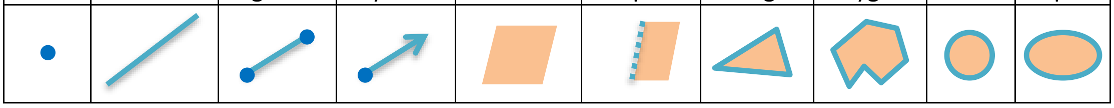
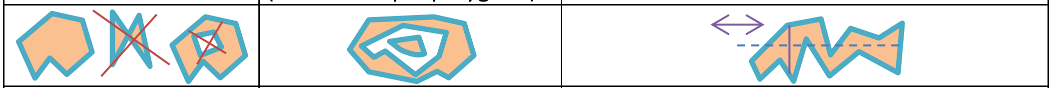
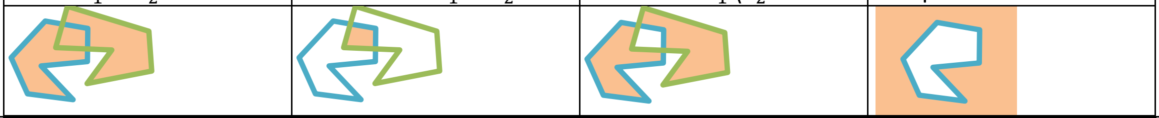
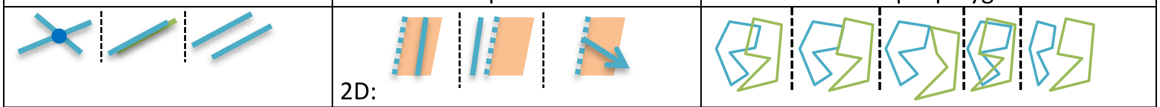
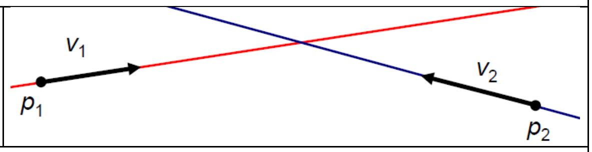
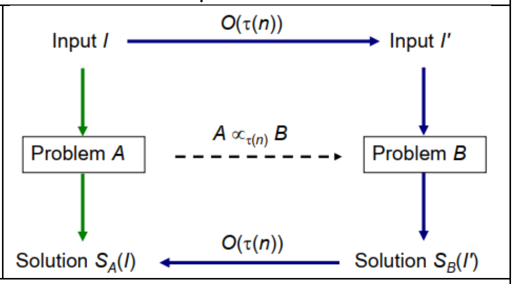
ALGORITHMS

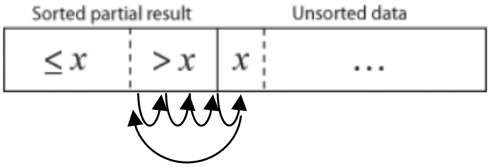
1. Introduction to Computational Geometry

