Australian National University
Research School of Computer Science

COMP3600/COMP6466 in 2016 – Quiz Three

Due: 23:55pm Friday, September 23

Submit your work electronically through Wattle. The total mark of this quiz worths 20 points, which is worth of 4.5 points of the final mark.

Question 1 (4 points).
Given a positive integer sequence $a_1, a_2, \ldots, a_n$, A maximum weighted decreasing subsequence is a subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that the weighted sum $\sum_{j=1}^{k} a_{i_j}$ of the sequence is the maximum one and $a_{i_j} \geq a_{i_{j'}}$ if $j < j'$, where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Following the four steps in the design of DP algorithms, devise a DP algorithm that takes a sequence $a[1 \ldots n]$ and returns a maximum weighted decreasing subsequence. Analyse the running time of your proposed algorithm.

Answer:

- Let $a_1, a_2, \ldots, a_n$ be the given sequence of n distinct positive integers. The key property is: for any j, the maximum-weighted decreasing sequence ending with a_j either consists of a_j only, or a maximum-weighted decreasing sequence ending with some earlier a_i that is larger than a_j with $i < j$. This gives a recurrence.
- Denote by $W(j)$ the weight of the maximum-weighted decreasing sequence ending with a_j for all j with $1 \leq j \leq n$. Then, for $1 \leq j \leq n$,

$$W(j) = \max_{i<j \& a_i \geq a_j} \{a_j, W(i) + a_j\},$$

and

$$W(1) = a_1.$$

Define another one dimensional index array I as follows.
$I(j) = i_0$ if $W(j) = W(i_0) + a_j$ if $a_{i_0} \geq a_j$ and $1 \leq i_0 < j \leq n$, otherwise, $I(j) = j$.

The maximum weight of the maximum-weighted decreasing subsequence of the given sequence thus is $\max_{1 \leq i \leq n}\{W(i)\}$.

- Algorithm Implementation and the time complexity analysis.

  Answer: The algorithm implementation is quite straightforward. We calculate $W(1), W(2), \ldots, W(n)$, then find an index i such that $W(i)$ is the maximum one. Clearly, this algorithm takes $\sum_{j=1}^{n} O(j) = O(n^2)$ time as the calculation of each $W(j)$ takes $O(j)$ time.

- Use the pseudo-code to describe the procedure of printing out the found subsequence.

  Answer: In order to find the subsequence, we have an index array I in step 3 for book-keeping purpose of calculating $W(j)$. Having both arrays $W(.)$ and $I(.)$, the following procedure can be used to find the maximum-weighted increasing subsequence.

  Let $W(j)$ be the maximum value and l the length of the subsequence, then the maximum-weighted increasing subsequence is $a_{I^{l-1}(j)}, a_{I^{l-2}(j)}, \ldots, a_{I^1(j)}, a_j$ where $I^k(j) = I(I^{k-1}(j))$, $I^0(j) = j$ and $I^1(j) = I(j)$.

Question 2 (5 points).
Given n items with item i having weight $w_i > 0$ and a profit $p_i > 0$ for all i, $1 \leq i \leq n$, assume that each item can be cut into an arbitrary fraction if needed. The fractional knapsack problem is to pack as many items as possible to a knapsack with capacity W such that the total profit of items (or a fraction of an item) in the knapsack is maximized. Show how to solve this fractional knapsack problem in $O(n)$ time. Notice that you are asked to devise a linear running time algorithm for the problem, while an $O(n \log n)$ time algorithm can easily be devised. (*Hint: adopt the greedy strategy and the linear selection algorithm*).

