

## CS61B Lecture #17

[There are no Lecture #16 notes.]

### Administrative:

- Reminder: auto-grader run sometime tonight.

### Today:

- Asymptotic complexity (from last time)
- Iterators, ListIterators
- Containers and maps in the abstract
- Views

Readings for Today: *Data Structures*, Chapter 2.

Readings for next Topic: *Data Structures*, Chapter 3.

## Some Intuition on Meaning of Growth

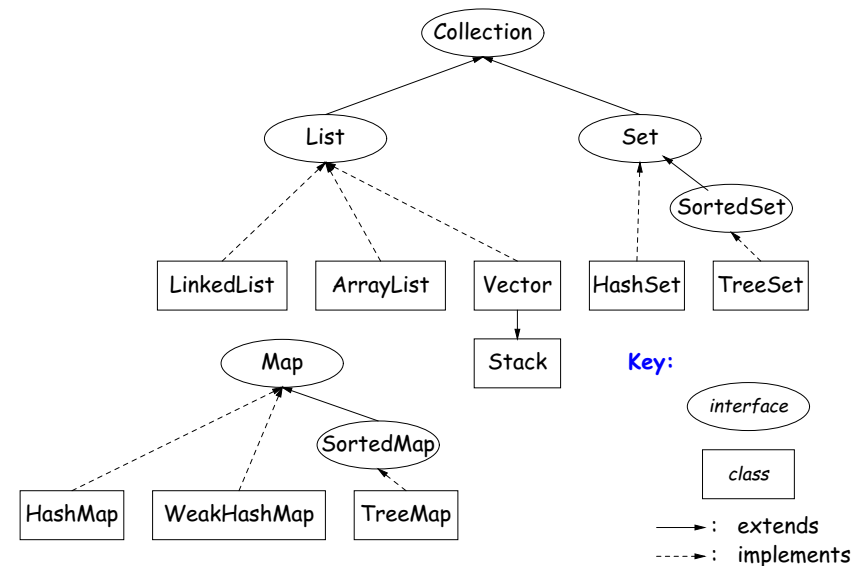
- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size  $N$ .
- Entries show the *size of problem* that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.
- $N$  = problem size

Time ( $\mu\text{sec}$ ) for problem size $N$	1 second	Max $N$ Possible in 1 hour	1 month	1 century
$\lg N$	$10^{300000}$	$10^{1000000000}$	$10^{8 \cdot 10^{11}}$	$10^{9 \cdot 10^{14}}$
$N$	$10^6$	$3.6 \cdot 10^9$	$2.7 \cdot 10^{12}$	$3.2 \cdot 10^{15}$
$N \lg N$	63000	$1.3 \cdot 10^8$	$7.4 \cdot 10^{10}$	$6.9 \cdot 10^{13}$
$N^2$	1000	60000	$1.6 \cdot 10^6$	$5.6 \cdot 10^7$
$N^3$	100	1500	14000	150000
$2^N$	20	32	41	51

## 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)

- Design point: Optional operations may throw

UnsupportedOperationException

- An alternative design would have separate interfaces:

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

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

## Problem: How to Retrieve?

- Collections don't always have an order—no first, no  $n^{\text{th}}$ , no get.
- So how to get things out?
- Even for types of Collection that *do* have an ordering, indexing (as for arrays) not always best (fastest) way to get elements.
- Abstraction to the rescue: define retrieval interface:

```
package java.util;
public interface Iterator {
    /** True iff there's more. */
    boolean hasNext ();
    /** Return next item and then move on. */
    Object next ();
    /** Remove last item returned by next() from underlying
     * Collection. May throw exception if unsupported. */
    void remove ();
}
```

- Iterator is a kind of "moving finger" through a Collection.

## 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 extends Iterator:
  - Adds previous and hasPrevious.
  - nextIndex gives position in list.
  - add, remove, and set allow one to iterate through a list, inserting, removing, or changing as you go.

