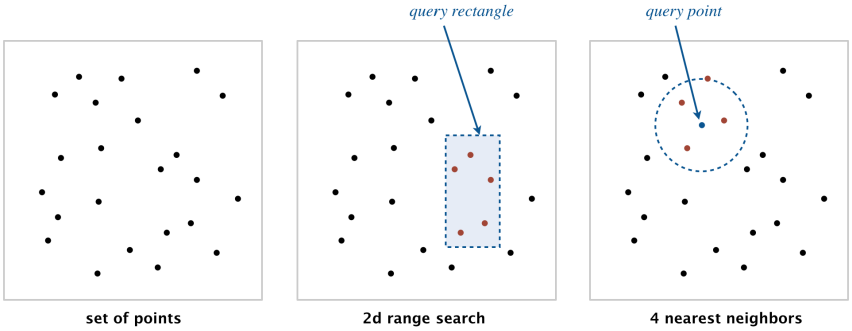
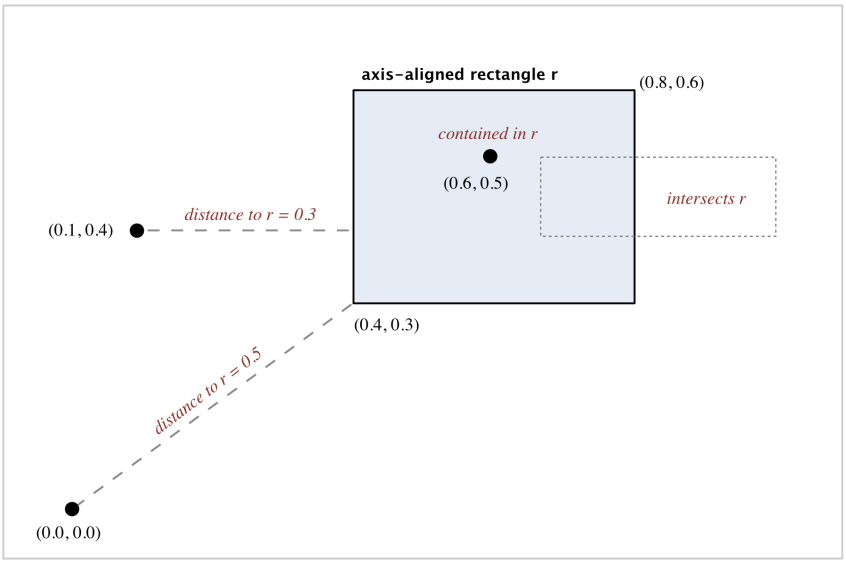
In this assignment[1], we will create a symbol-table data type whose keys are two-dimensional points. We will use a *2d-tree* to support efficient *range search* (find all of the points contained in a query rectangle) and *nearest-neighbor search* (find a closest point to a query point). 2d-trees have numerous applications, ranging from classifying astronomical objects and computer animation to speeding up neural networks and data mining.



Geometric primitives

To get started, use the following geometric primitives for points and axis-aligned rectangles in the plane.



The immutable data type `Point2D` represents points in the plane. An implementation is provided. Here is its API that you may use:

class Point2D	
`Point2D(double x, double y)`	construct the point (x, y)
`double x()const`	x-coordinate
`double y()const`	y-coordinate
`double distanceSquaredTo(const Point2D& that)const`	square of Euclidean distance between two points
`bool operator<(const Point2D& that)const`	for use in an ordered symbol table
`bool operator==(const Point2D& that)const`	does this point equal that object?
`friend std::ostream& operator<< (std::ostream& os, const Point2D& point)`	string representation with format `(x, y)`

[1] *Adapted from* https://www.cs.princeton.edu/courses/archive/fall25/cos226/assignments/kdtree/specification.php

