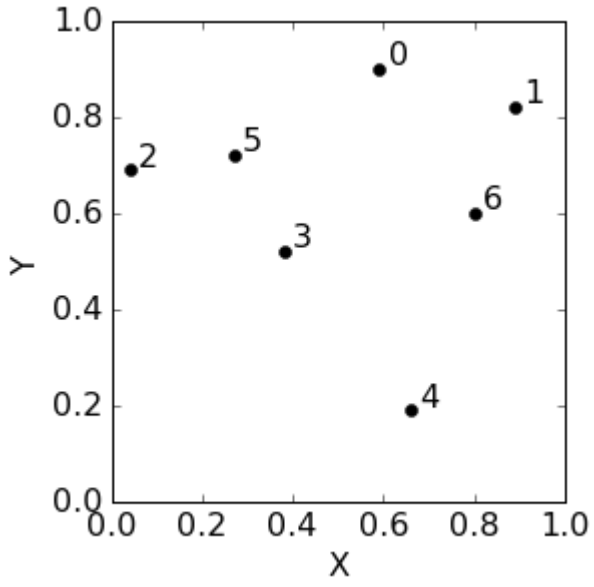
A worked-out example for KD-trees

In applications where the number of features is not too high, KD-trees enable efficient queries for nearest neighbor search.

Dataset

Let us consider a toy example with six data points in 2D:

	X	Y
Data point 0	0.59	0.9
Data point 1	0.89	0.82
Data point 2	0.04	0.69
Data point 3	0.38	0.52
Data point 4	0.66	0.19
Data point 5	0.27	0.72
Data point 6	0.8	0.6



KD-tree construction: first split

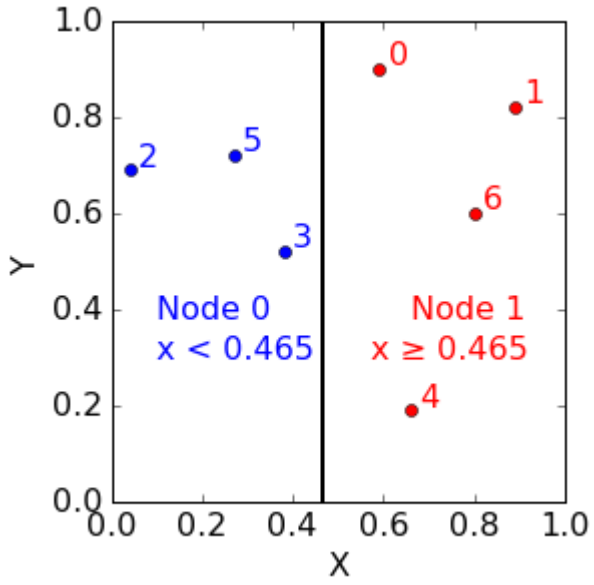
Let's split data into two groups. We do so by comparing the X coordinate of each data point against 0.465. (We'll soon get back to how we got to choose this point of reference, in the next section.)

	X	Y	**X vs 0.465 ?**
Data point 0	0.59	0.9	**>= 0.465**
Data point 1	0.89	0.82	**>= 0.465**
Data point 2	0.04	0.69	**< 0.465**
Data point 3	0.38	0.52	**< 0.465**
Data point 4	0.66	0.19	**>= 0.465**
Data point 5	0.27	0.72	**< 0.465**
Data point 6	0.8	0.6	**>= 0.465**

The data points whose X coordinate is less than 0.465 are grouped together in one node; the remaining data points are grouped together in another.

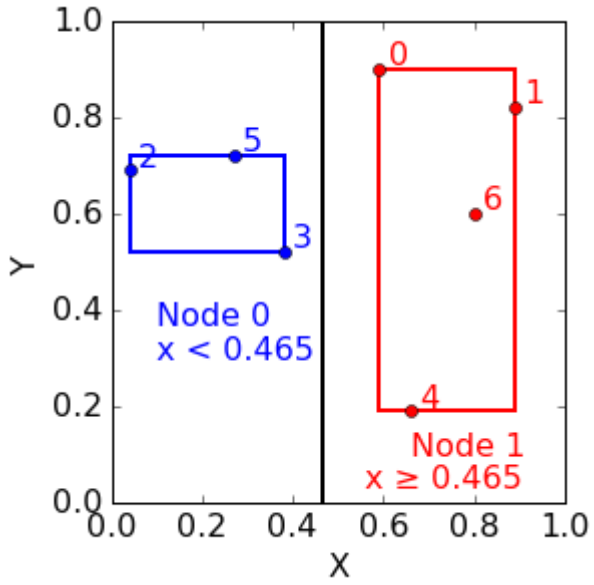
Node 0	X	Y
Data point 2	0.04	0.69
Data point 3	0.38	0.52
Data point 5	0.27	0.72

Node 1	X	Y
Data point 0	0.59	0.9
Data point 1	0.89	0.82
Data point 4	0.66	0.19
Data point 6	0.8	0.6

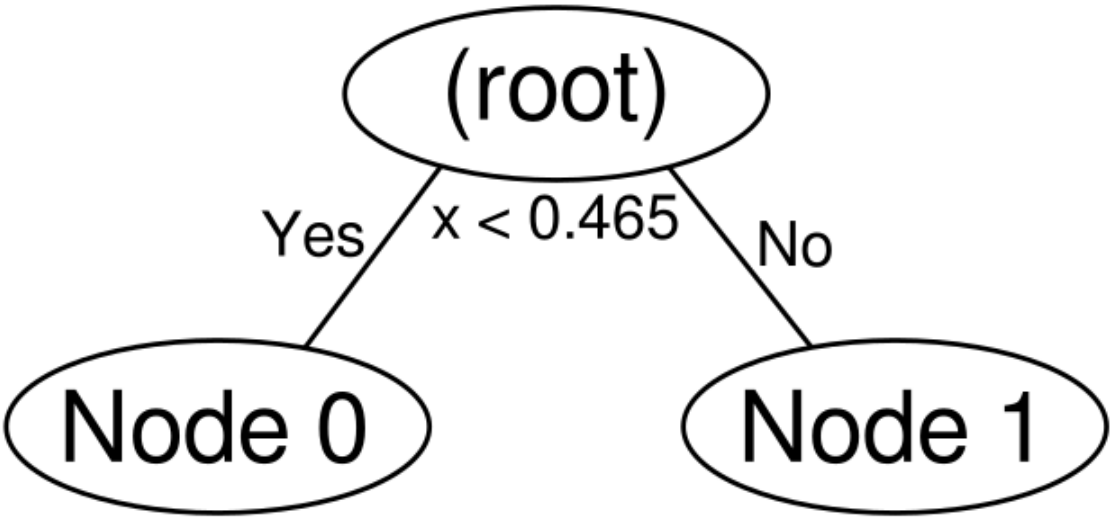


At each node, we keep one additional piece of information: the tight coordinate bounds of the points inside the node. This information will be useful in performing nearest neighbor search.

