

Lab4: 二维树

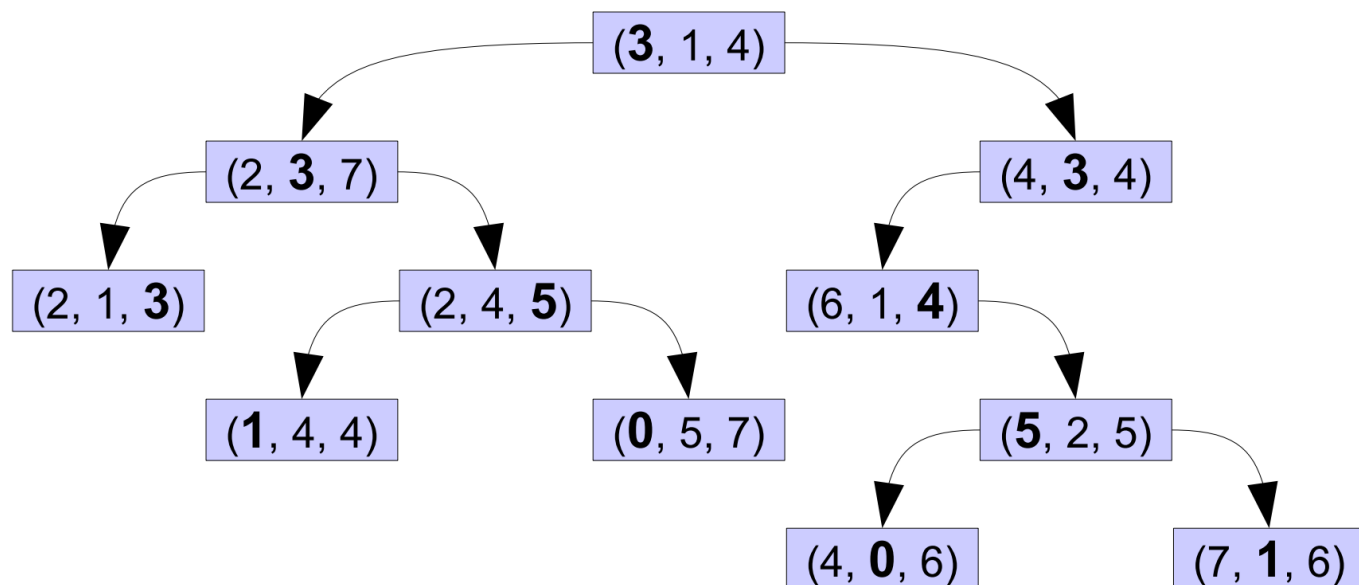
在本次实验中，你将实现一种特殊的数据结构，称为二维树（2D-Tree，意为“二维树”），它能够高效地支持搜索“容器中哪个值最接近X？”。

二维树可以应用于现实生活中的问题。例如，许多学生喜欢石楠，当石楠的季节到来时，我们可能希望选择一个离它最近的宿舍。给定一组用坐标表示的宿舍，任务是使用各种距离计算方法（如曼哈顿距离、欧几里得距离，甚至是哈弗赛因距离）找到离给定石楠坐标最近的宿舍。

1. 背景

从高层次来看，kd树是一种广义的二叉搜索树，用于存储k维空间中的点。这意味着你可以使用kd树来存储笛卡尔平面中的点、三维空间中的点等。然而，kd树不能存储其他数据类型（如字符串），并且存储在kd树中的所有数据必须具有相同的维度。

通过查看一个示例，最容易理解kd树的工作原理。以下是一个存储三维空间点的kd树（因此它是一个3D树）：



观察kd树的每一层，每个节点的特定分量被加粗。在树的第 n 层，节点的第 $(n \% 3)$ 个分量被强调，树的层级和分量索引均从0开始。例如，在所示示例中，位于第0层的唯一节点 $(3, 1, 4)$ 的第0分量3被加粗。每个节点的功能类似于二叉搜索树节点，但仅根据加粗的分量进行区分。例如，在左子树中，所有节点的第0分量小于树根节点的第0分量，而在右子树中，所有节点的第0分量至少等于树根节点的第0分量。以kd树的左子树为例，其根节点 $(2, 3, 7)$ 的第1分量3被加粗。其左子树中所有节点的第1分量值严格小于3，右子树也遵循相同的原则。这种模式在整个树中是一致的。

