CS633 Lecture 05 Orthogonal Range Search and Geometric Trees

Jyh-Ming Lien

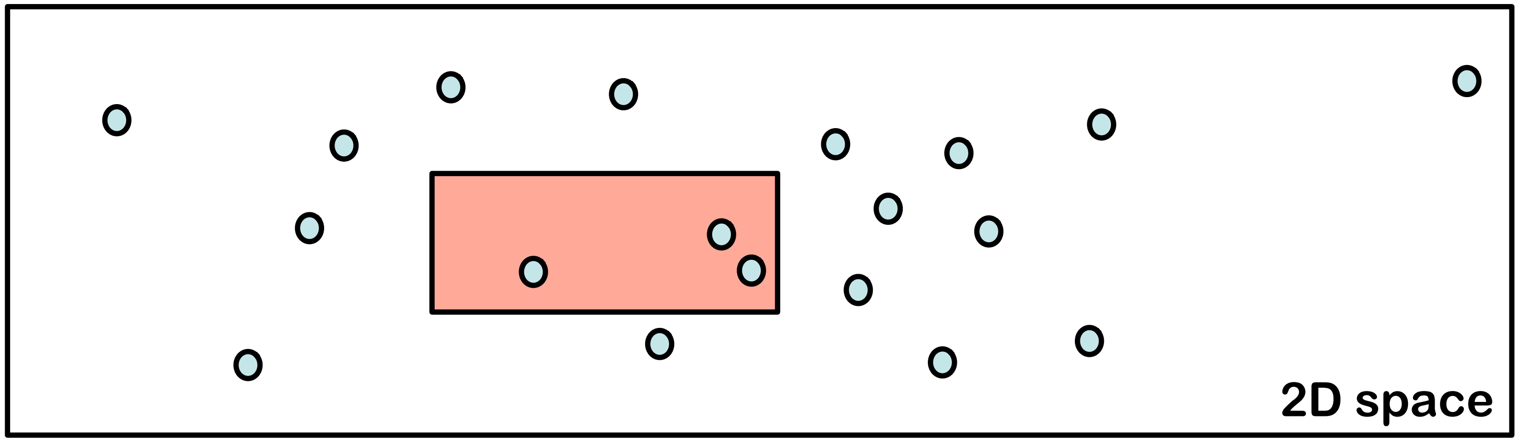
Deptartment of Computer Science

George Mason University

Based on Chapter 5 of the textbook
And Ming Lin's lecture note at UNC

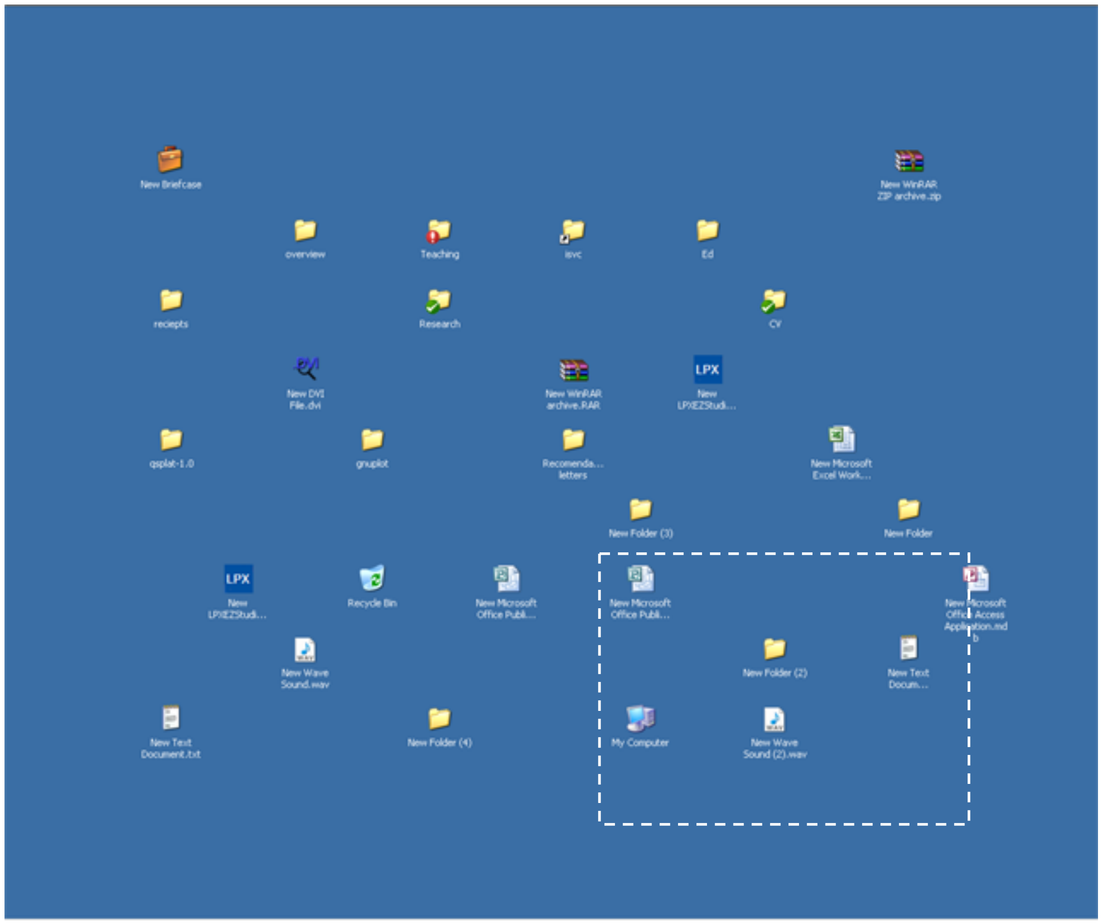
Orthogonal Range Searching

- Given a set of k-D points and an orthogonal range (whose boundaries are parallel to the coordinate axes), find all points enclosed by this query range



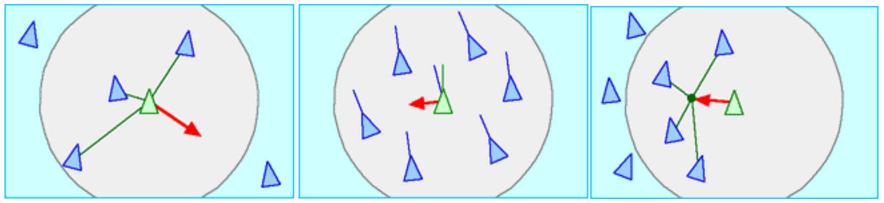
- Brute force: O(n), is this necessary?

Selecting Desktop Icons



Orthogonal Range Searching

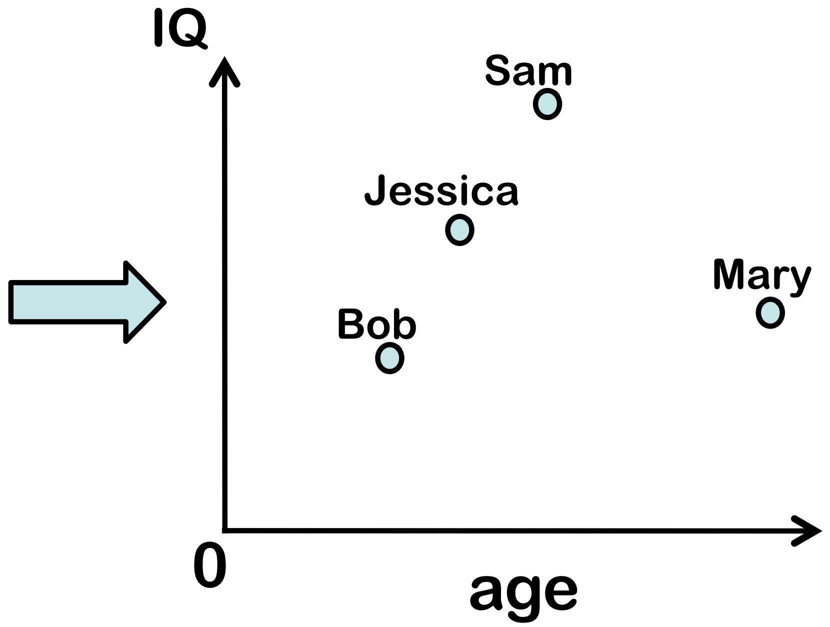
- Driving Applications
  - Database
  - Geographic Information System
  - Simulating group behaviors (bird homing)



Interpret DB Queries Geometrically

- Transform records in database into points in multi-dimensional space.

name	age	IQ
Bob	12	75
Jessica	21	132
Mary	88	89
Sam	34	180

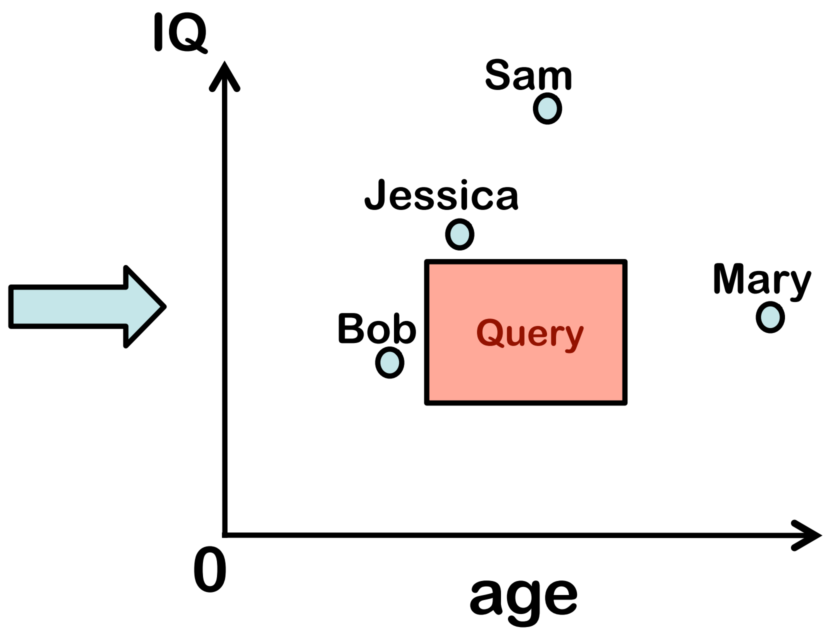


Interpret DB Queries Geometrically

- Transform queries on *d*-fields of records in the database into queries on this set of points in *d*-dimensional space

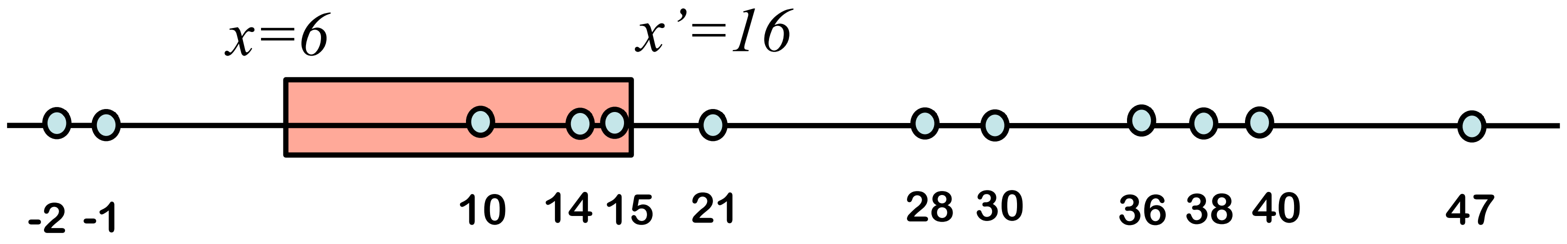
Query: age between 18 and 38, IQ between 70 and 110

name	age	IQ
Bob	12	75
Jessica	21	132
Mary	88	89
Sam	34	180



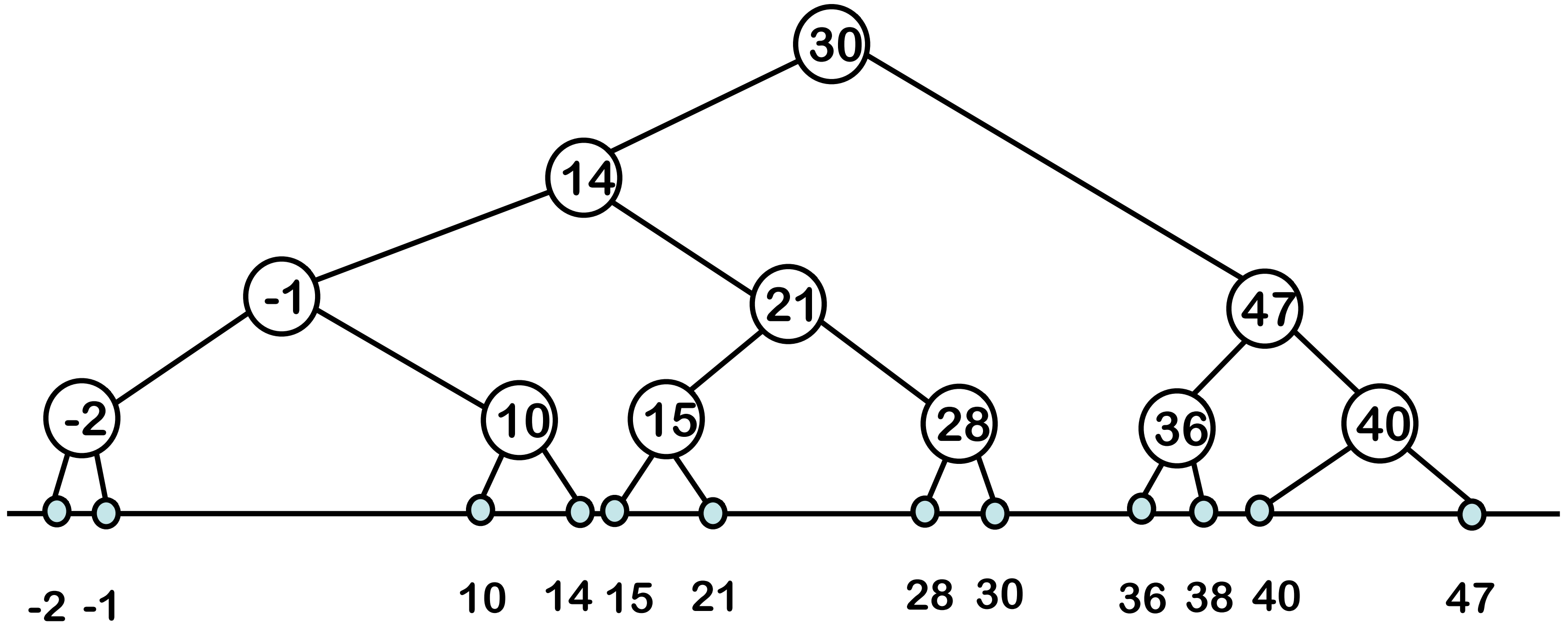
1-D Range Searching

- Let's solve a simple problem first
  - Let $P := \{p_1, p_2, \ldots, p_n\}$ be a given set of points on the real line. A query asks for the points inside a 1-D query rectangle -- i.e. an interval $[x:x']$



