جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الانظمة الطبية الذكية
Intelligent Medical Systems Department

Subject: Data Structure

Class: Second

Lecturer: Asst. Prof. Mehdi Ebady Manaa

Lecture: (11)

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.

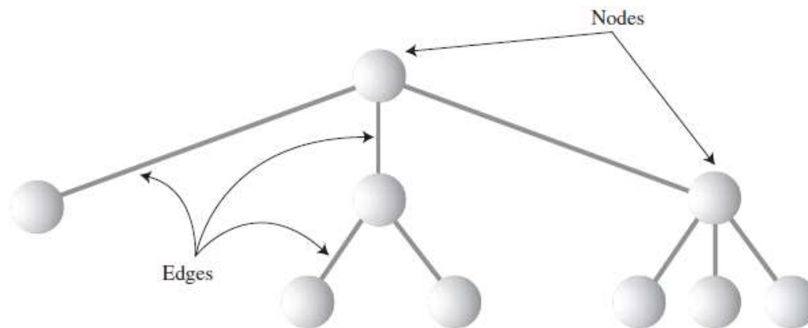


Figure 1: A general (non-binary) tree.

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.

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

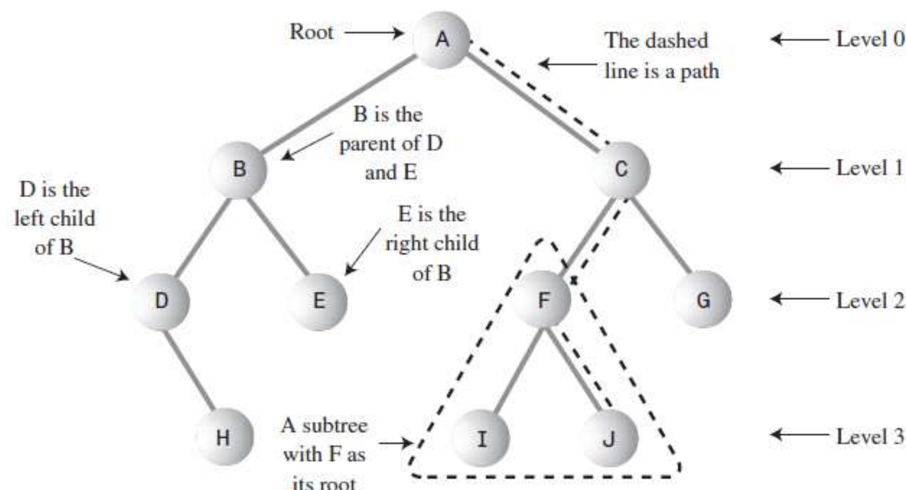


Figure 2: Tree terms. (**H, E, I, J, and G are leaf nodes**)

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** (and only one!) **path from the root to any other node**. Figure 3 shows a non-tree. You can see that it violates this rule.

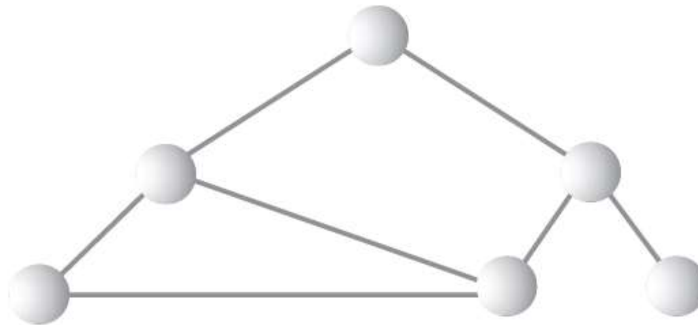


Figure 3: A non-trees.

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. Merely passing over a node on the path from one node to another is not considered to be visiting the node.

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 2**. 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 discussion is technically called a *binary search tree*.

