An Introduction To Range Searching

Jan Vahrenhold

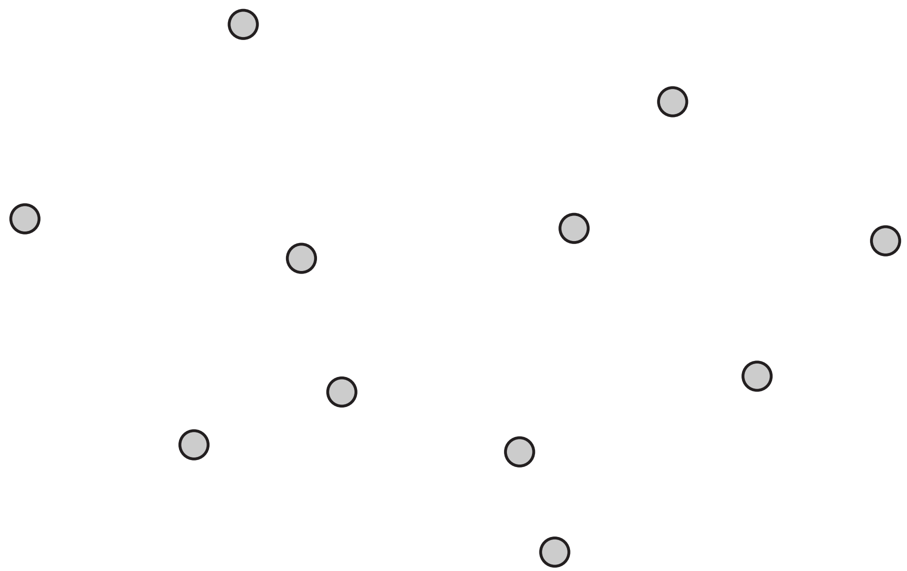
Department of Computer Science
Westfälische Wilhelms-Universität Münster, Germany.

Overview

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook

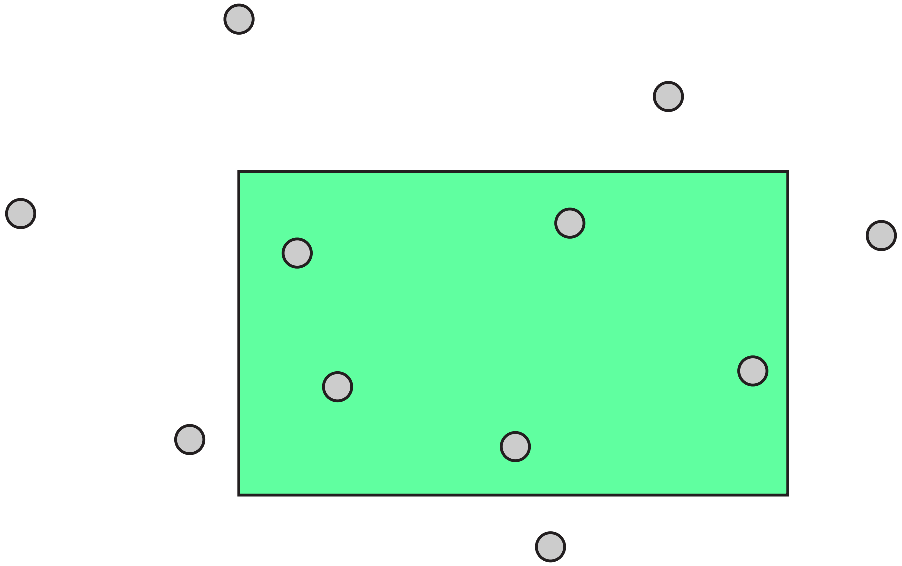
Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.



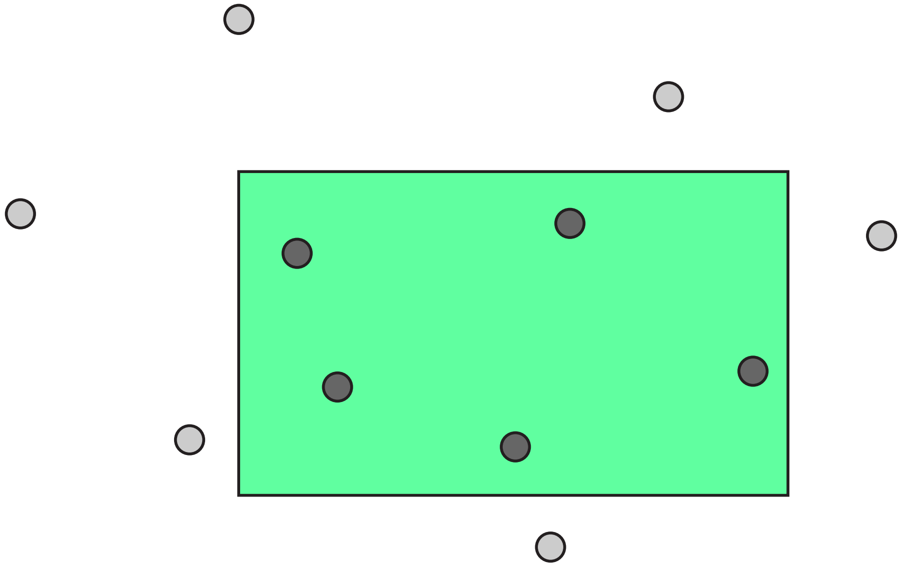
Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.



Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

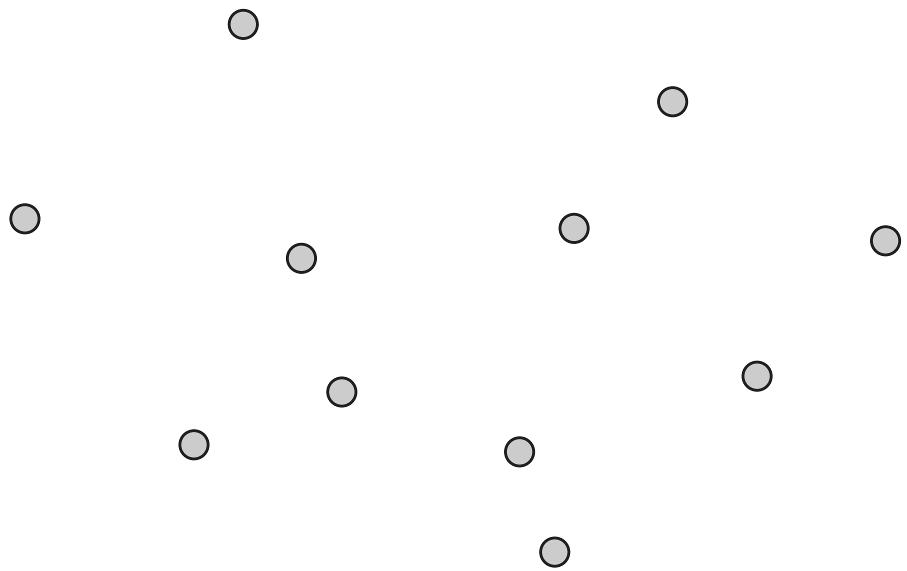
Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.



Problem Setting

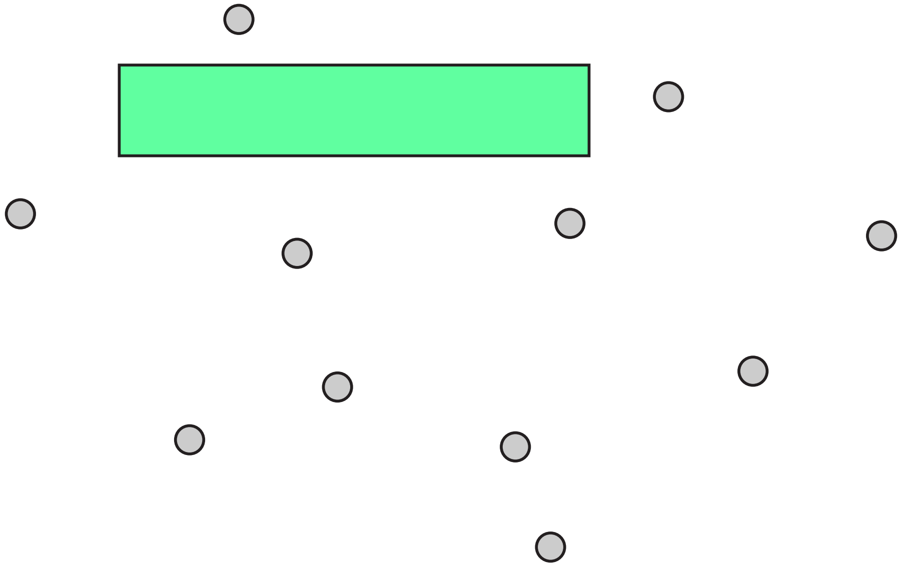
Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.



Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.



Problem Setting

Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.

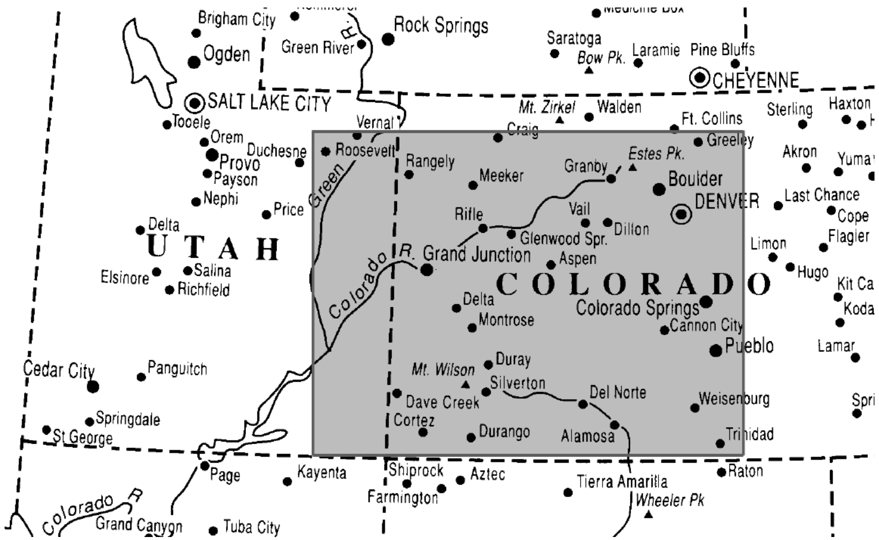
Applications: Geographic Information Systems; Databases having relations in which the keys can be totally ordered.

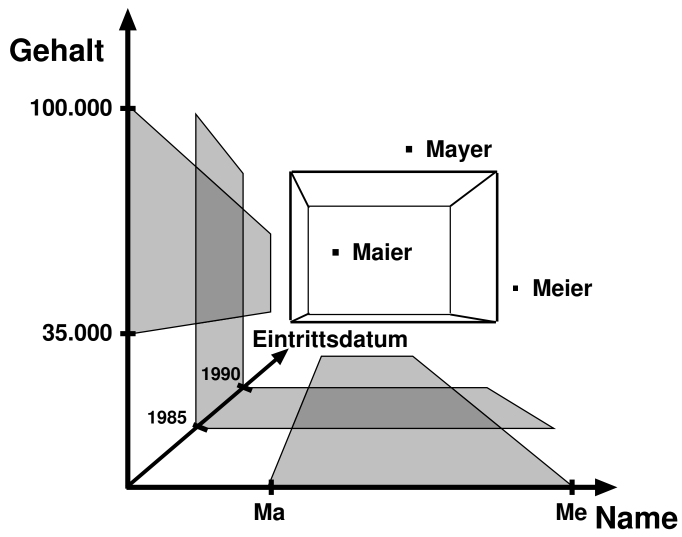
Problem Setting

Given: Collection $\mathcal{S}$ of n points in d dimensions ($\mathcal{S} \subset \mathbb{R}^d$).

Wanted: Algorithm for *efficiently* reporting all k points in $\mathcal{S}$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.

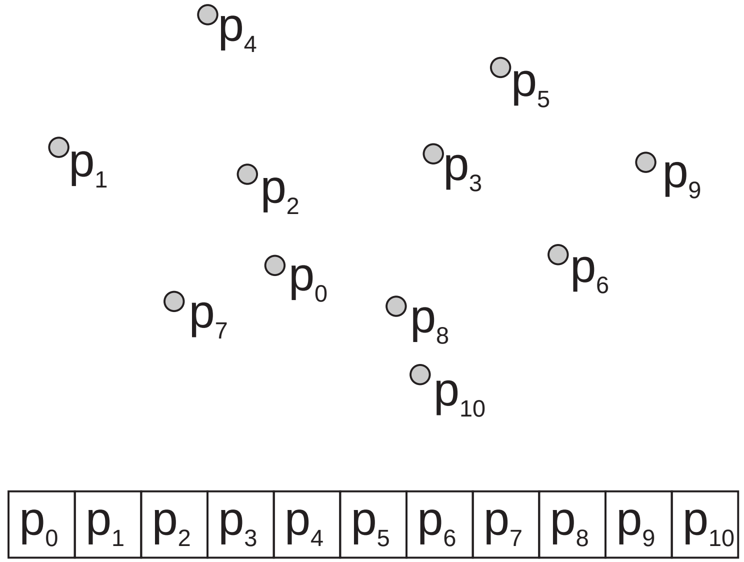
Applications: Geographic Information Systems; Databases having relations in which the keys can be totally ordered.





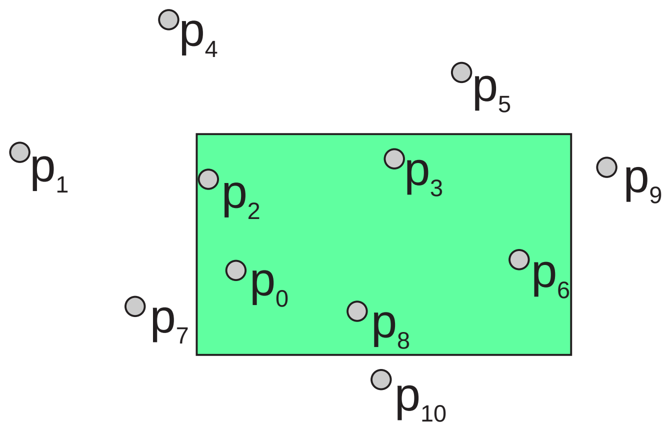
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



A First Approach

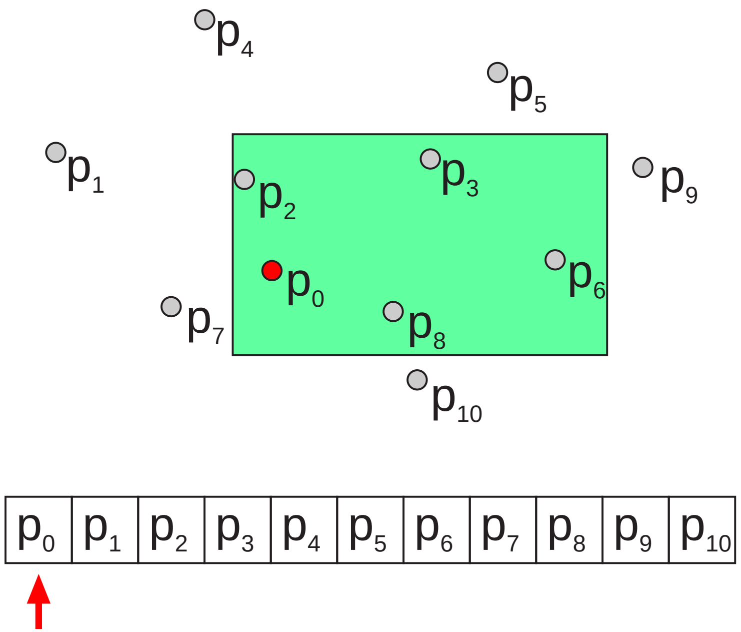
- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



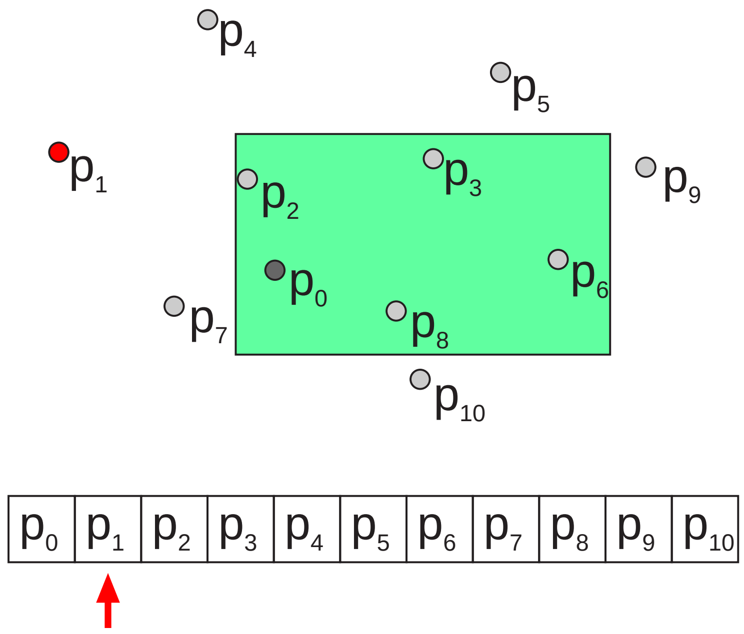
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

A First Approach

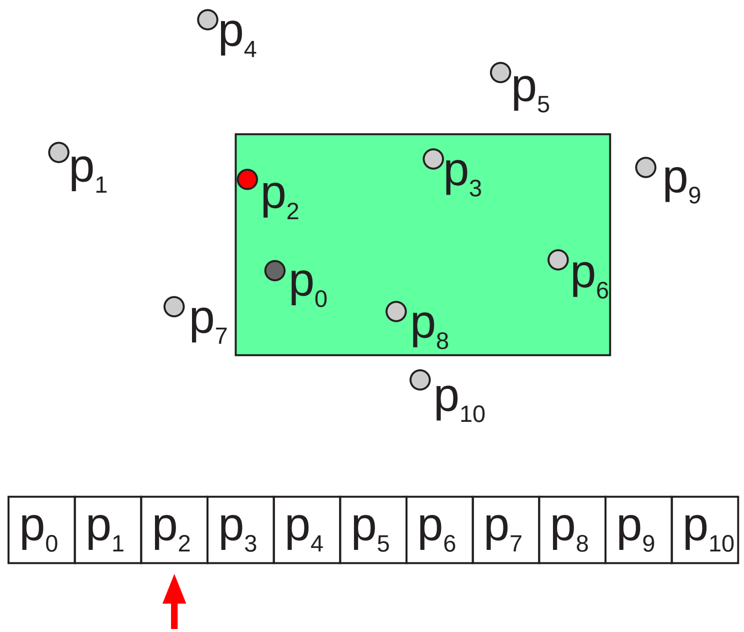
- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.

