**Report**

*Rohit\_Manral*

* **Program execution requirements**

**1.1** **Program environment**

The program was developed using Python version 3.7.4 (Latest version to-date). The program utilizes only inbuilt libraries of python such as sys, math and time. The program is divided into three files (Node.py, RTree.py, and main.py), those need to be in the same directory while running the program.

**1.2** **Input files and parameters**

The program requires two command line input from the user while running the *main.py* file in the Terminal / Command prompt i.e. dataset file (*dataset.txt*) and query file (*range\_query.txt*) paths with formatting mentioned in the problem statement. The following shows the placement of input files and usage of the program.

*python3 main.py {dataset}.txt {range\_query}.txt*

**1.3** **OS environments**

The program is compatible with almost all OS environments. For example, mac OS, Microsoft Windows, Linux, etc.

1. **Program documentation**

**2.1 Program organisation**

A brief, high level description of each file/class of my program as shown below.

|  |  |
| --- | --- |
| **Class/File Name** | **Description** |
| Node Class (Node.py)  [ This program (Node.py) refers to the code in "Core-Functions-R-Tree (Python)" on iLearn ] | It is used to extract perimeter, underflow & overflow conditions, and type (root &leaf) of every node in a RTree. |
| RTree Class (RTree.py)  [ This program (RTree.py) refers to the code in "Core-Functions-R-Tree (Python)" on iLearn ] | It is the main file/class for implementing RTree in python. As it first adds the data points in an MBR and finally returns the number of data points lying in an MBR after execution of various vital functions necessary for the stability of a RTree i.e. handle\_overflow, split, is\_covered, is\_intersect, choose\_subtree, and peri\_increase. |
| main (main.py) | Main file for running and testing RTree against sequential queries. |

* 1. **Function description**

**2.2.1 Function description of *main.py* file**

|  |  |
| --- | --- |
| **Function Name (parameters)** | **Description** |
| query\_sequential(points, query) | Function to perform sequential queries using the given files of data points and range queries.  Here, we will make a sequential search by traversing all 1,00,000 data points for every single range query given in the range\_query.txt file. Finally, we will get the number of data points lying inside a particular range query. |
| format\_data\_point (data) | Used for converting one data row of dataset.txt file into the format used in the program i.e. (x , y).  Here, we will split every row of dataset.txt file by whitespace and create two different dictionaries from it i.e. x and y. So that we can store the coordinate’s values of 1,00,000 data points. |
| format\_range\_query (data) | Used for converting one data row of range\_query.txt file into the format used in the program i.e. (x1 , x2 , y1 , y2).  Here, we are splitting every row of range\_query.txt file by whitespace and create four different dictionaries from it i.e. x1, x2, y1 and y2. So that we can store the values of lower-left and upper-right coordinates of 100 range queries. |
| main() | Main function that runs the files mentioned through command line arguments as mentioned above. Otherwise, exists with an error. |

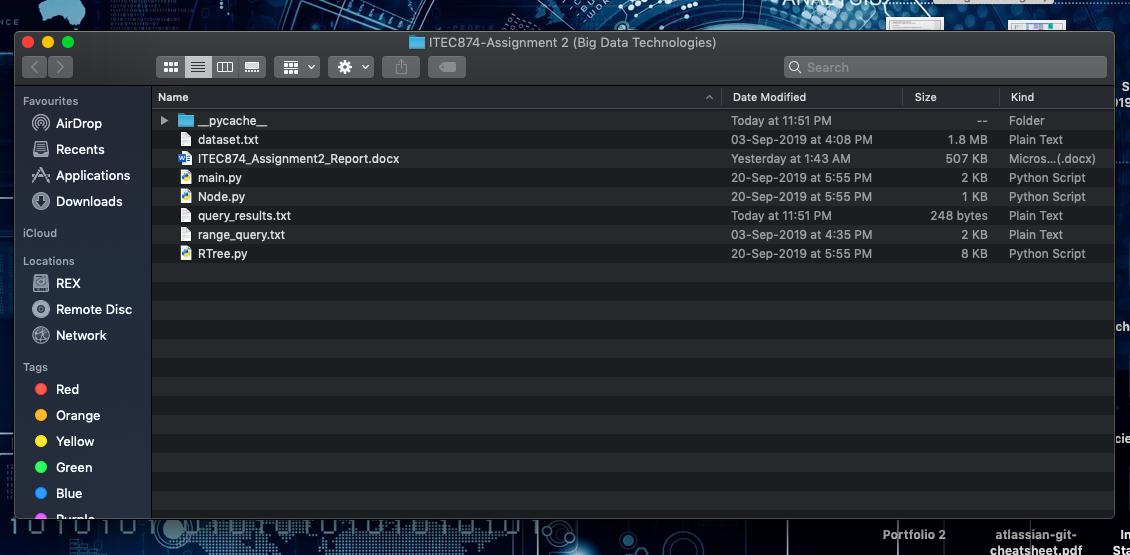
**2.2.1 Function description of *RTree.py* file**

