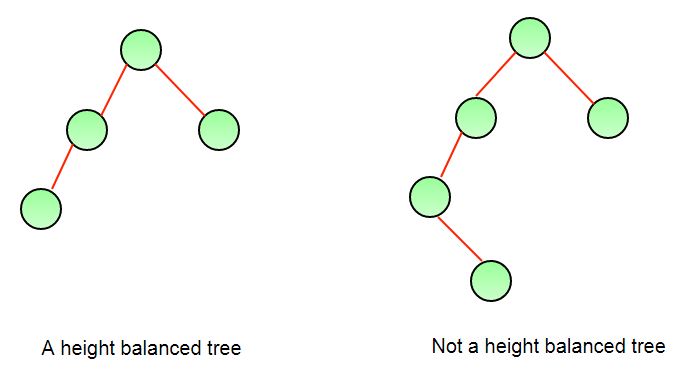
**Lesson 18 – Trees II**

**Reading Assignment:**

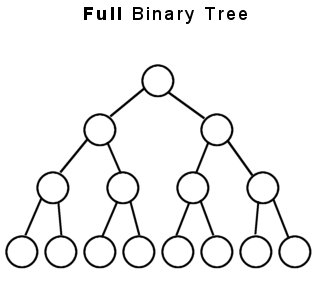
* Chapter 11, Sections 2 and 3 of the text.

**Learning Objectives:**

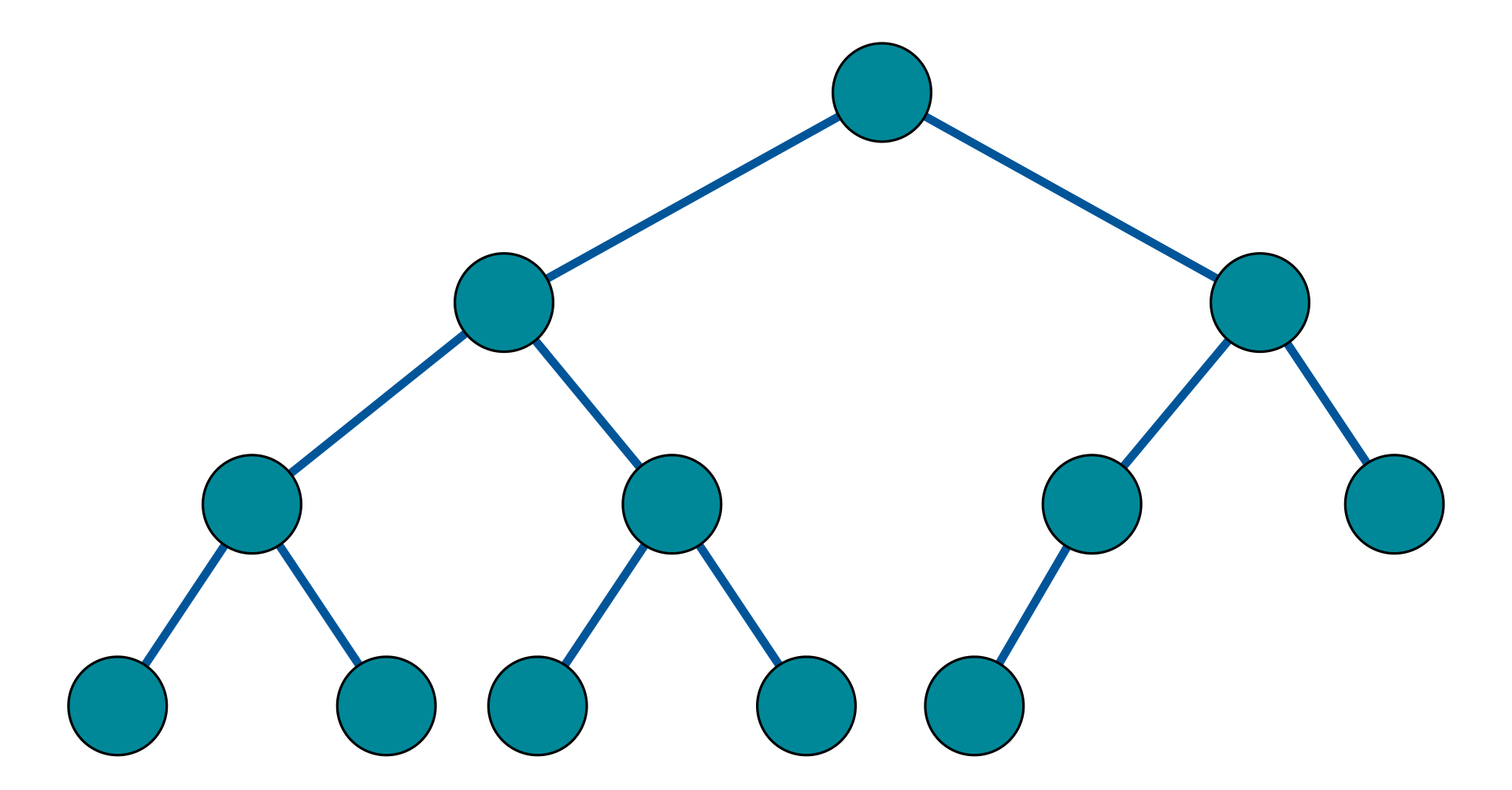
* List the basic operations of the ADT Binary Tree
* Describe the algorithm for traversing a Binary Tree
* Program a reference-based implementation of a Binary Tree
* A binary tree is **height balanced**, or simply **balanced**, if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.
  + For the height-balanced binary trees the height is defined to be logarithmic **O(log2 n)** in the number *n* of items.



