



3D Point Clouds

Lecture 2

Nearest Neighbors

主讲人 黎嘉信

Aptiv 自动驾驶
新加坡国立大学 博士
清华大学 本科





1. Binary Search Tree



2. Kd-tree



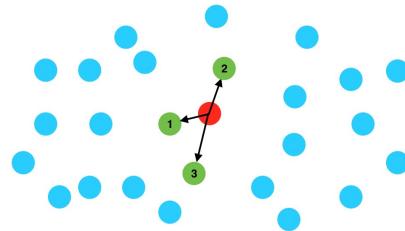
3. Octree



Nearest Neighbor (NN) Problem

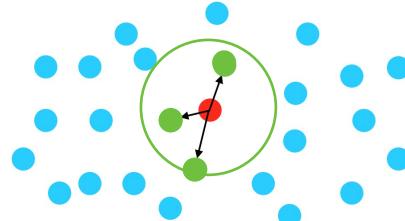
K-NN

- Given a set of points S in a space M , a query point $q \in M$, find the k closest points in S



Fixed Radius-NN

- Given a set of points S in a space M , a query point $q \in M$, find all the points in S , s.t.,
 $\|s - q\| < r$





Why NN problem is important?



It is almost everywhere

- What we have covered:
 - Surface normal estimation
 - Noise filtering
 - Sampling
- What we will cover:
 - Clustering
 - Deep learning
 - Feature detection / description
 -



Why don't we simply call an open-source library (flann, PCL, etc.)?

- They are not efficient enough.
- They are general lib, not optimized for 2D/3D.
- Most open-source octree implementation is in-efficient, while octree is most effective for 3D.
- Few GPU based NN library is available



Why NN is difficult for point clouds

-  For Images, a neighbor is simply $x + \Delta x, y + \Delta y$
-  For point clouds
 - Irregular – no grid based representation
 - Curse of dimensionality
 - Non-trivial to build grids
 - Non-trivial to sort or build spatial partitions
 - Huge data throughput in real-time applications
 - Velodyne HDL-64E – 2.2 million points per second / 110,000 points at 20Hz
 - Brute-force self-NN search is $110,000 \times 110,000 \times 0.5 = 6 \times 10^9$ comparisons @ 20Hz



Lecture Outline



Binary Search Tree (BST)

- Basic knowledge about trees
- 1D NN problem
- With Python codes



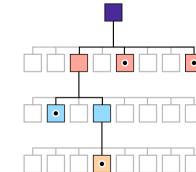
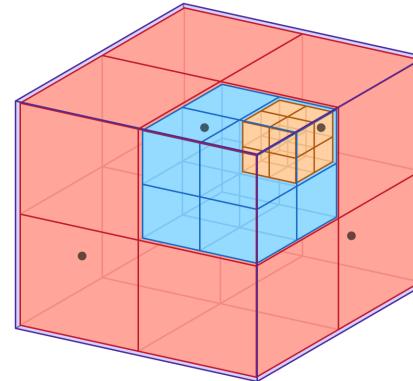
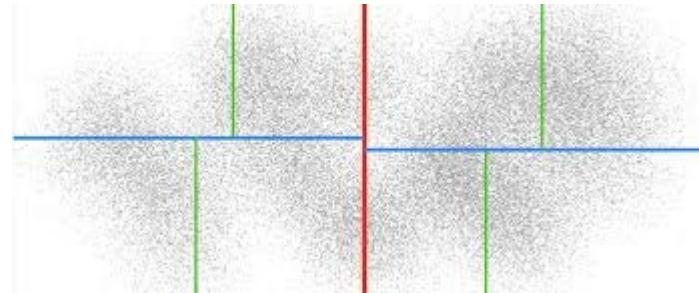
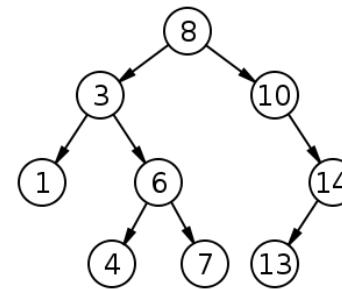
Kd-tree

- Works for data of any dimension
- Illustrated in 2D
- With Python codes



Octree

- Specifically designed for 3D data
- Illustrated in 2D/3D
- With Python codes





Core Ideas Shared by BST, kd-tree, octree

❶ NN by space partition

- Split the space into different areas,
- Search some areas only, instead of all the data points

❷ Stopping criteria

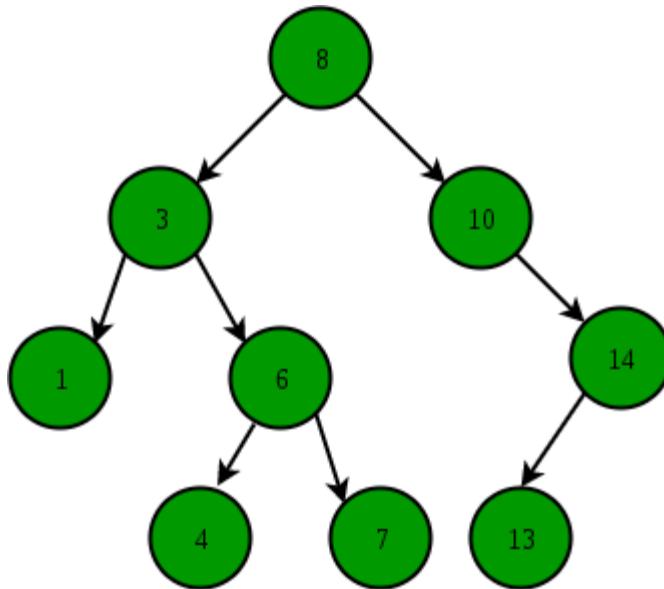
- How to skip some partitions?
 - Intersection of the “worst distance” with the partition boundaries
- How to stop the k-NN / Radius-NN search?
 - Search until the root
 - A partition completely contains the “worst distance”



Binary Search Tree (BST)

BST is a node-based tree data structure:

1. A node' s left subtree contains nodes with keys lesser than its key
2. A node' s right subtree contains nodes with keys larger than its key
3. The left / right subtree is BST





Binary Search Tree (BST)

From Wikipedia

Binary search tree

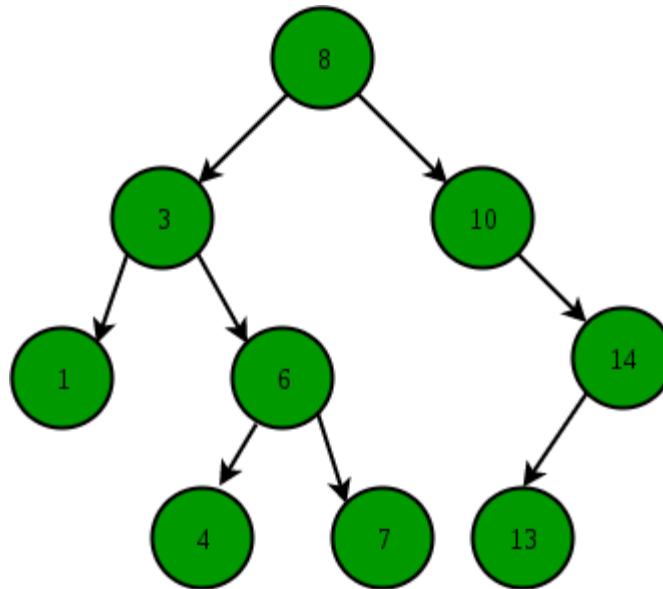
Type tree

Invented 1960

Invented P.F. Windley, A.D. Booth, A.J.T.
by Colin, and T.N. Hibbard

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$



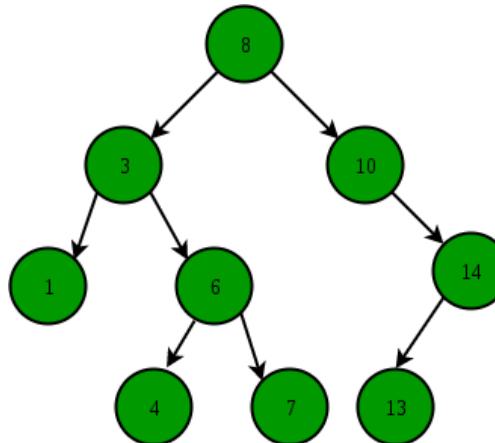


BST – Node definition

- ➊ A node contains
 - 1. Key
 - 2. Left child
 - 3. Right child
 - 4.

- ➋ The left/right child is a Node as well

```
class Node:  
    def __init__(self, key, value=-1):  
        self.left = None  
        self.right = None  
        self.key = key  
        self.value = value
```





BST – Construction / Insertion

- ➊ Given N 1D-points (scalar) denoted by an array
$$\{x_1, x_2, \dots, x_n\}, x_i \in \mathbb{R}$$

- ➋ Construct a BST that stores the points and its index in the array, e.g.
[100, 20, 500, 10, 30, 40]

Data generation

```
db_size = 10  
data = np.random.permutation(db_size).tolist()
```

Recursively insert each an element

```
def insert(root, key, value=-1):  
    if root is None:  
        root = Node(key, value)  
    else:  
        if key < root.key:  
            root.left = insert(root.left, key, value)  
        elif key > root.key:  
            root.right = insert(root.right, key, value)  
        else: # don't insert if key already exist in the tree  
            pass  
    return root
```

Insert each element

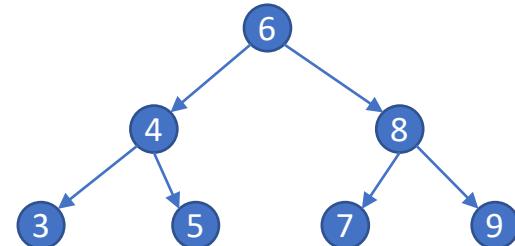
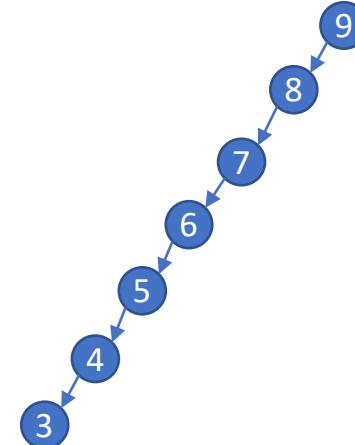
“value” in the Node is the index of a point in the array
Useful in later NN search

```
root = None  
for i, point in enumerate(data):  
    root = insert(root, point, i)
```



BST – Insertion Complexity

- ❖ The worst case is $O(h)$, where h is the height of the BST
- ❖ In the worst case, h is the number of points in BST.
 - Unbalanced tree – a chain in an extreme case
 - E.g., inserting [9, 8, 7, 6, 5, 4, 3] into an empty BST
- ❖ Tree balancing is another topic
 - Sort the array and insert as a balanced tree (select median as root)
 - AVL tree
 - Red-Black tree
 - etc.
- ❖ Best case $h = \log_2 n$





BST – Search

- Given a BST, and a query (key), determine *which node* equals to that query (key), if not, return *NULL*
- Can be done *recursively* or *iteratively*



BST – Search Recursively

Search till the leaf but not found.

```
def search_recursive(root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return search_recursive(root.left, key)
    elif key > root.key:
        return search_recursive(root.right, key)
```

Find a match

For sure just need to look at the left

For sure just need to look at the right



BST – Search Iteratively

- ❖ Use “current_node” to simulate a *Stack*, so that recursion is avoided
- ❖ In any case, you can write your own *Stack* to avoid recursion, but that may be complicated sometimes.

```
def search_iterative(root, key):  
    current_node = root  
    while current_node is not None:  
        if current_node.key == key:  
            return current_node  
        elif key < current_node.key:  
            current_node = current_node.left  
        elif key > current_node.key:  
            current_node = current_node.right  
    return current_node
```

- ❖ Search recursively or iteratively complexity same as insertion worst O(h)



BST – Search

Recursion

Pros:

- Easy to understand / implement
- Codes are short

Cons:

- Hard to trace step-by-step
- $O(n)$ storage, n is number of recursion (May be optimized by compiler)

Iteration

Pros:

- Avoid **stack-overflow**, e.g., in embedded system / GPU
- Easier in step-by-step tracing
- $O(1)$ storage

Cons:

- The logic is complicated



BST – Depth First Traversal

Hexagon icon Inorder – Left, Root, Right

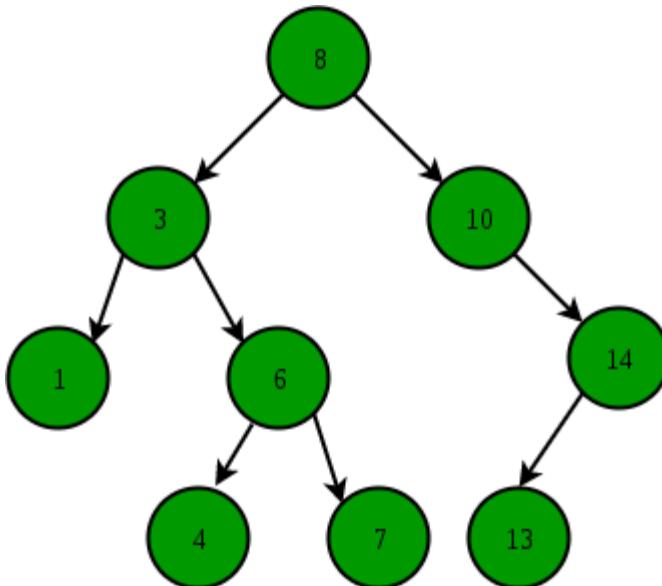
- E.g., sorting
- 1, 3, 4, 6, 7, 8, 10, 13, 14

Hexagon icon Preorder – Root, Left, Right

- E.g., copy a tree
- 8, 3, 1, 6, 4, 7, 10, 14, 13

Hexagon icon Postorder – Left, Right, Root

- E.g., delete a node
- 1, 4, 7, 6, 3, 13, 14, 10, 8



```
def inorder(root):
    # Inorder (Left, Root, Right)
    if root is not None:
        inorder(root.left)
        print(root)
        inorder(root.right)

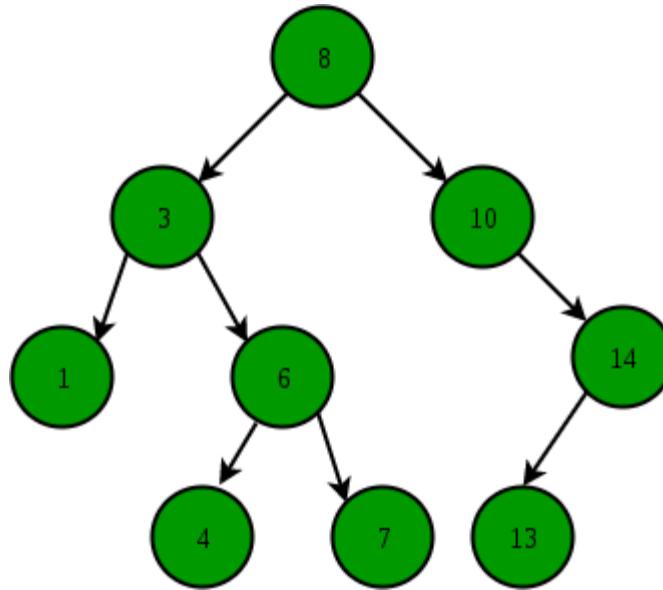
    1, 3, 4, 6, 7, 8, 10, 13, 14

def preorder(root):
    # Preorder (Root, Left, Right)
    if root is not None:
        print(root)
        preorder(root.left)
        preorder(root.right)

    8, 3, 1, 6, 4, 7, 10, 14, 13

def postorder(root):
    # Postorder (Left, Right, Root)
    if root is not None:
        postorder(root.left)
        postorder(root.right)
        print(root)

    1, 4, 7, 6, 3, 13, 14, 10, 8
```

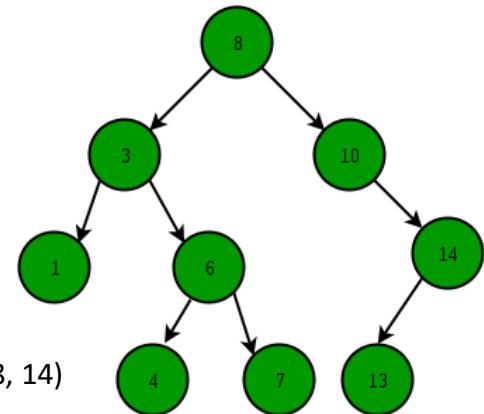




BST – 1NN Search

- Query point – 11

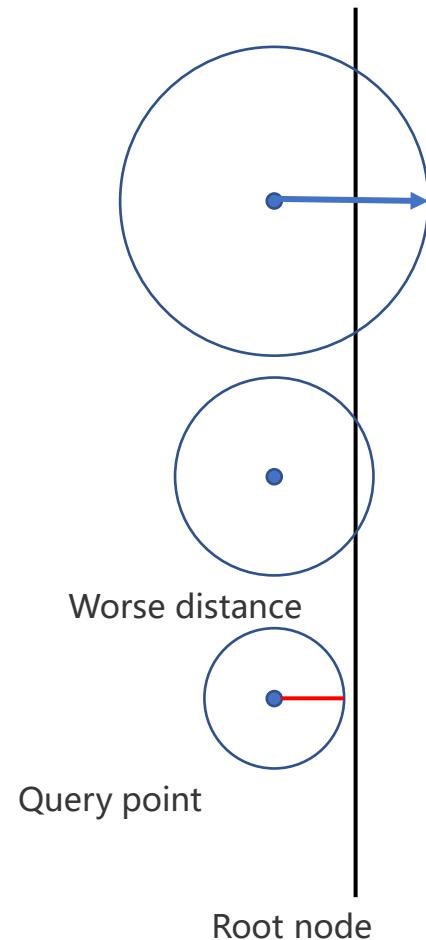
1. 8
 - a) **worst distance = 3** (11-8)
 - b) any point in 8's left tree is at least 3 away from query
 - c) do I need to go further? Yes! Right subtree is (8, +inf] but **worst distance=3** -> need to search (8, 14)
2. 10
 - a) **worst distance = 1** (11-10)
 - b) do I need to go further? Yes! Right subtree is (10, +inf] but **worst distance=1** -> need to search (10, 12)
3. 14
 - a) **worst distance = 1**
 - b) do I need to further? Yes! Left subtree is (10, 14) but **worst distance=1** -> need to search (10, 12)
4. Go back to 14
 - c) do I need to go right? No! Right subtree is (14, inf] but **worst distance=1** -> I just need to search (10, 12)
5. Go back to 10
 - c) do I need to go left? No! Left subtree is (8, 10) but **worst distance=1** -> I just need to search (10, 12)
6. Go back to 8
 - d) do I need to go left? No! Left subtree is [-inf, 8) but **worst distance=1** -> just need to search (10, 12)





BST – kNN Search

- Almost same as 1NN search
- Difference is how to compute **worst distance**
- Worst distance** is the largest distance that you should search around the query point
- Areas outside this “worst circle” can be skipped
- In kNN search, the worst distance is dynamic





BST – Worst Distance for kNN

- Build a container to store the kNN results
 - Example:
 - Existing container content
 - [1, 2, 3, 4, inf, inf]
- k results are sorted
 - **add_point(3.5)**
 - *Step 1.* Make space for 3.5
 - [1, 2, 3, **4, 4**, inf]
- worst_dist is the last one
 - *Step 2.* Put 3.5 in the correct position
 - [1, 2, 3, **3.5**, 4, inf]
- Add a result if
$$dist < worse_dist$$
 - *Step 3.* Update worst_dist

```
class KNNResultSet:  
    def __init__(self, capacity):  
        self.capacity = capacity  
        self.count = 0  
        self.worst_dist = 1e10  
        self.dist_index_list = []  
        for i in range(capacity):  
            self.dist_index_list.append(DistIndex(self.worst_dist, 0))  
  
    self.comparison_counter = 0  
  
    class DistIndex:  
        def __init__(self, distance, index):  
            self.distance = distance  
            self.index = index  
  
        def __lt__(self, other):  
            return self.distance < other.distance  
  
    def size(self):  
        return self.count  
  
    def full(self):  
        return self.count == self.capacity  
  
    def worstDist(self):  
        return self.worst_dist  
  
    def add_point(self, dist, index):  
        self.comparison_counter += 1  
        if dist > self.worst_dist:  
            return  
  
        if self.count < self.capacity:  
            self.count += 1  
  
        i = self.count - 1  
        while i > 0:  
            if self.dist_index_list[i-1].distance > dist:  
                self.dist_index_list[i] = copy.deepcopy(self.dist_index_list[i-1])  
                i -= 1  
            else:  
                break  
  
        self.dist_index_list[i].distance = dist  
        self.dist_index_list[i].index = index  
        self.worst_dist = self.dist_index_list[self.capacity-1].distance
```

Initialized to large value

Container to keep all the k neighbors

If a point is added, put it in a ordered position

```
def knn_search(root: Node, result_set: KNNResultSet, key):
    if root is None:
        return False

    # compare the root itself
    result_set.add_point(math.fabs(root.key - key), root.value)
    if result_set.worstDist() == 0: A special case – if the worst distance is 0, no
        need to search anymore

    if root.key >= key:
        # iterate left branch first If key != query, need to go through one subtree
        if knn_search(root.left, result_set, key):
            return True      May not need to search for the other subtree, depends on worst distance
        elif math.fabs(root.key-key) < result_set.worstDist():
            return knn_search(root.right, result_set, key)
        return False

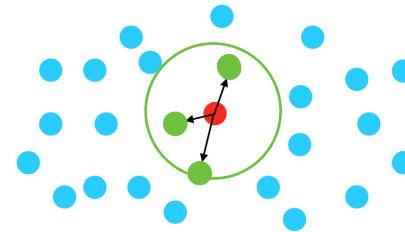
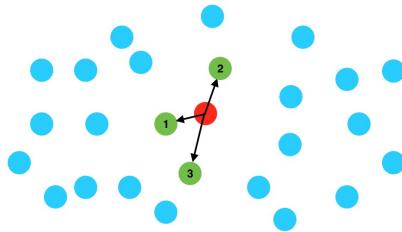
    else: # iterate right branch first
        if knn_search(root.right, result_set, key):
            return True
        elif math.fabs(root.key-key) < result_set.worstDist():
            return knn_search(root.left, result_set, key)
        return False
```

Similar to the “if” block above



Radius NN Search

- Same as kNN, in the sense that,
 - Worst distance circle intersects the boundary -> search
 - If not -> skip
- In implementation, we don't need to change the BST kNN search logics, except that, the worst distance is **fixed**, instead of dynamic





BST – Radius Result Set Manager

- ➊ Simpler than kNN result set manager.
- ➋ Worst distance is fixed.
- ➌ No need to maintain a sorted result set.

```
|     def add_point(self, dist, index):  
|         self.comparison_counter += 1  
|         if dist > self.radius:  
|             return  
  
|             self.count += 1  
|             self.dist_index_list.append(DistIndex(dist, index))
```



BST - kNN v.s. Radius NN

```
def knn_search(root: Node, result_set: KNNResultSet, key):
    if root is None:
        return False

    # compare the root itself
    result_set.add_point(math.fabs(root.key - key), root.value)
    if result_set.worstDist() == 0:
        return True | This part is gone in
    | radius search, because
    | worst_dist = r

    if root.key >= key:
        # iterate left branch first
        if knn_search(root.left, result_set, key):
            return True
        elif math.fabs(root.key-key) < result_set.worstDist():
            return knn_search(root.right, result_set, key)
        return False
    else:
        # iterate right branch first
        if knn_search(root.right, result_set, key):
            return True
        elif math.fabs(root.key-key) < result_set.worstDist():
            return knn_search(root.left, result_set, key)
        return False
```

```
def radius_search(root: Node, result_set: RadiusNNResultSet, key):
    if root is None:
        return False

    # compare the root itself
    result_set.add_point(math.fabs(root.key - key), root.value)

    if root.key >= key:
        # iterate left branch first
        if radius_search(root.left, result_set, key):
            return True
        elif math.fabs(root.key-key) < result_set.worstDist():
            return radius_search(root.right, result_set, key)
        return False
    else:
        # iterate right branch first
        if radius_search(root.right, result_set, key):
            return True
        elif math.fabs(root.key-key) < result_set.worstDist():
            return radius_search(root.left, result_set, key)
        return False
```



A complete script

```
db_size = 100
k = 5
radius = 2.0

data = np.random.permutation(db_size).tolist()

root = None
for i, point in enumerate(data):
    root = insert(root, point, i)

query_key = 6
result_set = KNNResultSet(capacity=k)
knn_search(root, result_set, query_key)
print('kNN Search:')
print('index - distance')
print(result_set)

result_set = RadiusNNResultSet(radius=radius)
radius_search(root, result_set, query_key)
print('Radius NN Search:')
print('index - distance')
print(result_set)
```

- Search in 100 points, takes 7 comparison only
- Complexity is around $O(\log_2(n))$, n is number of database points, if tree is balanced
- Worst $O(N)$

kNN Search:

index - distance
73 - 0.00
5 - 1.00
12 - 1.00
1 - 2.00
98 - 2.00

In total 7 comparison operations.

