Secrets of the Kd-Tree revealed

① The function object (aka the functor)

~~PointAxisCompare~~

point_c

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

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

{ public:

a) bool operator()
(const point_t & p1,
const point_t & p2)
const

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

can be 0, 1, 2

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() N needed for point_t * pointers

bool operator()
(const point_t * const &p1,
const point_t * const &p2)
const
{ return ... }

With point_c defined

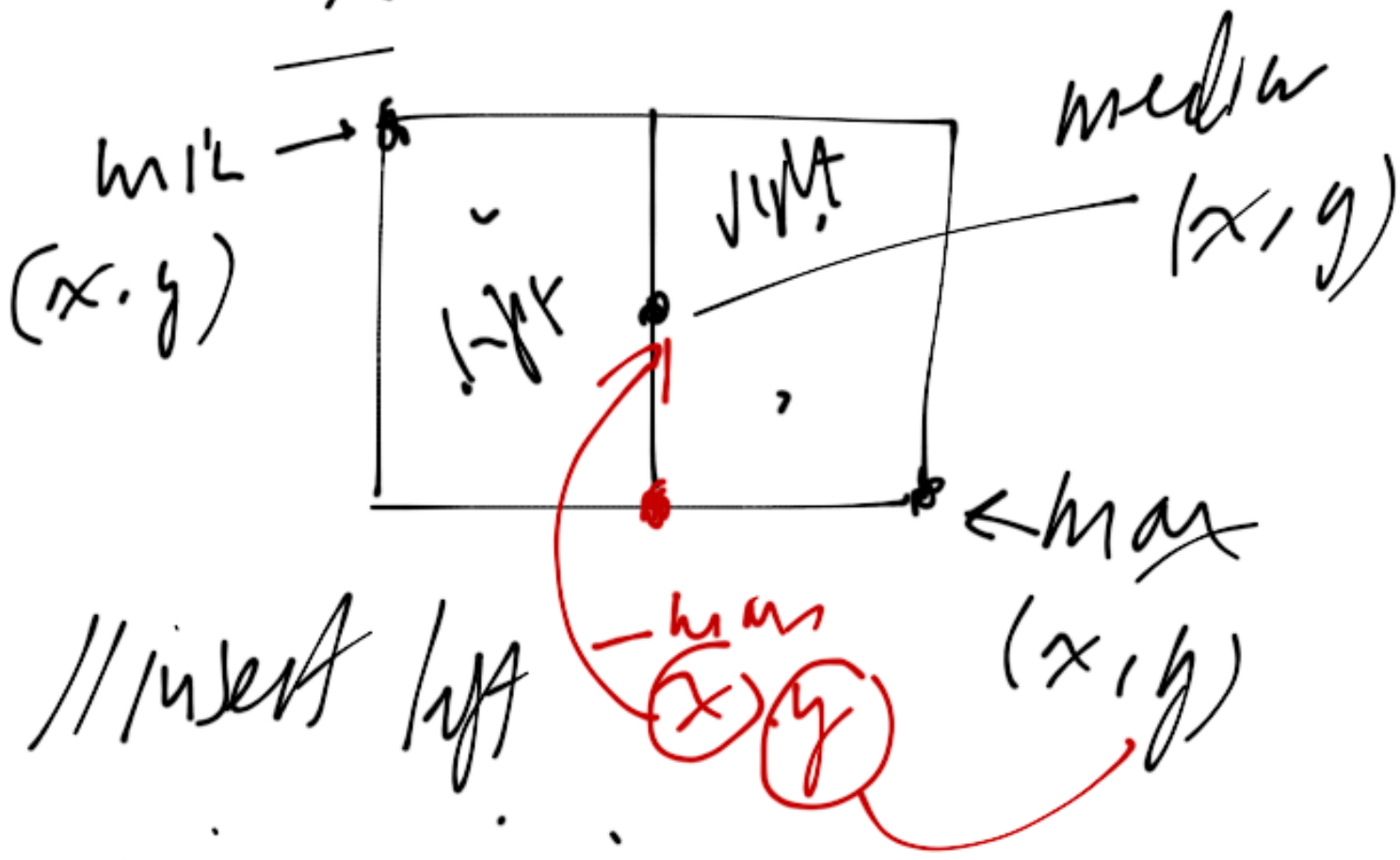
```
kdtree_t<point_t,
         point_p,
         point_c> kdtree;
```