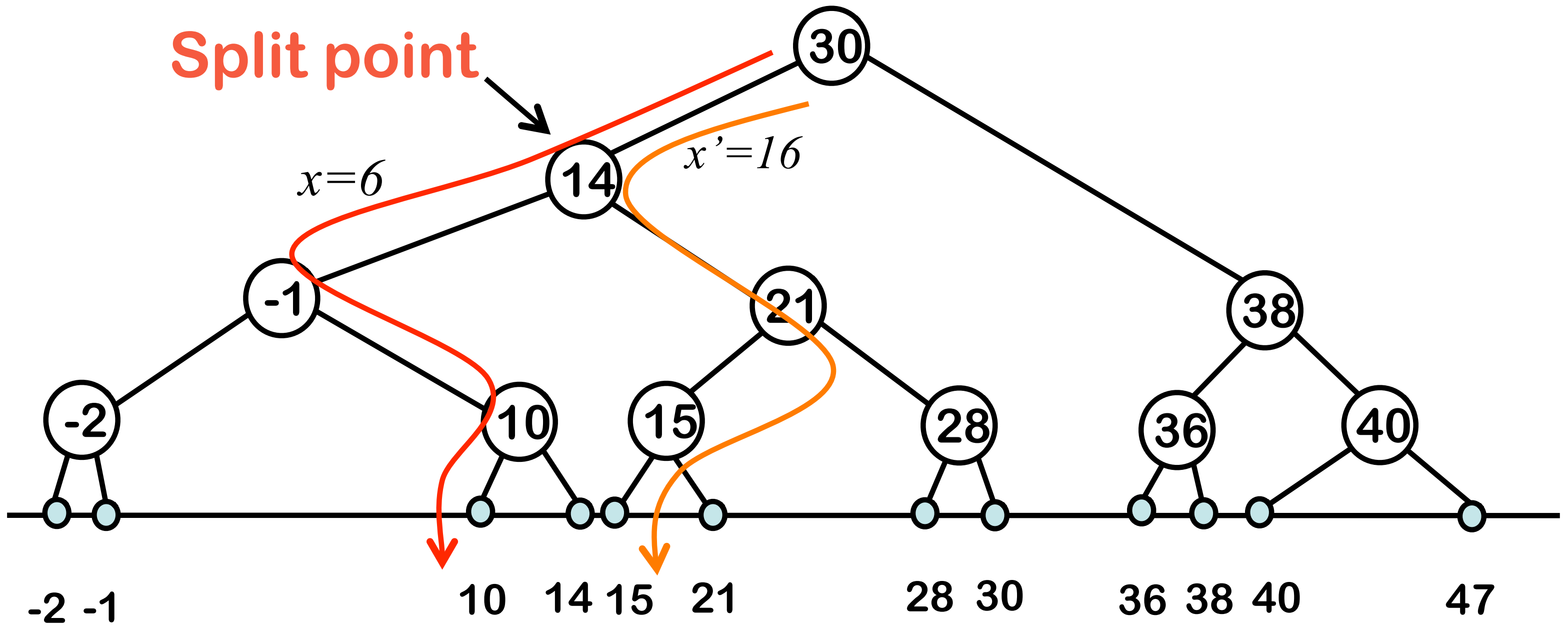
1-D Range Searching

- Use a balanced binary search tree *T*.
  - The leaves of *T* store the points of *P*
  - The internal nodes of *T* store splitting values to guide the search
    - The largest value in the left sub-tree



1-D Range Searching

- To report points in [*x*:*x'*], we search with *x* and *x'* in *T*.
  - Let *u* and *u'* be the two leaves where the search ends resp.
  - Then the points in [*x*:*x'*] are the ones stored in leaves between *u* and *u'*, plus possibly points stored at *u* & *u'*.



1D Range Query

Input: A range tree T and a range $[x{:}x']$

Output: All points that lie in the range.

```
1.  $v_{split} \leftarrow$ FindSplitNode($T, x, x'$)
2.  if $v_{split}$ is a leaf
3.     then Check if the point stored at $v_{split}$ must be reported
4.     else (* Follow the path to $x$ and report the points in
               subtrees right of the path *)
•       $v \leftarrow lc(v_{split})$
6.          while $v$ is not a leaf
7.              do if $x \leq x_v$
8.                     then ReportSubTree($rc(v)$)
9.                     $v \leftarrow lc(v)$
10.                else $v \leftarrow rc(v)$
11.             Check if the point stored at leaf $v$ must be reported
12. Similarly, follow the path to $x'$
```

$\rightarrow$ **move to left**

$\rightarrow$ **move to right**

Find Split Node

Input: A tree T and two values x and x' with $x \leq x'$

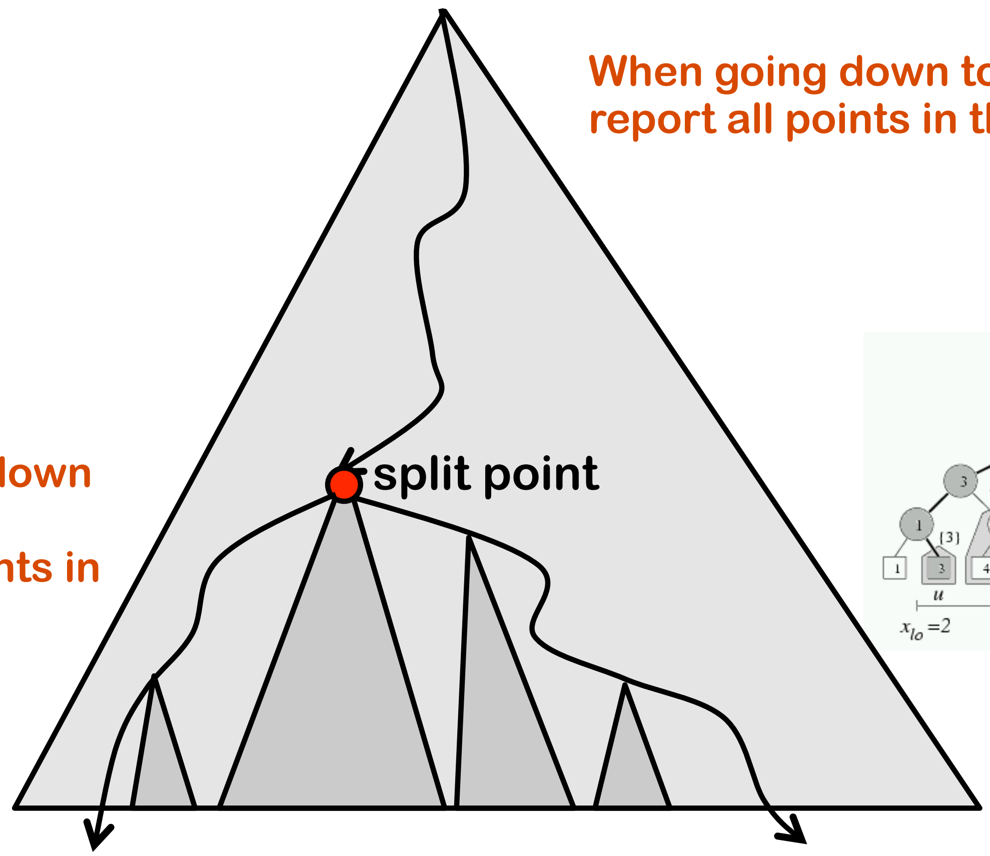
Output: The node v where the paths to x and x' splits, or the leaf where both paths end.

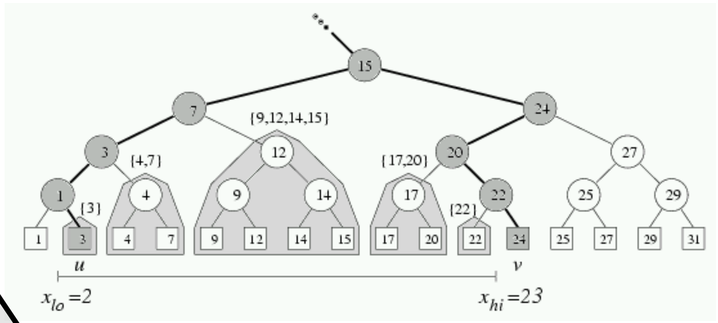
```
1. $v \leftarrow root\ (T)$
2. while $v$ is not a leaf and ($x' \leq x_v$ or $x > x_v$)
3.        do if $x' \leq x_v$
4.               then $v \leftarrow lc(v)$          (* left child of the node $v$ *)
5.               else $v \leftarrow rc(v)$          (* right child of the node $v$ *)
6. return $v$
```

1-D Range Searching

When going down to the right, report all points in the left

When going down to the left, report all points in the right





1D-Range Search Algorithm Analysis

- Let P be a set of n points in one-dimensional space
  - uses $O(n)$ storage and has $O(n \log n)$ construction time
  - The points in a query range can be reported
    - Time $O(k + \log n)$, where k is the number of reported points
      - The time spent in "ReportSubtree" is linear in the number of reported points, i.e. $O(k)$.
      - The remaining nodes that are visited on the search path of x or x'. The length is $O(\log n)$ and time spent at each node is $O(1)$, so the total time spent at these nodes is $O(\log n)$.

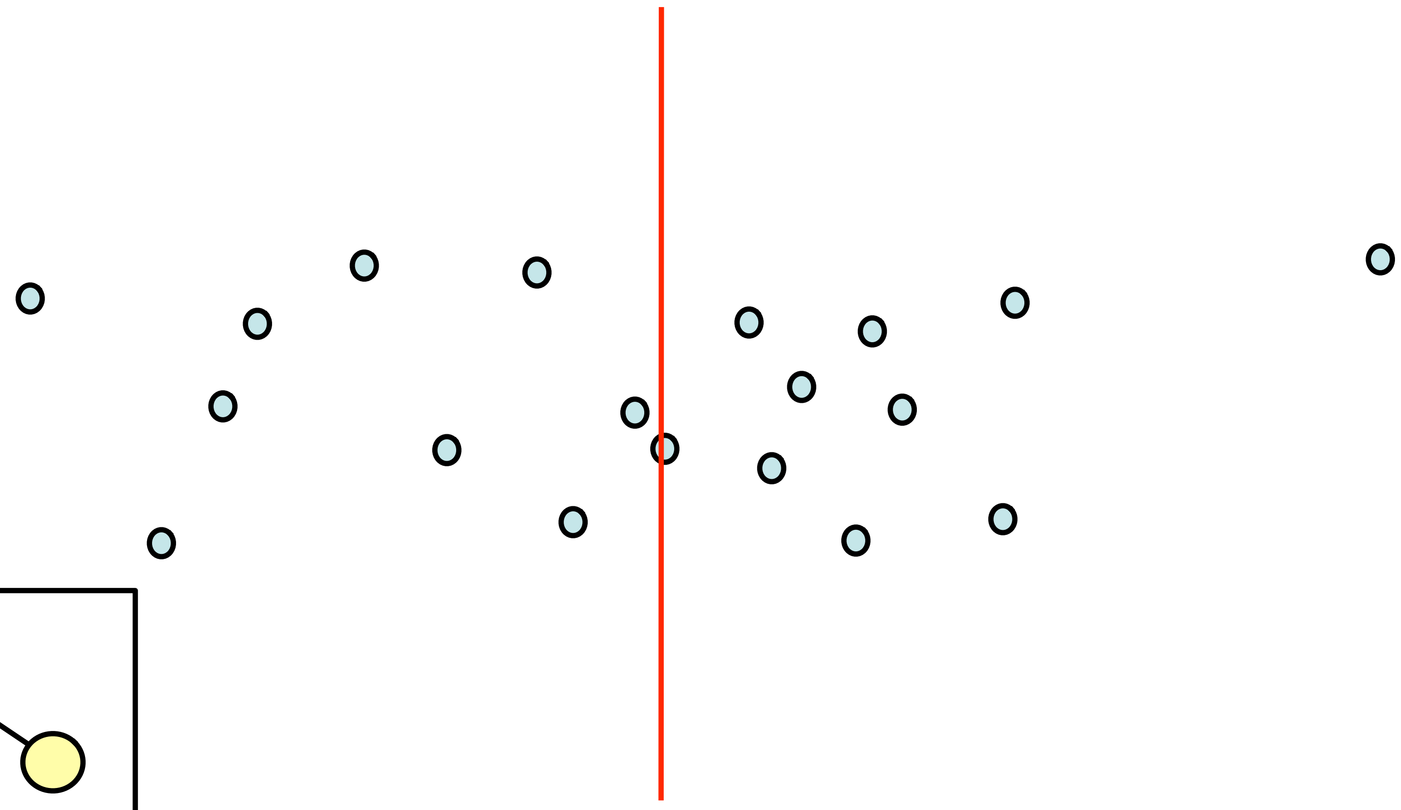
K-D-Range Search

- Extending binary search tree
  - K-D tree
    - Stack different dimensions in a tree
    - Require less memory
    - Slower
  - Range tree
    - Hierarchical structure of trees
    - Require more memory
    - Faster

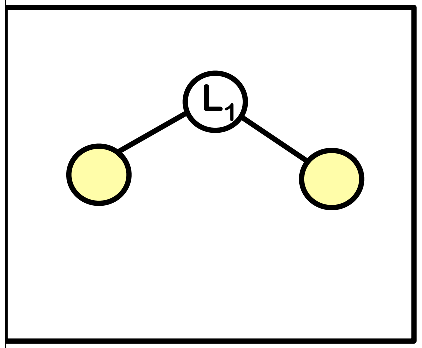
Kd-Trees

- Let's look at 2-D problems
  - A 2d rectangular query on *P* asks for points from *P* lying inside a query rectangle $[x{:}x']$ x $[y{:}y']$. A point $p := (p_x, p_y)$ lies inside this rectangle iff $p_x \in [x{:}x']$ and $p_y \in [y{:}y']$
  - At the root, we split *P* with *l* into 2 sets and store *l*. Each set is then split into 2 subsets and stores its splitting line. We proceed recursively as such
  - In general, we split with vertical lines at nodes whose depth is even, and split with horizontal lines at nodes whose depth is odd

