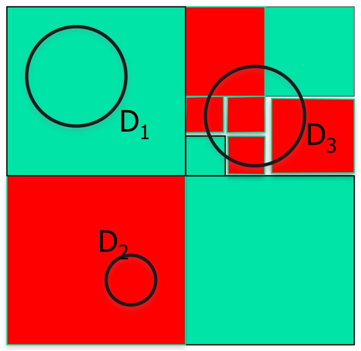
Accelerating Data Structures

QuadTrees



Assume we are given a red/green picture defined a $2^h \times 2^h$ grid. E.g. pixels. Each pixel is either **green** or **red**.

(more general and interesting examples – soon)

Need to represent the shape “compactly”

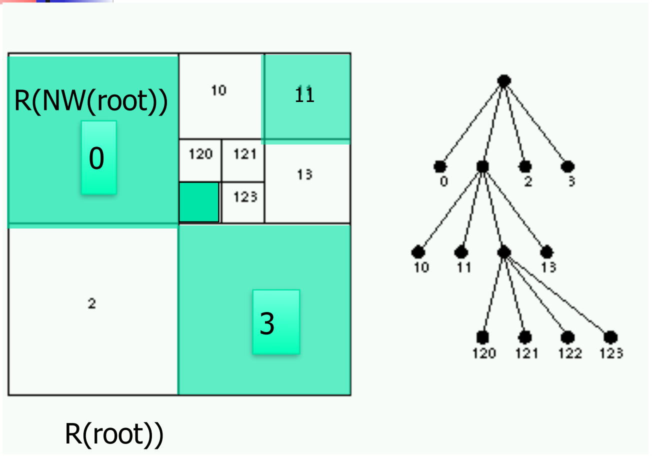
Need a data structure that could answers multiple types of queries. For example:

1. For a given point q, is q red or green ?

2. For a given query disk D, are there any green points in D ?

3. How many green points are there in D ?
4. Etc etc

Regions of nodes



A tree where each internal node has 4 children.

In general, every node v is associated with a region of the plane. Lets denote this region by R(v).

R(root) is the whole region of interest (e.g. input image or USA)

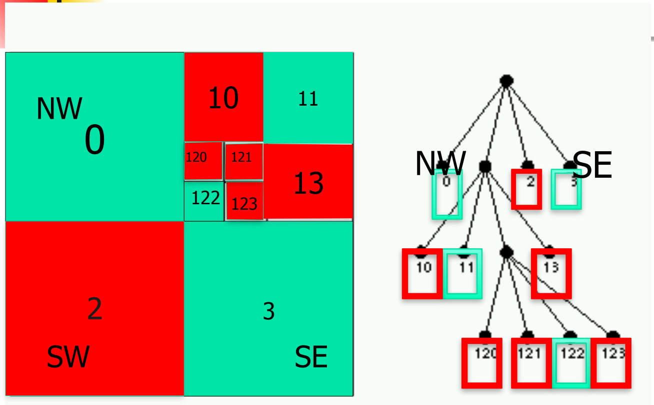
The smallest possible area of R(v) is a single **pixel**.

For every non-root node v, we have $R(v) \subset R(parent(v))$

R(v) = is the union of
R(NW(v)), R(NE(v)) R(SW(v)), R(SE(v))

Let NW(v) denote the North West child of v.
(similarly NE, SW, SE)

QuadTrees

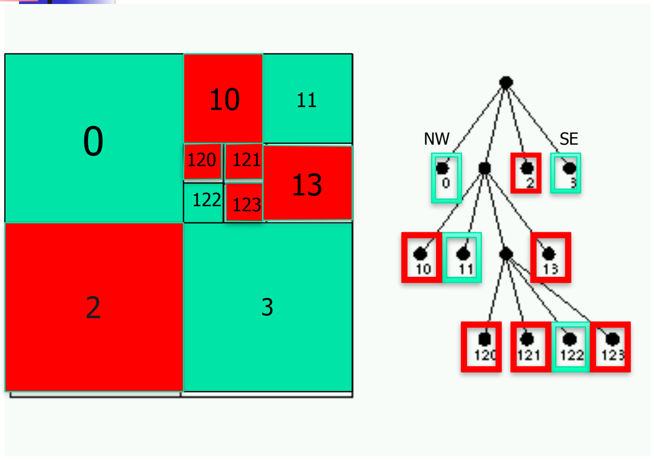


- Assume we are given a red/green picture defined on a $2^h \times 2^h$ grid of **pixels**.
- Each pixel has as a unique color (Green or Red)
- Every node $v \in T$ is **associated with a geometric region** $R(v)$

Alg constructQT for a shape S.
•**input** – a node $v \in T$, and a shape *S.*
•**Output** – a Quadtree T_v representing the shape of *S within R(v)*).

