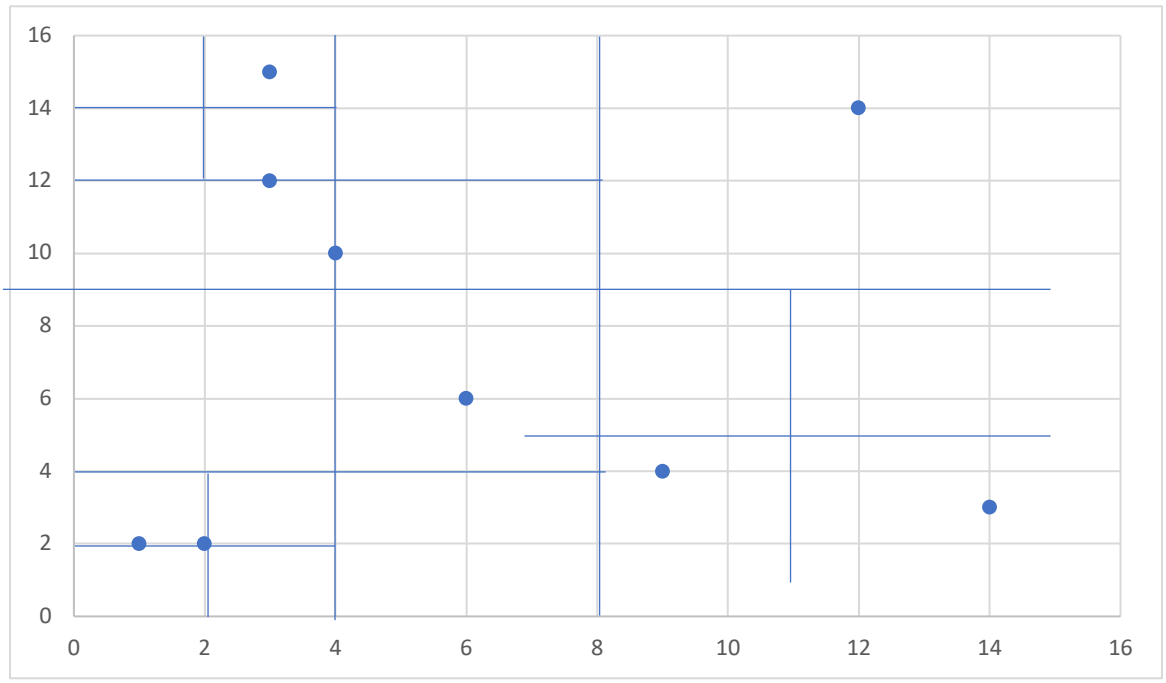
CS 600 HW 11
SHUCHI PARAGBHAI MEHTA
CWID: 20009083

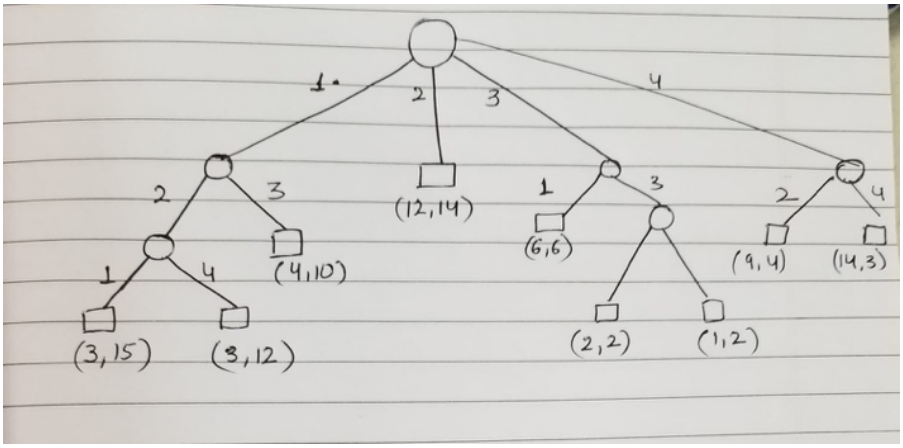
- 21.5.7

Draw a quadtree for the following set of points, assuming a 16×16 bounding box:

$\{(1,2),(4,10),(14,3),(6,6),(3,15),(2,2),(3,12),(9,4),(12,14)\}.$



Assuming that all of the intersection's points, whether in the north-east, north-west, south-east, or southwest, will appear in the first quadrant (South West),



- 21.5.13

Describe an efficient data structure for storing a set S of n items with ordered keys, so as to support a `rankRange`(a, b) method, which enumerates all the items with keys whose ***rank*** in S is in the range $[a, b]$, where a and b are integers in the interval $[0, n-1]$. Describe methods for object insertions and deletion, and characterize the running times for these and the `rankRange` method.

A balanced binary tree is a useful data structure for storing a set S of n items with ordered keys (AVL or Red-Black Trees).

The rankRange(a, b) method will function by counting all the items in the search tree that exist between x1 and x2 using in-order sorting and searching for the lower end of a range x1 and upper end of a range x2 where x1 and x2 satisfy ($x1 \leq x \leq x2$).

Insertion in AVL tree:
Algorithm insertAVL(k, e, T):
Input: A key-element pair, (k, e), and an AVL tree, T
Output: An update of T to now contain the item (k, e)

$v \leftarrow IterativeTreeSearch(k, T)$
if v is not an external node, then
return "An item with key k is already in T"

Expand v into an internal node with two external-node children

$v.key \leftarrow k$
$v.element \leftarrow e$
$v.height \leftarrow 1$
$rebalanceAVL(v, T)$

Deletion in AVL tree:
Algorithm removeAVL(k, T):
Input: A key, k, and an AVL tree, T
Output: An update of T to now have an item (k, e) removed

$v \leftarrow IterativeTreeSearch(k, T)$
if v is an external node, then
return "There is no item with key k in T"
if v has no external-node child then
Let u be the node in T with key nearest to k

Move u's key-value pair to v
$v \leftarrow u$
Let w be v's smallest-height child
Remove w and v from T, replacing v with w's sibling, z, and then rebalanceAVL(z, T)

Rebalance Tree:
Algorithm rebalanceAVL(v, T):
Input: A node, v, where an imbalance may have occurred in an AVL tree, T
Output: An update of T to now be balanced

$v.height \leftarrow 1 + max\{v.leftChild().height, v.rightChild().height\}$
while v is not the root of T do
$v \leftarrow v.parent()$
$if\ |v.leftChild().height - v.rightChild().height| > 1$ then
Let y be the tallest child of v and let x be the tallest child of y
$v \leftarrow restructure(x)$ // trinode restructure operation
$v.height \leftarrow 1 + max\{v.leftChild().height, v.rightChild().height\}$

Running Time:
The rankRange method requires $O(log\ n + k)$, where k is the total number of reported points in the range. Furthermore, both the insertion and delete methods will require $O(log\ n)$.

- 21.5.27

In computer graphics and computer gaming environments, a common heuristic is to approximate a complex two-dimensional object by a smallest enclosing rectangle whose sides are parallel to the coordinate axes, which is known as a ***bounding box***. Using this technique allows system designers to generalize range searching over points to more complex objects. So, suppose you are given a collection of n complex two-dimensional objects, together with a set of their bounding boxes,

$$\mathscr{S} = \{R_1, R_2, \ldots, R_n\}.$$

Suppose further that for some reason you have a data structure, D, that can store n four-dimensional points so as to answer four-dimensional range-search queries in $O(\log^3\ n + s)$ time, where s is the number of points in the query range. Explain how you can use D to answer two-dimensional range queries for the rectangles in $\mathscr{S}$, given a query rectangle, R, would return every bounding box, R_i, in $\mathscr{S}$, such that R_i is completely contained inside R. Your query should run in $O(\log^3\ n + s)$ time, where s is the number of bounding boxes that are output as being completely inside the query range, R.

