Each node has exactly 1 parent, except root (unique) has 0 parents.

Tree: made up of \( \text{Nodes} \Rightarrow \text{Node class} \Rightarrow \text{root} \Rightarrow \text{List: Node; List has a ref to head} \)

public class Node < T > {

    T content; // stuff node has

    List < Node > children; // holds all the children

    list of children for each of these nodes
public class Tree<T> {
    Node<T> root;

    // Methods
    
    // Insert / remove content

    Pre-order: process root node before its descendants
    Post-order: process node after all its descendants.

    Pre-order - typical for processing game playing trees

    Algorithm Pre-order (Node m)
    process (m)
    for each c ∈ m.children
        esp Preorder (c)

    Post-order traversal:
    for each c ∈ m.children
        PostOrder (c)
    process (m)
    Preorder (root) or Postorder (root)
Intuitively: $O(n)$ where $n$ is number of nodes in tree

$$T(n) = \frac{\text{Process (root)}}{O(1)} + \sum_{c \in \text{root.children}} T(m_c)$$

$m_c =$ number of nodes under child $c$

Notice: $n = 1 + \sum \frac{1}{c} m_c$

To make algorithms faster, we'd like to eliminate the sum => look at particular sub-trees

Depth / height of a node:

Depth of a node (recursively):
- depth (root) = 0
- depth (node) = 1 + depth (parent)
Height of a node

- Height (leaf) = 0
- Height (node) = 1 + \max_{c \in \text{node}.\text{children}} \text{height (c)}

**Binary Trees:** each node has at most 2 children

```java
public class Node<T> {
    T content;
    Node<T> left, right; // NULL indicates no child
    public class Node() { // initialize content
        left = null;
        right = null;
    }
    get/set methods for all relevant fields
}
```
public class BinaryTree<T> {
    Node<T> root;

    Pre-order
    Post-order traversal

    E.g. Pre-order (Node node) {
        process(node);  // be sure to handle null
        Pre-order(left);
        Pre-order(right);
    }

    In-order traversal:
    (2 * 5 + 7) * 11 => Expression Tree

    Computing value: traversal of tree.

    Compute children; then do operation (post-order)
    Construct tree: left subtree; parent; right subtree (in-order)
In-order traversal:

- inorder (left) 0(m)
- process (content)
- inorder (right) only looking in 1 subtree

Binary Search Trees (BST)

Goal: data structure that allows fast searches

Type of content at the node needs to be comparable (to allow ordering)

```
      5
     / \
    3   7
   / \  > 5
  1   4
```

At any node, left subtree content is smaller:
- right subtree content is larger.

Binary search: content to find
- compare the at node; if equal return true
- else either left or right but not both!
E.g.

Degenerate case:
- it is a binary search tree
- but imbalanced

One side has more nodes
- larger height of nodes

Balanced binary search trees (different conditions)
E.g. \( |\text{height (left)} - \text{height (right)}| \leq 1 \)

1. \( |\text{nodes (left)} - \text{nodes (right)}| \leq 1 \) = more stringent

Under balance: searching in \( O(\log_2 n) \)

Insert / remove: make sure the BST condition is not violated
  balance condition is not