Orthogonal Range Searching

Advanced Data Structures

Computation and Reasoning Laboratory
Graduate Course - Spring 2007

April 16, 2007

Outline

Outline

Querying a Database

- Do databases have something to do with geometry?

Querying a Database

- Do databases have something to do with geometry?
- Queries in a database can be interpreted geometrically.

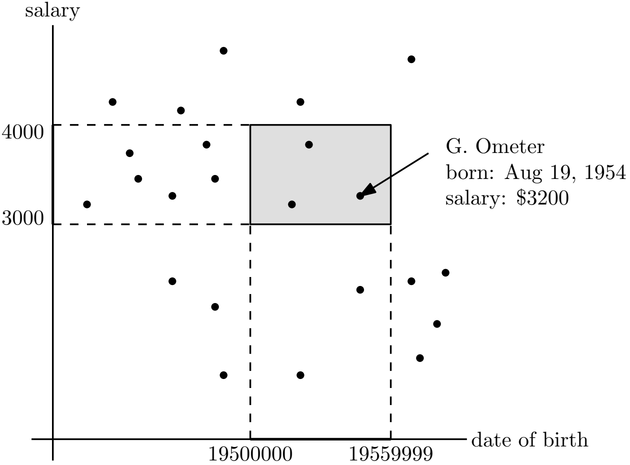
Querying a Database

- Do databases have something to do with geometry?
- Queries in a database can be interpreted geometrically.
- Transform records in a database into points in a multi-dimensional space.

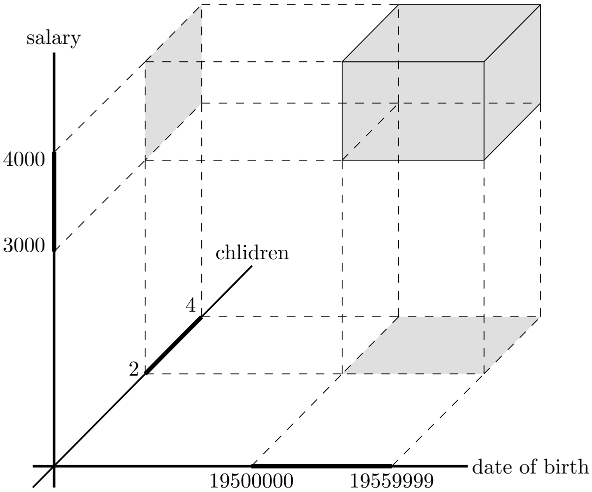
Querying a Database

- Do databases have something to do with geometry?
- Queries in a database can be interpreted geometrically.
- Transform records in a database into points in a multi-dimensional space.
- Transform queries about the records into queries on the set of points.

A typical query interpreted geometrically



A query in 3 dimensions



The geometric approach

- We are interested in answering queries on d fields of the records in our database.

The geometric approach

- We are interested in answering queries on d fields of the records in our database.
- Transform the records to points in d-dimensional space.

The geometric approach

- We are interested in answering queries on d fields of the records in our database.
- Transform the records to points in d-dimensional space.
- The transformed query asks for all points inside a d-dimensional axis-parallel box.

The geometric approach

- We are interested in answering queries on d fields of the records in our database.
- Transform the records to points in d-dimensional space.
- The transformed query asks for all points inside a d-dimensional axis-parallel box.
- Such a query is called “rectangular” or “orthogonal” range query.

The geometric approach

- We are interested in answering queries on d fields of the records in our database.
- Transform the records to points in d-dimensional space.
- The transformed query asks for all points inside a d-dimensional axis-parallel box.
- Such a query is called "rectangular" or "orthogonal" range query.
- We are going to present data structures for such queries.

Outline

The problem

Data

A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in 1-dimensional space (a set of real numbers).

Query

Which points lie inside a "1-dimensional query rectangle"? (i.e. an interval $[x : x']$)

Efficient Data Structures

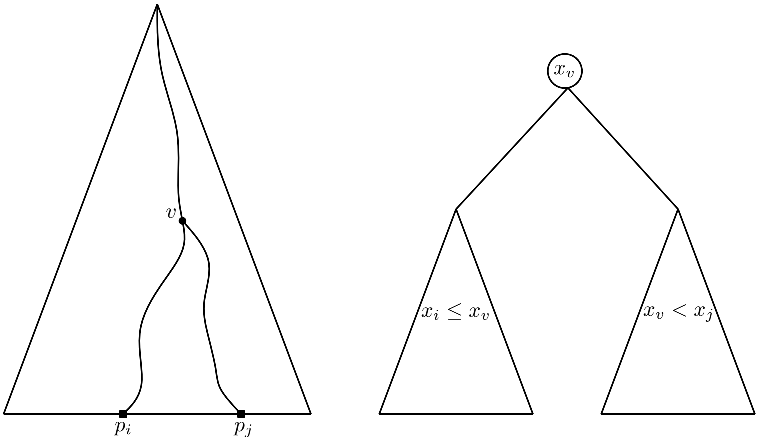
Arrays

- Solve the problem, but
- do not generalize,
- do not allow efficient updates.

