Computational Geometry

4. Range search

4.1 Motivation

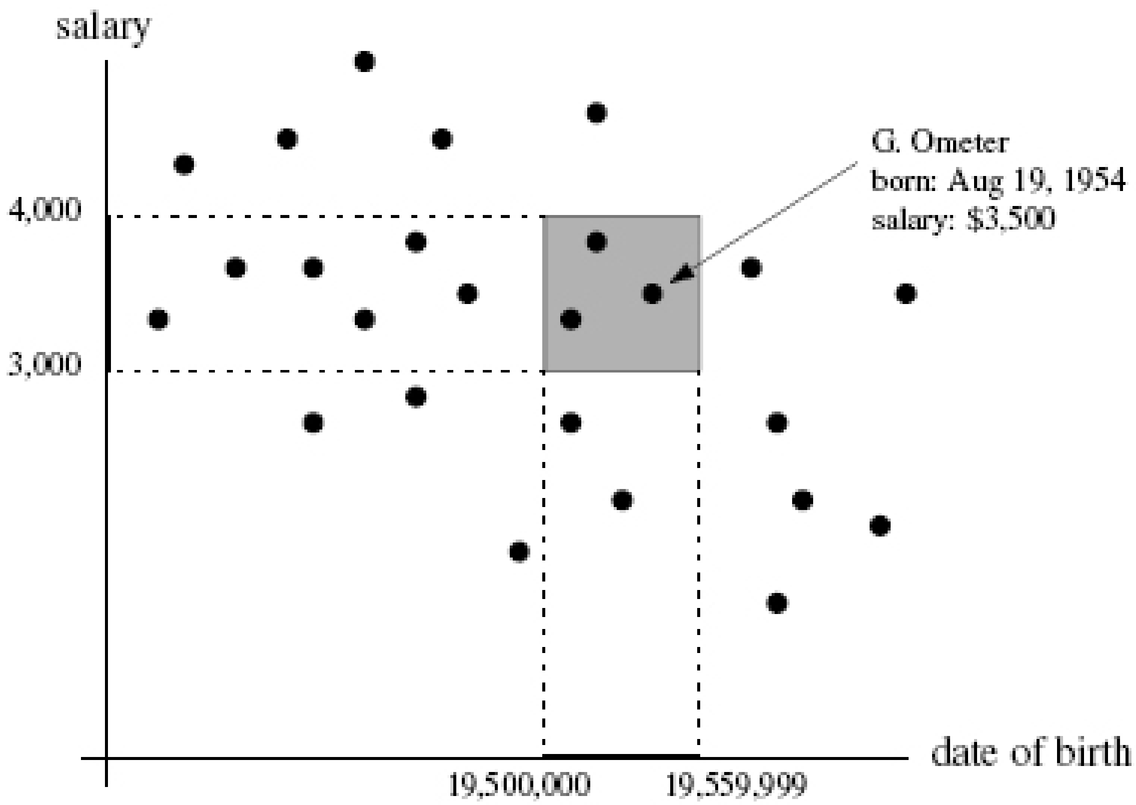
- Data base with names, dates of birth and salary of employees of a company.
- Query: All employees born between 1950 and 1955 earning between 3.000 \$ and 4.000 \$.
- Solution: Compute a single number from the birthday:

$$10.000 \times \mathit{Year} + 100 \times \mathit{Month} + \mathit{Day}$$

  - This yields an order for the birthdays that can be used with the income in a *2D-range-query*.

4.1 Motivation

- Geometric interpretation



4.1 Problem

- Input: Set $P = \{p_1, \ldots, p_n\}$ of n points in a k-dimensional space (kd-space) and a k-dimensional *axis-aligned query range* D.
- Output: All points in $P \cap D$.
- Goal
  - Many queries for the same point set P with different query ranges D should be computed as fast as possible.
- Idea
  - Store P in a data structure, that supports range-queries as efficiently as possible.

4.1 Problem

More examples for different dimensions

- $k = 1$
  - Enumerate all elements in a sequence of keys between a and b.
- $k = 2$
  - Enumerate all cities in a square of 100 km edge length and center in Kaiserslautern.
- $k \geq 3$
  - Data bases queries, e.g. find all persons, that
    - are 20 to 30 years old,
    - earn 30.000 € to 40.000 € and
    - do not posses a cell phone.