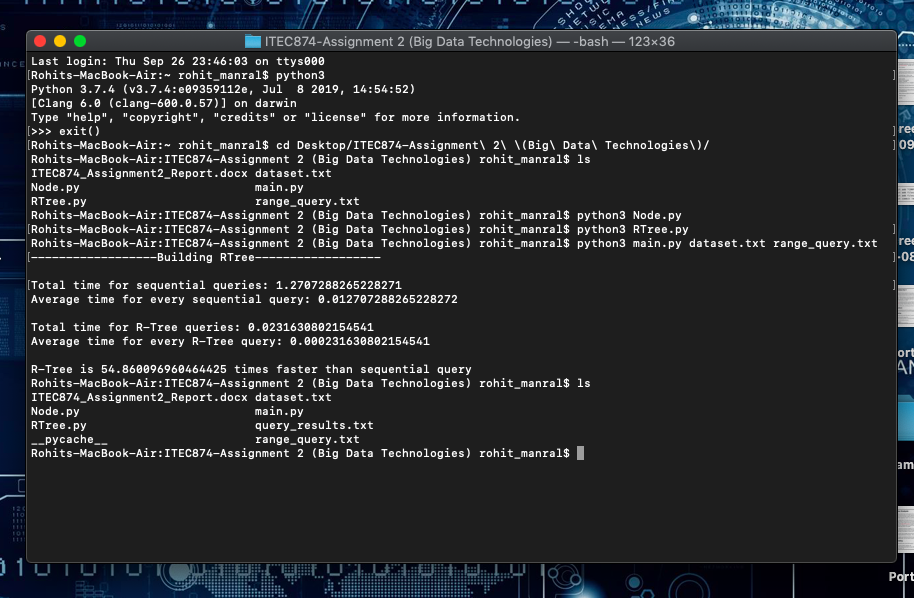
|  |  |
| --- | --- |
| **Function Name (parameters)** | **Description** |
| query\_rtree(self, node, query) | returns number of data points lying in an MBR (range query).  Here,   * if the passed node is a leaf node, then it will return number of data points stored at the node that are covered by range query using covering\_check(). * However, if the node is not leaf, then again run the query\_rtree() for each child node of the passed node, if it is intersecting with the given query. Finally, return the number of data points in a range query (MBR). |
| intersection\_check(self, node, query) | checks if two MBRs are intersecting or not. If the boundaries of any of the two MBRs will be colliding with each other, then we will find:   * |center1\_x - center2\_x| <= length1 / 2 + length2 / 2 and * |center1\_y - center2\_y| <= width1 / 2 + width2 / 2 |
| covering\_check(self, point, query) | checks that an MBR is covering a given data point or not.  If the value of x and y coordinates will lie between x1 & x2 and y1 & y2 respectively. Then, the query is covering the given data point, otherwise not. |
| insert(self, u, p) | Basically, used to insert a data point in a desired MBR.  Firstly, insert() will check if the target node is leaf or not.   * If the target node (u) is a leaf node, then it will directly add the data point and check for the overflow condition of the node. * However, if the target node (u) is not a leaf node, then it will choose an optimal subtree and again perform the insert(). Finally, update the MBR. |
| subtree\_select(self, u, p) | returns the child node whose MBR requires the minimum increase in the perimeter to cover a data point p.  Here,   * if the node u will be a leaf node, then it will simply return u. * However, if u is not a leaf node, then for each child node of u check is the minimum increase is greater than perimeter increase of the child or not. If yes, then return that child as the best child. |
| perimeter\_increased(self, node, p) | returns increase of perimeter by subtracting the original perimeter from the new perimeter.  Here,   * (max([x1, x2, p['x']]) - min([x1, x2, p['x']]) + max([y1, y2, p['y']]) - min([y1, y2, p['y']])) is the new perimeter. * node.perimeter\_mbr() is the original perimeter. |
| overflow\_handling(self, u) | Used to manage the overflow condition (when value of B exceeds 4) of an MBR by splitting and adding new child nodes to it.  Here,   * Firstly, we will split the overflowed node into 2 sub nodes using split\_node(). * If u is root node, create a new root with s1 & s2 as its' children and update the MBR. * If u is not root node, delete u, and set s1 and s2 as u's parent's new children, then again check the overflow condition for parent node. Finally, update the MBR. |
| split\_node(self, u) | It just splits an node (u) into two nodes.  Here,   * If u is a leaf node, it creates two different kinds of divides. * If it is an internal node, it creates four different kinds of divides. |
| child\_node\_add(self, node, child) | adds a new child node of an existing node. |
| data\_point\_add(self, node, data\_point) | Used to directly add data points in a node of RTree. |
| mbr\_updation(self, node) | Used for updating the MBR after performing some functions on it. |

