Carleton UNIVERSITY

FINAL
EXAMINATION
Fall 2004

**DURATION: 3 HOURS** **No. Of Students: 20**

**Department Name & Course Number:** **Systems and Computer Engineering**
**SYSC 2100**

**Course Instructor (s): Thomas Kunz**

AUTHORIZED MEMORANDA

***NONE***

**Students MUST count the number of pages in this examination question paper before beginning to write, and report any discrepancy to a proctor. This question paper has ___________pages + cover page = ___ pages in all.**

**This examination question paper may not be taken from the examination room.**

|  |  |
|---|---|
| **In addition to this question paper, students require: an examination booklet** | **no** |
| **Scantron Sheet** | **no** |

**Name:** ____________________________________________

**Student Number:** ____________________________________________

## Question 1: Algorithm Analysis (10 marks)

Four “Mystery” classes have been created that implement various collections. Using different values for n, sample run-times were measured (similar to assignment 1). With these, hypothesize what the time estimate (log n, quadratic, etc.) is for each Mystery class:

**Mystery1.class**

```
    n Value     |    Run-Time
--------------------------------
  n = 30        | 1.797 seconds

  n = 60        | 7.204 seconds
```

**Mystery2.class**

```
  n Value  |    Run-Time
---------------------------
  n = 250  | 0.061 seconds

  n = 500  | 0.061 seconds
```

**Mystery3.class**

```
     n Value      |    Run-Time
----------------------------------
  n = 10000000    |  0.485 seconds

  n = 20000000    |  1.638 seconds
```

**Mystery4.class**

```
  n Value  |    Run-Time
---------------------------
  n = 8    | 0.0010 seconds

  n = 16   | 0.088 seconds
```

**Answers:**

```
Mystery1 -  O(n log n)

Mystery2 -  O(1)

Mystery3 -  O(n)

Mystery4 -  O(2^n)
```

**Question 2: ADT List (10 marks)**

The Java Collections Framework provides two implementations of the ADT List: *ArrayList* and *LinkedList*.

1. Explain why there are two distinct implementations of that ADT

   **Answer** (2 marks):
   **N**ot all operations are equally efficient. Some operations are more efficiently supported with an array-based implementation, some are more efficient with a linked-list-based implementation. In general, an *ArrayList* would be faster than a *LinkedList* for any task in which random access was critical. A *LinkedList* would be faster than an *ArrayList*: for inserting or deleting elements far from the last element in the list.

2. Briefly sketch the code for the following task (task1): for each of *n* indexes, randomly generated, retrieve the list element at that index.

   **Answer** (2 marks):
   ```
   public void task1 (List myList)
   {
     Random rand = new Random();

     for (int i=0;i<myList.size();i++)
       myList.get(Math.abs(rand.nextInt())%myList.size());
   } // method task1
   ```

3. Briefly sketch the code for the following task (task2): repeatedly remove the first element (at index 0) from a list until the list is empty

   **Answer** (2 marks):
   ```
   public void task2 (List myList)
   {
     while (!myList.isEmpty())
       myList.remove(0);
   } // method task2
   ```

4. Hypothesize the runtime complexity of each piece of code when using ArrayList and when using LinkedList.

   **Answer** (4 marks):
   ```
   task1 & ArrayList: O(n)
   task1 & LinkedList: O(n^2)

   task2 & ArrayList: O(n^2)
   task2 & LinkedList: O(n)
   ```

**Question 3: Bags, Stacks, and Queues (10 marks)**

1. Suppose that rateSet is a Bag object of Double elements. Write the code to print each element in rateSet whose value is greater than 0.5.

