Introduction to Geometric Data Structures

Partha P. Goswami
(ppg.rpe@caluniv.ac.in)

Institute of Radio Physics and Electronics
University of Calcutta
92, APC Road, Kolkata - 700009, West Bengal, India.

Outline

INTRODUCTION

- In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Usually efficient data structures are a key to designing efficient algorithms.

Introduction

- In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Usually efficient data structures are a key to designing efficient algorithms.
- Different kinds of data structure are suited to different kinds of application, and some are highly specialized to specific tasks.

INTRODUCTION

- In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Usually efficient data structures are a key to designing efficient algorithms.
- Different kinds of data structure are suited to different kinds of application, and some are highly specialized to specific tasks.
- Computational geometry often require preprocessing geometric objects into a simple and space-efficient structure so that the operations on the geometric objects can be performed repeatedly in an efficient manner.

INTRODUCTION

- Classic data structures like lists, trees, and graphs are by themselves not sufficient to represent geometric objects as either they are generally one dimensional in nature or do not capture the rich structural properties of the geometric objects in the domain.

INTRODUCTION

- Classic data structures like lists, trees, and graphs are by themselves not sufficient to represent geometric objects as either they are generally one dimensional in nature or do not capture the rich structural properties of the geometric objects in the domain.
- Therefore, researchers designed a number of different data structures to solve various geometric problems.

INTRODUCTION

- Classic data structures like lists, trees, and graphs are by themselves not sufficient to represent geometric objects as either they are generally one dimensional in nature or do not capture the rich structural properties of the geometric objects in the domain.
- Therefore, researchers designed a number of different data structures to solve various geometric problems.
- In this lecture we introduce a few simple and basic geometric data structures.

INTRODUCTION

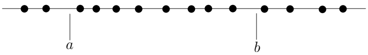
- Classic data structures like lists, trees, and graphs are by themselves not sufficient to represent geometric objects as either they are generally one dimensional in nature or do not capture the rich structural properties of the geometric objects in the domain.
- Therefore, researchers designed a number of different data structures to solve various geometric problems.
- In this lecture we introduce a few simple and basic geometric data structures.
- In our discussion, we consider several problems which we want to solve in repetitive query mode. This means, data set is given a priori and we are allowed to preprocess the data-set. Queries come repetitively and we want to answer them efficiently. Various data structures will be introduced whose use lead efficient solution of the problems considered.

Outline

1-DIMENSIONAL RANGE SEARCHING

Problem

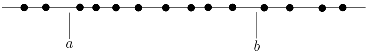
Given a set P of n points $p_1, p_2, \ldots, p_n$ *on the real line, report points of P that lie in the range* $[a, b], a \leq b$.



1-DIMENSIONAL RANGE SEARCHING

Problem

Given a set P of n points $p_1, p_2, \ldots, p_n$ on the real line, report points of P that lie in the range $[a, b], a \leq b$.

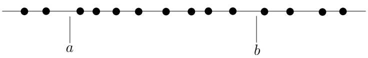


- Using array for storing P, we can use binary search on the array to answer such a query in $O(\log n + k)$ time where k is the number of points of P reported.

1-DIMENSIONAL RANGE SEARCHING

Problem

Given a set P of n points $p_1, p_2, \ldots, p_n$ on the real line, report points of P that lie in the range $[a, b], a \leq b$.



- Using array for storing P, we can use binary search on the array to answer such a query in $O(\log n + k)$ time where k is the number of points of P reported.
- Problems with this solution are that it can't be generalized to higher dimensions and it does not allow for efficient updates.

1-DIMENSIONAL RANGE SEARCHING

- A better solution would be to use a balanced binary search tree $\mathcal{T}$ in the following way:

1-DIMENSIONAL RANGE SEARCHING

- A better solution would be to use a balanced binary search tree $\mathcal{T}$ in the following way:
  - The leaves store the points of P and the internal nodes store splitting values.

1-DIMENSIONAL RANGE SEARCHING

- A better solution would be to use a balanced binary search tree $\mathcal{T}$ in the following way:
  - The leaves store the points of P and the internal nodes store splitting values.
  - Let v be an internal node of $\mathcal{T}$ and x_v be the splitting value stored at V.

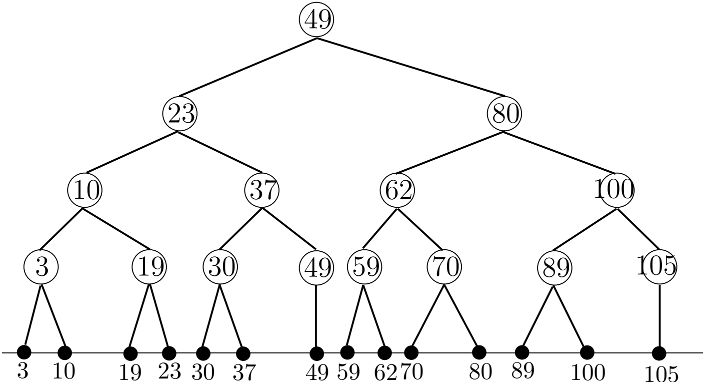
1-DIMENSIONAL RANGE SEARCHING

- A better solution would be to use a balanced binary search tree $\mathcal{T}$ in the following way:
  - The leaves store the points of P and the internal nodes store splitting values.
  - Let v be an internal node of $\mathcal{T}$ and x_v be the splitting value stored at V.
  - We assume that the left subtree of a node v contains all the points smaller than or equal to x_v.

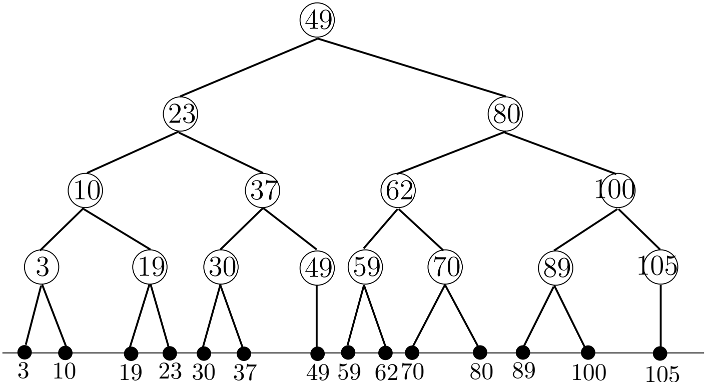
1-DIMENSIONAL RANGE SEARCHING

- A better solution would be to use a balanced binary search tree $\mathcal{T}$ in the following way:
  - The leaves store the points of P and the internal nodes store splitting values.
  - Let v be an internal node of $\mathcal{T}$ and x_v be the splitting value stored at V.
  - We assume that the left subtree of a node v contains all the points smaller than or equal to x_v.
- Here is an example.

1-DIMENSIONAL RANGE SEARCHING



1-DIMENSIONAL RANGE SEARCHING



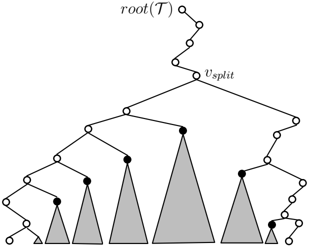
- Such a balanced binary search tree $\mathcal{T}$ on n points can be constructed in $n \log n$ time and it uses n storage.

1-DIMENSIONAL RANGE SEARCHING

- Let the query range be $[a, b]$. To report all points in the query range, we proceed as follows.

1-DIMENSIONAL RANGE SEARCHING

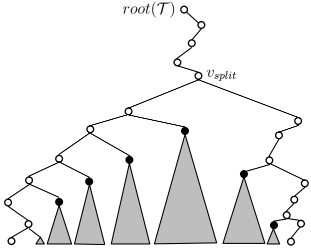
- Let the query range be $[a, b]$. To report all points in the query range, we proceed as follows.
- We start search with a and b at the root of $\mathcal{T}$ and find out the node, say v_{split}, where the paths to a and b split. This particular node is called the split node.



1-DIMENSIONAL RANGE SEARCHING

- Let the query range be $[a, b]$. To report all points in the query range, we proceed as follows.
- We start search with a and b at the root of $\mathcal{T}$ and find out the node, say v_{split}, where the paths to a and b split. This particular node is called the split node.

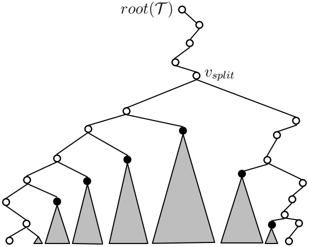
- Starting from v_{split} we first follow the search path of a. At each node where the path goes left, we report all the leaves in the right subtree because this is in between the two search paths.



1-DIMENSIONAL RANGE SEARCHING

- Let the query range be $[a, b]$. To report all points in the query range, we proceed as follows.
- We start search with a and b at the root of $\mathcal{T}$ and find out the node, say v_{split}, where the paths to a and b split. This particular node is called the split node.

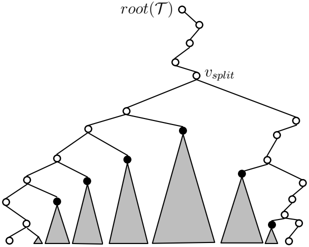
- Starting from v_{split} we first follow the search path of a. At each node where the path goes left, we report all the leaves in the right subtree because this is in between the two search paths.
- Similarly, we follow the path of b and we report the leaves in the left subtree of nodes where the path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



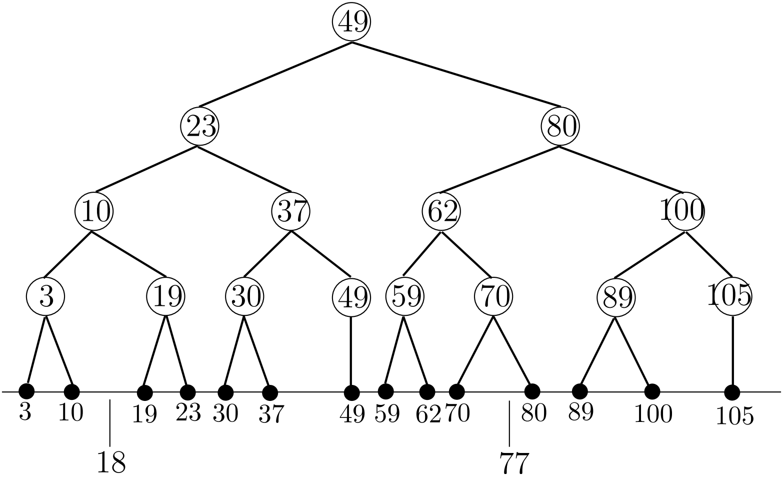
1-DIMENSIONAL RANGE SEARCHING

- Let the query range be $[a, b]$. To report all points in the query range, we proceed as follows.
- We start search with a and b at the root of $\mathcal{T}$ and find out the node, say v_{split}, where the paths to a and b split. This particular node is called the split node.

- Starting from v_{split} we first follow the search path of a. At each node where the path goes left, we report all the leaves in the right subtree because this is in between the two search paths.
- Similarly, we follow the path of b and we report the leaves in the left subtree of nodes where the path goes right.
- Finally we check the points stored at the leaves where paths end; we may or may not need to report them.

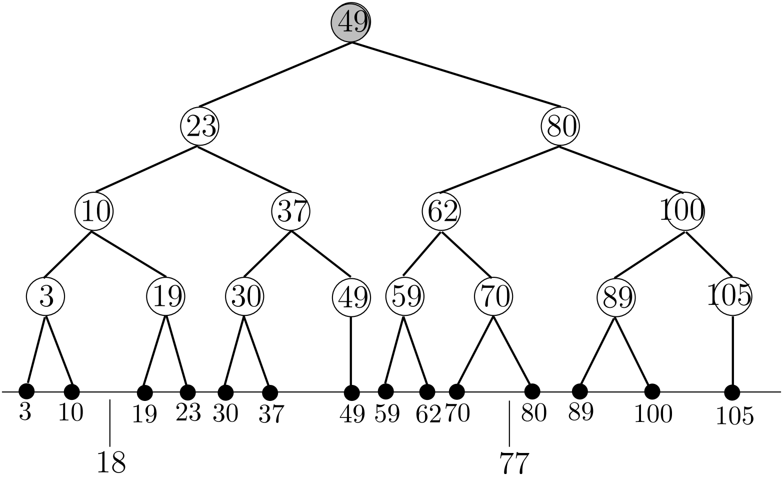


1-DIMENSIONAL RANGE SEARCHING



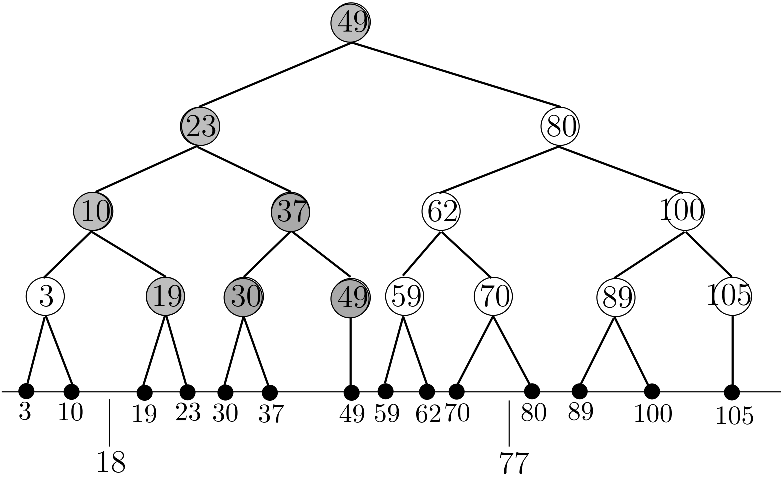
Searching with the query range $[18, 77]$.

