Computational Geometry

Chapter 5

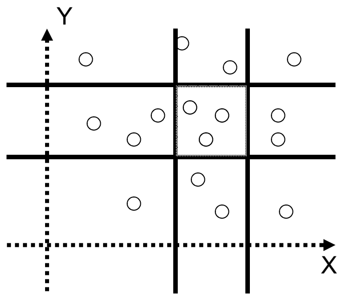
Orthogonal Range Searching

On the Agenda

- ❑ *k*-D Trees
- ❑ Range Trees

Orthogonal Range Searching

- **Problem:** Given a set of *n* points in $\Re^d$, preprocess them such that reporting or counting the *k* points inside a *d*-dimensional axis-parallel box will be efficient.



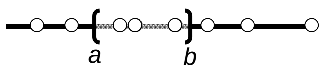
- Desired *output-sensitive* query time complexity – O(*k*+f(*n*)) for reporting and O(f(*n*)) for counting, where f(*n*)=o(*n*), e.g. f(*n*)=O(log *n)*.

- **Sample application:** Report all cities within 20 KM radius of Tel Aviv. (Here the range is actually a circle.)

Range Searching: 1D

- In a one-dimensional space, points are real numbers, and a range is defined by two numbers *a* and *b*.

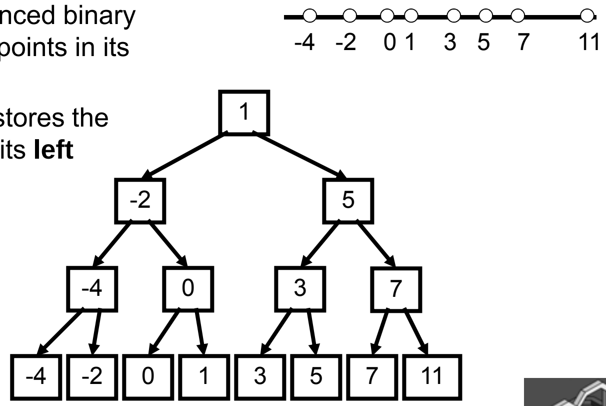
- A simple O(log *n*)-time algorithm:
  - Sort points (O(*n* log *n*) time preprocessing).
  - (Binary) search for *a* and *b* in the list (O(log *n*) time).
  - List all values in between.



- Cannot be easily generalized to higher dimensions. (Why not ?).

Range Searching: 1D Tree

- Range tree solution:
  - Sort points.
  - Construct a balanced binary tree, storing the points in its leaves.
  - Each tree node stores the **largest** value of its **left** subtree.

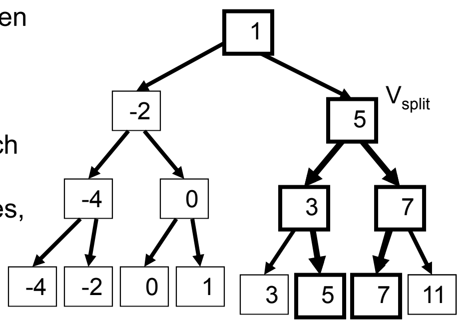


Range Searching in a 1D Tree

Input Range: 3.5-8.2

- Finding a leaf: O(log *n*) time.
- Find the two boundaries of the given range in the leaves *u* and *v*.
- Report all the leaves in *maximal subtrees* between *u* and *v*.
- Mark the vertex at which the search paths diverge as V_{split}.
- Continue to find the two boundaries, reporting values in the subtrees:

  When going towards the left (right) endpoint of the range: If going left (right), report the entire right (left) subtree.
- When reaching a leaf, it needs to be checked.



