Appendices

Appendix A: Experiment's Program

```
set_Min = 1
set_Max = 100 //changes

for(int trial = 1; trial <= 10; trial++) {
   AvlTree avl = new AvlTree();
   SplayTree spl = new SplayTree();


long startAVL = System.nanoTime();
for (double a = set_Min; a <= set; a++) {
   avl.insert(a);
}
long endAVL = System.nanoTime();

long startSPL = System.nanoTime();
ArrayList<Integer> list = new ArrayList<Integer>();
for (int b = set_Min; b <= set_Max; b++) {
   list.add(new Integer(b));
}
   Collections.shuffle(list);
   for (int b = 0; b < set_Max; b++) {
      spl.insert(b);
}
long endSPL = System.nanoTime();

System.out.println("AVL Trial " + trial + ": " + (endAVL - startAVL));
System.out.println("Splay Trial " + trial + ": " + (endSPL - startSPL));

}
```

Appendix B: Raw Data

B1: Splay Tree

Set Sizes	100		200		300		400		500	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	201744	0.000201744	436492	0.000436492	2078642	0.002078642	2197050	0.00219705	3173905	0.003173905
Test 2	156784	0.000156784	282684	0.000282684	252479	0.000252479	856579	0.000856579	1079528	0.001079528
Test 3	93055	0.000093055	323414	0.000323414	709368	0.000709368	433070	0.00043307	1397736	0.001397736
Test 4	149931	0.000149931	99583	0.000099583	632834	0.000632834	144919	0.000144919	357274	0.000357274
Test 5	71035	0.000071035	244933	0.000244933	870898	0.000870898	433070	0.00043307	1475498	0.001475498
Test 6	328255	0.000328255	1099802	0.001099802	178642	0.000178642	635810	0.00063581	995519	0.000995519
Test 7	248163	0.000248163	336590	0.00033659	564300	0.0005643	543831	0.000543831	514676	0.000514676
Test 8	163265	0.000163265	208628	0.000208628	1171240	0.00117124	1252773	0.001252773	354114	0.000354114
Test 9	52125	0.000052125	292247	0.000292247	474865	0.000474865	433070	0.00043307	659687	0.000659687
Test 10	80748	0.000080748	195845	0.000195845	164060	0.00016406	383044	0.000383044	115026	0.000115026
Avg.	**154510**	**0.0001545105**	**352022**	**0.0003520218**	**709732**	**0.0007097328**	**731322**	**0.0007313216**	**1012296**	**0.0010122963**

Set Sizes	600		700		800		900		1000	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	2416212	0.002416212	1160855	0.001160855	3017459	0.003017459	4477437	0.004477437	1841080	0.00184108
Test 2	999352	0.000999352	3492116	0.003492116	1693436	0.001693436	3818446	0.003818446	5390321	0.005390321
Test 3	1213070	0.00121307	405161	0.000405161	559638	0.000559638	329510	0.00032951	3167011	0.003167011
Test 4	1001918	0.001001918	516346	0.000516346	1468798	0.001468798	527725	0.000527725	1183997	0.001183997
Test 5	1973998	0.001973998	2460033	0.002460033	2336282	0.002336282	872963	0.000872963	715350	0.00071535
Test 6	670895	0.000670895	1870045	0.001870045	981249	0.000981249	2001991	0.002001991	1459759	0.001459759
Test 7	313603	0.000313603	966336	0.000966336	1440729	0.001440729	1602935	0.001602935	1431305	0.001431305
Test 8	2026700	0.0020267	318571	0.000318571	599940	0.00059994	1924004	0.001924004	3036682	0.003036682
Test 9	298094	0.000298094	1429579	0.001429579	720056	0.000720056	703180	0.00070318	782415	0.000782415
Test 10	774814	0.000774814	1012839	0.001012839	1192644	0.001192644	965214	0.000965214	1006353	0.001006353
Avg.	**1168866**	**0.0011688656**	**1363188**	**0.0013631881**	**1401023**	**0.0014010231**	**1722341**	**0.0017223405**	**2001427**	**0.0020014273**

