CMPT 225: Data Structures & Programming – Unit 13 – General Trees

Dr. Jack Thomas
Simon Fraser University

Spring 2021

Today’s Topics

- The Tree ADT
- Preview of Tree Variants
- Trees in Java
- Implementing a Tree
- Analyzing our Tree
- Traversing a Tree

Tree: The ADT

- A data structure storing a non-linear set of data elements.
- These elements are organized into a hierarchy.
- Methods of a Tree include:
  - **Element**: Returns the object stored in a given node.
  - **Root**: Returns the root of a Tree.
  - **Parent**: Returns the parent of a given node.
  - **Children**: Returns a collection of the nodes that are children of a given node.
  - **isInternal**: Tests whether a node is internal.
  - **isExternal**: Tests whether a node is external (a leaf).
  - **isRoot**: Tests whether a node is the root.

Tree: The ADT

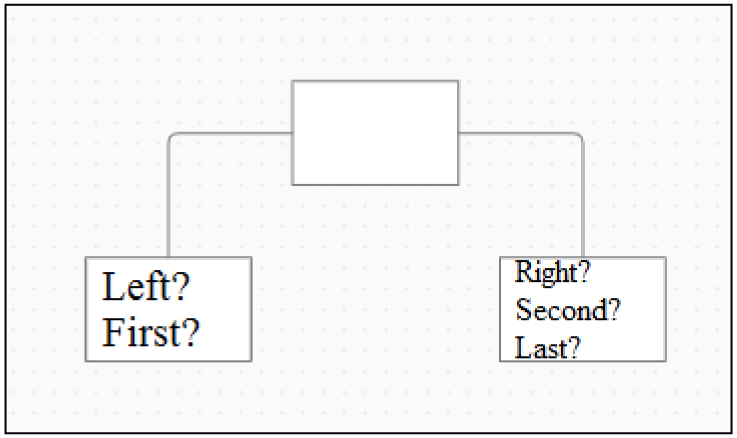
- There are also a number of generic methods we have seen or will see again:
  - **Size**: Returns the number of nodes of the tree.
  - **isEmpty**: Tests whether the tree has nodes or not.
  - **Iterator**: Returns an iterator (a type of collection) of all of the elements stored in the nodes of the tree.
  - **Positions**: Returns a collection of all nodes of the tree.
  - **Replace**: Swaps the element stored in a given node with a given element.

How do we Add? Remove? Search? Sort?

- The ADT for the most general sort of Tree is actually so general that it doesn’t provide methods for a lot of core functions
- Instead, the many variant trees each define these methods according to their own constraints and structures.
- Like with roots, there are many different constraints or properties that can be applied to a tree to give it some special structure.