The immutable data type `RectHV` represents axis-aligned rectangles. An implementation is provided. Here is its API that you may use:

class RectHV	
`RectHV(double xmin, double ymin, double xmax, double ymax)`	construct the rectangle $[xmin, xmax] \times [ymin, ymax]$
`double xmin()const`	minimum x-coordinate of rectangle
`double ymin()const`	minimum y-coordinate of rectangle
`double xmax()const`	maximum x-coordinate of rectangle
`double ymax()const`	maximum y-coordinate of rectangle
`bool contains(const Point2D& p)const`	does this rectangle contain the point p (either inside or on boundary)?
`bool intersects(const RectHV& that)const`	does `this` rectangle intersect `that` rectangle (at one or more points)?
`double distanceSquaredTo(const Point2D& p)const`	square of Euclidean distance from point p to closest point in rectangle
`bool operator==(const RectHV& that)const`	does `this` rectangle equal `that` rectangle?
`friend std::ostream& operator<<` `(std::ostream& os, const RectHV& rectangle)`	string representation with format `[xmin, xmax] x [ymin, ymax]`

Do not modify these data types.

Brute-force implementation

Write a C++ class named `PointST` that utilizes a balanced binary search trees to represent a symbol table whose keys are two-dimensional points. Implement the following API:

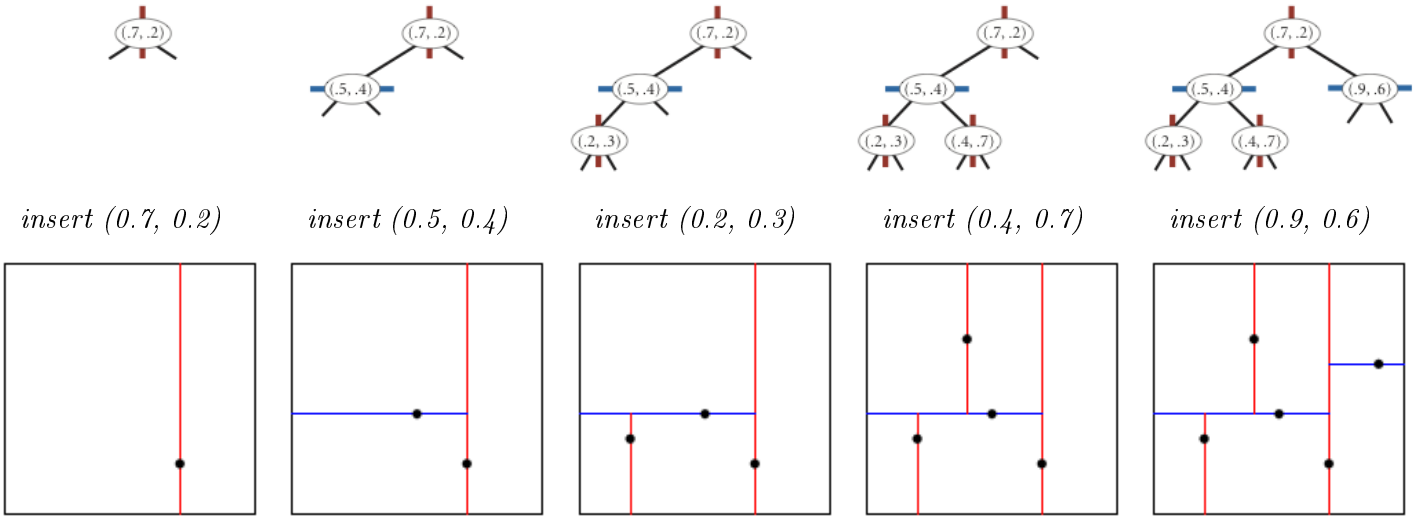
class PointST	
`PointST()`	construct an empty symbol table of points
`bool empty()const`	is the symbol table empty?
`int size()const`	number of points
`void put(const Point2D& p, const Value& val)`	associate the value `val` with point `p`
`Value& get(const Point2D& p)`	value associated with point `p`
`Value& operator[](const Point2D& p)`	value associated with point `p`
`bool contains(const Point2D& p)const`	does the symbol table contain point `p`?
`std::vector<Point2D>` `range(const RectHV& rect)const`	all points that are inside the rectangle (or on the boundary)
`Point2D nearest(const Point2D& p)const`	nearest neighbor of point `p`; throw `std::runtime_error` if the symbol table is empty
`PointST::iterator begin()const`	iterator to first `[point,value]` pair in the symbol table
`PointST::iterator end()const`	iterator to one past the last `[point,value]` pair in the symbol table

Implementation requirements. You must use `std::map`; do not implement your own red-black BST.
Unit testing. See the "FAQs - Testing" section of for more information.
Performance requirements. In the worst case, your implementation must support `size()` in constant time; `put()`, `get()` and `contains()` in $\Theta(\log n)$ time; and `nearest()`, and `range()` in $\Theta(n)$ time, where n is the number of points in the symbol table.

