UNIVERSITY OF TORONTO

Faculty of Arts and Science

APRIL-MAY 2005 EXAMINATIONS

CSC 148H1 S, 148H5 S and A48H3 S

Duration — 3 hours

No aids allowed

ANSWERS

Answer all questions in the space provided in this paper.

Check that this test paper has 15 pages, including this cover page.

Comments are not required except where indicated, though they may help us to mark your answers.

You need not throw or catch exceptions, unless explicitly asked to.

Efficiency and style are not of the highest importance, but may count for marks in the case of major difficulties.

Helper methods are allowed unless specifically prohibited.

1.	_____	/22
2.	_____	/20
3.	_____	/13
4.	_____	/20
5.	_____	/15
6.	_____	/10
Total	_____	**/100**

continued …

1. [22 marks = 11 × 2]

Each of these short questions is worth 2 marks, and they are separate, independent questions unless otherwise stated.

Part I: TRUE/FALSE: State "True" or "False" and explain briefly. No marks will be given if there is no explanation.

(a) An abstract class can implement an interface.

True. There's nothing about either to forbid it.

(b) Searching for an element in a binary search tree with N nodes takes O(log N).

False. If the tree isn't roughly balanced (or non-bushy, or not like a linked list), then it could be O(N).

(c) If a method M of class A thows an exception and does not catch it, the exception will be caught by the parent class of A.

False. It might be caught by the calling method, but that doesn't have to be in the parent class.

(d) If we have two algorithms A1 and A2, and A1 takes time O(N) while A2 is $O(N^3)$, then A1 always runs faster than A2, for any input.

False. For very large N, A1 will be faster than A2, but perhaps not for smaller N.

Part II: SHORT ANSWER: Answer the question and give an explanation.

(e) How many non-null links are there in a doubly-linked list with N nodes?

2N-2. Each node has two outgoing links, but the last ones in each direction are null.

(Or: 2N. Counting the head and tail pointers, each node is pointed to by two non-null pointers.)

(f) How many non-null links are there in a binary tree with N nodes?

N-1. There's one incoming to each node except the root.

1. (continued)

(g) Given a method header

```
public void doSomething(params) throws ExceptionA, ExceptionB, ExceptionC
```

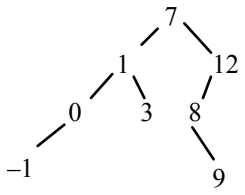
can any exceptions other than `ExceptionA`, `ExceptionB` and `ExceptionC` be thrown in the body of `doSomething()`?

Yes. Any RuntimeException can be thrown without a "throws".

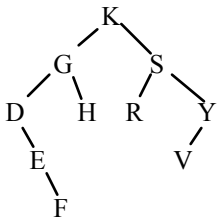
(h) Is inserting an element into a stack always more efficient than inserting it into a queue?

No. It depends on the implementation.

(i) The integers 7, 1, 12, 8, 3, 0, –1, 9 are inserted in that order into an initially empty binary search tree. Draw the tree after the last insertion. (No explanation is required.)



(j) Here is a binary tree:



Write the node labels in the order they would be printed in an in-order traversal of the tree. (No explanation is required.)

D E F G H K R S V Y

(k) If we perform merge sort as in the lectures and lecture slides, are the subarrays being merged always adjacent to each other?

Yes. Every merge reverses a previous split, and the splits always make adjacent subarrays.

