Student ID: .....

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

******
NOTE!!
THIS
CONTAINS
SOLUTIONS
*****

EXAMINATIONS – 2014

TRIMESTER 2

COMP 103
INTRODUCTION TO
DATA STRUCTURES
AND ALGORITHMS

Time Allowed: Two Hours

Instructions: Closed Book.

Attempt ALL Questions.

Answer in the appropriate boxes if possible — if you write your answer elsewhere, make it clear where your answer can be found.

The exam will be marked out of 120 marks.

Documentation on some relevant Java classes, interfaces, and exceptions can be found at the end of the paper. You may tear that page off if it helps.

There are spare pages for your working and your answers in this exam, but you may ask for additional paper if you need it.

Only silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted in this examination.

Non-electronic foreign language to English dictionaries are permitted.

Other materials are not allowed.

Questions	**Marks**
1. General questions	[10]
2. Using collections	[10]
3. Array-based implementation of a collection	[10]
4. Recursion, Sorting	[10]
5. Linked Lists and Trees	[30]
6. Binary Search Trees	[20]
7. Priority Queues and Heaps	[20]
8. Hashing	[10]

SPARE PAGE FOR EXTRA ANSWERS

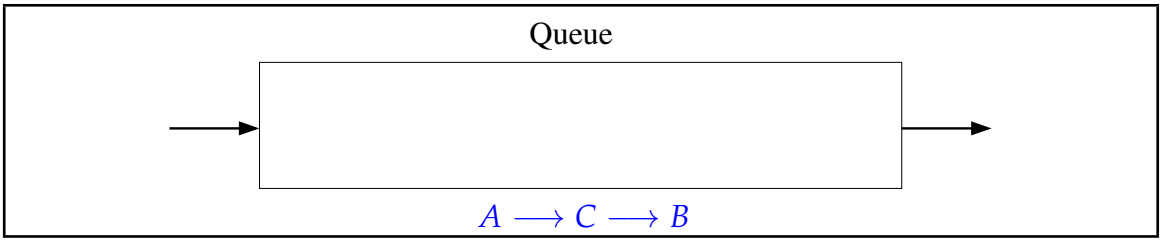
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. General questions [10 marks]

(a) [2 marks] What is the asymptotic ("big-O") cost of searching for an item in a *Bag* that is implemented using an *unsorted* array?

$\mathcal{O}(n)$

(b) [2 marks] Suppose we start with an empty queue and carry out the following operations in order: offer(A), offer(B), poll(), offer(C), offer(A). Draw the queue after these have been carried out.



(c) [2 marks] How many comparisons are required to find a node that is a leaf in a binary search tree (BST) if it contains n items and is perfectly balanced?

$\log_2(n+1)$

(d) [2 marks] Name a fast sorting algorithm which has the same best, average, and worst case "big-$\mathcal{O}$" costs. (You should assume the "basic" algorithm, as was described in lectures).

MergeSort - the other fast sorts all have different worst or best case costs

(e) [2 marks] State the property a binary tree must satisfy in order to be a partially ordered binary tree.

The value in each node must be less than (or equal to) the values in its children.

Question 2. Using collections [10 marks]

Imagine you are writing a program to help university students to figure out what timetable they will have, if they choose to take certain courses. As part of this, you need to write a method checkCourses, which will take two arguments:

- a Set of courses selected by a student, such as `COMP103`, `ENGR112`, `MATH492`.
- a *Map* containing all the courses in the university as keys. For each key, the value is a *List* giving the times of lectures for that course. The keys, and the times, are all just *String* objects. Here are some example elements from this Map, as `key --> value` pairs:

```
JAPA308  --> {TUE4pm, WED10am, FRI2pm}
COMP103  --> {MON4pm, TUE4pm, THU4pm}
RELI489  --> {WED1pm, FRI8am}
ENGR212  --> {MON2pm, THU4pm, FRI9am}
  :
```

You may assume that both the *Set* and the *Map* are correctly constructed elsewhere in the program.

(a) [10 marks] Write the java method checkCourses, which checks the chosen *Set* of courses for clashes. A clash occurs when two courses have a lecture **at the same time**.

