VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS - CO2003

ASSIGNMENT 2

IMPROVING THE kNN ALGORITHM USING THE kD-TREE DATA STRUCTURE

Ho Chi Minh City, 04/2024

ASSIGNMENT'S SPECIFICATION

Version 1.1

1 Assignment's outcome

After completing this assignment, students review and make good use of:

- Object-oriented programming.
- Tree data structures.
- Sorting algorithms.

2 Introduction

In assignment 1, students were tasked with implementing a simple k-nearest neighbors (kNN) classification algorithm using the Brute-force algorithm. However, the time complexity of this algorithm is O(k * n * d) because we need d loops to compute dimensions, n loops to compute distances between points in the dataset, and k loops to determine the k nearest points.

In this assignment, students are required to implement a tree data structure called kD-Tree to reduce the time complexity of the kNN algorithm. Subsequently, students must re-implement a new kNN class using the implemented kD-Tree data structure and apply it to the MNIST dataset from assignment 1.

Information regarding the MNIST dataset, the k-nearest neighbors algorithm, Euclidean distance, training, and prediction processes have been described in detail in assignment 1.

3 k-D Tree

Before introducing the k-D Tree, let's review the binary search tree (BST). A binary search tree (BST) is a hierarchical data structure consisting of nodes, where each node has at most two children, called the left child and the right child. In a BST, the left child of a node contains a value less than the node's value, and the right child contains a value greater than the node's value. This characteristic makes search, insertion, and deletion operations efficient, typically with a time complexity of O(log n) for balanced trees.

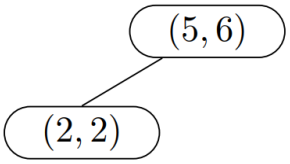
The "k" in k-D Tree represents the dimensionality of the data. While BST is a one-dimensional structure, k-D Tree is a multi-dimensional extension of BST, designed to handle data with multiple dimensions efficiently.

In a k-D tree, each node represents a k-dimensional point in space. Like BST, each node has a left and a right child, but determining whether a child node is on the left or right is based on a specific dimension at each tree level. For example, in a 2-D tree (k-D Tree with k = 2), moving from the root node to the next level, we go left or right depending on the x value (the first dimension). We go left or right for the next level based on the y value (the second dimension). Then, we switch back to x, then y, and so on. The value used to determine the current level is called the splitting axis. Subsequent sections will further introduce why it's called the splitting axis.

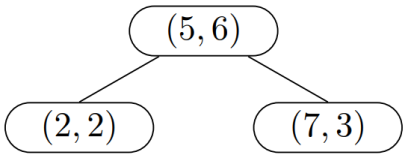
To facilitate understanding, let's refer to the following example. Let's insert all the listed points into a 2-D tree:

(5, 6), (2, 2), (7, 3), (2, 8), (8, 7), (8, 1), (9, 4), (3, 5)

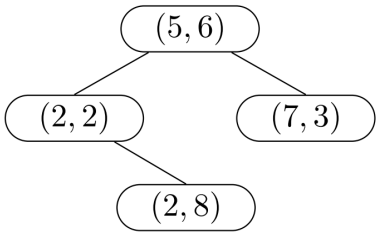
The root node is (5, 6). This root level splits based on the x-value, so when we insert (2, 2), we go to the left:



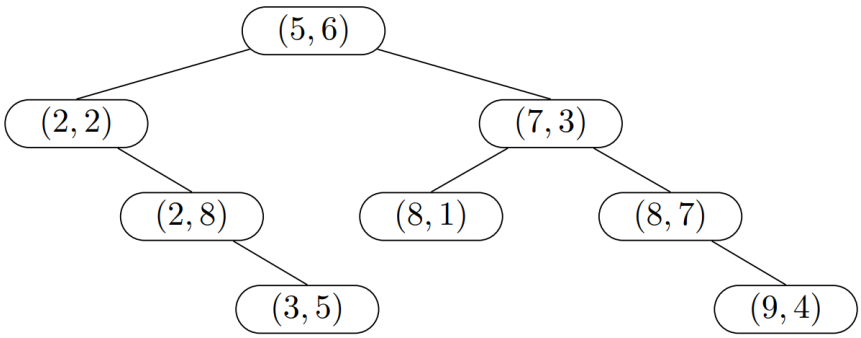
When we insert (7, 3), first we check its x-value compared to (5, 6). It's greater, so we go right and place it there:



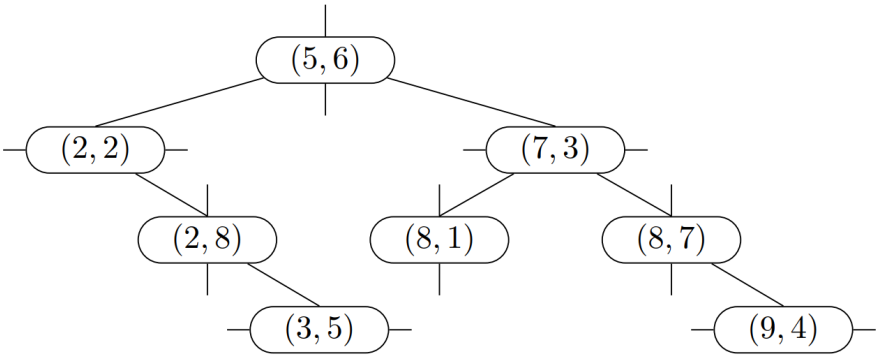
Now when we insert (2, 8), first we check its x-value compared to (5, 6). It's smaller, so we go left and reach (2, 2). Then, we check its y-value compared to (2, 2). It's greater, so we go right and place it there:



Continuing this process, we eventually get the final tree:

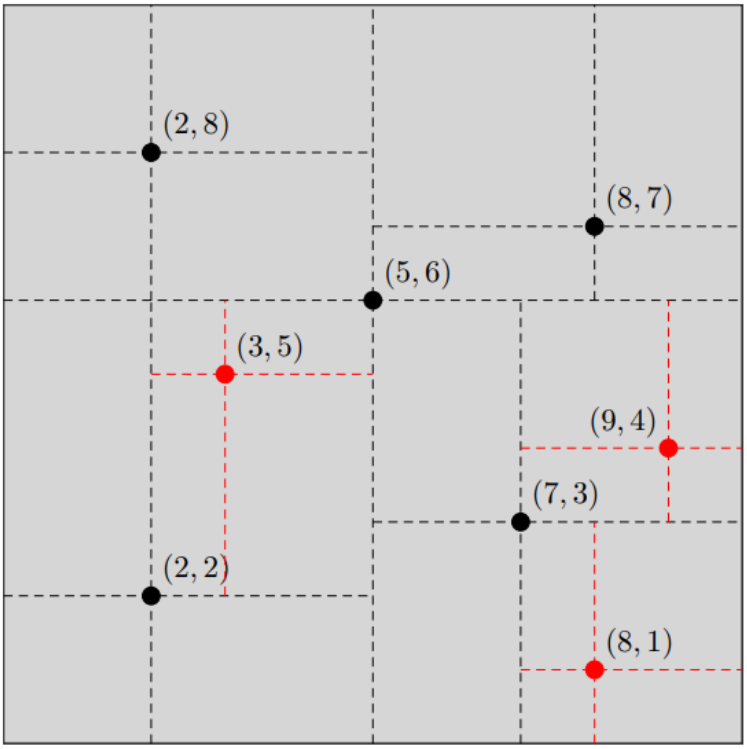


Typically, to track the splitting planes, we denote nodes as follows. We place a vertical bar to indicate splitting along the x-axis and a horizontal bar to indicate splitting along the y-axis:



Building a k-D tree involves partitioning space along different dimensions until all points are properly placed. During search, the tree is traversed based on dimensions, enabling efficient search in multiple dimensions. This makes k-D trees particularly useful in spatial data-related applications, such as nearest neighbor search, range search, and k-nearest neighbor search.

Another representation of a 2-dimensional k-D tree could be as follows:



When we set the point (5, 6) as the root node, this node splits the 2D plane into two sides, left and right, along the vertical axis (x-axis). Consequently, points with values less than 5 along the vertical axis will be on the left side of the axis split by the point (5, 6); otherwise, they will be on the right side. Similarly, considering the point (2, 2) as a child of (5, 6), we observe:

- This point divides the space into the rectangle on the left side of the point (5, 6).
- This point also divides the space by a horizontal axis (2, 2) where its child will be the axis (3, 5).

Therefore, we can refer to the points in the k-D tree as splitting planes because they partition the space of the points on a plane.

Not Just 2D: In the example above, we only illustrated with a 2D tree. k-D Trees can operate with data of multiple dimensions. For instance, when managing points in 3D space, we can continuously cycle through the splitting order of the tree by x, then y, then z, then x again, then y again, and so forth.

In this assignment, the image data is in 784 dimensions, so we will partition the k-D tree from the first dimension corresponding to column 1x1, the second dimension being column 1x2, ..., and continue in this manner until the cycle repeats.

Equal Coordinates: By convention, if we descend the tree to insert a node and are at a node where we split by axis α, and the node we want to insert has the same coordinate value, we go to the right.

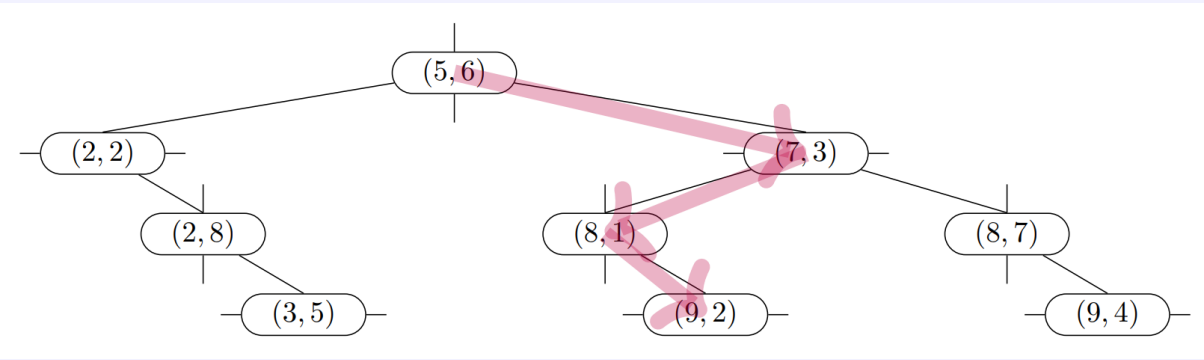
For example, if we insert (20, 30) and encounter a node (20, 40) where the split is along the x-axis, we go to the right subtree.

3.1 Insertion

Insertion in a k-D tree operates much like a binary search tree with the adjustment that we first need to determine the current splitting plane. From there, we determine whether the branch of the node to be inserted is on the left or right side of the splitting plane. Once we reach a leaf node (no further branches to descend), depending on the splitting plane of this node, we place the new node.

Example 3.1

For example, inserting (9, 2) into our k-d tree from before proceeds as follows. We start at the root node, compare along the splitting plane x, and go to the right. At (7, 3), we compare along the splitting plane y and go to the left. At (8, 1), we compare along the splitting plane x and insert to the right. Note that the inserted node will have the appropriate splitting plane for its level.



3.2 Tree Construction

The introduction demonstrated tree construction by inserting points into an empty tree in sequence from the first point to the last. However, this method may lead to an unbalanced tree.

To build a balanced k-D tree, we follow the following algorithm:

1. As we traverse down the tree, determine the current splitting plane.
2. Sort the list of points along the axis of the current splitting plane.
3. Determine the next point to be inserted as the median of the sorted list.
4. The left subtree of the newly inserted point will be the tree built from the list of points less than the median. The right subtree of the newly inserted point will be the tree built from the list of points greater than or equal to the median.

