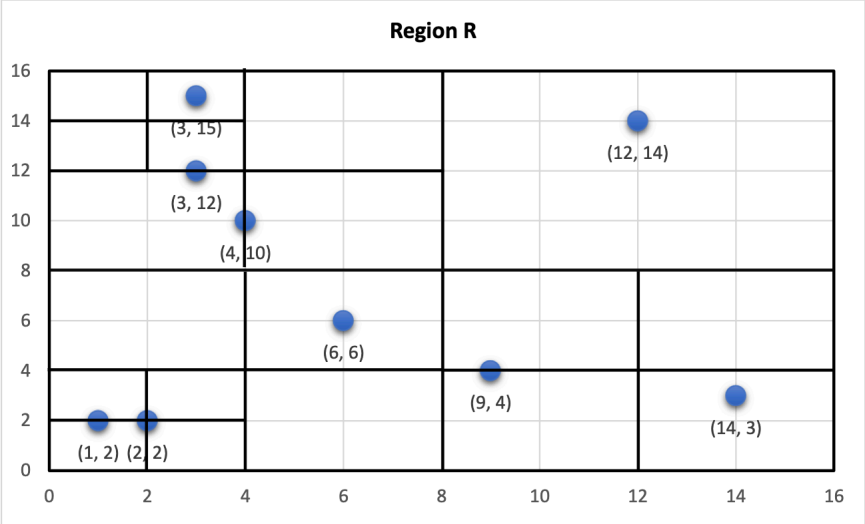
Homework 11 - Solution

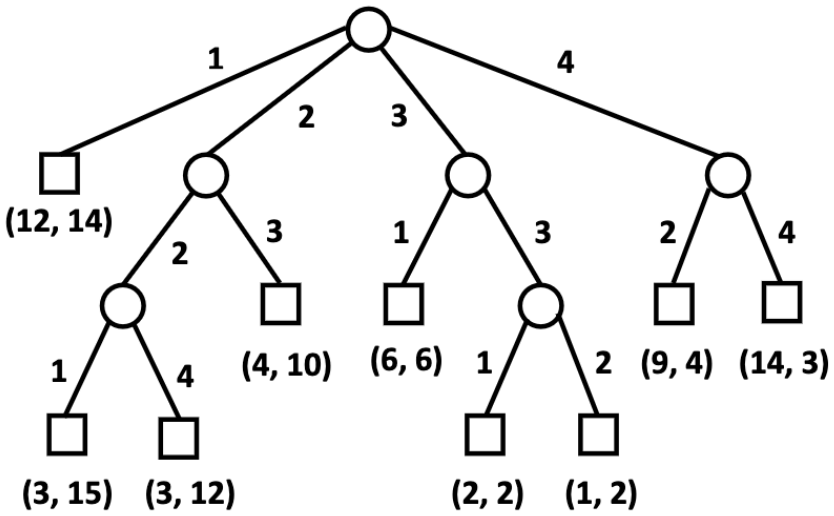
R-21.7



For the above plot, following are the criteria for points on the boundaries:

- A point is within the region if it is on the bottom line of the box.
- A point is within the region if it is on the right line of the box.
- A point is within the region of the right upper box if it is on the center of the four boxes.

Quad Tree



C-21.4

Use a balanced binary tree, T, which stores the items of S in its external nodes, ordered by their keys. At each internal node, v, store the number of external nodes in the subtree rooted at v. With this additional information, we can perform a binary search for any rank in the set. Thus, we can use an algorithm like the 1DTreeRangeSearch to find all the items with ranks in the range [a, b] in O (log n + k) time, where k is the number of answers.

We store the Set *S* in a Balanced Binary Search Tree say AVL. Each internal node stores a key & value having the number of items in the left-sub tree rooted at that node.

Insertion: If tree is empty, we insert the item as root. If the tree is not empty we do 1) If the key of new node is greater than the root, we traverse to the right sub-tree 2) If the key of new node is lesser than the root, then increment the root value by 1 & traverse the left sub-tree. We continue doing this until we find a right place to insert. Insertion in AVL will be *log(n).*

Deletion: We need to locate the node to be deleted. If the root is greater than the item, then we decrement the root value by 1 and go to the left sub-tree. If root is smaller, then go to the right sub-tree. Keep checking and updating until the node is located. Once we found it, we delete it and rebalance the tree. Thus totally, deletion runs in *log(n)*.

rangeRank(a,b): Start from root, if the rank of root is less than a, we recurse to the right sub-tree and find the node with rank a. If we find the node, we recurse at the root node & display *a-b* elements. If the rank of root is greater than *a*, we recurse to the left sub-tree and find the node with rank *a*. If we find the node, we recurse at the root node & display *a-b* elements. The running time is *log(n)+k* where *k* is the number of elements reported.

A-21.8

Since we want to do a two-dimensional range query with a four-dimensional range query tool, we will convert these two-dimensional nodes to four-dimensional. For example, say R_1 object is (x_1, y_1), we just need to simply upgrade it to $(x_1, y_1, 1, 1)$. Assume that for data structure D, the four-dimensional range query is defined by $PSTSearch(x_1, x_2, y_1, y_2, v)$. There is a four-sided range, defined by x_1, x_2, y_1 and y_2, and a node v of a priority search tree T.

In this case, we will call $PSTSearch(a, b, 0, 2, T.root())$ to get the desired result. Since it is called only once, the running time is $O(log^3\ n + s)$.

R-22.7

Algorithm planeSweep(*P*, *n*)

Input: a sorted array of points in a plane *P* from left to right with length *n*

Output: a pair of closest points

$p1 \leftarrow P[0]$

$p2 \leftarrow P[1]$

$minDistance \leftarrow distance(p1, p2)$

$for\ i\ from\ 0\ to\ n-1, do$
    $c \leftarrow halfCircle(P[i], minDistance)$
    $if\ any\ point\ p\ inside\ c\ then$
    $minDistance \leftarrow distance(P[i], p])$
        $p1 \leftarrow P[i]$
        $p2 \leftarrow p$
$return\ p1, p2$

C-22.8
Here we use modified Segment Intersect Algorithm. We modify the algorithm so that, we always check the intersection pairs that are only intersected at the endpoint of two segments. Finding this will be $O(1)$ so the running time will be $O(n \log n + s)$. And s here will only the number of endpoints intersection.

Through modified Segment Intersect algorithm, we can get endpoint intersection segment pairs in $O(n \log n)$. If segment p and q are intersected only at endpoints, then the output will contain (p, q). But now, we redefine p and q such that we add (p, q) as the edge between them. By doing for other segments we obtain a graph G. Then, we do DFS and test if there is a circle inside. If so then there is a simple polygon in C. Else, no simple polygon in C. Since DFS runs in $O(n)$, the running time is $O(n \log n)$.

A-22.7
We can perform two Graham scans on the red and blue points in set S. We can first split up the red and blue points to give us two separate sets, taking $O(n)$ time. Then we perform the two Graham scans, both taking $O(n\ log\ n)$ time, but giving us two simple polygons with no nonconvex vertices. Therefore, if any of the two polygons intersect, which will also take $O(n\ log\ n)$ time, we know that there is no line that can exist to separate the red and blue points. If they don't intersect, a line does exist between the points.