CMPSCI 187: Programming With Data Structures

Lecture #17: The List ADT's
19 October 2011

The List ADT's

- Kinds of Lists: Ordered, Unordered, Indexed
- Operations Common to All Lists
- Iterators
- Adding to an Ordered List
- Adding to an Unordered List
- Polymorphism Again

Kinds of Lists: Ordered, Unordered, Indexed

- An **ordered list** is one where the elements are sorted according to some meaningful rule. We need a definition of what it means for one element to be "less than" another -- then we insist that every element of the list is "less than or equal to" the one that comes after it.

- Examples of orderings are alphabetical order on a String field, numerical order on a numerical field, earliest or latest time entering the list (making a queue or stack), or some combination of factors.

- If a list is **unordered**, the elements still come in some order because the list is a linear structure. But the order has no particular meaning, and the structure does not have to preserve any properties of it.

- An **indexed list** is one where every element is numbered, as in an array.

Operations Common to All Lists

- The `removeFront` and `removeRear` operations of a deque reappear in L&C's `ListADT` interface, under the names `removeFirst` and `removeLast`. The `first` and `last` operations occur under the same names, as do `isEmpty`, `size`, and `toString`.

- General lists have a new operation `remove`, which takes a potential element as a parameter and removes it from the list, also returning it. Although their interface listing does not say so, `remove` throws an `ElementNotFoundException` if the parameter element is not in the list.

- We can avoid throwing this exception by first running the method `contains`, which returns a boolean telling whether the parameter element is in the list.

Iterators

- The final operation we have that is common to all lists is to produce all of its elements one at a time, without changing the list itself. Java offers a particular format for this called an **iterator**.

- The `iterator` method of a list returns a new `Iterator` object. This object implements the `Iterator` interface, which has the two methods `public boolean hasNext( )` and `public T next( ).` The first tells whether there are elements yet to be returned, and the second returns the next element in sequence.

- You are never guaranteed that an iterator will give you elements in a particular order, only that it will give you each element exactly once.

- What should happen if another process alters the list while an iterator exists?

Adding to an Ordered List

- If we have an ordered list and a new element, there is only one place where the new element can go and keep the list ordered. (If there are elements "equal" to the new one already in the list, the new one could go anywhere among them, but in that case it probably doesn't matter which is which.)

- Thus the `OrderedListADT` interface has only one `add` method, which takes an element as parameter and inserts that element into the correct position. This interface extends `ListADT`, and has only the one new operation.

- But `OrderedListADT` assumes that there is a comparison operation on the elements, because its implementations will need to use the `compareTo` method of the base class. We could insist that the class of the objects implements the `Comparable` interface, but actually it's more complicated.

Adding to an Unordered List

- Let's first look at the add operations for an unordered list. Since it is now legal to add an element anywhere in the list, we can now have the `addToFront` and `addToRear` operations that we had with deques.

- We also have a method `addAfter` that takes an extra parameter `target`, an element that is already in the list (if the target is not there we get an `ElementNotFoundException`). The new element goes into the list directly after the target element.

- All these operations also make sense for indexed lists, along with additional ones to add or remove elements at particular indices. Note that when we add or remove an element, lots of indices may need to change to fill in the gap or to make room for the new element.

Polymorphism Again

- These list interfaces are generic, so that the interface `OrderedListADT` defines types `OrderedListADT<T>` for any class `T`, in principle. But we can't have an ordered list without a way to compare elements.

- One way to handle this would be to replace the type parameter `<T>` with `<T extends Comparable<T>>`. This would mean that the generic structure would only work when T is a class that supports the `compareTo` operation with other T elements.

- Just having a list of `Comparable` objects isn't good enough, because implementing `Comparable` just means that you have a `compareTo` operation, not that you can always compare one such object to another.

How We Deal With Comparability

- In their implementations of the ordered list classes, L&C take the same approach as the classes in Collections. Each interface or class is generic, and defined for every possible T.

- When we want to compare two T elements, we cast one of them into a `Comparable<T>` and put it in a variable of type `Comparable<T>`. If this works, the compiler will know that we can compare this element to a T element. If it does not work, we will get a `ClassCastException`.

- This means that the classes we put into ordered lists will generally implement the Comparable interface for themselves. If we do this, the casts will always work, and the code outside of the classes will be simpler.