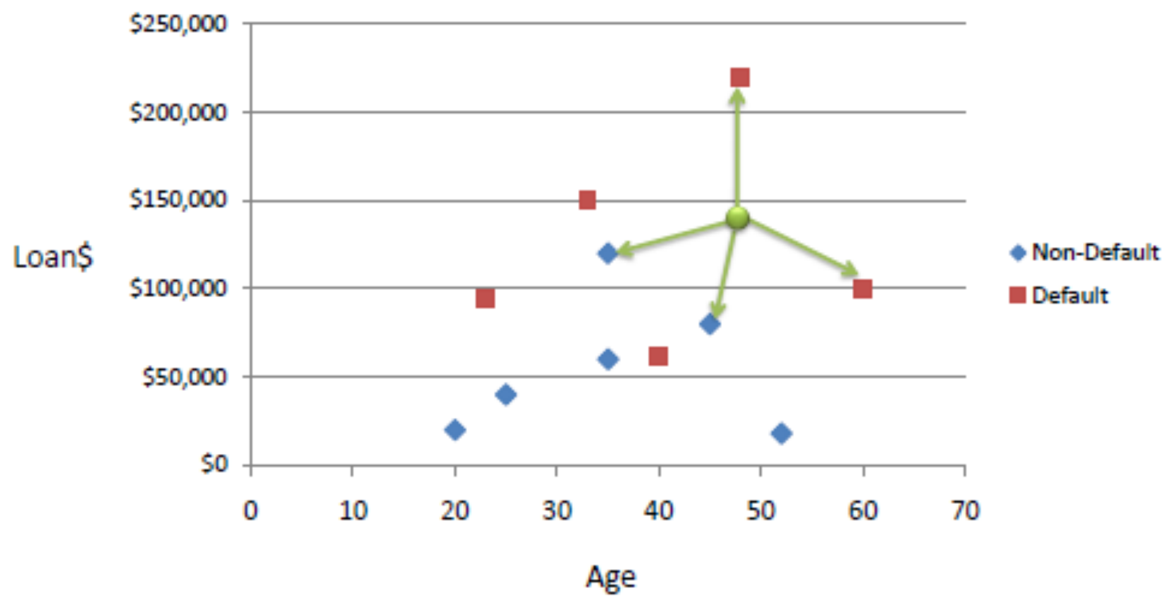
K Nearest Neighbor Classification

plus similarity and distance calculations

K Nearest Neighbor Classification

- K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions).
- A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function.



Type of Data in Clustering Analysis

- Interval-scaled variables
- Binary variables
- Nominal, and ordinal variables
- Variables of mixed types
- Text
- Temporal

Standardize Numeric Data

- Standardize data
  - Calculate the mean absolute deviation:

$$s_f = \frac{1}{n}(|x_{1f} - m_f| + |x_{2f} - m_f| + ... + |x_{nf} - m_f|)$$

  Where

$$m_f = \frac{1}{n}(x_{1f} + x_{2f} + ... + x_{nf}).$$

  - Calculate the standardized measurement (*z-score*)

$$z_{if} = \frac{x_{if} - m_f}{s_f}$$

- Normalizing data

$$z_{if} = \frac{x_{if} - m_f}{\sigma_f}$$

Similarity/Dissimilarity Between Objects

- Distances are normally used to measure the similarity or dissimilarity between two data objects
- Some popular ones include: *Minkowski distance*:

$$d(i,j) = \sqrt[q]{(|x_{i_1} - x_{j_1}|^q + |x_{i_2} - x_{j_2}|^q + \ldots + |x_{i_p} - x_{j_p}|^q)}$$

Where $i = (x_{i1}, x_{i2}, \ldots, x_{ip})$ and $j = (x_{j1}, x_{j2}, \ldots, x_{jp})$ are two p-dimensional data objects, and q is a positive integer

- If $q = 1$, d is Manhattan distance

$$d(i,j) = |x_{i_1} - x_{j_1}| + |x_{i_2} - x_{j_2}| + \ldots + |x_{i_p} - x_{j_p}|$$

Similarity/Dissimilarity Between Objects

If q = *2*, *d* is Euclidean distance:

$$d(i,j) = \sqrt{(|x_{i_1} - x_{j_1}|^2 + |x_{i_2} - x_{j_2}|^2 + \ldots + |x_{i_p} - x_{j_p}|^2)}$$

- Properties
  - *d(i,j)* $\geq$ 0
  - *d(i,i)* = 0
  - *d(i,j)* = *d(j,i)*
  - *d(i,j)* $\leq$ *d(i,k)* + *d(k,j)*

Triangular inequality

Other Similarity/Distance measures

- Sets as vectors: measure similarity by the cosine distance.

$$x_i = \left[x_{i1}, x_{i2}, \ldots x_{ip} \right]$$

$$x_j = \left[x_{j1}, x_{j2}, \ldots x_{jp} \right]$$

$$\cos(x_i, x_j) = \frac{x_i \bullet x_j}{|x_i| * |x_j|} = \hat{x}_i \bullet \hat{x}_j$$

7

Similarity/Dissimilarity for Binary Data

- Symmetric attribute: both states are equally valuable, carrying the same weight, e.g., gender 0, 1 → coding each state to be 0 or 1 arbitrarily
- Asymmetric attribute: outcomes of the states are not equally important, e.g., outcome of a medical test → positive or negative
  - assign the more important outcome to value 1
  - assign the less important outcome to value 0
  - e.g., 1 : HIV positive, 0: HIV negative. The agreement of two patients having "1"s for this attribute is more significant then an agreement of "0"s.

Similarity/Dissimilarity for Binary Data

- A contingency table for binary data

		Object j		
		1	0	*sum*
	1	a	b	$a+b$
Object i	0	c	d	$c+d$
	sum	$a+c$	$b+d$	p

- Simple matching coefficient (if the binary variable is *symmetric*):

$$d(i,j) = \frac{b+c}{a+b+c+d}$$

- Jaccard coefficient (if the binary variable is *asymmetric*):

$$d(i,j) = \frac{b+c}{a+b+c}$$

Dissimilarity between Binary Variables

- Example

Name	Gender	Fever	Cough	Test-1	Test-2	Test-3	Test-4
Jack	M	Y	N	P	N	N	N
Mary	F	Y	N	P	N	P	N
Jim	M	Y	P	N	N	N	N

  - gender is a symmetric attribute
  - the remaining attributes are asymmetric binary
  - let the values Y and P be set to 1, and the value N be set to 0

		Jack	
Mary		1	0
	1		
	0		

		Jack	
Jim		1	0
	1		
	0		

		Jim	
Mary		1	0
	1		
	0		

Dissimilarity between Binary Variables

Jaccard coefficient

$$d(jack, mary) = \frac{0+1}{2+0+1} = 0.33$$

$$d(jack, jim) = \frac{1+1}{1+1+1} = 0.67$$

$$d(jim, mary) = \frac{1+2}{1+1+2} = 0.75$$

Gender is not yet included in the computation

		Jack	
Mary		1	0
	1		
	0		

		Jack	
Jim		1	0
	1		
	0		

		Jim	
Mary		1	0
	1		
	0		

Nominal Attributes

- A generalization of the binary attribute in that it can take more than 2 states, e.g., red, yellow, blue, green
- Method 1: Simple matching
  - m: # of matches, p: total # of variables

$$d(i,j) = \frac{p-m}{p}$$

- Method 2: use a large number of binary attributes
  - creating a new binary variable for each of the M nominal states

Ordinal Attributes

- An ordinal attribute can be discrete or continuous
- order is important, e.g., rank
- Can be treated like interval-scaled
  - replacing x_{if} by their rank $r_{if} \in \{1, \ldots, M_f\}$
  - map the range of each attribute onto [0, 1] by replacing i-th object in the f-th attribute by

$$z_{if} = \frac{r_{if} - 1}{M_f - 1}$$

  - compute the dissimilarity using methods for interval-scaled attributes

Attributes of Mixed Types

- A database may contain different types of attributes
  - symmetric binary, asymmetric binary, nominal, ordinal, and interval.
- How to combine the dissimilarity/distance from data of a mixture of types?

