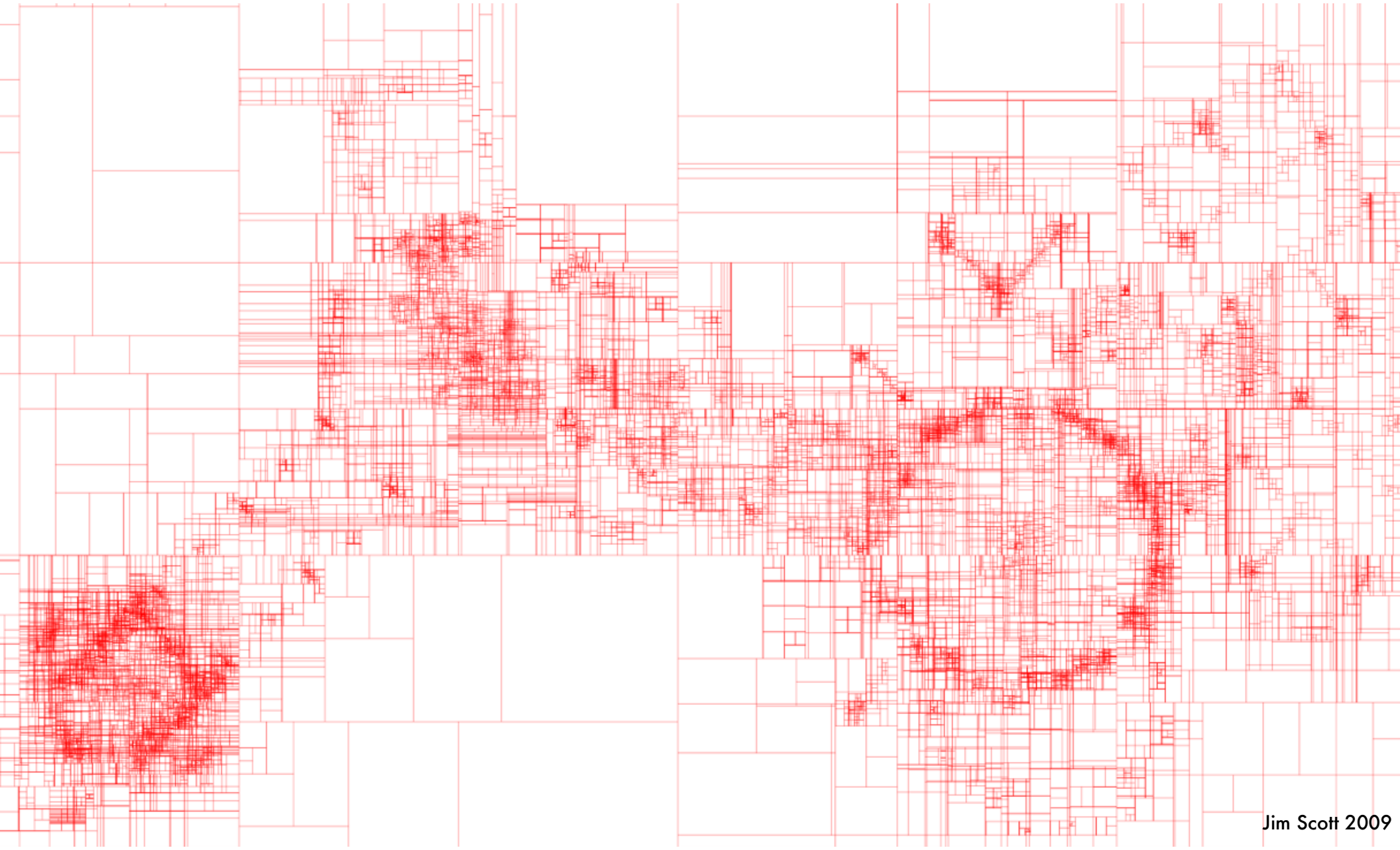
Parallel Physically Based Path-tracing and Shading Part 3 of 2

CIS565 Fall 2012
University of Pennsylvania
by Yining Karl Li



Spatial Acceleration Structures: KD-Trees

*Some portions of these slides are adapted from Philipp Slusallek (Saarland University)'s Computer Graphics course.

Space partitioning

- Let's say we want to cast a ray into a scene and find the nearest object the ray intersects. What is the easiest way for us to do this?

Space partitioning

- Let's say we want to cast a ray into a scene and find the nearest object the ray intersects. What is the easiest way for us to do this?
  - Loop through every object, return the closest one.

Space partitioning

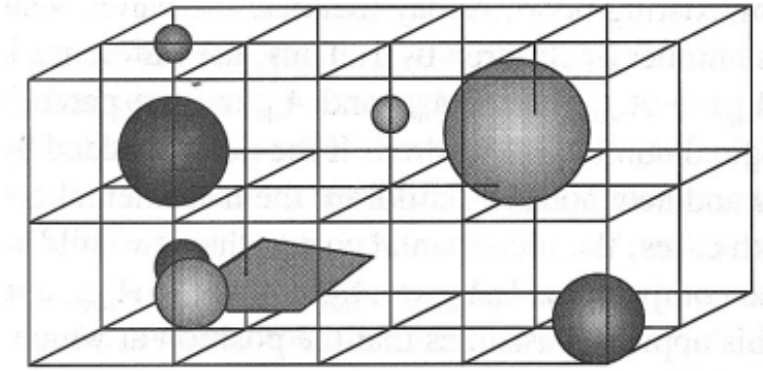
- Let's say we want to cast a ray into a scene and find the nearest object the ray intersects. What is the easiest way for us to do this?
  - Loop through every object, return the closest one.
- What if we have 100 million triangles?
  - Intersection testing every single triangle for every single ray in every single path for every single iteration for every single pixel adds up to a lot of wasted computation
  - Solution: space partitioning!

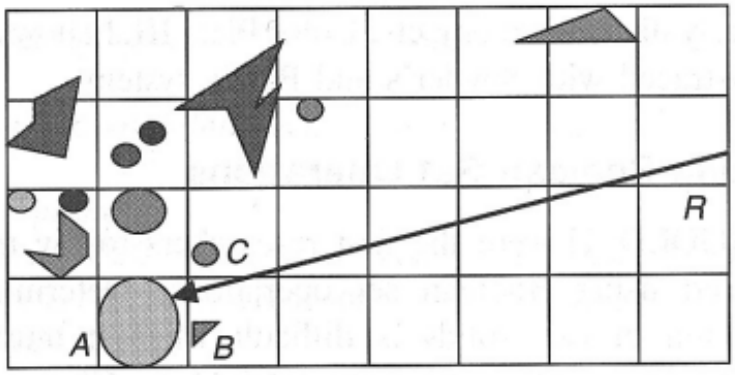
Space partitioning

- Let's say we want to cast a ray into a scene and find the nearest object the ray intersects. What is the easiest way for us to do this?
  - Loop through every object, return the closest one.
- What if we have 100 million triangles?
  - Intersection testing every single triangle for every single ray in every single path for every single iteration for every single pixel adds up to a lot of wasted computation
  - We need a way to quickly determine what objects are in the space immediately surrounding a given point, i.e. where the ray is
  - Solution: spatial partitioning!

Uniform Grid Partitioning

- What if we overlay a uniform voxel grid, assign each object to the grid cell it is in, and raymarch through the grid and intersect against only objects in the current grid cell?



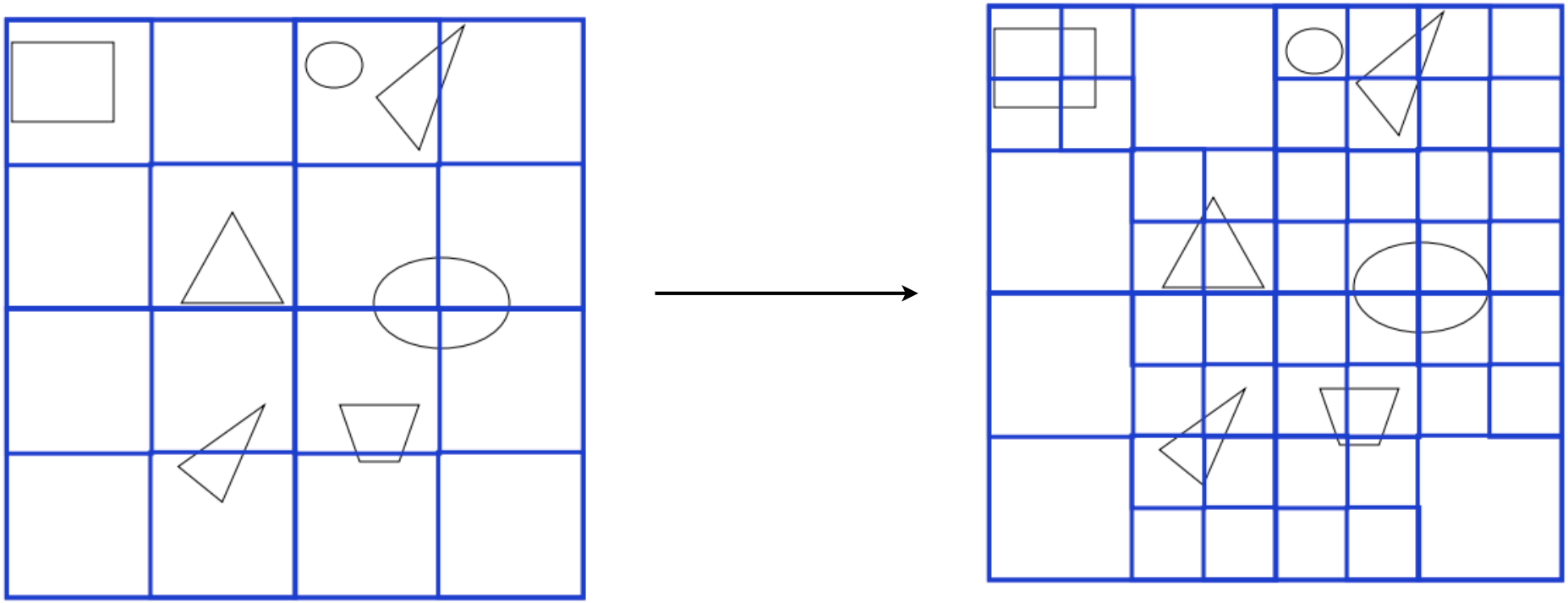


Uniform Grid Partitioning

- Uniform grid partitioning is a good start, BUT, there are some potential problems:
  - Memory bound can get out of hand rapidly, since the resolution of the grid has to be proportional to the scene size
  - A lot of space is potentially wasted if we have a "teapot in a football stadium" type scenario
  - Also of no help whatsoever if we have a "angry mob in a stadium" type scenario
  - What if objects span multiple voxels in the grid?

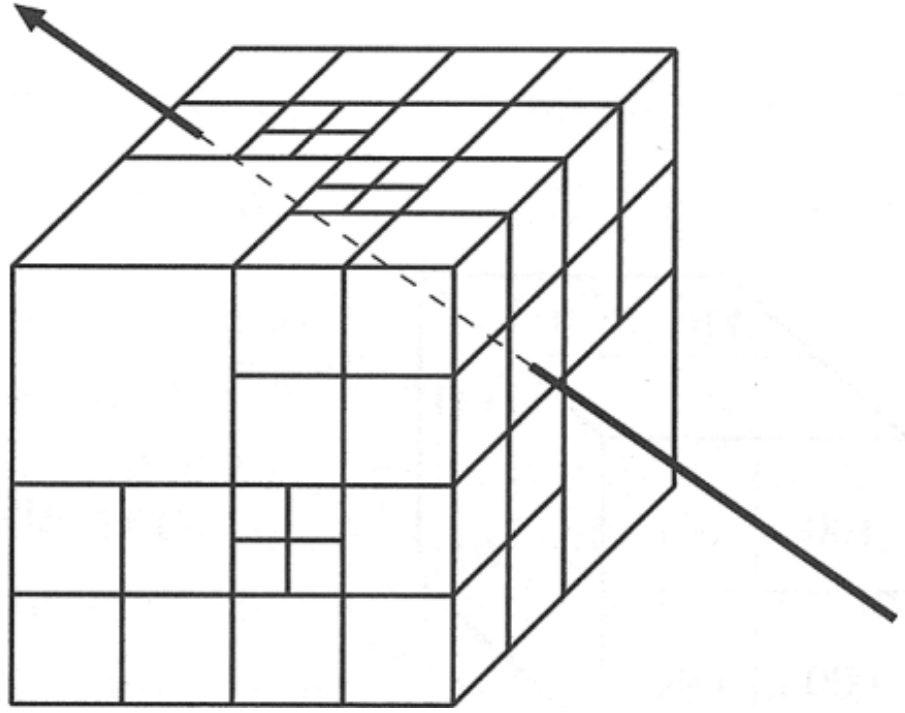
Octrees

- We can solve some of the problems with uniform grids by allowing cells to be hierarchically subdivided…



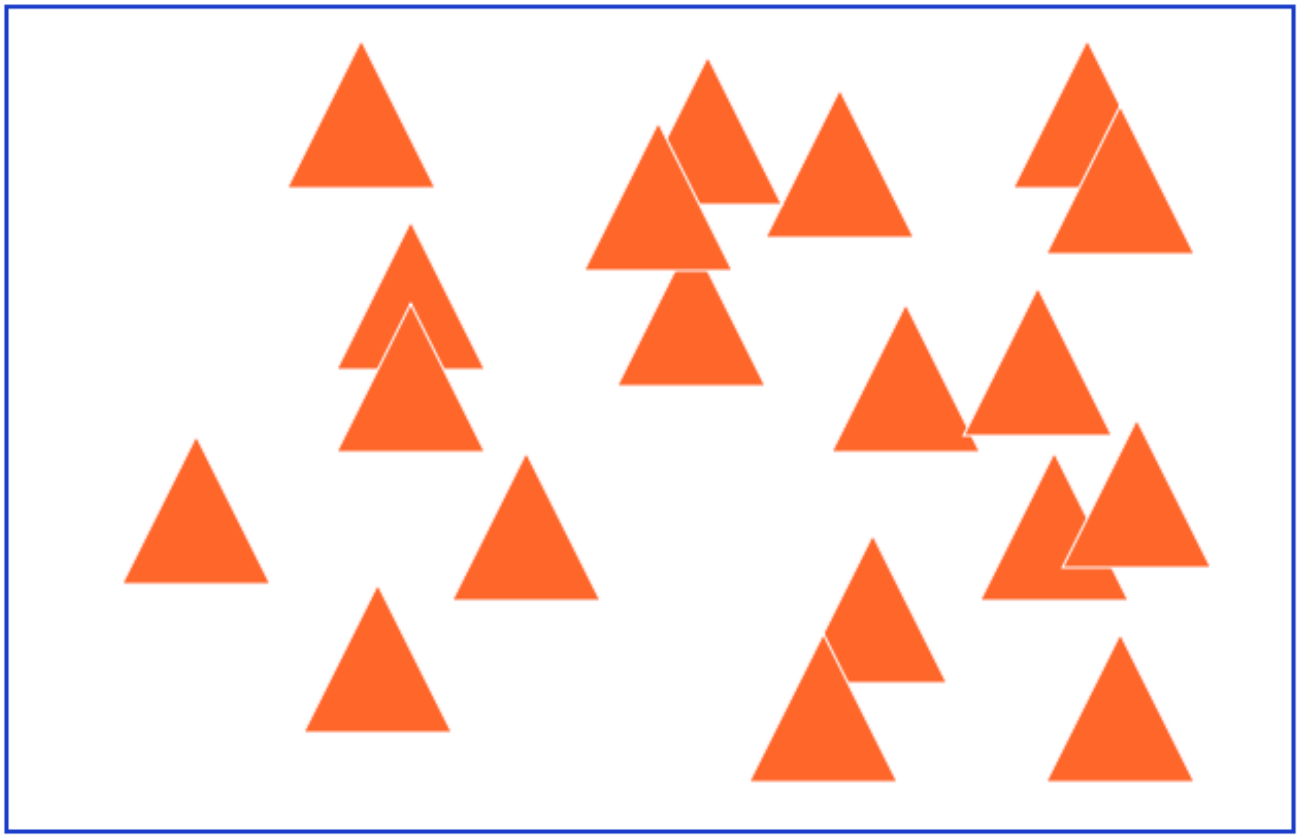
Octrees

- However, traversal for octrees can be quite complex, and we still have a lot of potentially wasted space
- Constructing octrees can also take a while, since geometry-heavy areas will need more time to refine



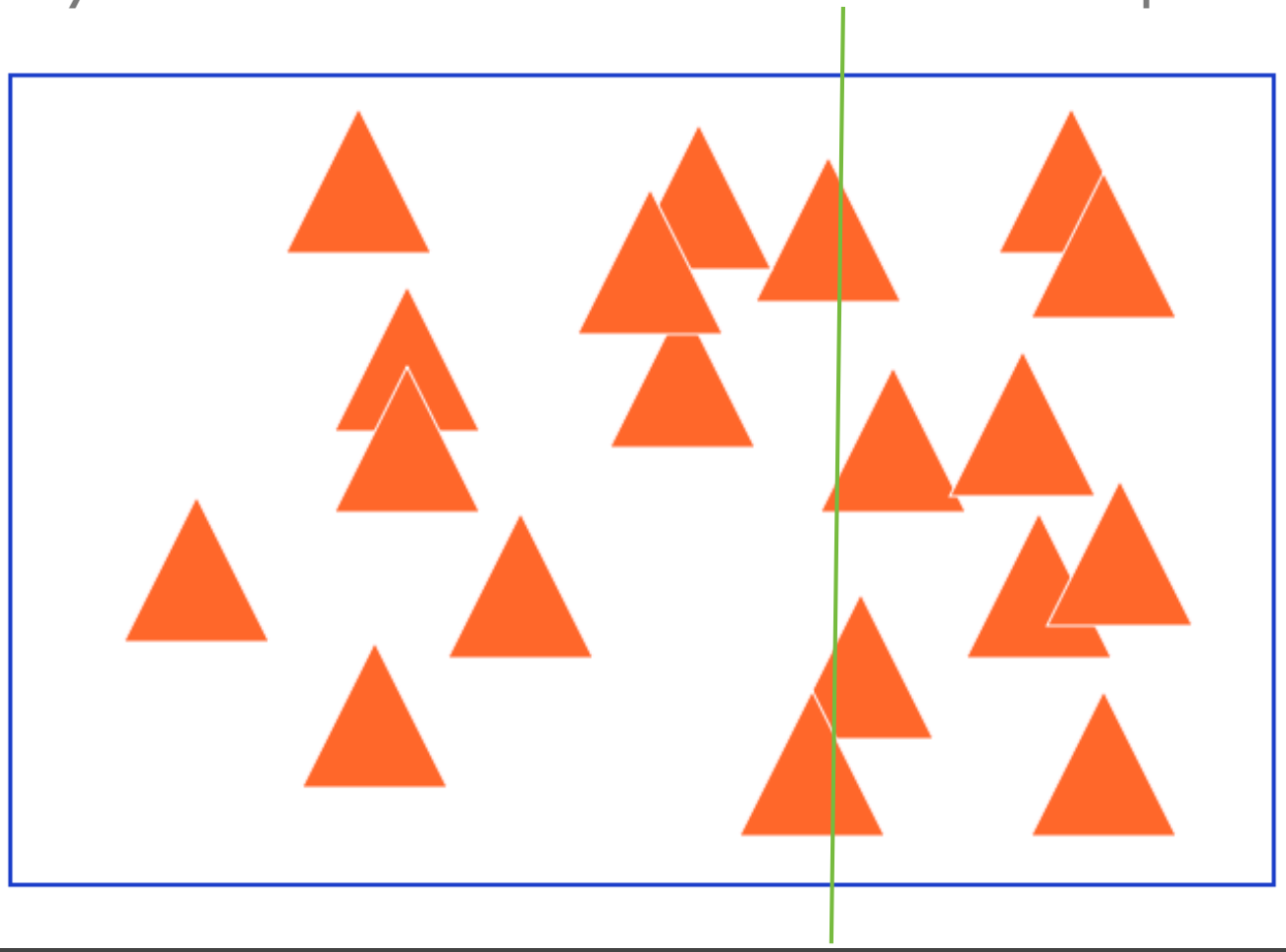
KD-Trees

- What if instead of using a uniform grid, we recursively split the scene with a split boundary based on the densities of each side of the split boundary?