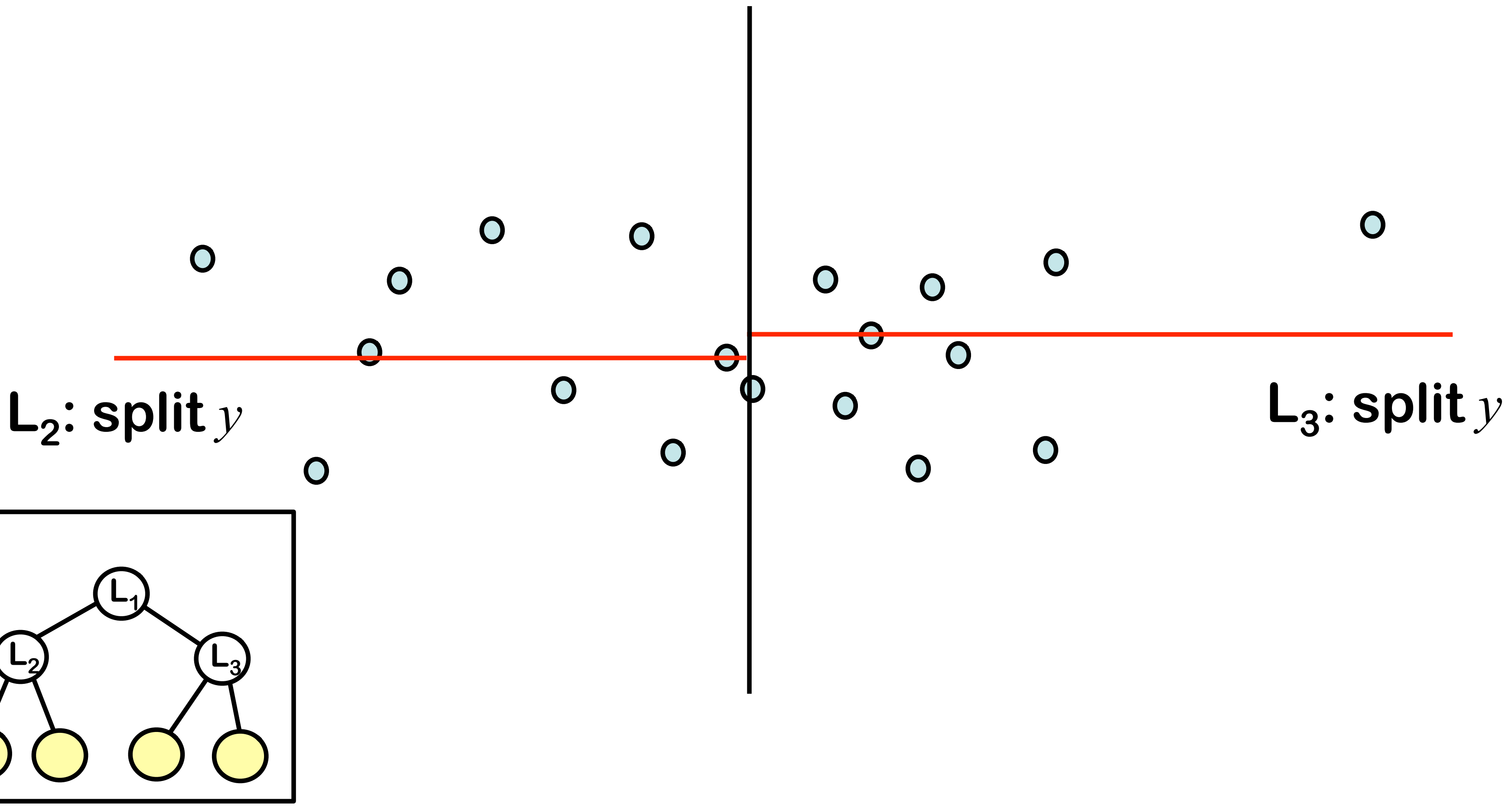
Kd-Trees



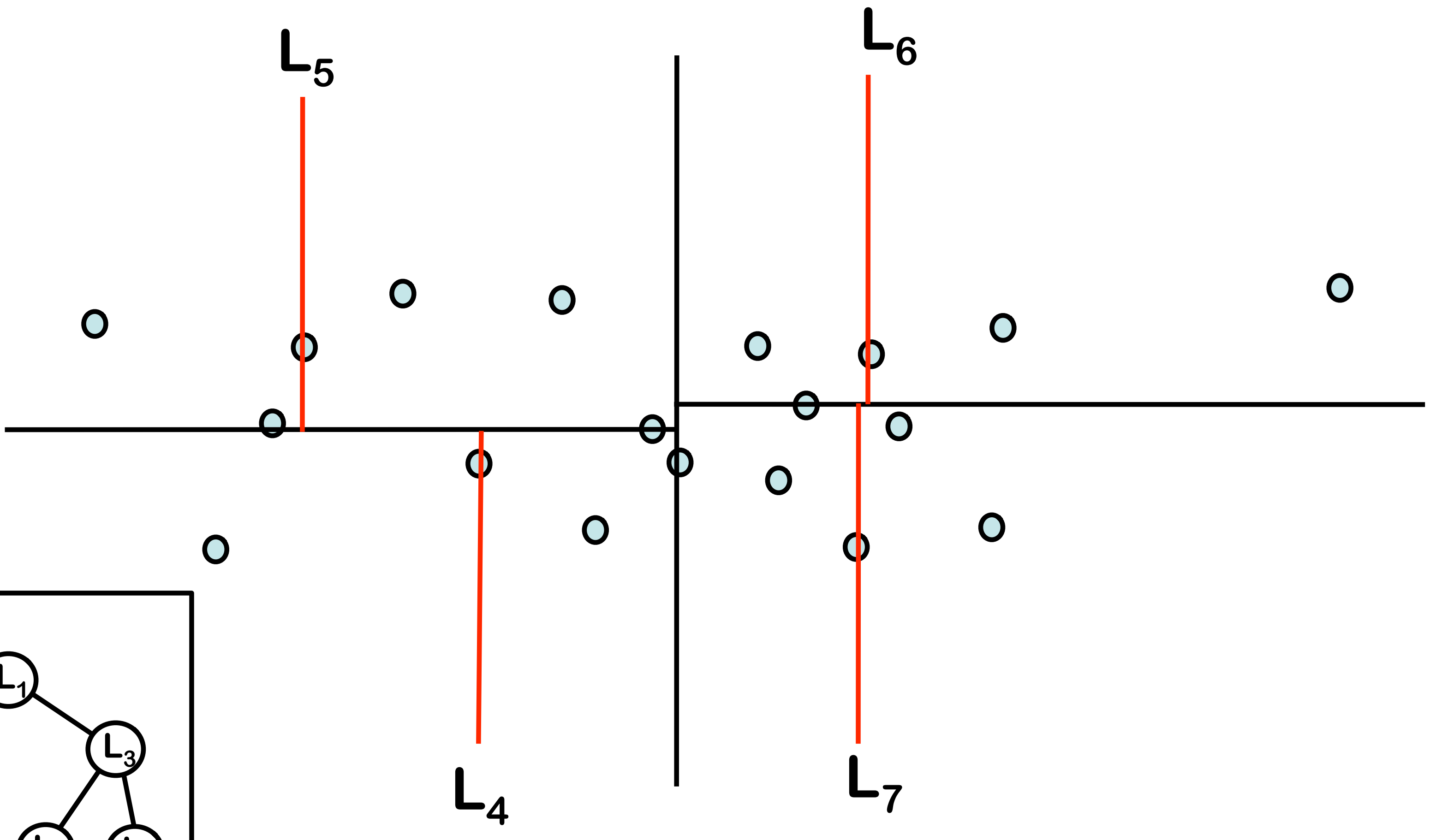
L_1: split x

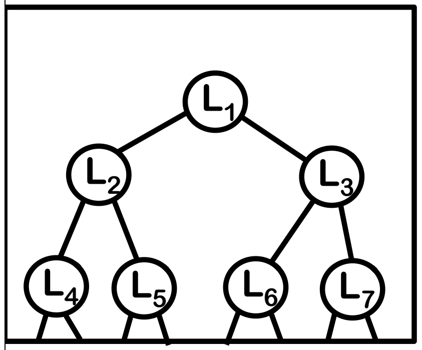


Kd-Trees



Kd-Trees





BuildKDTree(*P, depth*)

Input: A set *P* of points and the current depth, *depth*.

Output: The root of kd-tree storing *P*.

```
1.  if $P$ contains only 1 point
2.      then return a leaf storing at this point
3.      else if $depth$ is even
4.              then Split $P$ into 2 subsets with a vertical line $l$ thru median
                     $x$-coordinate of points in $P$. Let $P_1$ and $P_2$ be the sets of
                     points to the left or on and to the right of $l$ respectively.
5.              else Split $P$ into 2 subsets with a horizontal line $l$ thru median
                     $y$-coordinate of points in $P$. Let $P_1$ and $P_2$ be the sets of
                     points below or on $l$ and above $l$ respectively.
5.          $v_{left} \leftarrow$ BuildKDTree($P_1$, $depth + 1$)
6.          $v_{right} \leftarrow$ BuildKDTree($P_2$, $depth + 1$)
```

Construction Time Analysis

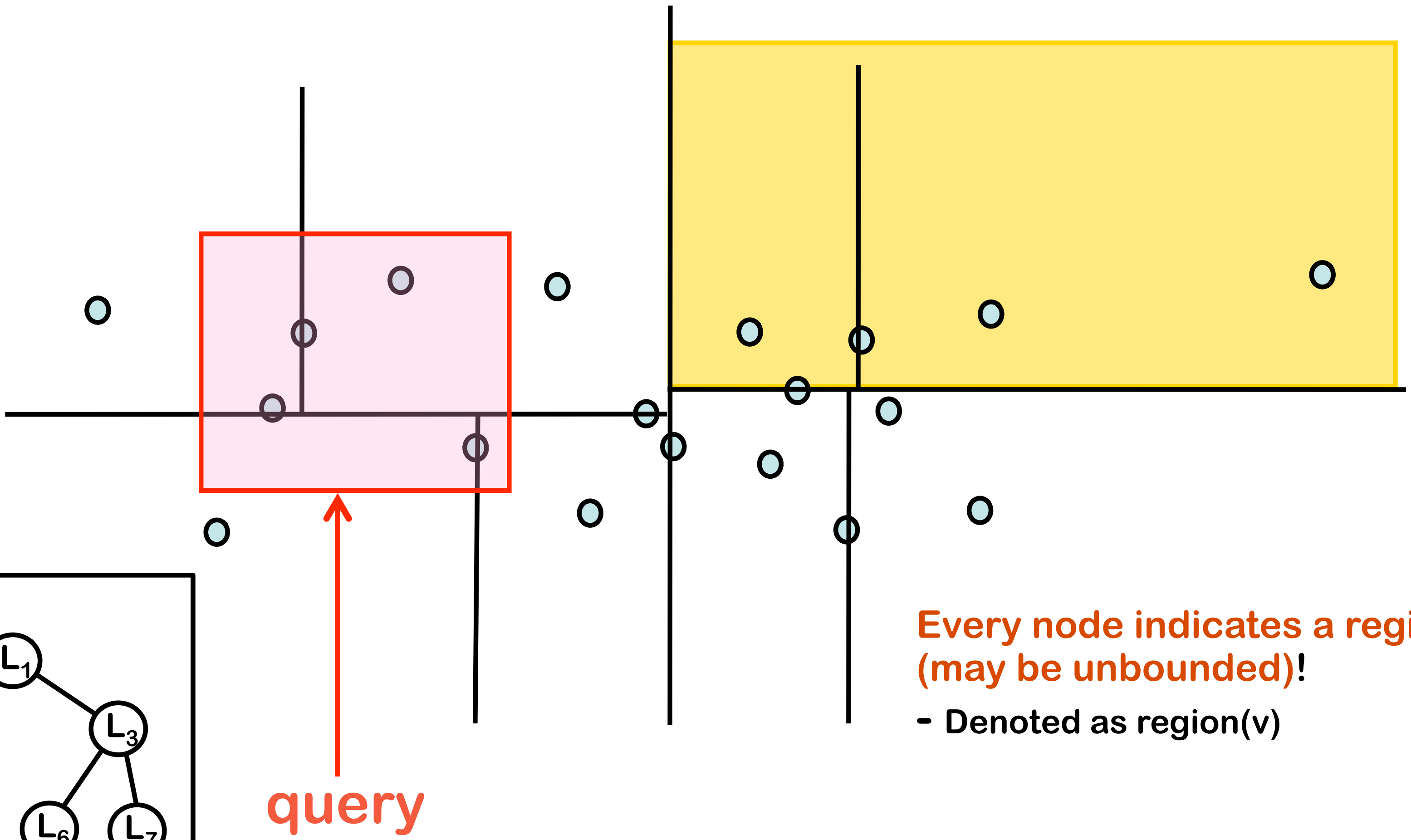
- The most expensive step is median finding, which can be done in linear time.

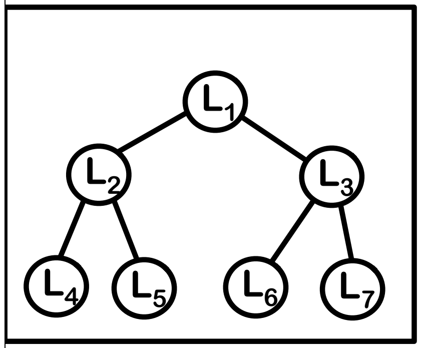
- $T(n) = O(1)$, if $n = 1$
- $T(n) = O(n) + 2\,T(n/2)$, if $n > 1$

$\Rightarrow T(n) = O(n \log n)$

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

Querying on a *kd-Tree*





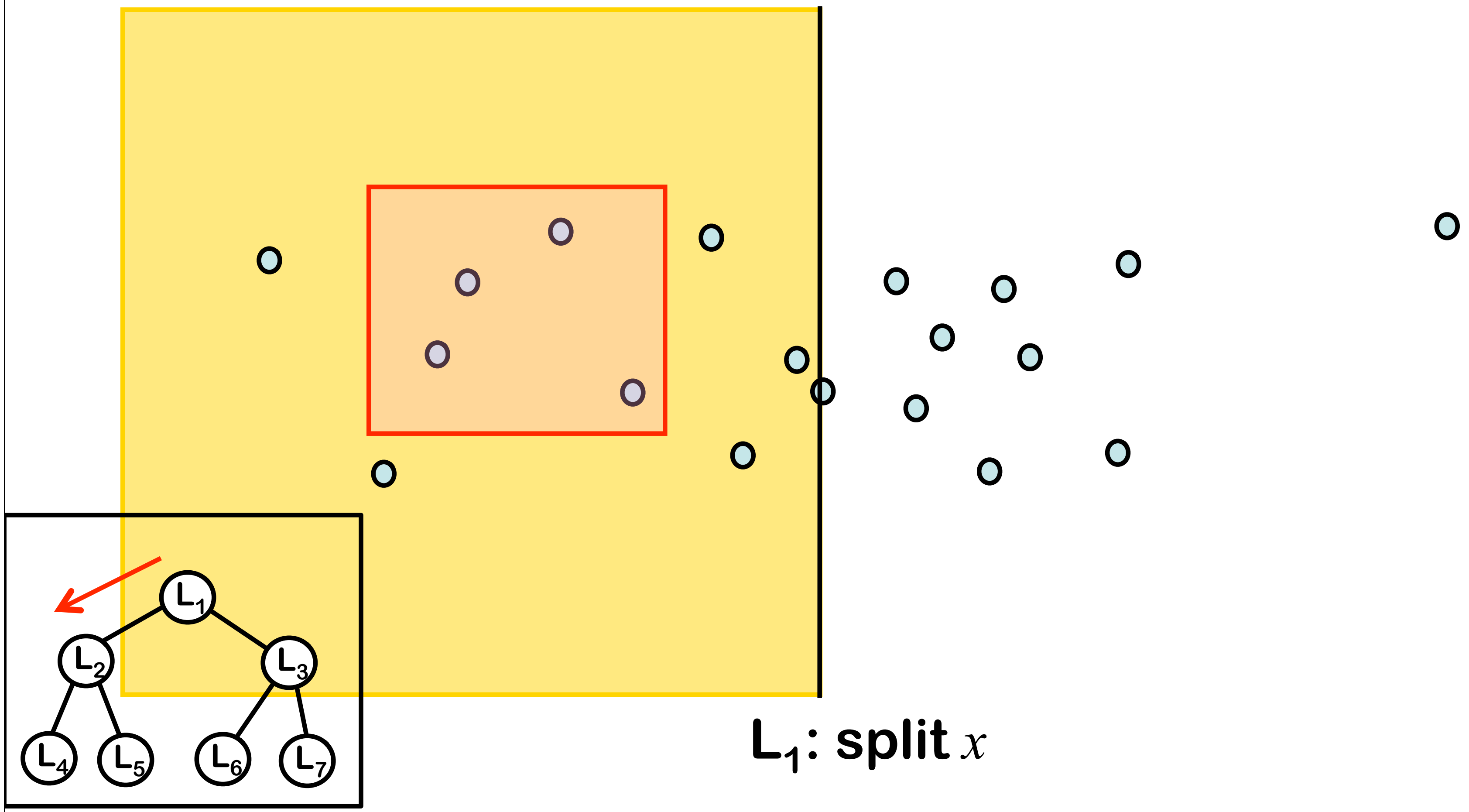
Every node indicates a region (may be unbounded)!

