

# COMP6210

## Big Data



**MACQUARIE**  
University  
SYDNEY • AUSTRALIA

### Report for Assignment 2

Submitted by:

Name	Student ID	Email
Anuj Adhikari	48547743	anuj.adhikari@students.mq.edu.au
Bharat Poudel	48687448	bharat.poudel@students.mq.edu.au
Gopal Datt Bhatt	48489131	gopaldatt.bhatt@students.mq.edu.au

## 1. Group Member Information

Name	Student ID	Email:	Assigned Task
Golap Datt Bhatt	48489131	gopaldatt.bhatt@students.mq.edu.au	Task 1
Anuj Adhikari	48547743	anuj.adhikari@students.mq.edu.au	Task 2
Bharat Poudel	48687448	Bharat.poudel@students.mq.edu.au	Project Report

## 2. Program Execution Requirements

### 2.1 Program Environment (e.g., OS, database, CPU, etc.)

#### i. Operating System Compatibility:

The program is designed to run on **Windows, macOS** and platforms that support **Python version 3.6 or higher**. Both Task were developed and tested on **macOS using Spyder IDE**.

#### ii. Python Version:

The code requires **Python 3.6 or newer** so that it can make use of built-in libraries available in the python version and above.

#### iii. Required Libraries:

The code uses **time, sys, os, and heapq**, which are pre-installed libraries in python. No external dependencies that present that require manual installations.

#### iv. Hardware Recommendations:

Since the dataset is very large, the hardware requirement for smooth performance are as follows:

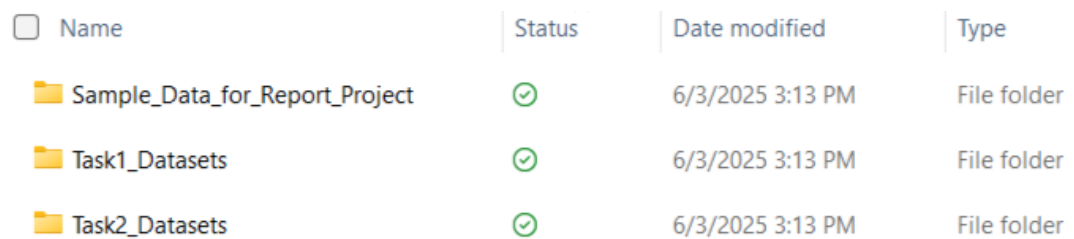
- **4 GB of RAM and above**
- **Any Dual-core processor and above.**







#### v. Development Environment:

Both of the Task 1 and Task 2 were developed in different IDE/environment. Task 1 was developed in VS code while Task 2 in Spyder IDE which means both tasks were extensively tested and debugged in these respective IDE, but the program executes flawlessly in other IDEs such as PyCharm, or command-line etc. which supports python.

## 2.2 Input Files and Parameters (directory settings and other parameters)

To ensure correct execution, the project requires specific input files organized within a structured directory. Essential parameters should also be set within the code for accurate processing. Below is an overview of the directory structure for both Task 1 and Task 2, followed by a screenshot illustrating this setup.



<input type="checkbox"/> Name	Status	Date modified	Type
 Sample_Data_for_Report_Project		6/3/2025 3:13 PM	File folder
 Task1_Datasets		6/3/2025 3:13 PM	File folder
 Task2_Datasets		6/3/2025 3:13 PM	File folder

**Figure 1: Screenshot of Main Directory Structure of Tasks**

To ensure correct execution, both Task 1 and Task 2 rely on specific input files located within a structured directory setup. Below are details on the directory structure for each task, along with screenshots for easy reference.

### Task 1: Nearest Neighbor Search

For Task 1, the input files, output file, and code file are in the following structure. This setup includes the dataset, query file, and the necessary Python script to execute the nearest neighbor search algorithms.

- Query\_points.txt contains the user location in the format [ID, longitude (x), and latitude (y)].
- Parking\_dataset.txt records parking lot locations in the format [ID, longitude (x), and latitude (y)].
- Restaurant\_dataset.txt Contains restaurant locations in 2D format in the format [ID, longitude (x), and latitude (y)].
- Shop\_dataset.txt stores shop locations in the format [ID, longitude (x), and latitude (y)].
- task1\_NN\_search.py contains the code to implement NN search algorithm using all three methods.
- output\_restaurant.txt file contains the output queries for three methods with total and average execution time.

