Higher-order Sequential Operations

Many of the operations we wish to perform over lists have common structure. In this chapter, we investigate the most common of these patterns and how we can abstract them into powerful, generic list manipulation algorithms.

Iteration

The most common operation we’ll perform over a List is the traversal. We’ll walk the list, doing something to each element, e.g., finding the largest element, summing up all the elements, or transforming all the elements. To do this, we would normally write a for-loop:

```java
List<T> list = /* ... */;
for (int i = 0; i < list.size(); i++) {
    ... list.get(i) ...
}
```

If our list is an array list, then this loop has linear time complexity. We walk the list, performing a constant time get operation at each. However, if the list is a linked list, then this loop has quadratic time complexity because get takes linear time for a list.

This seems silly, though. We know that we can implement such a traversal in linear time with a linked list by doing a standard linked list traversal and holding onto the current node rather than the current index. However, outside the linked list, this is impossible because the Node class is internal to the LinkedList.

We need some mechanism for abstracting the way that we traverse a particular list, whether that list is an array list or linked list. This design pattern in programming is called iteration and in Java, we use an Iterator object to perform the traversal.

We can think of an iterator as a cursor. Initially, the iterator points at the beginning of the list, and we can fetch the element that the cursor currently references. The List interface of the Java standard library provides a method, iterator(), that produces such an iterator object. Here is an example of using an iterator object to walk a List:

```java
List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

The two critical operations of the Iterator interface are:

- boolean hasNext(): returns true if the cursor is pointing to an element of the list, i.e., there are still elements left to process.
- T next(): retrieves the element currently under the cursor and advances the cursor one position.
We can combine these two operations using a while-loop to perform precisely the same behavior as the for-loop above. In addition to this solution being more concise than a traditional for-loop, less can “go wrong” with this code. There are no indices to worry about—indeed, the iterator object provides exactly the set of operations we need to perform a forward traversal and nothing more. However, in return, the iterator can only move forward, among other restrictions. While other kinds of iterators that Java provide backwards movement (e.g., the ListIterator class), in general we cannot capture all of the patterns of iteration we might want to perform, e.g., processing every second or third element. Furthermore, using the basic iterator, we cannot modify the underlying list through the iterator object.

**Iterator Implementation**

How is the Iterator interface implemented? We could perform the iteration using methods of the List interface, but as we noted at the top of this reading, this makes linked list traversal quadratic instead of linear. Therefore, we must implement the Iterator interface differently for ArrayList and LinkedList. The actual Iterator interface from the Java standard library requires three methods:

- **T next():** returns the next element in the iteration.
- **boolean hasNext():** returns true iff the iteration has more elements.
- **void remove():** removes from the underlying collection the last element returned by this iterator (optional operation).

The remove gives our iterators slightly more flexibility by allowing us to remove elements from the list while iterating over it. However, removal of an element may not be supported by the class that implements the Iterator interface. Our list data structures (ArrayList and LinkedList) support remove, but we might want to iterate over something whose elements cannot be directly modified, e.g., an immutable List. In this case, remove will throw an UnsupportedOperationException.

To see how to implement the Iterator interface for each of ArrayList and LinkedList, it is useful to appeal to the code for traversing either structure. For example, for ArrayList, we might traverse the elements as follows:

```java
for (int i = 0; i < size; i++) {
    ... data[i] ...
}
```

(Note that we iterate up to the size of the backing array rather than the length of the array because these slots from indices 0 to (size - 1) are the currently “active” ones.) The most important part of this loop is how we keep track of the iteration. In this case, we simply keep track of our current index. Likewise, the iterator for an array list, call it ArrayListIterator, would need to store the current index it is on as state, i.e., a field. To check to see if it has a next element, the iterator checks that the current index is less than the array list’s size. To get the next element and advance the iterator we use array indexing and increment the current index.

In contrast, a linked list traversal looks as follows:
Node<T> cur = first;
while (cur != null) {
    ... cur.value ...
}

Unlike the array list, we must keep track of the current node that we are on and advance the iterator by advancing our “current” reference. By doing this, traversing our linked list with an iterator takes linear time as desired.

For-each Loops

Because traversal with an iterator is so common in Java (and in general, programming), the language provides special syntax for this called the for-each loop:

List<String> list = /* ... */;
for (String str : list) {
    ... str ...
}

A for-each loop looks somewhat similar to a for-loop. However, a for-each loop only has two components inside of the parenthesis. We first specify a variable that each element of the list is bound to while the loop runs, and then we specify the list that we are traversing. The above snippet of code is equivalent to this snippet:

List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    ... str ...
}

In fact, when you use a for-each loop, the Java compiler produces this while-loop! You can think of the for-each loop as syntactic sugar for doing this sort of traversal.

We can use a for-each loop on anything that produces an iterator. Java checks for this by seeing if a class implements the Iterable interface. The Iterable interface requires one method of its implementors:

- Iterator<T> iterator(): returns an iterator to the “beginning” of this object.

This is precisely the method we use from the List interface to grab our iterators. If you want your own classes to be usable in a for-each loop, then you must:

- Implement a class that in turn implements the Iterator that implements traversal behavior for that class.
- Have your class implement the Iterable interface. The iterator() method you define should return an instance of the class’s corresponding Iterator sub-class.
Streams and Higher-order Functions

So far, we have studied sequential structures of fixed-size, arrays, and variable-size, lists. Note that these variable-sized lists, while possessing the capacity to grow, are ultimately finite-sized. That is, the size of the list is always some finite value (although it can grow as necessary). With this in mind does it make sense to study sequences with a potentially infinite amount of values? Such a structure seems unusable! We identified with iterators that the most common operation over a sequence is the traversal, but how can we traverse a structure with an infinite number of values?

Indeed, this is impractical to do; the best we can do is sample finite prefixes of the infinite sequence. What we’ll do instead is rather than traversing the structure repeatedly to perform operations over it, we’ll build up a collection of computations that will fire only when we ask for elements from the stream. This lazy model of computation where our transformations over the data do not occur until it is necessary is not just useful for these infinite structures, they are also useful for finite structures like lists.

0.0.1 Streams

A stream is a potentially infinite sequence of values. For example, network traffic from the Internet or input from the user can be thought of as a stream because we do not know if the data from the source (the Internet or the user) will ever end. Finite sequences can naturally be thought of as a stream, i.e., lists or files. In particular, very, very large sequences of data, say in the gigabytes or terabytes, are effectively infinite streams as we cannot practically traverse them using our standard list machinery.

In Java 8, the stream package contains a number of classes to create and manipulate streams. In particular, the Stream class represents this abstract data type. Suppose that we wish to read in data from a file as a Stream object. To represent this data as a Stream, we have two choices: (1) read in the data into a list and then make a stream out of the list or (2) read in the data as a stream directly.

The first approach should be familiar to you. In Java, we use the Scanner class to read in a file line by line, word by word, etc.. We can then read the data into a list. From there, we can use the stream() method of the List class to create a Stream object that reads from the list.

```java
// throws FileNotFoundException
Scanner src = new Scanner(new File("data.txt"));
List<String> data = new LinkedList<>();
while (src.hasNextLine()) {
    // Assuming that one datum appears per line in the file...
    data.add(src.nextLine());
}
Stream<String> stream = data.stream();
```

Note that the Stream class, like the List, is generic in the carrier type of the Stream.

The alternative approach uses more Java 8 features to perform this line-by-line reading concisely:
// throws IOException
Stream stream = Files.lines(Paths.get("data.txt"));

The Files class exposes a static method `lines(path)` that creates a stream to a file that will
process it line-by-line. We must pass the method a relative path to the file in question represented
by a Path object. We create a path through the static get method of the Paths class.

There are trade-offs to using each approach. With the first approach, we must traverse the
entire data to put it all into list. We then affix a stream onto the list which acts essentially as an
iterator, a pointer to the front of the list. With the second approach, we do not actually sample
from the data source, i.e., the list until it is necessary avoiding the up-front cost of traversing the
List. On the other hand, it is easier to perform parsing and pre-data processing on the data using
scanners than trying to turn all of that parsing into transformations on streams (although it is
possible).

Stream Processing with Higher-order Functions

With a stream in hand, we do not traverse it to perform transformations and analysis on the data.
We instead build up a pipeline of operations to perform to the stream. This includes three big
ticket operations on streams:

- `Stream<U> map(Function<T, U> f)`: transforms the elements of the stream by applying the
  function `f` to each element.
- `Stream<T> filter(Function<T, Boolean> f)`: filters the stream, keeping all the elements
  of the stream that produce `true` when given to the function `f`
- `U reduce(U init, BiFunction<U, T, U> f)`: also called `fold`, reduces the stream to a sin-
  gle value by starting with the given initial value and then applying `f` to each element,
  accumulating a final result in the process.

For example, we can count the number of occurrences of strings in our stream the start with an
“h”, ignoring case with:

```java
stream.map(s -> s.toLowerCase())
  .filter(s -> s.startsWith("h"))
  .count()
```

The call to `map` makes all the strings lower case. The call to `filter`, removes all strings that do
not start with “h”. Note that the `map` occurs first in our method chain, guaranteeing that all the
strings are lower case. Finally, we then count the number of strings that are left.

After building up this pipeline of operations, we can either use summarizing functions like
`count` or `reduce` to collapse the stream into a single value. Alternatively, the `Stream` class possesses
an `iterator()` method that we can use to get an Iterator into the stream and perform the final
traversal. Note that both summarizing functions and the iterator `consume` the stream when used.
To re-analyze the data, we must create a new stream to the data and use that for further analysis.