Attributes of Mixed Types

- Use weighted formula to combine their effects.
  - Feature value missing, or asymmetric binary with

    $x_{if} = x_{jf} = 0$ ➔ $\delta_{ij}(f) = 0$
  - Otherwise ➔ $\delta_{ij}(f) = 1$
  - feature is interval-based: use the normalized distance ($\delta_{ij}^{(f)}$ is weight on feature f)

$$d(i,j) = \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)} d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}}$$

  - feature is ordinal
    - compute ranks r_{if} and
    - and treat z_{if} as interval-scaled

$$z_{if} = \frac{r_{if} - 1}{M_f - 1}$$

Practice Question

- Compute the distance between (obj1, obj2),

	Gender	Age	Heart Rate	Fever	Cough	Category
Obj1	F	18	120	N	N	Severe-1
Obj2	M	36	89	N	N	Normal

For Age: m=42, s=3.5,
For heart rate: m=95, s=10
Possible values for Category include : Normal, Severe-1, Severe-2, Dying

For simplicity in demonstration, use Manhattan distance for interval data.

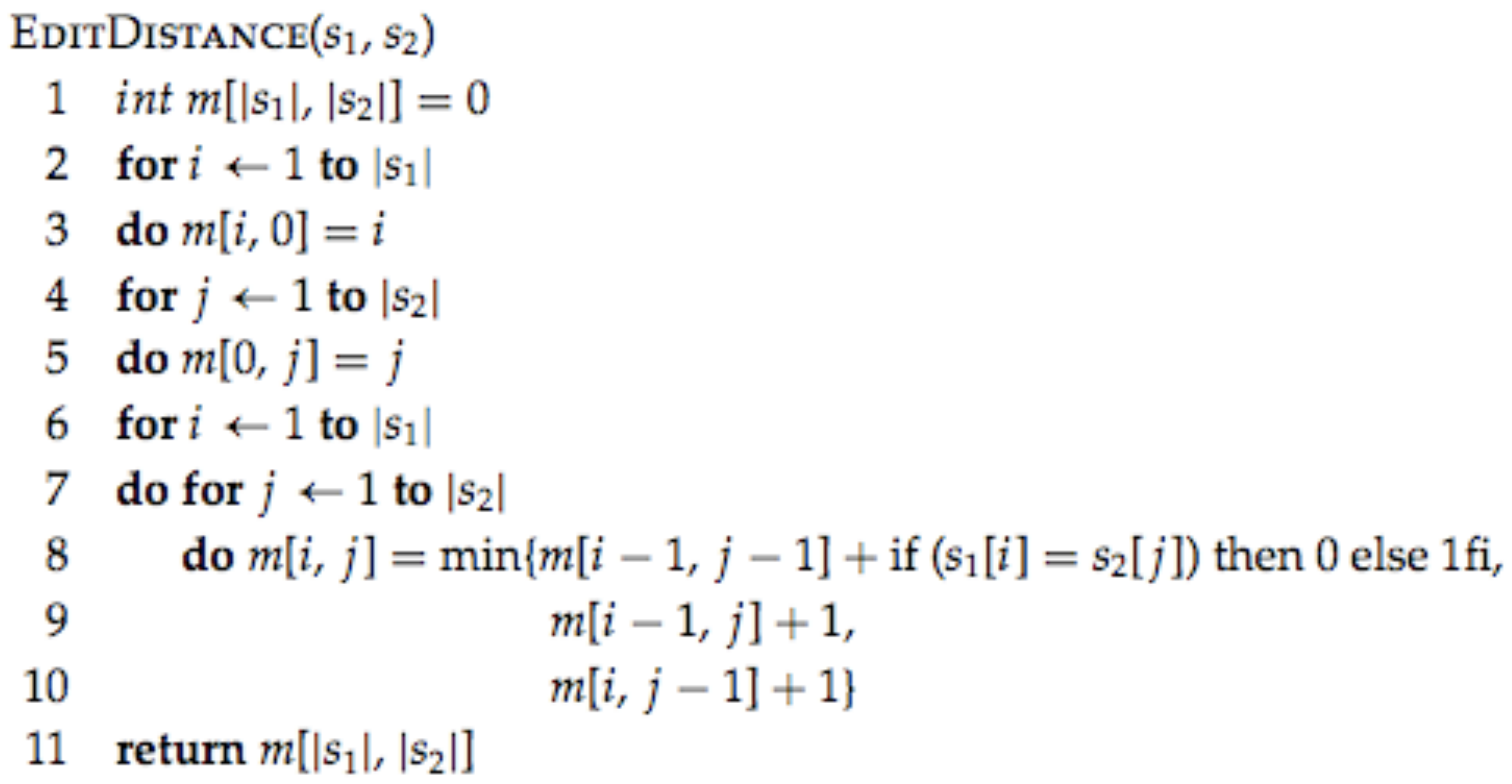
Other Similarity/Distance measures

- Measure distance between words/address/query, or between DNA sequences by edit distance
  - Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
  - Operations are typically character-level
    - Insert, Delete, Replace, (Transposition)
  - E.g., the edit distance from ***dof*** to ***dog*** is 1
    - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
    - from ***cat*** to ***dog*** is 3.
- Generally computed by dynamic programming.

Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming.

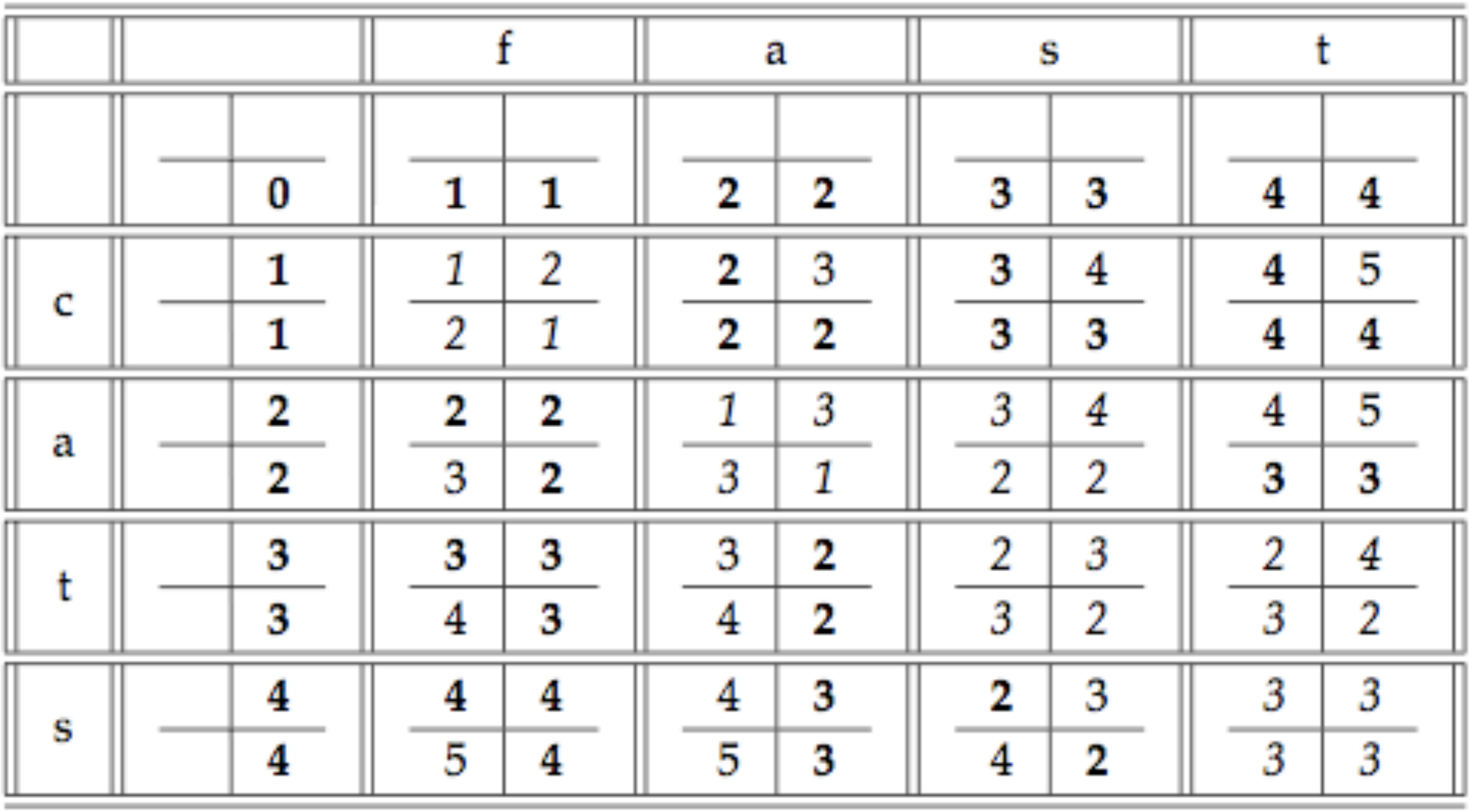
Edit Distance



```
EDITDISTANCE($s_1, s_2$)
    $int\ m[|s_1|, |s_2|] = 0$
    for $i \leftarrow 1$ to $|s_1|$
    do $m[i, 0] = i$
    for $j \leftarrow 1$ to $|s_2|$
    do $m[0, j] = j$
    for $i \leftarrow 1$ to $|s_1|$
    do for $j \leftarrow 1$ to $|s_2|$
        do $m[i, j] = \min\{m[i-1, j-1] +$ if $(s_1[i] = s_2[j])$ then 0 else 1fi,
                          $m[i-1, j] + 1,$
                          $m[i, j-1] + 1\}$
    return $m[|s_1|, |s_2|]$
```

