Universidade do Porto
Faculdade de Engenharia
FEUP

Development of verified data structures with Dafny: verified implementation of a sorted set using a binary search tree

Eduardo Carreira Ribeiro, December, 2020

MIEIC/MFES

1. Introduction

A **set** is, in computer science, an abstract data type that can **store unique values, without any particular order**. It does not allow duplicates.

A **sorted set, or tree set,** is a set that maintains its **elements in ascending order**, sorted according to the elements' natural ordering or according to a comparator provided at creation time.

A sorted set can be implemented using another data structure: a **binary search tree**. It is a rooted binary tree (meaning that each node has at most 2 child nodes) whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree. Using a binary search tree, we can implement the **basic operations for the sorted set** (`insert`, `delete`, and `contains`) in **logarithmic time complexity (O(log n))**, where n is the number of elements in the BST. We also will provide a way of iterating in order through all of the BST nodes, returning an ordered sequence of elements (the `asSeq` method).

In this report we show how to use Dafny to develop a verified (proved correct) implementation of a sorted set, using a binary search tree.

2. Overview and explanation of concepts

In order to better separate functionality and make the code easier to implement and verify, the implementation of the sorted set was divided in two classes, one being the `BSTNode` class, that represents a node of a binary search tree, that contains its value, its left child and its right child, if they exist. This class can then be used to create a functioning binary search tree that can be used to make the sorted set. The other class is the `TreeSet` class, that contains the root of the BST that implements the set, and offers some methods and functions that allow interaction with the set and its elements.

We start by specifying in Dafny the type parameter that is going to be the type of the elements in our data structures. Due to some limitations of generics in Dafny, the type parameter (`T`) is instantiated for a concrete type.

Some common use helper functions were also created to help in the verification of some other methods. These functions are pretty self-explanatory: the `isMin()` function receives an element and a set, and checks if that value is the minimum value in the set; the `isSorted()` function receives a sequence and checks if it is in ascending order, `noDuplicates()` checks if a sequence has no duplicates, and finally `checkEqual()`, that receives a sequence and a set and checks if they both have the same content.

```
type T = int // for demo purposes, but could be real, etc.

// helper function to check if a value is the smallest element in a set
predicate isMin(x: T, arr: set<T>)
{
 forall elem :: elem in arr ==> x <= elem
}

// helper function to check if a sequence is in strictly increasing order
predicate isSorted(arr: seq<T>)
{
 forall k1, k2 :: 0 <= k1 < k2 < |arr| ==> arr[k1] <= arr[k2]
}

// helper function that indicates if a sequence has duplicate elements or not
predicate noDuplicates(arr: seq<T>) {
 forall k1, k2 :: 0 <= k1 < k2 < |arr| ==> arr[k1] != arr[k2]
}

// helper function that checks if the contents of a given sequence are the same as
the elements of a given set
predicate checkEqual(arr1: seq<T>, arr2: set<T>) {
 // check if same length
 (|arr1| == |arr2|) &&
 // every element of the seq appears on the set
 (forall i :: 0 <= i < |arr1| ==> arr1[i] in arr2) &&
```

```
 // every element of the set appears on the seq
 (forall elem :: elem in arr2 ==> elem in arr1)
}
```

In Dafny, in order to verify the correctness of class methods, we need to specify their contracts (pre- and post-conditions), and define the class invariant. The class invariant (specified by the predicate `Valid()`) is a constraint that defines the valid states of instances of the class, and it needs to hold before and after the execution of any class method (and after the execution of the class constructor). The Dafny keyword `autocontracts` is used to automatically add the verification of the `Valid()` predicate (the class invariant) to the pre- and post-conditions of methods and constructors.

However, something that we should have in mind when specifying these pre- and post-conditions is that the contracts of public operations of a class should not expose the internal state representation to their clients. Contracts should refer to abstract (public) state representations, instead of concrete (private) state representations. In order to avoid this, we used what is called state abstraction; in addition to specifying the concrete fields and internal state of the class, we also create ghost/abstract fields that are related to the concrete ones, and we use these abstract fields in the method contracts. These ghost variables are used for verification purposes only, and do not go into executable code.

3. Implementation and verification

3.1. Data representation

Starting with the `BSTNode` class, we can see that its internal state representation is composed of 3 fields: the `value` of the node, the `left` child of the node, and its `right` child. The child nodes are of type `BSTNode?` (notice the `?`), to indicate that the fields might be null, in the case that the node does not have a left or right child. As mentioned earlier, each class also needs to have some abstract fields to be used in the static verification of its methods, in order to "hide" internal state representations from external users and clients. For that, we have the `elems` field, a set that represents all elements that are in the BST which root is the current node, and the special `Repr` field, a set of objects that contains all node objects that are in the BST (note that all classes in Dafny extend this `object` class). These abstract variables are also updated accordingly in each method/function, so their content can reflect the content of the actual concrete variables of the class. The constructor of the class receives the node value and creates a node with no children.

```
// Represents a binary search tree node, for elements of type T.
class {:autocontracts} BSTNode {
 // ghost variable representing a set of all the elems that are in the BST which
root is this node
 ghost var elems: set<T>;
```

```
 // ghost variable representing a set of all the node objects that are in the BST
which root is this node
 ghost var Repr: set<object>;

 // value of the node
 var value: T;

 // left child of the node (? - may be null)
 var left: BSTNode?;

 // right child of the node (? - may be null)
 var right: BSTNode?;

 // constructor of the class. Creates a single node with no children
 constructor (data: T)
   ensures elems == {data}
   ensures Repr == {this}
 {
   value := data;
   left := null;
   right := null;
   elems := {data};
   Repr := {this};
 }
```

