Geometric Data Structures: Range Queries

Sujoy Bhore

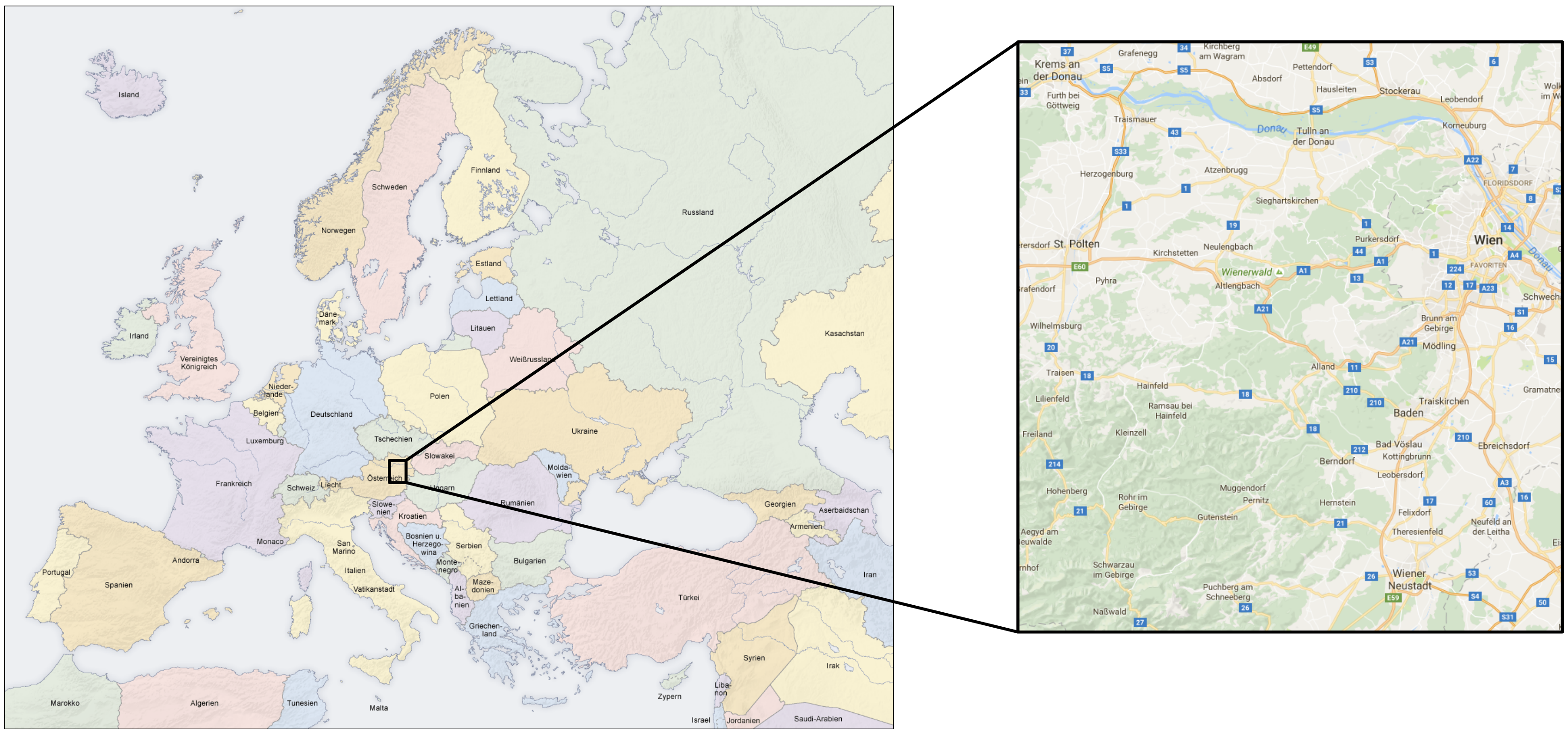
Indian Institute of Technology Bombay

Motivation: Maps

A navigation system should display the current map view. How can we efficiently choose the data to display?

Motivation: Maps

A navigation system should display the current map view. How can we efficiently choose the data to display?

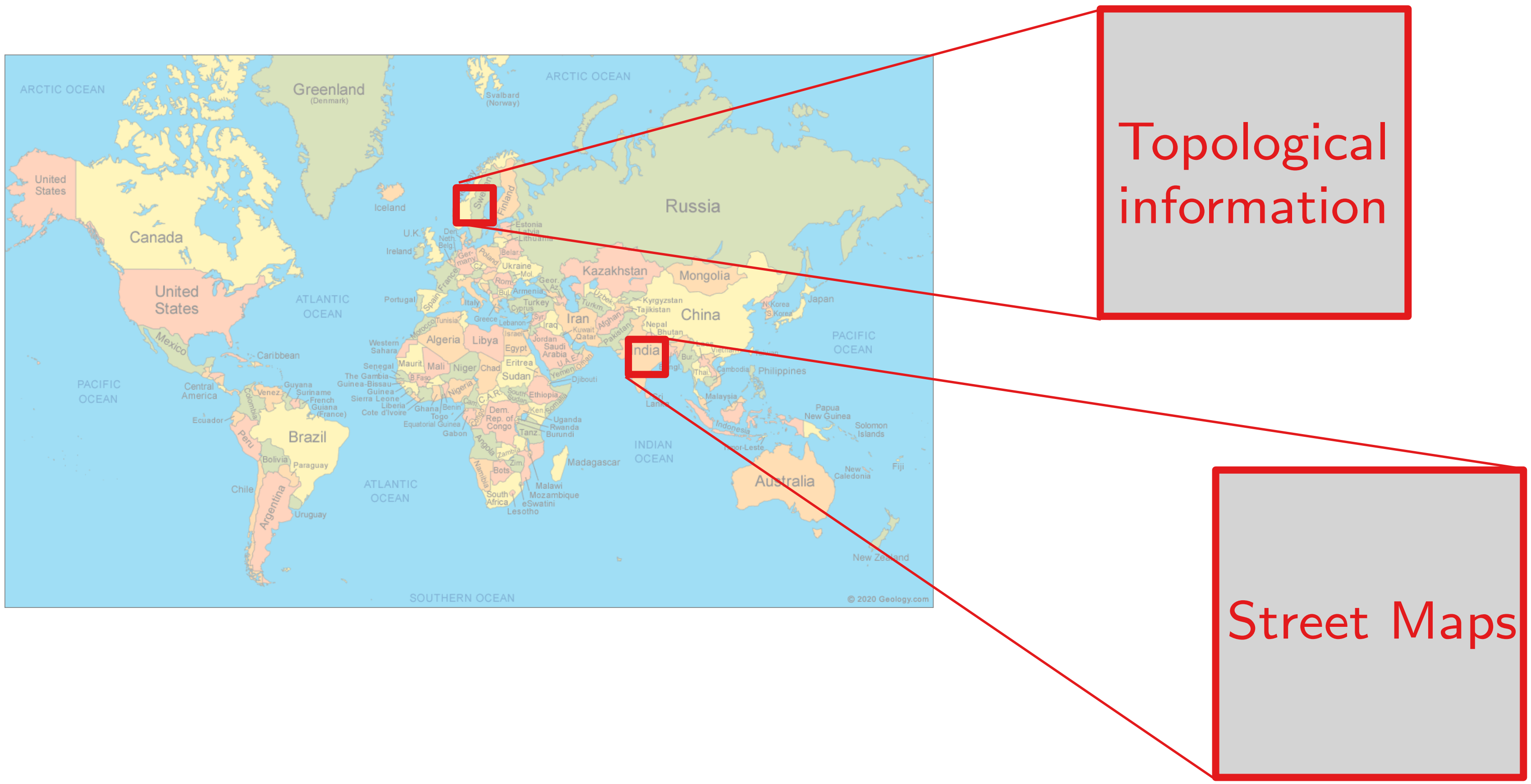


Motivation: Maps

A navigation system should display the current map view. How can we efficiently choose the data to display?

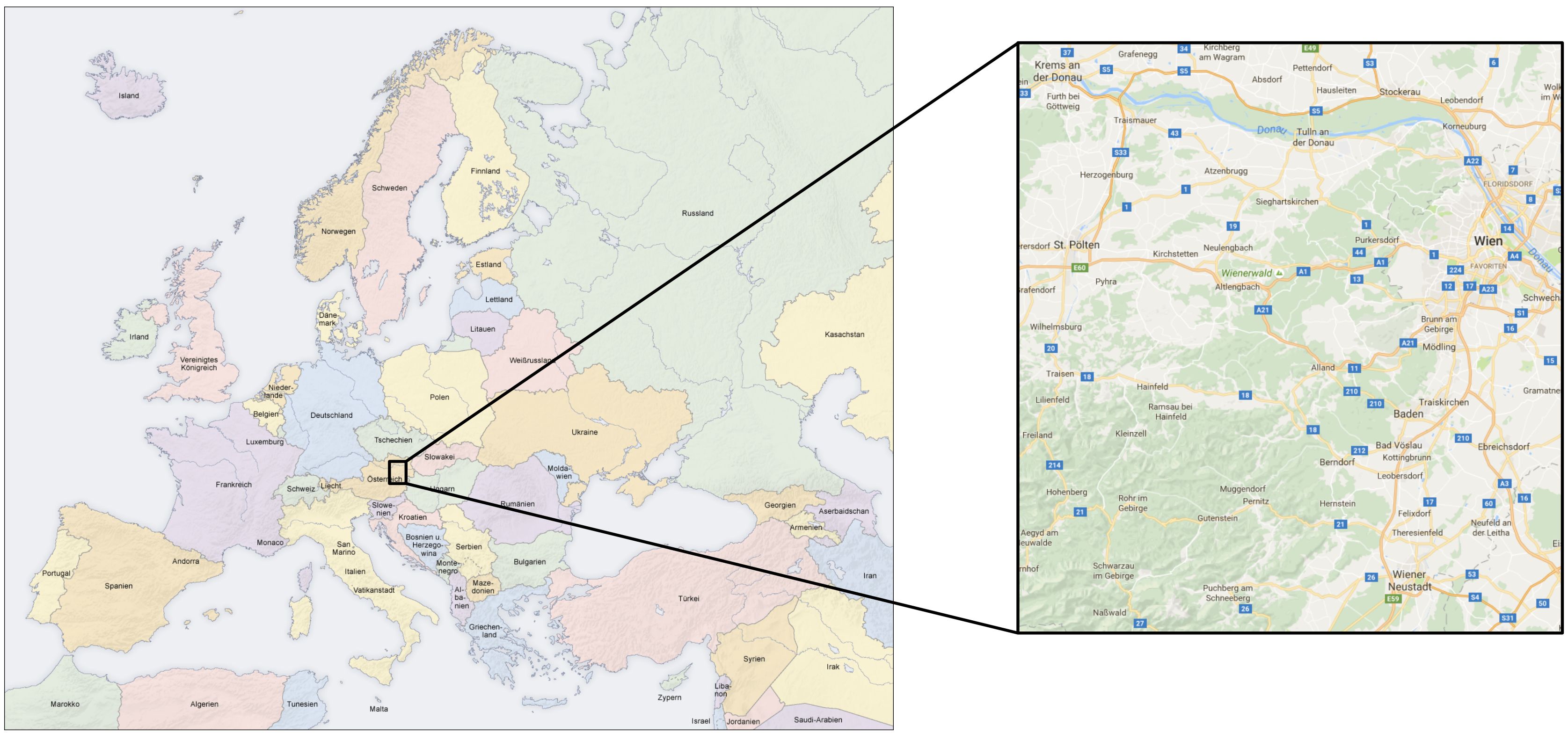
Motivation: Maps

A navigation system should display the current map view. How can we efficiently choose the data to display?



Motivation: Maps

A navigation system should display the current map view. How can we efficiently choose the data to display?



Evaluating each map feature is unrealistic.

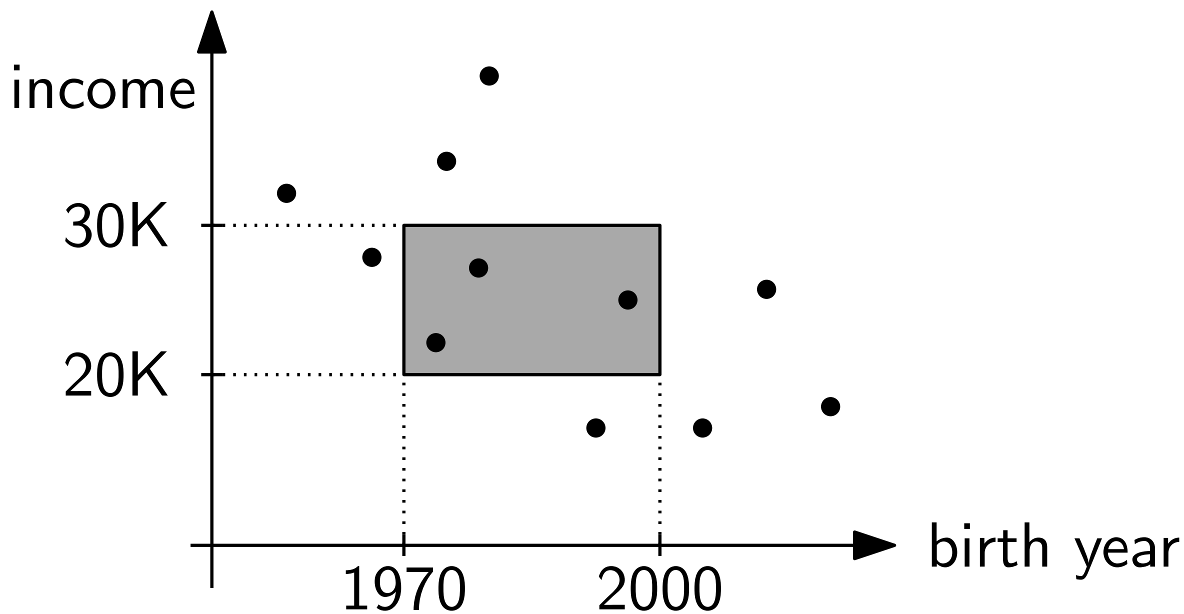
We want a fast data structure for answering range queries

Motivation: Geometry in Databases

In a personnel database, the employees of a company are anonymized and their monthly income and birth year are stored. We now want to perform a search: which employees have an income between 25K and 35K and were born between 1970 and 2000?

Motivation: Geometry in Databases

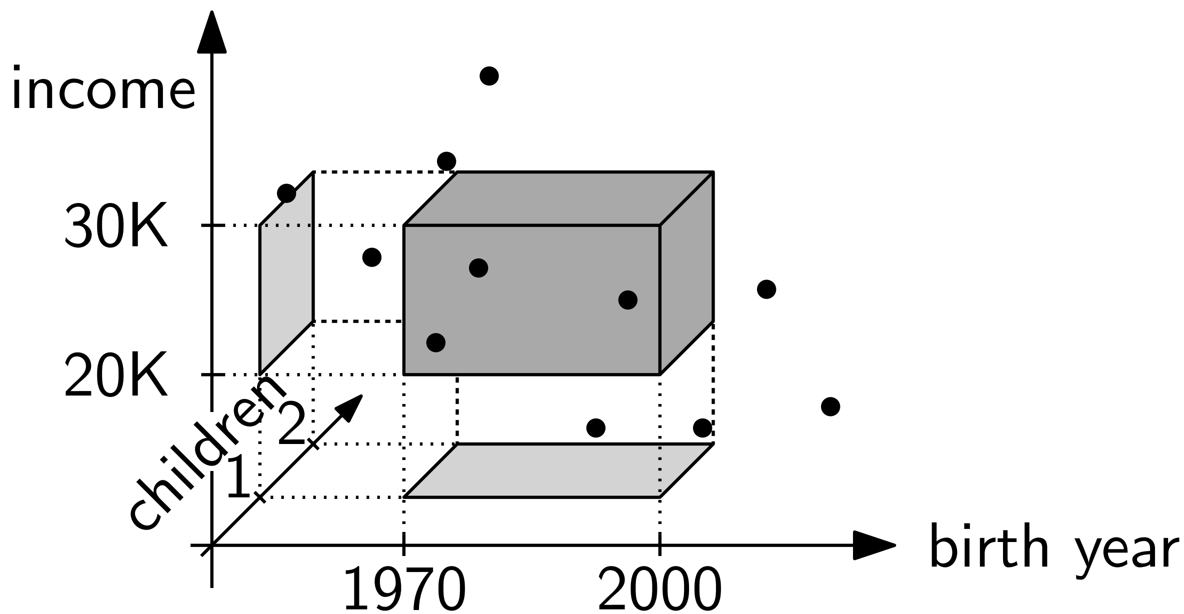
In a personnel database, the employees of a company are anonymized and their monthly income and birth year are stored. We now want to perform a search: which employees have an income between 25K and 35K and were born between 1970 and 2000?



Geometric Interpretation:
Entries are points: (birth year, income level) and the query is an axis-parallel rectangle

Motivation: Geometry in Databases

In a personnel database, the employees of a company are anonymized and their monthly income and birth year are stored. We now want to perform a search: which employees have an income between 25K and 35K and were born between 1970 and 2000?



- This problem can easily be generalized to d dimensions.

Orthogonal Range Queries

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Orthogonal Range Queries

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Orthogonal Range Queries

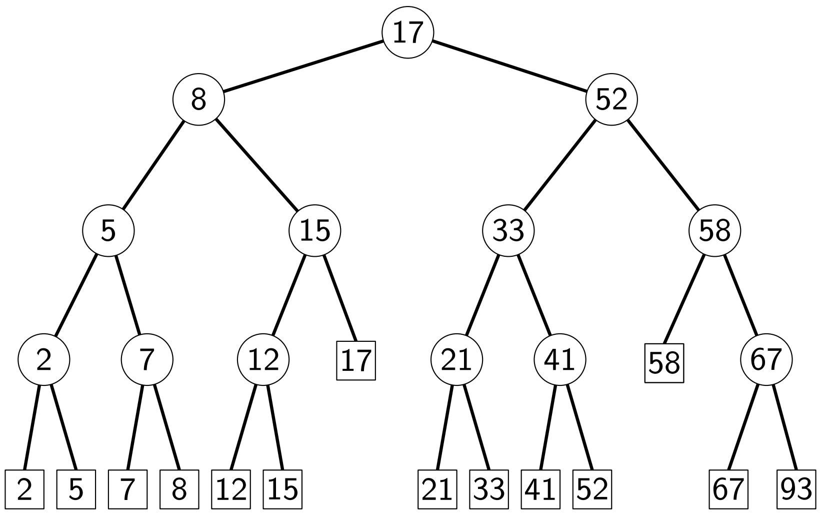
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Orthogonal Range Queries

Given: n points in $\mathbb{R}^d$

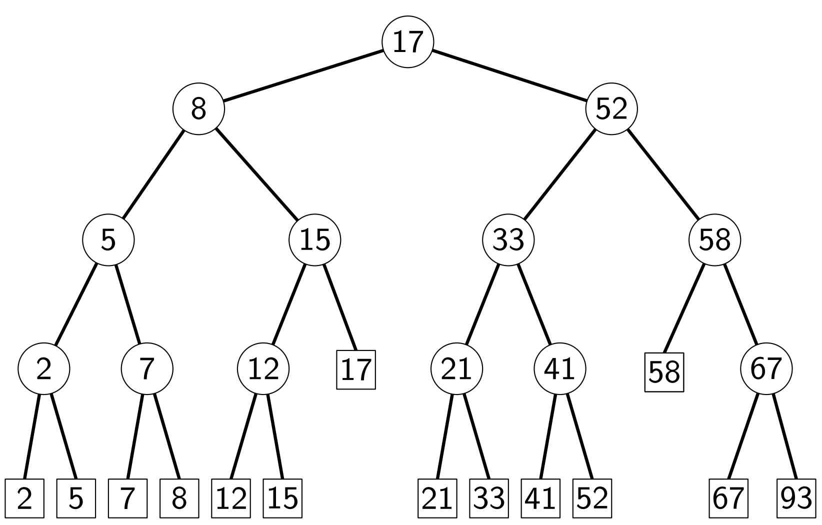
Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v

largest element in the left subtree



Orthogonal Range Queries

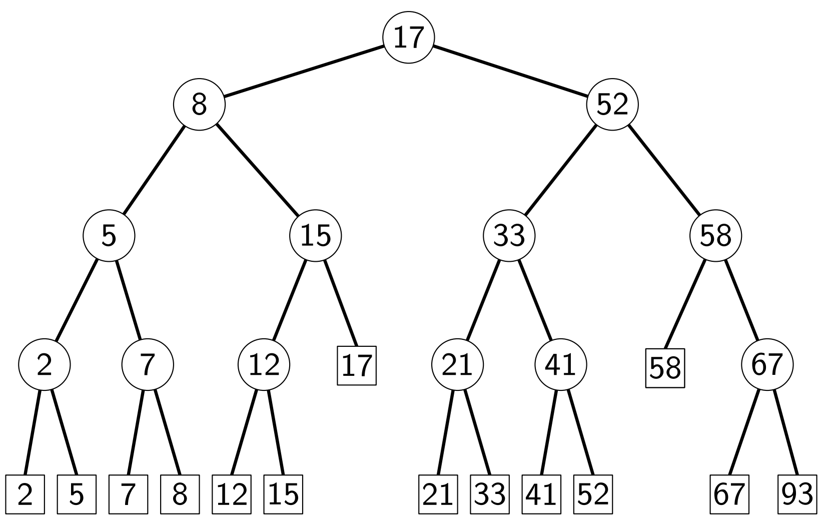
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

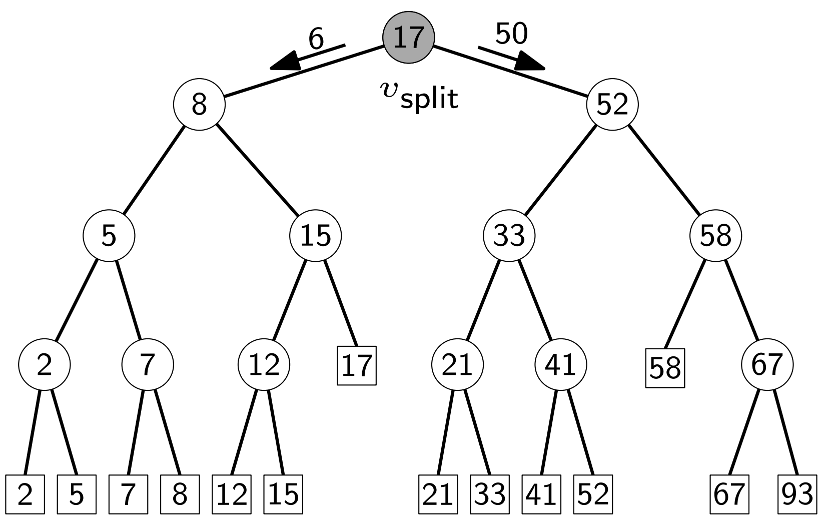
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

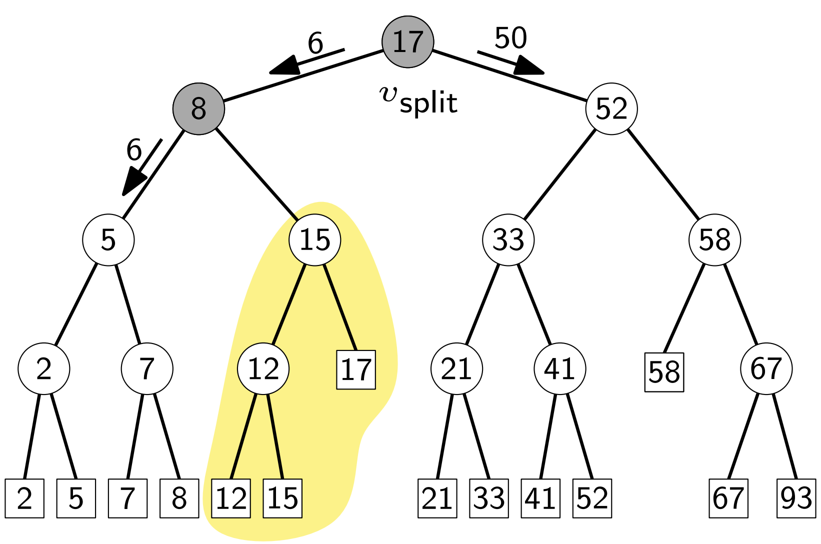
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

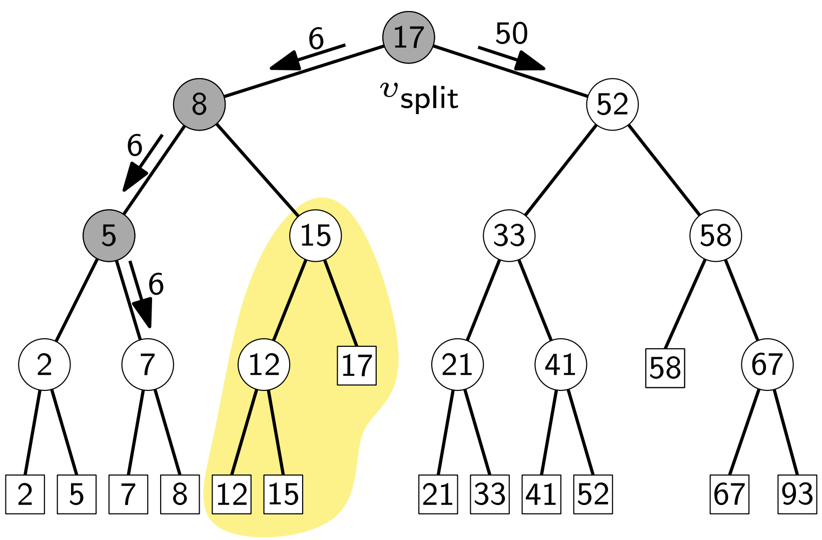
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

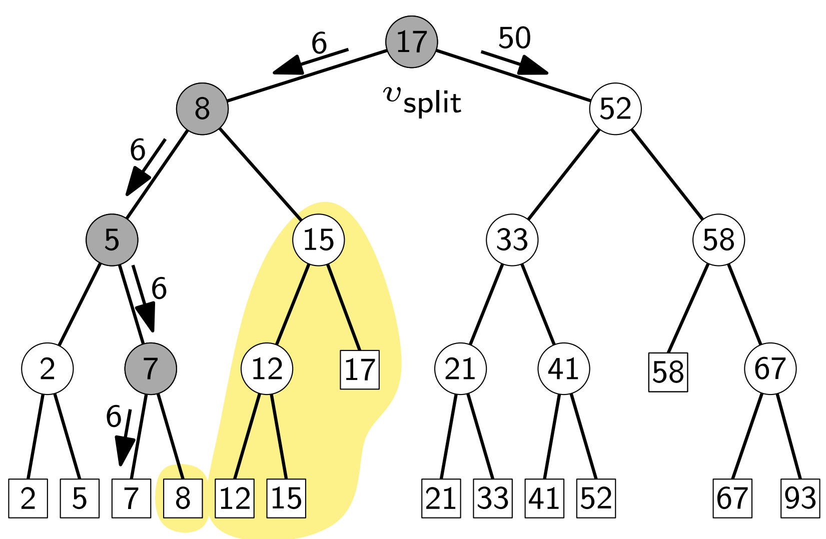
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

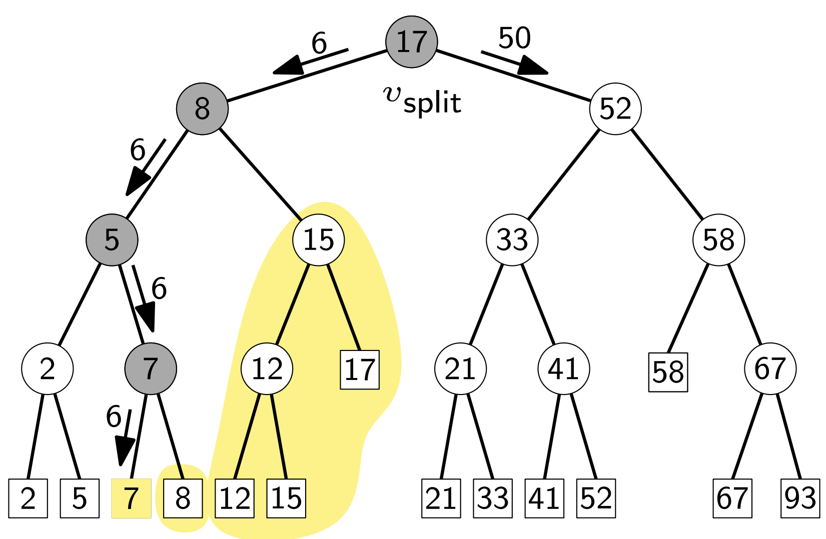
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

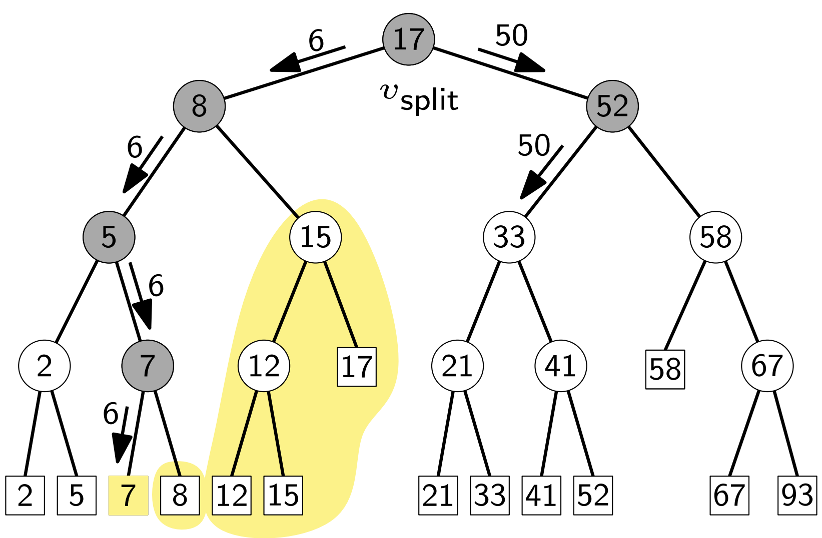
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

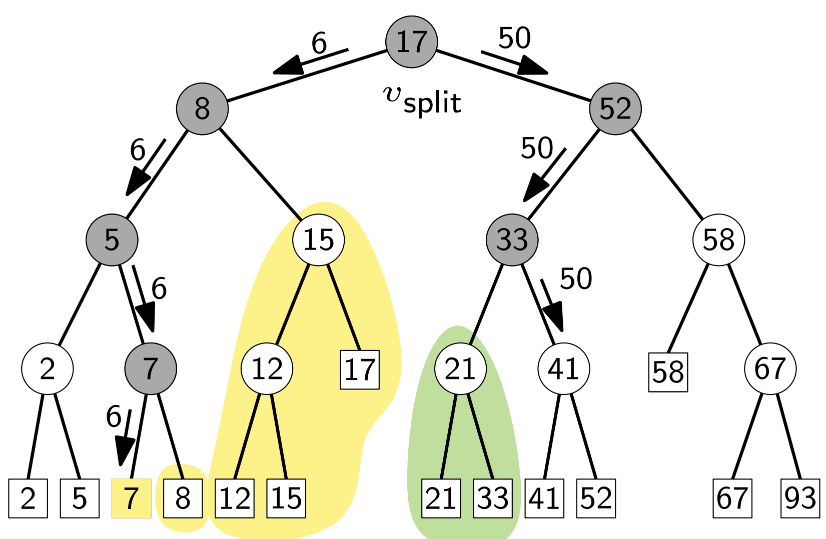
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

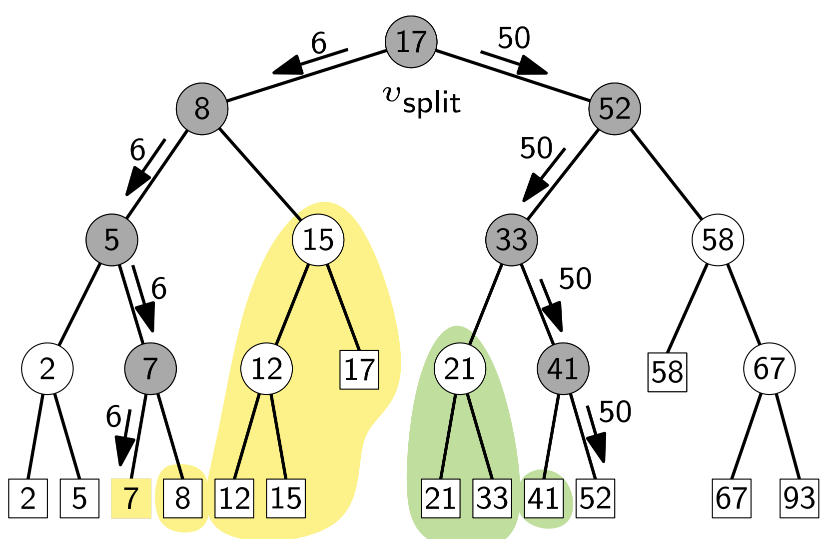
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

Search for all points in [6,50]

Orthogonal Range Queries

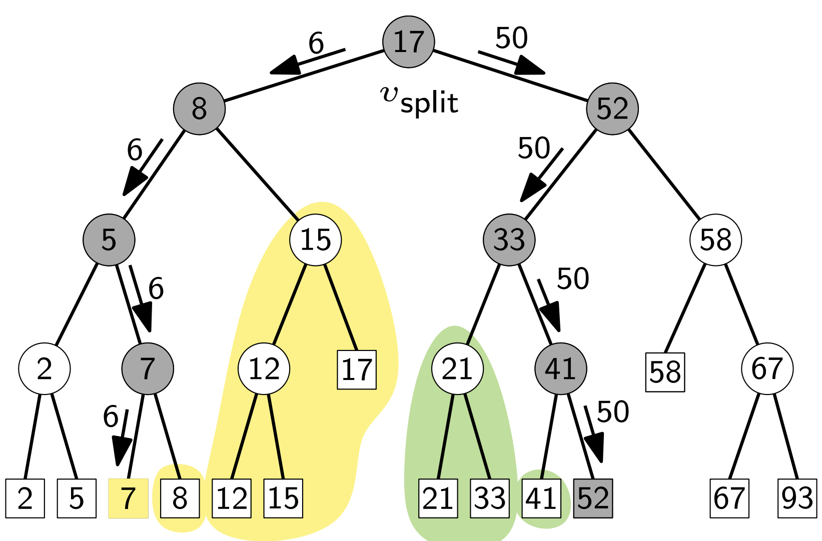
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:
Search for all points in [6,50]

Orthogonal Range Queries

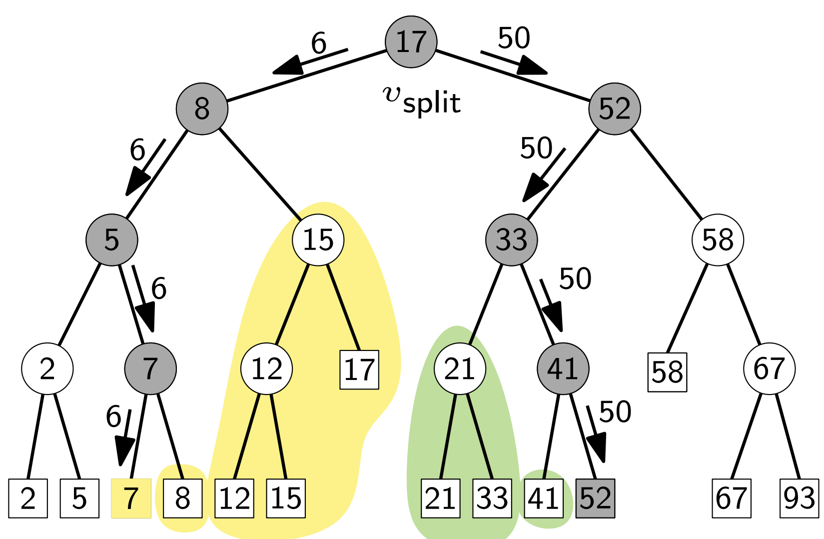
Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Task: Design a data structure for the case $d = 1$.

Solution: balanced binary search tree:

- stores points in the leaves
- internal node v stores pivot value x_v



Example:

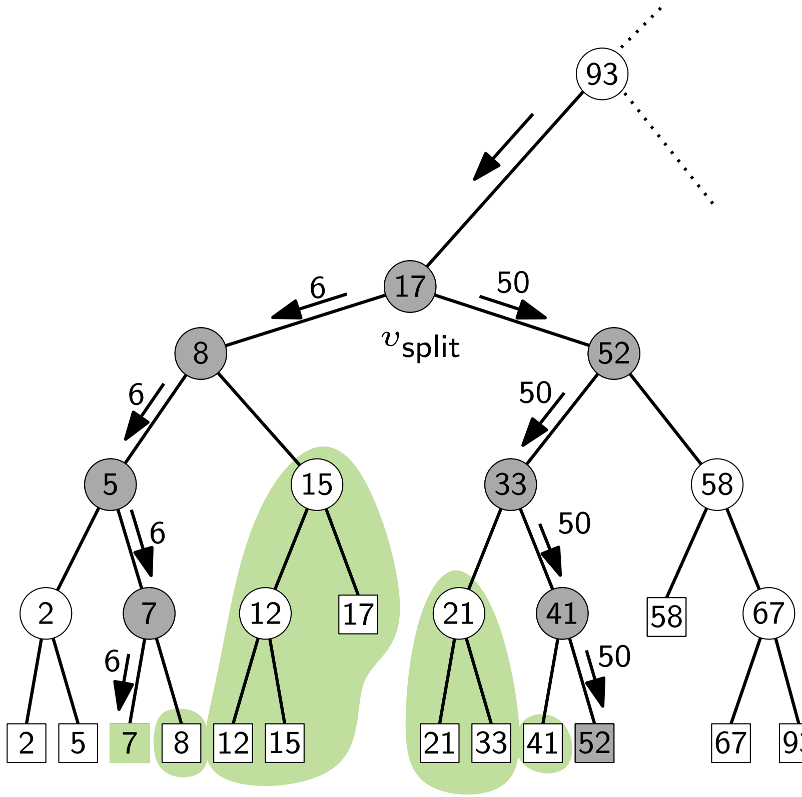
Search for all points in [6,50]

Answer:

Points in the leaves between the search paths, i.e., $\{7,8,12,15,17,21,33,41\}$

1dRangeQuery

Where do the two paths diverge?



1dRangeQuery

FindSplitNode$(T, [x, x'])$

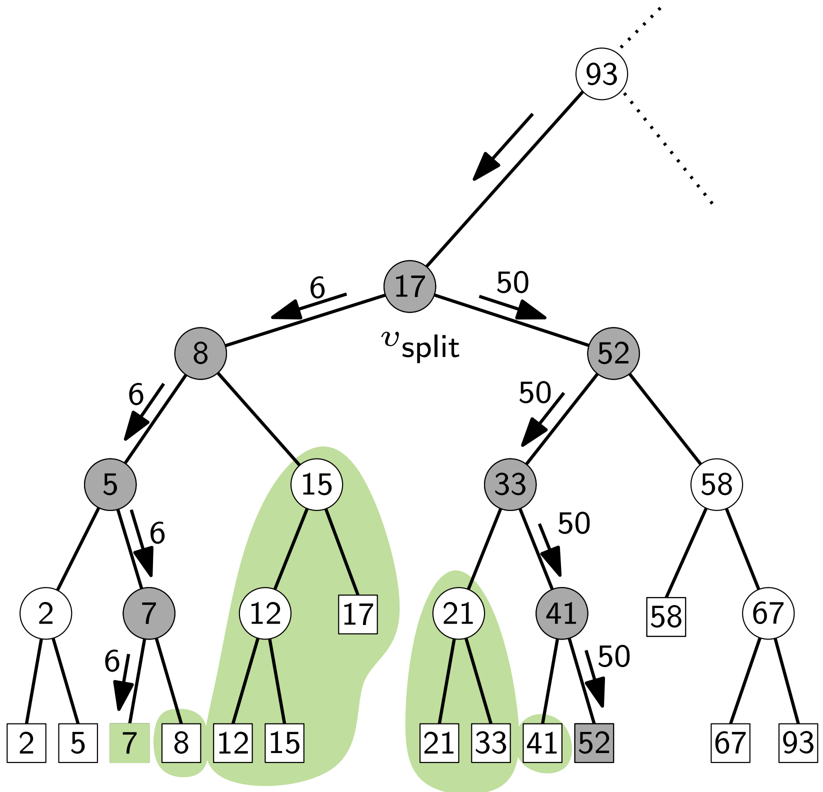
$v \leftarrow$ root(T)

while v not a leaf and $(x' \le x_v$ or $x > x_v)$ **do**

  if $x' \le x_v$ **then** $v \leftarrow$ lc(v) **else** $v \leftarrow$ rc(v)

return v

Where do the two paths diverge?



1dRangeQuery

FindSplitNode$(T, [x, x'])$

$v \leftarrow$ root(T)

while v not a leaf and $(x' \leq x_v$ or $x > x_v)$ **do**

  if $x' \leq x_v$ **then** $v \leftarrow$ lc(v) **else** $v \leftarrow$ rc(v)

return v

Where do the two paths diverge?

1dRangeQuery$(T, [x, x'])$

$v_{\text{split}} \leftarrow$ FindSplitNode$(T, [x, x'])$

if v_{split} is leaf **then** check v_{split}

else

  $v \leftarrow$ lc(v_{split})

  while v not a leaf **do**

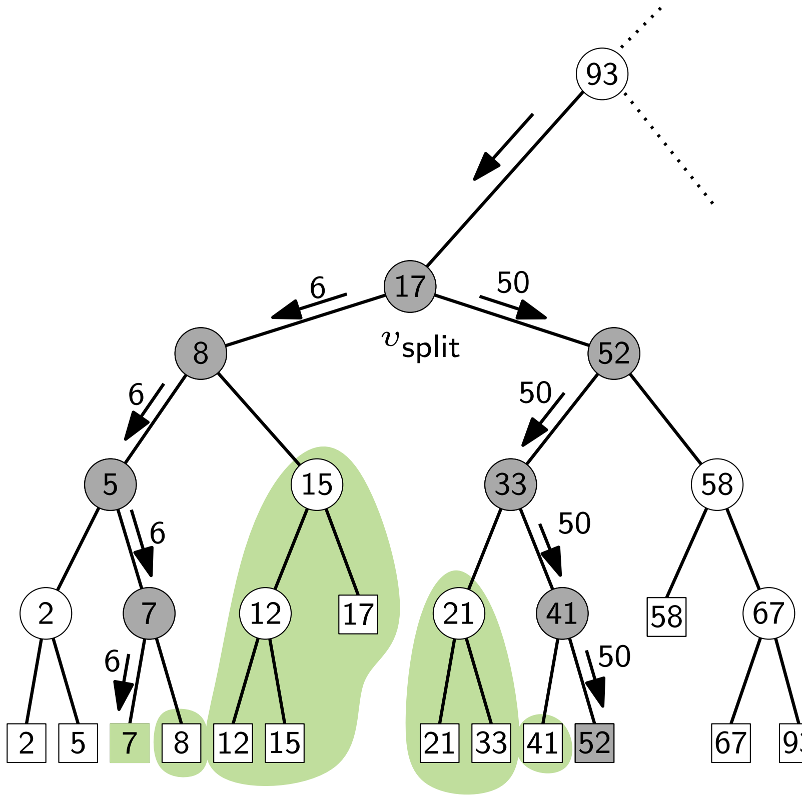
    if $x \leq x_v$ **then**

      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)

    else $v \leftarrow$ rc(v)

  if $x \leq x_v$ **then** report v

  `// analog. for` x' `and rc(`v_{split}`)`



1dRangeQuery

FindSplitNode$(T, [x, x'])$

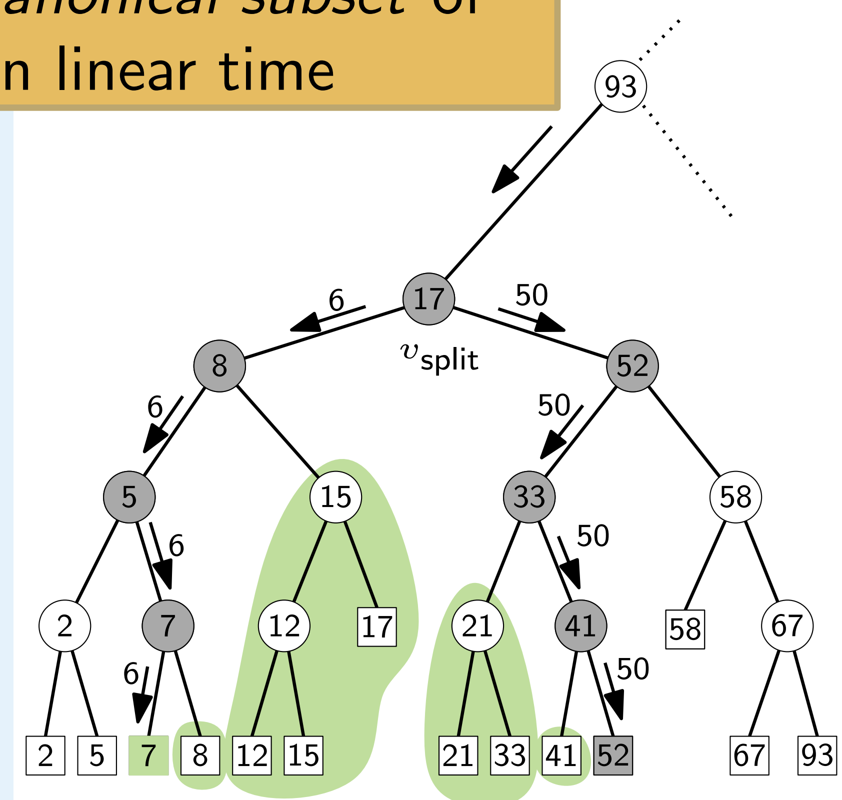
$v \leftarrow$ root(T)
while v not a leaf and $(x' \leq x_v$ or $x > x_v)$ **do**
  if $x' \leq x_v$ **then** $v \leftarrow$ lc(v) **else** $v \leftarrow$ rc(v)
return v

Where do the two paths diverge?

1dRangeQuery$(T, [x, x'])$

$v_{\text{split}} \leftarrow$ FindSplitNode$(T, [x, x'])$
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  if $x \leq x_v$ **then** report v
  `// analog. for` x' `and rc(`v_{split}`)`

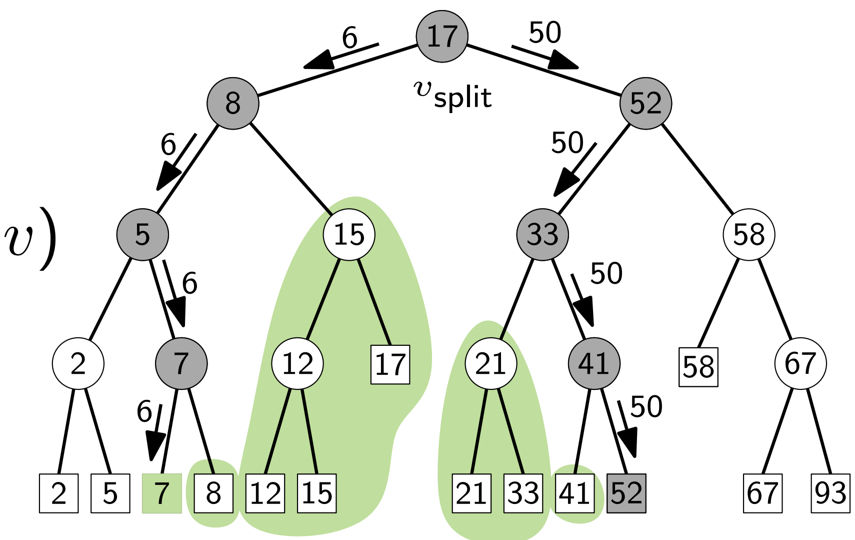
Can find *canonical subset* of all leaves in linear time



Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

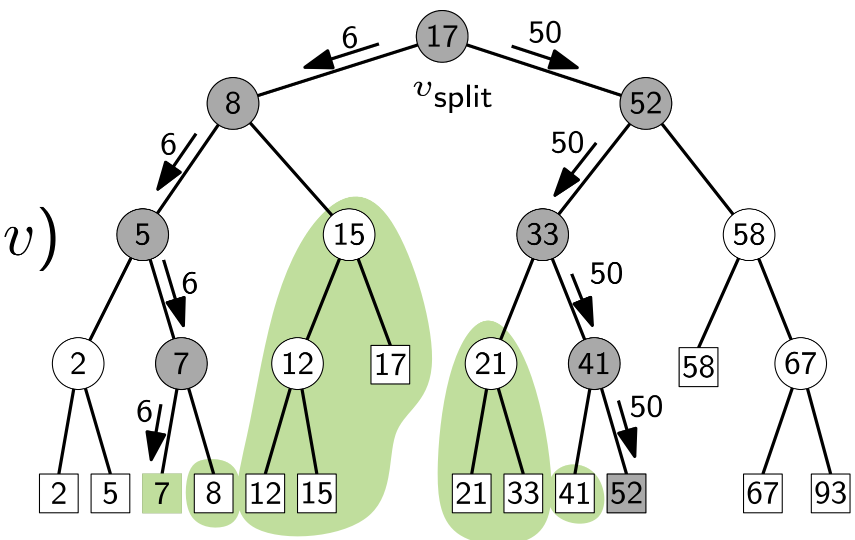
$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`

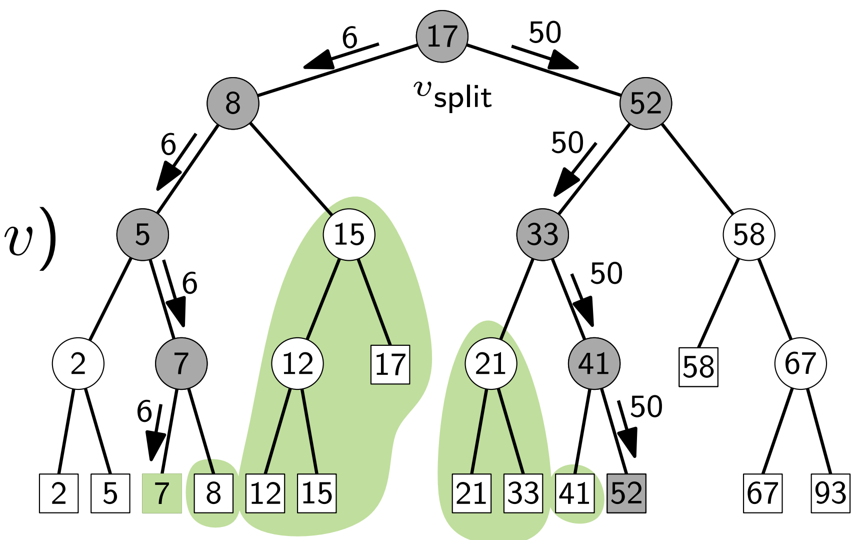


Theorem 1: A set of n points in $\mathbb{R}$ can be preprocessed in $O(n \log n)$ time and $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points.

Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \le x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



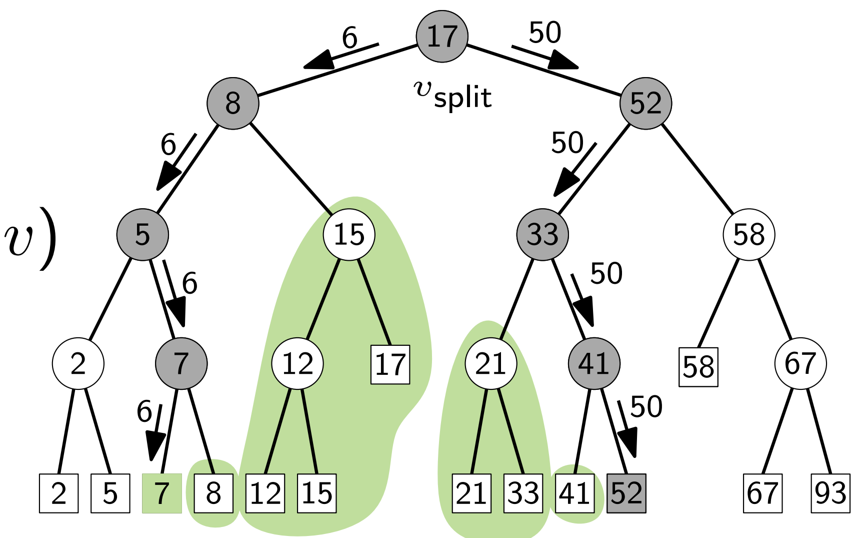
standard BST

Theorem 1: A set of n points in $\mathbb{R}$ can be preprocessed in $O(n \log n)$ time and $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points.

Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



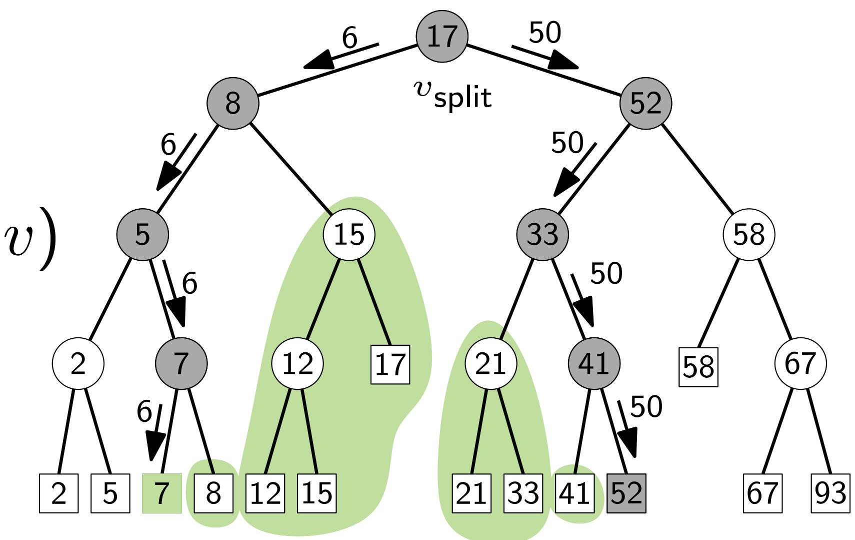
standard BST

Theorem 1: A set of n points in $\mathbb{R}$ can be preprocessed in $O(n \log n)$ time and $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points. Output sensitive!

Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x') $O(\log n)$
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`



standard BST

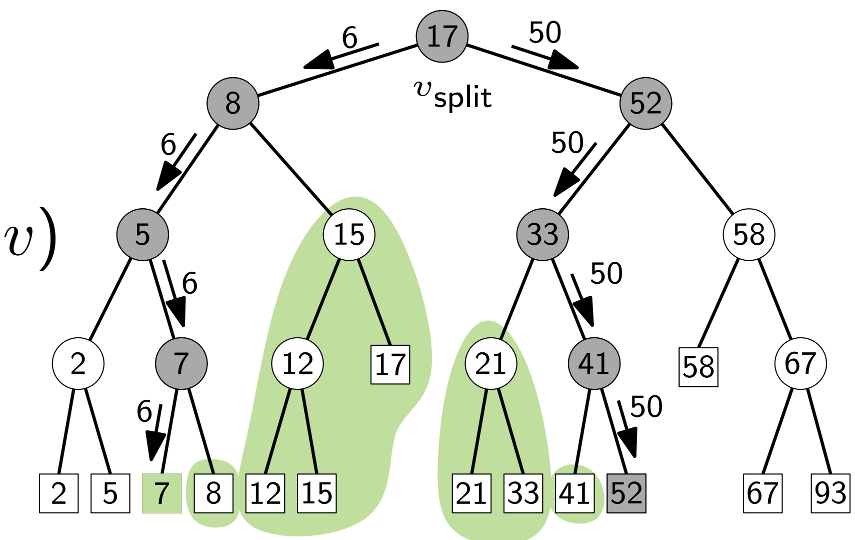
Theorem 1: A set of n points in $\mathbb{R}$ can be preprocessed in $O(n \log n)$ time and $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points. Output sensitive!

Analysis of 1dRangeQuery

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x') $O(\log n)$
if v_{split} is leaf **then** check v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree(rc(v)); $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  report v
  `// analog. for` x' `and rc(`v_{split}`)`

Proof sketch -
- two search paths of length $O(\log n)$.
- reporting subtree takes $O(k_v)$ time, where k_v is the # of leaves in the subtree of v.
- $\sum k_v \implies O(\log n + \sum k_v) = O(k \log n)$.



standard BST

Theorem 1: A set of n points in $\mathbb{R}$ can be preprocessed in $O(n \log n)$ time and $O(n)$ space so that we can answer range queries in $O(k + \log n)$ time, where k is the number of reported points. Output sensitive!

Orthogonal Range Searching for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Orthogonal Range Searching for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Orthogonal Range Searching for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Solution approaches:

- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree**
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

Orthogonal Range Searching for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Solution approaches:

- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree**
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

Temporary assumption: general position, that is, no two points have the same x- or y-coordinates

Orthogonal Range Searching for $d = 2$

Given: Set P of n points in $\mathbb{R}^2$

Goal: A data structure to efficiently answer range queries of the form $R = [x, x'] \times [y, y']$

Ideas for generalizing the 1d case?

Solution approaches:

- *one* search tree, alternate search for x and y coordinates
  $\rightarrow$ kd-**Tree** ✓
- *primary* search tree on x-coordinates,
  several *secondary* search trees on y-coordinates
  $\rightarrow$ **Range Tree**

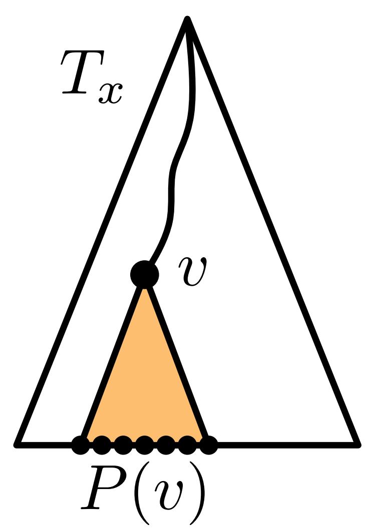
Temporary assumption: general position, that is, no two points have the same x- or y-coordinates

Range Trees

Range Trees

Idea: Use 1-dimensional search trees on two levels:

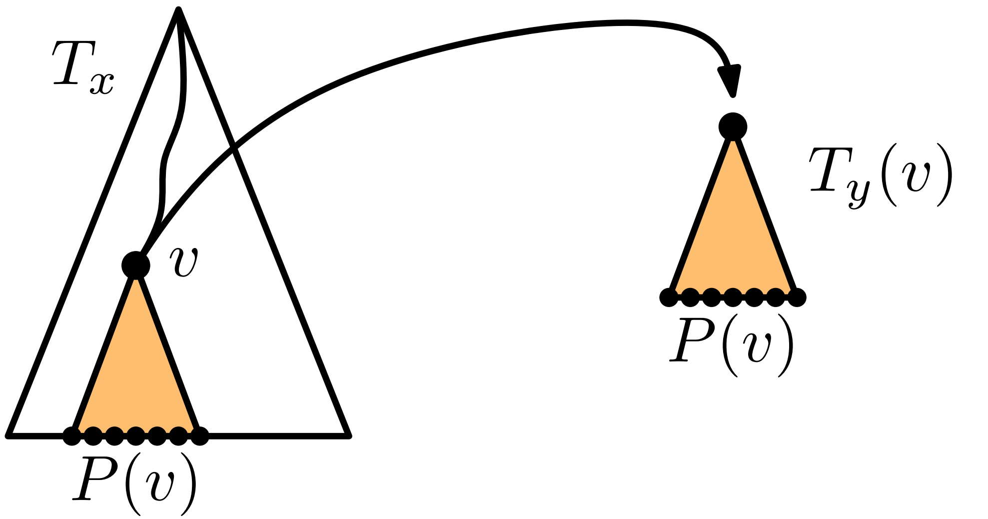
- a 1d search tree T_x on x-coordinates



Range Trees

Idea: Use 1-dimensional search trees on two levels:

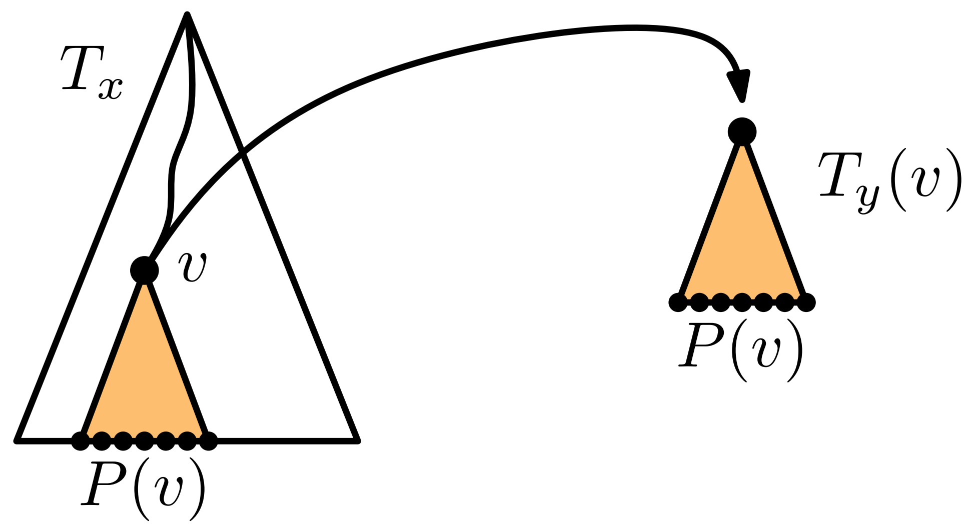
- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates



Range Trees

Idea: Use 1-dimensional search trees on two levels:

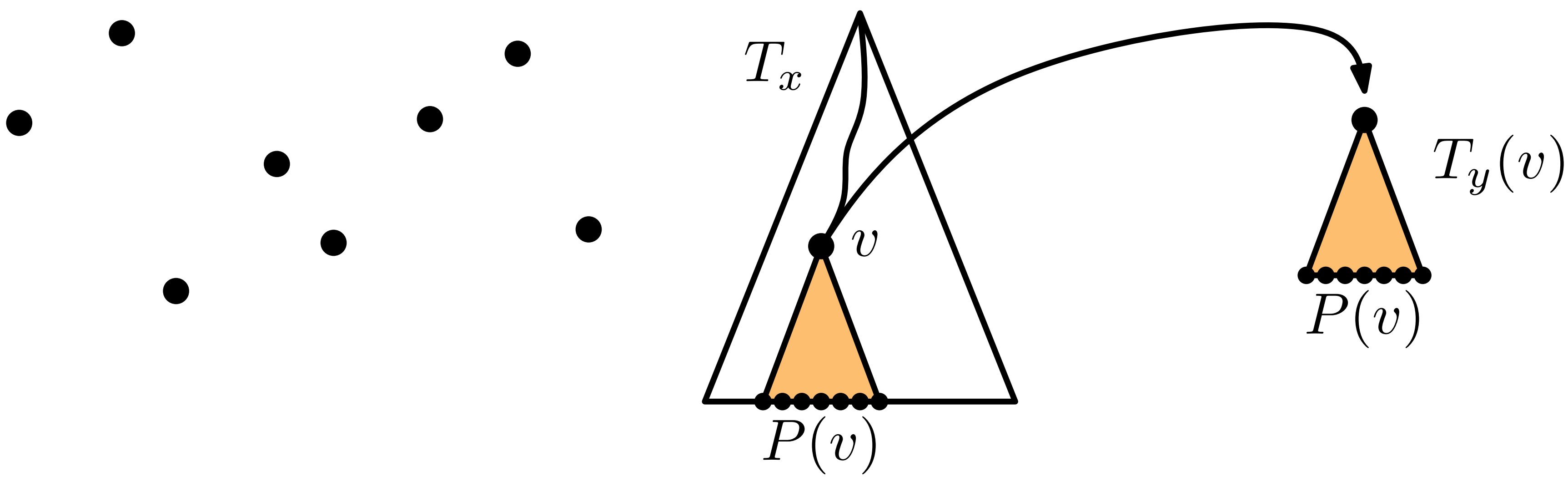
- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x



Range Trees

Idea: Use 1-dimensional search trees on two levels:

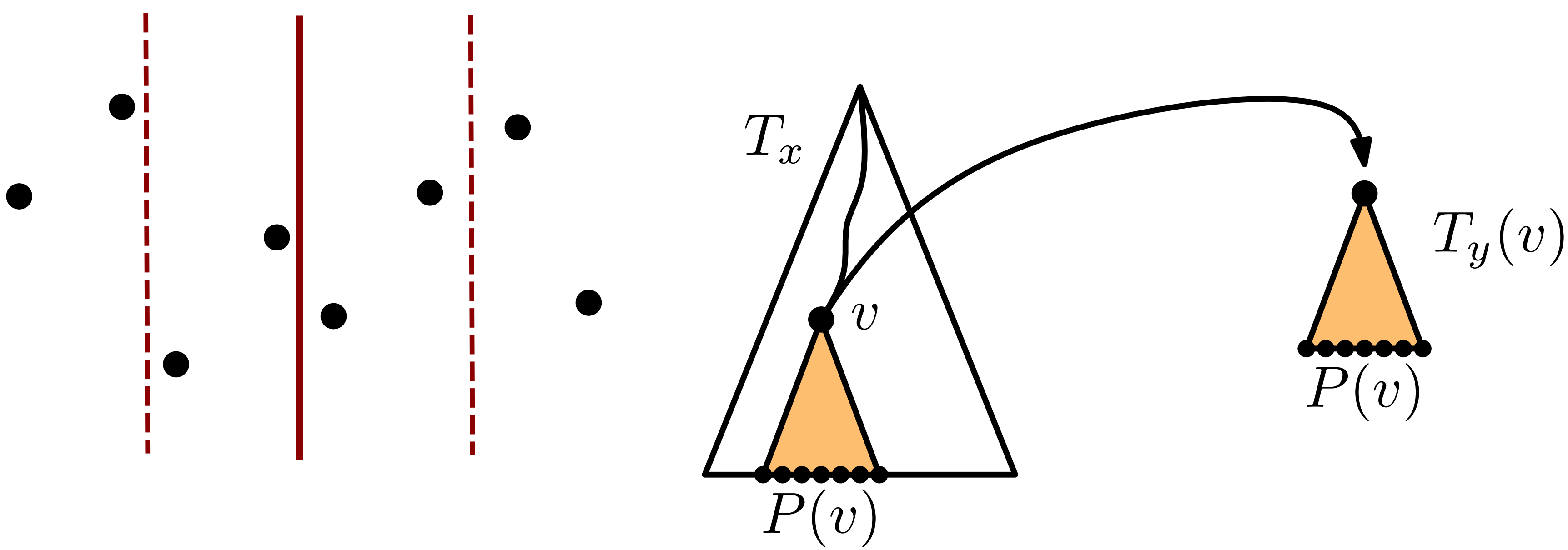
- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x



Range Trees

Idea: Use 1-dimensional search trees on two levels:

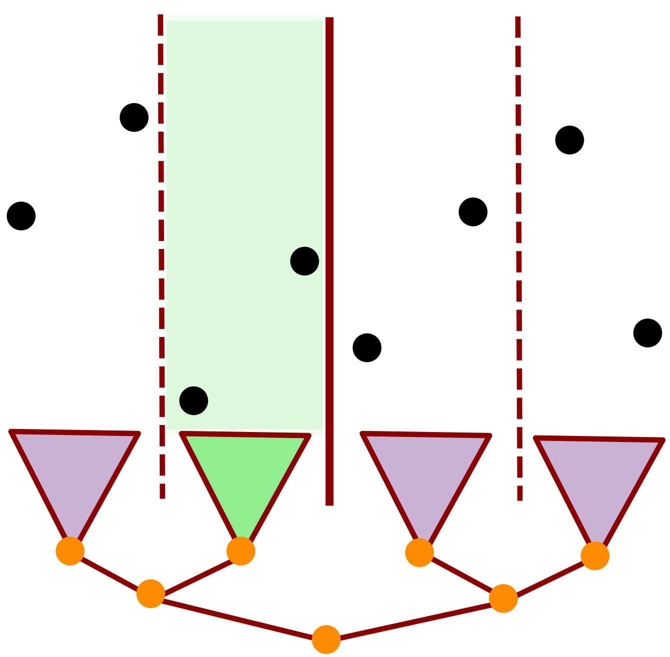
- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x

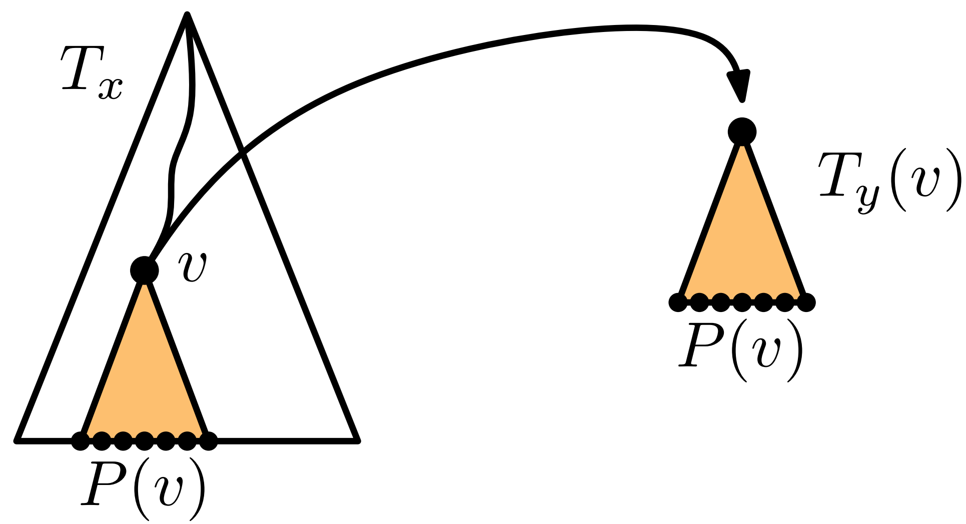


Range Trees

Idea: Use 1-dimensional search trees on two levels:

- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x

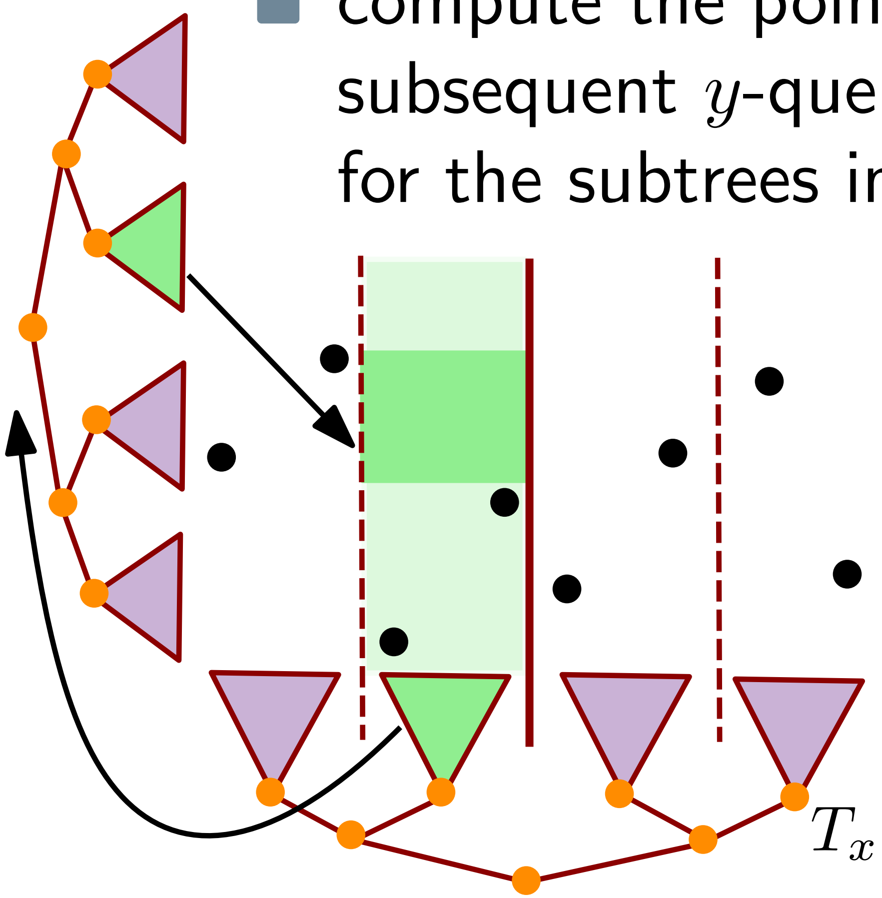


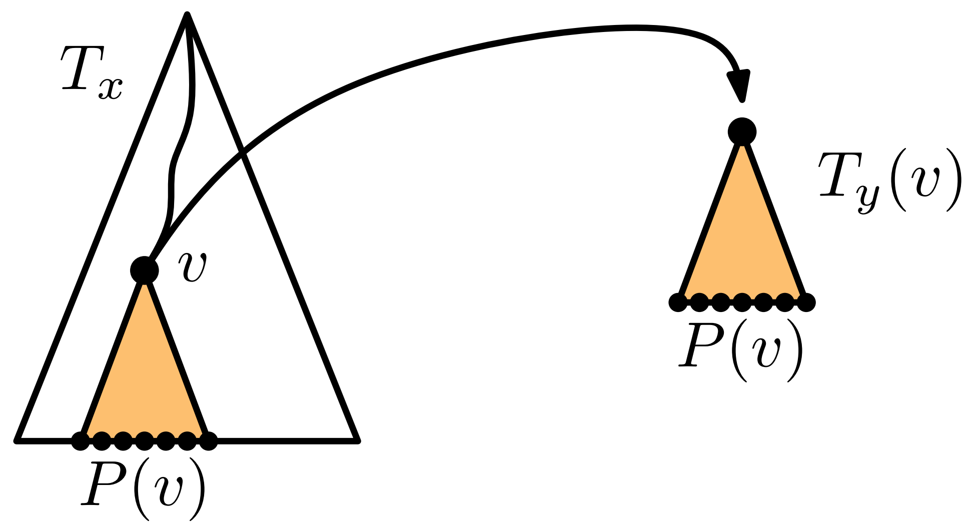


Range Trees

Idea: Use 1-dimensional search trees on two levels:

- a 1d search tree T_x on x-coordinates
- in each node v of T_x a 1d search tree $T_y(v)$ to store the canonical subset $P(v)$ based on y-coordinates
- compute the points by x-query in T_x and subsequent y-queries in the auxiliary structures T_y for the subtrees in T_x





Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**

  create leaf v for the point in P

else

  split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ and $P_2 = P \setminus P_1$

  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)

  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)

  create node v with pivot x_{median} and children v_{left} and v_{right}

$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates

return v

Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**
  create leaf v for the point in P
else
  split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ and $P_2 = P \setminus P_1$
  $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
  $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
  create node v with pivot x_{median} and children v_{left} and v_{right}
$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates
return v

(1-D search tree as before.)

Range Trees: Construction

BuildRangeTree(P)

if $|P| = 1$ **then**
    create leaf v for the point in P
else
    split P at x_{median} into $P_1 = \{p \in P \mid p_x \leq x_{\text{median}}\}$ and $P_2 = P \setminus P_1$
    $v_{\text{left}} \leftarrow$ BuildRangeTree(P_1)
    $v_{\text{right}} \leftarrow$ BuildRangeTree(P_2)
    create node v with pivot x_{median} and children v_{left} and v_{right}
$T_y(v) \leftarrow$ binary search tree for P w.r.t y-coordinates
return v

(1-D search tree as before.)

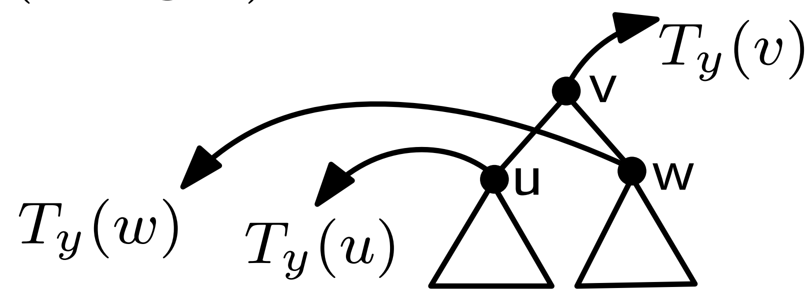
Task: How much space and runtime does BuildRangeTree use?

Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.



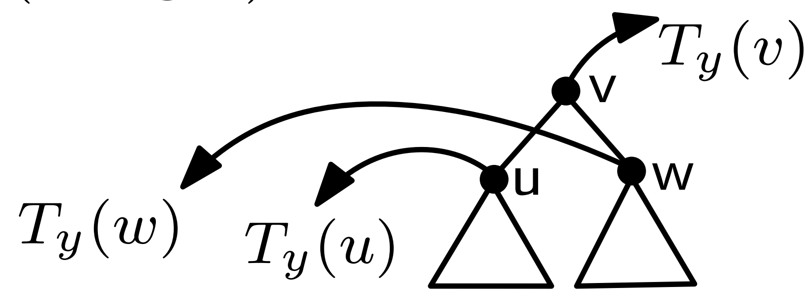
Proof:

Space -

- Tree T_x uses $O(n)$ space.
- Canonical subsets of each level of T_x are a partition of P.
- Each point $p \in P$ appears once per level and all trees T_y of one level need together $O(n)$ space.
- T_x has $O(\log n)$ levels $\implies$ $O(n \log n)$ space.

Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

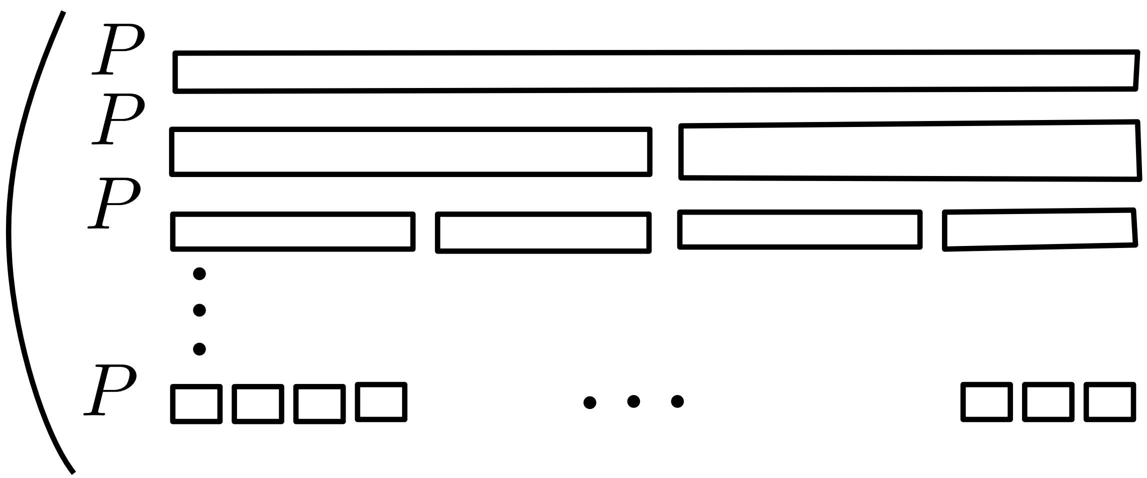


Proof:

Space -

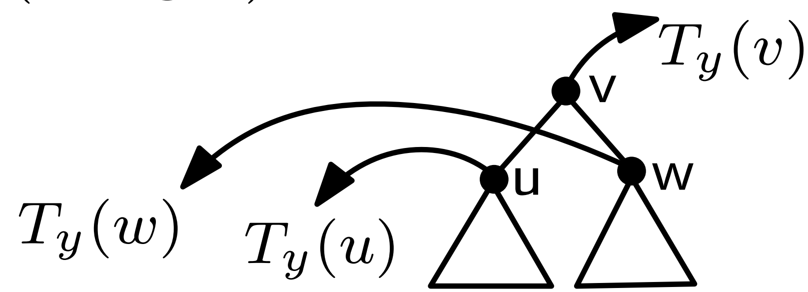
- Tree T_x uses $O(n)$ space.
- Canonical subsets of each level of T_x are a partition of P.
- Each point $p \in P$ appears once per level and all trees T_y of one level need together $O(n)$ space.
- T_x has $O(\log n)$ levels $\implies$ $O(n \log n)$ space.

$O(\log n)$ times.



Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

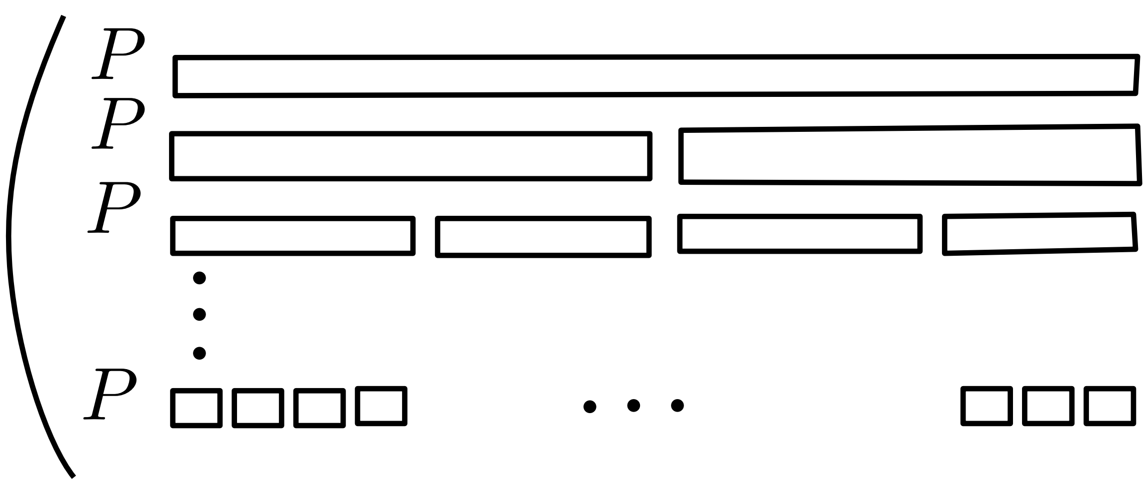


Proof:

Space -

- Tree T_x uses $O(n)$ space.
- Canonical subsets of each level of T_x are a partition of P.
- Each point $p \in P$ appears once per level and all trees T_y of one level need together $O(n)$ space.
- T_x has $O(\log n)$ levels $\implies$ $O(n \log n)$ space.

$O(\log n)$ times.



Note: we have $O(n)$ secondary trees, but, total space is $O(n \log n)$.

Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

Time -

- Constructing T_x takes $O(n \log n)$ time as before.
- For the T_y' pre-sort P by y-coordinates once in $O(n \log n)$ time.
- From this sorted list build $T_y(v)$ bottom-up in $O(n)$ time for the root v of T_x.
- Filter the sorted list into sorted lists of subsets P_1 and P_2 and construct $T_y(u)$, $T_y(w)$ again in $O(n)$ time per level for $u = lc(v)$, $w = rc(v)$; then recurse.
- $O(\log n)$ levels $\implies$ $O(n \log n)$ time in total.

Range Trees: Construction

Lemma 3: A Range Tree for n points in $\mathbb{R}^2$ uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

Time -

- Constructing T_x takes $O(n \log n)$ time as before.
- For the T_y' pre-sort P by y-coordinates once in $O(n \log n)$ time.
- From this sorted list build $T_y(v)$ bottom-up in $O(n)$ time for the root v of T_x.

per level - $O(n)$ time.

- Filter the sorted list into sorted lists of subsets P_1 and P_2 and construct $T_y(u)$, $T_y(w)$ again in $O(n)$ time per level for $u = lc(v)$, $w = rc(v)$; then recurse.
- $O(\log n)$ levels $\implies$ $O(n \log n)$ time in total.

Range Queries in a Range Tree

Reminder:

1dRangeQuery(T, x, x')

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** check v_{split}
else
    $v \leftarrow$ lc(v_{split})
    while v not leaf **do**
        if $x \leq x_v$ **then**
            ReportSubtree(rc(v))
            $v \leftarrow$ lc(v)
        else $v \leftarrow$ rc(v)
    report v
    `//` `analogous for` x' `and rc(`v_{split}`)`

Range Queries in a Range Tree

Reminder:

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery(**$T, [x, x'] \times [y, y']$**)**

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** ~~check v_{split}~~ **if** $v_{\text{split}} \in [x, x'] \times [y, y']$ **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not leaf **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$
      $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  ~~report v~~ **if** $v \in [x, x'] \times [y, y']$ **then** report v
  `// analogous for` x' `and rc(`v_{split}`)`

Range Queries in a Range Tree

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery**$(T, [x, x'] \times [y, y'])$

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** ~~check v_{split}~~ **if** $v_{\text{split}} \in [x, x'] \times [y, y']$ **then** report v_{split}
else
$v \leftarrow$ lc(v_{split})
while v not leaf **do**
if $x \leq x_v$ **then**
~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$
$v \leftarrow$ lc(v)
else $v \leftarrow$ rc(v)
~~report v~~ **if** $v \in [x, x'] \times [y, y']$ **then** report v
`//` `analogous for` x' `and rc(`v_{split}`)`

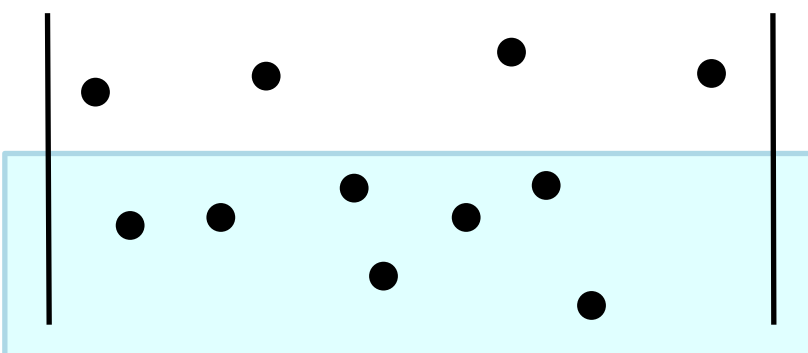
Lemma 4: A range query in a Range Tree takes $O(\log^2 n + k)$ time, where k is the number of reported points.

Range Queries in a Range Tree

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery**$(T, [x, x'] \times [y, y'])$

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is leaf **then** ~~check v_{split}~~ **if** $v_{\text{split}} \in [x, x'] \times [y, y']$ **then** report v_{split}
else
  $v \leftarrow$ lc(v_{split})
  while v not leaf **do**
    if $x \leq x_v$ **then**
      ~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$
      $v \leftarrow$ lc(v)
    else $v \leftarrow$ rc(v)
  ~~report v~~ **if** $v \in [x, x'] \times [y, y']$ **then** report v
  `// analogous for` x' `and rc(`v_{split}`)`

Lemma 4: A range query in a Range Tree takes $O(\log^2 n + k)$ time, where k is the number of reported points.



Cannonical subset in x-dim.

Range Queries in a Range Tree

~~**1dRangeQuery**(T, x, x')~~ **2dRangeQuery**$(T, [x, x'] \times [y, y'])$

$v_{\text{split}} \leftarrow$ FindSplitNode(T, x, x')

if v_{split} is leaf **then** ~~check v_{split}~~ **if** $v_{\text{split}} \in [x, x'] \times [y, y']$ **then** report v_{split}

else

$v \leftarrow$ lc(v_{split})

while v not leaf **do**

if $x \leq x_v$ **then**

~~ReportSubtree(rc(v))~~ 1dRangeQuery$(T_y(\text{rc}(v)), y, y')$

$v \leftarrow$ lc(v)

else $v \leftarrow$ rc(v)

~~report v~~ **if** $v \in [x, x'] \times [y, y']$ **then** report v

// analogous for x' and rc(v_{split})

Lemma 4: A range query in a Range Tree takes $O(\log^2 n + k)$ time, where k is the number of reported points.

- each visited node v in T_x: $O(1) + 1D$-RangeQuery $(O(k_v + \log n))$ ... recall $k_v = |p_v|$.
- $\sum O(k_v + \log n) = \underbrace{\sum O(k_v)}_{O(k)} + \sum O(\log n) = O(k) + O(\log^2 n)$.

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A	3	10	19	23	30	37	59	62	70	80	100	105

B	10	19	30	62	70	80	100

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A											
3	10	19	23	30	37	59	62	70	80	100	105

B						
10	19	30	62	70	80	100

Search interval [20,65]

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays

A	3	10	19	23	30	37	59	62	70	80	100	105

Can we do better than two binary searches?

B	10	19	30	62	70	80	100

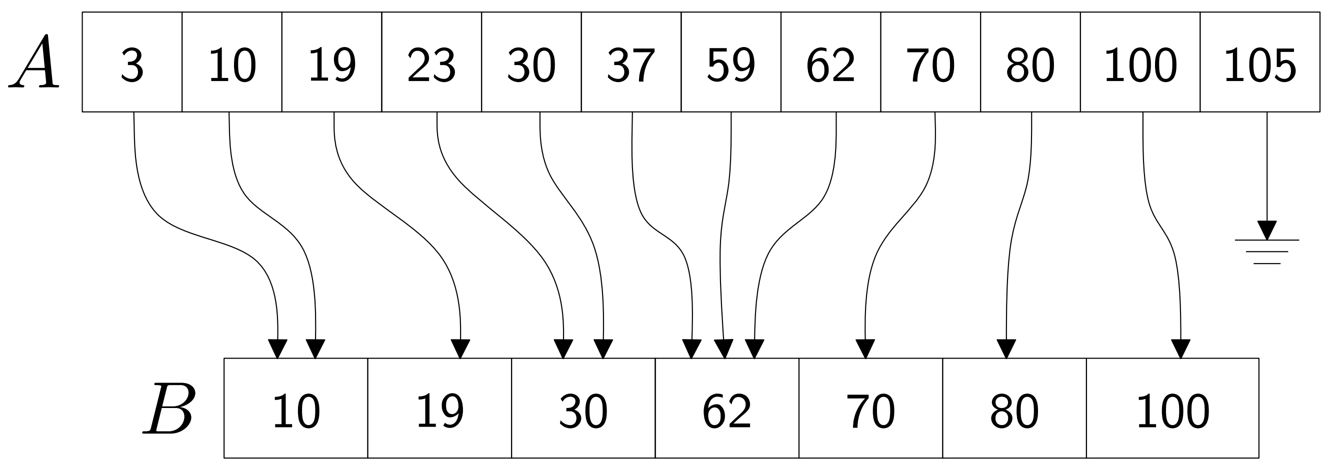
Search interval [20,65]

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



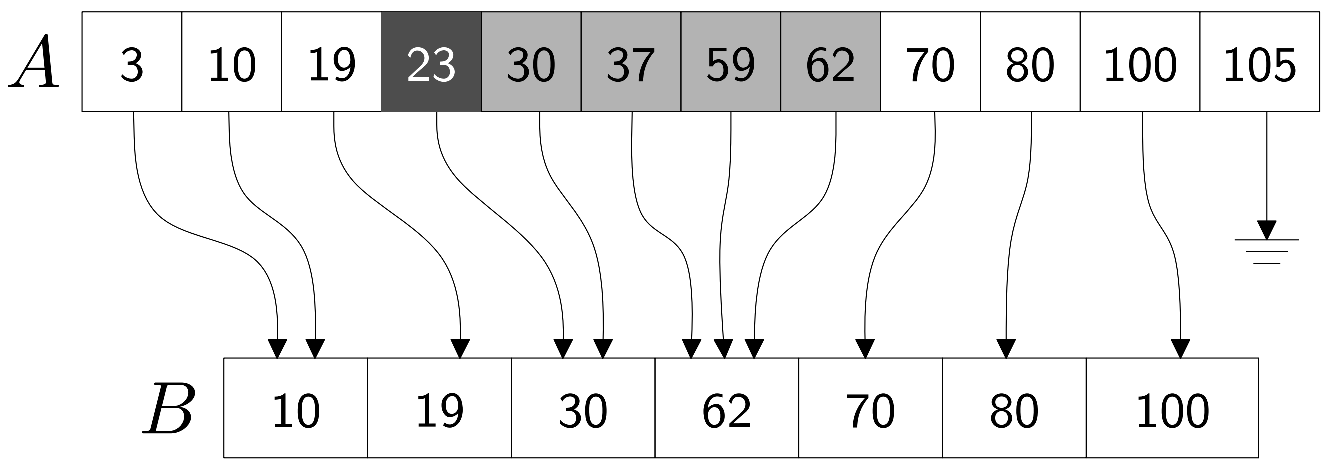
link $a \in A$ with smallest $b \geq a$ in B

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



link $a \in A$ with smallest $b \geq a$ in B

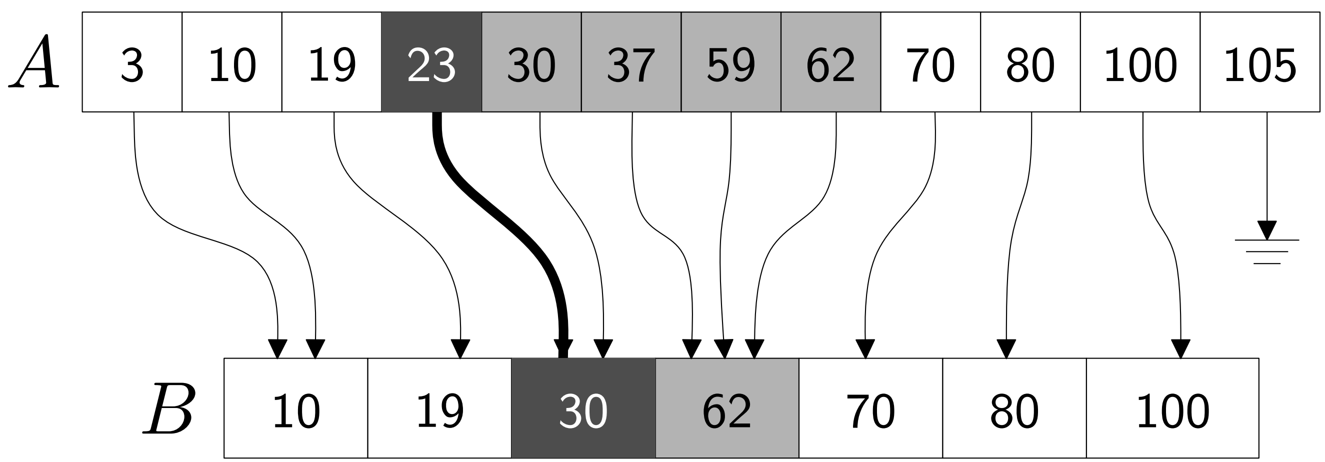
Search interval [20,65]

Speed-up with Fractional Cascading

Observation: Range queries in a Range Tree perform $O(\log n)$ 1d range queries, each taking $O(\log n + k_v)$ time. The query interval $[y, y']$ is always the same!

Idea: Use this property to accelerate the 1d queries to $O(1 + k_v)$ time

Example: Two sets $B \subseteq A \subseteq \mathbb{R}$ in sorted arrays



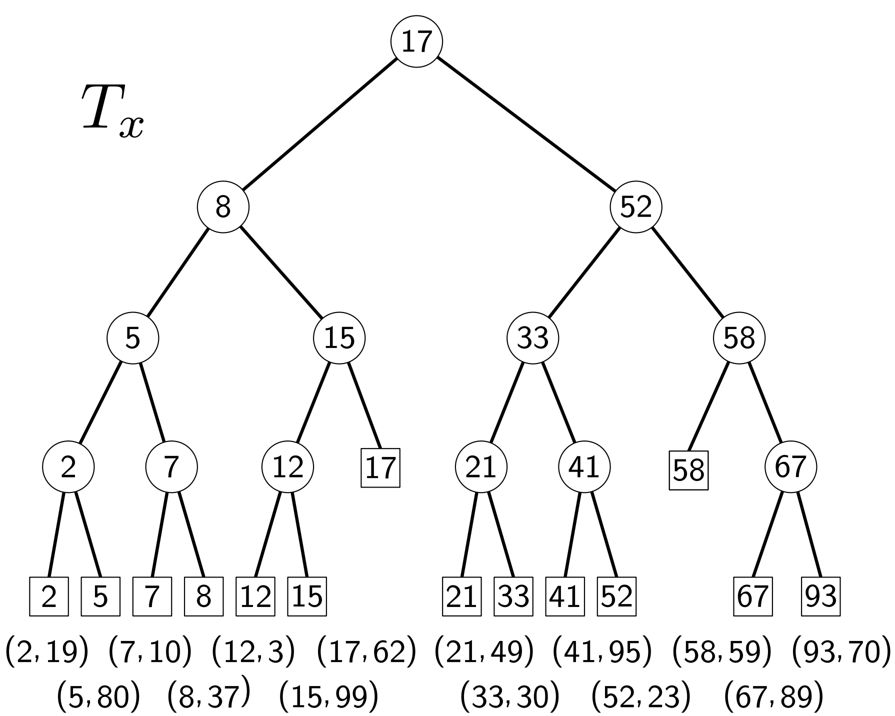
link $a \in A$ with smallest $b \geq a$ in B

Search interval [20,65]

Pointer yields starting point for second search in $O(1)$ time

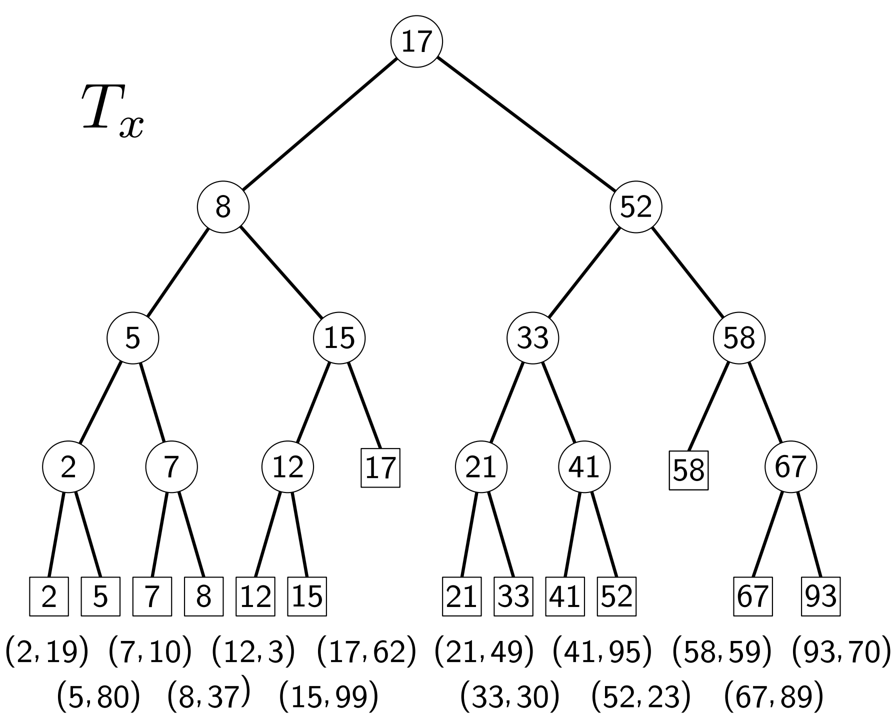
Speed-up with Fractional Cascading

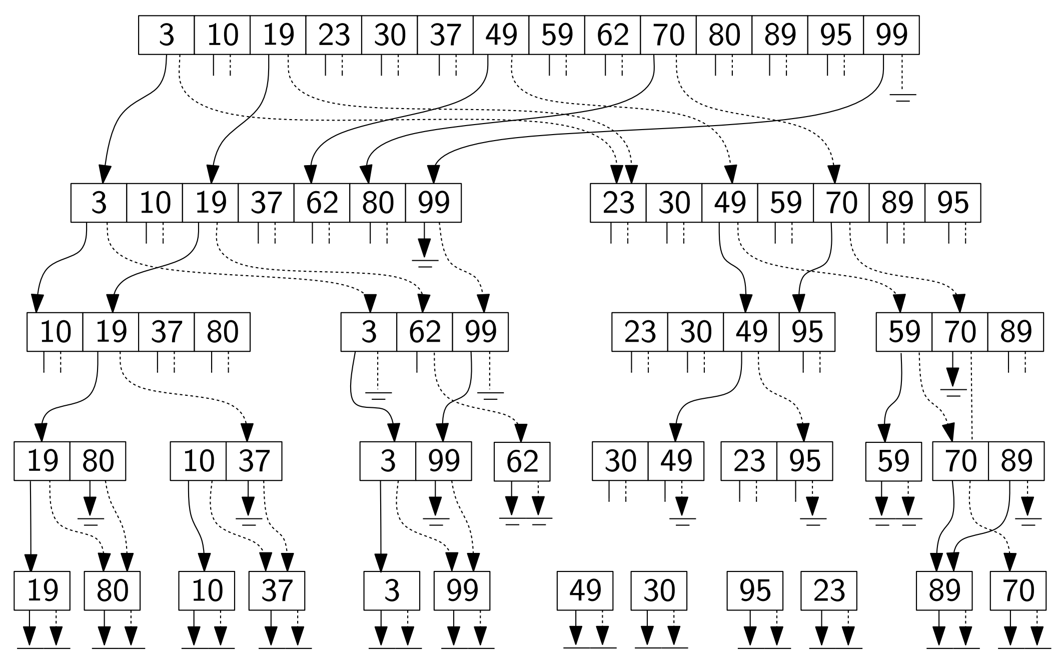
- In Range Trees we have $P(\mathsf{lc}(v)) \subseteq P(v)$ and $P(\mathsf{rc}(v)) \subseteq P(v)$ for the canonical subsets.



Speed-up with Fractional Cascading

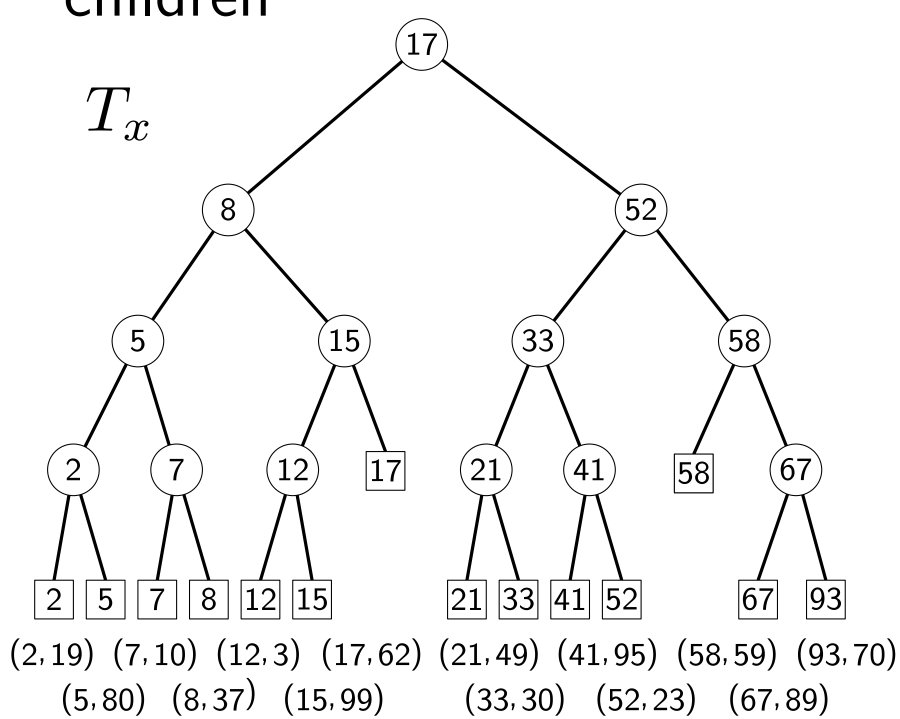
- In Range Trees we have $P(\mathrm{lc}(v)) \subseteq P(v)$ and $P(\mathrm{rc}(v)) \subseteq P(v)$ for the canonical subsets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathrm{lc}(v))$ and $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**

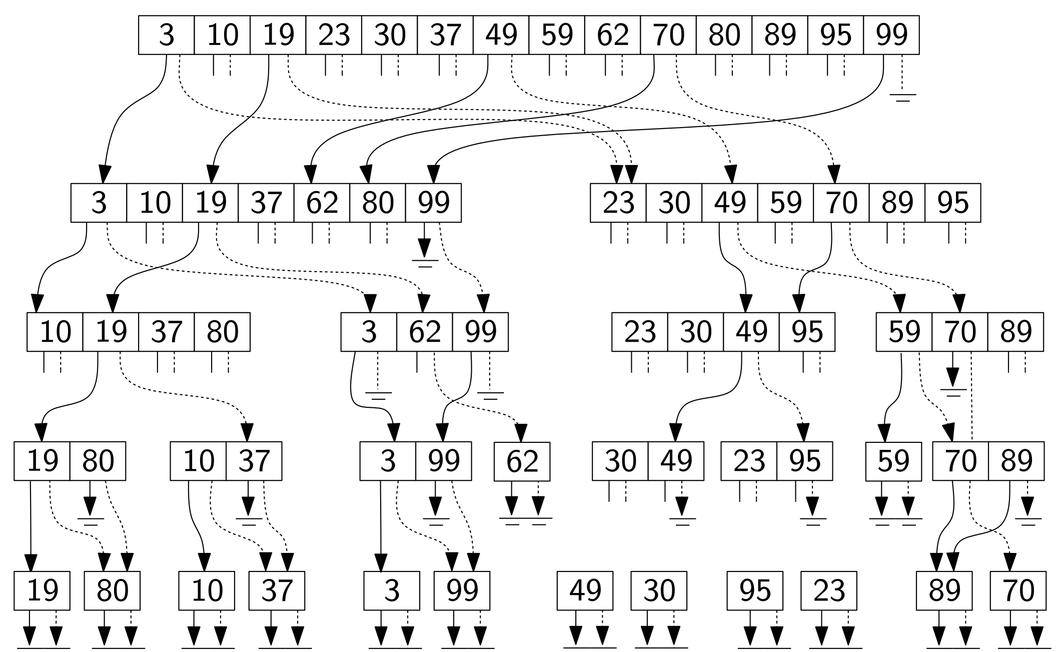




Speed-up with Fractional Cascading

- In Range Trees we have $P(\mathrm{lc}(v)) \subseteq P(v)$ and $P(\mathrm{rc}(v)) \subseteq P(v)$ for the canonical subsets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathrm{lc}(v))$ and $A(\mathrm{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- Only in the split node the first binary search takes $O(\log n)$ time, then it takes $O(1)$ time to follow the pointers when descending into the children





Speed-up with Fractional Cascading

- In Range Trees we have $P(\mathsf{lc}(v)) \subseteq P(v)$ and $P(\mathsf{rc}(v)) \subseteq P(v)$ for the canonical subsets.
- Define for each array element $A(v)[i]$ two pointers into the arrays $A(\mathsf{lc}(v))$ and $A(\mathsf{rc}(v))$
  $\rightarrow$ **Layered Range Tree**
- Only in the split node the first binary search takes $O(\log n)$ time, then it takes $O(1)$ time to follow the pointers when descending into the children

Theorem 2: A Layered Range Tree on n points in $\mathbb{R}^2$ can be constructed in $O(n \log n)$ time and space. Range queries take $O(\log n + k)$ time, where k is the number of reported points.

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

$$p = (p_x, p_y)$$

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Rectangle $R = [x, x'] \times [y, y']$

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Rectangle $R = [x, x'] \times [y, y']$

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x|-\infty), (x'|+\infty)] \times [(y|-\infty), (y'|+\infty)]$$

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x|-\infty), (x'|+\infty)] \times [(y|-\infty), (y'|+\infty)]$$

Then:

Arbitrary Point Sets

So far: points in general position, that is no two points have the same x- or y-coordinate

Idea: Use pairs of numbers $(a|b)$ with lexicographic order (algorithm only assumes totally ordered set)

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

unique coordinates

Rectangle $R = [x, x'] \times [y, y']$

$$\downarrow$$

$$\hat{R} = [(x| - \infty), (x'| + \infty)] \times [(y| - \infty), (y'| + \infty)]$$

Then: $p \in R \ \Leftrightarrow\ \hat{p} \in \hat{R}$

Summary

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing		
Space		
Query time		

Summary

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^2 n + k)$

Summary

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Fractional Cascading

Summary

Given: n points in $\mathbb{R}^d$

Output: data structure that efficiently answers queries of the form $[a_1, b_1] \times \cdots \times [a_d, b_d]$

$\rightarrow$ We have seen two alternatives

	kd-Tree	Range Tree
Preprocessing	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n \log n)$
Query time	$O(\sqrt{n} + k)$	$O(\log^{\cancel{2}} n + k)$

Discussion

How can the data structures generalize to d dimensions?

Discussion

How can the data structures generalize to d dimensions?

- kd-Trees extend easily by dividing the points alternately in the d coordinates. Space remains $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.

Discussion

How can the data structures generalize to d dimensions?

- kd-Trees extend easily by dividing the points alternately in the d coordinates. Space remains $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Higher-dimensional Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space grows to $O(n \log^{d-1} n)$; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Discussion

How can the data structures generalize to d dimensions?

- kd-Trees extend easily by dividing the points alternately in the d coordinates. Space remains $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Higher-dimensional Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space grows to $O(n \log^{d-1} n)$; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Is it possible to query for other objects (e.g., polygons) with these data structures?

Discussion

How can the data structures generalize to d dimensions?

- kd-Trees extend easily by dividing the points alternately in the d coordinates. Space remains $O(n)$, construction $O(n \log n)$ and the query time is $O(n^{1-1/d} + k)$.
- Higher-dimensional Range Trees can be built recursively: the auxiliary search tree on the first coordinate is a $(d-1)$-dimensional Range Tree. The construction and space grows to $O(n \log^{d-1} n)$; a query takes $O(\log^d n + k)$ time, and with fractional cascading, $O(\log^{d-1} n + k)$ time.

Is it possible to query for other objects (e.g., polygons) with these data structures?

Yes, range searching for other objects (e.g. polygons) can be reduced to higher-dimensional range searching for points (see exercise).
Queries for objects that are only partially contained in the query range will be covered in the next lecture.

Extension: Dynamic Range Queries

Question: Can we adapt the data structures for dynamic point sets?

- Inserting points
- Removing points

Extension: Dynamic Range Queries

Question: Can we adapt the data structures for dynamic point sets?

- Inserting points
- Removing points

1) **Divided kd-trees** [van Kreveld, Overmars '91]
support updates in $O(\log n)$ time, but the query time is $O(\sqrt{n \log n} + k)$

Extension: Dynamic Range Queries

Question: Can we adapt the data structures for dynamic point sets?

- Inserting points
- Removing points

1) **Divided kd-trees** [van Kreveld, Overmars '91]
support updates in $O(\log n)$ time, but the query time is $O(\sqrt{n \log n} + k)$

2) **Augmented dynamic range trees** [Mehlhorn, Näher '90]
support updates in $O(\log n \log \log n)$ time and queries in $O(\log n \log \log n + k)$ time