# CSCI-2300: Data Structures and Algorithms
# Project 2 — Kd-Trees

## Due Date

The due date is **Tuesday, March 28, 2006 at 11:59:59 pm.** See the syllabus for late policies. See below for submission guidelines.

## Introduction

Binary search trees provide a data structure for 1-dimensional *range searching*, where we want to find all items with value greater than or equal to $value_{min}$ and less than or equal to $value_{max}$. In this project, you will look at implementing a data structure called the **kd-tree** and algorithms that can perform such range queries in multiple dimensions.

The focus of this programming project is *orthogonal range searching* in 2 dimensions. You will implement 2-dimensional **kd-trees**, which are an extension of binary search trees to 2 dimensions. For example, given an input set of points in the plane, if you wish to find all points whose $x$ coordinates lie in a specified range $[x_{min}, x_{max}]$ and $y$ coordinates lie in a specified range $[y_{min}, y_{max}]$, a 2-d kd-tree would be an appropriate data structure to find all points in the rectangular region $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. The use of kd-trees is not restricted to geometric problems. For example, we may wish to find all students at RPI who were born between 1982 and 1985 and whose GPA is between 3.0 and 3.5. The birthdates could be viewed as points along the $X$ axis and the GPA as points along the $Y$ axis. Extended to multiple dimensions, this method could be used to perform queries on a database of items with multiple keys with each key corresponding to a different dimension.

## Kd-Trees

**Constructing a kd-tree:** Consider a set $P$ of $n$ points in 2 dimensions. Each point $p_i \in P$ is specified by its coordinates $x_i, y_i$. The idea is to split the points based on $x$-coordinate, then on $y$-coordinate, then again on $x$-coordinate, and so on in alternating fashion. At the root, we split $P$ with a vertical line into two subsets of equal size. The subset of points to the left of or on the splitting line is stored in the left subtree. The subset of points to the right of the splitting line is stored in the right subtree. Call the two subsets $P_{left}$ and $P_{right}$ respectively. At the left child of the root, we split $P_{left}$ into two subsets with a horizontal line. Points below or on this horizontal line are stored in the left subtree of the left child, and points above this horizontal line are stored in the right subtree of the left child. The set $P_{right}$ is split into two subsets with a (different) horizontal line and the subsets are stored in the left and right subtrees of the right child. In general, we split

with a vertical line at nodes whose depth is even, and split with a horizontal line at nodes whose depth is odd. This process of splitting the points at a node continues until the size of the subset to store in a subtree reaches a minimum value, $c$. When this happens, a leaf node is created to store all the points in this subset. Note that internal nodes do not store any points. See Figure 1 for an example when $c = 1$.

