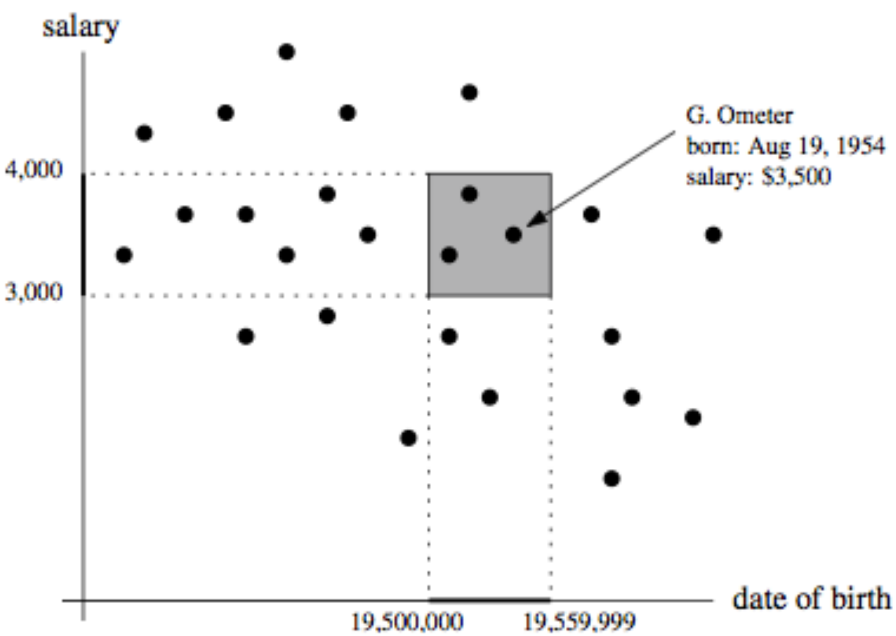
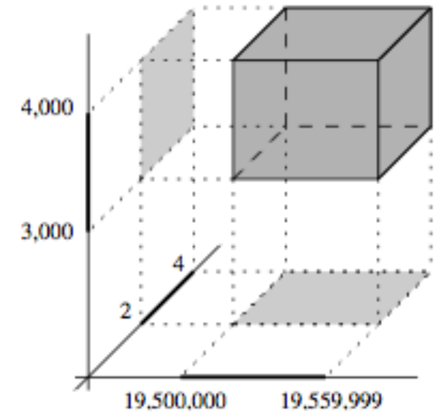
Querying a database



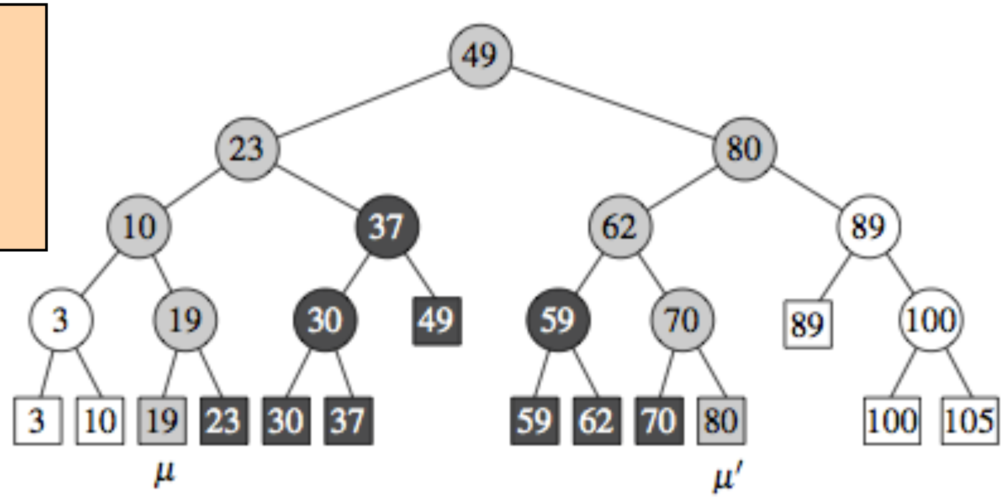


Orthogonal searching
5 : Linear Programming
pages 95-116

1D Range Searching

We can solve the 1-dimensional range searching problem efficiently using a well-known data structure:

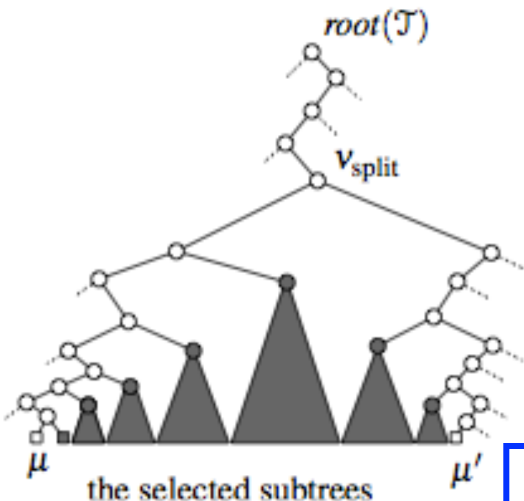
a balanced binary search tree !