Ordered Trees

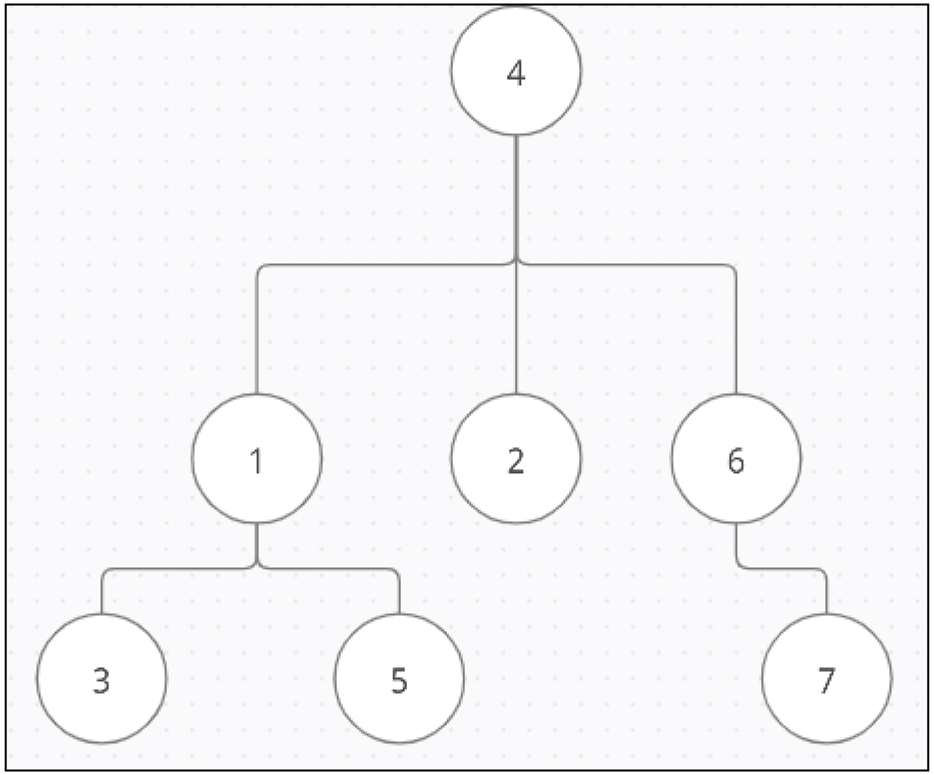
- When we draw a tree, we often reflexively read each node's children from left to right, but this isn't fundamental to a tree.



- If the children do have some relative position to each other, we call this an ordered tree – like how a list node's next and previous links have a direction, a tree node's children are also ordered from first to last.

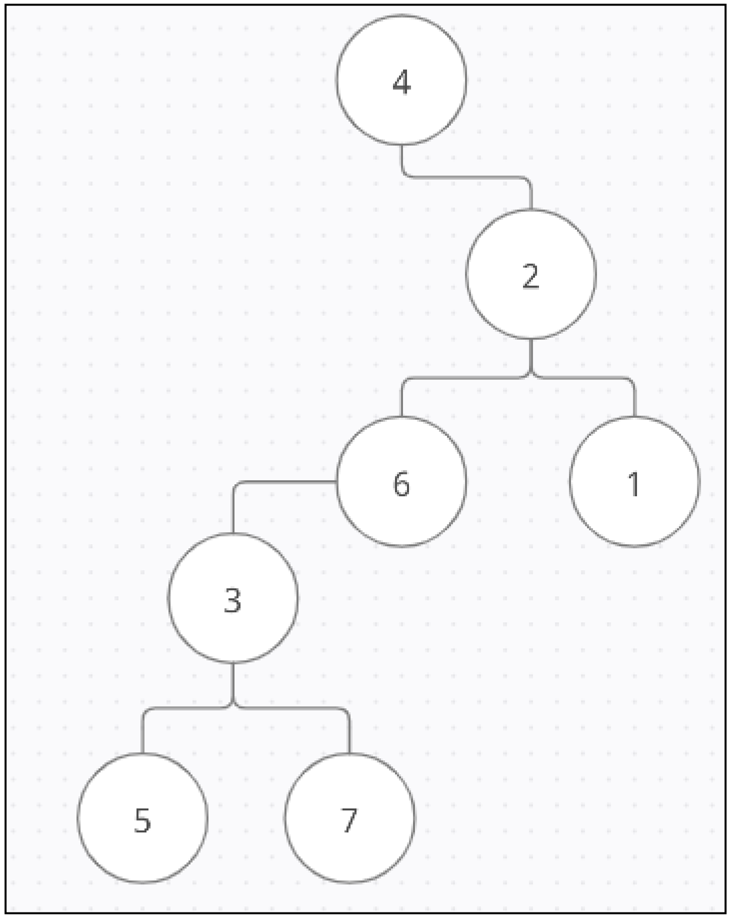
Balanced Trees

- In a balanced tree, the difference between the height of any node’s subtrees is within one.
- A perfectly balanced tree has all leaves with the same depth.