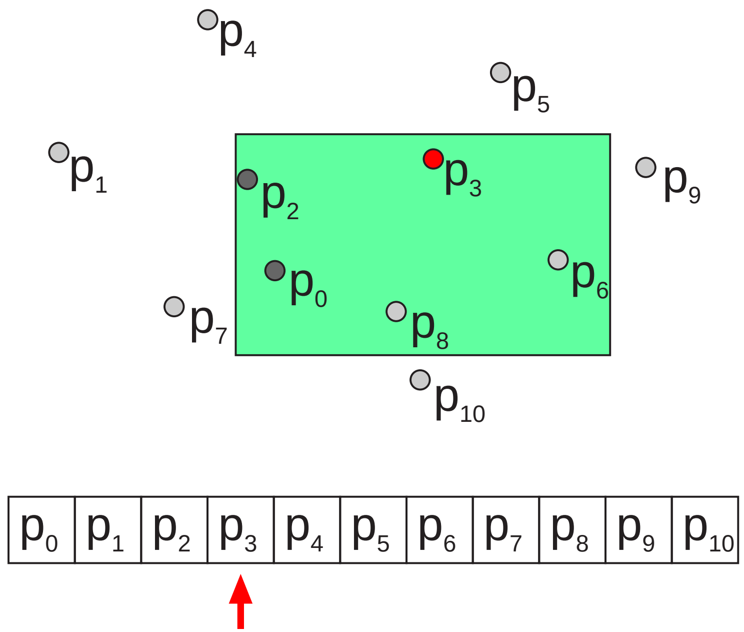


- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



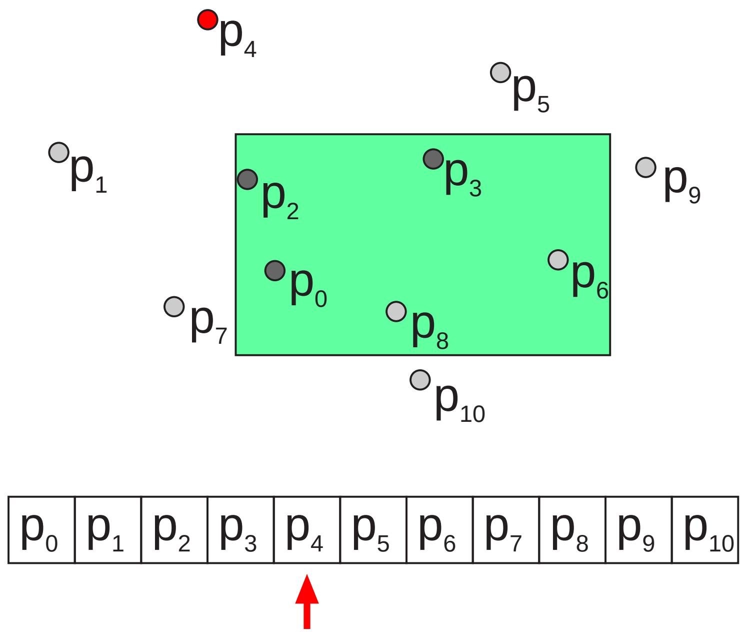
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



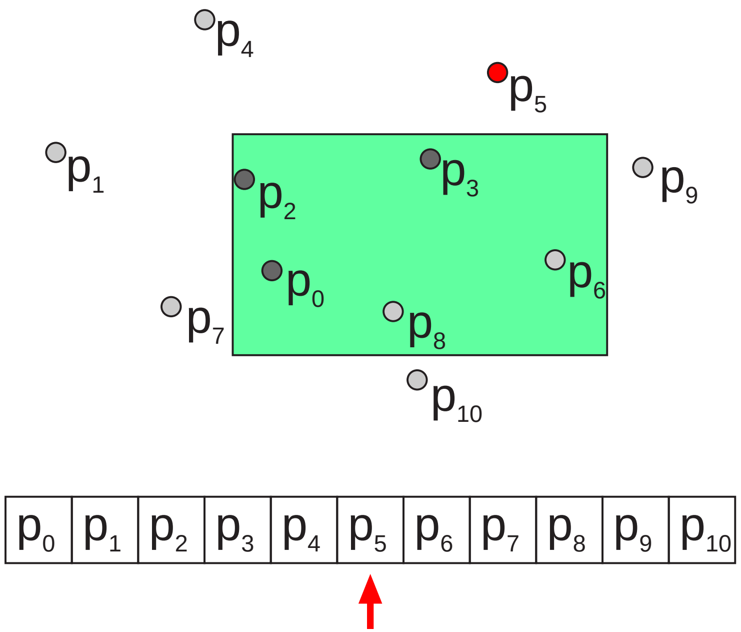
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



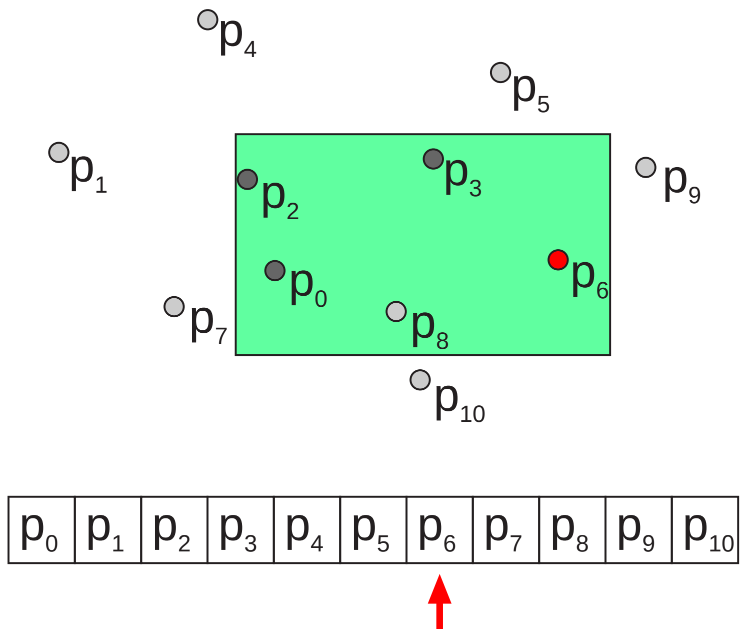
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.

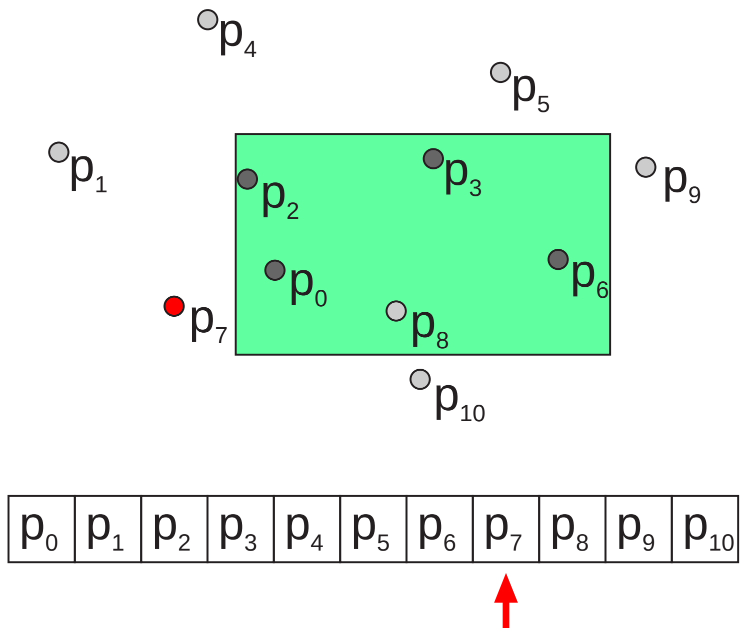


A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.

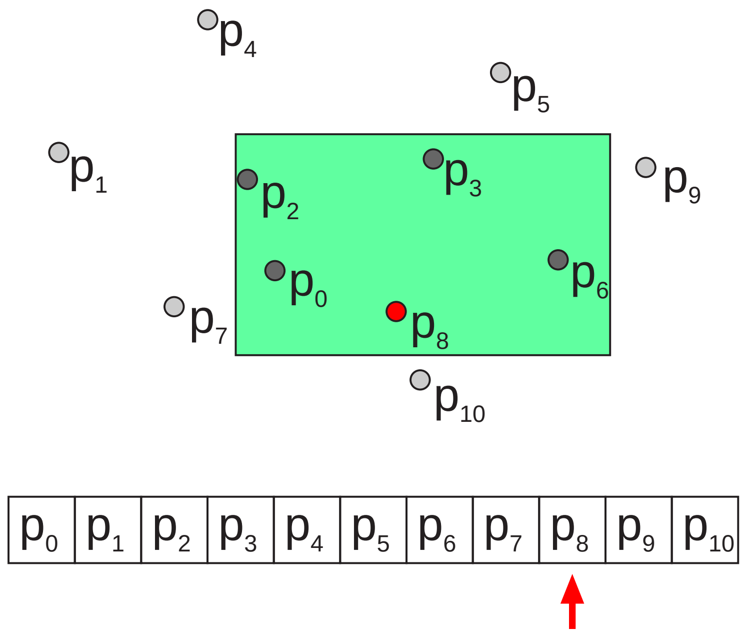


- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



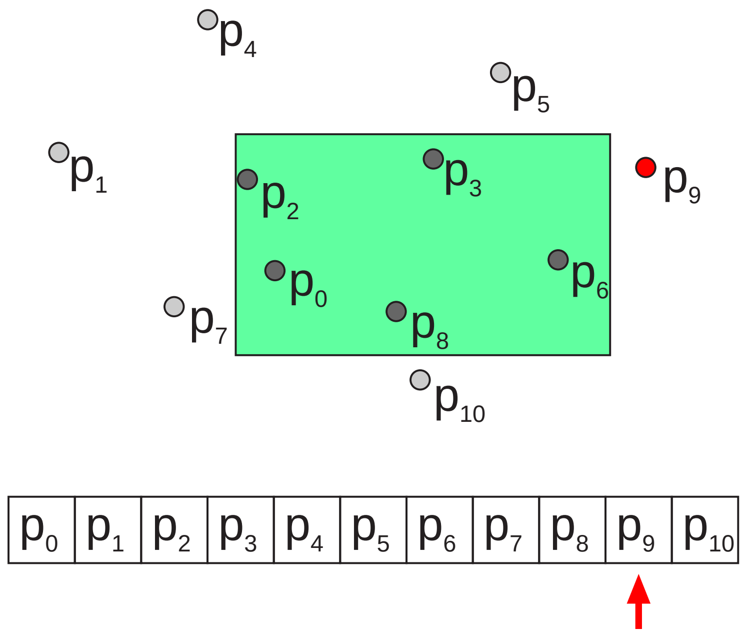
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



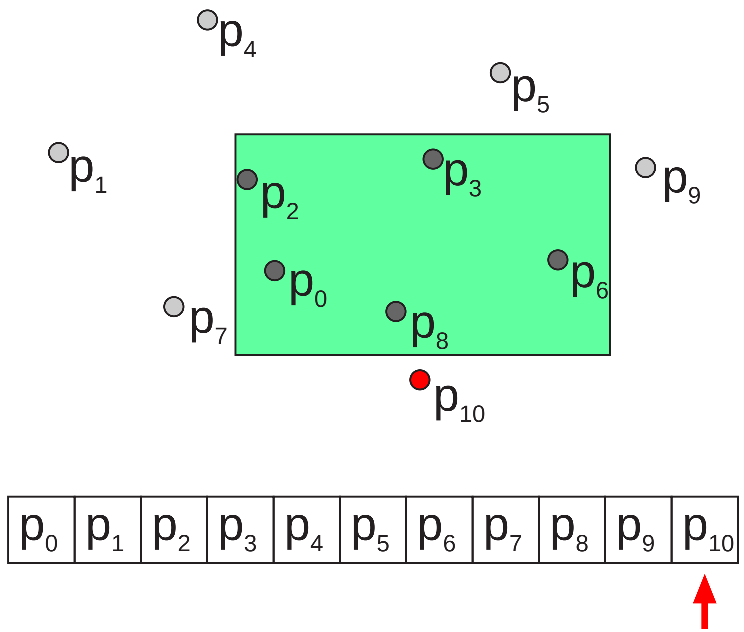
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



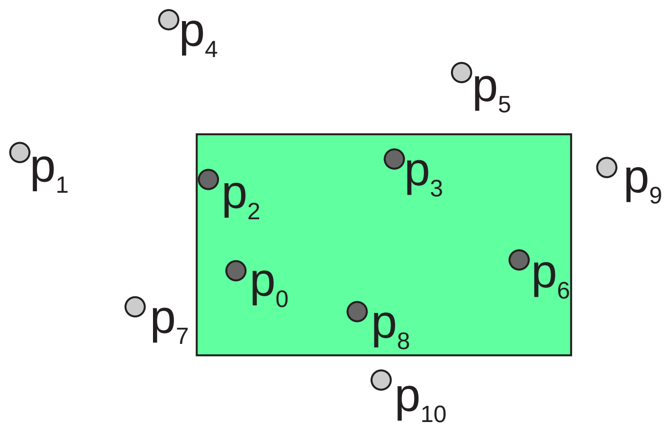
A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



A First Approach

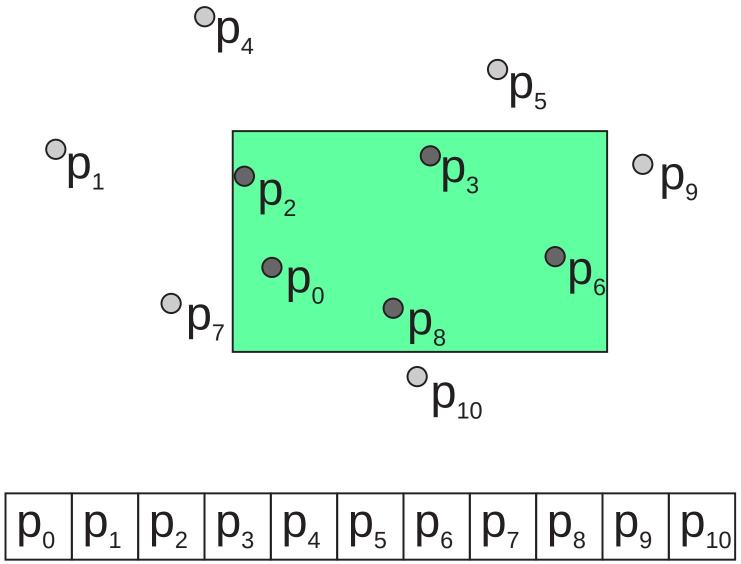
- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

A First Approach

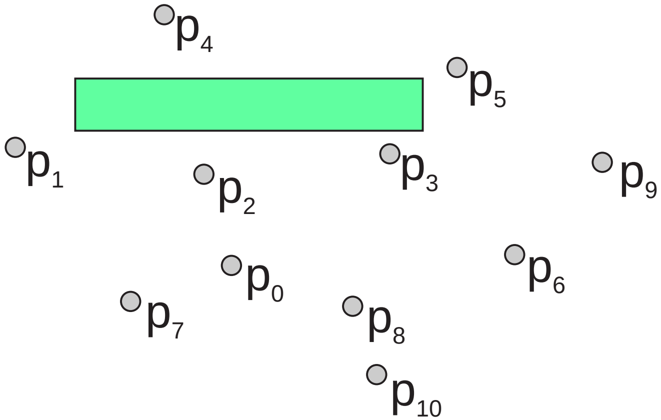
- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



- Need to scan the whole array, regardless of how many points are reported. Complexity: $\Theta(n)$ time and space.

A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.

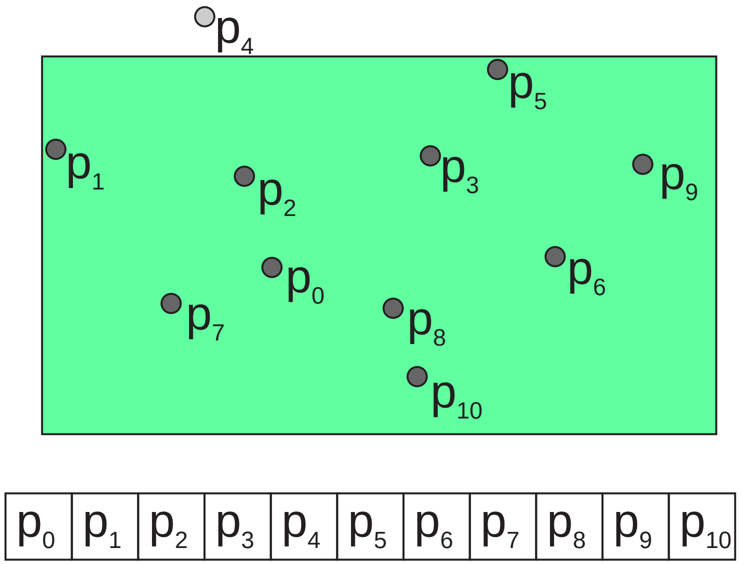


p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

- Need to scan the whole array, regardless of how many points are reported. Complexity: $\Theta(n)$ time and space.

A First Approach

- Assume that $\mathcal{S} = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each p_i whether $p_i \in D$.



- Need to scan the whole array, regardless of how many points are reported. Complexity: $\Theta(n)$ time and space.

- Change the model to also include k (the number of points reported) as a parameter.
  - Algorithm on previous slide has complexity $\mathcal{O}(n+k) = \mathcal{O}(n)$.

- Time complexity: preprocessing time $\Leftrightarrow$ query time
- Can disregard preprocessing time for many applications (one-time operation).

- Query time composed of two components:
  - Search time: Time to locate the first element to be reported.
  - Retrieval time: Time to fetch and report all k elements to be reported.
- Space requirement (lower bound for preprocessing time).

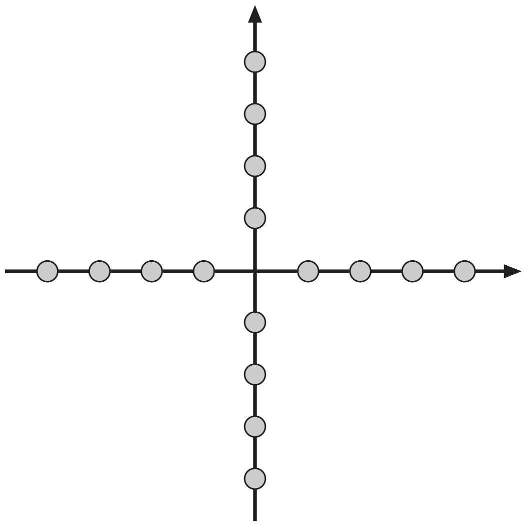
Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).

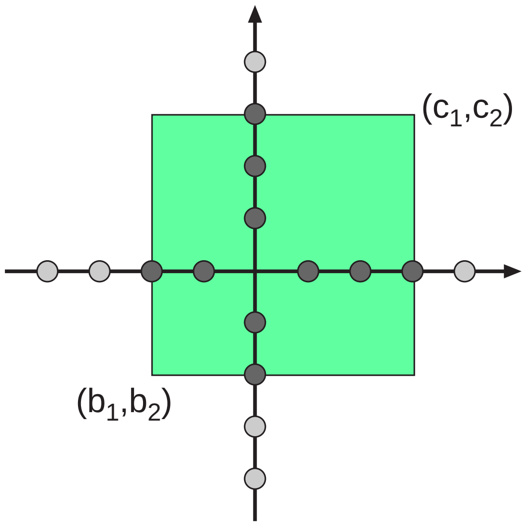
Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)$ $2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)$ $2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.

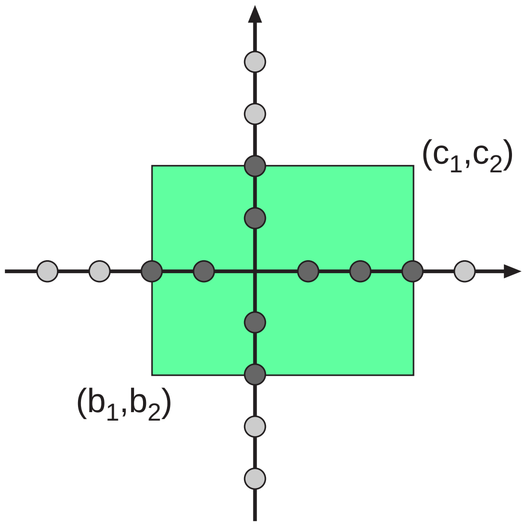


- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)$ $2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.

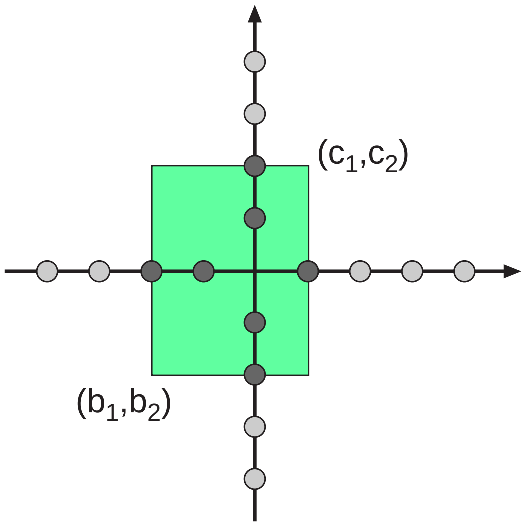


Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)\ 2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.

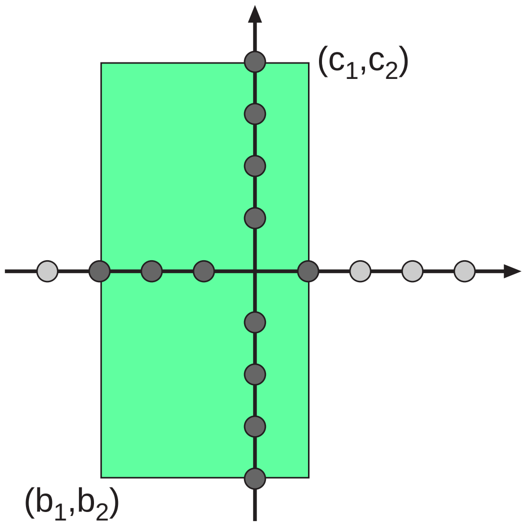


- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)$ $2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.



