前言

基础的数据结构如二叉树衍生的的平衡二叉搜索树通过左旋右旋调整树的平衡维护数据，靠着二分算法能满足一维度数据的logN时间复杂度的近似搜索。对于大规模多维度数据近似搜索，Lucene采用一种BKD结构，该结构能很好的空间利用率和性能。

本片博客主要学习常见的多维数据搜索数据结构、KD-Tree的构建、搜索过程以针对高维度数据容灾的优化的BBF算法，以及BKD结构原理。

感受 算法之美 结构之道 吧~

目录

一、多维数据空间搜索结构

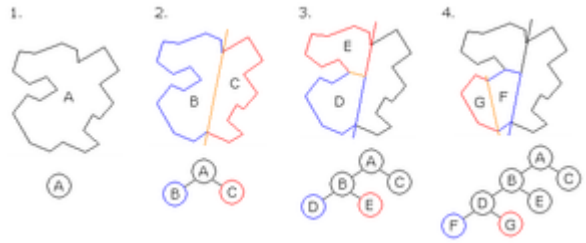
BKD-Tree是基于KD-B-Tree改进而来，而KD-B-Tree又是KD-Tree和B+Tree的结合体，KD-Tree又是我们最熟悉的二叉查找树BST(Binary Search Tree)在多维数据的自然扩展，它是BSP(Binary Space Partitioning)的一种。B+Tree又是对B-Tree的扩展。以下对这几种树的特点简要描述。

1、KD-Tree

kd是K-Dimensional的所写，k值表示维度，KD-Tree表示能处理K维数据的树结构，当K为1的时候，就转化为了BST结构

> 维基百科：在计算机科学里，k-d树（k-维树的缩写）是在k维欧几里德空间组织点的数据结构。k-d树可以使用在多种应用场合，如多维键值搜索（例：范围搜寻及最邻近搜索）。k-d树是空间二分算法（binary space partitioning）的一种特殊情况。

首先看BSP，Binary space partitioning(BSP)是一种使用超平面递归划分空间到凸集的一种方法。使用该方法划分空间可以得到表示空间中对象的一个树形数据结构。这个树形数据结构被我们叫做BSP树。



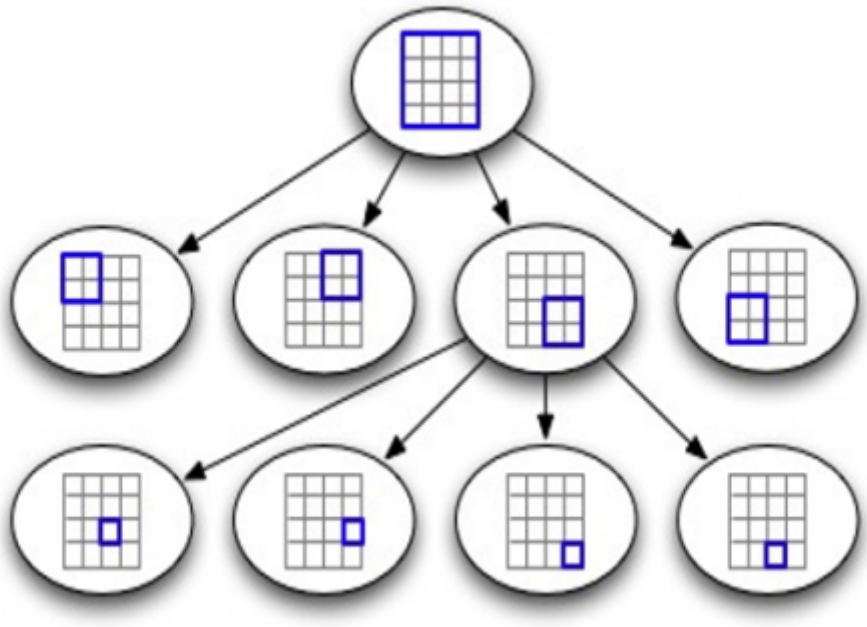
可以分为轴对齐、多边形对齐BSP，这两种方式就是选择超平面的方式不一样，已轴对齐BSP通过构建过程简单理解，就是选择一个超平面，这个超平面是跟选取的轴垂直的一个平面，通过超平面将空间分为两个子空间，然后递归划分子空间。
空间划分思想可以转化为坐标点划分，一般可以应用在游戏中如物体定位等，比如二维空间的四叉树和三维空间的八叉树都是参考BSP划分算法。

1.1、BSP树和四叉树的关系

BSP算法和四叉树的关系

- **BSP树**：BSP树使用平面进行递归的二分划分，将空间划分为两个子空间。每个节点要么是叶子节点（包含实际对象），要么是内部节点（包含一个分割平面）。分割平面通常由空间中的一条直线表示。
- **四叉树**：四叉树将空间划分为四个象限，每个象限都是父节点的子节点。每个节点要么是叶子节点（包含实际对象），要么是内部节点（包含四个子节点）。

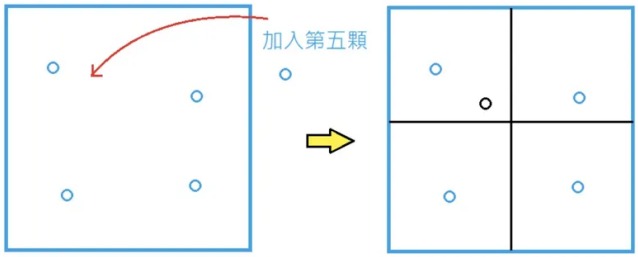
四叉树又分为点四叉树和边四叉树，以边四叉树为例，具体的实现源码参考：空间搜索优化算法之——四叉树 - 掘金



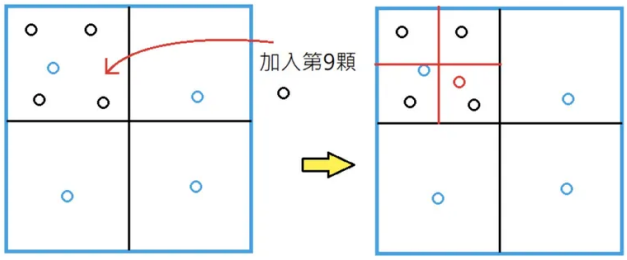
插入流程:

假設我先設定每個區域只能容納4個物體，只要超過該容量，就要分割該區域。

左圖方形區域已經有4個物體，想再加入第5個時，就必須分割成4個子區域，再將第5顆分類到最近的左上角區域中，如右圖:



以此類推，當要放入第9顆物體時，發現黑色區域也滿了，所以就再往下分割紅色區域，並放入離該物體最近的右下角區域中，如下圖:



1.2、KD-Tree和BSP树的关系

KD-Tree是一种特殊的BSP树，它的特点有：

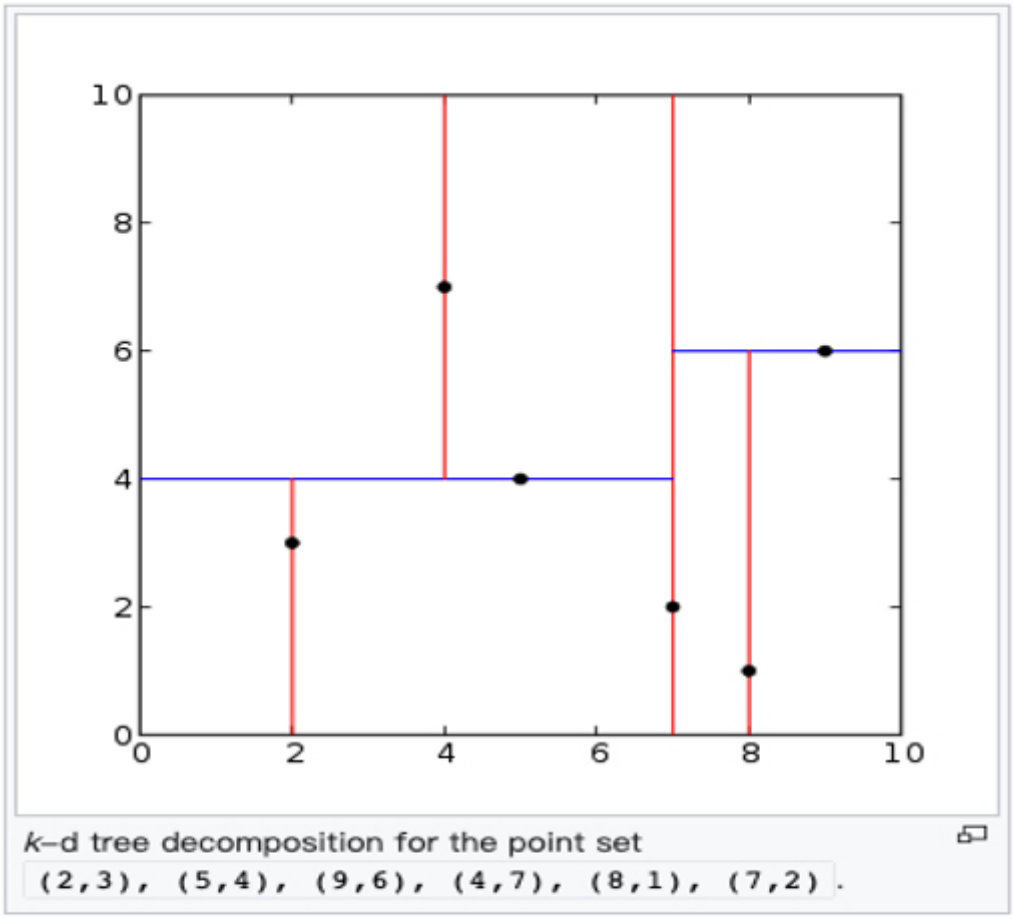
- 每一层都是一种划分维度，而BSP划分的维度为轴划分、边划分，是同一维度的划分。
- 每个节点代表垂直于当前维度的超平面，将空间划分为两部分
- k维空间，按树的每一层循环选取，当前节点为i维，下一层节点为(i+1)%k维