1-DIMENSIONAL RANGE SEARCHING



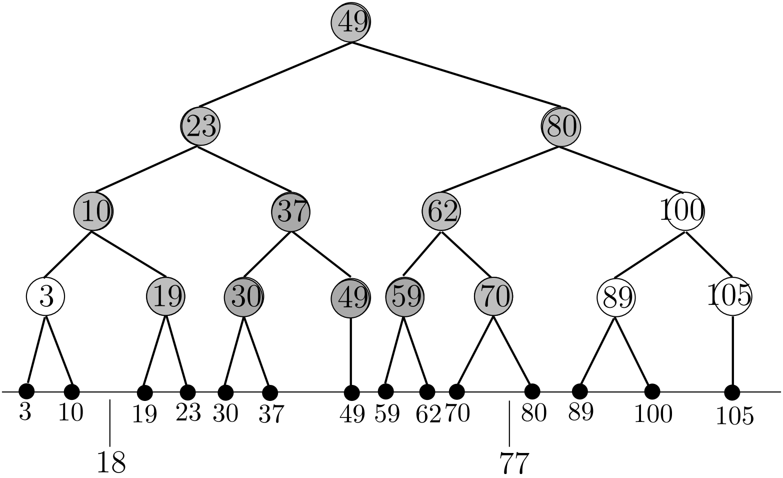
Searching with the query range [18, 77].

1-DIMENSIONAL RANGE SEARCHING



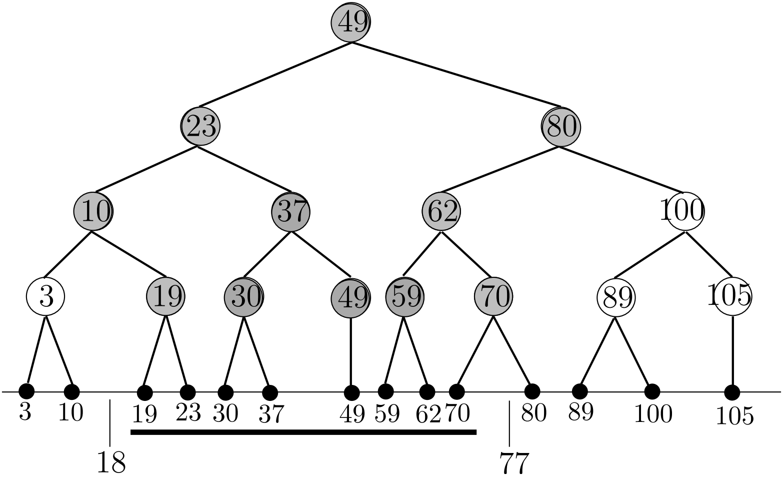
Searching with the query range $[18, 77]$.

1-DIMENSIONAL RANGE SEARCHING



Searching with the query range $[18, 77]$.

1-DIMENSIONAL RANGE SEARCHING



Searching with the query range $[18, 77]$.

1-DIMENSIONAL RANGE SEARCHING

- To analyze the query time, we note that in the worst case all the points could be in the query range, implying a query time to be $\Theta(n)$.

1-DIMENSIONAL RANGE SEARCHING

- To analyze the query time, we note that in the worst case all the points could be in the query range, implying a query time to be $\Theta(n)$.
- But it seems bad! Because, to achieve this query time, we do not need any data structure! Simply check each point against the query range.

1-DIMENSIONAL RANGE SEARCHING

- To analyze the query time, we note that in the worst case all the points could be in the query range, implying a query time to be $\Theta(n)$.
- But it seems bad! Because, to achieve this query time, we do not need any data structure! Simply check each point against the query range.
- But this query time can not be avoided if we really have to report all the points.

1-DIMENSIONAL RANGE SEARCHING

- To analyze the query time, we note that in the worst case all the points could be in the query range, implying a query time to be $\Theta(n)$.
- But it seems bad! Because, to achieve this query time, we do not need any data structure! Simply check each point against the query range.
- But this query time can not be avoided if we really have to report all the points.
- Actually, it can be seen that our algorithm is output-sensitive, meaning that its running time is sensitive to the size of the output.

1-DIMENSIONAL RANGE SEARCHING

- We note that in the reporting phase the algorithm takes, for each subtree, linear time in the number of points reported. So total time spent in the reporting phase is $O(k)$, if k points are reported.

1-DIMENSIONAL RANGE SEARCHING

- We note that in the reporting phase the algorithm takes, for each subtree, linear time in the number of points reported. So total time spent in the reporting phase is $O(k)$, if k points are reported.
- Since the tree we have used is balanced binary, length of maximum search is $O(\log n)$, Total query time is $O(\log n + k)$.

1-DIMENSIONAL RANGE SEARCHING

- We note that in the reporting phase the algorithm takes, for each subtree, linear time in the number of points reported. So total time spent in the reporting phase is $O(k)$, if k points are reported.
- Since the tree we have used is balanced binary, length of maximum search is $O(\log n)$, Total query time is $O(\log n + k)$.
- To summarize, we have the result:

RESULT

Theorem

Let P be a set of n points in 1-dimensional space. The set P can be stored in a balanced binary search tree, which uses $O(n)$ storage and has $O(n \log n)$ construction time, such that the points in a query range can be reported in time $O(k + \log n)$, where k is the number of points reported.

2-DIMENSIONAL RANGE SEARCHING

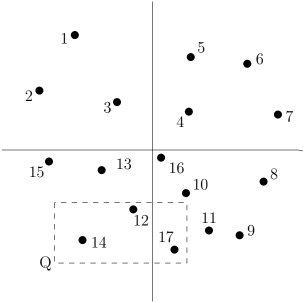
- We now move on to 2-dimensional rectangular range searching problem. The problem is defined as follows.

2-DIMENSIONAL RANGE SEARCHING

- We now move on to 2-dimensional rectangular range searching problem. The problem is defined as follows.

Problem

Given a set P of n points in the plane, report points inside a query rectangle $Q := [x : x'] \times [y : y']$ *whose sides are parallel to the axes.*

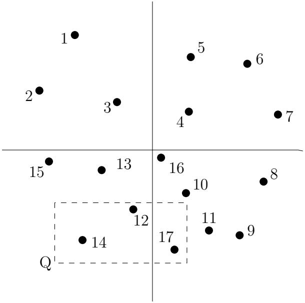


2-DIMENSIONAL RANGE SEARCHING

- We now move on to 2-dimensional rectangular range searching problem. The problem is defined as follows.

Problem

Given a set P of n points in the plane, report points inside a query rectangle $Q := [x : x'] \times [y : y']$ whose sides are parallel to the axes.



- For the example shown in the Figure, the answer is 14, 12, and 17.

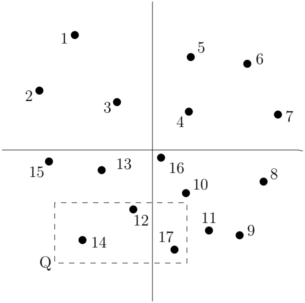
2-DIMENSIONAL RANGE SEARCHING

- We now move on to 2-dimensional rectangular range searching problem. The problem is defined as follows.

Problem

Given a set P of n points in the plane, report points inside a query rectangle $Q := [x : x'] \times [y : y']$ whose sides are parallel to the axes.

- For the example shown in the Figure, the answer is 14, 12, and 17.
- We assume that no two points in P have the same x-coordinate and no two points have the same y-coordinate.



2-DIMENSIONAL RANGE SEARCHING

- We observe that

2-DIMENSIONAL RANGE SEARCHING

- We observe that
  - A point $p := (p_x, p_y)$ lies inside a query rectangle $Q := [x : x'] \times [y : y']$ if and only if

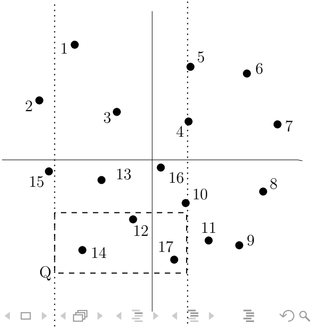
$$p_x \in [x : x'] \text{ and } p_y \in [y : y']$$

2-DIMENSIONAL RANGE SEARCHING

- We observe that
  - A point $p := (p_x, p_y)$ lies inside a query rectangle $Q := [x : x'] \times [y : y']$ if and only if

$$p_x \in [x : x'] \text{ and } p_y \in [y : y']$$

  - A 2-dimensional rectangular range query is composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one on the y-coordinate.



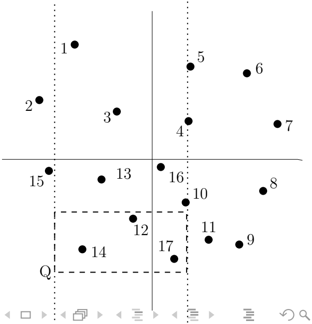
2-DIMENSIONAL RANGE SEARCHING

- We observe that
  - A point $p := (p_x, p_y)$ lies inside a query rectangle $Q := [x : x'] \times [y : y']$ if and only if

$$p_x \in [x : x'] \text{ and } p_y \in [y : y']$$

  - A 2-dimensional rectangular range query is composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one on the y-coordinate.

- Direct application of the method suggested by the second observation may lead to a cost which exceed the actual output size of the 2-d range query.



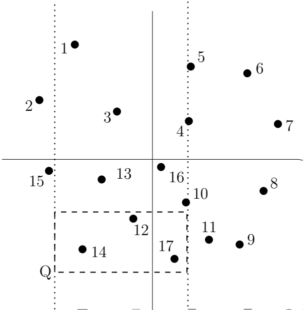
2-DIMENSIONAL RANGE SEARCHING

- We observe that
  - A point $p := (p_x, p_y)$ lies inside a query rectangle $Q := [x : x'] \times [y : y']$ if and only if

$$p_x \in [x : x'] \text{ and } p_y \in [y : y']$$

  - A 2-dimensional rectangular range query is composed of two 1-dimensional sub-queries, one on the x-coordinate of the points and one on the y-coordinate.

- Direct application of the method suggested by the second observation may lead to a cost which exceed the actual output size of the 2-d range query.
- We, however, can try splitting both x- and y-coordinates alternatively. This leads to a data structure called Kd-tree.



KD-TREE

- Kd-tree is defined as follows

KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.

KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.
  - P_{left}, the subset of points to the left or on l, is stored in the left subtree. P_{right}, the subset to the right of l, is stored in the right subtree.

KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.
  - P_{left}, the subset of points to the left or on l, is stored in the left subtree. P_{right}, the subset to the right of l, is stored in the right subtree.
  - At the left child of the root we split P_{left} into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree. The left child itself stores the splitting line.

KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.
  - P_{left}, the subset of points to the left or on l, is stored in the left subtree. P_{right}, the subset to the right of l, is stored in the right subtree.
  - At the left child of the root we split P_{left} into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree. The left child itself stores the splitting line.
  - Similarly the set P_{right} is split with a horizontal line into two subsets, which are stored in the left and right subtree of the right child.

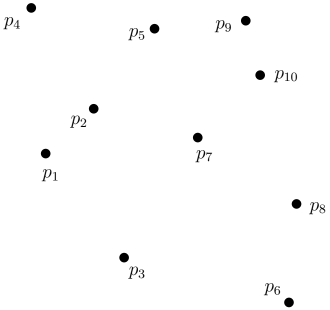
KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.
  - P_{left}, the subset of points to the left or on l, is stored in the left subtree. P_{right}, the subset to the right of l, is stored in the right subtree.
  - At the left child of the root we split P_{left} into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree. The left child itself stores the splitting line.
  - Similarly the set P_{right} is split with a horizontal line into two subsets, which are stored in the left and right subtree of the right child.
  - At the grand children of the root, we split again by a vertical line.