When we search with the interval [18 : 77] in the tree, we have to report all the points stored in the dark grey leaves plus the leaf of item 19.

Let $P := \{p_1, p_2, \ldots, p_n\}$ be the given set of points on the real line.

Finding the split node ?



We first search for the split node where the paths to *x* and *x'* split.

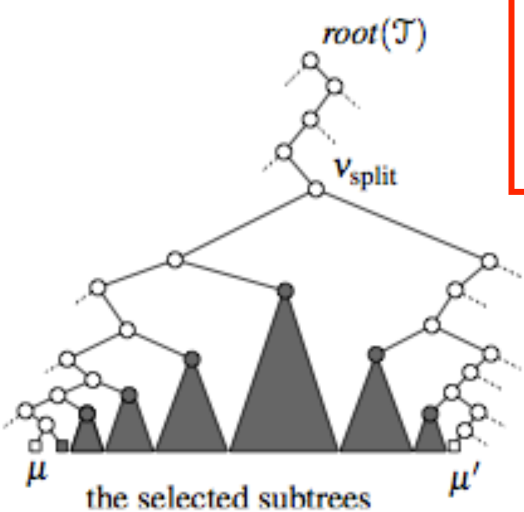
Let *lc(v)* and *rc(v)* denote the left and right child of a node *v*.

FINDSPLITNODE($\mathcal{T}, x, x'$)
Input. A tree $\mathcal{T}$ and two values x and x' with $x \leqslant x'$.
Output. The node ν where the paths to x and x' split, or the leaf where both paths end.

```
ν ← root(T)
while ν is not a leaf and (x' ≤ x_ν or x > x_ν)
    do if x' ≤ x_ν
          then ν ← lc(ν)
          else ν ← rc(ν)
return ν
```

Query algorithm

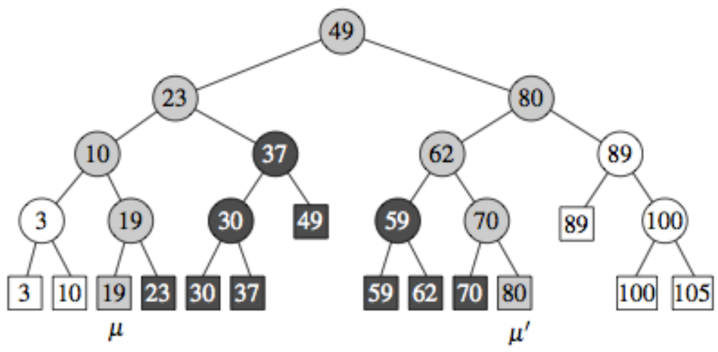
```
Algorithm 1DRANGEQUERY(T, [x : x'])
Input. A binary search tree T and a range [x : x'].
Output. All points stored in T that lie in the range.
    v_split ← FINDSPLITNODE(T, x, x')
    if v_split is a leaf
      then Check if the point stored at v_split must be reported.
      else (* Follow the path to x and report the points in subtrees right of the
              path. *)
              v ← lc(v_split)
              while v is not a leaf
                 do if x ⩽ x_v
                      then REPORTSUBTREE(rc(v))
                           v ← lc(v)
                      else v ← rc(v)
              Check if the point stored at the leaf v must be reported.
              Similarly, follow the path to x', report the points in subtrees left of
              the path, and check if the point stored at the leaf where the path
              ends must be reported.
```



One traverses the subtree rooted at a given node and reports the points stored at its leaves.

Since the number of internal nodes of any binary tree is less than its number of leaves, the time is linear in the number of reported points.

Performance of this data structure



The time spent in a query
is linear in the number of reported points : O(k).

The remaining nodes are nodes on the search path.
The paths of a balanced tree have length O(logn).
The time we spend at each node is O(1).

The query algorithm is output-sensitive !

A balanced binary search tree uses O(n) storage and is built in O(n logn) time.

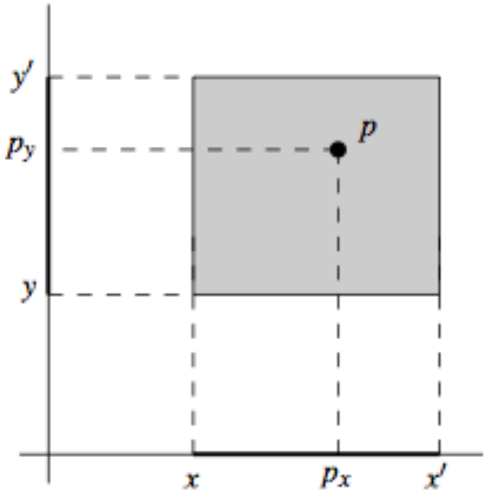
Theorem 5.2 *Let P be a set of n points in 1-dimensional space. The set P can be stored in a balanced binary search tree, which uses $O(n)$ storage and has $O(n\log n)$ construction time, such that the points in a query range can be reported in time $O(k+\log n)$, where k is the number of reported points.*