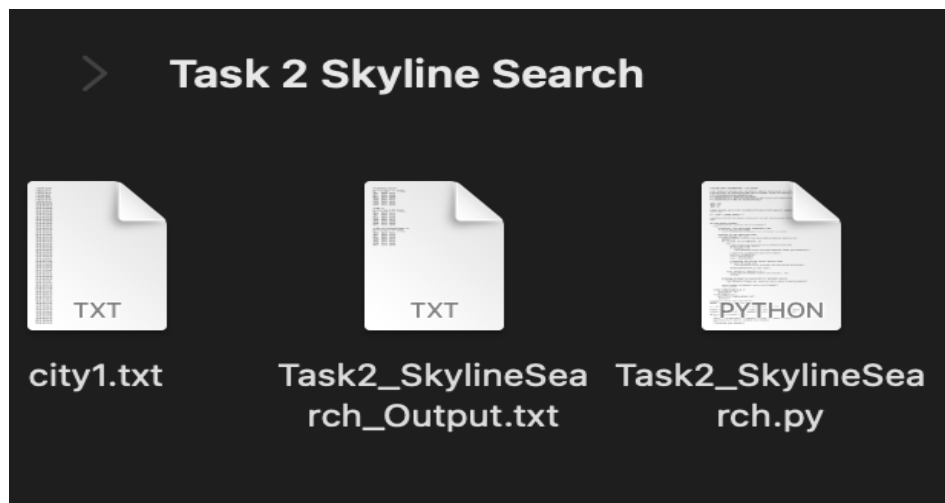


**Figure 2: Directory Structure for Task 1 – NN Search**

## Task 2: Skyline Search

For Task 2, the input file, output file, and code file are in the following structure. This setup includes the dataset and the necessary Python script to execute the skyline search algorithms. The screenshot below provides a visual representation of the directory structure for Task 2.

- Input file for city datasets is in city1.txt.
- Output files are in Task2\_SkylineSearch\_Output.txt.
- Python code is located in Task2\_SkylineSearch.py.



**Figure 3: Directory Structure for Task 2 - Skyline Search**

## Task 3: Project Report

Project Report Compilation Task 3 involves compiling the final project report, which includes descriptions, diagrams, requirements, and an analysis of results

for both Task 1 and Task 2.

## **2.3 Additional Requirements**

### **1. Input File Format Compliance**

All dataset and query files for both the Nearest Neighbor Search (Task 1) and Skyline Search (Task 2) must strictly follow the specified formats. Particular attention should be given to Task 1 query files to ensure attribute alignment with the main dataset, avoiding mismatches during search operations.

### **2. Output Storage Management**

Designated output folders (Task1outputs and Task2outputs) must be created with proper write permissions. This ensures smooth saving of results and prevents file write errors caused by access restrictions.

### **3. Robust Error Monitoring**

Integrated error handling captures common issues like file access problems or invalid input formats. A debug mode is also available to track program execution step-by-step, making troubleshooting and performance analysis more efficient.

### **4. Optimized Resource Utilization**

To prevent memory overuse on low-resource systems, batch processing of queries is advised. Smaller processing loads per cycle ensure better CPU and memory management without compromising accuracy.

### **5. Pre-Execution Quality Checks**

Before running the full system, trial runs were conducted using reduced-size datasets. This validated directory setup, data integrity, and algorithm behavior—especially for Skyline Search—ensuring consistent and accurate outputs under test conditions.

