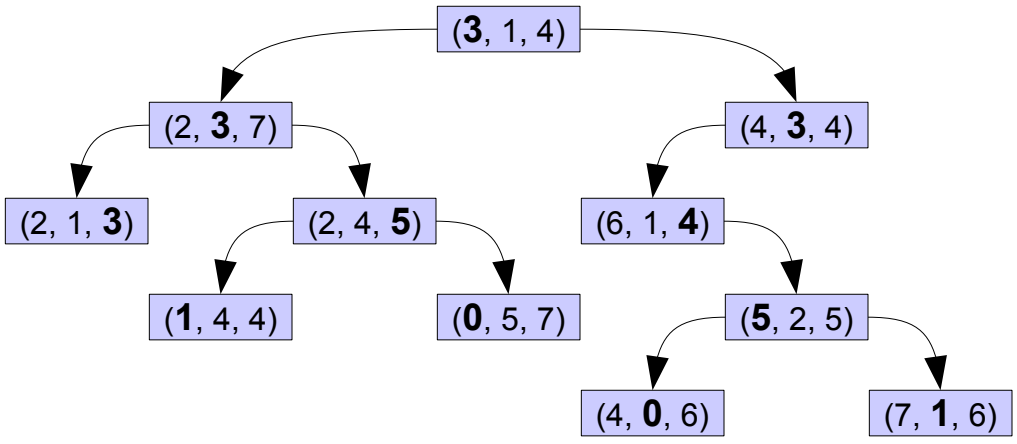
Assignment 3: `KDTree`

Due: Monday, 3rd December, 11:59pm

Over the past seven weeks, we've explored a wide array of STL container classes. You've seen the linear `vector` and `deque`, along with the associative `map` and `set`. One property common to all these containers is that they are *exact*. An element is either in a `set` or it isn't. A value either appears at a particular position in a `vector` or it does not. For most applications, this is exactly what we want. However, in some cases we may be interested not in the question "is X in this container," but rather "what value in the container is X *most similar to*?" Queries of this sort often arise in data mining, machine learning, and computational geometry. In this assignment, you will implement a special data structure called a *kd-tree* (short for "*k*-dimensional tree") that efficiently supports this operation.

At a high level, a kd-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 `string`s. 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)rd 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 com-

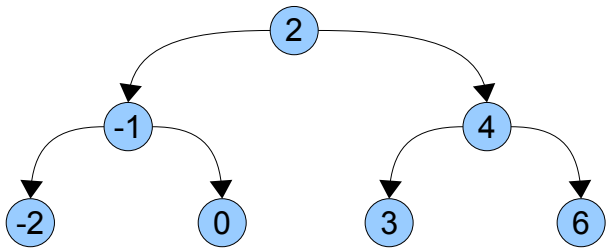
ponent of every node in the right subtree has a first component at least as a large as the root node's. Similarly, consider the kd-tree's left subtree. The root of this tree has the value (2, **3**, 7), with the three in bold. If you look at all the nodes in its left subtree, you'll notice that the second 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.

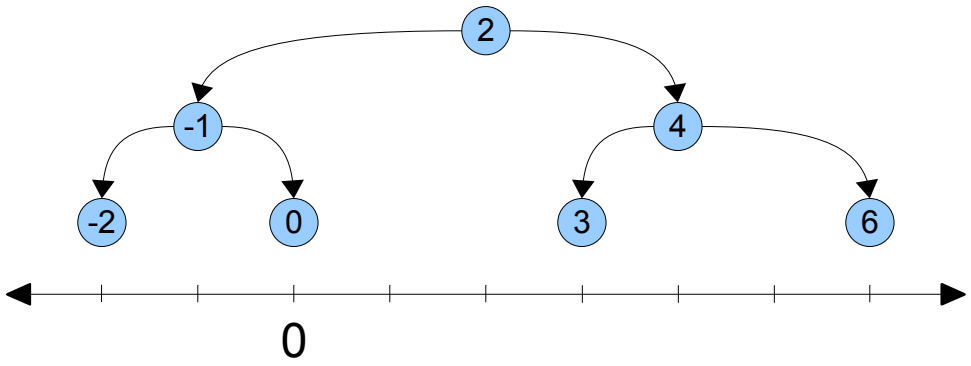
The Geometric Intuition Behind kd-Trees

You might be wondering why kd-trees store their data as they do. After all, it's not immediately obvious why you'd compare a different coordinate at each level of the tree. It turns out that there is a beautiful geometric meaning behind this setup, and by exploiting this structure it's possible to perform nearest-neighbor lookups extremely efficiently (in time better than O(n)) using a kd-tree.

