INFORMATION SYSTEMS
ASSIGNMENT 3

Submission by:

Ayça Avcı [S4505972]
Deepshi Garg [S4199456]

TASK 4.2:
Implementing the QuadTree (*def recurse(self,bbox, depth)* in quadtree.py)

- The QuadTree is a tree data structure, where each parent has exactly 4 children. In our example, it is used to define points in a 2 dimensional spatial space. Thus, a tree defines an area within a bounding box on a 2D cartesian plane. The children nodes of a parent BoundingBox split into 4 equal sized BoundingBoxes. These children boxes fill the exact space as their parent.
- In our implementation, we build the QuadTree using recursion as follows:
    - It takes the current bounding box *bbox* and *depth* level as input
    - First, we define the limiting condition of the recursion. The depth of the current level should be less than or equal to the maximum depth of the QuadTree. If it is greater than tree depth, we exit.
    - If the above condition satisfies, we need to divide the current bounding box into 4 equal parts. Thus, by geometry, the height and width of each child box would be half the original values. We calculate them as *new_width* and *new_height.*
    - Now we define the vertices of the bounding boxes. These include the vertices of original bounding box, i.e., *(min_x, min_y), (min_x, max_y), (max_x, min_y)* and *(max_x, max_y).* In addition, child boxes will also have a vertex at the center of the parent box, i.e, *(med_x, med_y)*.
    - Next, we instantiate all the 4 child bounding boxes : *upper_left, upper_right, bottom_left* and *bottom_right.*
    - We add these child bounding boxes to the list of all the *quads* of the tree. If there are no *quads* already existing at this *depth*, we instantiate the mapping at this *depth*, otherwise, we simply append to the previously existing mapping.
    - Now we increase the *depth* by 1 for computing child nodes of next level
    - Recursion is called for each of the child nodes to compute their children

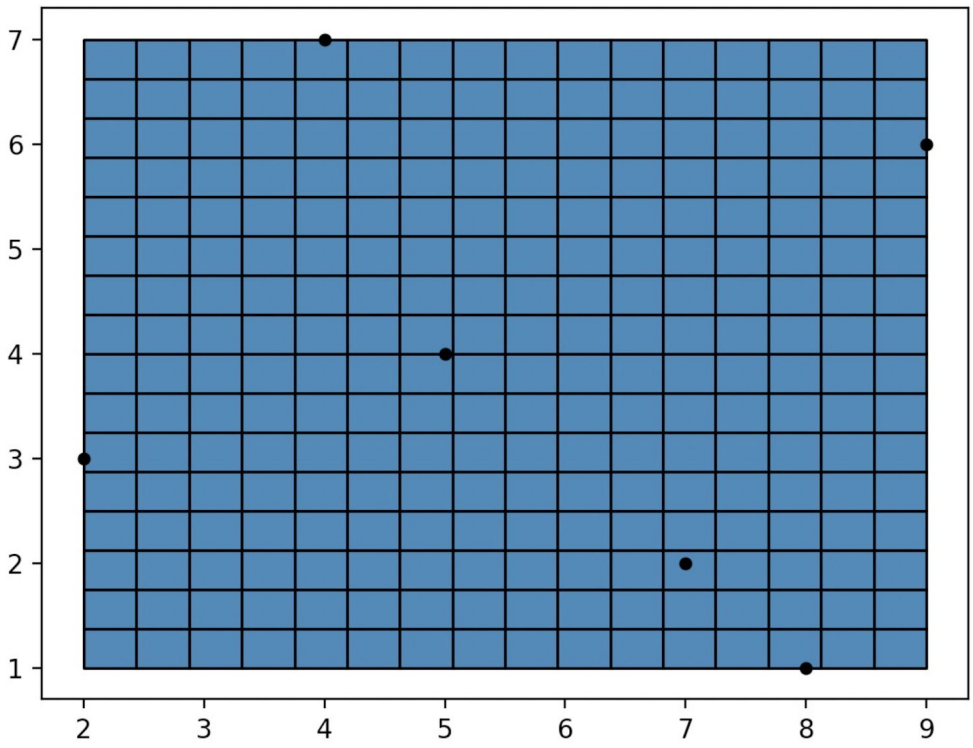
```
def recurse(self, bbox, depth):

      if depth > self.depth:
            return

      new_width = bbox.width()/2
      new_height = bbox.height()/2

      min_x, max_x = bbox.data[0, 0], bbox.data[0, 1]
```

```
      min_y, max_y = bbox.data[1, 0], bbox.data[1, 1]
      med_x, med_y = min_x + new_width, min_y + new_height

      upper_left = bb.BoundingBox(min_x, med_x, med_y, max_y)
      upper_right = bb.BoundingBox(med_x, max_x, med_y, max_y)
      bottom_left = bb.BoundingBox(min_x, med_x, min_y, med_y)
      bottom_right = bb.BoundingBox(med_x, max_x, min_y, med_y)

      if not depth in self.quads:
            self.quads[depth] = [upper_left, upper_right, bottom_left,
bottom_right]
      else:
            self.quads[depth].append(upper_left)
            self.quads[depth].append(upper_right)
            self.quads[depth].append(bottom_left)
            self.quads[depth].append(bottom_right)

      depth = depth + 1
      self.recurse(upper_left, depth)
      self.recurse(upper_right, depth)
      self.recurse(bottom_left, depth)
      self.recurse(bottom_right, depth)

      return
```

