Question 1. [0 MARK]

Write and then sign the declaration below. When you take a photo of it, place your **picture ID beside** and hide any sensitive information. Note you may have multiple questions on one page and in one photo.

> "***I have not consulted any resources including but not limited to classmates, tutors, textbooks, webpages, cheatsheets.***"

Signature: ______________________ Student Number: ______________________

For each of the following questions, state the letter corresponding to the **best correct** answer.

Question 2. [2 MARKS]

Suppose we insert n keys into a regular Binary Search Tree. The i^{th} key is computed as $i * (-1)^i$ for $i \in \{1 \dots n\}$. For example, the first few keys inserted are -1, 2, -3, etc. Then the height of the resulting tree is:

(a) $\Theta(1)$
(b) $\Theta(\log n)$
(c) $\Theta(n)$ ✓
(d) $\Theta(n \log n)$
(e) None of the above

Question 3. [2 MARKS]

In every AVL tree with at least 500 distinct keys, the second largest must be stored at:

(a) The root
(b) A leaf
(c) A non-leaf node other than the root
(d) A node that is either a leaf or the parent of a leaf✓
(e) Could be any node in the tree

Question 4. [4 MARKS]

Let $g(n) = \frac{n}{13} - \log n^2 + 10\pi$. State both the tightest *upper* bound to $g(n)$ and the tightest *lower* bound to $g(n)$ that you find in the options listed below.

(a) $O(1)$	(e) $\Omega(1)$
(b) $O(\log n)$	(f) $\Omega(\log n)$
✓(c) $O(n \log n)$	✓(g) $\Omega(\sqrt{n})$
(d) $O(n\sqrt{n})$	(h) $\Omega(n \log n)$

Question 5. [5 MARKS]
Prove or disprove

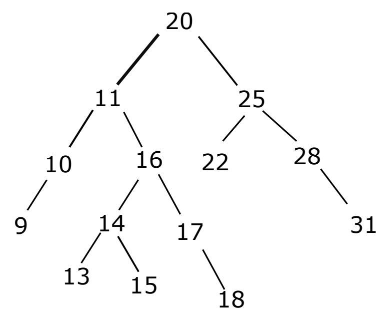
$$2^{\sqrt{\log_2 n}} \in O(n^{\frac{1}{3}})$$

.

Prove. Taking logarithms of both sides gives $\sqrt{\log_2 n}$ and $\frac{1}{3}\log_2 n$. Notice that the first term is $(\log_2 n)^{\frac{1}{2}} \leq \frac{1}{3}\log_2 n$ once $n \geq 9$.

Question 6. [4 MARKS]
Insert 17 and then 15 to the following AVL tree. Draw only your final tree.

SOlution.



Question 7. [8 MARKS]
You have collected statistics for an instructor on grades. For each grade you keep track of the number of students with that grade. For example, if in a class, there are 5 students with the grade of 76, you would store (76, 5). The instructor would like to be able to update this information as new assignments are assigned and as students drop the class. This means the usual operations of `insert(grade)` and `delete(grade)` need to be implemented.

`Insert(g)`: Insert or update grade `g`. If `g` already exists, then add 1 to the number of occurrences. If `g` does not exist add it to the data structure with occurrence of 1.

`Delete(g)`: Delete or update grade `g`. If `g` already exists with number of occurrences greater than 1, then subtract 1 from the number of occurrences. If `g` exists and has occurrence 1, remove `g` from the data structure.

In addition, the instructor would like to perform the following query:

`Max_grade(k)`: Return the maximum grade `g` such that `g` has occurred at least `k` times. For example, if the grade pairs are (50, 5), (75, 3), (80, 2), (90, 3), then `Max_grade(3)` would return 90 but `Max_grade(4)` would return 50.

Explain the data structure you would use to implement the three operations `Insert(g)`, `Delete(g)` and `Max_grade(k)`. If your algorithms for `Insert(g)` and `Delete(g)` are similar to those of a known data structure discussed in class you can simply explain any modifications you need to make. However, you should carefully explain how you implement `Max_grade(k)`. You may refer to data structures and their complexities discussed in class without proof.

Sample Soln.

Consider an AVL tree with grade as key and number of occurrences as value. Also at each node `x` store the largest number of occurrences of a grade in the subtree rooted `x` (call this field `max`).

For notational purposes we will pass both the root and as well as the required parameter for each operation.

`Insert(v, g)` and `Delete(v, g)` are the same as the standard AVL tree operations with the additional update requirement of the `max` field. When a grade is inserted, after updating it's value, travel back up the tree updating the `max` value as needed. Similarly, for delete.

When looking for `Max_grade(,v,k)` we will always try to go right as long as there is `max` value of at least `k`.

```
Max_grade(v, k):
  if v.right.max ≥ k:
    #there is a larger than that at v, with the required number of occurrences
    return Max_grade(v.right, k)
  else if v.value ≥ k
    # v.value has the desired value >= k,and will be larger than any grade in the left subtree
so return v.grade
    return v.grade
  else
    # recurse in the left subtree because we haven't found a large enough k value yet
    return Max_grade(v.left, k)
```