## **3. Program Documentation**

### **3.1 Program Organization**

<b>File Name</b>	<b>Description (detailed information)</b>
<b>Task 1</b>	Nearest Neighbor Search: Identifies the closest data points to a set of query points using three methods: Sequential Scan, BF Algorithm (Best First with R tree), and Divide-and-Conquer. Each method is structured to process and output results for nearest neighbor searches.
<b>restaurant_dataset.txt</b>	Task 1 is performed using the restaurant dataset which contains the location of restaurants in the format [ID, longitude (x), and latitude (y)].
<b>task1_NN_search.py</b>	This is the main program file containing functions, class and main program which implements NN search algorithm using sequential scan, BF search using custom rtree and BF search with Divide-and-Conquer.
<b>output_restaurant.txt</b>	This is the output file containing 200 output queries for each method which shows the nearest restaurant for different query points. The file gives ID and location coordinates for each query along with the total and average time of execution for each method.
<b>Task 2</b>	Skyline Search: Determines optimal points (skyline points) from a dataset based on criteria such as minimizing cost and maximizing size, using three methods: Sequential Scan, BBS Algorithm (Branch and Bound with R-tree), and Divide-and-Conquer.
<b>Task2_SkylineSearch.py</b>	Contains the main functions for Task 2. Implements Sequential Scan, BBS Algorithm with R-tree, and Divide-and-Conquer approaches to identify skyline points in the dataset.
<b>city1.txt</b>	Dataset containing city points with attributes like cost and size, used as the input data for Task 2 skyline calculations.
<b>Task2_SkylineSearch_Output.txt.</b>	Output file that stores results from each skyline search algorithm (Sequential Scan, BBS, and Divide-and-Conquer). It includes skyline points' IDs, attributes, and execution times for performance comparison.
<b>Task 3</b>	Project Report Compilation: Compiles the final report with descriptions, diagrams, requirements, and result analysis for Tasks 1 and 2.
<b>report.docx/pdf</b>	Document file for the final project report, stored in the main directory. Includes a comprehensive overview of Tasks 1 and 2 with all relevant findings and performance comparisons.

### 3.2 Function Description

Function Name (parameters)	Description (detailed information)
<b>TASK 1 : NN Search</b>	
<b>read_points(file_path)</b>	Reads a file containing spatial points. Each line is parsed into a tuple (id, x, y) and returned as a list. Skips lines with missing or malformed data.
<b>euclidean_distance_sq(x1, y1, x2, y2)</b>	Computes squared Euclidean distance between two 2D points. Square root is omitted for speed since it's not needed for comparison.
<b>sequential_scan(dataset, queries)</b>	Performs brute-force nearest neighbor search: For each query point, it scans all dataset points to find the closest one based on Euclidean distance.
<b>RTreeNode(is_leaf) (Class)</b>	Represents a node in the R-tree. Contains entries (points or

	<p>children) and an MBR</p> <p>(minimum bounding rectangle).</p> <p>Uses <code>__slots__</code> for memory efficiency.</p>
<b>get_mbr(points)</b>	<p>Computes the Minimum Bounding Rectangle for a list of points: returns (min_x, min_y, max_x, max_y).</p>
<b>enlarge_mbr(mbr1, mbr2)</b>	<p>Merges two MBRs and returns the smallest rectangle that contains both. Used to expand node MBRs during tree building.</p>
<b>build_rtree(points, max_entries=12)</b>	<p>Constructs an R-tree spatial index from raw data points.</p> <p>Groups points into leaf nodes, then builds upper layers using <code>build_level</code>.</p>
<b>build_level(entries, is_leaf)</b>	<p>Groups entries into fixed-size nodes for the R-tree. Computes and assigns MBRs to each node. Used recursively to build</p>



	tree layers.
<b>mbr_distance_sq(mbr, x, y)</b>	Computes the squared distance from a point (x, y) to the closest point within an MBR. Returns 0 if the point is inside the MBR.
<b>best_first_search_rtree(rtree, query)</b>	Performs nearest neighbor search using best-first traversal of the R-tree. Uses a priority queue (min-heap) to explore the closest MBRs first and prune distant branches.
<b>bf_rtree_search(dataset, queries)</b>	Wrapper that builds an R-tree from the dataset and performs best-first nearest neighbor search for all query points. Returns results and timing.
<b>bf_divide_and_conquer(dataset, queries)</b>	Optimized search: splits dataset into left/right halves based on x-coordinate, builds two R-trees, and compares both results

	for each query.
<b>write_output(all_results, output_path)</b>	Writes all algorithm results to a text file. Includes query results, total time, and average query time for each method.
<b>if __name__ == "__main__" (Script Entry)</b>	Main script logic: prompts user for dataset, loads data, runs all algorithms (sequential, R-tree, divide-and-conquer), and saves results.
<b>TASK 2 : Skyline Search</b>	
<b>load_dataset(filename)</b>	Loads , validates dataset from text file and parses each line into (ID, cost, size) tuples while validating data types and finally returns dataset as list of tuples with inclusion of error handling.
<b>is_skyline_point(point, skyline)</b>	Checks to see if a point belongs in skyline by testing domination against other skyline points.

