CSCI 570 Analysis of Algorithm Homework 1

Name: Aagam Shah
USC ID: 5420023430

1. 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)

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

Answer 1:

Simplifying some of the function f(n)

1. $2^{\log_2 n}$
   Applying the property $a^{\log_a b} = b$
   Thus $2^{\log_2 n} = n$

2. $n^{1/\log_2 n}$
   Using change of base theorem: $\log_b a = \frac{\log_x a}{\log_x b}$ we get $\frac{1}{\log_2 n} = \frac{\log_2 2}{\log_2 n} = \log_n 2$
   Applying the property $a^{\log_a b} = b$

   $n^{\frac{1}{\log_2 n}} = n^{\log_n 2} = 2$

Comparing the functions having log in them

1. **Comparing $log^2 n$ vs $\log_2 n^{\sqrt{\log_2 n}}$**

Let's compare function $log^2 n$ with $\log_2 n^{\sqrt{\log_2 n}}$, taking log on both the sides we get

1. $log(log^2 n) = 2\, log(\log n)$
2. $log\left(log\, n^{\sqrt{\log_2 n}}\right) = \sqrt{\log_2 n} * log(log\, n)$

When we compare the $2\, log(\log n)$ with $\sqrt{\log_2 n} * log(log\, n)$ we know that $\sqrt{\log_2 n} > 2$ hence we can say that

$$\log_2 n^{\sqrt{\log_2 n}} > log^2 n$$

Comparing functions having n in them

Consider the f(n) = $2^{\log_2 n}$, Applying the property $a^{\log_a b} = b$, Thus $2^{\log_2 n} = n$

2. **Comparing n^{logn} vs $2^{\log_2 n}$**

Let's compare n^{logn} and $2^{\log_2 n} = n$ taking log on both sides we get

1. $\log(n^{logn}) = \log n * \log n$
2. $\log(n) = logn$

When we compare $\log n * \log n$ vs $\log n$ we know that $\log n * \log n > \log n$ $thus$

$$n^{logn} > 2^{\log_2 n}$$

3. **Comparing n^{logn} vs $\sqrt{n}^{\sqrt{n}}$**

Let's compare n^{logn} and $\sqrt{n}^{\sqrt{n}}$ taking log on both sides we get

1. $\log(n^{logn}) = \log n * \log n$
2. $\log \sqrt{n}^{\sqrt{n}} = \sqrt{n} * \log \sqrt{n} = \sqrt{n} * \log n^{\frac{1}{2}} = \frac{\sqrt{n}}{2} * \log n$

4. **Comparing log (n^{logn}) vs log $\sqrt{n}^{\sqrt{n}}$**

When we compare log (n^{logn}) vs log $\sqrt{n}^{\sqrt{n}}$ we can see that since $\frac{\sqrt{n}}{2} > \log n$ thus

$$\sqrt{n}^{\sqrt{n}} > n^{logn}$$

Subsequence of the function containing n

$$2^{log\, n} < n^{\log n} < \sqrt{n}^{\sqrt{n}}$$

5. **Comparing $n^{\log_2 n}$ vs $n * \log(logn)$**

Let's compare the function $n^{\log_2 n}$ $with$ $n * \log(logn)$, taking log on both sides we get

1. $\log\left(n^{\log_2 n}\right) = \log(n^{log_2 n}) = \log n * \log n$
2. $\log(n * \log(\log n)) = \log n + \log(\log(logn)$

When we compare $\log n * \log n$ vs $\log n + \log(\log(\log n)$ here $\log(\log(logn)$ has a very small value hence we consider $\log n * \log n$ vs $\log n$ and we know that $\log n * \log n > \log n$ $thus$

$$n * \log(logn) < n^{\log_2 n}$$

6. **Comparing $2^{\log_2 n}$ vs $n * \log(logn)$**

As we know that $2^{\log_2 n} = n$ < n * log (log n) thus subsequence created is

$$2^{\log_2 n} < n * \log_2 \log_2 n < n^{\log_2 n}$$

As we know that Constant take the lowest place in growth place then followed by log n and then followed by n

Constant < Logarithm < Polynomial < Exponential

Final Sequence formed in the increasing order of the growth rate is given as follows

$$n^{\frac{1}{\log_2 n}} < log^2 n < \log_2 n^{\sqrt{\log_2 n}} < 2^{\log_2 n} < n * \log_2 \log_2 n < n^{\log_2 n} < \sqrt{n}^{\sqrt{n}}$$

2. The Fibonacci sequence $F_1, F_2, F_3 \ldots$, is defined as follows. $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$. For example, $F_3 = F_1 + F_2 = 2$ and $F_4 = F_3 + F_2 = 3$. Prove by induction that $F_n = \frac{a^n - b^n}{\sqrt{5}}$, where $a = \frac{1+\sqrt{5}}{2}$ and $b = \frac{1-\sqrt{5}}{2}$.

Answer 2:

Base case

Let P(k) be the predicate

$F_k = \frac{a^k - b^k}{\sqrt{5}} \quad \forall k \geq 3$

Consider $F_3 = F_2 + F_1$ as we know the value of $F_1 = F_2 = 1$ thus we calculate the value of F_3

$F_n = \frac{a^n - b^n}{\sqrt{5}}$ i.e. $F_3 = \frac{a^3 - b^3}{\sqrt{5}} = \frac{(a-b)*(a^2 + b^2 - ab)}{\sqrt{5}} =$

$$\frac{\left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right) * \left(\left(\frac{1+\sqrt{5}}{2}\right)^2 + \left(\frac{1-\sqrt{5}}{2}\right)^2 - \left(\frac{1+\sqrt{5}}{2}\right)\left(\frac{1-\sqrt{5}}{2}\right)\right)}{\sqrt{5}}$$

$$\frac{\sqrt{5} * \left(\left(\frac{3+\sqrt{5}}{2}\right) + \left(\frac{3-\sqrt{5}}{2}\right) - 1\right)}{\sqrt{5}} = \frac{6}{2} - 1 = 2$$

$F_3 = F_2 + F_1$ *holds true for the base case considering* $n = 3$

Inductive Hypothesis

Assume that F_k holds true for $3 \leq k < n$

consider k = n-1

$F_{n-1} = \frac{a^{n-1} - b^{n-1}}{\sqrt{5}}$

Inductive Step

we need to prove that F_k is also true where k = n

Proof for Inductive Step

$$F_n = F_{n-1} + F_{n-2}$$

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

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

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

$$F_n = \frac{1}{\sqrt{5}} \left(a^{n-1} \left(\frac{a+1}{a}\right) - \left(b^{n-1} \left(\frac{b+1}{b}\right)\right)\right) \quad \text{--- Eq}$$

Let's calculate

$$a + 1 = \frac{1 + \sqrt{5}}{2} + 1 = \frac{1 + \sqrt{5} + 2}{2} = \frac{3 + \sqrt{5}}{2}$$

$$a^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{2(3+\sqrt{5})}{4} = \frac{3+\sqrt{5}}{2}$$

Thus we can see that

$a + 1 = a^2$ ----- eq 1

$$b + 1 = \frac{1 - \sqrt{5}}{2} + 1 = \frac{1 - \sqrt{5} + 2}{2} = \frac{3 - \sqrt{5}}{2}$$

$$b^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{2(3-\sqrt{5})}{4} = \frac{3-\sqrt{5}}{2}$$

Thus we can see that

$b + 1 = b^2$ ---- eq 2

Applying Eq1 and Eq 2 into Eq we get

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

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

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

Thus, we can see that the equation
$F_n = \frac{1}{\sqrt{5}}\left(a^n - b^n\right)$ holds true for all value of n where $n \leq 3$

3. 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.