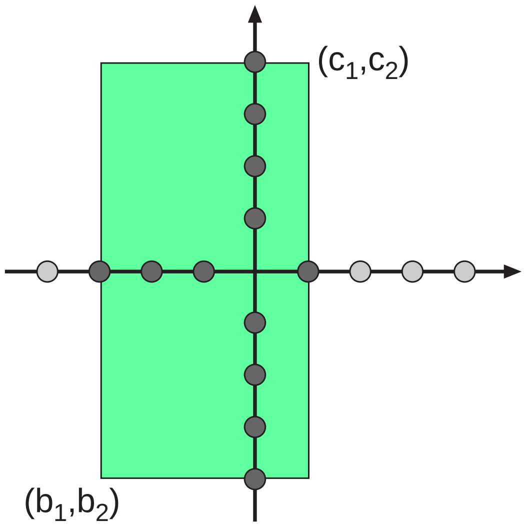
Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)\ 2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.
  - Query ranges not-empty, each produces a different answer.



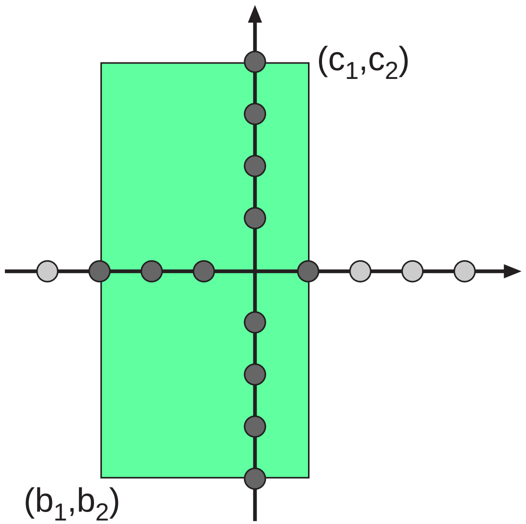
Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)\ 2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.
  - Query ranges not-empty, each produces a different answer.
  - Overall: $a^{2d} = (n/(2d))^{2d}$ different answers.



Lower Bounds [Bentley & Maurer, 1980]

- Parameters: n points, k points reported, d dimensions.
- Space requirement: $\Omega(n)$.
- Retrieval time: $\Omega(k)$.
- Search time: Using binary decision tree ($\rightarrow$ sorting lower bound).
- Lower bound construction:
  - $(n =)\ 2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
  - $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.
  - Query ranges not-empty, each produces a different answer.
  - Overall: $a^{2d} = (n/(2d))^{2d}$ different answers.

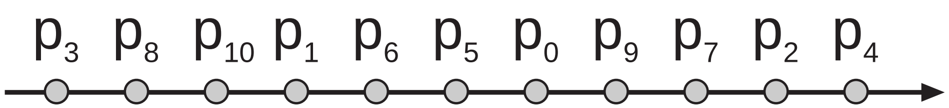


  - Depth of decision tree: $\Omega\left(\log (n/(2d))^{2d}\right) = \Omega(d \cdot \log n)$.
  - Lower bound not tight for all d.

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook

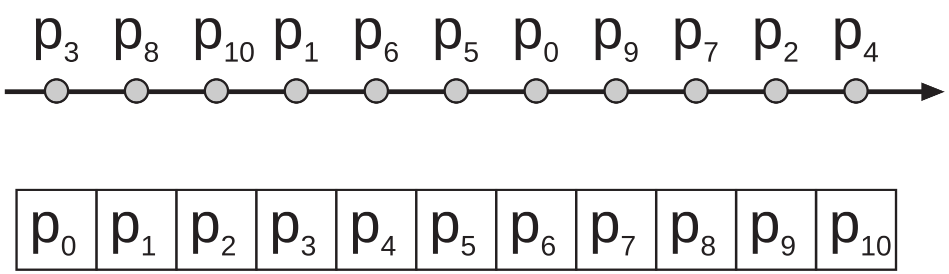
One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.



One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

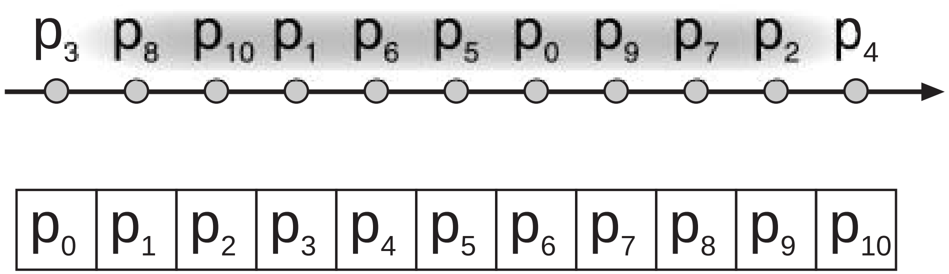


One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.

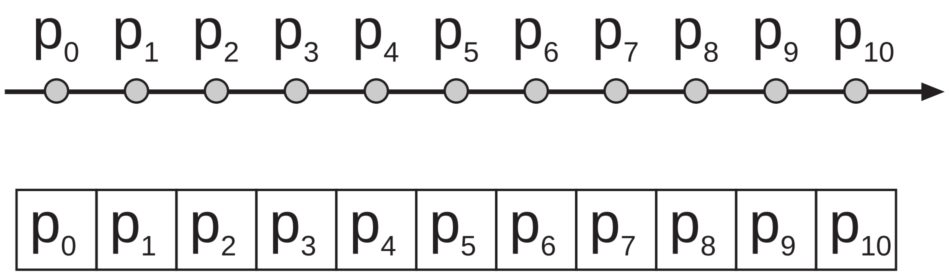


One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

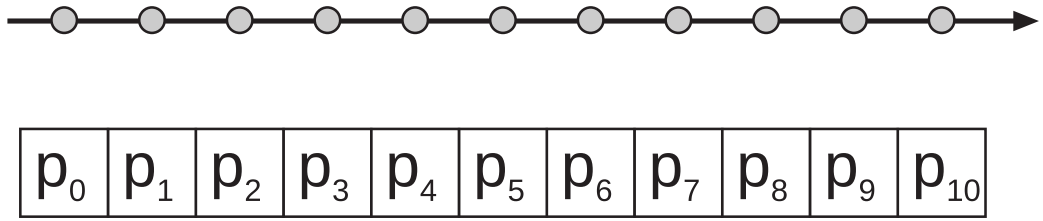
- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

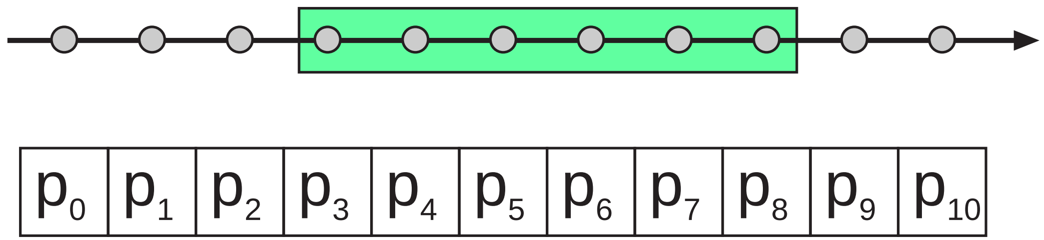
- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

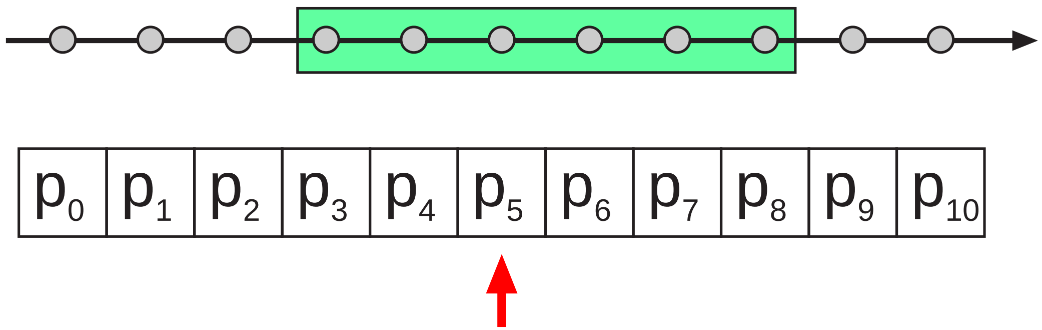
- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.

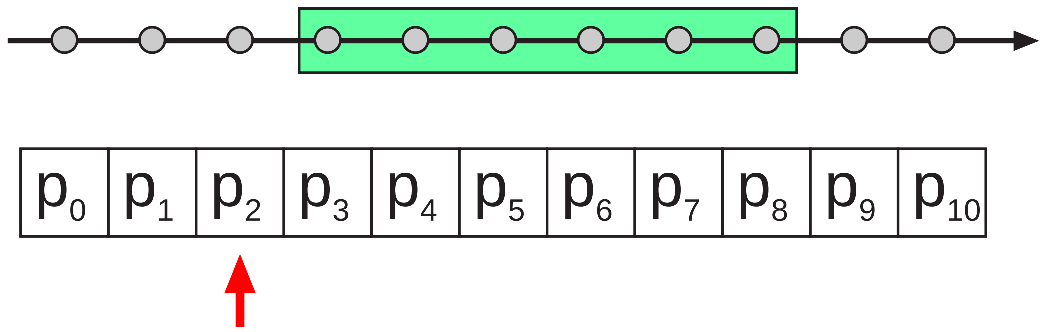


Query: Binary search for smallest $p_i \geq x_1 \ldots$

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.

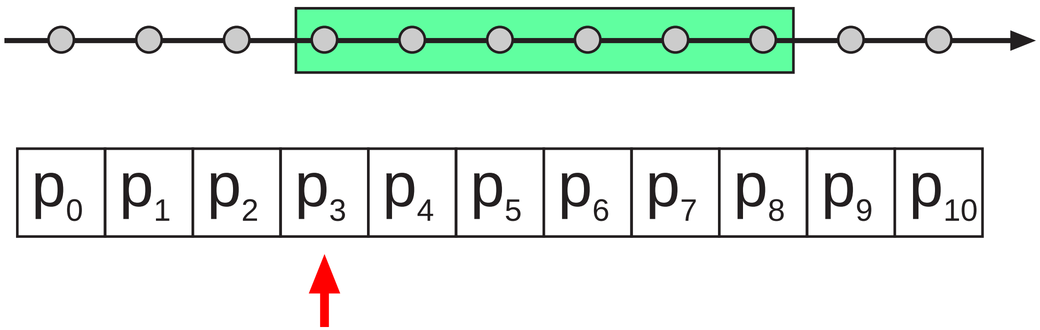


Query: Binary search for smallest $p_i \geq x_1 \ldots$

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.

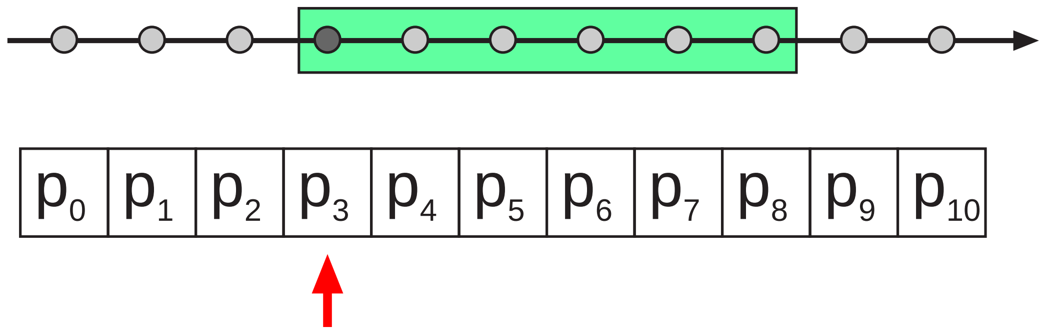


Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

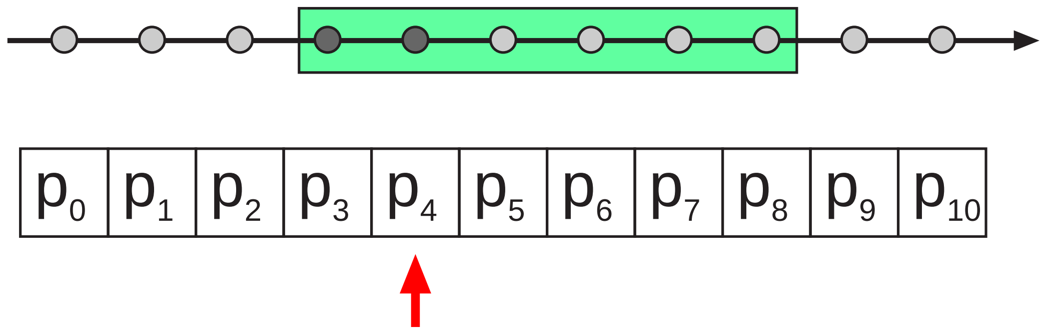
$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



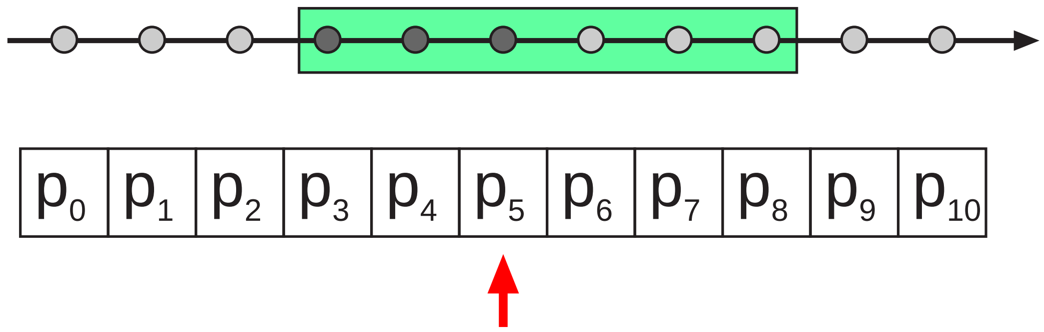
Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

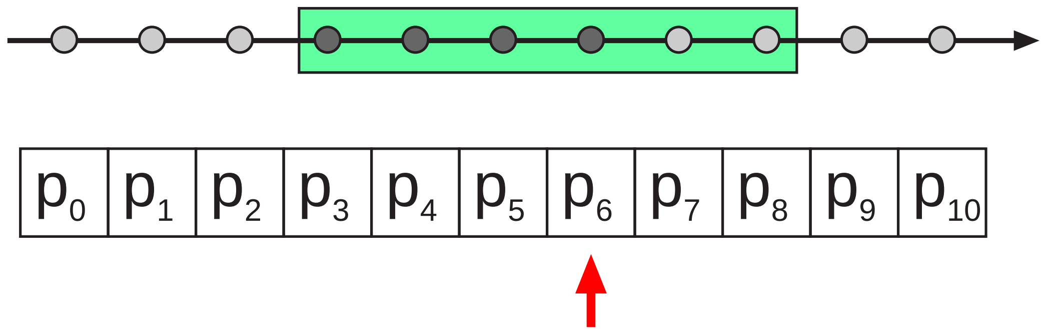
$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



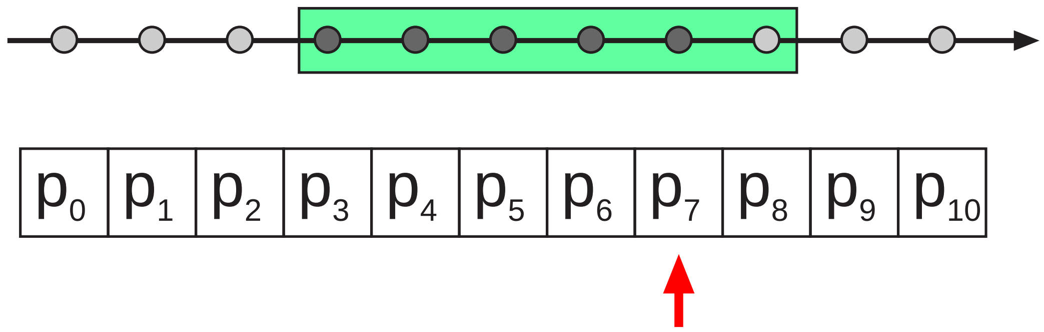
Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

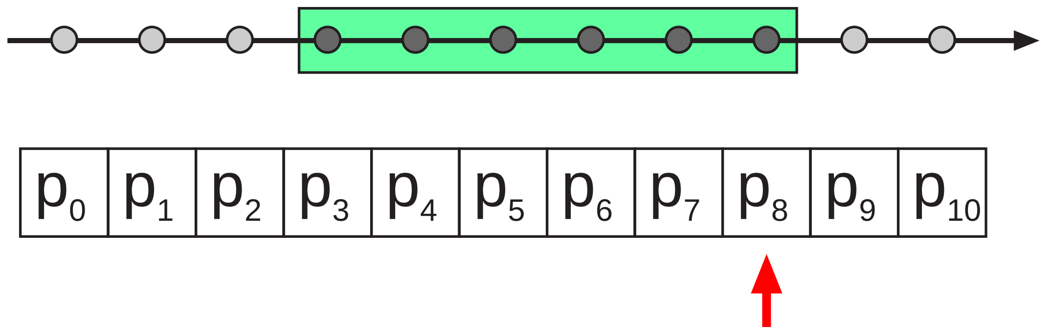
$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

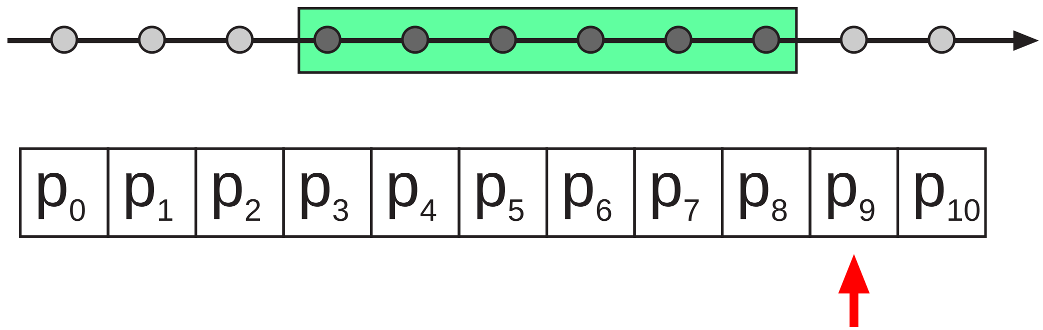
$\ldots$ scan forward until first $p_i < x_2$ (or end of array).

One-Dimensional Range Searching

- Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:

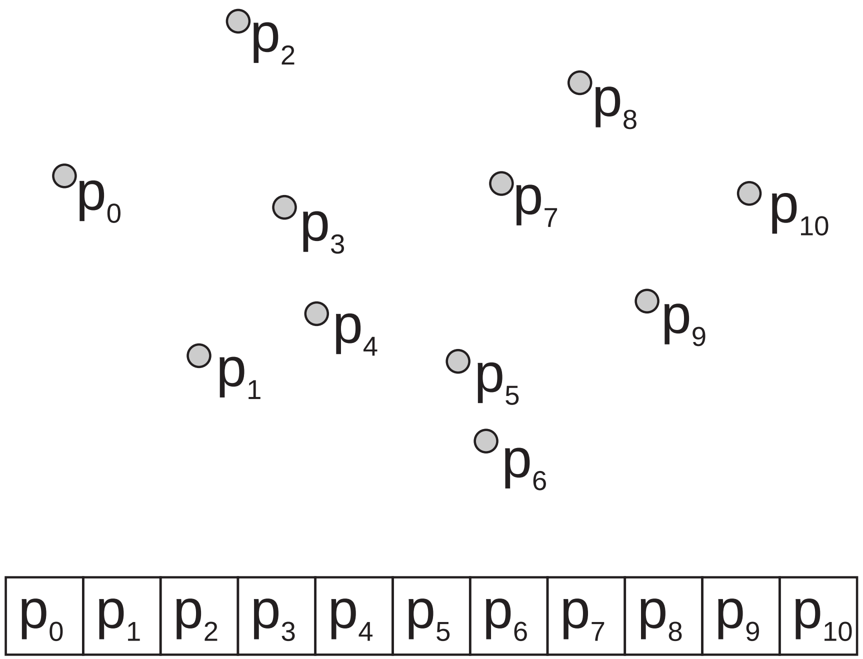
- Sort the points, e.g., using *heapsort* in $\mathcal{O}(n \log_2 n)$ time.