- Denoted as region(v)

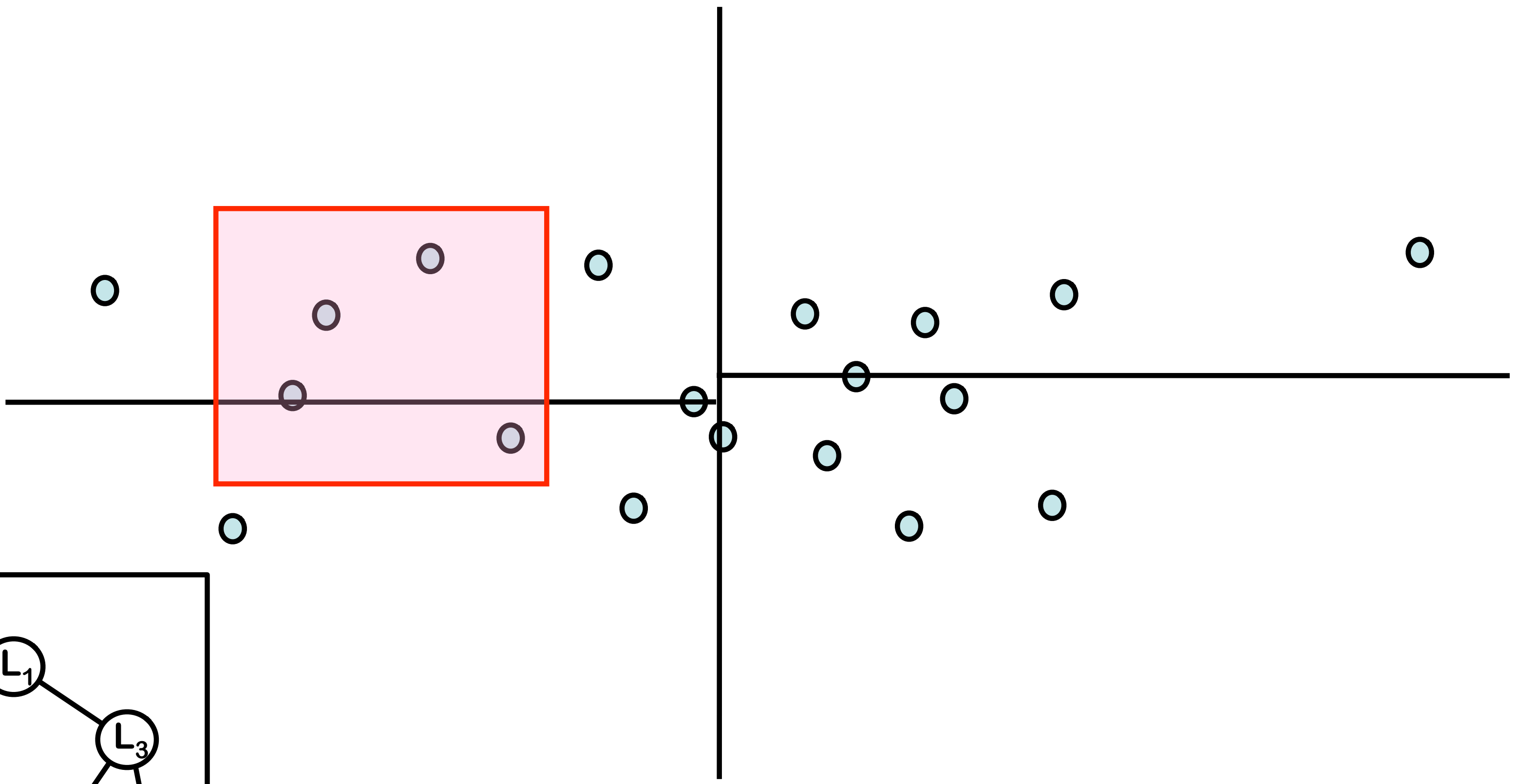
Querying on a *kd-Tree*

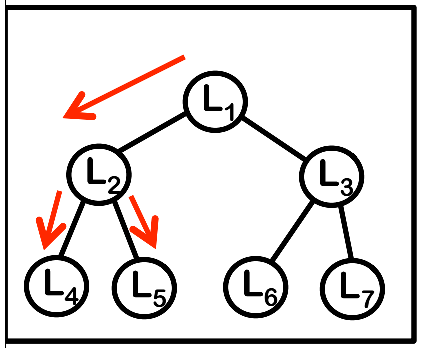
- We traverse the *kd-tree*
  - Visit only nodes whose region is intersected by the query rectangle.
  - When a region is fully contained in the query rectangle, we report all points stored in its sub-trees.
  - When the traversal reaches a leaf, we have to check whether the point stored at the leaf is contained in the query region

Querying on a *kd-Tree*

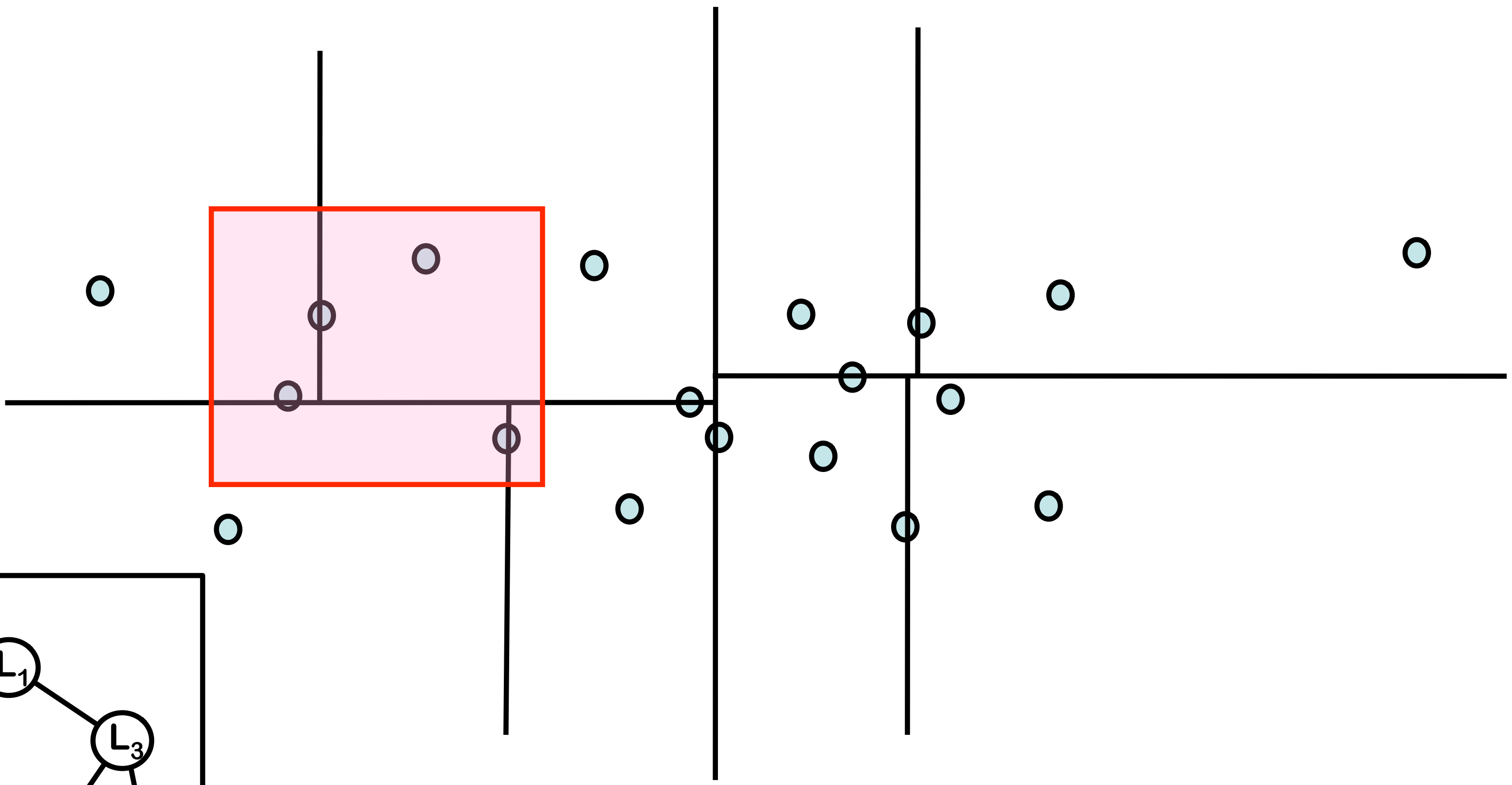


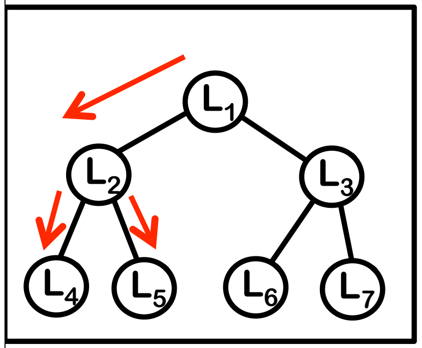
Querying on a *kd-Tree*





Querying on a *kd-Tree*





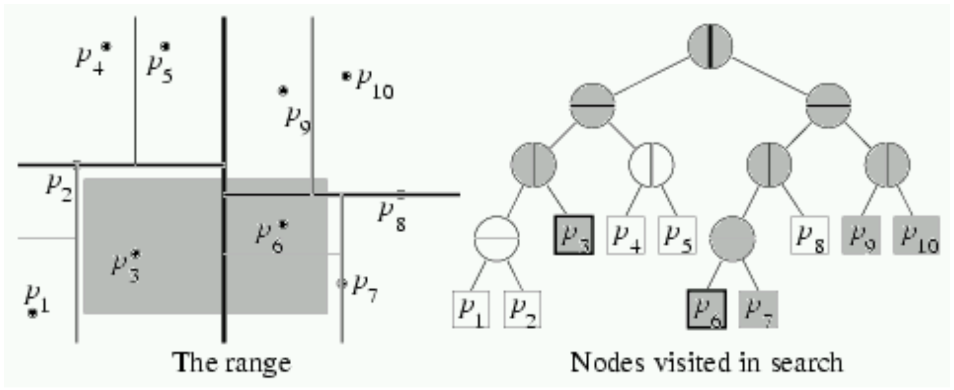
SearchKDTree(v, R)

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

```
1.  if $v$ is a leaf
2.     then Report the point stored at $v$ if it lies in $R$.
3.     else if $region(lc(v))$ is fully contained in $R$
4.            then ReportSubtree($lc(v)$)
5.            else if $region(lc(v))$ intersects $R$
6.                   then SearchKDTree($lc(v)$, $R$)
7.          If $region(rc(v))$ is fully contained in $R$
8.            then ReportSubtree($rc(v)$)
9.            else if $region(rc(v))$ intersects $R$
10.                  then SearchKDTree($rc(v)$, $R$)
```

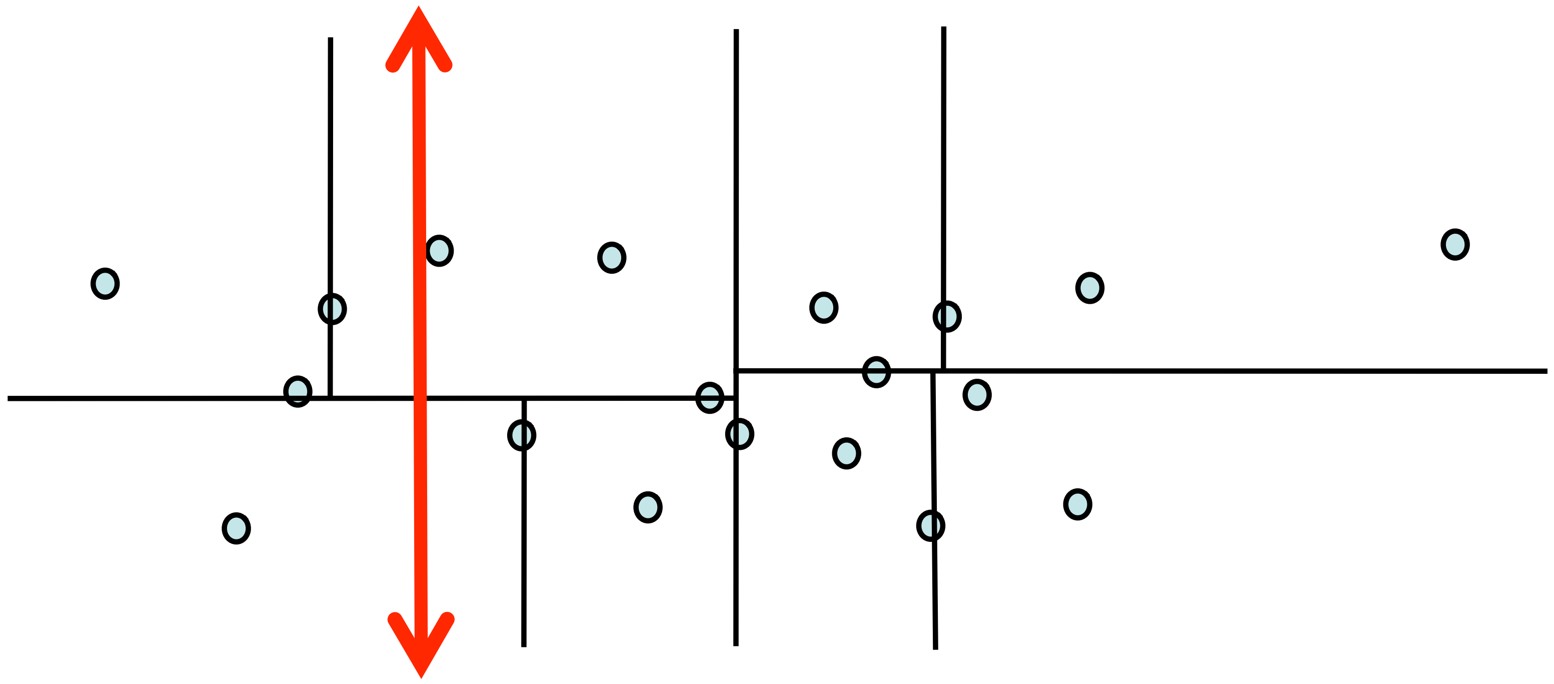
2D KD-Tree Query Time Analysis

- Query time = # of vertices visited + time to report K points
  - See the grey nodes below
- How many vertices will be visited?
  - How many regions intersecting the query range?
  - How many regions intersecting a orthogonal line?
    - which times 4 is be the upper bound of the grey nodes below



2D KD-Tree Query Time Analysis

- How many regions intersecting a orthogonal line?
  - T(n) = 1+T(n/2) = *log n*

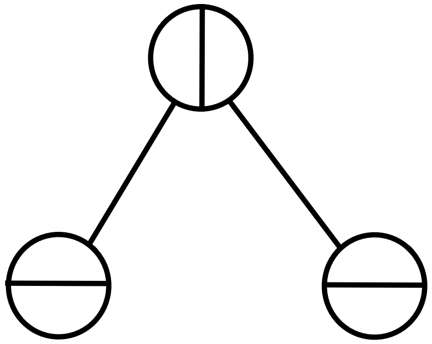


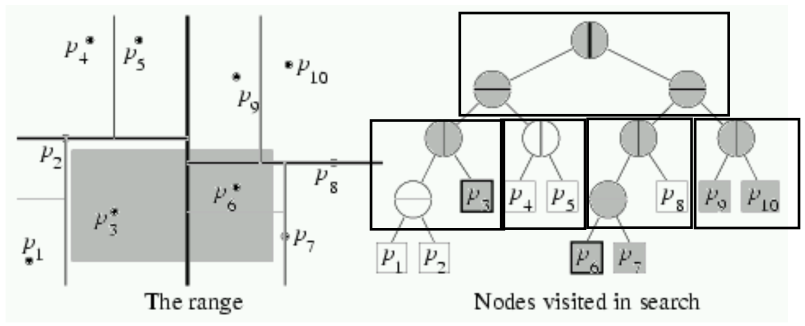
2D KD-Tree Query Time Analysis

- We count intersection in 3-sub-trees!

  - ❑ $Q(n) = O(1)$, if $n = 1$
  - ❑ $Q(n) = 2 + 2\,Q(n/4)$, if $n > 1$

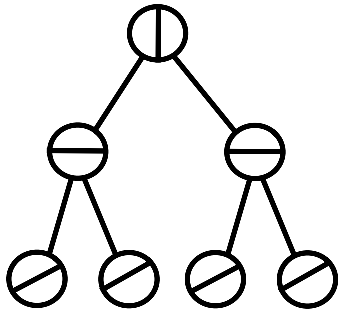
  $\Rightarrow Q(n) = O(n^{\log_4 2}) = O(n^{1/2})$





KD KD-Tree Query Time Analysis

- A d-dimensional kd-tree for a set of n points takes $O(d \cdot n)$ storage and $O(d \cdot n\log n)$ time to construct. The query time is bounded by $O(n^{1-1/d} + k)$.



