MinDisk(P) :

(1) If $|P| \leq 3$, then return the disk passing through these points. Otherwise, randomly permute the points in P yielding the sequence $\langle p_1, p_2, \ldots, p_n \rangle$.
(2) Let D_2 be the minimum disk enclosing $\{p_1, p_2\}$.
(3) for $i = 3$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ MinDiskWith1Pt$(P[1..i-1], p_i)$.

MinDiskWith1Pt(P, q) :

(1) Randomly permute the points in P. Let D_1 be the minimum disk enclosing $\{q, p_1\}$.
(2) for $i = 2$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ MinDiskWith2Pts$(P[1..i-1], q, p_i)$.

MinDiskWith2Pts(P, q_1, q_2) :

(1) Randomly permute the points in P. Let D_0 be the minimum disk enclosing $\{q_1, q_2\}$.
(2) for $i = 1$ to $|P|$ do
  (a) if $p_i \in D_{i-1}$ then $D_i = D_{i-1}$.
  (a) else $D_i =$ Disk(q_1, q_2, p_i).

Lecture 11: Orthogonal Range Searching and kd-Trees

Reading: Chapter 5 in the 4M's.

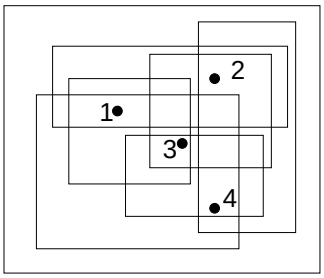
Range Queries: We will shift our focus from algorithm problems to data structures for the next few lectures. We will consider the following class of problems. Given a collection of objects, preprocess them (storing the results in a data structure of some variety) so that queries of a particular form can be answered efficiently. Generally we measure data structures in terms of two quantities, the time needed to answer a query and the amount of space needed by the data structure. Often there is a tradeoff between these two quantities, but most of the structures that we will be interested in will have either linear or near linear space. Preprocessing time is an issue of secondary importance, but most of the algorithms we will consider will have either linear of $O(n \log n)$ preprocessing time.

In a *range queries* we are given a set P of points and region R in space (e.g., a rectangle, polygon, halfspace, or disk) and are asked list (or count or compute some accumulation function of) the subset of P lying within the region. To get the best possible performance, the design of the data structure is tailored to the particular type of region, but there are some data structures that can be used for a wide variety of regions.

An important concept behind all geometric range searching is that the subsets that can be formed by simple geometric ranges is much smaller than the set of possible subsets (called the *power set*) of P. We can define any range search problem abstractly as follows. Given a particular class of ranges, a *range space* is a pair (P, R) consisting of the points P and the collection R of all subsets of P that be formed by ranges of this class. For example, the following figure shows the range space assuming rectangular ranges for a set of points in the plane. In particular, note that the sets $\{1, 4\}$ and $\{1, 2, 4\}$ cannot be formed by rectangular ranges.

Today we consider *orthogonal rectangular range queries*, that is, ranges defined by rectangles whose sides are aligned with the coordinate axes. One of the nice things about rectangular ranges is that they can be decomposed into a collection of 1-dimensional searches.

Canonical Subsets: A common approach used in solving almost all range queries is to represent P as a collection of *canonical subsets* $\{S_1, S_2, \ldots, S_k\}$, each $S_i \subseteq S$ (where k is generally a function of n and the type of ranges),



R = { {},
{1}, {2}, {3}, {4},
{1,2}, {1,3}, {2,3},{2,4},{3,4},
{1,2,3}, {1,3,4}, {2,3,4} ,
{1,2,3,4} }

Figure 40: Rectangular range space.

such that any set can be formed as the disjoint union of canonical subsets. Note that these subsets may generally overlap each other.