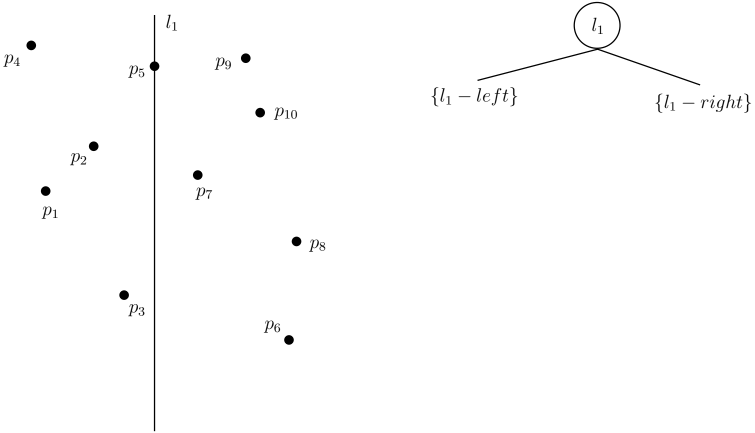
KD-TREE

- Kd-tree is defined as follows
  - At the root we split the set P with a vertical line l into two subsets of roughly equal size and the splitting line is stored at the root.
  - P_{left}, the subset of points to the left or on l, is stored in the left subtree. P_{right}, the subset to the right of l, is stored in the right subtree.
  - At the left child of the root we split P_{left} into two subsets with a horizontal line; the points below or on it are stored in the left subtree of the left child, and the points above it are stored in the right subtree. The left child itself stores the splitting line.
  - Similarly the set P_{right} is split with a horizontal line into two subsets, which are stored in the left and right subtree of the right child.
  - At the grand children of the root, we split again by a vertical line.
  - We split vertically at even depths and split horizontally at odd depths.

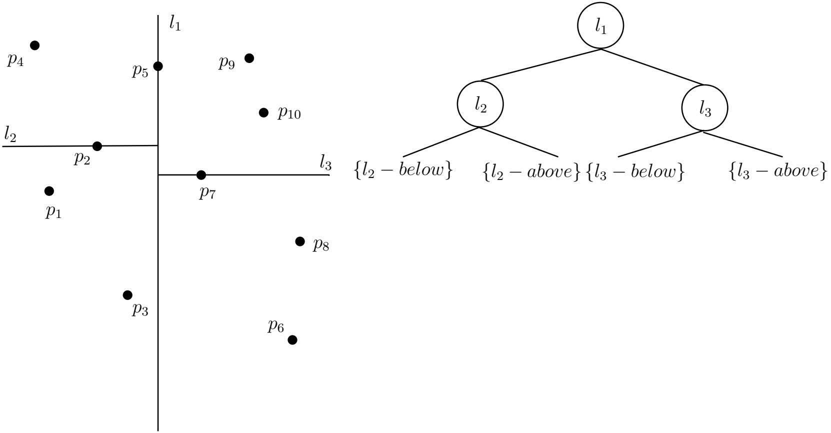
EXAMPLE



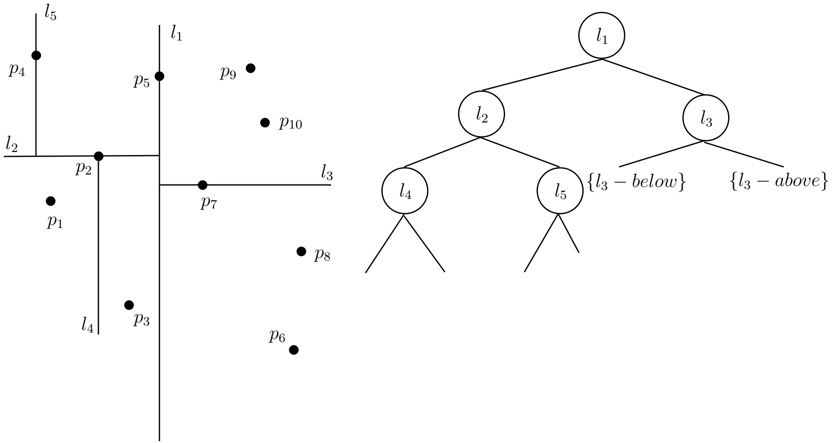
EXAMPLE



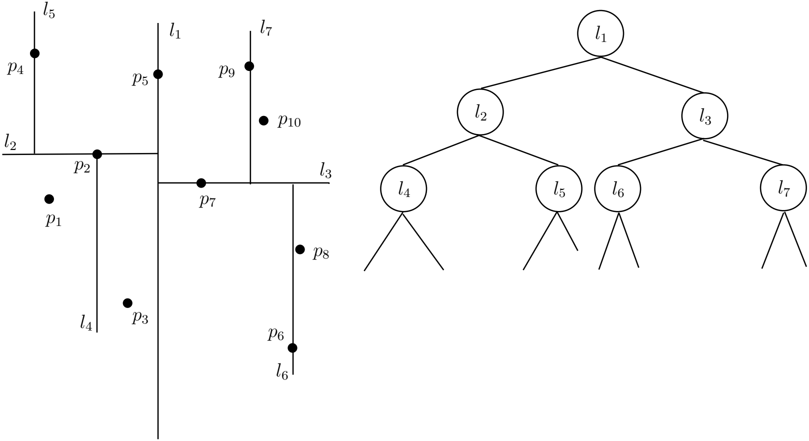
EXAMPLE


l_1
l_2
l_3
p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}
$\{l_2 - below\}$
$\{l_2 - above\}$
$\{l_3 - below\}$
$\{l_3 - above\}$

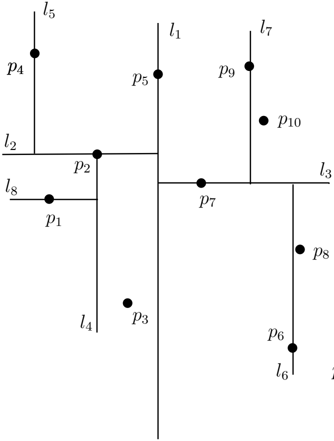
EXAMPLE


l_5
l_1
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
p_7
p_1
p_8
l_4
p_3
p_6
l_1
l_2
l_3
l_4
l_5
$\{l_3 - below\}$
$\{l_3 - above\}$

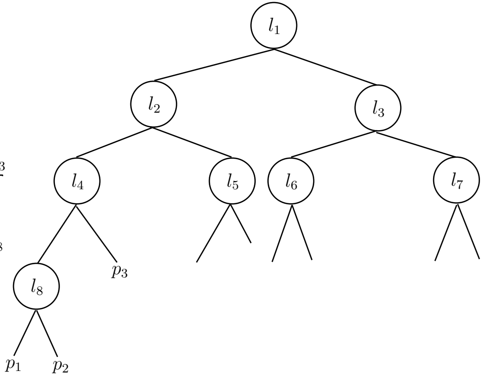
EXAMPLE



EXAMPLE

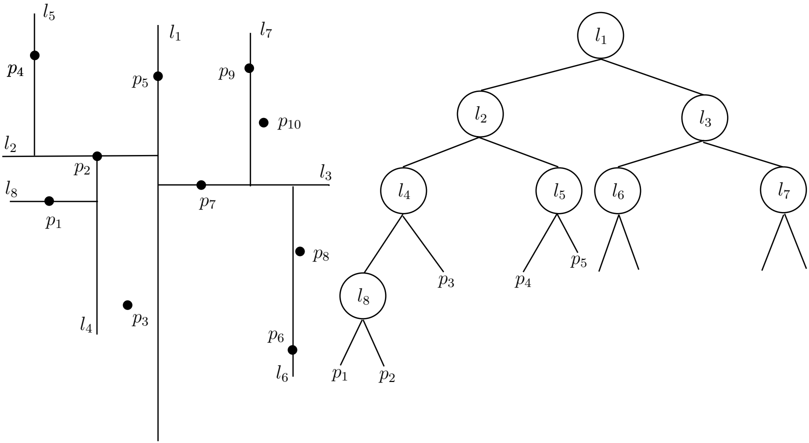


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
p_6
l_6

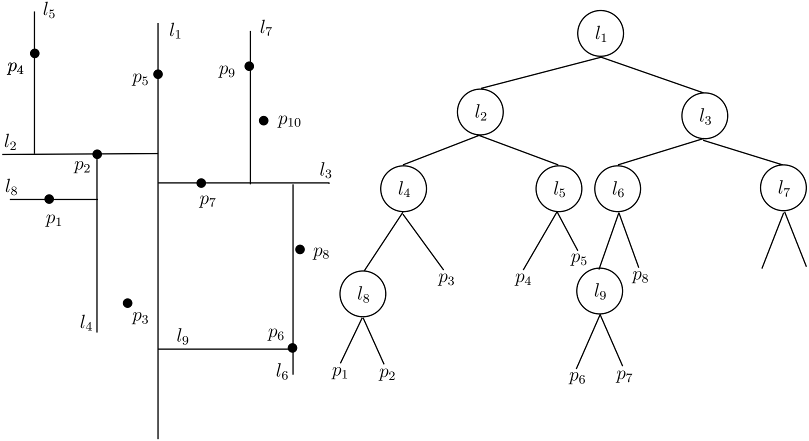


l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_1
p_2

EXAMPLE

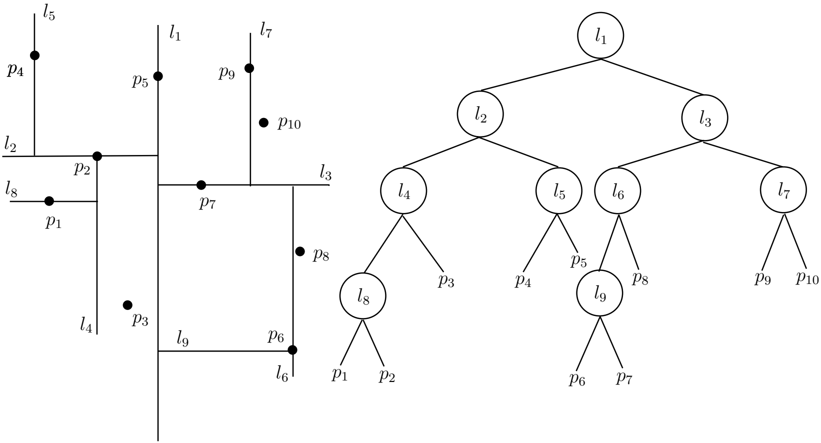

l5
l1
l7
p4
p5
p9
p10
l2
p2
l3
l8
p7
p1
p8
l4
p3
p6
l6
l1
l2
l3
l4
l5
l6
l7
p3
p4
p5
l8
p1
p2

EXAMPLE



l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6
l_1
l_2
l_3
l_4
l_5
l_6
l_7
p_3
p_4
p_5
p_8
l_8
l_9
p_1
p_2
p_6
p_7

EXAMPLE



l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
l_4
p_3
l_9
p_6
l_6
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
l_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

CONSTRUCTION

```
Algorithm BuildKdTree(P, depth)
  If P contains only one point
    then return a leaf storing this point.
  else if depth is even
    l := vertical line through median
           x-coordinates of the points in P
    P1 := set of points to the left of or on l
    P2 := set of points to the right of l
  else
    l := horizontal line through median
           y-coordinates of the points in P
    P1 := set of points below or on l
    P2 := set of points above l
  v-left = BuildKdTree(P1, depth+1)
  v-right = BuildKdTree(P2, depth+1)
  return a node v with value l, left child
           v-left, and right child v-right
```

COMPLEXITY

- In the pseudocode, median of a set of n numbers is assumed to be the $\lfloor n/2 \rfloor$-th smallest number.

COMPLEXITY

- In the pseudocode, median of a set of n numbers is assumed to be the $\lfloor n/2 \rfloor$-th smallest number.
- Assuming that point set P is given as two lists one sorted with respect to x-coordinates of the points in P and the other sorted with respect to y-coordinates of the points, construction time $T(n)$ of the kd-tree satisfies the recurrence relation

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

COMPLEXITY

- In the pseudocode, median of a set of n numbers is assumed to be the $\lfloor n/2 \rfloor$-th smallest number.
- Assuming that point set P is given as two lists one sorted with respect to x-coordinates of the points in P and the other sorted with respect to y-coordinates of the points, construction time $T(n)$ of the kd-tree satisfies the recurrence relation

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

- Hence, a kd-tree for a set of n points uses $O(n)$ storage and can be constructed in $O(n \log n)$ time.

QUERY

- We now move to the query algorithm.

QUERY

- We now move to the query algorithm.
- First observe that, in a kd-tree, each node corresponds to a region of the plane.

QUERY

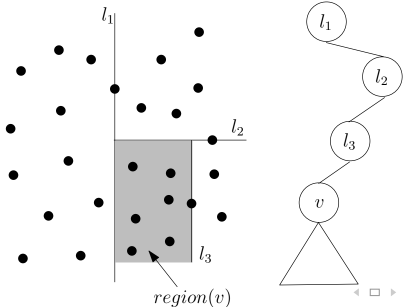
- We now move to the query algorithm.
- First observe that, in a kd-tree, each node corresponds to a region of the plane.
- In general, region corresponding to a node v, which we denote by $region(v)$, is a rectangle which can be unbounded on one or more sides.