2D Range Searching

How can we generalize the data structure used for 1-dimensional range queries

-which was just a binary search tree-

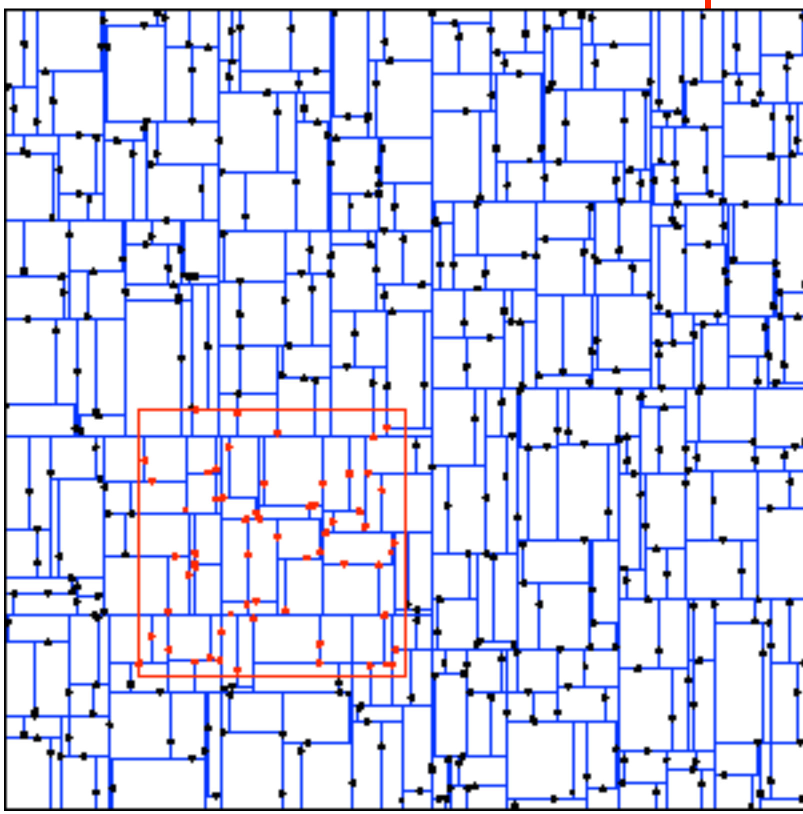
to 2-dimensional range queries?

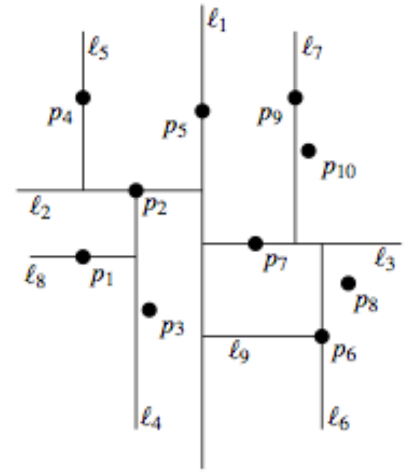


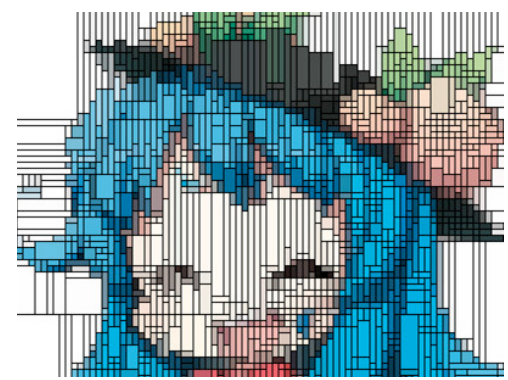
A 2-dimensional rectangular range query on P asks for the points from P lying inside a query rectangle $[x:x'] \times [y:y']$. A point $p := (p_x, p_y)$ lies inside this rectangle if and only if