In the `TreeSet` class, we also have as abstract fields the `elems` and `Repr` variables, that represent the elements and objects in the set, and as a concrete variable the `root` node of the BST that implements this sorted set. The constructor of the class creates an empty set.

```
// Represents a sorted set of elements of type T. Implemented with a binary search
tree
class {:autocontracts} TreeSet {
 // ghost variable representing a set of all the elems that are in the set
 ghost var elems: set<T>;

  // ghost variable representing a set of all the objects that are in the set
(including the set itself)
 ghost var Repr: set<object>;

 // variable that represents the root of the BST that implements this sorted set
 var root: BSTNode?

 // constructor of the class. Creates an empty set
 constructor()
   ensures elems == {}
   ensures Repr == {this}
```

```
{
  root := null;
  Repr := {this};
  elems := {};
}
```

3.2. Class invariants (Valid())

The class invariant of the `BSTNode` class needs to enforce some constraints, some related to the ordering of its elements. The `Valid()` predicate is also responsible for "connecting" the internal state representation with the abstract variables of the class, and ensuring that the state of the concrete variables at a given time is always reflected in the ghost fields that are used for static verification. The class invariant checks, among other things, if the left and right children are valid (if they exist), that the elements contained in the left subtree are smaller than the current node, and that the elements in the right subtree are higher. A more in-depth explanation of the predicate can be found in the code comments.

```
// class invariant of the BSTNode class
predicate Valid()
{
  // check if this node is in Repr
  this in Repr &&

  // check correctness for left subtree:
  // check if Repr contains left and all nodes of the left subtree
  // check if left's Repr does not contain the current node, to avoid cycles
  // check class invariant for left node
  // check if all nodes on left subtree are less than the current node
  (left != null ==>
    left in Repr &&
    left.Repr <= Repr && this !in left.Repr &&
    left.Valid() &&
    (forall elem :: elem in left.elems ==> elem < value)) &&

  // check correctness for right subtree:
  // check if Repr contains right and all nodes of the right subtree
  // check if right's Repr does not contain the current node, to avoid cycles
  // check class invariant for right node
  // check if all nodes on right subtree are higher than the current node
  (right != null ==>
    right in Repr &&
    right.Repr <= Repr && this !in right.Repr &&
    right.Valid() &&
    (forall elem :: elem in right.elems ==> value < elem)) &&
```

```
  // case where the node does not have any subtrees:
  // connect ghost variable elems with its intended value
  (left == null && right == null ==>
    elems == {value}) &&

  // case where only left subtree exists:
  // connect ghost variable elems with its intended value
  (left != null && right == null ==>
    elems == left.elems + {value}) &&

  // case where only right subtree exists:
  // connect ghost variable elems with its intended value
  (left == null && right != null ==>
    elems == {value} + right.elems) &&

  // case where both left and right subtrees exist:
  // check if left and right subtrees are disjoint (i.e. no node belongs to both)
  // connect ghost variable elems with its intended value
  (left != null && right != null ==>
    left.Repr !! right.Repr &&
    elems == left.elems + {value} + right.elems)
}
```

The class invariant for the `TreeSet` class essentially checks if the root node of the BST is in a valid state, and also connects its abstract variables with the internal state.

```
// class invariant of the TreeSet class
predicate Valid()
{
  this in Repr &&
  // if the root is null, then the set must be empty, and vice-versa
  (root == null <==> elems == {}) &&
  // if the root exists...
  (root != null ==>
    root in Repr && root.Repr <= Repr && this !in root.Repr &&
     // check root's class invariant and connect ghost variable with its intended
value
    root.Valid() &&
    elems == root.elems)
}
```

To end this section, just a reminder that these `Valid()` predicates are automatically verified and enforced as pre- and post conditions of all methods (and as post-conditions of all constructors), because of the `autocontracts` keyword.

3.3. insert() method

The `insert()` method calls the static method `insertAux()` to try to insert the wanted value in the BST, returning the new root of the tree.

```
// method that inserts a new value into the set
// creates a new node and inserts it in the BST
method insert(x: T)
  requires x !in elems
  ensures fresh(Repr - old(Repr))
  ensures elems == old(elems) + {x}
{
  // insert value using helper method; returns new root of BST
  var new_root := insertAux(x, root);
  // update BST root
  root := new_root;
  // update ghost variables
  elems := root.elems;
  Repr := root.Repr + {this};
}
```

To understand the `insertAux()` method, we need to understand how the insertion operation is typically done in a BST. A new node is always inserted at a leaf, i.e. it is not going to be inserted in the "middle" of a tree and it is not going to have any children. 1 out of 4 scenarios can pan out:

- if the value we want to insert is equal to the current node's value, then it is already present in the BST and we don't need to do anything;
- if the value is smaller than the current node, we should try to insert it in the left subtree. So we recursively do the insert operation on the root of the left subtree, if it exists;
- similarly, if the value is higher, we try to insert it in the right subtree, if it exists;
- if the current node is null/empty, then we have found a place where we can insert the value, so we create a new node. If that value is smaller than the value of the previous node, then the new node is going to be the left child of it, otherwise it becomes the right child.

