



EXAMINATIONS — 2009

MID YEAR

COMP103
Introduction to
Data Structures and Algorithms
SOLUTIONS

Time Allowed: 3 Hours

- Instructions:**
1. Attempt **all** of the questions.
 2. *Read each question carefully before attempting it.* (Suggestion: You do not have to answer the questions in the order shown. Do the questions you find easiest first.)
 3. This examination will be marked out of **180** marks, so allocate approximately 1 minute per mark.
 4. Write your answers in the boxes in this test paper and hand in all sheets.
 5. Non-electronic translation dictionaries are permitted.
 6. Calculators are allowed.
 7. Documentation on relevant Java classes and interfaces is at the end of the paper.

Questions	Marks
1. Basic Questions	[20]
2. Using Collections	[24]
3. Implementing Array-based Collections	[16]
4. Linked Collections	[15]
5. Sorting	[20]
6. Trees	[30]
7. Partially Ordered Trees and Priority Queues	[25]
8. Hashing	[30]

Question 1. Basic Questions

[20 marks]

(a) [2 marks] Suppose we add items A , B and C to an initially empty Set, then $\text{remove}(C)$, $\text{remove}(B)$, and add C again. What is the contents of the Set?

A, C

(b) [2 marks] Suppose we add items A , B and C to an empty stack, in that order, then remove two (using pop), then add C again, then do one more pop . What will the next pop return?

A

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

$O(n)$

(d) [2 marks] Which specific interface do the items in a collection need to implement, in order for a programmer to be able to use the `for each` syntax (such as "`for (String str : wordCollection)`") in Java?

`Iterable`

(e) [2 marks] Name a fast ($O(n \log n)$) sorting algorithm which is not "in-place", that is, it requires a second array.

`MergeSort.`

(f) [2 marks] What is the difference between a binary tree and a binary search tree?

BST is a binary tree in which, for every node, everything in the left subtree is less than it and everything in the right is greater than (or equal to) it.

(g) [2 marks] If a collection contains 2^k elements and they are stored in a perfectly balanced binary search tree, what is the depth of the tree?

k

(h) [2 marks] If you use a stack to store nodes during an iterative tree traversal, will you get a breadth-first or depth-first traversal?

depth-first

(i) [2 marks] What is the main drawback of using linear probing to resolve collisions in hashing?

Collisions build up into long chains, which slows access to them.

(j) [2 marks] What is the defining property of a perfect hash function?

No collisions at all, on the set it is storing.

Question 2. Using Collections

[24 marks]

For this question, assume a `PriorityQueue` implementation in which the head of the queue is the *largest* according to the natural ordering determined by `compareTo`.

(a) [5 marks] Suppose you have a large `List` of `Person` objects, where the `Person` class has a field `ageInYears` specifying the person's age. In words, describe how you could use a `Priority Queue` to efficiently find the 10 oldest people in the list.

Go through the list of `Persons` adding (offering) them to a `Priority Queue`, with priority given by their age. Then pop the first 10 off the queue. Because it's a priority queue these will be the oldest (highest priority).

Given a large text document, it might be useful to list all the unique words in the text together with how often each word occurs. Here is a class for representing words and their occurrences:

```
class WordFrequency implements Comparable <WordFrequency> {
    String word;
    int occurrences=0;
    public WordFrequency(String w, int c) {
        word = w;
        occurrences=c;
    }
    public void incrementCount() {
        occurrences += 1;
    }
    public int compareTo(WordFrequency other) {
        ...
    }
}
```

(b) [3 marks]

Complete the `compareTo` method of the `WordFrequency` class.

```
public int compareTo(WordFrequency other) {
    return (this.occurrences - other.occurrences);

}
}
```

(c) [8 marks] Complete the countWordOccurrences method that takes a scanner to a text file, and returns a map with words as the keys and WordFrequency objects as the values.

```
public Map <String, WordFrequency> countWordOccurrences(Scanner sc) {  
  
    // make a map  
    Map <String, WordFrequency> wordmap = new HashMap <String, WordFrequency> ();  
  
    while (sc.hasNext()) {  
        String str = sc.next();  
        // test whether str is in the key set  
        if (wordmap.keySet().contains(str)  
            wordmap.get(str).incrementCount();  
        else  
            wordmap.put(str, new WordFrequency(str,1) );  
    }  
    return wordmap;  
  
}
```

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.