2. [20 marks = 10 + 10]

This question has two parts, both referring to the same original code for an array-based stack class. This code compiles successfully and passes reasonable tests:

```
public class Stack {
    private final int MAX = 100;
    private Object[] stack = new Object[MAX];
    private int tos = 0;

    // Representation invariant: tos >= 0
    // If tos = 0, this Stack is empty.
    // If tos > 0, the contents of this Stack are stack[0], stack[1],
    //   ..., stack[tos - 1].


    // Precondition:   ! isFull()
    //
    public void push(Object data) {
        stack[tos] = data;
        tos += 1;
    }


    // Precondition:   ! isEmpty()
    //
    public Object pop() {
        tos = tos - 1;
        return stack[tos];
    }


    // Precondition:    ! isEmpty()
    //
    public Object top() {
        return stack[tos - 1];
    }


    // Precondition:    none
    //
    public boolean isEmpty() {
        return tos == 0;
    }


    // Precondition:    none
    //
    public boolean isFull() {
        return tos == MAX;
    }
}
```

(a) Before each method of the `Stack` class shown above, we have left space for a precondition. Fill in each space with the word "none" if no precondition is required, or with an appropriate precondition.

2. (continued)

Here is the same code again (without the space for preconditions).

```
public class Stack {
    private final int MAX = 100;
    private Object[] stack = new Object[MAX];
    private int tos = 0;     private int tos = -1; // some other value OK?

    // Representation invariant: tos >= 0
tos >= -1
    // If tos = 0, this Stack is empty.
if tos = -1, the stack is empty
    // If tos > 0, the contents of this Stack are stack[0], stack[1],
    //   ..., stack[tos - 1].
if tos >= 0, the contents are stack[0],...,stack[tos]

    public void push(Object data) {
        stack[tos] = data;    tos += 1;  // just swap the two lines
        tos += 1;             stack[tos] = data;
    }

    public Object pop() {
        tos = tos - 1;      Object result = stack[tos];
        return stack[tos];  tos = tos - 1;
    }                       return result; // or just return stack[tos--];

    public Object top() {
        return stack[tos - 1];   return stack[tos];
    }

    public boolean isEmpty() {
        return tos == 0;    return tos == -1;
    }

    public boolean isFull() {
        return tos == MAX;  return tos == MAX - 1;
    }
}
```

(b) The private variable `tos` in the `Stack` class is the index of the array element that would be filled the next time `push()` is called. Modify the code so that `tos` is the index of the top element actually in use. In other words, `tos` is to be the index of the top array element occupied by a value that has been "pushed" onto the stack.

Write your changes on the code above. Don't forget to fix the comments. You do not need to add preconditions as in part (a).

continued …

3. [13 marks = 8 + 5]

(a) Here is a method we wish to analyze:

```
 1    public static int[] g(int N) {
 2        int[] a = new int[N]; // cost: O(N)
 3        for (int i = 0; i < N; i++) {
 4            a[i] = N*i;
 5            for (int j = 0; j < N; j++) {
 6                a[i] += i*j;
 7            }
 8        }
 9        for (int i = 0; i < N; i++) {
10            a[i] -= i*i;
11        }
12        return a;
13    }
```

What is the time complexity of the method `g()`? That is, what is its big-O classification?

Show your work; only 1 mark is allotted to the actual answer. You may refer to the lines of the code by mentioning the numbers beside the lines.

```
Line 2: cost = N

Lines 3-8:
  Inner loop, lines 5-7:
    Loop body costs 1
    Number of iterations = N
    Total loop cost = N*1
  Line 4: cost = 1
  Cost of outer loop body = 1 + N*1 = O(N)
  Number of iterations = N
  Total loop cost for outer loop = N*O(N) = O(N^2)

Lines 9-11:
  Loop body costs 1
  Number of iterations = N
  Total loop cost = 1*N

Line 12: cost = 1

Total cost = N + O(N^2) + N + 1 = O(N^2)
```


continued …

3. (continued)

(b) Suppose `h(int M)` is a method that:

- has complexity O(t(M)), where *t* is some mathematical function;
- returns the value u(M), where *u* is some other mathematical function. (Perhaps `h`'s name ought to be "`calculate_u`".)

Also, suppose that some other method contains this loop:

```
int sum = 0;
for (int i = 0; i < h(N)*h(N); i++) {
    sum += i;
}
```

What is the complexity of the loop?

Assume that the value of N has been calculated or read from input before the loop begins, and that `h(N)` needs to be calculated every time it is encountered in the "`for`" statement. (Some compilers might be able to avoid some recalculations, but you are to ignore that possibility.)