4.2 Multi-dimensional search

Naïve approach

- Test all points sequentially, if the actual point lies in the query range $D = [x_l, x_r] \times [y_l, y_r]$.
- Run time: $O(n)$ for every range search.
- Inefficient, if the search result for every range-query contains only a small constant number of points.
- An output-size sensitive algorithm would be beneficial, where the run time depends on the number of points in the query range.

4.2 Multi-dimensional search

Quad-tree approach (grid-method)

- Place regular grid over the set of points P.
- Consider only points in grid cells, that intersect the query range.
- Efficient, if the points are *uniformly* distributed.
- Inefficient, if the points are concentrated in one grid area and the grid has many empty grid cells.

4.2 Multi-dimensional search

BSP-tree approach

- Partition the point set P along arbitrary (hyper-)planes.
- Inefficient, because the intersection of a rectangular query range with arbitrary half-spaces is an arbitrary convex polygon (convex polytope in kd).
- Efficient, if the (hyper-)planes are axis-aligned:

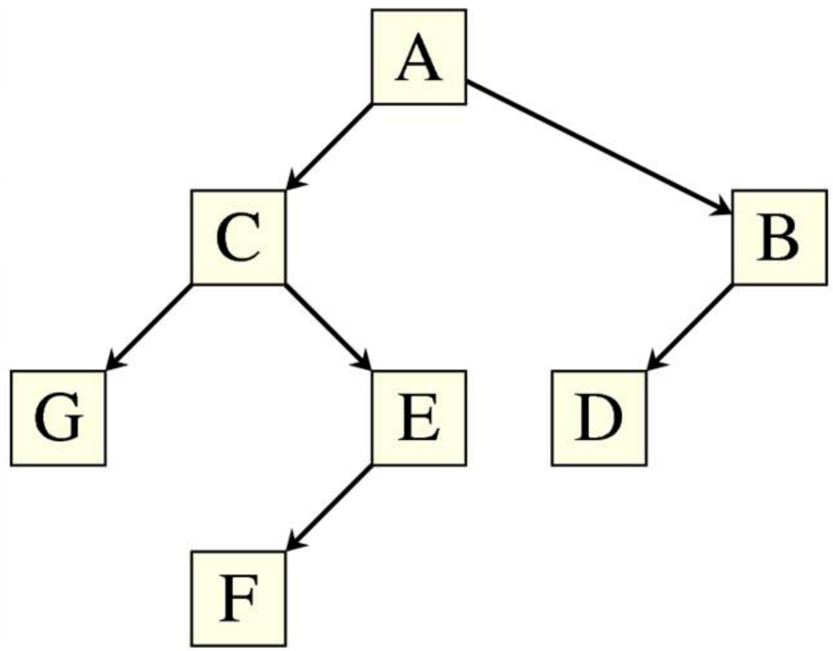
two-dimensional trees (kd-trees).