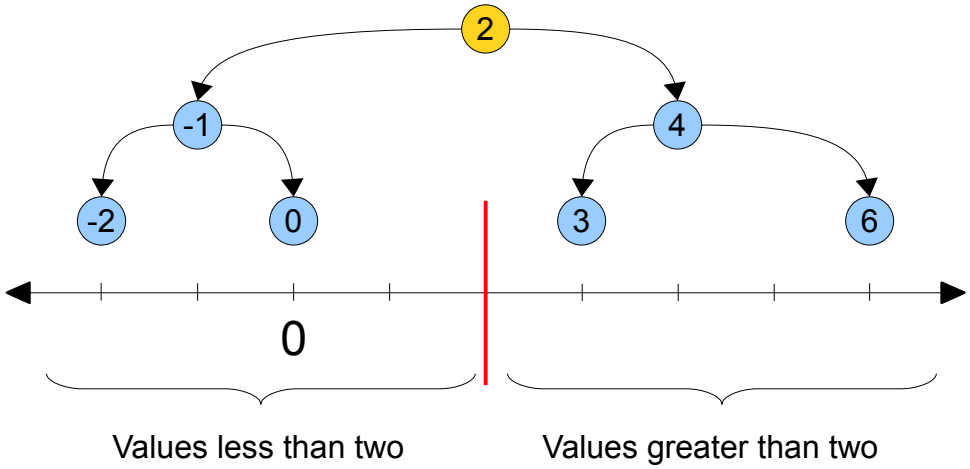
In order to make the intuition behind the coordinate-by-coordinate comparison clear, we'll quickly return to the standard binary search tree formulation you're familiar with to explore an aspect of BSTs that you may not have immediately noticed. Consider a BST where each node holds a real number. In this discussion, we'll use this tree as a reference:



Because the BST holds a collection of real numbers, we can overlay this BST with the number line. This is shown below:



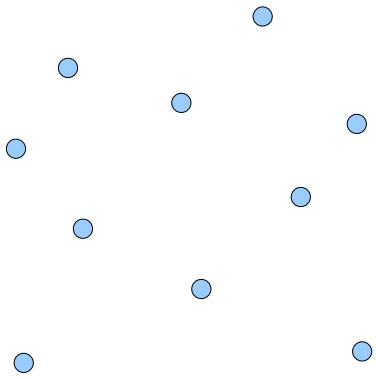
Now, suppose that we traverse the BST looking for zero. We begin at the root and check whether the root node has the value we're looking for. Since it doesn't, we determine which of the two subtrees to descend into, then recursively look in that subtree for zero. Mathematically, this is equivalent to splitting the real number line into two regions – numbers less than two and numbers greater than to two. This is shown here:



Notice that all of the nodes in the left subtree are in the left partition and all the nodes in the right subtree are in the right partition. Since 0 < 2, we know that if zero is contained in this tree at all, it must be in the left partition. This immediately rules out the possibility that zero is in the right subtree, and so we can recursively descend into the left subtree without worrying about missing the node for zero.

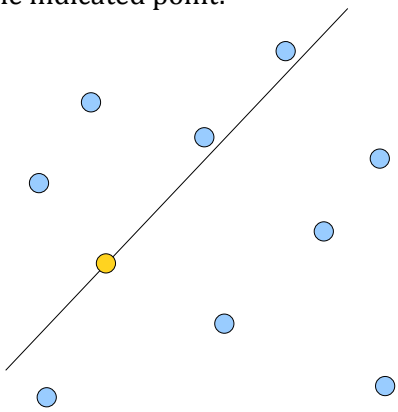
The above discussion highlights the key insight that makes binary search trees possible. Each node defines some partition of the real line into two segments, and each of the node's subtrees is fully contained within one of the segments. Searching a BST can thus be thought of as continuously splitting space in half, then continuing the search only in the half that contains the value in question.

The main reason for mentioning this line of reasoning is that it is possible to scale this up to data of higher dimensions. Suppose, for example, that we have the following collection of points in the plane:



Suppose that we want to build a binary search tree out of these points. If we use the familiar definition of a BST, we would pick some node as the root, then build a subtree out of the remaining nodes that are “less than” the root node and one subtree out of the values that are “greater than” the root node. Unfortunately, there isn't a particularly good definition of what it means for a point in space

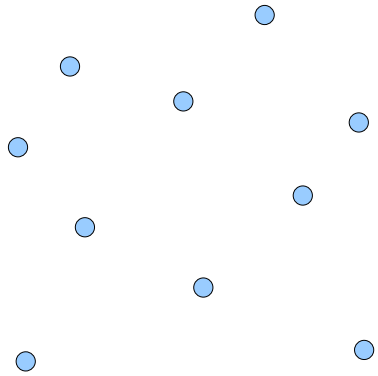
to be less than another. But let's instead consider the view of a BST we discussed above. In a BST, each point naturally split the entire real line into two regions. In two dimensions, we can split the *plane* into two regions around a point by drawing a line through that point. For example, if we draw the following line through the indicated point:



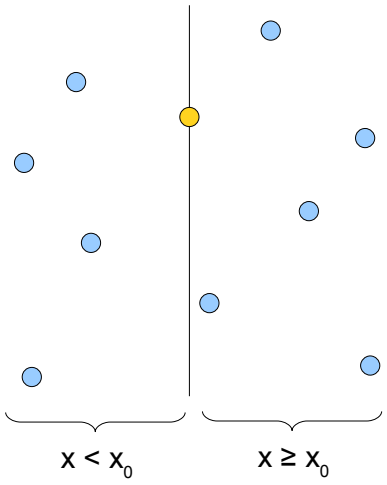
Then we've split the plane into two distinct regions, one above the line and one below this line. This observation gives us a way to build a binary search tree in multiple dimensions. First, pick an arbitrary point in space and draw a line (chosen however you'd like) through it. Next, separate the remaining points into points to one side of the line and points on the other. Finally, recursively construct binary search trees out of those points. This technique is known as *binary space partitioning* (since each step splits space into two regions), and trees generated this way are known as *binary space partitioning trees* or *BSP trees*.

But BSP trees are not restricted to just the two-dimensional plane; the same technique works in arbitrarily many dimensions. In three dimensions, we could partition *space* into two regions by drawing a *plane* through a point, then taking the regions above and below the plane as the two half-regions. When working with BSP trees, one often uses the term *splitting hyperplane* to refer to the object passing through a point that splits space in half. In two dimensions, a hyperplane is a line, while in three it's a plane. In a standard binary search tree, this "hyperplane" is just a point.

