AALBORG UNIVERSITY

Advanced Algorithms

Lecture 8

Computational Geometry Algorithms:

Range Searching

Bin Yang

byang@cs.aau.dk

ILO of Lecture 8

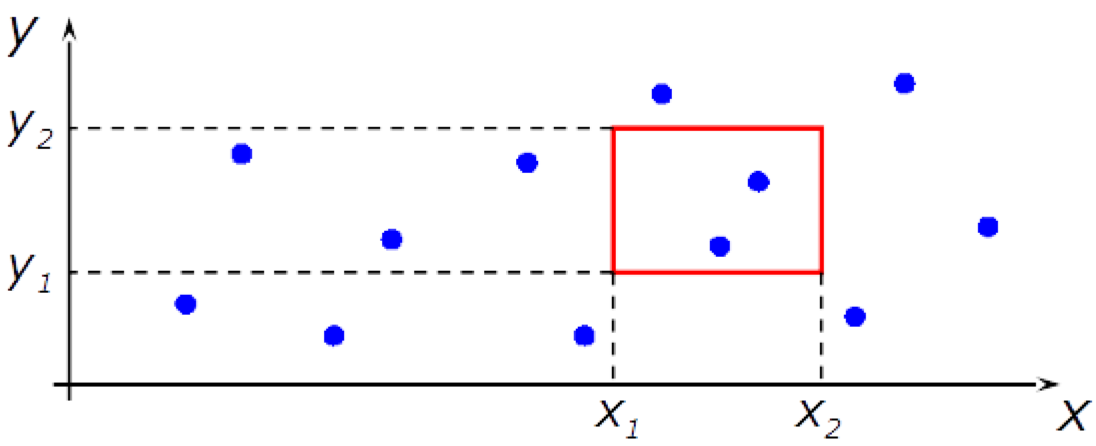
- Computational Geometry: range searching
  - to understand and to be able to analyze the balanced binary search tree based 1D range searching algorithm;
  - to understand and to be able to analyze the kd-trees and the range trees;
  - to understand how data structures can be used to trade the space used for the running time of queries.

Agenda

- Range Searching
- 1D range searching
- 2D range searching
- KD-trees
- Range trees

Range searching

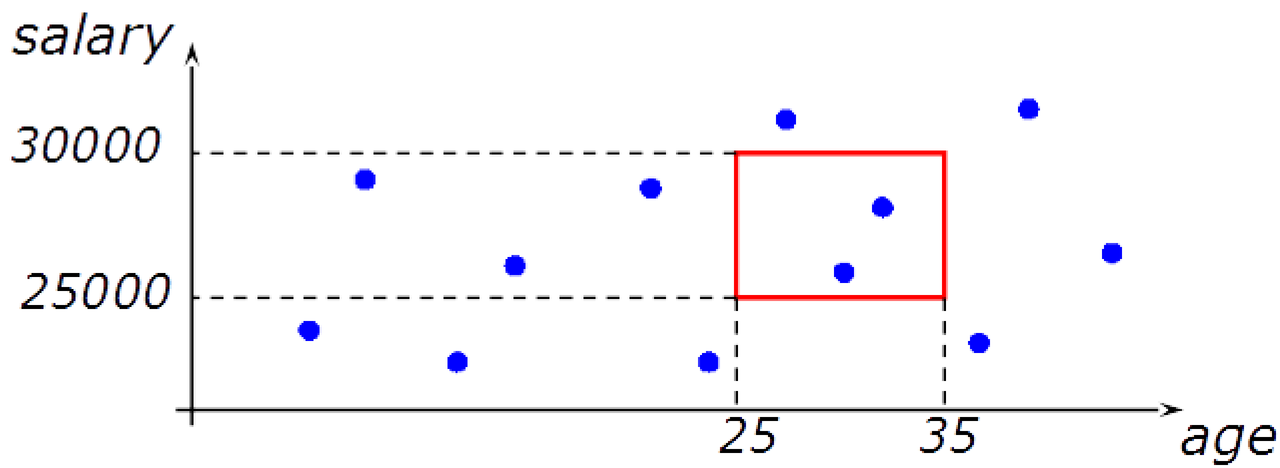
- How to efficiently find points that are inside of a rectangle?
  - E.g., road intersections in a region, cars in a parking lot.
- Orthogonal range (Rectangular range) search
  - Given an orthogonal range $([x_1, x_2], [y_1, y_2])$
  - Find all points (x, y) such that $x_1<x<x_2$ and $y_1<y<y_2$



- The range can be in an *n*-dimensional space.
  - $([l_1, u_1], [l_2, u_2], [l_3, u_3], \ldots, [l_n, u_n])$

When to use range searching

- Geographic information systems
  - Report all the cars in AAU campus.
- Often useful in a multi-attribute database query
  - Consider a database for personnel administration.
    - Name, address, age, salary of each employee.
    - Report all employees whose ages are between 25 to 35 and who earn between 25.000 dkk to 30.000 dkk per month.



Agenda

- Range Searching
- 1D range searching
- 2D range searching
- KD-trees
- Range trees

1D Range Searching

- How do we conduct a 1D range search $[x_1, x_2]$?
- Naive method:
  - Check every point to see if it is in range $[x_1, x_2]$.
  - $\Theta(n)$