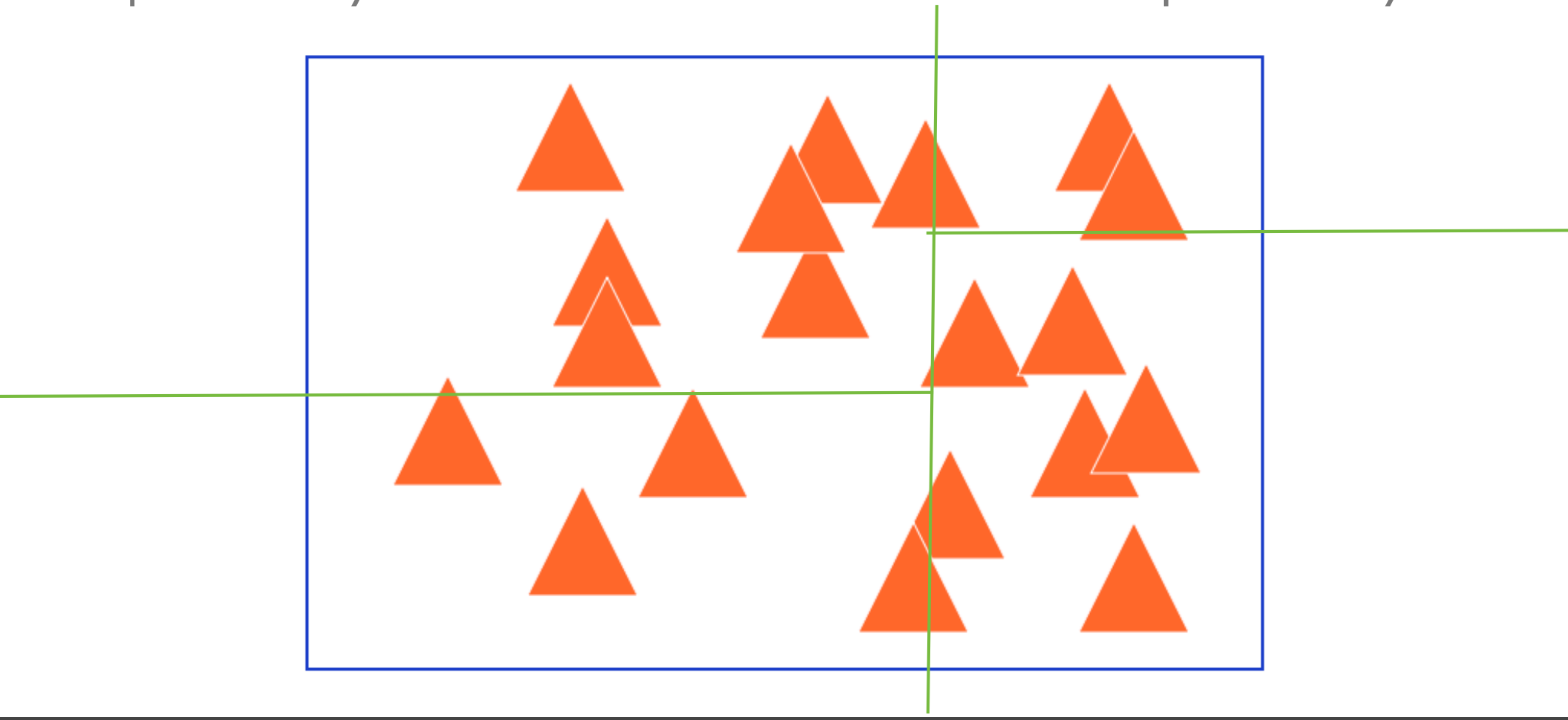
KD-Trees

- What if instead of using a uniform grid, we recursively split the scene with a split boundary based on the densities of each side of the split boundary?



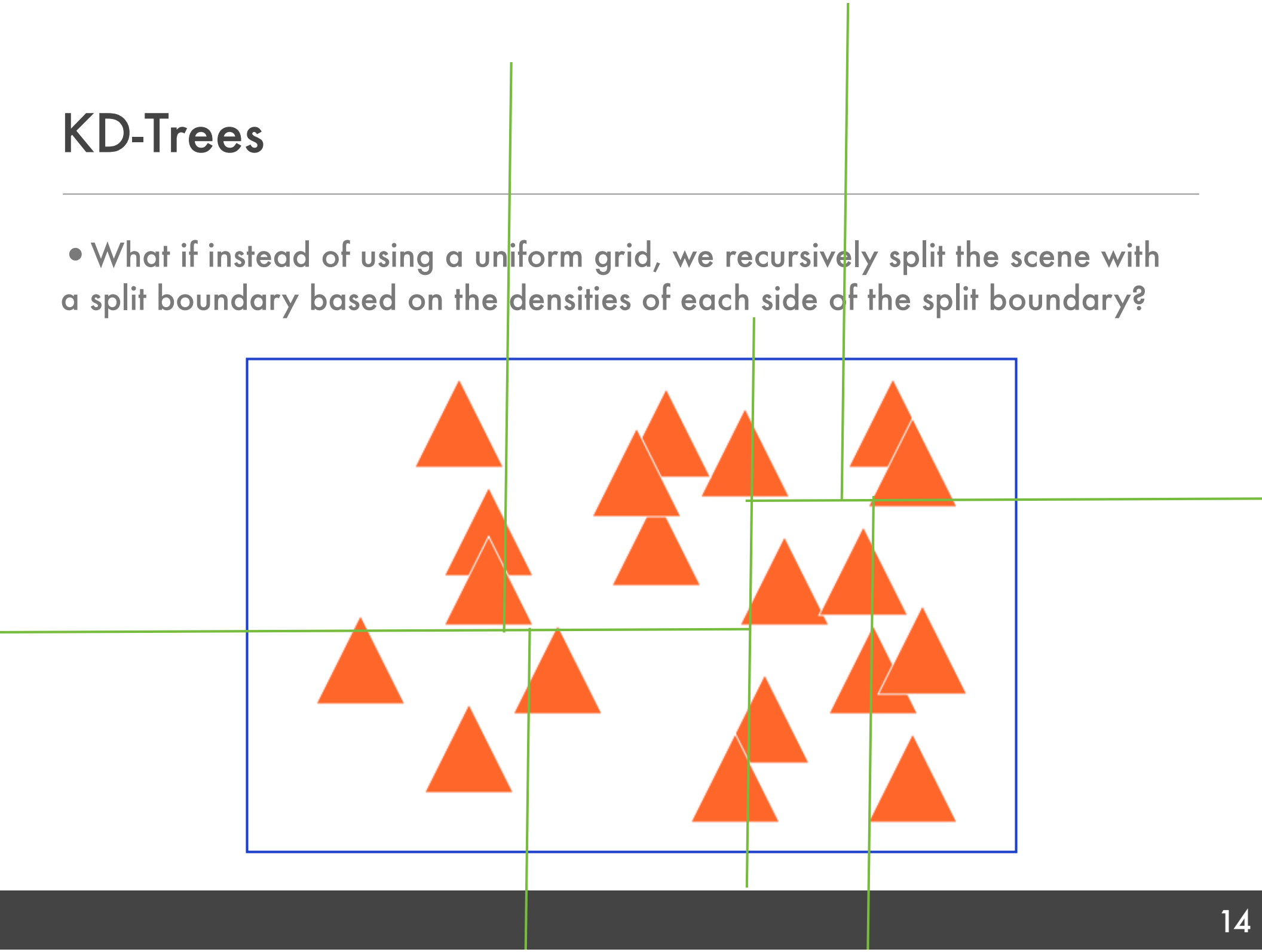
KD-Trees

- What if instead of using a uniform grid, we recursively split the scene with a split boundary based on the densities of each side of the split boundary?



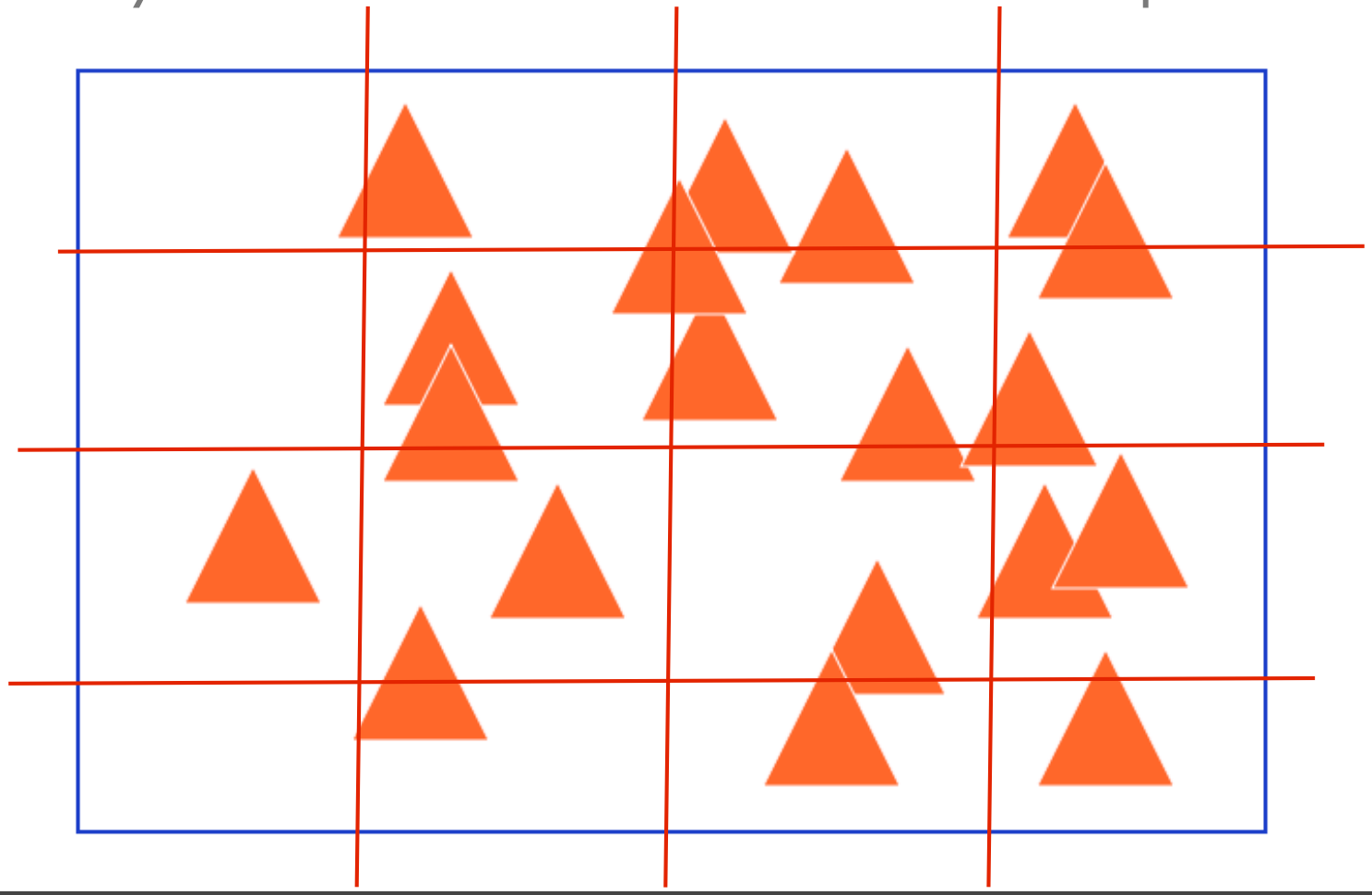
KD-Trees

- What if instead of using a uniform grid, we recursively split the scene with a split boundary based on the densities of each side of the split boundary?

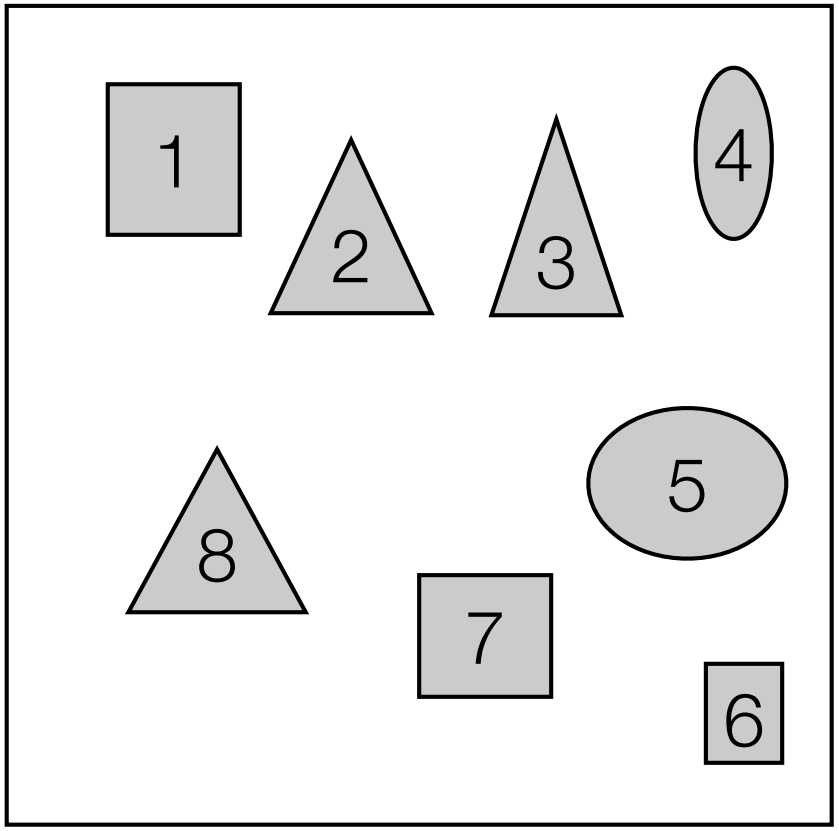


KD-Trees

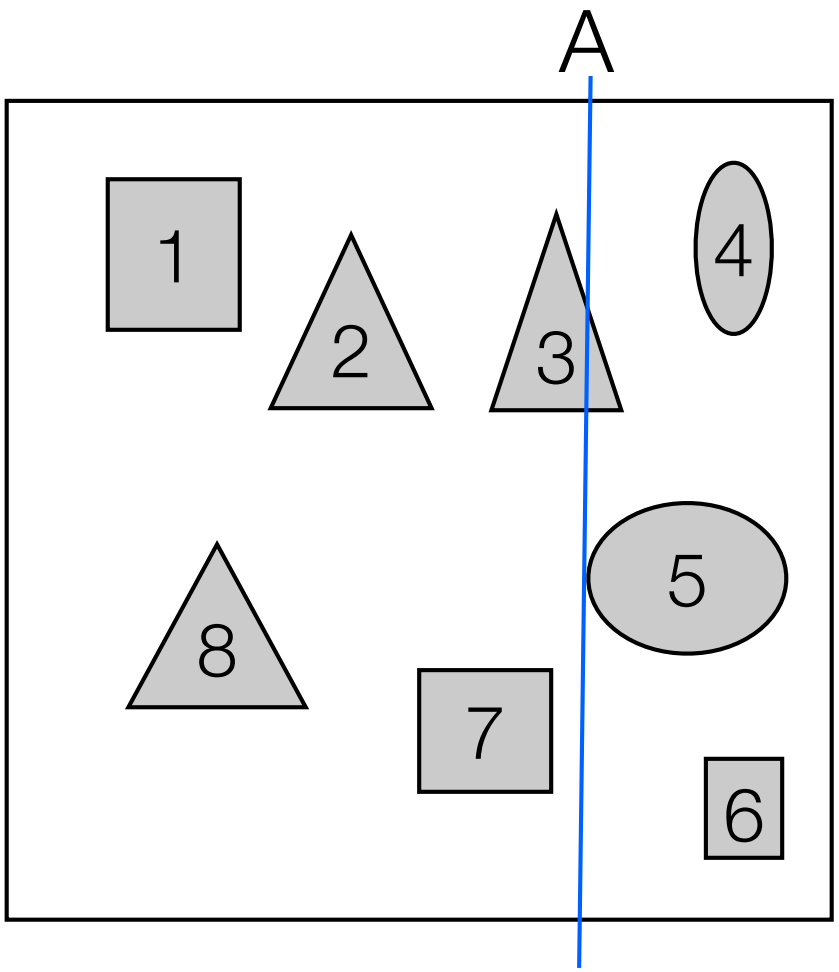
- What if instead of using a uniform grid, we recursively split the scene with a split boundary based on the densities of each side of the split boundary?