$$p_x \in [x:x'] \qquad \text{and} \qquad p_y \in [y:y'].$$

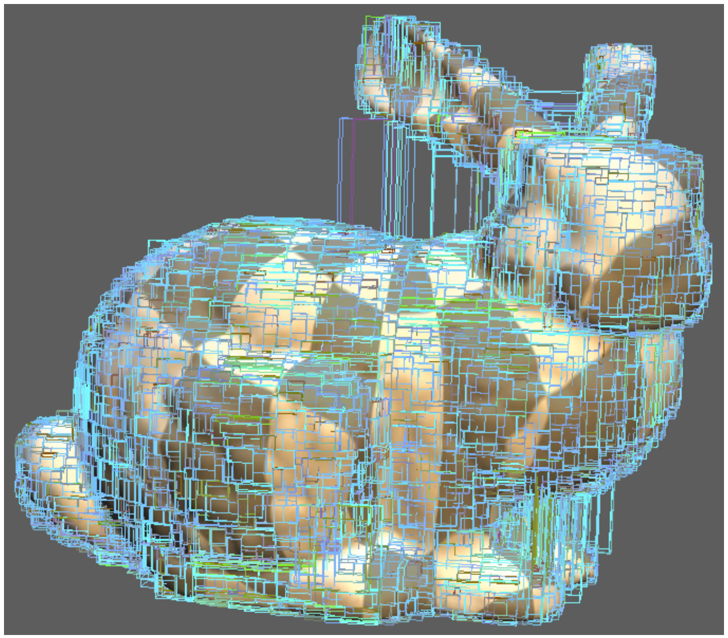
Kd-trees

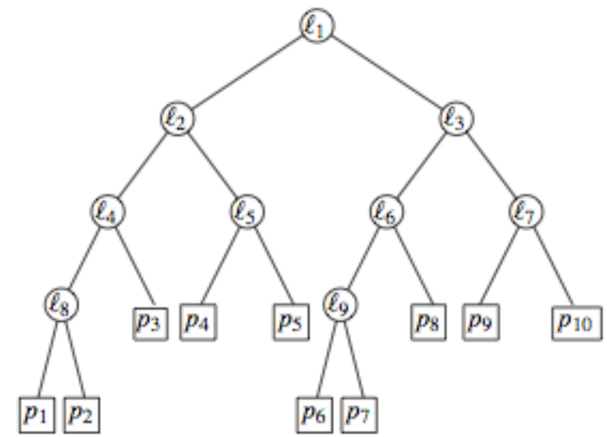




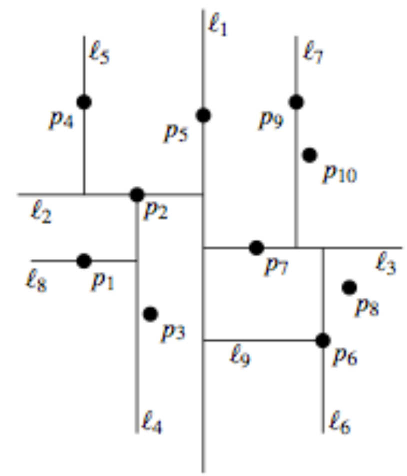


Lemma 5.3 *A kd-tree for a set of n points uses $O(n)$ storage and can be constructed in $O(n \log n)$ time.*





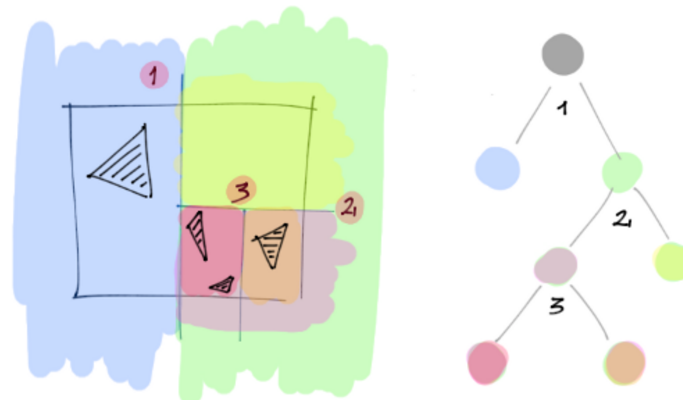
$$T(n) = \begin{cases} O(1), & \text{if } n = 1, \\ O(n) + 2T(\lceil n/2 \rceil), & \text{if } n > 1, \end{cases}$$

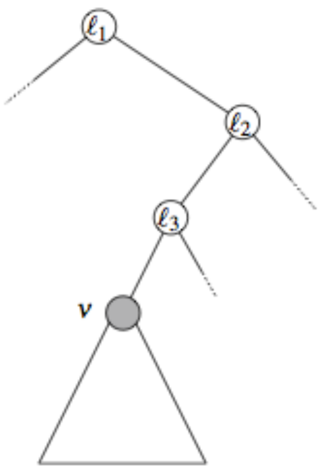


Building Kd-trees

Lemma 5.3 *A kd-tree for a set of n points uses $O(n)$ storage and can be constructed in $O(n \log n)$ time.*

Nodes of a kd-tree…

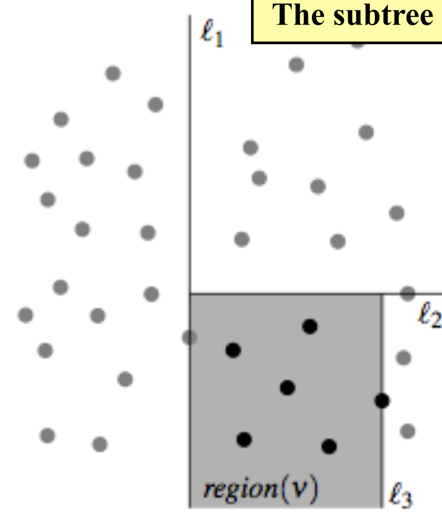




The left child of the root corresponds to the left half- plane and the right child corresponds to the right half-plane.

