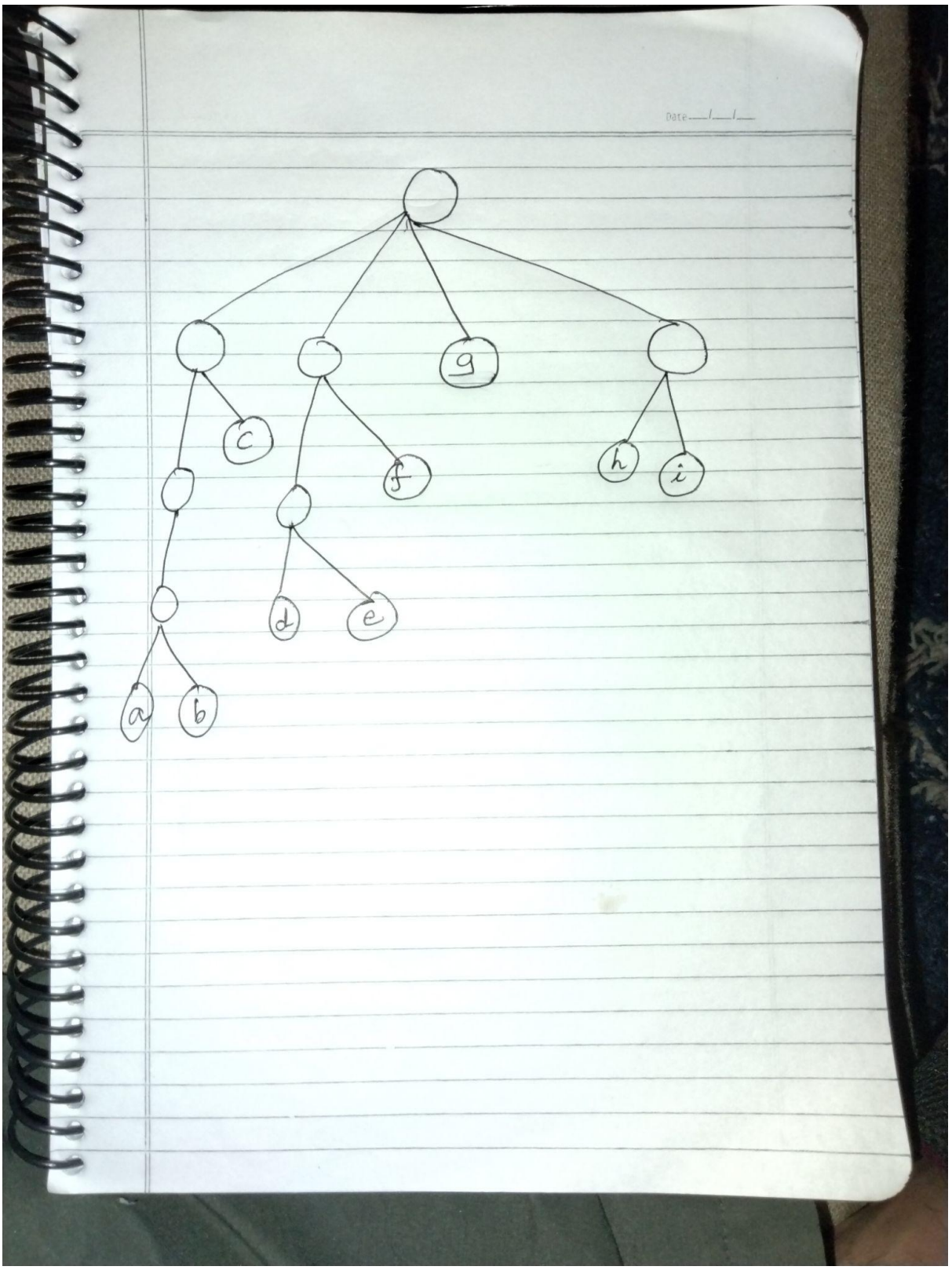
1.
Solution

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
01		a														
02		b														
03												d			f	
04										e						
05																
06						c										
07																
08																
09				h												
10																
11																
12														g		
13																
14			i													
15																
16																

And the quadtree will look like this



2.

Solution:

A balanced binary tree is an efficient data structure for storing a set S of n items with ordered keys (AVL or Red-Black Trees). The rank range(a, b) technique works by looking for the

lower end of a range x1 and the higher end of a range x2 where x1 and x2 fulfill (x1 <= x <= x2) and counting all the items in the search tree that exist between x1 and 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 ← 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 ← k

v.element ← e

v.height ← 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 ← 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←u

Let w be v’s smallest-height child

Remove w and v from T , replacing v with w’s sibling, z

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 ← 1 + max{v.leftChild().height, v.rightChild().height}

while v is not the root of T do

v ← 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 ← restructure(x) // trinode restructure operation

v.height ← 1 + max{v.leftChild().height, v.rightChild().height}

3.

Solution:

We can use the Range Trees data structure for querying two-dimensional data. We can store an array ordered by Y-coordinates instead of an auxiliary tree. We will do a binary search for y1 at the split. We can use pointers to maintain track of the result of the binary search for y1 in each of the arrays along the path while we continue to look for x1 and x2. This method is also known as fractional cascading search.

Running Time: Running time of the algorithm for the d dimension is $O(\log^{d-1} n + s)$ and for 4 dimensions it will be $O(\log^3 n + s)$.

4.

Solution:

Let S be a set of *n* points. For finding the minimum distance between two points P and Q it can be calculated as

dist(a, b) = $\sqrt{(x1 - x2)\ raised\ to\ 2 + (y1 - y2)\ raised\ to\ 2}$

Consider a sweep line SL is the vertical line through point *p* of *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,…,n] containing set S points sorted by x-coordinates.

δ *:= 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) < δ*)*

then

A[1] ← p,

A[2] ← q

d ← 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)

Element sorting will require O(n log n) time. Insertion and deletion of an element in the dictionary will require O(log n) time. And in S, each range query takes O. (log n). As a result, the algorithm's total running time is O. (n log n)

5.

Solution:

We may utilize a collection C of n horizontal and vertical line segments to build a basic polygon.

1. Using a plane sweep, find all overlapping pairs of segments having the same coordinates.

2. As the seep line SL moves from left to right, determine the locations in the plane that share similar horizontal and vertical points.

3. If a similar coordinate is identified, put it in the dictionary; every time we locate another line segment, we check the dictionary to see if they share endpoints.

4. Using this form, we can return all the coordinated items in the dictionary and see if they form a closed loop. It denotes the existence of a polygon.

The total number of points in collection C is n. The Plane Sweep algorithm will use O. (n log n). Moving for coordinates for the line segments will take O(n). As a result, the method runs in O(n log n) time.

6.

Solution:

We may utilize the convex hull property to find a line L that separates the blue and red points into two distinct sets. Separately forming a convex hull around blue and red spots. Then determine whether or not both convex hulls intersect. If the convex hulls intersect, there is a line that divides the red and blue points separately.

It will take O to create a convex hull using the Graham Scan Algorithm.