B2: AVL Tree

Set Sizes	100		200		300		400		500	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	110863	0.000110863	1102057	0.001102057	889309	0.000889309	382803	0.000382803	921303	0.000921303
Test 2	246155	0.000246155	390278	0.000390278	563106	0.000563106	791573	0.000791573	792251	0.000792251
Test 3	480647	0.000480647	582118	0.000582118	661011	0.000661011	1703945	0.001703945	589451	0.000589451
Test 4	372377	0.000372377	348519	0.000348519	2170925	0.002170925	1729676	0.001729676	636593	0.000636593
Test 5	1452811	0.001452811	101846	0.000101846	175012	0.000175012	621867	0.000621867	924509	0.000924509
Test 6	1051363	0.001051363	175065	0.000175065	961058	0.000961058	591686	0.000591686	1730891	0.001730891
Test 7	282112	0.000282112	193408	0.000193408	492268	0.000492268	871308	0.000871308	497046	0.000497046
Test 8	333940	0.00033394	2069873	0.002069873	1036203	0.001036203	451748	0.000451748	1927493	0.001927493
Test 9	411330	0.00041133	494060	0.00049406	656422	0.000656422	628317	0.000628317	501453	0.000501453
Test 10	115329	0.000115329	700666	0.000700666	760935	0.000760935	550865	0.000550865	1728430	0.00172843
Avg.	**485692**	**0.0004856927**	**615789**	**0.000615789**	**836625**	**0.0008366249**	**832379**	**0.0008323788**	**1024942**	**0.001024942**

Set Sizes	600		700		800		900		1000	
Unit	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec	NanoSec	Sec
Test 1	662959	0.000662959	2634935	0.002634935	4157917	0.004157917	3540938	0.003540938	6721712	0.006721712
Test 2	2402168	0.002402168	1357045	0.001357045	2046193	0.002046193	2232438	0.002232438	3846201	0.003846201
Test 3	493687	0.000493687	1127999	0.001127999	166998	0.000166998	652454	0.000652454	2037499	0.002037499
Test 4	811742	0.000811742	712934	0.000712934	1834990	0.00183499	4720899	0.004720899	1455077	0.001455077
Test 5	1398992	0.001398992	881823	0.000881823	1707961	0.001707961	598151	0.000598151	408907	0.000408907
Test 6	914825	0.000914825	1972825	0.001972825	2654468	0.002654468	1616466	0.001616466	674037	0.000674037
Test 7	3630656	0.003630656	665437	0.000665437	1497781	0.001497781	1008058	0.001008058	1787696	0.001787696
Test 8	706410	0.00070641	3013506	0.003013506	1095276	0.001095276	1041388	0.001041388	370219	0.000370219
Test 9	495274	0.000495274	1072825	0.001072825	218558	0.000218558	1251639	0.001251639	4183004	0.004183004
Test 10	318222	0.000318222	1638759	0.001638759	606952	0.000606952	851391	0.000851391	1045791	0.001045791
Avg.	**1183494**	**0.0011834935**	**1507808**	**0.0015078088**	**1598709**	**0.0015987094**	**1751382**	**0.0017513822**	**2253014**	**0.0022530143**