2d-tree implementation

Write a C++ class `KdTreeST` that uses a 2*d*-tree to implement the same API (but renaming `PointST` to `KdTreeST`). A 2*d*-tree is a generalization of a BST to two-dimensional keys. The idea is to build a BST with points in the nodes, using the x- and y-coordinates of the points as keys in strictly alternating sequence, starting with the x-coordinates.

- *Search and insert.* The algorithms for search and insert are similar to those for BSTs, but at the root we use the x-coordinate (if the point to be inserted has a smaller x-coordinate than the point at the root, go left; otherwise go right); then at the next level, use the y-coordinate (if the point to be inserted has a smaller y-coordinate than the point in the node, go left; otherwise go right); then at the next level, use the x-coordinate; and so forth.



- *Level-order traversal.* With `begin()` and `end()` methods, you must return the points in level order: first the root, then all children of the root (from left/bottom to right/top), then all grandchildren of the root (from left to right), and so forth. The level-order traversal of the above 2d-tree is $(0.7, 0.2)$, $(0.5, 0.4)$, $(0.9, 0.6)$, $(0.2, 0.3)$, $(0.4, 0.7)$. These methods are useful to assist you (when debugging) and us (when grading).

The prime advantage of a 2*d*-tree over a BST is that it supports efficient implementation of range search and nearest-neighbor search. Each node corresponds to an axis-aligned rectangle, which encloses all of the points in its subtree. The root corresponds to the entire plane $[(-\infty, -\infty), (+\infty, +\infty)]$; the left and right children of the root correspond to the two rectangles split by the x-coordinate of the point at the root; and so forth.

- *Range search.* To find all points contained in a given query rectangle, start at the root and recursively search for points in both subtrees using the following pruning rule: if the query rectangle does not intersect the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). That is, search a subtree only if it might contain a point contained in the query rectangle.

- *Nearest-neighbor search.* To find a closest point to a given query point, start at the root and recursively search in both subtrees using the following pruning rule: if the closest point discovered so far is closer than the distance between the query point and the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). That is, search a node only if it might contain a point that is closer than the best one found so far. The effectiveness of the pruning rule depends on quickly finding a nearby point. To do this, organize the recursive method so that when there are two possible subtrees to go down, you choose

first the subtree that is on the same side of the splitting line as the query point; the closest point found while exploring the first subtree may enable pruning of the second subtree.

Unit testing. See the "FAQs - Testing" section of for more information.
Performance requirements. Your implementation should follow all of the rules specified in the previous section, and it should include all the pruning rules.

Submission

Submit only `PointST.hpp` and `KdTreeST.hpp`. We will supply `Point2D.hpp` and `RectHV.hpp`. You may not call library functions except those in `std::set` and `std::map`. Finally, submit `readme.txt` and `acknowledgments.txt` files and answer the questions.

Grading rubric

file	*marks*
`PointST.hpp`	10
`KdTreeST.hpp`	24
`readme.txt`	6
	40

Frequently Asked Questions and other notes

FAQs – General

What should I do if a point is inserted twice in the data structure? The data structure represents a symbol table, so you should replace the old value with the new value.
How do I return an iterator? Define an inner iterator class within `KdTreeST` that maintains the current node being visited, along with any additional data structures needed to perform a level-order (breadth-first) traversal of the tree.
Refer to the implementation in `bst.hpp` for guidance and examples.
What should `begin()` return if there are no points in the data structure? The API says to return an iterator, so you should return the `end()` iterator.
What should `range()` return if there are no points in the range? The API says to return a `vector`, so you should return an empty vector.
What should nearest() return if there are two (or more) nearest points? The API does not specify, so you may return any nearest point (up to floating-point precision).

FAQs – `Point2D` and `RectHV`

Is a point on the boundary of a rectangle considered inside it? Do two rectangle intersect if they have just one point in common? Yes and yes.
How can I create a RectHV for the entire plane or a halfplane?
You can use the values `std::numeric_limits<double>::infinity` or its negation for one (or more) of the coordinates when you create a `RectHV`.

