Forests of search trees

Stanislav Protasov

Agenda

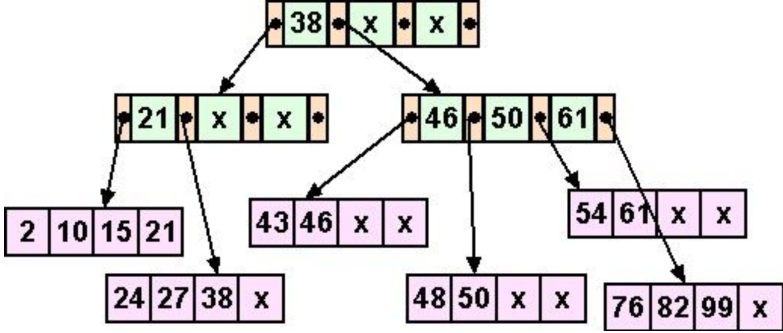
ANNS with trees:

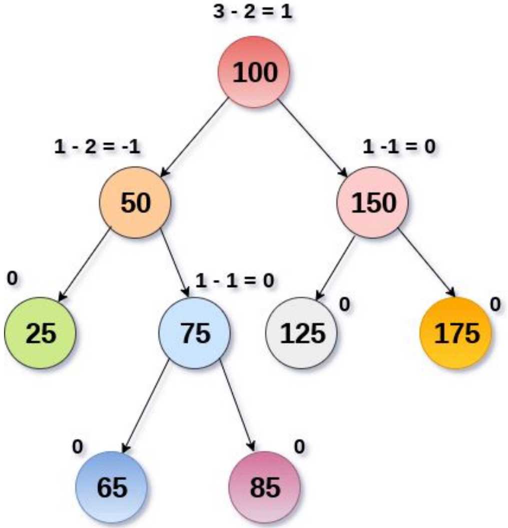
- Search trees
- Quad trees
- KD-trees
- Annoy
- And some others

Search Trees

Refresher for [B]ST

- K-ary (usually binary) trees
- Built upon comparable keys (scalars)
- Similar search procedure
- Preserved balance property, ensures O(log(N)) max path length
- Can be *homogeneous* (AVL) and not (*B+ tree*)

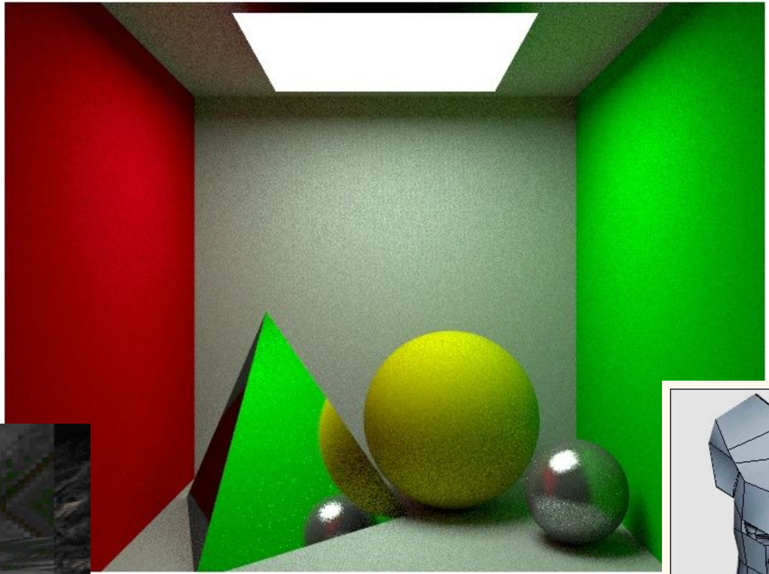


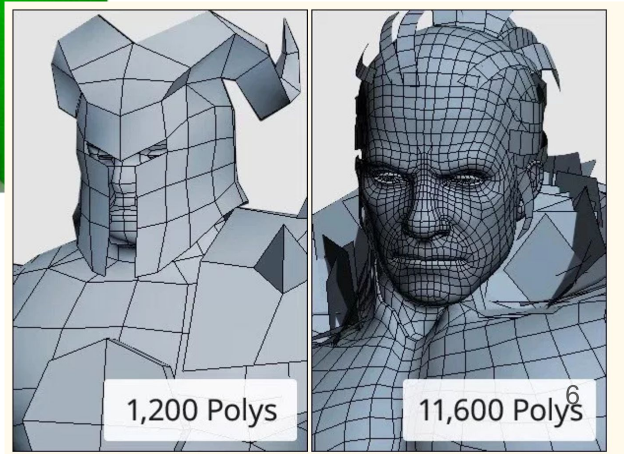


AVL Tree

But what if we have vectors?