• If *S* is fully green in *R(v),* or S is fully red in *R(v)* – then
• *v* is a leaf, labeled Green or Red. Return ;
•Otherwise, divide *R(v)* into 4 equal-sized quadrants, corresponding to nodes v.*NW,* v.*NE,* v.*SW,* v.*SE*.
• Call constructQT recursively for each quadrant.

QuadTrees



Consider a picture stored on an $2^h \times 2^h$ grid. Each pixel is either red or green.

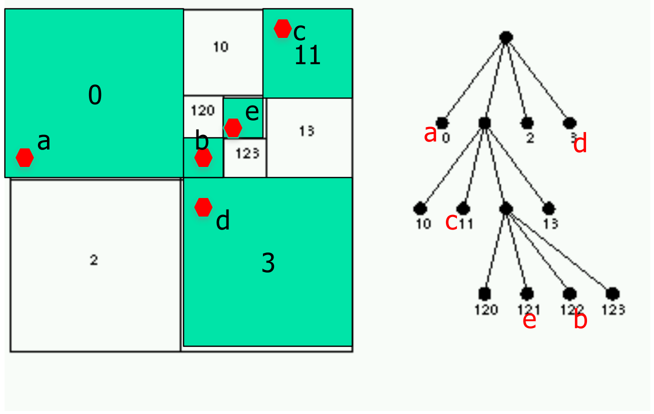
We can represent the shape “compactly” using a QT.

Height – at most h.
Point location operation – given a point q, is it black or white
– takes time O(h)
- could it be much smaller ?

Many other operations are very simple to implement.

QuadTree for a set of points

given: a set of points $S = \{a, b, c, d, e\}$, each with its (x,y) coordinates



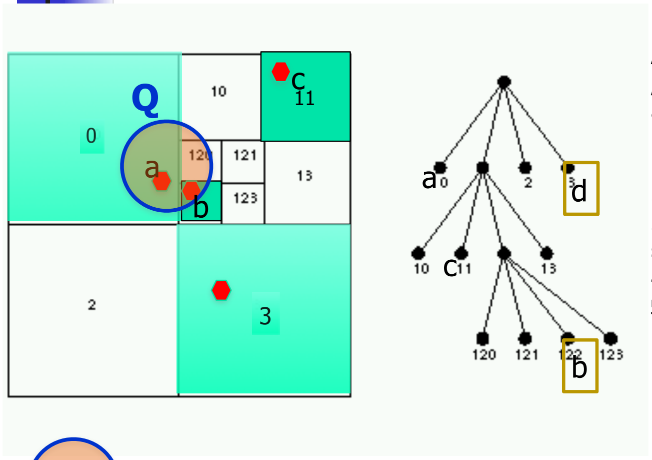
Now consider a set of points (red) but on a $2^h \times 2^h$ grid.

Splitting policy: Split until each quadrant contains ≤1 point.

Build a similar QT, but we stop splitting a quadrant when it contain ≤1 point (or some other small constant)
Point location operation – given a point q, is it black or white
– takes time O(h) (in practice, usually much less)

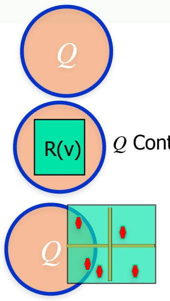
Many other **splitting polices** are very simple to implement.
(eg. A leaf could contain contains ≤17 points)

QuadTrees for a set of points



```
Report(Q,v)
// Q – a query disk
/* report all the points in stored at the subtree rooted at v, which are contained inside Q. */

1.If v is NULL – return.
2.If R(v) is disjoint from Q –return NULL.
3.If R(v) is fully contained in Q – report all points in the subtree rooted at v.
4.If v is a leaf – check each point in R(v) if inside Q
5.Else  //$R(v)$ Partially overlaps $Q$
    Report(Q, NW(v)) and
    Report(Q, NE(v))  and
    Report(Q, SW(v)) and
    Report(Q, SE(v))
```



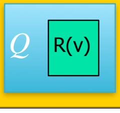
Q disjoint from $R(v)$; $Q \bigcap R(v) = \varnothing$

Q Contains $R(v)$;