Answer: Let the knapsack capacity be W and item i have weight w_i with the profit p_i for all i with $1 \leq i \leq n$. We order the items by the *decreasing order* of p_i/w_i. Let $i_1, i_2, \ldots, i_n$ be the sorted order of items. The greedy strategy adopted is to pack items $i_1, i_2, \ldots, i_{j-1}$ and item i_j with $(W - \sum_{l=1}^{j-1} w_{i_l})$ if $w_{i_j} \geq W - \sum_{l=1}^{j-1} w_{i_l}$. Then, the maximum profit of this solution is $S = \sum_{l=1}^{j-1} p_{i_l} + \frac{p_{i_j}}{w_{i_j}} \cdot (W - \sum_{l=1}^{j-1} w_{i_l})$. We claim that this solution S is an optimal solution, by contradiction.

Assume that part of an item $i_{j'}$ with $j' > j$, $w'_{i_{j'}}$, is added to the solution, then the same amount of an item (e.g, i_k with $k \leq j$) will be removed from the optimal solution,

then the profit delivered by the updated solution is

$$\begin{aligned} S' &= \sum_{l=1}^{j-1} p_{i_l} + \frac{p_{i_j}}{w_{i_j}} \cdot (W - \sum_{l=1}^{j-1} w_{i_l}) + (\frac{p_{i_{j'}}}{w_{i_{j'}}} \cdot w'_{i_k} - \frac{p_{i_k}}{w_{i_k}} \cdot w'_{i_k}) \\ &= S + w'_{i_k}(\frac{p_{i_{j'}}}{w_{i_{j'}}} - \frac{p_{i_k}}{w_{i_k}}) \\ &\leq S, \quad \text{as } \frac{p_{i_k}}{w_{i_k}} \geq \frac{p_{i_{j'}}}{w_{i_{j'}}} \ . \end{aligned} \tag{1}$$

Thus, the solution S is optimal. Clearly if the items are sorted, it takes $O(n)$ time. Otherwise, the sorting takes $O(n \log n)$ time.

The above solution with $O(n)$ time is assumed that the ratios of profits to weights of items are sorted. We now show that it still takes $O(n)$ time without this assumption. The detailed steps are as follows. Let R be the set of n items.

- Step 1. Calculate the ratio $r_i = p_i/w_i$ of each item i for all i with $1 \leq i \leq n$;
- Step 2. Find **the median** of the ratio sequence, let γ be the median;
- Step 3. Partition the items (their weights) into 3 disjoint subsets: $R_1 = \{w_i \mid p_i/w_i < \gamma\}$, $R_2 = \{w_i \mid p_i/w_i = \gamma\}$, $R_3 = \{w_i \mid p_i/w_i > \gamma\}$;
- Step 4. if the weighted sum of all items in R_3 is larger than W, then apply the algorithm on R_3 recursively. Otherwise, if the weighted sum of items in R_2 and R_3 is no less than W, make use of all items in R_3 and and add fractional items in R_2 to make up W. Otherwise, apply the algorithm on R_1 with the knapsack capacity of $W' = W - \sum_{item_i \in R_2 \cup R_3} w_i$ recursively. Clearly, it is an application of the linear selection algorithm, which takes $O(n)$ time.

Question 3 (2 points).
Show that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then, there exists an optimal code whose codeword lengths are monotonically increasing. (*Hint: adopt the greedy strategy*)

Answer: We show this by contradiction. Assume that there are two characters x and y with frequencies f_x and f_y, we further assume that $f_x < f_y$.Let c_x and c_y be the length of code words of x and y. Let $\mathcal{A}$ be the set of character in a document. Following the proposed approach, we assume there is a method for the character coding $c(\cdot)$ for the characters in $\mathcal{A}$ such that the length of the document after coding is $\sum_{z \in \mathcal{A}} f_z \cdot c_z$ is minimized. This coding has a property, that is, if $f_x < f_y$, then

$$c_x > c_y \tag{2}$$

Now, we assume that there is another coding c' for the document, in which all other character code words are identical to their corresponding ones in the original method, the only difference lies in the two characters x and y with $\{x, y\} \subseteq \mathcal{A}$, i.e., we swap the coding between x and y, in other words, for any $z \in \mathcal{A} \setminus \{x, y\}$ we have $c'_z = c_z$, while $c'_x = c_y$ and $c'_y = c_x$. Now, we have $c'_x < c'_y$ as $c_x > c_y$, the length of the document under this new coding thus is

