QUAD TREE DATA STRUCTURE

GROUP DETAILS:

PRATTAY BARI (2351071)	BRANCH	CSE
GIRIRAJ ROY (2351090)	YEAR	2nd / 3rd Sem
SOURYA ADHIKARY (2351091)	SECTION	B
AYUSH BASAK (2351094)	SUBJECT	Data Structures and Algorithms
KAUSIK KUMAR DAS (2351095)		
RUDRA ROY (2351096)		

What is a Quad Tree?

A quadtree is a tree data structure invented in 1974 by Raphael Finkel and J.L. Bently.

It is used to partition a two-dimensional space (Usually Square) by recursively subdividing it into four square or rectangular quadrants.
It is particularly useful in spatial indexing, image compression, and other applications that involve 2D data.

Some Common Features of a Quad Tree:

1. They decompose space into adaptable cells.
2. Each cell (or bucket) has a maximum capacity. When maximum capacity is reached the bucket splits.
3. The tree directory follows the spatial decomposition of the quadtree.

Types of Quad Tree:

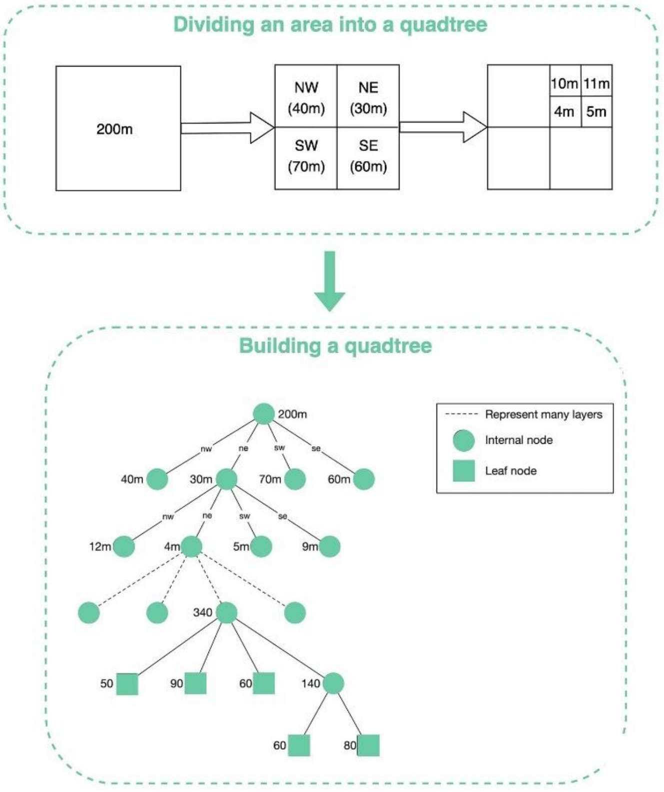
Point Quadtree: Represents points in 2D space. Each node stores a point, and the space is recursively divided into four quadrants based on the position of the points. Used for spatial indexing and searching.

Region Quadtree: Used to represent 2D regions (like images or maps). It divides space into quadrants, where each node represents a square region. Subdivision occurs until regions become homogeneous (same value within a quadrant).

Edge Quadtree: Focuses on representing line segments. Space is recursively divided, and the nodes store or subdivide based on the edges intersecting a quadrant. It's useful for vector data or representing boundaries.
Polygonal Quadtree: Similar to region quadtrees but used to represent polygons or more complex shapes. The subdivision continues until the nodes accurately capture the polygon's geometry. Used in graphics and GIS**.**
Compressed Quadtree: Optimized for space efficiency. Large homogeneous regions are represented by a single node, skipping intermediate subdivisions. This type is used in image compression and geographic information systems for handling sparse data.

Key Concepts of Quadtree :

Node Structure: Each node in a quadtree represents a square or rectangular bounded region. If the region contains more data points than a given threshold or is non-homogeneous it is subdivided into four equal sub-regions. These sub-regions are the children of that node.



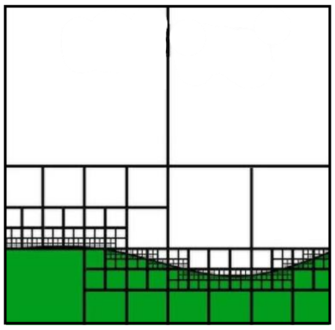
Subdivision:
The space is recursively divided into four quadrants:
Top-left (NW)
Top-right (NE)
Bottom-left (SW)
Bottom-right (SE)

Root Node:
Root nodes are the nodes which are created in the beginning and its boundary is the whole region of the quad tree

Leaf and Internal Nodes:
Leaf Nodes are regions where no further subdivision is needed, usually because they meet a certain condition like homogeneity.

Internal Nodes are nodes that have been subdivided and have four child nodes.

Applications of Quadtree:



1. Image Representation: A quadtree can represent an image where each node represents a region with uniform colour. The tree structure allows for efficient storage and quick lookups.

2.Spatial Indexing: Quadtrees are useful in geographic information systems (GIS) to represent spatial data such as maps, where each quadrant of the map might represent a different region.

3.Collision Detection: In computer graphics or gaming, quadtrees can efficiently detect objects in close proximity.

4. **Conway's Game of Life** simulation program can be written using Quad Tree

5. Solution of multidimensional fields : computational fluid dynamics and electromagnetism field

And many other things etc.

Common Operations Done on a Quad Tree:

1. Creation of a Quad Tree
2. Insertion of a point in a Quad Tree
3. Deletion of a point
4. Searching a point
9. Querying points within a given range
5. Updating a points value
7. Subdivide
10. Display all the points

○ **Rather than these there are some helping functions needed like**

1. Boundary intersection check
2. Points boundary check
3. A node is leaf node or not etc.

Structures of a Quad Tree Representation:

Structure of a point

```
struct Point {
   int val;
   float x;
   float y;
};
```

x and y contain the (x, y) coordinate of the point on the 2d plane

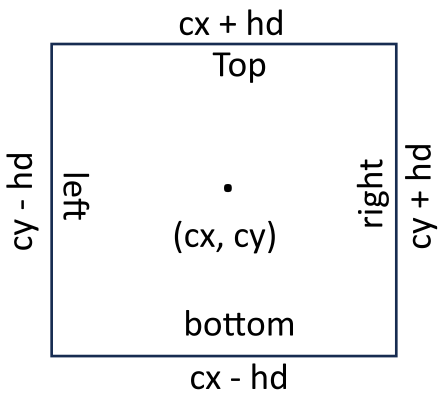
val holds the needed data to be stored at the point

This holds the information about the region of a node in contains the location of the centre point(cx, cy) and the half of its edge length(hd)

Using these two values the boundary can be easily calculated

Structure of a boundary of a region

```
typedef struct BoundryBox {
   Point *center;
   float halfDimension;
};
```

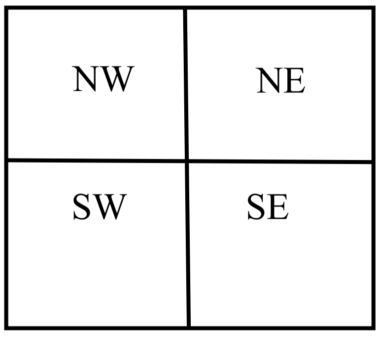


Structure of a Quad tree Node

```
struct Quadtree {
   BoundryBox *boundry;
   Point **points;

   struct QuadTree* NW;
   struct QuadTree * NE;
   struct QuadTree* SW;
   struct QuadTree* SE;
};
```

Holds the boundary of the region and a specific no of points (threshold value)

Also has pointer for all the child nodes



Creation of a Quad Tree:

Step 1: Define Structures

First, we need the Point, BoundaryBox, and QuadTree structures (as described earlier)

Step 2: Create A node

Create the BoundryBox for the node
Allocate memory of threshold size to store points in that node
Set the child pointers as NULL

```
Function CreateQuadTreeNode(boundary):
        // Allocate memory for a new QuadTree node
1       qt = Allocate memory for QuadTree
        // Initialize the four child quadrants to NULL (no subdivisions yet)
2       qt->NE = NULL
3       qt->NW = NULL
4       qt->SE = NULL
5       qt->SW = NULL
        // Assign the boundary region to the new node
6       qt->boundary = boundary
        // Allocate memory for storing points (array of pointers of threshold size)
7       qt->points = Allocate memory for Point array with threshold size
        // Initialize all points in the node to NULL (no points inserted yet)
8       for i from 0 to QT_NODE_CAPACITY - 1:
9               qt->points[i] = NULL
        // Return the initialized quadtree node
10      return qt
```

Step 3: Create Root node

Make the root pointer point to the first created node

Step 4: Insert points

Now as the root node is created insert points at the node if is exceeds the threshold value then subdivide the node and insert at the appropriate child node

Creation of a Point:

The function Point_new takes two parameters, x and y, which represent the coordinates of the point.

```
Function Point_new(x, y):
1       p  = Allocate memory for a new Point
        // Step 2: Set the x, y coordinate of the Point
 2      p->x = x   ,   p->y = y
        // Step 4: Set the initial value of the Point to a maximum integer value
  3     p->val = INT_MAX
```

Creation of a Boundary Box:

The function BoundaryBox_new takes a centre point and a halfDimension as parameters and is responsible for creating a new BoundaryBox.

```
Function BoundaryBox_new(centre, halfDimension):
    1       Allocate memory for a new BoundaryBox box
            // Set the centre of the box
    2       Box->centre = centre
            // Step 3: Set the halfDimension of the box
    3       Box->halfDimension = halfDimension
```

Insertion of a Point in Quad Tree:

Step 1: Boundary Check

First, it checks if the point lies within the boundary of the current quadtree node

Step 2: Check Capacity

If the current node has space and has not been subdivided the point is added to this node's points array

Step 3: Subdivide

If the node is full and has not been subdivided, the node is subdivided into four quadrants

Step 4: Recursive Insertion

After subdividing (if necessary), the function attempts to insert the point into one of the four child quadrants (NW, NE, SW, SE). The first successful insertion returns true

Step 5: Failure

If the point does not fit into any quadrant, the function returns false, indicating that the point could not be inserted (which should not happen unless there's an issue with boundaries)

```
Function QuadTree_insert(root, point):
    Check if the point lies within the boundary of the root
    points_size = Calculate current number of points stored in this node
    // If there is space in this node and it hasn't been subdivided
    if points_size < capacity of Node && root->NW == NULL:
        // Insert the point into the current node
        root->points[points_size] = point
        return true  // Successfully inserted
    // If the node is full and hasn't been subdivided, subdivide it
    if root -> NW == NULL:
        Subdivide the node into four quadrants
    // Try to insert the point into one of the four child quadrants
    If QuadTree_insert(root->NW, point):
        return true
    If QuadTree_insert(root->NE, point):
        return true
    If QuadTree_insert(root->SW, point):
        return true
    If QuadTree_insert(root->SE, point):
        return true

    // If the point cannot be inserted into any child node, return false
    return false
```

<u>Dry Run of Insertion:</u>

<u>Scenario 1: Inserting a point into the root node (when it's not full)</u>

1. **Initial state of the root node**:
    - Let the root node has a boundary and can hold 4 points
    - The root node is not subdivided yet
    - The centre of the root is (10, 10) and half Dimension = 10
    - Let's say there are **2 points** already stored in the root, and new point (5, 5)
2. **Insert (5, 5) into the root**:
    - **Step 1**: The algorithm checks if (5, 5) lies within the boundary of the root node. (5, 5) lies inside the boundary.
    - **Step 2**: The algorithm calculates the current number of points in the root node (points_size = 2).
    - **Step 3**: Since the root node has space (points_size < 4) and is not subdivided (root->NW == NULL), the point (5, 5) is inserted into the root node.
    - **Step 4**: The point (5, 5) is stored at index 2 in root->points.
    - **Step 5**: The algorithm returns true, indicating that the point was successfully inserted into the root.

Final state of the root node:

- The root now contains 3 points.
- No subdivision has occurred, and the node can accept 1 more point before subdivision.

Scenario 2: Inserting a point into a child node (after the root is full and subdivided)

1. **Initial state of the root node**:
    - The root node already has **4 points** stored, and it's **full**.
    - The points stored in the root are:
        - (2, 3) , (4, 4) , (5, 5) , (7, 6)
    - The root is **not subdivided** yet.
2. **Attempt to insert a new point (6, 7)**:
    - **Step 1**: The algorithm checks if (6, 7) lies within the boundary of the root node.(6, 7) lies inside the boundary.
    - **Step 2**: The algorithm calculates the current number of points in the root (points_size = 4), which equals the capacity of the node.
    - **Step 3**: Since the node is full and hasn't been subdivided, the algorithm subdivides the root node into 4 child quadrants (NW, NE, SW, SE).
3. **Subdividing the root**:
    - **Step 4 :**The algorithm creates 4 child quadrants
4. **Insert (6, 7)** into the appropriate quadrant:
    - **Step 5**: The algorithm checks each child quadrant:
        - It checks if (6, 7) lies within which quadrant and finds the SW quadrant, but it doesn't.
    - **Step 6**: The point (6, 7) is stored in the SW quadrant.
    - **Step 7**: The algorithm returns true

Final state after subdivision and insertion:

- The root node is subdivided into 4 quadrants.
- The NE quadrant contains the points (5, 5), (7, 6), and the newly inserted point (6, 7).
- The SW quadrant contains the points (2, 3) and (4, 4).

Time Complexity of Insertion:

Overall Time Complexity: **O(log(n))**.

Subdividing a node of a Quad Tree:

Step 1: Function Definition:

The function takes a root quadtree node as input

Step 2: Check for Subdivision

It first checks if the node is already subdivided. If it's not a leaf node (meaning it has already been subdivided), it returns the existing root without making changes.

Step 3: Calculate Half Dimension:

Calculate halfDim, which is half the dimension of the current node's boundary. This will be used to define the boundaries of the child nodes.

Step 4: Create The Child Nodes:

It calculates the centre point for the North West (NW), North East (NE), South West (SW) and South East (SE) quadrant and creates a new child node for each

```
Function QuadTree_subdivide(root):
1       Check if the node is already subdivided
2       Step 2: Calculate half of the dimension for the child nodes
3       halfDim = root.boundary.halfDimension / 2
        //Create North West child node
 4      nw_center = Point_new(root.boundary.center.x - halfDim,
                                        root.boundary.center.y + halfDim)
 5      root.NW = QuadTree_new(BoundaryBox_new(nw_center, halfDim))
        // Create North East child node

 6      ne_center = Point_new(root.boundary.center.x + halfDim,
                                        root.boundary.center.y + halfDim)
 7      root.NE = QuadTree_new(BoundaryBox_new(ne_center, halfDim))
        // Create South West child node
```

```
8     sw_center = Point_new(root.boundary.center.x - halfDim,
                                                  root.boundary.center.y - halfDim)
9     root.SW = QuadTree_new(BoundaryBox_new(sw_center, halfDim))
      // Create South East child node
10    se_center = Point_new(root.boundary.center.x + halfDim,
                                                            root.boundary.center.y - halfDim)
11    root.SE = QuadTree_new(BoundaryBox_new(se_center, halfDim))
12    return root
```

QuadTree_new,BoundryBox_new,Point_new are the functions to create new Quad Tree Node,New boundary box and New Point

---

Searching a Point in a Quad Tree:

Step 1: Function Definition

The function takes a root quadtree node and a point to search for.

Step 2: Null Check

It checks if the root node is NULL, indicating an empty quadtree.

Step 3: Boundary Check

It verifies if the point is within the current node's boundary.

Step 4: Point Array Check

It loops through the node's points array to see if the point exists in the current node.

Step 5: Subdivision Check

If the node has not been subdivided, it returns false.

Step 6: Recursive Search

If the node is subdivided, it recursively searches in the appropriate child quadrant based on the point's location.

```
Function Point_Search(root, point):
1     if root is NULL:
2         return false  // The quadtree is empty
3     Check if the point is within the boundary of the current node
      // Check if the point is in the points array of the current node
4    for i from 0 up to capacity of the Node:
5       if root->points[i]!= NULL && root->points[i]->x == point->x and root->
points[i]->y == point->y:
6           return true
```

```
   // If the node has not been subdivided
7    if root->NW is NULL:
8        return false  // The point is not found
   // Recursively search in the child quadrants
9     if the point is in top left childs boundry
10        return Point_Search(root->NW, point)
11   else if the point is in top right childs boundry
12        return Point_Search(root->NE, point)
13    else if the point is in bottom left childs boundry
14        return Point_Search(root->SW, point)
15    else:
16        return Point_Search(root->SE, point)
```

Dry Run for searching:

Assumptions:

- The quadtree structure has been subdivided.
- Each node can store up to 4 points.
- The quadtree root has four child nodes (NW, NE, SW, SE).
- Points already in the quadtree:
    - **Root**: Contains 4 points
    - **NW**: Contains (1, 11)
    - **NE**: Contains (7, 11)
    - **SW**: Contains (2, 3)
    - **SE**: Contains (7, 2)

Scenario 1: Successful Search

Let's search for point (7, 11).

1. **Step 1-2**: Check if the root is NULL. It is not, so proceed.
2. **Step 3**: Check if point (7, 11) lies within the boundary of the root. It does, so proceed.
3. **Step 4-6**: Iterate over the points in the root node to check if (7, 11) is stored there.
4. **Step 7**: Check if the root is subdivided. It is (child nodes exist), so proceed.
5. **Step 9-16**: Check which child node's boundary contains (7, 11):
    - It's not in the NW boundary.
    - **Step 12**: It is in the NE boundary, so the function recurses into the NE
6. **Recursive Call on NE**:
    - **Step 1-2**: Check if NE node is NULL. It is not.
    - **Step 3**: Check if point (5, 5) lies within the boundary of NE. It does.

