Computer Science 210
Data Structures
Siena College
Fall 2016

Topic Notes: Ordered Structures

We have considered two special-purpose data types, stacks and queues, that are essentially *restricted* versions of the more general structures we considered earlier.

While we implemented our stacks and queues using arrays and vectors and linked lists, the interfaces to these linear structures limited access to the internal representation, and allowed us to choose an appropriate way to orient the data within the structures to make the operations in the restricted interface as efficient as possible. Moreover, this restriction meant we, as designers and authors of the data structure, could explicitly *prevent* a user of the structure from accessing or modifying it in an unexpected way.

Ordered Structure Concept

Now, we consider structures that have another restriction placed on them: that their contents are maintained in some order.

These *ordered structures* allow us to search among their contents efficiently, or to process the contents in a particular order.

These can be implemented using the structures we know so well, but again we will want to restrict the interface so as to guarantee that the ordered nature of the structures is not violated.

These is one additional complication here: if we are going to order the objects in our structures, we need a mechanism for comparing them.

Earler, we saw two approaches to allow comparison of Java objects: We could require that the objects implement the `Comparable` interface, or we could require that an appropriate `Comparator` class be provided for the objects.

Recall that `Comparable` is a Java interface that requires a method:

```
public int compareTo(T item);
```

and `Comparator` is a Java interface that requires a method:

```
public int compare(T item1, T item2);
```

Let's consider how the `Comparable` interface and `Comparator` objects might be of use in defining objects that can be placed into an ordered structure. In particular, let's begin by considering a *Comparable Association*.

It is an extension of the `Association` class from way back that also implements `Comparable`, therefore adding a `compareTo` method. Recall that `Associations` are key/value pairs. For a `ComparableAssociation`, we require that the key be `Comparable`, so the ordering of the `ComparableAssociation` is inherited from the ordering of the `Comparable` keys.

See Structure Source:
`structure5/ComparableAssociation.java`

These `ComparableAssociations` may be compared and placed in an ordered structure.

Implementations

We will implement two ordered structures, one based on a `Vector` and the other on a linked list.

In structure, each of these implements an interface called `OrderedStructure`.

See Structure Source:
`structure5/OrderedStructure.java`

It's an empty interface! What good is this? All it does it defines a type. But that means we can use that type in places where we want to require one of our ordered structures, but do not want to commit to a particular implementation (as when we wanted a `List` but did not want to commit to a specific one).

But since it does extend the `Stucture` interface, it requires our basic set of operations.

See Structure Source:
`structure5/Structure.java`

But in this case, we (as implementers) will enforce the restriction on implementations that the contents will be stored in order.

Ordered Vectors

We'll first consider an `OrderedVector` of `Comparable` objects.

As we did with the linear structures, we don't extend the underlying data type, but rather *encapsulate* it.

So use a regular `Vector` as the underlying representation, but we *restrict the interface* to enforce that our structure remain ordered.

See Structure Source:
`structure5/OrderedVector.java`

What are the complexities of the methods here?

- `contains` can make use of a binary search! Well, that was the whole point, wasn't it? But this is good! We now have a structure with an $O(\log n)$ `contains` method.
- `add` now requires a search for the proper position at which to add. We use an $O(\log n)$

binary search. Plus there is a worst-case $O(n)$ cost to move everything up beyond the add position.

- `remove` can use a binary search as well, again $O(\log n)$ to find the position of the item to be removed, followed by a worst-case $O(n)$ cost to shift down the contents of the `Vector`.

An important question here is why did we not extend `Vector` instead of having one protected inside the class? Our answer is that the public interface to the `Vector` class is not restrictive enough! Since the `OrderedVector` would also *be* a `Vector` with all of its public methods, users could modify the structure with general-purpose `Vector` operators and break the ordering!

Again, we need to **restrict** the functionality to ensure that our structure functions correctly and that it can be made to perform its public functionality more efficiently.

Ordered Lists

Which of our list implementations make sense for our list-based `OrderedStructure`?

Consider the operations allowed. We need only search from the beginning and add/remove values at arbitrary positions. The doubly linked and circular lists are no better at these than a singly linked list, so it makes sense to go with the simplest one that works.

We could implement this with a protected `SinglyLinkedList`, just as we did with the protected `Vector` inside of our `OrderedVector`.

But think about how we'd have to do for `add`. We would need to create an iterator over the list to compare the object we're adding with each object in the list. Then we'd know where to add it. But adding it would require a new search all the way from the beginning! That's inefficient.

So we want to break open the `SinglyLinkedList` and use some of its internals without using the whole thing. Essentially our `OrderedList` will implement its own list by using the same `Node` structure that is used in `SinglyLinkedList`. But we'll manage the details differently in `OrderedList`. Fortunately, we have a very restrictive interface, so there are not many methods to worry about.

So we'll have a counted singly linked list that keeps itself ordered.

See Structure Source:
`structure5/OrderedList.java`

Unfortunately, our important operations are still $O(n)$. Our linked list does not allow direct access to arbitrary elements, forcing us to settle for a linear search when finding the correct position for an object being added or removed or searched.

Adding an optional `Comparartor`

An additional feature of this implementation is that it allows use of a `Comparator` for alternate orderings of our data. In fact, it does in a way that allows it to work without modification if you wish to order `Comparables` by their "natural" ordering, but will allowing alternate orderings using a `Comparator`.

What was added to or modified in the ordered structures to support this?

1. An instance variable to store the comparator. The very odd syntax for the type parameter here means that we can specify a `Comparator` for anything that `E` is – any of the classes it extends or interfaces it implements. So long as it can compare objects of type `E`.
2. Add a new constructor that takes an appropriate `Comparator` as its parameter.
3. Modify the default constructor to create and use a `NaturalComparator` – a simple `Comparator` that just uses the required `compareTo` method of our `Comparable` objects.
4. Change the `compareTo` calls to `compare` calls.

Our structure is actually a bit overrestrictive. We require that the elements we add extend `Comparable`, even though we'll only use their `compareTo` method when using the `NaturalComparator`.