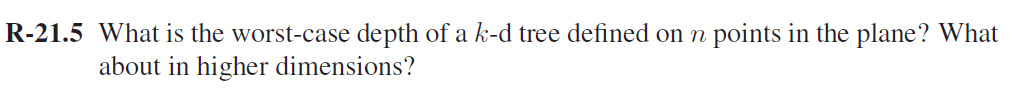
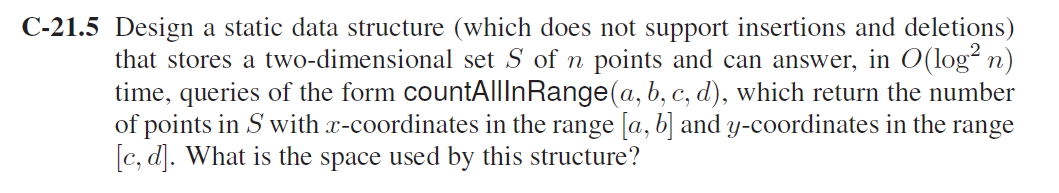
Homework 11

Atiq Patel CWID:10432883



**Solution:** A k-d tree consists of binary search trees. The maximum depth a binary search tree can have is log n where n is the number of the elements in the plane. Thus the worst case depth k-d tree can have is O(logn) in the plane where n are the number of points. For higher dimensions the scalability increases, for example if it were a 3d plane. The worst case would have been O (log3n) and for 4d it would have been O (log4 n) and so on…

In general for d number of dimensions the worst case depth would be O (logd n) which is **O (d log n)** where n are the number of points in the plane.



**Solution:** This can be implemented by using a slight modification in two-dimensional range trees:

The primary structure would be based on the Two dimensional Binary Search Tree which keeps x coordinates and y coordinates separately.

Also when consider 1DTreeRangeSearch as the same Range Query Algorithm 3.11 from the book when mentioned in the below pseudocode.

**Algorithm countAllInRange(a*, b, c, d, v, T*)**:

*Input:* Search keys *a, b, c and d.* node *v* in the primary structure *T* of a two-dimensional range tree; type *t* of node *v*

*Output:* The number of points in S with x-coordinates in the range [a,b] and y coordinates in the range [c,d].

if *T.*isExternal(*v*) then

return *∅*

if *a* *≤ element\_xcordinate*(*v*) *≤ b*  then

if *c* *≤ element\_ycordinate*(*v*) *≤ d*  then

*M ← {*element(*v*)*}*

else

*M ← ∅*

if *t* = “left” then

*L ←* countAllInRange(*a, b, c, d, T.*leftChild(*v*)*,* “left”)

*R ←* 1DTreeRangeSearch(*c, d, T.*rightChild(*v*))

else if *t* = “right” then

*L ←* 1DTreeRangeSearch(*c, d, T.*leftChild(*v*))

*R ←* countAllInRange(*a, b, c, d, T.*rightChild(*v*)*,* “right”)

else

// *t* = “middle”

*L ←* countAllInRange(*a, b, c, d, T.*leftChild(*v*)*,* “left”)

*R ←* countAllInRange(*a, b, c, d, T.*rightChild(*v*)*,* “right”)

else

*M ← ∅*

if *x*(*v*) *< a* then

*L ← ∅*

*R ←* countAllInRange(*a, b, c, d, T.*rightChild(*v*)*, t*)

else

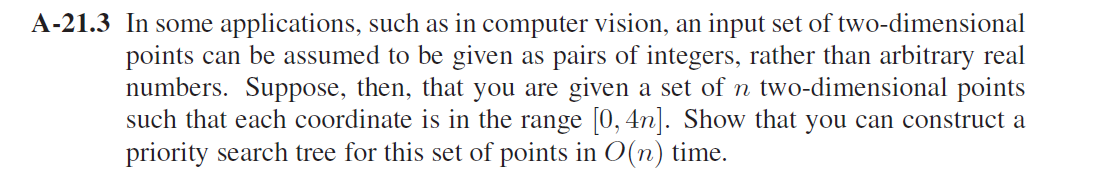
// *x*(*v*) *> b*

*L ←* countAllInRange(*a, b, c, d, T.*leftChild(*v*)*, t*)

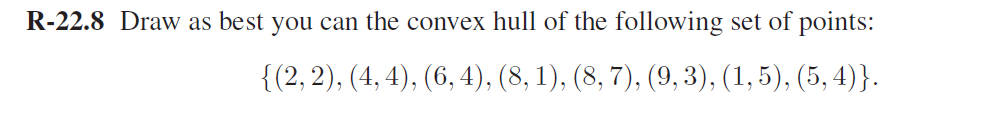
*R ← ∅*

return *L ∪M ∪ R*

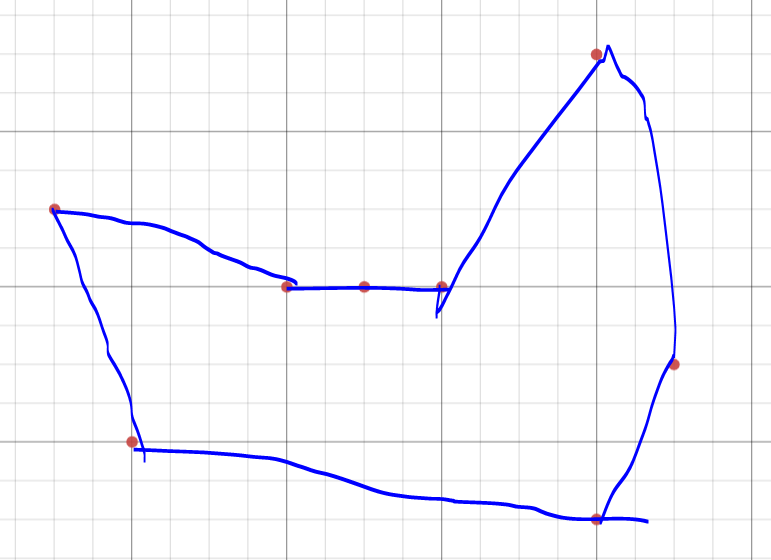
This would in turn give us **O(log2n)** being a 2Dimensional kd tree we have to make use of two binary search so it becomes like a nested binary search tree. The space used would be **O (n)** where n is0 the number of nodes.