Note that various methods can be used to find the median point. This assignment requires students to find the median point using **Mergesort** to sort the list, then select the middle point as the median. If there are 2 middle points, the first point among those 2 will be chosen

Example 3.2

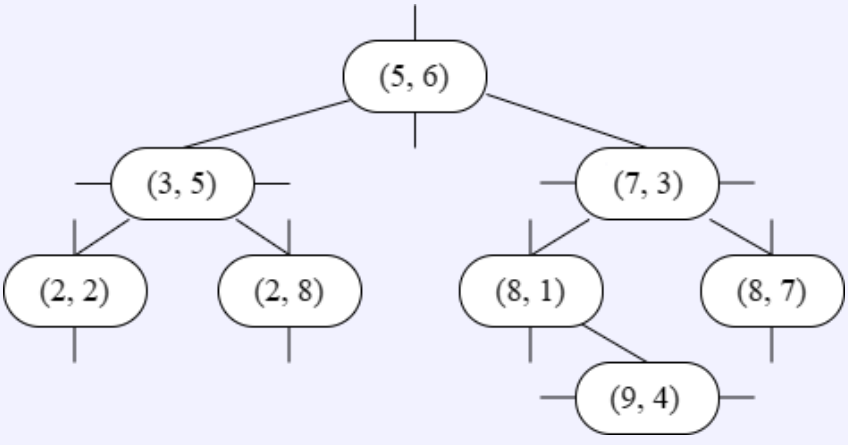
With the algorithm presented above and the list of points to be added as follows:

(5, 6), (2, 2), (7, 3), (2, 8), (8, 7), (8, 1), (9, 4), (3, 5)

The first dimension is x; we sort the points along the x-axis and obtain the list:

(2, 2), (2, 8), (3, 5), (5, 6), (7, 3), (8, 1), (8, 7), (9, 4)

The median along the x-axis will be the point (5, 6). This point will be the root node of the tree. We continue to construct the left subtree of the root node with the list (2, 2), (2, 8), (3, 5), and the right subtree of the root node will be built from the list (7,3), (8, 1), (8, 7), (9, 4). The next dimension for the two subtrees will be the y-axis.
Continuing in this manner, we will eventually obtain the tree as follows:



3.3 Deletion

3.3.1 Finding Replacement Node

When performing deletion in binary search trees, we typically search for a replacement node by finding the smallest node greater than the node to be deleted. This element is easy to find - go right (if possible) then go left as far as possible. However, this algorithm relies on the fact that we always split along a single splitting plane, and with k-d trees, this is no longer the case.

Observation shows that when searching for a replacement node, we are actually searching for the smallest node in the right subtree of the node to be deleted. So, let's imagine that we just have a k-d tree and want to find the minimum node along the α axis corresponding to the splitting plane of the node being deleted.

The algorithm for finding the replacement node is described recursively as follows:

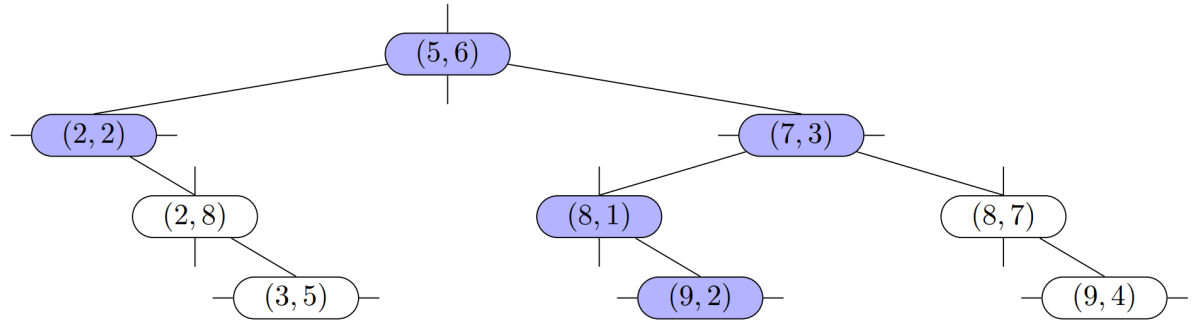
- If we are at a node split by α, the target will be in the left subtree. If the left subtree is NULL, simply return the current node. Otherwise, continue recursively to the bottom of the left subtree.
- If we are at a node split by a dimension other than α, the target could be in both subtrees or at the node itself. We recursively traverse both subtrees, take the results from both subtrees as well as from the node itself, and select a node with the smallest value along the α axis. If there are multiple, we prioritize the node itself, the left subtree, and finally the right subtree.

Example 3.3

Let's assume with the tree in the Insertion section, we want to find the node with the minimum value along the y-axis. We start from (5, 6) but since it splits along the x-coordinate, we have to check both subtrees.
We recursively go to the subtree starting from (2, 2), note that (2, 2) splits along the y-coordinate so we need to go left. However, we can't go left, so we simply return (2, 2).
We recursively go to the subtree starting from (7, 3), note that (7, 3) splits along the y-coordinate so we need to go left. We go left to (8, 1) but since it splits along the x-coordinate, we have to check both subtrees but there's only one subtree, and it returns (9, 2).
So, the subtree of (8, 1) returns the minimum value between (8, 1) and (9, 2), where minimum means the smallest y-coordinate, so it's (8, 1). The subtree of (7, 3) also returns this value. The subtree of (5, 6) returns the minimum value between (2, 2), (8, 1), and

(5, 6), so it's (8, 1).

This can be illustrated in the diagram below, where colored nodes are the nodes we actually need to analyze.



3.3.2 Deletion Algorithm

In the case of 1-D (BST), this process is simple - go right, then go left as far as possible to find the next element in order, replace the deleted node with that next element, and then either reconnect the tree (if the next element is not a leaf) or simply cut off the old next element node (if it is a leaf).

This won't work for k-D Trees because reconnecting the tree would make the lower part of the reconnected tree all jumbled up. So what can we do?

Before presenting the deletion algorithm, let's denote u_α to mean the value of dimension α of node u. For example: $(17, 42, 100)_x = 17$, $(5, 6)_y = 6$, and so on.

Our deletion process will work as follows. This process is recursive, assuming the node to be deleted is u:

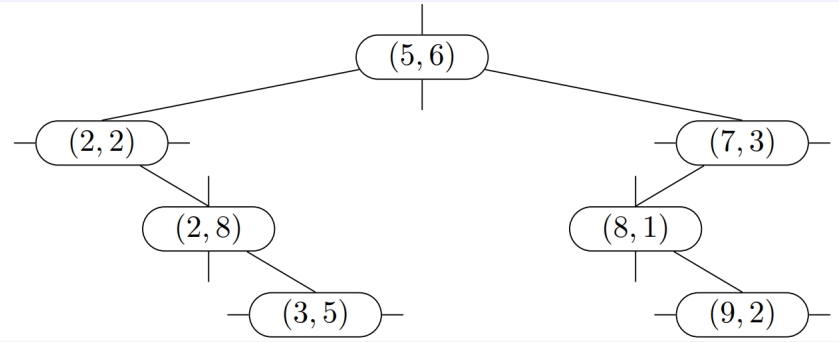
(a) If u is a leaf: Delete it and end.

