CPSC 331 — Term Test #1 Solutions

February 12, 2007

Name: ____

Please **DO NOT** write your ID number on this page.

Instructions:

Answer all questions in the space provided.

Point form answers are acceptable if complete enough to be understood.

No Aids Allowed.

There are a total of 45 marks available on this test.

Duration: 90 minutes

ID Number: ____________________________________

0.4pt0pt

Question	**Score**	**Available**
1		10
2		8
3		6
4		4
5		10
6		7
Total:		45

0.4pt0pt

(10 marks) 1. Short answer questions — you do *not* need to provide any justifications for your answers. Just fill in your answer in the space provided.

(a) True or false: black-box tests of the functions of an ADT can be determined by someone who has access to the interface but not the implementation.

Answer: T

(b) True or false: when used together, the proper use of black-box and white-box testing guarantees the correctness of a program.

Answer: F

(c) True or false: an algorithm that runs in worst-case time $f(n)$ for inputs of size n is faster for all worst-case inputs than an algorithm that runs in time $g(n)$ if $f(n) \in o(g(n))$.

Answer: F

(d) True or false: if f and g are functions such that $f \in o(g)$, then $g \in \omega(f)$.

Answer: T

(e) In general, is the non-empty queue `q` in its original state after the statement `q.enqueue(q.dequeue())` is executed? Yes or no?

Answer: No

(f) What are the maximum and minimum numbers of leaf nodes in a binary tree with 5 nodes?

Maximum: <u>3</u>   Minimum: <u>1</u>

(g) Consider the `insert` function for the `dictionary` abstract data type. Using big-Oh notation, fill in the following table to indicate the asymptotic running time as a function of n, where n is the number of entries in the dictionary, assuming that a search has already been performed to determine whether the element to insert is already in the dictionary.

Data Structure	**worst-case running time**
unordered array	$O(1)$
ordered linked list	$O(n)$
binary search tree	$O(n)$

2. Consider the following algorithm that searches an integer array for a specified value.

PRECONDITION: `k` is a nonnegative integer, `A` is non-null array of integers
POSTCONDITION: `idx = -1 OR (idx >= 0 AND A[idx] = k)`

```
int idx = -1
int i = 0
int n = A.length
while i < n AND idx < 0 do
  if A[i] == k then
    idx = i
  end if
  i = i + 1
end while
return idx
```

(1 marks) (a) Give a loop invariant for the loop in this algorithm.

Solution:

```
I(j):  i=j; 0 <= i <= n;
       ((idx >= 0) && (k == A[idx])) ||
         (idx == -1 && k != A[l] for 0 <= l < i)
```

Comments: One common error was to use the postcondition verbatim as the loop invariant. This is not correct because the loop invariant has to be dependent on the loop index i.

(3 marks) (b) What three properties have to be satisfied by this loop invariant?

Solution:

- $I(0)$ is true before the first execution of the loop body.
- If $I(i)$ is true after the ith execution of the loop body and there is a $i+1$st execution, then $I(i+1)$ is true after the $i+1$st execution.
- If there is an ith execution but not a $i+1$st execution, then $I(i)$ implies the postcondition.

Comments: This question was fairly well-done — we were just looking for the definition of the properties here as opposed to a proof that the loop invariant from the previous question satisfies them.

(2 marks) (c) Give a **loop variant** for the loop in this algorithm. You do not need to justify your answer.

Solution: The loop variant is:

$$f(n, i) = n - i$$

Justification (*not* required for your answer):

- i increases by 1 after each iteration of the loop so $f(n, i) = n - i$ decreases by 1,
- $f(n, i) = 0$ when $i = n$ and the loop terminates when $i = n$)

Comments: This question was fairly well-done. Some common errors:

- the loop variant must at least be a function of n and i. Some people incorporated idx, but this is not necessary because one only has to show that if the loop variant is ≤ 0 the loop terminates — it is not necessary to capture every possible termination condition.
- $f(n, i) = n - 1 - i$ is not correct because this function is equal to 0 when $i = n - 1$ but the loop does *not* terminate in that case.

(2 marks) (d) Use your loop variant to derive a worst-case bound on the number of iterations of the loop. Simply stating the bound without justifying how it is derived from the loop variant will earn only 1 mark.

Solution: The loop variant evaluated at the initial value of i gives an upper bound on the number of iterations of the loop. In this case, we get $f(n, 0) = n - 0 = n$.

Comments: Many people gave worst-case bounds on the number of steps as opposed to the number of iterations of the loop. Credit was nevertheless awarded if there was a clear statement as part of the analysis about the number of iterations of the loop.

