Recall these ADTs:

Set ADT:

    boolean find (ElementType x)
    void insert (ElementType x)
    void remove (ElementType x)

Dictionary ADT:

    AnotherType find (ElementType x)
    void insert (ElementType x, AnotherType y)
    void remove (ElementType x)

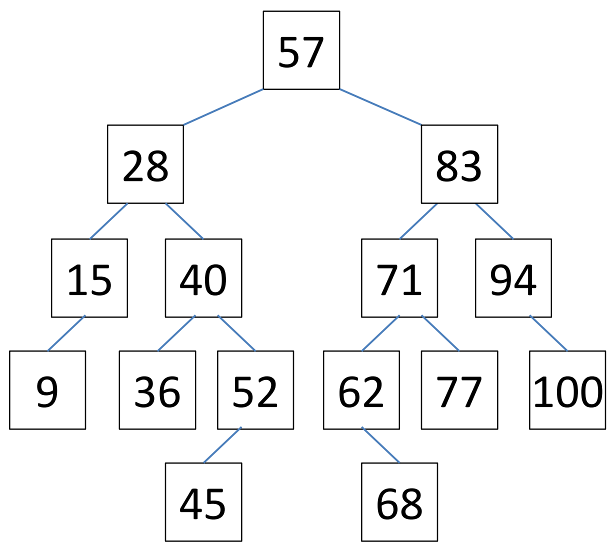
Here the dictionary acts as a function or mapping:
find(x) = y

Previously we implemented Set or Dictionary using hash tables. Now we implement these ADTs using binary search trees.

Binary Search Trees:

Recall that

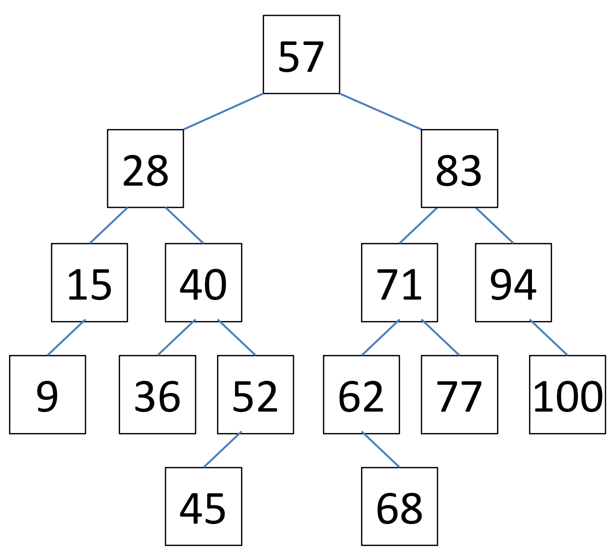
- If node X is in left subtree of node Y, then X.data ≤ Y.data
- If node Z is in right subtree of node Y, then Z.data ≥ Y.data



Note: inorder traversal of BST yields ascending order

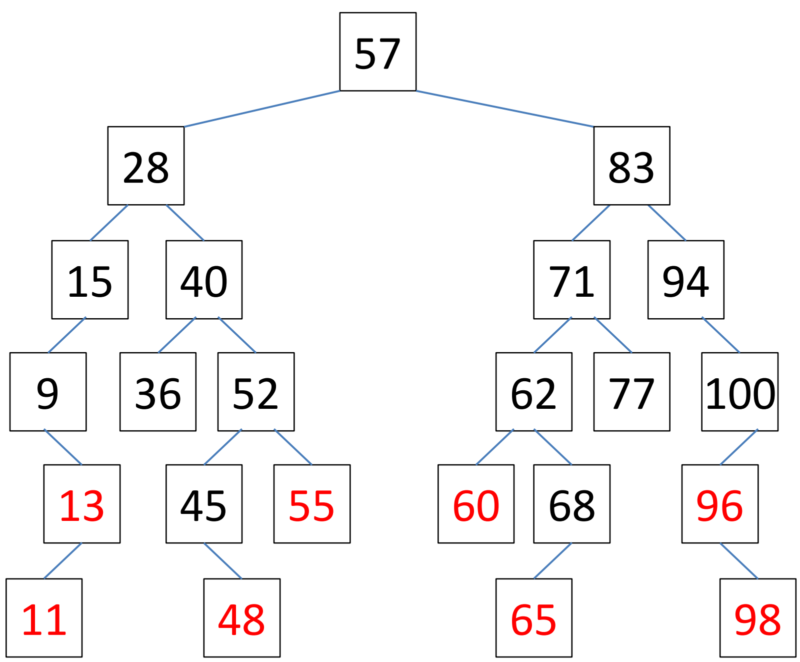
```
class Node {
        ElementType data;
        Node left, right;
        Node (ElementType x)
                { data = x;  left = null;  right = null; }
}
class BinarySearchTree {
        Node root;
        BinarySearchTree( ) { root = null; }

        boolean find (ElementType x) {
                return find (x, root);
        }
        boolean find (ElementType x, Node p) {
                if (p==null) return false;
                if (x==p.data) return true;
                if (x<p.data) return find (x, p.left);
                /* if (x>p.data) */ return find (x, p.right);
        }
        …
```



```
void insert (ElementType x) {
	root = insert (x, root);
}
Node insert (ElementType x, Node p) {
	if (p==null)
		return new Node (x);
	if (x<p.data)
		p.left = insert (x, p.left);
	else if (x>p.data)
		p.right = insert (x, p.right);
	else
		throw exception (“Duplicate key”);
	return p;
}
…
```