(b) If u has a right subtree: Let α be the dimension u splits. Find a replacement node r in $u.right$ (the right subtree of u) such that the value along dimension α of r is the smallest. We copy the point of r onto u (overwrite u's point). Then, we recursively call deletion on the old node r.

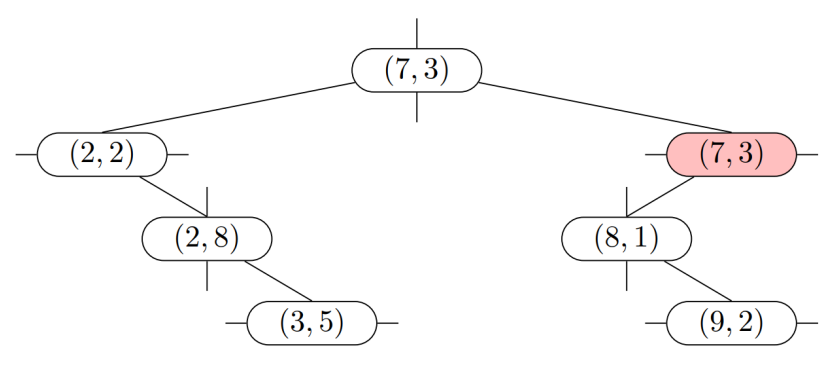
(c) If u doesn't have a right subtree, it will have a left subtree: Let α be the dimension u splits. Find a replacement node r in $u.left$ (the left subtree) such that the value along dimension α of r is the smallest. We copy the point of r onto u (overwrite u's point), and then we move the left subtree of u to become the right subtree of u. Then, we recursively call deletion on the old node r.

Example 3.4

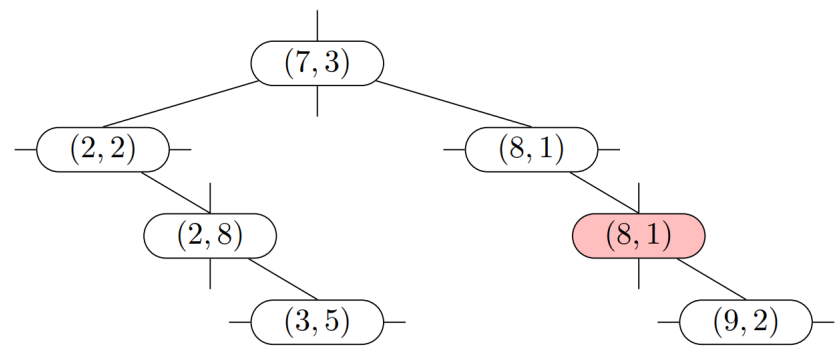
Let's delete (5, 6) from this tree:



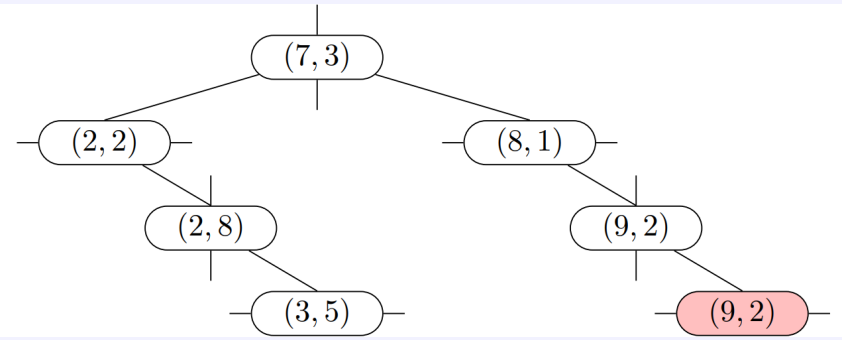
Since (5, 6) splits along the x-axis and has a right subtree, we find the node in the right subtree with the smallest x-coordinate (7, 3). We replace (5, 6) with (7, 3). There are temporarily two copies of (7, 3). The red copy is the one to be deleted.



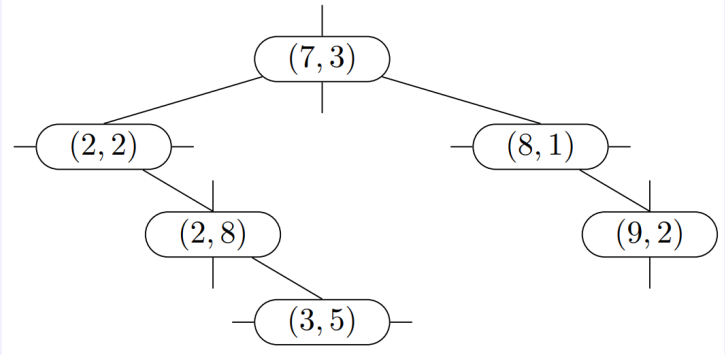
Since the red (7, 3) splits along the y-axis but doesn't have a right subtree, we choose the node in the left subtree with the smallest y-coordinate, which is (8, 1). We replace (7, 3) with (8, 1) and then move the old left subtree to the right: There are temporarily two copies of (8, 1). The red copy is the one to be deleted.



Since the red (8, 1) splits along the x-axis and has a right subtree, we choose the node in the right subtree with the smallest x-coordinate (9, 2). We replace (8, 1) with (9, 2). There are temporarily two copies of (9, 2). The red copy is the one to be deleted.



Since the red (9, 2) is a leaf node, we delete it and end:



3.4 Search

Searching in a k-d tree operates much like a binary search tree with the adjustment that we traverse branches based on the splitting plane for each node.

3.5 Nearest Neighbor Search

The nearest neighbor search aims to find the point in the tree that is closest to a specific input point (called the **query** point). This search can be efficiently performed by utilizing the properties of the tree to eliminate large portions of the search space quickly.