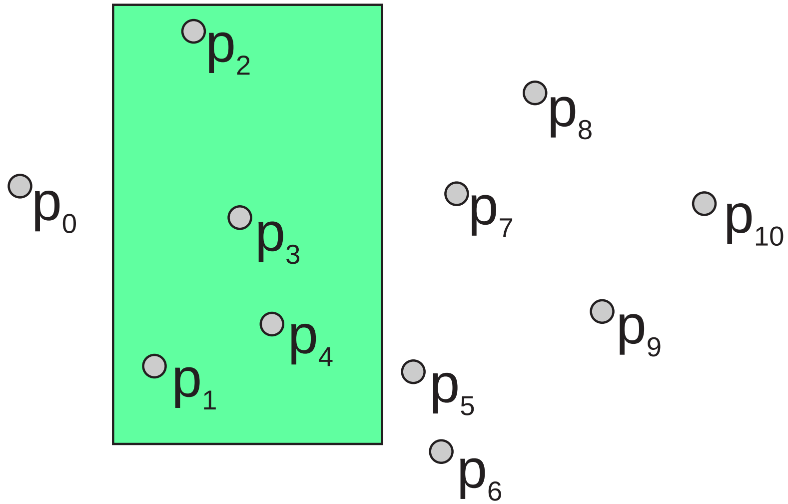
Query: Binary search for smallest $p_i \geq x_1 \ldots$ $\mathcal{O}(\log_2 n)$

$\ldots$ scan forward until first $p_i < x_2$ (or end of array). $\mathcal{O}(k+1)$

Does Sorting Help in Two Dimensions?

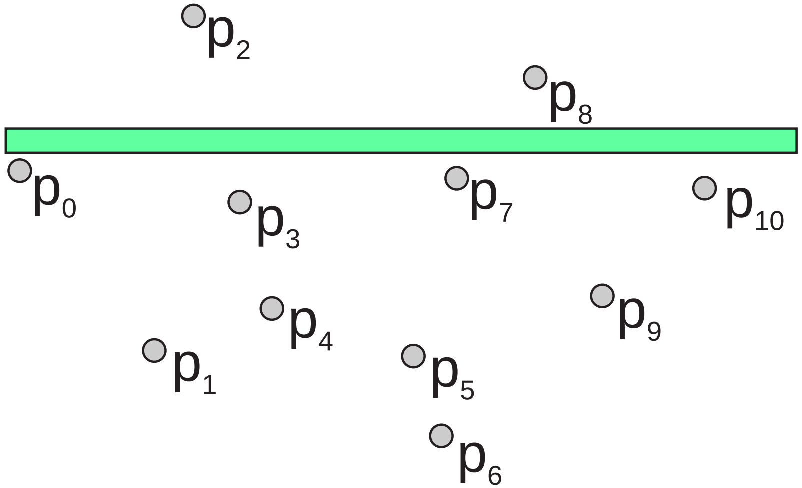


Does Sorting Help in Two Dimensions?

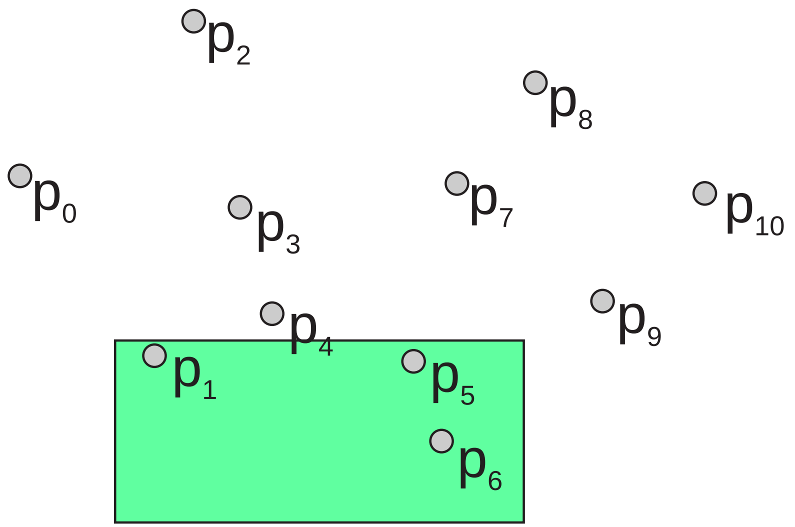


p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

Does Sorting Help in Two Dimensions?

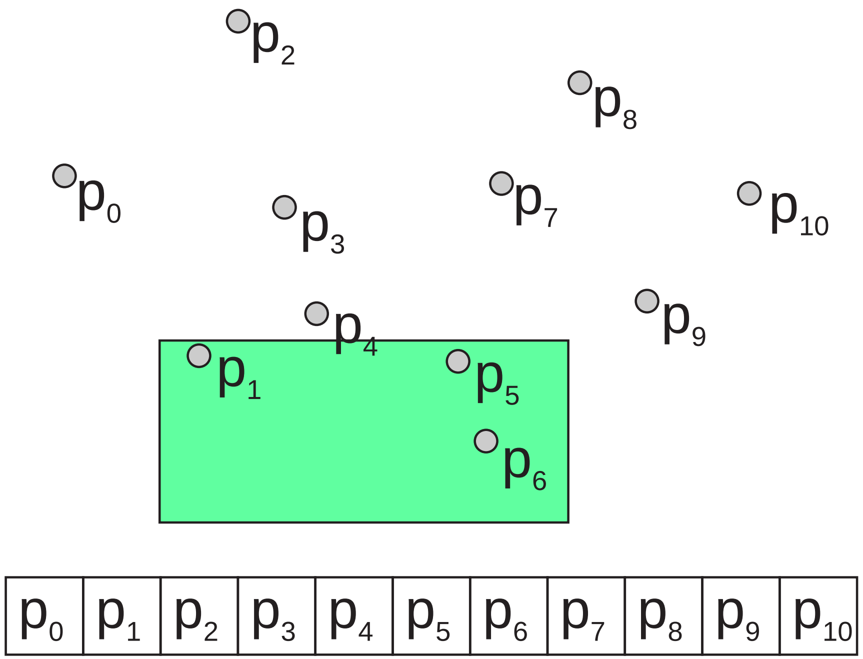


p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}



p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

Does Sorting Help in Two Dimensions?



- There is no total order on points in two dimensions sorting according to which guarantees $\Theta\left(2 \cdot \log_2 n + k\right)$ query time for range searching.

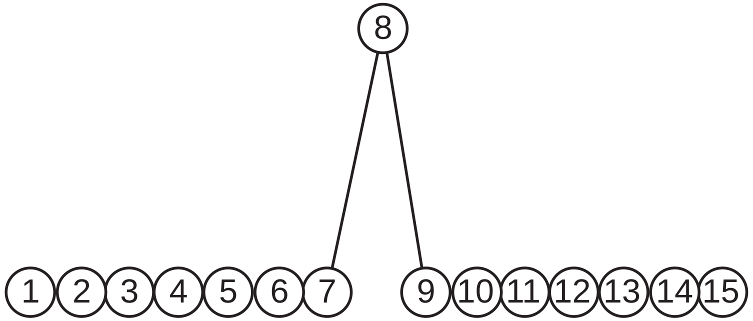
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

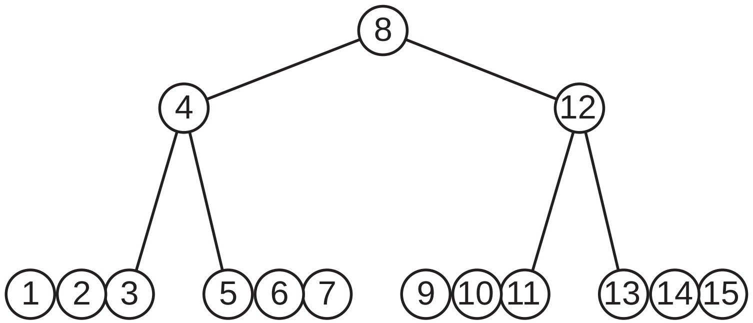
Recap: One-Dimensional Range Searching

- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.

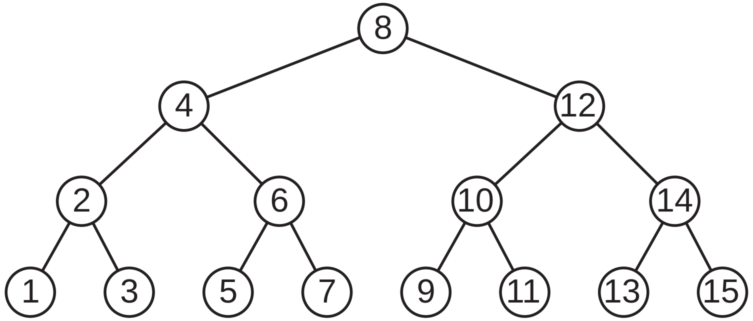
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.



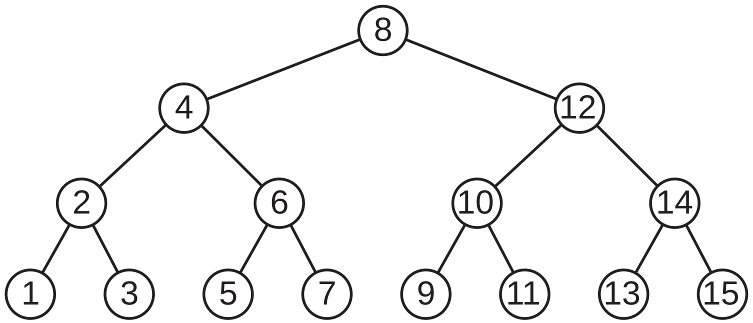
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.



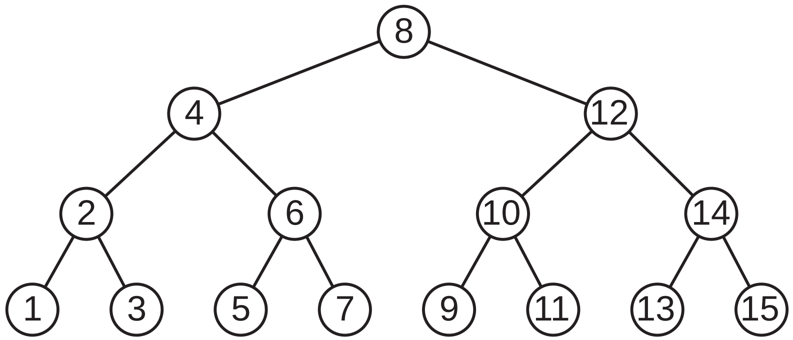
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.



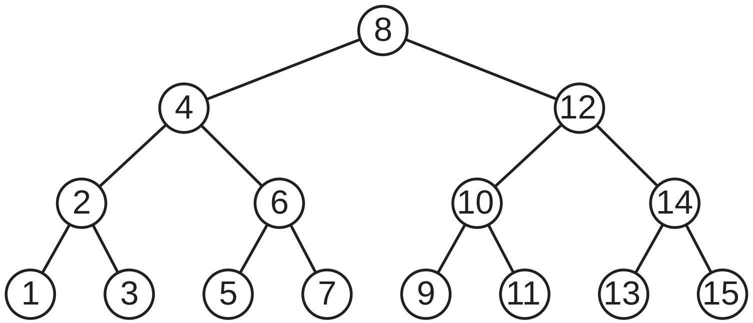
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.



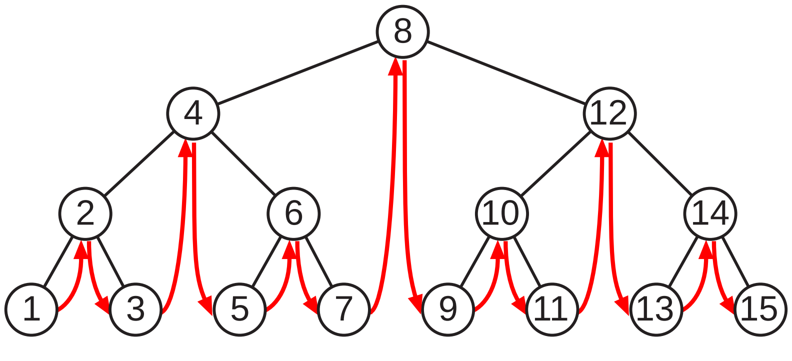
- **Time Complexity:**
  - Preprocessing time: $\mathcal{O}(n \log n)$
  - Query time: $\mathcal{O}(\log n + k)$
- **Space Complexity:** $\mathcal{O}(n)$.
- `Insert`s/`Delete`s possible.



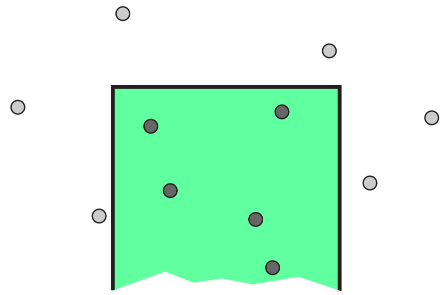
- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.



- **Time Complexity:**
  - Preprocessing time: $\mathcal{O}(n \log n)$
  - Query time: $\mathcal{O}(\log n + k)$
- **Space Complexity:** $\mathcal{O}(n)$.
- `Inserts`/`Deletes` possible.

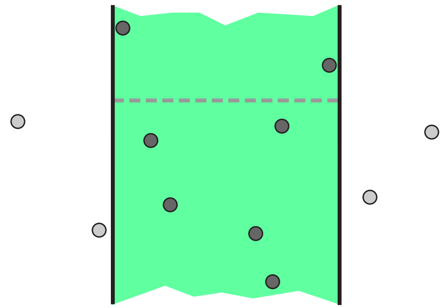


Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.



Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

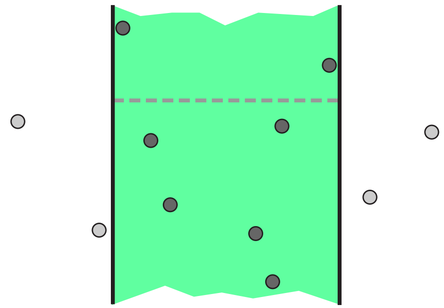


Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times \,]-\infty, y]$.

Look at two subproblems:

- Report all points in $[x_1, x_2] \times \mathbb{R}$

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.



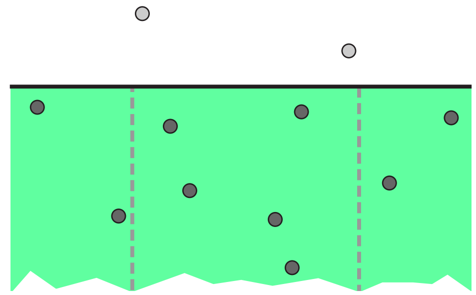
Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.

Look at two subproblems:

- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.

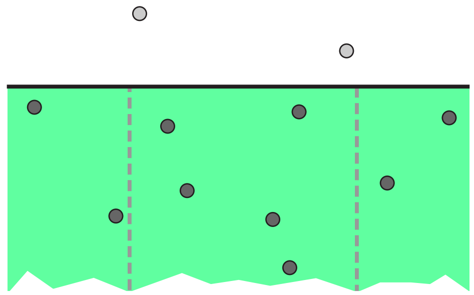


Look at two subproblems:

- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.
- Report all points in $\mathbb{R} \times]-\infty, y]$

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.

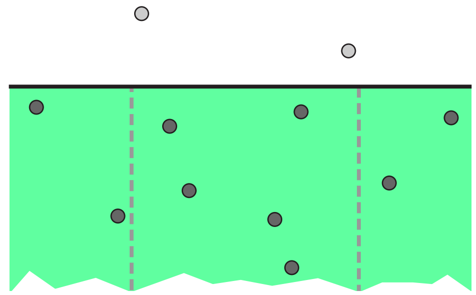


Look at two subproblems:

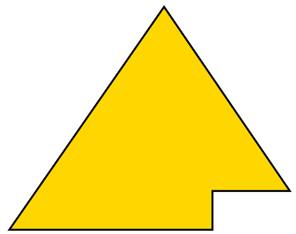
- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.
- Report all points in $\mathbb{R} \times]-\infty, y]$ using, e.g., a heap:

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.



Look at two subproblems:

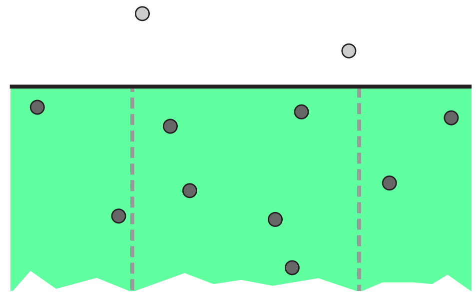


- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.
- Report all points in $\mathbb{R} \times]-\infty, y]$ using, e.g., a heap:
  - Almost complete binary tree.

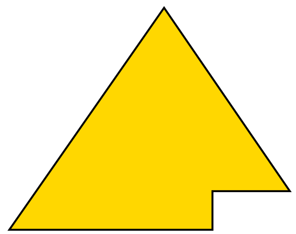
Three-sided (1.5-dim.) Range Searching

Given: Point set $\mathcal{S} = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

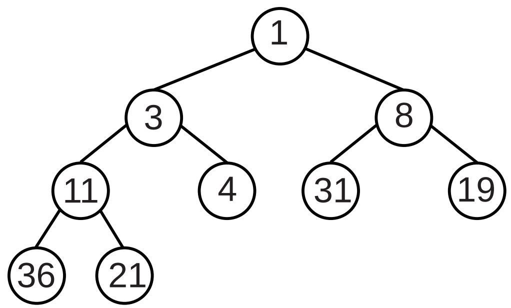
Wanted: Method to efficiently retrieve all $p \in \mathcal{S}$ that, for given (x_1, x_2, y), fall into $[x_1, x_2] \times]-\infty, y]$.



Look at two subproblems:



- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.
- Report all points in $\mathbb{R} \times]-\infty, y]$ using, e.g., a heap:
  - Almost complete binary tree.
  - $\texttt{key}(v) \leq \min\{\texttt{key}(\texttt{LSON}(v)), \texttt{key}(\texttt{RSON}(v))\}$.



Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.

Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.
- No (direct) way of incorporating heap property.

Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.
- No (direct) way of incorporating heap property.

Heap with search tree property:

- Heap not unique.

Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.
- No (direct) way of incorporating heap property.

Heap with search tree property:

- Heap not unique.
- More precisely: Children of a node may be switched.

Combining the best of both worlds(?)

Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.
- No (direct) way of incorporating heap property.

Heap with search tree property:

- Heap not unique.
- More precisely: Children of a node may be switched.

Priority Search Tree:

- Binary tree $\mathcal{H}$ storing a two-dimensional point at each node s.t. the heap property w.r.t. the y-coordinates is fulfilled.

Binary search tree with heap property:

- Binary search tree unique w.r.t. *inorder*-traversal.
- No (direct) way of incorporating heap property.

Heap with search tree property:

- Heap not unique.
- More precisely: Children of a node may be switched.

Priority Search Tree:

- Binary tree $\mathcal{H}$ storing a two-dimensional point at each node s.t. the heap property w.r.t. the y-coordinates is fulfilled.
- Additional requirement: $\forall v \in \mathcal{H} : \exists x_v \in \mathbb{R} :$

$$l \leq x_v < r \quad \forall l \in \texttt{LSUBTREE}(v), \forall r \in \texttt{RSUBTREE}(v).$$

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.
- Else let p_{min} be the point in $\mathcal{S}$ having the minimum y-coordinate.

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.
- Else let p_{min} be the point in $\mathcal{S}$ having the minimum y-coordinate.
- Let x_{mid} be the median of the x-coordinates in $\mathcal{S} \setminus \{p_{\text{min}}\}$.
- Partition $\mathcal{S} \setminus \{p_{\text{min}}\}$:

$$\begin{aligned} \mathcal{S}_{\text{left}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x \leq x_{\text{mid}}\} \\ \mathcal{S}_{\text{right}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x > x_{\text{mid}}\} \end{aligned}$$

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.
- Else let p_{min} be the point in $\mathcal{S}$ having the minimum y-coordinate.
- Let x_{mid} be the median of the x-coordinates in $\mathcal{S} \setminus \{p_{\text{min}}\}$.
- Partition $\mathcal{S} \setminus \{p_{\text{min}}\}$:

$$\begin{aligned} \mathcal{S}_{\text{left}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x \leq x_{\text{mid}}\} \\ \mathcal{S}_{\text{right}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x > x_{\text{mid}}\} \end{aligned}$$

- Construct search tree node v storing x_{mid} and set $p(v) := p_{\text{min}}$.

