Analysis of Algorithms Homework 1

USC ID: 3725520208
Name: Anne Sai Venkata Naga Saketh
Email: annes@usc.edu

1Q) Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if f(n) = O(g(n))

Solution:

The increasing order of time complexity or growth rate is as follows: 1 < log (n) < n < n.log(n) < n^2 < 2^n < n!

Simplifying the given terms in the question to compare them with each other and determine the increasing order:

a) $\sqrt{n}^{\sqrt{n}} = n^{\sqrt{n} \times 1/2} = \boldsymbol{n^{\sqrt{n}/2}}$
b) $\boldsymbol{n^{\log(n)}}$ is already in its simplest format and cannot be simplified further
c) $\boldsymbol{n.\log(\log(n))}$ is already in its simplest form and cannot be simplified further
d) $n^{\frac{1}{\log(n)}}$

1 in the numerator of the exponent can be written as log2
$n^{\frac{1}{\log(n)}} = n^{\frac{\log 2}{\log(n)}}$

By using the logarithmic properties $\frac{\boldsymbol{log_x^a}}{\boldsymbol{log_x^b}} = \boldsymbol{log_b^a}$
Exponent $\frac{\log 2}{\log n}$ can be converted to $\boldsymbol{log_n^2}$, resulting in

$n^{\boldsymbol{log_n^2}}$

Using logarithmic properties again $\boldsymbol{a^{log_a^x} = x}$, this can be simplified as **'2'**

e) $2^{\log(n)}$ = by using same logarithmic property stated above this term can be simplified as **n**
f) $log^2 n$ can be simplified as $(\log n) * (log n)$, this can be converted to $(\log n)^2$
g) $(\log(n))^{\sqrt{\log(n)}}$ is in its simplest format

By comparing $log^2 n$ and $(\log(n))^{\sqrt{\log(n)}}$ by substituting large values of n for example 2^{16} *or* 2^{64}, we found that $(\log(n))^{\sqrt{\log(n)}} > log^2 n$

As per the increasing order of growth rate we have considered above, we shall consider and compare the

1. Constant terms
2. Log(n) terms
3. n terms
4. n.log(n) terms
5. n^2 terms
6. 2^n $terms$
7. n! terms

so as per the simplification done above, the final increasing order of growth rate shall be,

$$\boldsymbol{n^{\frac{1}{\log(n)}} < log^2 n < (\log(n))^{\sqrt{\log(n)}} < 2^{\log(n)} < n.\log(log(n)) < n^{\log(n)} < \sqrt{n}^{\sqrt{n}}}$$

2Q) The Fibonacci sequence F_1, F_2, F_3 . . . , is defined as follows. $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$.

Solution:

$$Given\ Statement\ is\ F(n) = \frac{a^n - b^n}{\sqrt{5}}\ where\ a = \frac{1+\sqrt{5}}{2}\ and\ b = \frac{1-\sqrt{5}}{2}$$

Step 1: (Base case, prove the statement is true for value n = 1)

As per the induction process, we must prove F_n for n = 1 and

given in the question that F(1) = 1

$F(1) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} = \frac{(1+\sqrt{5}-1+\sqrt{5})}{2*\sqrt{5}} = \frac{2*\sqrt{5}}{2*\sqrt{5}} = 1 = F_1$ (given in Q)

Since the statement holds true for the base case, we shall proceed with the next step

Step 2: (Induction Hypothesis) Assuming the statement is true for n

$$Assume\ F(n) = \frac{a^n - b^n}{\sqrt{5}}\ where\ a = \frac{1+\sqrt{5}}{2}\ and\ b = \frac{1-\sqrt{5}}{2}\ is\ true$$

Step 3: (Induction Step): Need to prove that the above statement is also true for the value $n = n + 1$

As per the question F(n+1) = F(n) + F(n-1)

$$F(n) = \frac{a^n - b^n}{\sqrt{5}} and\ F(n-1) = \frac{a^{n-1} - b^{n-1}}{\sqrt{5}}$$

$$F(n+1) = \frac{a^n - b^n}{\sqrt{5}} + \frac{a^{n-1} - b^{n-1}}{\sqrt{5}}$$

$$= \frac{a^n - b^n + a^{n-1} - b^{n-1}}{\sqrt{5}} = \frac{a^n + a^{n-1} - b^n - b^{n-1}}{\sqrt{5}} = \frac{a^{n+1}(\frac{1}{a^2} + \frac{1}{a}) - b^n(\frac{1}{b^2} + \frac{1}{b})}{\sqrt{5}}$$

By solving $\left(\frac{1}{a^2} + \frac{1}{a}\right)$ by substituting the value of a $= \frac{1+\sqrt{5}}{2}$ $we\ get$ $\left(\frac{1}{a^2} + \frac{1}{a}\right) = \frac{8+4\sqrt{5}}{8+4\sqrt{5}} = 1$

$and\ by\ solving$ $\left(\frac{1}{b^2} + \frac{1}{b}\right)$ by substituting the value of b = $\frac{1-\sqrt{5}}{2}$ we get $\left(\frac{1}{b^2} + \frac{1}{b}\right) = \frac{8-4\sqrt{5}}{8-4\sqrt{5}} = 1$

So now, by substituting $\left(\frac{1}{a^2} + \frac{1}{a}\right)$ with 1 and $\left(\frac{1}{b^2} + \frac{1}{b}\right)$ with 1 we will get

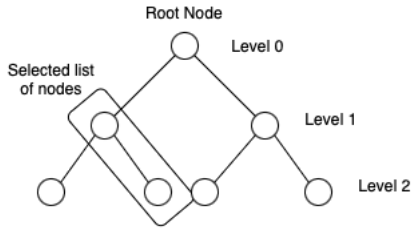
$$= \frac{a^{n+1}(1) - b^{n+1}(1)}{\sqrt{5}} = \frac{a^{n+1} - b^{n+1}}{\sqrt{5}}$$

$$F(n+1) = \frac{a^{n+1} - b^{n+1}}{\sqrt{5}}$$

Conclusion: Hence by mathematical induction we have proved that the above statement is true for all values of n.