Originated from Computer Graphics



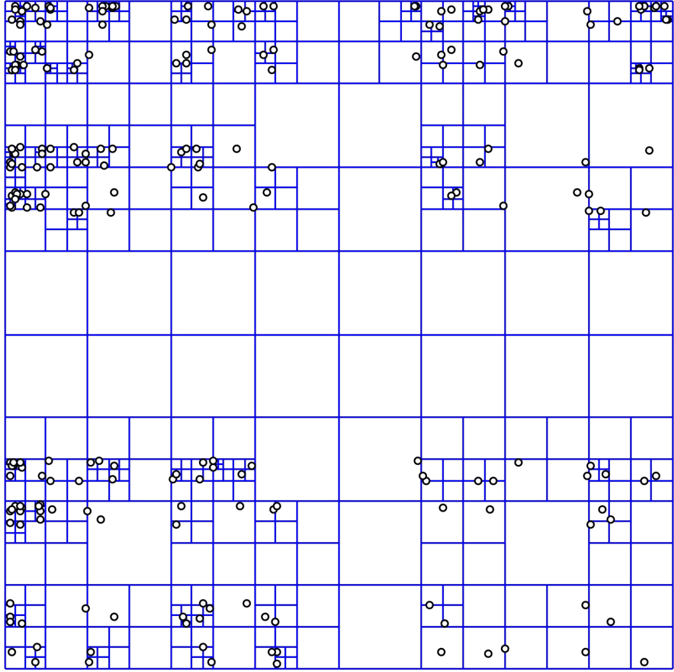


Trivial case: vector is a scalar

- Binary search trees:
  - Splay, RB, AVL trees are best for RAM
- N-ary search trees:
  - B-trees, LSM-trees are used with hard drives
- **Search:**
  - **Exact search is O(log(N))**
  - **K nearest neighbour search O(log(N) + K)**
  - **Range search O(log(N) + K)**

QuadTree (1974)

- Forms of Quad Trees:
  - Region
  - Point
  - Edge
  - Polygon



- All forms of quadtrees share some common features:
  - decompose space into **adaptable** cells
  - Each cell (or bucket) has a **maximum capacity**.
    When maximum capacity is reached, the bucket **splits**

QuadTree search

```
function queryRange(range) {
  pointsInRange = [];
  if (!this.boundary.intersects(range))
    return pointsInRange;

  for (int p = 0; p < this.points.size; p++) {
    if (range.containsPoint(this.points[p]))
      pointsInRange.append(this.points[p]);
  }
  if (this.northWest == null) // no children
    return pointsInRange;

  pointsInRange.appendArray(this.northWest->queryRange(range));
  pointsInRange.appendArray(this.northEast->queryRange(range));
  pointsInRange.appendArray(this.southWest->queryRange(range));
  pointsInRange.appendArray(this.southEast->queryRange(range));
  return pointsInRange;
}
```

QuadTree insertion #1

```
function insert(p) {
   if (!this.boundary.containsPoint(p))
     return false; // object cannot be added
   if (this.points.size < QT_NODE_CAPACITY && northWest == null) {
     this.points.append(p);
     return true;
   }
   if (this.northWest == null) this.subdivide();

   if (this.northWest->insert(p)) return true;
   if (this.northEast->insert(p)) return true;
   if (this.southWest->insert(p)) return true;
   if (this.southEast->insert(p)) return true;
}
```

QuadTree insertion #2

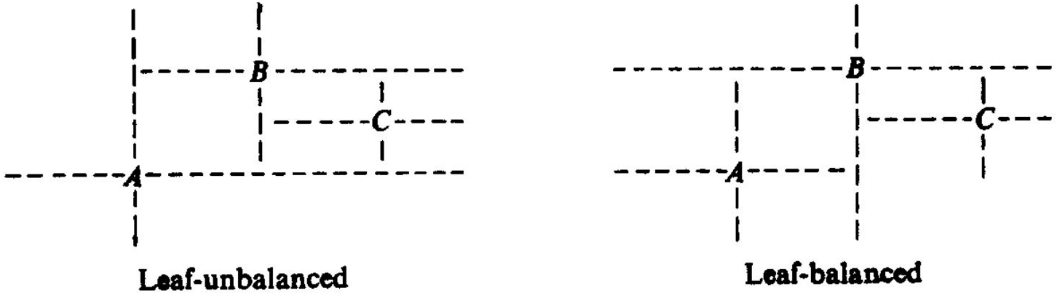


Fig. 2. Single balance

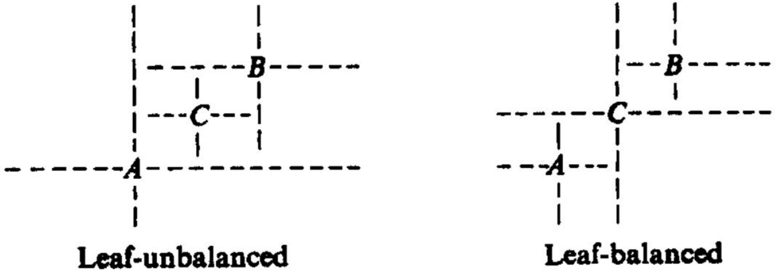


Fig. 3. Double balance

Table 2. Data on leaf-balanced insertion

QuadTree deletion

*<<… In fact, it seems that **one cannot do better than to reinsert all of the stranded nodes**, one by one, into the new tree. This answer is not very satisfactory, and it is a matter of some interest whether there exists any merging algorithm that works faster than* `n log n`*, where* `n` *is the total number of nodes in the two trees to be merged…>>*

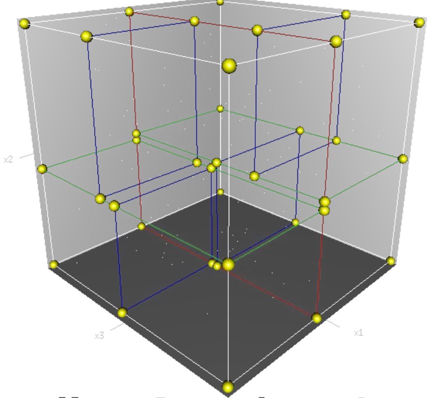
QuadTree optimization

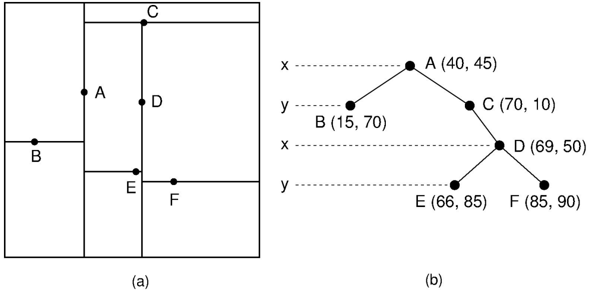
By an **optimized tree** we will mean a quad tree such that every node K has this property: **No subtree of K accounts for more than ½** of the nodes in the tree whose root is K.

