Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 9

Programming Project

You will be able to choose from two projects:

- Project 1: Perfect Mine Sweeper Solver
  - model mine sweeper game and solver in Alloy
  - implement game and solver in Java
  - use run-time checking via jmlc/jmlrac
- Project 2: Verifying Dijkstra's Algorithm
  - implement Dijkstra's shortest path algorithm in Dafny
  - verify implementation against interface of a priority queue
  - implement and verify the priority queue against its interface

More details forthcoming this week.

Today’s Topics:
Class Invariants and Framing

Class Invariants

- Class invariants are properties that must hold at the entry and exit point of every method, for every instance of a class.
- They often express properties about the consistency of the internal representation of an object.
- They are typically transparent to clients of an object.
- They are sometimes also called object invariants or instance invariants.

The Problem with Class Invariants

There are some problems with class invariants:

- Ownership: invariants can depend on fields of other objects.
  - For example, the invariant of `List` accesses Node fields.
- Callback: invariants can be temporarily violated.
  - While the invariant is violated, we call a different method that calls back to the same object.
- Atomicity: invariants can be temporarily violated.
  - While the invariant is violated, another thread accesses object.

The Problem with Class Invariants

```
public class SomeClass {
  /*@ invariant inv; @*/
  /*@ requires P;
    @ ensures Q;
    @*/
  public void doSomething() {
    //@ assume(P);
    //@ assume(inv);
    ...code of doSomething
    //@ assert(Q);
    //@ assert(inv);
  }
}
```

```
public class OtherClass {
  public void caller(SomeClass o)
  {
    ...some other code
    //@ assert(P);
    o.doSomething();
    //@ assume(Q);
  }
}
```

- Is it enough to check the highlighted assumes and asserts?
- No, this would be unsound!

