<table>
<thead>
<tr>
<th>Operation</th>
<th>Rep A: List</th>
<th>Rep B: Ordered List</th>
<th>Rep C: Ordered Array</th>
<th>Rep D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{add}(x) \times \in S )</td>
<td>( \sim n )</td>
<td>( \sim n/2 )</td>
<td>( \sim n/2 + \log_2 n )</td>
<td></td>
</tr>
<tr>
<td>( \text{contains}(x) \times \notin S )</td>
<td>( \sim n )</td>
<td>( \sim n/2 )</td>
<td>( \sim \log_2 n )</td>
<td></td>
</tr>
<tr>
<td>( \text{remove}(x) \times \in S )</td>
<td>( \sim n/2 )</td>
<td>( \sim n/2 )</td>
<td>( \sim n/2 + \log_2 n )</td>
<td></td>
</tr>
<tr>
<td>( \text{intersect}(S) \times \emptyset \cap T = \emptyset )</td>
<td>( \sim n \times m )</td>
<td>( \sim n + m )</td>
<td>( \sim n + m )</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>T</td>
<td>= m )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rep A: Unordered List

add(x) traverse the entire list, stopping if x is found
if x was not found, add it at the end

contains(x) check all items for equality to x

remove(x) find list item for x
if found it, fix up pointers

intersect(T) take the smaller list
- for each element e in the larger list 2
  - if e is larger list 2
    - put e in the result

|S| = n

Time \( \approx n \)

Time \( \approx n \)

Time \( \approx n/2 \)

5 2 1 3 6
2 4 3 7 8 5
\( n \times m \)
for each element e in S

\[\text{if } T \text{ contains } e \leftarrow \text{time} = m\]

\[\text{put } e \text{ into result set}\]

\[n \times m\]
Sorted List: no duplicates, ordered increasing

add(x)
- Traverse the list until either finding x, a larger item, or the end
  - if x was not found, insert x before final position

contains(x)
- Look through the list until either finding x or a larger item or the end

remove(x)
- Look through the list until finding x (take it out) or a larger item (do nothing) or tail (do nothing)
Intersection for sorted lists:

**Approach 1**: Use same alg:

\[
\text{for (E e : S) if T.contains(e) R.add(e)}
\]

\[
\text{Rev: for (E e : S) if T.contains(e) time \( t \leftarrow R.addAtEnd(e) \)}
\]

\[
\frac{n \times m}{2}
\]

\[
\text{Approach 2:}
\]

\[
\text{~ } M + N
\]

\[
R = \{5, 9, 16\}
\]
Two-finger algorithm:

let ptrA = beginning of list A
let ptrB = beginning of list B

while (ptrA + ptrB are both not null)
    if both ptrs refer to the same value
        include it in result
        move both ptrs forward
    else
        move forward the pointer that refers to the smaller number

Comparable interface

compareTo (Object x) \Rightarrow \begin{cases} -1 & \text{if this} < x \\ 0 & \text{if this} \approx x \\ 1 & \text{if this} > x \end{cases}
Phone book:

At each step: eliminate half of the remaining items

Binary Search

\[ f(x) = x \]

\[ \log_2 n \]
Binary Search:

leftEnd = left end of data
rightEnd = right end of data

While (leftEnd is to the left of rightEnd)
look at middle between leftEnd and rightEnd
if it’s equal to the target element
    return true (found it!)
else if it’s less (need to go right)
    leftEnd = to the immediate right of the middle element
else
    rightEnd = to the immediate left of middle element
Rep C: A Set as an Array (ordered)

\textbf{add}(x):
\begin{enumerate}
  \item do binary search to find desired location
  \item if it's already there, return false
  \item \hline\hline\hline
  \item put \textit{x} in the empty slot
\end{enumerate}

\[
\log_2 n + 1 + n/2 + 1 \Rightarrow \log_2 n + \frac{n}{2}
\]

If we run out of space, double the array size.
Adv. of List is that you can splice things into the middle with low cost.

Problem: Finding the right spot is slow

What if — had a way of quickly getting to the middle of the list?
Binary Tree