

Data Abstraction

The lecture notes are based on chapter 5 of **Program Development in Java**, by Barbara Liskov and John Guttag, Addison-Wesley, 2001.

1 Overview

A programming language provides a fixed collection of types.

What if you want a type that is not provided? Then just implement it!

E.g., to implement a set object, use an array to store elements of the set. However, if we change the implementation to use a linked list, then the behavior of the set object *should not change*.

In data abstraction, we *abstract from the details of implementation of a data object*.

2 Abstract Data Type

Consider the above implementation of a set using an array. If we allow any module to access the array directly, then when we change the implementation to a linked-list, all modules that use the set object must be modified to process the new representation properly. To insulate the “using” modules from such changes of representation, we use the following **data hiding** idea:

1. implement some operations that access the array, and provide the standard set operations, e.g., `is-member?`, `insert`, `remove`, and
2. require that all accesses to the set object be via these operations.

Thus, when the representation is changed from an array to a linked list, only the implementations of the set operations need to be changed to process the new representation properly. The “using” modules do not need to be modified.

Since the number of using modules is usually far greater than the number of object operations, this is very important in saving effort and localizing code changes.

Such an implementation of set is an example of an **abstract data type**:

abstract data type = data + operations

The Liskov & Guttag book uses the term “data abstraction” for abstract data type. These mean the same thing.

Implementing abstract data types is very easy to do in an OO language:

data = instance variables

operations = methods

We can have many instances of the abstract data type. Each instance is an object. More on implementation below.

3 Specifying Data Abstractions

A specification of an abstract data type consists of:

1. A *class header*, (e.g., `public class IntSet`) which starts with an OVERVIEW statement describing informally the data type being specified, i.e., describing the objects.
2. A list of *constructors*: methods that initialize an object.
3. A list of *instance methods*: methods that provide access to an object.

Each constructor and method is a procedure, and so is specified using a REQUIRES clause and an EFFECT clause.

Consider the abstract data type “set of integers.” An example specification for this data type, `IntSet`, is given on p. 81 of Liskov & Guttag.

We can use the specifications of data abstractions in two ways:

1. to reason that the implementation of the data abstraction is correct, and
2. to reason that the program *using* the abstraction is correct.

4 Using Data Abstractions

We only need to know the specification of an abstract data type to invoke it. We don't need to know the implementation.

5 Implementing Data Abstractions

To implement a data abstraction, we do four things:

1. Select a **representation**
2. Define the **abstraction function** and **representation invariant** (see below for detailed discussion)
3. Implement constructors to initialize the representation properly, i.e., so that the representation invariant is true after the object is initialized
4. Implement methods to use/modify the representation properly, i.e., so that the representation invariant is preserved by each method call

5.1 Selecting a representation

A representation is a set of variables that are used to store the state of an instance of the data type, i.e., they are the *instance variables* of an object which is an instance of the data type.

The type of each of these variables can be a primitive type (provided by the programming language being used), or it can be another abstract data type.

Selecting a good representation is important, since some representations are much better than others, e.g., Arabic numerals are much better than roman numerals.

Criteria for a good representation:

1. It must enable all of the specified operations (constructors and methods) to be implemented with reasonable efficiency
2. It must enable the most frequent operations to be executed quickly. Thus, the right representation may depend on the pattern of usage, e.g, for `IntSet`:
an array is better if there are many accesses and few insertions
a linked list is better if there are many insertions and few accesses

5.1.1 Programming language considerations

To implement data hiding, we declare all instance variables to be `private`, i.e., not accessible by code outside of the defining class. Thus, all access to instance variables is mediated by the methods of the class itself.

5.2 Implement constructors and methods

These have already been specified. Implementation consists of providing code for each constructor and method that conforms to its specification. Additional “helper” methods may also be specified and implemented. These are usually declared `private`, and are only used by the public constructors and methods that provide the external interface of the data type.

In Liskov & Guttag, examples of implementation of data types are given on p. 88 for `IntSet`, and p. 91–92 for `Poly`, a data abstraction for polynomials. Note the addition of the private `getIndex` method in the implementation of `IntSet`.

5.3 The Abstraction Function

Let o be an object of some class \mathcal{C} which implements an abstract data type. At any time, the instance variables of o have particular values. This collection of values is called a *concrete state*¹.