Geometry Primitives	Point · Line · Segment · Ray · Plane · Halfplane · Triangle · Polygon · Circle · Ellipse  **sets**: unordered, ordered
Polygon types	simple Polygon (SP) · general polygon (=set of simple polygons) · monotone polygons  simple Polygon: without intersection, without whole general polygon: wholes allowed, polygon in wholes allowed monotone polygons: any line perpendicular (rechtwinklig) to line intersects with boundary 0,1 or 2 times
Boolean operations	union $P_1 \cup P_2$ · intersection $P_1 \cap P_2$ · difference $P_1 \backslash P_2$ · complement $\neg P$ 
Intersection examples	intersect two lines · intersect half plane with line · intersect two simple polygons  $L_1 \cap L_2 = \{Point, Line, \emptyset\}$ $H \cap L = \begin{cases} 3D: Point, Line, \emptyset, Ray \\ 2D: Line, \emptyset, Ray \end{cases}$ $SP_1 \cap SP_2 = \{SP, Segment, Point, [SP], \emptyset\}$
Vector operations	see Skalarprodukt Vektorprodukt
Line intersection	$Line_1: \binom{x_1}{y_1} = p_1 + s * \vec{v_1}$ $Line_2: \binom{x_2}{y_2} = p_2 + t * \vec{v_2}$ $Intersection: Line_1 = Line_2$ 
main problem precision model	Main problem is that when we store the intersection point in a double, and determine later if the intersection point lies on the line, it gives $\emptyset$ **solution 1**: use data type rational instead of double -> needs more computation time because rational is stored with two big integers **solution 2**: draw circle around point which symbolise the point -> creates new problems
Problem and Approach	**Geometrical problems** - computational (compute all line intersections of a line set) - decision (is a given point inside a polygon) **Standard Approach** transform problem and input to geometrical equivalent, choose construction paradigm, choose data structure, choose complexity analysis technique, solve problem geometrical, transform solution to original problem domain 
Linear Searching	**task**: searching in a list **given**: List L of numbers, $\|L\| = n$ **ask**: is a given $x \in L$? **worst case**: $T_{wc}(n) = \max T(I)$ $over\ all\ instances\ I\ of\ size\ n$ $x \in L: T_{wc} = O(n), \quad x \notin L: T_{wc} = O(n)$ **average case**: $T_{avg}(n) = \sum_I P[I] * T[I]$ $over\ all\ instances\ I\ of\ size\ n$ $x \in L: P[pos] = \frac{1}{n}$ $T_{avg(n)} = \sum_{i=1}^{n} P[i] * T(to\ find\ x\ at\ pos\ i) = \sum_{i=1}^{n} \frac{1}{n} * O(i) = \sum_{i=1}^{n} \frac{i}{n} = \frac{n(n+1)}{2 * n} = \frac{n+1}{2} \approx \frac{n}{2}$
Complexity Theory	-> see Big-O-Notation
library	JavaGeom (Java, not supported anymore) JTS Topology Suite (Java) CGAL (C++): most professional LEDA (C++): little bit old

sort algorithms

Insertion sort $O(n^2)$

go from left to right throug the array and move each element 'x' as far left as it has to be




```
void insertSort(T[] a) {
   for (int i=1; i < a.length; i++) {
      int x = a[i];
      int j = i - 1;
      // shift previous values to the right
      while(j ≥ 0 && a[j] > x) {
         a[j + 1] = a[j];
         j--;
      }
      a[j + 1] = x; // insert on the left
   }
}
```

Mergesort $O(n \log n)$

```
// l=left, r=right (common in C++)
void mergeSort(T[] a,int l,int r){
   if (l < r) { // n>1
      // divide into two equal parts
      int m = l + (r - l)/2;
      // sort the two parts
      mergeSort(a, l, m);
      mergeSort(a, m + 1, r);
      // merge them two into one
      merge(a, l, m, r);
   }
}
```

```
void merge(T[] a, int l, int m, int r){
   T[] b; int i = l, j = m + 1, k = l;
   while (i <= m && j <= r) { // both have element
      if (a[i] <= a[j]) { b[k] = a[i]; i++;}
      else { b[k] = a[j]; j++; }
      k++;
   }
   if (i > m) { // add rest from right part
      for (int h=j; h <= r; h++) b[k+h-j] = a[h];
   } else { // add rest from left part
      for (int h=i; h <= m; h++) b[k+h-i] = a[h];
   }
   for (int h=l; h <= r; h++) a[h] = b[h];
}
```

Quicksort $O(n \log n)$

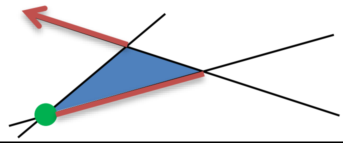
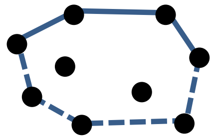
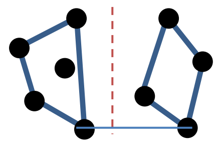
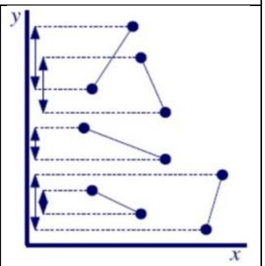
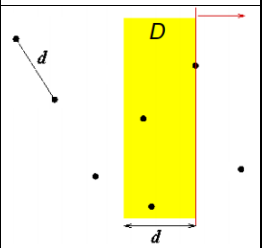
```
void quicksort(T[] a) {
   sort(a, 0, a.length - 1);
}
```

```
void sort(T[] a, int l, int r) {
   int i = l, j = r;
   T p = a[l]; // pivot element
   do {
      while(a[i] < p) i++; // from left
      while(p < a[j]) j--; // from right
      if (i ≤ j) { // exchange
         T tmp = a[i]; a[i] = a[j]; a[j] = tmp;
         i++; j--;
      }
   } while(i < j);
   if (j > l) sort(a, l, j); // smaller than pivot
   if (i < r) sort(a, i, r); // larger than pivot
}
```

