

Kd-tree insertion

will be Point

will be Point*

- with a templated kdNode defined as follows:

```
template <typename T, typename P >
```

```
class kdTree {
```

```
private:
```

```
struct kdNode {
```

```
    P data; // used to store Point* pointers to data
```

```
    T min, max; // the min, max of type Point
```

```
    // (node's local copies of  
    // bounding box extents)
```

```
    kdNode *left, *right; // subtrees
```

```
    int axis; // axis this node splits on (x, y)
```

```
    kdNode (const P & d = P(), \
```

```
            const T & in = T(), \
```

```
            const T & ix = T(), \
```

```
            kdNode *l = NULL, \
```

```
            kdNode *r = NULL, \
```

```
            int a = 0): \
```

```
        data(d), min(in), max(ix), left(l), right(r),
```

```
        axis(a) {};
```

```
}
```

```
}
```

- Note: template <typename T, typename P>
let's use operate on generic type &
pointer to generic type
- if you find template confusing, you can
define wrapper in form of point: point*

```
e.g.) class kdTree {  
    struct Node {  
        Point * data,  
        Point min, max  
        :  
    }  
};
```

- if using templates, don't forget to specialize the
kdTree class (at bottom of implementation, kdTree.cpp).

```
template class kdTree<Point, Point*>;  
template std::ostream& operator<<(std::ostream&,  
    const kdTree<Point, Point*>&);
```

- don't forget forward declaration at top
of kdTree.h as well.

⊗ final operand operators (< <> (...))
↑
this says that the data
type in arguments is itself templated

- a Point class is just a simple 2D point;
(no really)

class Point {

private:

Vector<double>

point;

} in its own
point.h of
point.cpp
files

public:

Point (double x=0.0, double y=0.0) {

point.push_back(x);

point.push_back(y); }

// hij }

const double & operator[] (int k) const

{ return point[k]; }

// access

double & operator[] (int k)

{ return point[k]; }

// mutator

// // of friends

// plus methods

int dim() const { return point.size(); }

}

- in main.cpp, declare the kdTree

```
kdTree<Point, Point> kdTree;
```

- in kdTree.h:

```
class kdTree {  
public:  
    kdTree(): root(NULL) {} // constructor  
    :  
private:  
    kdNode * root;  
};
```

the basic insert mechanism (p. 551)

↑
min differs

- in kdTree.h:

public:

```
void insert (std::vector<P>& x,  
            const T& min,  
            const T& max)
```

```
{ root = insert (root, x, min, max, 0); }
```

↑
print insert returns node
(diff + from root)

↑
depth level

private:

```
kdNode * insert (kdNode *  
                std::vector<P>& x,  
                const T&  
                const T&  
                int);
```

- in kdTree.cpp (among other things), we have:

```

template < typename T, typename P >
typename kdTree<T,P>::kdNode &
kdTree<T,P>::insert (kdNode && t,
                    std::vector<P> & x,
                    const T & min,
                    const T & max,
                    int d)

```

}

P median; // median point (pointer to)
T -min, -max; // bounding box rectangle —
 each is a point,
 so -min has {x,y}
 -max has {x,y}

```

typename std::vector<P> left, right;
int     axis = d % x[0].dim();
// don't do this if x.empty()!!
if (x.empty()) return NULL; // list empty, end of recursion

```

// find median — can use Geit's code

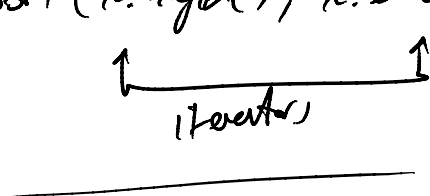
// on std::vector use STL's sort rather

// need #include <algorithm>

```

sort(x.begin(), x.end(), PointAxisCompare(axis));

```



↑
what's this?
• Pointer, a function object

can't
 simply use
 operator < here
 because we need to toggle
 coordinates:

a functor, a function object
 (not a pointer to a fun,
 but sort of similar)

sometimes we want

`point1[0] < point2[0]` // sort on x

sometimes

`point1[1] < point2[1]` // sort on y

!!! return to this functor below

`int m = x.size() / 2;` // index to median

// now construct left & right lists

`for (int i = 0; i < (int)x.size(); i++)`

`if (i < m) left.push_back(x[i]);`

`else if (i > m) right.push_back(x[i]);`

`else media = x[m]; // (i == m)`

Note: dealing with pointers —
 if we have point values,
 there'd be a lot of copies
 going on here

// create new node

`TreeNode node = new TreeNode(media, min, max,
 null, null, d+1);`

// add left subtree

[set values of `_min`, `_max`] !!!

`node->left = insert(node, left, _min, _max, d+1);`

~~Case 0~~
node → left = insert(node, left, -min, -max, d+1);

// add right subtree

[set values -min, -max] !!

node → right = insert(node, right, -min, -max, d+1);

return node;

}

- what is the Point Axis Compare ?

- It's function object, a class that has:

```
bool operator()
```

- where to define it?

at bottom of point.h file, below
class Point { ... };

```
class PointAxisCompare  
{
```

```
public:  
    PointAxisCompare(int in_axis = 0): axis(in_axis) {};
```

function constructor

```
    bool operator()(const Point& p1,  
                    const Point& p2) const  
    { return (p1[axis] < p2[axis]); }
```

```
    bool operator()(const Point* & p1,  
                    const Point* & p2) const  
    { return ((*p1)[axis] < (*p2)[axis]); }
```

private:

```
    int axis;
```

```
}
```