QUERY

- We now move to the query algorithm.
- First observe that, in a kd-tree, each node corresponds to a region of the plane.
- In general, region corresponding to a node v, which we denote by $region(v)$, is a rectangle which can be unbounded on one or more sides.
- It is bounded by the splitting lines stored at its ancestors.

QUERY

- We now move to the query algorithm.
- First observe that, in a kd-tree, each node corresponds to a region of the plane.
- In general, region corresponding to a node v, which we denote by *region*(v), is a rectangle which can be unbounded on one or more sides.
- It is bounded by the splitting lines stored at its ancestors.
- Following figure makes this clear:



QUERY

- Given the notion of region, we can now describe the query algorithm:

QUERY

- Given the notion of region, we can now describe the query algorithm:
  - Given a query rectangle Q , we traverse the kd-tree but visit only those nodes v for which $region(v) \cap Q \neq \phi$.

QUERY

- Given the notion of region, we can now describe the query algorithm:
  - Given a query rectangle Q , we traverse the kd-tree but visit only those nodes v for which $region(v) \cap Q \neq \phi$.
  - When $region(v) \subset Q$ we report all the points stored in the subtree of v.

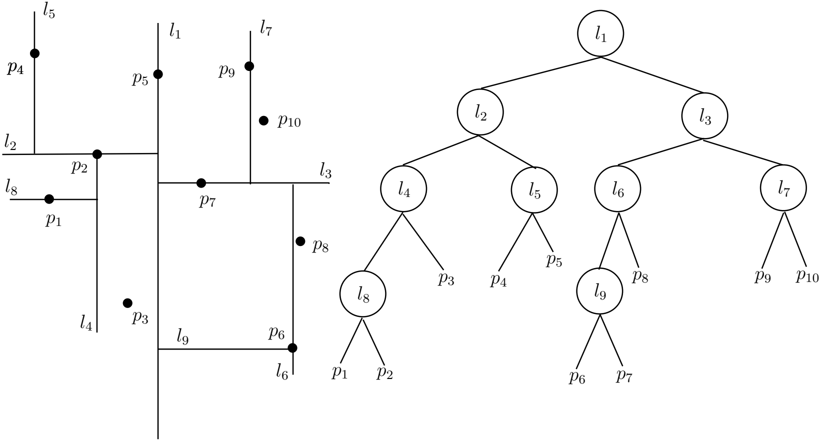
QUERY

- Given the notion of region, we can now describe the query algorithm:
  - Given a query rectangle Q , we traverse the kd-tree but visit only those nodes v for which $region(v) \cap Q \neq \phi$.
  - When $region(v) \subset Q$ we report all the points stored in the subtree of v.
  - When the traversal reaches a leaf p_i, we explicitly check whether $p_i \in Q$ and, if so, report p_i.

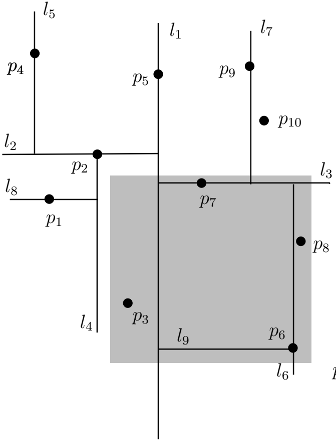
QUERY

- Given the notion of region, we can now describe the query algorithm:
  - Given a query rectangle Q , we traverse the kd-tree but visit only those nodes v for which $region(v) \cap Q \neq \phi$.
  - When $region(v) \subset Q$ we report all the points stored in the subtree of v.
  - When the traversal reaches a leaf p_i, we explicitly check whether $p_i \in Q$ and, if so, report p_i.
- An example follows.

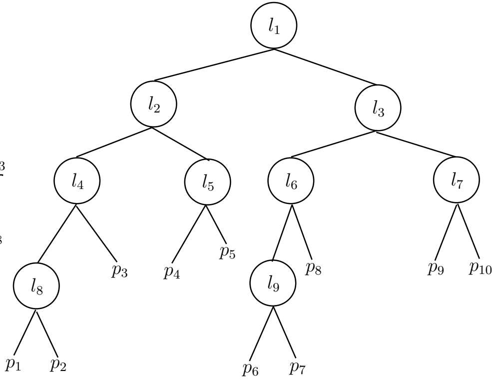
EXAMPLE


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
l_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

EXAMPLE

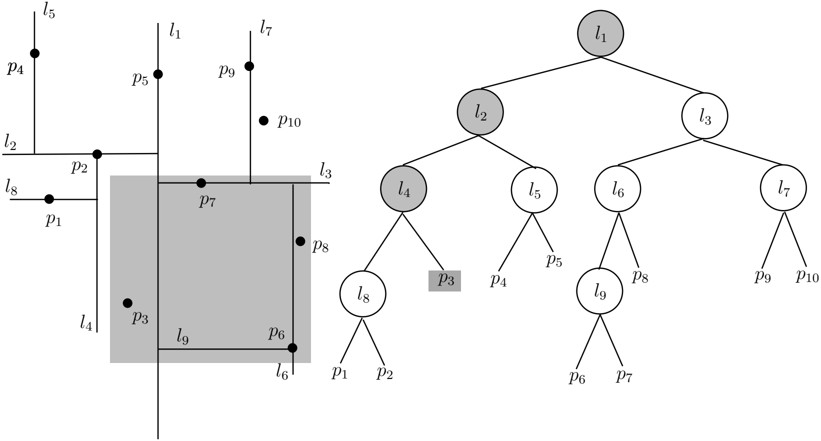


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6



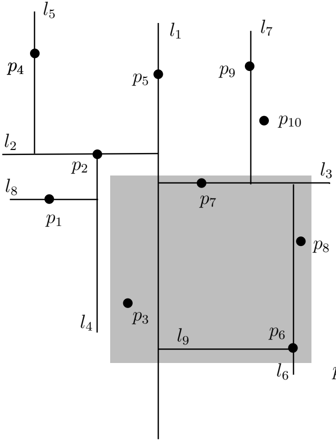
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
l_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

EXAMPLE

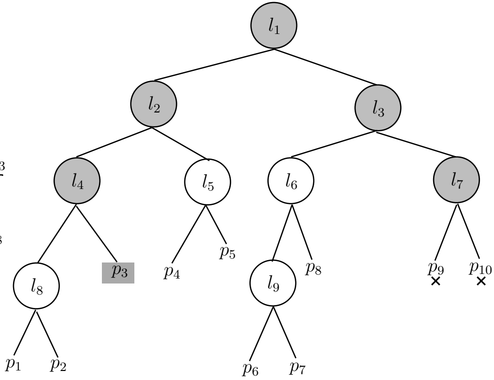


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
l_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

EXAMPLE

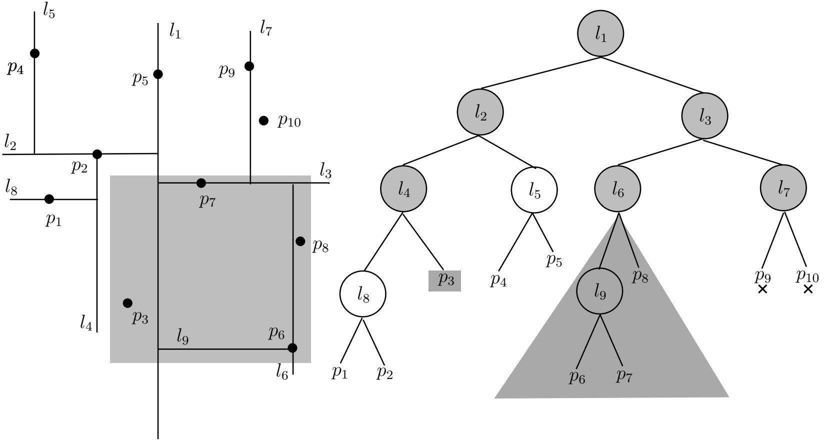


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6



l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
l_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

EXAMPLE


l_5
l_1
l_7
p_4
p_5
p_9
p_{10}
l_2
p_2
l_3
l_8
p_7
p_1
p_8
p_3
l_4
l_9
p_6
l_6
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
p_3
p_4
p_5
p_8
p_9
p_{10}
l_9
p_1
p_2
p_6
p_7

PSEUDOCODE

```
Algorithm SearchKdTree(v, Q)
  If v is a leaf
    then report the point stored at v if it
           lies in Q
  else if region(lc(v)) is fully contained in Q
    then ReportSubtree(lc(v))
  else if region(lc(v)) intersects Q
    then SearchKdTree(lc(v), Q)
    If region(rc(v)) is fully contained in Q
      then ReportSubtree(rc(v))
    else if region(rc(v)) intersects Q
      then SearchKdTree(rc(v), Q)
```

QUERY

- Remaining question is, for a node v how to compute and maintain $region(v)$ so that intersection with Q can be tested?

QUERY

- Remaining question is, for a node v how to compute and maintain $region(v)$ so that intersection with Q can be tested? Two possibilities are there:

- Compute $region(v)$ for all nodes v during the preprocessing phase and store it.

QUERY

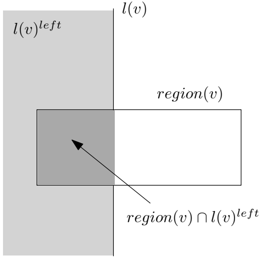
- Remaining question is, for a node v how to compute and maintain $region(v)$ so that intersection with Q can be tested? Two possibilities are there:

- Compute $region(v)$ for all nodes v during the preprocessing phase and store it.
- Maintain the current region through the recursive calls using the lines stored in the internal nodes.

QUERY

- Remaining question is, for a node v how to compute and maintain $region(v)$ so that intersection with Q can be tested? Two possibilities are there:

- Compute $region(v)$ for all nodes v during the preprocessing phase and store it.
- Maintain the current region through the recursive calls using the lines stored in the internal nodes.
  For instance, the region corresponding to the left child of a node v at even depth can be computed as
  $region(lc(v)) = region(v) \cap l(v)^{left}$
  where $l(v)$ is the splitting line stored at v, and $l(v)^{left}$ is the half-plane to the left of and including $l(v)$.



COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.

COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.
- One group consists of those nodes for which the algorithm traverses corresponding subtrees and report points stored there.

COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.
- One group consists of those nodes for which the algorithm traverses corresponding subtrees and report points stored there.
- Total time required for processing these nodes is $O(k)$, where k is the total number of reported points.

COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.
- One group consists of those nodes for which the algorithm traverses corresponding subtrees and report points stored there.
- Total time required for processing these nodes is $O(k)$, where k is the total number of reported points.
- For the other group of nodes, the algorithm takes constant time for processing each node.

COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.
- One group consists of those nodes for which the algorithm traverses corresponding subtrees and report points stored there.
- Total time required for processing these nodes is $O(k)$, where k is the total number of reported points.
- For the other group of nodes, the algorithm takes constant time for processing each node.
- An upper bound for the number of such nodes is given by the recurrence relation

$$T'(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T'(n/4) + 2 & \text{if } n > 1 \end{cases}$$

COMPLEXITY

- The nodes visited by the algorithm during the search can be classified into two groups.
- One group consists of those nodes for which the algorithm traverses corresponding subtrees and report points stored there.
- Total time required for processing these nodes is $O(k)$, where k is the total number of reported points.
- For the other group of nodes, the algorithm takes constant time for processing each node.
- An upper bound for the number of such nodes is given by the recurrence relation

$$T'(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T'(n/4) + 2 & \text{if } n > 1 \end{cases}$$

- Solution of this recurrence is $O(\sqrt{n})$.

RESULT

Theorem

A kd-tree for a set P of n points in the plane uses $O(n)$ space 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.

Range tree

- To further improve the query time of the 2-dimensional rectangular range searching problem we introduce another data structure called range tree.

RANGE TREE

- To further improve the query time of the 2-dimensional rectangular range searching problem we introduce another data structure called range tree.
- We have observed earlier that a 2-dimensional range query is essentially composed of two 1-dimensional sub-queries, one on the x-coordinate of points and one on the y-coordinate.

RANGE TREE

- To further improve the query time of the 2-dimensional rectangular range searching problem we introduce another data structure called range tree.
- We have observed earlier that a 2-dimensional range query is essentially composed of two 1-dimensional sub-queries, one on the x-coordinate of points and one on the y-coordinate.
- One idea from this observation leads to Kd-tree. We now use the same observation in a different way to obtain range tree.

RANGE TREE

- Let P be a set of n points in the plane that we want to preprocess for rectangular range query. Let $Q := [x : x'] \times [y : y']$ be the query range.

RANGE TREE

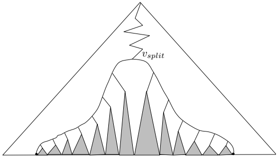
- Let P be a set of n points in the plane that we want to preprocess for rectangular range query. Let $Q := [x : x'] \times [y : y']$ be the query range.

- If we only care about the x-coordinate then the query is a 1-dimensional range query.

