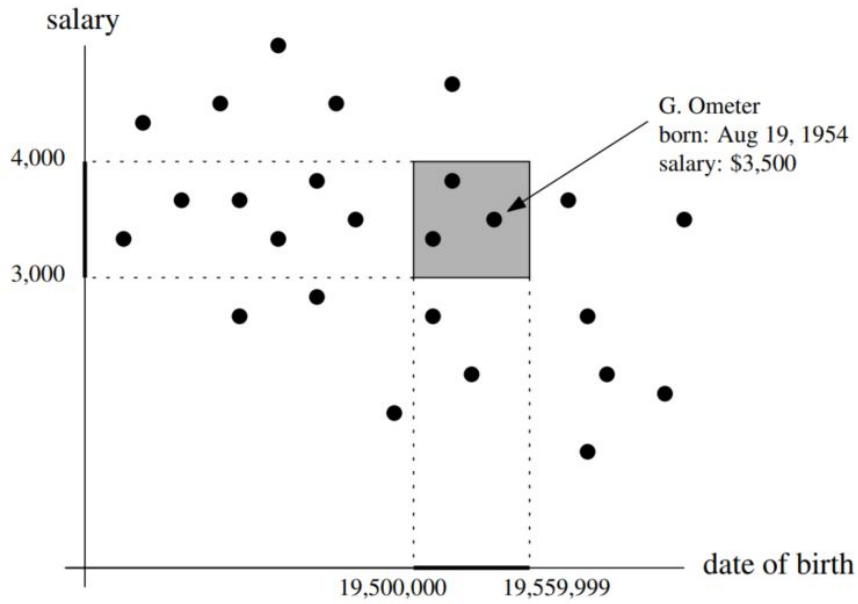
FTP_Alg
Orthogonal Range Searching and Kd–Trees

jungkyu.canci@hslu.ch

HS2024

Orthogonal Range Searching

So far we have mainly considered 1-dimensional data structures (i.e. linear array). We used binary trees for sorting data structures with a 1-dimensional key. Consider the following:

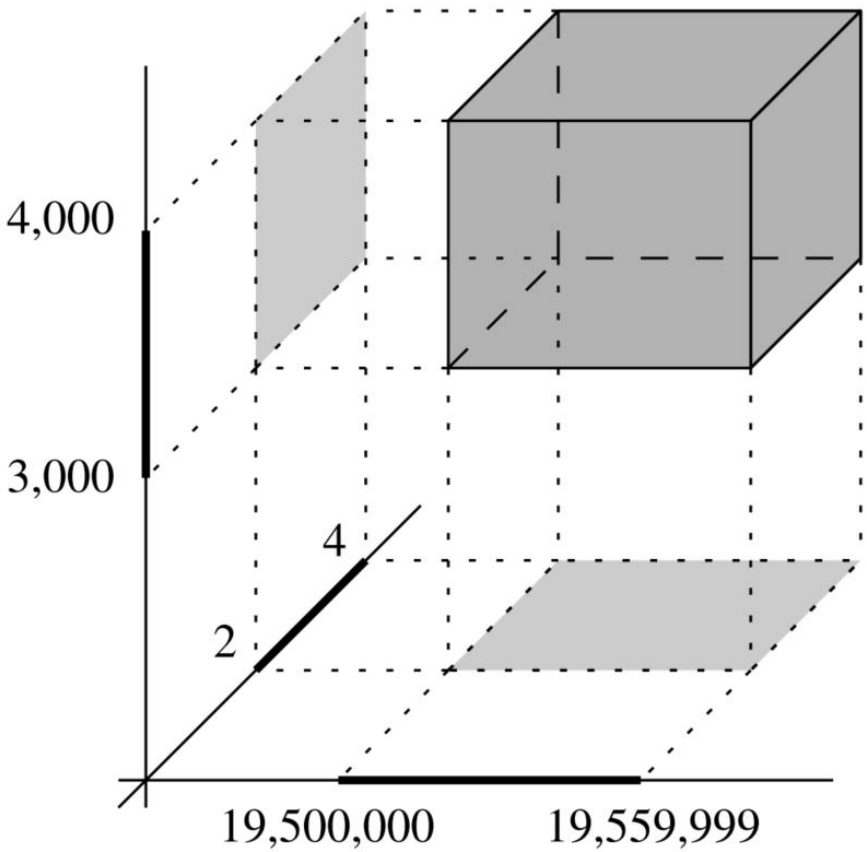


It represents a database for personnel administration. It contains a list of employees stored by using the monthly salary and the date of birth (format $10'000 \times \mathit{year} + 100 \times \mathit{month} + \mathit{day}$).

The grey area represents the answer to the query asking "for all employees born between 1950 and 1955 and earning between 3000 and 4000 dollars". The grey area is called **orthogonal range query**.

d-Dimensional Range Searching

Orthogonal range query are also called **rectangle range query**. The one considered before was a 2 dimensional rectangle range query.



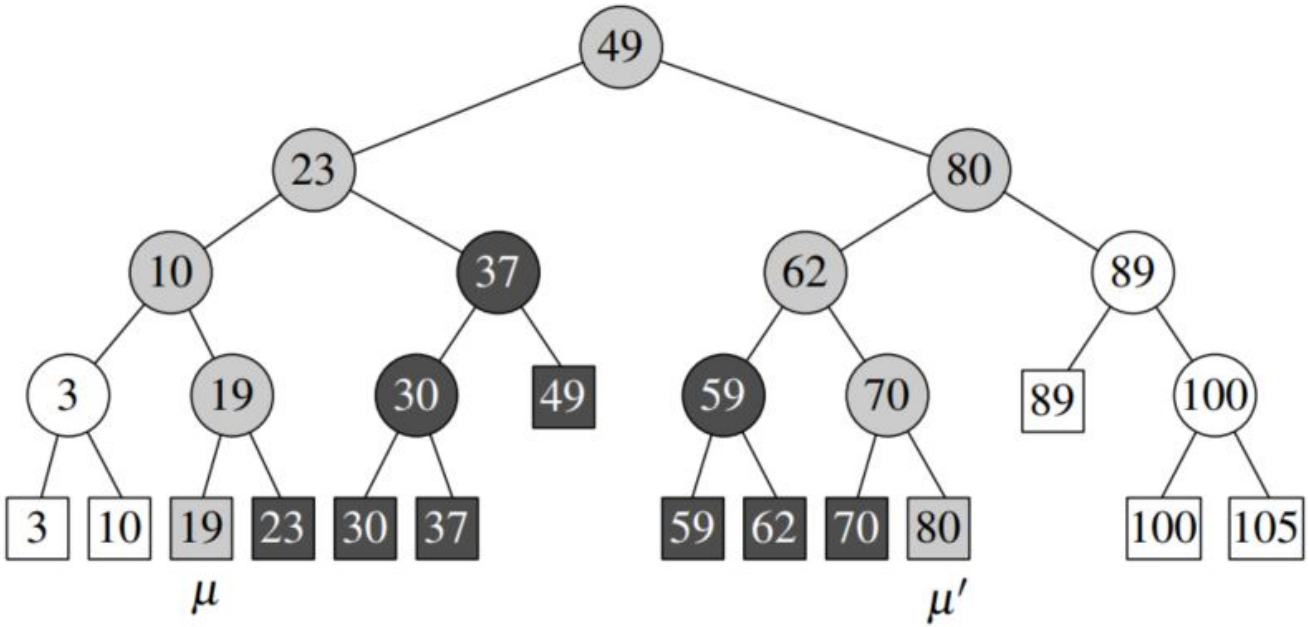
Right we have a three dimensional rectangle range query, where we consider the previous data base, where we add as information the number of children of a employee. The rectangle identifies employees born in the interval $[1950, 1955]$ with monthly salary in $[3'000, 4'000]$ with 2, 3 or 4 children.