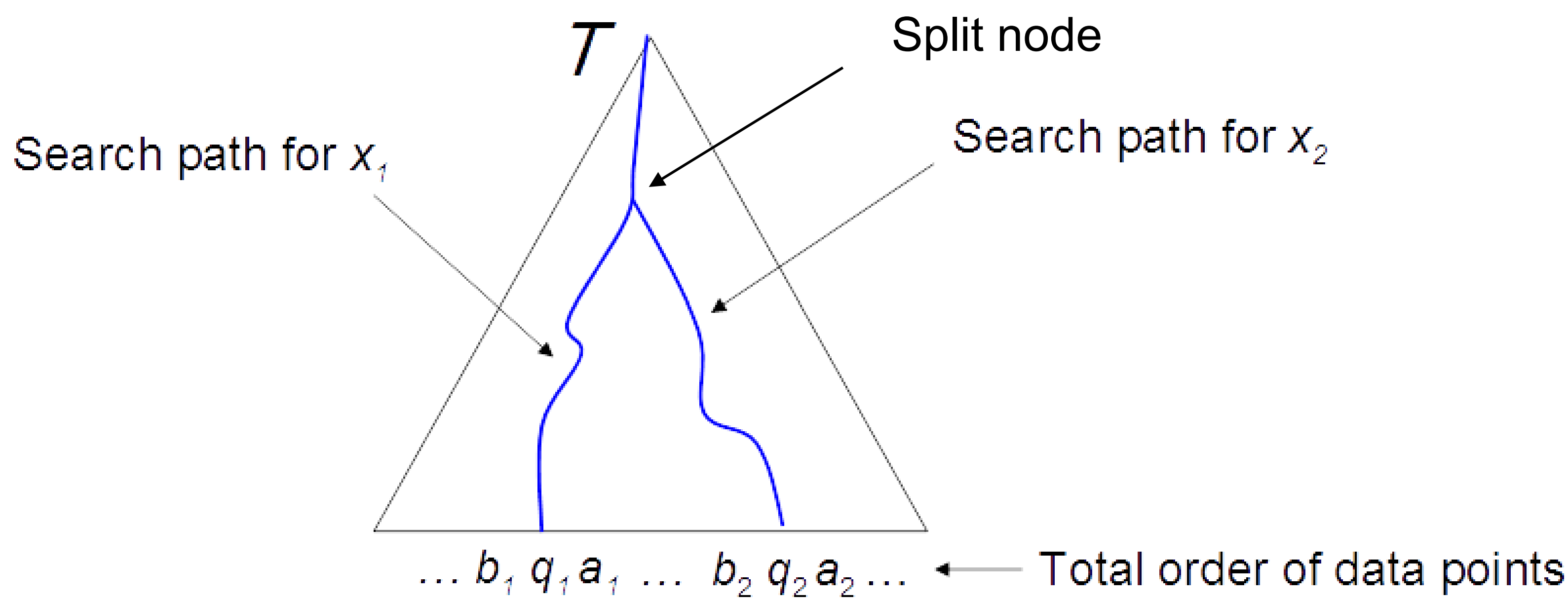
- What if we use a balanced binary search tree?

Rules of the game

- We preprocess the data into a data structure
- Then, we perform range searches on the data structure
- Analyses:
  - Run time for building the data structure
  - Run time for processing range searches
  - Space that the data structure takes

1D Range Searching

- Balanced binary search tree where all data points are stored in the leaves
  - Internal nodes store copies of points.
    - The left sub-tree ≤ internal node < the right sub-tree
- Where do we find the answer to a query?

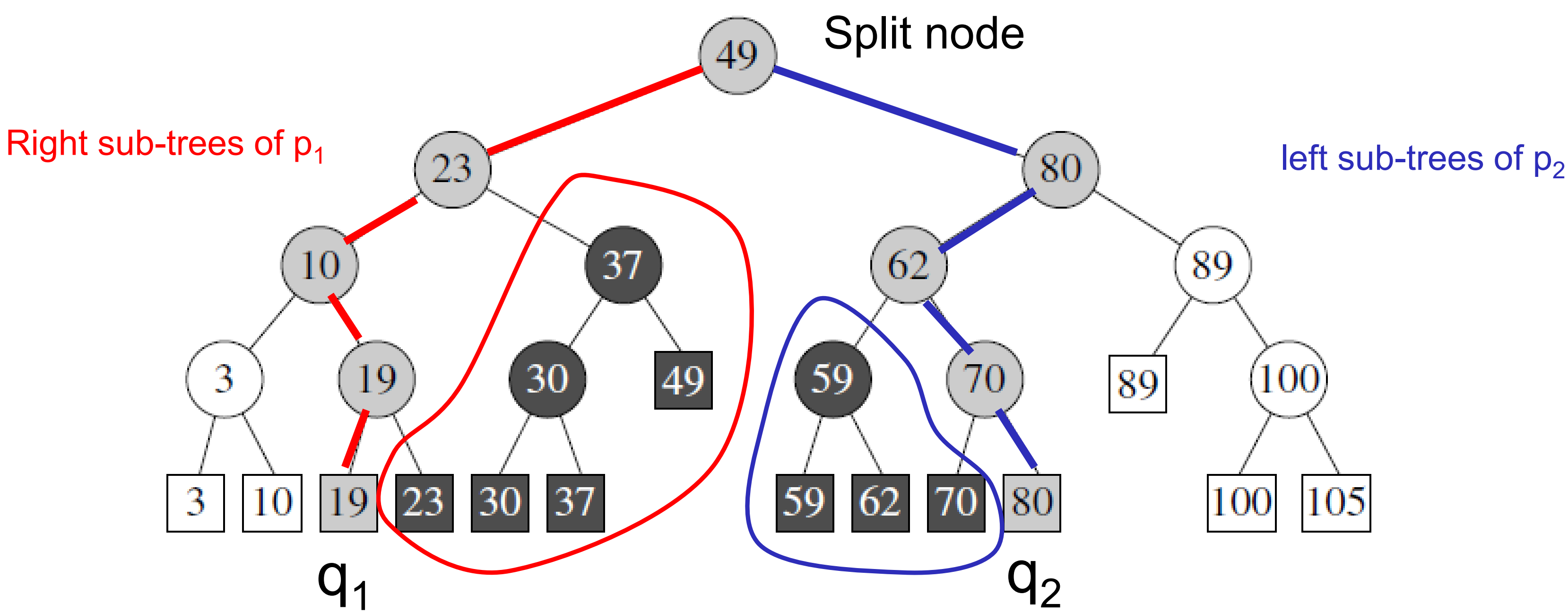


1D Range Searching

- Sketch of the algorithm:
  - Find *the split node* where the paths to x_1 and x_2 separate.
  - Continue path p_1 to search for x_1 and identify the leave q_1.
  - Continue path p_2 to search for x_2 and identify the leave q_2.
  - When leaves q_1 and q_2 are reached, check if they belong to the range.
  - Report all leaves that are in the sub-trees in between search paths p_1 and p_2.
    - Right sub-trees of p_1 and left sub-trees of p_2.