Figure 3.5 Dynamic programming algorithm for computing the edit distance between strings s_1 and s_2.

An Example



			f		a		s		t	
		0	1	1	2	2	3	3	4	4
c		1	1	2	2	3	3	4	4	5
		1	2	1	2	2	3	3	4	4
a		2	2	2	1	3	3	4	4	5
		2	3	2	3	1	2	2	3	3
t		3	3	3	3	2	2	3	2	4
		3	4	3	4	2	3	2	3	2
s		4	4	4	4	3	2	3	3	3
		4	5	4	5	3	4	2	3	3

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Finding nearest neighbors efficiently

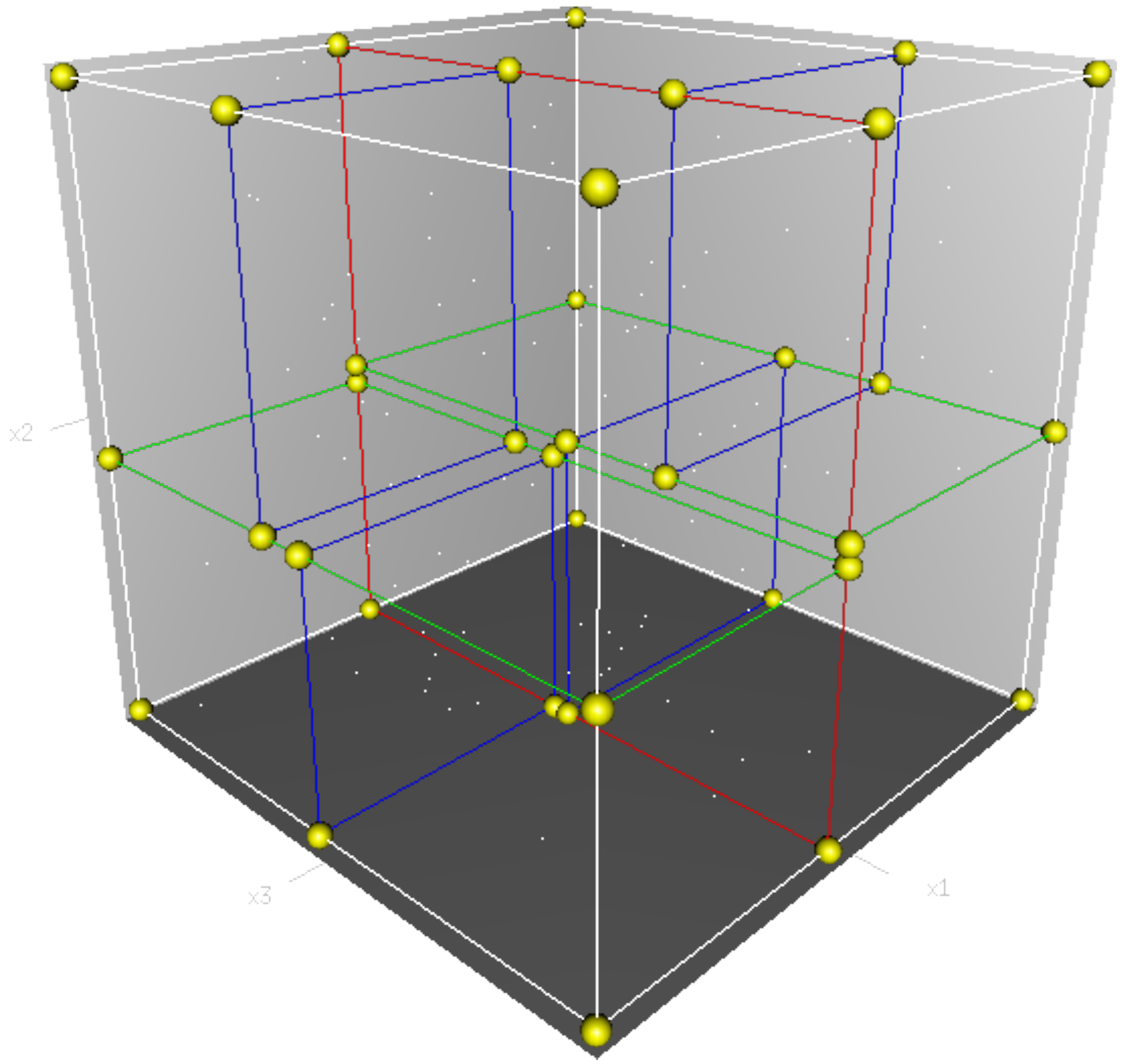
- Simplest way of finding nearest neighbor: linear scan of the data
    - Classification takes time proportional to the product of the number of instances in training and test sets
- Nearest-neighbor search can be done more efficiently using appropriate data structures
- Two methods that represent training data in a tree structure:

kD-trees and *ball trees*

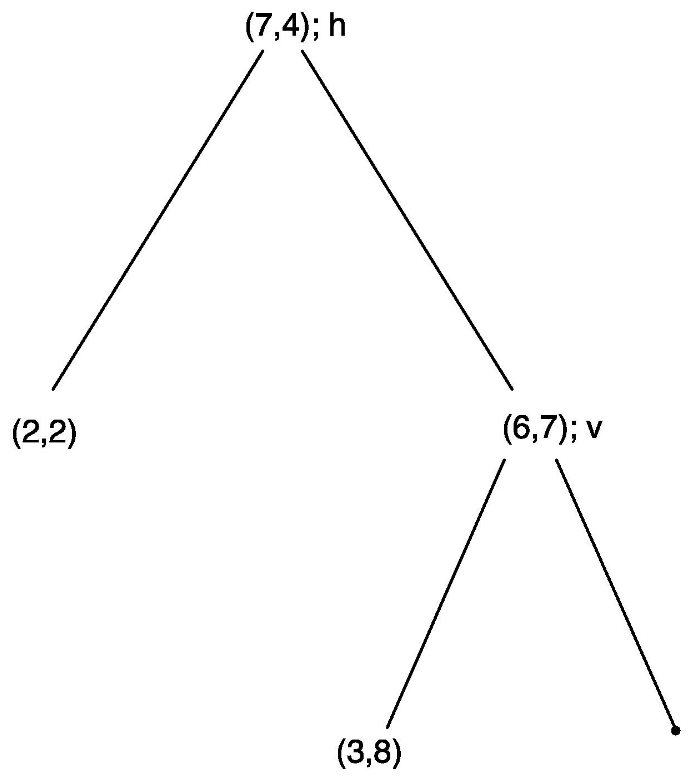
(k-dimensional tree)

