CS 600 A - Advanced Algorithms – Homework 11

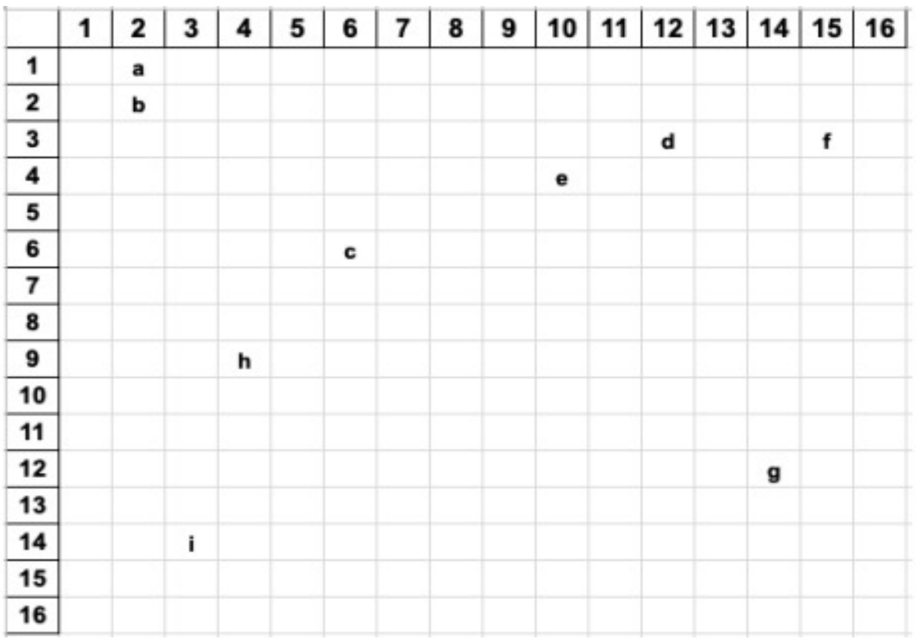
Name: Deepali Nagwade | 20013393

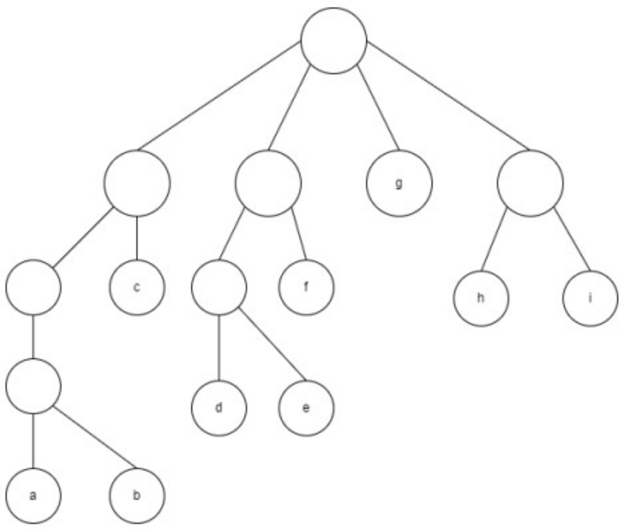
Question 21.5.7: Draw a quadtree for the following set of points, assuming a 16 x 16 bounding box:

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

Answer: Here is a quadtree for the following set of points, assuming a 16 x 16 bounding box:

First let's construct a 16x16 box and locate points here.





Question 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.

Answer: To efficiently support the rankRange(a, b) method for a set S of n items with ordered keys, we can use an augmented data structure based on a balanced binary search tree (BST) or a self-balancing binary search tree, such as an AVL tree or a Red-Black tree. Here's a description of the data structure and its methods:

Augmented Binary Search Tree (BST):

1. **Node Structure:** Each node in the BST is augmented to store additional information about the size of its subtree (the number of nodes in its subtree).

```
class Node:
    def __init__(self, key, left=None, right=None, size=1):
        self.key = key
        self.left = left
        self.right = right
        self.size = size  # Size of the subtree rooted at this node
```

2. **Insertion:** When inserting a new item with a key **k**, insert it as in a regular BST. While traversing the tree, update the **size** field of each visited node.

```
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.key:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    root.size += 1
    return root
```

