YORK UNIVERSITÉ UNIVERSITY

EECS 2011: Fundamentals of Data Structures, Summer 2017
Midterm Test

Last Name(s): solutions

Given Name(s): _____

Student Number: _____

eecs account: _____

Instructions

1. Read the questions carefully.
2. The test has 6 questions and 8 pages (including this one).
3. Time Allowed: 75 minutes.
4. No aids are permitted.
5. Answer each question in the space provided.
6. You can use the last page as a scrap page. Nothing written on it will be marked.
7. If a question is not clear, state your assumptions and proceed with an answer.
8. Do *not* use red ink. Answers written in red do *not* count.
9. Write legibly. Unreadable answers do *not* count.
10. This paper must be returned with your answers.

Question	Maximum	Mark
1	10	
2	10	
3	9	
4	6	
5	6	
6	9	
Total	50	

1. [10 points] True or false (circle the answer on the right, 1 point for correct answers, −0.5 points for wrong answers, 0 points minimum)

a) If $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$ (true) false

b) $f(n) = 5n^{2.5} + 10n\log(n^{2.5})$ is $\Omega(n^{2.5})$ (true) false

c) $f(n) = n^2\log(n) + n$ is $O(n^3\log n)$ (true) false

d) $O(n^2\log n)$ is the best asymptotic 'big-O' characterization of the following function:
$f(n) = 50 + 10n^2\log n + 2n^2 + n^3$ true (false)

e) Algorithms of complexity $\Theta(n^n)$ are generally infeasible (true) false

f) The average running time of search in an unsorted array is $O(\log n)$ true (false)

g) Stack is a FIFO (first-in-first-out) data structure true (false)

h) `ArrayList` implementation of `List` works better than `LinkedList` for operations such as insertion at the beginning or at the end true (false)

i) The worst-case asymptotic performance of *quicksort* is better than that of *bubble sort* true (false)

j) One can merge two linked lists in a constant amount of time (true) false

2. [10 points] Write an implementation of a method `removeNulls()` that removes all *null* nodes from a doubly linked list specified below.

   Note: the amount of space provided does not imply the amount of code expected. Make sure you handle boundary cases properly.

```
public class DoublyLinkedList <E>{
      int size = 0; //number of elements in the list
      Node<E> first; //pointer to the first element of the list
      Node<E> last; //pointer to the last element of the list

      private static class Node<E> {
            E item; Node<E> next; Node<E> prev;

            Node(Node<E> prev, E element, Node<E> next) {
                  this.item = element; this.next = next; this.prev = prev;
            }
      }
      public void removeNulls() {
            //to implement
```

2pt list walk
2pt == null
 	
2pt check for head/tail node
 	
2pt pointer setting
 	
1pt x.prev and x.next nulling
 	
1pt size--
	`for (Node<E> x = first; x != null; x = x.next) {`
	`  if (x.item == null) {`
	`  if (x.prev == null) {`
	`  first = x.next;`
	`  } else {`
	`  x.prev.next = x.next;`
	`  x.prev = null;`
	`  }`
	`  if (x.next == null) {`
	`  last = x.prev;`
	`  } else {`
	`  x.next.prev = x.prev;`
	`  x.next = null;`
	`  }`
	`  size--;`
	`  }`
	`}`

```
      }
}
```

3. [9 points] What is the big-Θ running time of the following code fragments? Briefly justify your answer. No need for a rigorous formal proof.

a)
```
int sum = 0;
for (int i = 1; i < N; i *= 4)
   for(int j = 0; j < N; j++)
      sum++;
```

(N + N + N + N...) $\log_4 N$ times = $\Theta(N \log N)$

b)
```
int sum = 0;
for (int i = N; i > 0; i /= 4)
   for (int j = 0; j < i; j++)
      sum++;
```

N + N/4 + N/16 + ... + 1 = N (1 + 1/4 + 1/16 + ...) = $\Theta(N)$

c)
```
int sum = 0;
for (int i = 0; i < N; i += 4)
   for (int j = 0; j < i; j++)
      sum++;
```

|

xxxx

xxxxxxxx

xxxxxxxxxxxx

xxxxxxxxxxxxxxxx

....

- triangular sum => $\Theta(N^2)$

4. [6 points] Recursion.

a) Indicate the (final) return values for each call of the method `beaver` below.

```
public static int beaver(int n) {
    if (n < 0) { return -beaver( -n ); }
    else if (n == 0) { return 0; }
    else { return beaver( n / 2 ) * 2 + 2 - (n % 2); }
}
```

(1 pt each)

beaver (0) 0

beaver (5) 9

beaver (-6) -8

beaver (2) 4

b) The following method is correct syntactically; however, it will not function properly. Why? How could you possibly fix it?

```
public static int moose(int n) {
    if (( n % 5) == 0 ) { n--; }
    if (n == 0) { return 0; }
    else { return (1 + moose (n - 2)); }
}
```

It will not terminate for some values of *n*, for example, for *n* = 1.

Fix: change the base case to include <= instead of == or change the recursive calls.

5. [6 points] Given a stack, an integer k and a linked list of integers, how do you reverse the order of the first k elements of the list, leaving the other elements in the same relative order? For example, if $k = 4$, and the list that has the following elements [**1 2 3 4** 5 6 7 8 9] the output should be [**4 3 2 1** 5 6 7 8 9]. Provide a pseudo code of a method (or a Java code, or an informal description of the algorithm) that accomplishes this task. The running time and the space complexity of your algorithm should be $O(n)$. For the list, your can only remove the items from the front or from the back, or add them to the front or to the back; i.e., no in-the-middle operations.

Move the first k elements of the list to the stack.

Move the stack elements to the back of the list.

Move the other (list size – k) elements from the front of the list to the back of the list.

(partial marks possible, also, there could be many variations of the algorithm)

–4 points for using in-the-middle operations (using indices to access elements in the middle)

–2 ponts for using another list

–1 point for using two stacks

6. [9 points] Let A be an unsorted array of numbers. Provide a pseudo code of a method (or a Java code, or an informal description of the algorithm) that takes such an array as an input and outputs an array with all positive numbers to the right of all the negative numbers. E.g. [3 4 −2 −8 −1 −4 6 9] should produce [−2 −8 −1 −4 3 4 6 9] or [−8 −1 −2 −4 3 4 6 9] or [−1 −4 −2 −8 9 3 4 6], etc. The running time of your algorithm should be $O(n)$ and it should only require a constant amount of space.

Using two indices, move from the ends of the array inwards until the left current element is positive and the right current element is negative;

If the indices crossed, done. Otherwise, swap the left and right elements and continue moving inwards from where you left off.

Each element is visited once => linear time.

Need only 2 variables for indices and one for swapping => constant space

(partial marks are possible, also, there could be other variations of the algorithm)

–3 points for non-constant space (e.g., an additional array)

–5 points for non-linear running time

SCRAP PAGE. WILL NOT BE MARKED!