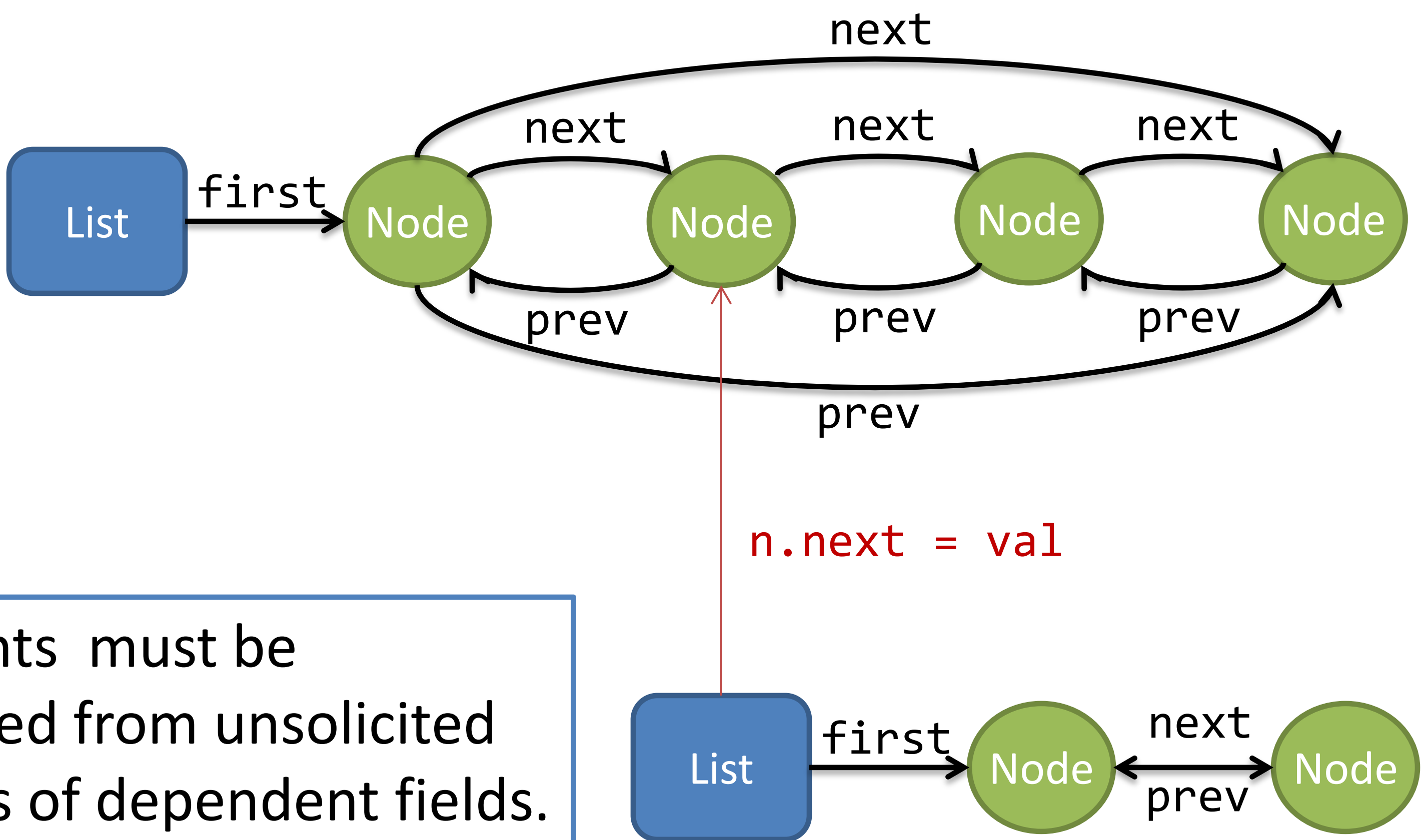
Invariants May Depend on Other Objects

Consider a doubly linked list:

```
class Node {
  Node prev, next;
  /*@ invariant this.prev.next == this &&
                this.next.prev == this; @*/
}
class List {
  private Node first;
  public void add() {
    Node newnode = new Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
  }
}
```

The invariant of `this` depends on the fields of `this.next` and `this.prev`. Moreover the `List.add` function changes the fields of the invariants of `Node`.

Invariants May Depend on Other Objects



Invariants must be protected from unsolicited updates of dependent fields.

List Example

First observation: the invariant should be put into the `List` class:

```
class Node { Node prev, next; }
class List {
  private Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
                 n.prev.next == n && n.next.prev == n); @*/
  public void add() {
    Node newnode = new Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

List Example

Second observation:
Node objects must not be shared between two different lists.

```
class Node {
  /*@ ghost Object owner; @*/
  Node prev, next;
}
class List {
  private Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n); n.prev.next == n &&
                 n.next.prev == n && n.owner == this); @*/
  public void add() {
    Node newnode = new Node();
    //@ set newnode.owner = this;
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
```

List Example

Third observation: One may only change the owned fields.

```
class Node {
  /*@ ghost Object owner; @*/
  Node prev, next;
}
class List {
  private Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n); n.prev.next == n &&
                n.next.prev == n && n.owner == this); @*/
  public void add() {
    Node newnode = new Node();
    //@ set newnode.owner = this;
    newnode.prev = first.prev;
    newnode.next = first;
    //@ assert(first.prev.owner == this)
    first.prev.next = newnode;
    //@ assert(first.owner == this)
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
```

The Owner-As-Modifier Property

JML supports a type system for checking the owner-as-modifier property, when invoked as

`jmlc --universes`.

The underlying type system is called Universes:

- The class `Object` has a ghost field `owner`.
- Fields can be declared as `rep`, `peer`, `readonly`.
  - `rep Object x` adds an implicit invariant (or requires) `x.owner == this`.
  - `peer Object x` adds an implicit invariant (or requires) `x.owner == this.owner`.
  - `readonly Object x` does not restrict owner, but does not allow modifications of `x`.
- The `new` operation supports `rep` and `peer`:
  - `new /*@rep@*/Node()` sets owner field of new node to `this`.
  - `new /*@peer@*/Node()` sets owner field of new node to `this.owner`.

List with Universes Type System

```
class Node { /*@ peer @*/ Node prev, next; }
class List {
  private /*@ rep @*/ Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
               n.prev.next == n && n.next.prev == n &&
               n.owner == this); @*/
  public void add() {
    Node newnode = new /*@ rep @*/ Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

The Universes Type System

A simple type system can check most issues related to ownership:

- `rep T` can be assigned without cast to `rep T` and `readonly T`.
- `peer T` can be assigned without cast to `peer T` and `readonly T`.
- `readonly T` can be assigned without cast to `readonly T`.

