Augmentation

The idea of augmentation is to add information to parts of a standard data structure to allow it to perform more complex tasks. You've practiced this concept already on Problem set 2, in order to allow skip lists to support $O(\log n)$ time RANK and SELECT methods. We'll look at a similar problem of creating "order statistic trees" by augmenting 2-3-4 trees to support RANK and SELECT. (The same methodology works for augmenting BSTs.)

Order Statistic Trees

Problem: Augment an ordinary 2-3-4 tree to support efficient RANK and SELECT-RANK methods without hurting the asymptotic performance of the normal tree operations.

Idea: In each node, store the rank of the node in its subtree. An example order statistic tree is shown in Figure 1.

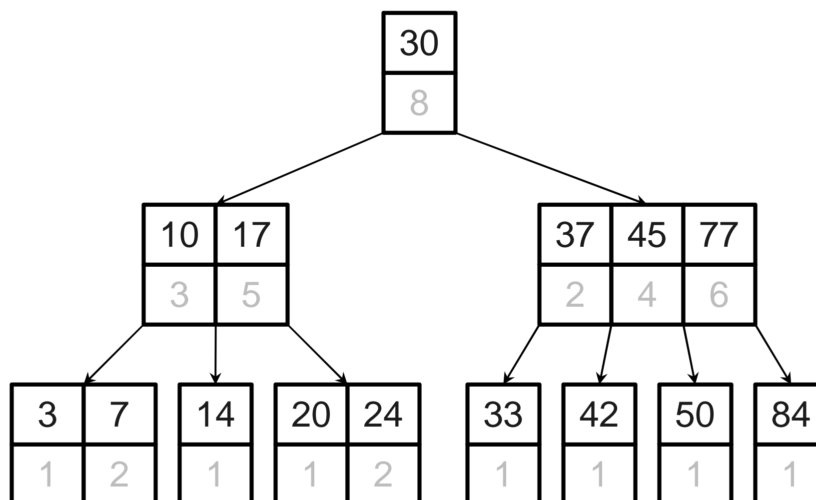


Figure 1: An example order statistic tree. The original values stored in the tree are shown in the top box in a darker gray, while their associated rank values are shown in the bottom box in a lighter gray.

RANK, SELECT:

We will examine SELECT, noting that the procedure for RANK is similar. Suppose we are searching for the ith order statistic. Starting from the root of the tree, look at the rank associated with the current node, n. If $n.rank = i$, return n. If $n.rank > i$, recursively SELECT the ith order statistic from the left child of n. Otherwise $n.rank < i$, so recursively select the $(i - n.rank)$th order statistic from the right child of n.

INSERT:

The INSERT method for order statistic trees augments the normal insert method for 2-3-4 trees. When an element x is inserted into a subtree rooted at some node, all elements to the right of the subtree into which x is inserted must increment their rank values by 1. When x is inserted into a leaf, all elements to the right of x's position must increment their rank values by 1. Note that this requires a constant number of increments per level descended during the insert process, so the runtime of this operation is still $O(\lg n)$

After x is inserted, promotions may occur to maintain the structure of the 2-3-4 tree. Whenever an element n with rank r is promoted, first all elements to the right of n in its former level should have their ranks decremented by r, then the rank of n should be incremented by the former largest rank in its new level. This requires a constant number of rank manipulations per promotion, of which $O(\lg n)$ may occur, so INSERT still takes $O(\lg n)$ time.

Theorem Suppose we are augmenting a balanced search tree T of n nodes with a value f on each node. If the value of f for each node x depends on only the information in nodes x, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$, then we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations. (CLRS, 346)

Range Trees

Problem: We have n points in d dimensions. For example, we may have a database of n records with d numeric fields, or we may have a collection of n restaurants with d associated values for each restaurant (longitude, latitude, food, decor, service, cost, etc.) We want to make queries efficiently about these points within an axis-aligned box, e.g. "how many restaurants are within price range [p_1, p_2] and between intersections i_1 and i_2 on Main St." Our goal is to preprocess these points into a static data structure to support fast queries.

