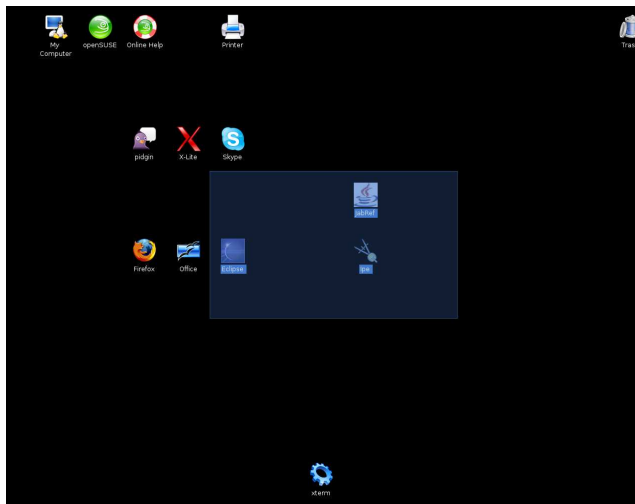


# Orthogonal Range Searching

(*kd*-trees and range trees)

---

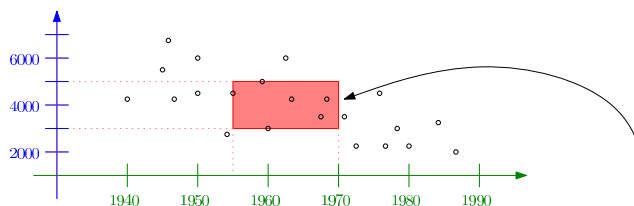
Orthogonal Range Searching – Motivational Examples – Selecting icons on a desktop



---

Orthogonal Range Searching – Motivational Examples – Querying a database

- database with records of employees
- a record is a pair: (*birthday, salary*)
- IDEA: represent each record as a point in 2-dimensional space:



- NOW: a database query such as:

“find all employees born between 1955 and 1970  
who earn between \$3000 and \$5000 per month”

corresponds to the 2-dimensional range query in the point set:

“find all points  $(x, y)$  with  $1955 \leq x \leq 1970$  and  $3000 \leq y \leq 5000$ ”

---

**Input:** set  $P$  of  $n$  points

**Task:** preprocess  $P$  into a data structure such that

- time for preprocessing is small
- data structure consumes little memory
- time for orthogonal (axis parallel) range queries is small

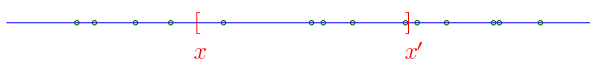
## Range Searching in one dimension

**Input:** set  $P$  of  $n$  real numbers

**Aim:** preprocess  $P$  into a data structure such that

- time for preprocessing is  $O(n \log n)$
- data structure consumes  $O(n)$  space
- time for range queries is  $O(\log n + k)$

**Range query**  $[x, x']$ : report all points in  $P$  that are between  $x$  and  $x'$



**Idea:** use a balanced binary search tree  $T$

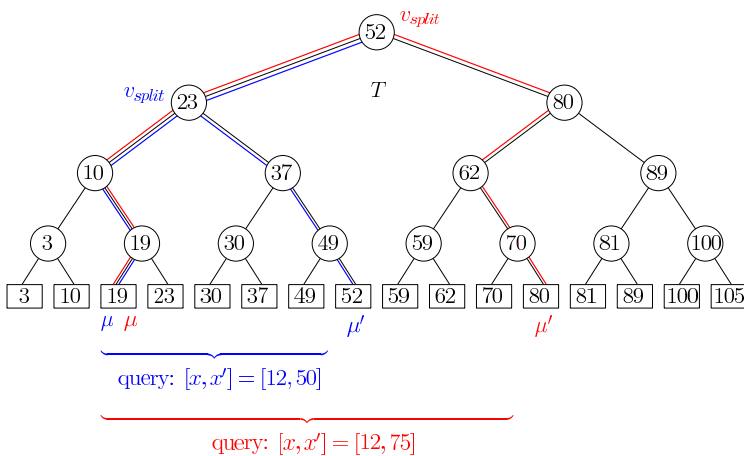
Note that leaves of  $T$  store the points of  $P$  and the internal nodes store splitting values.

