CS425 GAME PROGRAMMING

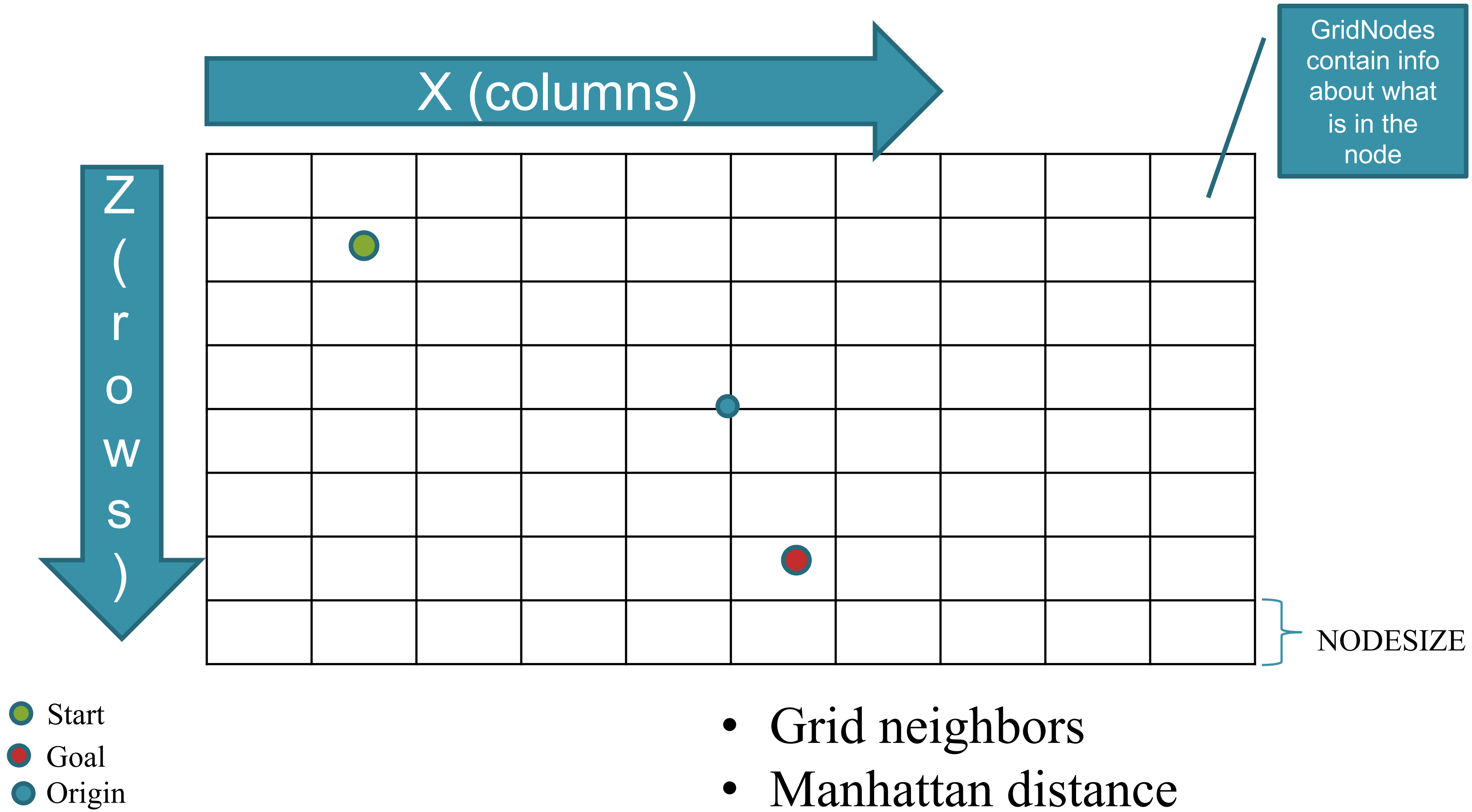
Scene Management

Scene Management

- Collision checking (in particular for broad range query)
- Visibility checking
- Proximity query (distance between two objects)
- Nearest Neighbors Query

- What does GSM (Generic Scene Manager) do?

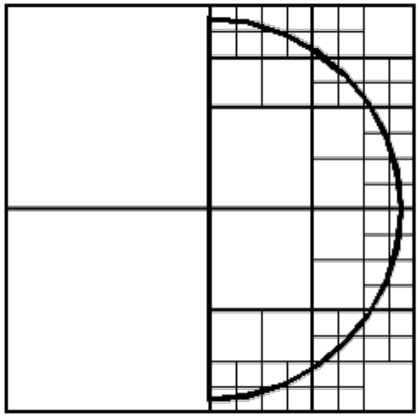
Regular Grid

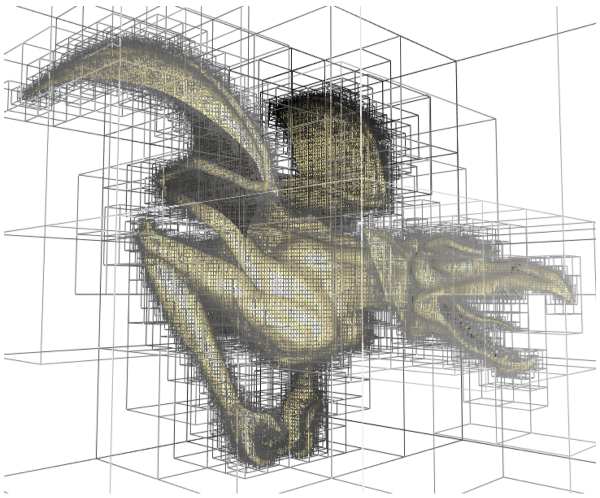


- Grid neighbors
- Manhattan distance

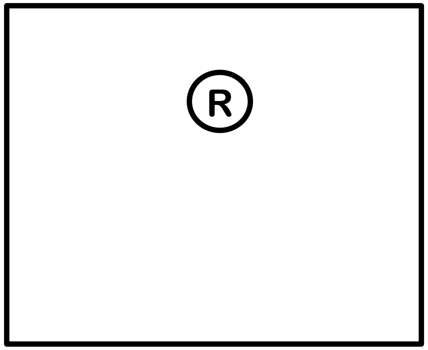
Trees and Adaptive

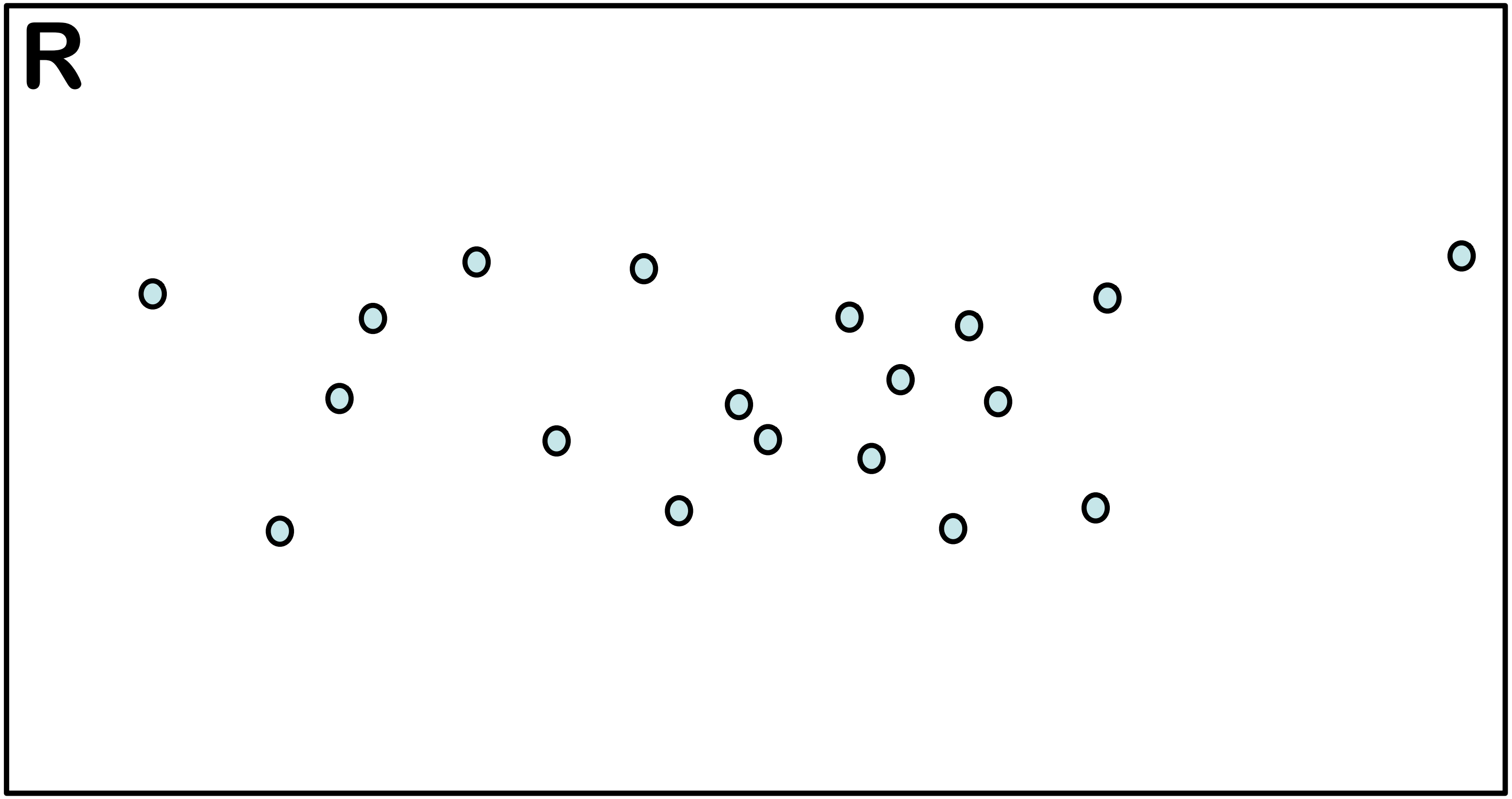
- Quad/Oct-tree
  - Extensions of binary search trees
  - Adaptive subdivision of cells
  - Termination conditions may vary
    - Depth
    - Cell size
    - Contents in the cell
- Applications
  - Compute distance field
  - Motion planning
  - Texturing
  - Fill holes
  - …..many many more