$$\begin{aligned}
\sum_{z \in \mathcal{A}} f_z \cdot c'_z &= \sum_{z \in \mathcal{A}} f_z \cdot c_z - (f_x \cdot c_x + f_y \cdot c_y) + (f_x \cdot c_y + f_y \cdot c_x) \\
&= \sum_{z \in \mathcal{A}} f_z \cdot c_z + f_y(c_x - c_y) - f_x(c_x - c_y) \\
&> \sum_{z \in \mathcal{A}} f_z \cdot c_z, \text{since } f_x < f_y \text{ and } c_x > c_y, \qquad (3)
\end{aligned}$$

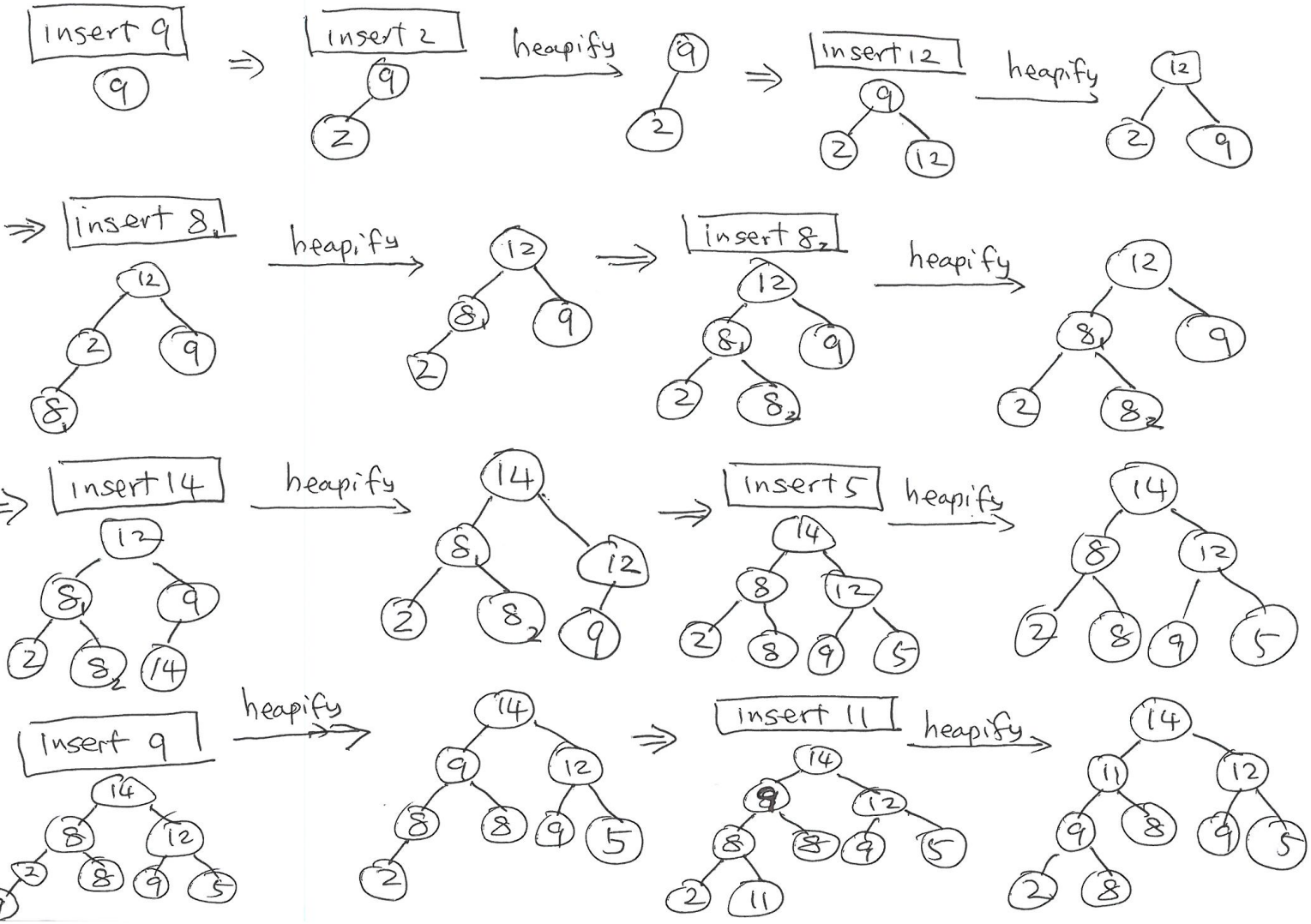
which means that the new coding delivers a solution that has a longer length of the document, this leads to a contradiction that the length is the smallest one. Thus, for any coding c'' if we aim to minimize the encoding length of the document, we must have $c''_x > c''_y$ if $f_x < f_y$.