**2.2.1 Function description of *Node.py* file**

|  |  |
| --- | --- |
| **Function Name (parameters)** | **Description** |
| perimeter\_mbr(self) | returns the perimeter (addition of length and breadth) of an MBR. |
| underflow\_check(self) | Checks that a node is in underflow condition i.e. B < 2.  Here,   * when node is leaf, it will return true, if length of data points would be smaller than smallest integer not less than B/2. * when node is not leaf, it will return true, if length of child nodes will be smaller than smallest integer not less than B/2. |
| overflow\_check(self) | Checks that a node is in overflow condition i.e. B > 4.  Here,   * when node is leaf, it will return true, if length of data points will be greater than B. * when node is not leaf, it will return true, if length of child nodes will be greater than B. |
| root\_node\_check(self) | Checks that a node is root node or not.  If a node will not have any parent node. |
| leaf\_node\_check (self) | Checks that a node is leaf node or not.  It will return true when length of the child nodes of a node is 0. |

**Screenshots**



**Figure:** Screenshot – Directory after running the code

**Figure:** Screenshot – Running Console

**Working of R-Tree**

**Step 1 : Selection of 1 Range Query and at least 10 Data Points**

**Step A : Selection of a range query**

I have selected 12th range query (37632, 38632, 15609, 16609) from “range\_query.txt” file.

