Sets, Bags, and Tables

Wellesley College CS230
Lecture 13
Thursday, March 15
Handout #23

Exam 1 due Friday, March 16

Overview of Today's Lecture

1. Sets = conceptually unordered collections of elements without duplicates (but may need order on elements to implement efficiently).
2. Bags = conceptually unordered collections of elements with duplicates.
3. Tables = associations of keys and values

Mathematical Sets

In mathematics, a set is an unordered collection of elements w/o duplicates.

A set is written by enumerating its elements between braces. E.g., the empty set (with no elements) is written {}.

Insertion: inserts an element into a set if it's not already there:

insert("a", {}) = {"a"}
insert("b", {"a"}) = {"a", "b"} = {"b", "a"} // order is irrelevant
insert("a", {"a", "b"}) = {"a", "b"} = {"b", "a"}
insert("c", {"a","b"}) = {"a", "b", "c"} = {"a", "c", "b"} = {"b", "a", "c"}
= {"b", "c", "a"} = {"c", "a", "b"} = {"c", "b", "a"}

Deletion: removes an element if it's in the set:

delete("a", {}) = {}
delete("a", {"a", "b"}) = {"b"}
delete("b", {"a", "b"}) = {"a"}
delete("c", {"a", "b"}) = {"a", "b"}

Size (Cardinality): |S| is the number of elements in S.

|{}| = 0; |{"a"}| = 1; |{"a", "b"}| = 2; |{"a", "b", "c"}| = 3;

13-3

More Sets

Membership: $x \in S$ is true if x is in the set S and false otherwise.

"a" ∈ {} = false; "a" ∈ {"a", "b"} = true; "c" ∈ {"a", "b"} = false;

Subset: $S1 \subseteq S2$ is true if all elements in S1 are also in S2.

{} ⊆ {"a", "b"} is true; {"a"} ⊆ {"a", "b"} is true; {"b"} ⊆ {"a", "b"} is true;
{"a", "b"} ⊆ {"a", "b"} is true; {"a", "b"} ⊆ {"a"} is false; {"a"} ⊆ {} is false;

Union: $S1 \cup S2$ = all elements in at least one of S1 and S2.

{"a", "b", "d", "e"} ∪ {"b", "c", "d", "f"} = {"a", "b", "c", "d", "e", "f"}

Intersection: $S1 \cap S2$ = all elements in at both S1 and S2.

{"a", "b", "d", "e"} ∩ {"b", "c", "d", "f"} = {"b", "d"}

Difference: S1 - S2 or S1/S2 = all elements in S1 that are not in S2.

{"a", "b", "d", "e"} / {"b", "c", "d", "f"} = {"a", "e"}

13-4

A Set Interface in Java, Part 1

```
/** Interface to mutable sets of elements of type T.  A set is an unordered
    collection  that does not contain duplicate elements. WARNING  This
    is different than the standard java.util.Set interface. */

import java.util.*; // import Iterator, Iterable

public interface Set<T> extends Iterable<T> {
  public boolean insert (T x); // Inserts x into this set; returns true if new member
  public boolean delete (T x); // Deletes x from this set; returns true if successful
  public boolean isMember(T x); // Returns true if x is a member of this set, else false.
  public T choose(); // Returns an arbitrary elt from this collection;
                     // throws RuntimeException if empty.
  public T deleteOne(); // Deletes and returns an arbitrary elt from this collection.
                        // throws RuntimeException if empty.
  public void union (Set<T> s); // Modify this set to contain the union of its
                                // elements with those of s.
  public void intersection (Set<T> s); // Modify this set to contain the intersection of its
                                       // elements with those of s.
  public void difference (Set<T> s); // Modify this set to contain the difference of its
                                     // elements with those of s.
  ... more methods on next page ...
```


A Set Interface in Java, Part 2

```
  // The remaining Set methods are similar to Stack, Queue, PQueue methods
  public int size(); // Returns the number of elements in this set.
  public boolean isEmpty(); // Returns true if this set empty, false otherwise
  public void clear(); // Removes all elemets from this set
  public Set<T> copy(); // Returns a "shallow" copy of this set.
  public Iterator<T> iterator(); // Returns an iterator yielding the elements of this set.
  public String toString(); // Returns a string indicating type and contents of this set.
                            // The elements may be listed in any order.
}
```

How to Implement A Set?