Tight bounds	X	Y
Node 0	**0.04 <= X <= 0.38**	**0.52 <= Y <= 0.72**
Node 1	**0.59 <= X <= 0.89**	**0.19 <= Y <= 0.9**



We encode the hierarchy among the nodes as a binary tree:



KD-tree construction: recursive split

After the first split, we further subdivide each node in recursive fashion. There are some decisions we have to make each time we split a node. **Note that there are many possible heuristics when it comes to splitting decisions; we choose a particular set for the sake of illustration.**

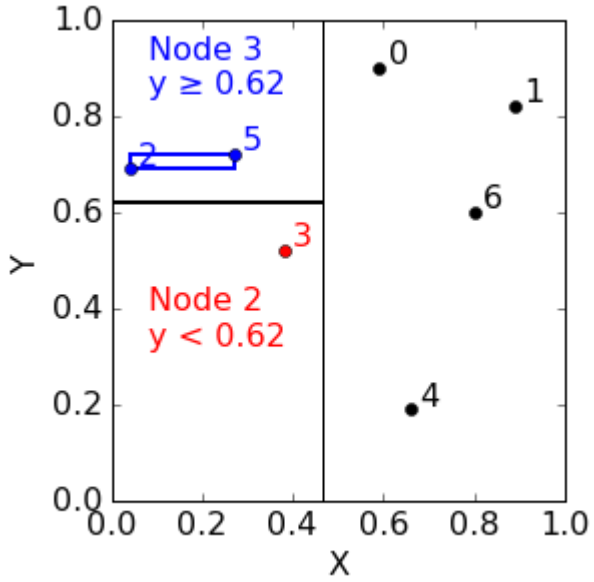
- **Which dimension (axis) do we split along?** For this example, **we alternate between the X axis and the Y axis.** This is why the first split used the X coordinates.
- **Which value do we split at? Let's split at the midpoint between the smallest and largest coordinates among the member points.** In the first split, for instance, we took the midpoint in the smallest and largest X coordinates, giving 0.465. Notice we say "member points": when it comes to subsequent splits, we consider only those points that are inside the node being split. (Note: some heuristics may use dimensions of the box instead of the member points.)
- **When do we stop? Let's stop when the node contains two nodes.**

With this set of heuristics, let's first subdivide Node 0. The node will be divided into two nodes, Nodes 2 and 3, as follows:

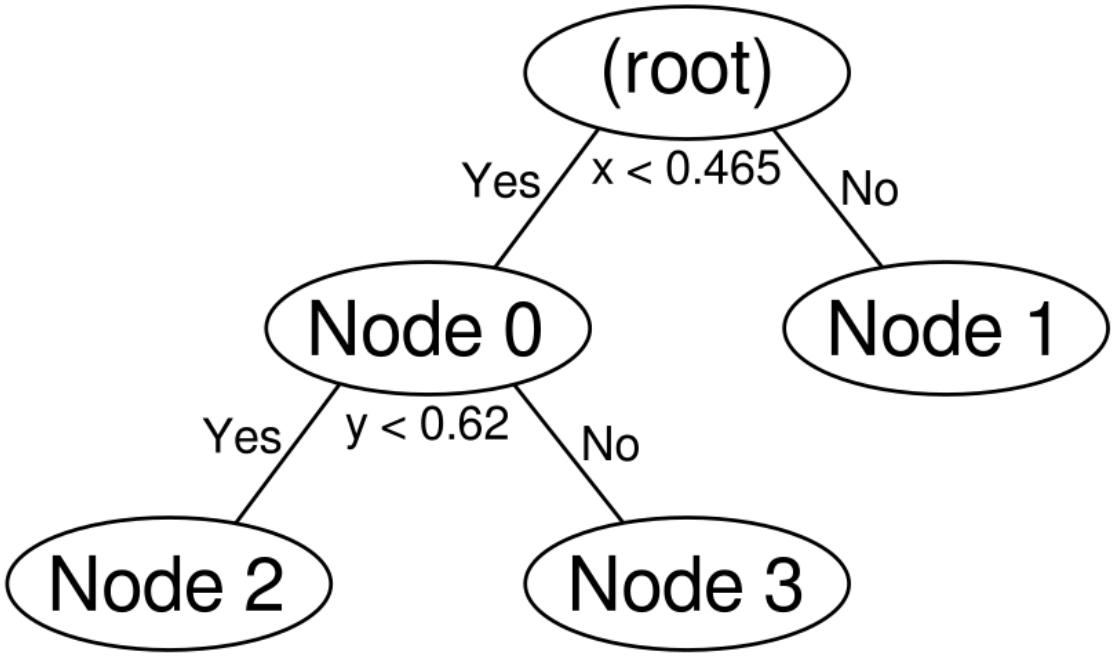
- Since the first split used the X axis, use the Y axis.
- Take the midpoint of the Y coordinates of the member points, i.e. data points 2, 3, and 5. The midpoint is (0.52+0.72)/2 = 0.62.
- The points whose Y coordinate is less than 0.62 go into Node 2; the others go into Node 3.
- Compute the tight bounds for Nodes 2 and 3.

Node 0	X	**Y**	**Y vs 0.62 ?**	**Which node?**
Data point 2	0.04	**0.69**	**>= 0.62**	**To Node 3**
Data point 3	0.38	**0.52**	**< 0.62**	**To Node 2**
Data point 5	0.27	**0.72**	**>= 0.62**	**To Node 3**

Tight bounds	X	Y
Node 0	0.04 <= X <= 0.38	0.52 <= Y <= 0.72
Node 1	0.59 <= X <= 0.89	0.19 <= Y <= 0.9
Node 2	**0.38 <= X <= 0.38**	**0.52 <= Y <= 0.52**
Node 3	**0.04 <= X <= 0.27**	**0.69 <= Y <= 0.72**



Note that Nodes 2 and 3 have much smaller boxes than Node 0. Tighter bounds would help speed up nearest neighbor search. The hierarchy has now two levels, with a new subtree added to the left:



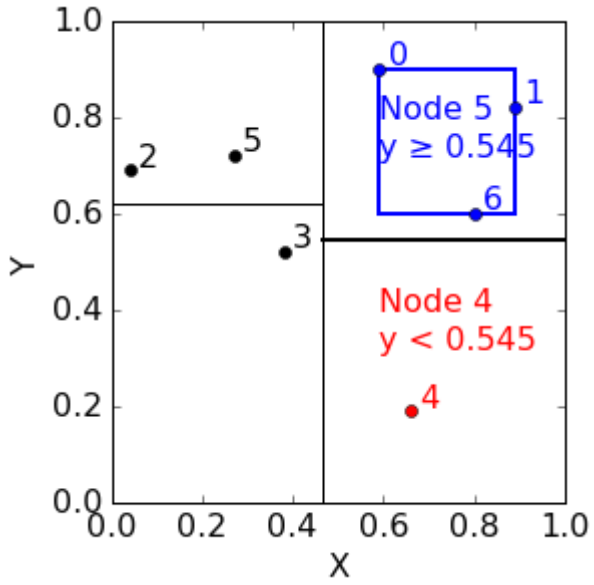
It only remains to split Node 1, as Nodes 2 and 3 have fewer than 2 points each. Node 1 will be divided into two nodes, Nodes 4 and 5, as follows:

- Since the first split used the X axis, use the Y axis.
- Take the midpoint of the Y coordinates of the member points, i.e. data points 0, 1, 4, and 6. The midpoint is (0.19+0.9)/2 = 0.545.
- The points whose Y coordinate is less than 0.545 go into Node 4; the others go into Node 5.
- Compute the tight bounds for Nodes 4 and 5.

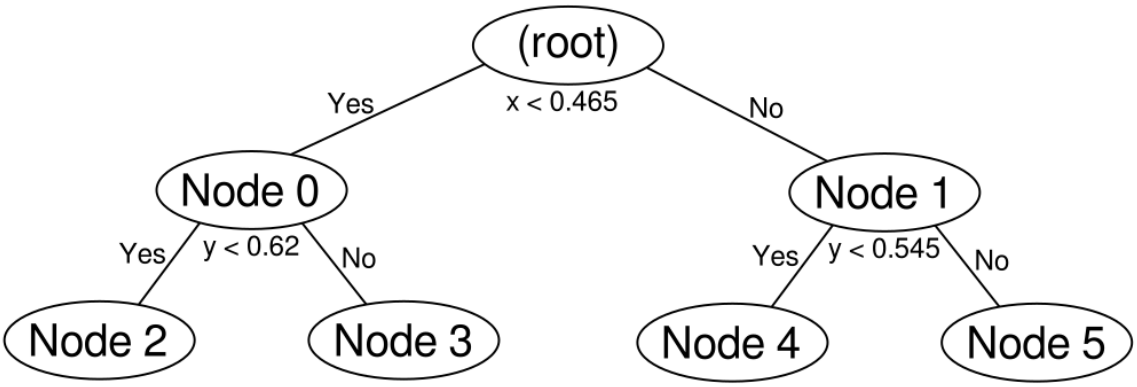
Node 1	X	**Y**	**Y vs 0.545 ?**	**Which node?**

Data point 0	0.59	**0.9**	**>= 0.545**	**To Node 5**
Data point 1	0.89	**0.82**	**>= 0.545**	**To Node 5**
Data point 4	0.66	**0.19**	**< 0.545**	**To Node 4**
Data point 6	0.8	**0.6**	**>= 0.545**	**To Node 5**

Tight bounds	X	Y
Node 0	0.04 <= X <= 0.38	0.52 <= Y <= 0.72
Node 1	0.59 <= X <= 0.89	0.19 <= Y <= 0.9
Node 2	0.38 <= X <= 0.38	0.52 <= Y <= 0.52
Node 3	0.04 <= X <= 0.27	0.69 <= Y <= 0.72
Node 4	**0.66 <= X <= 0.66**	**0.19 <= Y <= 0.19**
Node 5	**0.59 <= X <= 0.89**	**0.6 <= Y <= 0.9**



The node hierarchy gets a new subtree added to the right:



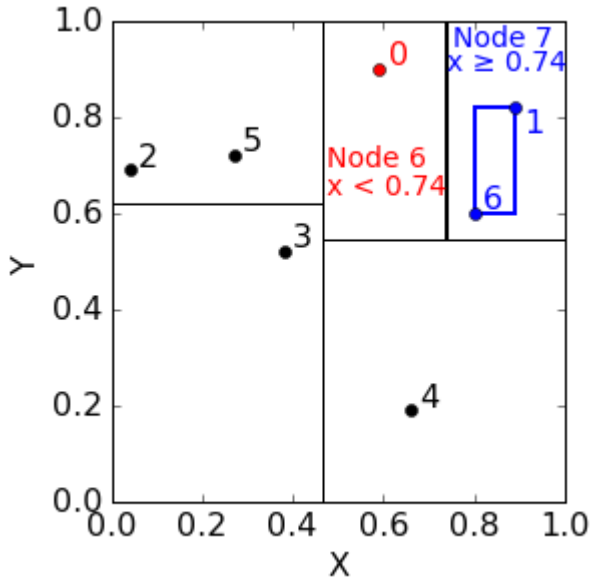
Finally, we split Node 5, as it's still got 3 data points.

- Since we used the Y axis for the previous split, let's use the X axis this time.

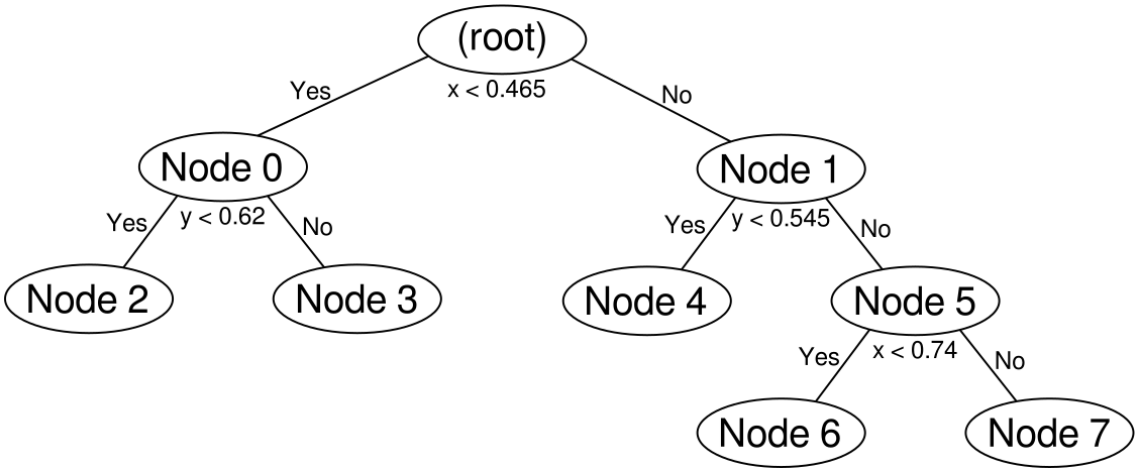
- Take the midpoint of the X coordinates of the member points, i.e. data points 0, 1, and 6. The midpoint is (0.59+0.89)/2 = 0.74.
- The points whose Y coordinate is less than 0.74 go into Node 6; the others go into Node 7.
- Compute the tight bounds for Nodes 6 and 7.