Example

- Searching for [18, 77]
  - Find the split node: 49, because 18 ≤ 49< 77
  - Search for 18 on the left subtree of 49
  - Search for 77 on the right subtree of 49
  - Grey nodes are on the search paths for 18 and 77
    - Red path p_1 for 18, blue path p_2 for 77.
    - q_1=19, q_2=80
  - Black nodes are the sub-trees in between the search paths.

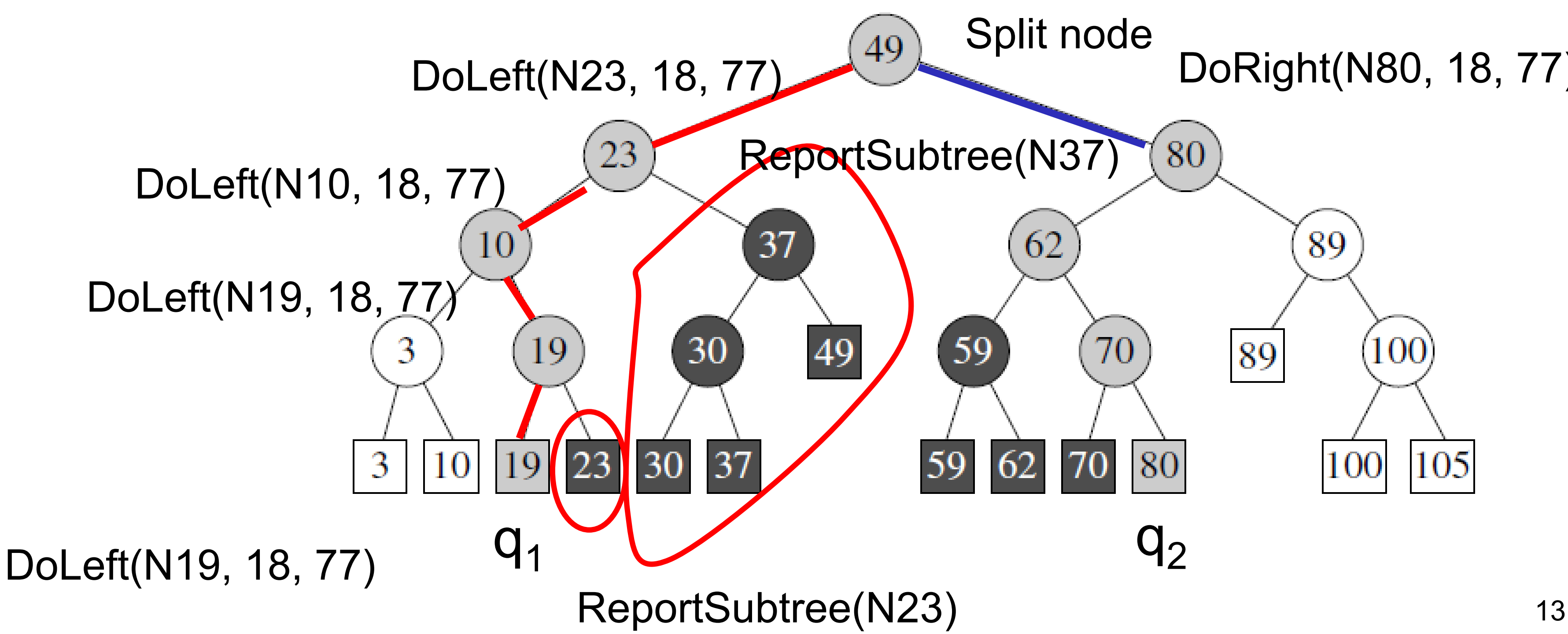


Pseudo code

```
1DRangeSearch(T, x1, x2)
v ← FindSplit(T, x1, x2)
if v is a leaf then
    if x1 ≤ v.key ≤ x2 then return v
else return DoLeft(v.leftChild, x1, x2) ∪
            DoRight(v.rightChild, x1, x2)
DoLeft(v, x1, x2)
if v is a leaf then
    if x1 ≤ v.key ≤ x2 then return v
else
    if x1 ≤ v.key then return ReportSubtree(v.rightChild) ∪
                              DoLeft(v.leftChild, x1, x2)
    else return DoLeft(v.rightChild, x1, x2)
DoRight(v, x1, x2)
if v is a leaf then
    if x1 ≤ v.key ≤ x2 then return v
else
    if x2 > v.key then return ReportSubtree(v.leftChild) ∪
                              DoRight(v.rightChild, x1, x2)
else return DoRight(v.rightChild, x1, x2)
```

Example

```
DoLeft(v, x1, x2)
if v is a leaf then
    if x1 ≤ v.key ≤ x2 then return v
else
    if x1 ≤ v.key then return ReportSubtree(v.rightChild) ∪
                              DoLeft(v.leftChild, x1, x2)
    else return DoLeft(v.rightChild, x1, x2)
```



Correctness

- The reported points must lie in the query range $[x_1, x_2]$.
  - If *p* is stored at the leaf where the path to x_1 or to x_2 ends, then p is tested explicitly for inclusion in the query range.

```
01 if v is a leaf then
02     if x₁ ≤ v.key ≤ x₂ then return v
```

  - If p is reported in the call of *ReportSubtree*(v.rightChild) in doLeft()

```
if x₁ ≤ v.key then return ReportSubtree(v.rightChild) ∪
                          DoLeft(v.leftChild, x₁, x₂)
```

    - $x_1 \le$ v.key.
    - p is in v's right sub-tree, so we have v.key < p.
    - Since it is doLeft, it must be in v_{split}'s left tree, thus p $\le v_{split}$.key.
    - x_2 is in the right sub-tree of v_{split}, thus v_{split}.key<x_2.
    - $x_1 \le$ v.key < p $\le v_{split}$.key<x_2.
  - If p is reported in the call of *ReportSubtree*(v.leftChild) in doRight()
    - $x_1 \le v_{split}$.key < p $\le$ v.key<x_2.
- All points that lie in the query range have been reported.

