Lab4: 2D-Tree

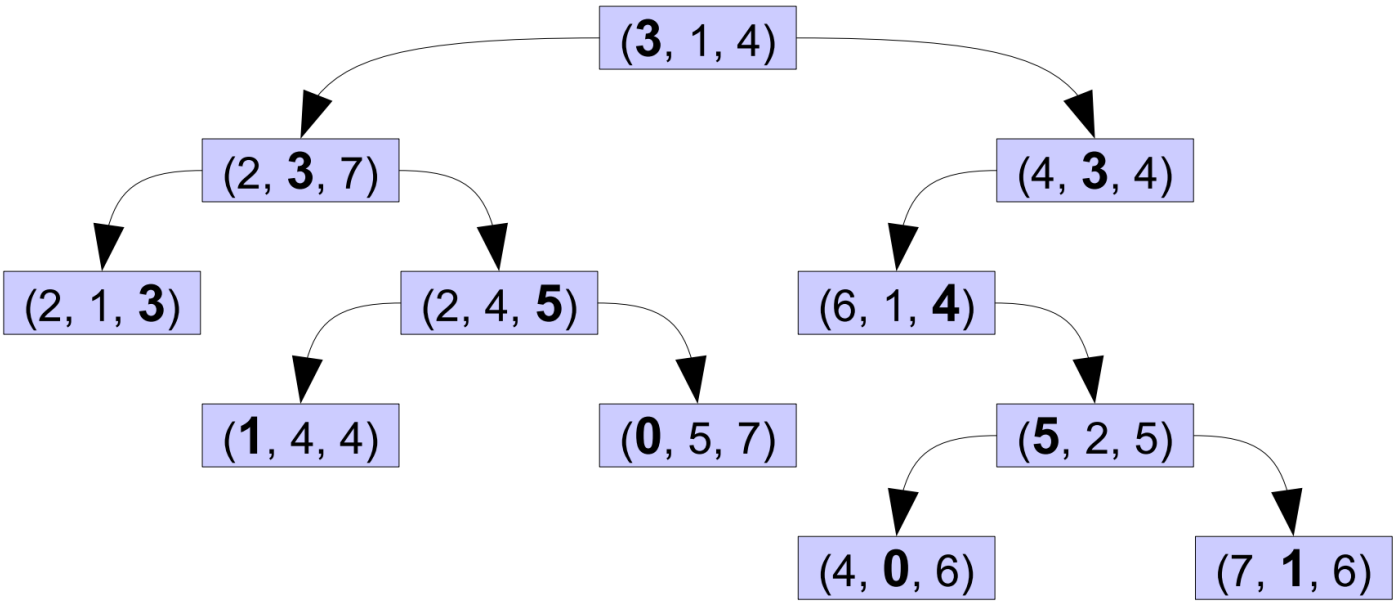
In this assignment, you will implement a special data structure called a 2D-Tree (short for “2-dimensional tree”) that efficiently supports to search “what value in the container is X closest to?”.

2D-trees can be applied to real-life problems. For example, many students love Photinia, and as the season of Photinia approaches, we may want to choose a dormitory closest to it. Given a set of dormitories represented as coordinates, the task is to find the dormitory that is closest to a given Photinia coordinate using various distance calculation methods like Manhattan, Euclidean, and even Haversine distances.

1. Story

At a high level, a kd-tree is a generalized binary search tree that stores points in k-dimensional space. This means you can use a kd-tree to store points in the Cartesian plane, in three-dimensional space, etc. However, a kd-tree cannot store other data types like strings, and all data stored in a kd-tree must have the same dimension.

It's easiest to understand how a kd-tree works by checking out an example. Here is a kd-tree that stores points in three-dimensional space (so it is a 3D-tree):



Observe that within each level of the kd-tree, a specific component of each node is highlighted in **bold**. At level `n` of the tree, the `(n % 3)` component is emphasized for every node, with both the tree level and the component index starting at 0. For instance, in the depicted example, the component 0 of the sole node `(3, 1, 4)` at level 0, which is **3**, is bolded. Each node functions similarly to a binary search tree node, but it differentiates based on the bolded component alone. For example, in the left subtree, component 0 of all

nodes is less than that of the tree's root, whereas in the right subtree, component 0 of all nodes is at least equal to that of the root node. Taking the left subtree of the kd-tree as another example, its root node `(2, 3, 7)` has component 1, **3**, in bold. All nodes in its left subtree have their component 1 values strictly less than 3, and the right subtree adheres to the same principle. This pattern is consistent throughout the tree.

