UNIT 4

Introduction

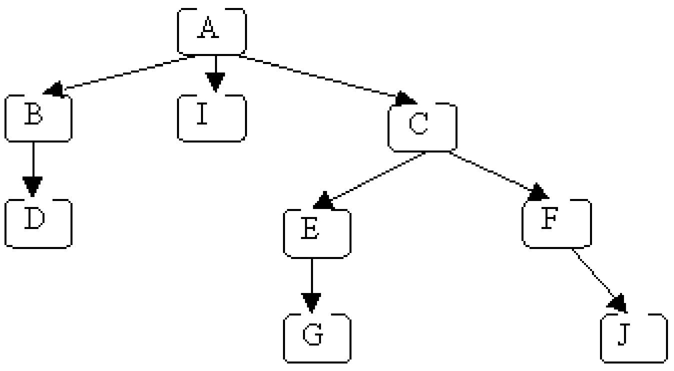
Lists, stacks, and queues, are all **linear** structures: in all three data structures, one item follows another. Trees will be our first non-linear structure:

- More than one item can follow another.
- The number of items that follow can vary from one item to another.

Trees have many uses:

- representing family genealogies
- as the underlying structure in decision-making algorithms
- to represent priority queues (a special kind of tree called a **heap**)
- to provide fast access to information in a database (a special kind of tree called a **b-tree**)

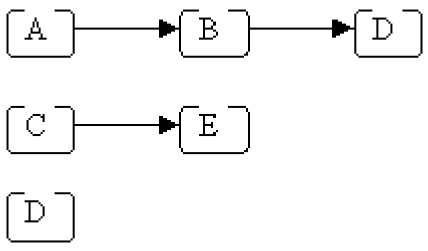
Here is the conceptual picture of a tree (of letters):



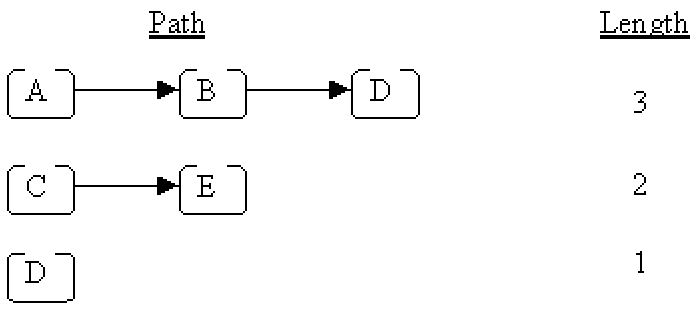
- each letter represents one **node**
- the arrows from one node to another are called **edges**
- the topmost node (with no incoming edges) is the **root** (node A)
- the bottom nodes (with no outgoing edges) are the **leaves** (nodes D, I, G & J)

So a (computer science) tree is kind of like an upside-down real tree...

A **path** in a tree is a sequence of (zero or more) connected nodes; for example, here are 3 of the paths in the tree shown above:



The **length** of a path is the number of nodes in the path, e.g.:

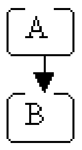


The **height** of a tree is the length of the longest path from the root to a leaf; for the above example, the height is 4 (because the longest path from the root to a leaf is A → C → E → G, or A → C → E → J). An empty tree has height = 0.

The **depth** of a node is the length of the path from the root to that node; for the above example:

- the depth of J is 4
- the depth of D is 3
- the depth of A is 1

Given two connected nodes like this:



Node A is called the **parent**, and node B is called the **child**.

A **subtree** of a given node includes one of its children and all of that child's **descendants**. The descendants of a node **n** are all nodes reachable from **n** (**n**'s children, its children's children, etc.). In the original example, node A has three subtrees:

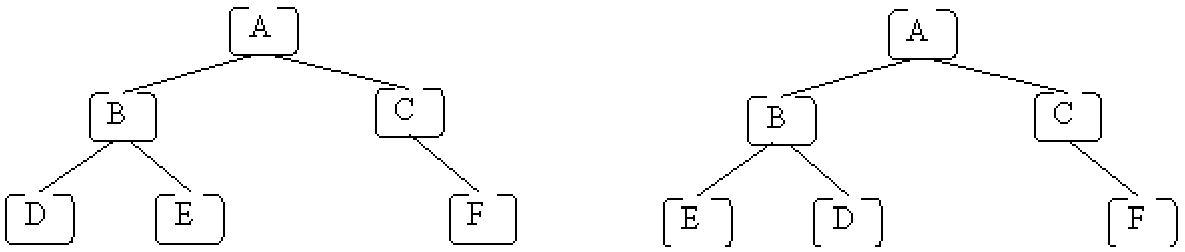
1. B, D
2. I

3. C, E, F, G, J.

An important special kind of tree is the **binary** tree. In a binary tree:

- Each node has 0, 1, or 2 children.
- Each child is either a left child or a right child.