Exercise 2

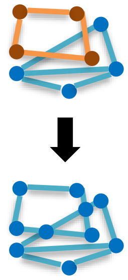
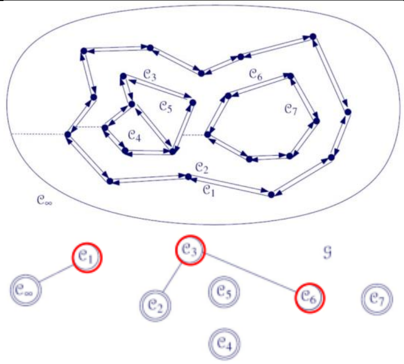
$$T(1) = c_1$$
$$T(n) = 2 * T\left(\frac{n}{2}\right) + n$$
$$T(n) = 2^i * T\left(\frac{n}{2^i}\right) + i * n, i \in \mathbb{N}$$
$$2^i = n \rightarrow i = \log_2 n$$
$$T(n) = n * T\left(\frac{n}{n}\right) + \log_2(n) * n$$
$$T(n) = n * c_1 + n \log_2(n)$$
$$T(n) = n(\log_2 n + c_1)$$

2.+3. Construction Paradigms

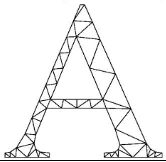
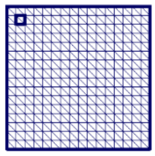
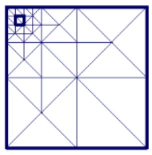
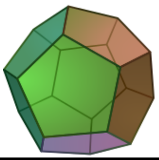
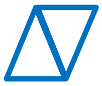
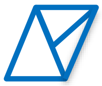
<table>
<tr><td>Constraints</td><td>Are like halfplanes</td></tr>
<tr><td>Incremental construction</td><td>a geometric structure is incrementally constructed, after each step a valid geometric structure is available.
<table>
<tr><td>Line arrangements</td><td>Convex Hull (CH)</td></tr>
<tr><td>Input: n lines in 2D</td><td>Input: n Points in 2D</td></tr>
<tr><td>Output: Arrangement = the lines induce a subdivision of the plane that consists of vertices, edges and faces</td><td>Output: Clockwise ordered list of points that are the vertices.</td></tr>
<tr><td></td><td></td></tr>
<tr><td>n_V: vertex = where two lines cross
n_E: edge = a segment or ray on a line
n_F: face = plane between lines</td><td>finite set P with n points</td></tr>
<tr><td>Insert a new line $I_i \rightarrow O(i)$
Total complexity $O(n^2)$</td><td>compute the convex hull $\rightarrow O(n \log n)$</td></tr>
</table></td></tr>
<tr><td>Convex hull algorithm 1</td><td>1. Sort all points by x -> $O(n \log n)$
2. Compute upper hull from left to right: For all points: -> $O(n)$
while the last three points makes a "right turn" remove the second last point $O(n)$
3. Compute lower hull from right to left in same way</td></tr>
<tr><td>Graham Scan (for convex hull)</td><td>1. Find the point P with the lowest y-coordinate -> $O(1)$
2. Sort the points in increasing order of the angle they and P make with x-axis -> $O(n \log n)$
3. For all points: -> $O(n)$
while the last three points form a "right turn" remove the second last point $O(n)$</td></tr>
<tr><td>Divide and Conquer
(for convex hull)</td><td>1. Divide: points into 2 subsets
2. Conquer: find convex hull for each subset
3. Merge: with upper and lower tangent
$$T(n) \leq 2 * T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$</td></tr>
<tr><td>Plane Sweep</td><td>move a line from one side to the other and handle points</td></tr>
<tr><td>Line Segment Intersection (LSI)</td><td>3 types of events (points): 'start', 'intersection' and 'end' event
as soon as two line segments become neighbours, check for intersection point (I)
two datastructures: event queue Q (with m event points) and
Status T (binary search tree)
$$O((n+I)\ \log n)$$ $$I \in O(n) \rightarrow O(n \log n)$$ $$I \in O(n^2) \rightarrow O(n^2 \log n)$$
</td></tr>
<tr><td>Use case: DEM (digital elevation model) -> determine the horizon</td><td>1. compute triangulation (Delaunay)
2. backface removal
3. cylindrical projection
4. computing the horizon (divide and conquer)</td></tr>
<tr><td>Closest Pair</td><td>Given a set S of n points in the plane, find a pair of closest neighbors.
naive approach: $O(n^2)$
plane sweep paradigm or divide-and-conquer $O(n \log n)$
1. lexicographically sorting points S $\rightarrow O(n \log n)$
2. empty ordered set D: $O(1)$
3. event handling (each points is added to D and removed from D once) $2 * \log n$
4. query $D(p) \rightarrow O(\log n)$
5. compute the distance and update closest pair $\rightarrow O(1)$
</td></tr>
<tr><td>Voronoi Diagram</td><td>will come later</td></tr>
<tr><td>All-Nearest-Neighbors</td><td>given a set S of n points in the plane, find a nearest neighbor of each
-> compute voronoi diagram in $O(n \log n)$ and extract solution in $O(n)$
-> or use plane sweep paradigm to compute directly in $O(n \log n)$</td></tr>
</table>