   **Answer** (4 marks):

   If you are using the facilities in Java 1.5 (as per textbook), the code is:

   ```
   for (Double d : rateSet)
           if (d > 0.5)
                   System.out.println (d);
   ```

   If you are using the facilities of Java 1.3 (as per lab), the code is:

   ```
   Iterator itr = rateSet.iterator();
   double result;
   while (itr.hasNext()) {
           result = (Double(itr.next()).double_val;
           if (result > 0.5)
                   System.out.println (result);
   }
   ```

2. Suppose we added each of the following methods to the ArrayList class:
   **public boolean** addFirst (E element)
   **public boolean** addLast (E element)
   **public** E getFirst()
   **public** E getLast()
   **public** E removeFirst()
   **public** E removeLast()
   Estimate worstTime(*n*) for each method.

   **Answer** (6 marks):

   addFirst: linear in *n*
   addLast: linear in *n* (because of the possibility of resizing)
   getFirst: constant
   getLast: constant
   removeFirst: linear in *n*
   removeLast: constant

**Question 4: Binary Trees/Binary Search Trees (15 marks)**

1. Define the *depth()* method for a binary tree. Assume that the internal representation of the tree is based on a the node structure we discussed in class:

```
private static class BTNode {
    Object element;
    BTNode left;
    BTNode right;
}
```

and that the class contains an instance variable root that contains the root of the tree.

```
   /**
    * The depth of this BinarySearchTree has been calculated and returned.
    *
    * @return an int containing the height of the BinarySearchTree.
    **/
   public int depth();
```

**Answer** (10 marks):

```
public int depth()
{
  return d(root);
}


protected int d (BTNode p)
{
  if (p==null)
    return -1;
  return Math.max(d(p.left), d(p.right)) + 1;
}
```

2. Describe in English how to remove each of the following from a binary search tree:

a. an element with no children

**Answer** (1 mark):
Simply remove the element and remove the corresponding subtree-link from the element's parent (if the element is not the root).

b. an element with one child

**Answer** (2 mark):
Replace the parent-element link and the element-child link with a parent-child link.

c. an element with two children

**Answer** (2 mark):
Copy the element's immediate successor into element, and then remove that immediate successor (by part a or part b).

**Question 5: Sorting (15 marks)**

1. InsertionSort is another way to sort a collection, besides HeapSort. Assume we want to sort an array of integers. The following code implements insertionSort:

```
public static void insertionSort (int[ ] x)
{
      for (int i = 1; i < x.length; i++)
            for (int j = i; j > 0 && x [j -1] > x [j]; j--)
                  swap(x[i], x[j]); // exchanges the values
} // method insertionSort
```

   Estimate the runtime complexity (averageTime(*n*) and worstTime(*n*)) for InsertionSort.

   **Answer** (5 marks):
   They are both O(n^2): the outer loop iterates over all N elements in the array, the inner loop iterates over up to I elements, which is upper-bounded by N as well.

2. Consider the following, consecutive improvements to Insertion Sort:
   a) Replace the call to the method swap with in-line code:

```
public static void insertionSort (int[ ] x)
{
      int temp;
      for (int i = 1; i < x.length; i++)
            for (int j = i; j > 0 && x [j -1] > x [j]; j--)
            {
                  temp = x [j];
                  x [j] = x [j - 1];
                  x [j - 1] = temp;
            } // inner for
} // method insertionSort
```

   b) Notice that in the inner loop in part a), temp is repeatedly assigned the original value of x [i]. For example, suppose the array x has 32 46 59 80 35 and j starts at 4. Then 35 hops its way down the array, from index 4 to index 1. The only relevant assignment from temp is that last one. Instead, we can move the assignments to and from temp out of the inner loop:

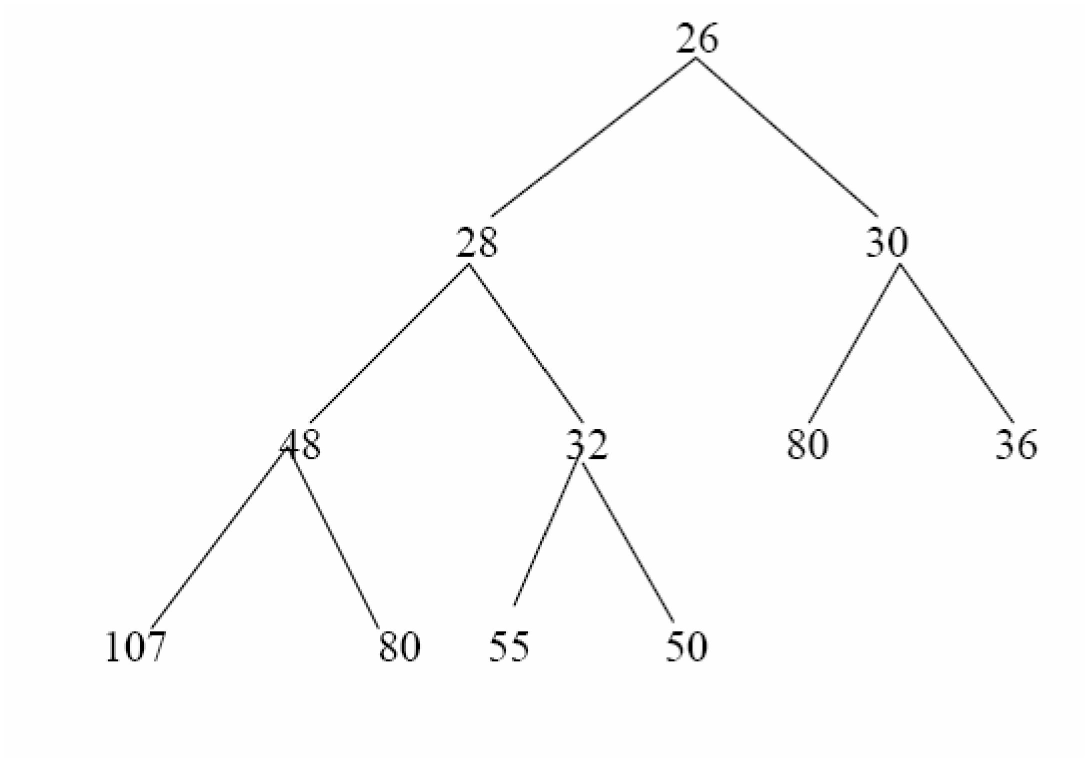
```
int temp, j;
for (int i = 1; i < x.length; i++)
{
      temp = x [i];
      for (j = i; j > 0 && x [j -1] > temp; j--)
            x [j] = x [j - 1];
            x [j] = temp;
} // outer for
```

   Will these changes affect the estimates for worstTime(*n*) and averageTime(*n*)?

   **Answer** (10 marks):
   The changes have no effect on the estimates for worstTime(n) or averageTime(n) because they have no effect on the number of innerloop iterations.
   But the improvements have a significant effect on run time because they reduce the number of statements executed during each inner-loop iteration. Basically, the first improvement reduces the number of statements executed during each inner-loop iteration from seven (two statements for the continuation condition, one statement for decrementing j, one statement for the call to swap, and three statements for the swapping) to six. The second improvement further reduces that number to four, and the two statements removed are timeconsuming moves. The elapsed time was reduced by about 30%.

