Universidade do Porto
Faculdade de Engenharia
FEUP

# Development of verified data structures with Dafny: sorted set implemented with a binary search tree

**Mário Mesquita, December, 2020**

**MIEIC/MFES**



# 1. Introduction

A TreeSet is a data structure that holds values in sorted order and without duplicates. This project's goal is to develop a verified implementation of it through a Binary Search Tree, guaranteeing O(h) time complexity on the *contains*, *insert* and *delete* operations, where *h* is the height of the tree. Additionally, an *asSeq* method was created to return the TreeSet elements in their sorted order.
The implementation was based on the BinaryTree example of the official dafny github repository.

# 2. Specification

We start by specifying in Dafny the syntax and semantics (contracts) of the services provided by the implemented classes, TreeSet and BSTNode. Due to some limitations of generics in Dafny, the type parameter (`T`) is instantiated for a concrete type.
The TreeSet's concrete state is abstracted through the use of a set of elements (*elems)* and a set of objects (*Repr*), which are only used for specification and verification purposes, defined as ghost variables. The first variable represents the collection of values that are present in the BST. The second variable holds a reference to all of the objects (nodes) in the BST and represents the dynamic frame of the object's representation (as explained in page 4 of this article). The operations' pre- and post-conditions are then described through these variables.
Similarly, the BSTNode class is also implemented using state abstraction and the same variables: *elems* and *Repr*.

```
// binary tree reference:
// https://github.com/dafny-lang/dafny/blob/master/Test/dafny1/BinaryTree.dfy
```

```
type T = int // example type, but could be other one

// check if a sequence is sorted
predicate isSorted(s: seq<T>) {
    forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}

// check if a sequence does not have duplicates
predicate noDuplicates(s: seq<T>) {
    forall i, j :: 0 <= i < j < |s| ==> s[i] != s[j]
}

// check if a sequence and a set have the same content (same elements and size)
predicate sameContent(s1: seq<T>, s2: set<T>) {
    (forall i :: 0 <= i < |s1| ==> s1[i] in s2) &&
    (forall i :: i in s2 ==> i in s1) &&
    (|s1| == |s2|)
}

// Node of a Binary Search Tree
class BSTNode {
    // Abstract state used for specification & verification purposes
    ghost var elems: set<T> // Holds the set of values in the subtree starting in this node (inclusive).
    ghost var Repr: set<object> // Set of objects that are part of the subtree starting in this node (inclusive)

    // initialize node with given value and no children
    constructor(x: T)
        ensures elems == {x}
        ensures Repr == {this}

      // check if a given value is in the subtree starting in this node (inclusive)
    function method contains(x: T) : bool
        decreases Repr
        ensures contains(x) <==> x in elems

    // delete a given value from the subtree starting in this node (inclusive)
    // returns the new root of this subtree
```

```
    // https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/
    method delete(x: T) returns (node: BSTNode?)
        requires x in elems
        decreases Repr
        ensures node != null ==> node.Valid()
        ensures node == null ==> old(elems) <= {x}
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {x}
        ensures fresh(Repr - old(Repr))

    // deletes the min value of a BST node, returning its values
    // since this is a BST, it will be the leftmost node and it is represented
by its value and right subtree
    method deleteMin() returns (min: T, node: BSTNode?)
        decreases Repr
        ensures fresh(Repr - old(Repr))
        ensures node == null ==> old(elems) == {min}
        ensures node != null ==> node.Valid()
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {min}
         ensures min in old(elems) && (forall x :: x in old(elems) ==> min <=
x)
}

// TreeSet class, that represents a sorted collection of type T values,
without duplicates
// Implemented using a BST, allowing most operations to be implemented in
logarithmic time
class TreeSet {

    // Abstract state
    ghost var elems: set<T> // Set of values in the BST
     ghost var Repr: set<object>  // Set of all objects (nodes) that are part
of the BST, including this TreeSet object

    // initialize TreeSet with null root and 0 elements
    constructor()
        ensures elems == {}
        ensures fresh(Repr - {this})

    // check if a given value is in the TreeSet
    function method contains(x: T): bool
```

```
        ensures contains(x) <==> x in elems

     // insert a value on the TreeSet. If it is repeated it will have no
effect.
    method insert(x: T)
        requires x !in elems
        ensures elems == old(elems) + {x}
        ensures fresh(Repr - old(Repr))

     // recursive helper method to insert a value 'x' on the BST starting at
node 'n'
    // returns a new node 'm' on whose subtree 'x' exists
    // if x already is in the BST nothing will be changed
    static method insertHelper(x: T, n: BSTNode?) returns (m: BSTNode)
        requires n == null || n.Valid()
        requires n != null ==> x !in n.elems
        modifies if n != null then n.Repr else {}
        ensures m.Valid()
        ensures n == null ==> fresh(m.Repr) && m.elems == {x}
        ensures n != null ==> m == n && n.elems == old(n.elems) + {x}
        ensures n != null ==> fresh(n.Repr - old(n.Repr))
        decreases if n == null then {} else n.Repr

    // delete a value x from the TreeSet if it exists
    method delete(x: T)
        requires root != null && x in root.elems
        ensures elems == old(elems) - {x}
        ensures fresh(Repr - old(Repr))

    // return the TreeSet values as a sequence, ordered and without duplicates
    method asSeq() returns (res: seq<T>)
        ensures isSorted(res)
        ensures noDuplicates(res)
        ensures sameContent(res, elems)

     // recursive helper method that returns the values on a given node's
subtree as a sequence
    // in order to ensure the values are sorted, uses Inorder traversal of the
BST
    static method asSeqHelper(node: BSTNode?) returns (res: seq<T>)
        requires node == null || node.Valid()
        decreases if node == null then {} else node.Repr
```

```
        ensures node == null <==> res == []
        ensures node != null ==>
            node.Valid() &&
            node.elems == old(node.elems) &&
            node.Repr == old(node.Repr) &&
            node.value == old(node.value) &&
            sameContent(res, node.elems)
        ensures noDuplicates(res)
        ensures isSorted(res)
}
```