Ans 3)

Proof by Contradiction

Contradictory Statement: **Assume that in the Breath-first search tree T of a connect graph G having nodes X and Y differ by level more than 1**

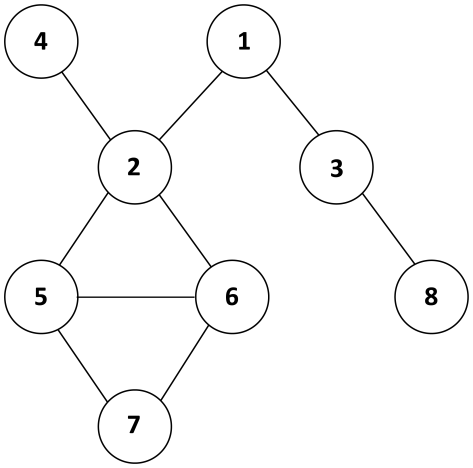
$$\boldsymbol{L_x - L_y \geq 2.}$$

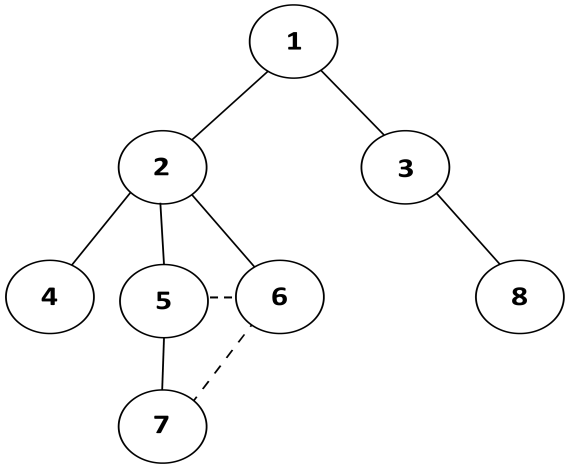
Let's consider the 3 cases which can occur in this scenario when we create a Graph with Node X and Node Y having a common Edge and traverse it using BFS to create a BFS Tree t

1. Node X and Node Y have a common Parent in the Tree T
2. Node X is the Child of Node Y
3. Node Y is the Child of Node X

Consider Case 1: Node X and Node Y have a common Parent in Tree T

Graph G Breath First Search Tree





When we run BFS on Graph G we get the Tree T as shown above

Consider the Node X and Node Y be Node having the number 5 and number 6

Node X = 5
Node Y = 6

When we create the BFS tree we can see they both come under the same parent i.e. Node 2.

Thus, we can see that the

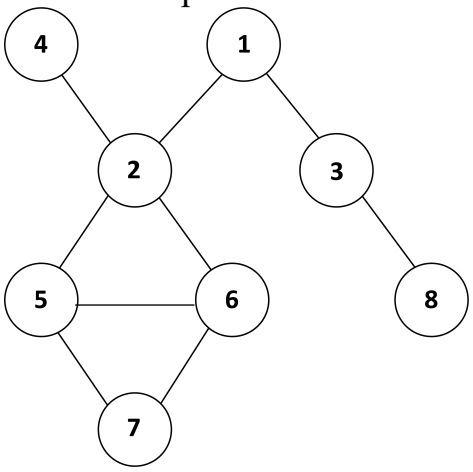
Node X and Node Y are in the same Level in the BFS Tree.

Consider the level of X be L_x and Level of Y be L_y thus we can see that

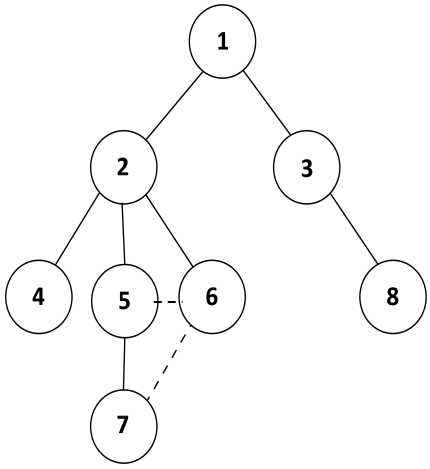
$$\boldsymbol{L_x - L_y = 0}$$

Consider Case 2: Node X is the Child of Node Y

Graph G



Breath First Search Tree



When we run BFS on Graph G we get the Tree T as shown above

Consider the Node X and Node Y be Node having the number 5 and number 6

Node X = 2
Node Y = 5

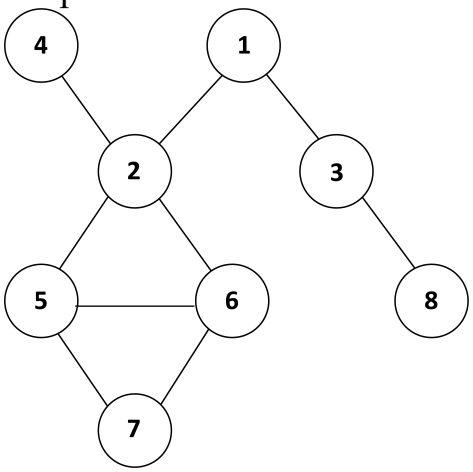
When we create the BFS tree we can see that Node 2 is the parent of Node 5.

Consider the level of X be L_x and Level of Y be L_y thus we can see that

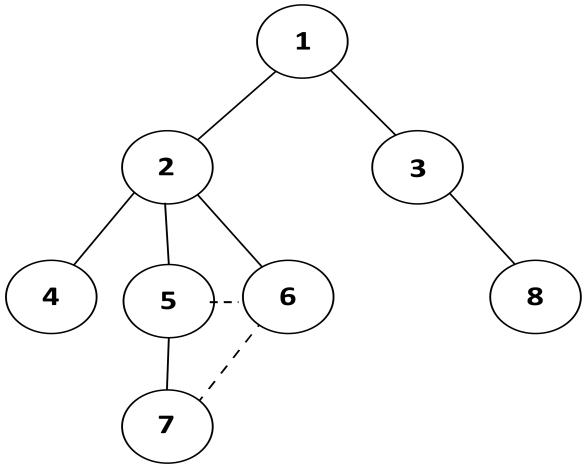
$$\boldsymbol{L_y - L_x = 1}$$

Consider Case 3: Node Y is the Child of Node X

Graph G



Breath First Search Tree



When we run BFS on Graph G we get the Tree T as shown above

Consider the Node X and Node Y be Node having the number 7 and number 6

Node X = 7
Node Y = 6

When we create the BFS tree we can see that Node 6 is the parent of Node 7.

Consider the level of X be L_x and Level of Y be L_y thus we can see that

$$\boldsymbol{L_x - L_y = 1}$$

As we can see that in none of the possible combination the difference of levels between Node X and Node Y of the Breath First Search tree differs by more than 1, As we can see here, we are contradicting the assumption we have taken earlier hence by contradiction we can say that T the level of x and the level of y differ by at most 1.

4. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. 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.

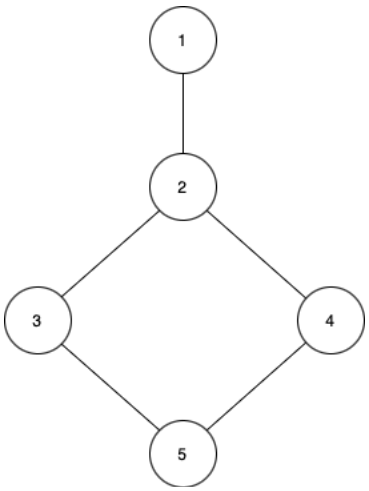
Ans 4)

Prove by Contradiction

Contradiction Statement: Graph G does not have the same structure as Tree T and G has edges that do not belong to T.

Consider a Graph G is undirected and connected but which has a cycle in it and then we create the BFS and DFS Tree

Graph G with Vertex V and Edges E



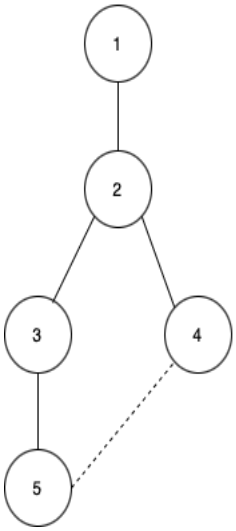
When we create the BFS sequence of this Tree

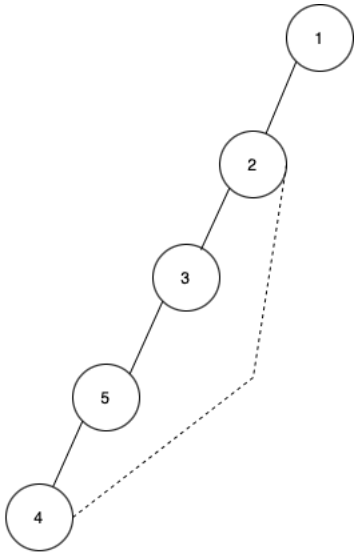
BFS Traversal Path: 1, 2, 3, 4, 5

DFS Traversal Path: 1, 2, 3, 5, 4

BFS Traversal Tree

DFS Traversal Tree





As we can see here that when we create a graph G which has an cycle here with Vertex 2, 3, 4 and 5. When we create the DFS Traversal for these Vertex we can see that all the vertices V1 to V5 come under a same path going in depth as the traversal suggest, while on the other hand when we see the BFS Traversal we can see that from the Vertex 2 the Tree branches into 2 branches since in BFS we explore all the neighboring vertex 2 as the parent and vertex 3 and 4 as their children.

There is the contradiction since we know that the Graph G should create Tree which has the same output for BFS and DFS and as we can see here that in this case the Tree created for BFS and DFS are different, as we are contradicting our assumption given above
Hence, we can say via Contradiction Graph G have the same structure as Tree T when BFS and DFS Traversals are same, and G cannot contain any edge that does not belong to T

5. 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.

Diameter of Graph: We calculate the diameter of the graph using a Breath First Search Approach, where using BFS Tree we can calculate the shortest path from a node. Here we traverse nodes in BFS pattern but one additional item we do is we add the distance from the Node X to Node Y. i.e., when we are traversing BFS the node of the predecessor is its ancestor +1. Using this logic and considering the root node of the BFS tree has distance 0 we can find the distance from source to all nodes. And running this algorithm for all Vertex gives us all distance from any vertex to another vertex. Using this and logic for calculating the diameter we can find the diameter of the graph G.

Algorithm:

Input: Graph G having V Vertex's and E Edge's
Output: Diameter of the Graph (Diameter)

1. For each vertex V in graph G
    a. Initialize Variable Diameter, set value Diameter = 0
    b. Diameter = Max (Diameter, runBFS (vertex v))
    c. Return Diameter

Function Find_Path_BFS (vertex start_vertex):

1. Initialize an Array Visited [] of size equal to number of vertex and value 'False'
2. Initialize a Queue Q
3. Initialize an Array Distance [] of size equal to number of vertex and value 0
4. Q.enqueue(start_vertex)
5. While (queue is not empty):
    a. Vertex v = queue.dequeue()
    b. For all Neighboring vertex neighbour_v of Vertex v
        i. If(visited[neighbour_v] is not False):
            1. Set Visited[neighbour_v] = True
            2. Set Distance[neighbour_v] = Distance[v] + 1
            3. Queue.enqueue(neighbour_v)
    c. Find the Max in Distance [] array
    d. Return the max number

Time Complexity:

Since we are using BFS in the underlying algorithm the runtime of BFS = O(V+E) where V = Vertex and E = Edge. Run time for BFS is O(V+E) because every vertex is enqueued once and processed once resulting in O(V) time and every edge is checked once to find neighboring nodes resulting in O(E) time hence net time is O(V+E) for BFS Traversal in a Graph G with V Vertex and E edges. And this is running for all Vertex V in the Graph to find the diameter of the graph the net time complexity becomes O (V * (V + E))

Runtime Complexity = O (V * (V + E)) where V = No of Vertex and E = No of Edges for the Graph G.

6. You have been given an unweighted and undirected graph G = (V, E), and an edge $e \in E$.
    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 work if the graph is unweighted but directed? If not, how would you modify your algorithm to make it work on an unweighted-directed graph.

Ans 6)

Let's consider the Graph G with Vertex V and Edges E is connected and undirected. Here we use DFS Algorithm to find whether there is cycle or not

Logic Explanation: Given a Graph G with V vertex and E edges. And an edge e(i,j) which represents the edge that we need to find if it creates the cycle or not. Here we remove the edge (i,j) and then do DFS Traversal from the vertex j. if by traversing in DFS we find the vertex i. then we can say that there exists another path from i to j and there will be a cycle in the graph g which has been created by that edge (i,j). conversely if the graph had been a tree or any graph without a cycle. We could have not been able to reach the vertex i from vertex j using DFS traversal.

a) Unweighted and Undirected Graph

Let's consider we have an Unweighted and Undirected Graph G with V Vertex and E Edges. Also, we do get the Edge e as e(i,j) which states that the Vertex i is connected to Vertex j in graph G.

Input: Graph G with V Vertex and E Edges and Edge e(i,j) which can create a cycle
Output: Print if the Graph has Cycle with Edge e

We consider that the Graph has been stored in the form of Sparse Matrix.

Algorithm:

1. Initialize a Graph G with V Vertex and E Edges and Edge e which needs to verified if they create a cycle or not
2. Remove the Edge e (i , j) from the Graph G
    a. Set the mat[i][j] = mat[j][i] = 0 in the matrix
3. Initialize an Array sequence [] as empty
4. Initialize an Array visited [] and set value as False
5. Set start_node as i
    a. Start_node = i
6. Sequence = RunDfs(start_node, sequence, visited)
7. If (sequence. contains(i) or sequence.contains(j))
    a. Print("There Exist a cycle")
8. Else
    a. Print("There is No Cycle")

RunDfs(Node node, sequence[],visited[]):

1. Set Visited[node] = True
2. Sequence.append(node)
3. Get all neighbouring nodes of Node node:
    a. If(!visited[neighbouring_node])
        i. RunDFS(neighbouring_node, sequence,visited)

This code has a time complexity of this algorithm is O(V+E) since we are running DFS Traversal from the node containing the Edge e.

