Note: much of the documentation here is described in terms of interfaces, mostly to save space: in this way we don't have to describe every method for every class. Each section lists the classes we have learned about that implement the given interface. You will not need to use all the classes or methods described here. A big part of Problem 2 is to figure out the appropriate Java API tools to use.

`Collection<ElmtType>` Interface

Some classes that implement this interface are: **`ArrayList`, `LinkedList`, `TreeSet`,** and **`HashSet`.**

Selected methods:

`boolean contains(elmt)`
: Returns true iff this element is in this collection

`int size()`
: Returns number of elements in this collection

`boolean add(elmt)`
: Ensures that element is in this collection.
Returns true iff this collection changed as a result of this call

`boolean remove(elmt)`
: Removes an instance of this element from this collection.
Returns true iff this collection changed as a result of this call

`boolean isEmpty()`
: Returns true iff this collection contains no elements.

`Iterator<ElmtType> iterator()`
: Returns an iterator over the elements in this collection.

`Iterator<ElmtType>` Interface

Some classes that implement this interface are: **`Scanner`, `ListIterator`**

Methods:

`boolean hasNext()`
: Returns `true` iff the iteration has more elements.

`ElmtType next()`
: Returns the next element in the iteration. Each successive call returns a different element in the underlying collection. For `Scanner` the `ElmtType` is always `String`.

`void remove()`
: Removes from the underlying collection the last element returned by the iterator. (`Scanner` does not implement this optional method.)

[More other side]

`Map<KeyType, ValueType>` Interface:

Two possible implementations:

`TreeMap<KeyType, ValueType>`

`HashMap<KeyType, ValueType>`

Selected methods:

`ValueType put(key, value)`

Associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value is replaced by the specified value. Returns the previous value associated with specified key, or `null` if there was no mapping for key.

`ValueType get(key)`

Returns the value to which this map maps the specified key or `null` if the map contains no mapping for this key.

`ValueType remove(key)`

Removes the mapping for this key from this map if it is present, otherwise returns `null`.

`int size()`	Number of key-value mappings in this map.
`boolean isEmpty()`	Returns true if this map contains no key-value mappings.

Miscellaneous

Some other stuff you may have used before and forgotten.

Suppose you have an initialized `String` variable called `s`:

`new Scanner(s)`

constructs a new `Scanner` that produces values scanned from the specified string.

`s.trim()`

removes leading and trailing whitespace from a string (returns a copy with the changes)

`s.split(regex)`

splits this string around matches of given regular expression and returns the pieces in an array of strings. regex is type String.

`Node` type used in problem 3 (this is the only part of the code handout with C++ code):

```
struct Node {
  int data;
  Node * next;
  Node() { data = 0; next = NULL; }
  Node(int d) { data = d; next = NULL; }
  Node(int d, Node * n) { data = d; next = n; }
};
```

[More other side]