What does any of this discussion have to do with kd-trees? To answer that question, let's return to our original collection of points in two-dimensional space, as shown here:

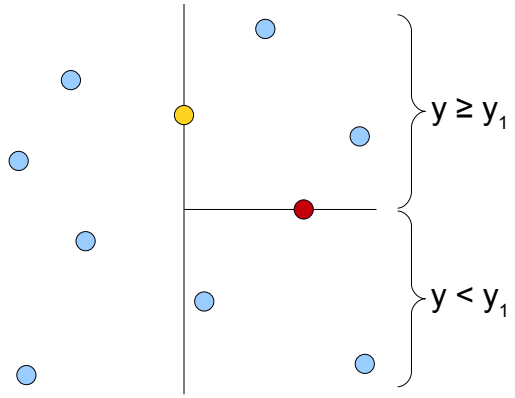


Suppose that we want to build a kd-tree out of these data points. We begin by choosing some node (which we'll say is at (x_0, y_0) for notational simplicity) and splitting the data set into two groups, one of points whose x components are less than the splitting node's, and one of points whose x components are at least as large as the splitting node's. We can visualize the split like this:

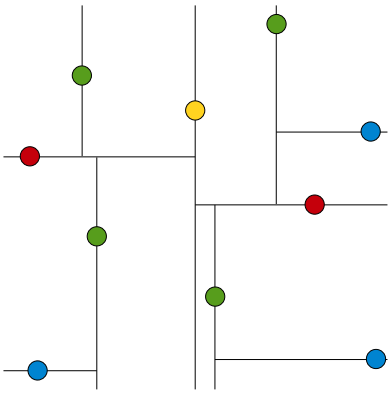


Notice that this is essentially equivalent to running a splitting hyperplane through one of the points. In that sense, a kd-tree is a special case of a BSP tree with a special rule that determines which splitting hyperplanes to use. However, we've done so without needing to write any code that manipulates hyperplanes or half-spaces. All of the complex geometry is taken care of implicitly.

Let's continue building this kd-tree. We recursively build a kd-tree in the right half-space (the points to the right of the central node) by picking the some point and splitting the data horizontally through it, as seen here:

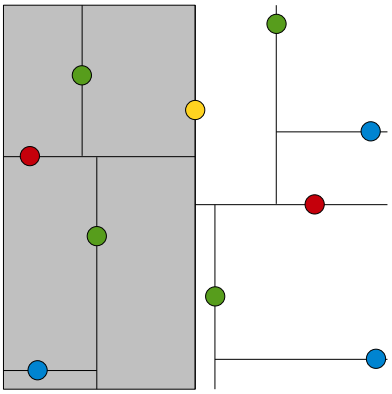


If we continue this construction to completion, our resulting kd-tree will look like this:

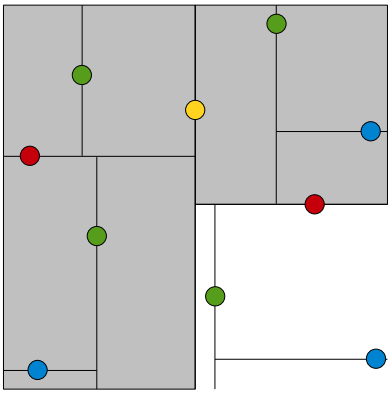


Here, the gold node is the root, nodes one level down are red, nodes two levels deep are green, and nodes three levels deep are blue.

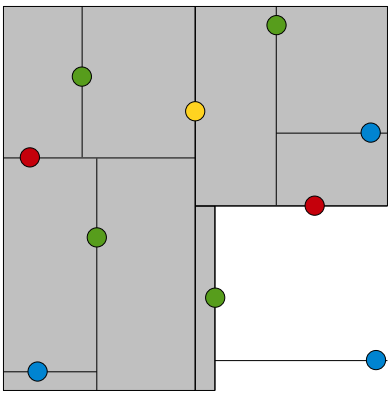
To give you a better sense for the geometric intuition behind this kd-tree, let's trace through what happens when we try looking up whether a given point is in the kd-tree. In particular, let's see what happens as we try to look up the node in the bottom-right corner of the kd-tree. We begin at the root of the kd-tree and consider whether our node's x coordinate is less than or greater than the root node's x coordinate. This is equivalent to splitting the plane vertically at the root node, then asking which half-space our node is in. Our node happens to be in the right half-space, and so we can ignore all of the nodes in the left half-space and recursively explore the right. This is shown graphically below, where the grayed-out region corresponds to parts of the plane we will never look in:



Now, we check whether our node is above or below the red node, which is the root of the tree in this half-space. Our node is below it, so we can discard the top half-space and look in the bottom. This is shown here:



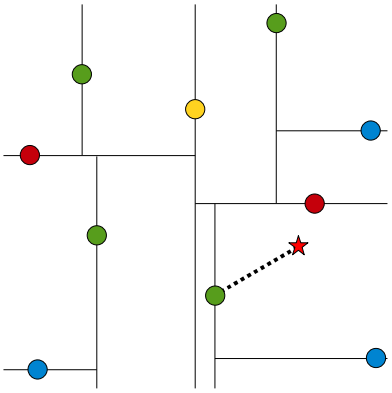
Next, we check whether we're to the left or the right of the green node that's the root of this region of space. We're to the right, so we discard the sliver of a half-space to the left of that node and continue on:



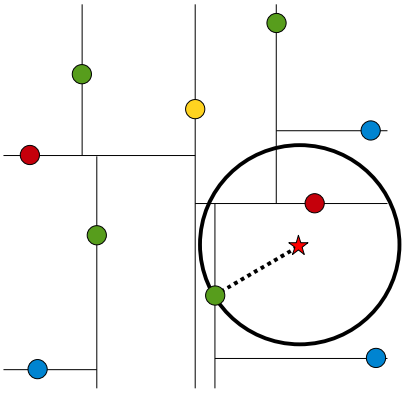
At this point, we have reached the node we're looking for, and the search algorithm terminates.

