STUDENT NAME: ______________________

STUDENT ID: ______________________

McGill University
Faculty of Science
School of Computer Science

FINAL EXAMINATION

COMP-250: Introduction to Computer Science - Fall 2011

December 13, 2011
2:00-5:00

Examiner: Prof. Doina Precup

Associate Examiner: Prof. Michael Langer

Write your name at the top of this page. Answer directly on exam paper. Three blank pages are added at the end in case you need extra space. There are 13 questions worth 350 points in total. The value of each question is found in parentheses next to it. Please write your answer on the provided exam. Partial credit will be given for incomplete or partially correct answers.

SUGGESTIONS: READ ALL THE QUESTIONS BEFORE YOU START! THE NUMBER OF POINTS IS NOT ALWAYS PROPORTIONAL TO THE DIFFICULTY OF THE QUESTIONS. SPEND YOUR TIME WISELY!

GOOD LUCK!

1. [50 points] **Short questions**

   (a) True or false: is $f(n) = 10n \log_2 n + n^2 - 50n$ in $O(n \log_2 n)$? Justify your answer.

   Solution: False. By the limits rule:

$$\lim_{n\to\infty} \frac{f(n)}{n \log_2 n} = \lim_{n\to\infty} \left(10 + \frac{n}{\log_2 n} - 50\frac{1}{\log_2 n}\right) = \infty$$

   due to the middle term (for which the limit can be shown using l'Hopital rule).

   (b) Suppose you have a Java program with two classes, A and B, which are in the same package. From class A, can you call any method of class B? Justify your answer.

   Solution: No, you cannot call methods that are declared private in B.

   (c) True or false: there exists an algorithm that can find a cycle that goes through each node of a graph exactly once, if such a cycle exists, in time polynomial in the number of nodes and edges in the graph.

   Solution: False. This is the Hamiltonian cycle [roblem, which is NP-complete (and hence could be solved in polynomial tome only if P=NP, which is believed to be false).

   (d) In a Java package, can more than one class have a main method?

   Solution: Yes (and you can choose which one will execute by specifying the name of the class whose main you want to execute).

   (e) You have to develop a text editor and you want to implement an "undo" feature. What data structure will you use for this task?

   Solution: A stack. When a modification is made, it is pushed onto the stack. The undo feature just has to pop, so the last change is disregarded and we revert to the previous state of the file.

2. [30 points]**Big-Oh**

   For the following algorithms, state what $O()$ is and explain your answer:

   (a)
```
**Algorithm** f1($n$)
$i \leftarrow 1$
**while** $i < n$
   **print**($i$)
   $i \leftarrow i + 5$
**while** $i > 1$
   **print**($i$)
   $i \leftarrow i/2$
```

   Solution: We have a sequence of two loops. The first one is $O(n)$. The second one is $O(\log n)$. This is because i starts at n, then we halve it every time. We will only be able to do this operation $\log_2 n$ times. As a result, the code is $O(n)$ (as the more expensive loop dominates $O()$)

   (b)
```
**Algorithm** f2($n$)
$i \leftarrow 1$
**while** $i < n$
   **for** $j = 1$ **to** $i$ **do**
       **print**($j$)
   $i \leftarrow i * 2$
```

   Solution: The outer loop is $O(\log n)$, the inner loop is $O(n)$, and the loops are nested, so using the product rule we get $O(n \log n)$.

3. [10 points] **Tree printing**

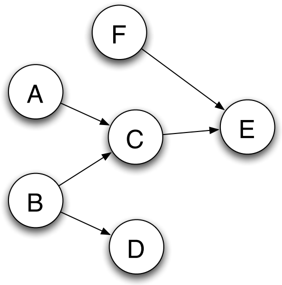
   Suppose you have a general tree and you want to print the tree layer by layer: first the root, then all nodes at depth 1, then all nodes at depth 2 etc. Explain what strategy you would use to achieve this task (no pseudocode is necessary).

   Solution: Breadth-first traversal, which in the case of trees is also known as pre-order traversal (we process a node before all its successors).

4. [50 points] **A graph problem**

   Suppose that you are taking an evening certification program, in which you take only one course each term. Courses have a prerequisite structure, but there are no co-requisites. You can only take a course after you have taken *all* its prerequisites. In order to help you make a schedule, the program advisor provides you with a graph, in which each course is a node, and each arc represents a prerequisite structure.

   For example, consider the graph below:



In this graph, A and B are prerequisites for C, B is a prerequisite for D and C and F are prerequisites for E. There are many legal schedules, for example: (A,B,D,C,F,E); (A,F,B,C,E,D); (B,A,D,C,F,E); and many others. The schedule: (A,C,B,D,F,E) is *not* legal, because C is taken before B (which is one of its prerequisites).

