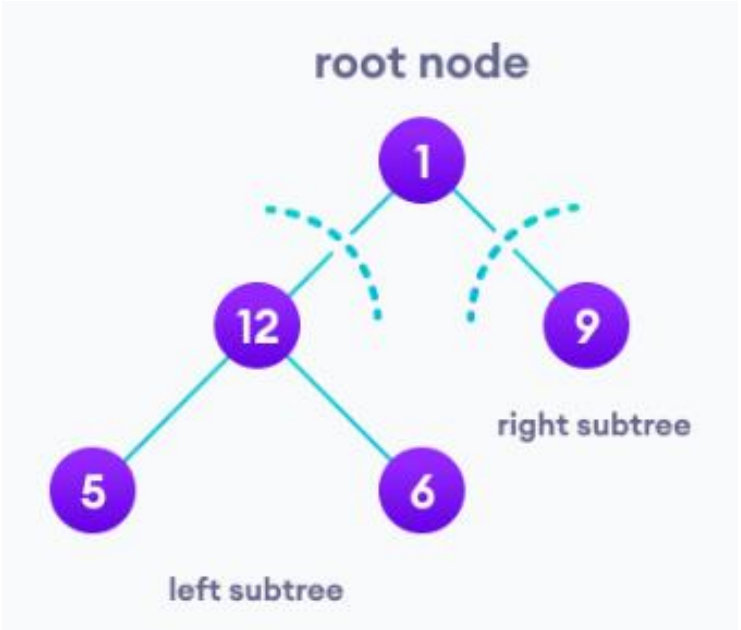
Inorder Traversal

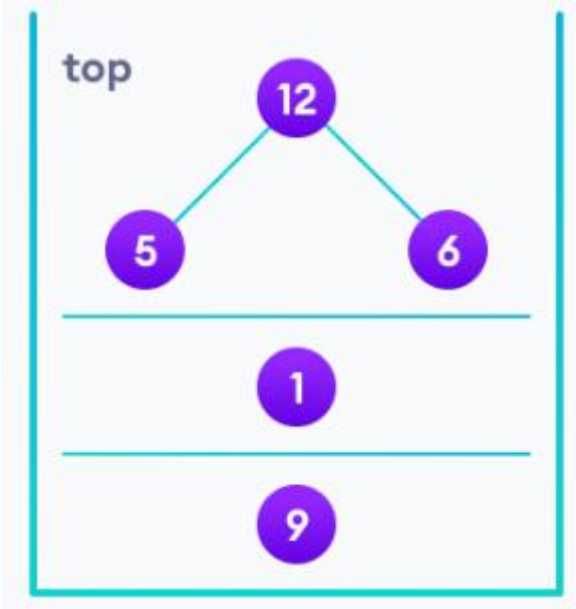
1. Follow the below steps to implement the idea (**left-root-right**):
   - ➢Traverse left subtree
   - ➢Visit the root and print the data.
   - ➢Traverse the right subtree



Inorder Traversal

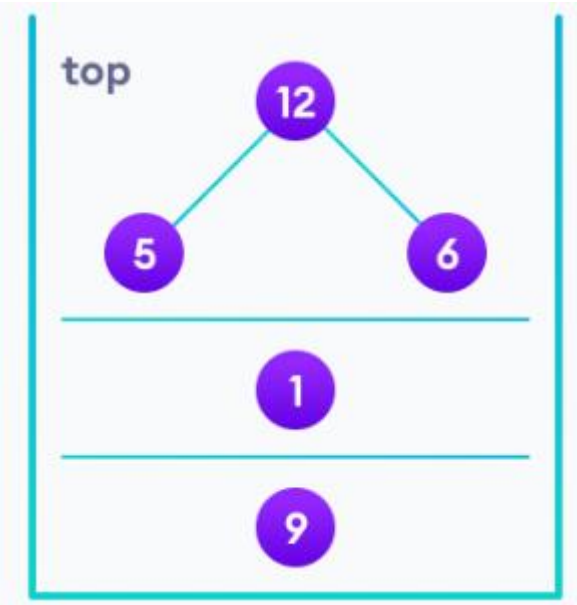
1. We traverse the left subtree first.
2. We also need to remember to visit the root node and the right subtree when this tree is done.
3. Use a stack so that we remember what needs to be done.