- Let's try $d = 3$
- Extend to d-D
  - ❑ $Q(n) = 2^{d-1} + 2^{d-1}\, Q(n/2^d)$, if $n > 1$

$\Rightarrow Q(n) = O(n^{\log_{(2^\wedge d)} 2^\wedge(d-1)}) = O(n^{(\log_2 2^\wedge(d-1)/\log_2 2^\wedge(d))})$

$\Rightarrow Q(n) = O(n^{(d-1)/d})$

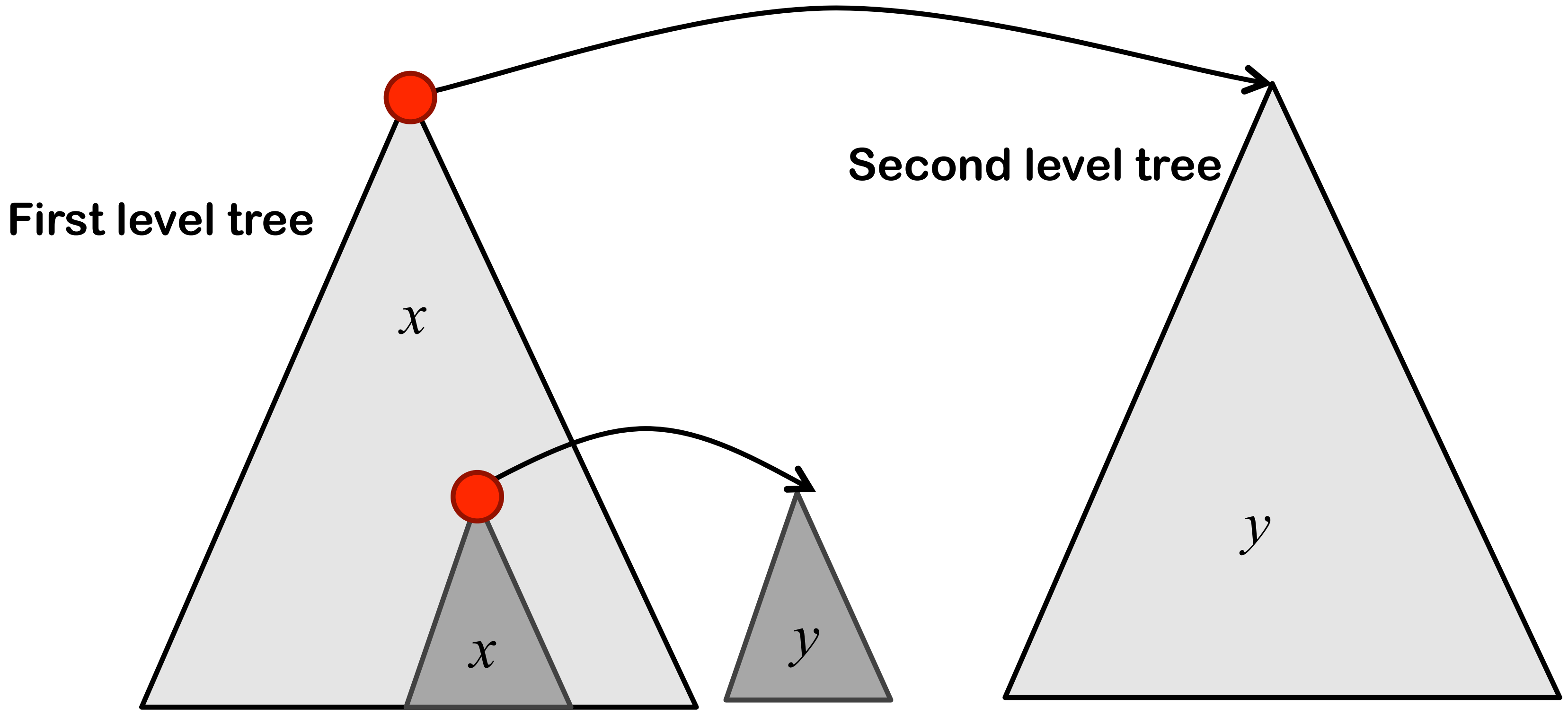
K-D-Range Search

- Extending binary search tree
  - K-D tree
    - Stack different dimensions in a tree
    - Require less memory
    - Slower
  - Range tree
    - Hierarchical structure of trees
    - Require more memory
    - Faster

Basics of Range Trees

- Range tree is more efficient but requires more space (store the same data in multiple copies!)
- 2D Range tree has two levels
  - First level is a 1D BST on x-axis (x-BST)
  - For each node v of x-BST, we build a 1D BST on y-axis for values in the sub-tree of v
    - *Canonical subset* of v

Basics of Range Trees



Build2DRangeTree(P)

Input: A set P of points in the plane.

Output: The root of 2-dimensional tree.

```
1. Build a binary search tree $T_{assoc}$ on the set $P_y$ of $y$-coordinates of the points
   in $P$. Store at leaves of $T_{assoc}$ the points themselves.
2. if $P$ contains only 1 point
3.    then Create a leaf $v$ storing this point and make $T_{assoc}$
            the associated structure of $v$.
4.    else Split $P$ into 2 subsets: $P_{left}$ containing points with
            $x$-coordinate ≤ $x_{mid}$, the median $x$-coordinate, and
            $P_{right}$ containing points with $x$-coordinate ≥ $x_{mid}$
5.          $v_{left} \leftarrow$ Build2DRangeTree($P_{left}$)
6.          $v_{right} \leftarrow$ Build2DRangeTree($P_{right}$)
7.          Create a node $v$ storing $x_{mid}$, make $v_{left}$ left child of $v$ & $v_{right}$
             right child of $v$, and make $T_{assoc}$ the associated structure of $v$
8. return $v$
```

Size of Range Trees

- Given a set of n points in 2D, the size of the range tree is:

$$\mathrm{Size}(n)=n+2\times\mathrm{Size}(n/2)$$

- Time of building the range tree:

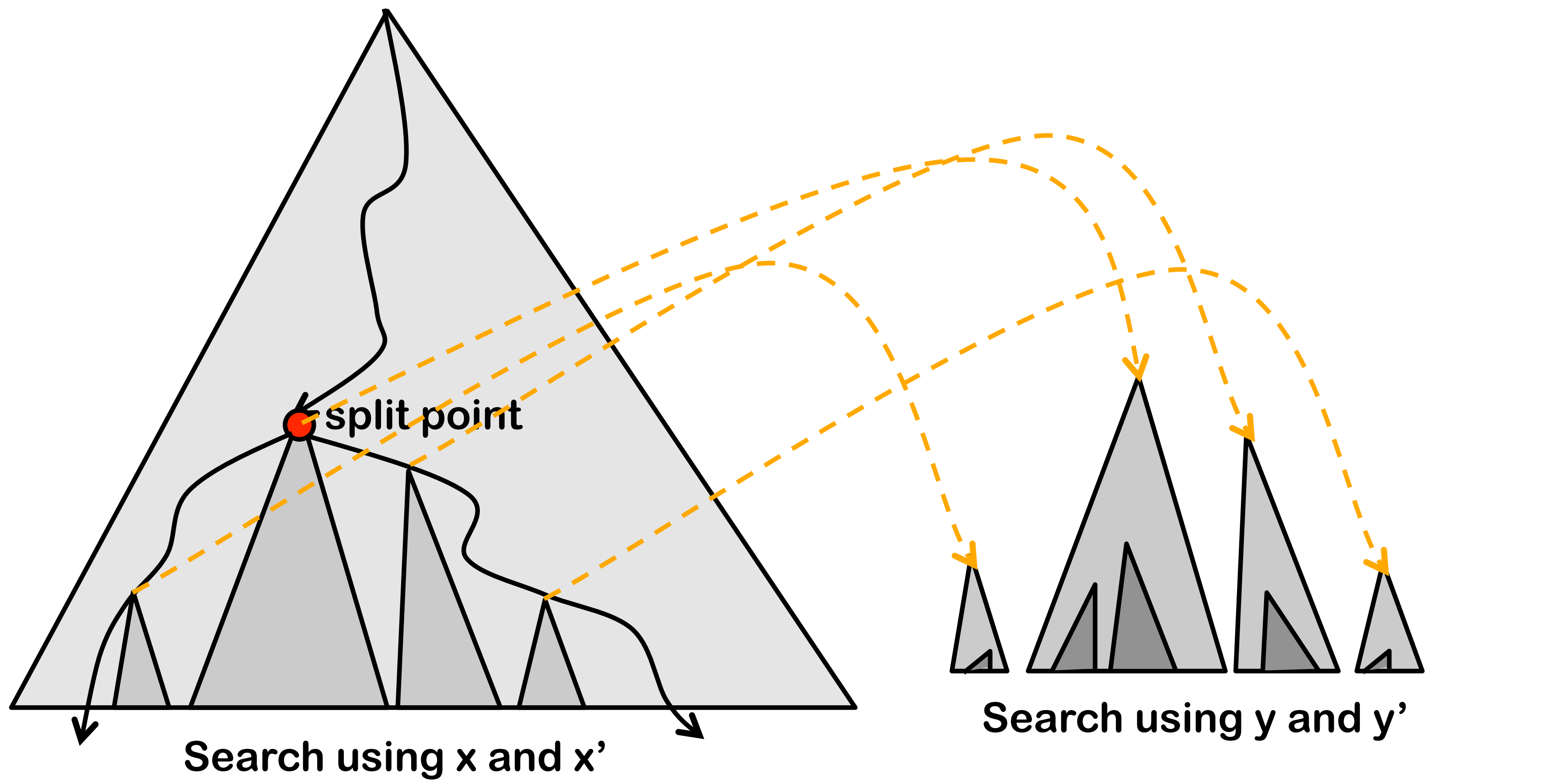
$$\mathrm{Time}(n)=O(n\log n)+2\times\mathrm{Time}(n/2) = O(n\log n)$$

Query Range Trees

- Similar to 1D, to report points in [*x*:*x'*] **x** [*y*:*y'*], we search with *x* and *x'* in *T*. Let *u* and *u'* be the two leaves where the search ends resp. Then the points in [*x*:*x'*] are the ones stored in leaves between *u* and *u'*, plus possibly points stored at *u* & *u'*.

- We can perform the 2D range query similarly by only visiting the associated binary search tree on *y*-coordinate of the canonical subset of *v*, whose *x*-coordinate lies in the *x*-interval of the query rectangle.

Query Range Trees



2DRangeQuery

Input: A 2D range tree T and a range $[x{:}x'] \times [y{:}y']$

Output: All points in T that lie in the range.

```
1.  $v_{split} \leftarrow$ FindSplitNode($T, x, x'$)
2.  if $v_{split}$ is a leaf
3.      then Check if the point stored at $v_{split}$ must be reported
4.      else (* Follow the path to $x$ and call 1DRangeQuery on
                the subtrees right of the path *)
•          $v \leftarrow lc(v_{split})$
6.          while $v$ is not a leaf
7.                do if $x \leq x_v$
8.                      then 1DRangeQuery($T_{assoc}(rc(v))$, $[y{:}y']$)
9.
10.                     else $v \leftarrow rc(v)$
11.         Check if the point stored at leaf $v$ must be reported
12.         Similar procedure for $x'$
```

$\rightarrow$ **move to left**

$\rightarrow$ **move to right**

Range Tree Query Algorithm Analysis

- 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.

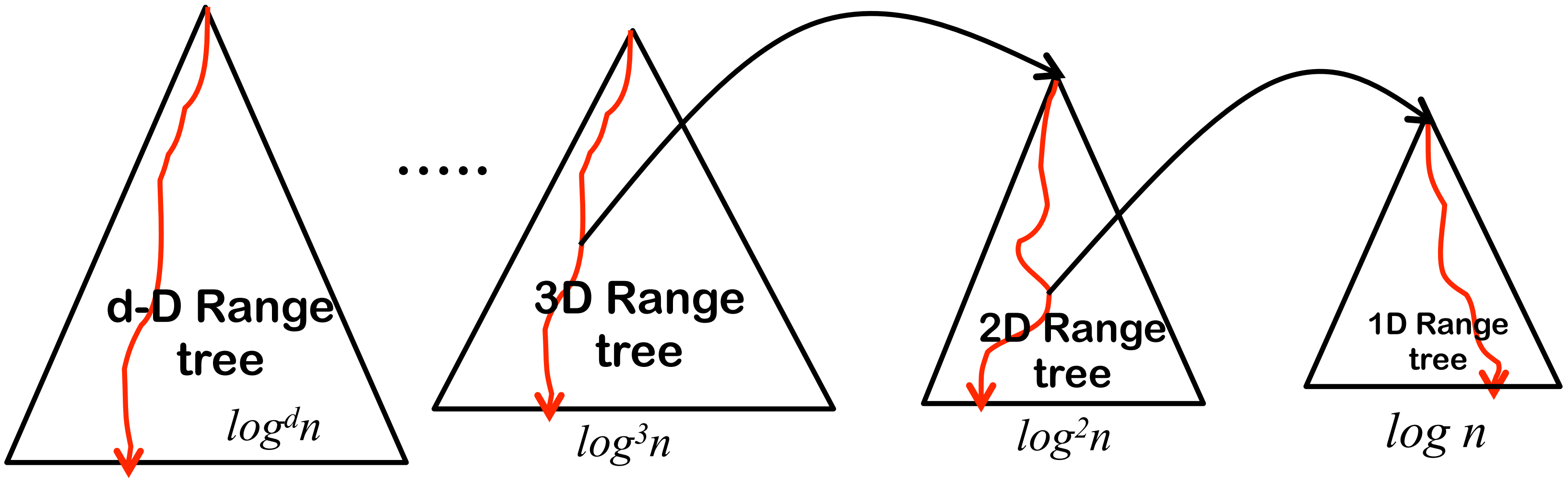
$$\text{Time}(n) = \text{Sum}_i(\text{ 1D BST of node } i\text{ })$$

$$< \text{Sum}_i(\ O(\log n)\ ) < (\log n)(\log n)$$

$$= O(\log^2 n)$$

Higher-D Range Trees

- Let P be a set of n points in d-dimensional space, where $d \geq 2$. A range tree for P uses $O(n\log^{d-1}n)$ storage and it can be constructed in $O(n\log^{d-1}n)$ time. One can report the points in P that lies in a rectangular query range in $O(\log^d n + k)$ time, where k is the number of reported points.



Range Trees

- Degenerate Cases
  - Same x or same y coordinate
  - This can also apply to K-d tree

- Can we do better than $O(\log^2 n + k)$?
  - Fractional Cascading

General Sets of Points

- Replace the coordinates, which are real numbers, by elements of the so-called *composite-number space*. The elements of this space are pair of reals, denoted by (a|b). We denote the order by:

$$(a|b) < (a'|b') \Leftrightarrow a < a' \text{ or } (a=a' \text{ and } b<b')$$

- Many points are distinct. But for the ones with same x- or y-coordinate, replace each point

$$p := (p_x, p_y) \text{ by } p' := ((p_x| p_y), (p_y| p_x))$$

- Replace $R := [x:x'] \times [y:y']$ by

$$R' := [(x| -\infty) : (x'| +\infty)] \times [(y| -\infty) : (y'| +\infty)]$$

General Sets of Points

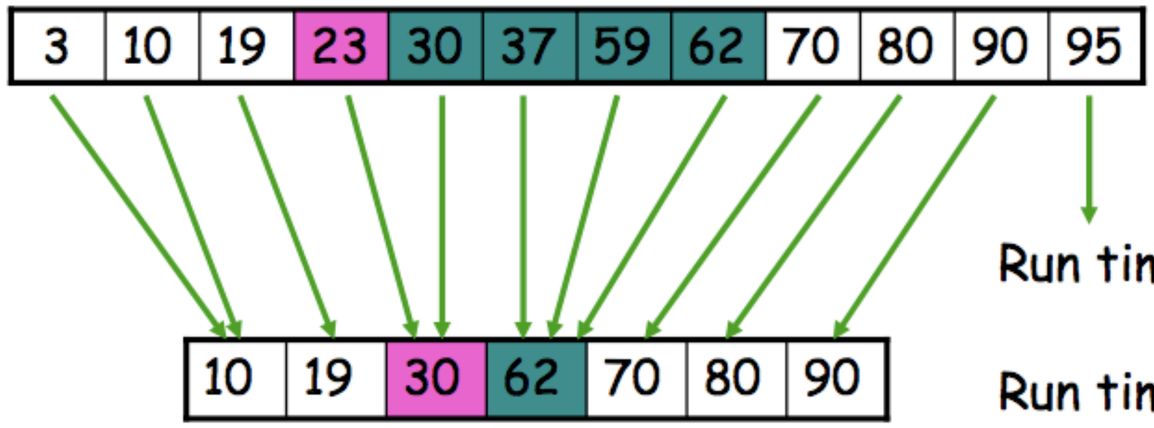
- Points (5,10) (5,21) in the range [4:8]×[2:50]
  - Convert points to (5|10,10|5), (5|21,21|5)
  - Convert range to [4| -∞ :8| +∞]×[2| -∞ :50|+∞]
  - Is the converted points in the converted range?