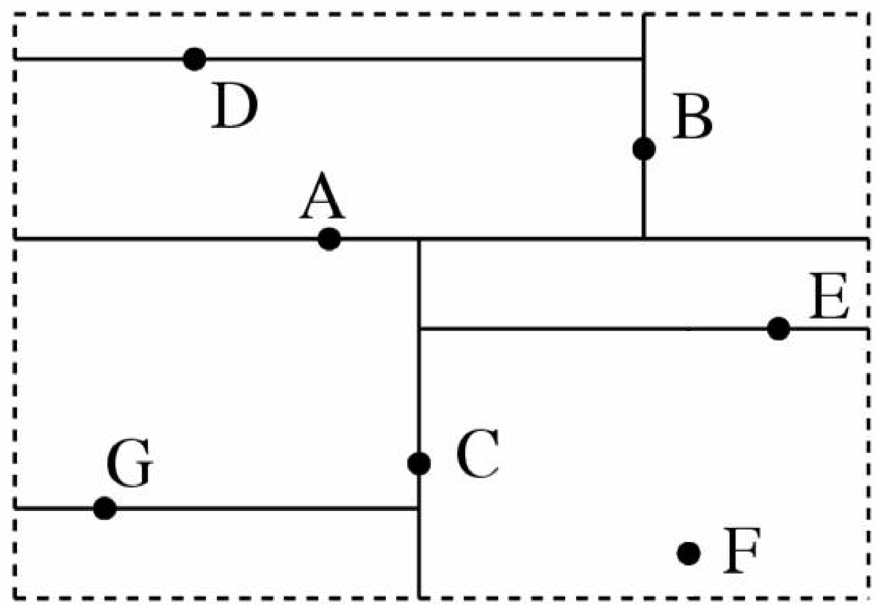
4.3 kd-trees

- In the sequel only two-dimensional range search, i.e. $k = 2$.
- Extension to higher dimensions is then straight-forward.
- Idea
  - Partition 2d search space in a way similar to a binary search tree for the one-dimensional search space.
  - Use alternating the x- und y-coordinates as keys.
  - Construct a binary tree, representing a partition of the plane:
    - The knots correspond to the n points in 2d.
    - On an even level of the tree (level=depth+1) use the x-coordinate as key, otherwise the y-coordinate.

4.3 *k*d-trees

- A 2d-search tree and the resulting partition of the plane.





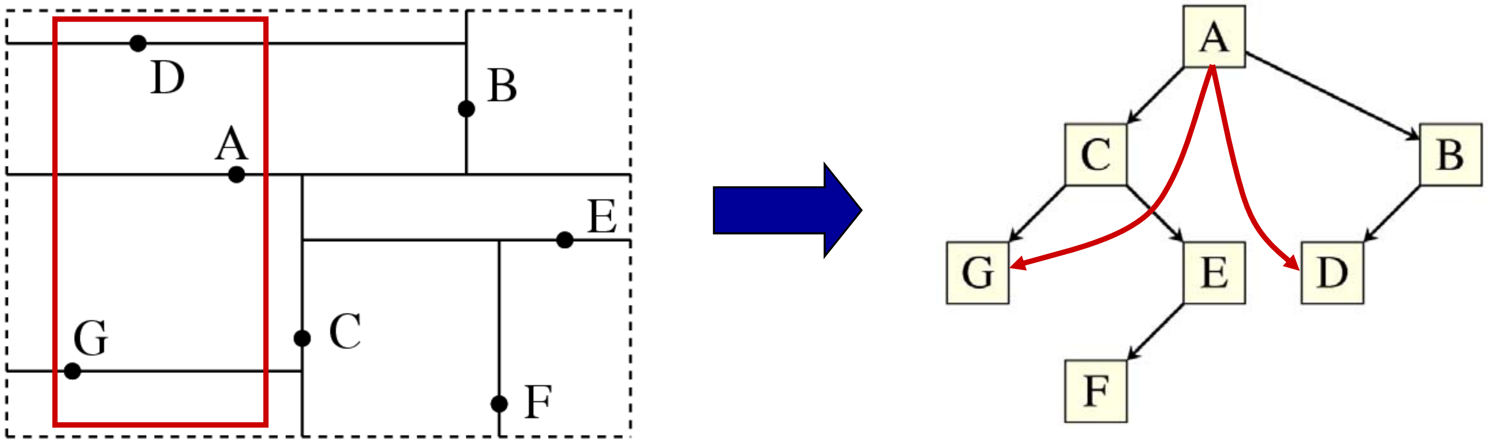
4.3 kd-trees

Properties

- For a knot in the tree v_x representing point p_x, whose key is its x-coordinate (even level), we have:
  - The left sub-tree of v_x contains the points left of p_x.
  - The right sub-tree of v_x contains the points right of p_x.
  - The sub-trees of children of v_x are determined by y-coordinates.
- For a knot in the tree v_y representing point p_y, whose key is its y-coordinate (odd level), we have:
  - The left sub-tree of v_y contains the points below of p_y.
  - The right sub-tree of v_y contains the points above of p_y.
  - The sub-trees of children of v_y are determined by x-coordinates.

Search in a 2d-tree

- The search is analogous to the usual search in a binary search tree.
- **Attention:** For every visited knot in the tree, the sub-trees of the knot are determined by the x-coordinate **or** the y-coordinate.
- **Attention:** For a range search both sub-trees must be considered.
- Example:



4.3 *k*d-trees

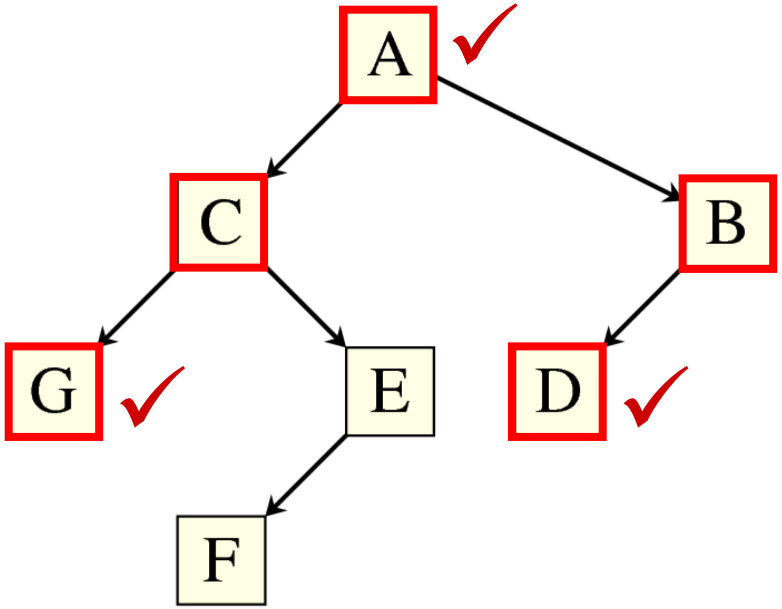
Algorithm 1: RangeSearch(knot *k*, direction *d*, query range *D*)

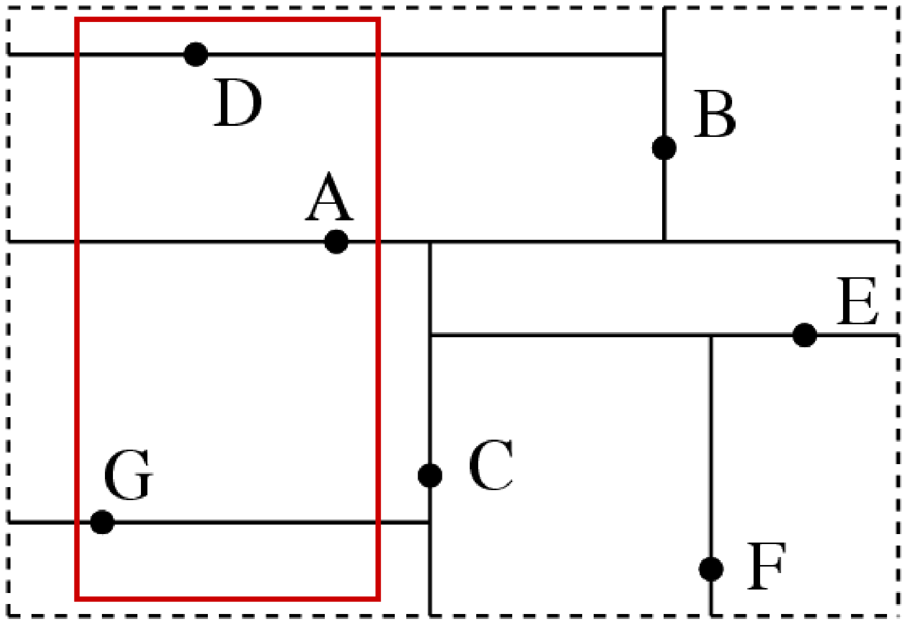
```
if (k ≠ NULL) then {
  if (d == vertical) then {
    (l,r) = (D.y1,D.y2);
    coord = k.y;
    dNew  = horizontal;
  } else {
    (l,r) = (D.x1,D.x2);
    coord = k.x;
    dNew  = vertical;
  }
  if (k ∈ D     ) then Add k to the output;
  if (l < coord) then RangeSearch(k.left ,dNew,D);
  if (r > coord) then RangeSearch(k.right,dNew,D);
}
```

Call: RangeSearch(1,*n*,*root*,*vertical*)

4.3 *k*d-trees

- Example





4.3 *k*d-trees

Run time

- If the 2d-tree is balanced, the height of the tree is $O(\log n)$.
- If the 2d-tree is unbalanced, a path in the tree might degenerate to a linear list resulting in a bad run time.

- Goal: The construction of the tree must guarantee the balancedness.
- Solution: Partition the points at the median of the sequence of x- and y-coordinates.

4.3 *k*d-trees

Partitioning at the median

- Sort all points both by x- and y-coordinates
  - ➡ Two sorted sequences X and Y.
- Subdivide Y at its median and take it as the root of the tree
  - ➡ Two sub-sequences Y_1 and Y_2.
- Partition X into two sequences X_1 and X_2, such that X_1 contains the same points as Y_1 and X_2 the same points as Y_2.
- Partition X_1 and X_2 recursively at the median and subdivide Y_1 and Y_2 accordingly as above, until these sequences contain only one point.
  - These points are the leafs of the tree.