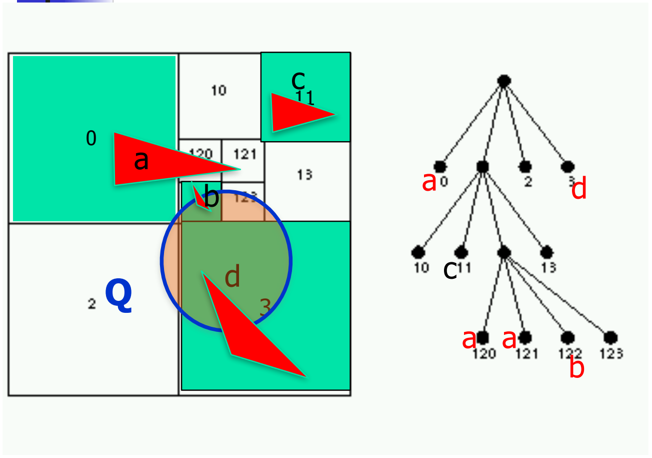
$R(v)$ Partially overlaps Q

Comment: In practice, it is much easier to work with query region which is an axis-parallel rectangle (why?). We use disks in the slides for visualization.

For example, to check if $R(v) \subseteq Q$, it is enough to check MinX, MinY, MaxX,MaxY



QuadTrees for shape



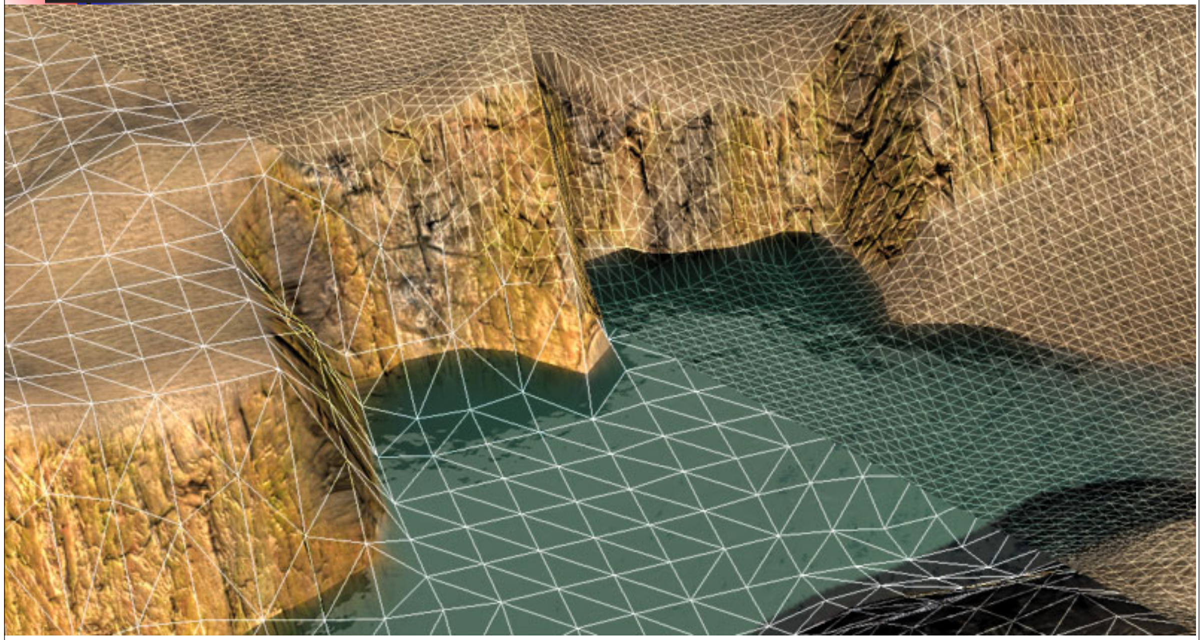
Input: Set S of triangles
S={$t_{1 \ldots} t_n$ }

Splitting policy: Split quadrant if it intersects more than 1 triangle of S.

Note – a triangle might be stored in multiple leaves.
Some leaves might store no triangles.

Finding all triangles inside a query region Q – essentially same Report Report(Q,v) as before
(minor modifications)

