Secrets of the kd-tree revealed

① the function object
(a.k.a. the functor)

point_c

typename<T, P, C> kdtree_t...

point_t point_t *

originally based on text's
version on p. 34-35
("comparator")

class point_c

{

public:

a) bool operator( )

(const point_t & p1,

const point_t & p2)

const

{

return (p1[axis] < p2[axis])

}

an obj (class) with just one member function '( )'

can be 0, 1, or 2 for photons

5) private:
    int axis;
}; // point_c class def.

where is this set / initialized?
in point_c constructor

c) point_c(int inaxis = 0):
    axis(inaxis) { };

d) one more version of operator() is needed for point_t* pointers

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

With point_t defined,

```
kdtree_t <point_t,
          point_t*,
          point_t>
```

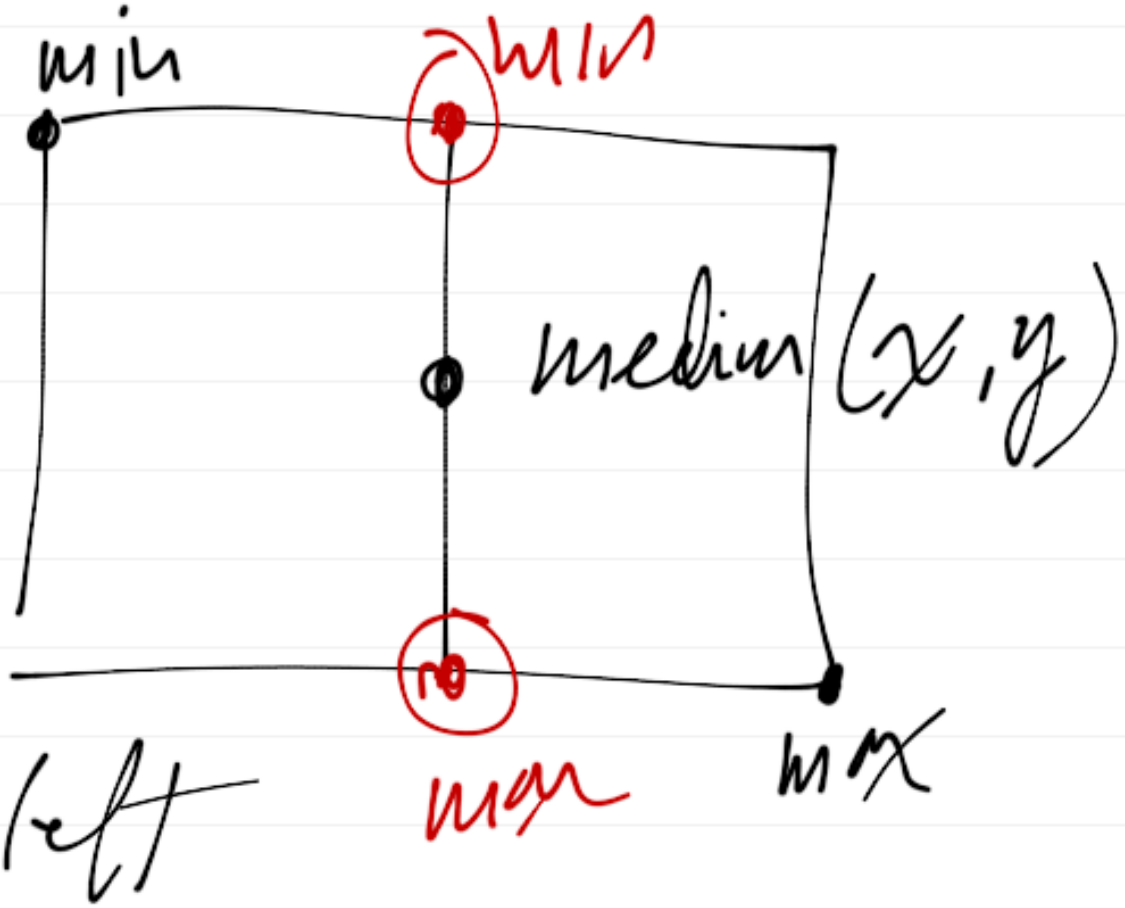
② kdtree insertion &
min & max

```
kdtree_t<T, I, C> ::
  insert(kdnode_t* &t,
    std::vector<I> &x,
    const T &min,
    const T &max,
    int d)
{
```