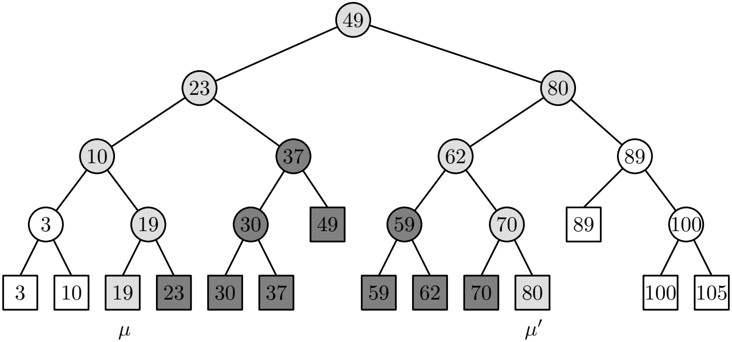
Balanced Binary Search Trees (BBST)

- The leaves of T store the points of P,
- internal nodes store splitting values that guide the search.

Balanced Binary Search Trees



A search with the interval $[18 : 77]$



A search with the interval $[x, x']$

- Search for x and x' in T. The search ends to leaves μ and μ'.
- Report all the points stored at leaves between μ and μ' plus, possibly, the points stored at μ and μ'.

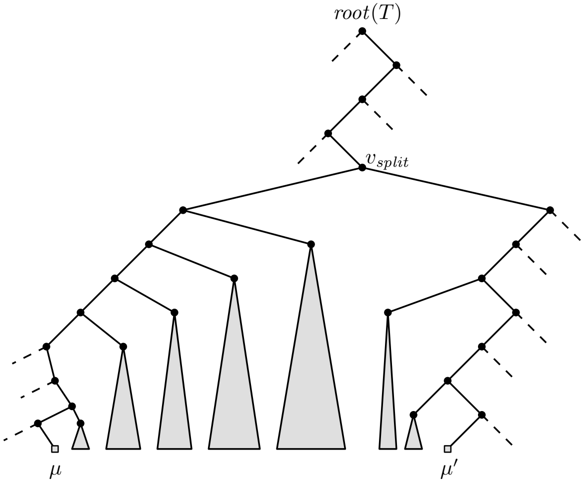
A search with the interval $[x, x']$

- Search for x and x' in T. The search ends to leaves μ and μ'.
- Report all the points stored at leaves between μ and μ' plus, possibly, the points stored at μ and μ'.

Remark

The leaves to be reported are the ones of subtrees that are rooted at nodes whose parents are on the search paths to μ and μ'.

The selected subtrees



Algorithms

FindSplitNode(T, x, x')

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

1D-RangeQuery($T, [x : x']$)

$v_{split} \leftarrow$ FindSplitNode(T, x, x')
if v_{split} is a leaf **then**
  check if $x_{v_{split}}$ must be reported
else {follow the path to x}
  $v \leftarrow lc(v_{split})$
  while v is not a leaf **do**
    if $x \leq x_v$ **then**
      ReportSubtree$(rc(v))$ {subtrees right of path}
      $v \leftarrow lc(v)$
    else
      $v \leftarrow rc(v)$
  check if x_v must be reported
  ...

Correctness and Performance

- Any reported point lies in the query range.
- Any point in the range is reported.

Correctness and Performance

- Any reported point lies in the query range.
- Any point in the range is reported.
- $O(n)$ storage.
- $O(n \log n)$ preprocessing.
- $\Theta(n)$ worst case case query cost.

Correctness and Performance

- Any reported point lies in the query range.
- Any point in the range is reported.
- $O(n)$ storage.
- $O(n \log n)$ preprocessing.
- $\Theta(n)$ worst case case query cost.
- $O(k + \log n)$ output sensitive query cost: $O(k)$ to report the points plus $O(\log n)$ to follow the paths to x, x'.