- On running the above code for the following input, we got the image shown below:
    - Input

```
python3 plot_kdtree.py --quadtree 4 --quadshow True
```

    - Output Image



TASK 4.3:
Implementing the KDTree (*def closest(self, point, sidx = si.StorageIndex())* in kdtree.py)

- The *closest* function returns the unique list of *keys* that fall within the bounding box which contains the given *point*. In terms of KDTree formation, it returns all the data points within the same leafnode.
- Our implementation is as follows:
    - First we check if the current *index* is the leafnode or not. If it is a leafnode, we return all the data points stored as *elements* in this node.
    - If the current *index* is not a leafnode, we retrieve its *axis* and *partition* value from the *storage.*
    - Next we decide whether we need to traverse towards the left or right. This decision is stored in the variable *left*. It is decided to move *left* if the *point* value is less than or equal to the partition value for the *axis* of this node, which means that in the KDTree structure, this *point* lies to the left of the *partition*. If the *point* value is more than the *partition* value for the current *axis*, we traverse right
    - Based on the traversal direction, a recursive call is made
    - The *keys* obtained from each recursion call are finally returned to the caller.

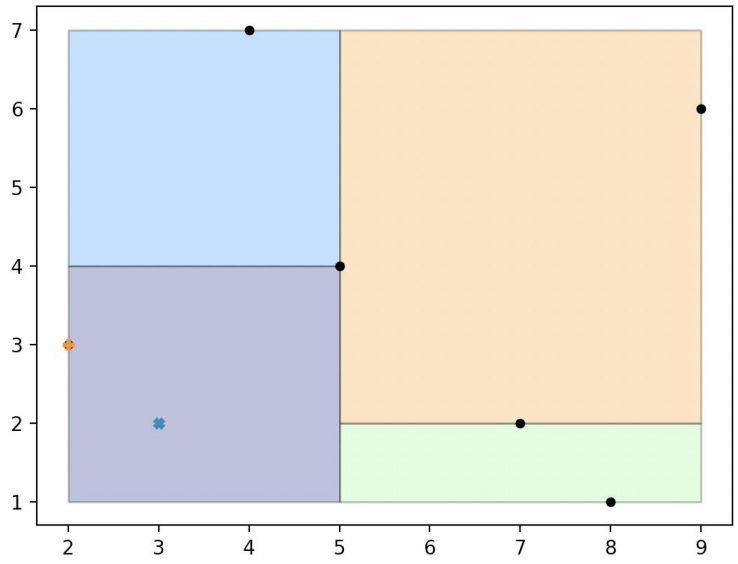
```
def closest(self, point, sidx=si.StorageIndex()):
```

```
    if "elements" in self.storage[sidx.storage()]:
          return self.storage[sidx.storage()]["elements"]
    else:
          axis = self.storage[sidx.storage()]["axis"]
          partition = self.storage[sidx.storage()]["partition"]
          left = point[axis] <= partition

          keys = []
          if left:
                keys.extend(self.closest(point, sidx.left()))
          else:
                keys.extend(self.closest(point, sidx.right()))

    return keys
```