Node 5	**X**	Y	**X vs 0.74 ?**	**Which node?**
Data point 0	**0.59**	0.9	**< 0.74**	**To Node 6**
Data point 1	**0.89**	0.82	**>= 0.74**	**To Node 7**
Data point 6	**0.8**	0.6	**>= 0.74**	**To Node 7**

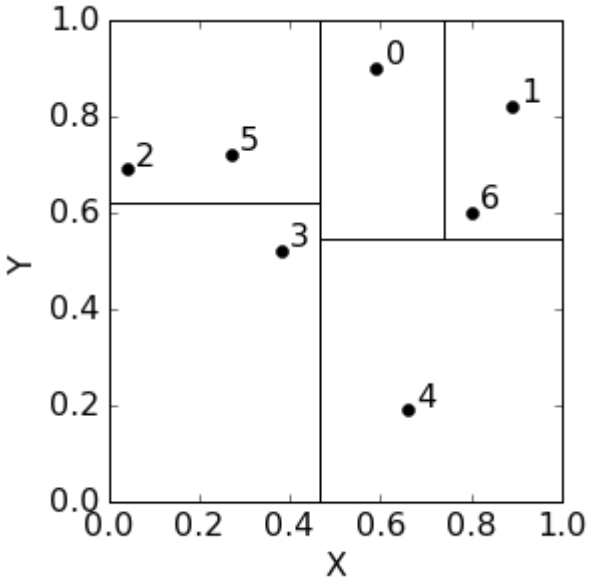
Tight bounds	X	Y
Node 0	0.04 <= X <= 0.38	0.52 <= Y <= 0.72
Node 1	0.59 <= X <= 0.89	0.19 <= Y <= 0.9
Node 2	0.38 <= X <= 0.38	0.52 <= Y <= 0.52
Node 3	0.04 <= X <= 0.27	0.69 <= Y <= 0.72
Node 4	0.66 <= X <= 0.66	0.19 <= Y <= 0.19
Node 5	0.59 <= X <= 0.89	0.6 <= Y <= 0.9
Node 6	**0.59 <= X <= 0.59**	**0.9 <= Y <= 0.9**
Node 7	**0.8 <= X <= 0.89**	**0.6 <= Y <= 0.82**



The hierarchy of nodes has now three levels, with a new subtree in the very right:

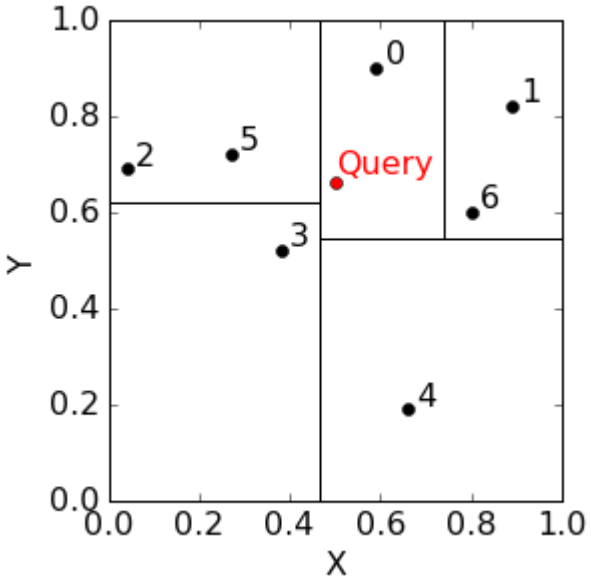


Now that each node contains at most two data points, we're done.



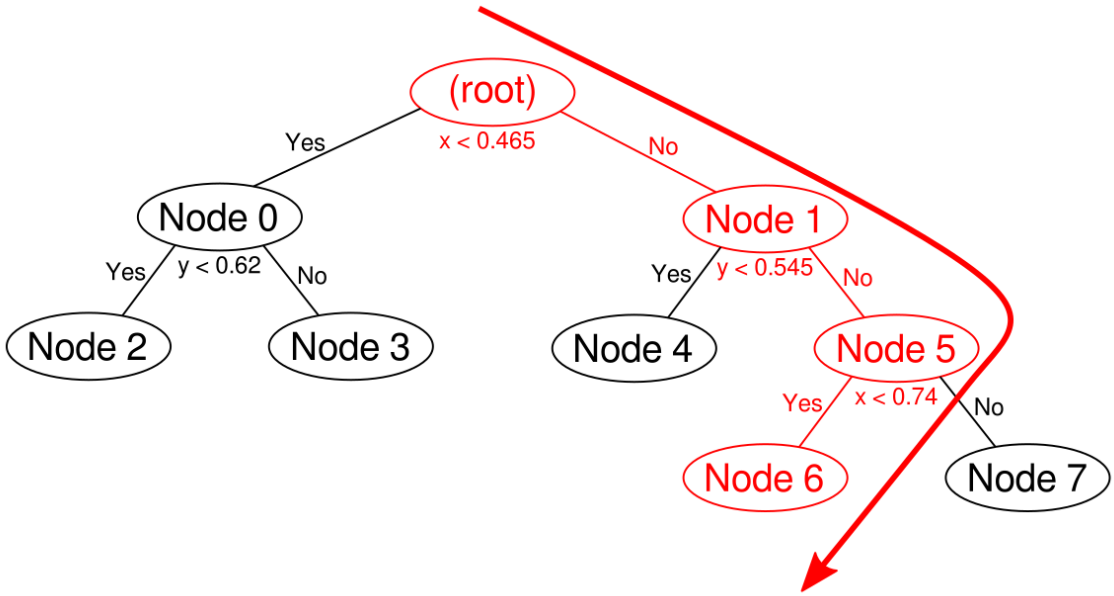
Querying the KD-tree

Suppose we'd like to find the nearest neighbor of the **query point (0.5, 0.66)**.