There are many ways to select canonical subsets, and the choice affects the space and time complexities. For example, the canonical subsets might be chosen to consist of n singleton sets, each of the form $\{p_i\}$. This would be very space efficient, since we need only $O(n)$ total space to store all the canonical subsets, but in order to answer a query involving k objects we would need k sets. (This might not be bad for reporting queries, but it would be too long for counting queries.) At the other extreme, we might let the canonical subsets be the power set of P. Now, any query could be answered with a single canonical subset, but we would have 2^n different canonical subsets to store. (A more realistic solution would be to use the set of all ranges, but this would still be quite large for most interesting range spaces.) The goal of a good range data structure is to strike a balance between the total number of canonical subsets (space) and the number of canonical subsets needed to answer a query (time).

One-dimensional range queries: Before consider how to solve general range queries, let us consider how to answer 1-dimension range queries, or *interval queries*. Let us assume that we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ on the line, which we will preprocess into a data structure. Then, given an interval $[x_{lo}, x_{hi}]$, the goal is to report all the points lying within the interval. Ideally we would like to answer a query in time $O(\log n + k)$ time, where k is the number of points reported (an output sensitive result). Range counting queries can be answered in $O(\log n)$ time with minor modifications.

Clearly one way to do this is to simply sort the points, and apply binary search to find the first point of P that is greater than or equal to x_{lo}, and less than or equal to x_{hi}, and then list all the points between. This will not generalize to higher dimensions, however.

Instead, sort the points of P in increasing order and store them in the leaves of a balanced binary search tree. Each internal node of the tree is labeled with the largest key appearing in its left child. We can associate each node of this tree (implicitly or explicitly) with the subset of points stored in the leaves that are descendents of this node. This gives rise to the $O(n)$ *canonical subsets*. For now, these canonical subsets will not be stored explicitly as part of the data structure, but this will change later when we talk about range trees. This is illustrated in the figure below.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the leftmost leaf u whose key is greater than or equal to x_{lo} and the rightmost leaf v whose key is less than or equal to x_{hi}. Clearly all the leaves between u and v, together possibly with u and v, constitute the points that lie within the range. If $key(u) = x_{lo}$ then we include u's canonical (single point) subset and if $key(v) = x_{hi}$ then we do the same for v. To form the remaining canonical subsets, we take the subsets of all the *maximal subtrees* lying between u and v.

Here is how to compute these subtrees. The search paths to u and v may generally share some common subpath, starting at the root of the tree. Once the paths diverge, as we follow the left path to u, whenever the path goes to the left child of some node, we add the canonical subset associated with its right child. Similarly, as we follow

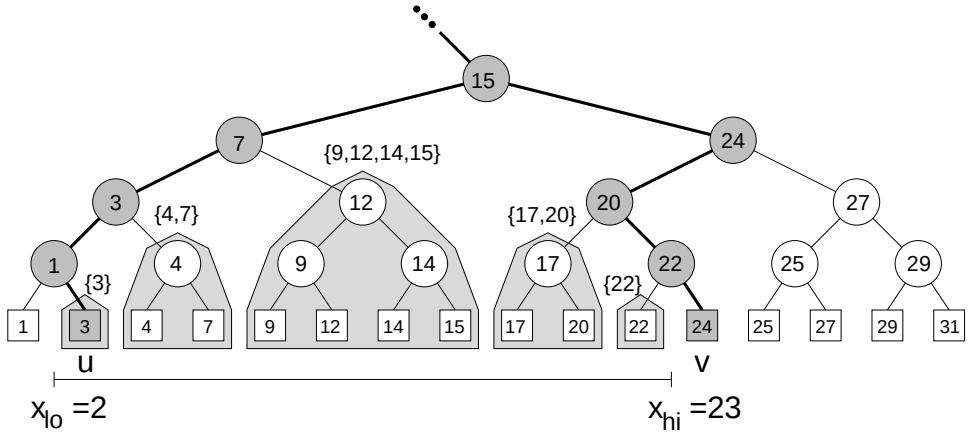


Figure 41: Canonical sets for interval queries.

the right path to v, whenever the path goes to the right child, we add the canonical subset associated with its left child.