Given how kd-trees store their data, we can efficiently **query whether a given point is stored in a kd-tree** as follows. Given a point P, start from the root of the tree. If the root node is P, return the root node. If component 0 of P is strictly less than the first component of the root node, then look for P in the left subtree. This time we need to compare component 1 of P. Otherwise, then component 0 of P is no less than component 0 of the root node, and we descend into the right subtree. We continue this process, cycling through which component is considered at each step, until we fall off the tree or find the node in question. Inserting into a kd-tree is similarly analogous to inserting into a regular BST, except that each level only considers one part of the point.

2. Lab Requirement

You must implement all the functions marked `DO NOT CHANGE SIGNATURE` in our handout, otherwise there might be some compiling issue. For those functions with no such mark, you can choose whether to use it as a helper or simply discard it. You can also add your own funtions. Here are some notes about our project structure.

Calculator.h

This file contains three ways for calculating distance in 2D space. We have provided the earth distance calculation method known as Haversine distance since it is too difficult. You need to implement the other two distance calculation methods. They are much easier, so don't get misled by the complicated Haversine distance calculation method.

Comparator.h

This file contains some helper functions for comparing float point numbers . If you don't like it and feel like creating your own comparing functions, go ahead and do it.

TreeNode.h

`TreeNode` is a class which is used to store two-dimensional data. In 2D-Tree, each tree node can be expressed in the form (x, y). It stores the coordinates with vector.

1. `TreeNode(initializer_list<double> coords)` initialize a new `TreeNode` object;
2. `const double &operator[](int index) const` returns the corresponding dimension. For example,

`node[0]` returns x.

3. `int dimension() const` returns how many dimensions we have. In our case it will always be 2.

Tree.h

`TwoDimenTree` is a class for the tree which supports the following functions.

1. `TwoDimenTree()` is the initializer of class `TwoDimenTree`.
2. `istream &operator>>(istream &in, TwoDimenTree &tree)` input a 2D-Tree and the rule is provided in Section 4.
3. `TreeNode *findNearestNode(const TreeNode &target)` search the 2D-Tree and find the closest point to target;
    i. The distance between two points is calculated by `double calculateDistance(const TreeNode &nodeA, const TreeNode &nodeB) const`, which is provided by `DistanceCalculator *calculator`.
    ii. Comparing floating-point numbers in C++ may cause problems. Thus **we provided some functions for comparing in Comparator.h**. You can use these functions directly or modify them for your own use.
    iii. **If there are two points that have the same distance** to the target, then you are supposed to choose the point whose x is smaller. If their x's are also the same, then choose the point whose y is smaller;
4. `~TwoDimenTree()` the deconstruct function which is used to reclaim all the data in the 2D-Tree, don't forget to implement it.

You must implement the Tree completely by yourself. We have provided the main function and the test cases. Run main.cpp and check if you can pass all the test cases.

3. Guidance

The algorithm to find the nearest node in the 2D-Tree:

Let the target point be `target(t_0,t_1)`.

Maintain a global best estimate of the nearest neighbor `guess`. Set `guess` to NULL.

Maintain a global value of the distance to that neighbor `bestDist`. Set `bestDist` to positive infinity.

Starting from the root, execute the following procedure:

```
if cur == NULL
  return
```

If the current location is closer to target than the best known location, update the best known location. In our case, you should use `double calculateDistance(const TreeNode &nodeA, const TreeNode &nodeB) const`, which is provided by `DistanceCalculator *calculator` to calculate the distance between two points.

```
if isLessThan(distance(cur, target), bestDist)
  bestDist = distance(cur, target)
  guess = cur
```

If `distance(cur, target)` and `bestDist` are equal, then choose the one between `cur` and `guess` as `guess` according the rule we provided in Section 2.