Each time your method finds a pair of courses that have a clash, it should print an informative line of text. For example, if passed the *Map* as in the example above, and the *Set* of courses `JAPA308`, `COMP103`, `ENGR212`, it should print:

```
JAPA308 and COMP103 clash at TUE4pm.
COMP103 and ENGR212 clash at THU4pm.
```

checkCourses should return a Set containing all the clash times that were found.

Hint: it might help to write out pseudocode for what you need to do, first.

```
public Set<String> checkCourses(
public Set<String> checkCourses(Map<String,Set<String>> vuwtt, Set<String> courses) {
    // make a new list for the times
    List <String> clashtimes = new ArrayList<String> ();
    for (String course1 : courses)
        for (Str course2 : courses)
            if (course1 != course2)
            {
                // go through the Map's values for each ...
                for (String slot1 : vuwtt.get(course1))
                    for (String slot2 : vuwtt.get(course2))
                        if (slot1 == slot2)
                        {
                            UI. println ('%s and %s clash at %s',course1, course2, slot1);
                            clashtimes.add(slot1);
                        }
            }
    return clashtimes;









}
```

Question 3. Array-based implementation of a Collection [10 marks]

A Set is a type of collection that has no structure, but it is not allowed to contain duplicates. Part of the code for the ArraySet class is shown below.

```
import java.util.*;

public class ArraySet<E> extends AbstractCollection<E> {
    private static int INITIALCAPACITY = 10;
    private int count = 0;
    private E[] data;

    public ArraySet() {
        data = (E[]) new Object[INITIALCAPACITY];
    }

    public boolean add(E item) {
        for (int i=0; i<count; i++){
            if (item.equals(data[i])) return false;
        }
        ensureCapacity();
        data[count]=item;
        count++;
        return true;
    }

    private void ensureCapacity () {
        ...
    }


    public boolean remove(Object item) {
        ...
    }

    public Iterator <E> iterator() {
        return new ArraySetIterator <E> (this);
    }
}
```

(Question 3 continued)

(a) [5 marks] Complete the remove method which will remove an item from the set, if the item is present. remove will return true if and only if it modifies the set. Note that a *Set* is a collection in which order does not matter. Your method should exploit this to be more efficient than it would be if this were an *ArrayList* rather than an *ArraySet*.

```
public boolean remove(Object item) {

    if (item == null) return false;
    for( int i = 0; i < count; i++) {
        if (data[i].equals(item)) {
            count--;
            data[i] = data[count];
            return true;
        }
    }
    return false;



}
```

(b) [5 marks] You have seen the ensureCapacity method in the course, but suppose one also wanted to have a reduceCapacity method, in order to save memory by reducing the capacity of the array if that was possible. Complete the reduceCapacity method below. If the number of items in the Set falls to less than 1/3 (one third) of the current capacity of the array (*i.e.* `data.length`), then the method should reset `data` to refer to a new array of **half** the current capacity.

```
private void reduceCapacity() {


    if (count > data.length/3) return;
    int newLength = data.length/2;
    if (newLength < 2) return;
    E [ ] newArray = (E[ ])(new Object[newLength]);
    for ( int i = 0; i < count; i++)
            newArray[i] = data[i];
    data = newArray;



}
```

Question 4. Recursion, Sorting [10 marks]

Suppose we ran a sorting algorithm on the following array (assume left-to-right, A-Z):

C	E	G	K	F	I	A	D	J	H

(a) [2 marks] Which two elements would Selection Sort swap first?