A concrete state represents a (single) value of the abstract data type, i.e., an *abstract state*. Different concrete states can represent the same abstract state, e.g., in the implementation of `IntSet`, both vectors `[12]` and `[21]` represent $\{1, 2\}$. Thus, the relation between concrete and

¹More accurately, a concrete state is an assignment to each instance variable of a value from its type.

abstract states is a function, and is called the *abstraction function*. E.g., for `IntSet`, with representation object a vector $v[0..(v.size - 1)]$, an abstraction function is:

$$\text{set}[v] = \{x \mid \exists i : 0 \leq i < v.size : x = v[i]\}$$

Note that, e.g., $\text{set}([1\ 2]) = \text{set}([2\ 1]) = \{1, 2\}$. The order in which 1 and 2 appear in the vector is irrelevant to the abstract value represented. This is documented by definition of the set itself, since $\text{set}([1\ 2]) = \text{set}([2\ 1])$. Hence, the abstraction function tells us which aspects of the implementation are “internal” (e.g., the order of elements) and which affect the abstract value (e.g., the elements themselves). This is what abstraction is: deciding which information should appear externally, “at the interface”, and which should be hidden in the implementation.

A key requirement for the implementation is that the abstract and concrete operations must “commute” w.r.t. the abstraction function. Let `AF` be an abstraction function, c be a concrete state, `abstract-op` be an abstract operation, and `concrete-op` the corresponding concrete operation, i.e., method. Then,

$$\text{AF}(\text{concrete-op}(c)) = \text{abstract-op}(\text{AF}(c)).$$

For example, let s be an object of type `IntSet`, and let $\text{AF}(s) = \{1, 7, 11\}$. Then $s.\text{insert}(3)$ should result in a value for s such that $\text{AF}(s) = \{1, 3, 7, 11\}$, i.e., it should have the same effect as $\text{AF}(s) = \text{AF}(s) \cup \{3\}$.

5.4 The Representation Invariant

Not all states of the concrete representation are “legal”.

Legal states are specified by a *representation invariant*.

E.g., for `IntSet`, with representation object $v[0..(v.size - 1)]$ a representation invariant is:

$$\forall i, j : 0 \leq i < j < v.size : v[i] \neq v[j]$$

i.e., v contains no duplicates.

The representation invariant captures the underlying assumptions on which we build the abstraction. Different team members who are, e.g., implementing different methods of the data abstraction, can rely on the representation invariant as the “interface” between themselves.

The abstraction function only needs to be defined on concrete states where the representation invariant holds.

Checking the representation invariant at run time provides a way of finding bugs in the implementation, i.e., testing the implementation.

5.5 Implementing the abstraction function and representation invariant

The abstraction function can be implemented as a method that outputs (as a string) the value of the abstract state that is represented by the current concrete state. This can be useful, e.g., for debugging.

The representation invariant can be implemented as a method that checks if the invariant is true for the current concrete state. If so, it outputs “true”, and otherwise it outputs “false”. This is also useful for debugging.

6 Properties of Implementations

6.1 Benevolent side effects

A “query only” method can change the concrete object as a “side effect” if that does not change the abstract value, i.e., can change from c to c' if $\text{AF}(c) = \text{AF}(c')$. This may be useful to speed up subsequent operations.

6.2 Exposing the Representation

If the implementation makes an instance variable available to code outside of the implementation, then the implementation *exposes the representation*.

This could happen, e.g., if (1) the instance variables are not declared `private`, or (2) the instance variables are private, but a reference (pointer) to the instance variables is returned by some method of the implementation.

If the representation is exposed, then external code could inadvertently make the representation invariant false. This destroys the assumptions under which the implementation has been coded, and may lead to incorrect results in subsequent method calls to the implementation. E.g., consider if the `IntSet` implementation has duplicate elements inserted into v .

So, exposing the representation is very bad. It destroys modularity. Consider it a design/coding error.

7 Reasoning about data abstractions

Preserving the representation invariant:

One proves that the representation invariant I is preserved by *data type induction*, i.e.,:

1. Show that all constructors create objects that satisfy I , and
2. show that if an instance method that makes changes (a mutator) is invoked on an object that satisfies I , then the object still satisfies I when the method returns (note: it is OK to violate I in the middle of a method call). Also, any objects of the same type that are constructed, e.g., as return values, or that are modified, e.g., as parameter reference types, must also satisfy I upon termination of the method.

Advantage of this approach: we can reason about each method *in isolation*.

Pitfall: this approach is unsound if the representation is exposed.

