CSE332: Data Abstractions

Lecture 23: Wrapping Up

James Fogarty

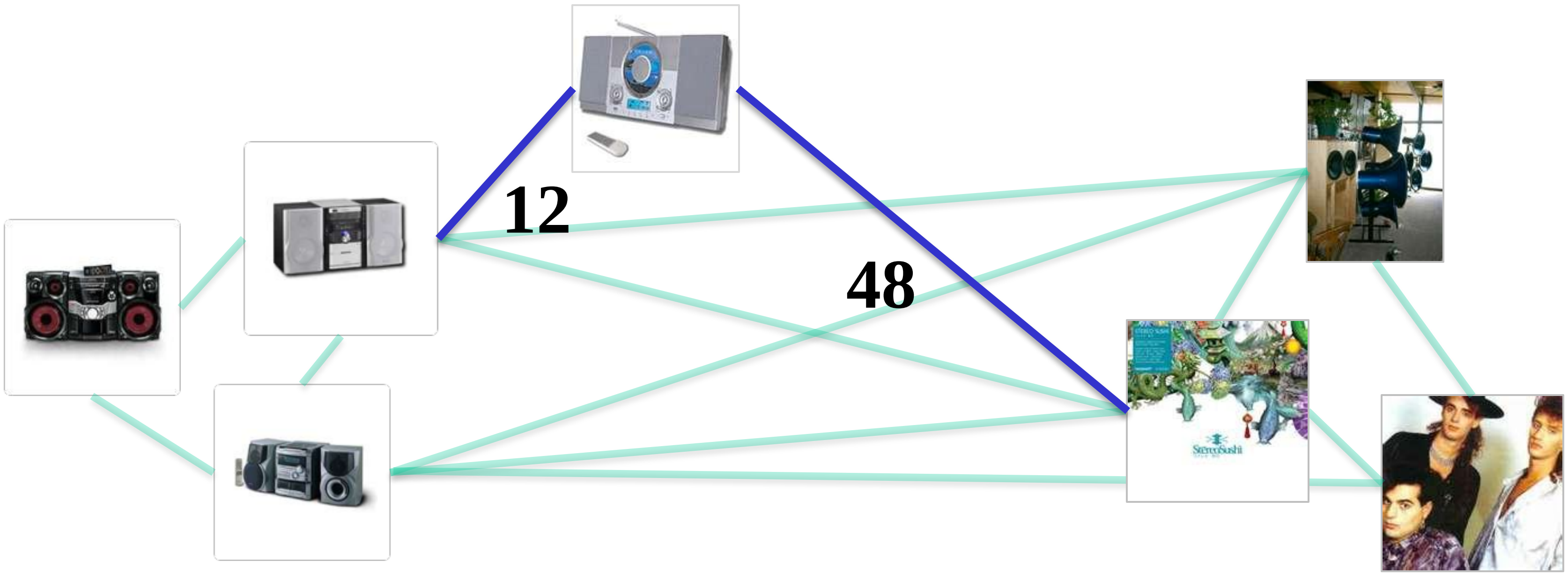
Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman, Richard Ladner, Steve Seitz

The Good News

- You have learned a set of tools that allow you to think about, and talk about, a wide variety of computing problems

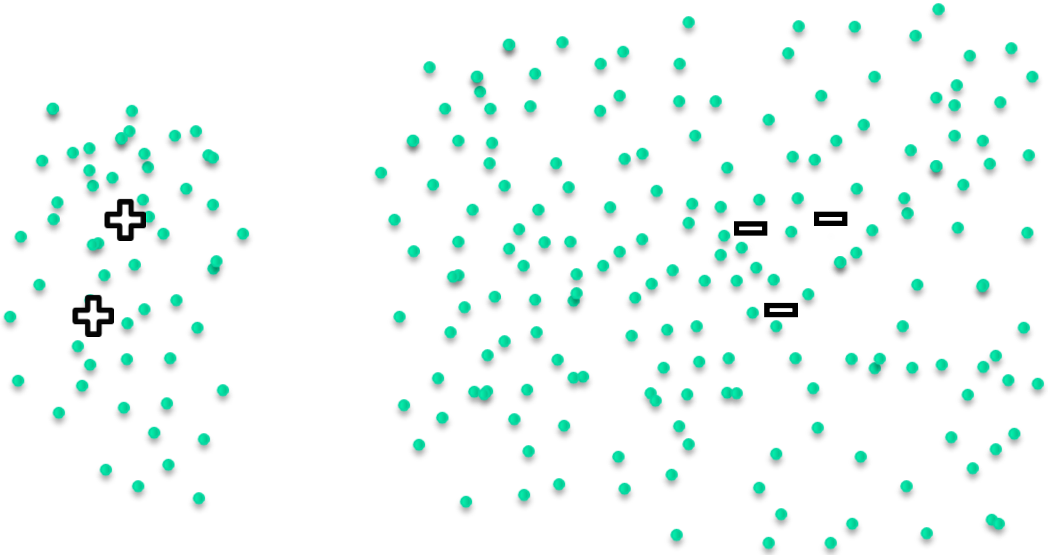
CueFlik: Learning Image Similarity



1 – (12 / (12 + 48)) = 75% likely a 'product' image

AAAI 2011, Amershi *et al.*
CHI 2010, Amershi *et al.*
UIST 2009, Amershi *et al.*
CHI 2008, Fogarty *et al.*

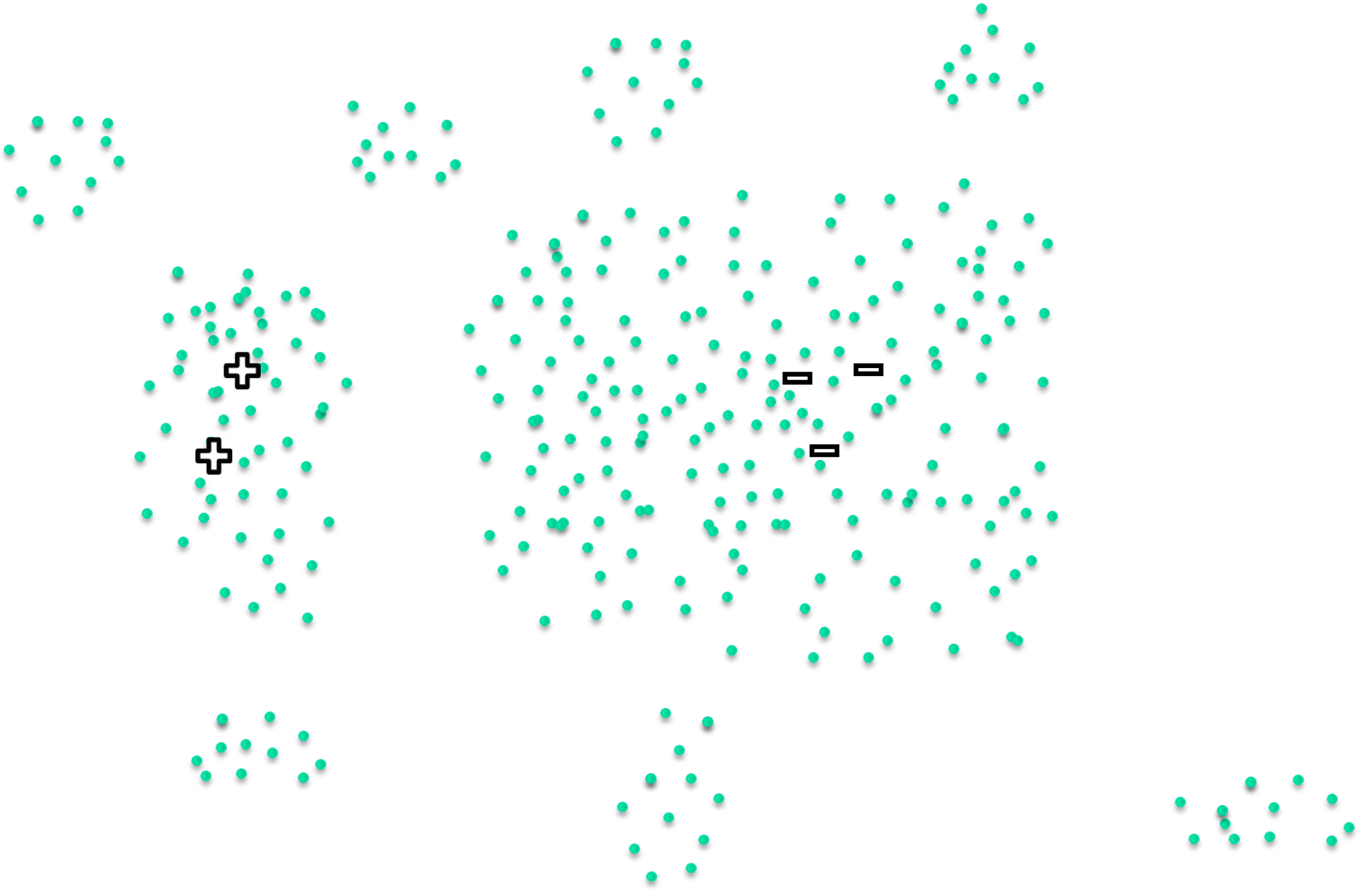
How Can We Decide Which are Positive?



KNN Graph

- Compute pairwise distance between every pair
(using domain knowledge to determine the distance metric)
- Preserve edges corresponding only to the k nearest neighbors of each vertex in the graph
- Run a search from your positive and negative examples, classify each based on whichever is closer
- KNN greatly reduces $|E|$, from $|V^2|$ to $k|V|$ (i.e., dense to sparse)
- The classification strategy is also semi-supervised, respecting the distribution of your data
    - Imagine two interlocking spirals

How About Now?



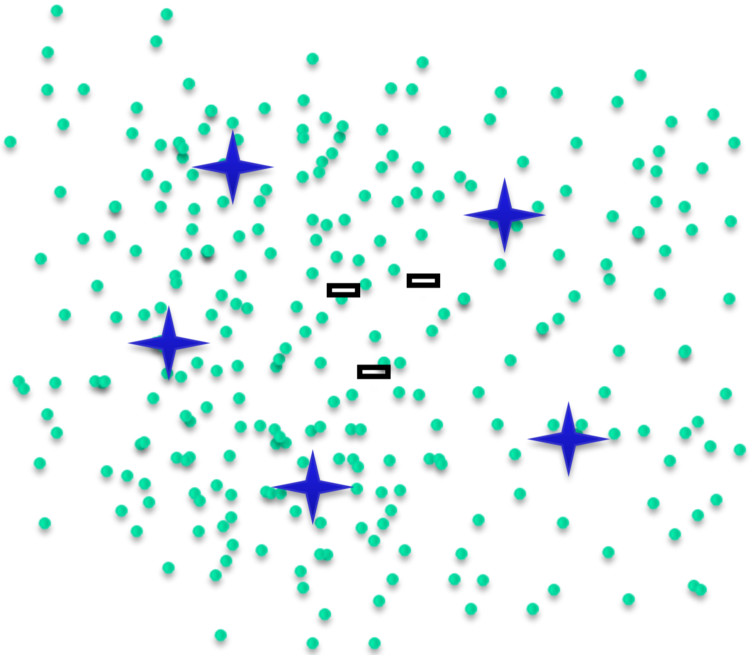
Disconnected Graph

- The KNN transformation does not necessarily preserve a path between your labels and all of your data

- You have to decide what this means and what to do about it

- What kinds of tools can you work with?

How Can We Choose a Representative Set?



The Bad News

- We do not know how to efficiently the items which are "the most representative subset of these elements"

- Our implementation is greedy
  - Choose single most representative item
  - Given that choice, choose another
  - Repeat until desired number of items

- It gets worse, there are many such problems
  - Learn about P and NP in CSE 312

Some Better News

- We have lots of cool algorithms, not just those you have seen

Amortized Algorithms

- Recall our stack implemented as an array
  - Doubles its size if it runs out of room
  - How can we claim **`push`** is $O(\mathbf{1})$ time if resizing is $O(\mathbf{n})$ time?
  - We *cannot*, but we *can* claim it's an $O(\mathbf{1})$ amortized operation

We will just do a simple example

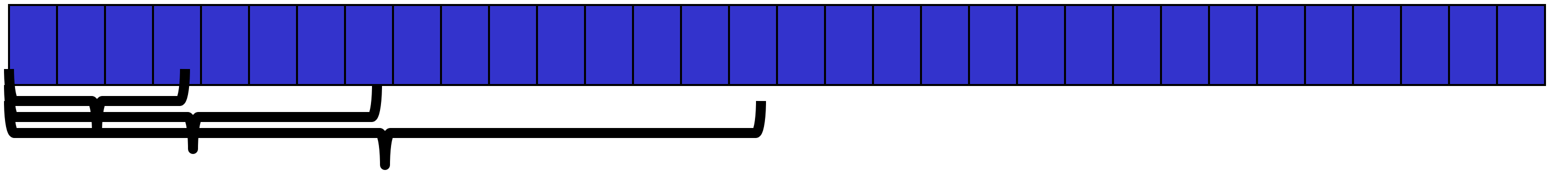
- There are entire families of data structures based on this
- The text has more complicated examples and proof techniques
- The *idea* of how amortized describes average cost is essential

Amortized Complexity

If a sequence of **`M`** operations takes $O($**`M f(n)`**$)$ time,
we say the amortized runtime is $O($**`f(n)`**$)$

- The worst case time per operation can be larger than **`f(n)`**
  - For example, maybe **`f(n)=1`**, but the worst-case is **`n`**
- But the worst-case for *any* sequence of **`M`** operations is $O($**`M f(n)`**$)$
- Amortized guarantee ensures the average time per operation for any sequence is $O($**`f(n)`**$)$
  - This is a stronger guarantee than "average case" $O($**`f(n)`**$)$

Amount of Copying in a "Doubling" Stack



After **`M`** operations, we have done **`< 2M`** total element copies

Let **`n`** be the size of the array after **`M`** operations

- Then we've done a total of:

  `n/2 + n/4 + n/8 + … INITIAL_SIZE < n`

  element copies
- We must have done at least enough **`push`** operations to cause resizing up to size **`n`**:

  $\mathbf{M} \geq \mathbf{n/2}$
- So

  $\mathbf{2M} \geq \mathbf{n} >$ *number of element copies*

Other Approaches to Growing / Shrinking

- If array grows by a constant amount (say 100), operations are not amortized $O(\mathbf{1})$
  - After 1000 operations, you may have done 900 + 800 + 700 + … + 300 + 200 + 100 copies (i.e., N^2)

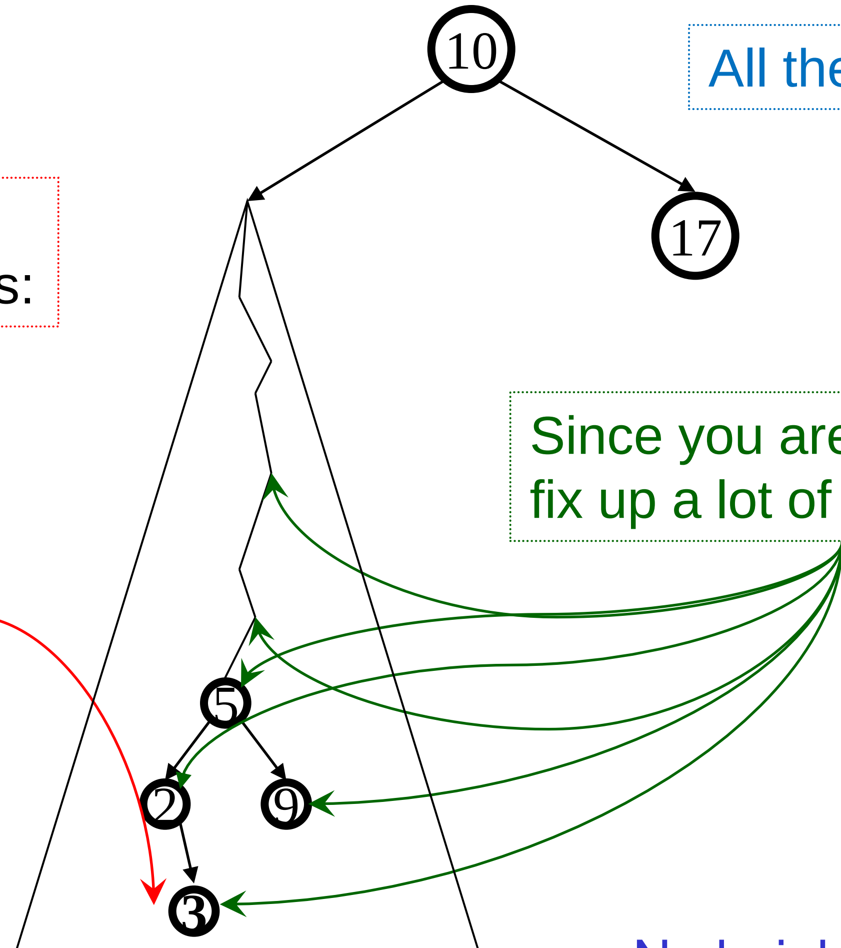
- If array shrinks when 1/2 empty, operations are not amortized $O(\mathbf{1})$
  - **`pop`** and shrink, **`push`** and grow, **`pop`** and shrink, …

- If array shrinks when 3/4 empty, it is amortized $O(\mathbf{1})$
  - Proof is more complicated, but basic idea remains: by the time o fan expensive operation, many cheap operations

Splay Tree Basic Idea

All the way to the root!

If you are forced to make a deep access:



Since you are down there, fix up a lot of deep nodes!

No height "bookkeeping"

Amortized O(log n) operations

Usefulness of Amortized Algorithms

- Proofs are complicated, with “potential functions” to describe how cheap operations “pay for” later expensive operations
  - But this has nothing to do with complexity of the code
  - Often simple, with better constant factors

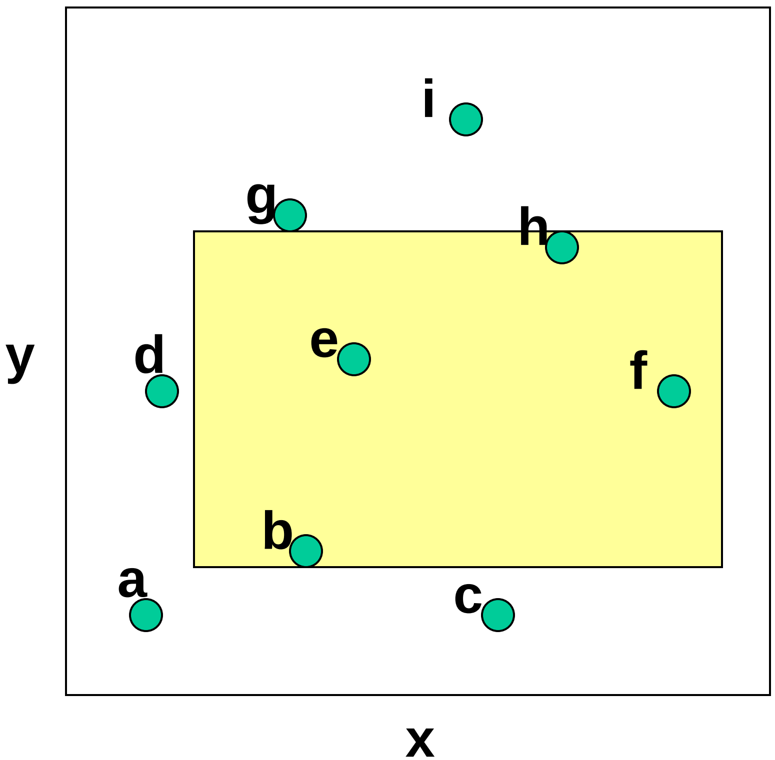
- When the average cost per operation is all we care about (i.e., sum over all operations), amortized is perfectly fine
  - This is a very common situation

- If every operation must finish quickly, amortized bounds are weak
  - Real-time software
  - Concurrency setting, where you are holding the lock

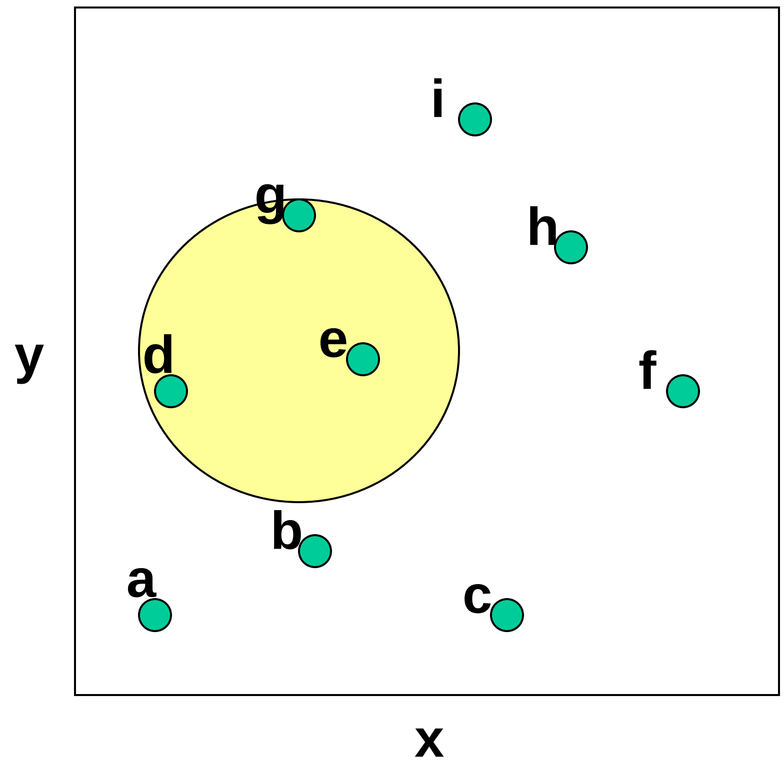
Range Queries

- Project 3 Grouped Census Data into Blocks
- What if you wanted to keep the original data
  and efficiently answer queries at arbitrary precision
- Balanced trees can allow accessing on one dimension
  - “Give me all blocks between longitudes x and y”
  - “Give me all blocks between latitudes x and y”
- But what about access on both dimensions?
  - “Give me all blocks in a rectangle”

