**Solution 1:**

**Algorithm SubMap( K1, K2)**

If (k1<k2)

SubmMapRemove( K1,K2,root());

Return;

}

**Algorithm SubMapRemove(K1, k2, Node P)**

If P internal

If P`s key < k1 // then the key should be on the right hand side.

SubMapRemove(k1,k2, right(p));

Else

SubMapRemove(k1,k2, left(p));

If P`s key < k2 {

Remove(P);

rebalanceTree(root());

SubMapRemove(k1,k2, right(p));

}

**Algorithm Void remove (Node p)**

If p is External && p==root()

Root=null;

Return;

If p is External

If p==parent.left

Parent.left == null;

Else

Parent.right=null;

return;

if p== root

swap p with inorder Succesor (the left most element in the right subtree) and then

remove (p);

}

**Algorithm Node InorderSucessor(Node p)**

If p==null

Rertun null;

While p.left != null

p= p.left;

Return p;

**End algorithm**

In this Question, we need to delete nodes with key net ween k1 and k2.

In submap we check if the key k1 is less than k2 as we expected it to be a range.so we need k1<k2

So if k1<k2 we go to SubMapRemove function and check for node P if it is internal we check p`s key if its less than k1 then the values we need to remove has to be on the right subside tree. Or else its should be on the left.

check for the value of p `key if it lies within the range k1 and k2 by checking gain is its less than k2. If it is , we remove the node and then rebalance if balance is disrupted and then call SubMapRemove function on that node .

When we delete a node, three possibilities arise.

1) Node to be deleted is leaf: Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

2) Node to be deleted has only one child: Copy the child to the node and delete the child

50 50

/ \ delete (30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Here the algorithmis (S+h) as the worst case running time is dependent on h, h is the height of the tree . the operation relies on the traversal of the tree from the root and the maximum path path length within a tree is proportional to the height of the tree.

So fo the current height of the tree h with (O(log n)), the number of entries reported by submap within range k1 and k2 as S.

The space usage is 0(n) where n is the number of entries stored in the map.

**Solution 2.**

**We can solve the given problem with quick sort as follows by two functions i.e. Sort and Matching**

**sort is used for sorting and Matching is used to match both bolts and nuts of same type.**

**Sort Function:**

Algorithm Sort (K[ ] S, int left, int right, pivot)

while (left < = right) // Start from left and right side. Move these pointers towards each other

while (S[left] < pivot)

left ++ // move the left pointer until we get the element bigger than the pivot

end while

while (S[right] > pivot)

right -- // move the right pointer until we get the element smaller than pivot

end while

if (left <= right), then // indices did not strictly cross

swap (S, left, right) // swap values and move the indexes

left++

right--

end if 1`

end while

return left // left will be partition point and return the index

End algorithm

**Matching Function (quick sort)**

**Algorithm Match** (nuts, bolts, left, right)

if (left >= right), then **// base case: subarray is trivially sorted**

return  
   
 pivot 🡨 nuts[random(left, right+1)] **// choose a nut as pivot in [left...right] at random**  
   
index 🡨 Sort(bolts, left, right, pivot) **// sort bolts around the chosen nut**

Sort (nuts, left, right, bolts[index]) **// sort nuts around the bolt at the pivot position**

**// nut and bolt at the pivot position match**

Match (nuts, bolts, left, index-1) **// solve the subproblem [left...index-1] recursively**  
  
Match (nuts, bolts, index + 1, right) **// solve the subproblem [index+1...right] recursively**

**End Algorithm**

**Explanation**

Take any nut at random and using it to sort the bolts into two groups. One group has diameter smaller than the chosen nut and the other group has the diagram larger than the chosen nut. This results in finding out the bolt that matches with the nut and forming two sub-groups.

Next use the bolt and sort the nuts into two groups. One group has diameter smaller than the bolt and the other group has diameter larger than the bolt.

It is given that every bolt has a unique matching nut. So, the size of the group of bolts with smaller diameter is equal to the size of the group of nuts with smaller diameter. Same result holds for the bigger group also.

**Therefore, repeat the above method on each of the 2 matching groups. This method is repeated till all matching is done.**

**Running time analysis**

Each round needs a maximum of O(n) comparisons in order to sub divide the groups into two smaller groups. And there are a total of O(log n) such groups.

So, the total comparisons = **O(n log n**). This is the time complexity.

For a node of depth i, we expect that:

– i/2 ancestors are good calls

– the size of the input sequence for the current call is at most (3/4)i/2n

Therefore, we have

* For a node of depth 2log 4/3 n, the expected input size is one.
* The expected height of the quick-sort tree is O(log n)
* The amount of work done at the nodes of the same depth is O(n)
* Thus, **the expected running time of** randomized quick-sort is **O(n log n)**

**Solution 3.**

1. **To prove :**

**for every graph 𝐺 : 𝑟𝑎𝑑𝑖𝑢𝑠 ≤ 𝑑𝑖𝑎𝑚𝑒𝑡𝑒𝑟 ≤ 2 𝑟𝑎𝑑𝑖𝑢𝑠.**

**Proof.**

Since the radius is the minimum eccentricity of any vertex, i.e. radius is a particular shortest path in the graph and the diameter is the maximum eccentricity of any vertex, i.e. diameter is the longest shortest path in the graph ,therefore radius cannot be larger than the diameter.

So, 𝑟𝑎𝑑𝑖𝑢𝑠 ≤ 𝑑𝑖𝑎𝑚𝑒𝑡𝑒r

Let u and v be vertices of G such that d(u, v) = diam (G).

Let w be a central vertex of G, so that e(w) = rad (G). This means that no vertex is at a distance greater than rad G from w.

By the triangle inequality: d(u, v) ≤ d(u, w) + d(w, v)

d(u, v) ≤ 2 e(w) (distance from u to w and w to v is at most

eccentricity of w)

= 2 rad(G) (since w is a central vertex)

Therefore, in particular d(u, w) and d(w, v) are both less than or equal to rad (G).

Therefore, d(u, w) + d(w, v) ≤ 2 rad (G).

This establishes that rad (G) ≤ diam (G ) ≤ 2 rad (G).

**Hence, proved that for every graph 𝐺: 𝑟𝑎𝑑𝑖𝑢𝑠 ≤ 𝑑𝑖𝑎𝑚𝑒𝑡𝑒𝑟 ≤ 2 𝑟𝑎𝑑𝑖𝑢𝑠.**

***Part b*** .

The **eccentricity** e(v) of a graph vertex v in a connected graph G is the maximum graph distance between v and any other vertex u of G.

The maximum eccentricity is the **graph diameter** (longest shortest path in the graph). The minimum graph eccentricity is called the **graph radius.**

The **center** of a graph G is the set of vertices of graph eccentricity equal to the graph radius (i.e., the set of central points).

If e(v) = rad(G), then v is a central vertex. The set of all such vertices make the center of G.

**Algorithm eccentricity(v)**

We will use BFS for finding the eccentricity of vertex 'v'. list[] contains the adjacency list of the graph. distance[] contains the distance i from v, initially value of all the elements are INFINITY. visited[] contains the boolean value representing whether a particular node has been visited before or not.

BFS(v)

1.Enqueue v into queue Q

mark visited[v]=true

2. Fill the distance[v]=0 //distance[u] represents distance between "v" and "u".

3. while(!Q.isempty())

top 🡨 Q.front()

Q.pop()

for (i 🡨 0 to list[top].size() ){

node 🡨 list[top][i]

if(!visited[node]) then

visited[node]=true

q 🡨 push(node) //push the node.

**//update the distance value**

if(distance[node]>distance[top]+1) then

distance[node] 🡨 distance[top]+1

end if

end for loop

end while loop

**/\*traverse distance matrix and return the maximum value\*/.**

int max 🡨 0

for(int i🡨0; i<n; i++)

if(distance[i]>max), then

max = dist[i]

end for loop

return max //represents the eccentricity of the vertex v

**Algorithm ends**

\* Using above Pseudo Code , we can find the eccentricities of all the vertices. Store those into an array ecc[1 ….n]

**Algorithm diameter()**

if (size 🡨 1), then // only one element in array ecc

return ecc[1] // return First element

else

return max{ecc[1..n]} //diameter is the maximum eccentricity

**end algorithm**

**Algorithm radius()**

if (size 🡨 1), then // only one element in array ecc

return ecc[1] // return First element

else

return min{ecc[1..n]} // radius is the minimum eccentricity

**end algorithm**

**// center() return all those nodes which satisfy this condition ecc[i]=rad**

**//** If e(v) = rad(G), then v is a central vertex. The set of all such vertices make the center of G.

**Algorithm Iterable<Integer> center()**

centers 🡨 new LinkedQueue<Integer>()

radius 🡨 min{ecc[1..n]} // radius is minimum eccentricity   
        for (int i 🡨 1; i < ecc.size; i++)   
            if (ecc[i] == radius) then //if ecc(i) == radius   
                centers.enqueue(i)  
        end if  
        return centers  
    End algorithm

**Running time Analysis:**

**BFS takes O(V+E) time, so eccentricity will take V\*O(V+E) approximately O(V\*V + V\*E).**

**Once all the eccentricities are calculated you can find the diameter and radius using those eccentricities with the time complexity of O(n) to find the max and min element in an array ecc.**

**Time complexity of center is O(n) since it is iterating through n elements of ecc array and enqueue operation takes O(1). Therefore, overall time complexity is O(n).**

**Solution 4 :**

Dijkstra’s algorithm is used to find the shortest paths from a given vertex to target point. It uses the heap to store all the shortest path vertices in a tree like data structure and at last gives the shortest path from source to destination.

**Algorithm to find the maximum bandwidth of paths between a and b**

**Algorithm maxBandWidth(G,a,b)**

**Input:** A weighted graph G with nonnegative edge weights, and two distinguished vertices a and b

**Output:** Maximum bandwidth of overall paths between a and b

Initialise D[a] 🡨 infinity and D[u]🡨zero for each vertex u is not equal to a in G

Let a priority queue Q contain all vertices of G representing keys as Dlabels

**while** Q is not empty **do**

**//** Call the method removeMaxElement() and store the element in u

u 🡨 Q.removeMaxElement()

//Check u is equal to the target vertex b. If the condition is true, then stop the operation here.

if(u=b) then

return D[u] //return the target vertex

else

**for** each vertex z adjacent to u such that z is in Q **do**

// calculate the bandwidth of all path between a and b and store the value in d.

d🡨min{D[u],w(u,z)} // d represents bandwidth and w represents weight

if (d > D[z]) then //check if value of d is greater than D[z]

D[z] 🡨 d // if true, change the key value of z in “Q” to D[z]

**end algorithm**

**Explanation:**

The method “maxBandWidth” used to find the maximum bandwidth of overall paths between “a” and “b” with weighted graph G as input can be achieved by modifying the Dijkstra’s algorithm. Instead of representing the shortest path from “a” to “u”, the label D[u] represents the maximum bandwidth of any path from “a” to “u”. The maximum bandwidth for path from “a” through “u” to a vertex “z” adjacent to “u” is O(n2 ). Therefore, this updates D[z] to max{D[z],min{D[u],w((u,z))}}.

-The final return statement is not required in this algorithm, because “u=b” at some point during the main loop and this algorithm will terminate and return the result then.

-The label D[a] is initialized to an infinite value because the minimum value is to be found for D[a].

-Similarly, the label D[u] is initialized to zero because the maximum value is to be found.

**Running time analysis:**

The running time of this algorithm is same as the running time of Dijkstra’s algorithm is **O((n+m)log n)** if the priority queue is implemented as a heap. If the priority key is implemented as an unsorted sequence, then running time is O(n2).