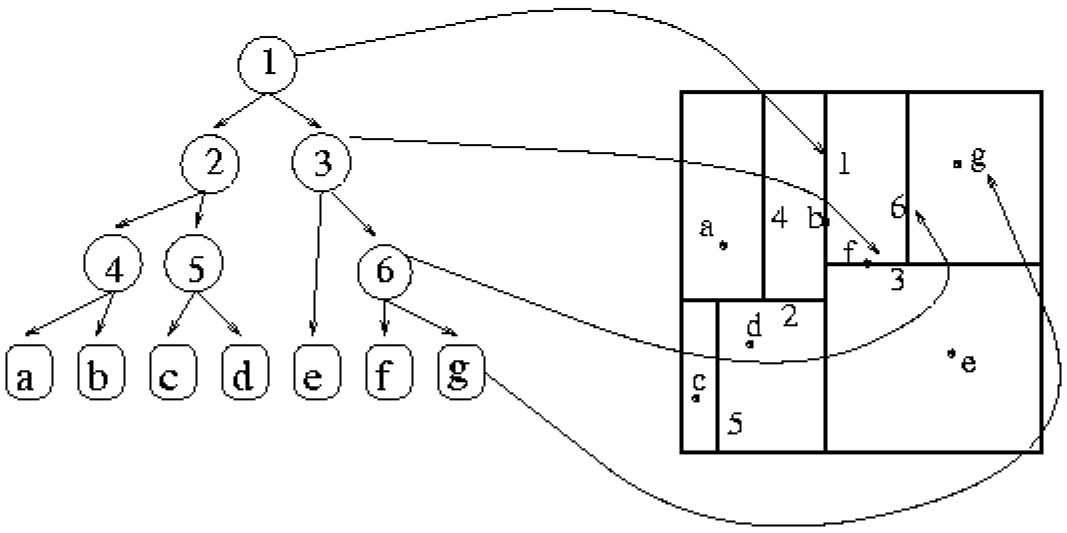
KD-Trees

We are given a set of points $P = \{p^1, p^2, ..., p^n\} \subseteq \mathbb{R}^d$ and we wanto to construct a structure for storing, analyzing and managing points efficiently.We can construct a KD-Tree in the following manner:

- choose a splitting axis i
- compute the median of the points across this axis $med(\{p_i^1, ..., p_i^n\})$
- divide out points in 2 sets:
  - $P^+ = \{p \in P : p_i > m\}$
  - $P^- = \{p \in P : p_i \leq m\}$
- recursively apply the same to P^+ and P^-
- contstruct the node v that stores the pointer to the children, m, i and maybe the bounding box of P
- stopping criterion $|P| = 1$

Note: i is called the **discriminator**, the couple (m, i) defines the **splitting plane**



since we take every time the median, so we spliit in half, given **n** points the depth of the tree is $O(logn)$

