**class java.lang.Object**

* **boolean** equals(Object other)
* String toString()
* **int** hashCode()

**interface java.lang.Comparable\***

* **int** compareTo(Object other) *// return value < 0 if* this *is less than* other *// return value = 0 if* this *is equal to* other *// return value > 0 if* this *is greater than* other

**class java.lang.Integer extends Object** **implements java.lang.Comparable\***

* Integer(**int** value) *// constructor*
* **int** intValue()
* **static int** Integer.MIN\_VALUE
* **static int** Integer.MAX\_VALUE

**class java.lang.Double extends Object** **implements java.lang.Comparable**\*

* Double(**double** value)  *// constructor*
* **double** doubleValue()
* **static double** Double.MIN\_VALUE
* **static double** Double.MAX\_VALUE

**class java.lang.String extends Object implements java.lang.Comparable\***

* **int** length()
* **char** charAt(**int** index)
* **int** indexOf(String str)  *// returns the index of the first occurrence of* str*;*

*// returns* -1 *if not found*

* **boolean** contains(String str)
* String substring(**int** from, **int** to) *// returns the substring beginning at* from *and*

*// ending at* (to -1)

* String substring(**int** from) *// returns* substring(from, length())
* String[] split(String regex) *// splits this string around matches of the given   
   // regular expression*
* char[] toCharArray() *// converts this string to a new character array*

**class java.lang.Math**

* **static int** abs(**int** x)
* **static double** abs(**double** x)
* **static** **double** pow(**double** base, **double** exponent)
* **static double** random() *//returns a* double *in the range [0.0, 1.0)*
* **static** **double** sqrt(**double** x)

***\*****The AP Java subset uses the "raw"* Comparable *interface, not the generic* Comparable<T> *interface.*

**interface java.util.List<E>**

* **int** size()
* **boolean** add(E obj) *// appends* obj *to end of list; returns* true
* **void** add(**int** index, E obj) *// inserts* obj *at position* index *(0 <= index <= size)*

*// moving elements at position* index *and higher*

*// to the right (adds 1 to their indices); adjusts size*

* E get(**int** index)
* E set(**int** index, E obj) *// replaces the element at* index *with* obj

*// returns the element formerly at* index

* E remove(**int** index) *// removes element from position* index*, moving*

*// elements at position* index + 1 *and higher to the*

*// left (subtracts 1 from their indices) and adjusts*

*// size; returns the element formerly at* index

* Iterator<E> iterator() *//every list can instantiate a private Iterator object*
* ListIterator<E> listIterator()

**class java.util.ArrayList<E> implements java.util.List<E>**

**class java.util.LinkedList<E>** **implements java.util.List<E>, java.util.Queue<E>**

Methods in addition to the List methods:

* **void** addFirst(E obj)
* **void** addLast(E obj)
* E getFirst()
* E getLast()
* E removeFirst()
* E removeLast()

**interface java.util.Queue<E>** *// implemented by* LinkedList<E>

* **boolean** add(E obj) *// enqueues* obj *at the end of the queue; returns* true
* E remove() *// dequeues and returns the element at the front*
* E peek() *// returns the element at the front of the queue;*

*// returns* null *if the queue is empty*

* **boolean** isEmpty()

**class java.util.PriorityQueue<E>** *// E should implement* Comparable*\**

* **boolean** add(E obj)  *// adds* obj *to the priority queue; returns* true
* E remove()  *// removes and returns the minimal element*
* E peek()  *// returns the minimal element*

*// returns* null *if the priority queue is empty*

* **boolean** isEmpty()

**class java.util.Stack<E>**

* E push(E item) *// pushes* item *onto the top; returns* item
* E pop() *// removes and returns the element at the top*
* E peek() *// returns the element at the top of the stack;*

*// throws an exception if the stack is empty*

* **boolean** isEmpty()

**interface java.util.Iterator<E>**

* **boolean** hasNext()
* E next()
* **void** remove() *// removes the last element that was returned* by next

**interface java.util.ListIterator<E>** **extends java.util.Iterator<E>**

Methods in addition to the Iterator methods

* **void** add(E obj) *// adds* obj *before the element that will be returned by* next
* **void** set(E obj) *// replaces the last element returned by* next *with* obj

**interface java.util.Set<E>**

* **int** size()
* **boolean** contains(Object obj)
* **boolean** add(E obj) *// if* obj *is not present in this set, adds* obj *and  
   // returns* true*; otherwise, returns* false
* **boolean** remove(Object obj) *// if* obj *is present in this set, removes* obj *and*

*// returns* true*; otherwise, returns* false

* Iterator<E> iterator()

**class java.util.HashSet<E> implements java.util.Set<E>**

**class java.util.TreeSet<E> implements java.util.Set<E>**

**interface java.util.Map<K, V>**

* **int** size()
* **boolean** containsKey(Object key)
* V put(K key, V value) *// associates* key *with* value

*// returns the value formerly associated with* key

*// or* null *if* key *was not present*

* V get(K key) *// returns the value associated with* key

*// or* null *if there is no associated value*

* V remove(K key) *// removes and returns the value associated with*

*// key; returns* null *if there is no associated value*

* Set<K> keySet()

**class java.util.HashMap<K, V> implements java.util.Map<K, V>**

**class java.util.TreeMap<K, V> implements java.util.Map<K, V>**

**Implementation classes for linked list and tree nodes**

Unless otherwise noted, assume that a linked list implemented from the ListNode class does not have a dummy header node.

**public class** ListNode { **private** Object value; **private** ListNode next;

**public** ListNode(Object initValue, ListNode initNext) { value = initValue; next = initNext; }

**public** Object getValue() { **return** value; } **public** ListNode getNext() { **return** next; }

**public void** setValue(Object theNewValue) { value = theNewValue; }

**public void** setNext(ListNode theNewNext) { next = theNewNext; }

}

Unless otherwise noted, assume that a tree implemented from the TreeNode class does not have a dummy root node.

**public class** TreeNode { **private** Object value;

**private** TreeNode right; **private** TreeNode left;

**public** TreeNode(Object initValue) { value = initValue; left = null; right = null; }

**public** TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)

{ value = initValue; left = initLeft; right = initRight; }

**public** Object getValue() { **return** value; } **public** TreeNode getLeft() { **return** left; } **public** TreeNode getRight() { **return** right; }

**public** **void** setValue(Object theNewValue) { value = theNewValue; }

**public void** setLeft(TreeNode theNewLeft) { left = theNewLeft; }

**public void** setRight(TreeNode theNewRight) { right = theNewRight; }

}