To answer a range reporting query we simply traverse these canonical subtrees, reporting the points of their leaves. Each tree can be traversed in time proportional to the number of leaves in each subtree. To answer a range counting query we store the total number of points in each subtree (as part of the preprocessing) and then sum all of these over all the canonical subtrees.

Since the search paths are of length $O(\log n)$, it follows that $O(\log n)$ canonical subsets suffice to represent the answer to any query. Thus range counting queries can be answered in $O(\log n)$ time. For reporting queries, since the leaves of each subtree can be listed in time that is proportional to the number of leaves in the tree (a basic fact about binary trees), it follows that the total time in the search is $O(\log n + k)$, where k is the number of points reported.

In summary, 1-dimensional range queries can be answered in $O(\log n)$ time, using $O(n)$ storage. This concept of finding maximal subtrees that are contained within the range is fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Let see next how can we extend this to higher dimensional range queries.

Kd-trees: The natural question is how to extend 1-dimensional range searching to higher dimensions. First we will consider kd-trees. This data structure is easy to implement and quite practical and useful for many different types of searching problems (nearest neighbor searching for example). However it is not the asymptotically most efficient solution for the orthogonal range searching, as we will see later.

Our terminology is a bit nonstandard. The data structure was designed by Jon Bentley. In his notation, these were called "k-d trees," short for "k-dimensional trees". The value k was the dimension, and thus there are 2-d trees, 3-d trees, and so on. However, over time, the specific value of k was lost. Our text uses the somewhat nonstandard form "kd-tree" rather than "k-d tree." By the way, there are many variants of the kd-tree concept. We will describe the most commonly used one, which is quite similar to Bentley's original design. In our trees, points will be stored only at the leaves. There are variants in which points are stored at internal nodes as well.

The idea behind a kd-tree is to extend the notion of a one dimensional tree. But for each node we subdivide space either by splitting along x-coordinate of the points or along the y-coordinates. Each internal node t of the kd-tree is associated with the following quantities:

$t.cutDim$	the cutting dimension
$t.cutVal$	the cutting value
$t.size$	the number of points in t's subtree

In dimension d, the cutting dimension may be represented as in integer ranging from 0 to $d-1$. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to $t.cutVal$ are stored in the left subtree and the remaining points are stored in the right subtree. (See the figure below.) If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side. This is done to allow us to balance the number of points in the left and right subtrees if there are many equal coordinate values. When a single point remains (or more generally a small constant number of points), we store it in a leaf node, whose only field $t.point$ is this point (or generally a pointer to this point).

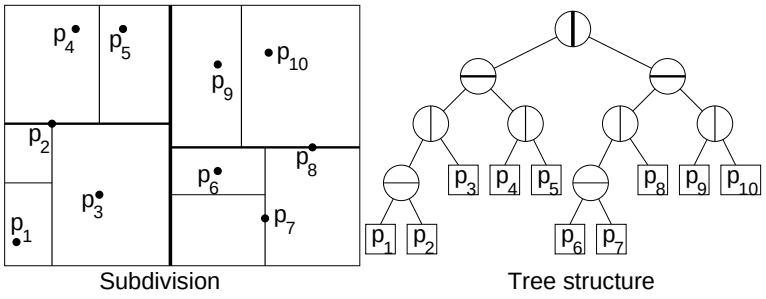


Figure 42: A kd-tree and the associated spatial subdivision.

The cutting process has a geometric interpretation. Each node of the tree is associated implicitly with a rectangular region of space, called a *cell*. (In general these rectangles may be unbounded, but in many applications it is common to restrict ourselves to some bounded rectangular region of space before splitting begins, and so all these rectangles are bounded.) The cells are nested in the sense that a child's cell is contained within its parent's cell. Hence, these cells define a *hierarchical decomposition* of space. This is illustrated in figure.