Outline

The problem

Data

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

Query

Which points lie inside a query rectangle $[x : x'] \times [y : y']$?

The problem

Data

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

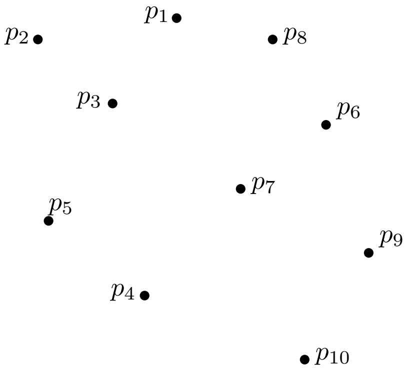
Query

Which points lie inside a query rectangle $[x : x'] \times [y : y']$?

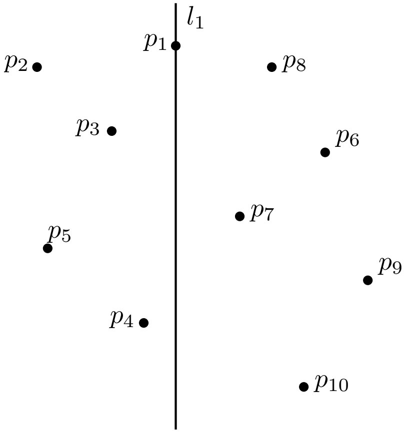
Remark

A point $p = (p_x, p_y)$ lies inside this rectangle iff $p_x \in [x, x']$ and $p_y \in [y, y']$.

