

Here's a partial generic BST interface:

```
public class BST<T extends Comparable<? super T>> {

    class BinaryNode {
        . . .
        T          element; // the data in the node
        BinaryNode left;    // pointer to the left child
        BinaryNode right;   // pointer to the right child
    }

    BinaryNode root;       // pointer to root node, if present

    public BST( ) { . . . }

    public boolean isEmpty( ) { . . . }
    public T      find( T x ) { . . . }
    public boolean insert( T x ) { . . . }
    public boolean remove( T x ) { . . . }
    public void   clear( ) { . . . }

    public boolean equals(Object other) { . . . }
    // private methods follow
}
```

Here's a partial generic BST interface, adapted from Weiss:

```
public class BST<T extends Comparable<? super T>> {  
    . . .  
}
```

```
public int compareTo(Object o)
```

Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

See Weiss 1.5.6

Here's a partial generic BST node interface, adapted from Weiss:

```
class BinaryNode {  
  
    // Constructors  
    public BinaryNode( T elem) {  
        . . .  
    }  
  
    public BinaryNode( T elem, BinaryNode lt, BinaryNode rt ) {  
        . . .  
    }  
  
    // The node class and the following members have package access:  
  
    T          element;    // The data in the node  
    BinaryNode left;      // Pointer to the left child  
    BinaryNode right;     // Pointer to the right child  
}
```

The BST find() function provides client access to data objects within the tree:

```
public T find( T x ) {  
    return find( x, root );  
}
```

```
private T find( T x, BinaryNode<T> sRoot ) {  
    if ( sRoot == null )  
        return null;  
  
    int compareResult = x.compareTo( sRoot.element );  
  
    if ( compareResult < 0 )  
        return find( x, sRoot.left );  
    else if ( compareResult > 0 )  
        return find( x, sRoot.right );  
    else  
        return sRoot.element;    // Match  
}
```

Warning:

be sure you understand the potential dangers of supplying this function... and the benefits of doing so...

The public `insert ()` function is just a stub to call the recursive helper:

```
public boolean insert( T x ) {  
    root = insert( x, root );  
    . . .  
}
```

Warning:

the BST definition in these notes does not allow for duplicate data values to occur, the logic of insertion may need to be changed for your specific application.

The stub simply calls the helper function..

The helper function must find the appropriate place in the tree to place the new node.

The design logic is straightforward:

- locate the parent "node" of the new leaf, and
- hang a new leaf off of it, on the correct side

The insert () helper function:

```
private BinaryNode insert( T x, BinaryNode sRoot ) {  
  
    if ( sRoot == null )  
        return new BinaryNode( x, null, null );  
  
    int compareResult = x.compareTo( sRoot.element );  
  
    if ( compareResult < 0 )  
        sRoot.left = insert( x, sRoot.left );  
    else if ( compareResult > 0 )  
        sRoot.right = insert( x, sRoot.right );  
    else  
        ; // Duplicate; do nothing  
    return sRoot;  
}
```

When the parent of the new value is found, one more recursive call takes place, passing in a null pointer to the helper function.

Note that the insert helper function must be able to modify the node pointer parameter, and that the search logic is precisely the same as for the find() function.

```
private BinaryNode insert( T x, BinaryNode sRoot ) {  
  
    if ( sRoot == null )  
        return new BinaryNode( x, null, null );  
  
    int compareResult = x.compareTo( sRoot.element );  
  
    if ( compareResult < 0 )  
        sRoot.left = insert( x, sRoot.left );  
    . . .  
    return sRoot;  
}
```

When we install the new node, we must modify a reference in the parent node.

Java references are primitives and we cannot pass a reference to a primitive.

The design here:

- creates the new node during a call that "falls off a branch of the tree"
- installs the new node after returning to the call in the parent node

The public `delete ()` function is very similar to the insertion function:

```
public boolean delete( T x ) {  
    root = delete( x, root );  
    . . .  
}
```

The `delete ()` helper function design is also relatively straightforward:

- locate the parent of the node containing the target value
- determine the deletion case (as described earlier) and handle it:
 - parent has only one subtree
 - parent has two subtrees

The details of implementing the delete helper function are left to the reader...

Some binary tree implementations employ parent pointers in the nodes.

- increases memory cost of the tree (probably insignificantly)
- increases complexity of insert/delete/copy logic (insignificantly)
- provides some unnecessary alternatives when implementing insert/delete
- may actually simplify the addition of iterators to the tree (later topic)

The given BST template may also provide additional features:

- a function to provide the size of the tree
- a function to provide the height of the tree
- a function to display the tree in a useful manner

It is also useful to have some instrumentation during testing. For example:

- log the values encountered and the directions taken during a search

This is also easy to add, but it poses a problem since we generally do not want to see such output when the BST is used.

I resolve this by adding some data members and mutators to the template that enable the client to optionally associate an output stream with the object, and to turn logging of its operation on and off as needed.