# 2. Implementation and verification

## 2.1. Data representation

As mentioned before, the TreeSet's internal state is represented by a Binary Search Tree. As such, it only requires a reference to the BST's root, which can be null if the tree is empty.

```
    // Concrete state implementation
    var root: BSTNode? // root of the BST representation, may be null
```

As for the BSTNode class, the state is represented by a value and references for the left and right nodes.

```
    // Concrete state implementation
    var value: T // value of this node
    var left: BSTNode?  // elements smaller than 'value' (? - may be null)
    var right: BSTNode? // elements greater than 'value' (? - may be null)
```

The TreeSet's class invariant must ensure:

- If the root is null, *elems* is empty (and vice-versa);
- If the root is not null: the root is Valid, the TreeSet elements are the same as the BST root's elements, the TreeSet's *Repr* variable is consistent with the root's *Repr*.
- *Repr* contains the TreeSet object

```
    // Class invariant with the integrity constraints for the above variables
    predicate Valid()
    {
         this in Repr &&
        (root == null <==> elems == {}) && // null root implies no elements,
and vice versa
        (root != null ==> elems == root.elems && // TreeSet elements must be
the same as the root elements
                    root in Repr && // the root must be in Repr
```

```
                    root.Repr <= Repr && // root Repr must be a subset of the
TreeSet Repr
                    this !in root.Repr && // TreeSet obj must not be in the
root Repr
                    root.Valid() // the BST must always be valid
        )}
```

As for the BSTNode class invariant:

- *Repr* contains the BSTNode object
- The left subtree must have values smaller than the current node's value, and its *Repr* must be consistent.
- The right subtree must have values higher than the current node's value, and its *Repr* must be consistent.
- The current node's *elems* must be equal to the current *value* plus the left *elems* and the right *elems*.

```
// Class invariant with the integrity constraints for the above variables
    predicate Valid()
    {
        this in Repr &&
        (left != null ==>
            left in Repr && left.Repr <= Repr && this !in left.Repr &&
            left.Valid() && (forall v :: v in left.elems ==> v < value)) &&
        (right != null ==>
            right in Repr && right.Repr <= Repr && this !in right.Repr &&
            right.Valid() && (forall v :: v in right.elems ==> value < v)) &&

        (left == null && right == null ==>
            elems == {value} &&
            |elems| == 1
        ) &&
        (left != null && right == null ==>
            elems == left.elems + {value} &&
            |elems| == |left.elems| + 1
        ) &&
        (left == null && right != null ==>
            elems == {value} + right.elems &&
            |elems| == |right.elems| + 1
        ) &&
        (left != null && right != null ==>
            left.Repr !! right.Repr && left.elems !! right.elems && //
disjoint
            elems == left.elems + {value} + right.elems &&
            |elems| == |left.elems| + 1 + |right.elems|
        )}
```