3Q) Let T be a breadth-first search tree of a connected graph G. Let (x, y) be an edge of G, where x and y are nodes in G. Prove by contradiction that in T the level of x and the level of y differ by at most 1.

Solution:



Let G be a connected graph.

As per the steps of contradiction.

Step 1: Assume a statement that is contradicting the given statement: The level of x and level of y differ more than 1

In a Breadth First Search, it searches the nodes along the breadth and uses a queue to store the traverse all the nodes of a graph.

As per the Breadth First Search technique, for a node to be discovered, it needs to be a neighboring node (in the same level) which means that the length is '0'.

Else, it should be one of its respective child nodes, which means that the length is 1 (differ by level of 1)

Since Breadth First Search traverses the tree/graph breadth wise first, for a node to be discovered in first place, it should either be a neighboring node or child node so the length or the difference in levels can be at most 1.

This is contradicting to the statement that we have assumed.

Hence by proof of contradiction, since the statement we assumed is not feasible. The given statement holds true that an edge **(x, y)** in a Graph and their level of x and the level of y in the BFS of the graph differ by at most 1.

4Q) We have a connected graph G = (V, E), and a specific vertex u ∈ V. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth- first search tree rooted at u, and obtain the same tree T. Prove by contradiction that G has the same structure as T, that is, G cannot contain any edges that do not belong to T.

Solution:

Given that for a graph G = (V, E) the BFS tree computed at node u is the same as DFS tree computed at node same u (Tree T).

As per the given statement, considering Breadth First Search of a graph is same as the Depth First search of the graph, there are a very limited set of possibilities which can be fit for this condition.

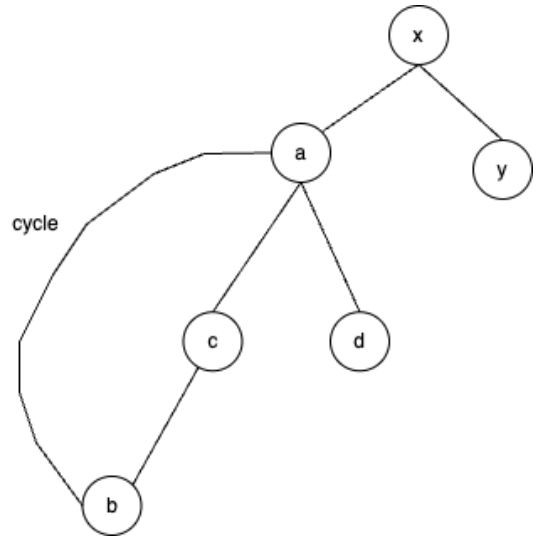
1. A graph in which the root node has exactly two children
2. A graph in which every node has only one child
3. A Graph should not contain cycles, if the graph contains a cycle, then the BFS and DFS cannot be the same, since because of the cycle, the way of computation of BFS and DFS will defer and will never result the same.

Proof by Contradiction:

Consider an edge (a, b) that exists in Graph G, but does not belong to the Tree T.

i.e., $G \neq T$

Now if we compute the DFS tree for the given graph G, such that (a, b) are having an edge is possible only if either a or b is a parent node of the other or in other case, if they are having a cycle between them as shown below.



Also, in the case of a BFS tree, for two nodes to be connected, they should be either at the same level or must have a maximum level difference of 1. Else if the level difference between the two vertices is more than 1, then they might not be discovered using BFS.

In the above assumed graph, both the vertices (a, b) are connected through a direct edge, so the level difference between them is 1.

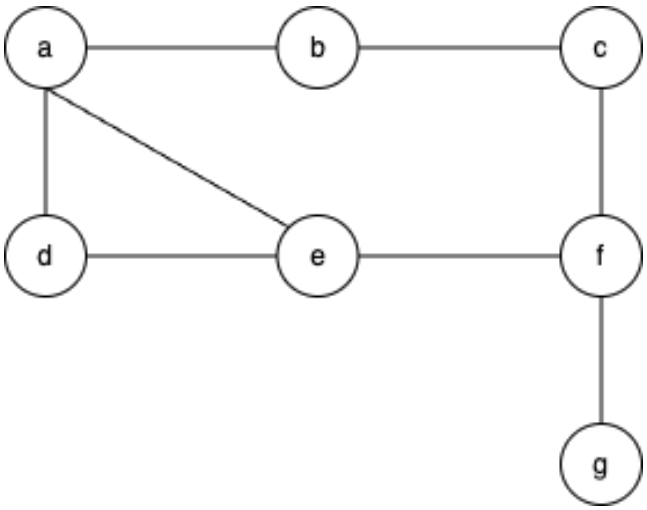
As per the given statement, since both the BFS and the DFS of the given graph G are the same. It states that edge (a, b) is existing in the Graph G will also exist in the tree T.