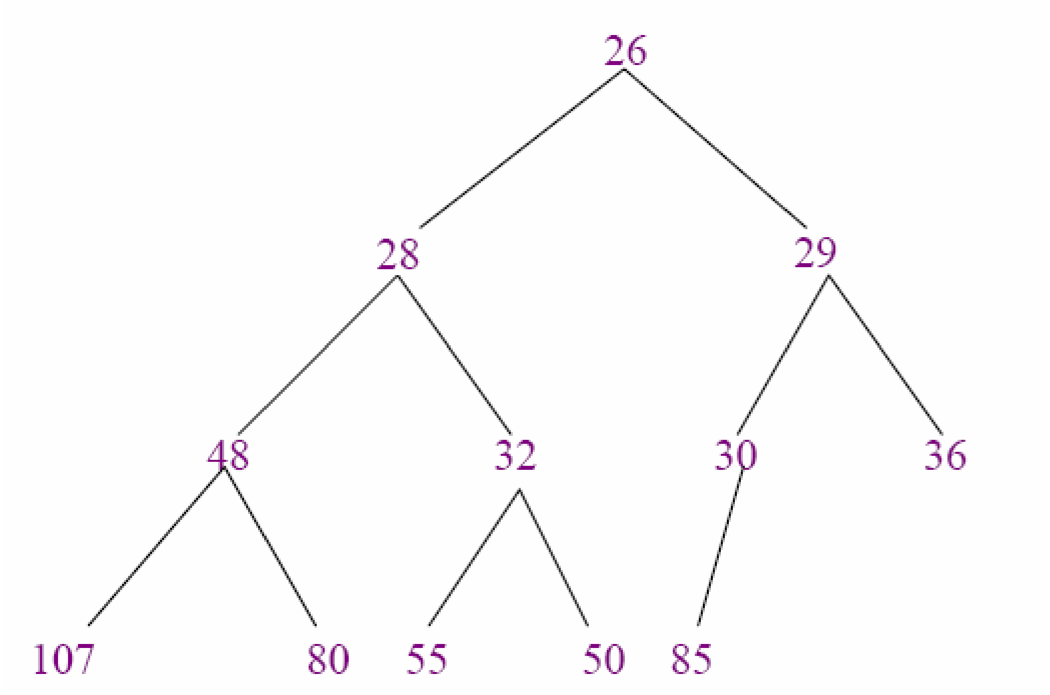
**Question 6: Heaps/Priority Queues (15 marks)**

1. Show the resulting heap after each of the following alterations is made, consecutively, to the following heap:



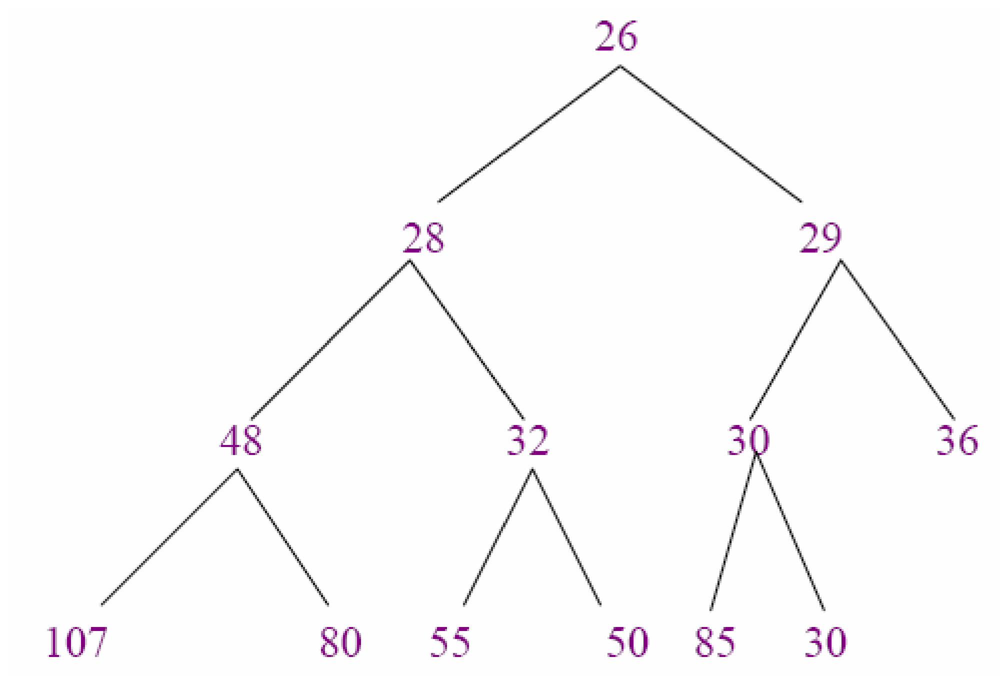
- add (29)