Runtime complexity = O(V+E)

This code won't work for Directed and unweighted graph. We need to modify one item to make it work for all cases i.e. when we run the DFS we need to run from the vertex j of edge(i,j) since by doing this we will find the cycle in the Graph.

Algorithm:

1. Initialize a Graph G with V Vertex and E Edges and Edge e which needs to verify if they create a cycle or not
2. Remove the Edge e(i , j) from the Graph G
    a. Set the mat[i][j] = mat[j][i] = 0 in the matrix
3. Initialize an Array sequence [] as empty
4. Initialize an Array visited [] and set value as False
5. Set start_node as j vertex from the edge(i,j)
    a. Start_node = j
6. Sequence = RunDfs(start_node, sequence, visited)
7. If (sequence.contains(i) or sequence.contains(j))
    a. Print("There Exist a cycle")
8. Else
    a. Print("There is No Cycle")

RunDfs(Node node, sequence[],visited[]):

4. Set Visited[node] = True
5. Sequence.append(node)
6. Get all neighbouring nodes of Node node:
    a. If(!visited[neighbouring_node])
        i. RunDFS(neighbouring_node, sequence, visited)

This code has a time complexity of this algorithm m is O(V+E) since we are running DFS Traversal from the node containing the Edge e.

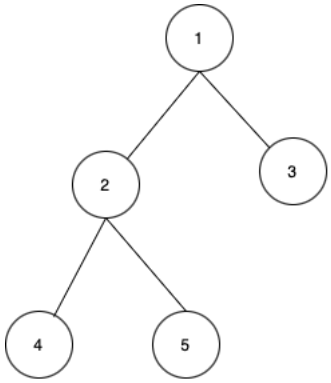
Runtime complexity = O(V+E)

7. 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.

Ans 7)

Proof by Induction on Number of Nodes with Two Children

Base Case:
Let us consider a Binary Tree T.



As we can see here that the number of Nodes with 2 Children are Node 1 having Children Node (2,3) and Node 2 having Children Node (4,5) and No of Leaf nodes: Node 3, Node 4 and Node 5.

No of Nodes having 2 Children: 2
No of Leaf Nodes: 3

$$\boldsymbol{N_C = N_L - 1}$$

Where -
$\boldsymbol{N_C}$: ***No of Nodes having 2 Children*** **and** $\boldsymbol{N_L}$: ***No of Leaf Nodes***

Thus, the base case holds true for $\boldsymbol{N_C}$ = 2

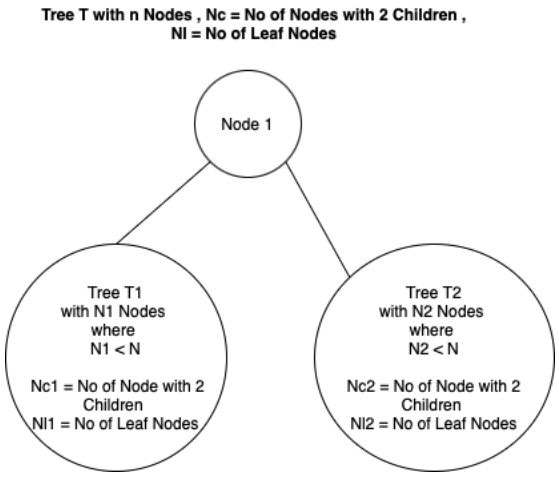
Induction Hypothesis:

Assume that $N_C = N_L - 1$ Where N_C : $No\ of\ Nodes\ having\ 2\ Children$ and N_L : $No\ of\ Leaf\ Nodes$ holds true for Tree having nodes with 2 children N_C < n

Inductive Step: Prove that $N_C = N_L - 1$ Where N_C : $No\ of\ Nodes\ having\ 2\ Children$ and N_L : $No\ of\ Leaf\ Nodes$ holds true for Tree having nodes N_C = n

Proof of Inductive Step:

Let's consider that we have 2 Tree T1, T2 both the trees have nodes that are N_C < n where n is the number of nodes in the Binary Tree T with 2 children. Thus from Inductive Hypothesis we can state that for all the Binary Tree T which have N_C < n the property $N_C = N_L - 1$ Where N_C: $No\ of\ Nodes\ having\ 2\ Children$ and N_L : $No\ of\ Leaf\ Nodes$ holds true.



Consider Tree T1

Number of Nodes in T1 = N1
Number of Nodes with 2 Children in N1 = N_{C1}
Number of Leaf Nodes in N1 = N_{L1}

Here as we can see that N_{C1}< n i.e., N1 does not contain all nodes that are in the Binary tree N with 2 children since there are few nodes in N2 that are not present in N1 so according to Inductive Hypothesis

$N_{C1} = N_{L1} - 1$ holds true for Tree T1 Where
N_{C1}: $No\ of\ Nodes\ having\ 2\ Children\ in\ Tree\ T1$ and N_L :
$No\ of\ Leaf\ Nodes\ in\ Tree\ T1$

Consider Tree T2

Number of Nodes in T2 = N2
Number of Nodes with 2 Children in N2 = N_{C2}
Number of Leaf Nodes in N2 = N_{L2}

Here as we can see that N2 < n i.e., N2 does not contain all nodes that are in the Binary tree N with 2 children since there are few nodes in N1 that are not present in N2 so according to Inductive Hypothesis

$N_{C2} = N_{L2} - 1$ holds true for Tree T2 Where
N_{C2}: $No\ of\ Nodes\ having\ 2\ Children\ in\ Tree\ T2$ and N_L :
$No\ of\ Leaf\ Nodes\ in\ Tree\ T2$

When we consider the entire Tree T with n nodes, we can see we have 3 components to this Tree

1. Node 1
2. Sub Tree T1 with N1 number of nodes
3. Sub Tree T2 with N2 number of nodes

Calculating the number of Nodes having two children of the Tree lets state it as N_C

$$N_C = N_{C1} + N_{C2} + 1$$

Here as we can see that Number of nodes having two children for Tree T is given by $N_{C1} + N_{C2} + 1$ where $N_{C1} = No\ of\ Nodes\ with\ 2\ Children\ in\ Tree\ T1$ and $N_{C2} = No\ of\ Nodes\ with\ 2\ Children\ in\ Tree\ T2$

Whereas when we see the entire Tree T there is the node 1 which is connecting the Subtree T1 and T2 and thus Node 1 has 2 Children i.e., the Subtree T1 and Subtree T2 thus we need to add 1 to the equation.

$$N_C = N_{C1} + N_{C2} + 1$$

$$N_C = (N_{L1} - 1) + (N_{L2} - 1) + 1$$

Since Tree 1 and Tree 2 having Nodes N < n thus the Inductive Hypothesis holds true for them

$$N_C = (N_{L1} + N_{L2} - 2) + 1$$

$$N_C = (N_{L1} + N_{L2}) - 1$$

As we can see here that $N_L = N_{L1} + N_{L2}\ according\ to\ the\ Tree\ T$

$$\boldsymbol{N_C = N_L - 1}$$

Thus, using Induction, we can prove that that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

9. 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 equals 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.