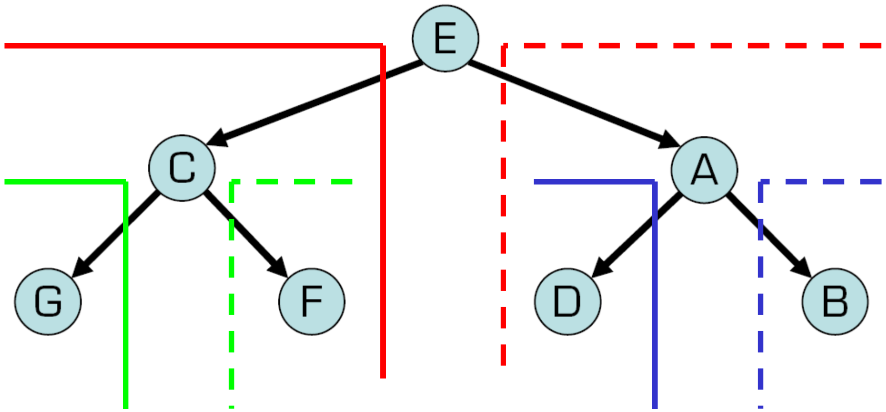
4.3 *k*d-trees

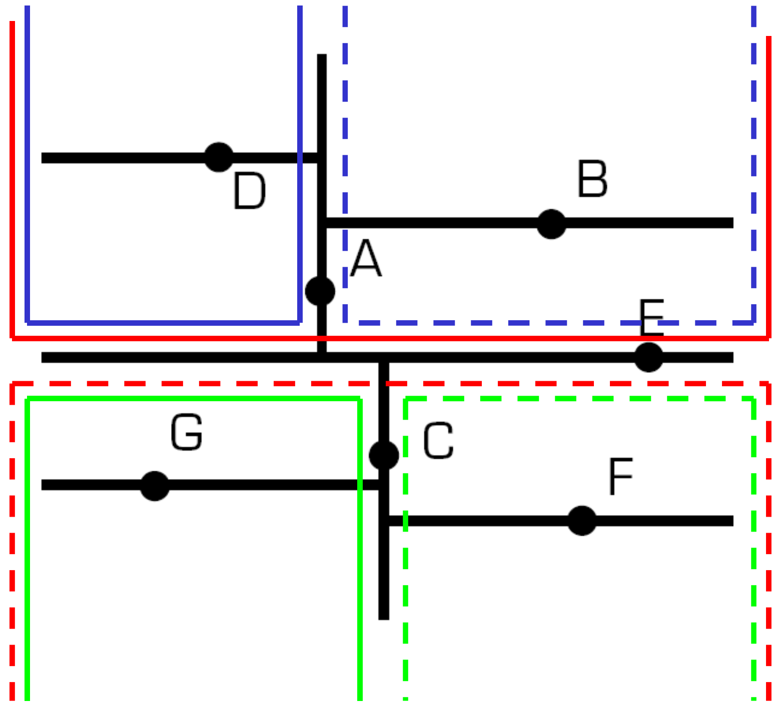
- Example for the partition

X: G D A C B F **E**
Y: F G C **E** A B D

X_1: G **C** F
Y_1: F G **C**

X_2: D **A** B
Y_2: **A** B D





- Global, pre-sorted sequences X and Y.

Algorithm 2: ConstructBalanced2DTree

Input: left index l, right index r, knot k, direction d

```
if (l ≤ r) then {
   m = ⌈(l+r)/2⌉;
   if (d == vertical) then {
      k.value = Y[m];
      PartitionField(X,l,r,m);
   } else {
      k.value = X[m];
      PartitionField(Y,l,r,m);
   }
   ConstructBalanced2dTree(l  ,m-1,k.left ,!d);
   ConstructBalanced2dTree(m+1,r  ,k.right,!d);
}
```

Call: ConstructBalanced2DTree(1,n,*root*,*vertical*)

4.3 *k*d-trees

Run time

- Construction
  - Sorting of sequences: $O(n \log n)$.
  - Partitioning of the sequences: $O(n)$.
  - Recursive calls: $T(n) = T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + O(n)$
  - Solution of the recursion: $O(n \log n)$.

