Iterators cont. and Ordered Data Structures

Instructors: Sam McCauley and Dan Barowy

April 11, 2022

Last week's quiz solution

- Reminder: goal is to duplicate a stack of `Integer`s using one additional stack

Last week's quiz solution

```java
public static void dupStack(Stack<Integer> s, Stack<Integer> newS) {
    Stack<Integer> temp = new StackList<Integer>();
   while(s.size() != 0) {
      temp.push(s.pop());
   }
   //now temp has the contents of s in reverse order
   while(temp.size() != 0) {
      int value = temp.pop();
      s.push(value);
      newS.push(value);
   }
}
```

Comments

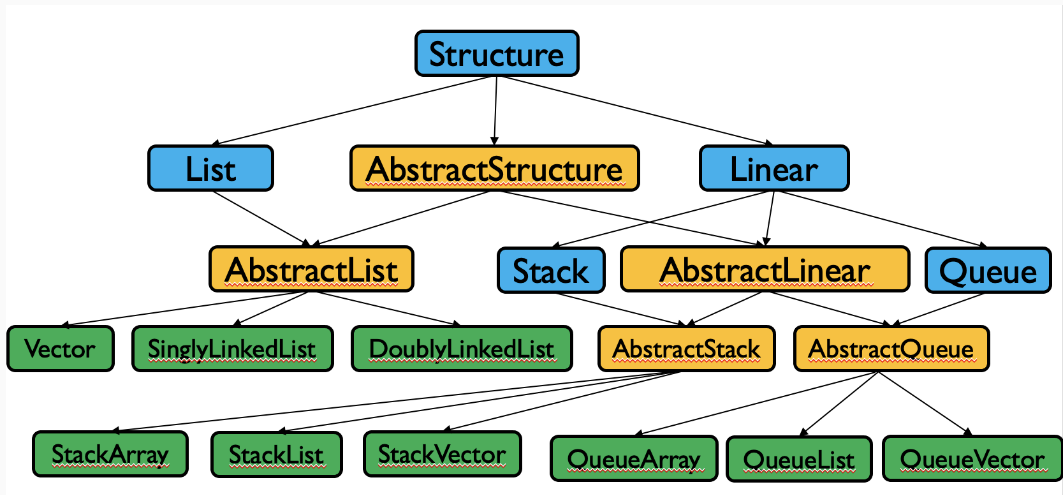
- Can also use `peek()`
- Using a for loop with `s.size()` doesn't work
  - a for loop checks its condition (and in this case calls `size()`) each loop iteration
  - As we pop items from `s` its size chagnes
  - It is possible to make a for loop work; just can't have something like `i < s.size()` as the condition.
- Could also use recursion
- One of the hardest quizzes so far, so we tried to give some partial credit. (And bear in mind that quizzes are averaged with attendance at the end of the day.)

Iterator Wrap Up

Let's make a practice iterator

- Can we make an iterator that takes *another* iterator as an argument, and gives its elements in reverse order? (OK if "destroys" original iterator using `next()` calls)
  - Would need to store them
  - What data structure would be best to use?

Iterable data structures



`Structure<E>` extends `Iterable<E>`, so all of these are iterable

(Arrays are too)

for each loops

```java
int[] grades = { 100, 78, 92, 87, 89, 90 };
int sum = 0;
for (int g : grades)
  sum += g;
```

- For-each loops work using iterators!
- Can do the above with any `Iterable` data structure

for each loops

```
Stack<String> strStack = new StackList<String>();
strStack.push("0");
strStack.push("1");
strStack.push("2");
strStack.push("3");
strStack.push("4");
for (String s : strStack)
   System.out.println(s);
```

Let’s look at the `StackList` code that’s used here.

Care about iterators

- Like for-each loops, iterators work best on data structures that are not changing

- Be very careful about changing the data structure while iterating over it!

Generalizing iterators

- Bear in mind: *anything* with a `hasNext()` and `next()` method is an iterator
- It's often used for iterating over a data structure, but doesn't have to be
- Let's make an iterator that prints the Fibonacci numbers

Ordered Structures

Keeping data in order

- Our data structures so far have not been optimized for searching
- In practice, we often keep items in sorted order
- We know how to sort, and we know how to maintain items in a list
- How can be build a data structure that maintains the items in sorted order?

A comment about the `OrderedVector` data structure

- This is not a particularly useful data structure long term (there are better options)
- You probably will not ever use this in practice the way we're describing it here
  - (Of course, you'll use sorted arrays and sorted vectors, etc. But usually that's for situations where you insert everything and sort once; not updating the sorted order for each new item.)
- Goal for today: more practice using OOP; first example of an ordered data structure; more discussion of sorting in Java
- `structure5` code is a bit less polished for `OrderedVector` than for other classes

How to sort?

