



- (4) What happens if we walk off the end of a list? (Using the next() method.)

ADT interface answer: In Homework 4, if you invoke next() on the last node in a list, it returns null. In Homework 5, it returns an invalid node instead. There are two reasons for this change. First, it provides consistency, because invoking next() at the end of a list yields the same result as removing a node. Second, if you call a method on the result--for instance, n.next().item()--it throws an InvalidNodeException instead of a NullPointerException. This eliminates ambiguity; you can catch an InvalidNodeException without wondering why it was thrown, whereas many different bugs can cause NullPointerExceptions.

Implementation answer: Recall that our implementation uses a doubly-, circularly-linked list with a sentinel node. Any sentinel is considered an invalid node. This simplifies the implementations of the next() and prev() methods in the DList class.

However, if you apply next() to a sentinel, you won't get the first node of the list; you'll get an InvalidNodeException. Why? When n is the last node in a list, why not let n.next().next() be the first node? First, the fact that the implementation uses a sentinel should be completely hidden from the application. Second, we want to be able to change the implementation without breaking the application. Suppose we switch from DLists to SLists that don't have sentinels. We would need to "fix" SList so that n.next().next() still behaves the way it does with DLists. It's better not to allow applications to take advantage of such quirks from the start.

- (5) How do we access an item?

ADT interface answer: In Homework 4, each node's "item" field is public. In Homework 5, we make the "item" field protected; applications must use the item() and setItem() methods to access it. Why? To make sure that applications can't store items in deleted nodes or sentinels. Any attempt to invoke item() or setItem() on an invalid node causes an exception. Why? So that the implementation can be changed without breaking an application. Suppose, for instance, that an application stores items in sentinel nodes. Would the application still work the same way if you switched from DLists to SLists, which don't have sentinel nodes?

This may seem like a strange justification. But in real-world programming, programmers often take advantage of undocumented quirks, like being able to store items in sentinel nodes. Once applications have been written that depend on these quirks, the quirks become "features" that must be preserved in any new List implementation. That's why ADTs should never do `_more_` than what the documentation says they do.

In Frederick P. Brooks, Jr.'s famous book on software engineering, "The Mythical Man-Month" (page 65), he writes

Invalid syntax always produces some result; in a policed system that result is an invalidity indication `_and_nothing_more_`. In an unpoliced system all kinds of side effects may appear, and these may have been used by programmers. When we undertook to emulate the IBM 1401 [hardware] on System/360 [an operating system], for example, it developed that there were 30 different "curios"--side effects of supposedly invalid operations--that had come into widespread use and had to be considered as part of the definition. The implementation as a definition [of the functionality] overprescribed; it not only said what the machine must do, it also said a great deal about how it had to do it.

By ensuring that an implementation does not produce any result not specified in the interface--even for invalid inputs--a programmer makes it easy to fix bugs, optimize performance, and add new features without compromising existing applications.

This lecture's lesson is that design decisions can be complicated and have unexpected repercussions.

Our design decisions for the Homework 5 lists, described above, will carry over to our tree interfaces, which you'll encounter in an upcoming assignment.

One final thought. Why don't we simply keep a boolean "valid" flag in each ListNode, and use that to distinguish valid nodes from invalid ones? It would make the implementation clearer, and therefore more maintainable. However, it would also make each ListNode occupy more memory. I chose reduced memory use over readability, but this was an arbitrary choice.