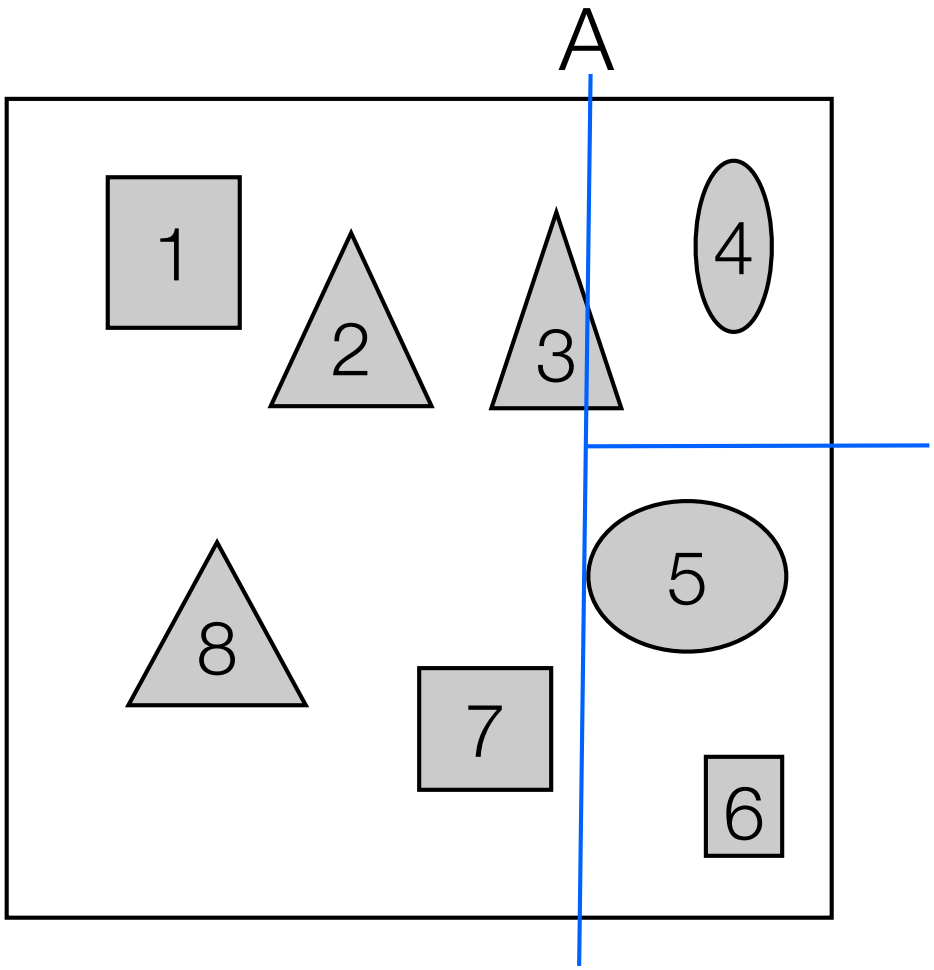
KD-Trees: Explicit Example

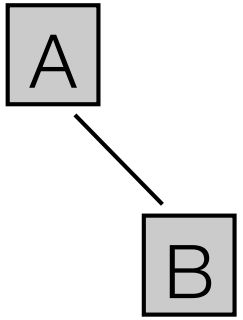


KD-Trees: Explicit Example

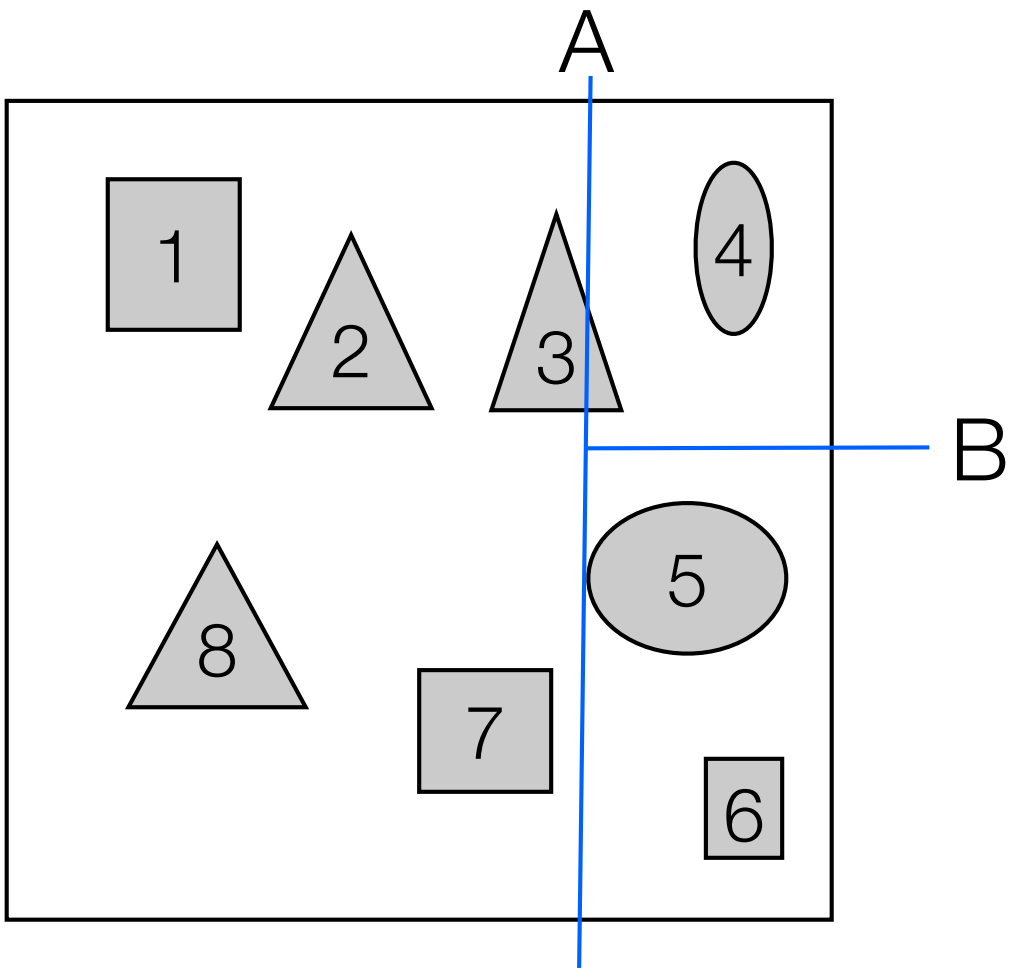


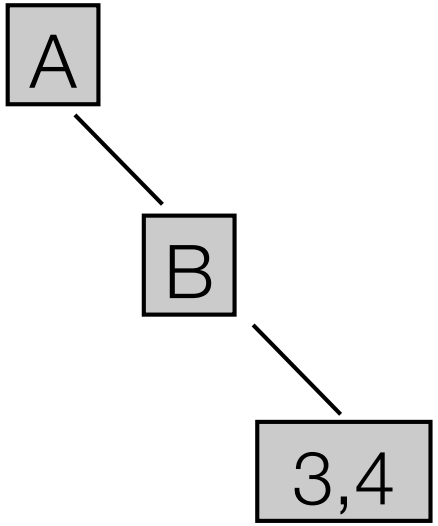
KD-Trees: Explicit Example



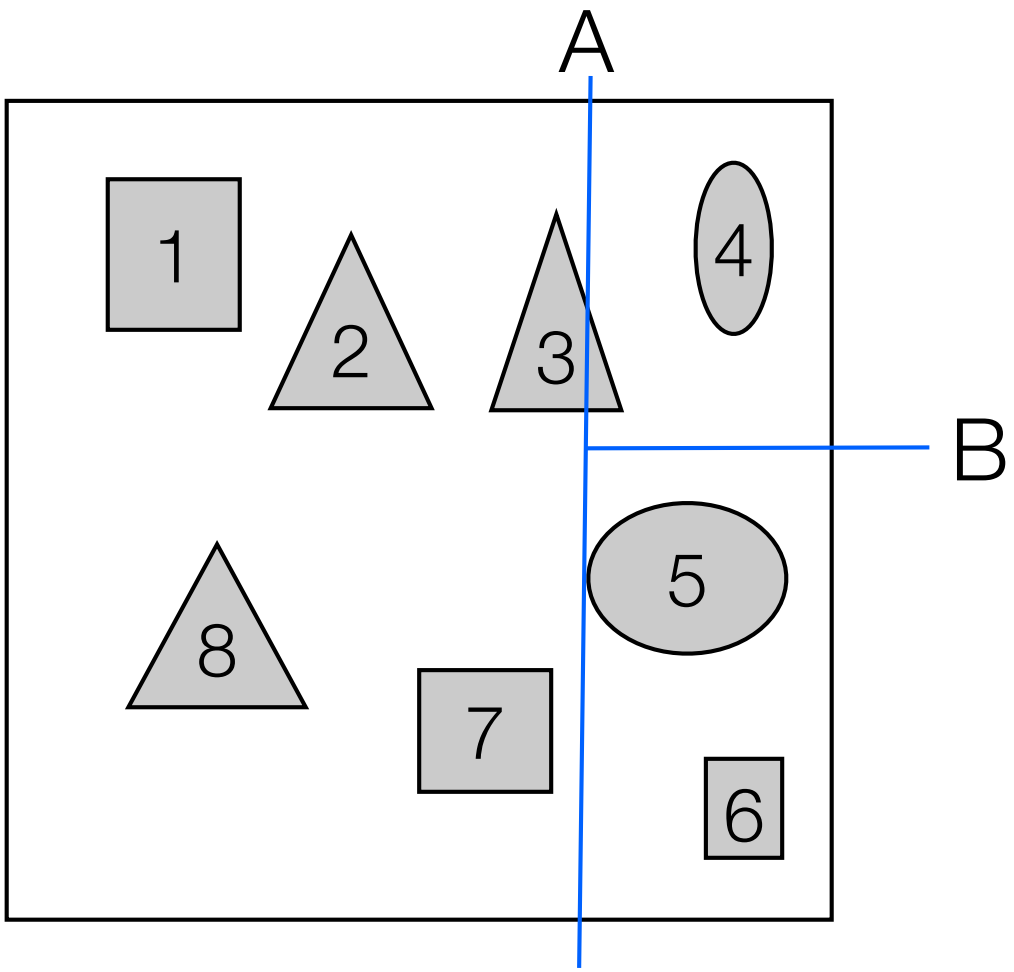


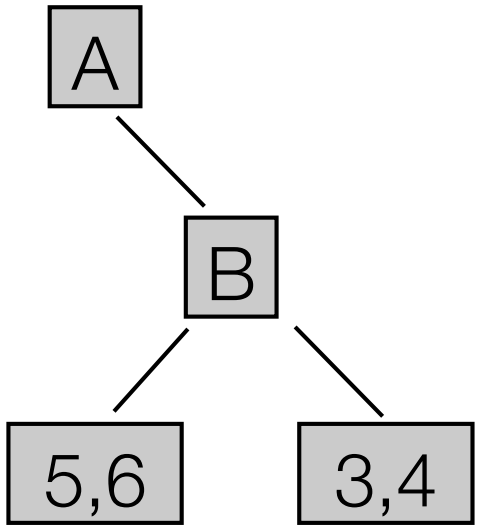
KD-Trees: Explicit Example



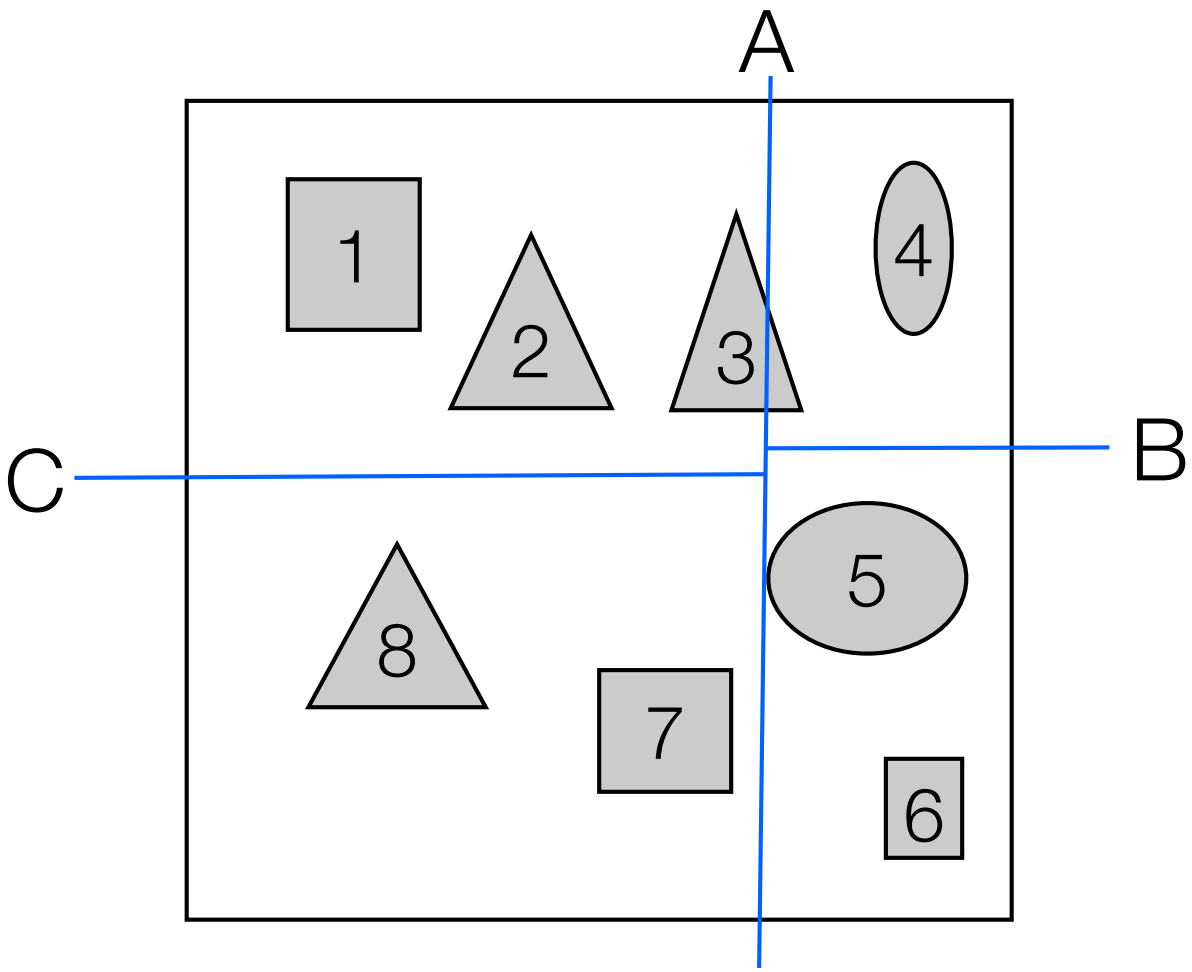


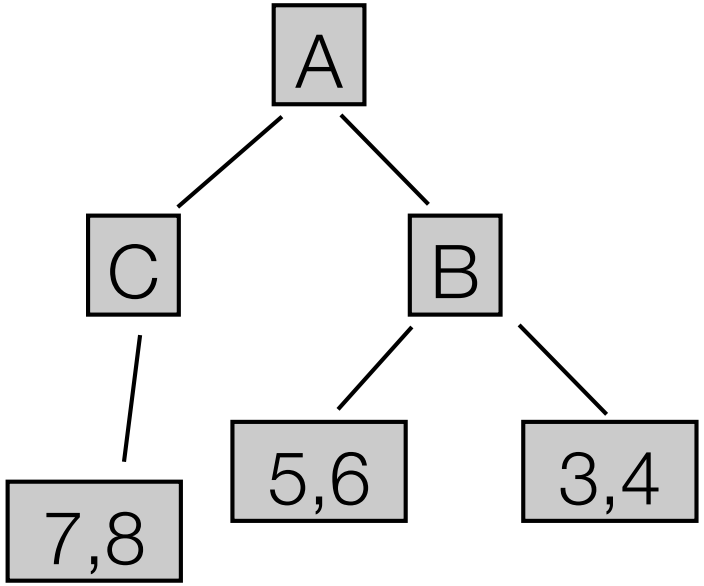
KD-Trees: Explicit Example



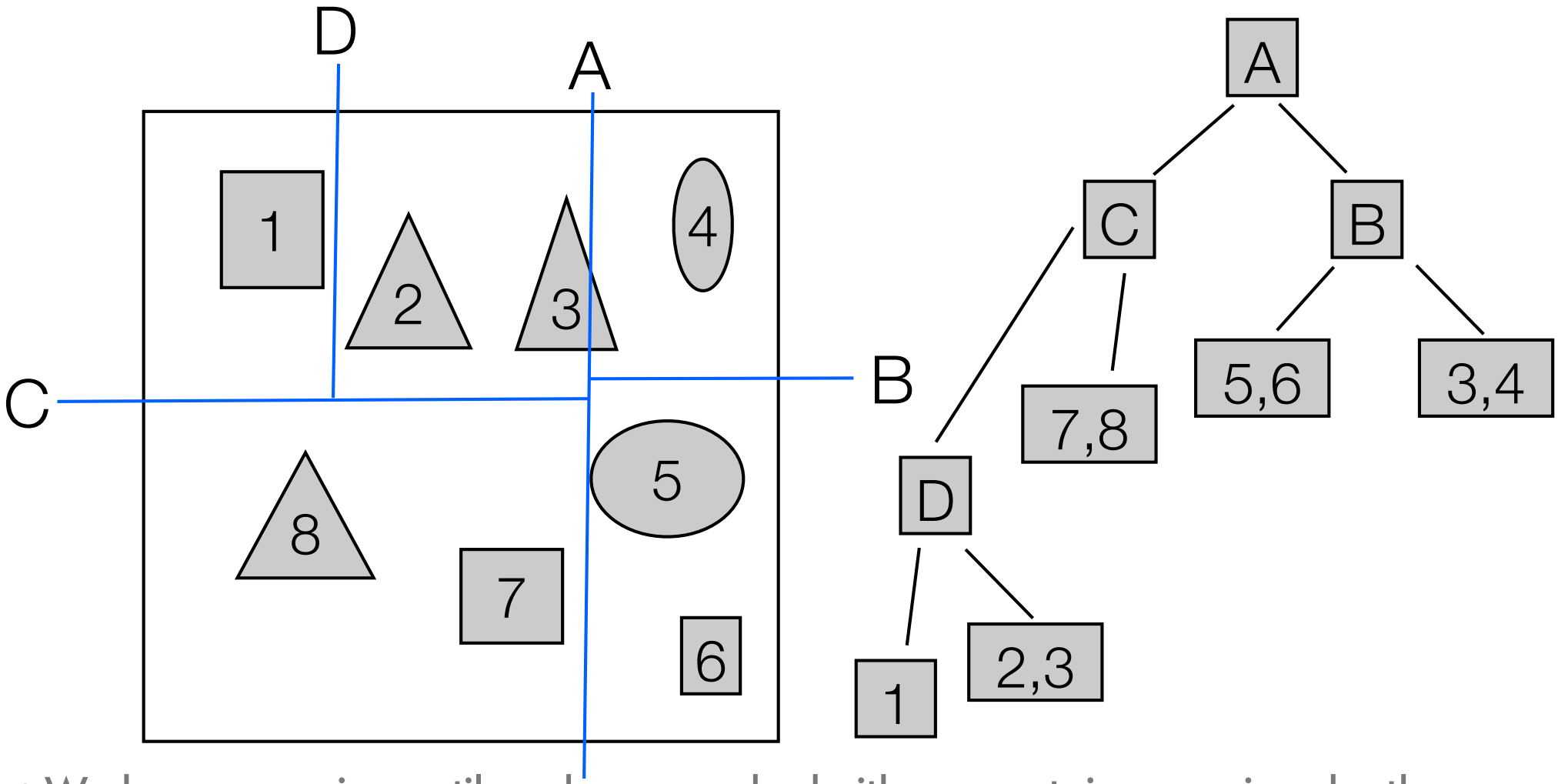


KD-Trees: Explicit Example



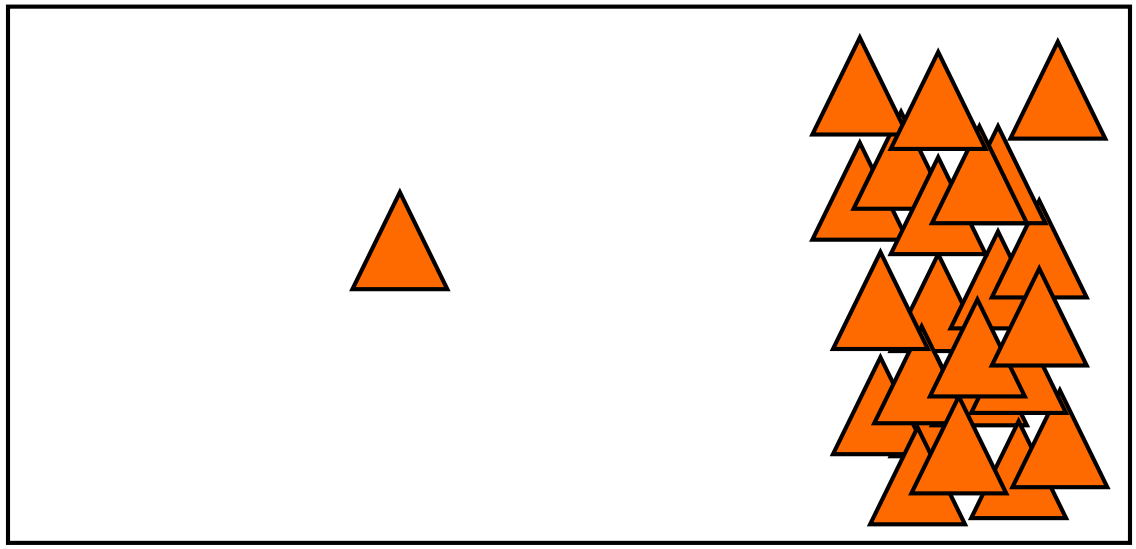


KD-Trees: Explicit Example



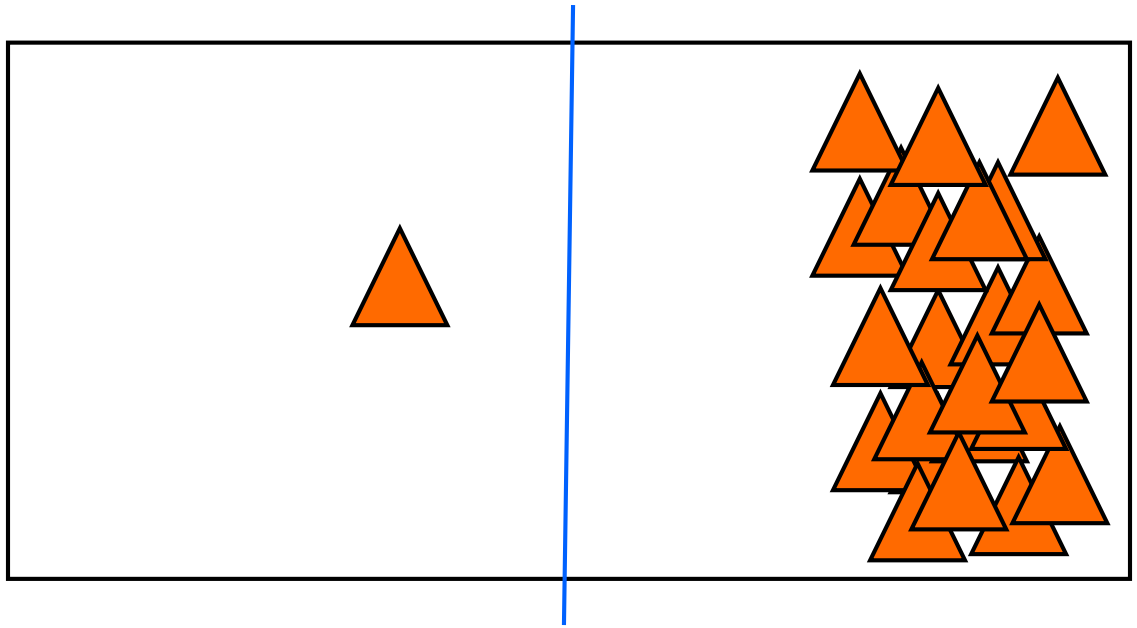
- We keep recursing until we have reached either a certain recursion depth or until we have reached a minimum number of objects per leaf node

KD-Tree Splitting Criteria



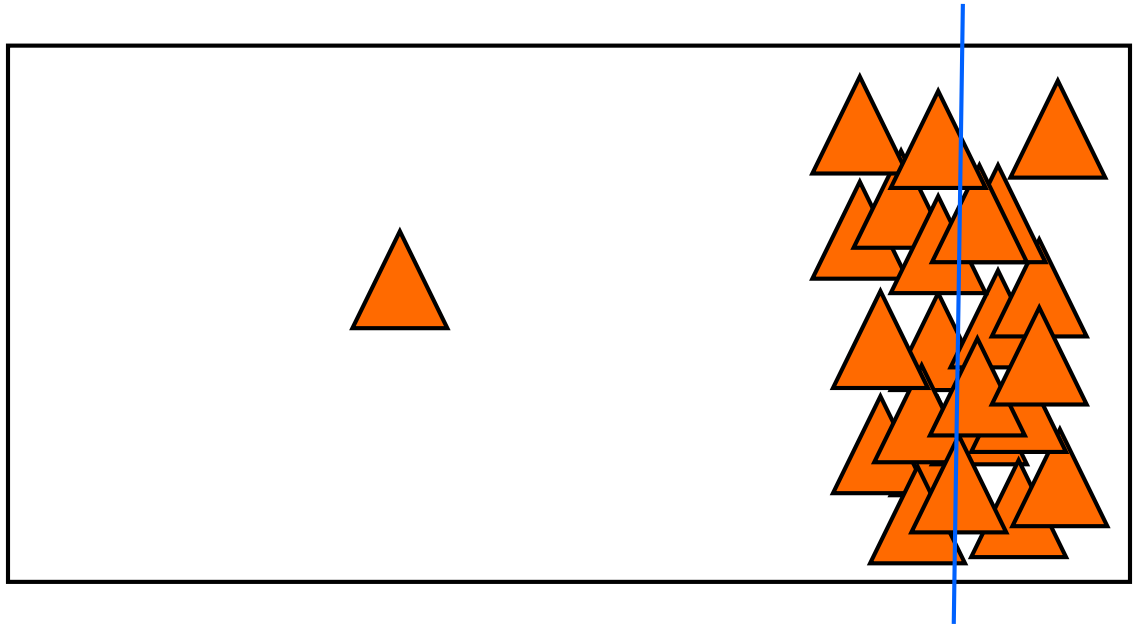
- How should we choose our splitting criteria?

KD-Tree Splitting Criteria



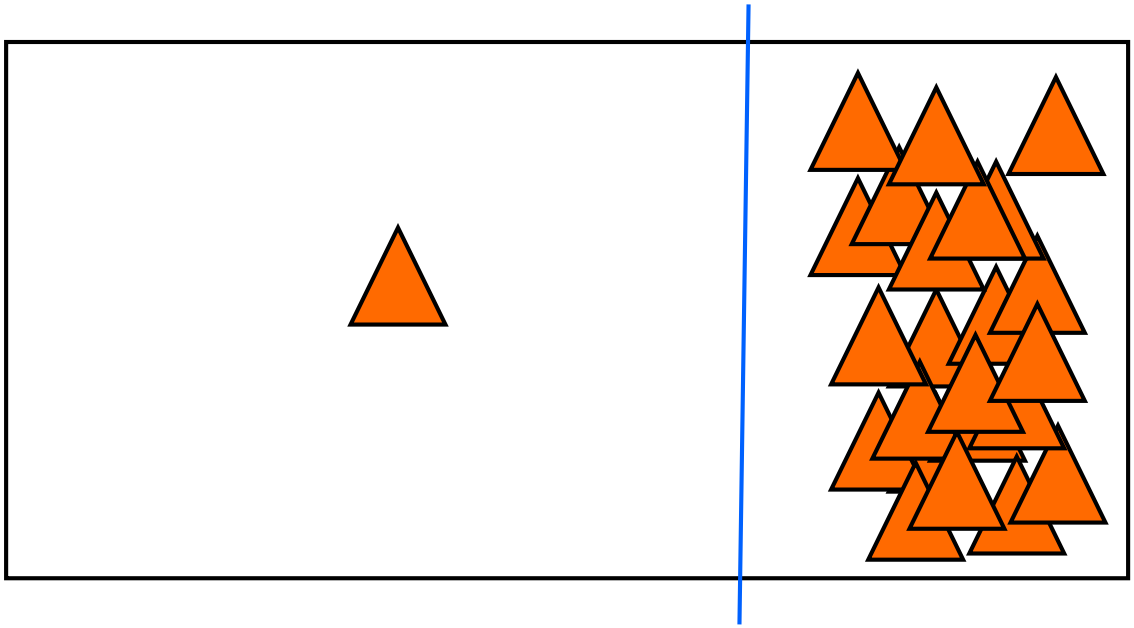
- What if we split right down the middle of the volume?
  - Pro: Probability of hitting the left and right subtrees is equal
  - Con: Does not take into account left and right costs

KD-Tree Splitting Criteria



- What if we split at the cost median?
  - Pro: The left and right costs are equal
  - Con: Does not take into account left and right hit probabilities

KD-Tree Splitting Criteria

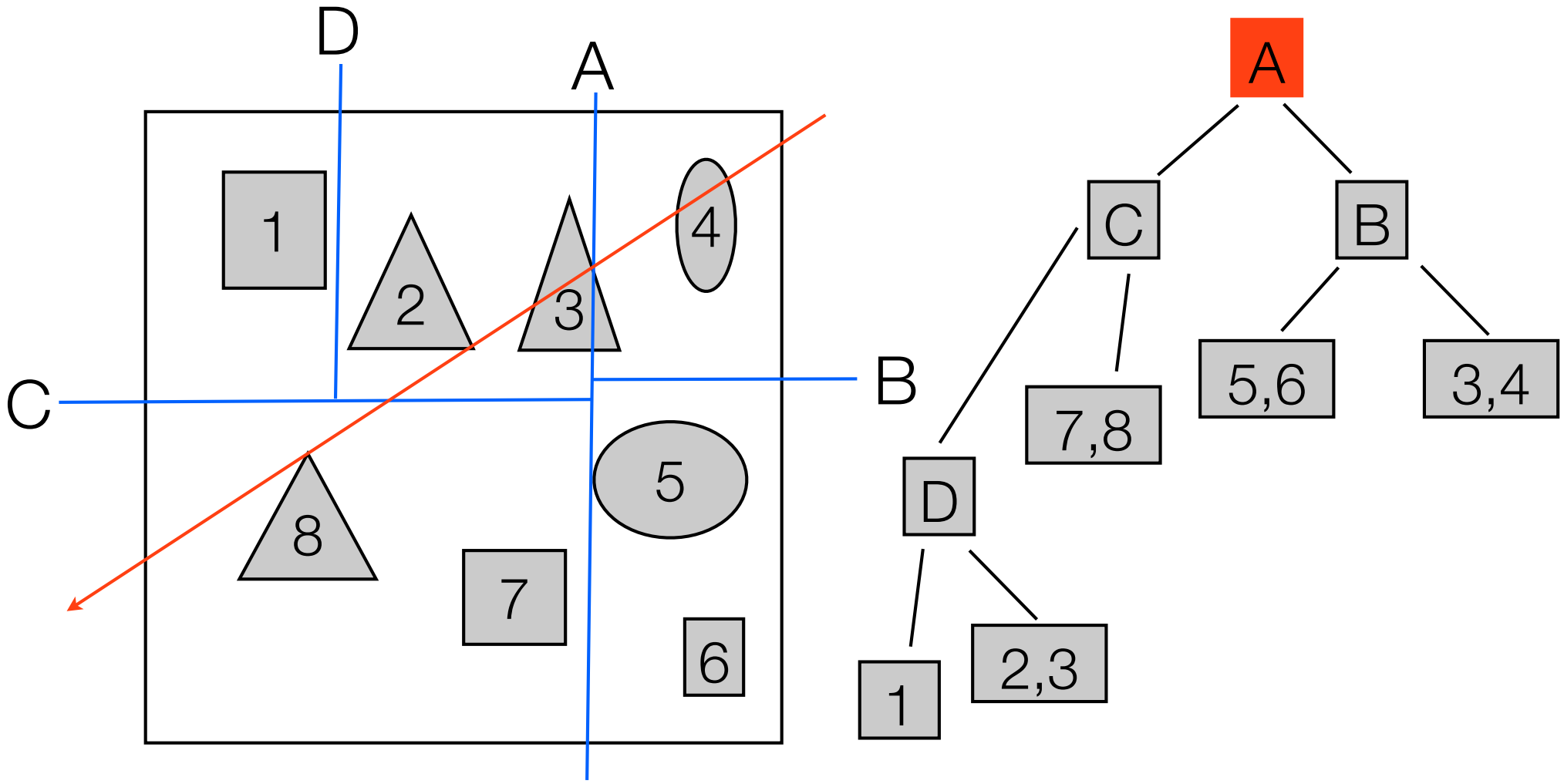


- Cost optimized split: find splits such that the probability of hitting each child weighted by the cost of each child is equal
  - Probability of hitting a node is proportional to surface area
  - Good heuristic: surface area of a node multiplied by the number of objects in the node. This is the *surface area heuristic*.