	<p>The logic where point p dominates q if p has lower/equal cost AND higher/equal size is also implemented. It also iterates through current skyline for non-domination check. The result is false if point is dominated else true.</p>
<b>skyline_sequential(dataset)</b>	<p>Implements <math>O(n^2)</math> sequential scan algorithm where every point is compared against other using nested loops and returns a list of non-dominated points.</p>
<b>Node.__init__(self, entries, is_leaf=True)</b>	<p>This is the constructor for R-tree node class. It stores entries which are data points and also sets node type flag. This calculates MBR.</p>
<b>Node.compute_mbr(self)</b>	<p>Calculates Minimum Bounding Rectangle (MBR) by using spatial boundaries of entries.</p>

	<p>The cost and size values are taken from provided data points for leaf nodes or combines child MBRs for internal nodes. It returns (min_cost, max_cost, min_size, max_size).</p>
<p><b>Node.mindist(self)</b></p>	<p>Calculates minimum distance heuristic for priority queue ordering by performing sum on minimum cost and size whole also estimating the likelihood of containing skyline points where lower values indicate higher processing priority. It is used by heapq for node expansion order in BFS.</p>
<p><b>Node.__lt__(self, other)</b></p>	<p>Enables Node comparison for heapq operations by implementing less-than operator. It compares mindist() values for priority ordering. It makes the automatic processing</p>

	<p>of nodes in increasing mindist order for optimal search possible.</p>
<b>bulk_load(dataset, max_entries=100)</b>	<p>Constructs balanced R-tree where bulk loading method is used. This function also sorts dataset by cost for spatial locality and chunks into leaf nodes. It recursively builds tree levels bottom to up by grouping nodes until a single root remains.</p>
<b>mbr_dominated(mbr, skyline)</b>	<p>Determines if entire MBR can be skipped by checking skyline point domination and verifies if any skyline point has cost <math>\leq</math> MBR minimum cost AND size <math>\geq</math> MBR maximum size. It returns true if region can be skipped.</p>
<b>skyline_bbs(dataset)</b>	<p>Implements Branch and Bound Skyline algorithm using R-tree</p>

	<p>spatial indexing. It builds R-tree, initializes priority queue, and performs best-first traversal with mindist approach. It returns skyline points.</p>
<p><b>skyline_bbs_divide_and_conquer(dataset)</b></p>	<p>Implements divide-and-conquer BBS variant splitting dataset into subspaces for separate processing. Sorting via cost is done and splitting is done at median into left and right halves. It applies BBS to each half and merges with dominance.</p>
<p><b>run_all_algorithms()</b></p>	<p>It executes skyline algorithms with evaluation metrics and error handling. It measures the execution times also has try-catch blocks.</p>
<p><b>write_results_to_file(results, filename="Task2_SkylineSearch_Output.txt")</b></p>	<p>The main result of the algorithms implemented are written into a file named Task2_SkylineSearch_Output I.</p>