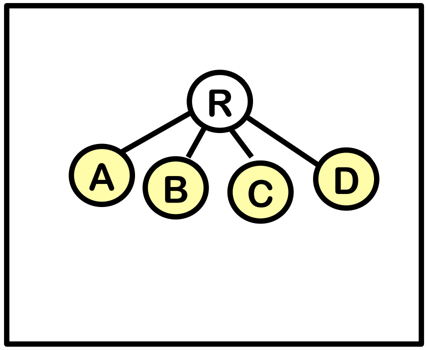


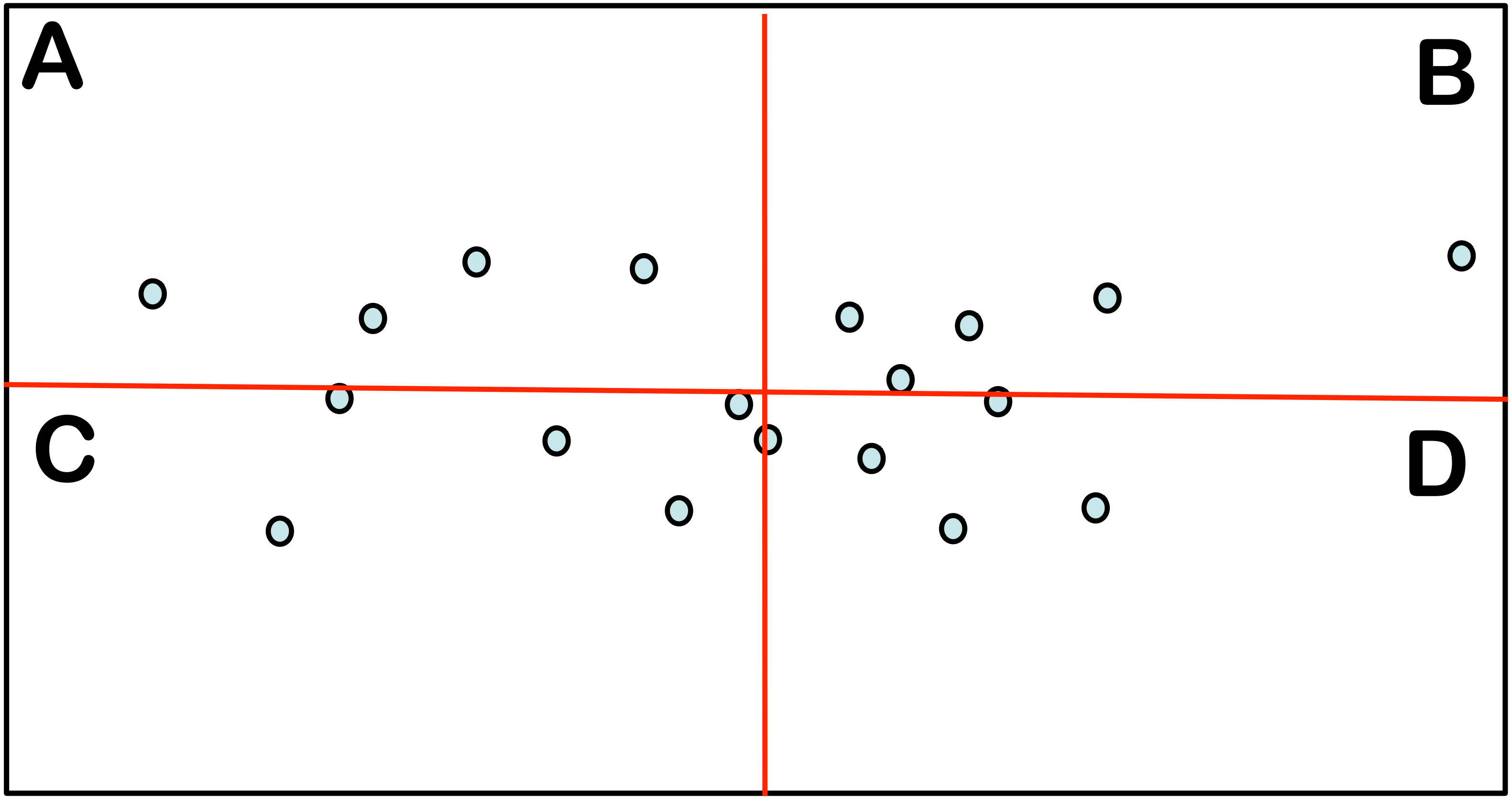
Quad Trees





Quad Trees





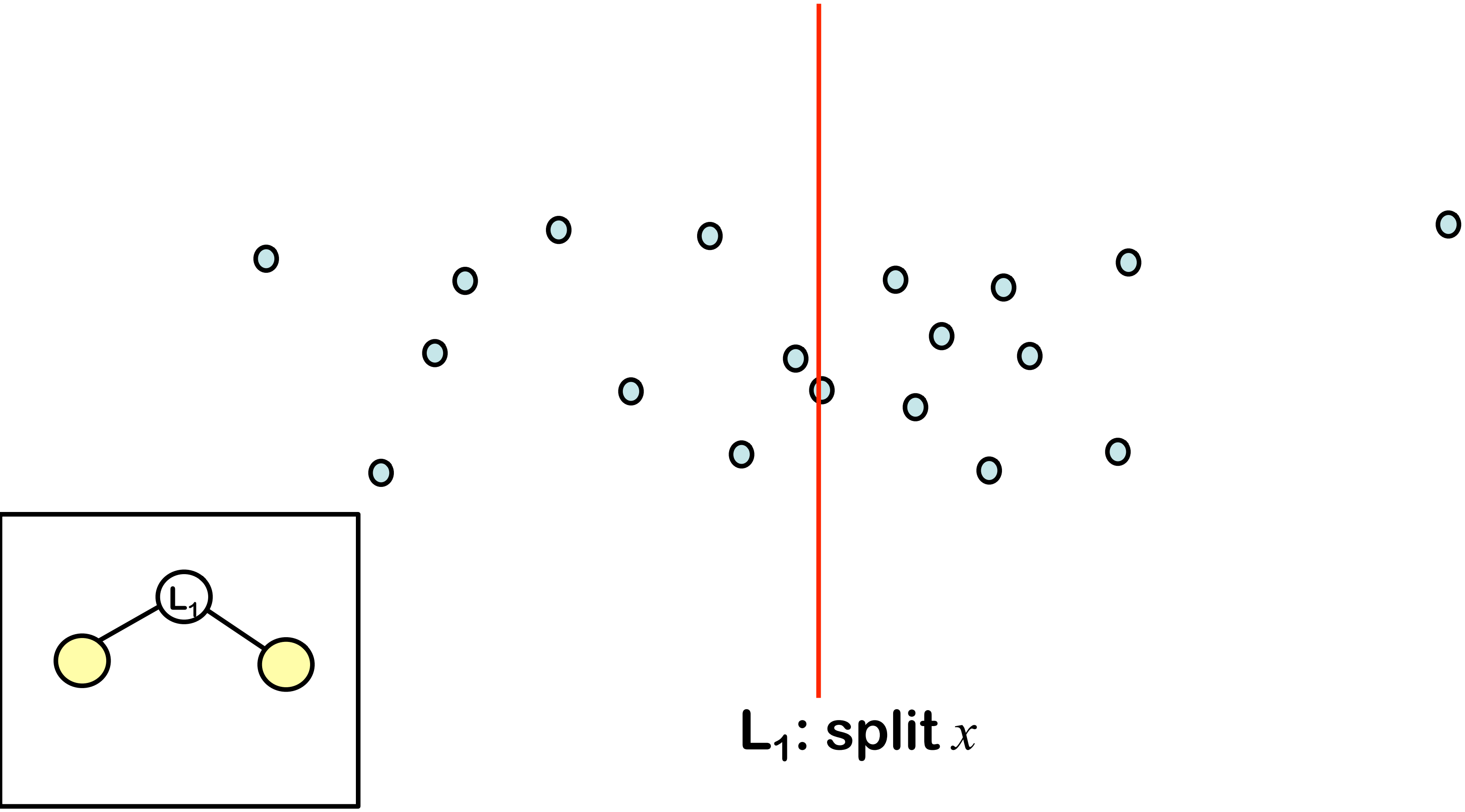
Range Search Data Structures

- 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

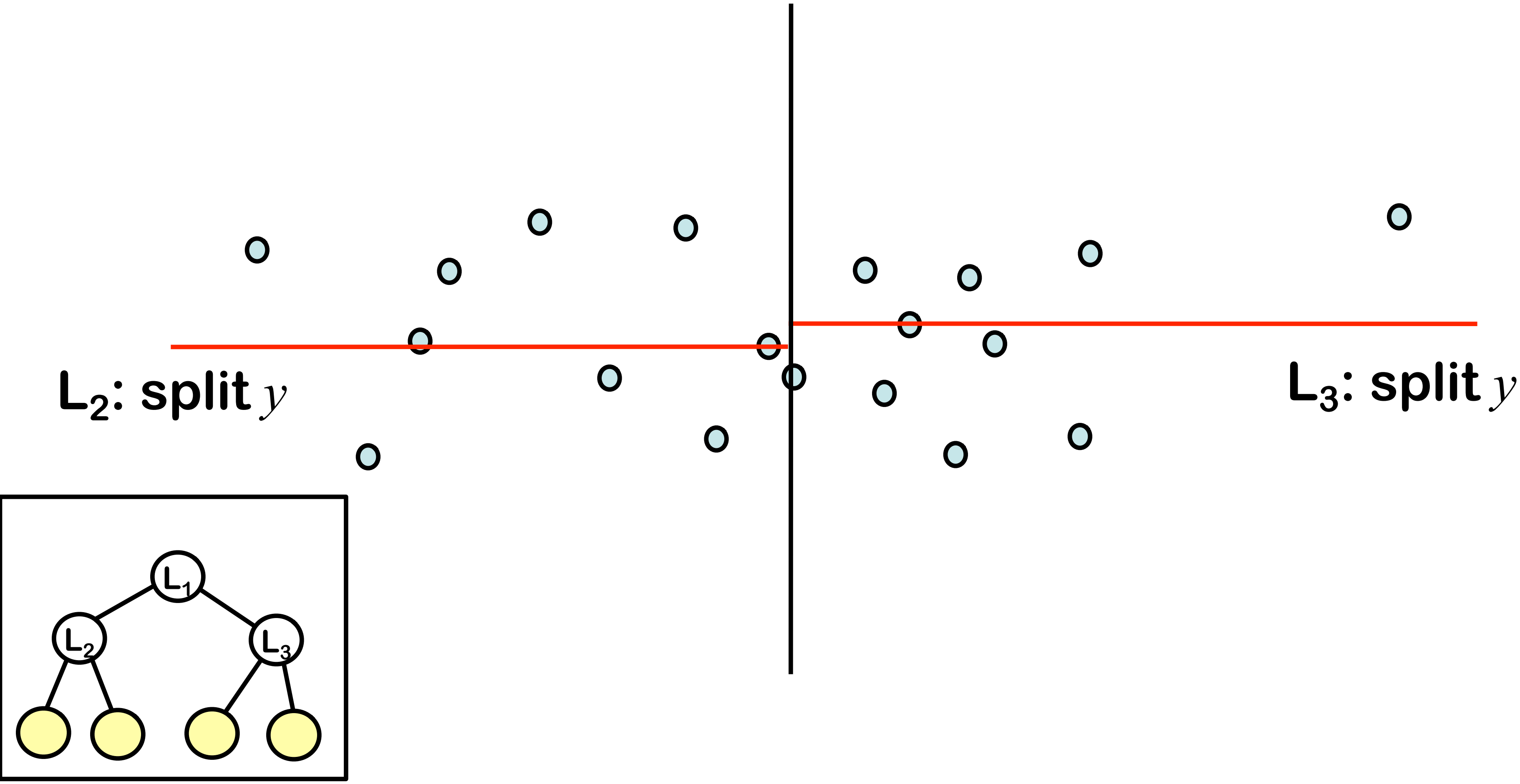
Kd-Trees

