2IL50 Data Structures

2023-24 Q3

Lecture 9: Range Searching

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Augmenting data structures

Methodology for augmenting a data structure

1. Choose an underlying data structure.
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

You don’t need to do these steps in strict order!

Red-black trees are very well suited to augmentation …

Augmenting red-black trees

Theorem [RB-tree Augmentation]

Augment an RB-tree with field f, where $x.f$ depends only on information in x, $x.\mathrm{left}$, and $x.\mathrm{right}$ (including $x.\mathrm{left}.f$ and $x.\mathrm{right}.f$).
Then we can maintain values of f in all nodes during insert and delete without affecting $O(\log n)$ performance.

When we alter information in x, changes propagate only upward on the search path for x ...

Examples

- OS-tree
  new operations: OS-Select and OS-Rank
- Interval-tree
  new operation: Interval-Search

Range Searching

Application: Database queries

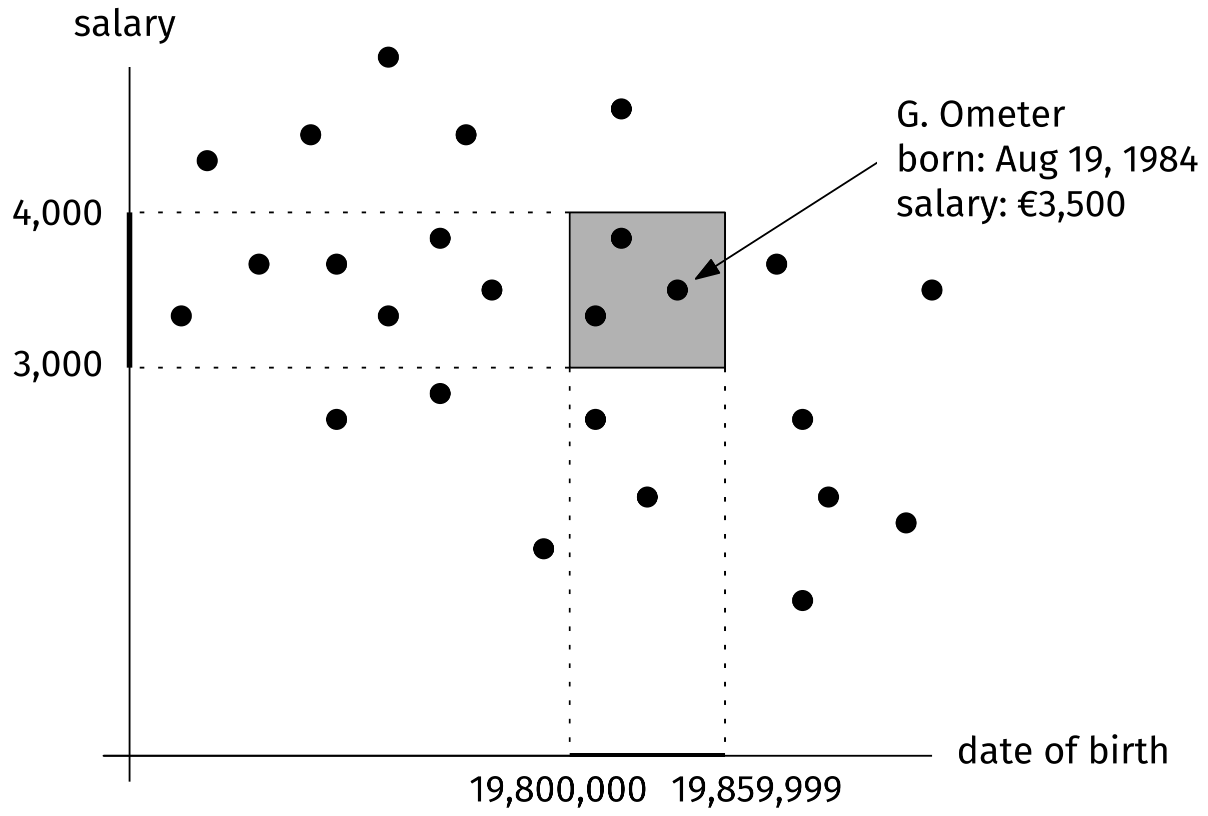
Example: Database for personnel administration
(name, address, date of birth, salary, ...)

Query: Report all employees
born between 1980 and 1985 who earn
between €3,000 and €4,000 per month.

More parameters?

Report all employees
born between 1980 and 1985 who earn
between €3000 and €4000 per month
and have between two and four children.

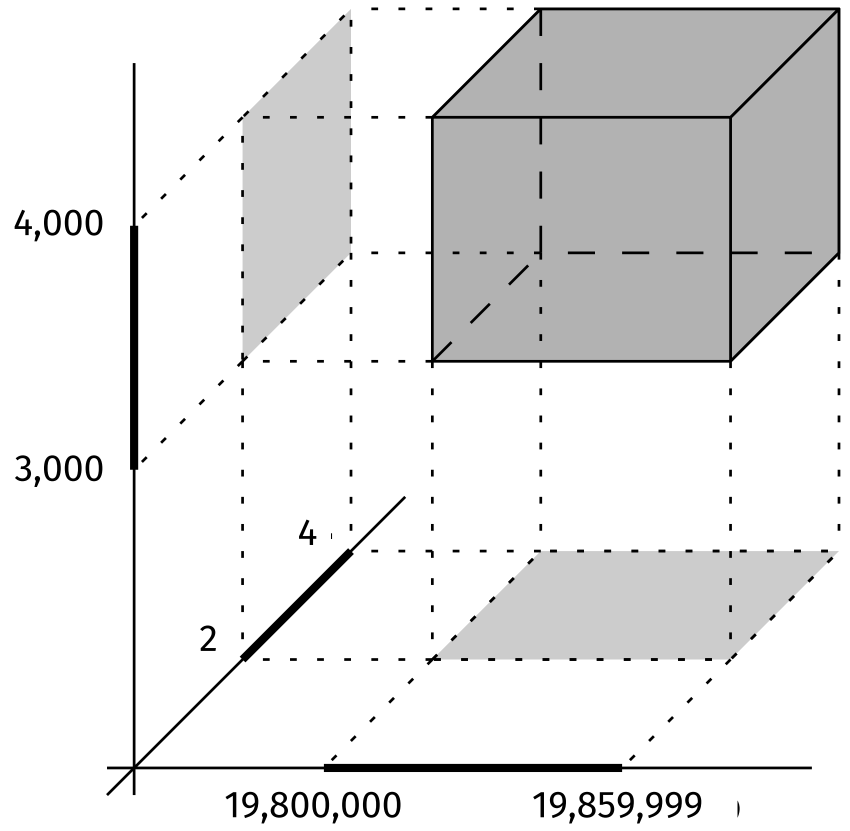
➡ more dimensions



Application: Database queries

Report all employees
born between 1980 and 1985 who earn
between €3,000 and €4,000 per month
and have between two and four children.

Rectangular range query
or
orthogonal range query



1-Dimensional range searching

$P = \{p_1, p_2, \ldots, p_n\}$ set of points on the real line

Query: given a query interval $[x : x']$ report all $p_i \in P$ with $p_i \in [x : x']$.

Solution: Use a balanced binary search tree T.

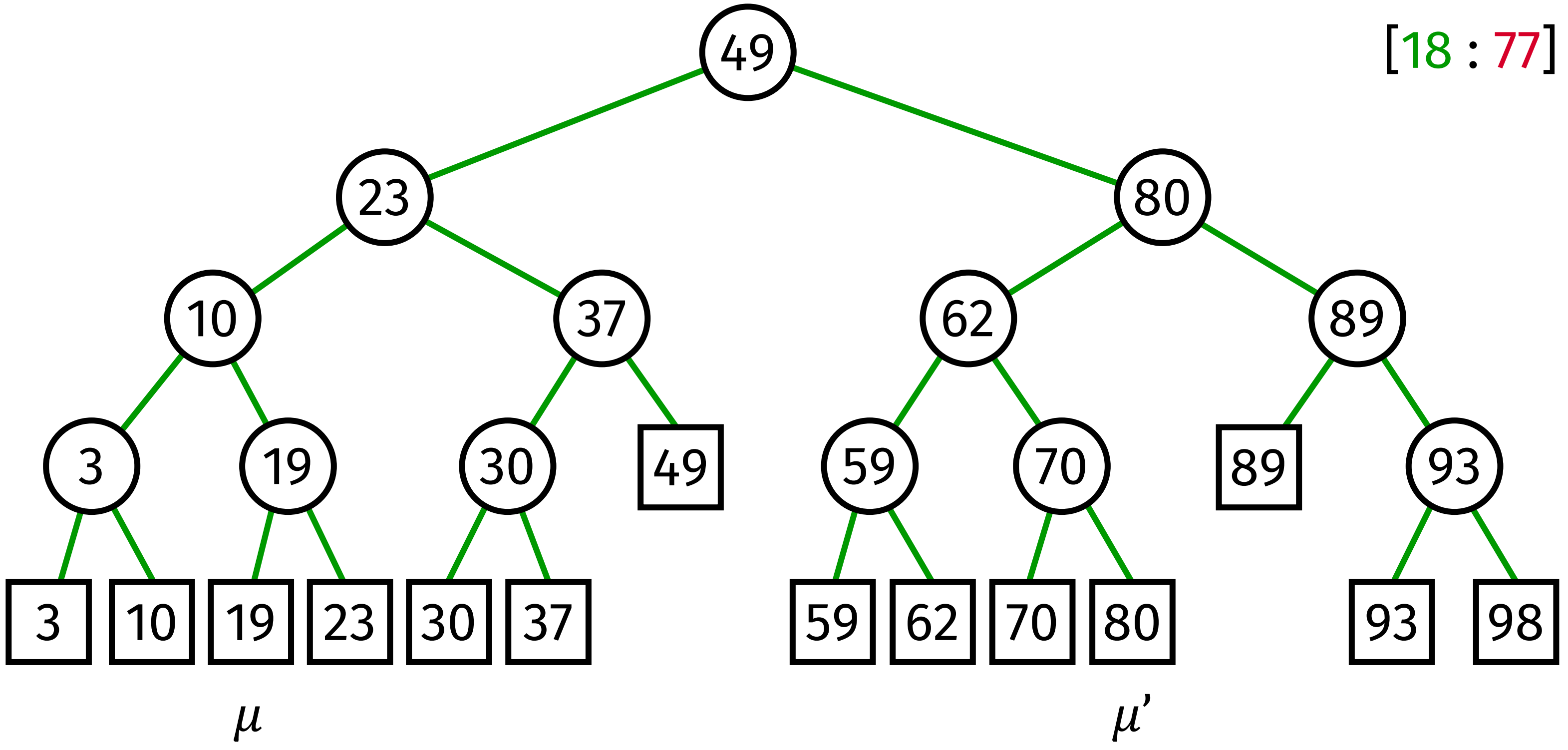
- leaves of T store the points p_i
- internal nodes store splitting values (node v stores value x_v)

1-Dimensional range searching

Query $[x : x']$ ➡ search with x and x' ➡ end in two leaves μ and μ'

Report 1. all leaves between μ and μ'

2. possibly points stored at μ and μ'



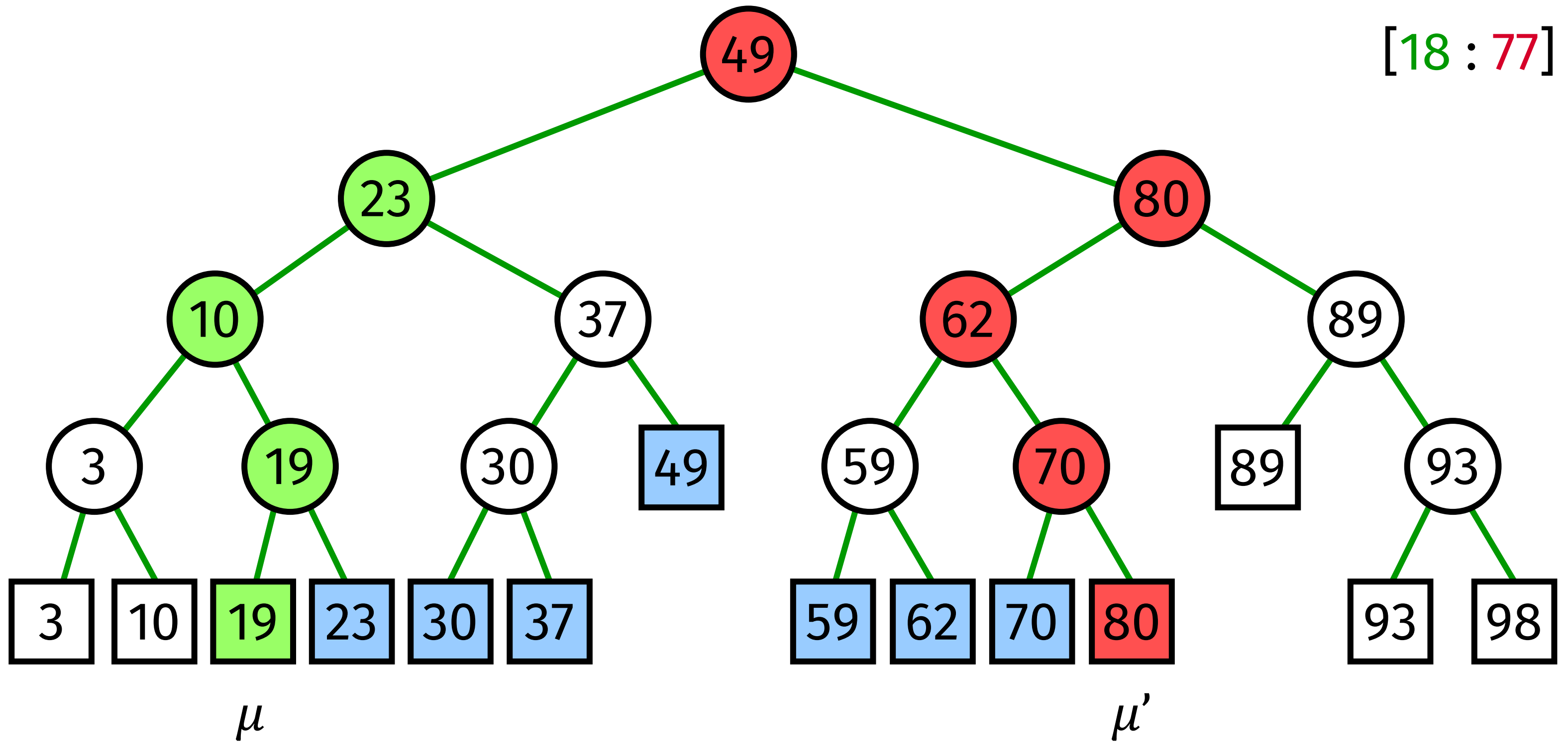
1-Dimensional range searching

Query $[x : x']$ ➡ search with x and x' ➡ end in two leaves μ and μ'

Report 1. all leaves between μ and μ'

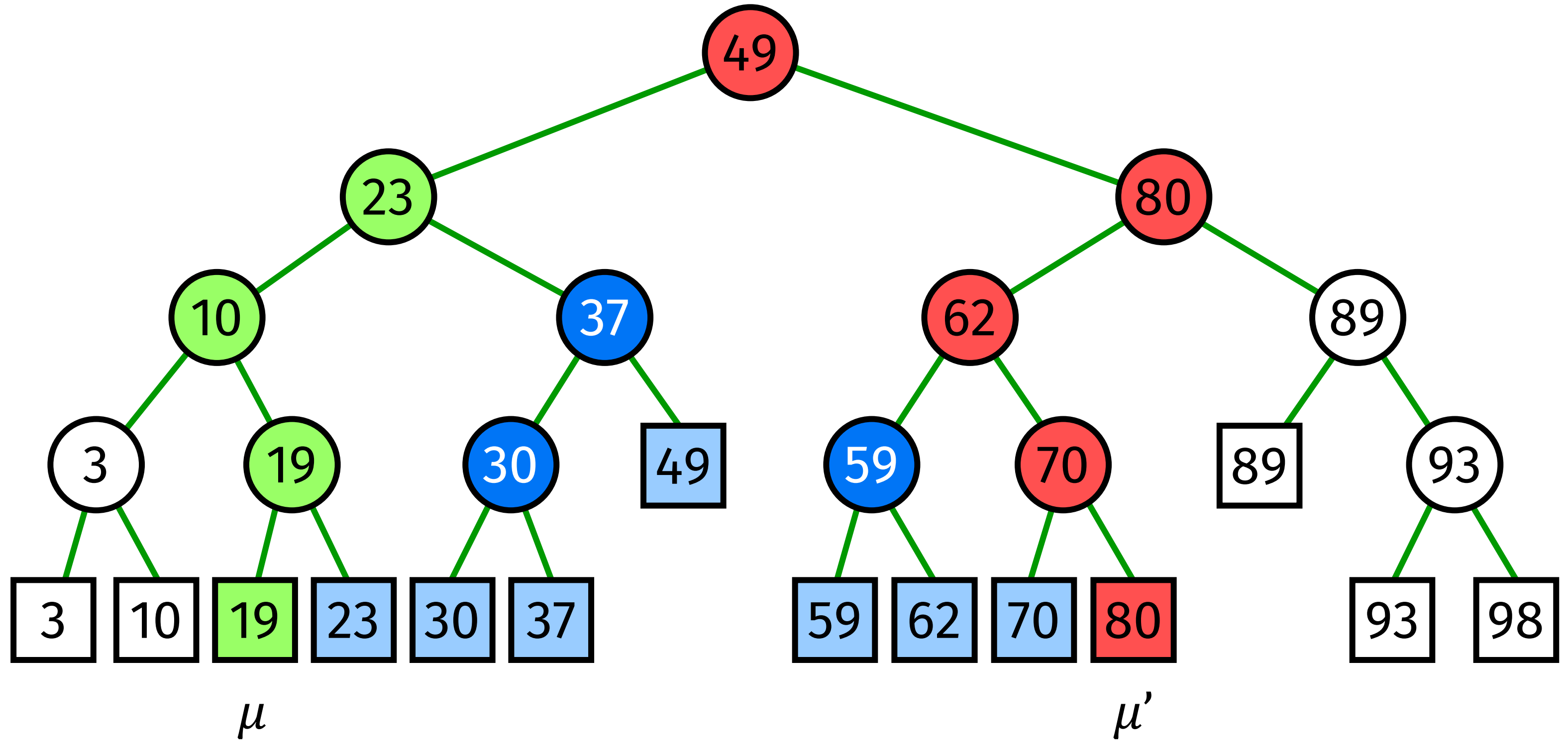
2. possibly points stored at μ and μ'

How do we find all leaves between μ and μ'?



1-Dimensional range searching

How do we find all leaves between μ and μ'?

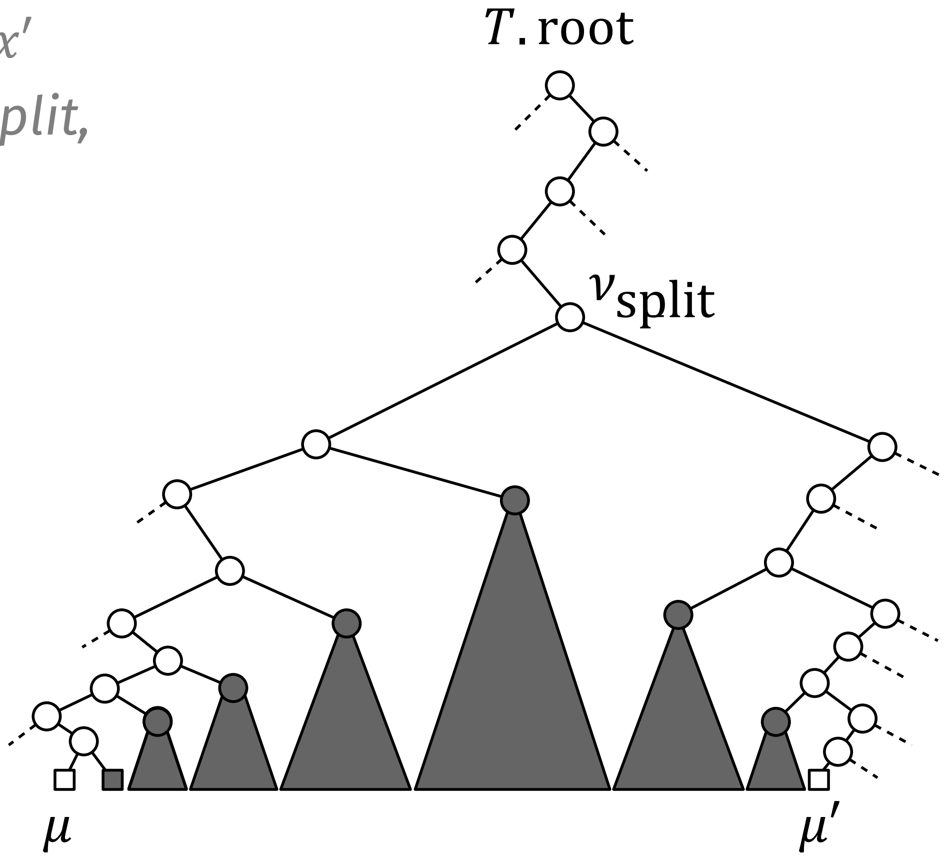


Solution: They are the leaves of the subtrees rooted at nodes v in between the two search paths whose parents are on the search paths.

➡ we need to find the node v_{split} where the search paths split

1-Dimensional range searching

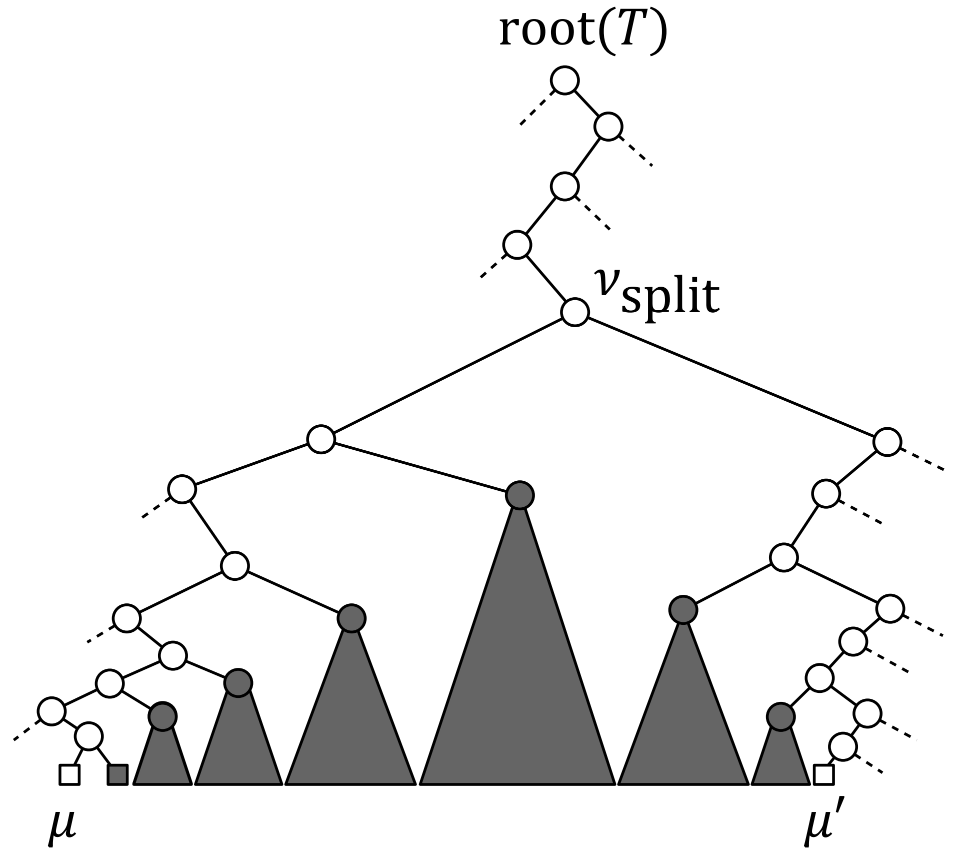
```
FindSplitNode($T$, $x$, $x'$)
    // Input: a tree $T$ and two values $x$ and $x'$ with $x \le x'$
    // Output: the node $v$ where the paths to $x$ and $x'$ split,
    // or the leaf where both paths end
1   $v = T.\mathrm{root}$
2   while $v$ is not a leaf and $(x' \le x_v$ or $x > x_v)$
3       if $x' \le x_v$
4           $v = v.\mathrm{left}$
5       else
6           $v = v.\mathrm{right}$
7   return $v$
```



1-Dimensional range searching

1. Starting from ν_{split} follow the search path to x.
   - when the paths goes left,
     report all leaves in the right subtree
   - check if μ ∈ $[x : x']$

2. Starting from ν_{split} follow the search path to x'.
   - when the paths goes right,
     report all leaves in the left subtree
   - check if μ’ ∈ $[x : x']$



1-Dimensional range searching

```
1DRangeQuery($T$, $[x : x']$)
    // Input: a binary search tree $T$ and a range $[x : x']$
    // Output: all points stored in $T$ that lie in the range
1  $v_{\text{split}} = $ FindSplitNode($T, x, x'$)
2  if $v_{\text{split}}$ is a leaf
3      check if the point stored at $v_{\text{split}}$ must be reported
4  else    // follow the path to $x$ and report the points in subtrees right of the path
5      $v = v_{\text{split}}.\text{left}$
6      while $v$ is not a leaf
7          if $x \leq x_v$:    ReportSubtree($v.\text{right}$);  $v = v.\text{left}$
8          else:               $v = v.\text{right}$
9      check if the point stored at the leaf $v$ must be reported
10     // similarly, follow the path to $x'$, report the points in subtrees left of the path,
       // and check if the point stored at the leaf where the path ends must be reported
```

1-Dimensional range searching

```
1DRangeQuery($T$, $[x : x']$)
    // Input: a binary search tree $T$ and a range $[x : x']$
    // Output: all points stored in $T$ that lie in the range
    $v_{\text{split}} = $ FindSplitNode($T$, $x$, $x'$)
    if $v_{\text{split}}$ is a leaf
        check if the point stored at $v_{\text{split}}$ must be reported
    else    // follow the path to $x$ and report the points in subtrees right of the path
        $v = v_{\text{split}}.\text{left}$
        while $v$ is not a leaf
            if $x \leq x_v$:    ReportSubtree($v.\text{right}$);  $v = v.\text{left}$
            else:               $v = v.\text{right}$
        check if the point stored at the leaf $v$ must be reported
        // similarly, follow the path to $x'$, report the points in subtrees left of the path,
        // and check if the point stored at the leaf where the path ends must be reported
```

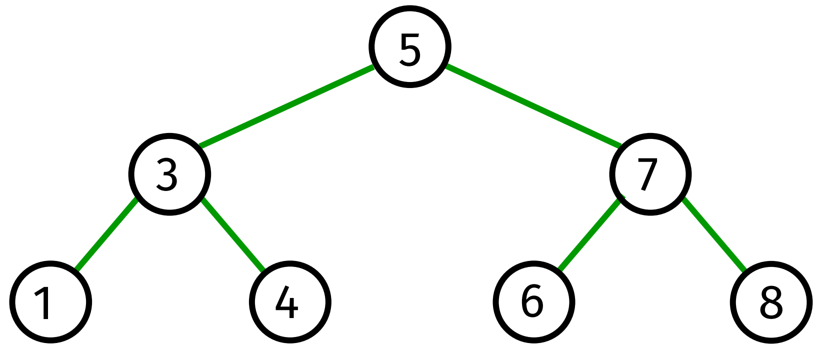
Correctness?

Need to show two things:

1. every reported point lies in the query range
2. every point in the query range is reported

Intermezzo: Trees and their Leaves

What can we say about the relationship between #nodes in a tree and #leaves in a tree?



Full binary tree: #leaves = #inner nodes + 1

➡ #nodes = 2 · #leaves - 1

Fact: #nodes = O(#leaves) holds for all balanced trees.

1-Dimensional range searching

```
1DRangeQuery($T$, $[x : x']$)
    // Input: a binary search tree $T$ and a range $[x : x']$
    // Output: all points stored in $T$ that lie in the range
1   $v_{\text{split}} = $ FindSplitNode($T$, $x$, $x'$)
2   if $v_{\text{split}}$ is a leaf
3       check if the point stored at $v_{\text{split}}$ must be reported
4   else    // follow the path to $x$ and report the points in subtrees right of the path
5       $v = v_{\text{split}}.\text{left}$
6       while $v$ is not a leaf
7           if $x \leq x_v$:   ReportSubtree($v.\text{right}$);  $v = v.\text{left}$
8           else:              $v = v.\text{right}$
9       check if the point stored at the leaf $v$ must be reported
10      // similarly, follow the path to $x'$, report the points in subtrees left of the path,
        // and check if the point stored at the leaf where the path ends must be reported
```

Query time?   ReportSubtree $= O(1 +$ reported points$)$

➡ total query time $= O(\log n$ + reported points$)$

Storage? $O(n)$

2-Dimensional range searching

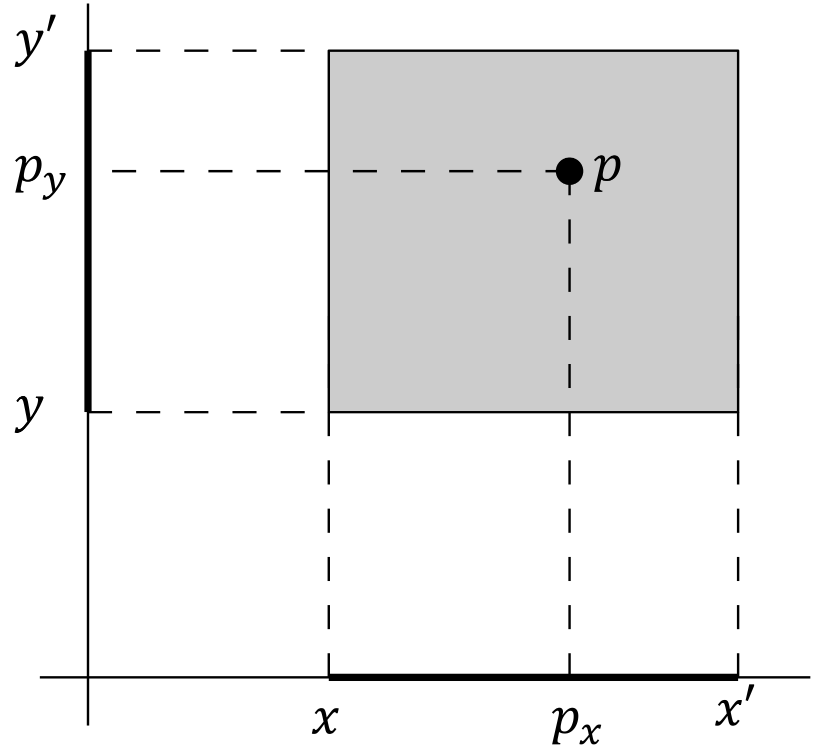
$P = \{p_1, p_2, \dots, p_n\}$ set of points in the plane

for now: no two points have the same x-coordinate, no two points have the same y-coordinate

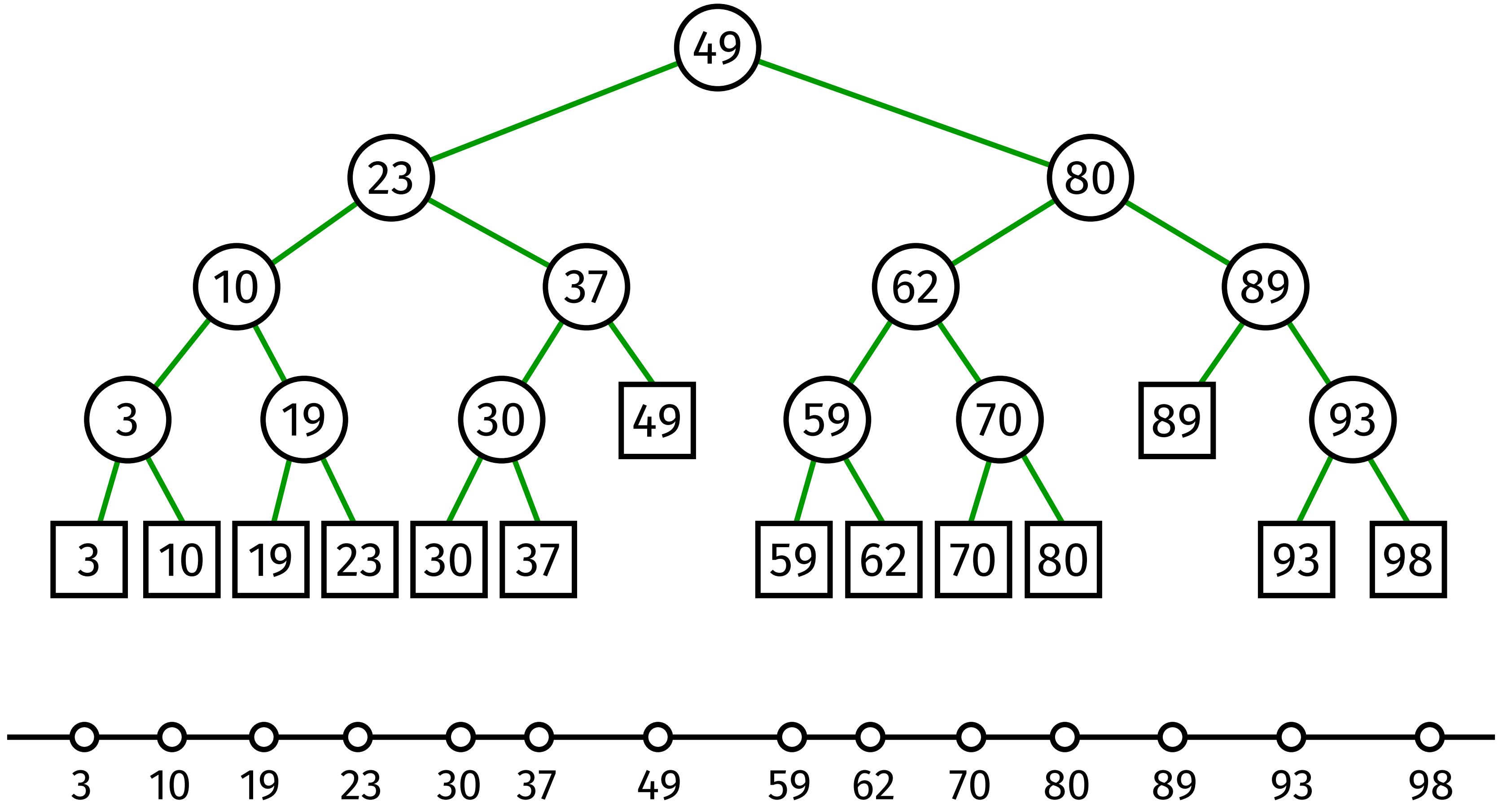
Query: given a query rectangle $[x : x'] \times [y : y']$ report all $p_i \in P$ with
$p_i \in [x : x'] \times [y : y']$, that is, $p_x \in [x : x']$ and $p_y \in [y : y']$

➡ a 2-dimensional range query is composed of two 1-dimensional sub-queries

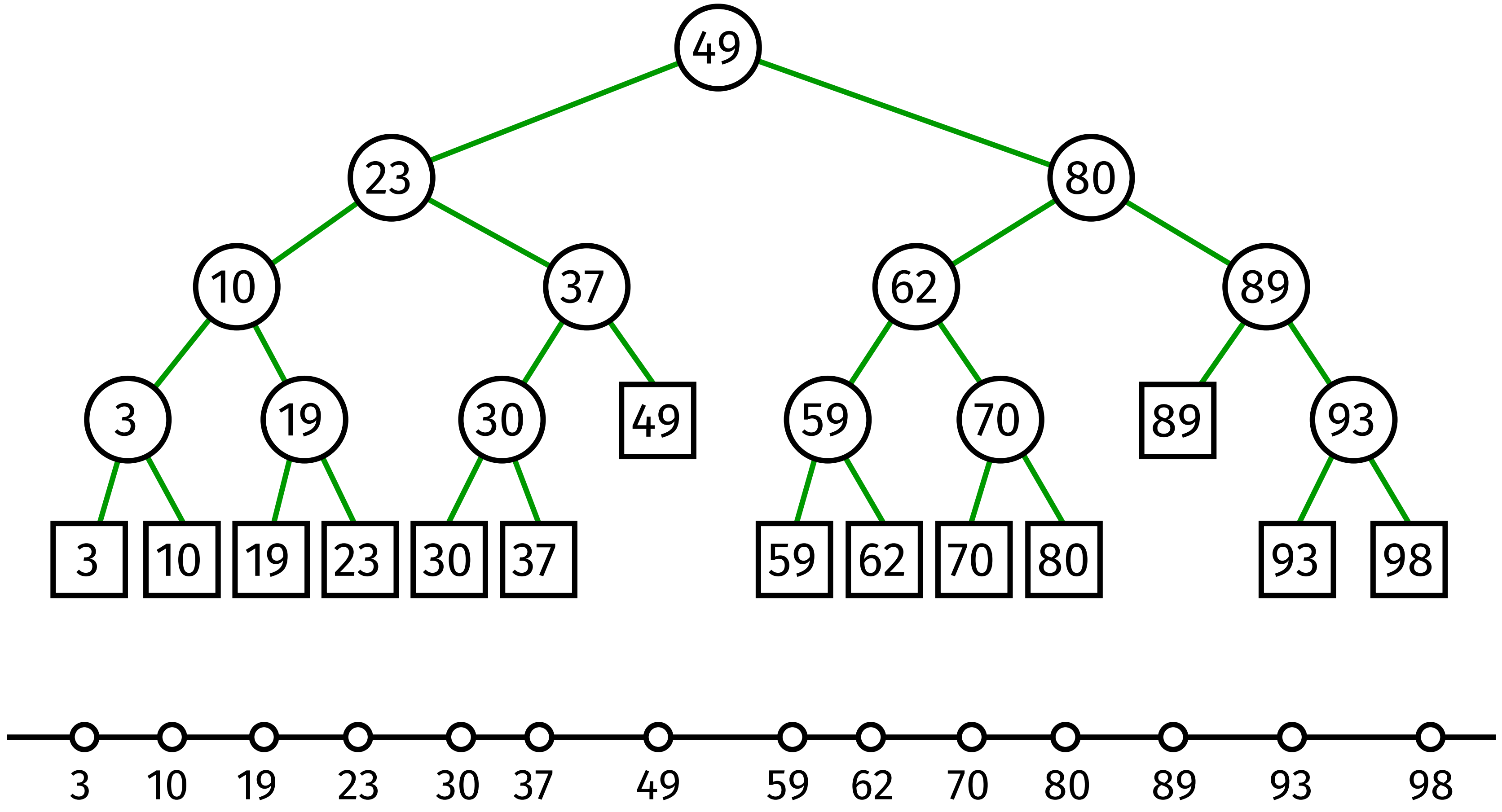
How can we generalize our 1-dimensional solution to 2 dimensions?



Back to one dimension ...

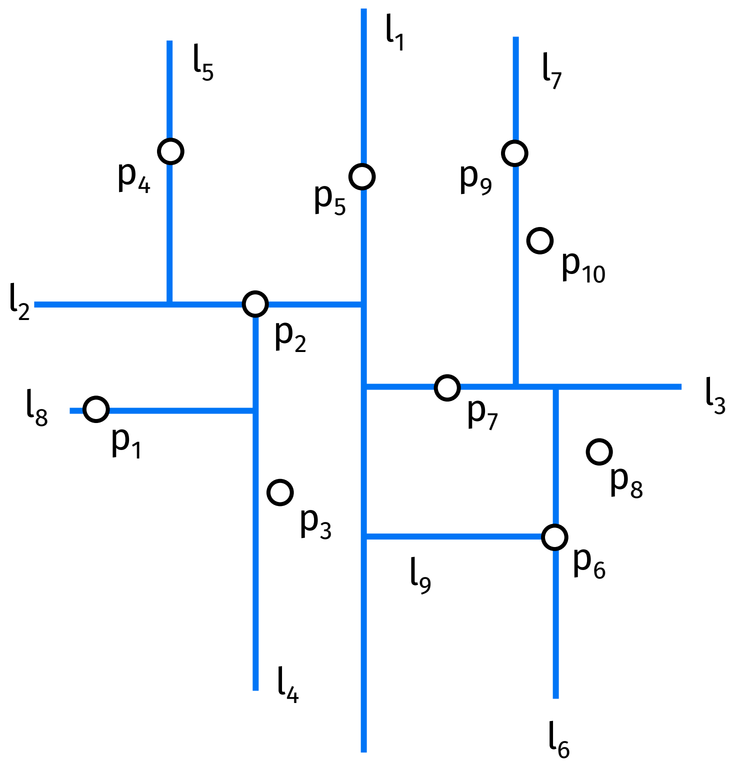


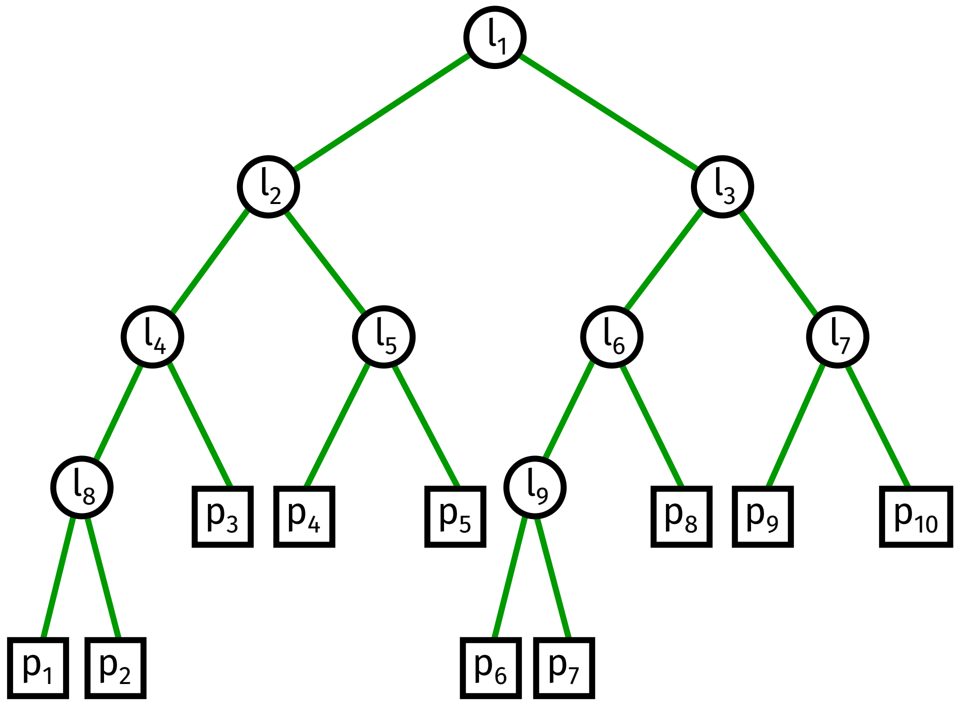
Back to one dimension ...



And now in two dimensions …

Split alternating on x- and y-coordinate





2-dimensional kd-tree

Kd-trees

```
BuildKDTree($P$, depth)
    // Input: a set of points $P$ and the current depth
    // Output: the root of a kd-tree storing $P$
 1  if $p$ contains only one point
 2          return a leaf storing this point
 3  else
 4          if depth is even
 5                  Split $P$ into two subsets with a vertical line $l$ through the median $x$-coordinate of the
                    points in $P$. Let $P_1$ be the set of points to the left or on $l$, and let $P_2$ be the set of points
                    to the right of $l$.
 6          else
 7                  Split $P$ into two subsets with a horizontal line $l$ through the median $y$-coordinate of the
                    points in $P$. Let $P_1$ be the set of points below or on $l$, and let $P_2$ be the set of points
                    above $l$.
 8  $v_{\text{left}}$ = BuildKdTree($P_1$, depth + 1)
 9  $v_{\text{right}}$ = BuildKdTree($P_2$, depth + 1)
10  return a node $v$ storing $l$ with left child $v_{\text{left}}$ and right child $v_{\text{right}}$
```

Kd-trees

```
BuildKDTree($P$, depth)
    // Input: a set of points $P$ and the current $depth$
    // Output: the root of a kd-tree storing $P$
1   if $p$ contains only one point
2       return a leaf storing this point
3   else
4       if depth is even
5           Split $P$ into two subsets with a vertical line $l$ through the median $x$-coordinate of the
            points in $P$. Let $P_1$ be the set of points to the left or on $l$, and let $P_2$ be the set of points
            to the right of $l$.
6       else
7           Split $P$ into two subsets with a horizontal line $l$ through the median $y$-coordinate of the
            points in $P$. Let $P_1$ be the set of points below or on $l$, and let $P_2$ be the set of points
            above $l$.
8   $v_{\text{left}}$ = BuildKdTree($P_1$, depth + 1)
9   $v_{\text{right}}$ = BuildKdTree($P_2$, depth + 1)
10  return a node $v$ storing $l$ with left child $v_{\text{left}}$ and right child $v_{\text{right}}$
```

Running time? $T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & \text{if } n > 1 \end{cases}$

➡ $T(n) = O(n \log n)$

presort to avoid linear time median finding …

Kd-trees

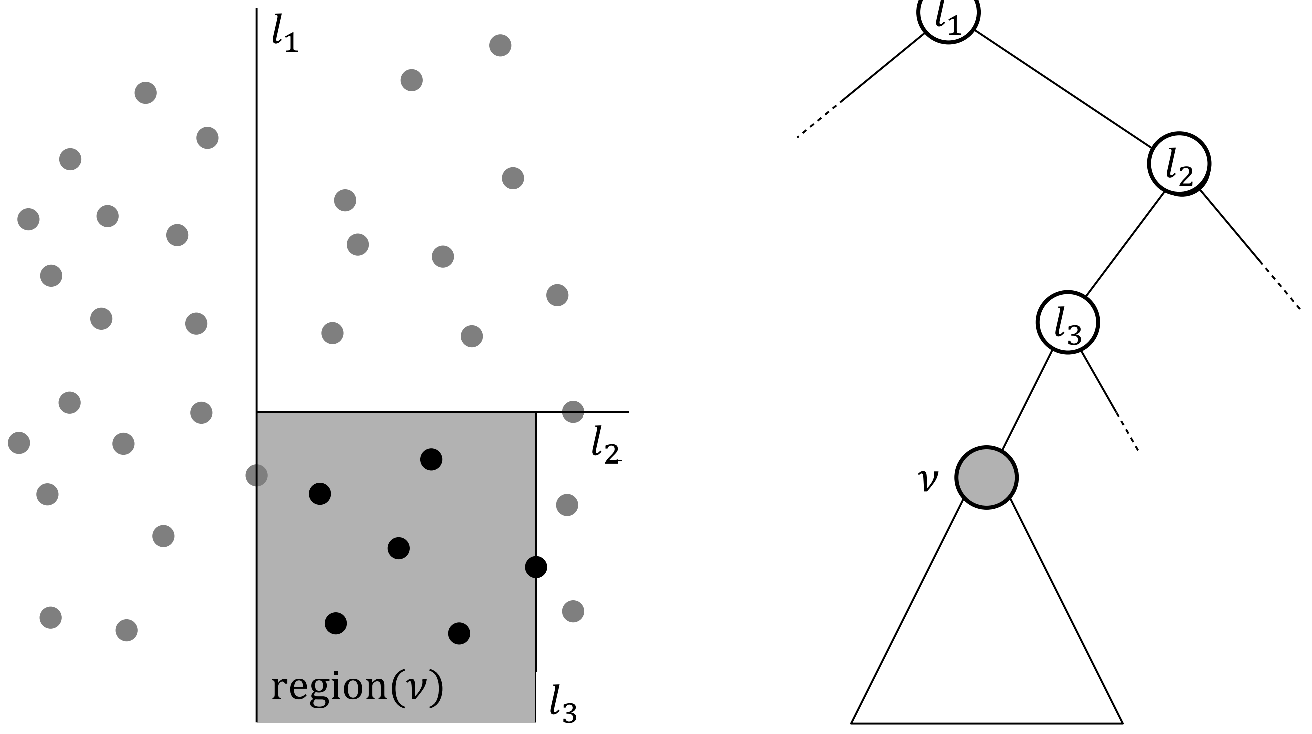
```
BuildKDTree($P$, depth)
    // Input: a set of points $P$ and the current $depth$
    // Output: the root of a kd-tree storing $P$
 1  if $p$ contains only one point
 2      return a leaf storing this point
 3  else
 4      if depth is even
 5          Split $P$ into two subsets with a vertical line $l$ through the median $x$-coordinate of the
            points in $P$. Let $P_1$ be the set of points to the left or on $l$, and let $P_2$ be the set of points
            to the right of $l$.
 6      else
 7          Split $P$ into two subsets with a horizontal line $l$ through the median $y$-coordinate of the
            points in $P$. Let $P_1$ be the set of points below or on $l$, and let $P_2$ be the set of points
            above $l$.
 8  $v_{\text{left}} = $ BuildKdTree($P_1$, depth + 1)
 9  $v_{\text{right}} = $ BuildKdTree($P_2$, depth + 1)
10  return a node $v$ storing $l$ with left child $v_{\text{left}}$ and right child $v_{\text{right}}$
```

Storage? $O(n)$

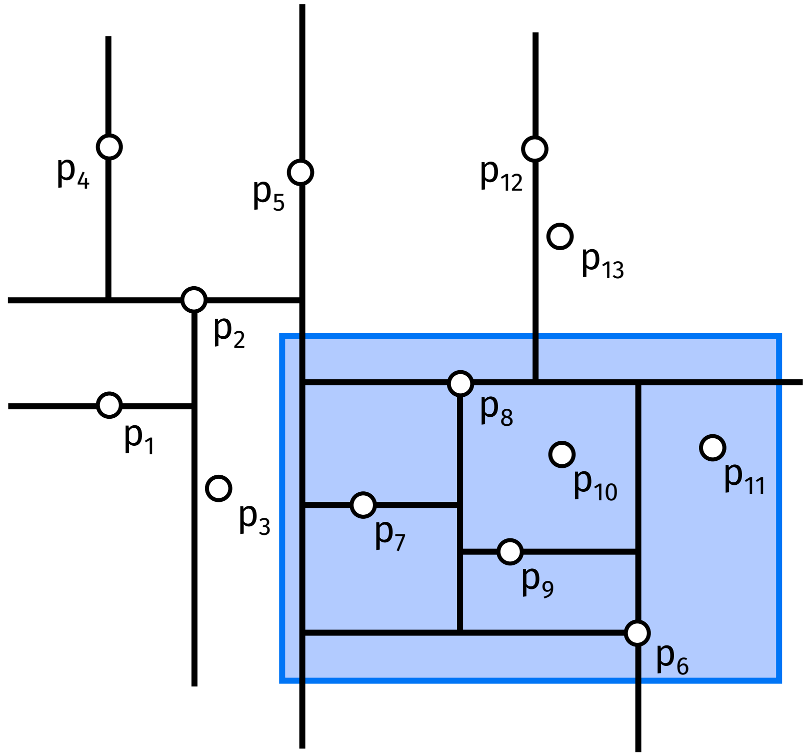
Querying a kd-tree

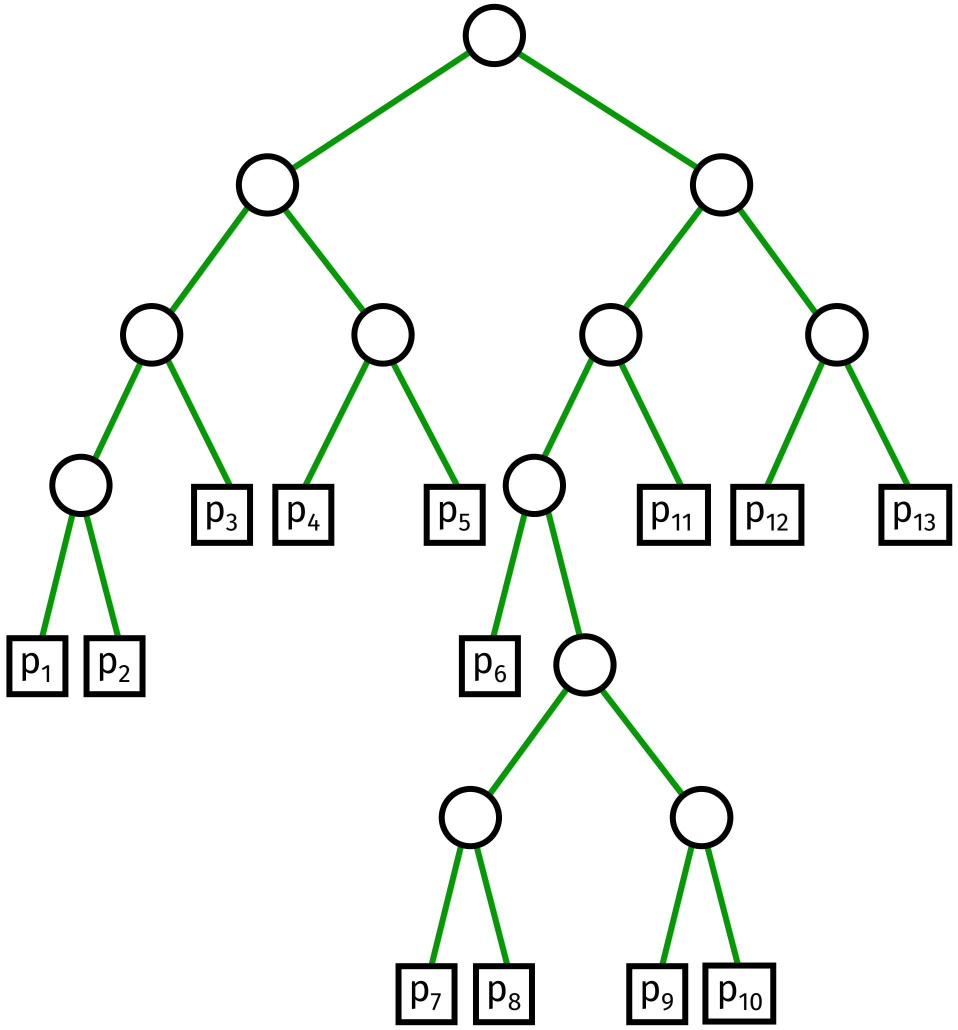
Each node v corresponds to a region $\text{region}(v)$.
All points which are stored in the subtree rooted at v lie in $\text{region}(v)$.

1. if region is contained in query rectangle, report all points in region.
2. if region is disjoint from query rectangle, report nothing.
3. if region intersects query rectangle, refine search (test children or point stored in region if no children)



Querying a kd-tree





Disclaimer: This tree cannot have been constructed by BuildKDTree …

Querying a kd-tree

```
SearchKDTree($v$, $R$)
    // Input: the root of (a subtree of) a kd-tree, and a range $R$
    // Output: all points at leaves below $v$ that lie in the range
 1  if $v$ is a leaf
 2      report the point stored at $v$ if it lies in $R$
 3  else
 4      if $\mathrm{region}(v.\mathrm{left})$ is fully contained in $R$
 5          ReportSubtree($v.\mathrm{left}$)
 6      else if $\mathrm{region}(v.\mathrm{left})$ intersects $R$
 7          SearchKDTree($v.\mathrm{left}$, $R$)
 8      if $\mathrm{region}(v.\mathrm{right})$ is fully contained in $R$
 9          ReportSubtree($v.\mathrm{right}$)
10      else if $\mathrm{region}(v.\mathrm{right})$ intersects $R$
11          SearchKDTree($v.\mathrm{right}$, $R$)
```

Query time?

Querying a kd-tree: Analysis

Time to traverse subtree and report points stored in leaves
is linear in number of leaves

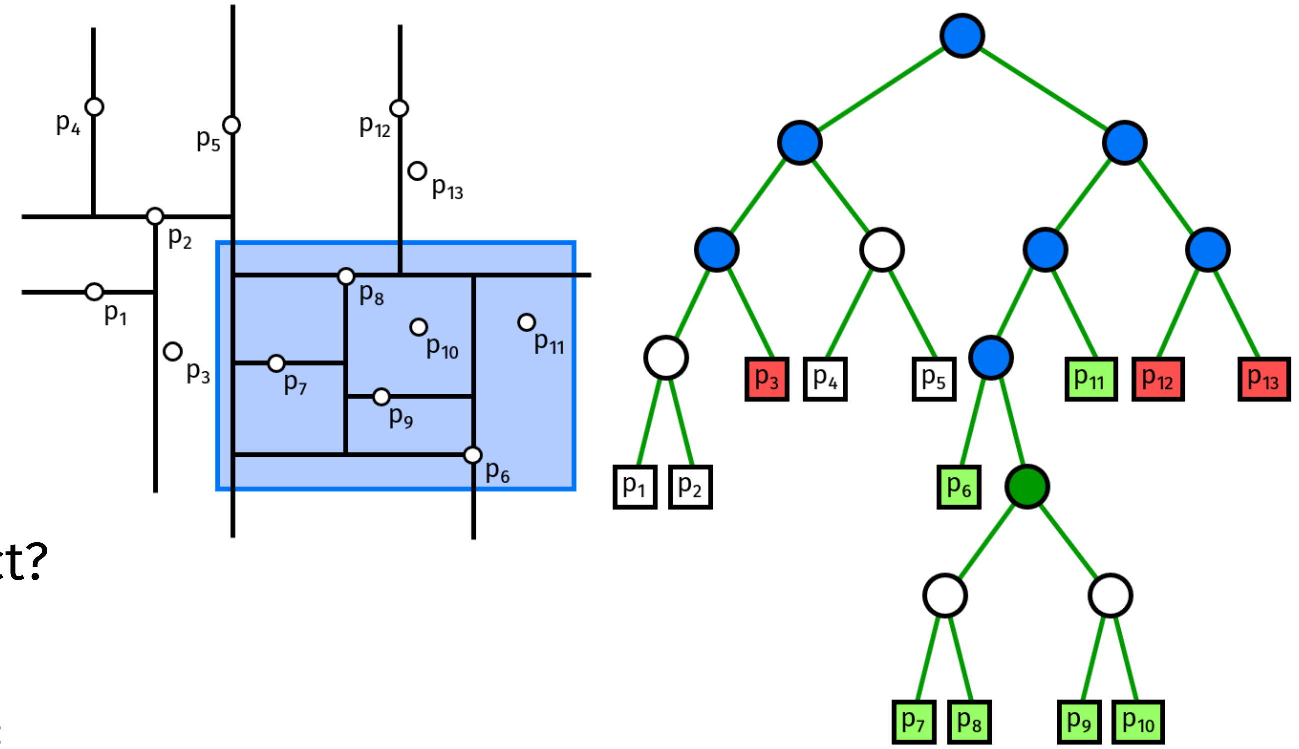
➡ ReportSubtree takes $O(1 + k)$ time, k total number of reported points

Need to bound number of nodes visited
that are not in the traversed subtrees
(blue nodes)

The query range properly intersects
the region of each such node

We are only interested in an upper bound ...

How many regions can a vertical line intersect?



Querying a kd-tree: Analysis

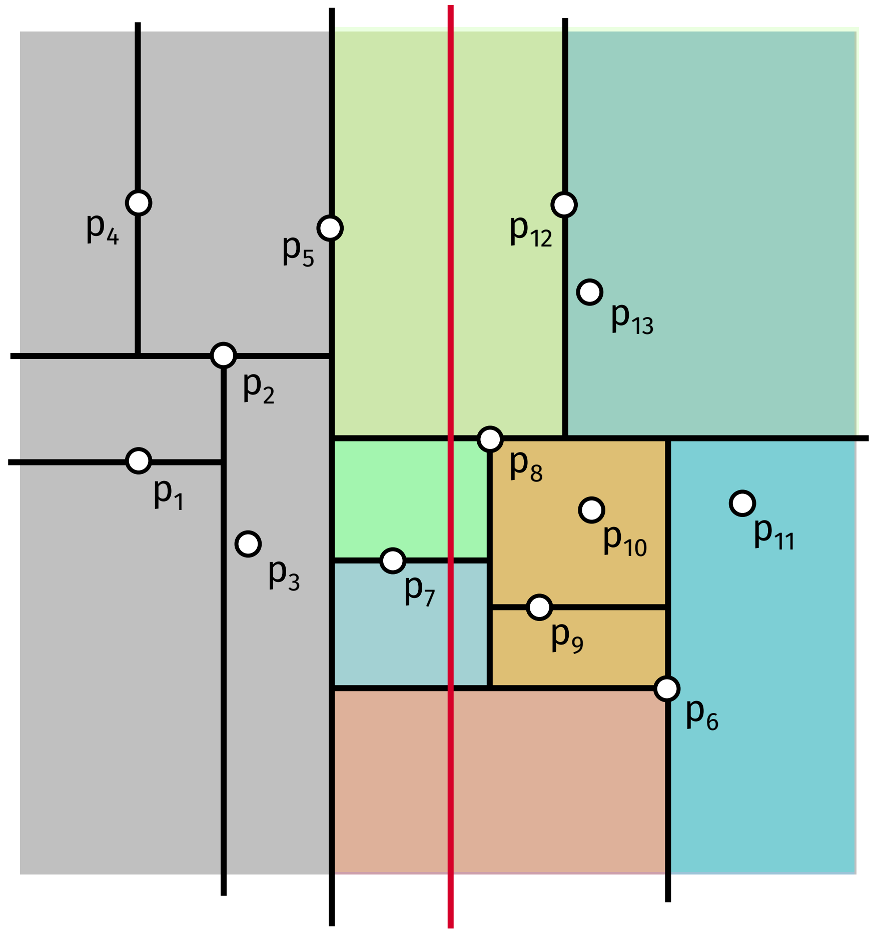
Question: How many regions can a vertical line intersect?

$Q(n)$: number of intersected regions

Answer 1: $Q(n) = 1 + Q(n/2)$

Answer 2: $Q(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2 + 2Q(n/4) & \text{if } n > 1 \end{cases}$

Master theorem ➡ $Q(n) = O(\sqrt{n})$



KD-trees

Theorem

A kd-tree for a set of n points in the plane uses $O(n)$ storage and can be built in $O(n \log n)$ time.
A rectangular range query on the kd-tree takes O($\sqrt{n} + k$) time, where k is the number of reported points.

If the number k of reported points is small, then the query time O($\sqrt{n} + k$) is relatively high.

Can we do better?

Trade storage for query time ...

Back to 1 dimension … (again)

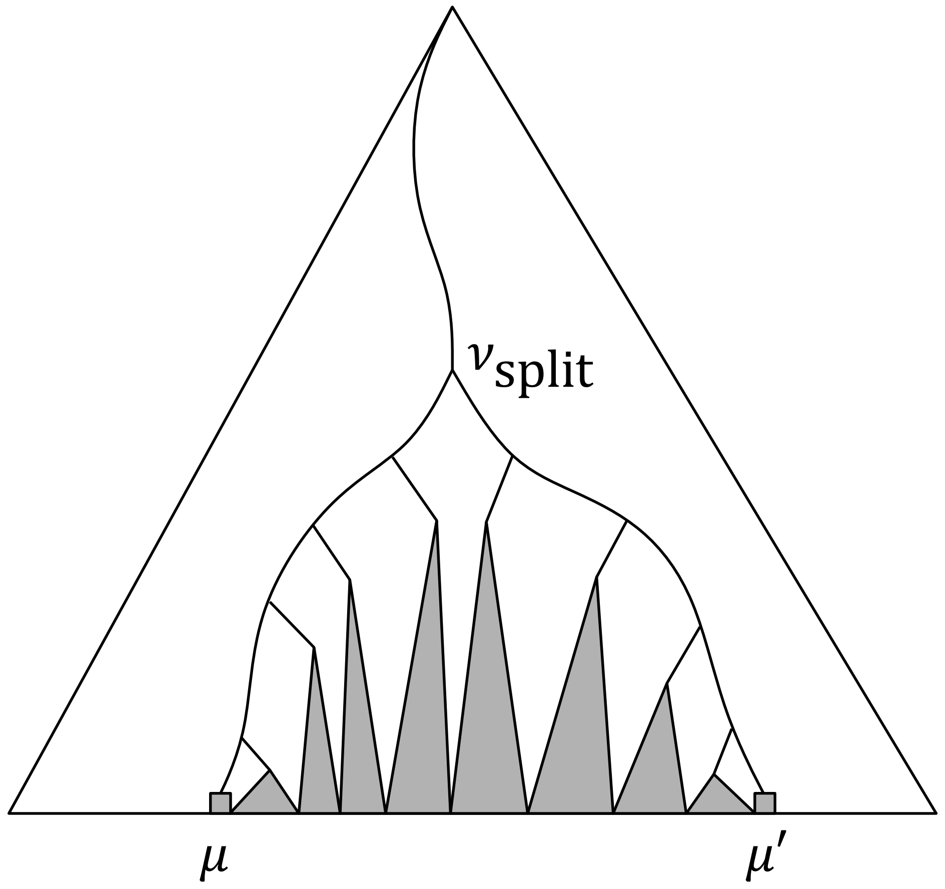
A 1DRangeQuery with $[x : x']$ gives us all points whose x-coordinates lie in the range $[x : x'] \times [y : y']$.

These points are stored in $O(\log n)$ subtrees.

Canonical subset of node v

points stored in the leaves of the subtree rooted at v

Idea: store canonical subsets in binary search tree on y-coordinate

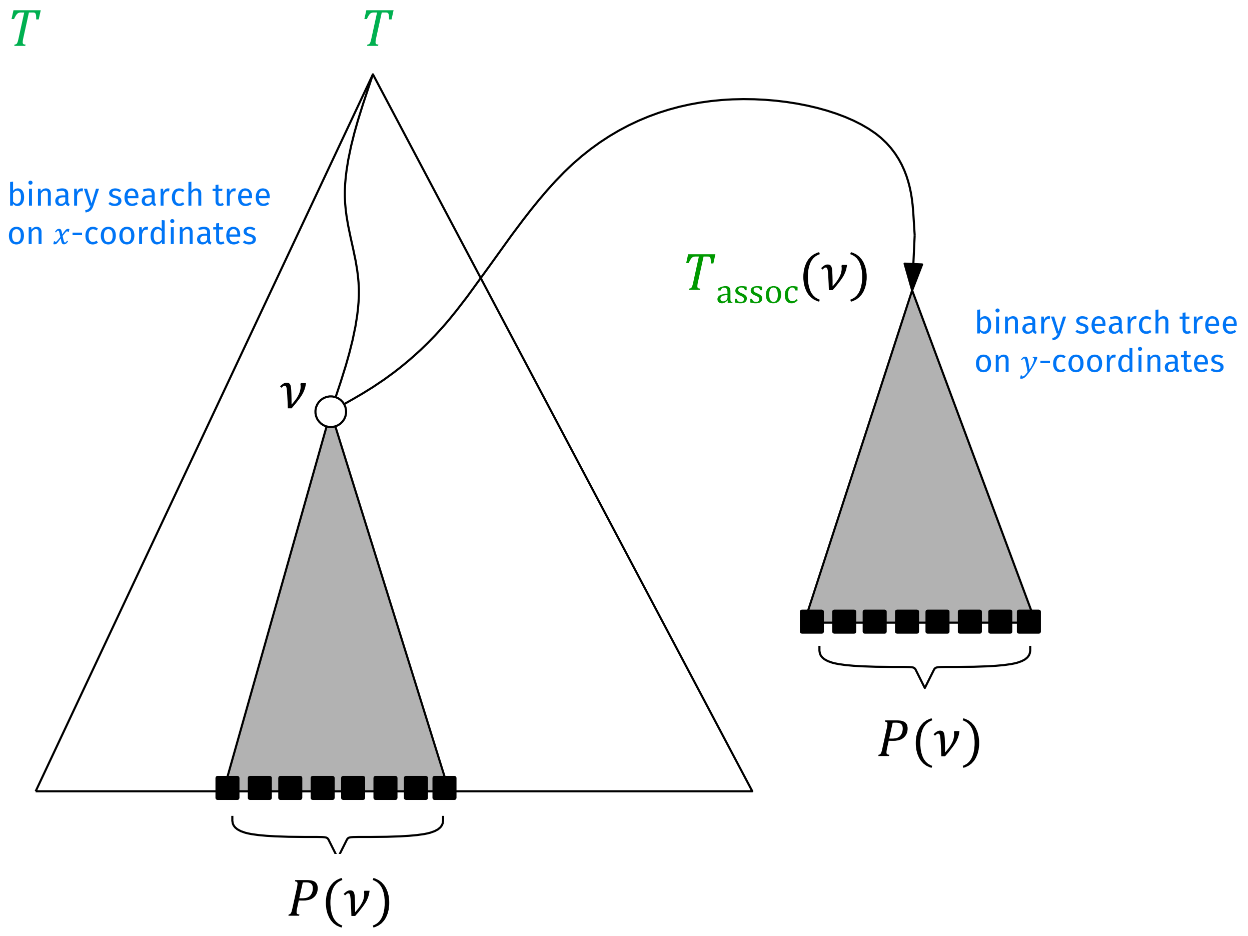


Range trees

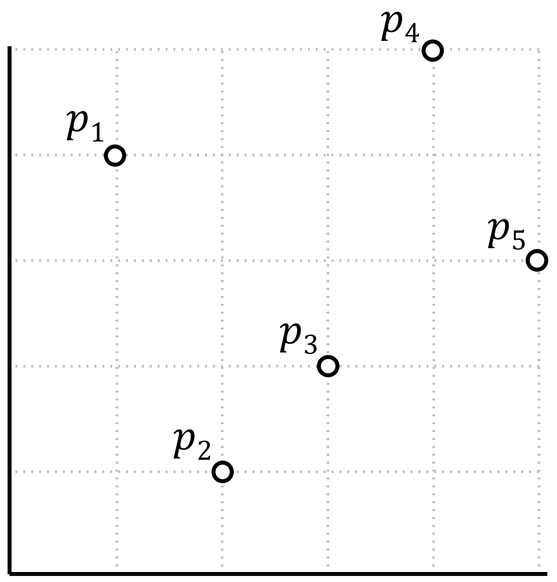
The main tree is a balanced binary search tree T built on the x-coordinates of the points in P.

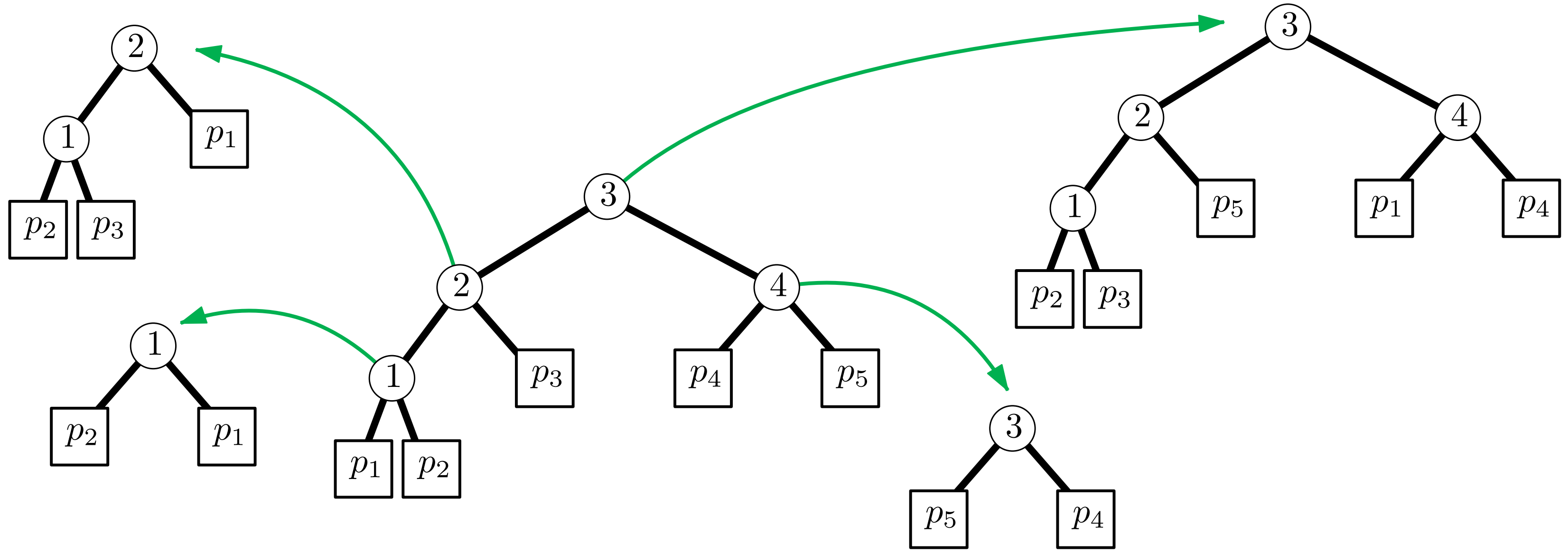
For any internal or leaf node v in T, the canonical subset $P(v)$ is stored in a balanced binary search tree $T_{\text{assoc}}(v)$ on the y-coordinate of the points.

The node v stores a pointer to the root of $T_{\text{assoc}}(v)$, which is called the associated structure of v.


T
binary search tree on x-coordinates
v
$P(v)$
$T_{\text{assoc}}(v)$
binary search tree on y-coordinates
$P(v)$

Range tree example





Each leaf of the main tree also has an associated structure, we omit these for clarity of presentation.

Range trees

```
Build2DRangeTree($P$)
    // Input: a set of points $P$ in the plane
    // Output: the root of a 2-dimensional range tree
1   Construct the associated structure: Build a binary search tree $T_{\text{assoc}}$ on the set $P_y$ of $y$-coordinates
    of the points in $P$. Store at the leaves of $T_{\text{assoc}}$ not just the $y$-coordinates of the points in $P_y$, but
    the points themselves.
2   if $p$ contains only one point
3       Create a leaf $v$ storing this point, and make $T_{\text{assoc}}$ the associated structure of $v$.
4   else
5       Split $P$ into two subsets; one subset $P_{\text{left}}$ contains the points with $x$-coordinate less than or
        equal to $x_{\text{mid}}$, the median $x$-coordinate, and the other subset $P_{\text{right}}$ contains the points with $x$-
        coordinate larger than $x_{\text{mid}}$.
6       $v_{\text{left}} =$ Build2DRangeTree($P_{\text{left}}$)
7       $v_{\text{right}} =$ Build2DRangeTree($P_{\text{right}}$)
8       Create a node $v$ storing $x_{\text{mid}}$, make $v_{\text{left}}$ the left child of $v$, make $v_{\text{right}}$ the right child of $v$, and
        make $T_{\text{assoc}}$ the associated structure of $v$.
9   return $v$
```

Range trees

```
Build2DRangeTree($P$)
    // Input: a set of points $P$ in the plane
    // Output: the root of a 2-dimensional range tree
1   Construct the associated structure: Build a binary search tree $T_{\text{assoc}}$ on the set $P_y$ of $y$-coordinates
    of the points in $P$. Store at the leaves of $T_{\text{assoc}}$ not just the $y$-coordinates of the points in $P_y$, but
    the points themselves.
2   if $p$ contains only one point
3       Create a leaf $v$ storing this point, and make $T_{\text{assoc}}$ the associated structure of $v$.
4   else
5       Split $P$ into two subsets; one subset $P_{\text{left}}$ contains the points with $x$-coordinate less than or
        equal to $x_{\text{mid}}$, the median $x$-coordinate, and the other subset $P_{\text{right}}$ contains the points with $x$-
        coordinate larger than $x_{\text{mid}}$.
6       $v_{\text{left}} = $ Build2DRangeTree($P_{\text{left}}$)
7       $v_{\text{right}} = $ Build2DRangeTree($P_{\text{right}}$)
8       Create a node $v$ storing $x_{\text{mid}}$, make $v_{\text{left}}$ the left child of $v$, make $v_{\text{right}}$ the right child of $v$, and
        make $T_{\text{assoc}}$ the associated structure of $v$.
9   return $v$
```

Running time? $O(n \log n)$

presort to build binary search trees in linear time ...

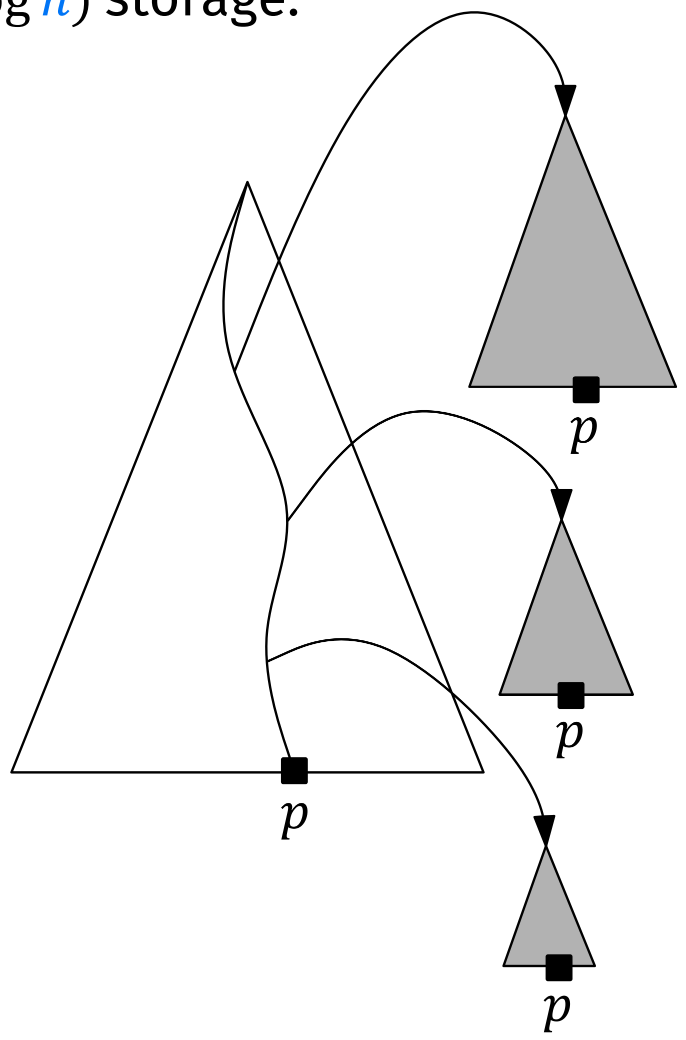
Range trees: Storage

Lemma

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

Proof:

- each point is stored only once per level
- storage for associated structures is linear in number of points
- there are $O(\log n)$ levels



Querying a range tree

```
2DRangeQuery($T$, $[x : x'] \times [y : y']$)
    // Input: a 2-dimensional range tree $T$ and a range $[x : x'] \times [y : y']$
    // Output: all points in $T$ that lie in the range
1   $v_{\text{split}} = $ FindSplitNode($T, x, x'$)
2   if $v_{\text{split}}$ is a leaf
3           check if the point stored at $v_{\text{split}}$ must be reported
4   else    // follow the path to $x$ and call 1DRangeQuery on the subtrees right of the path
5           $v = v_{\text{split}}.\text{left}$
6           while $v$ is not a leaf
7                   if $x \leq x_v$:      1DRangeQuery($T_{\text{assoc}}(v.\text{right})$, $[y: y']$);  $v = v.\text{left}$
8                   else:                 $v = v.\text{right}$
9           check if the point stored at the leaf $v$ must be reported
10          // similarly, follow the path from $v_{\text{split}}.\text{right}$ to $x'$, call 1DRangeQuery with the range $[y: y']$
            // on the associated structures of subtrees left of the path, and check if the point stored
            // at the leaf where the path ends must be reported
```

Querying a range tree

```
2DRangeQuery($T$, $[x : x'] \times [y : y']$)
    // Input: a 2-dimensional range tree $T$ and a range $[x : x'] \times [y : y']$
    // Output: all points in $T$ that lie in the range
1   $v_{\text{split}} = $ FindSplitNode($T, x, x'$)
2   if $v_{\text{split}}$ is a leaf
3       check if the point stored at $v_{\text{split}}$ must be reported
4   else   // follow the path to $x$ and call 1DRangeQuery on the subtrees right of the path
5       $v = v_{\text{split}}.\text{left}$
6       while $v$ is not a leaf
7           if $x \leq x_v$:    1DRangeQuery($T_{\text{assoc}}(v.\text{right})$, $[y : y']$);  $v = v.\text{left}$
8           else:               $v = v.\text{right}$
9       check if the point stored at the leaf $v$ must be reported
10      // similarly, follow the path from $v_{\text{split}}.\text{right}$ to $x'$, call 1DRangeQuery with the range $[y : y']$
        // on the associated structures of subtrees left of the path, and check if the point stored
        // at the leaf where the path ends must be reported
```

Running time? $O(\log^2 n + k)$

Range trees

Theorem

A range tree for a set of n points in the plane uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.
A rectangular range query on the range tree takes $O(\log^2 n + k)$ time, where k is the number of reported points.

Conclusion

	kd-tree	Range tree
storage	$O(n)$	$O(n \log n)$
query time	$O(\sqrt{n} + k)$	$O(\log^2 n + k)$