

Trees

What Is a Tree?

A tree consists of **nodes** connected by **edges**. Figure 1 shows a tree. In such a picture of a tree the nodes are represented as circles, and the edges as lines connecting the circles.

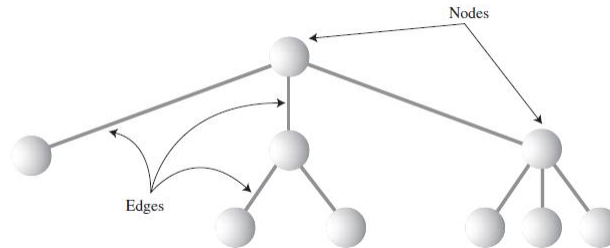


Figure 1: A general (non-binary) tree.

Why might you want to use a tree?

Usually, because it combines the advantages of two other structures:

- ✓ An ordered array and
- ✓ A linked list.

You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list.

In computer programs, **nodes** often represent such entities as people, car parts, airline reservations, and so on.

Edges are likely to be represented in a program by **references**, if the program is written in Java.

Typically, **there is one node in the top row of a tree**, with lines connecting to more nodes on the second row, even more on the third, and so on. **Thus, trees are small on the top and large on the bottom.**

This may seem upside-down compared with real trees, but generally a program starts an operation at the **small end of the tree**, and it's (arguably) more natural to think about going from top to bottom, as in reading text. There are different kinds of trees.

Tree Terminology

Many terms are used to describe particular aspects of trees. Fortunately, most of these terms are related to real-world trees or to family relationships (as in parents and children), so they're not hard to remember. Figure 2 shows many of these terms applied to a binary tree. **H, E, I, J, and G are leaf nodes.**

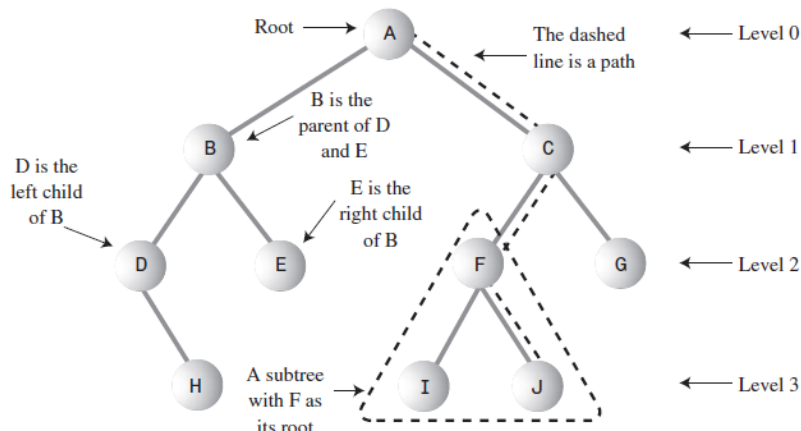


Figure 2: Tree terms.

Path

Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a *path*.

Root

The node at the **top** of the tree is called the **root**. There is only one root in a tree. For a collection of nodes and edges to be defined as a tree, **there must be one path from the root to any other node**. Figure 3 shows a non-tree. You can see that it violates this rule.

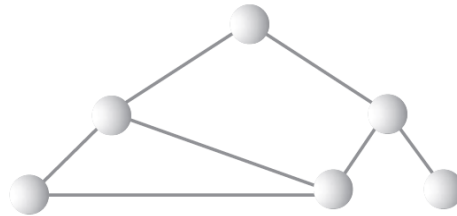


Figure 3: A non-trees.

Parent

Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node.

Child

Any node may have one or more lines running downward to other nodes. These nodes below a given node are called its *children*.

Leaf

A node that has no children is called a *leaf node* or simply a *leaf*. There can be only one root in a tree, but there can be many leaves.

Subtree

Any node may be considered to be the root of a *subtree*, **which consists of its children**, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its descendants.

Visiting

A node is *visited* when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or displaying it.

Traversing

To *traverse* a tree means to **visit all the nodes in some specified order**. For example, you might visit all the nodes in order of ascending key value.

Levels

The *level* of a particular node refers to how many generations the node is from the root. If we assume the root is Level 0, and then its children will be Level 1, its grandchildren will be Level 2, and so on.

Keys

We've seen that one data field in an object is usually designated a *key value*. This value is used to search for the item or perform other operations on it. In tree diagrams, when a circle represents a node holding a data item, the key value of the item is typically shown in the circle.

