

```

for (int k=0; k < pts.size(); ++k) Vector<Point> pts;
for (int i=0; i < 2; ++i) mean;
for (int j=0; j < 2; ++j) mean.dim() =
cov[i][j] += (pts[k][i] - mean[i]) * (pts[k][j] - mean[j]);

```

$u \times u$   
 where  $u \equiv$  dimensionality of  $pts = 2$   
 $mean.dim()$

$O(u)$   
 $O(2)$   
 $O(2)$  }  $O(u) \times O(2) \times O(2)$

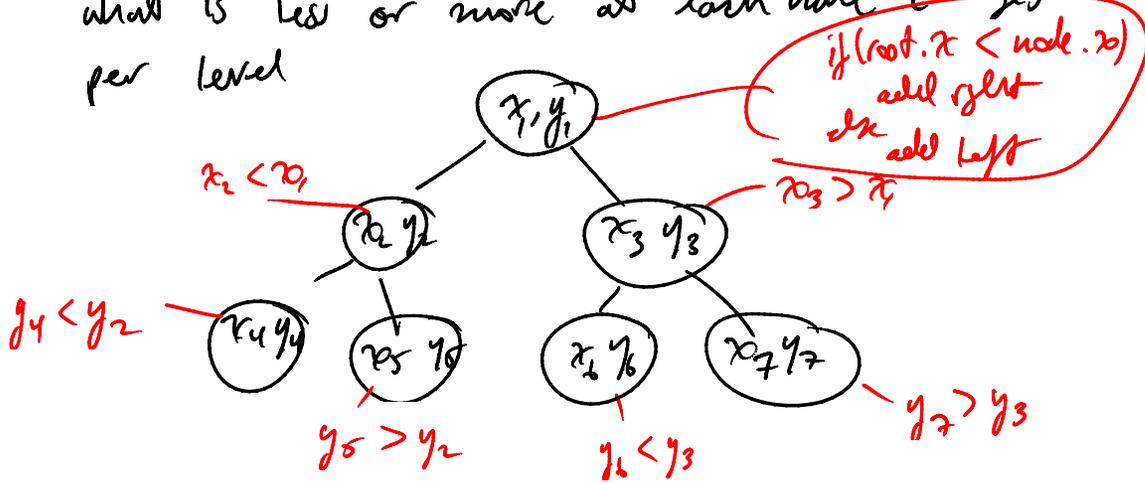
$||$   
 cov.row()  
 cov.col()  
 matrix divide  
 cov(2,2);

Next topic: kd-trees

- p. 549 of text

- what are kd-trees:

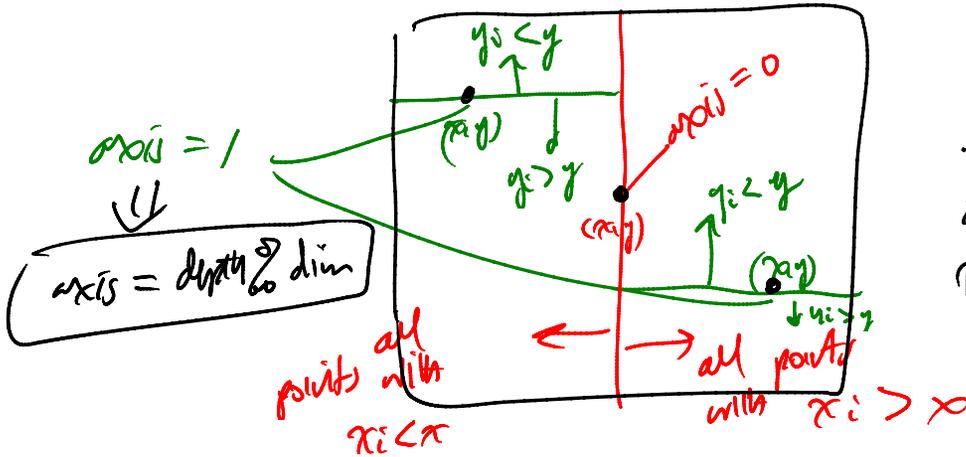
- just binary search trees, but criterion on what is less or more at each node changes per level