8 Example: IntSet

We present an implementation of the `IntSet` data type. We include:

1. overview statement and definitions for the rep invariant and abstraction function

2. method specifications
3. code sketches
4. the actual code, annotated with assertions

```
import java.util.Scanner;

public class IntSet {

    // OVERVIEW: an IntSet is a mutable unbounded set of integers,
    // e.g., {x_1, ..., x_n}

    // Instance variables
    private int[] els;
    private int top;

    /* IMPLEMENTATION: uses an integer array els[] and an integer top that
     * indexes els. els[0:top-1] consists of exactly the elements of IntSet.
     *
     * REPRESENTATION INVARIANT REP(els,top): 0 <= top <= els.length
     *
     * ABSTRACTION FUNCTION: AF(els,top) = {x | (exists i : 0 <= i < top: els[i] = x)}
     */

    // Abbreviations: AF = AF(els,top), REP = REP(els,top)

    // CONSTRUCTORS

    public IntSet() {
        // EFFECTS: Creates an empty IntSet, i.e., AF = emptyset

        els = new int[2]; //Allocate two spaces initially.
        top = 0;
    }

    // METHODS

    private int getIndex(int x) {

        // EFFECTS: if x in AF, returns i s.t. 0 <= i < top and els[i] = x,
        //           else returns -1.

        int i = 0;
```

```

// {inv: x notin els[0:i-1] and 0 <= i <= top}
while (i != top && els[i] != x) { //NB: if you switch conjuncts, can get index out
    //of bounds. Better to put els[i] != x inside loop body.
    //{x notin els[0:i-1] and 0 <= i <= top and els[i] != x and i != top}
    //{x notin els[0:i] and 0 <= i < top}
    i = i+1;
    //{x notin els[0:i-1] and 0 <= i <= top}
} //endwhile
//{x notin els[0:i-1] and 0 <= i <= top and (els[i]=x || i = top)}

if (i == top)
    //{x notin els[0:top-1]}
    return -1;
else
    //{0 <= i < top and els[i]=x}
    return i;
//endif
}

```

```

public boolean isIn(int x) {

    //EFFECTS: if x in AF, returns true else returns false

    return (getIndex(x) != -1);
}

```

```

public void remove(int x) {

    //EFFECTS: AF_post = AF - x
    //MODIFIES: els, top

    /* CODE SKETCH
    * while some occurrence of x remains in els
    *     remove it
    * if top <= els.length/4
    *     halve the size of els
    */

    //{REP and AF = S}
    int i;
    while (true) {
        //{invariant: REP and (AF = S or AF = S-x) }
        i = getIndex(x); //Next occurrence of x to be removed.
        if (i == -1) //No occurrence, so return.
            //{REP and (AF = S or AF = S-x) and x notin S}
            //{REP and AF = S - x} //Satisfies EFFECTS.
            break;
    }
}

```

```

else {
    //Remove the occurrence.
    //{REP and (AF = S or AF = S-x) and x = els[i] and 0 <= i < top}
    els[i] = els[top-1];
    top = top - 1;
    //{REP and (AF = S or AF = S-x) }
}
} //endwhile

//{REP and AF = S - x} //Satisfies EFFECTS.
if (top <= els.length/4) {
int[] a = new int[els.length/2];
//Loop to implement a[0:top-1] = els[0:top-1]
for(int j = 0; j < top; j++) a[j] = els[j];
    //{REP(a,top) and AF(a,top) = S-x and 0 < top <= a.length/2}
    els = a;
//{REP and AF = S-x and top <= els.length/2}
}
}

```

```

public void insert(int x) {

    //EFFECTS: AF_post = AF U x
    //MODIFIES: els, top

    /* CODE SKETCH
    * if no space available in els
    *     double size of els
    * endif
    * insert x
    */

    //{REP and AF = S}
    if (top == els.length) { //No space available in els.
        //{REP and AF = S and 0 < top = els.length}
        int[] a = new int[2*top];
        //Loop to implement a[0:top-1] = els[0:top-1]
        for(int i = 0; i < top; i++) a[i] = els[i];
            //{REP(a,top) and AF(a,top) = S and 0 < top < a.length = 2*top}
            els = a;
            //{REP and AF = S and top < els.length}
        }
    //else
        //{REP and AF = S and top != els.length}
        //{REP and AF = S and top < els.length}
    //endif
    //{REP and AF = S and top < els.length}

    //Now insert x, since top < els.length, so space is available in els.

