COMP 3804/Math 3804: Assignment 2

Solutions given by Jag Zhang

School of Computer Science Carleton University

Your assignment should be submitted online on Brightspace as a single .pdf file. The filename should contain your name and student number. No late assignments will be accepted.You can type your assignment or you can upload a scanned copy of it. Please, use a good image capturing device. Make sure that your upload is clearly readable. If it is difficult to read, it will not be graded.

Question 1 [15 marks]

Consider a $4-heap$ to be a heap-ordered tree in which each internal node has 4 children, all leaves are on at most two adjacent levels, and, on the last level, all leaves are as much to the left as possible. Let n be the total number of nodes in the tree and assume that $n = 4 * k + 1$, for some positive integer k.

1. State precisely:

    - how many leaf-nodes a $4 - heap$ on n nodes can have minimally and maximally,

      First, when we take the minimal value of positive integer $k = 1$, there will be only two levels in the heap. **The number of leaf-nodes will be minimally 4.**

      Second, the number of leaf-nodes depends on the total number of nodes n. It reaches the maximum when each level is full (complete heap). Then, the number of nodes on each level will be 4^l for level l. The total number of nodes will be the summation of the geometric series: $n = 1 + 4 + 4^2 + ... = \frac{4^{l+1}-1}{4-1}$. For the maximal level l, we have $4^{l+1} = 3n + 1$. Because there are 4^l leaf-nodes on the lowest level, **the number of leaf-nodes will be maximally $4^l = \frac{3n+1}{4}$.** We can also show this by another method in question 5.

      Other equivalent answers like $\lceil \frac{3n}{4} \rceil$, $4^{\log_4(3n+1)-1}$ should also be considered as correct.

    - how many nodes there are on each level, i, (excluding the lowest level),

      All the levels except the lowest level are full. **Thus, there will be 4^i nodes on each level i in the 4-heap.**

    - an expression, phrased in terms of n, for the height of the tree,

      Similar to question 1, suppose the height of the tree is h. There should be $1+4+4^2..+4$ to $1 + 4 + 4^2.. + 4^h$ nodes in the 4-heap. Then, the total number of nodes falls in the range $[\frac{4^h+11}{3}, \frac{4^{h+1}-1}{3}]$. **In this case, the height i can be written as $h = \lfloor \log_4 3n \rfloor$ or $h = \lceil \log_4 (3n+1) \rceil - 1$.** Other similar answers may also be correct. (Here we follow the definition that a leaf node has height 0)

    - how many nodes there are on all levels together (excluding the lowest level),

      Suppose there are l levels excluding the lowest level. The total number of nodes excluding the lowest level should be $\sum_{i=0}^{l} 4^i = \frac{4^{l+1}-1}{4-1}$. As we already have the height of tree: $h = \lfloor \log_4 3n \rfloor$, which has a relation $h = l + 1$ with number of levels l. **Thus, there are $\frac{4^h-1}{4-1} = \frac{4^{\lfloor \log_4 3n \rfloor}-1}{3}$ nodes on all levels together excluding the lowest level.** Other similar answers may also be correct.

- a relation between the total number of internal nodes and the number of leaves.

  For every 4 leaf-nodes we add to the heap, one leaf-node from the original heap becomes an internal node. Thus, the number of leaf-nodes is 3 times the number of internal nodes. Consider there are only two levels in the heap, the number of the leaf-nodes is 4, and there is only one internal node. **Let the number of internal nodes be i, then the number of leaf-nodes is $3i + 1$.**

  Because we have the total number of nodes in the heap $n = 4k+1$, the number of internal nodes will be $i = k = \frac{n-1}{4}$. The number of leaf-nodes will be $3i + 1 = 3k + 1 = \frac{3n+1}{4}$.

2. Assume the tree is stored in an array H[1 ... n]. (Note: we start with index 1 not 0.) Give address formulae for: the children of an internal node stored at H[i] and the parent node of a node stored at H[i].