- We've seen two ways to sort items: using comparable items, or using a comparator
    - `Comparable<E>`: iterface with a `compareTo()` method (takes one argument). Used for class types that have an *intrinsic* order, like integers or strings
    - `Comparator<E>`: interface with a `compare()` method taking two arguments. Has much more flexibility in how to compare items—but need to make a class to specify how you want them compared
- We'll use comparable items today. So: the data structure we're looking at can only store class types `E` that implement `Comparable<E>`

Goal: Ordered Vector

- Want to create a vector of (comparable) items, such that the items are always in sorted order
- We'll also add in an efficient method to locate items. What algorithm should we use?
  - Binary search! It has $O(\log n)$ running time.
- How do we create our class? Two choices:
  - Create a subclass using the `extends` keyword (as we did with `MyVector`) (*is-a* relationship)
  - Create a class that *contains* a `Vector` and interfaces with it using the public vector methods (as we did with `StackVector`) (*has-a* relationship)
  - What are some advantages of each?
- We'll do the second: maintain a `Vector`
  - Helps us have more control over how someone accesses the vector. For example, we don't really want a `set()` method

Beginning the `OrderedVector` class

```
public class OrderedVector<E extends Comparable<E>>
                  implements OrderedStructure<E> {

   protected Vector<E> data;

   public OrderedVector() {
      data = new Vector<E>();
   }

   public void add(E value) {
      int pos = locate(value);
      data.add(pos, value);
   }
```

Implementing `locate()`

- Finds an item in an `OrderedVector` using binary search
- We'll be using an iterative version of binary search (not recursive)
- Recall the invariant of binary search:
- If the item we're looking for is in the array, it is located somewhere within `low...high`

Locate

```
protected int locate(E target) {
   Comparable<E> midValue;
   int low = 0; // lowest location
   int high = data.size(); // highest location
   int mid = (low + high)/2; // low <= mid <= high
   while (low < high) {
      midValue = data.get(mid);
      if (midValue.compareTo(target) < 0) {
         low = mid + 1;
      }
      else {
         high = mid;
      }
      mid = (low+high)/2;
   }
   return low;
}
```

Filling in the rest of `OrderedVector`

- Now that we have `locate()` the rest is pretty easy!
- We already used `locate()` to fill in `add`
- Let's use `locate()` to fill in `contains()` and `remove()`

Final OrderedVector Methods

```
public boolean contains(E value) {
   int pos = locate(value);
   return pos < size() && data.get(pos).equals(value);
}
public Object remove (E value) {
   if (contains(value)) {
      int pos = locate(value);
      return data.remove(pos);
   }
   else {
      return null;
   }
}
```

These can be found in the `structure5` `OrderedVector` class.

OrderedVector Performance

- Locate?
  - $O(\log n)$
- Add?
  - $O(n)$: locate is $O(\log n)$, but shifting items down is $O(n)$. So overall $O(n + \log n) = O(n)$.
- Contains?
  - $O(\log n)$ (just a call to locate and $O(1)$ extra work)
- Remove?
  - Like add: locate, and then remove (shifting items down as necessary); $O(n)$.

OrderedList

- Let's talk through how to implement an ordered Linked List (say a `SinglyLinkedList`)
- How can we binary search in a singly linked list? What's the challenge of doing so?
- Idea of binary search: we compare the item we are searching for to the middle element in the range `low...high` (using a call to `get()`)

Locating in a Linked List

- How long does finding `get(mid)` take in a linked list?
- $O(n)$ just to find *one* mid item
- We can show: $O(n)$ time for `locate()` in total
- Takeaway: ordered a linked list does not lead to faster search!
- The `OrderedList` class is still included in `structure5` however

A Note of Care About Ordered Structures

- This issue is common to all the structures we use that keep items in some order based on their contents
- No good way around it
- Problem: we need to assume that the objects do not change without telling the `OrderedVector` about it
- Let's look at an example

Sorting Students by Grade

- We can easily change the `Student` class to allow comparison by age
- Then we can store students in an ordered list by age
- Let's look at an example
- What happens when the age changes?
- Answer: `OrderedVector` doesn't know the age changes, so doesn't stay sorted

An Example that *Does* Work

- Let's store a list of associations between the population of a county and the percentage of people who voted third party in the 2020 election
- So we'd like an `OrderedVector<Association<Integer, Double>>`
- Wait a minute—the `OrderedVector` can only store things that implement `Comparable`. But `Association` doesn't implement `Comparable`
- The type of the key (`Integer`) does implement `Comparable`, however
- Enter: the `ComparableAssociation`. (Some of you may have used this in lab 5.)

ComparableAssociation summary

```
public class ComparableAssociation<K extends Comparable<K>,V>
          extends Association<K,V>
          implements Comparable<ComparableAssociation<K,V>>, Map.Entry<K,V>
{
    public int compareTo(ComparableAssociation<K,V> that)
    {
        return this.getKey().compareTo(that.getKey());
    }
}
```

(This is an example of a class that `implements` two different interfaces. We'll talk about `Map.Entry` in 3 or so weeks.)

Finishing Our Example

- We can store an `OrderedVector` of `ComparableAssociation<Integer,Double>`
- But, what happens when we change one of the `ComparableAssociations`?
- In particular, what happens when the *population* of one of the counties changes? (I.e. we change the key?)
- Answer: `Association` does not allow us to change the key
- Takeaway: if you're storing a class type in an ordered data structure, control access so that the sorted order cannot change
    - If possible