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 most similar to?”.

2d-tree can be applied to real life problems. For example, as we all know, Photinia is widely and deeply loved by all our cute students. And here comes the season of Photinia. Assuming that we are all given another chance to decide our dormitory, of course we would choose a dormitory as close to Photinia as possible. **And here comes the problem.**

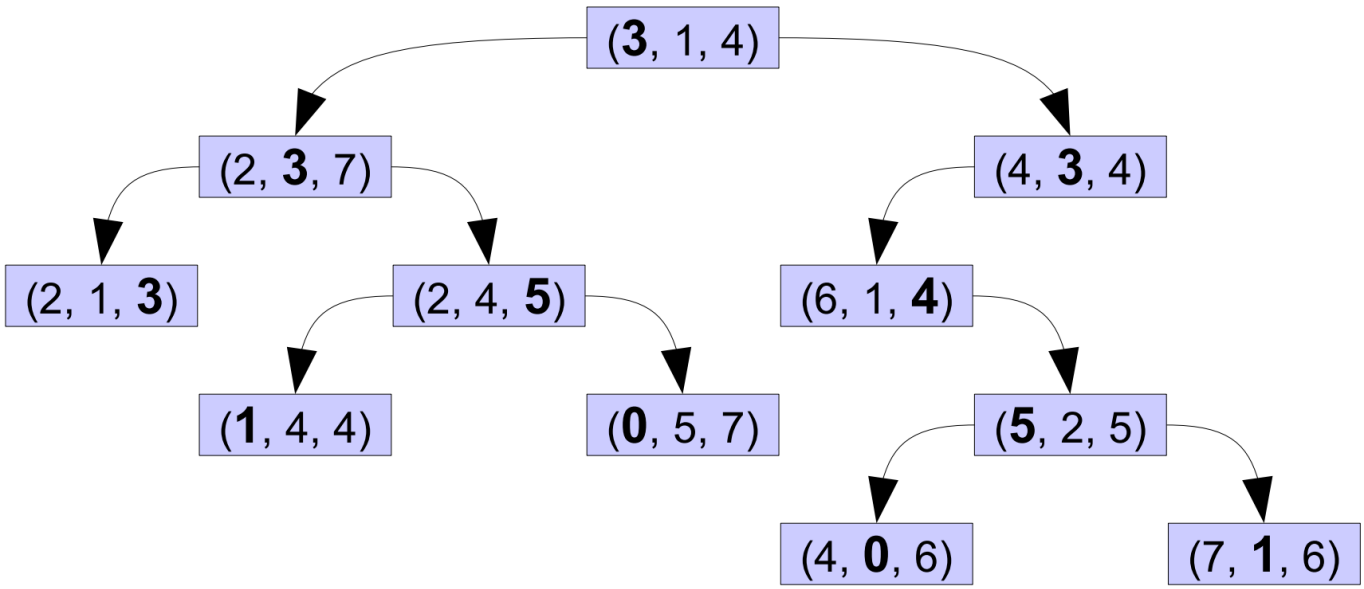
Given an array of domitory d, which contains the coordinate of domitory (x,y). That is `d = { (x1,y2), (x2,y2), ... , (xN,yN) }` . (N for the number of dormitory). And given one coordinate of Photinia (Px,Py). Now we want to pick the closest dormitory(xi,yi) to the Photinia(Px,Py).

And from below, we will learn how to solve this exciting problem!!! (by 2-d tree)

1. Story

At a high level, a kd-tree(short for “k-dimensional tree”) is a generalization of a binary search tree that stores points in k-dimensional space. That is, you could use a kd-tree to store a collection of points in the Cartesian plane, in three-dimensional space, etc. You could also use a kd-tree to store biometric data, for example, by representing the data as an ordered tuple, perhaps (height, weight, blood pressure, cholesterol). However, a kd-tree cannot be used to store collections of other data types, such as strings. Also note that while it's possible to build a kd-tree to hold data of any dimension, all of the data stored in a kd-tree must have the same dimension. That is, you can't store points in two-dimensional space in the same kd-tree as points in four-dimensional space.

It's easiest to understand how a kd-tree works by seeing an example. Below is a kd-tree that stores points in three-dimensional space:



Notice that in each level of the kd-tree, a certain component of each node has been bolded. If we zero-index the components (i.e. the first component is component zero, the second component is component one, etc.), in level n of the tree, the (n % 3)-th component of each node is shown in bold. The reason that these values are bolded is because each node acts like a binary search tree node that discriminates only along the bolded component. For example, the first component of every node in the left subtree is less than the first component of the root of the tree, while the first component of every node in the right subtree has a first component at least as a large as the root node's. Similarly, consider the kd-tree's left subtree. The root of this tree has the value (2, 3, 7), with the three in bold. If you look at all the nodes in its left subtree, you'll notice that the second component has a value strictly less than three. Similarly, in the right subtree the second component of each node is at least three. This trend continues 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 at the root of the tree. If the root node is P, return the root node. If the first component of P is strictly less than the first component of the root node, then look for P in the left subtree, this time comparing the second component of P. Otherwise, then the first component of P is at least as large as the first component of the root node, and we descend into the right subtree and next time compare the second component of P. 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 should implement the TreeNode and BinaryDimonTree(2D-Tree) classes.

