CMPT 280

Topic 18: k-D Trees

Mark G. Eramian

University of Saskatchewan

References

- Textbook, Chapter 18

2-D Trees

2-D Range Search

- A k-*dimensional tree* or k-D tree is a data structure that stores and organizes points in a d-dimensional space.
- A 2-D tree (kd-tree with $k = 2$) is a binary tree which stores keyed data item whose key is a pair of values each of a comparable type.
- It works like a combination of two ordered binary trees: one for the x dimension and one for the y dimension.
- At the even levels, the branch is based on the x-coordinate.
- At odd levels, the branch is based on the y-coordinate.

Recall the 2D Range Searching Problem:

Given a query range of $([x_1, x_2], [y_1, y_2])$, determine the points in the region.

2D Range Searching:

- Search the 2D kd-tree using the x range $[x_1, x_2]$ at the odd levels and the y range $[y_1, y_2]$ at even levels. When both subtrees of a node need to be searched, return the union of the found keys in the subtrees as was done in the 1D range search algorithm.
- Extra check to make sure that key in the current node has both coordinates in range, if it does, add it to the union.

2-D Trees
2-D Range Search

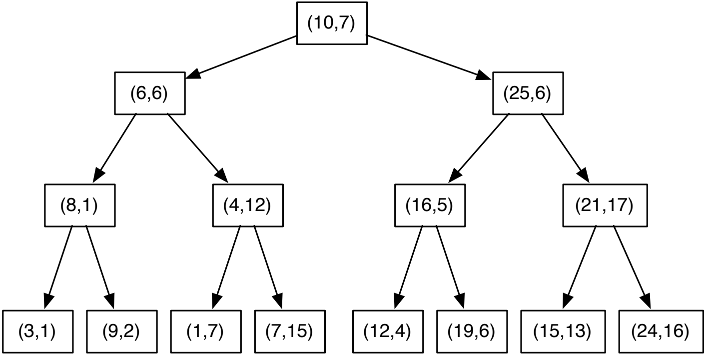
- Example: `Find all people with age between 20 and 30 who are at least 6 feet tall`.
- The basic technique can be applied to higher dimensional data. Each dimension of data takes its turn at being the one used for branching.
- This type of tree is great for database queries.

2-D Trees

2-D Range Search — Branching

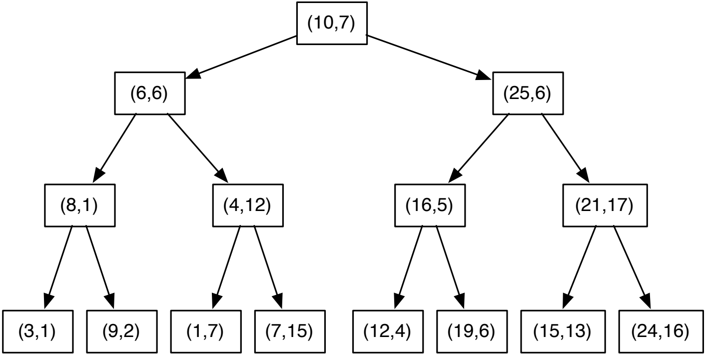
- Readings stated: Left subtrees contain coordinates where dimension (d mod depth) is **strictly less than** that of the branching coordinate and right subtrees contain coordinates where dimension (d mod depth) is **greater or equal to** that of the branching coordinate (i.e. consistent with ordered binary trees and AVL trees).
- This is only true if the set of coordinates for each dimension are unique.
- Otherwise, the left subtrees can also contain coordinates with dimension (d mod depth) **equal to** the branching coordinate.
- Consequence: You must search **both** subtrees even when the branching coordinate equals the minimum or maximum of the range.

Exercise 1



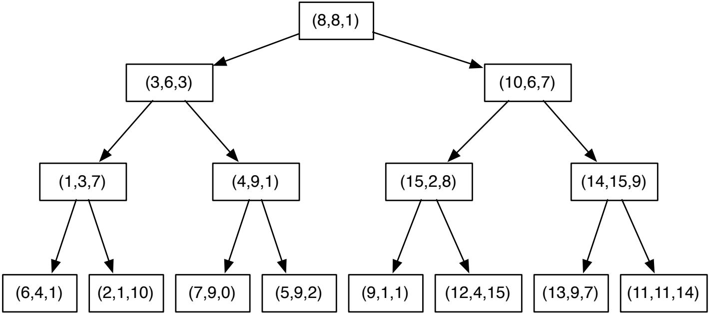
Which nodes will be visited in a 2D range search of this tree for the query range $([9, 20], [13, 16])$? Which keys are returned?

Exercise 2



Which nodes will be visited in a 2D range search of this tree for the query range $([9, 20], [4, 16])$? Which keys are returned?

Exercise 3



Which nodes will be visited in a 3D range search of this tree for the query range $([4, 9], [6, 10], [5, 12])$? Which keys are returned?

Sketch of Tree-Building Algorithm

Suppose we are given a set of keyed data items where keys are pairs of values of comparable type. To build a 2-D tree, we do the following:

```
Order the elements by the x-coordinate of their keys.
Pick the key with the median x, make this element the root of the tree.

Recursively process the keys smaller than the median by
partitioning on the other coordinate.  The median becomes the left
subtree of the root.

Recursively process the keys larger than the median by
partitioning on the other coordinate.  The median becomes the right
subtree of the root.
```

Example: Building a 2-D tree

Given the following keys (we omit showing the associated elements):

$$(1,7),(9,2),(21,17),(3,1),(10,7),(24,16),(4,12),(12,4),$$

$$(25,6),(6,6),(15,13),(7,15),(16,5),(8,1),(19,6),$$

build a 2-D tree. If we do this right, we should end up with the tree from Exercises 1 and 2. Let's go to the chalkboard...

The full example is written out in the Exercise Solutions for this topic.

Note: we must revise branching when keys for a single dimension are not unique! See how (19,6) is to the left of (25,6)? When keys are equal, we must branch both ways if the values of each coordinate in the set of points are not unique (which will usually be the case).

Building k-D Trees

- Similar process for any k – just step through each dimension at each level.
- It's straightforward on paper, but in practice it's hard to implement efficiently. Sorts are slow: $O(n \log n)$.
- A lot of effort goes into doing this without sorting, and without using extra space beyond the input list of points and the k-D tree.
- Turns out, we don't need to sort. We just need to partition the keys into median, smaller than median and larger than median. There's an $O(n)$ algorithm for that!

Next Class

- Next class reading: Chapter 19: Graphs.