- On running the above code for the given input command, the following output graph was obtained:
    - Input

```
python3 plot_kdtree.py --closest "3 2"
```

    - Output



TASK 4.4:
Using the QuadTree depth to subsample the KDTree (in plot_kdtree.py)

- To implement the subsampling the KDTree using the QuadTree depth, we followed the below steps that are specified in the assignment pdf as well:
    - Initially, the *quad* field for all records in the *db* is set to the max quad-level, i.e., *args.quadtree*.
    - Next we retrieve all the quads of the *quadtree* formed using the *quadrants()* function and store them into *all_quads* variable.
    - All levels of the *quadtree* are iterated in reverse order (maximum level to 0).
    - If *all_quads* contains quads for this *level,* they are retrieved into *current_level_quads.*
    - Next, we iterate through every *quad* in *current_level_quads,* and calculate its *centroid* using the *centroid()* method of *BoundingBox* class.
    - We find all the closest keys for this *centroid* in the KDTree. We convert these keys into data points using the *query()* method of *Database* class.
    - Next we want to find the single closest point to the centroid. This is done by calculating the Euclidean Distance between *centroid* and each *point* in the closest_points list obtained in the previous step.
    - After the *closes_point* is found out, we change its *quad* value in the *db* to be the current quad value, i.e., *level*

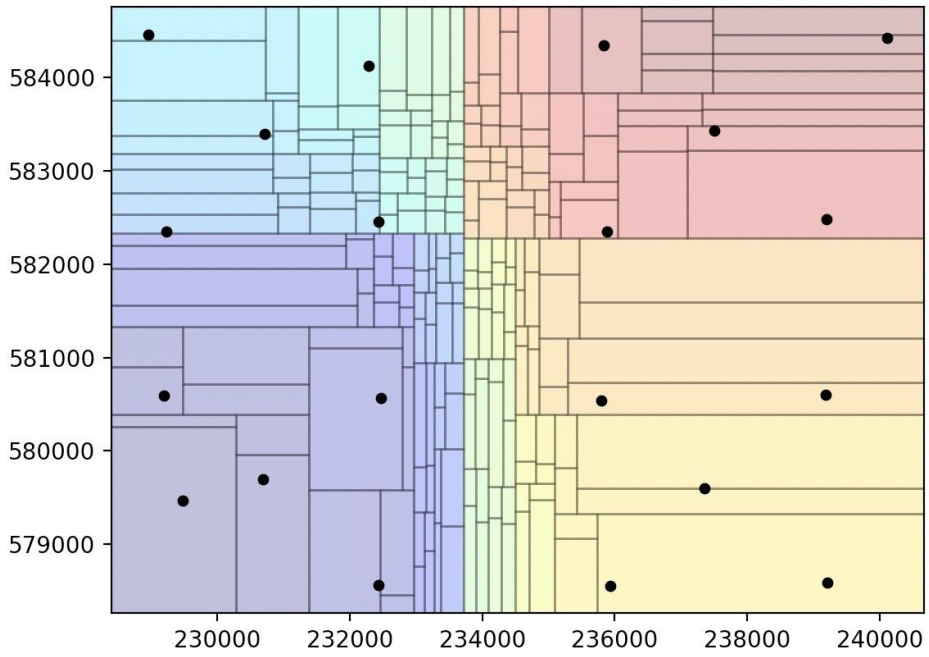
```
db_keys = dtb.keys()

for key in db_keys:
    dtb.update_field(key, "quad", args.quadtree)

all_quads = quadtree.quadrants()

for level in range(args.quadtree-1, 0, -1):
    if level in all_quads:
        current_level_quads = all_quads[level]

        for quad in current_level_quads:
            centroid = quad.centroid()
            closest_points = dtb.query(tree.closest(centroid))

            shortest_distance = float("inf")
            closest_point = closest_points[0]

            for point in closest_points:
                current_point_distance = math.sqrt(
        ((point[1]-centroid[0])**2) + ((point[2]-centroid[1])**2))

                if current_point_distance < shortest_distance:
                    shortest_distance = current_point_distance
                    closest_point = point
```

```
            closest_point_key = closest_point[0]
            dtb.update_field(closest_point_key, "quad", level)
```