   Suppose a node x is the j^{th} node on the i^{th} level for $i = 0, 1, ...$ and $j = 1, 2, ...$ Then, there are $\frac{4^i-1}{3}$ nodes before node x. The index of node x will be $x = \frac{4^i-1}{3} + j$. Let y be the first children of x. We do the same thing to y: there are $\frac{4^{i+1}-1}{3}$ nodes before node y. As x is k^{th} node on level i. Because each node on level i before x has 4 children. The index of the first children y will be $y = \frac{4^{i+1}-1}{3} + 4j - 3 = 4(\frac{4^i-1}{3} + j) - 2$. Because we have $x = \frac{4^i-1}{3} + j$, then the index of y should be $y = 4x - 2$.

   In this case, for arbitrary index i, the 4 children nodes of node H$[i]$ will be H$[4i-2]$, H$[4i-1]$, H$[4i]$, H$[4i+1]$. Reversely, the parent node of H$[i]$ will be H$[\lfloor\frac{i+2}{4}\rfloor]$. Other similar answers may also be correct. One can also find the formulae by plotting the tree.

Question 2 [11 marks]

1. For standard binary heaps, we also store the heap in an array H[1....n]. Now, state parent/child index relations, but use the bit representation of the array indices. So, state (child to parent) and (parent to child) array index relations using the binary representation of indices. So, we get e.g., the children of the root are stored at binary(10) and binary(11) and the parent of binary(10) and binary(11) is the root stored at binary(1).

   For node x stored at binary$(b_1...b_m)$, the parent node is stored at the index without the last bit. We can move the binary representation of node x 1 bit to the right. Thus, the parent node of x will be

   parent(x) = binary$(b_1...b_m) >> 1$ = binary$(b_1...b_{m-1})$

   Similarly, the two children nodes are stored adjacently to 1 bit to the left of the binary representation of node x. Thus, they are stored at the location where the last bit of 0 or 1 is added to the binary index of node x:

   leftChild(x) = binary$(b_1...b_m) << 1 + 0$ = binary$(b_1...b_m0)$

   rightChild(x) = binary$(b_1...b_m) << 1 + 1$ = binary$(b_1...b_m1)$

2. Let the i^{th}parent(node) be recursively defined as follows:

   for $i = 0$ the i^{th}parent(node) is the node itself.

   for $i > 1$, i^{th}parent(node) is the parent of $(i-1)^{st}$parent(node).

   Now, state a simple formula (in binary) for the i^{th} parent of a node stored at binary$(b_1...b_m)$.

   To solve the formula for the i^{th} parent, we can recursively apply the parent function in the previous question. Suppose we have a node x. Because each time we call the parent function, the binary representation of node x moves 1 bit to the right. After we call the parent function for i times, the i^{th} parent of node x will be:

$$\begin{aligned} \text{parent}(x, \text{i}) &= \text{parent}(\text{parent}(\ldots\text{parent}(x)) \\ &= \text{binary}(b_1 \ldots b_m) >> 1 >> 1 \ldots >> 1 \\ &= \text{binary}(b_1 \ldots b_m) >> \text{i} \\ &= \text{binary}(b_1 \ldots b_{m-i}) \end{aligned}$$

