```
// Basic node stored in unbalanced binary search trees
// Note that this class is not accessible outside
// of this package.

class BinaryNode<AnyType>
{
    AnyType              element;  // The data in the node
    BinaryNode<AnyType> left;     // Left child
    BinaryNode<AnyType> right;    // Right child



// Constructor
    BinaryNode( AnyType theElement )
    {
        element = theElement;
        left = right = null;
    }

     }



public class BinarySearchTree<AnyType extends Comparable<?
super AnyType>>

{

      /** The tree root. */
    protected BinaryNode<AnyType> root;



     /**
     * Construct the tree.
     */
    public BinarySearchTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree.
     * @param x the item to insert.
     * @throws DuplicateItemException if x is already present.
     */
    public void insert( AnyType x )
    {
        root = insert( x, root );
    }
```

```
/**
 * Remove from the tree..
 * @param x the item to remove.
 * @throws ItemNotFoundException if x is not found.
 */
public void remove( AnyType x )
{
    root = remove( x, root );
}

/**
 * Remove minimum item from the tree.
 * @throws ItemNotFoundException if tree is empty.
 */
public void removeMin( )
{
    root = removeMin( root );
}

/**
 * Find the smallest item in the tree.
 * @return smallest item or null if empty.
 */
public AnyType findMin( )
{
    return elementAt( findMin( root ) );
}

/**
 * Find the largest item in the tree.
 * @return the largest item or null if empty.
 */
public AnyType findMax( )
{
    return elementAt( findMax( root ) );
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public AnyType find( AnyType x )
{
    return elementAt( find( x, root ) );
}

/**
 * Make the tree logically empty.
 */
```

```
    public void makeEmpty( )
    {
        root = null;
    }

    /**
     * Test if the tree is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return root == null;
    }

    /**
    * Internal method to get element field.
     * @param t the node.
     * @return the element field or null if t is null.
     */
    private AnyType elementAt( BinaryNode<AnyType> t )
    {
       return t == null ? null : t.element;
    }

    /**
     * Internal method to insert into a subtree.
     * @param x the item to insert.
     * @param t the node that roots the tree.
     * @return the new root.
     * @throws DuplicateItemException if x is already present.
     */
    protected BinaryNode<AnyType> insert( AnyType x,
BinaryNode<AnyType> t )
    {
        if( t == null )
            t = new BinaryNode<AnyType>( x );
        else if( x.compareTo( t.element ) < 0 )
            t.left = insert( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = insert( x, t.right );
        else
            throw new DuplicateItemException( x.toString( ) );
// Duplicate
        return t;
    }

   /**
     * Internal method to remove from a subtree.
     * @param x the item to remove.
     * @param t the node that roots the tree.
     * @return the new root.
```

```
     * @throws ItemNotFoundException if x is not found.
     */
    protected BinaryNode<AnyType> remove( AnyType x,
BinaryNode<AnyType> t )
    {
        if( t == null )
            throw new ItemNotFoundException( x.toString( ) );
        if( x.compareTo( t.element ) < 0 )
            t.left = remove( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = remove( x, t.right );
        else if( t.left != null && t.right != null ) // Two
children
        {
            t.element = findMin( t.right ).element;
            t.right = removeMin( t.right );
        }
        else
            t = ( t.left != null ) ? t.left : t.right;
        return t;
    }

    /**
     * Internal method to remove minimum item from a subtree.
     * @param t the node that roots the tree.
     * @return the new root.
     * @throws ItemNotFoundException if t is empty.
     */
    protected BinaryNode<AnyType> removeMin(
BinaryNode<AnyType> t )
    {
        if( t == null )
            throw new ItemNotFoundException( );
        else if( t.left != null )
        {
            t.left = removeMin( t.left );
            return t;
        }
       else
            return t.right;
    }

    /**
     * Internal method to find the smallest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the smallest item.
     */
    protected BinaryNode<AnyType> findMin( BinaryNode<AnyType>
t )
    {
        if( t != null )
```

```
            while( t.left != null )
                t = t.left;

        return t;
    }

    /**
     * Internal method to find the largest item in a subtree.
     * @param t the node that roots the tree.
     * @return node containing the largest item.
     */
    private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t
)
    {
        if( t != null )
            while( t.right != null )
                t = t.right;

        return t;
    }

    /**
     * Internal method to find an item in a subtree.
     * @param x is item to search for.
     * @param t the node that roots the tree.
     * @return node containing the matched item.
     */
    private BinaryNode<AnyType> find( AnyType x,
BinaryNode<AnyType> t )
    {
        while( t != null )
        {
            if( x.compareTo( t.element ) < 0 )
                t = t.left;
            else if( x.compareTo( t.element ) > 0 )
                t = t.right;
            else
                return t;    // Match
        }

        return null;         // Not found
    }

            // Test program




/**
 * Exception class for duplicate item errors
```

```
 * in search tree insertions.
 */
public class DuplicateItemException extends RuntimeException
{
    /**
     * Construct this exception object.
     */
    public DuplicateItemException( )
    {
        super( );
    }
    /**
     * Construct this exception object.
     * @param message the error message.
     */
    public DuplicateItemException( String message )
    {
        super( message );
    }
}
```

```
/**

How to user BinarySerach tress. See example
*/

public static void main( String [ ] args )
    {
        BinarySearchTree<Integer> t = new
BinarySearchTree<Integer>( );
        final int NUMS = 4000;
        final int GAP  =   37;

        System.out.println( "Checking... (no more output means
success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( i );

        for( int i = 1; i < NUMS; i += 2 )
            t.remove( i );

        if( t.findMin( ) != 2 || t.findMax( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i += 2 )
             if( t.find( i ) != i )
```

```
                System.out.println( "Find error1!" );

        for( int i = 1; i < NUMS; i += 2 )
            if( t.find( i ) != null )
                System.out.println( "Find error2!" );
    }
}
```