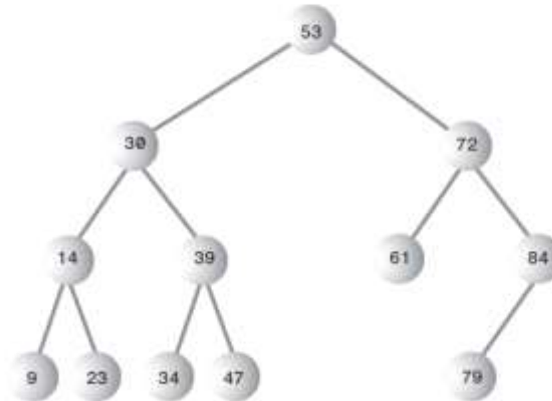


Figure 4: A binary search trees.

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 in Java Code

Let's see how we might implement a binary tree in Java. As with other data structures, there are several approaches to representing a tree in the computer's memory.

The most common is to store the nodes at unrelated locations in memory, and connect them **using references** in each node that point to its children. You can also represent a tree in memory as an **array**, with nodes in specific positions stored in corresponding positions in the array. For our sample Java code we'll use the approach of connecting the nodes using **references.**

The Node Class: First, we need a class of node objects. These objects contain the data representing the objects being stored (employees in an employee database, for example) and also references to each of the node's two children. Here's how that looks:

```
class Node
  {
```

```
  int iData;                 // data used as key value

  float fData;               // other data

  node leftChild;            // this node's left child

  node rightChild;           // this node's right child


  public void displayNode()

    {

   } }
```

Some programmers also include a reference to the node's parent. This simplifies some operations but complicates others, so we don't include it. We do include a method called displayNode()to display the node's data, but its code isn't relevant here.

There are other approaches to designing class Node. Instead of placing the data items directly into the node, **you could use a reference** to an object representing the data item:

```
class Node

  {

  person p1;                 // reference to person object

  node leftChild;            // this node's left child

  node rightChild;           // this node's right child

  }

class person

  {

  int iData;

  float fData;

  }
```

This approach makes it conceptually clearer that the node and the data item it holds aren't the same thing. But it results in somewhat more complicated code, so we'll stick to the first approach.

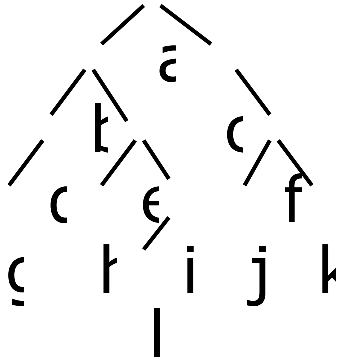
<u>The Tree Class:</u> We'll also **need a class** from which to instantiate the tree itself: the object that holds all the nodes. **We'll call this class Tree**. It has only one field: a Node variable that holds the root. It doesn't need fields for the other nodes because they are all accessed from the root.The "**Tree class has a number of methods"**. They are used for finding, inserting, and deleting nodes; for different kinds of traverses; and for displaying the tree. Here's a skeleton version:

```
class Tree
  {
  private Node root;            // the only data field in Tree


  public void find(int key)  {
                                            }
  public void insert(int id, double dd)   {
                                                              }
  public void delete(int id)   {
                                               }
  // various other methods
  }  // end class Tree
```

<u>Tree: Levels, Depth and Height</u>

- The **depth** of node **n** is the length of the path from the root <u>to the node</u>. The set of all nodes at a given depth is sometimes called a **level** of the tree. The root node is at depth zero.



- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero
  - e is at depth 2
- The depth of a binary tree is the depth of its deepest node
  - This tree has depth 4

- The **height** of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of **zero**, i.e., number of nodes which must be traversed from the root to reach a leaf of a tree
- The degree of a node is the number of subtrees of the node
- The node with **degree 0** is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

No. of Nodes on Binary Tree is $\mathbf{2^L}$, where the **L** is the level

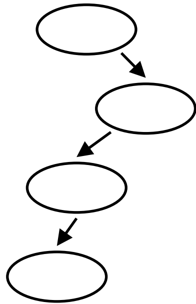
Binary Tree Traversals

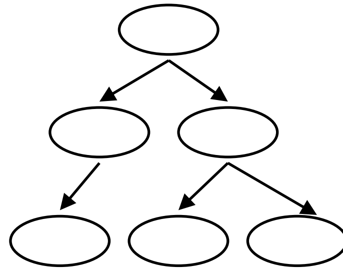
- only 3 traversals remain
  - inorder, postorder, preorder