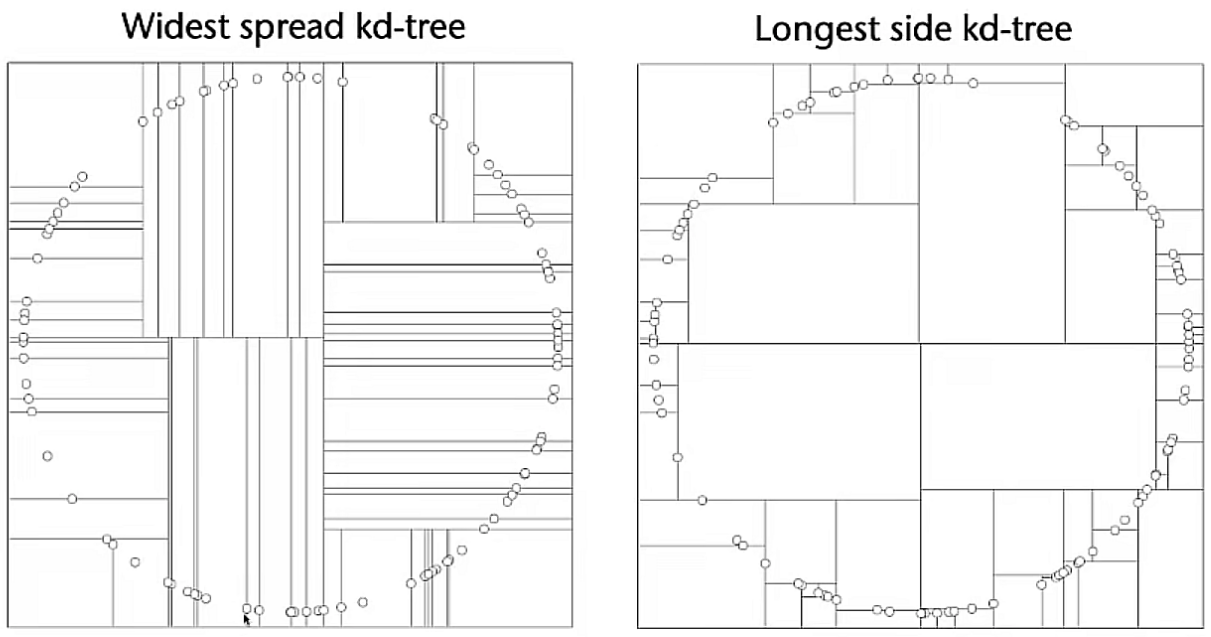
Construction complexity

we are splitting along the median on the discriminator axis, but every time we need to keep a list of nodes coordinates for every axis sorted and linked between them. this means that when we split we need to sort the lists again. we have to make d sorting, and a standard quick algorithm for sorting need $O(n\log n)$, this gives us a complexity of $O(d\cdot n\log n)$. when we apply the recursion we need a time $O(dn)$ for the creation of the lists and then the recursive step is applied 2 times to the halves of the set, so we get $T(n)=O(dn)+2T(\frac{n}{2})$, summing up the 2 parts we get that $T(n)=O(d\cdot n\log n)$. **More,** to store the tree we just need to store the points, so we get a space complexity of $O(n)$

Some variation of the kd-tree

- we can store the median point into he node, leaving leaves only for “free” points
- **Binning:** we can stop when $|P|\leq b$, a certain threshold
- **longest spread kd-tree:** we can choose the axis of the discriminator such that the bounding box of $P(v)$ has the largest extent along i
- **longest side kd-tree:** we can split along the largest extent of $P(v)$

NOTE: $P(v)$ is the set of points into the region of the node v, in a certain way like the square of a point for the octrees



Along the dimension
with the widest spread of the *points*,
at the median of the coords

Along the dimension with the longest
side of the *region*,
at the coordinate closest to the middle
of the extent of the region

Applications

the nearest neighbors problem

The nearest neighbor problem is a classic problem in computational geometry, which asks for the closest point to a given point in a set of points. In other words, given a set of points in some space $P \subseteq \mathbb{R}^d$, and a query point $q \in \mathbb{R}^d$, the nearest neighbor problem is to determine which point in the set $p^*$is closest to the query point, i.e. $p^* \in P | \forall p \in P : ||p^* - q|| \leq ||p - q||$.

ANSATZ

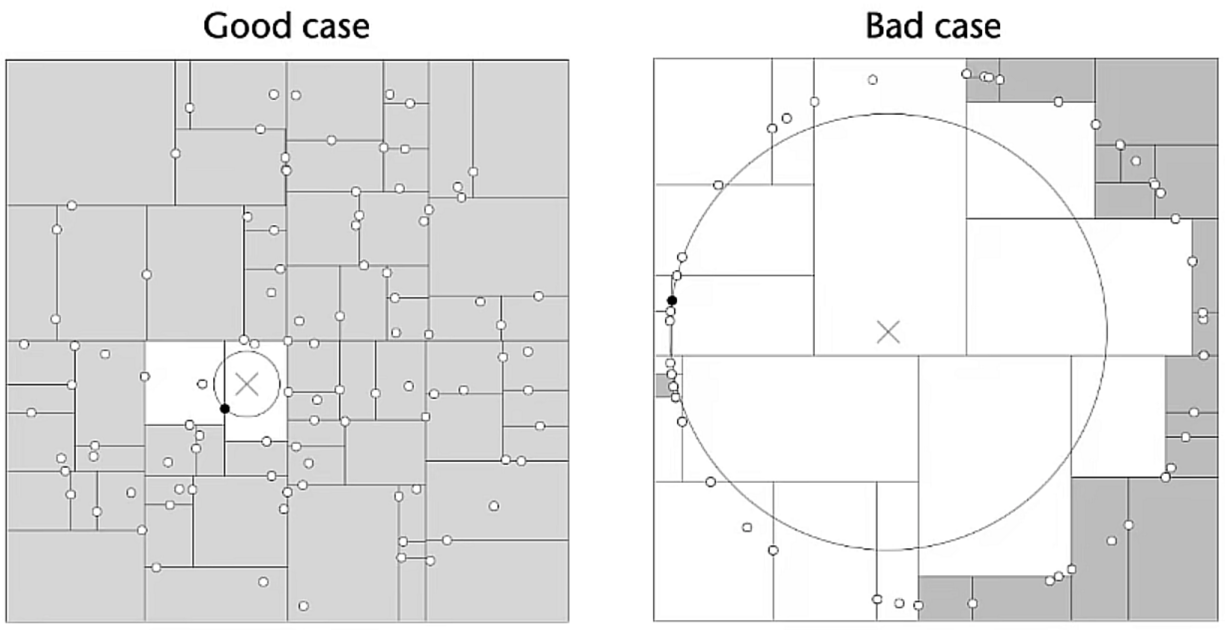
1. Start with a ball centered at the query point with an infinitely large radius.
2. Begin at the root node and move down the tree. At each node, first check if the point in that node is closer to the query point than the radius of the ball. If it is,

update the radius of the ball to be the distance from the query point to the current node.