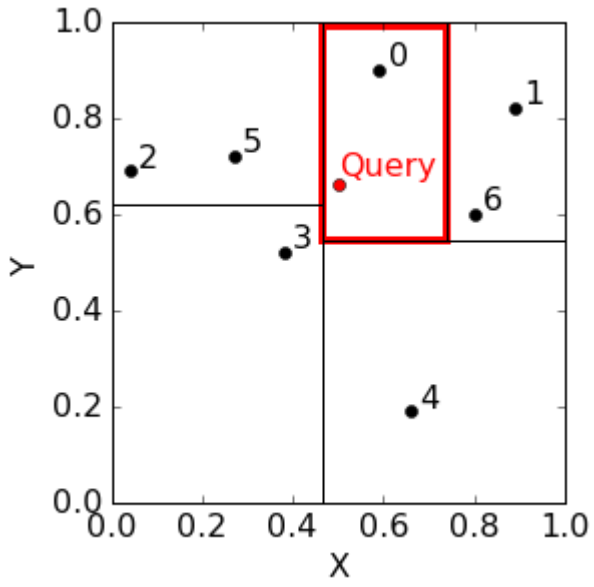
By visual inspection, we see that the query point falls inside Node 6. Alternatively, we can locate the query point by **traversing down** the hierarchy of nodes.

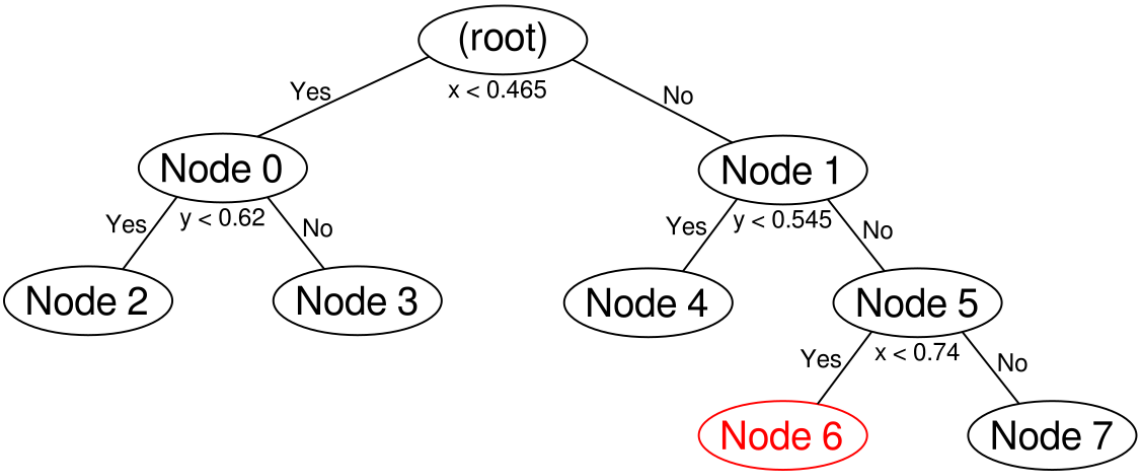
- X < 0.465 ? If Yes, go to Node 0; otherwise, go to Node 1 => **0.5 >= 0.465, so go to Node 1.**
- Y < 0.545 ? If Yes, go to Node 4; otherwise, go to Node 5 => **0.66 >= 0.545, so go to Node 5**.
- X < 0.74 ? If Yes, go to Node 6; otherwise, go to Node 7 => **0.5 < 0.74, so go to Node 6.**



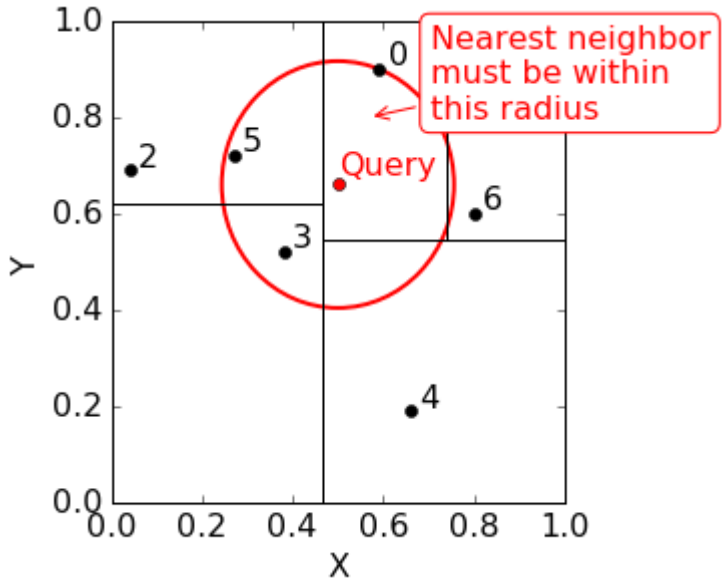
Once the query point is located in a **leaf node** (i.e. bottom-most), we perform **backtracking**: starting from the leaf node, we traverse up, updating our current guess for the nearest neighbor as we go.

Let's start with the leaf node, Node 6. It contains one data point, namely data point 0.