* In a **full binary tree** of height *h*, all nodes that are at a level less than *h* have two children each. Intuitively, a full binary tree has no missing nodes. When proving properties about full binary trees—such as how many nodes they have—the following recursive definition of a full binary tree is convenient:
  + If *T* is empty, *T* is a full binary tree of height 0;
  + If *T* is not empty and has height *h* > 0, *T* is a full binary tree if its root’s subtrees are both full binary trees of height *h* – 1.



* A **complete binary tree** of height *h* is a binary tree that is full down to level *h* – 1, with level *h* filled in from left to right. More formally, a binary tree *T* of height *h* is complete if:
  + All nodes at level *h* – 2 and above have two children each, and
  + When a node at level *h* – 1 has children, all siblings to its left have two children each, and
  + When a node at level *h* – 1 has one child, it is a left child.
* A complete binary tree is balanced.



**Tree Traversals:**

* When traversing any binary tree, an algorithm has three choices of when to “visit” the root *r*.
  + preorder,
  + inorder, and
  + postorder
* The preorder traversal algorithm is as follows:

// Traverses the binary tree binTree in preorder.

// Assumes that “visit a node” means to display

// the node’s data item.

+preorder(in binTree:BinaryTree)

{

if (binTree is not empty)

{

Display the data in the root of binTree

preorder(Left subtree of binTree’s root)

preorder(Right subtree of binTree’s root)

}

}

* The inorder traversal algorithm is as follows:

// Traverses the binary tree binTree in inorder.

// Assumes that “visit a node” means to display

// the node’s data item.

+inorder(in binTree:BinaryTree)

{

if (binTree is not empty)

{

inorder(Left subtree of binTree’s root)

Display the data in the root of binTree

inorder(Right subtree of binTree’s root)

}

}

* The postorder traversal algorithm is as follows:

// Traverses the binary tree binTree in postorder.

// Assumes that “visit a node” means to display

// the node’s data item.

+postorder(in binTree:BinaryTree)

{

if (binTree is not empty)

{

postorder(Left subtree of binTree’s root)

postorder(Right subtree of binTree’s root)

Display the data in the root of binTree

}

}

  public String inOrder() {

        String s = "";

        if (!this.isEmpty()) {

            s += this.left.inOrder();

            s += " " + this.root.toString() + " ";

            s += this.right.inOrder();

        }

        return s;

    }

    public String postOrder() {

        String s = "";

        if (!this.isEmpty()) {

            s += this.left.postOrder();

            s += this.right.postOrder();

            s += " " + this.root.toString() + " ";

        }

        return s;

    }

    public String preOrder() {

        String s = "";

        if (!this.isEmpty()) {

            s += " " + this.root.toString() + " ";

            s += this.left.postOrder();

            s += this.right.postOrder();

        }

        return s;

    }

**Exam 02 Review:**

* Points: 100
* Exam sections:
  + Matching (10 Points)
  + True/False (20 points)
  + Multiple Choice (30 points)
  + Short answer (30 points)
  + Coding (10 points)
* Topics:
  + Be able to describe algorithms or code methods for array-based or reference-based implementations of:
    - Stack ADT
    - Queue ADT
    - Priority Queue ADT
  + Be able to describe algorithms or code methods for variants of the Node class.
  + Big-O Analysis:
    - Simplify cost functions.
    - Determine time-complexity of a given algorithm.
    - Determine time-complexity for an operation for a particular ADT:
      * array-based.
      * reference-based.
  + Sorting Algorithms:
    - Comparison-based vs. non-comparison-based.
    - High-level understanding of common sorting algorithms.
    - Big-O complexity of common sorting algorithms.
  + Trees:
    - General Trees.
    - Binary Trees.
    - Full Binary Trees.
    - Complete Binary Trees.
    - Binary Search Trees.