- kd-tree insertion
- kd-tree nn, knn queries

- basic insertion (p. 551)

```
public:
void insert(typename P (photon_t *)
    std::vector<P> & x,
    const T & min,
    const T & max)
{ root = insert(root, x, min, max, 0); }
```

recursion level

```
private:
  KdNode_t *root;
  KdNode_t *insert(
      KdNode_t *&,
    std::vector<P> &,
      const T &, const T &,
      int);
```

in KdTree.cpp :

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

↳ comparison function object or functor

```
typename KdTree_t<T,P,C>::
    KdNode_t *
KdTree_t<T,P,C>::insert(
    KdNode_t*t, ...x, ...m1, m2, d)
```

```
{
  int axis =
      x.empty() ? 0 :
          d % x[0]->dim();
```

in photon_t class, hardcoded to 3

```
  int m = 0; // median index
  P median; // ptr to median element
  T _min, _max; // bounding
                // not. of subspaces
```

```
std::vector<P> left, right;
  std::vector<P>::iterator it;
(typename)
  if (x.empty()) return NULL;
// find median by sorting
// (not super efficient)
  (expect O(n lg n) — there
is a find-kth-median alg.
that runs in O(n)
  — see Quicksort
```

```
sort(x.begin(), x.end(),
     C(axis));
```

the comparison functor

```
#include <algorithm>
// get median
m = x.size()/2;
```

```
// create left & right subtrees
for (int i = 0; i < x.size(); i++) {
    if (i < m)
        left.push_back(x[i]);
    else if (i > m)
        right.push_back(x[i]);
    else
        median = x[m];
}
```

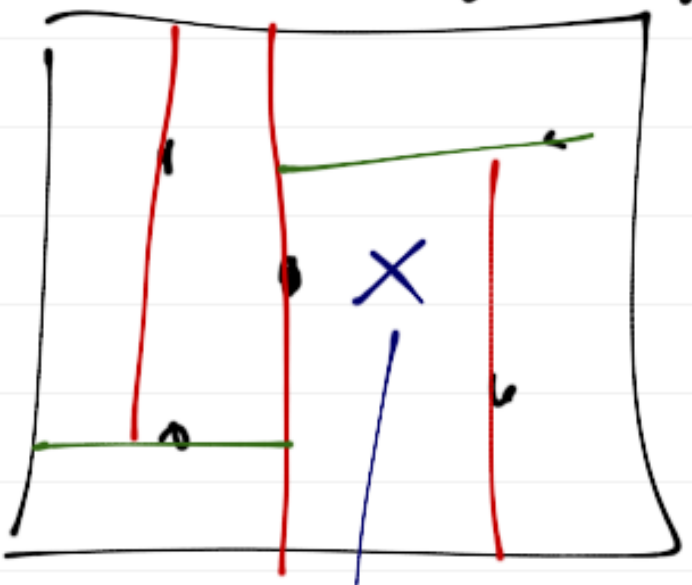
```
// create new node
Kdnode_t * node =
  new Kdnode_t(median,
               min,
               max,
               NULL,
               NULL,
               axis);
```

```
// recursively add left tree
_min =

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

```
// recursively add right tree
_min =
_max =
_min[axis] =
node->right = insertNode(
        right, _min,
        _max, d+1);
return node;
}
```

- nearest-neighbor query



query point

- input:
  q, the query element
  (same type as
  what's in tree, i.e.,
  fake point or platoon)
  t, tree node (root to start), and

and &r, dist. thus far
to closest element or
tree (set to ∞ initially),
and the node or p &

to result, nearest node
(or return value)

input: t, g, p, r

t: tree node
g: query graph
p: result (out)
r: dist to root

```
if (t == NULL) return;
    // end recursion
dist = g.distance(t->data);
// compute distance from g
   to current node t
```

$$dist = \sqrt{(q[0] - (*t \to data)[0])^2 + (q[1] - (*t \to data)[1])^2 + (q[2] - (*t \to data)[2])^2}$$