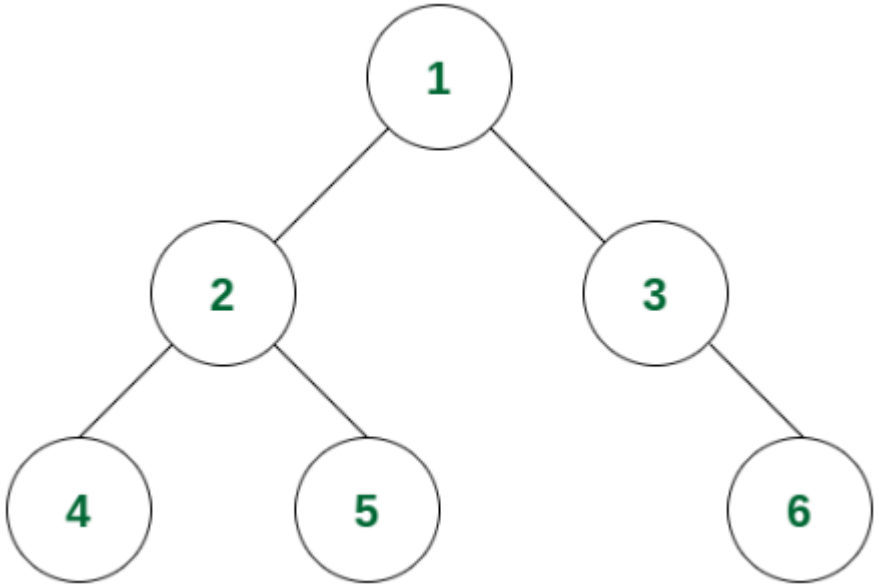
Inorder Traversal

1. Now we traverse to the subtree pointed on the TOP of the stack.
2. Again, we follow the same rule of inorder:
   - Left subtree → root → right subtree
3. After traversing the left subtree, we are left with this stack.
4. Since there are no subtrees left to traverse, we can pop the elements from the stack to get the inorder traversal of the tree.
5. 5 – 12 – 6 – 1 – 9



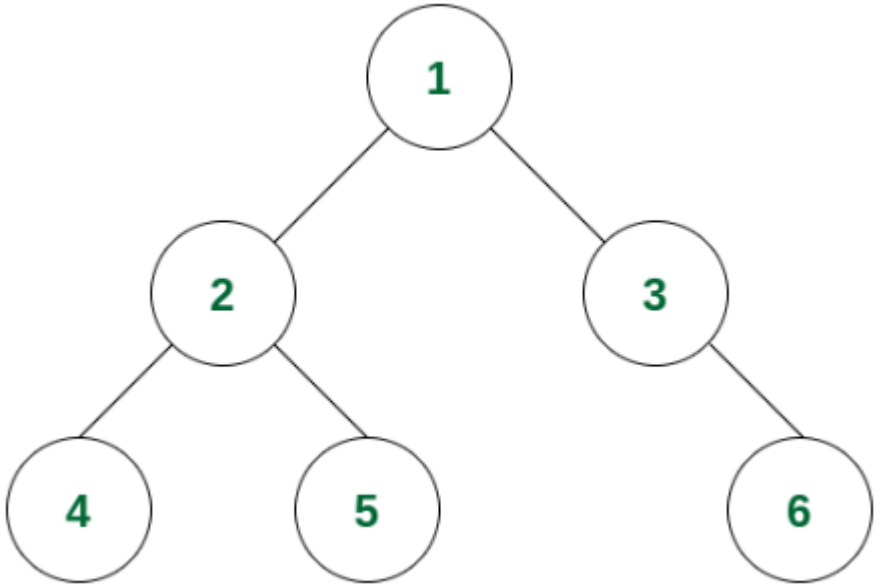
Preorder Traversal

1. Follows the **Root-Left-Right** policy where:
   - ➢ The root node of the subtree is visited first.
   - ➢ Then the left subtree is traversed.
   - ➢ At last, the right subtree is traversed.
2. Step 1: At first the root will be visited, i.e. node 1.
3. Step 2: After this, traverse in the left subtree. Now the root of the left subtree is visited i.e., node 2 is visited.



