— kd-tree insertion

— kd-tree nn, knn queries

— basic insertion mechanism (p. 551)

typename P (photon_t *)

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

```
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 >

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

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

in photon_t
class, hardwired
to 3

```
      int   m = 0; //median
                   index
      P     median;
      T     _min, _max;
// bounding vol. of subspaces
```

```
std::vector<P> left, right;

typename std::vector<P>::
        iterator it;

if (x.empty()) return NULL;

// find median by sorting
// (not very efficient)

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

functor

```
#include <algorithm>
```

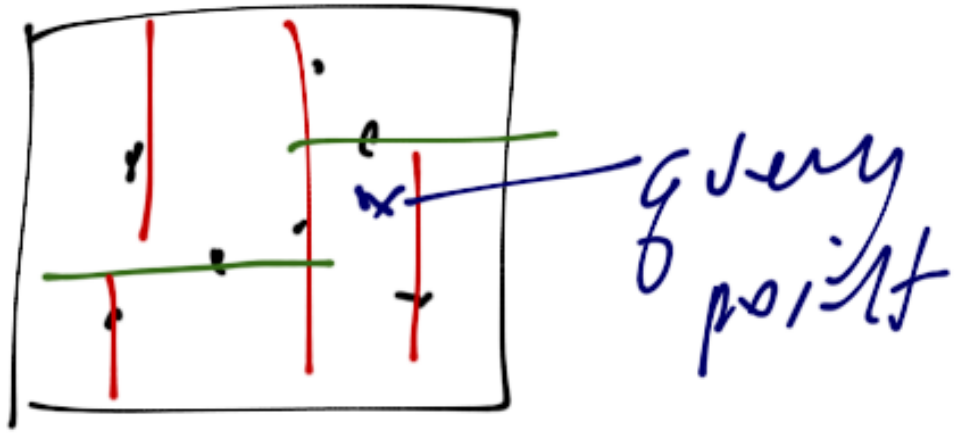
in photon, ...

```
// get median
m = x.size()/2;

// create left & right subtrees

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 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 = insert(node,
                 right, _min,
return node;     _max, d+1);
```

- Nearest-neighbor query



- the query itself:

input: q, the query node (fake photon),

t, the node (root to start with),

& r, dist. thus far to closest

(dist set to ∞ initially)

$\forall$ a node, or PQ

(pointer reference to nearest node)

if (t==NULL) return;

//end of recursion

dist = q.distance(t→data)

// compute distance from q

to current node t

$$dist = \sqrt{(q[0] - (*t \to data)[0])^2 + (q[1] - \text{''}\ [1])^2 + (q[2] - \text{''}\ [2])^2}$$

```
if (dist < r) {
    r = dist;
    p = t -> data;
}
```

(as we descend tree test against each node we encounter

// traverse down "closer" side of tree

axis = t -> axis;

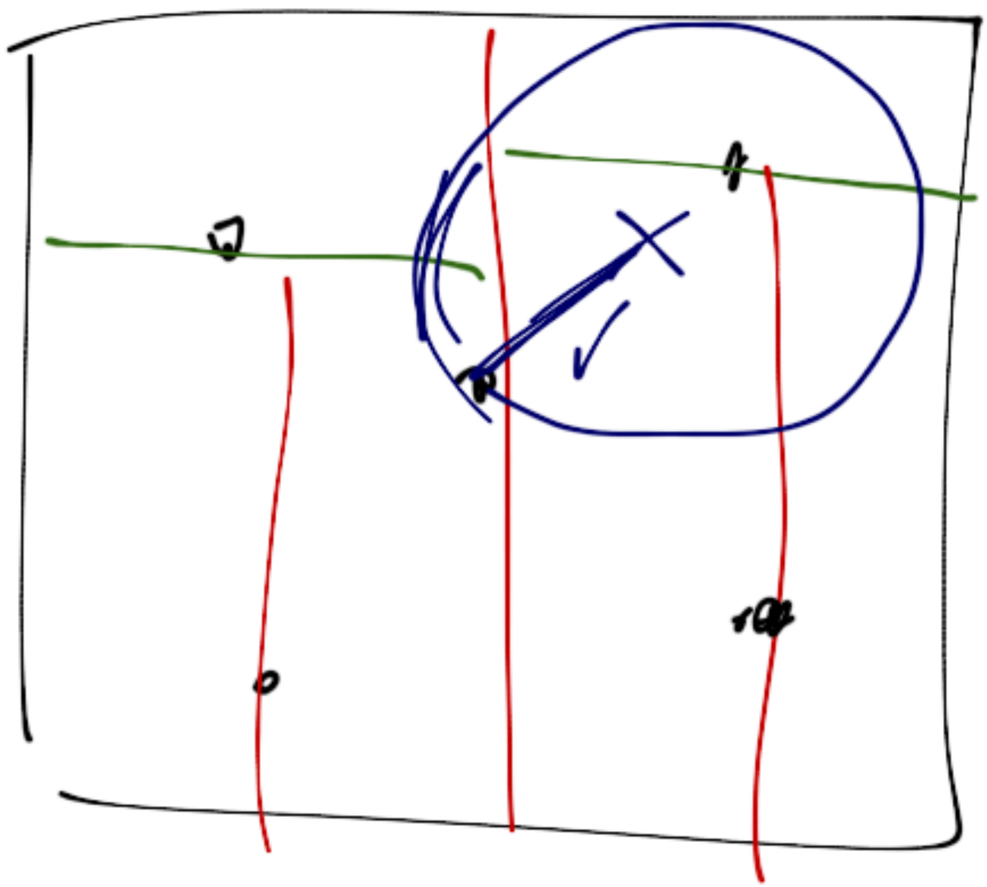
if (q[axis] <= (t -> data)[axis])

nn(t -> left, q, p, r);

// as we return, check to see if circle def. 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 neighbors:

  - 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
NN query, just keep
a sorted list of K
points found thus far

⋮

```
dist = q.distance(t->data)

if ((int)p.size() < k)

   if (p.empty() ||

      (dist > q.distance
              (p.back()))

      p.push_back(t->data)
else {
```

what we find: a vector of photon pts

//insert into list
```
for(pit = p.begin();
    pit != p.end();
    pit++){
  p.insert(pit, 1,
           t->data)
}
```
⋮

//insert node into sorted list.

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