"A tree may grow a thousand feet tall, but its leaves will return to its roots."

- Chinese Proverb

---

**Binary Search Trees**

- A binary tree is a tree where each node has at most two children, referred to as the left and right child.
- A binary search tree is a binary tree where every node's left subtree has values less than the node's value, and every right subtree has values greater. A new node is added as a leaf.

![Binary Search Tree Diagram]

- root
- parent
- left child 11
- right child 19

**Performance of Binary Trees**

- For the three core operations (add, access, remove) a binary search tree (BST) has an average case performance of $O(\log N)$.
- Even when using the naïve insertion / removal algorithms.
- no checks to maintain balance.
- balance achieved based on the randomness of the data inserted.
Sample Insertion

- 100, 164, 130, 189, 244, 42, 141, 231, 20, 153
  (from HotBits: www.fourmilab.ch/hotbits/)

If you insert 1000 random numbers into a BST using the naïve algorithm what is the expected height of the tree? (Number of links from root to deepest leaf.)

Properties of a BST

- The minimum value is in the left most node
- The maximum value is in the right most node
  - useful when removing an element from the BST
- An **inorder traversal** of a BST provides the elements of the BST in ascending order
  - There are 4 traditional types of traversals
    - preorder traversal: process the root, then process all sub trees (left to right)
    - in order traversal: process the left sub tree, process the root, process the right sub tree
    - post order traversal: process the left sub tree, process the right sub tree, then process the root
    - level order traversal: starting from the root of a tree, process all nodes at the same depth from left to right, then proceed to the nodes at the next depth.

Worst Case Performance

- In the worst case a BST can degenerate into a singly linked list. Performance goes to $O(N)$
  - 2 3 5 7 11 13 17

Results of Traversals

- To determine the results of a traversal on a given tree draw a path around the tree.
  - start on the left side of the root and trace around the tree. The path should stay close to the tree.

<table>
<thead>
<tr>
<th>Pre order: process when pass down left side of node</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 42 20 72 164</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In order: process when pass underneath node</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 42 72 100 164</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post order: process when pass up right side of node</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 72 42 164 100</td>
</tr>
</tbody>
</table>
Implementing BSTs

- There is not a standalone binary search tree class in the Java standard library
- The TreeMap and TreeSet classes use BSTs as the internal storage container (also called the backing collection)
- To formulate questions regarding binary trees and binary search trees APCS development committee has provided a TreeNode class – there has been a binary tree question on every test since 1999. Not always BST

The TreeNode class

```java
public class TreeNode {
    private Object value;
    private TreeNode left;
    private TreeNode right;

    public TreeNode(Object initValue) {
        value = initValue; left = null; right = null;
    }

    public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight) {
        value = initValue; left = initLeft; right = initRight;
    }

    public Object getValue() { return value; }
    public TreeNode getLeft() { return left; }
    public TreeNode getRight() { return right; }

    public void setValue(Object theNewValue) { value = theNewValue; }
    public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }
    public void setRight(TreeNode theNewRight) { right = theNewRight; }
}
```

More on Implementation

- Many ways to implement BSTs
- Using nodes is just one and even then many options and choices
- APCS usually implements an empty tree with the root equal to null, as opposed to a dummy node

```java
public class BinarySearchTree {
    private TreeNode root;
    private int size;

    public BinarySearchTree() {
        root = null;
        size = 0;
    }

    public boolean contains(Comparable data) {
        return conHelp(data, root);
    }

    public boolean conHelp(Comparable data, TreeNode t) {
        return false;
    }

    public boolean conHelp(Comparable data, TreeNode t) {
        boolean result;
        if( t == null ) // fell off the tree. Item not present
            result = false;
        else {
            Comparable dataInNode = (Comparable)(t.getValue());
            int dir = dataInNode.compareTo( data );
            if( dir > 0 ) // data in node > data to insert. go left
                return conHelp(data, t.getLeft());
            else if( dir < 0 ) // data in node < data to insert. go right
                return conHelp(data, t.getRight());
            else // found it!
                result = true;
        }
        return result;
    }
}
```

Access an Element

```java
public boolean contains(Comparable data) {
    return conHelp(data, root);
}

public boolean conHelp(Comparable data, TreeNode t) {
    boolean result;
    if( t == null ) // fell off the tree. Item not present
        result = false;
    else {
        Comparable dataInNode = (Comparable)(t.getValue());
        int dir = dataInNode.compareTo( data );
        if( dir > 0 ) // data in node > data to insert. go left
            return conHelp(data, t.getLeft());
        else if( dir < 0 ) // data in node < data to insert. go right
            return conHelp(data, t.getRight());
        else // found it!
            result = true;
    }
    return result;
}
```
Add an Element, Recursive

```java
public boolean add(Comparable data)
{
    int oldSize = size;
    root = addHelp(data, root);
    return oldSize != size;
}

public TreeNode addHelp(Comparable data, TreeNode t)
{
    if( t == null )
    {
        // fell off tree.
        // Create and return new node
        size++;
        return new TreeNode(data);
    }
    else
    {
        // which way do I go?
        Comparable dataInNode = (Comparable)(t.getValue());
        int dir = dataInNode.compareTo( data );
        if( dir > 0 )
            // data in node > data to insert. go left
            t.setLeft( addHelp(data, t.getLeft()) );
        else if( dir < 0 )
            // data in node < data to insert. go right
            t.setRight( addHelp(data, t.getRight()) );
        // else date in node equals data to insert. No action
        return t;
    }
}
```