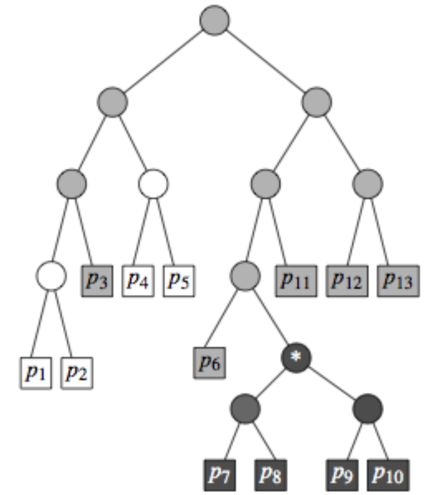
A point is stored in the subtree rooted at a node ν if and only if it lies in region(ν).

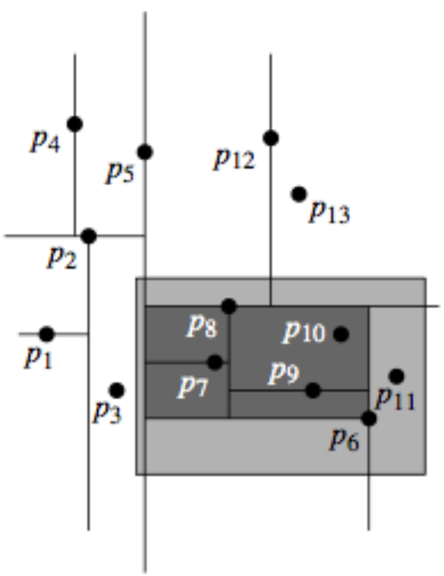
The subtree of the node stores the black dots.



… and regions of the plane

Recursive Query Procedure





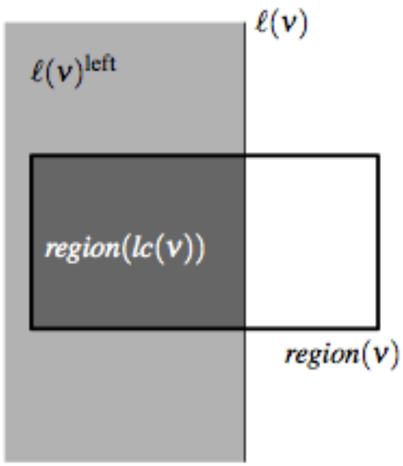
Algorithm SEARCHKDTREE(ν, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below ν that lie in the range.

```
if ν is a leaf
   then Report the point stored at ν if it lies in R.
   else if region(lc(ν)) is fully contained in R
          then REPORTSUBTREE(lc(ν))
          else if region(lc(ν)) intersects R
                 then SEARCHKDTREE(lc(ν), R)
        if region(rc(ν)) is fully contained in R
          then REPORTSUBTREE(rc(ν))
          else if region(rc(ν)) intersects R
                 then SEARCHKDTREE(rc(ν), R)
```

Recursive Query Procedure

Algorithm SEARCHKDTREE(ν, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below ν that lie in the range.

if ν is a leaf
    then Report the point stored at ν if it lies in R.
    else **if** $region(lc(\nu))$ is fully contained in R
        then REPORTSUBTREE($lc(\nu)$)
        else **if** $region(lc(\nu))$ intersects R
            then SEARCHKDTREE($lc(\nu), R$)
        if $region(rc(\nu))$ is fully contained in R
        then REPORTSUBTREE($rc(\nu)$)
        else **if** $region(rc(\nu))$ intersects R
            then SEARCHKDTREE($rc(\nu), R$)



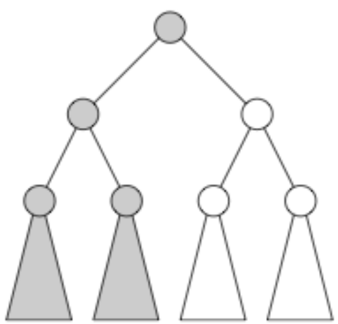
$$region(lc(\nu)) = region(\nu) \cap \ell(\nu)^{\text{left}},$$

where $\ell(\nu)$ is the splitting line stored at ν, and $\ell(\nu)^{\text{left}}$ is the half-plane to the left of and including $\ell(\nu)$.