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.
- Else let p_{min} be the point in $\mathcal{S}$ having the minimum y-coordinate.
- Let x_{mid} be the median of the x-coordinates in $\mathcal{S} \setminus \{p_{\text{min}}\}$.
- Partition $\mathcal{S} \setminus \{p_{\text{min}}\}$:

$$\begin{aligned} \mathcal{S}_{\text{left}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x \leq x_{\text{mid}}\} \\ \mathcal{S}_{\text{right}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x > x_{\text{mid}}\} \end{aligned}$$

- Construct search tree node v storing x_{mid} and set $p(v) := p_{\text{min}}$.
- Recursively compute v's children $\mathcal{H}(\mathcal{S}_{\text{left}})$ and $\mathcal{H}(\mathcal{S}_{\text{right}})$.

Use recursive definition [McCreight, 1985]:

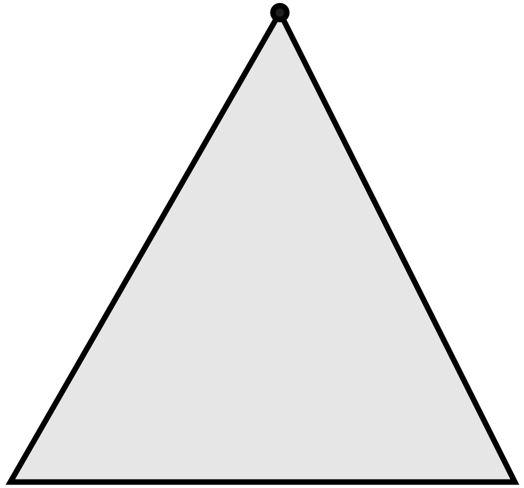
- Build priority search tree $\mathcal{H}(\mathcal{S})$ for a given set $\mathcal{S}$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $\mathcal{S} = \emptyset$, construct $\mathcal{H}(\mathcal{S})$ as an (empty) leaf.
- Else let p_{min} be the point in $\mathcal{S}$ having the minimum y-coordinate.
- Let x_{mid} be the median of the x-coordinates in $\mathcal{S} \setminus \{p_{\text{min}}\}$.
- Partition $\mathcal{S} \setminus \{p_{\text{min}}\}$:

$$\begin{aligned} \mathcal{S}_{\text{left}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x \leq x_{\text{mid}}\} \\ \mathcal{S}_{\text{right}} &:= \{p \in \mathcal{S} \setminus \{p_{\text{min}}\} \mid p.x > x_{\text{mid}}\} \end{aligned}$$

- Construct search tree node v storing x_{mid} and set $p(v) := p_{\text{min}}$.
- Recursively compute v's children $\mathcal{H}(\mathcal{S}_{\text{left}})$ and $\mathcal{H}(\mathcal{S}_{\text{right}})$.
- Complexity: $\mathcal{O}(n)$ space; $\mathcal{O}(n \log n)$ time (why?).

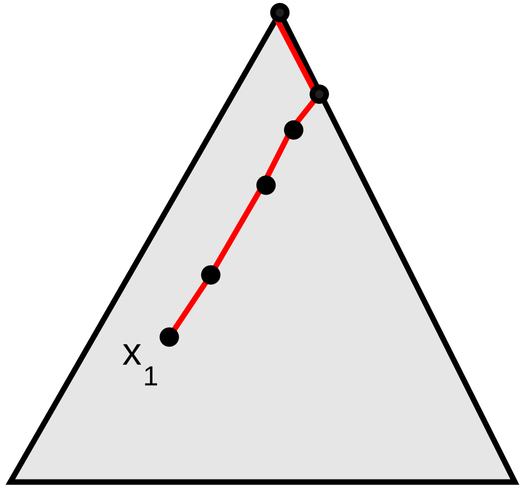
Query range $[x_1, x_2] \times [-\infty, y]$**:**

- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.



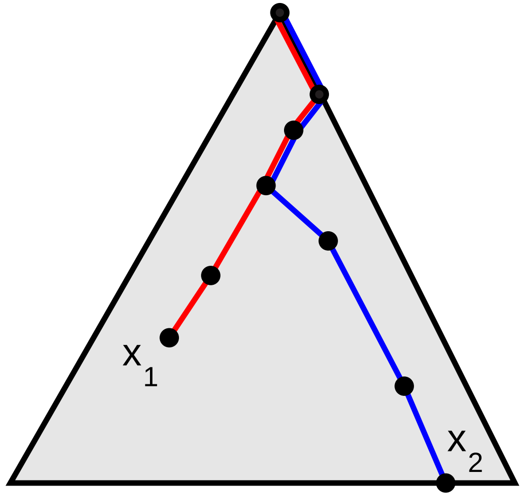
Query range $[x_1, x_2] \times [-\infty, y]$:

- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.



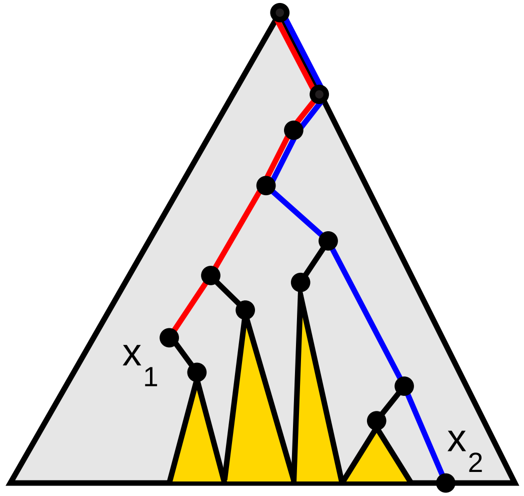
Query range $[x_1, x_2] \times [-\infty, y]$**:**

- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.
- Check all points on these paths.



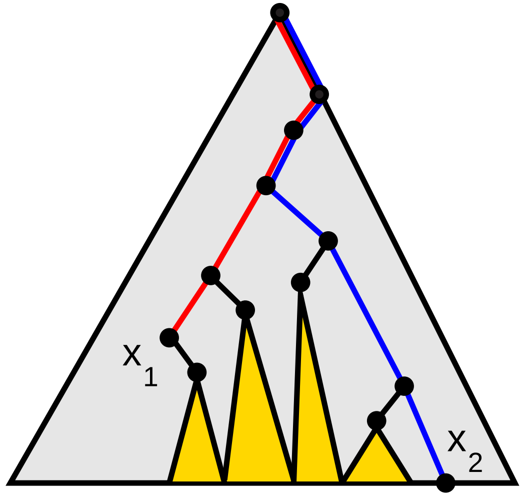
Query range $[x_1, x_2] \times [-\infty, y]$**:**

- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.
- Check all points on these paths.
- All subtrees “embraced” by these paths contain points in $[x_1, x_2] \times \mathbb{R}$.



Querying a priority search tree

Query range $[x_1, x_2] \times [-\infty, y]$**:**



- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.
- Check all points on these paths.
- All subtrees "embraced" by these paths contain points in $[x_1, x_2] \times \mathbb{R}$.
- Query these subtrees a follows:

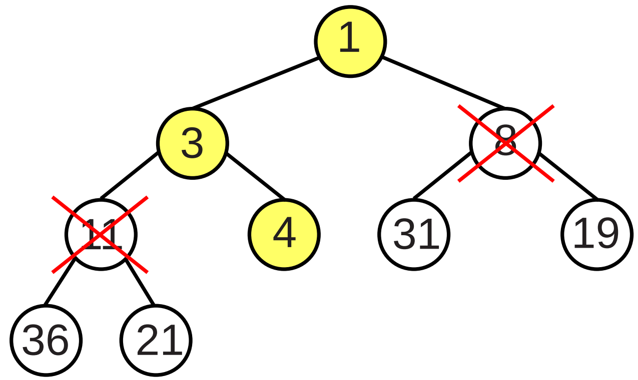
SearchInSubtree(v, y)

    if v not a leaf **and** $p(v).y \leq y$ **then**

        Report $p(v)$;

        SearchInSubtree(LSON$(v), y)$;

        SearchInSubtree(RSON$(v), y)$;

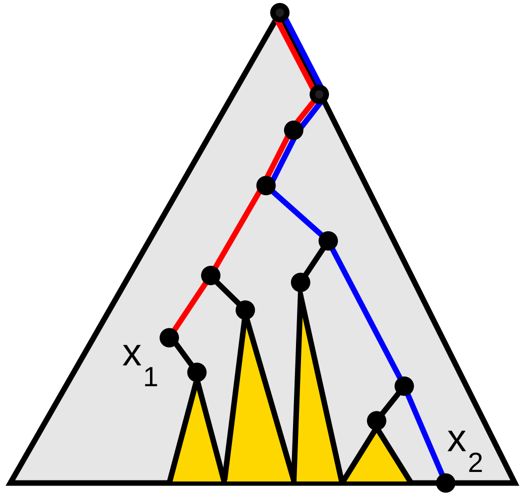


Example for $y = 5$.

Query time: $\mathcal{O}(1 + k_v)$.

Querying a priority search tree

Query range $[x_1, x_2] \times [-\infty, y]$**:**



- Queries for x_1 and x_2 result in two search paths in $\mathcal{H}$.
- Check all points on these paths.
- All subtrees "embraced" by these paths contain points in $[x_1, x_2] \times \mathbb{R}$.
- Query these subtrees a follows:

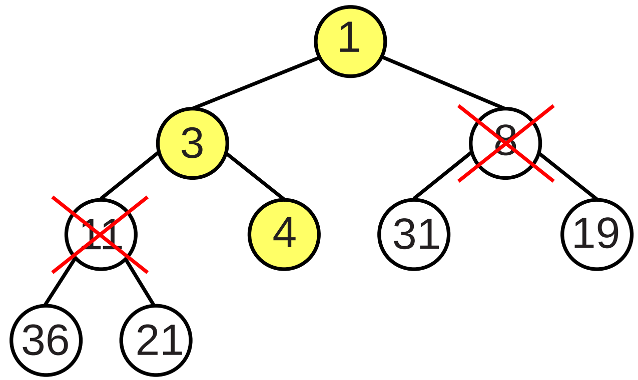
SearchInSubtree(v, y)

if v not a leaf **and** $p(v).y \leq y$ **then**

Report $p(v)$;

SearchInSubtree(LSON$(v), y)$;

SearchInSubtree(RSON$(v), y)$;



Example for $y = 5$.

Query time: $\mathcal{O}(1 + k_v)$.

Missing Components:

- A more detailed description of the query algorithm.
- Proof of correctness.

} $\Rightarrow$ [de Berg et al., 2000]

Theorem 2.1
Priority search trees allow for answering three-sided range queries on points in $\mathbb{R}^2$ with time and space complexities as follows:

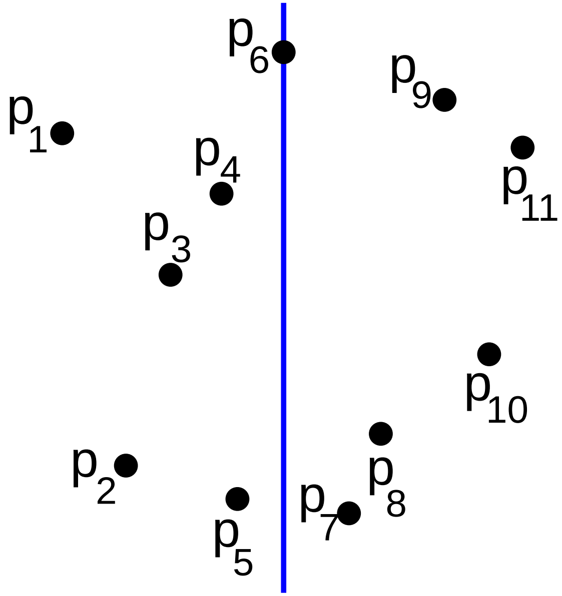
Preprocessing time:	$\Theta(n \log n)$
Query time:	$\mathcal{O}(\log n + k)$
Space requirement:	$\Theta(n)$

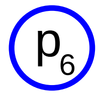
Overview

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook

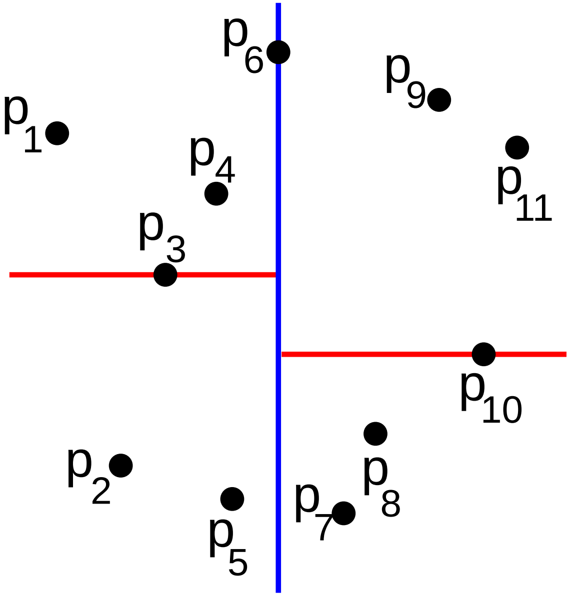
- Extend the concept of binary search by bisection to higher dimensions.
- Instead of intervals, partition (hyper-)rectangles; do the partitioning alternating parallel to the coordinate axes.
- R_i is partitioned into R_j and R_k $\Rightarrow |R_j| \approx |R_k| \approx \frac{1}{2}|R_i|$.
- Structure corresponding to partitioning: balanced binary tree (kD-tree [Bentley, 1975]).
- Node v corresponds to hyperrectangle $R(v)$, $R(\texttt{root}) = \mathbb{R}^d$; children correspond to sub-hyperrectangles.
- Each node v is augmented to store:
  - $\mathcal{S}(v)$: points contained in $R(v)$ (implicitly).
  - $\ell(v)$: representation of split axis.
  - $p(v)$: median of $\mathcal{S}(v)$ w.r.t. $\ell(v)$.

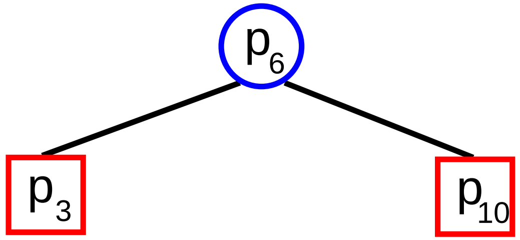
Example





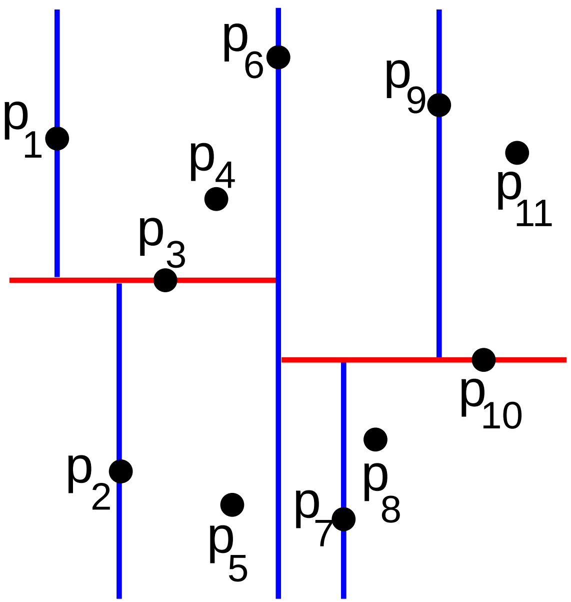
Alternating partitioning along the coordinate axes.

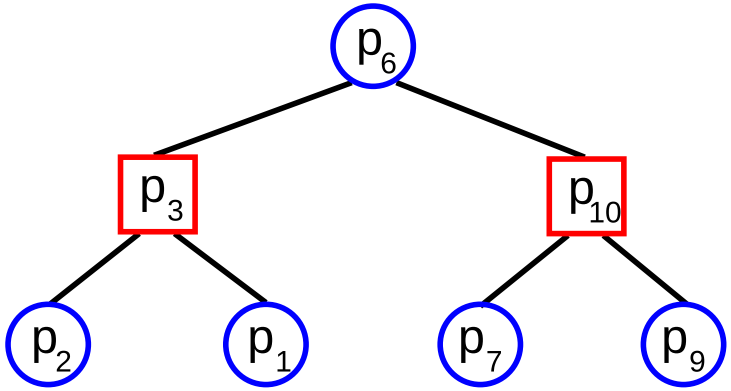




Alternating partitioning along the coordinate axes.

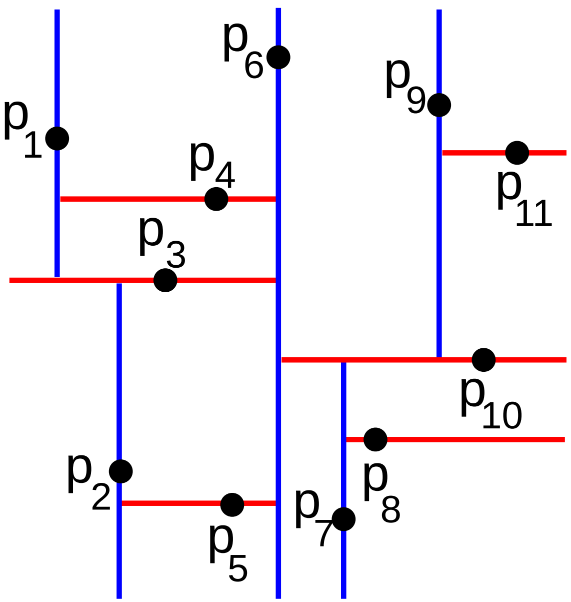
Example

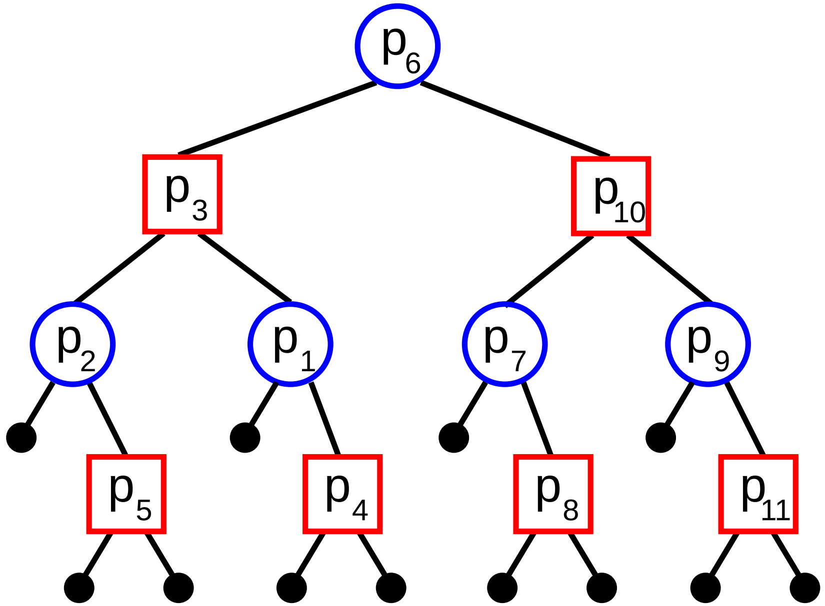




Alternating partitioning along the coordinate axes.

Example

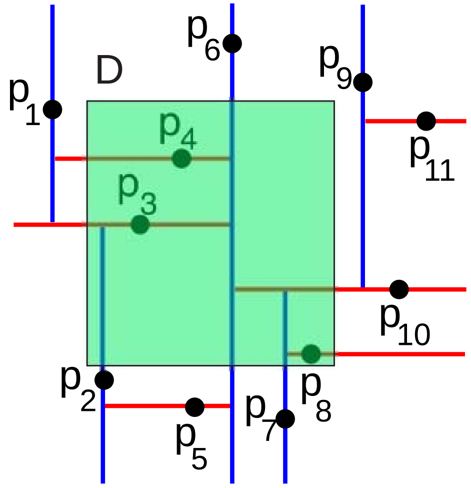


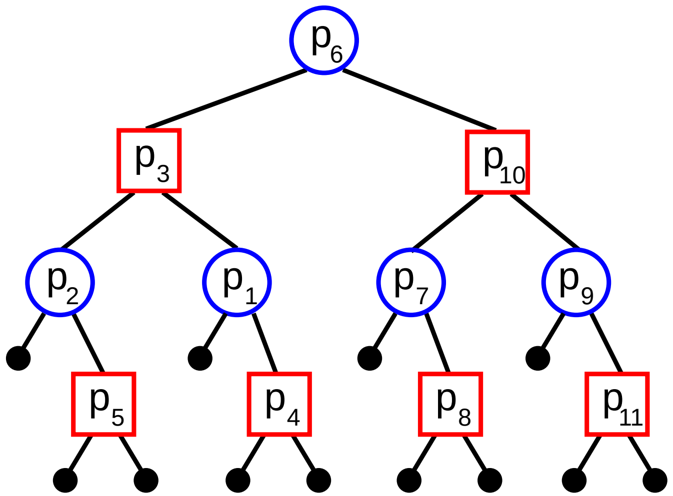


Alternating partitioning along the coordinate axes.

