Advanced Track Project

Geometric Data Structure

Project Mentor: Prof. Surender Baswana

Associate Professor,

Department of Computer Science And Engineering, IIT Kanpur

Amit Kumar(13094)

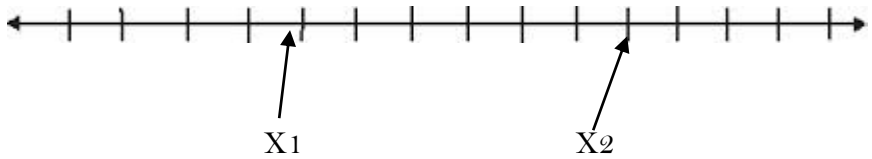
Himanshu Shukla(13309)

Junior Undergraduates, Computer Science and Engineering, IIT Kanpur.

Problem : There are n points in a plane. We want to build a data structure for these points so that given any rectangle parallel to the axes, we can report all the points inside the rectangle.

Abstract: This report contains all our proceedings and methods which we applied to reach up to the desired data structure. The methods include solving a simpler version of the problem and then extending the solution to original problem and then implementing it to look up at the real life efficiency of the designed data structure.

Simpler Version of the Problem: A simpler version of the problem can be that you are given some points on the real number line and given two points X_1 and X_2 and we have to report the number of points lying in the range (X1,X2). A pictorial view of the problem is as follows.



Now since we are given the points on the number line so we can assume that the number of values are static say equal to N.

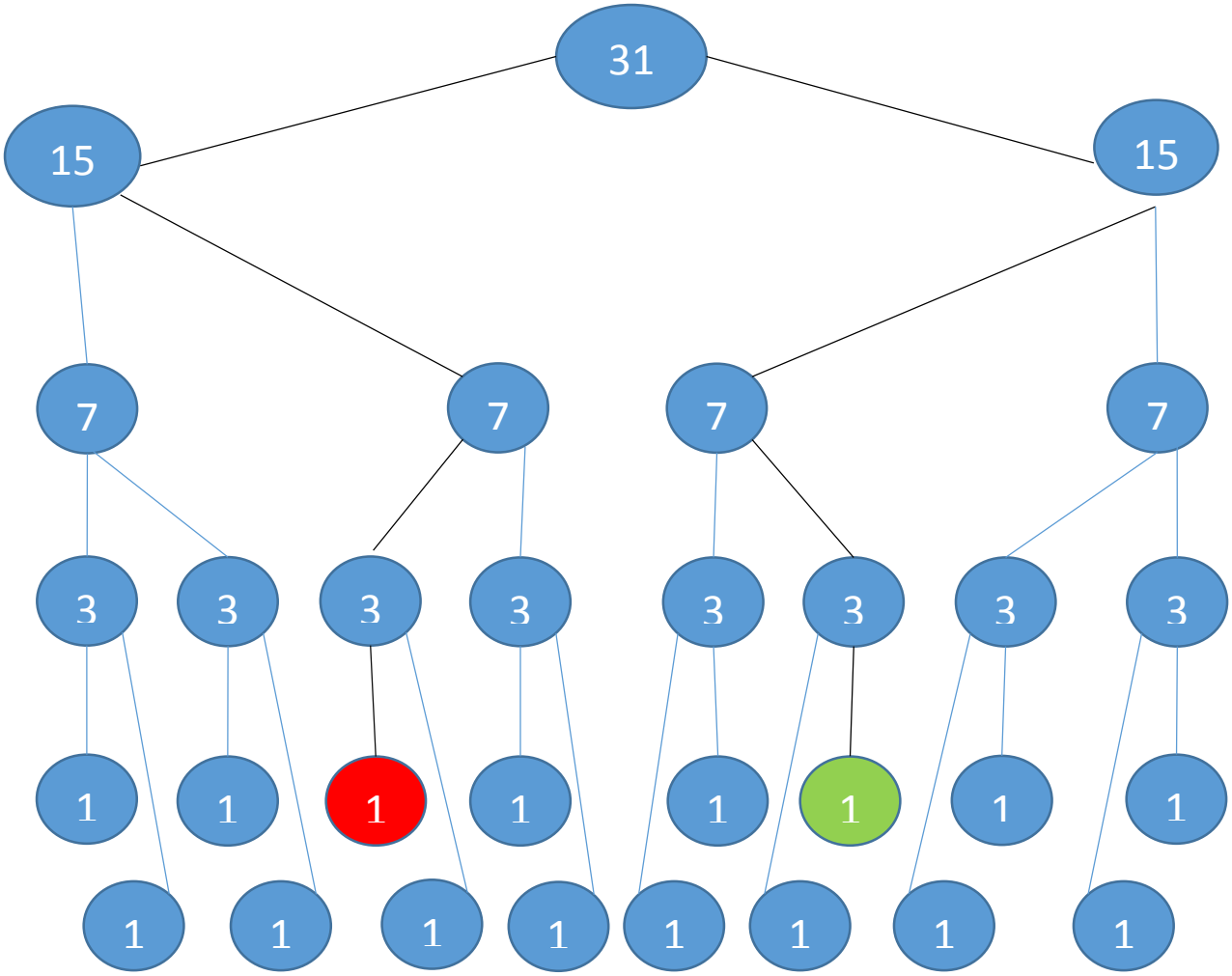
Now this has an easy solution i.e. store the elements in an array and sort them and then we can report the number of points in the query range in log(N) time. A pseudo code for the same is given as follows.

```
Query(A[N],X1,X2) // X1 < X2
{
        x1,x2; /*variables storing the indices of points in the neighbourhood of the
                        query points*/
        x1=Binary_U(X1); /*A binary search type function
gives the index of the point which is
                successor of X1 in the array*/
        x2=Binary_L(X2); /*Similar to Binary_U but
        gives index of predecessor of the element*/
        return (x2-x1+1);
}
```

The time complexity of this algorithm is trivially **O(log(N))** and the data structure takes **O(N)** space also the preparation time is **O(N*log(N))** using quick sort (average time complexity). But this is a very rigid implementation and adding any point in this data structure will require **O(N)** preparation time as this is the time required for one insertion in an sorted array. Hence we construct a dynamic data structure which takes update operations also in **O(log(N))** time.

Dyanamic Version: We will use Binary tree to store the points and keep an additional field at each node i.e. number of elements in the sub-tree at that node. Now if we are given two

points X1, X2 WLOG assuming that X1<X2 then we can find the successor and predecessor of the elements X1 and X2 respectively **O(log(N))** time. And finding the number of element >= X1 and <X2 take a time of **O(log(N))** as follows.



The above figure shows a perfectly balanced binary search tree where each node represents the number of entries inside in the sub tree below it and suppose the points X1 and X2 are represented by red and green respectively. Keep the ***count1*** to be the variable which stores the number of values >X1 and value be the field at each node that stores the number of elements at the sub-tree at that node.

For X1 each time we move left update ***count1= value (parent (v))-value(v)+1*** where v= the current node. We do nothing if we move to right. Similarly we keep a count2 for X2 and get all the points >X2.

The answer to our problem will be difference between ***count1*** and ***count2***. For reporting the elements we just traverse (in order) from X1 to X2. If we apply this algorithm in the tree show on the previous page we have ***count1=16+4+2*** and ***count2 =8***; the value of difference=14 hence we have 14 elements in the range of (X1, X2).

It is already known that insertion in a Red-Black tree takes **O (log (N))** time and the additional field could be updated in **O (log (N))** time. Hence any update operation on the above

data structure can be performed in **O (log (N))** time, hence we conclude that it's a dynamic data structure with time complexity **O (log (N))**. A pseudo-code for the same is as follows:

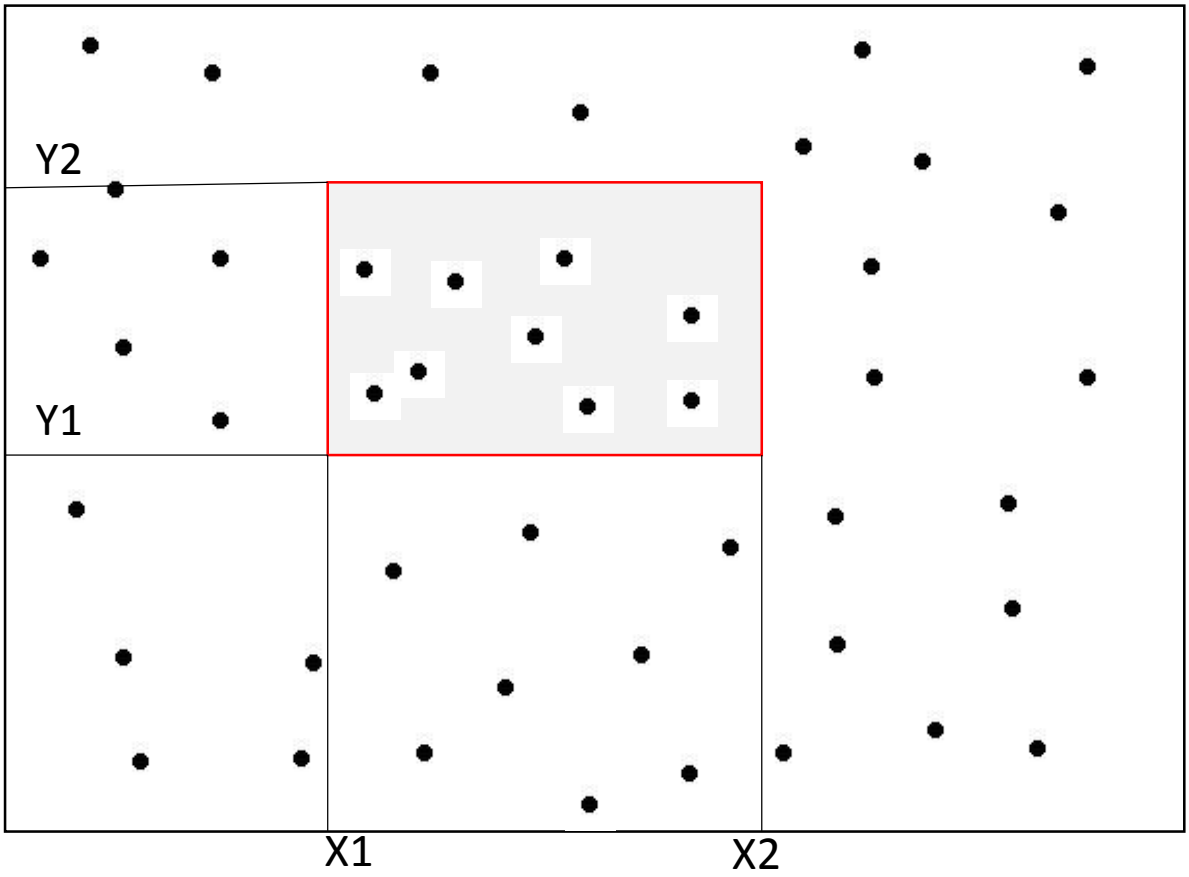
```
Query_dynamic (T, X1, X2)
{
    count2=count1=0;
    node v <-head;
    while (X1 is not found)
    {
        if (X1<v)
        {
            count1+=value(v)-value(left(v))+1;
            v<-v.left;
        }
        else if (X1>v)
        {
            v<- right (v);
        }
        else if (X1=v)
        {
            count1=count1+value (right (v));
        }
    }
    //A similar code is written for X2 to find the value of count2.
    return count1-count2;
```

} */* This code has been written assuming that X1 and X2 are present in the tree but if they are not present we can just insert them apply the algorithm and then delete the nodes.*/*

For reporting all the points we can do in-order traversal of tree from X1 to X2.

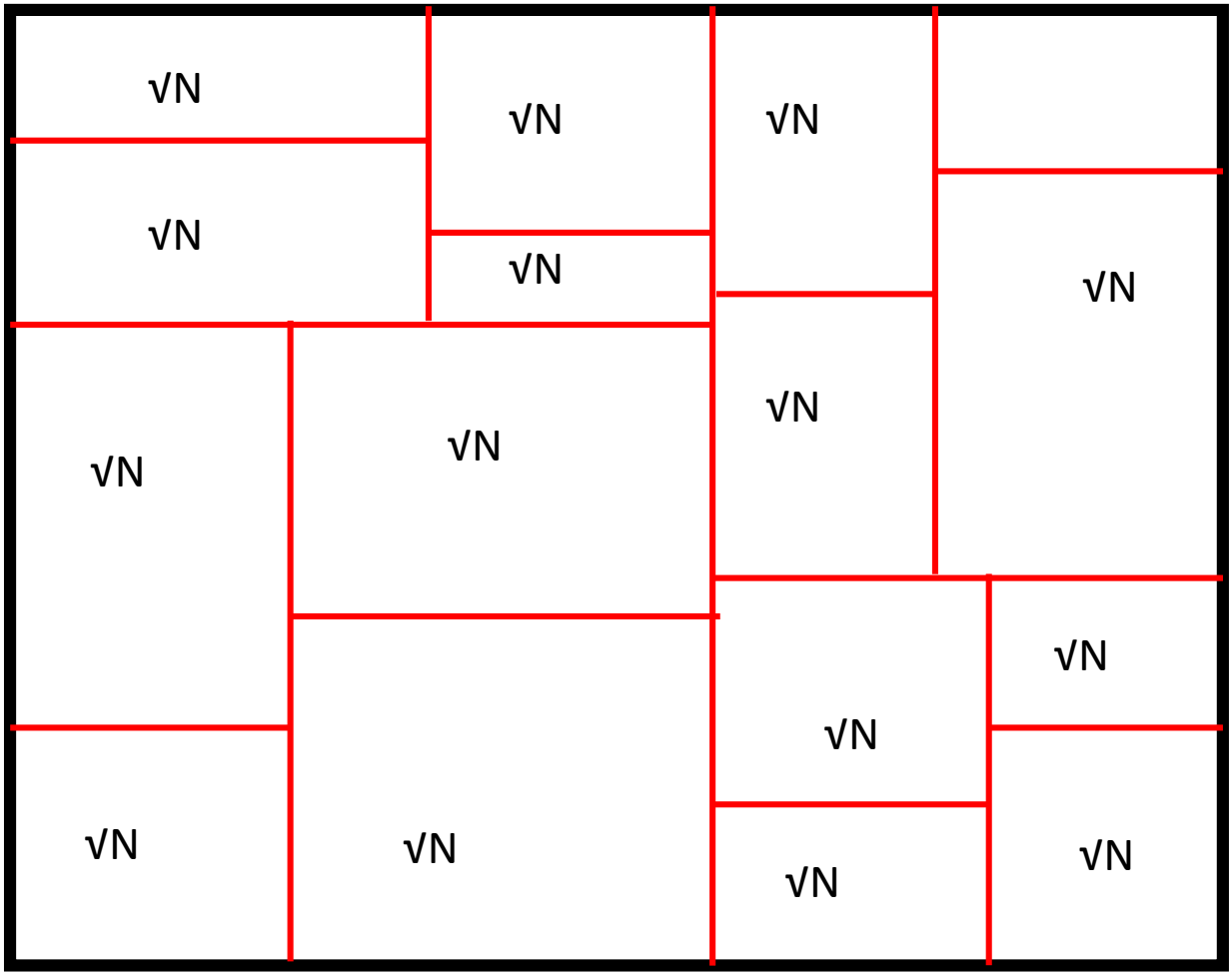
```
In_order (T, X1, X2)
{
    node v=head;
    if (v= =NULL) // base case
    {
        return ;
    }
    If (X1<v) /*if value of X1 is<value in v then move left*/
    {
        In_order (left (v), X1, X2);
    }
    If (X1<=v and X2>=v) /* if value in v lies in [X1,Y1] then print it*/
    {
        Print (v);
    }
    if (X2>v) /*if value of X2 is>value in v then move right*/
    {
        In_order (right (v), X1, X2);
    }
    return 0;
}
```

Lets now extend the data structure into a more general problem that is we are given N points with there in the form of ordered pairs (X,Y) and we have to design a data structure to answer a query which is given in the form of a query rectangle (X1,Y1), (X2,Y2) and the sides of the rectangle are parallel to the axes and report the number of points in the query rectangle. The pictorial view of the problem is as follows:



We aim to design an efficient data structure. One trivial algorithm will do the job in O (N) time as we can always use our previous algorithm to get all the points in the range $[X1, X2]$ and then for all those values check whether it lies in the range$[Y1, Y2]$. But one can easily realize that the trivial algorithm over the previous data structure will only be good if the data is of the order 10^8 after that it becomes in-efficient hence there is need to design a data structure that can handle data up to the value of $N = 10^{18}$ as in many places we come across such large values of N. If we are some how able to design a data structure with space complexity of the O (N) and time complexity of the $O\left(N^{\frac{1}{2}}\right)$ then we are done.

The reason being time taken to process each query will be of the order few seconds. This can be done as follows: Look at the following picture:



The division is made so that each of the simple rectangle that we see contains n^ (1/2) points. The division algorithm

is as follows.

Get the point x1 such that the number of points on the left of it = $[[\sqrt{N}]/2] * [\sqrt{N}]$

and on the right of it

$= N - [[\sqrt{N}]/2] * [\sqrt{N}]$ and then y1 and y2 on the either side of the x1 such that the number of points above and below y1

$=[[[\sqrt{N}]/2]/2] * [\sqrt{N}]$

and $[[[\sqrt{N}]/2] - [[[\sqrt{N}]/2]/2]] * [\sqrt{N}]$

and similarly for $Y2$ and then get $X2$ above $Y1$; $X3$ below $Y1$ and similarly $X4$ and $X5$ for $Y1$ and the procedure goes on. Until each of the

partition contains $\sqrt{N}$ elements. Now if we are given a query rectangle then there are two possibilities:

- either the rectangle lies completely on the boundaries of the partitions.

- At least one of the sides of the rectangle does not lie completely on the boundaries of the partitions.

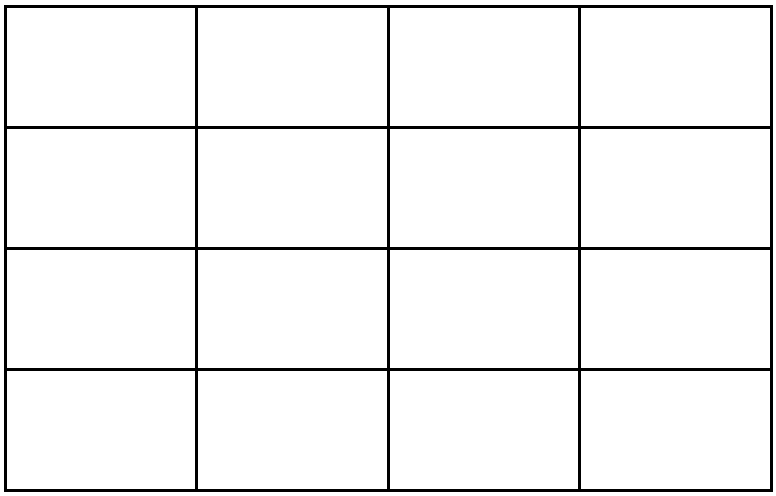
In the first case the task is easy and the answer comes in $O(\sqrt{N})$ time trivially.

In the second case requires the following **observation:**

The number of partitions which are cut by the boundaries of the query rectangle are tightly bounded by $O\left(N^{\frac{1}{4}}\right)$.

Proof: We prove this by a simple observation and by creating a bijection with a known case.

Suppose we had to make N equal partitions on a plane then the following is an obvious way.



Let us look at it with a different perspective. Any two partitions in this partitioning are called independent if they do not share any edge among themselves. Now the partitions that our algorithm creates can be created by just laterally shifting the internal edges of the partitions as follows:

- Keep the midline partition intact.
- Shift the horizontal midline in the left half upwards and the on the right half downwards.
- Then keep on applying the shifting appropriately to get the required partitioning.

Hence this partitioning is equivalent to the partitioning that our algorithm will create. Hence the number of partitions for $\sqrt{N}$ partitions on the axis will be $N^{\frac{1}{4}}$.

Hence the number of partitions which are cut by the boundaries of the query rectangle are tightly bounded by $O\left(N^{\frac{1}{4}}\right)$.

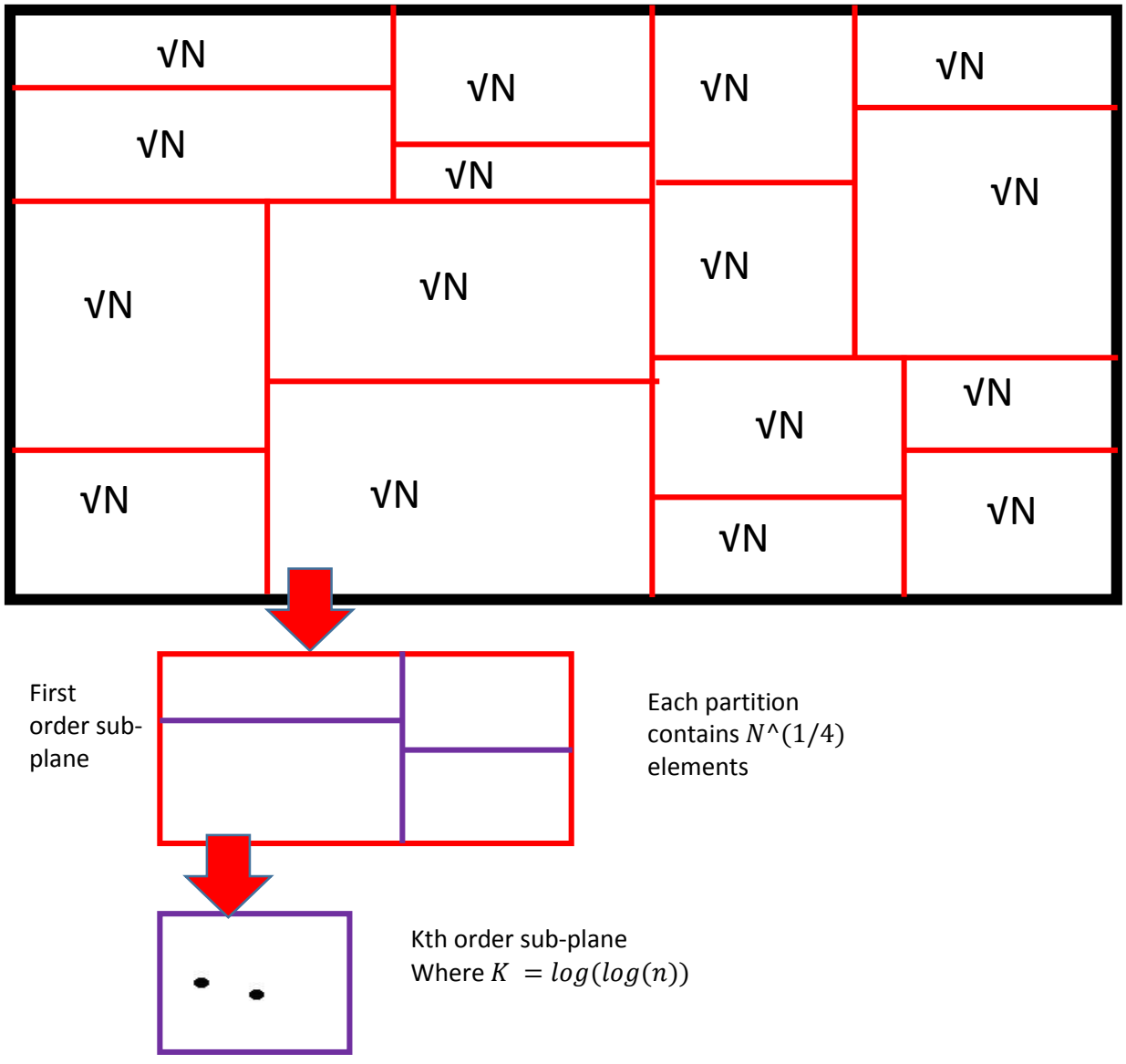
Now we can go inside each of the partition that are not completely lying inside and check for each element if lies in the range $[Y1, Y2]$. The time complexity for this will be $\boldsymbol{c\sqrt{N} + dN\text{^}(3/4)}$.

And note that the extra space required for such a data structure will be of the O (√N). But our aim is to build a data structure that answers my query in O (√N) time. We will use this virtual data structure to build upon our required virtual data structure.

We observe that when we put our N elements into our data structure it takes time~ $c\sqrt{N} + dN^{\frac{3}{4}}$. Now if we put ($\sqrt{N}$) elements in the data structure then the time complexity for these √N elements will turn out to be $c\,N\hat{}\left(\frac{1}{4}\right) + dN^{\frac{3}{8}}$. The time complexity is reducing hence we can use this phenomena to solve our problem.

We extend our data structure as follows:

Consider a machine **"Create"** that creates the partitioning given the points. We first send our original plane of N points into it and it creates the partitioning a described by the algorithm. Note that each partition contains √N points, we call this partition to be a ***first order sub-plane.*** We send all the first order sub-planes into the machine **Create,** which creates $\boldsymbol{N^{\frac{1}{4}}}$ partitions in each ***first order sub-plane***. We call the partitions formed, ***second order sub-plane***. We keep going on until there are only one or two elements in each ***Kth order sub-plane*** ($\boldsymbol{K = log(log(N))}$). Note that this virtual data structure has space complexity = $O(N)$ as we have just created partitions in the plane. A pictorial view of the data structure looks as follows:



The time complexity of the answering one query in this data structure can be calculated as follows:

Suppose we take $O(\sqrt{N})$ time to calculate the number of sub-planes present completely inside the query rectangle and by our previous analysis we had already shown that the number of the sub-planes that will not completely lie inside the boundaries of the query rectangle will be at max of the $O(N\text{^}(1/4))$ in 0th order sub-plane. So the same analysis is valid for successive order sub-planes and hence the recurrence for the time-complexity of the algorithm will be:

$$\boldsymbol{T(N) = c * \sqrt{N} + N\text{^}(1/4)(T(\sqrt{N}))}.$$

On unfolding it we get to the following:

$$\boldsymbol{T(N) = \sqrt{N} + \sqrt{N} + \sqrt{N} + \ldots\ldots + \sqrt{N} \quad (log(log(N)))\ times}$$

Hence the time complexity is of the

$$\boldsymbol{O(\sqrt{N} * log(log(N)))}.$$

We have still not reached our required time complexity but lets have a closed look at time complexity we got. If we put the value of N=10000000000000000000 then the value of $log(log(N)) \sim 6$. So the value of $(\sqrt{N} * log(log(N))) \sim 6\sqrt{N}$.

While calculation we have assumed that we take $O(N\text{^}(1/2))$ time for answering that how many partitions of the first order sub-plane are there inside this.

The data structure we have thought we have assumed that it takes $O(\sqrt{N})$ to tell the number of sub-planes lying completely inside the query rectangle. Suppose we make our data structure that can tell us the number of sub-planes lying inside the query rectangle in $O(1)$ time then the recurrence becomes:

$$\boldsymbol{T(N) = c + N\text{^}(1/4)(T(\sqrt{N}))}.$$

The result of this recurrence has an upper bound

$$\boldsymbol{= c\sqrt{N} * log(log(N)/2}$$

and the average upper bound comes out to be

$$\boldsymbol{\sqrt{N} * log(log(N)/(N\text{^}(-1/\sqrt{log(N)}))}\ ,$$

this value comes out to be $\sim \sqrt{N}/100$ for $N = 10000000000000000000$.

Now we go on to implementation of this virtual data structure that we have discussed till now. We will implement a data structure and then compare its space and time complexity with our virtual data structure.

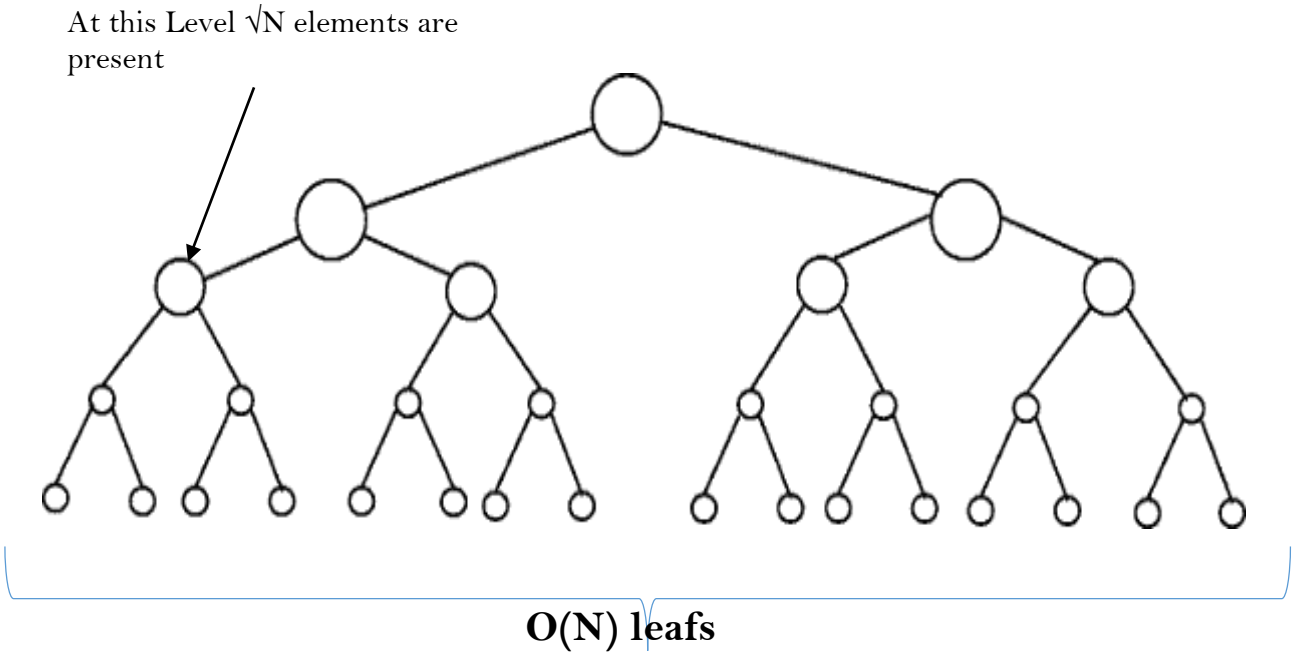
THEORETICAL IMPLEMENTATION

Now we will use the data structure that we designed for the single dimensional version of this problem will modify it so that it has all the properties of the virtual data structure we discussed.

Hence we use a Binary tree for implementation which will have the following fields:

- Partition co-ordinates i.e. X1, Y1 and X2, Y2 that to represent the sub-plane.
- Count that contains the number of values that are lying in that partition.

The root node contains the (X1, Y1) and (X2, Y2) equal to the value of the origin and the point at the maximum distance from the origin and count value = number of points (N).



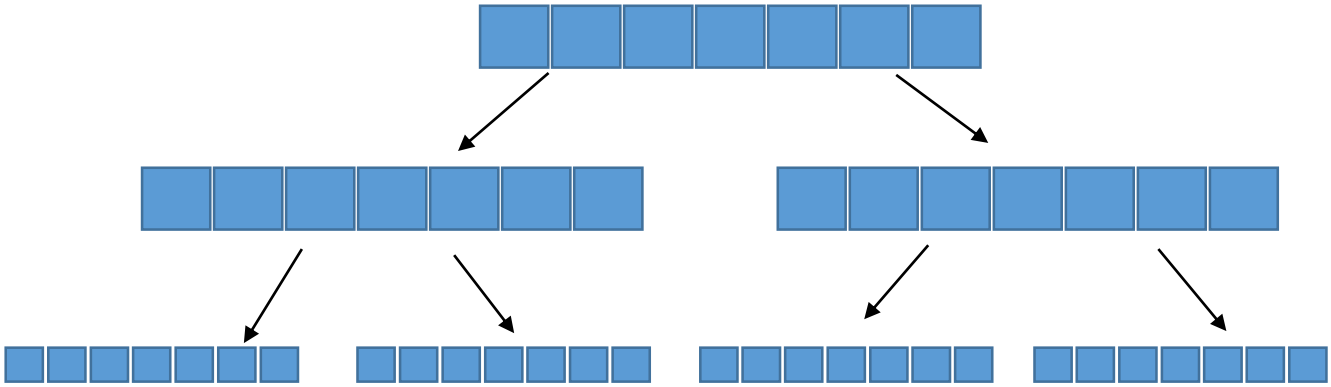
If this is a complete binary tree for the worst case, then at the k^{th} level number of node is 2^k , where k starts from 0.

We have divided the 2-D plane into $N^{\frac{1}{2}}$ partition each containing $N^{\left(\frac{1}{2}\right)}$ points. Now this $N^{\frac{1}{2}}$ partitions is further divided into $N^{\frac{1}{4}}$ partitions and this way it continues until we have n partition each containing 1 point. N leafs denotes the n partitions having n points.

Now, each node in the above tree represent a partition and has **seven** fields for x1,y1,x2,y2, count, a and b . If the partition is like

Thus (x1,y1) and (x2,y2) the coordinates to determine a partition.
Count represent the number of points in the sub-tree partitions.

If the count in the partition is 1 then a, b stores the co-ordinates of the point inside it.



PSEUDO CODE

```
Sum=head->count;
Check (pointer, x1,y1,x2,y2)
{
If (partition denoted by pointer is partially in query rectangle)
        return1;
If (partition denoted by pointer is completely out of query rectangle)
        return 0;
If (partition is completely in the query rectangle)
        return 2;
}
counter ( coordinates of query rectangle, pointer)
{
Node= pointer;
If (check (Node->left, x1, y1, x2, y2) ==1)
{
        If ((Node->left)->count==1)
                check whether the point stored is in the query rectangle or not
```

```
                if NO;
                sum =sum-1;
        Else
                counter (x1, y1, x2, y2, Node->left);
}
If (check (Node->right, x1, y1, x2, y2) ==1)
{

        If ((Node->left) ->count==1)
                check whether the point stored is in the query rectangle or not
                if NO
                sum=sum-1;
        Else
                Counter (x1, y1, x2, y2, Node->right);
}
If (check (Node->left, x1, y1, x2, y2) ==0)
{
        sum= sum-Node->left->count;
}
If (check (Node->right, x1, y1, x2, y2) ==0)
{
        sum=sum-Node->right->count;
}
}
```

Analysis of this Data structure:

Space Analysis:

We use the concept that number of leaves in a complete Binary Tree is one less than the number of internal nodes so if we look at the plane wise partition then we see that there are $\sqrt{N}$ ***nodes*** representing the co-ordinates of the first order sub-plane so exactly $\sqrt{N}-1$ internal nodes are there including the root. Now these $\sqrt{N}$ ***nodes*** serve as our roots for the formation of second order sub-plane. Hence the total number of nodes are :

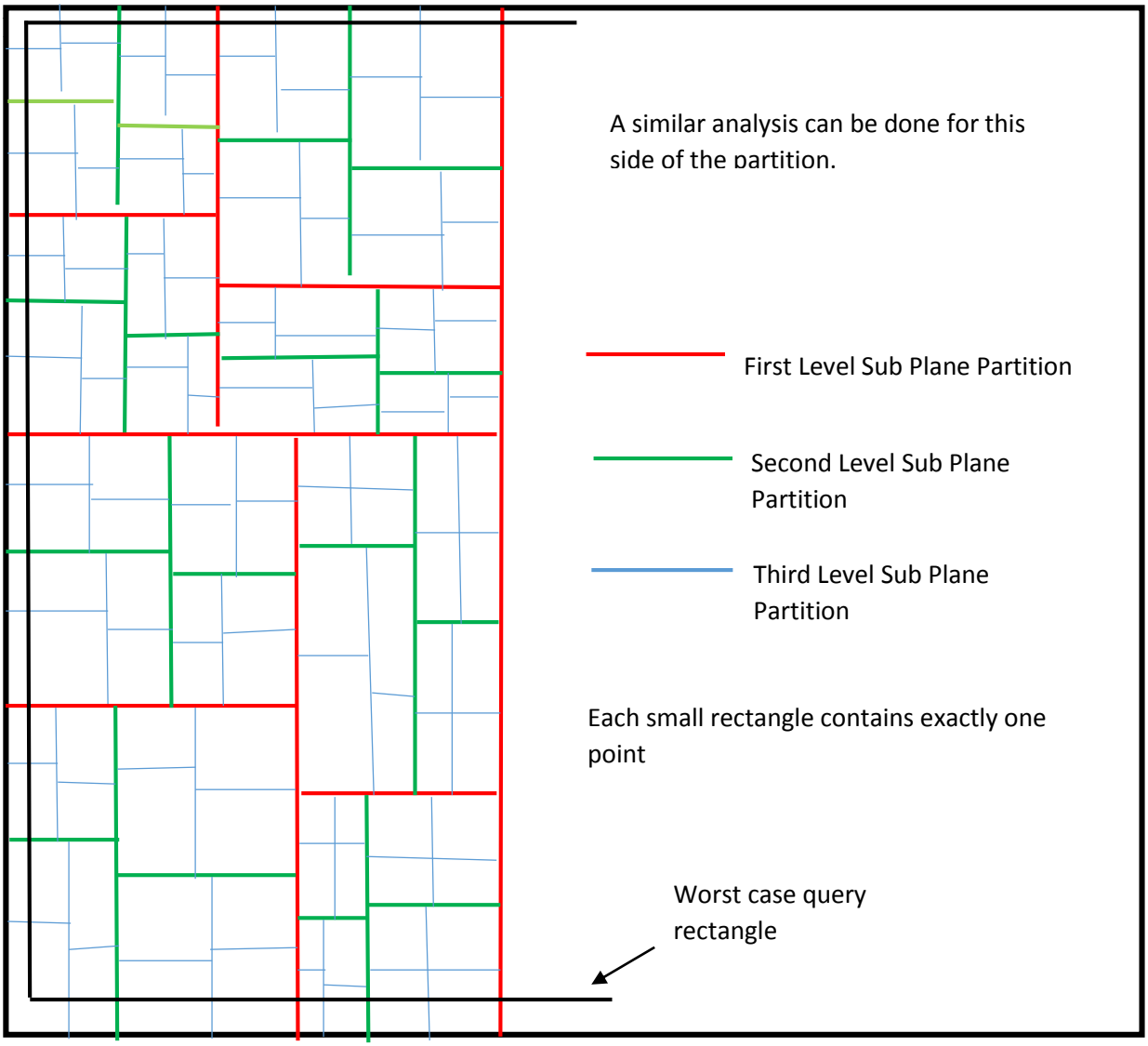
$$S(N) = N^{\frac{1}{2}} - 1 + N^{\frac{1}{2}}\left(N^{\frac{1}{4}} - 1 + N^{\frac{1}{4}}\left(N^{\left(\frac{1}{8}\right)}\right.\right.$$
$$\left. - 1\left(\ldots\ldots\ldots \left(N^{\left(\frac{1}{\frac{\log(N)}{2}}\right)} \right)\left(N^{\frac{1}{\log(N)}} \right) \ldots\ldots \right)\right)$$

This series sums up to $\frac{N}{2}$. Hence the space complexity is of **O (N).**

Also we can see that we are doing nothing but partitioning the points such that at the end one partition will contain number of points either one or two. So the number of leaf nodes in this=$O(N)$ and hence using the property of binary tree total number of nodes are of the $O(Number\ of\ leaf\ nodes)$ total space complexity comes out to be $O(N)$.

Time Analysis:

Note that the division leads us to a very exciting feature of the tree i.e. all the final level partitions will have exactly one point in each partition. For the time analysis we look at the worst case. But before that lets look at the plane after we have achieved all the partitions.



We can observe that the number of partitions at the last level that have to checked manually will be at most $= \mathbf{4} * \boldsymbol{N}^{\frac{\mathbf{1}}{\mathbf{2}}} - \mathbf{4}$. Now lets look at how much time does the algorithm takes to reach these $\mathbf{4} * \boldsymbol{N}^{\frac{\mathbf{1}}{\mathbf{2}}} - \mathbf{4}$ nodes in the implementation. We observe that the number of operations that the algorithm performs is proportional to the number of the nodes it visits while it reaches these $\mathbf{4} * \boldsymbol{N}^{\frac{\mathbf{1}}{\mathbf{2}}} - \mathbf{4}$ nodes.

Claim: We claim that the case shown is the worst case.

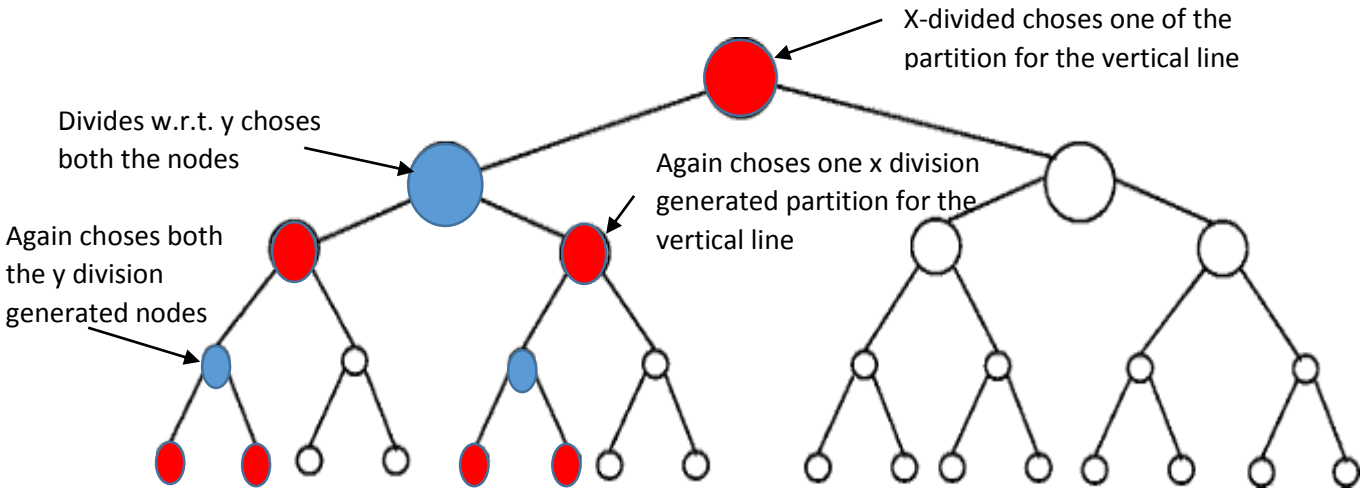
Proof: We prove this by saying that we will enclose any query by the smallest partition just bigger than the query rectangle. Hence the maximum of all such partitions will be the original main plane.

Note: From now on all the complexity analysis will be done using $\boldsymbol{N} = \mathbf{2}^{\mathbf{2}^{\boldsymbol{K}}}$ and the analysis is done for the worst case.

Observation 1: We observe that any vertical or horizontal line can pass through at most through $\boldsymbol{N}^{\frac{\mathbf{1}}{\mathbf{2}}}$ partitions. This evident by construction.

Observation 2: We further observe that between two successive partitions with respect to X direction there is level of Y- partitioned nodes and hence there will be about $.\mathbf{5} * \boldsymbol{log}(\boldsymbol{N})$ X partitioned levels and $.\mathbf{5} * \boldsymbol{log}(\boldsymbol{N})$ Y-partitioned levels.

Observation 3: We further observe that there are only three possibilities for the any node either the partition represented by it is completely outside or it is completely inside or it is or the line passes through this partition.

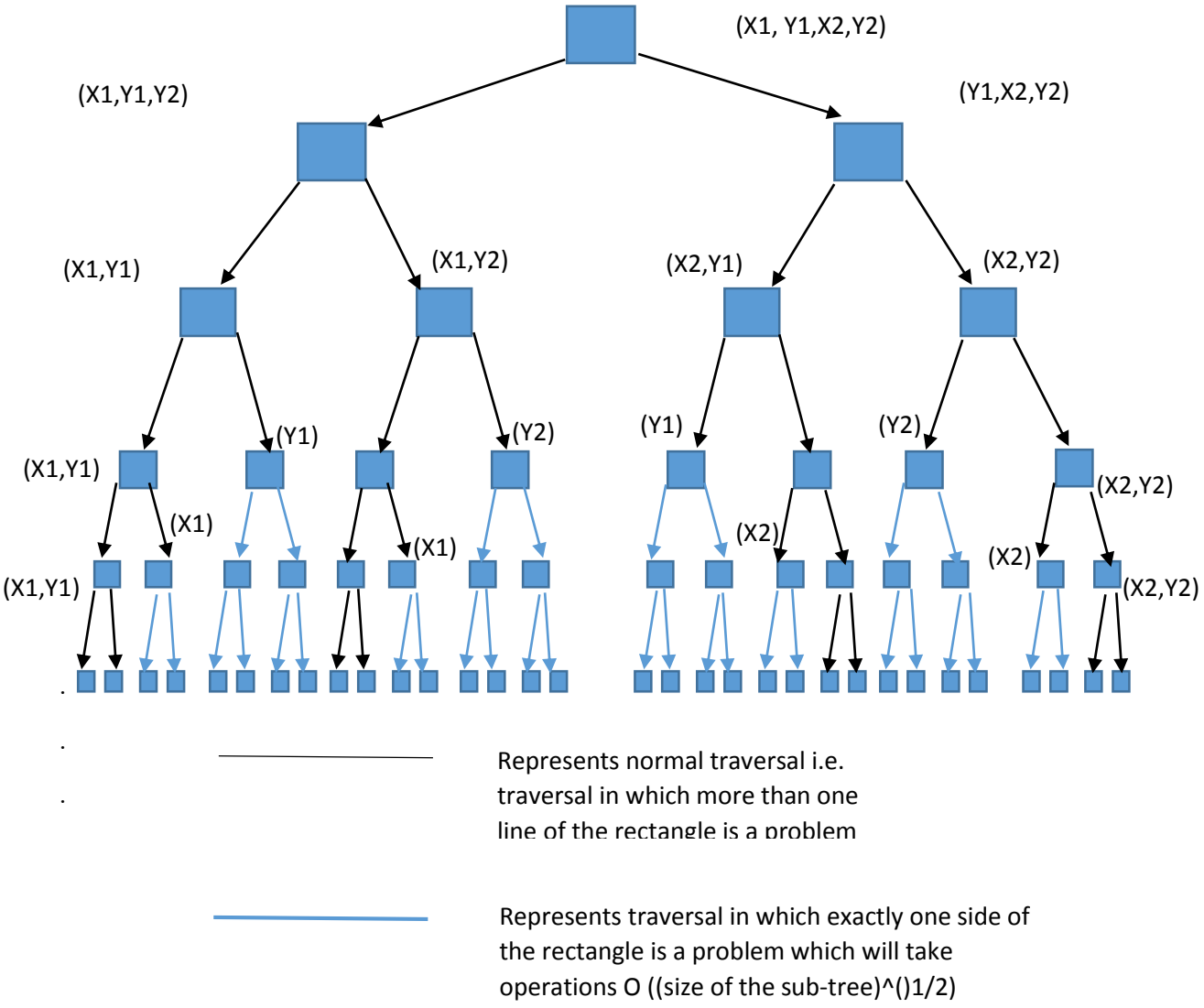


Now choose any line (vertical or horizontal) without loss of generality lets say vertically oriented, the line chosen is completely extended line to analyse the worst case. Now when the algorithm starts from the root it has three possibilities by observation 3. If the first two cases are there then we will leave the sub-tree or add its count in $O(1)$ time but for the third case I will have to traverse. Since it's the single line while passing through the root it will chose exactly one path as either the line is left of the mid partition or on the left. Then division of Y does not has any affect. Hence will traverse both the nodes then again division with X

occurs and this goes on so we can write the recurrence for such a process: Let $T(N)$ be the time for doing answering the query for a line in the dataset of size N, then

$$\boldsymbol{T(N) = 2 + 2T(N/4)}$$

Using master theorem we can prove that the order for such a $T(N)$ will be $O(N\text{^}1/2)$. By unfolding method we get the expression for $T(N) = 3 * N\text{^}(1/2) - 2$. Now lets look into our case when we have four such lines. As one can observe that there will be problems with the partitions that enclosing the corner of the rectangle which adds to the non-triviality of this analysis. We look into the following figure to get an insight how the algorithms traverses the tree when there are four lines and how it handles the corner cases.



Note: (a, b, c, d) represents the sides of the query rectangle which pass through the partiton

Hence after visiting first 3 levels the algorithm starts leaving the sub-trees. At each step after 3rd level exactly 4 sub-trees are generated in which we have to move with respect to the only one side and the size of the sub-trees generated is half of the size of the sub-trees generated at the previous level. Hence we have the following series:

Assuming that for each sub-tree generated we take O ((size of the sub-tree)^(1/2)) operations we get:

$$S(N) = 4 * \left(c\left(\frac{N}{8}\right)^{.5}\right) + 4 * \left(c\left(\frac{N}{16}\right)^{.5}\right) + 4 * \left(c\left(\frac{N}{32}\right)^{.5}\right) \ldots\ldots\ldots + 4 * \left(c\left(\frac{N}{N}\right)^{.5}\right)$$

$$S(N) = 2 * c * N^{.5} * (3.416)$$

$$S(N) \sim 7 * c * N^{.5}$$

Hence the order is still $N^{.5}$.

And in the further sections we will show the practical implementation data which shows number of operations $\sim 8 * N^{.5}$

Note that I have assumed that $N = 2^{2^k}$. Because of which my tree became a perfectly balanced binary tree.

But suppose its not the case then in that case for any order sub-plane if size of that order sub-plane is N then $k^2 < N < (k+1)^2$ for some k

For max deformation according to the division algorithm we chose $N = (k+1)^2 - 1 = k^2 + 2k$

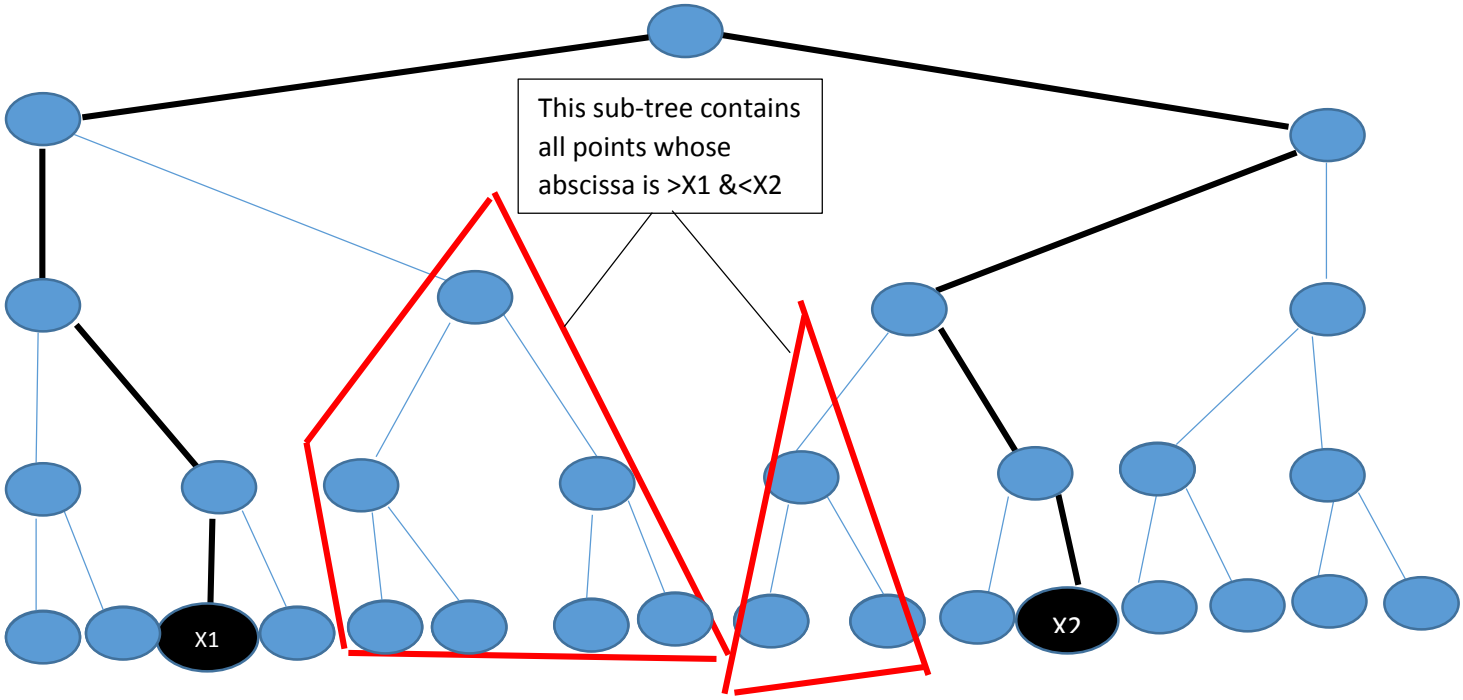
Then the left sub-tree at the root of each order sub-plane will have $\frac{k^2}{2}$ nodes and the right sub-tree will have $\frac{k^2}{2} + 2k$ nodes now we can look that the tree at the left has only $O(N^{.5})$ nodes less thus it's a nearly balanced binary tree.

Note: In the above proof for the skewed tree we have proved for a generic root of a particular order sub-plane. Hence N is the number of nodes for that root (sub-tree).

This Data Structure is very efficient if we look at the space complexity but suppose that we have a large number of queries then $N^{\frac{1}{2}}$ is not a very efficient complexity and this type of data structure is useful if we have very restricted space. So we like to have a data structure that can answer us in $\sim$**O (log(N))** time and can use some more extra space. We know that if the space complexity is $\boldsymbol{N^2}$ then we can answer each query in $O(1)$ time. So we give us a freedom to use a space of about $\boldsymbol{O(N(log)(N))}$. Lets look how to get that.

ALTERNATE DATA STRUCTURE

Since we already saw a data structure that takes $O(\log(N))$ when we had points lying on the real line. We take inspiration from it and sort the points with respect to the X co-ordinate then use our data structure that we used in the one dimensional version to store these points as shown:



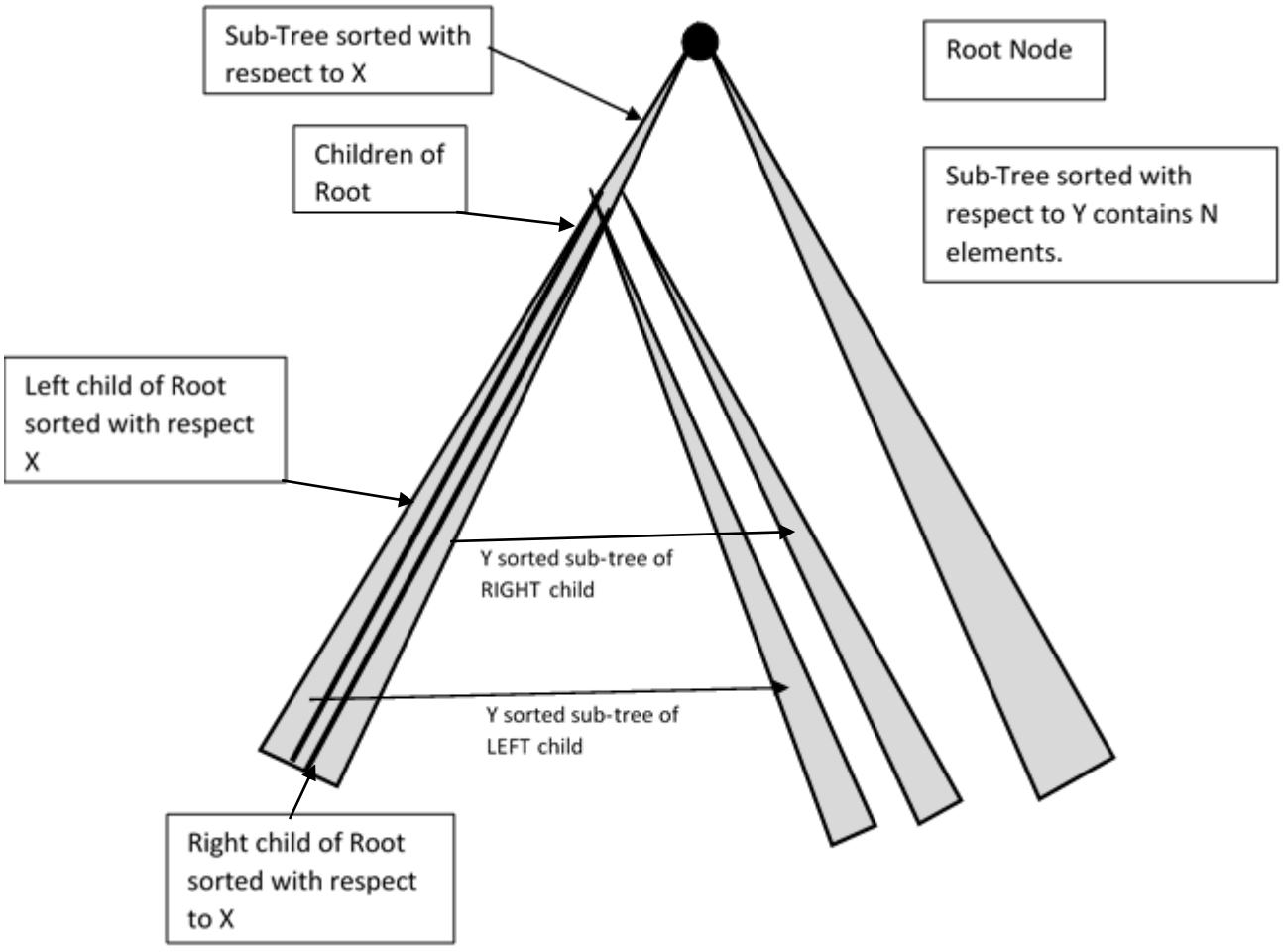
There are at most $\mathbf{2log}(N)$ such sub-trees possible in the path of X1 and X2 as marked in the diagram. The proof of this is simple as there as exactly $\mathbf{2log}(N)$ amount of nodes traversed while reaching X1 &X2. Now we see the problem that is of finding those points which have their Y co-ordinate lying between Y1 &Y2. By observation one can easily see that if we can get these points of these sub-trees that are marked in the diagram in $\mathbf{log}(N)$ time then We would be able to reduce the time to $O((\mathbf{log}(N))^2)$ which is actually as good as $O(\mathbf{log}(N))$ in practical purposes. We can easily see that we can in no way get this from the present fields that are available to us in the Data Structure. Hence we need to store some other field and one can easily observe that if we had a sub-tree which is sorted with respect to Y at the marked positions in the previous diagram then we would have easily got the time complexity of $O((\mathbf{log}(N))^2)$ because we would be needing only $O(\mathbf{log}(N))$ time to get the number of points in the range $[Y1, Y2]$.

This gives us an excellent idea that we can at each node store the pointers to the X sorted sub-tree and Y sorted sub-tree and use the Y sorted sub-tree after we have got the information that all the points in the sub-tree have their abscissa in the range $[X1, X2]$. This is our virtual data structure that will have a time complexity of $O((\mathbf{log}(N))^2)$ and the space complexity of $O(N(log)(N))$. This is evident from the fact that at every level in the tree we are storing $O(N)$ space. This can be proved because at the first node there are two extra fields

each of size $(N)/2$ then moving progressively to the lower level and we can see that this sum comes out to be $O(N(log)(N))$.

Each node is of the type **(X, Y, Count, LEFT_X, RIGHT_X, LEFT_Y, RIGHT_Y)** where RIGHT_Y and LEFT_Y represent the LEFT and the RIGHT children of the node which contain the same elements as the RIGHT_X and LEFT_X but sorted with respect to Y.

A pictorial view of the Data Structure is as follows:



IMPLEMENTATION:

PSEUDO CODE

We will use the function Query_Dynamic for the implementation of this data structure with slight changes.

```
Fast_DS ( Head_X, X1, X2, Y1, Y2)

{

    Node tmpX1, tempX2 <-Head_X;
```

```
While (tmpX1 or tmpX2 not = NULL){
        If (tmpX1= tmpX2! = NULL)
        {
                If (tmpX1.X>=X1 and tmpX1.X<X2)
                {
                        If (tmpX1.Y>=Y1 and tmpX1.Y<Y2)
                                Count ++;
                        tmpX1=tmpX1.LEFT_X;
                        tmpX2=tmpX2.RIGHT_X;
                }
                Else if (tmpX1>X1 and tmpX2>=X2)
                {
                        tmpX1=tmptx2=tmpX1.LEFT_X;
                }
                Else
                {
                        tmpX1=tmptx2=tmpX1.RIGHT_X;
                }
        }
        Else
        {
                If (tmpX1!=NULL){
                If (tmpX1.X>X1)
                {
                        Count=Count + Query_Dynamic (tmpX1.RIGHT_Y, Y1, Y2);
                        If (tmpX1.Y>=Y1 and tmpX1.Y<Y2)
                        Count ++;
                        tmpX1=tmpX1.LEFT_X;
                }
                Else if (tmpX1.X<X1)
                {
```

```
        tmpX1=tmpX1.RIGHT_X;
        If (tmpX1.Y>=Y1 and tmpX1.Y<Y2)
        Count ++;
}
Else
{
        Count=Count + Query_Dynamic (tmpX1.RIGHT_Y, Y1, Y2);
        If (tmpX1.Y>=Y1 and tmpX1.Y<Y2)
        Count ++;
        Tempx1=NULL;
}
}
If (tmpX2!=NULL)
{
If (tmpX2.X>=X2)
{
        tmpX2=tmpX2.LEFT_X;
        If (tmpX2.Y>=Y1 and tmpX2.Y<Y2)
                Count ++;
}
Else if (tmpX2.X<X2)
{
        Count = Count + Query_Dynamic(tmpX2.LEFT_Y, Y1, Y2);
        tmpX2=tmpX2.RIGHT_X;
        If (tmpX2.Y>=Y1 and tmpX2.Y<Y2)
                Count ++;

}
Else
{
        Count=Count + Query_Dynamic (tmpX2.LEFT_Y, Y1, Y2);
```

```
                    Tempx2=NULL;
                    If (tmpX2.Y>Y1 and tmpX2.Y<Y2)
                        Count ++;
                }
                }
                }
        Return Count ;
}
```

the output contains number of points inclusive of those at the lower boundary and exclusive of those at the outer boundary.

PRE-PROCESSING TIME:

This Data Structure is static an requires the pre-processing time of $O(N * (log(N))^2)$. This can be proven by construction. We can use one of the self-balancing binary trees and we can easily get $O(log(N))$ bound for constructing a data Structure which is the initially in the X-sorted form. Now we look at the time required to construct the second dimension sub-trees at each node. For constructing the two sub-trees at the root we will require $O(N * log(N))$ time

$O(N/2 * log(N))$ each then for constructing the four sub-trees at the second level we will again require time $O(N * log(N))$ time as each sub-tree construction requires $O(N)$ insertions. Similarly I go on and we find that maximum levels we can go is $log(N)$, so the pre-processing time is bounded by $O(N * (log(N))^2)$

PRACTICAL IMPLEMENTATION:

- First Data structure has been implemented using array implementation of Binary Tree.
- Static form of Second Data Structure has been implemented using Red-Black Tree in both the dimensions.
- Dynamic form of Second Data structure has been implemented using Skewed Binary Search Tree in Primary Dimension(X-Direction) and Red Black Tree in the secondary Dimension.(Analysis of this Data Structure has been done at the end of this report.)

PRACTICAL ANALYSIS

ANALYSIS FOR O(√N) APPROACH AFTER IMPLEMENTING

We have used random function to generate different values of N. Also for getting the query time we have taken average over 20 query for a particular set of input. Further, we have taken average over 5 inputs for each value of N.

CLOCKS_PER_SEC for the system = 1000000

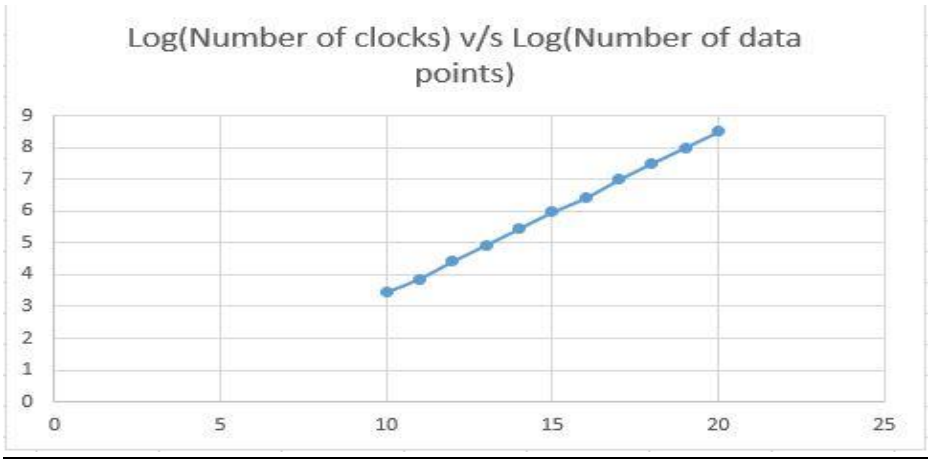
OBSERVATION TABLE

Number of Points	Average value of number of clocks	Ratio (clocks of N2)/(clocks of N1)	Theoretical Ratio
1024	10.9		
2048	14.6	1.34	1.414
4096	21.4	1.46	1.414
8192	30.45	1.422	1.414
16384	43.72	1.436	1.414
32768	62.3	1.425	1.414
65536	85.63	1.375	1.414
131072	127.8	1.49	1.414
262144	182.5	1.417	1.414
524288	259.15	1.42	1.414
1048576	364.6	1.405	1.414

Time = CLOCKS/CLOCKS_PER_SECS

If the complexity of problem is $(c\sqrt{N})$ then, if the value of N is doubled the time should increase by a factor of √2 i.e.1.414.

PLOT:

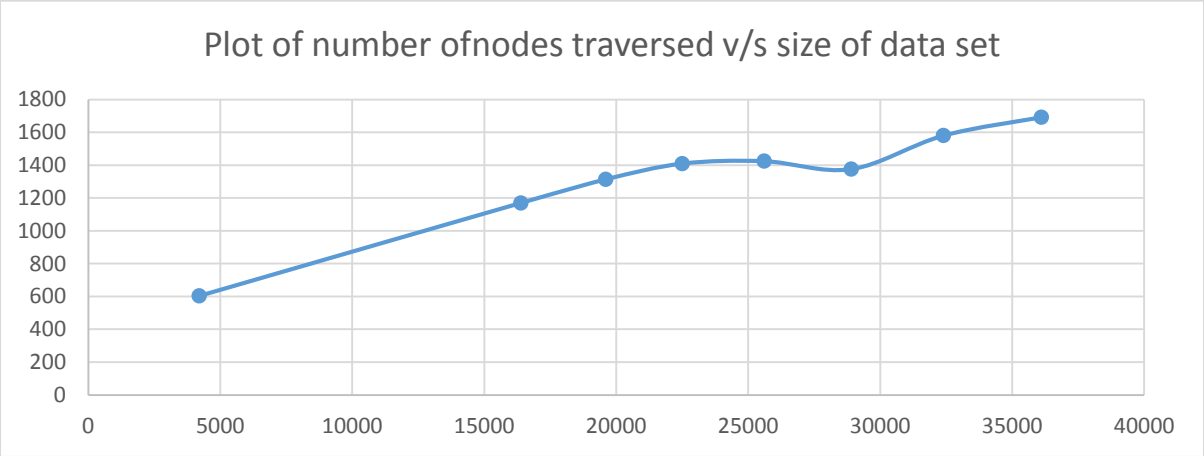


ANALYSIS FOR NUMBER OF NODES TRAVERSED FOR DIFFERENT VALUES OF N ON IMPLEMENTED CODE

OBSERVATION TABLE

Value of N	√N	Nodes Traversed	(Nodes traversed)/√N
1024	32	252	7.875
4196	64	603	9.4218
16384	128	1170	9.14
19600	140	1314	9.385
22500	150	1410	9.4
25600	160	1425	8.90
28900	170	1376	8.094
32400	180	1581	8.783
36100	190	1632	8.589

PLOT:



INFERENCE:

Number of node traversed by the data structure is of O (√N) as we can see that number of nodes traversed divided by √N gives almost constant value for wide range of data.

For the sake of simplicity, we can say that number of nodes traversed would never be greater than 20*√N. Thus practical application suggest that the algorithm is O (√N).

PRE- PROCESSING TIME ANALYSIS OF IMPLEMENTED CODE OF O(√N) APPROACH:

Intially two arrays, one containing the x coordinate and other containing the y coordinate are taken. We sorted this in terms of x coordinate (quick sort is used for sorting). Now the array was divide into two partition containing lets say a and b elements according to partition algorithm already mentioned in O(1).

Note that sorting in this context would refer to sorting in increasing order.

We can easily conclude that

$$a + b = N$$

Now the first a elements of array are sorted w.r.t to y coordinates and also the remaining b elements are sorted w.r.t to y coordinate.

