Trees

a non-introduction to horticulture

Reading LDC Ch 16
A node can have only one parent, but may have multiple children
- Children of the same parent are siblings
- The root is the only node which has no parent
- A node that has no children is a leaf node
- A node that is not the root and has at least one child is an internal node
  - A node is an ancestor of another node if it is above it on the path from the root.
  - Nodes that can be reached by following a path from a particular node are the descendants of that node

A tree is a non-linear hierarchical structure
- Tree is comprised of a set of nodes in which elements are stored and edges connect one node to another
Tree Terminology

- A **subtree** is a tree structure that makes up part of another tree.
- We can follow a **path** through a tree from parent to child, starting at the root.
- The **path length** is determined by counting the number of edges that must be followed to get from the root to the node.
- The **level** of a node is the length of the path from the root to the node.
- The **height** of a tree is the length of the *longest path* from the root to a leaf.
We classify trees by the maximum number of children any node in the tree may have.

**General trees** have no limit to the number of children a node may have.

**n-ary trees** limits each node to no more than $n$ children.

**Binary trees** are those in which nodes may have at most two children.

---

**Tree Classifications**

- We classify trees by the maximum number of children any node in the tree may have.
- **General trees** have no limit to the number of children a node may have.
- **n-ary trees** limits each node to no more than $n$ children.
- **Binary trees** are those in which nodes may have at most two children.
Tree Classifications

- We classify trees by the maximum number of children any node in the tree may have.
- General trees have no limit to the number of children a node may have.
- $n$-ary trees limits each node to no more than $n$ children.
- Binary trees are those in which nodes may have at most two children.

- A tree is **balanced** if all of the leaves of the tree are on the same level or at least within one level of each other.
- A balanced binary tree with $n$ nodes has a height of $O(\log_2 n)$.
- A balanced $n$-ary tree with $m$ nodes will have a height of $O(\log_n m)$. 
Balanced, Complete and Full

- A tree is **balanced** if all of the leaves of the tree are on the same level or at least within one level of each other.

- An n-ary tree is **full** if all leaves of the tree are at the same height and every non-leaf node has exactly n children.

- A tree is **complete** if it is full, or full to the next-to-last level with all leaves at the bottom level on the left side of the tree.
There’s more than one?
A traversal simply dictates the order in which the elements of a collection are “visited”

Traversing a tree (visiting all nodes in some sequence) is more interesting than traversing a linear structure

What types of traversals could you imagine?
DFS-style Tree Traversals

- Nodes are visited **before** any subtrees are visited

- Visit the root in **between** the traversals of the left and right subtrees

- Visit the root node **after** the traversals of the left and right subtrees

**Preorder**

- Visit Node
- Traverse (left)
- Traverse (right)

**Inorder**

- Traverse (left)
- Visit Node
- Traverse (right)

**Postorder**

- Traverse (left)
- Traverse (right)
- Visit Node
BFS-style Tree Traversal

- Visit the nodes on each level, left to right, top to bottom starting at the root

Enqueue the root node of the tree

While the queue is not empty{
    Dequeue node
    Visit node
    Enqueue left child of node
    Enqueue right child of node
}
Using an Array to store a Tree

- Can you do it? How?
Tree Implementations
Strategies for Implementing Trees

- There are methods that use arrays or links
- Array-based implementations are the less obvious choice, but sometimes useful
  - Computed Links in Array
  - Stored Links in an Array
- Link-based implementations are more powerful and efficient, but also more complicated
- You should know about all options!
Computed Links in an Array

- Place tree nodes in specific indices of the array
- A node’s index can be used to calculate the indices of its parent and children

D is in location 3.
Where can you find D’s children?
How about D’s parent?
Stored Links in an Array

- Array positions are allocated on a first-come, first-served basis.
- Each element of the array is an object that stores a **reference** to the tree element and the array index of each child.

D is in location 4.

