Topic 18
Binary Search Trees

"Yes. Shrubberies are my trade. I am a shrubber. My name is 'Roger the Shrubber'. I arrange, design, and sell shrubberies."

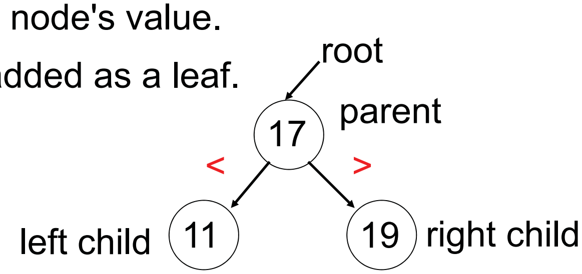
-Monty Python and The Holy Grail

The Problem with Linked Lists

- Accessing a item from a linked list takes O(N) time for an arbitrary element
- Binary trees can improve upon this and reduce access to O(log N) time for the average case
- Expands on the binary search technique and allows insertions and deletions
- Worst case degenerates to O(N) but this can be avoided by using balanced trees (AVL, Red-Black)

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 in which every node's left subtree holds values less than the node's value, and every right subtree holds values greater than the node's value.
- A new node is added as a leaf.



Attendance Question 1

- After adding N distinct elements in random order to a Binary Search Tree what is the expected height of the tree?

A. $O(N^{1/2})$

B. O(logN)

C. O(N)

D. O(NlogN)

E. $O(N^2)$

Implementation of Binary Node

```
public class BSTNode
{	private Comparable myData;
	private BSTNode myLeft;
	private BSTNode myRightC;

	public BinaryNode(Comparable item)
	{	myData = item;	}

	public Object getValue()
	{	return myData;	}

	public BinaryNode getLeft()
	{	return myLeft;	}

	public BinaryNode getRight()
	{	return myRight;	}

	public void setLeft(BSTNode b)
	{	myLeft = b; }
	// setRight not shown
}
```


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.)

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

More on Implementation

- Many ways to implement BSTs
- Using nodes is just one and even then many options and choices

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

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

Add an Element, Recursive

Add an Element, Iterative

Attendance Question 2

- What is the best case and worst case Big O to add N elements to a binary search tree?

	Best	Worst
A.	O(N)	O(N)
B.	O(NlogN)	O(NlogN)
C.	O(N)	O(NlogN)
D.	O(NlogN)	$O(N^2)$
E.	$O(N^2)$	$O(N^2)$

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

Remove an Element

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

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

Using Polymorphism

- Examples of dynamic data structures have relied on *null terminated ends.*
  - Use null to show end of list, no children
- Alternative form
  - use structural recursion and polymorphism

BST Interface

```
public interface BST {
   public int size();
   public boolean contains(Comparable obj);
   public boolean add(Comparable obj);
}
```

EmptyBST

```
public class EmptyBST implements BST {

    private static EmptyBST theOne = new EmptyBST();

    private EmptyBST(){}

    public static EmptyBST getEmptyBST(){ return theOne; }

    public NEBST add(Comparable obj) { return new NEBST(obj); }

    public boolean contains(Comparable obj) { return false; }

    public int size() { return 0; }
}
```


Non Empty BST – Part 1

```
public class NEBST implements BST {

    private Comparable data;
    private BST left;
    private BST right;

    public NEBST(Comparable d){
        data = d;
        right = EmptyBST.getEmptyBST();
        left = EmptyBST.getEmptyBST();
    }

    public BST add(Comparable obj) {
        int val = obj.compareTo( data );
        if( val < 0 )
          left = left.add( obj );
        else if( val > 0 )
          right = right.add( obj );
        return this;
    }
```


Non Empty BST – Part 2

```
public boolean contains(Comparable obj){
        int val = obj.compareTo(data);
        if( val == 0 )
            return true;
        else if (val < 0)
            return left.contains(obj);
        else
            return right.contains(obj);
    }

    public int size() {
        return 1 + left.size() + right.size();
    }

}
```