Range Queries

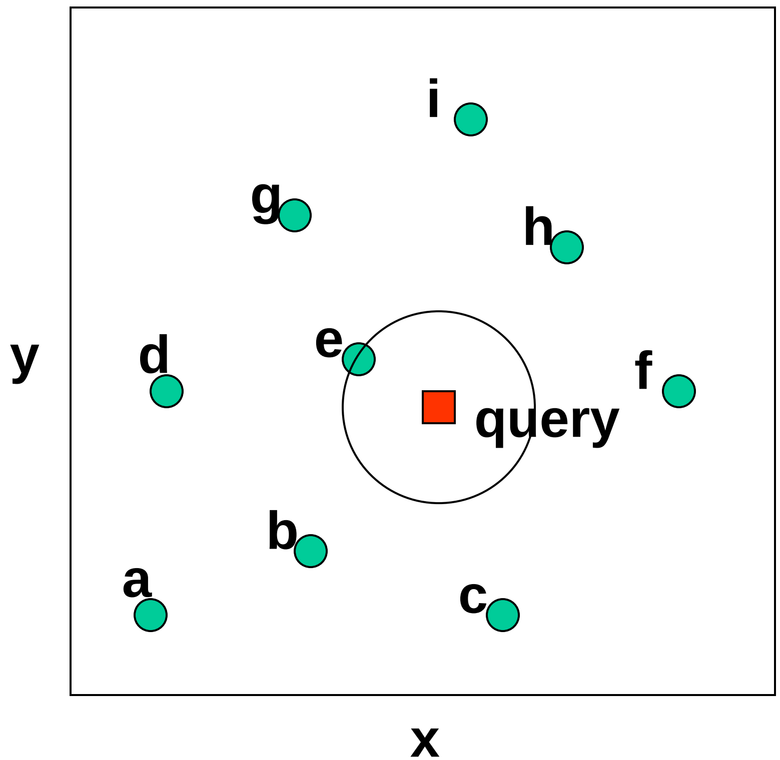


Rectangular query

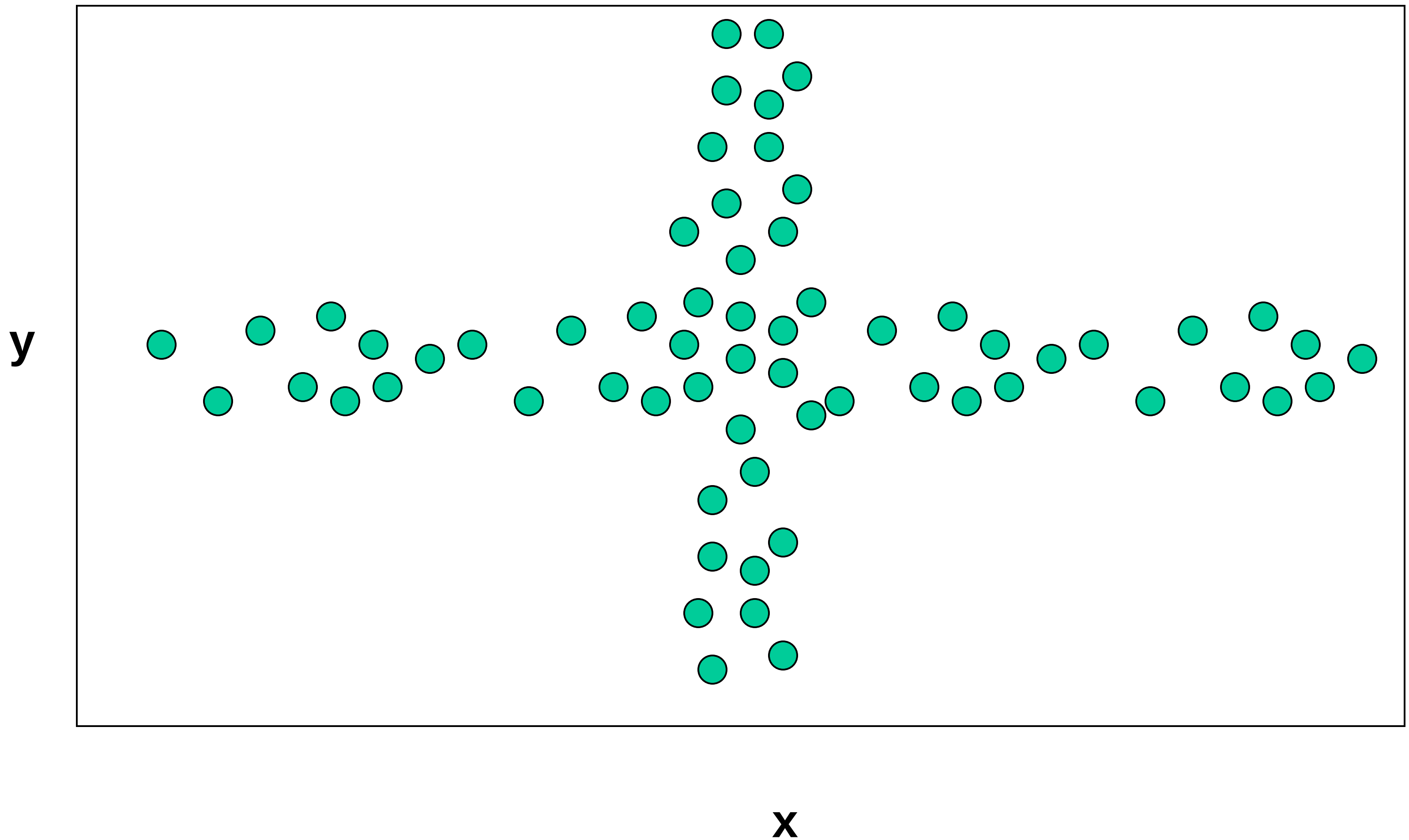


Circular query

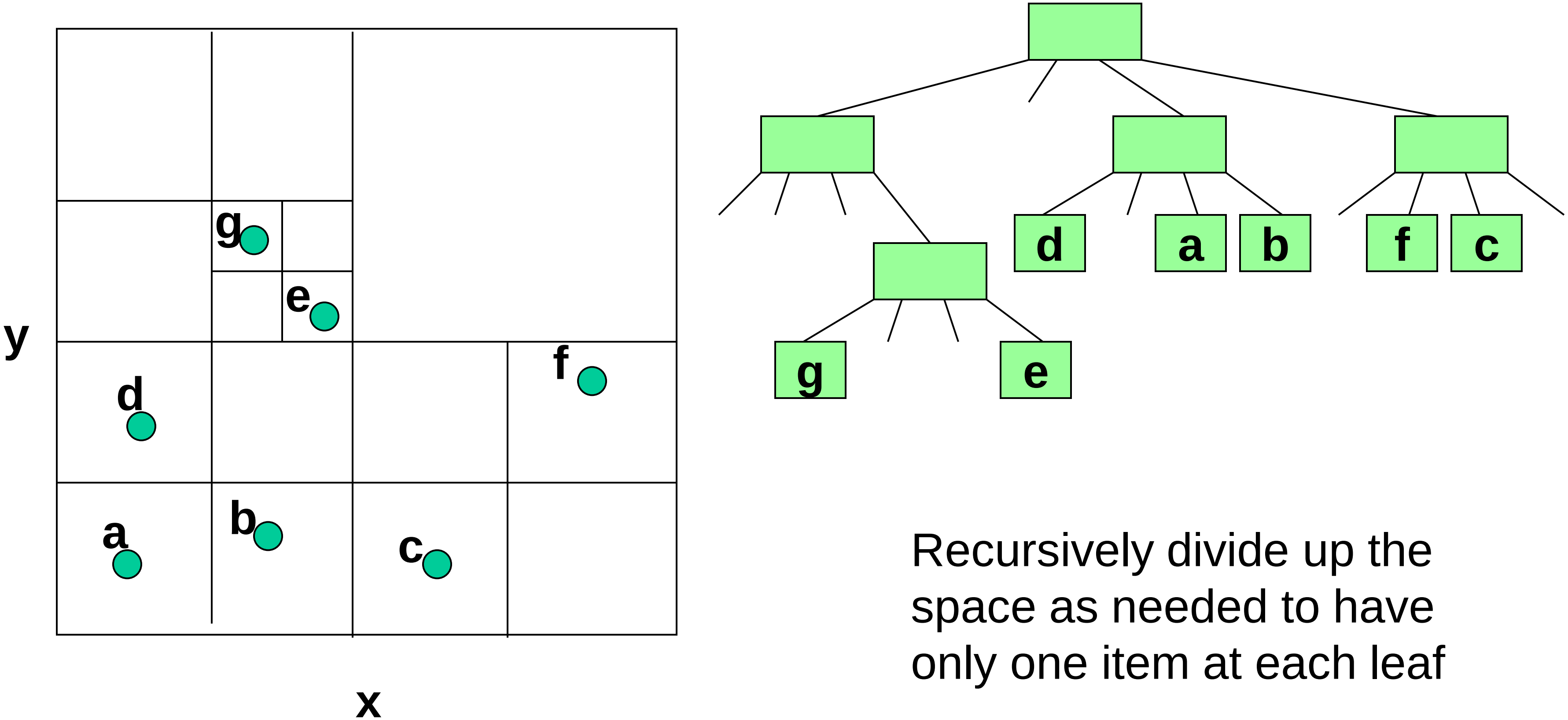
Nearest Neighbor Search



A Challenging Case for 1D Structure

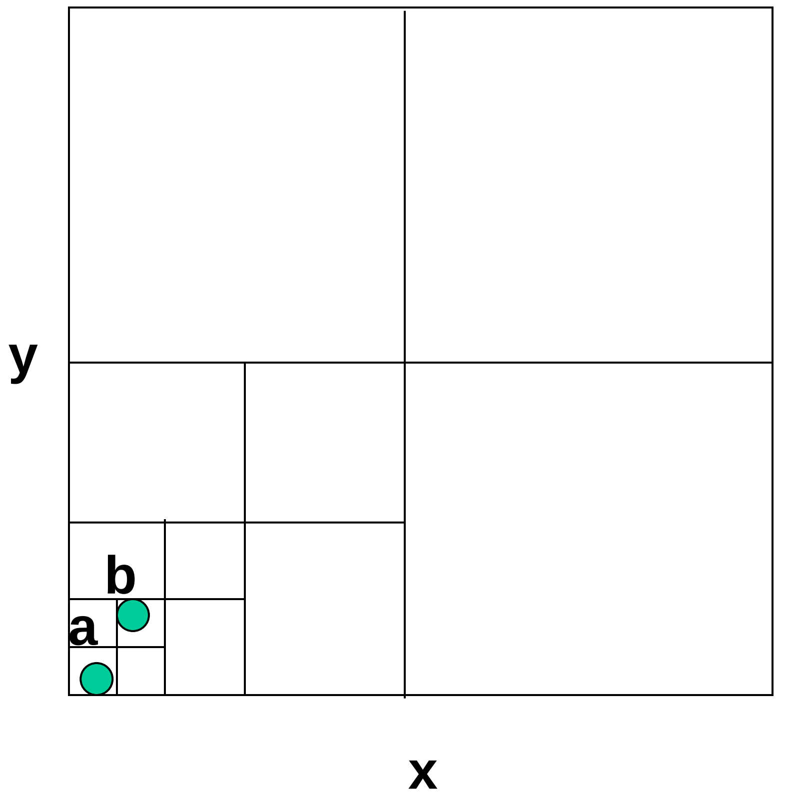


Quad Trees



Recursively divide up the space as needed to have only one item at each leaf

A Really Bad Case



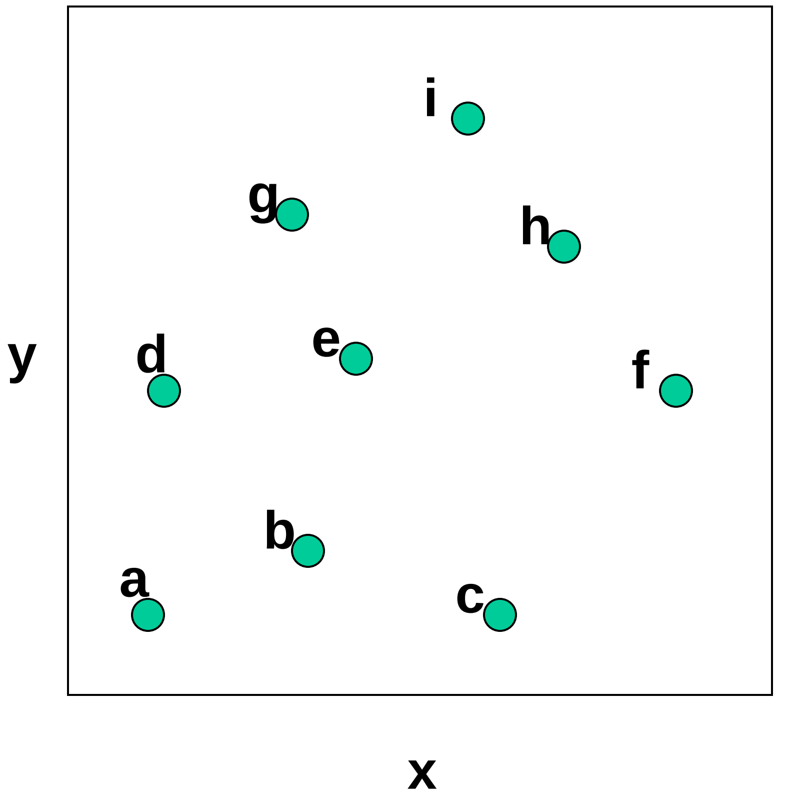
k-d Trees

- Jon Bentley, 1975, while an undergraduate
- Tree used to store spatial data.
  - Nearest neighbor search.
  - Range queries.
  - Fast look-up
- k-d tree are guaranteed $\log_2 n$ depth where n is the number of points in the set.
  - Traditionally, k-d trees store points in d-dimensional space which are equivalent to vectors in d-dimensional space.

k-d Tree Construction

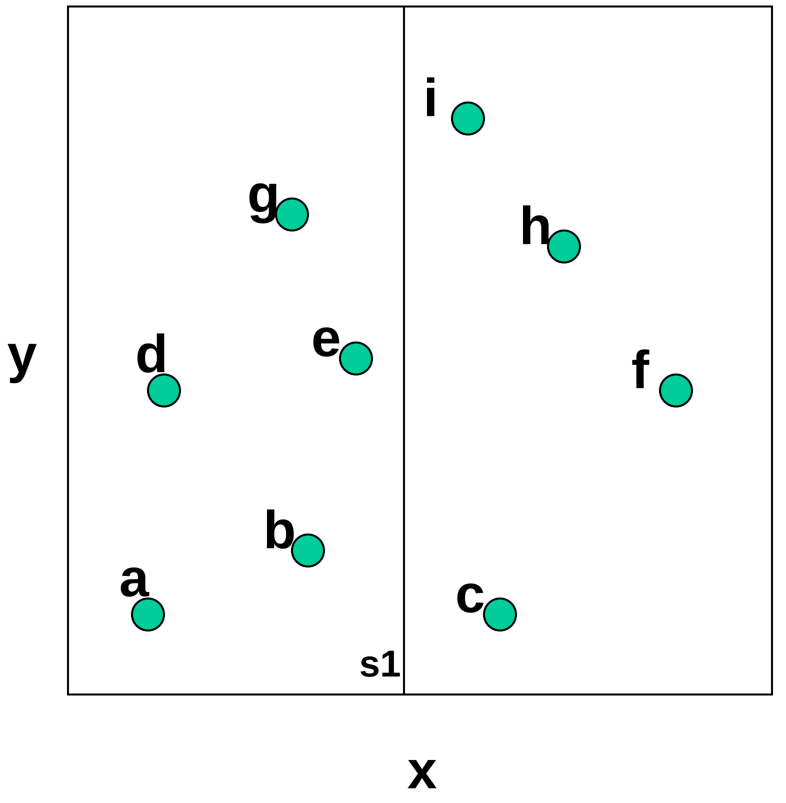
- If there is just one point, form a leaf with that point
- Otherwise, divide the points in half on one dimension
  - Book uses round-robin division
  - Could also divide on dimension with greatest spread
- Recursively construct k-d trees for the two sets of points

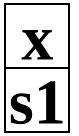
k-d Tree Construction



At each step divide perpendicular to the widest spread.

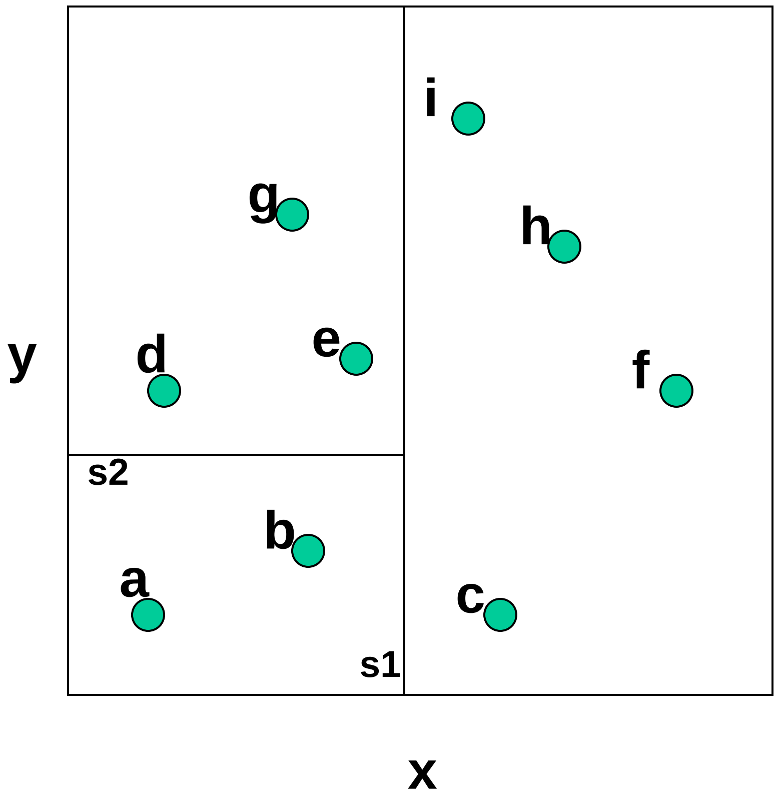
k-d Tree Construction

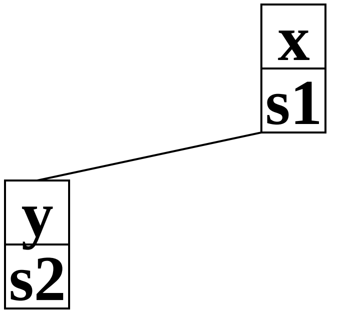




At each step divide perpendicular to the widest spread.

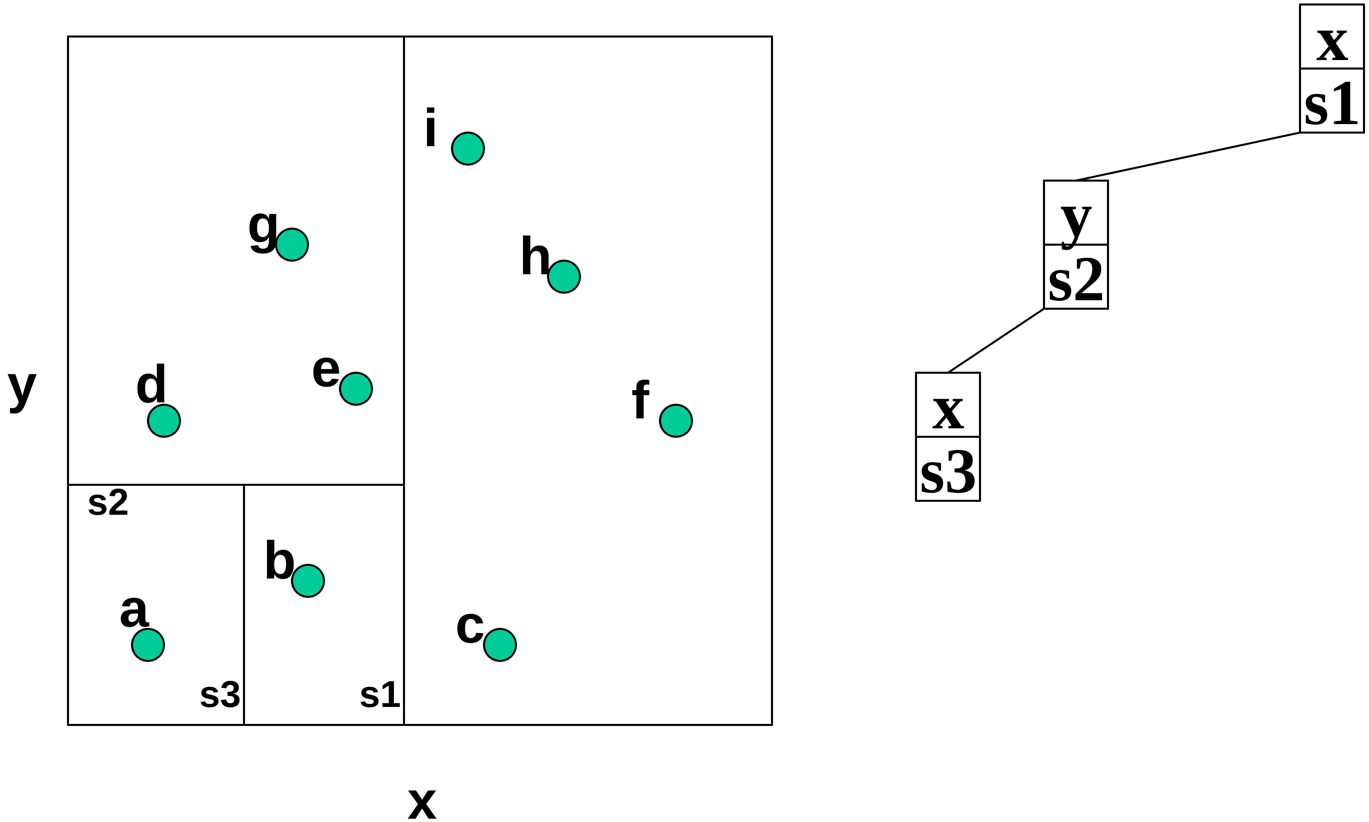
k-d Tree Construction





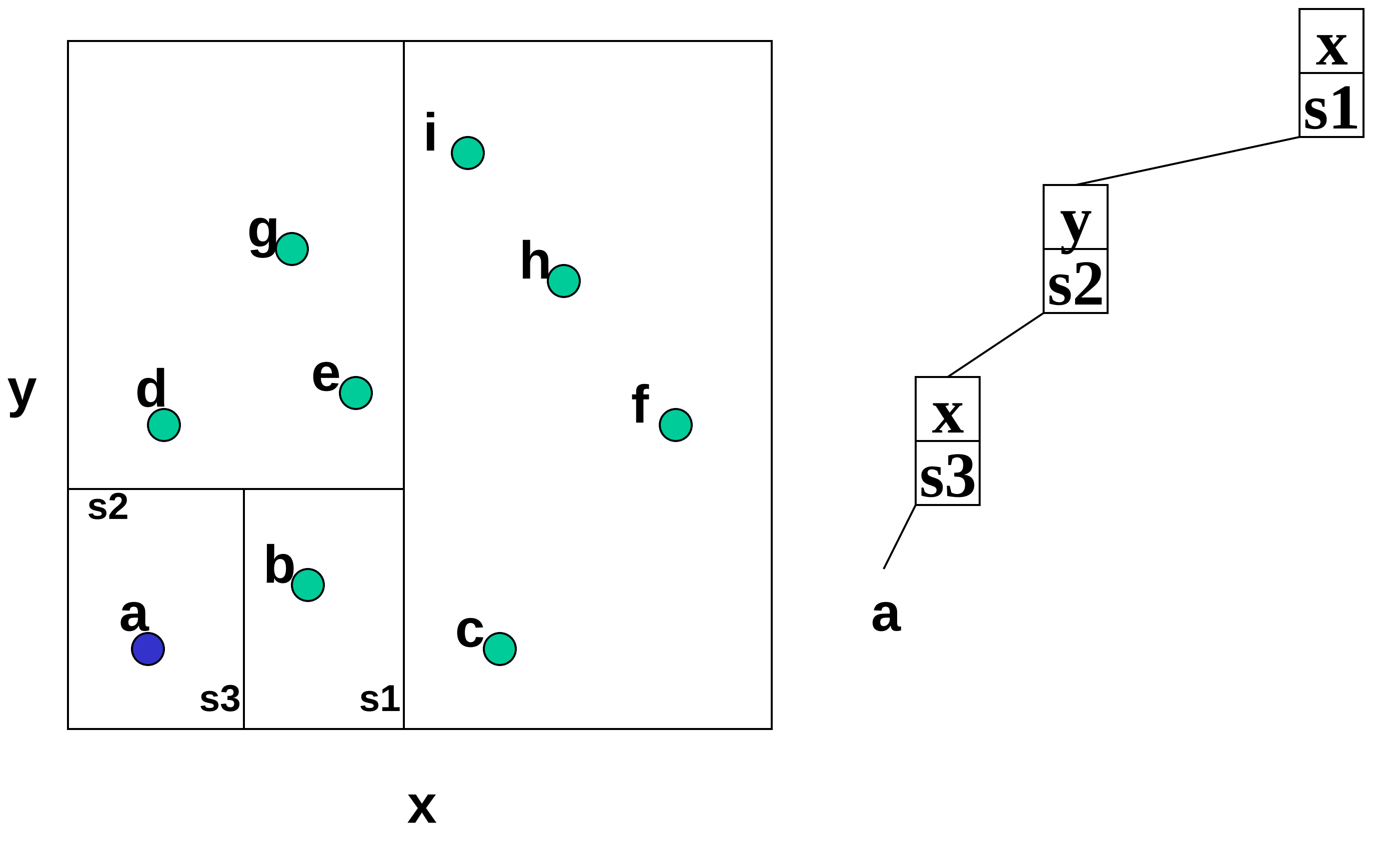
At each step divide perpendicular to the widest spread.

k-d Tree Construction



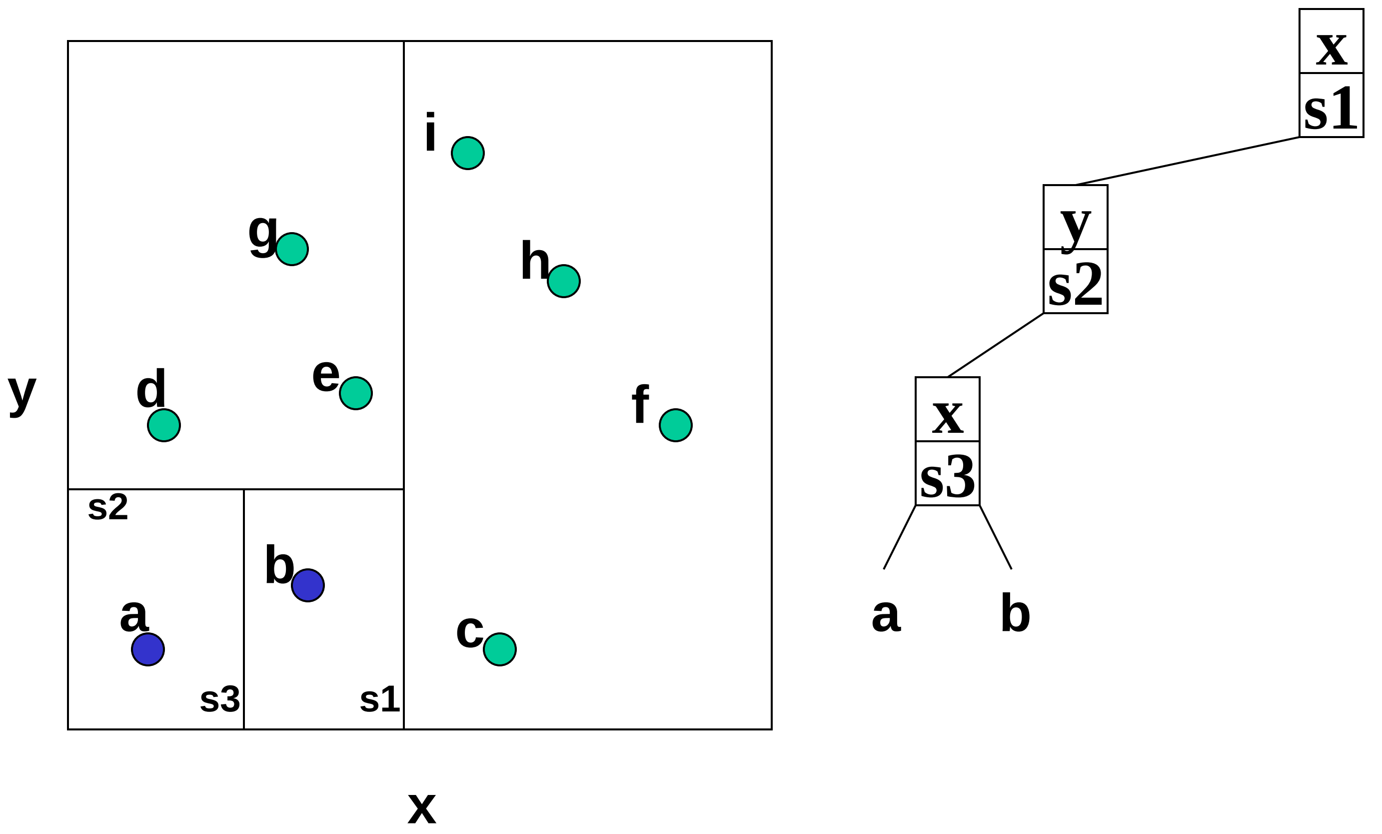
At each step divide perpendicular to the widest spread.

k-d Tree Construction



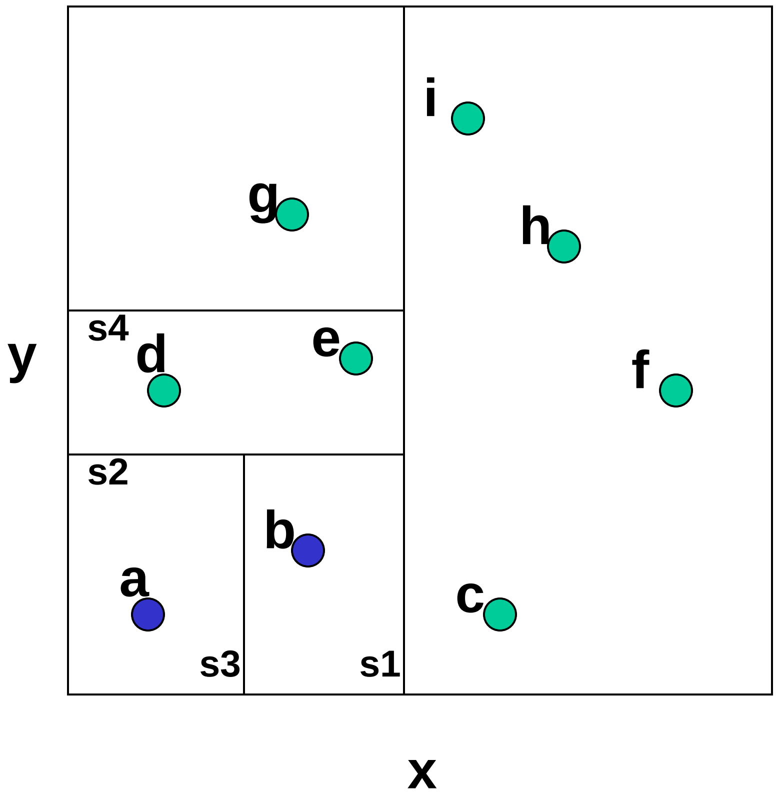
At each step divide perpendicular to the widest spread.

