CS61B Lecture 17

Monday, March 3, 2020

Data Types in the Abstract

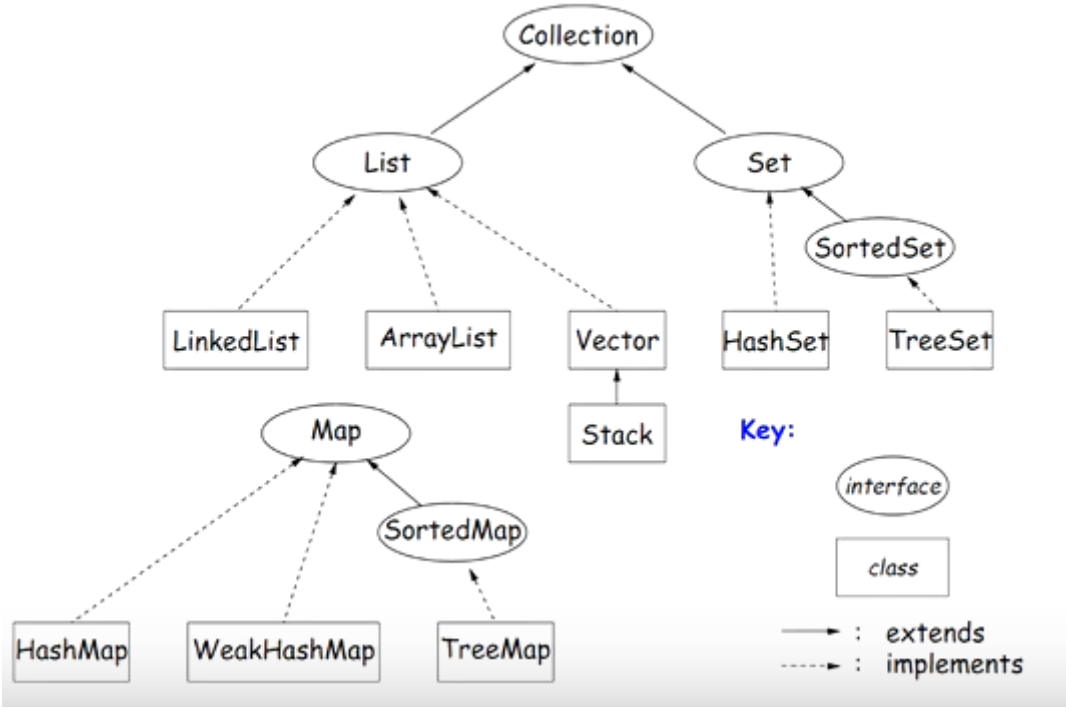
Most of the time, you should not need to worry about the implementation of data structures, search etc. What is important is what they do for us -- their specification.

Java includes several standard types in `java.util` to represent collections of objects:

- Six interfaces
  - `Collection`: General collections of items
  - `List`: Indexed sequences with duplication
  - `Set`, `SortedSet`: Collections without duplication
  - `Map`, `SortedMap`: Dictionaries (key > value)
- Concrete classes that provide actual instances: `LinkedList`, `ArrayList`, `HashSet`, `TreeSet`

To make change easier, purists would use the concrete types only for `new`, interfaces for parameter types and local variables. (i.e. design methods that take in the interface rather than the concrete type if you can)

As it turns out, because of the way interfaces are implemented in Java, this is a little slower (around 10%) than designing more specific methods, but usually it won't matter.



The Collection Interface

In the `Collection` interface, the following main functions are promised:

- Membership tests: `contains`, `containsAll`
- Other queries: `size`, `isEmpty`

- Retrieval: `iterator`, `toArray`
- Optional modifiers: `add`, `addAll`, `clear`, `remove`, `removeAll` (set difference), `retainAll` (intersect)

Optional Operations

Not all `Collection`s need to be mutable: often it makes sense just to get things from them. Some operations are therefore optional (`add`, `addAll` etc.)

The library developers decided to have all `Collection`s implement this, but allowed implementations to throw an `UnsupportedOperationException`.

An alternative design would have created separate interfaces:

```
interface Collection { contains, containsAll, ... }
interface Expandable extends Collection { add, addAll }
interface Shrinkable extends Collection { remove, removeAll, ... }
interface Modifiable extends Collection, Expandable, Shrinkable { }
```

You'd soon have lots of interfaces: too many even.

The List Interface

The `List` interface extends `Collection`, and is intended to represent indexed sequences. It also features new methods to those of `Collection`:

- Membership tests: `indexOf`, `lastIndexOf`
- Retrieval: `get(i)`, `listIterator()`, `sublist(B, E)`
- Modifiers: `add` and `addAll` with additional index to say where to add. Likewise, for removal operations. There is also a `set` operation to go with `get`.

The `ListIterator<Item>` extends `Iterator<Item>`:

- Adds `previous` and `hasPrevious`
- `add`, `remove`, and `set` allow one to iterate through a list: inserting, removing, or changing as you go.
- What advantage is there to saying `List L` rather than `ArrayList L` or `LinkedList L`.

Implementing `ArrayList`

The main concrete types in Java library for the `List` interface are `ArrayList` and `LinkedList`:

As you might expect, an `ArrayList` A uses an array to hold data. For example, a list containing the three items 1, 4, and 9 might be represented like this:

After adding four more items to `A`, its `data` array will be full, and the value of `data` will have to be replaced with a pointer to a new, bigger array that starts with a copy of its previous values.