  - **Step 4-6**: Iterate over the points in the NE node, found (7, 11) in the NE node.
  - **Step 6**: Return true because the point is found.

Scenario 2: Unsuccessful Search

Let's search for point (4, 4) (which is not present in any node).

Dry Run:

1. **Step 1-2**: Check if the root is NULL. It is not, so proceed.
2. **Step 3**: Check if point (4, 4) lies within the boundary of the root. It does, so proceed.
3. **Step 4-6**: Iterate over the points in the root node. Check if it is there
4. **Step 7**: Check if the root is subdivided. It is, so proceed.
5. **Step 9-16**: Check which child node's boundary contains (4, 4):
   - **Step 11:**It's not in the NW boundary.
   - **Step 12**: It's not in the NE boundary.
   - **Step 14**: It's not in the SW boundary.
   - **Step 16**: Check the SE boundary, but the point is not within SE either.
   - **Step 16**: All child quadrants have been checked, and the point was not found. return false

<u>Time Complexity of Searching:</u>

Overall Time Complexity: **O(log(n))**.

<u>Deletion of a point :</u>

Step1 : Search for the point

Step 2 : Delete the point

Deallocate the memory where point is stored make the pointer pointing to the point NULL and shift the rest of the elements forward by 1

```
free(point);
 points = NULL
 shift the rest by 1 position
```

<u>Dry Run of Deletion:</u>

It is same as searching just after finding the point we deallocate the memory

<u>Time Complexity of Searching:</u>

Overall Time Complexity: **O(log(n))**.

Updating a Points Value:

Step1 : Search for the point
Step 2 : Update the value

Ask the user for a new value and store it at the point

Dry Run of Updating:

It is same as searching just after finding the point we ask the user to enter a new value and change the original value with it

Time Complexity of Searching:

Overall Time Complexity: O(log(n)).

Querying Values in a Given Range:

Step 1: Function Definition

The function takes a root quadtree node and a range BoundaryBox as parameters to find all points within the specified range.

Step 2: Initialize Result Array

It allocates memory for the result array, which will store points found within the range. All elements of the result array are initialized to NULL.

Step 3: Intersection Check

It checks if the root's boundary intersects with the query range. If there's no intersection, it returns the empty result array.

Step 3: Get Points Size

Retrieves the number of points currently stored in the node

Step 4: Point Check and Add

It loops through the points in the current node and checks if each point is contained within the specified range. If so, it adds the point to the result array.

Step 5: Subdivision Check

it checks if the node has been subdivided by verifying if root.NW is NULL. If it hasn't been subdivided, it returns the result array.

Step 6: Recursive Query

It recursively queries each of the four child quadrants (NW, NE, SW, SE) for points within the specified range, adding any found points to the main result array.

Step 7: Return Result

Finally, it returns the complete result array containing all points found within the specified range.

```
Function QuadTree_query_range(root, range)
1       result = Allocate Memory for the array
2       Initialize all elements in the result array to NULL
3       Check if the root's boundary intersects with the query range
4       if not return result  // No intersection, return empty result
5       points_size = Get the no  of points in the current node
        // Check points in the current node and add to result if they are within range
6       for i from 0 to points_size - 1:
7           if  root->points[i] lies within the range
8               result[index++] = root->points[i]  // Add the point to the result
        //  Check if the node has been subdivided
9       if root->NW is NULL:
10          return result  // Return result if there are no child nodes
        // Recursively query each child quadrant and add results to the main result
11      i = 0
12      nw_r = QuadTree_query_range(root->NW, range)
13      while nw_r[i] is not NULL
14          result[index++] = nw_r[i++]
15      i = 0
16      ne_r = QuadTree_query_range(root->NE, range)
17      while ne_r[i] is not NULL
18          result[index++] = ne_r[i++]
19      i = 0
20      sw_r = QuadTree_query_range(root->SW, range)
21      while sw_r[i] is not NULL
22          result[index++] = sw_r[i++]
23      i = 0
24      se_r = QuadTree_query_range(root->SE, range)
25      while se_r[i] is not NULL
26          result[index++] = se_r[i++]
27      return result
```

Dry Run Range Query:

Let's go through a dry run of the QuadTree_query_range function with a specific scenario:

Assumptions:

- The quadtree has been subdivided and is holding points
  - **Root : (**1,1) , (2,11) , (11,11) , (9,9)
  - **NW**: (1,11), (2, 12).
  - **NE**: (6, 7), (6, 6).
  - **SW**: (2, 3).
  - **SE**: (8, 2).
- Query range is a rectangular boundary that encompasses the region from (4, 4) to (8,8) centre =(6,6), hd = 2
-

Scenario: Querying range (4, 4) to (8, 8)

Dry Run:

1. **Step 1**: Allocate memory for the result array of size 100
2. **Step 2**: Initialize all elements in the result array to NULL.
3. **Step 3**: Check if the root boundary intersects the query range, it does.
4. **Step 4**: Calculate the number of points stored in the root node.
5. **Step 6-8**:Check if the points root node, falls in the boundry then store them in results array
6. **Step 9**: Check if the root is subdivided. It is, so proceed with recursive queries to the child nodes.

Querying the NW (Northwest) Node:

8. **Step 11**: Set i = 0.
9. **Step 12**: Recursively call QuadTree_query_range for the **NW** node.
10. **Step 3**: The boundary of the NW node intersects the query range.
11. **Step 5**: The NW node contains two points: (1, 11) and (2,1 2).
12. **Step 6-8**: No points lie within the query range

Querying the NE (Northeast) Node:

14. **Step 15**: Set i = 0.
15. **Step 16**: Recursively call QuadTree_query_range for the **NE** node.
16. **Step 3**: The NE node's boundary intersect with the query range (6, 7) and (6, 6) are inside of query range)
17. **Step 4**: Begin transferring results from nw_r to result:
    result[0] = (6, 7)
    result[1] = (6, 6)

Querying the SW (Southwest) Node:

18. **Step 19**: Set i = 0.
19. **Step 20**: Recursively call QuadTree_query_range for the **SW** node.
20. **Step 3**: The boundary of the SW node intersects the query range.
21. **Step 5**: The SW node contains one point: (2, 3).
22. **Step 6-8**: The point (2, 3) don't lie within the query range

Querying the SE (Southeast) Node:

24. **Step 23**: Set i = 0.
25. **Step 24**: Recursively call QuadTree_query_range for the **SE** node.
26. **Step 3**: The SE node's boundary intersect with the query range.
27. Step 4: The point (8,2) don't lie within the query range
28. **Step 4**: Return an empty result from the SE node.

Final Result:

28. **Step 27**: Return the result array, which now contains:

- result[0] = (6, 7)
- result[1] = (6, 6)
- Remaining elements are NULL.

Output:
The query result will return the points (6,7),(6,6) which are all the points that lie within the range (4, 4) to (8,8)

Time Complexity of Searching:

Overall Time Complexity: **O(k + log(n))**, where k is the number of points in the range, and log(n) is the depth of the tree.

Displaying All the points:

It is nothing but querying in whole boundry of root ,pass the boundry parameter as root->boundry

Time Complexity:

To display the entire quadtree, each point must be printed. If there are n points, this operation takes O(n).

Boundary Intersection check:

Inputs: Two boundary boxes are passed, each having:
Checking overlap along the x-axis:

- Check if the right edge of boundry1 is greater than the left edge of boundry2 as well as the left edge of boubdry1 is less than the right edge of other.

Checking overlap along the y-axis:

- Similarly check if the top edge of boundry1 is greater than the bottom edge of boundry2 as well as if the bottom edge of boundry1 is less than the top edge of boundry2 .

If both of these are true then the boundaries overlap each other

---

Checking If BoundryBox Contains the Point:

X-coordinate check:

- If the x coordinate of the point is less than the left edge the point is outside the box on the left side.
- If the x coordinate of the point is greater than the right edge the point is outside the box on the right side.

Y-coordinate check:.

- If the y coordinate of the point is less than the bottom edge the point is outside the box below.
- If the y coordinate of the point is greater than the top edge the point is outside the box above.