So, this is contradicting the given statement.

Hence by contradiction, we proved that, for a graph G, if the BFS and the DFS computed from the same node u are equal to T, it implies that graph G and the tree T has the same structure. That is graph G doesn't contain any edges that doesn't belong to T.

5Q) You have been given an unweighted, undirected, and connected graph G = (V, E). Device an algorithm to determine maximum of the shortest paths' lengths between all pairs of nodes in graph G (Also called diameter of the graph). Also, determine the time complexity of your algorithm and justify your answer.

Solution:



We need to find the longest (maximum) of the shortest paths among all the nodes of the graph.

Algorithm:

Step 1 (TC – O(1)): let n be the number of nodes in the graph

Step 2 (TC – O(1)): Consider/Initialize a visited array V of length n and initialize all values in it to '0'

Step 3 (TC – O(1)): Initialize a Two-dimensional array Distance[n][n] and initialize all the values in it to '0'

Step 4 (TC – $O(n^2)$): For every node 'x' in the graph (building an Adjacency matrix)

> Step 4.a: Compute the distance to its adjacent nodes, here in this case it shall be '1' to the neighboring nodes and '0' to the self-node.

After completing this step, the below will be the adjacency matrix for the above graph:

	a	**b**	**c**	**d**	**e**	**f**	**g**
a	0	1	0	1	1	0	0
b	1	0	1	0	0	0	0
c	0	1	0	0	0	1	0
d	1	0	0	0	1	0	0
e	1	0	0	1	0	1	0
f	0	0	1	0	1	0	1
g	0	0	0	0	0	1	0

Step 5 (TC – $O(n^2)$): For every node 'x' in in the graph, compute the distances to the other nodes basing on the adjacency matrix that we have constructed in the previous step

Step 5.a: Consider the source node to be 'x'

Step 5.b: Calculate the distance from 'x' to all other nodes in the graph using the below steps

For all the nodes 'y' in the given graph:

If the other node is not the source node itself, and it is not an adjacent node for which the distance was already computed, then:

Using the adjacency matrix compute the shortest distance to the destination node from the source node.

Step 5.b.1: Start from the destination node 'y'. find to which adjacent nodes it is connected, then for every adjacent node, check the adjacent nodes until we arrive at the root node.

For example, if we need to find the distance from 'a' to 'g'. using the adjacency matrix, we already know the distance between 'g' to 'f', 'f' to 'e' and 'e' to 'a'. So, the sum of these distance will be the shortest distance between 'a' to 'g'.

Step 6 (TC – O(1)): Initialize a max variable to '0'.

Step 7 (TC – $O(n^2)$): Once all the shortest distances between any two nodes in the given graph are updated in the two-dimensional matrix that we have constructed, traverse the entire matrix to find the maximum value.

If the value found during the matrix traversal is greater than the max variable, then update the value in the variable.

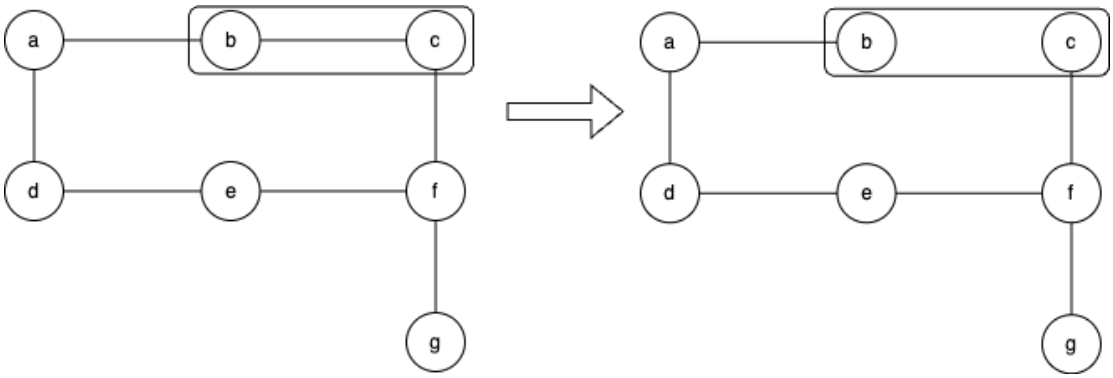
Step 8 (TC – O(1)): Once the maximum value from the matrix is found out. That maximum value is the diameter of the given graph G.

The Time complexity for this algorithm shall be $O(V^2)$, where V is the number of vertices in the given graph

6Q) You have been given an unweighted and undirected graph G = (V, E), and an edge e ∈ E. a. Your task is to devise an algorithm to determine whether the graph G has a cycle containing that specific given edge e. Also, determine the time complexity of your algorithm and justify your answer. Note: To become eligible for full credits on this problem, the running time of your algorithm should be bounded by O (V + E).

b. Will your algorithm still works if the graph is unweighted but directed? If not, how would you modify your algorithm to make it work on an unweighted-directed graph.

Solution:



6.a Solution:

Step 1 (TC – O(1)): Consider a graph G with V vertices and E edges, G(V, E) that contains a cycle.