For Dafny to automatically inject the class invariant (and `reads`/`modifies` clauses) in all class operations, we annotate the class headers with the `{:autocontracts}` attribute.

```
class {:autocontracts} TreeSet
class {:autocontracts} BSTNode
```

## 2.2. constructor

The body of the constructors, as well as its verifications, are straightforward.

```
    // TreeSet
    constructor()
        ensures root == null
        ensures elems == {}
        ensures Repr == {this}
    {
        root := null;
        elems := {};
        Repr := {this};
    }
```

```
    // BSTNode
    constructor(x: T)
        ensures value == x && left == null && right == null
        ensures elems == {x}
        ensures Repr == {this}
    {
        value := x;
        left := null;
        right := null;
        elems := {x};
        Repr := {this};
    }
```

## 2.3. contains

The *contains* method is implemented in the TreeSet class by calling the root's *contains* method. It is defined as a function method just to improve its readability.

```
    // check if a given value is in the TreeSet
    function method contains(x: T): bool
        ensures contains(x) <==> x in elems
    {
```

```
        root != null && root.contains(x)
    }
```

The BSTNode *contains* recursively calls itself on the node's children until it gets to the value it is looking for, returning true, or reaches a null node, returning false.

```
    // check if a given value is in the subtree starting in this node (inclusive)
    function method contains(x: T) : bool
        decreases Repr
        ensures contains(x) <==> x in elems
    {
        if x == value then // found value
            true
        else if left != null && x < value then // search in left tree
            left.contains(x)
        else if right != null && x > value then // search in right tree
            right.contains(x)
        else // not found
            false
    }
```

## 2.4. insert

The insert method on the TreeSet is simple, it calls a helper method *insertHelper* which inserts the given value on the BST and returns the updated root. It then takes that value and updates its state.

```
    // insert a value on the TreeSet. If it is repeated it will have no effect.
    method insert(x: T)
        requires x !in elems
        ensures elems == old(elems) + {x}
    {
        // attempt to insert new value and update root
        var newRoot := insertHelper(x, root);
        root := newRoot;

        // make sure TreeSet and BST states are consistent
        elems := root.elems;
        Repr := root.Repr + {this};
    }
```

As for the *insertHelper* method, it takes as arguments: *x*, the value to insert, and *n*, the top node of the current subtree. This is a recursive function and, in order to connect the parent nodes with the newly created node, at each step the method must return a node, *m*.
The base case is *n* being null, when a leaf is reached, where *m* is defined as a new BSTNode and propagated upwards. The recursive step is implemented by checking whether the new value *x* is smaller or larger than the current node's value, and calling *insertHelper* on the corresponding subtree.

```
    // recursive helper method to insert a value 'x' on the BST starting at
node 'n'
    // returns a new node 'm' on whose subtree 'x' exists
    // if x already is in the BST nothing will be changed
    static method insertHelper(x: T, n: BSTNode?) returns (m: BSTNode)
        requires n == null || n.Valid()
        requires n != null ==> x !in n.elems
        modifies if n != null then n.Repr else {}
        ensures m.Valid()
        ensures n == null ==> fresh(m.Repr) && m.elems == {x}
        ensures n != null ==> m == n && n.elems == old(n.elems) + {x}
        ensures n != null ==> fresh(n.Repr - old(n.Repr))
        decreases if n == null then {} else n.Repr
    {
        if n == null { // did not find x, create new node with that value
            m := new BSTNode(x);
        }
        else {
            if x < n.value { // insert x in left subtree and update 'n' values
                n.left := insertHelper(x, n.left);
                n.Repr := n.Repr + n.left.Repr;
            } else { // insert x in right subtree and update 'n' values
                n.right := insertHelper(x, n.right);
                n.Repr := n.Repr + n.right.Repr;
            }

            n.elems := n.elems + {x}; // update elems
               m := n; // return current node 'n' where 'x' has already been
inserted
        }
    }
```