For the best performance, how big should this new array be?

If we increase the size by 1 each time it gets full, or by any constant value, the cost of `N` additions will scale as

$$\Theta(N^2)$$

which makes `ArrayList` look much worse than `LinkedList` (which works like `IntList`).

However, this is a misleading analysis because we can make `ArrayList` s fast as `LinkedList`, and here's how:

Expanding Vectors Efficiently

When using array for an expanding sequence, it is best to double the size of the array every time we grow it, and here's why:

If our array is size `s`, doubling its size and moving `s` elements to the new array takes time proportional to 2`s`. In all cases, there is an additional

$$\Theta(1)$$

cost for each addition to account for actually assigning the new value into the array.

When you add up these costs for inserting a sequence of `N` items, the total cost turns out to be proportional to `N`, as if each addition took constant time, even though some of the additions actually take time proportional to `N` all by themselves!

In this case, it is not helpful to focus on the worst-case time, because the worst-case time is always `N` time, but to focus on the total sequence of operations, which is proportional to `N` as well.

Amortized Time

Suppose the actual costs of a sequence of `N` operations are

$$c_0, c_1, \ldots, c_{N-1}$$

which may differ from each other by arbitrary amounts and where

$$c_i \in O(f(i))$$

Consider another sequence

$$a_0, a_1, \ldots, a_{N-1}$$

where

$$a_i \in O(g(i))$$

If

$$\sum_{0 \le i < k} a_i \ge \sum_{0 \le i < k} c_i, \forall k$$

we say that the operations all run in

$$O(g(i))$$

amortized time, where amortized is derived from the French word *amortir*, meaning "to death". It means that the actual cost of a given operation,

$$c_i$$

may be arbitrarily larger than the amortized time,

$$a_i$$

as long as the total amortized time is always greater than or equal to the total actual time, no matter where the sequence of operations stops -- no matter what k is.

In cases of interest, the amortized time bounds are much less than the actual individual time bounds:

$$g(i) << f(i)$$

For example, in the case of insertion with array doubling,

$$f(i) \in O(N) \text{ and } g(i) \in O(1)$$

Amortization: Expanding Vectors (II)

Observe the below table and the general formula presented at the bottom:

To insert item #	Resizing cost	Cumulative cost	Resizing cost per item	Array size after insertions
0	0	0	0	1
1	2	2	1	2
2	4	6	2	4
3	0	6	1.5	4
4	8	14	2.8	8
5	0	14	2.33	8
...	...	...	...	...

To insert item #	Resizing cost	Cumulative cost	Resizing cost per item	Array size after insertions
7	0	14	1.75	8
8	16	30	3.33	16
...	...	...	...	...
15	0	30	1.88	16
...	...	...	...	...
2^(m)+1 to 2^(m+1)-1	0	2^(m+2)-2	~2	2^(m+1)
2^(m+1)	2^(m+2)	2^(m+3)-2	~4	2^(m+2)

If we spread out (amortize) the cost of resizing, we average at most about 4 time units resizing each item: "amortized resizing time is 4 units". The time to add N elements is now

$$\Theta(N)$$

and not

$$\Theta(N^2)$$

Demonstrating Amortized Time

To formalize the argument, associate a potential,

$$\Phi_i \geq 0$$

to the i-th operation that keeps track of "saved up" time from cheap operations that we can "spend" on later expensive ones. Start with

$$\Phi_0 = 0$$

Now, we pretend the cost of the i-th operation is actually

$$a_i$$

the amortized cost, defined as

$$a_i = c_i + \Phi_{i+1} - \Phi_i$$

where

$$c_i$$

is the actual cost of the operation. Rearranged through potential:

$$\Phi_{i+1} = \Phi_i + (a_i - c_i)$$

On cheap operations, we artificially set

$$a_i > c_i$$

so that we can increase

$$\Phi$$

On expensive ones, we typically have

$$a_i << c_i$$

and greatly decrease

$$\Phi$$

without letting it dip into negatives such that it is not "overdrawn", as in our bank account metaphor.

We try to do all this so that

$$a_i$$

remains as we desired without allowing

$$\Phi_i < 0$$

which requires that we choose our value of amortized cost such that the potential always stays ahead of the real cost.

Application to Expanding Arrays

When adding to our array, the real cost of adding element I when the array already has space is 1 unit. The array however does not initially have space when adding items 1, 2, 4, 8, 16... -- in other words, at powers of 2 for all positive n. So:

$$c_i = \begin{cases} 1 & \text{if i } \geq 0 \text{ and i} \neq 2^k, \forall k \in \mathbb{Z}^+ \\ 2i+1 & \text{if i } = 2^k, \forall k \in \mathbb{Z}^+ \end{cases}$$

Each operation at a power of 2, we are going to need to have saved up at least

$$2 \times 2^n = 2^{n+1}$$

units of potential to cover the expense of expanding the array, and we have this operation and the preceding

$$2^{n-1} - 1$$

operations in which to save up this much potential (everything since the preceding doubling operation).

So we choose:

$$a_0 = 1$$
$$a_i = 5, \forall i > 0$$

Apply our potential formula and:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a_i	1	3	5	1	9	1	1	1	17	1	1	1	1	1	1	1	33	1
c_i	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Phi_i	0	0	2	2	6	2	6	10	14	2	6	10	14	18	22	26	30	2

Pretending each cost is 5 *never* underestimates the true cumulative time.