

## Lecture 7: Representation Invariants

This is the first of a pair of lectures on the theory underlying data abstraction. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and are less likely to fall into subtle traps.

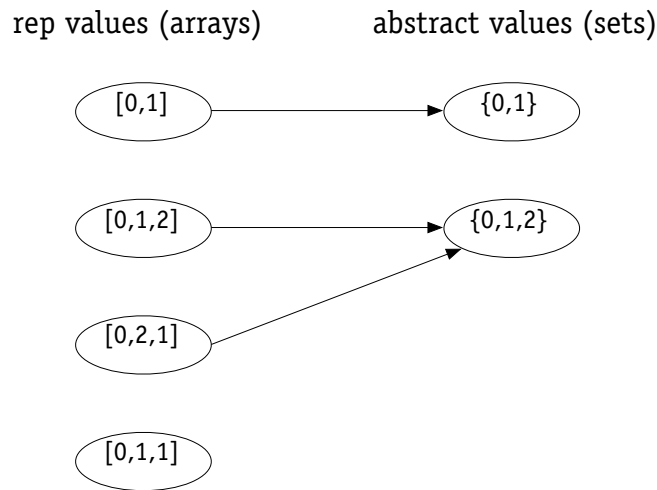
### 7.1 Two Spaces: Rep and Abstract

In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imagination. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type. A map type, for example, might have abstract values that are tables, each with two columns, one for keys and one for values. The fact that the map is implemented with some arrays and linked lists of pairs is not of interest to the user of the type. Similarly, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; again, the fact that it is implemented as an array of primitive integers, say, is not relevant to the user.

Now of course the implementor of the abstract type *must* be interested in the representation values, since it is her job to achieve the illusion of the abstract value space using the rep value space. Suppose, for example, that we choose to use an array of integers to represent a set of integers. Then these form our two value spaces. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents:



There are several things to note about this graph:

- Not all rep values are mapped. In this case, the array  $[0,1,1]$  is not mapped. If the type of the rep is non-trivial, it will not make sense to give an interpretation for any rep value; a list-like structure with entries, for example, can be convoluted into all kinds of networks that won't correspond to lists, and for which we won't want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In this case, we have decided that the array should not contain duplicates. This will allow us to terminate the remove method when we hit the first instance of a particular integer (since we know there can be at most one).
- Every abstract value is mapped to. The purpose of implementing the abstract type is to support operations on abstract values. Presumably then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable. (In fact, to be honest, this property is slightly tricky: since we haven't said exactly how the abstract value space is constructed, it would be reasonable to say that there is a type definition, akin to the type definition of the rep, and then a condition that determines which values are 'real' abstract values. But this is a question of how the abstract values are specified, which need not concern us here. The key idea is that you need to be able to represent all the meaningful abstract values.)
- Some abstract values are mapped to by more than one rep value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set as an ordered array, for example: both  $[0,1,2]$  and  $[0,2,1]$  might represent the set  $\{0,1,2\}$ .

In practice, we can illustrate a few elements of the two spaces and their relationships, but the graph as a whole is infinite. So we describe it by giving two things:

- A *representation invariant*, or *rep invariant*, that maps rep values to boolean:

7.1.1  $R : \text{Rep} - \text{Bool}$

For a rep value  $r$ ,  $R(r)$  is true if  $r$  is mapped by  $R$ . In other words,  $R$  tells us whether a given rep value is well formed. Alternatively, you can think of  $R$  as a set: it's the subset of rep values that are well formed.

- An *abstraction function* that maps rep values to the abstract values they represent:

7.1.2  $A : \text{Rep} - \text{Abs}$

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is onto, usually partial, and not necessarily injective.

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would of course be useless, since they would be saying something redundant that's already available elsewhere.

It's easy to see why the abstract value space alone doesn't determine  $A$  or  $R$ ; there can be several representations for the same abstract type. A set of integers could equally well be represented as a list of integers, for example, or if the integers are bounded, by a bit vector. Clearly we need two separate functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine  $A$  and  $R$ . The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. For example, instead of deciding, as we did above, that the arrays have no duplicates, we could allow duplicates, but at the same time require that the array be sorted – that is, the integers appear in non-decreasing order. This would allow us to perform a binary search and thus check membership in logarithmic rather than linear time. Even with the same type for the rep value space and the same rep invariant  $R$ , we might still have different interpretations  $A$ . Suppose  $R$  admits any array of integers. Then we could define  $A$ , as above, to interpret the array's elements as the elements of the set. But there's no a priori reason to let the rep fix the interpretation. Perhaps we'll interpret consecutive pairs of elements as subranges, so the array  $[1,5,9,9]$  represents the set  $\{1,2,3,4,5,9\}$ .