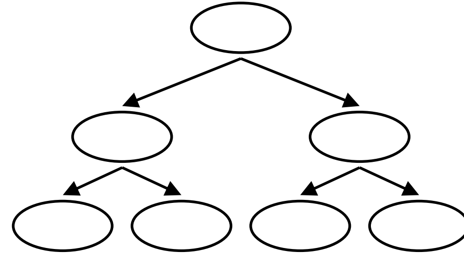
Types of Binary Trees

- Degenerate – only one child
- Complete – always two children

- Balanced – “mostly” two children
    - more formal definitions exist, above are intuitive ideas


Degenerate binary tree (similar to linked list)


Balanced binary tree(useful for search)

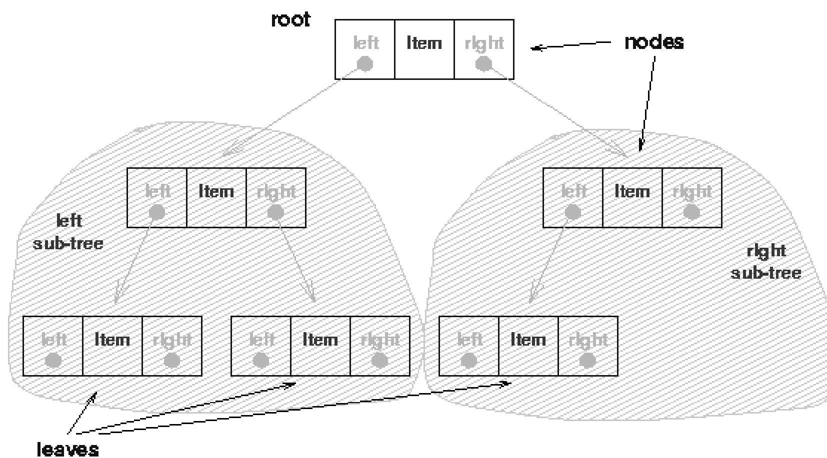

Complete binary tree

Binary Search Properties

The simplest form of tree is a **binary tree**. A binary tree consists of

- a *node* (called the **root** node) and
- left and right sub-trees. Both the sub-trees are themselves binary trees.

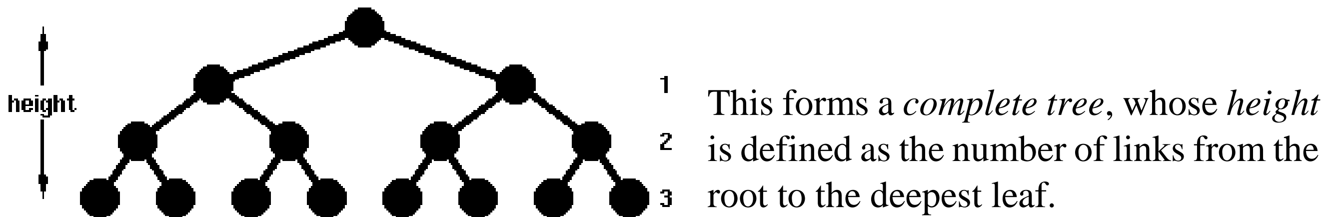
You now have a *recursively defined data structure*. (It is also possible to define a list recursively):



Binary tree

Complete Trees

Before we look at more general cases, let's make the optimistic assumption that we've managed to fill our tree neatly, i.e., that each leaf is the same 'distance' from the root.



This forms a *complete tree*, whose *height* is defined as the number of links from the root to the deepest leaf.

Complete tree

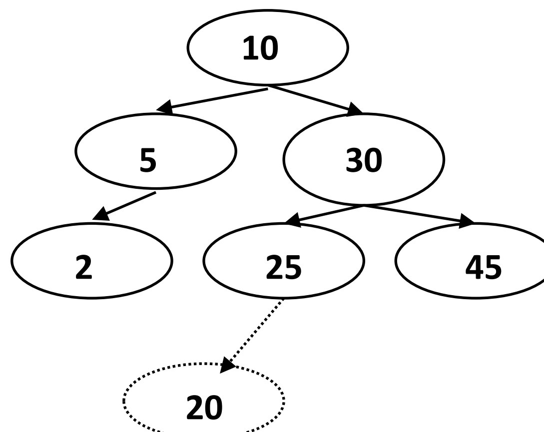
Binary Search Tree processes:

1-Binary Search Tree – Insertion

- Algorithm
    1. Perform search for value X
    2. Search will end at node Y (if X not in tree)
    3. If $X < Y$, insert new leaf X as new left subtree for Y
    4. If $X > Y$, insert new leaf X as new right subtree for Y
- Observations
    1. Insertions may unbalance tree

Example Insertion

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

2-Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

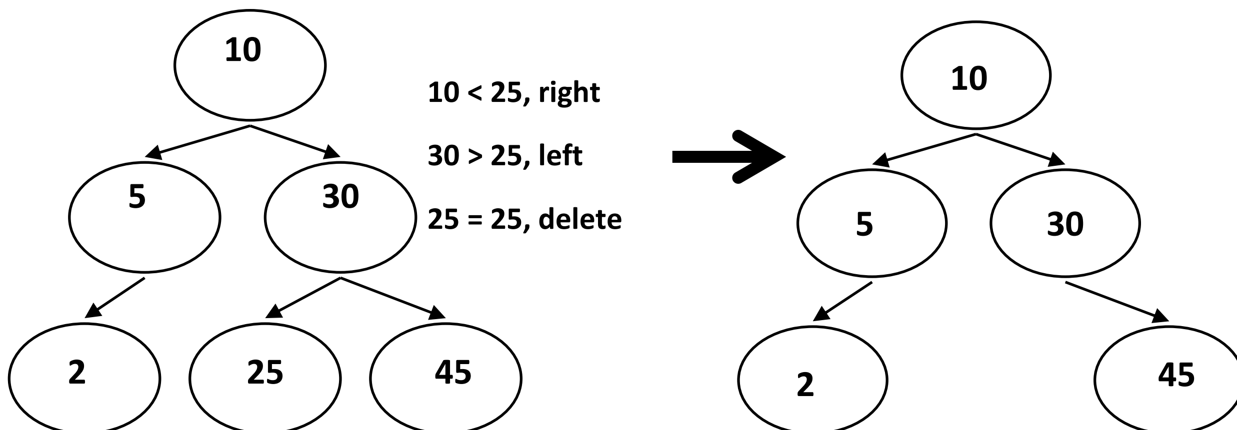
     a) **Replace with largest value Y on left subtree**

     OR smallest value Z on right subtree

  b) Delete replacement value (Y or Z) from subtree
- Observation
  - Deletions may unbalance tree

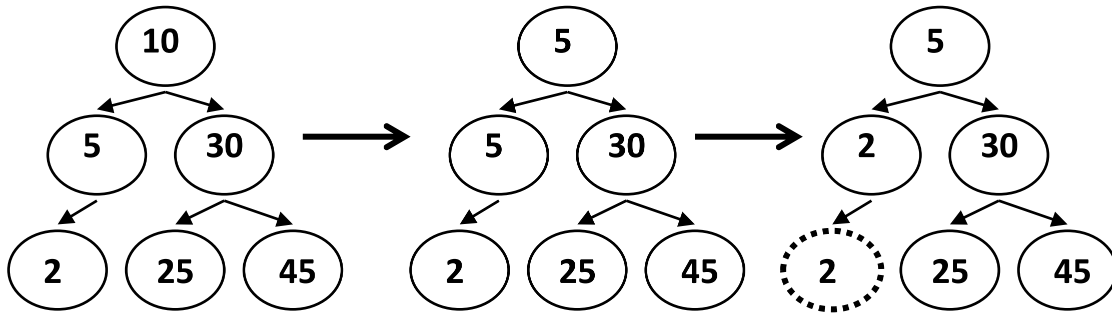
Example Deletion (Leaf)

Delete (25)



Example Deletion (Internal Node)

- Delete (10)