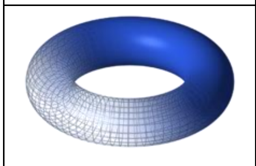
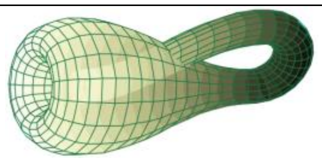
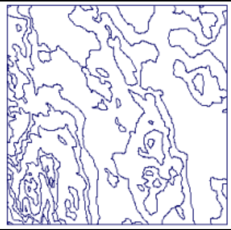
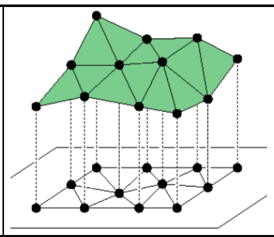
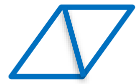
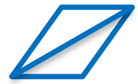
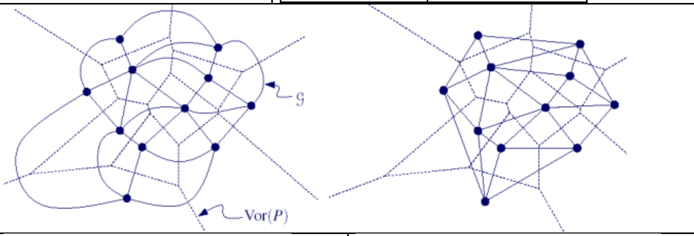
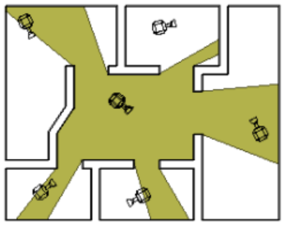
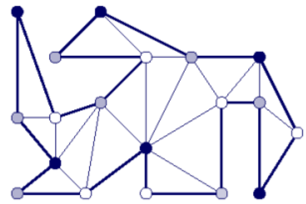
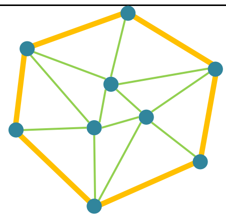
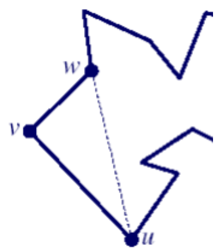
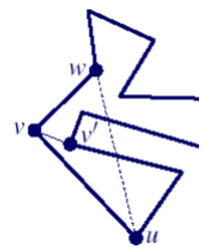
4. Planar Subdivisions

