Preliminary solution proposal, DALG-part

1. a) True since $\frac{n^2}{\frac{n}{\log n}} \to 0$ when $n \to \infty$.
   b) False since $n \log n \notin O(n)$ (because $\frac{n \log n}{n} \to \infty$ when $n \to \infty$).
   c) True since $\frac{\sqrt{n}}{\log n} \to \infty$ when $n \to \infty$.
   d) True since $\log n \in O(n^2)$ (because $\frac{n \log n}{n^2} \to 0$ when $n \to \infty$). Assume $f(n) \in O(n \log n)$. By transitivity of $O$-notation, $f(n) \in O(n^2)$.
   e) False. Find $f(n) \in \Omega(n)$ such that $f(n) \notin \Theta(n^2)$. For example $n \notin \Theta(n^2)$, since $n \notin \Omega(n^2)$ (because $\frac{n}{n^2} \to 0$ when $n \to \infty$).
   f) True. Assume $f(n) \in \Theta(n)$. By def. of $\Theta$, $f(n) \in \Omega(n)$ and hence $n \in O(f(n))$.

   Now, $\log n \in O(n)$ since $\frac{\log n}{n} \to 0$ when $n \to \infty$. By transitivity of $O$-notation, $\log n \in O(f(n))$ and therefore $f(n) \in \Omega(\log n)$. 

1
3.

Tree representation

```
  4
 /  \
5 /   \
 /     \
/       \
5       9
  /  \
14 /  \
   /   \n  63
```

Array representation

```
[4,5,9,14,23,42,49,63]  [5,14,9,63,23,42,49]  [3,4,9,5,23,42,49,63,14]
```

(a)  (b)  (c)

5.  a) There are five different ways to evaluate $ABCD$:

$\langle ABC \rangle D$ First, $X=BC$ requires $10 \cdot 40 \cdot 30 = 12000$ (scalar) multiplikations. Then $Y=AX$ requires $50 \cdot 10 \cdot 30 = 15000$ multiplikations. And finally, $YD$ requires $50 \cdot 30 \cdot 5 = 7500$ multiplications. This sums up to a total of 34500 scalar multiplikations.

$\langle (AB)C \rangle D$ $X=AB$ requires $50 \cdot 10 \cdot 40 = 20000$ multiplikations. $Y=XC$ requires $50 \cdot 40 \cdot 30 = 60000$ multiplikations. And finally, $YD$ requires $50 \cdot 30 \cdot 5 = 7500$ multiplications, summing up to a total of 87500 scalar multiplikations.

$A\langle (BC)D \rangle$ $X=BC$ requires $10 \cdot 40 \cdot 30 = 12000$ multiplikations. $Y= XD$ requires $10 \cdot 30 \cdot 5 = 15000$ multiplikations. And finally, $AY$ requires $50 \cdot 10 \cdot 5 = 2500$ multiplications, summing up to a total of 16000 scalar multiplikations.

$A\langle B(CD) \rangle$ $X=CD$ requires $40 \cdot 30 \cdot 5 = 6000$ multiplikations. $Y=BX$ requires $10 \cdot 40 \cdot 5 = 2000$ multiplications. And finally, $AY$ requires $50 \cdot 10 \cdot 5 = 2500$ multiplications, summing up to a total of 10500 scalar multiplikations.

$\langle AB \rangle (CD)$ $X=AB$ requires $50 \cdot 10 \cdot 40 = 20000$ multiplikations. $Y=CD$ requires $40 \cdot 30 \cdot 5 = 6000$ multiplikations. And finally, $XY$ requires $50 \cdot 40 \cdot 5 = 10000$ multiplications, summing up to a total of 36000 scalar multiplikations.

The most efficient way to evaluate $ABCD$ is therefore as $A\langle B(CD) \rangle$.

b) No. Construct a counterexample. For example, replace $D$ above with a matrix $D'$ with dimensions $30 \times 20$. $A\langle B(CD') \rangle$ will then require 42000 multiplikations. On the other hand, $A\langle (BC)D' \rangle$ will require 28000 and hence $A\langle B(CD') \rangle$ is not the best order to evaluate $ABCD'$.
c) In principle, the solution boils down to enumerating all possible ways to parenthesize the product $A_0A_1\cdots A_n$ and calculating the number of scalar multiplications required for each. This enumeration can be done by parenthesizing the top level, $(A_0A_1\cdots A_i)(A_{i+1}\cdots A_n)$, and then recursively go on to parenthesize the left and the right factors respectively. However, since we ultimately is searching for a way to parenthesise the product that minimizes the number of scalar multiplications required, an exhaustive search is unnecessary. In order to minimize the cost for evaluating the whole product $(A_0A_1\cdots A_i)(A_{i+1}\cdots A_n)$, we obviously have to minimize the cost of evaluating each of the two factors individually. Thus, if $M_{\text{Left}, \text{Right}}$ is the number of multiplications required in an optimal ordering, then, if $\text{Left} < \text{Right}$,