Analysis

- Building a balanced BST
  - O(nlgn) run time.
  - O(n) space.

- What is the worst case running time of a query?
  - It is *output-sensitive*:
    - Two traversals down the tree, each takes lgn.
    - Report the points in the sub-trees between two searching paths: O(k), where k is the number of reported data points.
  - In total: $\Theta(\lg n + k)$
  - In the worst case, all n points should be reported, so $\Theta(n)$
    - It is no better than the naive method without using an BST – checking each point to see if it is in the range.

Agenda

- Range Searching
- 1D range searching
- 2D range searching
- KD-trees
- Range trees

2D range searching

- How can we solve a 2D range search?
- A 2D range query is a conjunction of two 1D range queries.
  - $x_1 \leq x \leq x_2$ **and** $y_1 \leq y \leq y_2$
- Naïve idea:
  - have two BSTs on *x*-coordinate and on *y*-coordinate, respectively.
  - Ask two 1D range searches.
  - Return the intersection of their results.

- Mini-quiz: What is the worst-case running time (and when does it happen)? Is it output-sensitive?

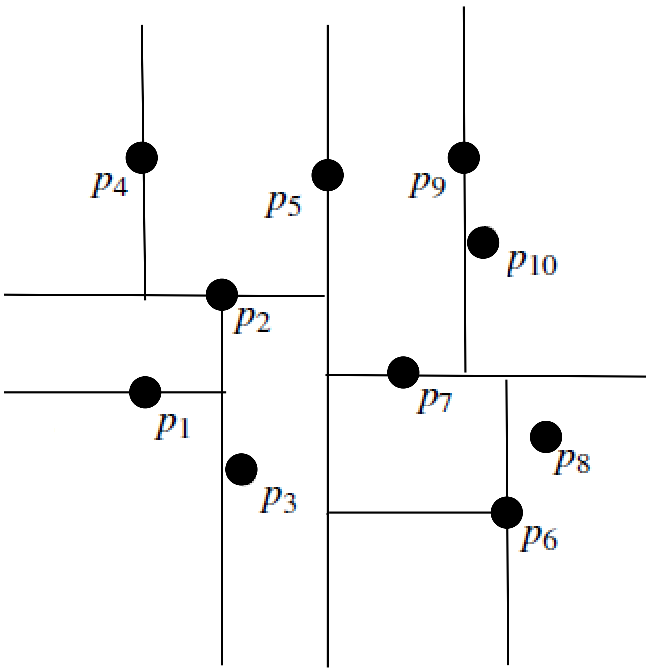
Agenda

- Range Searching
- 1D range searching
- 2D range searching
- KD-trees
- Range trees

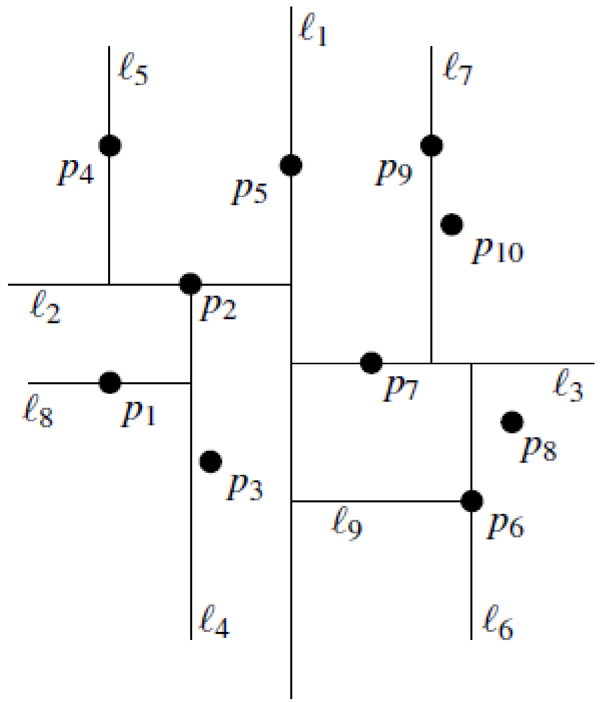
KD-tree

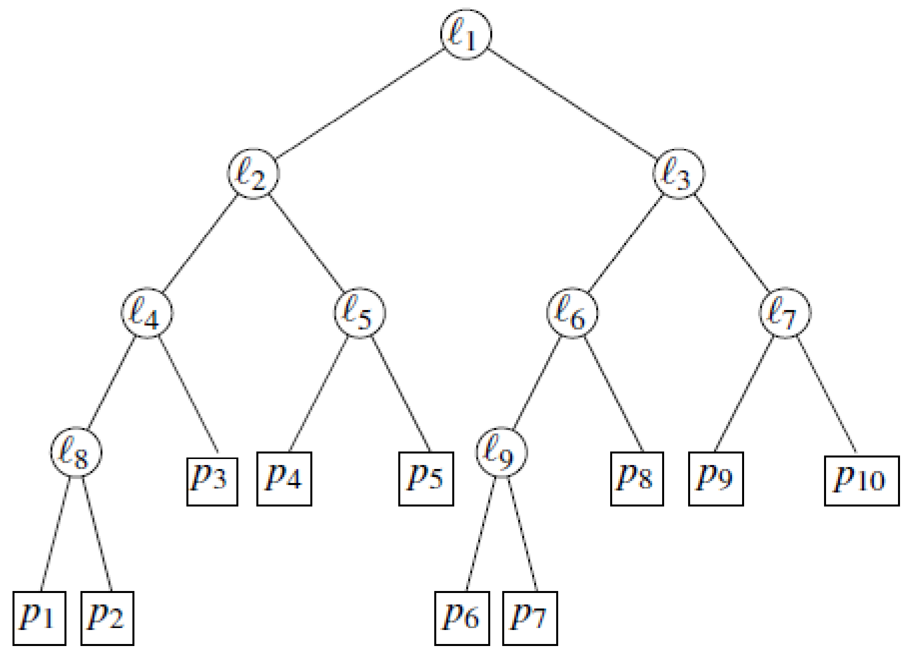
- Idea: generalization of binary search trees
- Kd-tree is still a binary tree
- Data points are at leaves
- For each internal node *v* :
  - **if** the depth of *v* is *even, x*-coordinates of left sub-tree ≤ *v* < *x*-coordinates of right sub-tree (*split with a vertical line*).
  - **if** the depth of *v* is *odd, y*-coordinates of left sub-tree ≤ *v* < *y*-coordinates of right sub-tree (*split with a horizontal line*).