- 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:= (px, py) lies inside this rectangle iff px $\in$ [x:x’] and py $\in$ [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

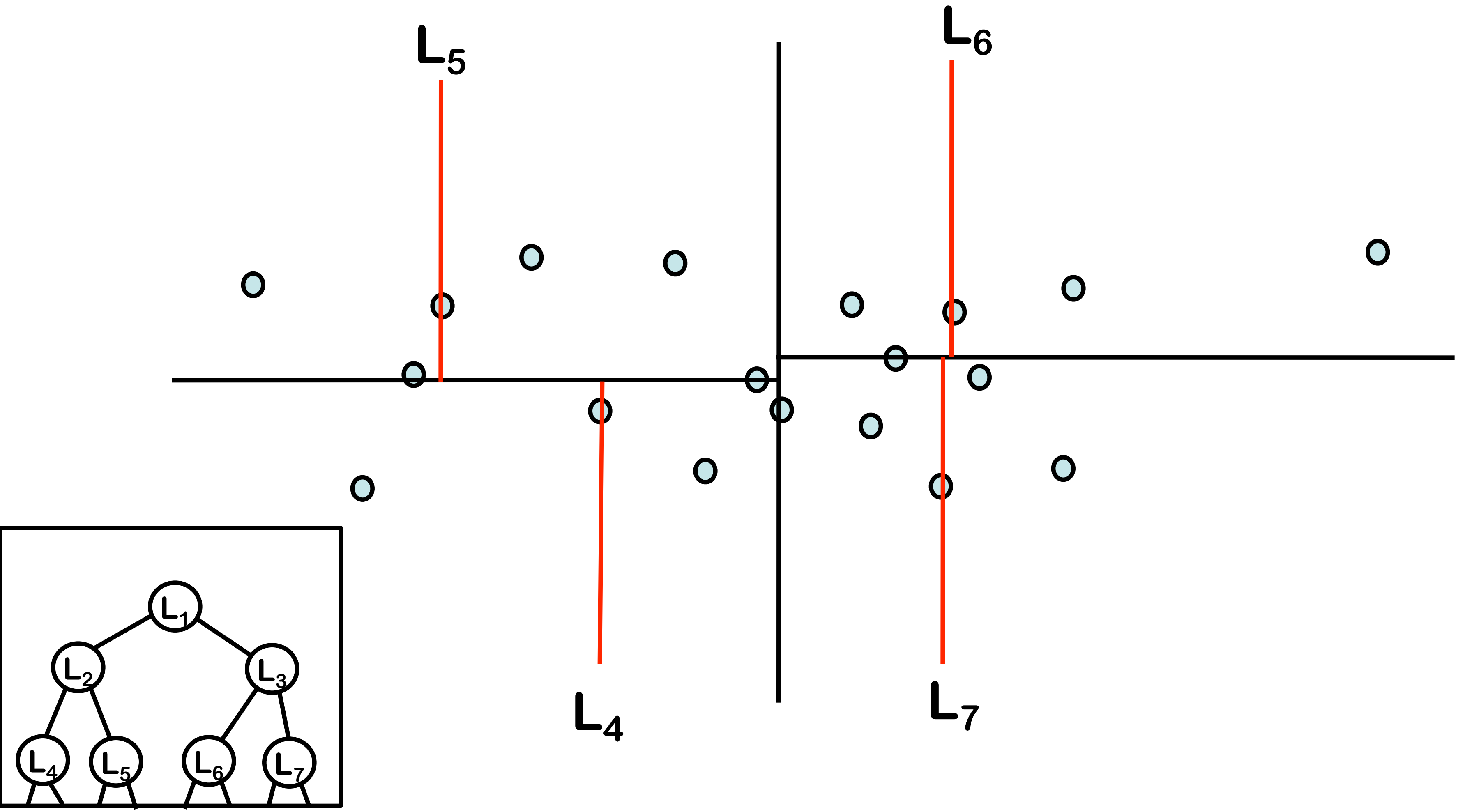
Kd-Trees



Kd-Trees



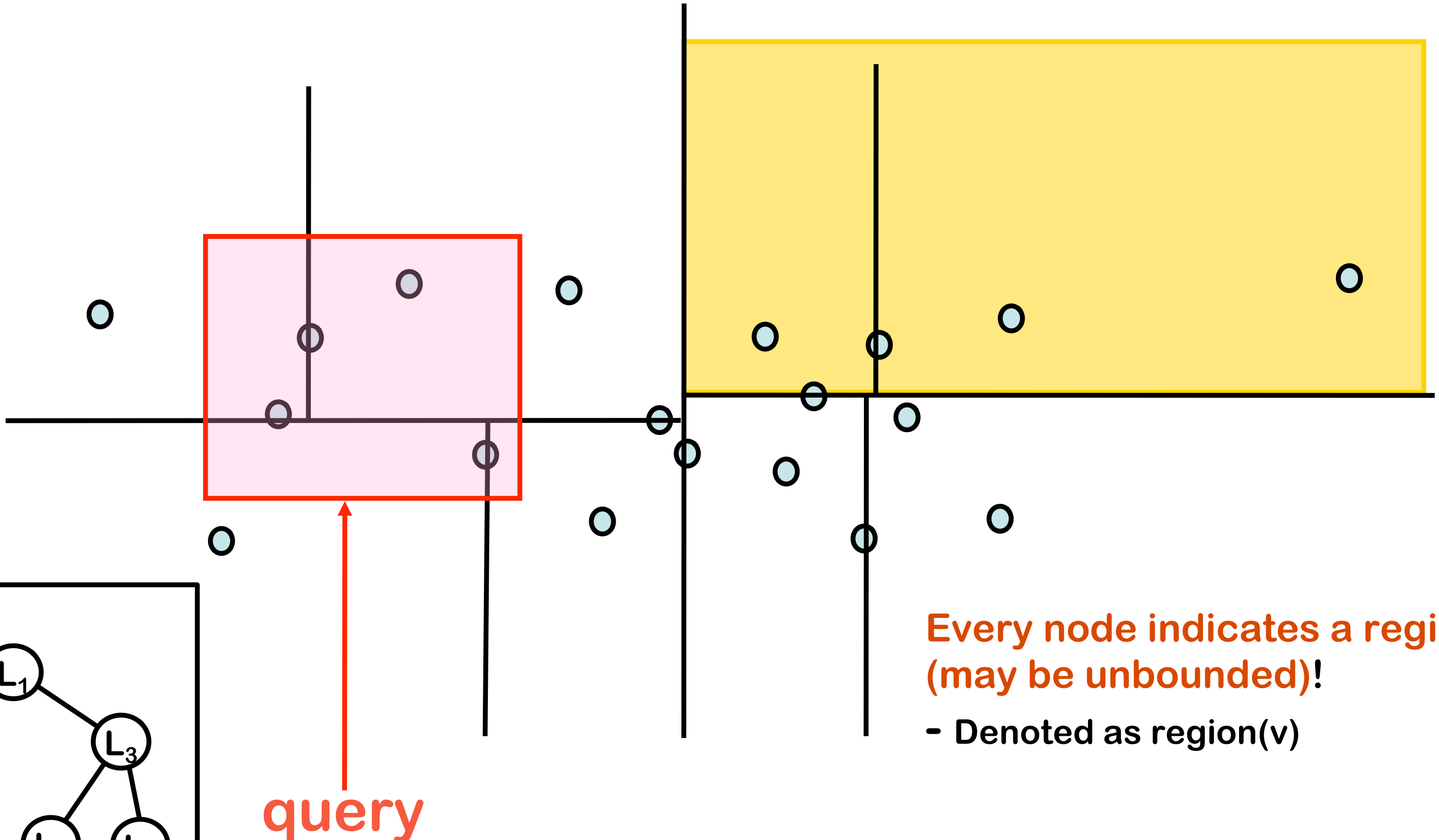
Kd-Trees

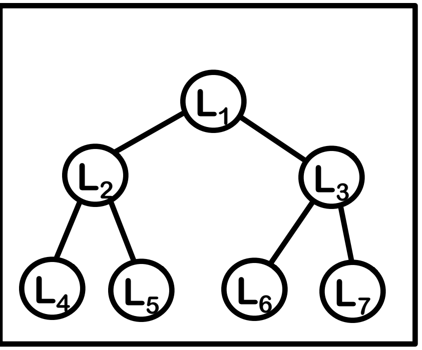


BuildKDTree(P, depth)