RANGE TREE

- Let P be a set of n points in the plane that we want to preprocess for rectangular range query. Let $Q := [x : x'] \times [y : y']$ be the query range.

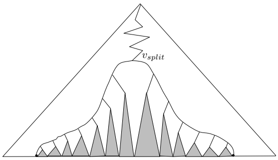
- If we only care about the x-coordinate then the query is a 1-dimensional range query.
- Recall that our algorithm for this is to first locate the split node and then traverse along two search paths towards the leaves. While traversing the search paths, our algorithm also reported fully all the subtrees lying between the search paths.



RANGE TREE

- Let P be a set of n points in the plane that we want to preprocess for rectangular range query. Let $Q := [x : x'] \times [y : y']$ be the query range.

- If we only care about the x-coordinate then the query is a 1-dimensional range query.
- Recall that our algorithm for this is to first locate the split node and then traverse along two search paths towards the leaves. While traversing the search paths, our algorithm also reported fully all the subtrees lying between the search paths.
- For any query range, $O(\log n)$ mutually disjoint subtrees are selected.



RANGE TREE

- Let us call the subset of points stored in the leaves of the subtree rooted at a node v the canonical subset of v, denoted by $P(v)$.

RANGE TREE

- Let us call the subset of points stored in the leaves of the subtree rooted at a node v the canonical subset of v, denoted by $P(v)$.
- Hence, the subset of points whose x-coordinate lies in a query range can be expressed as the disjoint union of $O(\log n)$ canonical subsets. Of these points, those having y-coordinate in the interval $[y : y']$ are to be reported.

Range tree

- Let us call the subset of points stored in the leaves of the subtree rooted at a node v the canonical subset of v, denoted by $P(v)$.
- Hence, the subset of points whose x-coordinate lies in a query range can be expressed as the disjoint union of $O(\log n)$ canonical subsets. Of these points, those having y-coordinate in the interval $[y : y']$ are to be reported.
- This implies that, after performing 1-dimensional query on x-coordinate, we have to perform $O(\log n)$ 1-dimensional query on y-coordinate each on an appropriate $P(v)$.

RANGE TREE

- We are thus lead to the following data structure for rectangular range queries on a set P of n points in the plane.

RANGE TREE

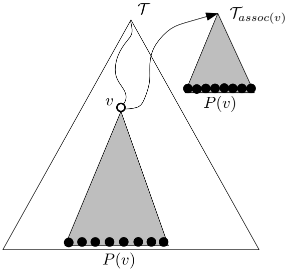
- We are thus lead to the following data structure for rectangular range queries on a set P of n points in the plane.

- The main tree is a balanced binary tree $\mathcal{T}$ built on the x-coordinate of the points in P.

RANGE TREE

- We are thus lead to the following data structure for rectangular range queries on a set P of n points in the plane.

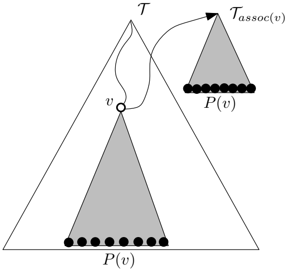
- The main tree is a balanced binary tree $\mathcal{T}$ built on the x-coordinate of the points in P.
- For any internal or leaf node v in $\mathcal{T}$, the canonical subset $P(v)$ is stored in a balanced binary search tree $\mathcal{T}_{assoc(v)}$ on the y-coordinate of the points. The node v stores a pointer to the root of $\mathcal{T}_{assoc(v)}$, which is called the associated structure of v.



RANGE TREE

- We are thus lead to the following data structure for rectangular range queries on a set P of n points in the plane.

- The main tree is a balanced binary tree $\mathcal{T}$ built on the x-coordinate of the points in P.
- For any internal or leaf node v in $\mathcal{T}$, the canonical subset $P(v)$ is stored in a balanced binary search tree $\mathcal{T}_{assoc(v)}$ on the y-coordinate of the points. The node v stores a pointer to the root of $\mathcal{T}_{assoc(v)}$, which is called the associated structure of v.



- This multi-level data structure is called range tree.

Construction

```
Algorithm Build2dRangeTree(P)
  Build the associated BBST Tassoc on the set Py
     of y-coordinates of the points in P. Leaves
     of Tassoc stores both the points and their
     y-coordinates.
  If P contain only one point
    then create a leaf v storing this point and
       make Tassoc the associate structure of v
  else
    Split P into two subsets Pleft containing
       points with x-coordinates less than or
       equal to median x-coordinate and Pright
       containing points with x-coordinates
       greater than median x-coordinate
```

CONSTRUCTION

```
        vleft := Build2dRangeTree(Pleft)
        vright := Build2dRangeTree(Pright)
        Create node v containing the median
           x-coordinate, vleft as left child, vright
           as right child, and Tassoc as associated
           structure
    return v
```

Complexity

- A range tree on a set of n points in the plane requires $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time.

COMPLEXITY

- A range tree on a set of n points in the plane requires $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time.
- Storage complexity follows because (i) the main tree requires $O(n)$ storage, (ii) a point p in P is stored only in the associated structure of nodes on the path in $\mathcal{T}$ towards the leaf containing p and (iii) a path contains at most $O(\log n)$ nodes.

Complexity

- A range tree on a set of n points in the plane requires $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time.
- Storage complexity follows because (i) the main tree requires $O(n)$ storage, (ii) a point p in P is stored only in the associated structure of nodes on the path in $\mathcal{T}$ towards the leaf containing p and (iii) a path contains at most $O(\log n)$ nodes.
- Regarding time complexity, we assume that the points are maintained in two lists, one sorted on x-coordinate of the points and the other on y-coordinate. Then from similar arguments as used in storage complexity the time complexity follows.

QUERY

- The query algorithm first selects, by the application of our 1-dimensional query algorithm, $O(\log n)$ canonical subsets that together contain points whose x-coordinate lie in the range $[x : x']$.

QUERY

- The query algorithm first selects, by the application of our 1-dimensional query algorithm, $O(\log n)$ canonical subsets that together contain points whose x-coordinate lie in the range $[x : x']$.
- Of these subsets, we then report the points whose y-coordinate lie in the range $[y : y']$. This can be done by applying the same 1-dimensional query algorithm on each of the $O(\log n)$ associated structures that stores the selected canonical subsets.

Complexity

- At each node v in the main tree $\mathcal{T}$ we spend constant time to decide whether to go left or right and possibly call 1-dimensional query algorithm on the associated structure. Total time spent for v is thus $O(\log n + k_v)$, where k_v is the number of points reported.

COMPLEXITY

- At each node v in the main tree $\mathcal{T}$ we spend constant time to decide whether to go left or right and possibly call 1-dimensional query algorithm on the associated structure. Total time spent for v is thus $O(\log n + k_v)$, where k_v is the number of points reported.
- Hence the total time spent on a search path is $\sum(O(\log n + k_v)$, where the summation extends over the nodes in the search path.

COMPLEXITY

- At each node v in the main tree $\mathcal{T}$ we spend constant time to decide whether to go left or right and possibly call 1-dimensional query algorithm on the associated structure. Total time spent for v is thus $O(\log n + k_v)$, where k_v is the number of points reported.
- Hence the total time spent on a search path is $\sum(O(\log n + k_v)$, where the summation extends over the nodes in the search path.
- Since there are $O(\log n)$ nodes in a search path and two such paths are involved in each query, total time is $O(\log^2 n + k)$ where $k = \sum k_v$ is the total number of points reported.

Result

Theorem

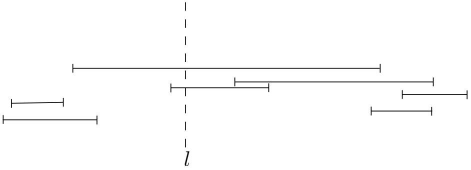
Let P be a set of n points in the plane. A range tree for P uses $O(n \log n)$ storage and can be constructed in $O(n \log n)$ time. By querying this range tree one can report the points in P that lie in a rectangular query range in $O(\log^2 n + k)$ time where k is the number of reported points.

OUTLINE

SEGMENT SEARCHING PROBLEM

Problem

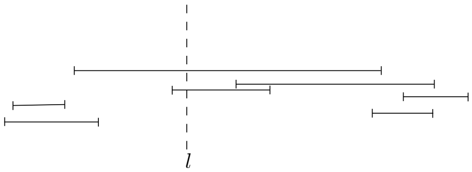
Given a set S of n horizontal line segments in the plane, preprocess them such that the segments intersecting a vertical query line l can be reported efficiently.



SEGMENT SEARCHING PROBLEM

Problem

Given a set S of n horizontal line segments in the plane, preprocess them such that the segments intersecting a vertical query line l can be reported efficiently.

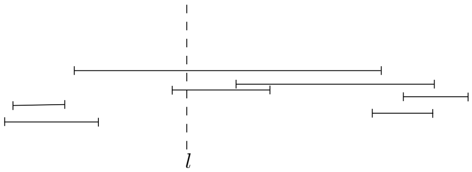


- Obviously, the problem can be solved in $O(n)$ time and there exists instances for which we have to report all the segments.

SEGMENT SEARCHING PROBLEM

Problem

Given a set S of n horizontal line segments in the plane, preprocess them such that the segments intersecting a vertical query line l can be reported efficiently.



- Obviously, the problem can be solved in $O(n)$ time and there exists instances for which we have to report all the segments.
- However, we are interested in an output-sensitive algorithm which is efficient in multi-shot query scenario.

Interval searching problem

- We first reduce the segment search problem to an 1-dimensional interval searching problem.

INTERVAL SEARCHING PROBLEM

- We first reduce the segment search problem to an 1-dimensional interval searching problem.
- Let $l := (x = q_x)$ denote the query line.

INTERVAL SEARCHING PROBLEM

- We first reduce the segment search problem to an 1-dimensional interval searching problem.
- Let $l := (x = q_x)$ denote the query line.
- A horizontal segment $s := (x, y)(x', y)$ is intersected by l if and only if $x \leq q_x \leq x'$. So only x-coordinates of the segment endpoints play a role here.

INTERVAL SEARCHING PROBLEM

- We first reduce the segment search problem to an 1-dimensional interval searching problem.
- Let $l := (x = q_x)$ denote the query line.
- A horizontal segment $s := (x, y)(x', y)$ is intersected by l if and only if $x \leq q_x \leq x'$. So only x-coordinates of the segment endpoints play a role here.
- Hence the problem becomes 1-dimensional and can be stated as follows.

Problem

Given a set of intervals on the real line, report the ones that contain the query point q_x.

DATA STRUCTURE

- Let $I = \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be the set of n closed intervals on the real line.

DATA STRUCTURE

- Let $I = \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be the set of n closed intervals on the real line.
- Let x_{mid} be the median of the $2n$ interval endpoints. So at most half of the interval endpoints lies to the left of x_{mid} and at most half of the endpoints lies to the right of x_{mid}.

DATA STRUCTURE

- Let $I = \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be the set of n closed intervals on the real line.
- Let x_{mid} be the median of the $2n$ interval endpoints. So at most half of the interval endpoints lies to the left of x_{mid} and at most half of the endpoints lies to the right of x_{mid}.
- If the query value q_x lies to the left of x_{mid} then the intervals that lie completely to the right of x_{mid} obviously do not contain q_x.

DATA STRUCTURE

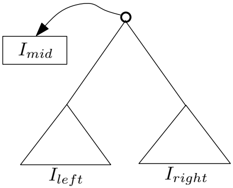
- Let $I = \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be the set of n closed intervals on the real line.
- Let x_{mid} be the median of the $2n$ interval endpoints. So at most half of the interval endpoints lies to the left of x_{mid} and at most half of the endpoints lies to the right of x_{mid}.
- If the query value q_x lies to the left of x_{mid} then the intervals that lie completely to the right of x_{mid} obviously do not contain q_x.
- We construct a binary search tree based on this idea.

Data structure

- The root of the tree contains x_{mid}. The right subtree of the tree stores the set I_{right} of the intervals lying completely to the right of x_{mid}, and the left subtree stores the set I_{left} of intervals completely to the right of x_{mid}.

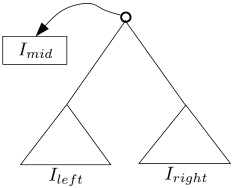
DATA STRUCTURE

- The root of the tree contains x_{mid}. The right subtree of the tree stores the set I_{right} of the intervals lying completely to the right of x_{mid}, and the left subtree stores the set I_{left} of intervals completely to the right of x_{mid}.
- The set I_{mid} of intervals containing x_{mid} is stored in a separate structure and we associate that structure with the root of our tree.



DATA STRUCTURE

- The root of the tree contains x_{mid}. The right subtree of the tree stores the set I_{right} of the intervals lying completely to the right of x_{mid}, and the left subtree stores the set I_{left} of intervals completely to the right of x_{mid}.
- The set I_{mid} of intervals containing x_{mid} is stored in a separate structure and we associate that structure with the root of our tree.