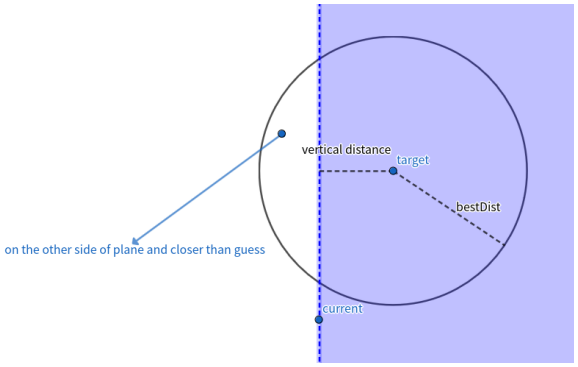
Recursively search the subtree whose range covers the target point.

In the structure of a kd-tree, a specific dimension is utilized for discrimination at each level. This dimension is determined by the calculation `i = depth % 2`, where `depth` represents the current level's depth and `i` is the index of the component used for discrimination in that level. For instance, `i = 0` at the root level, `i = 1` at level 1, `i = 0` at level 2, and so on.

To compare points based on a particular dimension, we have provided a `struct DimComparator`. For instance, if you wish to compare the first dimension of two nodes, you can do so by invoking `DimComparator(0)`, like this: `DimComparator(0)(node_a, node_b)`.

```
if isLessThan(t_i, cur_i)
  recursively search the left subtree on the next axis
else
  recursively search the right subtree on the next axis
```

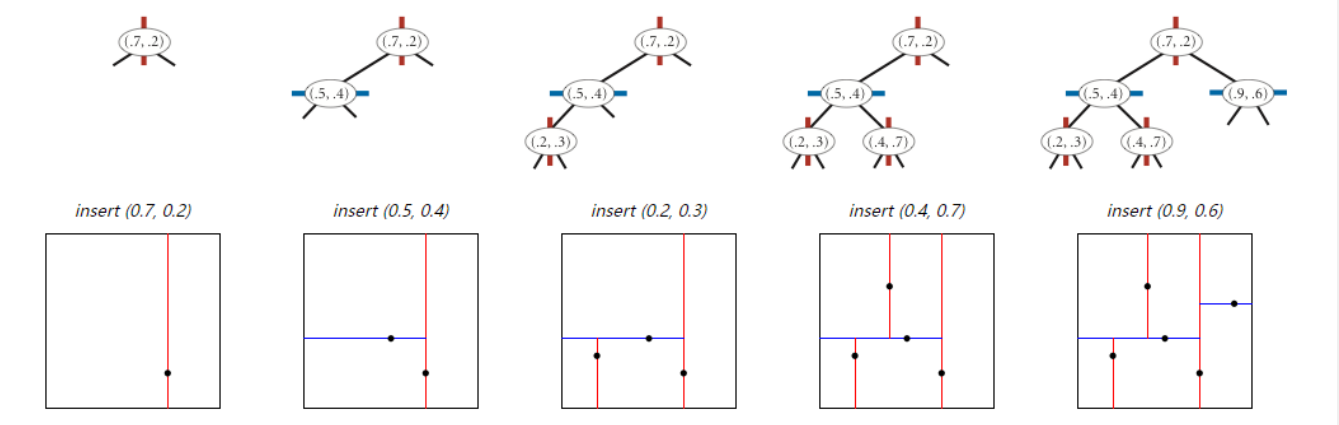
If the distance of the current point and the target point in the current dimension used for discrimination is less than `bestDist`, then there might be a point on the other side of the plane that is closer to target than `guess`. Thus we need to look on the other side of the plane by examining the other subtree.



In our case, you can use `double calculateVerticalDistance(const TreeNode &root, const TreeNode &target, int dim) const`

to calculate the difference value.

```
if isLessThan(calculateVerticalDistance(cur, target, dim), bestDist)
  recursively search the other subtree on the next axis
```

> To further understand how kd-tree helps us find the closest neighbors, we can check out the example below. A 2D-tree divides the unit square in a simple way: all the points to the left (or below) of the root go to the left subtree; all those to the right (or above) go to the right subtree; and so forth, recursively. 2D-tree divides the plane into several sectors, so that we only need to search the sectors near the target.
>
> 
>
>
> Actually, kd-tree is not suitable for calculating distance between two points on Earth, while ball tree is better at it. Check out this blog for more details. We have limited the samples so that kd-tree still works. Just treat our problem as a normal kd-tree.