Step 2 (TC – O(1)) : Consider an edge 'e' between nodes (b, c) in the given Graph G(V, E)

Step 3 (TC – O(1)): Delete or remove the edge 'e (b, c)' from the graph G(V, E)

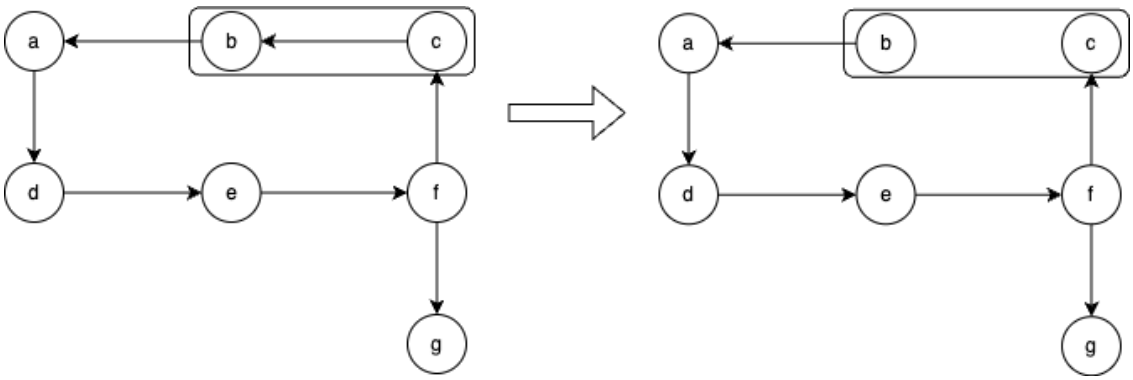
Step 4 (TC – O(V+E)): Apply Depth First Search(DFS) from either node 'b' or node 'c'.

> Step 4.1: **Case 1:** If the DFS is applied from node 'b'. And the DFS tree reaches the node 'c' even after deleting the direct edge between them, then it means that there is a cycle in the graph and the edge 'e' between the nodes (b, c) is a part of the cycle.
>
> **Case 2:** If the DFS is applied from node 'c'. And the DFS tree reaches the node 'b' even after deleting the direct edge between them, then it means that there is a cycle in the graph and the edge 'e' between the nodes (b, c) is a part of the cycle.

Time complexity of the above solution: O(1) + O(1) + O(1) + O(V + E), considering the maximum value of the growth rates and since O(1) is very small compared to O(V + E), the worst-case time complexity of the algorithm shall be **O(V + E)**

6.b Solution:

The above algorithm shall work for the directed and un-weighted graph as well, but the only change shall be that the Graph traversal algorithm(DFS) shall be applied only from the child node. If the Graph traversal is applied from the parent node, then the algorithm may not work.



Step 1 (TC – O(1)): Assume the Graph G with V vertices and E edges is a directed Graph and has a cycle.

Step 2 (TC – O(1)): Consider the edge ‘e’ between nodes ‘c’ and ‘b’ directed from one node to the other.

Step 3 (TC – O(1)): Delete the directed edge ‘e(c, b)’ in the graph G(V, E)

Step 4 (TC – O(V + E)): Apply Depth First Search(DFS) from the outward node of the edge. (In the above example apply DFS from node ‘b’). If the DFS can reach the node ‘c’ even after deleting the directed edge between them, then it means there is a cycle in the graph and the edge ‘e’ is a part of it.

> If the DFS is applied from the parent node (in the above example, from node ‘c’, since there are no nodes and directed edges that are outward from node ‘c’. the DFS will return NULL.

Time complexity of the above solution: O(1) + O(1) + O(1) + O(V + E), considering the maximum value of the growth rates and since O(1) is very small compared to O(V + E), the worst-case time complexity of the algorithm shall be **O(V + E)**

7Q) Use prove by induction to show that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

Solution:

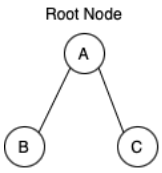


Figure 1

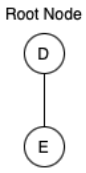


Figure 2

As per the Induction Hypothesis, we need to prove for Base case:

Step 1: Base Case: Prove that the above statement is true for a base Binary Tree.

As per the definition of a binary tree, A tree in which ever node has a maximum of 2 children. If every node in the tree has either ‘0’ or ‘2’ children, then it is called a Full Binary Tree.

Case 1: Consider Figure 1:

The Number of Nodes with Two children = ‘1’ (Node A)

The number of leaf nodes is = ‘2’ (Node B and Node C)

Therefore, the above statement: The nodes with exactly two children is 1 more than the leaf nodes, holds true.

Case 2: Consider Figure 2:

The Number of Nodes with Two children = ‘0’

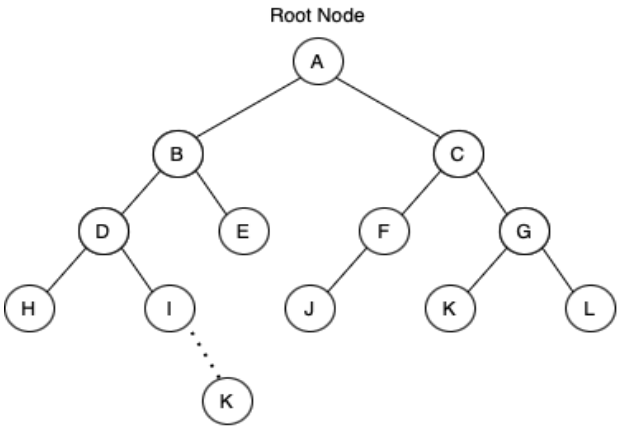
The number of leaf nodes is = ‘1’ (Node E)

Therefore, the given statement: The nodes with exactly two children is 1 more than the leaf nodes, holds true.