The code of the `insertAux()` method is the following:

```
 // helper method for inserting a new value in the sorted set
 static method insertAux(x: T, n: BSTNode?) returns (new_root: BSTNode)
   requires n == null || n.Valid()
   modifies if n != null then n.Repr else {}
   ensures new_root.Valid()
   ensures n == null ==> fresh(new_root) && fresh(new_root.Repr) && new_root.elems
== {x}
   ensures n != null ==> n == new_root && fresh(n.Repr - old(n.Repr)) && n.elems ==
old(n.elems) + {x}
   decreases if n == null then {} else n.Repr
 {
   // if the current node is null, we can insert the element here
   if n == null {
     // just need to create a new node, that is the root of its BST
     new_root := new BSTNode(x);
   }
   // new value is equal to the current node's value; don't change anything
   else if x == n.value {
     new_root := n;
   }
   else {
      // new value is less than current node's value; should be inserted in the left
subtree
     if x < n.value {
       // insert in left subtree, get new root of left subtree
       var new_lr := insertAux(x, n.left);
       // update new root and ghost variable
       n.left := new_lr;
       n.Repr := n.Repr + n.left.Repr;
     }
      // new value is higher than current node's value; should be inserted in the
right subtree
     else {
       // insert in right subtree, get new root of right subtree
       var new_rr := insertAux(x, n.right);
       // update new root and ghost variable
       n.right := new_rr;
       n.Repr := n.Repr + n.right.Repr;
     }

     // BST root already existed, so root didn't change
     new_root := n;
     // add newly added element to the root's elems
```

```
    n.elems := n.elems + {x};
  }
}
```

3.4. delete() method

The deletion of an element in the sorted set is done using the `delete()` method in the `TreeSet` class. This method calls the `delete()` method of the root node of the BST (of the class `BSTNode`), which returns the new root of the tree. The root is then updated, alongside with the ghost variables.

```
// method that deletes a value from the sorted set
method delete(x: T)
  requires x in elems
  ensures fresh(Repr - old(Repr))
  ensures elems == old(elems) - {x}
{
  if root != null {
    // call delete() method from BST root, to delete the value and get new root
    var new_root := root.delete(x);
    // update BST root
    root := new_root;
    // if the new root exists, update ghost variables accordingly
    if root != null {
      elems := root.elems;
      Repr := root.Repr + {this};
    }
    // root is null; set/BST is now empty. Reset the ghost variables
    else {
      elems := {};
      Repr := {this};
    }
  }
}
```

The code for the `delete()` method in the `BSTNode` class is the following:

```
// method that deletes a value from the BST
// returns the new root of the tree (null if the last node of the BST has been
deleted)
method delete(x: T) returns (node: BSTNode?)
  requires x in elems
  decreases Repr
  ensures fresh(Repr - old(Repr))
```

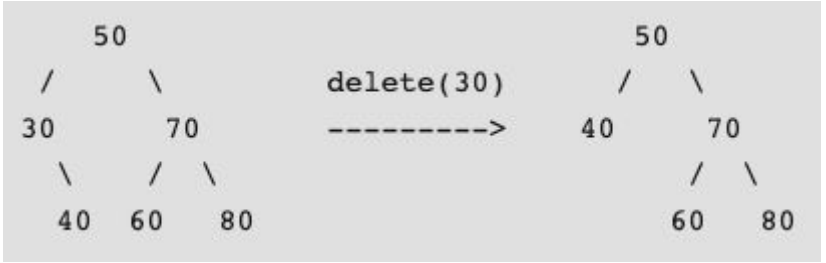
```
   ensures node == null ==> old(elems) <= {x}
    ensures node != null ==> node.Valid() && node.Repr <= Repr && node.elems ==
old(elems) - {x}
 {
   node := this;
   // if the 'x' value is less than the current value, try deleting in left subtree
   if left != null && x < value {
     // left.delete() will return the left subtree's new root
     var t := left.delete(x);
     // update left subtree's root
     left := t;
     // remove element from ghost variable
     elems := elems - {x};
     // update Repr variable
     if left != null { Repr := Repr + left.Repr; }
   }

    // if the 'x' value is higher than the current value, try deleting in right
subtree
   else if right != null && value < x {
     // right.delete() will return the right subtree's new root
     var t := right.delete(x);
     // update right subtree's root
     right := t;
     // remove element from ghost variable
     elems := elems - {x};
     // update Repr variable
     if right != null { Repr := Repr + right.Repr; }
   }

   // if the element that we want to delete is the current node (root),
   // then we need to get a new root for the tree
   else if x == value {
     // if this node is the last one in the tree, there are no nodes left
     if left == null && right == null {
       node := null;
     }
     // if there is no left subtree, new root is the root of the right subtree
     else if left == null {
       node := right;
     }
     // if there is no right subtree, new root is the root of the left subtree
     else if right == null {
       node := left;
     }
```

```
    // if both subtrees exist, new root is the inorder successor of the node
    else {
      // find in order successor of the node: the leftmost (smaller) element
      // in the right subtree
      var min_val, new_r := right.deleteMin();
      // update node's value
      value := min_val;
      // update node's right subtree root
      right := new_r;
      // update ghost variable
      elems := elems - {x};
      // update Repr variable
      if right != null { Repr := Repr + right.Repr; }
    }
  }
}
```