Binary Trees

If every node in a tree can have **at most two children**, the tree is called a **binary tree**. The two children of each node in a binary tree are called the **left child** and the **right child**, corresponding to their positions when you draw a picture of a tree, as shown in **Figure 4**. A node in a binary tree doesn't necessarily have the maximum of two children; it may have only a left child, or only a right child or it can have no children at all (in which case it's a leaf). The kind of binary tree we'll be dealing with in this lecture is technically called a **binary search tree**.

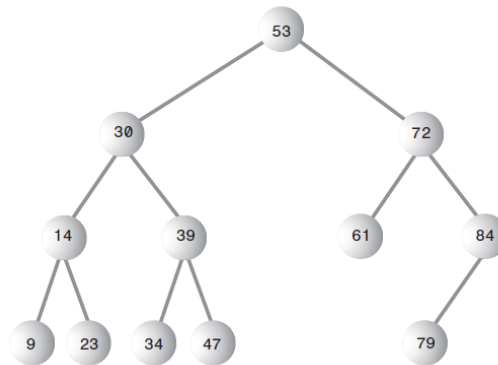


Figure 4: A binary search trees.

NOTE

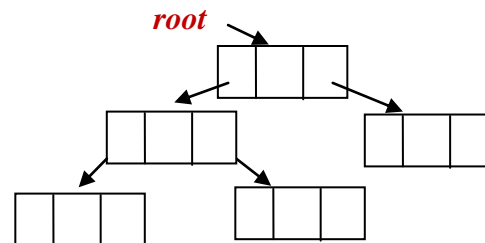
The defining characteristic of a binary search tree is this: A node's left child must have a key less than its parent, and a node's right child must have a key greater than or equal to its parent.

Representing the Tree

To store the nodes of a tree at unrelated locations in the computer's memory, you must connect them **using references** in each node that point to its children.

```
public class node
{
    int Data ;           // data used as key value
    node left ;
    node right ;

    public node (int d)   // constructor
    {
        Data = d;
    }
}
```



Root initially is null, and it is **static node**. Some programmers also include a reference to the node's parent. This simplifies some operations but complicates others, so we don't include it.

To create a binary tree, you can build a function, each call for it inserts (adds) one node to the tree, so that, the tree will be built by calling that function as you need (within for or while) with sending to it the read value you like to add it. You can use iterative manner or recursive manner to build this function.

There are many methods for finding, inserting, and deleting nodes, different kinds of traverses, and displaying the tree.

Slow Search, Insertion, Deletion in an Ordered Array

Imagine an array in which all the elements are arranged in order (**an ordered array**). You can quickly search such an array for a particular value, using a **binary search**.

You **check in the center of the array**; if the object you're **looking for is greater than** what you find there, you **narrow your search to the top half of the array**; if it's less, you narrow your **search to the bottom half**.

On the other hand, if you want to insert a new item into an **ordered array**, you first need to find where the item will go, and then move (shift) all the item with **greater keys up** one space in the array to make room for it.

Deletion involves the same multi move operation and is thus equally slow. If you're going to be doing a lot of insertions and deletions, an ordered **array is a bad choice**.

////////////////////////////////////

// Iterative manner of adding one node to tree

Main()

```
{ // read n   : number of nodes in tree
  // read d   : value of the root node
  Root = new node (d)
  For I = 1 to n-1      // n-1 because the root was created outside the for
  {
    // read d
    Insert (d)
  }
}
```

public void insert (int val)

```
{
  node current = root ;
  while (current != null)
  {
    If (val <= current . Data)
    {
      // insert left
      if (current . left == null)
      {
        current . left = new node (val) ;
        return;
      }
      else
        current = current . left;
    }
    Else
    {
      // insert right
      if (current . right == null)
      {
        current . right = new node (val) ;
        return;
      }
      else
        current = current . right;
    }
  }
}
```

// **Recursive** manner of adding one node to tree.

Main()

```
{ // read n   : number of nodes in tree
  // read d   : value of the root node
  Root = new node (d)
  For I = 1 to n-1      // n-1 because the root was created outside the for
  {
    // read d
    Insert (root, d)
  }
}
```

private static void **Insert** (**node** current , int val)

```
{
  If (val <= current . Data)
  {
    if (current . left == null)
      current . left = new node (val) ;
    else
      Insert ( current . left , val) ;
  }
  Else
  {
    if (current . right == null)
      current . right = new node (val) ;
    else
      Insert ( current . right , val) ;
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```