CS61B Lecture #17

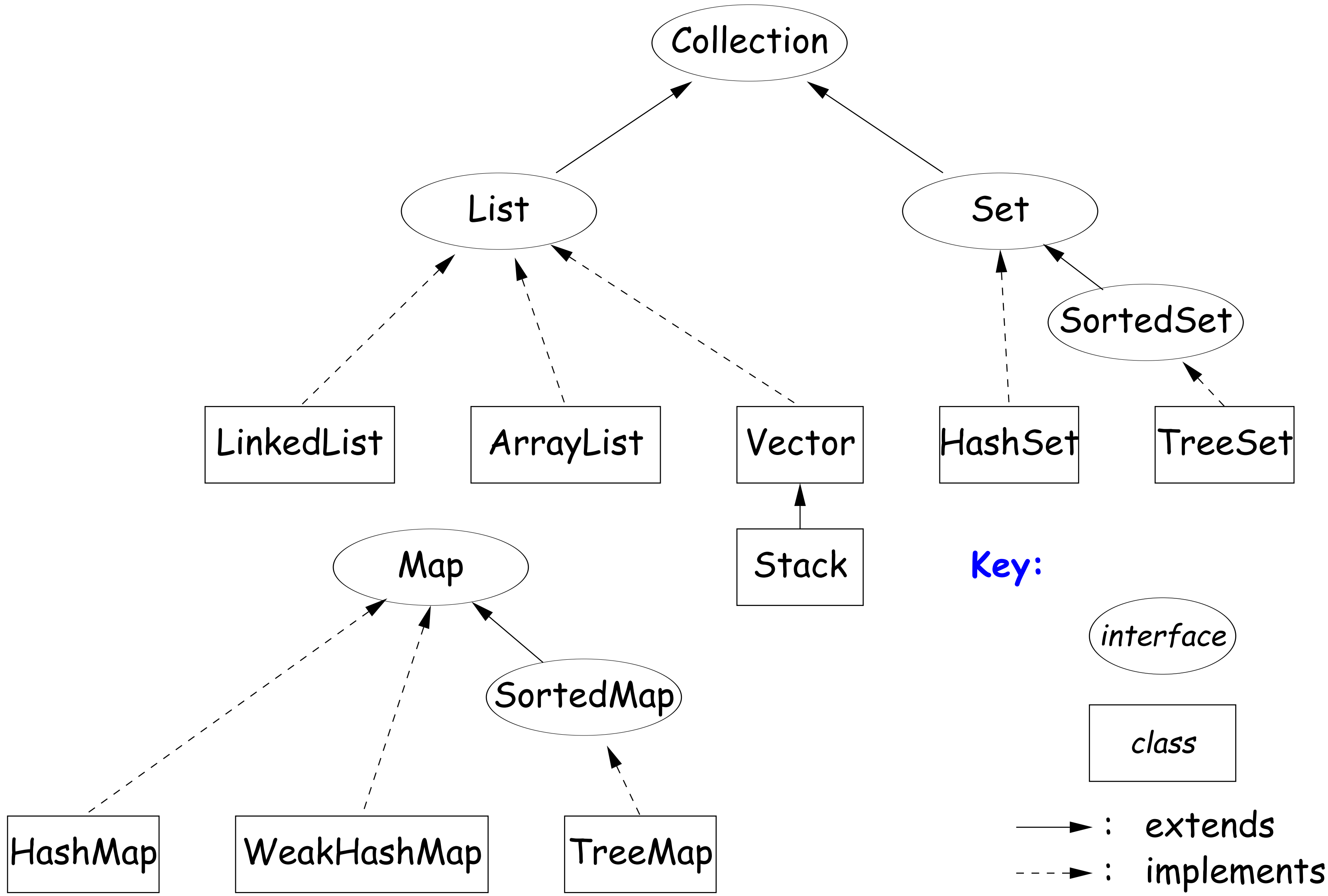
Topics

- Overview of standard Java Collections classes.
  - Iterators, ListIterators
  - Containers and maps in the abstract
- Amortized analysis of implementing lists with arrays.

Data Types in the Abstract

- Most of the time, should *not* worry about implementation of data structures, search, etc.
- What they do for us—their specification—is important.
- Java has 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 $\mapsto$ 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, local variables.

