YORK UNIVERSITÉ UNIVERSITY

MIDTERM TEST

CSE 2011 – Fundamentals of Data Structures

Summer 2013

First Name:________________________Last Name:________________________________

Student ID:____

CSE Number:____

TIME LIMIT: 80 minutes

- This is a closed-book test. No books, no papers, no calculators are allowed.
- Use the back of any page for scrap work.
- Nothing written on the back of the pages will be marked.
- Extra space for answers can be found on the last page. However, the provided space is generally sufficient.
- Programming problems will be marked based on both correctness and efficiency. Both Java code and pseudo-code are accepted.
- Do not leave during the last 15 minutes; remain seated until all tests have been collected.

Question	Value	
1	4	
2	4	
3	3	
4	6	
5	4	
6	6	
Total	27	

1- Growth Rate (4 pts)

Part I: For each of the functions f(N) given below, indicate the **tightest** bound possible (Big-O).

a) $f(N) = (N + N + N + N)^2$ _______**$O(N^2)$**_______ **0.5 pt**

b) $f(N) = (N/5) \log (N^4) + 66N$ _______**$O(N \log N)$**____ **0.5 pt**

c) $f(N) = N \log (100^3) + \sqrt{N}$ _______**$O(N)$**_________ **0.5 pt**

d) $f(N) = N.(N^2 \log N + N^2)$ _______**$O(N^3 \log N)$**____ **0.5 pt**

Part II: Do the following two functions grow at the same rate? If not, which function grows faster? **Explain** your answer (no mark will be given without an explanation).

2.0 pt

$N^{3.3}$ **$N^3.(\log N)^{17}$**

$f(N) = N^{3.3}$ $g(N) = N^3.(\log N)^{17}$

f(N) grows faster than g(N).

if we can prove that $\lim_{n\to\infty} \frac{f(N)}{g(N)} = \infty$, then f(N) grows faster than g(N).

The **only** way to prove it is using the L' Hôpital's rule and the following formula **three** time. In other words, **derivative** of the functions should be applied **three** time.

$$\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$$

2- Running Time Calculations (4 pts)

Describe the **tightest worst case** running time of the following java style pseudocode functions in Big-Oh notation in terms of **the variable n**. No proof/description is required.

I. **1.0 pt**

```
void test1(int n, int x, int y) {
     for (int i = 0; i < n; ++i) {
          if (x > y) {
               for (int j = 0; j < n; ++j)
                    System.out.println("j = " + j);

               for (int k = 0; k < n * n; ++k)
                    System.out.println("k = " + k);
          } else
               System.out.println("i = " + i);
     }
}
```

Runtime:

$O(n^3)$

II. **1.0 pt**

```
void test2(int n, int m) {
     if (m > n)
          return;
     System.out.println("m = " + m);
     test2(n, m+2);
}
```

Runtime:

$O(n)$

III. **1.0 pt**

```
void test3(int n) {
     if (n <= 0)
          return;
     System.out.println("n = " + n);
     test3(n/2);
}
```

Runtime:

$O(\log n)$

IV. **1.0 pt**

```
void test4(int n) {
     for (int i = 0; i < n; ++i) {
          j = 0;
          while (j < n) {
               System.out.println("j = " + j);
               j++;
          }
     }
}
```

Runtime:

$O(n^2)$

3- Solving a Recurrence Relation (3 pts) **3.0 pt**

Solve the following recurrence by finding a Big-Oh bound for T(N), given that T(1) = 1. The calculation **must** be shown for full marks.

$T(N) = T(N-2) + 3.N + 4$

$T(N) = T(N-2) + O(N)$

$T(N-2) = T(N-4) + O(N)$

$T(N-4) = T(N-6) + O(N)$

.

.

$T(1)$

we have **N/2** levels until we hit T(1). In each level, we do O(N) work. Therefore, total run time is N/2.O(N) which is $\mathbf{O(N^2)}$.

If someone provides more details, like the sum equation, it is even better but something like the above argument is enough.

4- Recursion (6 pts)

Part I
Write a **recursive** method called *precur()* that receives an integer *n* and returns 2^n. It should have **linear** run time in terms of *n*. Calculate the run time in terms of a recursive equation and drive the **Big-Oh** notation.

Algorithm 2.0 pt

```
in precure(int n){

if(n==1)
        return 2;
else
        return precure(n-1)*2;  // This is also correct: return precure( ┌n/2┐ )* precure( └n/2┘ )
                                // ┌ ┐ and └ ┘ are ceiling and floor funcitons
}
```

Runtime 1.0 pt

Runtime: T(n) = T(n-1) + O(1).
total runtime is O(N).

Part II

Write a **recursive** method called *precur()* that receives an integer *n* and returns 2^n. It should have **<u>logarithmic</u>** run time in terms of *n*. Calculate the run time in terms of a recursive equation and drive the **Big-Oh** notation.

Algorithm 2.0 pt

```
in precure(int n){

if(n==1){
        return 2;
}
else{
        if(n is EVEN) {
                int temp = precure(n/2);  // it is the floor
                return temp*temp;
        }else{
                int temp = precure(n/2);  //it is the floor
                return temp*temp*2;
        {
}
}
```

Runtime 1.0 pt

Runtime: T(n) = T(n/2) + O(1).
total runtime is O(log N).

Please note, any other solutions, like precure(n/2)* precure(n/2) will get the mark of **ZERO** because it is not **logarithmic**;

5- Trees (4 pts)

Part I.

What is the minimum and maximum number of nodes at depth **d** in a <u>proper</u> binary tree ? Be sure to list the nodes at depth d. **Do not** include nodes at depth d-1 or d+1 or other depths.

Minimum = **2** (if height ≥ 1, otherwise it is 1) **1.0 pt** Maximum = $\underline{\mathbf{2^d}}$ **1.0 pt**

Part II.
Give traversals of the tree shown below:

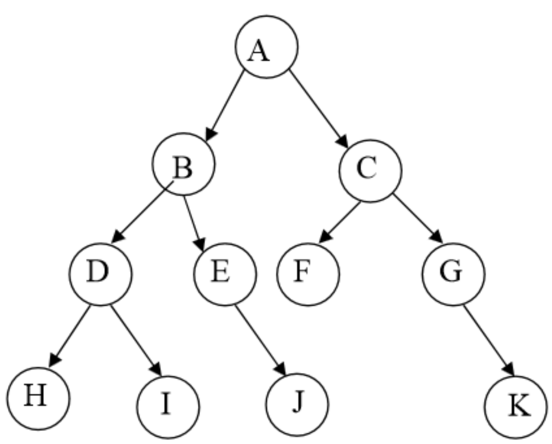
Preorder: A B D H I E J C F G K **0.5 pt**

Postorder: H I D J E B F K G C A **0.5 pt**

Inorder: H D I B E J A F C G K **0.5 pt**

Part III.

What is the height of the tree shown below: ____**3**____ **0.5 pt**



6- Stack and Deque (6 pts)

Suppose a linked list is implemented using **only** the following methods:

insertBeginning(E e)	inserts e at the beginning of the list
remove(Position<E> p)	removes p and returns the respective element
returnFirst()	returns the first Position
returnLast()	returns the last Position
size()	returns the size of the sequence

Part I: Implement the following methods of the **Stack** ADT using only the above methods: *push(), pop(), top()*

pop() = remove(returnFirst()) **0.75 pt**

top() = returnFirst() **0.5 pt**

push(p) = insertBeginning (p) **0.5 pt**

Part II: Implement the following methods of the **Deque** ADT using only the above methods:
addFirst(E e), removeFirst(), addLast(E e), removeLast()

addFirst(p) = insertBeginning(p) **0.75 pt**

---

removeFirst() = remove(returnFirst()) **0.75 pt**

---

addLast(p) = **2.0 pt**

```
{
        insertBeginning(p)
        for (int i=1; i<size(); i++)
                insertBeginning(remove(returnLast()));
}
```

---

removeLast() = remove(returnLast()) **0.75 pt**

THIS IS THE LAST PAGE