Plan for today
- Finish discussion of radix sort
- Briefly discuss bucket sort
- Return to ADT Taxonomy
- Introduce Set ADT (if time permits)
protected void radixsortImpl(Digitizer<? super E> digitizer) {
    Object[] from = new Object[getSize()];
    Object[] to = new Object[getSize()];
    int b = digitizer.getBase();
    int count[] = new int[b];
    int numDigits = 0;
    for (int i = 0; i < getSize(); i++) {
        from[i] = a[getPosition(i)];
        numDigits = max(numDigits, digitizer.numDigits((E) from[i]));
    }
    for (int d = 0; d < numDigits; d++) {
        Arrays.fill(count, 0);
        for (Object x : from)
            count[digitizer.getDigit((E) x, d)]++;
        for (int i = 1; i < b; i++)
            count[i] += count[i-1];
        for (int i = getSize()-1; i >= 0; i--)
            to[-count[digitizer.getDigit((E) from[i], d)]] = from[i];
        Object[] temp = from; from = to; to = temp;
    }
    for (int i = 0; i < getSize(); i++)
        put(getPosition(i), from[i]);
    version.increment();
}
Optimizing radix sort

Time complexity: \( C \cdot \frac{b}{r} (n + 2^r) \)

\# bits/digit

\# bits

base for a \( r \)-bit digit

Intuitively want to set \( r \) so \( n = 2^r \) so you reduce \# digits while not making base too high

Solving for \( r \) in \( n = 2^r \) yields \( r = \log_2 n \)
Example: \( c = 5 \), \( n = 1,000,000 \) 32 bit numbers

<table>
<thead>
<tr>
<th># bits/digit</th>
<th># digits</th>
<th>base ( e )</th>
<th>( c \cdot d(n+k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic &lt; radix</td>
<td>1</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>sort</td>
<td>2</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td>counting sort</td>
<td>16</td>
<td>2</td>
<td>65536</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>&gt;4 \times 10^9</td>
</tr>
</tbody>
</table>
What are the limitations of radix sort?

Requires that you can digitize all elements.

Instead of using a comparator to compare entire elements, a digitizer is used to extract each digit.

Think about these costs.
Bucket sort

Define Bucketizer that divides the range of all elements into $n$ buckets

$n = 5$

$O(n)$ Place each element in correct bucket

Insertion sort to finish up
ADT Taxonomy, Part II

collection of elements

manually positioned

algorithmically positioned
Algorithmically Positioned Collection

Untagged Collections:
- Set \( R \) organization depends on element itself
- Date
- Event

Tagged Collections (ungrouped):
- 1976 \( \rightarrow E_1 \)
- 1976 \( \rightarrow E_2 \)
- 1976 \( \rightarrow E_3 \)

Tagged Buckets Collection:
- 1976 \( \rightarrow \{ E_1, E_2, E_3 \} \)
Untagged Algorithmically Positioned Collection

notion of equivalence used

Tagged version Mapping
Tagged Ungrouped Algorithmically Positioned Collection

- Membership only (no duplicates tags) → Mapping
  - Max
    - Uniquely ordered
    - Any
    - Use prefix relations
    - Multiply ordered
      - Compare values
- Tags
- Insert into an untagged collection
  - Set
    - Tagged PriorityQueue
    - Tagged Ordered Collection
    - Tagged Digitized Ordered Collection
    - Tagged Spatial Collection
Tagged Grouped Algorithmically Positioned Collection

tag, bucket
(any data structure)
Set ADT

wrap a tagged element — mapping collection

wrap a tag → bucket of element — Bucket mapping ADT

Fundamental Methods to Support

insert (add, put e tag, e)

find

remove
Set

add(e)
contains (e)
equivalence depends on element

Mapping or Bucket Mapping

put (tag, e)
contains (tag)
equivalence depends on tag