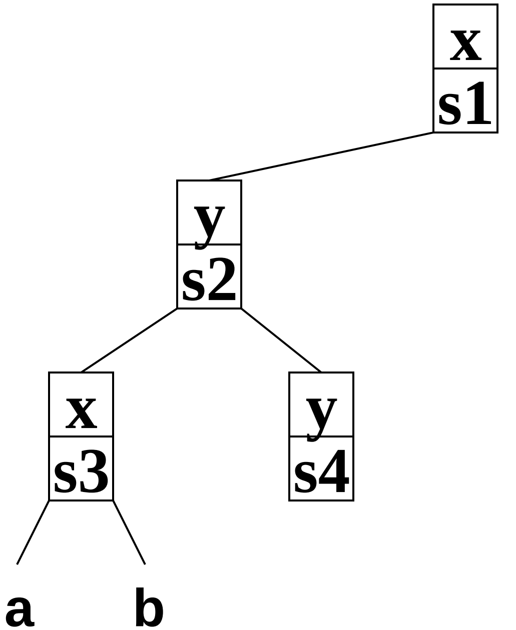
k-d Tree Construction



At each step divide perpendicular to the widest spread.

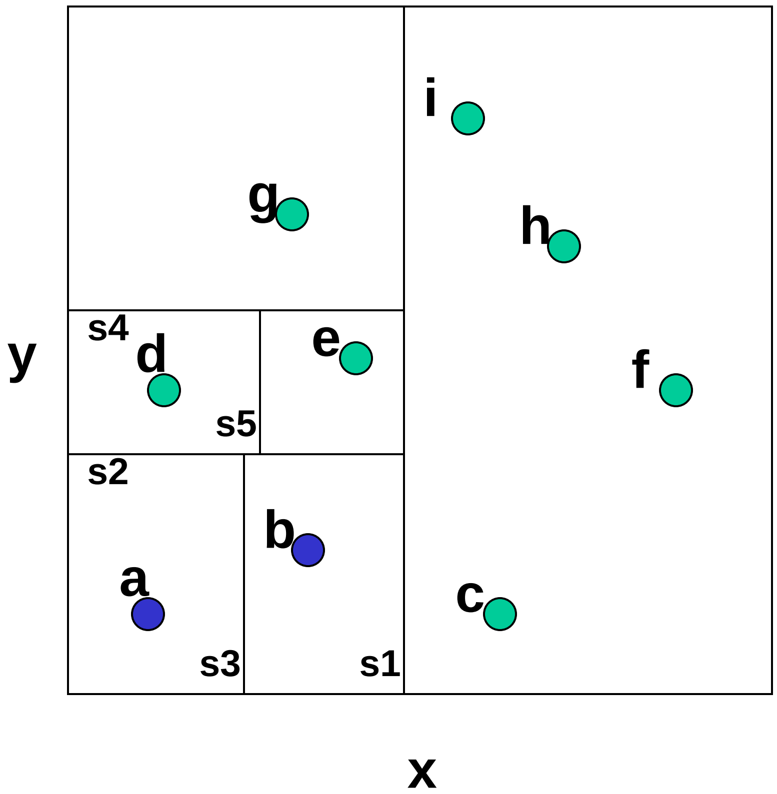
k-d Tree Construction

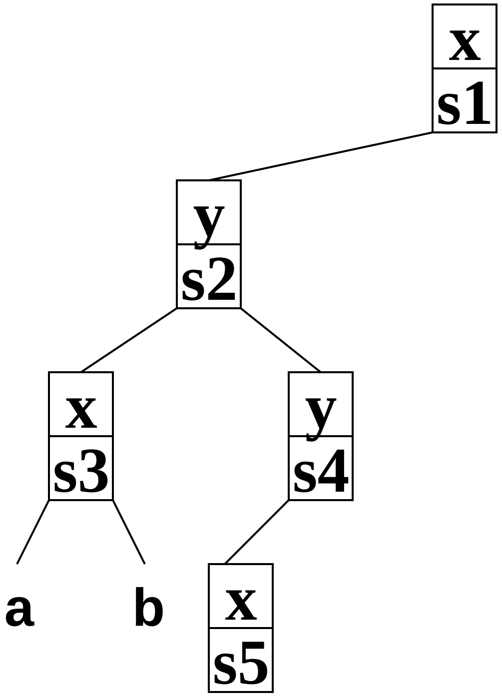




At each step divide perpendicular to the widest spread.

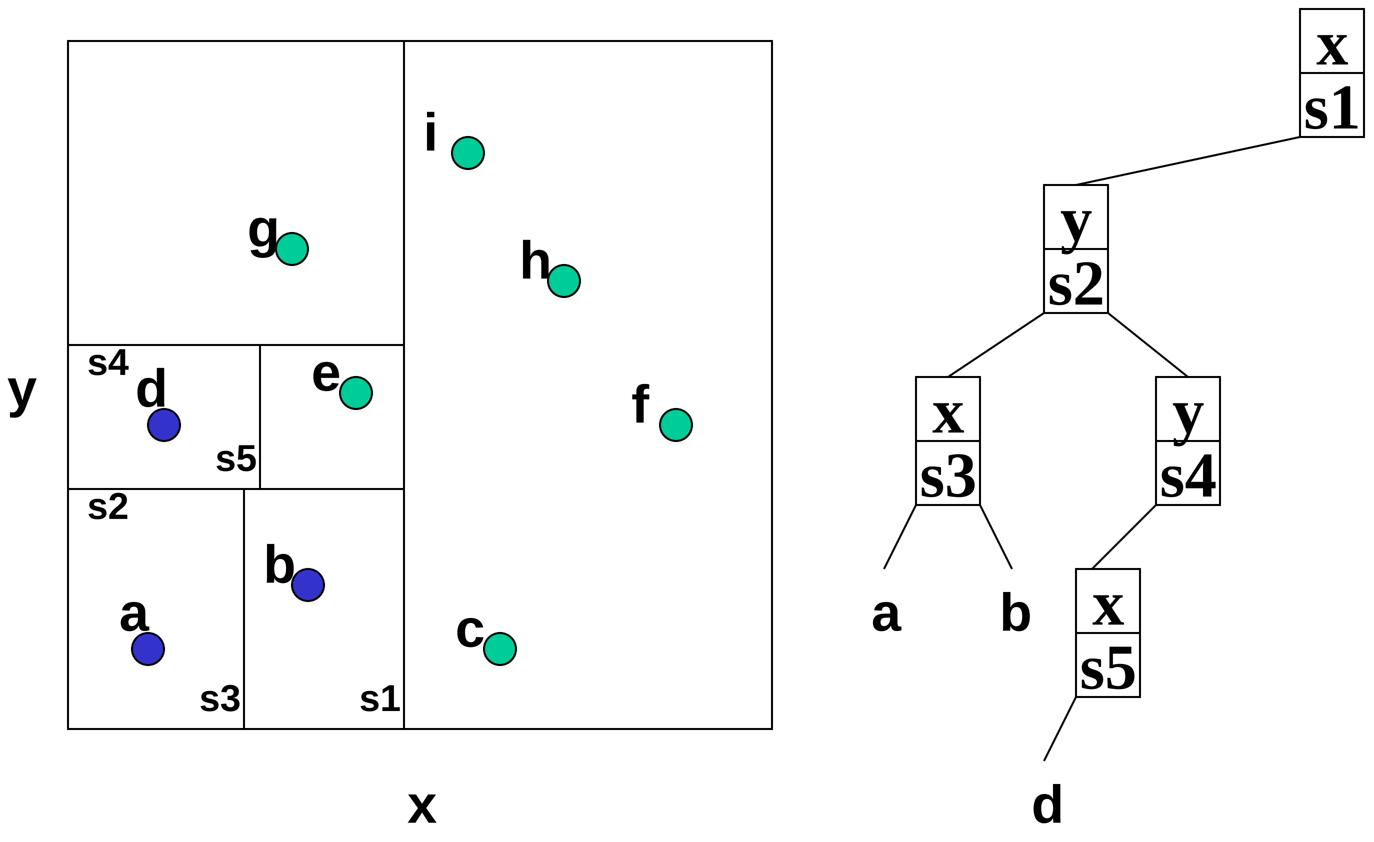
k-d Tree Construction





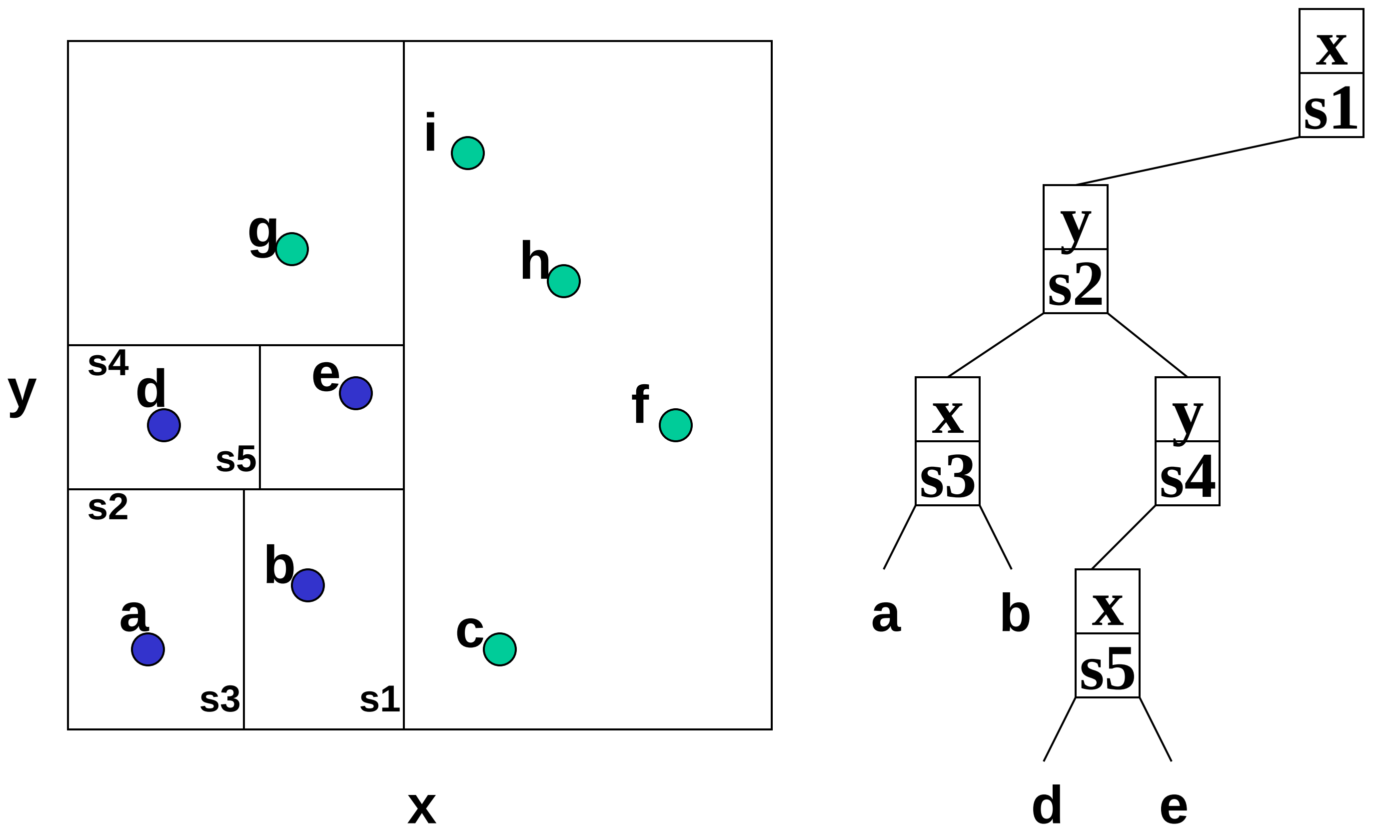
At each step divide perpendicular to the widest spread.

k-d Tree Construction



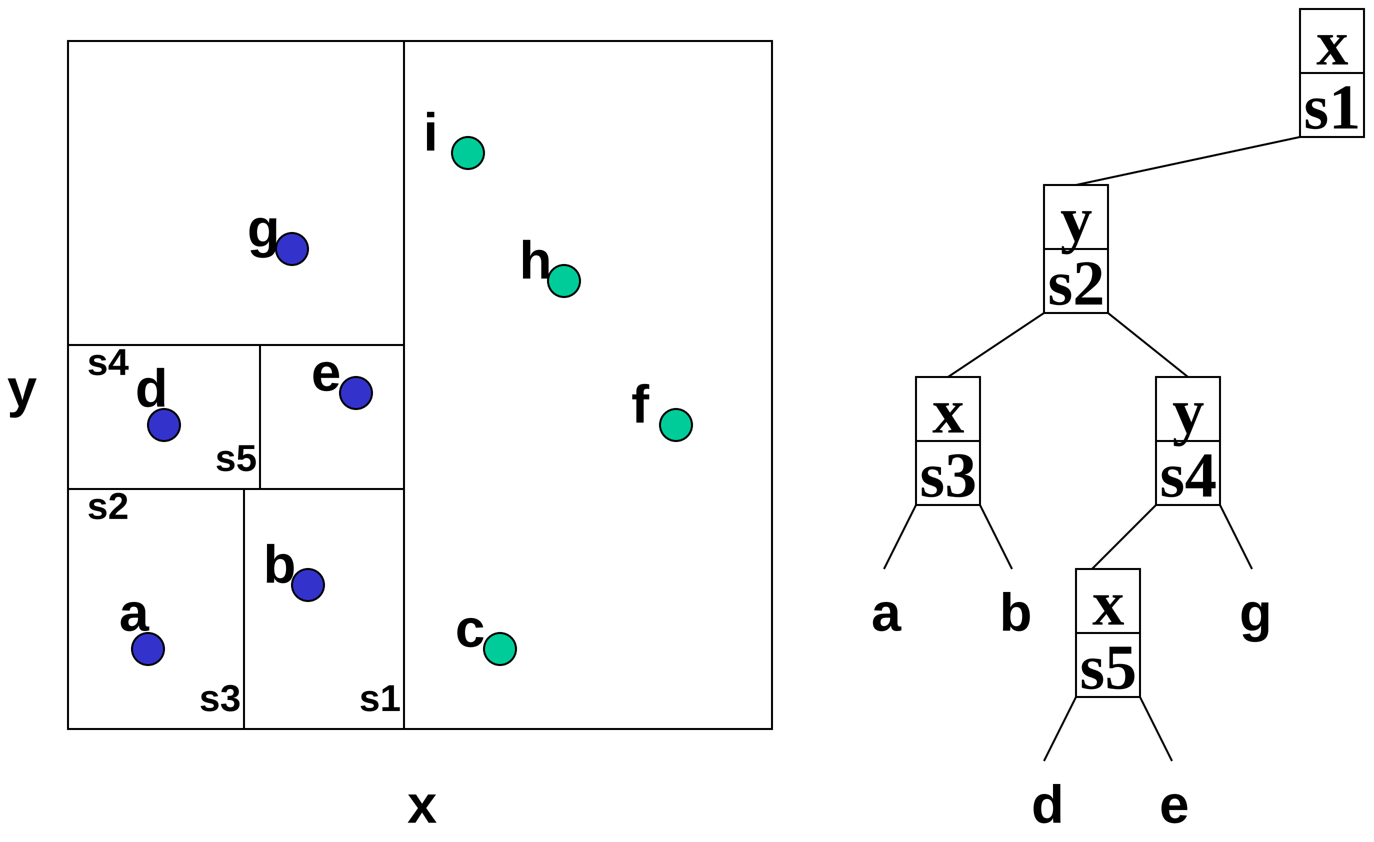
At each step divide perpendicular to the widest spread.

k-d Tree Construction



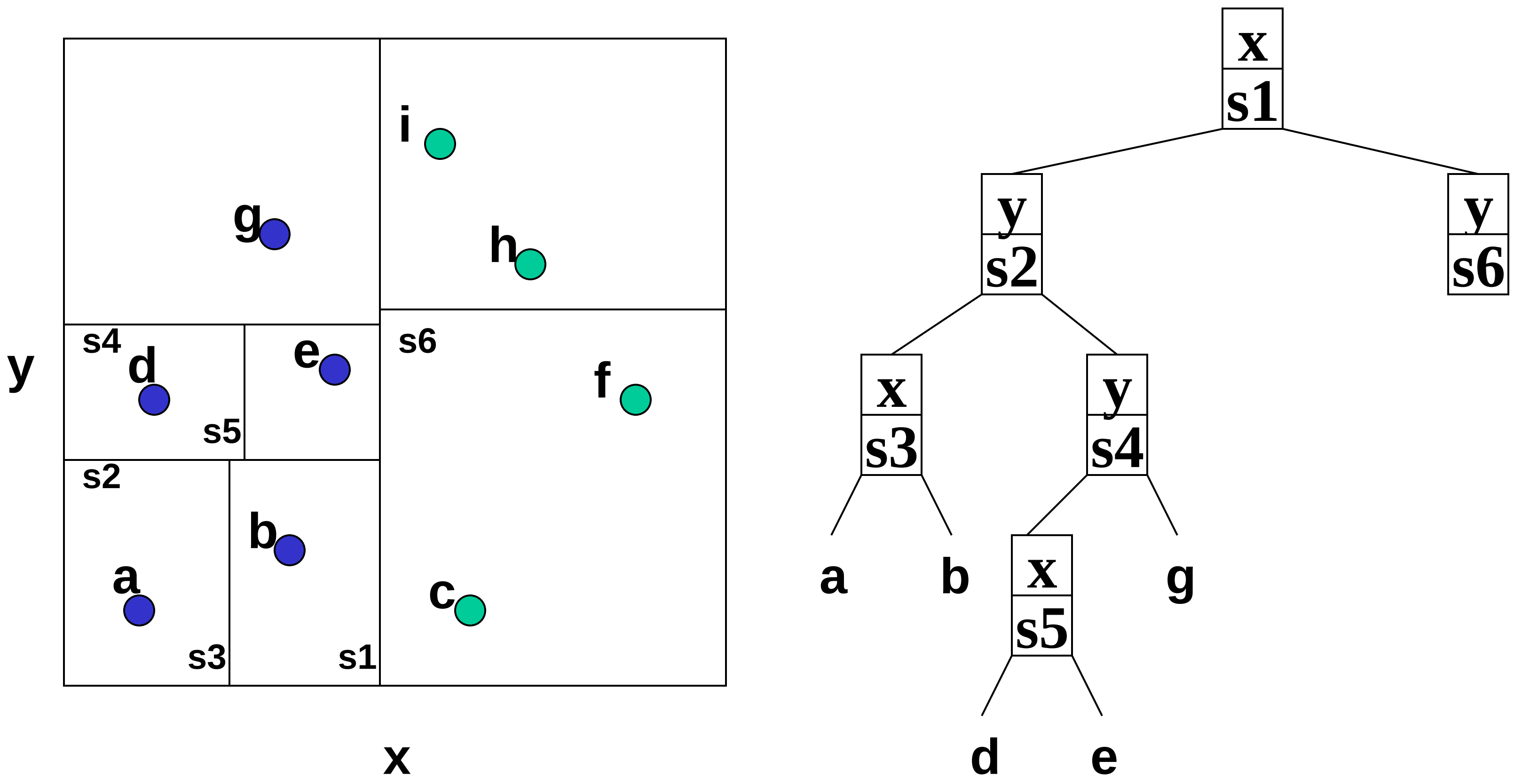
At each step divide perpendicular to the widest spread.

k-d Tree Construction



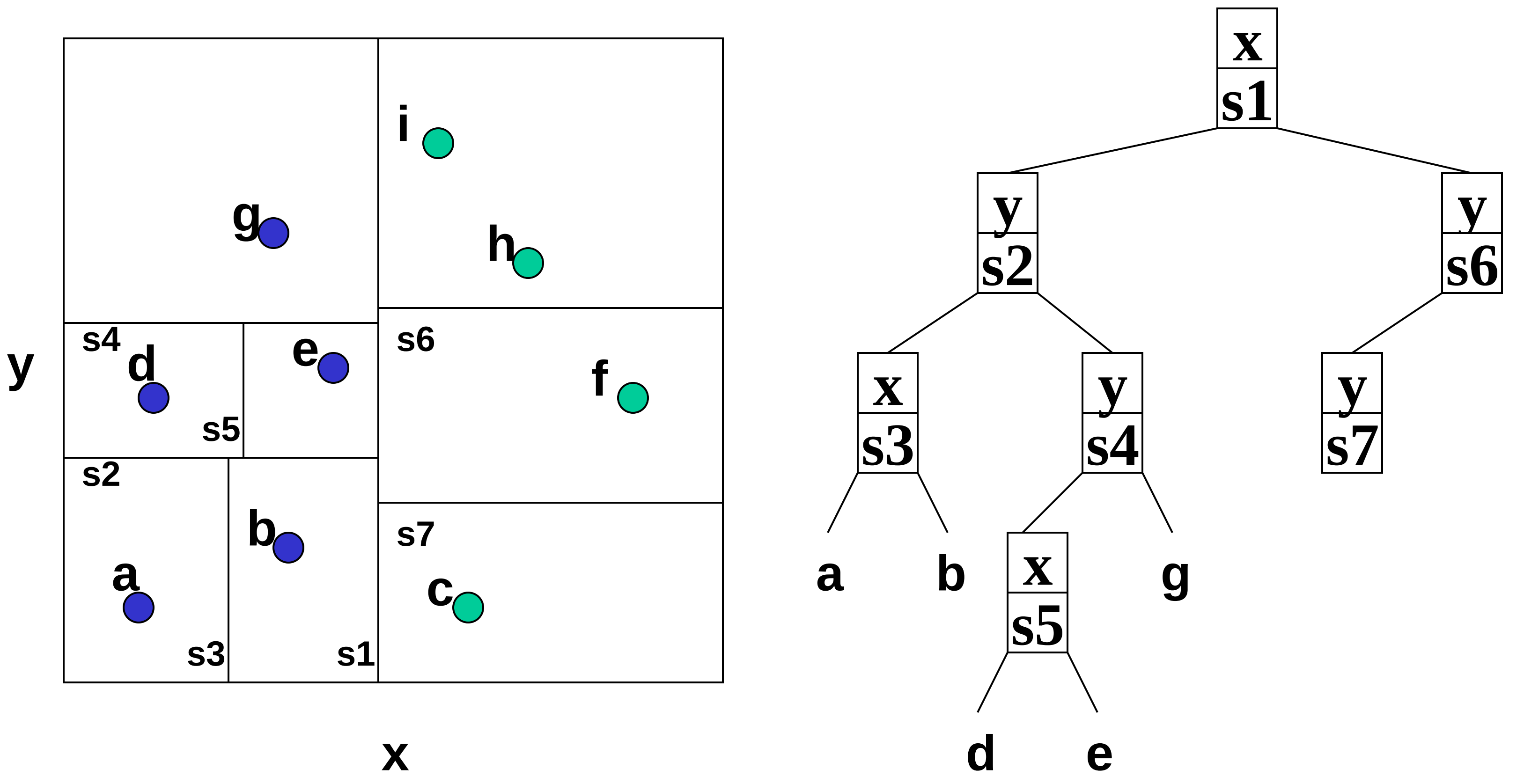
At each step divide perpendicular to the widest spread.