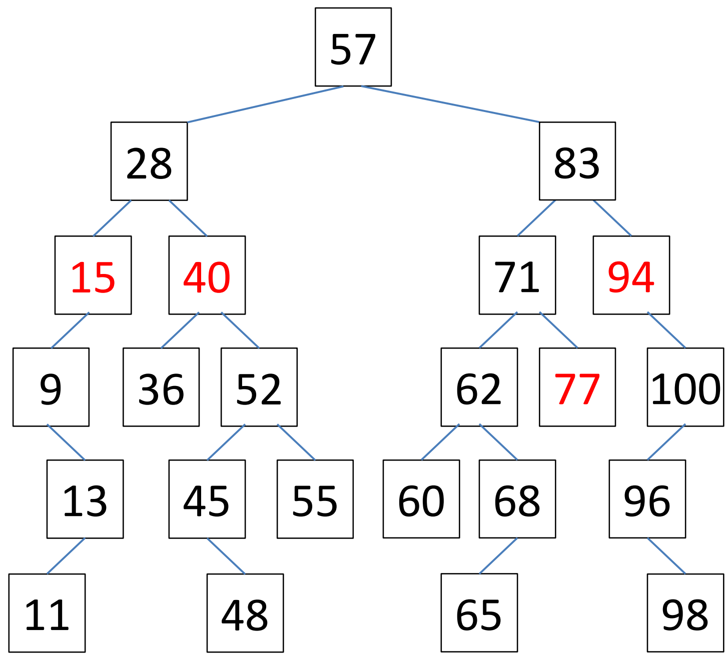
Examples: insert (13); insert (11); insert (55); insert (48);
insert (60); insert (65); insert (96); insert (98);

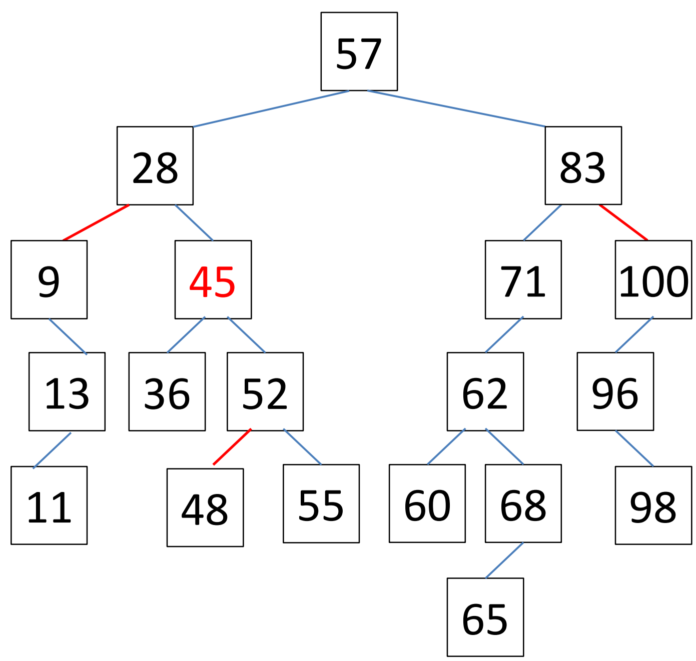


```
    void remove (ElementType x) {
        root = remove (x, root);
    }
    Node remove (ElementType x, Node p) {
        if (p==null)
            throw exception ("Key not found");
        if (x<p.data)
            p.left = remove (x, p.left);
        else if (x>p.data)
            p.right = remove (x, p.right);
        else if (p.right==null)
            p = p.left;
        else if (p.left==null)
            p = p.right;
        else {     // node p has two children
            p.data = findMin (p.right);   // successor of x
            p.right = remove (p.data, p.right);
        }
        return p;
    }
    ElementType findMin (Node p) {
        if (p.left==null)  return p.data;
        return findMin (p.left);
    }
}
```

[Alternatively, we could replace successor with predecessor by invoking findMax(p.left) rather than findMin(p.right).]

Examples: remove (77); remove (15);
remove (94); remove (40);





Analysis:

Each BST operation (find, insert, remove) runs in time $\theta(h)$ where h = height of tree

Worst-case time: $\theta(h) = \theta(n)$,
if keys are inserted in ascending order or descending order

Expected or average-case time: $\theta(h) = \theta(\lg n)$,
if keys are inserted and removed in random order

Note:

There exist other kinds of search trees such that the height and the running times are worst-case $\theta(\lg n)$, but these are topics for a later course