Radius NN Search:

index - distance
73 - 0.00
5 - 1.00
12 - 1.00
1 - 2.00
98 - 2.00

In total 5 neighbors within 2.000000.
There are 7 comparison operations.

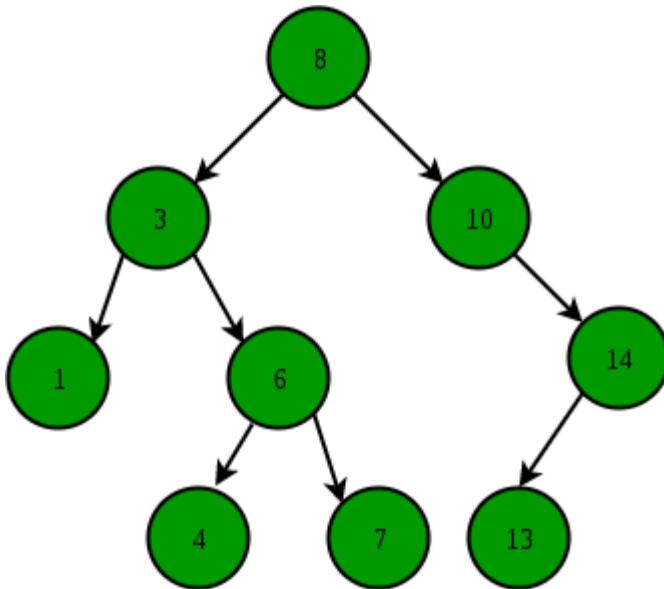


Binary Search Tree (BST)

BST based 1D kNN/RadiusNN search

- Naïve BST is for 1D data only

Tree based kNN/RadiusNN can be viewed as a Branch-n-Bound algorithm.

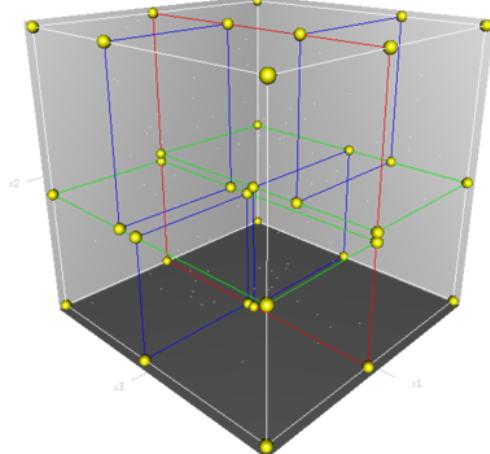




Kd-tree (k-dimensional tree)

- ❖ It is an extension of BST into high dimension
 - BST is 1-dimensional, how to extend?
 - BST in each dimension!
- ❖ Invented by Jon Louis Bentley in 1975
- ❖ The kd-tree is a binary tree where every leaf node is a **k-dimensional point**

k-d tree		
Type	Multidimensional BST	
Invented	1975	
Invented by	Jon Louis Bentley	
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$



A 3-dimensional kd tree:
1. Red
2. Green
3. Blue



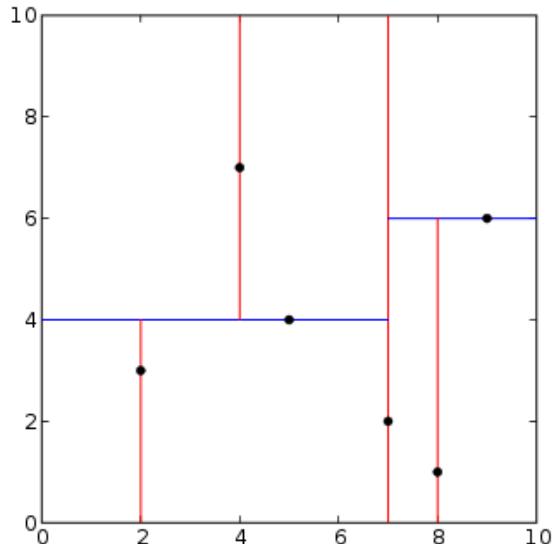
Kd-tree Construction

- ➊ If there is only one point, or number of points < leaf_size, build a leaf
- ➋ Otherwise, divide the points in half by a hyperplane perpendicular to the selected splitting axis
- ➌ Recursively repeat the first two steps.

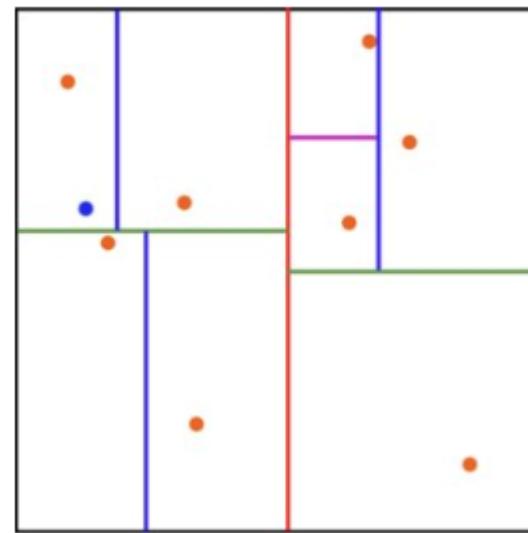


Kd-tree Construction – Two Conventions

Splitting position is one
of the points



Splitting position is **NOT**
one of the points

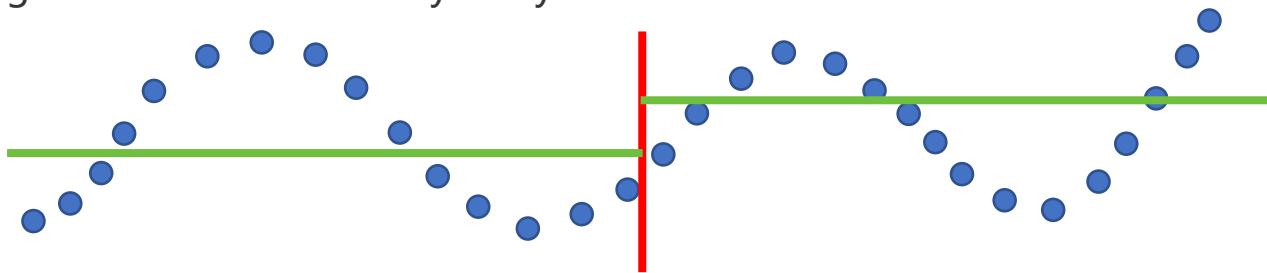




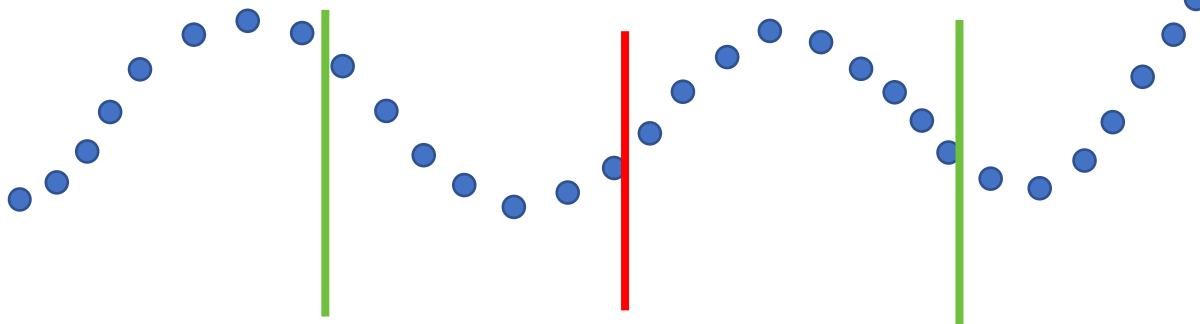
Kd-tree Construction

Division / Splitting Strategy

- Dividing axis is round-robin: x-y-z-x-y-z-x-.....

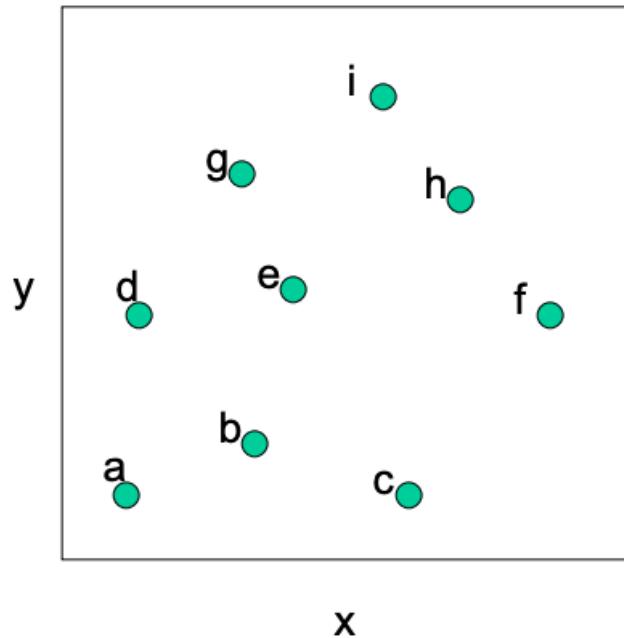


- Select the axis with the widest spread, so called "adaptive"



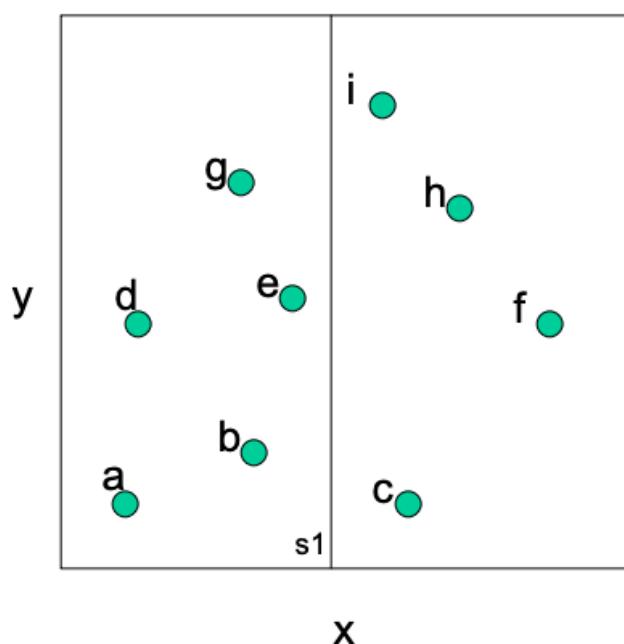


Kd-tree Construction





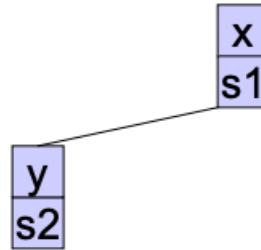
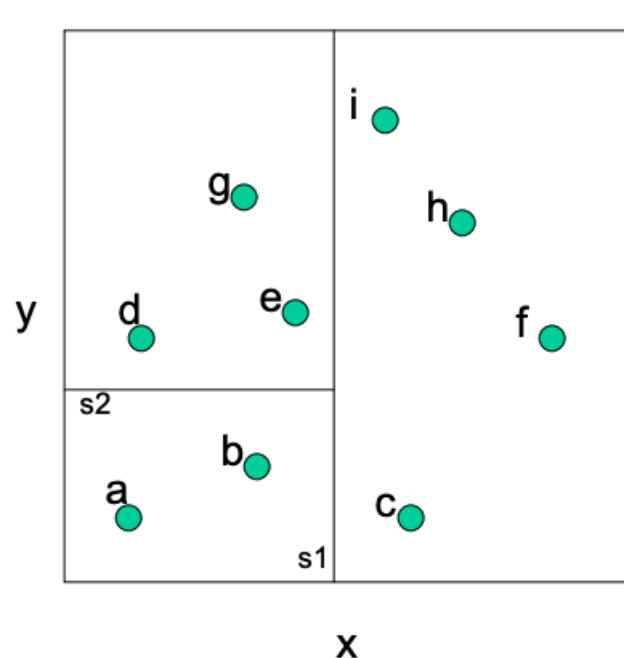
Kd-tree Construction



x
s1

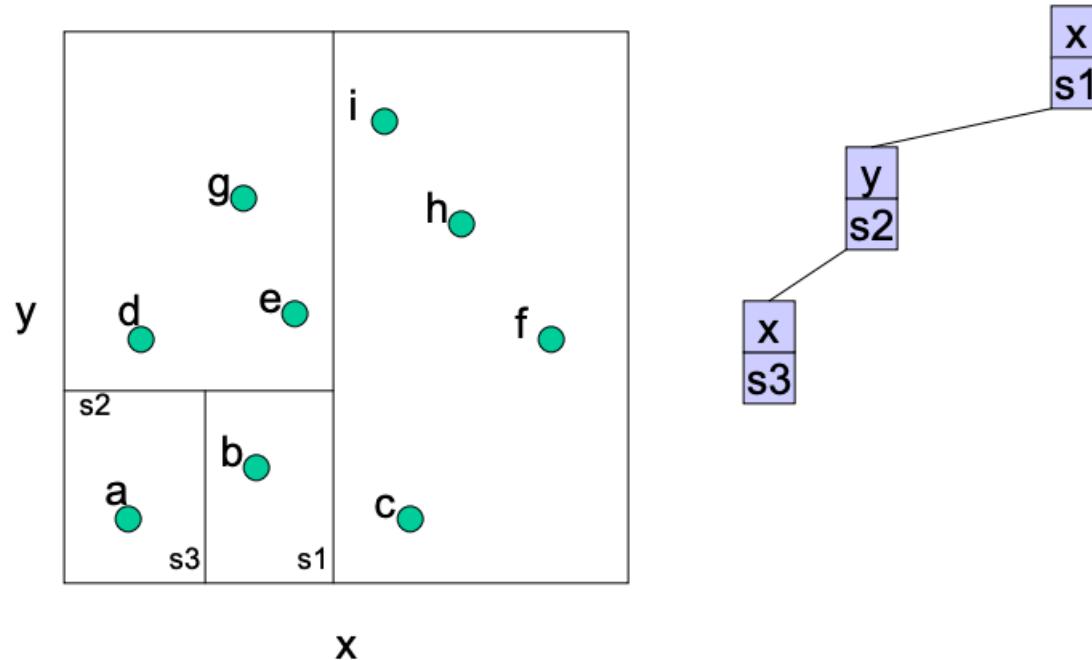


Kd-tree Construction



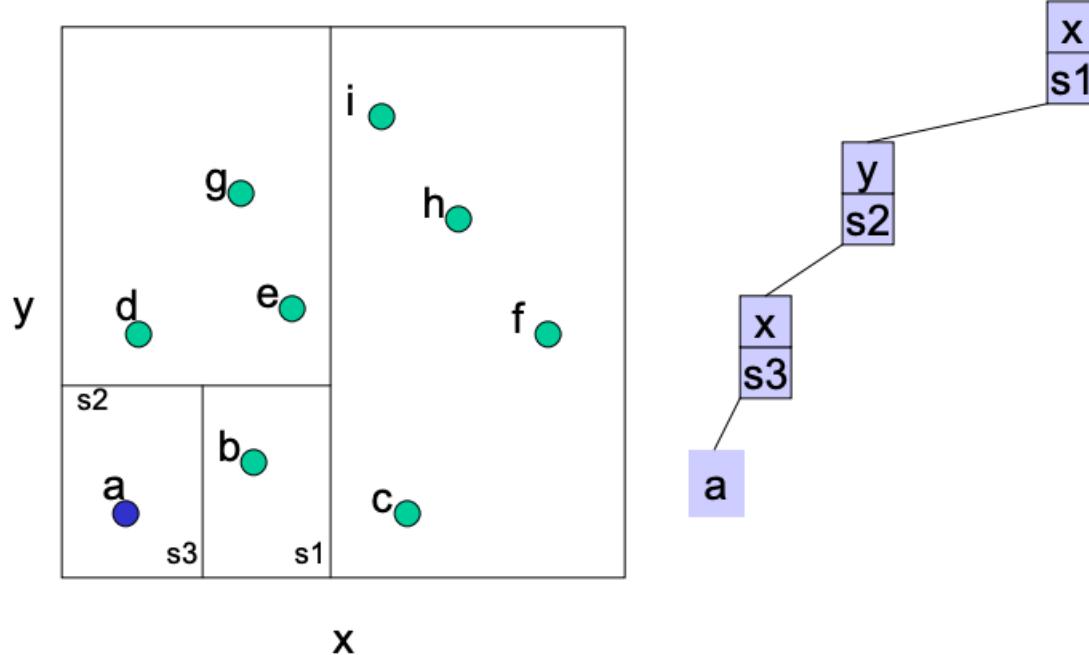


Kd-tree Construction



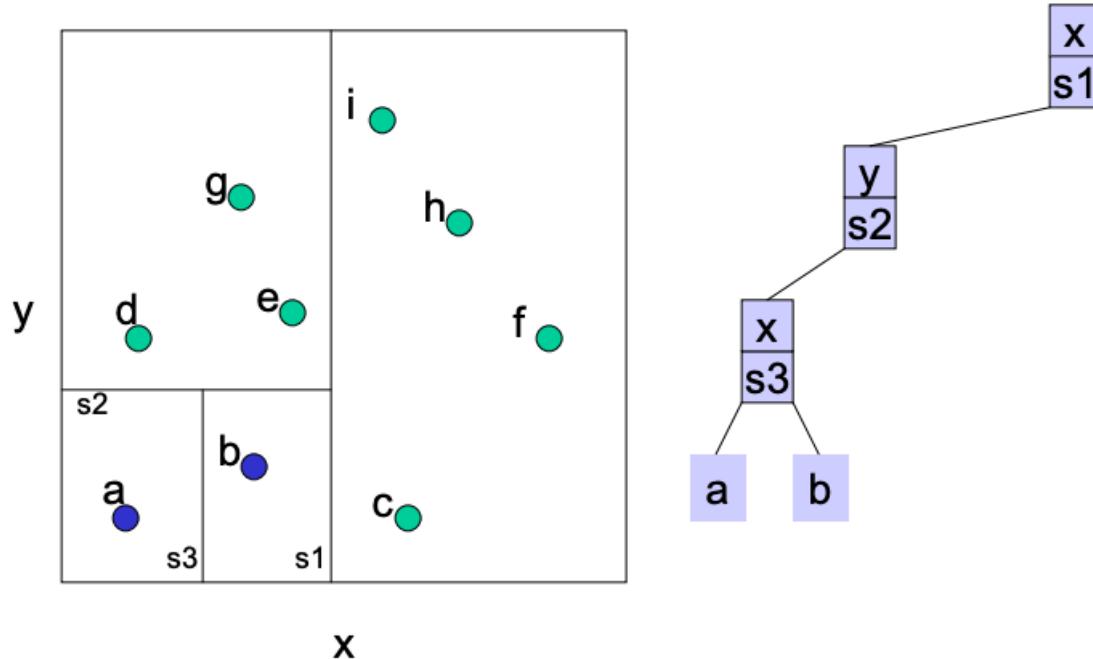


Kd-tree Construction



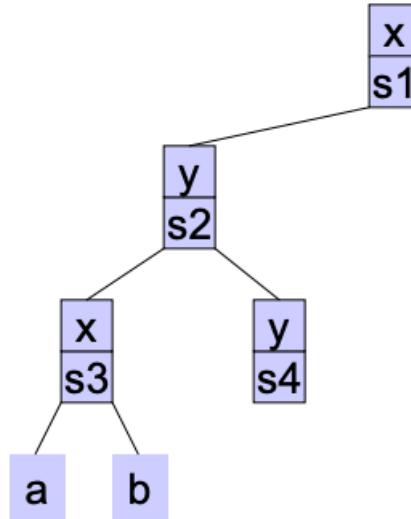
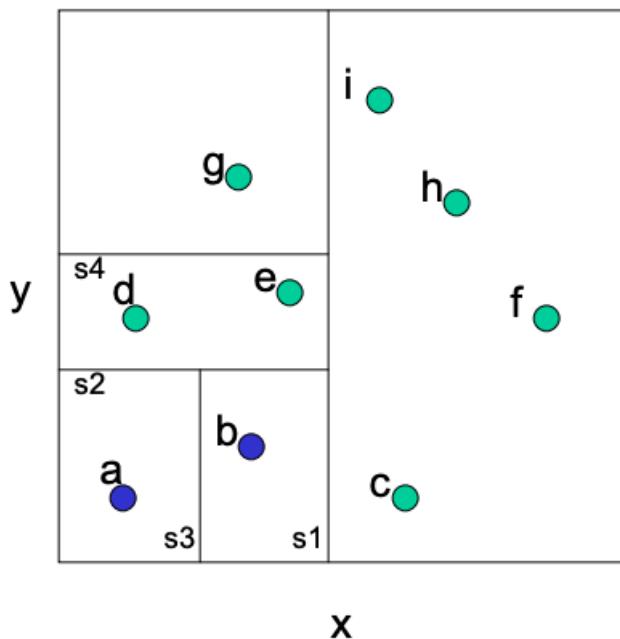


Kd-tree Construction



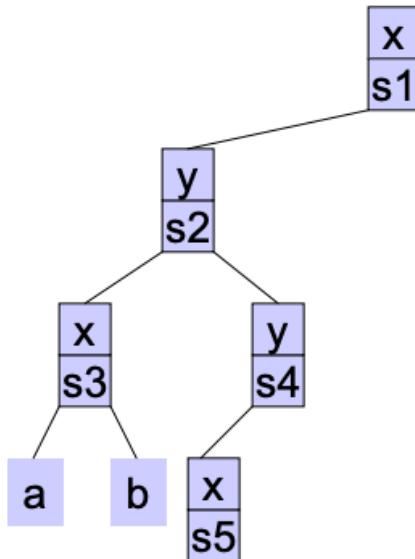
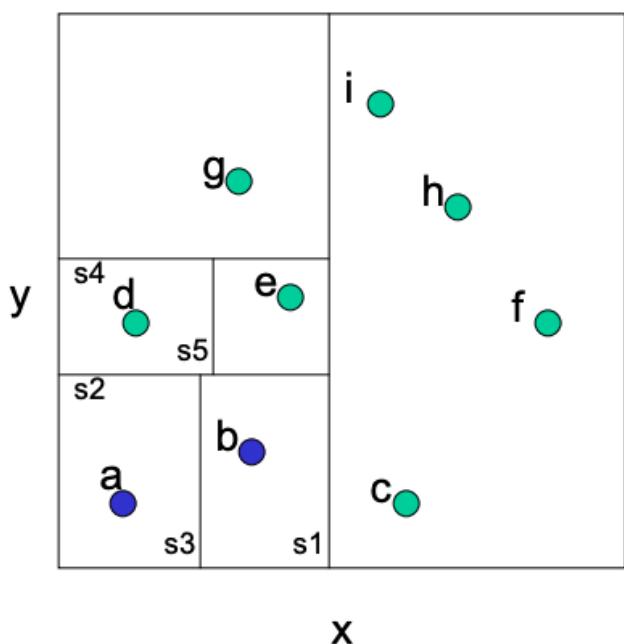