## Example of Use: Reverse a File

**Problem:** Print the lines of a file in reverse order.

```
BufferedReader r = ...; // Some source of lines
List items = new LinkedList ();
for (String s = r.readLine (); s != null; s = r.readLine ())
    items.add (0, s); // Add to front
for (int i = 0; i < items.size (); i += 1)
    System.out.println (items.get (i));
```

- **Disadvantage:** On a LinkedList, get( $k$ ) is a  $\Theta(k)$  operation, leading to  $\Theta(N^2)$  algorithm, for lists of size  $N$ .

## Faster Reversal

- The `iterator` method is intended to return an iterator that is tuned to the data structure, and generally  $O(1)$  in time.
- With ordered collection (like `List`), iterator is also ordered.

```
BufferedReader r = ...; // Some source of lines
List items = new LinkedList ();
for (String s = r.readLine (); s != null; s = r.readLine ())
    items.add (0, s);
for (Iterator i = items.iterator (); i.hasNext ();)
    System.out.println (i.next ());
```

Last modified: Mon Oct 7 00:15:22 2002

CS61B: Lecture #17 9

## Example of Use II: Inserting New Elements

**Problem:** After first instance of one object, insert a new object.

```
/** Insert OBJ after EXISTING in L. */
static void insertAfter (List L, Object obj, Object existing)
{
    for (ListIterator i = L.listIterator (); i.hasNext ();) {
        Object x = i.next ();
        if (existing.equals (x)) {
            i.add (obj);
            break;
        }
    }
}
```

- **Question:** How about this implementation:

```
int k = L.indexOf (existing);
if (k != -1)
    L.add (k+1, obj);
```

- **Important Question:** What advantage is there to saying `List L` rather than `LinkedList L` or `ArrayList L`.

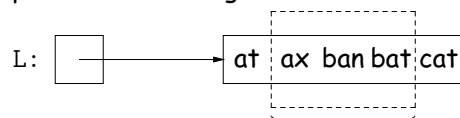
Last modified: Mon Oct 7 00:15:22 2002

CS61B: Lecture #17 10

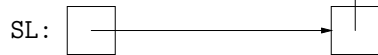
## Views

**New Concept:** A *view* is an alternative presentation of (interface to) an existing object.

- For example, the `sublist` method is supposed to yield a "view of" part of an existing list:



```
List L = new ArrayList();
L.add ("at"); L.add ("ax"); ...
List SL = L.sublist (1,4);
```



- **Example:** after `L.set(2, "bag")`, value of `SL.get(1)` is "bag", and after `SL.set(1, "bad")`, value of `L.get(2)` is "bad".
- **Example:** after `SL.clear()`, `L` will contain only "at" and "cat".
- **Small challenge:** "How do they do that?!"

Last modified: Mon Oct 7 00:15:22 2002

CS61B: Lecture #17 11

## Maps

- A `Map` is a kind of "modifiable function:"

```
package java.util;
public interface Map {
    Object get (Object key); // Value at KEY.
    Object put (Object key, Object value); // Set get(KEY) -> VALUE
    ...
}
-----
Map f = new TreeMap ();
f.put ("Paul", "George"); f.put ("George", "Martin");
f.put ("Dana", "John");
// Now f.get ("Paul").equals ("George")
// f.get ("Dana").equals ("John")
// f.get ("Tom") == null
```

Last modified: Mon Oct 7 00:15:22 2002

CS61B: Lecture #17 12

## Map Views

```
public interface Map { // Continuation
    ...
    /* VIEWS */
    /** The set of all keys. */
    Set keySet ();
    /** The multiset of all values */
    Collection values ();
    /** The set of all (key, value) pairs */
    Set entrySet ();
}
```

### Using example from previous slide:

```
for (Iterator i = f.keySet ().iterator (); i.hasNext (); )
    i.next () ==> Dana, George, Paul

for (Iterator i = f.values ().iterator (); i.hasNext (); )
    i.next () ==> John, Martin, George

for (Iterator i = f.entrySet ().iterator (); i.hasNext (); )
    i.next () ==> (Dana,John), (George,Martin), (Paul,George)

f.keySet ().remove ("Dana"); // Now f.get("Dana") == null
```