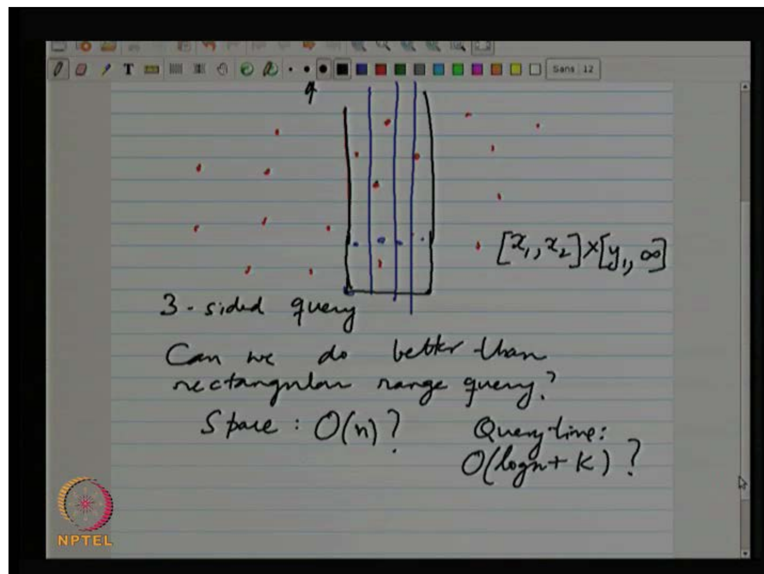
**Computational Geometry**
**Prof. Sandeep Sen**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Delhi**

**Module No. # 11**
**Range searching**
**Lecture No. # 03**
**Priority Search Trees**

So, we continue with what we started in the last lecture that is we started on orthogonal range searching, for which we defined two kind of data structures k d trees, k it is actually you know very universal and very general. And then we describe something called orthogonal range searching trees, which have a somewhat faster, not somewhat substantially faster query time, but at the expense of a little bit of blow up with the space. And I mention that you know, we despite all that you know in practice, you still prefer having something that is linear space, because space is and overhead that you have to carry all the time.

(Refer Slide Time: 01:17)



And then I pose this problem of searching, so given a set of points, if my query is not in axis line rectangle, but something there is, that is infinite in one side. So, it is a rectangle, it is say which (No audio from 01:38 to 01:48), which is where the top side is opened essentially, this is infinity essentially unbounded in this direction. So, when we have this kind of queries, it is also called as three sided query (No audio from 02:00 to 02:07) and we can certainly use our orthogonal range searching data structure to support this query. So, there is no problem, I can I can always put a bound on the upper side, which is way above this set of points, and I can answer that query using the orthogonal range searching data structures range query range trees.

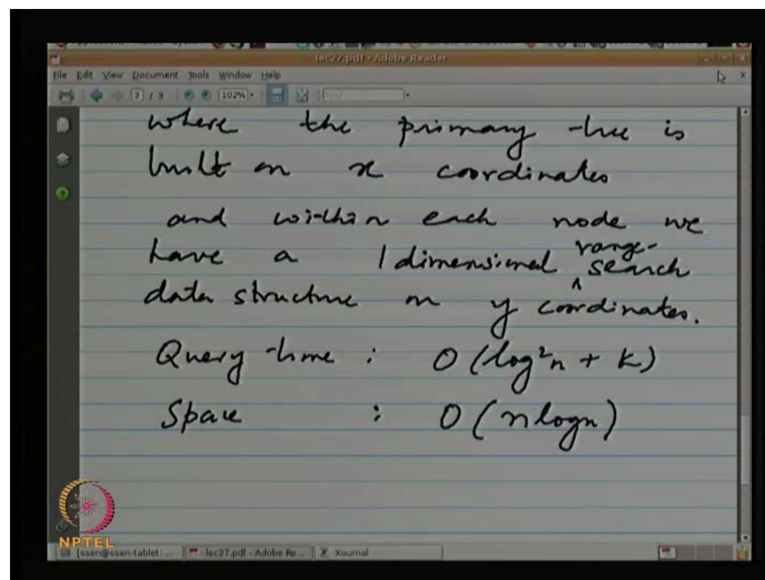
But the question here is at since this object is somewhat simpler, not too much simpler. So, it is not as simple as you know, so you remember when I try to explain it is exposed idea of the of the range searching tree, we talked about in infinite slab in both directions; that turned out to be kind of a one dimensional problem, but I did not want to use the one-dimensional solution, I actually use the k d tree solution to to illustrate the use of this, you know how do you how do you how do you look upon this as the union of the canonical intervals. So, this kind of query was handled using you know, (( )) range search tree, which is nothing but essentially you know look at these set of canonical intervals and some of the queries that you come that that that you get from each of these individual disjoint slabs. So, the range search data structure should be viewed upon as we where we take the x axis. And we can we can take any interval in the x axis and express it as a union of the intervals express by some of the nodes. And those are the once at a time, I calling canonical intervals and it can any interval can be express as a union of about two log n canonical intervals. This each each of those interval is represented by some kind of a node in the in the in the range search tree, but here now I am actually not talking about the about the unbounded slab, but I am actually bounding the slab in one direction.

And now, I I want to ask the question that can we do better than rectangular range in terms of both space and query time. (No audio from 04:43 to 04:49) So this is nothing but if the query is x 1, x 2 Cartesian products with y 1 infinity sorry y 1 gamma, so more specifically can we reduce the space linear and can we have the the query time to something like order log n plus that is a question I am posing. (No audio from 05:24 to 05:30) If you had some of you (( )) some of you (( )) some of you in the weekend geometry school and in one of the talks, I think Pankaj did make

an illusion to something called this what is the the way of searching, where you can search in more than one sorted list by using a data structure technique called fractional cascading.

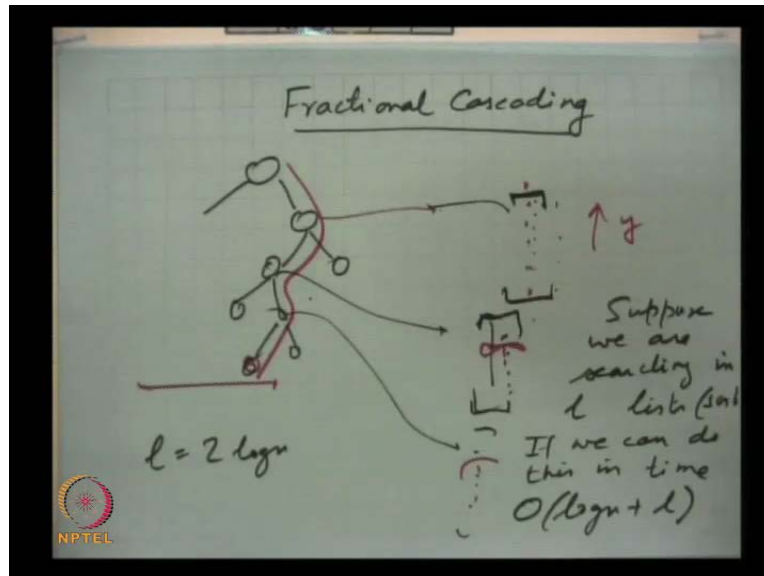
So, if you use fractional cascading so, in fact fractional cascading historically you know it it it was discovered in the context of answering rectangular range query, where the query time. So, what we discuss last time given this, we just quickly recap.

(Refer Slide Time: 06:21)




So, this is basically what we had proved the range search trees gave us a log square n plus k query time and n log n space. So, it is possible to improve this one this write on this. It is possible to improve this log square n to log n using fractional cascading. If just, you know data structure we come within more combustion. You have to build another layer of data structuring on that reduce from log square n to log n again is possible; however, the space remains the same. So, when when use fractional cascading so, maybe I should mention about a fractional cascading again. (No audio from 07:03 to 07:13)

(Refer Slide Time: 07:06)



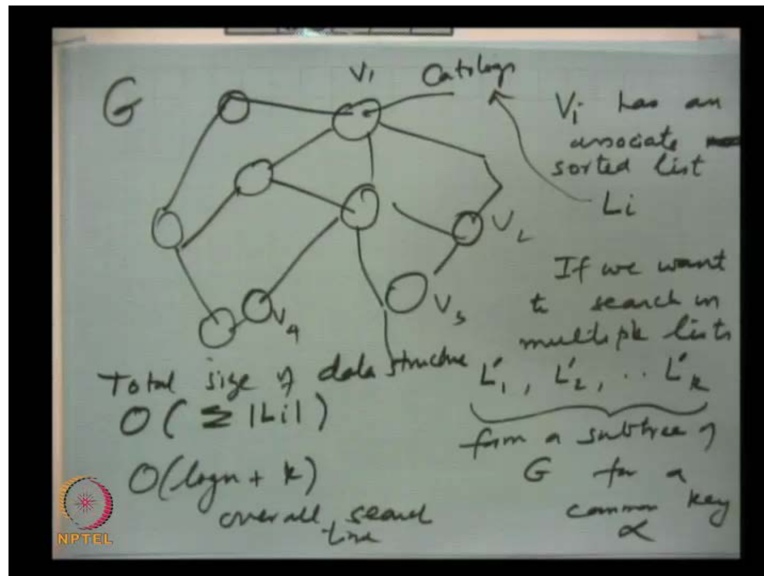

So, if you so the overall scheme or the overall technique of fractional cascading is quit general. Actually so the way we require it here is that we have is sequence of list. So, if we look at the range search tree. (No audio from 07:30 to 07:37) The range could take some particular path in the tree. And then we will do binary search in y direction because here, we are expressing some kind of interval using the the primary range search tree, but with each node you keep the y coordinate sorted. So, within each node, if I blow up this node you keep the data in one dimensional data structure, where the points are sorted in a y direction. So, that you do a binary search or something similar to binary search, where you can answer each query in time proportional to log n plus a number of points inside.

So, with the each node we have this kind of data structure. So, this and and we are looking at about two log n nodes one for the right path, one for the left path. So, you end up doing this query on order log n nodes and therefore, you get that log square n plus three numbers of points to be reported. Now, all this sorted list you know, that I maintain here I have another sorted list here I have another sorted list here. So, if I can somehow combine the binary searches of this this all the list that occur in this path then and I and (( )) spend then, what happens that see to locate the initial point to take log n time. After that we basically walk through and report the points. So, the initial binary search that is important. So, if I can if I am searching in k lists suppose, we are

searching in one lists, one sorted lists. So, if we can do this in time order log n plus l. I claim that you cannot hope to do anything better than this.

So, I I just paying for one binary search, I have to do once search. I cannot avoid that and some of based on that information. I am able to locate my initial point in each of the list. So, here also I am able to locate, where I start from here also move at start from and paying only constant among additional time for each sorted list. So, then and for in this particular context we one is basically two log n. So, then your search time becomes order log n plus of course, the number of points reported number of points reported only once, because we have ensured that the list over the different levels are disjoint. So, no point is reported more than once, even if it were reported. It can be reported twice or thrice, you still have been go of that so, this is the way fractional cascading technique is used to combine binary search over you know over a (( )). So, not just one list, but you know multiple list. And in general, since I brought up this topic.

(Refer Slide Time: 11:02)




It is even more general, actually what one can do is you can have an underlines. So, there we are talking about a tree data structures, primary data structure was tree. And the lists were basically the nodes of the tree. Now, you can think about this tree has special kind of a graph, but the graph is a tree. Now, what happens, if I have a graph structure just underline graph structure, I have an underline graph structure you know, which can be anything, which could not even

necessarily a cyclic graph. I could have any kind of structure. And I have not not planar not acyclic (( )) some some underlining graph structure and then with each node know. So, let us call the let us called the vertices, you know V 1, V 2, V 3, V 4 etcetera. So, V i has an associated node sorted list call it as L i.

So, in this structure, and if you want to do want to search in multiple lists. Now, the multiple list let us call them you know L prime 1, L prime 2 some L prime k. Now, we the only restriction here is that L prime, L prime, L prime 1, L prime 2 all that L prime k they are not arbitrary nodes, but they actually they form a connected sub graph well connected sub graph essentially means a connected tree. Because you know y should we have a sub graph, because if I search once in each node that is that is sufficient essentially the only restriction with this form is sub tree sub tree of so, if I call this underline graph G it form a sub tree of G. So if I have this restriction that the set of nodes that we are going to search for is the sub tree of the original graphical structure then you knows, we can apply this technique of fractional cascading build some data structure on these and you can get this kind of performance of log n plus k over all search time, what we are doing, we have a common key to search, if you want to search for a common key alpha. The keys are common; it is a same key that we searching. So, when we search in multiple lists, of course we are not searching for different keys, which are same key that we are searching for and we do not have to do independent binary searches.

You know we can build a data structure on this underline underline graph these are also sometimes called catalogs. So, these lists are also called catalogs. So, we can we can search through this whole set of catalogs. The k catalog we can search in log n plus k. And this version of the fractional cascading much more complicated than what happens in the trees, because in trees more or less the technique is that you merge these. You know the list of the… You know of the lower most level with the… you take a fraction of those keys. You know it is like you know taking every other key and and and pushing it up. So, that you know the gaps are not too much you merge them basically, but at the same time you maintain the… So, one one of course, requirement here is that total size of data structure should not be super linear. So, it should be order you know the sum of the size of the individual list.

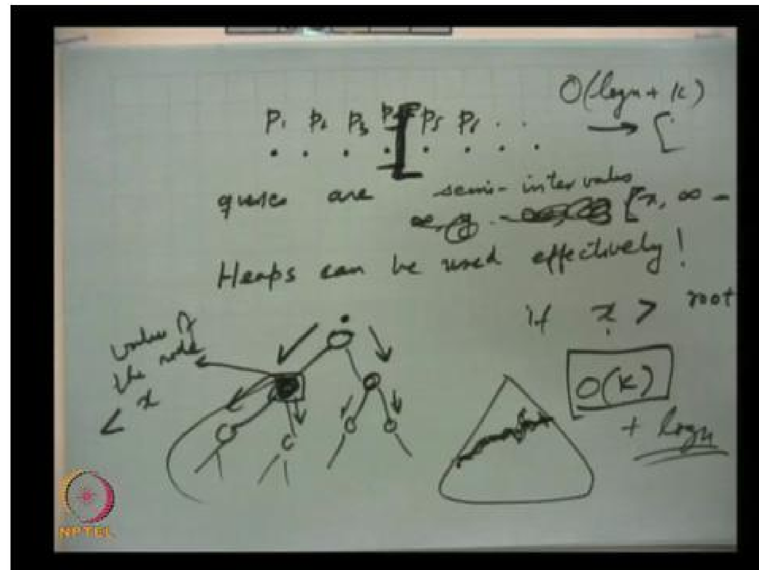
So, fractional cascading achieves this logarithmic bound the out blowing of the space too much. In the case of the… You know where whether the graph is a tree. It rather simple because we just we just take the lists of the lower most level merge. It take take every alternate key merge it, then of that take every alternative merge it. So, it is like a you know it is like a geometric series overall the space remains linear. The challenge in more general situation is that you know it is it is because there actually going to be actually cycles. So, the just that selecting every other key it is not going to work. So, it becomes you know very very complex, but there are some simple solutions also (( )) randomization. So, I do not want to discuss it too much here, but someone interested follow it up will be happy to do will be pointers.

So, the only reason, I mention this fractional cascading without getting the details many of these algorithms in you know for for range searching. And even for segmentries kind of data structures, whenever we slab this log square n in many cases. We can actually pose it has a fractional cascading problem and reduce the log square in to log n. And the the advantage is not so, much in practice you know log square n log n you know who (( )), but the real advantage. The real value of this is the theoretically, you are able to match the optimal bounds that is what is the real value of it. So, fractional cascading is hardly used in practice because it is it is really complicated, but this is simpler than main construction for this.

Let us get back again to this problem of quality search trees, where were somewhat restricted version of the rectangular query. We have a rectangle that is infinite in one direction so, it is called this three sided query and question. We are asking is can we do something better than, what we achieved using the straight forward range search trees that is from log square n, we I want to bring it down to log n and from the n log n space we want to get it down to alternate.

So, what you (( )) feeling because because brought it up surely something is possible. So, even before I get in to this let me make a very elementary observation about something some of you may be aware of, but I am not sure it it it it is it is a something simple that often gets (( )) or you know we do not pay enough attention to that.

(Refer Slide Time: 17:58)



So, I will pose a related, but simple problem; and that is some given a set of points. (No audio from 17:51 to 17:58) And I I want to pose the one dimension problem, and I have a my queries are let us say again semi intervals, let say infinity to sorry x infinity x, which I which, by which basically you can I could get queries like this. So, this is my query. So, when I get this query suppose these points are P 1, P 2, P 3, P 4, P 5, P 6 etcetera. So, for a query like this, the answer is P 1, P 2, P 3, P 4. It is it is a very simple one-dimensional problem. There are so many methods of doing it essentially again base and binary search. But there is another way of solving it, and may be you can just think about it for a moment. So, I claim that heaps can be used effectively, and why would be prefer heaps over. So, what you could just basically sort the sequence, and do the binary search and be done with it. And I want to again achieve a query time of order log n plus k that is my goal linear space or the log n plus k. So, why would we prefer why would we even bring up this heap?

(( ))

No, I am talking about static data structures.

(( ))

But you know my first we already giving me the solution. So, solution probably is also obvious, once I mention the heap, but why would I look for a heap? Why would I look for a heap base solution? What is their advantage of heap?

(( ))

Something more fundamental.

Yeah yeah yeah exactly right. So, I do not have to sort, I can build a heap in linear times. So, always remember this this important distinction that within search trees and heap. In terms of construction time in heap, you do not have to sort it can be built in heapified in linear time. So, wherever trying to really safe time in the pre process you can use heaps and how would you use heaps here, what kind of heap would you like to use. Some of you may be know the solution but so, let me just. So, when you build a heap again it is a it is a (( )) like a complete binary tree. So, what kind of heap would you build on these points max min whatever what so, should I turn around the problem actually because the way I pose the two dimensional problem was the upper one was sorry this is this is actually minus. I am sorry we should turn it around because this weight is increasing.

So, then I should actually say x comma infinity sorry. So, my my semi intervals are like this so, that way then we are saying main, but here if we (( )) use a max seek. So, the one with the highest coordinate it should be should be here and and so on so forth. And what is advantage you know if this point so, what what what we are going to do a how how how you are going to search the search the search the seek repeat again.

(( ))

Well delete means (( )) what we are not deleting we are searching. So, we have built a heap on this set of points. Now, we get this query this semi interval query, how you are going to answer the query?

(( ))

So, first we compare whether this x is a less than or sorry greater than this the this element. We build a heap on only on the x coordinates so, if x is greater than the the root value then there is

nothing to do will right. I mean mean that the no point can be within the set problem, because this is the highest thirty is the highest value. And if my x exceeds that my interval is somewhere here I (( )); obviously, not report any points. So, only if x is less than this point then we will report this point will report the point and explore if they more points to be reported. And now, here we are not going to walk along a single path, but we may have to branch for this, because it is a heap (( )) it is the heap.

So, the next two points are something that again I am going to compare with the x same strategy again if if the values here are smaller than x. Then this entire sub tree will have values smaller than x. So, I did not go down in the sub tree at all, but now it could be larger than x. In which case I again go down, when I go down, I have to traverse, I have to try both paths. So overall what is the search time do I require for, which is the strategy like this say again order k so, why it is order k.

(( ))

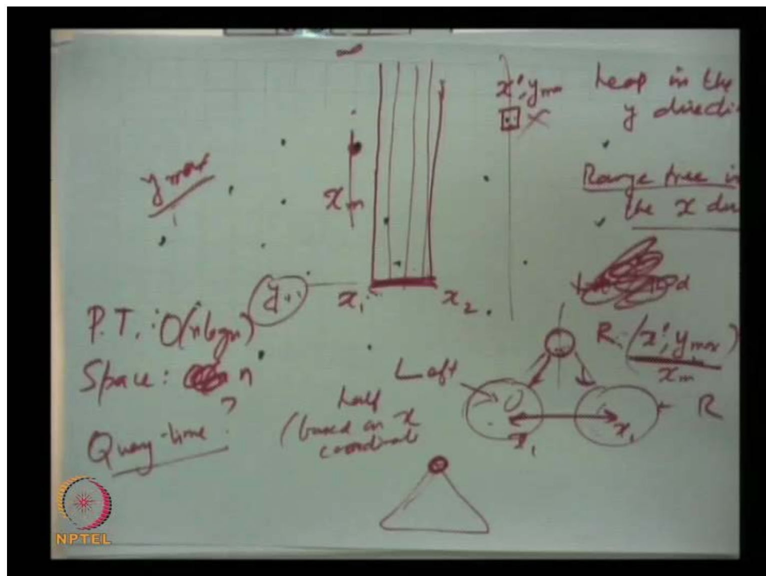
So, when we stop at a node, it basically means the value of the node is less than x. So; obviously, the sub tree did not be looked in to all those points will be smaller than x. I did not look in to it, but even this value may not be reported. So, we may have visited this node, but not reported any thing. See we are you know the overall way this range searching works is that you know, when we visited node it is like it is a charge you know I I encounter some charge, because I am paying the (( )). Then if I report some point then you know that cost is (( )) reporting the point, because I am I am I am reporting a point. So, it is meaning full or you know it is we are made a successful visit that we are reported a point.

And we are pay some constant cost for that, but here it may happened that we we visit this node. And we do not report any point, but then how many such nodes can be there, where we do not report any point. Total off in in well there are so, see this is the overall heap. Let us say, we you now you can actually kind of argue that you may you may stop your search at some nodes, when the when the depth will be you know some you know some kind of you know may be it is you go down little bit more, because it is a heap you know may be you go down, note down this path then this path. So, you have stopped at all these vertices of the of the heap and before this level you certainly report it something.

Otherwise you would not have gone down to the level. So, what is the extra number of nodes that you visited, where you know you did not report anything at most twice the number the previous level. So, we are not going to visit more than have in the number of nodes that you do not report any thing is no more than two times the number of points reported. So, that can be subsume so, therefore, it is order k and the good thing about this is, that we do not even have an additive log n term do you realize that. You did not even require that addition order log n term (no audio from 26:41 to 26:49) when you do a binary search you encounter that log n and then of course, you walk and report the points. Here, I have actually been able to not only build the heap much faster.

I am done it, in linear time my reporting is also only proportional to the number of points. And this is actually you know is fairly something that we surprise we may not expected this. So, heaps turn out to be a more effective data structure for one dimensional range searching. So, now you want to exploit this observations and all what do we have probably might have is (no audio from 27:39 to 27:47)

(Refer Slide Time: 27:53)




So, we have this set of points (no audio from 27:49 to 27:58) and we have a query like this. Some query is my three sided query x 1, x 2, y1 infinity. So, I could, but if this is the are two

dimension problems well at least more than a one dimension problem. If we use the range search tree, I mean use the technique of the range search tree again. We should thinking about splitting of this x 1, x 2 and two some union of canonical intervals put it through that tree and will get union of canonical intervals, but we need to do something more clever to be able to achieve. So, at every level, we are doing a binary search on the y and therefore, we encounter that log n plus sorry log n plus a number of points reported and multiplied by the number of levels. We got that log square n can we somehow now combine the ideal the heap and the and the range search tree.

(( ))

A heap at every node again sees, we do not even want to have a blow up this space. So, we do not want have a heap at every node, see one of the reasons that space blow up was there is a primary data structure. And this is the node there is another data structure and somehow this summation of everything was not linear. So, will will have have to combine will combine that heap and and this thing, but you know we have to do it.

(( ))

So, we need the heap coming from the way I drawn this picture. We should think about a heap in the y direction, because in the y direction, we have this semi interval heap in the y direction. And some kind of idea about this, you know range tree in the x direction, this range tree in the x direction may be better thought of as may be the one dimensional k d tree actually may be that is a better. We are thinking about if a k tree is a one that gives you also this I mean that the canonical intervals. So, whatever either you think about may be (( )) let us let us not confuse too much range tree in the x direction and heap in the y direction. And we should store the points in a ways such that a node should not contain more than constant number of points only then we can achieve the linear space.

So, how do you do it so, we are we have this this infinite (( )). It is like a it is like a well structured you know so, if the point in this data base are such that the highest of the y coordinate is lower than the bottom then we do not need to search the data structure at all. We do not need to go any further, if you find out that the value of y 1 is greater than the y max the y max let us say is the y largest y coordinates among all the points.

(( ))

Will do that will effectively will do that yes now, but when we do the heap on the y, how (( )) to combine the the the crucial question is, how are you going to combine the x and the y. See that vary fact that we are talking about the range tree in x direction means that will have them in sorted in the x direction no doubt about that.

(( ))

Exactly so, that precise what I am developing now. So, that what I am claiming is a root of the tree or this data structure should contain the y max the the point having the the corresponds to the maximum y coordinate. So, you know, if the if the set of points may be this is the point with the highest y coordinate. So, it is that whatever you know y prime y max or something. So, this one I propose we should make it is a root of the tree so, this one should be the root of the tree. Now, what is what mean just articulated was if first first first thing we do, while we do the range search is compare y 1 with the y coordinate of this point. So, this point is that y prime sorry x prime, y max, we stored that point here.

So, I compare the y 1 with y max, if y 1 is greater than y max; obviously, anything because all the points whatever they are they are all greater smaller than y 1. So, the the the output of the range search is an empty set. So, we do not need to go any further and look for points we are done. Just with the basis of why we are done, because the base itself is higher than the reset to the points. We we may not be that lucky, but a in some cases we did not go further, but this condition will hold for will try to mean this condition hold for any node basically. Once we find that the base is as a larger x coordinate. So, for any sub tree suppose this (( )) data structure that any sub tree, we should have the largest y point with a largest y coordinates (( )) root.

So, that we can always judge, we can always decide whether or not, we should be searching that sub tree. So, this kind of invariant will maintain all the time. After we have we have we have define the root of the tree now, we do the division on the x side. So we would like to actually achieve the kind of the balance tree. So, look at the medium of the y y coordinates and define so, this is half. Let us say the left half call it L left half based on y coordinates sorry x coordinates and this is the right half again based on the y coordinate. So, this is the set of points now, do not

compare the x coordinates with the x coordinates of this now x prime and this left half may not have any relation.

This x prime y max you know could be you know one one side of it, you know so, this one I am seeing this is already it is to the right may be it is also the right most point who knows. So, this point may not divide this x coordinate may not divide the points equally in to two half's, but I have stored in the root that point is gone I do not need to so, this point is not going to be consider for the further data structure. So, will eliminate this point now, for the remaining points, I look at the median of the x coordinates and put and recursively will construct the data structure. So, I have got the left half of points here and the right half of points here. And this does not include this point, but what will do in addition in the root, we have this point will also store the information about the median.

So, what is the value, may be this is the median value, this point may have the median value x m or somewhere that. So, we will also store an addition to this some value x n. So, that we know that, if x 1 is to the right of x m, we did not go and search this point set of or if that x 2 is to the left of x m, we did not go and search the set of points, but in some cases we may end up this x 1, x 2 may actually span the whole thing. So, x 1, x 2 may force us to take both the paths that would also happen. So, the interval could actually get split either adjacent node or then recursively will the data structure for this set of point and this set of points is using exactly the same observation.

So, I will I will choose as a root of the left sub tree, the point with the highest y coordinate among all the points in the left tree. The root of the right sub tree again will be the point that has the highest y coordinate among all the points in R and that is how basically this tree will be is going to be get bit. And simultaneously, you see that when I search like you range searching this interval may get splitted at the root or it may get (( )) one side at some point. It may get split and it something may be the forking node. So, it is like range searching, you know it is going to move down on the basis of the (( )) x coordinates, but simultaneously we are not going to explore any sub tree, where we already know the the node has the lower y coordinate than y 1.

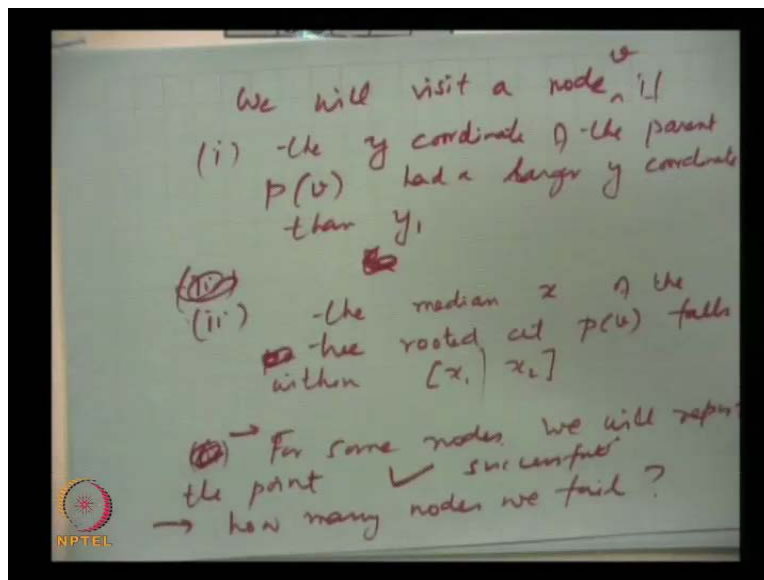
The moment we find that y 1 has a larger value than the y coordinate. Here it means that all the other y coordinates are also lower therefore, we do not explore that that is (( )). So, can you now tell me, when we do the query. So, is that clear the construction scale, what is the space in the

construction a point is stored exactly once space is n, point is stored exactly once and implicitly, we are sorting it because every time we are (( )) we are we are we are find the median and therefore, in the end all the x coordinates or at least we we are ended up running around median on all these levels. So, we are kind of sorting the points. So, we have it is it is different, because this point is taken how before it is sorted, but almost sorting point (( )) may be we are sorting half the number of points are similar.

So, it is certainly n log n and no more than n log n. So, pre processing time pre processing time is n log n space is order n, because not only each point is stored exactly once of course, plus the whatever the tree at the tree itself will take some pointers. So, that is why order n now, what is the query query time that is important thing, how many nodes are we going to visit first of all, how many nodes are going to (( )).

(( ))

(Refer slide Time: 39:22)



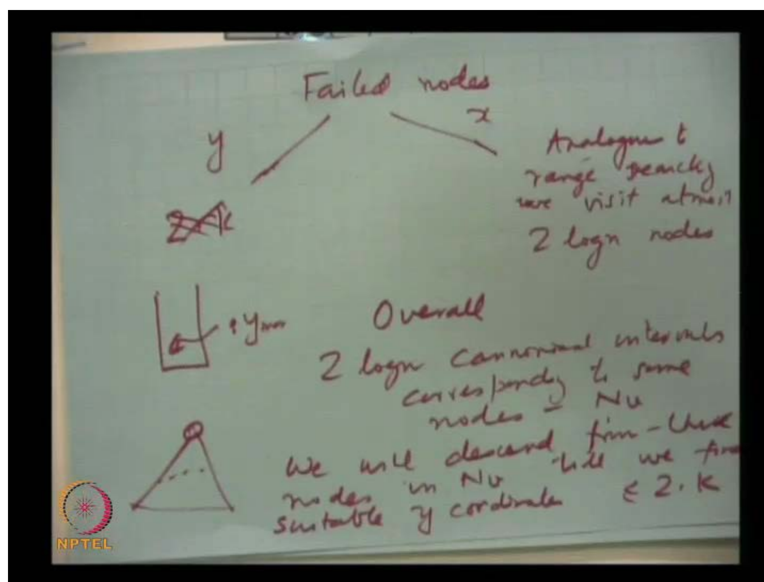

So, let us let us classify the node that we want to we will get will visit a node. If the many situation for visit a node the y coordinate of the parent node v of the parent let say P of v had a larger y coordinate than y 1 that is why went down y 1 was smaller than the y coordinate of the

root. So, we went down to that the two children so, that is one reason why we may end of visiting a node.

(( ))

So, I am I am just see that and and so, let me treat it separately. Actually that can be treat at once, but let me just over count little bit or the median x of the parent well median x of the tree rooted at the parent P falls within the interval x 1, x2 the median passes through some where here. So, the entire interval is not contain to one side so, this is one thing and there are of course, some of these nodes so, for for some nodes for some nodes we will report the point, which are basically successful nodes and there is no problem in charging it the problem is for some nodes. So, for how many nodes we fail. (No audio from 41:52 to 42:07) you are saying 2 k for the y coordinates. So, if there key points to be reported from a analogy of the heap. We can visit 2 k that is a nice way of counting. So So, you are you are looking at two kinds of failures failures failures in y n x.

(Refer Slide Time: 42:30)




So, successful of the once we do not need to bother about so, failed nodes are of two kinds one for the y and one for the x. So, for the a y what is being said is we can fail at most

(( ))

2 k two times number of points reported by the same argument as the heap. The parent reported something therefore, we went down, but it is not clear the parent should be reporting, because there is also a y x component

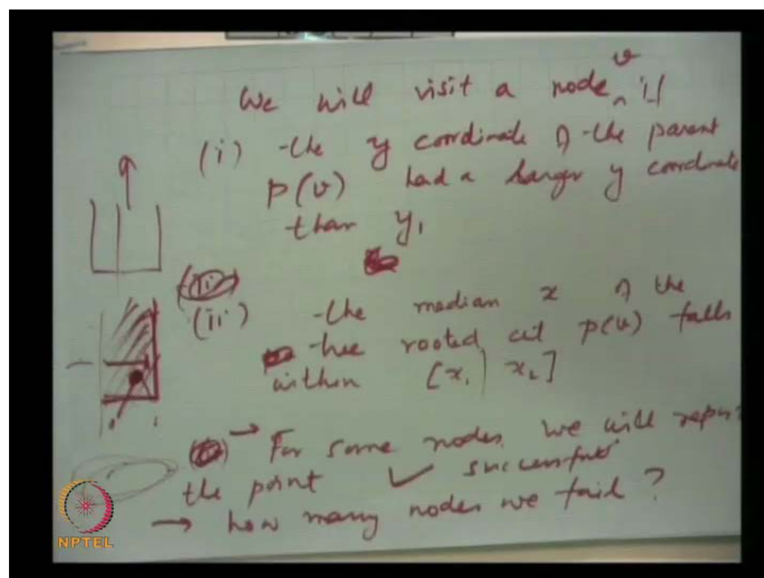
(( ))

So, the it could happen very good that is that is a good point. So, you have a situation like this, where the y max is here, but this point itself may not be inside. So now, it is it is slightly more tricky to (( )) your intuition is correct you know. So, how how should we count them?

(( ))

So, let us first look at the x (( )) how many how many nodes can be fail on that is why the x is concern. (no audio from 43:44 to 43:52) Now, use your your your range search tree (( )) so, we are we are no we are visiting no more than two log n nodes in the x direction exactly so, analogous to range searching we visit at most two log n nodes. So, even whether we fail or not you know it can added at most log n such things order log n such things. And I did not fully specify the search. So, this is how I was specifying the search.

(Refer Slide Time: 45:07)



So, we should actually you know like another range searching. If you find that you know for a sub tree for a sub tree then you know the range I am searching is split already. So, let us say, you know this this was whatever searching and and some point it split. And then now, we are only looking at not only infinite in the y direction, we have also split this so, we are actually looking at situation like this essentially after the fourth node. We are looking at situation like this because the other part is gone to the fork node. So, once it is fork, you the (( )) like this and if this entire region is contained within the within the sorry the this this this the intervals spanned by this node is contained completely within then we did not go down this thing is the is the is the way that you know we define the canonical intervals.

So, the the interval denoted by this x interval denoted by this may be up to the here, which is completely contain within this. So, we did not go down any further and in the in the range search tree, what did we do, we stop at this node. We did not want to go to the children node, but then we had to do a search in the y direction because all the points in the sub tree was also stored in this node. All the points are sub tree was stored in this node therefore, we could stop at this node, we would refer to stop at this node and do the y searching. And report only those points that are contained within the the y interval. Here, we are not storing all the points in this node. (No audio from 46:47 to 46:51) So we may be force to go down because not all the points in this sub tree are stored here.

So, how do we know, which points basically should be reported that lie within this region should be reported. So, we cannot effort to stop there, we have to go down, but now we know that, when we go down we are only looking for a one dimensional range query. And now I use my heap the analyze with the heap and only going to go down as long as I basically report points and I am not going to sort of visit more than twice the number of points that are actually repeat. So now, this whole thing if you if you if you if you if you put the whole argument together. Now, we are basically visiting about order k well 2k nodes, but there is also this log n factor why why would the log n factor be theirs.

(( ))

No no. So, the some should be something in the in the y direction after all we are going to like like the range search tree. We we are going to express the y interval in terms of two log n

canonical intervals. So, those log n I have to count any way. So, those log n will get counted plus I am not going to do a binary search in these these in these node that I am going to actually go down. And report the points in because they are stored as heap also. So, I am going to visit two log n nodes like I do in the in the range search tree in the x direction, but when I report the points I am not doing a binary search report the points. I am only going down those sub trees, which completely fall within, which basically define the canonical intervals, but within each of them. I am only going to visit twice the number of points reported.

So, overall so, overall then my analysis is so, search times so, let us this kind of argument. You know where we decouple them is not is not the right thing to do we cannot look at them completely separately. The things are happening actually together. So, over all there are two log n canonical intervals corresponding to some nodes nodes and within each of this canonical interval. Let us think about the sub tree of these canonical intervals within each of this sub tree. I am not going to visit more than twice the number of points. So we will descend from those nodes some nodes let us called it some N of v. So, node N of v till we find suitable y coordinates. And this is no more than and is no more than twice the number of points reported. So, the overall search time becomes order log n plus k. (No audio from 50:03 to 50:15)

So, we are never doing that sort of binary search in the y direction that is why we are saving. And because we are not doing that binary search in the y direction, we are not encounter in that log n cost for each of this canonical intervals. The heaps are buying us things, because in heap, we can do the semi interval range query, range search, range reporting in order k time that is all. And the point is being stored only once so, that is this space is order n. So, we got the ideal bound that we looking for. So, any questions or it is it is a very clever data structure. And it is actually can be used in practice, because there is no complication as such, I mean this is intermediate of the conceptual thing, but in terms of implementation is very clean. So, priority search tree is actually quite frequently used many, many situations. So, I will stop here today.