```java
import java.util.Comparator;
import java.util.NoSuchElementException;

import org.w3c.dom.Node;

/**
 * Implementation of Binary Search Tree data structure
 *
 * @author Andry Arthur
 *
 */
public class BSTree<E>
{
    private class Node
    {
        E data;
        Node left;
        Node right;
        Node parent;

        Node(E d)
        {
            data = d;
            parent = null;
            left = null;
            right = null;
        }
    }

    private Node root;

    private Comparator<E> comparator;

    public BSTree(Comparator<E> theComp)
    {
        root = null;
        comparator = theComp;
    }

    /**
     * Returns the root of the tree.
     *
     * @return  the root of the tree
     */
    public Node getNode()
    {
        return root;
    }

    /**
     * Adds the given item to the tree.
     *
     * @param item      item to be added on the tree
     */
    public void addLoop(E item)
    {
        Node nodeNew = new Node(item);
        Node curr = null;
        if(root == null) {
            root = nodeNew;
        }
        else {
            curr = root;
            while(curr != nodeNew) {
                if(comparator.compare(item, curr.data) < 0) {
                    if(curr.left == null) {
                        nodeNew.parent = curr;
                        curr.left = nodeNew;
                    }
                    else {
                        curr = curr.left;
                    }
                }
                else {
                    if(curr.right == null) {
                        nodeNew.parent = curr;
                        curr.right = nodeNew;
                    }
                    else {
                        curr = curr.right;
                    }
                }
            }
        }
        return;
    }

    /**
     * Checks if tree is empty.
     *
     * @return  true if tree is empty
     */
    public boolean isEmpty()
    {
        return root == null;
    }

    //methods that use loops

    /**
     * Returns the maximum value in the tree using loops.
     *
     * @return  the maximum value in the tree
     * @throws  NoSuchElementException when tree is empty
     */
    public E maxValueLoop()
    {
        if(isEmpty()) {
            throw new NoSuchElementException();
        }

        Node container = findMaxNodeLoop(root);

        return container.data;
    }

    /**
     * Returns the node with the maximum value in subtree using loops.
     *
     * @param curr      root of subtree
     * @return  node with the maximum value
     */
    private Node findMaxNodeLoop(Node curr)
    {
        Node currMax = curr;

        while(currMax.right != null) {
            currMax = currMax.right;
        }

        return currMax;
    }

    /**
     * Returns the minimum value using loops
     *
     * @return  the minimum value
     * @throws  NoSuchElementException when tree is empty
     */
    public E minValueLoop()
    {
        if(isEmpty()) {
            throw new NoSuchElementException();
        }

        Node container = findMinNodeLoop(root);

        return container.data;
    }

    /**
     * Returns node with the minimum value in substree using loops.
     *
     * @param curr      root of subtree
     * @return  node with minimum value
     */
    private Node findMinNodeLoop(Node curr)
    {
        Node currMin = curr;

        while(currMin.left != null) {
            currMin = currMin.left;
        }

        return currMin;
    }

    /**
     * Checks whether tree contains item
     *
     * @param item      item to be found in tree
     * @return  true if tree contains item
     */
    public boolean containsLoop(E item)
    {
        if(findNodeLoop(root, item) == null) {
            return false;
        }
        else {
            return true;
        }
    }

    /**
     * Returns the Node with the given item within the given subtree.
     *
     * @param curr      root of subtree
     * @param item      item to be found
     * @return  node with the given item within subtree
     */
    private Node findNodeLoop(Node curr, E item)
    {
        Node currNode = curr;

        while(currNode != null) {
            if(comparator.compare(item, currNode.data) == 0) {
                return currNode;
            }
            else if(comparator.compare(item, currNode.data) < 0) {
                currNode = currNode.left;
            }
            else {
                currNode = currNode.right;
            }
        }

        return currNode;
    }

    //recursive methods

    /**
     * Adds item onto the tree.
     *
     * @param item      item to be added
     */
    public void add(E item)
    {
        //root = new BSTreeUtils<E>().add(root, comparator, new Node(item));   // helper version;
        add(root, item);
    }

    /**
     * Adds the given item onto the tree using recursion
     *
     * @param curr      root of subtree
     * @param item      item to be added
     */
    private void add(Node curr, E item)
    {
        if(this.isEmpty()) {
            root = new Node(item);
        }
        else if (comparator.compare(curr.data, item) < 0) {
            if(curr.right == null) {
            curr.right = new Node(item);
            curr.right.parent = curr;
            }
            else {
                add(curr.right, item);
            }
        }
        else {
            if(curr.left == null) {
                curr.left = new Node(item);
                curr.left.parent = curr;
                }
                else {
                    add(curr.left, item);
                }
        }
    }

    /**
     * Returns the maximum value in the tree using recursion.