鉴于kd树存储数据的方式，我们可以高效地查询给定点是否存储在kd树中。给定一个点 P ，从树的根节点开始。如果根节点是 P ，则返回根节点。如果 P 的第0分量严格小于根节点的第一个分量，则在左子树中查找 P ，此时需要比较 P 的第1分量。否则， P 的第0分量不小于根节点的第0分量，则进入右子树。我们继续此过程，循环比较每一步的分量，直到超出树范围或找到目标节点。向kd树中插入节点的过程类似于向普通二叉搜索树中插入节点，只是每一层仅考虑点的一个分量。

2. 实验要求

你必须实现我们提供的标记为DO NOT CHANGE SIGNATURE的所有函数，否则可能会出现编译问题。对于未标记的函数，你可以选择将其用作辅助函数或直接忽略它。你也可以添加自己的函数。以下是项目结构的一些说明。

Calculator.h

此文件包含三种计算二维空间距离的方法。由于地球距离计算方法（即哈弗赛因距离）过于复杂，我们已提供该方法。你需要实现另外两种距离计算方法。它们更简单，因此不要被复杂的哈弗赛因距离计算方法误导。

Comparator.h

此文件包含一些用于比较浮点数的辅助函数。如果你不喜欢它并希望创建自己的比较函数，可以随意修改。

TreeNode.h

`TreeNode`是一个用于存储二维数据的类。在二维树中，每个树节点可以表示为\$(x,y)\$的形式。它使用向量存储坐标。

1. `TreeNode(initializer_list<double> coords)` 初始化一个新的`TreeNode`对象；
2. `const double &operator[](int index) const` 返回对应的维度。例如，`node[0]`返回`x`。
3. `int dimension() const` 返回维度数量。在我们的实验中，它始终为2。

Tree.h

`TwoDimenTree`是一个支持以下功能的树类。

1. `TwoDimenTree()` 是`TwoDimenTree`类的初始化函数。
2. `istream &operator>>(istream &in, TwoDimenTree &tree)` 输入一个二维树，规则见第4节。
3. `TreeNode *findNearestNode(const TreeNode &target)` 搜索二维树并找到最接近目标的点；
 1. 两点之间的距离通过`double calculateDistance(const TreeNode &nodeA, const TreeNode &nodeB) const`计算，该函数由`DistanceCalculator *calculator`提供。
 2. 在C++中比较浮点数可能会导致问题。因此，我们在`Comparator.h`中提供了一些比较函数。你可以直接使用这些函数，也可以根据需要修改它们。
 3. 如果有两个点与目标点的距离相同，则选择`x`较小的点。如果它们的`x`也相同，则选择`y`较小的点；
4. `~TwoDimenTree()` 析构函数，用于回收二维树中的所有数据，别忘了实现它。

你必须完全自行实现`Tree`类。我们已提供主函数和测试用例。运行`main.cpp`并检查是否通过所有测试用例。

3. 指导

寻找二维树中最近节点的算法：

设目标点为`target(t_0,t_1)`。

维护一个全局最佳邻居估计值`guess`，初始值为`NULL`。

维护一个全局到该邻居的距离值`bestDist`，初始值为正无穷。

从根节点开始，执行以下步骤：

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

如果当前位置比已知的最佳位置更接近目标点，则更新最佳位置。在我们的实验中，你应该使用`double calculateDistance(const TreeNode &nodeA, const TreeNode &nodeB)` `const`计算两点之间的距离，该函数由`DistanceCalculator *calculator`提供。

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

如果`distance(cur, target)`和`bestDist`相等，则根据第2节提供的规则，在`cur`和`guess`之间选择一个作为`guess`。