Triangulated terrain
(TIN – Triangulated irregular network

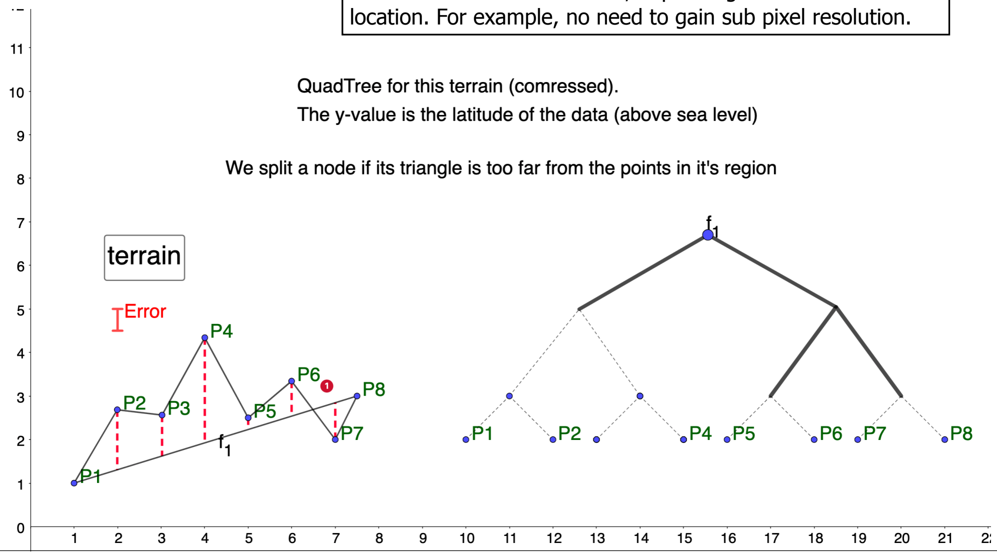


Each triangle approximately fits the surface below it

Other Splitting policies

- Example: 1-dimensional quad tree. Define on the data points {A,B…H}. Each point has $x-$value (location) and $y - value$ - elevation. If a set of points could be clustered, then we don't need to store them explicitly. Instead we could interpolate between them using a triangle (or in 1D, a segment)
- Splitting policy: Dont split if data in R(v) could be linearly interpolated with a error $\leq$ threshold.
- For simplicity, the triangle is show

Possibly we could keep all the internal nodes, and decide which one to use in real time, depending on the viewer' location. For example, no need to gain sub pixel resolution.

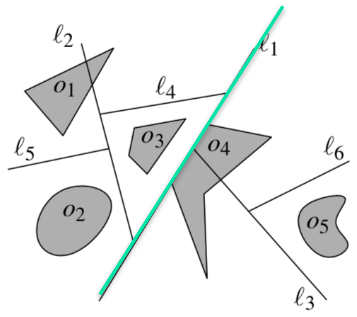


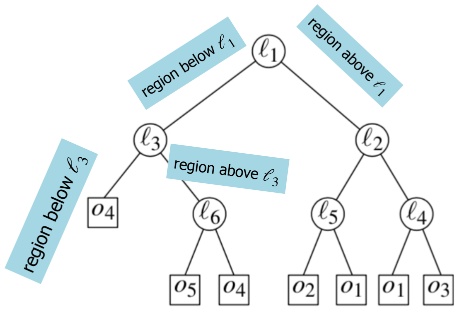
QuadTree for this terrain (comresssed).
The y-value is the latitude of the data (above sea level)

We split a node if its triangle is too far from the points in it's region

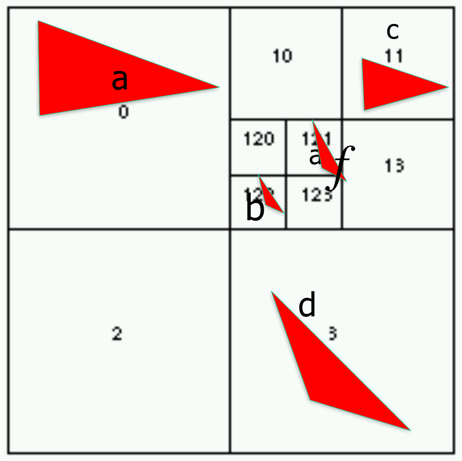
BSP (Binary Space Partition) Trees

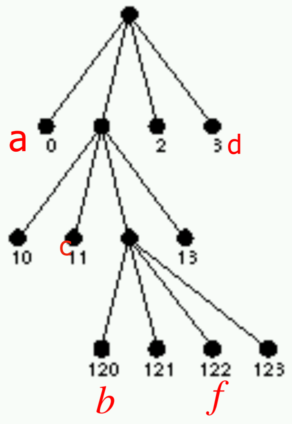
- Partition a set of objects using a set of arbitrary half-spaces (for clarity, we only show segments, and not the full line separating objects).
- Each internal node contains a halfplane (in 3D) or a line (in 2D).
- For each half-space, divide the objects into two groups, those above and those below the half-space boundary
- Store the resulting divisions in a binary tree
  - Example BSP of 5 objects in the plane
  - The same object might Possibly objects might be ``split". They are not physically split, but an object might be stored more than once in the tree (e.g. O4)
  - Confusing warning: Each internal node stores a line/plane. On the slides, we draw it 'chopped' to the region it is relevant to (e.g. ℓ_3 is not relevant to the objects in which are below ℓ_1, so we render it chopped at its intersection with ℓ_1)