- 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. if depth is even then
- Split P into 2 subsets with a **vertical line** l thru **median x-coordinate** of points in P. Let P1 and P2 be the sets of points to the left or on and to the right of l respectively.
- 4. else
- Split P into 2 subsets with a **horizontal line** l thru **median y-coordinate** of points in P. Let P1 and P2 be the sets of points below or on l and above l respectively.
- 5. vleft ← BuildKDTree(P1, depth + 1)
- 6. vright ← BuildKDTree(P2, depth + 1)
- 7. Create a node v storing l, make vleft & vright lef/right children of v
- 8. return v

Querying on a kd-Tree





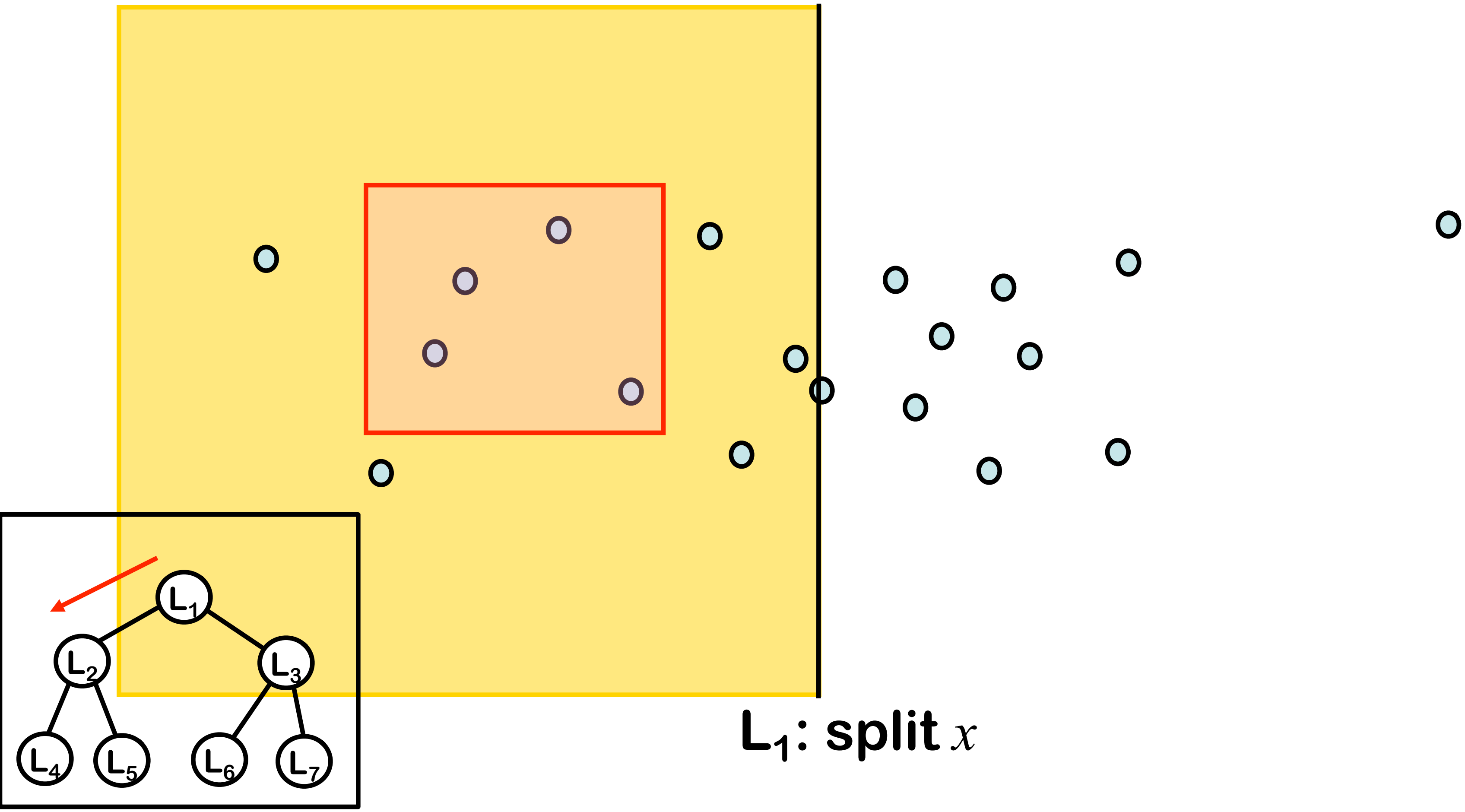
Every node indicates a region (may be unbounded)!

- Denoted as region(v)