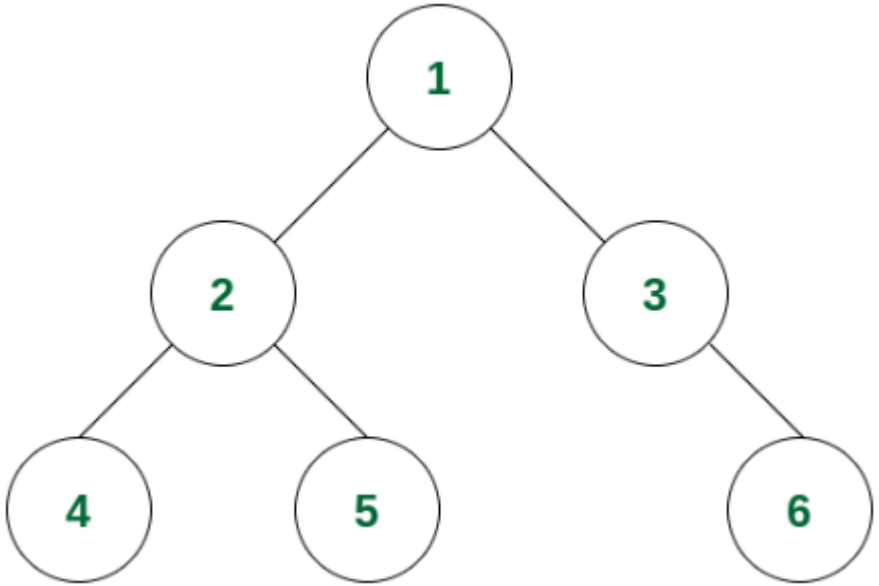
Preorder Traversal

1. Step 3: The left subtree of node 2 is traversed and the root of that subtree i.e., node 4 is visited.
2. Step 4: There is no subtree of 4 and the left subtree of node 2 is visited. So now the right subtree of node 2 will be traversed and the root of that subtree i.e., node 5 will be visited.
3. Step 5: The left subtree of node 1 is visited. So now the right subtree of node 1 will be traversed and the root node i.e., node 3 is visited.
4. Step 6: Node 3 has no left subtree. So the right subtree will be traversed and the root of the subtree i.e., node 6 will be visited. After that there is no node that is not yet traversed. So the traversal ends.
5. So the order of traversal of nodes is
   - 1 -> 2 -> 4 -> 5 -> 3 -> 6.



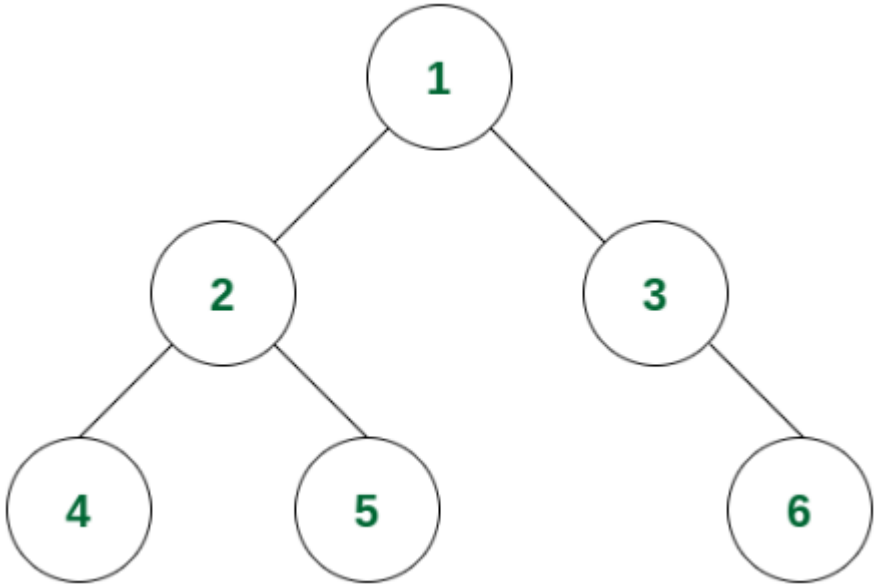
Postorder Traversal

1. Follows the **Left-Right-Root** policy such that for each node:
   - ➢ The left subtree is traversed first
   - ➢ Then the right subtree is traversed
   - ➢ Finally, the root node of the subtree is traversed
2. Step 1: The traversal will go from 1 to its left subtree i.e., 2, then from 2 to its left subtree root, i.e., 4. Now 4 has no subtree, so it will be visited.
3. Step 2: As the left subtree of 2 is visited completely, now it will traverse the right subtree of 2 i.e., it will move to 5. As there is no subtree of 5, it will be visited.