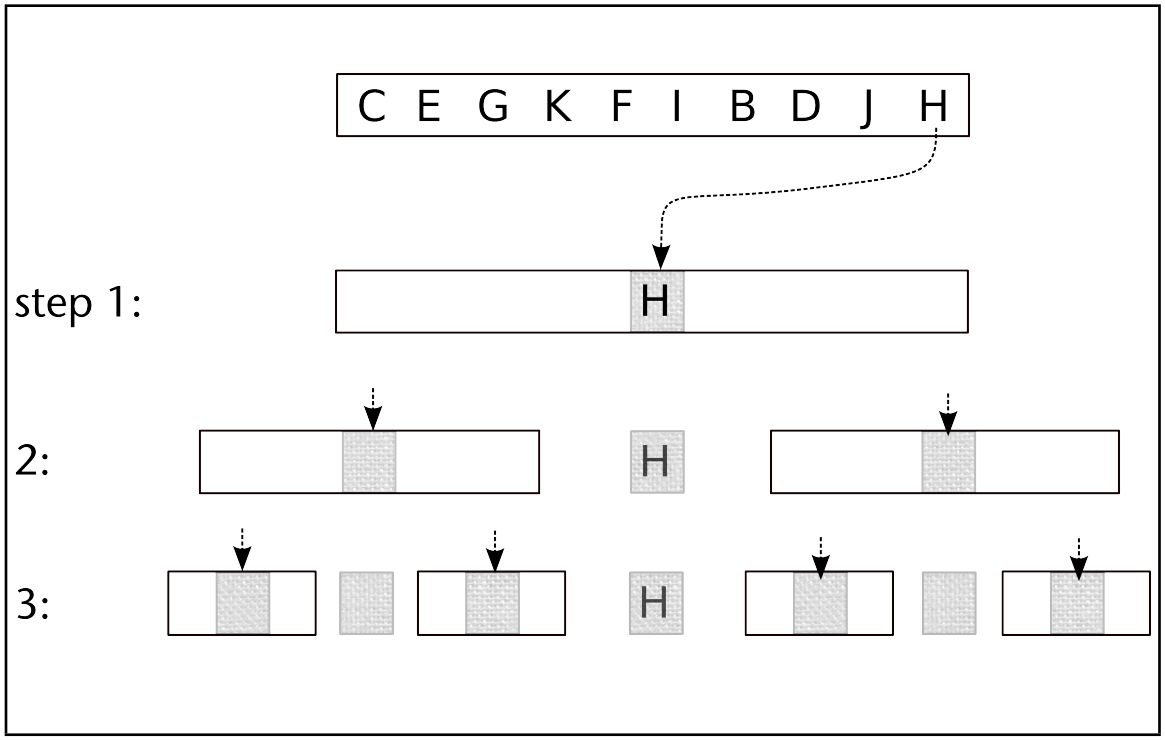
A and C

(b) [2 marks] Which two elements would Insertion Sort swap first?

F and K

(c) [6 marks] The diagram below shows QuickSort sorting an array, with arrows and shading indicating the new pivot points that are being chosen. Suppose we always choose the right-most element of the sub-array as the pivot (the first one is done for you).

Complete the diagram showing how QuickSort progresses at each step.
Note: boxes may contain different numbers of elements.



Question 5. Linked Lists and Trees [30 marks]

Consider the following declaration of LinkedNode, which can be thought of as the first node in a linked list.

```
public class LinkedNode <E> {
    // fields and constructor
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode ){
        value = item;
        next = nextNode;
    }

    // methods ...
    :
}
```

(a) [4 marks] Complete the size() method for the above LinkedNode class, that returns the number of items in the linked list which starts at the current node. You must give a *recursive* version.

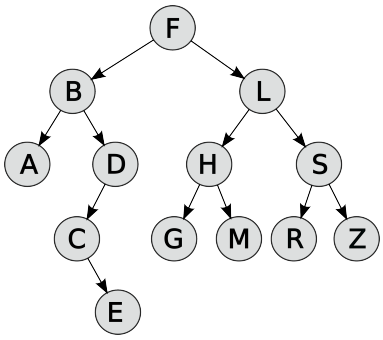
```
public ...
public int size() {

    if (next == null) return 1;

    return next.size() + 1;
}
}
```

(b) [2 marks] Other recursive methods for LinkedNode can be written, such as add, contains, and so on. But there is a difficulty with using LinkedNode as a full implementation of the *List* interface - what is that problem?

A linked list which was empty would presumably have no nodes, in which case no methods could be called on it. A proper linked list should be able to be empty and (eg) return 0 when size() is called on it.



(c) [2 marks] Which of the following *best* describes this tree? (Circle ONE).

Binary tree	Binary search tree	General tree	Partially ordered tree	Complete tree

(d) [4 marks] For the tree shown above, state the order in which the values would be processed in a *post-order traversal*.

A, E, C, D, B, G, M, H, R, Z, S, L, F

(e) [4 marks] The breadth-first traversal algorithm uses a collection to store the nodes that are to be visited. This collection keeps the nodes in the order that they are to be visited. For the tree shown above, draw the state of this collection at the point when node "C" has just been added to the collection.