② kdtree insertion &
min & max

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

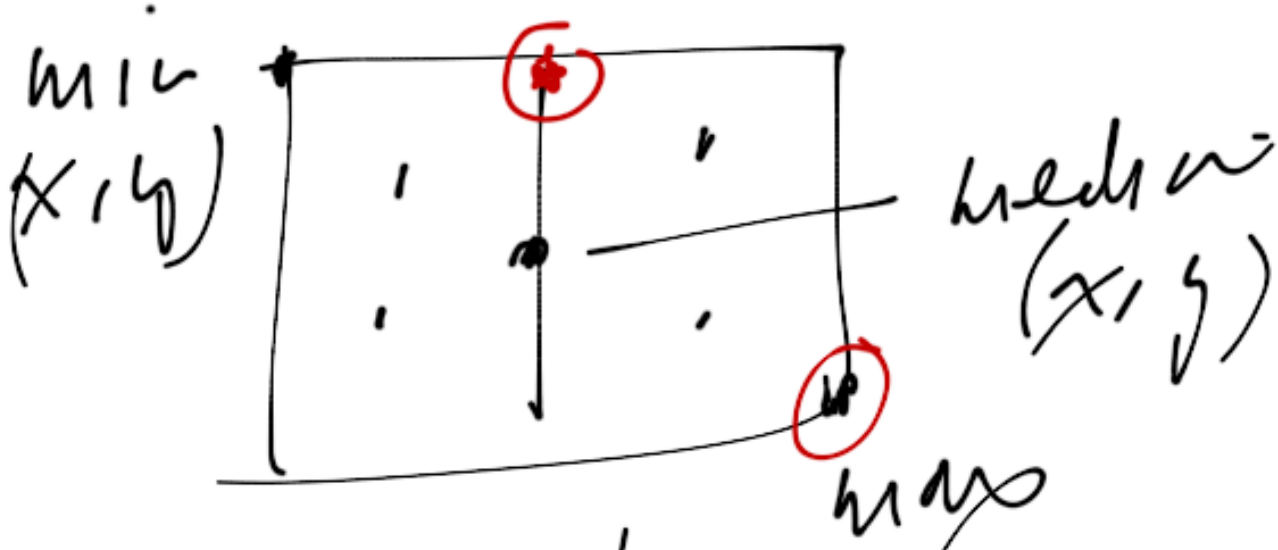
// local vars (at each
tree level)
T _min, _max;
// what gets sent
to next tree level
(in recursive call to
insert
// code to find
median, split on
axis (0, 1, 2)

e.g. median is midpoint in vertical or horizontal axis



//insert left

- min = min(min, x);
- max = max(max, x);
- max[axis] = (xmedian[axis]



//insert right

- min = min
- max = max
- min[axis] = (*median)[axis]

③ nn query (logic)

```
dist = q.distance(t→data)
```

- q ← query point_t
- t ← tree node ptr to point_t

```
if (dist < r) {
    r = dist;
    p = t→data;
} // traverse down L, R subtrees
```

kNN query : O(log(n))

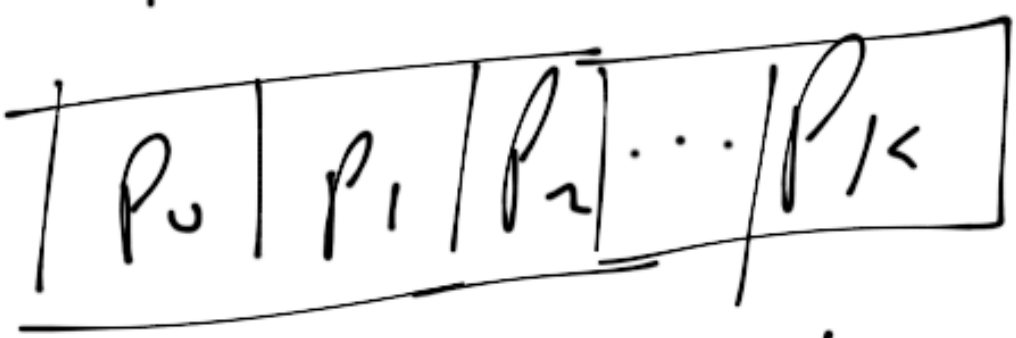
dist = q.distance(t->data)

if (dist < r) {

r = dist

p = t->data

p is now a vector !



for K closest points

— need to find K^{th} closest

distance : just look at p_K

assuming list is sorted

pK is just p.back()

p

std::vector

if(dist < q.distance(p.back()))

r = dist

put t → data

on list, maintain

list of size K

} CAVEATS: list may

have fewer than K, must be sorted

```
dist = q.distance(t -> data)
// do we keep this t or not?
if size of list < K {
    // insert t into
       sorted list
    if (list is empty ||
        dist > q.distance(
                 p.back()))
        put t -> data at end
} else {
    insert t -> data
    into proper position
}
```

else { // list has K or if already

// only insert to our list if dist < last node's or list

if (dist < g.distance(p.back())

insert t → data into proper position

done only in else (because we have K)

if (p.size() > k)

p.pop_back()

r = q.distance(p.back())