Example kd-tree



Example kd-tree





Building a kd-tree from point set P

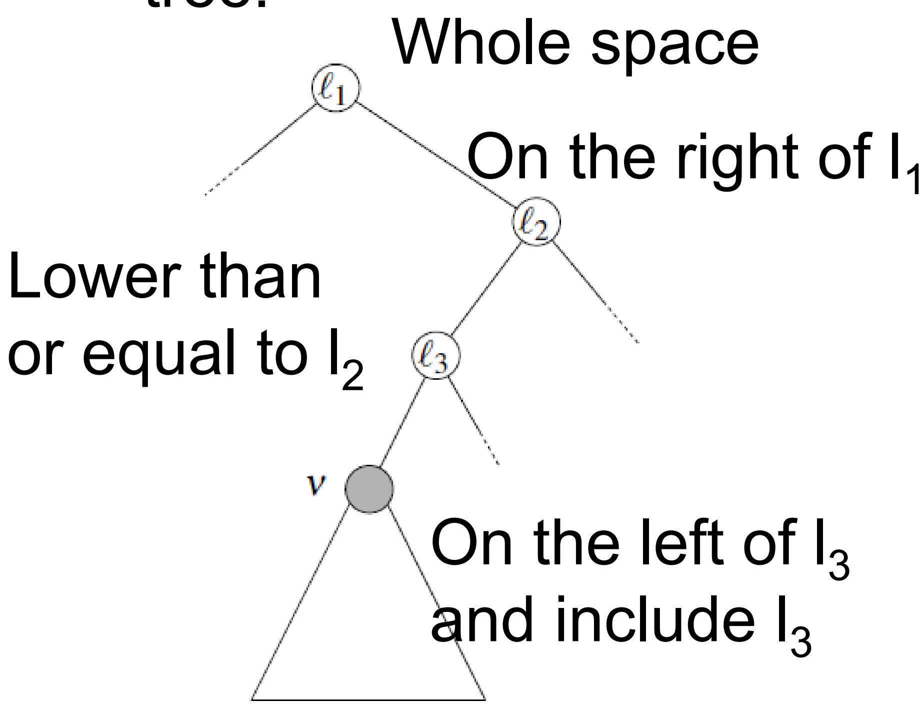
- Divide-and-conquer
  - Sort the points in P w.r.t. their *x*-coordinates into array *X*.
  - Sort the points in P w.r.t. their y-coordinates into array *Y*.
  - Base case: if P contains only one point, returns a leaf with the point.
  - Otherwise: divide into 2 sub-problems and conquer them recursively.
    - If the depth is even (split w.r.t. x-axis or a vertical line)
      - Take the median v of X and create a root v_{root}
      - Split X into sorted X_L and X_R & split Y into sorted Y_L and Y_R : s.t. for any $p \in X_L$ or $p \in Y_L$, $p.x \le v.x$ *and* for any $p \in X_R$ or $p \in Y_R$, $p.x > v.x$
      - Build recursively the left child of v_{root} from X_L and Y_L
    - If the depth is odd (split w.r.t. y-axis or a horizontal line)
      - Take the median v of Y and create a root v_{root}
      - Split X into sorted X_L and X_R & split Y into sorted Y_L and Y_R : s.t. for any $p \in X_L$ or $p \in Y_L$, $p.y \le v.y$ and for any $p \in X_R$ or $p \in Y_R$, $p.y > v.y$
      - Build recursively the right child of v_{root} from X_R and Y_R