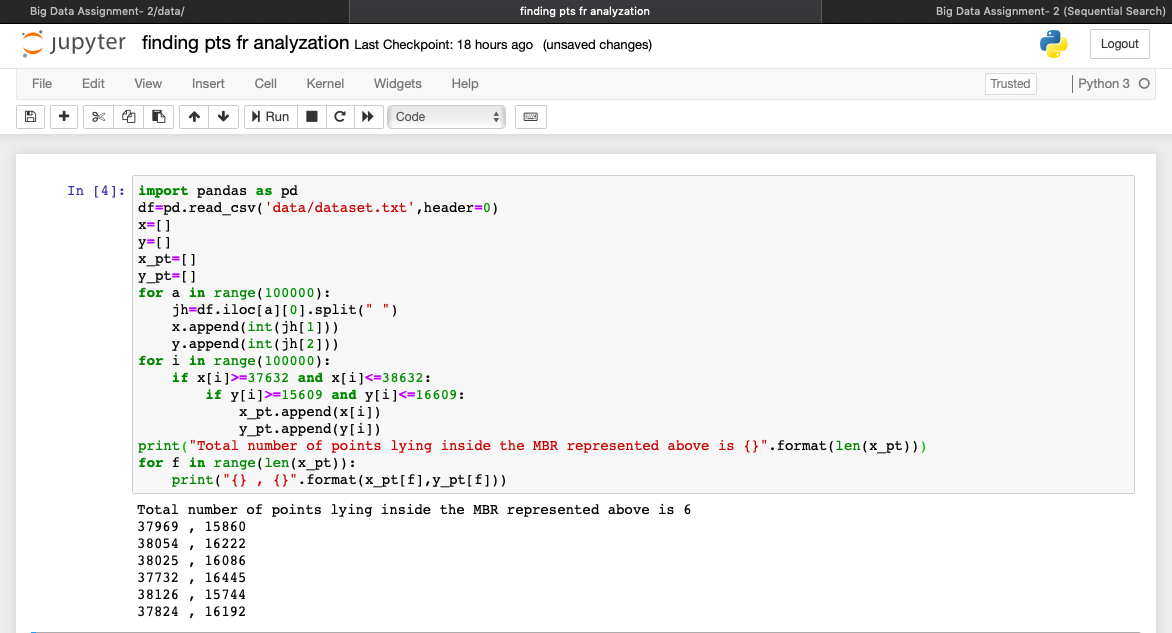
[ Format of a range query :- ( x\_lower , x\_upper , y\_lower , y\_upper) ]

**Representation of the selected Range Query on 2D axis**



**Step B : Selection of at least 10 Data Points**

I have created a python program to identify all the data points that lie in the range query mentioned above.



Here, I got total 6 data points as an output those definitely lie inside the selected Range Query and along with these data points, I will also select 4 other random data points from the dataset.txt file.

Therefore, the 10 selected data points from dataset.txt file are:

[ Format of a data point:- ( x , y) ]

(37969 , 15860)

(38054 , 16222)

(38025 , 16086)

(37732 , 16445)

(38126 , 15744)

(37824 , 16192)

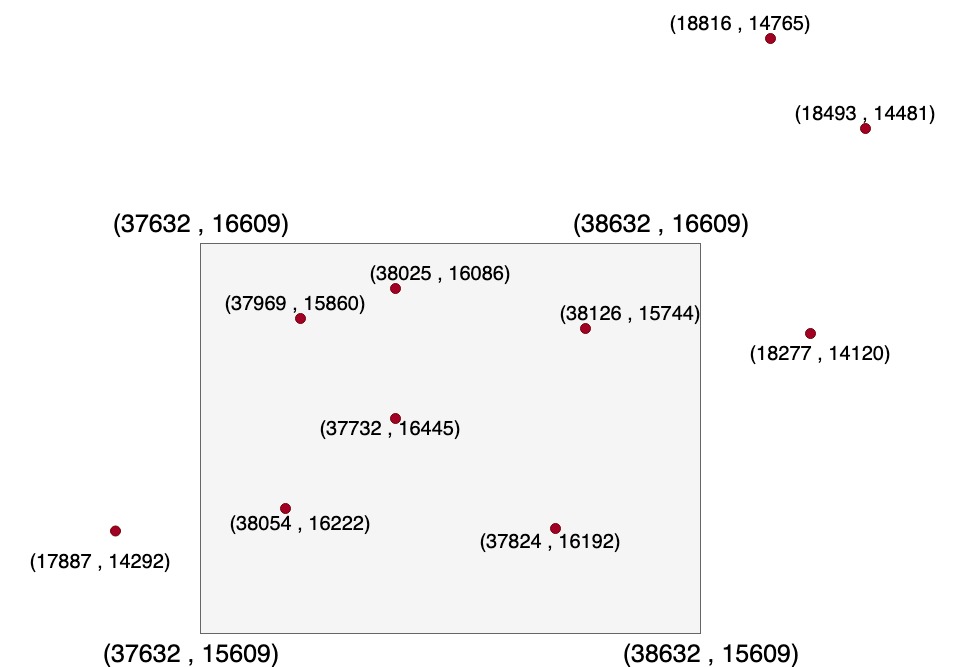
(18493 , 14481)

(17887 , 14292)

(18277 , 14120)

(18816 , 14765)

**Representation of the 10 selected data points with the previously selected Range Query on 2D axis**



So, after inserting these 10 data points in the selected range query(MBR) the figure will be something like that. The roughly designed figure above, is just to get an idea of the approximate positions of the data points and the range query on 2D axis.

