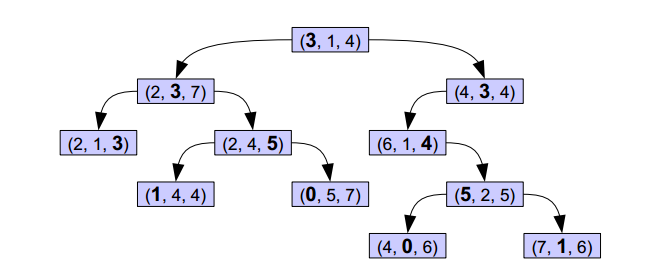
**KD树**

kd树（k-dimensional树的简称），是一种分割k维数据空间的数据结构。主要应用于多维空间关键数据的搜索（如：范围搜索和最近邻搜索）。

### 一个KDTree的例子



上图的树就是一棵KDTree，形似二叉搜索树，其实KDTree就是二叉搜索树的变种。这里的K = 3.

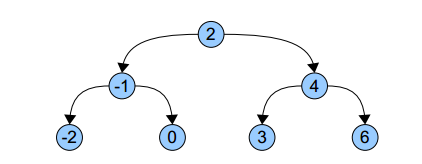
首先来看下树的组织原则。将每一个元组按0排序（第一项序号为0，第二项序号为1，第三项序号为2），在树的第n层，第 n%3 项被用粗体显示，而这些被粗体显示的树就是作为二叉搜索树的key值，比如，根节点的左子树中的每一个节点的第一个项均小于根节点的的第一项，右子树的节点中第一项均大于根节点的第一项，子树依次类推。

对于这样的一棵树，对其进行搜索节点会非常容易，给定一个元组，首先和根节点比较第一项，小于往左，大于往右，第二层比较第二项，依次类推。

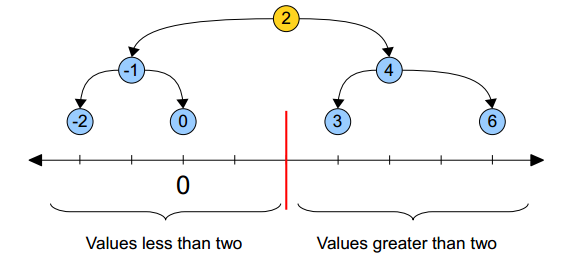
### 分割的概念

看了上面的例子，确实比较简单，但不知道为何要这样做，这里从几何意义出发，引出分割的概念。

先看一个标准的BSTree，每个节点只有一个key值。



将key值对应到一维的坐标轴上。

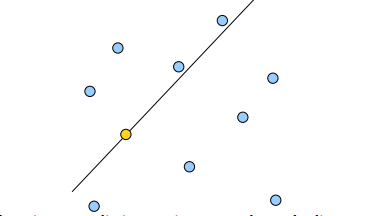


根节点对应的就是2，左子树都在2的左边，右子树都在2的右边，整个一维空间就被根节点分割成了两个部分，当要查找结点0的时候，由于是在2的左边，所以可以放心的只搜索左子树的部分。整个搜索的过程可以看成不断分割搜索区间的过程，直到找到目标节点。

这样的分割可以扩展到二维甚至更多维的情况。

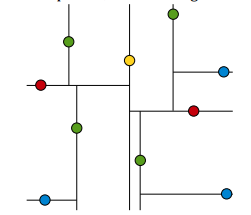
但是问题来了，**二维的节点怎么比较大小？**

在BSTree中，节点分割的是一维数轴，那么在二维中，就应当是分割平面了，就像这样：

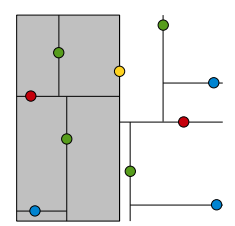
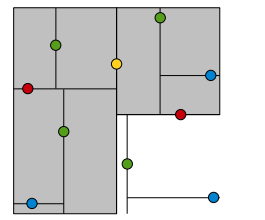
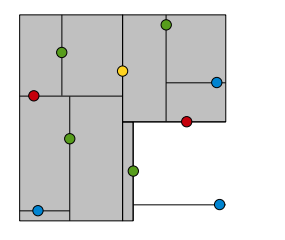


黄色的点作为根节点，上面的点归左子树，下面的点归右子树，接下来再不断地划分，最后得到一棵树就是赫赫有名的BSPTree（binary space partitioning tree）. 分割的那条线叫做分割超平面（splitting hyperplane），在一维中是一个点，二维中是线，三维的是面。

KDTree就是超平面都垂直于轴的BSPTree。同样的数据集，用KDTree划分之后就是这样：



黄色节点就是Root节点，下一层是红色，再下一层是绿色，再下一层是蓝色。为了更好的理解KDTree的分割，我们在图形中来形象地看一下搜索的过程，假设现在需要搜寻右下角的一个点，首先要做的就是比较这个点的x坐标和root点的x坐标值，由于x坐标值大于root节点的x坐标，所以只需要在右边搜寻，接下来，要比较该节点和右边红色节点y值得大小...后面依此类推。整个过程如下图：