- On running the above code for 2 different inputs, following outputs are received:
    - Input 1:

```
 python3 plot_kdtree.py --filename
../data/addressen_groningen/adressen_groningen.shp --bbox-depth 8
--max-depth 9 --plot kdtree-bb --quadtree 4 --quadlevel 3
```

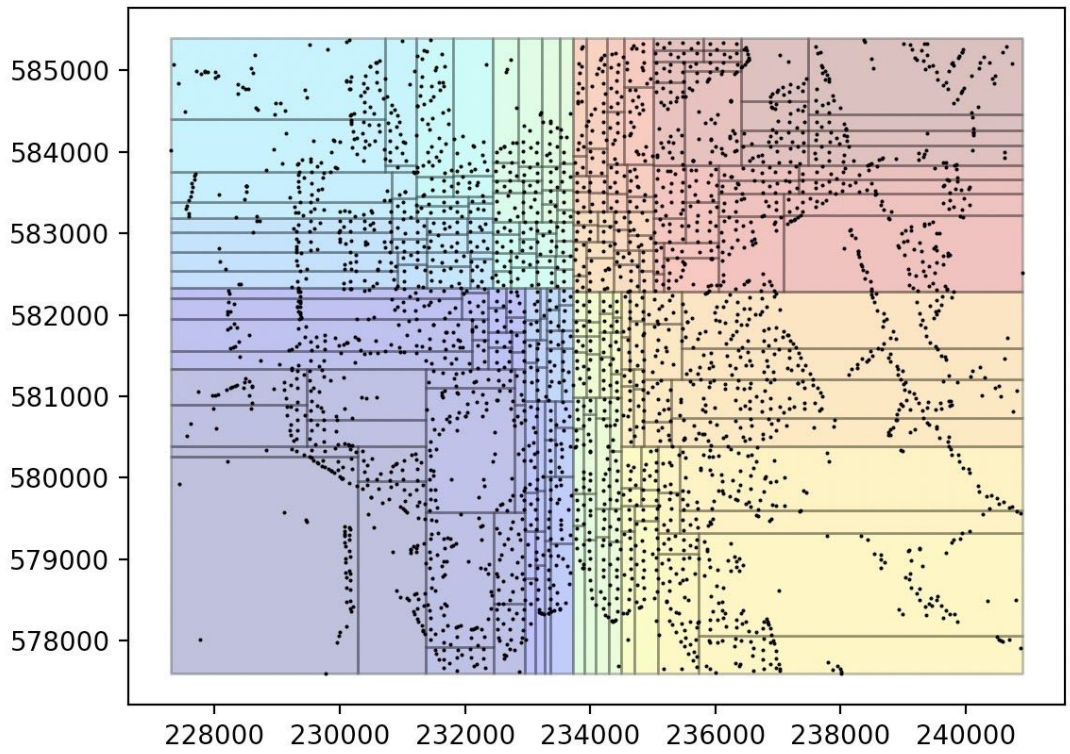
    - Output 1:



    - Input 2:

```
 python3 plot_kdtree.py --filename
../data/addressen_groningen/adressen_groningen.shp --bbox-depth 8
--max-depth 9 --plot kdtree-bb --quadtree 8 --quadlevel 7
```

    - Output 2:


585000
584000
583000
582000
581000
580000
579000
578000
228000
230000
232000
234000
236000
238000
240000