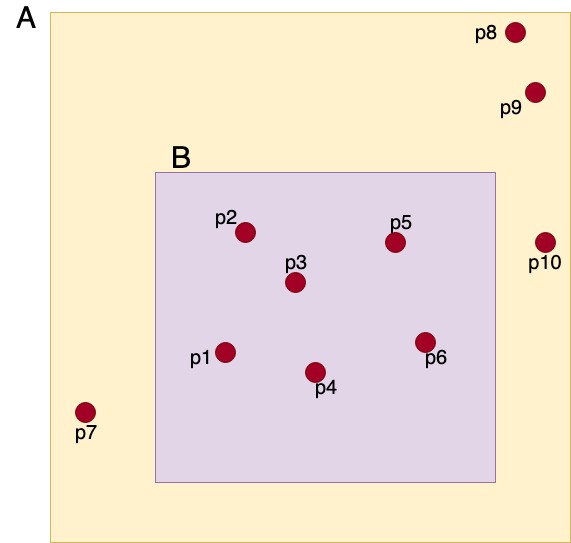
**Step 2 : RTree construction**

**Step A : insert(self, u, p)**

Firstly, insert() will check if the target node is leaf or not.

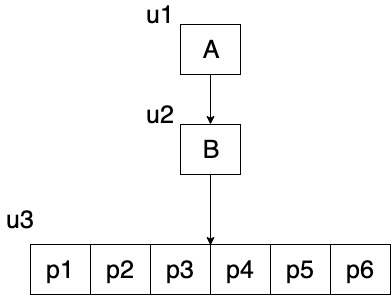
* If the target node (u) is a leaf node, then it will directly add the data point and check for the overflow condition of the node.
* However, if the target node (u) is not a leaf node, then it will choose an optimal subtree and again perform the insert(). Finally, update the MBR.

**Representation of MBR and Data points**:



Here, our main focus is on the MBR B, so we will perform some functions only on B to make it a stable MBR of RTree.

**R-Tree Representation of MBR and Data points**:



The value of B is fixed to 4 i.e. each MBR can have at most 4 data points. But, the MBR B (or node u3) has 6 data points inside it. So, the MBR is in overflow condition and now we will handle the overflow and make the MBR stable. For that, we will have to perform split\_node() and overflow\_handling().

**Step B : split\_node(self, u)**

Initially, we split the overflowed node u into 2 sub nodes into s1 and s2.

* u is a leaf node, create two different kinds of divides.
* However, if u is an internal node, create four different kinds of divides.

**Step C : overflow\_handling(self, u)**

Firstly, we will split the overflowed node into 2 sub nodes using split\_node().

* If u is root node, create a new root with s1 & s2 as its' children and update the MBR.
* If u is not root node, delete u, and set s1 and s2 as u's parent's new children, then again check the overflow condition for parent node. Finally, update the MBR.

Here,

m = no. of points in u = 6

Sort the points of u on y- dimension

for i = [0.4B] to m-[0.4B] = 2 to 4

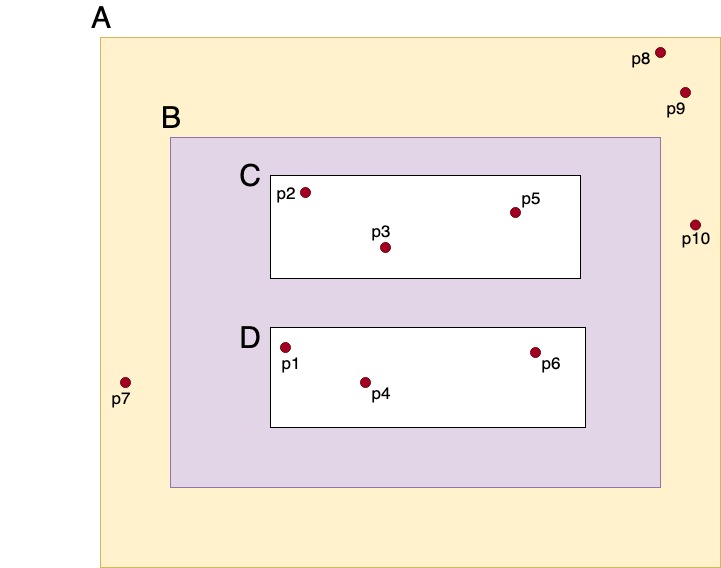
D= the set of the first i points in the list = {p4, p6, p1}

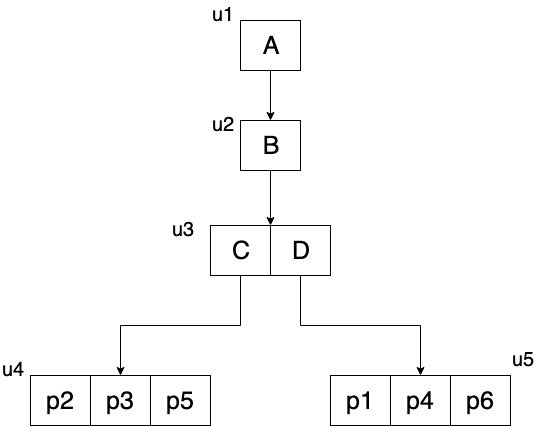
C= the set of the other i points in the list = {p3, p5, p2}

Calculating the perimeter sum of MBR(D) and MBR(C);

By recording the perimeter values, this is the best split so far.

Here, u3 (or B) is not a root node, delete u3, and set D and C as u3's parent's new children. Then, again checking, if the parent node is in underflow condition or not and finally updating the MBR.





Therefore, this is the final stable RTree.

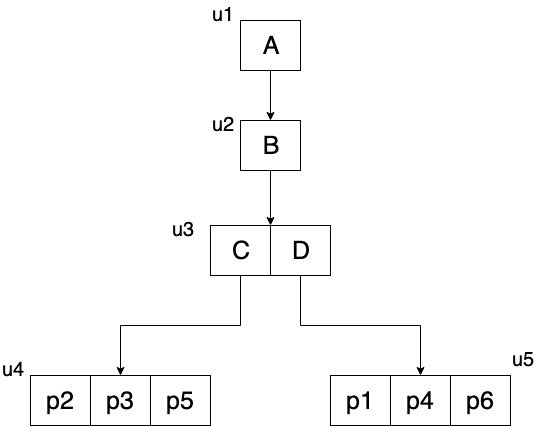
**Step 3 : Query Process**

After constructing the RTree, we will perform a query on it. The query process will return the number of data points that lie inside the range of a particular query.

**query\_rtree(self, node, query)**

Here,

* If the passed node is a leaf node, then it will return number of data points stored at the node that are covered by range query using covering\_check().
* If the node is not leaf, then again run the query\_rtree() for each child node of the passed node, if it is intersecting with the given query. Finally, return the number of data points in a range query (MBR).



**query\_rtree( u2 , query )**

Here, u2 is a non-leaf node, then we will again run the query\_rtree() for u3 (child node of u2) because it is intersecting with the given query. u3 is also a non-leaf node, then we will again run the query\_rtree() for child nodes of u3 i.e. u4 and u5. Both u4 and u5 are leaf nodes, then they will return number of data points (3+3=6) stored at both u4 and u5 that are covered by selected query.

Hence, the total number of data points lying inside node u2 (B) i.e. initially selected range query is 6, obtained by **query\_rtree( u2 , query ).**