	text format.
--	--------------

#### 4. Analyzing BF Algorithm based NN Search

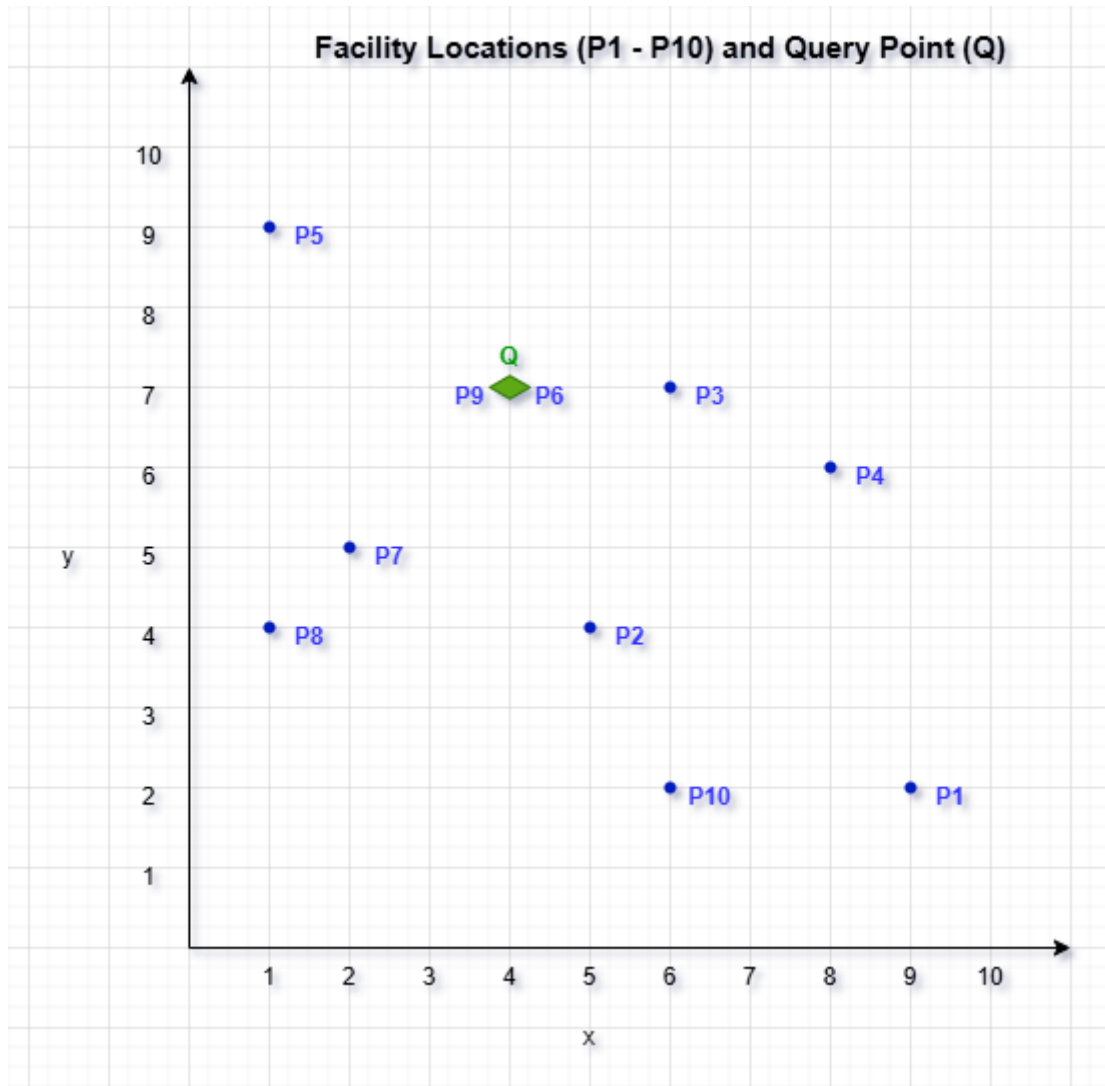
The objective of Task 1 is to identify the nearest neighbor to a query point (4, 7) from a given set of facility location points using an efficient spatial data structure, the R-tree, and the Best-First (BF) algorithm.

Here's the table for the **Facility Location** dataset, including the points and query points.

ID	1	2	3	4	5	6	7	8	9	10
X	9	5	6	8	1	4	2	1	4	6
Y	2	4	7	6	9	7	5	4	7	2

Query Point		
ID	X	Y
1	4	7

This table is organized to list each facility location with its ID, X, and Y coordinates. The query point is listed separately to clearly indicate it as the point of reference for the nearest neighbor search.



