Midterm Examination Problem Sheet

TIME: 04/18/2013, 10:20–12:20

This is a open-book exam. You can use any printed materials as your reference during the exam. Any electronic devices are not allowed.
Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.
Both English and Chinese (if suited) are allowed for answering the questions. We do not accept any other languages.

There are 8 problems in the exam, each worth 25 points—the full credit is 200 points. Some problems come with a few sub-problems to help you get partial credits. For the 8 problems, 3 of them are marked with * and are supposedly simple; 3 of them are marked with ** and are supposedly regular; 2 of them are marked with *** and are supposedly difficult. The is one bonus problem, numbered 1126, with 10 points. The problems are roughly ordered by difficulty.

(1) (25%, *) To count the number of boys/girls in some department, student C wrote the following program:

```
1  #include <iostream>
2  using namespace std;
3  void increase(int count){
4    count++;
5  }
6
7  int main(){
8    char ID[1126][11];
9    int num_student;
10   int countBoy = 0;
11   int countGirl = 0;
12
13   //some code that initializes the ID array
14   //with something like "A123456789" per row
15   //also, num_student will be initialized
16
17   for(int i = 0; i < num_student; i++){
18     if (ID[i][1] == '1')
19       increase(countBoy);
20     else
21       increase(countGirl);
22   }
23   cout << countBoy << ' ' << countGirl << endl;
24   return 0;
25 }
```

(a) (10%) Assume that `num_student` is 1126 with 460 boys and 666 girls, what is the output of the program above?

(b) (15%) If your answer above is not `460 666`, explain how you can modify the program above with **only one line** to output `460 666`. If your answer above is `460 666`, explain to the TAs how the memory slots of the local variable `countBoy` in `main` and the local variable `count` in `increase` are changed during the first call to `increase(countBoy)`.

(2) (25%, *) Consider a complete binary tree with 11 nodes where each nodes are numbered from $1, 2, \cdots, 11$ that corresponds to its position in the array representation, with root being node number 1. That is, node i has a left child (if any) at $2i$ and a right child (if any) at $2i + 1$.

(a) (10%) Draw the binary tree with clear illustrations on the node numbers.

(b) (5%) Do a post-order traversal in the binary tree above and print out the node numbers visited.

(c) (5%) Do a in-order traversal in the binary tree above and print out the node numbers visited.

(d) (5%) Do a pre-order traversal in the binary tree above and print out the node numbers visited.

(3) (25%, *) You have been using a singly-linked list for your application, where each node is of the form

```
struct Node{
  int value; Node* next;
};
```

But now there is a new need in your application that requires you to find the previous nodes quickly. Therefore, you need to write a program to convert your singly-linked list to a doubly-linked one. Therefore, you define the following `DNode` structure to represent a node in your doubly-linked list:

```
struct DNode{
  int value; DNode* next; DNode* prev;
};
```

(a) (15%) Use any pseudo-code to describe an algorithm that takes a singly-linked list (i.e. a `Node* head` entry) and returns the new doubly-linked list (i.e. some `DNode*` entry) **while deleting the memory allocated by the original singly-linked list**. Your algorithm needs to be "on the fly." That is, it visits each node of the singly-linked list once and creates the corresponding node in the doubly-linked list immediately.

(b) (10%) Modify your pseudo-code above to describe an algorithm that takes the same input but returns the new doubly-linked list **in reverse order**. Your algorithm still needs to be "on the fly." (*Hint: You only need to change very few lines.*)

(4) (25%, **) Suppose you have two nonempty stacks S and T and a deque D. Use any pseudo-code to describe an algorithm that passes through the elements within S and T **once**, and uses D so that S contains all the elements of T below all of its original elements, with both sets of elements still in their original order.

(5) (25%, **) The financial team in your company finds an old calculator implemented with the LISP language, which is based on prefix expressions of binary operators of the form

```
( - ( * 7 ( + 3 5 ) ) 1 )
```

But the team wants to evaluate the expression quickly, and therefore they want to transform the expression to its post-fix form. For simplicity of this problem, let's assume that there are parentheses in the expressions (which is true for LISP but not necessary in general). Use any pseudo-code to describe an algorithm transforms a valid prefix expression to a corresponding post-fix one (without parentheses). For instance, the post-fix expression that corresponds to the one above is

```
7 3 5 + * 1 -
```

Your algorithm is only allowed to use **one stack** with size larger than the length of the prefix expression, and at most $O(1)$ of additional memory. (*Hint: Think about parentheses matching.*)

For the following two problems, you can only use this definition:
Let f, g be functions from non-negative real numbers to non-negative real numbers. We say $f(n) = O(g(n))$ iff there is a real constant $c > 0$ and an real constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.
Note that we use real numbers here for the simplicity of your proof below (so you don't need to have floors and ceilings running in your proof).

(6) (25%, **) Using the fact that $\log_e n \leq n - 1$ for all $n \geq 1$, prove the following statement: "For any $k > 0$, $\log_2 m = O(m^k)$."

(7) (25%, ***) Generalize your proof above to prove the following statement (or if you think the statement is wrong, disprove it): "Assume that $f(n) = O(g(n))$ and $h(n)$ is a function from non-negative real numbers to non-negative real numbers. In addition, assume that

- $h(n)$ is monotonically increasing: $h(n') \geq h(n)$ for all $n' \geq n$
- The inverse function $h^{-1}(m)$ of h exists for any $m > 0$

Then, $f(h(n)) = O(g(h(n)))$."
(*Hint: If this is true, let $f(n) = \log_2 n$, $g(n) = n$, and $h(n) = n^k$ and you can "easily" prove the statement in the previous problem with a few more lines.*)

(8) (25%, ***) For the binary max-heap with integer keys that is represented with an array and satisfies the heap properties:

- The nodes form a complete binary tree
- The node with the largest key in each sub-tree is at the root of the sub-tree (hence, the node with the overall largest key is at the root of the full tree.)

(a) (5%) Student C thought that this property is true: "The node with the 2^k-th largest key of the max-heap is within the first $2^{k+1} - 1$ elements of the array." Show a heap without this property and thus prove student C wrong.

(b) (10%) For a set of distinct integers, define the rank of an integer i be the total number of integers that are in the set and are not smaller than i. For instance, the largest number within the set is of rank 1, and the 2nd-largest number is of rank 2, and so on. Now, consider a set of all the elements within a binary max-heap that contains distinct integers. When the heap is of size $2^k - 1$, prove that for a node at the third position of the array (the right child of the root), it is at least of rank 2 and at most of rank $2^{k-1} + 1$.

(c) (10%) Now, consider a binary max-heap that contains distinct integers and is of an arbitrary size ≥ 3, use any pseudo code to describe an algorithm that computes the rank of the node at the third position of the array. Your algorithm should run in $O(r)$ time, where r is the rank of the node.

(1126) (Bonus 10%, ???) We discussed about the difference between selection sort and heap sort, where it appears that the only change comes from replacing the linear search of the unordered batch (that can be either an array, a linked list, or almost anything) to a heap represented with an array. Given that the time complexity of heap sort, $O(N \log N)$ for N elements, is supposedly better than the time complexity of selection sort, $O(N^2)$, why would anyone wants to use selection sort? Make some wild guesses and convince the TAs of some scenarios that is worth considering selection sort instead of heap sort. (*Hint: Imagine that your boss, who is not of CS major, asks you this question. The TAs will grade qualitatively on whether you can convince your boss in a reasonable manner.*)