see Graph Theory

Overlay of Subdivision in DCEL	**Phase 1 (Vertices and Edges)** 1. copy existing two subdivisions S1 and S2 to a new subdivision D (not a proper DCEL) 2. run a plane sweep algorithm and transform D to a correct DCEL for O(S1, S2) (D is changed at intersection event points) **Phase 2 (Faces)** 3. create a face record for each face f in O(S1, S2) 4. set OuterComponent(f) to a half-edge on the outer boundary of f 5. create a list InnerComponents(f) to half-edges on the boundaries of the holes inside f 6. set IncidentFace() for each half-edge on the boundary of f 7. label f with the names of the faces in S1 and S2 that contain it $O(n \log n + k * \log n)$	
Boundary Cycles of the same Face	1. Create Graph G 2. a node represents one boundary cycle 2. draw an arc between two cycles if one of the cycles is the boundary of a hole and the other cycle has a half-edge immediately to the left of the leftmost vertex of the hole cycle	
Use Case: Boolean Operations	1. Compute Overlay 2. iterate through all faces and filter them depending of the Boolean operation 3. Create polygons from boundary cylces	

5. Polygon Triangulations

Types of subdivisions of a plane in triangles	**Triangulation** (no additional points) 	**Mesh** (add additional points)	
	2D of a Planar Point Set P: set of n points in the plane (not all collinear) k Points on boundary = 6 n Points totally = 9 m number of triangles = 10 $n_v - n_e + n_v = 2$ $n - \frac{3m + k}{2} + (m + 1) = 2$ $2n - 3m - k + 2m + 2 = 4$ $m = 2n - k - 2$ $n_e = 3n - 3 - k$	**uniform**  all edges looks the same	**non-uniform**  fine near the edges coarse far away from edges
	3D Triangulation of Convex Polytope P: set of n points in 3D (not all collinear) $O(n \log n)$ number of facets is at most: $6n - 20$ 	**conforming**  **well-shaped** all angles between 45° and 90°	**non-conforming**  **respect the input** edges of the component must be contained in the union of mesh

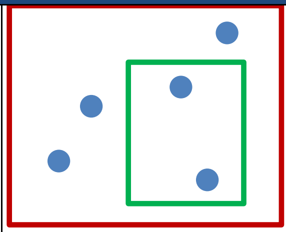
orientable vs non-orientable	**orientable**:  torus **non-orientable**:  klein bottle  möbius strip
2.5 Dimension & Triangulation	**2.5D**: 2D Surface in a 3D space. Each vertical line intersects it in exactly one or zero point. e.g. Terrain  
optimal Triangulation	Small skinny triangles are bad, because height interpolation is more error-prone maximization of minimum angle of a triangulation  good  bad
Delaunay Triangulation	The Delaunay triangulation is the dual of the voronoi diagram. It does not contain illegal edges. Can be computed in $O(n \log n)$. 
Art Gallery Problem	**Problem**: How many (360°) cameras do we need to guard a given gallery and how do we decide where to place them? **Complexity**: NP-hard! if convex → $O(1)$ **Upper Bounds**: on every edge -> n cameras in every triangle of triangulation: n-2 cams on every black vertex: $\left\lfloor \frac{n}{3} \right\rfloor$ cameras   3-Coloring of a Triangulation
Triangulation	A decomposition of a polygon P into triangles by a maximal set of non-intersecting diagonals (line segments between pairs of vertices)
Example P: set of n points k: points on convex hull	1. calculate convex hull 2. point-to-sth polygon  $n = 9$ $k = 6$ $m = 10$ $n_e = 18$ $\rightarrow m = 2n - k - 2$ $10 = 2 * 9 - 6 - 2$ $\rightarrow n_e = 3n - 3 - k$ $18 = 3 * 9 - 3 - 6$ $\rightarrow n_e = \frac{3m + k}{2}$ $n_e = \frac{3 * 10 + 6}{2} = 18$
Triangulating a simple polygon	1. Find a diagonal in P → $O(n)$ - let v be the leftmost vertex P - let u and w be the neighbors of v - try to connect u with w - if this fails we connect v to the vertex farthest from uw inside the triangle defined by u,v and w 2. Triangulate the two resulting subpolygons recursively $O(n)$ → $O(n^2)$  
Better approch	A simple polygon with n vertices can be triangulatedinto y-monotone polygons in $O(n \log n)$ time with sweep-line algorithm that uses $O(n)$ storage, and therefore **triangulated in $O(n \log n)$** time.