```
if (dist < r) {
    p = t -> data;
}
// as we descend tree
test against each node
we encounter
```

// traverse down "closer" side of tree

axis = t → axis;

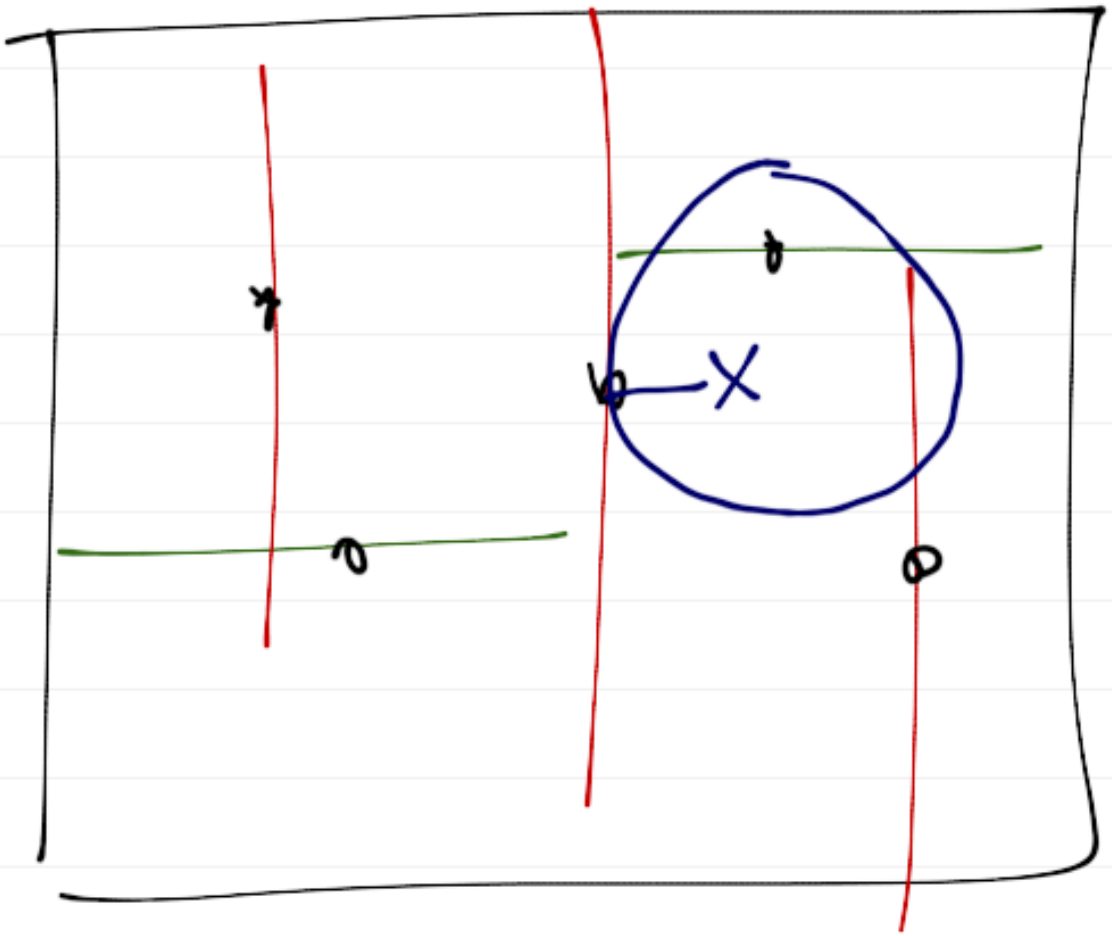
if (q[axis] <= (*t → data)[axis])

{ nn(t → left, q, p, r);

// as we return, check to see if circle defined by q & r intersects farther side of tree

```
if (q[axis] + r >
    (t -> data)[axis])
  nn(t -> right, q, p, r)
} else {
  nn(t -> right, q, p, r);
  if (q[axis] - r <=
      (t -> data)[axis])
    nn(t -> left, q, p, r)
  }
}
```

- see Thinh Nguyen's lecture



- K-nearest neighbor:
  - instead of only searching within a circle whose radius is "closest distance yet", search within a circle whose radius is K^{th} closest yet found. UNTIL K points have been found, keep distance at ∞

- otherwise, similar to our query, just keep a sorted list of K elements found thus far

P argument is now a std::vector<P>

→ use insertion sort

```
:
dist = q.distance(t->data)
if(p.size() < k)
    if(p.empty() ||
       dist > q.distance(
                  p.back()))
        p.push_back(t->data);
    else {
```

```
// insert into list
for (it = p.begin();
     it != p.end();
     it++) {
  if (dist < g.distance(
                  *it)) {
    p.insert(it, 1,
             t->data);
    break;
  }
}
```

— above if statement was

```
if (p.size() < K) {
    :
} else {
    // insert into list
    // repeat as done above
}
```

// only if there are K or list

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