QuadTrees/OctTree for shape - also a BSP



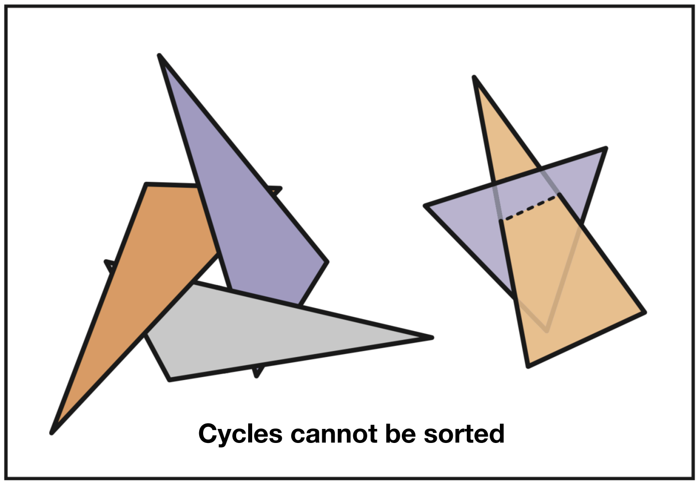


Input: Set S of triangles $S=\{t_{1\dots}t_n\}$

Splitting policy: Split quadrant if it intersects more than 1 triangle of S.

Painter's algorithm

- **Amounts to a topological sort of the graph of occlusions**
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort

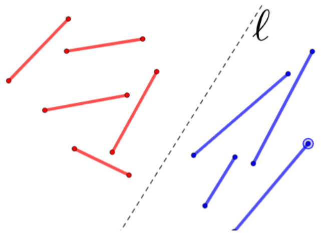


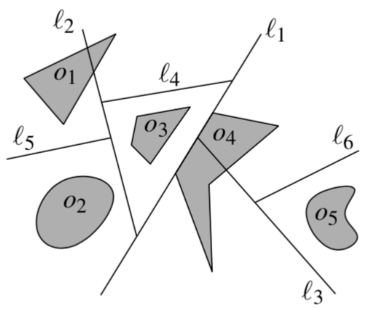
Painter Algorithm and BSP

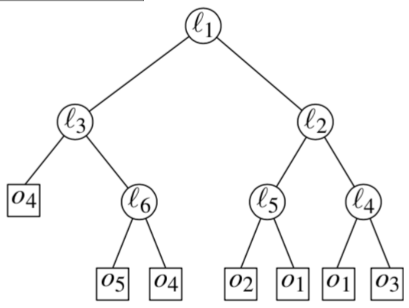
- Assume all objects (eg triangles $\in \mathbb{R}^3$, segments in $\mathbb{R}^2$) are stored in a BSP T, and none is split. Each object in a leaf in a leaf.
- Given a viewer's location, we can always render them (successfully) using the **painter algorithm**: This is how:
- The line/plane at the root *root(T)* partition the plane/space into two regions.

  Let's call the one containing the viewer v_{near}, and call the other v_{far}.

  1. Render the objects in the subtree of v_{far} (recursively)
  2. Render the objects in the subtree of v_{near} (recursively)
- **Claim**:
- No object in v_{far} occludes from the viewer an object in v_{near}.





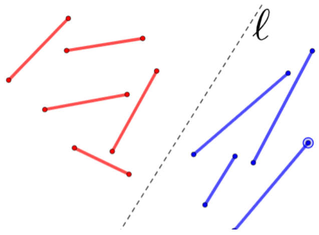


BSP are expressly useful also with Z-buffer

- Assume all objects (eg triangles $\in \mathbb{R}^3$, segments in $\mathbb{R}^2$) are stored in a BSP T. Each in a leaf. Then, for any viewer's location, we can always render them (successfully) using the painter algorithm: This is how:
- The line/plane at the root *root(T)* partition the plane/space into two regions.

  Let's call the one containing the viewer v_{near}, and call the other v_{far}.

  1. No need to visit nodes of the tree which are not visible to the viewer (outside its frustum). So this s a filtering part.
  2. Objects in v_{far} Render could be rendered less frequently (if the viewer moves) - recycle the same background. And/Or prepare billboard for theses objects - they are the background
  3. Render the objects in the subtree of v_{near} (recursively)



If something sounds too good to be true… it probably is.
To build the BSP we might need to split objects.

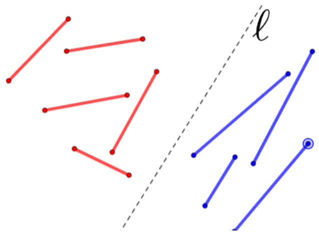
Puzzle: Find 3 segments in $\mathbb{R}^2$ where we could not build BSP without splitting segments (please come to the whiteboard to draw it).