3. Decide which of the two child nodes to move to by comparing the splitting coordinate of the current node with the corresponding coordinate of the query point. If the point's coordinate is less than the current node's, move to the left child, otherwise move to the right.
4. After reaching a leaf node, backtrack to check the other half-space, but only if it could intersect with your ball. we can quickly determine this by seeing if the distance from the splitting coordinate to the query point is less than the radius of the ball.
5. Repeat this process until we have traversed all relevant areas of the tree. The current best match is the nearest neighbor, and the final radius of the ball is the distance to the nearest neighbor.

This process is also known as a **"priority search"**. The ball shrinks as better matches are found, and this shrinking ball helps to avoid unnecessary searches of branches of the tree that fall outside of the ball. This can drastically improve performance, especially in higher dimensions.

```
NN(v, p, r):
  input: p=current candidate, r=ball radius
  output: new candidate(p, r)
  precondition:Ball(q, r) overlaps R(v)
  if v is leaf:
    p':=nearest neighbor of P(v) to q
    r':= ||p'-q||
    if r'<r:return (p', r')
  if qi<=m:
    p, r = NN(vl, p, r)
    if Ball(q, r) overlaps R(vr):
      p, r =NN(vr, p, r)
  else:
    p, r = NN(vr, p, r)
    if Ball(q, r) overlaps R(vl):
      p, r =NN(vl, p, r)
```



All leaves the NN algorithm had to visit are shown in white!

Texture syntesis

Given a texture I, i.e. an image where we can zoom in a particular window and obtain the same image we want to obtain a **bigger texture** T

We can define an input pixel p_i and an output pixel p_o , $N(p)$ is the neighborhood around p

```
init a random border around T
for all p0 in T in scan order:
  find pi in I such that d(N(pi),N(po)) is minumum
  po=pi
```

To compute $d(N(pi), N(po))$ we can use an euclidean metric, and to find the p_i in the third line we can use a KD-tree over the vector of pixels of $N(p_i)$

Vector of pixels: $\begin{bmatrix} r_1 & g_1 & b_1 & \dots & r_n & g_n & b_n \end{bmatrix}^T$

Some observations:

- all the pixels in T are deterministically determined by the random border
- The quality of T depends on the size of neighborhoods of the points

    - Possible solution: image pyramid
- The scan order and the shape of the neighborhoods should match

algorithm with image pyramid(imaging painting)

```
construct image pyramid over I=I0 bottom up
define a neighborhood N(p)
for each level l:
  build a kd tree
  for all the neighborhoods of all points in I^l:
    for all the layers l top down:
      build T^l
        for all the pixels p0 in T^l:
          find the nearest neighbor of N(p) in the kdtree for the level l
```

Stackless kdtrees ray traversal

Given a scene split in a kdtree and a ray, how to compute cleverly which leaves are traversed by the ray?

Given the directions $D = \{ \text{left}, \text{right}, \text{top}, \text{bottom}, \text{front}, \text{back}\}$

Took a direction $d \in D$, $\bar{d}$ is its opposite

Let v be a node in the kd-tree

We define a rope $v \xrightarrow{d} w$ where w is a node in the direction d with the neighbor of v in direction d if there is only one neighbor or the last common parent of the neighbors of v if there are several.

ANSATZ for construction the ropes:

replace $v \xrightarrow{d} w$ by ropes to the children of w if possible.

```
def pushDownRope(v,w,d)
  if the splitting plane of w is perpendicular to d:
    if d in {right, top, back}:
      return w1
    else:
      return w2
  else if the splitting plane of w is parallel to d:
    if the side of R(w) is in direction opposite to d:
      return w1
    else: return w2
  return w
```

```
def propagateRows(v):
  if v is leaf:
    return v
  for all d in D:
    if a rope from v to w in d exists
      w' =w
      do until w''=w':
        w''= pushdownrope(v,w', d)
      replace the rope in v with the rope from v to w' in d
  d=right, top or front if the splitting axe of v is x, y or x
  v1/v2=left/right child of v
  copy the existing ropes of v in v1/v2
  set rhe ropes from v1 to v2 in d and its opposite
  propagateRopesDown(v1)(v2)
def followRay():
  p0 = start point of the ray
  v = root, p=p0
  while v is not null:
    while v is an inner node:
      if p is left of the split plane:
        v=v1
      else:
        v=v2
    p'=closest interception point of the ray wiith the triangles in v
    if the interception exists and is inside R(v):
      return p'
    p=exit point of the ray out of v
    d=direction of the exit
    if exists a rope fron v to w in d:
      v=w
    else:
      v=null
   return null
```

Cool material

http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/kd.html

https://www.cse.iitd.ac.in/~ssen/cs852/scribe/RangeQueries.pdf

https://users.cs.utah.edu/~lifeifei/cs6931/kdtree.pdf