## 2.5. delete

The *delete* method on the TreeSet is simple, it calls the root's *delete* method, which deletes the given value and returns the new root. It then takes that value and updates its state.

```
    // delete a value x from the TreeSet if it exists
    method delete(x: T)
        requires root != null && x in root.elems
        ensures elems == old(elems) - {x}
    {
        // delete value from BST and update the TreeSet's root
        var newRoot := root.delete(x);
        root := newRoot;

        // update state
        elems := if root == null then {} else root.elems;
        Repr := if root == null then {this} else root.Repr + {this};
    }
```

The BSTNode *delete* method takes *x*, the value to delete, and returns the new root. It is called recursively on the tree until it reaches the node with value equal to *x*, at which point it must decide how to delete that node. If either of the children, left or right, is null, the decision is straightforward. However, if both *left* and *right* are BSTNodes, an additional measure is required, for which the method calls the *deleteMin* method.
This method will recursively find the node in that subtree that has the minimal value, which corresponds to the leftmost node. After reaching that node, it returns its values which are then used to update the node that is to be deleted. Overall, this operation corresponds to deleting the node with value *x* and pushing the smallest node in the right tree upwards to its position (reference).

```
    // delete a given value from the subtree starting in this node (inclusive)
    // returns the new root of this subtree
    // https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/
    method delete(x: T) returns (node: BSTNode?)
        requires x in elems
        decreases Repr
        ensures node == null ==> old(elems) <= {x}
        ensures node != null ==> node.Valid()
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {x}
    {
        node := this; // return itself by default

        if left != null && x < value { // value to delete is in left tree
            left := left.delete(x);
            elems := elems - {x};
            if left != null { Repr := Repr + left.Repr; }
```

```
        } else if right != null && value < x { // value to delete is in right
tree
            right := right.delete(x);
            elems := elems - {x};
            if right != null { Repr := Repr + right.Repr; }

        } else if x == value { // the current node is the one to delete
            // if there are no children there is nothing to do
            if left == null && right == null {
                node := null;
            }
            // if left is null, right becomes root
            else if left == null {
                node := right;
            }
            // if right is null, left becomes root
            else if right == null {
                node := left;
            }
            // both childs have values
            else {
                    // find inorder successor of this node and delete it,
returning its values
                var min, r := right.deleteMin();

                // copy contents of the inorder successor to current node
                value := min;
                right := r;

                // update state
                elems := elems - {x};
                if right != null { Repr := Repr + right.Repr; }
            }
        }
    }

    // deletes the min value of a BST node, returning its values
    // since this is a BST, it will be the leftmost node and it is represented
by its value and right subtree
    method deleteMin() returns (min: T, node: BSTNode?)
        decreases Repr
        ensures fresh(Repr - old(Repr))
```

```
        ensures node == null ==> old(elems) == {min}
        ensures node != null ==> node.Valid()
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {min}
         ensures min in old(elems) && (forall x :: x in old(elems) ==> min <=
x)
    {
        if left == null { // this is the minimum node, return values
            min := value;
            node := right;
        } else { // keep searching in left subtree
              min, left := left.deleteMin(); // update 'min' return value and
'left' subtree
            node := this;

            // update state
            elems := elems - {min};
            if left != null { Repr := Repr + left.Repr; }
        }
    }
```

## 2.6. asSeq

The *asSeq* method returns the values of the BST as a sequence, sorted, by calling the *asSeqHelper*. This method takes a node as an argument and recursively collects the values from the BST, using inorder traversal.

```
// return the TreeSet values as a sequence, ordered and without duplicates
    method asSeq() returns (res: seq<T>)
        ensures isSorted(res)
        ensures noDuplicates(res)
        ensures sameContent(res, elems)
    {
        res := asSeqHelper(root);
    }

      // recursive helper method that returns the values on a given node's
subtree as a sequence
    // in order to ensure the values are sorted, uses Inorder traversal of the
BST
    static method asSeqHelper(node: BSTNode?) returns (res: seq<T>)
        requires node == null || node.Valid()
        decreases if node == null then {} else node.Repr
```

```
        ensures node == null <==> res == []
        ensures node != null ==>
            node.Valid() &&
            node.elems == old(node.elems) &&
            node.Repr == old(node.Repr) &&
            node.value == old(node.value) &&
            sameContent(res, node.elems)
        ensures noDuplicates(res)
        ensures isSorted(res)
    {
        if node == null { // base case
            res := [];
        }
        else {
             var leftSeq := asSeqHelper(node.left); // get sequence of left
subtree
             var rightSeq := asSeqHelper(node.right); // get sequence of right
subtree
            res := leftSeq + [node.value] + rightSeq; // inorder traversal
        }
    }
```