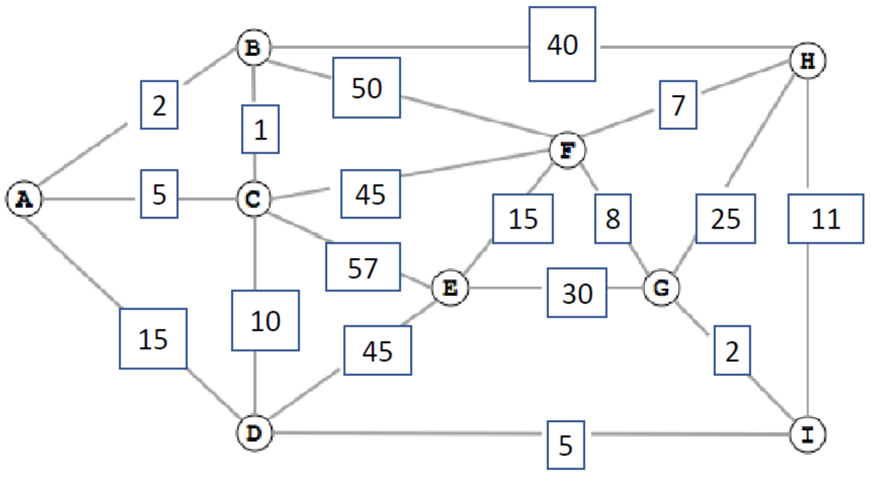


Figure 1: The input graph for Question 3.

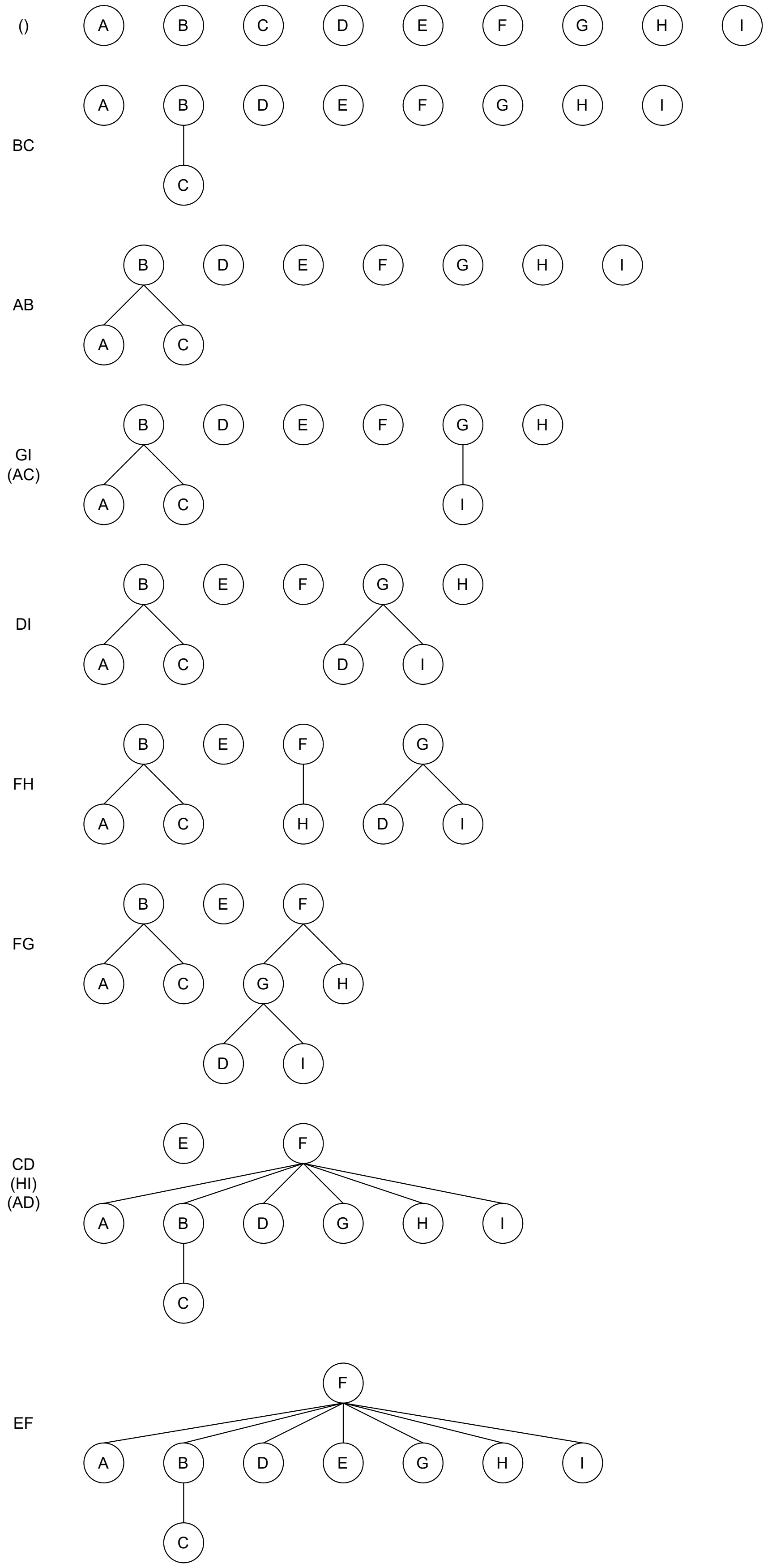
Question 3 [20 marks]