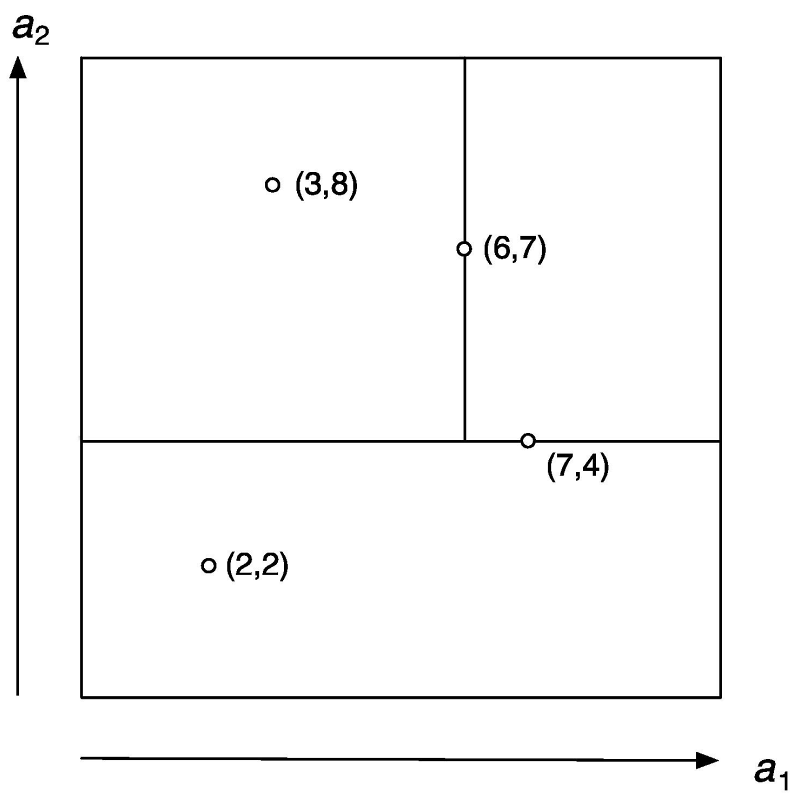
*k*D-tree example

- Binary tree in which every node is a *k*-dimensional point.
- Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces.



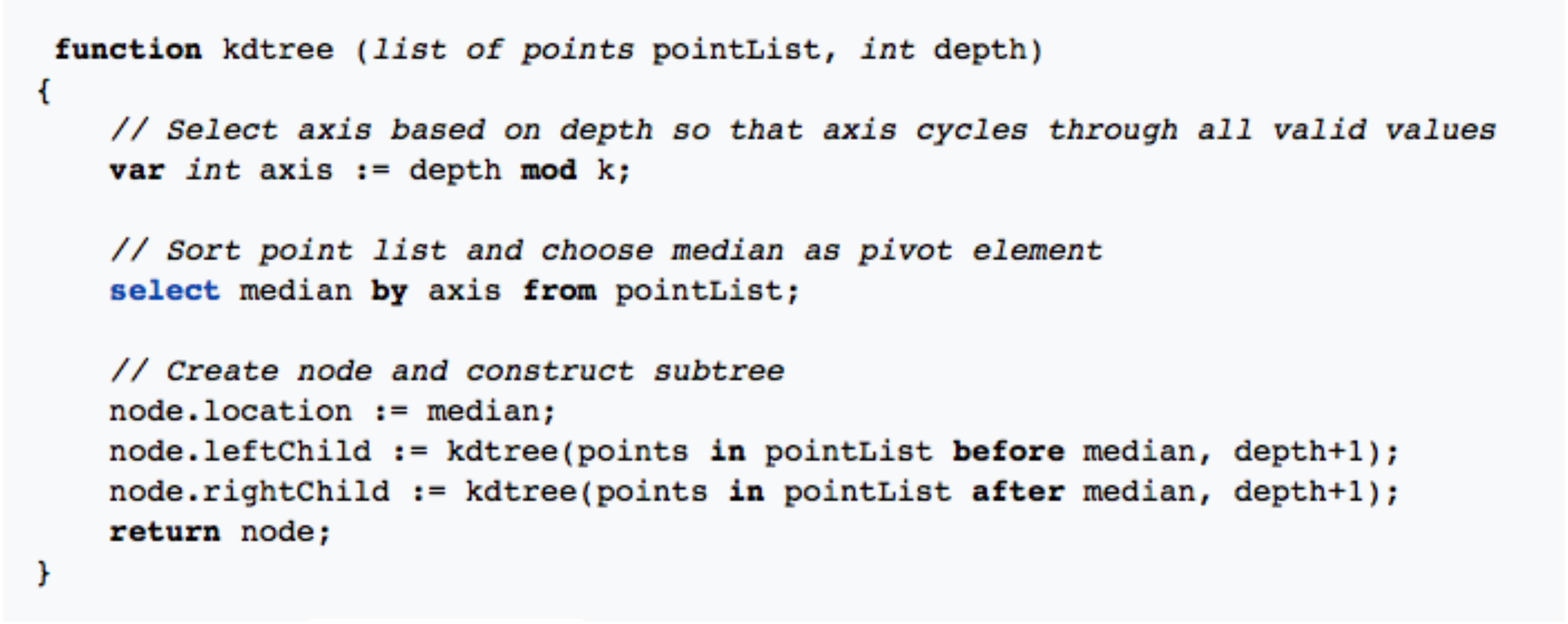
*k*D-tree example



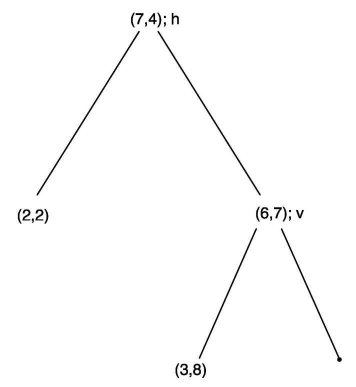


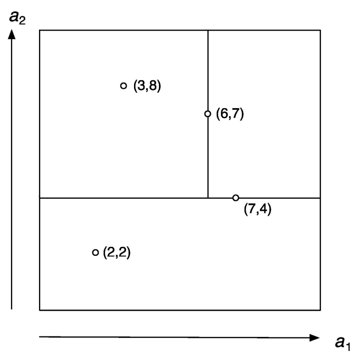
The hyperplane direction is chosen in the following way:
every node in the tree is associated with one of the k-dimensions,
with the hyperplane perpendicular to that dimension's axis.