6. Orthogonal Range Searching

Searching and Indexing

given: $set\ S = \{o_i | i = 1..n\}$
find: $subset\ R = \{o_i | P(o_i), i = 1..n\}$

assumption: S is incrementally updated
approach use a space partition tree

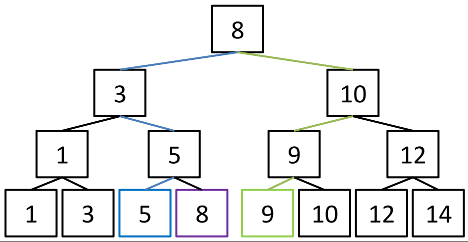


Binary Tree

searching in a tree: $O(\log n)$

One dim. range query/searching

Get elements between 4 and 9
1. search first element $O(\log n)$
2. search second element $O(\log n)$
3. get elements between $O(k)$
total $O(k + \log n)$

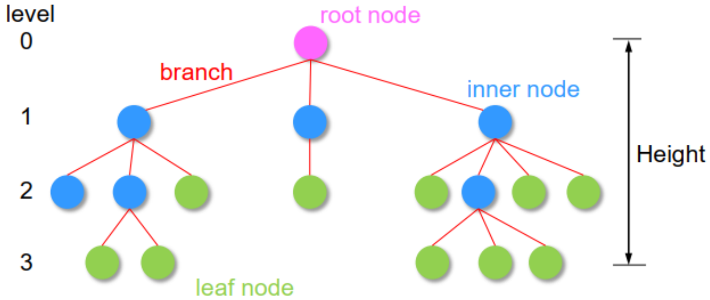


MSE

Mean Squared Error (at RGB example)

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (r_i - r_i')^2 + (g_i - g_i')^2 + (b_i - b_i')^2$$

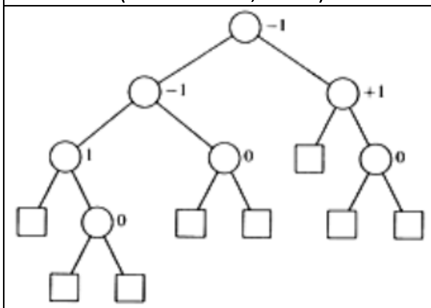
Tree (General)



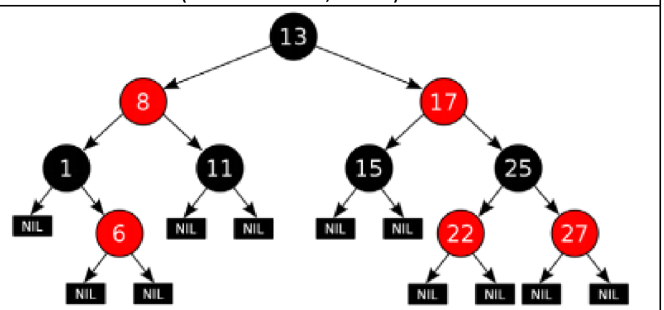
Binary Tree: each node has 0 (leaf) or 1,2 (inner node) following nodes
Search Tree: all nodes are sorted from left (lowest) to the right (highest)
Balanced Tree: each node has similar number of following nodes
-> height as small as possible

Types of balanced binary search tree's

AVL-Tree (used in C++, 1962)



Red-Black-Tree (used in JAVA, 1972)