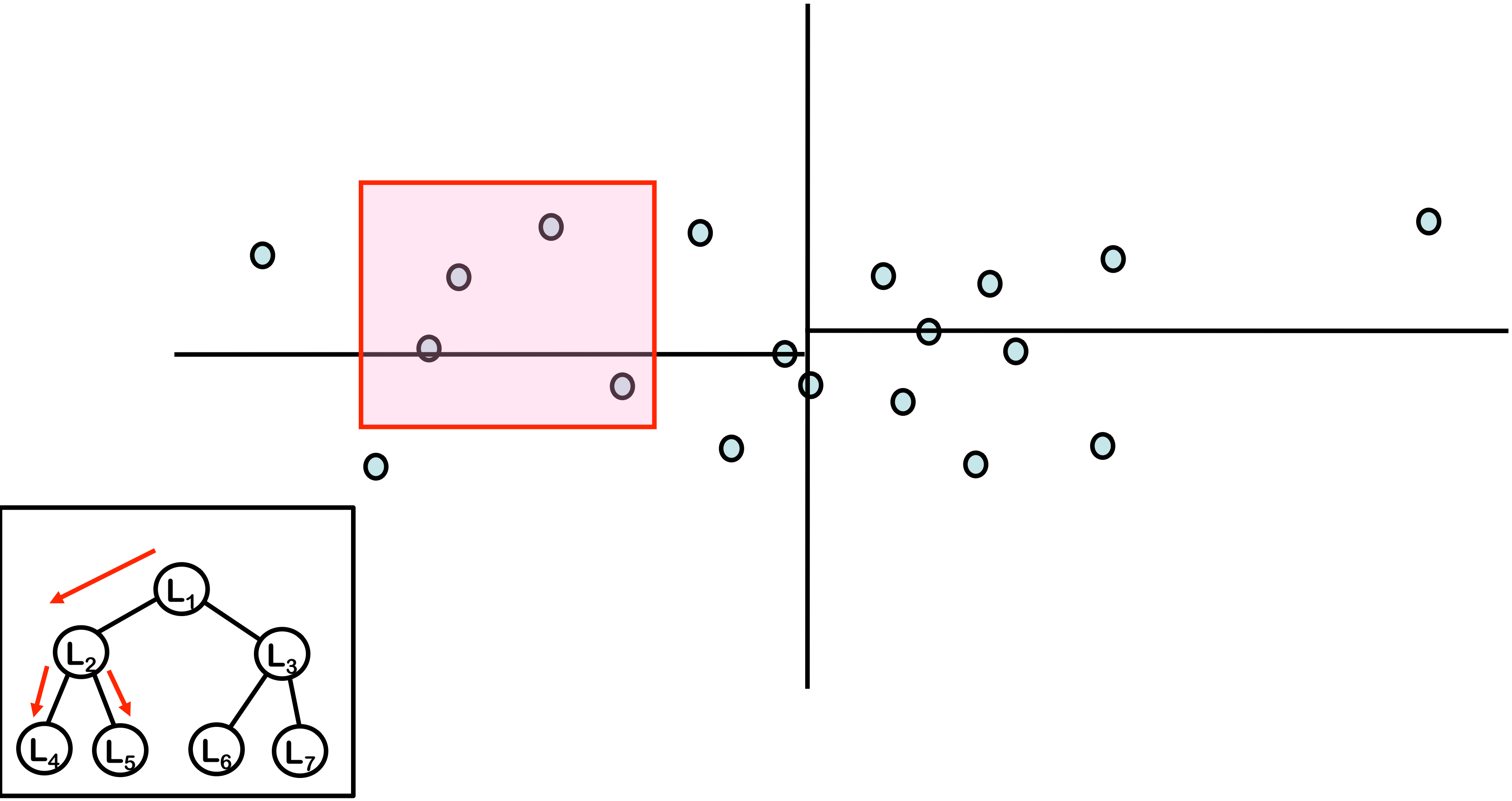
Querying on a kd-Tree

- 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

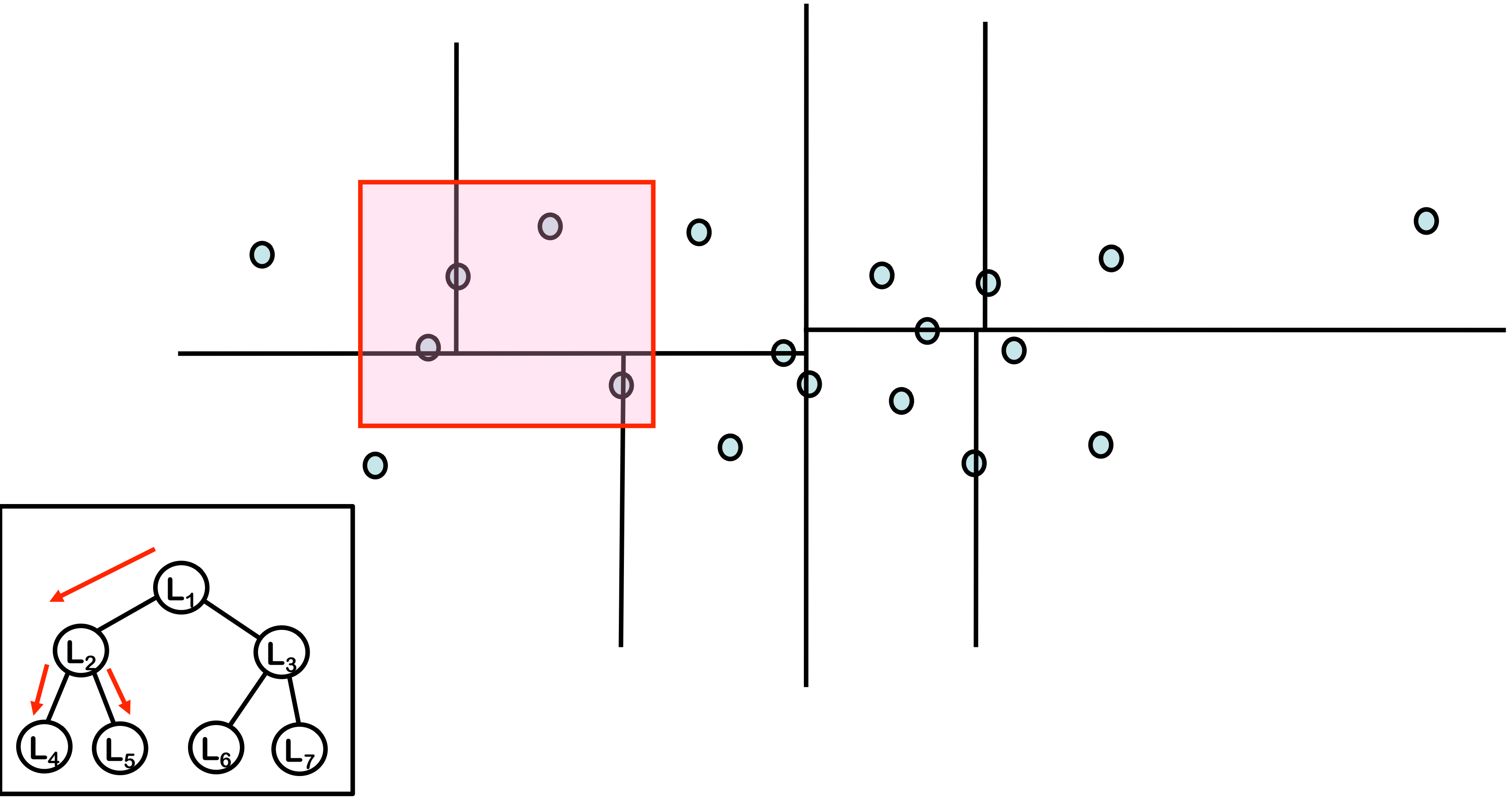
Querying on a kd-Tree



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.

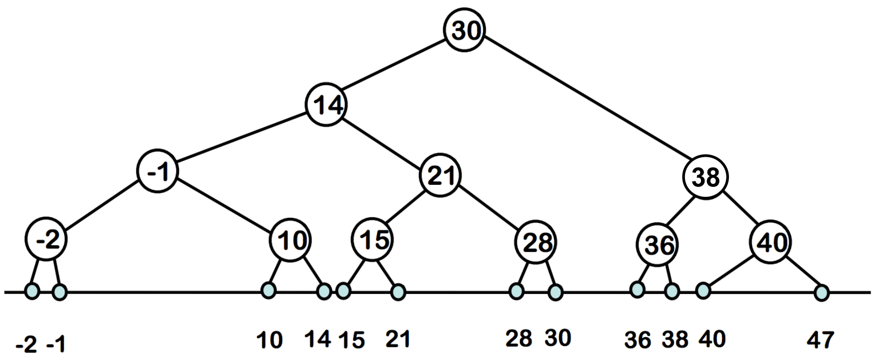
- 1. if v is a leaf
- 2. then Report the point stored at v if it lies in R.
- 3. else if region(lc(v)) is fully contained in R
- 4. then ReportSubtree(lc(v))
- 5. else if region(lc(v)) intersects R
- 6. then SearchKDTree(lc(v), R) //**recursive** call
- 7. If region(rc(v)) is fully contained in R
- 8.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

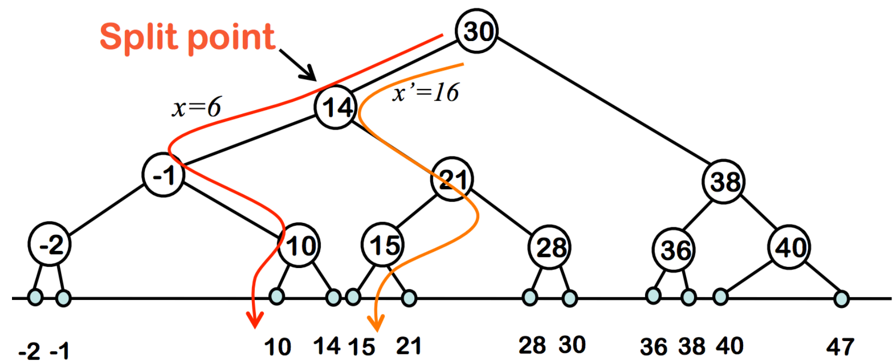
K-D-Range Search

- 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

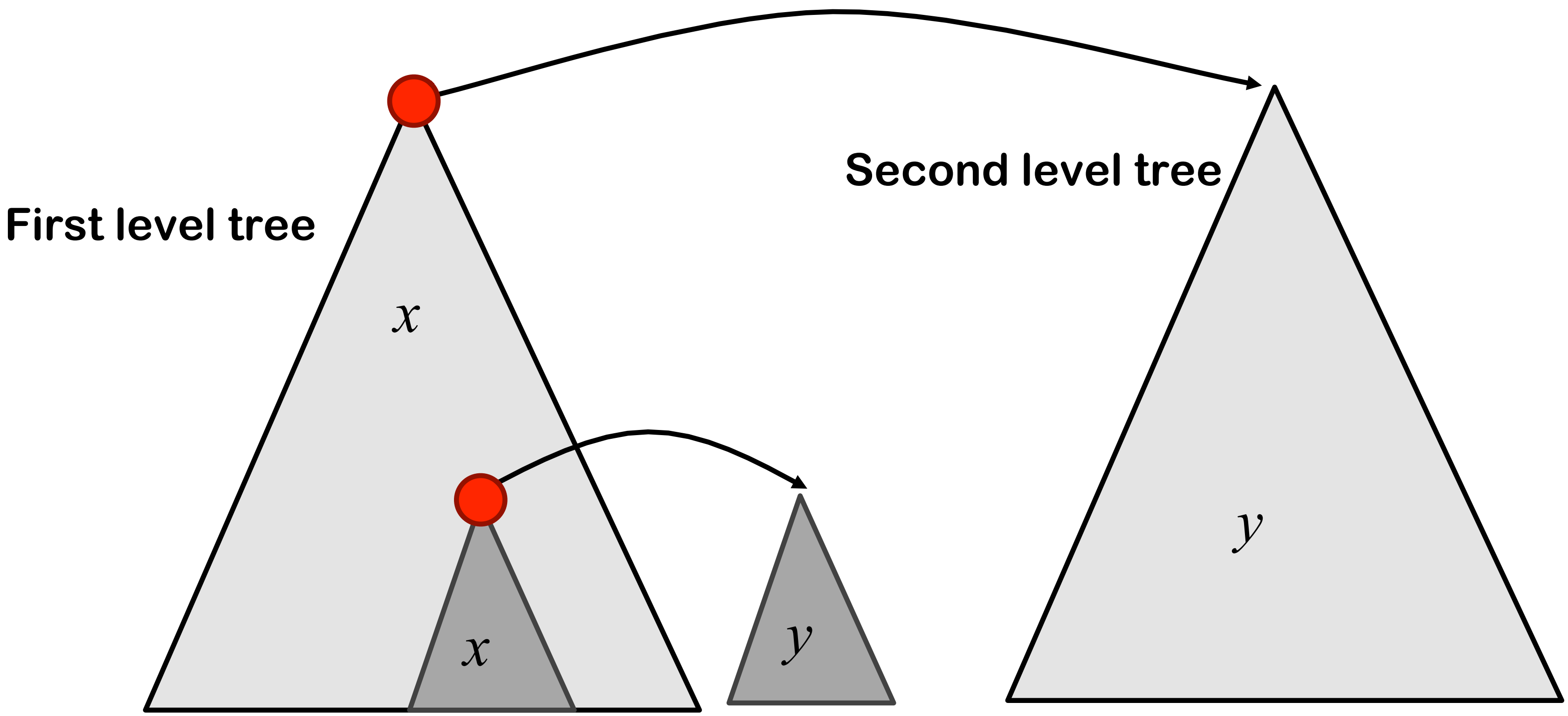
- 1D Rand Tree is a Binary Search Tree





- 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
- Range tree is **more efficient** but requires **more space** (store the same data in multiple copies!)