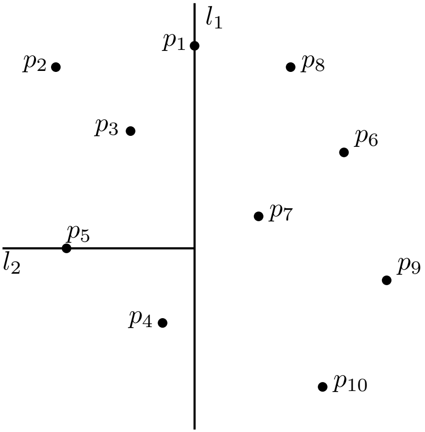
The way the plane is subdivided



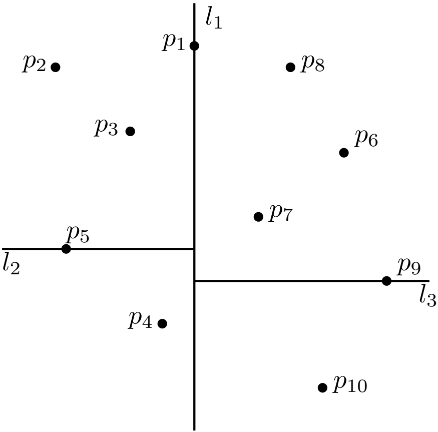
The way the plane is subdivided



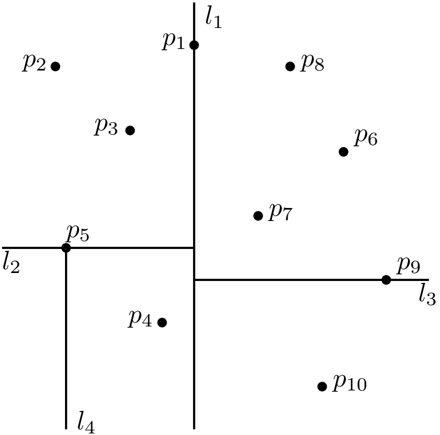
The way the plane is subdivided



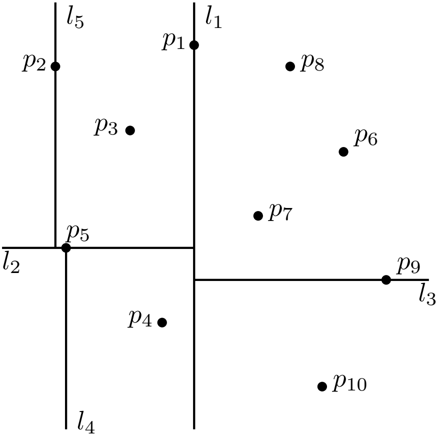
The way the plane is subdivided



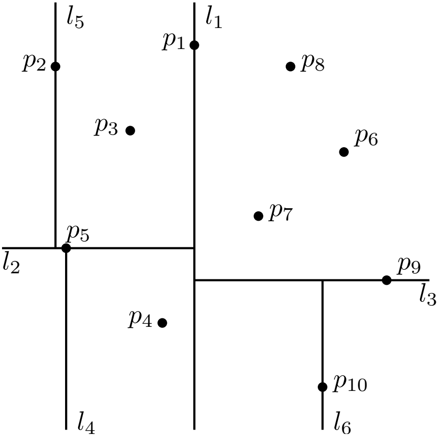
The way the plane is subdivided



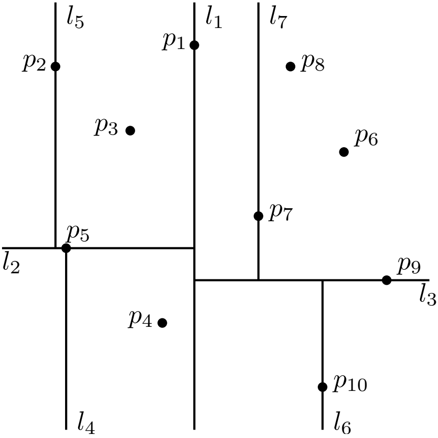
The way the plane is subdivided



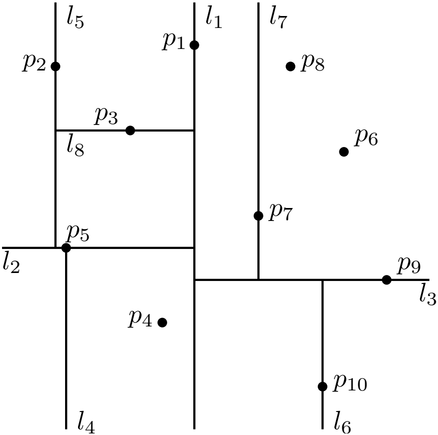
The way the plane is subdivided



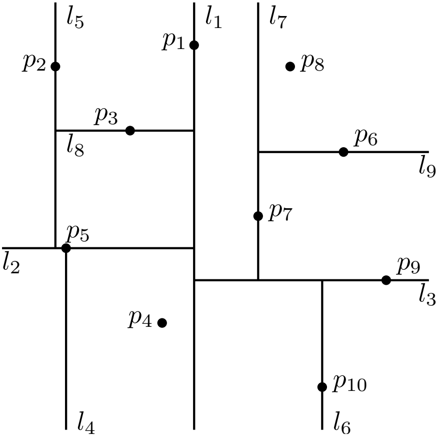
The way the plane is subdivided



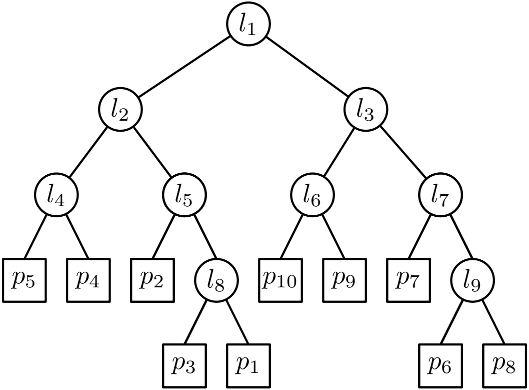
The way the plane is subdivided



The way the plane is subdivided



The corresponding binary tree



Algorithm

BuildKdTree(P, *depth*)

```
if P contains only one point then
  return a leaf storing this point
else
  if depth is even then
    split P with vertical l through median x-coord of points in P
    P_1 ← the set of points left of l or on l
    P_2 ← the set of points right of l
  else
    split P with horizontal l through median y-coord of points in P
    P_1 ← the set of points below l or on l
    P_2 ← the set of points above l
  v_left ← BuidKdTree(P_1, depth + 1)
  v_right ← BuidKdTree(P_2, depth + 1)
  create a node v storing l
  lc(v) → v_left
  rc(v) → v_right
  return v
```

Algorithm

BuildKdTree(P, *depth*)

if P contains only one point **then**
  return a leaf storing this point
else
  if *depth* is even **then**
    split P with vertical l through median x-coord of points in P
    $P_1 \longleftarrow$ the set of points left of l or on l
    $P_2 \longleftarrow$ the set of points right of l
  else
    split P with horizontal l through median y-coord of points in P
    $P_1 \longleftarrow$ the set of points below l or on l
    $P_2 \longleftarrow$ the set of points above l
  $v_{left} \longleftarrow \text{BuidKdTree}(P_1, depth + 1)$
  $v_{right} \longleftarrow \text{BuidKdTree}(P_2, depth + 1)$
  create a node v storing l
  $lc(v) \longrightarrow v_{left}$
  $rc(v) \longrightarrow v_{right}$
  return v

Remarks

- We should split at the $\frac{n}{2}$-th smallest coordinate.
- Preprocessing involves sorting both on x- and y-coordinate.