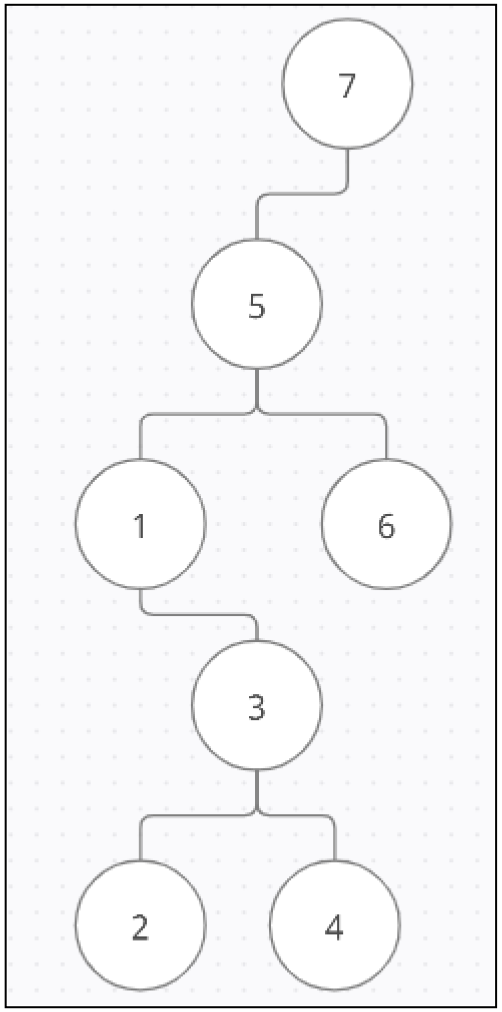
Binary Tree

- A **Binary Tree** restricts each node of a tree to at most two children.
- (We'll have more on this one later)



Binary Search Tree

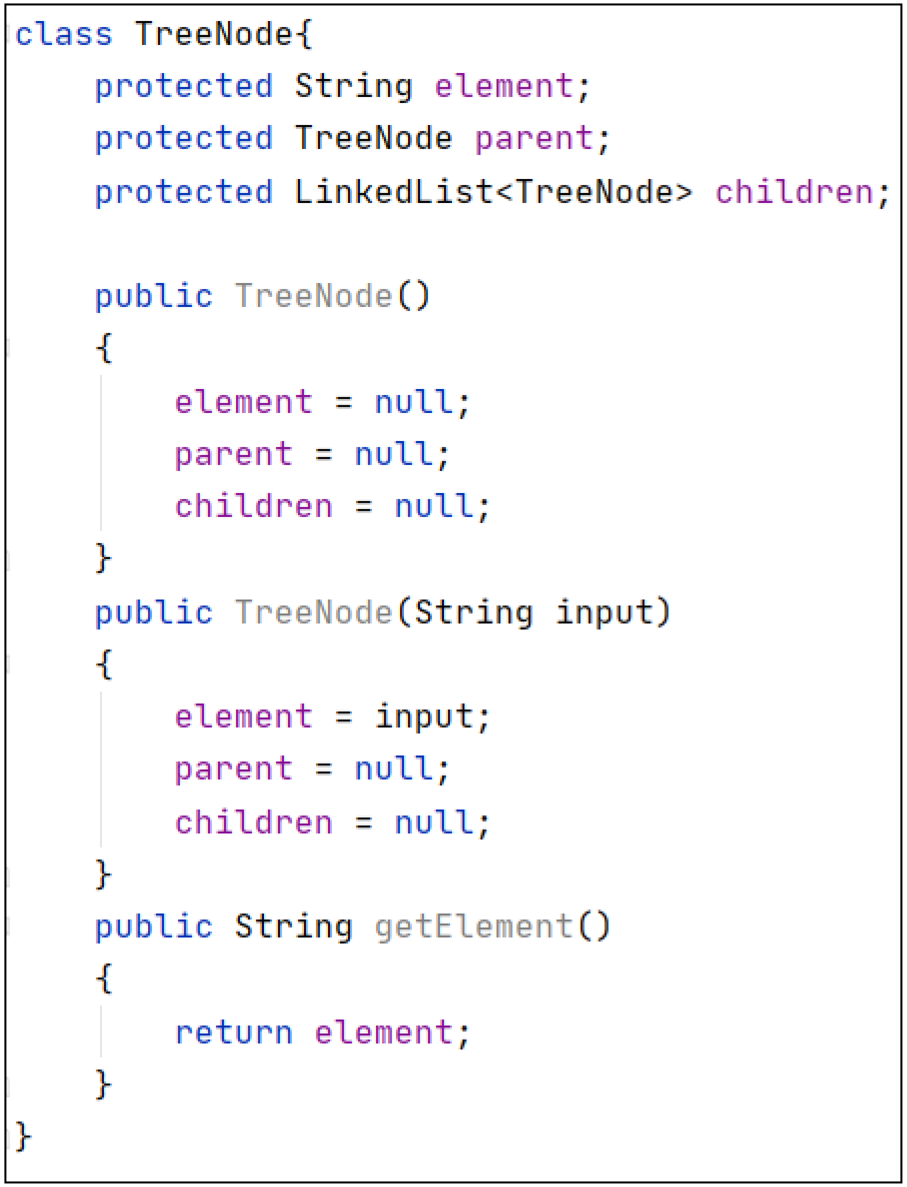
- A **Binary Search Tree** also orders the children such that, for every node in the tree, the value of the element in their "left" child (and all of that child's descendants) is less than that node's element, while the value of the element in their "right" child (and all of that child's descendants) is greater.
- (More on this one later too)