Again, show your work.

```
Initialization: cost = 1

Loop:
  Body cost = 1
  Number of iterations = h(N)*h(N) = u(N)^2
  At each iteration, we have to evaluate the test "i < h(N)*h(N)",
    and that costs 2*t(N).
  Total cost = number of iterations * cost of an iteration
    = u(N)^2 * (1 + 2*t(N))
    = O(u(N)^2 * t(N))
```


continued …

4. [20 marks = 2 + 2 + 3 + 5 + 5 + 3]

In class, we studied binary search trees that do not allow us to insert duplicate elements. However, sometimes we do need to store duplicates. For example, a database of student marks might contain one record for every mark by every student; so if you've taken two courses, there will be two records with the same key (your student number) and different data (your two marks). To accomplish this, we might use a data structure called a "BST with duplicates", or BSTD.

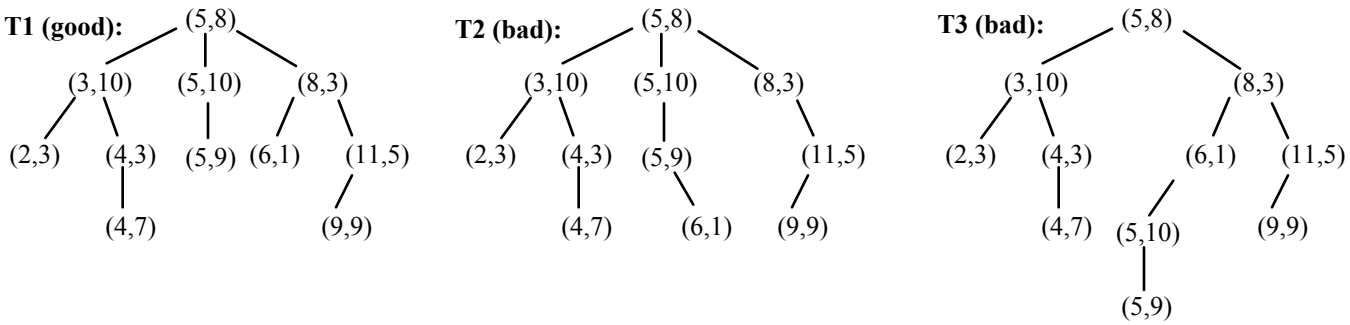
A node in a BSTD is defined as follows:

```
public class BSTDNode {
   public Comparable key;       /* for inserting and searching  */
   public Data data;            /* the rest of the data */
   public BSTDNode  left;       /* left subtree */
   public BSTDNode  same;       /* subtree of nodes with the same key */
   public BSTDNode  right;      /* right subtree */
   public BSTDNode (Comparable key, Data data) {
      this.key = key; this.data = data;
   }
}
public class Data {
    // various fields and methods that are not important
}
```

We assume that if we attempt to find a node with a key that is already present, we simply create a new node with this key, linking to it through the "`same`" instance variable. The order of the nodes with the same `key` is not important.

In the following two examples, we suppose that both `key` and `data` are integers, so (1, 3) refers to a node where key is 1 and data is 3.

Here are three trees. In each node, we show the `key` field and then the `data` field; for example, the root of the first tree is "(5,8)", meaning that `key` is 5 and `data` is 8. T1 is a valid BSTD, but T2 and T3 are not valid.



T2 is invalid because a node down the sequence of `same` links for key 5 has a non-null `right` child.

T3 is invalid because identical keys (5) are found in two places that are not connected by `same` links.

continued …

4. (continued)

(a) [2 marks]. List the elements of T1 (the first example tree on the previous page) in pre-order. Write a node as (key, data). Remember that the order of nodes is determined by the `key` value, not by the `data` value. Within a "same" sequence, proceed from top to bottom.