- preprocessing time is  $O(n \log n)$
- space usage  $O(n)$  space
- height of  $T$  is  $O(\log n)$

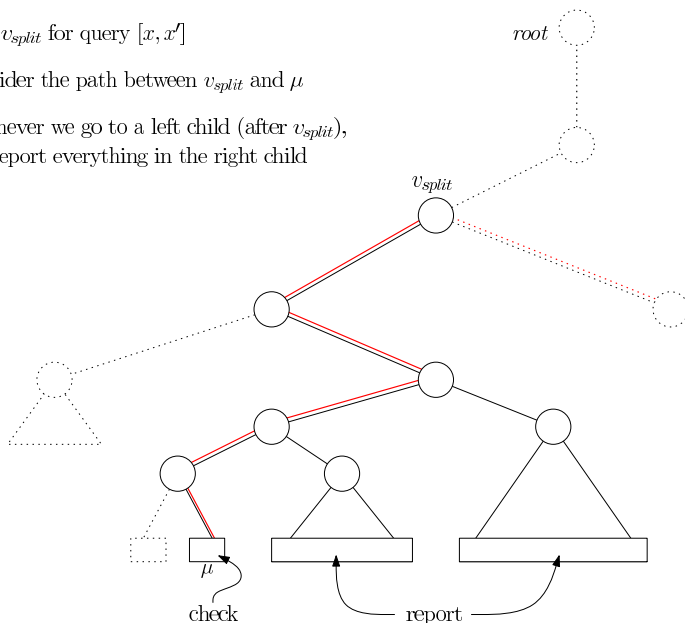
Query:  $[x, x']$

- 1) search for  $x$  and  $x'$  in  $T$   
Let  $\mu$  ( $\mu'$ ) be the leaf where the search for  $x$  ( $x'$ ) ends.
- 2) report all points in leaves between  $\mu$  and  $\mu'$ ,  
and check  $\mu$  and  $\mu'$

- splitting value stored at a node  $v$  is  $x_v$
- left subtree of  $v$  contains all points smaller than or equal to  $x_v$
- right subtree of  $v$  contains all points strictly larger than  $x_v$



- find  $v_{split}$  for query  $[x, x']$
- consider the path between  $v_{split}$  and  $\mu$
- whenever we go to a left child (after  $v_{split}$ ), we report everything in the right child



**Lemma:**

Algorithm 1DRANGEQUERY reports exactly those points that lie in the query range.

**Theorem:**

A set  $P$  of  $n$  1-dimensional points, can be stored in a balanced binary search tree, which uses  $O(n)$  space 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.

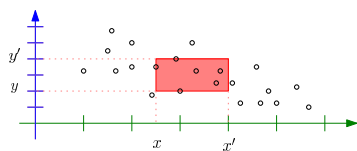
## Orthogonal Range Searching with $kd$ -trees

**Input:** set  $P$  of  $n$  points in 2-dimensional space

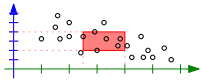
**Aim:** preprocess  $P$  into a data structure such that

- time for preprocessing is  $O(n \log n)$
- data structure consumes  $O(n)$  space
- time for orthogonal range queries is  $O(\sqrt{n} + k)$

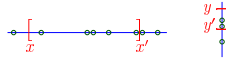
**Range query**  $[x, x'] \times [y, y']$ : report all points  $p = (p_x, p_y) \in P$  with  $x \leq p_x \leq x'$  and  $y \leq p_y \leq y'$



one 2-dimensional orthogonal range query



two 1-dimensional range queries



“ $\longleftrightarrow$ ”