- Range search in a balanced 2d-tree
  - Run time for R points in the query range D: $O(\sqrt{n} + R)$.
  - Proof: Black board.
  - For small R this is much faster than testing all points.

4.3 kd-trees

Higher dimensions

- Use the same algorithm!
- Partition for the construction of the tree cycle through the dimensions one by one.
- For the search also cycle through the dimensions one after the other.
- Run time and storage in k dimensions
  - Construction: $O(kn \log n)$
  - Range search: $O(kn^{1-\frac{1}{k}} + R)$
  - Storage: $O(n)$ (A kd-tree is a binary tree with n leaves.)

4.4 Range-trees

- For large n and small R the query time of a kd-tree is relatively large.

- Principle of kd-trees:

  Comparisons of x- and y-coordinates alternate.

- Principle of range-trees:

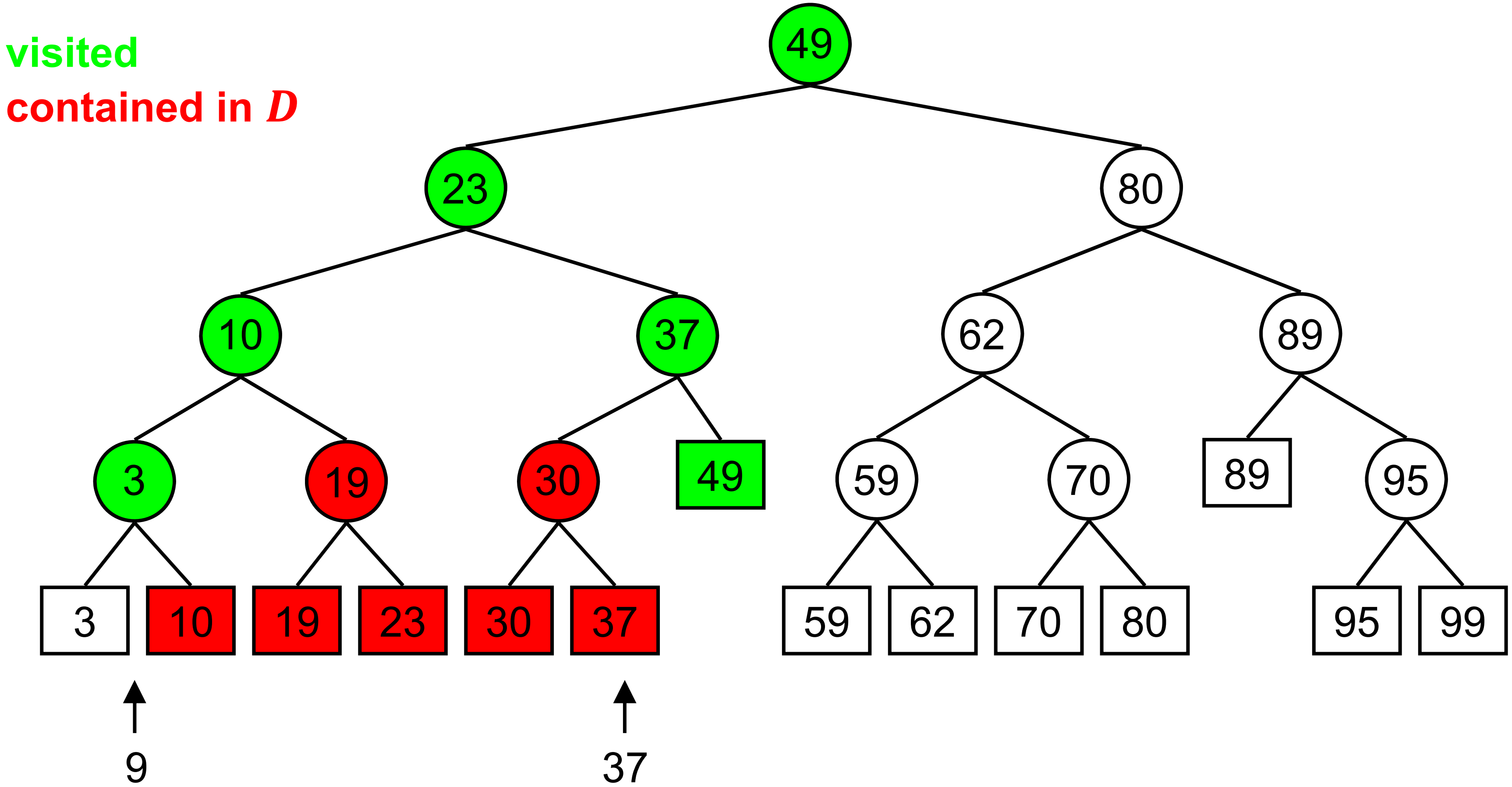
  Do comparisons of x- and y-coordinates subsequently.