k-d Tree Construction



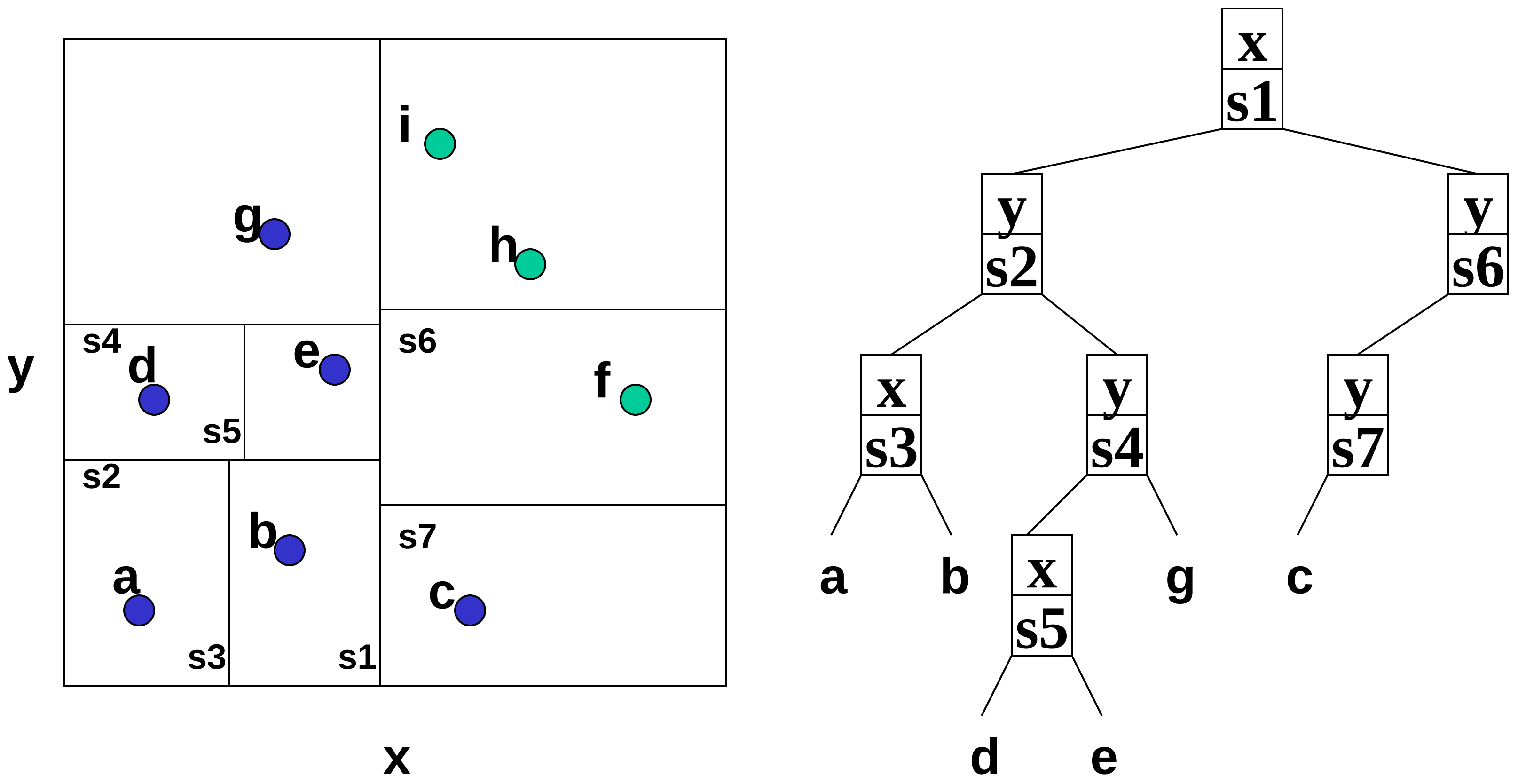
At each step divide perpendicular to the widest spread.

k-d Tree Construction



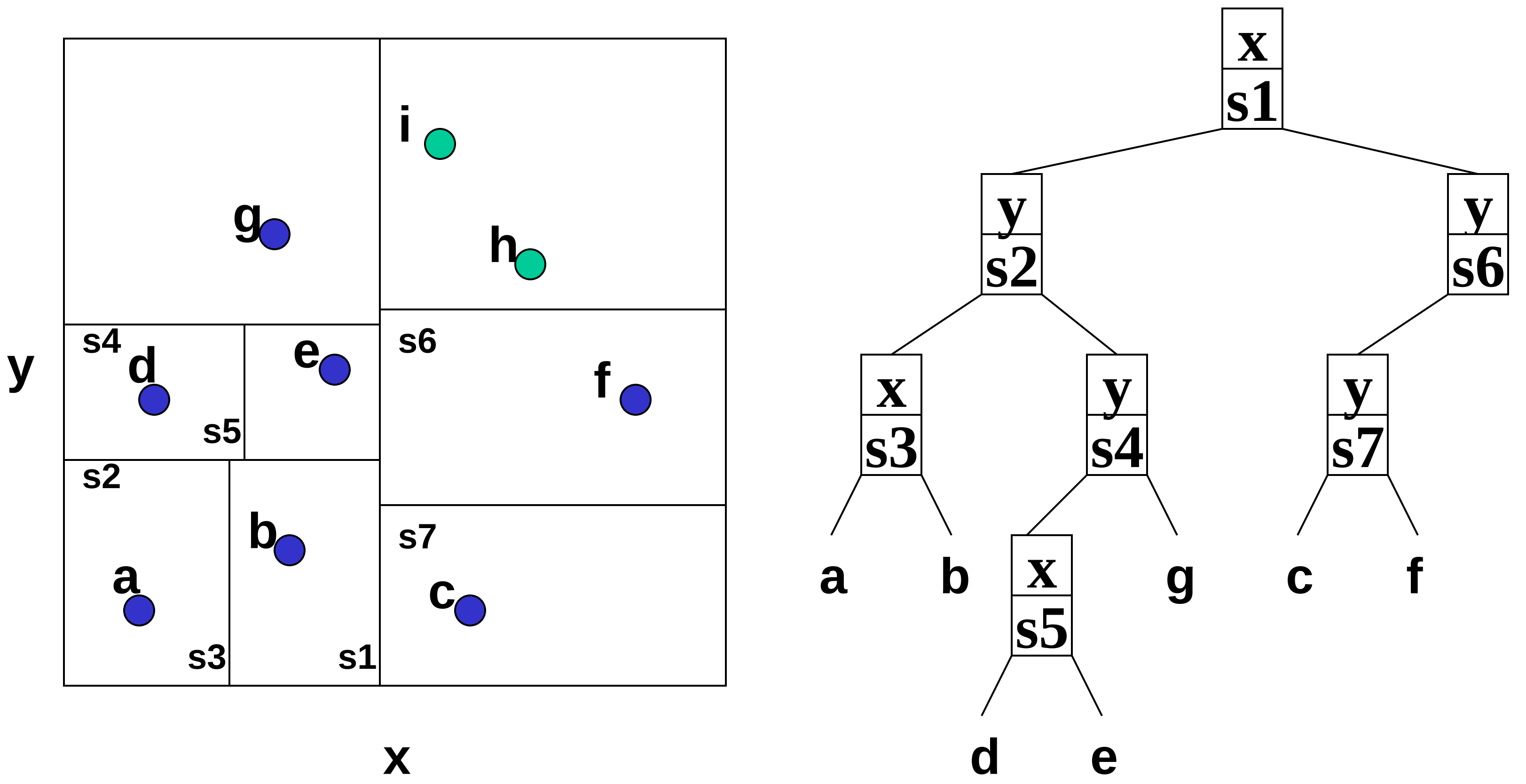
At each step divide perpendicular to the widest spread.

k-d Tree Construction



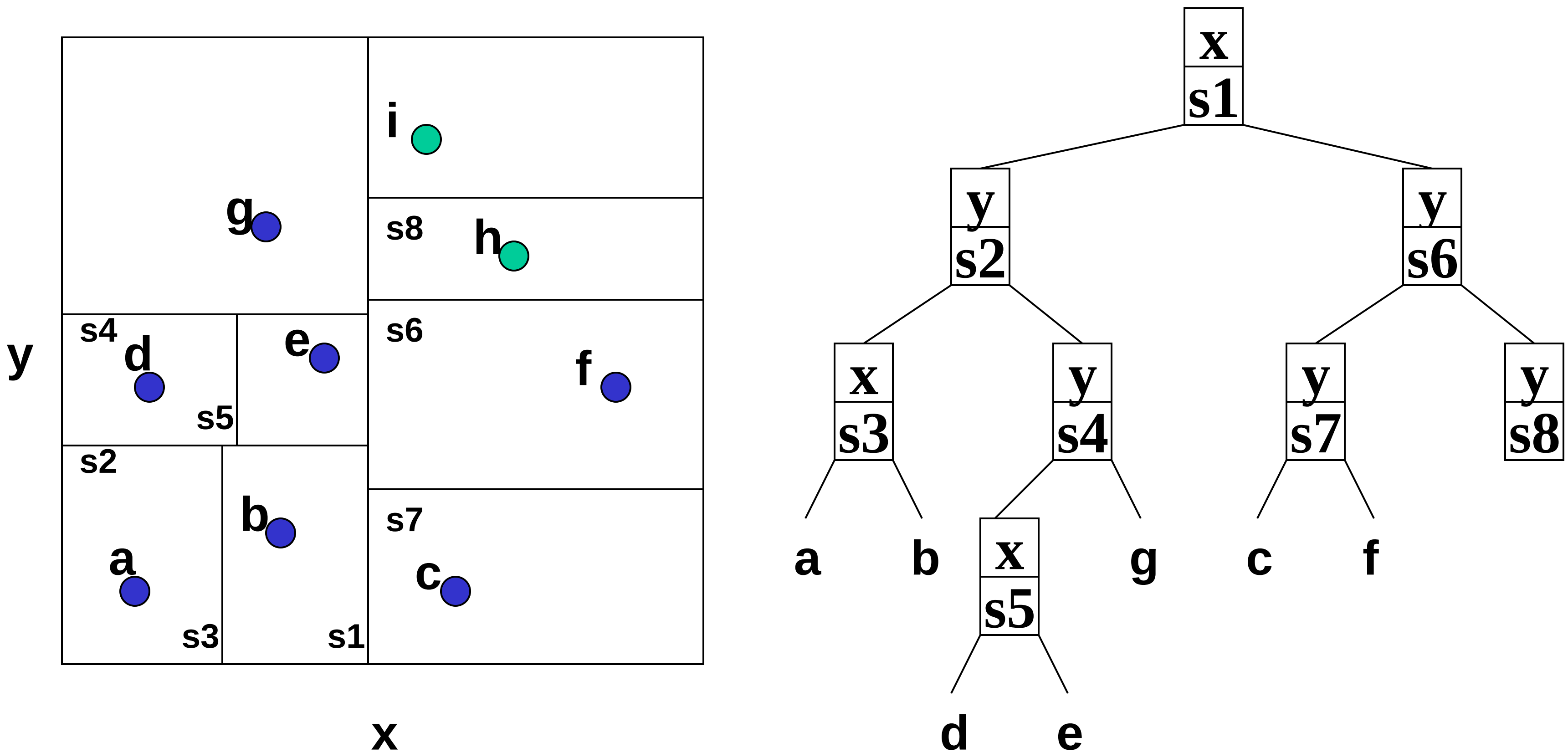
At each step divide perpendicular to the widest spread.

k-d Tree Construction



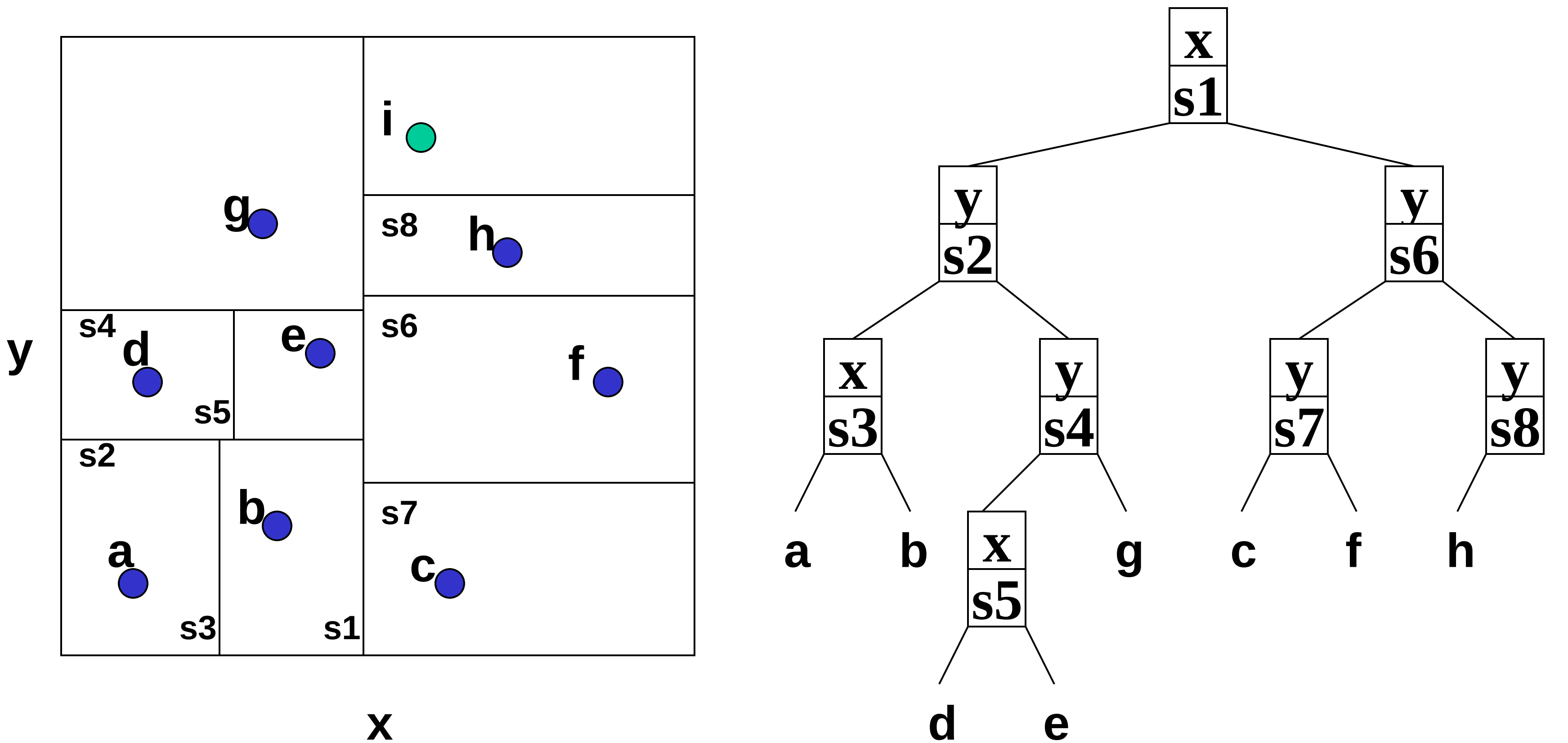
At each step divide perpendicular to the widest spread.

k-d Tree Construction



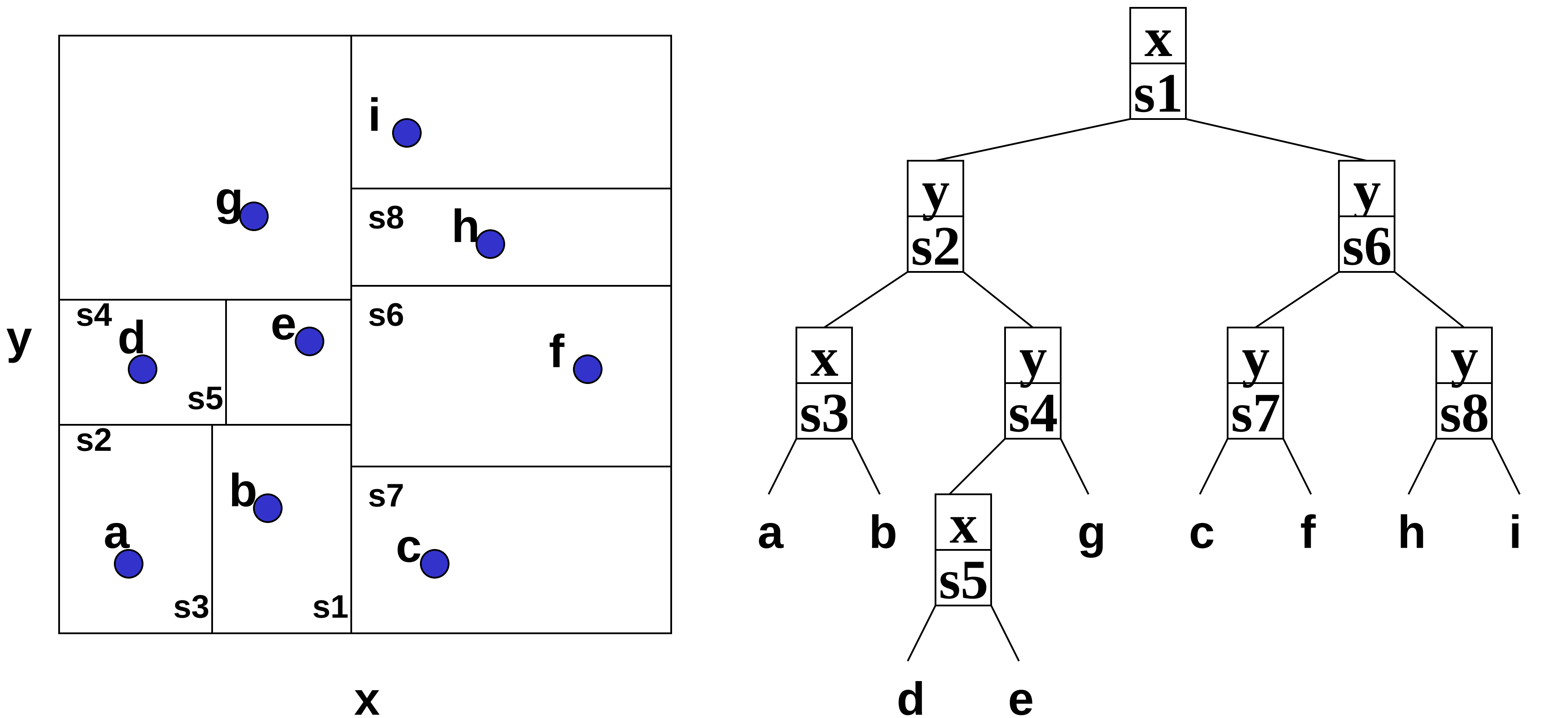
At each step divide perpendicular to the widest spread.

k-d Tree Construction



At each step divide perpendicular to the widest spread.

k-d Tree Construction

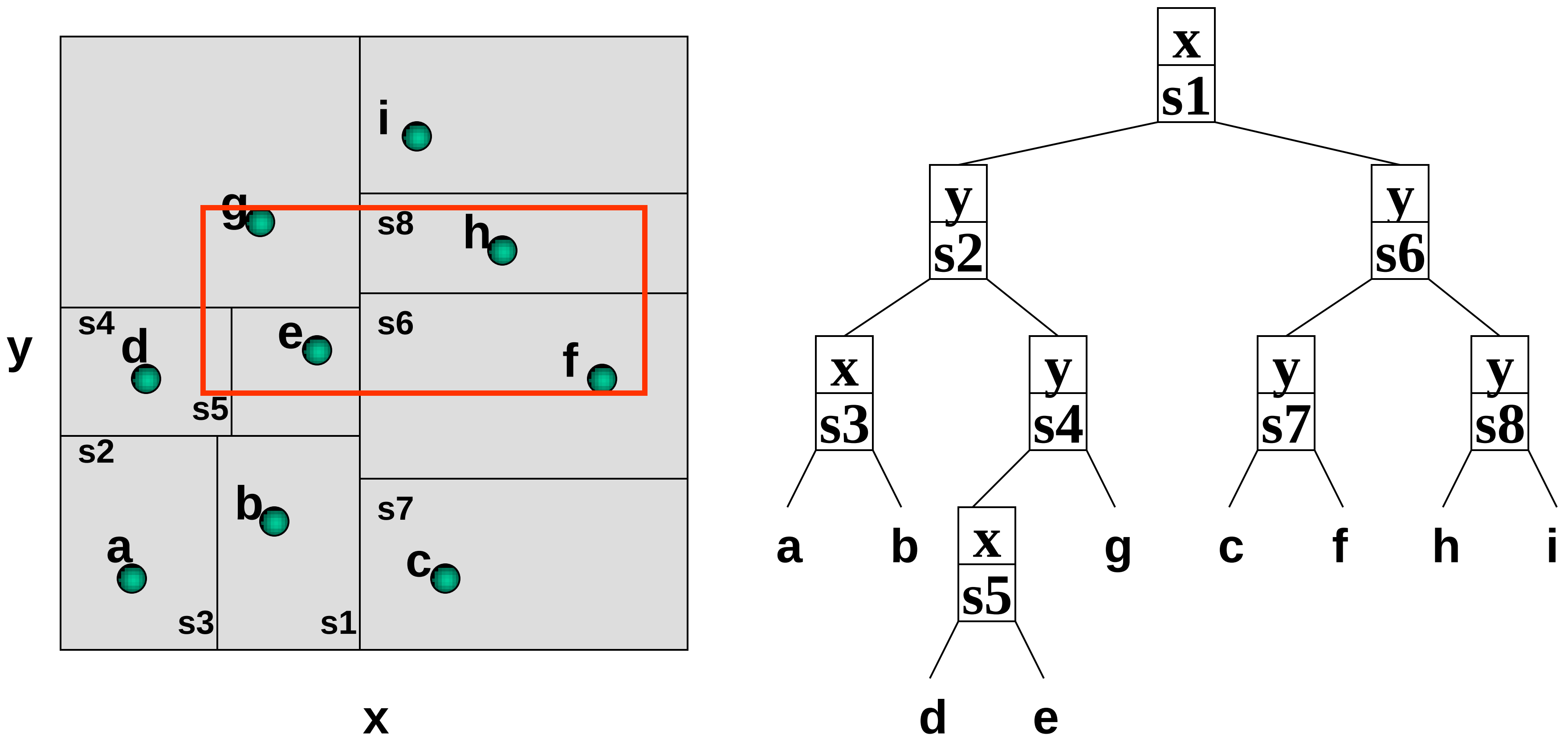


At each step divide perpendicular to the widest spread.

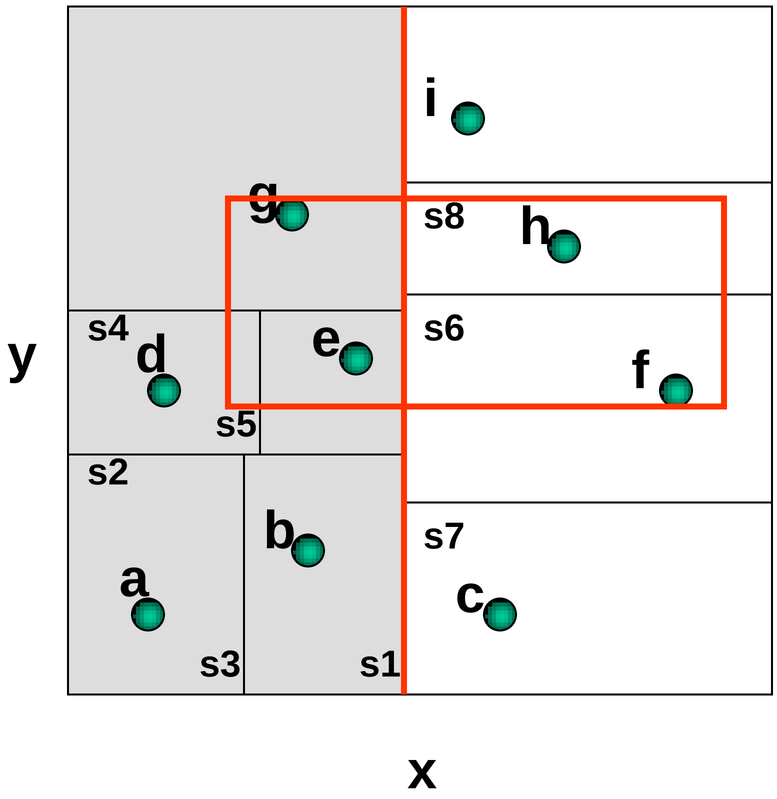
Rectangular Range Query

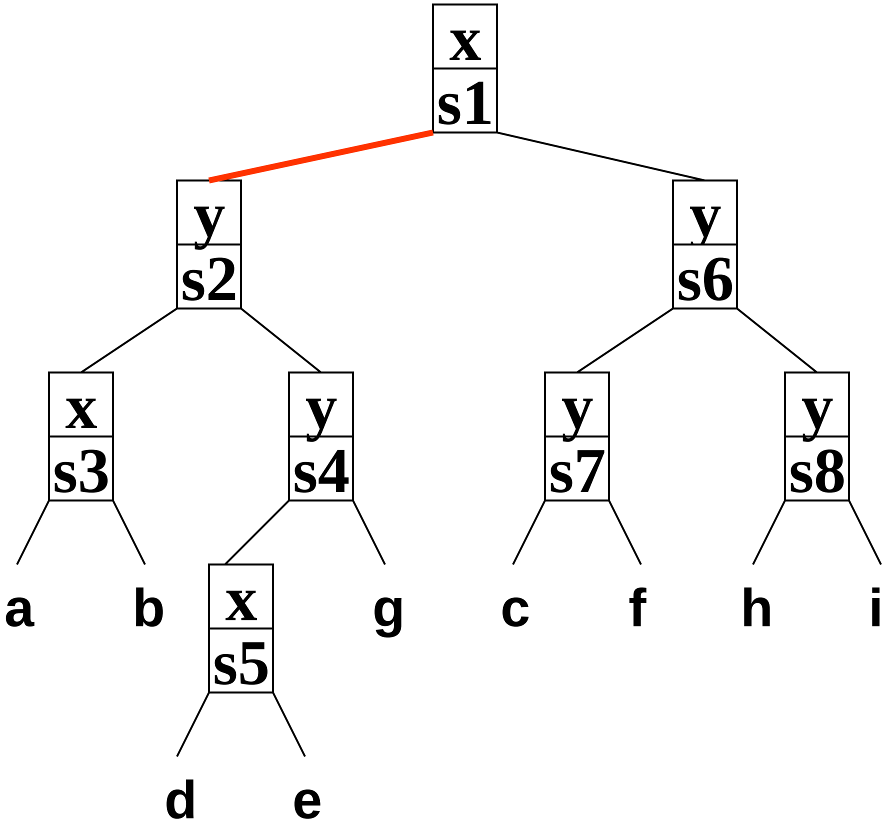
- Recursively search every cell that intersects the rectangle.