Let us summarize the performances of kd-trees

Nodes in a d-dimensional kd-trees
Query time is given by :

$$O(n^{1-1/d}+k).$$

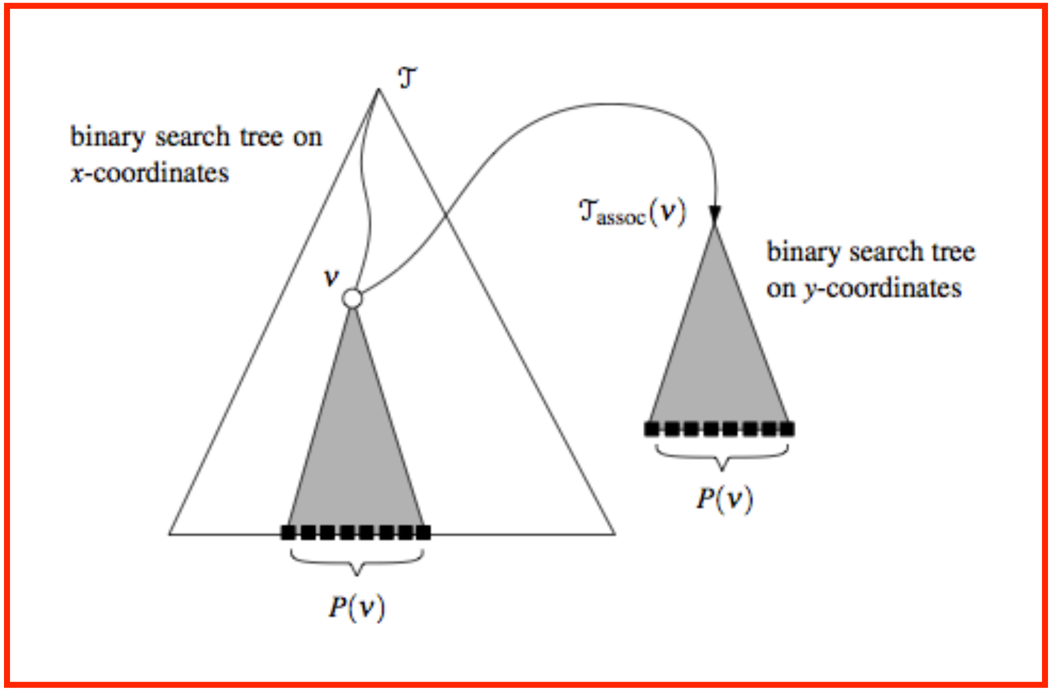


$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1. \end{cases}$$

This recurrence solves to $Q(n) = O(\sqrt{n})$.

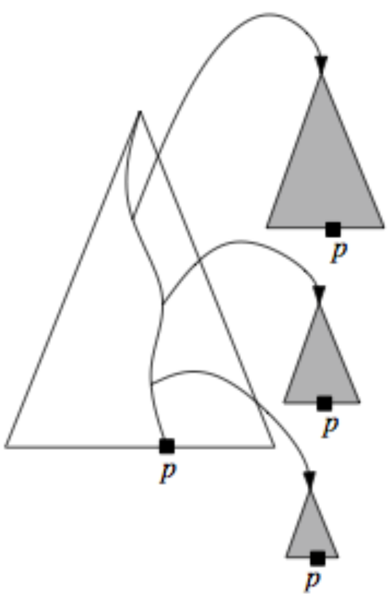
Theorem 5.5 *A kd-tree for a set P of n points in the plane uses $O(n)$ storage and can be built in $O(n \log n)$ time. A rectangular range query on the kd-tree takes $O(\sqrt{n}+k)$ time, where k is the number of reported points.*

Range-trees



Lemma 5.6 *A range tree on a set of n points in the plane requires $O(n \log n)$ storage.*

Building a range tree



Algorithm BUILD2DRANGETREE(P)
Input. A set P of points in the plane.
Output. The root of a 2-dimensional range tree.

1. Construct the associated structure: Build a binary search tree $\mathcal{T}_{\text{assoc}}$ on the set P_y of y-coordinates of the points in P. Store at the leaves of $\mathcal{T}_{\text{assoc}}$ not just the y-coordinate of the points in P_y, but the points themselves.
2. **if** P contains only one point
3. **then** Create a leaf ν storing this point, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν.
4. **else** Split P into two subsets; one subset P_{left} contains the points with x-coordinate less than or equal to x_{mid}, the median x-coordinate, and the other subset P_{right} contains the points with x-coordinate larger than x_{mid}.
5. $\nu_{\text{left}} \leftarrow$ BUILD2DRANGETREE(P_{left})
6. $\nu_{\text{right}} \leftarrow$ BUILD2DRANGETREE(P_{right})
7. Create a node ν storing x_{mid}, make ν_{left} the left child of ν, make ν_{right} the right child of ν, and make $\mathcal{T}_{\text{assoc}}$ the associated structure of ν.
8. **return** ν

Lemma 5.6 *A range tree on a set of n points in the plane requires $O(n \log n)$ storage.*

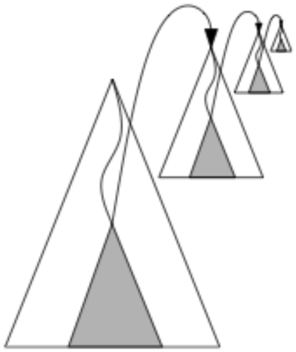
Query algorithm

Algorithm 2DRANGEQUERY($\mathcal{T}, [x : x'] \times [y : y']$)
Input. A 2-dimensional range tree $\mathcal{T}$ and a range $[x : x'] \times [y : y']$.
Output. All points in $\mathcal{T}$ that lie in the range.