4.4 Range-trees

- First the x-coordinates:
  - First level data structure: Use a balanced binary tree for the x-coordinates of the points.
  - Query: Use a one-dimensional range query to find all points with x-coordinates in $[x_l, x_r]$.
  - This yields a list of candidates that lie potentially in the query range D.
- Second the y-coordinates:
  - Second level data structure: Use a balanced binary tree for the y-coordinates of the candidates.
  - Query: Use a one-dimensional range query to find all candidates with y-coordinates in $[y_l, y_r]$.

4.4.1 1d-range-search

Example (1d-range-query): Search for $D = [9, 37]$



4.4.1 1d-range-search

1D-Range-Search: Search for all leaves in a balanced, binary search tree whose key lies in $[x_l, x_r]$.

Idea: Search for x_l and x_r in the tree simultaneously:

1. Find the node v_{split} where the paths to x_l and x_r split.
2. From v_{split} follow the path to x_l.
   - At a node where this path turns left, report all leaves of the right sub-tree.
3. From v_{split} follow the path to x_r.
   - At a node where this path turns right, report all leaves of the left sub-tree.

4.4.1 1d-range-search

Algorithm 3: FindSplitNode(tree T, left bound x_l, right bound x_r)

Output: Node where paths to x_l and x_r splits, or the leaf where both paths end.

v = Root(T);
while (v is not a leaf **and** ($x_l > x_v$ **or** $x_r \leq x_v$)) **do** {
  if ($x_r \leq x_v$) **then** v = LeftChild (v);
  else v = RightChild(v);
}
return v;

4.4.1 1d-range-search

Algorithm 4: 1dRangeSearch(tree T, left bound x_l, right bound x_r)

Output: All point in T that lie in the interval $[x_l, x_r]$

```
v_split = FindSplitNode(T, x_l, x_r);
if (v_split is a leaf) then Return the point in v_split,
                            if necessary;
else {
  v = LeftChild(v_split);
  while (v is not a leaf) do {
    if(x_l ≤ x_v) then {
           ReportSubtree(RightChild(v));
           v = LeftChild (v);
    } else v = RightChild(v);
  }
  Return the point in v, if necessary;
  // Analogous for RightChild(v_split) and x_r;

}
```

4.4.1 1d-range-search

Proposition 1

A one-dimensional range search can be computed using $O(n)$ storage in $O(n \log n)$ for the pre-processing and $O(R + \log n)$ for the range search, where R is the number of points, that lie in the query range.

Proof:
- **Pre-processing:** Store the 1d-point sequence in a balanced, binary search tree (e.g. AVL-tree)
  - Storage: $O(n)$
  - Construction time: $O(n \log n)$
- **Range query**:
  - To compute v_{split} and to follow the path to x_l and x_r takes $O(\log n)$, because the time spend in each node is $O(1)$ and the tree is balanced.
  - `ReportSubtree` is linear in the reported points which is in total $O(R)$.
  - Total: $O(R + \log n)$.