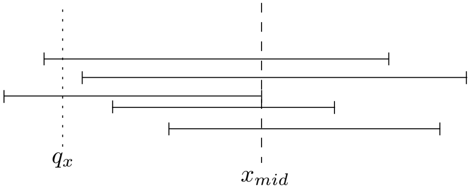
- The subtrees are constructed recursively in the same way.

DATA STRUCTURE

- We must now fix the associated structure such that it enables us to report the intervals in I_{mid} that contain a given q_x.

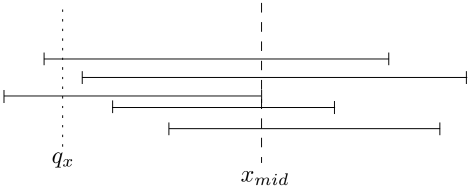
DATA STRUCTURE

- We must now fix the associated structure such that it enables us to report the intervals in I_{mid} that contain a given q_x.
- Observe that if q_x is on left of I_{mid}, then right endpoint of all the intervals are on right of q_x. Similarly, if q_x is on right of I_{mid} all left endpoints are on its left.



DATA STRUCTURE

- We must now fix the associated structure such that it enables us to report the intervals in I_{mid} that contain a given q_x.
- Observe that if q_x is on left of I_{mid}, then right endpoint of all the intervals are on right of q_x. Similarly, if q_x is on right of I_{mid} all left endpoints are on its left.



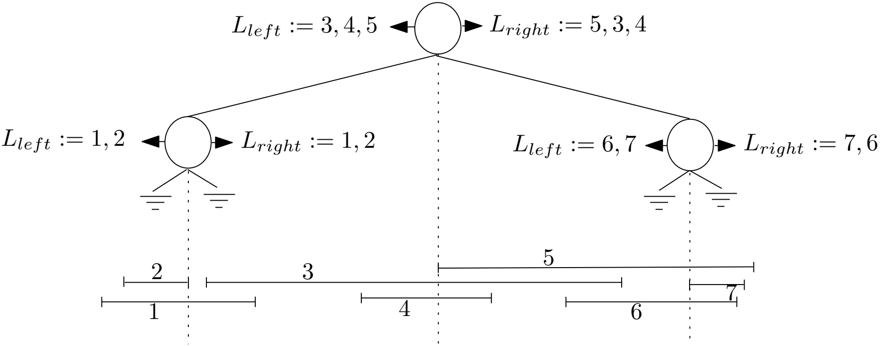
- We thus maintain two sorted lists of the intervals in I_{mid}, one sorted on left endpoints and the on right endpoints. A traversal of the appropriate list enable us to report intervals containing q_x in time proportional to the number of intervals reported.

INTERVAL TREE

- The whole data structure we thus arrived at for storing a given set I of intervals is called an interval tree.

INTERVAL TREE

- The whole data structure we thus arrived at for storing a given set I of intervals is called an interval tree.
- Following figure shows an example.



CONSTRUCTION

```
Algorithm ConstructIntervalTree(I)
  If I = Null
    then return an empty leaf
  Create a node v. Compute x-mid, the median
     of the set of interval endpoints and
     store x-mid with v
  Compute I-mid and construct two sorted lists
     for I-mid: a list L-left(v) sorted on left
     endpoint and a list L-right(v) sorted on
     right endpoint
  Store these two lists at v
  lc(v) := ConstructIntervalTree(I-left)
  rc(v) := ConstructIntervalTree(I-right)
  return v
```

COMPLEXITY

- Observe that each interval is only stored in a set I_{mid} once and, hence, only appears once in each of the two sorted lists. So total amount of storage required for all associated lists is bounded by $O(n)$. The tree itself uses $O(n)$ storage. Hence an interval tree on n intervals requires $O(n)$ storage.

Complexity

- Observe that each interval is only stored in a set I_{mid} once and, hence, only appears once in each of the two sorted lists. So total amount of storage required for all associated lists is bounded by $O(n)$. The tree itself uses $O(n)$ storage. Hence an interval tree on n intervals requires $O(n)$ storage.
- An interval tree on n intervals has $O(\log n)$ depth.

COMPLEXITY

- Observe that each interval is only stored in a set I_{mid} once and, hence, only appears once in each of the two sorted lists. So total amount of storage required for all associated lists is bounded by $O(n)$. The tree itself uses $O(n)$ storage. Hence an interval tree on n intervals requires $O(n)$ storage.
- An interval tree on n intervals has $O(\log n)$ depth.
- We assume that the set of endpoints are presorted. So median can be computed in constant time. I_{mid}, I_{left}, and I_{right} can be computed in $O(n)$ time. Creating the lists $L_{left}(v)$ $L_{right}(v)$ takes $O(|I_{mid}| \log |I_{mid}|)$ time. Hence time spend for creating the node v (not counting the recursive calls) is $O(n + |I_{mid}| \log |I_{mid}|)$. So total construction time is $O(n \log n)$.

QUERY PSEUDOCODE

```
Algorithm QueryIntervalTree(v,qx)
  If v is not a leaf
    then if qx < x-mid(v)
           then Walk along the list L-left(v),
                   starting at the interval with
                   the leftmost endpoint, reporting
                   all the intervals that contain
                   qx.
              QueryIntervalTree(lc(v),qx)
         else
           Walk along the list L-right(v),
                   starting at the interval with
                   the rightmost endpoint, reporting
                   all the intervals that contain
                   qx.
              QueryIntervalTree(rc(v),qx)
```

Query complexity

- At any node v we visit, we spend $O(1 + k_v)$ time, where k_v is the number of intervals that we report at v.

QUERY COMPLEXITY

- At any node v we visit, we spend $O(1 + k_v)$ time, where k_v is the number of intervals that we report at v.
- The sum of the k_v's over all visited nodes is k, total number of intervals reported.

QUERY COMPLEXITY

- At any node v we visit, we spend $O(1 + k_v)$ time, where k_v is the number of intervals that we report at v.
- The sum of the k_v's over all visited nodes is k, total number of intervals reported.
- Depth of the tree is $O(\log n)$.

QUERY COMPLEXITY

- At any node v we visit, we spend $O(1 + k_v)$ time, where k_v is the number of intervals that we report at v.
- The sum of the k_v's over all visited nodes is k, total number of intervals reported.
- Depth of the tree is $O(\log n)$.
- Hence, total query time is $O(\log n + k)$.

RESULT

Theorem

An interval tree for a set I of n intervals uses $O(n)$ *storage and can be built in* $O(n \log n)$ *time. Using the interval tree we can report all intervals that contain a query point in* $O(\log n + k)$ *time, where k is the number of reported intervals.*

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Let us consider a slightly more difficult segment search problem.

Problem

Given a set S of n horizontal line segments in the plane, preprocess them such that the segments intersecting a vertical query segment q can be reported efficiently.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Let us consider a slightly more difficult segment search problem.

Problem

Given a set S of n horizontal line segments in the plane, preprocess them such that the segments intersecting a vertical query segment q can be reported efficiently.

- We next show that the problem can be solved by using a data structure which is a modified form of interval tree.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Let q be the vertical query segment $q_x \times [q_y : q'_y]$.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Let q be the vertical query segment $q_x \times [q_y : q'_y]$.
- For a segment $s := [s_x : s'_x] \times s_y$ in S, $[s_x : s'_x]$ is called x-interval of the segment.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Let q be the vertical query segment $q_x \times [q_y : q'_y]$.
- For a segment $s := [s_x : s'_x] \times s_y$ in S, $[s_x : s'_x]$ is called x-interval of the segment.
- Suppose we have stored the segments in S in an interval tree $\mathcal{T}$ according to their x intervals.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

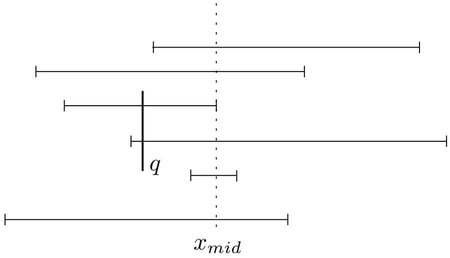
- Let q be the vertical query segment $q_x \times [q_y : q'_y]$.
- For a segment $s := [s_x : s'_x] \times s_y$ in S, $[s_x : s'_x]$ is called x-interval of the segment.
- Suppose we have stored the segments in S in an interval tree $\mathcal{T}$ according to their x intervals.
- If we use our `QueryIntervalTree` procedure to query $\mathcal{T}$ with a vertical query segment q, we can traverse the tree properly but problem arise when trying to report segments from I_{mid} containing q.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Suppose q_x lies to the left of x_{mid}. For a segment $s \in I_{mid}$ to be intersected by q, it is not sufficient that its left endpoint lies to the left of q, it is also required that its y-coordinate lies in the range $[q_y : q'_y]$.

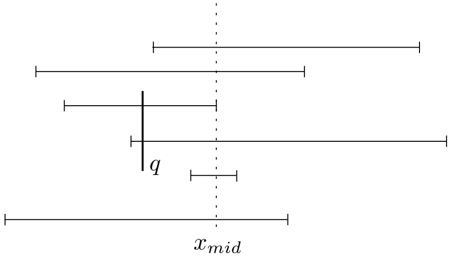
SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Suppose q_x lies to the left of x_{mid}. For a segment $s \in I_{mid}$ to be intersected by q, it is not sufficient that its left endpoint lies to the left of q, it is also required that its y-coordinate lies in the range $[q_y : q'_y]$.



SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

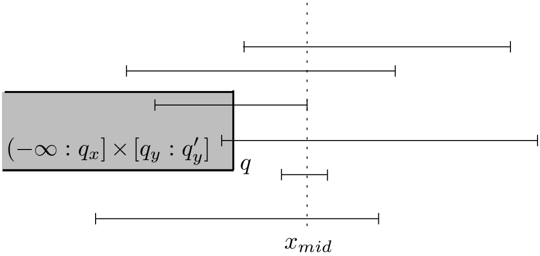
- Suppose q_x lies to the left of x_{mid}. For a segment $s \in I_{mid}$ to be intersected by q, it is not sufficient that its left endpoint lies to the left of q, it is also required that its y-coordinate lies in the range $[q_y : q'_y]$.



- So storing the endpoints in an ordered list is not enough.

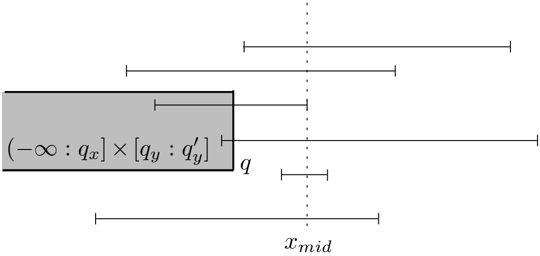
SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- We need an associated structure such that, given a query range $(-\infty : q_x] \times [q_y : q'_y]$, we must be able to report all the segments whose left endpoints lies in that range.



SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- We need an associated structure such that, given a query range $(-\infty : q_x] \times [q_y : q'_y]$, we must be able to report all the segments whose left endpoints lies in that range.



- Similarly, if q lies to the right of x_{mid}, we must be able to report all the segments whose right endpoints lies in the range $[q_x : +\infty) \times [q_y : q'_y]$.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Observe that this is nothing more than a 2-dimensional rectangular range query on a set of points, and we have already solved it.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Observe that this is nothing more than a 2-dimensional rectangular range query on a set of points, and we have already solved it.
- We can now spell out the modified data structure for storing the set S of horizontal line segments.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Observe that this is nothing more than a 2-dimensional rectangular range query on a set of points, and we have already solved it.
- We can now spell out the modified data structure for storing the set S of horizontal line segments.
- The main structure is an interval tree $\mathcal{T}$ on the x-intervals of the segments.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Observe that this is nothing more than a 2-dimensional rectangular range query on a set of points, and we have already solved it.
- We can now spell out the modified data structure for storing the set S of horizontal line segments.
- The main structure is an interval tree $\mathcal{T}$ on the x-intervals of the segments.
- Instead of the sorted lists $L_{left}(v)$ and $L_{right}(v)$, we have two range trees: a range tree $\mathcal{T}_{left}(v)$ on the left endpoints of the segments in $I_{mid}(v)$, and a range tree $\mathcal{T}_{right}(v)$ on the right endpoints of the segments in $I_{mid}(v)$.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Since storage requirement for range tree is a factor $\log n$ larger than that of sorted list, storage requirement of the modified structure is $O(n \log n)$.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Since storage requirement for range tree is a factor $\log n$ larger than that of sorted list, storage requirement of the modified structure is $O(n \log n)$.
- The construction time remains $O(n \log n)$.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Since storage requirement for range tree is a factor $\log n$ larger than that of sorted list, storage requirement of the modified structure is $O(n \log n)$.
- The construction time remains $O(n \log n)$.
- The query algorithm remains same except that instead of walking along the sorted list of segment endpoints, we perform query in the range tree.