It's a queue: (back) C - S - H (front)

For questions **(f)** and **(g)** to follow, consider the partial code given below for a class GenTreeNode that will be used to store integer values:

```
class GenTreeNode {
  public List<GenTreeNode> chillun;
  public int value;
  :
  public getValue() { return this.value; }
}
```

(f) [5 marks] Write the contains method for the GenTreeNode class, which returns true if and only if the value passed in as a parameter is stored in the subtree given by the node calling the method.

```
public boolean contains (int n) {


  // something like ...
  if (n == null) return false;
  if (n == value) return true;
  for (GenTreeNode nd : chillun)
      if (nd.contains(n)) return true;
  return false;









 }
```

(g) [5 marks] *(Harder)* Write an iterative contains method for a "wrapper" class GeneralTreeWrapper for a General Tree. Assume that GeneralTreeWrapper has a field named root which is a reference to a GeneralTreeNode object that is the root of the tree of GenTreeNode objects. Thus you may refer to this.root in your code. *Hint: it may help to start with pseudocode.*

```
public boolean contains (int n) {
  // something like ...
  if (n == null) return false;
  // make a stack (or queue) and put the root node on it
  Stack <GenTreeNode> st = new Stack <GenTreeNode> ();
  st.push(this.root);
  // while collection not empty, pop off, process and put children on.
  while (!st.isempty()) {
    GenTreeNode tmp = st.pop();
    if (tmp.getValue() == n) return true;
    for (GeneralTreeNode child : current.getChildren)
        st.push(child);
  }
  return false;









 }
```

(h) [4 marks] On average, how many item-to-item comparisons need to be carried out in order to find a specific item that is stored at depth 2 in a *Binary Tree* that has 15 nodes, assuming the tree is balanced, and a depth-first traversal is used? Give your reasoning.

8. Since the tree is not a BST there are no shortcuts to simply searching the whole tree. There are 15 nodes and it is balanced, the tree has depth 3. There are 4 nodes at depth 2. IF you do a depth-first traversal, you'll need 3,6,10 and 13 comparisons respectively. Sum=32, av=8.

Question 6. Binary search trees [20 marks]

Suppose you are given this ordered sequence of values stored in an array:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

If you inserted these nodes into a Binary Search Tree *in this order*, the resulting tree would be completely unbalanced: each element would be added as a child of the previous one, so you would end up with a linked list instead of a tree!

(a) [5 marks] In the array below, give a re-ordering of these elements that would produce a *balanced* Binary Search Tree. **Hint: Drawing the tree first might help you!**

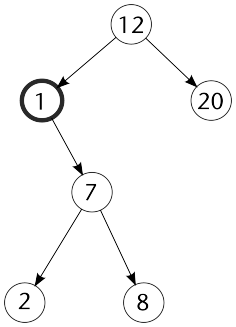
H	D	B	A	C	F	E	G	L	J	I	K	N	M	O

there are many arrays that will produce this - will be some sort of preorder though

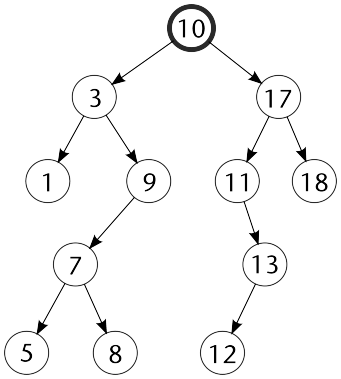
(b) [2 marks] Which kind of traversal would process the nodes in the order ABCDEFGHIJKLMNO when applied to *any* BST containing the above items?

An in-order depth-first traversal.

(c) [3 marks] Draw the tree that will result from removing the value "1" from the binary search tree (BST) shown below.



(d) [3 marks] Draw the tree that will result from removing "10" from the BST below.



(e) [7 marks] A BSTSet is a *Set* that uses a tree of BSTNode objects as the data structure. In the box below, complete the **pseudocode** for the add method in BSTNode. Since it is in BSTNode (not BSTSet) it will need to be a recursive method. The first couple of lines are done for you. Assume "item" is a reference to the value that is stored in the BSTNode.

```
public boolean add (E value) {
    if value equals item
    return false          // item already present


    if value < item      // belongs on left
        if there is no left child
            insert as left child, and return true
        else add value to left child
    else                 // belongs on right
        if there is no right child
            insert as right child, and return true
        else add value to right child



}
```

Student ID: .....