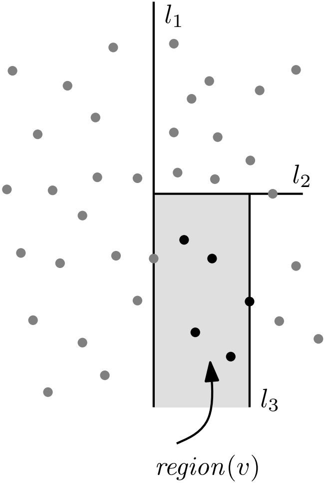
Building time and storage

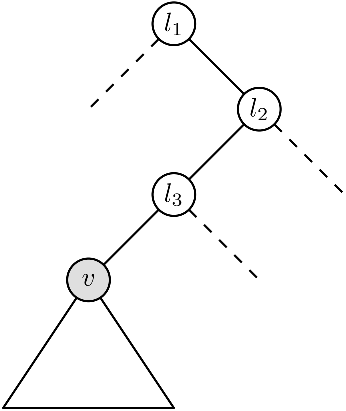
- The building time satisfies the recurrence:

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

- $T(n) = O(n \log n)$ which subsums the preprocessing time.
- $O(n)$ storage: each leaf stores a distinct point of P.

Nodes in a kd-tree and regions in the plane

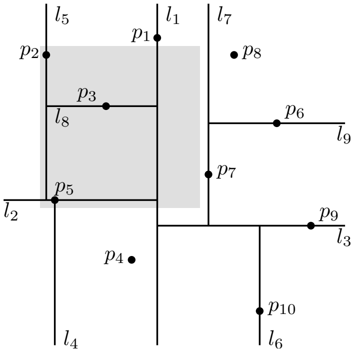


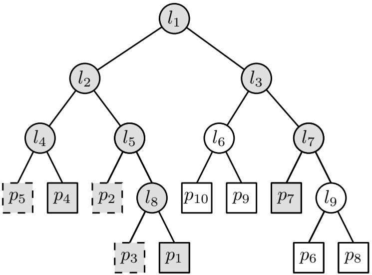


Regions and the query algorithm

- Internal nodes of a Kd-tree correspond to rectangular regions of the plane which can be unbounded on one or more sides.
- Such regions are bounded by splitting lines stored at ancestors of the internal nodes.
- $\mathit{region}(\mathit{root}(T))$ is the whole plane.
- A point is stored in a subtree rooted at a node v iff it lies in $\mathit{region}(v)$.
- We search the subtree rooted at v only if the query rectangle intersects $\mathit{region}(v)$.

A query on a kd-tree





Algorithm

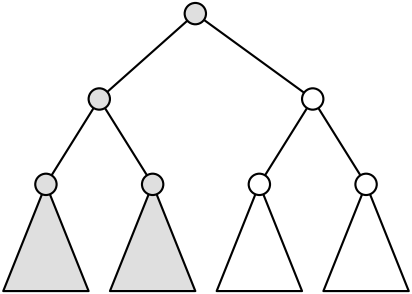
SearchKdTree(v, R)

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

- Works for any query range R (e.g. triangles).
- $O(k)$, in order to report k points.
- How many other nodes are visited?
- For these nodes v, the query range intersects $region(v)$.

Query time analysis

- Any vertical line intersects $region(lc(root(T)))$ or $region(rc(root(T)))$ but not both.
- If a vertical line intersects $region(lc(root(T)))$ it will always intersect the regions corresponding to both children of $lc(root(T))$.



Query time analysis

- The number of intersected regions in a kd-tree storing n points, satisfies the recurrence:

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

- $Q(n) = O(\sqrt{n})$. The total query time is $O(\sqrt{n} + k)$
- The analysis is rather pessimistic: In many practical situations the query range is small and will intersect much fewer regions.

Kd-trees in higher-dimensional spaces

- Kd-trees can be also used for point sets in 3- or higher-dimensional spaces.
- Assume the dimension d to be a constant:
- $O(d \cdot n)$ storage.
- $O(d \cdot n \log n)$ construction time.
- $O(n^{1-\frac{1}{d}} + k)$ query time.

Outline

The Range Tree approach

- 2D range queries are two 1D range queries one on $x-$ and one on $y-$coordinate.