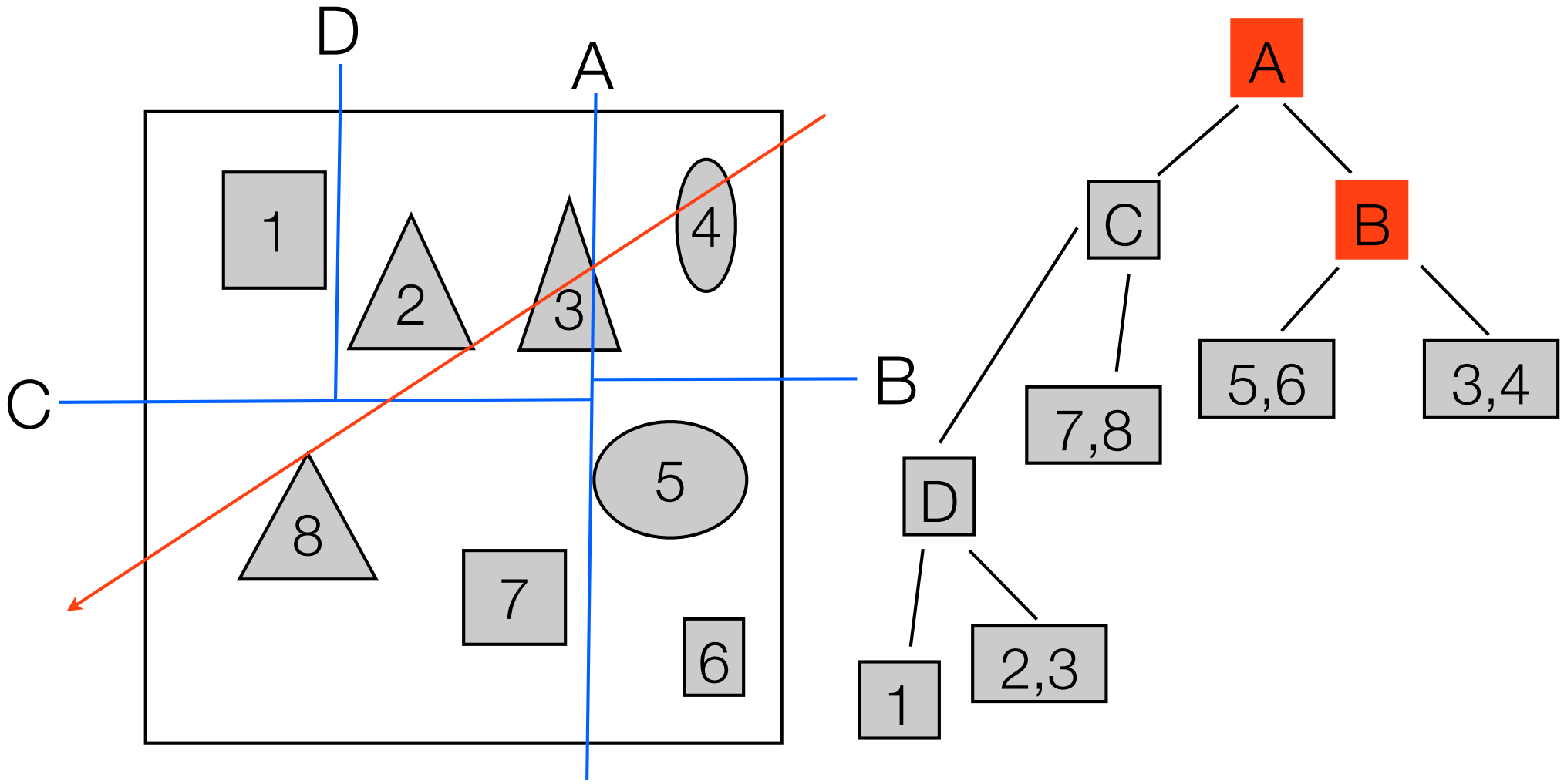
Building good KD-Trees

1. Pick an optimum split axis
2. Select a set of "candidate" split locations (can be random, or use bounding box edges, etc)
3. Sort the candidate locations via the surface area heuristic
4. Pick the candidate split with the lowest weighted cost
5. Recurse on the new children nodes until leaf nodes are reached
    - We want tall, stringy, narrow trees with a low number of objects per leaf and large empty cells
    - Ideal depth and leaf size will depend on the given scene
    - Internal nodes should contain a flag indicating leaf or internal status, a split axis, the split location, and pointers to the children

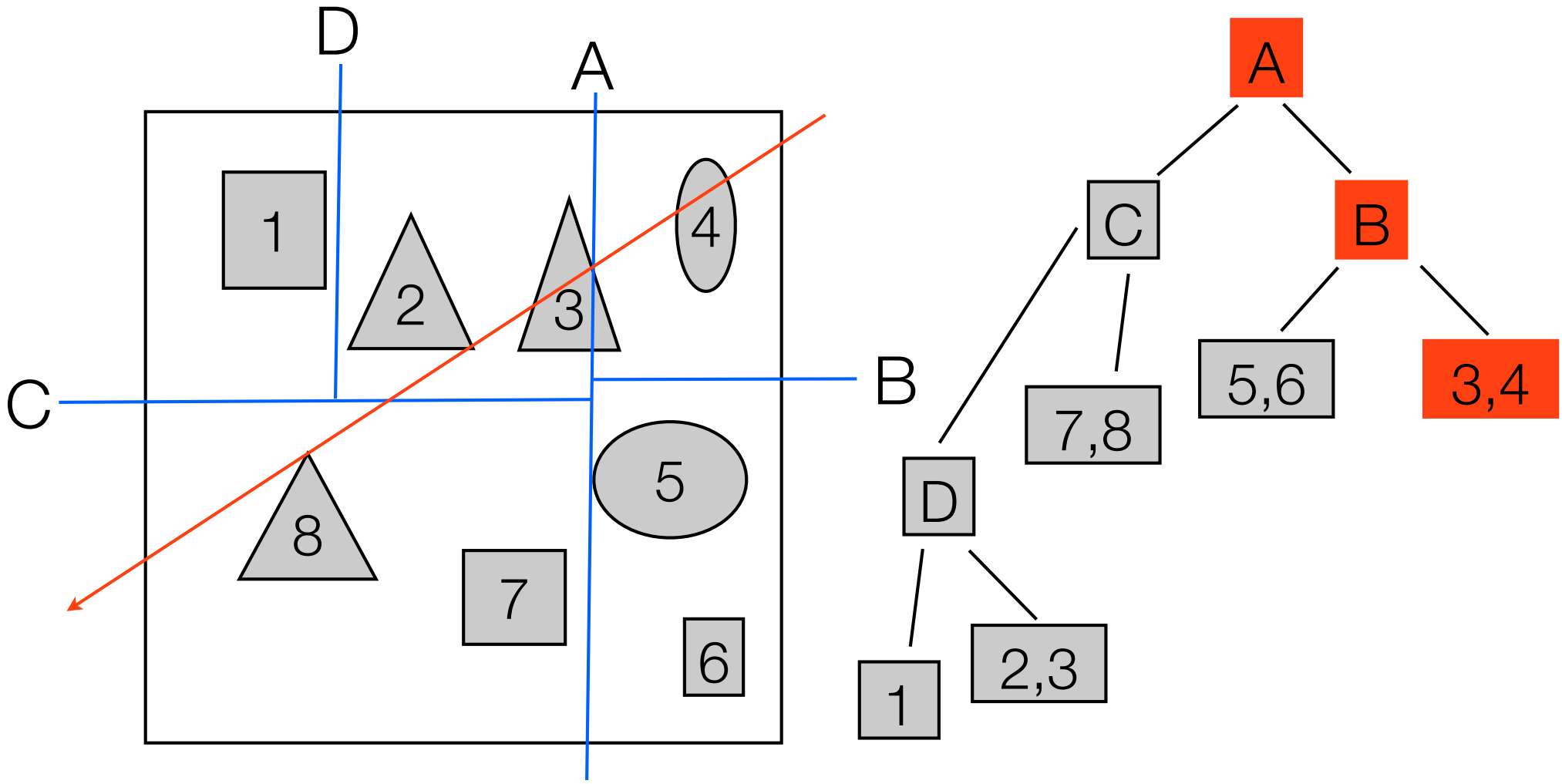
KD-Trees: Normal Traversal



KD-Trees: Normal Traversal

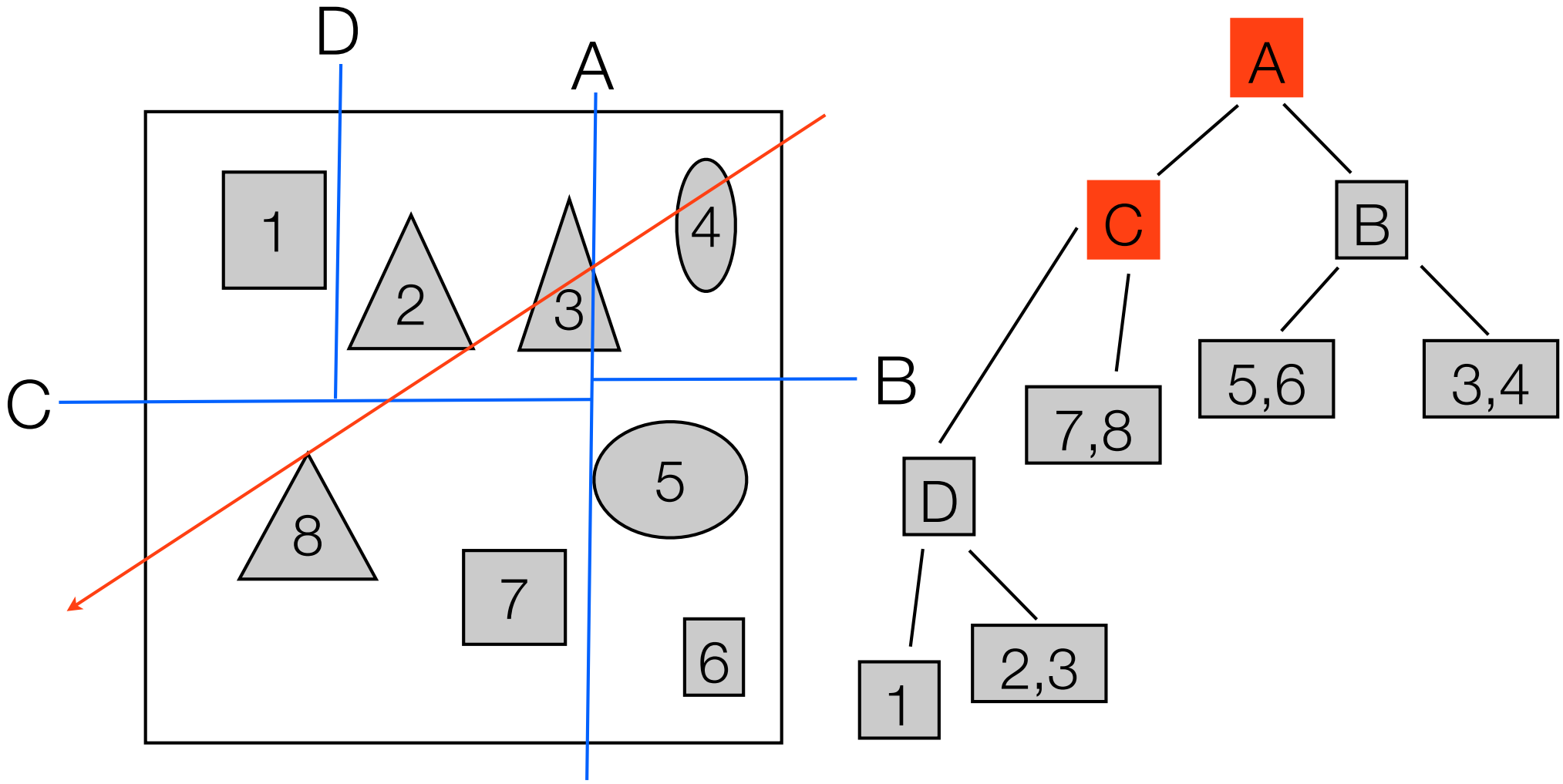


KD-Trees: Normal Traversal



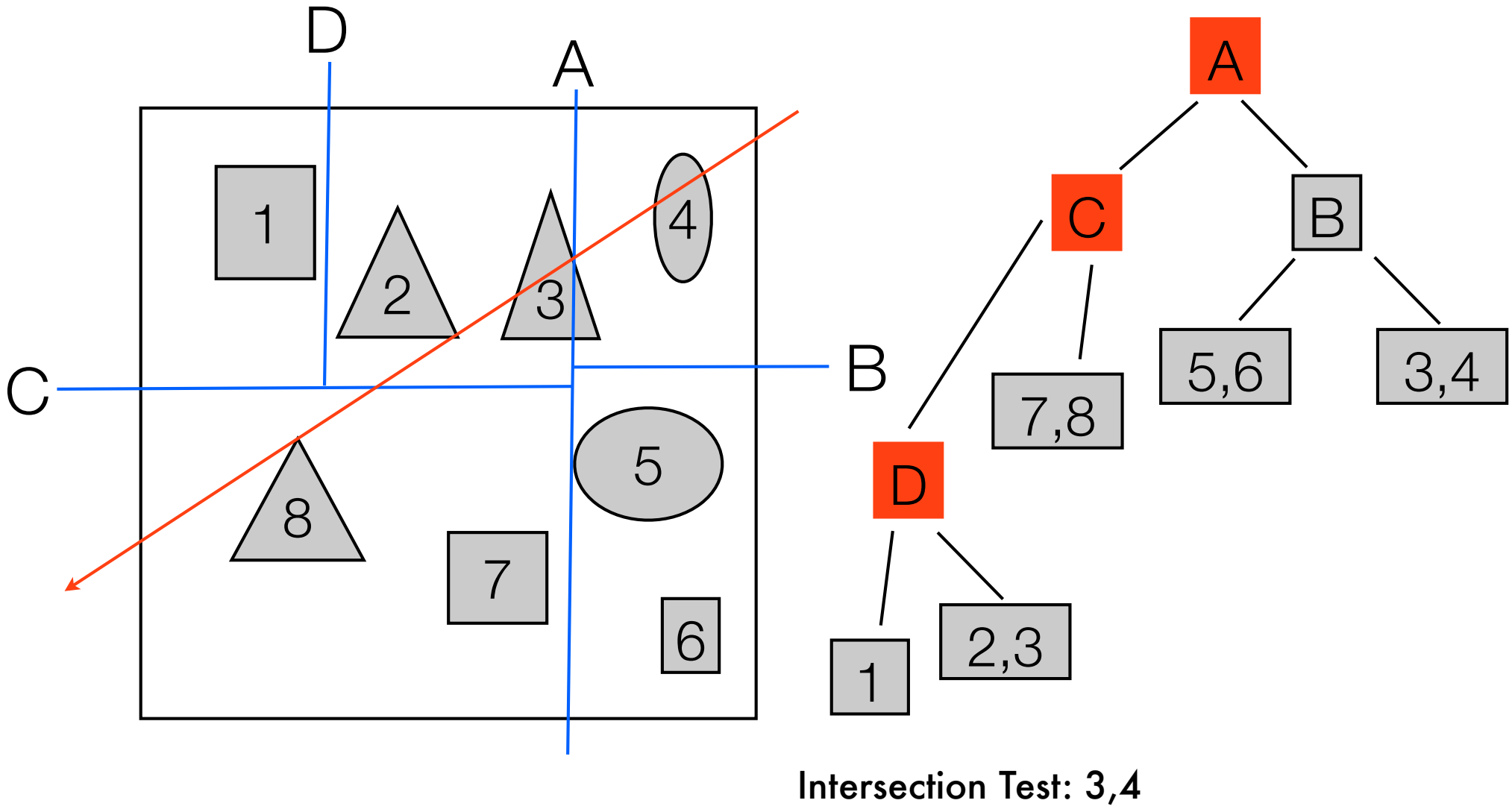
Intersection Test: 3,4

KD-Trees: Normal Traversal



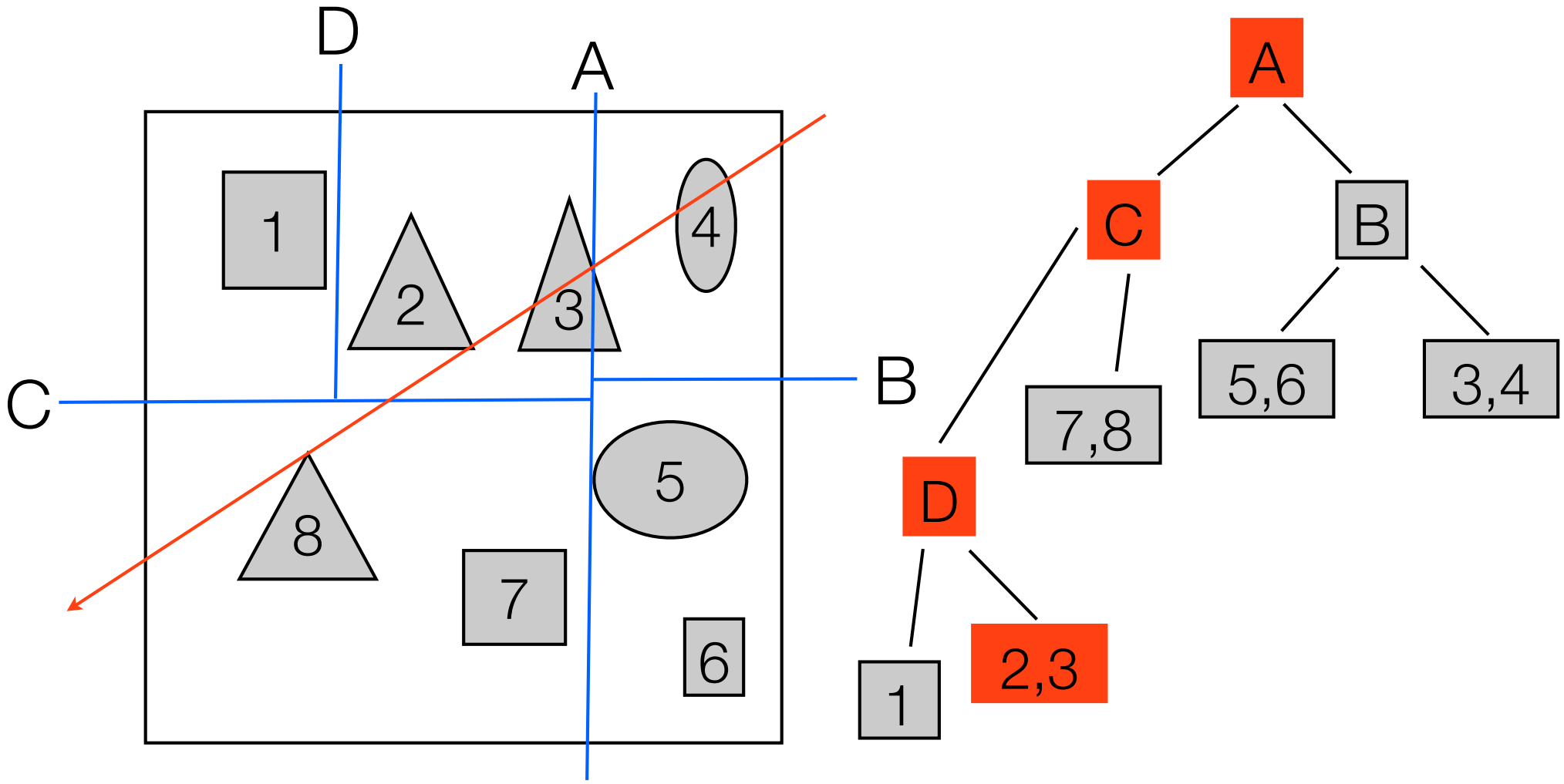
Intersection Test: 3,4

KD-Trees: Normal Traversal



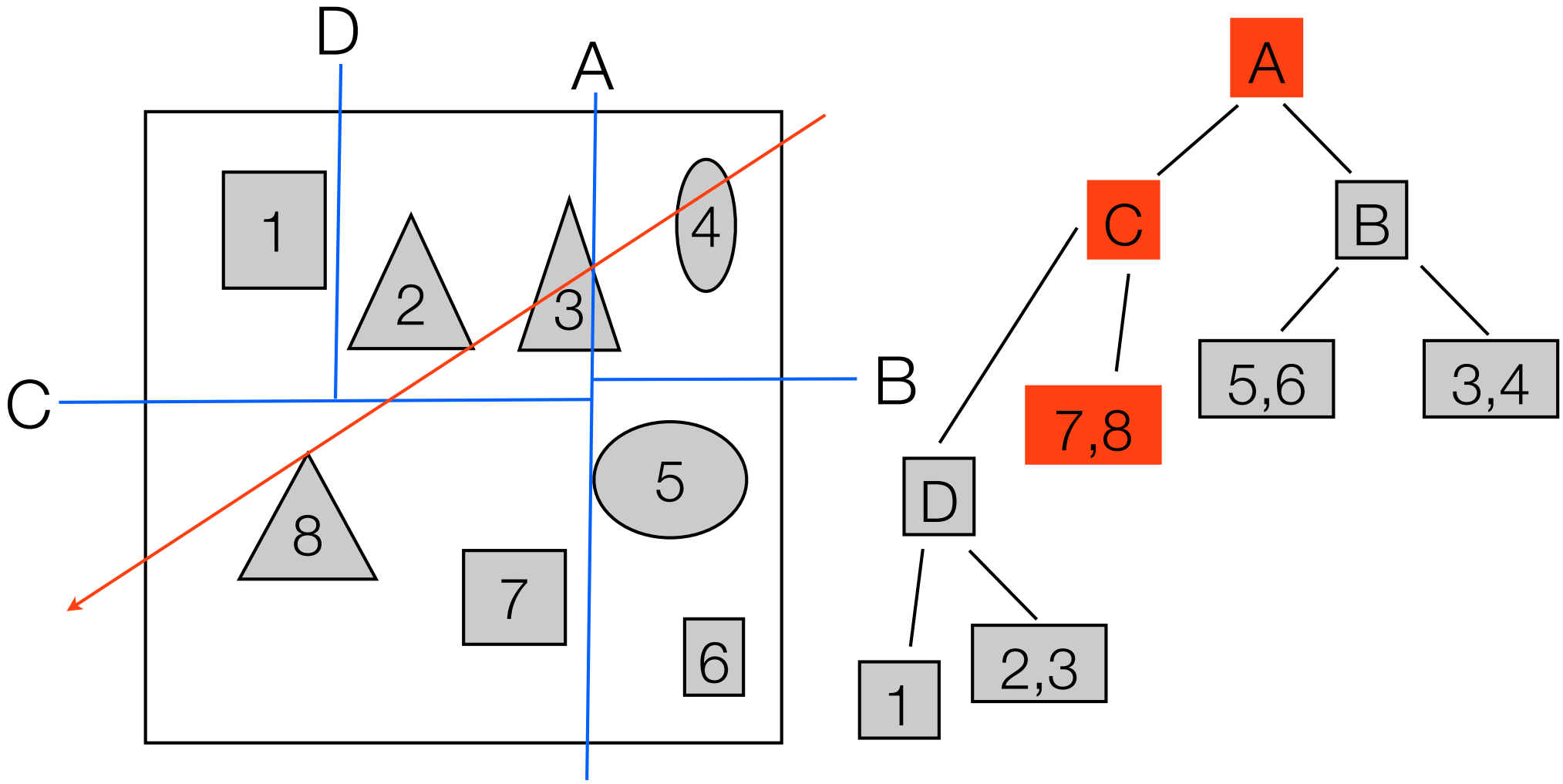
Intersection Test: 3,4

KD-Trees: Normal Traversal



Intersection Test: 3,4,2,3

KD-Trees: Normal Traversal



Intersection Test: 3,4,2,3,7,8

KD-Trees: Normal Traversal

Normally, KD-Tree traversal is implemented recursively, but on the GPU we have no stack and no recursion…

Stackless KD-Trees

- There are several possible ways to approach KD-Trees on the GPU:
  - Create a pseudo-stack
  - Rope-based KD-Trees (Popov et. al. 2007)
  - Breadth-first-search construction (Zhou et. al. 2008)
  - Finite state machine based

Stackless KD-Trees

- There are several possible ways to approach KD-Trees on the GPU:
  - Create a pseudo-stack
  - Rope-based KD-Trees (Popov et. al. 2007)
  - Breadth-first-search construction (Zhou et. al. 2008)
  - Finite state machine based

Why is the pseudo-stack a bad idea?

Stackless KD-Trees

- There are several possible ways to approach KD-Trees on the GPU:
  - Create a pseudo-stack