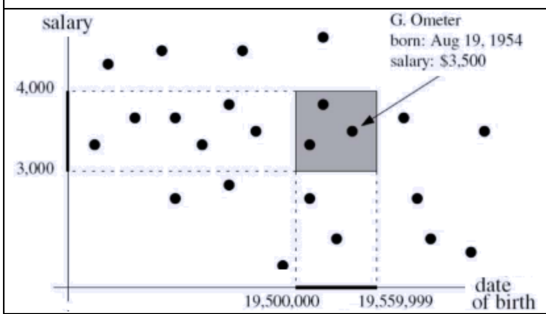
Operations

insertion/deletion: needs rebalancing (tree rotations) afterwards

Application

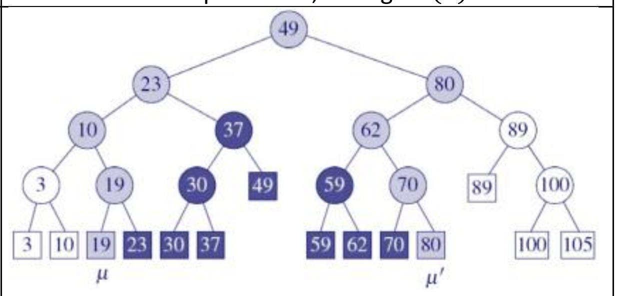
Querying a Database

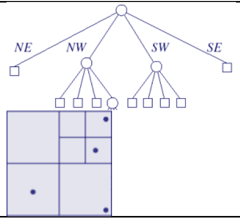
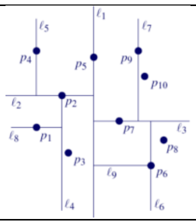
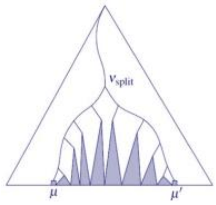
Who has a salary between 3000 and 4000
and ist born in 1954.

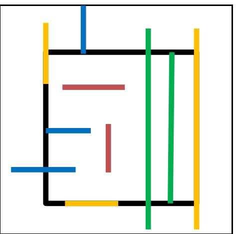
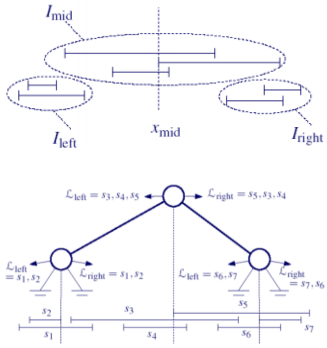
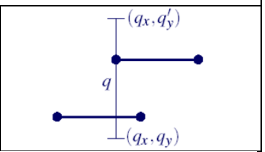
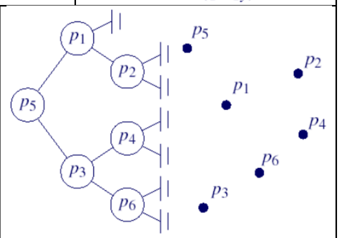


1-D Range Searching

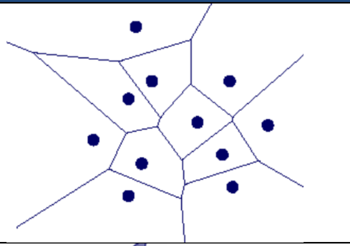
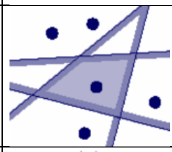
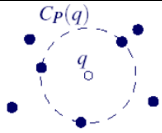
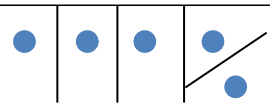
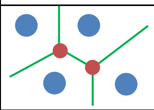
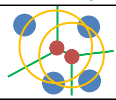
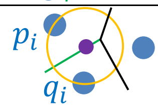
Find all items with keys in interval $[18:77]$
construction $O(n \log n)$, query $O(k + \log n)$ where
k = number of output nodes, storage $O(n)$



Range Search	Quadtree / Octree	Kd-Tree	Range Tree	Layered Range Tree
dimension	$d = 2\,/\,3 \text{ or more}$	$d \geq 2$	$d \geq 2$	$d \geq 2$
storage	$O\big((h+1) * n\big)$	$O(d * n)$	$O(n * \log^{d-1} n)$	$O(n * \log^{d-1} n)$
build time	$O\big((h+1) * n\big)$	$O(d * n * \log n)$	$O(n * \log^{d-1} n)$	$O(n * \log^{d-1} n)$
query time		$O\left(k + n^{1-\frac{1}{d}}\right)$	$O(k + \log^{d} n)$	$O(k * \log^{d-1} n)$
height	$\log\frac{s}{c} + \frac{3}{2}$ $c\text{: smallest dist}$ $s\text{: length square}$			
# nodes balanced	$O\big((h+1) * n\big)$ $O(m)$			
# leaves	$3 * \text{inner nodes} + 1$			
				
usage	triangulation, non-uniform mesh generator, simulation finite element method	nearest neighbor $O(\log n)$, Image Compression, k-means clustering, filter algorithm		

