

CS61B Lecture #15

Administrative:

- Test Review Session
- 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

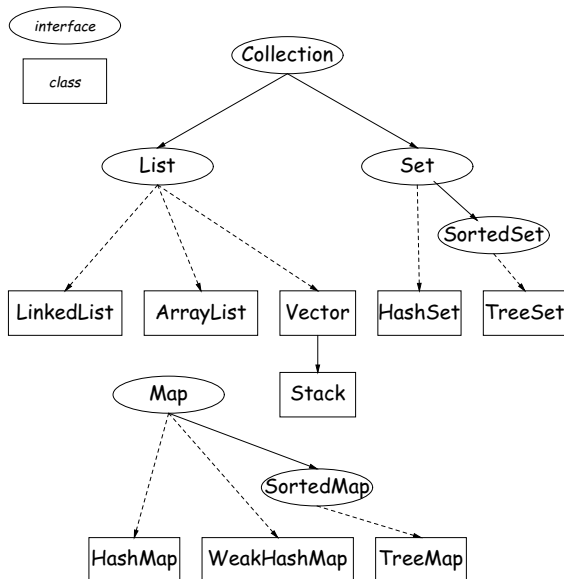
- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size N .
- Entries show the *size of problem* that can be solved in a second, hour, month (31 days), and century, for various relationships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

Time (μsec)	Max N Possible in			
	1 second	1 hour	1 month	1 century
$\lg N$	10^{300000}	10^{10^9}	$10^{8 \cdot 10^{11}}$	$10^{9 \cdot 10^{14}}$
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: 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`

Why not separate interfaces?

- Answer: avoid proliferation of lower types, interfaces (`Vector`, `readonly Vector`, `add-only remove Vector`).

Problem: How to Retrieve?

- Collections don't always have an order—no first, no n^{th}
- 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* (like arrays, but more general).
- 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.

Last modified: Fri Oct 5 10:21:28 2001

CS61B: Lecture #15 7

Examples of Use I: Linear Search

Problem: Find an element in list satisfying predicate.

- Assume following definition:

```
/** A one-argument predicate (true/false valued).
 * Calling 'test' applies the predicate. */
interface Predicate {
    /** True iff ARG passes the test. */
    boolean test (Object arg);
}
```

- Solution I:

```
/** The first element of L that satisfies P,
 * or null if none. */
static Object exists (List L, Predicate P) {
    for (int i = 0; i < L.size (); i += 1)
        if (P.test (L.get (i))) return L.get (i);
    return null;
}
```

- **Disadvantage:** On some Lists, `get(k)` can be a $\Theta(k)$ operation, leading to $\Theta(N^2)$ algorithm, for lists of size N .

Last modified: Fri Oct 5 10:21:28 2001

CS61B: Lecture #15 8

Faster Linear Search

- 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.

```
// [NOTE: Changed from the paper version
// handed out in lecture.]
/** The first element of L that satisfies P,
 * or null if none. */
static Object exists (List L, Predicate P) {
    for (Iterator i = L.iterator ();
         i.hasNext (); ) {
        Object obj = i.next ();
        if (P.test (obj))
            return obj;
    }
    return null;
}
```

Last modified: Fri Oct 5 10:21:28 2001

CS61B: Lecture #15 9

Example of Use II: Inserting New Elements

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

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

Last modified: Fri Oct 5 10:21:28 2001

CS61B: Lecture #15 10

Sublists and Views

Problem: Delete items $K, K+1, \dots, M$ from List L .

- The `sublist` operation is supposed to yield a "view of" part of an existing list.
- A *view* is an alternative presentation of (interface to) an existing object.
- Changes in original list reflected in sublist view, and vice-versa.
- For example, could solve problem like this:

```
L.sublist (K, M+1).clear ();
```

which modifies L by removing the elements in the sublist consisting of items K through $M+1$.

- Another example of a view: a `Map` is a kind of modifiable function:

```
Object get (Object key); // Value at KEY.
Object put (Object key, Object value);
// Set get(KEY) -> VALUE
```

- Provides three views, which change as the `Map` changes:

```
Set keySet (); // The set of all keys
Collection values (); // The multiset of mapped-to values
Set entrySet (); // The set of all (key, value) pairs
```

Last modified: Fri Oct 5 10:21:28 2001

CS61B: Lecture #15 11