Collection Structures in java.util



The Collection Interface

- Collection interface. Main functions promised:
  - Membership tests: `contains` ($\in$), `containsAll` ($\subseteq$)
  - Other queries: `size`, `isEmpty`
  - Retrieval: `iterator`, `toArray`
  - *Optional* modifiers: `add`, `addAll`, `clear`, `remove`, `removeAll` (set difference), `retainAll` (intersect)

Side Trip about Library Design: Optional Operations

- Not all `Collections` need to be modifiable; often makes sense just to get things from them.
- So some operations are optional (`add`, `addAll`, `clear`, `remove`, `removeAll`, `retainAll`)
- The library developers decided to have *all* Collections implement these, but allowed implementations to throw an exception:

  `UnsupportedOperationException`

- An alternative design would have created separate interfaces:

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

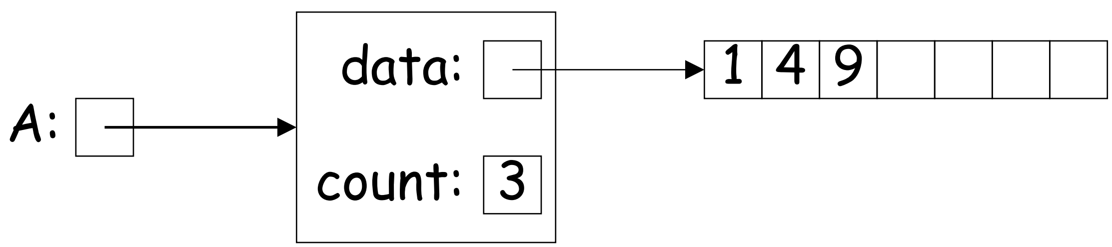
- You'd soon have lots of interfaces. Perhaps that's why they didn't do it that way.

The List Interface

- Extends `Collection`
- Intended to represent *indexed sequences* (generalized arrays)
- Adds 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. `set` operation to go with `get`.
- Type `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.
  - **Important Question:** What advantage is there to saying `List L` rather than `LinkedList L` or `ArrayList L`?

Implementing Lists (I): ArrayLists

- The main concrete types in Java library for interface `List` 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.
- Question: For 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 uses an `IntList`-like implementation.)

Expanding Vectors Efficiently

- When using array for expanding sequence, best to *double* the size of array to grow it. Here's why.
- If array is size s, doubling its size and moving s elements to the new array takes time proportional to $2s$.
- 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!

Amortized Time

- Suppose that 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

$$\sum_{0 \le i < k} a_i \ge \sum_{0 \le i < k} c_i \text{ *for all* } k,$$

  we say that the operations all run in $O(g(i))$ *amortized time*.
- That is, 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—i.e., no matter what k is.
- In cases of interest, the amortized time bounds are much less than the actual individual time bounds: $g(i) \ll f(i)$.
- E.g., for the case of insertion with array doubling, $f(i) \in O(N)$ and $g(i) \in O(1)$.

Amortization: Expanding Vectors (II)

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
⋮	⋮	⋮	⋮	⋮
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 for resizing on each item: "amortized resizing time is 4 units." Time to add N elements is $\Theta(N)$, *not* $\Theta(N^2)$.

Demonstrating Amortized Time: Potential Method

- 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 that the cost of the i^{th} operation is actually a_i, the *amortized cost*, defined

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

where c_i is the real cost of the operation. Or, looking at 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 Φ ($\Phi_{i+1} > \Phi_i$).
- On expensive ones, we typically have $a_i \ll c_i$ and greatly decrease Φ (but don't let it go negative—may not be "overdrawn").
- We try to do all this so that a_i remains as we desired (e.g., $O(1)$ for expanding array), without allowing $\Phi_i < 0$.
- Requires that we choose a_i so that Φ_i always stays ahead of c_i.

Application to Expanding Arrays

- When adding to our array, the cost, c_i, of adding element #i when the array already has space for it is 1 unit.
- The array does not initially have space when adding items 1, 2, 4, 8, 16,...—in other words at item 2^n for all $n \geq 0$. So,
  - $c_i = 1$ if $i \geq 0$ and is not a power of 2; and
  - $c_i = 2i + 1$ when i is a power of 2 (copy i items, clear another i items, and then add item #i).
- So on each operation #2^n we're going to need to have saved up at least $2 \cdot 2^n = 2^{n+1}$ 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 choose a_0 = 1 and $a_i = 5$ for $i > 0$. Apply $\Phi_{i+1} = \Phi_i + (a_i - c_i)$, and here is what happens:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c_i	1	3	5	1	9	1	1	1	17	1	1	1	1	1	1	1	33	1
a_i	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Φ_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 true cumulative time.