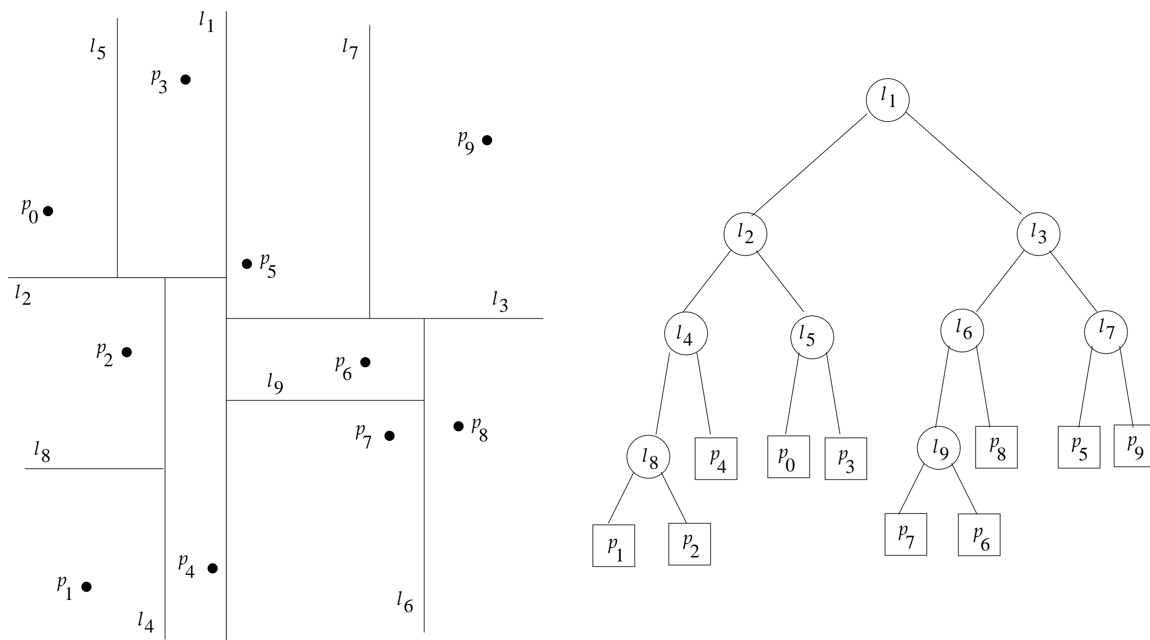


Figure 1: The input points and the subdivision of the plane with splitting lines is shown on the left, and the corresponding kd-tree is shown on the right. Interior nodes are indicated by circles, and leaf nodes, which contain individual points, are indicated by squares. Each node contains a description of the rectangular region it represents.

To ensure balance, at each internal node the splitting line passes through the median value of the points for the appropriate coordinate: the median $x$-coordinate if the depth is even or the median $y$-coordinate if the depth is odd. To make this specific, let $n$ be the number of points and suppose they are sorted by increasing $x$, so that $x_i < x_{i+1}$ for $i = 0, \ldots, n-2$. Let $m = \lfloor \frac{n-1}{2} \rfloor$. Then the vertical splitting line is at $x = (x_m + x_{m+1})/2$. In implementing this, the points should be stored in two sorted vectors, one ordered by $x$ coordinates and one ordered by $y$ coordinates, before the splitting process begins.

Each node, whether an internal node or a leaf node, corresponds to a rectangular region of the plane. Some of these rectangular regions will be unbounded. For example, the root corresponds to the entire plane, and the points in the left subtree of the root correspond to the left halfplane, and points in the right subtree correspond to the right halfplane. To make query processing simpler, each node of the kd-tree will store the bounding box of its rectangular region. You may assume unbounded means "bounded" by large negative or positive numbers. You should use $\pm 1.0E6$.

Here is pseudocode for the kd-tree construction algorithm.

```
Build_KdTree (P, depth, Bbox, c) {
  if P contains c or fewer points
    then return a leaf node v storing these points and the node rectangle Bbox
    else if depth is even
           then Split P into two subsets with a vertical line l through
                the median x-coordinate of the points in P.
                Let P_1 be the set of points to the left of or on l,
                and let P_2 be the set of points to the right of l.
                Let Bbox_1 be the part of Bbox to the left of l,
                and let Bbox_2 be the part of Bbox to the right of l.
           else Split P into two subsets with a horizontal line l through
                the median y-coordinate of the points in P.
                Let P_1 be the set of points below or on l,
                and let P_2 be the set of points above l.
                Let Bbox_1 be the part of Bbox below l,
                and let Bbox_2 be the part of Bbox above l.
         v_left <- Build_KdTree (P_1, depth + 1, Bbox_1, c)
         v_right <- Build_KdTree (P_2, depth + 1, Bbox_2, c)
         Create a node v storing l and the node rectangle Bbox,
         make v_left the left child of v, and
         make v_right the right child of v.
         return v
}
```

Remember that the depth at the root is zero and that the node rectangle at the root is infinite in all directions. For this project, infinity is represented by $1.0E6$, that is, $10^6$.