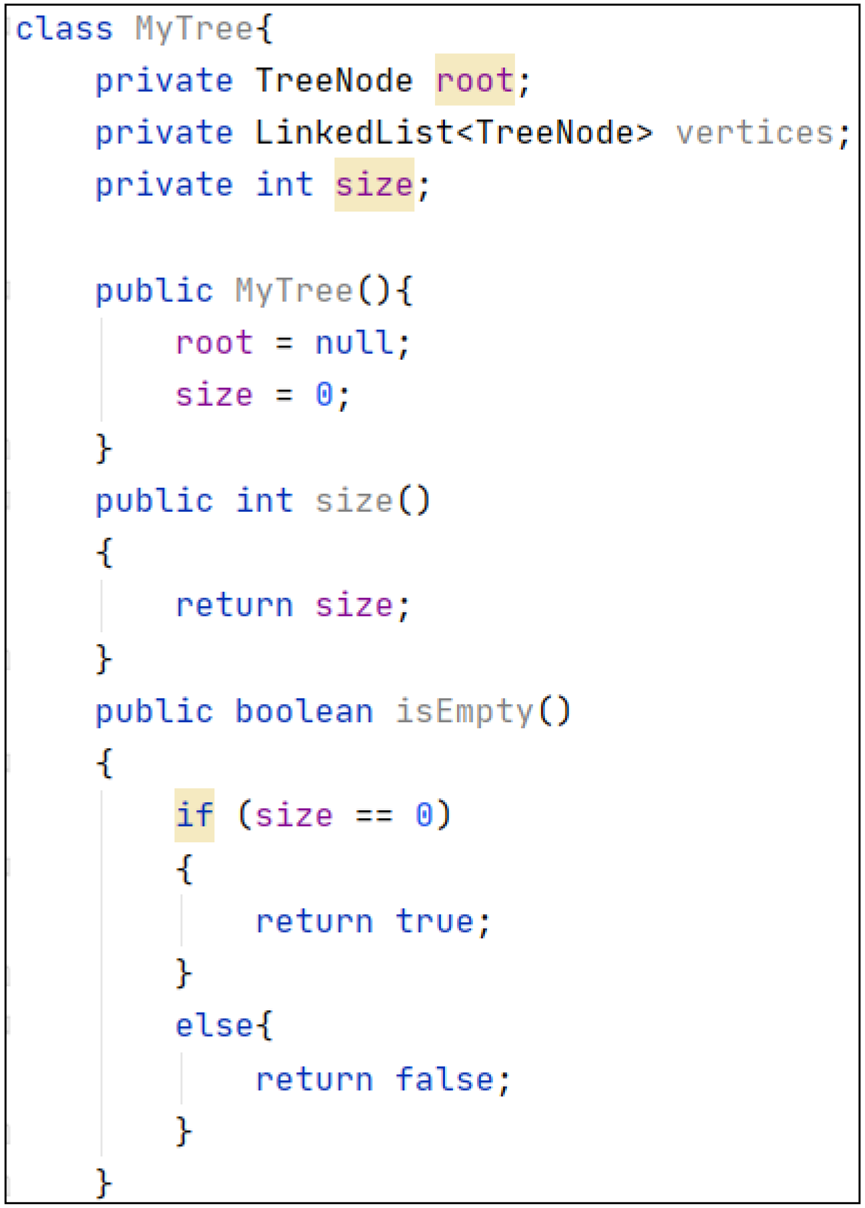
The Tree in Java

- The purely general Tree doesn’t just **not have a standard Java class**, it **doesn’t even have an interface**.
- There are **Tree classes in Java**, or interfaces relevant to Trees, but they are based on **specific variants** that we’ll discuss later.
- That doesn’t mean we can’t make our own!

