Menu

orials
views
Tips
 Tips
Tips
PI Tips
plications
raries
nes
twork
ums
g

Binary Search Tree Implementation in Java

User Rating: / 474

Poor ○ ○ ○ ○ [✓ ●] Best

In computer science, a binary search tree (BST) is a binary tree which has the following properties:

- Each node has a value.
- A total order is defined on these values.
- The left subtree of a node contains only values less than or equal to the node's value.
- The right subtree of a node contains only values greater than or equal to the node's value.

The major advantage of binary search trees is that the related sorting algorithms and search algorithms such
order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as s

sited Tips
Tips
 Tips
Tips
PI Tips
plications
raries
nes
views
ted Tips
Tips
 Tips
Tips
PI Tips
plications
raries
nes
views

tatistics

ed Users:	2831
Tips:	614
. Tips:	202
Tips:	183
PI Tips:	779
plications:	298
raries:	209
nes:	16
eviews:	

multisets, and associative arrays.

Following code shows how to implement a binary search tree in Java:

```java
// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// void removeMin( )      --> Remove minimum item
// 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
// ******************ERRORS********************************
// Exceptions are thrown by insert, remove, and removeMin if warranted

/**
 * Implements an unbalanced binary search tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class BinarySearchTree {
    /**
     * 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( Comparable 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( Comparable 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 Comparable findMin( ) {
    return elementAt( findMin( root ) );
}

/**
 * Find the largest item in the tree.
 * @return the largest item or 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;
}

/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field or null if t is null.
 */
private Comparable elementAt( BinaryNode 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 insert( Comparable x, BinaryNode t ) {
        if( t == null )
            t = new BinaryNode( 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 remove( Comparable x, BinaryNode 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 x is not found.
     */
    protected BinaryNode removeMin( BinaryNode 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 findMin( BinaryNode 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 findMax( BinaryNode 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 find( Comparable x, BinaryNode 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
    }

    /** The tree root. */
    protected BinaryNode root;


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

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

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

        for( int i = 2; i < NUMS; i+=2 )
            if( ((Integer)(t.find( new Integer( i ) ))).intValue( ) != i )
                System.out.println( "Find error1!" );

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


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

class BinaryNode {
    // Constructors
    BinaryNode( Comparable theElement ) {
        element = theElement;
        left = right = null;
    }

    // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}


/**
 * Exception class for duplicate item errors
 * in search tree insertions.
 * @author Mark Allen Weiss
 */
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 );
    }
}


/**
 * Exception class for failed finds/removes in search
 * trees, hash tables, and list and tree iterators.
 * @author Mark Allen Weiss
 */
public class ItemNotFoundException extends RuntimeException {
    /**
     * Construct this exception object.
     */
    public ItemNotFoundException( ) {
        super( );
    }

    /**
     * Construct this exception object.
     * @param message the error message.
     */
    public ItemNotFoundException( String message ) {
        super( message );
    }
}
```

Related Tips

- A custom combobox editor for use with the EditableComboBox class
- A demonstration of Java2D transformations
- A game of Tic-Tac-Toe that can be played between two client applets
- A Label that uses inline HTML to format its text
- A program to print limits of the primitive types
- A sample of inline DTD definition
- A sample of linked DTD definition
- A sample XML file

< Prev **Next >**

Page 1 of 0 (0 comments)

You can share your information about this topic using the form below!

Please do not post your questions with this form! Thanks.

Name *(required)*

E-Mail *(required)*

¶

Your email will not be displayed on the site - only to our administrator

Homepage*(optional)*

http://

Comment Enable HTML code : [✓ Yes] No