There are two key decisions in the design of the tree.

How is the cutting dimension chosen? The simplest method is to cycle through the dimensions one by one. (This method is shown in the above figure.) Since the cutting dimension depends only on the level of a node in the tree, one advantage of this rule is that the cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

One disadvantage of this splitting rule is that, depending on the data distribution, this simple cyclic rule may produce very skinny (elongated) cells, and such cells may adversely affect query times. Another method is to select the cutting dimension to be the one along which the points have the greatest *spread*, defined to be the difference between the largest and smallest coordinates. Bentley call the resulting tree an *optimized kd-tree*.

How is the cutting value chosen? To guarantee that the tree has height $O(\log n)$, the best method is to let the cutting value be the median coordinate along the cutting dimension. If there are an even number of points in the subtree, we may take either the upper or lower median, or we may simply take the midpoint between these two points. In our example, when there are an odd number of points, the median is associated with the left (or lower) subtree.

A kd-tree is a special case of a more general class of hierarchical spatial subdivisions, called *binary space partition trees* (or *BSP trees*) in which the splitting lines (or hyperplanes in general) may be oriented in any direction. In the case of BSP trees, the cells are convex polygons.

Constructing the kd-tree: It is possible to build a kd-tree in $O(n \log n)$ time by a simple top-down recursive procedure. The most costly step of the process is determining the median coordinate for splitting purposes. One way to do this is to maintain two lists of pointers to the points, one sorted by x-coordinate and the other containing pointers to the points sorted according to their y-coordinates. (In dimension d, d such arrays would be maintained.) Using these two lists, it is an easy matter to find the median at each step in constant time. In linear time

it is possible to split each list about this median element. For example, if $x = s$ is the cutting value, then all points with $p_x \leq s$ go into one list and those with $p_x > s$ go into the other. (In dimension d this generally takes $O(d)$ time per point.) This leads to a recurrence of the form $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$. Since there are n leaves and each internal node has two children, it follows that the number of internal nodes is $n - 1$. Hence the total space requirements are $O(n)$.

Theorem: A balanced kd-tree of n points can be constructed in $O(n \log n)$ time and has $O(n)$ space.

Range Searching in kd-trees: Let us consider how to answer orthogonal range counting queries. Range reporting queries are an easy extension. Let Q denote the desired range, t denote the current node in the kd-tree, and let C denote the rectangular cell associated with t. The search algorithm traverses the tree recursively. If it arrives at a leaf cell, we check to see whether the associated point, t.*point*, lies within Q, and if so we count it. Otherwise, t is an internal node. If t's cell C is disjoint from Q, then we know that no point in the subtree rooted at t is in the query range, and so there is nothing to count. If C is entirely contained within Q, then every point in the subtree rooted at t can be counted. (These points represent a canonical subset.) Otherwise, t's cell partially overlaps Q. In this case we recurse on t's two children and update the count accordingly.

kd-tree Range Counting Query

int rangeCount(Range Q, KDNode t, Rectangle C)

(1) if (t is a leaf)

  (a) if (Q contains t.*point*) return 1,

  (b) else return 0.

(2) if (t is not a leaf)

  (a) if ($C \cap Q = \emptyset$) return 0.

  (b) else if ($C \subseteq Q$) return t.*size*.

  (c) else, split C along t's cutting dimension and cutting value, letting C_1 and C_2 be the two rectangles. Return (rangeCount(Q, t.*left*, C_1) + rangeCount(Q, t.*right*, C_2)).

The figure below shows an example of a range search. White nodes have been visited in the search. Light shaded nodes were not visited because their cell is contained entirely within Q. Dark shaded nodes are not visited because their cell is disjoint from Q.

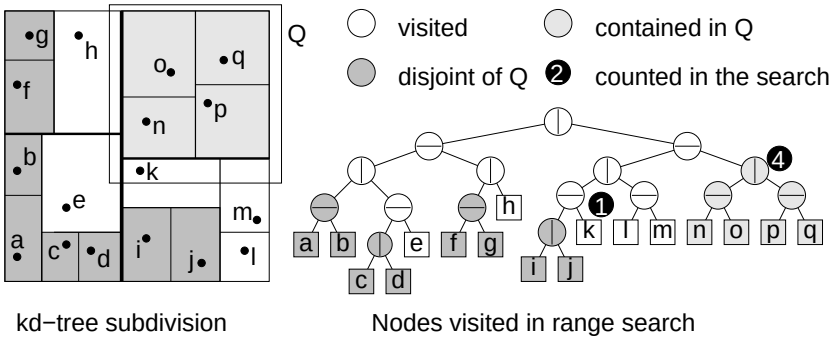


Figure 43: Range search in a kd-tree. (Note: This particular tree was not generated by the algorithm described above.)

Analysis of query time: How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree. Recall from the discussion above that a node is processed (both children visited) if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are processed in the search, we first bound the number of nodes that are stabbed.

Lemma: Given a balanced kd-tree with n points using the alternating splitting rule, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider a processed node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$.

Because we have an exponentially increasing number, the total sum is dominated by its last term. Thus is suffices to count the number of nodes stabbed at the lowest level of the tree. Since the tree is balanced, it has roughly $\lg n$ levels. Thus the number of leaf nodes processed at the bottommost level is $2^{(\lg n)/2} = 2^{\lg \sqrt{n}} = \sqrt{n}$. This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$. Since we only make recursive calls when a cell is stabbed, it follows that the total number of nodes visited by the search is $O(\sqrt{n})$.

Theorem: Given a balanced kd-tree with n points, orthogonal range counting queries can be answered in $O(\sqrt{n})$ time and reporting queries can be answered in $O(\sqrt{n} + k)$ time. The data structure uses space $O(n)$.

Lecture 12: Orthogonal Range Trees

Reading: Chapter 5 in the 4M's.

Orthogonal Range Trees: Last time we saw that kd-trees could be used to answer orthogonal range queries in the plane in $O(\sqrt{n} + k)$ time. Today we consider a better data structure, called *orthogonal range trees*.

An orthogonal range tree is a data structure which, in all dimensions $d \geq 2$, uses $O(n \log^{(d-1)} n)$ space, and can answer orthogonal rectangular range queries in $O(\log^{(d-1)} n + k)$ time, where k is the number of points reported. Preprocessing time is the same as the space bound. Thus, in the plane, we can answer range queries in time $O(\log n)$ and space $O(n \log n)$. We will present the data structure in two parts, the first is a version that can answer queries in $O(\log^2 n)$ time in the plane, and then we will show how to improve this in order to strip off a factor of $\log n$ from the query time.

Multi-level Search Trees: The data structure is based on the concept of a *multi-level search tree*. In this method, a complex search is decomposed into a constant number of simpler range searches. We cascade a number of search structures for simple ranges together to answer the complex range query. In this case we will reduce a d-dimensional range search to a series of 1-dimensional range searches.

Suppose you have a query which can be stated as the intersection of a small number of simpler queries. For example, a rectangular range query in the plane can be stated as two range queries: Find all the points whose x-coordinates are in the range $[Q.x_{lo}, Q.x_{hi}]$ and all the points whose y-coordinates are in the range $[Q.y_{lo}, Q.y_{hi}]$. Let us consider how to do this for 2-dimensional range queries, and then consider how to generalize the process. First, we assume that we have preprocessed the data by building a range tree for the first range query, which in this case is just a 1-dimensional range tree for the x-coordinates. Recall that this is just a balanced binary tree

on these points sorted by x-coordinates. Also recall that each node of this binary tree is implicitly associated with a *canonical subset* of points, that is, the points lying in the subtree rooted at this node.