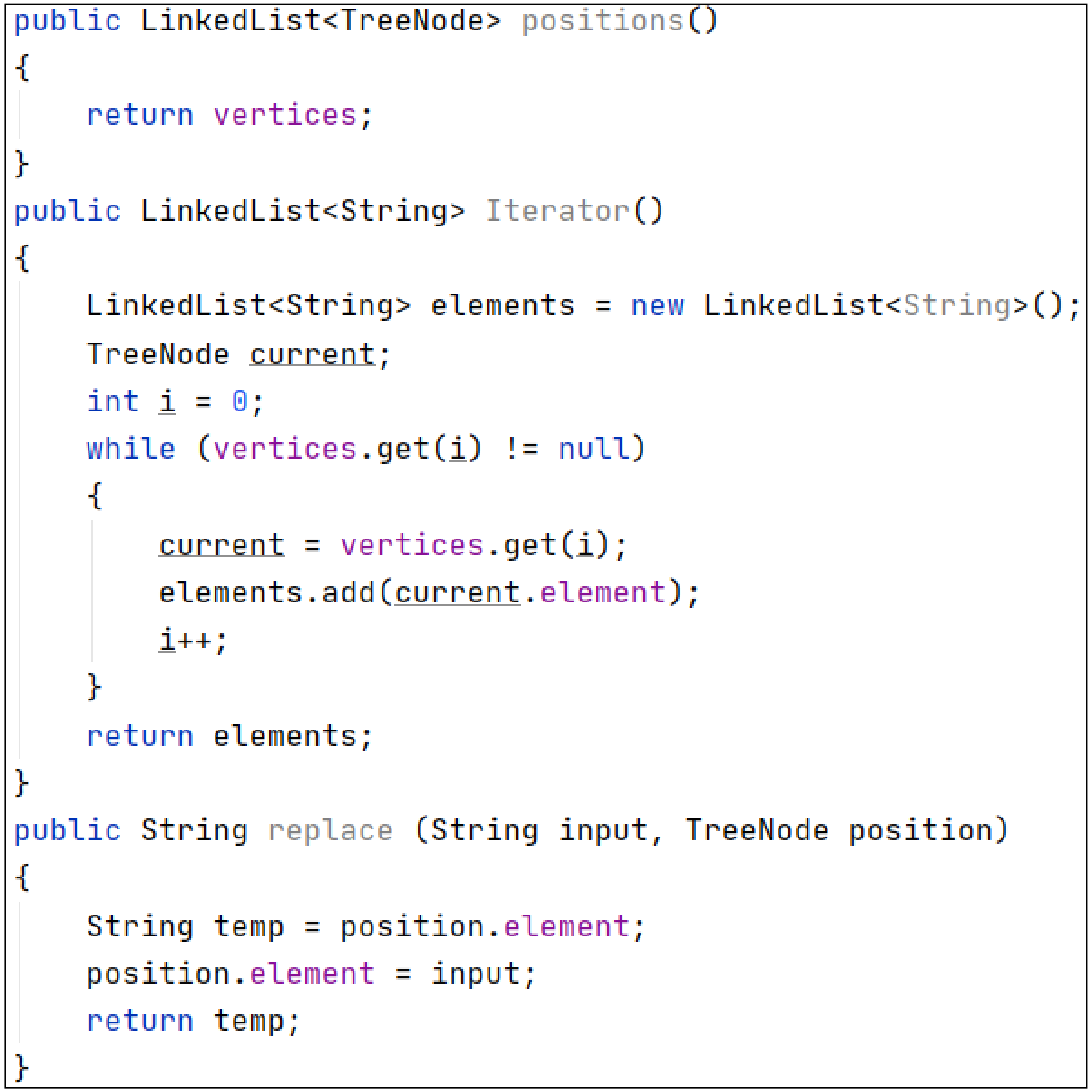
Implementing a Tree



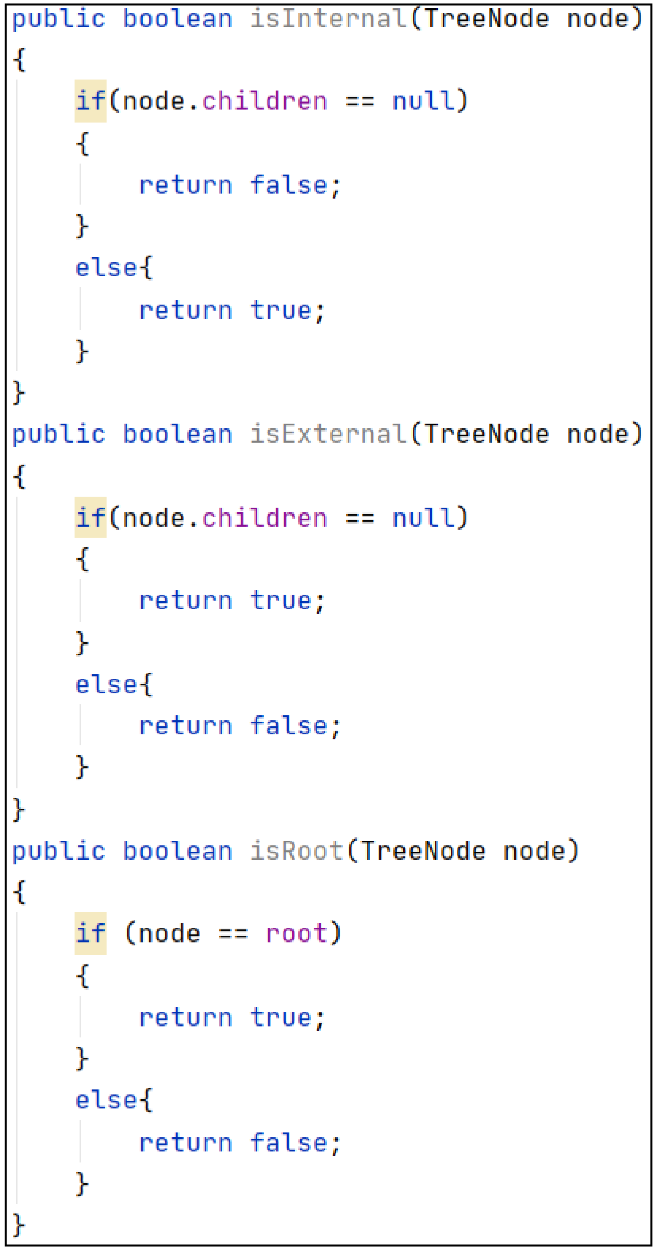
```java
class TreeNode{
    protected String element;
    protected TreeNode parent;
    protected LinkedList<TreeNode> children;

    public TreeNode()
    {
        element = null;
        parent = null;
        children = null;
    }
    public TreeNode(String input)
    {
        element = input;
        parent = null;
        children = null;
    }
    public String getElement()
    {
        return element;
    }
}
```