The essential point is that design of an abstract type means not only choosing the two spaces – the abstract value space for the specification and the rep value space for the implementation – but also deciding what rep values to use and how to interpret them.

In this lecture, we'll focus on the space of rep values and on the representation invariant R. In the next lecture, we'll look at the abstraction function A.

## 7.2 Rep Invariants for a Linked List

Here is the skeletal code of our linked list class (whose object model we discussed in our last lecture) and its associated class for list entries:

```

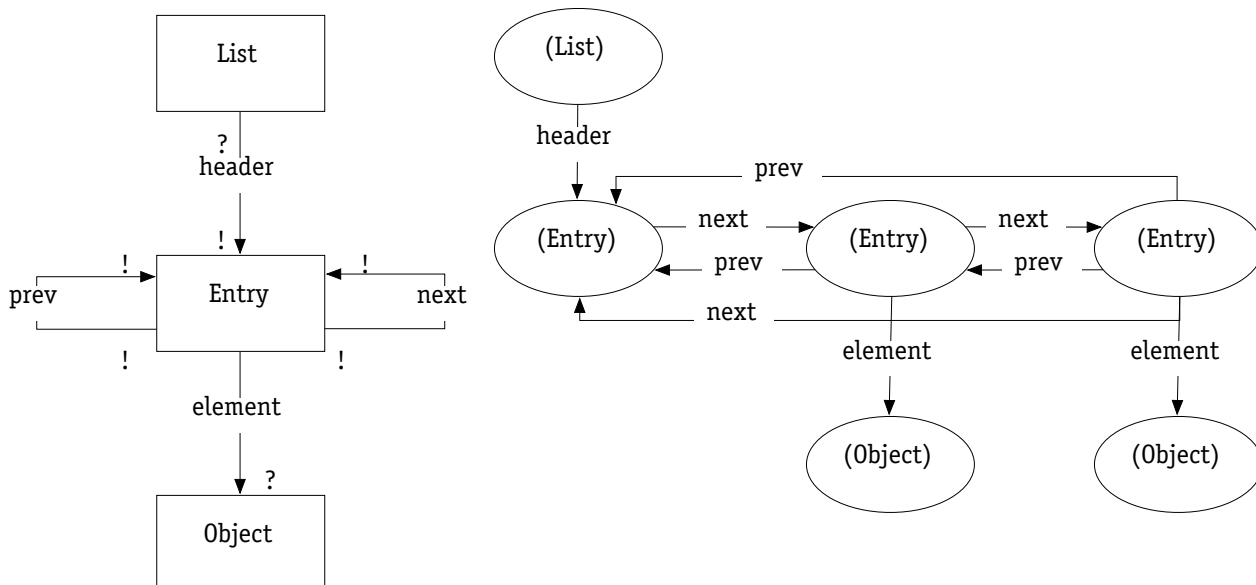
7.2.1 public class LinkedList {
7.2.2     Entry header;
7.2.3     ...
7.2.4 }

7.2.5 class Entry {
7.2.6     Object element;
7.2.7     Entry prev;
7.2.8     Entry next;
7.2.9     Entry (Object e, Entry p, Entry n) {element = e; prev = p; next = n;}
7.2.10 }

```

The list object has a field header that references an Entry object. An Entry object is a record with three fields: next and prev which may hold references to other Entry objects (or be null), and element, which holds a reference to an element object. The next and prev fields are links that point forwards and backwards along the list. In the middle of the list, following next and then prev will bring you back to the object you started with. There is always a dummy Entry at the beginning of the list whose element field is null, but this is not interpreted as an element.

Here is the object model and a sample snapshot showing a list with two elements:



What is the rep invariant for `LinkedList`? I haven't actually shown you enough to figure it out; as I said above, the declaration of the representation doesn't determine it, and one snapshot and some vague text doesn't give you the information you need. So let me tell you the rep invariant I have in mind. I want the entries to be doubly-linked in a chain that comes full circle, as illustrated in the sample snapshot. I want to ensure that there will also be a dummy entry for the header.