Rectangular Range Query

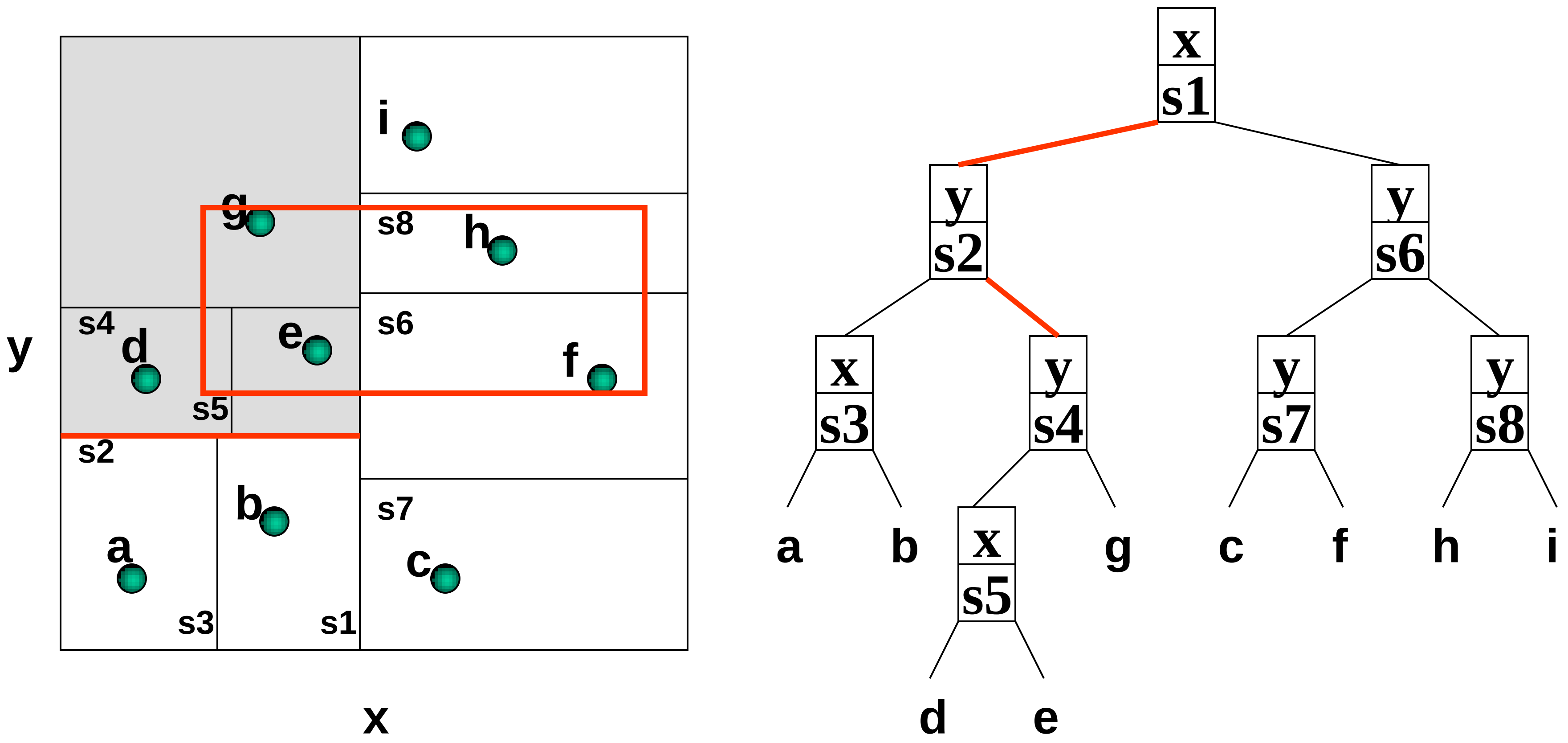


Rectangular Range Query

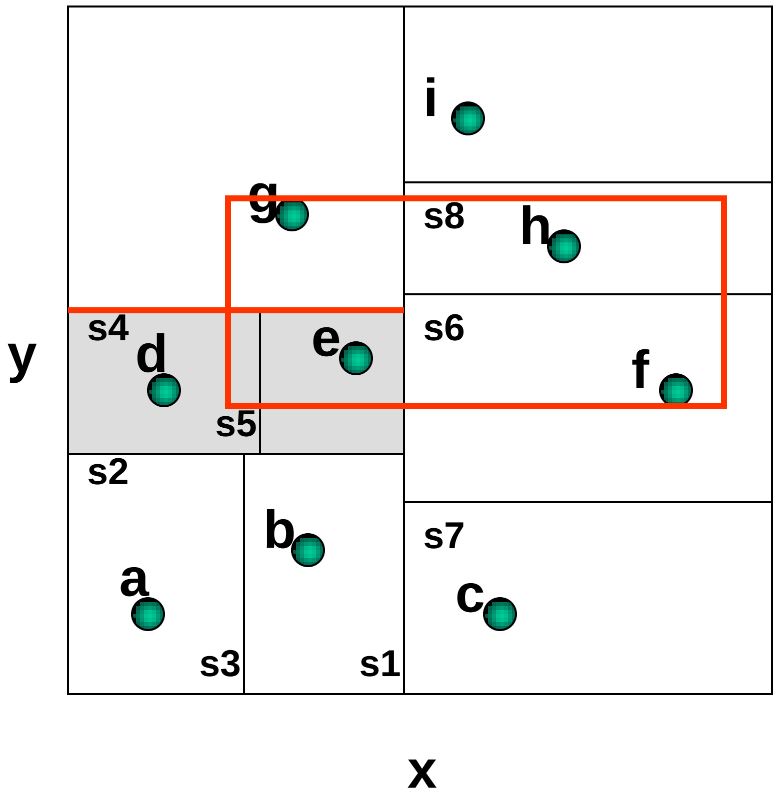


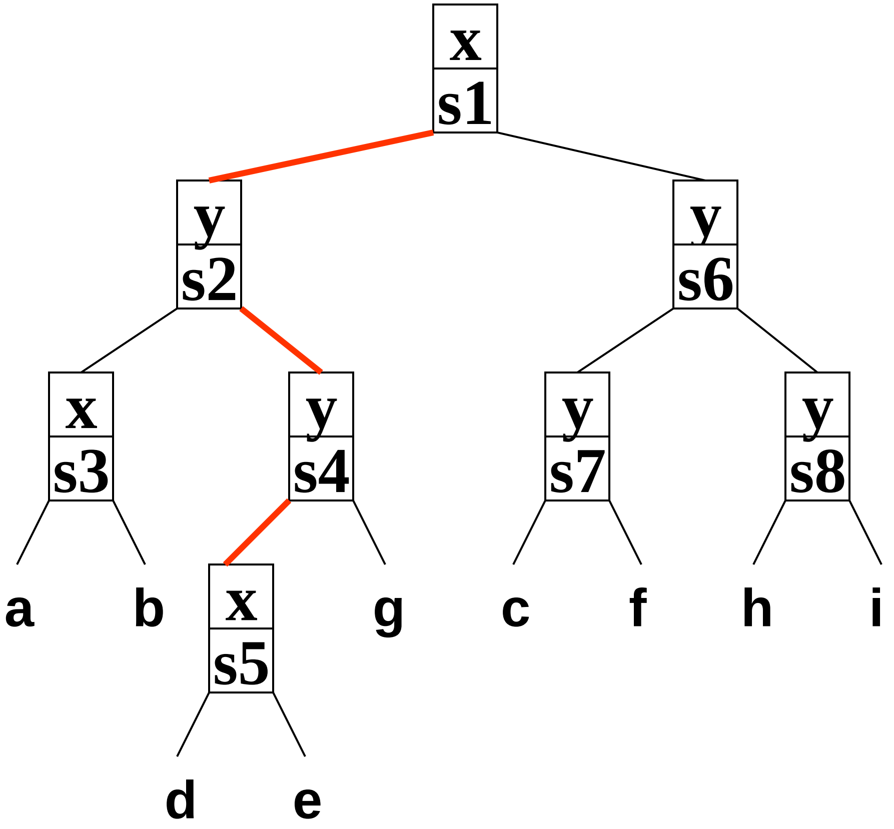


Rectangular Range Query

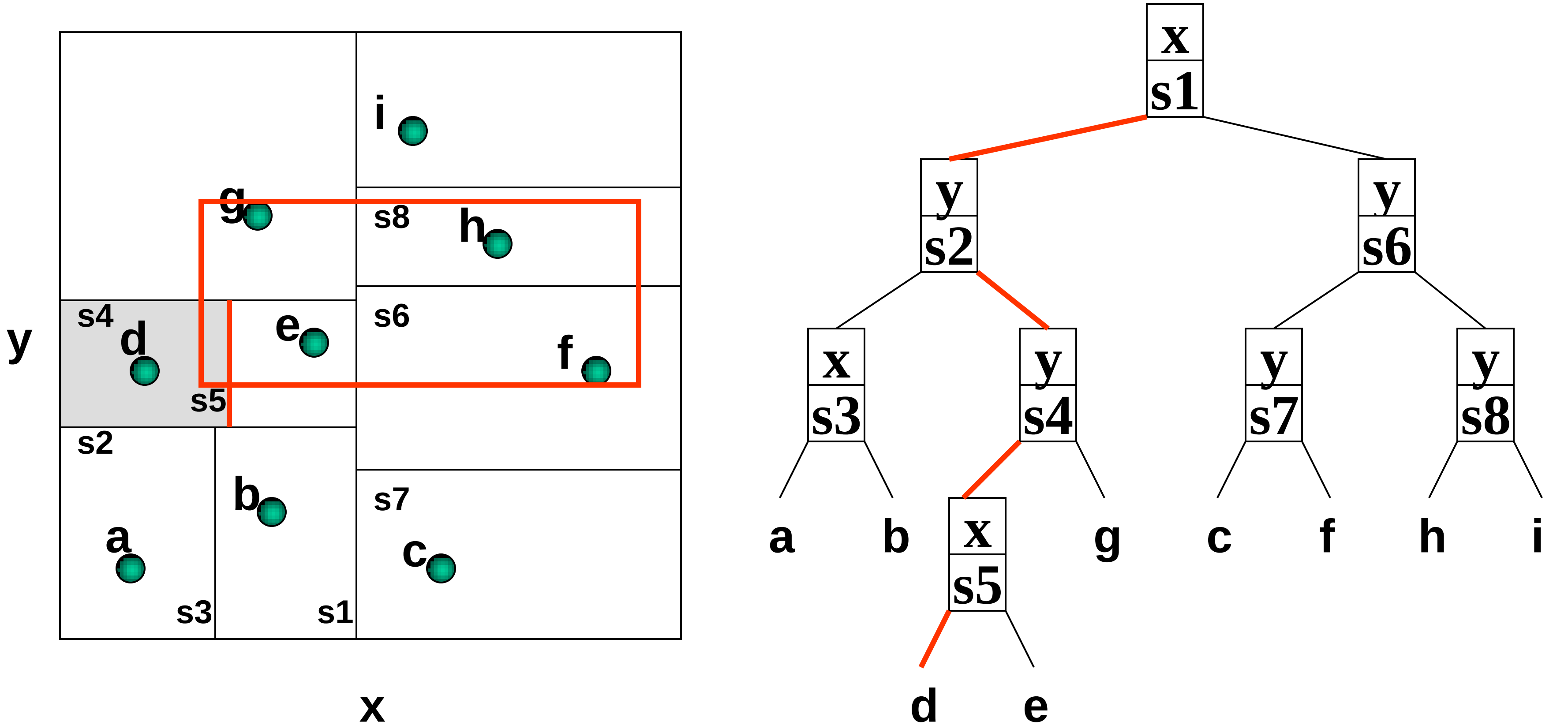


Rectangular Range Query

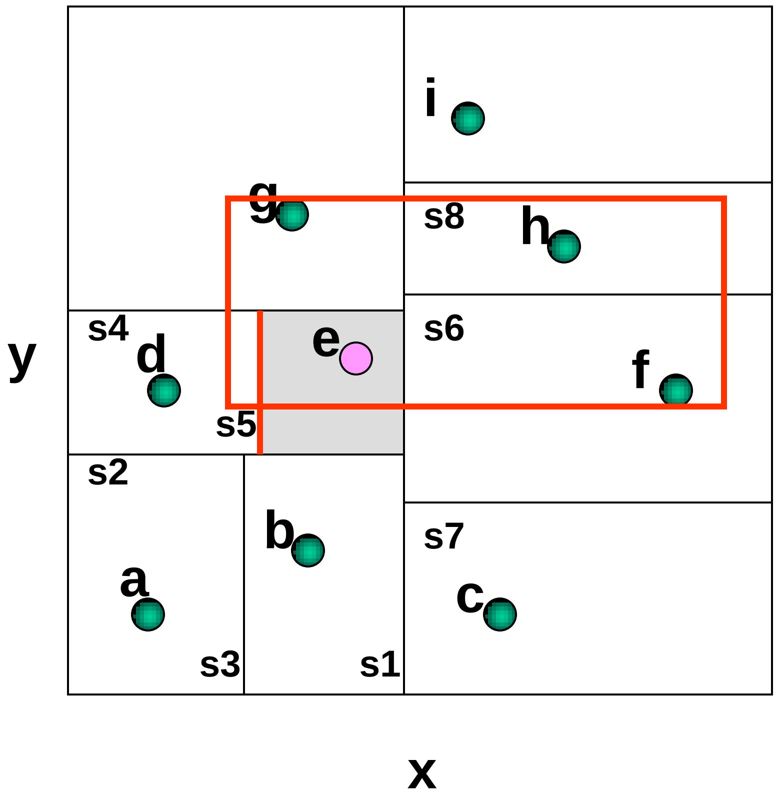


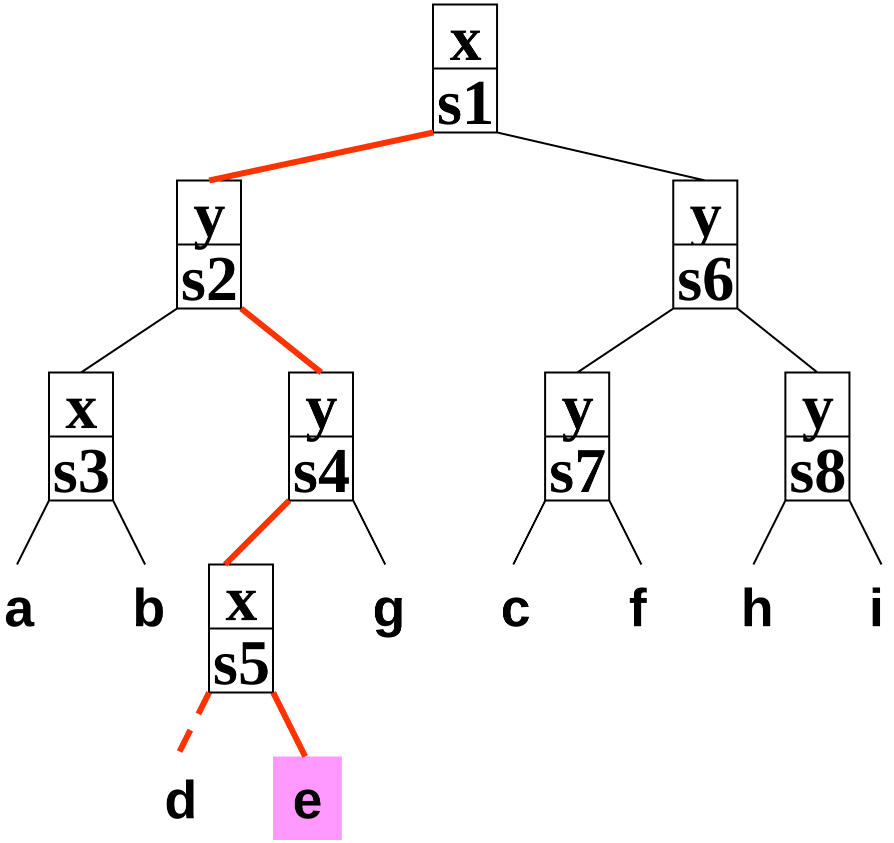


Rectangular Range Query

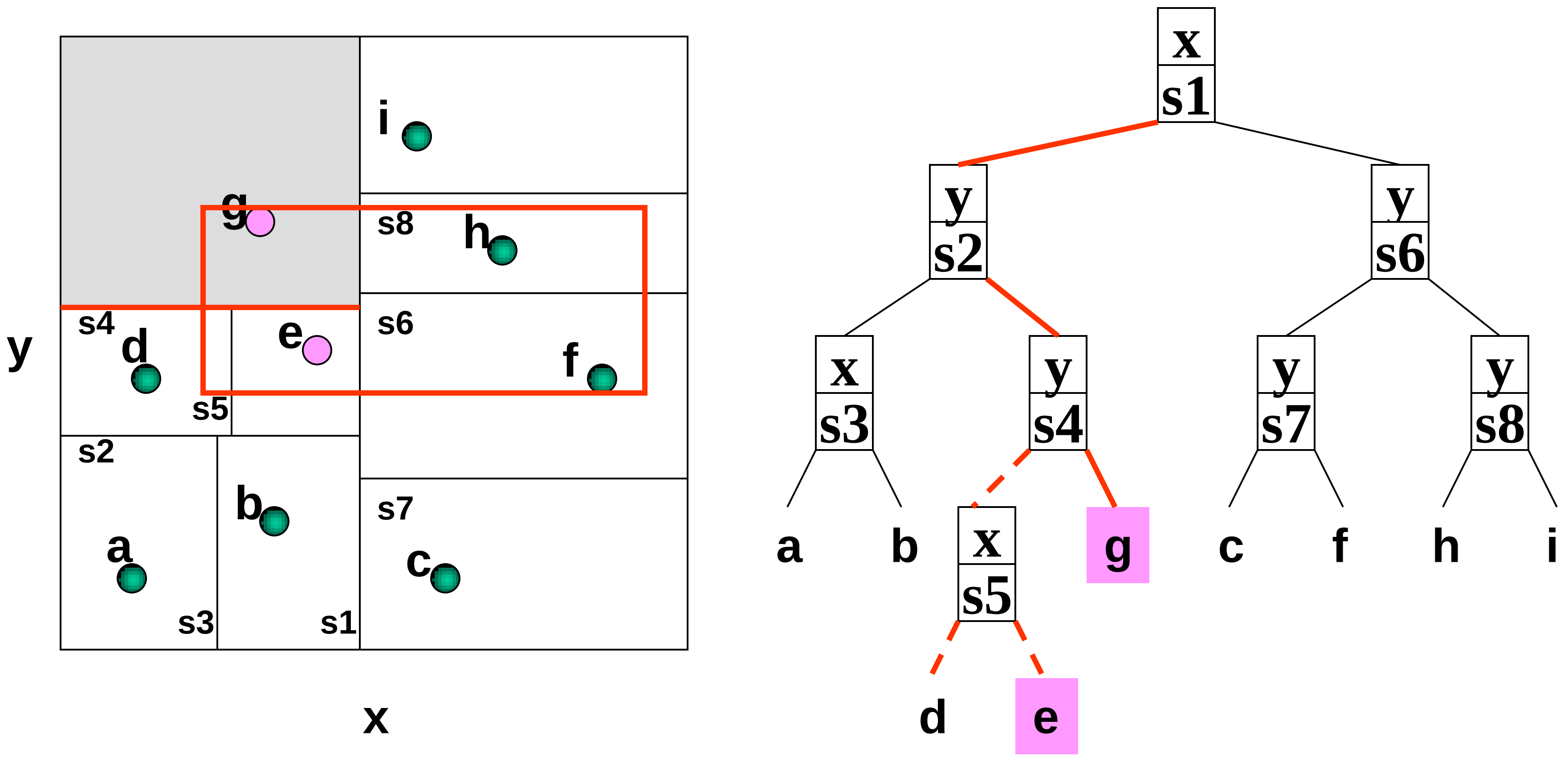


Rectangular Range Query

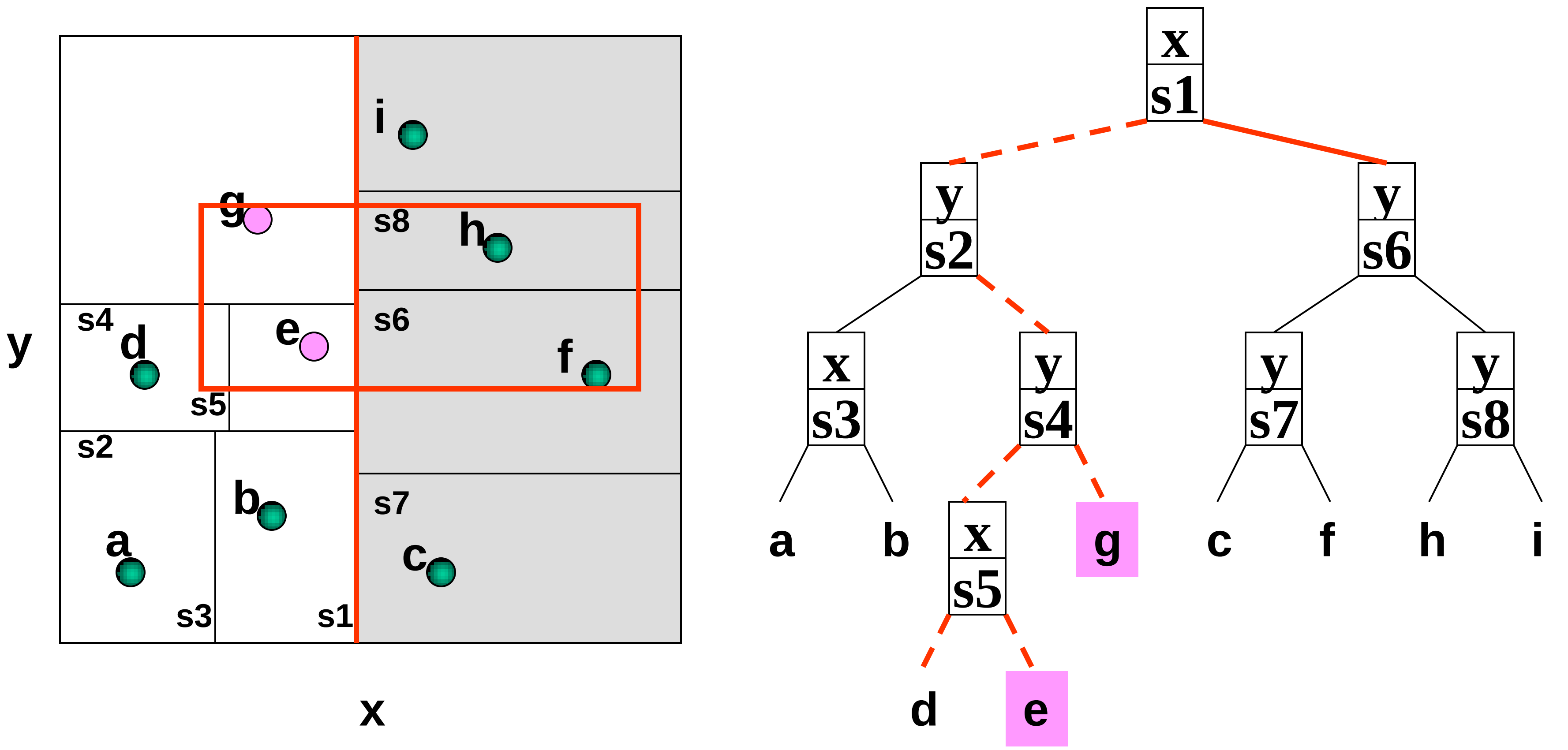




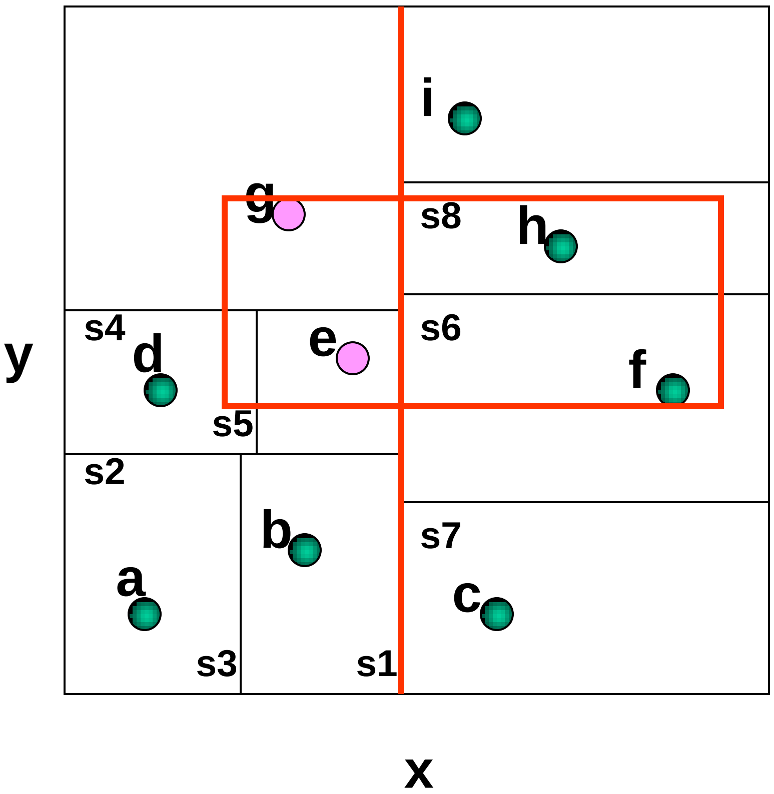
Rectangular Range Query

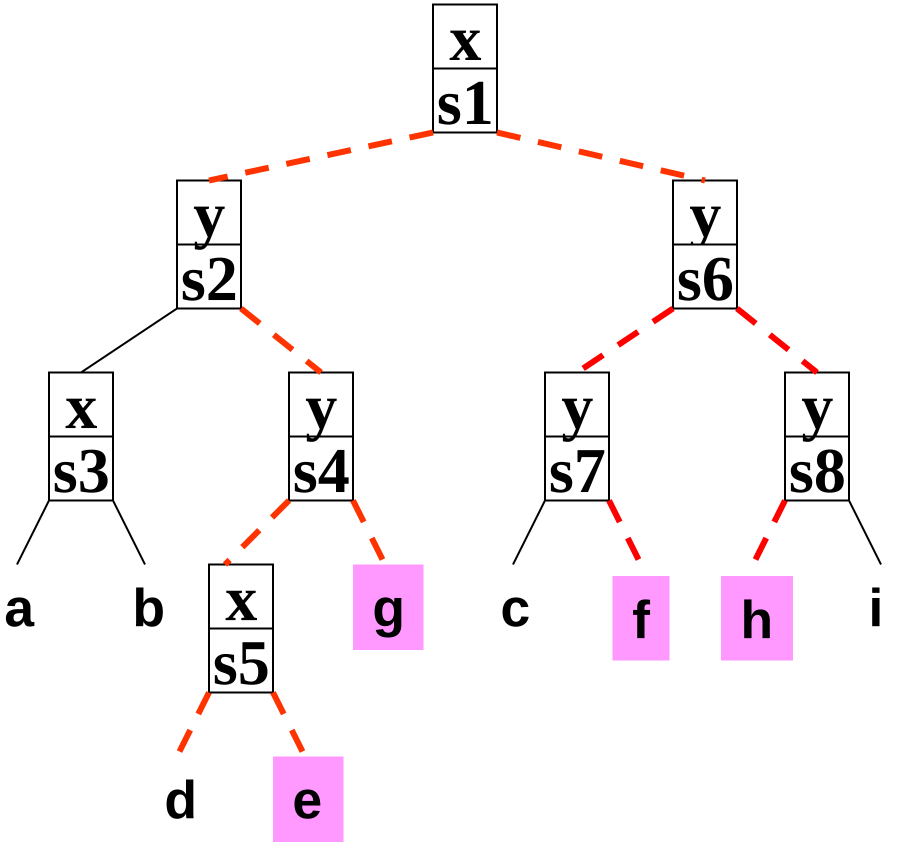


Rectangular Range Query

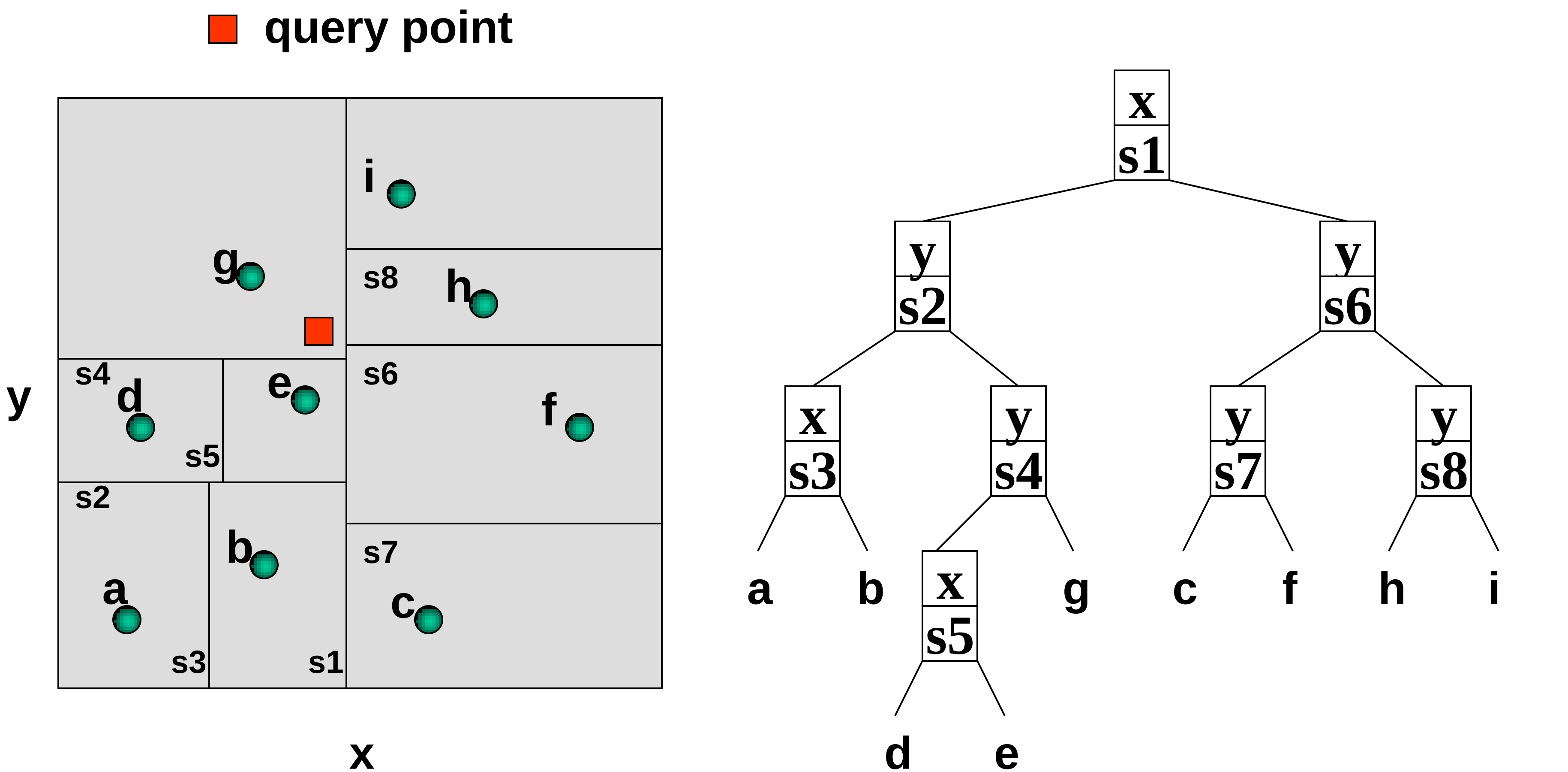


Rectangular Range Query

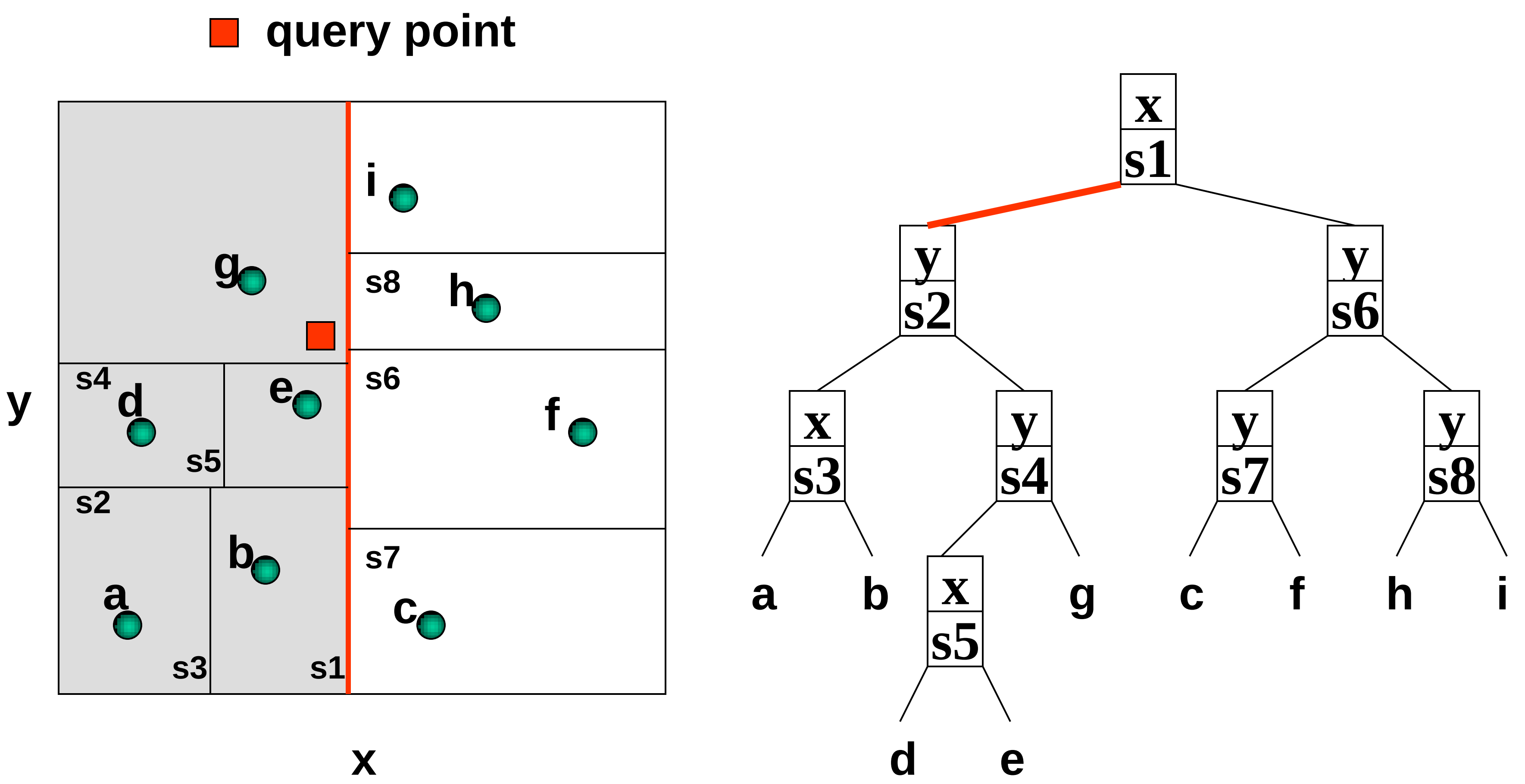




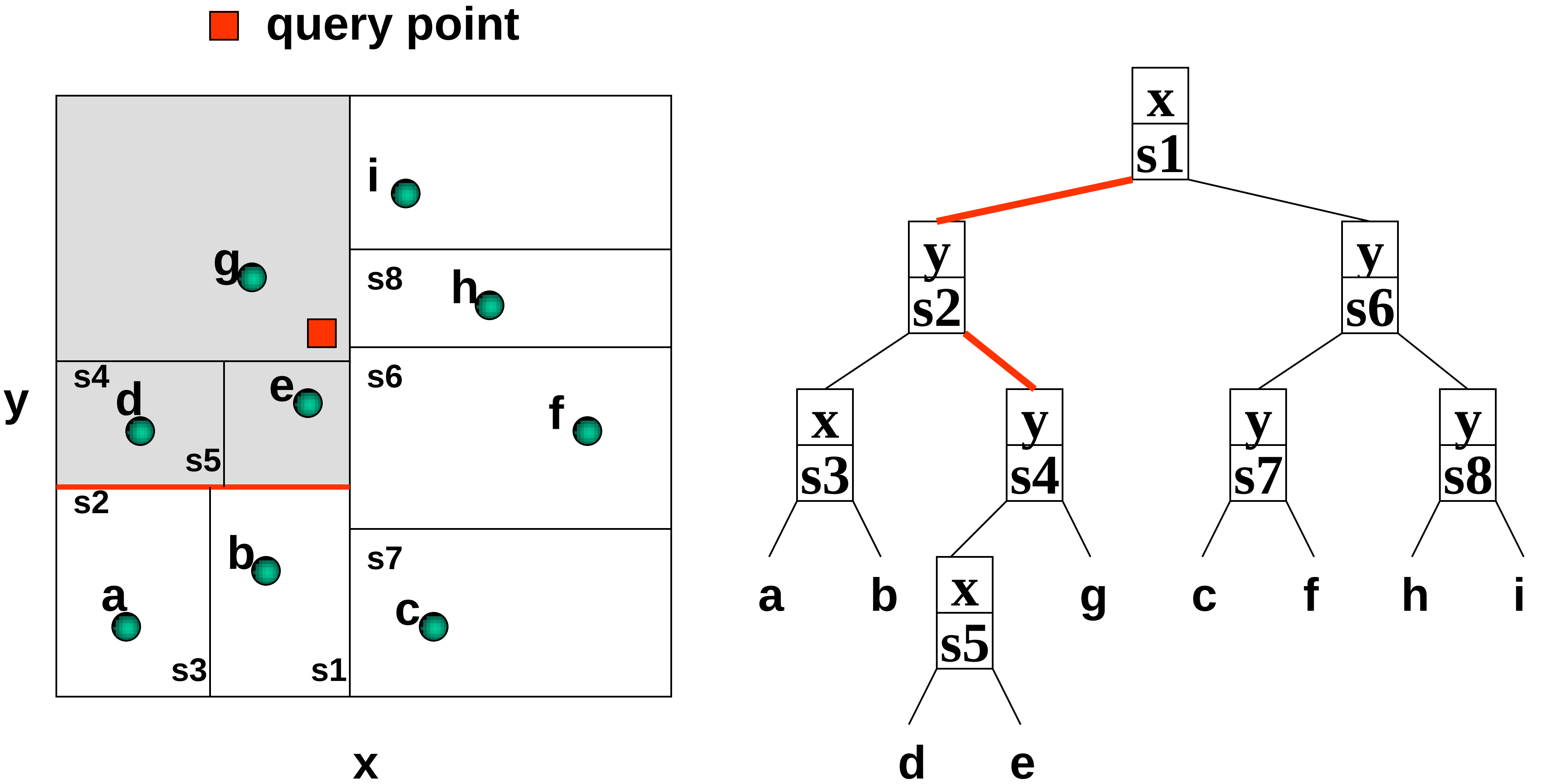
k-d Tree Nearest Neighbor Search