A simple recursive algorithm to complete optimization is this: Given a collection of **lexicographically ordered records**, we will first find one, **R**, which is to serve as the root of the collection, and then we will regroup the nodes into 4 subcollections which will be the four subtrees of R. The process will be called recursively on each subcollection… No subtree can possibly contain more than half the total number of nodes

Can you see any suboptimality?

[K-d trees](#) (1975)

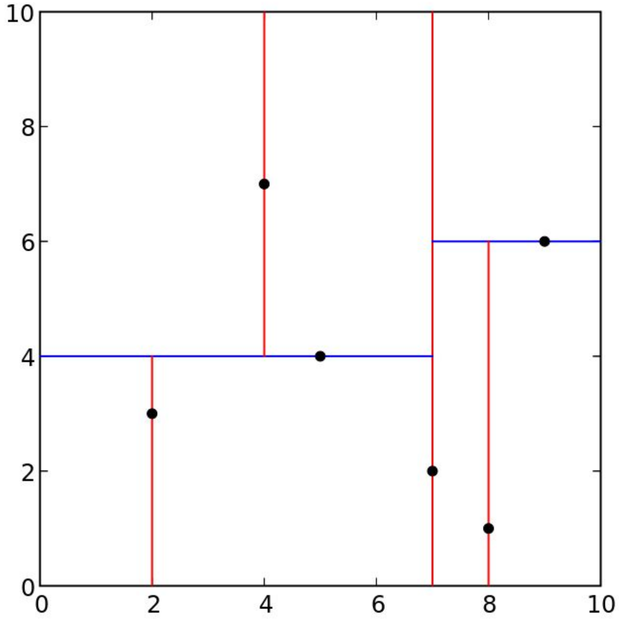


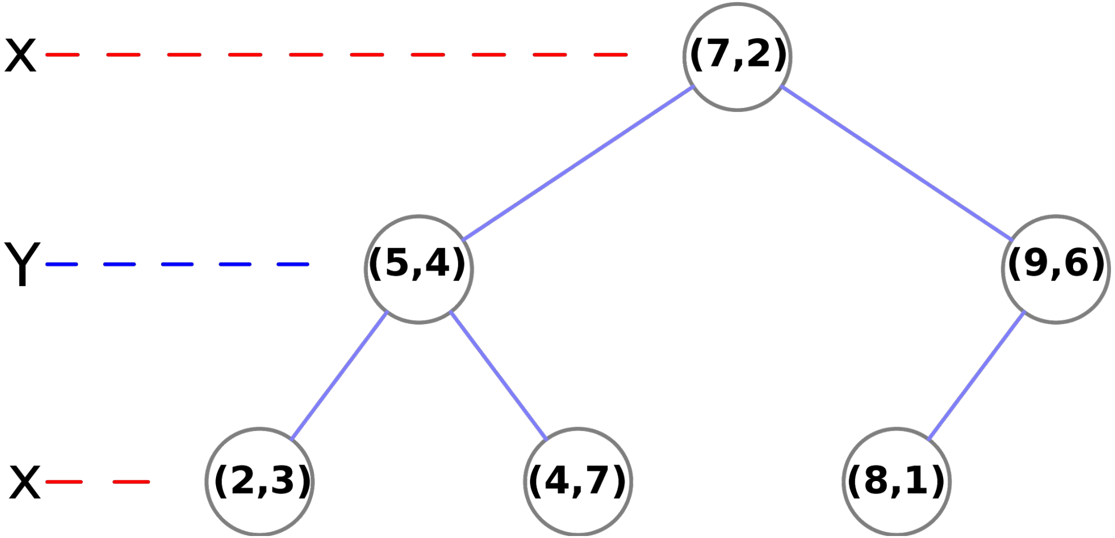


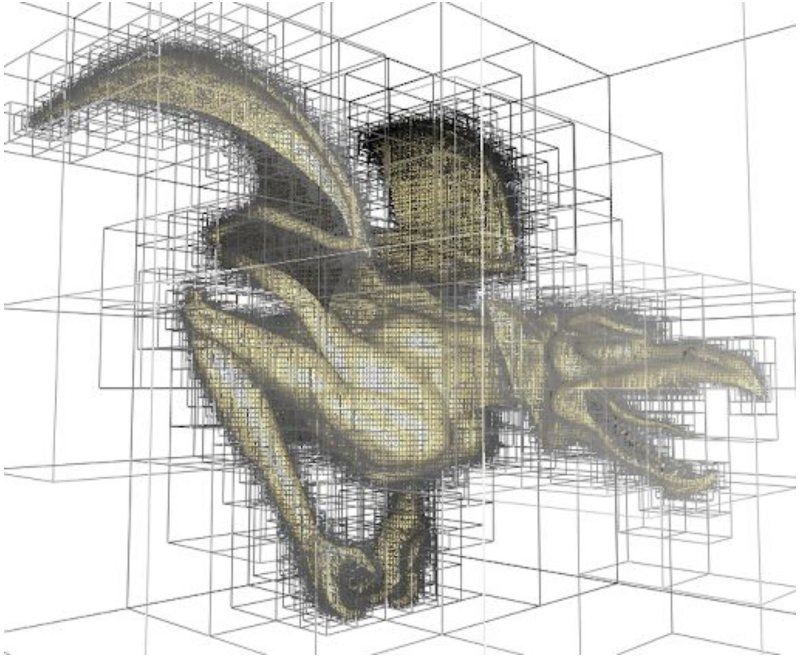
Ideas:

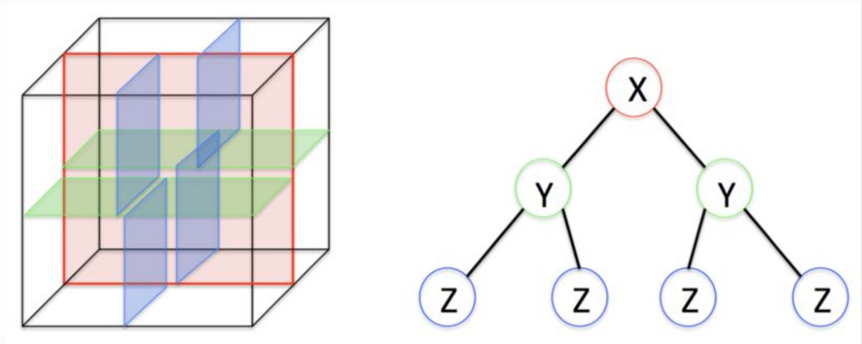
- Split points in 2 **equal (by #points)** subspaces, not 4
- Use **alternating coordinates** at each level `(x, y, z, x, y, z, ...)`
  - Thus, we need 2 levels to encode quadrants, but they are **equal**
  - And yes, this allows us to have **more than 2 dimensions**
- [**Cool demo**](#)

K-d trees: building example








X
Y
Y
Z
Z
Z
Z

K-d trees

Construction ("homogeneous"):

```
def buildKDTree(vectors, dim=0):
    if not vectors:
        return None      # stop condition, e.g.
    if len(vector) == 1:
        return Node(vector[0])
    vectors.sort(key = lambda x: x[dim])  # or Selection alg for O(N)
    med = len(vectors) // 2               # this will work only for no dups!
    left, med, right = vectors[:med], vectors[med], vectors[med+1:]
    node = Node(med)
    node.left = buildKDTree(left, (dim + 1) % K)
    node.right = buildKDTree(right, (dim + 1) % K)
    return node
```

K-d trees characteristics

Is built in $O(n(k + log(n)))$ time

Requires O(*kn*) memory (at most node for a point)

Runs range search for $O(n^{1-\frac{1}{k}} + a)$ where *a* — result size

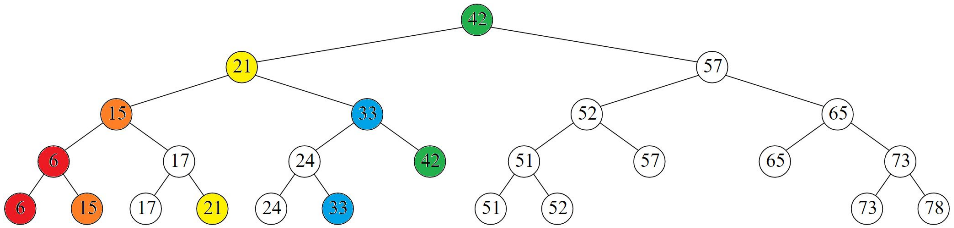
Runs 1-NN search in O(log(*n*)) time

To build hyperplanes it requires vector representation of keys

Faster range queries - range trees (1979)

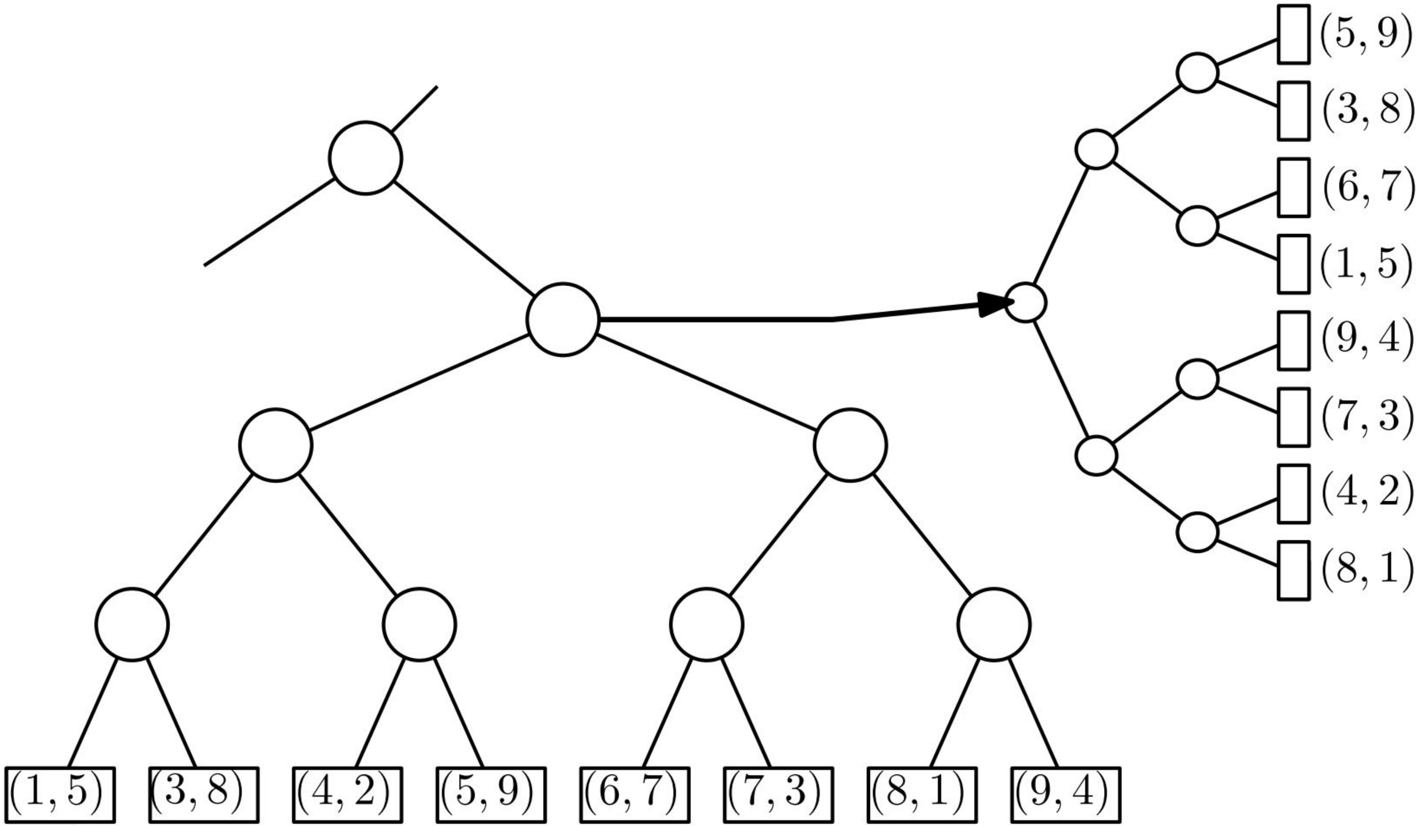
Ponts are in the leaves.

For **1-dimensional** case: **balanced non-homogeneous binary search tree** on those points. Internal nodes store predecessors (largest to the left)



Range trees in **higher dimensions** are constructed recursively by constructing a balanced binary **search tree on the first coordinate** of the points, and then, for **each vertex v** in this tree, constructing a **(d−1)-dimensional range tree** on the points contained in the **subtree of v**

Image source link



Sorting and looking for median is soooo boring...

Johnson-Lindenstrauss lemma

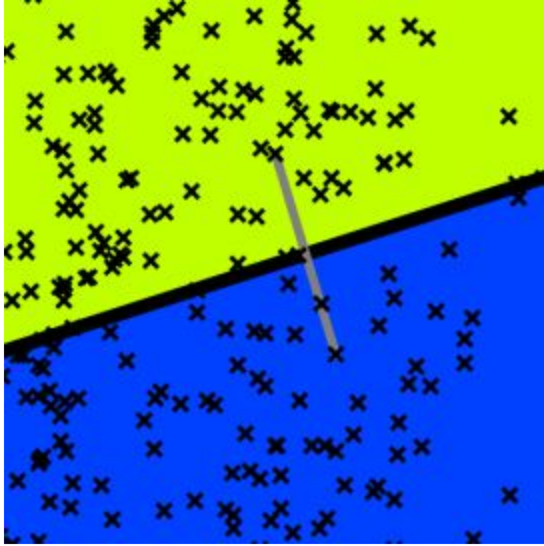
… **low-distortion embeddings** of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that **distances between the points are nearly preserved**. (Random projections).

Given $0 < \varepsilon < 1$, a set X of m points in $\mathbb{R}^N$, and a number $n > 8\ln(m)/\varepsilon^2$, there is a linear map $f : \mathbb{R}^N \to \mathbb{R}^n$ such that

$$(1-\varepsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\varepsilon)\|u-v\|^2$$

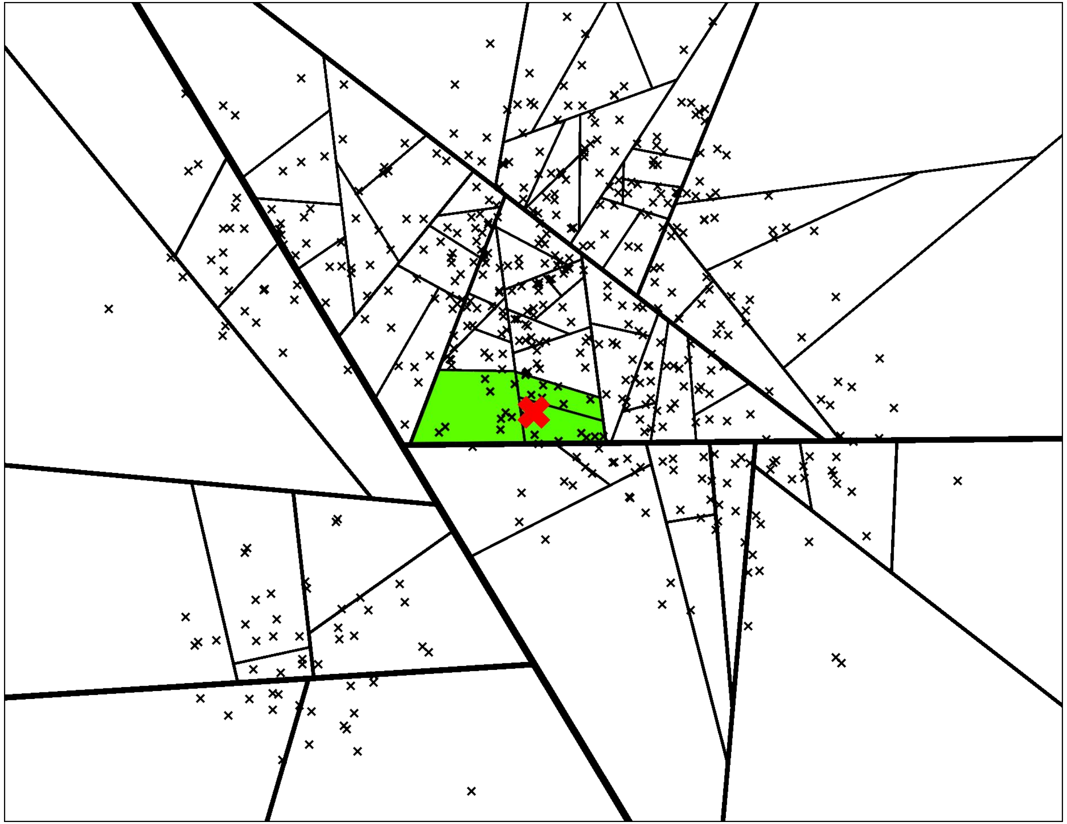
for all $u, v \in X$.

Annoy from Spotify (2015)

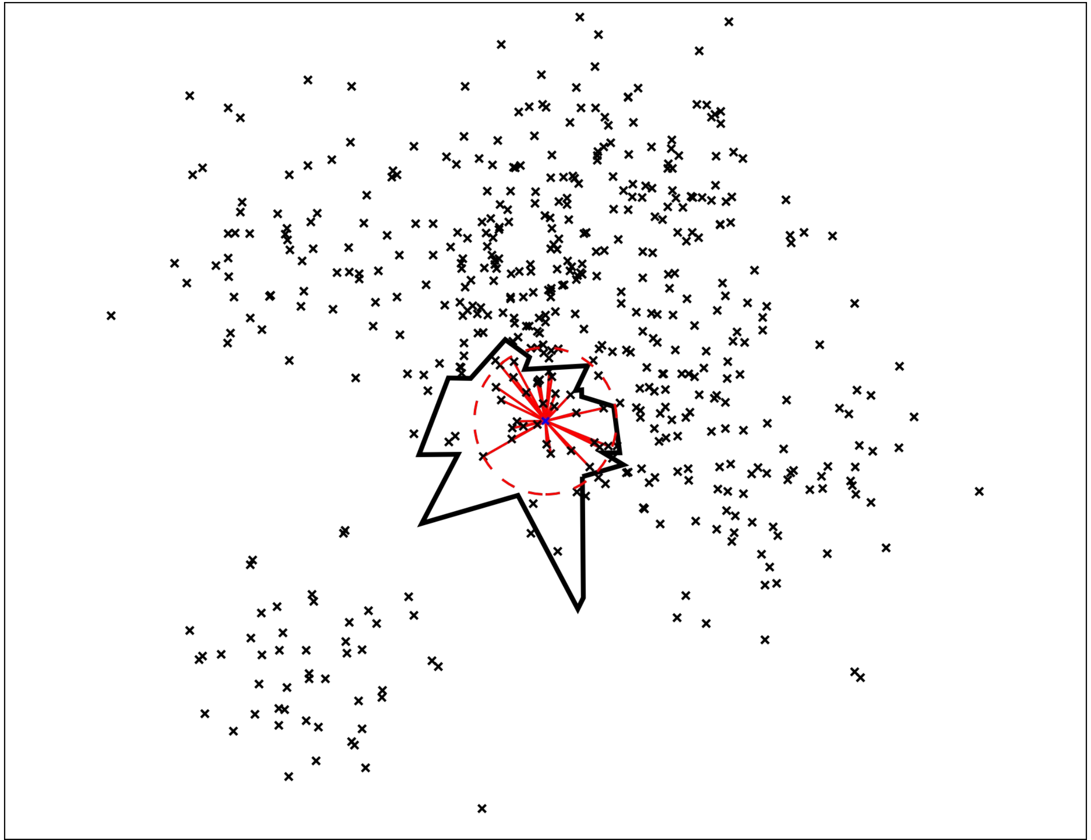


1. Instead of looking for a median, **select equidistant hyperplane for 2 random points** - then split is done in linear time (random projection)
2. Use "**soft threshold**" that allows traversing "wrong" branches for ANNS
3. Build **multiple search trees** over the same dataset (*compare to multiple searches in NSW*)
4. Generalization of binary space partitioning (BSP-tree) used in CG (Doom, Quake, ...) for visibility sorting.

Multiple trees (animation)



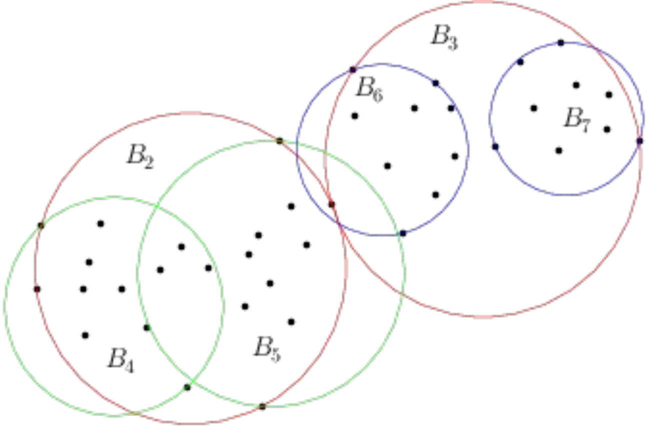
ANNS results with Annoy



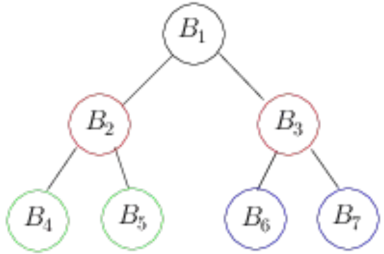
Oh no, I don’t have vectors!

Metric space

Vantage-point (VP) trees (1991)



Instead of dividing space by a plane, we can divide if by a **sphere** (or nested spheres, recursively). **Sphere** requires only **center** (one of dataset points) and **radius** (which can be estimated in any **metric space**). Radius is selected to split points into equal parts.




```
function Select_vp(S)
    P := Random sample of S;
    best_spread := 0;
    for p ∈ P
        D := Random sample of S;
        mu := Median_{d∈D} d(p, d);
        spread := 2nd-Moment_{d∈D} (d(p, d) − mu);
        if spread > best_spread
            best_spread := spread; best_p := p;
    return best_p;
```

```
function Make_vp_tree(S)
    if S = ∅ then return ∅
    new(node);
    node↑.p := Select_vp(S);
    node↑.mu := Median_{s∈S} d(p, s);
    L := {s ∈ S − {p} | d(p, s) < mu};
    R := {s ∈ S − {p} | d(p, s) ≥ mu};
    node↑.left := Make_vp_tree(L);
    node↑.right := Make_vp_tree(R)
    return node;
```

SEARCH

Ok, you must be lost...

All those trees recursively split the space into similar size parts

Quad Tree - works in R^2 only. Each node splits space into 4 non equal quadrants.

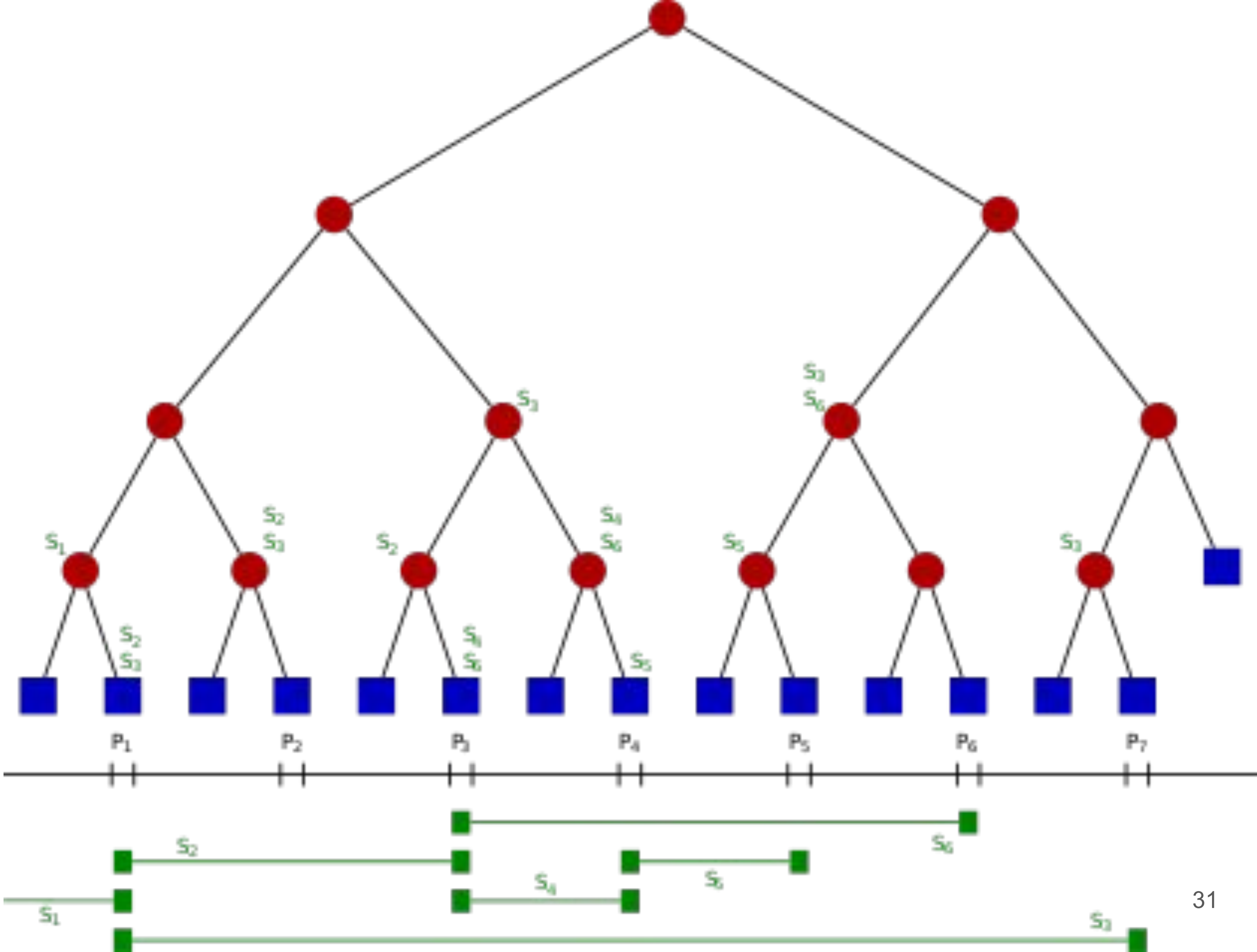
K-d Tree - works in R^K. Each node splits space into 2 equal parts.

Annoy - works in R^K. Instead of sorting and finding median - uses random separating hyperplanes. But compensate with multiple trees

Vantage-point tree - works for any metric space. Instead of hyperplanes uses spheres.

Offtopic: interval and BSP trees. When object is not a point

Interval tree



Interval tree

Tree that **holds intervals** and allows to search fastly which of them overlap the query (point or interval).

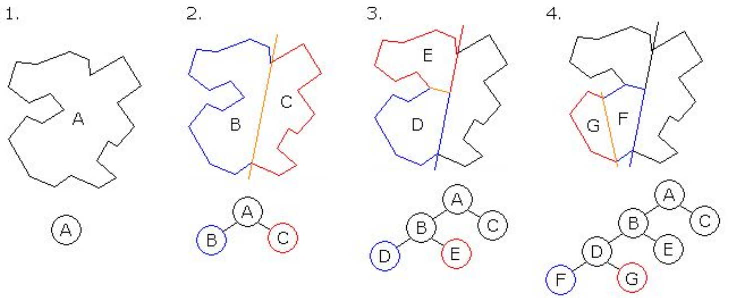
`Construction(L):`

1. You have a list of intervals ***L***.
2. By X_{center} split all intervals into “left”, “intersecting”, “right” lists.
3. Store “intersecting” in current node in 2 lists (sorted by start and by end).
4. Run `Construction(“left”)` and `Construction(“right”)` intervals.

`Search(p, node):`

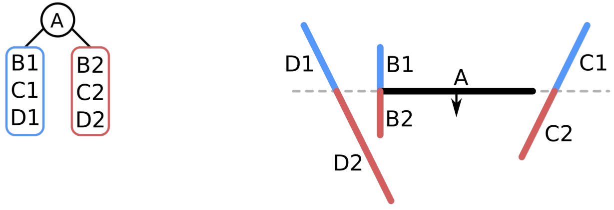
1. Compare p to node.X_{center}
2. Use sorted list in node to find intersecting
3. Go `Search(p, node.[left|right])` with respect to [1]

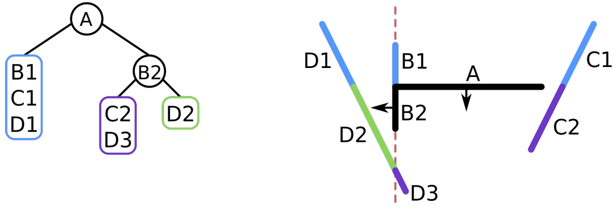
BSP-tree



To store polygons in a list:

- Choose a polygon *P* from a list *L*.
- Make a node *N*, and add *P* to the list of *N*.
- For each other polygon Q in the list:
    - If *Q* is <u>in front of</u> *P* plane, move Q to the list L_F ***"in front of P"***.
    - If *Q* is <u>behind</u> *P* plane, move Q to the list L_B ***"behind P"***.
    - If *Q* <u>intersects</u> *P* plane, **split** it into two polygons and move them to the respective lists.
    - If that polygon lies in the plane containing *P*, add it to the **list of *N***.
- Apply this algorithm to L_F and L_B.





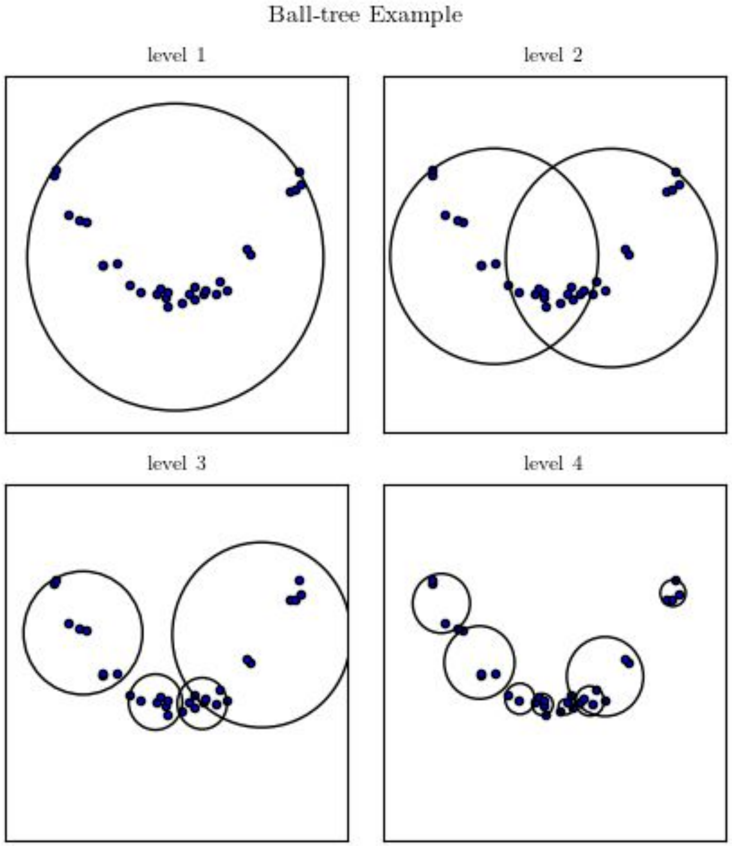
Any other trees left?
Yes, and lots of!

Ball-trees (1989)

1. Select a dimension of biggest variance
2. Split by pivot element (median)
3. Construct ball tree for “lefts” and “rights”.

Branch-and-bound powered:

Algorithm is searching the data structure with a test point *t*, and has already seen some point *p* that is closest to *t* among the points encountered so far, then any ***subtree whose ball is further*** from *t* than *p* can be ignored for the rest of the search.



See also

M-trees

R-trees and R*-trees

Octree

…