```

```

    els[top] = x;
    top = top + 1;
    //{REP and AF = S U x}
}

private boolean repOk() {

    //EFFECTS: returns the value of the representation
    //invariant 0 <= top <= els.length

    return(0 <= top && top <= els.length);
}

private String repToString() { //EXPOSES REP: USE ONLY FOR DEBUGGING

    //EFFECTS: returns the values of top and els packaged into a
    // single string.

    String st = "top = " + top + " els = ";
    for (int i = 0; i < els.length; i++)
        st = st + els[i] + " ";
    return(st);
}

public String toString() {

    //EFFECTS: returns the abstract set value packaged into a string.

    //This implementation is inefficient: has O(top^2) running
    //time. Should replace by an O(top \lg top) implementation in
    //the final production version

    IntSet s2 = new IntSet(); //new IntSet used to eliminate duplicates

    String st = "IntSet = {";

    //insert each element of this exactly once into s2
    for (int i = 0; i < top; i++)
        if (!s2.isIn(els[i])) s2.insert(els[i]);

    //now package the elements of s2 (i.e., s2.els since
    //no duplicates) into a string and return
    for (int i = 0; i < s2.top-1; i++)
        st = st + s2.els[i] + ", ";

    //if statement to check if top-1 is in range, in case AF = emptyset

```

```

    if (s2.top-1 >= 0) st = st + s2.els[s2.top-1] + "}";
    else st = st + "}";

    return(st);
}

public static void main(String[] args) throws Exception {

    IntSet s = new IntSet();

    //test.txt contains a test script. Each line consists of
    //"operation argument" where operation is one of "insert,"
    //"remove,", and "isIn" and argument is an integer.
    java.io.File test = new java.io.File("test.txt");
    Scanner input = new Scanner(test);

    //Print out initial rep and check rep invariant.
    System.out.println(s.repToString());
    System.out.println("rep inv is " + s.repOk());
    System.out.println();

    //Read through test.txt and perform operations.
    while (input.hasNext()) {

        String op = input.next();
        int arg = input.nextInt();

        if (op.equals("isIn")) {
            System.out.println("isIn(" + arg + ") = " + s.isIn(arg));
        }
        else if (op.equals("insert")) {
            System.out.println("insert(" + arg + ")");
            s.insert(arg);
            System.out.println(s.repToString());
            System.out.println(s.toString());
            System.out.println("rep is " + s.repOk());
        } else if (op.equals("remove")) {
            System.out.println("remove(" + arg + ")");
            s.remove(arg);
            System.out.println(s.repToString());
            System.out.println(s.toString());
            System.out.println("rep is " + s.repOk());
        }

        System.out.println(); //Blank line between results of
                               //successive operations.
    }
}

```

```
}  
}
```

9 Linked Lists

We now show how to specify and implement singly linked lists and their various operations. First we provide a specification. Note that `Node` is an inner class of `LinkedList`.

```
public class LinkedList {  
  
    /* OVERVIEW: Class to implement a linked list. A linked list consists  
    *   of a head Node, which points to the  
    *   next node, etc. Implementation uses a single instance variable h  
    *   of type Node (see inner class below) to represent the first node  
    *   in the list. The rest of the list is reached by following next pointers.  
    */  
  
    // Instance vars (i.e., the rep)                                // head of the list  
    Node h;  
  
    //Inner class that defines the Node type  
    private class Node {  
        // OVERVIEW: a Node is an object that contains an integer value and  
        //   a pointer to another Node.  
  
        int val;                                // integer value stored in the node  
        Node nxt;                               // pointer to the next node in the list  
  
    }  
  
    /* ABSTRACTION FUNCTION:  
    * The abstraction function gives the sequence of values stored in the  
    * successive nodes. It is defined recursively. + is sequence  
    * concatenation and lambda is the empty sequence.  
    *  
    *   AF(h) = h.val + AF(h.nxt)  
    *   AF(null) = lambda  
    *  
    *  
    * REPRESENTATION INVARIANT REP(h)  
    * The representation invariant requires that lists be acyclic: a node  
    * cannot point to an "earlier" node in the list.  
    * We first define a function reach(n) that gives all the nodes that are  
    * "reachable" from a node n:  
    *  
    *   reach(n) = n.nxt union reach(n.nxt)
```

```

*   reach(null) = emptyset
*
* Then
*
*   acyclic(n) = n notin reach(n)
*
* states that node n is not part of a cycle, since otherwise n would be
* reachable from itself. The rep. invariant states that every node in the
* list (including the head node h) is not part of a cycle:
*
*   REP(h): (forall n : n in h union reach(h) : acyclic(n))
*
* We use h union reach(h) in the range since h is not necessarily in
* reach(h). Also, REP(h) permits h = null. This is necessary,
* since h = null represents an empty list, which we otherwise could
* not represent if we required h != null as part of REP(h).
*
* Abbreviations: AF = AF(h), REP = REP(h)
*/

