Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Advanced Data Structures

Range and Kd Trees

Dr. Amin Allam

[For more details, refer to "Introduction to Algorithms" by *Thomas Cormen*, et al.]
[For more details, refer to "Advanced Data Structures" by *Peter Brass*]

1 Range tree

A *range query* asks for the set of stored *points* (*values*) belonging to the *query interval* (*range*) [*lo*, *hi*]. A simple *red-black* tree can answer *range queries*, by initially searching for the minimum stored value $\geq$ *lo* and reporting it, then successively executing the following GetSuccessor(node) procedure from the last visited *node* several times until a *node* containing a value $\geq$ *hi* is returned. After each call (except possibly the last one), the returned *node* value is reported.

```
GetSuccessor(node)

if (node ▷ right ≠ null)
then
    node ← node ▷ right
    while (node ▷ left ≠ null) node ← node ▷ left
    return node
else
    while (node ≠ root)
        if (node = node ▷ parent ▷ left) then return node ▷ parent
        node ← node ▷ parent
    return null
```

Note that the *parent* field need not to be explicitly stored in each *node*. Alternatively, Whenever we go from a *node* to its *left child*, the node is pushed into a stack of parents to be used whenever needed. Whenever we go from a *right child* to its *parent*, that *parent* is popped from the stack. The size of such stack is $O(\log n)$ where n is the number of values stored in the balanced tree.

The visited nodes belong to one of the following set of nodes:

• The $O(\log n)$ nodes on the path from *root* to the *node* having value $\geq$ *lo*.
• The k nodes containing values belonging to the *query range*.
• The $O(\log n)$ nodes on the path from *root* to the *node* having value $\geq$ *hi*.

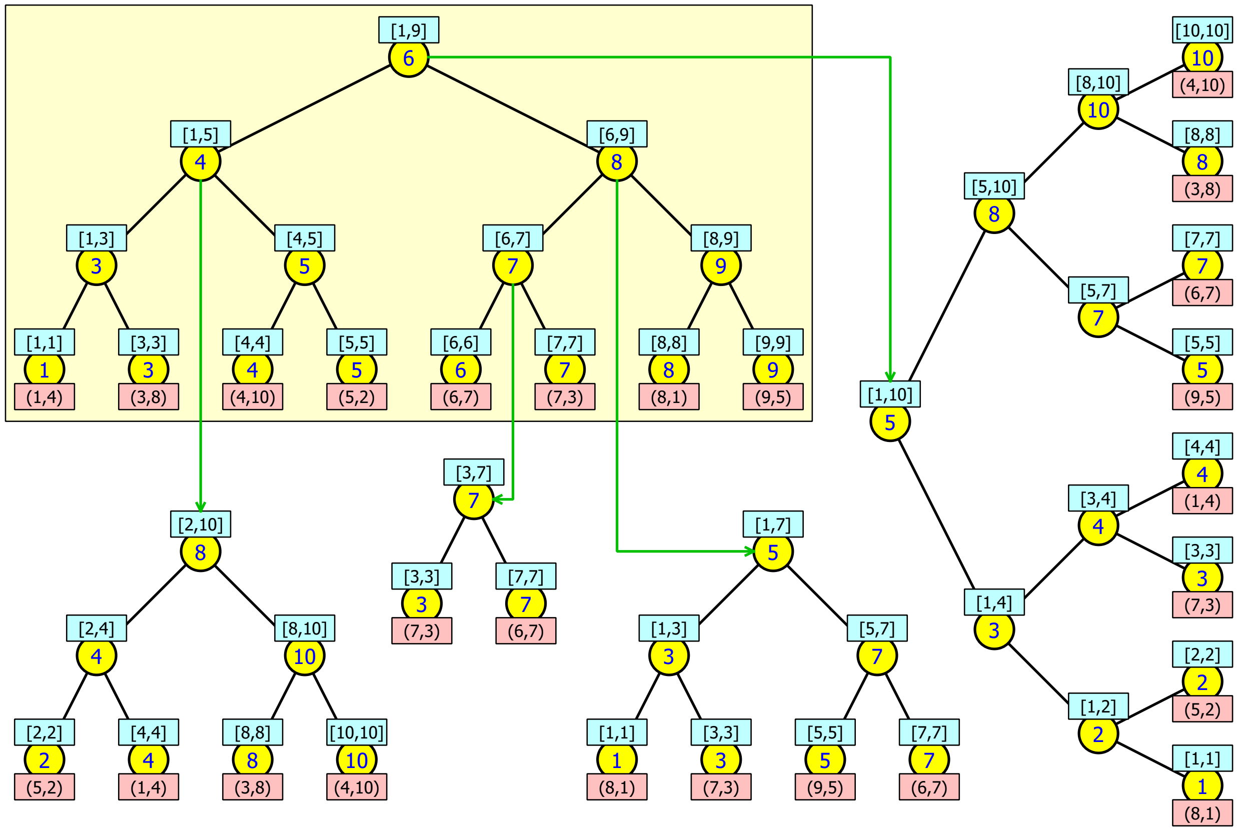
Each of the above *nodes* is visited at most 3 times: when it is reached from its *parent*, when it is reached from its *left child*, and when it is reached from its *right child*. Therefore, the complexity of executing GetSuccessor() k successive times to answer a *range query* is $O(\log n + k)$.