3. Assume that f, g, and h are functions mapping the natural numbers to the natural numbers:

$$f, g, h : \mathbb{N} \to \mathbb{N}.$$

(2 marks) (a) Define **big-Omega** by saying what it means when "$f \in \Omega(g)$." A written definition is required (pictures will receive no marks).

Solution: $f \in \Omega(g)$ if there exist real constants $c > 0$ and $N_0 \geq 0$ such that

$$f(n) \geq c \cdot g(n)$$

for all $n \geq N_0$.

Comments: The most common errors were:

- not defining the constants c and N_0
- forgetting the $n \geq N_0$ condition

(4 marks) (b) Prove that if $f \in \Omega(h)$ and $g \in \Omega(h)$, then $f + g \in \Omega(h)$.

Solution: We need to show that the definition of $f+g \in \Omega(h)$ is satisfied, specifically, that there exist real constants $c > 0$ and $N_0 \geq 0$ such that $f + g \geq ch$ for all $n \geq N_0$.
By the definition of Ω we have:

- If $f \in \Omega(h)$, there exist constants $c_1 > 0$ and $N_1 \geq 0$ such that $f \geq c_1 h$ for all $n \geq N_1$.
- If $g \in \Omega(h)$, there exist constants $c_2 > 0$ and $N_2 \geq 0$ such that $f \geq c_2 h$ for all $n \geq N_2$.

Thus, for all $n \geq \max(N_1, N_2)$ we have

$$\begin{aligned} f + g &\geq c_1 h + c_2 h \\ &= (c_1 + c_2)h \end{aligned}$$

and the result follows if we take $c = c_1 + c_2$ and $N_0 = \max(N_1, N_2)$.

Comments: Some common errors were:

- using the same constants for the functions f and g when applying the Ω definition
- omitting the $n \geq N_0$ condition required to prove $f + g \in \Omega(h)$
- only providing informal intuitive arguments. These do *not* constitute a proof, and are only an aid to intuitive understanding of the concept.

4. Consider the behavior of the following algorithm when it is given a positive integer n as input:

```
int count = 0
for i from 1 to n do
  for j from n downto 1 do
    for k from 1 to n/2 do
      count = count + 1
    end for
  end for
end for
```

(2 marks) (a) Give a function $T(n)$ such that the above algorithm uses $\Theta(T(n))$ steps on input n.

Solution: Simple answer: $T(n) = n^3$. Note that a precise formula is *not* required for this question. You only need to give a function $T(n)$ such that $f(n) \in \Theta(T(n))$ where $f(n)$ is some function that precisely describes the number of steps.

More complicated answer: carefully analyze the number of steps in this algorithm to obtain something like

$$\begin{aligned} T(n) &= 1 + n\left[n\left(4(n/2) + 2 + 2\right) + 2\right] + 2 \\ &= 2n^3 + 4n^2 + 2n + 3 \end{aligned}$$

Comments: This was fairly well-done. Almost any cubic polynomial in n was accepted.

(2 marks) (b) **Briefly** explain how you found the function $T(n)$.

Solution: Note that by assigning for a Θ bound we did *not* require a precise count of the number of steps. Thus, the following simple answer would suffice:

- inner loop requires $\Theta(n)$ steps ($n/2$ iterations of a loop body that requires constant time)
- middle loop executes the inner loop n times, costing $\Theta(n^2)$ steps
- overall algorithm executes the middle loop n times, costing $\Theta(n^3)$ steps.

Another alternative is:

- the algorithm consists of three nested loops, each of which executes cn times for some constant $c > 0$.
- the inner loop body requires a constant number of steps
- therefore, total number of steps will be expressed by some polynomial in n of degree 3.
- therefore, runtime is in $\Theta(n^3)$

More complicated answer:

- inner loop requires $4(n/2) + 2 = 2n + 2$ steps (4 steps for loop body executed $n/2$ times, plus 1 for initializing k, plus 1 for final test on k before termination)
- middle loop requires $n((2n+2)+2)+2 = 2n^2+4n+2$ steps ($2n+2+2$ steps for the loop body executed n times, plus 2 for initialization and termination test)
- outer loop requires $n(2n^2 + 4n + 2) + 2 = 2n^3 + 4n^2 + 2n + 2$ steps ($2n^2 + 4n + 2 + 2$ steps for the loop body executed n times, plus 2 for initialization and termination test)
- overall algorithm requires cost of the outer loop plus 1 for initializing *count*, so
$$T(n) = 2n^3 + 4n^2 + 2n + 3$$

Comments:

- It is not sufficient to say that the algorithm runs in time $\Theta(n^3)$ because it has three for-loops. You also need to take into account the number of executions of each for-loop as a function of n and the number of steps executed in each loop body.

5. The following questions deal with the `Stack` abstract data type.

(4 marks) (a) Define the stack ADT as presented in class.

Solution: A stack is a collection of objects that supports the following operations:

- `create()`: Initialize a stack to be empty
- `isEmpty()`: Report whether the stack is empty
- `top()`: Report the top element (without changing it), and report an error if the stack is empty.
- `push(x)`: Add a new element to the top of the stack
- `pop()`: Remove the top element from the stack, and report an error if the stack is empty

Omitting `create` was also acceptable.

Comments: This questions was not answered very well. Descriptions of ADT's require:

- description of data (in this case "collection of objects")
- description of supported operations (names, preconditions, and postcondigions). In this case only `pop` and `top` have postconditions

(4 marks) (b) Give Java code that implements the `push` and `pop` operations efficiently when a singly linked list is used to represent a stack. You may assume the existence of the following private internal class:

```
private class StackNode {
  private Object data;
  private StackNode next;

  private StackNode(Object x, StackNode n)
    { data = x; next = n; }
}
```

You may also assume that the top of the stack is of type `StackNode`.

How to Implement the "push" Operation:

```
void push(Object x) {
  top = new StackNode(x,top);
}
```

How to Implement the “pop” Operation:

```
void pop() {
  if (isEmpty())
    throw new EmptyStackException();
  top = top.next;
}
```

Comments: This was reasonably well-done. One common error was to identify the top of the stack with the tail of a singly linked list, meaning that the entire list had to be traversed for `pop` in order to retrieve the new top. In a linked implementation, the top must be identified with the head of the list in order to ensure that all operations require constant time in the worst-case.

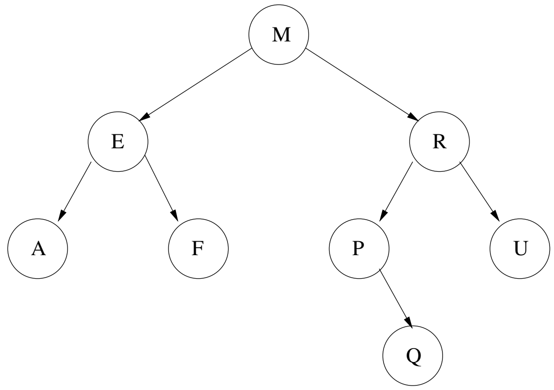
(2 marks) (c) Are there any advantages to using a **doubly linked list** to implement a stack? Why or why not?

Solution: There are no advantages to using a doubly linked list to implement a stack because

- the only required operations are adding to, removing from, or retrieving the data stored at the top of the stack
- identifying the top of the stack with the head of the list means that all these operations can be implemented efficiently (constant time) using just a singly-linked list, so the extra overhead of storing previous references is not required.

Comments: This was also reasonably well-done. The most common error, for those who realized that a DLL does *not* offer any advantages for implementing stacks, was forgetting to state why the previous pointers don’t help.

6. Consider the following binary search tree T:



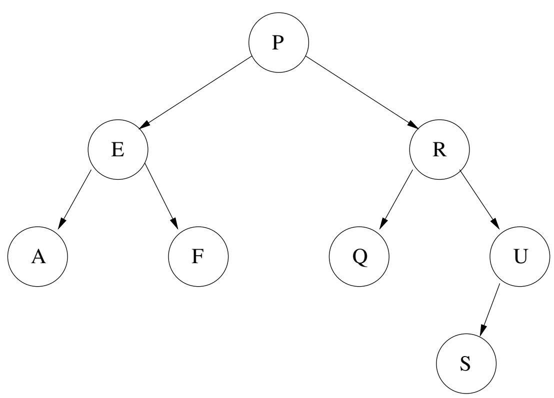
(3 marks) (a) Draw the binary search tree that would be obtained by

- deleting the node with key M, and
- inserting a node with key S

using the algorithms for insertion and deletion presented in class.

Note: although there are several different binary search trees that could possibly be produced by deleting M and inserting S, to get full credit for this question you must draw the **unique** search tree obtained using the specified algorithms.

Solution:



We also accepted solutions in which two trees are given, one of which is the tree resulting from deleting M from T and the other is the tree resulting from inserting S into T.

Comments: A number of people gave valid binary search trees containing the required keys that could not have been obtained using the algorithms described in class (as required).

(4 marks) (b) Give pseudocode for a **recursive** algorithm that computes the height of a binary tree T. Iterative algorithms will receive at most half credit for this question.

Solution:

```
int height(BST T) {
  if (T == null)
    return -1;
  else
    return 1 + max(height(T.left), height(T.right));
}
```

Comments: The most common error was incorrectly assigning an empty tree height 0 and a tree with one node height 1.