1-Dimensional Range Searching

- KD-Tree, which will be considered in the following part, are used for determining orthogonal range queries. We start by presenting 1–dimensional range searching.
- Let $P = [p_1, p_2, \ldots, p_n]$ be a set of n real numbers. We create a complete binary search tree (also called balanced binary search tree) $\mathscr{T}$.
- The points of P are stored in the node leafs of $\mathscr{T}$. The internal nodes store splitting values to guide the search.
- Let $x \leq x'$. To report points in a query range $[x, x']$, we look at the leaves μ and μ', where the searches end.
- The points of the rectangle range query are leaves between μ and μ'.

Split–Node

Next figure concerns a search with interval $[18, 77]$.

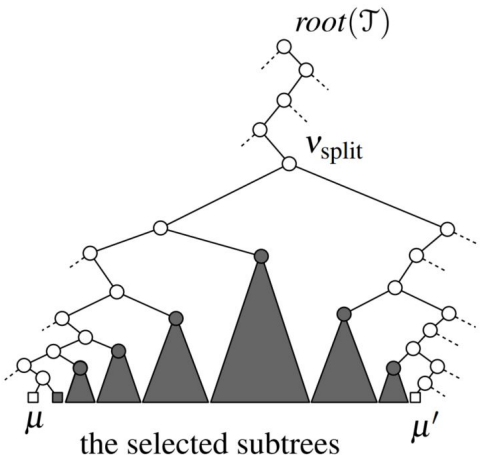


The picture represents the (sub)tree rooted in a so called split–node, that is the smallest (complete) tree containing the searched leaves. Thus first step in the query algorithm is to determine the split node.

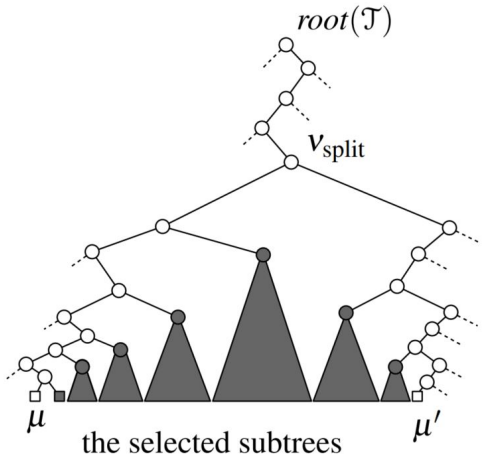
Split–Node

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

1. $v \leftarrow root(\mathcal{T})$
2. **while** v is not a leaf **and** ($x' \leqslant x_v$ **or** $x > x_v$)
3. **do if** $x' \leqslant x_v$
4. **then** $v \leftarrow lc(v)$
5. **else** $v \leftarrow rc(v)$
6. **return** v



Starting from v_{split} we then follow the search path of x. At each node where the path goes left, we report all the leaves in the right subtree, because this subtree is in between the two search paths. Similarly, we follow the path of x' and we report the leaves in the left subtree of nodes where the path goes right.



Finally, we have to check the points stored at the leaves where the paths end; they may or may not lie in the range $[x : x']$.

The query algorithm uses the subroutine REPORTSUBTREE, which 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, this subroutine takes an amount of time that is linear in the number of reported points.

1DRangeQuery-PseudoCode

Algorithm 1DRANGEQUERY($\mathcal{T}, [x : x']$)
Input. A binary search tree $\mathcal{T}$ and a range $[x : x']$.
Output. All points stored in $\mathcal{T}$ that lie in the range.