The Range Tree approach

- 2D range queries are two 1D range queries one on $x-$ and one on $y-$coordinate.
- Find first the points whose $x-$coordinate lies in $[x : x']$ and worry about the $y-$coordinate later.

The Range Tree approach

- 2D range queries are two 1D range queries one on $x-$ and one on $y-$coordinate.
- Find first the points whose $x-$coordinate lies in $[x : x']$ and worry about the $y-$coordinate later.
- During the 1D range query a logarithmic number of subtrees is selected.

The Range Tree approach

- 2D range queries are two 1D range queries one on $x-$ and one on $y-$coordinate.
- Find first the points whose $x-$coordinate lies in $[x : x']$ and worry about the $y-$coordinate later.
- During the 1D range query a logarithmic number of subtrees is selected.
- The leaves of these subtrees contain exactly the points whose $x-$coordinate lies in $[x : x']$.

The Range Tree approach

Canonical Subset of a node v

The subset of points $P(v)$ of P stored in the leaves of the subtree rooted at v.

$$P(root(\mathcal{T})) = P$$

The Range Tree approach

Canonical Subset of a node v

The subset of points $P(v)$ of P stored in the leaves of the subtree rooted at v.

$$P(root(\mathcal{T})) = P$$

- The subset of points whose $x-$coordinate lies in the query range is a disjoint union of $O(\log n)$ canonical subsets.

The Range Tree approach

Canonical Subset of a node v

The subset of points $P(v)$ of P stored in the leaves of the subtree rooted at v.

$$P(root(\mathcal{T})) = P$$

- The subset of points whose $x-$coordinate lies in the query range is a disjoint union of $O(\log n)$ canonical subsets.
- We are not interested in all the points in such subsets.

The Range Tree approach

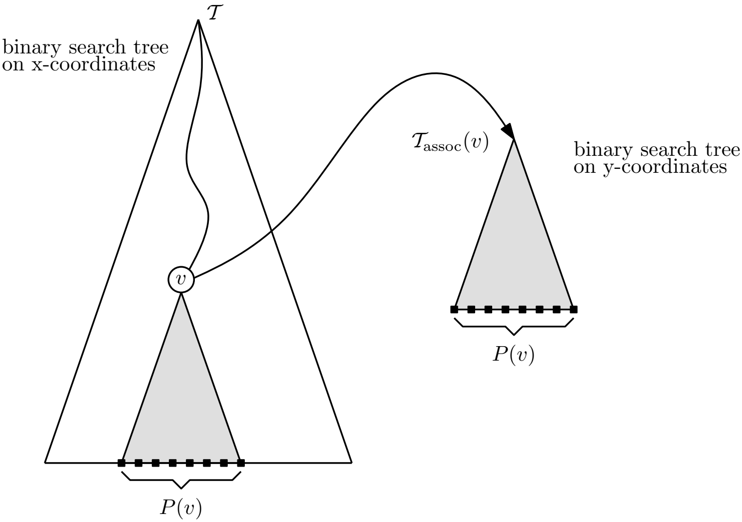
Canonical Subset of a node v

The subset of points $P(v)$ of P stored in the leaves of the subtree rooted at v.

$$P(root(\mathcal{T})) = P$$

- The subset of points whose $x-$coordinate lies in the query range is a disjoint union of $O(\log n)$ canonical subsets.
- We are not interested in all the points in such subsets.
- Report the ones whose $y-$coordinate lies in $[y : y']$: This is another 1D query.

A 2-dimensional Range Tree



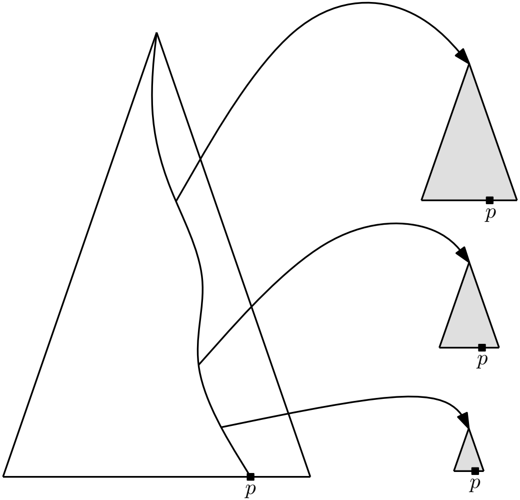
Algorithm

Build2DRangeTree(P)