SEGMENT SEARCH WITH VERTICAL QUERY SEGMENT

- Since storage requirement for range tree is a factor $\log n$ larger than that of sorted list, storage requirement of the modified structure is $O(n \log n)$.
- The construction time remains $O(n \log n)$.
- The query algorithm remains same except that instead of walking along the sorted list of segment endpoints, we perform query in the range tree.
- So at each of the $O(\log n)$ nodes v on the search path we spend $O(\log n + k_v)$ time, where k_v is the number of reported segments.

Segment search with vertical query segment

- Since storage requirement for range tree is a factor $\log n$ larger than that of sorted list, storage requirement of the modified structure is $O(n \log n)$.
- The construction time remains $O(n \log n)$.
- The query algorithm remains same except that instead of walking along the sorted list of segment endpoints, we perform query in the range tree.
- So at each of the $O(\log n)$ nodes v on the search path we spend $O(\log n + k_v)$ time, where k_v is the number of reported segments.
- The total query time therefore becomes $O(\log^2 n + k)$.

RESULT

Theorem

Let S be a set of n horizontal segments in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ time with a data structure that uses $O(n \log n)$ storage, where k is the number of reported segments. The structure can be built in $O(n \log n)$ time.

WINDOWING QUERY PROBLEM

- The tools we have thus developed can be used to solve another important query problem, known as windowing query problem. A simplified version of the problem is as follows.

Problem

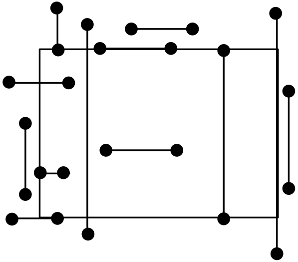
Let S be a set of n axis-parallel and mutually disjoint line segments in the plane. Preprocess the segments such that segments intersecting a query window $W := [x : x'] \times [y : y']$ can be reported efficiently.

WINDOWING QUERY PROBLEM

- The tools we have thus developed can be used to solve another important query problem, known as windowing query problem. A simplified version of the problem is as follows.

Problem

Let S be a set of n axis-parallel and mutually disjoint line segments in the plane. Preprocess the segments such that segments intersecting a query window $W := [x : x'] \times [y : y']$ can be reported efficiently.



Windowing query problem

- Though the segments can intersect the query window in a variety of ways, in most of the cases intersecting segments have at least one endpoint inside the window.

WINDOWING QUERY PROBLEM

- Though the segments can intersect the query window in a variety of ways, in most of the cases intersecting segments have at least one endpoint inside the window.
- Remaining intersecting segments cross window boundary twice.

WINDOWING QUERY PROBLEM

- Though the segments can intersect the query window in a variety of ways, in most of the cases intersecting segments have at least one endpoint inside the window.
- Remaining intersecting segments cross window boundary twice.
- First category of intersecting segments can be identified by using the range query data structure we have developed.

WINDOWING QUERY PROBLEM

- Though the segments can intersect the query window in a variety of ways, in most of the cases intersecting segments have at least one endpoint inside the window.
- Remaining intersecting segments cross window boundary twice.
- First category of intersecting segments can be identified by using the range query data structure we have developed.
- The segments that intersect window boundary twice can be identified by the application of our modified interval tree data structure twice: for determining horizontal segments that intersect one of the two vertical edges of the window and for determining vertical segments that intersect one of the two horizontal edges of the window (by reversing the role of x- and y-coordinates this can be dealt with).

WINDOWING QUERY PROBLEM

- We now extend the windowing query problem to accommodate segments of arbitrary orientations.

Problem

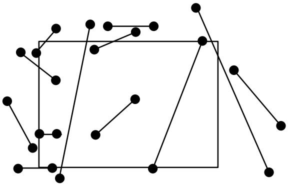
Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a query window $W := [x : x'] \times [y : y']$ can be reported efficiently.

WINDOWING QUERY PROBLEM

- We now extend the windowing query problem to accommodate segments of arbitrary orientations.

Problem

Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a query window $W := [x : x'] \times [y : y']$ can be reported efficiently.



- Though segments which have at least one endpoint inside the window can be determined as before, interval tree can no longer be used to find segments which intersect the window twice.

WINDOWING QUERY PROBLEM

- We introduce a data structure called segment tree which helps solve the problem. Specifically, we develop procedure to solve the following problem:

Problem

Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$ can be reported efficiently.

WINDOWING QUERY PROBLEM

- We introduce a data structure called segment tree which helps solve the problem. Specifically, we develop procedure to solve the following problem:

Problem

Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$ can be reported efficiently.

- It can be seen that, for solving the windowing query problem, it is sufficient to apply the procedure taking each of the four boundary edges of the window as the query segment.

SEGMENT TREE

- Let $I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be a set of n intervals on the real line.

SEGMENT TREE

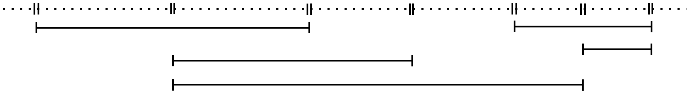
- Let $I := \{[x_1 : x_1'], [x_2 : x_2'], \ldots, [x_n : x_n']\}$ be a set of n intervals on the real line.
- Let $p_1, p_2, \ldots, p_m$ be the list of distinct interval endpoints, sorted from left to right, induced by the intervals in I. The regions in this partitioning are called elementary intervals. For the distinct endpoints p_i, $1 \leq i \leq m$, the elementary intervals from left to right are

$$(-\infty : p_1), [p_1 : p_1], (p_1 : p_2), [p_2 : p_2], \ldots, [p_m : p_m], (p_m : +\infty]$$

SEGMENT TREE

- Let $I := \{[x_1 : x'_1], [x_2 : x'_2], \ldots, [x_n : x'_n]\}$ be a set of n intervals on the real line.
- Let $p_1, p_2, \ldots, p_m$ be the list of distinct interval endpoints, sorted from left to right, induced by the intervals in I. The regions in this partitioning are called elementary intervals. For the distinct endpoints p_i, $1 \leq i \leq m$, the elementary intervals from left to right are

$$(-\infty : p_1), [p_1 : p_1], (p_1 : p_2), [p_2 : p_2], \ldots, [p_m : p_m], (p_m : +\infty]$$



SEGMENT TREE

- Build a binary search tree $\mathcal{T}$ whose leaves corresponds to these elementary intervals. We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.

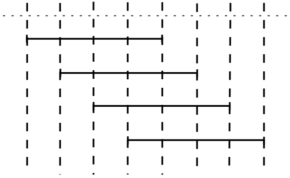
SEGMENT TREE

- Build a binary search tree $\mathcal{T}$ whose leaves corresponds to these elementary intervals. We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- If all the intervals in I containing $Int(\mu)$ would be stored at the leaf μ, then we would report the k intervals containing q_x in $O(\log n + k)$ time. So the query could be answered efficiently.

SEGMENT TREE

- Build a binary search tree $\mathcal{T}$ whose leaves corresponds to these elementary intervals. We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- If all the intervals in I containing $Int(\mu)$ would be stored at the leaf μ, then we would report the k intervals containing q_x in $O(\log n + k)$ time. So the query could be answered efficiently.

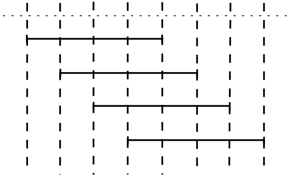
- However, observe that in this approach intervals that span a lot of elementary intervals are stored at many leaves increasing thereby the storage required. In the worst case, amount of storage may become quadratic.



SEGMENT TREE

- Build a binary search tree $\mathcal{T}$ whose leaves corresponds to these elementary intervals. We denote the elementary interval corresponding to a leaf μ by $Int(\mu)$.
- If all the intervals in I containing $Int(\mu)$ would be stored at the leaf μ, then we would report the k intervals containing q_x in $O(\log n + k)$ time. So the query could be answered efficiently.

- However, observe that in this approach intervals that span a lot of elementary intervals are stored at many leaves increasing thereby the storage required. In the worst case, amount of storage may become quadratic.
- For this reason, instead of storing at leaf level, intervals are stored as high as possible.



Segment tree

- Let us now describe segment tree formally.

SEGMENT TREE

- Let us now describe segment tree formally.
  - The skeleton of the segment tree is a balanced binary tree $\mathcal{T}$. The leaves of $\mathcal{T}$ corresponds to elementary intervals induced by the endpoints of the intervals in I in an ordered way: the leftmost leaf corresponds to leftmost elementary interval, and so on. The elementary interval corresponding ti leaf μ is denoted by $Int(\mu)$.

SEGMENT TREE

- Let us now describe segment tree formally.
  - The skeleton of the segment tree is a balanced binary tree $\mathcal{T}$. The leaves of $\mathcal{T}$ corresponds to elementary intervals induced by the endpoints of the intervals in I in an ordered way: the leftmost leaf corresponds to leftmost elementary interval, and so on. The elementary interval corresponding ti leaf μ is denoted by $Int(\mu)$.
  - Internal nodes of $\mathcal{T}$ correspond to intervals that are the union of the intervals of its two children, i.e., union of elementary intervals $Int(\mu)$ of the leaves in the subtree rooted by it.

SEGMENT TREE

- Let us now describe segment tree formally.
  - The skeleton of the segment tree is a balanced binary tree $\mathcal{T}$. The leaves of $\mathcal{T}$ corresponds to elementary intervals induced by the endpoints of the intervals in I in an ordered way: the leftmost leaf corresponds to leftmost elementary interval, and so on. The elementary interval corresponding ti leaf μ is denoted by $Int(\mu)$.
  - Internal nodes of $\mathcal{T}$ correspond to intervals that are the union of the intervals of its two children, i.e., union of elementary intervals $Int(\mu)$ of the leaves in the subtree rooted by it.
  - Each internal node or leaf v in $\mathcal{T}$ stores the interval $Int(v)$ and a set $I(v) \subseteq I$ of intervals (e.g., in a linked list). This canonical subset of node v contains the intervals $[x : x'] \in I$ such that $Int(v) \subseteq [x : x']$ and $Int(parent(v)) \not\subseteq [x : x']$.

CONSTRUCTION

- To construct a segment tree we proceed as follows.

Construction

- To construct a segment tree we proceed as follows.
- First we sort the endpoints of the intervals in I in $O(n \log n)$ time. This gives us the elementary intervals.

CONSTRUCTION

- To construct a segment tree we proceed as follows.
- First we sort the endpoints of the intervals in I in $O(n \log n)$ time. This gives us the elementary intervals.
- We then construct a balanced binary tree on the elementary intervals, and we determine for each node v of the tree the interval $Int(v)$ it represents. This can be done bottom-up in linear time.

CONSTRUCTION

- To construct a segment tree we proceed as follows.
- First we sort the endpoints of the intervals in I in $O(n \log n)$ time. This gives us the elementary intervals.
- We then construct a balanced binary tree on the elementary intervals, and we determine for each node v of the tree the interval $Int(v)$ it represents. This can be done bottom-up in linear time.
- It remains to compute the canonical subsets for the nodes. For this, we insert the intervals one by one into the segment tree. Code for insertion is as follows.

CONSTRUCTION

```
Algorithm InsertSegmentTree(v,[x:x′])
  If Int(v) is a subset of [x:x′]
    then store [x:x′] at v
  else if Int(lc(v)) and [x:x′] are not disjoint
    then InsertSegmentTree(lc(v),[x:x′])
    If Int(rc(v)) and [x:x′] are not disjoint
    then InsertSegmentTree(rc(v),[x:x′])
```

CONSTRUCTION

- How much time it takes to insert an interval?

Construction

- How much time it takes to insert an interval?
- At every node we visit we spend constant time (assuming that $I(v)$ is stored in a simple structure like a linked list).

CONSTRUCTION

- How much time it takes to insert an interval?
- At every node we visit we spend constant time (assuming that $I(v)$ is stored in a simple structure like a linked list).
- When we visit a node v, we either store $[x : x']$ at v, or $Int(v)$ contains an endpoint of $[x : x']$.

CONSTRUCTION

- How much time it takes to insert an interval?
- At every node we visit we spend constant time (assuming that $I(v)$ is stored in a simple structure like a linked list).
- When we visit a node v, we either store $[x : x']$ at v, or $Int(v)$ contains an endpoint of $[x : x']$.
- It can be shown that: (i) an interval is stored at most twice at each level, (ii) there is at most one node at every level whose corresponding interval contains x, and (iii) there is at most one node at every level whose corresponding interval contains x'.