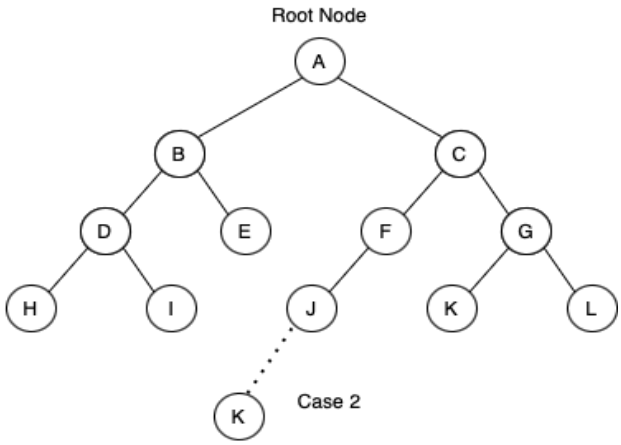
Step 2: Induction hypothesis: In this step we assume that the given statement is true for a Binary tree having N nodes, K nodes that have exactly two children and L leaf nodes this statement holds true.

Therefore, we derive that $\boldsymbol{K = L - 1}$

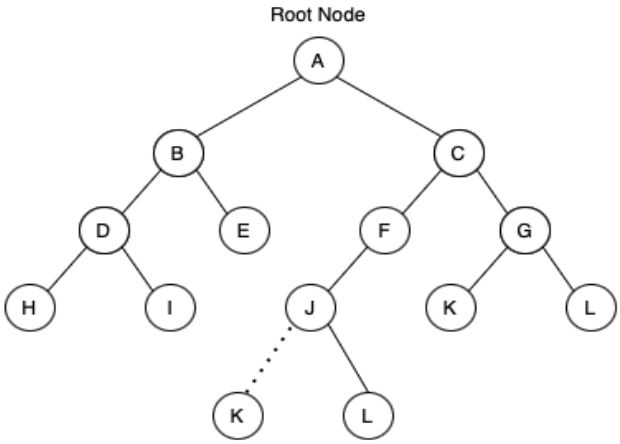
Step 3: Induction Step: In this step we assume the above statement is true for a binary tree with n nodes and prove that the above statement also holds true for the binary tree with N+1 nodes.



Case 1



Case 2



Case 3

Consider a Binary tree with N nodes and the above statement holds true

Let N be the total number of nodes, K be the total number of nodes with two children and L be the total number of leaf nodes.

Case 1: (Please refer the above Figure): Adding a node to a leaf node whose parent has two children

In this case there is no change in the number of leaf nodes and there is no change in the number of nodes with exactly two children.

Since there is no change, **the above statement holds TRUE for this case. i.e.,** $\boldsymbol{K = L - 1}$

Case 2: (Please refer the above figure): Adding a node to a leaf node whose parent has exactly one child

Even In this case, there is no change in the number of leaf nodes as well as the number of nodes with exactly two children.

Since there is no change, **the above statement holds TRUE for this case. i.e., $K = L - 1$**

Case 3: (Please refer the above figure): Adding a node to a node that is already having 1 child.

Here, the number of leaf nodes increases by 1. As well as the number of nodes with exactly two children also increases by 1

i.e., $K' = K + 1$ and $L' = L + 1$

now substituting them in the above equation we get $K + 1 = L - 1 + 1$

=> $K = L - 1 + 1 - 1$

=> $K = L - 1$

So even for this case, the above statement holds TRUE

Hence, as per the Mathematical induction hypothesis we have proved that the above statement that the number of nodes with exactly two children is one less than the number of leaf nodes.

8Q) This problem is divided into following subsections:
a. Consider a sequence of numbers: 4, 13, 7, 15, 21, 24,10 Construct a binomial min-heap H1 by reading the above numbers from left to right. Draw all the intermediate binomial heaps as well as the final binomial heap H1. Illustrate your work clearly and concisely.

b. Repeat a. for another sequence of numbers: 11,12,21,24,18,13,16 to construct another binomial min-heapH2.
c. Merge H1 and H2 to construct the final binomial heap H. Draw all the intermediate binomial heaps (while merging H1 and H2) as well as the final binomial heap H.

Solution:

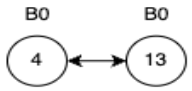
For constructing the binomial min heap, we need to construct the binomial trees as per the formula $B_k = B_{k-1} + B_{k-1}$

Following the above formula, we shall be joining the trees when two trees of the same rank. And then join them with the rest of the heap.

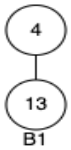
Below are the illustrations for constructions of the binomial min heaps H1 (**8.a Solution**) and H2 (**8.b solution**) and them merging them to form a final heap (**8.c solution**).