# 3. Static testing

For a sanity check of the specification, we write a simple test scenario. This test scenario is successfully checked *statically* by Dafny against the specification previously supplied. This gives some confidence that the operations' post-conditions are correct and complete.

```
method testTreeSet() {
    var s := new TreeSet();

    ghost var s0 := s.asSeq();
    assert s0 == [];

    s.insert(12);
    s.insert(24);
    s.insert(1);

    assert s.contains(1);
    assert s.contains(12);
    assert s.contains(24);
    assert !s.contains(2);
```

```
    assert !s.contains(20);

    assert !s.contains(64);
    s.insert(64);
    assert s.contains(64);
    ghost var s1 := s.asSeq();
    assert s1 == [1,12,24,64];
    s.delete(64);
    assert !s.contains(64);

    ghost var s2 := s.asSeq();
    assert s2 == [1,12,24];
}
```

To check the operations pre-conditions, we write test cases that violate them.

```
method testInvalidDelete() {
    var s := new TreeSet();
    s.delete(1);
}
```

```
method testDuplicateInsert() {
    var s := new TreeSet();
    s.insert(1);
    s.insert(1);
}
```

In both cases, Dafny signals the error “A precondition for this call might not hold”, as expected.

# 4. Putting it all together

```
type T = int // example type, but could be another one

// check if a sequence is sorted
predicate isSorted(s: seq<T>) {
    forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}

// check if a sequence does not have duplicates
predicate noDuplicates(s: seq<T>) {
    forall i, j :: 0 <= i < j < |s| ==> s[i] != s[j]
}
```

```
// check if a sequence and a set have the same content (same elements and
size)
predicate sameContent(s1: seq<T>, s2: set<T>) {
    (forall i :: 0 <= i < |s1| ==> s1[i] in s2) &&
    (forall i :: i in s2 ==> i in s1) &&
    (|s1| == |s2|)
}

// Node of a Binary Search Tree
class {:autocontracts} BSTNode {

    // Concrete state implementation
    var value: T // value of this node
    var left: BSTNode?  // elements smaller than 'value' (? - may be null)
    var right: BSTNode? // elements greater than 'value' (? - may be null)

    // Abstract state used for specification & verification purposes
    ghost var elems: set<T> // Holds the set of values in the subtree starting
in this node (inclusive).
    ghost var Repr: set<object> // Set of objects that are part of the subtree
starting in this node (inclusive)

    // initialize node with given value and no children
    constructor(x: T)
        ensures value == x && left == null && right == null
        ensures elems == {x}
        ensures Repr == {this}
    {
        value := x;
        left := null;
        right := null;
        elems := {x};
        Repr := {this};
    }

    // Class invariant with the integrity constraints for the above variables
    predicate Valid()
    {
        /*
            Must make sure:
```

```
            - Repr is valid: this node is in Repr, as well as the nodes of the
subtree. this node is not in the subtree's Repr.
            - left and right are always Valid
            - all values in left subtree are smaller than 'value'; on right
subtree must be larger
            - elems is equal to {value} + left.elems + right.elems
            - the size of elems must be 1 + |left.elems| + |right.elems|, to
ensure there are no duplicates
            - left.Repr and right.Repr must be disjoint sets (as well as
left.elems and right.elems)
        */
        this in Repr &&

        (left != null ==>
            left in Repr && left.Repr <= Repr && this !in left.Repr &&
            left.Valid() && (forall v :: v in left.elems ==> v < value)) &&

        (right != null ==>
            right in Repr && right.Repr <= Repr && this !in right.Repr &&
            right.Valid() && (forall v :: v in right.elems ==> value < v)) &&

        (left == null && right == null ==>
            elems == {value} &&
            |elems| == 1
        ) &&

        (left != null && right == null ==>
            elems == left.elems + {value} &&
            |elems| == |left.elems| + 1
        ) &&

        (left == null && right != null ==>
            elems == {value} + right.elems &&
            |elems| == |right.elems| + 1
        ) &&

        (left != null && right != null ==>
            left.Repr !! right.Repr && left.elems !! right.elems && //
disjoint
            elems == left.elems + {value} + right.elems &&
            |elems| == |left.elems| + 1 + |right.elems|
        )
```

```
    }

    // check if a given value is in the subtree starting in this node
(inclusive)
    function method contains(x: T) : bool
        decreases Repr
        ensures contains(x) <==> x in elems
    {
        if x == value then // found value
            true
        else if left != null && x < value then // search in left tree
            left.contains(x)
        else if right != null && x > value then // search in right tree
            right.contains(x)
        else // not found
            false
    }

    // delete a given value from the subtree starting in this node (inclusive)
    // returns the new root of this subtree
    // https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/
    method delete(x: T) returns (node: BSTNode?)
        requires x in elems
        decreases Repr
        ensures node == null ==> old(elems) <= {x}
        ensures node != null ==> node.Valid()
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {x}
    {
        node := this; // return itself by default

        if left != null && x < value { // value to delete is in left tree
            left := left.delete(x);
            elems := elems - {x};
            if left != null { Repr := Repr + left.Repr; }

        } else if right != null && value < x { // value to delete is in right
tree
            right := right.delete(x);
            elems := elems - {x};
            if right != null { Repr := Repr + right.Repr; }
```

```
        } else if x == value { // the current node is the one to delete
            // if there are no children there is nothing to do
            if left == null && right == null {
                node := null;
            }
            // if left is null, right becomes root
            else if left == null {
                node := right;
            }
            // if right is null, left becomes root
            else if right == null {
                node := left;
            }
            // both childs have values
            else {
                // find inorder successor of this node and delete it,
returning its values
                var min, r := right.deleteMin();

                // copy contents of the inorder successor to current node
                value := min;
                right := r;

                // update state
                elems := elems - {x};
                if right != null { Repr := Repr + right.Repr; }
            }
        }
    }

    // deletes the min value of a BST node, returning its values
    // since this is a BST, it will be the leftmost node and it is represented
by its value and right subtree
    method deleteMin() returns (min: T, node: BSTNode?)
        decreases Repr
        ensures fresh(Repr - old(Repr))
        ensures node == null ==> old(elems) == {min}
        ensures node != null ==> node.Valid()
        ensures node != null ==> node.Repr <= Repr && node.elems == old(elems)
- {min}
        ensures min in old(elems) && (forall x :: x in old(elems) ==> min <=
x)
```

```
    {
        if left == null { // this is the minimum node, return values
            min := value;
            node := right;
        } else { // keep searching in left subtree
            min, left := left.deleteMin(); // update 'min' return value and
'left' subtree
            node := this;

            // update state
            elems := elems - {min};
            if left != null { Repr := Repr + left.Repr; }
        }
    }
}

// TreeSet class, that represents a sorted collection of type T values,
without duplicates
// Implemented using a BST, allowing most operations to be implemented in
logarithmic time
class {:autocontracts} TreeSet {

    // Concrete state implementation
    var root: BSTNode? // root of the BST representation, may be null

    // Abstract state
    ghost var elems: set<T> // Set of values in the BST
    ghost var Repr: set<object>  // Set of all objects (nodes) that are part
of the BST, including this TreeSet object

    // initialize TreeSet with null root and 0 elements
    constructor()
        ensures root == null
        ensures elems == {}
        ensures Repr == {this}
    {
        root := null;
        elems := {};
        Repr := {this};
    }
```

```
    // Class invariant with the integrity constraints for the above variables
    predicate Valid()
    {
        this in Repr &&
        (root == null <==> elems == {}) && // null root implies no elements,
and vice versa
        (root != null ==> elems == root.elems && // TreeSet elements must be
the same as the root elements
                         root in Repr && // the root must be in Repr
                         root.Repr <= Repr && // root Repr must be a subset
of the TreeSet Repr
                         this !in root.Repr && // TreeSet obj must not be in
the root Repr
                         root.Valid() // the BST must always be valid
        )
    }

    // check if a given value is in the TreeSet
    function method contains(x: T): bool
        ensures contains(x) <==> x in elems
    {
        root != null && root.contains(x)
    }


    // insert a value on the TreeSet. If it is repeated it will have no
effect.
    method insert(x: T)
        requires x !in elems
        ensures elems == old(elems) + {x}
    {
        // attempt to insert new value and update root
        var newRoot := insertHelper(x, root);
        root := newRoot;

        // make sure TreeSet and BST states are consistent
        elems := root.elems;
        Repr := root.Repr + {this};
    }

    // recursive helper method to insert a value 'x' on the BST starting at
node 'n'
```

```
    // returns a new node 'm' on whose subtree 'x' exists
    // if x already is in the BST nothing will be changed
    static method insertHelper(x: T, n: BSTNode?) returns (m: BSTNode)
        requires n == null || n.Valid()
        requires n != null ==> x !in n.elems
        modifies if n != null then n.Repr else {}
        ensures m.Valid()
        ensures n == null ==> fresh(m.Repr) && m.elems == {x}
        ensures n != null ==> m == n && n.elems == old(n.elems) + {x}
        ensures n != null ==> fresh(n.Repr - old(n.Repr))
        decreases if n == null then {} else n.Repr
    {
        if n == null { // did not find x, create new node with that value
            m := new BSTNode(x);
        }
        else {
            if x < n.value { // insert x in left subtree and update 'n' values
                n.left := insertHelper(x, n.left);
                n.Repr := n.Repr + n.left.Repr;
            } else { // insert x in right subtree and update 'n' values
                n.right := insertHelper(x, n.right);
                n.Repr := n.Repr + n.right.Repr;
            }

            n.elems := n.elems + {x}; // update elems
            m := n; // return current node 'n' where 'x' has already been
inserted
        }
    }

    // delete a value x from the TreeSet if it exists
    method delete(x: T)
        requires root != null && x in root.elems
        ensures elems == old(elems) - {x}
    {
        // delete value from BST and update the TreeSet's root
        var newRoot := root.delete(x);
        root := newRoot;

        // update state
        elems := if root == null then {} else root.elems;
        Repr := if root == null then {this} else root.Repr + {this};
```

```
    }

    // return the TreeSet values as a sequence, ordered and without duplicates
    method asSeq() returns (res: seq<T>)
        ensures isSorted(res)
        ensures noDuplicates(res)
        ensures sameContent(res, elems)
    {
        res := asSeqHelper(root);
    }

    // recursive helper method that returns the values on a given node's
subtree as a sequence
    // in order to ensure the values are sorted, uses Inorder traversal of the
BST
    static method asSeqHelper(node: BSTNode?) returns (res: seq<T>)
        requires node == null || node.Valid()
        decreases if node == null then {} else node.Repr
        ensures node == null <==> res == []
        ensures node != null ==>
            node.Valid() &&
            node.elems == old(node.elems) &&
            node.Repr == old(node.Repr) &&
            node.value == old(node.value) &&
            sameContent(res, node.elems)
        ensures noDuplicates(res)
        ensures isSorted(res)
    {
        if node == null { // base case
            res := [];
        }
        else {
            var leftSeq := asSeqHelper(node.left); // get sequence of left
subtree
            var rightSeq := asSeqHelper(node.right); // get sequence of right
subtree
            res := leftSeq + [node.value] + rightSeq; // inorder traversal
        }
    }
}

// Test scenarios
```

```
method testTreeSet() {
    var s := new TreeSet();

    ghost var s0 := s.asSeq();
    assert s0 == [];

    s.insert(12);
    s.insert(24);
    s.insert(1);

    assert s.contains(1);
    assert s.contains(12);
    assert s.contains(24);
    assert !s.contains(2);
    assert !s.contains(20);

    assert !s.contains(64);
    s.insert(64);
    assert s.contains(64);
    ghost var s1 := s.asSeq();
    assert s1 == [1,12,24,64];
    s.delete(64);
    assert !s.contains(64);

    ghost var s2 := s.asSeq();
    assert s2 == [1,12,24];
}
```