Windowing in 2D and 3D	**Problem**: reporting all objects fully contained in, or intersecting, a given window. similar to range queries, but data are objects and search space is normally 2D or 3D. **Application**: GIS: report all map objects intersecting a given window VR: report all triangles intersecting the viewing volume	
simpler problem	**Problem:** Windowing of axis-parallel line segments 4 different cases: segments lying entirely in window segments intersect the boundary once segments intersect the boundary twice segment (partially) overlap the boundary segments with at least one endpoint inside window -> use range query segments with both endpoints outside window -> use an interval tree	
Interval tree	**Problem:** report all horizontal line segments that intersect the left edge (or vertical the bottom edge)	
construction	**Input:** a set I of n closed intervals $[x_i : x_i']$ **Preprocessing**: Sorting interval endpoints -> simplify median computation **Divide-and-Conquer**: - compute the median of I completely to the left of x_{mid} - build 3 subsets $(I_{left}, I_{right}, I_{mid})$ - create node v and store I_{mid} with v - create recursively interval tree with I_{left} and store root as left child of v - create recursively interval tree with I_{right} and store root as right child of v **2 Sorted Lists** L_{left}: contains all intervals of I_{mid} sorted on increasing left endpoints L_{right}: contains all intervals of I_{mid} sorted on decreasing right endpoints	
Analysis	storage $O(n)$, depth $O(\log n)$, construction $O(n \log n)$, query $O(k + \log n)$	
Extension	Replace two associated range tree T_{left} and T_{right} reporting all segments whose left endpoint lies in $(-\infty : q_x] \times [q_y : q_y']$ reporting all segments whose right endpoint lies in $[q_x : \infty) \times [q_y : q_y']$ storage $O(n \log n)$, construction $O(n \log n)$, intersection report $O(k + \log^2 n)$	
Priority Search Tree	storing two associated range trees per node in an interval tree is overkill, because the performed range queries are unbounded on one side	
Idea	replace range trees by two priority search trees (special x-y-ordered heaps)	
construction	1. search for the most left (min x) 2. split by median of y 3. repeat	
Analysis	storage $O(n)$, built $O(n \log n)$, query $O(k + \log n)$	

7. Voronoi Diagrams

Voronoi Diagram	Model where every point is assigned to the nearest site. **given**: set of distinct points in the plane: $P = \{p_1, p_2, \dots, p_n\}$ **search**: Voronoi diagram $Vor(P)$ **solution**: sweep line algorithm $O(n \log n)$	
single Voronoi cell	the bisector of two points p and q is the perpendicular bisector of the line seqment pq. this bisector splits the plane into two half-planes $h(p,q)$ containing p and $h(q,p)$ containing q	
Structure of the Voronoi diagram	let $C_p(q)$ be the largest empty circle with q as its center that does not contain any site of P in its interior.	
	if all sites are collinear, then $Vor(P)$ consists of $n-1$ parallel lines otherwise $Vor(P)$ is connected and its edges are either segments or half-lines (rays)	
	for $n \geq 3$: the number of vertices in $Vor(P)$ is at most $2n-5$ the number of edges is at most $3n-6$	 $2*4-5=3$ $3*4-6=6$
	At point q is a vertex of $Vor(P)$ if and only if its largest empty circle $C_p(q)$ contains three or more sites on its boundary	
	the bisector between sites p_i and p_j defines an edge of $Vor(P)$ if and only if there is a point q on the bisector such that $C_P(q)$ contains both p_i and p_j on its boundary but no other site.	
	Quad-Edge struct is suitable to store voronoi diagram <-> delaunay-triangulation	

	1. Range Tree (Endpoints) 2. Internal Tree (x-direction) 3. Internal Tree (y-direction)	

8 Heuristics

-> see O-Notation
-> see Complexity Theory
-> see Graph Theory
-> see MetaHeuristics

Optimization Problem	**Minimize** $f(s)$, subject to $s \in S$ Where f is the objective function, s the solution and S the set of all feasible solutions	
Brodal Queue	decrese $O(1)$ find min $O(1)$ delete min $O(\log n)$	