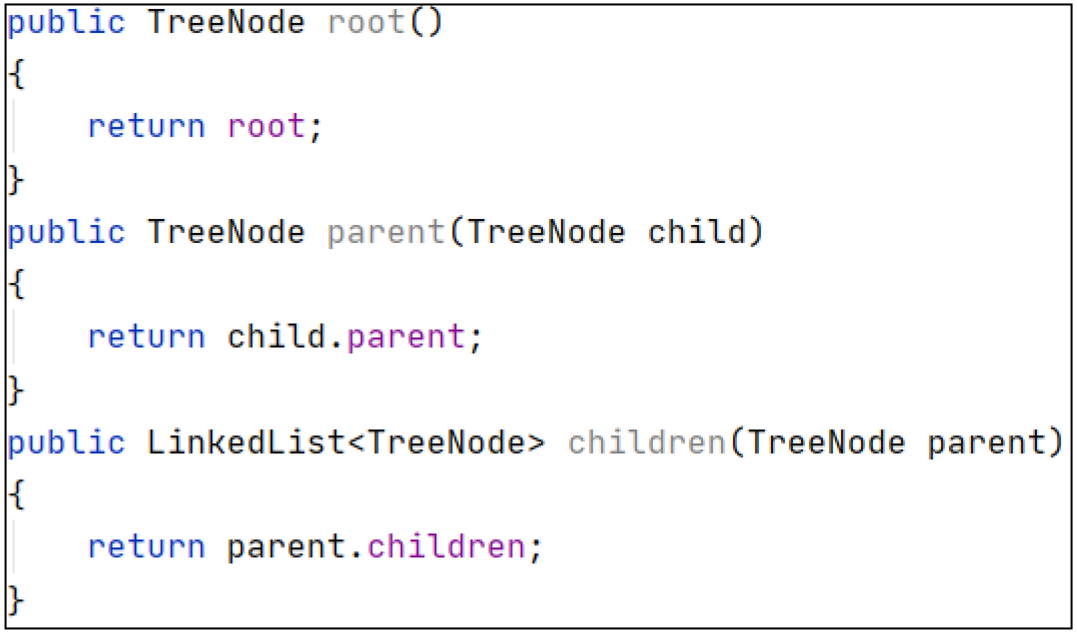
```java
class MyTree{
    private TreeNode root;
    private LinkedList<TreeNode> vertices;
    private int size;

    public MyTree(){
        root = null;
        size = 0;
    }
    public int size()
    {
        return size;
    }
    public boolean isEmpty()
    {
        if (size == 0)
        {
            return true;
        }
        else{
            return false;
        }
    }
```



```java
public LinkedList<TreeNode> positions()
{
    return vertices;
}
public LinkedList<String> Iterator()
{
    LinkedList<String> elements = new LinkedList<String>();
    TreeNode current;
    int i = 0;
    while (vertices.get(i) != null)
    {
        current = vertices.get(i);
        elements.add(current.element);
        i++;
    }
    return elements;
}
public String replace (String input, TreeNode position)
{
    String temp = position.element;
    position.element = input;
    return temp;
}
```



```java
public boolean isInternal(TreeNode node)
{
    if(node.children == null)
    {
        return false;
    }
    else{
        return true;
    }
}
public boolean isExternal(TreeNode node)
{
    if(node.children == null)
    {
        return true;
    }
    else{
        return false;
    }
}
public boolean isRoot(TreeNode node)
{
    if (node == root)
    {
        return true;
    }
    else{
        return false;
    }
}
```



```java
public TreeNode root()
{
    return root;
}
public TreeNode parent(TreeNode child)
{
    return child.parent;
}
public LinkedList<TreeNode> children(TreeNode parent)
{
    return parent.children;
}
```