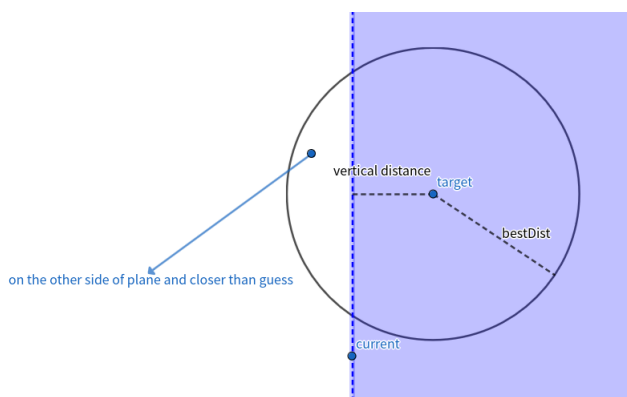
递归搜索覆盖目标点范围的子树。

在kd树的结构中，每一层使用特定维度进行区分。该维度由公式 $i = \text{depth} \% 2$ 计算，其中`depth`表示当前层的深度，`i`是该层用于区分的分量索引。例如，根节点层 $i = 0$ ，第1层 $i = 1$ ，第2层 $i = 0$ ，依此类推。

要基于特定维度比较点，我们提供了一个`struct DimComparator`。例如，如果你希望比较两个节点的第一个维度，可以通过调用`DimComparator(0)`实现，如：`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
```

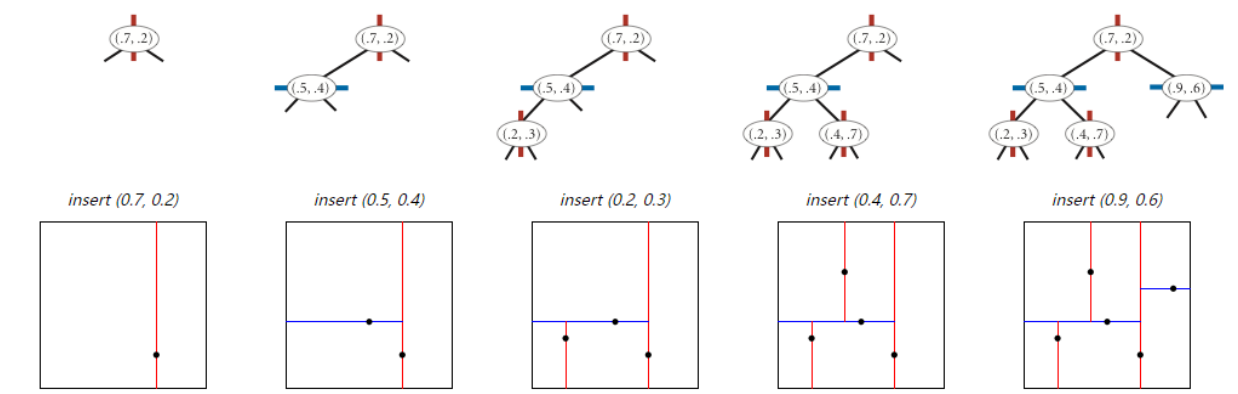
如果当前维度上目标点与当前点的距离小于`bestDist`，则可能在平面另一侧存在比`guess`更接近目标点的点。因此，我们需要通过检查另一子树来搜索平面另一侧。



在我们的实验中，你可以使用`double calculateVerticalDistance(const TreeNode &root, const TreeNode &target, int dim) const`计算差值。

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

为了进一步理解kd树如何帮助我们找到最近邻，可以查看以下示例。二维树以简单的方式划分单位正方形：根节点左侧（或下方）的所有点进入左子树；右侧（或上方）的所有点进入右子树；以此类推，递归进行。二维树将平面划分为多个扇区，因此我们只需搜索靠近目标点的扇区。



实际上，kd树并不适合计算地球上两点之间的距离，而球树更适合。有关更多详细信息，请查看这篇[博客](#)。我们已限制样本数量，使kd树仍然有效。只需将我们的问题视为普通kd树即可。

4. 测试与提交

输入与输出格式

测试文件由两部分组成。

- 第一部分用于构建二维树。
 - 第一行描述了我们正在计算的距离类型。目前有三种选项：曼哈顿距离、欧几里得距离和哈弗赛因距离。
 - 第二行包含一个整数M，表示树中的节点数量。
 - 接下来有M行，每行包含两个用空格分隔的整数。同一行的两个整数分别表示对应节点的两个维度的值。
- 第二部分包含findNearestNode方法的测试用例。
 - 第一行包含一个整数N，表示测试用例的数量。
 - 接下来有N行，每行包含四个用空格分隔的整数。每行的前两个整数表示要搜索的节点，后两个整数是搜索的正确答案。

第二部分的代码已在main.cpp中完成。请根据规则构建自己的二维树。

编码格式

请添加自己的函数和变量，不要更改或删除标记为DO NOT CHANGE SIGNATURE的函数。

提交

请将源代码压缩为7z文件，并重命名为lab4-XXX.7z，其中XXX为你的学号。然后上传到canvas。

7z文件应包含以下内容。我们将删除任何不在列表中的文件（例如main.cpp）。