Unfortunately, the above method cannot be easily generalized to *higher dimensions*, such as retrieving all two dimensional *points* inside a *query rectangle*.

The *orthogonal 2d range tree* is a *static* data structure which answers *2d range queries*. It retrieves all two dimensional *points* (x, y) inside a *query rectangle* $\{[qx \triangleright lo, qx \triangleright hi], [qy \triangleright lo, qy \triangleright hi]\}$. The *orthogonal 2d range tree* is built in the following way:

• All *input points* are sorted based on its first (x) coordinate. A *static binary search tree* keyed by x is built such *input points* are stored in *leaves*, as shown inside the light-orange rectangle below. The remaining levels are constructed such that the tree is balanced and the key of each *internal node* is the smallest x in its *right subtree*. Each *internal node* stores an *interval* spanning the smallest and largest x existing in its *subtree*. Contrary to ordinary binary search trees, *internal nodes* only guides the search queries and do not store actual *input points* data.

• For each *node* $node_x$ in the *basic tree* keyed by x constructed above: construct an *associated tree* exactly as described above, except that it stores only *input points* existing in the *subtree* of $node_x$ and keyed by the second (y) coordinate. The figure shows only some of these trees.



The following $O(\log^2 n + k)$ procedure reports all *k input points* belonging to a *query rectangle*:

• If the *range tree interval* is disjoint from the x *query interval*, stop following the path down.
• If the *range tree interval* partially overlaps the x *query interval*, follow both paths down.
• If the *range tree interval* is entirely contained in the x *query interval*, stop following the path down, and do the following starting from the root of the *associated tree* of the current node:

  • If the *range tree interval* is disjoint from the y *query interval*, stop following the path down.
  • If the *range tree interval* partially overlaps the y *query interval*, follow both paths down.
  • If the *range tree interval* is entirely contained in the y *query interval*, stop following the path down, and report all *input points* stored in the *leaves* of this subtree:

The above complexity follows because each *interval* of one *query* coordinate is actually decomposed similarly to the *canonical representation* decomposition of size $O(\log n)$ described in the *segment tree* lecture.

To retrieve all two dimensional *points* (x, y) inside the *query rectangle* {[x=1, x=8], [y=2, y=5]}: First, we search for the *interval* [x=1, x=8] in the *basic tree* to reach the *intervals* {[1,5],[6,7],[8,8]}. For each *node* associated with these *intervals*, we search for [y=2, y=5] in its *associated tree*.

Searching for [y=2, y=5] in the *associated tree* of [1, 5] shown in the left bottom corner in the above figure, starting from the *root* [2, 10] we reach the *node* [2, 4] which is entirely contained in [2, 5] so we report all *input points* in the *leaves* of the subtree of [2, 4] which are [5, 2] and [1, 4].

Searching for [y=2, y=5] in the *associated tree* of [6, 7], starting from the *root* [3, 7] we reach the *node* [3, 3] which is entirely contained in [2, 5] so we report the one *input point* this subtree of [3, 3] which is [7, 3]. Searching for [y=2, y=5] in the *associated tree* of [8, 8] (which is not shown in the figure) does not lead to any results.

An *orthogonal 2d range tree* can be built by sorting all *input points* based on its x coordinate (if two *input points* have equal x coordinate, they are compared based on their y coordinate), and then recursively calling the following procedure *root* $\leftarrow$ Build2dRangeTree(*p[0 . . . n]*):

```
Function Build2dRangeTree(p[ist . . . iend]): ist = start index, iend = 1+ last index
• imed ← ⌊(ist+iend)/2⌋ (index of median point of p[ist . . . iend-1]
• Construct root node    • root▷key ← p[imed]▷x    • root▷interval ← [p[ist]▷x, p[iend-1]▷x]
• root▷left ← Build2dRangeTree(p[ist . . . imed-1])
• root▷right ← Build2dRangeTree(p[imed . . . iend])
• root▷assoc_tree ← Construct static binary search tree for all p[ist . . . iend-1] points keyed by y
• return root
```

Let $T(n)$ be the time complexity of the above algorithm. The non-recursive part consists mainly of constructing *root▷assoc_tree*, which can be done by sorting *points* by y in $O(n \log n)$ then constructing higher levels in $O(n)$. The sorting part can be replaced by just $O(n)$ merging of two sorted arrays if the recursive function returns also the *points* sorted by y. Therefore, the non-recursive part is only $O(n)$ time and space. Thus, the time and space complexity of the whole algorithm is $T(n) = 2T(n/2) + O(n)$. Solving the recurrence leads to $T(n) = O(n \log n)$.