Step 1: Insert 4 (B0) 

Step 2: Insert 13 (B0) 

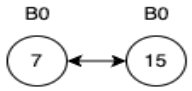


Step 3: Merge both the B0 trees to form a B1, as shown beside

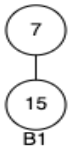


Step 4: Insert 7 (B0) 

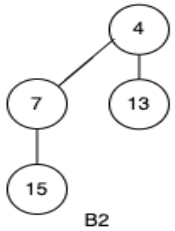
Step 5: Insert 15 (B0) 



Step 6: Merge both the B0 trees to form a B1, as shown beside

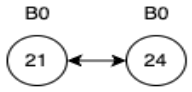


Step 7: Merge both the B1 trees to form a B2, as shown beside

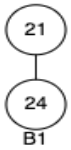


Step 8: Insert 21 (B0) 

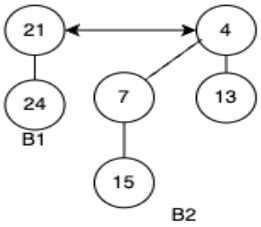
Step 9: Insert 24 (B0) 



Step 10: Merge both the B0 trees to form a B1, as shown beside

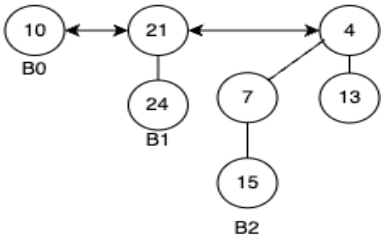


Step 11: Join both the B1 tree from step 10 along with B2 from step 7



Step 12: Insert 10 (B0) 

Step 13: Join both the B0 tree from step 12 along with heap from step 11

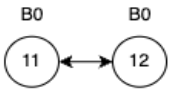


Heap H1

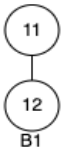
Solution for 8.a

Step 1: Insert 11 (B0) 11

Step 2: Insert 12 (B0) 12

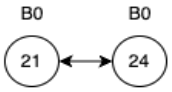


Step 3: Merge both the B0 trees to form a B1, as shown beside

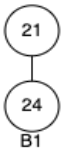


Step 4: Insert 21 (B0) 21

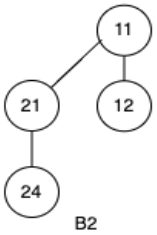
Step 5: Insert 24 (B0) 24



Step 6: Merge both the B0 trees to form a B1, as shown beside

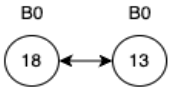


Step 7: Merge both the B1 trees to form a B2, as shown beside

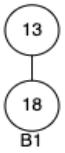


Step 8: Insert 18 (B0) 18

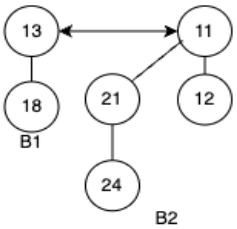
Step 9: Insert 13 (B0) 13



Step 10: Merge both the B0 trees to form a B1, as shown beside

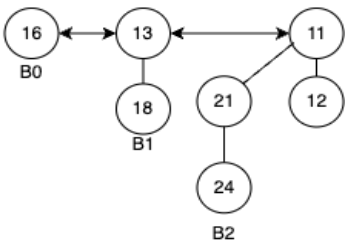


Step 11: Join both the B1 tree from step 10 along with B2 from step 7



Step 12: Insert 16 (B0) 16

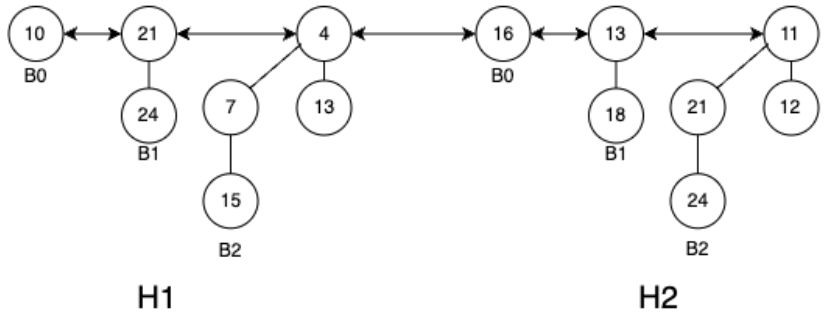
Step 13: Join both the B0 tree from step 12 along with heap from step 11