Nearest-Neighbor Lookup in kd-Trees

Now that you have a better geometric intuition for the kd-tree, we can talk about the most interesting operation on the kd-tree: nearest-neighbor lookup. This query works as follows: given a kd-tree and a point in space (called the *test point*), which point in the kd-tree is closest to the test point? (The point in the data set closest to the test point is called its *nearest neighbor*). Before we discuss the actual algorithm for doing nearest-neighbor lookup, we'll discuss the intuition behind the algorithm. Suppose that we have a guess of what we think the nearest neighbor to the test point is. For example, suppose that the test point is indicated by the star and that we think the nearest neighbor is the point connected to the star by the dashed line:



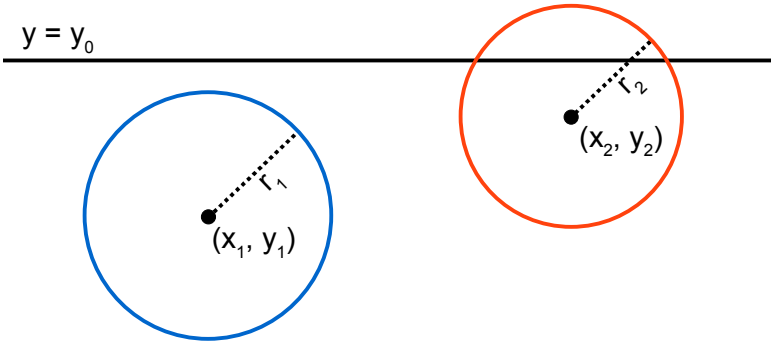
Given our guess of what the nearest neighbor is, we can make a crucial observation. If there is a point in this data set that is closer to the test point that our current guess, it must lie in the circle centered at the test point that passes through the current guess. This circle is shown here:



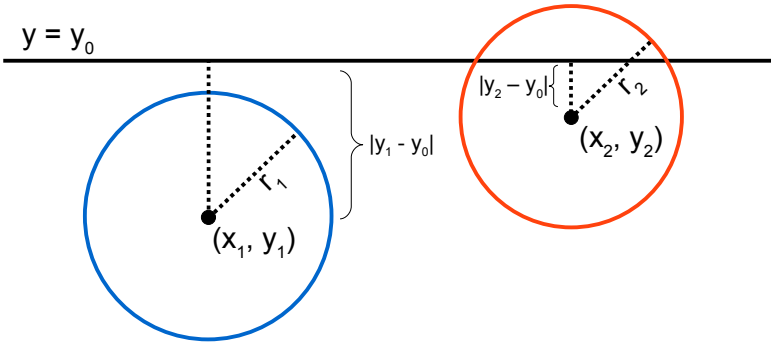
Although in this example this region is a circle, in three dimensions it would be a sphere, and in general we call it the *candidate hypersphere*.

The reason that this observation is so important is that it lets us prune which parts of the tree might hold the true nearest neighbor. In particular, notice that this circle is entirely to the right of the splitting hyperplane running vertically through the root of the tree. Consequently, any point to the left of the root of the tree cannot possibly be in the candidate hypersphere, and consequently can't be any better than our current guess. In other words, once we have a guess about where the nearest neighbor is, we can start eliminating parts of the tree where the actual answer cannot be. This general technique of searching a large space and pruning options based on partial results is called *branch-and-bound*.

From the picture it's clear that the circle of possible nearest neighbors does not cross the middle splitting hyperplane, but how can we determine this mathematically? In general, given a circle and a line (or, more generally, a hypersphere and a a hyperplane), it's a bit tricky to determine whether that circle intersects the line. Fortunately, though, the fact that we've chosen all of the splitting hyperplanes to be axis-aligned greatly simplifies this task. Below is an arbitrary line and two circles, one of which crosses the line and one of which does not:



Now, consider the distance from the centers of these circles to the line $y = y_0$. This is simply the absolute value of the difference between the circles' y coordinates and y_0, as seen here:



Notice that the distance $|y_1 - y_0|$ from the center of the blue circle to the line is greater than the radius of the circle, and so the circle does not cross the line. On the other hand, the distance from the center of the red circle to the line is less than the radius of the circle, and so some part of that circle does cross the line. This gives a general criterion for determining whether a candidate hypersphere crosses a particular splitting hyperplane. In particular, given a kd-tree node holding point $(a_0, a_1, a_2, ..., a_k)$ and hypersphere of radius r centered at $(b_0, b_1, b_2, ..., b_k)$, if the node partitions points based on their ith component, then the hypersphere crosses the node's splitting plane only if $|b_i - a_i| < r$.

To recap:

- Given a guess about which node is the nearest neighbor, we can construct a candidate hypersphere centered at the test point and running through the guess point. The nearest neighbor to the test point must lie inside this hypersphere.