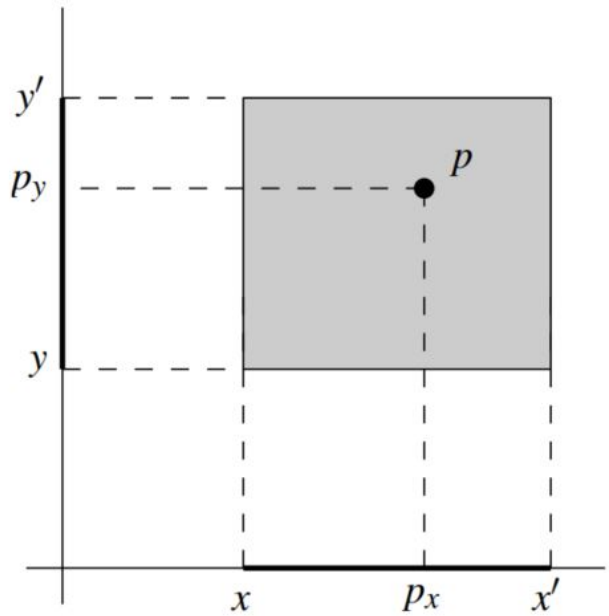
1. $v_{\text{split}} \leftarrow$FINDSPLITNODE($\mathcal{T}, x, x'$)
2. **if** v_{split} is a leaf
3. **then** Check if the point stored at v_{split} must be reported.
4. **else** ($*$ Follow the path to x and report the points in subtrees right of the path. $*$)
5. $v \leftarrow lc(v_{\text{split}})$
6. **while** v is not a leaf
7. **do if** $x \leqslant x_v$
8. **then** REPORTSUBTREE($rc(v)$)
9. $v \leftarrow lc(v)$
10. **else** $v \leftarrow rc(v)$
11. Check if the point stored at the leaf v must be reported.
12. 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.

1DRangeQuery-PseudoCode: Running time

Recall that for creating a binary balanced tree with n leaves we need a tree with $O(n)$ nodes and the running time to create it is $O(n \log n)$. Note that the FINDSPLITNODE take $O(\log n)$ time. REPORTSUBTREE is linear in the number of reported points k, thus is $O(k)$.
Therefore the above query algorithm needs $O(n)$ storage, $O(n \log n)$ for creating the binary search tree and $O(k + \log n)$ for reporting k nodes.

Kd–Trees

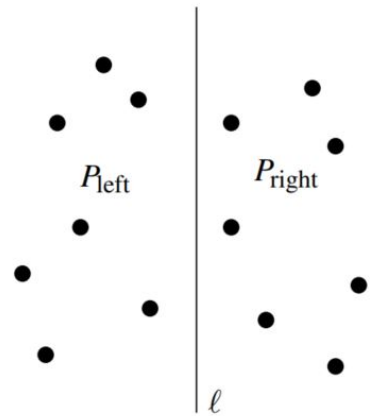
- Now we consider a 2–dimensional rectangular range searching problem.
- We consider a set of n–points in the plane, by giving their x–coordinates and y–coordinates.
- We consider a query rectangle $[x, x'] \times [y, y']$



As we have done in 1–dimensional range we create a data structure for the query. We defined a binary search tree by iterating the split in two subset of roughly the same size.