Range Trees is a data structure that can be used to query two-dimensional data. We can store an array that is ordered by Y-coordinates instead of an auxiliary tree.
We'll conduct a binary search for y1 at x_{split}. We can use pointers to maintain track of the outcome of the binary search for y1 in each of the arrays along the path as we continue to look for x1 and x2.
A different name for this technique is fractional cascading search.

Running time of the algorithm for d dimension is $O(logd - 1\ n\ +\ s)$ and for 4 dimension it will be

$O(log^3 n\ +\ s)$.

- 22.6.7

Give a pseudocode description of the plane-sweep algorithm for finding a closest pair of points among a set of n points in the plane.

S will be a set of n points. It is possible to compute the following formula to get the shortest distance between two places P and Q:

dist(a, b) =

d = dist(a, b)

x(p) and y(p) stand for the x and y coordinates for any point p in S.
Think of a sweep line SL as the vertical line passing through point p of a sweep line S.

Input: Set S of n points in the plane
Output: Finding Closest pair (p, q) of points

Algorithm closestPair()

Let X be the structure in an array $A[1, \dots, n]$ containing set S points sorted by x-coordinates.

$\delta := dist(A[1], A[2])$ (Minimum distance among all points to the left of SL)
Let Y be the empty dictionary.

$while(point\ p\ <=\ n)$
$p\ =\ p + 1$

when new point is found
$if\ (dist\ (p, q)\ <\ \delta)$
$then$
$A[1] \leftarrow p,$
$A[2] \leftarrow q$
$d \leftarrow dist(p, q)$

Insert p into dictionary Y
Search closest point q to the left of p to points in Y.

For (all points whose y coordinates lie in [y(p) - δ, y(p) + δ])

find q point closest to p
$return\ (p, q)$

It will take $O(n\ log\ n)$ time to sort all the elements. An element can only be added or removed from the dictionary once in $O(log\ n)$ time. Each range query in S also requires $O(log\ n)$.

Therefore, the algorithm's overall running time is $O(n\ log\ n)$

- 22.6.16

Let C be a collection of n horizontal and vertical line segments. Describe an $O(n \log n)$-time algorithm for determining whether the segments in C form a simple polygon.

As given, we can use, a collection C of n horizontal and vertical line segments to form a simple polygon.

1. Determine all intersecting segment pairs with equal coordinates using a plane sweep.
2. Find the coordinates that have common horizontal and vertical points in the plane as straight-line SL moves from left to right.
3. If a common coordinate is found, it should be stored in a dictionary so that it can be checked for shared endpoints whenever a new line segment is discovered.
4. Using this form, we may return every coordinated value from the dictionary and determine whether or not they form a closed loop. It indicates that a polygon is real.

There are n points in collection C. Algorithm Plane Sweep will take $O(n \log n)$. Each line will be covered while moving for coordinates for the line segments will take $O(n)$.

Algorithm therefore completes in $O(n \log n)$ time.

- 22.6.30

In machine learning applications, we often have some kind of condition defined over a set, S, of n points, which we would like to characterize—that is, "learn"—using some simple rule. For instance, these points could correspond to biological attributes of n medical patients and the condition could be whether a given patient tests positive for a given disease or not, and we would like to learn the correlation between these attributes and this disease. Think of the points with positive tests as painted "red," and the tests with negative tests as painted "blue." Suppose that we have a simple two-factor set of attributes; hence, we can view each patient as a two-dimensional point in the plane, which is colored as either red or blue. An ideal characterization, from a machine learning perspective, is if we can separate the points of S by a line, L, such that all the points on one side of L are red and all the points on the other side of L are blue. So suppose you are given a set, S, of n red and blue points in the plane. Describe an efficient algorithm for determining whether there is a line, L, that separates the red and blue points in S. What is the running time of your algorithm?

We can utilize the convex hull property to find the line L that separates the blue and red points into two sets. constructing a convex hull around the individual blue and red dots.

Then verifying that neither convex hull crosses the other. A line that separates the red and blue points separately exists if the convex hull intersects it.

Utilizing the Graham Scan Algorithm to create a convex hull, it will take $O(n \log n)$.