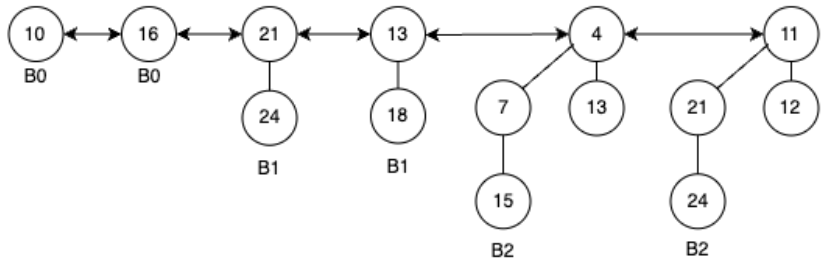
Heap H2

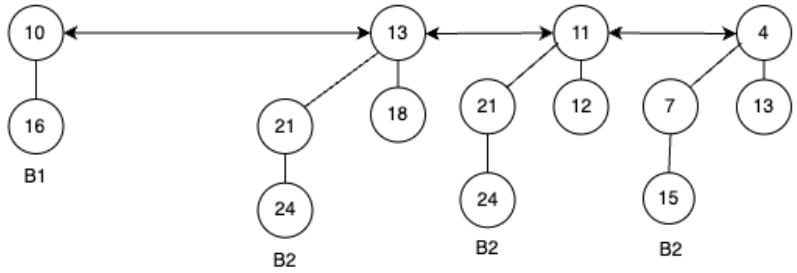
Solution for 8.b

Consider both the heaps H1 and H2 to merge

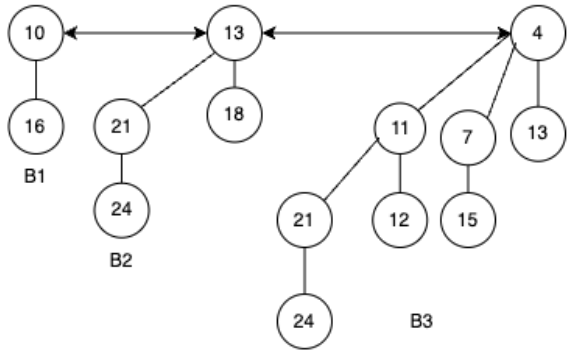


Merge or join the binomial trees with the same degree. i.e., join both the B0 trees to form a B1 tree and join both the B1 trees to form a B2 tree





Merge or join the binomial trees with the same degree. i.e., join the last two B2 trees to form a B3 tree



Merged Heap

Solution for 8.c

Therefore, this is the above Binomial Heap, that is formed by merging the Binomial Heaps H1 and H2.

9Q) Consider an array on which the following operations can be performed: addLast(e): An element 'e' is added at the end of the array. deleteEveryThird (): Removes every third element in the list i.e., removes the first, fourth, eighth, etc., elements of the array. You may assume that addLast(e) has a cost 1, and deleteEveryThird () has a cost equal to the number of elements in the list. What is the amortized cost of addLast and deleteEveryThird operations? Consider the worst sequence of operations. Justify your answer with proper explanation using accounting method.

Solution:

As per the accounting method, the cost given for addLast function and deleteEveryThird function is 1 and the number of elements in the array respectively.

Considering the worst sequence of operations and assuming that deleteEveryThird function cannot be executed when there are no elements in the considered array.

Try 1: The amortized cost that is assumed is:

addLast() – 2

deleteEveryThird() – 2

Operation	**Amortized Cost**	**Actual Cost**	**Aggregated Remaining/Cr in Bank**
AddLast – 1	2	1	1
AddLast – 2	2	1	2
AddLast – 3	2	1	3
AddLast – 4	2	1	4
AddLast – 5	2	1	5
AddLast – 6	2	1	6
AddLast – 7	2	1	7
AddLast – 8	2	1	8
AddLast – 9	2	1	9
AddLast – 10	2	1	10
DeleteEveryThird	2	10	2
DeleteEveryThird	2	6	-Out of Credits
DeleteEveryThird	--	--	--

So, for the above assumed amortized cost for both the functions cannot be achieved.

Try 2: So, assuming a new set of amortized costs for the functions as below,

addLast – 3

deleteEveryThird – 0

Operation	Amortized Cost	Actual Cost	Aggregated Remaining/Cr in Bank
AddLast – 1	3	1	2
AddLast – 2	3	1	4
AddLast – 3	3	1	6
AddLast – 4	3	1	8
AddLast – 5	3	1	10
AddLast – 6	3	1	12
AddLast – 7	3	1	14
AddLast – 8	3	1	16
AddLast – 9	3	1	18
AddLast – 10	3	1	20
DeleteEveryThird	0	10	10
DeleteEveryThird	0	6	4
DeleteEveryThird	0	4	0
DeleteEveryThird	0	2	-Out of Credits
DeleteEveryThird	--	--	--

So even with the above considered values of amortized cost, the above functions/problem is not feasible.

Try 3: So, assuming the following values for the amortized cost

addLast – 3

deleteEveryThird – 1

Operation	Amortized Cost	Actual Cost	Aggregated Remaining/Cr in Bank
AddLast – 1	3	1	2
AddLast – 2	3	1	4
AddLast – 3	3	1	6
AddLast – 4	3	1	8
AddLast – 5	3	1	10
AddLast – 6	3	1	12
AddLast – 7	3	1	14
AddLast – 8	3	1	16
AddLast – 9	3	1	18
AddLast – 10	3	1	20
DeleteEveryThird	1	10	11
DeleteEveryThird	1	6	6
DeleteEveryThird	1	4	3
DeleteEveryThird	1	2	2

DeleteEveryThird	1	1	2
AddLast – 11	3	1	4
AddLast – 12	3	1	6
AddLast – 13	3	1	8
DeleteEveryThird	1	3	6
DeleteEveryThird	1	2	5
DeleteEveryThird	1	1	5(positive credits)

Now let's consider another sequence of operations with the above amortized cost, and check if it is working

Operation	**Amortized Cost**	**Actual Cost**	**Aggregated Remaining/Cr in Bank**
AddLast – 1	3	1	2
AddLast – 2	3	1	4
AddLast – 3	3	1	6
AddLast – 4	3	1	8
AddLast – 5	3	1	10
AddLast – 6	3	1	12
AddLast – 7	3	1	14
AddLast – 8	3	1	16
AddLast – 9	3	1	18
AddLast – 10	3	1	20
AddLast – 11	3	1	22
AddLast – 12	3	1	24
AddLast – 13	3	1	26
AddLast – 14	3	1	28
AddLast – 15	3	1	30
AddLast – 16	3	1	32
AddLast – 17	3	1	34
AddLast – 18	3	1	36
AddLast – 19	3	1	38
AddLast – 20	3	1	40
DeleteEveryThird	1	20	21
DeleteEveryThird	1	13	9
DeleteEveryThird	1	8	2
DeleteEveryThird	1	5	-Out of Credits

So, with this sequence of operations, we are not able to achieve the sequence of operations.