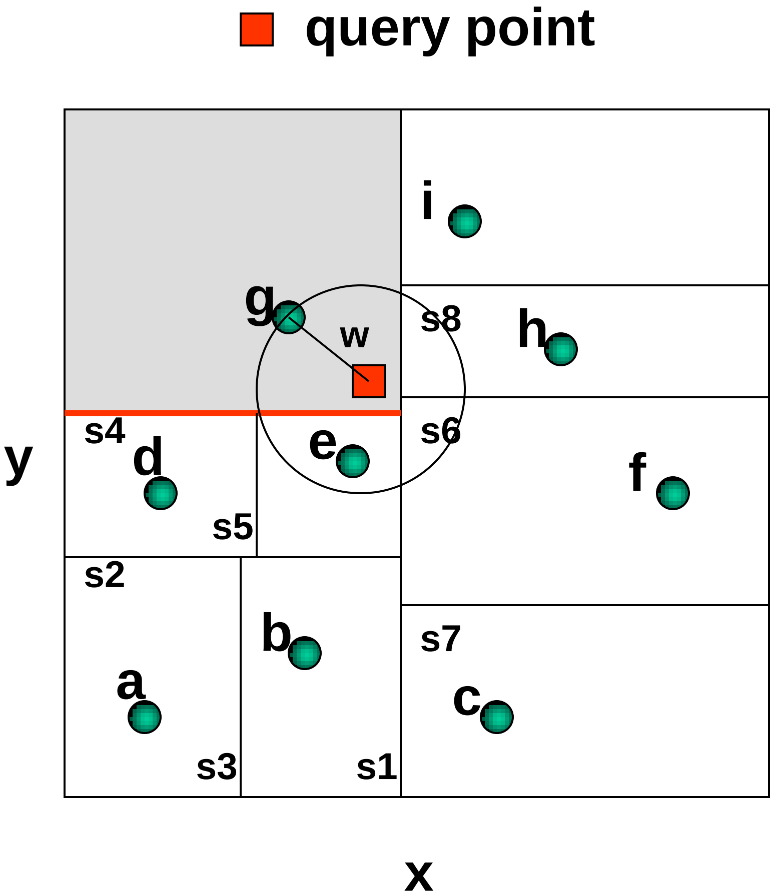
k-d Tree Nearest Neighbor Search

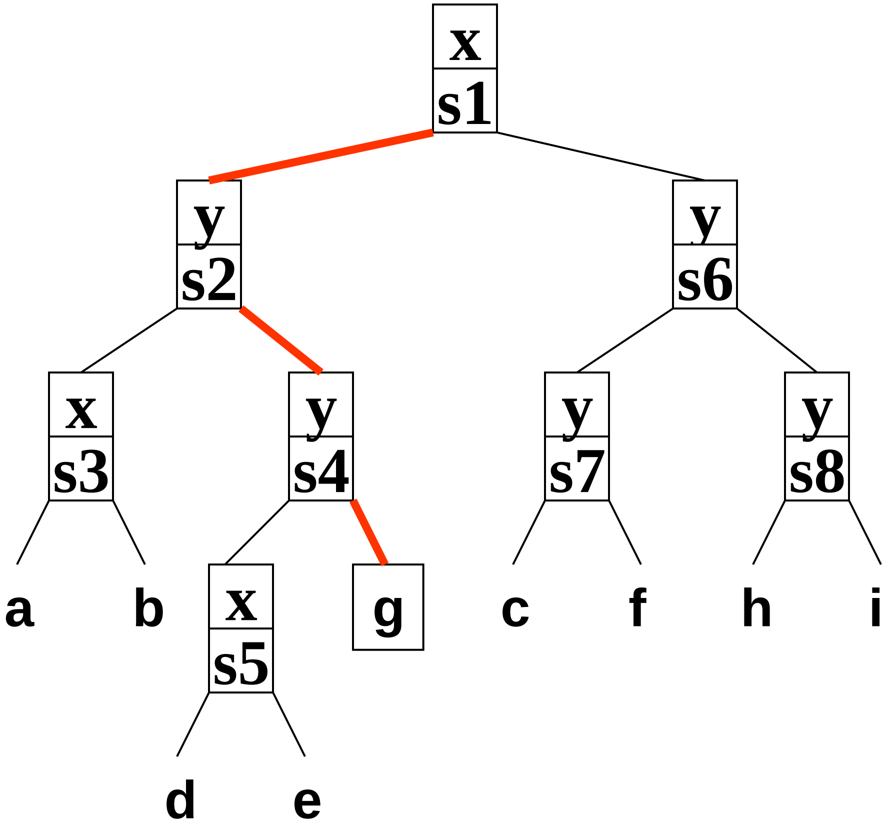


k-d Tree Nearest Neighbor Search

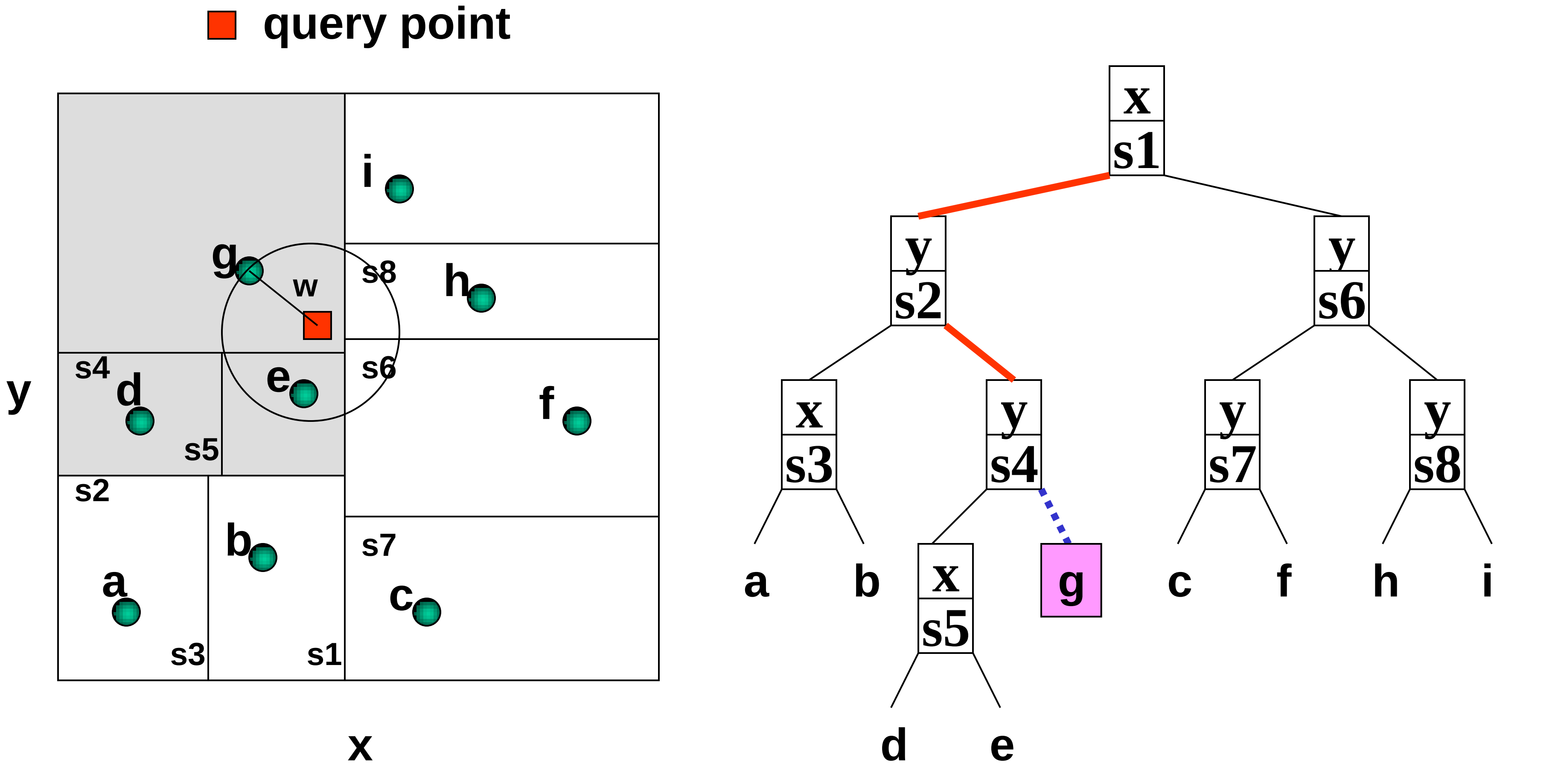


k-d Tree Nearest Neighbor Search



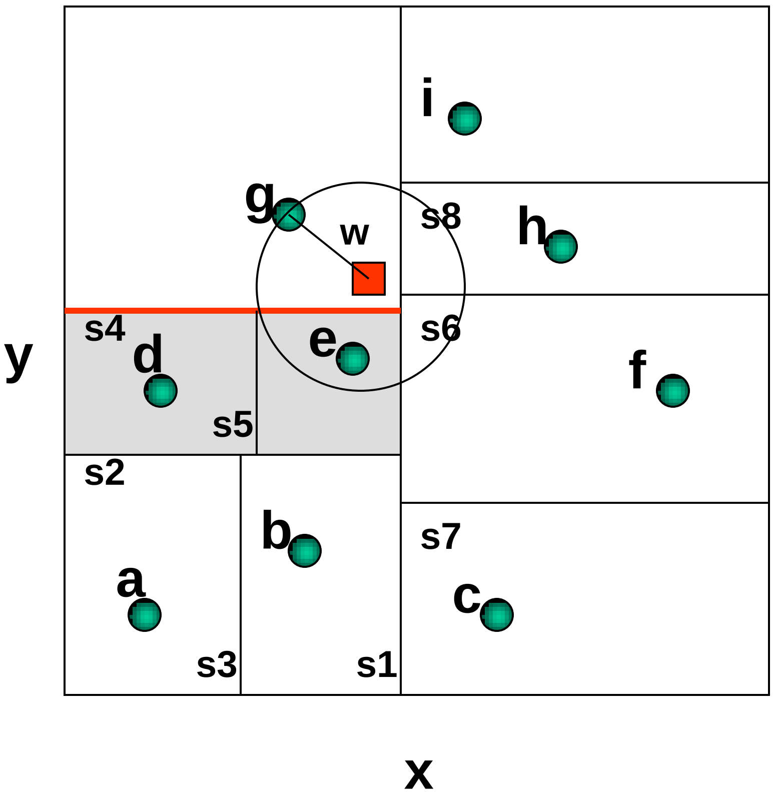


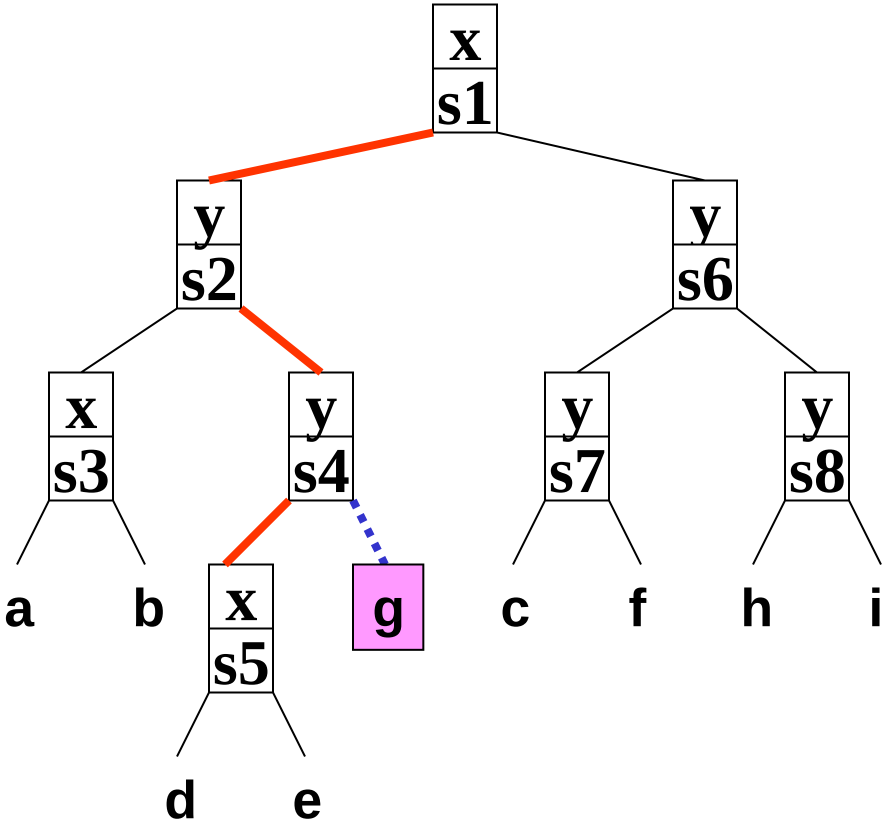
k-d Tree Nearest Neighbor Search



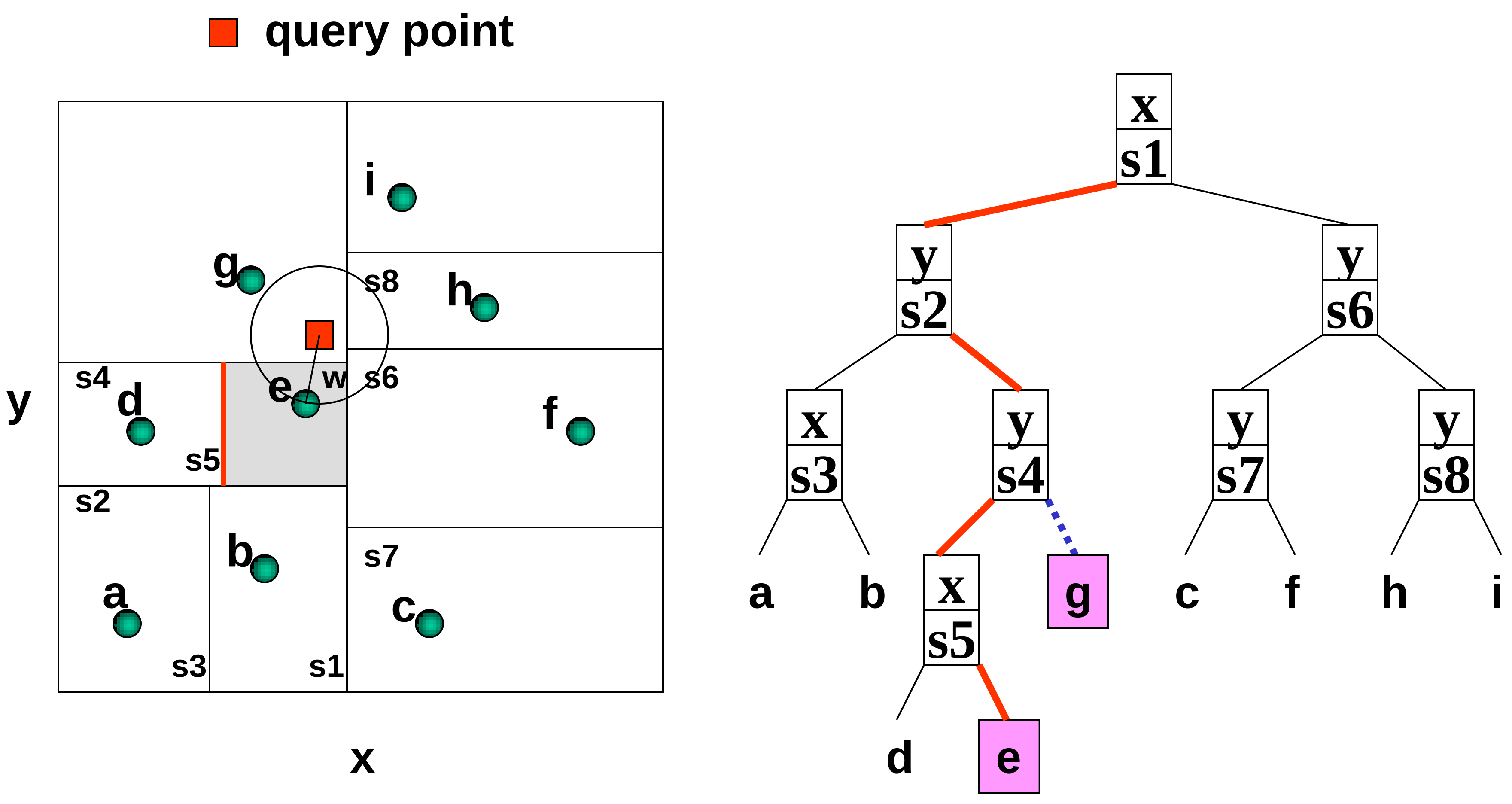
k-d Tree Nearest Neighbor Search

query point

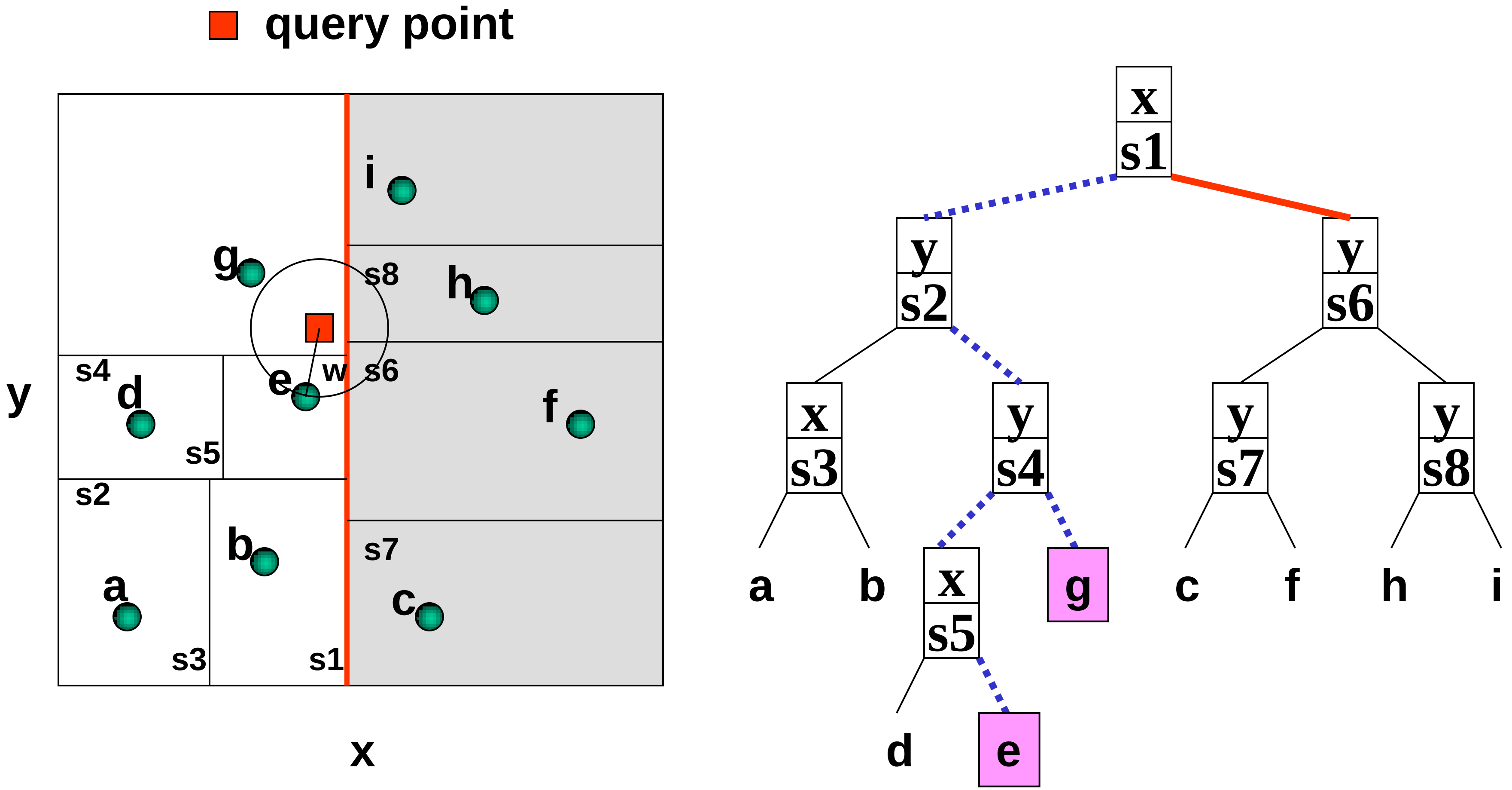




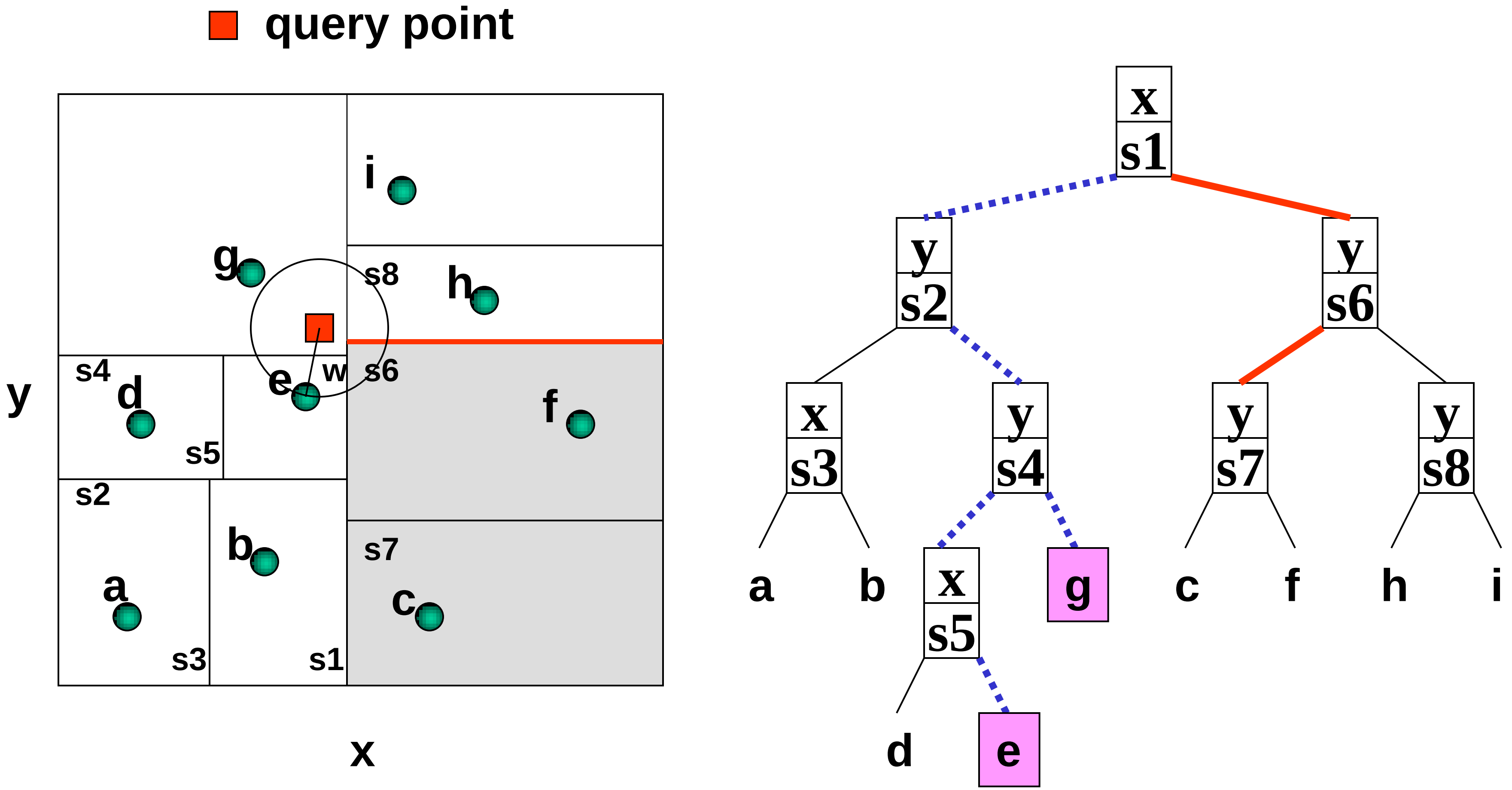
k-d Tree Nearest Neighbor Search



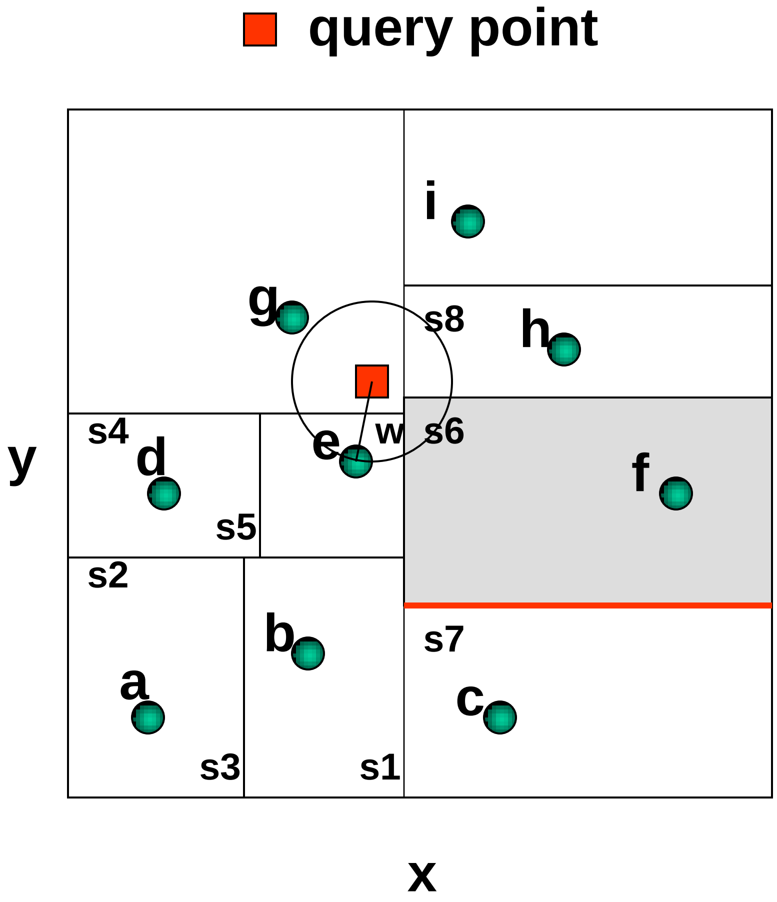
k-d Tree Nearest Neighbor Search

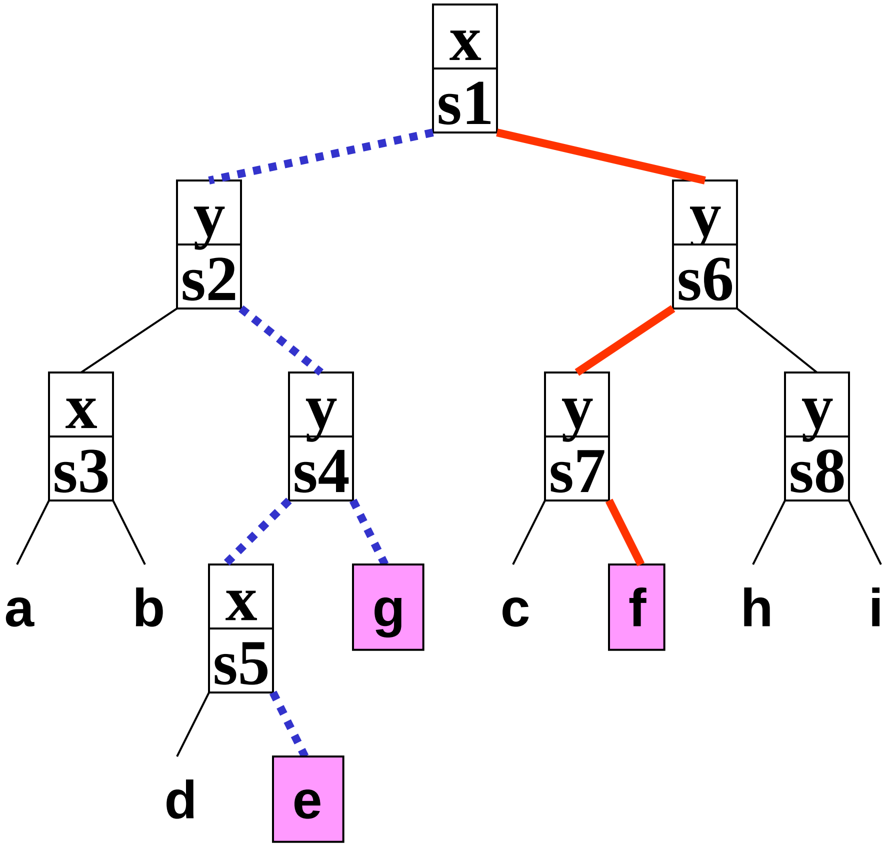


k-d Tree Nearest Neighbor Search

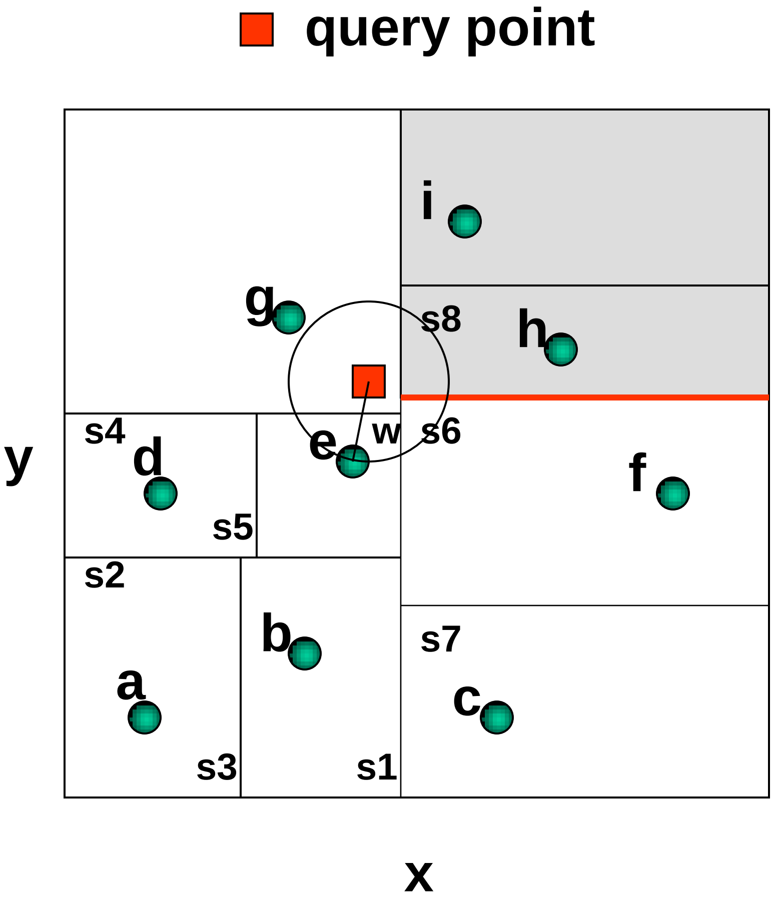


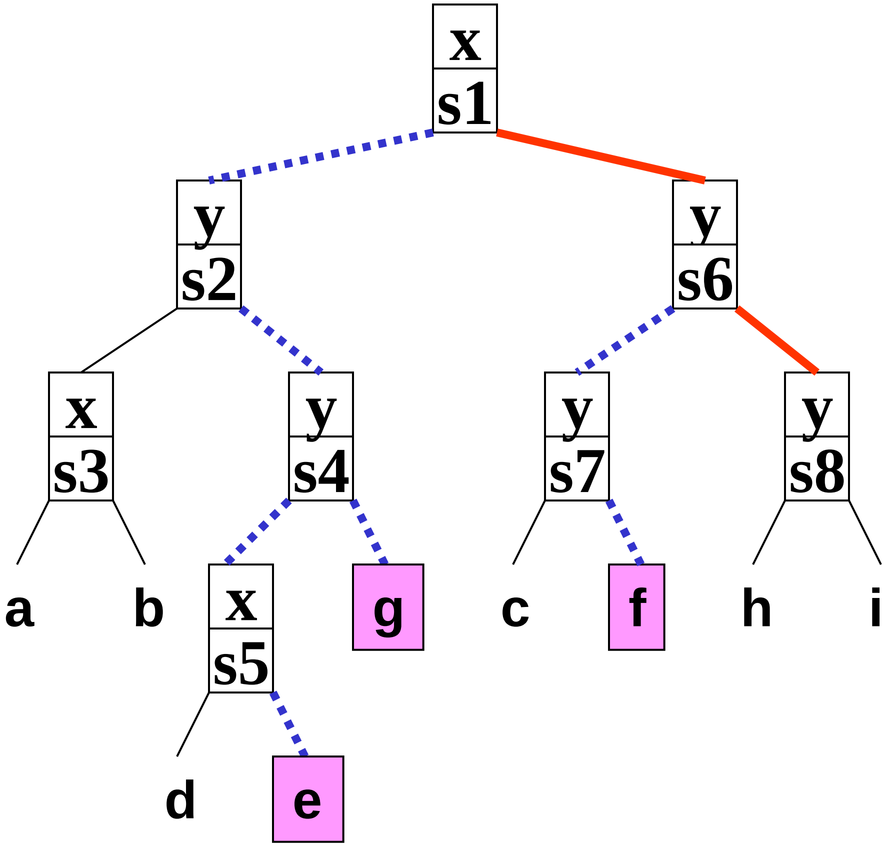
k-d Tree Nearest Neighbor Search