- If this hypersphere is fully to one side of a splitting hyperplane, then all points on the other side of the splitting hyperplane cannot be contained in the sphere and thus cannot be the nearest neighbor.

- To determine whether the candidate hypersphere crosses a splitting hyperplane that compares coordinate *i*, we check whether $|b_i - a_i| < r$.

These observations, taken together, suggest the following algorithm for finding the nearest neighbor to a test point:

```
Let the test point be (a0, a1, ..., ak).

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, guess) < bestDist
       bestDist = distance(curr, guess)
       guess = curr

   /* Recursively search the half of the tree that contains the test point. */
   if ai < curri
       recursively search the left subtree on the next axis
   else
       recursively search the right subtree on the next axis

   /* If the candidate hypersphere crosses this splitting plane, 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
```

Intuitively, this procedure works by walking down to the leaf of the kd-tree as if we were searching the tree for the test point. As we start unwinding the recursion and walking back up the tree, we check whether each node is better than the best estimate we have so far. If so, we update our best estimate to be the current node. Finally, we check whether the candidate hypersphere based on our current guess could cross the splitting hyperplane of the current node. If it doesn't, then we can eliminate all points on the other side of the splitting hyperplane from consideration and walk back up to the next node in the tree. Otherwise, we must look in that side of the tree to see if there are any closer points.

This algorithm can be shown to run in O(log n) time on a balanced kd-tree with n data points provided that those points are randomly distributed. In the worst case, though, the entire tree might have to be searched. However, in low-dimensional spaces, such as the Cartesian plane or three-dimensional space, this is rarely the case.

k-Nearest Neighbor Searches and Bounded Priority Queues

In this discussion, we've only considered the problem of finding the *single* nearest neighbor to a test point. A more interesting question is, given a test point and some number *k*, to find the *k-nearest-neighbors* of that point. This search is often referred to as a *k-NN search*. It turns out that the previous algorithm can easily be adapted to do a *k*-NN search instead of a 1-NN search. The algorithm is almost identical, except that instead of maintaining just the best point, we maintain a list of the *k* best points we've seen so far.

Before describing the algorithm, we'll introduce a special data structure called a *bounded priority queue* (or *BPQ* for short). A bounded priority queue is similar to a regular priority queue, except that there is a fixed upper bound on the number of elements that can be stored in the BPQ. Whenever a new element is added to the queue, if the queue is at capacity, the element with the highest priority value is ejected from the queue. For example, suppose that we have a BPQ with maximum size five that holds the following elements:

Value	A	B	C	D	E
Priority	0.1	0.25	1.33	3.2	4.6

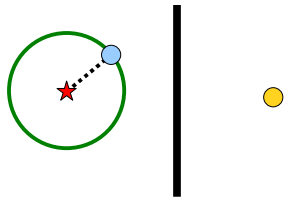
Suppose that we want to insert the element F with priority 0.4 into this bounded priority queue. Because this BPQ has maximum size five, this will insert the element F, but then evict the lowest-priority element (E), yielding the following BPQ:

Value	A	B	F	C	D
Priority	0.1	0.25	0.4	1.33	3.2

Now suppose that we wish to insert the element G with priority 4.0 into this BPQ. Because G's priority value is greater than the maximum-priority element in the BPQ, upon inserting G it will immediately be evicted. In other words, inserting an element into a BPQ with priority greater than the maximum-priority element of the BPQ has no effect. Given access to a BPQ, we can perform a *k*-NN search in a kd-tree as follows:

```
Let the test point be P = (y0, y1, ..., yk).

Maintain a BPQ of the candidate nearest neighbors, called 'bpq'
Set the maximum size of 'bpq' to k

Starting at the root, execute the following procedure:
   if curr == NULL
      return

   /* Add the current point to the BPQ.  Note that this is a no-op if the
    * point is not as good as the points we've seen so far.
    */
   enqueue curr into bpq with priority distance(curr, P)

   /* Recursively search the half of the tree that contains the test point. */
   if yi < curri
       recursively search the left subtree on the next axis
   else
       recursively search the right subtree on the next axis

   /* If the candidate hypersphere crosses this splitting plane, look on the
    * other side of the plane by examining the other subtree.
    */
   if:
      bpq isn't full
           -or-
      |curri - yi| is less than the priority of the max-priority elem of bpq
   then
      recursively search the other subtree on the next axis
```

There are two minor changes to this algorithm that differentiate it from the initial 1-NN search algorithm. First, when determining whether to look on the opposite side of the splitting plane, we use as the radius of the candidate hypersphere the distance from the test point to the maximum-priority point in the BPQ. The rationale behind this is that when finding the k nearest neighbors, our candidate hypersphere for the k nearest points needs to encompass all k of those neighbors, not just the closest. The other main change is that when we consider whether to look on the opposite side of the splitting plane, our decision takes into account whether the BPQ contains at least k points. This is extremely important! If we prune out parts of the tree before we have made at least k guesses, we might accidentally throw out one of the closest points. Consider the following setup:



Suppose that we wish to perform a 2-NN lookup for the test point indicated by the star. We recursively check the left subtree of the splitting plane, and find the point indicated in blue as a candidate nearest neighbor. Since we haven't found two nearest neighbors yet, we still need to look on the other side of the splitting plane for more neighbors, even though the candidate hypersphere does not cross the splitting hyperplane.