KD 树（KD-tree）和 BSP 树（Binary Space Partitioning tree）都是用于空间划分的数据结构，但它们有一些关键的区别，这也是为什么 KD 树被认为是 BSP 树的一种特殊情况的原因之一。

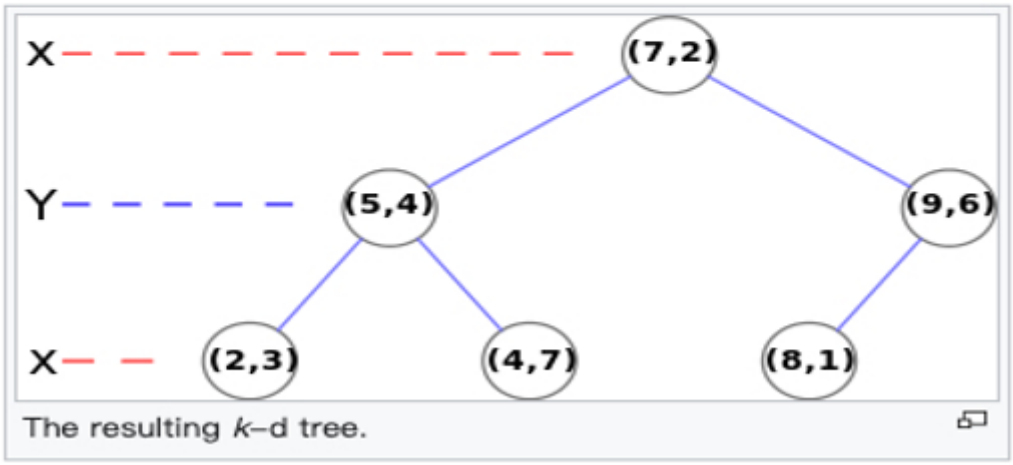
1. **维度划分方式不同**：
    - KD 树：KD 树是针对 k 维空间的树形数据结构，它在每个节点上通过轮流选择一个维度来划分空间，例如在二维空间中，它可能在 x 轴上进行一次划分，在 y 轴上进行下一次划分，以此类推。因此，KD 树在每一层都会选择一个维度进行划分。
    - BSP 树：BSP 树是一种二叉树，每个节点都代表一个超平面（hyperplane），用于将空间划分为两个子空间。BSP 树的划分方式不一定是轮流选择维度，而是根据一些准则（如最佳平面）选择划分的超平面。
2. **节点类型不同**：
    - KD 树：KD 树的节点可以是叶节点，也可以是非叶节点。非叶节点表示一个划分超平面，叶节点表示一个数据点。
    - BSP 树：BSP 树的每个节点都是一个划分超平面，它没有叶节点来表示数据点。
3. **适用场景不同**：
    - KD 树：KD 树主要用于 k 维空间中的最近邻搜索等问题，由于它在每个节点上都选择一个维度进行划分，因此在高维空间中可能会出现维度灾难（curse of dimensionality）的问题。
    - BSP 树：BSP 树更通用，可以用于任何维度的空间划分，常用于图形学中的空间分区和碰撞检测等问题。

因此，虽然 KD 树和 BSP 树都是空间划分的数据结构，但由于它们的设计和应用场景有所不同，KD 树被认为是 BSP 树的一种特殊情况。

下面是一个2维度的KD-tree，类似BST，只不过BST是一维的



k–d tree decomposition for the point set
(2,3), (5,4), (9,6), (4,7), (8,1), (7,2).



The resulting *k*–d tree.

先KD-Tree适宜处理多维数据，查询效率较高。不难知道一个静态多维数据集合建成KD-Tree后查询时间复杂度是O（lgN)。所有节点都存储了数据本身，导致索引数据的内存利用不够紧凑，相应地数据磁盘存储的空间利用不够充分。

此外KD-Tree不适宜处理海量数据的动态更新。原因和B树不适宜处理多维数据的动态更新的分析差不多，因为KD-Tree的分层划分是依维度依次轮替进行的，动态更新后调整某个中间节点时，变更的当前维度也同样需要调整其全部子孙节点中的当前维度值，导致对树节点的访问和操作增多，操作耗时增大。可见，KD-Tree更适宜处理的是静态场景的多维海量数据的查询操作。

1.3、KD-Tree和KNN算法的联系

KNN算法的实现就可以采KD-Tree：https://blog.csdn.net/v_july_v/article/details/8203674，这篇博客写的很详细，KNN算法简单理解就是给定一个测试元素，根据最靠近的K个元素判断测试元素的分类，当K=1的时候，就转化成了最紧邻算法，KD-Tree结构是支持最紧邻搜索的。

1.4、KD-Tree的原理

学习KD-Tree是如何构建、查询、删除元素的，使用Java实现一个简单二维的KD-Tree结构，实现寻找最近的n个点。

```
package org.example.kdtree;

import java.util.ArrayList;
import java.util.List;


/**
 * @author sichaolong
 * @createdate 2024/3/14 14:19
 */

class KDNode {
    int[] point;
    KDNode left;
    KDNode right;

    public KDNode(int[] point) {
        this.point = point;
        this.left = null;
        this.right = null;
    }
}

public class SimpleKDTreeDemo {
    private KDNode root;

    public SimpleKDTreeDemo() {
        this.root = null;
    }

    public void insert(int[] point) {
        this.root = insertNode(this.root, point, 0);
    }

    private KDNode insertNode(KDNode node, int[] point, int depth) {
        if (node == null) {
            return new KDNode(point);
        }

        int k = point.length;

        // 选定切割轴
        int axis = depth % k;

        if (point[axis] < node.point[axis]) {
            node.left = insertNode(node.left, point, depth + 1);
        } else {
            node.right = insertNode(node.right, point, depth + 1);
        }
```

```
        return node;
    }

    public List<int[]> search(int[] target, int n) {
        List<int[]> result = new ArrayList<>();
        searchNode(this.root, target, 0, n, result);
        return result;
    }

    private void searchNode(KDNode node, int[] target, int depth, int k,
List<int[]> result) {
        if (node == null) {
            return;
        }

        // 确定当前层的切割维度
        int axis = depth % k;

        if (target[axis] < node.point[axis]) {
            searchNode(node.left, target, depth + 1, k, result);
        } else {
            searchNode(node.right, target, depth + 1, k, result);
        }

        // 还没找够n个，就直接添加
        if (result.size() < k) {
            result.add(node.point);
        } else {
            // 上一个最近的点
            int[] farthestPoint = result.get(result.size() - 1);
            // 如果当前点距离更近，就替换
            if (distance(target, node.point) < distance(target, farthestPoint))
{
                result.remove(result.size() - 1);
                result.add(node.point);
            }
        }

        // 如果切割轴距离更近，就添加
        int[] farthestPoint = result.get(result.size() - 1);
        // 切割轴距离
        double splitDistance = Math.abs(target[axis] - node.point[axis]);
        // 切割轴距离更近
        if (splitDistance < distance(target, farthestPoint)) {
            if (target[axis] < node.point[axis]) {
                searchNode(node.right, target, depth + 1, k, result);
            } else {
                searchNode(node.left, target, depth + 1, k, result);
            }
        }
    }

    // 欧式距离
    private double distance(int[] point1, int[] point2) {
        int k = point1.length;
```

```
        double sum = 0;
        for (int i = 0; i < k; i++) {
            sum += Math.pow(point1[i] - point2[i], 2);
        }
        return Math.sqrt(sum);
    }

    public static void main(String[] args) {
        SimpleKDTreeDemo kdTree = new SimpleKDTreeDemo();
        int[][] points = {{2, 3}, {5, 4}, {9, 6}, {4, 7}, {8, 1}, {7, 2}};
        for (int[] point : points) {
            kdTree.insert(point);
        }

        int[] target = {6, 3};
        int n = 2;

        // 找出最近的n个点
        List<int[]> result = kdTree.search(target, n);
        System.out.println("The " + n + " nearest neighbors to the target point
" + java.util.Arrays.toString(target) + " are:");
        for (int[] point : result) {
            System.out.println(java.util.Arrays.toString(point));
        }
    }
}
```