□

4.4 Range-trees

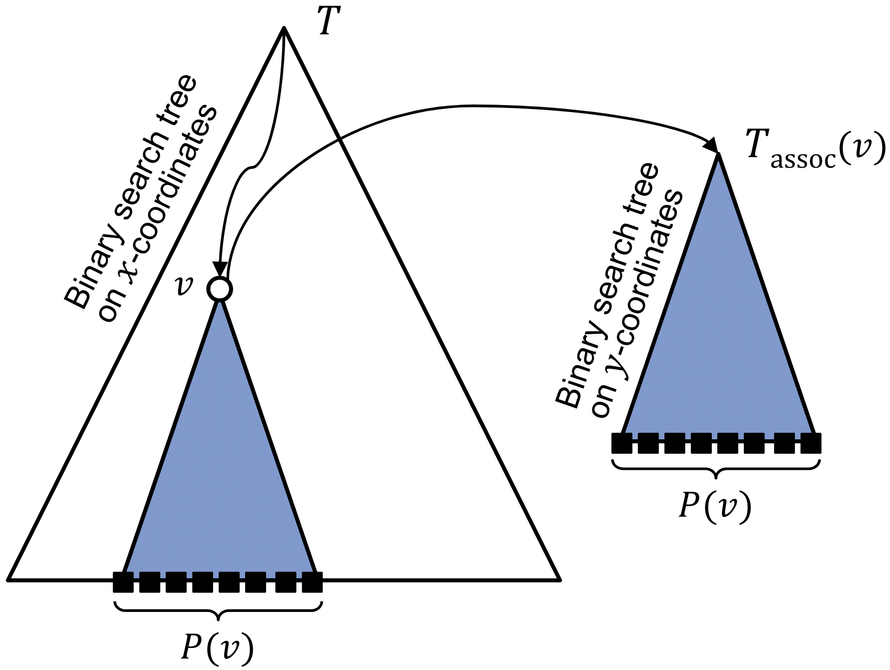
2D-Range-Search: Search for all leaves in a range tree whose key lies in $[x_l, x_r] \times [y_l, y_r]$.

- First level data structure: Balanced, binary search tree for the x-coordinates of the points.
- Every sub-tree rooted at a node v represents a subset of the points, the so-called canonical subset $P(v)$.
- For the range query we need only those points in the canonical subsets whose y-coordinates are in $[y_l, y_r]$.
- Second level data structure: Balanced, binary search tree for the y-coordinates of the points in the canonical subsets.

4.4 Range-trees

Range-tree

- Balanced, binary search tree for the x-coordinates of the points.
- At every inner node v store a balanced, binary search tree for the y-coordinates of the points in $P(v)$.
  - This is the so-called associated structure $T_{\text{assoc}}(v)$.
  - The leaves of $T_{\text{assoc}}(v)$ hold the points.



4.4 Range-trees

Proposition 2

A range tree for n 2d points uses $O(n \log n)$ meomory and can be constructed in $O(n \log n)$.

Proof:

- Storage:
  - At each level every point is stored in exactly one associated structure, which uses linear memory.
  - Thus, at every level $O(n)$ meomory is used in total for all associated structures.
  - Because there are $O(\log n)$ levels, $O(n \log n)$ of memory is used.
- Construction
  - To build a range-tree pre-compute two lists of the points sorted by x- and y-coordinates.
  - From these two lists build the search trees bottom up in $O(n \log n)$.

4.4 Range-trees

Algorithm 5: 2dRangeSearch(tree T, query range $[x_l,x_r]\times[y_l,y_r]$)

Output: All point in T that lie in the query range $[x_l,x_r]\times[y_l,y_r]$

```
v_split = FindSplitNode(T,x_l,x_r);
if (v_split is a leaf) then Return the point in v_split,
                        if necessary;
else {
  v = LeftChild(v_split);
  while (v is not a leaf) do {
    if(x_l≤x_v) then {
          1dRangeSearch(T_assoc(RightChild(v)),y_l,y_r);
          v = LeftChild (v);
    } else v = RightChild(v);
  }
  Return the point in v, if necessary;
  // Analogous for RightChild(v_split) and x_r;
⋮
⋮ }
```

4.4 Range-trees

Proposition 3
A query in a range tree of n 2d points takes $O(R + \log^2 n)$ time, where R is the number of points, that lie in the query range.

Proof:

- Every call of `1DRangeSearch` at a node v takes $O(R_v + \log n)$, where R_v is the number of points in $D \cap P(v)$, i.e. $\Sigma R_v = R$.
- Hence, the total time spend is in `1DRangeSearch`
$$\Sigma O(R_v + \log n),$$
where the summation is over all nodes v that are visited on the search path to x_l and x_r.
- Because the search paths to x_l and x_r have length $O(\log n)$, the total run time is $O(R + \log^2 n)$.

Higher dimensions

- Construct a k-dimensional range-tree recursively, i.e. the second level data structure is a $(k-1)$-dimensional range-tree.

- Run time and storage in k dimensions
  - Construction: $O(n \log^{k-1} n)$
  - Range search: $O(R + \log^{k} n)$
  - Storage: $O(n \log^{k-1} n)$