**Answer** (3 marks):



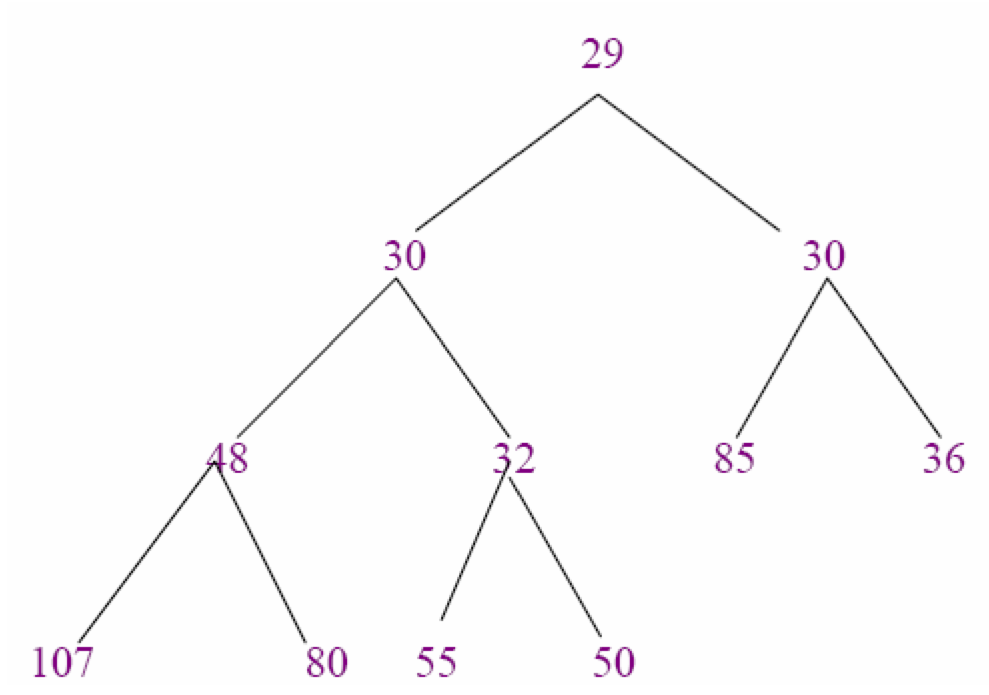
- add (30)

**Answer** (3 marks):



- removeMin(); removeMin()