*k*D-tree example

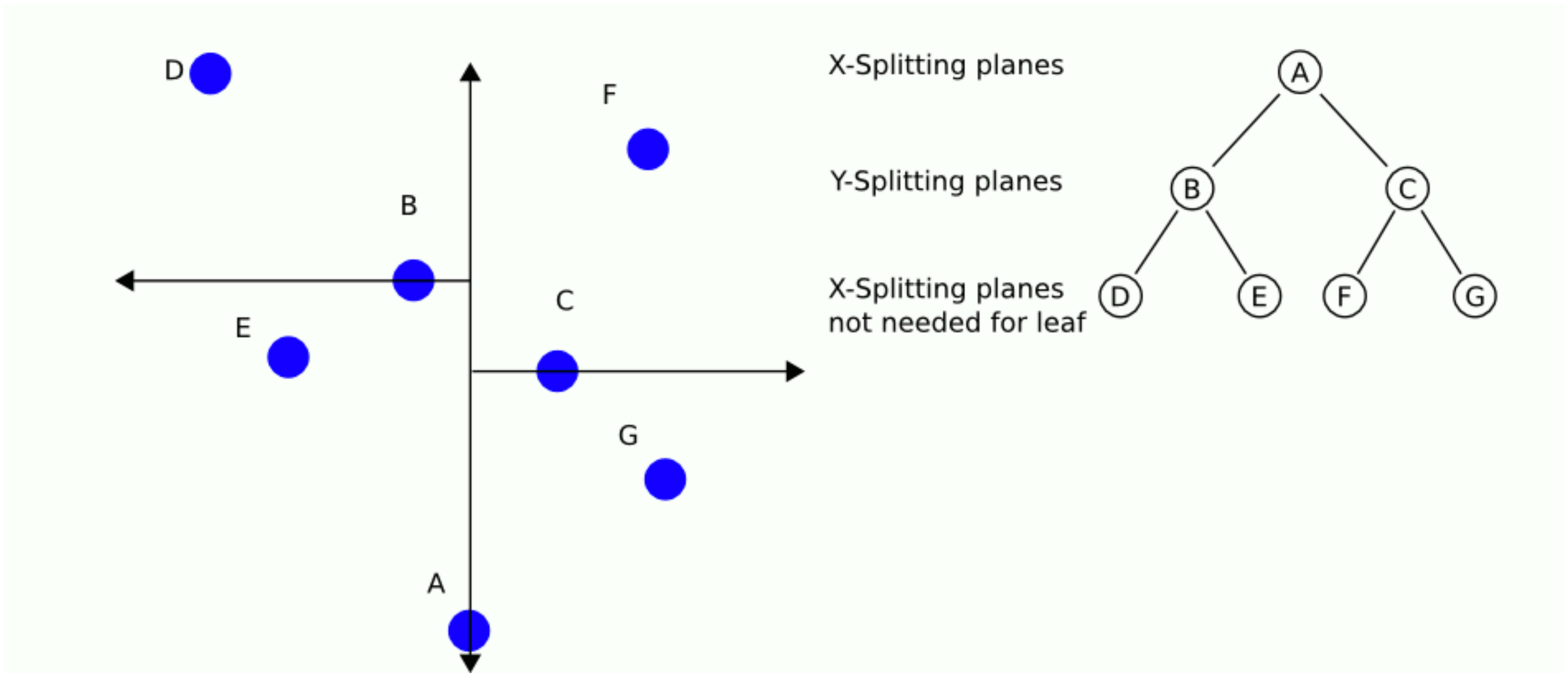


```
 function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtree
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

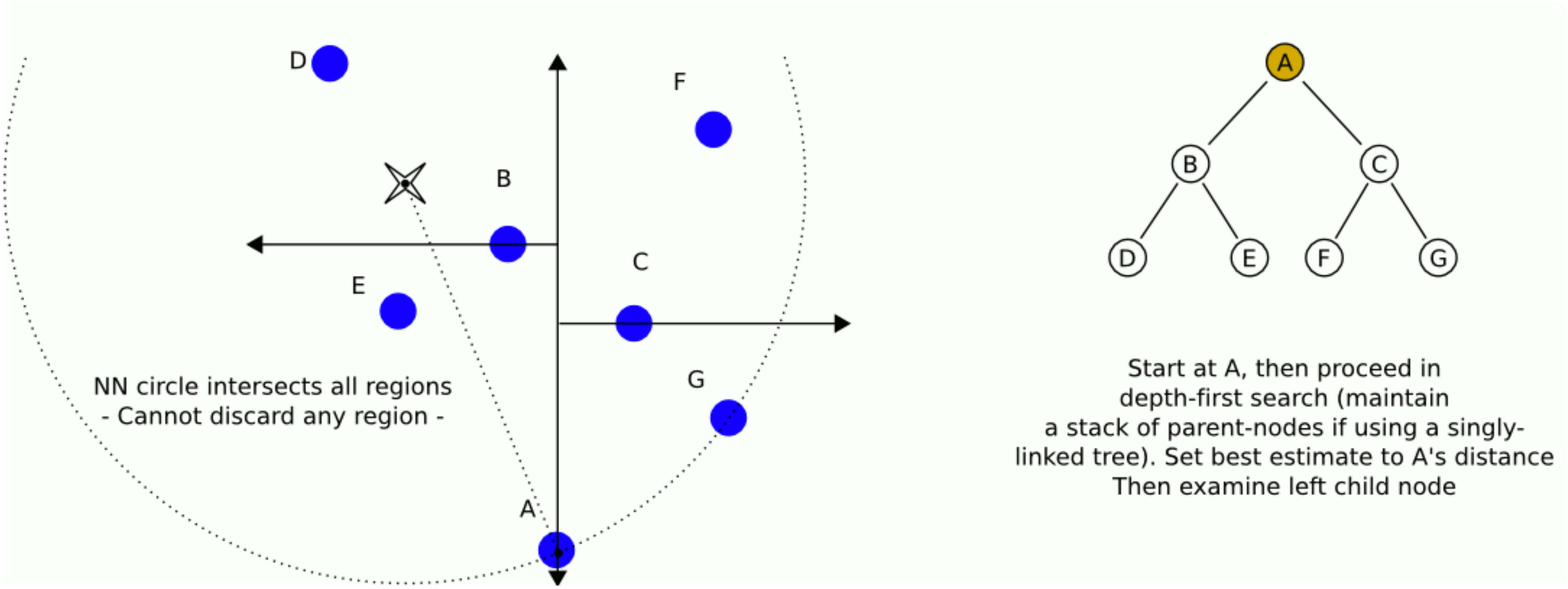




Using kD-trees: example (1)

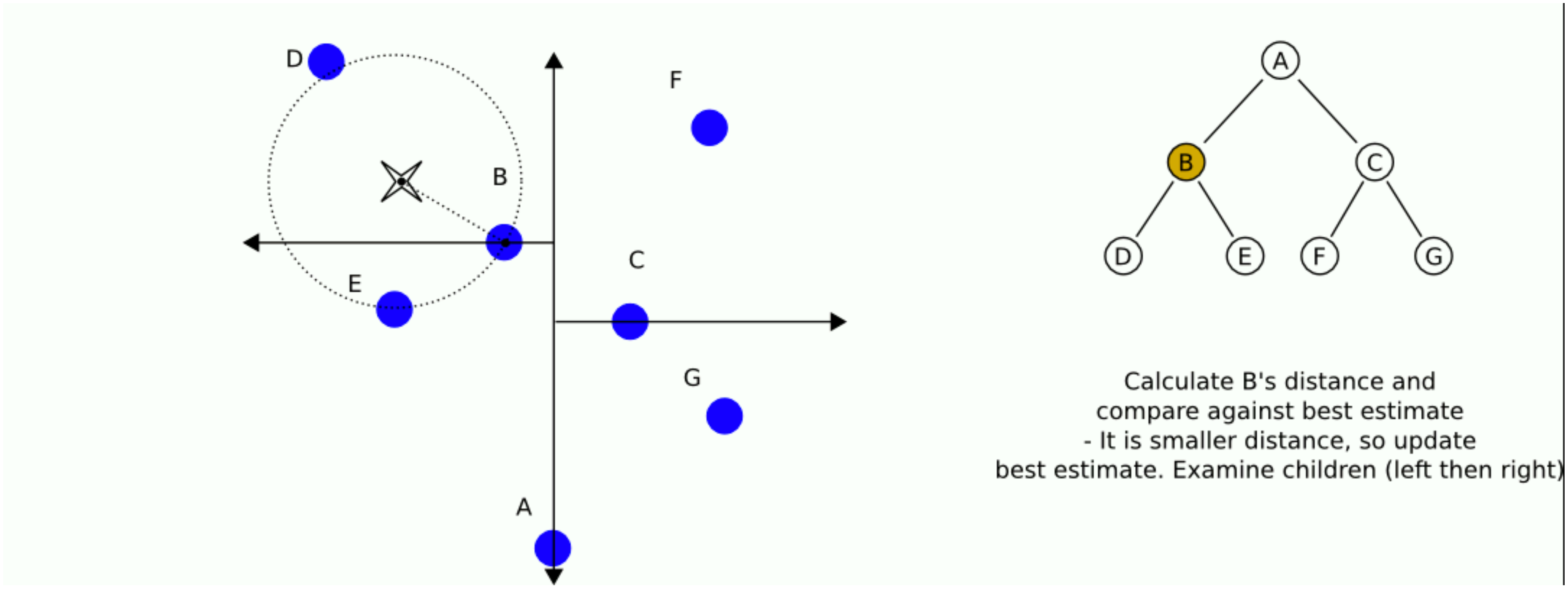


Using kD-trees: example (2)


D
F
B
C
E
G
A
NN circle intersects all regions
- Cannot discard any region -
A
B
C
D
E
F
G

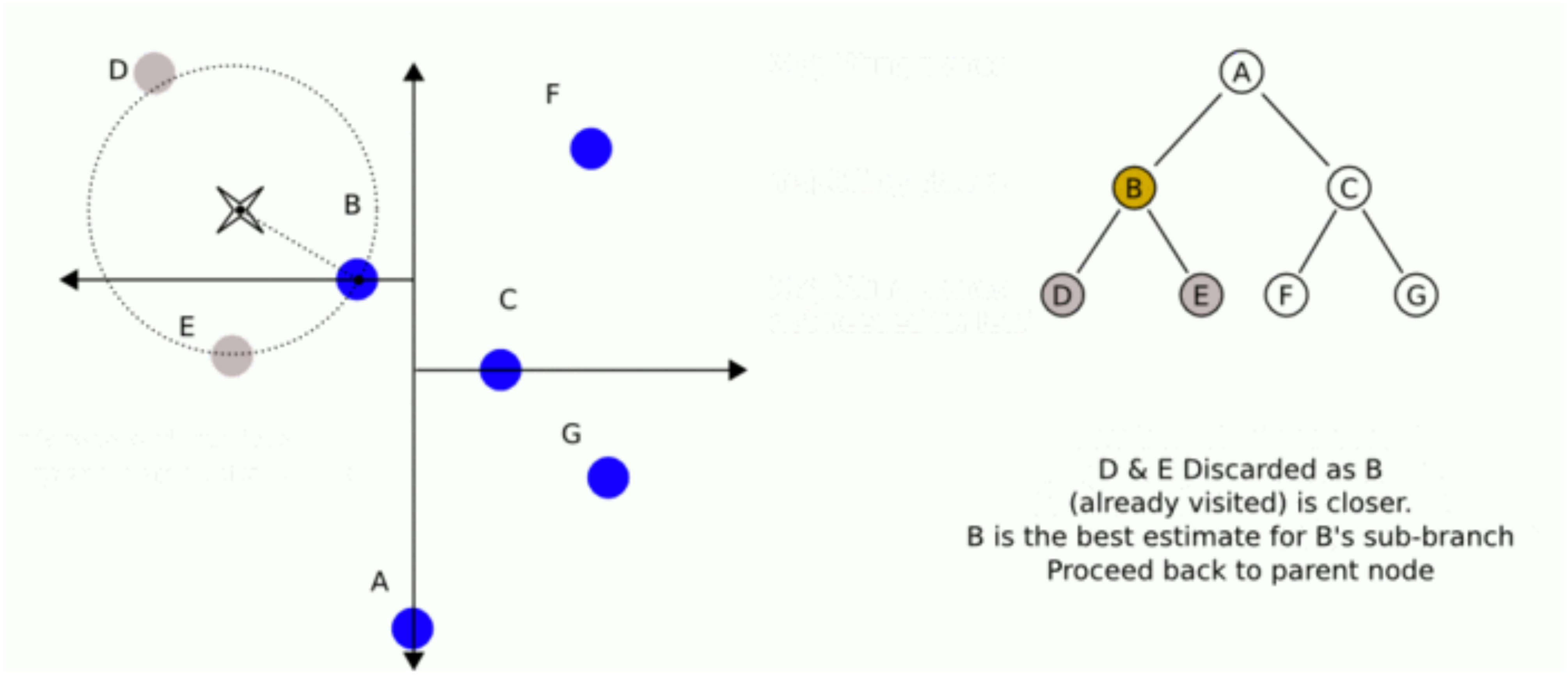
Start at A, then proceed in depth-first search (maintain a stack of parent-nodes if using a singly-linked tree). Set best estimate to A's distance Then examine left child node

Using kD-trees: example (3)



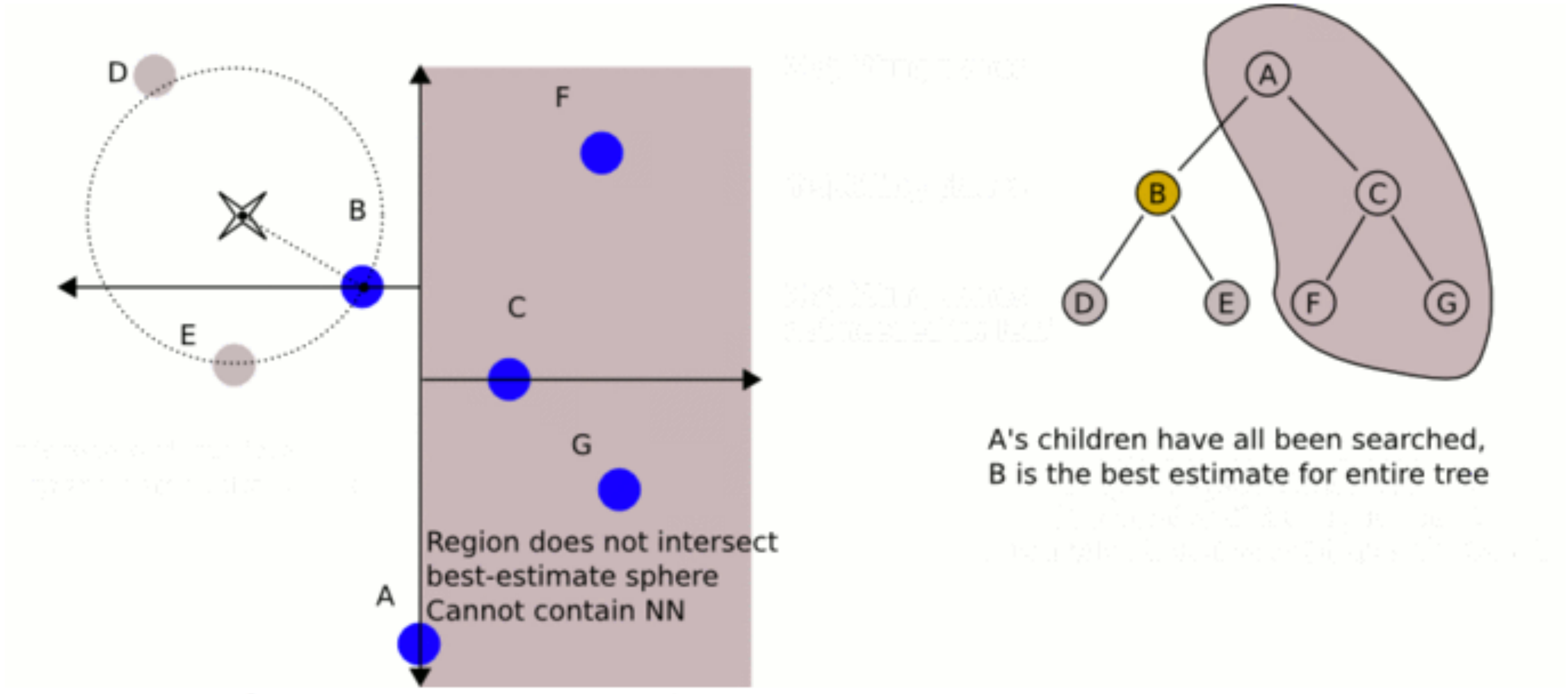
Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

Using kD-trees: example (4)



D & E Discarded as B (already visited) is closer. B is the best estimate for B's sub-branch Proceed back to parent node

Using kD-trees: example (5)


D
F
B
C
E
G
A
Region does not intersect