Kd-tree Construction



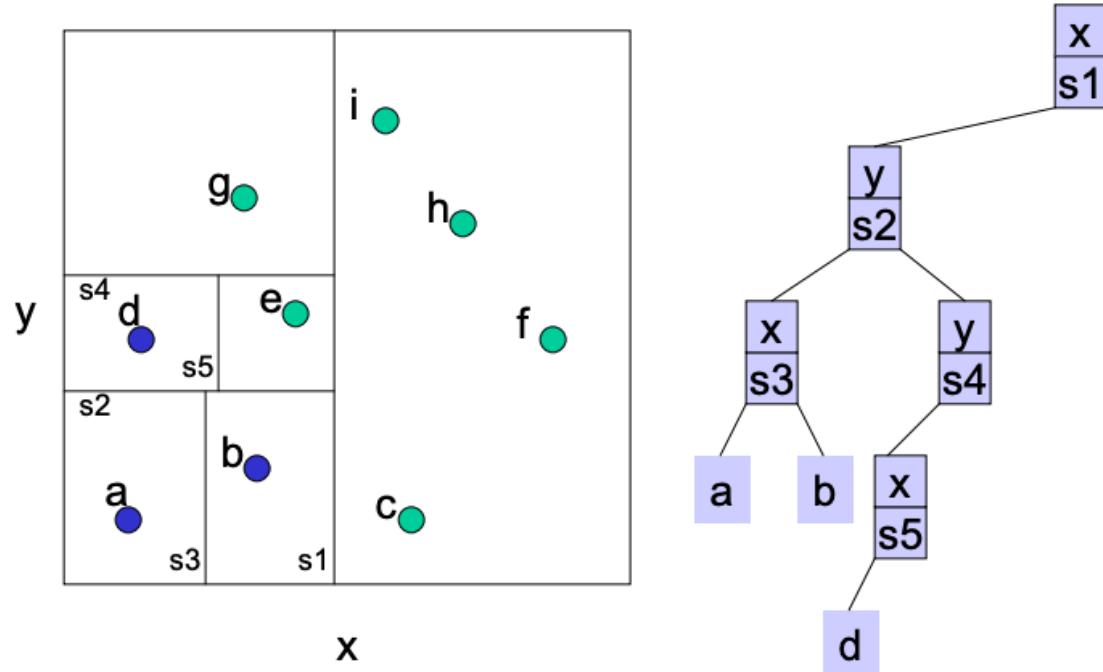


Kd-tree Construction



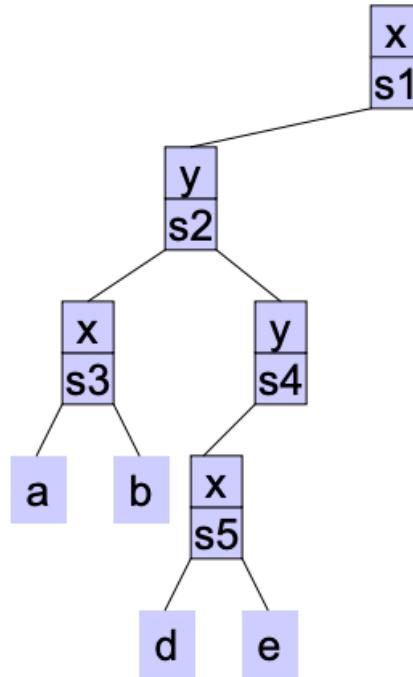
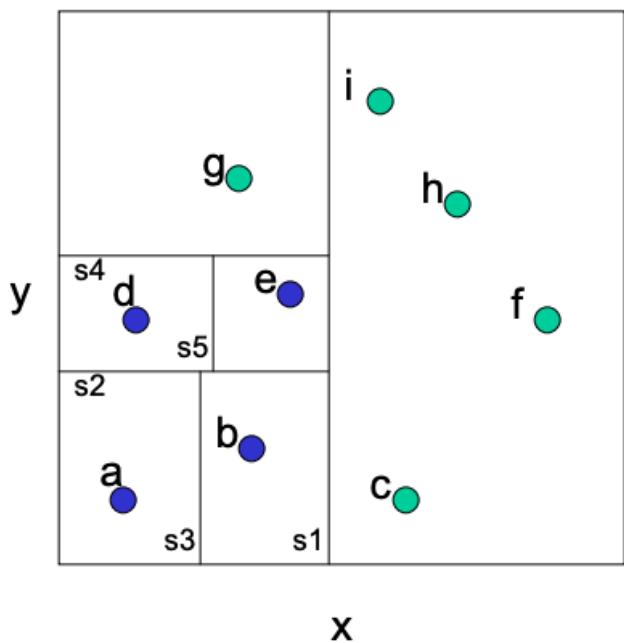


Kd-tree Construction



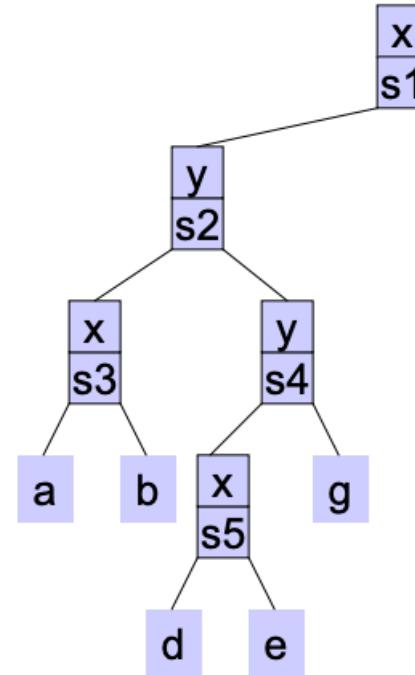
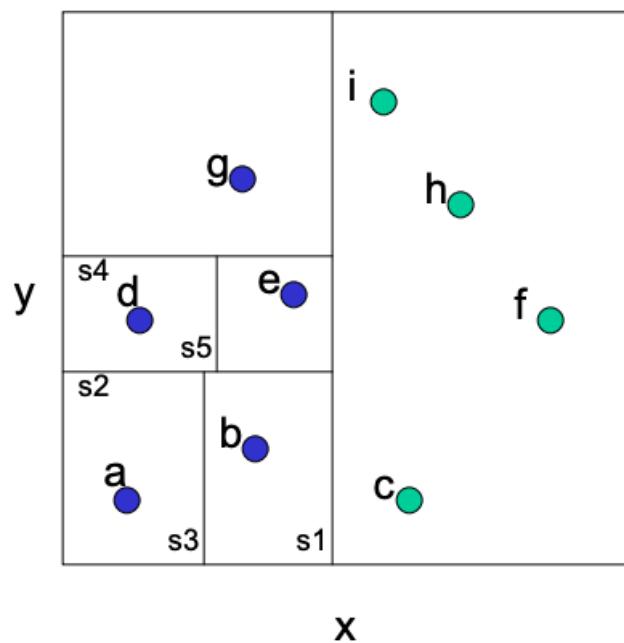


Kd-tree Construction



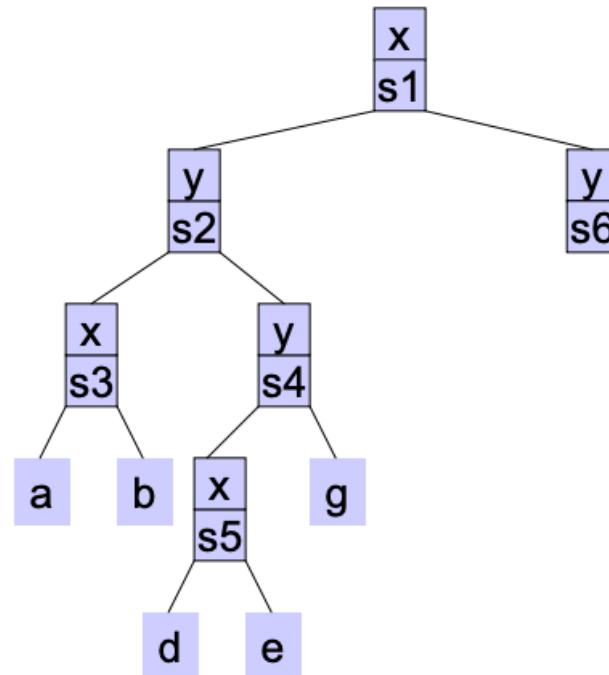
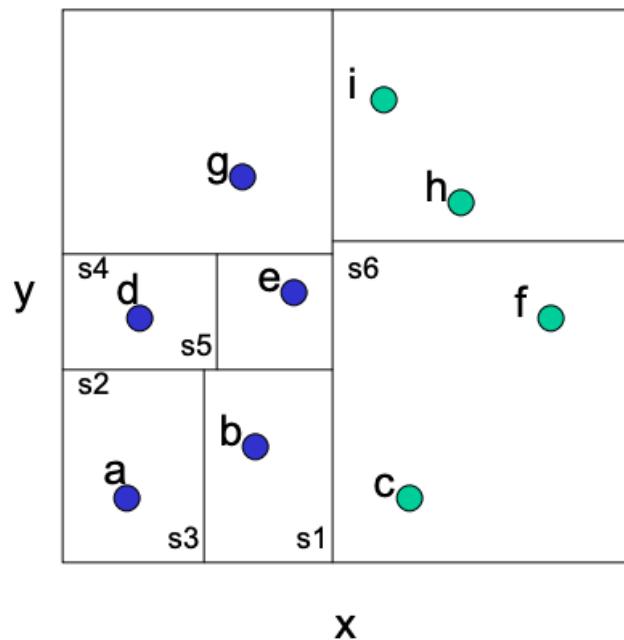


Kd-tree Construction



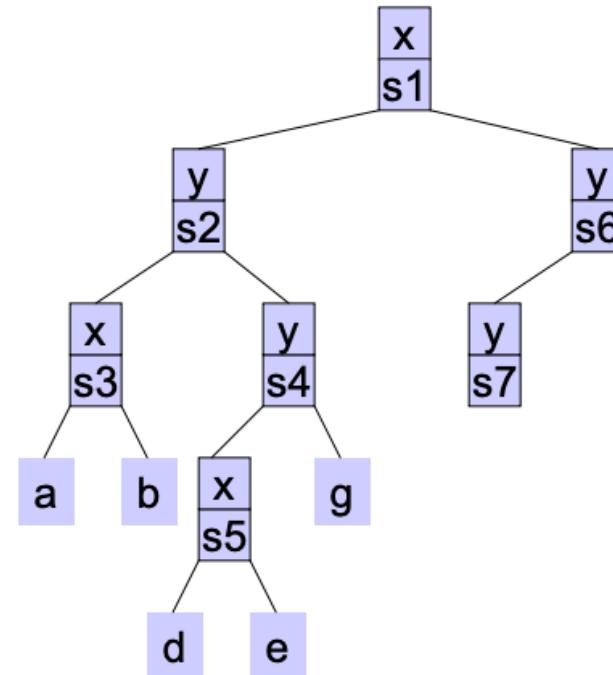
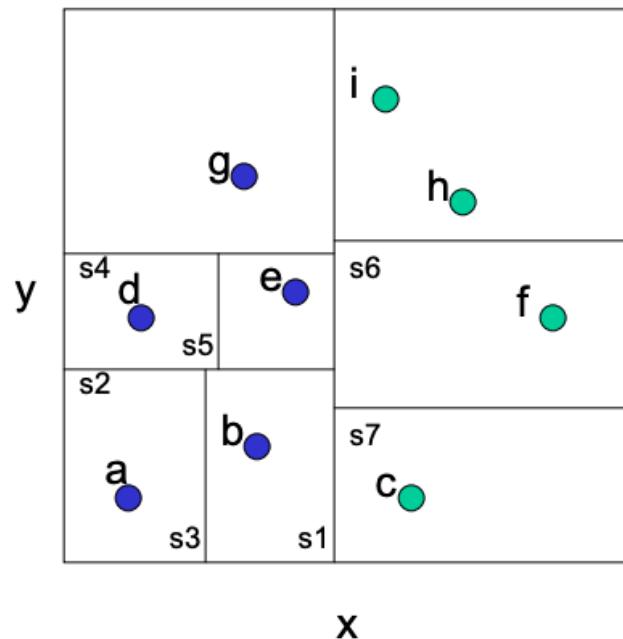


Kd-tree Construction



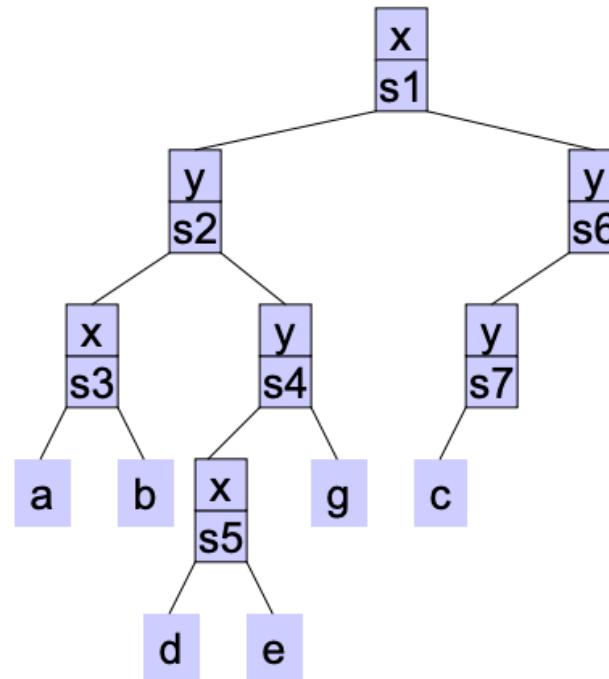
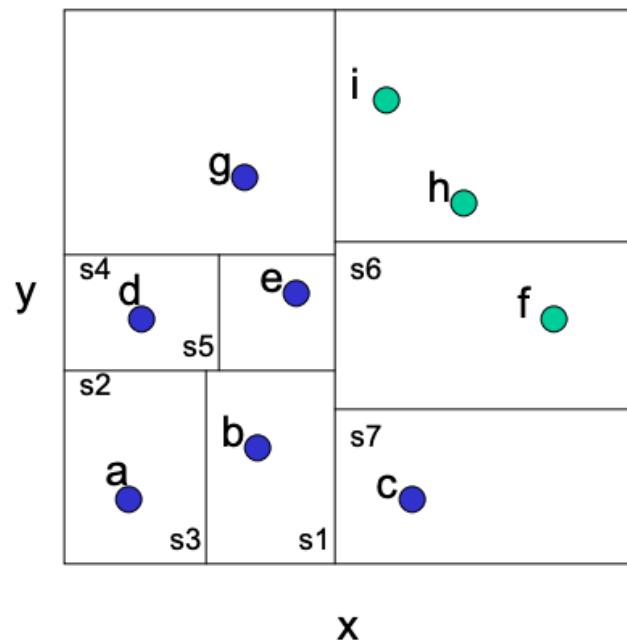


Kd-tree Construction



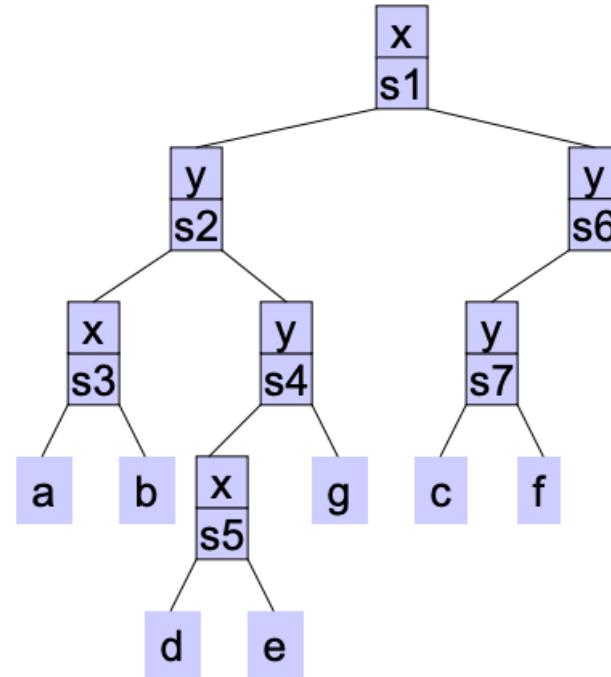
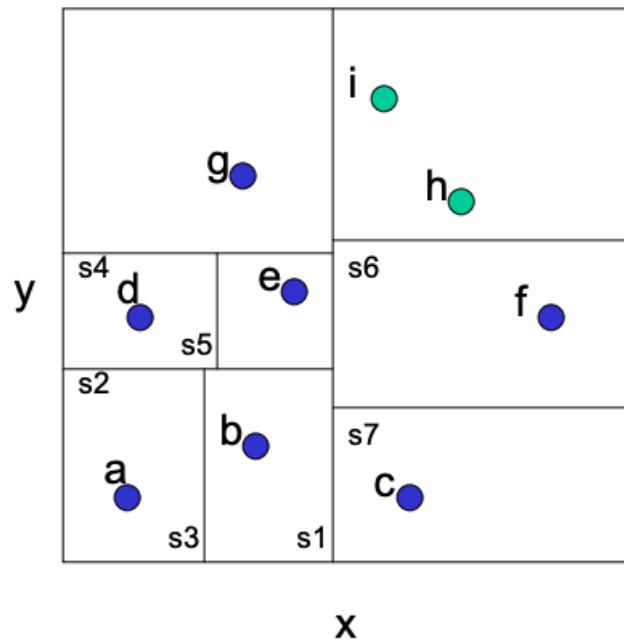


Kd-tree Construction



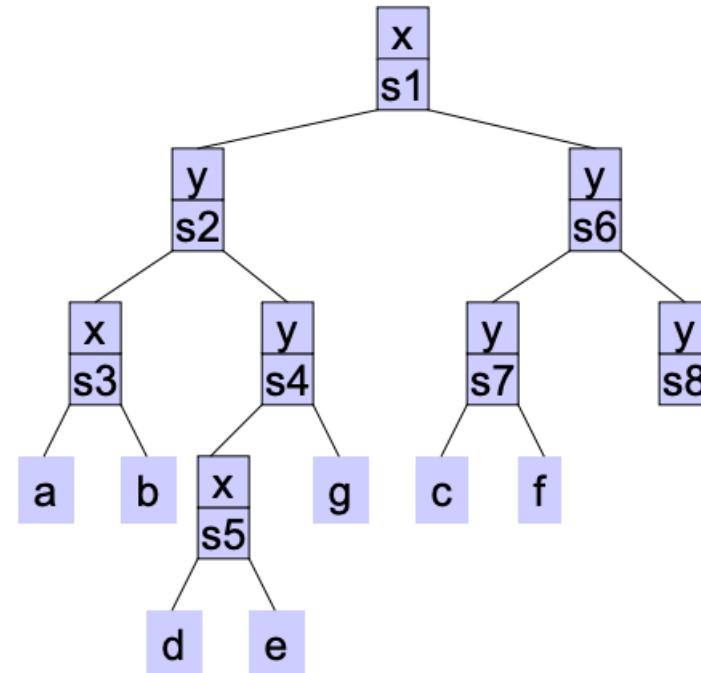
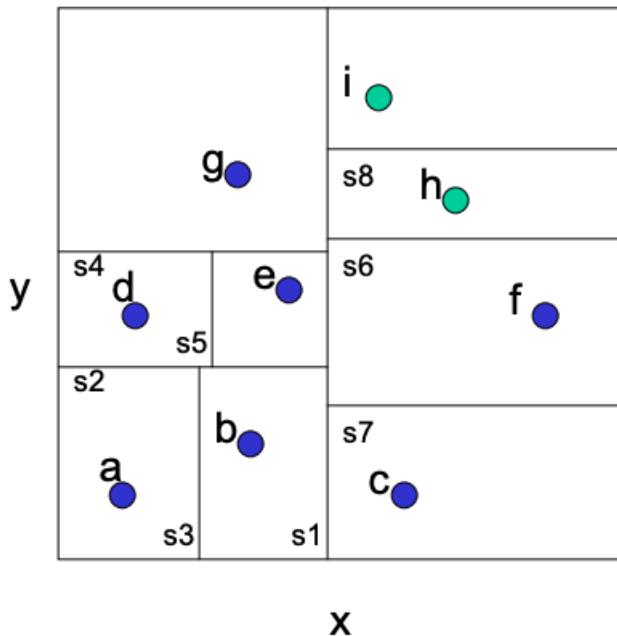


Kd-tree Construction



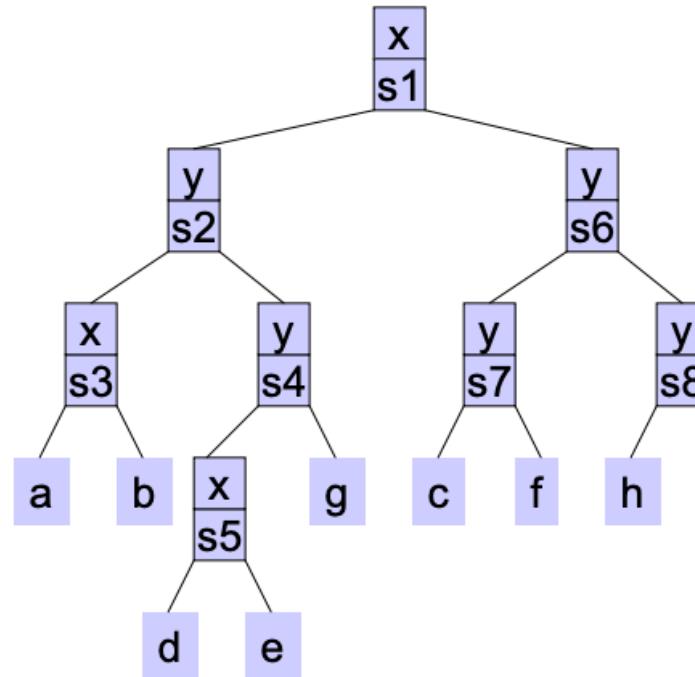
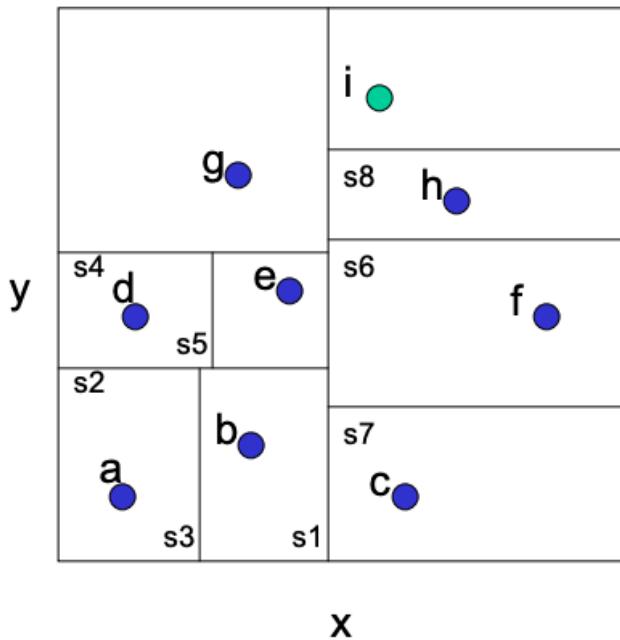


Kd-tree Construction



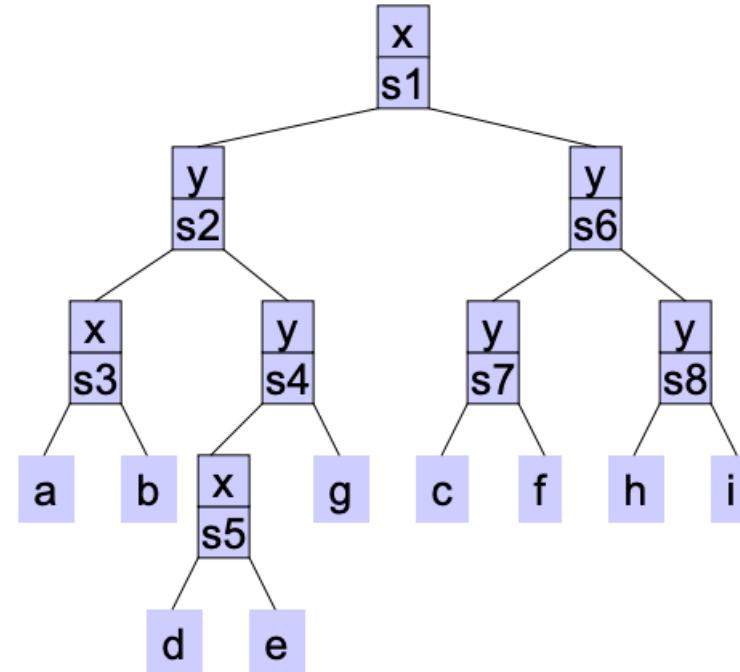
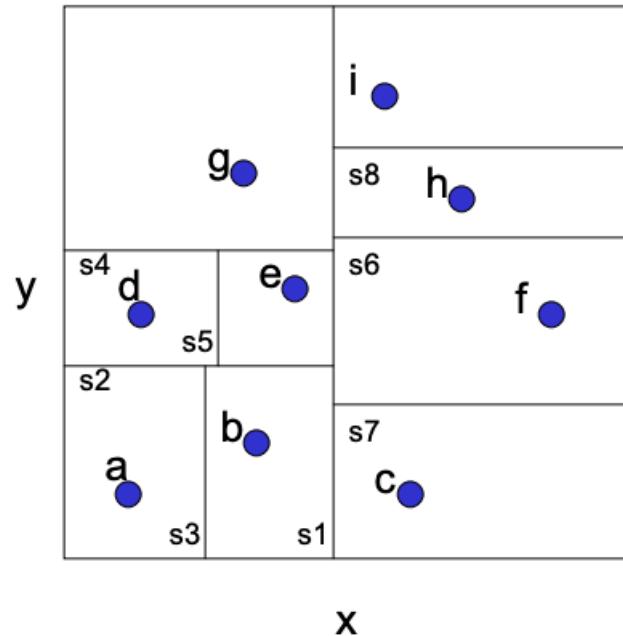


Kd-tree Construction





Kd-tree Construction





Kd-tree Construction



Talk is cheap, show me the code.
Linus Torvalds

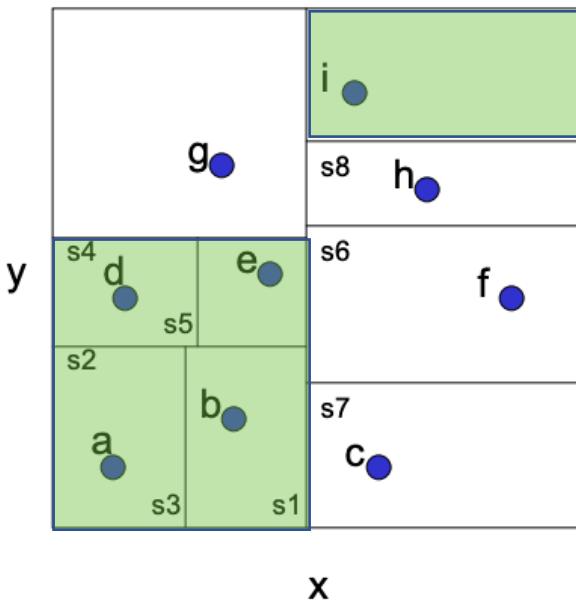


Kd-tree Node Representation

```
class Node:  
    def __init__(self, axis, value, left, right, point_indices):  
        self.axis = axis          Splitting position  
        self.value = value  
        self.left = left  
        self.right = right  
        self.point_indices = point_indices  
    def is_leaf(self):           Stores points that belongs to  
        if self.value is None:  
            return True  
        else:  
            return False
```

A node with
axis = x
value = ***
points = [a, b, d, e]

A leaf node with
axis = y
value = None
points = i



```

def kdrtree_recursive_build(root, db, point_indices, axis, leaf_size):
    """
    :param root:
    :param db: NxD
    :param db_sorted_idx_inv: NxD
    :param point_idx: M
    :param axis: scalar
    :param leaf_size: scalar
    :return:
    """

    if root is None:
        root = Node(axis, None, None, None, point_indices)

    # determine whether to split into left and right
    if len(point_indices) > leaf_size:
        # --- get the split position ---
        point_indices_sorted, _ = sort_key_by_val(db[point_indices, axis]) # M
        middle_left_idx = math.ceil(point_indices_sorted.shape[0] / 2) - 1
        middle_left_point_idx = point_indices_sorted[middle_left_idx]
        middle_left_point_value = db[middle_left_point_idx, axis]

        middle_right_idx = middle_left_idx + 1
        middle_right_point_idx = point_indices_sorted[middle_right_idx]
        middle_right_point_value = db[middle_right_point_idx, axis]

        root.value = (middle_left_point_value + middle_right_point_value) * 0.5
        # === get the split position ===
        root.left = kdrtree_recursive_build(root.left,
                                             db,
                                             point_indices_sorted[0:middle_right_idx],
                                             axis_round_robin(axis, dim=db.shape[1]),
                                             leaf_size)
        root.right = kdrtree_recursive_build(root.right,
                                              db,
                                              point_indices_sorted[middle_right_idx:],
                                              axis_round_robin(axis, dim=db.shape[1]),
                                              leaf_size)

    return root

```

A leaf node can contain
more than 1 point

Sort the points in this node, get the median position

```

def axis_round_robin(axis, dim):
    if axis == dim-1:
        return 0
    else:
        return axis + 1

```



Kd-tree Construction Complexity

❖ The example shown here is not optimal because of sorting at each level of the tree

- Time complexity of around $O(n \log n \log n)$
- Space complexity of $O(kn + n \log n)$ → can be easily reduced to $O(kn + n)$
 - Only store points at leaf

❖ Can we select median instead of sorting?