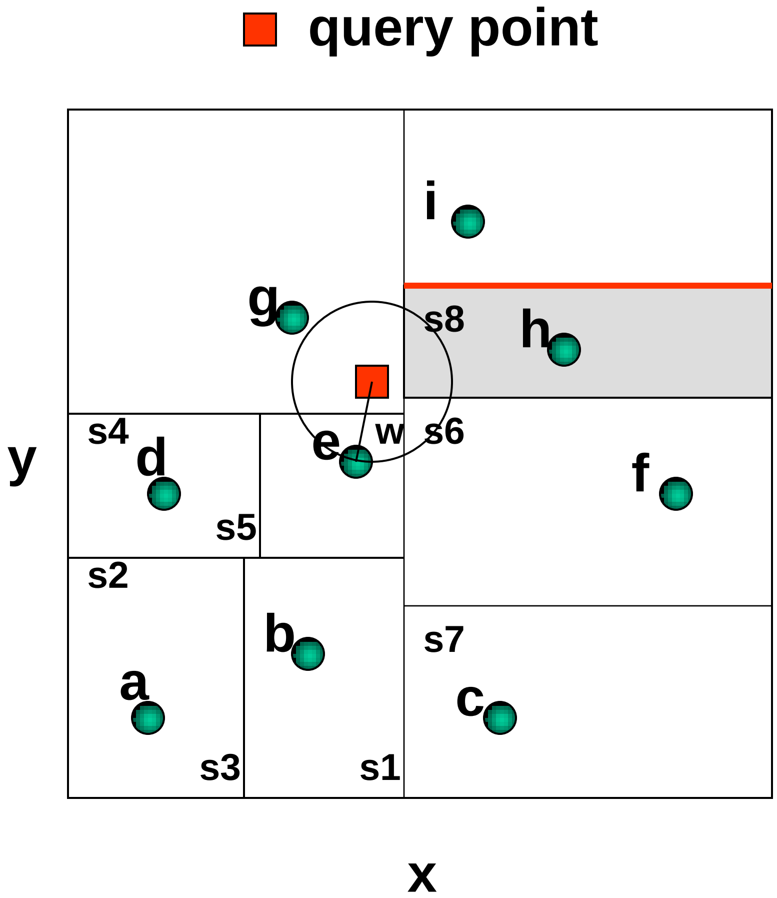


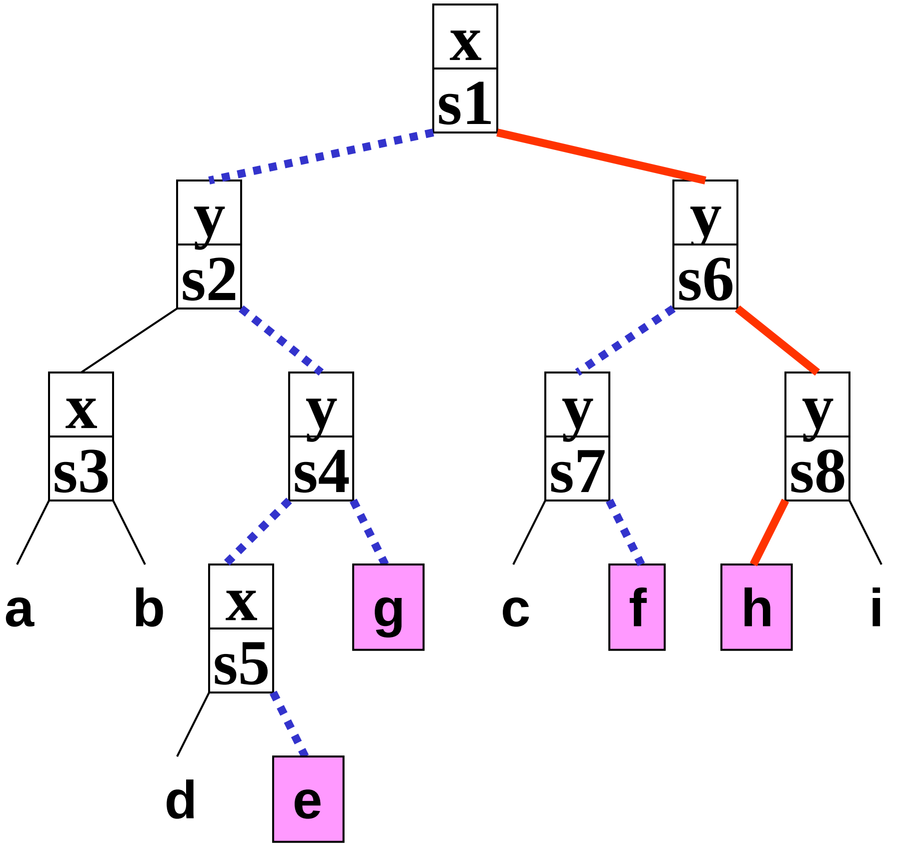
k-d Tree Nearest Neighbor Search



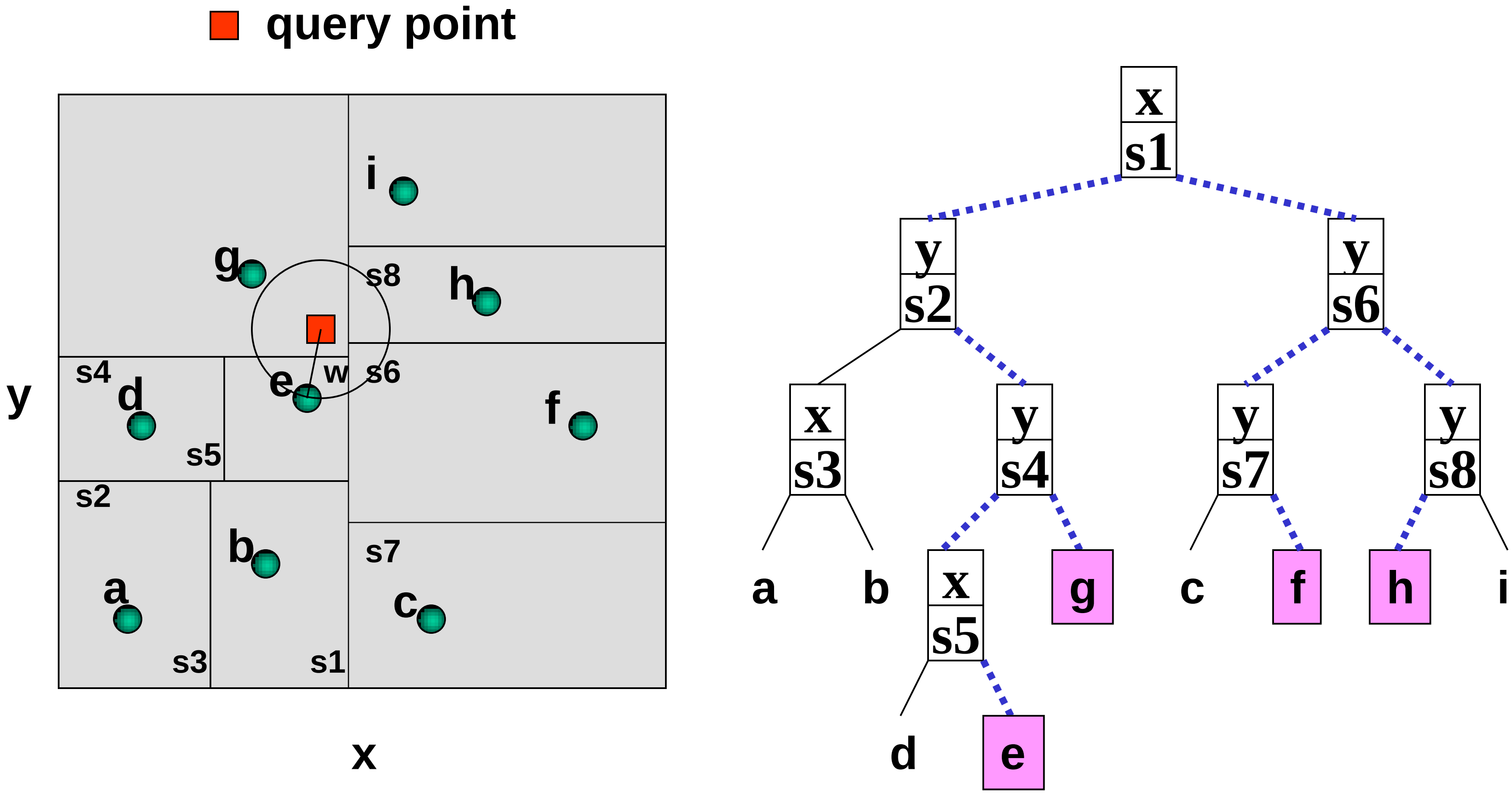


k-d Tree Nearest Neighbor Search

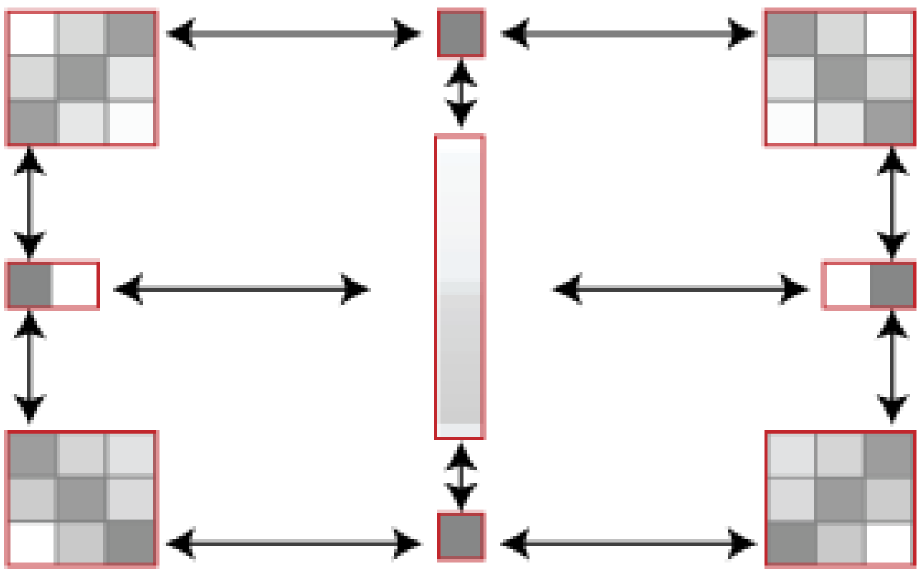




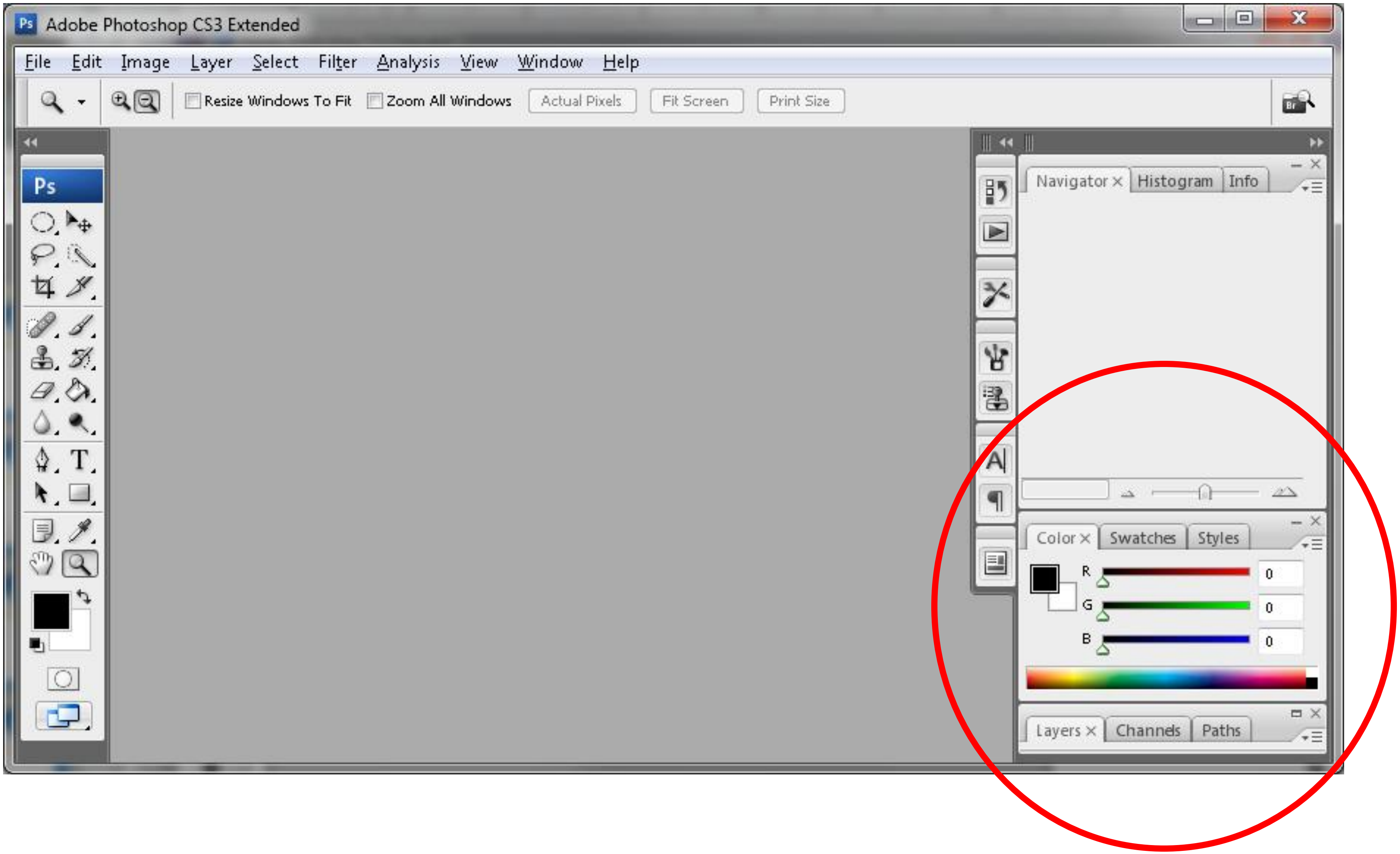
k-d Tree Nearest Neighbor Search



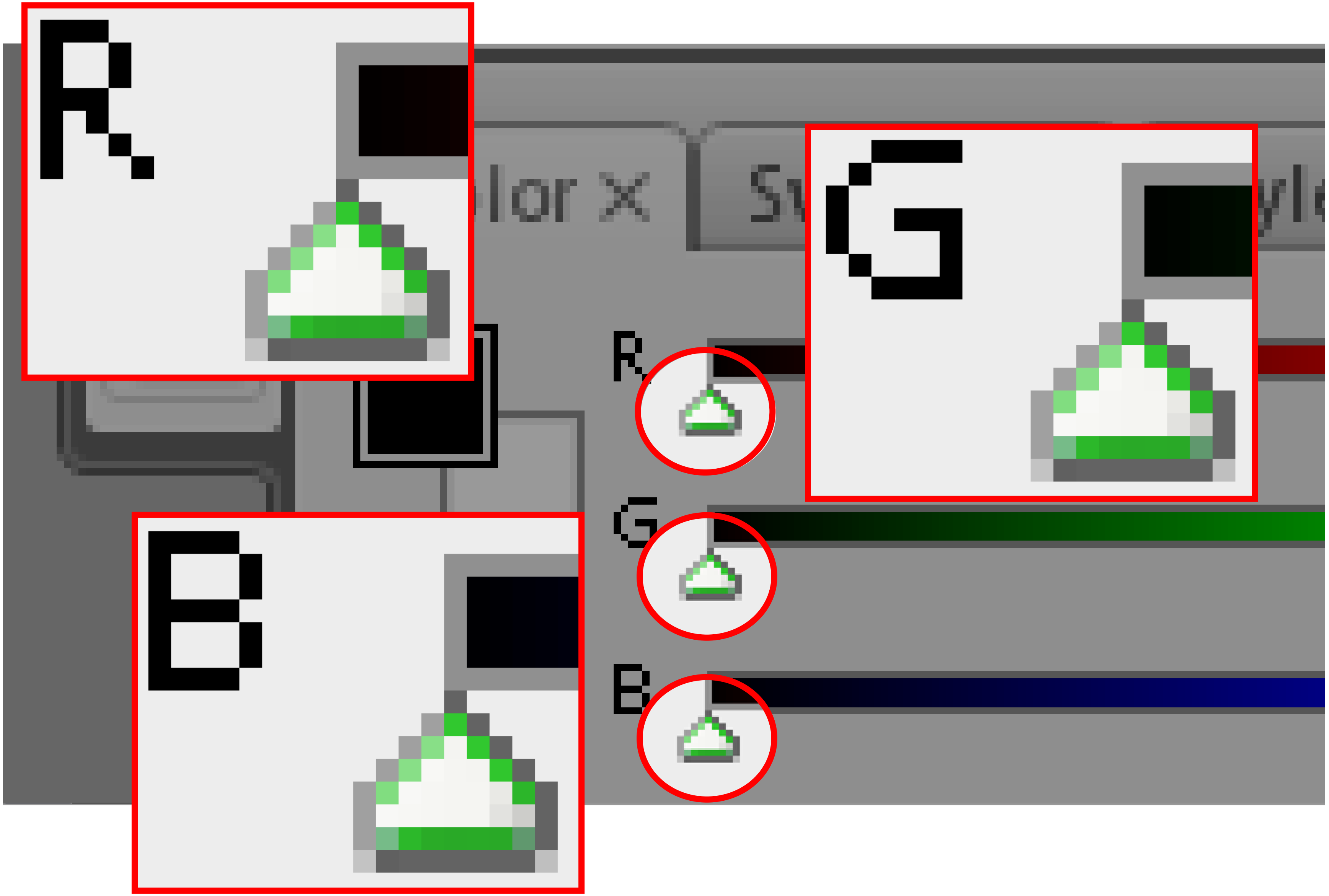
Prefab:
What if Anybody Could Modify Any Interface



CHI 2012, Dixon *et al.*
CHI 2011, Dixon *et al.*
CHI 2010, Dixon *et al.*



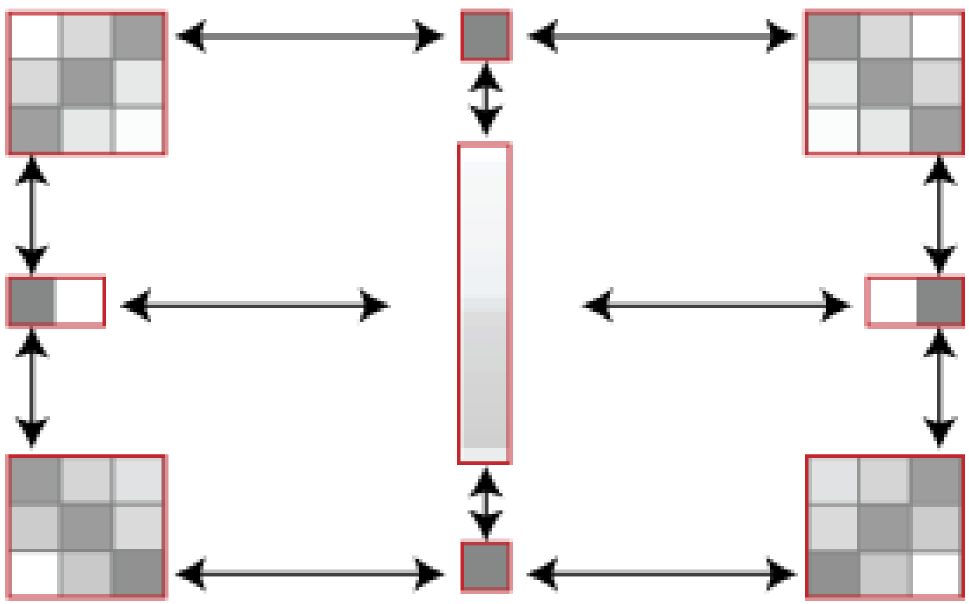
Adobe Photoshop CS3 Extended
File Edit Image Layer Select Filter Analysis View Window Help
Resize Windows To Fit
Zoom All Windows
Actual Pixels
Fit Screen
Print Size
Navigator
Histogram
Info
Color
Swatches
Styles
R
G
B
0
0
0
Layers
Channels
Paths


R
G
B
R
G
B

Prefab CHI 2011 Video

