What we have been doing:

Suppose we want to have a list:

```java
public interface List<E> {
    void add (E e, int rank);
    E get (int rank);
    E remove (int rank);
    int size ()
}
```

Initially:

- Adding 4 at rank 0
- Adding 5 at rank 1
- Adding 6 at rank 1

Get at rank 2: yields 6
Remove at rank 0, returns 4

6 4
Implementation Ideas

Use an array

\[ a: \begin{array}{ccccc}
  6 & 2 & | & 4 & 3 & 1 \\
\end{array} \]

First \[ a.length - 1 \]

Count = 5

Use a linked list

\[ \text{needs} \]
In a L.T.

\[
\text{get(int rank)}
\]

\[
\text{if (rank < 0 || rank >= count)}
\]

\[
\text{throw ...}
\]

\[
\text{if (rank < count/2)}
\]

\[
\text{Cell p = header.next;}
\]

\[
\text{for (int i = 0; i < rank; ++i)}
\]

\[
\text{p = p.next;}
\]

\[
\text{return p.value;}
\]

\[
\text{else}
\]

\[
\text{Cell p = header.prev;}
\]

\[
\text{for (int i = count-1; i >= rank; --i)}
\]

\[
\text{p = p.prev;}
\]

\[
\text{return p.value;}
\]

What's the running time in the worst case?

\[
C_1 \frac{n^2}{2} + C_2
\]
Algorithm analysis

Suppose you have an algorithm, such as `get()` on an array or list. You'd like to evaluate its efficiency. That might depend on lots of things:

- speed of computer
- quality of compiler
- choice of language
- amount of data
- luck?

What can we do?

```java
In an array implementation
E = get (int rank) 
if (rank < 0 || rank >= count)
    throw ... 
int pos = (First + rank) % length; 
return a[pos]; 
```

Each operation here runs in some fixed amount of time

Running time is constant

More precisely \( c_1 t \leq T \leq c_2 \)
Other algs have running times like $O(n^2) + O(n^2)$ or ones that are more complicated.

$$O(n^2), O(n), O(1)$$

These are big-$O$ classes.

We would like to consider how algs compare when $n$ gets large.

$O(1)$, time at most proportional to $n$, is better than $O(n)$, etc.

Some problems can be solved by a variety of algs.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$0.58n^3$ msecs.</th>
<th>19.5 msecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>.6 msecs</td>
<td>200 msecs</td>
</tr>
<tr>
<td>100</td>
<td>.6 msecs</td>
<td>2 secs</td>
</tr>
<tr>
<td>1000</td>
<td>.6 sec</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>10 min</td>
<td>3.2 min</td>
</tr>
<tr>
<td>100,000</td>
<td>7 days</td>
<td>32 min</td>
</tr>
<tr>
<td>10^4</td>
<td>19 years</td>
<td>5.4 hrs</td>
</tr>
</tbody>
</table>

Big-$O$ only matters as $n$ gets large.

Two algs in same big-$O$ class are not necessarily equally good.
Informally

\[ f(n) \in O(g(n)) \]

if \( f(n) \) is eventually upper-bounded by some constant times \( g(n) \)

\[ f_1(n) = 3n^2 + 100n \in O(n^2) \]
\[ f_2(n) = 2n + 10000 \in O(n^2) \]

It's also true that both of these are \( O(n^3) \)

\( f_1(n) \) is also lower-bounded \( \Omega(n^2) \)
\( f_2 \in \Omega(n) \)

Fortunately we can often handle things simply

\[ f(n) \in \Theta(g(n)) \text{ if } \lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0 \]

So let's review the list ops
Examples:

\begin{align*}
\text{sum} &= 0 \\
&\text{for } (\text{int } i = 0; i < n; ++i) \\
&\quad \text{for } (\text{int } j = 0; j < n; ++j) \\
&\qquad \text{sum} += i \times j \\
&\text{for } (\text{int } i = 0; i < n; ++i) \\
&\quad \text{for } (\text{int } j = i + 1; j < n; ++j) \\
&\quad \quad \ldots
\end{align*}