// local vars (at each
tree level)

T _min, _max;
// get sent to next
tree level in recursive
call
// code to find median,
split on axis (0, 1, 2)

e.g. median is midpoint in vector x, sorted on axis

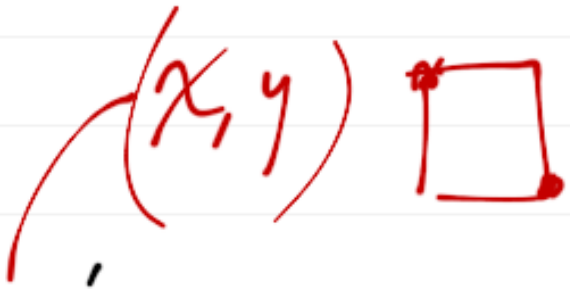


// insert left

```
_min = min;
_max = max;
_mx[axis] = (*median)[axis];
```

```
// insert right
_ min = mid;
_ max = max;
_ min[axis] = (+median)[axis]
```

range query

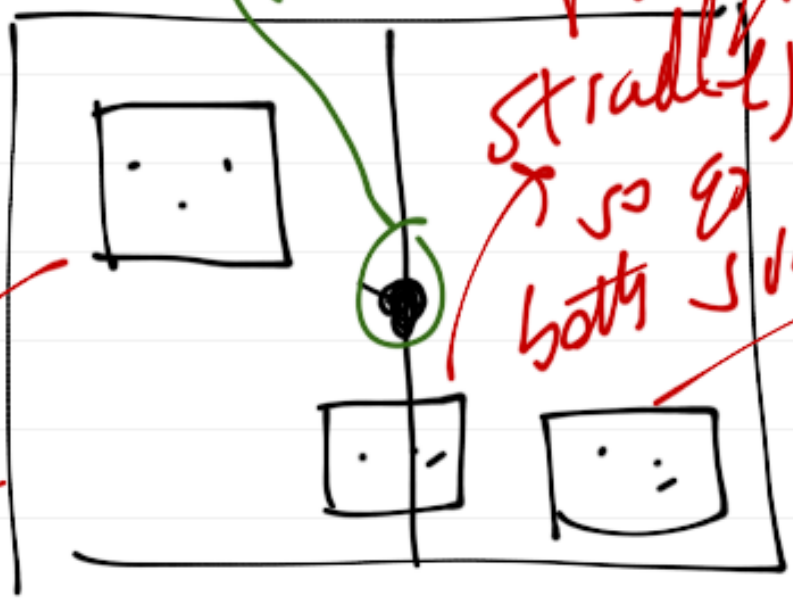



```
range(const T& min,
      const T& max,
      std::vector<P>& p)
{ range(root, min, max, p); }
```

range alg:

range(kdnode_t *kd,
... min, ... max, ... p)

3 cases:



if (!t) return; // step 1. recursive end check

// step 2:
test to see if data is within range

if ((min[0] <= (*t).data[0])
&& (*t).data[0] <= max[0])
&& (min[1] <= (*t).data[1])
&& (*t).data[1] <= max[1])
(*p).push_back(t->data);

...simplify:

```
for(int i=0; i<t->data->dim(); ++i)
  if( ((t->data)[i] < min(i))
    || ((max(i) < t->data)[i]))
      { in_range = false; break; }
if (in_range)
  p.push_back(t->data);
```

```
if( (*t->delta)[t->axis] >=
        min[t->axis]) &&
    (*t->delta)[t->axis] <=
        max[t->axis]) )
// range has intersects split
plane (standeller split axis)
range(t->left, min, max, p)
range(t->right, min, max, p)
```

```
else if ((*t->data)[t->axis]
        >= max(t->axis))
    range(t->left, min, max, p)
else if ((*t->data)[t->axis]
        <= min[t->axis])
    range(t->right, min, max, p)
```