Here are two examples of binary trees that are different:

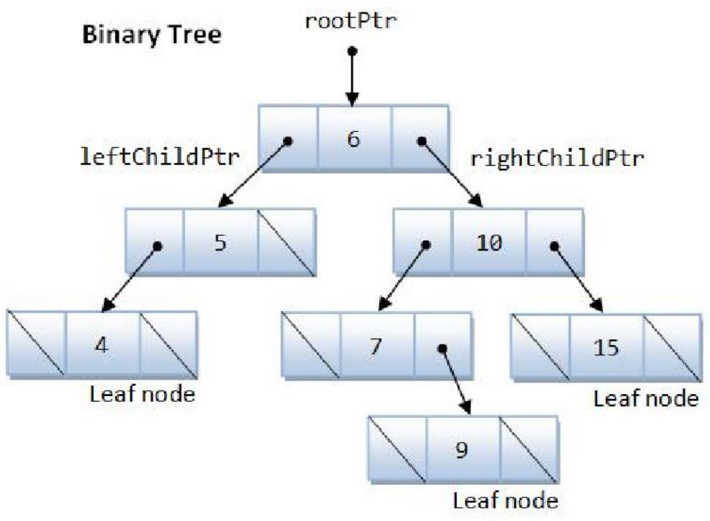


The two trees are different because the children of node B are different: in the first tree, B's left child is D and its right child is E; in the second tree, B's left child is E and its right child is D. Also note that lines are used instead of arrows. We sometimes do this because it is clear that the edge goes from the higher node to the lower node.

Binary trees

A binary tree is a structure comprising nodes, where each node has the following 3 components:

1. Data element: Stores any kind of data in the node
2. Left pointer: Points to the tree on the left side of node
3. Right pointer: Points to the tree on the right side of the node



Commonly-used terminologies

- ✓ Root: Top node in a tree
- ✓ Child: Nodes that are next to each other and connected downwards
- ✓ Parent: Converse notion of child
- ✓ Siblings: Nodes with the same parent
- ✓ Descendant: Node reachable by repeated proceeding from parent to child
- ✓ Ancestor: Node reachable by repeated proceeding from child to parent.
- ✓ Leaf: Node with no children
- ✓ Internal node: Node with at least one child
- ✓ External node: Node with no children

Structure code of a tree node

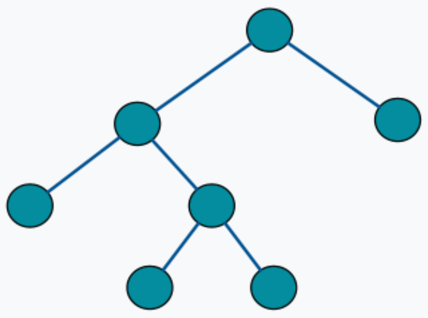
In programming, trees are declared as follows:

```
struct node

   {

        int data;                  //Data element

        struct node * left;         //Pointer to left node

        struct node * right;        //Pointer to right node

   };
```

Types of binary tree

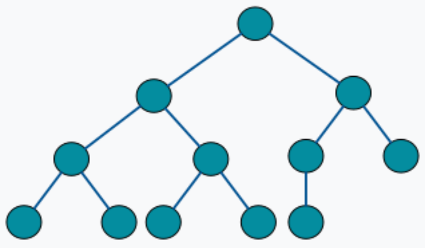
Tree terminology is not well-standardized and so varies in the literature.

- A **rooted** binary tree has a root node and every node has at most two children.



A full binary tree

- A **full** binary tree (sometimes referred to as a **proper** or **plane** binary tree) is a tree in which every node has either 0 or 2 children. Another way of defining a full binary tree is a recursive definition. A full binary tree is either:
  - A single vertex.
  - A graph formed by taking two (full) binary trees, adding a vertex, and adding an edge directed from the new vertex to the root of each 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. It can have between 1 and 2^h nodes at the last level h. An alternative definition is a perfect tree whose rightmost leaves (perhaps all) have been removed. Some authors use the term **complete** to refer instead to a perfect binary tree as defined above, in which case they call this type of tree an **almost complete** binary tree or **nearly complete** binary tree. A complete binary tree can be efficiently represented using an array.



A complete binary tree

- A **perfect** binary tree is a binary tree in which all interior nodes have two children *and* all leaves have the same *depth* or same *level*. An example of a perfect binary tree is the ancestry chart of a person to a given depth, as each person has exactly two biological parents (one mother and one father).
- In the **infinite complete** binary tree, every node has two children (and so the set of levels is countably infinite). The set of all nodes is countably infinite, but the set of all infinite paths from the root is uncountable, having the cardinality of the continuum. These paths correspond by an order-preserving

bijection to the points of the Cantor set, or (using the example of a Stern–Brocot tree) to the set of positive irrational numbers.

- A **balanced** binary tree is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1. One may also consider binary trees where no leaf is much farther away from the root than any other leaf. (Different balancing schemes allow different definitions of "much farther".)
- A **degenerate** (or **pathological**) tree is where each parent node has only one associated child node. This means that performance-wise[the tree will behave like a linked list data structure.

Traversing the tree

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)
```

```
void printInorder(struct Node* node)
{
   if (node == NULL)
      return;
   printInorder(node->left);
   cout << node->data << " ";
   printInorder(node->right);
}
```

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)
```

```
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;
    cout << node->data << " ";
    printPreorder(node->left);
    printPreorder(node->right);
}
```

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.
```

```
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;
    printPostorder(node->left);
    printPostorder(node->right);
    cout << node->data << " ";
}
```

Main applications of trees include

1. Manipulate hierarchical data.

2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms

6. Form of a multi-stage decision-making.

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure 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.

Binary Search Tree (Search and Insertion)

- ✓ The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.
- ✓ To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root’s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Binary Search Tree (Delete)

When we delete a node, three possibilities arise.

1. **Node to be deleted is leaf:** Simply remove from the tree.
2. **Node to be deleted has only one child:** Copy the child to the node and delete the child
3. **Node to be deleted has two child:** Replace with highest value in left sub tree or least value in right sub tree and delete the node.