Painter Algorithm and BSP

- Assume all objects (eg triangles $\in \mathbb{R}^3$, segments in $\mathbb{R}^2$) are stored in a BSP T. Each in a leaf. Then, for any viewer's location, we can always render them (successfully) using the painter algorithm: This is how:
- The line/plane at the root *root(T)* partition the plane/space into two regions.

  Let's call the one containing the viewer v_{near}, and call the other v_{far}.

  1. Render the objects in the subtree of v_{far} (recursively)
  2. Render the objects in the subtree of v_{near} (recursively)
- **Claim**:
- No object in v_{far} occludes from the view an object in v_{near}.



If something sounds too good to be true… it probably is.
To build the BSP we might need to split objects.

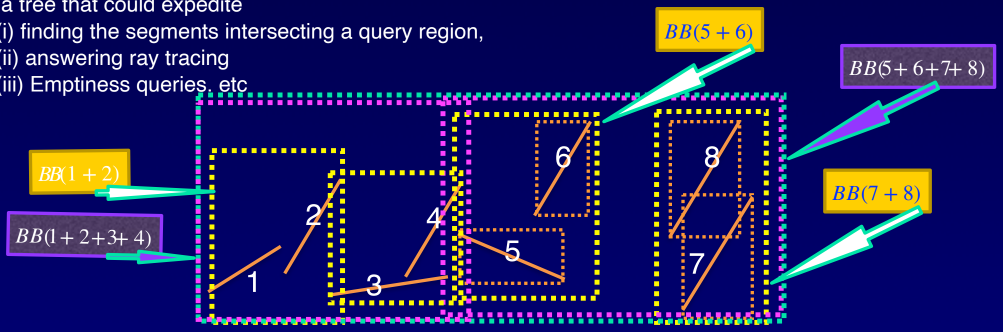
Puzzle: Find 3 segments in $\mathbb{R}^2$ where we could not build BSP without splitting segments (please come to the whiteboard to draw it).

In practice, with Z-buffer

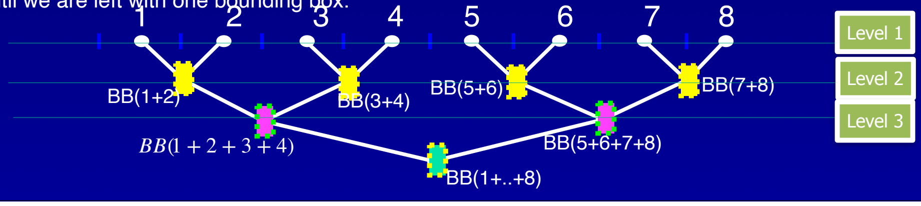
- Still use BSP

R-trees - described in 2d, but work in any dim

- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries, etc

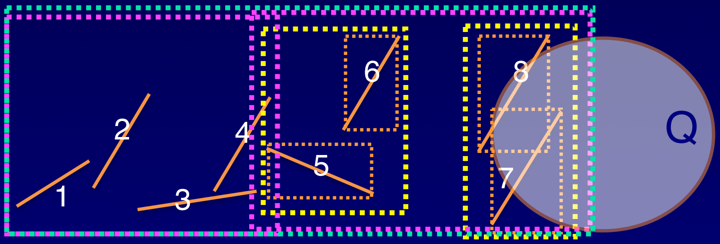


- We compute for each segment its bounding box (rectangle).
- These are the leaves of *T*. Call them ``Level 1”.
- Find the nearest pair of segments (say 7,8). Remove them from level 1, and replace them by a single BB encapsulate bot
  It corresponds to a node of level 2.
- Repeat until no vertex is left in level 1.
- Next, pick the nearest two BBs from level 2, and replace them by a vertex at level 3.
- In general, each internal node *v* in level **j** is created by merging two children nodes of level **j-1**.
  - $BB(v) = BB\left(BB(v\,.\,right) \bigcup BB(v\,.\,left)\right)$
- Repeat until we are left with one bounding box.



R-trees

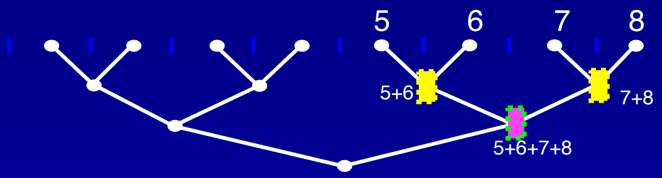
- Input: A set S of shapes (segments in this example. Triangles in graphics apps)
- Build a tree that could expedite
  - (i) finding the segments intersecting a query region,
  - (ii) answering ray tracing
  - (iii) Emptiness queries. etc



Report(Q, root(T))