`(5,8) (5,10) (5,9) (3,10) (2,3) (4,3) (4,7) (8,3) (6,1) (11,5) (9,9)`

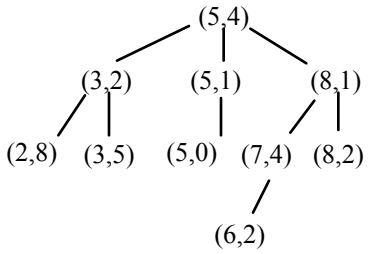
(b) [2 marks]. List the elements of T1 in post-order.

`(2,3) (4,3) (4,7) (3,10) (6,1) (9,9) (11,5) (8,3) (5,8) (5,10) (5,9)`

(c) [3 marks]. Suppose these elements have been inserted into an initially empty BSTD, in the order shown:

(5,4) (3,2) (8,1) (7,4) (3,5) (8,2) (2,8) (5,0) (6,2) (5,1)

Draw the resulting BSTD, showing it schematically, as in the example diagrams on the previous page, with "`/`" indicating a left child, "`\`" indicating a right child, and "`|`" indicating a "same child".



4. (continued)

Now we declare a class `LinkedBSTD` as follows:

```
public class LinkedBSTD {
   private BSTDNode root;  // root of the tree
   // various methods
}
```

(d) [5 marks]. Write a method `count()`, in the LinkedBSTD class, that counts the number of occurrences of a node with a given `key` in the tree. If the node is not in the tree, it returns 0. You may assume the existence of this method:

```
private static BSTDNode contains (BSTDNode t, Comparable k) {
```

The method `contains()` returns a reference to a highest descendant of `t` that has key `k`. If `t` itself has key `k`, then the return value is `t`. If no descendant of `t` has key `k`, then `null` is returned. Remember: *you don't have to write `contains()`*.

Your method `count()` may be iterative or recursive. Here is its header line to get you started:

```
public int count (Comparable k) {
```

ANSWER:

```
  BSTDNode where = contains(root, k);
  if (where == null) { Oops! we don't need this. (But it's OK.)
    return 0;
  }

  int count = 0;
  BSTDNode p = where;
  while (p != null) {
    count++;
    p = p.same;
  }
  return count;
}
```

a recursive count():

```
public int count(Comparable k) {
  return count(contains(root, k));
}

private int count(BSTDNode current){
  if(current == null) {
    return 0;
  }
  else {
    return 1 + count(current.same);
  }
}
```

4. (continued)

(e) [5 marks]. Write a method to insert a node with a given key and data into a BSTD.

You may *not* assume the existence of `contains()` from part (d). You may assume it is all right to have two nodes with both equal "`key`" fields and equal "`data`" fields.

```
public void insert (Comparable k, Data d) {
```

ANSWER:

```
  root = insert(root, k, d);
}

private BSTDNode insert(BSTDNode root, Comparable k, Data d) {
  if (root == null) {
    return new BSTDNode(k, d);
  }
  int comp = k.compareTo(root.key);
  if (comp == 0) {
    BSTDNode node = new BSTDNode(k, d);
    node.same = root.same;
    root.same = node;
    return root;
  }
  else if (comp < 0) {
    root.left = insert(root.left, k, d);
    return root;
  }
  else {
    root.right = insert(root.right, k, d);
    return root;
  }
}
```

(f) [3 marks]. The maximum number of nodes in an ordinary binary tree of height h is $2^{h+1}-1$. In a BSTD of height h, is the maximum number of nodes $3^{h+1}-1$? Explain your answer.

```
No. There is less branching down the "same" list, so the maximum number of
nodes must be less than 3^(h+1) - 1.
```

5. [15 marks = 10 + 5]

We have used iterators to "run through" all the elements of a list or other data structure. It is possible to use an iterator to run through other sequences of values, and in this question we will use iterators to run through integers with the goal of listing prime numbers. (A prime number is a positive integer divisible only by itself and 1. The first few prime numbers are 2, 3, 5, 7, …)

As a reminder, the `Iterator` interface looks like this:

```
public interface Iterator {
    boolean hasNext();
    Object next(); // throws NoSuchElementException if there is no "next"
    // Iterator also has a method remove(). We ignore it in this question.
}
```

We are going to write two iterators: one to list all the divisors of a given integer, and the other to list the prime numbers one after another.