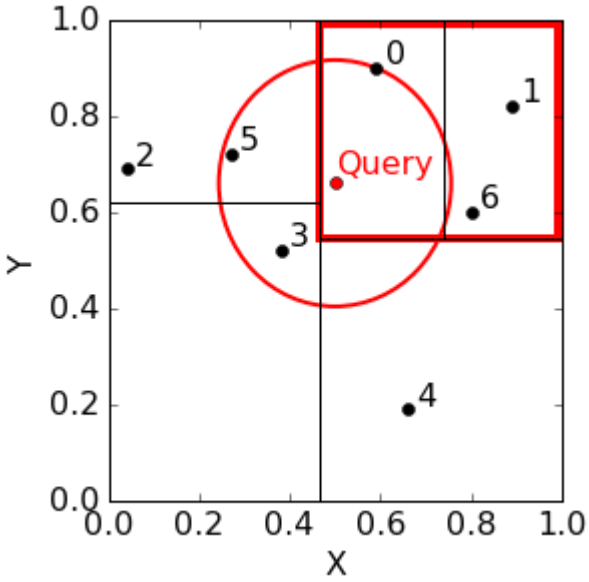


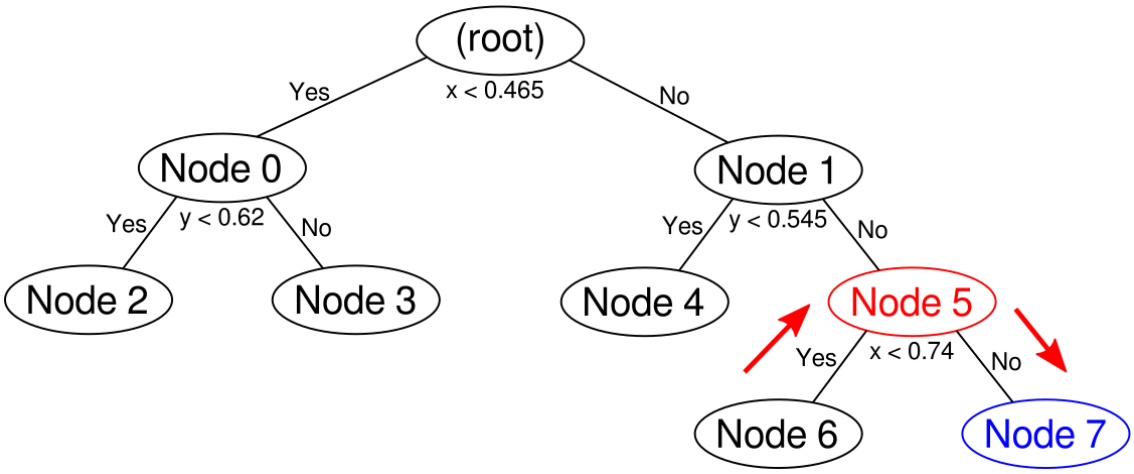


The Euclidean distance between the query point and data point 0 is approximately 0.25632. By definition, the nearest neighbor of the query must be closer to it than any other point. **So the nearest neighbor must be within distance 0.25632 of the query point. This observation will be crucial in performing an efficient nearest neighbor search.**



We now traverse one level up, to Node 5. **We do this because the nearest neighbor may not necessarily fall into the same node as the query point. Data point 0 is only the current estimate of the nearest neighbor; it may not be the true nearest neighbor.**



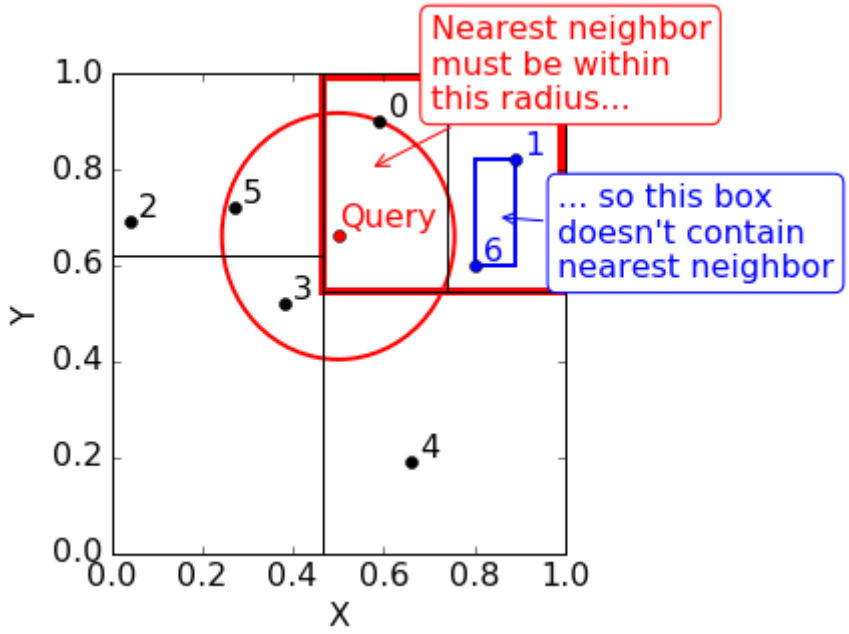


Do we need to inspect all data points in Node 5? It turns out we don't. **We look at the tight bounds of Node 7 to decide if we need to take a further look.** (If Node 5 contained the nearest neighbor but Node 6 did not, then by the process of elimination, Node 7 would contain the nearest neighbor.)

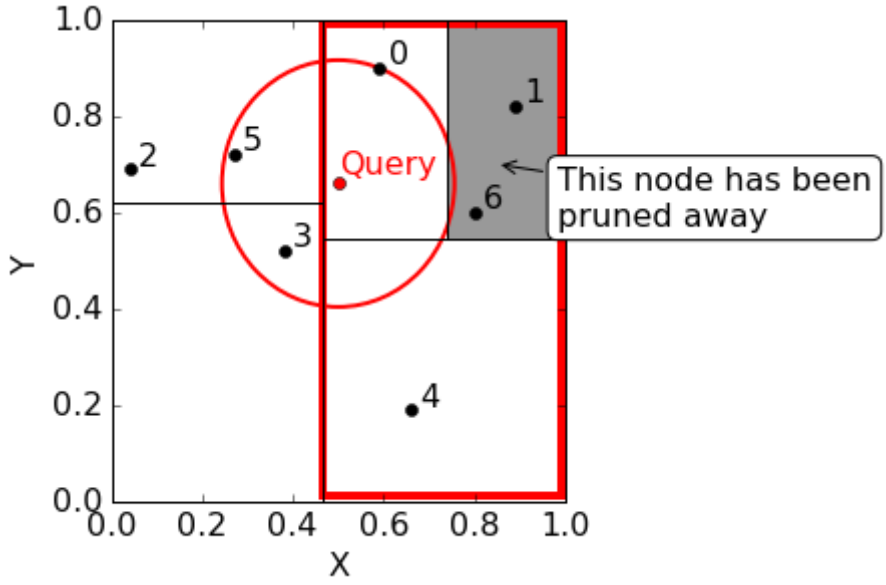
The bounds for Node 7 are 0.8 <= X <= 0.89 and 0.6 <= Y <= 0.82 for the X and Y axes respectively. By geometric argument, no point inside this box could fall inside the 0.25632 radius. That is, all points inside this box are farther from the query than data point 0. **In general, we do two things:**

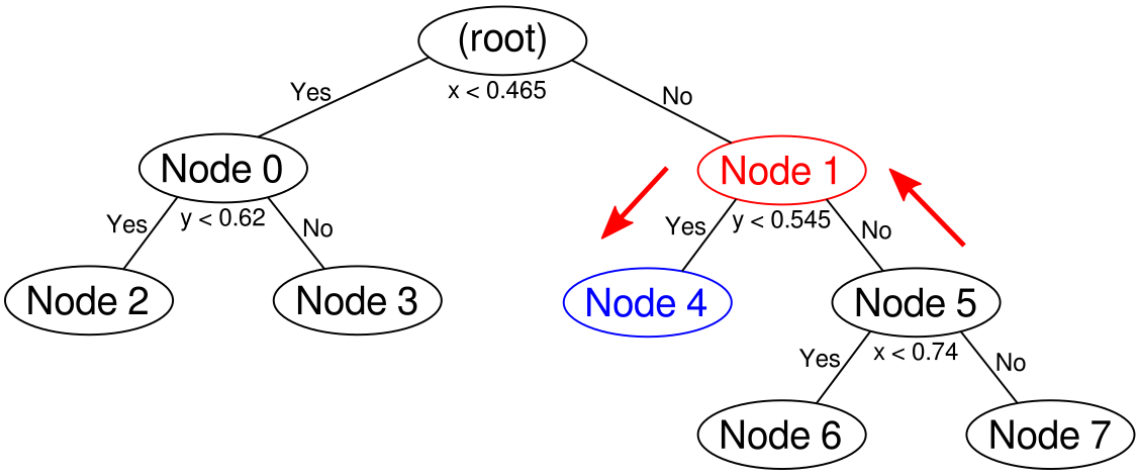
- **Draw a circle around the query point so that it touches the current estimate of the nearest neighbor (data point 0).**
- **See if the bounds for each nearby node has any overlap with the circle. If so, the estimate must be updated. Otherwise, the current estimate remains unmodified.**

