Documentation for COMP 103 Tests

Brief, simplified specifications of some relevant Java collection types and classes, including big-O costs for standard methods.

Note: *E* stands for the type of the item in the collection.

```
interface Collection<E>
    public boolean isEmpty()                  // cost: O(1) for standard collection classes
    public int size()                         // cost: O(1) for standard collection classes
    public void clear()                       // cost: O(1) for standard collection classes
    public boolean add(E item)                // cost: depends on class
    public boolean contains(Object item)      // cost: depends on class
    public boolean remove(Object element)     // cost: depends on class
    public void addAll(Collection<E> c)       // cost: depends on class   Adds all items in c to collection.
```

```
interface List<E> extends Collection<E>
    // Implementations: ArrayList
    isEmpty(), size(), clear()                // As for Collection
    public E get(int index)                   // cost: O(1)
    public E set(int index, E element)        // cost: O(1)
    public boolean contains(Object item)      // cost: O(n)
    public void add(int index, E element)     // cost: O(n) (unless index close to end.)
    public E remove(int index)                // cost: O(n) (unless index close to end.)
    public boolean remove(Object element)     // cost: O(n)
    public void sort((E e1, E e2)->{..});     // cost: O(n log(n)) in general
                                              //       O(n)        almost sorted
```

```
interface Set extends Collection<E>
    // Implementations: HashSet, TreeSet
    isEmpty(), size(), clear()                // As for Collection
    public boolean add(E item)                // cost: HashSet: O(1)  TreeSet: O(log(n))
    public boolean contains(Object item)      // cost: HashSet: O(1)  TreeSet: O(log(n))
    public boolean remove(Object element)     // cost: HashSet: O(1)  TreeSet: O(log(n))
```

```
class Stack<E> implements Collection<E>
    isEmpty(), size(), clear()                // As for Collection
    public E peek()                           // cost: O(1)
    public E pop()                            // cost: O(1)
    public E push(E element)                  // cost: O(1)
    // (peek and pop throw exception if the queue is empty)
```

```
interface Queue<E> extends Collection<E>
    // Implementations: ArrayDeque, LinkedList, PriorityQueue
    isEmpty(), size(), clear()        // As for Collection
    public E peek()                   // cost: O(1)
    public E poll()                   // cost: ArrayDeque, LinkedList: O(1)  PriorityQueue: O(log(n))
    public boolean offer(E element)   // cost: ArrayDeque, LinkedList: O(1)  PriorityQueue: O(log(n))
    // (peek and poll return null if the queue is empty)
```

```
interface Deque<E> extends Collection<E>
   // Implementations: ArrayDeque, LinkedList
   isEmpty(), size(), clear()                 // As for Collection
   public E peekFirst()                       // cost: O(1)     same as peek()
   public E peekLast()                        // cost: O(1)
   public E pollFirst()                       // cost: O(1)     same as poll()
   public E pollLast()                        // cost: O(1)
   public boolean offerFirst(E element)       // cost: O(1)
   public boolean offerLast(E element)        // cost: O(1)     same as offer(E element)
   public E pop()                             // cost: O(1)     throws exception if deque is empty
   public E push(E element)                   // cost: O(1)
   // (peek.. and poll.. methods return null if the deque is empty)
```

```
interface Map<K, V>
   // Implementations: HashMap, TreeMap
   isEmpty(), size(), clear()                 // As for Collection
   public V get(K key)                        // cost: HashMap: O(1) TreeMap: O(log(n))
   public V put(K key, V value)               // cost: HashMap: O(1) TreeMap: O(log(n))
   public V remove(K key)                     // cost: HashMap: O(1) TreeMap: O(log(n))
   public boolean containsKey(K key)          // cost: HashMap: O(1) TreeMap: O(log(n))
   public Set<K> keySet()                     // cost: O(1)
   public Collection<V> values()              // cost: O(1)
   public Set<Map.Entry<K,V>> entrySet() // cost: O(1)
   // get(..) returns null if key not present; put(..) & remove(..) return the old value, (if any)
```

```
class Collections:  // (static methods)
   public void sort(List<E> list);                          // cost = O(n log(n))  in general
                                                            //        O(n)         almost sorted
   public void sort(List<E> list, (E e1, E e2)->{..});// cost = O(n log(n))  in general
                                                            //        O(n)         almost sorted
   public void swap(List<E> list, int i, int j);            // cost = O(1)
   public void reverse(List<E> list);                       // cost = O(n)
   public void shuffle(List<E> list);                       // cost = O(n)
```

```
interface Comparable<E>          // Items can be compared for sorting or a priority queue.
   public int compareTo(E other); // Comparable objects must have a compareTo method:
    // returns     -ve if this comes before other;
    //             +ve if this comes after other,
    //              0   if this and other are the same
// Note: The String class is Comparable, and has a compareTo method
```

Integer and *Double* constants:

Integer.MAX_VALUE; *Integer*.MIN_VALUE;
Double.MAX_VALUE; *Double*.NaN; *Double*.POSITIVE_INFINITY; *Double*.NEGATIVE_INFINITY;