The Universes Type System

One needs to distinguish between the type of a field `peer Node prev` and the type of a field expression `rep Node first.prev`.

- If `obj` is a `peer` type and `fld` is a `peer T` field then `obj.fld` has type `peer T`.
- If `obj` is a `rep` type and `fld` is a `peer T` field then `obj.fld` has type `rep T`.
- If `obj = this` and `fld` is a `rep T` field then `this.fld` has type `rep T`.
- In all other cases `obj.fld` has type `readonly T`.

`readonly` References

To prevent changing `readonly` references, the following restrictions apply:

- If `obj` has type `readonly T,` then
  - `obj.fld = expr` is illegal.
  - `obj.method(...)` is only allowed if `method` is a `pure` method.
- It is allowed to cast `readonly T` references to `rep T` or `peer T`:
  - `(rep T) expr` asserts that `expr.owner == this`.
  - `(peer T) expr` asserts that `expr.owner == this.owner`.

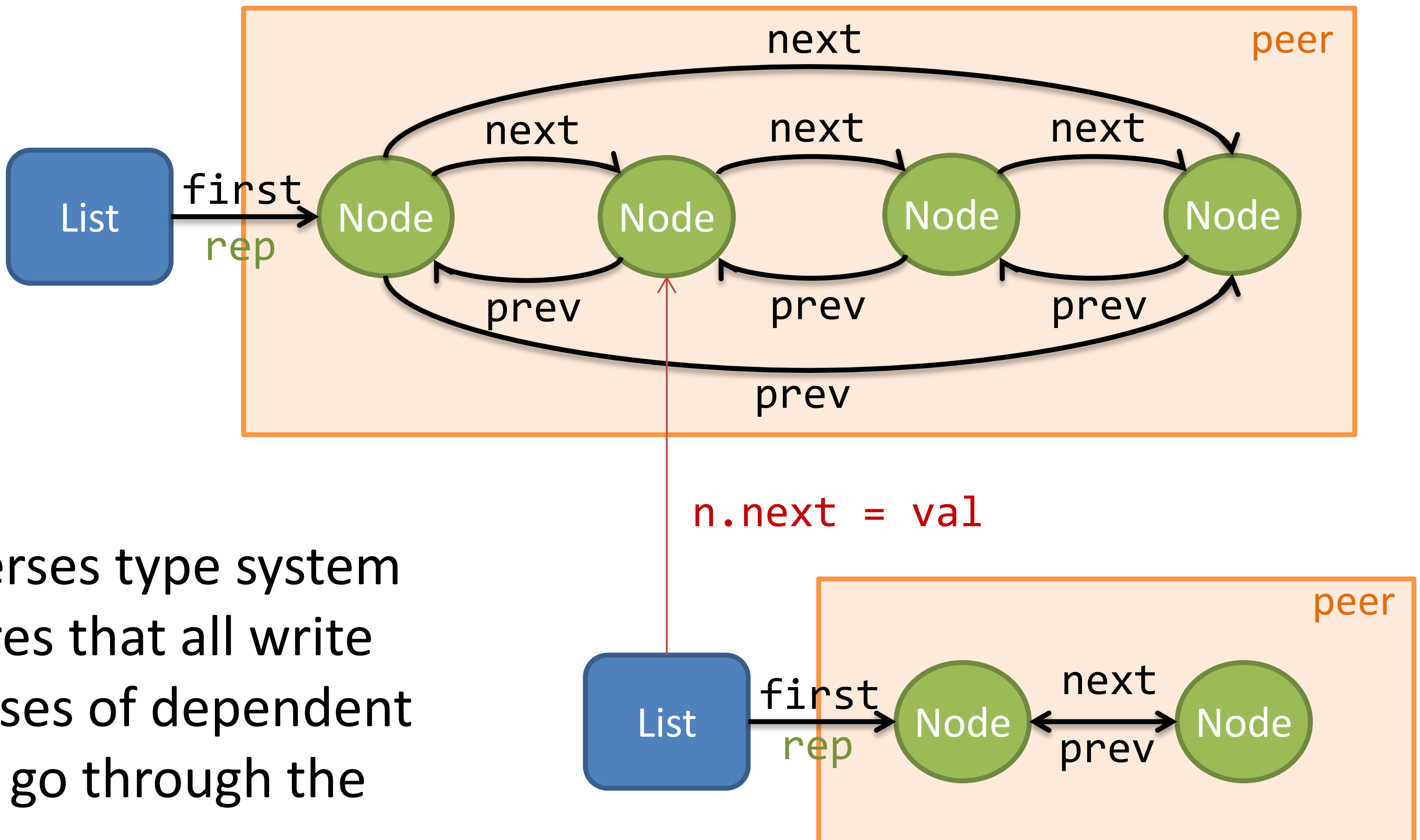
Modification only by Owner

All write accesses to a field of an object `obj` are

- in a method of the owner of `obj` or
- in a method of an object having the same owner as the object that was invoked (directly or indirectly) by the owner of `obj`.

Invariants that only depend on fields of owned objects can only be invalidated by the owner or methods that the owner invokes.

Modification only by Owner



Universes type system ensures that all write accesses of dependent fields go through the owner.

Limitations of Universes Type System

- The Universes type system can solve many ownership related problems.

but

- It’s granularity is often too coarse.
  - What happens if there is no unique owner?
  - What happens if invariants are temporarily violated?

Temporarily Violating Invariants

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/
  public void add(int v) {
    /* 1 */
    size++;
    /* 2 */
    if (size > content.length) {
      newContent = new int[2*size+1];
      ...
      content = newContent;
    }
    ...
    /* 3 */
  }
}
```

When do Invariants Hold?

- Before a public method is called. /* 1 */
- After a public method returns. /* 3 */
- However, it may be violated in between. /* 2 */

Calls to Private Methods

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/
  private /*@ helper @*/ void growContent() {
    ...
    content = newContent;
  }
  public void add(int v) {
    /* invariant should hold */
    size++;
    /* invariant may be violated */
    if (size > content.length)
      growContent();