The Assignment

Your assignment is to implement a class representing a kd-tree, which we'll call `KDTree`, that allows clients to build kd-trees, query kd-trees for membership, and execute *k*-NN lookups on them. In the course of doing so, you'll gain experience with class implementation, `const`-correctness, templates, copy functions, operator overloading, and exception-handling. Additionally, you'll get to experience firsthand the power of *k*-NN lookups by seeing applications that build off of your `KDTree` class. The amount of code that you actually need to write is not too great – on the order of two hundred lines – though it will require you to have a solid understanding of the language features we've explored over the past weeks.

To make it easier to complete the assignment, I've broken the program down into a series of five smaller steps. I advise completing the assignment in this order, but you're free to implement `KDTree` as you see fit.

Step Zero: Set up the Project

Unlike Evil Hangman, this assignment has a fair amount of starter code, mostly for the sample applications. Set it up like you would any other QT project.

Step One: Implement Basic Functionality

Now that you've gotten the project set up, it's time to start implementing `KDTree`. The `KDTree` implementation you'll be writing is actually a slight variant on the kd-tree structure described earlier in this handout that associates auxiliary data with each point. In a sense, your `KDTree` will act like a fancy `map` from points in space to values. For example, you could use a `KDTree` to map from longitude/latitude pairs to cities, from biometric data to disease prognosis, or from images to labels on those images.

Below is a partial specification of the `KDTree` class, highlighting the functions you'll need to write to get basic functionality working.

Basic (incomplete) `KDTree` interface

```
template <size_t N, typename ElemType> class KDTree {
public:
      KDTree();
      ~KDTree();

      size_t dimension() const;
      size_t size() const;
      bool   empty() const;

      void insert(const Point<N>& pt, const ElemType& value);

      bool contains(const Point<N>& pt) const;
      ElemType& operator[] (const Point<N>& pt);
      ElemType& at(const Point<N>& pt);
      const ElemType& at(const Point<N>& pt) const;
};
```

You may have noticed that `KDTree` has an unusual template signature:

```
template <size_t N, typename ElemType> class KDTree
```

You did not misread that – the `KDTree` implementation is parameterized over a `size_t` as well as a type. We have not discussed integer template arguments before, but they behave just like regular type template arguments. If you want to create a `KDTree` that maps from points in three-dimensional space to `string`s, you could declare it as

```
KDTree<3, string> myKDTree;
```

The keys in the `KDTree` are objects of type `Point<N>`, where `N` is the dimension of the `KDTree`. That is, a `KDTree<3, string>` uses `Point<3>`s as keys, a `KDTree<2, cityT>` would use `Point<2>`s as keys, etc. I've provided a fully-working implementation of `Point` in the starter code; it behaves like a fixed-size STL `vector<double>`. For example:

```
Point<3> pt;
pt[0] = 137.0;
pt[1] = 42.0;
pt[2] = 2.71828;
```

I advise looking over the `Point.h` header file to see what other functionality exists. You're free to extend this class however you feel, but you shouldn't need to do so for this assignment.

Given this detail about the `KDTree` and `Point` types, you should begin the assignment by implementing the following member functions on `KDTree`:

`KDTree();`	Constructs a new, empty `KDTree`.
`~KDTree();`	Destroys the `KDTree` and deallocates all its resources.
`size_t dimension() const;`	Returns the dimension of the points stored in the `KDTree`. (This is the value of the template parameter `N`).
`size_t size() const;` `bool empty() const;`	Returns the number of elements stored in the `KDTree` and whether or not it is empty, respectively.

`void`
`insert (const Point<N>& pt,`
`const ElemType& value);`
	Inserts the specified point into the `KDTree` with associated value `value`. If the point already exists in the `KDTree`, the old value is overwritten.

`bool` `contains(const Point<N>& pt) const;`	Returns whether the specified `Point` is contained in the `KDTree`.
`ElemType&` `operator[] (const Point<N>& pt);`	Returns a reference to the value associated with the point `pt`. If the point does not exist in the `KDTree`, it is added with the default value of `ElemType` as its value, and a reference to this new value is returned. This is the same behavior as the STL `map`'s `operator[]`. Note that this function does not have a `const` overload because the function may mutate the tree.

`ElemType&`
`at(const Point<N>& pt);`
 	
`const ElemType&`
`at(const Point<N>& pt) const;`
	Returns a reference to the value associated with the point `pt`, if it exists. If the point is not in the tree, then this function throws an `out_of_range` exception.
	
	This function *is* `const`-overloaded, since it does not change the tree.

Notice that the last four functions (`contains`, `operator[]`, and the two versions of `at`) all do some search of the `KDTree` looking for a particular value, differing only in their behavior when the point is not contained in the tree. `contains` returns false, `operator[]` adds a new element, and `at` throws an `out_of_range` exception. Rather than writing the code to traverse the tree four times and customizing the behavior when an element isn't found, I strongly suggest writing a helper function that searches the tree for a particular point, then returns a pointer to the node containing it. You can then implement these functions on top of this common subroutine. As an example, here's a simple implementation of `contains` that assumes the existence of a helper function `findNode`:

```
template <size_t N, typename ElemType>
bool KDTree<N, ElemType>::contains(const Point<N>& pt) const {
    return findNode(pt) != NULL;
}
```

To check whether you have your code working, you can run the first set of tests from the project `test-harness`. If these report any errors, be sure to correct them before moving on. You may also want to add tests of your own.