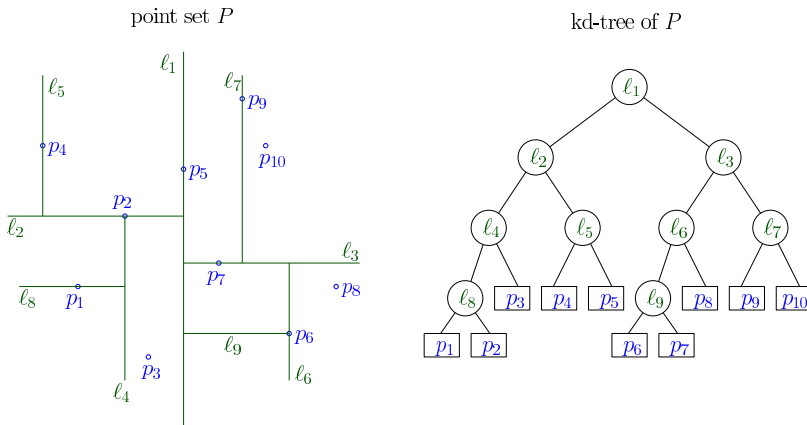
Recall: for 1D binary trees, we recursively split the set into two sets of “equal” size, one set with small values (left subtree) and one set with large values (right subtree) which will be connected to a root node

**Idea:**

- 1) split point set with respect to their  $x$ -coordinate
- 2) split point subset with respect to their  $y$ -coordinate
- 3) recursively repeat this

Orthogonal Range Searching – kd-trees – Construction

- vertical line splits set into points left of or on the line and points right of the line
- horizontal line splits set into points below or on the line and points above the line
- use median as splitting point; median of  $n$  elements is the  $\lceil \frac{n}{2} \rceil$ -th smallest element

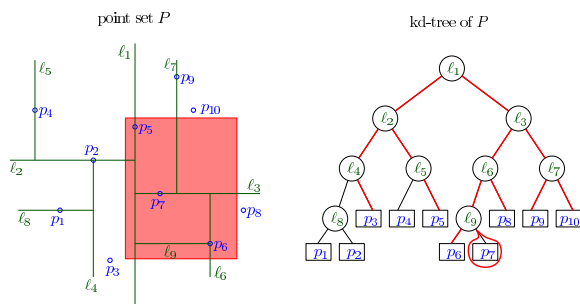


**Lemma:**

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

Orthogonal Range Searching – kd-trees – Querying

- a node  $v$  in the kd-tree  $T$  corresponds to a rectangular region,  $region(v)$
- all points in  $region(v)$  are leaves in the subtree with root  $v$
- for a query  $R$ , visit all nodes in  $T$  that intersect  $R$
- if a  $region(v)$  lies entirely within  $R$ , then report all points in the subtree of  $v$
- if  $v$  is a leaf, check if the corresponding point is inside  $R$



**Lemma:**

A query with an axis-parallel rectangle in a kd-tree storing  $n$  points can be performed in  $O(k + \sqrt{n})$  time, where  $k$  is the number of reported points.

**Theorem:**

A kd-tree for a set of  $n$  points in the plane uses  $O(n)$  space 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.

- worst case bounds are tight
- in practise often better bounds, because of small query regions

- in  $d$  dimensions:
  - split point set with respect to their 1st coordinate
  - split point subset with respect to their 2nd coordinate
  - ...
  - split point subset with respect to their  $d$ th coordinate
  - recursively repeat this
- complexity:  $O(n)$  space
  - $O(n \log n)$  preprocessing time (for constant  $d$ )
  - $O(n^{1-\frac{1}{d}} + k)$  query time

## Orthogonal Range Searching with Range trees

---

- if number of reported points is small then  $O(\sqrt{n})$  query time is very high
- better is not possible, if we only allow  $O(n)$  space

**Aim:** preprocess  $P$  into a data structure such that

- time for preprocessing is  $O(n \log n)$
  - data structure consumes  $O(n \log n)$  space
  - time for orthogonal range queries is  $O(\log^2 n + k)$
- range trees allow for faster queries, but need more space
  - “using kd-trees” vs. “using range trees” depends on application

- recall:



**Idea** for a query:  $[x, x'] \times [y, y']$

- first: find all points in  $P$  with  $x$ -coordinate in  $[x, x']$   
using a balanced binary search tree
- then: among those points, find all points with  $y$ -coordinate in  $[y, y']$   
using other balanced binary search trees