- Points (9,51) (4,51) NOT in the range [4:8×2:50]
  - Convert points to (9|51,51|9), (4|51,51|4)
  - Convert range to [4| -∞ :8| +∞]×[2| -∞ :50|+∞]
  - Is the converted points in the converted range?

Search in Subsets

- Given: Two ordered arrays A1 and A2. key(A2) ⊂ key(A1), query $[x, x']$
- Search: All elements e in A1 and A2 with x<key(e)<x'.
- Idea: pointers between A1 and A2
  - Each element in A1 points to the smallest larger element in A2

Example query : [20 : 65]

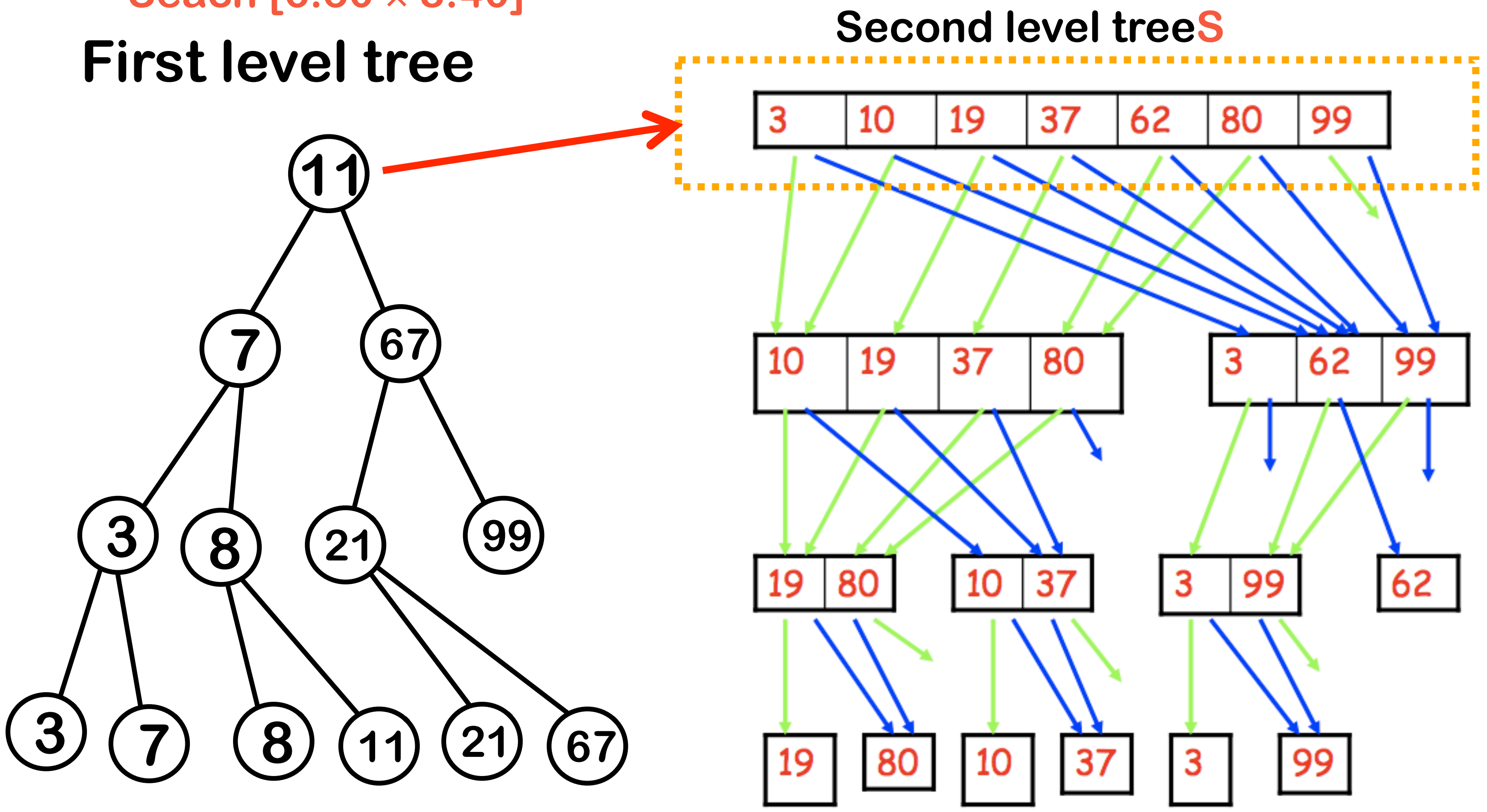


Run time : O(log n + k)

Run time : O(1 + k)

Fractional Cascading

Example: (7,80) (67,99) (3,19) (11,37) (21,3) (8,10) (99,62)
Seach [6:30 × 8:40]



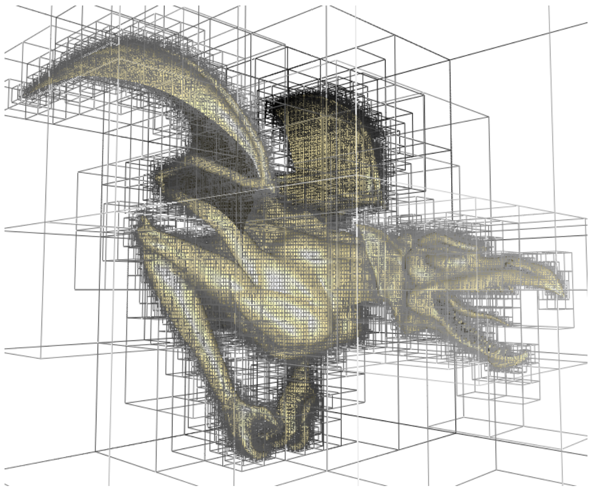
Fractional Cascading Algorithm Analysis

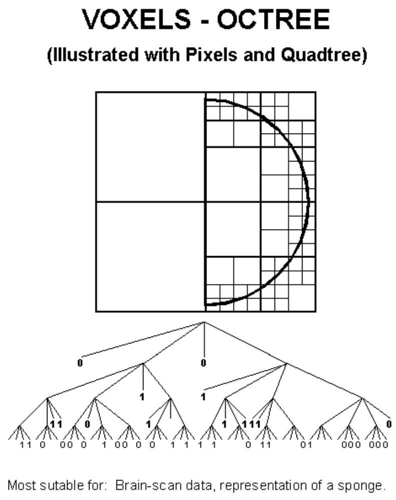
- Time complexity
  - Each node will take constant time to find values to report
  - O(log(n)+k)

- Yeah, we get rid of one "log"!! Same for d-D range Tree
  - Let P be a set of n points in d-dimensional space, with $d \geq 2$. A layered range tree for P uses $O(n\log^{d-1}n)$ storage and it can be constructed in $O(n\log^{d-1}n)$ time. With this range tree, one can report the points in P that lies in a rectangular query range in $O(\log^{d-1}n + k)$ time, where k is the number of reported points.

More Trees

- Binary space partitioning
- Quad/Oct-tree (Chapter 14, a very simple data structure)
  - Compute distance field
  - Motion planning
  - Texturing
  - Fill holes
  - …..many many more

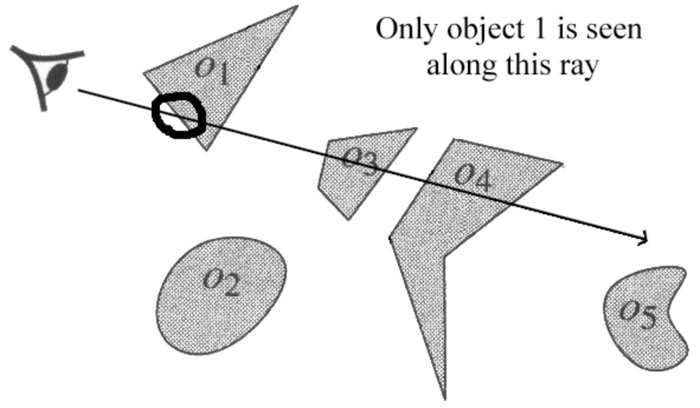



VOXELS - OCTREE
(Illustrated with Pixels and Quadtree)
Most sutable for: Brain-scan data, representation of a sponge.

Drawing the Visible Objects

We want to generate the image that the eye would see, given the objects in our space

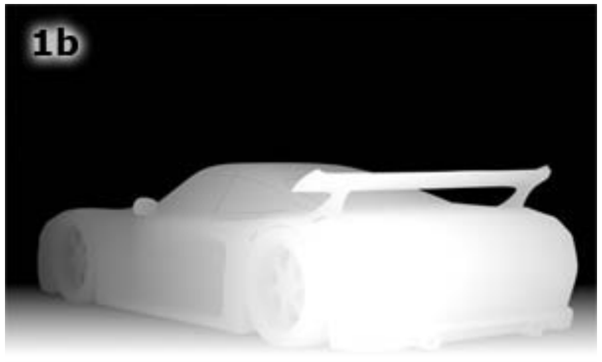
How do we draw the correct object at each pixel, given that some objects may obscure others in the scene?



Hidden surface removal

A Simple Solution:

- Keep a buffer that holds the z-depth of the pixel currently at each point on screen
- Draw each polygon: for each pixel, test its depth versus current screen depth to decide if we draw it or not



Z-buffer

Drawbacks to Z-buffering

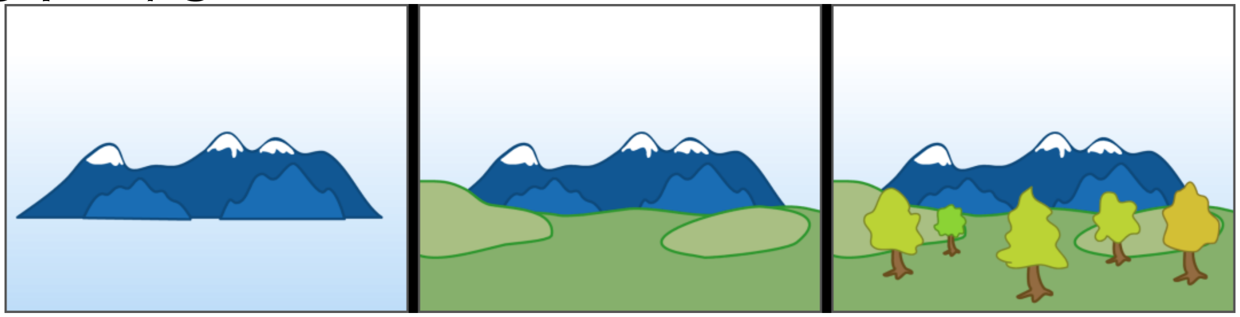
This used to be a **very expensive** solution!

- Requires memory for the z-buffer
  - extra hardware cost was prohibitive
- Requires extra z-test for every pixel

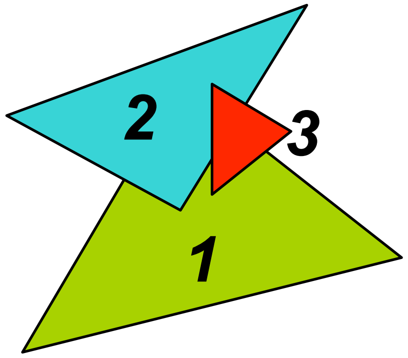
So, a software solution was developed …

The Painter's Algorithm

Avoid extra z-test & space costs by scan converting polygons in back-to-front order



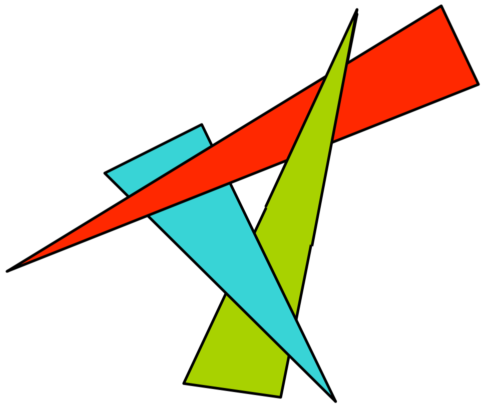
From wikipedia



Is there always a correct back-to-front order?

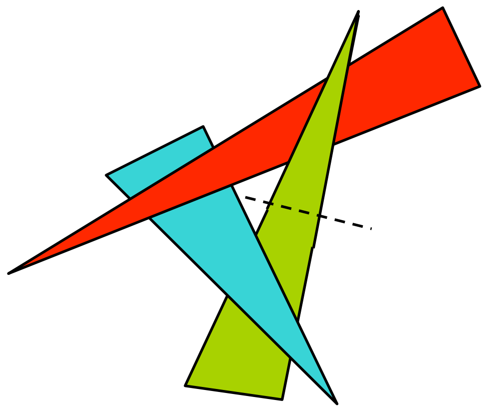
How Do We Deal With Cycles?

In 3 dimensions, polygons can overlap, creating cycles in which no depth ordering would draw correctly



How do we deal with these cases?

How Do We Deal With Cycles?



We can break one polygon into two and order them separately

- Which polygon?
- Where?

View dependent

BSP Trees

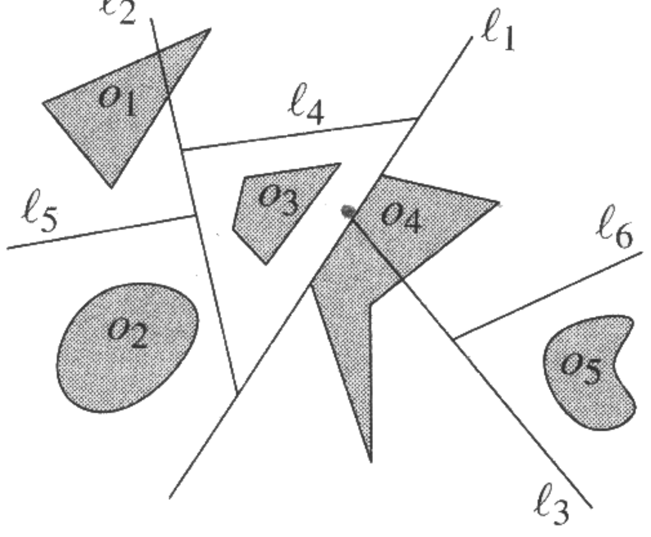
Having a pre-built BSP tree will allow us to get a correct depth order of polygons in our scene for any point in space.

We will build a data structure based on the polygons in our scene, that can be queried with any point input to return an ordering of those polygons.

The Big Picture

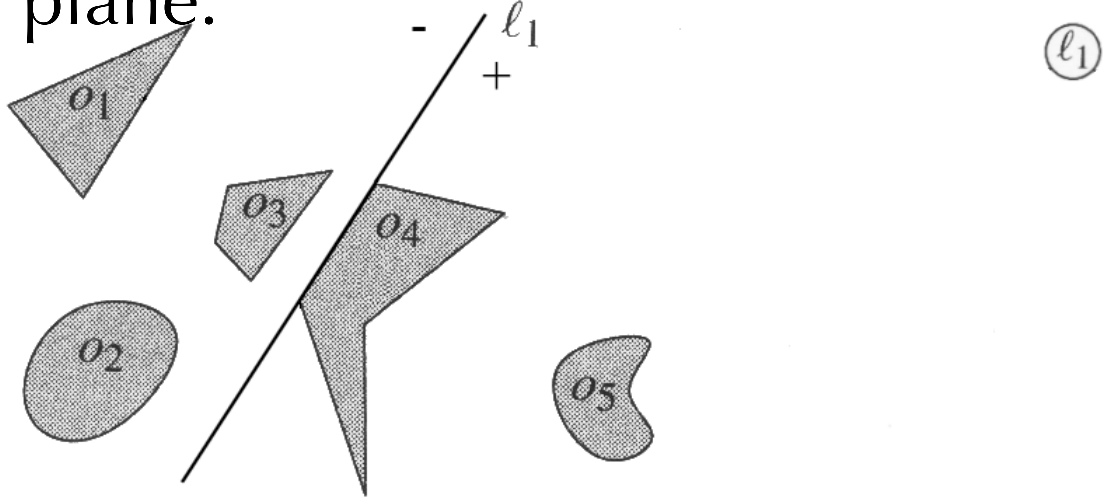
Assume that no objects in our space overlap

Use planes to recursively split our object space, keeping a tree structure of these recursive splits.