Ans 9) In Accounting Method, We assign some extra tokens during the inserting the elements and then store them in the bank and while deleting we use those tokens for the deletion.

Let's consider the case

Tokens

1. No of Tokens for addLast(e) = 4 Tokens
2. No of Tokens for deleteEveryThird() = N tokens , where N = no of elements in the list

Here we are taking 4 tokens for insertion. Out of which when we insert we spend 1 token for insertion itself. So for each Insertion we will have 3 tokens in the bank as surplus. Whenever we call deleteEveryThird() we will spend n tokens for deletion and the array size will be reduce since we are removing n/3 elements. Thus at the end of any such operation we will always have some surplus amount of tokens left in the bank. Consider that when we call deleteEveryThird() we have n/3 elements have been deleted

Consider the operations since when we delete we remove n/3 elements so we are left with 2n/3 elements then again when we delete we get 4n/9 elements and so on

Cost of Deletion = $n + 2n/3 + 4n/9 + 8n/27 + ..$
$= n[1 + 2/3 + 4/9 + 8/27 ..]$
On solving this Geometric Series we get
$= 3n$

Thus when we see individual cost

Actual Cost of Insert per operation = 1
Actual Cost of Delete per operation = 3

Amortized Cost of Insert = 4
Amortized Cost of Delete = 0

The Amortized Cost per operation = O(1)

Consider the example of an array containing 16 elements in it

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

When we insert

Insertion = 4 * 16 = 64 tokens

We get 64 tokens but it takes us (1 * 16) tokens to insert, thus we have only 64-16 = 48 tokens left in the bank

After Step 1:

Insertion of 16 tokens , Bank Balance = 48 Tokens.

Considering the worst sequence of operation let's assume that now multiple times deleteThirdElement() is called as the worst case

1. Call deleteThirdElement() - 16 tokens are removed immediately since delete cost us N tokens i.e. no of elements in the array and n/3 elements are deleted and we have 10 elements

   Bank Balance = 48 - 16 = 32 Tokens

   New Array

2	3	5	6	8	9	11	12	14	15

2. Call deleteThirdElement() - 10 tokens are removed immediately now due to n being 10 and now we again remove n/3 elements. Thus we have now 6 elements left

   Bank Balance = 32 - 10 = 22 tokens

   New Array:

3	5	8	9	12	14

3. Call deleteThirdElement() - 6 tokens are removed immediately now due to n being 6 and now we again remove n/3 elements. Thus we have now 4 elements left

   Bank Balance = 22 - 6 = 16 tokens

   New Array:

5	8	12	14

4. Call deleteThirdElement() - 4 tokens are removed immediately now due to n being 4 and now we again remove n/3 elements. Thus we have now 2 elements left

   Bank Balance = 16 - 4 = 12 tokens

   New Array:

8	12

5. Call deleteThirdElement() - 2 tokens are removed immediately now due to n being 2 and now we again remove n/3 elements. Thus we have now 1 elements left

   Bank Balance = 12 - 2 = 2 tokens

   New Array:

12

6. Call deleteThirdElement() - 1 tokens are removed immediately now due to n being 1 and now we again remove n/3 elements. Thus we have now 0 elements left and reached the max worst case

   Bank Balance = 12 - 1 = 11 tokens

   Still as we can see here we have 11 tokens left which means Bank has enough Surplus. Thus this case will work.

Taking case where we get WRONG answer when we take Tokens = 3

Let's also consider a **wrong answer** considering the number of tokens be 3, In this we can explain since it doesn't work when we consider the number of tokens for insertion be 3

Consider and array having 10 elements

1	2	3	4	5	6	7	8	9	10

We are taking the number of tokens as 3 thus we have 3 * 10 = 30 tokens but insert takes 10 tokens

Bank Balance = 20 tokens

1. DeleteEveryThird () - Removes 10 tokens and array size is reduced to 6

2	3	5	6	8	9

   Bank Balance = 20 - 10 = 10 tokens

2. DeleteEveryThird () - Removes 6 tokens and array size is reduced to 4

3	5	8	9

   Bank Balance = 10 - 6 = 4 tokens

3. DeleteEveryThird () - Removes 4 tokens and array size is reduced to 2

5	8

   Bank Balance = 4 - 4 = 0 tokens

Now since we have 0 tokens in Bank any more DeleteEveryThird () will give the Bank to negative.

Hence, here we can see that addLast(e) should take 4 tokens in the tightest bound for it to work and amortized cost = O (1)

10. 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 $O(1)$.

Ans 10) Lets consider that we had n operations,
We divide them into 2 parts

1. (n -log n) operation will cost us 1
2. log n operation will cost us $i + j$ as we know that i = 2^j
    a. Thus log n Operation will cost us $2^j + j$ for some integer j

Consider a sequence

i	2	4	8
j	1	2	3
2^j	2	4	8

Here as we can see for all these numbers of i and j the condition no 1 will be satisfied and we will have (log n) number of such choices.

Consider the sequence of i till 8 = $\log_2 n = \log_2 8 = 3$,
And as we can see that we have 3 possible combinations. Hence consider n operations so there will be (log n) operation costing us $i + j$ and (n - log n) operation costing us 1

Total Cost = (n - log n) $*$ 1 + (log n) * (i + j)

$= \sum_1^{n-\log n} 1 + \sum_1^{\log n} i + j$ where i = 2^j

$= \sum_1^{n-\log n} 1 + \sum_0^{\log n} 2^j + j$

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

$= 3n + \frac{(\log n)^2-\log n}{2} - 1$

$= O(n)$

Ans 8)

(a) Input sequence :- [4, 13, 7, 15, 21, 24, 10]

constructing Binomial Heap (min Heap) from left to right.

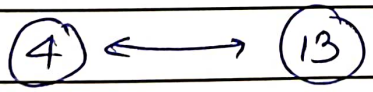
(1) Adding the element (4).

Initial we have an empty heap. and then we add (4)

B_0

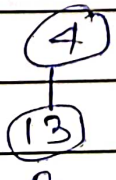
Here the Binomial heap is of the Rank 0 which is satisfies the properties.

(2) Adding the element (13).



B_0 B_0.

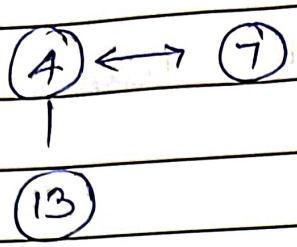
Since the order/rank of both binomial heap are same hence we merge them.



B_1

Here we have a B_1 binomial min heap

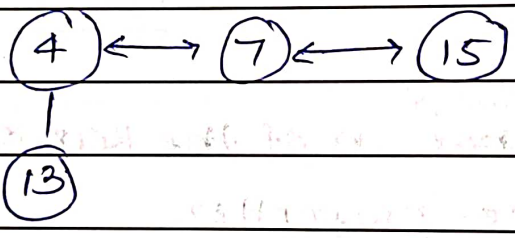
(3) Adding element (7)



B_1 B_0

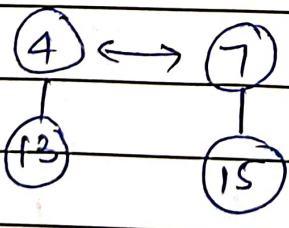
Both the order of heap are different hence we donot merge them.