Question 4 (6 points).
(a) Assuming an initial max-heap is empty, insert the keys $9, 2, 12, 8, 8, 14, 5, 9, 11, 10$ into the max-heap one by one (once a time) until all elements are inserted, then remove the key in the root repeatedly until the heap is empty again. (3 points)

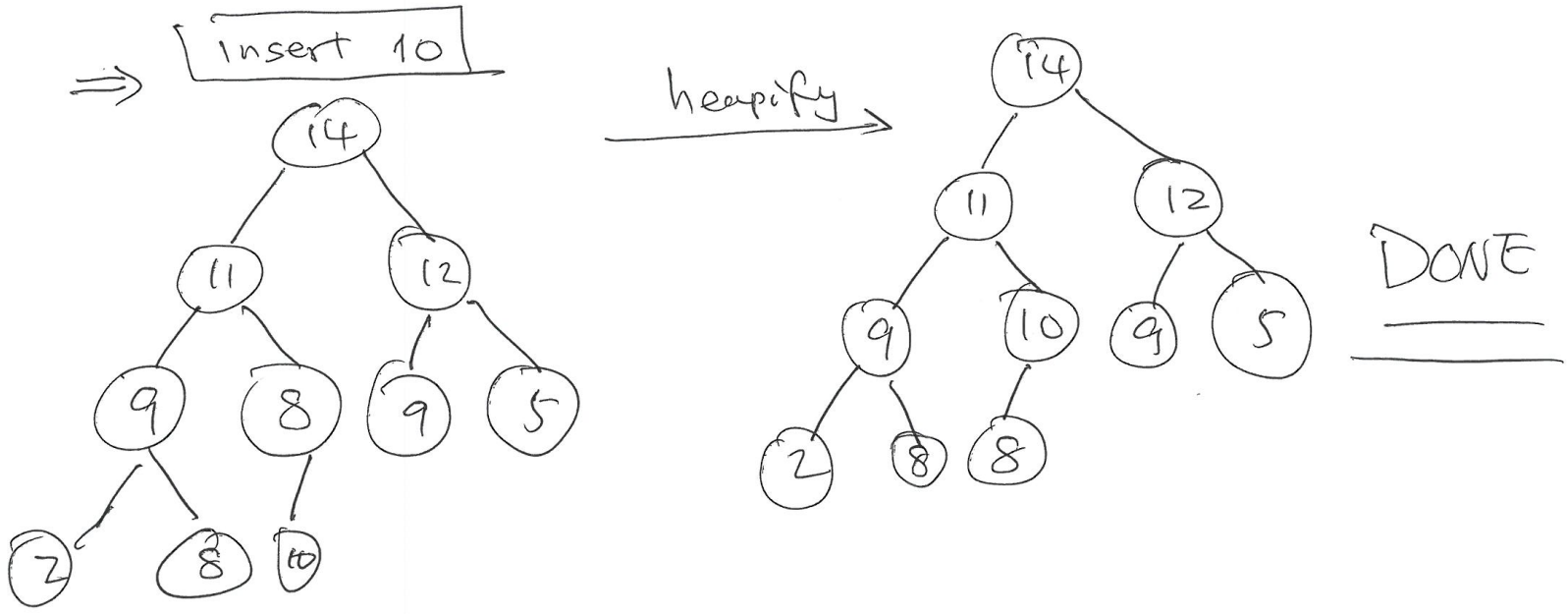
1. Use diagrams to illustrate each step of the insertion and deletion procedure.