   ->  ->

理解完KDTree之后，下面要说的就是关于KDTree的两个最重要的问题：

1.树的建立；

2.最近邻域搜索（Nearest-Neighbor Lookup）。

k-d树构建算法

　　k-d树是一个二叉树，每个节点表示一个空间范围。表1给出的是k-d树每个节点中主要包含的数据结构。

表1  k-d树中每个节点的数据类型

|  |  |  |
| --- | --- | --- |
| 域名 | 数据类型 | 描述 |
| Node-data | 数据矢量 | 数据集中某个数据点，是n维矢量（这里也就是k维） |
| Range | 空间矢量 | 该节点所代表的空间范围 |
| split | 整数 | 垂直于分割超平面的方向轴序号 |
| Left | k-d树 | 由位于该节点分割超平面左子空间内所有数据点所构成的k-d树 |
| Right | k-d树 | 由位于该节点分割超平面右子空间内所有数据点所构成的k-d树 |
| parent | k-d树 | 父节点 |

　　从上面对k-d树节点的数据类型的描述可以看出构建k-d树是一个逐级展开的递归过程。表2给出的是构建k-d树的伪码。

表2  构建k-d树的伪码

|  |
| --- |
| 算法：构建k-d树（createKDTree） |
| 输入：数据点集Data-set和其所在的空间Range |
| 输出：Kd，类型为k-d tree |
| 1.If Data-set为空，则返回空的k-d tree |
| 2.调用节点生成程序：  　　（1）确定split域：对于所有描述子数据（特征矢量），统计它们在每个维上的数据方差。以SURF特征为例，描述子为64维，可计算64个方差。挑选出最大值，对应的维就是split域的值。数据方差大表明沿该坐标轴方向上的数据分散得比较开，在这个方向上进行数据分割有较好的分辨率；  　　（2）确定Node-data域：数据点集Data-set按其第split域的值排序。位于正中间的那个数据点被选为Node-data。此时新的Data-set' = Data-set\Node-data（除去其中Node-data这一点）。 |
| 3.dataleft = {d属于Data-set' && d[split] ≤ Node-data[split]}     Left\_Range = {Range && dataleft}     dataright = {d属于Data-set' && d[split] > Node-data[split]}     Right\_Range = {Range && dataright} |
| 4.left = 由（dataleft，Left\_Range）建立的k-d tree，即递归调用createKDTree（dataleft，Left\_     Range）。并设置left的parent域为Kd；     right = 由（dataright，Right\_Range）建立的k-d tree，即调用createKDTree（dataleft，Left\_     Range）。并设置right的parent域为Kd。 |

　　以上述举的实例来看，过程如下：

　　由于此例简单，数据维度只有2维，所以可以简单地给x，y两个方向轴编号为0,1，也即split={0,1}。

　　（1）确定split域的首先该取的值。分别计算x，y方向上数据的方差得知x方向上的方差最大，所以split域值首先取0，也就是x轴方向；

　　（2）确定Node-data的域值。根据x轴方向的值2,5,9,4,8,7排序选出中值为7，所以Node-data = （7,2）。这样，该节点的分割超平面就是通过（7,2）并垂直于split = 0（x轴）的直线x = 7；

　　（3）确定左子空间和右子空间。分割超平面x = 7将整个空间分为两部分，如图2所示。x < =  7的部分为左子空间，包含3个节点{（2,3），（5,4），（4,7）}；另一部分为右子空间，包含2个节点{（9,6），（8,1）}。

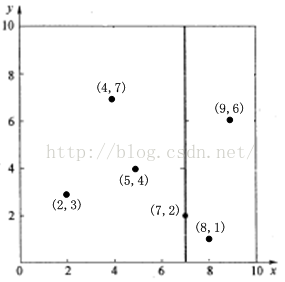


图2  x=7将整个空间分为两部分

中位数的代码为：

point\_list.sort(key=itemgetter(axis))median = len(point\_list) // 2 # choose medianlocation=point\_list[median]

**len(point\_list) // 2** 为序列的长度除2后向下取整，结果为3，由于Python序列的索引是从0开始的（差不多所有的编程语言索引都是从0开始的，matlab是个例外），所以实际中位数的取值就是排序后的第4个数，所以中位数的结果通过此程序得到的是7。现在我们从这段代码可以得到，在KD-Tree中中位数的求取是序列的长度除以2向下取整后的下一个数（升序排序）

　　如算法所述，k-d树的构建是一个递归的过程。然后对左子空间和右子空间内的数据重复根节点的过程就可以得到下一级子节点（5,4）和（9,6）（也就是左右子空间的'根'节点），同时将空间和数据集进一步细分。如此反复直到空间中只包含一个数据点，如图1所示。最后生成的k-d树如图3所示。