Postorder Traversal

1. Step 3: Now both the left and right subtrees of node 2 are visited. So now visit node 2 itself.
2. Step 4: As the left subtree of node 1 is traversed, it will now move to the right subtree root, i.e., 3. Node 3 does not have any left subtree, so it will traverse the right subtree i.e., 6. Node 6 has no subtree and so it is visited.
3. Step 5: All the subtrees of node 3 are traversed. So now node 3 is visited.
4. Step 6: As all the subtrees of node 1 are traversed, now it is time for node 1 to be visited and the traversal ends after that as the whole tree is traversed.
5. So the order of traversal of nodes is 4 -> 5 -> 2 -> 6 -> 3 -> 1.



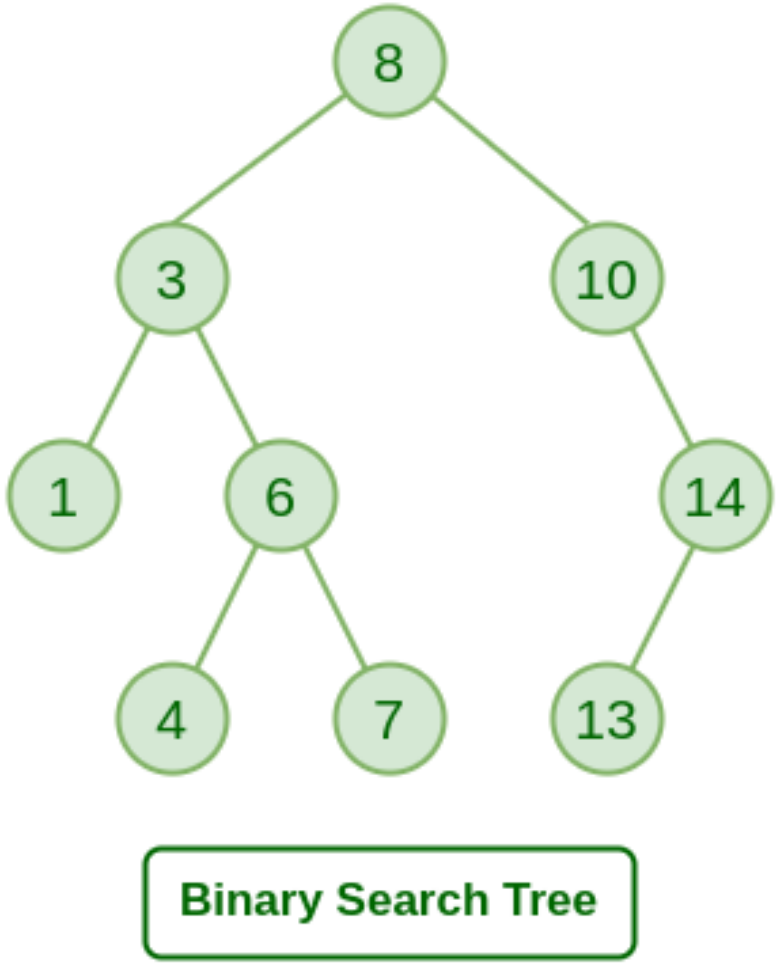
Binary Search Trees

Binary Search Tree

1. Deals with the problem: How should items in a list be stored so that an item can be easily located?
2. Primary goal: To implement a searching algorithm that finds items efficiently when the items are totally ordered.
3. We use Binary Search Trees (BST) for this purpose.

Binary Search Tree

1. A binary search tree (BST) is a binary tree where each node/vertex has a comparable key (associated value) and satisfies the restriction that
   - ➢ the key in any node is larger than the keys in all nodes in that node's left subtree and
   - ➢ the key in any node is smaller than the keys in all nodes in that node's right subtree.
   - ➢ The left and right subtree each must also be a binary search tree.