   Answer:

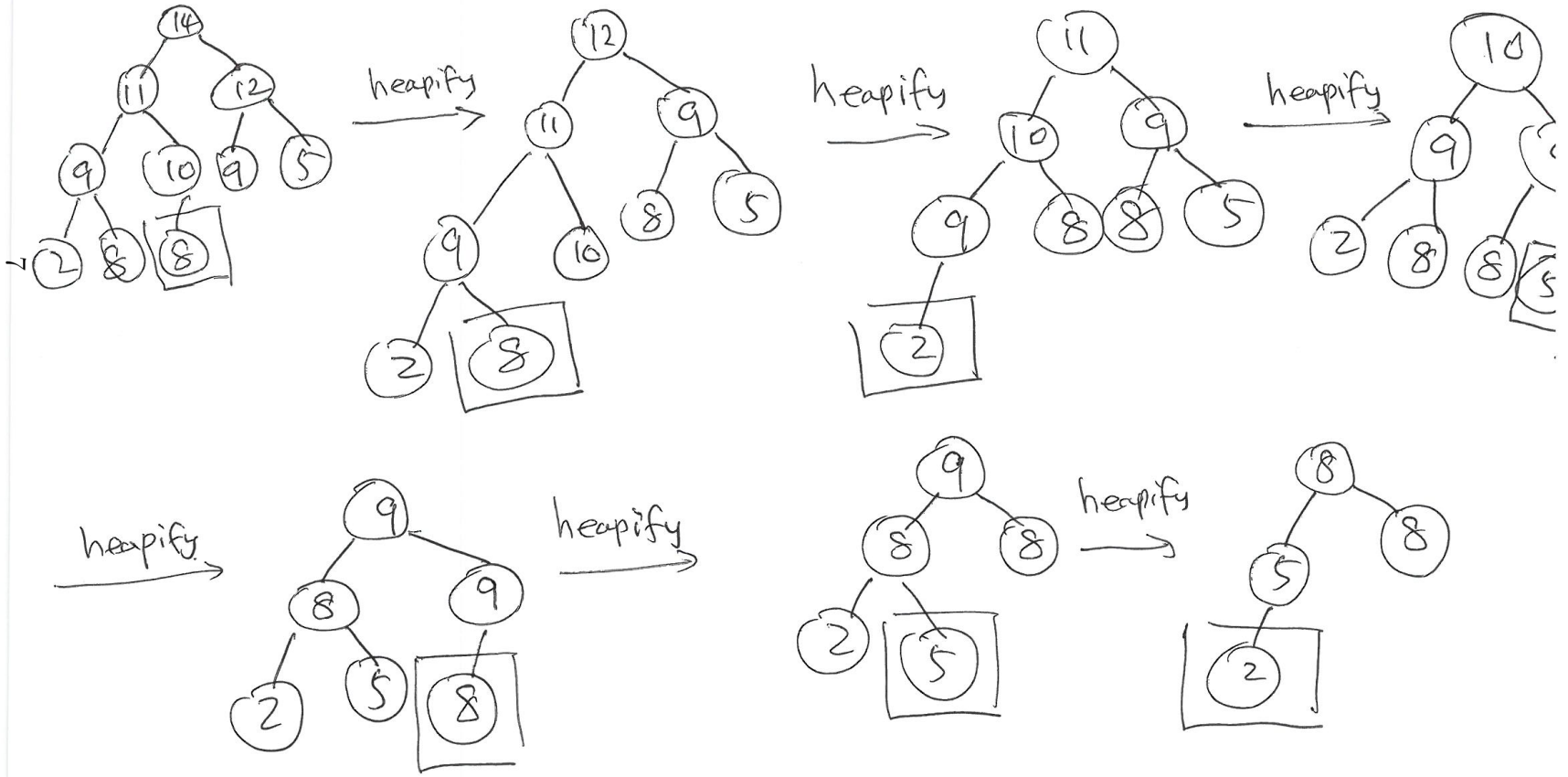
Q4(a) Page 1 . 9, 2, 12, 8_1, 8_2, 14, 5, 9, 11, 10