Appendix C: Splay Tree's Java Code (Mark Weiss)

```
// SplayTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// boolean contains( x )  --> Return true if x is found
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Throws UnderflowException as appropriate

/**
 * Implements a top-down splay tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class SplayTree<AnyType extends Comparable<? super AnyType>>
{
   /**
    * Construct the tree.
    */
   public SplayTree( )
   {
      nullNode = new BinaryNode<AnyType>( null );
      nullNode.left = nullNode.right = nullNode;
      root = nullNode;
   }

      private BinaryNode<AnyType> newNode = null;   // Used between different
inserts

   /**
    * Insert into the tree.
    * @param x the item to insert.
    */
```

```
public void insert( AnyType x )
{
   if( newNode == null )
      newNode = new BinaryNode<AnyType>( null );
   newNode.element = x;

   if( root == nullNode )
   {
      newNode.left = newNode.right = nullNode;
      root = newNode;
   }
   else
   {
      root = splay( x, root );

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

      if( compareResult < 0 )
      {
         newNode.left = root.left;
         newNode.right = root;
         root.left = nullNode;
         root = newNode;
      }
      else
      if( compareResult > 0 )
      {
         newNode.right = root.right;
         newNode.left = root;
         root.right = nullNode;
         root = newNode;
      }
      else
         return;   // No duplicates
   }
   newNode = null;   // So next insert will call new
}
```

```
/**
 * Remove from the tree.
 * @param x the item to remove.
 */
public void remove( AnyType x )
{
    if( !contains( x ) )
        return;

    BinaryNode<AnyType> newTree;

        // If x is found, it will be splayed to the root by contains
    if( root.left == nullNode )
        newTree = root.right;
    else
    {
        // Find the maximum in the left subtree
        // Splay it to the root; and then attach right child
        newTree = root.left;
        newTree = splay( x, newTree );
        newTree.right = root.right;
    }
    root = newTree;
}

/**
 * Find the smallest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *    amortized behavior.
 * A good alternative is to first call find with parameter
 *    smaller than any item in the tree, then call findMin.
 * @return the smallest item or throw UnderflowException if empty.
 */
public AnyType findMin( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    BinaryNode<AnyType> ptr = root;

    while( ptr.left != nullNode )
```

```
        ptr = ptr.left;

    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *     amortized behavior.
 * A good alternative is to first call find with parameter
 *     larger than any item in the tree, then call findMax.
 * @return the largest item or throw UnderflowException if empty.
 */
public AnyType findMax( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    BinaryNode<AnyType> ptr = root;

    while( ptr.right != nullNode )
        ptr = ptr.right;

    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return true if x is found; otherwise false.
 */
public boolean contains( AnyType x )
{
    if( isEmpty( ) )
        return false;

    root = splay( x, root );

    return root.element.compareTo( x ) == 0;
}
```

```
    /**
     * Make the tree logically empty.
     */
    public void makeEmpty( )
    {
        root = nullNode;
    }

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

      private BinaryNode<AnyType> header = new BinaryNode<AnyType>( null ); //
For splay

    /**
     * Internal method to perform a top-down splay.
     * The last accessed node becomes the new root.
     * @param x the target item to splay around.
     * @param t the root of the subtree to splay.
     * @return the subtree after the splay.
     */
    private BinaryNode<AnyType> splay( AnyType x, BinaryNode<AnyType> t )
    {
        BinaryNode<AnyType> leftTreeMax, rightTreeMin;
        header.left = header.right = nullNode;
        leftTreeMax = rightTreeMin = header;
        nullNode.element = x;   // Guarantee a match
        for( ; ; )
        {
            int compareResult = x.compareTo( t.element );

            if( compareResult < 0 )
            {
                if( x.compareTo( t.left.element ) < 0 )
                    t = rotateWithLeftChild( t );
```

```
            if( t.left == nullNode )
                break;
            // Link Right
            rightTreeMin.left = t;
            rightTreeMin = t;
            t = t.left;
        }

        else if( compareResult > 0 )
        {
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            if( t.right == nullNode )
                break;
            // Link Left
            leftTreeMax.right = t;
            leftTreeMax = t;
            t = t.right;
        }
        else
            break;
    }

    leftTreeMax.right = t.left;
    rightTreeMin.left = t.right;
    t.left = header.right;
    t.right = header.left;
    return t;
}


/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 */
        private static <AnyType> BinaryNode<AnyType>
rotateWithLeftChild( BinaryNode<AnyType> k2 )
{
    BinaryNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
```

```
    }

    /**
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     */
          private static <AnyType> BinaryNode<AnyType>
rotateWithRightChild( BinaryNode<AnyType> k1 )
    {
        BinaryNode<AnyType> k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        return k2;
    }

    // Basic node stored in unbalanced binary search trees
    private static class BinaryNode<AnyType>
    {
            // Constructors
        BinaryNode( AnyType theElement )
        {
            this( theElement, null, null );
        }

                    BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
BinaryNode<AnyType> rt )
        {
            element  = theElement;
            left     = lt;
            right    = rt;
        }

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

    private BinaryNode<AnyType> root;
    private BinaryNode<AnyType> nullNode;


        // Test program; should print min and max and nothing else
```

```
    public static void main( String [ ] args )
    {
        SplayTree<Integer> t = new SplayTree<Integer>( );
        final int NUMS = 40000;
        final int GAP  =   307;

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

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( i );
        System.out.println( "Inserts complete" );

        for( int i = 1; i < NUMS; i += 2 )
            t.remove( i );
        System.out.println( "Removes complete" );

        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.contains( i ) )
                System.out.println( "Error: find fails for " + i );

        for( int i = 1; i < NUMS; i += 2 )
            if( t.contains( i ) )
                System.out.println( "Error: Found deleted item " + i );
    }
}
```