best-estimate sphere
Cannot contain NN
A
B
C
D
E
F
G
A's children have all been searched,
B is the best estimate for entire tree

More on *k*D-trees

- Complexity depends on depth of tree, given by logarithm of number of nodes
- Amount of backtracking required depends on quality of tree (“square” vs. “skinny” nodes)
- How to build a good tree? Need to find good split point and split direction
    - Split direction: direction with greatest variance
    - Split point: median value along that direction
- Using value closest to mean (rather than median) can be better if data is skewed
- Can apply this recursively

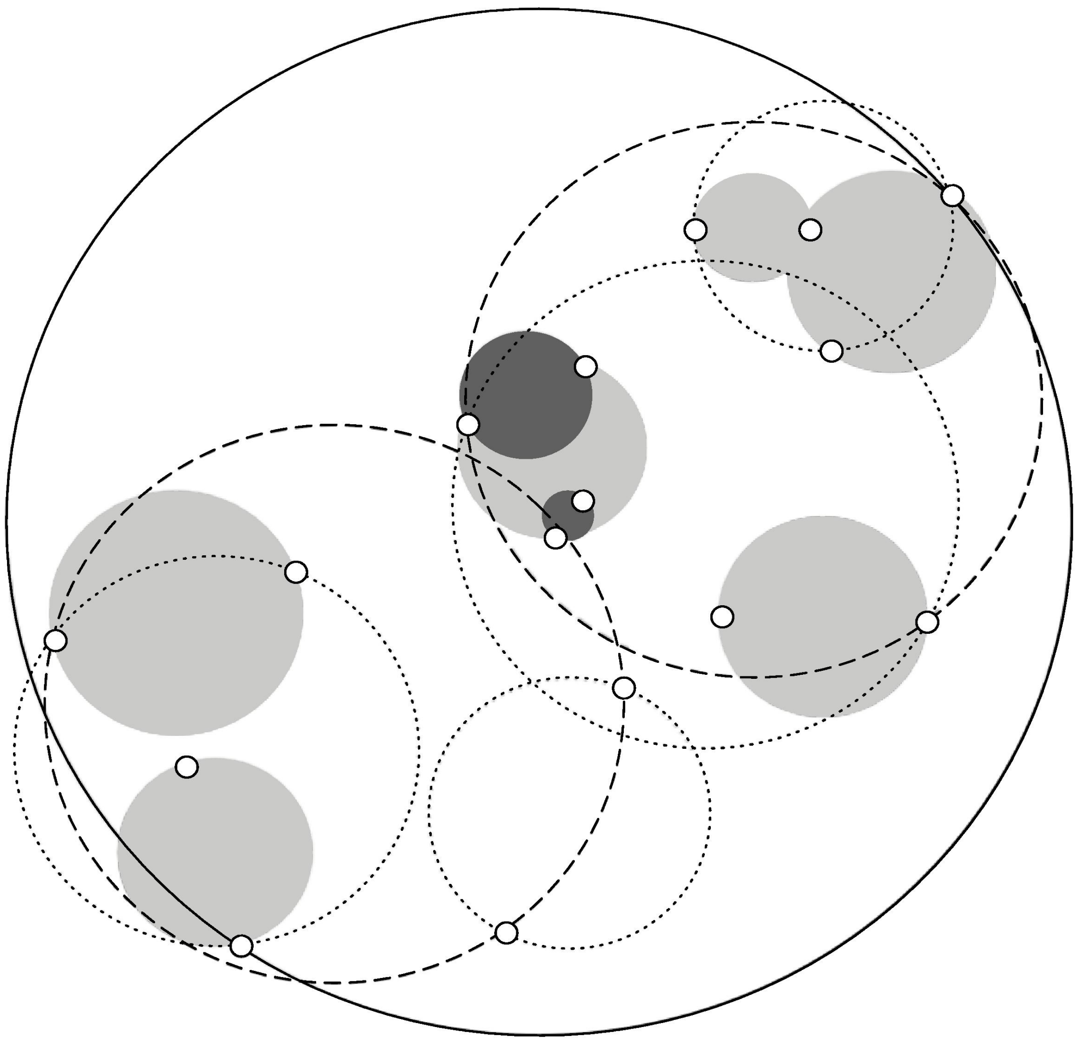
Building trees incrementally

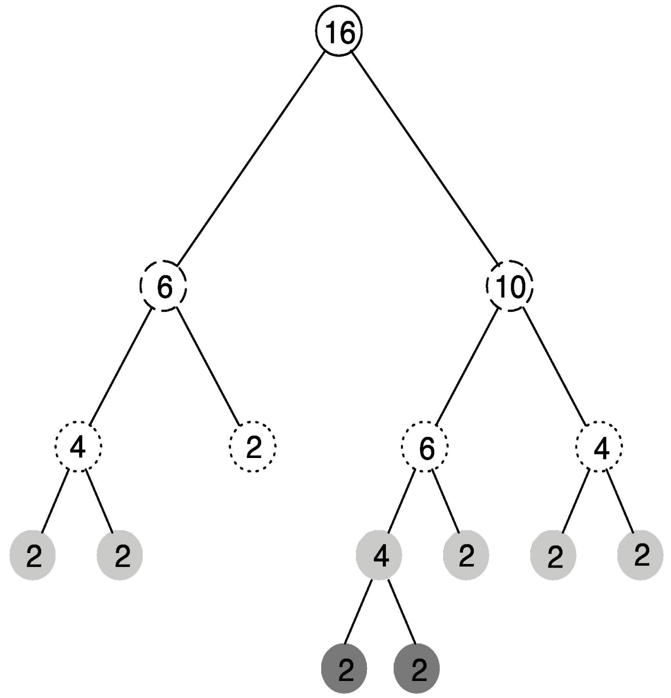
- Big advantage of instance-based learning: classifier can be updated incrementally
    - Just add new training instance!
- Can we do the same with *k*D-trees?
- Heuristic strategy:
    - Find leaf node containing new instance
    - Place instance into leaf if leaf is empty