Build a BST $\mathcal{T}_{assoc}$ on the set P_y
Store the points of P at the leaves of $\mathcal{T}_{assoc}$
if P contains only one point **then**
  Create a leaf v storing this point
  Associate $\mathcal{T}_{assoc}$ with v
else
  Split P into P_{left} and P_{right} through x_{mid}
  $v_{left} \leftarrow$ Build2DRangeTree(P_{left})
  $v_{right} \leftarrow$ Build2DRangeTree(P_{right})
  create a node v storing x_{mid}
  $lc(v) \leftarrow v_{left}$
  $rc(v) \leftarrow v_{right}$
  Associate $\mathcal{T}_{assoc}$ with v
return v

- Preprocessing involves maintaining two lists of points.
- One sorted on $x-$coordinate and one sorted on $y-$coordinate.
- The time spend at a node in the main tree is linear in the size of its canonical subset.

Range Tree storage



Storage and costruction time

- Each point is stored only once at a given depth.
- The total depth is $O(\log n)$: the amount of storage is $O(n \log n)$.
- Since the time spend at a node in the main tree is linear in the size of its canonical subset the total construction time is the same as the amount of storage.
- Presorting is $O(n \log n)$.
- Total construction time is $O(n \log n)$.

Algorithm

2DRangeQuery($\mathcal{T}$, $[x : x'] \times [y : y']$)

$v_{split} \leftarrow$ FindSplitNode($\mathcal{T}$, x, x')
if v_{split} is a leaf **then**
  check if $x_{v_{split}}$ must be reported
else {follow the path to x}
  $v \leftarrow lc(v_{split})$
  while v is not a leaf **do**
    if $x \leq x_v$ **then**
      1DRangeQuery($\mathcal{T}_{assoc}(rc(v))$, $[y : y']$)
      $v \leftarrow lc(v)$
    else
      $v \leftarrow rc(v)$
  check if x_v must be reported
  . . .

Query time analysis

- The time spend to report the points whose $y-$coordinate lie in the range $[y : y']$ is $O(\log n + k_v)$ where k_v is the number of points reported in this call.

Query time analysis

- The time spend to report the points whose $y-$coordinate lie in the range $[y : y']$ is $O(\log n + k_v)$ where k_v is the number of points reported in this call.
- $Q(n) = \sum_v O((\log n) + k_v)$ where the summation is over all nodes visited.

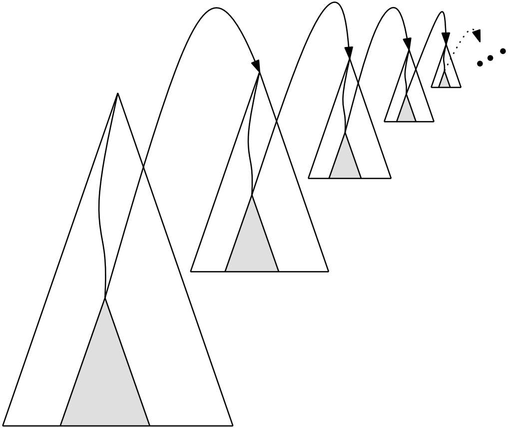
Query time analysis

- The time spend to report the points whose $y-$coordinate lie in the range $[y : y']$ is $O(\log n + k_v)$ where k_v is the number of points reported in this call.
- $Q(n) = \sum_v O((\log n) + k_v)$ where the summation is over all nodes visited.
- $\sum_v k_v = k$, the total number of reported points. The search paths of x and x' have length $O(\log n)$: $\sum_v O(\log n) = O(\log^2 n)$.

Query time analysis

- The time spend to report the points whose $y-$coordinate lie in the range $[y : y']$ is $O(\log n + k_v)$ where k_v is the number of points reported in this call.
- $Q(n) = \sum_v O((\log n) + k_v)$ where the summation is over all nodes visited.
- $\sum_v k_v = k$, the total number of reported points. The search paths of x and x' have length $O(\log n)$: $\sum_v O(\log n) = O(\log^2 n)$.
- $Q(n) = O(\log^2 n + k)$.

Higher-Dimensional Range Trees



Higher-Dimensional Range Trees

- P is a set on n points in $d-$dimensional space ($d \geq 2$):
- $O(n \log^{d-1} n)$ storage,
- $O(n \log^{d-1} n)$ construction time,
- $O(\log^d n + k)$ query time.

Outline

The idea of Fractional Cascading

- S_1, S_2 are two set of objects with real number keys.