**Solution:** For a PST the space requirement is O (n) follows the fact that every internal node of the priority search tree T stores a distinct item of S. The height of the tree follows from the halving number of nodes at each level. Here the nodes are in the range of 0 to 4n. So the worst case space requirement would be O (4n) which is basically O (n). If we use a sorting algorithm like Radix sort, our run time would be O (n). Thus, the whole priority search tree for these set of points would be constructed in **O (n)**.



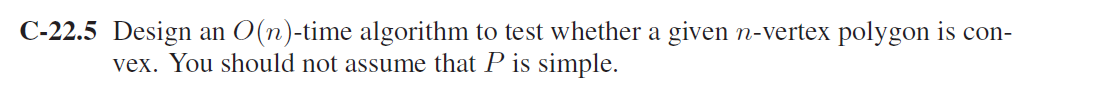
**Solution:** Plotting the diagram and connecting the dots.



Removing the non convex parts, we get the final diagram.

Final Diagram: The coordinates are (1, 5), (2, 2), (8, 1), (9, 3),(8, 7).





**Solution:**

**Algorithm CheckConvex(P):**

**Input:** An n-vertex polygon.

**Output:** The given polygon is convex or not.

We randomly pick a point and name it as Vertex A, consider b,c be the consecutive vertices of the vertex A in the polygon P. Vertex A would be the pivot point to check the angle between b and c. We check for the initial orientation.

OrientationCheck(a,b,c)

If angle(b)< angle(c) then

Orientation == CounterClockwise

If angle(b) = angle(c) then

Orientation == Colinear

If angle(b) > angle(c) then

Orientation == Clockwise

While (P.hasNextVertex())

**If** (Orientation(a,b,c) == Clockwise or Colinear) **do**

**Return** Polygon P is not conex.

**Else**

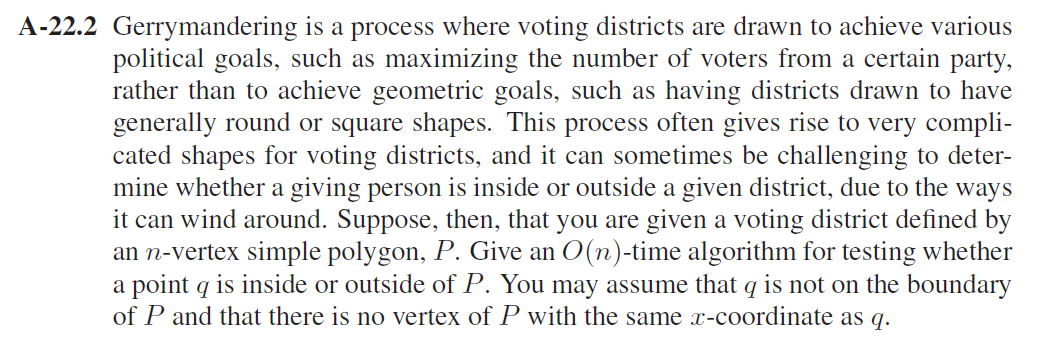
a=b

b=c

c= next vertex

**Return**  Polygon is convex

The algorithm does comparison in O (1) time but since the polygon has n vertices, the run time for this algorithm would **O(n)**.

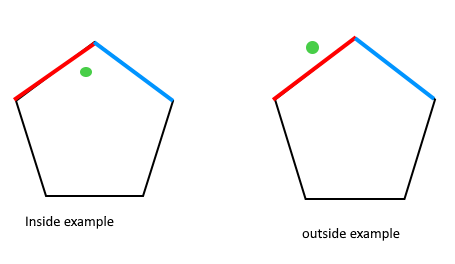


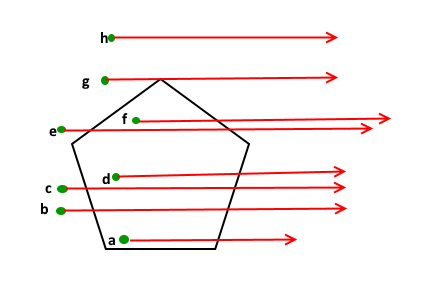
**Solution:** To check if the given point is inside the polygon P.

**1)** Frist you draw a horizontal line to the right of each point and extend it to infinity

**2)** Then you count the number of times the line intersects with polygon edges.

**3)** If the point is inside the polygon P either count of intersections is odd or point lies on an edge of polygon. If none of the conditions is true, then point must be outside of P.





**Algorithm PolygonCheck(P):**

**Input:** A polygon P with a point q with coordinates [x,y].

**Output:** To determine if the point q is inside the polygon P or not

J🡸points.length-1

result 🡸false

for i🡸0 to points.length& j🡸 points.length -1 **do**

**if** ((points[i].y > P.y) != (points[j].y > P.y) & (P.x < (points[j].x - points[i].x) \* (P.y - points[i].y) / (points[j].y-points[i].y) + points[i].x)) **do**

result =! Result

j🡸i+1

return result;

The run time of this algorithm is **O(n)** where n is the number of vertices.