How shall we write  $R$  down? I could write it like this

7.2.11  $R(l) = \dots \text{formula} \dots$

where `formula` is a logical formula that evaluates to true or false for a particular value of the representation space, the list `l`. I could also express it as a formula that asserts a property of all linked lists, since I intend  $R(l)$  to be true for every instance of `LinkedList`:

7.2.12  $\text{all } l: \text{LinkedList} \mid \dots \text{formula} \dots$

7.2.13 So to express the constraints I've mentioned, I might write

7.2.14  $R(l) = \{$

7.2.15  $\quad l.\text{header} \neq \text{null}$

7.2.16  $\quad \text{all } e: l.\text{header}.*\text{next} \{$

7.2.17  $\quad \quad e.\text{prev}.\text{next} = e$

7.2.18  $\quad \quad e.\text{next}.\text{prev} = e$

7.2.19  $\quad \quad \}$

7.2.20  $\}$

I've used the syntax of Alloy, a modelling language developed in my research group here at MIT. I don't expect you to learn Alloy, or indeed use anything so formal in your own invariants, but it's as good a language to explain things as any. To read this invariant, you need to know a few simple things: that formulas placed one after another are implicitly conjoined (that is, 'anded' together); that the expression `s.f` is the set of objects you get by starting with any object in the set `s` and following the field `f`; and that `s.*f` means that you follow the field `f` zero or more times.

So the invariant says that a linked list is well formed if and only if:

- the header field is not null (7.2.15);
- and for each entry object `e` that can be reached by following the header field and then zero or more steps of the `next` field (7.2.16) ...
- following the `prev` field and then the `next` field from `e` gets you back to `e` (7.2.17), and
- following the `next` field and then the `prev` field from `e` gets you back to `e` (7.2.18).

Note that the last two constraints (7.2.17 and 7.2.18) imply that the entries must have non-null `next` and `prev` fields.

Writing an invariant down precisely is like writing code. It can be tricky to make sure that you've said exactly what you mean. In this case, we've actually missed a constraint, since our rep invariant allows lists like this:

We can ensure that the chain of entries forms a loop by requiring that there be an incoming next edge into the header entry:

```
7.2.21  R(l) = {  
7.2.22      ...  
7.2.23      some e: l.header.*next | e.next = l.header  
7.2.24  }
```

and now we've characterized the backbone structure of the list.

When you write a rep invariant as a comment in the code, it's convenient to format it differently, as a collection of properties about the fields of the object, like this, for example:

```
7.2.25  public class LinkedList {  
7.2.26      Entry header;  
7.2.27      /* rep invariant:  
7.2.28         dummy entry exists:  
7.2.29             header != null  
7.2.30         entries form a loop:  
7.2.31             all e: header.*next {  
7.2.32                 e.prev.next = e  
7.2.33                 e.next.prev = e  
7.2.34             }  
7.2.35             some e: header.*next | e.next = header  
7.2.36         */  
7.2.37      ...  
7.2.38  }
```

The constraints of our rep invariant are the ones that make it possible to implement this data type in this way. If we didn't have them, then a method that manipulated the list would not be able to assume that the entries were connected in any particular way, and that of course would make coding the list impossible. So the rep invariant exists (at least in the programmer's mind) whether or not it's written down; the code can't be written without out.

There are other constraints that we might have included in the rep invariant:

- Should we insist that the element field of the header entry be null? There's no good reason to do this. It won't effect the performance or ease of coding of the list, since we don't intend ever to do anything with that field. Of course, we could be really perverse and hold on to a reference to an arbitrary element there, even after it is

removed from the list. This might indeed have a bad consequence – it would prevent everything reachable from it from being garbage collected.

- Should we insist that the element field of every other entry be *non-null*? Unlike the constraints about the backbone structure of the list, this is a constraint which has a major impact on the specification. If we include this constraint, our rep will no longer be *adequate*, since there will be no way to represent lists containing nulls. Of course we could modify the specification and change our abstract space so that it excludes such lists. But there's little reason to do this; it's no harder to build a list that allows nulls to be stored.
- Should we insist that no two entry objects have element fields that point to the same object? This again impacts the specification, by ruling out lists with duplicates, and also has implementation consequences: insertion of an element will now require a check that the element is not already a member of the list. It should also be noted that even if some application seems to call for a list without duplicates, it may be better to use two abstract types: a list that allows duplicates, and a set used to collect the elements and check that no element is inserted twice.