    ...
    /* invariant should hold, again */
  }
}
```

Sometimes an invariant may not hold before a private method call.

JML provides the annotation `/*@ helper @*/` for this.

Calls to Methods of Other Classes

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/
  private /*@helper*/ void growContent() {
    /* invariant may be violated */
    newContent = new int[2*size+1];
    System.arraycopy(content, 0, newContent, 0, content.length);
    content = newContent;
  }
  ...
}
```

- The invariant still needs not to hold, when other methods are called, because there is the callback problem.

The Callback Problem

```
public class Log {
  public void log(String p) {
    logfile.write("Log: “ + p + " list is “ + Global.theList);
} }
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/
  public void add(int v) {
  /* invariant should hold */
  size++;
  /* invariant may be violated */
  if (size > content.length) {
    Logger.log("growing array.");
    ...
  }
  public String toString() {
    /* invariant should hold */
    ...
} }
```

The Callback Problem

```
public class Log {
  public void log(String p) {
    logfile.write("Log: “ + p + " list is “ + Global.theList);
} }
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/
  public void add(int v) {
  /* invariant should hold */
  size++;
  /* invariant may be violated */
  if (size > content.length) {
    Logger.log("growing array.");
    ...
  }
  public String toString() {
    /* invariant should hold */
    ...
} }
```

implicit call to method toString

The Callback Problem

- A method of a different class can be called while an invariant is violated.
- This method may call a method of the first class.
- Who has to ensure that the invariant holds?
  - jmlrac complains that the invariant does not hold, but only at run-time.
  - How can we detect such violations statically?

Dynamic Frames

The Dynamic Frames Approach

- **Problem**: a class invariant implicitly universally quantifies over the set of all allocated objects.
  - adding more objects can break the class invariant.
  - contradicts compositional verification approach.
- **Solution used in Dafny**: Dynamic Frames
  - each object only keeps track of its own invariants.
  - each object maintains a ghost field for its own representation frame
  - frames of different objects are kept separate by adding appropriate disjointness constraints.
  - yields compositional verification approach.

Example: Tree Data Structure