The process of finding the nearest neighbor point in a k-d tree proceeds as follows:

1. Start at the root node of the tree.

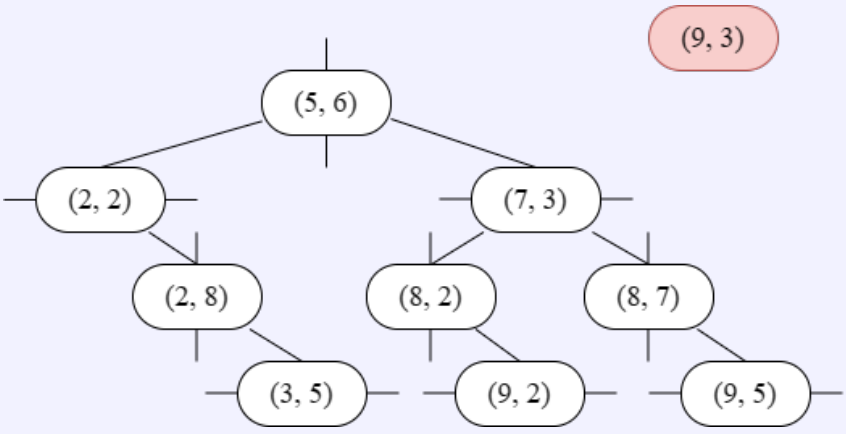
2. Traverse down the tree recursively, similar to when inserting a point, by comparing the coordinates of the current point with the splitting plane at each node.
3. When reaching a leaf node (or unable to move further), store it as a potential nearest neighbor.
4. Traverse back up the tree, checking each node:
   - If the current node is closer than the stored nearest neighbor, update the nearest neighbor.
   - Check if there could be closer points on the other side of the splitting plane:
     - Imagine a sphere around the query point with a radius equal to the distance to the current nearest neighbor. (1)
     - If this sphere intersects the splitting plane at the current node, explore the other side of the tree from that node. (2)
     - If not, continue traversing back up the tree, discarding the branch on the other side.
5. Repeat this process until reaching the root node, completing the search.

(1) For simplicity, let's call this radius R, which is the distance from the current nearest neighbor to the query point.

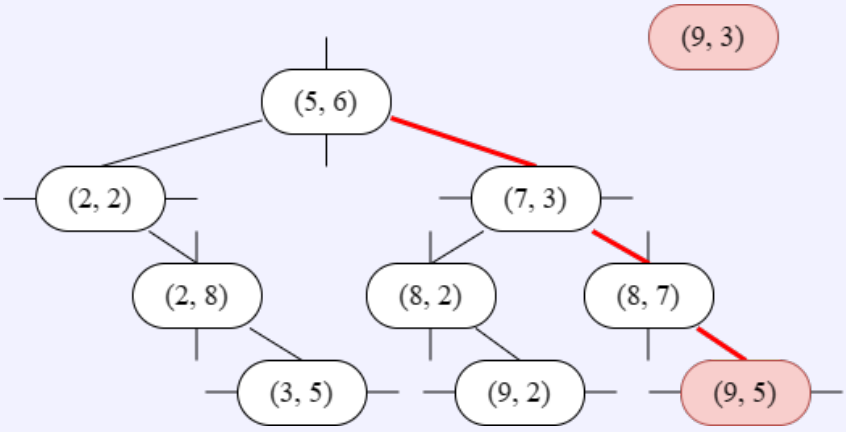
(2) For simplicity, let r be the distance along the current dimension of the current point being examined to the query point. For example, if the query point is (9, 3), the current point being examined is (8, 5), and its splitting plane is along the y-axis, then r = 2. The sphere intersects the splitting plane of the current node when R ¿= r.

Example 3.5

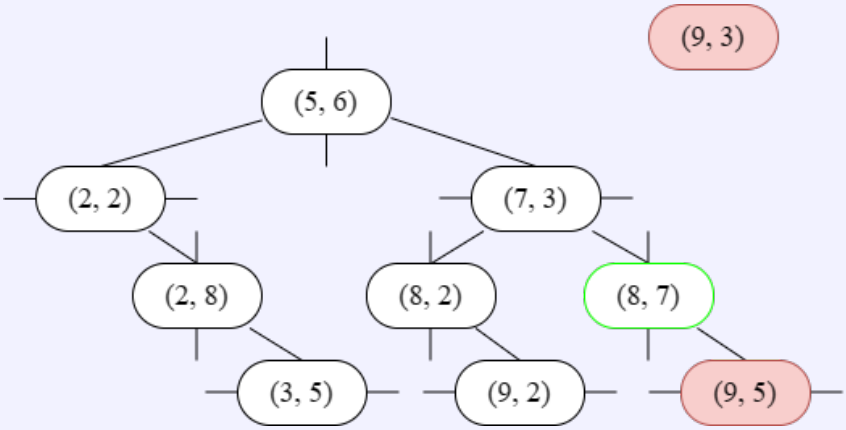
Consider an example with a 2-dimensional k-d tree as follows, and the query point is (9, 3):



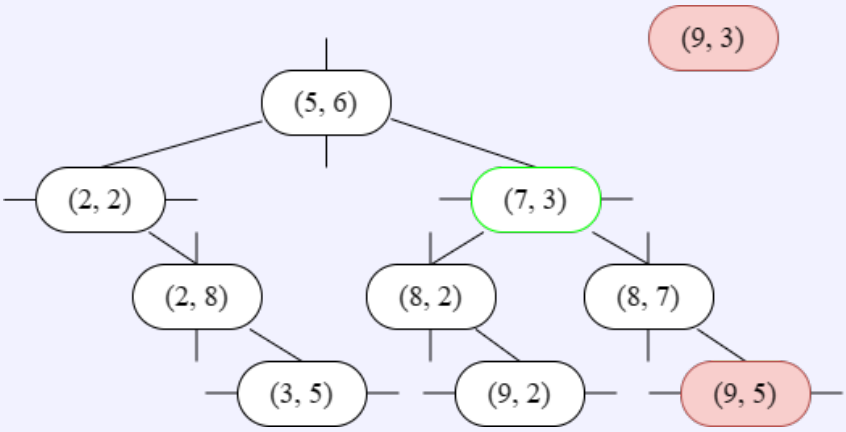
Following the algorithm to step 3, we have the nearest neighbor to the query point as (9, 5):



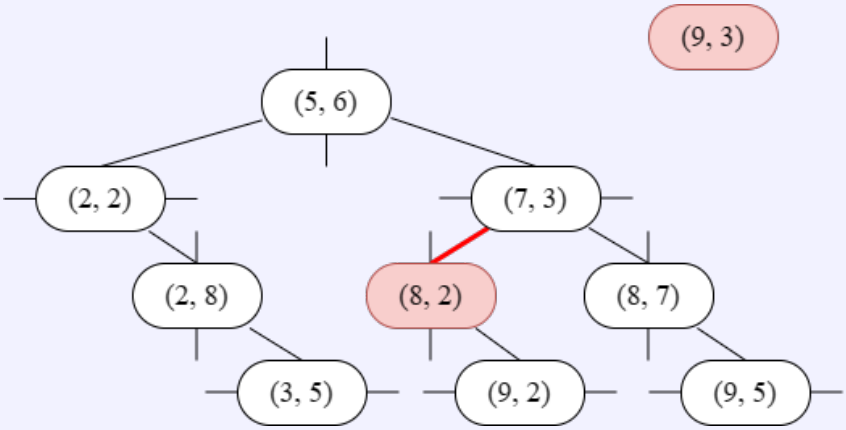
From this point, we start traversing back up to the node (8, 7), where R = 2, the distance between (8, 7) and (9, 3) is greater than R, and r = 1 < R. However, the left branch of (8, 7) is NULL, so we do not proceed further and continue up to the parent node:



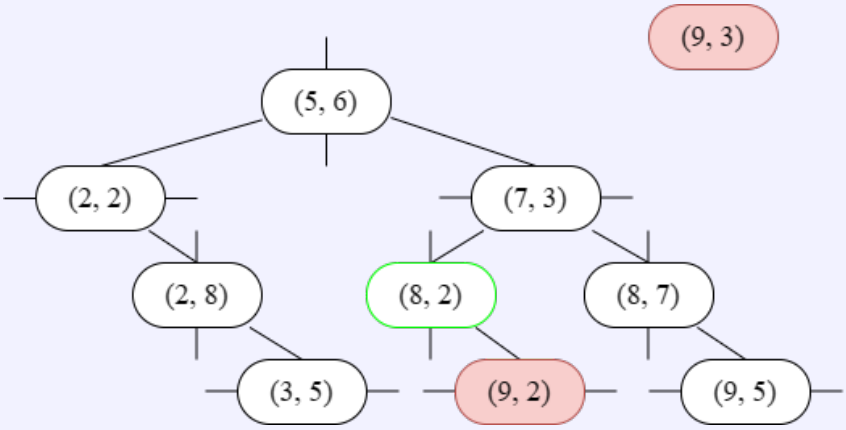
Examining the point (7, 3), R = 2, the distance between (7, 3) and (9, 3) is not less than R. r = 0 < R, so we explore the other branch of (7, 3), which is to perform the nearest neighbor search on the tree rooted at (8, 2):



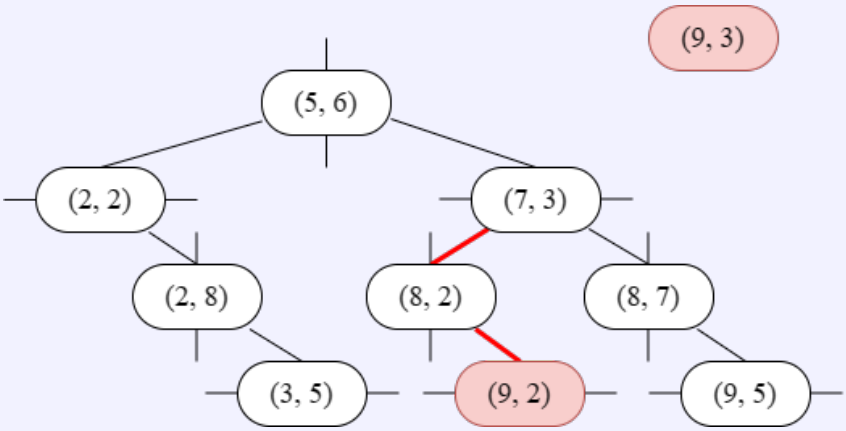
Performing the operations from steps 1 to 3 for the tree rooted at (8, 2). After reaching the point (9, 2), where the distance between (9, 2) and (9, 3) is $1 < R$, the nearest neighbor now is (9, 2):



Traversing back from node (9, 2) to node (8, 2), where $R = 1$, the distance between (8, 2) and (9, 3) is $\sqrt{2} > 1$. $r = 1 == R$, but (8, 2) does not have a left subtree, so we do not proceed further and continue up:



We skip the point (7, 3) as we have explored all branches of this point, continue traversing back up to the point (5, 6). By easy calculation, the algorithm stops at the point (5, 6):



The nearest neighbor algorithm returns the point (9, 2).

4 Requirements

In this assignment, students are required to implement the following classes to describe the data processing process and implement the k-nearest neighbor algorithm using the k-D Tree structure, including the kDTree class and the kNN class.

4.1 Class kDTree

The kDTree class stores data in the k-D tree structure and provides users with basic operations on the tree.

The description for the kDTree class is as follows:

```
struct kDTreeNode
{
    vector<int> data;
    kDTreeNode *left;
    kDTreeNode *right;
    kDTreeNode(vector<int> data, kDTreeNode *left = nullptr, kDTreeNode *
   right = nullptr)
    {
        this->data = data;
        this->left = left;
        this->right = right;
    }

    friend ostream &operator<<(ostream &os, const kDTreeNode &node)
    {
        os << "(";
        for (int i = 0; i < node.data.size(); i++)
        {
            os << node.data[i];
            if (i != node.data.size() - 1)
            {
                os << ", ";
            }
        }
        os << ")";
```

```
        return os;
    }
};

class kDTree {
private:
    int k;
    kDTreeNode* root;
public:
    kDTree(int k = 2);
    ~kDTree();

    const kDTree& operator=(const kDTree& other);
    kDTree(const kDTree& other);

    void inorderTraversal() const;
    void preorderTraversal() const;
    void postorderTraversal() const;
    int height() const;
    int nodeCount() const;
    int leafCount() const;

    void insert(const vector<int>& point);
    void remove(const vector<int>& point);
    bool search(const vector<int>& point);
    void buildTree(const vector<vector<int>>& pointList);
    void nearestNeighbour(const vector<int>& target, kDTreeNode* &best);
    void kNearestNeighbour(const vector<int>& target, int k, vector<
   kDTreeNode*> &bestList);
};
```

Below is detailed information about the methods and attributes in the `kDTree` class:

1. `int k;`
   - Attribute indicating the dimensionality of the k-D tree.
2. `kDTreeNode* root;`
   - Root node of the tree.
3. `kDTree();`
   - Constructor for the `kDTree` class.
   - `root` is assigned to NULL.

4. `~kDTree();`
   - Destructor for the `kDTree` class.
5. `kDTree& operator=(const kDTree& other);`
   - Assignment operator.
6. `kDTree(const kDTree& other);`
   - Copy constructor.
7. `void inorderTraversal() const;`
   - Traverse the tree in In-order and print each node in the format: Nodes separated by a single space, no trailing spaces at the end of a line.
   - Node format:

     $$(a_1, a_2, a_3, ..., a_n)$$

     where a_i is the value of the i^{th} dimension of the data point, and n is the number of dimensions in the data.
8. `void preorderTraversal() const;`
   - Traverse the tree in Pre-order and print each node in the same format as In-order traversal.
9. `void postorderTraversal() const;`
   - Traverse the tree in Post-order and print each node in the same format as In-order traversal.
10. `int height() const;`
    - Return the current height of the tree. If the tree has no elements, the height is 0.
11. `int nodeCount() const;`
    - Return the current number of nodes in the tree.
12. `int leafCount() const;`
    - Return the number of leaf nodes in the tree. Leaf nodes are nodes with no left or right children.
13. `void insert(const vector<int>& point);`
    - Insert a point with values contained in the point *point* into the k-D tree.
    - If the data length differs from the dimension k, the function does not add the data to the tree.

- The insertion method is described in the previous section.

14. `void remove(const vector<int>& point);`
    - Remove a data point in the tree that matches the point `point` passed. Two data points are considered equal when their lists of values are the same and have the same number of dimensions.
    - If the data point to be removed is not found, the function does nothing.
    - The deletion method is described in the previous section.
15. `bool search(const vector<int>& point);`
    - Check if the point `point` is in the tree. If found, return `true`; otherwise, return `false`.
16. `void buildTree(const vector<vector<int>>& pointList);`
    - Method used to build the tree from a given list of data points `pointList`.
    - Students can refer to the tree construction method in the previous section.
17. `void nearestNeighbour(const vector<int>& target, kDTreeNode *best);`
    - Method used to find the nearest neighbor of the point `target`. This nearest neighbor will be stored in the `best` pointer.
    - If there are two nearest neighbors with the same distance to `target`, the nearest neighbor is the one found first.
    - Students can refer to the nearest neighbor finding method in the previous section.
18. `void kNearestNeighbour(const vector<int>& target, int k, vector<kDTreeNode*> &bestList);`
    - Method used to find the k nearest neighbors of the point `target`. These k nearest neighbors will be stored in the list of pointers `bestList`. Note that this k is the number of nearest neighbors. Students should not confuse it with k in the k-D Tree, which is the dimensionality of the data.
    - If there are two nearest neighbors with the same distance to `target`, the nearer neighbor is the one found first.
    - Students can refer to the nearest neighbor finding method in the previous section.

4.2 class kNN

The class `kNN` is used to implement the functionalities of a prediction model using the kNN algorithm with a k-D Tree data structure. The design for the `kNN` class will be as follows:

```
class kNN {
private:
    int k;
public:
    kNN(int k = 5);
    void fit(const Dataset& X_train, const Dataset& y_train);
    Dataset predict(const Dataset& X_test);
    double score(const Dataset& y_test, const Dataset& y_pred);
};
```

Below is detailed information about the data members and methods in the `kNN` class:

1. `int k;`
    - The k value indicating the number of nearest neighbors. Students should avoid confusion with the k index of the k-D Tree.
2. `void fit(const Dataset& X_train, const Dataset& y_train);`
    - Loads the training data, where:
        - `X_train`: A data table containing features that are not labels. The number of rows in `X_train` is the number of images used for training, and the number of columns is the number of features.
        - `y_train`: A data table containing labels corresponding to the features. Since they are labels, the number of rows in `y_train` is the number of images, and the number of columns is only 1.
    - Hint: Students will build the k-D Tree in this method.
3. `Dataset predict(const Dataset& X_test);`
    - Executes the prediction based on the kNN algorithm.
    - `X_test` is a data table containing the features of the images to be predicted. The number of rows in `X_test` is the number of images used for prediction, and the number of columns is the number of features.
    - Note that students need to perform predictions for multiple new images, corresponding to the number of rows in the data table `X_test`.
    - The returned result is a data table with multiple rows and 1 column. Each cell in the column contains the label (class) predicted by the kNN algorithm for the respective image.
    - Hint: Students will find the k nearest neighbors in the k-D Tree built from the training set.

4. `double score(const Dataset& y_test, const Dataset& y_pred);`

    - Measures the accuracy of the prediction process.
    - `y_test` is a data table containing the true labels of the images to be predicted. The number of rows in `y_test` is the number of images used for prediction, and the number of columns is 1.
    - `y_pred` is a data table containing the predicted labels by the kNN algorithm (obtained from the `predict` function). The number of rows in `y_pred` is the number of images used for prediction, and the number of columns is 1.
    - The returned result is a real number calculated as follows:

$$\text{Accuracy} = \frac{\text{Number of correctly predicted images}}{\text{Total number of predicted images}}$$

      Where the number of correctly predicted images is the number of images whose predicted label matches the true label.

4.3 class Dataset and train_test_split Function

This assignment has **already implemented** the Dataset class and the train_test_split function. Students should read and understand these classes and functions to implement the test cases and the classes mentioned above. There are some changes in the description of the Dataset class compared to Assignment 1.

The Dataset class is used to store and manipulate tabular data, specifically in this assignment, the MNIST dataset. The description for the Dataset class is as follows:

```
class Dataset {
private:
    list<list<int>> data;
    vector<string> columnName;
public:
    Dataset();
    ~Dataset();
    bool loadFromCSV(const char* fileName);
    void printHead(int nRows = 5, int nCols = 5) const;
    void printTail(int nRows = 5, int nCols = 5) const;
    void getShape(int& nRows, int& nCols) const;
    void columns() const;
    bool drop(int axis = 0, int index = 0, std::string columns = "");
    Dataset extract(int startRow = 0, int endRow = -1, int startCol = 0, int
    endCol = -1) const;
    Dataset(const Dataset& other);
```

```
    Dataset& operator=(const Dataset& other);
    friend void train_test_split(Dataset& X, Dataset& y, double test_size,
                        Dataset& X_train, Dataset& X_test, Dataset& y_train,
    Dataset& y_test);
    friend class kNN;
};

void train_test_split(Dataset& X, Dataset& y, double test_size,
                        Dataset& X_train, Dataset& X_test, Dataset& y_train,
    Dataset& y_test);
```