Querying a 2D-tree

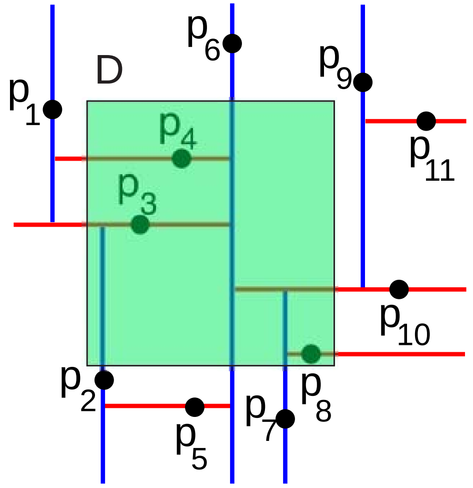
```
void search(node v, rectangle D, list⟨point⟩& result)

    double left, median, right;
    if v.type == "vertical" then
        left = D.x1; right = D.x2;
        median = v.p.x;
    else
        left = D.y1; right = D.y2;
        median = v.p.y;

    if left ≤ median ≤ right and
        D.contains(v.p) then
        result.append(v.p);

    if !isLeaf(v) then
        if left < median then
            search(leftSon(v), D, result);
        if median < right then
            search(rightSon(v), D, result);

    return;
```

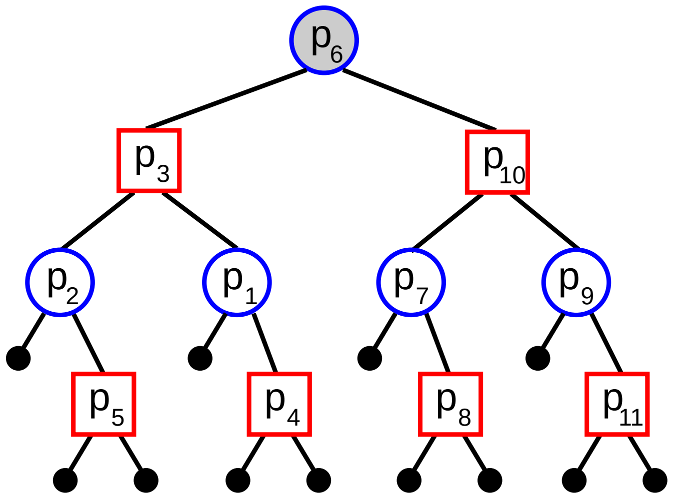




Querying a 2D-tree

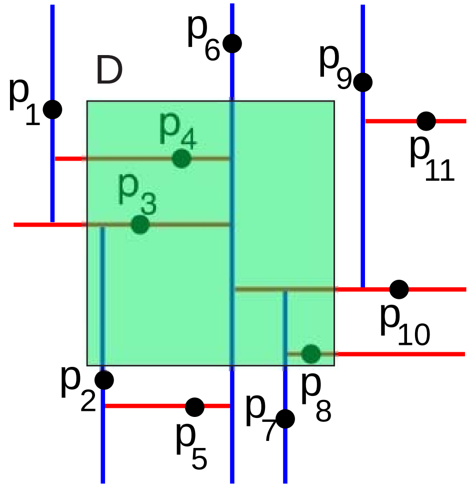
```
void search(node v, rectangle D, list⟨point⟩& result)

    double left, median, right;
    if v.type == "vertical" then
        left = D.x1; right = D.x2;
        median = v.p.x;
    else
        left = D.y1; right = D.y2;
        median = v.p.y;

    if left ≤ median ≤ right and
        D.contains(v.p) then
        result.append(v.p);

    if !isLeaf(v) then
        if left < median then
            search(leftSon(v), D, result);
        if median < right then
            search(rightSon(v), D, result);

    return;
```

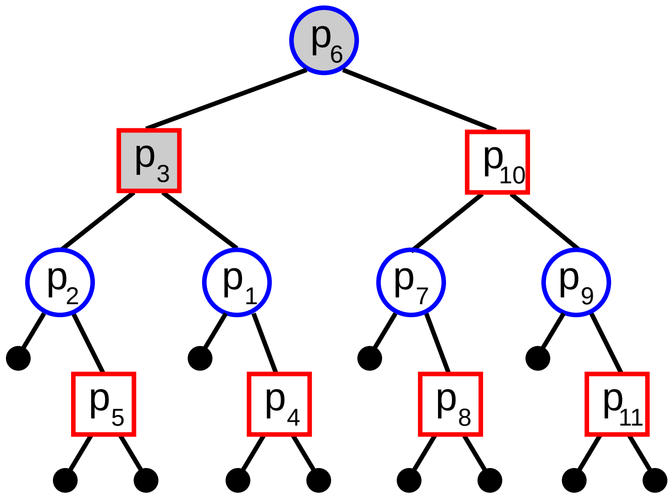




void search(node v, rectangle D, list⟨point⟩& result)

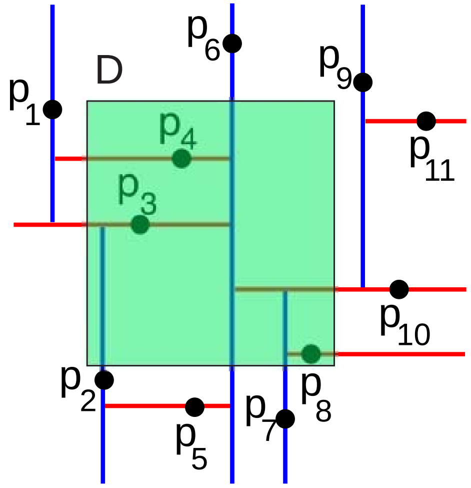
```
double left, median, right;
if v.type == "vertical" then
    left = D.x1; right = D.x2;
    median = v.p.x;
else
    left = D.y1; right = D.y2;
    median = v.p.y;

if left ≤ median ≤ right and
    D.contains(v.p) then
    result.append(v.p);

if !isLeaf(v) then
    if left < median then
        search(leftSon(v), D, result);
    if median < right then
        search(rightSon(v), D, result);

return;
```

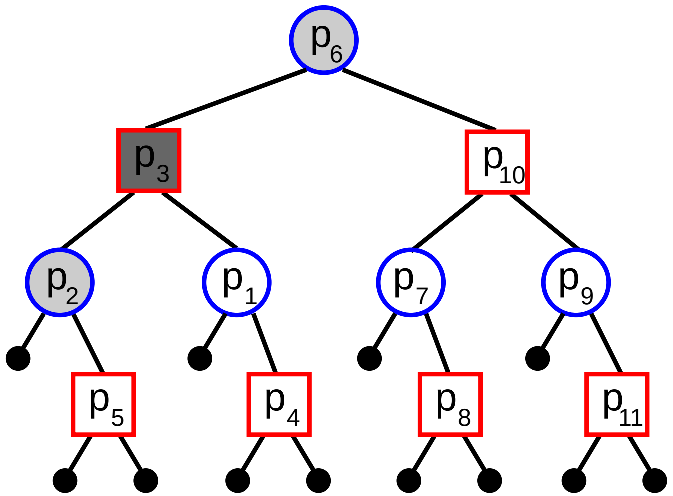




void search(node v, rectangle D, list⟨point⟩& result)

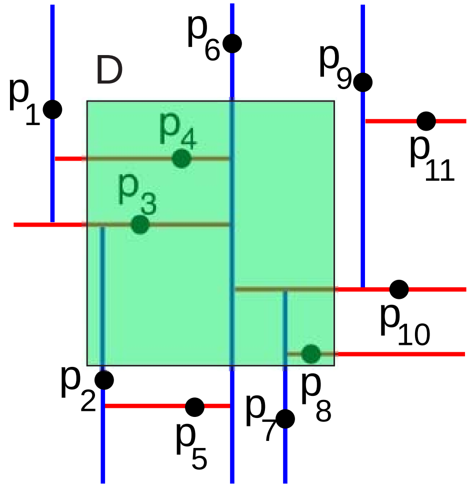
```
double left, median, right;
if v.type == "vertical" then
   left = D.x1; right = D.x2;
   median = v.p.x;
else
   left = D.y1; right = D.y2;
   median = v.p.y;

if left ≤ median ≤ right and
   D.contains(v.p) then
   result.append(v.p);

if !isLeaf(v) then
   if left < median then
      search(leftSon(v), D, result);
   if median < right then
      search(rightSon(v), D, result);

return;
```

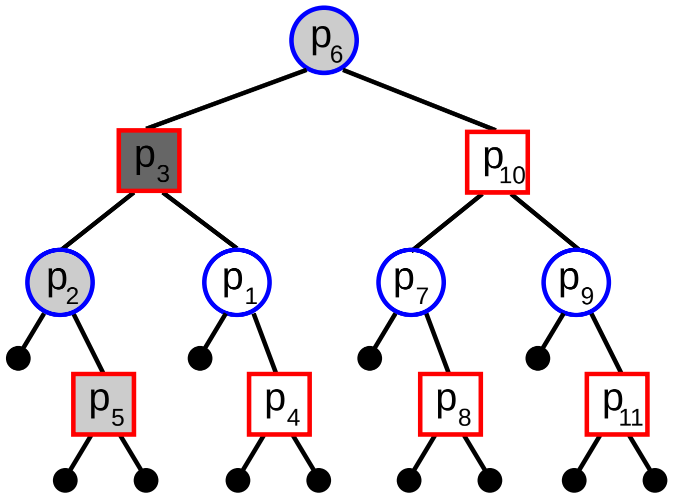




void search(node v, rectangle D, list⟨point⟩& result)

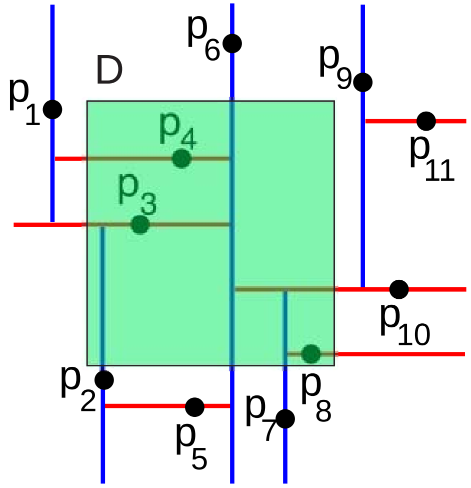
```
double left, median, right;
if v.type == "vertical" then
   left = D.x1; right = D.x2;
   median = v.p.x;
else
   left = D.y1; right = D.y2;
   median = v.p.y;

if left ≤ median ≤ right and
   D.contains(v.p) then
   result.append(v.p);

if !isLeaf(v) then
   if left < median then
      search(leftSon(v), D, result);
   if median < right then
      search(rightSon(v), D, result);

return;
```

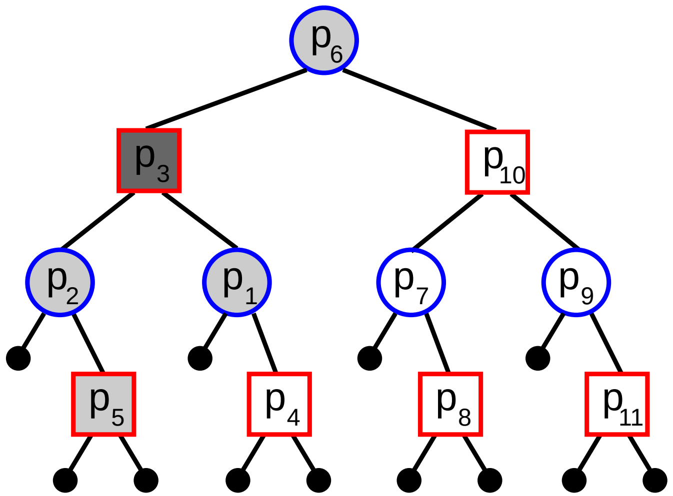




Querying a 2D-tree

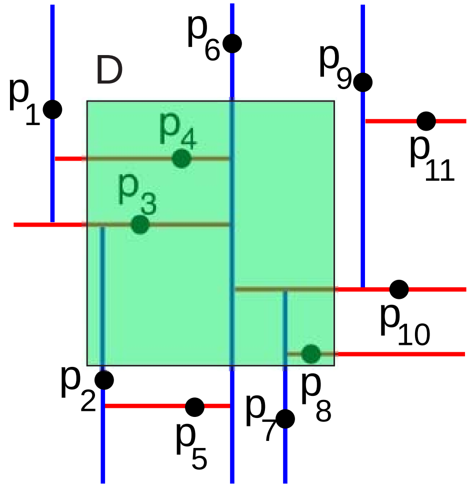
```
void search(node v, rectangle D, list⟨point⟩& result)

    double left, median, right;
    if v.type == "vertical" then
        left = D.x1; right = D.x2;
        median = v.p.x;
    else
        left = D.y1; right = D.y2;
        median = v.p.y;

    if left ≤ median ≤ right and
        D.contains(v.p) then
        result.append(v.p);

    if !isLeaf(v) then
        if left < median then
            search(leftSon(v), D, result);
        if median < right then
            search(rightSon(v), D, result);

    return;
```

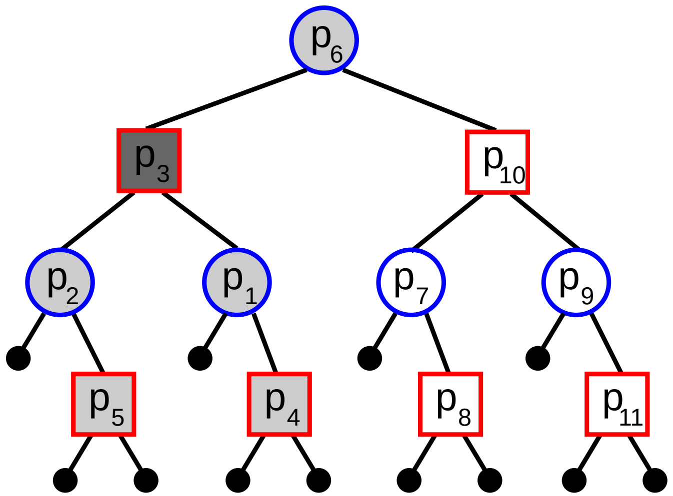




Querying a 2D-tree

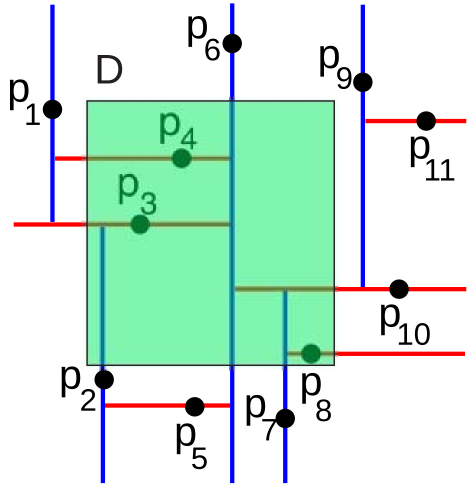
```
void search(node v, rectangle D, list⟨point⟩& result)

    double left, median, right;
    if v.type == "vertical" then
        left = D.x1; right = D.x2;
        median = v.p.x;
    else
        left = D.y1; right = D.y2;
        median = v.p.y;

    if left ≤ median ≤ right and
        D.contains(v.p) then
        result.append(v.p);

    if !isLeaf(v) then
        if left < median then
            search(leftSon(v), D, result);
        if median < right then
            search(rightSon(v), D, result);

    return;
```

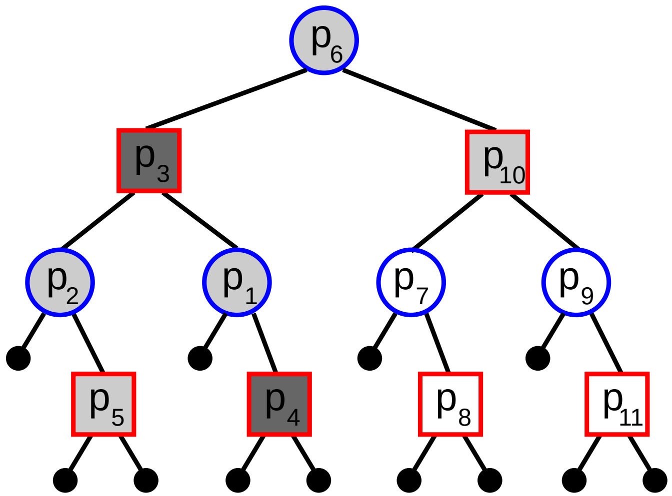




Querying a 2D-tree

```
void search(node v, rectangle D, list⟨point⟩& result)

    double left, median, right;
    if v.type == "vertical" then
        left = D.x1; right = D.x2;
        median = v.p.x;
    else
        left = D.y1; right = D.y2;
        median = v.p.y;

    if left ≤ median ≤ right and
        D.contains(v.p) then
        result.append(v.p);

    if !isLeaf(v) then
        if left < median then
            search(leftSon(v), D, result);
        if median < right then
            search(rightSon(v), D, result);

    return;
```

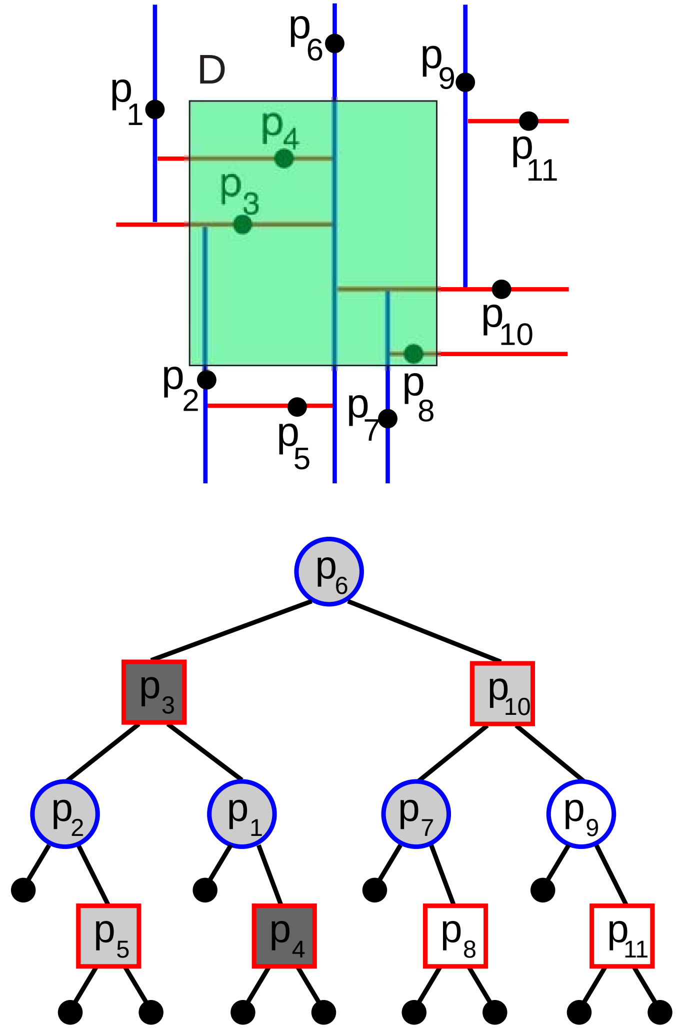




Querying a 2D-tree

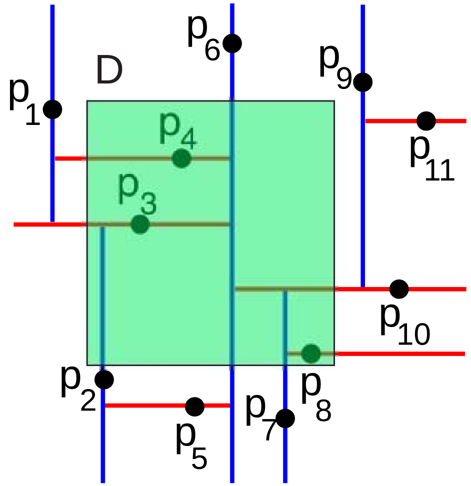
void search(node v, rectangle D, list⟨point⟩& result)

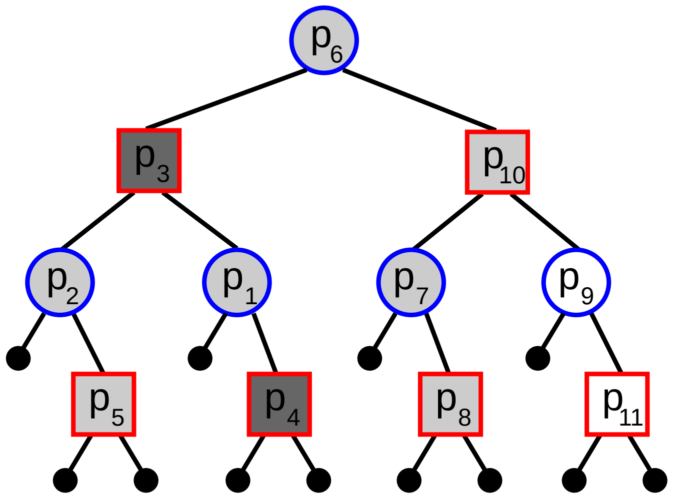
```
double left, median, right;
if v.type == "vertical" then
    left = D.x1; right = D.x2;
    median = v.p.x;
else
    left = D.y1; right = D.y2;
    median = v.p.y;

if left ≤ median ≤ right and
    D.contains(v.p) then
    result.append(v.p);

if !isLeaf(v) then
    if left < median then
        search(leftSon(v), D, result);
    if median < right then
        search(rightSon(v), D, result);

return;
```



void search(node v, rectangle D, list⟨point⟩& result)

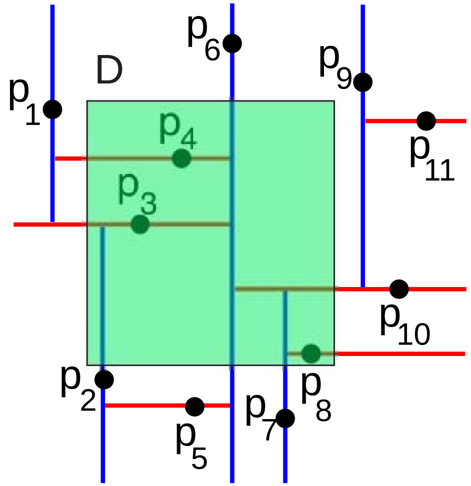
```
double left, median, right;
if v.type == "vertical" then
    left = D.x1; right = D.x2;
    median = v.p.x;
else
    left = D.y1; right = D.y2;
    median = v.p.y;

if left ≤ median ≤ right and
    D.contains(v.p) then
    result.append(v.p);

if !isLeaf(v) then
    if left < median then
        search(leftSon(v), D, result);
    if median < right then
        search(rightSon(v), D, result);

return;
```

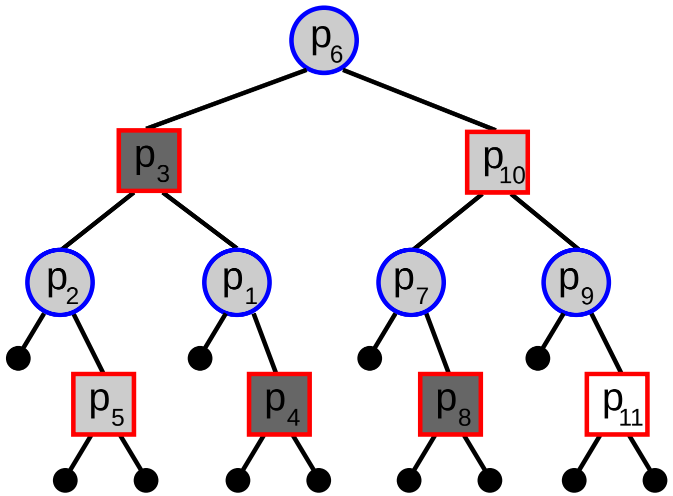




void search(node v, rectangle D, list⟨point⟩& result)

```
double left, median, right;
if v.type == "vertical" then
    left = D.x1; right = D.x2;
    median = v.p.x;
else
    left = D.y1; right = D.y2;
    median = v.p.y;

if left ≤ median ≤ right and
    D.contains(v.p) then
    result.append(v.p);

if !isLeaf(v) then
    if left < median then
        search(leftSon(v), D, result);
    if median < right then
        search(rightSon(v), D, result);

return;
```





Space requirement:

- $p \in R(v) \iff p = p(v) \vee p \in R(q)$ for any descendant q of v.
- $\mathcal{O}(1)$ space requirement per node, exactly one point stored at each node $\Rightarrow$ $\mathcal{O}(n)$ overall space requirement.

Complexity of a 2D-tree

Space requirement:

- $p \in R(v) \iff p = p(v) \vee p \in R(q)$ for any descendant q of v.
- $\mathcal{O}(1)$ space requirement per node, exactly one point stored at each node $\Rightarrow$ $\mathcal{O}(n)$ overall space requirement.

Construction time (preprocessing):

- Linear-time median finding per partitioning step, i.e., recurrence:

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + \mathcal{O}(n) \quad \in \quad \mathcal{O}(n \cdot \log n)$$

Space requirement:

- $p \in R(v) \iff p = p(v) \vee p \in R(q)$ for any descendant q of v.
- $\mathcal{O}(1)$ space requirement per node, exactly one point stored at each node $\Rightarrow$ $\mathcal{O}(n)$ overall space requirement.

Construction time (preprocessing):

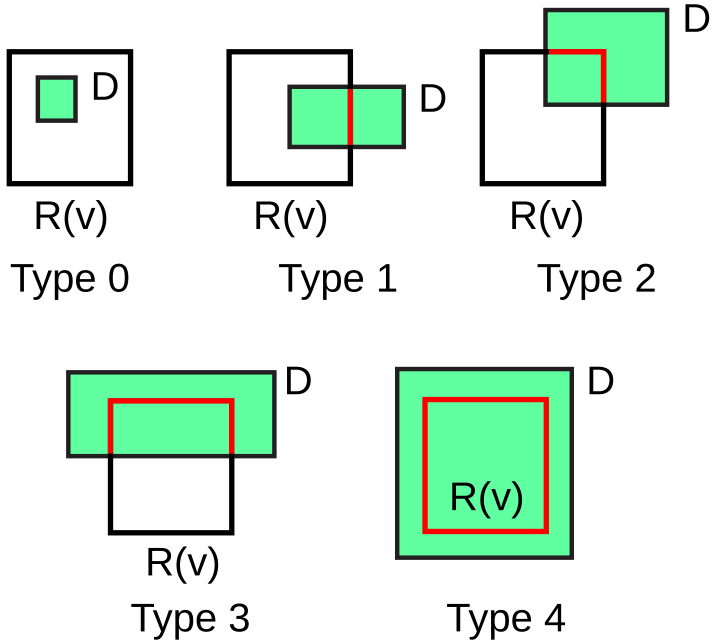
- Linear-time median finding per partitioning step, i.e., recurrence:

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + \mathcal{O}(n) \quad \in \quad \mathcal{O}(n \cdot \log n)$$

- Alternative: Replace median-finding by pre-sorting (copies of) the point by their x- and y-coordinates, respectively.
  - Can find median w.r.t. x-coordinate in $\mathcal{O}(1)$ time.
  - Can construct sorted y-arrays to be passed to the children in linear time.

- Query time proportional to number of nodes visited.

- v productive $\iff$ $p(v) \in D$.
- Nodes visited: productive and unproductive nodes.

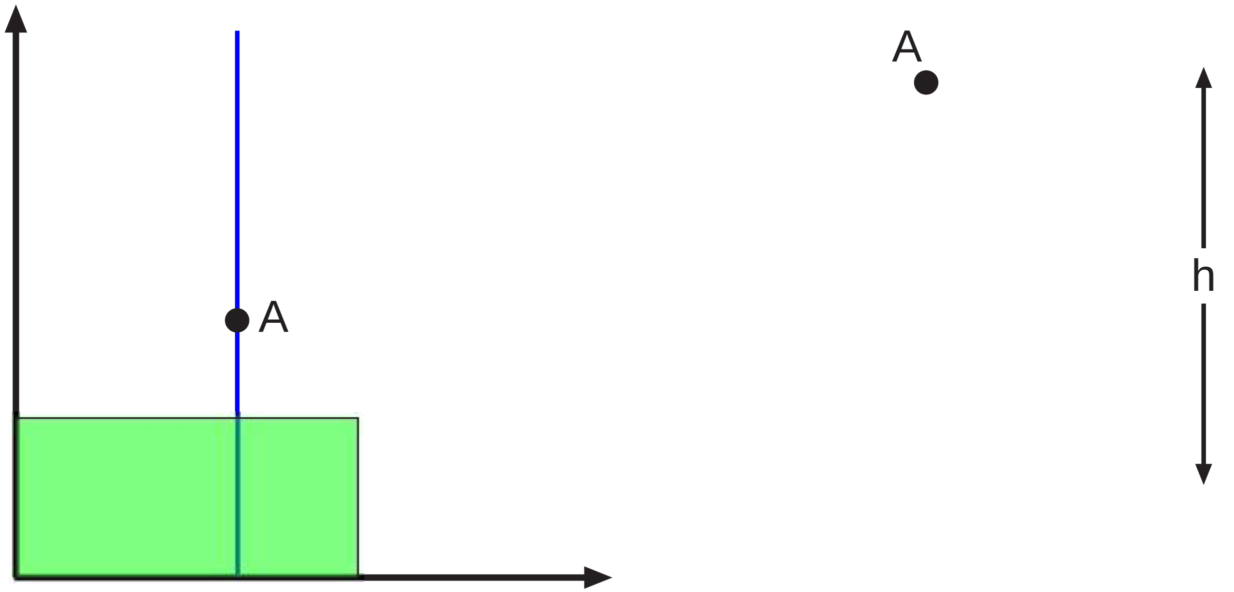


Definition 3.1
Let $R(v)$ be a rectangle and let $0 \leq i \leq 4$. D and $R(v)$ form a type-i situation $\iff$ i sides of $R(v)$ intersect the interior of D.

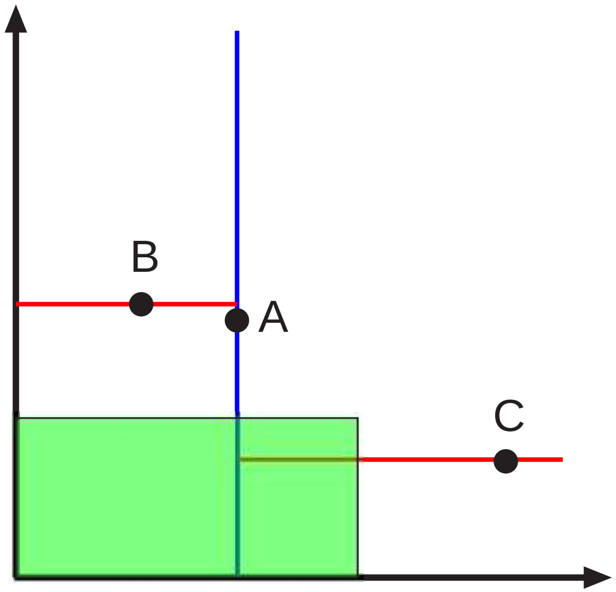
- Type-4 situation always productive, all other situations may be unproductive.

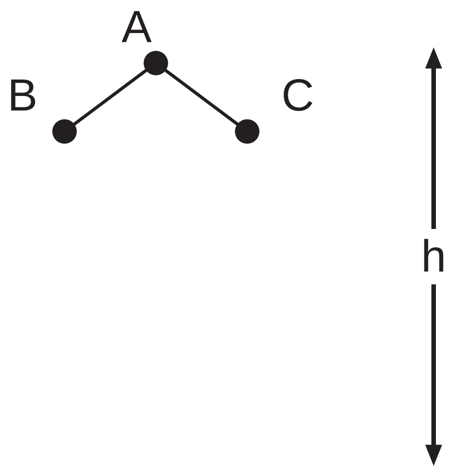
Constructing a worst-case situation–I

- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].



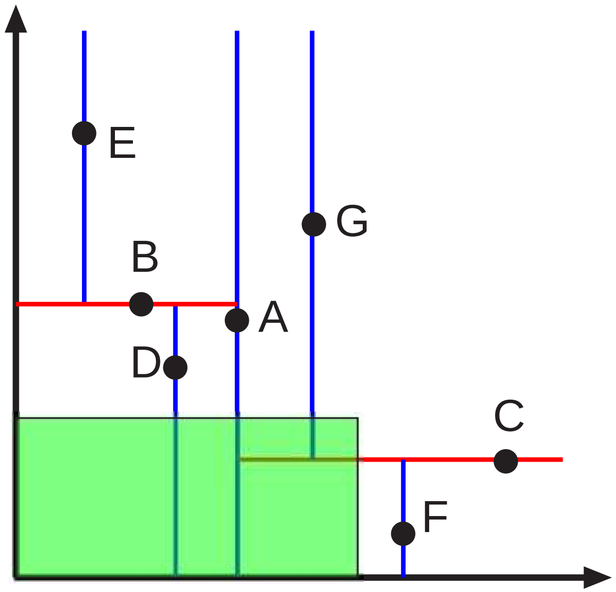
- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].

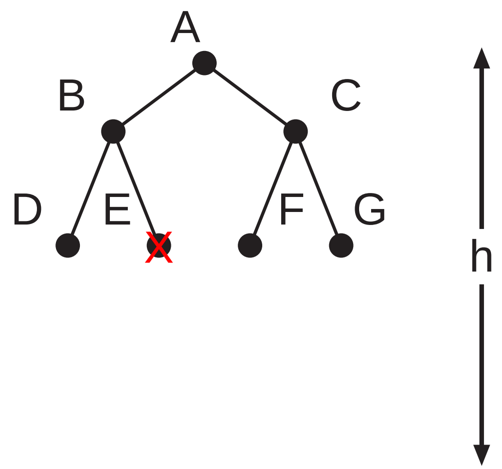




Constructing a worst-case situation–I

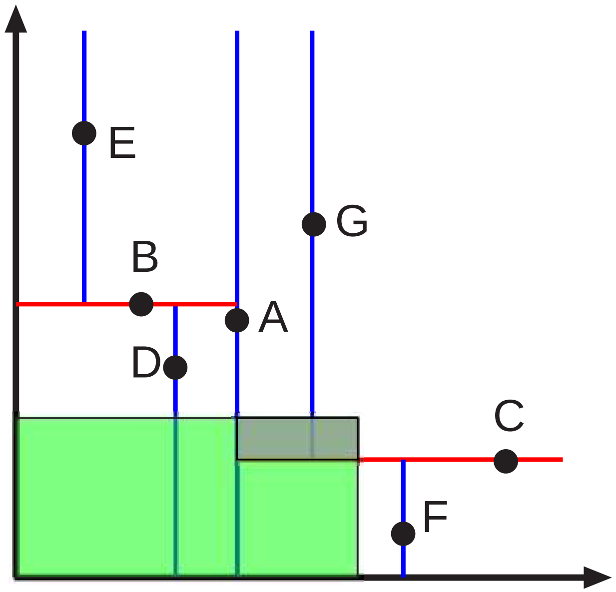
- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].

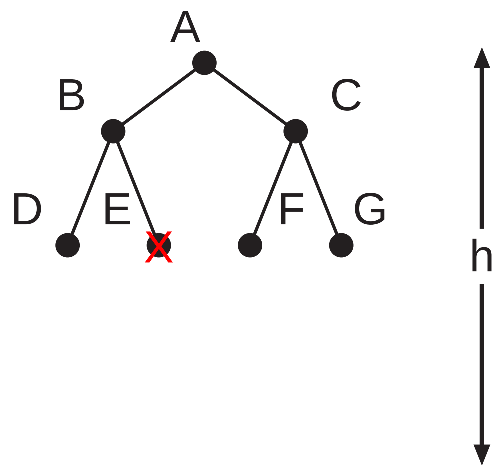




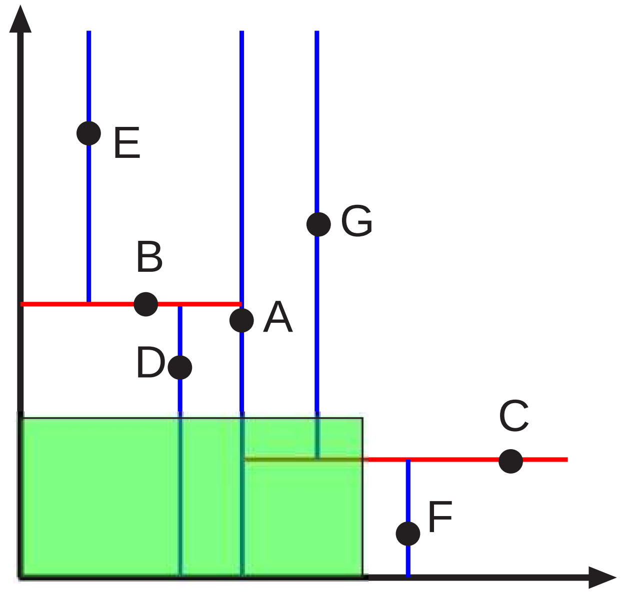
Constructing a worst-case situation–I

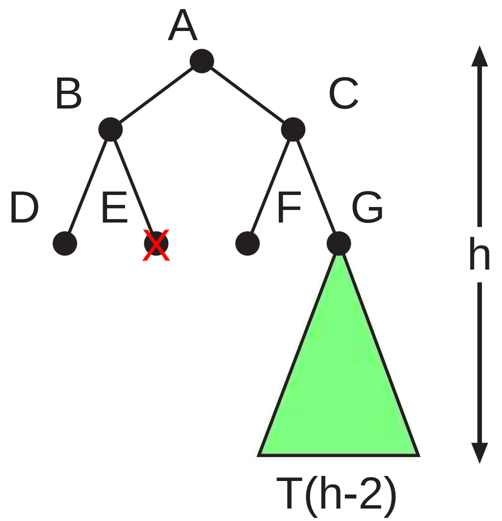
- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].



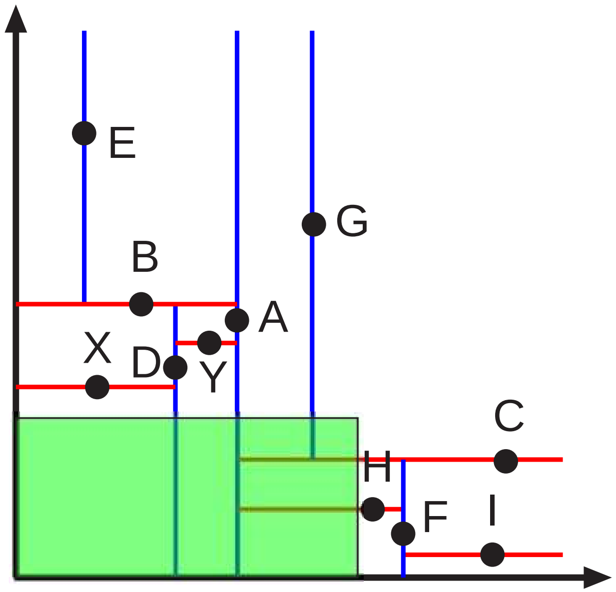


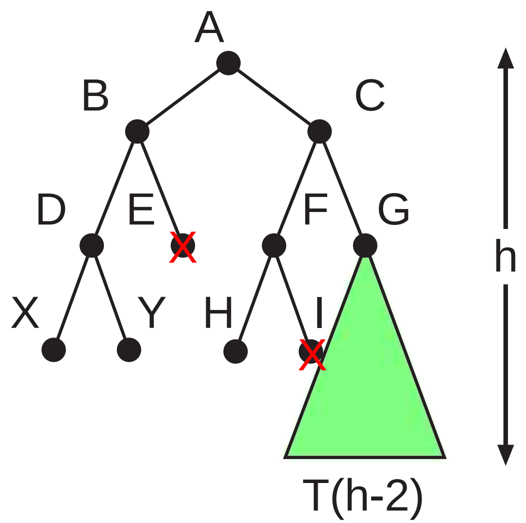
- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].



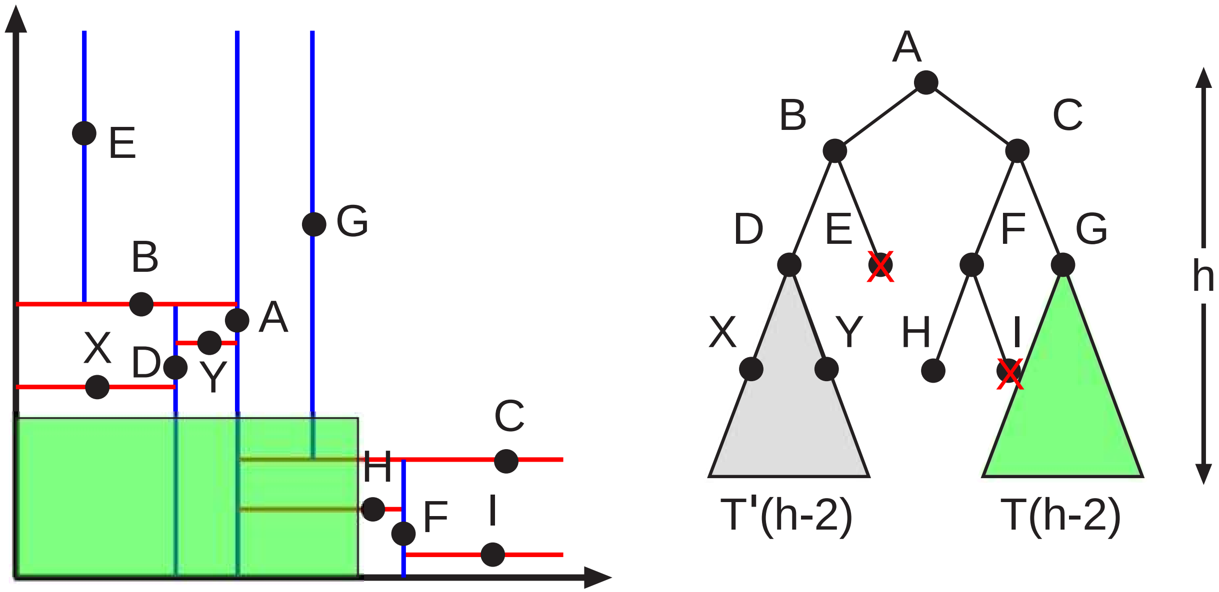


- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].

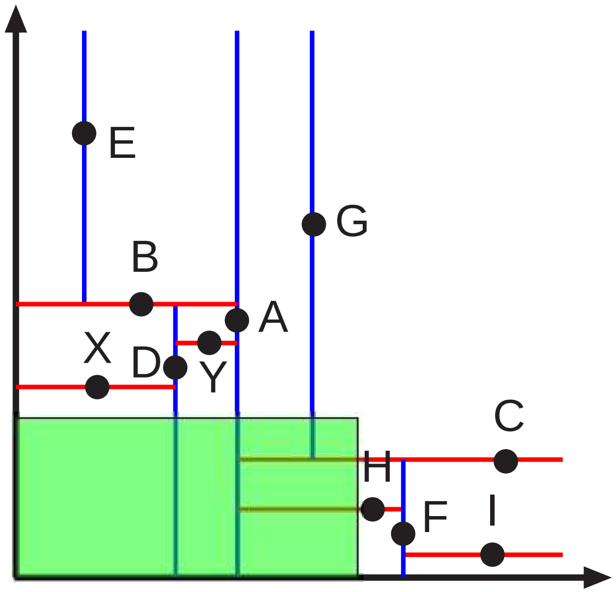


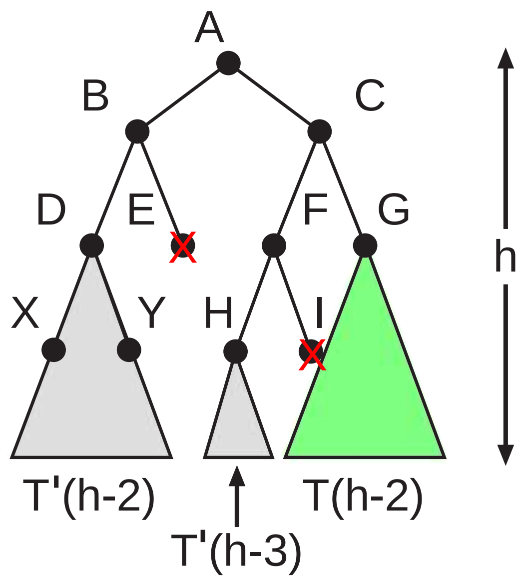


- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].



- Use self-replicating type-2/type-3 situations [Lee & Wong, 1977].



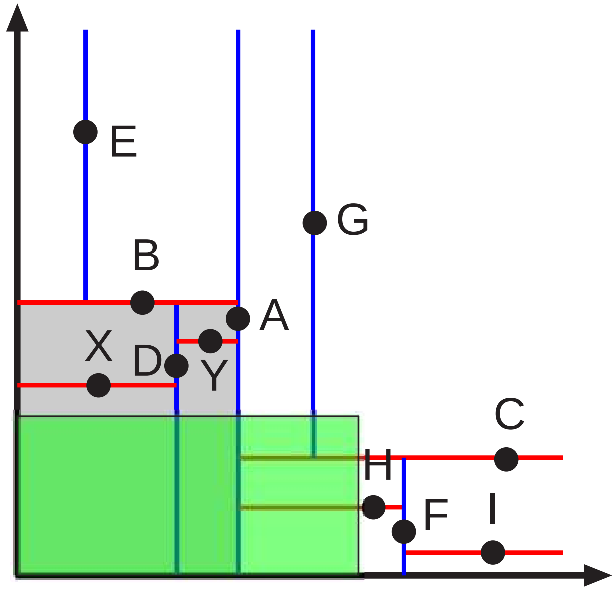


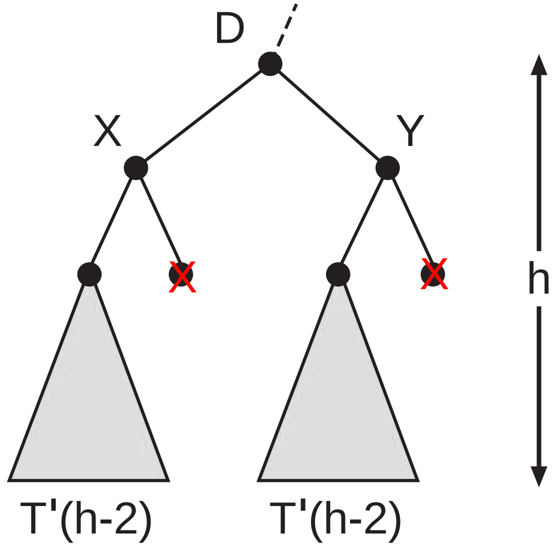
- Recurrence for worst-case query time:

$$T(h) = \underbrace{1}_{A} + \underbrace{1}_{B} + \underbrace{1}_{C} + \underbrace{T(h-2)}_{G} + \underbrace{T'(h-2)}_{D} + \underbrace{1}_{F} + \underbrace{T'(h-3)}_{H}$$

Constructing a worst-case situation—II

- A closer look at situation "subtree rooted at node D".





- Recurrence for this situation:

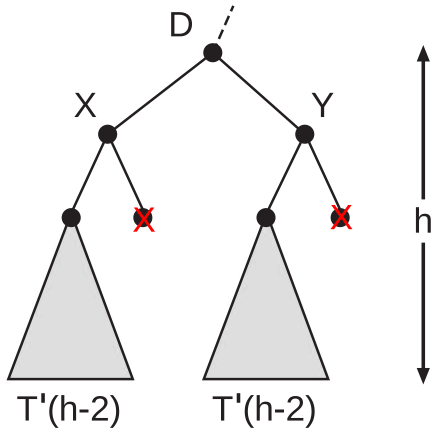
$$T'(h) = \underbrace{1}_{D} + \underbrace{1}_{X} + \underbrace{1}_{Y} + \underbrace{2 \cdot T'(h-2)}_{\text{Children of } X \text{ and } Y}$$

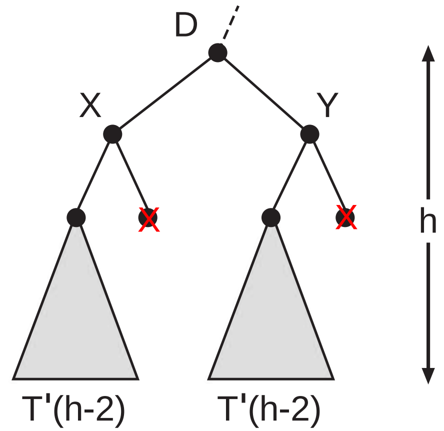
Constructing a worst-case situation–III

- The following recurrence holds for $T'(h)$:

$$T'(h) \;=\; 2 \cdot T'(h-2) + 3$$

with $T'(0) = 0$ and $T'(1) = 1$.





- The following recurrence holds for $T'(h)$:

$$T'(h) \;=\; 2 \cdot T'(h-2) + 3$$

with $T'(0) = 0$ and $T'(1) = 1$.

- Solve recurrence for $T'(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

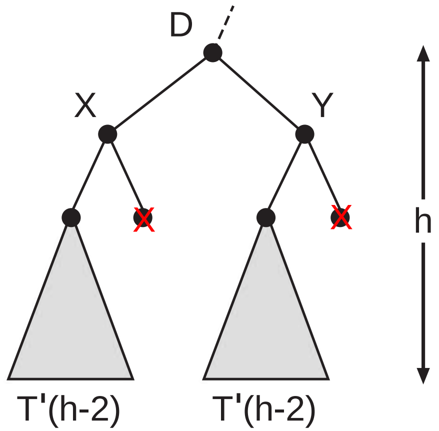
$$\begin{aligned} T'\big(2 \cdot i\big) &= 3 + 2 \cdot T'\big(2(i-1)\big) \\ &= 3 + 2 \cdot \Big(3 + 2 \cdot T'\big(2(i-2)\big)\Big) \\ &= \sum_{j=0}^{i-1} 3 \cdot 2^j \;=\; 3 \cdot 2^i - 3 \end{aligned}$$

Constructing a worst-case situation–III

- The following recurrence holds for $T'(h)$:

$$T'(h) \;=\; 2 \cdot T'(h-2) + 3$$

with $T'(0) = 0$ and $T'(1) = 1$.



- Solve recurrence for $T'(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

$$\begin{aligned} T'\big(2 \cdot i\big) &= 3 + 2 \cdot T'\big(2(i-1)\big) \\ &= 3 + 2 \cdot \Big(3 + 2 \cdot T'\big(2(i-2)\big)\Big) \\ &= \sum_{j=0}^{i-1} 3 \cdot 2^j \;=\; 3 \cdot 2^i - 3 \end{aligned}$$

Similarly: $T'(2 \cdot i + 1) = 4 \cdot 2^i - 3$.

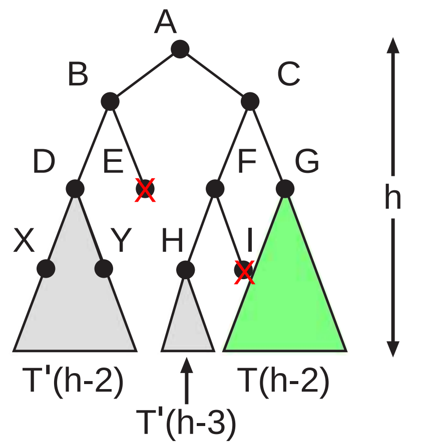
Constructing a worst-case situation–IV

- The following recurrence holds for $T(h)$:

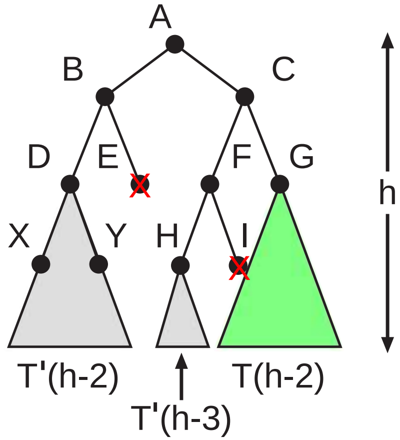
$$T(h) = T(h-2) + T'(h-2) + T'(h-3) + 4$$

$$T'(h) = \begin{cases} 4 \cdot 2^i - 3 & \text{for } h = 2 \cdot i + 1 \\ 3 \cdot 2^i - 3 & \text{for } h = 2 \cdot i \end{cases}$$

with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.



- The following recurrence holds for $T(h)$:



$$T(h) = T(h-2) + T'(h-2) + T'(h-3) + 4$$

$$T'(h) = \begin{cases} 4 \cdot 2^i - 3 & \text{for } h = 2 \cdot i + 1 \\ 3 \cdot 2^i - 3 & \text{for } h = 2 \cdot i \end{cases}$$

with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.

- Solve recurrence for $T(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

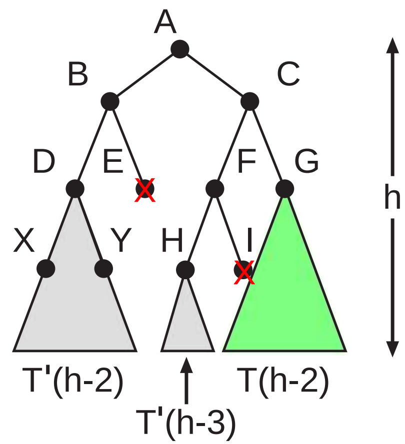
Constructing a worst-case situation–IV

- The following recurrence holds for $T(h)$:

$$T(h) = T(h-2) + T'(h-2) + T'(h-3) + 4$$

$$T'(h) = \begin{cases} 4 \cdot 2^i - 3 & \text{for } h = 2 \cdot i + 1 \\ 3 \cdot 2^i - 3 & \text{for } h = 2 \cdot i \end{cases}$$

with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.

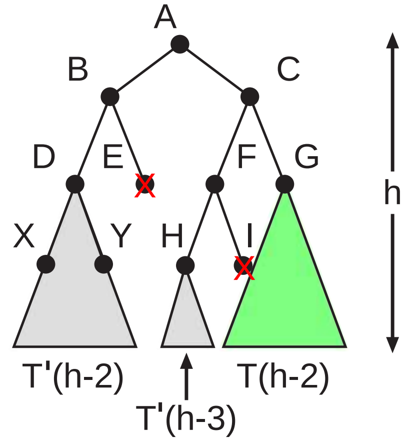


- Solve recurrence for $T(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

$$\begin{aligned} T\big(2 \cdot i\big) &= 4 + T\big(2(i-1)\big) + 3 \cdot 2^{i-1} - 3 + 4 \cdot 2^{i-2} - 3 \\ &= T\big(2(i-1)\big) + 5 \cdot 2^{i-1} - 2 \\ &= 5 \cdot \big(2^{h/2} - 1\big) - h \end{aligned}$$

Constructing a worst-case situation–IV

- The following recurrence holds for $T(h)$:



$$T(h) = T(h-2) + T'(h-2) + T'(h-3) + 4$$

$$T'(h) = \begin{cases} 4 \cdot 2^i - 3 & \text{for } h = 2 \cdot i + 1 \\ 3 \cdot 2^i - 3 & \text{for } h = 2 \cdot i \end{cases}$$

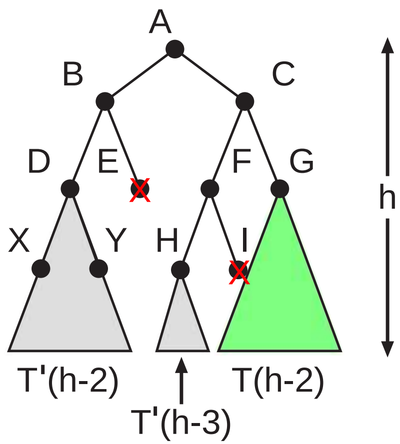
with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.

- Solve recurrence for $T(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

$$\begin{aligned} T\big(2 \cdot i\big) &= 4 + T\big(2(i-1)\big) + 3 \cdot 2^{i-1} - 3 + 4 \cdot 2^{i-2} - 3 \\ &= T\big(2(i-1)\big) + 5 \cdot 2^{i-1} - 2 \\ &= 5 \cdot \big(2^{h/2} - 1\big) - h \end{aligned}$$

Similarly: $T(2 \cdot i + 1) = 7 \cdot \big(2^{\lfloor h/2 \rfloor} - 1\big) - h + 2$.

- The following recurrence holds for $T(h)$:



$$T(h) = T(h-2) + T'(h-2) + T'(h-3) + 4$$

$$T'(h) = \begin{cases} 4 \cdot 2^i - 3 & \text{for } h = 2 \cdot i + 1 \\ 3 \cdot 2^i - 3 & \text{for } h = 2 \cdot i \end{cases}$$

with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.

- Solve recurrence for $T(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

$$\begin{aligned} T\big(2 \cdot i\big) &= 4 + T\big(2(i-1)\big) + 3 \cdot 2^{i-1} - 3 + 4 \cdot 2^{i-2} - 3 \\ &= T\big(2(i-1)\big) + 5 \cdot 2^{i-1} - 2 \\ &= 5 \cdot \left(2^{h/2} - 1\right) - h \end{aligned}$$

Similarly: $T(2 \cdot i + 1) = 7 \cdot \left(2^{\lfloor h/2 \rfloor} - 1\right) - h + 2$.

- Overall (for $n \leq 2^h - 1$): $T(n) \in \mathcal{O}\left(2 \cdot n^{1/2}\right)$.

Summary

- Worst-case query time independent of the number of points reported.
- kD-tree very relevant in practice!
- Extension to higher dimensions (points in $\mathbb{R}^d$): Do partitioning in a round-robin manner of the coordinate axes $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_d \rightarrow x_1 \rightarrow \ldots$

Theorem 3.2

Multidimensional search trees (kD-trees) allow for answering four-sided range queries on points in $\mathbb{R}^d, d \geq 2$ with time and space complexities as follows:

Preprocessing time:	$\Theta\,(d \cdot n \log n)$
Query time:	$\mathcal{O}\left(d \cdot n^{1-1/d} + k\right)$
Space requirement:	$\Theta\,(n)$

Overview

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook

Summary

Lower bounds:

- $\Omega\left(d \cdot \log_2 n + k\right)$ time, $\Omega\left(n\right)$ space.

Summary

Lower bounds:

- $\Omega(d \cdot \log_2 n + k)$ time, $\Omega(n)$ space.

Results:

- One dimension: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.

Lower bounds:

- $\Omega(d \cdot \log_2 n + k)$ time, $\Omega(n)$ space.

Results:

- One dimension: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.
- 1.5 dimensions: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.

Summary

Lower bounds:

- $\Omega(d \cdot \log_2 n + k)$ time, $\Omega(n)$ space.

Results:

- One dimension: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.
- 1.5 dimensions: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.
- Two dimensions: sub-optimal $\mathcal{O}(\sqrt{n} + k)$ algorithm, $\Theta(n)$ space.

Lower bounds:

- $\Omega\left(d \cdot \log_2 n + k\right)$ time, $\Omega\left(n\right)$ space.

Results:

- One dimension: optimal $\mathcal{O}\left(\log_2 n + k\right)$ algorithm, $\Theta\left(n\right)$ space.
- 1.5 dimensions: optimal $\mathcal{O}\left(\log_2 n + k\right)$ algorithm, $\Theta\left(n\right)$ space.
- Two dimensions: sub-optimal $\mathcal{O}\left(\sqrt{n} + k\right)$ algorithm, $\Theta\left(n\right)$ space.
- d dimensions: sub-optimal $\mathcal{O}\left(n^{1-1/d} + k\right)$ algorithm, $\Theta\left(n\right)$ space.

Summary

Lower bounds:

- $\Omega(d \cdot \log_2 n + k)$ time, $\Omega(n)$ space.

Results:

- One dimension: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.
- 1.5 dimensions: optimal $\mathcal{O}(\log_2 n + k)$ algorithm, $\Theta(n)$ space.
- Two dimensions: sub-optimal $\mathcal{O}(\sqrt{n} + k)$ algorithm, $\Theta(n)$ space.
- d dimensions: sub-optimal $\mathcal{O}\left(n^{1-1/d} + k\right)$ algorithm, $\Theta(n)$ space.

Outlook:

- Optimal query time possible of one is willing to spend superlinear space [Chazelle, 1990]. Beware: choosing the adequate model of computation is crucial.

Bibliography

[Bentley & Maurer, 1980] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[Bentley, 1975] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Chazelle, 1990] B. M. Chazelle. Lower bounds for orthogonal range searching. I: The reporting case. *Journal of the ACM*, 37(2):200–212, April 1990.

[de Berg et al., 2000] M. de Berg, M. J. van Kreveld, M. H. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, second edition, 2000.

[Lee & Wong, 1977] D.-T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[McCreight, 1985] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.