Write an algorithm that takes as argument such a graph and produces *a legal schedule* (it does not matter which one). You can make use of any data structures or algorithms discussed in class. You may assume that all of these are known, and do not need to reproduce them here.

Your algorithm must run in time polynomial in the size of the graph. Your algorithm is allowed also to modify the graph structure (adding or removing nodes or edges as needed).

Solution: This algorithm is also known as lexicographic ordering of the nodes in the graph. We will always look for a node with no arcs coming in. When we find such a node, we add it to the schedule (adding it at the end of a list). We then remove all outgoing arcs of this node, and the node itself. We will continue the process until no nodes are left (this terminates under the assumption that there are no cycles, which would make the study program badly designed).

Algorithm Ordering (Graph G)
Input: A directed acyclic graph G
Output: A List of courses, which will be taken in order

```
Double-linked list L ← empty
while (not G.vertices.isEmpty()
    for all $u$ ∈G.vertices
        if (u.inDegree=0) bf then
            for all $(u, v)$ ∈G.edges
                G.edges.remove($(u, v)$)
                v.inDegree← v.inDegree-1
            L.insert(u.content)
            G.vertices.remove(u)
return L
```

5. [10 points] **Inheritance**

Consider the following piece of Java code:

```
public class A {
        private int x;
        public A() { x=0; }
        public void increment() {x++};
        public void print() {System.out.println(x);}
}

public class B extends A {
        private int y;
        public B() { super(); y=2; }
        public void increment() { y++; }
        public void add {y = y+x; super.increment(); }
        public void print() {super.print(); System.out.println(y);}
}
```

What is the outcome of the following code:

```
B z = new B();
z.increment();
z.add();
z.print();
```

Solution: After the constructor call, z.x is 0 and z.y is 2. After the second line, z.y becomes 3. After the call to add, z.y stays the same (since z.x. is 0) and z.x becomes 1. Hence, the code prints:

```
1
3
```

6. [20 points] **More Big-Oh**

   Consider the following recursive algorithm:

```
Algorithm MyAlg (int $n$)

if ($n <= 1$) return 1
return $1 + 2*$MyAlg($n - 1$)
```

   (a) [10 points] Write a recurrence for the running time of the algorithm and use it to make a guess about the $O()$ of the algorithm

   Solution: We have:

$$\begin{aligned}
T(n) &= c + 2T(n-1) \\
&= c + 2\left(c + 2T(n-2)\right) = c(1+2) + 2^2T(n-2) \\
&= c(1+2) + 2^2\left(c + 2T(n-3)\right) = c(1+2+2^2) + 2^3T(n-3) \\
&= \ldots = c(1+2+2^2+\ldots 2^{n-1}) + 2^nT(0) = c\frac{2^n-1}{2-1} + c_0 2^n = 2^n(c+c_0) - c \in O(2^n)
\end{aligned}$$

   where we used also the formula for the geometric series.

   (b) [10 points] Prove by induction that the $O()$ you proposed is correct

   Solution: We need to show that $T(n) = 2^n(c+c_0) - c$ where $T(0) = c_0$.

   Base case: $T(0) = c_0 = c + c_0 - c$

   Induction step: $T(n) = c + 2T(n-1) = c + 2(2^n(c+c_0) - c) = 2^{n+1}(c+c_0) - c$ which concludes the proof

7. [60 points] **A different priority queue implementation**

   We talked in class about the implementation of priority queues using heaps. We will now assume that we have a system in which many requests may arrive which have *the same priority*. In this case, we want to process the requests with the same priority *in the order in which they arrive*; all requests with higher priority must be processed before requests with lower priority.

   Assume that you are given a Java class called Queue, which has the following methods:

   - Queue() - constructs an empty queue
   - void enqueue(Request r) - enqueues a request object
   - Request dequeue() - removes the first Request object from the queue and returns it; returns null if the queue is empty

   You are also give the LinkedList⟨E⟩ Java class, which can implement a list of objects of a specified type. You will need to make use of the following methods:

   - LinkedList() - constructs an empty list
   - void add(int index, E element) - adds the specified element in the requested position in the list
   - ListIterator⟨E⟩ listIterator (int index) - returns a list iterator starting at index
   - E remove(int index) - removes and returns the element at index
   - int size() - returns the number of elements in the list

   The ListIterator⟨E⟩ has two methods that you need

   - boolean hasNext() - returns true if there is at least one element left
   - E next() - returns the next element in the list

   Provide a class PriorityQueue.java, which implements the priority queue as a *list of queues*, ordered in decreasing order of priority. You should implement the following methods:

   (a) PriorityQueue() - constructs an empty priority queue

   (b) void enqueue (Request r, int priority) - this method looks for a queue of elements of the specified priority. If one exists, it enqueues r to this queue. If no such queue exists, it creates one, enqueues r in it, and puts it in the right place in the list of queues.