```

```
// CONSTRUCTORS
```

```

public LinkedList() {
    //EFFECTS: Creates an empty linked list.

    h = null;
}

```

```

public LinkedList(int i) {
    //EFFECTS: Creates a linked list consisting of a single node containing i

    h.val = i;
    h.next = null;
}

```

```
// METHODS
```

```

public void insert(int i) {
    // EFFECTS: inserts a new node containing value i at the head of this
    // MODIFIES: this
}

```

```

public void delete() {

```

```

    // REQUIRES: this != null
    // EFFECTS: deletes the first node of this
    // MODIFIES: this
}
}

```

How do we implement insert and delete? We formalize the specification of insert as follows.

```

public void insert(int i) {

    //{AF(h) = L}
    insert;
    //{AF(h) = i + L}
}

```

where L is a constant of type “sequence of integers”, which includes the empty sequence. We expand the postcondition, using the definition of AF:

$$h.val = i \wedge AF(h.next) = L$$

We must introduce a new node v to hold the inserted value i . Hence we require $v.val = i$. This is easy to establish using

```

Node v = new Node();
v.val = i;

```

The postcondition is $h.val = i \wedge AF(h.next) = L$. We can make $h.val = i$ true (upon termination) by setting h to v . However this does not in general make $AF(h.next) = L$ true. So, we calculate the precondition needed:

```

/{v.val = i /\ AF(v.next) = L}
h = v;
/{h.val = i /\ AF(h.next) = L}

```

$v.val = i$ is established by the previous piece of code. We can make $AF(v.next) = L$ true by exploiting the precondition $AF(h) = L$: just set $v.next$ to h . So, working backwards, we get:

```

/{v.val = i /\ AF(h) = L}
v.next = h;
/{v.val = i /\ AF(v.next) = L}
h = v;
/{h.val = i /\ AF(h.next) = L}

```

Now we add the code to create v and set $v.val$, add the header, and simplify the postcondition at the end to obtain the complete annotated `insert` method:

```

public void insert(int i) {

    //{AF(h) = L}
    Node v = new Node();
    //{AF(h) = L}
    v.val = i;
    //{v.val = i /\ AF(h) = L}
    v.nxt = h;
    //{v.val = i /\ AF(v.nxt) = L}
    h = v;
    //{h.val = i /\ AF(h.nxt) = L}
    //{AF(h) = i + L}
}

```

Note that we have to be careful when using the assignment axiom with pointer structures. For example, the postcondition $h.val = i \wedge AF(h.nxt) = L$ suggests the assignment $h.nxt := h$, since replacing $h.nxt$ by h in $AF(h.nxt) = L$ results in $AF(h) = L$, which is the precondition, i.e.,

```

://{AF(h) = L}
h.nxt = h;
://{h.val = i /\ AF(h.nxt) = L}

```

However, this is obviously wrong since it does not actually use the value i to be inserted. One problem is that $h.nxt = h$ violates the representation invariant $REP(h)$: it creates a cycle consisting of the single node h . Our solution above does not violate $REP(h)$.

The lesson is that we have to (1) be careful with pointers, (2) check that our rep. invariant makes sense, (3) check that our code preserves the rep. invariant, and (4) keep in mind that development like the above is only “semi formal”, and prone to logical error if we are not careful.

Developing sound proof rules for pointer structures is still a research problem. Some rules have been developed, but they are quite difficult to apply, and result in very detailed and tedious tableaux and proofs.

To implement delete we formalize the specification as follows.

```

public void delete() {

    //{AF(h) = i + L}
    delete;
    //{AF(h) = L}
}

```

We expand the precondition, using the definition of AF:

$$h.val = i \wedge AF(h.nxt) = L$$

This suggests the assignment $h := h.nxt$, which works. So we obtain:

```

public void delete() {

```

```

    //{AF(h) = i + L}
    //{h.val = i /\ AF(h.nxt) = L}
    h := h.nxt; \
    //{AF(h) = L}
}

```

10 Binary Trees

We will develop a representation for unordered binary trees, and illustrate it using a program to sum up the nodes of the tree (each of which contains an integer value).

