Collections and Iterators

Collection ADTs

- Most of the ADTs we'll see are for storing many objects and accessing/modifiying elements in some way
- We discussed Set yesterday
- One we've already seen, but not named is List
- There are lots of similarities between all of these ADTs, so in Java there is a `Collection` interface that they all implement

List ADT

- A list stores a collection of elements **in order**
- The ADT requires these operations
  - `T get(int i)` — get element at index i
  - `void set(T t, int i)` — replace the element at index i
  - `void add(T t, int i)` — add an element at index i, shifting from that spot and after to the right by one spot
  - `void remove(int i)` — remove the element at spot i, shifting anything after it left one spot
- You've seen the `ArrayList` data structure which uses an array for this

ArrayList implementation

- Since Java arrays are fixed size, growing/shrinking an arraylist is a bit tricky
- If we store an array that's the exact size we need, every time we add/remove an element we need to create a new array and copy all the elements over (what's the big O complexity of that?)
- An efficient implementation is to allocate more than we need, then make a bigger array if we every run out of space
- A common growth strategy is to double the array when we need to grow

Array List complexity

- What's the complexity of get/set?
- Add/remove are much trickier
- It depends on WHERE we add. Which index is fastest? Slowest?
- If we add to the back, it's usually fast, but sometimes it's actually really slow!
- Usually it's O(1), but sometimes it's O(N)!
- How do we analyze something like that?

Amortized analysis

- One way to handle the “usually fast, but sometimes slow” analysis it to consider an “averaged” version
- Consider calling add N times. We count up the total number of operations. To get an “average” cost of a single operation, we divide the total by N
- This is called the “amortized” complexity
- Amortization is a financial term related to spreading the cost of something over a period of time
- Here we're “spreading” the cost of an add where we have to grow the array across the ones were we don't

Amortized analysis of add to end

- The reason some adds are expensive is that we need to copy elements to a new bigger array
- To do an amortized analysis, we'll try to count the total number of times elements get copied to new arrays as we perform N adds, starting from an empty array
- Let's assume the Nth add completely fills our final array for simplicity
- Half of the elements have never been copied since they were added after the last time we grew the array
- 1/4 were copied once because they were added before the 2nd to last growth operation
- 1/8 were copied twice, ...

Analysis continued

- Total number of copies is 0*N/2 + 1*N/4 + 2*N/8 + 3*N/16 + ...
- You can see that the denominators grow WAY faster than the numerators, so the terms get small fast and we get a total value which is O(N)
- So if the total number is O(N), when we divide by N to get an amortized cost for each add, we get O(1)!
- Intuitively, we grow the array so infreqently that even though it's expensive the average cost of an add is still very low
- Note, that if added a fixed number of elements rather than multiplying when growing we'd actually get $O(N^2)$ amortized cost!!
- The analysis still works if we pick a different scale factor, and some implementation use 1.5 instead of 2 for complicated reasons

Missing feature: Iteration

- For any collection it's useful to be able to iterate through all the elements
- For different ADTs, the order we get the elements can be very different, but for lists's it's clear we want them in the “index” order
- In Java all collections support this by implementing the `Iterable<t>` interface
- It has one method: `Iterator<t> iterator()`
- An `Iterator` is an object which can go through a collection

Iterator

```
public interface Iterator<t> {
    boolean hasNext();
    T next(); //return "next" element and "advance" the iterator
    void remove(); //remove the element that the most recent call to next returned
    //must call next() again before we call remove() again
}
```

You've used this already

- Foreach loops use iterators which is why they work for library types (ie not just built-in arrays)
- `for(var x : collection)` is translated into the following:

```
var iterator = collection.iterator()
while(iterator.hasNext()){
    var x = iterator.next();
    //body here
}
```

- If your collection type implements `Iterable` you can use it in a foreach loop... pretty cool!

Implementing iterators

- An iterator is often implemented as an inner class so it can keep a reference to the collection it's a part of
- For an array based iterator, we typically just need to store the index of the element we'll return when we call `next()`
- The `iterator()` method is usually just `return new MyInnerIteratorClass()`
- There's some optional methods in the `Iterator` interface, but in this class we just need to implement `hasNext()`, `next()`, and `remove()`

Tricky stuff: remove

- If we include the remove operation, the trickiest part is remembering to “back up” the iterator when we call remove so that we don't skip anything when we call `next()` again
- Also it's important to make sure we throw an exception if the user calls `remove()` twice without `next()` in between