Step Two: Implement Nearest-Neighbor Lookup

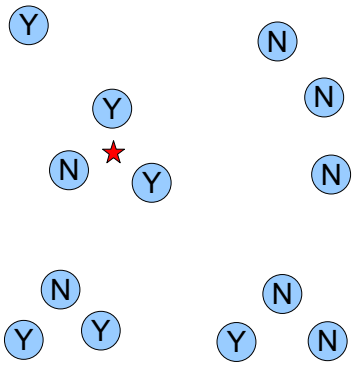
Now that you have the basic functionality ready, it's time to implement *k*-NN searches. Your next task is to implement the `kNNValue` function, which looks like this:

Extended (still incomplete) `KDTree` *interface*

```
template <size_t N, typename ElemType> class KDTree {
public:
      KDTree();
      ~KDTree();

      size_t dimension() const;
      size_t size() const;
      bool   empty() const;

      void insert(const Point<N>& pt, const ElemType& value);

      bool contains(const Point<N>& pt) const;
      ElemType& operator[] (const Point<N>& pt);
      ElemType& at(const Point<N>& pt);
      const ElemType& at(const Point<N>& pt) const;

      ElemType kNNValue(const Point<N>& pt) const;
};
```

This function takes in a point in space and a number of neighbors. It should then do a *k*-NN search in the kd-tree using `pt` as the test point. After doing so, it will have found a collection of the *k* near-

est points in space, along with the `ElemType` values associated with them. The return value of this function should be the most-frequently-occurring value associated with the *k*-nearest-neighbors of the test point. In the event of a tie, you can return any of the strings that tied for most frequent. For example, given the following collection of points (labeled with `string`s) and the indicated test point:



If we did a 3-NN lookup, the `kNNValue` function should return "Y".

The algorithm for doing a *k*-NN lookup assumes the existence of a bounded priority queue, and to make this assignment easier to complete I've provided you a `BoundedPQueue` class which does just that. You may want to look over its interface before starting work on this part of the assignment. You might be wondering why this function returns the most common label of the nearby points rather than the points themselves. This is mostly because the sample applications bundled with this project all use the *k*-NN search in the manner exported by this function and I didn't feel like needlessly duplicating code. ☺

The test harness contains two functions which test this function. Enable them and confirm that your code works before moving on to the next section.

Step Four: Implement Copy Functions

As written, the `KDTree` class has a destructor but no copy constructor or assignment operator. This means that C++ will provide the class default versions of these functions, which will cause a crashes. To prevent this, you will need to implement a copy constructor and assignment operator for the `KDTree` class. This results in the final interface of the `KDTree` class:

Complete `KDTree` interface

```
template <size_t N, typename ElemType> class KDTree {
public:
      KDTree();
      ~KDTree();
      KDTree(const KDTree& other);
      KDTree& operator= (const KDTree& other);

      size_t dimension() const;
      size_t size() const;
      bool   empty() const;

      void insert(const Point<N>& pt, const ElemType& value);

      bool contains(const Point<N>& pt) const;
```

```
      ElemType& operator[] (const Point<N>& pt);
      ElemType& at(const Point<N>& pt);
      const ElemType& at(const Point<N>& pt) const;

      ElemType kNNValue(const Point<N>& pt) const;
};
```

You are free to implement these functions as you see fit, but I strongly encourage you to read over Chapter 11 in the course reader before doing so. It is surprisingly easy to get these functions wrong, and you will want to ensure that you understand what to watch out for before you start coding them up.

The testing harness contains two tests that exercise the copy functions, one checking the basic functionality and one exclusively checking edge cases. Make sure that your implementation passes the tests before moving on.

Step Four: Run Sample Applications

Congratulations! You've just completed your `KDTree`. Take some time to play around with the sample applications that have been bundled with the starter code. There are three applications you can check out, each of which is described here:

- **Map Lookup**. This program presents a map of the world and lets you click on various locations. It then uses a 1-NN lookup to determine which country the selected location is in, along with the state/province within that country the location is in. The program uses official US government data from the National Geospatial-Intelligence Agency and US Geological Survey. If you'd like to retrieve the raw data files on which the data for this program is based, check out the following links:

  ftp://ftp.nga.mil/pub2/gns_data/geonames_dd_dms_date_20100503.zip
  http://geonames.usgs.gov/docs/stategaz/NationalFile_20091002.zip
  http://earth-info.nga.mil/gns/html/GEOPOLITICAL_CODES.xls

- **Color Naming**. Randall Munroe, author of the webcomic xkcd, ran a survey in which participants were shown a random color and asked to name that color. The results of the color survey were then released to the general public on his blag (http://blag.xkcd.com). The data set contains three million pairs of colors (encoded as RGB triplets) and the respondents' names for those colors. The Color Naming application pulls up the system color chooser dialog, lets clients choose colors, then reports the 3-NN name of that color based on a reduced subset of that data. If you'd like the raw data files I used to build the data set, you can find it online at the link below. Be warned – the data has not been filtered and some of the color names are *certainly* NSFW.

  http://xkcd.com/color/colorsurvey.tar.gz

- **Digit Classification**. Earlier in the quarter we made a brief foray into machine learning by writing a perceptron classifier that could recognize handwritten digits. An alternative means for performing this classification uses the *k*-NN algorithm. The Digit Classifier application presents you a canvas on which you can draw a digit between 0 and 9, then uses *k*-NN to guess what the digit you wrote was. The program has very good accuracy, though it

does make the occasional mistake. The raw data for this program was obtained from the MNIST database at

http://yann.lecun.com/exdb/mnist/

When running these sample programs, I suggest compiling them with optimization turned on and debugging turned off. Loading and processing megabytes of data takes time, and the overhead from debugging instrumentation can make the programs take a very long time to load. Even with optimization turned on, the programs can still take a while to load – the Color Naming program takes an especially long time to load since it has to build a kd-tree out of two million data points. Also, be aware that these programs will use a lot of RAM!