- If median finding is $O(n)$
- Kd-tree is $O(n \log n)$
- Median finding in $O(n)$ is complicated, but **possible!**

❖ $O(kn \log n)$ method

- Building a Balanced k-d Tree in $O(kn \log n)$ Time
 - Russel A. Brown, Journal of Computer Graphics Techniques, 2015



Kd-tree Construction Complexity

Simple methods that work well in practice

- Sample a **subset** of point in each node for sorting, instead of sorting all points
 - $O(n' \log n)$ or $O(n' \log n' \log n)$
- Use **mean** instead of median
 - An easy way to achieve $O(n \log n)$

They don't guarantee balanced kd-tree

- Balanced tree - each leaf node is approximately the same distance from the root
- Similar to the "chain" example in BST.

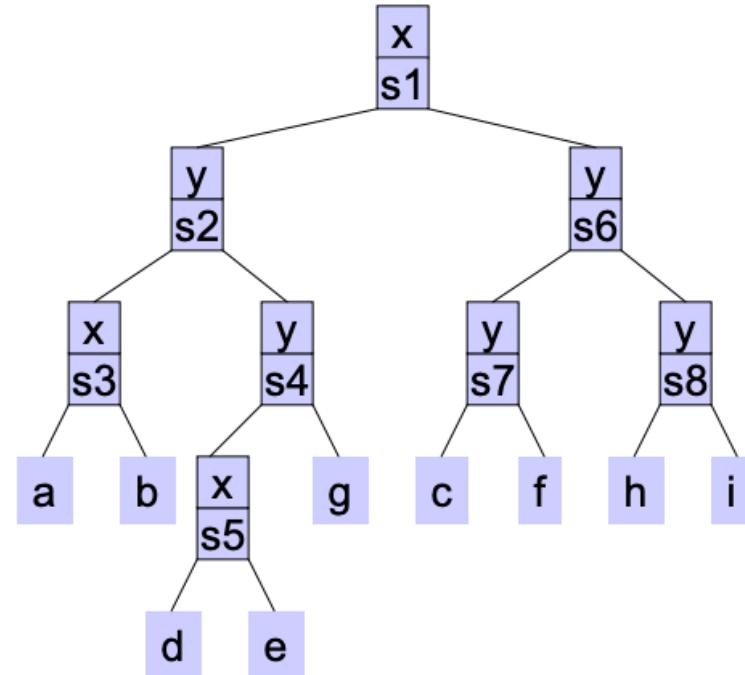
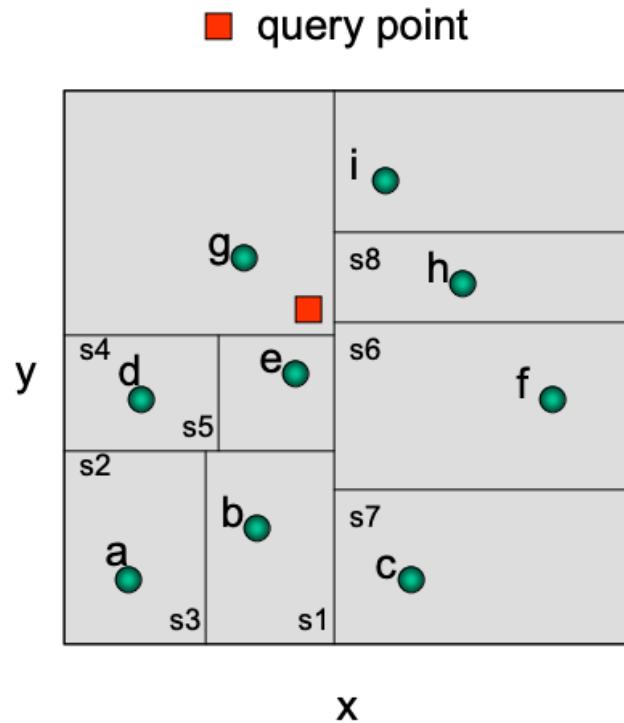


Kd-tree – kNN Search

-  Start from root
-  Reach the leaf node than covers the query point
 - Compare all points in the leaf node
-  Go up and traverse the tree

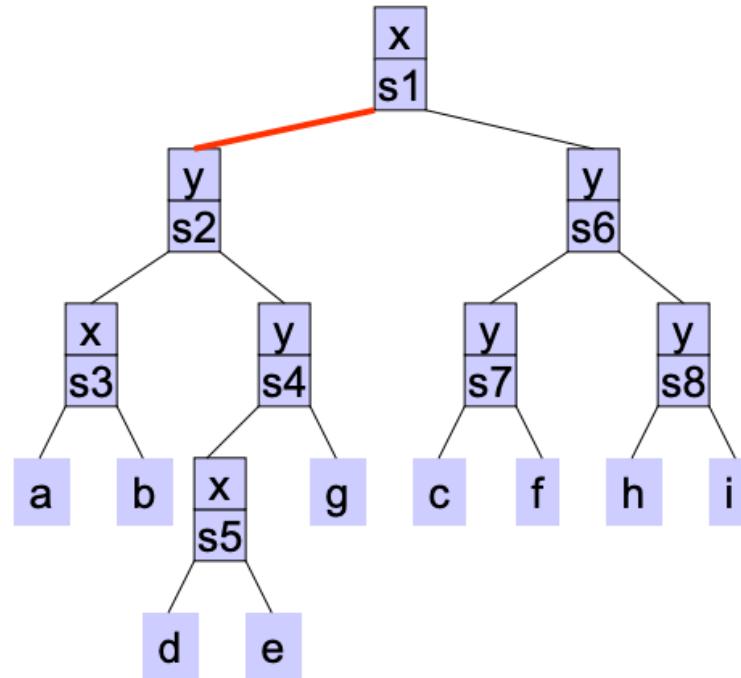
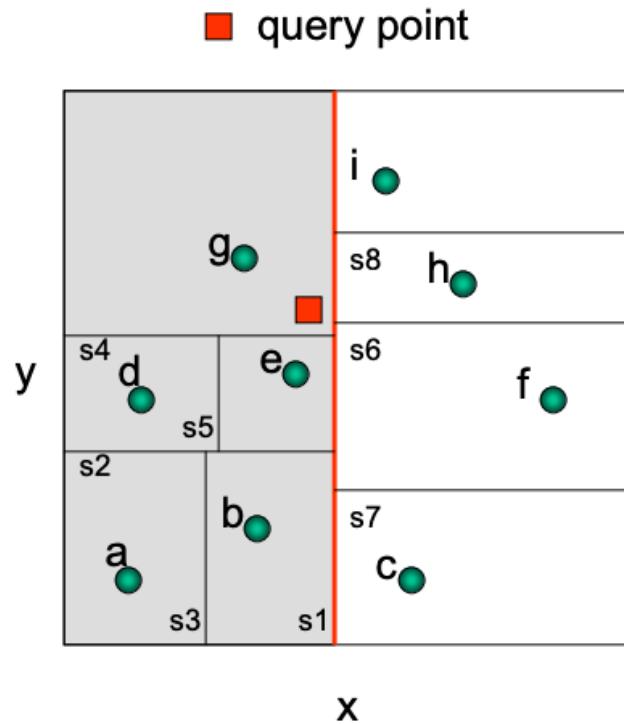


Kd-tree – kNN Search



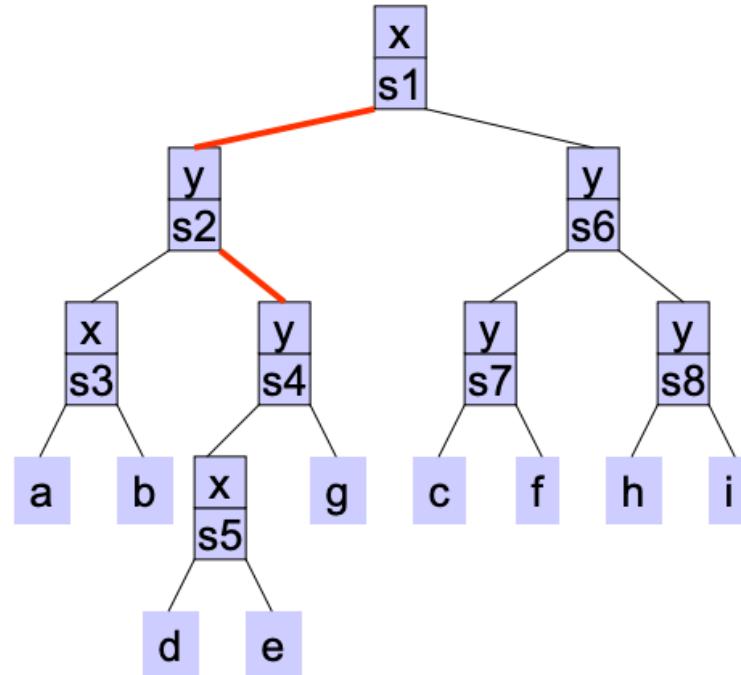
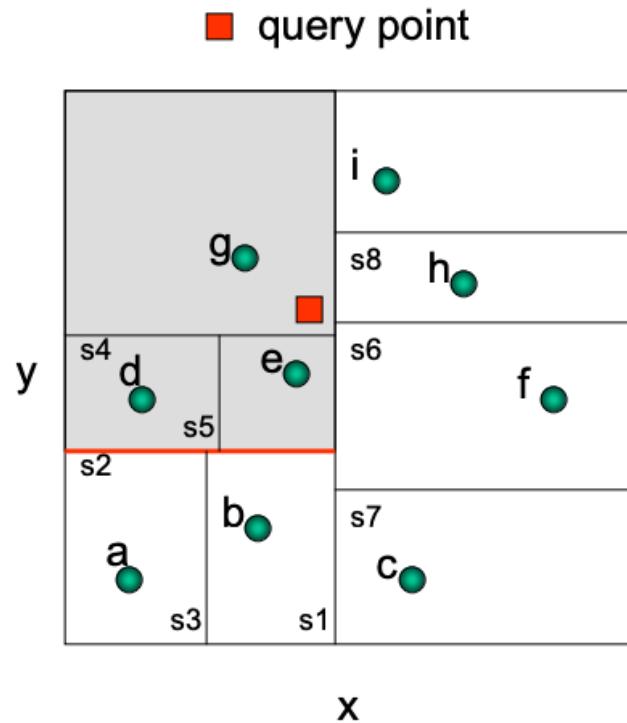


Kd-tree – kNN Search



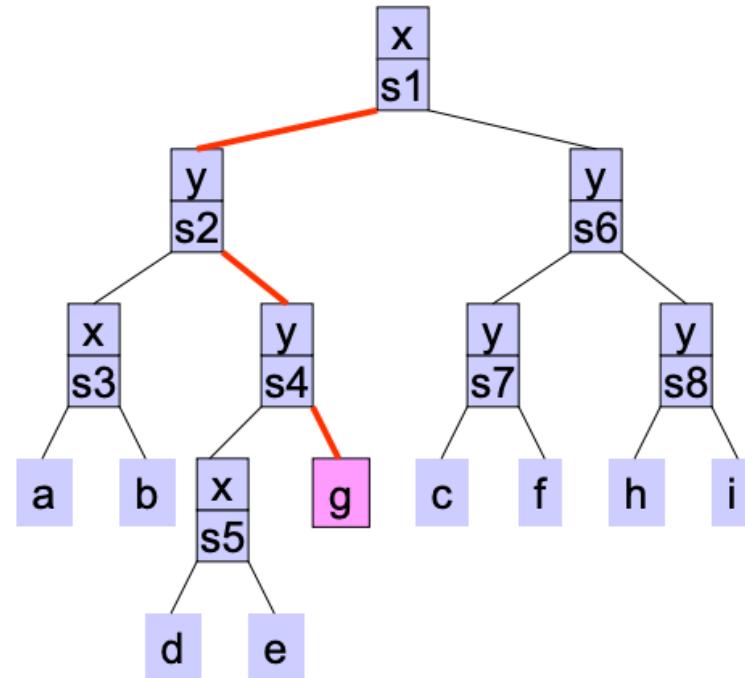
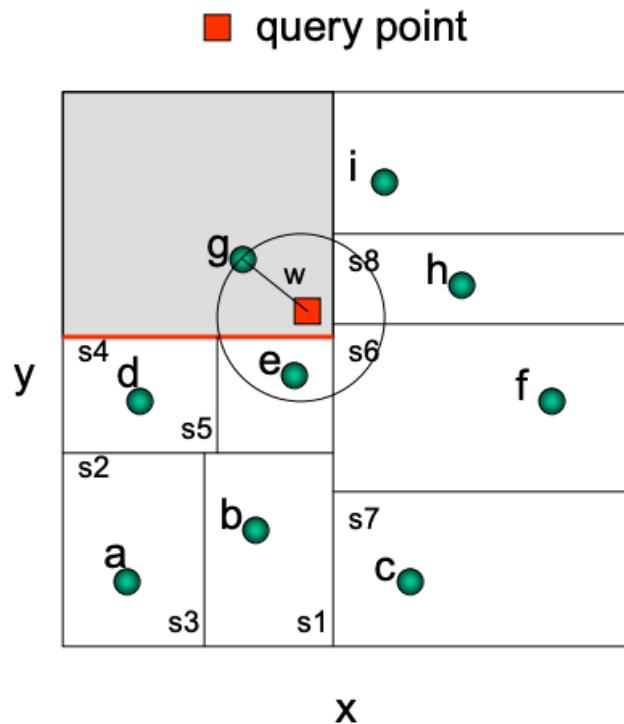


Kd-tree – kNN Search



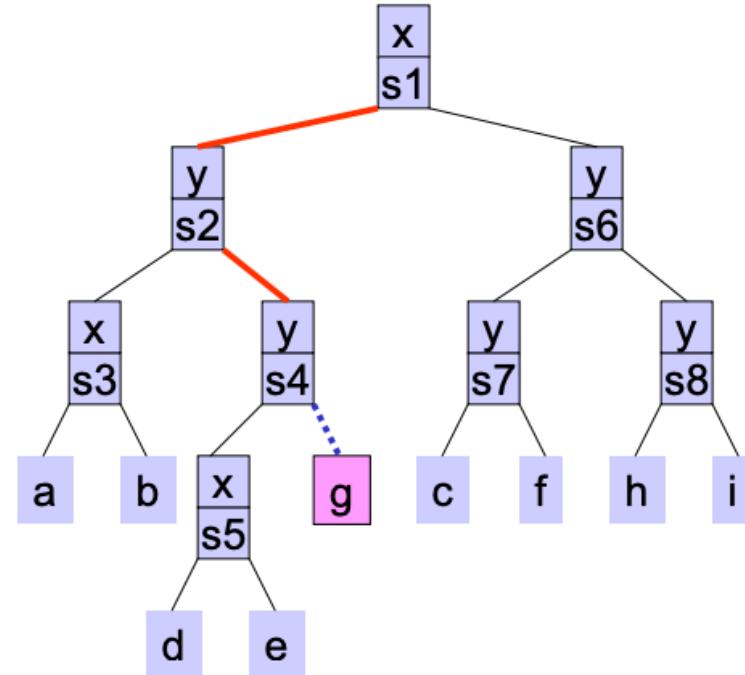
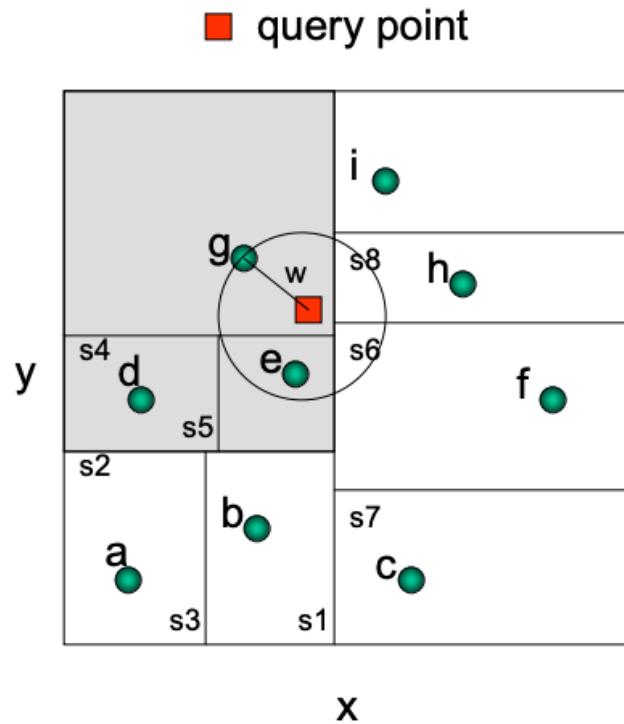


Kd-tree – kNN Search



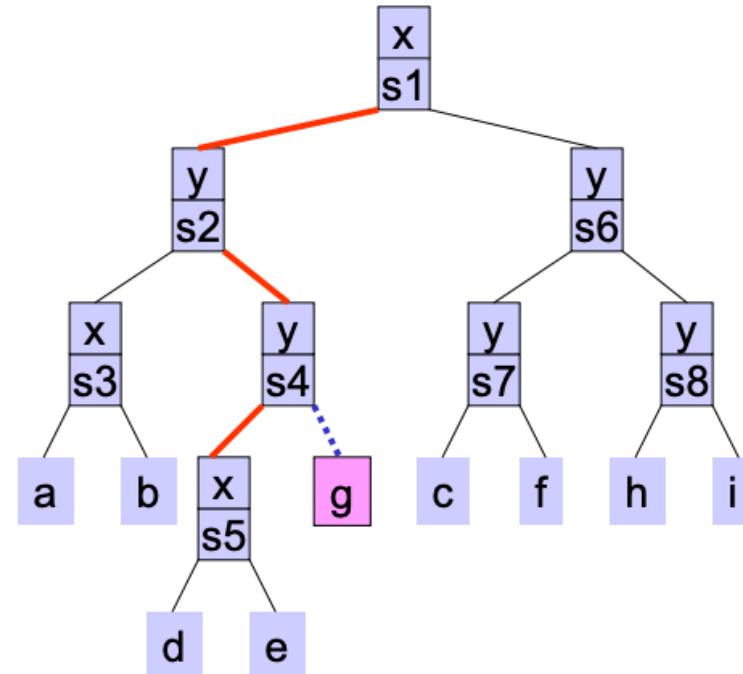
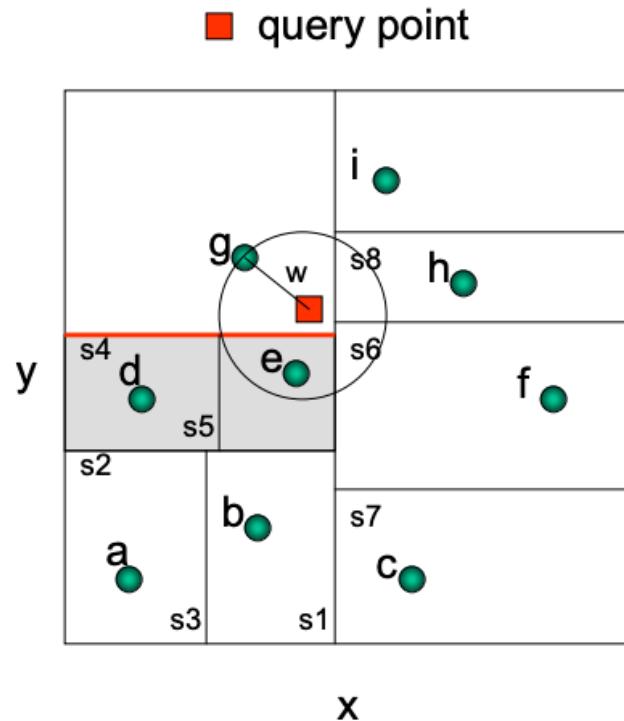


Kd-tree – kNN Search



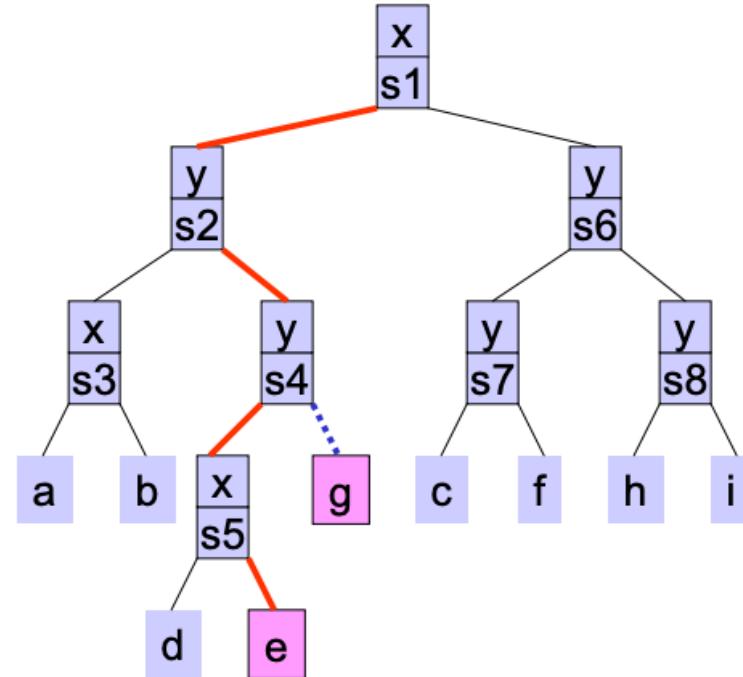
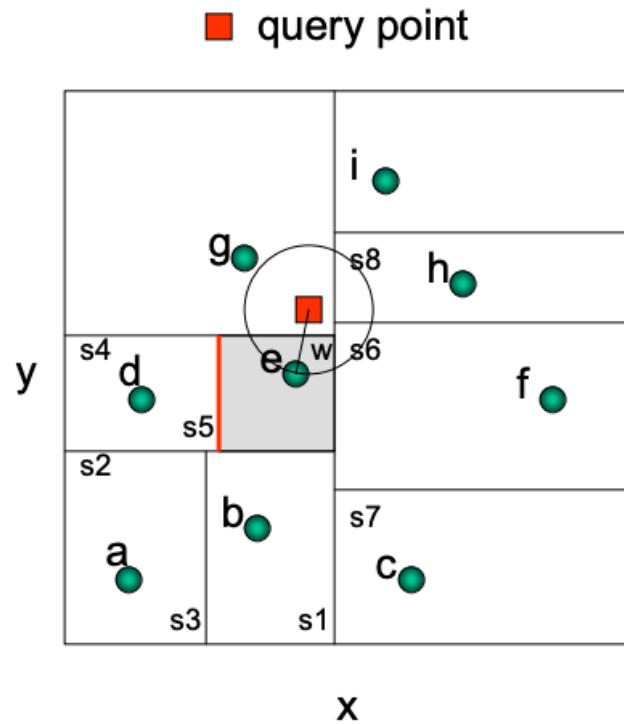