In the case of Node 7, there is no overlap with the circle, so the node is **pruned away from the search.**

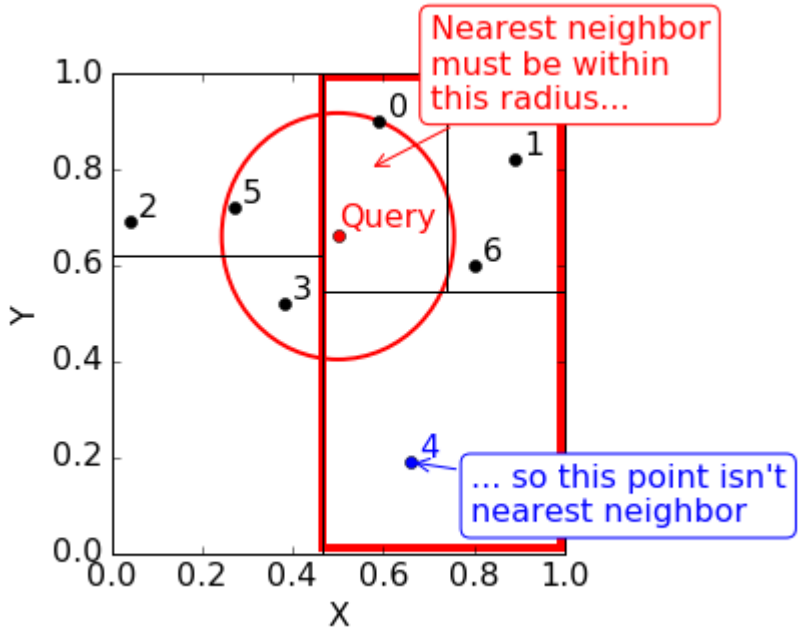


Let's traverse up another level:

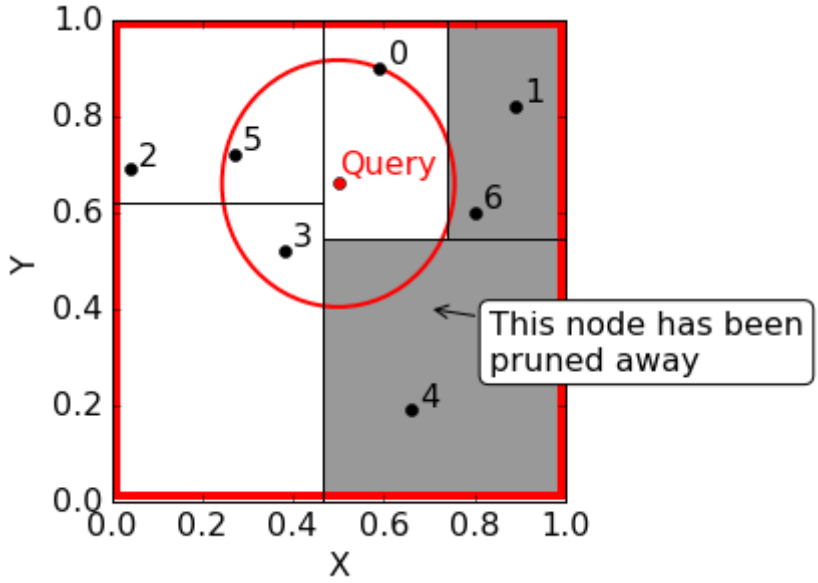


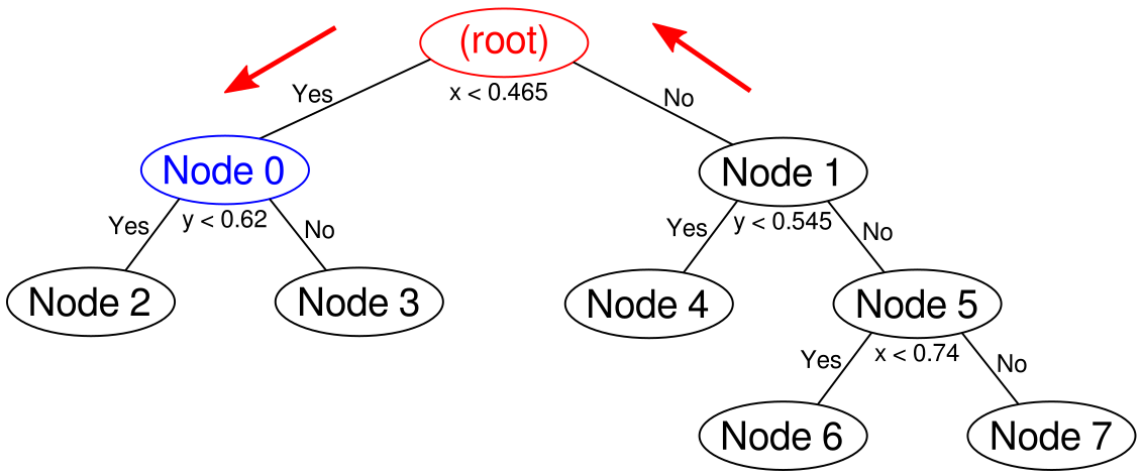


We inspect a nearby node, Node 4, for nearest neighbor search. Node 4 contains only a single point, and that point is far away from the circle. So Node 4 is pruned away as well.