    - Otherwise, split leaf according to the longest dimension (to preserve squareness)
- Tree should be re-built occasionally (i.e. if depth grows to twice the optimum depth)

Ball trees

- Problem in *k*D-trees: corners
- Observation: no need to make sure that regions don't overlap
- Can use balls (hyperspheres) instead of hyperrectangles
    - A *ball tree* organizes the data into a tree of *k*-dimensional hyperspheres
    - Normally allows for a better fit to the data and thus more efficient search

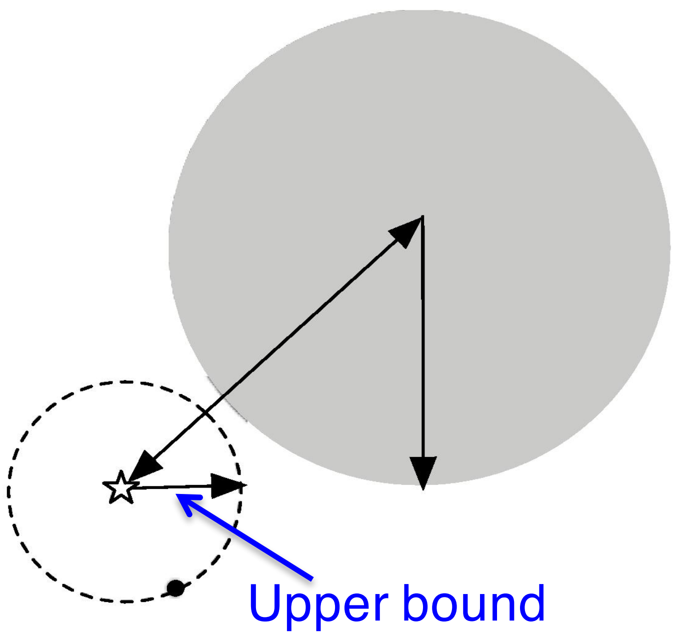
Ball tree example





Using ball trees

- Nearest-neighbor search is done using the same backtracking strategy as in *k*D-trees
- Ball can be ruled out from consideration if: distance from target to ball's center exceeds ball's radius plus current upper bound



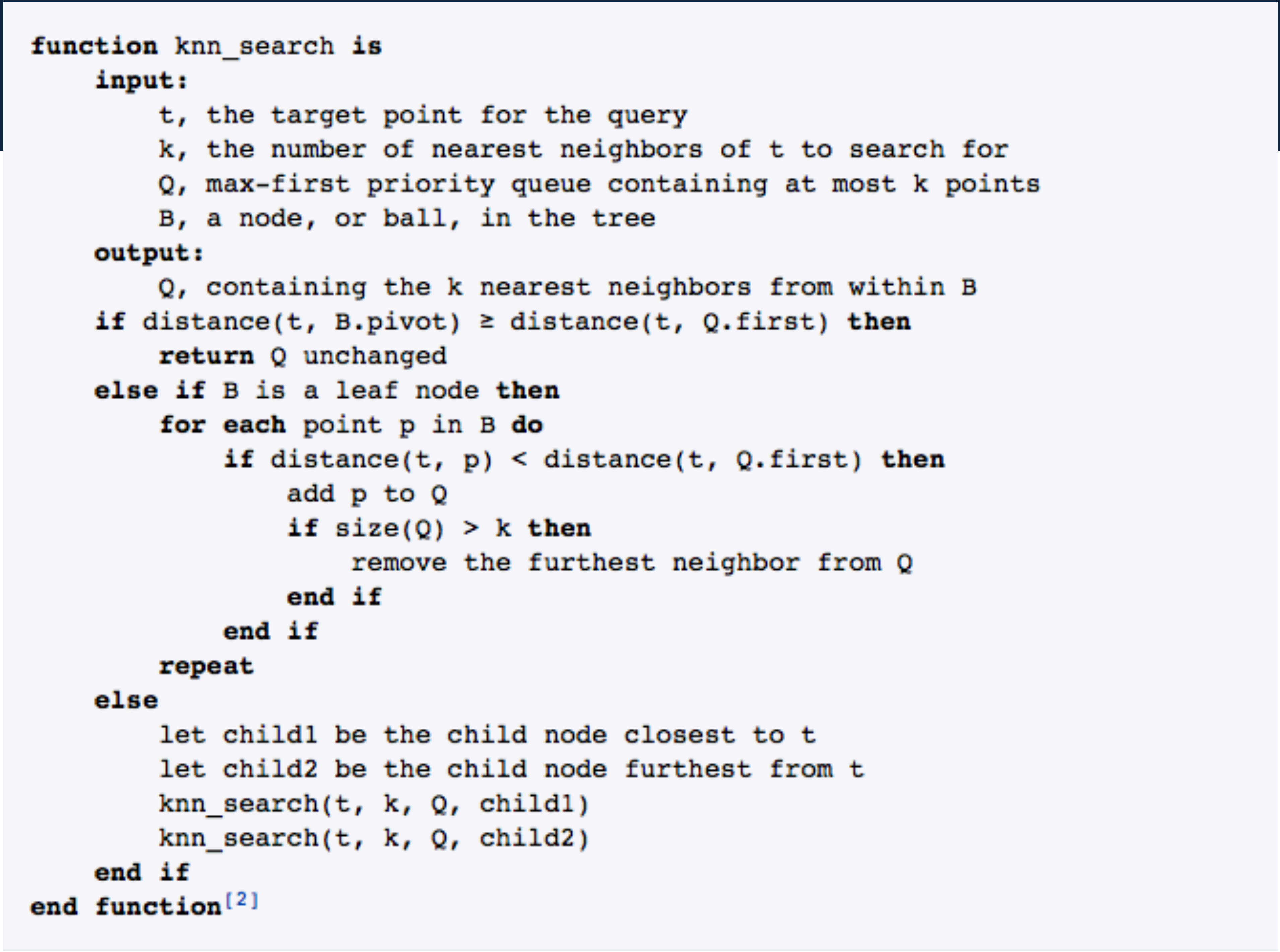
Building ball trees

- Ball trees are built top down (like *k*D-trees)
- Basic problem: splitting a ball into two