The method works similarly as other recursive operations in a BST: if the current value is smaller, do it in the left subtree; if it is bigger, try to delete in the right subtree. If the current value is equal to the value that we want to delete, then we delete that node. The tricky part is to restructure the tree in order to replace the deleted node with another node from the BST (if there is one). 1 out of 3 scenarios can occur, when deleting an element from a BST:

- the node to be deleted is a leaf. This is the easiest scenario, because we don't need to do anything else; we just delete the node and that's it.

- the node to be deleted has one child. In this case, just replace the deleted node with the child.



- the node to be deleted has a left and a right child. We need to replace the node with its inorder successor, i.e. the first node on the BST that has a higher value than that node, i.e. the leftmost element in the right subtree (the inorder predecessor can also be used, but in this implementation we use the successor).

To find the inorder successor, we use the `deleteMin()` method, that returns the leftmost/minimum element of the tree, but we call the method on the right subtree, thus getting the inorder successor of the node.

```
 // method that removes and returns the smaller element of a BST
 // used by the delete() method when it needs to find the inorder successor
 // of a node (i.e. the smallest element in the right subtree)
 // returns the value of the smallest element, and the new root of the tree
 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() && node.Repr <= Repr && node.elems ==
old(elems) - {min}
   ensures min in old(elems) && isMin(min, old(elems))
 {
   // the minimum element is the leftmost element in the left subtree
   // if the current node does not have a left child, then it is the min value
   if left == null {
     // update values to return
     min := value;
     node := right;
   }
    // node has left child; it is not the minimum. Keep iterating through left
subtree
   else {
     var new_lr;
     // delete the minimum in left subtree
     min, new_lr := left.deleteMin();
     // update new left subtree root
     left := new_lr;
     // root of the tree starting at this node is still this node
     node := this;
```

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

3.5. contains() method

In order to check if a given value is contained in the sorted set, the `TreeSet` class offers a `contains()` method that checks if the value is present in the underlying BST.

```
// method that checks if a given value is present in the set
// checks if it is present in the binary search tree
function method contains(x: T) : bool
  ensures contains(x) <==> x in elems
{
  root != null && root.contains(x)
}
```

To check if a value is in a binary search tree, we once again can take advantage of the fact that the elements in a BST are ordered. If we have the root of the BST, 1 out of 4 options will occur:

- if the node's value is equal to the element we are looking for, then we have found it and can conclude that the value is indeed contained in the BST;
- if the node's value is less than the element, we know that if the element is contained in the BST, it's going to be in the right subtree. So we can recursively call the `contains()` method on the root of the right subtree;
- similarly, if the node's value is higher, we call recursively on the root of the left subtree;
- if we need to call recursively but the needed subtree does not exist, we can conclude that the element is not contained in the BST.

This logic is implemented in the `contains()` method present in the `BSTNode` class.

```
// method that checks if a given value is present in the BST
function method contains(x: T) : bool
  ensures contains(x) <==> x in elems
{
  // if value is equal to the root, return true
  if x == value then true

  // if the value is smaller, check on left subtree
  else if left != null && x < value then left.contains(x)
```

```
  // if the value is higher, check on right subtree
  else if right != null && x > value then right.contains(x)

  // value is not present in the BST
  else false
}
```

3.6. asSeq() method

This method returns a strictly ascending sequence of all nodes/values in the sorted set. In the `TreeSet` class, that is done by generating an in-order sequence of all BST elements, starting at the root.

```
// method that returns an ordered sequence of all elements in the sorted set
method asSeq() returns (res: seq<T>)
    ensures isSorted(res)
    ensures noDuplicates(res)
    ensures checkEqual(res, elems)
{
  if root == null {
    // if root is null then sequence should be empty
    res := [];
  }
  else {
    // else, return the inorder sequence of all BST elements from the root
    res := root.inorderSeq();
  }
}
```

Switching over to the `BSTNode` class, the `inorderSeq()` method starts by recursively generating the in-order sequence of left and right subtrees (if they exist). Assuming these sequences are valid in-order sequences, we know that all values in the left sequence are smaller than the current value, and all elements in the right sequence are higher. Thus, the in-order sequence starting at the root node can be generated by appending the root node to the end of the left sequence, and appending the right sequence to the end of the result.