- Comparison of node values changes at each level  
(coord's)

$x \rightarrow y \rightarrow x \rightarrow y \rightarrow x \rightarrow y \rightarrow \dots$

- in our case, this comparison depends on current axis



↓  
each node should keep track of which axis it compared against

so --

```
struct KDNode {
```

```
    int axis; // which coord this node splits on
```

```
    Point element // what's stored at this node  
                  (should this be a pointer?)
```

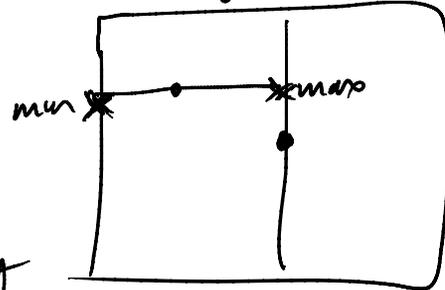
```
    Point min, max // - bounding box - ignore for now
```

```
    struct KDNode * left;
```

```
    struct KDNode * right;
```

```
    struct KDNode * parent;
```

}  
if null  
then root



// all we need is a constructor

```
KDNode (int axis)
```

```
    const Point & el,
```

```
    const Point & imin,
```

```
    const Point & imax,
```

```
    KDNode * lt,
```

```
    KDNode * rt,
```

```
    KDNode * parent)
```

3

What about KDTree class?

- basically all it needs is a root node

```
class KDTree {
```

```
private:
```

```
    KDNode *root;
```

```
public:
```

```
    KDTree() : root(NULL) {};
```

```
    KDTree(vector<Point> pts, Point min, Point max)
```

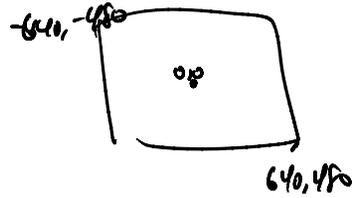
```
    :
```

- what I want to do:

Point min(-640, -480), max(640, 480)

vector<Point> pts;

KDTree\* kdTree(NULL);



```
// read in points  
while (std::cin >> ...)
```

```
// remove duplicates  
...
```

```
kdTree = new KDTree(pts, min, max);
```

*constructor*

- constructor :

```
KDTree :: KDTree (vector<Point> pts, Point min, Point max)
```

```
{  
    int depth = 0;  
    if (pts.empty()) { root = NULL; return; }
```

```
    root = insert(NULL, pts, depth, min, max)  
}
```

↓  
recursive function operating on KD-trees  
(pretty close to what's in the text on p. 551)

```
KDNode * KDTree::insert (KDNode * parent, vector<Point * > pts,  
                          int depth, Point min, Point max)
```

```
{  
  if (pts.empty()) return NULL;
```

```
  // decide which coord to split on
```

```
  int axis = depth % pts[0] -> dim();
```

```
  if (pts.size() == 1)
```

```
    return (new KDNode (axis, pts[0], min, max, NULL, NULL,  
                       parent) );
```

```
// sort the points on x, y, or z - coord;  
vector<Point> spoints = pts; // make copy of incoming  
                             point list
```

```
sort(spoints.begin(), spoints.end(), Point::Compare(axis));
```

```
int median = spoints.size()/2;
```

```
// make two lists (left, right)
```

```
vector<Point> lpoint, rpoint;
```

```
for (int i=0; i < median; ++i)
```

```
lpoint[i] = spoint[i] // core dump since lpoint is empty  
lpoint.push_back(spoint[i]);
```

```
for (int i=median+1; i < (int) spoint.size(); ++i)  
    rpoint.push_back(spoint[i]);
```

↑  
note how we  
sort on x, y, or  
z at each level

```
KNode *node = new KNode ( arr[i],  
                          spoint (median),  
                          min, max, NULL, NULL,  
                          parent );
```

```
node->left = insert (node,  
                    lpoint, depth+1, min, max)
```

```
node->right = insert (node, rpoint,  
                     depth+1, min, max)
```

```
return node;
```

```
}
```

left &  
right  
subtrees -  
not yet  
created

these need to  
be set - you  
can ignore though.