2D Range Trees



Build2DRangeTree(P)

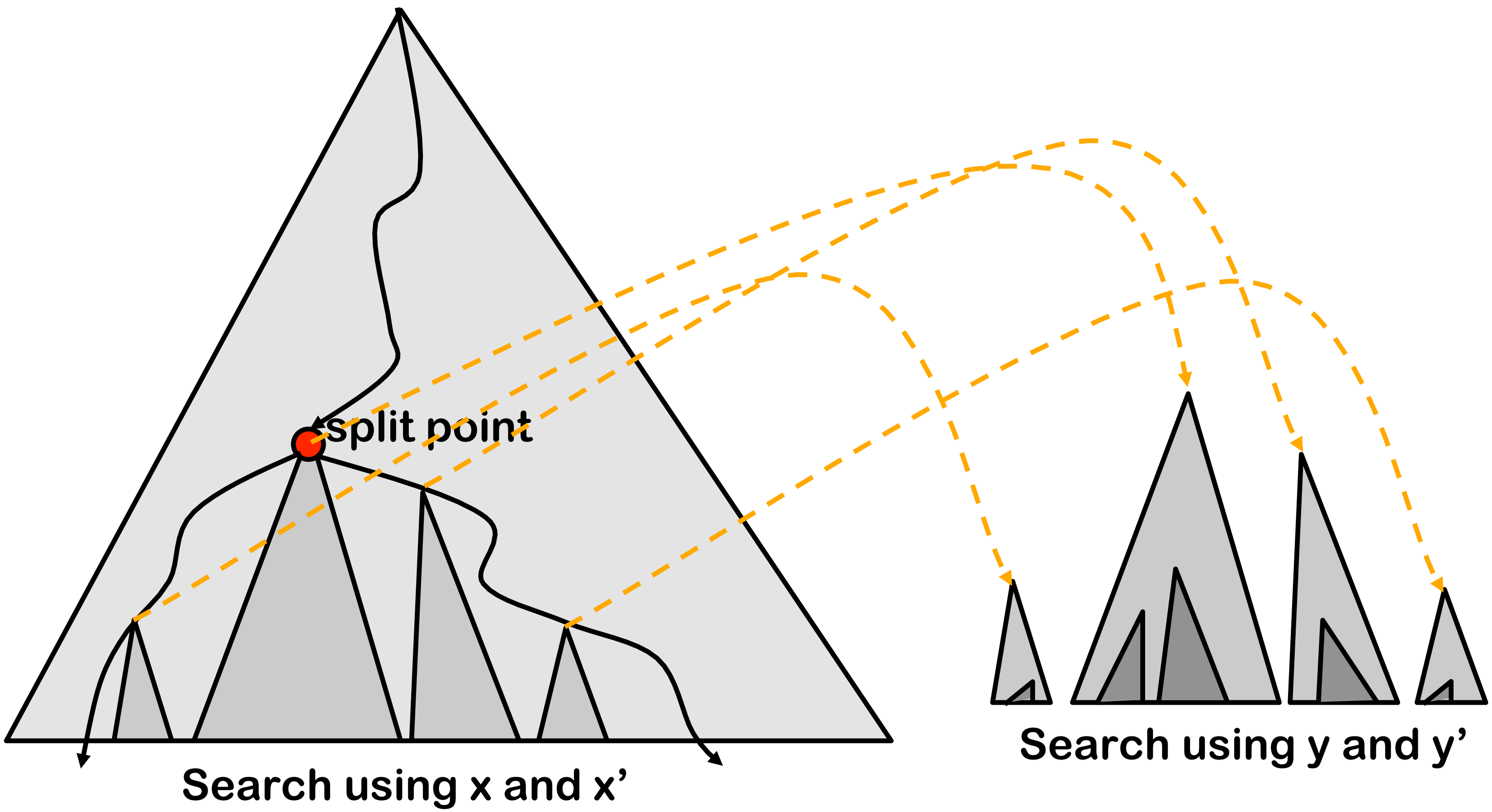
```
Input: A set P of 2-D points on the plane.
Output: The root of 2-D range tree.

Node Build2DRangeTree(P)
{
        if( P contains only 1 point )
        {
                Create a node v storing this point
                v.next_level = Build1DRangeTree(y-coordinates of the points in P);
        }
        else
        {
                Split P into 3 subsets:
                    x_mid: the median x-coordinate;
                    P_left containing points with x-coordinate <= x_mid;
                    P_right containing points with x-coordinate > x_mid;

                 vleft = Build2DRangeTree(Pleft);
                 vright = Build2DRangeTree(Pright);

                 Create a node v storing xmid;
                 make vleft left child of v;
                 make vright right child of v;
                 v.next_level = Build1DRangeTree(y-coordinates of the points in P);
        }

        return v
}
```

Build 2D 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



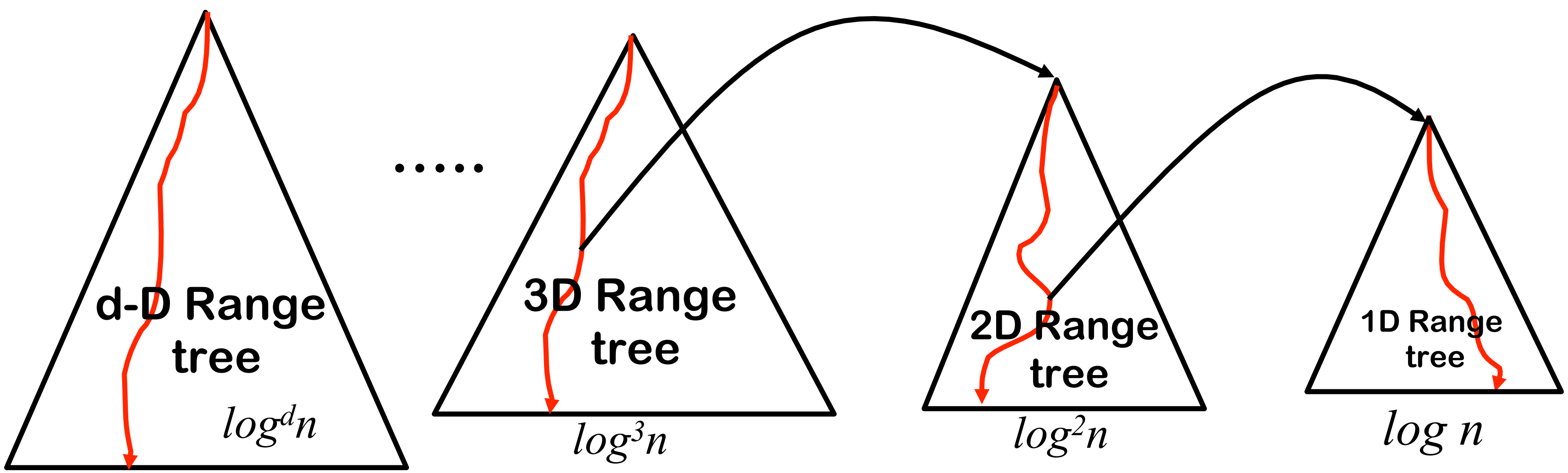
2D RangeQuery

```
Input: A range R=[x:x', y:y']
Output: All points that lie in the range.

RangeSearch_2D(x,x', y, y')
{

        vsplit = FindSplitNode(x,x')

        if( vsplit is a leaf )
        {
                Check if the point stored at vsplit must be reported
        }
        else {
                // Follow the path to x and report the points in subtrees right of the path
                v = vsplit.left
                while(v is not a leaf) do {
                        if( x <= v.x )
                        {
                          if( v.right.next_level!=null )
                                v.right.next_level.RangeSearch_1D(y,y');
                          else
                          {
                                  if(v.right is not a leaf) {
                                        ReportSubTree(v.right);
                                  }
                                  else{
                                        Check if the point stored at leaf v.right must be reported
                                  }
                          }
                          v = v.left
                        }
                        else{ v = v.right}
                } //end while
                Check if the point stored at leaf v must be reported
```

Do the same for the upper bound x'

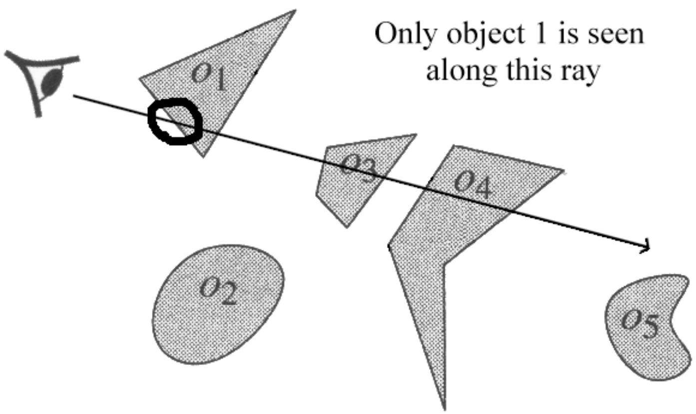
Higher-D Range Trees

- 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.



Drawing the Visible Objects

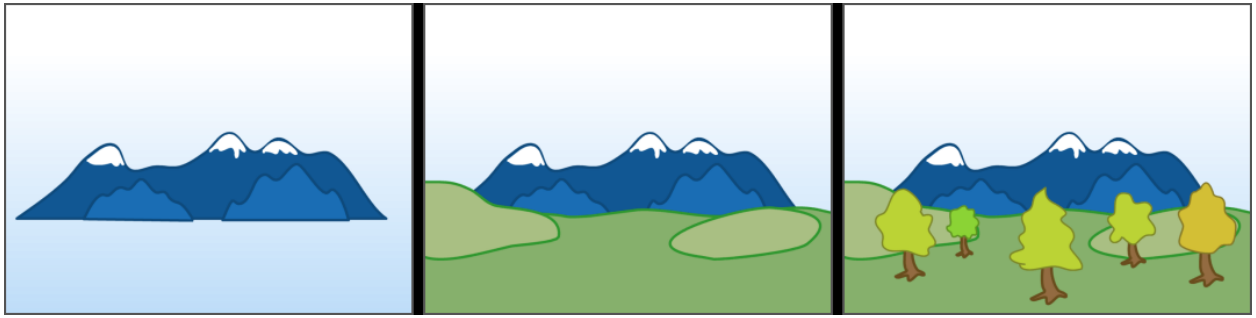
- We want to generate the image that the eye would see, given the objects in our space
- How do we draw the correct object at each pixel, given that some objects may obscure others in the scene?



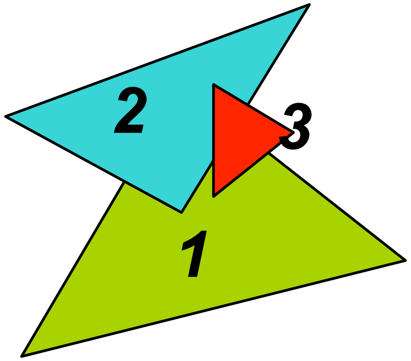
Hidden surface removal

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?

BSP Trees

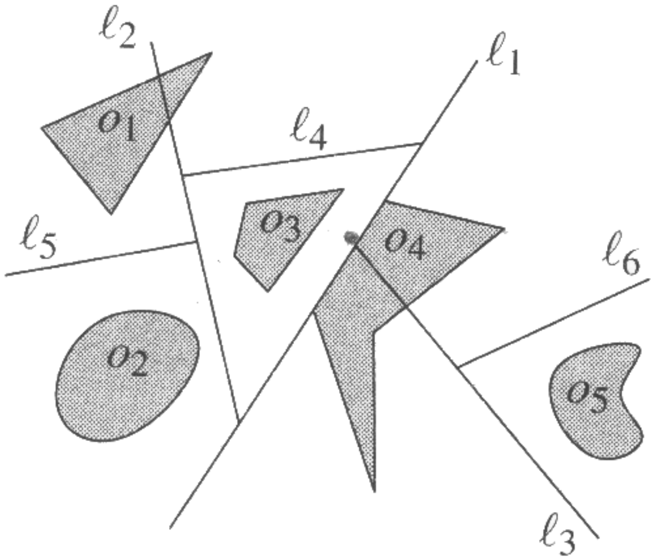
- Having a pre-built BSP tree will allow us to get a correct depth order of polygons in our scene for any point in space.

- 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

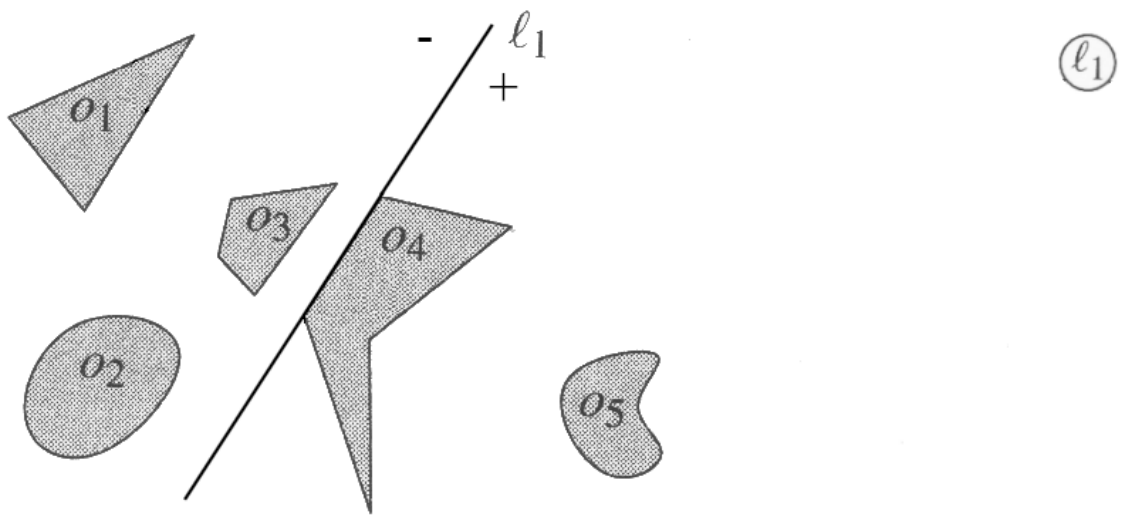
Assume that no objects in our space overlap

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



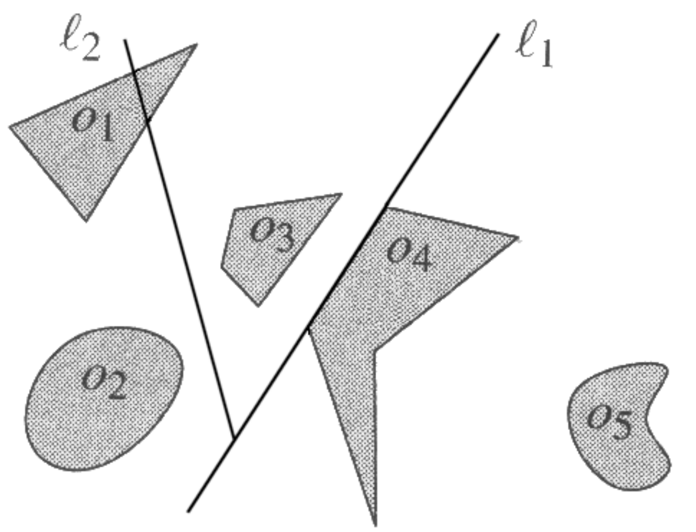
Choose a Splitting Line

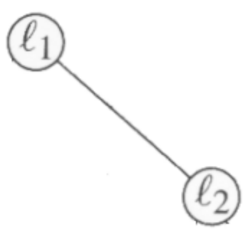
Choose a splitting plane, dividing our objects into three sets – those on each side of the plane, and those fully contained on the plane.



Choose More Splitting Lines

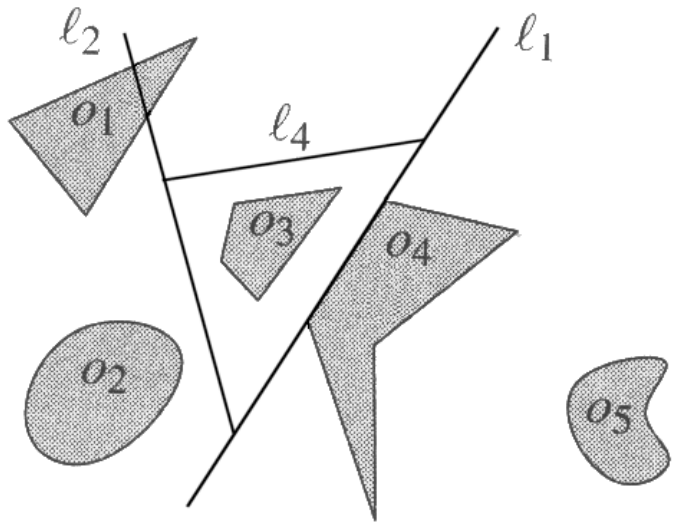
- What do we do when an object (like object 1) is divided by a splitting plane?
- It is divided into two objects, one on each side of the plane.