```
lab4-XXX.7z
| --- lab4
|     | --- Tree.h
|     | --- Tree.cpp
|     | --- TreeNode.h
|     | --- TreeNode.cpp
|     | --- Calculator.h
|     | --- Calculator.cpp
|     | --- Comparator.h (如果需要运行程序)
```

以下结构也可以接受。我们会将lab4目录外的文件移动到其中。

```
lab4-XXX.7z
| --- Tree.h
| --- Tree.cpp
| --- TreeNode.h
| --- TreeNode.cpp
| --- Calculator.h
| --- Calculator.cpp
| --- Comparator.h (如果需要运行程序)
```

提示：

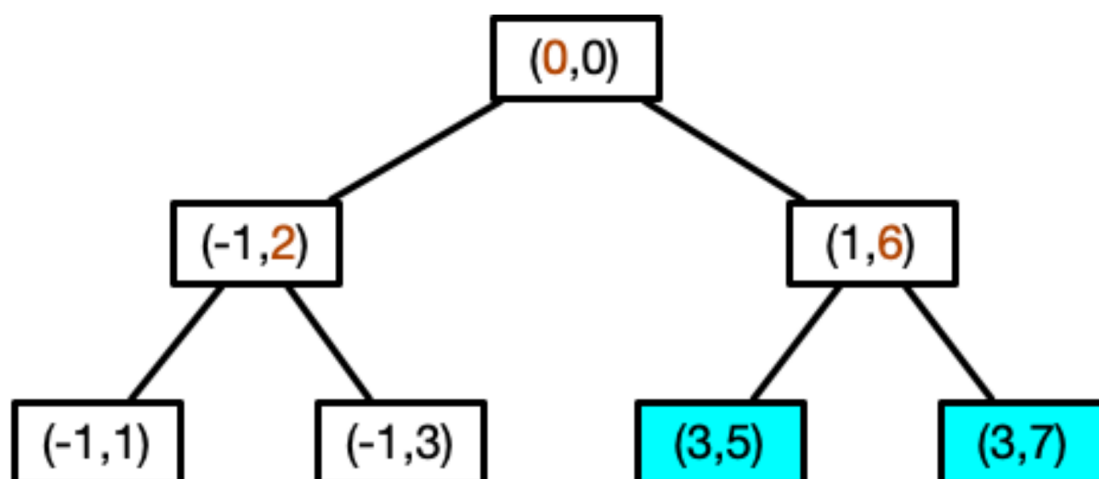
1. $1000 \leq M \leq 5000$, $30000 \leq N \leq 100000$ 。
2. 在实际案例中，搜索时间更为重要，因此你需要减少findNearestNode的执行时间。也就是说，在处理类似3.txt的输入时，你的二维树不应退化为链表。我们使用30个测试用例测试提交的代码。每个测试用例应在1.5秒内完成，总运行时间限制为45秒。TA实现的程序在测试机器上运行约10秒。

5. 示例

测试用例1：

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

在此示例中， $M=7$ ， $N=1$ 。你可以构建一个包含7个节点的二维树，如下所示：



我们只有一个测试用例，即找到最接近点 $(3,6)$ 的点。由于点 $(3,5)$ 和点 $(3,7)$ 与点 $(3,6)$ 的距离相同，并且它们的 x 相同，我们选择 y 较小的点 $(3,5)$ 。