2022

K-D Trees.

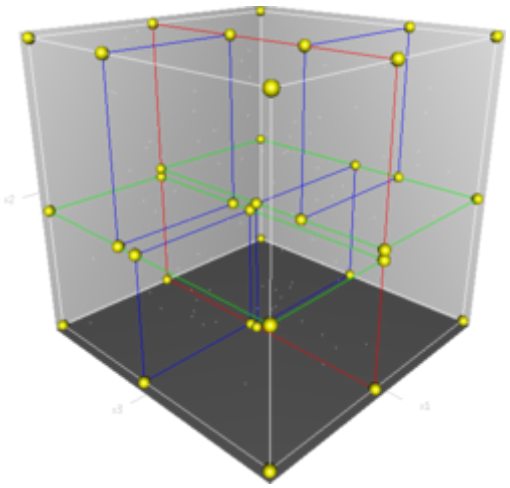
ESTRUCTURAS DE DATOS - UCLM

ALFONSO SEBASTIAN SIMORTE RAMOS
AMPARO NATIVIDAD MENDOZA ESCRIBANO
SARA HIDALGO SALAS
KEVIN ALFONSO GÓMEZ SANDOVAL
PAULINO ESTEBAN BERMÚDEZ RODRÍGUEZ

Contenido

Introducción.

Los algoritmos pueden ser difíciles de implementar y propone una forma de manejar la complejidad de un algoritmo mediante el desarrollo en etapas, comenzando desde una versión simple y funcional, y progresando a optimizaciones cada vez más complejas, manteniendo la corrección del programa en cada paso.



Encapsular el comportamiento complejo debajo de una interfaz simple.

Las abstracciones mejoradas ocultan ejecuciones complejas debajo de una interfaz sencilla. Los programas a menudo son gradualmente más complicados, aumentando la dificultad de las estructuras de datos implementadas, es por ello que, administramos la complejidad mientras mantenemos la ilusión de ser algo sencillo.

> "*Un árbol K-D (K-Dimensional o K-Dimension) es una estructura de datos de partición de espacio usada para almacenar y recuperar información de manera eficiente, preparada para organizar puntos en un espacio k-dimensional.*"

Los K-D Tree son usados normalmente en problemas de búsqueda espacial, como encontrar puntos cerca en un espacio multidimensional o realizar consultas de rangos en un conjunto de puntos.

Árbol K-D.

Un árbol K-D es un tipo de árbol binary tree, donde se tiene a cada nodo como representante de un punto en un espacio K-dimensional.

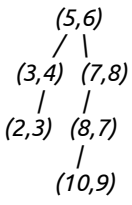
Los nodos del árbol se dividen en dos grupos usando como referencia un plano que corta a través del espacio K-dimensional. Los nodos a un lado del plano tienen un valor menor en la dimensión que se está dividiendo, mientras que los nodos del otro lado tienen un valor mayor. Esto permite que el árbol K-D se divida en regiones separadas del espacio, lo que facilita la búsqueda de puntos específicos.

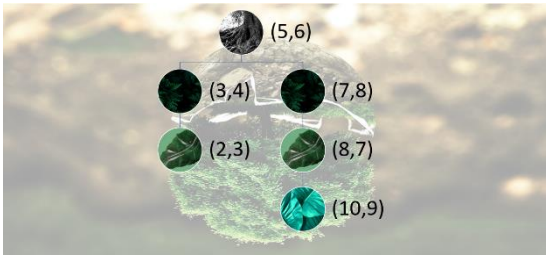
Un árbol K-D es muy útil cuando se trata de realizar búsquedas en espacios multidimensionales, ya que permite la búsqueda eficiente de puntos o un rango de puntos. Además, los árboles K-D son *relativamente fáciles de implementar* y pueden ser usados en una amplia variedad de aplicaciones.

Un ejemplo de árbol de este tipo sería:

Supongamos que queremos construir un árbol K-D para almacenar un conjunto de puntos en un espacio bidimensional.
Primero, seleccionamos un punto del conjunto de manera aleatoria y lo colocamos en el nodo raíz del árbol. A continuación, dividimos el espacio en dos regiones usando una línea que pasa a través del punto seleccionado.
Los puntos a un lado de la línea tienen un valor menor en la dimensión que estamos dividiendo, mientras que los puntos del otro lado tienen un valor mayor.
A continuación, seleccionamos otro punto del conjunto y lo colocamos en el nodo izquierdo del árbol si tiene un valor menor en la dimensión que estamos dividiendo, o en el nodo derecho si tiene un valor mayor.
Por último, repetimos este proceso para cada uno de los puntos restantes del conjunto, dividiendo el espacio en dos regiones cada vez que añadimos un nuevo punto al árbol.

Finalmente, obtendríamos un árbol similar a este:





En este ejemplo, el punto (5,6) es el nodo raíz del árbol. Los puntos (3,4) y (7,8) son los hijos del nodo raíz, y así sucesivamente. Cada vez que dividimos el espacio en dos regiones, utilizamos una línea vertical para representar la división en la primera dimensión y una línea horizontal para representar la división en la segunda dimensión.

Una vez que se ha construido el árbol K-D, podemos utilizarlo para realizar búsquedas eficientes en el conjunto de puntos.
Podríamos buscar todos los puntos dentro de un rango específico o encontrar el punto más cercano de una ubicación.
Aunque podría parecer una versión generalizada de quadtree y octree, su implementación es diferente. Cualquier nodo interno en esta estructura divide el espacio en 2 mitades. El hijo izquierdo del nodo representa la mitad izquierda, mientras que el hijo derecho representa la mitad derecha.

El espacio se divide en 2 mitades independientemente del número de dimensiones. Para ser más precisos, cada nodo interno representa un hiperplano que corta el espacio en 2 partes. Para el espacio de 2 dimensiones, eso es una línea y para el espacio de 3 dimensiones, eso es un plano.
Cada nodo del árbol representa un punto en el espacio.

El procedimiento general para construir un árbol k-d es dividir recursivamente el espacio en dos a lo largo del eje que tiene una mayor extensión. Cada nodo del árbol indica a lo largo de que dimensión el nodo que dividió el espacio.

Construcción.

La construcción de un árbol K-D estático implica crear el árbol con un conjunto de puntos fijo.
Luego, realizar búsquedas en el árbol sin modificar los puntos del árbol, esto es en contraposición a la construcción dinámica de un árbol K-D, en la que se pueden realizar operaciones de inserción y eliminación de puntos en el árbol.

La ventaja de la construcción estática de un árbol K-D es que es más rápida y eficiente en términos de memoria que la construcción dinámica, ya que no se requiere almacenar información adicional para realizar operaciones de inserción y eliminación. Sin embargo, la desventaja es que no se pueden realizar operaciones de inserción y eliminación en el árbol una vez que se ha construido.

Ejemplo de construcción de un K-D Tree estático:

```
import kdtrees.KDTree;

public class Main {
  public static void main (String[] args) {
    // Crea una instancia de KDTree con un conjunto de puntos
    KDTree<Double> tree = new KDTree<Double>(2);
    tree.insert(new double[] {0, 0}, 0.0);
    tree.insert(new double[] {1, 0}, 1.0);
    tree.insert(new double[] {0, 1}, 2.0);
    tree.insert(new double[] {1, 1}, 3.0);
  }
}
```

En este ejemplo, se crea una instancia de KDTree con dos dimensiones y se insertan algunos puntos en el árbol. Una vez que el árbol se ha construido, puedes realizar búsquedas de puntos cercanos utilizando los métodos de búsqueda proporcionados por la biblioteca.

Algoritmo.

**) Consideramos un espacio de dos dimensiones, pero se pueden aplicar a cualquier espacio. Estas son las principales tareas del K-D tree:*

Búsqueda.

Esta función comprueba si el punto existe en el espacio. Comenzando con el nodo raíz como nodo actual.

1. Si el nodo actual representa el punto (x,y) devuelve ‘true’
2. Si el nodo actual no es un nodo hoja, vaya al paso siguiente, de lo contrario devuelve ’false’
3. Sea el nodo actual el punto (X,Y) si el nodo divide el espacio a lo largo del eje x, compare x con X. Si x < X, establezca el nodo actual como hijo izquierdo, de lo contrario, establezca el nodo actual como hijo derecho. Si el nodo dividió el espacio a lo largo del eje y, compare y e Y. Vuelva al paso 1.

Insertar.

Cada operación de inserción divide el espacio.

1. Busque en el árbol (x,y) hasta que se alcance un nodo hoja.
2. Si el árbol está vacío, agregue un nuevo nodo como raíz que represente el punto (x,y). Aquí, el espacio se puede dividir a lo largo de cualquier eje. Indique el eje a lo largo de cual se divide el espacio y finalice la inserción.
3. Inserte un nuevo nodo donde el punto (x,y) debería haber existido y haga que almacene (x,y). Si el padre dividió el espacio a lo largo del eje x, hafa que el punto divida el espacio a lo largo del eje y, de lo contrario, haga que divida el espacio a lo largo del eje x.

Algortimo	Promedio	Peor de los casos
Espacio	O(n)	O(n)
Buscar	O(log n)	O(n)
Insertar	O(log n)	O(n)
Borrar	O(log n)	O(n)

En caso de que el árbol se vaya a construir a partir de un conjunto dado de puntos, la estrategia a seguir es encontrar el punto medio con respecto al espacio a dividir. Inserte ese punto usando el método anterior y repita para encontrar nodos secundarios.

Considere la inserción de puntos:

(5, 25), (15, 55), (30, 40), (35, 20), (50, 50) en orden. Se puede ilustrar como:

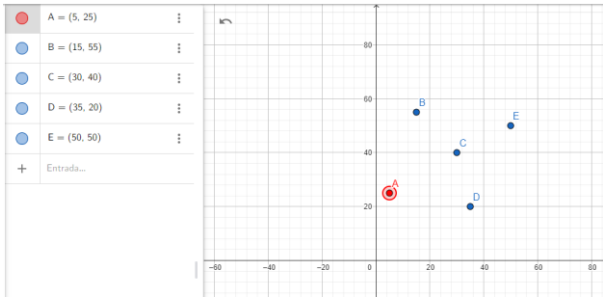


Ilustración 1insert (5,25)

Tomamos el punto Y (15,55). Y desde X (5,25) apuntamos a él.

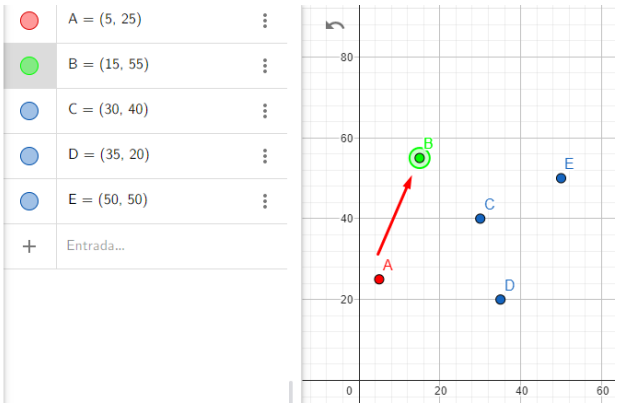


Ilustración 2 insert (15,55)

Seguimos con el siguiente punto (C).

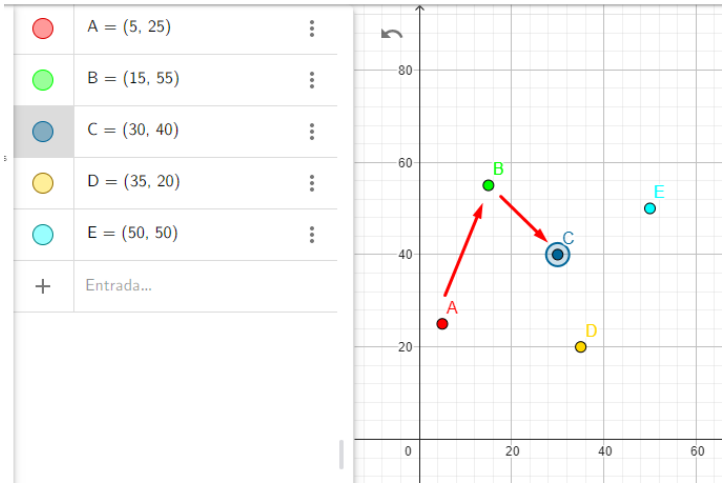


Ilustración 3 insert (30,40)

Continuamos con D.

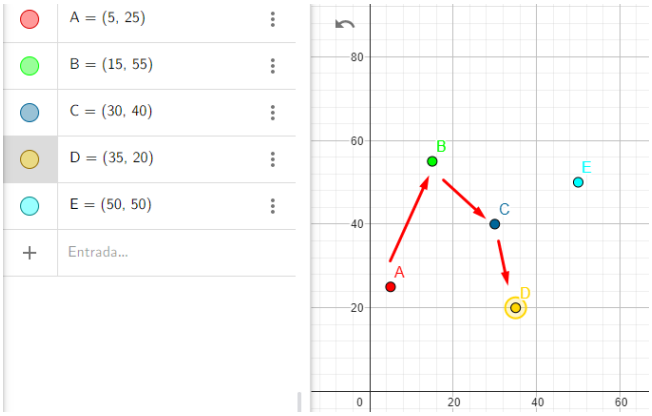


Ilustración 4 Insert (35,20)

Por último, a partir del punto (35,20) como asignación X apuntamos hacia el último (50,50) con asignación Y.

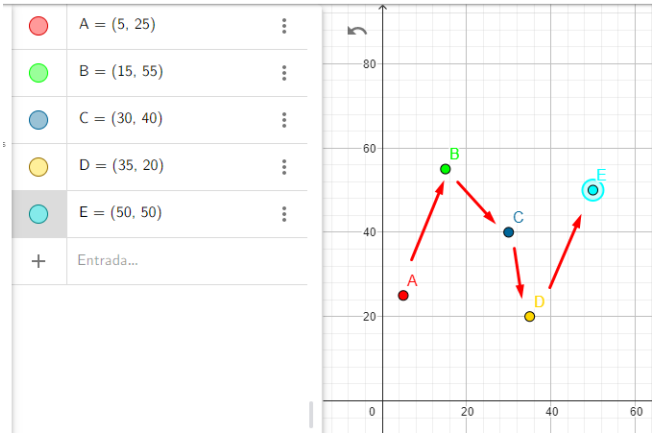


Ilustración 5 Insert (35,20)

Es importante tener en cuenta que la construcción del árbol a partir de un conjunto dado de puntos da un árbol equilibrado, mientras que no exista tal garantía en las inversiones consecutivas. Usando la estrategia mediana, el árbol se vería así:

Equilibrio.

El equilibrio de un K-D se refiere a la distribución de puntos entre los nodos del árbol. Un árbol equilibrado es un árbol en el que cada nodo tiene aproximadamente el mismo número de puntos hijos.

Esto se consigue dividiendo los puntos de tal manera que cada nodo tenga un número similar de puntos.

Un K-D desequilibrado, por el contrario, es un árbol en el que algunos nodos tienen muchos más puntos que otros. Esto puede ocurrir si los puntos se dividen de manera desproporcionada entre los nodos, por ejemplo, si la mayoría de los puntos caen en una sola región del espacio.

El árbol K-D equilibrado tiene varias ventajas sobre un árbol K-D desequilibrado:

Mayor eficiencia: Un árbol K-D equilibrado tiene menos nodos en total que un árbol K-D desequilibrado, lo que significa que las operaciones de búsqueda y inserción son más rápidas.

Mayor escalabilidad: Un árbol K-D equilibrado puede manejar conjuntos de datos más grandes sin sufrir una degradación significativa del rendimiento.

Mayor precisión: Un árbol K-D equilibrado es menos propenso a producir resultados inexactos en las búsquedas de puntos cercanos debido a la distribución desproporcionada de puntos en el árbol.

Para mantener el equilibrio de un árbol, es importante utilizar un algoritmo de construcción que tenga en cuenta la distribución de puntos en el espacio. Algunos algoritmos, como el algoritmo de construcción de árboles K-D mediano, se diseñan específicamente para mantener el equilibrio del árbol.

Complejidad.

Complejidad temporal.

Depende principalmente del tamaño del conjunto de puntos que se está almacenando y de la forma en que se está usando el árbol.

La complejidad temporal de la inserción de un nuevo punto en el árbol es de

- Insertar: θ(log2N), O(N)

Donde N es el número de puntos almacenados en el árbol. Ya que cada inserción requiere recorrer un número constante de nodos del árbol, lo que implica un tiempo de ejecución lineal en el tamaño del árbol.

La complejidad temporal de la búsqueda de puntos cercanos a una ubicación dada también es de

- Buscar: θ(log2N), O(N)

Se debe a que la búsqueda se realiza recorriendo el árbol de manera similar a la inserción, lo que implica un tiempo de ejecución lineal en el tamaño del árbol.

En general, el árbol K-D es muy eficiente para realizar búsquedas espaciales en conjunto de puntos de alta dimensión. Sin embargo, es importante tener en cuenta que la complejidad temporal puede aumentar si se realizan muchas inserciones o búsquedas en el árbol. También es posible que la complejidad temporal sea mayor si el árbol no está equilibrado adecuadamente.

- Complejidad del espacio: O(n).

Es importante tener en cuenta que el K-D puede consumir una gran cantidad de memoria si se están almacenando muchos puntos en él. Por lo tanto, es importante tener en cuenta la complejidad del espacio al usar este tipo de estructura de datos.

Degradación del rendimiento del K-D por datos de alta densidad o puntos de consulta lejanos a los puntos del K-D buscados.

Esto se debe a que el árbol K-D usa la técnica de divide y vencerás para organizar los datos, lo que implica: *dividir el espacio en regiones cada vez más pequeñas a medida que se añaden más dimensiones.*

A medida que aumenta el número de dimensiones, el espacio se divide en regiones cada vez más pequeñas y el K-D se vuelve más profundo. Esto puede llevar a una degradación del rendimiento al realizar inserciones o búsquedas en el árbol, ya que requiere más tiempo para recorrer un árbol más profundo y con altas probabilidades de que esté desequilibrado.

Si se sabe de antemano que las consultas de búsqueda se realizarán a gran distancia de los puntos almacenados en el árbol, puede ser más adecuado utilizar una estructura de datos diferente, como un índice R-Tree o un cubo espacial. Estas estructuras de datos pueden ser

más eficientes en este tipo de escenarios debido a su capacidad para almacenar puntos de manera más densa y reducir el número de nodos que se tienen que recorrer durante la búsqueda.

Sin embargo, el árbol K-D sigue siendo una estructura de datos muy eficiente para realizar búsquedas en conjuntos de puntos de alta dimensión.

Variaciones comunes del árbol K-D.

Hay algunas variaciones comunes del árbol K-D que se utilizan en diferentes escenarios:

Árbol K-D equilibrado.

Variación del árbol K-D usada para garantizar que el árbol esté equilibrado, mejorando la complejidad temporal de las operaciones de búsqueda e inserción. Un árbol equilibrado se construye de tal manera que cada nodo tenga aproximadamente el mismo número de puntos hijos.

Árbol K-D orientado al rendimiento.

Variación del árbol K-D usada para mejorar el rendimiento de la búsqueda de puntos cercanos. Un árbol K-D orientado al rendimiento se construye de tal manera que los nodos se dividan de tal manera que los puntos cercanos se encuentren en nodos hijos más cercanos.

Árbol K-D estático.

Variación del árbol K-D en la que se asume que el conjunto de puntos no cambiará después de que se construya el árbol. Un árbol K-D estático se construye de manera más eficiente que un árbol K-D dinámico, ya que no se necesitan tener en cuenta las operaciones de inserción y eliminación.

Árbol K-D dinámico.

Variación del árbol K-D en la que se permite que el conjunto de puntos cambie después de que se construya el árbol. Un árbol K-D dinámico se puede construir de manera más eficiente utilizando técnicas como el rebalanceo del árbol o la re-compresión del árbol.

Implementación en código.

Java.

```java
import java.util.List;

public class KDNode {
  // Coordenadas del punto almacenado en este nodo
  private double[] point;
  // Índice de la dimensión que se está dividiendo
  private int depth;
  // Nodo izquierdo y derecho del árbol
  private KDNode left;
  private KDNode right;

  public KDNode(double[] point, int depth) {
    this.point = point;
    this.depth = depth;
  }

  // Método para insertar un punto en el árbol
  public void insert(double[] point) {
    // Determine which side of the dividing plane the point belongs to
    int dim = depth % point.length;
    if (point[dim] < this.point[dim]) {
      if (left == null) {
        left = new KDNode(point, depth + 1);
      } else {
        left.insert(point);
      }
    } else {
      if (right == null) {
        right = new KDNode(point, depth + 1);
      } else {
        right.insert(point);
      }
    }
  }

  // Método para buscar puntos cercanos a una ubicación dada
  public List<double[]> search(double[] target, double radius) {
    // TODO: implement search method
  }
}
```

Para usar este árbol K-D, primero se necesita crear una instancia de KDNode y pasarle un punto para almacenar en el nodo raíz del árbol. A continuación, podría usar el método 'insert' para añadir puntos adicionales al árbol y el método 'search' para buscar puntos cercanos a una ubicación dada.

C++

```cpp
#include <vector>

class KDNode {
 public:
  // Coordenadas del punto almacenado en este nodo
  std::vector<double> point;
  // Índice de la dimensión que se está dividiendo
  int depth;
  // Nodo izquierdo y derecho del árbol
  KDNode* left;
  KDNode* right;

  KDNode(std::vector<double> point, int depth) : point(point), depth(depth), left(nullptr),
right(nullptr) {}

  // Método para insertar un punto en el árbol
  void insert(std::vector<double> point) {
    // Determine which side of the dividing plane the point belongs to
    int dim = depth % point.size();
    if (point[dim] < this->point[dim]) {
      if (left == nullptr) {
        left = new KDNode(point, depth + 1);
      } else {
        left->insert(point);
      }
    } else {
      if (right == nullptr) {
        right = new KDNode(point, depth + 1);
      } else {
        right->insert(point);
      }
    }
  }

  // Método para buscar puntos cercanos a una ubicación dada
  std::vector<std::vector<double>> search(std::vector<double> target, double radius) {
    // TODO: implement search method
  }
};
```

Cumple las mismas necesidades que en el código para Java, necesita de la instancia KDNode y pasarle un punto para almacenar en el nodo raíz del árbol. Pudiéndose, luego usar los métodos insert y search para los puntos que se indiquen.

Python.

```python
class KDNode:
    # Coordenadas del punto almacenado en este nodo
    point = []
    # Índice de la dimensión que se está dividiendo
    depth = 0
    # Nodo izquierdo y derecho del árbol
    left = None
    right = None

    def __init__(self, point, depth):
        self.point = point
        self.depth = depth

    # Método para insertar un punto en el árbol
    def insert(self, point):
        # Determine which side of the dividing plane the point belongs to
        dim = self.depth % len(self.point)
        if point[dim] < self.point[dim]:
            if self.left is None:
                self.left = KDNode(point, self.depth + 1)
            else:
                self.left.insert(point)
        else:
            if self.right is None:
                self.right = KDNode(point, self.depth + 1)
            else:
                self.right.insert(point)

    # Método para buscar puntos cercanos a una ubicación dada
    def search(self, target, radius):
        # TODO: implement search method
```

Estos son solo una forma posible de implementar un árbol K-D en distintos lenguajes de programación, pero hay muchas otras formas de hacerlo y puede haber algunas variaciones en la forma en que se implementen los métodos y se almacenen los datos.

Aplicaciones

El uso de los árboles K-D está muy extendido gracias a su alta eficiencia, de tal forma que podemos encontrar aplicaciones como:

- Biblioteca KDTree de Python: Esta es una implementación de árbol K-D que se incluye en la biblioteca estándar de Python. Esta implementación utiliza un enfoque dinámico y permite realizar búsquedas y operaciones de inserción en el árbol. Ejemplo:

```python
# Ejemplo de busqueda de puntos crecanos en un conjunto de puntos.
from scipy.spatial import KDTree

# Crea una instancia de KDTree con un conjunto de puntos
points = [[0, 0], [1, 0], [0, 1], [1, 1]]
tree = KDTree(points)

# Realiza una búsqueda de puntos cercanos a una ubicación dada
query_point = [0.5, 0.5]
distance, indices = tree.query(query_point, k=2)

# Imprime los puntos más cercanos y sus distancias
print(f"Los puntos más cercanos son: {points[indices[0]]} y {points[indices[1]]}")
print(f"Las distancias son: {distance[0]} y {distance[1]}")
```

- Árbol K-D de Boost: Esta es una implementación de árbol K-D que se incluye en la biblioteca Boost de C++. Esta implementación utiliza un enfoque estático y permite realizar búsquedas y operaciones de inserción en el árbol. Ejemplo:

```cpp
// Ejemplo de K-D Tree de Boost para realizar búsqueda de puntos cercanos
// en un conjunto de puntos.
//
// Se crea una instancia de RTree y se inserta algunos puntos en el árbol
// A continuación, se realiza una b´usqueda de los puntos más cercanos.

#include <boost/geometry.hpp>
#include <boost/geometry/index/rtree.hpp>

namespace bg = boost::geometry;
namespace bgi = boost::geometry::index;

int main() {
  // Crea una instancia de RTree con un conjunto de puntos
  typedef bg::model::point<float, 2, bg::cs::cartesian> point;
  typedef std::pair<point, unsigned> value;
  bgi::rtree<value, bgi::rstar<16>> rtree;
  rtree.insert(value(point(0, 0), 0));
  rtree.insert(value(point(1, 0), 1));
  rtree.insert(value(point(0, 1), 2));
  rtree.insert(value(point(1, 1), 3));
```

```
  // Realiza una búsqueda de puntos cercanos a una ubicación dada
  point query_point(0.5, 0.5);
  std::vector<value> result_n;
  rtree.query(bgi::nearest(query_point, 2), std::back_inserter(result_n));

  // Imprime los puntos más cercanos y sus distancias
  std::cout << "Los puntos más cercanos son: " << bg::wkt<point>(result_n[0].first) << " y "
            << bg::wkt<point>(result_n[1].first) << std::endl;
  std::cout << "Las distancias son: " << bg::distance(query_point, result_n[0].first) << " y "
            << bg::distance(query_point, result_n[1].first) << std::endl;
  return 0;
}
```

- flann: Esta es una biblioteca de C++ que proporciona una implementación de árbol K-D y otras estructuras de datos para realizar búsquedas espaciales. Esta biblioteca incluye varias opciones de configuración y está diseñada para ser rápida y eficiente en términos de memoria.

```
#include <flann/flann.hpp>
#include <flann/io/hdf5.h>

int main() {
  // Crea una instancia de flann con un conjunto de puntos
  flann::Matrix<float> dataset;
  dataset.rows = 4;
  dataset.cols = 2;
  dataset[0][0] = 0;
  dataset[0][1] = 0;
  dataset[1][0] = 1;
  dataset[1][1] = 0;
  dataset[2][0] = 0;
  dataset[2][1] = 1;
  dataset[3][0] = 1;
  dataset[3][1] = 1;
  // Construye un árbol K-D con los puntos
  flann::Index<flann::L2<float>> index(dataset, flann::KDTreeIndexParams(4));
  index.buildIndex();
  // Realiza una búsqueda de puntos cercanos a una ubicación dada
  float query[2] = {0.5, 0.5};
  flann::Matrix<float> query_matrix(query, 1, 2);
  flann::Matrix<int> indices(new int[query_matrix.rows * 2], query_matrix.rows, 2);
  flann::Matrix<float> dists(new float[query_matrix.rows * 2], query_matrix.rows, 2);
  index.knnSearch(query_matrix, indices, dists, 2, flann::SearchParams(128));
  // Imprime los puntos más cercanos y sus distancias
  std::cout << "Los puntos más cercanos son: (" << dataset[indices[0][0]][0] << ", " <<
dataset[indices[0][0]][1]
            << ") y (" << dataset[]
```

- ELKI: Esta es una biblioteca de Java que proporciona una implementación de árbol K-D y otras estructuras de datos para realizar búsquedas espaciales. Esta biblioteca incluye varias opciones de configuración y está diseñada para ser rápida y escalable.

```
import de.lmu.ifi.dbs.elki.database.Database;
import de.lmu.ifi.dbs.elki.database.StaticArrayDatabase;
import de.lmu.ifi.dbs.elki.database.ids.DBIDIter;
import de.lmu.ifi.dbs.elki.database.ids.DBIDUtil;
import de.lmu.ifi.dbs.elki.database.ids.DBIDs;
import de.lmu.ifi.dbs.elki.database.query.knn.KNNQuery;
import de.lmu.ifi.dbs.elki.database.relation.Relation;
import de.lmu.ifi.dbs.elki.distance.distancefunction.EuclideanDistanceFunction;
import de.lmu.ifi.dbs.elki.index.tree.metrical.mtreevariants.MTree;
import de.lmu.ifi.dbs.elki.index.tree.metrical.mtreevariants.MTreeFactory;
import de.lmu.ifi.dbs.elki.math.geometry.SpatialComparable;
import de.lmu.ifi.dbs.elki.math.geometry.SpatialUtil;
import de.lmu.ifi.dbs.elki.math.linearalgebra.
```

Implementaciones de código abierto

- ALGLIB es una biblioteca de código abierto de matemáticas y ciencia de datos que proporciona una amplia variedad de funciones para tareas como optimización, análisis estadístico, procesamiento de señales y más. Un ejemplo de implementación podría ser: *Buscador de vecino más cerca de un punto específico con ALGLIB en C#.*
  Se define una matriz de datos de dos dimensiones, luego se construye el K-D Tree con los datos mediante la función *kdtreebuild*, se busca el vecino más cercano a un punto específico utilizando la función *kdtreequeryknn*, y finalmente se imprime el resultado.

```
using System;
using Alglib;
class Program {
    static void Main() {
        // Define the data set
        double[,] data = {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
        // Create the K-D Tree
        kdtree tree;
        alglib.kdtreebuild(data, 2, out tree);
        // Search for the nearest neighbor to a point
        double[] query = {2, 3};
        int[] idx;
        double[] dist;
        alglib.kdtreequeryknn(tree, query, 1, true, out idx, out dist);
        // Print the result
        Console.WriteLine("Nearest neighbor is point " + idx[0] + " at distance " +
dist[0]);
    }
}
```

- SciPy biblioteca de Python para computación científica, contiene implementaciones de algoritmos de búsqueda de vecinos más cercanos basados en árboles k -d. Ejemplo: *Mismo que ALGLIB pero aplicado a Python.*

  Se define un arreglo numpy que contiene los datos y se construye el árbol K-D con los datos mediante la clase KDTree de scipy. Luego, se busca el vecino más cercano a un punto específico utilizando el método query de la clase KDTree y se imprime el resultado en pantalla. La clase KDTree de Scipy es una implementación simple pero eficiente, y permite diversas aplicaciones como la búsqueda de vecinos más cercanos, región de búsqueda, entre otras.

```python
from scipy.spatial import KDTree
import numpy as np

# Define the data set
data = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])

# Create the K-D Tree
tree = KDTree(data)

# Search for the nearest neighbor to a point
query = [2, 3]
dist, ind = tree.query(query, k=1)

# Print the result
print("Nearest neighbor is point ", ind[0], " at distance ", dist[0])
```

- scikit-learn una biblioteca de Python para el aprendizaje automático, contiene implementaciones de árboles k -d para respaldar las búsquedas de vecinos de radio y vecinos más cercanos. Ejemplo: *Seguimos el ejemplo aplicado para ALGLIB*

```python
from sklearn.neighbors import KDTree
import numpy as np

# Define the data set
data = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])

# Create the K-D Tree
tree = KDTree(data)

# Search for the nearest neighbor to a point
query = [2, 3]
dist, ind = tree.query(query, k=1)

# Print the result
print("Nearest neighbor is point ", ind[0], " at distance ", dist[0])
```

Bibliografia.

Construcción de kd-trees para segmentos (1D) (1library.co)

k -d árbol DescripciónyOperaciones en árboles k -d (hmong.es)

Bibliography - OneDrive (sharepoint.com)

K-d tree explained

k-d tree - Wikipedia

Árbol Dimensional K / (Árbol K D) (opengenus.org)

SAH KD-tree construction on GPU | Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics

Experiencias con streaming Construcción de SAH KD-Trees | Publicación de la Conferencia IEEE | IEEE Xplore

Kd-Trees Assignment (princeton.edu)

COS 226 K-d Tree Assignment - Documentos de Google

k-d Tree - CSE 373 (washington.edu)

Multidimensional Data, Video 7 K d Tree Nearest Finding - YouTube