构建

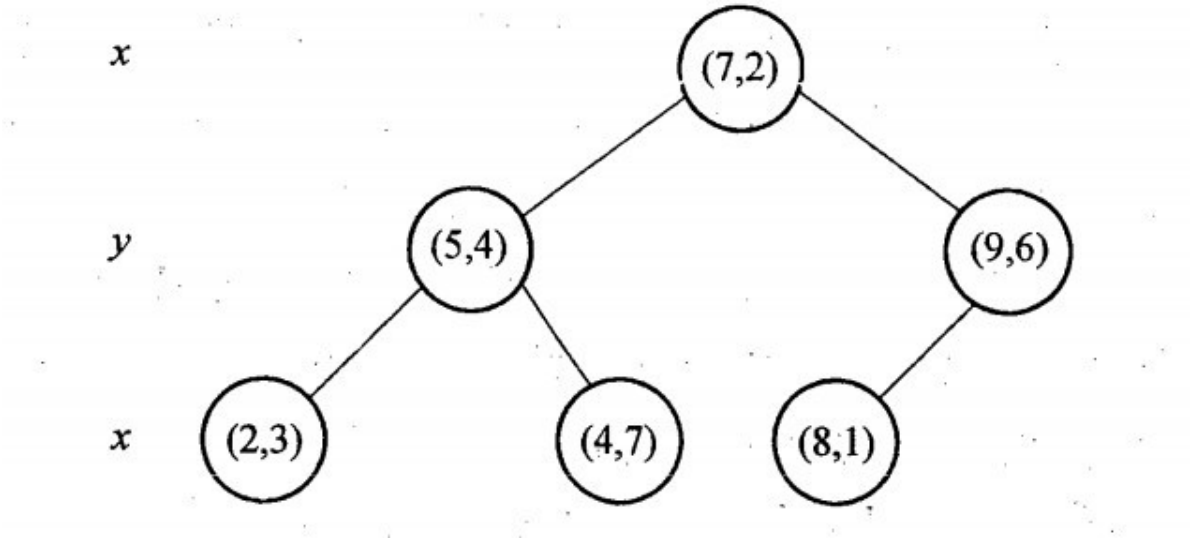
树的构建就是依靠递归，对于KD-Tree的构建步骤

1. 根据元素各个维度的方差，确定split域作为划分左、右子树的边界
2. 确定当前层的根结点，一般是取中间值
3. 划分左右子树

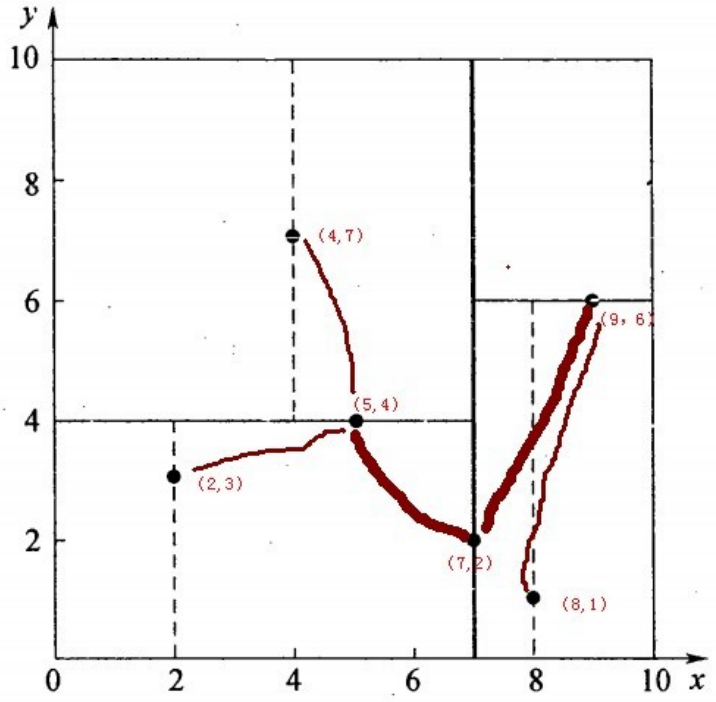
举例KD-Tree的构建过程， 6个二维数据点{(2,3)，(5,4)，(9,6)，(4,7)，(8,1)，(7,2)}构建kd树的具体步骤为：

1. 计算x维度的方差为1.24，y维度的方差为0.83，选定split域为x维度，方差的计算公式
$$s^2 = \frac{\sum_{i=1}^{n} (x_i - x)^2}{n}$$
2. 确定当前层的根结点为（7，2），经过该点垂直于split域的平面为 分割超平面
3. 左子树为（2，3）、（5，4）、（4，7），右子树为（9，6）、（8，1）

然后递归的交替使用x、y维度继续构建左、右子树，最终的结果，奇数层split域为x，偶数层为y。



使用x、y坐标轴表示KD-Tree



ps：上面的代码并没第一步，首次插入的节点被定为根节点

搜索

查询最紧邻的点，二维KD-Tree不像BST那样，因为按照维度分层，找到的叶子节点不一定是最紧邻的点，需要回溯，回溯到上一层父节点，查找父节点的其他子空间（分割平面划分的另外一个空间）是否可能有更近的点，依据就是以当前点为圆心，最近的距离为半径画圆，判断是否可能有其他点在圆内（判断的依据就是圆是否触达分割平面，是否包含其他点），距离度量同样使用欧式距离。

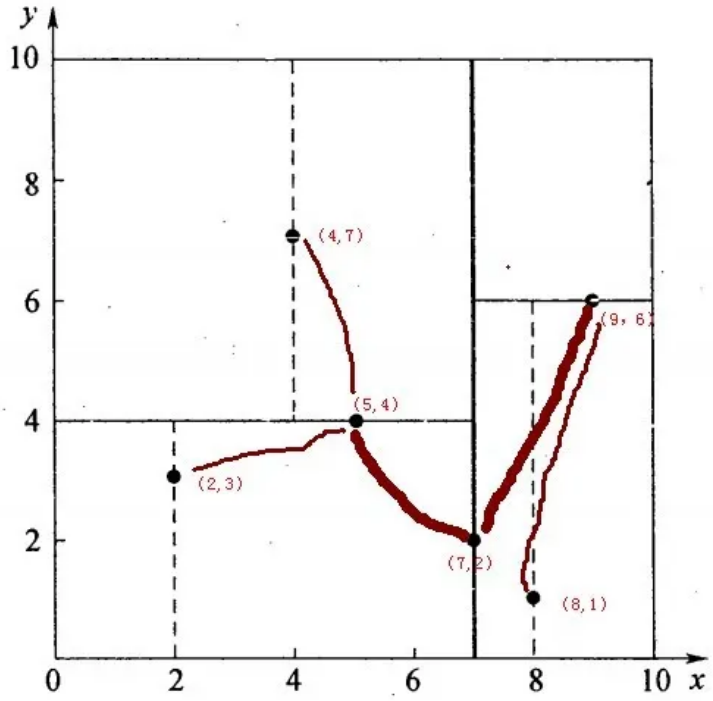
搜索过程，如果点是随机分布的，那么搜索的时间复杂度为O(lgN)，巧妙的地方就是回溯直接取栈元素就行

