Generic BST Interface

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

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

    private static class BinaryNode<T> {
       . . .
    }

    private BinaryNode<T> root;

    public BST( ) { . . . }
    public void    clear( ) { . . . }
    public boolean isEmpty( ) { . . . }
    public boolean contains( T x ) { . . . }
    public T       find( T x ) { . . . }
    public T       findMin( ) { . . . }
    public T       findMax( ) { . . . }
    public void    insert( T x ) { . . . }
    public void    delete( T x ) { . . . }
    public void    printTree( ) { . . . }
    . . .
}
```

Generic BST Interface

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 1.5.5

Generic BST Node

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

```
private static class BinaryNode<T> {
   // Constructors
   BinaryNode( T theElement ){

      this( theElement, null, null );
   }

   BinaryNode( T theElement, BinaryNode<T> lt,
               BinaryNode<T> rt ){

      element  = theElement;
      left     = lt;
      right    = rt;
   }

   T              element;  // The data in the node
   BinaryNode<T> left;     // Left child
   BinaryNode<T> right;    // Right child
}
```

BST `contains()` Implementation

The BST `contains()` function takes advantage of the BST data organization:

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

```
private boolean contains( T x, BinaryNode<T> t ) {
   if ( t == null )
      return false;

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      return contains( x, t.left );
   else if ( compareResult > 0 )
      return contains( x, t.right );
   else
      return true;    // Match
}
```

Search direction is determined by relationship of target data to data in current node.

BST `find()` Implementation

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> t ) {
   if ( t == null )
      return null;

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      return find( x, t.left );
   else if ( compareResult > 0 )
      return find( x, t.right );
   else
      return t.element;    // Match
}
```

Warning:

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

BST `insert()` Implementation

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

```
public void 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

BST `insert()` Helper

The `insert()` helper function:

```
private BinaryNode<T> insert( T x, BinaryNode<T> t ) {

   if ( t == null )
      return new BinaryNode<T>( x, null, null );

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      t.left = insert( x, t.left );
   else if ( compareResult > 0 )
      t.right = insert( x, t.right );
   else
      ;  // Duplicate; do nothing
   return t;
}
```

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.

BST `delete()` Implementation

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

```
public void 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…

Parent Pointers

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)

Some Refinements

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.