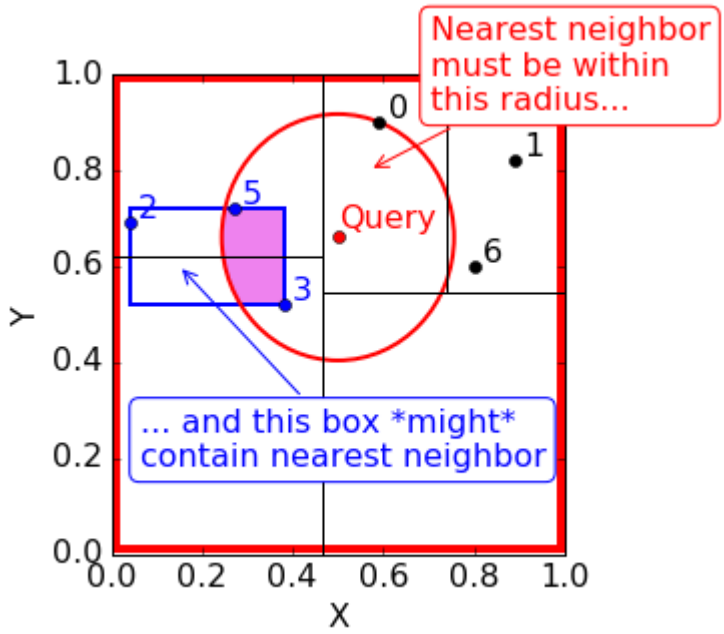


We traverse up one last time to reach the root node.

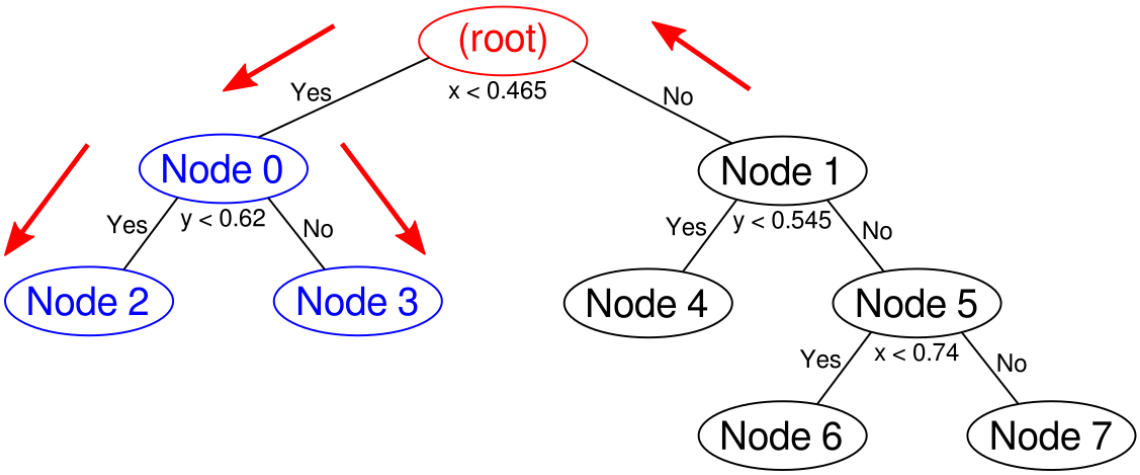




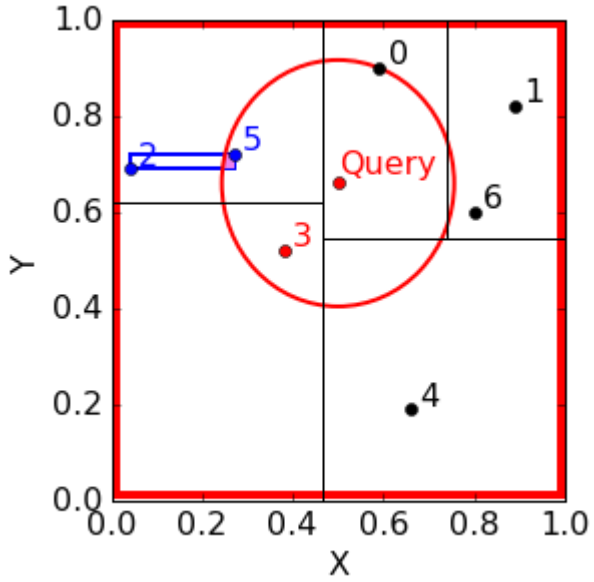
This time, the bounding box for Node 0 overlaps with the circle, indicating that Node 0 may contain a point that's closer to the query than data point 0 is.



To revise our estimate of the nearest neighbor, we inspect the child nodes of Node 0.



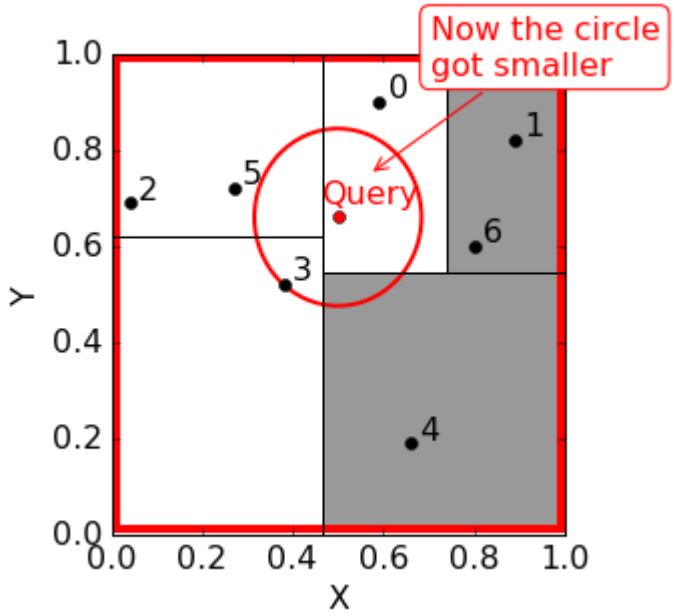
The bounding box for Node 2 has a slight overlap with the circle, and the sole data point in Node 3 is inside the circle as well. Therefore, we **fail to prune** Nodes 2 and 3. Since both are leaf nodes, we have to inspect all data points in these nodes.



We compare data points 0, 2, 3, and 5 using their distances from the query, and data point 3 comes out to be the closest:

	Distance from the query (0.5, 0.66)
Data point 0 (0.59, 0.9)	0.25632
Data point 2 (0.04, 0.69)	0.46098
Data point 3 (0.38, 0.52)	**0.18439**
Data point 5 (0.27, 0.72)	0.23770

The nearest neighbor estimate is now revised to be (0.38, 0.52):



Since we've reached the root node, we are done: data point 3 is indeed the true nearest neighbor of the query. Notice again how we could prune away two nodes from the nearest neighbor search just by using bounding boxes.

Note. This reading used two-dimensional data for simplicity, but the process can be generalized to 3 dimensions or higher, by replacing "circles" with hyper-spheres and "rectangles" with hyper-rectangles.

标记为完成