Appendix D: AVL Tree's Java Code (Mark Weiss)

D1: AvlTree.java

```
// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements an AVL tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class AvlTree
{
   /**
    * Construct the tree.
    */
   public AvlTree( )
   {
      root = null;
   }

   /**
    * Insert into the tree; duplicates are ignored.
    * @param x the item to insert.
    */
   public void insert( Comparable x )
   {
      root = insert( x, root );
   }
```

```
/**
 * Remove from the tree. Nothing is done if x is not found.
 * @param x the item to remove.
 */
public void remove( Comparable x )
{
   System.out.println( "Sorry, remove unimplemented" );
}

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

/**
 * Find the largest item in the tree.
 * @return the largest item of null if empty.
 */
public Comparable 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 Comparable find( Comparable 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;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
   if( isEmpty( ) )
      System.out.println( "Empty tree" );
   else
      printTree( root );
}

/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field or null if t is null.
 */
private Comparable elementAt( AvlNode 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.
 */
private AvlNode insert( Comparable x, AvlNode t )
{
```

```
    if( t == null )
        t = new AvlNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
    else
        ;  // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}

/**
 * Internal method to find the smallest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the smallest item.
 */
private AvlNode findMin( AvlNode t )
{
    if( t == null )
        return t;

    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 AvlNode findMax( AvlNode t )
{
    if( t == null )
        return t;

    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 AvlNode find( Comparable x, AvlNode 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;   // No match
}

/**
 * Internal method to print a subtree in sorted order.
 * @param t the node that roots the tree.
 */
private void printTree( AvlNode t )
{
    if( t != null )
    {
        printTree( t.left );
```

```
            System.out.println( t.element );
            printTree( t.right );
        }
    }

    /**
     * Return the height of node t, or -1, if null.
     */
    private static int height( AvlNode t )
    {
        return t == null ? -1 : t.height;
    }

    /**
     * Return maximum of lhs and rhs.
     */
    private static int max( int lhs, int rhs )
    {
        return lhs > rhs ? lhs : rhs;
    }

    /**
     * Rotate binary tree node with left child.
     * For AVL trees, this is a single rotation for case 1.
     * Update heights, then return new root.
     */
    private static AvlNode rotateWithLeftChild( AvlNode k2 )
    {
        AvlNode k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
        k1.height = max( height( k1.left ), k2.height ) + 1;
        return k1;
    }

    /**
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     * Update heights, then return new root.
     */
    private static AvlNode rotateWithRightChild( AvlNode k1 )
```

```
{
    AvlNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}

/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then return new root.
 */
private static AvlNode doubleWithLeftChild( AvlNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}

/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then return new root.
 */
private static AvlNode doubleWithRightChild( AvlNode k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}

  /** The tree root. */
private AvlNode root;


    // Test program
public static void main( String [ ] args )
{
    AvlTree t = new AvlTree( );
    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( new MyInteger( i ) );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 1 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 1 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 1; i < NUMS; i++ )
             if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                 System.out.println( "Find error1!" );
    }
}
```

D2: AvlNode.java

```
// Basic node stored in AVL trees
// Note that this class is not accessible outside
// of package DataStructures

class AvlNode
{
        // Constructors
    AvlNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    AvlNode( Comparable theElement, AvlNode lt, AvlNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
        height   = 0;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    AvlNode    left;         // Left child
    AvlNode    right;        // Right child
    int        height;       // Height
}
```

Appendix E: Binary Search Tree C Code

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
   int key;
   struct node *left;
   struct node *right;
};

struct node *getNewNode(int val)
{
   struct node *newNode = malloc(sizeof(struct node));
   newNode->key   = val;
   newNode->left  = NULL;
   newNode->right = NULL;

   return newNode;
}

struct node *insert(struct node *root, int val)
{
   if(root == NULL)
      return getNewNode(val);

   if(root->key < val)
      root->right = insert(root->right,val);

   else if(root->key > val)
      root->left = insert(root->left,val);

   return root;
}

void inorder(struct node *root)
{
   if(root == NULL)
      return;
   inorder(root->left);
```

```
    printf("%d ",root->key);
    inorder(root->right);
}

int main()
{
    struct node *root = NULL;
    root = insert(root,100);.
    root = insert(root,50);
    root = insert(root,150);
    root = insert(root,50);

    inorder(root);

    return 0;
}
```