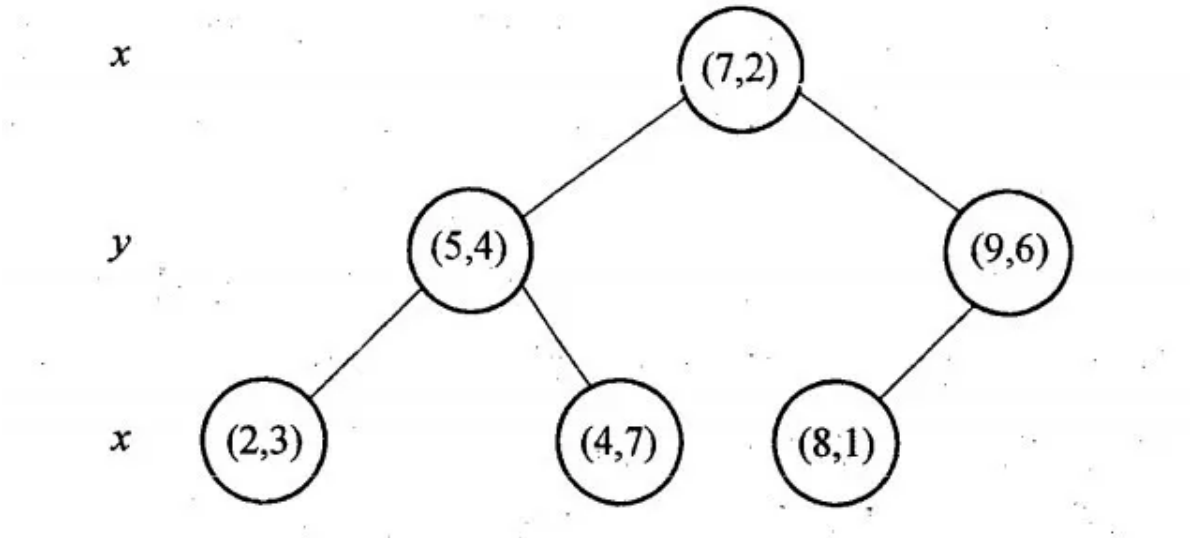
1. 从根节点递归的向下搜索，各维度交替向左、右子树搜索。
2. 找到叶子节点，计算距离，记录为临时最近距离以及临时最近节点target point，因为叶子节点不一定是最紧邻的点。
3. 回溯
    1. 回溯的父节点到搜索节点距离是否小于 临时最近距离，如果小于，更新临时target point。

2. 临时最近节点target point为圆点，临时最近距离为r画圆，圆是否和其他维度域分割平面相交，如果相交，需要搜索其他维度区域（假如当前进入的是root的左子树，本来不应该搜索右子树的，但是圆和其他维度空间相交就又可能其他空间有更近的点，需要搜索计算距离和临时最近点比较）

4. 回溯到根节点，找到最邻近节点。

举例，搜索（2.2，3.2）最紧邻的点





1. 首先从根结点（7，2）出发搜索，首先按照x维度为split域，进入左子树（5，4）
2. 接着按照y维度为split域名，进入左子树（2，3），找到了叶子节点（2，3），计算欧式距离为0.1414，计算父节点（5，4）距离为2.91 > 0.1414，因此目前target点（2，3）最近距离为0.1414。
3. 回溯判断
    1. 回溯到（5，4）按照（2.1，3.1）以0.1414为半径画圆，发现和y = 4这条分割域平面无交点，继续回溯，
    2. 回溯到（7，2）按照（2.1，3.1）以0.1414为半径画圆，发现和x = 7这条分割域平面无交点，至此回溯结束。

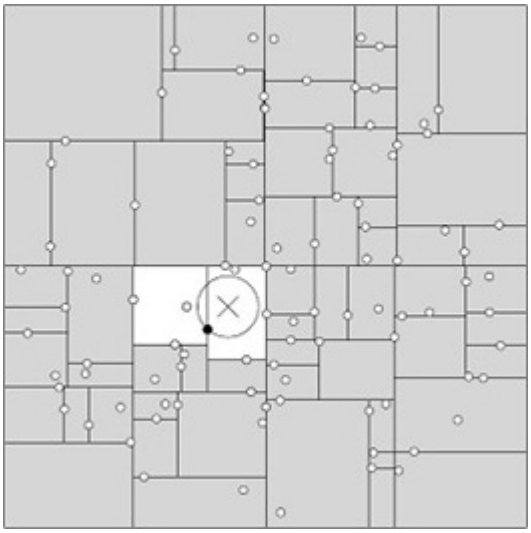
4. 找到最邻近的点为（2，3），最近距离为0.1414

举例，搜索（2，4.5）最紧邻的点

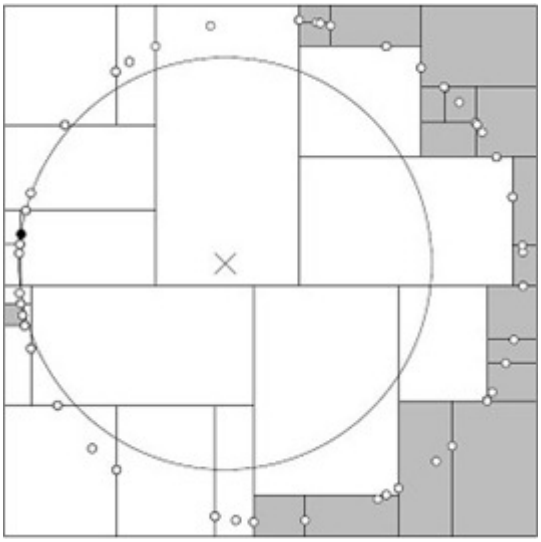
1. 首先从根结点（7，2）出发搜索，首先按照x维度为split域，进入左子树（5，4）
2. 接着按照y维度为split域名，进入右子树（4，7），找到了叶子节点（4，7），计算欧式距离为3.202，计算父节点（5，4）距离为3.04 < 3.202，因此目前target点（5，4）最近距离为3.04。
3. 回溯判断
    1. 回溯到（5，4）按照（2，4.5）以3.04为半径画圆，发现与y = 4这条分割域平面有交交点，所以需要搜索（5，4）的左空间（2，3），计算（2，3）距离（2，4.5）为1.5 < 3.04，因此目前target点（2，3）最近距离为1.5。
    2. 回溯到（7，2）按照（2，4.5）以半径1.5画圆，发现不和x = 7 这条分割域平面有交交点，至此回溯结束。
4. 找到最邻近的点（2，3），最近距离为1.5。

上面两个demo证明叶子节点不一定是最紧邻的target节点，需要以当前叶子节点（temp target节点）和 搜素节点 的欧式距离为r画圆，看圆是否和某个split域平面相交，如果相交，还需要去相交域接着找是否存在更紧邻的点，下面就是递归，直到圆和域切割面不在相交，最紧邻的target才找到。

一般来说，叶子节点只需要找几个即可



但是当点分布的比较糟糕，就需要递归查找很多域，因此当维数比较多的时候，KD-Tree树的性能会迅速下降，一般数据规模 N >> K平方 才能发挥比较不错的性能，比如100个2维度的点，其中 100 远远大于 2*2；实验结果表明当特征空间的维数超过20 的时候容易线形灾难。



1.5、KD-Tree搜索算法优化之BBF算法

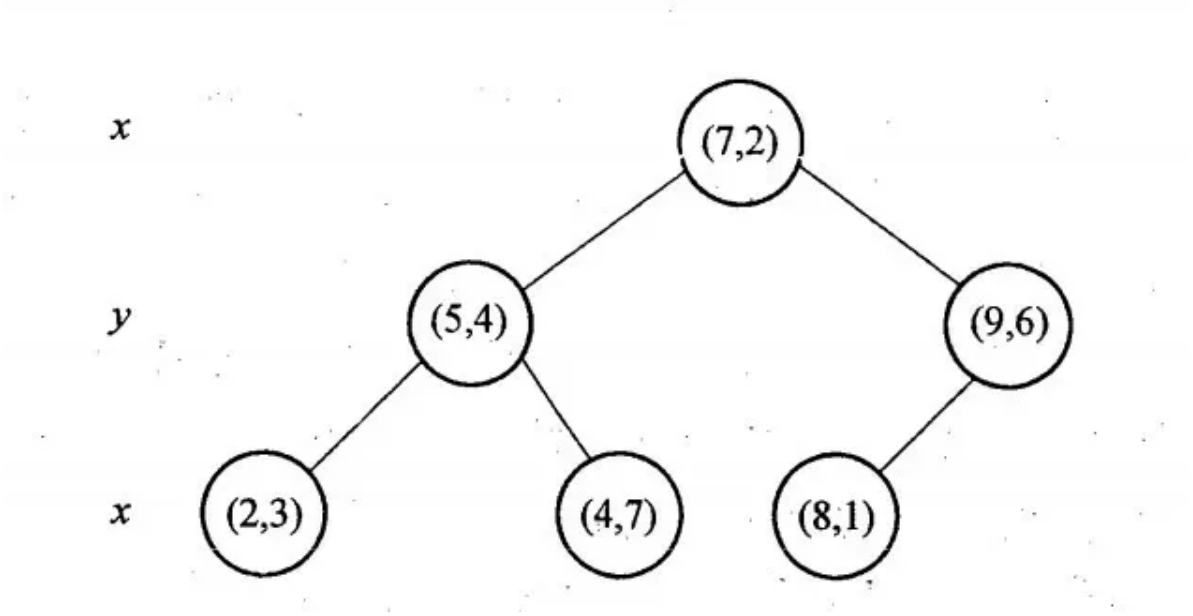
BBF（Best-Bin-First）查询算法，它是由发明sift算法的David Lowe在1997的一篇文章中针对高维数据提出的一种近似算法，此算法能确保优先检索包含最近邻点可能性较高的空间，此外，BBF机制还设置了一个运行超时限定。采用了BBF查询机制后，kd树便可以有效的扩展到高维数据集上。

上述的KD-Tree搜索过程得知，搜索回溯是有查询路径决定的，**查询的路径并没有考虑到数据本身的一些性质**，减少回溯到其他区域空间的次数，就能一定程度降低搜索计算次数，一个改进的思路就是对数据做一些处理，便于搜素的路径可控，如按各自分割超平面（也称bin）与查询点的距离排序，也就是说，回溯检查总是从优先级最高（Best Bin）的树结点开始。

对于BBF算法，就是把回溯的栈换成了有序的优先队列，然后按照优先队列里面的子树进行递归。

- 首先是为每一层的节点排个优先级，也就是各个节点到当前层计算维度轴的距离，记为abs(q[i]-v)，i为当前所选维度，v为到维度轴的距离。
- 搜索节点的时候，使用优先队列记录那些同层未被选择的兄弟节点，或者表兄弟节点。只有搜索到叶子节点才会回溯，才从优先队列取节点，回溯的时候直接从优先队列找，此时找到的基本上是理论最近点。然后递归更新找到的最紧邻的点，直到优先队列为空。
- 找到最紧邻的点。

举例，还是以上面搜索（2，4.5）最紧邻的点

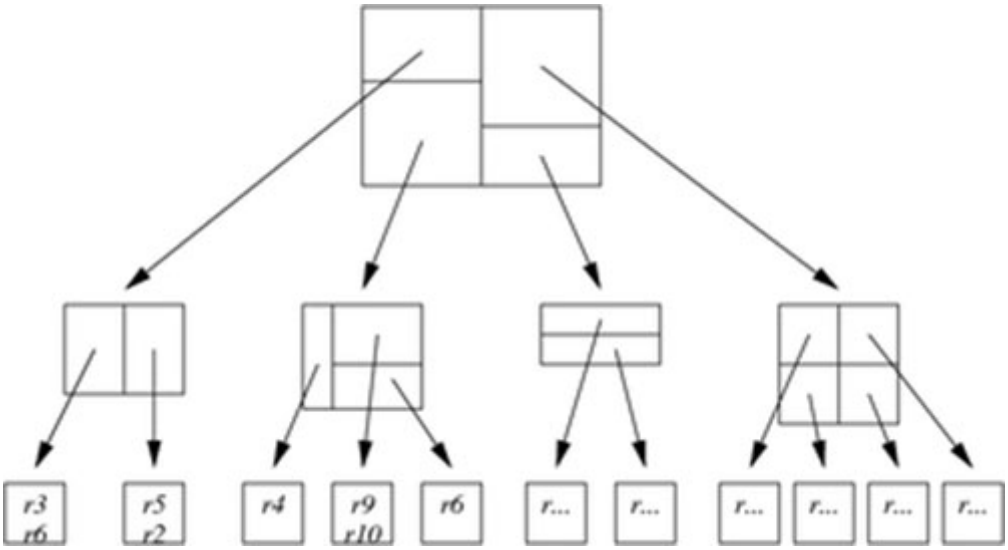


1. 首先将根节点放入优先队列。
2. 首先从根结点（7，2）出发搜索，首先按照x维度为split域，进入左子树（5，4），此时把右子树根节点（9，6）放入优先队列，此时队列顶元素为（7，2）
3. 接着按照y维度为split域名，进入右子树（4，7），将左子树根节点（2，3）放入优先队列，此时队列元素有{（2，3）、（7，2），（5，4）}，优先队列队顶元素为（5，4）。找到了叶子节点（4，7），计算欧式距离为3.202，计算父节点（5，4）距离为3.04 < 3.202，因此目前target点（5，4）最近距离为3.04。
4. 回溯判断：提取优先队列队顶元素（2,3），重复步骤2，直到优先队列为空。
5. 找到最邻近的点（2，3），最近距离为1.5。

ps：针对KD-Tree结构存在的问题，还有很多优化的数据结构如球树、R树、VP树、MVP树。
球树简单理解就是不在像KD-Tree使用split域将整个空间分割成一个个矩形，而是分成了一个个圆形，这样可以很好的处理KD-Tree不能很好的处理位于举矩形空间角落的点。
VP树又叫至高树，而在vpt中，首先从节点中选择一个数据点（可随机选）作为制高点（vp），然后算出其它点到vp的距离大小，最后根据该距离大小将数据点均分为二，递归建树。
R树：https://zh.wikipedia.org/wiki/R%E6%A0%91

2、KD-B-Tree

KD-B-Tree（K-Dimension-Balanced-Tree）顾名思义，结合了KD-Tree和B+Tree。它主要解决了KD-Tree的二叉树形式树高较高，对磁盘IO不够友好的缺点，引入了B+树的多叉树形式，不仅降低了树高，而且全部数据都存储在叶子节点，增加了磁盘空间存储利用率。一个KD-B-Tree结构的示意图如下。它同样不适宜多维数据的动态更新场景，原因同KD-Tree一样。



无法应对频繁插入、删除的操作：同样是因为KD-B-Tree的分层划分是依维度依次轮替进行的，动态更新后调整某个中间节点时，变更的当前维度也同样需要调整其全部子孙节点中的当前维度值，导致对树节点的访问和操作增多，操作耗时增大。

> 要将点插入 K-D-Btree，log（N/B） I/O 中遵循根到叶的路径，在叶中插入该点后，叶和路径上的其他节点将被拆分，就像在 B 树中一样。然而，与 B 树不同，但与 kd 树类似，内部节点 v 的拆分可能会导致需要拆分根植于 v 子节点的几个子树，参见图 1。因此，更新可能非常低效，也许更重要的是，空间利用率会大大降低，因为拆分过程可能会产生许多近乎空的叶子[22]。

二、BKD-Tree原理

2.1、概述

BKD-Tree(或BK-D-Tree,全称是Block-K-Dimension-Tree) 是一个多维数据索引结构，为外部存储器设计，是基于KD树（KD-B-Tree）的一个拓展，使用对数方法维护一组KD-B-Tree将静态结构动态化，能实现KD-B-Tree几乎100%的空间利用以及快速查询处理，并且这种效果是在大量更新中完成的，解决了KD-B-Tree静态数据结构更新导致结构质量恶化的问题。

> 摘自论文：在本文中，我们提出了一种新的索引结构，称为Bkd-tree，用于索引大型多维点数据集。 Bkdtree 是一种基于 kd-tree 的 I/O 高效动态数据结构。我们提出了一项广泛的实验研究的结果，表明与之前将 kd-tree 的外部版本动态化的尝试不同，Bkd-tree 保持了其高空间利用率和出色的性能。查询和更新性能与对其执行的更新数量无关。

> 摘自论文：高效索引文件必须具有较高的空间利用率并且能够快速的处理查询，并且应在大量更新负载下保持这两个属性。
> 针对第二点意味着一个高效多维数据索引结构必须要求在面对数据插入、删除的过程中，结构应该尽可能少的改变，大多数索引结构的质量会随着对他们进行大量更新导致恶化，**因此在保持高空间利用率的同时处理大量更新负载的问题被认为是一个重要的研究问题。**

空间数据库常用的两种查询

- 正交范围查询：比如查找年龄在[a,b]，体重在[c,d]，年龄在[p,q]的员工有哪些？可以使用x、y、z建立坐标空间，然后求出位于相交长方体内的点即可。
- 窗口查询：二维中窗口查询是一个轴对称的矩形，目标是找出矩形区域内所有的点。其他类似支持窗口查询的多维数据结构如KD-B-Tree、hB-Tree、R-Tree等。

> **我们不是维护一棵树并在插入后动态地重新平衡它，而是维护一组 对数（N/M）静态K-D-B树，并通过定期重建一组精心选择的结构来执行更新（M是内存缓冲区的容量，以点数为单位）**。通过这种方式，我们可以保持静态 K-D-B 树接近 100% 的空间利用率。O'Neill等[20]和Jagadishet等[14]也使用了维护多个树以加快插入时间的想法。它们的结构用于在单个属性上索引点，并且其技术无法扩展以有效地处理多维点。
> 要使用 Bkd 树回答窗口查询，我们必须查询所有 log2(N/M) 结构，而不仅仅是一个，但从理论上讲，我们实际上保持最坏情况的最优 O(√N/B + K/B)

2.2、构建

B-KD-Tree 由一组平衡的 KD-Tree 组成。每个 kd 树在磁盘上的布局（或组织）方式与 K-DB 树的布局方式类似，但是也不安全一样，每个kd树必须是完全二叉树，即节点个数满足2的幂。

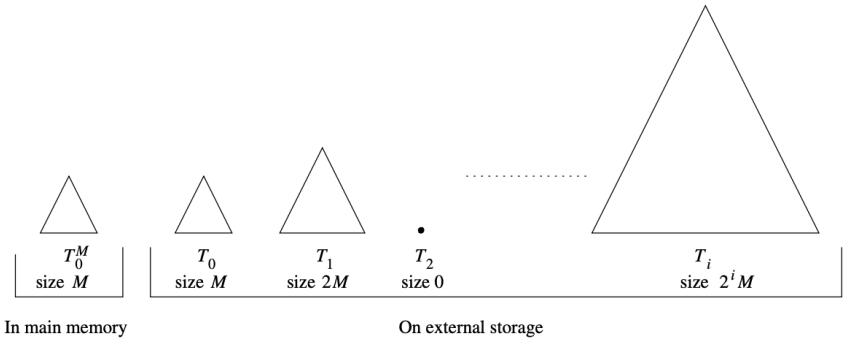


Fig. 4. The forest of trees that makes up the data structure. In this instance, T_2 is empty

对数算法

为什么要将这群KD-B-Tree设计成完全二叉树，巧妙的采用对数算法，其实就是为了尽可能提高磁盘空间利用率，参考论文中的算法论证，为了在磁盘上存储给定的kd-tree，假设需要存储的点数据数量为N，
1、假设点被打包在 N/B 个块中，也就是N / B 个叶子节点。
2、对于非叶子节点，我们执行以下算法，让 Bi 是适合一个非叶子块的点数，需要满足 N/B = Bip ，也就是P个非叶子节点，能满足指向叶子节点的指针。
对于任意P，**发现当B**i是2的一个精确幂这种情况下，内部节点可以很容易地在 O(N/(BBi)) 个块中存储，**尽量使非叶子节点数量为整数并且不造成过多的空间浪费**
因为 B * Bi = （ N / Bip ） * Bi = N * Bi 1-p ，有个计算规律，如

- 20 = 1
- 21 = 2
- 20 + 21 = 3
- 22 = 4
- 21 + 22 = 5
- ...

> 摘自论文
> 从kd-tree的根节点v开始，我们存储从v开始的广度优先搜索遍历得到的节点，直到遍历了Bi个节点。然后递归地对树的其余部分进行阻塞。使用这个程序，所有内部节点所需的块数是O(N/(BBi))，根节点到叶子节点路径（在点查询期间遍历的路径）所触及的块数是logBi(N/B)+1 = Θ(logB(N/B))。
> 如果N/B不是Bi的幂，我们填充包含kd-tree根的块，以便能够像上面那样组织树的其余部分。
> 如果N/B不是2的幂，kd-tree是不平衡的，上述阻塞算法可能会导致磁盘块的利用不足。
> 为了缓解这个问题，我们修改了kd-tree的分裂方法，在2的幂元素处分裂，而不是在中位数元素处分裂。并以 2 个元素的秩幂进行拆分，而不是在中值元素处拆分。更准确地说，当从一组 p 点构造节点 v 的两个子节点时，我们将 2 个点分配给左边的子节点，其余的分配给右边的子节点。这样，只有包含最右边路径的块（最多为 log（N/B））才能未满。从现在开始，当提到 kd 树时，我们将指如上所述存储在磁盘上的树。

对数方法（Logarithmic Method）是一种用于动态数据结构的技术，它旨在提高处理大量数据时的效率，特别是在外部存储器（如磁盘）环境中。这种方法的核心思想是将数据结构分解为多个较小的、易于管理的部分，每个部分都可以独立地进行更新和维护。这样，即使在数据量很大的情况下，也可以通过只处理数据的一个小子集来保持高效的操作。
在对数方法中，数据结构通常被组织成一系列层级，每一层都包含一组数据或索引。这些层级通常是基于对数尺度的，因此得名“对数方法”。例如，在Bkd树中，数据被分为多个块，每个块包含一定数量的点，这些块被组织成树形结构，每个树形结构包含多个层级，每个层级对应不同的数据块大小。
这种方法的优势在于：

1. **动态更新**：可以高效地处理数据的插入和删除，因为每次更新只影响数据结构的一小部分。
2. **空间利用率**：通过对数据进行分块和层级组织，可以减少存储空间的浪费，提高空间利用率。
3. **查询性能**：虽然查询可能需要访问多个层级，但由于每个层级的数据量较小，因此查询性能仍然可以保持在可接受的水平。

对数方法在处理外部存储器中的数据时特别有用，因为它可以减少对磁盘I/O操作的依赖，从而提高整体性能。在实际应用中，这种方法可以用于多种数据结构，如Bkd树、R树等，以优化它们在处理大量数据时的性能。

ps：LSM树应该也是使用的这个算法。

批量构建

> 传统上，kd树是自上而下构建的，如图 2（左列）所示。第一步是对两个坐标上的输入进行排序。然后（在第 2 步中）我们以递归方式构造节点，从根开始。对于节点 v，我们通过读取与 v 相关的两个排序集合之一的中位数来确定拆分位置（当拆分与 x 轴正交时，我们使用在 x 上排序的文件，当拆分与 y 轴正交时，我们使用在 y 上排序的文件）。最后，我们扫描这些排序的集合，并将它们中的每一个分成两个集合，然后递归地构建 v 的子集。
> 由于 N 个点上的 kd 树具有高度 log2（N/B），并且每个输入点在每一级被读取两次并写入两次，执行IO次数：O（（N/B） log2（N/B）。排序成本 O（（N/B） log M/B（N/B））次 I/O，加上排序成本 总共 O（（N/B） log2（N/B））次 I/O。

而BKD使用的并不是传统的算法，而是一种改进的散装算法Bulk Load（grid），该算法不是一次构造一层节点，而是一次构造 kd 树的整个 （log2（M/B）） 高度子树，其中M是需要存储的总点数，B为每块可以存储的点数，上述已经证明B=2是较为合适的。

Algorithm Bulk Load (binary)

(1) Create two sorted lists;
(2) Build kd-tree top-down:
Starting with the root node, do the following steps for each node, in a depth-first-search manner:
(a) Find the partitioning line;
(b) Distribute input into two sets, based on partitioning line;

Algorithm Bulk Load (grid)

(1) Create two sorted lists;
(2) Build $\log_2 t$ levels of the kd-tree:
(a) Compute t grid lines orthogonal to the x axis and t grid lines orthogonal to the y axis;
(b) Create the grid matrix A containing the grid cell counts;
(c) Create a subtree of height $\log_2 t$, using the counts in the grid matrix;
(d) Distribute input into t sets, corresponding to the t leaves;
(3) Build the bottom levels either in main memory or by recursing step (2).

Fig. 2. Two algorithms for bulk loading a kd-tree

(1) 创建两个已排序的列表；
(2) 构建t个点，构建kd-tree的log2 t层，其中t = Θ(min{M/B, √M})，其实就是需要内存M能容纳t个点

- (a) 计算t条垂直于x轴和t条垂直于y轴的网格线；
- (b) 创建包含网格单元计数的网格矩阵A，然后通过简单地扫描输入文件来计算每个网格单元中的点数。这些计数存储在一个t×t的网格矩阵A中，并保存在内部内存中（矩阵的大小t2，最多为总点数M）
- (c) 使用网格矩阵中的计数创建高度为log2 t的子树， 假设根节点使用垂直线划分点。通过首先计算（使用矩阵A中的单元格计数）包含线的垂直条带Xk，可以确定这条分割线。之后我们可以轻松地计算出定义分割的块。接下来，网格矩阵A被分裂成两个新矩阵A<和A>，分别存储来自分割线左侧和右侧的网格单元计数。这可以通过扫描垂直条带Xk的内容来完成。
- (d) 将输入分配到t个集合中，对应于t个叶子节点；

(3) 在主内存中或者通过递归步骤(2)构建底层节点。

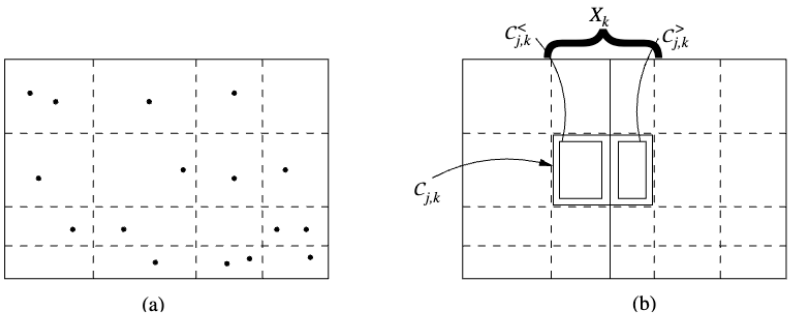


Fig. 3. Finding the median using grid cells. (a) Each strip contains N/t points. (b) Cells $C^<_{j,k}$ and $C^>_{j,k}$ are computed by splitting cell $C_{j,k}$

解释上图，使用网格单元格查找中位数，（a)矩阵的每个横、竖条带包含 N / t 个矩形，每个条带包含 N / t个点，如上图是 N / t = 4。（b）矩阵说的是将Cj,k的点按照Xk 拆分为 C j<k 和 C j > k 两个部分。构对于Xk 这条线的选取，需要通过（a）矩阵统计的点来找中位数节点，目的就是创建平衡的二叉树，

构建建算法总结：上述的构建算法简单理解就是先排序，用输入文件的三次遍历（排序，计数、读取）来构建kd树的上面log2 t层，其中t = Θ(min{M/B, √M})，具体的分步骤首先我们首先确定输入点上的一个t×t网格用于计数，扫描找出中位数点作为根节点v，然后两边递归的建树，维护的矩阵A可以统计点数量，每次找出中间点所在的X、Y轴(三维可以拓展到Z轴），然后递归创建t个点的树，对应的是log 2 t层。

> 摘自论文
> If the main memory can hold t + 1 blocks—one for each rectangle in the partition, plus one for the input—the distributioncan be done in 2N/B I/Os. This explains the choice of t = Θ(min(M/B, √M)).
> Finally, the bottom levels of the tree are constructed (in Step 3) by recursing Step 2 or, if the point set fits in internal memory, by loading it in memory and applying the binary bulk load algorithm to it.

t的要求

为什莫要求t = Θ(min(M/B, √M))？ ps："O(√M)" 表示t的增长速度不会超过内部内存大小M的平方根，这是一个渐进符号，用来描述t随着M的增长而增长的速率上限。
保

"对于我们生成的每个节点，内部内存中的矩阵A的大小增加了t个单元格" 这句话意味着，每当算法生成一个新的节点时，它都会在矩阵A中为这个节点分配t个单元格。这是因为矩阵A被用来存储与kd树的每个节点相关的信息，例如每个节点分割的点数。每个节点都需要t个单元格来存储这些信息，因为矩阵A的每个单元格代表了一个特定的数据点。

由于t ≤ O(√M)，在生成log2 t层（因为是满二叉树，这是因为每次分割都会将点集分成两部分，2 log2 t层共 t节点的树之后，看的出来树的高度（层数）与t的对数成线性关系。这意味着，即使t很大，树的高度也不会超过内部内存可以容纳的范围。因此，即使在构建了t个节点的树时候，**证为创建树节点过程中，所需要创建的单元格仍然适合内存，矩阵A的大小也不会超过内部内存的限制，从而确保了算法可以在内存中有效地运行，而不需要将数据交换到外部存储器中。**

> 如果主内存可以容纳t+1个块——每个矩形一个块，加上一个用于输入的块——那么分配可以在2N/B I/Os中完成。这解释了为什么选择t = Θ(min(M/B, √M))。

这段描述涉及到kd树批量加载算法中的一个特定步骤，即如何将输入点分配到由kd树定义的矩形区域中。在这个过程中，主内存的使用对于算法的性能至关重要。
首先，我们需要理解几个关键概念：

- 主内存：计算机中用于临时存储数据的部分，通常比磁盘存储要快得多。
- 块：数据存储的基本单位，通常是指可以一次性从磁盘读入或写入主内存的数据量。
- N：输入点的总数。
- B：可以放入一个磁盘块中的点数。
- M：主内存的容量，以点数计。
- t：用于定义网格矩阵A的尺寸，即矩阵A的行数和列数。

在这个算法的上下文中，"t+1个块" 指的是为每个矩形区域（由kd树的节点定义）分配一个块，以及一个额外的块用于存储输入数据。这意味着，如果主内存足够大，可以同时容纳这些块，那么数据的分配过程可以更高效地完成。
"2N/B I/Os" 表示完成分配过程所需的输入/输出操作次数。这是算法性能的一个关键指标，因为磁盘I/O操作通常比内存操作要慢得多。理想情况下，我们希望尽量减少磁盘I/O操作，以提高算法的效率。
选择 "t = Θ(min(M/B, √M))" 是为了平衡内存使用和算法效率。这里的 "Θ" 表示t的增长速度与 "min(M/B, √M)" 相似，也就是说，t的值不会超过这两个表达式中的较小者。这样做的目的是确保矩阵A的大小不会超过主内存的容量，同时也不会导致过多的磁盘I/O操作。

- "M/B" 表示主内存可以容纳的块数，这是一个固定的容量。
- "√M" 表示主内存容量的平方根，这通常与输入数据点集的大小相关联。

通过选择t为这两个值中的较小者，算法确保了在内存中处理数据的效率，同时避免了不必要的磁盘I/O操作。这样，算法可以在不超过主内存容量的情况下，有效地处理大规模数据集。

在论文中提到的 "2N/B" 表示的是在执行数据分配过程中所需的磁盘输入/输出（I/O）操作次数。这个数字是用来估算在将输入数据点分配到kd树定义的矩形区域时，需要从磁盘读取和写入数据的次数。这

为什么是 "2N/B" 呢？这是因为在批量加载kd树的过程中，每个数据点需要被读取两次：

- 第一次读取是统计每个小矩形中点数量。这涉及到对输入数据点进行扫描，以计算每个网格单元格中的点数，并将这些点数存储在矩阵A中。
- 第二次读取是为了根据矩阵A中的信息，再次读取数据点，并根据它们所属的区域将它们写入到相应的磁盘块中。

因此，对于每个数据点，都需要两次磁盘I/O操作：一次读取和一次写入。由于有N个数据点，每个数据点需要2次I/O操作，所以总共需要的I/O操作次数是 "2N"。但是，由于每个磁盘块可以容纳B个数据点，所以每次I/O操作可以处理B个数据点。因此，总的I/O操作次数是 "2N" 除以 "B"，即 "2N/B"。
这个计算假设了数据点均匀分布在磁盘块中，且每次I/O操作都能最大化地利用磁盘块的容量。在实际情况中，可能会有一些额外的I/O操作，例如处理磁盘块的元数据或者处理不完全填满的磁盘块。但是，"2N/B" 提供了一个理论上的下限估算，用于理解在理想情况下算法的I/O复杂度。

动态更新

Bkd树（Block-kd tree）是一种由多个kd树组成的数据结构，用于在平面上索引N个点。这种结构特别适合于外部存储器环境，如磁盘，因为它可以有效地处理大量数据。

- N：数据集中点的总数。
- M：内部内存缓冲区的容量，以点数计。
- Bkd树：由log2(N/M)个kd树组成的数据结构，每个kd树都可能包含不同数量的点。
- T：表示Bkd树中的第i个kd树。
- Ti：第i个kd树，它要么是空的，要么包含2^i * M个点。
- T0：第一个kd树，它在内部存储器中，最多包含M个点。

Bkd树的组织方式允许它在执行插入和查询操作时保持高空间利用率和良好的性能。这种组织方式类似于对数方法，这是一种用于动态数据结构的技术，它可以在保持数据结构性能的同时，有效地处理数据的插入和删除。
下图展示Bkd树的结构，包括如何在内部存储器和外部存储器之间分配不同的kd树。这种结构的设计目的是为了优化I/O操作，因为在外部存储器上进行I/O操作通常比在内部存储器上要慢得多。
总的来说，Bkd树是一种能够动态扩展以适应大量数据的多维索引结构，它通过将数据分散到多个kd树

中来提高性能，同时保持了高空间利用率和对数级别的更新性能。

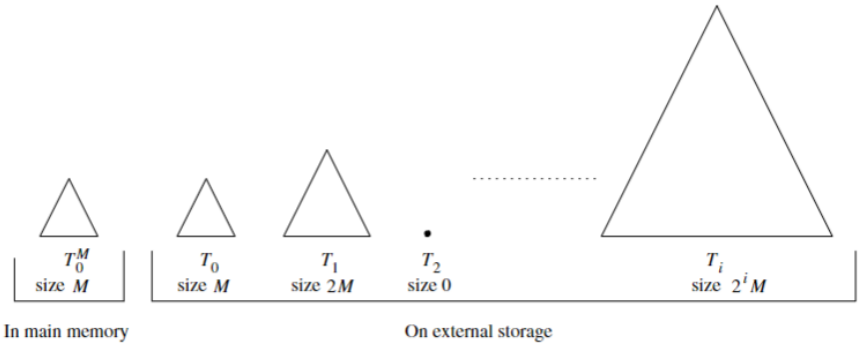


Fig. 4. The forest of trees that makes up the data structure. In this instance, T_2 is empty