Analyzing the Tree

- Asymptotic analysis for the Tree's methods:
  - **Size, isEmpty**: O(1)
  - **Positions, iterator**: O(n)
  - **Replace**: O(1)
  - **Root, parent**: O(1)
  - **Children**: O(# of children of the given node)
    - A special case, since this isn't directly connected to n.
  - **isInternal, isExternal, isRoot**: O(1)
- The **run-time analysis** for methods that add, remove, sort, and search a tree will **vary greatly based on the variant** of tree we're dealing with.

Traversing a Tree

- Adding, removing, sorting, and retrieving all typically involve having to **traverse** a tree, navigating from the root (like the head in a list) to wherever you need to go.
- For example, say we want to **find the depth of a node**. We could use a simple recursive algorithm that returns zero if the node it’s considering is the root, or recursively call itself on the node’s parent and add one if it isn’t.

Depth()

```
Algorithm depth(T, p):
    Input: A tree T and a node (position) p.
    Output: The depth of the node p
    if p.isRoot()
        return 0
    else
        return 1 + depth(T, p.parent())
```

- While the worst-case run time could be O(n), the depth of the tree doesn’t always grow proportionately with the number n of inputs.

Height()

- Finding the height of a node depends on finding the depth of its deepest descendent, which can be found with the help of depth().

```
Algorithm height(T):
    Input: A tree T.
    Output: The height of the tree T
    h = 0
    for each p in T.positions() do
        if p.isExternal() then
        h = max(h, depth T,p))
    return h
```

Problems with Height()

- Unfortunately, that algorithm requires us to visit every node in the tree to find all of the external ones, and then calls depth() (an O(n) function) on each of them to find the deepest.
- This sounds like an O(n^2) operation, and doesn’t take advantage of any of the features of the tree’s hierarchy.

Height2(), Leveraging Recursion

- If every node can be the root of a subtree, and the height of every tree is the height of their highest child's subtree + 1, then…

```
Algorithm height2(T, p):
        Input: A tree T and a node p.
        Output: The height of the tree T
        if p is external then
                return 0
        else
                h = 0
                for each q in p.children() do
                        h = max(h, height2(T, q))
                return 1 + h
```

Height2() and the Benefits of Smart Traversal

- Starting from root, **this function makes a recursive call on each child**, each of which will either return in constant time (if external) or make a recursive call on each of their children.
- The algorithm **"visits" each node once**, starting from root and working its way down. Its runtime is in the realm of 2n, or **O(n)**.
- This is a **preorder traversal**, which works down through a node's children to the leaves. A **postorder traversal** works up through the parents to the root.

Recap – Come Up With Your Own "Lumberjack" Joke If You Want

- The general **Tree** has an **ADT** of standard methods and properties that apply to the many variants of trees.
- Some **variant properties** include whether the tree is **ordered**, **balanced**, **binary**, or a full **binary search tree**.
- Standard Java **doesn't include built-in classes or interfaces for Trees**, but we can make our own easily enough.
- **Analysis** of a Tree's performance will **depend a lot on our chosen variant**.
- **Traversing** a Tree is where a lot of the value is, and will motivate us learning more about types of Trees.