$\nu_{\text{split}} \leftarrow$ FINDSPLITNODE($\mathcal{T}, x, x'$)
if ν_{split} is a leaf
  then Check if the point stored at ν_{split} must be reported.
  else (∗ Follow the path to x and call 1DRANGEQUERY on the subtrees right of the path. ∗)
    $\nu \leftarrow lc(\nu_{\text{split}})$
    while ν is not a leaf
      do if $x \leqslant x_\nu$
        then 1DRANGEQUERY($\mathcal{T}_{\text{assoc}}(rc(\nu)), [y : y']$)
          $\nu \leftarrow lc(\nu)$
        else $\nu \leftarrow rc(\nu)$
    Check if the point stored at ν must be reported.
    Similarly, follow the path from $rc(\nu_{\text{split}})$ to x', call 1DRANGE-QUERY with the range $[y : y']$ on the associated structures of subtrees left of the path, and check if the point stored at the leaf where the path ends must be reported.

Nodes in a d-dimensional ange-trees
Query time is given by :

$$O(\log^d n + k).$$



Let us summarize the performances of range trees

Theorem 5.8 *Let P be a set of n points in the plane. A range tree for P uses $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time. By querying this range tree one can report the points in P that lie in a rectangular query range in $O(\log^2 n + k)$ time, where k is the number of reported points.*

Composite Number Space

$$p := (p_x, p_y)$$

$$\hat{p} := ((p_x|p_y), (p_y|p_x))$$

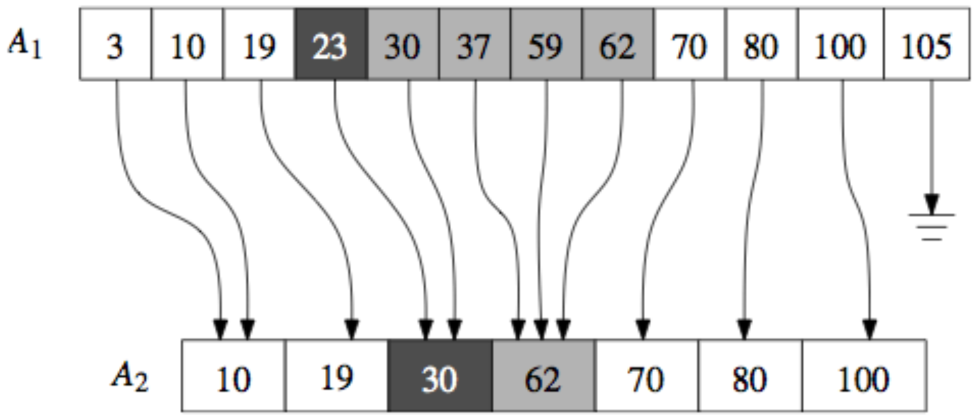
$$[x : x'] \times [y : y']$$

$$[(x|-\infty) : (x'|+\infty)] \times [(y|-\infty) : (y'|+\infty)].$$

The first coordinate of any two composite points are distinct
The same holds true for the second coordinate.
We construct kd-trees and range trees for this space with the order defined by

$$(a|b) < (a'|b') \quad \Leftrightarrow \quad a < a' \text{ or } (a = a' \text{ and } b < b').$$