What does the notation [0.5, 0.75] x [0.25, 0.375] mean when specifying a rectangle? It is the Cartesian product of the x-interval $[0.5, 0.75]$ and the y-interval $[0.25, 0.375]$: the rectangle that includes all points with both $0.5 \leq x \leq 0.75$ and $0.25 \leq y \leq 0.375$. Note that the arguments to the RectHV constructor are in the order `xmin`, `ymin`, `xmax`, and `ymax` but the `ostream` operator `<<` uses the Cartesian product notation.

FAQs – `PointST`

In which order should the iterator return the points in PointST? The API does not specify the order, so any order is fine.

FAQs – `KdTreeST`

What makes KdTreeST difficult? How do I make the best use of my time? Debugging performance errors is one of the biggest challenges. It is very important that you understand and implement the key optimizations described in the assignment specification:

- *Range-search pruning.* Do not search a subtree whose corresponding rectangle does not intersect the query rectangle.
- *Nearest-neighbor pruning.* Do not search a subtree if no point (that could possibly be) in its corresponding rectangle could be closer to the query point than the best candidate point found so far. Nearest-neighbor pruning is most effective when you perform the recursive-call ordering optimization because find a good candidate point early in the serach enables you to do more pruning.
- *Nearest-neighbor recursive-call ordering.* When there are two subtrees to explore, choose first the subtree that is on the same side of the splitting line as the query point. This rule implies that if one of the two subtrees contains the query point, you will consider that subtree first.

Do not begin `range()` or `nearest()` until you understand these rules.

I'm nervous about writing recursive search tree code. How do I even start on `KdTreeST.hpp`? Use `BST.hpp` as a guide. The trickiest part is understanding how the `put()` method works. You do not need to include code that involves storing the subtree sizes (since this is used only for ordered symbol table operations).
Will I lose points for a non-recursive implementation of range search? No. While we recommend using a recursive implementation (both for elegance and as a warmup for nearest-neighbor search), you are free to implement it without using recursion.
What should I do if a point has the same x-coordinate as the point in a node when inserting or searching in a 2d-tree? Go to the right subtree as specified in the assignment under *Search and insert*.

Testing

Sample input files. The `input` folder in `kdtree.zip` contains sample input files in the specified format. It also contained visualizations for some of the small input files. The example in the assignment specification uses `input5.txt`.
Testing the bounding boxes. If you include the `RectHV` bounding boxes in the k-d tree nodes, you want to make sure that you got it right. Otherwise, the mistake might not manifest itself until either range search and/or nearest neighbor search. Here are the bounding boxes corresponding to the nodes in `input5.txt`:

$(0.7, 0.2)$: $[-\infty, \infty] \times [-\infty, \infty]$

$(0.5, 0.4)$: $[-\infty, 0.7] \times [-\infty, \infty]$

$(0.9, 0.6)$: $[0.7, \infty] \times [-\infty, \infty]$

$(0.2, 0.3)$: $[-\infty, 0.7] \times [-\infty, 0.4]$

$(0.4, 0.7)$: $[-\infty, 0.7] \times [0.4, \infty]$

Here, we are following the **`operator<<`** method format of `RectHV` which is $[xmin, xmax] \times [ymin, ymax]$ instead of $(xmin, ymin)$ to $(xmax, ymax)$.

Testing `range()` and `nearest()` in `KdTreeST`. A good way to test these methods is to perform the same sequence of operations on both the `PointST` and `KdTreeST` data types and identify any discrepancies. The key is to implement a reference solution in which you have confidence. The brute-force implementation `PointST` can serve this purpose in your testing.

FAQs – Timing

How do I measure the number of calls per second to `nearest()`? Here is one reasonable approach.

- Read the file `input1M.txt` and insert those 1 million points into the k-d tree.
- Perform m calls to `nearest()` with random points in the unit square.
- Meausre the total CPU time t in seconds for the calls to `nearest()`. You can use `Stopwatch.hpp`.
- Use m/t as an estimate of the number of calls per second.

To get a reliable estimate, choose m so that the CPU time t is neither negligible (e.g., less than 1 second) nor astronomical (e.g., more than 1 hour). When measuring the CPU time, Do not include the time to read in the 1 million points or construct the k-d tree.

