1	2	3	4	5	6	7	Σ
10	20	10	10	15	15	20	100

Student number	2	0						
Name								

Final Exam — Data Structures (CS206C)

December 17, 2015, 9:00–11:45

Instructions:

- This booklet has **three pages** with **7 problems** in total. Check that you have all!
- The space provided should be sufficient for your answer. If you need scratch space, use the back side. If you need more space for your answer, you can also use the back side (but *indicate clearly on the front side* that your answer continues on the back).
- This is a **closed book** exam. You are not allowed to consult any book or notes.
- The questions have to be answered in **English**. Write clearly!
- To ensure a quiet exam environment, we will **not answer questions** during the exam. If you think there is a mistake in the question, explain so, and use common sense to answer the question.
- Before you start: Write your name and student number (one digit per square!) on **all pages** of this exam booklet (−5 points for missing names or unreadable numbers).
- Relax. Breathe. This is just an easy, silly, and stupid final exam.

Problem 1: (10 pts) For each of the following terms, indicate whether it is the name of an *abstract data type* (ADT) or the name of a *data structure* (DS). (1 point for correct answer, −1 point for wrong answer).

• Doubly-linked list	ADT — DS
• Queue	ADT — DS
• Stack	ADT — DS
• Binary search tree	ADT — DS
• Priority Queue	ADT — DS
• Binary Heap	ADT — DS
• Hash table	ADT — DS
• AVL-tree	ADT — DS
• Singly-linked list	ADT — DS
• Map	ADT — DS
• Sorted singly-linked list	ADT — DS
• Set	ADT — DS

Problem 2: (20 pts) For each of the following statements, say whether they are *true* or *false* (2 points for correct answer, −2 points for wrong answer).

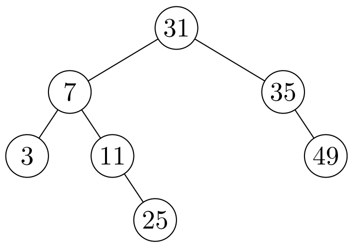
• *Merge-Sort* is an in-place sorting algorithm.	True — False
• A hash table of size 5,000 using chaining can efficiently store 8,000 elements.	True — False
• *Heap-Sort* is an in-place sorting algorithm.	True — False
• A hash table of size 18,000 using linear probing can efficiently store 17,990 elements.	True — False
• *Merge Sort* runs in $O(n \log n)$ time in the worst case on a sequence of size n.	True — False
• The expected search time in a hash table with chaining is $O(1/\lambda)$.	True — False
• A stack can be implemented effiently by one singly-linked list.	True — False
• A priority queue can be implemented efficiently by an AVL-tree.	True — False
• Hash tables can efficiently find the smallest key greater than a given search key.	True — False
• 3^n is $O(2^n)$.	True — False

Problem 3: (10 pts) Consider a search tree T storing 70 keys. What is the *worst-case height* of T in the following cases (that is, the largest possible height)? Give the exact answer (not Big-Oh notation).

- T is a binary search tree:
- T is an AVL tree:

Student number	2	0						
Name								

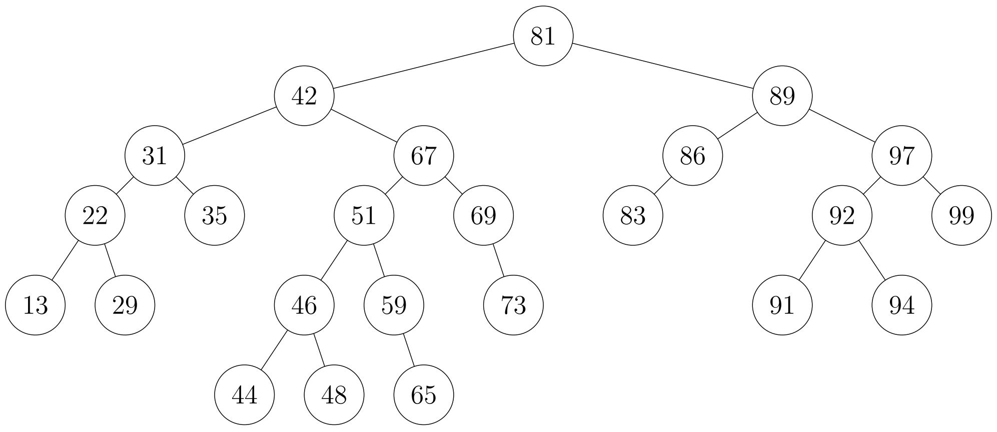
Problem 3: (10 pts) Write the (a) pre-order sequence, (b) in-order sequence, and (c) post-order sequence for the binary search tree below. Then write in (d), (e), (f) three *different* sequences of insertions that would create this tree (using insertions without rebalancing).



(a)
(b)
(c)

(d)
(e)
(f)

Problem 5: (15 pts) You are given the following AVL-tree T. (a) Insert the key 66 into this tree and draw the resulting tree. (b) Start again from the tree T (*not from the tree you got in part (a)*!), delete the key 81 and draw the resulting AVL-tree. For each operation, indicate the number of single rotations and double rotations that were performed.



Student number	2	0						
Name								

Problem 6: (15 pts) We are implementing a class `SinglyLinkedList` to store a singly-linked list. The nodes are objects of the following node class:

```
class Node:
  def __init__(self, el, next):
    self.el = el
    self.next = next
```

Write the method `remove_all` for the `SinglyLinkedList` class. It removes all nodes from the list whose element `el` is equal to `x`. The front of the list is in `self.front`. Handle all special cases correctly.

```
  def remove_all(self, x):
```

Problem 7: (20 pts) Consider a binary search tree that stores a set of strings. The nodes of the tree are objects of the following class:

```
class Node():
  def __init__(self, el, left, right):
    self.el = el
    self.left = left
    self.right = right
```

Write a *recursive* function `upper_neighbor(n, x)` that returns the smallest string y that is larger or equal to x in the subtree with root n. If there is no such string y, the function must return `None`.

```
def upper_neighbor(n, x):
```