Fractional Cascading

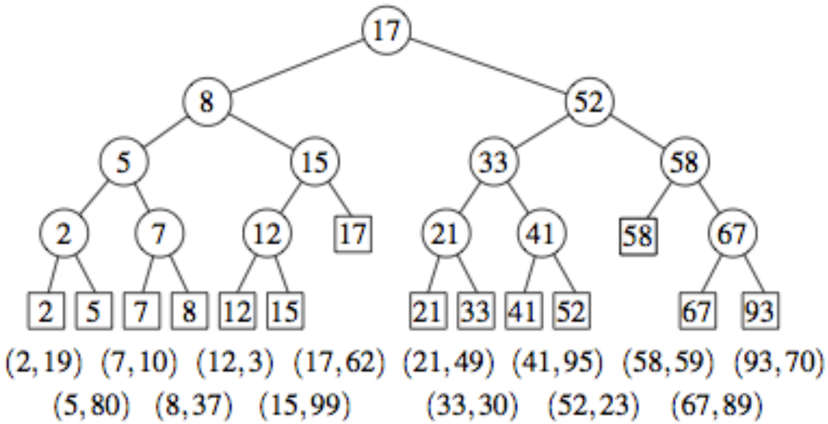


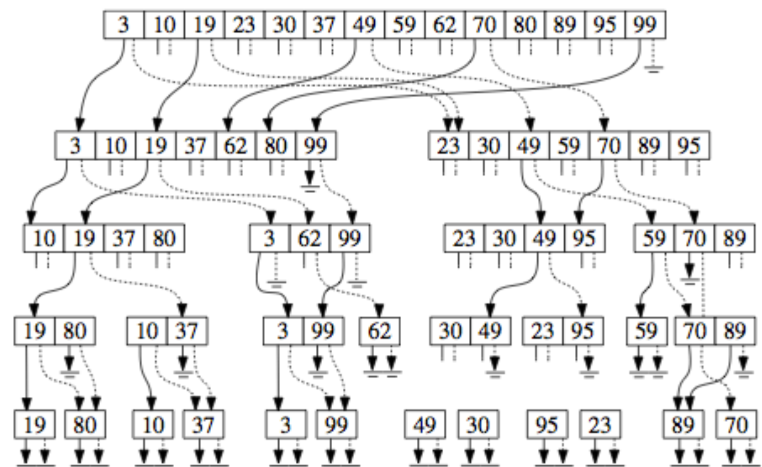
We query with the range [20 : 65].

First we use binary search in A1 to find 23, the smallest key larger than or equal to 20.
From there we walk to the right until we encounter a key larger than 65.
The objects that we pass have their keys in the range, so they are reported.

Then we follow the pointer from 23 into A2.
We get to the key 30, which is the smallest one larger than or equal to 20 in A2.
From there we also walk to the right until we reach a key larger than 65.
We report the objects from S2 whose keys are in the range.

Layered Range Tree





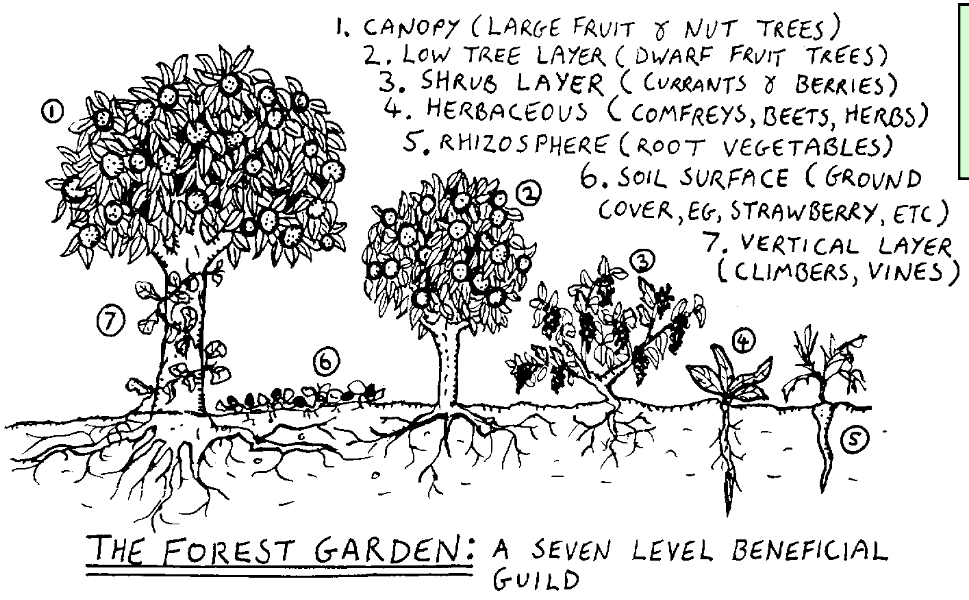
Theorem 5.11 *Let P be a set of n points in d-dimensional space, with $d \geqslant 2$. A layered range tree for P uses $O(n \log^{d-1} n)$ storage and it can be constructed in $O(n \log^{d-1} n)$ time. With this range tree one can report the points in P that lie in a rectangular query range in $O(\log^{d-1} n + k)$ time, where k is the number of reported points.*

In conclusion :-)

- **Kd tree**
  space : O(n) – build : O(n log n)
  query : O(k + sqrt n)

- **Range tree**
  space : O(n log n) – build : O(n log n)
  query : O(k + log^2 n)

- **Layered Range tree**
  space : O(n log n) – build : O(n log n)
  query : O(k + log n)



Exercice 5

5.1 In the proof of the query time of the kd-tree we found the following recurrence:

$$Q(n) = \begin{cases} O(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1. \end{cases}$$

Prove that this recurrence solves to $Q(n) = O(\sqrt{n})$. Also show that $\Omega(\sqrt{n})$ is a lower bound for querying in a kd-tree by defining a set of n points and a query rectangle appropriately.