```
class TreeNode {
  var data: int;
  var left: TreeNode;
  var right: TreeNode;

  constructor Init(x: int)
  {
    data := x;
    left := null;
    right := null;
  }
  ...
}
```

Example: Tree Data Structure

```
class TreeNode {
  var data: int;
  var left: TreeNode;
  var right: TreeNode;
  ...
  method Insert(x: int)
  {
    if (x == data) { return; }
    if (x < data) {
      if (left == null) {
        left := new TreeNode.Init(x);
      } else {
        left.Insert(x);
      }
    } else { ...
    }
  }
```

Adding Ghost Field for Dynamic Frame

```
class TreeNode {
  var data: int;
  var left: TreeNode;
  var right: TreeNode;
  ghost var Repr: set<object>;

  constructor Init(x: int)
    modifies this;
  {
    ...
    Repr := {this};
  }
  ...
}
```

Adding Ghost Field for Dynamic Frame

```
method Insert(x: int)
  modifies Repr;
{
  if (x == data) { return; }
  if (x < data) {
    if (left == null) {
      left := new TreeNode.Init(x); }
    else {
      left.Insert(x);
    }
    Repr := Repr + left.Repr;
  } else { ...
  }
}
```

Adding Ghost Field for Dynamic Frame

```
method Insert(x: int)
  modifies Repr;
{
  if (x == data) { return; }
  if (x < data) {
    if (left == null) {
      left := new TreeNode.Init(x); }
    else {
      left.Insert(x);
    }
    Repr := Repr + left.Repr;
  } else { ...
  }
}
```

Error: assignment may update an object not in the enclosing context's modifies clause.

Tie Repr Field to Actual Frame

Repr is self framing

```
predicate Valid
  reads this, Repr;
{
  this in Repr && null !in Repr &&
  (left != null ==>
      left in Repr &&
      left.Repr <= Repr && this !in left.Repr &&
      left.Valid) &&
  (right != null ==>
      right in Repr &&
      right.Repr <= Repr && this !in right.Repr &&
      right.Valid) &&
  (left != null && right != null ==>
      left.Repr !! right.Repr)
}
```

Tie Repr Field to Actual Frame

```
predicate Valid
  reads this, Repr;
{
  this in Repr && null !in Repr &&
  (left != null ==>
      left in Repr &&
      left.Repr <= Repr && this !in left.Repr &&
      left.Valid) &&
  (right != null ==>
      right in Repr &&
      right.Repr <= Repr && this !in right.Repr &&
      right.Valid) &&
  (left != null && right != null ==>
      left.Repr !! right.Repr)
}
```

implicit ownership

Tie Repr Field to Actual Frame

```
predicate Valid
  reads this, Repr;
{
  this in Repr && null !in Repr &&
  (left != null ==>
      left in Repr &&
      left.Repr <= Repr && this !in left.Repr &&
      left.Valid) &&
  (right != null ==>
      right in Repr &&
      right.Repr <= Repr && this !in right.Repr &&
      right.Valid) &&
  (left != null && right != null ==>
      left.Repr !! right.Repr)
}
```

Left and right subtree are disjoint

Tie Repr Field to Actual Frame

```
class TreeNode {
  ...
  ghost var Repr: set<object>;
  predicate Valid { ... }
  constructor Init(x: int)
    modifies this;
    ensures Valid;
  { ... }
  method Insert(x: int)
    requires Valid;
    modifies Repr;
    ensures Valid;
    decreases Repr;
  { ... }
}
```

Check that invariant is maintained

Repr is also a ranking function for the recursive calls to Insert

Let’s look at a client of TreeNode

```
method Client()
 {
    var s1 := new TreeNode.Init(1);
    var s2 := new TreeNode.Init(2);
    s2.Insert(3);

    assert s1.Valid;
 }
```

Error: assertion violation

Let’s look at a client of TreeNode