Kd-tree – kNN Search



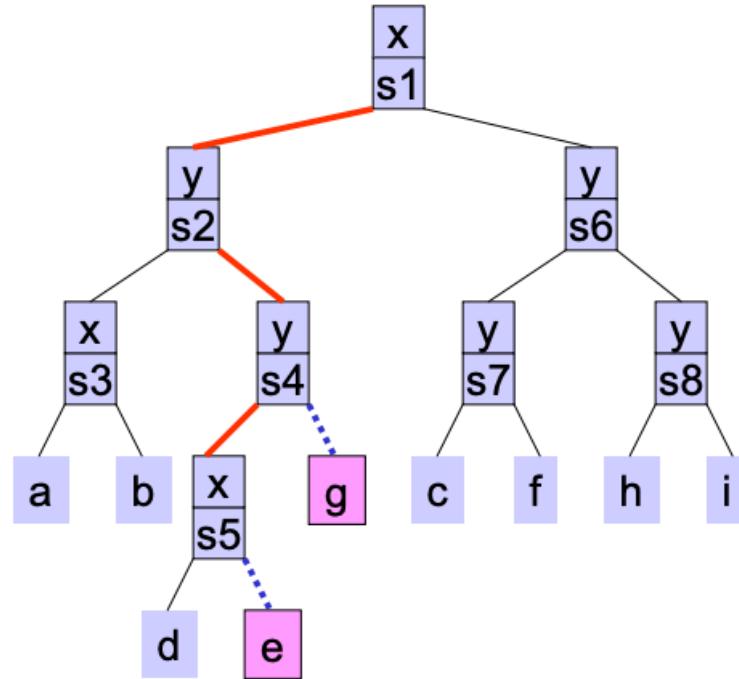
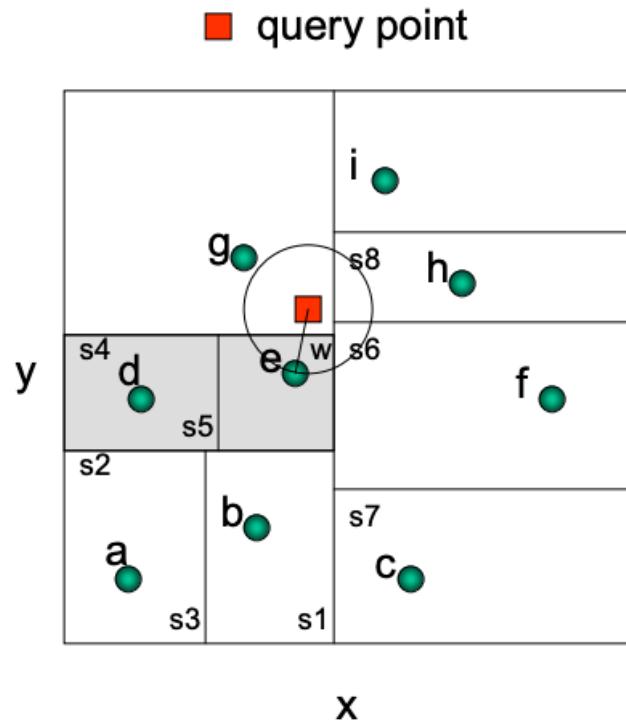


Kd-tree – kNN Search



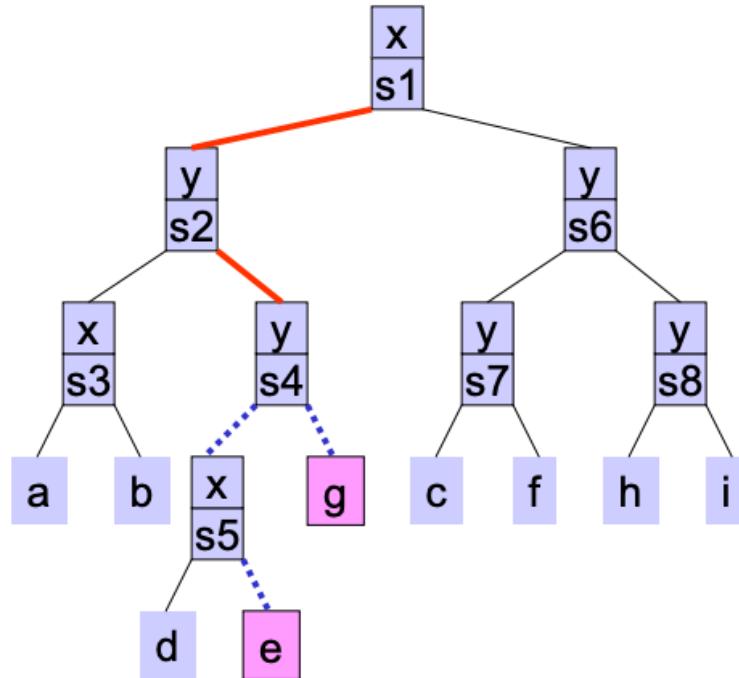
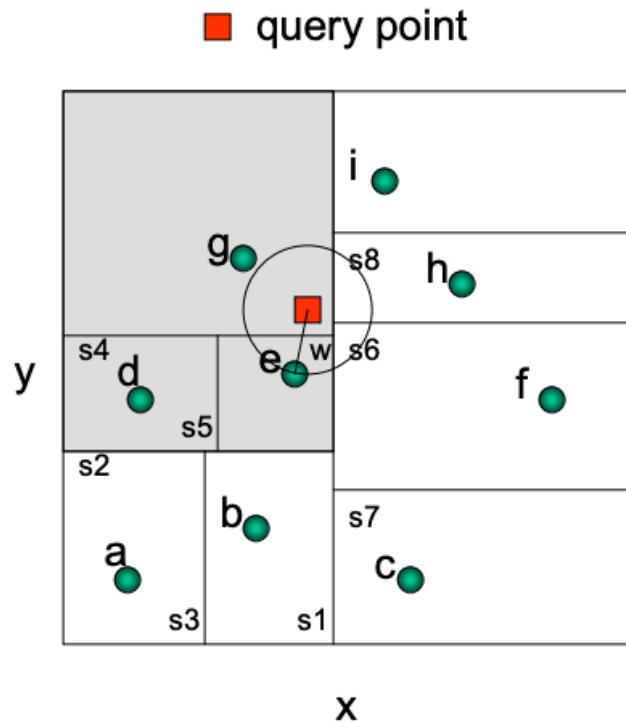


Kd-tree – kNN Search



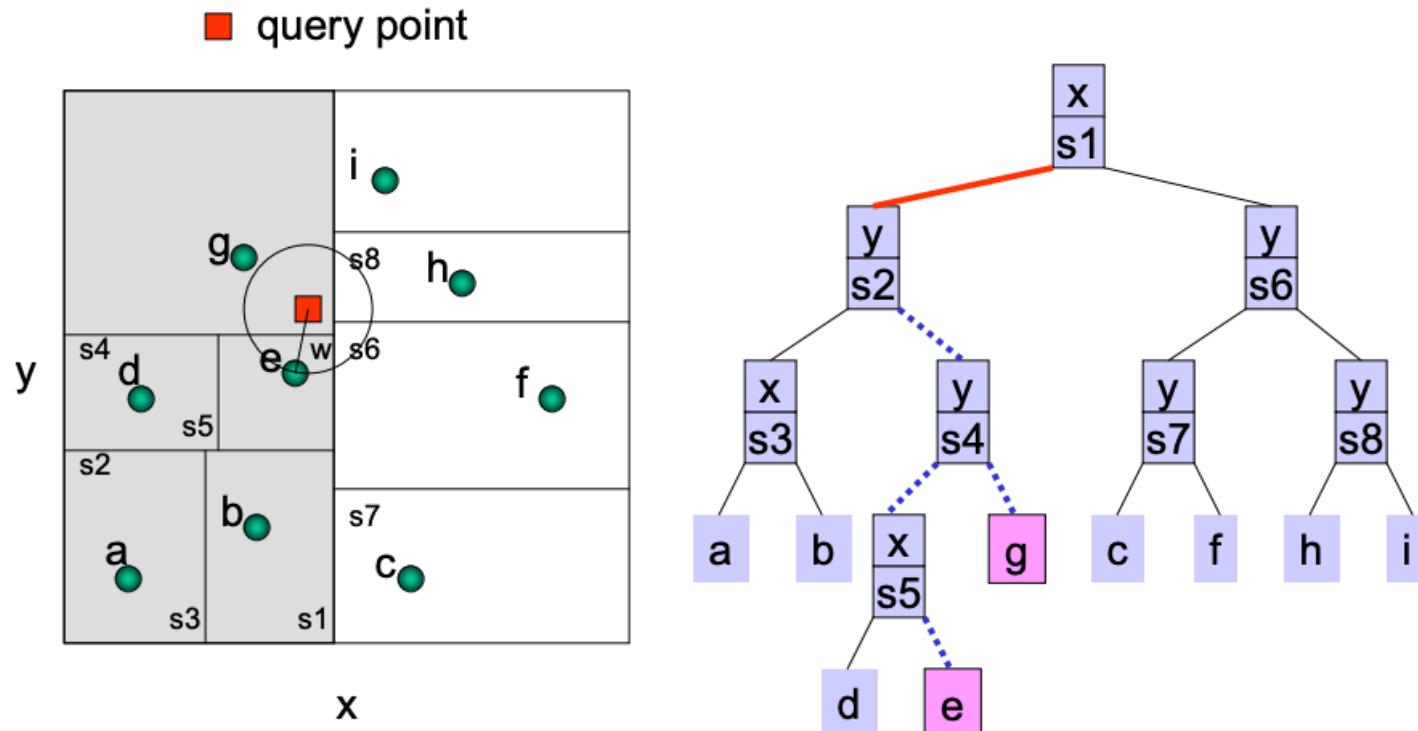


Kd-tree – kNN Search



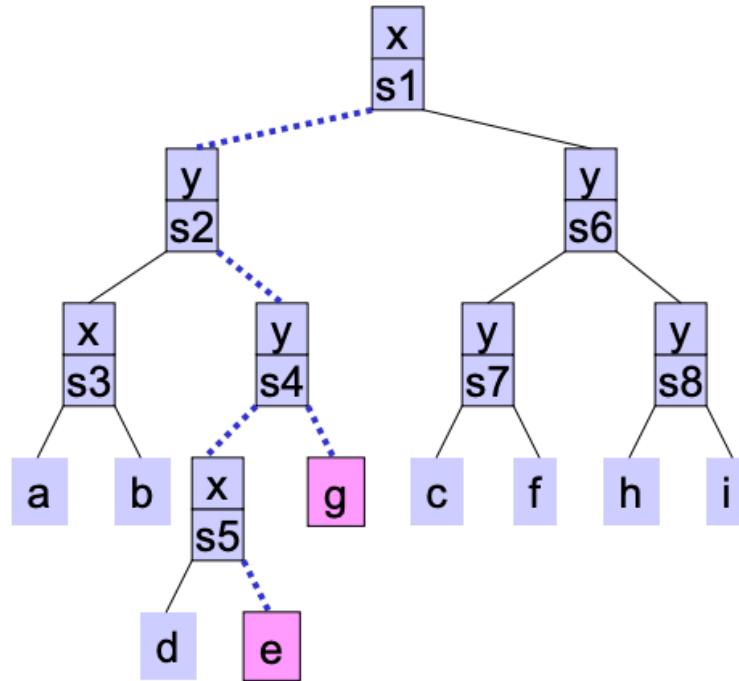
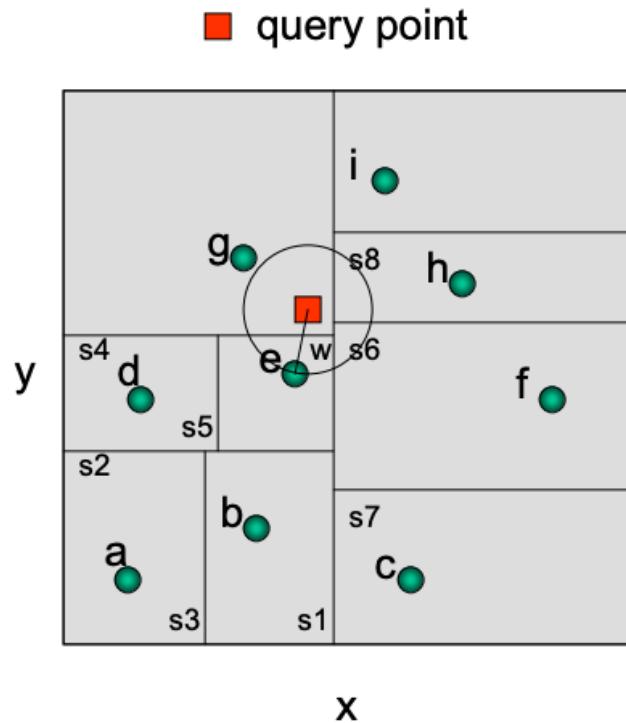


Kd-tree – kNN Search



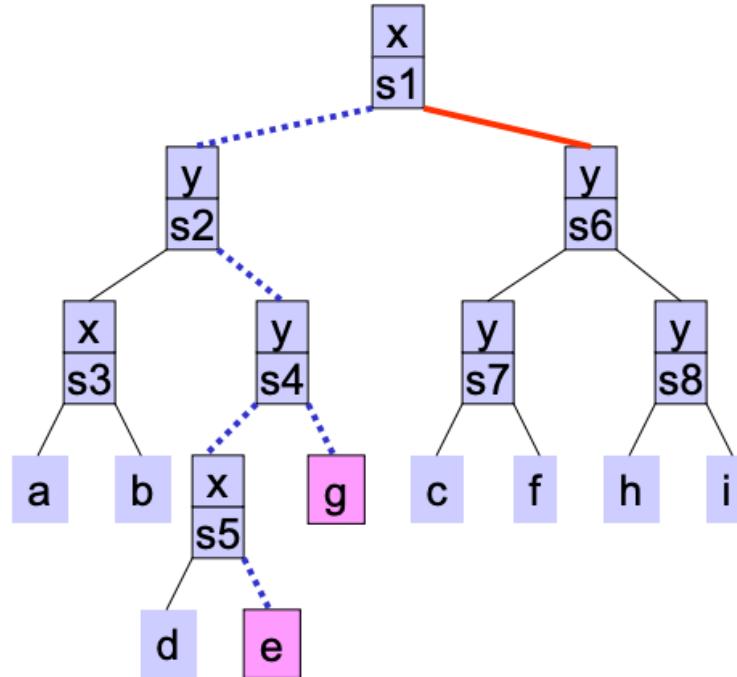
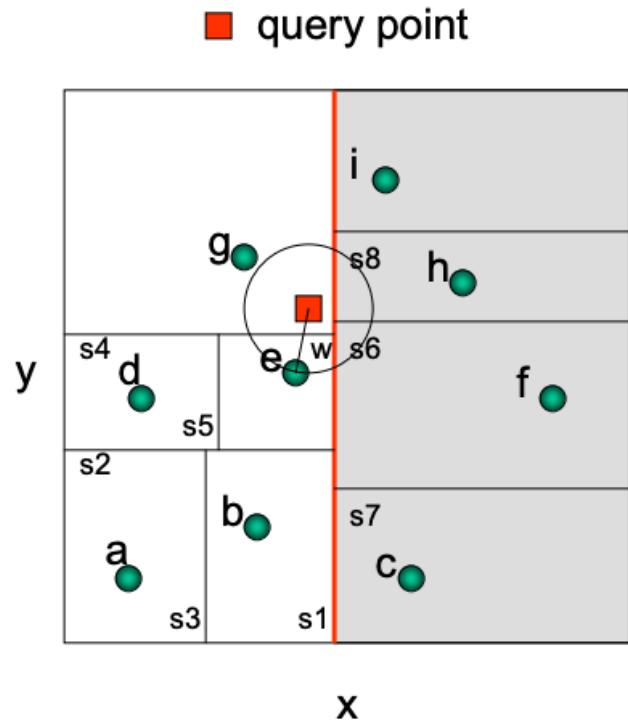


Kd-tree – kNN Search



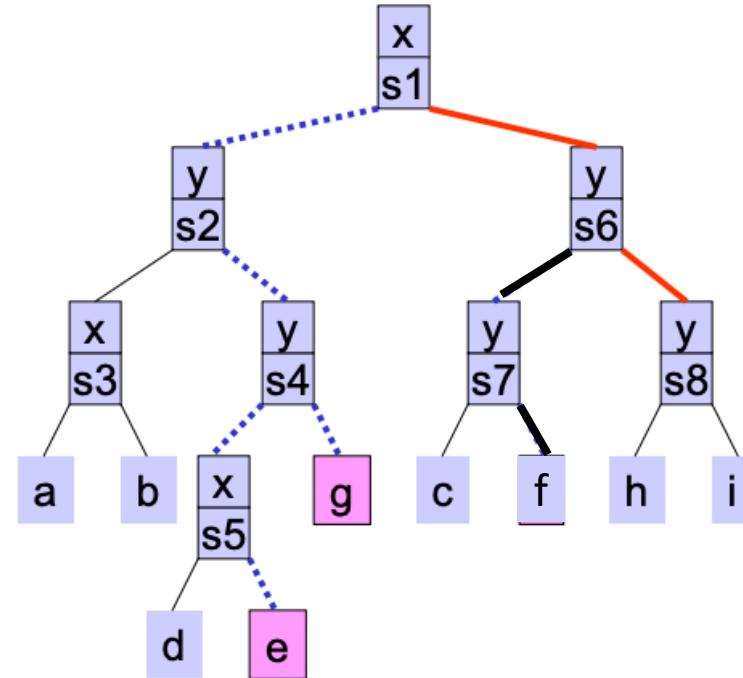
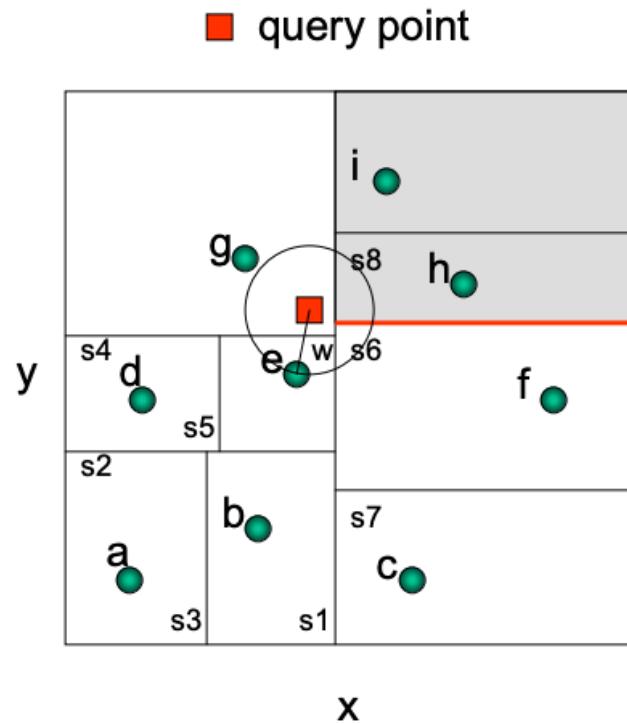


Kd-tree – kNN Search



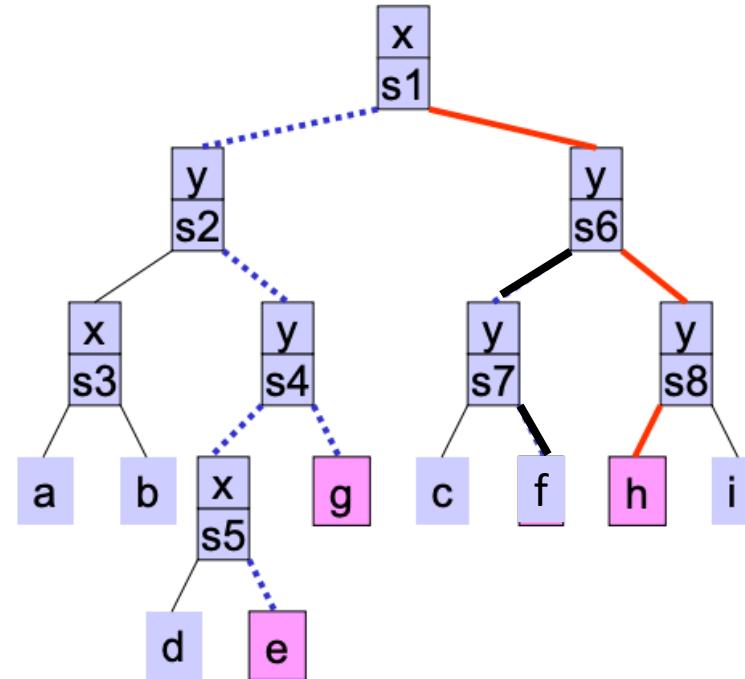
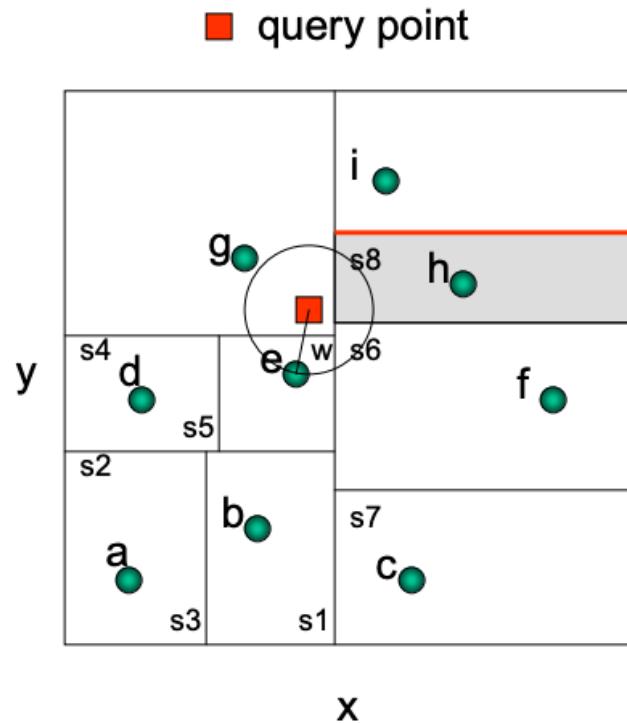


Kd-tree – kNN Search



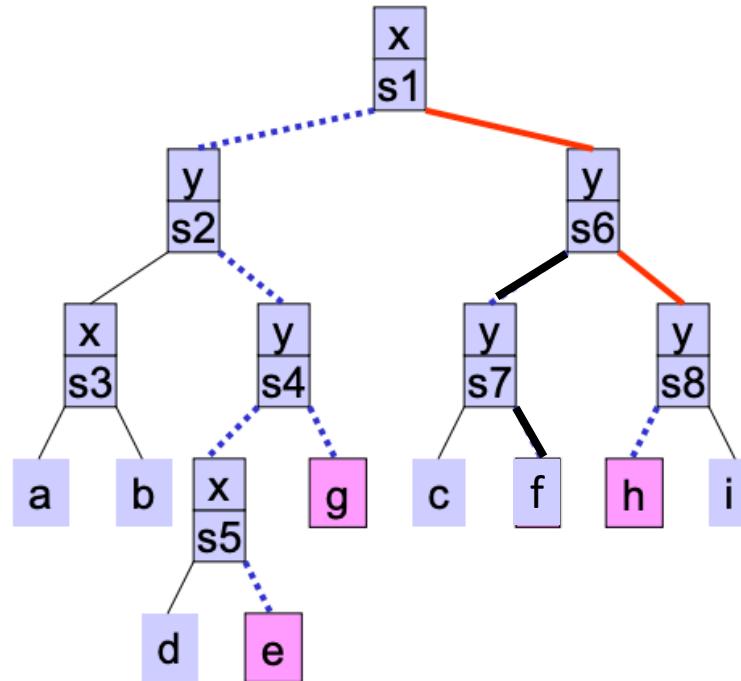
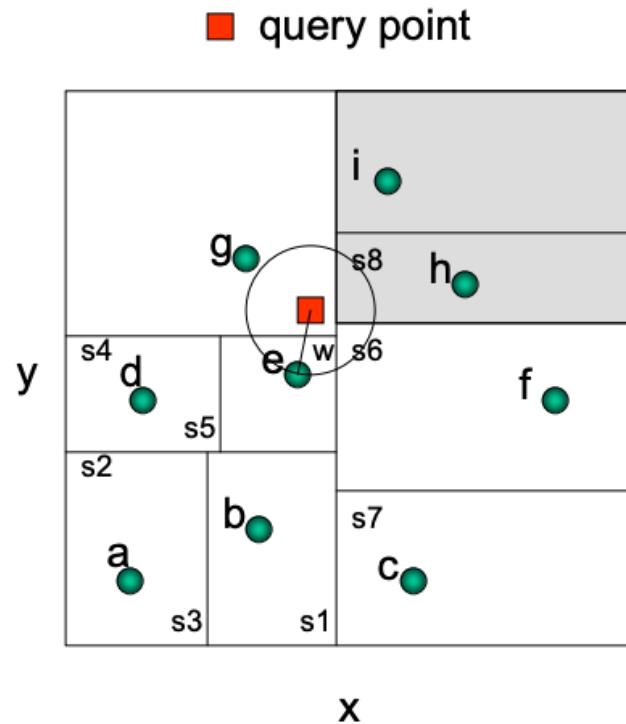