(d) [8 marks]

Complete the method `printMostCommonWords` that takes the above map as an argument and prints out the 100 most common words in it.

```
public void printCommonWords(Map <String, WordFrequency> wordmap) {  
    // Here's one way, using a priority queue  
    PriorityQueue <WordFrequency> pq = new PriorityQueue <WordFrequency>();  
    for (WordFrequency wf : wordmap.values())  
        pq.offer(wf);  
    for (int i=0; i<100; i++)  
        System.out.println(pq.poll().word);  
  
    OR....  
    read it into a list , sort it , and print out the first 100.
```

```
}
```

Question 3. Implementing Array-based Collections

[16 marks]

The following gives an implementation of `ArrayQueue`, which inserts at the end of an array if there is space, and moves all the elements in the array down when it polls.

```
public class ArrayQueue <E> implements Queue <E> {
    private E [ ] data;
    private int count;
    private static final int INITIALCAPACITY = 16;

    public ArrayQueue() {
        data = (E [ ]) (new Object[INITIALCAPACITY]);
    }
    :
    :
    public boolean offer (E item) {
        if (count == data.length) return false;
        data[count] = item;
        count++;
        return true;
    }

    public E poll () {
        if (count == 0) return null;
        E ans = data[0];
        for (int i=0; i < count-1; i++)
            data[i] = data[i+1];
        count = count-1;
        return ans;
    }
}
```

(a) [2 marks] What is the *average*-case asymptotic (“big-O”) cost of offer?

$O(1)$

(b) [2 marks] What is the *average*-case cost of poll?

$O(n)$

The `ArrayQueue` class shown on the previous page uses an array of fixed size. This means it will not be able to store more elements than the size of the array. Suppose we replace the `offer` method with the following instead, which uses a `doubleAndCopy` method:

```
public boolean offer (E item) {
    if (count == data.length) doubleAndCopy();
    data[count] = item;
    count++;
    return true;
}
```

(c) [5 marks] Complete the `doubleAndCopy()` method, which doubles the size of the array when the queue is full.

```
private void doubleAndCopy() {

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

}
```

This `ArrayQueue` implementation on the previous page is also inefficient, because every time an element is removed by a `poll()` it moves all the remaining elements down. Suppose, instead, we avoid having to move the elements down by storing the location of the first element of the queue in a new variable.

(d) [2 marks] If we do this, what is the *average*-case cost (in “big-O” notation) of `poll`?

$O(1)$

(e) [5 marks] What is the problem with this idea? Describe a strategy for solving this problem which has the same cost, but avoids the problem you identified.

The queue will quickly run out of space and have to double in size, even if this is unnecessary, because it cannot reuse the space at the beginning of the list (2 marks). A poor, but acceptable strategy would be to copy, but not necessarily double when you run out of space (2 marks). (or) The best strategy would be to wrap around to use the front of the array when you reach the end, only double if you really run out of space.

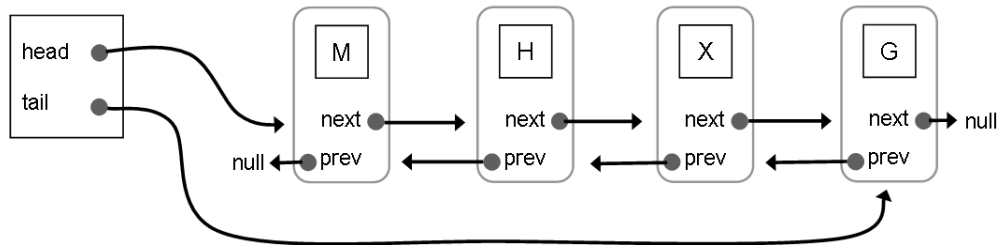
SPARE PAGE FOR EXTRA ANSWERS

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

Question 4. Linked Collections

[15 marks]

In lectures we discussed linked lists. The following diagram shows an example of a “doubly linked” list, in which each node has a link both to the next node in the list as usual, but it also has a link to the previous node. Additionally, the header node has links to both ends of the list in this version:



The code below gives an implementation of a doubly linked list, and lists some of its methods:

```
public class DoublyLinkedList <E> implements AbstractList <E> {  
    private ListNode<E> head = null;  
    private ListNode<E> tail = null;  
    public DoublyLinkedList(){} // constructor  
    public int size () {...  
    public boolean isEmpty(){...  
    public void add(E item){...  
    public boolean remove(E element){...  
    public Iterator <E> iterator () {...
```

DoublyLinkedList uses the ListNode class:

```
public class ListNode <E> {  
    private E value;  
    private ListNode<E> next, prev;  
    public ListNode(E item, ListNode<E> nextNode, ListNode<E> prevNode ){  
        value = item;  
        next = nextNode;  
        prev = prevNode;  
    }  
    public E get() { return value; }  
    public ListNode<E> next() { return next; }  
    public void set(E item) { value = item; }  
    public void setNext(ListNode<E> nextNode) { next = nextNode; }  
    public void setPrev(ListNode<E> prevNode) { prev = prevNode; }  
    :  
}
```

(a) [2 marks] What is the *average*-case asymptotic cost of removing an element from the start of the list?

O(1)

(b) [2 marks] What is the *average*-case asymptotic cost of removing an element from the end of the list?

O(1)

(c) [11 marks] Complete the `remove` method for `DoublyLinkedList`, to remove all occurrences of element from the list. You will need to find the element, re-link those on either side of it, take care to deal with the end cases correctly, and update `head` and `tail` if necessary.

```
private void remove(E element) {
```

```
    find element (3)
```

```
    relink prev/next (3)
```

```
    check for ends (null) (2)
```

```
    set first and last correctly (2)
```

```
}
```

Question 5. Sorting

[20 marks]

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

B	D	F	J	E	H	A	C	I	G
---	---	---	---	---	---	---	---	---	---

(a) [2 marks] Which elements would selection sort swap first?

A and B

(b) [2 marks] Which elements would insertion sort swap first?

E and J

(c) [8 marks] The “fast” sorting algorithms we met in COMP103 were MergeSort, QuickSort, TreeSort and HeapSort. Which two of these have *worst-case* costs that scale as $O(n^2)$?

Give the algorithm name, and the reason (or describe what the worst case is for this algorithm).

Algorithm: QuickSort

Reason: bad choice of pivot, e.g. nearly ordered, and left-hand cell as pivot

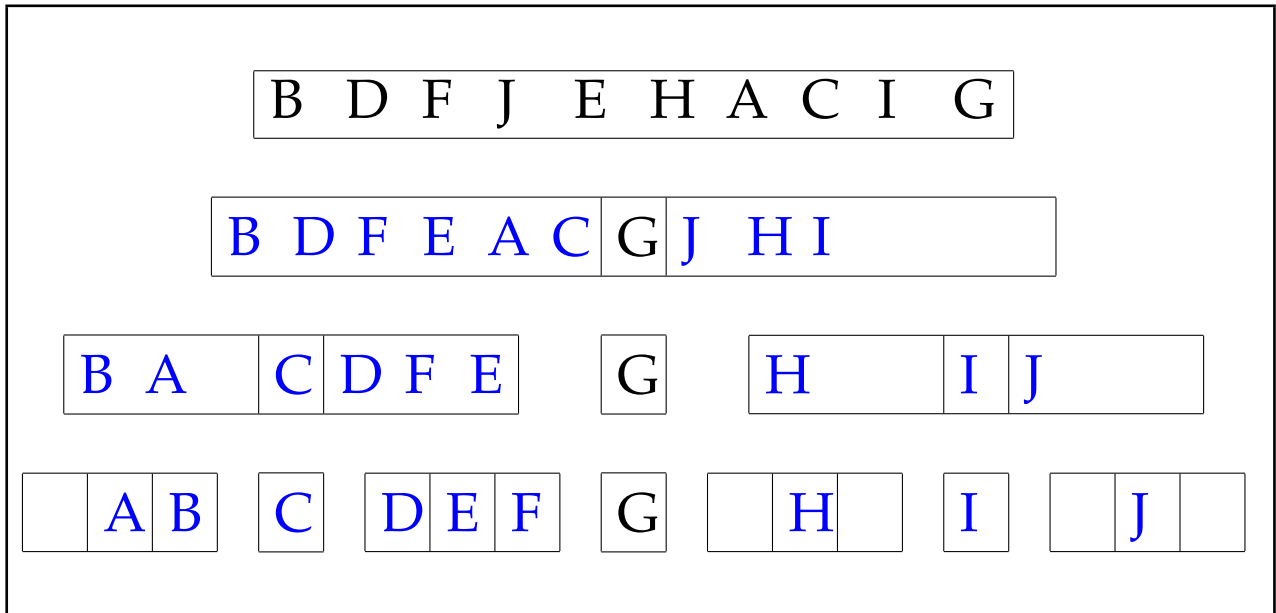
Algorithm: TreeSort

Reason: if nearly sorted already, leads to very unbalanced tree.

The diagram below represents QuickSort sorting an array. Complete the diagram to show the elements in each sub-array as the algorithm progresses. Suppose our pivot algorithm always chooses the right-most element of the sub-array as the pivot (the first one is done for you).

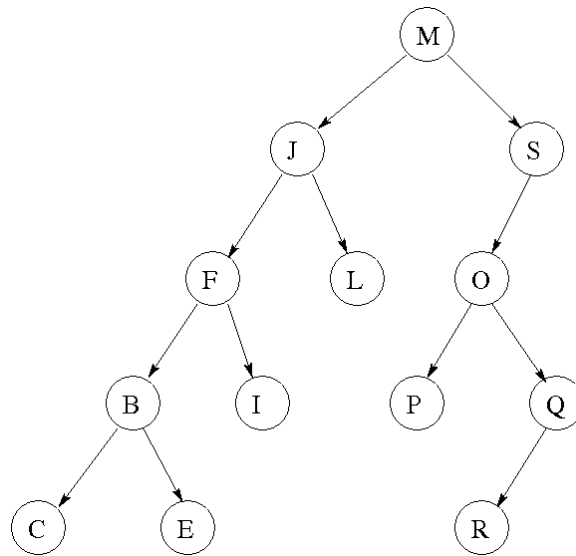
Hint! boxes may contain different numbers of elements.

(d) [8 marks] Complete the diagram showing how QuickSort progresses at each step.



Question 6. Trees

[30 marks]



(a) [2 marks] What is the depth of this tree?

4

(b) [2 marks] What is the minimum depth that a tree with the same elements could have?

4

(c) [4 marks] The tree shown above is almost, but not entirely a Binary Search Tree. Modify the tree to be a correct Binary Search Tree with the same elements.

Swap B (C , E) to be C (B, E), swap O (P, (Q (R ,)) to P(O, Q(, R))

The following algorithm traverses a tree and prints out all of the nodes in the tree using a stack.

```
public static void print (BinaryTreeNode tree) {  
    Stack<BinaryTreeNode> stack = new Stack<BinaryTreeNode>();  
    stack.push(tree);  
    while (!stack.isEmpty()) {  
        BinaryTreeNode node = stack.pop();  
        System.out.println(node.value);  
        if (node.right != null) stack.push(node.right);  
        if (node.left != null) stack.push(node.left);  
    }  
}
```

(d) [6 marks] Show the state of the stack when 'B' is printed if this algorithm is run on the (unmodified) tree at the beginning of this question.

S L I

Question 6(d) traversed a binary tree which has *left* and *right* child nodes. A general tree can have any number of child nodes, so they are often stored in a list. The following code implements a `GeneralTreeNode` class.

```
class GeneralTreeNode<E> {
    List<GeneralTreeNode> children = new ArrayList<GeneralTreeNode>();
    E value;
    public GeneralTreeNode(E value) {
        this.value = value;
    }
}
```

(e) [6 marks] Complete this `printNodesWithTwoChildren()` method which **recursively** prints out the value of all nodes in the tree which have two children.

You may use any recursive traversal algorithm.

```
public static void printNodesWithTwoChildren(GeneralTreeNode<E> node) {
    if (node.children.size() == 2) {
        System.out.println(node.value);
    }

    for (GeneralTreeNode<E> n: node.children) {
        printNodesWithTwoChildren(n);
    }
}
```

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 then 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!

(f) [5 marks] Give (draw) 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

(g) [5 marks] (*Hard*) Given an *arbitrary* (unsorted) ordered sequence of nodes, explain how you could produce a balanced Binary Search Tree.

Hint: do not write Java code

* use a recursive divide-and-conquer algorithm to find (and insert) the middle element, then find the middle of the left sub-sequence, etc. (or) * construct an empty array of the same length, then, treating the array as a heap, perform a pre-order depth-first traversal and at each point insert the next item from the sequence. Generous partial marks for any reasonable attempt. Full marks for any reasonable $n \log n$ /divide and conquer solution.

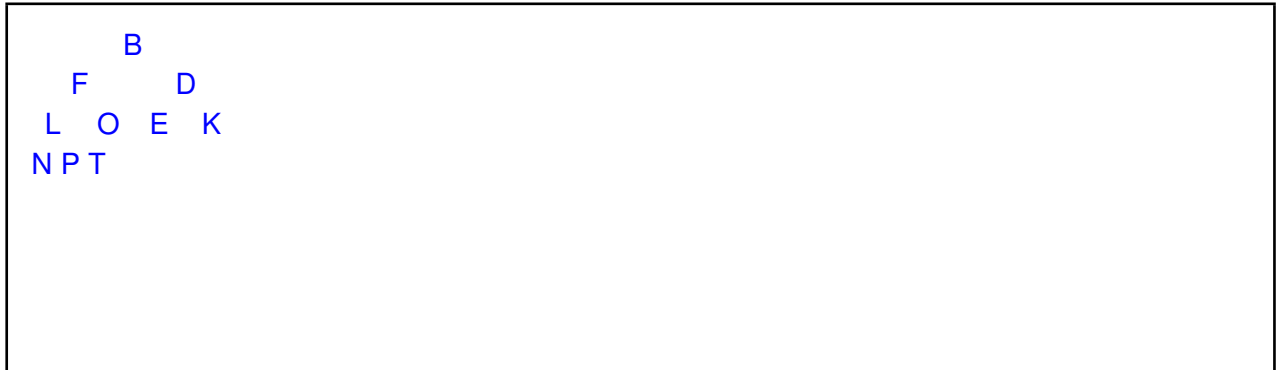
Question 7. Partially Ordered Trees and Priority Queues

[25 marks]

The following array represents a heap (partially ordered tree).

B	F	D	L	O	E	K	N	P	T
---	---	---	---	---	---	---	---	---	---

(a) [3 marks] Draw the heap as a tree.



(b) [3 marks] Redraw the tree after the following operations: Add 'R', Add 'C'.



(c) [3 marks] Redraw the tree from your previous answer after removing an element (poll).



In lectures we discussed implementing a Partially Ordered Tree (POT) as an Array. It is also possible to implement a POT using linked nodes. The following two classes show the fields (but not the methods) of an implementation of a POT using linked nodes:

```

class POT<E extends Comparable> {
    POTNode tree; //top node in the tree
    POTNode last; //last node in the tree
    ...
}

class POTNode<E extends Comparable> {
    E value;
    POTNode parent; //parent node
    POTNode left; //left child
    POTNode right; //right child
    POTNode previous; //previous node in the tree
    ...
}

```

The rest of this question considers different parts of this implementation.

Removing (polling) a value from a partially ordered tree in an array requires removing the element at the front, moving the last element into its place, then calling "pushdown" to find the correct place for the element you moved. It is similar for a POT implemented with linked nodes.

```

// these methods belong to the POT class
public E poll () {
    if (tree == null) return null; //the POT is empty
    E value = tree.value;

    //move the value from the last node to the top of the tree
    tree.value = last.value;

    //remove the last node
    if (last.parent != null) {
        if (last.parent.left == last) last.parent.left = null;
        else last.parent.right = null;
    }
    if (last.previous != null) last = last.previous;
    else { //tree is now empty
        last = null;
        tree = null;
    }

    //push down the value in the top node to its correct place
    if (tree != null) pushdown(tree);
}

```

(d) [6 marks] Implement the pushdown method for the POT class:

```
public void pushdown(POTNode<E> node) {  
  
    if (node.left == null) return;  
  
    POTNode<E> least = left;  
    if (node.right != null) {  
        if (node.right.value.compareTo(node.left.value) < 0)  
            least = node.right;  
    } # least is now the lower of the two children, or the left if one.  
  
    E value = node.value;  
    if (value > least.value) { # need to swap with least and recurse pushdown  
        node.value = least.value;  
        least.value = value;  
        pushdown(least);  
    }  
  
}
```

Adding an element to this POT implementation is slightly more complex. We need to add a new node at the first available place in the POT (at the end), then push the value up to its correct place in the tree.

Here is a *pseudo-code* description of the method for adding an element:

- create a new node, and set its value to the new value
- find the first node in the tree which does not have two children, store as parent of our node
- find the last node in the tree, store as previous of our node
- update the last pointer to point to our node
- set the left (or right if the parent has a child) field in the parent to point to our node
- perform "pushup" on our node

(e) [3 marks] This description does not include any error checks.

Are there any situations where an error could occur, and if so what should the method do to resolve them?

Tree could be null - set last and tree to point to a new node and return (no bubble)

(f) [3 marks] How could we find the first available place in the tree? (Step 2 in the method)

Perform a breadth-first traversal of the tree until we encounter a node which does not have two children.

(g) [4 marks] Why is a node-based implementation a bad choice for implementing a partially ordered tree?

not efficient to find where to put the last element in the array (and previous). heap is $\log(n)$, this is $n \log n$.

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 8. Hashing

[30 marks]

(a) [3 marks] With probing in Hash Tables, why is it a good idea to rehash to a larger table well before the current one is full?

Performance degrades with the number of collisions. A very full table means a very large number of collisions.

(b) [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.

(c) [5 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.

(d) [4 marks] When writing an implementation of 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.

(e) [4 marks] Why is it important that two items that are `“.equal()"` have the same `hashCode`?

If two objects are equal, they should appear only once in a set. But if their hash-codes differ, they will appear twice in the set.

(f) [10 marks]

Suppose you want to write a program to store information about your collection of music CDs. You decide to use a `HashSet`, and you write a class `MusicCD`, with the fields shown below.

A `MusicCD` is identified by its `artist`, `albumTitle`, and `dateBought` fields.

```
public class MusicCD{
    private final String artist ;
    private final String albumTitle;
    private final Date dateBought;

    private String whoBorrowed; // may change
    private String whereStored; // may change
    private int myRating; // a number between 1 and 5, may change
    :
```

For your own purposes you also want to keep track of `whoBorrowed` (which could be null), `whereStored` (e.g. "under my bed") and `myRating` information. These three fields may change over time, but don't affect the identity of the CD itself.

Complete the following `hashCode` and `equals` methods for the `MusicCD` class. You may assume that the `String` and `Date` classes have appropriate `hashCode` functions.

```
public int hashCode(){
    return (( artist .hashCode()*1023+dateBought.hashCode()*1023 +
            albumTitle.hashCode()* 1023;

}
```

```
public boolean equals(Object obj){

    if (! obj instanceof MusicCD)
        return false;
    MusicCD other = (MusicCD) obj;
    return ( artist .equals(other. artist ) &&
            albumTitle.equals(other.albumTitle) &&
            dateBought.equals(other.dateBought));

}
```

Note that it is not safe to compare the hashCodes — — — if two `MusicCD`s are equal, then they must have the same hashCode, but if they have the same hashCode, they are not necessarily equal!

```
}
```

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.

Appendices

Possibly useful formulas:

- $1 + 2 + 3 + 4 + \dots + k = \frac{k(k+1)}{2}$
- $1 + 2 + 4 + 8 + \dots + 2^k = 2^{k+1} - 1$
- $a + (a + b) + (a + 2b) + \dots + (a + kb) = \frac{(2a+kb)(k+1)}{2}$
- $a + as + as^2 + as^3 + \dots + as^k = \frac{as^{k+1} - a}{s-1}$

Table of base 2 logarithms:

n	1	2	4	8	16	32	64	128	256	512	1024	1,048,576
$\log_2(n)$	0	1	2	3	4	5	6	7	8	9	10	20

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

public 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
```

public interface Iterator <E>

```
public boolean hasNext();
public E next();
public void remove();
```

public interface Iterable <E>

```
public Iterator <E> iterator();
```

// Can use in the "for each" loop

public interface Comparable <E>

```
public int compareTo(E o);
```

// Can compare this to another E

public interface Comparator <E>

```
public int compare(E o1, E o2);
```

// Can use this to compare two E's

DrawingCanvas class:

```
public void drawLine(int x, int y, int u, int v) // Draws line from (x, y) to (u, v)
public void drawOval(int x, int y, int wd, int ht) // Draws outline of oval
public void drawString(String str, int x, int y) // Prints str at (x, y)
```

```

public interface Collection<E>
    public boolean isEmpty();
    public int size ();
    public boolean contains(Object item);
    public boolean add(E item);           // returns false if failed to add item
    public Iterator <E> iterator();

```

```

public interface List<E> extends Collection<E>
    // Implementations: ArrayList
    public E get(int index);
    public void set(int index, E element);
    public void add(int index, E element);
    public void remove(int index);
    public void remove(Object element);

```

```

public interface Set extends Collection<E>
    // Implementations: ArraySet, SortedArraySet, HashSet
    public boolean contains(Object element);
    public boolean add(E element);
    public boolean remove(Object element);

```

```

public interface Queue<E> extends Collection<E>
    // Implementations: ArrayQueue, LinkedList
    public E peek ();           // returns null if queue is empty
    public E poll ();          // returns null if queue is empty
    public boolean offer (E element);

```

```

public 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

```

```

public 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 value removed, or null
    public boolean containsKey(K key);
    public Set<K> keySet();        // returns set of all keys in Map
    public Collection<V> values(); // returns collection of all values
    public Set<Map.Entry<K, V>> entrySet(); // returns set of (key–value) pairs

```

Scanner class:

```

public boolean hasNext()           // Returns true if there is more to read
public boolean hasNextInt()        // Returns true if the next token is an integer
public String next()               // Returns the next token (chars up to a space/line)
public String nextLine()           // Returns string of chars up to next newline
public int nextInt ()              // Returns the integer value of the next token

```