Running-Time Analysis

- k: output size
- Leaves: $O(k)$ time
- Internal nodes: $O(k)$ time (since this is a binary tree)
- Paths: $O(\log n)$ time
- Total: $O(\log n + k)$ time

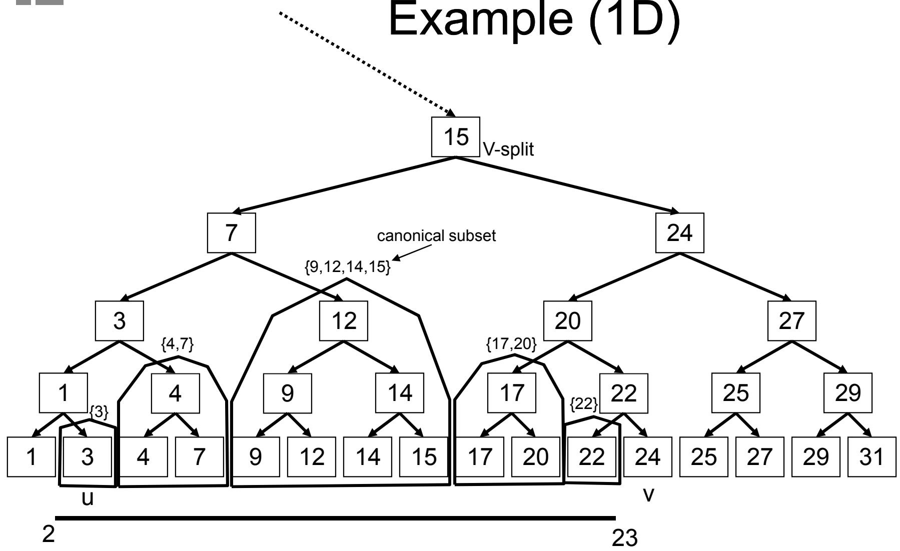
- Worst case: $k = n \rightarrow \Theta(n)$ time
- Counting: $O(\log n)$ even in the worst case. How?

General Idea

- Build a data structure storing a "small" number of canonical subsets, such that:
  - The canonical sets may overlap.
  - Every query may be answered as the union of a "small" number of canonical sets.
- Needs the geometry of the space to enable this.

- Two extremes:
  - Singletons: $O(k)$ query time, even for counting.
  - Power set: $O(1)$ query time, $O(2^n)$ storage.

Example (1D)

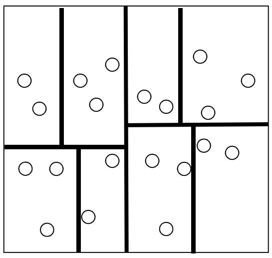


The canonical subsets are subtrees (overkill in 1D).
What is the space consumption?

2D Trees

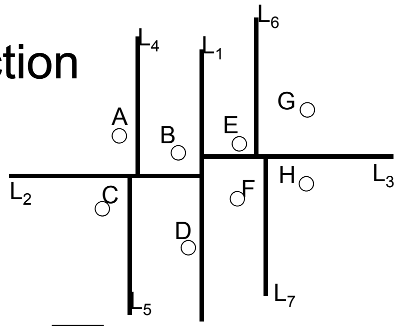
- Input: A set of points in 2D.
- Bound the points by a rectangle.
- Split the points into two equal-size subsets, using a horizontal or vertical line.
- Continue recursively to partition the subsets, alternating the directions of the lines, until point subsets are small enough (of constant size).
- Canonical subsets are subtrees.

- In higher (k) dimensions: Split directions alternate between the k axes.
- In k-D it is called "k-D tree".
  In 2-D: Used to be called "2-D tree"; now (slang) called "2-D k-D tree".



2D Tree: Construction

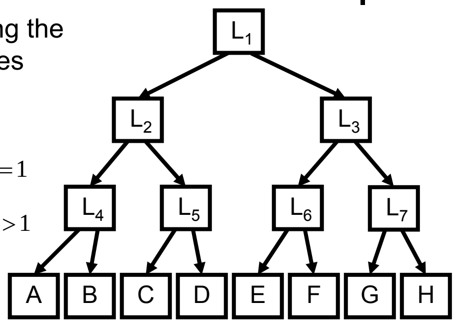
- ❑ Partition the plane into axis-aligned rectangular regions.
- ❑ Nodes represent partition lines, and leaves represent input points.