(a) Write a class `DivisorsIterator` that implements `Iterator`. The constructor for a DivisorsIterator takes one parameter `k`, an `int`. The `next()` method returns the divisors of `k` between 2 and `k` – 1. (Of course, if `k` is a prime number, there are no such divisors.) For example, a `DivisorsIterator` created with "`new DivisorsIterator(12)`" would return `Integer`s representing 2, 3, 4 and 6 in that order.

The divisors are returned as objects of the class `Integer`. Here are two facts about Integer that might be helpful:

- You can create an Integer representing the number 5 by giving the command

  ```
  Integer myInt = new Integer(5);
  ```

- The method `myInt.intValue()` returns the value of `myInt` as an `int`.

Here are some parts of `DivisorsIterator` to get you started:

```
class DivisorsIterator
   private int k;
                // What other instance variable(s) might you need?


   public DivisorsIterator(int k) {
```

ANSWER:

```
   (DivisorsIterator ought to implement Iterator.
    Be a little nice if they point this out.)
   private int nextDiv; // another instance variable

   public DivisorsIterator(int k) {
     this.k = k;
     nextDiv = 1;
     getReadyForNext();
   }

   private void getReadyForNext() { // named from the May 2004 exam
     nextDiv++;
     while (nextDiv < k) {  // OK to omit all those silly {} pairs
       if (k % nextDiv == 0) {  // nextDiv is a divisor of k
         return;
       }
       nextDiv++;
     }
   }
```

5. (continued)

```
    public boolean hasNext() {
      return nextDiv < k;
    }

    public Object next() {
      if (! hasNext()) {
        throw new NoSuchElementException();
      }
      Object result = new Integer(nextDiv);
      getReadyForNext();
      return result;
    }
  }
```

(b) Write a class `Primes` that implements `Iterator`. Its constructor takes no parameters, and its `next()` method returns 2, 3, 5, 7 and so on, represented as `Integer`s. Its `hasNext()` always returns `true`, because there are an infinite number of prime numbers. (We ignore the fact that the `Integer` and `int` types in Java have maximum values.)

Use `DivisorsIterator` from part (a) to complete this part.

ANSWER:

```
public class Primes implements Iterator {
  private int nextPrime = 2;

  public boolean hasNext() {
    return true;
  }

  public Object next() {
    Object result = new Integer(nextPrime);
    while (true) {
      nextPrime++;
      if (! new DivisorsIterator(nextPrime).hasNext()) {
        break;
      }
    }
    return result;
  }
}
```

6. [10 marks = 3 + 7]

(a) Here are the definitions for a list node class and a linked list that uses the list node class:

```
public class Node {
    public int data;
    public Node next;
    public Node(int d, Node n) {
        data = d;
        next = n;
    }
}

public class List {
    private Node front;

    public void printIt() {
        rPrint(front);
    }

    private void rPrint(Node p) {
        if (p != null) {
            rPrint(p.next);
            System.out.println(p.data);
        }
    }

    public void printIt2() {
        if (front == null || front.next == null) {
            return;
        }
        else {
            Node c = front;
            while (c.next != null) {
                c = c.next;
            }
            System.out.println(c.data);
            while (c != front) {
                Node p = front;
                while (p.next != c) {
                    p = p.next;
                }
                System.out.println(p.data);
                c = p;
            }
        }
    }
```

The methods `printIt()` and `printIt2()` are meant to do the same task. What is that task?

Print the list in reverse order.

continued …

6. (continued)

(b) One of the two methods `printIt()` and `printIt2()` in part (a) contains a bug (a programming error). What is the bug and under what circumstances will it affect the output?

```
printIt2() prints nothing if there is exactly one item in the list,
because when from.next == null it returns immediately.
```