Kd-tree – kNN Search



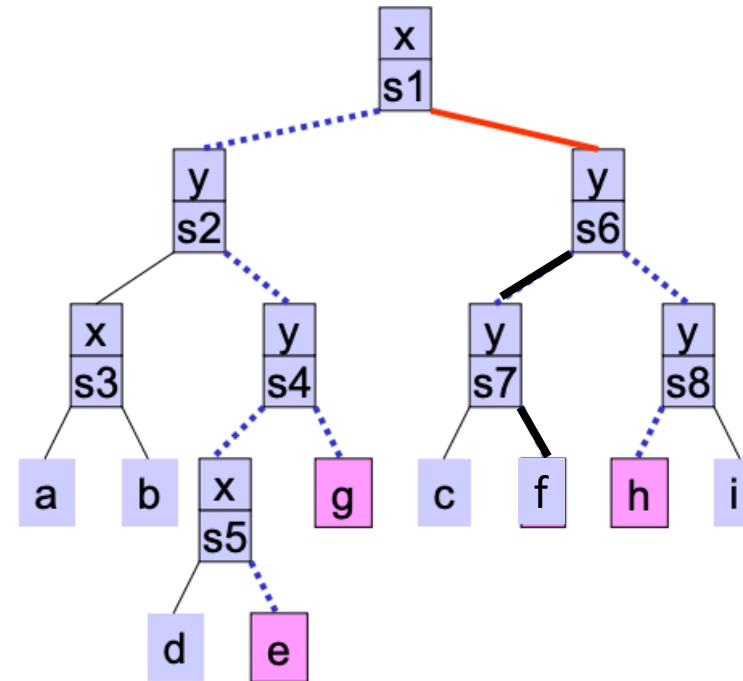
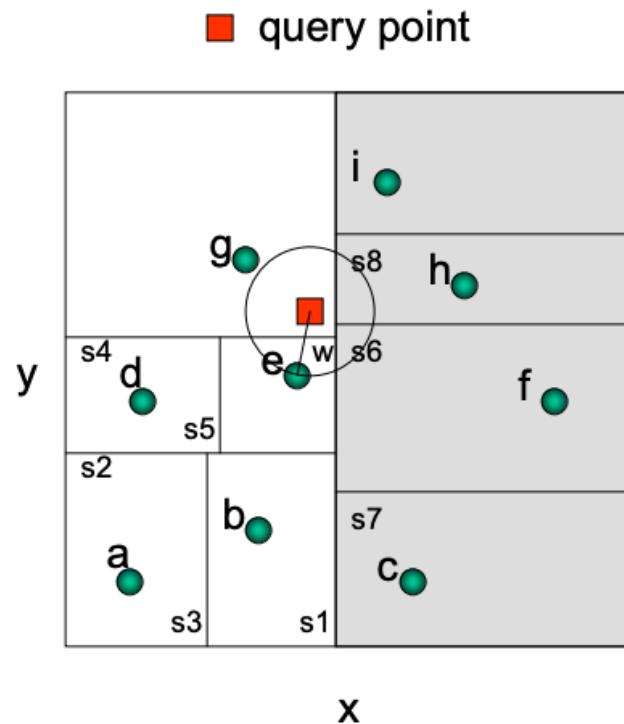


Kd-tree – kNN Search



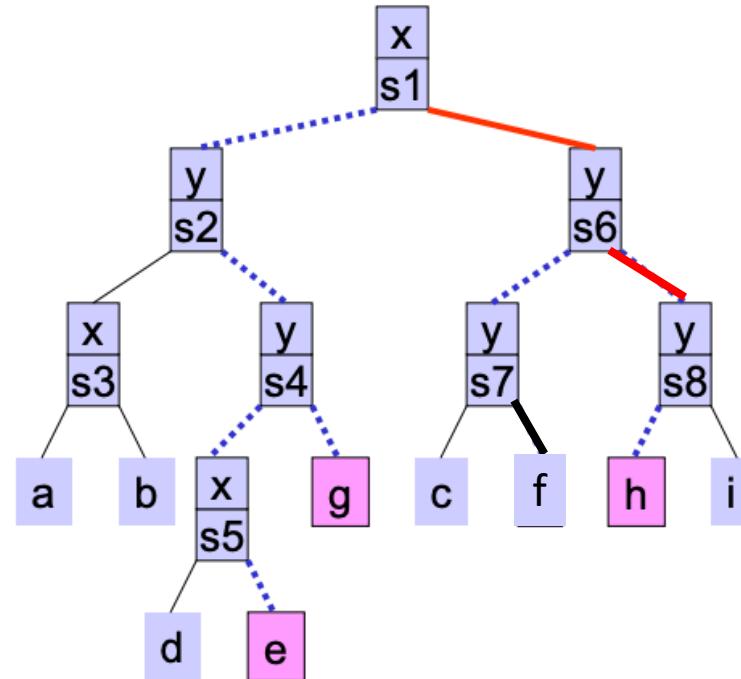
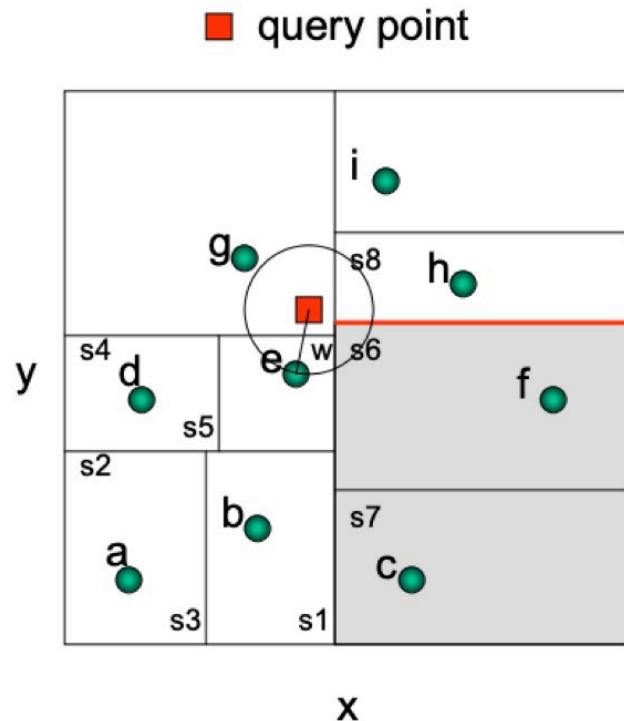


Kd-tree – kNN Search



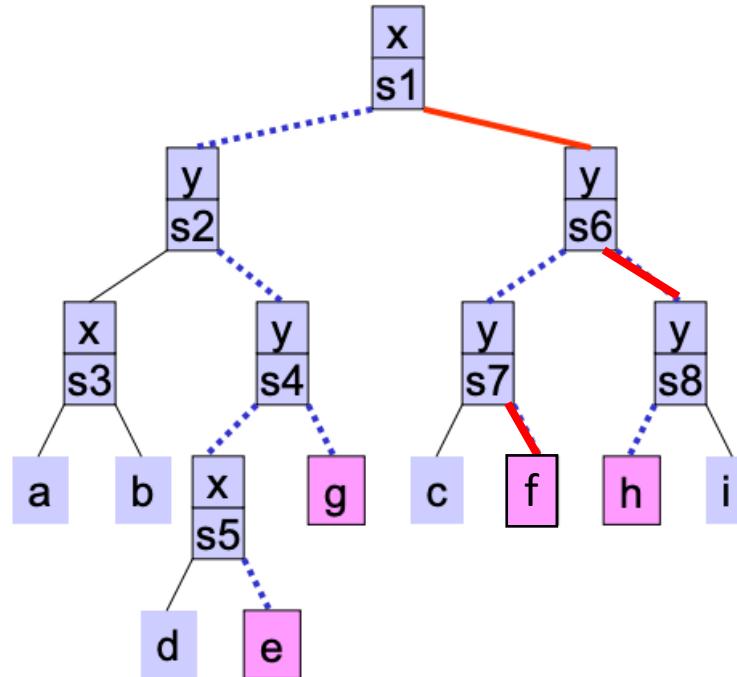
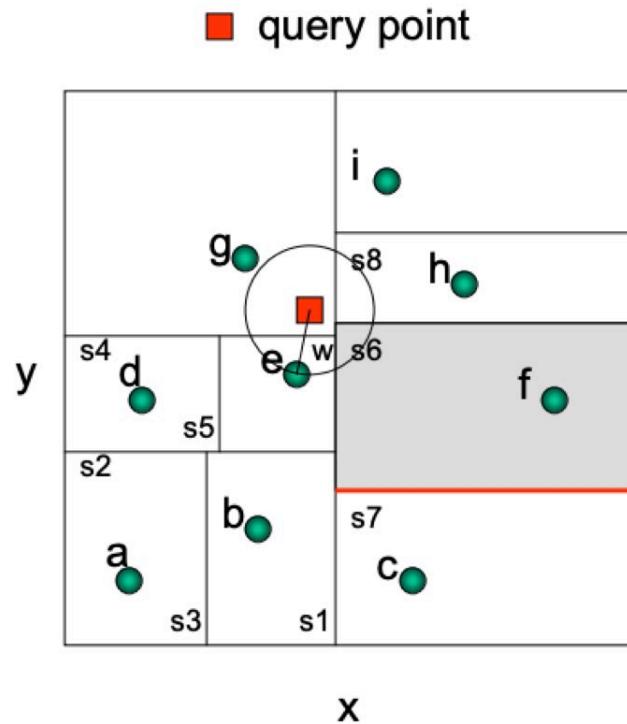


Kd-tree – kNN Search



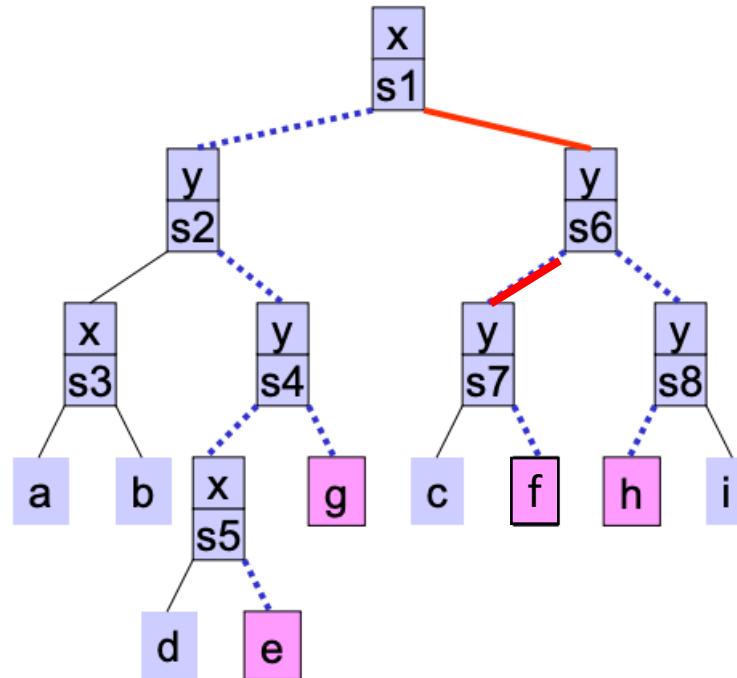
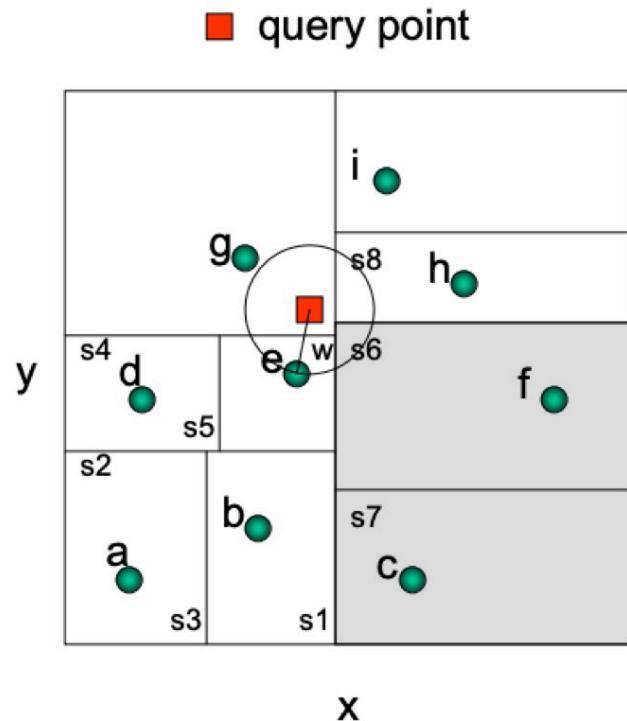


Kd-tree – kNN Search



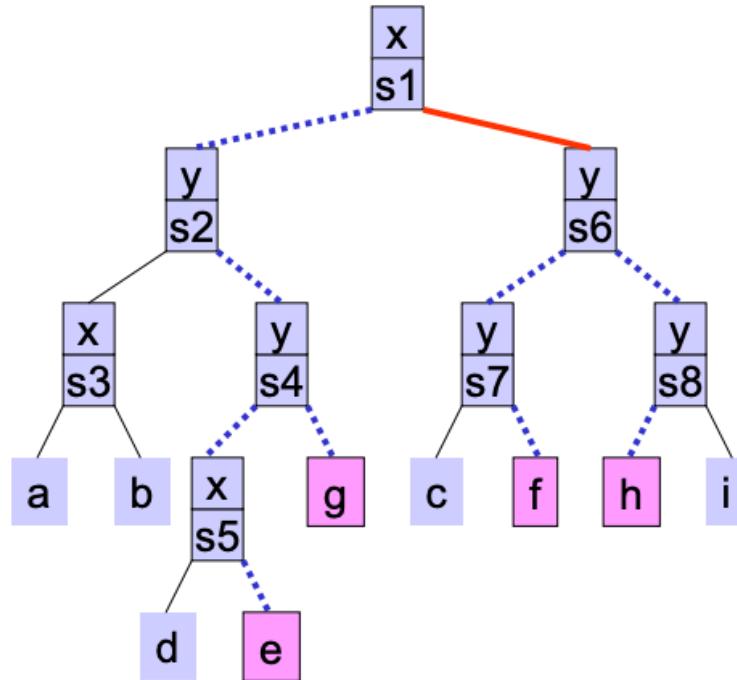
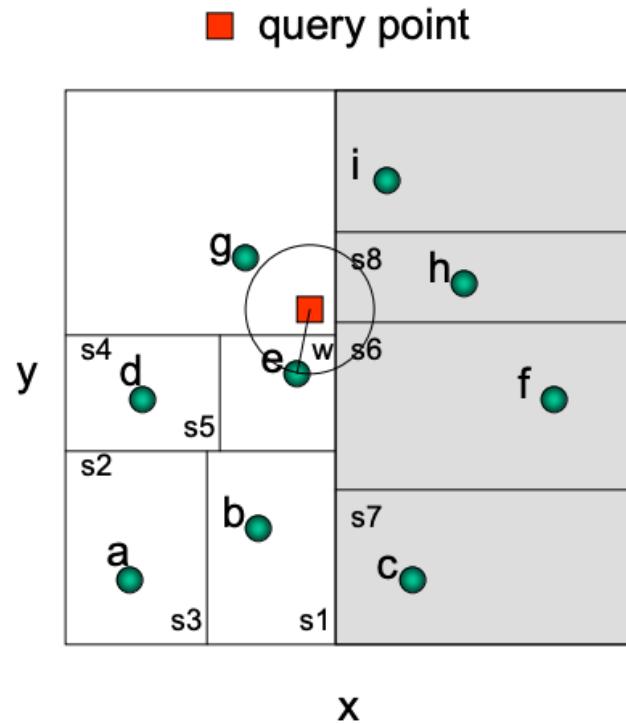


Kd-tree – kNN Search



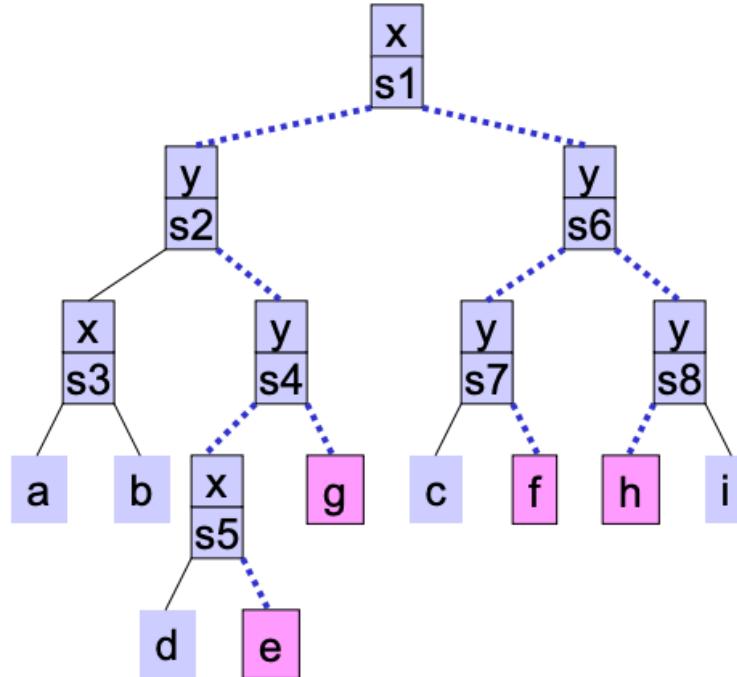
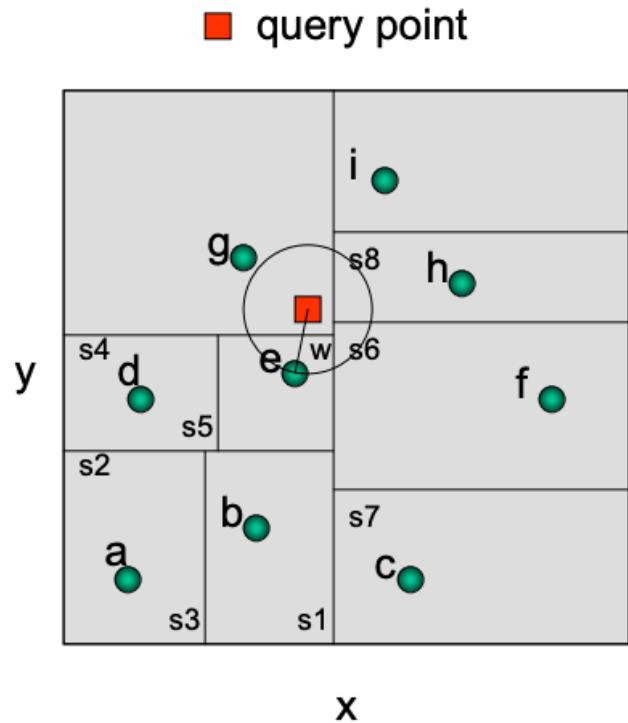


Kd-tree – kNN Search



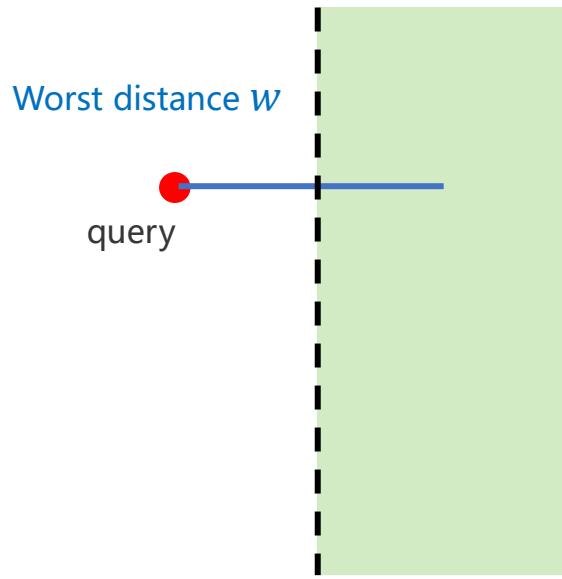


Kd-tree – kNN Search





Kd-tree – kNN Search



splitting_value

Criteria of a partition intersects with the worst-distance range:

$q[\text{axis}]$ inside the **partition**

OR

$|q[\text{axis}] - \text{splitting_value}| < w$

```
def knn_search(root: Node, db: np.ndarray, result_set: KNNResultSet, query: np.ndarray):
    if root is None:
        return False

    if root.is_leaf():          Compare query to every point inside the leaf, put into the result set
        # compare the contents of a leaf
        leaf_points = db[root.point_indices, :]
        diff = np.linalg.norm(np.expand_dims(query, 0) - leaf_points, axis=1)
        for i in range(diff.shape[0]):
            result_set.add_point(diff[i], root.point_indices[i])
        return False

    if query[root.axis] <= root.value:      q[axis] inside the partition
        knn_search(root.left, db, result_set, query)
        if math.fabs(query[root.axis] - root.value) < result_set.worstDist():
            knn_search(root.right, db, result_set, query)

    else:                                |q[axis] – splitting_value| < w
        knn_search(root.right, db, result_set, query)
        if math.fabs(query[root.axis] - root.value) < result_set.worstDist():
            knn_search(root.left, db, result_set, query)

    return False
```



Kd-tree Radius-NN Search

- Exactly the same as kNN search except:

- Use *RadiusNNResultSet*, similar to BST search
- Fixed worst distance, instead of dynamic

```
if query[root.axis] <= root.value:  
    radius_search(root.left, db, result_set, query)  
    if math.fabs(query[root.axis] - root.value) < result_set.worstDist():  
        [radius_search(root.right, db, result_set, query)]  
else:  
    radius_search(root.right, db, result_set, query)  
    if math.fabs(query[root.axis] - root.value) < result_set.worstDist():  
        [radius_search(root.left, db, result_set, query)]  
  
return False
```



Kd-tree Search Complexity

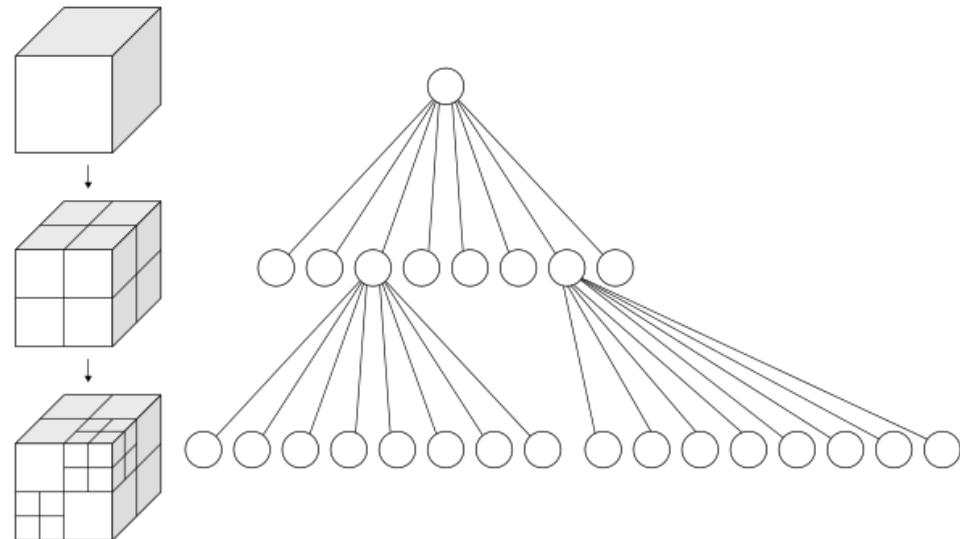
- ➊ 1NN search is $O(\log n)$ for a balanced kd-tree

- ➋ kNN/radiusNN complexity is hard to analyze
 - Depends on the distribution of points
 - Depends on k or r
 - Varies from $O(\log n)$ to $O(n)$



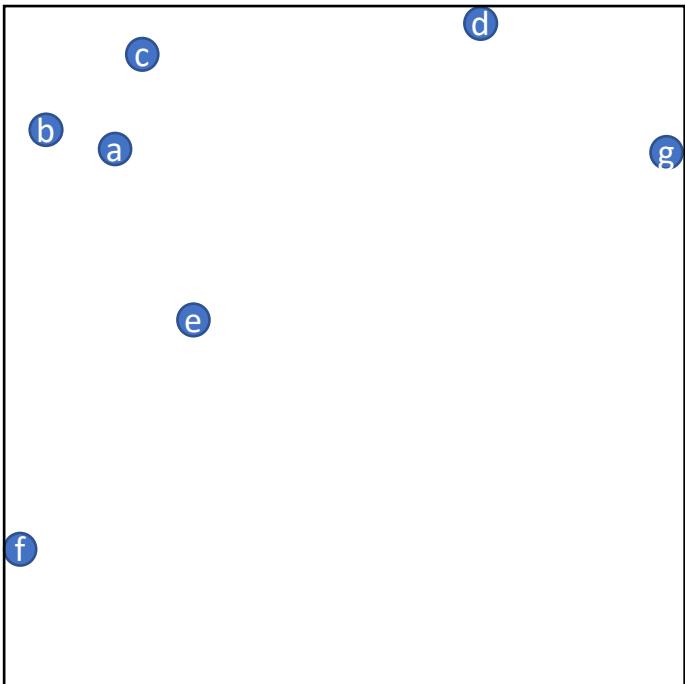
Octree

- ❖ Each node has 8 children
- ❖ oct – tree
- ❖ Specifically for 3D, $2^3=8$
- ❖ In kd-tree, it is non-trivial to determine whether the NN search is done, so we have to go back to root every time
- ❖ Octree is more efficient because we can stop without going back to root





Octree Construction

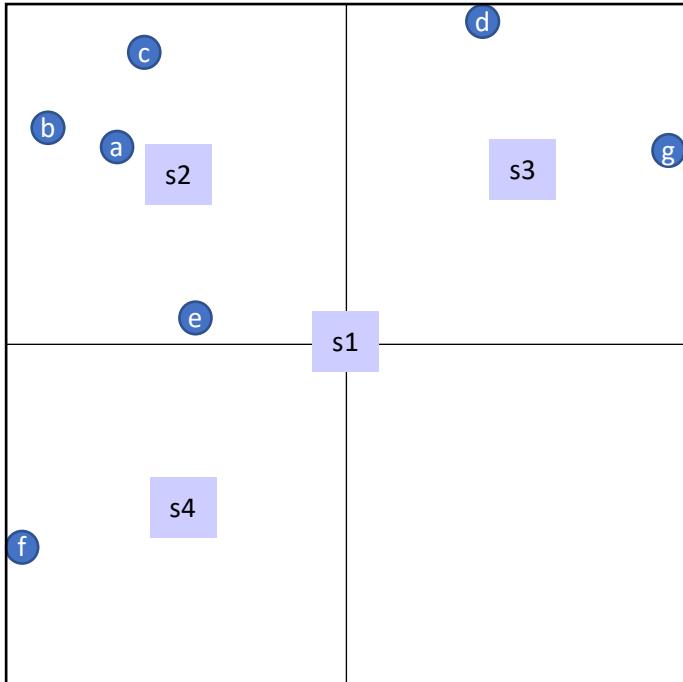


- Determine the extent of the first octant
- Octant is an element in the octree
- Octant is a cube.