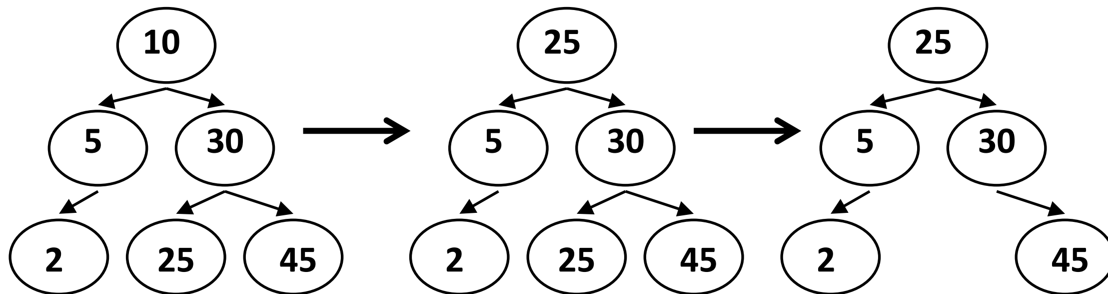
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

3- Finding a Node: **Finding a node with a specific key** is the simplest of the major tree operations.

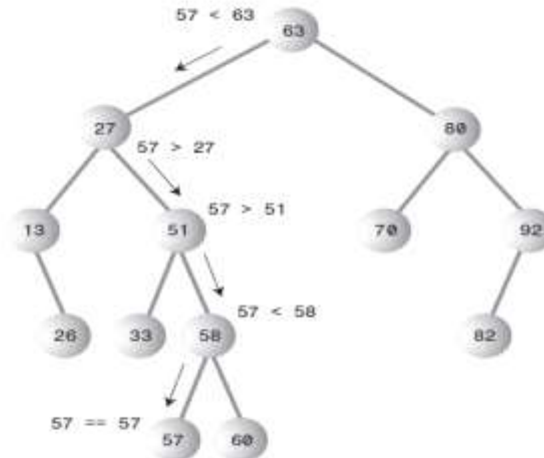


Figure 5: Finding node 57.

In **Figure 5** the arrow starts at the root.

- ✓ The program compares the key value 57 with the value at the **root**, which is 63 and ,
- ✓ **The key is less**, so the program knows the desired node must be on the left side of the tree—either the root's left child or one of this child's descendants.
- ✓ The **left child** of the root has the value 27, so the comparison of 57 and 27 will show that the desired node is in the right **subtree** of 27.
- ✓ The arrow will go to 51, the root of this **subtree**. Here, 57 is again greater than the 51 node, **so we go to the right**, to 58, and then to the **left**, to 57.
- ✓ This time the comparison shows 57 equals the node's key value, so we've found the node we want.

Balanced Search Trees

- Kinds of balanced binary search trees
  - height balanced vs. weight balanced
  - "Tree rotations" used to maintain balance on insert/delete

- Non-binary search trees

    - **2/3 trees:** each internal node has 2 or 3 children**,** all leaves at same depth (height balanced)

    - **B-trees :**Generalization of 2/3 trees**,** Each internal node has between k/2 and k children

        - Each node has an array of pointers to children, Widely used in databases

Other (Non-Search) Trees

- Parse trees

    - Convert from textual representation to tree representation

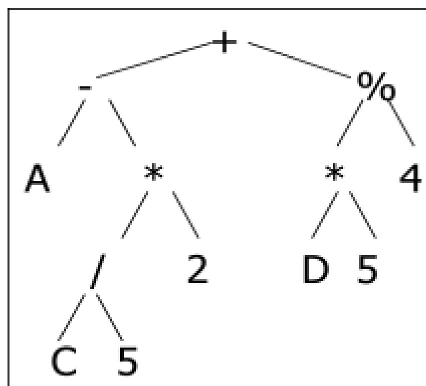
    - Textual program to tree: Used extensively in compilers

    - Tree representation of data

        - E.g. HTML data can be represented as a tree, called DOM (Document Object Model) tree

        - XML: Like HTML, but used to represent data, Tree structured

Parse Trees: Expressions, programs, etc. can be represented by tree structures**,** E.g. Arithmetic Expression Tree**,** A-(C/5 * 2) + (D*5 % 4)



Tree Traversal: Goal: visit every node of a tree

- **in-order traversal, the output: A – (C / 5 * 2) + (D * 5 % 4)**