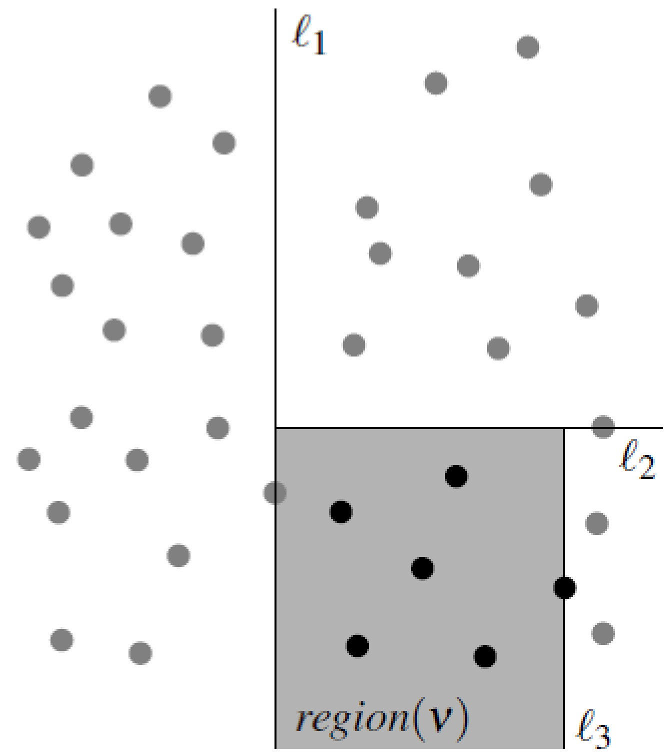
Run-time of building a kd-tree

- What is the running time of building a kd-tree?
- Sorting points in P according to x- and y-coordinates, respectively.
  - Two times of sorting. Each takes $\Theta(nlgn)$
- What is the recurrence?
  - Divide: finding the median Θ(1) (as sorted already) and split X and Y in Θ(n).
  - Conquer: 2T(n/2), 2 sub-problems, each sub-problem is with half the size of the original problem.
  - Combine: constant, connect the left/right children with the root.
  - $T(n)=2T(n/2)+\Theta(n)$
    - Master method, case 2: $\Theta(nlgn)$
- In total, Θ (nlgn).

Querying the kd-tree

- The region of an internal node region(v)
- We can maintain region(v) when we traverse down the tree.





Querying algorithm

- Given a range query with range R:
- Start traversing the kd-tree from the root node v.
  - If region (v) **does not intersect** R, do not go deeper into the sub-tree rooted at v.
  - If region (v) is **fully contained** in R, report all points in the subtree rooted at v.
  - If region (v) **only intersects** with R, go recursively into v' s children, and check its children nodes.
- Note that we are checking if the region of an internal node of a kd-tree is fully contained in the query range R
  - We are not checking if the query range R is fully contained in the region of an internal node of a kd-tree.

Pseudo code

Algorithm SEARCHKDTREE(v, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below v that lie in the range.

```
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)
```

Leaf node

Left sub-tree

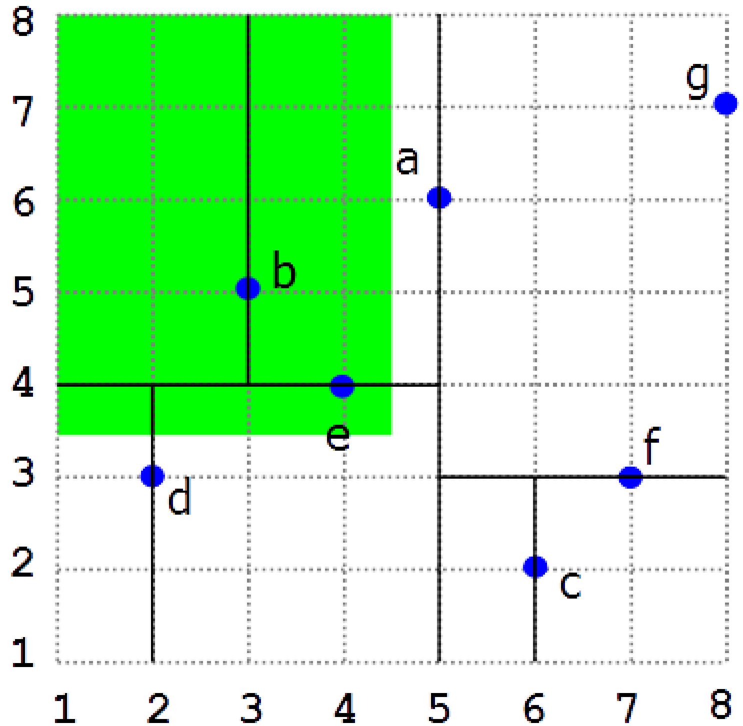
Right sub-tree

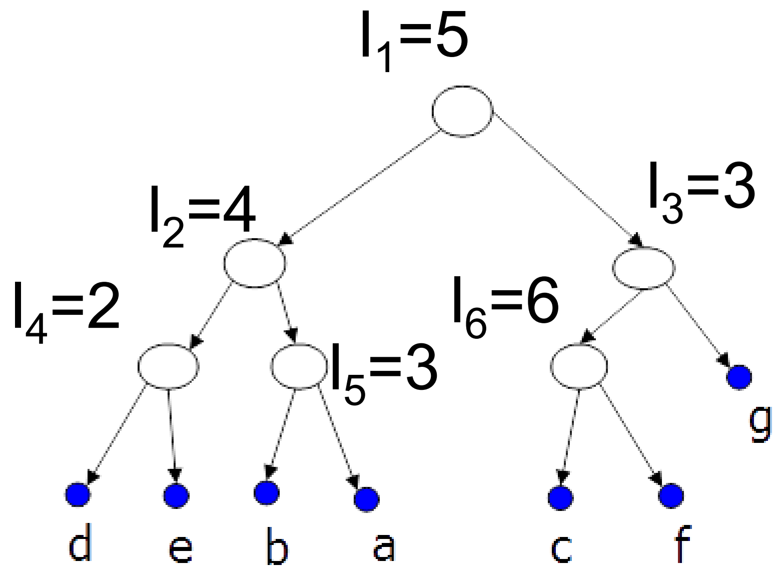
Internal node

lc(v) and rc(v) return the left and right child node of node v.

Mini-quiz (also on Moodle)

- Range searching [1, 4.5], [3.5, 8]
  - Which leave nodes need to be checked?