Try 4: So, assuming the following values for the amortized cost

addLast – 4

deleteEveryThird – 1

Operation	Amortized Cost	Actual Cost	Aggregated Remaining/Cr in Bank
AddLast – 1	4	1	3
AddLast – 2	4	1	6
AddLast – 3	4	1	9
AddLast – 4	4	1	12
AddLast – 5	4	1	15
AddLast – 6	4	1	18
AddLast – 7	4	1	21
AddLast – 8	4	1	24
AddLast – 9	4	1	27
AddLast – 10	4	1	30
AddLast – 11	4	1	33
AddLast – 12	4	1	36
AddLast – 13	4	1	39
AddLast – 14	4	1	42
AddLast – 15	4	1	45
AddLast – 16	4	1	48
AddLast – 17	4	1	51
AddLast – 18	4	1	54
AddLast – 19	4	1	57
AddLast – 20	4	1	60
AddLast – 21	4	1	63
AddLast – 22	4	1	66
AddLast – 23	4	1	69
AddLast – 24	4	1	72
AddLast – 25	4	1	75
DeleteEveryThird	1	25	51
DeleteEveryThird	1	16	36
DeleteEveryThird	1	10	27
DeleteEveryThird	1	6	22
DeleteEveryThird	1	4	19
DeleteEveryThird	1	2	18
DeleteEveryThird	1	1	18 (positive credits)

So, with this current amortized cost, even after the following operations and the array is empty, we still have some credits left in the bank. So, this is a positive value of credits, and the problem is feasible.

Hence the Amortized Cost is as follows for the following functions:

addLast() function – 4

deleteEveryThird() function – 1

10Q) Given a sequence of n operations, suppose the i-th operation cost i + j if i = 2^j for some integer j; otherwise, the cost is 1. Prove that the amortized cost per operation is O(1).

Solution: The total amortized cost = $\frac{total\ cost}{number\ of\ operations}$

The below is the sequence of operations and their associated cost

i-th Operation	Respective Cost
1 operation	1
2 operation	3
3 operation	1
4 operation	6
5 operation	1
6 operation	1
7 operation	1
8 operation	11
9 operation	1
10 operation	1
11 operation	1
12 operation	1…….

The total cost of n operations is 1+3+1+6+1+1+11+1+1+1+……..

$$total\ cost = 1 + 3 + 1 + 6 + 1 + 1 + 11 + 1 + 1 + 1 + \cdots \ldots \ldots \ldots$$

The number of '1' terms are $\boldsymbol{\left(n - log(n)\right) * 1}$

The sum of the remaining terms is $3 + 6 + 11 + 20 + \cdots$ which is in the format of Arithmetic Geometric Progression of $\boldsymbol{2^k + k}$

Now, $Amortized\ Cost = \frac{Total\ cost}{Total\ number\ of\ operations}$ = $\frac{n - \log(n) + \sum_{k=1}^{\log(n)} (2^k + k)}{n}$

= $\frac{n - \log(n) + \sum_{k=1}^{\log(n)} (2^k) + \sum_{k=1}^{\log(n)} (k)}{n}$

$\sum_{k=1}^{\log(n)} (2^k)$ is a Geometric progression, so by using the formula of the sum of terms in GP, we get.

$$\sum_{k=1}^{\log(n)} (2^k) = \frac{2(2^{\log(n)} - 1)}{(2-1)} = \frac{2(n-1)}{(1)} = 2n - 2$$

$\sum_{k=1}^{\log(n)}(k)$is an Arithmetic progression, so by using the formula of the sum of terms in AP, we get.

$$\sum_{k=1}^{\log(n)} (k) = \frac{\log(n)}{2}(1 + \log(n)) = \frac{\log(n) + (\log(n))^2}{2}$$

Now calculating the Amortized cost,

$= \frac{n - \log(n) + \sum_{k=1}^{\log(n)}(2^k) + \sum_{k=1}^{\log(n)}(k)}{n}$

And replacing the values, we get

$= \frac{n - \log(n) + 2n - 2 + \frac{\log(n) + (\log(n))^2}{2}}{n}$

$= \frac{3n - 2 - \log(n) + \frac{\log(n)}{2} + \frac{(\log(n))^2}{2}}{n}$

$= \frac{3n - 2 - \frac{\log(n)}{2} + \frac{(\log(n))^2}{2}}{n}$

Since the values of $\frac{\log(n)}{2}$ and $\frac{(\log(n))^2}{2}$ are very less when compared to the value of **n**, we can ignore them, resulting in

$= \frac{O(3n-2)}{n}$ and since 2 is a constant value, comparing it to the value of **n.** we can ignore it.

$= \frac{O(3n)}{n}$ = **O(1).**

Therefore, the Amortized Cost per operation is O(1).