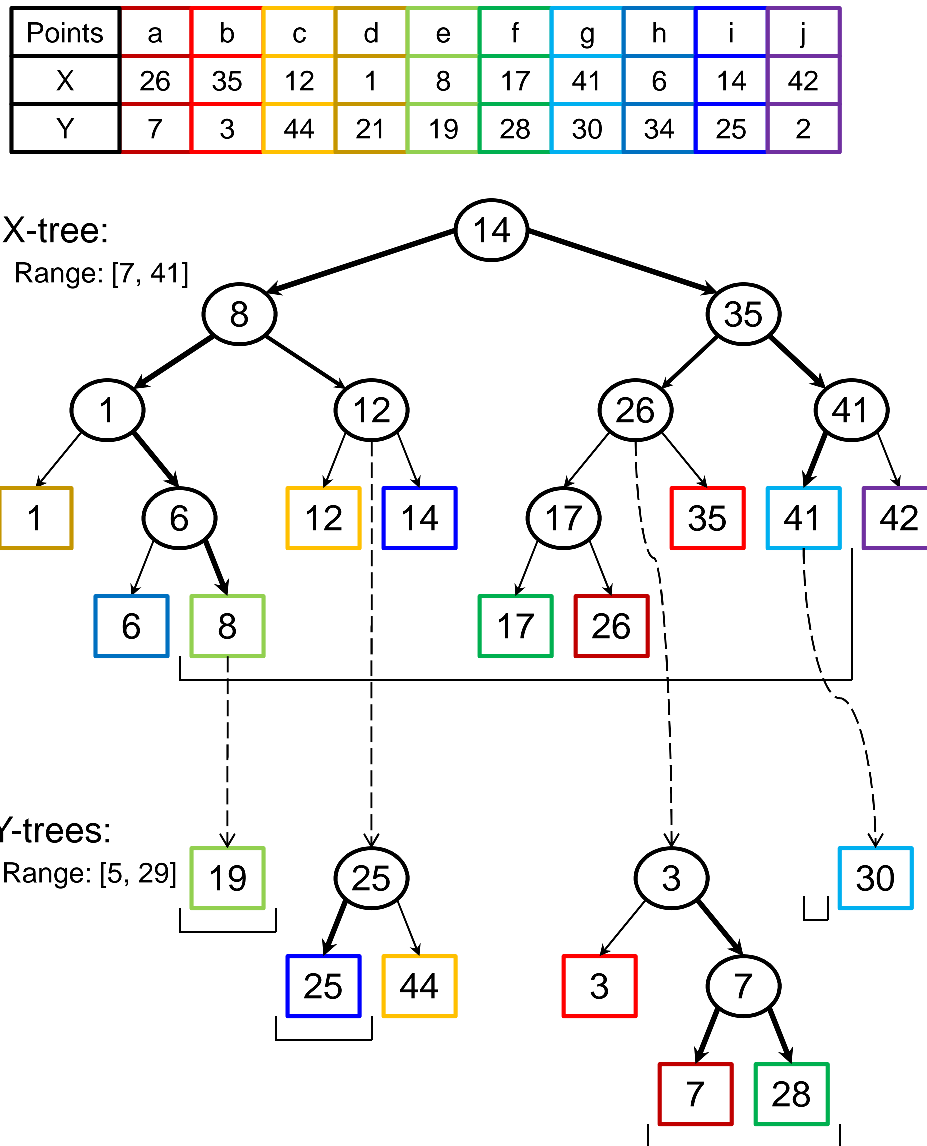
We will develop an example range tree using the points and associated data shown in Figure 2. Each of these points has a label - a through f - and two associated values - X and Y. To track these points in the range tree, each point is associated with a color. Each point in the range tree is colored with the appropriate color and labeled with the dimension on which that range tree is keyed.

1D Range Trees

First consider the problem of finding the points within a range along 1 dimension. For example consider finding all points in Figure 2 with X values in the range $[7, 41]$.

Idea: Use a balanced search tree, such as a red-black tree. The leaves store the points, while internal nodes store copies of the points such that each internal node stores the maximum value in its left subtree. An example 1D range tree is the X-tree shown in Figure 2.

RANGE-QUERY($[x_1, x_2]$): Search for x, then search for y. Remove the prefixes in common with both search results, then return the subtrees "in between."



Points	a	b	c	d	e	f	g	h	i	j
X	26	35	12	1	8	17	41	6	14	42
Y	7	3	44	21	19	28	30	34	25	2

Figure 2: An example 2D range tree. The points being searched are shown in the chart at the top, with the columns colored by each points associated color. (This color is not stored in the actual tree, but is a visual guide for the reader to associate values in the trees below with the original points.) The X-tree for these points is shown below this chart, along with an indication of the X-range searched for and found in the tree. The Y-trees associated with each subtree found in the X-tree search is shown at the bottom, with dashed arrows indicating the association of each Y-tree to a root of a subtree in the X-tree, and the results of the Y-range search on each of these Y-trees indicated.

In Figure 2 the search paths for finding 7 and 41 are shown by the very thick black arrows. The subtrees in between can be found as right branches not followed during the search for 7 and left branches not followed during the search for 41; these branches are indicated by the moderately thick arrows in the graph. For our purposes we only care about finding the roots of each subtree in between our range values.

Query Time: It takes $O(\lg n)$ to perform the search in the tree for each end of the given range; this follows from the runtime of the SEARCH operation on balanced search trees. While performing this search the roots of the subtrees known to be "in between" can be stored, so all of these subtrees can be found in $O(\lg n)$ time. Finally removing the prefixes in common to both searches can be done by retracing the nodes examined along both search paths and eliminating all but the last of the common nodes. Each search path is $O(\lg n)$ in length, so this step takes $O(\lg n)$ time as well. Therefore the query to report all of the subtrees takes $O(\lg n)$ time.

2D Range Trees

Now consider the problem of finding points within two 1-dimensional ranges. For examply consider finding all points in Figure 2 with X values in the range $[7, 41]$ and Y values in the range [5, 29].

Idea: Use a 1D range tree to find subtrees representing points matching $x = [x_1, x_2]$. With each node n in the x-tree, store a secondary 1D range tree containing all points in the x-tree subtree rooted at n keyed on $y = [y_1, y_2]$. Note that all of these y-trees are generated during preprocessing, and thus are available during any query.

A portion of an example 2D range tree is shown in Figure 2. The root of each subtree found in the appropriate X range is tied to another 1D range tree by a dashed arrow. There is actually 1 Y-tree associated with each node in the X-tree, but the other Y-trees are not shown. Each Y-tree contains the same set of points as are present in the X-tree subtree to which they are associated. These points are keyed in the Y-tree by their Y-value.
To query, first find all subtrees in primary x-tree, then recursively query each y-tree associated with an x-tree subtree found. In Figure 2 the Y-tree associated with each subtree root found in the X range query is queried by the Y range, yeilding the indicated results.

Query Time: The initial query on the x-tree takes $O(\lg n)$ time, and returns $O(\lg n)$ subtree roots. Each of these subtree roots is associated with a y-tree, each of which is queried in $O(\lg n)$ time. Therefore the total query time for a 2D range tree is $O(\lg n + (\lg n)(\lg n)) = O((\lg n)^2)$.

Space: Each element e belongs to $\lg n$ subtrees in the x-tree, and therefore will appear in $\lg n$ y-trees. Since there are n elements, our total space consumption is $O(n \lg n)$.

*d*D Range Trees

To add a third dimension, z, augment each subtree in each y-tree with another range tree keyed on z. Repeat this process of augmenting all subtrees in each tree for the previous dimension with a tree keyed on the next dimension until all d dimensions are covered.

Query Time: By generalizing the reasoning for the query time for 2D range trees, we find that the query time for a dD range tree is $O((\lg n)^d)$.

Space: By generalizing the reasoning for the space consumption for a 2D range tree, the space consumption for a dD range tree is $O(n(\lg n)^{d-1})$.