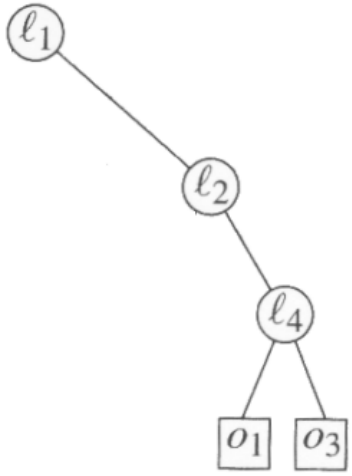




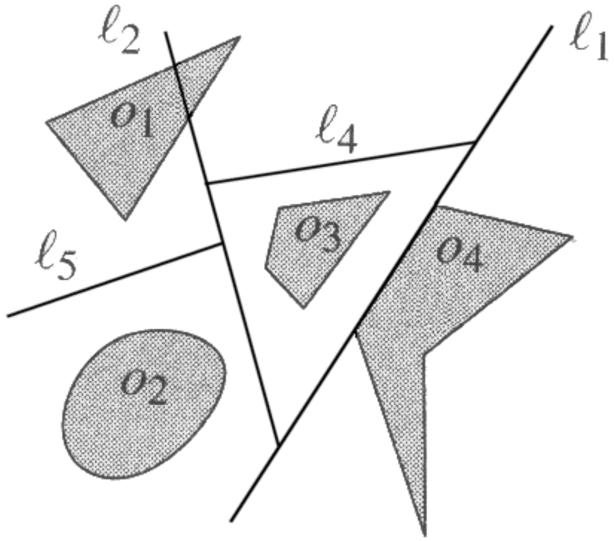
Split Recursively Until Done

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

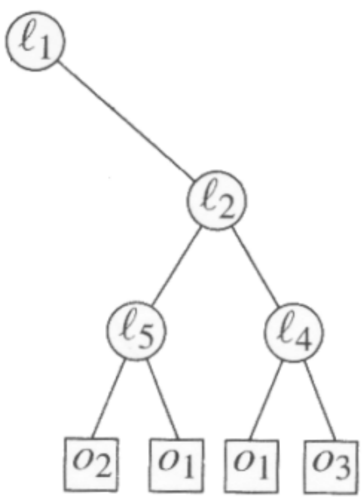




Continue

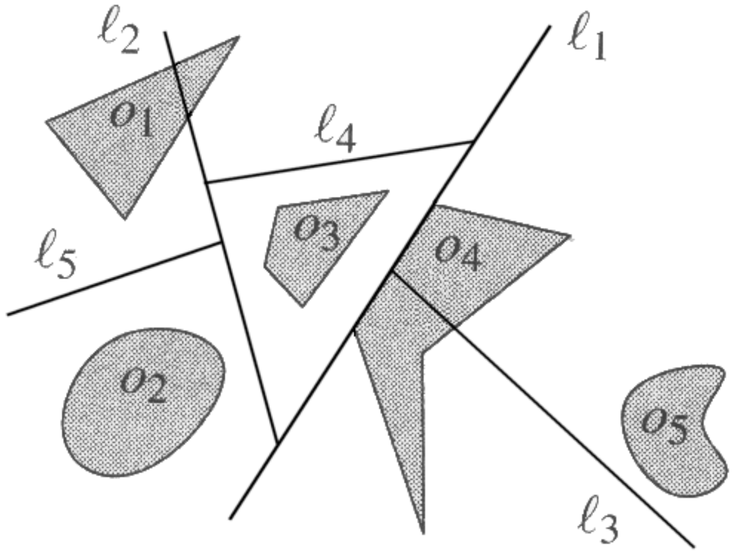


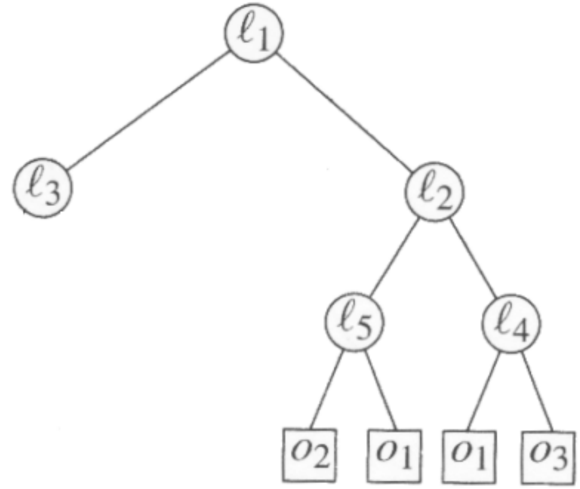
ℓ_2
ℓ_1
o_1
ℓ_4
o_3
ℓ_5
o_4
o_2
o_5



ℓ_1
ℓ_2
ℓ_5
ℓ_4
o_2
o_1
o_1
o_3

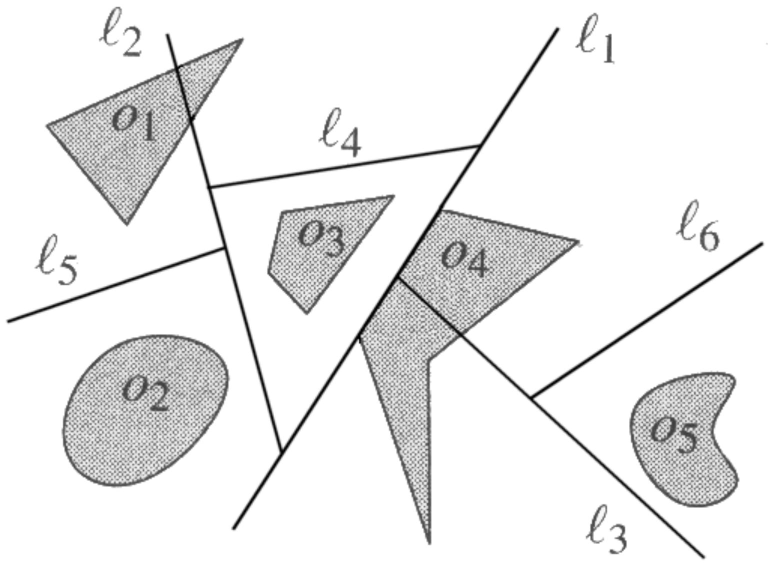
Continue

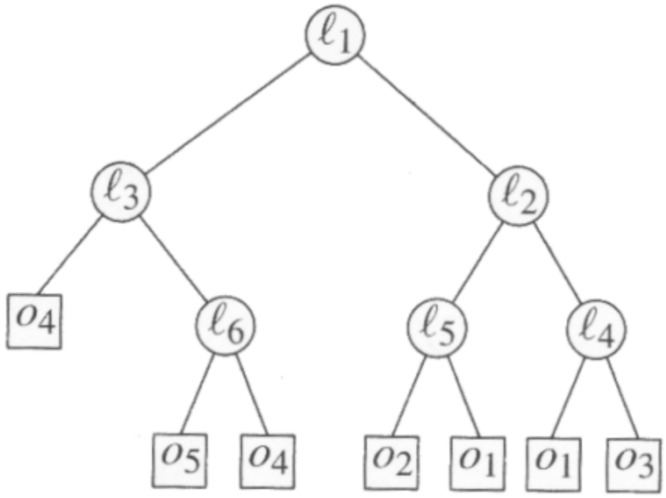

ℓ_2
ℓ_1
o_1
ℓ_4
o_3
o_4
ℓ_5
o_2
o_5
ℓ_3


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.

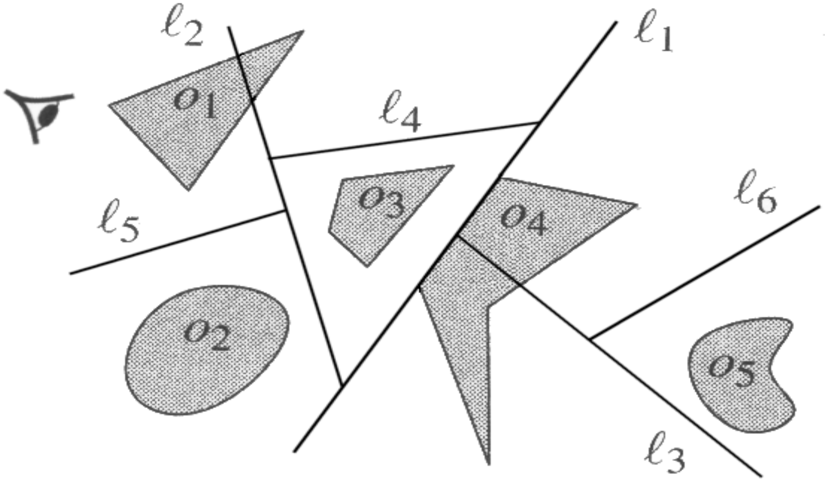


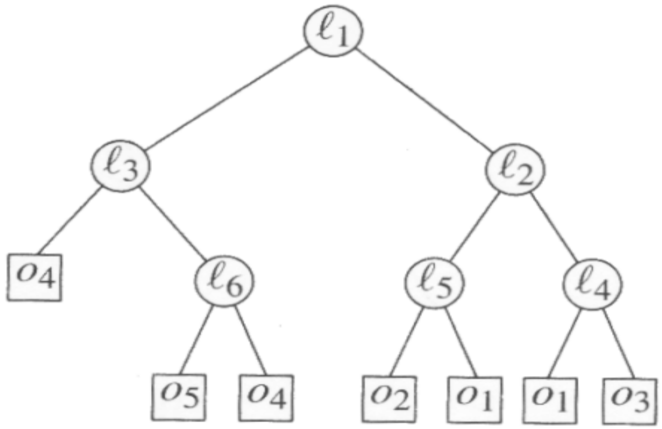


Querying the Tree

- 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?

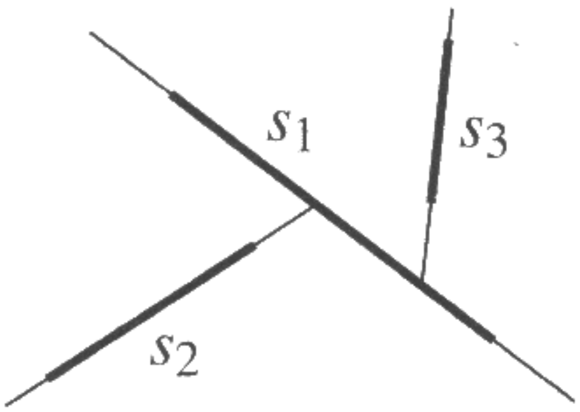
Building a BSP Tree From Line Segments in the Plane

- 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

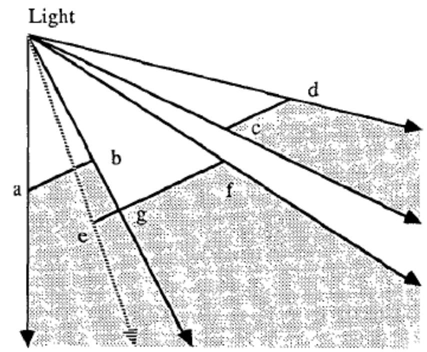
- Return a new node whose children are these +/- half space BSP’s, and which contains the list of segments entirely contained along the line through s.

Our Building Algorithm Extends to 3D!

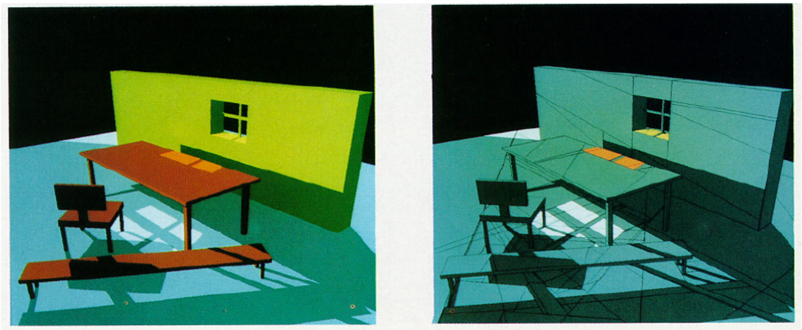
The same procedure we used to build a tree from two-dimensional objects can be used in the 3D case – we merely use polygonal faces as splitting planes, rather than line segments.

More applications

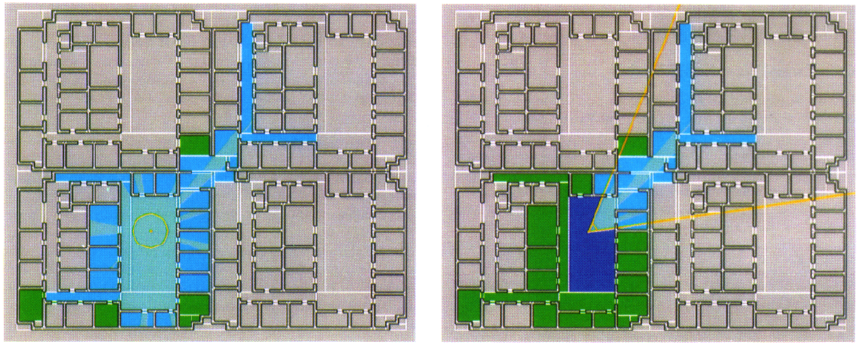
- Querying a point to find out wh it exists within



- Shadows



- Visibility



- Blending
    - Opengl stuff

Questions for You

- What are the scene managers in Unity and Unreal? What data structures are used in Unity and Unreal to assist collision, visibility and proximity queries?