Advice, Tips, and Tricks

Here are a few specific pointers that might make your life a lot easier as you go through this assignment:

- ***Don't hesitate to ask questions!*** This assignment uses many of the C++ techniques we've seen over the past few weeks. If you're having trouble getting your code to compile, or can't remember what keyword you're supposed to be using somewhere, email the staff list (106l-staff@cs.stanford.edu), me, or go to the LaIR and I can try to point you in the right direction.

- ***This assignment is not as hard as it may seem.*** This handout is fairly dense, but the actual amount of code you need to write is not that great. You are only responsible for implementing a few functions, some of which can be implemented in a single line of code. If you take the time to think through how all the functions are related to one another, you can save yourself much coding effort by implementing the functions in terms of each other.

- ***Watch out for `typename` weirdness when implementing functions.*** Your implementation of `KDTree` will require the use of a nested type to represent nodes in the tree. If you write any private helper functions that return objects of this type, you will need to use the `typename` keyword when implementing those functions. For example, suppose that you define a helper struct called `Node` and then define a function that returns a `Node*`, as shown here:

```
private:
    struct Node {
        /* ... */
    };
    Node* findNode(const Point<N>& pt);
```

The implementation of this function would then have this signature:

```
template <size_t N, typename ElemType>
    typename KDTree<N, ElemType>::Node*
    KDTree<N, ElemType>::findNode(const Point<N>& pt);
```

That's a real mouthful, and unfortunately it's the only way to communicate to the compiler what you're trying to implement. Make sure you understand the use of `typename`, along with why the template arguments are duplicated in two places.

- ***Be careful about `const`-correctness.*** If you create any private member functions to assist in the implementations of the `KDTree` public interface, make sure those member functions are marked `const` where appropriate. In particular, `contains`, `at`, and `kNNValue` are `const`, so if they call any member functions, those functions must be marked `const` as well. You will get some fairly ferocious compiler errors if you try calling a non-`const` member function from a `const` member function, so be wary.

- ***Use `fabs` instead of `abs`.*** The `<cmath>` header file exports two similar-sounding functions to compute absolute value, `abs` and `fabs`. In this assignment, you should not use the `abs` function. `abs` works on integral values, so if you pass in a `double`, the returned value will be incorrectly rounded to an `int`. `fabs` is designed to work on `float`s and `double`s, and is a much more appropriate function.

- ***Remember the `const_cast`/`static_cast` trick.*** The `KDTree` contains two functions named `at` that differ only in their `const`ness. Rather than writing two copies of the same code, you can use the `const_cast`/`static_cast` trick to implement the non-`const` version in terms of the `const` version. Look over the lecture code for the `Vector` class for more details.

Extensions

If you're interested in sharpening your C++ skills, want to do more advanced operations on the kd-tree, or feel like spending a lazy Sunday coding away furiously, why not add some extensions to your `KDTree`? Below is a list of possible extension ideas, some of which are straightforward, while others will require significant time and effort. If you end up completing any of these, let me know and I'd be glad to look over what you've written!

- **Build the kd-tree more intelligently**. Traditionally, kd-trees are not built one element at a time, but rather from a complete data set all at once. To ensure that the tree is balanced, the elements are sorted by their first component, the median is used as the root of the tree, and the remaining elements are then recursively subdivided into children of the root node. Implement a new constructor for the `KDTree` class to build up the tree in this fashion.

- **Add support for other distance metrics**. When doing nearest-neighbor lookup, we use Euclidean distance as a measure of "closeness" between two points and try to find a point in the kd-tree with the least Euclidean distance to the test point. However, it's possible to use all sorts of other distance metrics, such as Manhattan distance or the maximum norm. Add support to KDTree to try out these new distance metrics. How does the behavior of the sample applications change?

- **Choose axes more intelligently**. The current kd-tree implementation cycles through which axis it splits on with each level of the tree. A more clever idea would be to split along the longest axis of the data set with the goal of spreading the points out more evenly. Update the `KDTree` class to use this functionality.

- **Add support for range searches**. One common operations on kd-trees is a *range search*, where the input is a rectangle in space and the output is the set of points in the kd-tree contained in that rectangle. This gives a much better algorithm for the CityFinder program

than the one we wrote earlier in the quarter. Research how to implement this function, then add it to `KDTree`.

- **Add support for element removal**. The `KDTree` you've written can have new elements added, but cannot remove existing elements. Develop an algorithm to remove arbirary points from a kd-tree, then update your `KDTree` interface to support this.

- **Be creative!** Think of any clever uses for a kd-tree? How about something you could do to make the kd-tree more efficient? If you have any ideas you'd like to try out, by all means go for it and I'd love to see what you come up with.

Deliverables

To submit the assignment, upload your updated `KDTree.h` file, along with any other files you might have edited, to paperless. If you've added any extensions or special features I should be aware of, let your grader know in your comments. I would also appreciate it if you offered some feedback on this assignment as well as the class as a whole – was it interesting? Too easy? Too hard? Just right? Finally, pat yourself on the back – you've just completed the last assignment of CS106L and are now a veteran C++ programmer. Congratulations!

Good luck!