(4) Adding element (15)



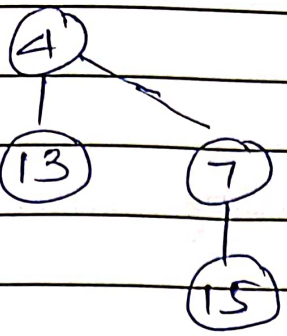
B_1 B_0 B_0.

Here since we have 2 Tree having order 0 we merge them.



B_1 B_1

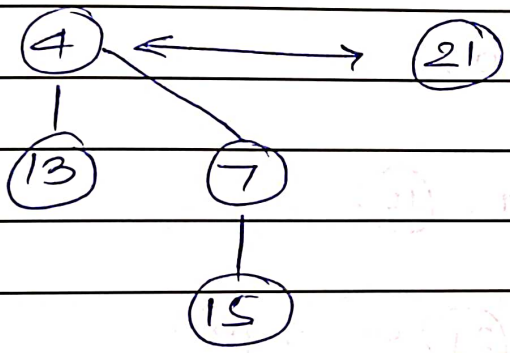
Again we have the 2 tree having same order thus we merge them again



B_2

The newly created tree has order 2.

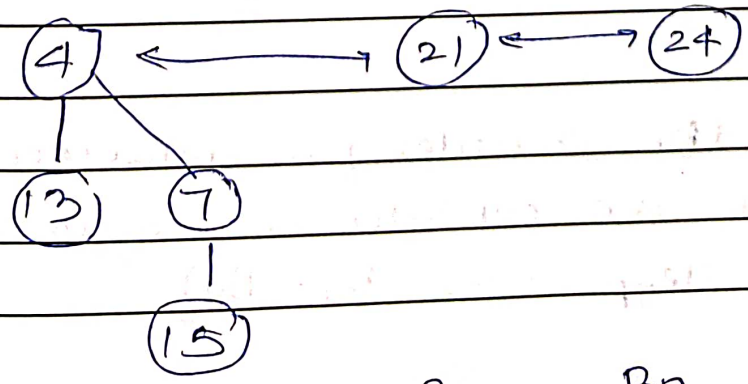
(5) Adding element 21



B_2 B_0 .

Since both the degree are different we move further.

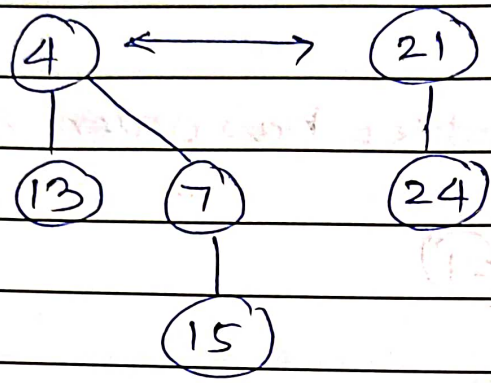
(6) Adding element 24



B_2 B_0 B_0

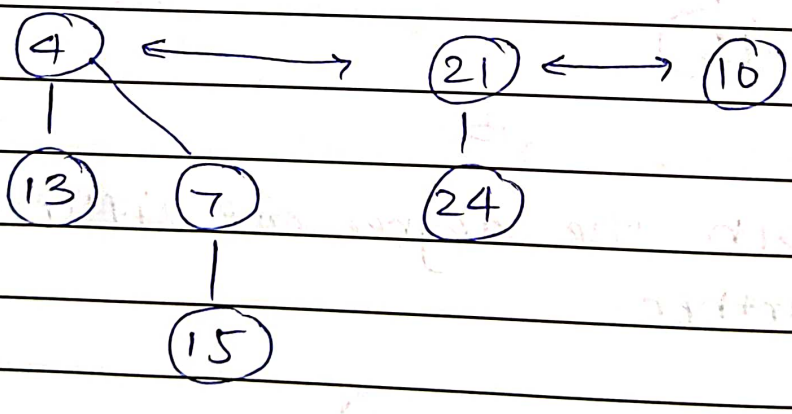
(7) ~~Adding element~~ (10)

Since we have 2 tree having order 0, we merge them.



B_2 B_0.

(7) Adding element (10)



B_2 B_1 B_0.

Since all the 3 order of binomial tree are different we consider this as our final Binomial Heap ($B_2 - B_1 - B_0$).

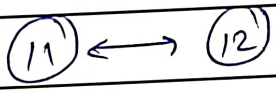
(b) Input sequence:- [11, 12, 21, 24, 18, 13, 16]

constructing Binomial min heap ~~from~~ from left to right

(1) Adding element (11)

B_0

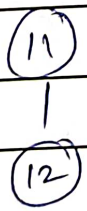
(2) Adding element (12)



B_0 B_0

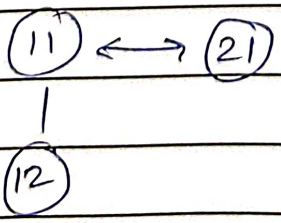
tree

since the order of both are same, we merge them



B_1

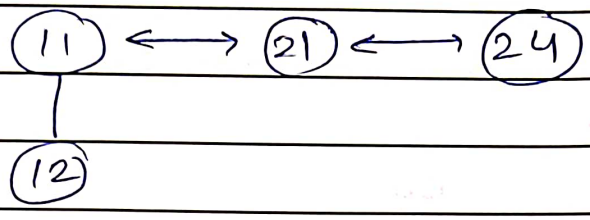
(3) Adding element (21).



B_1 B_0.

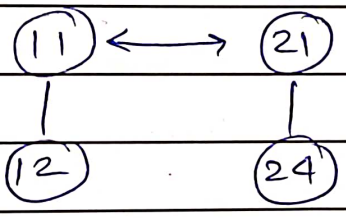
Here the order of both binomial tree are different we move forward.

(4) Adding element (24).



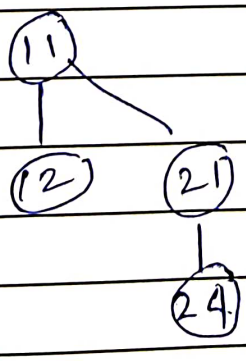
B_1 B_0 B_0

Here we have 2 Binomial tree with same rank 0 hence we merge them.



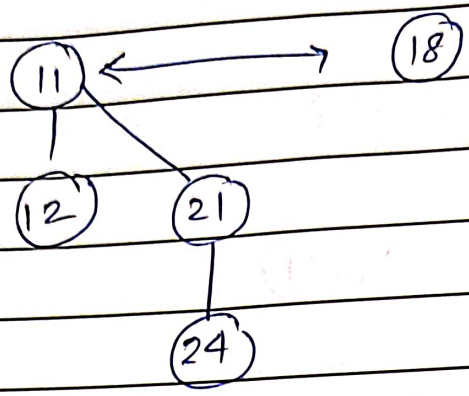
B_1 B_1

Again we have the same issue, 2 Binomial tree have same rank 1 thus we merge them.



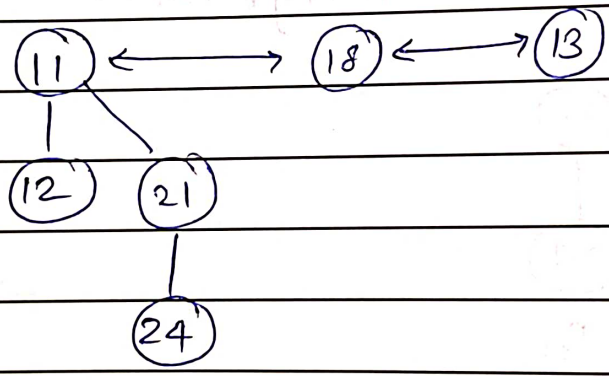
B_2

(5) Adding element 18.



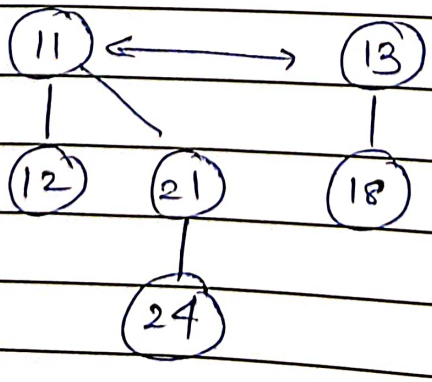
B_2 B_0.

(6) Adding element 13.

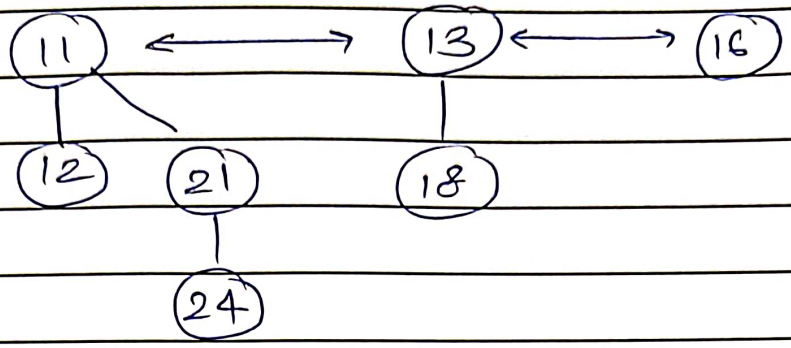


B_2 B_0 B_0.

Since, here we have 2 Binomial tree having rank = 0, we merge them.



(7) Adding element (16).

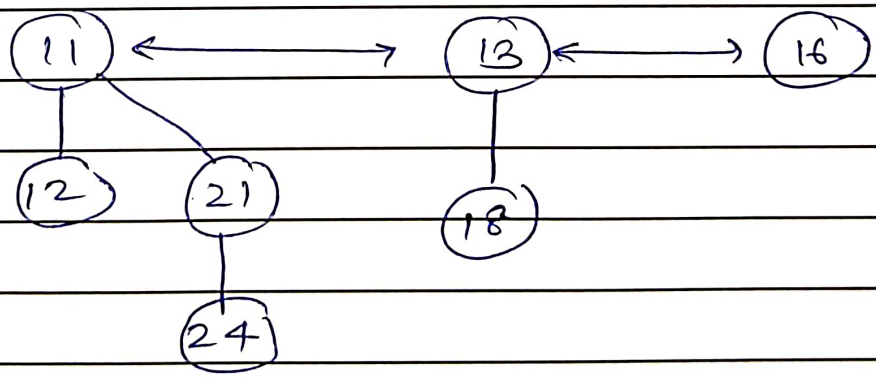


B_2 B_1 B_0.

Here as we can see we have the Binomial min heap created of 3 binomial tree having rank ($B_2 - B_1 - B_0$).

Binomial

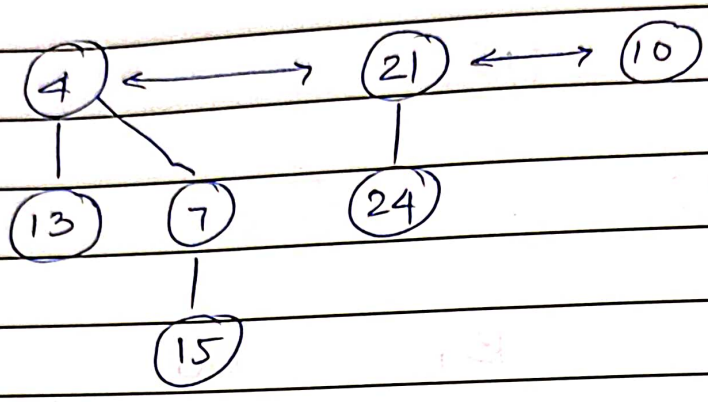
Final min-heap.



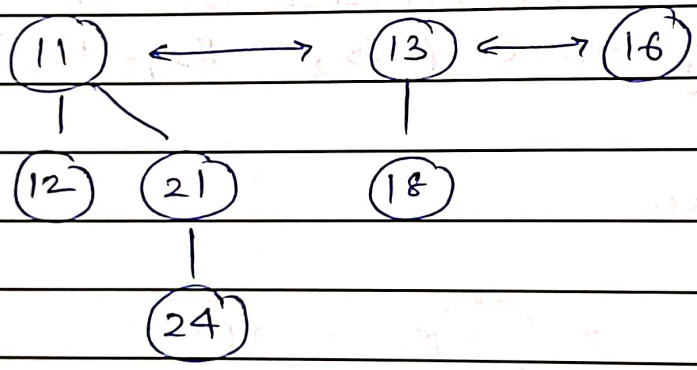
B_2 B_1 B_0.

(c) Merging 2 Binomial Heap.

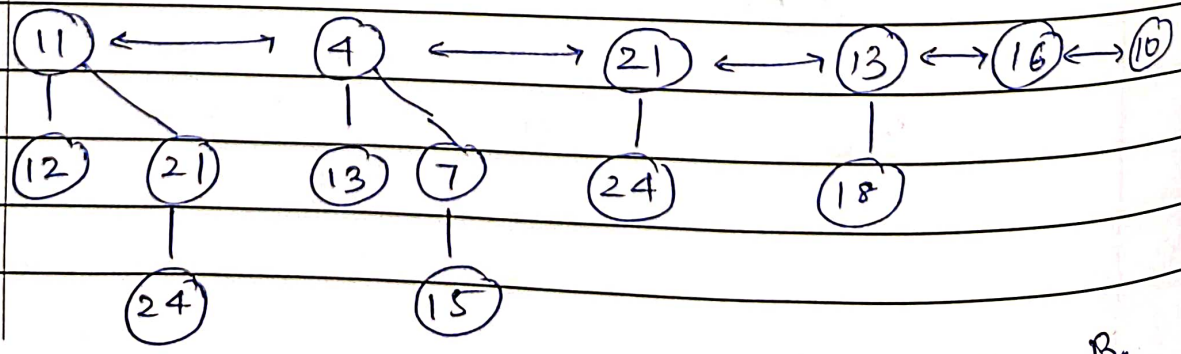
Heap H1 :-



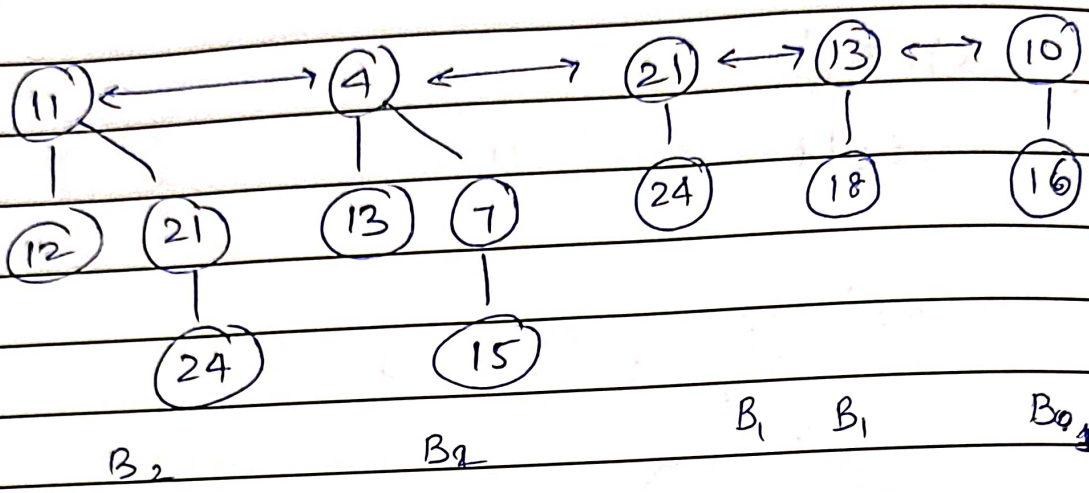
Heap H_2 :-



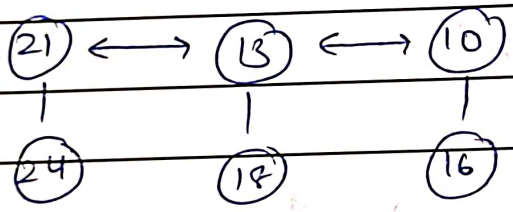
(1) ordering the 2 heap in increasing order of their rank, we get.



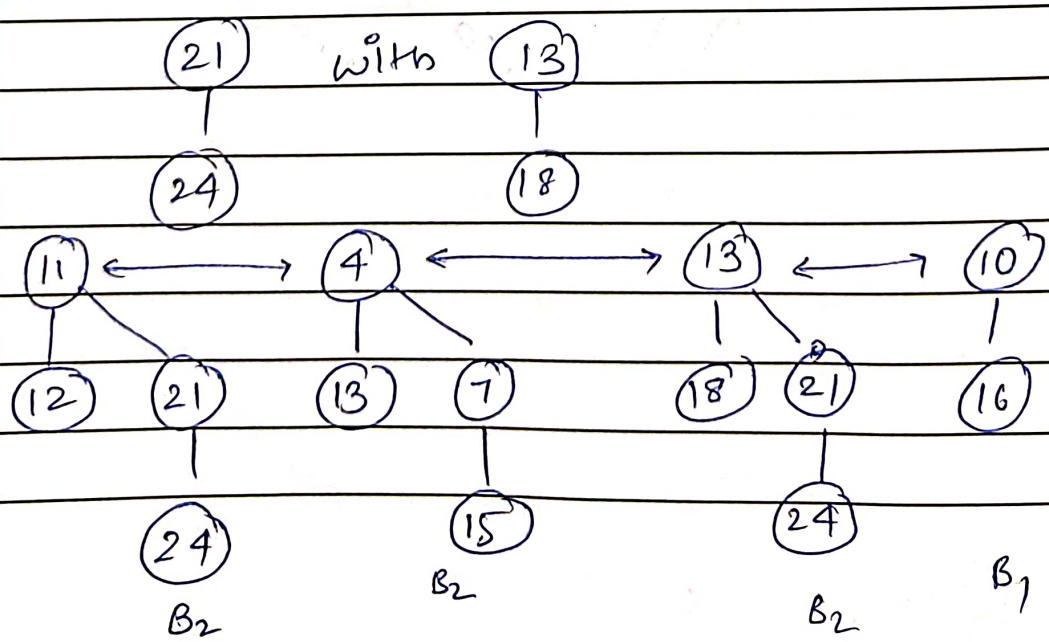
(2) Merging the binomial tree having same rank ie 0 initially



consider the Binomial tree.



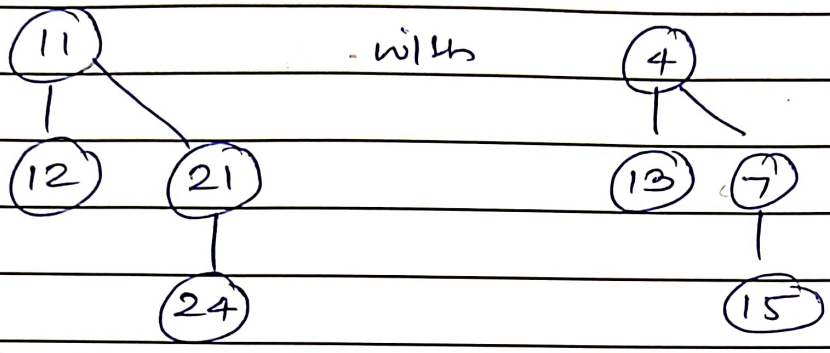
We merge the 2 Binomial tree with Rank 1 in the higher order ie merging



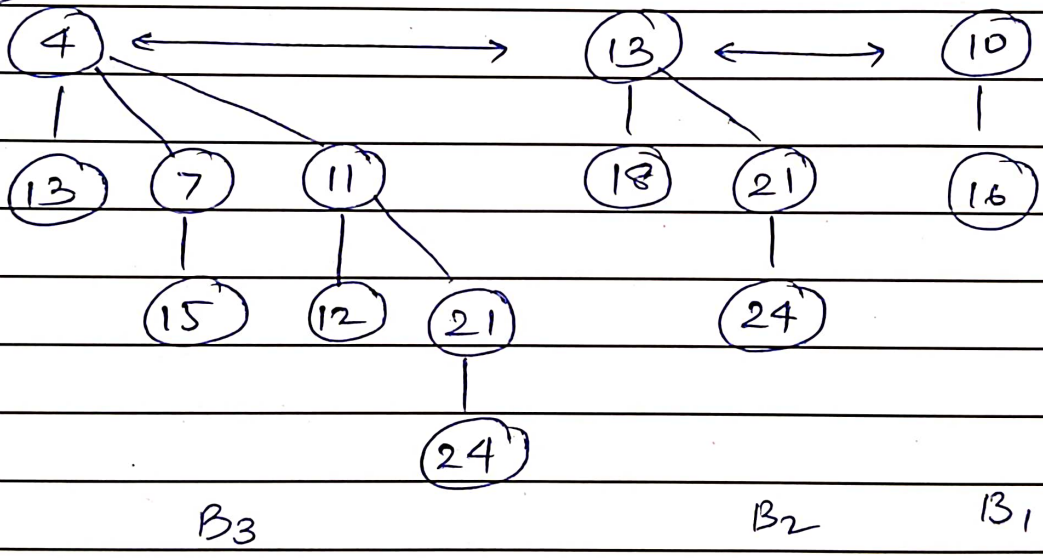
(3) Merging the ~~nodes~~ binomial tree. having same order B_2

here we consider the last 2 binomial tree.

merging



Final Heap



Finall merged heap has order = $(B_3 - B_2 - B_1)$