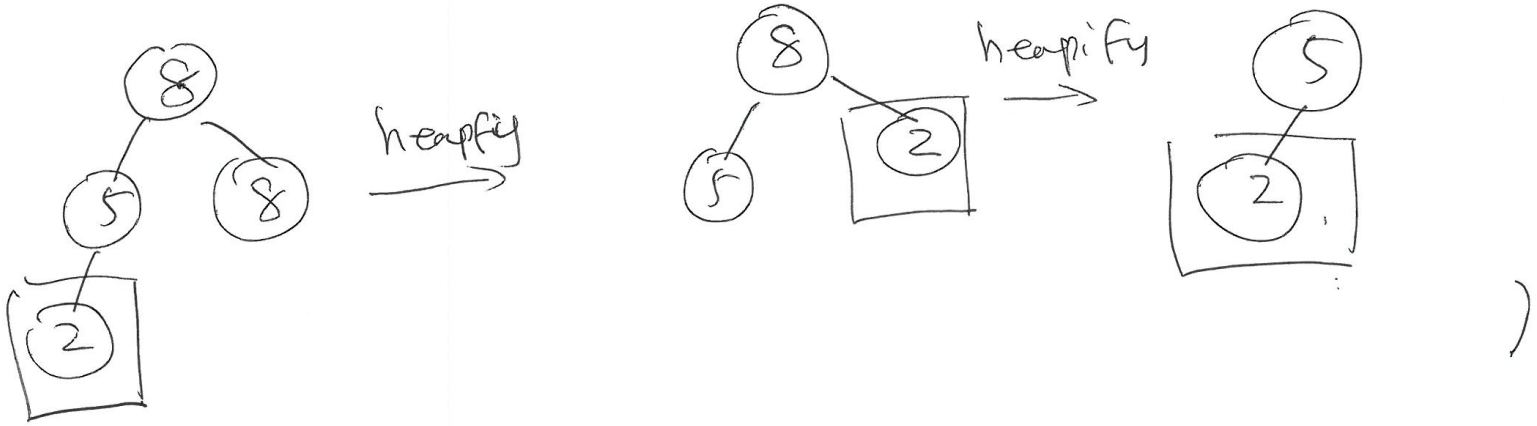
Q4(a) Page 2.



Q4(a) Page 3 Delete the root, put the last element (in the array) into the root position, then perform heapify in operation



Q4(a) Page 4:



heapify 2

Done

2. What is the time complexity of sorting in this fashion if there are n keys to be inserted to and then removed from the max-heap?

   Answer: The time complexity is $O(n \log n)$ if there are n elements. Since the initialization of a max-heap takes $O(1)$ time, performing heapifying operation takes $O(\log n)$ time when inserting an element into the heap. It takes $O(1)$ time to remove an element from the root and $O(\log n)$ time to add another element to the root as heapifying the max-heap is needed after that.

(b) In the open addressing schema, three probing techniques: `linear probing`, `quadratic probing`, and `double hashing` have been introduced. (1) How many different probing sequences can be generated for each of the schemes? justify your answer. (2) What are the advantages and disadvantages among these techniques? (3 points)

Answer: Assume that there are m slots in the hash table. Then,

- there are $O(m)$ probing sequences for `linear probing`,

  $$h(k, i) = (h'(k) + i) \mod m,$$

  $0 \leq h'(k) \leq m - 1$ and $0 \leq i \leq m$, as a different $h'(k)$ leads to a different probing sequence.

