Trees and Traversals

Weiss, from the beginning of chapter 19 through 19.3 (pages 687–706)
Trees

- Trees establish hierarchical (child-parent) relationship between values.
- For example, a reporting structure within a company has a hierarchy:

```
CEO
   /       \
  /         \
Vice President (Finance)  Vice President (Engineering)
                     /       \                     /       \
                  /         \                  /         \
                Manager  Manager  Manager  Manager
                       /           \           /           \
                      /             \         /             \
                     Programmer  Programmer Designer  Designer
```
Trees

• A tree has recursive definition. A tree is either:
  ▪ Empty (a node with no value) or
  ▪ A node with a value, a left sub-tree, and a right sub-tree

• This tree is called a *binary* tree. Each node has at most two children. Sub-trees are recursive occurrence of the tree definition.

• The top most element of tree is called *root*.

• Any two nodes, which are connected by an *edge*, have the parent-child relationship.

  For example, 9 is the parent of 7. 7 is a child of 9.

• The bottom most elements are *leaves*. They do not have children.
Trees

- A unique path traverses from the root to each node. The number of edges on this path is called path length.
- For a path from node u to node v, u is the ancestor of v and v is the descendent of u.
- Depth of a node is the length of a path from root to the node. For example, the depth of node 6 is 3.
- Height of a node is the length of a path from node to the deepest leaf. For example, the height of node 9 is 2.
- Height of the root node is known as the height of tree (3).
class BinaryNode<AnyType> {
    private AnyType element;
    private BinaryNode<AnyType> left;
    private BinaryNode<AnyType> right;
    public BinaryNode( ) { this( null, null, null ); }
    public BinaryNode ( AnyType element Node<AnyType> left, Node<AnyType> right ) {
        this.element = element;
        this.left = left;
        this.right = right;
    }
}

public class BinaryTree<AnyType> {
    private BinaryNode<AnyType> root;
    public BinaryTree( ) { root = null; } // empty tree
    public BinaryTree( AnyType rootElement ) { // one-node tree
        root = new BinaryNode<AnyType>( rootElement, null, null );
    }
}
Recursion in Tree

• Size of a node is one plus the number of descendents of the node.
  ▪ Recursive definition: one plus the sizes of the left and right subtrees
• Size of a tree is the size of tree root.

• The base case occurs when tree is empty. Size is zero.

public static <AnyType> int sizeHelper( BinaryNode<AnyType> t ) {
    if ( t == null ) {
        return 0;
    }
    return 1 + sizeHelper( t.left ) + sizeHelper( t.right );
}

• sizeHelper is a helper method. It is called within another method another method (e.g., size).

    public int size() {return BinaryNode.sizeHelper(root); }
Binary Search Tree

- Sequential search takes $O(N)$ time. You can sort elements and then use the binary search algorithm that takes $O(\log N)$ time.
- Sorting is an overhead for searching.

- Binary search tree is a tree that defines a kind of sortedness of elements. A binary search tree is either
  - An empty tree
  - A node with a value and left and right sub-trees. All values in the *left* sub-tree are *less* than the node value and all values in the *right* sub-tree are *greater* than the node value.
Binary Search Tree

- When *inserting* a node into a binary search tree:
  - If the sub-tree is empty, replace it with the node. The new sub-tree does not have left and right sub-trees.

```java
private static BinaryNode<AnyType> insertH(AnyType e, BinaryNode<AnyType> t) {
    if(t == null) { return new BinaryNode<AnyType>(AnyType, null, null); }
    else if(e.compareTo(t.element) < 0) {
        t.left = insertH(e, t.left);
    } else if (e.compareTo(t.element) > 0) {
        t.right = insertH(e, t.right);
    } else { throw new DuplicateItemException( e.toString( ));}
    return t; // returns the updated root of a sub-tree
}

public void insert(AnyType e) { root = insertH(e, root); }
```
Binary Search Tree

• When *removing* a node from a binary search tree:
  ▪ If the node is a *leaf*, just remove the node and replace its parent’s child link to null.
  ▪ If the node has *one child*, adjust its parent’s child link to bypass the deleted node.

```
3
 /   \
2   5
    /
   6
```

```
3
 /   \
2   5
    /
   6
```

```
remove 2
```

```
3
 /   \
5   6
```

```
3
 /   \
5   6
```

```
3
 /   \
2   6
```

```
3
 /   \
2   6
```

```
remove 5
```

```
3
 /   \
2   5
    /
   6
```

```
3
 /   \
2   5
    /
   6
```
Binary Search Tree

- When *removing* a node $v$ from a binary search tree:
  - If the node has *two children*, 1) replace the value in the node $v$ with the smallest value in the right sub-tree of node $v$ and 2) remove the node with the smallest value.

```
    1
   / \
  8   15
 /     \
13     14
```

```
    1
   / \
  8   15
   /   \
13     14
```

```
    1
   / \
  8   15
```

Binary Search Tree

• Order property
  – Elements in the left sub-tree are < the value and elements in the right sub-tree are > the value

• Access an element by following node links according to the order property

• Find minimum and maximum elements

![Diagram of a binary search tree before and after removing 5]
Tree Traversal

- Consider a toString method that displays the values of tree nodes.
- The *order* of traversing (visiting) nodes is important. Here is an ordering for traversing the nodes:
  - If tree is empty, display an empty string
  - If tree is not empty, display the value of this node. Then, recursively, display values of the left sub-trees and right sub-tree, *in-order*.

Output: [5, 2, 1, 3, 8, 7, 6, 9, 10]
- This traversal is called the **pre-order** traversal.
Tree Traversal

• There are two more traversals:
  
  - **In-order** traversal: recursively, display values of the left sub-tree. Then, display the value of this node. After that, display the values of the right sub-tree.
    Output: [1, 2, 3, 5, 6, 7, 8, 9, 10]
    ➢ The elements are displayed in the *sorted* order.

  - **Post-order** traversal: recursively, display values of the left sub-tree and right sub-tree. Then, display the value of this node.
    Output: [1, 3, 2, 6, 7, 10, 9, 8, 5]
Time Complexity

- For each of the tree traversals, every node is visited. The traversals take $O(N)$ time.
- The running time of *search* and *insert* operations depends on path length. A binary search tree looks like
  - A linked list. At the worst case, the running time is $O(N)$.
  - A *perfect* tree. The running time is the length of a path from root to leaf. It is computed based on the number of nodes $N$.
  - $N$ is computed based on the number of nodes at each *level* of tree.
Time Complexity

• The number of nodes at level $i$ of a perfect binary search tree is $2^i$.

Proof by induction:

• $i = 0$ (first level): $2^0 = 1$

• $i = k + 1$: At level $k$, there are $2^k$ nodes. Each node has two children. Thus, the number of nodes at level $k + 1$ is $2^k \cdot 2 = 2^{k+1}$.

For a tree with height $h$, the total number of nodes is:

$$N = \sum_{i=0}^{h} 2^i = 2^0 + 2^1 + \cdots + 2^h$$

$$N = 2^{h+1} - 1$$

$$N = 2^{h+1} - 1 \rightarrow N + 1 = 2^{h+1} \rightarrow \log_2 N + 1 = \log_2 2^{h+1} \rightarrow \log_2 N + 1 = h + 1 \rightarrow \log_2 N + 1 - 1 = h$$

The height ($h$) is $O(\log N)$. 