When sorting *input points* based on its y coordinate, if two *input points* have equal y coordinate values, they are compared based on their x coordinate values. The reason is that *binary search trees* do not behave properly if equal keys occur. We need to differentiate between keys using any method, such that the same chosen method is used every time two keys need to be differentiated. If several *points* coincide, only one of them should be stored in the tree.

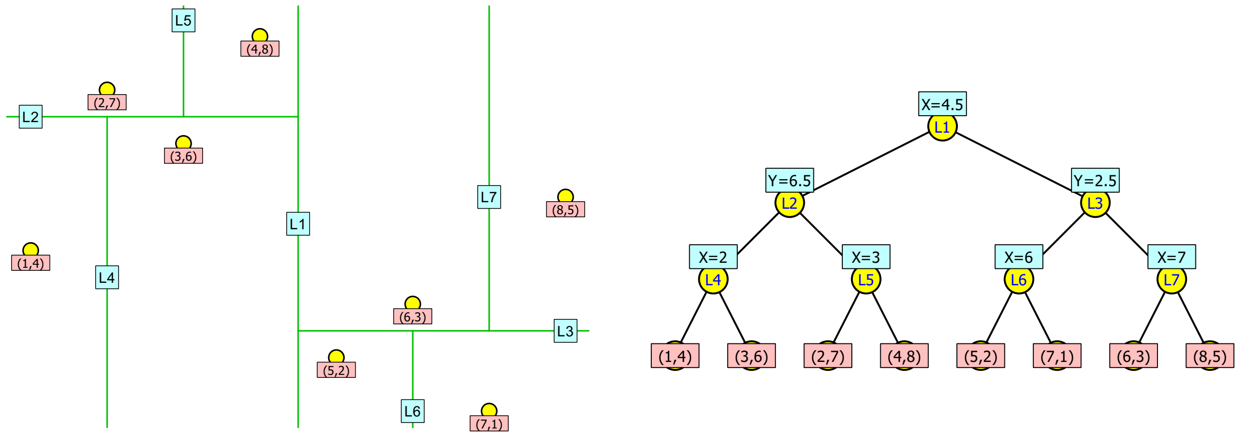
Similarly, the above technique can be generalized to any number of dimensions d. An *orthogonal range tree with d dimensions* mainly consists also of a *basic tree* based on the first coordinate (x) values of all points, but the *associated tree* of each *node* $node_x$ should be an *orthogonal range tree with $d - 1$ dimensions* based on all coordinate values (except the first one (x)) of all points belonging to the subtree of $node_x$. The construction time and space complexity of an *orthogonal range tree with d dimensions* having n d-dimensional points is $T(n) = O(n \log^{d-1} n)$. The *query* time complexity is $T(n) = O(\log^d n + k)$ where k is the number of *points* satisfying the *query*.

2 Kd tree

Kd tree is a *static* data structure that supports d-dimensional *orthogonal range queries* in a set of n d-dimensional points, exactly as *orthogonal range tree* described before, but with different time and space requirements. *Kd tree* requires only $O(n)$ space and $O(n \log n)$ construction time, regardless of the number of dimensions. However, the *query* time complexity $O(n^{1-\frac{1}{d}} + k)$ if the output consists of k points. In particular, the *query* time complexity for 2 dimensions is $O(\sqrt{n}+k)$ which is worse than the $O(\log^2 n + k)$ time complexity of the *orthogonal 2d range tree*.

Consider eight *2d input points* (x,y) shown in the left figure below to be stored in a *Kd tree* (*2d tree*). First, we divide the *input points* into two equal (or almost-equal) subsets based on their first (x) coordinate values. Then, we find a vertical line (whose equation is *X=constant*) that divides the two subsets. The line L1 (X=4.5) is chosen for that purpose, as shown in the left figure. The equation of the separating line L1 is then assigned to the *root* of the *Kd tree* (in the right figure). The *left subtree* should contain all points lying to the left of L1 (have less x coordinate values than the constant in L1 equation). The *right subtree* should contain all points lying to the right of L1. *Internal nodes* only act as separators, and *input points* are stored only in *leaves*.