  - Rope-based KD-Trees (Popov et. al. 2007) We'll look at this one today
  - Breadth-first-search construction (Zhou et. al. 2008)
  - Finite state machine based

The other ones would make great final project ideas!

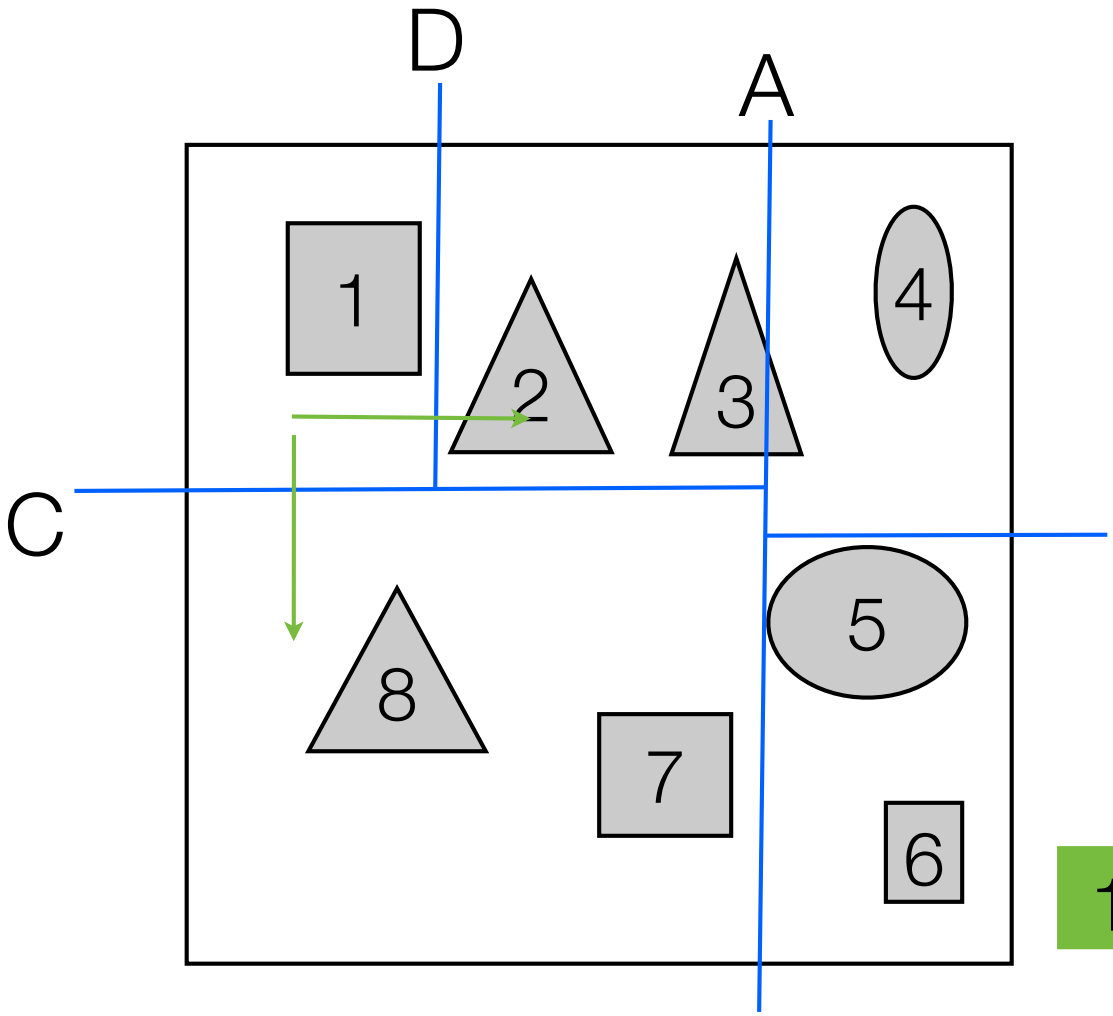
Stackless KD-Trees: Rope KD-Trees

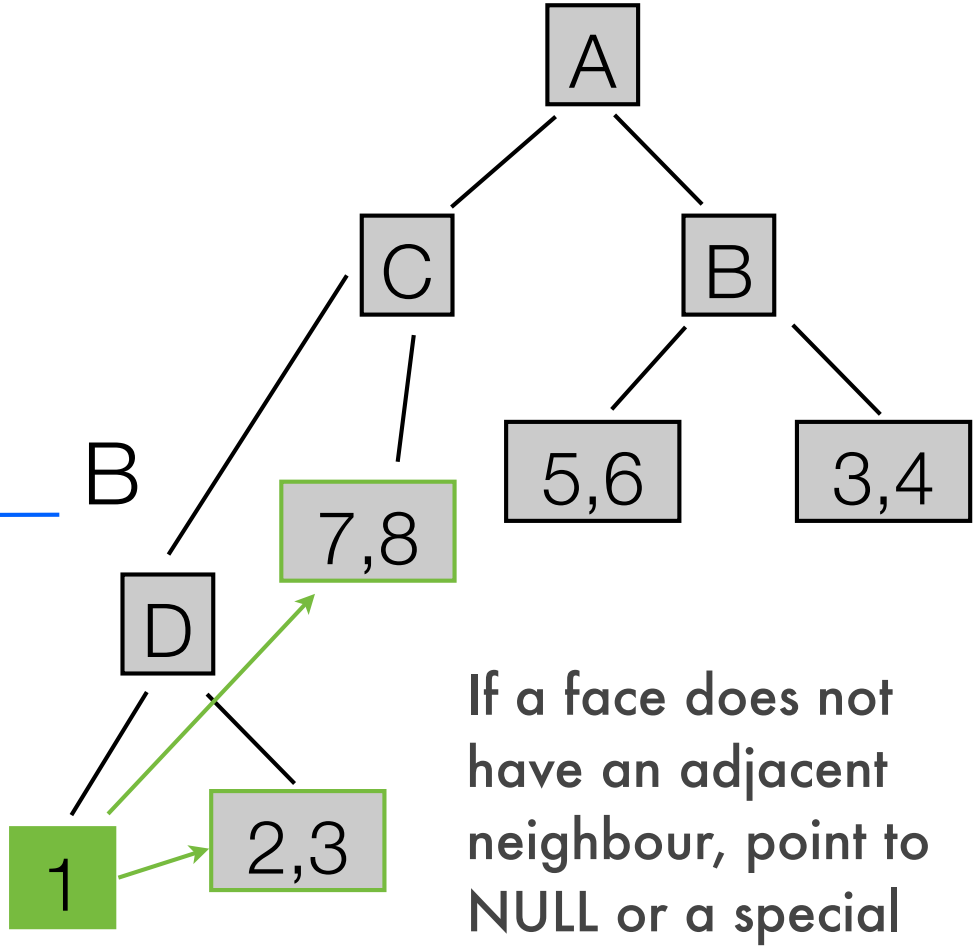
- The goal of KD-Tree traversal is to create an ordered front-to-back list of objects that we need to intersection test
  - Normally we do this by traversing up and down the KD-Tree, which requires a stack to keep track of where we have been in the tree
  - What if we can get rid of the up traversal?

Stackless KD-Trees: Rope KD-Trees

- The goal of KD-Tree traversal is to create an ordered front-to-back list of objects that we need to intersection test
  - Normally we do this by traversing up and down the KD-Tree, which requires a stack to keep track of where we have been in the tree
  - What if we can get rid of the up traversal?
  - Solution: have each node store pointers of its children or contents, AND store pointers to all six of its adjacent neighbours ("ropes")

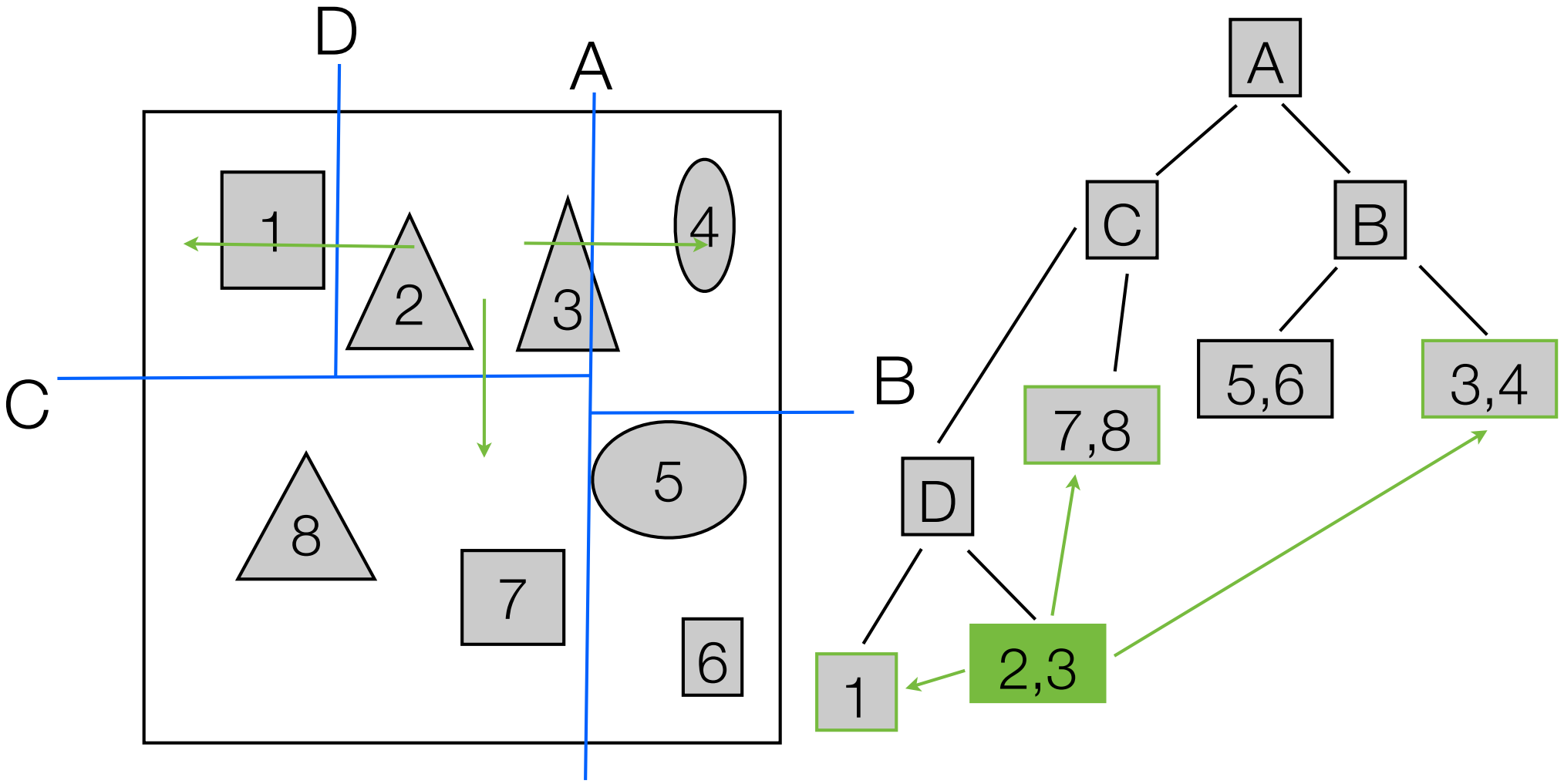
Stackless KD-Trees: Rope KD-Trees



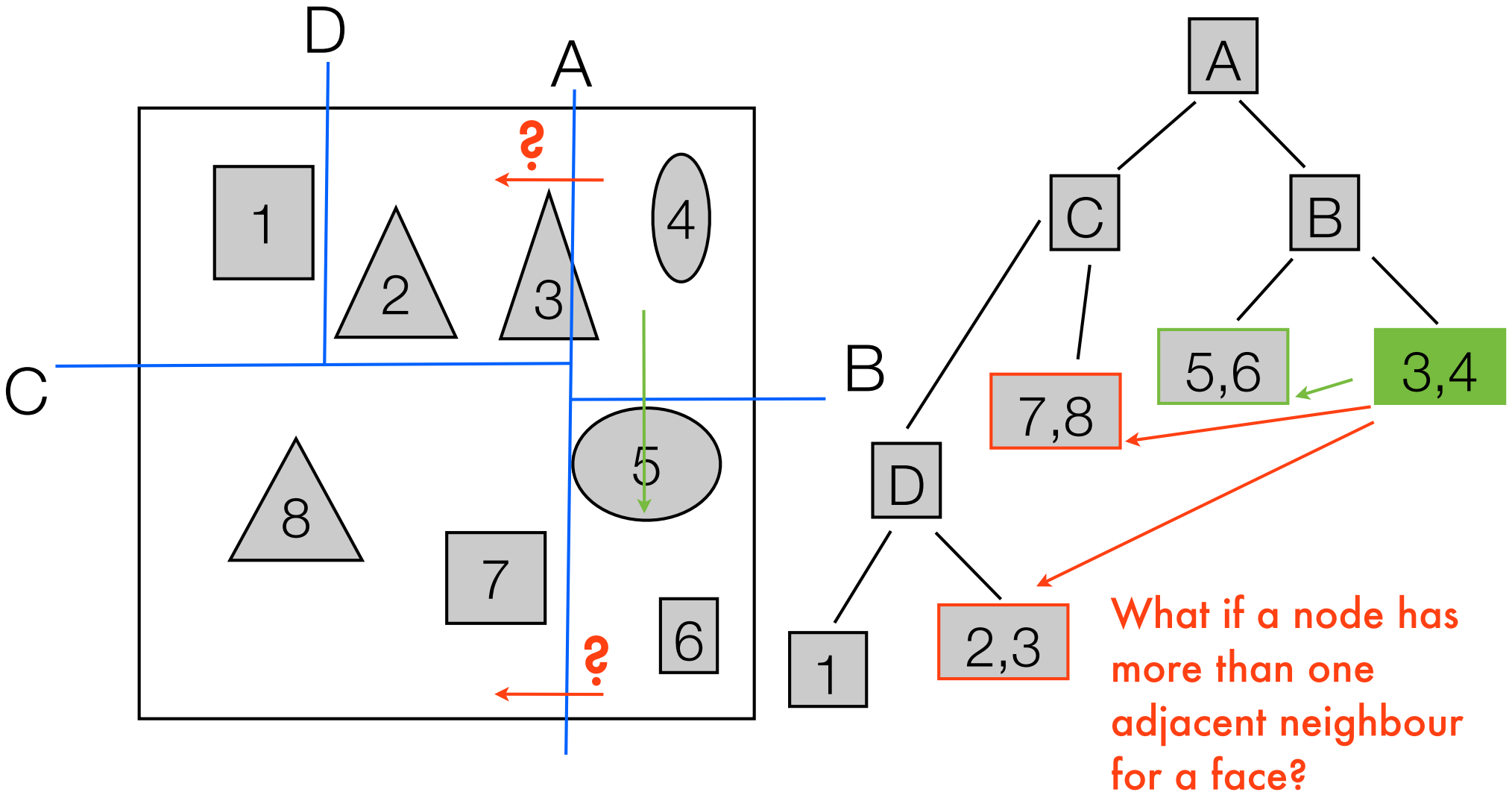


If a face does not have an adjacent neighbour, point to NULL or a special empty node