```
function construct_balltree is
    input:
        D, an array of data points
    output:
        B, the root of a constructed ball tree
    if a single point remains then
        create a leaf B containing the single point in D
        return B
    else
        let c be the dimension of greatest spread
        let L,R be the sets of points lying to the left and right of the median along dimension c
        create B with two children:
            B.pivot = c
            B.child1 = construct_balltree(L),
            B.child2 = construct_balltree(R)
        return B
    end if
end function
```

Nearest Neighbor with Ball Tree

- At each node *B*, it may perform one of three operations, before finally returning an updated version of the priority queue:
  - If the distance from the test point *t* to the current node *B* is greater than the furthest point in *Q (a maximum first heap)*, ignore *B* and return *Q*.
  - If *B* is a leaf node, scan through every point enumerated in *B* and update the nearest-neighbor queue appropriately. Return the updated queue.
  - If *B* is an internal node, call the algorithm recursively on *B*'s two children, searching the child whose center is closer to *t* first. Return the queue after each of these calls has updated it in turn.



```
function knn_search is
    input:
        t, the target point for the query
        k, the number of nearest neighbors of t to search for
        Q, max-first priority queue containing at most k points
        B, a node, or ball, in the tree
    output:
        Q, containing the k nearest neighbors from within B
    if distance(t, B.pivot) ≥ distance(t, Q.first) then
        return Q unchanged
    else if B is a leaf node then
        for each point p in B do
            if distance(t, p) < distance(t, Q.first) then
                add p to Q
                if size(Q) > k then
                    remove the furthest neighbor from Q
                end if
            end if
        repeat
    else
        let child1 be the child node closest to t
        let child2 be the child node furthest from t
        knn_search(t, k, Q, child1)
        knn_search(t, k, Q, child2)
    end if
end function[2]
```

Discussion of nearest-neighbor learning

- Often very accurate
- Assumes all attributes are equally important
  - Remedy: attribute selection or weights
- Possible remedies against noisy instances:
  - Take a majority vote over the *k* nearest neighbors
  - Removing noisy instances from dataset (difficult!)
- Statisticians have used *k*-NN since early 1950s
  - If $n \rightarrow \infty$ and $k/n \rightarrow 0$, error approaches minimum
- *k*D-trees become inefficient when number of attributes is too large (approximately > 10)
- Ball trees (which are instances of *metric trees*) work well in higher-dimensional spaces