Each node of the tree is given by:

```

public class Node {
    int i;          //integer value stored in the node
    Node l, r,     //pointers to left child and right child
    ...
}

```

We will manipulate the instance variables i, l, r using references and assignments, i.e., we assume that our tree traversal method is part of a class that has access to these instance variables. If not, we can always replace references and assignments by the obvious getter and setter methods.

```

public class BinaryTree {

    /* OVERVIEW: Class to implement an unordered binary tree, which consists
    *   of a root Node, which points to a left child and a right child, which
    *   are Nodes, and may be the roots of left and right subtrees.
    *   Uses a single instance variable r of type Node (see inner class below)
    *   to represent the root.
    */

    // Instance vars (i.e., the rep)
    Root r;                               // head of the list

    private class Node {
        // OVERVIEW: a Node is an object that contains an integer value and
        //   left and right pointers to other nodes.

        int val;                            // integer value stored in the node
        Node left, right;                    // pointers to the next node in the list
    }

    /* ABSTRACTION FUNCTION:
    * The abstraction function gives the tree of the values stored in the

```

```

* successive nodes. It is defined recursively. + is sequence
* concatenation and lambda is the empty sequence. We use (...) to
* indicate the tree structure in preorder notation: root first,
* left subtree enclosed in (...), then right subtree enclosed in (...)
*
*   AF(r) = r.val + ( AF(r.left) ) + ( AF(r.right) )
*   AF(null) = lambda
*
* REPRESENTATION INVARIANT REP(r):
* The representation invariant requires that trees be acyclic: a node
* cannot have a child pointer to an "ancestor" node. It is defined by first
* defining a function desc(n) that gives all the nodes that are
* "reachable" from a node n, i.e., the descendants of n:
*
*   desc(n) = n.left union desc(n.left) union n.right union desc(n.right)
*   desc(null) = emptyset
*
* Then
*
*   acyclic(n) = n notin desc(n)
*
* states that node n is not a descendant of itself. The rep. invariant
* states that every node in the tree (including the root r) is not amongst
* its own descendants:
*
*   REP(r): (forall n : n in r union in desc(r) : acyclic(n))
*
* Abbreviations: AF = AF(r), REP = REP(r)
*/

// CONSTRUCTORS

public BinaryTree(int v) {
    //EFFECTS: Creates a binary tree consisting of a single node containing v

    r.val = i;
    r.left = null;
    r.right = null;
}

```

10.1 The Tree Traversal Problem

We must visit each node in the tree at least once and compute the sum of the values stored at all the nodes. The specification is as follows:

```

public int add(Node r)
    //REQUIRES: REP(r)

```

```
//EFFECTS: returns SUM(r) where
SUM(r) = (SIGMA n : n in r union desc(r) : n.val)
```

A tree is a naturally recursive data structure and many algorithms are most naturally expressed as recursion on the left and right subtrees. We can sum a tree by recursively computing the sum of the left and right subtrees and then adding the value of the root. Using the proof rule for conditional correctness of recursive procedures, we can assume that the recursive calls work correctly. Termination is easy: since the recursion is on subtrees, we can simply use the number of nodes in the tree as the variant function: this is obviously always ≥ 0 , and it decreases on each recursive call. We obtain the following (where `ret` is an auxiliary variable denoting the returned value):

```
public int add(Node r) {
  //REQUIRES: REP(r)
  //EFFECTS: returns SUM(r) where
    SUM(r) = (SIGMA n : n in r union desc(r) : n.val)

  int sumL, sumR;                                //sum of left and right subtrees, resp.

  {REP(r)}
  if (r == null) then
    {REP(r) /\ r=null}
    return 0;
    //{ret = 0 = SUM(r) = (SIGMA n : emptyset : n.val)}
  else {
    {r != null /\ REP(r)}
    {REP(r.left)}
    sumL = add(r.left);
    {sumL = SUM(r.left) = (SIGMA n : n in r.left union desc(r.left) : n.val)}

    {r != null /\ REP(r)}
    {REP(r.right)}
    sumR = add(r.right);
    {sumR = SUM(r.right) = (SIGMA n : n in r.right union desc(r.right) : n.val)}

    return(r.val + sumL + sumR);
    {ret = r.val +
      (SIGMA n : n in r.left union desc(r.left) : n.val) +
      (SIGMA n : n in r.right union desc(r.right) : n.val)
      =
      r.val + (SIGMA n : n in desc(r) : n.val)
      =
      (SIGMA n : n in r union desc(r) : n.val)
    }
  }
}
}
```