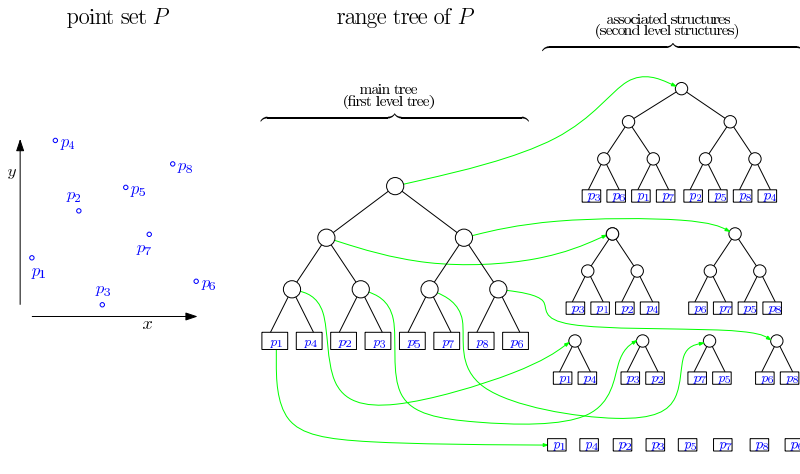
- the main data structure is a balanced binary search tree built on  $x$ -coordinates
- the points in the leaves of the subtree of node  $v$  are called *canonical subset*  $P(v)$  of  $v$

**Observations:**

The subset of points with  $x$ -coordinate in  $[x, x']$  can be computed by a 1-dimensional range query and can be expressed as the disjoint union of  $O(\log n)$  canonical subsets.

Of these points, we only want to report the points that lie in  $[y, y']$ . This is another 1-dimensional range query.

- for each node  $v$ ,  $P(v)$  is stored in a balanced binary search tree  $T_a(v)$  built on the  $y$ -coordinates of the points
- $T_a(v)$  is called *associated structure* of  $v$



**Lemma:**

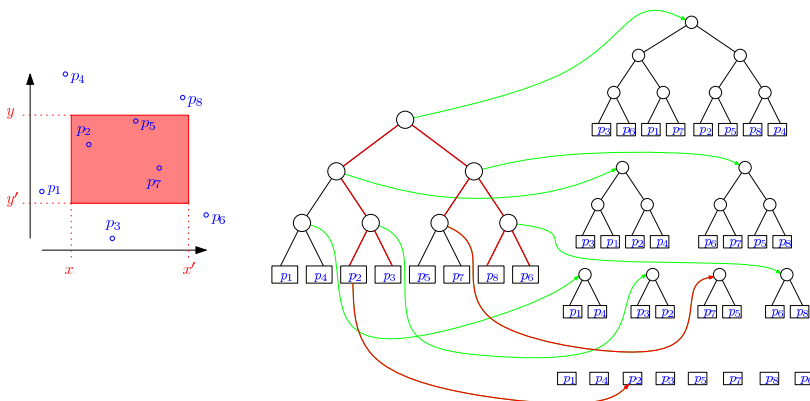
A range tree on a set of  $n$  points in the plane requires  $O(n \log n)$  space.

**Lemma:**

A range tree on a set of  $n$  points in the plane can be computed in  $O(n \log n)$  time.

query  $[x, x'] \times [y, y']$

- perform a 1-dimensional query in the main tree with range  $[x, x']$
- whenever we find subtrees with points in the range  $[x, x']$ , instead of reporting them all, we perform another 1-dimensional range query in the associated tree with range  $[y, y']$



**Lemma:**

A query with an axis-parallel rectangle in a range tree storing  $n$  points takes  $O(\log^2 n + k)$  time, where  $k$  is the number of reported points.

**Theorem:**

Let  $P$  be a set of  $n$  points in the plane.

- a range tree for  $P$  uses  $O(n \log n)$  space
- a range tree for  $P$  can be computed in  $O(n \log n)$  time
- rectangular range queries take  $O(\log^2 n + k)$  time, where  $k$  is the number of reported points

- in  $d$  dimensions:
  - construct a balanced binary search tree on 1st coordinate
  - for each node  $v$  construct an associated structure  $T_a(v)$ ;  
 $T_a(v)$  is a  $(d - 1)$ -dimensional range tree of the points in  $P(v)$ , restricted to their last  $d - 1$  coordinates  
(recursively constructed)
- complexity:  $O(n \log^{d-1} n)$  space  
 $O(n \log^{d-1} n)$  preprocessing time  
 $O(\log^d n + k)$  query time