Choose a Splitting Line

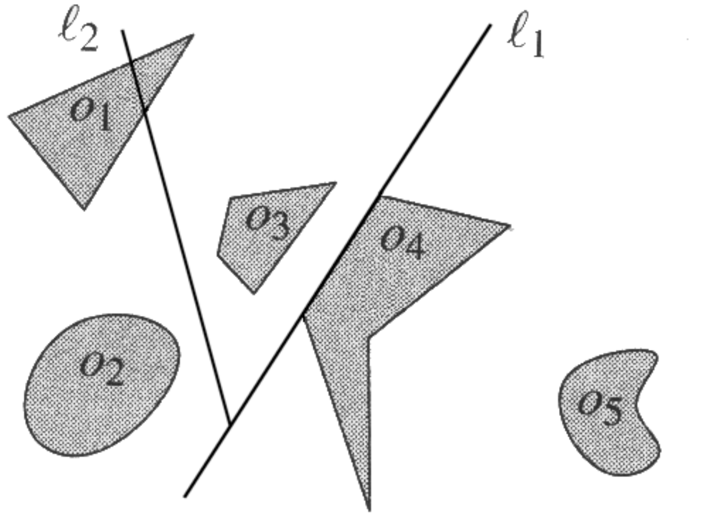
Choose a splitting plane, dividing our objects into three sets – those on each side of the plane, and those fully contained on the plane.

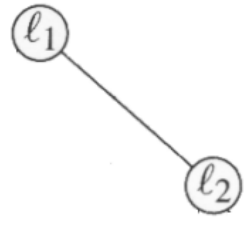


Choose More Splitting Lines

What do we do when an object (like object 1) is divided by a splitting plane?

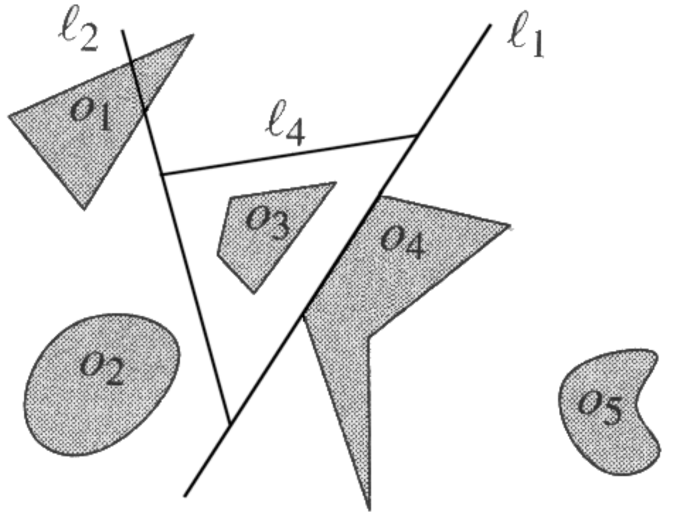
It is divided into two objects, one on each side of the plane.

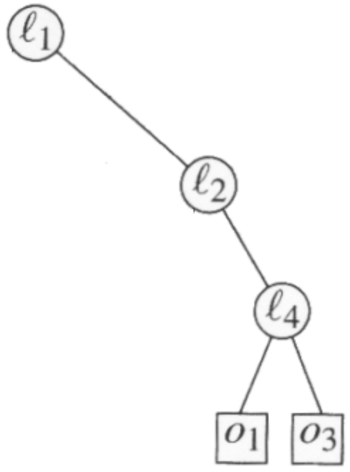




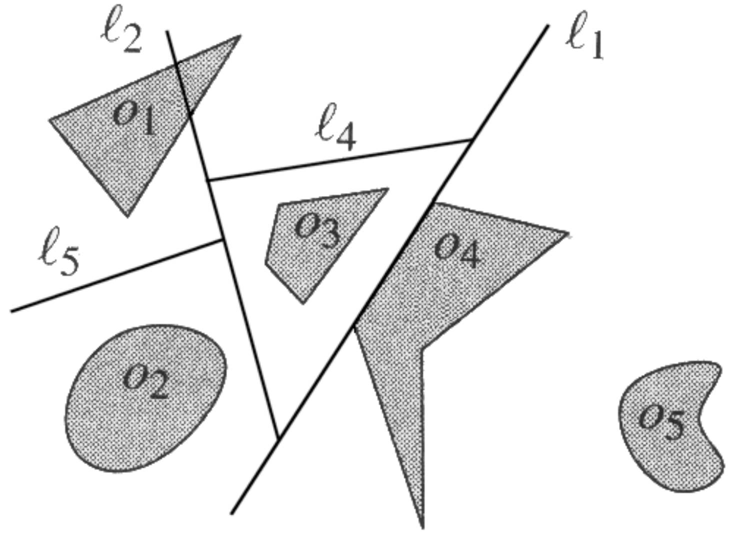
Split Recursively Until Done

When we reach a convex space containing exactly zero or one objects, that is a leaf node.

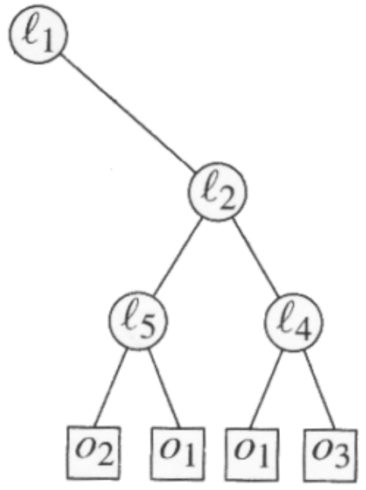




Continue

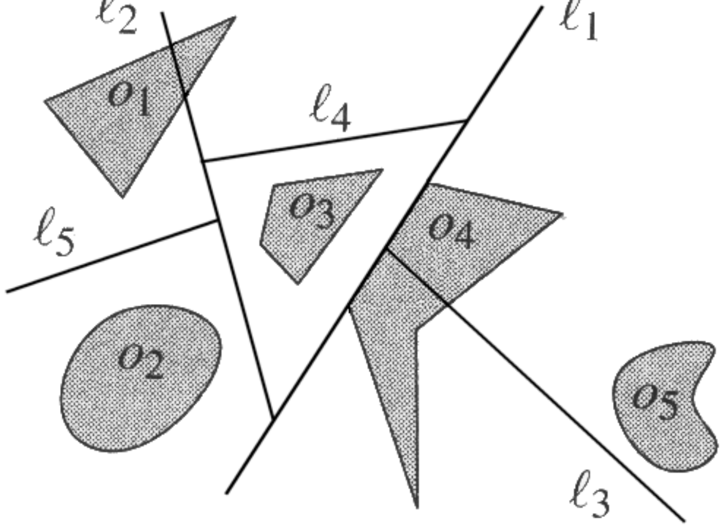


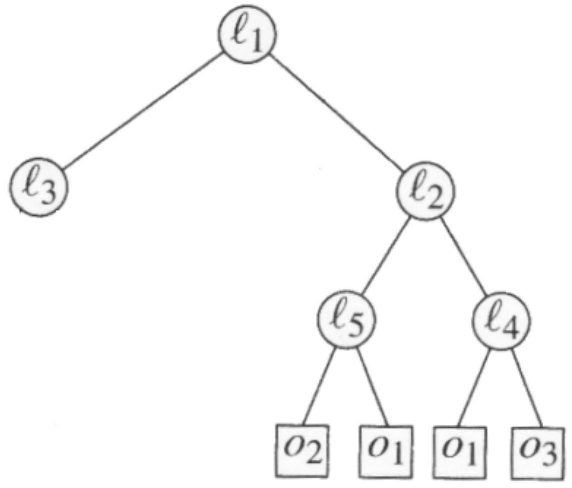
ℓ_2
ℓ_1
o_1
ℓ_4
ℓ_5
o_3
o_4
o_2
o_5



ℓ_1
ℓ_2
ℓ_5
ℓ_4
o_2
o_1
o_1
o_3

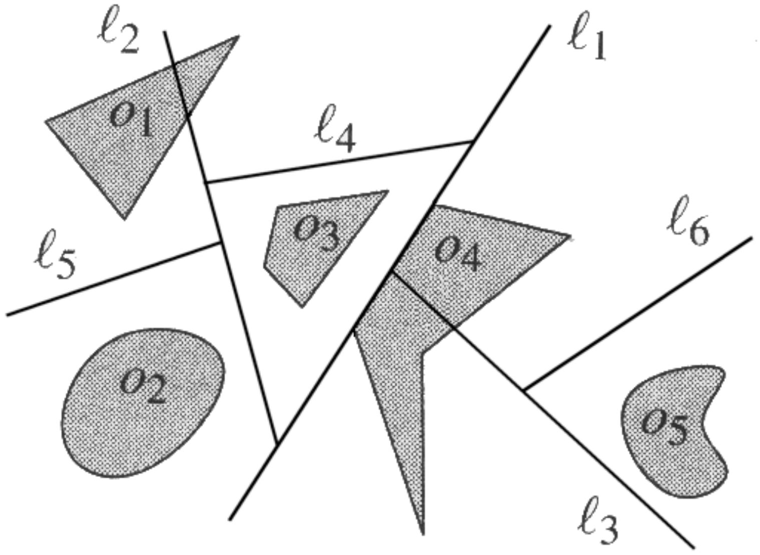
Continue

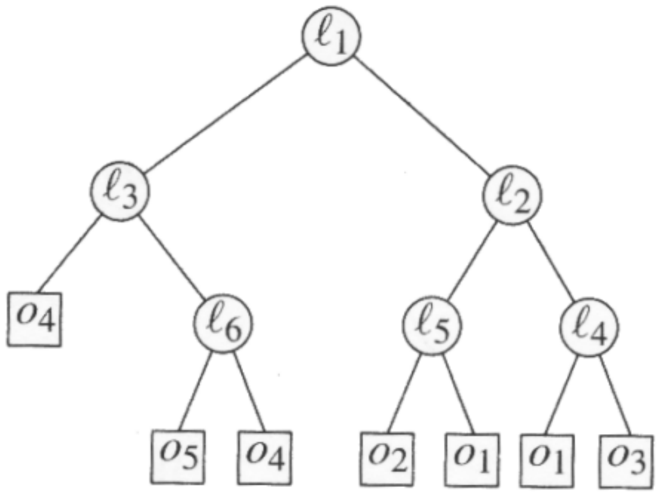

ℓ_2
ℓ_1
o_1
ℓ_4
o_3
o_4
ℓ_5
o_2
o_5
ℓ_3


ℓ_1
ℓ_3
ℓ_2
ℓ_5
ℓ_4
o_2
o_1
o_1
o_3

Finished

Once the tree is constructed, every root-to-leaf path describes a single convex subspace.



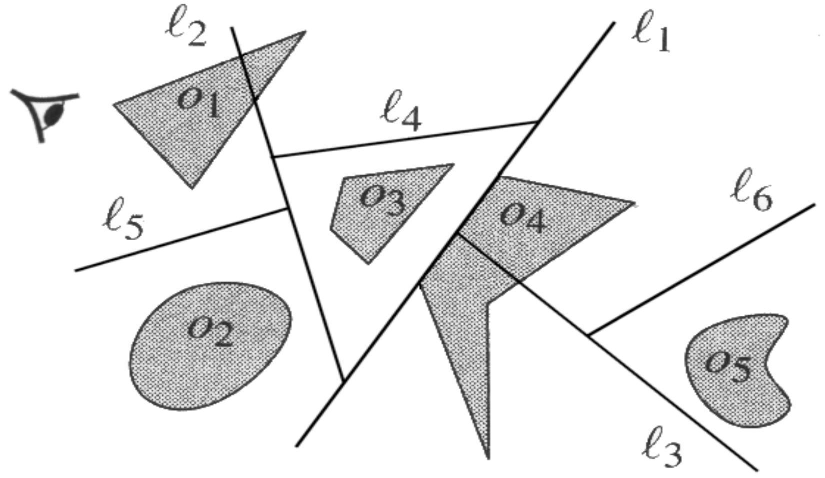


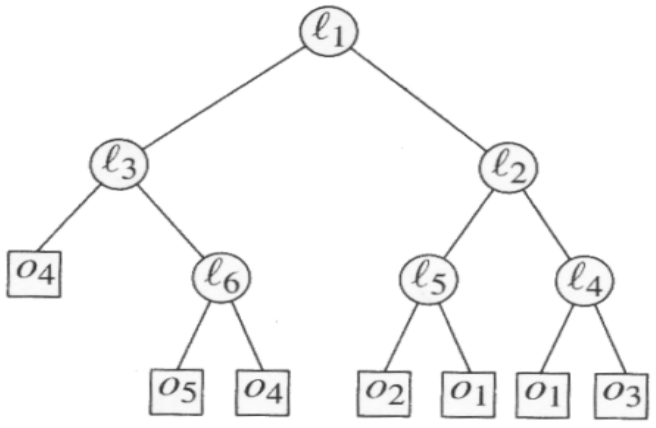
Querying the Tree

If a point is in the positive half-space of a plane,

- everything in the negative half-space is farther away -- so draw it first, using this algorithm recursively
- draw objects on the splitting plane
- draw objects into the positive half-space, recursively

What Order Is Generated From This Eye Point?





How much time does it take to query the BSP tree, asymptotically?

Structure of a BSP Tree

- Each internal node has a +half space, a -half space, and a list of objects contained entirely within that plane (if any exist).
- Each leaf has a list of zero or one objects inside it, and no subtrees
- The *size* of a BSP tree is the total number of objects stored in the leaves & nodes of the tree
  - This can be larger than the number of objects in our scene because of splitting

K-d tree and Octree are special cases of BSP

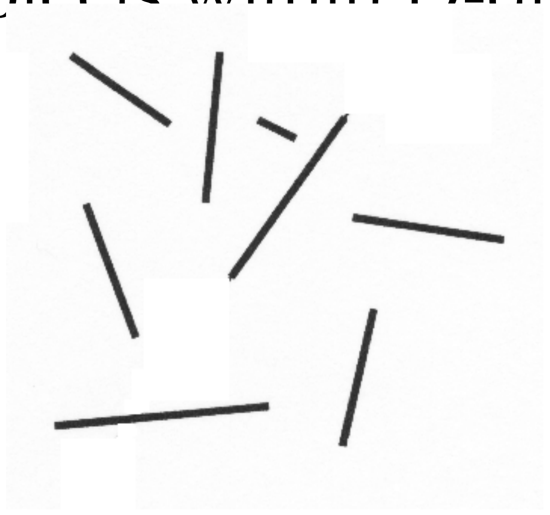
Building a BSP Tree

- How do we pick splitting lines/planes?

Building a BSP Tree From Line Segments in the Plane

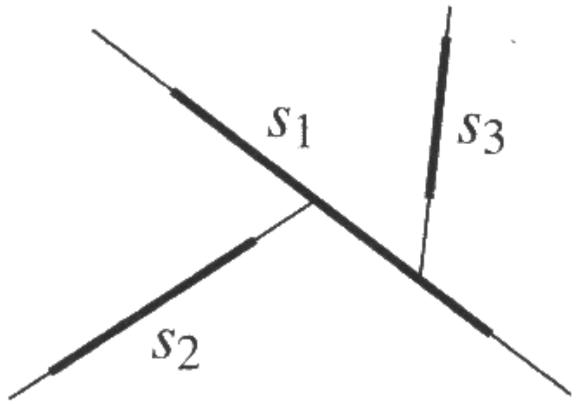
We'll now deal with a formal algorithm for building a BSP tree for line segments in the 2D plane.

This will generalize for building trees of D-1 dimensional objects within D-dimensional spaces.