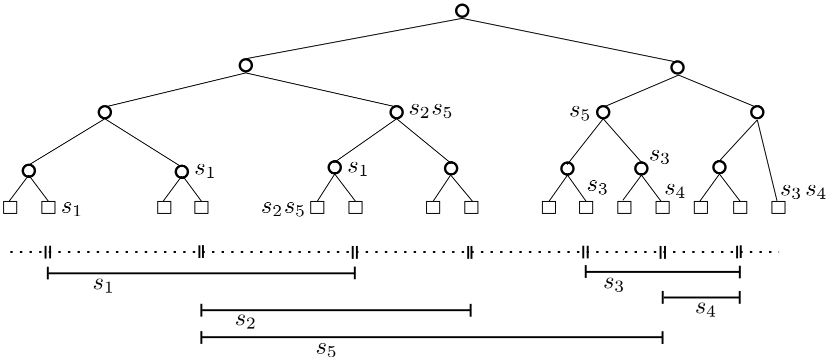
Construction

- How much time it takes to insert an interval?
- At every node we visit we spend constant time (assuming that $I(v)$ is stored in a simple structure like a linked list).
- When we visit a node v, we either store $[x : x']$ at v, or $Int(v)$ contains an endpoint of $[x : x']$.
- It can be shown that: (i) an interval is stored at most twice at each level, (ii) there is at most one node at every level whose corresponding interval contains x, and (iii) there is at most one node at every level whose corresponding interval contains x'.
- These facts together imply that we visit at most 4 nodes per level, i.e, time to insert a single interval is $O(\log n)$. Segment tree on n intervals can have at most $4n + 1$ leaves (Prove!), which in turn imply that it has depth $O(\log n)$.

CONSTRUCTION

- How much time it takes to insert an interval?
- At every node we visit we spend constant time (assuming that $I(v)$ is stored in a simple structure like a linked list).
- When we visit a node v, we either store $[x : x']$ at v, or $Int(v)$ contains an endpoint of $[x : x']$.
- It can be shown that: (i) an interval is stored at most twice at each level, (ii) there is at most one node at every level whose corresponding interval contains x, and (iii) there is at most one node at every level whose corresponding interval contains x'.
- These facts together imply that we visit at most 4 nodes per level, i.e, time to insert a single interval is $O(\log n)$. Segment tree on n intervals can have at most $4n + 1$ leaves (Prove!), which in turn imply that it has depth $O(\log n)$.
- Total time required for constructing a segment tree is $O(n \log n)$. Space requirement is also $O(n \log n)$ (Prove!).

EXAMPLE



Query

- Query algorithm is simple:

```
Algorithm QuerySegmentTree(v,qx)
  Report all intervals in I(v)
  If v is not a leaf
    then if qx belongs to Int(lc(v))
           then QuerySegmentTree(lc(v),qx)
         else  QuerySegmentTree(rc(v),qx)
```

QUERY

- Query algorithm is simple:

```
Algorithm QuerySegmentTree(v,qx)
  Report all intervals in I(v)
  If v is not a leaf
    then if qx belongs to Int(lc(v))
           then QuerySegmentTree(lc(v),qx)
         else  QuerySegmentTree(rc(v),qx)
```

- The query algorithm visits one node per level, so $O(\log n)$ nodes in total. At a node v we spend $O(1 + k_v)$ time, where k_v is the number of reported intervals. Hence query time is $O(\log n + k)$, where k is total number of reported intervals.

RESULT

Theorem

A segment tree for a set I of n intervals uses $O(n \log n)$ storge and can be constructed in $O(n \log n)$ time. Using the segment tree we can report all intervals that contain a query point in $O(\log n + k)$ time, where k is the number of reported segments.

WINDOWING QUERY PROBLEM

- We now go back to our windowing query problem. The problem we wanted to solve is:

Problem

Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$ *can be reported efficiently.*

Windowing query problem

- We now go back to our windowing query problem. The problem we wanted to solve is:

Problem

Let S be a set of n mutually disjoint line segments with arbitrary orientations in the plane. Preprocess the segments such that segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$ can be reported efficiently.

- We now show that segment tree data structure we have developed can be augmented to solve this problem.

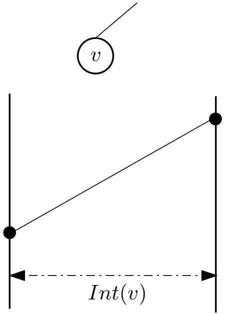
WINDOWING QUERY PROBLEM

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.

WINDOWING QUERY PROBLEM

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.

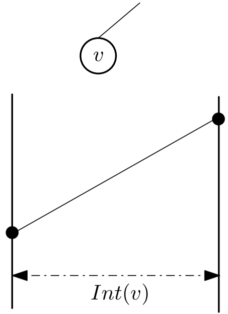
- We consider a node v in $\mathcal{T}$ to correspond to the vertical slab $Int(v) \times [-\infty : +\infty]$.



WINDOWING QUERY PROBLEM

- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.

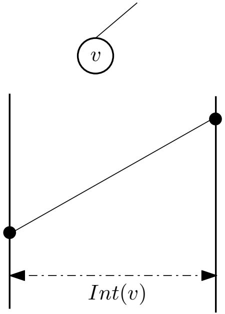
- We consider a node v in $\mathcal{T}$ to correspond to the vertical slab $Int(v) \times [-\infty : +\infty]$.
- A segment is in the canonical subset of v if it completely crosses the slab corresponding to v but not the slab corresponding to the parent of v. We denote this subset by $S(v)$.



WINDOWING QUERY PROBLEM

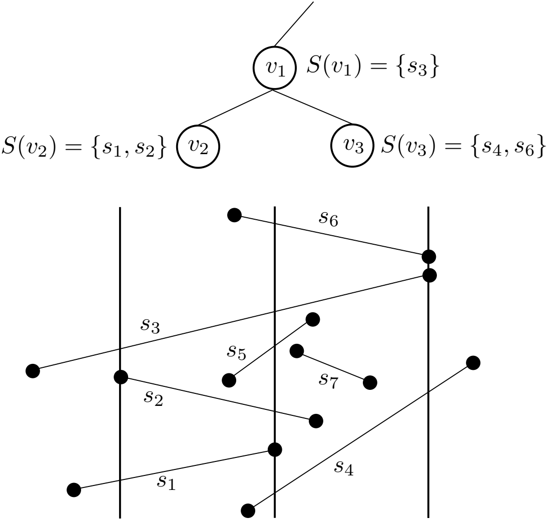
- Build a segment tree $\mathcal{T}$ on the x-intervals of the segments in S.

- We consider a node v in $\mathcal{T}$ to correspond to the vertical slab $Int(v) \times [-\infty : +\infty]$.
- A segment is in the canonical subset of v if it completely crosses the slab corresponding to v but not the slab corresponding to the parent of v. We denote this subset by $S(v)$.



- Let us see an example.

EXAMPLE

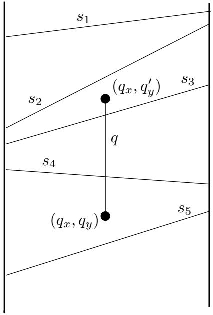


Windowing query problem

- When we search with q_x in $\mathcal{T}$ we find $O(\log n)$ canonical subsets on the search path that collectively contain all the segments whose x-interval contain q_x.

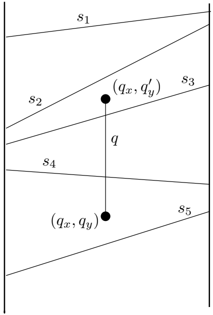
WINDOWING QUERY PROBLEM

- When we search with q_x in $\mathcal{T}$ we find $O(\log n)$ canonical subsets on the search path that collectively contain all the segments whose x-interval contain q_x.
- A segment s in such a canonical subset is intersected by q iff the lower endpoint of q is below s and the upper endpoint is above s.



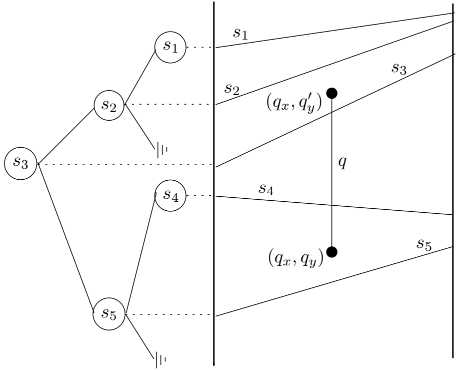
WINDOWING QUERY PROBLEM

- When we search with q_x in $\mathcal{T}$ we find $O(\log n)$ canonical subsets on the search path that collectively contain all the segments whose x-interval contain q_x.
- A segment s in such a canonical subset is intersected by q iff the lower endpoint of q is below s and the upper endpoint is above s.
- To identify proper segments we use the fact that the segments in the canonical subset $S(v)$ span the slab corresponding to v and that they do not intersect each other – i.e., they can be vertically ordered.



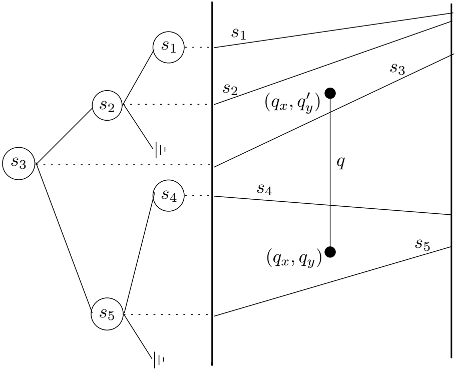
WINDOWING QUERY PROBLEM

- Hence we can store $S(v)$ in a search tree $\mathcal{T}(v)$ according to their vertical order.



WINDOWING QUERY PROBLEM

- Hence we can store $S(v)$ in a search tree $\mathcal{T}(v)$ according to their vertical order.
- By searching $\mathcal{T}(v)$ we can find the intersected segments in $O(\log n + k_v)$ time, where k_v is the number of intersected segments.



WINDOWING QUERY PROBLEM

- So the over all query algorithm is like this. We search with q_x in the segment tree in the usual way, and at every node v on the search path we search with upper and lower endpoint of q in $\mathcal{T}(v)$ to report the segments in $S(v)$. Since search in $\mathcal{T}(v)$ takes $O(\log n + k_v)$ time where k_v is the number of reported segments at v, total query time is $O(\log^2 n + k)$, where k is the total number of reported segments.

WINDOWING QUERY PROBLEM

- So the over all query algorithm is like this. We search with q_x in the segment tree in the usual way, and at every node v on the search path we search with upper and lower endpoint of q in $\mathcal{T}(v)$ to report the segments in $S(v)$. Since search in $\mathcal{T}(v)$ takes $O(\log n + k_v)$ time where k_v is the number of reported segments at v, total query time is $O(\log^2 n + k)$, where k is the total number of reported segments.
- Because the associated structure of any node v uses storage linear in size of $S(v)$, total amount of storage remain $O(n \log n)$.

WINDOWING QUERY PROBLEM

- So the over all query algorithm is like this. We search with q_x in the segment tree in the usual way, and at every node v on the search path we search with upper and lower endpoint of q in $\mathcal{T}(v)$ to report the segments in $S(v)$. Since search in $\mathcal{T}(v)$ takes $O(\log n + k_v)$ time where k_v is the number of reported segments at v, total query time is $O(\log^2 n + k)$, where k is the total number of reported segments.
- Because the associated structure of any node v uses storage linear in size of $S(v)$, total amount of storage remain $O(n \log n)$.
- The associated structure can be built in $O(n \log n)$ time, leading to a preprocessing time of $O(n \log^2 n)$.

Result

Theorem

Let S be a set of n disjoint segment of arbitrary orientations in the plane. The segments intersecting a vertical query segment can be reported in $O(\log^2 n + k)$ *time with a data structure that used* $O(n \log n)$ *storage, where k is the number of reported segments. The data structure can be built in* $O(n \log^2 n)$ *time.*

RESULT

Theorem

Let S be a set of n disjoint segment of arbitrary orientations in the plane. The segments intersecting an axis parallel rectangular query window can be reported in $O(\log^2 n + k)$ *time with a data structure that used* $O(n \log n)$ *storage, where k is the number of reported segments. The data structure can be built in* $O(n \log^2 n)$ *time.*

Outline

FUTURE DIRECTIONS

- We have confined ourselves in the plane only. Each of the problems can be generalized in space and higher dimensions.

FUTURE DIRECTIONS

- We have confined ourselves in the plane only. Each of the problems can be generalized in space and higher dimensions.
- The queries we have considered are called reporting queries. Another type of queries is often important where we want to count instead of report. Such queries are called counting queries.

FUTURE DIRECTIONS

- We have confined ourselves in the plane only. Each of the problems can be generalized in space and higher dimensions.
- The queries we have considered are called reporting queries. Another type of queries is often important where we want to count instead of report. Such queries are called counting queries.
- Both objects to be searched and query shape can vary. For example they can be triangles, circles, ellipses, tetrahedrons, simplexes, in higher space and higher dimensions.

REFERENCES

- Mark de Berg, Marc van Kreveld, Mark Overmars and Otfried Schwarzkof, *Computational Geometry: Algorithms and Applications*, Springer, 1997.
- Herbert Edelsbrunner, *Algorithms in Computational Geometry*, Springer, 1987.
- Joseph O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1998.
- Franco P. Preparata and Michael Ian Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
- http://en.wikipedia.org/wiki/Computational_geometry

Thank you!