## Querying the kd-tree

Having constructed a kd-tree, we can use it to perform a variety of queries:

**Orthogonal range searching queries:** Report all points stored in the kd-tree that fall into a rectangular region $R$ specified by minimum and maximum values of $x$ and $y$. Output these in **increasing order of** $x$. Here is the pseudocode for the query algorithm. It reports all points at leaves below node $v$ that lie in the rectangular region $R$. The procedure Report_Subtree($v$) prints all points in the leaf nodes of $v$.

```
Query_KdTree (v, R) {
   if v is a leaf
      then report the points stored at v that lie in R
      else if node rectangle(v_left) is fully contained in R
               then Report_Subtree(v_left)
               else if node rectangle(v_left) intersects R
                        then Query_KdTree (v_left, R)
            if node rectangle(v_right) is fully contained in R
               then Report_Subtree(v_right)
               else if node rectangle(v_right) intersects R
                        then Query_KdTree (v_right, R)
}
```

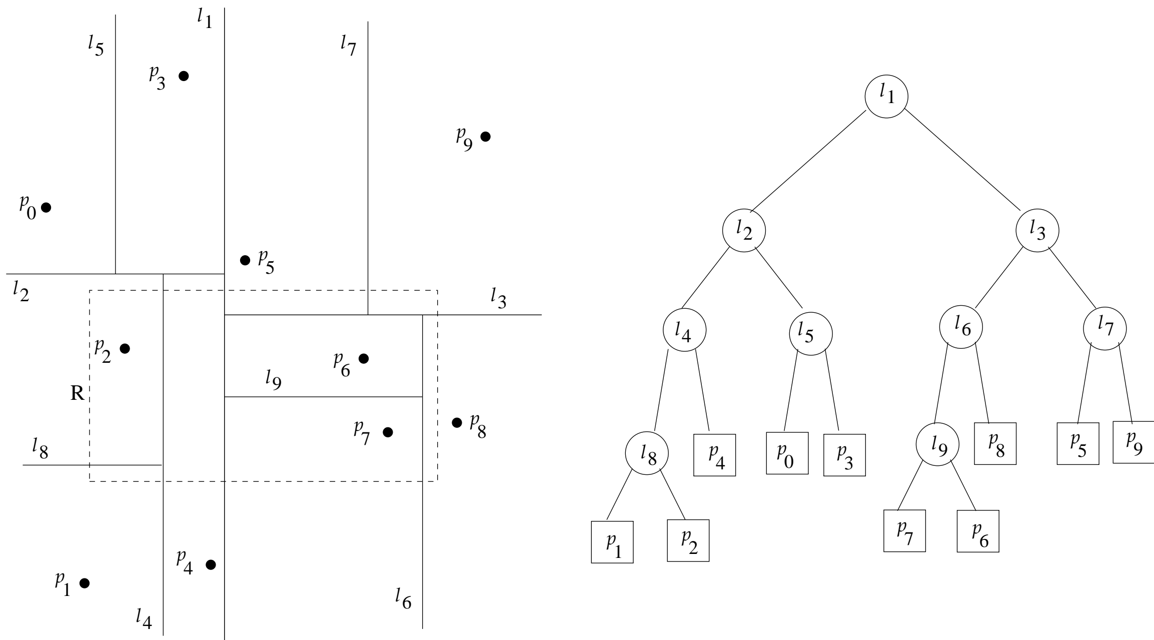


Figure 2: The points returned by the orthogonal range query, specified by the dotted rectangle $R$, are $p_2$, $p_6$, $p_7$.

**Exact match:** An *exact match query* occurs when both keys of the item match the requested query. Geometrically, this query checks whether the query point is present in the set of points $P$.

**Partial match:** This occurs when at least one key of the item matches the requested query. Geometrically, this corresponds to finding all points on a vertical line if the $x$-coordinate is specified, or all points on a horizontal line if the $y$-coordinate is specified.

**Nearest neighbor:** Given a point $x, y$, find the point in the kd-tree that is closest.

## Project tasks

Given a file containing floating point locations, build a kd-tree containing the points, output a description of the tree, and then answer queries using the tree:

**Command line:** The command line will look like:

```
kdtree points.txt c
```

where `points.txt` is the name of a file containing float $x$ and $y$ coordinates, and `c` is an integer giving the upper bound on the number of points stored at a leaf. Your program must use `argc` and `argv` to get the name of the points file and the value of c from the command line; you must not hard-code these. The queries will come from `cin` and your program output should go to `cout`.