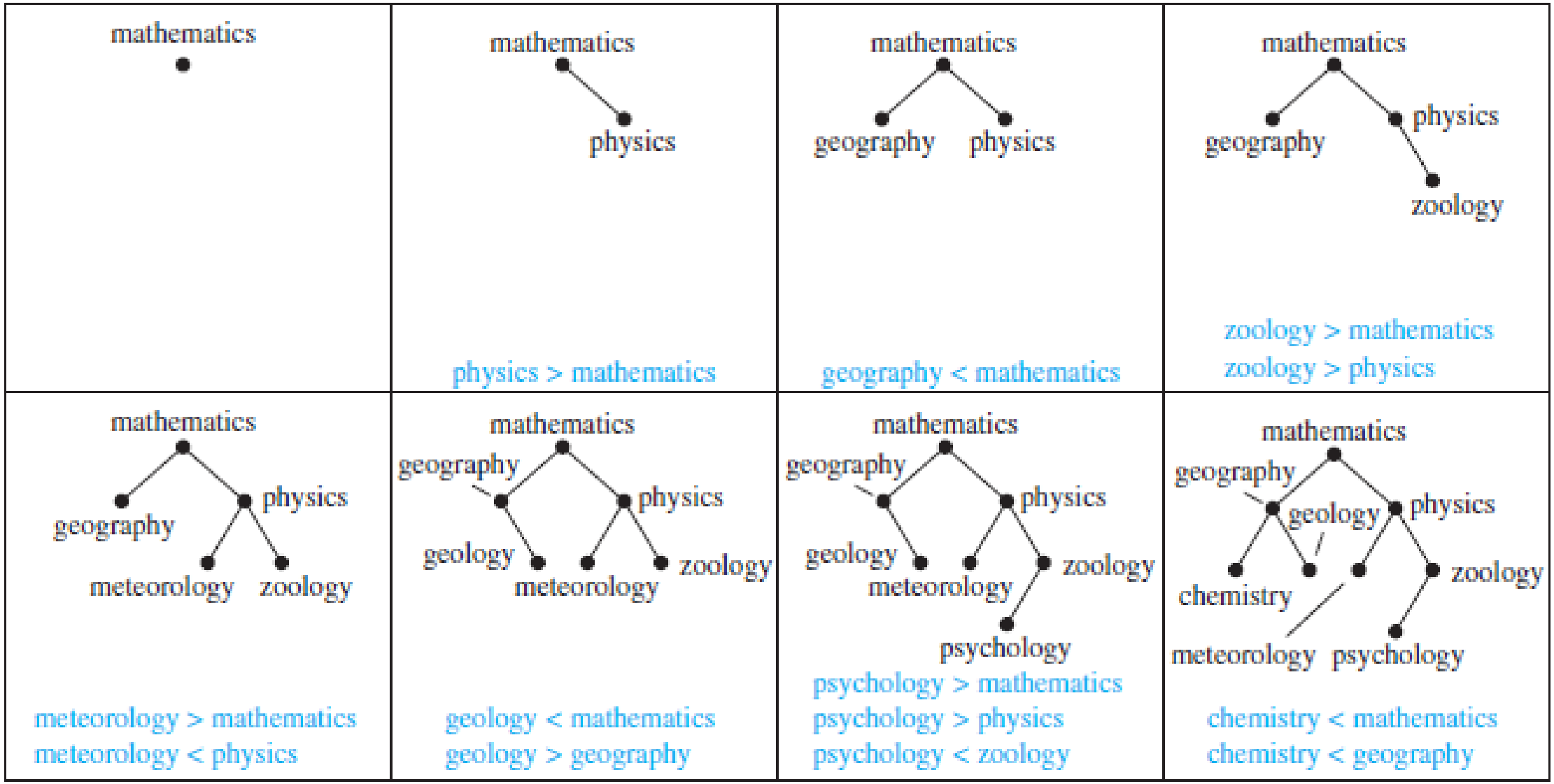


Creating a Binary Search Tree

1. Vertices need to be assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.
2. Given a list of items, do the following:
   - Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root.
   - To add a new item, first compare it with the keys of vertices already in the tree.
   - Starting at the root and moving to the left if the item is less than the key of the respective vertex, or moving to the right if the item is greater than the key of the respective vertex.
   - When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child.
   - Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child

Creating a Binary Search Tree

1. Example: Form a binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).



Binary Search Tree

1. Once we have a binary search tree, we need
   - a way to locate items in the binary search tree,
   - as well as a way to add new items

