"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).

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 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. $O(\log N)$
C. $O(N)$
D. $O(N\log N)$
E. $O(N^2)$

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 BinaryNode(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
```
Worst Case Performance

- Insert the following values into an initially empty binary search tree using the traditional, 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?

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;
}
```
Clicker Question 2

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?

- **Best**
  - A. \( O(N) \)
  - B. \( O(N\log N) \)
  - C. \( O(N) \)
  - D. \( O(N\log N) \)
  - E. \( O(N^2) \)

- **Worst**
  - A. \( O(N) \)
  - B. \( O(N\log N) \)
  - C. \( O(N\log N) \)
  - D. \( O(N^2) \)
  - E. \( O(N^2) \)

---

Remove an Element

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

---

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

---

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

BST Interface

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

EmptyBST

```java
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

```java
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;
    }
}
```
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();
}