插入与删除算法

Algorithm Insert(p)	**Algorithm Delete**(p)
(1) Insert p into in-memory buffer T_0^M; (2) If T_0^M is not full, return; otherwise, find the first empty tree T_k and extract all points from T_0^M and T_i, $0 \leq i < k$ into a file F; (3) Bulk load T_k from the items in F; (4) Empty T_0^M and T_i, $0 \leq i < k$.	(1) Query T_0^M with p; if found, delete it and return; (2) Query each non-empty tree in the forest (starting with T_0) with p; if found, delete it and return;

Fig. 5. The **Insert** and **Delete** algorithms for the Bkd-tree

插入算法，插入一个点p

（1） 将 p 插入内存缓冲区 T0M 中

（2）如果 T0M 未满，则返回;否则，找到第一个空树 Tk，从 T0M 和 Ti （0 <= i < k)中提取所有点，写入磁盘File

（3）从磁盘File中的元素中批量加载Tk;

（4） 清空 T0M 和 Tk，0 ≤ i < k。

删除算法，删除一个点p

（1）用p查询 T0M;如果找到，请将其删除并返回;

（2）用p查询森林中每棵非空树（以T0开头）;如果找到，请将其删除并返回;

The algorithms for inserting and deleting a point are outlined in Figure 5. The simplest of the two is the deletion algorithm. We simply query each of the trees to find the tree T_i containing the point and delete it from T_i. Since there are at most $\log_2(N/M)$ trees, the number of I/Os performed by a deletion is $O(\log_B(N/B)\log_2(N/M))$.

两者中最简单的是删除算法。我们只需查询每棵树，找到包含该点的树 Ti，然后将其从 Ti 中删除。由于最多有 log（N/M ） 树，因此删除执行的 I/O 数为 O（log（N/B） log（N/M ））。

> The insertion algorithm is fundamentally different. Most insertions ($M-1$ out of M consecutive ones) are performed directly on the in-memory structure T_0^M. Whenever T_0^M becomes full, we find the smallest k such that T_k is an empty kd-tree. Then we extract all points from T_0^M and T_i, $0 \leq i < k$, and bulk load the tree T_k from these points. Note that the number of points now stored in T_k is indeed $2^k M$ since T_0^M stores M points and each T_i, $1 \leq i < k$, stores exactly $2^i M$ points (T_k was the first empty kd-tree). Finally, we empty T_0^M and T_i, $0 \leq i < k$. In other words, points are inserted in the in-memory structure and gradually "pushed" towards larger kd-trees by periodic reorganizations of small kd-trees into one large kd-tree. The larger the kd-tree, the less frequently it needs to be reorganized.
>
> To compute the amortized number of I/Os performed by one insertion, consider N consecutive insertions in an initially empty Bkd-tree. Whenever a new kd-tree T_k is constructed, it replaces all kd-trees T_j, $1 \leq j < k$, and the in-memory structure T_0^M. This operation takes $O((2^k M/B) \log_{M/B}(2^k M/B))$ I/Os (bulk loading T_k) and moves exactly $2^k M$ points into the larger kd-tree T_k. If we divide the construction of T_k between these points, each of them has to pay $O((1/B) \log_{M/B}(2^k M/B)) = O((1/B) \log_{M/B}(N/B))$ I/Os. Since points are only moving into larger kd-trees, and there are at most $\log_2(N/M)$ kd-trees, a point can be charged at most $\log_2(N/M)$ times. Thus the final amortized cost of an insertion is $O\left(\frac{\log_{M/B}(N/B) \log_2(N/M)}{B}\right)$ I/Os.

插入算法则完全不同。大多数插入操作（M-1次连续插入中的M次）直接在内部存储器结构TM0上执行。每当TM0变满时，我们找到最小的k，使得Tk是一个空的kd树。然后我们从TM0和Ti（0 ≤ i < k）中提取所有点，并批量加载这些点到树Tk中。

注意，现在存储在Tk中的点数确实是2^k * M，因为TM0存储M个点，而每个Ti（1 ≤ i < k）存储恰好2^i * M个点（Tk是第一个空的kd树）。就是上面巧妙的对数算法。

最后，我们清空TM0和Ti（0 ≤ i < k）。

换句话说，点首先被插入到内部存储器结构中，然后逐渐"推"向更大的kd树，通过定期重组小kd树到一个大的kd树。kd树越大，需要重新组织的次数就越少。

为了计算一次插入操作的摊销I/O次数，考虑在最初为空的Bkd树中连续插入N个点。每当构建一个新的kd树Tk时，它将替换所有kd树Tj（1 ≤ j < k），以及内部存储器结构TM0。这个操作需要O((2^k * M/B) logM/B(2^k * M/B)) I/Os（批量加载Tk），并将恰好2^k * M个点移动到更大的kd树Tk中。如果我们将构建Tk的操作分配给这些点，每个点最多需要支付O((1/B) logM/B(2^k * M/B)) = O((1/B) logM/B(N/B)) I/Os。

由于点只移动到更大的kd树中，并且最多有log2(N/M)个kd树，因此一个点最多被花费log2(N/M)次。因此，一次插入操作的最终摊销成本是O(logM/B(N/B) log2(N/M)) I/Os。

查询

查询的话就是从森林中依次查询。

要对Bkd树进行窗口查询，我们只需查询所有的log2(N/M)个kd树。在一个存储N个点的kd树上进行窗口查询的最坏情况性能是O(√(N/B) + K/B) I/Os，其中K是查询窗口中的点数。

由于构成Bkd树的kd树在几何上是逐渐增大的，Bkd树的窗口查询的最坏情况性能也是O(√(N/B) + K/B) I/Os。然而，由于kd树的平均窗口查询性能通常比最坏情况性能要好得多[24]，因此重要的是要研究使用多个kd树对Bkd树与kd树相比的实际性能有何影响。

2.3、论文实验结果

下面内容翻译自论文，主要围绕BKD的空间利用率、批量构建性能、插入性能、查询性能与多个KDB-Tree和hTree做对比

论文中的实验结果部分详细地展示了Bkd树在多个方面的性能表现，并通过与其他数据结构的比较来验证其效率。以下是对实验结果部分的详细总结：

1. 空间利用率：

- Bkd树实现了接近100%的空间利用率，这表明它能够高效地使用存储空间来存储数据点。
- K-D-B树和hBΠ树的空间利用率明显较低，特别是K-D-B树，在最坏情况下空间利用率可以降至28%，而hBΠ树可以降至36%。这说明这些结构对于数据分布和插入顺序非常敏感。

2. 批量加载性能：

- 论文比较了两种kd树批量加载算法：二分法和网格法。网格法在构建时间上至少比二分法快两倍，并且随着数据集大小的增加，网格法的速度优势更加明显。
- 在I/O次数上，网格法比二分法节省了大约三分之一的I/O操作，这意味着网格法在磁盘读写操作上更加高效。

3. 插入性能：

- Bkd树在插入操作上比K-D-B树快得多。在平均情况下，Bkd树的插入速度可以比K-D-B树快100倍，这主要是由于Bkd树的动态重组机制，它将点逐渐从内存结构转移到更大的磁盘结构中。
- 对于TIGER数据集，由于数据的局部性，K-D-B树在某些情况下可能在时间上表现得更好，但Bkd树在I/O次数上仍然更优。

4. 查询性能：

- 尽管Bkd树需要查询多个kd树，但其窗口查询性能与K-D-B树相当。实验结果显示，Bkd树在处理大规模数据集时能够在保持高查询性能的同时，显著提高插入性能和空间利用率。
- 在不同大小的查询窗口上，Bkd树和K-D-B树的性能几乎相同，这表明Bkd树在查询性能上没有牺牲。

5. 实验设置：

- 实验使用了不同类型的数据集，包括从TIGER/Line数据生成的实际道路特征点、均匀分布的点和沿正方形对角线分布的点。
- 实验在一台配备Pentium III/500MHz处理器和36GB SCSI硬盘的工作站上进行，操作系统为FreeBSD 4.3。

6. 实验结论：

- Bkd树在插入性能和空间利用率方面显著优于K-D-B树和hBΠ树，尤其是在处理大量数据时。
- Bkd树的查询性能与K-D-B树相当，即使需要查询多个kd树，也不会显著影响查询效率。
- 网格法批量加载算法在I/O效率和运行时间上都优于二分法，特别是在处理大规模数据集时。

综上所述，Bkd树作为一种新型的动态多维索引结构，能够有效地处理和索引大型数据集，同时保持良好的查询和更新性能，使其成为空间数据库中一个有价值的工具。

参考

- kd-tree-维基百科
- 论文 Bkd-Tree: A Dynamic Scalable kd-Tree
- K-D树、K-D-B树、B-K-D树
- 空间索引技术在58搜索中的落地实践 – BKD技术原理深入剖析_ITPUB博客
- 从K近邻算法、距离度量谈到KD树、SIFT+BBF算法_kd树标准差-CSDN博客