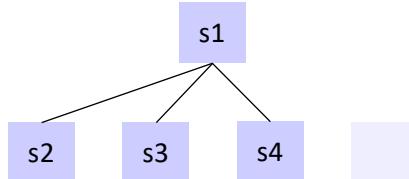
s1



Octree Construction

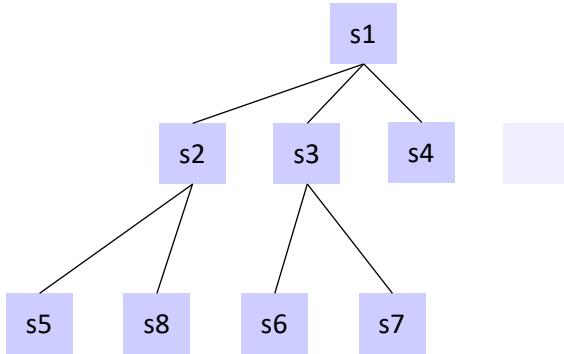
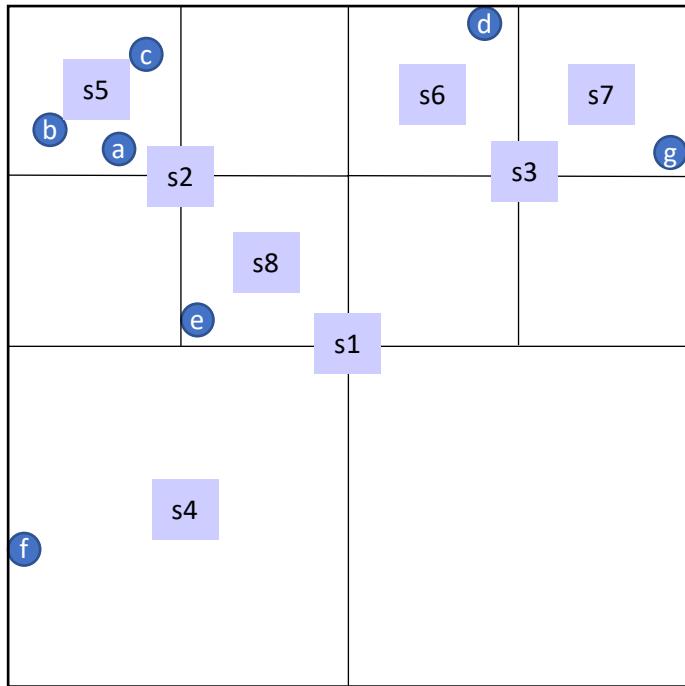


- Determine whether to further split the octant
- `leaf_size` = 1 here
- `min_extent` – avoid infinite splitting when there are repeated points



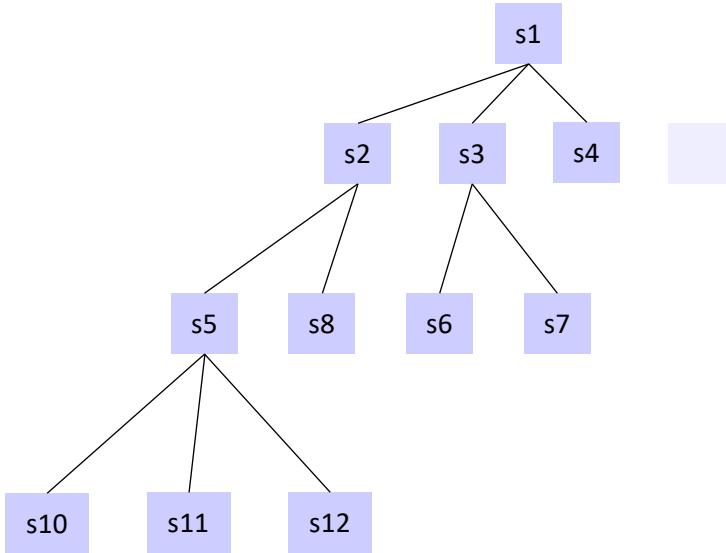
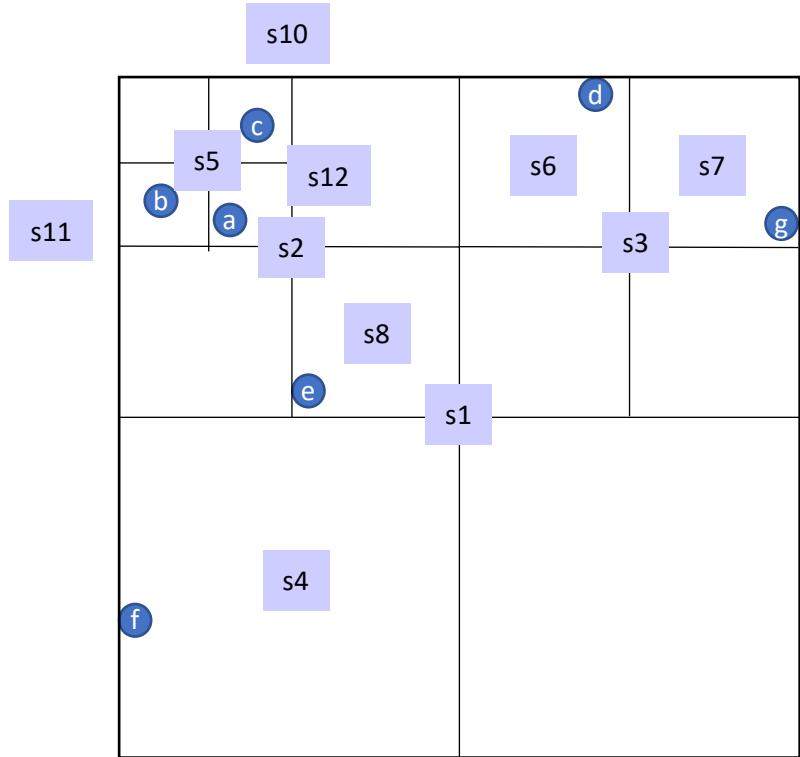


Octree Construction





Octree Construction





Octree Construction

```
class Octant:  
    def __init__(self, children, center, extent, point_indices, is_leaf):  
        self.children = children  
        self.center = center  
        self.extent = extent  
        self.point_indices = point_indices  
        self.is_leaf = is_leaf
```

Annotations pointing to specific variables:

- Array of length 8: points to `point_indices`
- Center of the cube: points to `center`
- Point inside octant: points to `is_leaf`
- 0.5 * length: points to `extent`

```

def octree_recursive_build(root, db, center, extent, point_indices, leaf_size, min_extent):
    if len(point_indices) == 0:
        return None

    if root is None:
        root = Octant([None for i in range(8)], center, extent, point_indices, is_leaf=True)

    # determine whether to split this octant
    if len(point_indices) <= leaf_size or extent <= min_extent:
        root.is_leaf = True
    else:
        root.is_leaf = False
        children_point_indices = [[] for i in range(8)]
        for point_idx in point_indices:
            point_db = db[point_idx]
            morton_code = 0
            if point_db[0] > center[0]:
                morton_code = morton_code | 1
            if point_db[1] > center[1]:
                morton_code = morton_code | 2
            if point_db[2] > center[2]:
                morton_code = morton_code | 4
            children_point_indices[morton_code].append(point_idx)

        # create children
        factor = [-0.5, 0.5]
        for i in range(8):
            child_center_x = center[0] + factor[(i & 1) > 0] * extent
            child_center_y = center[1] + factor[(i & 2) > 0] * extent
            child_center_z = center[2] + factor[(i & 4) > 0] * extent
            child_extent = 0.5 * extent
            child_center = np.asarray([child_center_x, child_center_y, child_center_z])
            root.children[i] = octree_recursive_build(root.children[i],
                                                       db,
                                                       child_center,
                                                       child_extent,
                                                       children_point_indices[i],
                                                       leaf_size,
                                                       min_extent)

    return root

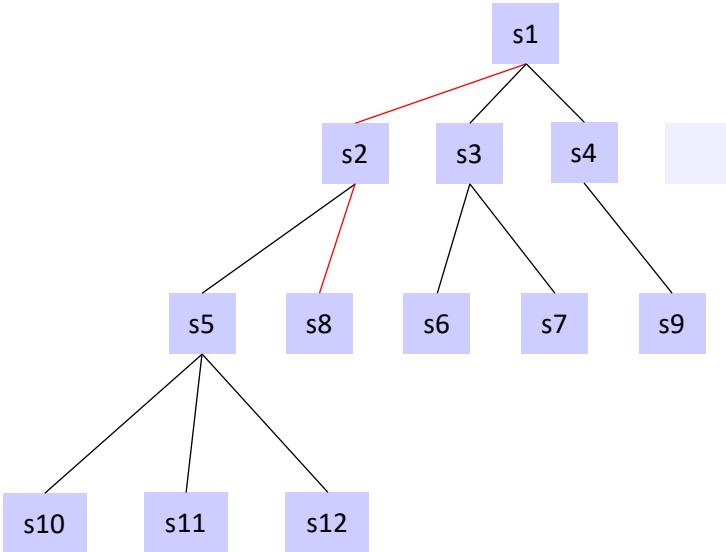
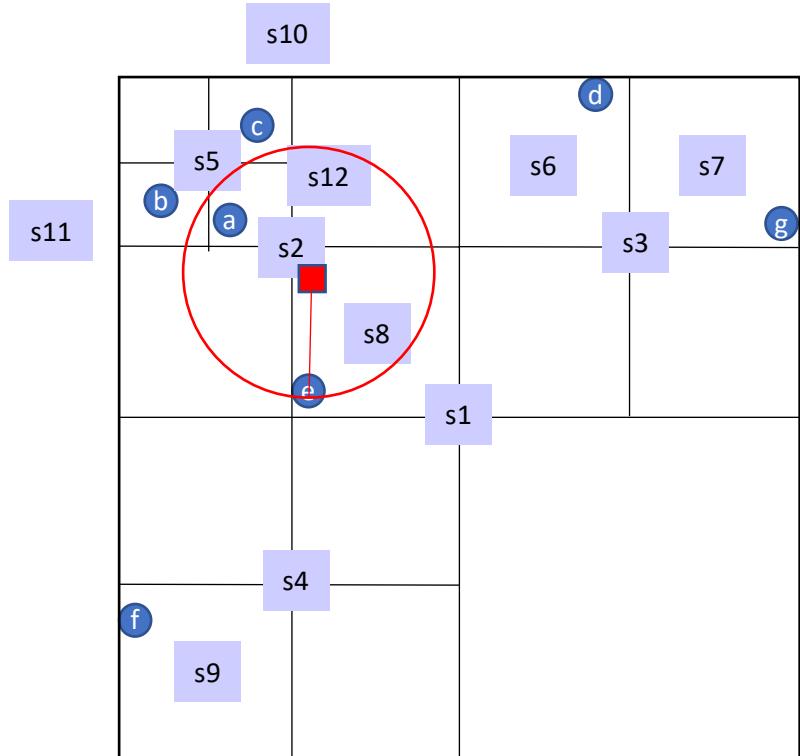
```

Determine which child a point belongs to

Determine child center & extent

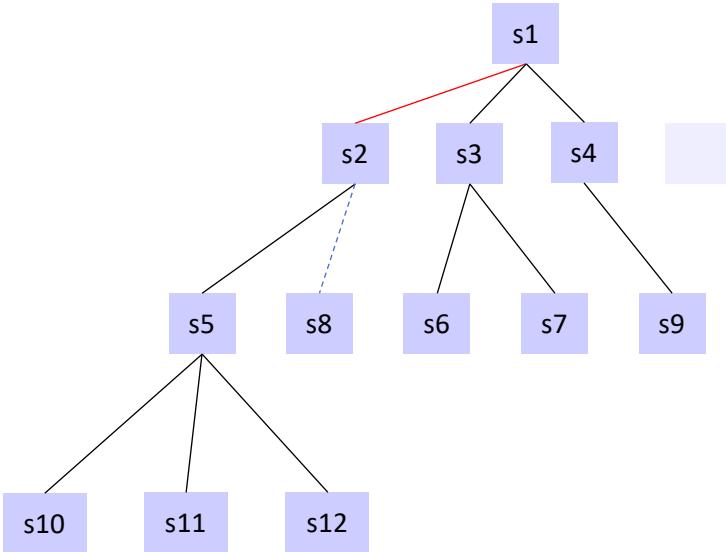
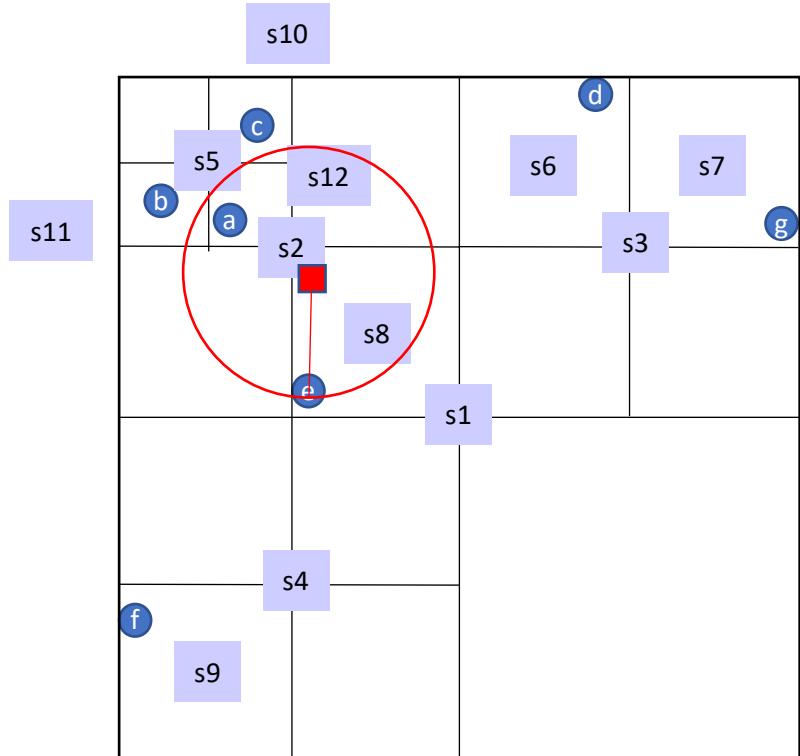


Octree kNN Search



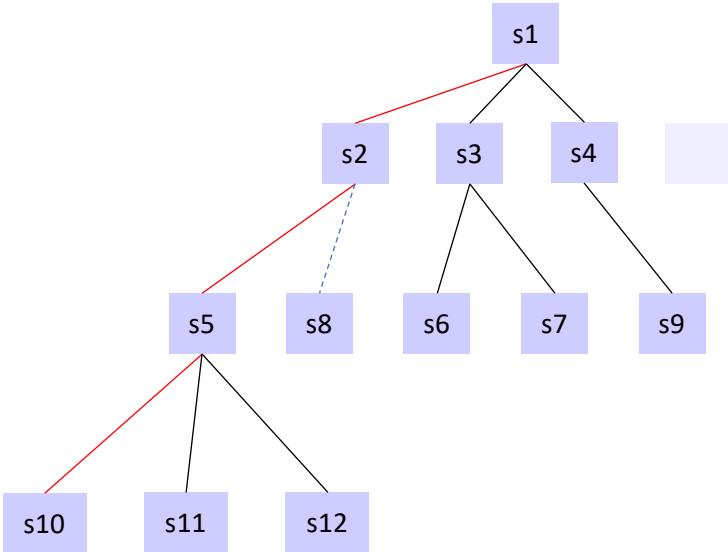
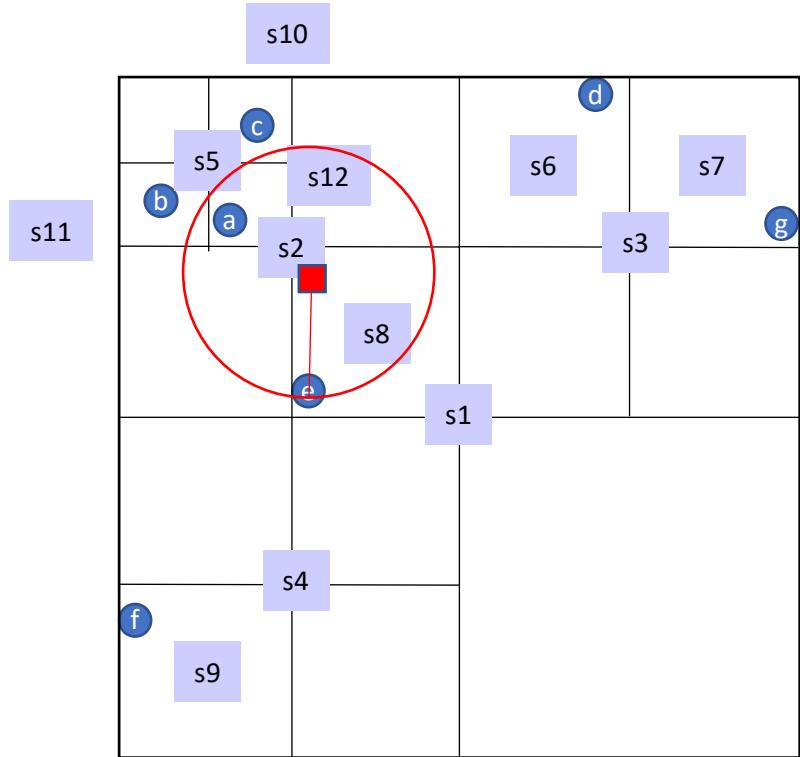


Octree kNN Search



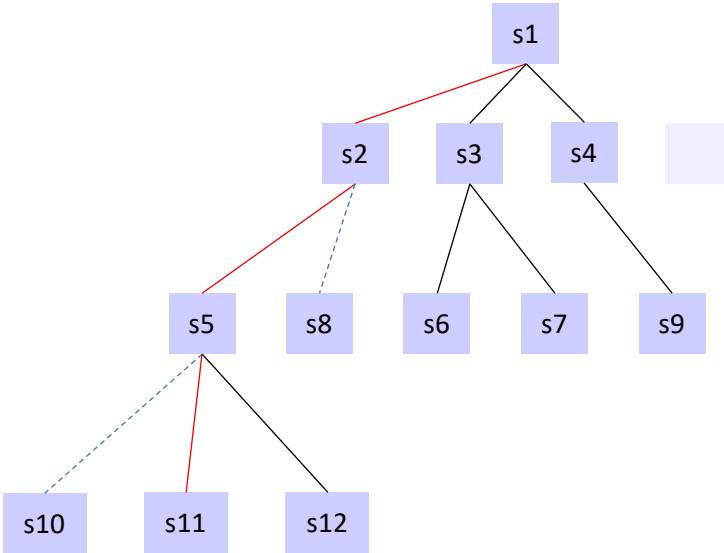
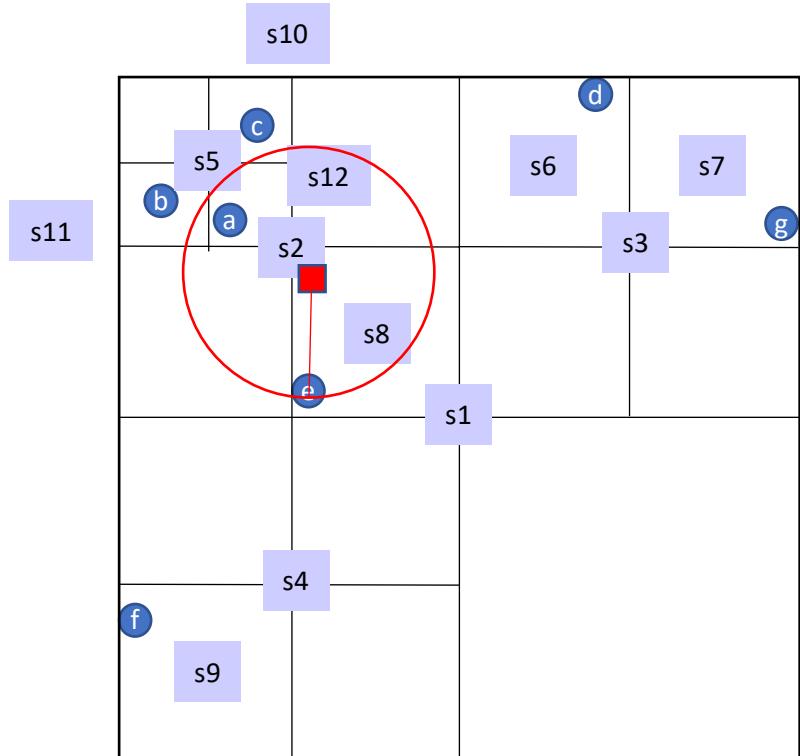


Octree kNN Search



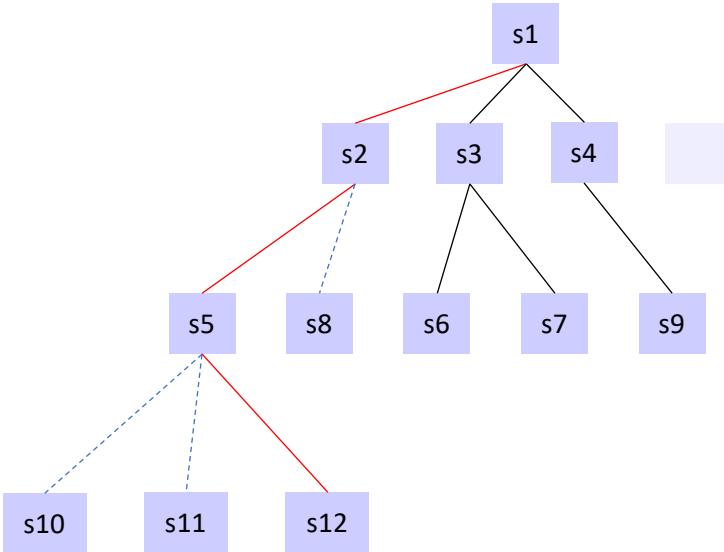
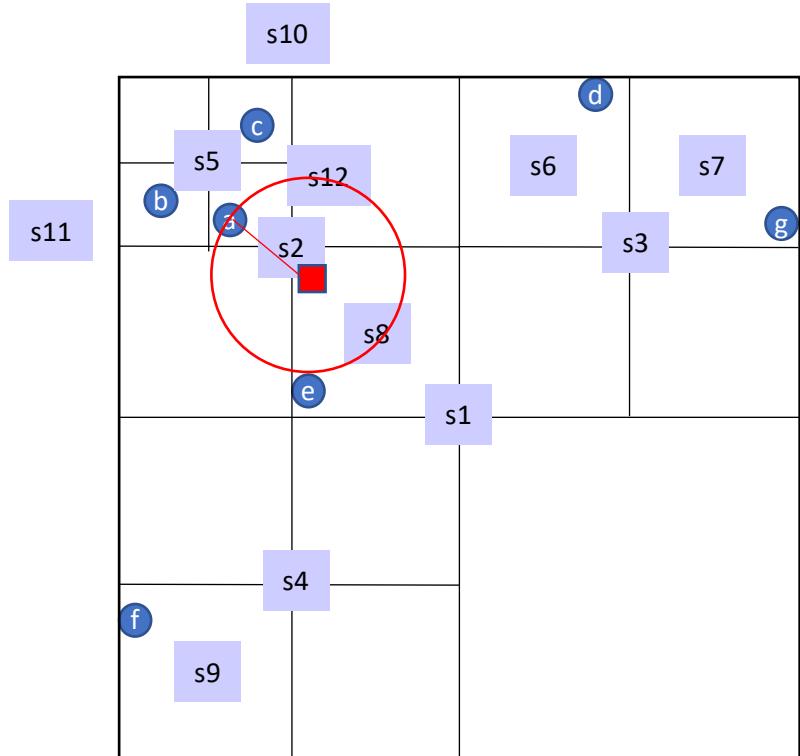