Checking If the Node is Leaf Node:

If all of the child node pointers are NULL then the node is a leaf node

Program to implement Quad Tree in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define QT_NODE_CAPACITY 4
#define MAX_ARRAY_SIZE 1000

typedef struct Point {
   int val;
   float x;
   float y;
} Point;

typedef struct BoundryBox {
   Point *center;
   float halfDimension;
} BoundryBox;

typedef struct QuadTree {
   BoundryBox *boundry;
   Point **points;

   struct QuadTree* NW;
   struct QuadTree* NE;
   struct QuadTree* SW;
   struct QuadTree* SE;

} QuadTree;

//Helping functions
Point *Point_new(float x, float y);
BoundryBox *BoundryBox_new(Point *center, float halfDimension);
bool BoundryBox_cotains_point(BoundryBox *boundry, Point *point);
bool BoundryBox_intersects_BoundryBox(BoundryBox *self, BoundryBox *other);
int QuadTree_points_size(Point *points[]);
bool Is_LeafNode(QuadTree quad);
```

```
//Creates a Quadtree Node
QuadTree *QuadTree_new(BoundryBox *boundry);
bool QuadTree_insert(QuadTree *root, Point *point);

//Divides a Quadtree in 4 sub Nodes
QuadTree *QuadTree_subdivide(QuadTree *root);
//Deletes a point
int Delete_Point(QuadTree *root,Point * point);
//updates value of a point
bool Point_Update(QuadTree *root,Point * point);

//Searching functions
bool Point_Search(QuadTree *root,Point * point);
Point **QuadTree_query_range(QuadTree *root, BoundryBox *range);

//Functions for display
void Point_print(Point *point);
void BoundryBox_print(BoundryBox *box);
void display_QuadTree(QuadTree *quad);

Point *Point_new(float x, float y) {
   Point *p = (Point *)malloc(sizeof(Point));
   p->x = x;
   p->y = y;
   p->val = __INT_MAX__;
   return p;
}

void Point_print(Point *point) {
   printf("(%2.2f, %2.2f)\n", point->x, point->y);
   printf("Value: %d\n",point->val);
}

BoundryBox *BoundryBox_new(Point *center, float halfDimension) {
   BoundryBox *box = (BoundryBox *)malloc(sizeof(BoundryBox));
   box->center = center;
   box->halfDimension = halfDimension;
   return box;
}
```

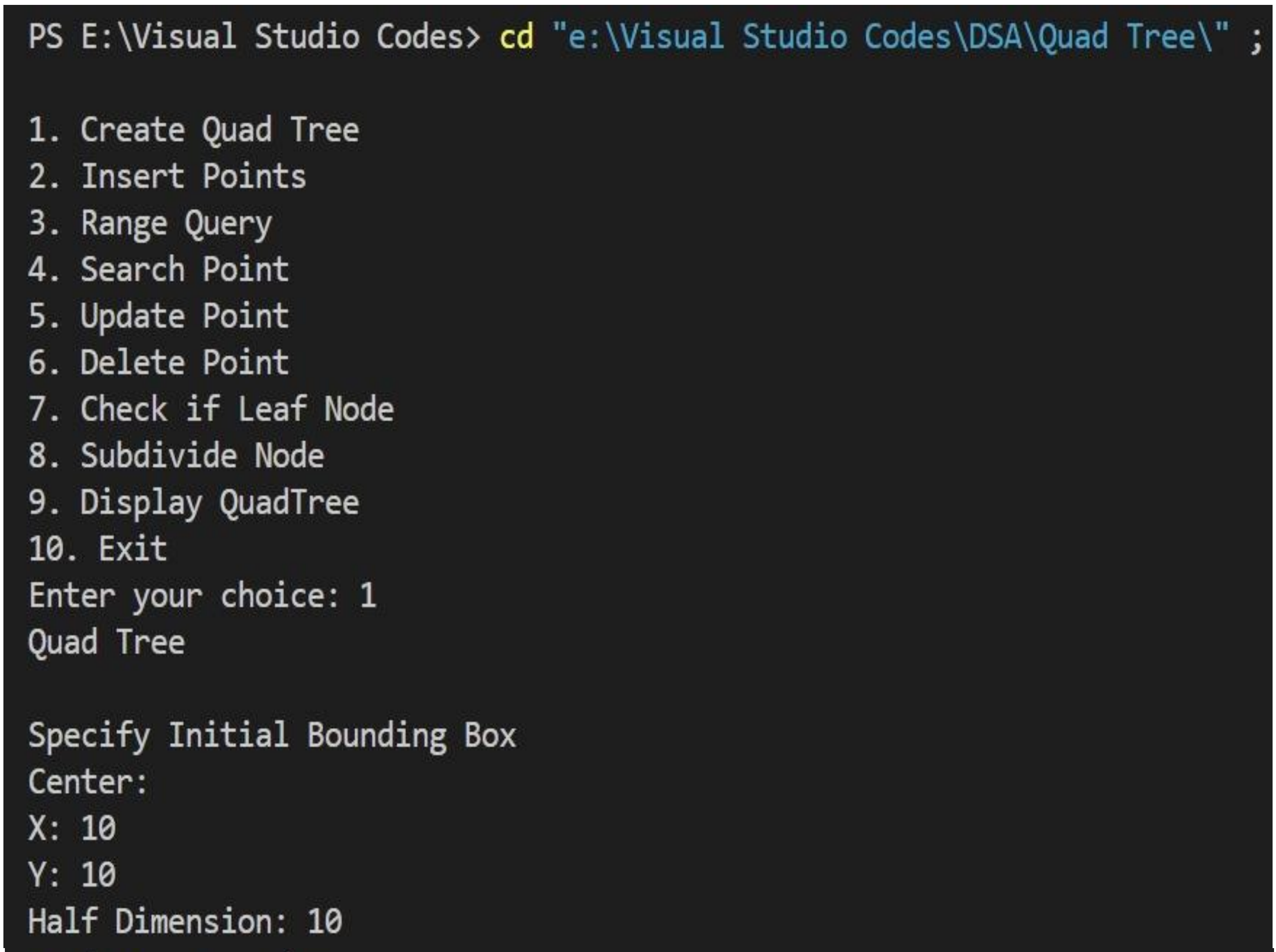
```
bool BoundryBox_cotains_point(BoundryBox *boundry, Point *point) {
   if (point->x < boundry->center->x - boundry->halfDimension || point->x >
boundry->center->x + boundry->halfDimension) {
      return false;
   }

   if (point->y < boundry->center->y - boundry->halfDimension || point->y >
boundry->center->y + boundry->halfDimension) {
      return false;
   }

   return true;
}

bool BoundryBox_intersects_BoundryBox(BoundryBox *self, BoundryBox *other) {
if (self->center->x + self->halfDimension > other->center->x – other
                                                   ->halfDimension) {
      if (self->center->x - self->halfDimension < other->center->x + other
                                                   ->halfDimension) {
         if (self->center->y + self->halfDimension > other->center->y – other
                                                   ->halfDimension) {
            if (self->center->y - self->halfDimension < other->center->y + other
                                                   ->halfDimension) {
               return true;
            }
         }
      }
   }
   return false;
}

void BoundryBox_print(BoundryBox *box) {
   printf("\n");
   printf("(%3.2f, %3.2f)---------(%3.2f, %3.2f)\n", box->center->x - box-
>halfDimension, box->center->y + box->halfDimension, box->center->x + box-
>halfDimension, box->center->y + box->halfDimension);
   printf("    |                    |\n");
   printf("    |                    |\n");
   printf("    |                    |\n");
   printf("    |                    |\n");
```

```
    printf("(%3.2f, %3.2f)---------(%3.2f, %3.2f)\n", box->center->x - box-
>halfDimension, box->center->y - box->halfDimension, box->center->x + box-
>halfDimension, box->center->y - box->halfDimension);
    printf("\n");
}

QuadTree *QuadTree_new(BoundryBox *boundry) {
    QuadTree *qt = (QuadTree *)malloc(sizeof(QuadTree));
    qt->NE = NULL;
    qt->NW = NULL;
    qt->SE = NULL;
    qt->SW = NULL;

    qt->boundry = boundry;

    qt->points = (Point **)malloc(sizeof(Point*) * QT_NODE_CAPACITY);

    for (size_t i = 0; i < QT_NODE_CAPACITY; i++)
    {
        qt->points[i] = NULL;
    }

    return qt;
}

bool Is_LeafNode(QuadTree quad){
    if(quad.NW == NULL)
        return true;
    return false;
}

int QuadTree_points_size(Point *points[]) {
    int i;
    for (i = 0; i < QT_NODE_CAPACITY; i++)
    {
        if (points[i] == NULL) {
            return i;
        }
    }
    return i;
}

QuadTree *QuadTree_subdivide(QuadTree *root) {
    if(!Is_LeafNode(*root)){
        printf("Already Subdivided\n");
        return root;
    }
    float halfDim = root->boundry->halfDimension / 2;
    // North West
    Point *nw_p = Point_new(root->boundry->center->x - halfDim, root->boundry-
>center->y + halfDim);
    root->NW = QuadTree_new(BoundryBox_new(nw_p, halfDim));

    // North East
    Point *ne_p = Point_new(root->boundry->center->x + halfDim, root->boundry-
>center->y + halfDim);
    root->NE = QuadTree_new(BoundryBox_new(ne_p, halfDim));

    // South West
    Point *sw_p = Point_new(root->boundry->center->x - halfDim, root->boundry-
>center->y - halfDim);
    root->SW = QuadTree_new(BoundryBox_new(sw_p, halfDim));

