Definitions

- A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
  - Elements of a set are like keys of a map, but without any auxiliary values.

- A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.

- A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.
  - For example, the index of a book maps a given term to one or more locations at which the term occurs.
Sets and Multimaps

Set ADT

- **add(e):** Adds the element $e$ to $S$ (if not already present).
- **remove(e):** Removes the element $e$ from $S$ (if it is present).
- **contains(e):** Returns whether $e$ is an element of $S$.
- **iterator():** Returns an iterator of the elements of $S$.

There is also support for the traditional mathematical set operations of **union, intersection, and subtraction** of two sets $S$ and $T$:

- $S \cup T = \{ e : e \text{ is in } S \text{ or } e \text{ is in } T \}$
- $S \cap T = \{ e : e \text{ is in } S \text{ and } e \text{ is in } T \}$
- $S - T = \{ e : e \text{ is in } S \text{ and } e \text{ is not in } T \}$

- **addAll(T):** Updates $S$ to also include all elements of set $T$, effectively replacing $S$ by $S \cup T$.
- **retainAll(T):** Updates $S$ so that it only keeps those elements that are also elements of set $T$, effectively replacing $S$ by $S \cap T$.
- **removeAll(T):** Updates $S$ by removing any of its elements that also occur in set $T$, effectively replacing $S$ by $S - T$.

Storing a Set in a List

- We can implement a set with a list.
- Elements are stored sorted according to some canonical ordering.
- The space used is $O(n)$.

Nodes storing set elements in order

© 2014 Goodrich, Tamassia, Goldwasser
Generic Merging

- Generalized merge of two sorted lists \( A \) and \( B \)
- Template method \( \text{genericMerge} \)
- Auxiliary methods
  - \( \text{aIsLess} \)
  - \( \text{bIsLess} \)
  - \( \text{bothAreEqual} \)
- Runs in \( O(n_A + n_B) \) time provided the auxiliary methods run in \( O(1) \) time

Algorithm \( \text{genericMerge}(A, B) \)

\[
\begin{align*}
S & \leftarrow \text{empty sequence} \\
\text{while } & \neg A.\text{isEmpty}() \land \neg B.\text{isEmpty}() \text{ do} \\
& a \leftarrow A.\text{first}().\text{element}(); \\
& b \leftarrow B.\text{first}().\text{element}(); \\
& \text{if } a < b & \text{aIsLess}(a, S); A.\text{remove}(A.\text{first}()); \\
& \text{else if } b < a & \text{bIsLess}(b, S); B.\text{remove}(B.\text{first}()); \\
& \text{else } & \{ b = a \\ \\
& \text{bothAreEqual}(a, b, S) & A.\text{remove}(A.\text{first}()); B.\text{remove}(B.\text{first}()); \\
& \text{while } & \neg A.\text{isEmpty}() & \text{aIsLess}(a, S); A.\text{remove}(A.\text{first}()); \\
& \text{while } & \neg B.\text{isEmpty}() & \text{bIsLess}(b, S); B.\text{remove}(B.\text{first}()); \\
& \text{return } & S
\end{align*}
\]

Using Generic Merge for Set Operations

- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time
Multimap

- A multimap is similar to a map, except that it can store multiple entries with the same key.
- We can implement a multimap $M$ by means of a map $M'$.
  - For every key $k$ in $M$, let $E(k)$ be the list of entries of $M$ with key $k$.
  - The entries of $M'$ are the pairs $(k, E(k))$.

<table>
<thead>
<tr>
<th>Multimaps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get($k$)</strong>: Returns a collection of all values associated with key $k$ in the multimap.</td>
</tr>
<tr>
<td><strong>put($k$, $v$)</strong>: Adds a new entry to the multimap associating key $k$ with value $v$, without overwriting any existing mappings for key $k$.</td>
</tr>
<tr>
<td><strong>remove($k$, $v$)</strong>: Removes an entry mapping key $k$ to value $v$ from the multimap (if one exists).</td>
</tr>
<tr>
<td><strong>removeAll($k$)</strong>: Removes all entries having key equal to $k$ from the multimap.</td>
</tr>
<tr>
<td><strong>size()</strong>: Returns the number of entries of the multiset (including multiple associations).</td>
</tr>
<tr>
<td><strong>entries()</strong>: Returns a collection of all entries in the multimap.</td>
</tr>
<tr>
<td><strong>keys()</strong>: Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).</td>
</tr>
<tr>
<td><strong>keySet()</strong>: Returns a nonduplicative collection of keys in the multimap.</td>
</tr>
<tr>
<td><strong>values()</strong>: Returns a collection of values for all entries in the multimap.</td>
</tr>
</tbody>
</table>
Java Implementation

```java
public class HashMultimap<K, V> {
    Map<K, List<V>> map = new HashMap<>(); // the primary map
    int total = 0; // total number of entries in the multimap

    /* Constructs an empty multimap. */
    public HashMultimap() {
    }

    /* Returns the total number of entries in the multimap. */
    public int size() { return total; }

    /* Returns whether the multimap is empty. */
    public boolean isEmpty() { return (total == 0); }

    /* Returns a (possibly empty) iteration of all values associated with the key. */
    Iterable<V> get(K key) {
        List<V> secondary = map.get(key);
        if (secondary != null)
            return secondary;
        return new ArrayList<>(); // return an empty list of values
    }
}
```

Java Implementation, 2

```java
    /* Adds a new entry associating key with value. */
    void put(K key, V value) {
        List<V> secondary = map.get(key);
        if (secondary == null)
            secondary = new ArrayList<>();
        map.put(key, secondary); // begin using new list as secondary structure
        total++;
    }

    /* Removes the (key,value) entry, if it exists. */
    boolean remove(K key, V value) {
        boolean wasRemoved = false;
        List<V> secondary = map.get(key);
        if (secondary != null)
            wasRemoved = secondary.remove(value);
        if (wasRemoved)
            total--; // remove secondary structure from primary map
        return wasRemoved;
    }
```

© 2014 Goodrich, Tamassia, Goldwasser  Sets and Multimaps
Java Implementation, 3

```java
41  /** Removes all entries with the given key. */
42  public static <V> Iterable<V> removeAll(K key) {
43      List<V> secondary = map.get(key);
44      if (secondary != null) {
45          total -= secondary.size();
46          map.remove(key);
47      } else
48          secondary = new ArrayList<>(); // return empty list of removed values
49          return secondary;
50  }
51  /** Returns an iteration of all entries in the multimap. */
52  public static <K,V> Iterable<Map.Entry<K,V>> entries() {
53      List<Map.Entry<K,V>> result = new ArrayList<>();
54      for (Map.Entry<K,List<V>> secondary : map.entrySet()) {
55          K key = secondary.getKey();
56          for (V value : secondary.getValue())
57              result.add(new AbstractMap.SimpleEntry<K,V>(key, value));
58      }
59      return result;
60  }
```

© 2014 Goodrich, Tamassia, Goldwasser   Sets and Multimaps