This will take

$$alog(a) + blog(b)$$

to find the upper bound we can say

$$alog(a) + blog(b) < (a + b)\, log(N)$$
$$< Nlog(N)$$

We will continue in this dividing the partition and sorting w.r.t to x and y alternatively. For a general case if it has k partitions, $(a1, a2, a3 \ldots . ak)$ we can say the time taken

$$< (a1 + a2 + a3 \ldots . ak) log(N)$$
$$< Nlog(N)$$

If for simplicity of analysis we take value of N to be of the form 2^{2^k} then it will for a complete binary tree and the number of level would be $log(N)$. Thus total pre-processing time would be of $O(N * log(N))$.

ANALYSIS FOR O(LOG²N) APPROACH:

We have used random function to generate different values of N. Also for getting the query time we have taken average over 20 query for a particular set of input. Further, we have taken average over 5 inputs for each value of N.

Since the complexity of this is $c\ (logn)^2$, if we take value of n to be in the power of 2,n1, lets suppose 2^k ,then take another value of n to be $2^{k+1}, n2$ theoretically,

$$T2/T1 = (\log(n2) / \log(n1))\text{^}(2)$$

$$T2 = T1 * ((\log(n2))/(\log(n1)))\text{^}(2)$$

$$T2 = T1 * \left(\frac{k+1}{k}\right)^2$$

OBSERVATION TABLE

Value of N	Avg. for 5 values (CLOCKS)	Theoretical value $T2=T1*(k+1)^2/k^2$
1024	1.7625	
2048	2.2125	2.133
4096	3.7125	2.6331
8192	6.46	4.351
16384	8.725	7.492
32768	11.825	10.02
65536	15.425	13.4542
131072	18.5625	17.413
262144	22.8	20.811
524288	26.55	25.404
1048576	30.4	28.7

Time = CLOCKS/CLOCKS_PER_SEC

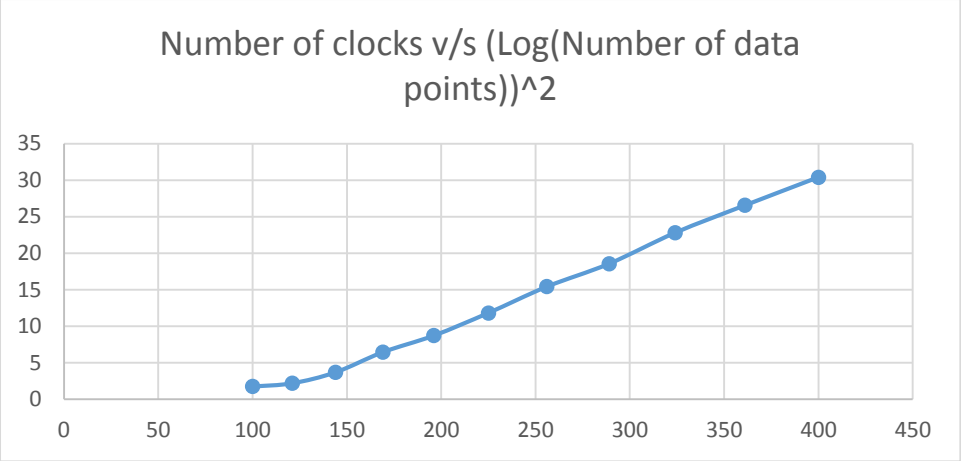
CLOCKS_PER_SEC for the system = 1000000

In the last column Theoretical value is calculated using the successive levels.

INFERENCE:

We can see that value coming is almost equal to the value calculated theoretically taking $\log^2 n$ to be the order. Thus, it can be inferred that the time complexity of the implemented data structure is O ($\log^2 n$). The plot of Number of clocks v/s $(Log(Number\ of\ points))\text{^}2$ is linear.

PLOT:



EXTENSION TO DYNAMIC FORM:

The second Data Structure can be extended to the dynamic form. But the update operations are not so straight forward as they seem to be at first glance. The problem is with the implementation of the Static Data Structure which due to the fact that a Self-balancing Binary Search tree uses rotation of the tree in that case if we directly apply rotations then this will lead to change of Y-sorted sub-tree at each node involved in rotations and one can easily observe that we will require to merge two arbitrary Binary Search trees that will $take\ O(Sum\ of\ the\ sizes\ of\ the\ two\ trees)$. But one point to note is that this problem lies only in the Primary dimension. Hence in the secondary direction we can maintain a red-black tree. Another point to note is that if we do not rotate the tree then we are done but this will lead to skewing of the tree. But for random data we can practically observe that the tree is near to nearly balanced Binary Tree order of insertion is approximately of the order $(\log(N))^2$ and the query time is also $(\log(N))^2$. The worst case order is $\sim N^2$. But for random values this Data Structure works as good as the Static Data Structure.

Observation of Time Complexity after practically implementing the data Structure.

Problem generated for further motivation in this field: This problem leads to a curious question that does there exist a Data Structure which can answer the Range queries in $O(log(N))$ and does not require rotations for its balancing.

ANALYSIS OF TIME COMPLEXITY FOR DYNAMIC VERSION:

Our dynamic version involves the simple binary tree which can be skewed depending upon the input values. But we have already seen that a binary tree performs similar to balanced binary tree in case of random input. Our approach works well for the case where there is random input.

Observation Table

Value of N	Avg time (CLOCKS)	Theoretical value $T2=T1*(k+1)^2/k^2$

2048	2.70	
4096	3.4	3.21
8192	4.65	3.99
16384	5.26	5.39
32768	8.34	6.04
65536	10.63	9.48
131072	13.05	12.00
262144	15.25	14.63

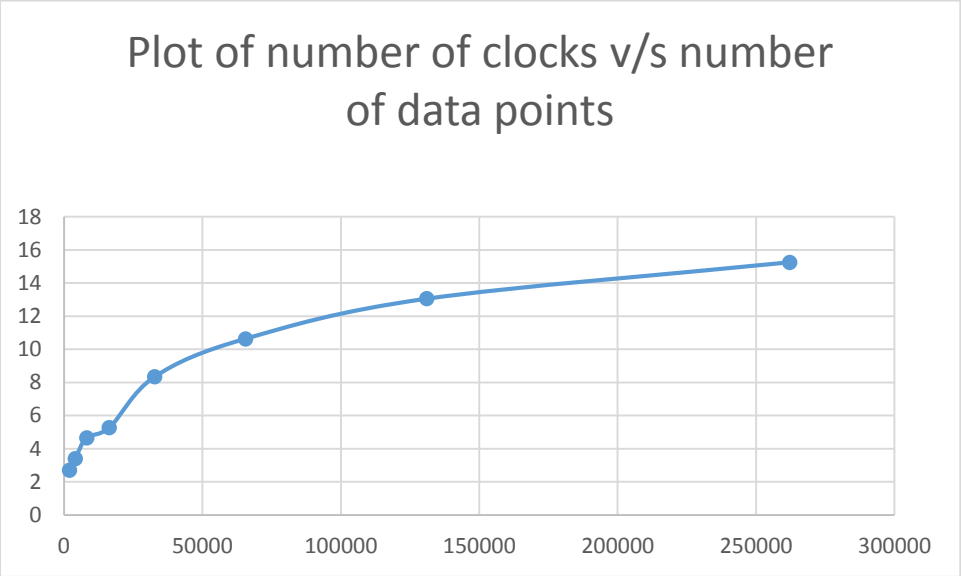
Time = CLOCKS/CLOCKS_PER_SECS

CLOCKS_PER_SECS for the system = 1000000

INFERENCE:

We can see that value coming is almost equal to the value calculated theoretically taking $\log^2 n$ to be the order. Thus, it can be inferred that the skew tree is performing the update in O ($\log^2 n$) for random input. So, our approach would work well if input are random.

PLOT:



COMPARISON OF BOTH THE DATA STRUCTURES.

Number of Points	Clocks for $(Log(N))^2$	Clocks for √N
1024	1.7625	10.9
2048	2.2125	14.6
4096	3.7125	21.4
8192	6.46	30.45
16384	8.725	43.72
32768	11.825	62.3
65536	15.425	85.63
131072	18.5625	127.8
262144	22.8	182.5
524288	26.55	259.15
1048576	30.4	364.6

INFERENCE :

We can easily see that the $(\log(N))^2$ data structure significantly(by a factor of 10) outperforms the $\sqrt{N}$ data structures as the value of N approaches 10^6 and for the number of points less 100000, both the algorithm works fine as approach 1 takes more time but not much (0.000128 sec averagely) and space is O(n). On the other hand approach 2 takes more space approximately $(17 * N)$ for the number of points less than 100000. So, for these many points space is not a matter and with respect to time, this obviously outperforms approach 1. So for these many points both the approach works good(preferably approach 2 as its reduces the time to very less (0.000017 sec averagely).

But for the number of points above 1000000, you will have to choose between these 2 algorithms depending upon your need and constraints. Though, the approach 2 give result in very less time but at the same time space starts creating problem because $log(N)$ factors start becoming significant.

- If space is a problem and there are not more queries then we should prefer approach 1 for the values of N >10^6.
- If space is not a problem and queries are significantly large then we should prefer approach 2 for the values of N > 10^6.
- If N< 10^6 and queries are large then we should prefer approach 2.
- If N< 10^6 and queries are not large then we should prefer approach 1 to save the space complexity.

For number of points above 10^8, you should clearly favour approach 1, O(n) space , as the space required for this by approach 2 would cross the space chunk provided by the RAM, thus you will have to access your external memory ,which would take more time to give the result.

GUI – IMPLEMENTATION

We have implemented the static Data Structure of $(\log(N))^2$ complexity in the GUI. Both the Data Structures can be included in the GUI, but for simplicity we have implemented for $(\log(N))^2$.The GUI has been made Python language using the Tk-inter library and Matplotlib.pyplot library by embedding the graphs generated by Matplotlib.pyplot library into the Tk-inter window. The features of the GUI are available in the "Read-Me" given in the directory containing the GUI codes. Few looks of the GUI are as follows: