"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 an 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 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.
BST Insertion

- Add the following values one at a time to an initially empty binary search tree using the naïve algorithm:

  90 20 9 98 10 28 -25

- What is the resulting tree?
Traversals

- What is the result of an inorder traversal of the resulting tree?
- How could a preorder traversal be useful?
Clicker 1

After adding N distinct elements in random order to a Binary Search Tree what is the expected height of the tree? (using the simple insertion algorithm)

A. O(logN)
B. O(N^{1/2})
C. O(N)
D. O(N\log N)
E. O(N^2)
After adding N distinct elements to a Binary Search Tree what is the **worst case** height of the tree? (using the simple insertion algorithm)

A. $O(\log N)$  
B. $O(N^{1/2})$  
C. $O(N)$  
D. $O(N\log N)$  
E. $O(N^2)$
Worst Case Performance

- Insert the following values into an initially empty binary search tree using the simple, naïve algorithm:

  2 3 5 7 11 13 17

- What is the height of the tree?
- What is the worst case height of a BST?
Node for Binary Search Trees

```java
public class BSTNode<E extends Comparable<E> { 
    private Comparable<E> myData;
    private BSTNode<E> myLeft;
    private BSTNode<E> myRight;

    public BSTNode(E item) {
        myData = item;
    }

    public E getValue() {
        return myData;
    }

    public BSTNode<E> getLeft() {
        return myLeft;
    }

    public BSTNode<E> getRight() {
        return myRight;
    }

    public void setLeft(BSTNode<E> b) {
        myLeft = b;
    }
    // setRight not shown
}
```
More on Implementation

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

```java
public class BinarySearchTree<E extends Comparable<E>> {
    private BSTNode<E> root;
    private int size;
}
```
Add an Element, Recursive
Add an Element, Iterative
What are the best case and worst case order to add N distinct elements, one at a time, to an initially empty binary search tree using the simple add algorithm?

<table>
<thead>
<tr>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>B. O(NlogN)</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>C. O(N)</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>D. O(NlogN)</td>
<td>O(N^2)</td>
</tr>
<tr>
<td>E. O(N)</td>
<td>O(N^2)</td>
</tr>
</tbody>
</table>

```java
// given int[] data
// no duplicates in data
BST<Integer> b = new BST<Integer>();
for(int x : data) {
    b.add(x);
}
```
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
Alternate Implementation

- In class 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
public interface BST<E extends Comparable<? super E>> {

    public int size();
    public boolean contains(E obj);
    public boolean add(E obj);
}
public class EmptyBST<E extends Comparable<? super E>> implements BST<E> {

    private static EmptyBST theOne = new EmptyBST();

    private EmptyBST() {}

    public static EmptyBST getEmptyBST(){ return theOne; }

    public BST add(E obj) { return new NEBST(obj); }

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

    public int size() { return 0; }
}
Non Empty BST – Part 1

public class NEBST <E extends Comparable<? super E>> implements BST<E> {

    private E data;
    private BST left;
    private BST right;

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

    public BST add(E obj) {
        int direction = obj.compareTo( data );
        if( direction < 0 )
            left = left.add( obj );
        else if( direction > 0 )
            right = right.add( obj );
        return this;
    }
}
Non Empty BST – Part 2

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

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