The idea of Fractional Cascading

- S_1, S_2 are two set of objects with real number keys.
- The problem is to report all objects in S_1 and S_2 whose keys lie in $[y : y']$.

The idea of Fractional Cascading

- S_1, S_2 are two set of objects with real number keys.
- The problem is to report all objects in S_1 and S_2 whose keys lie in $[y : y']$.
- The keys are in sorted order in arrays A_1 and A_2.

The idea of Fractional Cascading

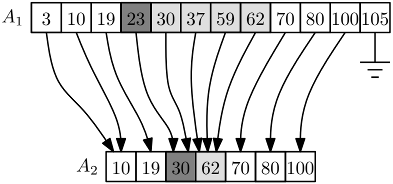
- S_1, S_2 are two set of objects with real number keys.
- The problem is to report all objects in S_1 and S_2 whose keys lie in $[y : y']$.
- The keys are in sorted order in arrays A_1 and A_2.
- Solution: two binary searches in A_1 and A_2.

The idea of Fractional Cascading

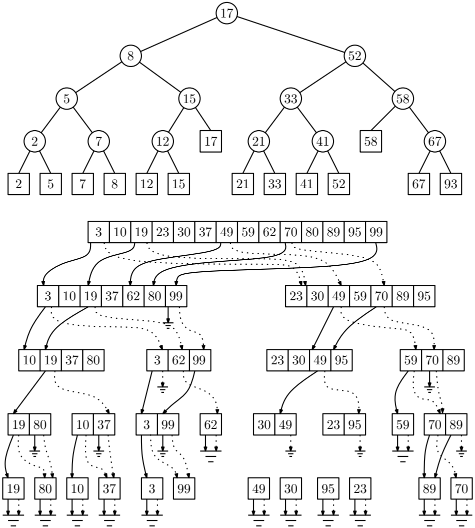
- S_1, S_2 are two set of objects with real number keys.
- The problem is to report all objects in S_1 and S_2 whose keys lie in $[y : y']$.
- The keys are in sorted order in arrays A_1 and A_2.
- Solution: two binary searches in A_1 and A_2.
- If $S_2 \subseteq S_1$ we can avoid the binary search in A_2.

The idea of Fractional Cascading

- S_1, S_2 are two set of objects with real number keys.
- The problem is to report all objects in S_1 and S_2 whose keys lie in $[y : y']$.
- The keys are in sorted order in arrays A_1 and A_2.
- Solution: two binary searches in A_1 and A_2.
- If $S_2 \subseteq S_1$ we can avoid the binary search in A_2.



A Layered Range Tree



A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.

A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.
- At v_{split} find the entry in $A(v_{split})$ whose $y-$coordinate is larger than or equal to y in $O(\log n)$ time.

A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.
- At v_{split} find the entry in $A(v_{split})$ whose $y-$coordinate is larger than or equal to y in $O(\log n)$ time.
- For all $O(\log n)$ nodes on the paths to x and x' maintain pointers to the entries in A whose $y-$coordinate is larger than or equal to y in $O(1)$ time.

A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.
- At v_{split} find the entry in $A(v_{split})$ whose $y-$coordinate is larger than or equal to y in $O(\log n)$ time.
- For all $O(\log n)$ nodes on the paths to x and x' maintain pointers to the entries in A whose $y-$coordinate is larger than or equal to y in $O(1)$ time.
- Report the points of $A(v)$ in $O(1 + k_v)$ time, k_v is the number of reported points at node v.

A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.
- At v_{split} find the entry in $A(v_{split})$ whose $y-$coordinate is larger than or equal to y in $O(\log n)$ time.
- For all $O(\log n)$ nodes on the paths to x and x' maintain pointers to the entries in A whose $y-$coordinate is larger than or equal to y in $O(1)$ time.
- Report the points of $A(v)$ in $O(1 + k_v)$ time, k_v is the number of reported points at node v.
- Total query time becomes $O(\log n + k)$.

A query in a Layered Range Tree

- The query range is $[x : x'] \times [y : y']$.
- At v_{split} find the entry in $A(v_{split})$ whose $y-$coordinate is larger than or equal to y in $O(\log n)$ time.
- For all $O(\log n)$ nodes on the paths to x and x' maintain pointers to the entries in A whose $y-$coordinate is larger than or equal to y in $O(1)$ time.
- Report the points of $A(v)$ in $O(1 + k_v)$ time, k_v is the number of reported points at node v.
- Total query time becomes $O(\log n + k)$.
- Fractional cascading also imporves the query time of higher-dimensional range trees by a logarithmic factor.