Octree kNN Search



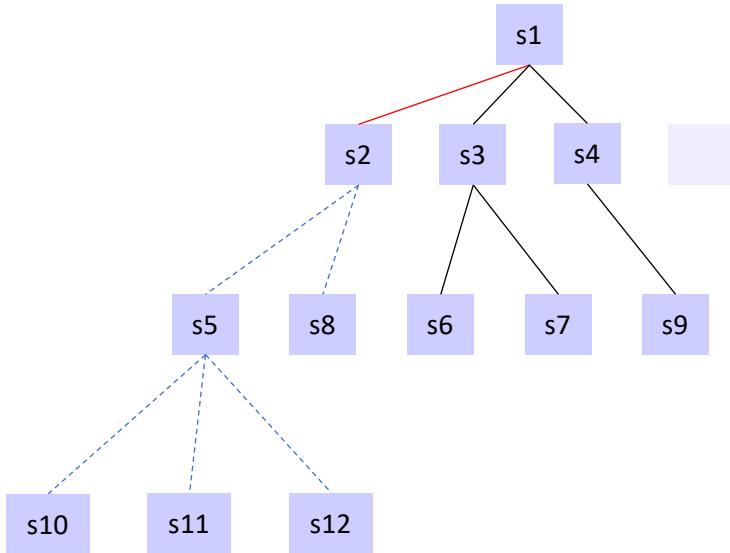
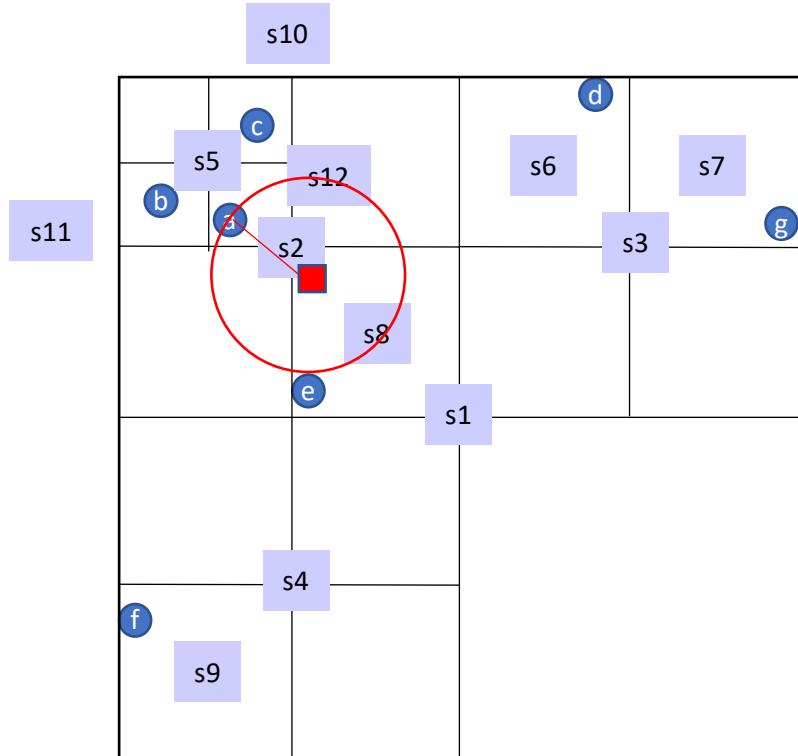


Octree kNN Search





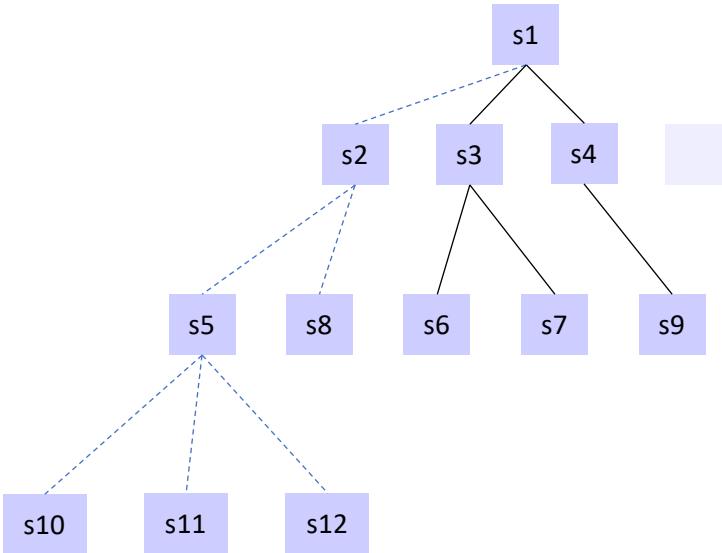
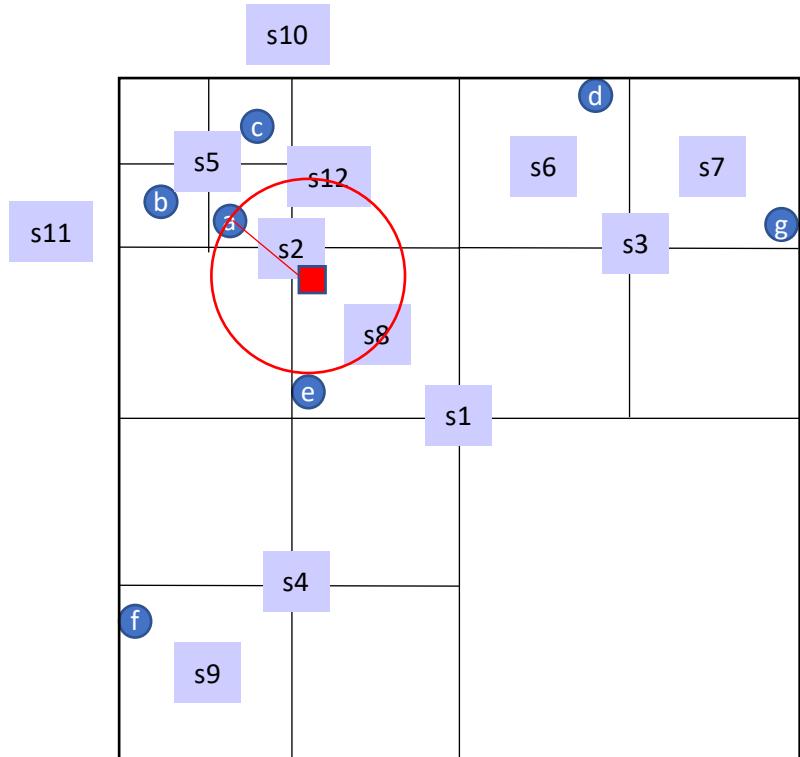
Octree kNN Search





Octree kNN Search

Query ball inside s2, search end!



```
def octree_knn_search(root: Octant, db: np.ndarray, result_set: KNNResultSet, query: np.ndarray):
    if root is None:
        return False

    if root.is_leaf and len(root.point_indices) > 0:
        # compare the contents of a leaf
        leaf_points = db[root.point_indices, :]
        diff = np.linalg.norm(np.expand_dims(query, 0) - leaf_points, axis=1)
        for i in range(diff.shape[0]):
            result_set.add_point(diff[i], root.point_indices[i])
        # check whether we can stop search now
        return inside(query, result_set.worstDist(), root)
```

Compare all points in a leaf

```
# go to the relevant child first
morton_code = 0
if query[0] > root.center[0]:
    morton_code = morton_code | 1
if query[1] > root.center[1]:
    morton_code = morton_code | 2
if query[2] > root.center[2]:
    morton_code = morton_code | 4

if octree_knn_search(root.children[morton_code], db, result_set, query):
    return True
```

Determine & search the most relevant child

```
# check other children
for c, child in enumerate(root.children):
    if c == morton_code or child is None:
        continue
    if False == overlaps(query, result_set.worstDist(), child):
        continue
    if octree_knn_search(child, db, result_set, query):
        return True
# final check of if we can stop search
return inside(query, result_set.worstDist(), root)
```

If an octant is not overlapping with query ball, skip

If query ball is inside an octant, stop



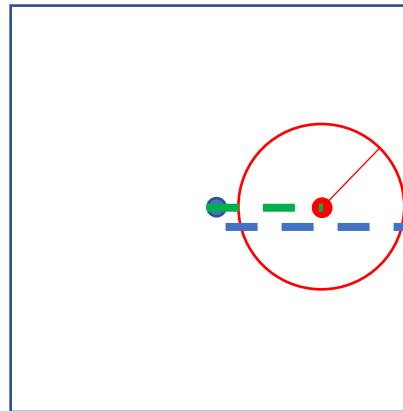
Function *inside*

```
def inside(query: np.ndarray, radius: float, octant:Octant):
    """
    Determines if the query ball is inside the octant
    :param query:
    :param radius:
    :param octant:
    :return:
    """
    query_offset = query - octant.center
    query_offset_abs = np.fabs(query_offset)
    possible_space = query_offset_abs + radius
    return np.all(possible_space < octant.extent)
```

Green dash line

Blue dash line

Red line





Function *overlaps*

```
def overlaps(query: np.ndarray, radius: float, octant:Octant):
    """
    Determines if the query ball overlaps with the octant
    :param query:
    :param radius:
    :param octant:
    :return:
    """
    query_offset = query - octant.center
    query_offset_abs = np.fabs(query_offset)
```

```
# completely outside, since query is outside the relevant area
max_dist = radius + octant.extent
if np.any(query_offset_abs > max_dist):
    return False
```

Case 1

```
# if pass the above check, consider the case that the ball is contacting the face of the octant
if np.sum((query_offset_abs < octant.extent).astype(np.int)) >= 2:
    return True
```

Case 2

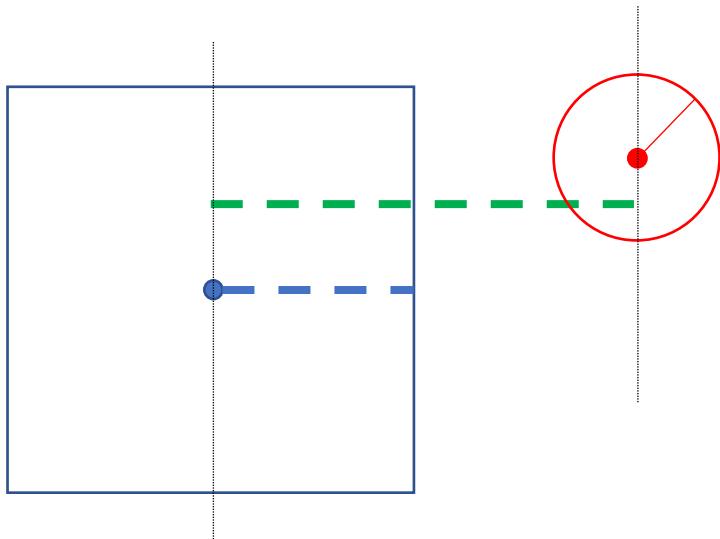
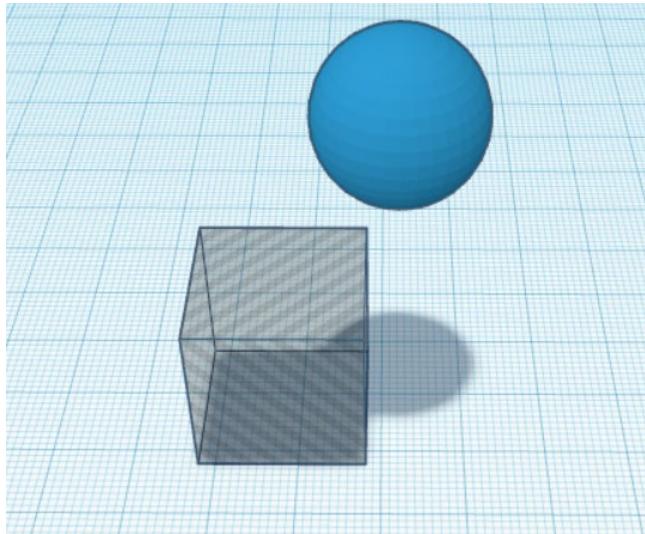
```
# consider the case that the ball is contacting the edge or corner of the octant
# since the case of the ball center (query) inside octant has been considered,
# we only consider the ball center (query) outside octant
x_diff = max(query_offset_abs[0] - octant.extent, 0)
y_diff = max(query_offset_abs[1] - octant.extent, 0)
z_diff = max(query_offset_abs[2] - octant.extent, 0)

return x_diff * x_diff + y_diff * y_diff + z_diff * z_diff < radius * radius
```

Case 3



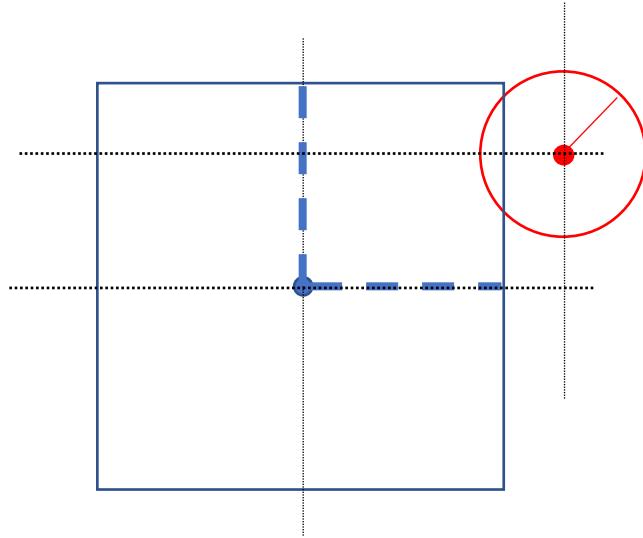
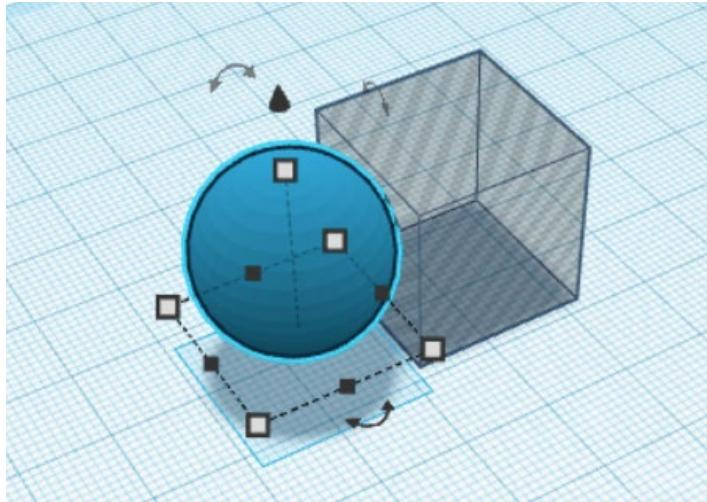
Function *overlaps* – Case 1



```
# completely outside, since query is outside the relevant area
max_dist = radius + octant.extent
if np.any(query_offset_abs > max_dist):
    return False
```



Function *overlaps* – Case 2

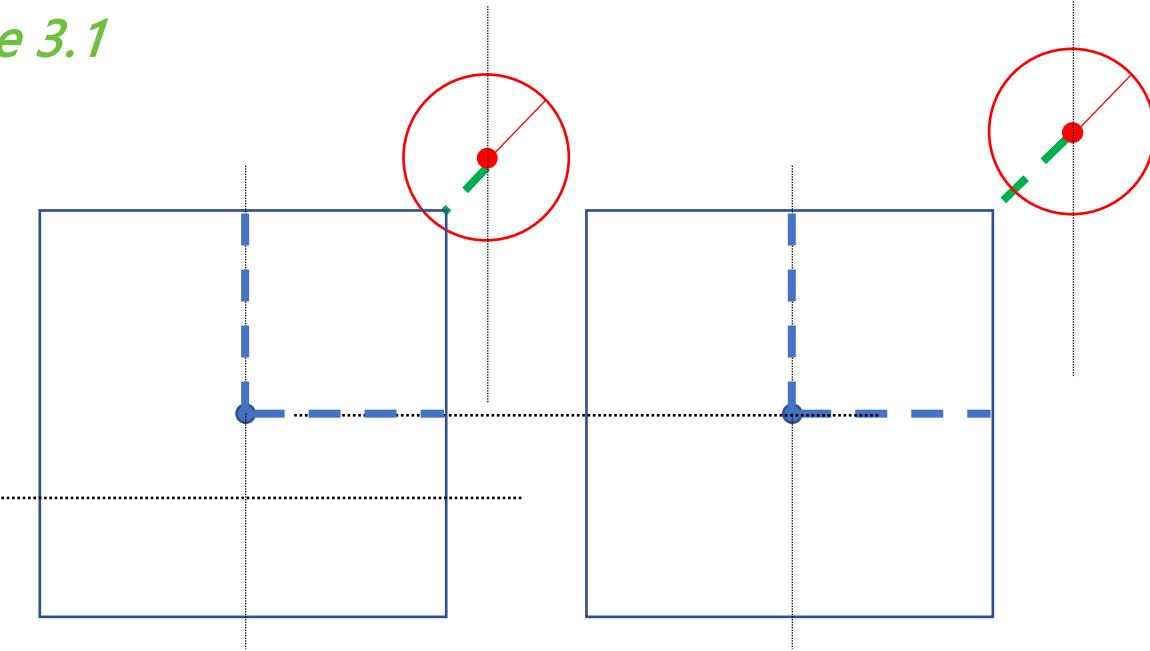
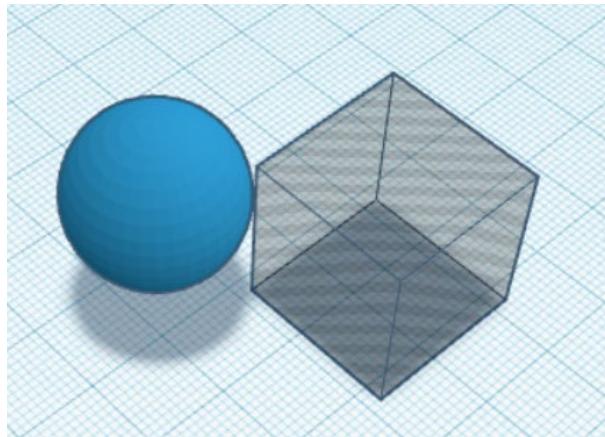


Check if the ball is contacting the face of the octant

```
if np.sum((query_offset_abs < octant.extent).astype(np.int)) >= 2:  
    return True
```



Function *overlaps* – Case 3.1



```
# consider the case that the ball is contacting the edge or corner of the octant
# since the case of the ball center (query) inside octant has been considered,
# we only consider the ball center (query) outside octant
x_diff = max(query_offset_abs[0] - octant.extent, 0)
y_diff = max(query_offset_abs[1] - octant.extent, 0)
z_diff = max(query_offset_abs[2] - octant.extent, 0)

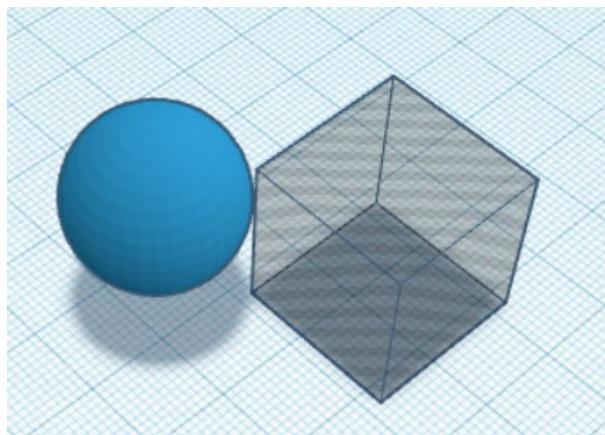
return x_diff * x_diff + y_diff * y_diff + z_diff * z_diff < radius * radius
```



Function *overlaps* – Case 3.2

In 3D, there is the case that the cube's edge cut into the query ball

```
| # consider the case that the ball is contacting the edge or corner of the octant
| # since the case of the ball center (query) inside octant has been considered,
| # we only consider the ball center (query) outside octant
| x_diff = max(query_offset_abs[0] - octant.extent, 0)
| y_diff = max(query_offset_abs[1] - octant.extent, 0)
| z_diff = max(query_offset_abs[2] - octant.extent, 0)
|
| return x_diff * x_diff + y_diff * y_diff + z_diff * z_diff < radius * radius
```



That's why there is a “*max*” to reduce this case into 3.1



Octree Radius NN Search

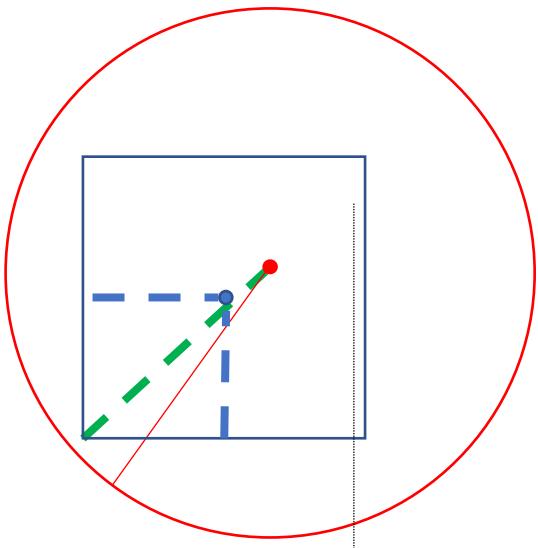
Simple one: replace KNNResultSet with RadiusNNResultSet

Better one:

- If the query ball *contains* the octant, just compare the query with all point
- No need to go into children of that octant



Function *contains*



```
def contains(query: np.ndarray, radius: float, octant:Octant):
    """
    Determine if the query ball contains the octant
    :param query:
    :param radius:
    :param octant:
    :return:
    """

    query_offset = query - octant.center
    query_offset_abs = np.fabs(query_offset)

    query_offset_to_farthest_corner = query_offset_abs + octant.extent
    return np.linalg.norm(query_offset_to_farthest_corner) < radius
```

Green dash line

Red line



Octree Search Complexity

- ➊ 1NN search is $O(\log n)$
- ➋ kNN/radiusNN complexity is hard to analyze
 - Depends on the distribution of points
 - Depends on k or r
 - Varies from $O(\log n)$ to $O(n)$

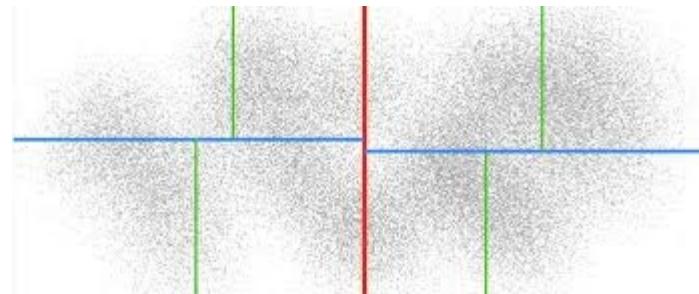
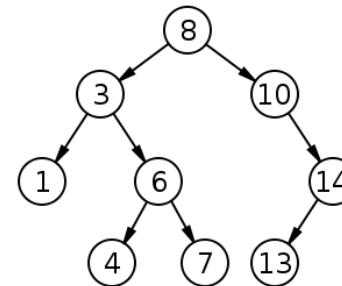


BST / Kd-tree / Octree



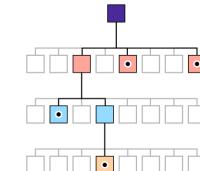
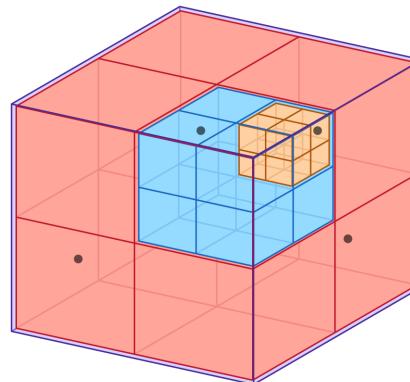
Dimension

- BST for one dimension
- Kd-tree works for any dimension
- Octree is optimized for 3D



Idea

- Same – space partition





Summary

-  Space partition
-  Find a method to skip some partitions
-  Pythons codes:
<https://github.com/lijx10/NN-Trees>



Homework

- We provide one $N \times 3$ point cloud
- 8-NN search for each point to the point cloud
- Implement 3 NN algorithms
 1. Numpy brute-force search
 2. `scipy.spatial.KDTree`
 3. Your own kd-tree/octree in python or C++
- Report timing using method 1 as baseline
- This is a competition!
 - Timing of method 3 determine your grade