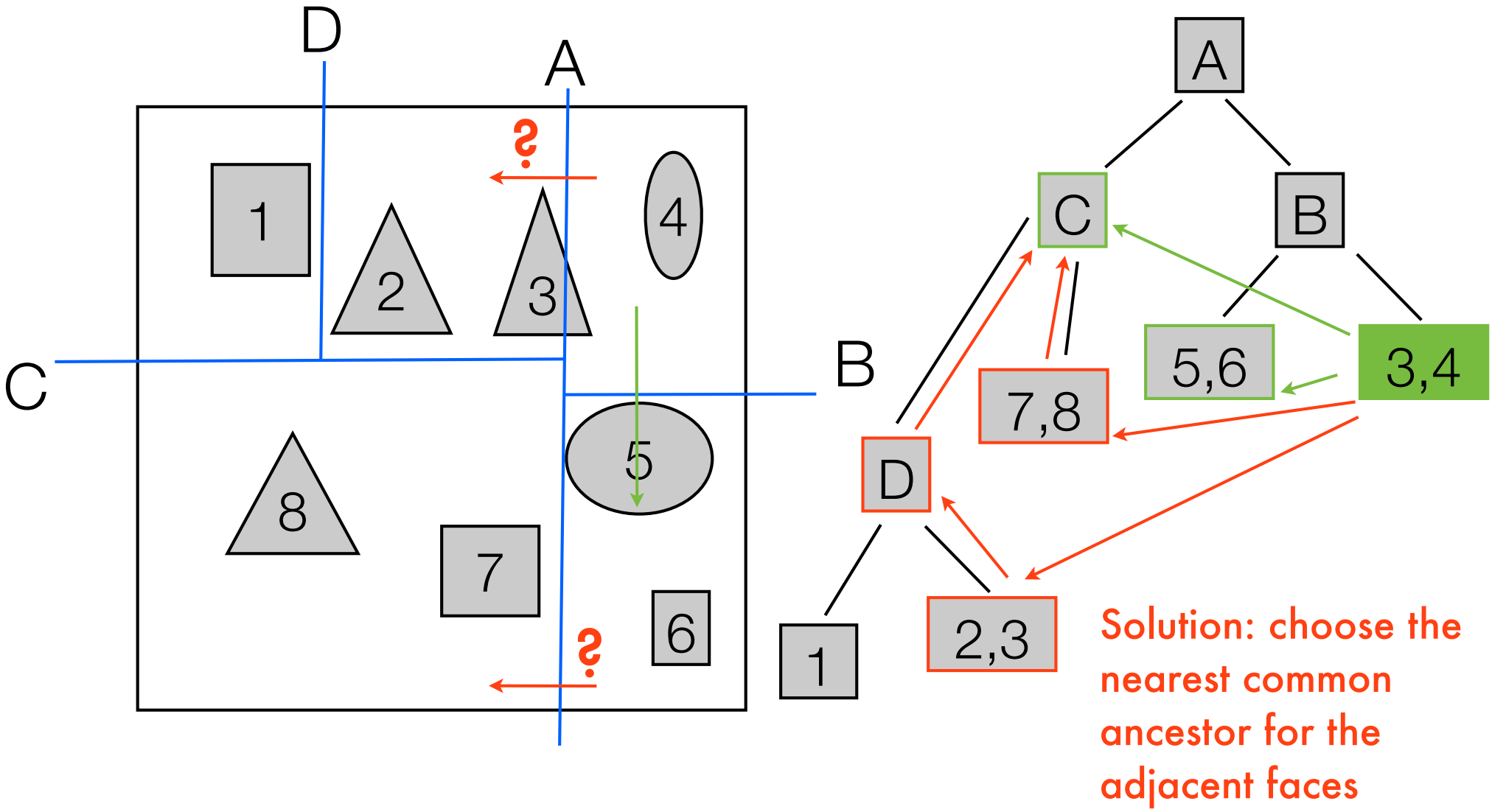
Stackless KD-Trees: Rope KD-Trees



Stackless KD-Trees: Rope KD-Trees



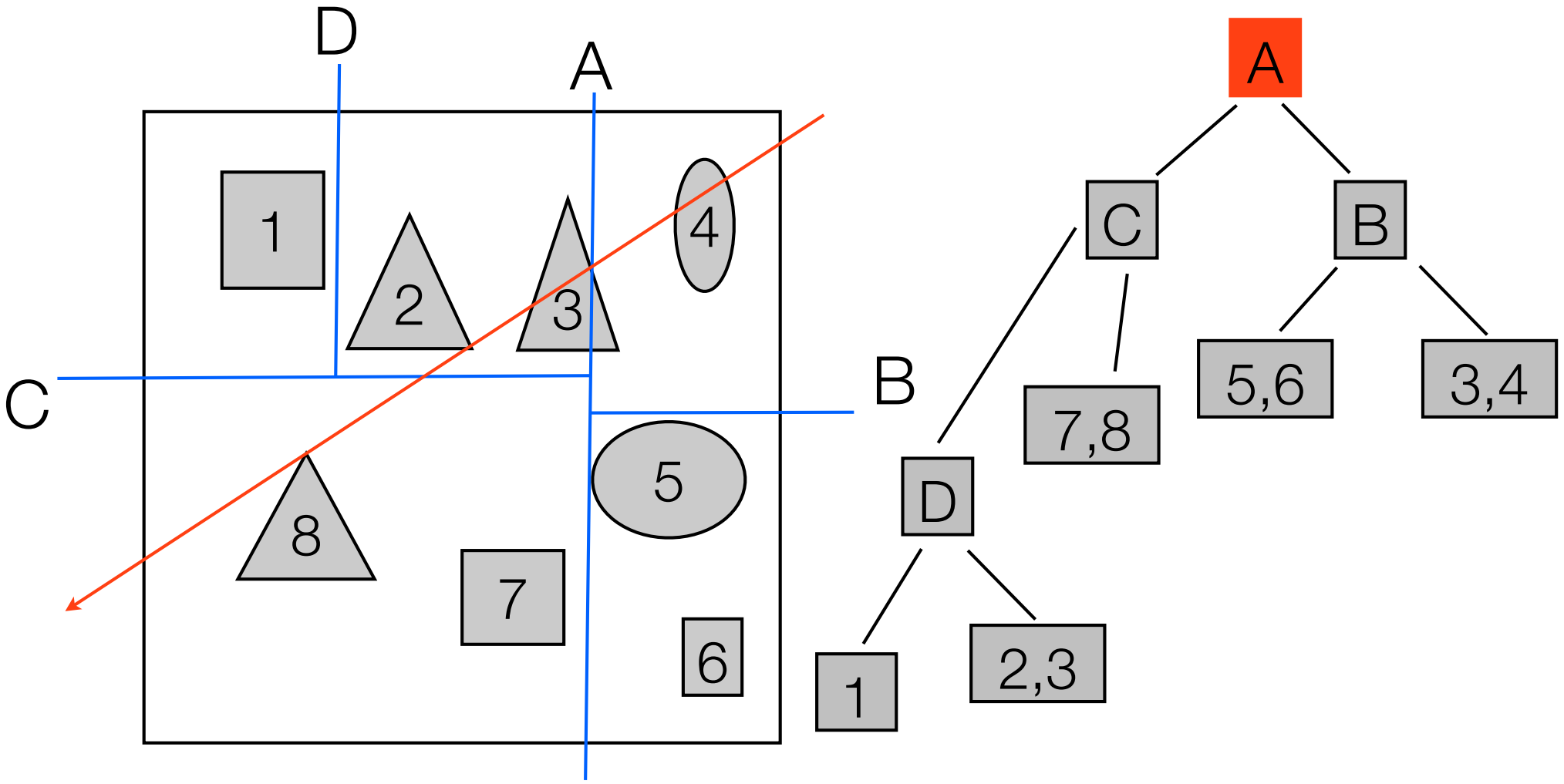
Stackless KD-Trees: Rope KD-Trees



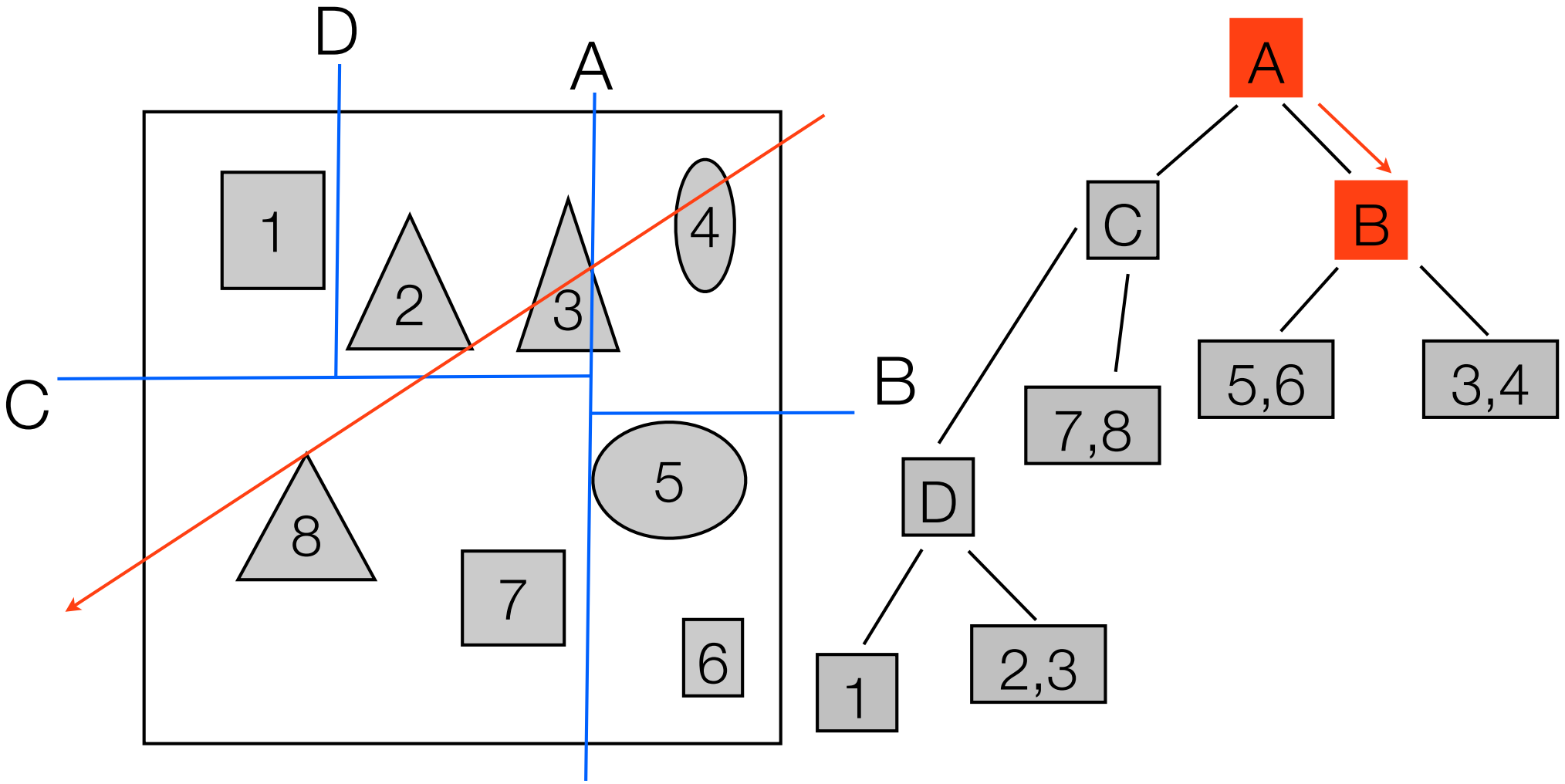
Stackless KD-Trees: Rope KD-Trees

- Since we now know what nodes are adjacent to any given node, we can now traverse without having to unwind previous steps of our path!
- Upon reaching a leaf node, instead of going back up to the parent, we examine which face of the node the ray exits through and follow that rope to the neighbour node

