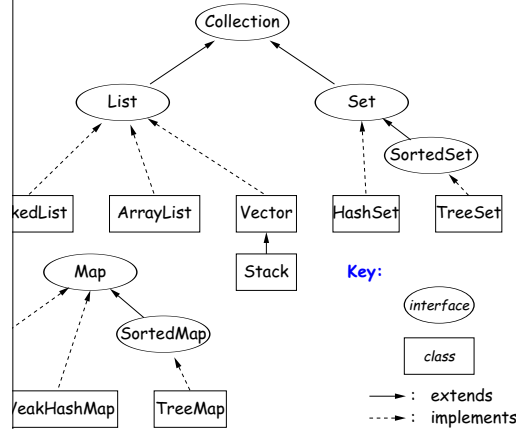


Topics

standard Java Collections classes.
List, ListIterators
Stacks and maps in the abstract
Analysis of implementing lists with arrays.

Collection Structures in java.util



about Library Design: Optional Operations

Operations need to be modifiable; often makes sense just to have them from them.
Optional operations are optional (add, addAll, clear, remove, removeAll,

developers decided to have *all* Collections implement the same methods to throw an exception:

UnsupportedOperationException

Alternative design would have created separate interfaces:

```
Collection { contains, containsAll, size, iterator, ... }
Expandable extends Collection { add, addAll }
Modifiable extends Collection { remove, removeAll, ... }
ImmutableCollection
```

Alternative design would have created separate interfaces.
They have lots of interfaces. Perhaps that's why they didn't do it that way.

CS61B Lecture #17

Data Types in the Abstract

When designing an abstract data type, you should *not* worry about implementation of data structures like arrays, linked lists, etc.

What matters for us—their specification—is important.

General standard types (in java.util) to represent collections:

Categories:

Collection: General collections of items.

List: Indexed sequences with duplication

Set: Collections without duplication

Map: Dictionaries (key \mapsto value)

Concrete classes that provide actual instances: LinkedList, ArrayList, Stack, TreeSet.

When designing an abstract data type, you should change easier, purists would use the concrete types only and interfaces for parameter types, local variables.

The Collection Interface

Collection interface. Main functions promised:

Membership tests: contains (\in), containsAll (\subseteq)

Statistics: size, isEmpty

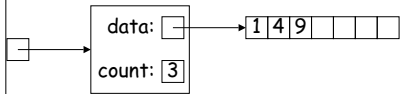
Iteration: iterator, toArray

Modifiers: add, addAll, clear, remove, removeAll (set difference), retainAll (intersect)

Implementing Lists (I): ArrayLists

Concrete types in Java library for interface List are ArrayList and LinkedList:

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



If you add four more items to A, its data array will be full, and the data will have to be replaced with a pointer to a new array that starts with a copy of its previous values.

For best performance, how big should this new array be? If you double the size by 1 each time it gets full (or by any constant factor), the cost of N additions will scale as $\Theta(N^2)$, which makes ArrayList look much worse than LinkedList (which uses an amortized implementation.)

Amortized Time

If the actual costs of a sequence of N operations are c_1, \dots, c_N , which may differ from each other by arbitrary amounts, then the total cost is $\sum_{i=1}^N c_i \in O(f(i))$.

For a given operation i , we define its amortized cost a_i as the average cost of that operation over the sequence a_0, a_1, \dots, a_{N-1} , where $a_i \in O(g(i))$.

$$\sum_{0 \leq i < k} a_i \geq \sum_{0 \leq i < k} c_i \text{ for all } k,$$

then the operations all run in $O(g(i))$ amortized time.

The amortized cost of a given operation, a_i , may be arbitrarily greater than or equal to the total actual time, but the sequence of operations stops—i.e., no matter what the actual cost is, the amortized time is bounded.

Of interest, the amortized time bounds are much less than the actual time bounds: $g(i) \ll f(i)$.

In the case of insertion with array doubling, $f(i) \in O(N)$ and $g(i) \in O(1)$.