As a sorted vector:

How long does isMember() take?
insert()?

As a sorted list:

How long does isMember() take?
insert()?

Mathematical Bags (Multi-Sets)

In mathematics, a bag (multi-set) is an unordered collection of elements **with** duplicates.

A bag is written by enumerating its elements (with duplicates) between braces. E.g., the empty bag (with no elements) is written {}. The bag with one "a" is written {"a", "a"}. The bag with two "a"s is written {"a", "a"}.

Most set notions (insertion, deletion, membership, subset, union, intersection, difference, cardinality) generalize to bags as long as duplicates are accounted for. E.g:

insert("a", {"a", "b"}) = {"a" ,"a", "b"} = {"a", "b", "a"} = {"b", "a", "a"}

delete("a", {"a", "a", "a", "b", "c", "c"}) = {"a" ,"a", "b", "c", "c"}

{"a", "b"} ⊆ {"a", "a", "b"} = true; {"a", "a", "b"} ⊆ {"a", "b"} = false;

{"a", "a", "b", "d"} ∪ {"a", "d", "e"} = {"a", "a", "a", "b", "d", "d", "e"}

{"a", "a", "a", "b", "c"} ∩ {"a", "a", "b", "b"} = {"a", "a", "b"}

{"a", "a", "a", "b", "c"} / {"a", "a", "b", "b"} = {"a", "c"}

|{"a", "a", "a", "b", "c", "c"}| = 6

New Bag Operations

Multiplicity: we shall write #(x,B) (my nonstandard notation) for the number of occurrences of element x in bag B, a.k.a. the multiplicity of x in B.

Suppose B1 = {"a","a", "a", "b", "c", "c"}. Then:
#("a", B1) = 3; #("b", B1) = 1; #("c", B1) = 2; #("d", B1) = 0;

DeleteAll: delete(x,B) deletes one occurrence of x from B. Sometimes we want to delete all occurrences of x from B. For this we use deleteAll(x,B).

delete("a", {"a", "a", "a", "b", "c", "c"}) = {"a" ,"a", "b", "c", "c"}

deleteAll("a", {"a", "a", "a", "b", "c", "c"}) = {"b", "c", "c"}

deleteAll("d", {"a", "a", "a", "b", "c", "c"}) = {"a", "a", "a", "b", "c", "c"}

CountDistinct: |B| counts all element occurrences in B, including duplicates. Sometimes we want just the number of distinct elements. For this we use countDistinct(B).

|{"a", "a", "a", "b", "c", "c"}| = 6

countDistinct({"a", "a", "a", "b", "c", "c"}) = 3

A Bag Interface in Java

```
/** Interface to mutable bags of elements of type T.  A bag is an unordered
    collection  that does not contain duplicate elements. */

import java.util.*; // import Iterator, Iterable

public interface Bag<T> extends Iterable<T> {

  // Methods that are new to bags

  public int multiplicity (T x);  // Return the multiplicity of x in this bag.
  public void deleteAll (T x);  // Delete all occurrences of x in this bag.
  public int countDistinct ();  // Return the number of distinct elements in this bag.
  public Iterator<T> distinctElements  ();  // Return an iterator yielding the
                                            // distinct elements of this bag.

  // All the remaining methods of Bag<T> are those of Set<T>
  … other methods go here …
}
```

Bags are not Sets!

Why not use inheritance to relate Bag<T> and Set<T>?

```
public interface Bag<T> extends Set<T> {
  public int multiplicity (T x);  // Return the multiplicity of x in this bag.
  … other bag-specific methods …
}
```

Because bags generally don't behave like sets! A bag instance cannot always be used where a set is expected. For example:

```
public static boolean isItASet (Set<String> s) {
  s.delete("foo");
  return ! s.isMember("foo");
}
```

isItASet() should return true for *every* set. But it will return false for a bag that contains more than one occurrence of "foo"!

A bag *implementation* might inherit methods from a set *implementation*. But the *type* of a bag is different than the *type* of a set.