- ❑ The bottleneck is finding the median, but this requires only linear time!
- ❑ Time complexity:

$$T(n) = \begin{cases} O(1) & n = 1 \\ O(n) + 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

$$T(n) = O(n \log n)$$

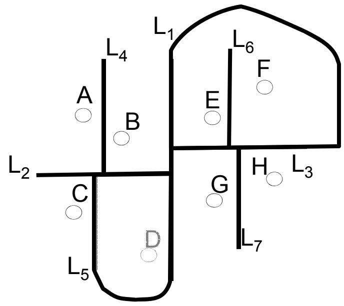


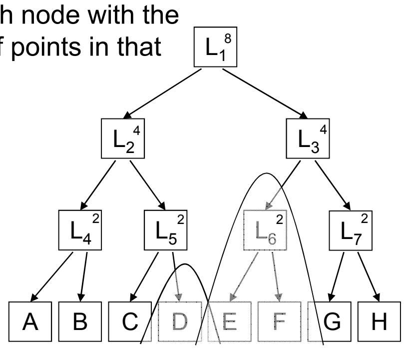
Two Possible Improvements

- ❑ Instead of finding the median from scratch each time:
  - Spend (twice) O(*n* log *n*) preprocessing time on sorting all points (once according to x, and once according to y).
  - Finding the median will be easier, but will still require linear time.
- ❑ Questions:
  - Why linear and not, say, logarithmic time?
  - Is it an asymptotic improvement?
- ❑ To overcome the last pitfall, copy the point subsets to the children trees (to avoid “jumps”). Thus, finding the median will require **constant** time. Unfortunately asymptotically there will be no improvement. Why?

Range Counting/Reporting

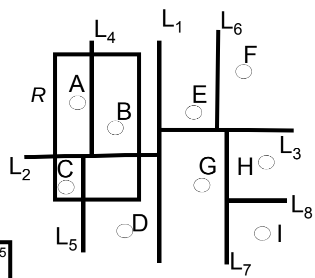
- ❑ Each node in the tree defines an axis-parallel rectangle in the plane, bounded by the lines marked by this vertex's ancestors.
- ❑ Label each node with the number of points in that rectangle.

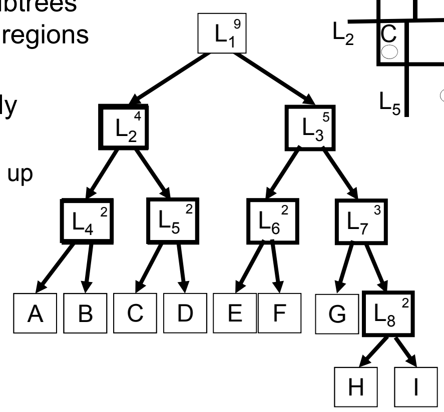




Range Counting/Reporting (cont.)

- ❑ Given an axis-parallel range query *R*, search for this range in the tree.
- ❑ Traverse only subtrees which represent regions **overlapping** *R*.
- ❑ If a subtree entirly contained in R:
  - Counting: Add up its count.
  - Reporting: Report entire subtree.





Time-Complexity Analysis

- k nodes are reported. How much time is spent on internal nodes? The nodes visited are those that are **stabbed** by R but are not contained in R. How many such cells exist?
- **Theorem**: Every side of R stabs $O(\sqrt{n})$ cells of the tree.
- **Proof**: Extend the side (w.l.o.g., horizontal) to a full line. In the first level it stabs two children, and in the next level it stabs two out of the four grandchildren. Thus, the recursive equation is:

$$Q(n) = \begin{cases} 1 & n = 1 \\ 2 + 2Q\left(\frac{n}{4}\right) & \text{else} \end{cases}$$
$$= O\left(\sqrt{n}\right)$$

- Total query time: $O(\sqrt{n} + k)$.