Binary Search Tree

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

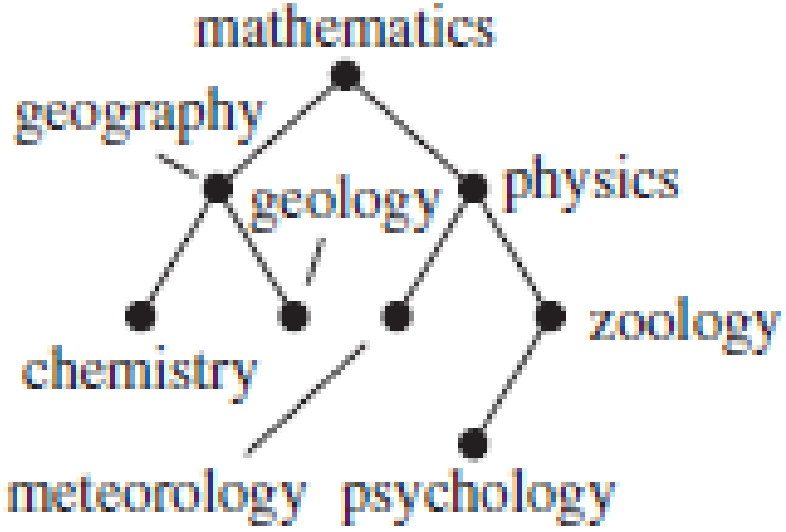
procedure *insertion*(T: binary search tree, x: item)
$v :=$ root of T
{a vertex not present in T has the value *null*}
while $v \neq$ *null* and *label*(v) $\neq x$
    if $x <$ *label*(v) **then**
        if left child of $v \neq$ *null* **then** $v :=$ left child of v
        else add *new vertex* as a left child of v and set $v :=$ *null*
    else
        if right child of $v \neq$ *null* **then** $v :=$ right child of v
        else add *new vertex* as a right child of v and set $v :=$ *null*
if root of $T =$ *null* **then** add a vertex v to the tree and label it with x
else if v is null or *label*(v) $\neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {$v =$ location of x}

Algorithm – Stepwise Explanation

1. Algorithm 1 is a procedure that locates an item x in a binary search tree if it is present, and adds a new vertex with x as its key if x is not present.
2. In the pseudocode, v is the vertex currently under examination and label(v) represents the key of this vertex.
3. The algorithm begins by examining the root.
4. If x equals the key of v, then the algorithm has found the location of x and terminates;
5. if x is less than the key of v, we move to the left child of v and repeat the procedure;
6. if x is greater than the key of v, we move to the right child of v and repeat the procedure.
7. If at any step we attempt to move to a child that is not present, we know that x is not present in the tree, and we add a new vertex as this child with x as its key.

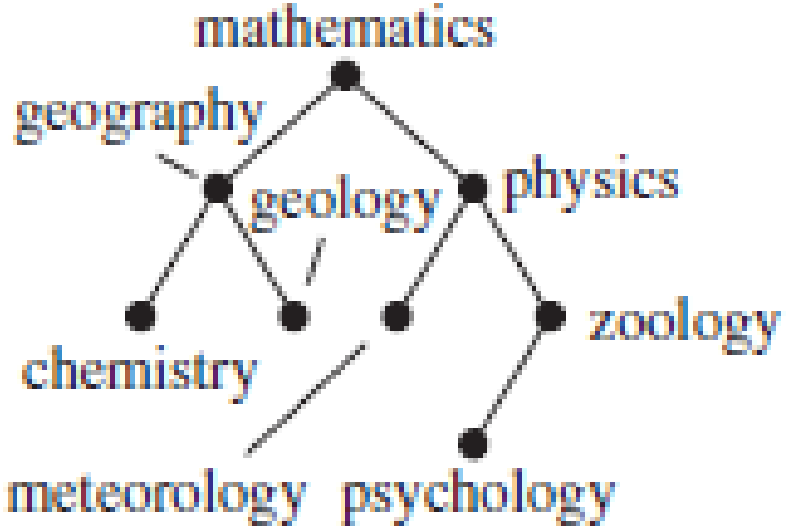
Binary Search Tree

1. **Example: Use Algorithm 1 to insert the word oceanography into the binary search tree in the previous example**
2. Algorithm 1 begins with v, the vertex under examination, equal to the root of T , so label(v) = mathematics.
3. Because v ≠ null and label(v) = mathematics < oceanography, we next examine the right child of the root.
4. This right child exists, so we set v, the vertex under examination, to be this right child.
5. At this step we have v ≠ null and label(v) = physics > oceanography, so we examine the left child of v.



Binary Search Tree

1. This left child exists, so we set v, the vertex under examination, to this left child (which at this point is the vertex with the key meteorology).
2. At this step, we also have v ≠ null and label(v) = meteorology < oceanography, so we try to examine the right child of v.
3. However, this right child does not exist, so we add a new vertex as the right child of v and we set v := null.
4. We now exit the while loop because v = null.
5. Because the root of T is not null and v = null, we use the else if statement at the end of the algorithm to label our new vertex with the key oceanography.



Binary Search Tree

1. Can a binary search tree have duplicate key values? Justify your answer.

Thanks