3. **Deletion:** When deleting a node with a key **k**, delete it as in a regular BST. While traversing the tree, update the **size** field of each visited node.

```
def delete(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = delete(root.left, key)
    elif key > root.key:
        root.right = delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        root.key = minValueNode(root.right)
        root.right = delete(root.right, root.key)
    root.size = 1 + getSize(root.left) + getSize(root.right)
    return root
```

4. **rankRange(a, b):** The **rankRange** method can be implemented recursively. Given a key **k** and a node in the tree, the rank of **k** in the BST is equal to the size of the left subtree plus 1.

```
    def rankRange(root, a, b):
        if root is None:
            return []
        rank = getSize(root.left) + 1
        if rank > b:
            return rankRange(root.left, a, b)
        elif rank < a:
            return rankRange(root.right, a - rank, b - rank)
        else:
            return [root.key] + rankRange(root.right, 0, b - rank)
```

Running Times:

1. **Insertion:**
    - Average Case: O(log n) for balanced BSTs.
    - Worst Case: O(n) for unbalanced BSTs, but self-balancing trees guarantee O(log n) worst-case time.
2. **Deletion:**
    - Average Case: O(log n) for balanced BSTs.
    - Worst Case: O(n) for unbalanced BSTs, but self-balancing trees guarantee O(log n) worst-case time.
3. **rankRange(a, b):**
    - O(log n) on average, as each recursive call to **rankRange** reduces the problem size by roughly half.

By using an augmented BST and maintaining the size of each subtree, we can efficiently support the **rankRange(a, b)** method while maintaining logarithmic time complexity for insertions and deletions.

The efficiency of the rankRange method stems from the ability to navigate the tree based on the sizes of the subtrees.

Question 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, $\mathcal{S}$ = R 1 , R 2 , … , 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 $\mathcal{S}$, given a query rectangle, R , would return every bounding box, R i , in $\mathcal{S}$, 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 .

Answer: The answer to this question lies in the fact that a bounding box is a four-dimensional point. This means that, given a query rectangle, R, one can find all of the bounding boxes, R i , in $\mathcal{S}$ that are completely contained inside R by simply querying the data structure D for all points that lie inside R. This query will 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.

Given a query rectangle, R, one can find all of the bounding boxes, R i , in $\mathcal{S}$ that are completely contained inside R by simply querying the data structure D for all points that lie inside R. This query will 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.

Question 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.

Answer: Pseudo-code :

1 . First we can know that both x axis and y axis and draw the lines .

2 . Then sort the list of points in the x axis from left to right .

3 , Then check the closest point in the x axis .

4 . Find the closest pair in the x plane .

5 . Then sort the list of points in the y axis from right to left .

6 . Then check the closest point in the y axis .

7.. Find the closest pair in the y plane .

8 . And compare the closest pair of x plane and y plane .

9 . If X plane elements is closer than y plane then display the result of x coordinate elements .

10.if Y plane elements is closer than x plane then display the result of y coordinate elements .

Question 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.

Answer: It is given that in a collection C of n horizontal and vertical line segments to form a simple polygon we can use.

1 . Using plane sweep , determine all pairs of intersecting segments with common coordinates.

2 . As seep line SL move from left to right find the coordinates which have common horizontal and vertical points in the plane .

3 . If a common coordinate is found then store it in dictionary , whenever we find another line segment check in the dictionary if they have common endpoints .

4 . In this form we can return all the coordinated in the dictionary and check if they form the closed loop . It means there exists a polygon . Running time : Number of points in collection C is n . Plane Sweep algorithm will take O (n log n) . Moving for coordinates for the line segments will cover each line will take O (n) . So , algorithm runs in O (n log n) time.

We can use the modified sweep line algorithm for detection of pairs of intersecting lines and store the pairs in the dictionary and if we find a vertical line which is intersecting two horizontal lines and start a convex polygon , we just need to find another vertical line which will close the polygon by finding another pair of intersecting lines .

Question 22.6.7 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 coloured 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 L 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?

Answer: 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(n log n) to create a convex hull using the Graham Scan Algorithm.