13-11

Bag Example: Word Frequencies

```
// Print the frequencies of words in the give file in alphabetical order
public static void frequencies (String filename) {









}
```

13-12

Digression: Ordered Sets and Bags

Sets and bags are unordered collections of elements.

However, we can also have ordered notions of sets and bags, in which the order of elements matters.

The order might be specified by a Comparator or the Comparable compareTo() method, in which case elements would have a well-defined index. For instance, in the ordered bag

{"a", "a", "a", "b", "c", "c"}

"a"s are at indices 0, 1, 2; "b" is at index 3, and "c"s are at indices 4 & 5.

Since elements have a well-defined index, we could request the element at a given index or delete the element at an index from an ordered set/bag.

Inserting elements into or deleting elements from an ordered set/bag can shift the indices of other elements, similar to what happens in a vector.

We will not consider implementations of ordered sets or bags this semester.

Tables Associate Keys with Values

Example: a phone directory

Key (name)	Value (extension)
Ann	3924
Bob	8763
Carol	5816
…	…

Call each row a table entry.

There is at most one entry in a table with a given key.

Conceptually entries are *not* ordered by key
(but they may be ordered in implementation for efficiency.)

Core table operations:
- Get a value from the table for a given key.
- Put a value into the table for a given key
  (overwriting any existing entry with that key)

A Table Interface in Java

```
/** Interface to mutable tables that map keys of type T to values of type V.
    This is somewhat similar to the standard java.util.Map<K,V> interface in Java.
     Unlike Maps, Tables do not allow null values to be associated with a key. */

public interface Table<K,V> {
  public void put(K key, V value); // Modifies table to associate key with non-null value.
                                   // If an entry with key already exists, it is overwritten.
                                   // If value is null, a RuntimeException is thrown.
  public V get(K key); // Return the value associated key in this table, else null.
  public boolean remove (K key); // Remove the entry with the given key from this table.
                                   // The table is unchanged if there is no such key.
                                   // Returns true if there was an entry, else false.
   public Iterator<K> keys(); // Returns an iterator of all keys in the table.
   public int size(); // Returns the number of entries in this table.
   public boolean isEmpty(); // Returns true if this table has no entries, else false.
   public void clear(); // Removes all entries from this table.
   public Table<K,V> copy(); // Returns a "shallow" copy of this table.
   public String toString(); // Returns a string representation of this table.
}
```


Our Phone Directory in Java

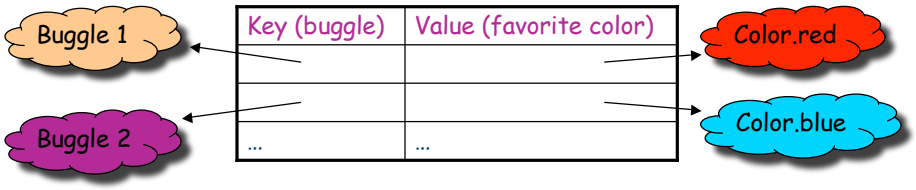
Key (name)	Value (extension)
Ann	3924
Bob	8763
Carol	5816
…	…

```
Table<String,Integer> directory = new TableVectorSorted<String,Integer>();

directory.put("Bob", new Integer(8763));
directory.put("Carol", new Integer(2673));  // Carol's old extension
directory.put("Ann", new Integer(3924));
directory.put("Carol", new Integer(5816));  // Carol's new extension

directory.get("Bob") → 8763 // An Integer, not an int
directory.get("Carol") → 5816 // Most recent extension
directory.get("Diana") → null // No extension in table
```

Table Keys and Values can be of Any Type




```
Table<Buggle,Color> favorites = new TableVectorSorted<Buggle,Color>();

Buggle b1 = new Buggle();
Buggle b2 = new Buggle();

favorites.put(b1, Color.red);
favorites.put(b2, Color.blue);
```


What are Tables Good For?

- Adding virtual instance variables to objects
  (e.g. person's favorite ice cream flavor)
- All sorts of directories (phone directories, Linux file directories)
- Databases provide sophisticated table management
- Indexing files and web pages
- Program analysis and execution (symbol tables, environments)
- Implementing bags