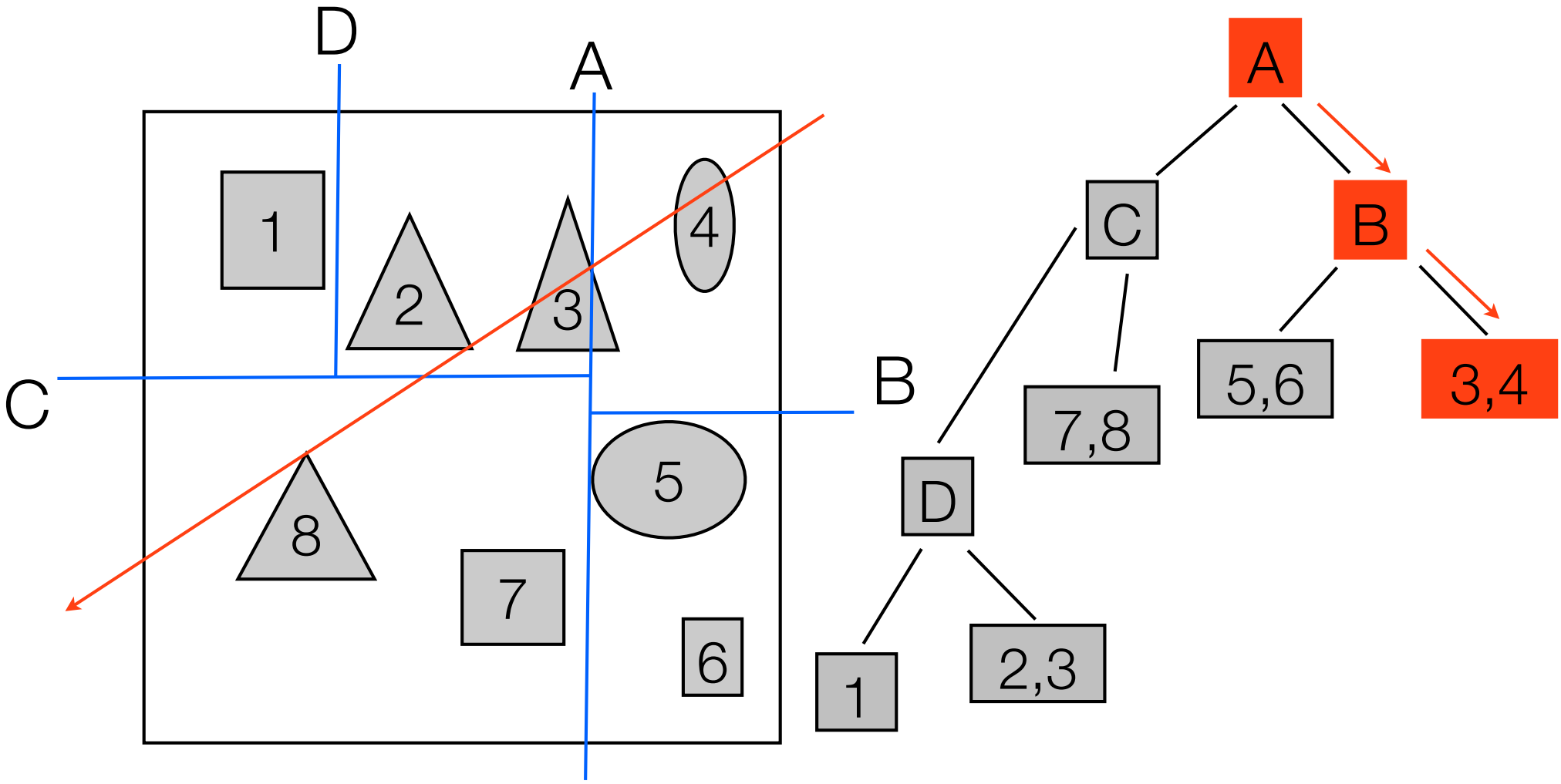
Stackless KD-Trees: Traversal



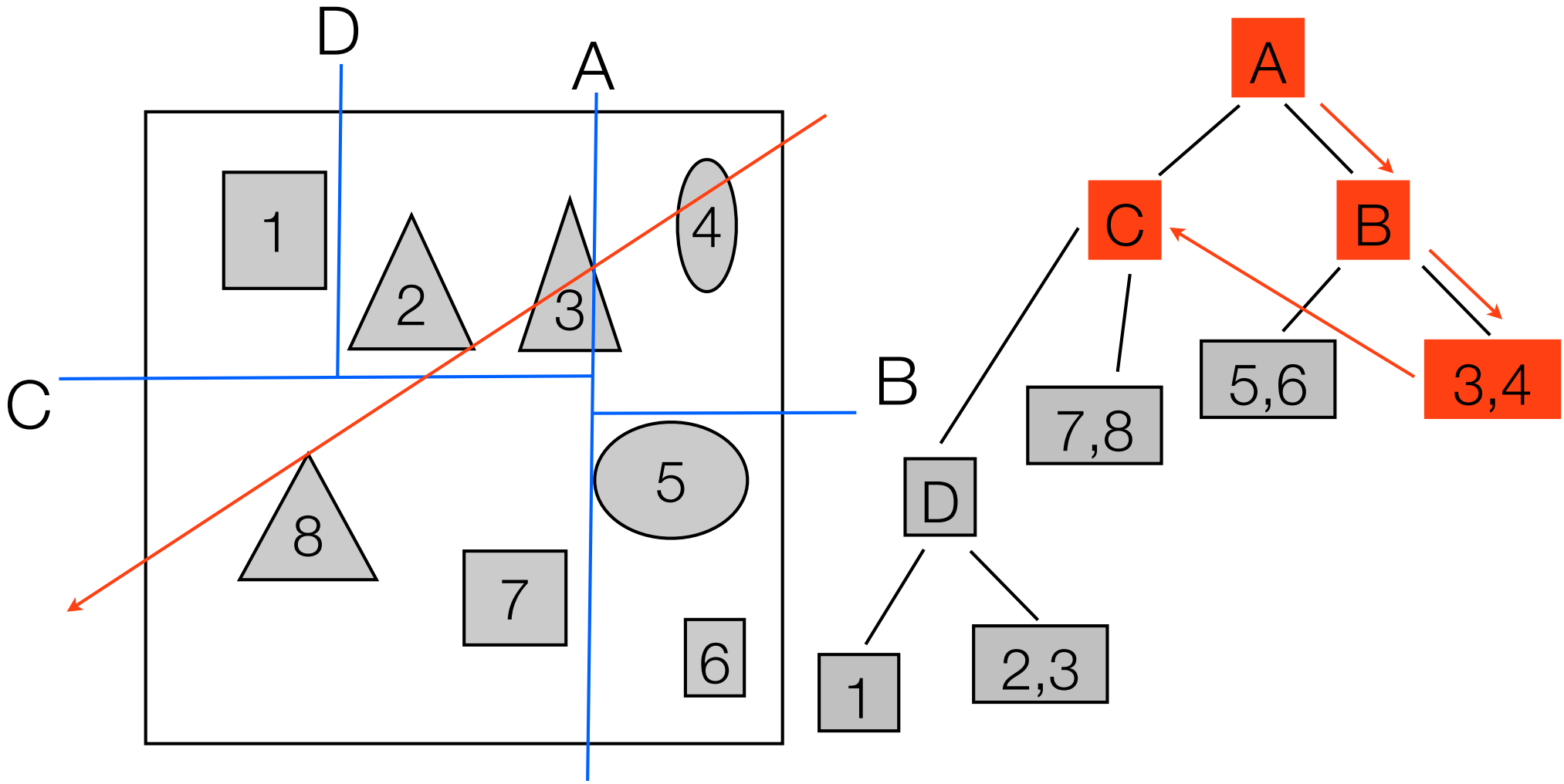
Stackless KD-Trees: Traversal



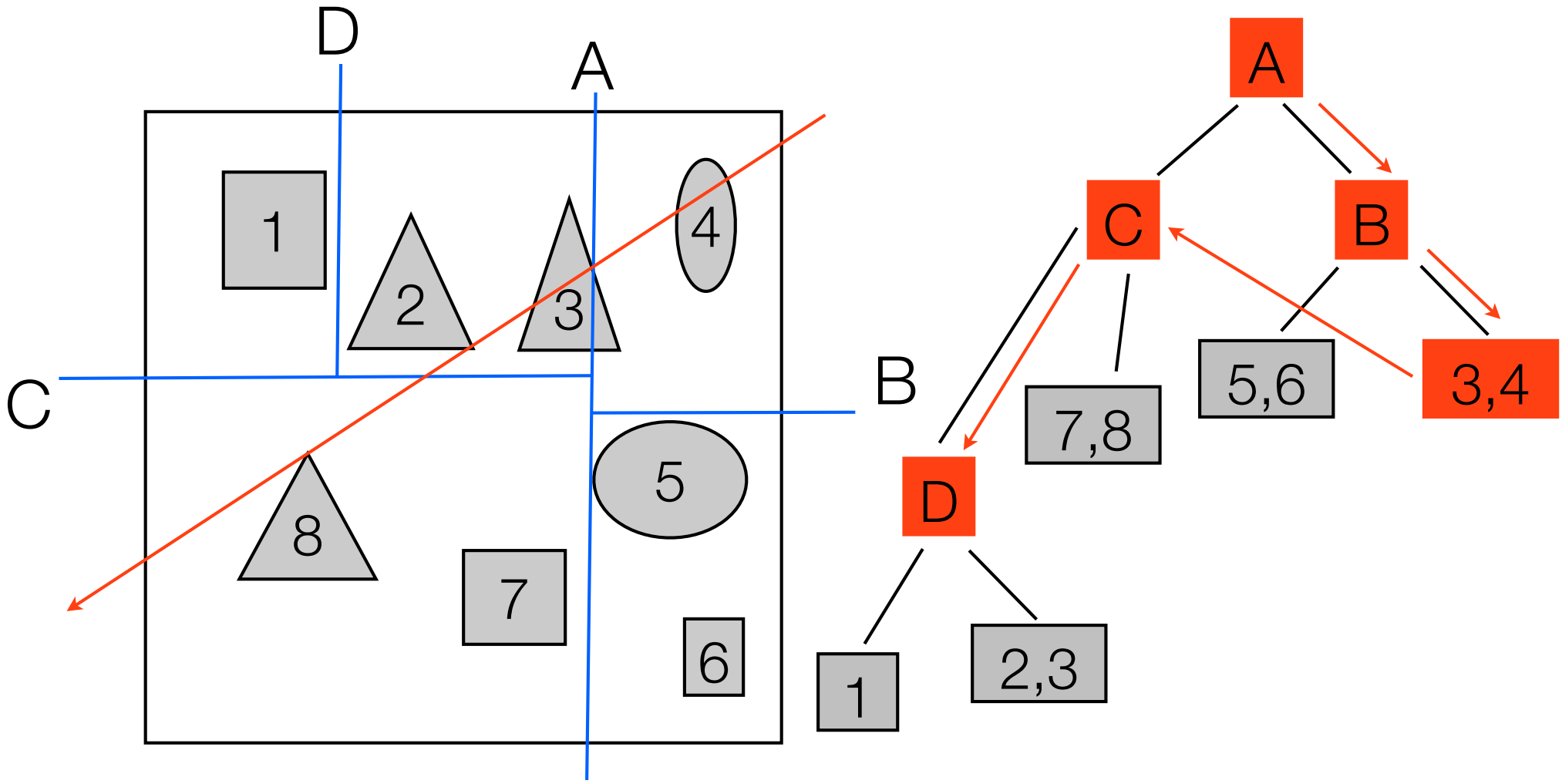
Stackless KD-Trees: Traversal



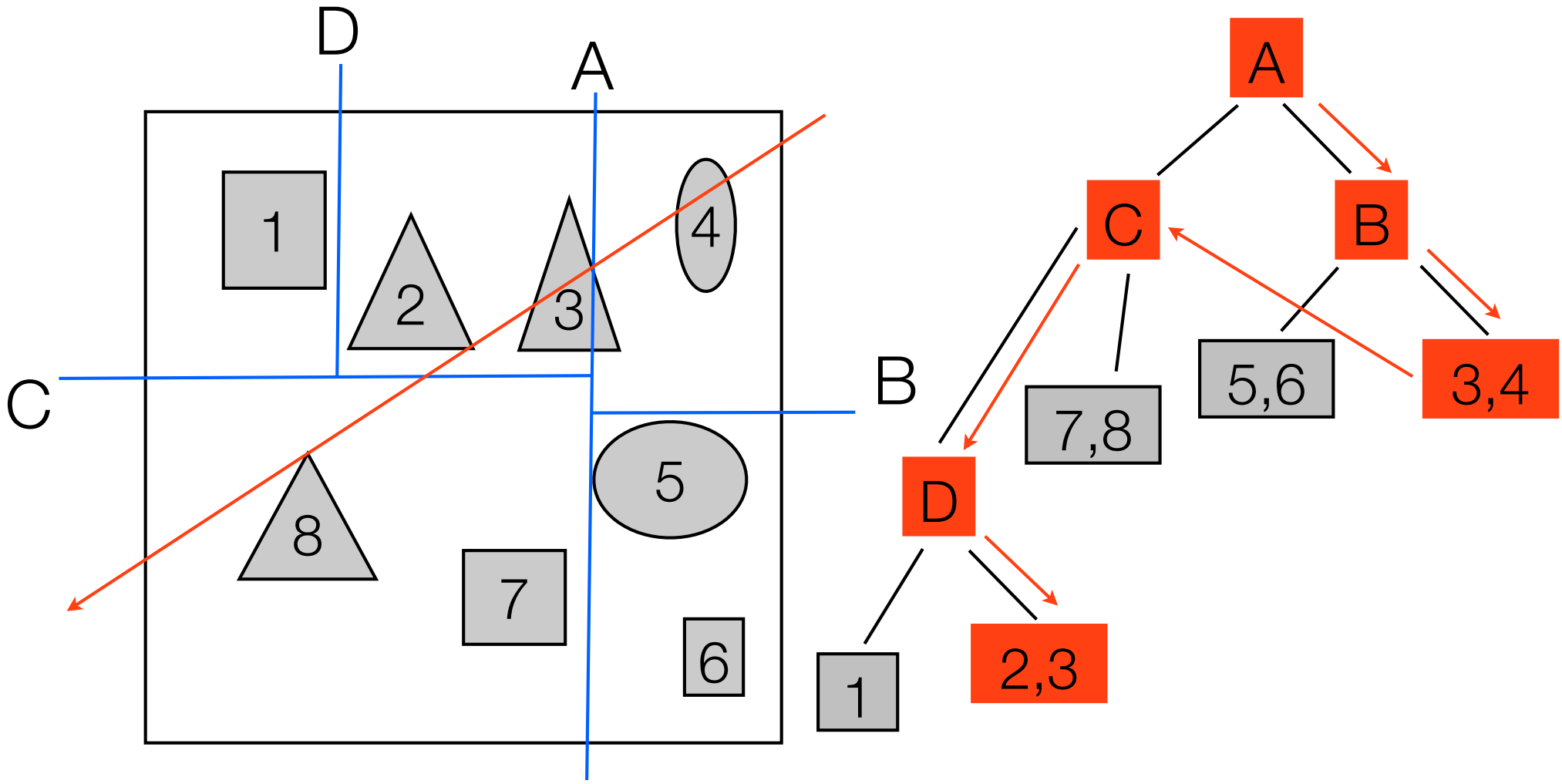
Stackless KD-Trees: Traversal



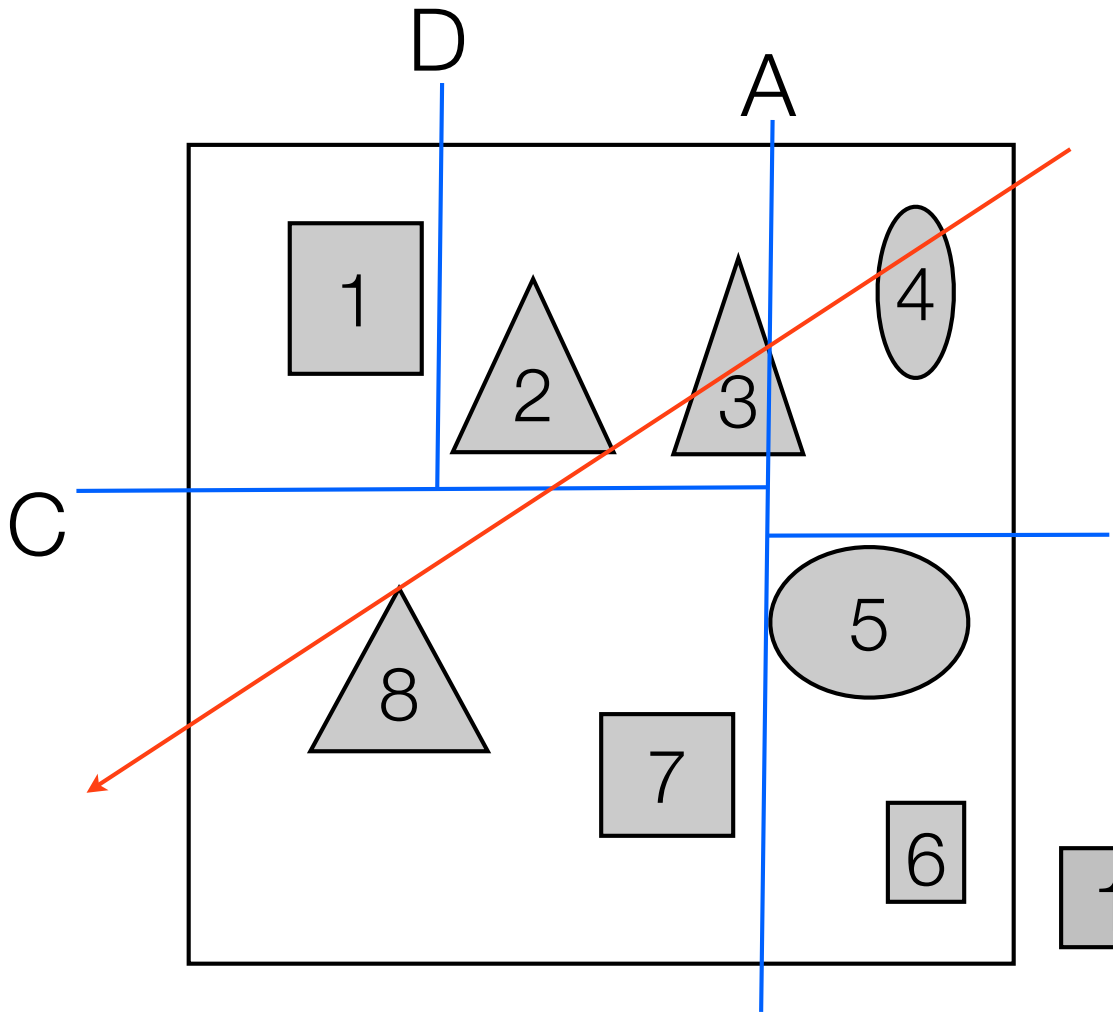
Stackless KD-Trees: Traversal

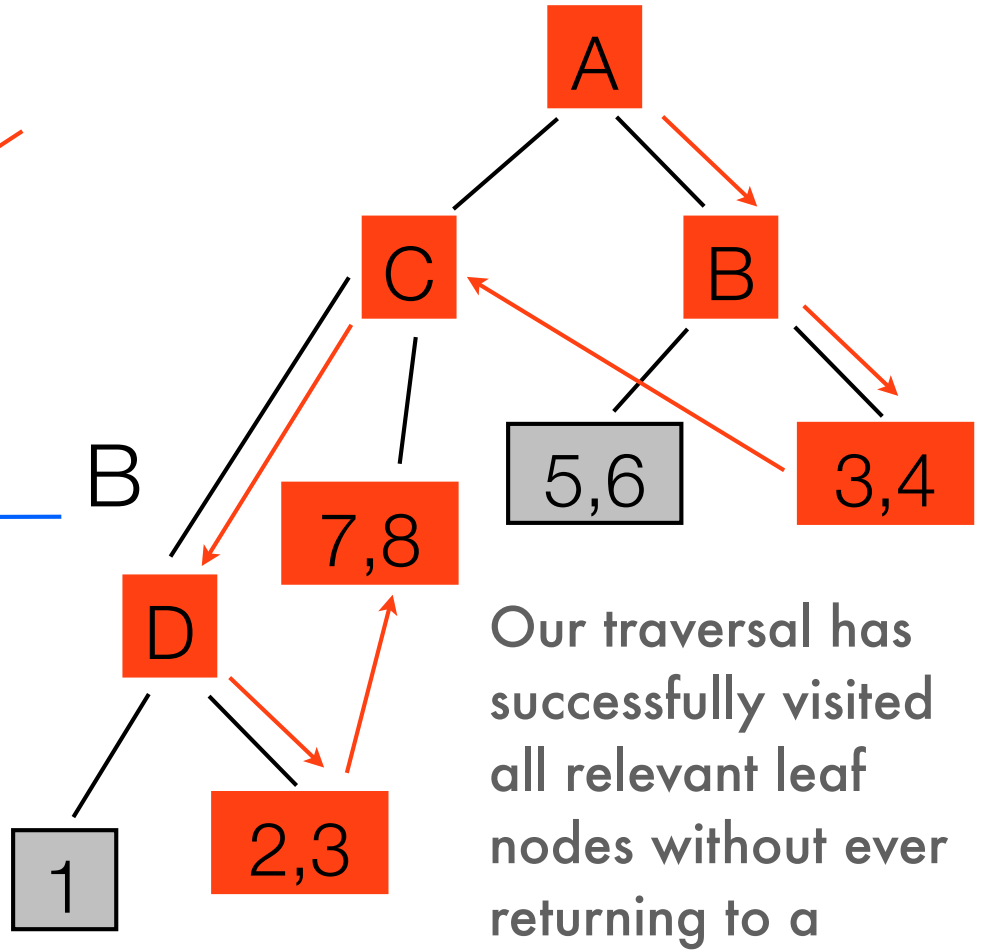


Stackless KD-Trees: Traversal



Stackless KD-Trees: Traversal





Our traversal has successfully visited all relevant leaf nodes without ever returning to a parent node that we already visited!