### 7.3 The Impact on Implementation

The rep invariant, more than anything else, determines what the code of the abstract type will look like. On the one hand, a strong invariant is a good thing: when you're writing an observer, you can make a lot of assumptions about the structure, so you can take short cuts, assume ordering, not worry about certain configurations, and so on. On the other hand, a strong invariant is troublesome: when you're writing a producer (or mutator) you'll have to create an object (or an object state) that satisfies the invariant, which will generally involve more code than it would otherwise.

Let's see how this plays out for some methods of `LinkedList`. Let's start with an observer, a method `getLast` that returns the last element of the list:

```
7.3.1  Object getLast () {  
7.3.2      Object p = header.prev;  
7.3.3      if (p == header) throw new NoSuchElementException ();  
7.3.4      return p.element;  
7.3.5  }
```

Note first of all the unprotected dereferences. We don't need to check that `header` is non-null before dereferencing its `prev` field, and we don't need to check that `header.prev` is non-null either. The rep invariant tells us that these will never be null, even if the list is empty. We don't need to search along the list until we get to the end, because the rep invariant tells us that the entries for a cycle, so we can go the other way instead. A poor design, without a good rep invariant, might make this simple method quite complicated.

There is a different question about this method that we'll address later, namely how we actually know that however you obtain this particular entry (whether by following the `prev` pointer from the header, or following the `next` pointer until you reach the entry just before the header), it corresponds to the last element of the list. For the method's code to be justified in this respect, we need to *interpret* the representation, and that's where the abstraction function `A` will come in.

Now let's examine a constructor that makes an empty list:

```
7.3.6   LinkedList () {  
7.3.7       header = new Entry (null, null, null);  
7.3.8       header.prev = header.next = header;  
7.3.9   }
```

Statement 7.3.7 creates a new entry with null fields, and assigns it to the header field of the new list. Statement 7.3.8 then sets its `next` and `prev` fields to point to itself. The choice of statements here is guided, again, by the rep invariant. If it dictated instead that an empty list should have no dummy entry, or should have a dummy entry with null `next` and `prev` fields, the code would be quite different.

Finally, let's look at a mutator: a method that adds an element to the end of a list:

```
7.3.10  void add (Object o) {  
7.3.11      Entry e = new Entry (o, header.prev, header);  
7.3.12      e.prev.next = e;  
7.3.13      e.next.prev = e;  
7.3.14  }
```

Statement 7.3.11 creates a new entry whose `element` field holds the new element, and which sits between the last entry of the list and the header. Note that this will still work nicely if there is no entry but the header because the rep invariant requires the header to be connected to itself in a loop in that case, so `header.prev` and `header` will evaluate to the same object. Statements 7.3.12 and 7.3.13 set the `next` and `prev` pointers of the entries on either side; if there was only a header entry, these will be fields of the same object. The result of all this pointer bashing is that whatever state the object started in, if it satisfied the rep invariant before, it is guaranteed to satisfy the rep invariant after.

What we have seen is that the rep invariant makes *modular reasoning* possible. To check whether an operation is implemented correctly, we don't need to look at any other methods. Instead, we appeal to the principle of *induction*. We ensure that every constructor creates an object that satisfies the invariant, and that every mutator and producer preserves the invariant: that is, if given an object that satisfies it, it produces one that also satisfies it. Now we can argue that every object of the type satisfies the rep invariant, since it must have been produced by a constructor and some sequence of mutator or producer applications.

## 7.4 Rep Exposure

The notion of rep invariants that we have espoused offers a systematic method for checking the correctness of abstract data type implementations. (We haven't yet considered how to ensure that a mutator or producer generates the right instance of the type, only that it produces a well-formed instance. When we study abstraction functions, we'll look at that issue.)

Our method says that we can consider the operations one by one, and then appeal to induction to show that every instance will be well formed. A crucial aspect of this method is local reasoning: we can examine the operations individually, and certainly don't need to look at client code.