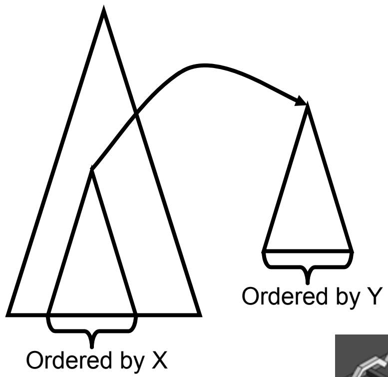
*k*d-Trees: Higher Dimensions

- For a d-dimensional space:
  - Same algorithm.
  - $O(d)$ time is needed to handle a point.
  - Construction time: $O(d\,n \log n)$.
  - Space Complexity: $O(d\,n)$.
  - Query time complexity: $O(d\,(n^{1-1/d}+k))$.
- Note: For large d, full scan is almost equally good!
- **Question:** Are kd-trees useful for non-orthogonal range queries, e.g., disks, convex polygons?

- Fact: After $O(d\,n \log^{d-1} n)$ preprocessing time, using d-D *range trees*, orthogonal range queries can be solved in $O(d\,(\log^{d-1} n + k))$ time using $O(d\,n \log^{d-1} n)$ space.

Multi-Level Data Structure

- Construct a tree ordered by *x* coordinates.
- Each inner vertex *v* contains a pointer to a secondary tree, that contains all the points of the primary subtree ordered by *y* coordinates.
- Points are stored **only** in the secondary trees.



Range Tree: Construction

- Same as a 1D-Tree, except that in each level the secondary trees are built as well.
- **Theorem**: The space complexity is $\Theta(n \log n)$.
- **Proof**: The size of the primary tree is $\Theta(n)$. Each of its $\Theta(\log n)$ levels corresponds to a collection of secondary trees that contains **all** the n points.
- Construction time (naïve analysis):

$$T(n) = \begin{cases} O(1) & n = 1 \\ O(n \log n) + 2T\left(\frac{n}{2}\right) & \text{else} \end{cases}$$

$$= O(n \log^2 n)$$

Range Tree: Improved Construction

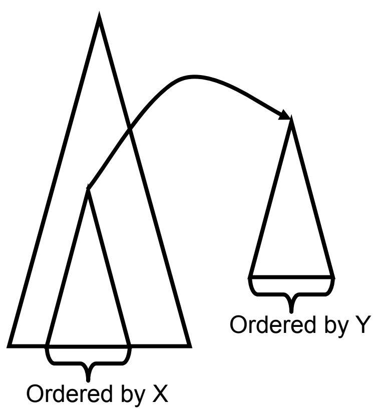
- However, there is no need for **repeated** sorting by *y* coordinates!
- Instead, we can sort by *y* only **once** (in O(*n* log *n*) time), and copy data in the recursive calls in linear time.
- The resulting recursive equation is:

$$T(n) = \begin{cases} O(1) & n = 1 \\ O(n) + 2T\left(\frac{n}{2}\right) & \text{else} \end{cases}$$
$$= O(n \log n)$$

- Overall: O(*n* log *n*) time.

Range Tree: Search

- Given a 2D range, we simulate a 1D search and find subtrees sorted by *x*.
- Instead of reporting the entire subtrees, we invoke a search in the secondary trees sorted by *y*, and report only the points in the query range.



Search: Analysis

- Time complexity:

$$T(n) = O(\log n) + \sum_{v} (\log n + k_v) = O(\log^2 n + k)$$

traversing primary structure | calls to secondary structure | traversing secondary structure | reporting

- The running time can be reduced to O(log n + k) by using *fractional cascading*.

Points in Non-General Position

- Question: How can we handle sets of points which are not in general position, i.e., with multiple points with the same x coordinate?
- Answer: By two-step order checks. When comparing according to x, resolve ties by y, and vice versa.
- This splits points into two sides, having the same effect as infinitesimally rotating the plane.
- **Theorem**: The modified order checks preserve the correctness of the algorithms.