SPARE PAGE FOR EXTRA ANSWERS

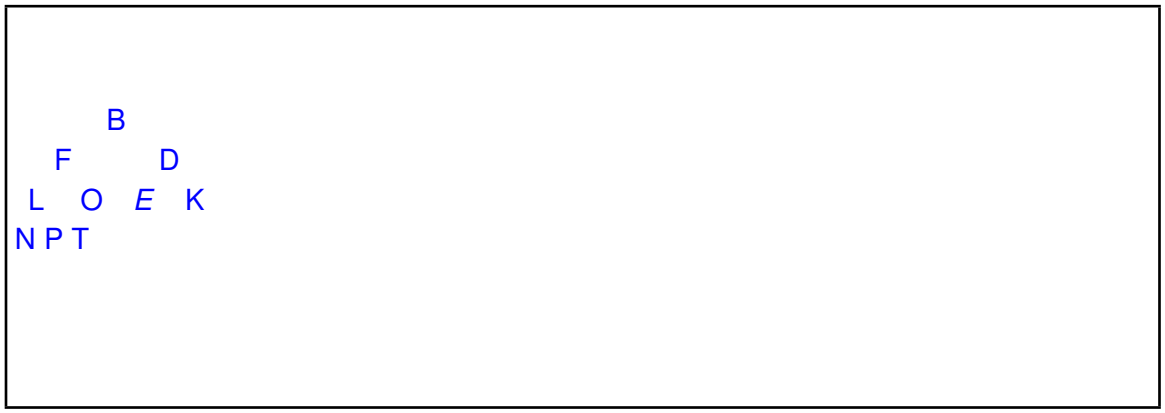
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 7. Priority Queues and Heaps [20 marks]

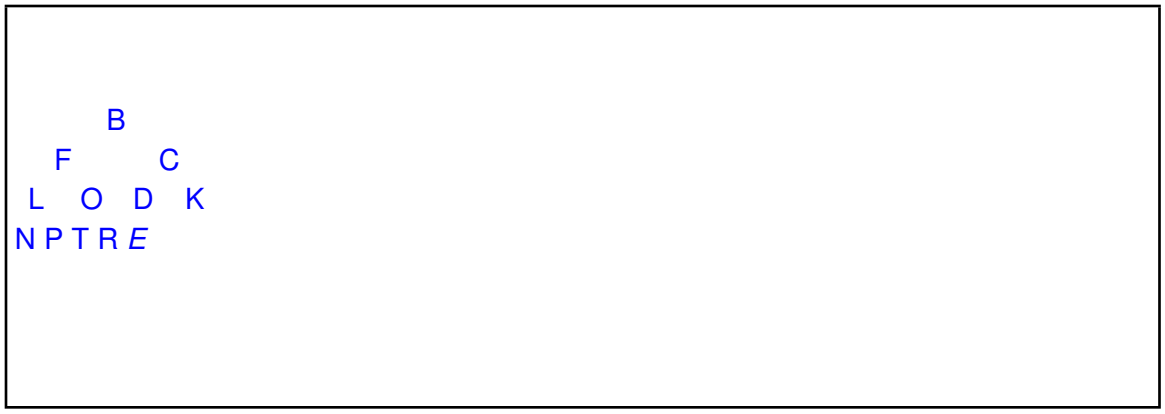
A **heap** is a complete, partially ordered binary tree in an array. The following array represents a heap.

B	F	D	L	O	E	K	N	P	T

(a) [4 marks] Draw the heap as a tree.



(b) [4 marks] Redraw the POT (partially ordered tree) after the following operations have been carried out: `Add 'R', Add 'C'`.



(c) [4 marks] Redraw the tree from your answer to **(b)**, after removing an element (poll) from the heap.

```
      C
   F      D
 L   O  E   K
N P T R
```

(d) [2 marks] If a complete partially ordered binary tree is stored in an array, and the index of the root is 0, what are the indexes of the children of the node at index *i*?