In fact, this method is not always sound. It has a proviso: that the representation must not be exposed. Representation exposure is a nasty problem, because it can arise unexpectedly, and have disastrous effects that are hard to pin down. The simplest form of rep exposure involves allowing client code to manipulate the representation directly. This is easy to rule out, however, by making all fields of the abstract type private, so we don't usually even regard it as a kind of exposure.

Instead, the rep exposure we'll be concerned with arises because an object inside the representation is accessible from the outside, through a different path. Two common ways in which this happens are

- that a reference to an object that is part of the rep is passed out, as the result of an operation;
- or that an object is passed in and made part of the rep despite being accessible by an existing reference from the outside.

We saw an example of rep exposure in our last lecture. If we were so foolish as to provide a method

```
7.4.1    public Entry getEntry (int i)
```

that returns the entry at index *i* in the list, subsequent modifications to the entry could break the invariant.

A more plausible exposure, which is quite common, arises from implementing a method that returns a collection. When the representation already contains a collection object of the appropriate type, it is tempting to return it directly. For example, `List` has a method `toArray` that returns an array of elements corresponding to the elements of the list. If we had implemented the list itself as an array, we might just return the array itself. If the rep invariant requires some other part of the rep to be related to the array (eg, a size field to correspond to the index at which a null reference first appears) a modification to this array may break the invariant:

```
7.4.2    a = p. toArray (); // exposes the rep
```

```
7.4.3    a[ i] = null; // ouch! breaks invariant
7.4.4    p. get (i); // now behaves unpredictably
```

Once the invariant is violated, all hell breaks loose: subsequent operations may behave in arbitrary ways.

A more subtle variant of this problem arises with iterators. Many Java classes have a method that returns an iterator. Building an iterator is work, so we might be tempted to use one that's already provided by the Java library. Suppose our representation includes a field `vec` that holds a set of elements, and we want to implement a method

```
7.4.5    public Iterator elements ()
```

that returns an iterator over these elements. Noticing that the `Vector` class provides its own method that returns an iterator, we implement our method like this:

```
7.4.6    public Iterator elements () {
7.4.7    return vec. iterator ();
7.4.8    }
```

Unfortunately, this is a rep exposure. Classes that implement the `Iterator` interface in Java must offer `add` and `remove` methods. So the result of this method is an object that can actually be modified outside the abstract type. Since the `iterator` method of `Vector` returns an iterator that shares state with the vector it is called on, this object will be part of the state of our representation.

Another variety of rep exposure can happen between two objects of a type. This is a rather strange kind of exposure, since it involves violations due to calls to abstract operations. Suppose, for example, we have a linked list whose representation is

```
7.4.9    public class LinkedList {
7.4.10    Entry header;
7.4.11    int size;
7.4.12    class Entry {
7.4.13        Entry next;
7.4.14        Object element;
7.4.15        Entry (Entry n, Object e) {next = n; element = e;}
7.4.16    }
7.4.17    ...
```

We want a `cdr` operation (as in Scheme) that returns a list containing all but the first element. Here is a bad implementation:

```
7.4.18    public LinkedList cdr () {
```

```

7.4.19     LinkedList p = new LinkedList ();
7.4.20     p. header. next = this. header. next. next;
7.4.21     p. size = this. size -1;
7.4.22     return p;
7.4.23     }

```

We create a new list object, with its own dummy entry, and we make the next field of this dummy entry point to the second element of the original list. If we now make a call to remove on one of the lists, the rep invariant of the other list will be violated, since the size field will no longer correspond to the number of elements in the list. We should have copied the list instead.

Object models can help expose representation invariants. Two arrows pointing at the same box indicate potential sharing in the heap, and thus potential exposure. Whenever the source end of an arrow is not marked with a multiplicity, you should be concerned that sharing may lead to a rep exposure between objects of the type.

## 7.5 Element Equality & Rep Exposure

Rep exposure is actually a very subtle notion, because it is not always clear what belongs to the rep. Is it a rep exposure for list operations to return elements of the list? Let's see why it might be.

Suppose our list representation has the invariant that there are no duplicates (eg. because the list actually represents a set). Furthermore, let's say that our notion of equality is based on the contents of the elements. For example, if the elements are themselves lists, we'll regard two elements to be equal if they contain the same sequence of elements. Now we have a rep exposure: we can create duplicates simply by modifying the elements of the list from outside, making two equal when they were not equal previously.