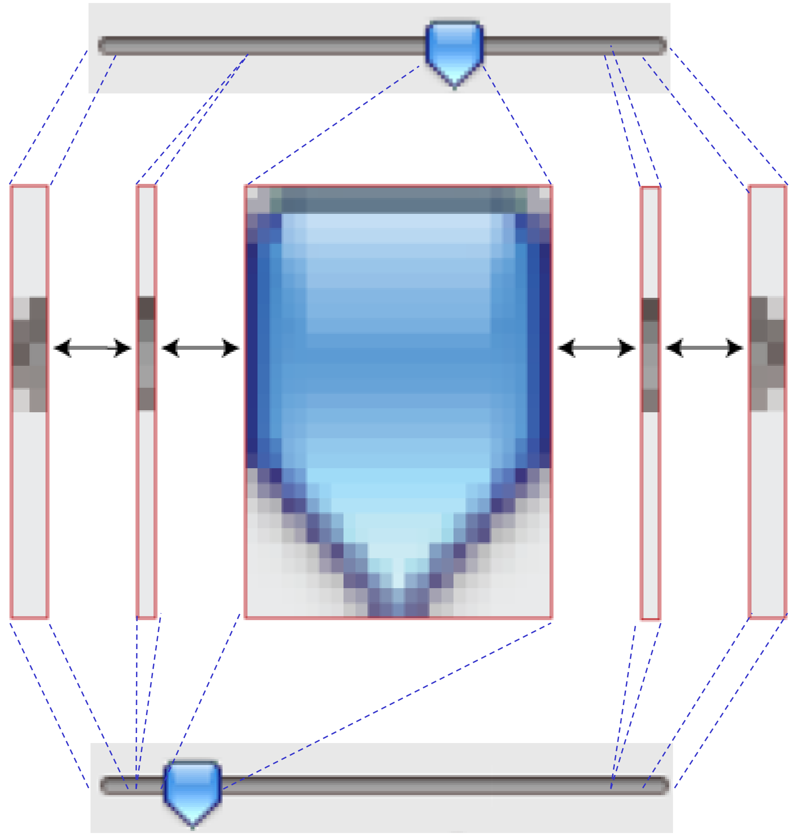
Decomposing Widgets into Their Parts

Windows Vista Steel Button Prototype



Decomposing Widgets into Their Parts

Mac Slider Prototype



Problem

- Efficiently match a large library of “pixel patches” in images

- Can break this down as a dictionary matching problem
  - Match a dictionary of strings in text

Aho-Corasick Algorithm

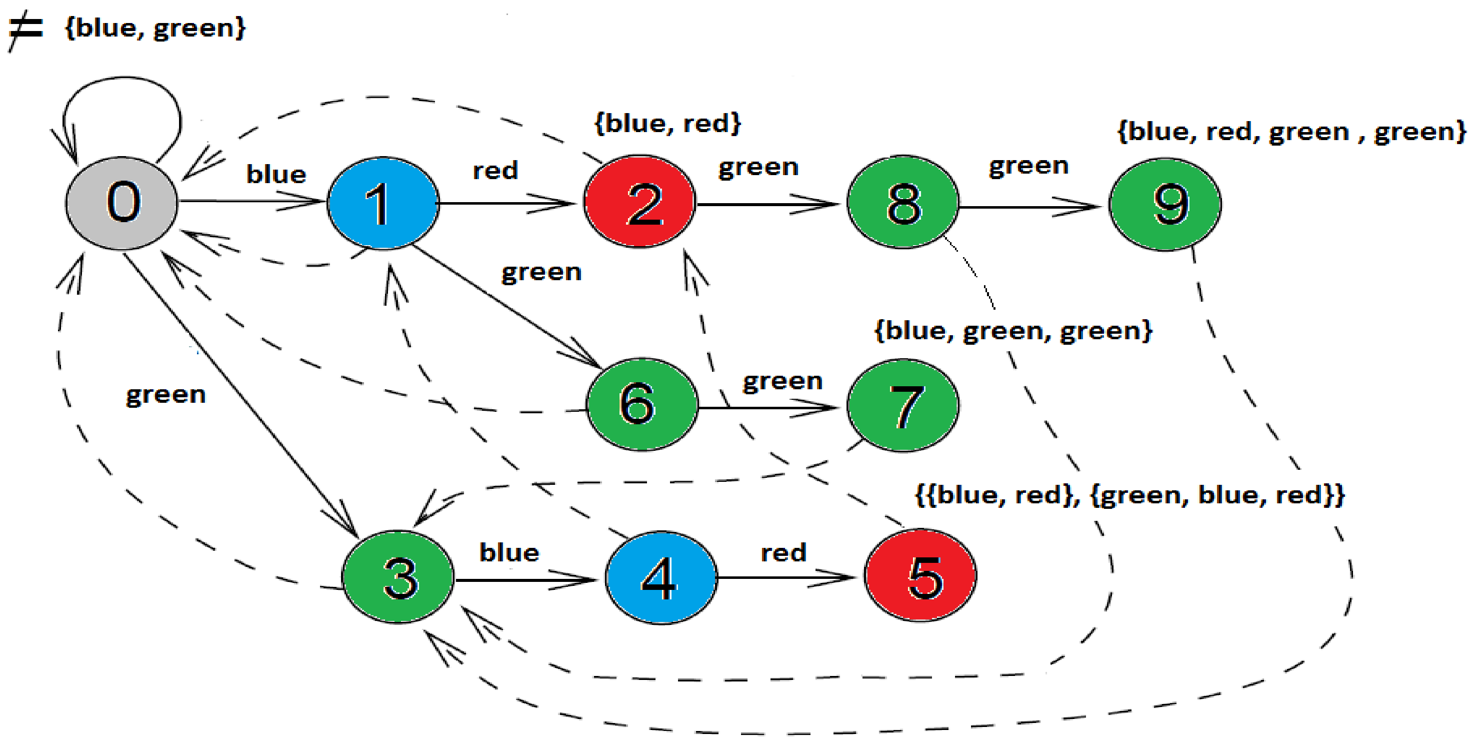
- Pre-process dictionary to create a finite state machine
  - Follow an edge for ever ‘character’
  - Output any ‘strings’ when arriving at a node

- Linear in the length of the ‘text’ we examine
  - Ignoring pre-processing (which is fine in this application)

Aho-Corasick Algorithm

- Matching a dictionary of pixel rows:

  {blue, red}, {blue, green, green},
  {blue, red, green, green}, {green, blue, red}



Moral of the Story