- There are $O(m)$ probing sequences for `quadratic probing`,

  $$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

  $0 \leq h'(k) \leq m - 1$, $0 \leq i \leq m$, c_1 and c_2 are constants, as a different $h'(k)$ leads to a different probing sequence.

- There are $O(m^2)$ probing sequences for `double hashing`.

  $$h(k, i) = (h_1(k) + i h_2(k)) \mod m,$$

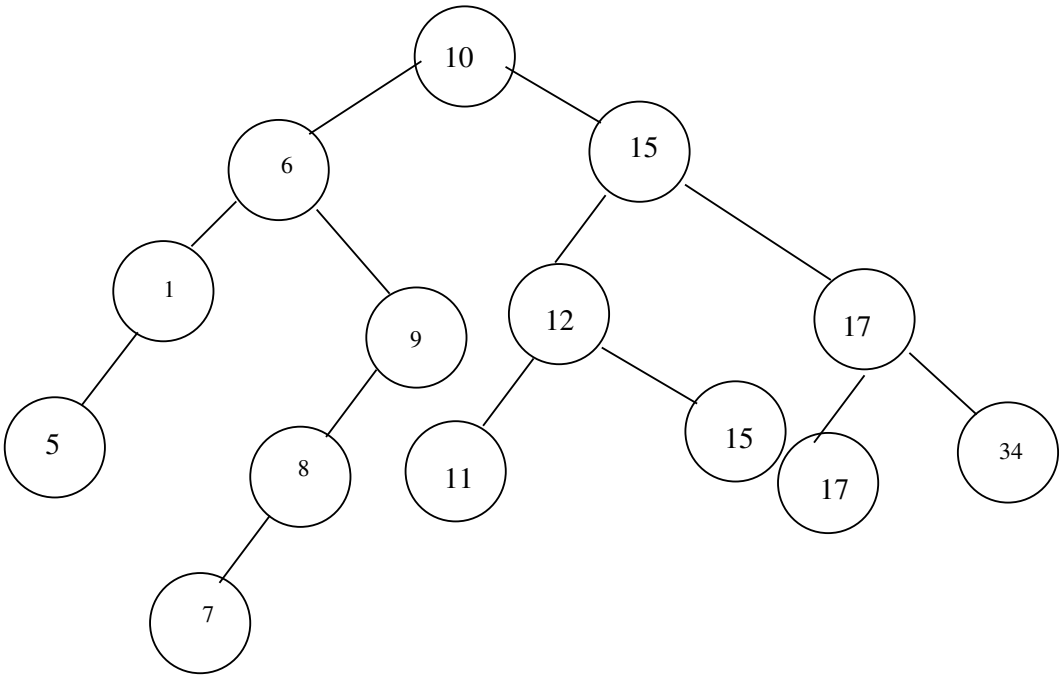
  $0 \leq h_1(k), h_2(k) \leq m - 1$ and $0 \leq i \leq m$, as a different pair of $(h_1(k), h_2(k))$ leads to a different probing sequence and there are $O(m^2)$ such pairs.

`Linear probing` is very simple, and takes less time, but it suffices the primary clustering problem. `Quadratic probing` avoids the primary clustering problem, its computation is easy, but it suffices the secondary clustering problem. `Double hashing` is an ideal hash approach. Although it prevents both primary and secondary clustering problems, it takes a much longer running time.

Question 5 (3 points).
Given an element sequence 10, 6, 15, 9, 12, 17, 1, 11, 34, 8, 7, 15, 17.

- Illustrate the final binary search tree by inserting the elements into the sequence one by one.

  Answer:



- Assume that a new element 12 will be inserted to the tree, show the resulting tree after the insertion.

Answer: