Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

K-d trees

prerequisites:

Binary Search Tree.

K-d trees
introduction

There are multiple problems related to **multidimensional searches**:

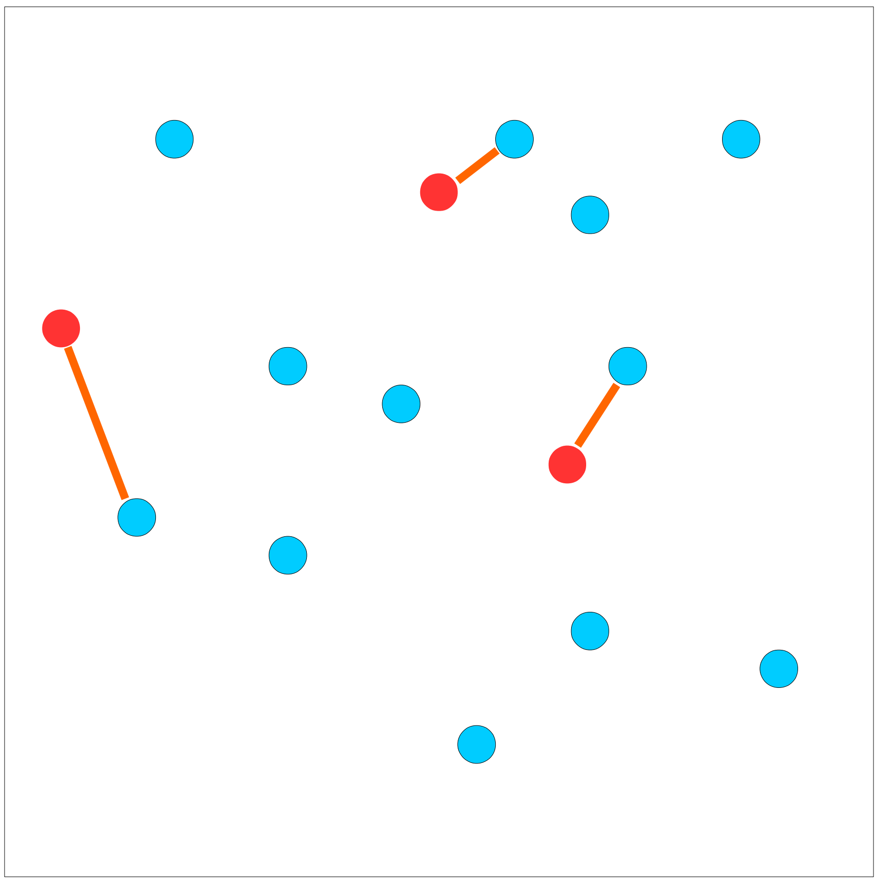
range search,
nearest neighbor search,
handling extensional objects, ...

And there is a variety of data structures which address that problems:

Quadtrees,
K-d trees,
Range trees,
Interval trees, ...

However, more often K-d trees are used for **nearest neighbor searches**, as they provide better time complexity there.

Nearest neighbor search



Given a collection of points in *N*-dimensional space,

And a query point in it,

Find the point which is **closest** to the query point.

Definition of NN search problem

Nearest neighbor search

usage

1) Various geometrical problems:

approximate search for objects...

2) Machine learning:

*supervised ML, both classification and regression (better known as **k-NN**, standing for k-nearest neighbors)...*

3) N-body problem:

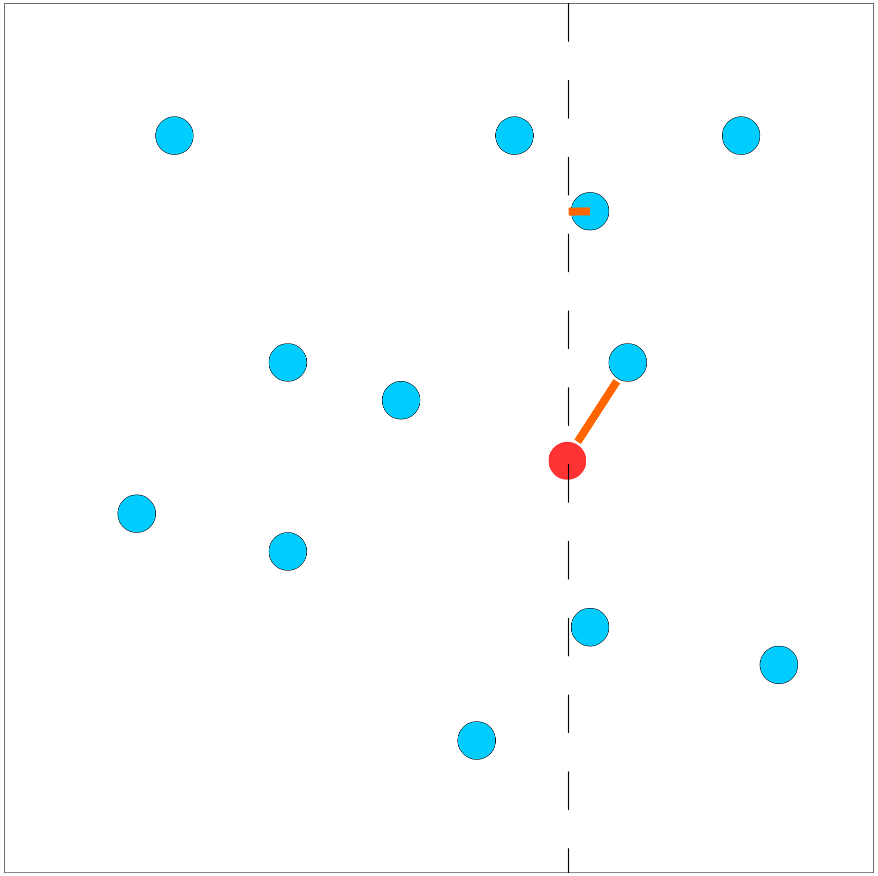
simulating gravitational interaction between N bodies, precalculating influence of entire regions to individual bodies...

4) Color reduction:

compress given image, using at most M colors to represent it...

Nearest neighbor search

trivial solutions



Sorting all points only by X-coordinate will not help...

Closest by X-coordinate doesn't mean the **closest**.

... and vice versa.

Sorting all points only by X-coordinate

Nearest neighbor search

trivial solutions

Same about sorting only by Y-coordinate, or any other direction...

So, we must somehow take into account both X and Y coordinates **simultaneously**.

K-d trees

Invented in **1975**, by **Jon L. Bentley**.

Jon Louis Bentley,
Stanford University

K-d trees

At first, let's concentrate on **2-dimensional** K-d trees:

Structure:

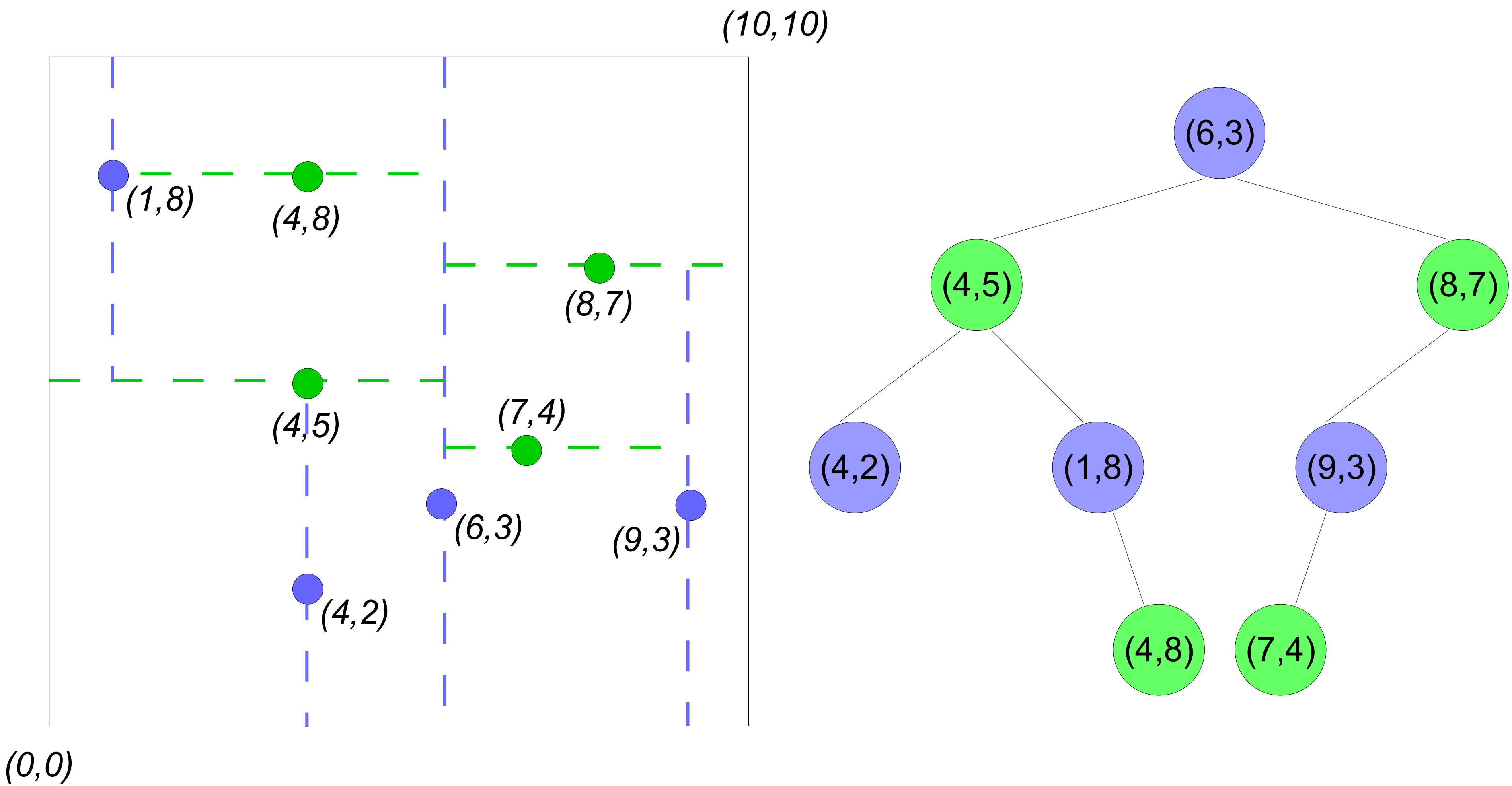
1) A simple binary tree.

2) Every 2D point corresponds to a node, and vice-versa,
both leaves and intermediate nodes do store points.

3) Intermediate nodes can act slightly different:
an intermediate node splits it's are either by **X-axis** (vertical line),
or by **Y-axis** (horizontal line).

K-d trees

examples



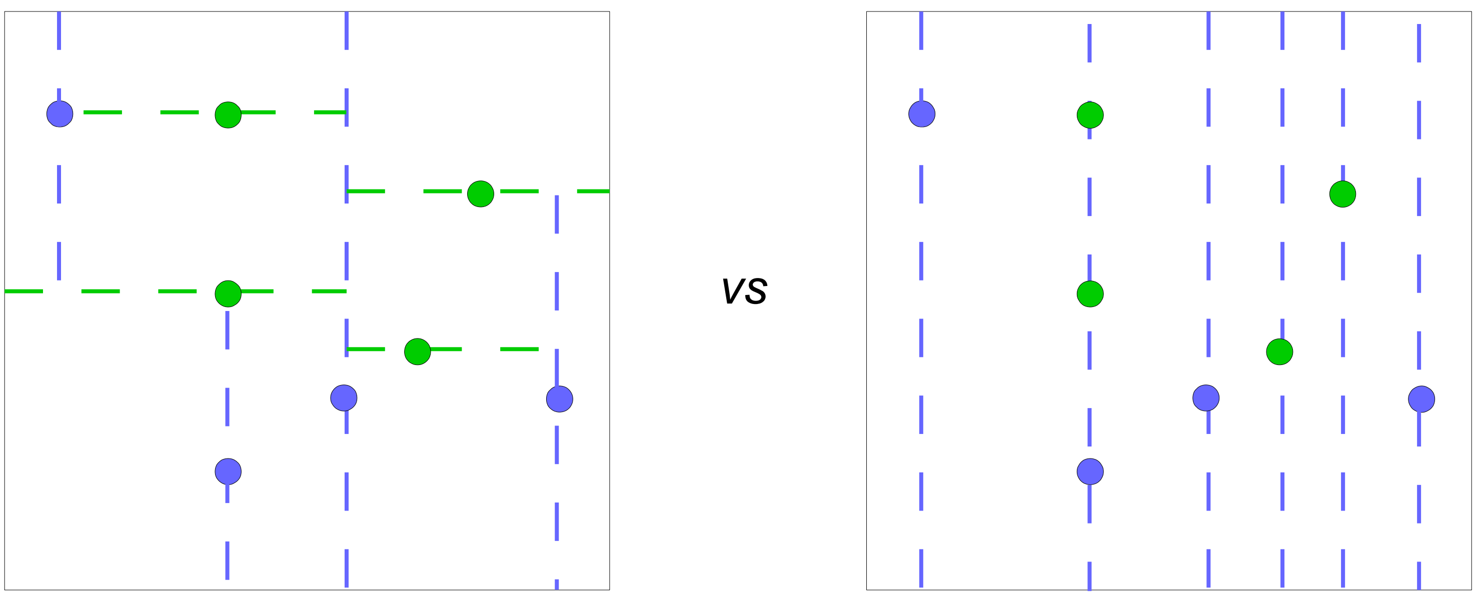
An incremental example of a K-d tree.

K-d trees

properties

Generally, the axes by which splits are performed, do interleave:
... so we have – XYXYXY...

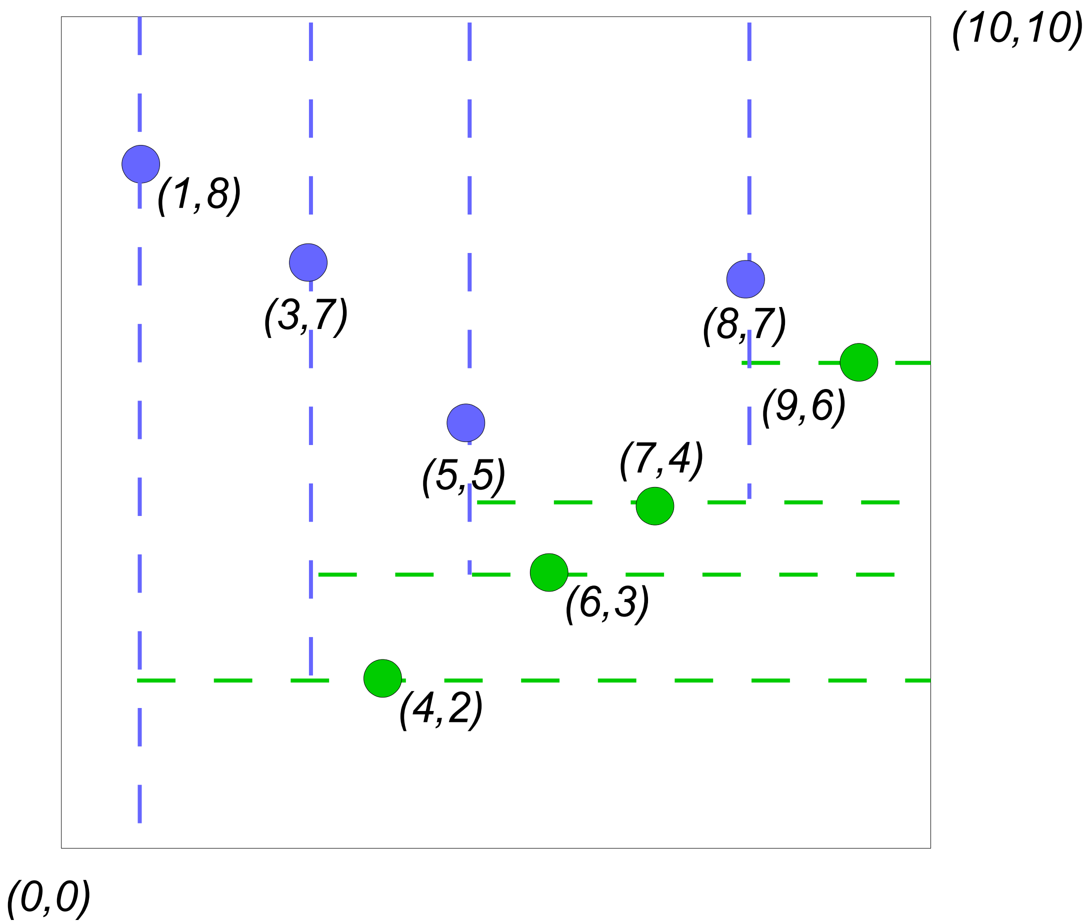
Property #1: Area corresponding to each node is more like a ***"square"*** and not a ***"rectangle"***.

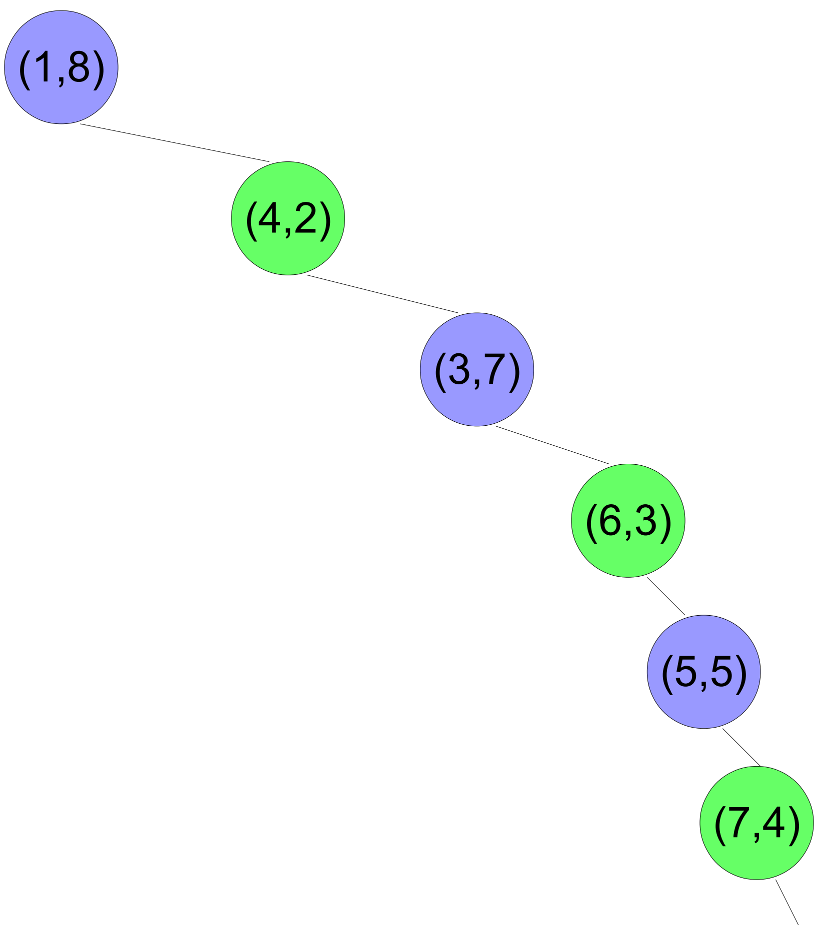


K-d trees

properties

Property #2: If K-d tree has "*n*" points, then:
best partitioning will result in "$log_2 n$" tree height,
worst partitioning will result in "*n*" tree height.





K-d trees

properties

Those estimates **do not depend** on
coordinates of the points, neither on
sequence of split axes e.g. XYXYXY...

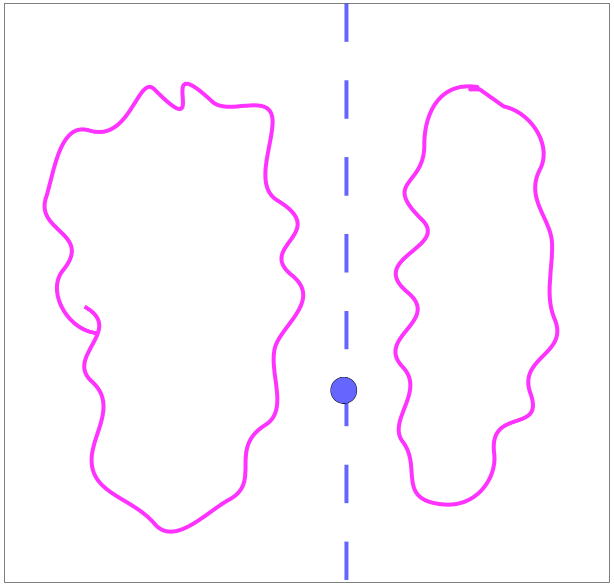
Exercise:

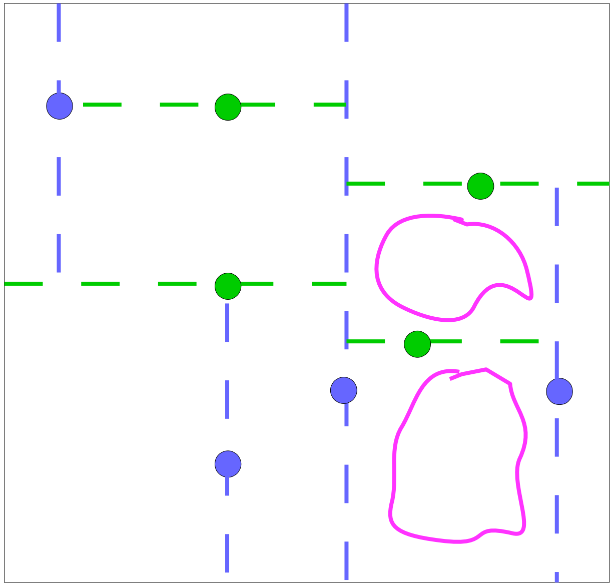
In which order points of the previous example should be inserted, for us to have a more or less balanced K-d tree?

K-d trees

properties

Property #3: After any split, subtree of the first half is **not related** anymore to subtree of the second half.





K-d trees

exercises

Exercise:

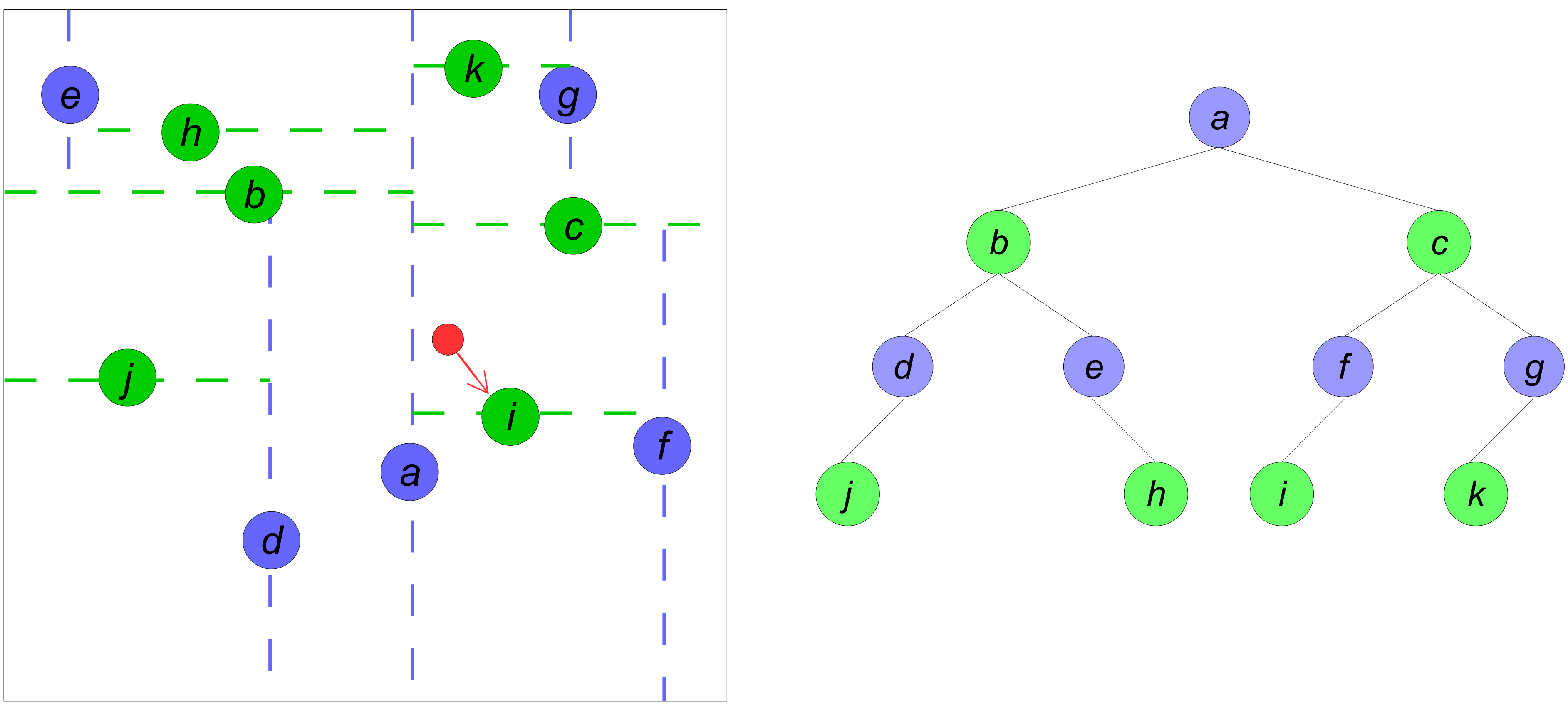
Insert the following points into an initially empty K-d tree:
(2,3) , (8,7) , (5,1) , (1,9) , (7,4) , (4,8) , (3,2) , (9,6).

K-d trees

nearest neighbor search

Top use of K-d trees is to perform **nearest neighbor** (NN) search.

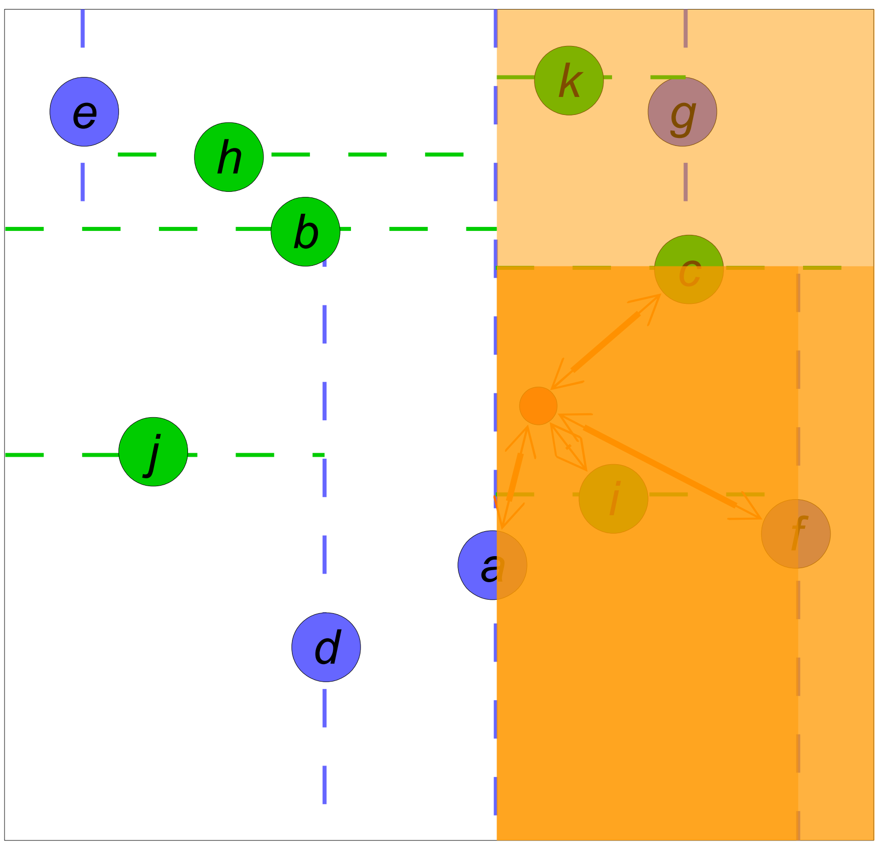
Given a **query point** "*q*", find the point from K-d tree which is closest to "*q*".

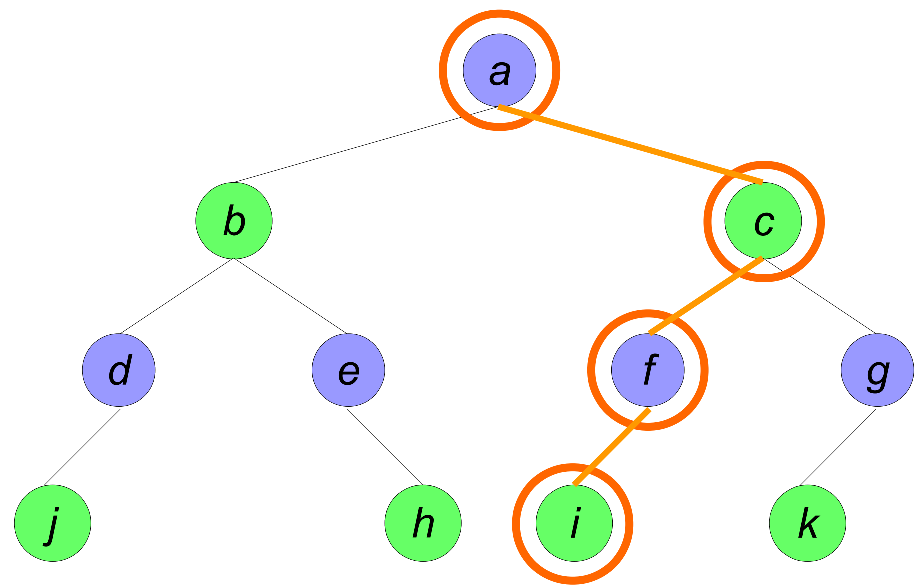


K-d trees

nearest neighbor search

We'll try to search by the correct branch in the tree, every time **updating current best** distance.



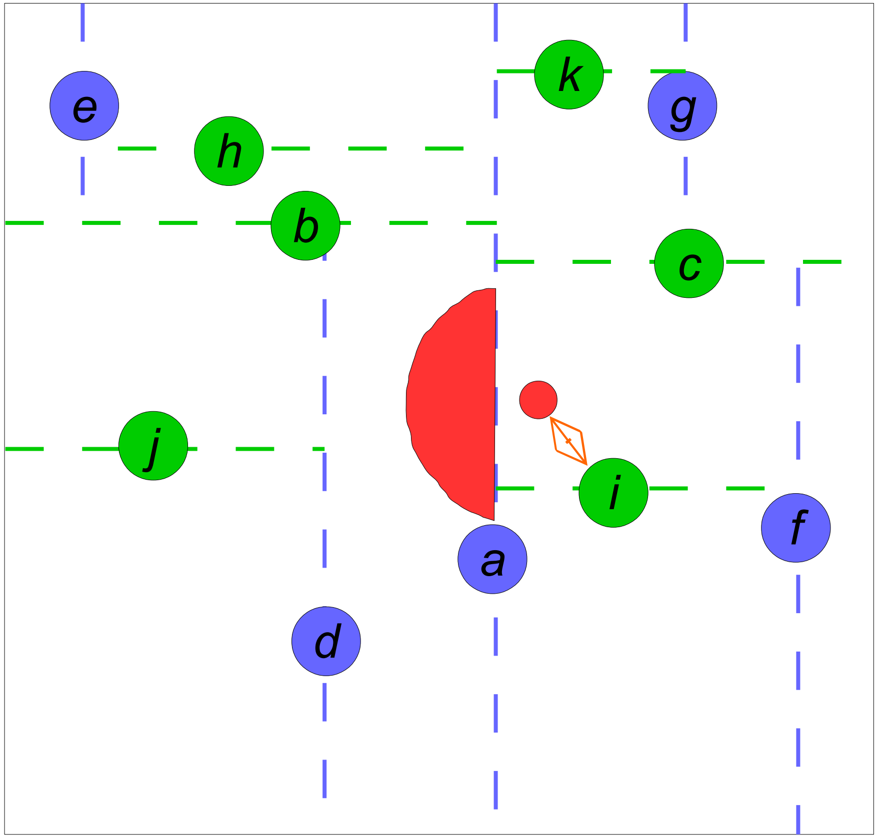


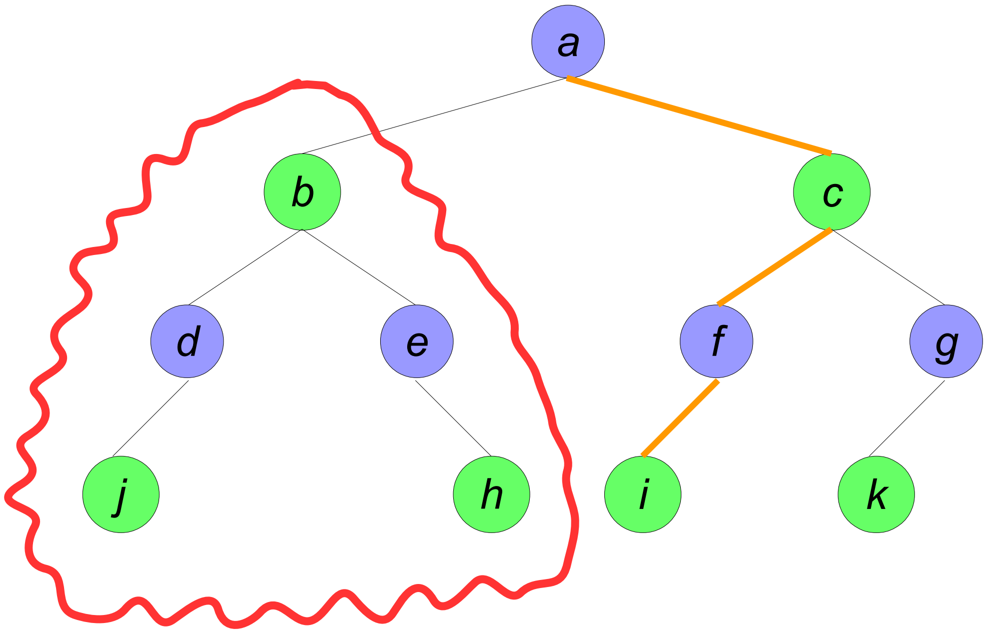
K-d trees

nearest neighbor search

On this example we have found the proper answer - "*i*".

However, if there would be some extra **points in the left half**, our answer will become incorrect.





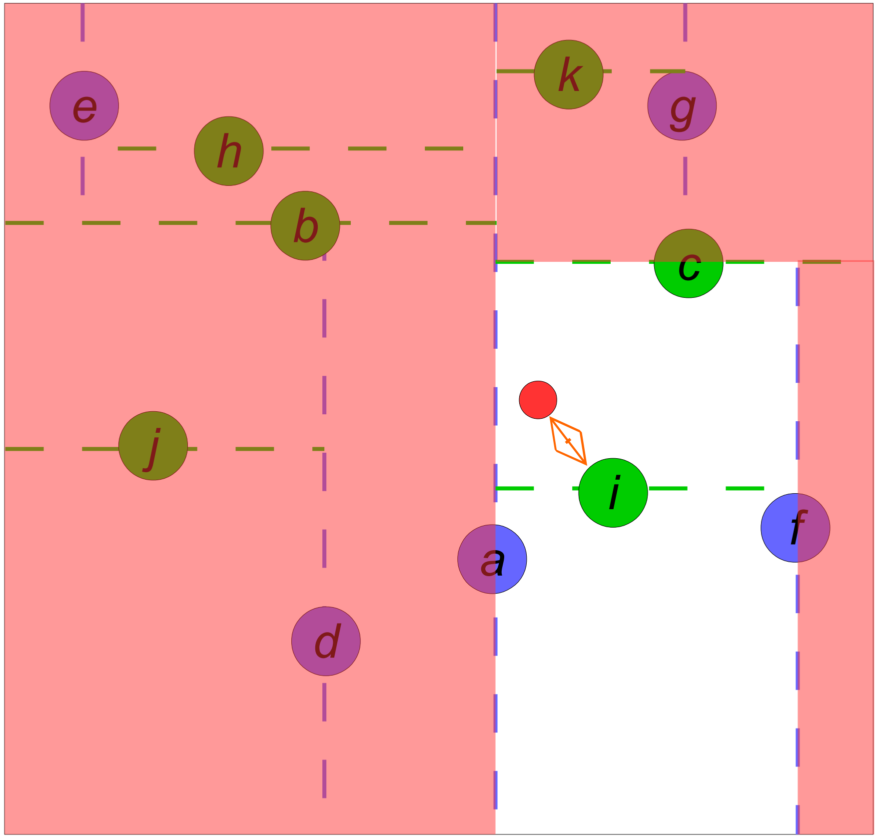
So, do we need to observe also the other branch?

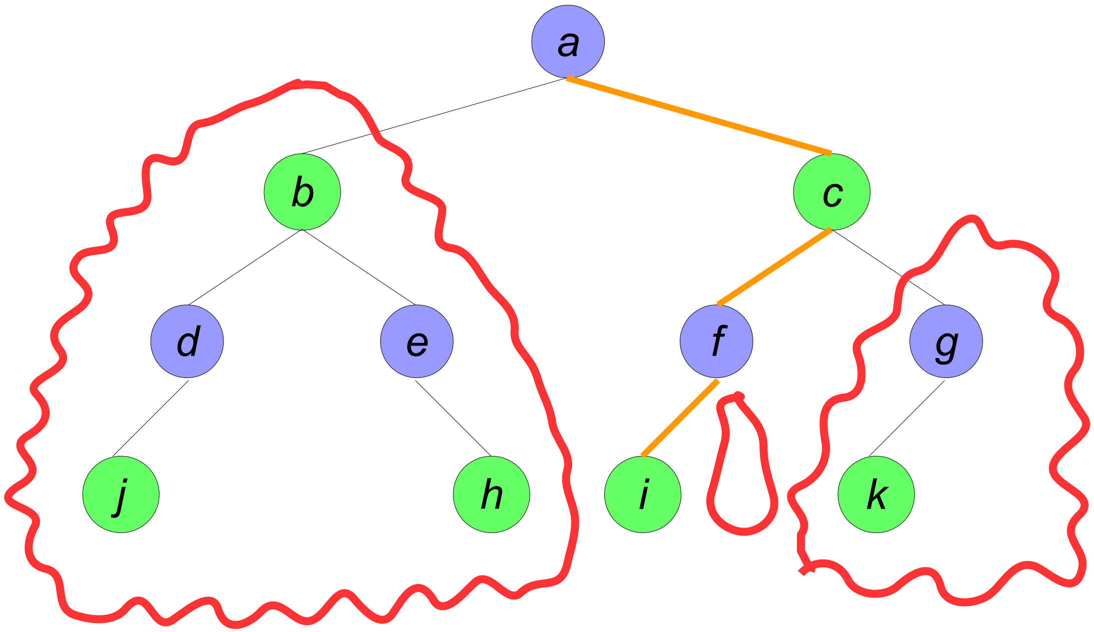
K-d trees

nearest neighbor search

If we decide to always observe also the other branch, then we should do it on every level...

... which will result in **inspecting entire tree**, and *O(n)* time complexity.



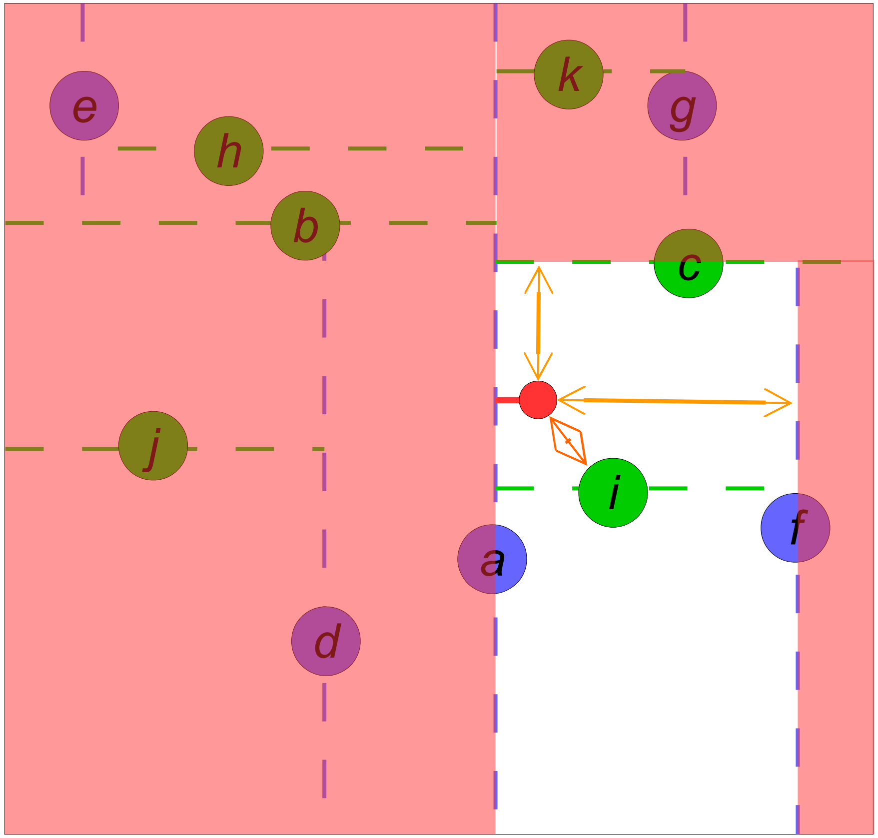


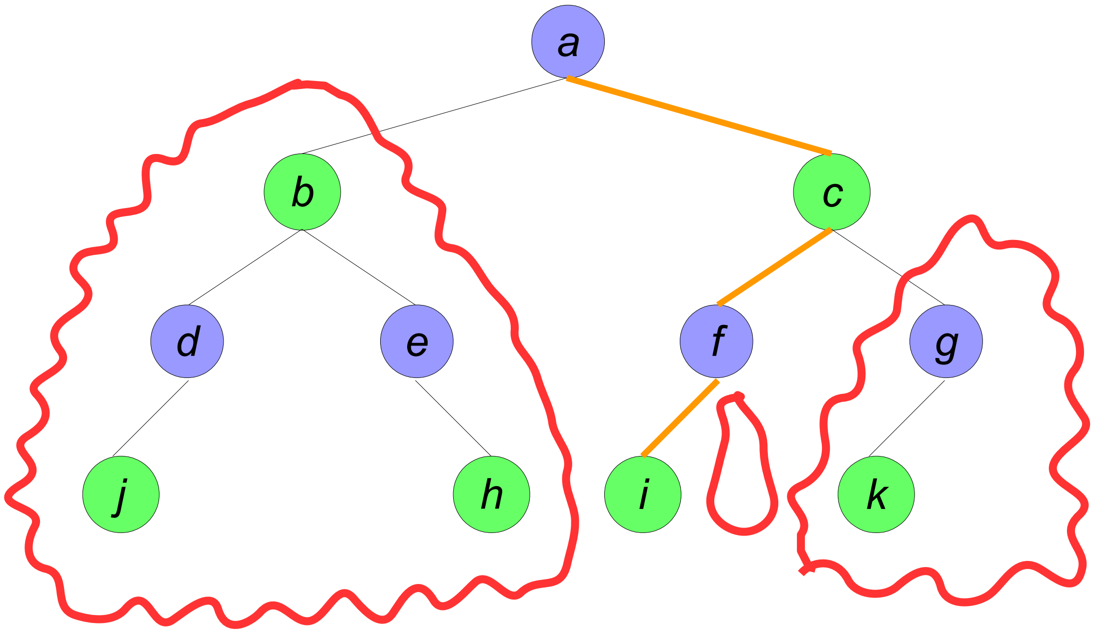
K-d trees

nearest neighbor search

So how should we do then... ?

Key idea is: during exit from recursion, inspect the other half too only if distance to it is less than current best minimum.



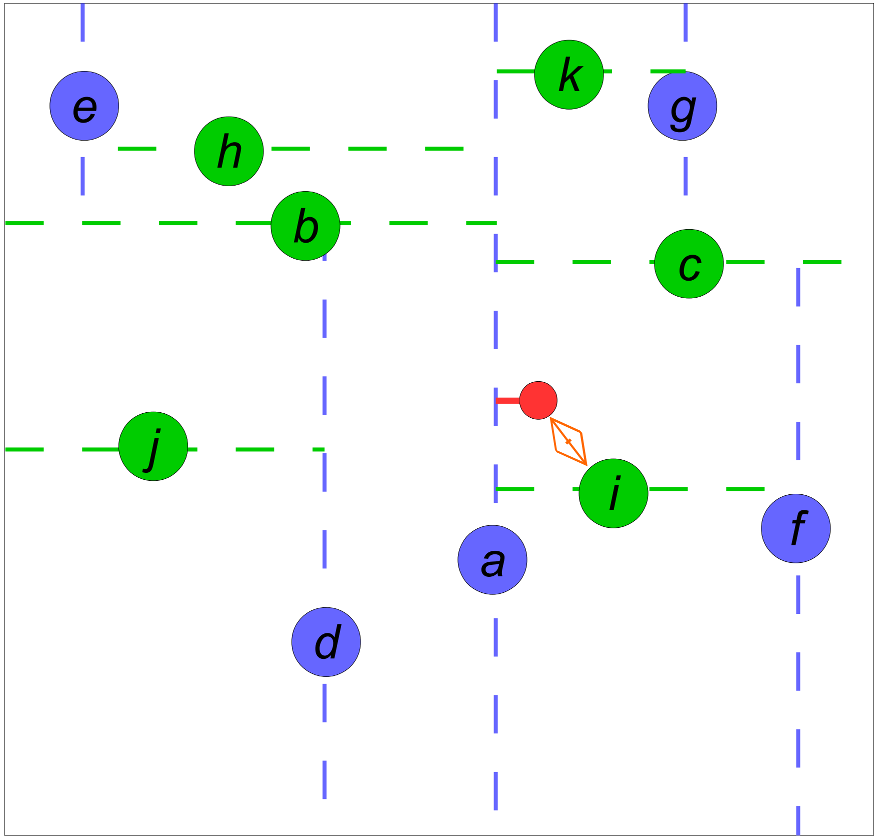


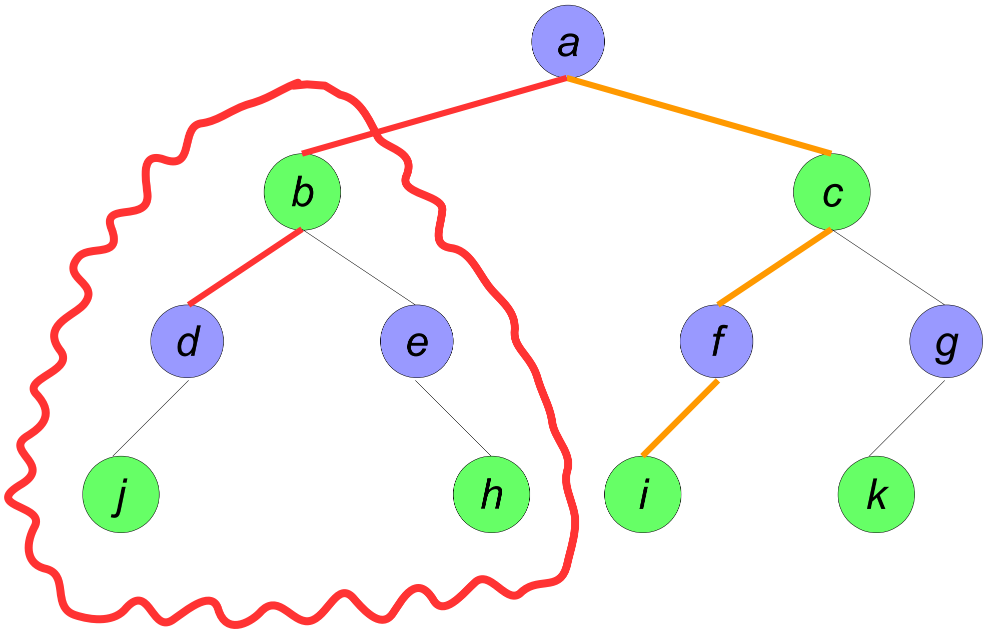
So, in this example, **only 1** extra branch need to be inspected.

K-d trees

nearest neighbor search

Once we do that too, here are the edges of the tree which will be inspected at the end.





K-d trees

nearest neighbor search

Now we can write pseudocode of the nearest neighbor search procedure:

```
procedure NN_Search( n: Node, q: Point, currMin: NumberRef,
                     currNN: NodeRef )
    // Update current minimum
    if dist(q,n) < currMin
        currMin := dist(q,n)
        currNN := n
    // Branch the search
    if q is on left (or bottom) half of n
        NN_Search( left{n}, q, currMin, currNN )
        // If need to check also the other branch
        if dist(q,plane{n}) < currMin
            NN_Search( right{n}, q, currMin, currNN )
    else
        NN_Search( right{n}, q, currMin, currNN )
        // If need to check also the other branch
        if dist(q,plane{n}) < currMin
            NN_Search( left{n}, q, currMin, currNN )
```

K-d trees

nearest neighbor search

We have designed the algorithm for NN search.

... but **how efficient** is it?

K-d trees

nearest neighbor search

Observation #1: During the NN search:

current best minimum can
only decrease,

while distance to the other half
generally increases,

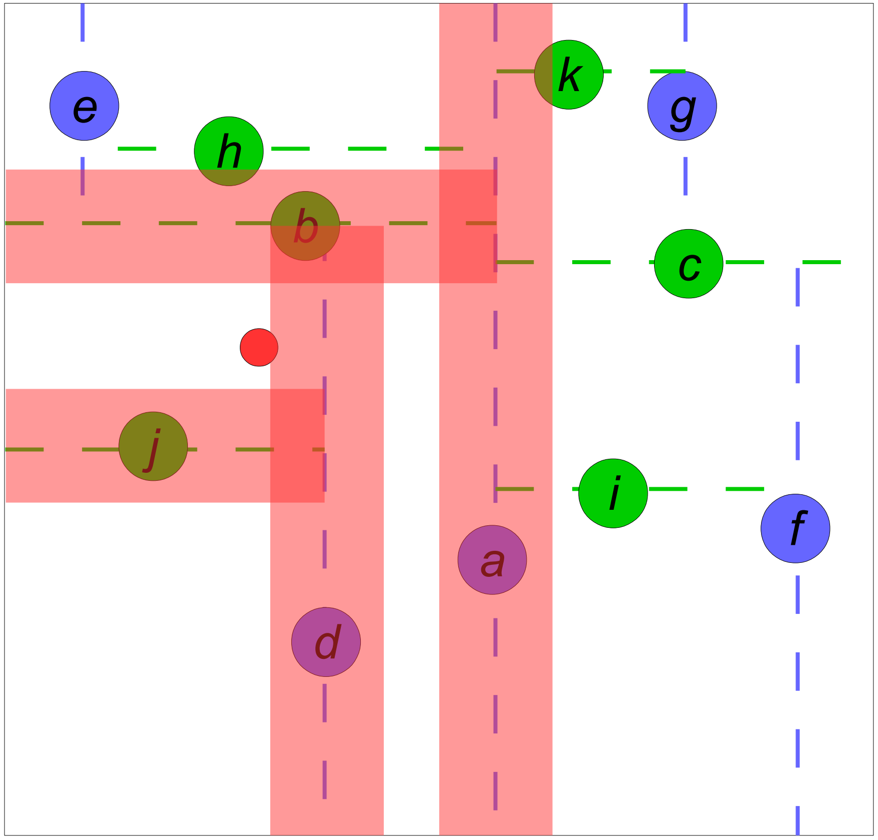
so it becomes **less and less probable**,
that we will need to observe
also the other half.

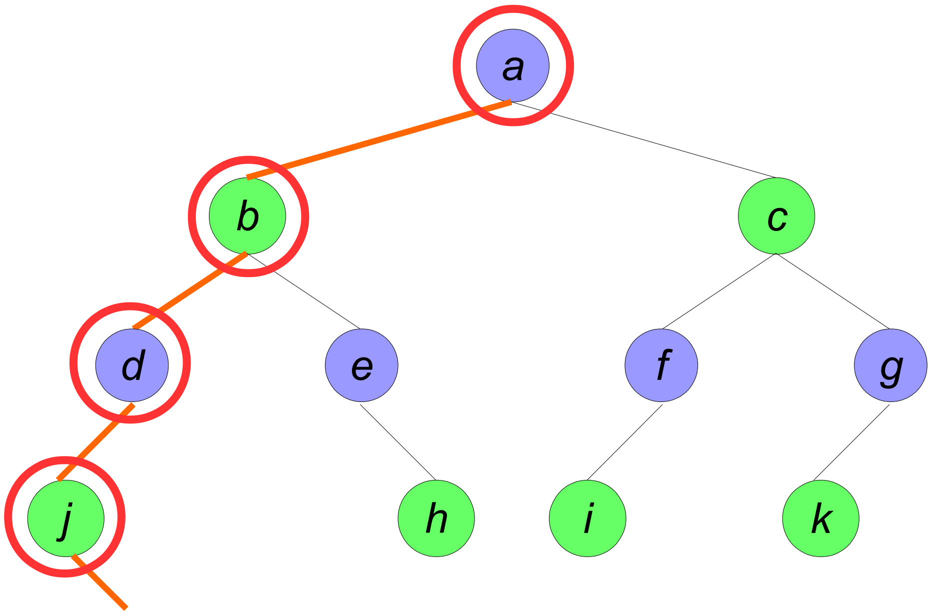
K-d trees

nearest neighbor search

Observation #2: Need to observe also the other half-plane will generally occur only for such query points, which are "*close enough*" to the other half-plane.

And that holds generally for nodes which are **deep enough**.





K-d trees

nearest neighbor search

So time complexity of NN search is only by **some constant** greater than tree height "*h*", so it gives:

O(h) = O(log n)

if the points are **distributed uniformly**.

The NN search algorithm can be easily extended to find k-nearest neighbors – ***k*-NN**, for the query point "*q*". In order to do that:

we keep "*k*" current best minimums, instead of just 1,

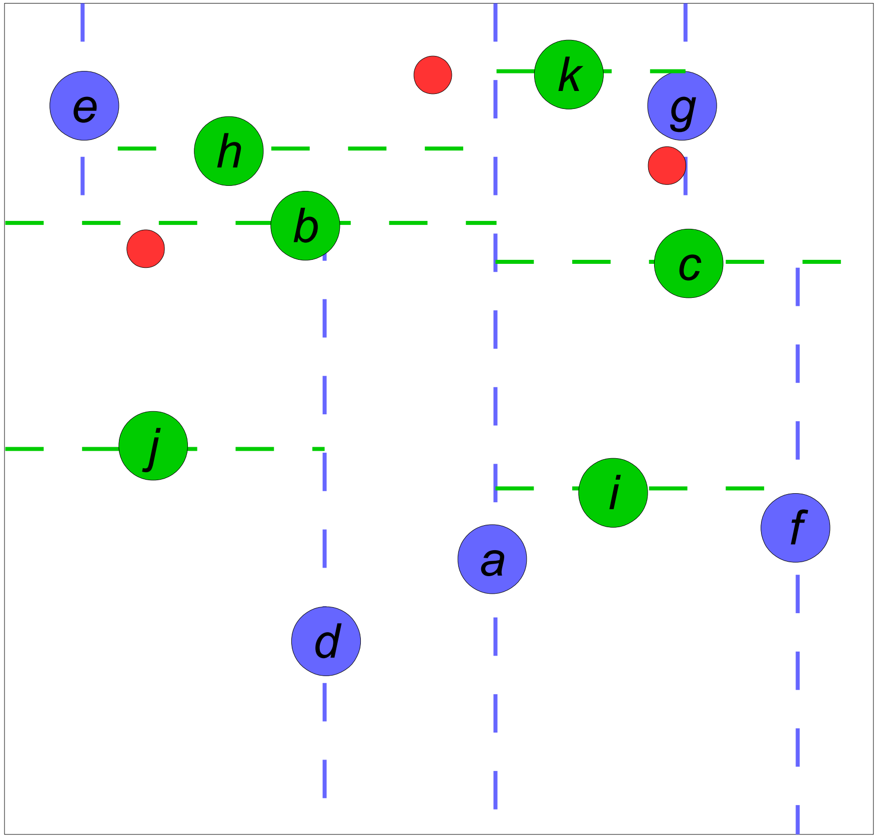
and we inspect also the other half-plane if distnace to it is less than any of the "*k*" current bests.

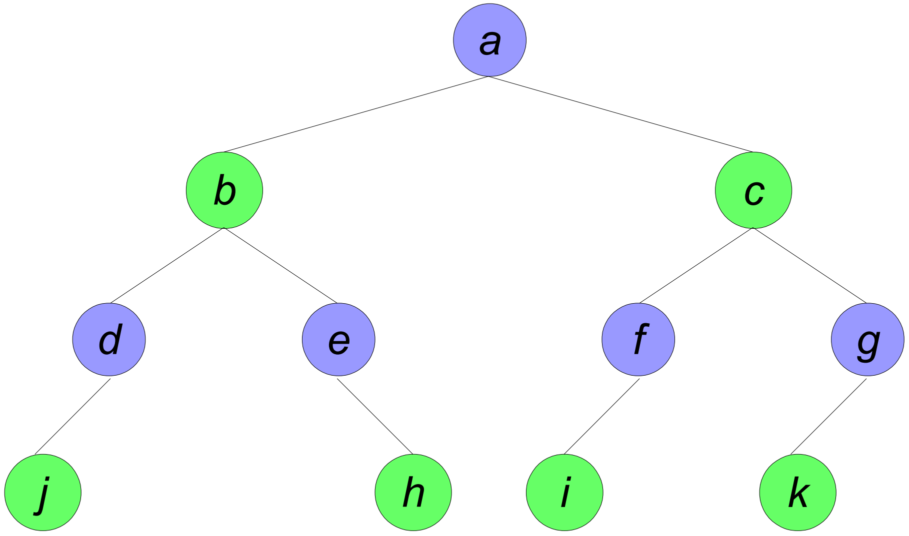
K-d trees

nearest neighbor search

Exercises:

For given K-d tree, perform NN search from the specified points:

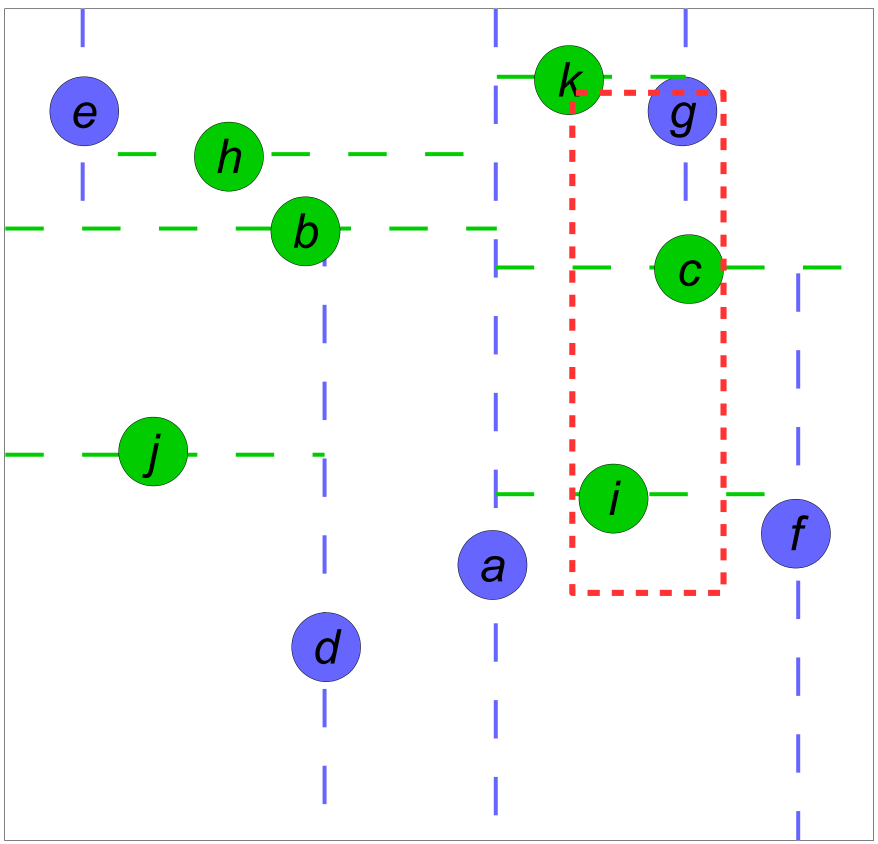


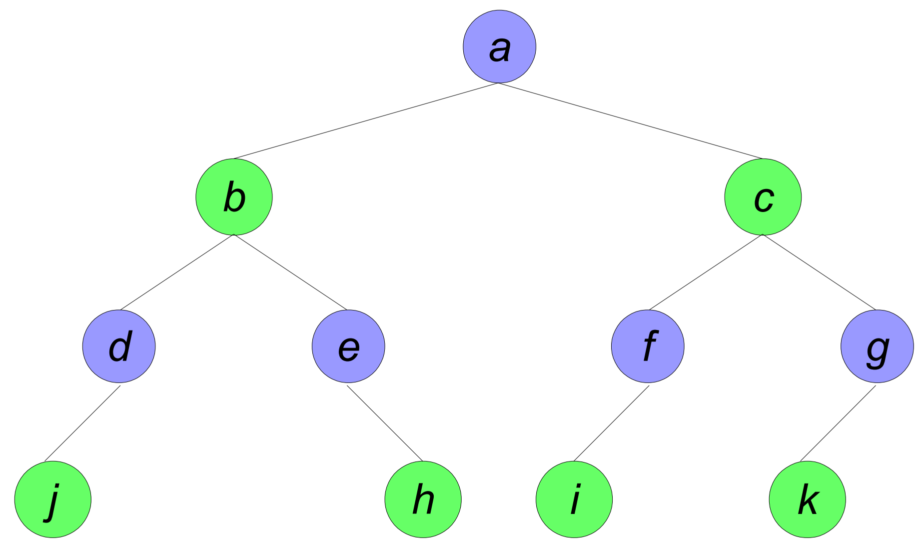


K-d trees

range search

For a range search we need to report all the points which are inside **query rectangle** "*q*".



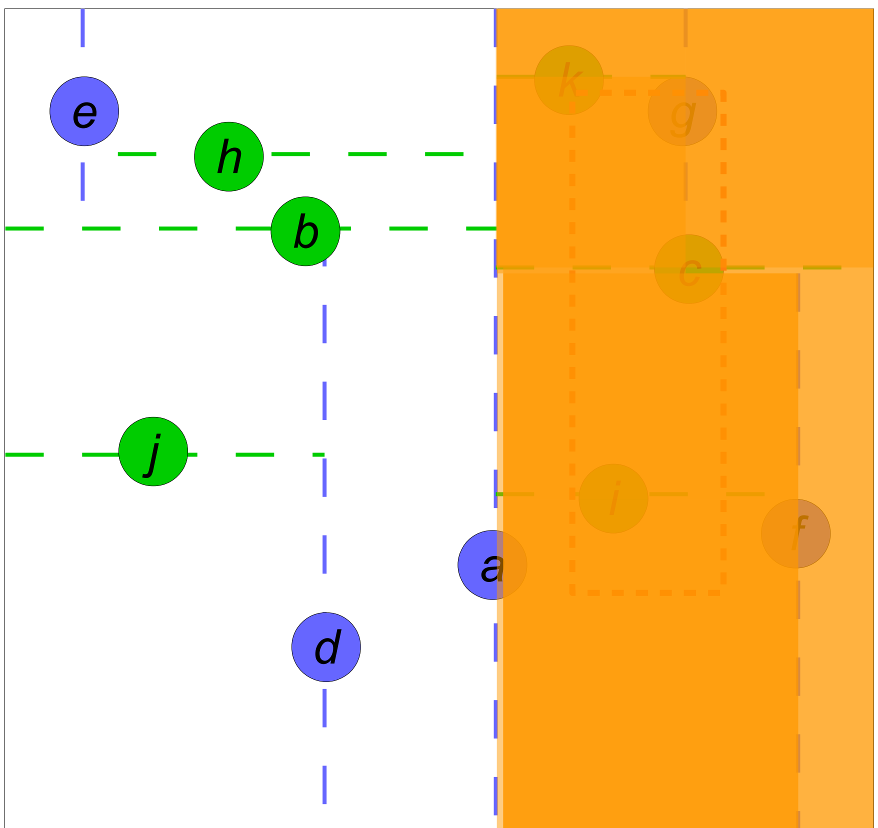


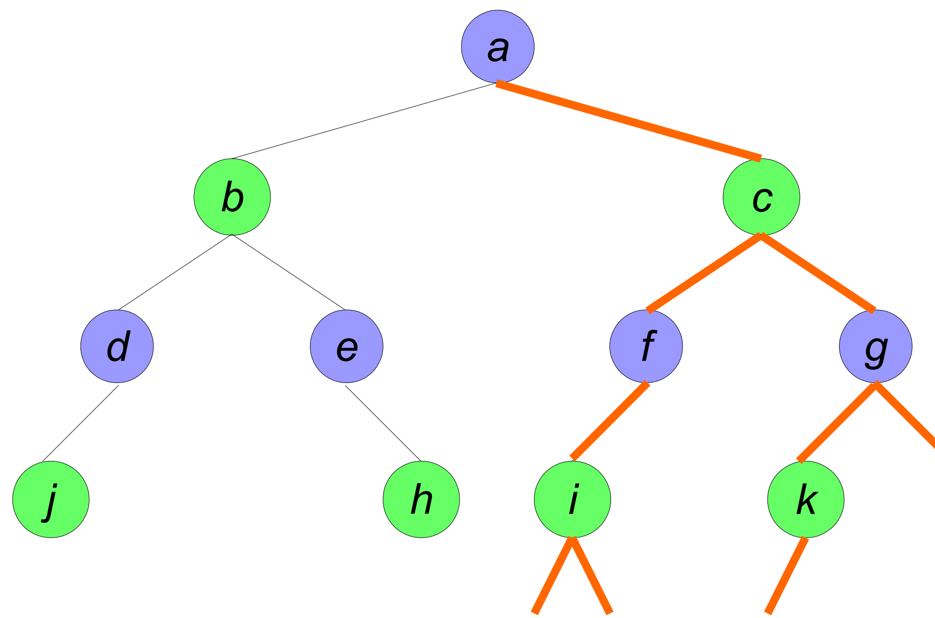
So, here we need to report "*g*", "*c*", "*i*".

K-d trees

range search

The algorithm is simple:
if "*q*" fits entirely in only half-plane, we continue from there,
otherwise we continue from both half-planes.



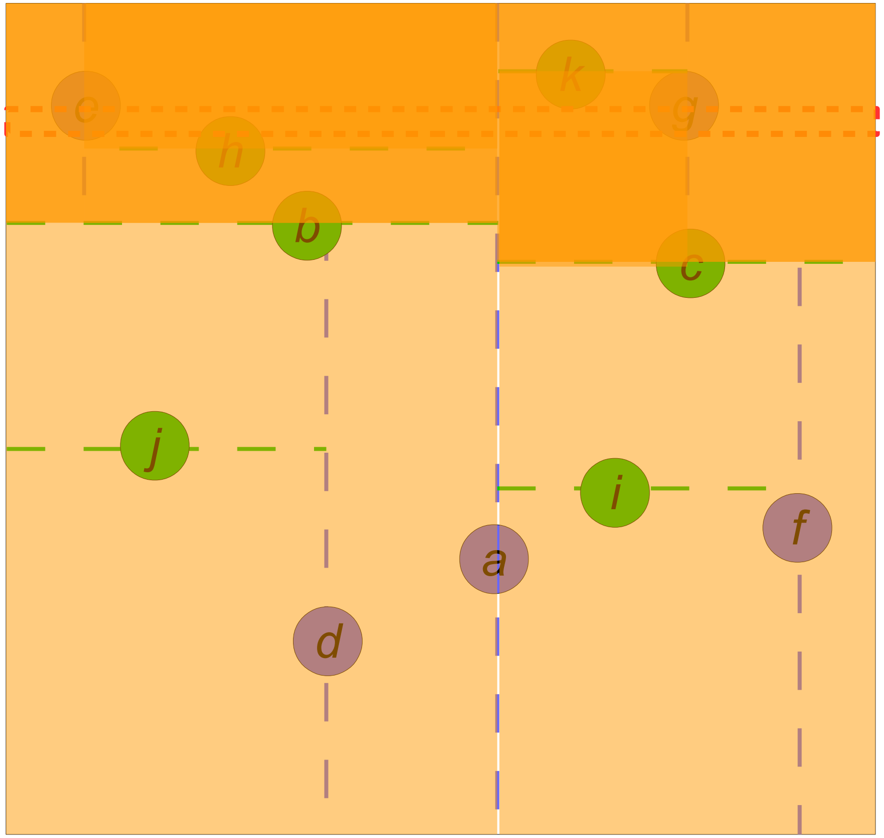


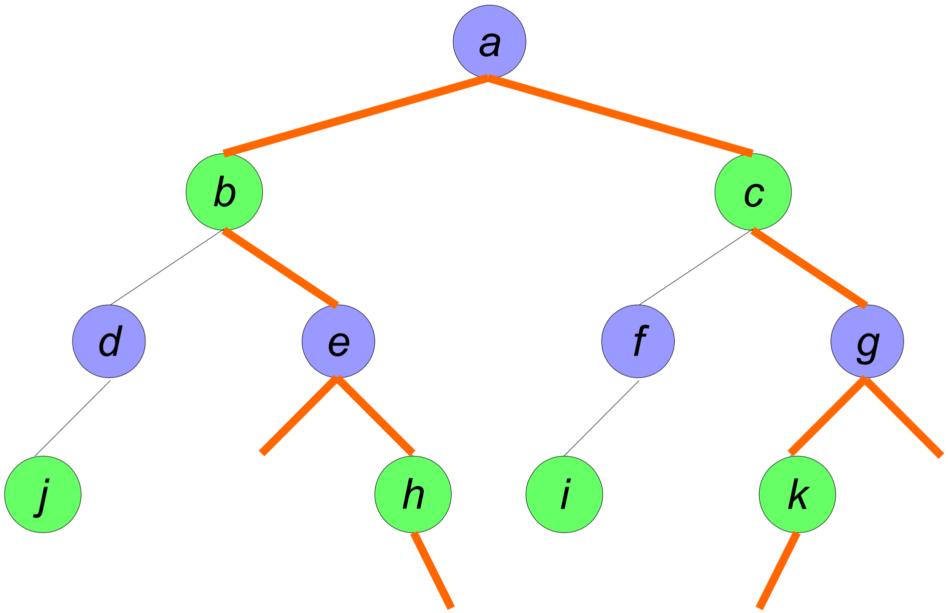
K-d trees

range search

But what is **time complexity** of such range search?

At first, let's consider the case when "*q*" is very **thin** and very **long**.





K-d trees
range search

As we can see:
the search splits on every **odd** level,
and the search doesn't split on every **even** level.

So we have:
*f(n) = 2*f(n/2)* for odd levels,
f(n) = f(n/2) for even levels,
or combining it:
*f(n) = 2*f(n/4)*

Continuing recursively, we obtain:

$$f(n)=2f\left(\frac{n}{4}\right)=4f\left(\frac{n}{16}\right)=2^{\frac{k}{2}}f\left(\frac{n}{2^k}\right)=2^{\frac{\log_2 n}{2}}=2^{\log_2 n*\frac{1}{2}}=\sqrt{2^{\log_2 n}}=\sqrt{n}$$

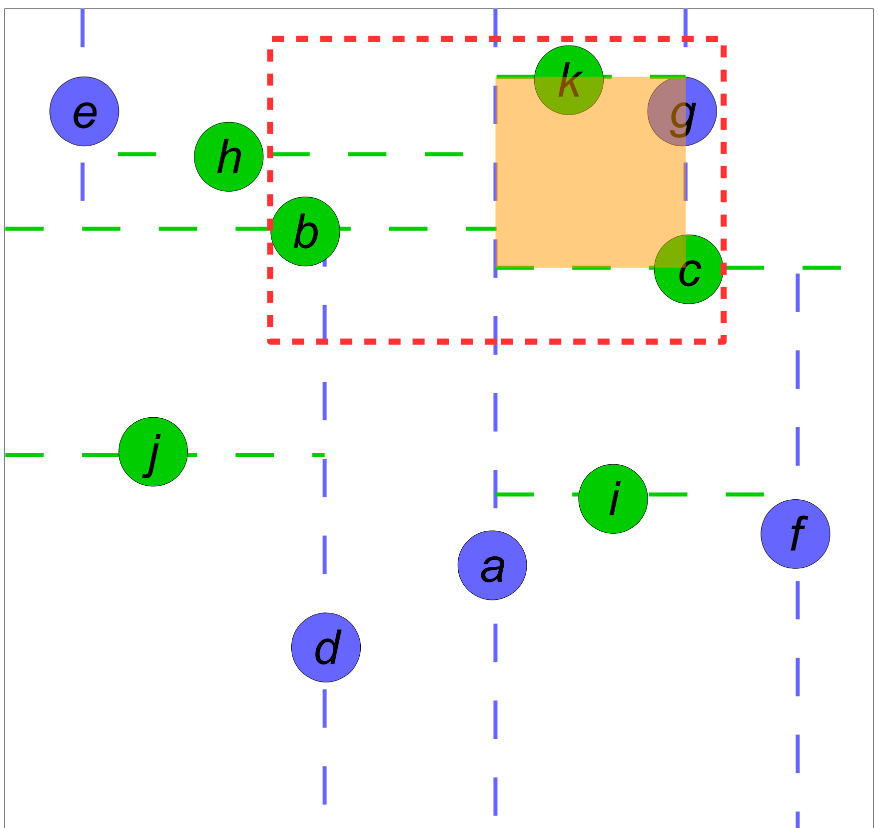
So time complexity for such thin rectangle is ***O(√n)***.

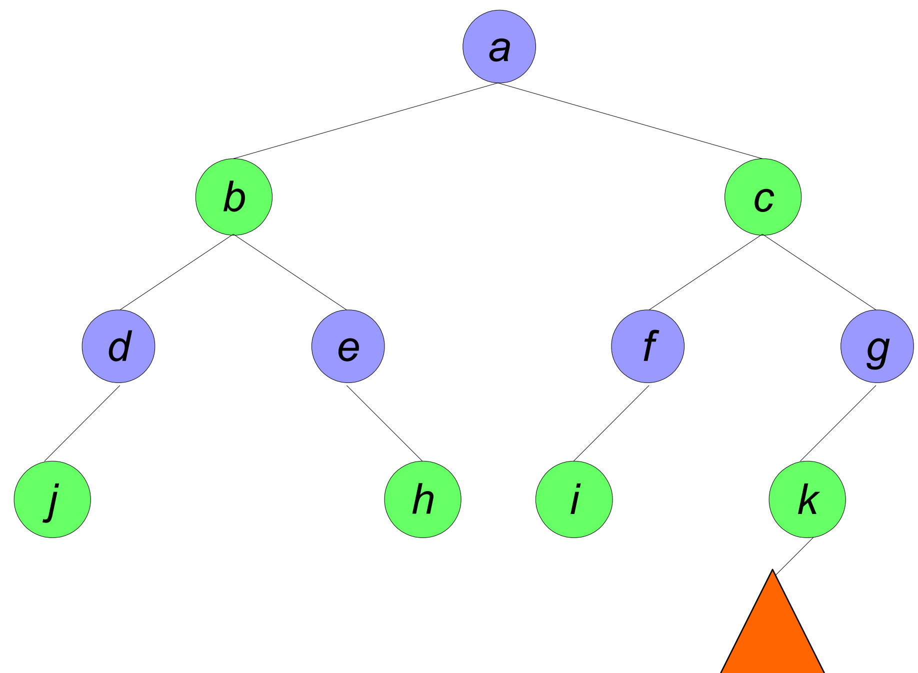
K-d trees

range search

But what about a **normal** query rectangle?

Let's note that for some "*q*" there can be such subtrees, which will be **reported completely**. That is when their area lies inside "*q*".





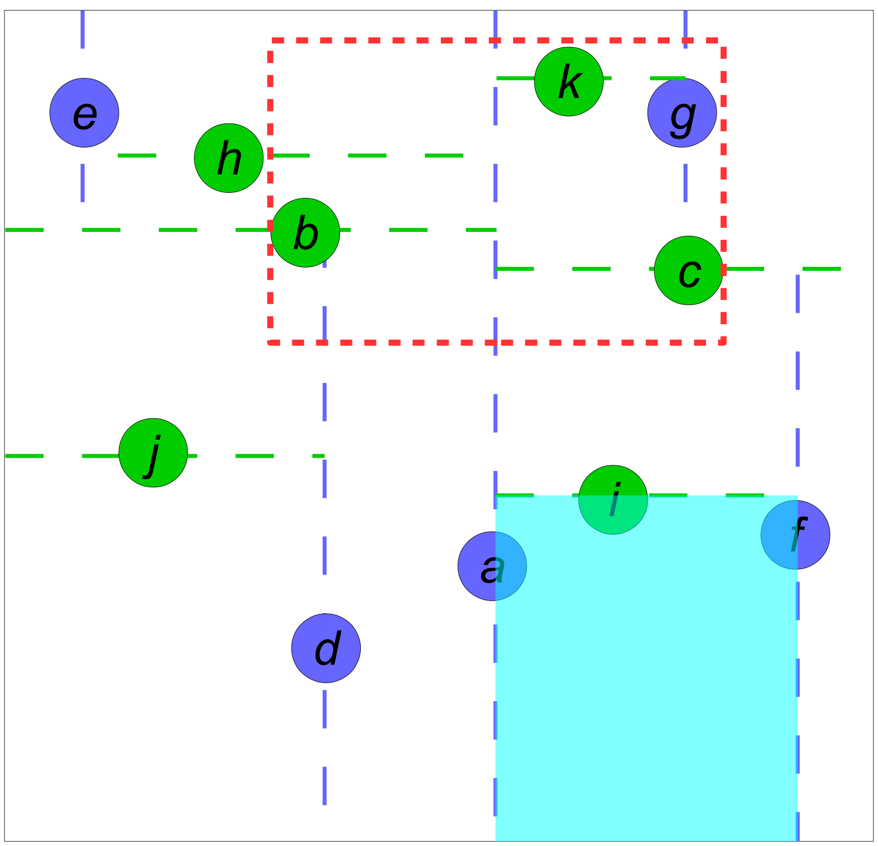
Here, subtree left to "*k*"
will be reported completely.

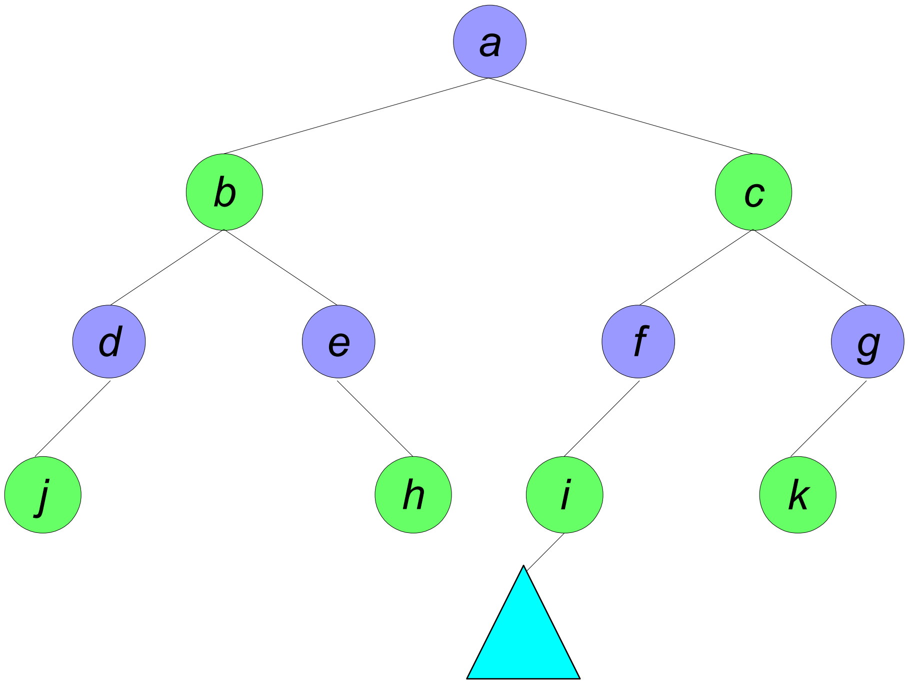
K-d trees

range search

From the other side, some other subtrees will **not be touched** at all.

Here, subtree left to "*i*" will not be touched.



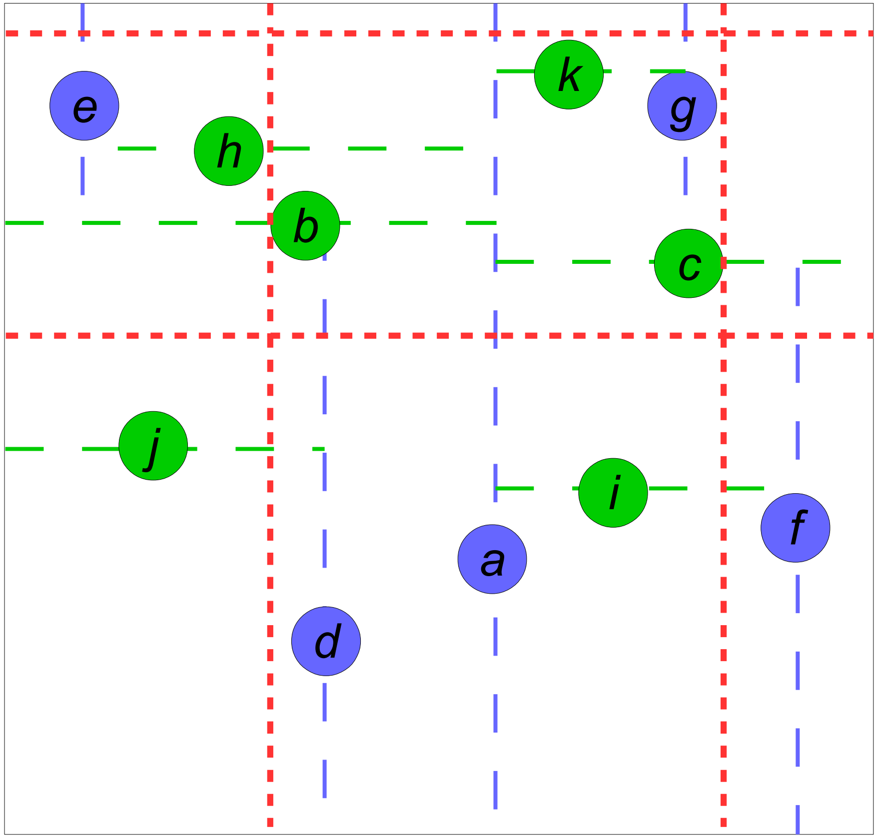


K-d trees

range search

It remains to analyze the subtrees, which will be **reported partially**.

And number of such subtrees is not greater than of those, intersecting the following 4 thin rectangles.



So we have:

Completely reported subtrees:
O(k),

Partially reported subtrees:
$O(4*\sqrt{n}) = O(\sqrt{n})$,

Not touched subtrees:
no time spent.

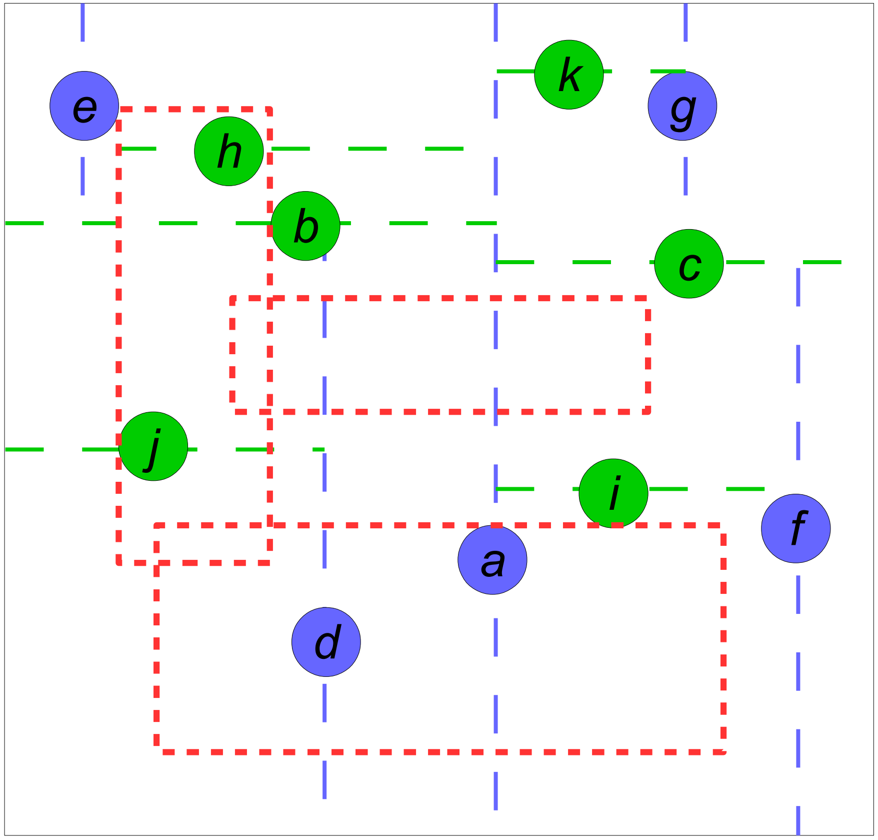
Which in sum gives:
$\boldsymbol{O(\sqrt{n} + k)}$, where "*k*" is number of reported points.

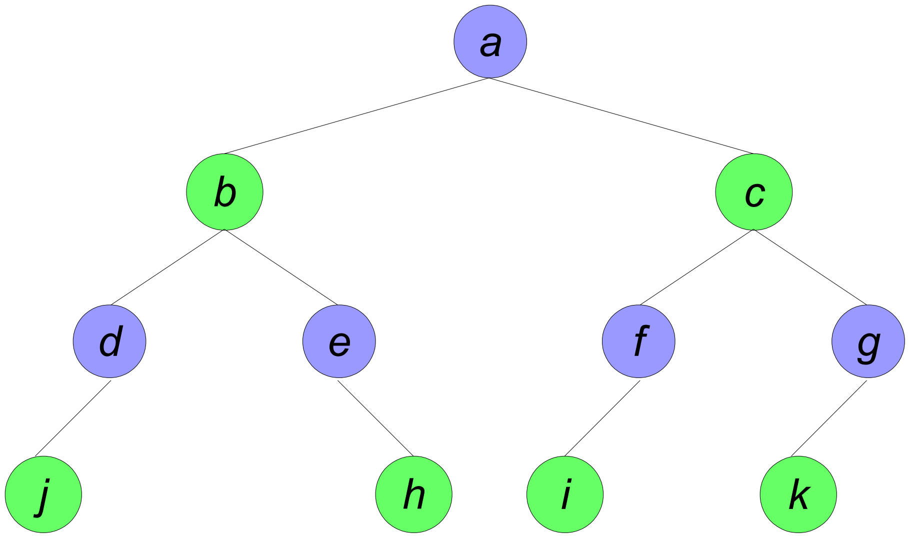
K-d trees

range search

Exercises:

Perform range searches with the following query rectangles:

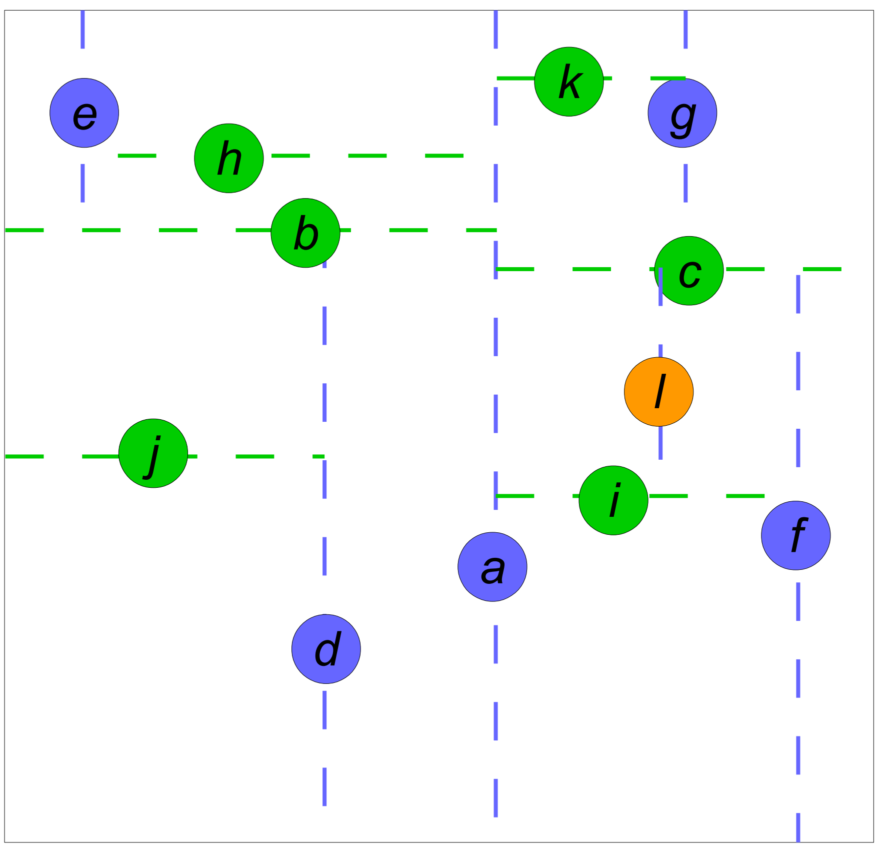


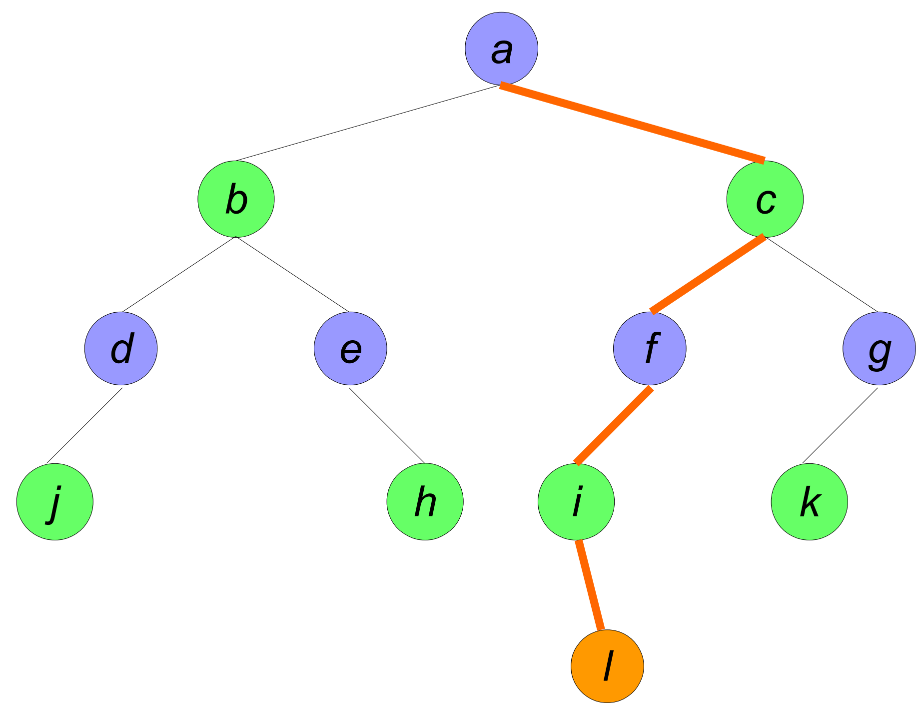


K-d trees

adding points

Adding new point is quite similar to checking for presence:



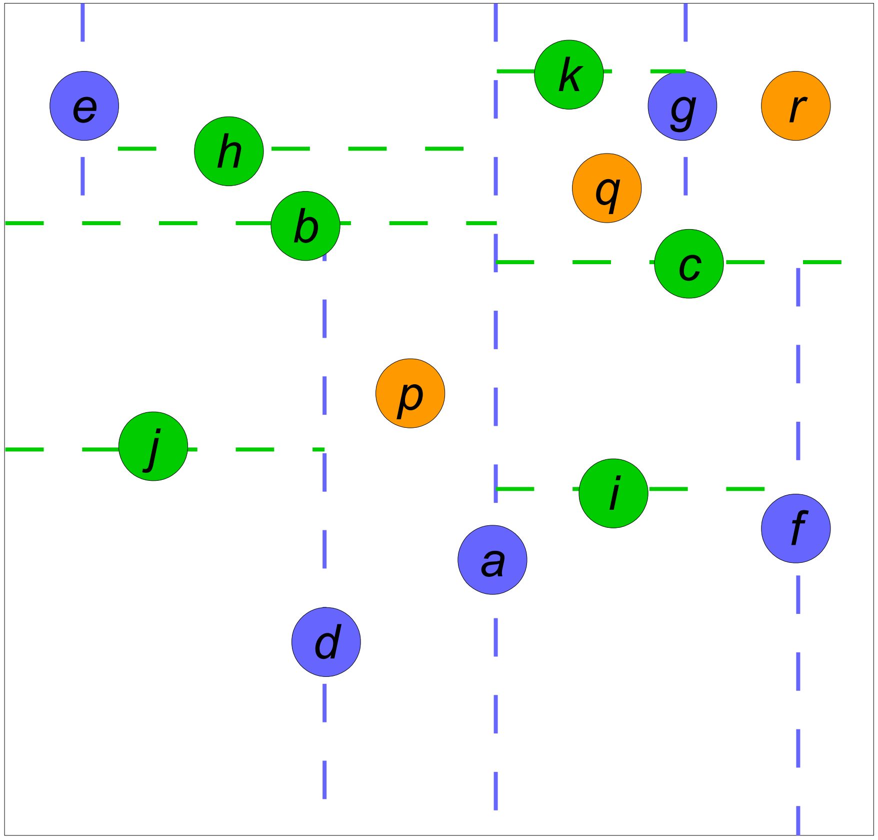


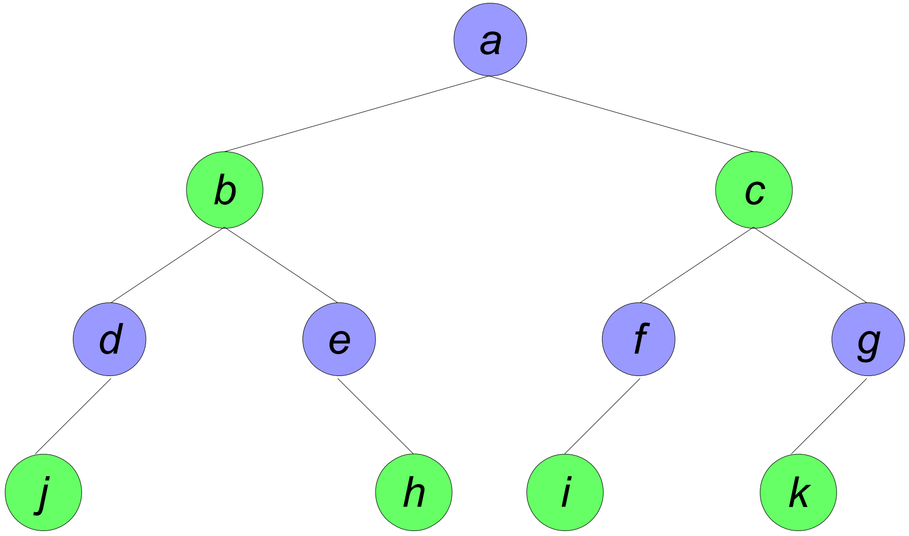
K-d trees

adding points

Exercises:

Insert the following points into the K-d tree:

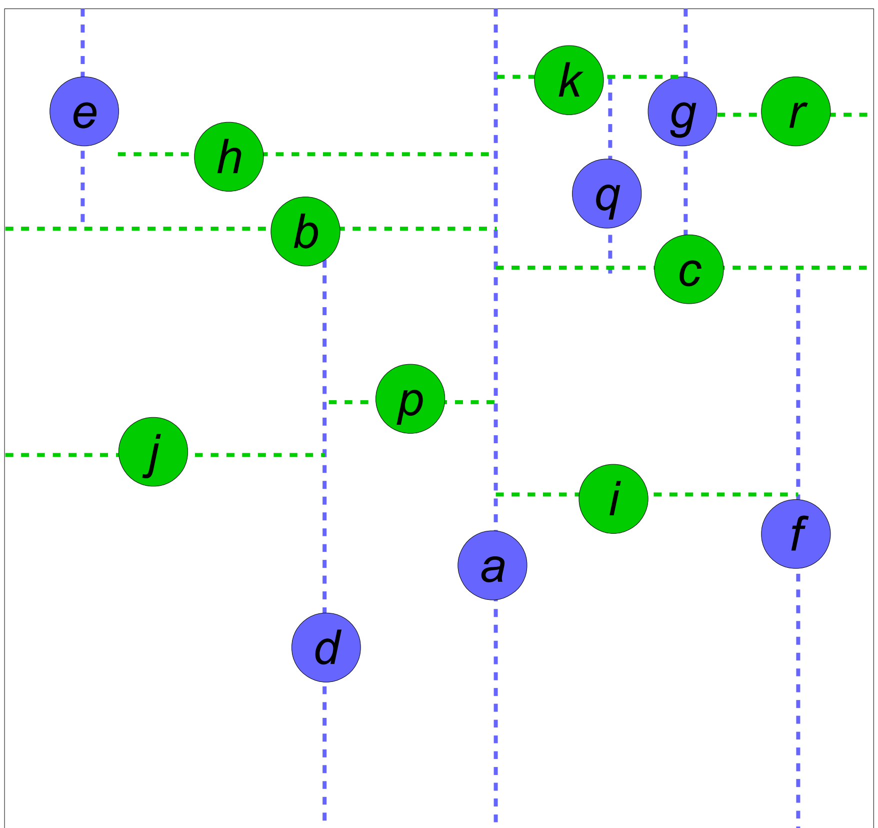


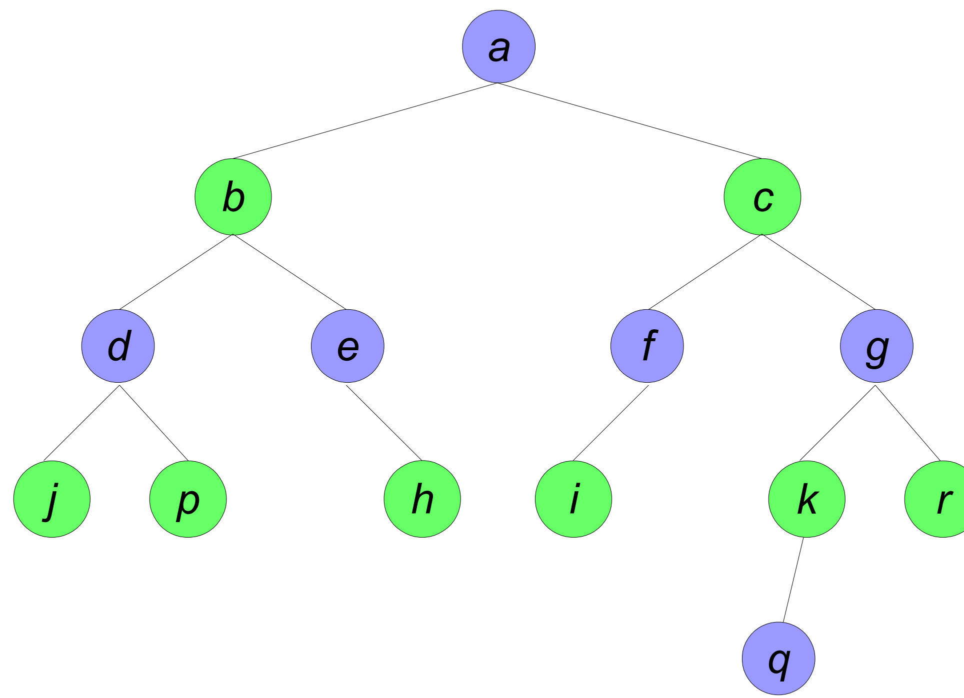


K-d trees

removing points

If node is a leaf, removal is trivial.

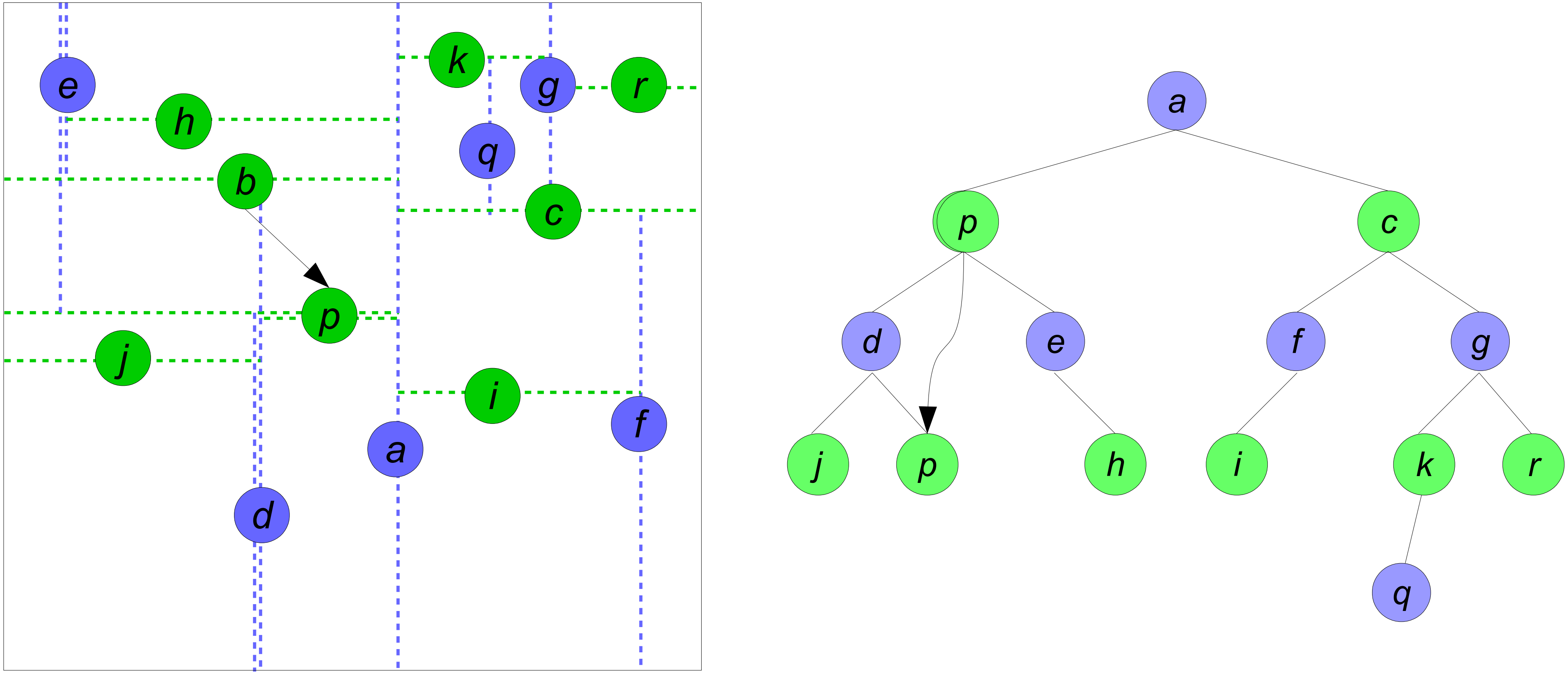




K-d trees

removing points

Other scenario: if it has descendants, it must be replaced by the "nearest" one from them (i.e. nearest one from it's region).

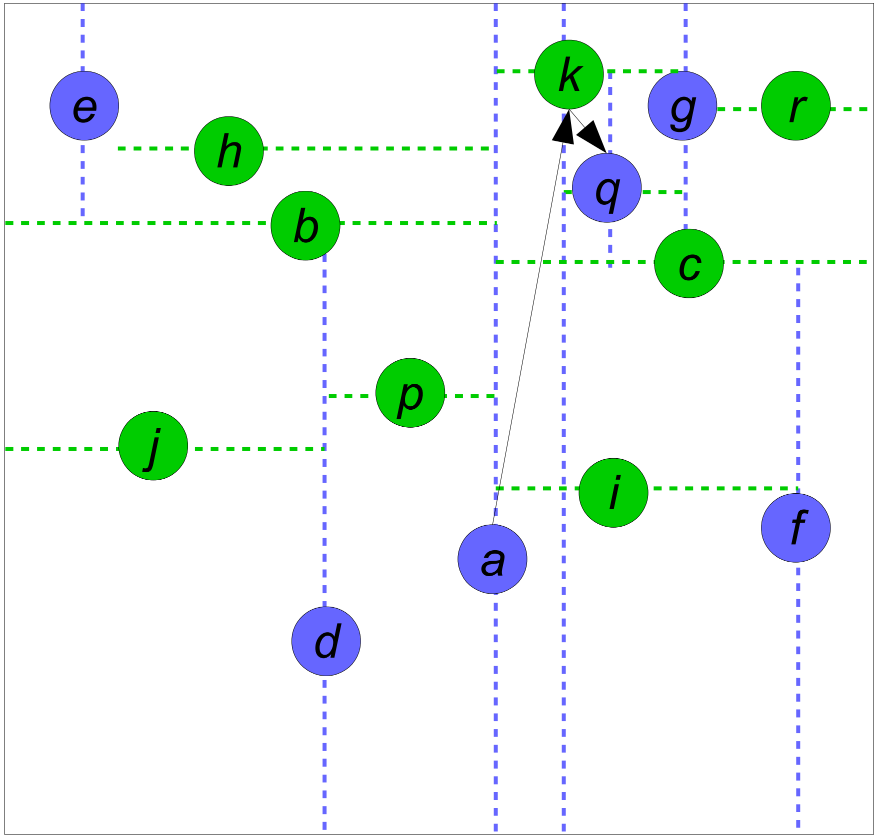


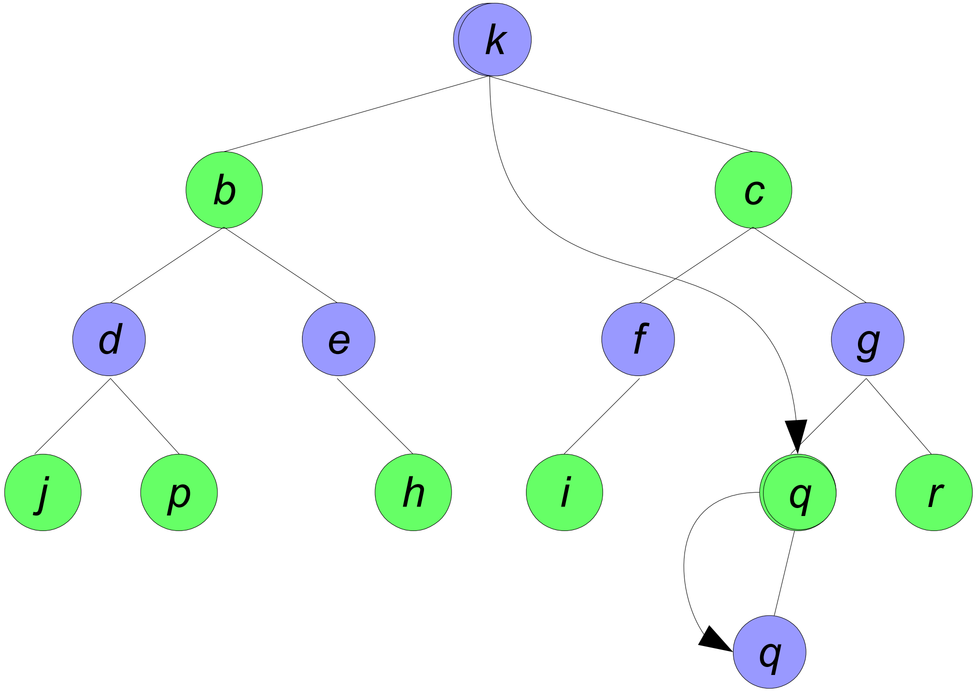
K-d trees

removing points

But it can be also that the "nearest" one has descendants too.

In that case it must be removed from the tree recursively.



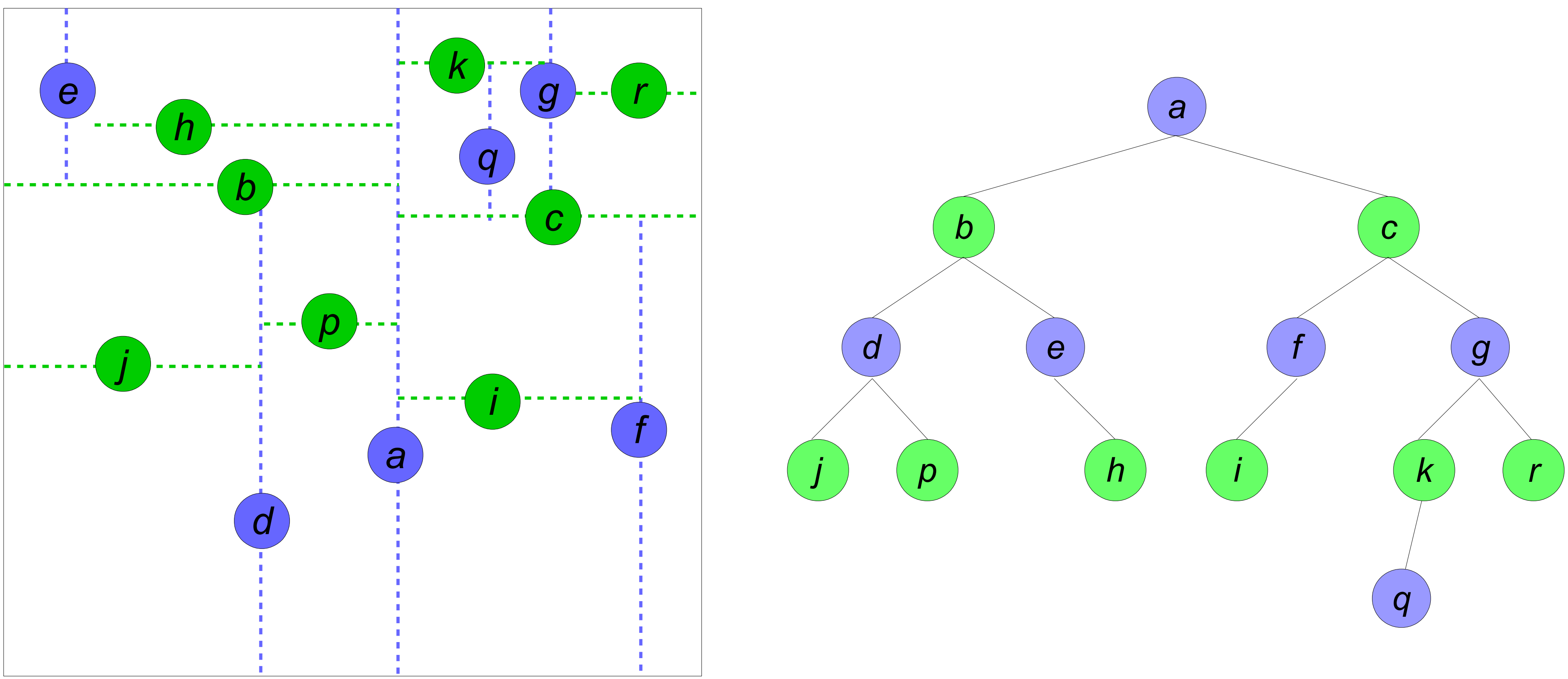


K-d trees

removing points

Exercise **1**:

Remove the following point from the K-d tree:

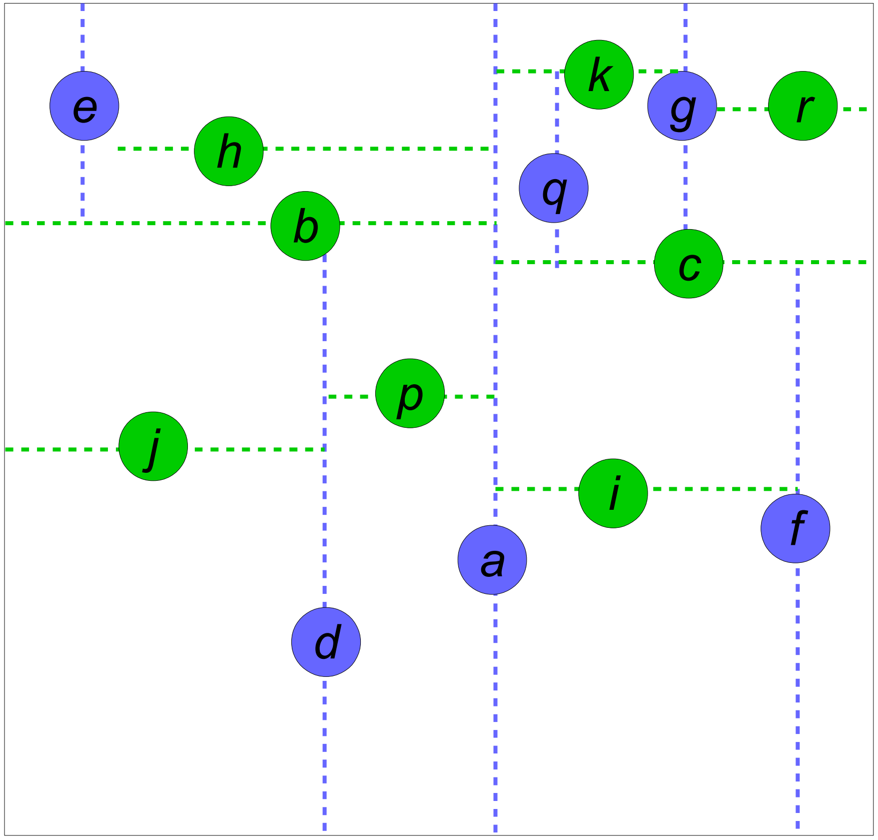


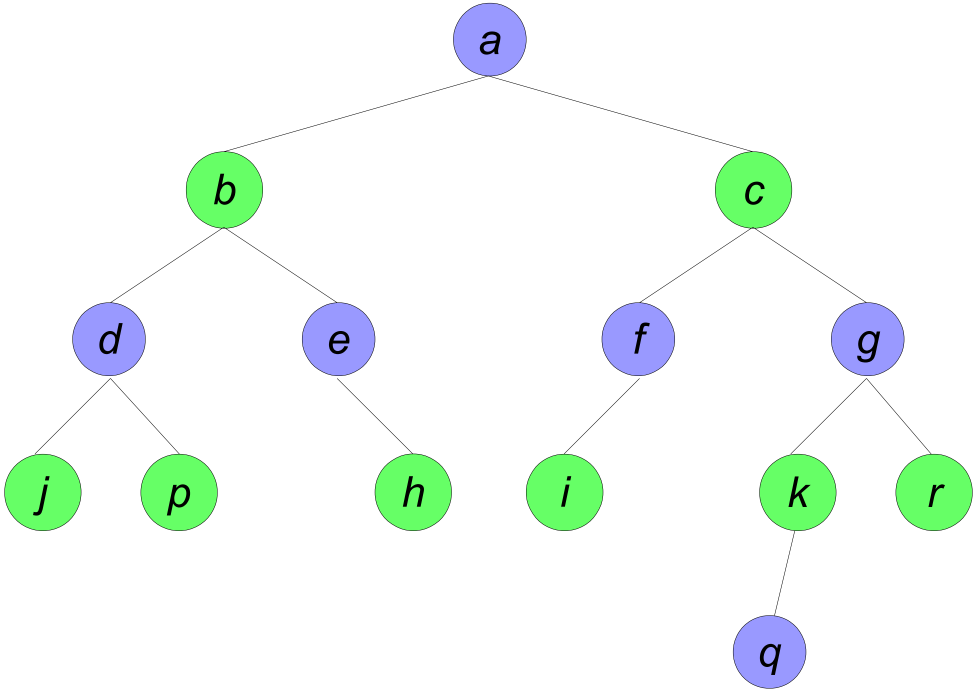
K-d trees

removing points

Exercise **2**:

Remove the following point from the K-d tree:





K-d trees

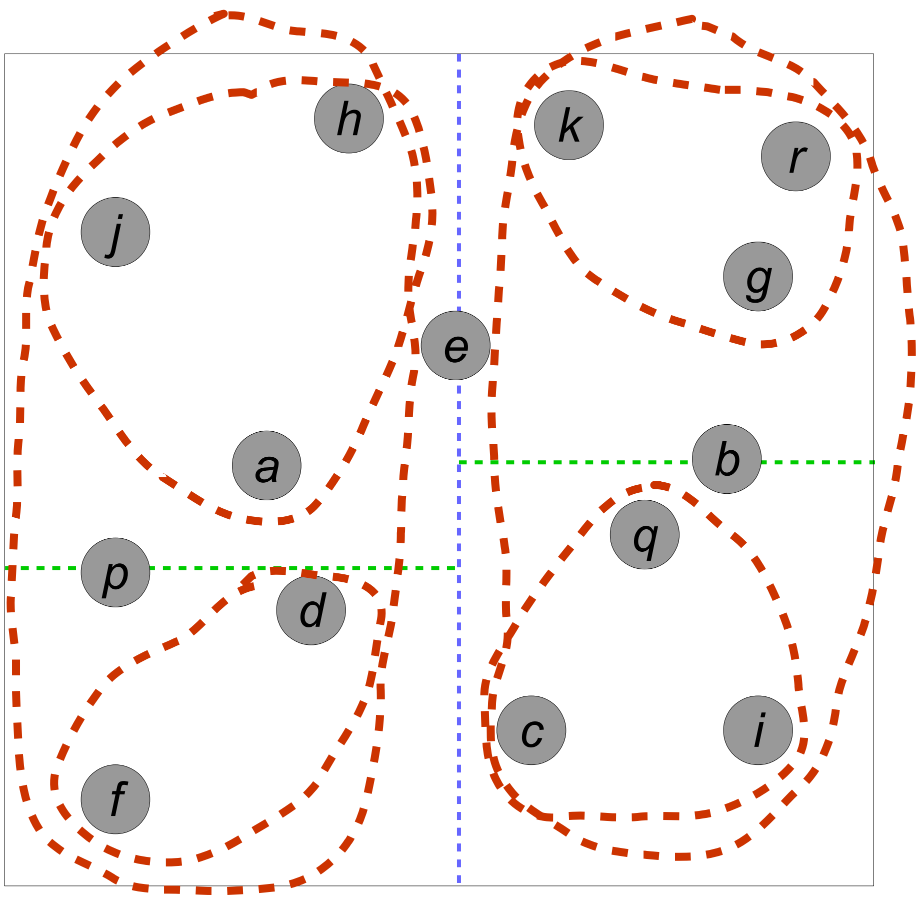
batch construction (approach 1)

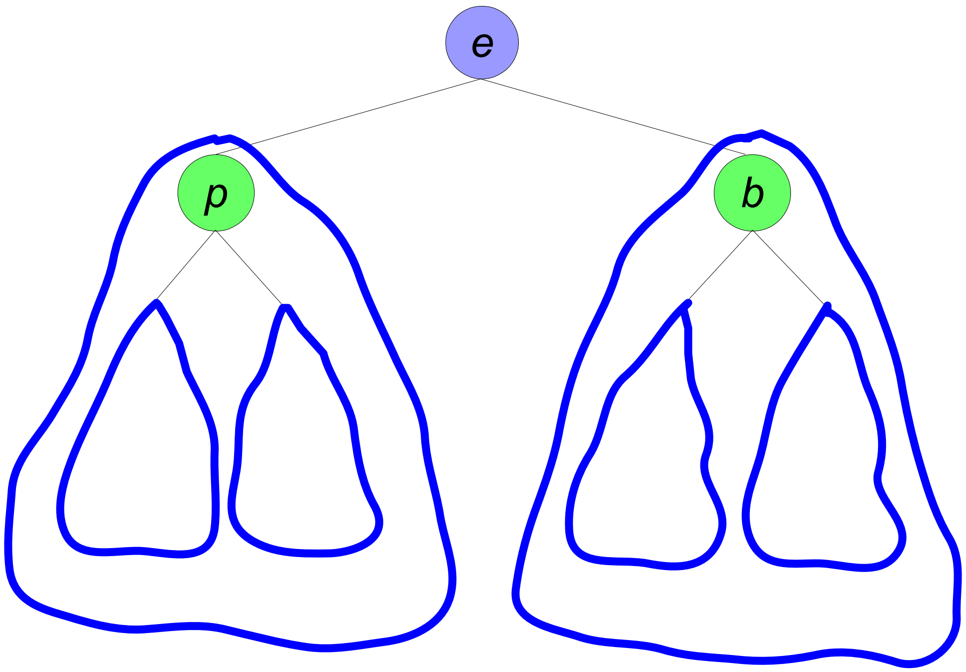
The most straightforward way:

1) Find the median,

2) Partition by it,

3) Recursively construct subtrees.





K-d trees

batch construction (approach 1)

So the question is: **how to find median points?**

method	*result*	*time*
Median of a sample	not always a good median	*O(1)*
Complete sort	best result	*O(N*logN)*
QuickSort-based median	best result	expected *O(N)*

K-d trees

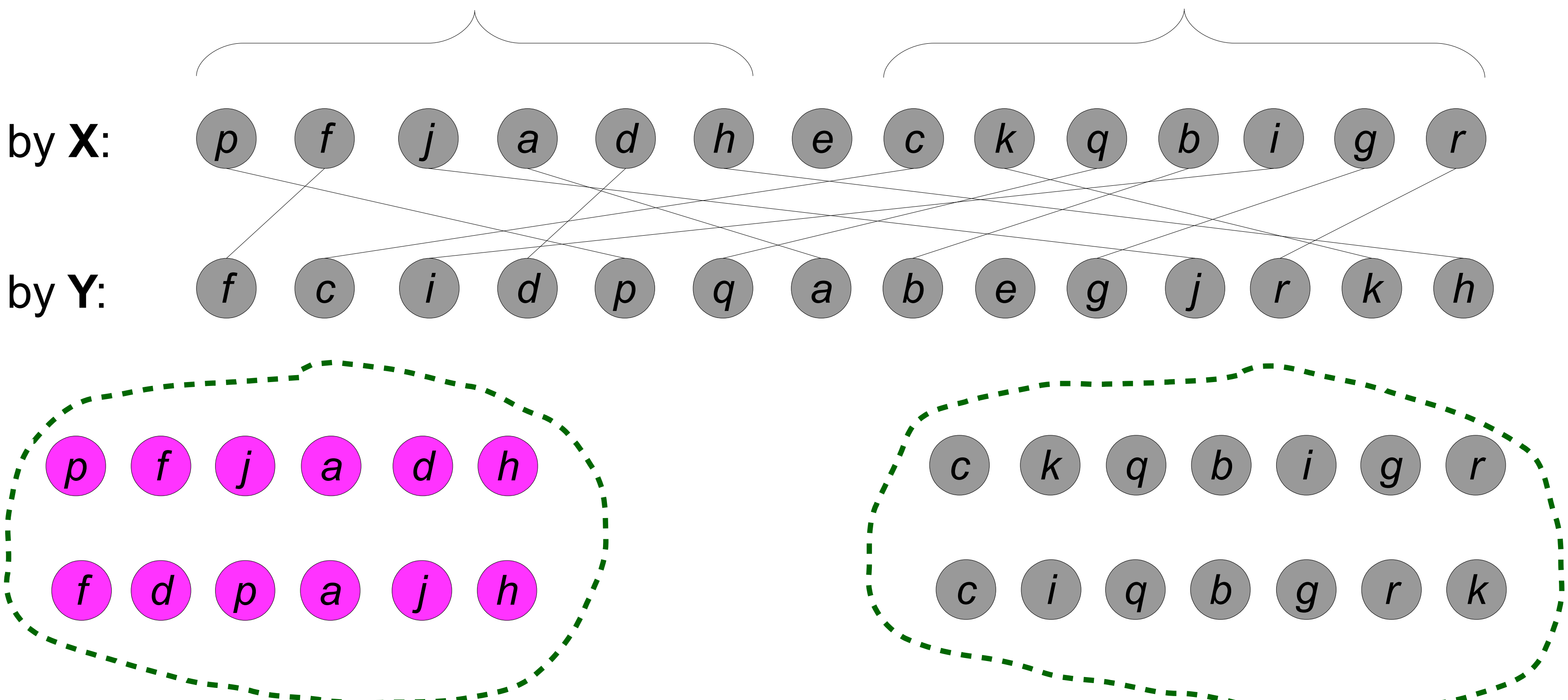
batch construction (approach 2)

Sort the points **by both X and Y** coordinates. So at each step:

find the median & divide into 2 parts,

rewrite 1st items & 2nd items from the other sequence,

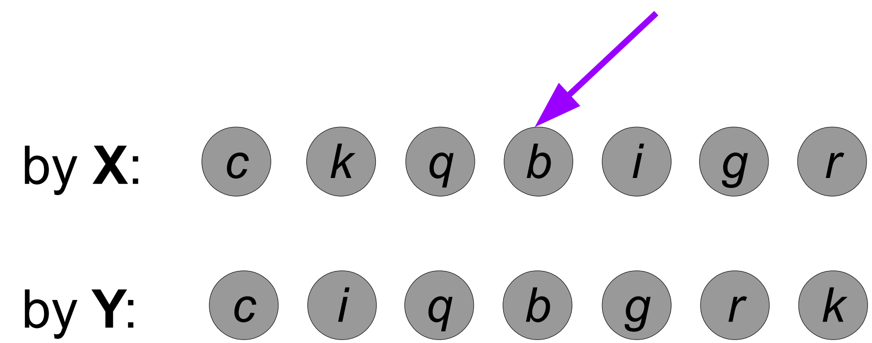
we again have necessary sorted lists: continue recursively.



K-d trees

batch construction (approach 2)

Having that said, on each step we have both sorted lists.



Picking median is trivial.

Time to construct 2 pair of smaller lists, from lists of size N is:

  O(N).

Overall time complexity of batch construction:

  *O(N*logN)* for sorting by both X and Y.

  N + 2(N/2) + 4*(N/4) + ... + N*1 = O(N*logN)* for construction itself.

Summary: ***O(N*logN)***.

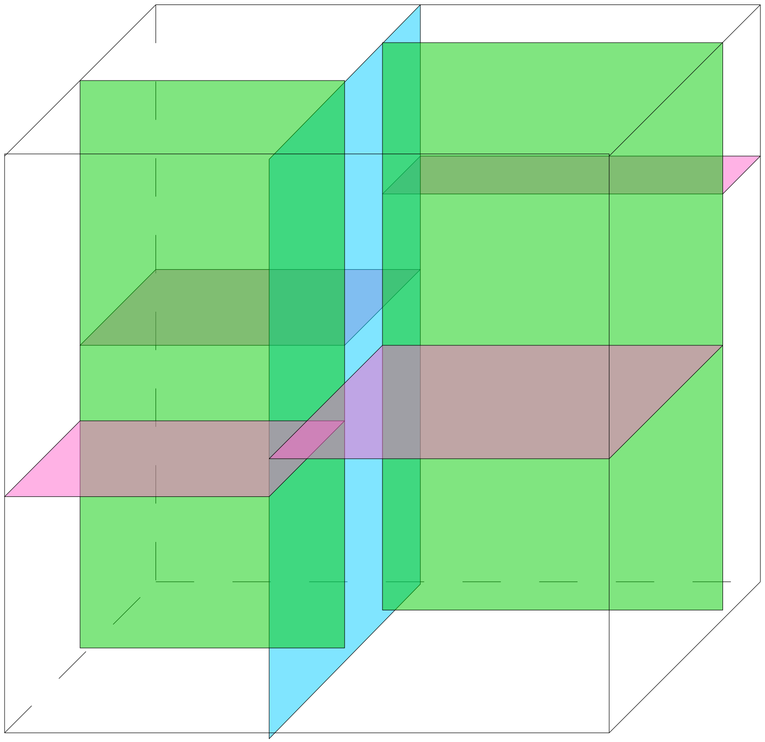
K-d trees

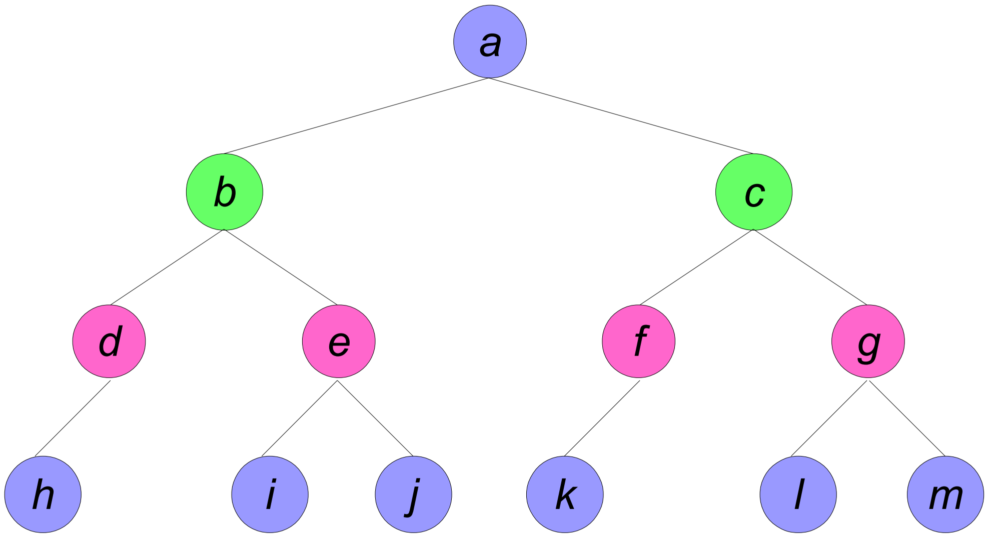
higher dimensions

K-d trees can be easily extended to **3D** and higher dimensions.

All what we should do is just to perform splits by multiple axes:

XYZXYZXYZ...





Generally, all algorithms of the K-d tree **remain unchainged**.

K-d trees

higher dimensions

Time complexity of range search will be:

$$O\left(D * n^{1-\frac{1}{D}} + k\right)$$ *where D is number of dimensions.*

We can node that together with increase of *D*, time complexity of range search becomes more and more **closer to linear**.

In some sense, this holds also for NN search. For example, if *n~D* then NN search runs almost with same performance, as linear search.

If we want K-d tree to behave efficiently, we must ensure that ***n<<D***.

K-d trees

comparison with other data structures

	Quadtree	K-d tree	Range tree
NN Search:		$O(\log N)$	
Range search:		$O(DN^{1-1/D} + k)$	$O(\log^D N + k)$
Construction:	$O(DN*\log N)$	$O(DN*\log N)$	$O(N*\log^{D-1} N)$
Insert:	$O(\log N)$	$O(\log N)$	$O(\log^D N)$
Remove:	$O(\log N)$	$O(\log N)$	$O(\log^D N)$
Space:	$O(2^D*N)$	$O(N)$	$O(N*\log^{D-1} N)$
Higher dimensions:	bad scaling	good scaling	good scaling