**Answer** (3 marks):

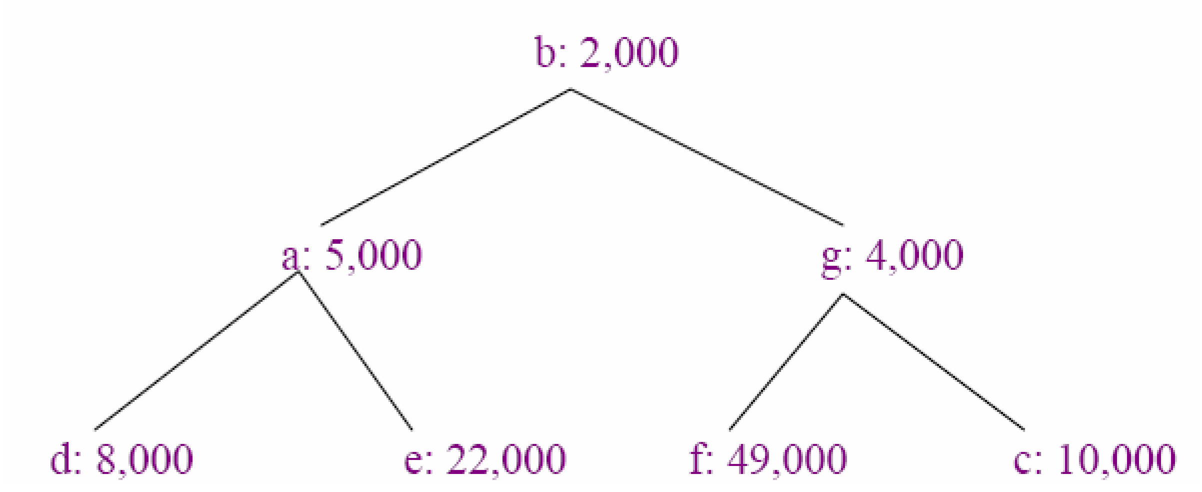


2. If each of the letters ‘a’ through ‘f’ appears at least once in the original message, explain why the following cannot be a Huffman code:
   a: 1100
   b: 11010
   c: 1111
   d: 1110
   e: 10
   f: 0

   **Answer** (3 marks):
   The code for ‘b’ can be shortened to 1101, so the given code cannot be a Huffman code because a Huffman code is minimal.

3. For the following character frequencies, create the heap of character-frequency pairs (highest priority = lowest frequency). Assume that the pairs are added in alphabetical order to an initially empty heap.
   a: 5,000
   b: 2,000
   c: 10,000
   d: 8,000
   e: 22,000
   f: 49,000
   g: 4,000

   **Answer** (3 marks):



**Question 7: Hashing (15 marks)**

1. Assume that $p$ is a prime number. Use modular algebra to show that for any positive integers *index* and *offset* (with offset not a multiple of $p$), the following set has exactly $p$ elements:
   { (*index* + *k* * *offset*) % *p*; *k* = 0, 1, 2, …, *p* – 1}

   **Answer** (5 marks):
   Because the value of $k$ ranges from 0 through $p - 1$, the set cannot have more than $p$ elements.

   Assume the given set has fewer than $p$ elements. Then there must be integers *k1* and *k2* such that *0 <= k1 < k2 < p* and (*index* + *k2* * *offset*) % *p* = (*index* + *k1* * *offset*) % *p*

   This implies that ((*k2* – *k1*) * *offset*) % *p* = 0 and therefore that (*k2* – *k1*) * *offset* is a multiple of *p*. Because *p* is prime, it follows that either *k2* – *k1* is a multiple of *p* or *offset* is a multiple of *p*.

   But *k2* – *k1* cannot be a multiple of *p* because both *k1* and *k2* are less than *p*. And *offset* was given as not being a multiple of *p*. This contradiction completes the proof that the given set has exactly *p* elements.

2. Compare the space requirements for chained hashing and open-address hashing. Assume that a reference occupies four bytes and a boolean value occupies one byte. Under what circumstances (size, loadFactor, table.length) will chained hashing require more space? Under what circumstances will double hashing require more space?

   **Answer** (5 marks):
   For each entry, chained hashing requires three more bytes than open-address hashing, so under normal circumstances, chained hashing will require more space. But suppose, for a certain application, a table size of about 100000 and a load factor of 2 is optimal in terms of time and space efficiency. Then if size is 110000, for example, chained hashing will require only 110000 entries, but open-address hashing will require 200000 entries because the load factor must be less than 1.

3. In open-addressing with double hashing, insert the following keys into a table of size 13:
   20, 33, 49, 22, 26, 202, 140, 508, 9
   The second hash function calculates the quotient of the key and the table size, also called open-addressing with quotient-offset collision handler.

   Here are the relevant remainders and quotients:

| **Key** | **key % 13** | **key / 13** |
|---|---|---|
| 20 | 7 | 1 |
| 33 | 7 | 2 |
| 49 | 10 | 3 |
| 22 | 9 | 1 |
| 26 | 0 | 2 |
| 202 | 7 | 15 |
| 140 | 10 | 10 |
| 508 | 1 | 39 |
| 9 | 9 | 0 |

**Answer** (5 marks):

| index | key |
|---|---|
| 0 | 26 |
| 1 | 508 |
| 2 | 202 |
| 3 | |
| 4 | 140 |
| 5 | |
| 6 | |
| 7 | 20 |
| 8 | |
| 9 | 33 |
| 10 | 49 |
| 11 | 22 |
| 12 | 9 |