Chapter 6

Sequential Structures

The first of the fundamental data types that we'll study are *sequential types*. A sequential type is simply a container for data where the container maintains an ordering between elements. As a result, a common operation we can perform over a sequential types is to *iterate* over the elements of the container in-order.

The canonical example of a sequential type built-in to most languages is the array. Arrays have the following properties in addition to being sequential:

- *Homogeneous*. An array holds elements of the same type.
- *Fixed-size*. Once created, an array's size cannot change.

Other sequential types differ along these two dimensions. For example, a heterogeneous fixed-size sequential type is called a *tuple*. We frequently utilize tuples of two elements, *pairs*, and tuples of three elements, *triples*. Tuples are built-in to some languages, but not Java. So we have to define a class for each tuple type that we're interested in using, *e.g.*, a `Point` class:

```
public class Point {
    private int fst;
    private int snd;
    public Point(int fst, int snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public int getFst() { return fst; }
    public int getSnd() { return snd; }
}
```

6.1 The List

Frequently, we do not know the number of elements we need to store into a sequence up front, for example, because the input comes from the user during execution of the program. Fixed-sized types are insufficient in these situations. We instead need a *variable-sized* type that can grow at runtime to accommodate our storage needs.

6.2 Linked- and Array-based Lists

6.3 Amortized Analysis

Recall the basic set up for an array list and its `add` method:

```
public class ArrayList<T> {
    private Object[] arr;
    private int front;
    public ArrayList() {
        arr = new Object[16];
        front = 0;
    }
    private void ensureCapacity() {
        if (front >= arr.length) {
            arr = Arrays.copyOf(arr, arr.length * 2);
        }
    }
    public add(T v) {
        ensureCapacity();
        arr[front++] = v;
    }
}
```

Analyzing the time complexity of `add` is tricky because its behavior does not follow the standard patterns we've seen up to this point. In particular, there are two cases that dictate the perform of `add`:

1. If `front < arr.length`, then there is room in the backing array for the new element. Adding the element onto the end of the list takes $\mathcal{O}(1)$ time since it consists of an array copy and an increment.
2. If `front >= arr.length`, then we must make room for the new element in the backing array. We do so by creating a new array that is twice the size, copy the elements over, and then copy in the new element. This takes $\mathcal{O}(n)$ time (where n is the number of elements in the array).

For an individual `add` operation, the best case is the former case and the worst case is the latter case. However, what is the average case? We usually define the average case in terms of the shape of the input, *e.g.*, assuming that the element that `indexOf` searches for lies in the middle of the list. Here, there is no "average" shape here—either the backing array is full or it isn't. This makes characterizing average difficult.

Certainly, calling the average case $\mathcal{O}(n)$ in line with the worst case is not incorrect (we are giving upper-bounds, after all), but such a bound isn't consistent with our intuition as to how `add` works. By doubling the array size on every growth, we hope that we will perform many more $\mathcal{O}(1)$ additions than $\mathcal{O}(n)$ additions, effectively making the runtime $\mathcal{O}(1)$. Our current definition of time complexity do not support this idea of analyzing the runtime of a method over many invocations—it only considers a single operation in isolation. Therefore, we need additional machinery to analyze `add` faithful.

This machinery is called *amortized analysis* where we look at not just a single call to `add`, but many calls to `add` to underhand its "average" complexity. There are three different basic techniques for performing amortized analysis:

- *The aggregate method* which directly characterizes the total complexity of a series of operations and then declares the average complexity to be the average of this amount.
- *The accounting method* which charges cheap operations additional time in order to pay for the time debt incurred by more expensive operations.
- *The potential method* which builds a potential function that describes the "time" built-up by cheap operations that can be used by more expensive operations. Unlike the accounting method, the potential function is only based on the current state of the data structure, not the history of operations that led to that point.

Here, we will use the accounting method to analyze `add` because it most closely works with our intuition that the cheap `add` operations should "pay" for the expensive `add` operations.

Consider an array list with starting capacity two and a series of add operations on it along with the number of array accesses they perform:

Add Op	Actual Cost
1	1
2	1
3	3
4	1
5	5
6	1
7	1
8	1

The third and fifth additions induce array doubling (corresponding to when the backing array is at two and four elements, respectively). Each of these perform $k + 1$ array accesses where k is the number of elements at the time of the copy plus one additional copy for the new element. The remaining additions perform a single array access.

The goal of the banker's method is to assign a single cost to all of the `add` operations c, the *charge cost*, such that the charge cost allows us to pay for the actual cost of successive add operations without going negative. For example, suppose we consider $c = 1$ which corresponds to charging each `add` operation one array access.

Add Op	Actual Cost	Charge Cost	Balance
1	1	1	0
2	1	1	0
3	3	1	**-2**

The balance column is the amount of cost left over after using the balance and charge cost to pay for the actual cost. Because the charge cost is one, the first two operations are paid for without leaving any balance left over. Unfortunately, this means that when we try to pay for the third operation, our balance is empty, so our charge cost of 1 is not enough to pay the actual cost (3) of the operation. This means that $c = 1$ does not work as a charge cost for `add`.

Next, let's try $c = 2$:

Add Op	Actual Cost	Charge Cost	Balance
1	1	2	1
2	1	2	2
3	3	2	1
4	1	2	2
5	5	2	**-1**

Note that by charging 2 array accesses per `add`, we build up enough of a balance to pay off the first expensive addition. However, the charge cost is not sufficient to pay for the second expensive addition.

Finally, let's try $c = 3$:

Add Op	Actual Cost	Charge Cost	Balance
1	1	3	2
2	1	3	4
3	3	3	4
4	1	3	6
5	5	3	4
6	1	3	6
7	1	3	8
8	1	3	10
9	9	3	4

With $c = 3$, we have enough for the second array copy (at the fifth operation) However, this charge cost also gives us enough left-over balance to pay for the third array copy (at the ninth operation), too! Furthermore, if we look at the left-over balance after each array copy operation, we see that we are left with 4, so it seems like this pattern might continue indefinitely. How can we prove this is the case?

Consider the cost of `add` when we copy an array. If the array contains `n` elements, we must pay one array access to copy each old element along with one addition array access to copy the new element. From this, we can justify our choice of $c = 3$ for each element added to the array

- 1 array access goes towards the actual insertion of an element.
- 1 array access goes towards the first time this element is copied into a larger array.
- 1 array access goes towards the insertion of one of the elements in the lower half of the array.

Note that if the currently copied array is size n, then $n/2$ elements have been added to the array since the last copy. The final charge accounts for the copying of the older $n/2$ into the new array.

With $c = 3$, we see that we can assign `n` a constant amount of time that is accurate over many operations of `add`. Therefore, we conclude that `add` has *amortized* $\mathcal{O}(1)$ complexity.