(2 * i + 1) and (2 * i + 2)

(e) [6 marks] The HeapQueue class implements a *Priority Queue*, by using a heap.

Assume that the HeapQueue contains the following fields:

```
private E[] data;
private int count;
private Comparator <E> comp;
```

where comp is a comparator that considers values with higher priority to be larger than values with lower priority. When new items are added to the heap, they are "bubbled up" to their correct position to ensure the array remains a valid heap.

Complete the following bubbleUp(*int* i) method that moves the value at index i up the partially ordered tree into its correct position.
(*Hint: use the comparator, and use recursion on* bubbleUp).

```
private void bubbleUp(int i) {
    if (i==0) return;

    int parent = (i-1)/2;
    if (comp.compare(data[parent], data[i]) < 0){
        E temp = data[i];
        data[i]=data[parent];
        data[parent] = temp;
        bubbleUp(parent);
    }









}
```

Question 8. Hashing [10 marks]

(a) [4 marks] Explain why iterating over the items stored in a HashSet can be a slow process.

The only way to iterate is to step through the array, passing over the nulls in many of the positions. If the array is large (and the set is small) this will be slow, as it scales with the size of the array, not the size of the set.

(b) [3 marks] Once a Hash Table gets too full, it needs to move the items to an larger array. Why is doubling and copying the array elements (as we did for ArrayList for example) *not* the correct way to do this, and what is the correct way?

Double and copy would put elements into the same positions in a larger array, but the hashed code depends on the size of the array, so a new hash of an existing elt won't lead to it in the array. The right thing to do is to rehash every element to its new position in the new array.

(c) [3 marks] When implementing a HashSet and using open addressing (probing), the operation add is easy to achieve but remove requires more thought. Why is remove more challenging?

The element you want to remove may be part of a chain, and setting its position to null would prevent probing from reaching the elements further down the chain. Replacing null by a placeholder (tombstone) fixes this.

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Appendix (may be removed)

Brief (and simplified) specifications of some relevant interfaces and classes.

```
interface Collection<E>
    public boolean isEmpty()
    public int size()
    public boolean add(E item)
    public boolean contains(Object item)
    public boolean remove(Object element)
    public Iterator<E> iterator()

interface List<E> extends Collection<E>
    // Implementations: ArrayList, LinkedList
    public E get(int index)
    public E set(int index, E element)
    public void add(int index, E element)
    public E remove(int index)
    // plus methods inherited from Collection

interface Set extends Collection<E>
    // Implementations: ArraySet, HashSet, TreeSet
    // methods inherited from Collection

interface Queue<E> extends Collection<E>
    // Implementations: ArrayQueue, LinkedList, PriorityQueue
    public E peek () // returns null if queue is empty
    public E poll () // returns null if queue is empty
    public boolean offer (E element) // returns false if fails to add
    // plus methods inherited from Collection

class Stack<E> implements Collection<E>
    public E peek () // returns null if stack is empty
    public E pop () // returns null if stack is empty
    public E push (E element) // returns element being pushed
    // plus methods inherited from Collection

interface Map<K, V>
    // Implementations: HashMap, TreeMap, ArrayMap
    public V get(K key) // returns null if no such key
    public V put(K key, V value) // returns old value, or null
    public V remove(K key) // returns old value, or null
    public boolean containsKey(K key)
    public Set<K> keySet() // returns a Set of all the keys
```

```
interface Iterator<E>
    public boolean hasNext();
    public E next();
    public void remove();

interface Iterable<E>                  // Can use in the "for each" loop
    public Iterator<E> iterator();

interface Comparable<E>                // Can compare this to another E
    public int compareTo(E o);         // -ve if this less than o; +ve if greater than o;

interface Comparator<E>                // Can use this to compare two E's
    public int compare(E o1, E o2);    // -ve if o1 less than o2; +ve if greater than o2
```

```
class Collections
    public static void sort( List<E>)
    public static void sort( List<E>, Comparator<E>)
    public static void shuffle( List<E>, Comparator<E>)

class Arrays
     public static <E> void sort(E[ ] ar, Comparator<E> comp);

class Random
    public int nextInt( int n);    // return a random integer between 0 and n-1
    public double nextDouble(); // return a random double between 0.0 and 1.0

class String
    public int length()
    public String substring( int beginIndex, int endIndex)
```