The root of the problem here is not the passing in or out of the element objects: that can't be avoided. Rather, it's the notion of *equality*. If our set determined equality of the elements using reference equality, so that two elements are equal when they are the same object, the rep exposure would not arise. That would itself lead to problems though, since it would result in two strings that represent the same sequence of characters being regarded as distinct. The best approach, therefore, is to have the set call the equals method of the element type, and for equals of every type to be reference equality when the type is mutable.

Unfortunately, the collection classes in Java are not designed in this way. Two LinkedList objects, for example, are regarded as equal if they contain equal elements, even if they are distinct list objects. Now if we insert such lists as keys into a HashMap or Hashtable,

a subsequent modification to a list can break the hash table invariant. This can lead to very strange behaviour:

```
7.5.1    LinkedList k;  
7.5.2    Hashmap m;  
7.5.3    ...  
7.5.4    m. put (k, v); // insert the list as a key into the hash map  
7.5.5    k. add (e); // mutate the key; breaks the rep invariant of m  
7.5.6    x = m. get (k); // now x may not be v!
```

The problem is that the key is stored in a fixed slot in the hash table according to its computed hashcode. Mutating the key may change its hashcode, so that when looked up a second time, the hash table code looks in the wrong slot.

To work around this problem, you should either wrap objects before you insert them into hashtables (so that they have an equals method you define yourself), or you should make sure you never mutate keys.

## 7.6 Rep Invariants as Assertions

Many rep invariants can be translated straightforwardly into code. In our `LinkedList` implementation, for example, it's easy to check that the `prev` and `next` pointers commute, that the header is non-null, etc. Even if the check is expensive, it's worth coding it up, since expensive properties tend to be tricky ones that are more likely to be violated.

The rep invariant assertion checker can be coded as a method `checkRep` of no arguments that throws an exception if the rep invariant is violated, with a message indicating which constraint is broken. Calls to `checkRep` can be placed at the start and end of every public mutator and at the end of every constructor.

Although our induction argument suggests that placing it at the start of mutators should not be necessary, there is a risk of representation exposure which would cause the rep to change between the end of one operation and the start of the next. For an observer, the call to `checkRep` should be placed at the start, and at the end also if it changes the rep (which we'll see is actually often a reasonable thing to do, when we consider 'beneficent side effects' in the next lecture), and even if it supposedly doesn't (since you may be wrong).

If the check is very expensive, you'll probably want to comment it out or turn it off in the release version. Otherwise it makes sense to leave it in. Be careful in how you judge performance here; novices are often much too ready to worry about performance improvements that turn out to be negligible. Before dropping a check, you should have some evidence that it's expensive, such as an analysis with a profiler showing that in-

deed the check is a hotspot, or a theoretical argument, for example that the check turns a constant time operation into a linear time one.

Checking rep invariants is especially useful because it helps to localize bugs. Suppose you forget to update a size field in one of the mutator operations of a linked list. This bug will not cause problems until a subsequent operation tries to make use of size, and even then some operations may succeed. A call to get with a low index, for example, might work just fine. When an operation finally fails, the bug will be obscure. Perhaps get with a high index fails because the implementation counts back from the high end of the list, for example. It will take some debugging to discover that there is no fault with the get operation at all, but that it was passed a bad object. With checks on the rep invariant inserted, however, the bug would be noticed as soon as the offending operation executed, and the programmer's attention would be drawn to the operation that actually contains the error, not the operation that first fails.

## 7.7 Summary

Why use rep invariants? Recording the invariant may seem like extra work, but actually it saves work:

- It makes modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what's going on before you can confidently add a new method.
- It helps catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.

Moreover, choosing a strong invariant is often a good design decision:

- It tends to result in cleaner code (eg, because there are no null reference checks).
- It can help avoid errors. For example, you might allow a linked list to contain unused elements beyond the end, and use the size field to avoid running over the end, but this is asking for trouble (and will prevent the unused elements from being garbage collected).
- It can improve performance (eg, because of caching or redundancy).

You should therefore design and record the rep invariant as part of the design of the representation, before you start coding. When you're trying to understand an abstract data type, writing down the rep invariant is a good place to start.