Add an Element, Iterative

```java
public boolean otherAdd(Comparable data)
{
    int oldSize = size;
    // empty tree?
    if( root == null )
    {
        root = new TreeNode(data);
        size++;
    }
    else
    {
        int dir = -1;
        TreeNode lead = root;
        TreeNode trailer = root;
        while( lead != null && dir != 0 )
        {
            trailer = lead;
            dir = ((Comparable)(lead.getValue())).compareTo(data);
            if( dir > 0 )
                lead = trailer.getLeft();
            else if( dir < 0 )
                lead = trailer.getRight();
            if( lead == null )
            {
                if( dir > 0 )
                    trailer.setLeft( new TreeNode(data) );
                else
                    trailer.setRight( new TreeNode(data) );
                size++;
            }
        }
    return oldSize != size;
}
```

Remove an Element

- An exercise for you
- Three cases
  - node is a leaf, 0 children (easy)
  - node has 1 child (easy)
  - node has 2 children (interesting)

Maps - public interface

- A map is an object that maps keys to values
- A map cannot contain duplicate keys; each key can map to at most one value
- but those values could be complicated things like Lists, Sets, Stacks, Queues, or other Maps
  - an interesting way of representing a graph -> a map of sets
Methods

- `int size()`  
- `boolean isEmpty()`  
- `boolean containsKey(Object key)`  
- `boolean containsValue(Object value)`  
- `Object get(Object key)`  
- `Object put(Object key, Object value)`  
- `Object remove(Object key)`  
- `void putAll(Map t)`  
- `void clear()`  
- `Set keySet()`  
- `Collection values()`  
- `Set entrySet()`  
- `boolean equals(Object o)`  
- `int hashCode()`  

What to know for AP

- `HashMap`  
- `TreeMap`  

Methods:
- `put`  
- `get`  
- `remove`  
- `containsKey`  
- `size`  
- `keySet`

public Object
`remove`(Object key)

- If the map contains a mapping of this key to a value then the mapping is removed  
- Returns previous value associated with specified key, or null if there was no mapping for key

public boolean
`containsKey`(Object key)

- Returns true if this map contains a mapping for the specified key
public int size()

- Returns the number of key-value mappings in the Map

public Set keySet()

- Returns a set view of the keys contained in this map
- The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll retainAll, and clear operations.
- It does not support the add or addAll operations

Example

Map m = new TreeMap();
String[] words = {"blue", "red", "green", "yellow", "black"};
int j = 0;
for(int i = 1; i<=5; i++)
    {m.put(new Integer(i), words[j]);
     j++;}
System.out.println(m);
//lets run this then add to it

Example II

Create a Frequency Table of Characters in a String

public Map createFrequencyTable(String source)
{
    Map result = new TreeMap();
    Integer one = new Integer(1);
    Integer freq;
    Character temp;
    for(int i = 0; i < source.length(); i++)
    {
        temp = new Character( source.charAt(i));
        freq = (Integer)result.get( temp );
        if( freq == null )
            result.put( temp, one );
        else
            result.put( temp, new Integer( freq.intValue() + 1 ));
    }
    return result;
}
Sets - The Set Interface

- It is a collection that contains no duplicate elements
- It has at most one null element
- Implemented by
  - AbstractSet
  - HashSet
  - LinkedHashSet
  - TreeSet
- Extends the Collection interface

Methods

- int size()
- boolean isEmpty()
- boolean contains(Object x)
- Iterator iterator()
- Object[] toArray()
- boolean add(Object x)
- boolean remove(Object x)
- boolean containsAll(Collection c)
- boolean addAll(Collection c) // union
- boolean retainAll(Collection c) // intersection
- boolean removeAll(Collection c) // difference
- void clear()
- boolean equals(Object x)
- int hashCode()
**boolean contains (Object x)**

- Returns true if and only if the set contains the specified element

**boolean remove (Object x)**

- Removes the specified element from the set (if it exists)
- Returns true if the set contained the specified element (in other words, if the set changed as a result of the call)

**int size()**

- Returns the number of elements in the set

**Iterator iterator()**

- Returns an iterator over the elements in this set
- The elements of the set are in no particular order (unless this set is an instance of some class that provides a guarantee)