SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2018

Topic Notes: Ordered Structures

We have considered special-purpose linear structures 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

Let's think back to our table of efficiencies of our list structures. In all cases, remember what entries looked like for the `contains` method:

- $O(1)$ in the best case, when we happen to find what we're looking for right away
- $O(n)$ in the worst case, when we either find what we're looking for at the end, or the item is not there at all

This makes sense, since in each of these cases we must employ a linear search, and that has the complexities above, regardless of the data structure that stores the values.

But we have also seen that there is a more efficient way to search. If the data can be sorted, we can use a binary search, and achieve a worst case of $O(\log n)$ to find (or not) the element we're looking for.

So now, we consider structures that have a different kind of restriction placed on them to permit a binary search: that their contents are maintained in some order.

Structures that support more efficient searching are a theme for a few of the data structures that remain for us to study this semester.

In addition to having the potential to support a binary search, these *ordered structures* also make it easy (sometimes even trivial) to traverse their contents in sorted order.

The restriction that contents are maintained in order means the only `add` operation would add wherever necessary to maintain ordering, but such structures could support `remove` by value operations (but not likely a `get` by index and definitely not a `set` by index).

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.

Consider also how structures restricted in this way can help avoid errors. With linear structures, the restrictions can help a programmer ensure that data that is intended to be accessed only in LIFO or FIFO ordering is only accessed that way – any attempt to use other operations would trigger an error at compile time. With an ordered structure, errors can also be avoided. Sure, a program could use an `ArrayList` or linked list and it would be up to anyone developing or modifying that code to remember that the contents of the structure must be kept ordered any time it is modified. By using a restricted ordered structure, it would not be possible to modify the structure in a way that breaks that ordering.

Determining an Ordering

These is one additional complication here, which can also be seen as a further restriction: if we are going to order the objects in our structures, we need a mechanism for comparing them.

We have seen approaches that 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 `Association`s 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 `ComparableAssociation`s may be compared and placed in an ordered structure.

Naive Implementations of Ordered Structures

We will initially consider implementations of two ordered structures, one based on a `Vector` and the other on a linked list. For simplicity, these structures require that only objects of a `Comparable` type be stored, but we will see in the list version that an option to support a `Comparator` is also possible.

Ordered Vectors

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

As we did with the linear structures, we *encapsulate* the underlying data type. A `Vector` is used 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`.

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` (which is the same as our `SimpleListNodes` we know well). 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. Certainly not the most useful structure we've developed.

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 `Comparable`s 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.

This 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`.

Doing Better

We can do better for ordered structures by moving away from our arrays, vectors, and linked lists. Back at this topic soon once we have started studying data structures with branches.