Suppose that we want to find a minimum spanning tree of the graph shown on Figure 1.

- Run Prim's algorithm on this graph. Start from vertex A and whenever there is a choice of vertices, always select in the alphabetic order. Draw a table showing the intermediate values at each step.

step	vertex	A	B	C	D	E	F	G	H	I
0		0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
1	A		2/A	5/A	15/A	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
2	B			1/B	15/A	∞/nil	50/B	∞/nil	40/B	∞/nil
3	C				10/C	57/C	45/C	∞/nil	40/B	∞/nil
4	D					45/D	45/C	∞/nil	40/B	5/D
5	I					45/D	45/C	2/I	11/I	
6	G					30/G	8/G		11/I	
7	F					15/F			7/F	
8	H					15/F				
9	E									

- Run Kruskal's algorithm on the same graph. Clearly show the disjoint-set data structure (i.e., the structure of the directed trees) at every intermediate step, assuming union using path compression.


()
A B C D E F G H I
BC
A B D E F G H I C
AB
B D E F G H I A C
GI
(AC)
B D E F G H A C I
DI
B E F G H A C D I
FH
B E F G A C H D I
FG
B E F A C G H D I
CD
(HI)
(AD)
E F A B D G H I C
EF
F A B D E G H I C

Here we assume both searching of edges with least weights and calling union function are in alphabetic order. The left column shows the edges we traverse in each iteration, and the edges in the parentheses show the results after calling find function. To make the plot shorter, we only keep the union function in each iteration. Other equivalent implementation of the disjoint-set may also be correct.

Question 4 [20 marks]

Let $G = (V, E)$ be a graph with distinct weights. Let $e_i, i = 1...|E|$ be the list of edges, sorted by weight. Let $j < k$, be two indices such that e_j is not in the minimum spanning tree of G, but e_k is. Prove or disprove that the removal of e_j (from the graph) cannot disconnect G.

Prove: Removal of e_j (from the graph) cannot disconnect G.

Proof. We can either prove it directly or by contradiction. Here we prove it directly with two steps:
First, we show the edge e_j is in a cycle.
We follow the process of Kruskal's algorithm to find MST of G. Let w_i denote the weight of edge e_i. Because e_i is sorted by the weight, for $j < k$, we have $w_1 < w_2 < ... < w_j < ... < w_k < ... < w_{|E|}$. In the process of Kruskal's algorithm, in each iteration, we take the edge with minimum weight. For an edge, $e = (u, v)$, the connection of two vertices u and v should connect two trees. In other words, u and v should not be both in the path of the tree we created. Adding the edge e should not create a cycle in the tree.
Because e_j is not in the MST of G, there can be two possible cases: First, e_j creates a cycle in the generated tree. Second, e_j has larger weights than other edges, and it should not be taken in the MST.
We know the fact that $w_k > w_j$, and e_k is in the MST of the graph G. If it falls in the second case, Kruskal's algorithm will take e_j instead of e_k in the MST, which shows a contradiction to Kruskal's algorithm. Thus, taking e_j in the MST must create a cycle in the MST.
Second, we show removal of e_j does not disconnect G.
Because e_j along with the edges in the MST of G are all the edges in graph G, edge e_j should also be in a cycle of graph G. Recall the fact that the removal of an edge from the cycle does not disconnect the graph. The removal of edge e_j cannot disconnect graph G.

$\square$