We can also see that the post-conditions of this method verify and enforce all properties that we expect from an in-order sequence: the sequence needs to be sorted, containing no duplicates, and its content needs to be exactly the same as the content of the BST.

```
// method that returns the inorder sequence of all the elements of the BST
 // inorder sequence of a binary tree:
 // inorder sequence of the left subtree + root + inorder sequence of the right
subtree
method inorderSeq() returns (res: seq<T>)
```

```
  decreases Repr
  ensures isSorted(res)
  ensures noDuplicates(res)
  ensures checkEqual(res, elems)
{
  // inorder sequence of the BST which root is the left child of the current node
  var left_seq : seq<T> := [];

  // inorder sequence of the BST which root is the right child of the current node
  var right_seq : seq<T> := [];

  if left != null {
    // if left subtree is not null, calculate the left subsequence
    left_seq := left.inorderSeq();
  }
  if right != null {
    // if right subtree is not null, calculate the right subsequence
    right_seq := right.inorderSeq();
  }

  // return the inorder sequence
  res := left_seq + [value] + right_seq;
}
```

3.7. isEmpty() method

Besides all the methods previously explained, a simple `isEmpty()` method was created, that checks if the sorted set contains any elements or not. That is calculated by simply checking if the BST that implements the sorted set is empty or not, that is, if the root of the BST exists.

```
// simple method that checks if the set is empty or not
predicate method isEmpty()
  ensures isEmpty() <==> (|elems| == 0)
{
  root == null
}
```

4. Static Testing

This section shows some test cases that were written in order to check if Dafny could verify the correctness of the use of the `TreeSet` method calls in various contexts. These tests are checked *statically* by Dafny against the specification previously supplied. This gives some confidence that the post-conditions specified for the operations are correct and complete.

The first test method briefly covers the `insert()`, `delete()` and `contains()` methods, when used properly and correctly. The test first creates a `TreeSet` instance and checks if it is empty, then inserts some elements and checks the set content, and finally deletes all elements. This test is successfully checked and passed by Dafny.

```
// A test case, checked statically by Dafny, that covers the
// insert(), delete() and contains() methods of the TreeSet class
method testTreeSetInsertionDeletion()
{
 // create new set
 var s := new TreeSet();
 // check that set is empty
 assert s.isEmpty();

 // insert two elements
 s.insert(2);
 s.insert(9);
 // check if the set is not empty anymore
 assert !s.isEmpty();
 // check that, for any contains() call that we do on the TreeSet,
 // it will only be true if the argument of the call is either 2 or 9,
 // because they are the only elements on the set
 var x;
 var isPresentInSet := s.contains(x);
 assert isPresentInSet <==> x == 2 || x == 9;

 // delete element
 s.delete(9);
  // add new element
 s.insert(4);

 // check that the set is not empty; still some elements left
 assert !s.isEmpty();
 var y;
 var isPresentInSet2 := s.contains(y);
 assert isPresentInSet2 <==> y == 2 || y == 4;

 // delete rest of the elements
 s.delete(2);
 s.delete(4);

 // check that the elements that were deleted are no longer in the set
 assert !s.contains(2);
 assert !s.contains(4);
 assert !s.contains(9);
```

```
 // check that set is empty again
 assert s.isEmpty();
}
```

This next test covers the `asSeq()` method and tests it in different scenarios, when the set is empty, after inserting some elements, etc. Once again, Dafny validates this test.

```
// A test case, checked statically by Dafny, that covers the
// isSeq() method of the TreeSet class
method testTreeSetSequence()
{
 // create new set
 var s := new TreeSet();
  // check that the sequence that the asSeq() method returns is empty
 var sequence := s.asSeq();
 assert sequence == [];

 // insert a bunch of elements
 s.insert(2);
 s.insert(10);
 s.insert(8);

 // check that the returned sequence has the same elements and is sorted
 sequence := s.asSeq();
 assert sequence == [2, 8, 10];

 // delete some of the elements
 s.delete(8);

 // check the sequence of elements
 sequence := s.asSeq();
 assert sequence == [2, 10];

 // delete the rest of the elements
 s.delete(2);
 s.delete(10);

 // check that the set is now completely empty
 assert s.isEmpty();

 // check that the sequence that the asSeq() method returns is now empty
 sequence := s.asSeq();
 assert sequence == [];
}
```

In this next test, it is incorrectly attempted to delete an element from the set that does not exist. This should be statically identified by Dafny and it should warn the user that the operation is invalid, because one of the pre-conditions for the `delete()` method is that the element to be deleted needs to be in the set, which is not the case. As predicted, Dafny alerts the user that a pre-condition might not hold, when calling the `delete()` method.

```
// A test case, checked statically by Dafny, that should fail
// because of incorrect call to the TreeSet delete() method
method testTreeSetFailDelete()
{
 // create new set
 var s := new TreeSet();

 // insert some elements
 s.insert(2);
 s.insert(4);
 s.insert(6);

 // this method call fails the static analysis done by Dafny, because
 // a precondition for this method is not verified: the number 7 is not
 // on the set, so it cannot be deleted
 s.delete(7);
}
```

Similarly, this next and last test case checks if Dafny can detect a wrong call to the `insert()` method, when we try to insert an element into the set that is already there.

```
// A test case, checked statically by Dafny, that should fail
// because of incorrect call to the TreeSet insert() method
method testTreeSetFailInsert()
{
 // create new set
 var s := new TreeSet();

 // insert some elements
 s.insert(2);
 s.insert(4);
 s.insert(6);

 // this one should also fail the static analysis, once again because it
 // does not satisfy the precondition that an element that is already
 // on the set cannot be inserted again
 s.insert(2);
}
```

5. Putting it all together

This is the final code, with all the helper functions, classes and tests that are part of this sorted set implementation:

```
type T = int // for demo purposes, but could be real, etc.

// helper function to check if a value is the smallest element in a set
predicate isMin(x: T, arr: set<T>)
{
 forall elem :: elem in arr ==> x <= elem
}

// helper function to check if a sequence is in strictly increasing order
predicate isSorted(arr: seq<T>)
{
 forall k1, k2 :: 0 <= k1 < k2 < |arr| ==> arr[k1] <= arr[k2]
}

// helper function that indicates if a sequence has duplicate elements or not
predicate noDuplicates(arr: seq<T>) {
 forall k1, k2 :: 0 <= k1 < k2 < |arr| ==> arr[k1] != arr[k2]
}

// helper function that checks if the contents of a given sequence are the same as
the elements of a given set
predicate checkEqual(arr1: seq<T>, arr2: set<T>) {
 // check if same length
 (|arr1| == |arr2|) &&
 // every element of the seq appears on the set
 (forall i :: 0 <= i < |arr1| ==> arr1[i] in arr2) &&
 // every element of the set appears on the seq
 (forall elem :: elem in arr2 ==> elem in arr1)
}

// ***********************************************************************
// Represents a binary search tree node, for elements of type T.
class {:autocontracts} BSTNode {
 // ghost variable representing a set of all the elems that are in the BST which
root is this node
 ghost var elems: set<T>;

 // ghost variable representing a set of all the node objects that are in the BST
which root is this node
 ghost var Repr: set<object>;
```

```
// value of the node
var value: T;

// left child of the node (? - may be null)
var left: BSTNode?;

// right child of the node (? - may be null)
var right: BSTNode?;

// constructor of the class. Creates a single node with no children
constructor (data: T)
  ensures elems == {data}
  ensures Repr == {this}
{
  value := data;
  left := null;
  right := null;
  elems := {data};
  Repr := {this};
}

// class invariant of the BSTNode class
predicate Valid()
{
  // check if this node is in Repr
  this in Repr &&

  // check correctness for left subtree:
  // check if Repr contains left and all nodes of the left subtree
  // check if left's Repr does not contain the current node, to avoid cycles
  // check class invariant for left node
  // check if all nodes on left subtree are less than the current node
  (left != null ==>
    left in Repr &&
    left.Repr <= Repr && this !in left.Repr &&
    left.Valid() &&
    (forall elem :: elem in left.elems ==> elem < value)) &&

  // check correctness for right subtree:
  // check if Repr contains right and all nodes of the right subtree
  // check if right's Repr does not contain the current node, to avoid cycles
  // check class invariant for right node
  // check if all nodes on right subtree are higher than the current node
  (right != null ==>
```

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

  // case where the node does not have any subtrees:
  // connect ghost variable elems with its intended value
  (left == null && right == null ==>
    elems == {value}) &&

  // case where only left subtree exists:
  // connect ghost variable elems with its intended value
  (left != null && right == null ==>
    elems == left.elems + {value}) &&

  // case where only right subtree exists:
  // connect ghost variable elems with its intended value
  (left == null && right != null ==>
    elems == {value} + right.elems) &&

  // case where both left and right subtrees exist:
  // check if left and right subtrees are disjoint (i.e. no node belongs to both)
  // connect ghost variable elems with its intended value
  (left != null && right != null ==>
    left.Repr !! right.Repr &&
    elems == left.elems + {value} + right.elems)
}

// method that checks if a given value is present in the BST
function method contains(x: T) : bool
  ensures contains(x) <==> x in elems
{
  // if value is equal to the root, return true
  if x == value then true

  // if the value is smaller, check on left subtree
  else if left != null && x < value then left.contains(x)

  // if the value is higher, check on right subtree
  else if right != null && x > value then right.contains(x)

  // value is not present in the BST
  else false
}
```

```
 // method that deletes a value from the BST
 // returns the new root of the tree (null if the last node of the BST has been deleted)
 method delete(x: T) returns (node: BSTNode?)
   requires x in elems
   decreases Repr
   ensures fresh(Repr - old(Repr))
   ensures node == null ==> old(elems) <= {x}
    ensures node != null ==> node.Valid() && node.Repr <= Repr && node.elems == old(elems) - {x}
 {
   node := this;
   // if the 'x' value is less than the current value, try deleting in left subtree
   if left != null && x < value {
     // left.delete() will return the left subtree's new root
     var t := left.delete(x);
     // update left subtree's root
     left := t;
     // remove element from ghost variable
     elems := elems - {x};
     // update Repr variable
     if left != null { Repr := Repr + left.Repr; }
   }

    // if the 'x' value is higher than the current value, try deleting in right subtree
   else if right != null && value < x {
     // right.delete() will return the right subtree's new root
     var t := right.delete(x);
     // update right subtree's root
     right := t;
     // remove element from ghost variable
     elems := elems - {x};
     // update Repr variable
     if right != null { Repr := Repr + right.Repr; }
   }

   // if the element that we want to delete is the current node (root),
   // then we need to get a new root for the tree
   else if x == value {
     // if this node is the last one in the tree, there are no nodes left
     if left == null && right == null {
       node := null;
     }
     // if there is no left subtree, new root is the root of the right subtree
```

```
    else if left == null {
      node := right;
    }
    // if there is no right subtree, new root is the root of the left subtree
    else if right == null {
      node := left;
    }
    // if both subtrees exist, new root is the inorder successor of the node
    else {
      // find in order successor of the node: the leftmost (smaller) element
      // in the right subtree
      var min_val, new_r := right.deleteMin();
      // update node's value
      value := min_val;
      // update node's right subtree root
      right := new_r;
      // update ghost variable
      elems := elems - {x};
      // update Repr variable
      if right != null { Repr := Repr + right.Repr; }
    }
  }
}

// method that removes and returns the smaller element of a BST
// used by the delete() method when it needs to find the inorder successor
// of a node (i.e. the smallest element in the right subtree)
// returns the value of the smallest element, and the new root of the tree
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() && node.Repr <= Repr && node.elems ==
old(elems) - {min}
  ensures min in old(elems) && isMin(min, old(elems))
{
  // the minimum element is the leftmost element in the left subtree
  // if the current node does not have a left child, then it is the min value
  if left == null {
    // update values to return
    min := value;
    node := right;
  }
  // node has left child; it is not the minimum. Keep iterating through left
subtree
```

```
    else {
      var new_lr;
      // delete the minimum in left subtree
      min, new_lr := left.deleteMin();
      // update new left subtree root
      left := new_lr;
      // root of the tree starting at this node is still this node
      node := this;
      // update ghost variables
      elems := elems - {min};
      if left != null { Repr := Repr + left.Repr; }
    }
  }

  // method that returns the inorder sequence of all the elements of the BST
  // inorder sequence of a binary tree:
  // inorder sequence of the left subtree + root + inorder sequence of the right
subtree
  method inorderSeq() returns (res: seq<T>)
    decreases Repr
    ensures isSorted(res)
    ensures noDuplicates(res)
    ensures checkEqual(res, elems)
  {
    // inorder sequence of the BST which root is the left child of the current node
    var left_seq : seq<T> := [];

    // inorder sequence of the BST which root is the right child of the current node
    var right_seq : seq<T> := [];

    if left != null {
      // if left subtree is not null, calculate the left subsequence
      left_seq := left.inorderSeq();
    }
    if right != null {
      // if right subtree is not null, calculate the right subsequence
      right_seq := right.inorderSeq();
    }

    // return the inorder sequence
    res := left_seq + [value] + right_seq;
  }
}

// *****************************************************************
```

```
// Represents a sorted set of elements of type T. Implemented with a binary search tree
class {:autocontracts} TreeSet {
 // ghost variable representing a set of all the elems that are in the set
 ghost var elems: set<T>;

 // ghost variable representing a set of all the objects that are in the set (including the set itself)
 ghost var Repr: set<object>;

 // variable that represents the root of the BST that implements this sorted set
 var root: BSTNode?

 // constructor of the class. Creates an empty set
 constructor()
   ensures elems == {}
   ensures Repr == {this}
 {
   root := null;
   Repr := {this};
   elems := {};
 }

 // class invariant of the TreeSet class
 predicate Valid()
 {
   this in Repr &&
   // if the root is null, then the set must be empty, and vice-versa
   (root == null <==> elems == {}) &&
   // if the root exists...
   (root != null ==>
     root in Repr && root.Repr <= Repr && this !in root.Repr &&
      // check root's class invariant and connect ghost variable with its intended value
     root.Valid() &&
     elems == root.elems)
 }

 // method that checks if a given value is present in the set
 // checks if it is present in the binary search tree
 function method contains(x: T) : bool
   ensures contains(x) <==> x in elems
 {
   root != null && root.contains(x)
 }
```

```
 // method that inserts a new value into the set
 // creates a new node and inserts it in the BST
 method insert(x: T)
   requires x !in elems
   ensures fresh(Repr - old(Repr))
   ensures elems == old(elems) + {x}
 {
   // insert value using helper method; returns new root of BST
   var new_root := insertAux(x, root);
   // update BST root
   root := new_root;
   // update ghost variables
   elems := root.elems;
   Repr := root.Repr + {this};
 }

 // helper method for inserting a new value in the sorted set
 static method insertAux(x: T, n: BSTNode?) returns (new_root: BSTNode)
   requires n == null || n.Valid()
   modifies if n != null then n.Repr else {}
   ensures new_root.Valid()
    ensures n == null ==> fresh(new_root) && fresh(new_root.Repr) && new_root.elems
== {x}
    ensures n != null ==> n == new_root && fresh(n.Repr - old(n.Repr)) && n.elems ==
old(n.elems) + {x}
   decreases if n == null then {} else n.Repr
 {
   // if the current node is null, we can insert the element here
   if n == null {
     // just need to create a new node, that is the root of its BST
     new_root := new BSTNode(x);
   }
   // new value is equal to the current node's value; don't change anything
   else if x == n.value {
     new_root := n;
   }
   else {
      // new value is less than current node's value; should be inserted in the left
subtree
     if x < n.value {
       // insert in left subtree, get new root of left subtree
       var new_lr := insertAux(x, n.left);
       // update new root and ghost variable
       n.left := new_lr;
```

```
      n.Repr := n.Repr + n.left.Repr;
    }
     // new value is higher than current node's value; should be inserted in the
right subtree
    else {
      // insert in right subtree, get new root of right subtree
      var new_rr := insertAux(x, n.right);
      // update new root and ghost variable
      n.right := new_rr;
      n.Repr := n.Repr + n.right.Repr;
    }

    // BST root already existed, so root didn't change
    new_root := n;
    // add newly added element to the root's elems
    n.elems := n.elems + {x};
  }
}

// method that deletes a value from the sorted set
method delete(x: T)
  requires x in elems
  ensures fresh(Repr - old(Repr))
  ensures elems == old(elems) - {x}
{
  if root != null {
    // call delete() method from BST root, to delete the value and get new root
    var new_root := root.delete(x);
    // update BST root
    root := new_root;
    // if the new root exists, update ghost variables accordingly
    if root != null {
      elems := root.elems;
      Repr := root.Repr + {this};
    }
    // root is null; set/BST is now empty. Reset the ghost variables
    else {
      elems := {};
      Repr := {this};
    }
  }
}

// method that returns an ordered sequence of all elements in the sorted set
method asSeq() returns (res: seq<T>)
```

```
    ensures isSorted(res)
    ensures noDuplicates(res)
    ensures checkEqual(res, elems)
 {
   if root == null {
     // if root is null then sequence should be empty
     res := [];
   }
   else {
     // else, return the inorder sequence of all BST elements from the root
     res := root.inorderSeq();
   }
 }

 // simple method that checks if the set is empty or not
 predicate method isEmpty()
   ensures isEmpty() <==> (|elems| == 0)
 {
   root == null
 }
}

// ***************************************************************
// A test case, checked statically by Dafny, that covers the
// insert(), delete() and contains() methods of the TreeSet class
method testTreeSetInsertionDeletion()
{
 // create new set
 var s := new TreeSet();
 // check that set is empty
 assert s.isEmpty();

 // insert two elements
 s.insert(2);
 s.insert(9);
 // check if the set is not empty anymore
 assert !s.isEmpty();
 // check that, for any contains() call that we do on the TreeSet,
 // it will only be true if the argument of the call is either 2 or 9,
 // because they are the only elements on the set
 var x;
 var isPresentInSet := s.contains(x);
 assert isPresentInSet <==> x == 2 || x == 9;

 // delete element
```

```
 s.delete(9);
  // add new element
 s.insert(4);

 // check that the set is not empty; still some elements left
 assert !s.isEmpty();
 var y;
 var isPresentInSet2 := s.contains(y);
 assert isPresentInSet2 <==> y == 2 || y == 4;

 // delete rest of the elements
 s.delete(2);
 s.delete(4);

 // check that the elements that were deleted are no longer in the set
 assert !s.contains(2);
 assert !s.contains(4);
 assert !s.contains(9);

 // check that set is empty again
 assert s.isEmpty();
}

// A test case, checked statically by Dafny, that covers the
// isSeq() method of the TreeSet class
method testTreeSetSequence()
{
 // create new set
 var s := new TreeSet();
  // check that the sequence that the asSeq() method returns is empty
 var sequence := s.asSeq();
 assert sequence == [];

 // insert a bunch of elements
 s.insert(2);
 s.insert(10);
 s.insert(8);

 // check that the returned sequence has the same elements and is sorted
 sequence := s.asSeq();
 assert sequence == [2, 8, 10];

 // delete some of the elements
 s.delete(8);
```

```
 // check the sequence of elements
 sequence := s.asSeq();
 assert sequence == [2, 10];

 // delete the rest of the elements
 s.delete(2);
 s.delete(10);

 // check that the set is now completely empty
 assert s.isEmpty();

 // check that the sequence that the asSeq() method returns is now empty
 sequence := s.asSeq();
 assert sequence == [];
}

// A test case, checked statically by Dafny, that should fail
// because of incorrect call to the TreeSet delete() method
method testTreeSetFailDelete()
{
 // create new set
 var s := new TreeSet();

 // insert some elements
 s.insert(2);
 s.insert(4);
 s.insert(6);

 // this method call fails the static analysis done by Dafny, because
 // a precondition for this method is not verified: the number 7 is not
 // on the set, so it cannot be deleted
 s.delete(7);
}

// A test case, checked statically by Dafny, that should fail
// because of incorrect call to the TreeSet insert() method
method testTreeSetFailInsert()
{
 // create new set
 var s := new TreeSet();

 // insert some elements
 s.insert(2);
 s.insert(4);
 s.insert(6);
```

```
    // this one should also fail the static analysis, once again because it
    // does not satisfy the precondition that an element that is already
    // on the set cannot be inserted again
    s.insert(2);
}
```

6. Sources and References

The code was based in this Binary Tree implementation on the official Dafny repository. Some references were also used to study the various operations of a Binary Search Tree and how to effectively implement them, like search, insertion and deletion of elements. The images used in this report are from these sources.

The Powerpoints and content from the Formal Methods in Software Engineering course were also used to check more Dafny-related details, namely this "Dafny Quick Reference" Powerpoint and this presentation about contracts and state abstraction.