   (c) Request dequeue() - this method returns the request with the highest priority. While looking for requests, if it finds empty queues, it should remove them.

   Please make sure that you declare all needed variables and you write the code for all the methods.

Solution:

```
public class PriorityQueue {
LinkedList<Integer> priorities = new LinkedList<Integer>();
LinkedList<Queue> queues = new LinkedList<Queue>();

public PriorityQueue() { }
public void enqueue (Request r, int priority) {
    int i = 0;
    ListIterator<Integer> iter = priorities.listIterator(i);
    while (iter.hasNext()) {
        int p = iter.next().getValue();
        if (p==priority) {
            queues.listIterator(i).next().enqueue(r);
            return;
        } else if (p<priority) {
            Queue q = new Queue();
            queues.add(i,q);
            q.enqueue(r);
            priorities.add(i,new Integer(priority));
            return;
        }
        i++;
    }
    // If we get here, we must make a new queue, this is currently lowest priority
    Queue q = new Queue();
    queues.add(i,q);
    q.enqueue(r);
    priorities.add(i,new Integer(priority));
    return;
}
public Request dequeue() {
    int i = 0;
    ListIterator<Integer> iter = queues.listIterator(0);
    while (iter.next().size()==0) {
        queues.remove(i);
        priorities.remove(i);
    if (iter.hasNext()) iter.next().dequeue();
    }
    return;
}
}
```

8. [10 points] **Binary trees**

   Here is a mystery algorithm that works on binary trees:

```
**Algorithm** Mystery (BinaryTreeNode n)

if (n==null) return 1
return 1 + Mystery(n.left) + Mystery (n.right)
```

   What quantity does the algorithm compute when called with the root of a binary tree?

   Solution: It computes the number of nodes in the tree.

9. [20 points] **Heaps**

   Write, in pseudocode, an algorithm that takes as input two heaps h and k and returns a new heap l, which is the result of merging h and k. In other words, l should be a heap and contain all the elements of h and k. You may assume that you are provided the usual methods for heaps, and also that the heaps are implemented using arrays. State the $O()$ of your algorithm.

   Solution: There are several possible answers, but the easiest is to clone one heap, and then repeatedly removeMin() from the second heap and insert it.

```
Algorithm MergeHeaps( Heap $h$, Heap, $k$)
Input: Two heaps
Output: A heap that contains the elements of both.

Heap $l$ = $h$.clone();
while (!$k$.isEmpty()) {
    $l$.insert(k.removeMin())
}
```

   The code assumes that the removeMin returns an object that has both the content and the priority of the top element.

10. [30 points] **Binary trees**

Suppose that you are given a BinaryTree package in Java, with two classes: BinaryTree.java and BinaryTreeNode.java. The BinaryTree class contains a field called *root*, which is of type BinaryTreeNode. The BinaryTreeNode.java class contains the following fields:

```
Object content;
BinaryTreeNode left;
BinaryTreeNode right;
```

(a) [20 points] Write an "equals" method for the BinaryTree class. If needed, you can also add methods to the BinaryTreeNode class. The method you write should test for the equality of the current tree object with the tree passed in as a parameter. If the two are equal, it should return true, otherwise it should return false.

Solution: The method in the BinaryTree class is just a wrapper, calling the method in BinaryTreeNode, which does the interesting work.

```
public boolean equals(BinaryTree t){
    return root.equals(t.root);
}
```

Inside BinaryTreeNode, we have a recursive method which has to compare the content of the two nodes.

```
public boolean equals(BinaryTreeNode n) {
    if (n==null) return false;
    if (!content.equals(n.content)) return false;
    boolean result=true;
    if (left==null) result = result && (n.left==null);
    if (right==null) result = result && (n.right==null);
    if (!result) return result;
    return (left.equals(n.left) && right.equals(n.right));
}
```

(b) [5 points] What is the $O()$ of your method, assuming each of the two trees has n nodes? You do *not* have to prove the answer.

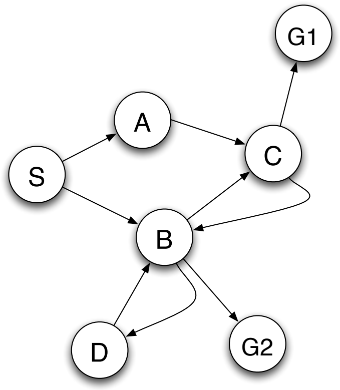
Solution: This is $O(n)$, where n is the number of nodes in the largest tree. This is because in the worst case, we need to touch every pair of corresponding nodes to see if they are equal.

(c) [5 points] If you wrote the same method for binary search trees, could you make its $O()$ better? Justify your answer.

Solution: No, we would still need to comport both the left and right subtrees to determine equality.

11. [20 points] **Search in graphs**

Consider the graph in the figure below.



