Solutions & Marking keys to Workshop Test 2

(Please Note: the sample solutions are for reference only.- there are alternative solutions for each of the questions)

1. (2 marks)
   a) A binary tree has 600 nodes. What is the maximum possible depth of the tree? And what is the minimum possible depth of the tree?
   b) List the key properties (or features) of a binary search tree.

Answer:

a) A binary tree reaches maximum possible depth when it is extremely ill-balanced, i.e., each node has one child only (except the leaf node/s). In such a case, $d = n-1$.
Therefore, for $n = 600$, the max possible depth is d = 600 - 1 =599. -- (0.5 mark)

A binary tree reaches minimum possible depth when it is balanced.
An n-node balanced binary tree of depth d has at least 2^d nodes and at most $2^{d+1} - 1$ nodes, i.e. $2^d \le n \le 2^{d+1} - 1$, or $d \le \log_2 n < d+1$. Or $d = floor(\log_2 n)$.
For n =600, the minimum possible depth is $d = floor(\log_2 (600)) = 9$ -- (0.5 mark)

b) The key property of a BST is:
For any node in the BST, if it contains element with key *elem*, then
- Its left subtree (if nonempty) contains only elements with keys less than *elem*.
- Its right subtree (if nonempty) contains only elements with keys greater than *elem*.

-- (1 mark)

2. (2 marks)

The two elementary array sorting algorithms, *Selection* and *Insertion,* have the same time complexity of $O(n^2)$ on average case. However, their performances, in terms of number of comparisons, can be quite different in some special cases even on average cases. Compare the two algorithms and determine

a) If A[ ] is already sorted, which of the above-mentioned sorting algorithms may achieve an O(*n*) time complexity to complete the sorting procedure? Explain your answer using an example.

b) Which algorithm is better to sort A[0…n-1], in general case, in term of the number of comparisons? Why?

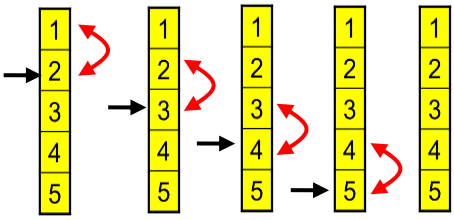
Answer (reference only):

Accept similar results as long as the procedures are correct.

a) Insertion sort algorithm.

When running the Insertion sorting algorithm (see slide 14 in Lecture04), as the array has been sorted, Step 1.2 would make one comparison only to determine the exact position (or index number) where current arrar component should insert into (And, in addition, as the array is sorted, the current component, *Val*, will be inserted into the original place/position where the current element was taken from). – This means that the **for** loop needs O(1) time to complete the comparison and insertion. This will repeat for *n*-1 time in Step 1, therefore the time complexity will be (n-1) *O(1) time, or O(n) time.

An example: let *n* = 5, and A[] = [1, 2, 3, 4, 5] .



On the other hand, Selection sort algorithm would always need $O(n^2)$ time to sort the array because Step 1.1 will have to scan the unsorted part to find the least.

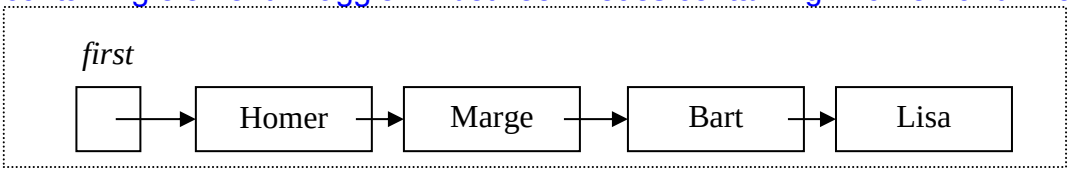
-- (1 mark)

b) *Insertion* algorithm is better than *Selection* sort in terms of number of comparisons ($n^2/4$ vs $n^2/2$), while *Selection* algorithm is better than *Insertion* sort in terms of number of copies ($2n$ vs. $n^2/4$) (see the results from slides 36 of Lecture 04).

For comparison, *Selection* will need to scan the *whole range of unsorted part of the array* t SELECT the minimum value, while *Insertion* sort may only need to scan *a part of the sorte part of the array* to determine the position where the current value is to be INSERTed.

In general, as the number of comparisons of Insertion algorithm (i.e., ~ $n^2/4$) is less than that the Selection algorithm (i.e., ~ $n^2/4$), Insertion algorithm is better if the object size is not to large. In case of large object sizes, people may prefer Selection sort algorithm due to smale number of moves (see the result from slides 36 of Lecture 04). -- (1 mark) *we have*

3. (1 mark)
Given the following linked list, write one Java-like sentence/s to insert a node containing element "Maggie" in between nodes containing "Homer" and "Marge".



