Data Structures in the Java API

Vector

- **From the `java.util` package.**
- **`Vectors` can resize themselves dynamically.**
- **Inserting elements into a `Vector` whose current size is less than its capacity is a relatively fast operation.**
- **Inserting an element into a full `Vector` is a slow operation since the `Vector` must be resized.**
- **When a `Vector` is resized, the default new size is twice the original size.**
- **A `Vector` can be trimmed to size but this leaves no room for inserts without resizing again.**

Vector (cont'd)

- **`Vector`s store references to `Object`s.**
- **To store values of a primitive data type in a `Vector`, use the type-wrapper classes from the `java.lang` package.**
- **The `Vector` methods `contains` and `indexOf` perform linear searches of a `Vector`'s contents.**

Stack

- **The class `Stack` extends the class `Vector`.**
- **Additional methods provided:**

```
public boolean push(Object obj);
public Object pop();
public Object peek();
public boolean empty();
public int search(Object obj);
   *NOTE: Top of stack is position 1!
```

- **The user of `Stack` may perform operations on a stack that are not necessarily typical of a stack since `Stack` inherits methods from `Vector`. This may corrupt the stack if used incorrectly.**

Queues

- **The standard Java library does not have an explicit class for the queue abstraction.**
- **But: The class `LinkedList` does provide the ability to add and remove elements to or from either end of a collection.**
- **The standard Java library does not have an explicit class for the priority queue abstraction.**
- **But: The class `TreeSet` can be used as a priority queue.**

Dictionary

- **A `Dictionary` maps keys to values.**
- **`Dictionary` is an abstract class.**
- **`Dictionary` provides the public interface methods required to maintain a table of key-value pairs, where each key in the table is unique.**
- **`Hashtable` is a subclass of Dictionary.**

Dictionary (cont'd)

- **`Dictionary` abstract methods:**

```
boolean isEmpty();
Object get(Object key);
Object put(Object key, Object value);
Object remove(Object key);
```

Hashtable

- **`Hashtable` is a subclass of `Dictionary`.**
- **`Hashtable` uses chaining to resolve collisions.**
- **`Hashtable`'s default load factor is .75**
- **When the number of entries in the table exceeds the product of the load factor and the current capacity (number of chains), the capacity is increased by calling the `rehash` method.**
- **Additional methods:**

```
public boolean containsKey(Object key);
public boolean contains(Object value);
```

Arrays

- **`Arrays` provides static methods for array manipulation. (extends Object)**

  ```
  public static int binarySearch
      (Object[] a, Object key);
  ```

  The array `a` must be sorted first before use of this method. If the array is not sorted, the results are undefined.

  ```
  public static void sort(Object[] a);
  ```

  The sorting algorithm used is a tuned version of quick sort adapted from Bentley and McIlroy ("Engineering a Sort Function", 1993).

- **The `Arrays` methods are heavily overloaded!**

Collection

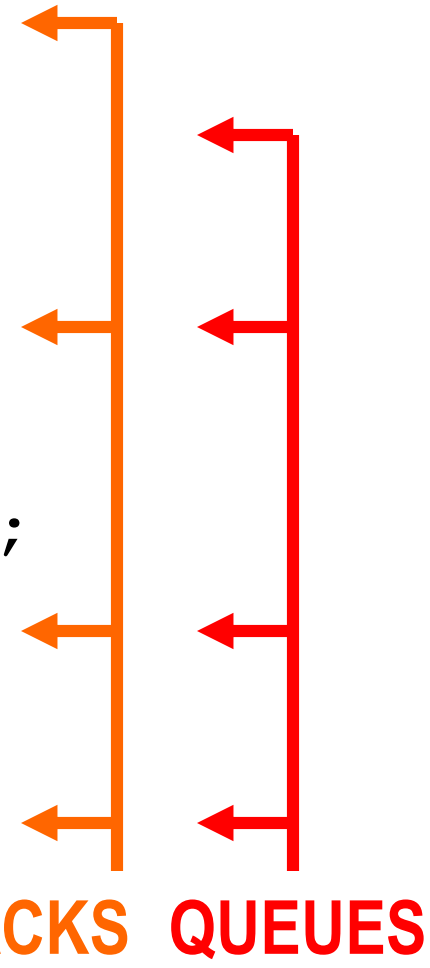
- **`Collection` is the interface from which specific collections are derived.**
- **`Collection` contains bulk operations for adding, clearing, comparing and retaining objects in the collection.**
- **`Collection` provides a method that returns an `Iterator` to examine all elements in the collection. An `Iterator` can remove an element from a collection.**
- **`Iterator` specifies the methods `hasNext`, `next` and `remove`.**

List

- **`List` is the interface that defines an ordered collection that can contain duplicate elements.**
- **`List` is zero-based (the first element is at index 0).**
- **Interface `List` is implemented by the classes:**
  - **`ArrayList` (resizable array implementation)**
  - **`LinkedList` (linked-list implementation)**
  - **`Vector`**
- **`LinkedList` can be used to implement stacks, queues, and double-ended queues (dequeues).**

LinkedList

```
public void add(int index, Object obj);
public void addFirst(Object obj);
public void addLast(Object obj);
public Object get(int index);
public Object getFirst();
public Object getLast();
public boolean remove(Object obj);
public Object removeFirst();
public Object removeLast();
public int size();
```



Set

- **`Set` is the interface that defines an unordered collection that cannot contain duplicate elements.**
- **Interface `SortedSet` extends `Set` and maintains its elements in sorted order.**
- **Interface `Set` is implemented by the classes:**
  - **`HashSet` (hash table implementation)**
  - **`TreeSet` (red-black tree implementation)**
- **Class `TreeSet` implements `SortedSet`.**
- **`TreeSet` can be used to implement priority queues (heaps) with log n time cost for basic operations(add, remove and contains).**

TreeSet

```
public boolean add(Object obj);
public boolean contains(Object obj);
public Object first();
public SortedSet headSet(Object obj);
public boolean isEmpty();
public Iterator iterator();
public Object last();
public SortedSet tailSet(Object obj);
public boolean remove(Object obj);
public SortedSet subSet(Object o1,
  Object o2);
```

PRIORITY QUEUES

Map

- **`Map` is the interface that defines a collection that associates keys to values and cannot contain duplicate keys. (one-to-one mapping)**
- **Interface `SortedMap` extends `Map` and maintains its keys in sorted order.**
- **Interface `Map` is implemented by the classes:**

  - `HashMap` (hash table implementation)

  - `TreeMap` (red-black tree implementation)
- **Class `TreeMap` implements `SortedMap`.**
- **Class `HashMap` is roughly equivalent to `Hashtable` except that it permits `null` as a key or value.**

TreeMap

```
public Object put(Object key,
  Object value);
public boolean containsKey(Object key);
public Object firstKey();
public SortedMap headMap(Object key);
public Object lastKey();
public SortedMap tailMap(Object key);
public boolean remove(Object key);
public SortedMap subMap(Object key1,
  Object key2);
public Collection values();
```

Collections

- **`Collections` provides static methods for manipulation of collections.**

```
public static int binarySearch
    (List list, Object key);
public static void sort(List list);
```

The sort is guaranteed to be stable (equal elements will not be reordered as a result).

The sort is a modified version of merge sort.

```
public static void shuffle(List list);
public static void reverse(List list);
public static Object max(Collection c);
public static Object min(Collection c);
```

Should I use the API?

- **In general, use of the API will result in code that runs efficiently and uses memory wisely and make coding much faster assuming you know the API well.**
- **Some uses of the API will violate the standard definitions of data structures and may result in corrupt data if used incorrectly.**
- **For time-critical or data-critical applications, user-defined data structures are preferable.**