TreeNode is a class which is used to store the binary dimension data. You should provide the following functions (at least):

1. **int getX():** return x(the first value in the tree node);
2. **int getY():** return y(the second value in the tree node);

BinaryDimonTree is a class for the Tree which supports the following functions (at least):

1. **BinaryDimonTree():** the construct function of class BDT;
2. **istream &operator>>(istream &in, BinaryDimonTree &tree):** input a 2D-Tree and the rule is given below;
3. **TreeNode *find_nearest_node(int x, int y):** search the 2D-Tree and find the point whose distance is smallest to the Point(x, y), and output the result to the console;

   p.s. the distance between Point(x1, y1) and Point(x2, y2) is equal to [(x1-x2)^2 + (y1-y2)^2]^(1/2);

   p.s. consider this definition of distance may be changed in the future, think about how to implement it better to reduce future work to adapt to new definition of distance;

   p.s. If there are two points that have the same distance to the Point(x, y), 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. **~BInaryDimonTree():** the deconstruct function which is used to reclaim all the data in the 2D-Tree, and you should explain your implementation to us when your lab is examined.

You must implement the Tree completely by yourself. **And the names of the classes and their functions must be same with the above.**

We have provided the main function and the test cases, you should just run the main.cpp and check if you can pass through all the test cases

3. Guidance

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

```
Let the test point be A(a0,a1)
Maintain a global best estimate of the nearest neighbor, called 'guess'
Maintain a global value of the distance to that neighbor, called 'bestDist'
Set 'guess' to NULL.
Set 'bestDist' to infinity.
Starting at the root, execute the following procedure:
    if curr == NULL
        return

/*
 * If the current location is better than the best known location,
 * update the best known location.
 */
if distance(curr, test_point) < bestDist
    bestDist = distance(curr, test_point)
    guess = curr

/*
 * Recursively search the half of the tree that contains the test point.
 * i means the dimension index which is used to discriminate in the current level,
 * for example: the i = 0 when curr is the root
 */
if ai < curri
    recursively search the left subtree on the next axis
else
    recursively search the right subtree on the next axis

/*
 * If the vertical distance between the points is smaller than the best distance,
 * look on the other side of the plane by examining the other subtree.
 */
if |curri – ai| < bestDist
    recursively search the other subtree on the next axis
```

4. Test and Submission

Input & Output Format:

The test file is divided into two parts.

- The first part is for constructing a 2D-tree.
    - The first line contains an integer M which represents the number of nodes in the tree.
    - Next, there are M lines following, each contains two integers divided 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 find_nearest_node method.
  - The first line contains an integer N which represents the number of test cases.
  - Next, there are M lines following, each contains four integers divided by spaces. 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：

1. Please add your own functions and variables in file **Tree.h** and do not change or delete those functions with tag `/* DO NOT CHANGE */`. Do not include other headers or libraries.
2. Please write your code in Tree.cpp and do not include other headers or libraries.
3. Do not modify **main.cpp**

Submission：

1. Please compress your source code into a 7z file and rename it to lab4-XXX.7z, where `XXX` is your student ID.
2. Only Tree.h and Tree.cpp are supposed to be changed.
3. Upload it to canvas

The 7z file should include as below. Only `Tree.h` and `Tree.cpp` are needed to be uploaded :)

```
lab4-XXX.7z
   |--- lab4
          |--- Tree.h
          |--- Tree.cpp
```

Hint:

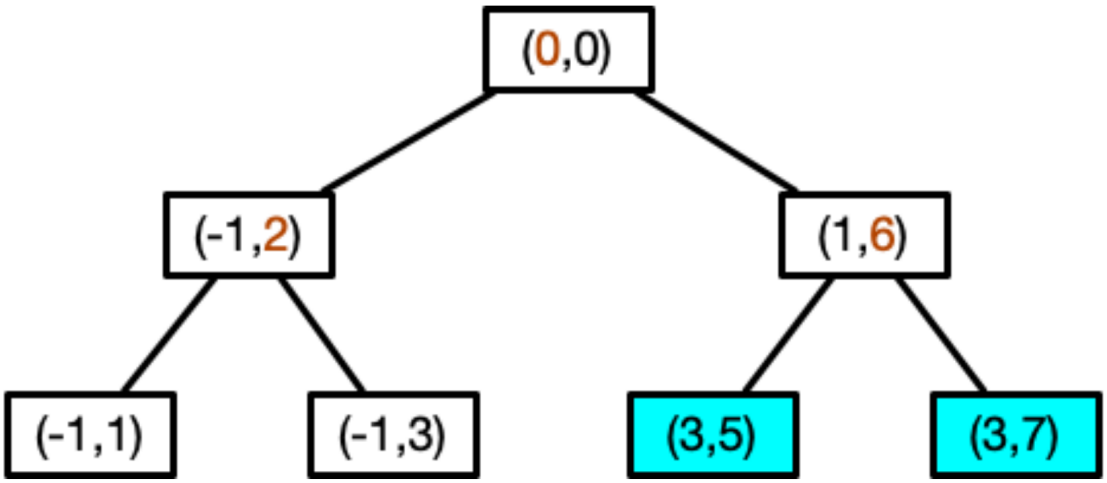
1. 1000 <= M <= 5000, 30000 <= N <= 100000。
2. In real-world cases, search time is more important, so you are supposed to reduce the execution time of find_nearest_node.

5. Example

Testcase 1:

```
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 whose distance is smallest to Point(3,6). As both Point(3,5) and Point(3,7) have the same distance to Point(3,6), we choose Point(3,5) as it has smaller y.