4. Test and Submission

Input & Output Format

The test file consists of two parts.

- The first part is for constructing a 2D-tree.
  - The first line is the description of what kind of distance we are calculating here. Currently we have three options: Manhattan, Euclidean and Haversine.
  - The second line contains an integer `M` which represents the number of nodes in the tree.
  - Next, there are `M` lines following, each contains two integers delimited by a space. The two integers at the same line represent the values on two dimensions respectively for the corresponding node.
- The second part contains the test cases for `findNearestNode` method.
  - The first line contains an integer `N` which represents the number of test cases.
  - Next, there are `N` lines following, each contains four integers delimited by a space. In each line, the

first two integers represent the node to search and the other two integers are correct answer of the search.

The code for the second part has been already completed in main.cpp. Please refer to the rules and construct your own 2D-tree.

Coding format

Please add your own functions and variables and do not change or delete those functions with tag `DO NOT CHANGE SIGNATURE`.

Submission

Please compress your source code into a 7z file and rename it to lab4-XXX.7z, where `xxx` is your student ID. Then upload it to canvas.

The 7z file should include as below. We will remove any other file that is not in the list (eg. main.cpp).

```
lab4-XXX.7z
   |--- lab4
          |--- Tree.h
          |--- Tree.cpp
          |--- TreeNode.h
          |--- TreeNode.cpp
          |--- Calculator.h
          |--- Calculator.cpp
          |--- Comparator.h (if you need it to run your program)
```

The following structure is also acceptable. We will move every file that is not in the lab4 directory into it.

```
lab4-XXX.7z
   |--- Tree.h
   |--- Tree.cpp
   |--- TreeNode.h
   |--- TreeNode.cpp
   |--- Calculator.h
   |--- Calculator.cpp
   |--- Comparator.h (if you need it to run your program)
```

Hint:

1. 1000 <= M <= 5000, 30000 <= N <= 100000。
2. In real-world cases, search time is more important, so you are supposed to reduce the execution time of

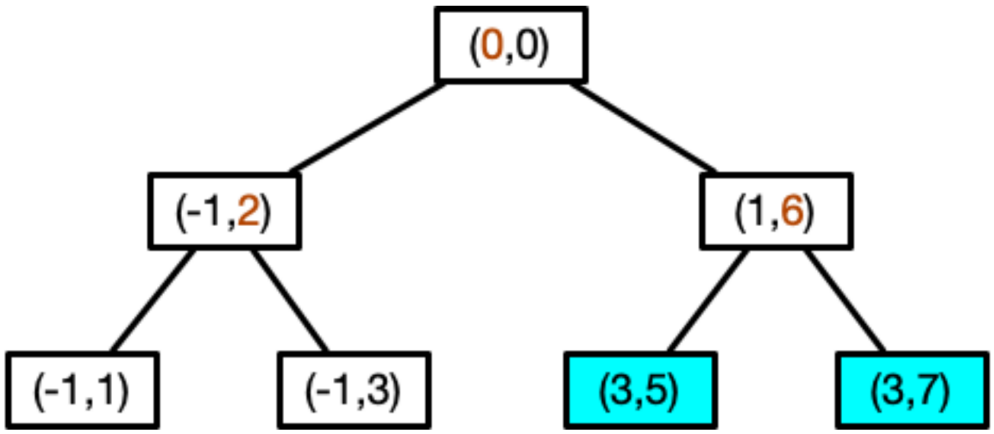
`findNearestNode`. That is to say, the 2D-Tree in your program shouldn't degenerate into a linked list when handling input like 3.txt. 30 test cases are used to test the submitted code. Every test case should finish in 1.5s, and the total runtime is limited to 45s. The program implemented by TA takes about 10s on test machine.

5. Example

Testcase 1:

```
Euclidean
7
0 0
-1 2
1 6
-1 1
-1 3
3 5
3 7
1
3 6 3 5
```

In this case, M=7 and N=1. You can construct a 2D tree with 7 nodes like this:



We have only one testcase, that is to find the point which is the closest to Point(3,6). As both Point(3,5) and Point(3,7) have the same distance to Point(3,6) and they have the same x, we choose Point(3,5) as it has smaller y.