Auto-Partitioning

Auto-partitioning: splitting only along planes coincident on objects in our space

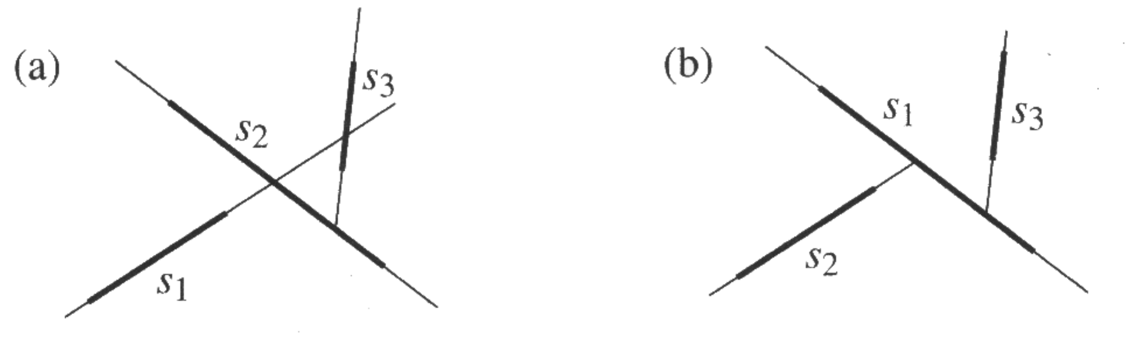


Algorithm for the 2d Case

- If we only have one segment 's', return a leaf node containing s.
- Otherwise, choose a line segment 's' along which to split
- For all other line segments, one of four cases will be true:
  1) The segment is in the +half space of s
  2) The segment is in the -half space of s
  3) The segment crosses the line through s
  4) The segment is entirely contained within the line through s
- Split all segments who cross 's' into two new segments -- one in the +half space, and one in the -half space
- Create a new BSP tree from the set of segments in the +half space of s, and another on the set of segments in the -half space
- Return a new node whose children are these +/- half space BSP’s, and which contains the list of segments entirely contained along the line through s.

How Small Is the BSP

- Different orderings result in different trees



- Greedy approach?
  - pick the line with fewest intersections
  - doesn't always work -- sometimes it does very badly
  - it is costly to find

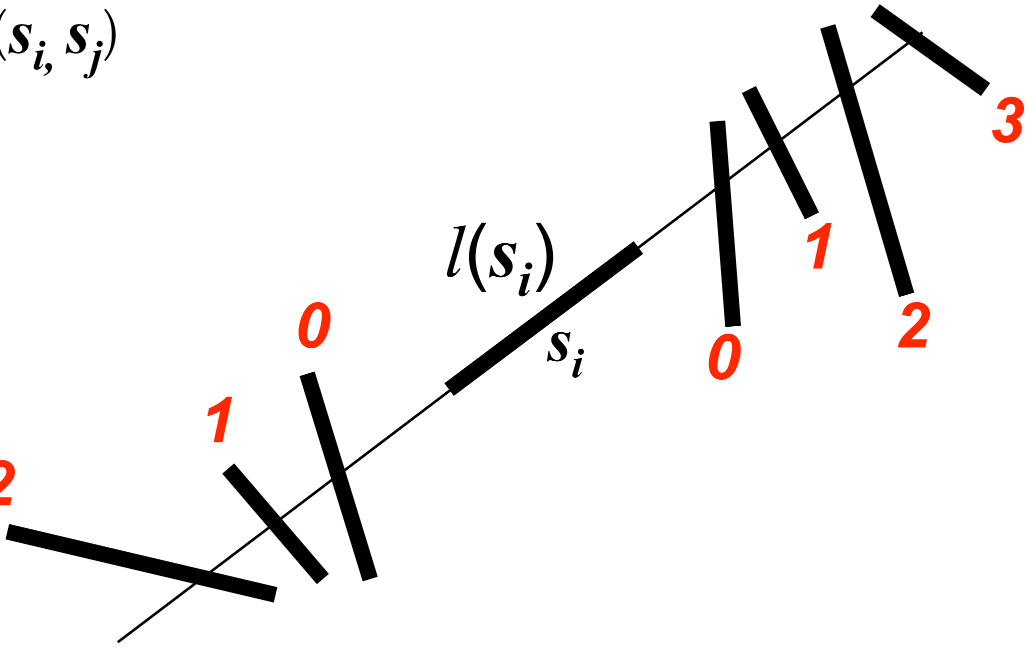
Random Approach Works Well

If we randomly order segments before building the tree, then we do well in the average case

- Expected number of fragments, i.e., size of leaves $O(n \log n)$
- Expected running time: $O(n^2 \log n)$

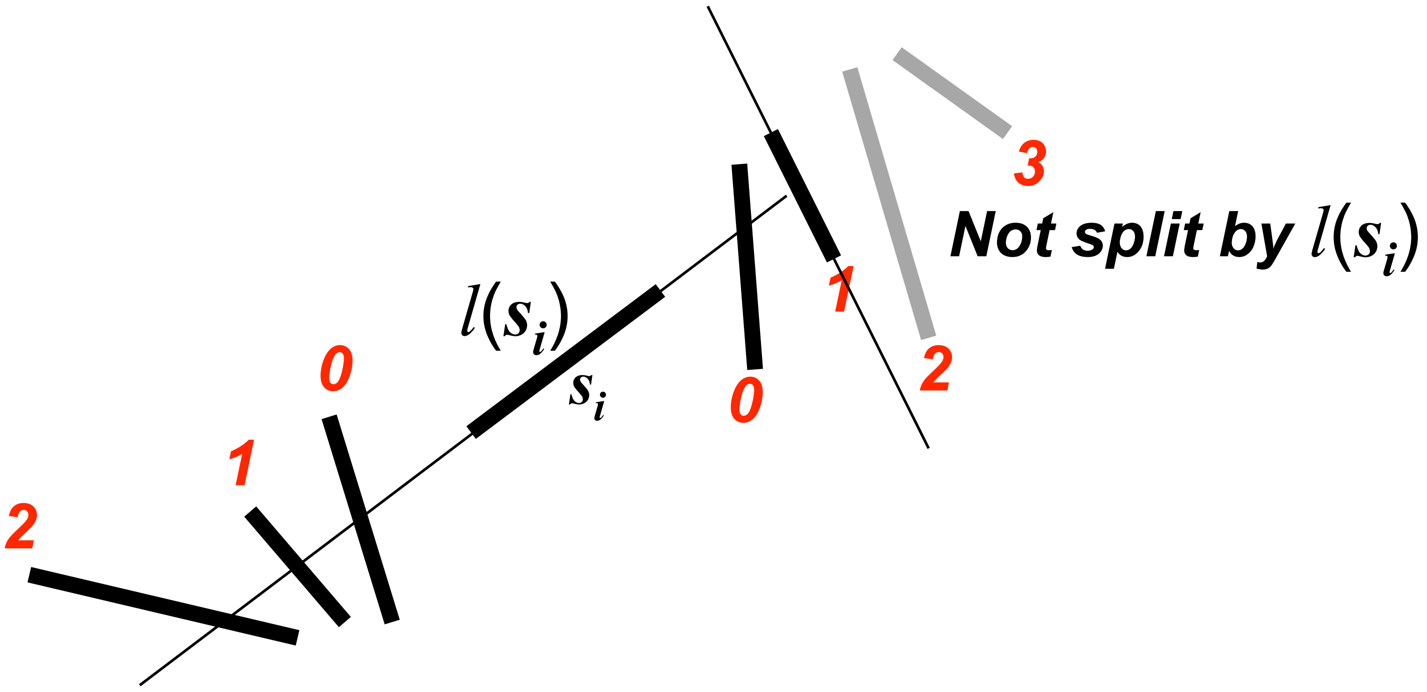
Expected Number of Fragments

- dist($s_{i,}$ s_j)
  - number of segments between $s_{i,}$ and s_j
  - The probability that s_j is split by $l(s_i)$ depends on dist($s_{i,}$ s_j)



Expected Number of Fragments

$l(s_i)$ splits s_j if no segment between s_i and s_j is selected as a splitting line before s_i



Expected Number of Fragments

$\therefore$ s_i needs to be the first to be selected among the (dist(s_i, s_j)+2) segments to cut s_j

$\Rightarrow$Pr[$l(s_i)$ cuts s_j]$\leq$ 1/(dist(s_i, s_j)+2)

$\therefore$ number of segments generated by s_i

$\leq$ sum$_j$(1/(dist(s_i, s_j)+2))

$\leq$ 2$\times$sum$_k$(1/(k+2))

$\leq$2 ln n = O(log n)

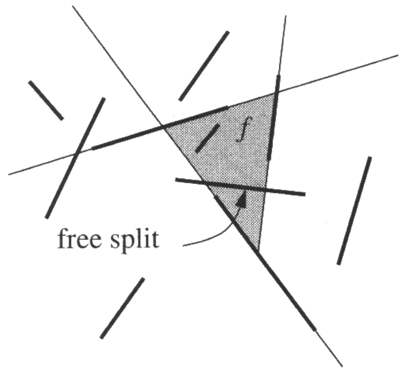
$\therefore$ number of fragments is O(n log n)

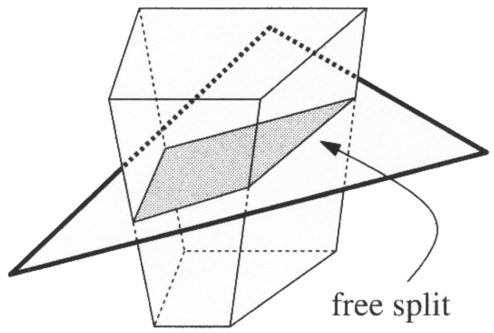
Expected Running Time

- Each split will take O(n) or shorter time
- There can be O($n \log n$) splits
- Total time to build a BSP: O($n^2 \log n$)

Optimization: Free Splits

- Sometimes a segment will entirely cross a convex region described by some interior node -- if we choose that segment to split on next, we get a "free split" -- no splitting needs to occur on the other segments since it crosses the region



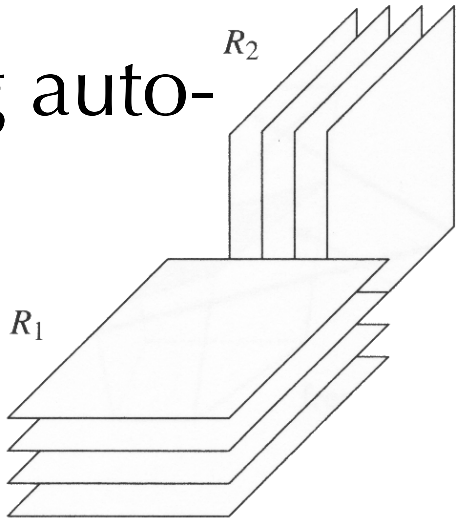


Our Building Algorithm Extends to 3D!

The same procedure we used to build a tree from two-dimensional objects can be used in the 3D case – we merely use polygonal faces as splitting planes, rather than line segments.

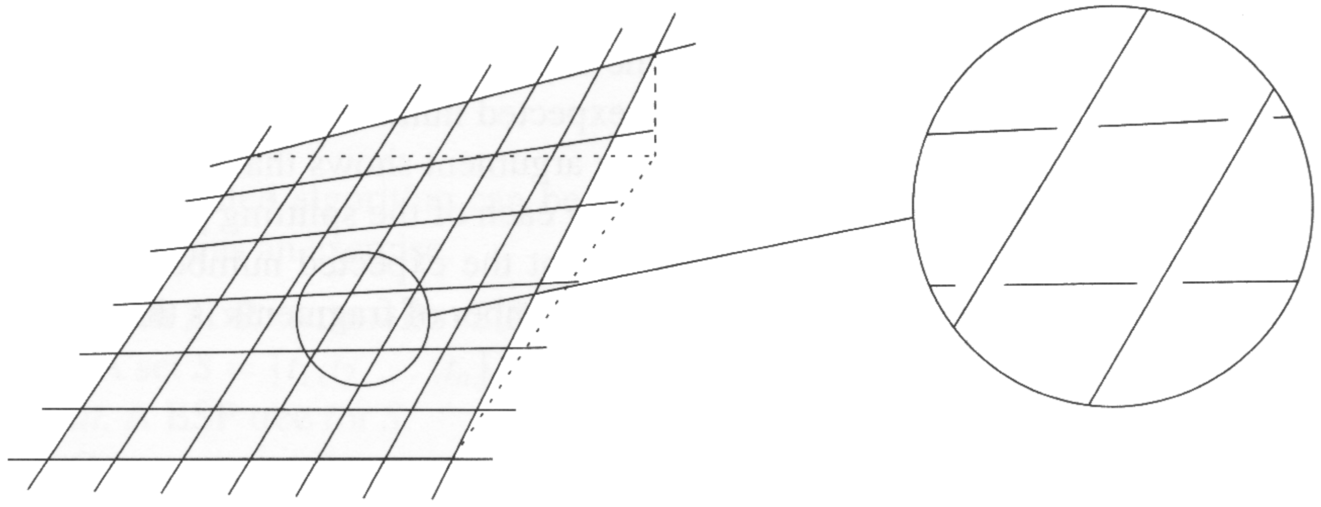
More analysis: How good are auto-partitions?

- There are sets of triangles in 3-space for which auto-partitions guarantee $\Omega(n^2)$ fragments
- Can we do better by not using auto-partitions?



Sometimes, No

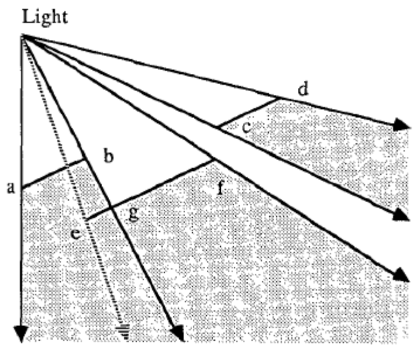
There are actually sets of triangles for which *any* BSP will have $\Omega(n^2)$



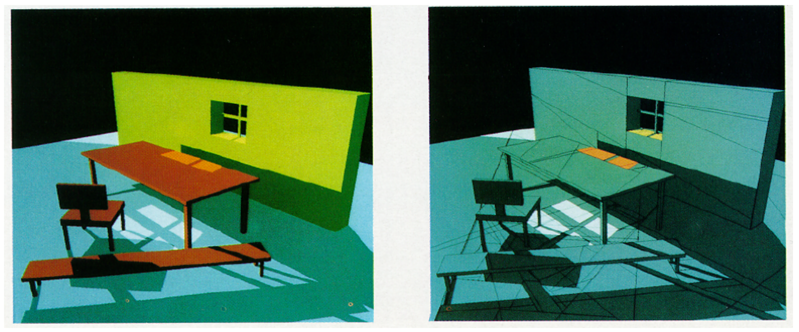
Fortunately, these cases are artificial and rarely seen… BSP usually performs well in practice

More applications

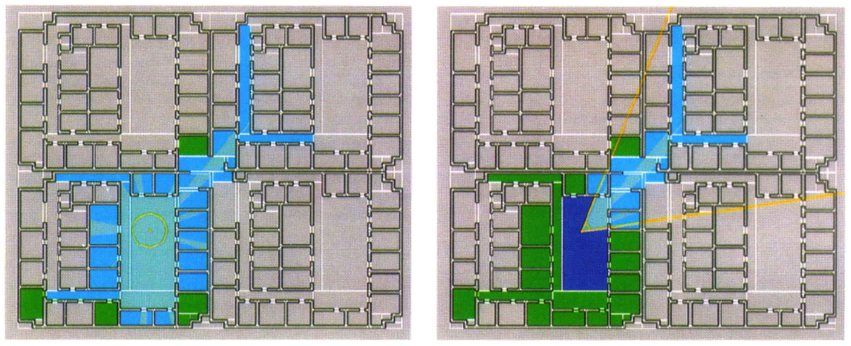
- Querying a point to find out what convex space it exists within



- Shadows



- Visibility



- Blending
  - Opengl stuff

Conclusion

- Range search
  - K-D tree
    - Interleave dimensions
  - Range tree
    - Hierarchical dimensions
    - Fractional cascade
- Other popular trees
  - Binary space partitioning
  - Quad/Oct-tree