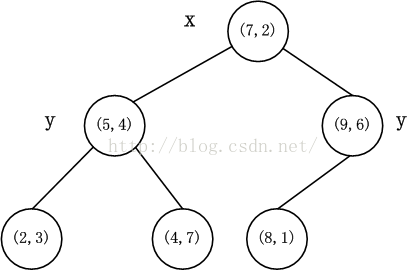


图3  上述实例生成的k-d树

　　注意：每一级节点旁边的'x'和'y'表示以该节点分割左右子空间时split所取的值。

### 邻近搜索

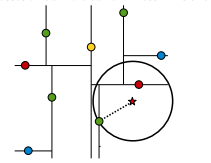
给定一个KDTree和一个节点，求KDTree中离这个节点最近的节点.(这个节点就是最临近点)

这里距离的求法用的是欧式距离。

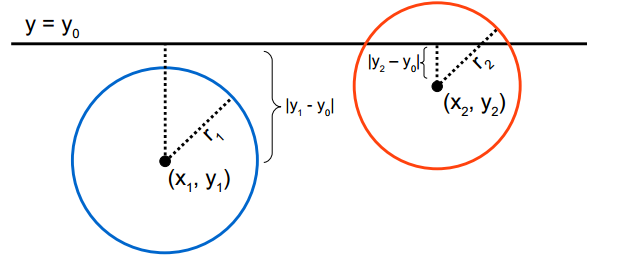
IMG_256

基本的思路很简单：首先通过二叉树搜索（比较待查询节点和分裂节点的分裂维的值，小于等于就进入左子树分支，等于就进入右子树分支直到叶子结点），顺着“搜索路径”很快能找到最近邻的近似点，也就是与待查询点处于同一个子空间的叶子结点；然后再回溯搜索路径，并判断搜索路径上的结点的其他子结点空间中是否可能有距离查询点更近的数据点，如果有可能，则需要跳到其他子结点空间中去搜索（将其他子结点加入到搜索路径）。重复这个过程直到搜索路径为空。

这里还有几个细节需要注意一下，如下图，假设标记为星星的点是 test point， 绿色的点是找到的近似点，在回溯过程中，需要用到一个队列，存储需要回溯的点，在判断其他子节点空间中是否有可能有距离查询点更近的数据点时，做法是以查询点为圆心，以当前的最近距离为半径画圆，这个圆称为候选超球（candidate hypersphere），如果圆与回溯点的轴相交，则需要将轴另一边的节点都放到回溯队列里面来。

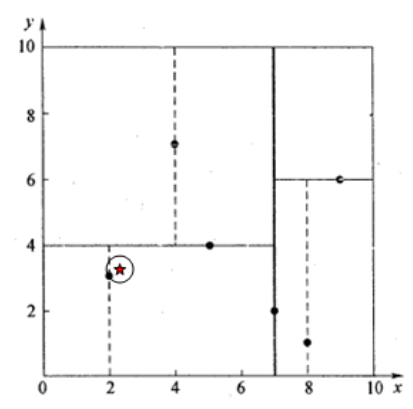


判断轴是否与候选超球相交的方法可以参考下图：



下面再用一个例子来具体说一下查询的过程。

假设我们的k-d tree就是上面通过样本集{(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)}创建的。  
我们来查找点(2.1,3.1)，在(7,2)点测试到达(5,4)，在(5,4)点测试到达(2,3)，然后search\_path中的结点为<(7,2), (5,4), (2,3)>，从search\_path中取出(2,3)作为当前最佳结点nearest, dist为0.141；  
然后回溯至(5,4)，以(2.1,3.1)为圆心，以dist=0.141为半径画一个圆，并不和超平面y=4相交，如下图，所以不必跳到结点(5,4)的右子空间去搜索，因为右子空间中不可能有更近样本点了。  
于是在回溯至(7,2)，同理，以(2.1,3.1)为圆心，以dist=0.141为半径画一个圆并不和超平面x=7相交，所以也不用跳到结点(7,2)的右子空间去搜索。  
至此，search\_path为空，结束整个搜索，返回nearest(2,3)作为(2.1,3.1)的最近邻点，最近距离为0.141。



再举一个稍微复杂的例子，我们来查找点(2,4.5)，在(7,2)处测试到达(5,4)，在(5,4)处测试到达(4,7)，然后search\_path中的结点为<(7,2), (5,4), (4,7)>，从search\_path中取出(4,7)作为当前最佳结点nearest, dist为3.202；  
然后回溯至(5,4)，以(2,4.5)为圆心，以dist=3.202为半径画一个圆与超平面y=4相交，如下图，所以需要跳到(5,4)的左子空间去搜索。所以要将(2,3)加入到search\_path中，现在search\_path中的结点为<(7,2), (2, 3)>；另外，(5,4)与(2,4.5)的距离为3.04 < dist = 3.202，所以将(5,4)赋给nearest，并且dist=3.04。  
回溯至(2,3)，(2,3)是叶子节点，直接平判断(2,3)是否离(2,4.5)更近，计算得到距离为1.5，所以nearest更新为(2,3)，dist更新为(1.5)  
回溯至(7,2)，同理，以(2,4.5)为圆心，以dist=1.5为半径画一个圆并不和超平面x=7相交, 所以不用跳到结点(7,2)的右子空间去搜索。

至此，search\_path为空，结束整个搜索，返回nearest(2,3)作为(2,4.5)的最近邻点，最近距离为1.5。