Where can you find D’s children?
How about D’s parent?
Tree Implementation with Linked Nodes

- Each tree node can be defined using a separate class – similar to `LinearNode` or linked lists.
- Nodes contain a reference to the data stored in the node, and references for each of the possible children of the node.
  - **Binary** tree: 2 references required - left and right children.
  - **n-ary** tree: n references required - one for each possible child.
- Trees organized this way lend themselves to recursive processing for many operations.
package javafoundations;

import java.util.Iterator;
import javafoundations.exceptions.EmptyCollectionException;

public interface BinaryTree<T> extends Iterable<T> {
    /**
     * Returns the element stored in the root of the tree.
     * @return The element at the root of the tree.
     */
    public T getRootElement() throws EmptyCollectionException;

    /**
     * Searches for a specific element in the binary tree.
     * @param target The target element we are looking for.
     * @return True if the binary tree contains an element that matches the specified element.
     */
    public boolean contains(T target) throws EmptyCollectionException;
}
/**
 * Indicates whether the tree has nodes or not.
 * @return True if the binary tree contains no elements, and false otherwise.
 */
public boolean isEmpty();

/**
 * Counts the number of nodes in the binary tree.
 * @return The number of elements in this binary tree.
 */
public int size();

/**
 * Override the toString method to print out the elements in the binary tree.
 * @return The string representation of the binary tree.
 */
public String toString();
}
A Binary Tree Implementation

- A possible set of operations for a binary tree is shown in the BinaryTree interface
- BinaryTree has no methods to add a particular element, or to remove a particular element from the tree
- Refined versions of binary tree (such as binary search trees) will define those methods based on specific characteristics
- BinaryTree is still useful in certain situations
javafoundations.BTNode

```java
package javafoundations;

public class BTNode<T> {
    protected T element;
    protected BTNode<T> left, right;

    public BTNode(T elmt) {
        element = elmt;
        left = right = null;
    }
}
```

(more...)
javafoundations.BTreeNode

// Returns the element stored in this node.
public T getElement()
    { return element; }

// Sets the element stored in this node.
public void setElement (T element)
    { this.element = element; }

// Returns the left subtree of this node.
public BTreeNode<T> getLeft()
    { return left; }

// Sets the left child of this node.
public void setLeft (BTreeNode<T> left)
    { this.left = left; }

// Returns the right subtree of this node.
public BTreeNode<T> getRight()
    { return right; }

// Sets the right child of this node.
public void setRight (BTreeNode<T> right)
    { this.right = right; }
What’s an Iterator?
9.3 – The Iterator Interface

- An iterator is an object that provides a means of processing a collection of objects, one at a time

By implementing the Iterator interface, a class formally establishes that: objects of that type are iterators

Now, the for-each version of the for loop can be used to process the items in the iterator
Implementing an iterator using array

```java
import java.util.*;

public class ArrayIterator<T> implements Iterator<T> {

    private int DEFAULT_CAPACITY = 10;
    private int count;  // the number of elements in the iterator
    private int current;  // the current position in the iteration
    private T[] items;    // the iterator's storage for elements

    // Sets up this iterator.
    public ArrayIterator() {
        items = (T[]) (new Object[DEFAULT_CAPACITY]);
        count = 0;
        current = 0;
    }

    // (more...)
```
Implementing an iterator using array

```java
public void add (T item) {
    if (count == items.length)
        expandCapacity();
    items[count] = item;
    count++;
}

private void expandCapacity() {
    T[] larger = (T []) (new Object[items.length*2]);
    int location = 0;
    for (T element : items)
        larger[location++] = element;
    items = larger;
}
```

(more...)

Implementing an iterator using array

```java
// Returns true if this iterator has at least one more element to deliver in the iteration.

public boolean hasNext() {
    return (current < count);
}

// Returns the next element in the iteration. If there are no more elements in this iteration, a NoSuchElementException is thrown.

public T next() {
    if (!hasNext())
        throw new NoSuchElementException();
    current++;
    return items[current - 1];
}

// The remove operation is not supported in this collection.

public void remove() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
}
```
We can also implement it using a LinkedList
Take a closer look at the tree implementation
private boolean recursiveFind(BTNode<T> startNode, T target) {
    if (startNode.getElement().equals(target))
        return true;

    boolean targetFound = false;
    if (startNode.getLeft() != null)
        targetFound = recursiveFind(startNode.getLeft(), target);
    if (!targetFound && startNode.getRight() != null)
        targetFound = recursiveFind(startNode.getRight(), target);

    return targetFound;
}
private int recursiveCount(BTNode<T> startNode) {
    int size = 1;
    if (startNode.getLeft() != null)
        size += recursiveCount(startNode.getLeft());
    if (startNode.getRight() != null)
        size += recursiveCount(startNode.getRight());
    return size;
}