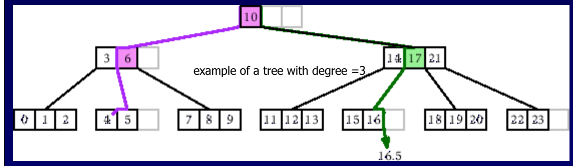
// Once a query region Q is given, we need to report the segments inside Q. (or segments intersecting Q, or their numbers etc)

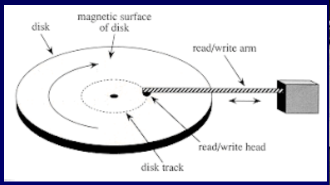
1. Check if Q intersects BB(root), and if $BB(root) \subseteq Q$
2. If Q does not intersect BB(root), we are done. Return
3. If $BB(root) \subseteq Q$, then all segments in the tree are inside Q. (sometimes it is sufficient just to find their number, which could be stored in root.)
4. If root is a leaf, check if the (unique) segment it stores is $\subseteq Q$.
5. Otherwise, check recursively: Call report(Q, root.right) and report(Q, root.left)

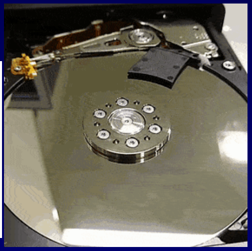


R-trees in practice. Memory Hierarchy, and advantages of multiple children
Large degree helps

- In practice, it is sometimes preferable to create trees with a very large degrees (instead of binary trees).
- For example, each internal node, will have between 100 to 500 children







- Consider a very simplistic model of the computer memory - fast main memory, and slow secondary memory. (your computer follows this model, probably with more than 2 types of memory, and probably SSD instead of disks, but this model still applies.
- Only small portion of the tree could be stored in the main memory.
- Consider point location operation (find the segment containing a query point)
- We start the search by visiting the root, then one of its children, one of its grand-children … until we reach a leaf.
- The seek-time in disks, and even in SSD, is much slower than the seek-time for main memory. Therefor, once the head of the disks is located in the correct place, we usually read a bucket - about 4KByte of memory.
- The bottleneck of the *search/insert/delete* operation is the number of seek operations (number of I/Os).
- The number of seek-operation is proportional to height of the tree.
- Say $n = 10^9$. The height of a tree of degree 2 with n leaves is $\log_2(10^9) \approx 30$, so 30 seek operations are needed.
- If each node contains about 1000 segments, or keys, then the height (and number of I/Os) is only $\log_{1000}(10^9) = 3$
- The root and possibly its children are always in main memory, so this number of only 1 or 2.
- R-trees are the most popular and important data structures for very large spatial data.
- If the stored items are 1-dimensional (rather than multi-dim), then B-trees are used instead of R-trees. They are very convent for insertion/deletion and other operations.

More trees

kD-trees

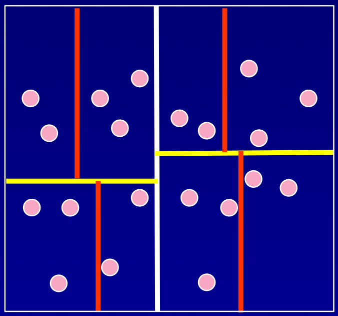
range trees

(in textbook - Chapter 5)

Range Searching

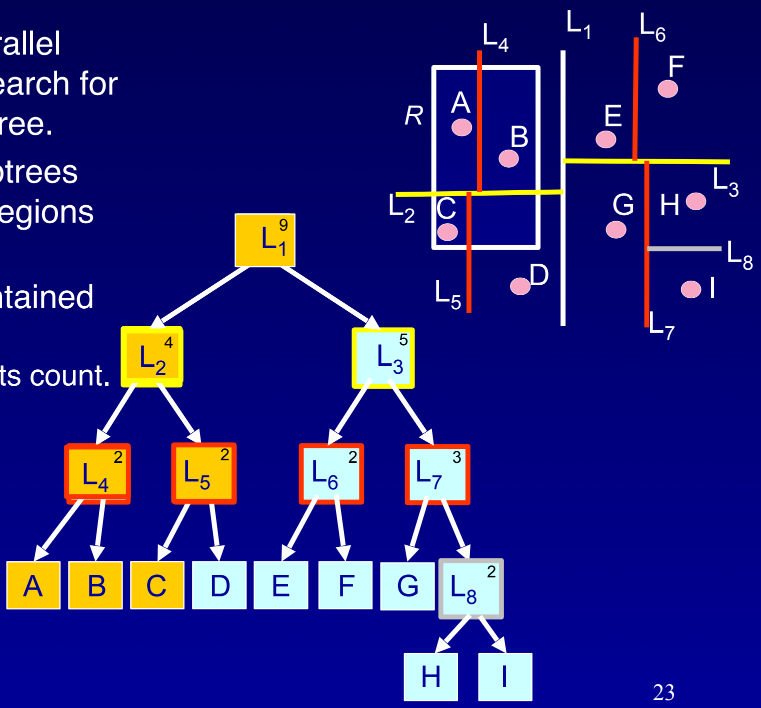
2D-Trees (and in higher dimension, kD-trees)