Analysis of the querying algorithm

- When region (v) is fully contained in R, we traverse the whole sub-tree rooted at v.
  - Assume that the total number of points in the output is k, then $\Theta(k)$.
- When region (v) intersects R.
  - R has four edges, i.e., line segments. For each edge, identify how many regions can an edge intersect at most, i.e., an upper bound .
  - Assume that we consider a vertical edge l.
  - At root v with a vertical splitting line, l is either in the region(v.left) or the region(v.right).
    - T(n)=1+T(n/2)
  - At a node in the next level, it is with a horizontal splitting line, so l may intersect both regions.
    - T(n/2)=1+2T(n/4)
  - We have to consider “going down two steps” together.
    - We have recurrence T(n) = 2+2T(n/4). After solving it, we have $\Theta(\sqrt{n})$
- In total, O($\sqrt{n}$ + k).

kd-tree summary

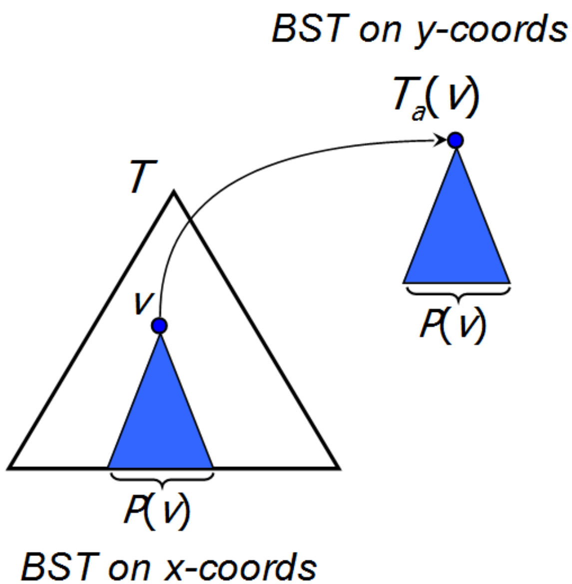
- Kd-tree:
  - Building (preprocessing time): $\Theta(n \log n)$
  - Size: $\Theta(n)$
- Range queries:
  - $O(\sqrt{n} + k)$.

Agenda

- Range Searching
- 1D range searching
- 2D range searching
- KD-trees
- Range trees

Range trees

- **Canonical subset** $P(v)$ of a node v in a balanced BST is a set of points (leaves) stored in a sub-tree rooted at v.
  - When v is the root, it contains all the points.
  - When v is a leaf node, it contains the point itself in the leave.
- **Range tree**
  - The main tree is a BST T on the *x*-coordinates of points
  - Each node v of T stores a pointer to a BST $T_a(v)$ (**associated structure** of v), which stores the canonical subset $P(v)$ organized on the *y*-coordinate
  - 2D points are stored in all leaves!
- Range tree is a *multi-level data structure*.
  - When nodes have pointers to associated structures.

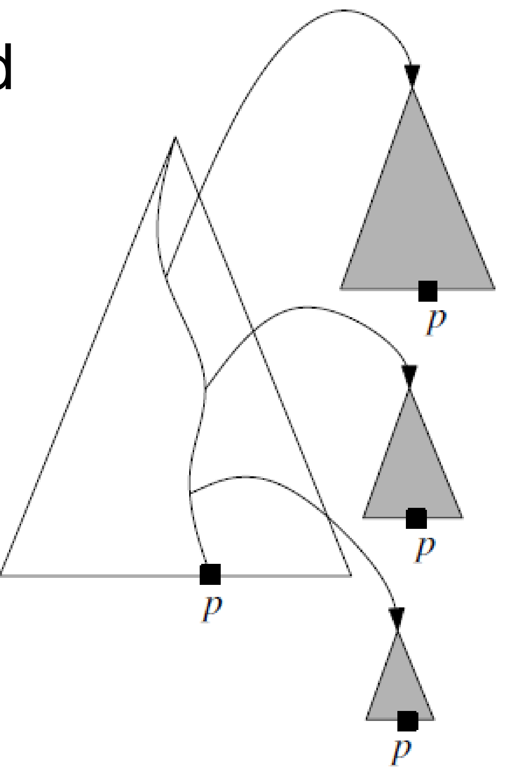


Building a range tree

- Sort the points on x-axis and on y-axis (two arrays: X, Y)
  - $\Theta(n \log n)$
- Divide-and-conquer:
- Divide
  - Take the median v of X and create a root,
    - Constant time.
  - Build its associated structure using Y.
    - Run-time: $\Theta(n)$, building a BST on sorted points can be done in linear time.
- Conquer
  - Split X into sorted X_L and X_R , split Y into sorted Y_L and Y_R
    - For any $p \in X_L$ or $p \in Y_L$, $p.x \leq v.x$
    - For any $p \in X_R$ or $p \in Y_R$, $p.x > v.x$
  - Build recursively the left child from X_L and Y_L and the right child from X_R and Y_R
- $T(n)=2T(n/2)+ \Theta(n)$, Thus, run-time: $\Theta(n \log n)$

Storage of a range tree

- A point p is stored only in the associate structures of nodes that are on the path in the main tree T towards the leaf containing p.
- Thus, in each level of the tree, p is stored exactly once in the associated structure.
- The storage for each level is $\Theta(n)$
- The height of the main tree is $\Theta(\lg n)$
- Total storage: $\Theta(n\lg n)$



Range query on range trees

- How do we perform range query on such a range tree?
  - For x-range [x1, x2]
    - Use the 1DRangeSearch on the main tree T
  - For y-range [y1, y2]
    - Replace ReportSubtree (v) with 1DRangeSearch (T_a (v), y_1 , y_2)
    - T_a(v) indicates the associated structure of node v.

Range Query

2DRangeSearch(T, x_1, x_2, y_1, y_2)
01 v ← *FindSplit*(T, x_1, x_2)
02 **if** v is a leaf **then**
03   **if** $x_1 \le$ v.key.x $\le x_2$ **and** $y_1 \le$ v.key.y $\le y_2$ **then return** v
04 **else return** *DoLeft*(v.leftChild, x_1, x_2, y_1, y_2) ∪
            DoRight(v.rightChild, x_1, x_2, y_1, y_2)

The x coordinates of the leaves in v.rightChild must lie in [x1, x2], then we check if the y coordinates lie in [y1, y2].