     *
     * @return  the maximum value in the tree
     */
    public E maxValue()
    {
        Node max = findMaxNode(root);

        if(max == null) {
            throw new NoSuchElementException();
        }

        return max.data;
    }

    /**
     * Returns the node with the maximum value in subtree using recursion.
     *
     * @param curr      root of subtree
     * @return  node with the maximum value
     */
    private Node findMaxNode(Node curr)
    {
        if(curr == null) {
            return null;
        }
        else if(curr.right == null) {
            return curr;
        }
        else {
            return findMaxNode(curr.right);
        }
    }

    /**
     * Returns the minimum value using recursion.
     *
     * @return  the minimum value
     * @throws  NoSuchElementException when the tree is empty
     */
    public E minValue()
    {
        Node min = findMinNode(root);

        if(min == null) {
            throw new NoSuchElementException();
        }

        return min.data;
    }

    /**
     * Returns node with the minimum value in substree using recursion.
     *
     * @param curr      root of subtree
     * @return  node with minimum value
     */
    private Node findMinNode(Node curr)
    {
        if(curr == null) {
            return null;
        }
        else if(curr.left == null) {
            return curr;
        }
        else {
            return findMinNode(curr.left);
        }
    }

    /**
     * Checks whether tree contains item
     *
     * @param item      item to be found in tree
     * @return  true if tree contains item
     */
    public boolean contains(E item)
    {
        return findNode(root, item) != null;
    }

    /**
     * Returns the Node with the given item within the given subtree.
     *
     * @param curr      root of subtree
     * @param item      item to be found
     * @return  node with the given item within subtree
```

```java
     */
    private Node findNode(Node curr, E item)
    {
        Node nodeFind = null;

        if(curr == null) {
            return null;
        }
        else if(comparator.compare(curr.data, item) == 0) {
            nodeFind = curr;
            return nodeFind;
        }
        else {
            if(comparator.compare(curr.data, item) < 0) {
                return findNode(curr.right, item);
            }
            else {
                return findNode(curr.left, item);
            }
        }
    }

    /**
     * Removes given item from the tree.
     *
     * @param item      item to be removed
     * @return  given item from the tree
     */
    public boolean remove(E item)
    {
        Node nodeToRemove = findNode(root, item);
        if(nodeToRemove == null) {
            return false;
        }
        else if(nodeToRemove == root) {
            root = null;
        }
        else if(nodeToRemove.left != null && nodeToRemove.right != null) {
            removeHasBoth(nodeToRemove);
        }
        else if(nodeToRemove.left == null || nodeToRemove.right == null) {
            removeMissing(nodeToRemove);
        }
        return true;
    }

    /**
     * Removes given node that is parent to either one or no children.
     *
     * @param node      node to be removed
     */
    private void removeMissing(Node node)
    {
        Node parent = node.parent;

        if(node.left == null && node.right == null) {
            if(comparator.compare(parent.left.data, node.data) == 0) {
                parent.left = null;
            }
            else {
                parent.right = null;
            }
        }
        else if (node.left != null) {
            if(comparator.compare(parent.left.data, node.data) == 0) {
                parent.left = node.left;
            }
            else {
                parent.right = node.left;
            }
            node.left.parent = parent;
        }
        else {
            if(comparator.compare(parent.right.data, node.data) == 0) {
                parent.right = node.right;
            }
            else {
                parent.left = node.right;
            }
            node.right.parent = parent;
        }
        //node.parent = null;
        return;
    }

    /**
     * Removes given node that is parent to two other nodes
     *
     * @param node      node to be removed
     */
    private void removeHasBoth(Node node)
    {
        Node parent = node.parent;

        parent.right = node.right;
        node.right.parent = parent;
        node.right.left = node.left;
        node.left.parent = node.right;
    }

    /**
     * Returns a string version of the tree.
     *
     * @return  string version of the tree
     */
    public String toString()
    {
        return new BSTreeUtils<E>().toString(root);
    }
}
```