UNIVERSITÉ YORK UNIVERSITY

Department of Computer Science and Engineering

EECS 2011: Fundamentals of Data Structures (Winter 2017)

Final Examination

April 9, 2017

Instructions:

- Examination time: 180 min.
- Print your name and CS student number in the space provided below.
- This examination is closed book and closed notes. No calculators or other computing devices may be used.
- There are 8 questions. The points for each question are given in square brackets, next to the question title. The overall maximum score is 100.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.

FIRST NAME: ______________________________

LAST NAME: ______________________________

STUDENT #: ______________________________

Question	Points
1	/ 10
2	/ 15
3	/ 11
4	/ 9
5	/ 15
6	/ 15
7	/ 15
8	/ 10
Total	/ 100

1. **True/False Choice** [10 points]

Indicate whether each of the following statements is true or false:

	Question	**Answer**
1.	A self-organizing list consists of four nodes with, with the following access probability: 0.4 (1st node), 0.3 (2nd node), 0.2 (3rd node) and 0.1 (4th node). The average number of steps per single access is 2.	[✓ True] False
2.	A self-organizing list should implement 'move to front' heuristics if we want the list to be able to quickly adapt to a changing access pattern.	[✓ True] False
3.	The space requirement of a vector-based implementation of an ill-balanced binary tree is $O(n^2)$.	True [✓ False]
4.	Consider the order in which the **leaves** of a proper binary tree are visited by preorder, postorder and inorder traversal. In all 3 cases the leaves are visited in different order.	True [✓ False]
5.	Insertion in an AVL tree is "commutative". That is, inserting x and then y into an AVL trees results in exactly the same tree as inserting y and then x.	True [✓ False]
6.	In an AVL tree, after inserting a key, we have to do at most one (single or double) rotation to maintain the balance property.	[✓ True] False
7.	In a m-ary heap, if the heap stores N items, then an insert() is more costly than removeMin().	True [✓ False]
8.	A binary search tree with n elements can be converted into a heap in O(n) time.	[✓ True] False
9.	Open addressing works well even when there are more keys than can be stored directly in the hash table.	True [✓ False]
10.	Suppose that instead of a linked list, each bucket of a hash table 'with separate chaining' is implemented as an AVL tree. In that case the worst case running time to insert n keys into the table is O(log(n)).	True [✓ False]

2. Basics Potpourri [15 points]

(a) Big-O [3 points]
For each of the following big-O statements, circle the right answer – true or false.

$\log(3^n)$ is $O(\log(2^n))$.	**True** False
$\log_3(n)$ is $O(\log_2(n))$.	**True** False
$\log(n^2)$ is $O(\log(n^3))$.	**True** False

(b) RT analysis [3 points]
What is the execution running time of **add(8,n)** (see the code below), as a function of n ($n \geq 0$). (Express the running time using big-O notation; you do not need to provide an exact analysis.)

```
int add(a,b) {
     if (a==0)  return b;
     else if (a<0)  return add(a+1, b-1);
     else return  add(a-1, b+1) }
```

Solution:

O(1)

(c) RT analysis [3 points]
What is the running time of the following algorithm, as a function of n. (Express the running time using big-θ notation; you do not need to provide an exact analysis.) Assume $n \geq 1$.

```
int i=1;
int sum=0;
while (i<n²) {
     for (int j=n²; j<1; j=j/2) {
          sum++;
     }
     i += 2;
}
```

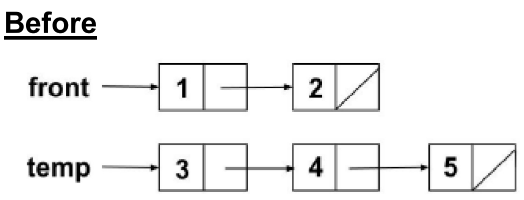
Solution:

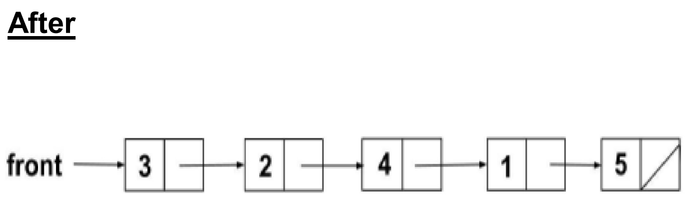
$O(n^2)$

(d) Linked-list coding [6 points]

Write the code that will turn the below "before" picture into "after" picture by modifying links between the nodes shown. There may be more than one way to write the code, but you are NOT allowed to change any existing node's data field value. You also should not create new ListNode objects, but you may create a ListNode variable(s) to refer to any existing node if you like.

To help maximize partial credit in case you make mistake, you need to include (brief) comments with your code that describes the links you are trying to change. Your code should consist of no more than 7 to 8 lines of code.





Note: "front" and "temp" in the "before" picture are reference variables pointing to the first node of the respective lists. You can also assume that you are using the ListNode class as defined in class:

```
public class ListNode {
     public int data;          // data stored in this node
     public ListNode next;     // a link to the next node in the list

     public ListNode()   { … }
     public ListNode(int data)   { … }
     public ListNode(int data, ListNode next)   { … }

     public int getElement() { … }
     public ListNode getNext() { … }
     public void setElement(int newElement) { … }
     public void setNext(ListNode newNext) { … }
}
```

Solution:

```
ListNode current = front.getNext();
front.setNext((temp.getNext()).getNext());    // 1 -> 5
(temp.getNext()).setNext(front);              // 4 -> 1
current.setNext(temp.getNext());              // 2 -> 4
temp.setNext(current);                        // 3 -> 2
front = temp;
```

3. Implementation [11 points]

(a) Recursive tracing [6 points]
What does the following code print out for each of the four given inputs?

```
public static void mystery(int x) {
     if (x>0) {
          mystery(x/5);
          System.out.print(x%5); }
}
```

input	output
2	2
5	10
36	121

(b) Queues and Stacks [5 points]
Palindromes are words or phrases that appear the same whether read backwards or forwards, e.g. “eye”, “madam”, “never odd or even”, etc.

Complete the Boolean method, **isPalindromeQS**(char[] text), which uses one queue and one stack, and returns true if the text contained in the given character array is a palindrome. You may assume the array contains no space characters, and all characters are lower case.

```
boolean isPalindromeQS(char[] text) {

     Stack s = new Stack();
     Queue q = new Queue();

     for (int i = 0; i < text.length; ++i) {
          s.push(new Character(text[i]));
          q.enqueue(new Character(text[i]));
     }

     while (!s.isEmpty()) {
          if (!q.getFront().equals(s.peek()))
               return false;
          s.pop();
          q.dequeue();
     }

     return true;
}
```

4. Priority Queue [9 points]

Consider the following algorithm for sorting an array (a) with the help of a priority queue (q).

```
q = new priority queue
for every element x in a
  q.insert(x)
i = 0
while q is not empty
  a[i] = q.findMinimum()
  q.deleteMinimum()
  i = i+1
```

Assuming that n is the length of the input array, what is the big-O complexity of this algorithm, if the priority queue is implemented using of the following? (Justify your answer!)

a) an unsorted array

Solution:

$O(n^2)$. Insert takes (amortised) O(1) time so the for-loop takes O(n) time. But findMinimum and deleteMinimum take O(n) time so the while-loop takes $O(n^2)$ time.

b) a binary heap

Solution:

O(n log n). Insert takes (amortised) O(log n) so the for-loop takes O(n log n) time. DeleteMinimum also takes O(log n) so the while-loop takes O(n log n) time.

c) a sorted array

Solution:

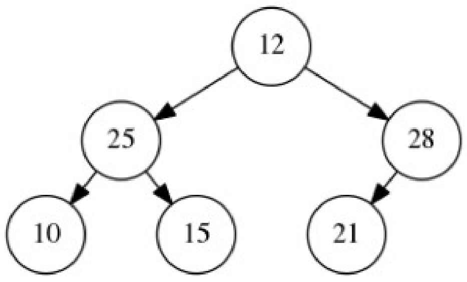
$O(n^2)$. Insert takes O(n) time so the for-loop takes $O(n^2)$ time.

Note, even if you assume that the search for the location of a new element is done using binary search (runs in log(n)), once this location is found a new 'spot' for the new element has to be made available (i.e., subsequent element have to be shifted) – hence the overall time of insert is O(n) in the worst case.

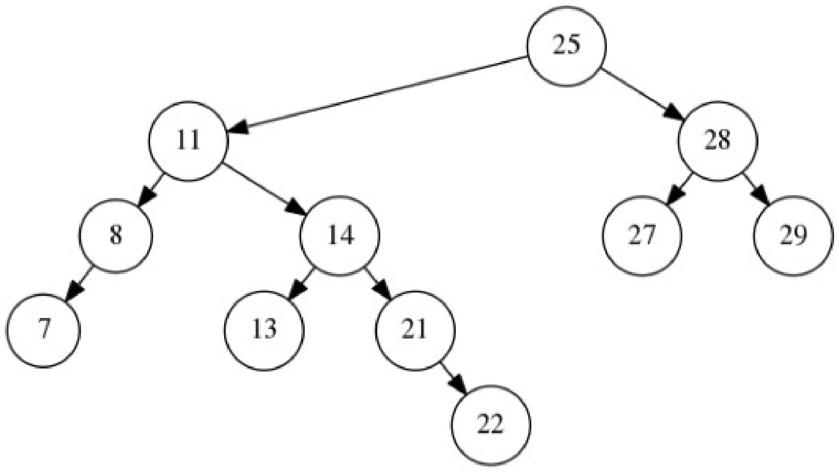
5. AVL Tree [15 points]

5.1 [6 points]
Take a look at the following two binary trees. Which of them is an AVL tree? Justify your answer.

Tree A



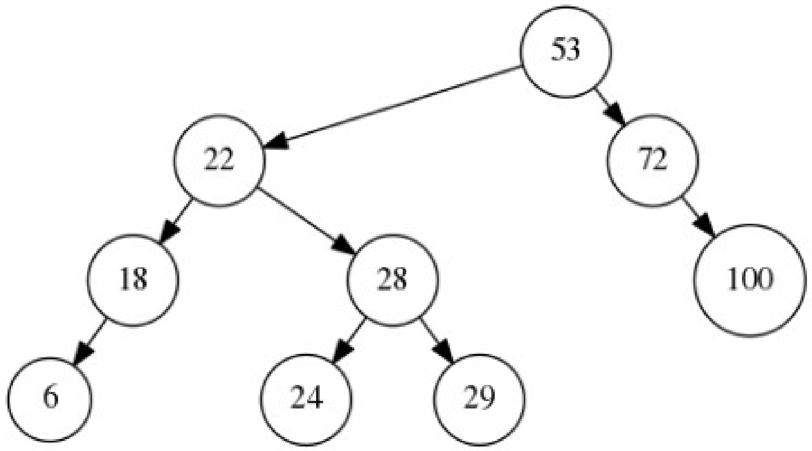
Tree B



Solution:

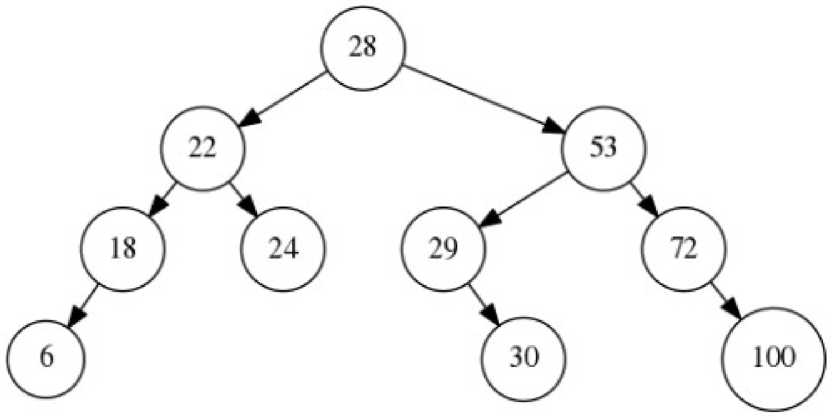
A is not a binary search tree (25 > 12) and B is not balanced enough (height of left child of root = 4, height of right child of root = 2)

5.2 [9 points]
Insert 30 into the below tree using the AVL insertion algorithm. Write down the final AVL tree.



Solution:

After inserting 30 using BST insert, the tree is unbalanced (root's left child height = 4, right child height = 2). It's a left-right tree so a double rotation fixes it:



6. **Heaps** [15 points]

The vector shown below represents the underlying array of a heap.

0	1	2	3	4	5	6	7	8	9
	17	**28**	**34**	**32**	**97**	**36**	**52**	**35**	

Suppose the following operations are performed (in this order!) on the given heap:

RemoveMin(); RemoveMin(); Insert(38); Insert(15).

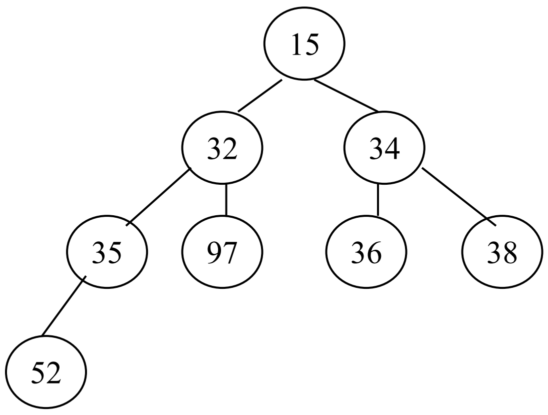
How will the heap, i.e. the underlying array, look like now? (For full credit, provide the content of the underlying array after each of the 4 operations.)

Solution:

After first min removal:	28 32 34 35 97 36 52
After second min removal:	32 35 34 52 97 36
After inserting 38:	32 35 34 52 97 36 38

After inserting 15:

0	1	2	3	4	5	6	7	8	9
	15	**32**	**34**	**35**	**97**	**36**	**38**	**52**	



7. **Hash Table** [15 points]

7.1 Linear Hashing [6 points]

Consider the following hash table implemented using linear probing, where the hash function is defined as

H(x) = x mod 10

0	1	2	3	4	5	6	7	8	9
10	1	11		14	5	XXX	7	18	

The value that was previously at index 6 has been deleted, which is represented by the XXX in the hash table. Which of the following values might have been stored there, before it was deleted?

26 2 17 24 6 13

There may be several correct answers, and you should write down (i.e., circle) all of those that apply. Additionally, provide a brief justification of your answer.

Solution:

Because of probing, an element is either stored at the index corresponding to its hash or a greater index. So, the deleted value's hash could have been any number ending with 4, 5, or 6. Out of the given numbers, the ones that satisfy this requirement are:

26, 24, 6

7.2 Double Hashing [6 points]

Consider a hash table of size 7 with hash function

$$H_1(k) = k \bmod 7$$

Draw the table that results after inserting, in the given order, the following values:

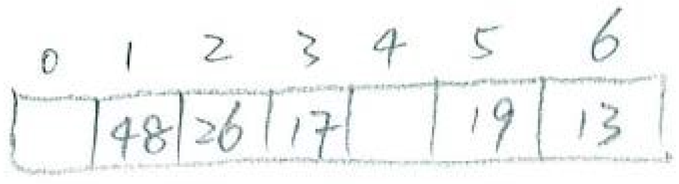
19, 26, 13, 48, 17

Collisions are handled by double hashing using a second hash function

$$H'(k) = 5 - (k \bmod 5)$$

Solution:

19 – hashes to (and gets stored) in 5
26 – hashes to 5 (which is occupied), and then second-hashes to (26 + 5 – 26 mod 5) mod 7 = 2; since 2 is empty, 26 gets stored there
13 – hashes to (and gets stored) in 6
48 – hashes to 6 (which is occupied), and then second-hashes to (48 + 5 – 48 mod 5) mod 7 = 1; since 1 is empty, 48 gets stored there
17 – hashes to (and gets stored) in 3



0	1	2	3	4	5	6
	48	26	17		19	13

7.3 Hashing vs. BST [3 points]

Hash tables typically have better performance, when it comes to searching for a particular element, than balanced binary search trees. Even so, both are widely used in practice. One reason is that hash table does not support all the operations that BST does.

Give examples of (at least) two operations which can be efficiently implemented for a binary search tree but not for a hash table.

Solution:

Example 1 – searching for the minimum element takes O(log(n) in BST, and O(n) in hash table.
Example 2 – printing all elements in increasing order takes O(n) in BST, and O(n^2) in hash table.

8. Algorithm Design [10 points]

Design an algorithm that takes:

- an array containing n distinct natural numbers
- a number $k \leq n$

and calculates the sum of the k largest numbers in the array.

For example, if the array is {3, 7, 5, 12, 6} and k=3, then the algorithm should return 25=(12+7+6).

You may freely use standard data structures and algorithm from the course in your solution.

Write down your algorithm as pseudocode – you do not need to write fully detailed Java code.

Your algorithm should run in O(n·log(k)) time!

Solution:

```
h = new heap
for each x in array
        h.insert(x)
        if h.size() > k  then  h.deleteMin()
sum = 0
while h not empty do
        sum = sum + h.min()
        h.deleteMin()
```

The idea is to loop through the array and keep adding these elements to heap h while ensuring that h is no larger than k, and those k elements are the greatest elements of the array seen so far.
Whenever h contains (k+1) elements, we remove the smallest one so that it only contains the k greatest elements.
Afterwards we sum up and dismantle the whole heap.

NOTE – regarding the grading of Q.8:

Any solution with RT of O(nlog(n)) received 2/10 points.

Any solution with RT of O(klog(n)), but no discussion/comment on the algorithm's RT received 6/10 points.

Any solution with RT of O(klog(n)), but no specific comment on how such a solution relates to the required O(nlog(k)) running time – whether it is better or worse – received 8/10 points.

Any solution with RT of O(klog(n)), and proper justification (explaining that klog(n) is O(nlog(k)), received 10/10 points.

Any solution with the RT of O(nlog(k)) – which was actually requested – obviously received 10/10.