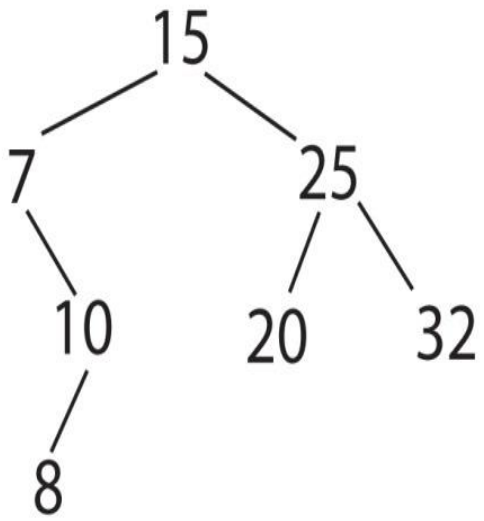
Binary Search Trees

You should remember Binary Search Trees from the Data Structures class. They are binary trees with a data value stored at each node. They have the particular property that at every node, all of the values in the node's left subtree are less than the node's value, and all of the values in its right subtree are greater than the node's value. Here is a picture of a typcal BST:



Binary Search Trees are not necessarily balanced. You saw in CSCI 151 AVL Trees or 2-3 Trees or Red/Black Trees or some other technique that extended BST properties to guarantee that the trees never became too imbalanced. Here we will look at implementing the simple BST properties.

For our tree data structure we will store the data value, the left subtree and the right subtree as a list of 3 elements:

```
(define make-tree (lambda (v t1 t2)
        (list v t1 t2)))

(define left-subtree (lambda (t)
        (cadr t)))

(define right-subtree   (lambda (t)
        (caddr t)))

(define value (lambda (t)
        (car t)))
```

The search function for a BST is pretty simple, and is almost the same in Scheme as it is in Java:

```
(define searchBST (lambda (v t)
    (cond
        [(null? t) #f]
        [(= v (value t)) #t]
        [(< v (value t)) (searchBST v (left-subtree t))]
        [else (searchBST v (right-subtree t))])))
```

To build a BST we will write a function that inserts into a tree one value at a time. Inserting into an empty tree is easy: we just return a leaf node:

```
(define make-leaf (lambda (v)
                    (make-tree v null null)))
```

Inserting into a non-empty tree requires walking down the path from the root to the insertion point. We can't reassign links in a list, so we rebuild the tree just along the search path:

```
(define insertBST (lambda (v t)
                  (cond
                      [(null? t) (make-leaf v)]
                      [(= v (value t)) t]
                      [ (< v (value t))
                            (make-tree (value t)
                                        (insertBST v (left-subtree t))
                                        (right-subtree t))]
                      [else
                            (make-tree (value t)
                                        (left-subtree t)
                                        (insertBST v (right-subtree t)))])))
```