Lab 3 Design Document

1. Class Designs

1.1 Node Class

Description: This is my Node class. Which represents the nodes in a quadtree and stores the points if the node is a leaf node.
Constructor *Node:* I have two constructors for my array. Both constructors have parameters that include the x and y dimensions of the quadrant and a PointDictionary pointer. Except one of my constructors also includes a Point pointer that points to the head of the linked list. That constructor will be called when the children Nodes are created to ensure that all the Nodes point to the same head_point. The latter constructor will only be used for the first Node that is created when the Quadtree is made.
Destructor ~*Node():* My destructor is a recursive function that will repeatedly delete the children Nodes if the current Node is not a leaf node. If the node is a leaf node then it will delete the point_array, point_dictionary and head_point.
Private Member Variables:

- *double x0_coor, y0_coor, x1_coor, y1_coor:* stores the dimensions of that current node
- *int array_index:* stores the current index in the points_array, so we know if the array is full or not and we need to split the node
- *int capacity:* stores the max number of points allowed in a node (int m), which is compared with array_index
- *Node *top_left, *top_right, *bottom_left, *bottom_right:* pointers to the four children of a node, it is initially set as *nullptr* and new nodes are only made when that node is split
- *Point **points_array:* an array of Point* pointers, which will store all the points in a leaf node and will be *nullptr* for a non-leaf node
- *Point *head_point: stores the head point in a linked list that stores all the points found when calling the RANGE and SEARCH commands*
- *PointDictionary *pointdictionary:* a pointer to a PointDictionary object that stores all the points added to the Quadtree

Member Functions:

1. *bool checkInsert (double x, double y):* checks to see if the point that is trying to be inserted is within the quadtree and doesn't already exist before the insert function is called and the point is inserted.
2. *void insert (double x, double y, bool split_node):* A recursive function that will repeatedly call itself if the current node is not a leaf node and insert the point in the correct node. If the node is not a leaf node, it will check the position of the node and attempt to insert it in the correct children. If the child Node is still not a leaf node, then it will continue to attempt to insert it into the correct children until it reaches a leaf node. When it gets to a leaf node, it will check if there is still space in the array to store the point. If so, then the point is stored and that point is added to the point dictionary. If not, then the node will be split, and the point will be inserted in the correct child Node according to its coordinates.
3. *void split():*This function is called when a new point is inserted but the points array is full, so the Node needs to be split. New Nodes are created for each Child Node with their corresponding dimensions. Then in a loop, each point in the point array will be inserted into the correct child Node according to its coordinates. If that child Node becomes full or if that child Node is not a leaf node, then it will call split again and continue to do so until the point can be inserted. The points_array variable will become *nullptr* which indicates that the node doesn't have any Points stored and is not a leaf node anymore.
4. *int pointPos(double x, double y):* This function checks the position of a point and returns which quadrant it should be in
5. *point* range(double xr0, double yr0, double xr1, double yr1, Point* head):* This is a recursive function that will repeatedly call itself if the current node is not a leaf node. If the node is not a leaf node, it will take the dimensions given for the range and call the function again on the children Nodes that the range encompasses. If the child Node is also not a leaf node, it will repeatedly call the function on the children Nodes that it encompasses until it reaches a leaf node. When it does reach a leaf node, it will go through all the points in their point array and check to see if that point is within the range dimensions given. If it is, then it will be added to the linked list and the head_point variable is returned. This ensures that all the nodes will be adding their points to the same linked list so if the range spans multiple different leaf nodes, all the points are stored in one collective list to be outputted.
6. *void displayRange(double xr0, double yr0, double xr1, double yr1, Point* head):* This function will call the range function in order to get the list of points that are within the range dimensions and output the x and y coordinates of the points in the linked list. It will then delete the points in the list and set head_point to *nullptr* so it can be filled again when the command is called again.
7. *void search (double x, double y, double d):* This function calls the range function in order to get the list of points that are within the square dimensions and then checks the distance of each point in the list that it provides to see if it is within the distance required. It will then delete the points in the list and set head_point to *nullptr* so it can be filled again when the comman is called again.

1.2 Point Class

Description: This is my point class. It stores each individual point that is inserted into the Quadtree and their x and y coordinates.

Constructor *Point():* I have two constructors for my Point class. One constructor doesn't have any parameters at all. This is used when the array of Point objects is made to store the points in a Node but the points are not yet inserted. My other constructor has parameters that includes the x and y values of the Point. This is used when the point needs to be created and inserted into the array.

Destructor ~*Point():* My destructor will set the next_point pointer to *nullptr*, disconnecting the Points in the linked list.

Private Member Variables:

- *double x_coor, y_coor:* stores the x and y coordinates of the Point
- Point *next_point: Pointer that points to the next Point object in a linked list. It is initially not linked to any nodes.

Member Functions:

1. *double get_x_coor, get_y_coor:* getter functions to get the x and y coordinates of the Point
2. *Point *get_next():* getter function to get the next Point object in the linked list
3. *Void set_next():* setter function to set the next Point object in the linked list

1.3 PointDictionary Class

Description: My PointDictionary class is a dynamic array class that stores and keeps track of all the points that are inserted into the quadtree.

Constructor *PointDictionary():*

Destructor ~*PointDictionary():*

Private Member Variables:

- *int max_points:* This variable stores the max number of points that are allowed in each node. It will also be the initial capacity of the dynamic array and the amount that the array will increase by each time it is full.
- *int capacity:* This variable stores the total capacity of the array and it will increase as the array become full
- *int size:* This variable stores the current size of the array and the array will increase when the size is equal to the capacity.
- *Point **point_dictionary_array:* This is a dynamic array of Point pointers which will store all the pointers inserted into the quadtree

Member Functions:

1. *void addPoint (Point *point):* This function adds a new point into the dynamic array. If the array is full, the capacity is increased and the points are inserted into the array with the bigger capacity
2. *bool duplicate (double x, double y):* This function loops through the array and checks to the x and y coordinates in the parameters are equal to any points in the array. This makes sure that a new point to be inserted does not already exist in the quadtree.
3. *void nearest (double x, double y):* This function loops through the array and checks their distance from the x and y coordinates given in the parameters and outputs the Point that is the closest distance.
4. *void num():* This function outputs the size of the array+1, which also represents the total number of points that are stored in the quadtree.

2. Expected Runtime

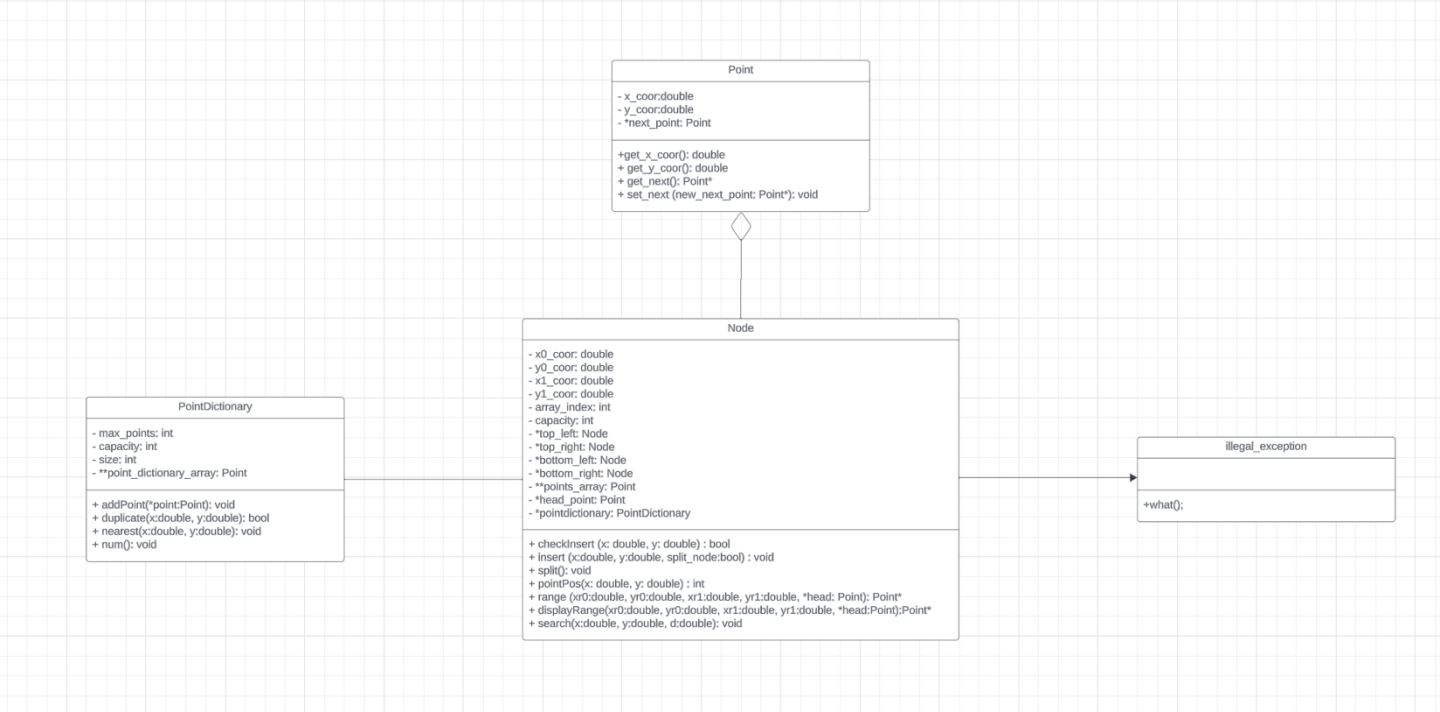
2.1 "INSERT" command

When the "INSERT" command is called, it will call my insert function which is a recursive function that continuously calls itself until it reaches a leaf node where it can either insert the point or must split the node and then insert it. Thus it will continue running, performing O(1) operation each time until it reaches the depth of the quadtree, which will be the leaf nodes. Thus, the runtime will be O(D).

2.2 "RANGE" command

When the "RANGE" command is called, it will call my range function which is a recursive function that continuously calls itself until it reaches a leaf node. If the range dimensions are only in one single leaf node, then multiple range functions don't have to be called in order to find all the points in the range dimensions. If only one range is called then it will perform a O(1) operation each time until it reaches the leaf node, which will be the depth of the quadtree. Thus, the runtime will be O(D).

3. UML Design


Point
- x_coor:double
- y_coor:double
- *next_point: Point
+get_x_coor(): double
+ get_y_coor(): double
+ get_next(): Point*
+ set_next (new_next_point: Point*): void
Node
- x0_coor: double
- y0_coor: double
- x1_coor: double
- y1_coor: double
- array_index: int
- capacity: int
- *top_left: Node
- *top_right: Node
- *bottom_left: Node
- *bottom_right: Node
- **points_array: Point
- *head_point: Point
- *pointdictionary: PointDictionary
+ checkInsert (x: double, y: double) : bool
+ insert (x:double, y:double, split_node:bool) : void
+ split(): void
+ pointPos(x: double, y: double) : int
+ range (xr0:double, yr0:double, xr1:double, yr1:double, *head: Point): Point*
+ displayRange(xr0:double, yr0:double, xr1:double, yr1:double, *head:Point):Point*
+ search(x:double, y:double, d:double): void
PointDictionary
- max_points: int
- capacity: int
- size: int
- **point_dictionary_array: Point
+ addPoint(*point:Point): void
+ duplicate(x:double, y:double): bool
+ nearest(x:double, y:double): void
+ num(): void
illegal_exception
+what();