How do I generate a uniformly random point in the unit square? Generate two floating-point number in range $(0.0, 1.0)$ – one for the x-coordinate and one for the y-coordinate.

Possible Progress Steps

These are purely suggestions for how you might make progress on `KdTreeST.hpp`. You do not have to follow these steps.

1. **Implement `PointST`.** This should be straightforward if you use either `std::map` and are familiar with the `Point2D` and `RectHV` APIs that you may use. After completing this step, you are only about 15% done with the assignment.
2. **Complete the k-d tree worksheet.** The file `kdtree-worksheet.pdf` contains a set of practice problems for the core k-d tree methods. See `kdtree-worksheet-sol.pdf` for solution.
3. **Node data type.** There are several reasonable ways to represent a node in a 2d-tree. One approach is to include the point, a link to the left/bottom subtree, a link to the right/top subtree, and an axis-aligned rectangle corresponding to the node.

```
template<typename Value>
struct Node {
    Point2D p;      // the point
    Value val;      // the symbol table maps the point to this value
    RectHV rect;    // the axis-aligned rectangle corresponding to this node
    Node* lb;       // the left/bottom subtree
    Node* rt;       // the right/top subtree
}
```

4. **Writing `KdTreeST`.**

   - Write `empty()` and `size()`. These should be very easy.
   - Write a simplified version of `put()` which does everything except set up the `RectHV` for each node. Write the `get()` and `contains()` method, and use these to test that `put()` was implemented properly. Note that `put()` and `get()` are best implemented by using private helper methods analogous to those found in `bst.hpp`. We recommend using the orientation (vertical or horizontal) as an argument to these helper methods.

     A common error is to not rely on your base case (or cases). For example, compare the following two `get()` methods for searching in a BST:

```
Value& get(Node* x, Key key) {
    if (x == null) throw std::runtime_error;

    if      (key < x->key) return get(x->left,  key);
    else if (x->key < key) return get(x->right, key);
    else              return x->val;
}
```

```
Value& overlyComplicatedGet(Node* x, Key key) {
    if (x == null) throw std::runtime_error;

    if (key < x->key) {
        if (x->left == null)  return null;
        else                 return overlyComplicatedGet(x->left, key);
    }
    else if (x->key < key) {
        if (x->right == null) return null;
        else                  return overlyComplicatedGet(x->right, key);
    }
    else                     return x->val;
}
```

     The second method performs extraneous null checks that would otherwise be caught by the base case. When there are multiple base cases, these extraneous checks proliferate and make your code harder to read, maintain, and debug. Trust in the base case.

- Implement the iterator and `begin()` / `end()` method. Use this to check the structure of your k-d tree.
- Add code to `put()` which sets up the `RectHV` for each Node.
- Write the `range()` method. Test your implementation as described in the testing section above.
- Write the `nearest()` method. This is the hardest method. We recommend doing it in stages.
  - First, find the nearest neighbor without pruning. That is, always explore both subtrees. Moreover, always explore the left subtree before the right subtree.
  - Next, implementing the pruning rule: if the closest point discovered so far is closer than the distance between the query point and the rectangle corresponding to a node, there is no need to explore that node (or its subtrees).
  - Finally, implement the recursive-call order optimization: when there are two subtrees to explore, always choose first the subtree that is on the same side of the splitting line as the query point.

  Test your implementation as described in the testing section above.

Optimizations

These are many ways to improve performance of your 2d-tree. Here are some ideas.

- **Squared distances.** Whenever you need to compare two Euclidean distances, it is often more efficient to compare the squares of the two distances to avoid the expensive operation of taking square roots. Everyone must implement this optimization because it is easy to do; it is likely a bottleneck; and you are permitted to call `distanceSquaredTo()` but not `distanceTo()`.
- **Range search.** Instead of checking whether the query rectangle intersects the rectangle corresponding to a node, it suffices to check only whether the query rectangle intersects the splitting line segment: if it does, then recursively search both subtrees; otherwise, recursively search the one subtree where points intersecting the query rectangle could be.
- **Save memory.** You are not required to explicitly store a RectHV in each 2d-tree node (though it is probably wise in your first version).