Below is detailed information about the data members and methods in the Dataset class:

1. `list<list<int>> data;`
   - The main field to store the data in tabular form.
2. `vector<string> data;`
   - The field to store information about the columns of the data.
3. `Dataset();`
   - Default constructor.
4. `~Dataset();`
   - Destructor.
5. `Dataset(const Dataset& other);`
   - Copy constructor.
6. `Dataset& operator=(const Dataset& other);`
   - Assignment operator.
7. `bool loadFromCSV(const char* fileName);`
   - Method used to load data from the file `fileName`, specifically in this assignment, the file `mnist.csv`. The information in the file will be stored in the `data` variable and other proposed variables by the students.
   - The function returns true if the data loading is successful, otherwise false.
8. `void printHead(int nRows = 5, int nCols = 5) const;`
   - Prints the first `nRows` rows and only the first `nCols` columns of the data table.
   - Print format:
     - For the first row, print the names of the columns of the data table.

  - From the second row onwards, print the values of each cell in the table, with each element separated by a space, without any trailing spaces at the end of the line.
- Exception: If `nRows` is greater than the number of rows in the data table, print all rows in the data table. If `nCols` is greater than the number of columns in the data table, print all columns in the data table. If `nRows` or `nCols` is less than 0, do not print anything.

9. `void printTail(int nRows = 5, int nCols = 5) const;`
   - Prints the last `nRows` rows and only the last `nCols` columns of the data table.
   - Print format: Same as `printHead` method.
   - Exception: Same as `printHead` method.
10. `void getShape(int& nRows, int& nCols) const;`
    - Method returns the total number of rows and columns of the current object through two reference parameters `nRows` and `nCols`.
11. `void columns() const;`
    - Prints the names of all columns of the data table. Each name is separated by a space, without any trailing spaces at the end.
12. `bool Dataset::drop(int axis = 0, int index = 0, std::string columnName = "");`
    - Deletes a row or column of the data table.
    - If `axis` is not 0 or 1, the function does nothing and returns false.
    - If `axis` == 0, deletes a row at position `index`, then returns true. If index ¿= number of rows or ¡ 0, the function does nothing and returns false.
    - If `axis` == 1, deletes a column with the name matching `columnName`, then returns true. If columnName is not in the list of column names, the function does nothing and returns false.
13. `Dataset extract(int startRow = 0, int endRow = -1, int startCol = 0, int endCol = -1) const;`
    - Method used to extract a part of the data table, then returns the extracted data table.
    - Where: `startRow` is the starting row, `endRow` is the ending row, `startCol` is the starting column, `endCol` is the ending column.
    - If `endRow` == -1, all rows will be included. Similarly for `endCol` == -1.

- Test cases will ensure that `startRow`, `endRow`, `startCol`, and `endCol` are within valid ranges (from 0 to the number of rows/columns) and that `start` $\leq$ `end`.

5 Instructions

To complete this assignment, students must:

1. Read through this entire description file.
2. Download the initial.zip file and extract it. After extraction, students will receive the following files: main.cpp, main.hpp, kDTree.hpp, kDTree.cpp, Dataset.hpp, Dataset.o, and the mnist.csv file. Students must only submit the kDTree.hpp and kDTree.cpp, so they should not modify the main.hpp file when running the program.
3. Students should use the following command to compile:
   g++ -o main main.cpp kDTree.cpp Dataset.o -I . -std=c++11
   This command is used in the command prompt/terminal to compile the program. If students use an IDE to run the program, they need to note the following: add all necessary files to the IDE's project/workspace; change the IDE's compilation command accordingly. IDEs typically provide buttons for compiling (Build) and running the program (Run). When pressing Build, the IDE will execute a corresponding compilation command, usually only compiling main.cpp. Students need to configure the IDE to change the compilation command: add the kDTree.cpp and Dataset.o files, add the options -std=c++11, -I .
4. The program will be graded on the UNIX platform. The platform and compiler used by students may differ from the actual grading environment. The submission platform on BKeL is set up similarly to the actual grading environment. Students must run the program on the submission platform and fix any errors that occur on BKeL to ensure correct results during actual grading.
5. Modify the kDTree.hpp, kDTree.cpp files to complete this assignment and ensure the following requirements are met:

   - All methods described in this specification must be implemented so that compilation is successful. If students cannot implement a method, provide an empty implementation for that method. Each testcase will call some of the described methods to check the return results.
   - There are only 2 **include** directives in the file kDTree.hpp, which are **#include "main.hpp"** and **#include "Dataset.hpp"**. There is only one include directive

in the file kDTree.cpp, which is **#include "kDTree.hpp"**. Additionally, no other **#include** directive is allowed in these files.

6. The main.cpp file provides some simple testcases in functions with the format **"tc<x>()"**.
7. Students are allowed to write additional methods and attributes within the required classes. Students are also permitted to write additional classes beyond the required implementations.
8. Students are required to design and use data structures based on the types of lists they have learned.
9. Students must deallocate all dynamically allocated memory when the program ends.

6 Submission

Students must only submit 2 files: kDTree.hpp and kDTree.cpp, before the deadline specified in the "Assignment 2 - Submission" path. There are some simple testcases used to check the students' work to ensure that the results can be compiled and run. Students can submit their work as many times as they want, but only the last submission will be graded. Because the system cannot handle the load when too many students submit their work at the same time, students should submit their work as early as possible. Students will bear the risk if they submit their work close to the deadline. Once the deadline for submission has passed, the system will be closed, and students will not be able to submit anymore. Submissions through other methods will not be accepted.

7 Other regulations

- Students must complete this assignment on their own and must prevent others from stealing their results. Failure to do so will result in disciplinary action according to the university's cheating policy.
- Any decision made by the teachers in charge of this assignment is the final decision.
- Students are not provided with testcases after grading, but are only provided with information on testcase design strategies and distribution of the correct number of students according to each test case.
- Assignment contents will be harmonized with a question in exam with similar content.

8 Changelog

- Update the kDTreeNode struct: Add overloading for the ¡¡ operator.
- Modify the prototype of the nearestNeighbor method of the kDTree class.
- Example 3.2 in the Tree Construction section: Change the result (swap the positions of node (7,3) and node (9,4)).

——————————END——————————