Calculating Amortized Time: Potential Method

For a given operation i , if the actual cost is c_i and the amortized cost is a_i , then the difference $a_i - c_i$ is the "potential" Φ_i . We associate a potential, $\Phi_i \geq 0$, to the i^{th} operation that keeps track of "saved up" time from cheap operations that can be "spent" on later expensive ones. Start with $\Phi_0 = 0$.

We define the cost of the i^{th} operation to be a_i , the amortized cost, defined as:

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

where Φ_i is the potential before the operation. Or, looking at potential:

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

For a given operation, we artificially set $a_i > c_i$ so that we can increase the potential (i.e., $\Phi_{i+1} > \Phi_i$).

In most cases, we typically have $a_i \ll c_i$ and greatly decrease Φ so that it goes negative—may not be "overdrawn".

To bound all this so that a_i remains as we desired (e.g., $O(1)$ for array doubling), without allowing $\Phi_i < 0$.

If we choose a_i so that Φ_i always stays ahead of c_i .

The List Interface

Section

represent indexed sequences (generalized arrays)

Methods to those of Collection:

Utility methods: indexOf, lastIndexOf.

Methods: `get(i)`, `listIterator()`, `subList(B, E)`.

Methods: `add` and `addAll` with additional index to say where to add; `remove` for removal operations. `set` operation to go with `set`.

`Iterator<Item>` extends `Iterator<Item>`:

Methods: `hasNext` and `hasPrevious`.

Methods: `add`, `remove`, and `set` allow one to iterate through a list, inserting, removing, or changing as you go.

Question: What advantage is there to saying List L is an ArrayList L or LinkedList L?

Expanding Vectors Efficiently

For a given sequence, the best way to double the size of the array is to double the size of the array. Here's why.

If you double the size of the array, doubling its size and moving s elements to the new array is proportional to $2s$.

There is an additional $\Theta(1)$ cost for each addition to the array, so the total cost is actually proportional to $2s + N$.

If you add up these costs for inserting a sequence of N items, it 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!

Amortization: Expanding Vectors (II)

	Resizing Cost	Cumulative Cost	Resizing Cost per Item	Array Size After Insertions
	0	0	0	1
	2	2	1	2
	4	6	2	4
	0	6	1.5	4
	8	14	2.8	8
	0	14	2.33	8
	:	:	:	:
	0	14	1.75	8
	16	30	3.33	16
	:	:	:	:
	0	30	1.88	16
	:	:	:	:
	0	$2^{m+2} - 2$	≈ 2	2^{m+1}
-1	2^{m+2}	$2^{m+3} - 2$	≈ 4	2^{m+2}

By amortizing the cost of resizing, we average at most 2 units for resizing on each item: "amortized resizing cost." Time to add N elements is $\Theta(N)$, not $\Theta(N^2)$.

Application to Expanding Arrays

When adding element $#i$ to our array, the cost, c_i , of adding element $#i$ when ready has space for it is 1 unit.

Does not initially have space when adding items 1, 2, 4, 8, ... in other words at item 2^n for all $n \geq 0$. So,

$c_i \geq 0$ and is not a power of 2; and

When i is a power of 2 (copy i items, clear another i items, then add item $#i$).

At operation $\#2^n$ we're going to need to have saved up at least 2^{n+1} units of potential to cover the expense of expanding and we have this operation and the preceding $2^{n-1} - 1$ operations which to save up this much potential (everything since the last doubling operation).

$c_1 = 1$ and $a_i = 5$ for $i > 0$. Apply $\Phi_{i+1} = \Phi_i + (a_i - c_i)$, and happens:

4	5	6	7	8	9	10	11	12	13	14	15	16	17	
9	1	1	1	17	1	1	1	1	1	1	1	33	1	<i>Pretending each cost is</i>
5	5	5	5	5	5	5	5	5	5	5	5	5	5	<i>5 never underestimates</i>
6	2	6	10	14	2	6	10	14	18	22	26	30	2	<i>true cumulative time.</i>