```
method Client()
 {
    var s1 := new TreeNode.Init(1);
    var s2 := new TreeNode.Init(2);
    s2.Insert(3);

    assert s1.Valid;
 }
```

Error: assertion violation

We need to maintain the disjointness of frames!

Maintaining Disjointness of Frames

```
class TreeNode {
  ...
  ghost var Repr: set<object>;
  predicate Valid
  { ...
    (left != null && right != null ==>
      left.Repr !! right.Repr)
  }
  ...
  method Insert(x: int)
    requires Valid;
    modifies Repr;
    ensures Valid;
    decreases Repr;
  { ... }
}
```

Error: Related location.

Error: this postcondition might not hold.

Maintaining Disjointness of Frames

```
constructor Init(x: int)
  modifies this;
  ensures Repr == {this};
  ensures Valid;
{ ... }

method Insert()
  requires Valid;
  modifies Repr;
  ensures fresh(Repr – old(Repr));
  ensures Valid;
  decreases Repr;
{ ... }
```

Repr is only extended with freshly allocated objects

Specifying Functional Correctness

```
class TreeNode {
  ...
  ghost var Contents: set<int>;
  predicate Valid
  { ...
    Contents == (if left == null then {} else left.Contents) +
                (if right == null then {} else right.Contents) +
                {data}
  }
  ...
  constructor Init(x: int)
    ...
    ensures Contents == {x};
  { ...
    Contents := {x};
  }
```

Specifying Functional Correctness

```
method Insert(x: int)
  ...
  ensures Contents == old(Contents) + {x};
{
  if (x == data) { return; }
  if (x < data) {
    if (left == null) {
      left := new TreeNode.Init(x); }
    else {
      left.Insert(x);
    }
    Repr := Repr + left.Repr;
  } else { ...
  }
  Contents := Contents + {x};
}
```

Verification successful.

Let’s take a look at `Find`

```
method Find(x: int) returns (present: bool)
  requires Valid;
  ensures present <==> x in Contents;
  decreases Repr;
{
  if (x == data) {
    present := true;
  } else if (left != null && x < data) {
    present := left.Find(x);
  } else if (right != null && data < x) {
    present := right.Find(x);
  } else {
    present := false;
  }
}
```

Let's take a look at Find

```
method Find(x: int) returns (present: bool)
  requires Valid;
  ensures present <==> x in Contents;
  decreases Repr;
{
  if (x == data) {
    present := true;
  } else if (left != null && x < data) {
    present := left.Find(x);
  } else if (right != null && data < x) {
    present := right.Find(x);
  } else {
    present := false;
  }
}
```

Error: this postcondition might not hold.

Specifying the Representation Invariant

```
predicate Valid
  reads this, Repr;
{
  ...
  (left != null ==>
      ... &&
      (forall y :: y in left.Contents ==> y < data)) &&
  (right != null ==>
      ... &&
      (forall y :: y in right.Contents ==> y > data)) &&
  ...
  Contents == (if left == null then {} else left.Contents) +
              (if right == null then {} else right.Contents) +
              {data}
}
```

Tree is sorted

Let's take a look at `Find`

```
method Find(x: int) returns (present: bool)
  requires Valid;
  ensures present <==> x in Contents;
  decreases Repr;
{
  if (x == data) {
    present := true;
  } else if (left != null && x < data) {
    present := left.Find(x);
  } else if (right != null && data < x) {
    present := right.Find(x);
  } else {
    present := false;
  }
}
```

Verification successful.

Other Approaches to Frame Problem

- pack/unpack mechanism (Spec#, VCC)
  - based on ownership principle
  - solve callback problem by adding a ghost fields that keep track of object consistency.
- implicit dynamic frames (Chalice, VeriCool)
  - like dynamic frames
  - no modifies clauses needed
  - no explicit maintenance of `Repr` field needed
  - frames are encoded implicitly in pre- and postconditions.
- separation logic (VeriFast, jStar, ...)
  - similar to implicit dynamic frames
  - disjointness of frames comes for free.