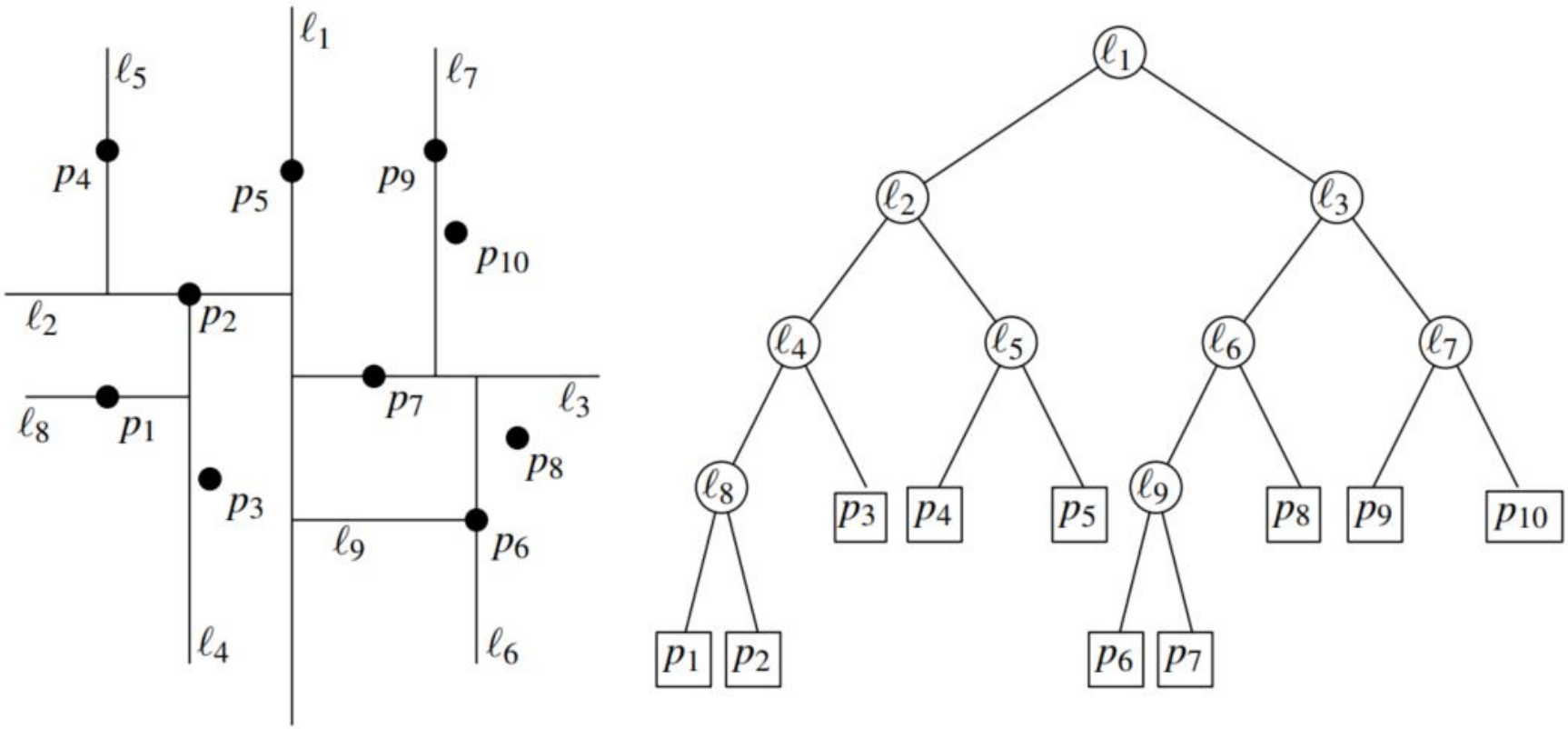
Building a Kd–Tree

We generalize the above procedure, but this time we have to consider two coordinates, x and y. We start by considering a vertical line l, which splits the set P (on which we apply the query) in two sets of roughly equal size. This first splitting line l_1 is stored at the root.



The points in P_{left} (in P_{right}) will be stored on the left (right) part of the tree. Now we divide P_{left} (P_{right}) with a horizontal line called l_2 (l_3), then we divide all subregions with 4 vertical lines... and so on.

Building a Kd–Tree: Example



We construct a Kd–Tree with the following recursive procedure, whose input is a set of points P and a *depth*. The depth is important in the recursion as we alternate splitting with an horizontal line (depth is an even number) or with a vertical line (depth is an odd number).

BUILDKDTREE(P, *depth*)

Algorithm BUILDKDTREE(P, *depth*)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.

1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** *depth* is even
4. **then** Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.
5. **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.
6. $v_{\text{left}} \leftarrow$ BUILDKDTREE(P_1, *depth* $+ 1$)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE(P_2, *depth* $+ 1$)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

We will see as an exercise, that the procedure for a set of n points has running time is $O(n \log n)$ and uses $O(n)$ storage.

Reference

The material of these slides is taken form the book "Computational Geometry" by de Berg et al., Section 5.1 and Section 5.2