Observe that the answer to any 1-dimensional range query can be represented as the disjoint union of a small collection of $m = O(\log n)$ canonical subsets, $\{S_1, S_2, \ldots, S_m\}$, where each subset corresponds to a node t in the range tree. This constitutes the first level of the search tree. To continue the preprocessing, for the second level, for each node t in this x-range tree, we build an *auxiliary tree*, t_{aux}, each of which is a y-coordinate range tree, which contains all the points in the canonical subset associated with t.

The final data structure consists of two levels: an x-range tree, such that each node in this tree points to auxiliary y-range tree. This notion of a tree of trees is basic to solving range queries by leveling. (For example, for d-dimensional range trees, we will have d-levels of trees.)

Query Processing: Given such a 2-level tree, let us consider how to answer a rectangular range query Q. First, we determine the nodes of the tree that satisfy the x portion of the range query. Each such node t is associated with a canonical set S_t, and the disjoint union of these sets $O(\log n)$ sets consistute all the points of the data set that lie within the x portion of the range. Thus to finish the query, we need to find out which points from each canonical subset lie within the range of y-coordinates. To do this, for each such node t, we access the auxiliary tree for this node, t_{aux} and perform a 1-dimensional range search on the y-range of the query. This process is illustrated in the following figure.

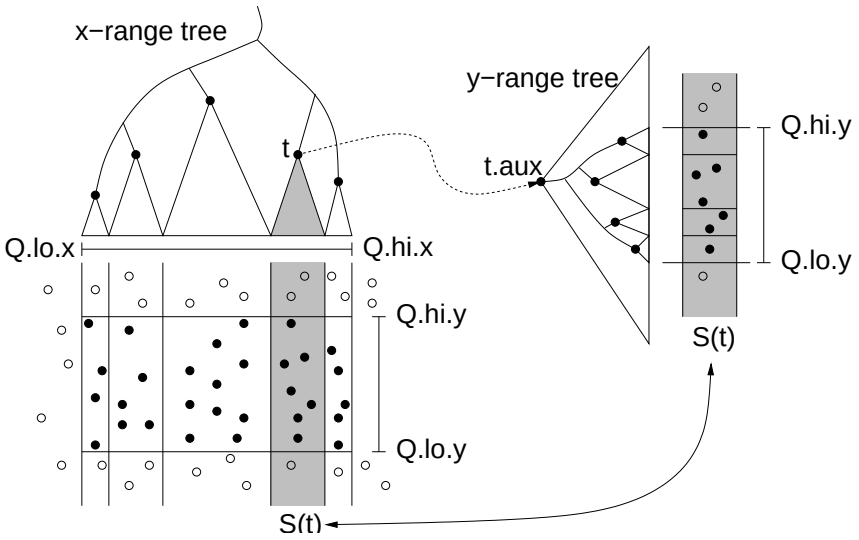


Figure 44: Orthogonal range tree search.

What is the query time for a range tree? Recall that it takes $O(\log n)$ time to locate the nodes representing the canonical subsets for the 1-dimensional range query. For each, we invoke a 1-dimensional range search. Thus there are $O(\log n)$ canonical sets, each invoking an $O(\log n)$ range search, for a total time of $O(\log^2 n)$. As before, listing the elements of these sets can be performed in additional k time by just traversing the trees. Counting queries can be answered by precomputing the subtree sizes, and then just adding them up.

Space: The space used by the data structure is $O(n \log n)$ in the plane (and $O(n \log^{(d-1)} n)$ in dimension d). The reason comes by summing the sizes of the two data structures. The tree for the x-coordinates requires only $O(n)$ storage. But we claim that the total storage in all the auxiliary trees is $O(n \log n)$. We want to count the total sizes of all these trees. The number of nodes in a tree is proportional to the number of leaves, and hence the number of points stored in this tree. Rather than count the number of points in each tree separately, instead let us count the number of trees in which each point appears. This will give the same total. Observe that a point appears in the auxiliary trees of each of its ancestors. Since the tree is balanced, each point has

$O(\log n)$ ancestors, and hence each point appears in $O(\log n)$ auxiliary trees. It follows that the total size of all the auxiliary trees is $O(n \log n)$. By the way, observe that because the auxiliary trees are just 1-dimensional trees, we could just store them as a sorted array.