Answer (reference only):

(one sentence)

```
first.succ = new SLLNode("Maggie", first.succ);
```

(or use two sentences)

```
SLLNode  c = new SLLNode("Maggie", first.succ);   -- (0.5 mark)
first.succ = c;                                    -- (0.5 mark)
```

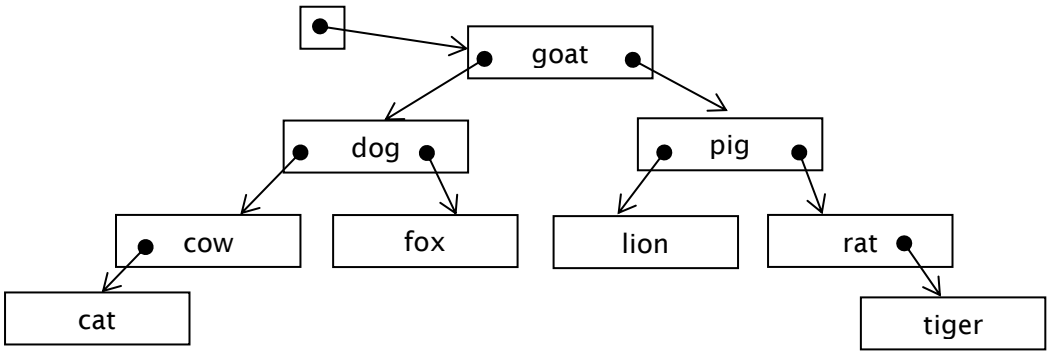
4. (3 marks)
a) Insert the following animals into an empty BST (you may show the final BST only):

goat, pig, dog, cow, rat, cat, tiger, fox, lion

b) After the insertions, show the results of *pre-order*, *in-order*, and *post-order* traversals of the BST.

Answer:

(a) After insertion of all data,



-- (1.5 marks)

(b) **Pre-Order**: *goat, dog, cow, cat, fox, pig, lion, rat, tiger* -- (0.5 mark)

In-Order: *cat, cow, dog, fox, goat, lion, pig, rat, tiger* -- (0.5 mark)

Post-Order: *cat, cow, fox, , dog, lion, tiger, rat, pig, goat* -- (0.5 mark)

5. (2 marks)

An array *A*[0 … *n-1*], where *n* > 10, stores the result of assignment 1 of this unit. Assume that all marks are integers, and the array is unsorted. By a preliminary rule, all students who achieved a mark on the top-10 mark-list of the class will be qualified to receive an award.

a) Write an algorithm, *print_top10(A, n),* that prints the 10 top marks of the class.
b) Analyse your algorithm using *O*-notation.

Option 1

a) Algorithm Print_top10(A, n)

1. if (n <= 10) terminate; // may not need this sentence
2. Sort the array A using Selection-sort algorithm; // or whatever array sorting algorithms
3. for i= n-1, n-2 …, n-10, repeat //print top 10 marks
   Print A[i];
4. Terminate. -- (1 mark)

b)

Let n be the number of elements in the array. Step 2 need O(n^2) to sort the array (depending on which sorting algorithm you used, this might be O($n \log_2 n$)). All other steps are done in a constant time O(1).

Therefore the time complexity of the algorithm is:

S1 + S2 + S3 + S4

O(1) + O(n^2) + O(1) + O(1) = O(n^2) -- (0.5 mark)

Overall: -- (0.5 mark)

Option 2 (*mimic Selection sort algorithm, for the 1st 10 steps*)

a) Algorithm Print_top10(A, n)

1. if (n <= 10) terminate; // may not need this sentence
2. for *i* =0, 1, …, 9, repeat //print top 10 marks
   2.1 Set p such that A[p] is the greatest value of A[0…*n*-1];
   2.2 If p ≠ *i*, swap A[p] and A[i]; //change the position of the array elements
   2.3 Print A[*i*];
3. Terminate. -- (1 mark)

b)

Let n be the number of elements in the array. Step 2.1 needs O(*n*) to complete the array search. Steps 2.2 & 2.3 each needs O(1) time. These steps are repeated for 10 times in Step2, so Step 2 need 10*O(*n*) = O(*n*) time. All other steps are done in a constant time O(1).

Therefore the time complexity of the algorithm is:

S1 + S2 + S3

O(1) + O(*n*) + O(1) = O(*n*) -- (0.5 mark)

Overall: -- (0.5 mark)