(a) Suppose that you are using breadth-first search to find a path from vertex S to one of the vertices G1 and G2 (you will stop when you found the first one of these). Show the state of the queue after each node expansion, assuming that successors of a node are enqueued in *reverse alphabetical order*. What path do you find?

Solution: We start with the queue containing just S. We dequeue S, and the queue becomes: B A. We dequeue B and enqueue its successors, so we get: A G2 D C. We dequeue A and enqueue its successors, so the queue becomes: G2 D C. Note that we assume we used a "visited" flag to mark C and not enqueue it again (otherwise, it would appear in the queue twice, once for the path through B and once for the one through A). Finally, we dequeue G2 and stop, since this is a goal state. The path is S - B - G2 (and note that this is the shortest path from S to a goal state).

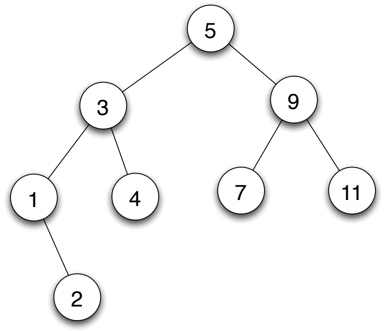
(b) Suppose that you are using depth-first search for the same task. Show the state of the stack after each node expansion, assuming that successors of the same node are pushed in *reverse alphabetical order*. What path do you find?

Solution: The first stack is:
A
B
We pop A, mark it as visited, and push its successors, so the stack becomes:
C
B
We pop C, mark it as visited, and push its successors:
B
G1
B
We pop B, mark it as visited, and push its successors that have not been visited yet, so the stack becomes:
D
G2

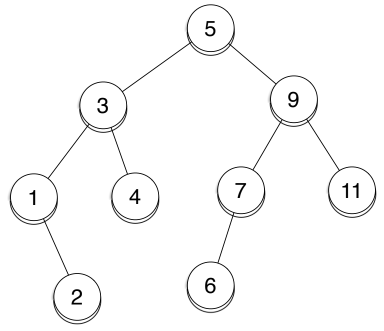
G1
B
We pop D, mark it as visited, but since B has already been visited, we do not put it back on the stack anymore. Finally, we pop G2, which is a goal state, and this end the algorithm. The corresponding path is: S - A - C - B - G2. Note that this ends at the same goal state as breadth-first search but through a circuitous route.

12. [20 points] **Binary Search Trees**

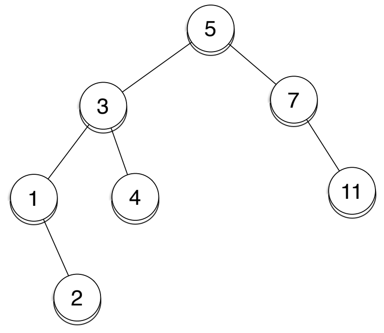
Consider the binary search tree in the figure below.



(a) Draw the tree resulting from the insertion of value 6.



(b) Draw the tree resulting from the removal of value 9 from the initial tree (not the tree after the insertion of 6).



(Note that we could have also "promoted" 11 instead of 7, that is a correct solution as well).

13. [20 points] **Using data structures**

    Suppose that you are hired by an English professor at your old high school for a summer project. You are supposed to write a software that can detect plagiarists among the students.

    All students submit their essays as typed, simple text programs. You have access to their files, as well as to a large data base of "pre-made" assignments that your professor downloaded form various web sites (roughly 1000 files). Your program should output pairs of files that it considers unusually similar.

    (a) [10 points] As a first try, you consider simply counting the number of words that two documents have in common, and sorting all the pairs of files in decreasing order of this measure. Describe what data structures and algorithms you would use for this problem, and why.

    Solution: Several correct approaches are possible, but the one that comes to mind most quickly is to use a very sparse vector (implemented as an array), with one entry for each possible word in the dictionary. The entries are 1 if the word is in the document and 0 otherwise. Hence, counting the number of words in common of two documents amounts to doing a dot-product of the two vectors (Which amounts to a loop throughout the two vectors). We can then sort the documents by this quantity, or we can use a heap so we can always extract the pair that is most similar.

    (b) [10 points] A better way to detect plagiarism is to look for common or very similar phrases, rather than just counting isolated words. In this case, you need to find sequences of several words that are common between the documents and count them. What algorithms and data structures would you use in this case? Justify your answer.

    Solution: Again, several answers are possible, but we will continue in the spirit of the previous answer. If there are particular phrases we want to target, we can use the same approach as above, but with a larger "dictionary" that includes both words and phrases. If that is too big, we need to search for pairs of words from one document in the other. An inverted index (hash table with words as keys and documents in buckets) would be helpful to see if, given one word, the following ones are also in the document. IF so, we would then scan to see if the words are adjacent.