We claim that it is possible to construct a 2-dimensional range tree in $O(n \log n)$ time. Constructing the 1-dimensional range tree for the x-coordinates is easy to do in $O(n \log n)$ time. However, we need to be careful in constructing the auxiliary trees, because if we were to sort each list of y-coordinates separately, the running time would be $O(n \log^2 n)$. Instead, the trick is to construct the auxiliary trees in a bottom-up manner. The leaves, which contain a single point are trivially sorted. Then we simply merge the two sorted lists for each child to form the sorted list for the parent. Since sorted lists can be merged in linear time, the set of all auxiliary trees can be constructed in time that is linear in their total since, or $O(n \log n)$. Once the lists have been sorted, then building a tree from the sorted list can be done in linear time.

Multilevel Search and Decomposable Queries: Summarizing, here is the basic idea to this (and many other query problems based on leveled searches). Let (S, R) denote the range space, consisting of points S and range sets R. Suppose that each range of R can be expressed as an intersection of simpler ranges $R = R_1 \cap R_2 \cap \ldots \cap R_c$, and assume that we know how to build data structures for each of these simpler ranges.

The multilevel search structure is built by first building a range search tree for query R_1. In this tree, the answer to any query can be represented as the disjoint union of some collection $\{S_1, S_2, \ldots, S_m\}$ of canonical subsets, each a subset of S. Each canonical set corresponds to a node of the range tree for R_1.

For each node t of this range tree, we build an auxiliary range tree for the associated canonical subset S_t for ranges of class R_2. This forms the second level. We can continue in this manner, with each node of the range tree for the ranges R_i being associated with an auxiliary range tree for ranges R_{i+1}.

To answer a range query, we solve the first range query, resulting in a collection of canonical subsets whose disjoint union is the answer to this query. We then invoke the second range query problem on each of these canonical subsets, and so on. Finally we take the union of all the answers to all these queries.

Fractional Cascading: Can we improve on the $O(\log^2 n)$ query time? We would like to reduce the query time to $O(\log n)$. As we descend the search the x-interval tree, for each node we visit, we need to search the corresponding y-interval tree. It is this combination that leads to the squaring of the logarithms. If we could search each y-interval in $O(1)$ time, then we could eliminate this second log factor. The trick to doing this is used in a number of places in computational geometry, and is generally a nice idea to remember. We are repeatedly searching different lists, but always with the same key. The idea is to merge all the different lists into a single massive list, do one search in this list in $O(\log n)$ time, and then use the information about the location of the key to answer all the remaining queries in $O(1)$ time each. This is a simplification of a more general search technique called *fractional cascading*.

In our case, the massive list on which we will do one search is the entire of points, sorted by y-coordinate. In particular, rather than store these points in a balanced binary tree, let us assume that they are just stored as sorted arrays. (The idea works for either trees or arrays, but the arrays are a little easier to visualize.) Call these the *auxiliary lists*. We will do one (expensive) search on the auxiliary list for the root, which takes $O(\log n)$ time. However, after this, we claim that we can keep track of the position of the y-range in each auxiliary list in constant time as we descend the tree of x-coordinates.

Here is how we do this. Let v be an arbitrary internal node in the range tree of x-coordinates, and let v_L and v_R be its left and right children. Let A_v be the sorted auxiliary list for v and let A_L and A_R be the sorted auxiliary lists for its respective children. Observe that A_v is the disjoint union of A_L and A_R (assuming no duplicate y-coordinates). For each element in A_v, we store two pointers, one to the item of equal or larger value in A_L and the other to the item of equal or larger value in A_R. (If there is no larger item, the pointer is null.) Observe that once we know the position of an item in A_v, then we can determine its position in either A_L or A_R in $O(1)$ additional time.