DoLeft(v, x_1, x_2, y_1, y_2)
01 **if** v is a leaf **then**
02   **if** $x_1 \le$ v.key.x $\le x_2$ **and** $y_1 \le$ v.key.y $\le y_2$ **then return** v
03 **else**
        1DRangeSearch(T_a(v.rightChild), y_1, y_2)
04   **if** $x_1 \le$ v.key.x **then return** ~~*ReportSubtree*(v.rightChild)~~ ∪
            DoLeft(v.leftChild, x_1, x_2, y_1, y_2)
05   **else return** *DoLeft*(v.rightChild, x_1, x_2, y_1, y_2)

DoRight(v, x_1, x_2, y_1, y_2)
// similar to *DoLeft*, but with modified lines 04-05

Runtime of range query

- Worst-case: We need to query the associated structures on all nodes on the path down in the main tree.
  - At a node v on the path down in the main tree, we make a recursive call on its associated structure.
    - If node v is in level j, its canonical set has $\frac{n}{2^j}$ points , thus its associated structure, i.e., the BST on y-coord, has depth lg $\frac{n}{2^j}$=lgn - j
    - Thus, the cost for this call is $\Theta(\lg n - j + k_v)$, where k_v is the number of reported points in the sub-tree that is rooted at v.
  - Then, sum over all possible node v from level 0 to lgn.
    - $\sum_v \Theta(\lg n - j + k_v)$
    - $\sum_v \Theta(k_v) = k$, the total points that are in the 2D range.
    - $\sum_v \Theta(\lg n - j) = \lg^2 n - (0+1+\ldots+\lg n) = \Theta(\lg^2 n)$. There will be in total at most lgn nodes along the path because the height of the tree is lgn.
- Thus, the total cost $O(\lg^2 n + k)$.

Range-trees vs kd-trees

- Building trees runtime
  - $\Theta(n \log n)$ vs $\Theta(n \log n)$
- Range search runtime
  - $\Theta(\lg^2 n + k)$ vs $\Theta(\sqrt{n} + k)$
  - Which one is faster?
- Storage
  - $\Theta(n\lg n)$ vs $\Theta(n)$
  - Which one takes more space?
- An example on trading space for efficiency!

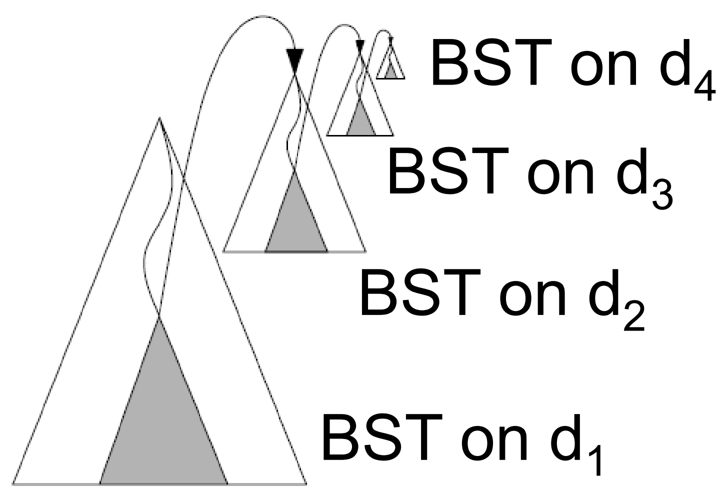
2-dimensional vs n-dimensional

- Kd-tree
  - Split on d_1
  - Split on d_2
  - ..
  - Split on d_n
  - Split on d_1
  - Split on d_2
  - …
  - Split on d_n

- What about the run time of a rang query? How many levels should you consider together?
  - Exercise 5

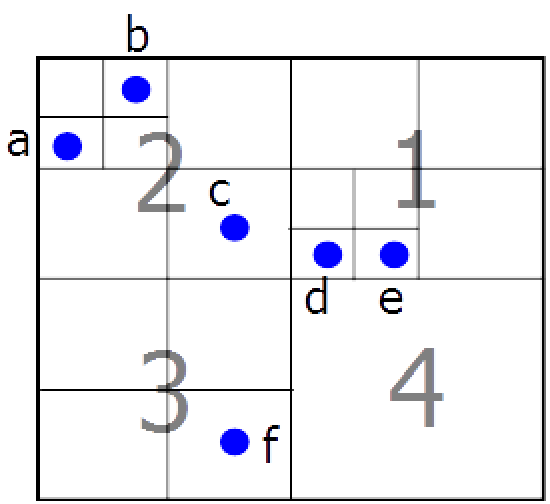
2-dimensional to n-dimensional

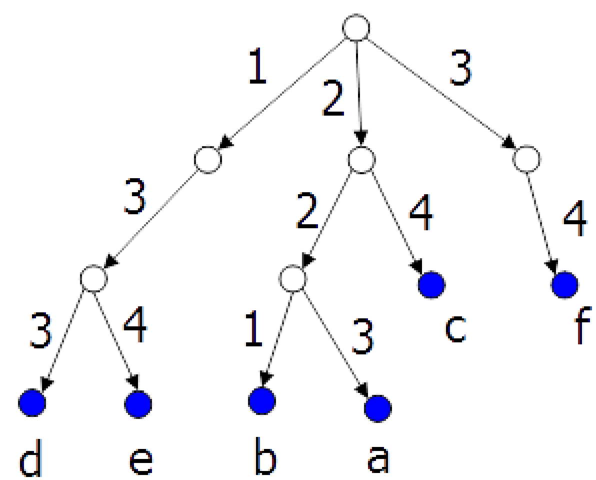
- n-dimensional range tree
  - Main tree on d_1
  - Each interval node has an associated structure which is a (n-1)-dimensional range tree
  - In each internal node in a (n-1)-dimensional range tree has an associated structure which is a (n-2)-dimensional range tree.
  - …



Quad-trees

- *A four-way partition tree*
- Linear space
- Good average query performance





ILO of Lecture 8

- Computational Geometry: range searching
  - to understand and to be able to analyze the balanced binary search tree based 1D range searching algorithm;
  - to understand and to be able to analyze the kd-trees and the range trees;
  - to understand how data structures can be used to trade the space used for the running time of queries.

Self-study 2 on 2nd March

- Please send your solutions to Sean or Tung before 9 March.