**kd-tree class:** Create a class to store the kd-tree and implement the query responses. Clearly this is where the bulk of your work will occur. The class should not be templated and the implementation should be split into `kdtree.h` and `kdtree.cpp` files. The constructor to the class should create the tree from a single vector of points.

**outputting a description:** After calling the constructor, your program should call a function that outputs a description of the tree. This should output the leaf nodes of the tree in an in-order traversal. For each leaf node, output the bounding box of the node rectangle and output the points stored in increasing order of $x$ value.

**handling queries:** The input format for the four different types of queries are:

```
range xmin xmax ymin ymax
exact x y
partial x y
nearest x y
```

Where $x$, $y$, $xmin$, $xmax$, $ymin$ and $ymax$ are floating point numbers, with $xmin < xmax$ and $ymin < ymax$.

**Analysis:** Write the recurrence that describes the time to build a kd-tree with $n$ points. Justify the recurrence and evaluate the recurrence.

Also write the recurrence that describes the time to perform a orthogonal range query on a kd-tree with $n$ points. Justify the recurrence and evaluate the recurrence. Assume $k$ is the number of points reported by the range query.

Sample data files will be provided on the course web site. Your output format must exactly match the example output format.

## Notes

1. You may assume that all $x$ values and all $y$ values are distinct.

2. Even though we are dealing with floating point numbers that have finite precision, you may assume that two numbers $a$ and $b$ are equal if and only if $a == b$ in the code. In practice, a more sophisticated equality check would be needed.

3. Points on the boundary of the query rectangle are considered to be inside the query region.

4. You should create at least two "helper classes" to store $x, y$ points and to store the minimum and maximum $x$ and $y$ values that define a bounding rectangle. (Remember, unbounded rectangles may be represented using extreme values. Use $\pm 1.0E6$.) Create utility functions to determine if a point is in a rectangle, whether two rectangles intersect, etc. Create a rectangle that is unbounded on all sides at the root and pass modified versions of this rectangle down the tree during the construction process. The splitting lines determine the modifications of this rectangle.

5. You have not been given complete algorithms for all queries. Of special note, the algorithm for finding the nearest point is more complex than finding the leaf node whose rectangle contains the point. On the other hand, examining all leaf nodes is unacceptable.

## Grading Criteria

The total of 100 points is broken up into:

- 45 points for proper construction of the kd-tree and outputting a correct description,
- 15 points for correct range queries,
- 5 points for exact match,
- 5 points for partial match,
- 10 points for nearest neighbor search,
- 20 points for compilation, program structure, and documentation.

Within these criteria, your grade will depend on correct execution **and** query algorithm efficiency.

Your program must compile using VC++ .NET or g++ (version 2.9.5 or higher, check using `g++ --version`). The structure of your code will be judged for quality of the comments, quality of the data structure design, and especially the logic of the implementation. The comments need not be extremely long; just explain clearly the purpose of each class and each function within each class.

Please include your discussion of the running time analysis in a `readme.txt` file.

## Submission Guidelines

Your submission must include only your source code (.h and .cpp suffix) and a brief report as a `readme.txt` file. Your source code must include the `kdtree.h` and `kdtree.cpp` files, along with any additional source code files required by your kd-tree implementation. Your submission should NOT include any files with a .exe, .dsp, .dsw, .ncb, or .opt suffix. Every file should have your name in a comment line at the top. Your `readme.txt` file should have a brief description of your program design, the breakdown of the files, which compiler you used (VC++ .NET will be taken as the default), a summary of what you think works and fails in your program, **and** a short description of each of your query algorithms.

Exact details of the web-based submission procedure you must follow will be posted on the project web page. Briefly, you should submit a single zip, tar, or gzip file containing your source code files and `readme.txt` file. You can submit your project multiple times; only the most recent project submission will be graded. The most recent project submission will also be used to compute late days, if any.

## Final Warning

**A project that does not follow the submission guidelines will receive a 10 point deduction.** Proper submission is entirely **your responsibility**. Contact your TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files and **do not change them after submitting**. Don't even copy them. It is a good idea to keep your submission files in a private (protected) directory on RCS to have a valid timestamp. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.