Then, for each subset of the two subsets created above, we attempt to divide its nodes based on their second (y) coordinate values. The horizontal line L2 (Y=6.5) divides the left subset of four points, and the horizontal line L3 (Y=2.5) divides the right subset of four points. The *Kd tree* nodes of the second level are created and assigned such line equations as shown in the right figure. Note that if *nodes* of a any level contain vertical separators, *nodes* in the following level should contain horizontal separators, and vice versa. Similarly, *nodes* of the third level contains vertical lines where each line separates the two *input points* existing in the leaves of its subtree.



To retrieve all two dimensional *points* (x, y) inside the *query rectangle* {[x=0, x=4], [y=5, y=7.5]}, we start from the root having the separator line L1 (X=4.5). Obviously, the *query rectangle* lies entirely to the left of that separator, because $query \triangleright x \triangleright hi < L1 \triangleright x$ (4<4.5). Thus, we exclude the right subtree from our search and follow the left path only.

The L2 (Y=6.5) separator is not helpful since it lies inside $query \triangleright y$ interval, so the search follows both left and right paths down. The L4 (X=2) and L5 (X=3) separators are not helpful as well, so we follow both directions from both nodes to obtain four *points*. Comparing them against the original *query rectangle*, only (3,6) and (2,7) are reported.

A *2d tree* can be built by recursively calling the following procedure *root* ← Build2dTree(*p[]*, *X*), where NextCoord(*X*)=*Y* and NextCoord(*Y*)=*X*:

```
Function Build2dTree(p[], Coord):
• medcord ← The median Coord value of all points in p[]
• pleft[] ← Points of p[] having Coord values < medcord
• pright[] ← Points of p[] having Coord values ≥ medcord
• Construct root node    • root▷sep_line ← A Coord value separating those of pleft[] and pright[]
• root▷left ← Build2dTree(pleft, NextCoord(Coord))
• root▷right ← Build2dTree(pright, NextCoord(Coord))
• return root
```

Since calculating the median of n values requires a $O(n)$ randomized algorithm, the time complexity of the above procedure is $T(n) = 2T(n/2)+O(n)$. Solving the recurrence: $T(n) = O(n \log n)$. The space complexity is $S(n) = 2S(n/2) + S(1)$. Solving the recurrence: $S(n) = O(n)$.

A *2d tree* can be queried by recursively calling the following procedure Query(*root*, *query*, *X*):

```
Query(node, query, Coord)

if (node is leaf) report the stored point if it is contained in query
else if (Coord=X)
    if (query▷x▷hi < node▷sep_line) then Query(node▷left, query, Y)
    else if (query▷x▷lo ≥ node▷sep_line) then Query(node▷right, query, Y)
    else Query(node▷left, query, Y), Query(node▷right, query, Y)
else if (Coord=Y)
    if (query▷y▷hi < node▷sep_line) then Query(node▷left, query, X)
    else if (query▷y▷lo ≥ node▷sep_line) then Query(node▷right, query, X)
    else Query(node▷left, query, X), Query(node▷right, query, X)
```

To understand the $O(\sqrt{n})$ part of the *query* time complexity, consider a very thin horizontal *query rectangle*. At the first level (*Coord*=*X*), the search goes to both left and right directions (so now two *nodes* of the second level are visited in addition to the *root* in first level). At the second level (*Coord*=*Y*), the search goes to only one direction (so each of the two visited *nodes* in the second level will lead to one *node* in the third level, so only two more *nodes* are visited in the third level).

Thus, the number of visited *nodes* is 1 (root) + 2 (second level) + 2 (third level) + 4 (fourth level) + 4 + 8 + 8 + . . . + $2^{\frac{1}{2}\log_2 n}$ + $2^{\frac{1}{2}\log_2 n}$ (because we know that the number of added terms (levels) equals to the tree height = $\log_2 n$). Therefore the sum equals $1 + 2(2^0 + 2^1 + 2^2 + \cdots + 2^{\frac{1}{2}\log_2 n}) = 1 + 2(2^{(\frac{1}{2}\log_2 n)+1} - 1) = 4(2^{\frac{1}{2}\log_2 n}) - 1$. Since $2^{\frac{1}{2}\log_2 n} = 2^{\log_2(n^{\frac{1}{2}})} = n^{\frac{1}{2}} = \sqrt{n}$, the sum is $O(\sqrt{n})$.

A *Kd tree* can be generalized to higher dimensions by cycling through different dimensions. For example, to handle 3-dimensional *input points* (x,y,z), the first tree level should separate *points* based on their *x* coordinate values. The second tree level should separate *points* based on their *y* coordinate values. The third tree level should separate *points* based on their *z* coordinate values. The fourth tree level should separate *points* based on their *x* coordinate values, and so on.

There exist *dynamic* insert and delete operations for *Kd trees*. However, the suggested implementations are not guaranteed to effectively maintain the balance and defined characteristics of the *Kd tree*. Thus, *Kd tree* cannot be considered a *dynamic* data structure.