Here is a quick illustration of the general idea. Let v denote a node of the x-tree, and let v_L and v_R denote its left and right children. Suppose that (in bottom to top order) the associated nodes within this range are:

$\langle p_1, p_2, p_3, p_4, p_5, p_6 \rangle$, and suppose that in v_L we store the points $\langle p_2, p_3, p_5 \rangle$ and in v_R we store $\langle p_1, p_4, p_6 \rangle$. (This is shown in the figure below.) For each point in the auxiliary list for v, we store a pointer to the lists v_L and v_R, to the position this element would be inserted in the other list (assuming sorted by y-values). That is, we store a pointer to the largest element whose y-value is less than or equal to this point.

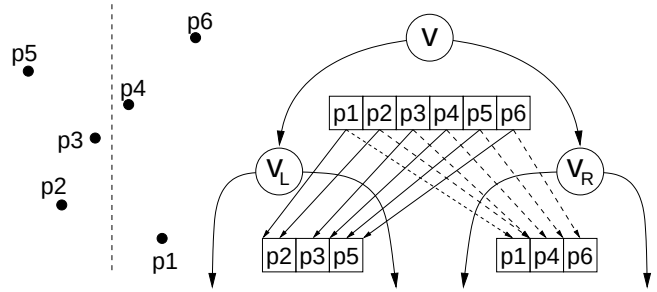


Figure 45: Cascaded search in range trees.

At the root of the tree, we need to perform a binary search against all the y-values to determine which points lie within this interval, for all subsequent levels, once we know where the y-interval falls with respect to the order points here, we can drop down to the next level in $O(1)$ time. Thus (as with fractional cascading) the running time is $O(2 \log n)$, rather than $O(log^2 n)$. It turns out that this trick can only be applied to the last level of the search structure, because all other levels need the full tree search to compute canonical sets.

Theorem: Given a set of n points in R^d, orthogonal rectangular range queries can be answered in $O(\log^{(d-1)} n + k)$ time, from a data structure of size $O(n \log^{(d-1)} n)$ which can be constructed in $O(n \log^{(d-1)} n)$ time.

Lecture 13: Hereditary Segment Trees and Red-Blue Intersection

Reading: Segment trees are presented in Chapt 10 of the 4M's. However, most of today's material has been taken from the paper, "Algorithms for Bichromatic Line-Segment Problems and Polyhedral Terrains," by B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, *Algorithmica*, 11, 1994 116–132.

Red-Blue Segment Intersection: We have been talking about the use of geometric data structures for solving query problems. Often data structures are used as intermediate structures for solving traditional input/output problems, which do not involve preprocessing and queries. (Another famous example of this is HeapSort, which introduces the heap data structure for sorting a list of numbers.) Today we will discuss a variant of a useful data structure, the *segment tree*. The particular variant is called a *hereditary segment tree*. It will be used to solve the following problem.

Red-Blue Segment Intersection: Given a set B of m pairwise disjoint "blue" segments in the plane and a set R of n pairwise disjoint "red" segments, count (or report) all *bichromatic pairs* of intersecting line segments (that is, intersections between red and blue segments).

It will make things simpler to think of the segments as being open (not including their endpoints). In this way, the pairwise disjoint segments might be the edges of a planar straight line graph (PSLG). Indeed, one of the most important application of red-blue segment intersection involves computing the overlay of two PSLG's (one red and the other blue) This is also called the *map overlay problem*, and is often used in geographic information systems. The most time consuming part of the map overlay problem is determining which pairs of segments overlap. See the figure below.

Let $N = n + m$ denote the total input size and let k denote the total number of bichromatic intersecting pairs. We will present an algorithm for this problem that runs in $O(k + N \log^2 N)$ time for the reporting problem and