$$M_{\text{Left}, \text{Right}} = \min_{\text{Left} \leq i < \text{Right}} \{ M_{\text{Left},i} + M_{i+1,\text{Right}} + \text{rows}_{\text{Left}} \cdot \text{columns}_i \cdot \text{columns}_{\text{Right}} \}$$

This recursive equation is encoded in the algorithm below.

\[\text{function } \text{FindMin}(\text{integer Left, Right}): \text{integer}\]

\[\text{CompMin} \leftarrow -1\]

\[\text{if } (\text{Right} \leq \text{Left}) \text{ then}\]

\[\text{return 0}\]

\[\text{else}\]

\[\text{for } i=\text{Left} \text{ to } \text{Right}-1 \text{ do}\]

\[\text{Comp1} = \text{FindMin}(\text{Left},i)\]

\[\text{Comp2} = \text{FindMin}(i+1,\text{Right})\]

\[\text{Comp3} = \text{Complexity}(\text{Left},i,i+1, \text{Right})\]

\[\text{Comp} = \text{Comp1} + \text{Comp2} + \text{Comp3}\]

\[\text{if } (\text{Comp} < \text{CompMin} \text{ or CompMin} = -1) \text{ then}\]

\[\text{CompMin} \leftarrow \text{Comp}\]

\[\text{return CompMin}\]
7. a) The hashtable contains the entries: [22, empty, 12, 3, 13, 4, 36, 7, 23, 9]
   b) 9 probes are required for \textit{findElement}(22).
   c) The \textit{load factor} is the parameter \(n/N\) (or \(\lceil n/N \rceil\)), where \(n\) is the size of the dictionary to be stored and \(N\) is the size of the hash table. The expected running time of dictionary operations - \textit{findElement}, \textit{insertItem} and \textit{removeItem} - is proportional to the number of probes required, which is \(O(\lceil n/N \rceil)\) (assuming a good hash function, i.e. such that a given key is equally likely to be mapped to any one of the \(N\) hash buckets). As the load factor increases, the risk of collisions increases and with it the number of probes. It is therefore desirable to keep the load factor below some specified threshold. As more and more items are inserted in the dictionary, the load factor increases. To retain a low load factor, the hash table must be made larger. Rehashing is the process of reinserting all the items in the dictionary into a new hash table, using a (new) hash function defined for that table. The cost of rehashing can be amortized. Doubling the size of the table with each rehashing operation, makes the amortised cost of an \textit{insertItem} operation still being \(O(1)\) (assuming the load factor is kept below 1).
   d) In open addressing, \textit{findElement}, for example, stops probing when either the item sought for is found, or when an empty bucket is found (in which case it is concluded that the item is not in the dictionary). Removing an element creates an empty bucket, which may cause \textit{findElement} to stop probing, when the item searched for indeed exists in the dictionary and would have been found if probing had continued passed the empty bucket. The dictionary operation \textit{removeElement} must therefore be implemented in such a way that this situation cannot occur. A common way to solve this is to replace the deleted item with a special marker “deactivated item”.

\textit{findElement} (and \textit{removeElement}) are implemented so that probing will (as before) stop when an empty bucket is encountered, but will continue if the bucket has the “deactivated item”-marker attached to it.

\textit{insertElement} stops probing when either an empty bucket or a bucket with the “deactivated item”-marker is encountered places the new item there (and removes the marker).

9. a) No. There are two nodes that have only one child.
   b) No. Levels 2 and 3 are not filled completely. Secondly, levels 3 and 4 are not filled from left to right.
   c) No. The subtree rooted at the left child of the root has height 4 and the subtree rooted at the right child of the root has height 2. Thus the root has balance -2 (see figure below).
d) Yes. All nodes have balances -1, 0 or 1 (see figure above).

e) Preorder: 47, 35, 11, 3, 7, 15, 43, 45, 49, 53, 55
   Inorder: 3, 7, 11, 15, 35, 43, 45, 47, 49, 53, 55
   Postorder: 7, 3, 15, 11, 45, 43, 35, 49, 55, 53, 47

f) The tree resulting from a single left rotation at the root is shown in the figure below.
   It is an AVL-tree since all nodes have balances -1, 0 or 1.