Binary Search Trees (BSTs)
Consider the search operation FindKey (): find an element of a particular key value in a binary tree.

This operation takes $O(n)$ time in a binary tree.

In a binary tree of $10^6$ nodes $\rightarrow 10^6$ steps required at least.

In a BST this operation can be performed very efficiently: $O(\log_2 n)$.

A binary search tree of $10^6$ nodes $\rightarrow \log_2(10^6) \approx 20$ steps only are required.
A binary search tree is a binary tree such that for each node, say N, the following statements are true:

1. If L is any node in the left subtree of N, then L is less than N.
2. If R is any node in the right subtree of N, then R is greater than N.
BST: Example.
The search operation in a binary search tree can be carried out as:

While (the target element is not found and there is more tree to search) do
  if the target element is “less than” the current element then search the left subtree else search the right subtree.
Elements: The elements are nodes (BSTNode), each node contains the following data type: Type

Structure: hierarchical structure; each node can have two children: left or right child; there is a root node and a current node. If N is any node in the tree, nodes in the left subtree < N and nodes in the right subtree > N.

Domain: the number of nodes in a BST is bounded; type/class name is BST
ADT Binary Search Tree

Operations:

1. **Method** FindKey (int tkey, boolean found).
   
   **results:** If bst contains a node whose key value is tkey, then that node is made the current node and found is set to true; otherwise found is set to false and either the tree is empty or the current node is the node to which the node with key = tkey would be attached as a child if it were added to the BST.
2. **Method** Insert (int k, Type e, boolean inserted)
   **requires**: Full (bst) is false. **input**: key, e.
   **results**: if bst does not contain k then inserted is set to true and node with k and e is inserted and made the current element; otherwise inserted is set to false and current value does not change. **output**: inserted.

3. **Method** Remove_Key (int tkey, boolean removed)
   **input**: tkey
   **results**: Node with key value tkey is removed from the bst and removed set to true. If BST is not empty then root is made the current. **output**: removed
4. Method Update(Type e)
   requires: Empty(bst) is false. results: current node’s element is replaced with e. These operations have the same specification as ADT Binary Tree.
5. Method Traverse (Order ord)
6. Method DeleteSub ( )
7. Method Retrieve (Type e)
8. Method Empty ( boolean empty ).
ADT Binary Search Tree

9. **Method** Full (boolean full)
public class BSTNode <T> {
    public int key;
    public T data;
    public BSTNode<T> left, right;
    /** Creates a new instance of BSTNode */
    public BSTNode(int k, T val) {
        key = k;
        data = val;
        left = right = null;
    }
}
public class BST <T> {
    BSTNode<T> root, current;
    /** Creates a new instance of BST */
    public BST() {
        root = current = null;
    }
    public boolean empty() {
        return root == null ? true : false;
    }
    public T retrieve() {
        return current.data;
    }
}
public boolean findkey(int tkey) {
    BSTNode<T> p = root, q = root;
    if (empty()) return false;
    while (p != null) {
        q = p;
        if (p.key == tkey) {
            current = p; return true;
        } else if (tkey < p.key)
            p = p.left;
        else
            p = p.right;
    }
    current = q; return false;
}
public boolean insert (int k, T val) {
    BSTNode<T> p, q = current;
    if (findkey(k)) {
        current = q; /* findkey() has modified current */
        return false; /* key already in the BST */
    }
    p = new BSTNode<T>(k, val);
    if (empty()) {
        root = current = p; return true;
    } else {
        /* current is pointing to parent of the new key. */
        if (k < current.key)
            current.left = p;
        else
            current.right = p;
        current = p; return true;
    }
}
There are three cases:

- Case 1: Node to be deleted has no children.
- Case 2: Node to be deleted has one child.
- Case 3: Node to be deleted has two children.

In all these cases it is always a leaf node that gets deleted.
BST Deletion: Case 1

Delete 80
BST Deletion: Case 2

Delete 110
BST Deletion: Case 3

Delete 60

Rightmost Node in Left Subtree
public boolean remove_key (int tkey){
    Flag removed = new Flag(false);
    BSTNode<T> p;
    p = remove_aux(tkey, root, removed);
    current = root = p;
    return removed;
}
private BSTNode<T> remove_aux(int key, BSTNode<T> p, Boolean flag) {
    BSTNode<T> q, child = null;
    if (p == null)
        return null;
    if (key < p.key)
        p.left = remove_aux(key, p.left, flag); // go left
    else if (key > p.key)
        p.right = remove_aux(key, p.right, flag); // go right
    else {
        flag = true;
        if (p.left != null && p.right != null) { // two children
            q = find_min(p.right);
            p.key = q.key; p.data = q.data;
            p.right = remove_aux(q.key, p.right, flag);
        }
    }
}
else {
    if (p.right == null) // one child case
        child = p.left;
    else if (p.left == null) // one child case
        child = p.right;
    return child;
}
return p;
private BSTNode<T> find_min(BSTNode<T> p) {
    if (p == null) return null;
    while (p.left != null) {
        p = p.left;
    }
    return p;
}
ADT Binary Search Tree

```java
public boolean update(int key, T data) {
    remove_key(current.key);
    return insert(key, data);
}
```