    // South East
    Point *se_p = Point_new(root->boundry->center->x + halfDim, root->boundry-
>center->y - halfDim);
    root->SE = QuadTree_new(BoundryBox_new(se_p, halfDim));

    return root;
}

bool QuadTree_insert(QuadTree *root, Point *point) {
    if (!BoundryBox_cotains_point(root->boundry, point)) {
        return false;
    }

    int points_size = QuadTree_points_size(root->points);

    if (points_size < QT_NODE_CAPACITY && root->NW == NULL) {
        printf("points size : %d\n",points_size);
        root->points[points_size] = point;
        return true;
    }

    if (root->NW == NULL) {
        QuadTree_subdivide(root);
    }

    if (QuadTree_insert(root->NW, point)) return true;
    if (QuadTree_insert(root->NE, point)) return true;
    if (QuadTree_insert(root->SW, point)) return true;
    if (QuadTree_insert(root->SE, point)) return true;

    return false;
}

Point **QuadTree_query_range(QuadTree *root, BoundryBox *range) {
    Point **result;
    result = (Point **)malloc(sizeof(Point *) * MAX_ARRAY_SIZE);

    int index = 0;
    for (int i = 0; i < MAX_ARRAY_SIZE; i++) {
        result[i] = NULL;
    }

    if (!BoundryBox_intersects_BoundryBox(root->boundry, range)) {
        return result;
    }

    int points_size = QuadTree_points_size(root->points);
    for (int i = 0; i < points_size; i++)
    {
        if (BoundryBox_cotains_point(range, root->points[i])) {
            result[index++] = root->points[i];
        }
    }

    if (root->NW == NULL) {
        return result;
    }

    int i;

    i = 0;
    Point **nw_r = QuadTree_query_range(root->NW, range);
    while (nw_r[i] != NULL && i < MAX_ARRAY_SIZE) {
        result[index++] = nw_r[i++];
    }

    i = 0;
    Point **ne_r = QuadTree_query_range(root->NE, range);
    while (ne_r[i] != NULL && i < MAX_ARRAY_SIZE) {
        result[index++] = ne_r[i++];
    }

    i = 0;
    Point **sw_r = QuadTree_query_range(root->SW, range);
    while (sw_r[i] != NULL && i < MAX_ARRAY_SIZE) {
        result[index++] = sw_r[i++];
    }

    i = 0;
    Point **se_r = QuadTree_query_range(root->SE, range);
    while (se_r[i] != NULL && i < MAX_ARRAY_SIZE) {
        result[index++] = se_r[i++];
    }

    return result;
}

bool Point_Search(QuadTree *root,Point * point){
    if(root == NULL){
        return false;
    }
    if (!BoundryBox_cotains_point(root->boundry,point)){
        // printf("Point not found\n");
        return false;
    }
    for(int i=0;i<QT_NODE_CAPACITY;i++){
        if(root->points[i]!=NULL && root->points[i]->x == point->x && root-
>points[i]->y == point->y){
            return true;
        }
    }


    if(root->NW!=NULL && BoundryBox_cotains_point(root->NW->boundry,point))
        return Point_Search(root->NW,point);
    else if(root->NE!=NULL && BoundryBox_cotains_point(root->NE-
>boundry,point))
        return Point_Search(root->NE,point);
    else if(root->SW!=NULL && BoundryBox_cotains_point(root->SW-
>boundry,point))
        return Point_Search(root->SW,point);
    else if( root->SE != NULL)
        return Point_Search(root->SE,point);
    else
        return false;
}

int Delete_Point(QuadTree *root,Point * point){
   if(root == NULL){
        printf("Point is not present\n");
        return false;
    }
    if (!BoundryBox_cotains_point(root->boundry,point)){
        printf("Point not found\n");
        return false;
    }
    for(int i=0;i<QT_NODE_CAPACITY;i++){
        if(root->points[i]!=NULL && root->points[i]->x == point->x && root-
>points[i]->y == point->y){
            Point_print(root->points[i]);
            printf("Deleted\n");
            free(root->points[i]);
            root->points[i]=NULL;
            //shifts the rest values by 1 place
            for(int j=i;j<QT_NODE_CAPACITY-1;j++){
                root->points[j]=root->points[j+1];
            root->points[j+1]=NULL;
         }
         return true;
      }
   }
   if(root->NW!=NULL && BoundryBox_cotains_point(root->NW->boundry,point))
      return Delete_Point(root->NW,point);
   else if(root->NE!=NULL && BoundryBox_cotains_point(root->NE-
>boundry,point))
      return Delete_Point(root->NE,point);
   else if(root->SW!=NULL && BoundryBox_cotains_point(root->SW-
>boundry,point))
      return Delete_Point(root->SW,point);
   else if( root->SE != NULL)
      return Delete_Point(root->SE,point);
   else{
      printf("Point is not present\n");
      return false;
   }
}

bool Point_Update(QuadTree *root,Point * point){
   if(root == NULL){
      printf("Point is not present\n");
      return false;
   }
   if (!BoundryBox_cotains_point(root->boundry,point)){
      printf("Point not found\n");
      return false;
   }
   for(int i=0;i<QT_NODE_CAPACITY;i++){
      if(root->points[i]!=NULL && root->points[i]->x == point->x && root-
>points[i]->y == point->y){
         printf("Enter the new val:");
         scanf("%d",&(root->points[i]->val));
         return true;
      }
   }
   if(root->NW!=NULL && BoundryBox_cotains_point(root->NW->boundry,point))
      Point_Update(root->NW,point);
   else if(root->NE!=NULL && BoundryBox_cotains_point(root->NE-
>boundry,point))
      Point_Update(root->NE,point);
   else if(root->SW!=NULL && BoundryBox_cotains_point(root->SW-
>boundry,point))
      Point_Update(root->SW,point);
   else if( root->SE != NULL)
      Point_Update(root->SE,point);
   else{
      printf("Point is not present\n");
      return false;
   }
}

void display_QuadTree(QuadTree *quad){
   Point **res = QuadTree_query_range(quad,quad->boundry);
   printf("The Points:\n");
   int j = 0;
   while (res[j] != NULL && j < MAX_ARRAY_SIZE) {
      Point_print(res[j]);
      j++;
   }
   printf("\n");
}

int main(){
   int choice;
   QuadTree *qt = NULL;

   printf("\n1. Create Quad Tree\n");
   printf("2. Insert Points\n");
   printf("3. Range Query\n");
   printf("4. Search Point\n");
   printf("5. Update Point\n");
   printf("6. Delete Point\n");
   printf("7. Check if Leaf Node\n");
   printf("8. Subdivide Node\n");
   printf("9. Display QuadTree\n");
   printf("10. Exit\n");
   while(true)
{
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice)
      {
         case 1: {
            printf("Quad Tree\n\n");
            printf("Specify Initial Bounding Box \n");
            printf("Center: \n");
            float x, y;
            printf("X: ");
            scanf("%f", &x);
            printf("Y: ");
            scanf("%f", &y);

            float hd;
            printf("Half Dimension: ");
            scanf("%f", &hd);

            Point *center = Point_new(x, y);
            BoundryBox *boundry = BoundryBox_new(center, hd);
            printf("Quad Tree Boundary\n");
            BoundryBox_print(boundry);

            qt = QuadTree_new(boundry);
            break;
         }
         case 2: {
            if (qt == NULL) {
               printf("Quad Tree not created yet. Create the Quad Tree first!\n");
               break;
            }
            int count;
            printf("Enter Number of Points to insert: ");
            scanf("%d", &count);
            int i = 1;
            while (count > 0) {
               printf("Point %d\n", i);
```

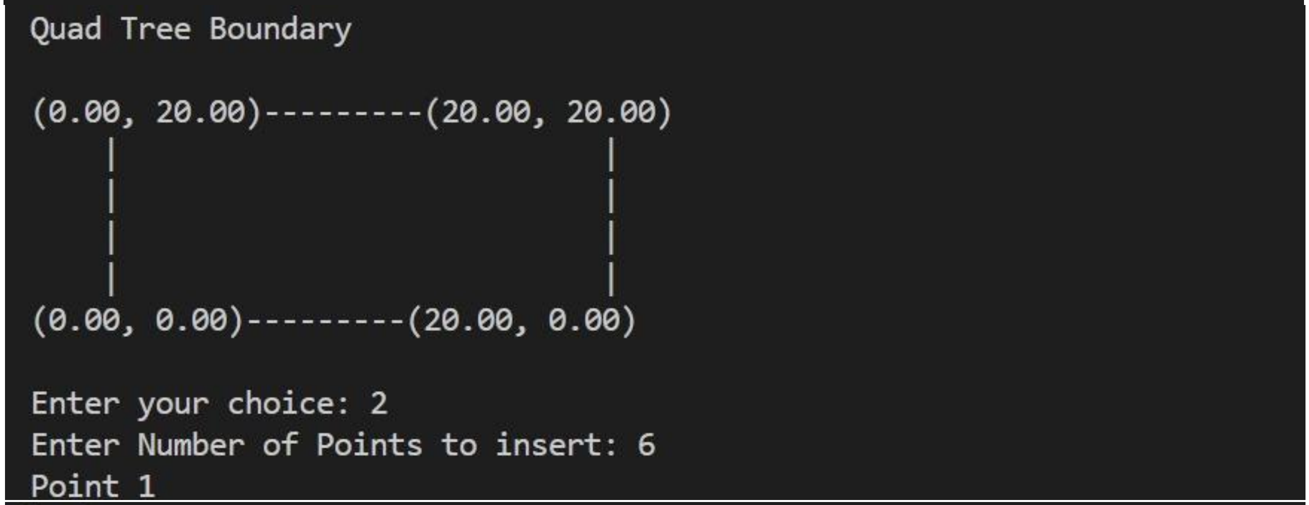
```
        float x, y;
        printf("X: ");
        scanf("%f", &x);
        printf("Y: ");
        scanf("%f", &y);
        Point *p = Point_new(x, y);
        printf("Val: ");
        scanf("%d", &p->val);

        if (Point_Search(qt, p)){
            printf("Point exists previously\n");
        }
        else if (!QuadTree_insert(qt, p)){
            printf("Point outside boundary. Not inserted\n");
        }
        else{
            printf("Point inserted successfully\n");
            i++;
            count--;
        }
    }
    break;
}
case 3: {
    if (qt == NULL) {
        printf("Quad Tree not created yet. Create the Quad Tree first!\n");
        break;
    }
    printf("\nRange Query\n\n");
    printf("Specify Axis Aligned Bounding Box for Range Query\n");
    printf("Center: \n");
    float x_r, y_r;
    printf("X: ");
    scanf("%f", &x_r);
    printf("Y: ");
    scanf("%f", &y_r);

    float hd_r;
    printf("Half Dimension: ");
    scanf("%f", &hd_r);
    // Manage the hd if provided hd exceeds boundary
    if(x_r+hd_r>(qt->boundry->center->x) + (qt->boundry->halfDimension)){
        hd_r=(qt->boundry->center->x) + (qt->boundry->halfDimension)-x_r;
    }if(y_r+hd_r>(qt->boundry->center->y) + (qt->boundry->halfDimension)){
        hd_r=(qt->boundry->center->y) + (qt->boundry->halfDimension)-y_r;
    }
    Point *center_r = Point_new(x_r, y_r);
    BoundryBox *boundry_r = BoundryBox_new(center_r, hd_r);
    printf("Range Search Boundary\n");
    BoundryBox_print(boundry_r);
    Point **res = QuadTree_query_range(qt, boundry_r);
    printf("Result\n");
    int j = 0;
    while (res[j] != NULL && j < MAX_ARRAY_SIZE) {
        Point_print(res[j]);
        j++;
    }
    break;
}
case 4: {
    float a, b;
    printf("Enter the point:\n");
    printf("X: ");
    scanf("%f", &a);
    printf("Y: ");
    scanf("%f", &b);
    Point *new_point = Point_new(a, b);
    if (Point_Search(qt, new_point)) {
        printf("Point found.\n");
    } else {
        printf("Point not found.\n");
    }
    break;
}
case 5: {
    float a, b;
    printf("Enter the point to update:\n");
    printf("X: ");
    scanf("%f", &a);
    printf("Y: ");
    scanf("%f", &b);
    Point *new_point = Point_new(a, b);
    if (Point_Update(qt, new_point)) {
        printf("Point updated successfully.\n");
    } else {
        printf("Point not found or update failed.\n");
    }
    break;
}
case 6: {
    float a, b;
    printf("Enter the point to delete:\n");
    printf("X: ");
    scanf("%f", &a);
    printf("Y: ");
    scanf("%f", &b);
    Point *new_point = Point_new(a, b);

    if (Delete_Point(qt, new_point)) {
        printf("Point deleted successfully.\n");
    } else {
        printf("Point not found or deletion failed.\n");
    }
    break;
}
case 7: {
    // Check if a node is a leaf node
    if (Is_LeafNode(*qt)) {
        printf("The current root node is a leaf node.\n");
    } else {
        printf("The current root node is not a leaf node.\n");
    }
    break;
}
case 8: {
    // Subdivide the node
    if (qt == NULL) {
        printf("Quad Tree not created yet. Create the Quad Tree first!\n");
        break;
```

```
            }
            if (QuadTree_subdivide(qt)) {
                printf("Node subdivided successfully.\n");
            } else {
                printf("Failed to subdivide the node.\n");
            }
            break;
        }
        case 9: {
            // Display the QuadTree
            if (qt == NULL) {
                printf("Quad Tree not created yet. Create the Quad Tree first!\n");
                break;
            }
            printf("Displaying QuadTree:\n");
            display_QuadTree(qt);
            break;
        }
        case 10: {
            exit(1);
            printf("Exiting program.\n");
            break;
        }
        default: {
            printf("Invalid choice. Please try again.\n");
            break;
        }
    }
  }
  return 0;
}
```

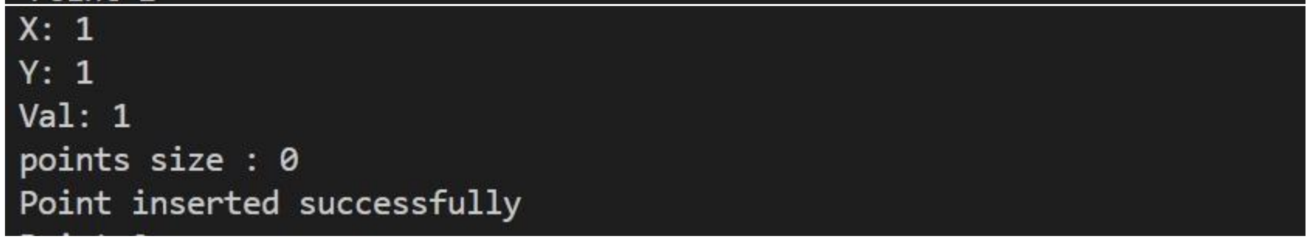
OUTPUT:



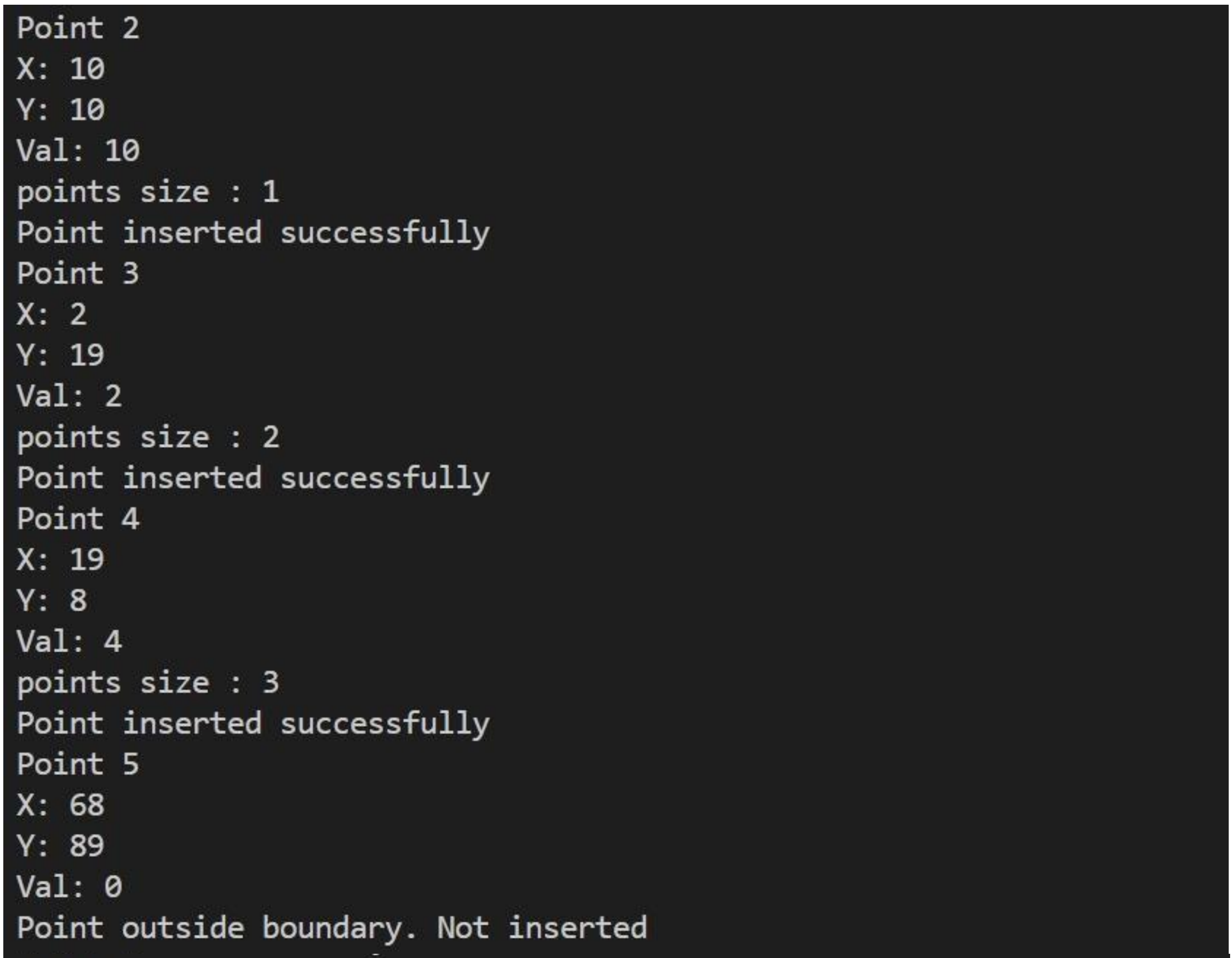
```
PS E:\Visual Studio Codes> cd "e:\Visual Studio Codes\DSA\Quad Tree\" ;

1. Create Quad Tree
2. Insert Points
3. Range Query
4. Search Point
5. Update Point
6. Delete Point
7. Check if Leaf Node
8. Subdivide Node
9. Display QuadTree
10. Exit
Enter your choice: 1
Quad Tree

Specify Initial Bounding Box
Center:
X: 10
Y: 10
Half Dimension: 10
```



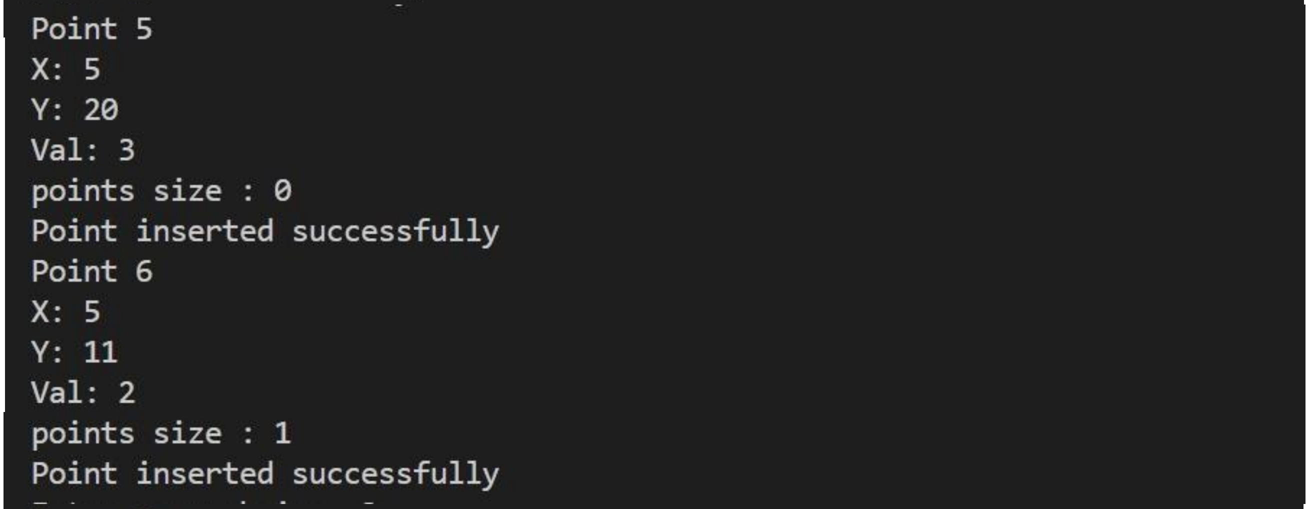
```
Quad Tree Boundary

(0.00, 20.00)---------(20.00, 20.00)
    |                           |
    |                           |
    |                           |
    |                           |
(0.00, 0.00)---------(20.00, 0.00)

Enter your choice: 2
Enter Number of Points to insert: 6
Point 1
```



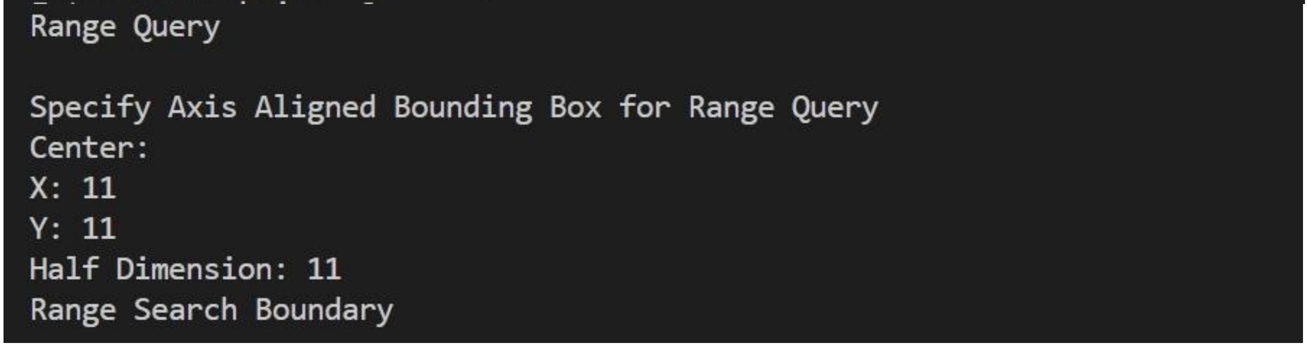
```
X: 1
Y: 1
Val: 1
points size : 0
Point inserted successfully
```



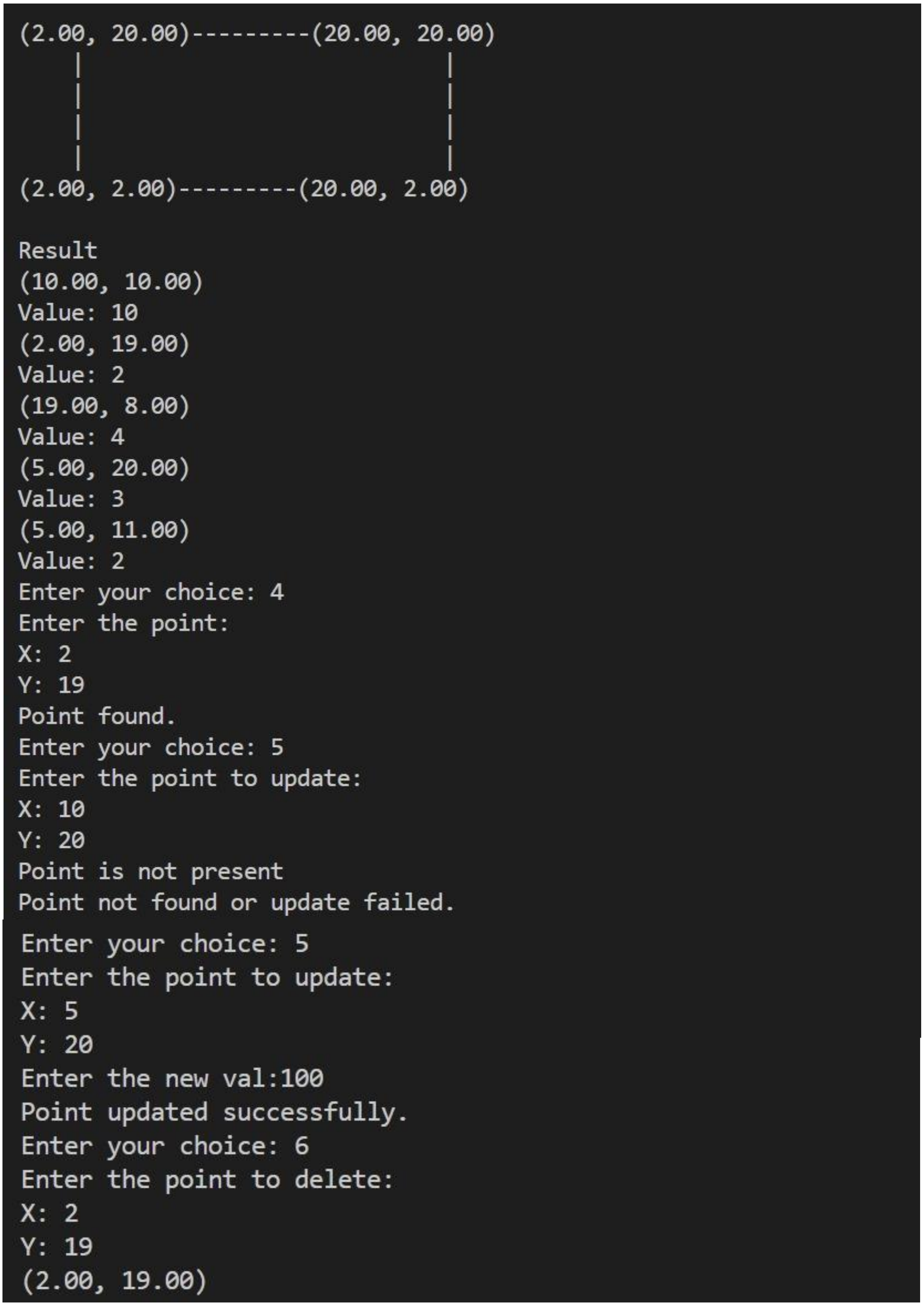
```
Point 2
X: 10
Y: 10
Val: 10
points size : 1
Point inserted successfully
Point 3
X: 2
Y: 19
Val: 2
points size : 2
Point inserted successfully
Point 4
X: 19
Y: 8
Val: 4
points size : 3
Point inserted successfully
Point 5
X: 68
Y: 89
Val: 0
Point outside boundary. Not inserted
```



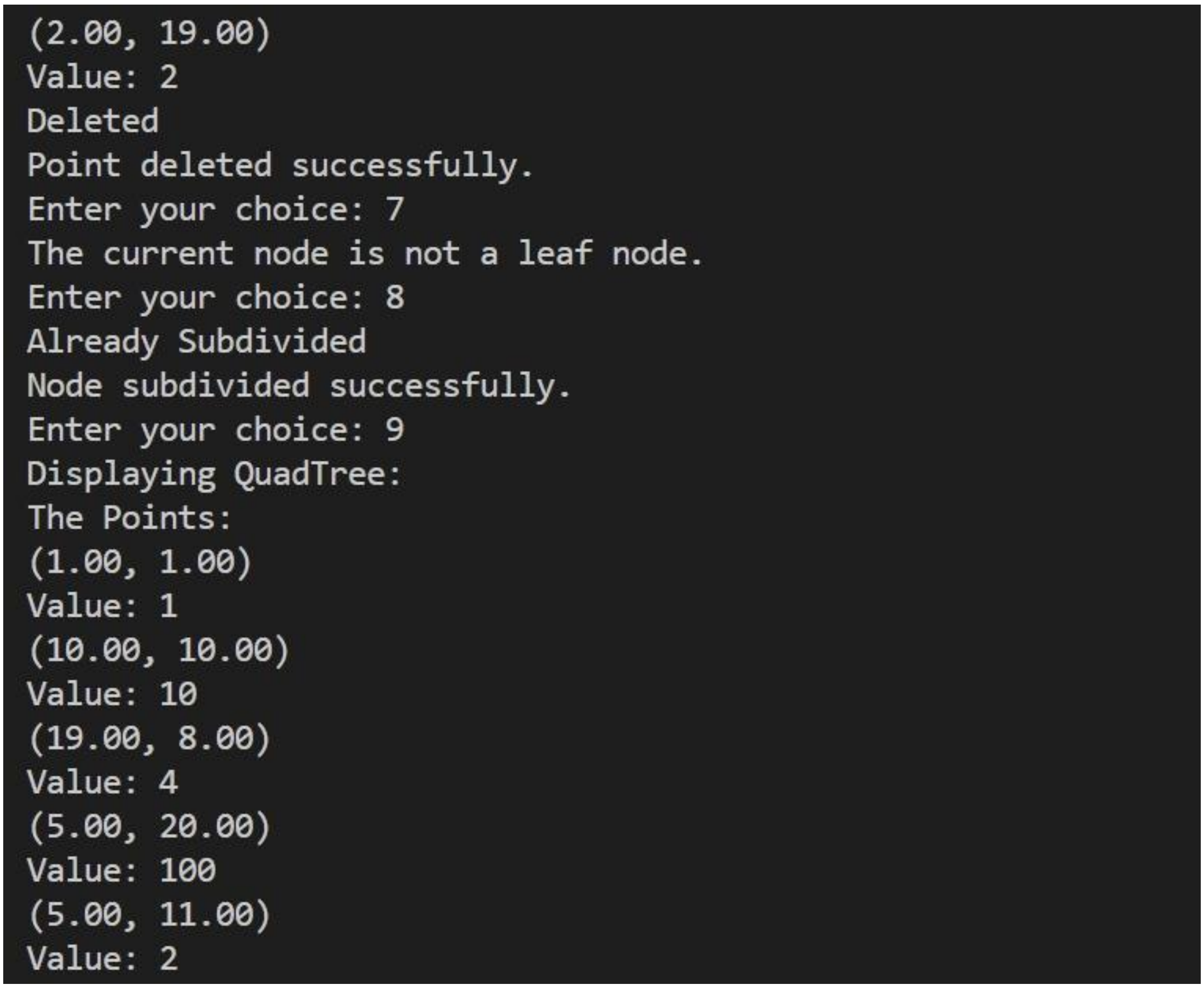
```
Point 5
X: 5
Y: 20
Val: 3
points size : 0
Point inserted successfully
Point 6
X: 5
Y: 11
Val: 2
points size : 1
Point inserted successfully
```



```
Range Query

Specify Axis Aligned Bounding Box for Range Query
Center:
X: 11
Y: 11
Half Dimension: 11
Range Search Boundary
```



```
(2.00, 20.00)---------(20.00, 20.00)
    |                            |
    |                            |
    |                            |
    |                            |
(2.00, 2.00)---------(20.00, 2.00)

Result
(10.00, 10.00)
Value: 10
(2.00, 19.00)
Value: 2
(19.00, 8.00)
Value: 4
(5.00, 20.00)
Value: 3
(5.00, 11.00)
Value: 2
Enter your choice: 4
Enter the point:
X: 2
Y: 19
Point found.
Enter your choice: 5
Enter the point to update:
X: 10
Y: 20
Point is not present
Point not found or update failed.
Enter your choice: 5
Enter the point to update:
X: 5
Y: 20
Enter the new val:100
Point updated successfully.
Enter your choice: 6
Enter the point to delete:
X: 2
Y: 19
(2.00, 19.00)
```



```
(2.00, 19.00)
Value: 2
Deleted
Point deleted successfully.
Enter your choice: 7
The current node is not a leaf node.
Enter your choice: 8
Already Subdivided
Node subdivided successfully.
Enter your choice: 9
Displaying QuadTree:
The Points:
(1.00, 1.00)
Value: 1
(10.00, 10.00)
Value: 10
(19.00, 8.00)
Value: 4
(5.00, 20.00)
Value: 100
(5.00, 11.00)
Value: 2
```

Peculiar Properties Which Helped:

1. Recursive Subdivision
A quadtree recursively divides a 2D space into four quadrants or regions (hence the name "quad"). Each node in the tree has up to four children, corresponding to the four quadrants: northeast, northwest, southeast, and southwest. Which allowed to store a 2D data efficiently in special representation while creating and inserting points in a quad tree

2. Adaptiveness
A quadtree can adapt to the density of objects in the space it represents. More subdivisions are made in areas with high object density, while areas with few or no objects remain less subdivided. This helped in Searching as to search a element we

might not need to search the whole tree for a element if we can find it as a less density part

3. Efficient Range Queries

Quadtrees are efficient for spatial queries like point searches, range searches. Particularly in two-dimensional geometric data, because their hierarchical structure allows for quick elimination of large sections of the space. So range Queries are much faster for Quad Trees.

5. Balanced and Unbalanced Quadtrees

Some quadtrees aim to maintain balance (all leaves at the same depth), while others may be unbalanced, allowing more subdivisions where needed. This depends on the specific quadtree variant being used. This dynamic structure frees us from the hastle of maintain balance like we need to do in 'AVL Trees' which makes Insertion Deletion easier.

9. Memory Efficiency

Quadtrees are memory-efficient for representing sparse data. If an area of space is empty or homogeneous, it will not be subdivided further, saving memory.

DISCUSSION:

This Quad Tree assignment demonstrates an efficient spatial data structure used to partition a two-dimensional space, enabling rapid querying and management of points within a region. The quadtree structure recursively subdivides the space into quadrants, organizing points to optimize operations like insertion, searching, and querying within a range.

Key components of your implementation include functions for inserting points into the appropriate quadrant, subdividing nodes when capacity is reached, and recursively querying ranges to retrieve points within a specified boundary. The recursive nature of quadtree operations allows for efficient space management, even when dealing with large datasets or complex spatial queries.

Your focus on ensuring that nodes are subdivided dynamically, only when necessary, highlights the efficiency of the structure, reducing memory usage and computational overhead. The querying function efficiently gathers points from relevant child nodes without having to check every point in the entire dataset. This is crucial for applications involving spatial queries, like graphics rendering, geographic information systems (GIS), and collision detection in gaming.

The challenge in your implementation lies in managing node capacity, subdivision logic, and ensuring that the boundary checks for both insertion and querying are accurate. Through careful attention to these details, your quadtree provides a balance between memory usage and query performance.