- Given a set of points in 2D.
- Bound the points by a rectangle.
- Split the points into two (almost) equal size groups, using a horizontal line, or vertical line. (first horizontal, then vertical, back to horizontal etc)
- (in $\mathbb{R}^3$, split by a plane orthogonal to the
  - $x-$axis,
  - then orthogonal to $y-$axis,
  - then $z-$axis,
  - and back to $x - axis$ etc
- Continue recursively to partition the subsets, until they are small enough.



Range Counting/Reporting

- Given an axis-parallel range query *R*, search for this range in the tree.
- Traverse only subtrees which represent regions intersecting *R*.
- If a subtree is contained entirely in *R*:
  - Counting: Add its count.
  - Reporting: Report entire subtree.



Runtime Complexity

- *k* nodes are reported. How much time is spent on internal nodes? The nodes visited are those that are **stabbed** by *R* but not contained in *R*. How many such nodes are there ?
- **Theorem**: Every side of *R* stabs $O(\sqrt{n})$ cells of the tree.
- **Proof**: Extend the side to a full line (WLOG - horizontal):
  - In the first level it stabs two children.
  - In the next level it stabs (only) two of the four grandchildren.
  - Thus, the recursive equation is:
- Total query time: $O(\sqrt{n})$ for counting,
- $O(k+\sqrt{n})$ for reporting

$$Q(n) = \begin{cases} 1 & n=1 \\ 2+2Q\left(\frac{n}{4}\right) & otherwise \end{cases}$$
$$= O\left(\sqrt{n}\right)$$

- Lets try to understand where this $\sqrt{n}$ appeared
- Write $n = 2^h$ (where h-height(T).
- If we start a search in the root of the tree, and go down, for every two levels that we are diving, the number of nodes only increased by a factor of 2. Hence we could only reach $2^{h/2} = \sqrt{n}$ nodes

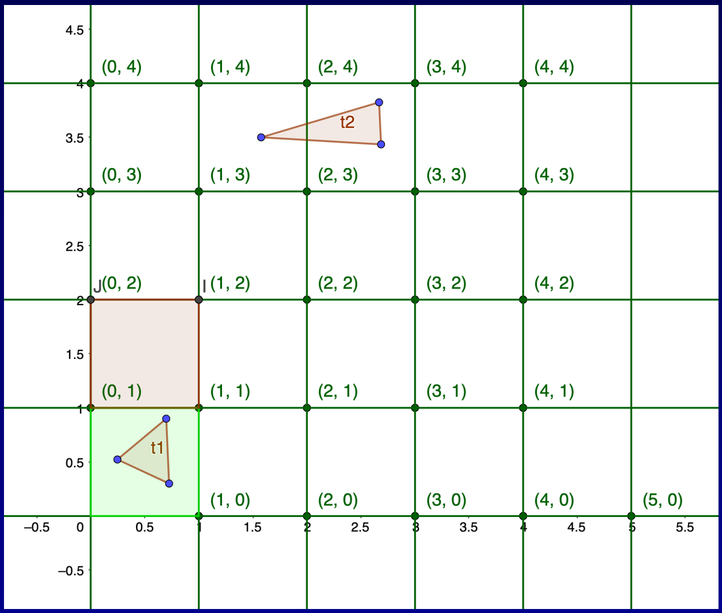
Kd-Trees – Higher Dimensions

- For a *d*-dimensional space:
  - Construction time: O(*d n*log*n*).
  - Space Complexity: O(*dn*).
  - Query time complexity: O($dn^{1-1/d}+k$).

Question: Are kd trees useful for non-orthogonal range queries, e.g. disks, convex polygons ?

Fact: Using *interval trees*, orthogonal range queries may be solved in O($\log^{d-1}n+k$) time and O($n\log^{d-1}n$) space.

Cell Decomposition and Hashing



(0, 4)
(1, 4)
(2, 4)
(3, 4)
(4, 4)
t2
(0, 3)
(1, 3)
(2, 3)
(3, 3)
(4, 3)
J
(0, 2)
I
(1, 2)
(2, 2)
(3, 2)
(4, 2)
(0, 1)
(1, 1)
(2, 1)
(3, 1)
(4, 1)
t1
(1, 0)
(2, 0)
(3, 0)
(4, 0)
(5, 0)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5