**Figure 4 : Visualization of Facility Locations and Query Point (Q) on an X-Y Grid**

The figure displays the query point (Q) and the spatial distribution of facility locations (P1–P10) on an X–Y coordinate grid. The query point (Q) is indicated in red for convenience, and each facility location is represented by a blue dot with its unique ID. A clear layout of each point's location on the grid is provided by the X and Y axes, which run from 1 to 10. The R-tree is constructed using this visual arrangement as a starting point because it shows the spatial relationships between points, which will guide the grouping of points into Minimum Bounding Rectangles (MBRs) for an effective nearest neighbor search.



## 4.1 R-Tree Construction

### Step by Step R-Tree Construction

#### Step 1: Insert Point P1

- This is the first point being inserted into an empty R-tree.
- P1 is inserted into a newly created leaf node u1.

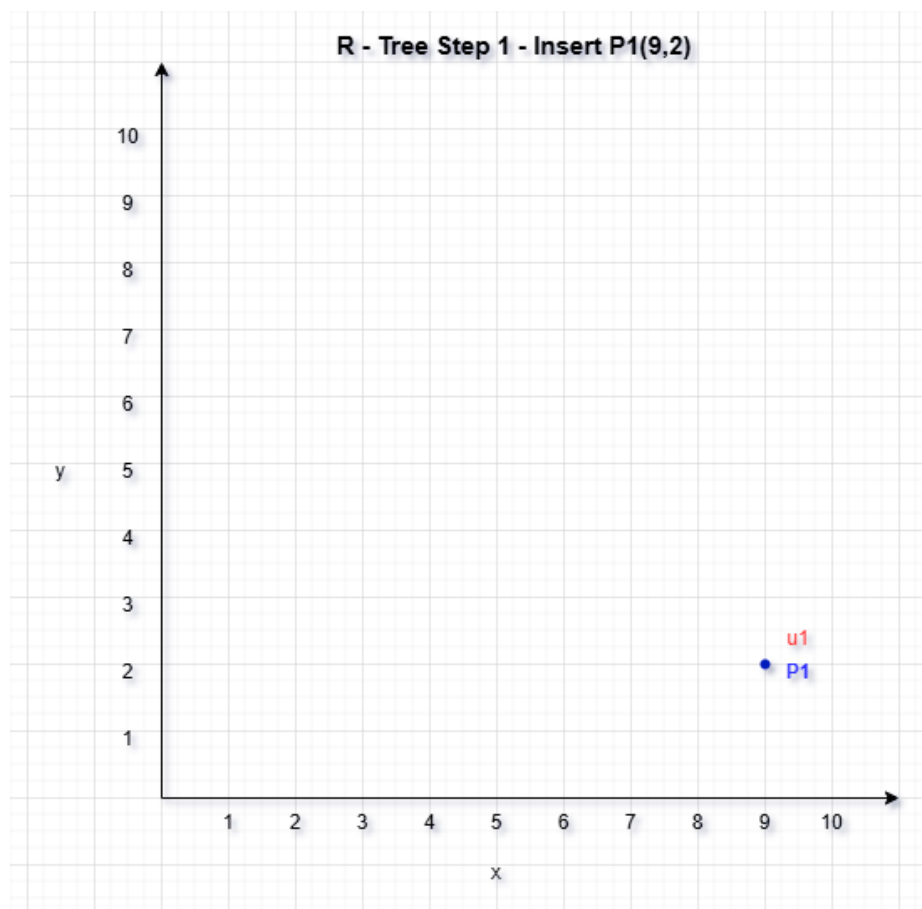
#### MBR (Minimum Bounding Rectangle):

- Since P1 is the only point, the MBR becomes a degenerate rectangle at its coordinates:

$$X_{\min} = X_{\max} = 9$$

$$Y_{\min} = Y_{\max} = 2$$

- The MBR simply wraps the point P1 with no expansion needed.



**Figure 5 :Initial Insertion of P1 into the R-Tree**

- u1 acts as both the **root and a leaf node** at this stage.

## Step 2: Insert Point P2

- Point P2 (5, 4) is inserted into the existing leaf node u1, which already contains P1 (9, 2).
- Since u1 still has capacity (fan-out > 2), no split is required.

## Updated MBR for Node u1:

- After adding P2, the MBR must now cover both P1 and P2:

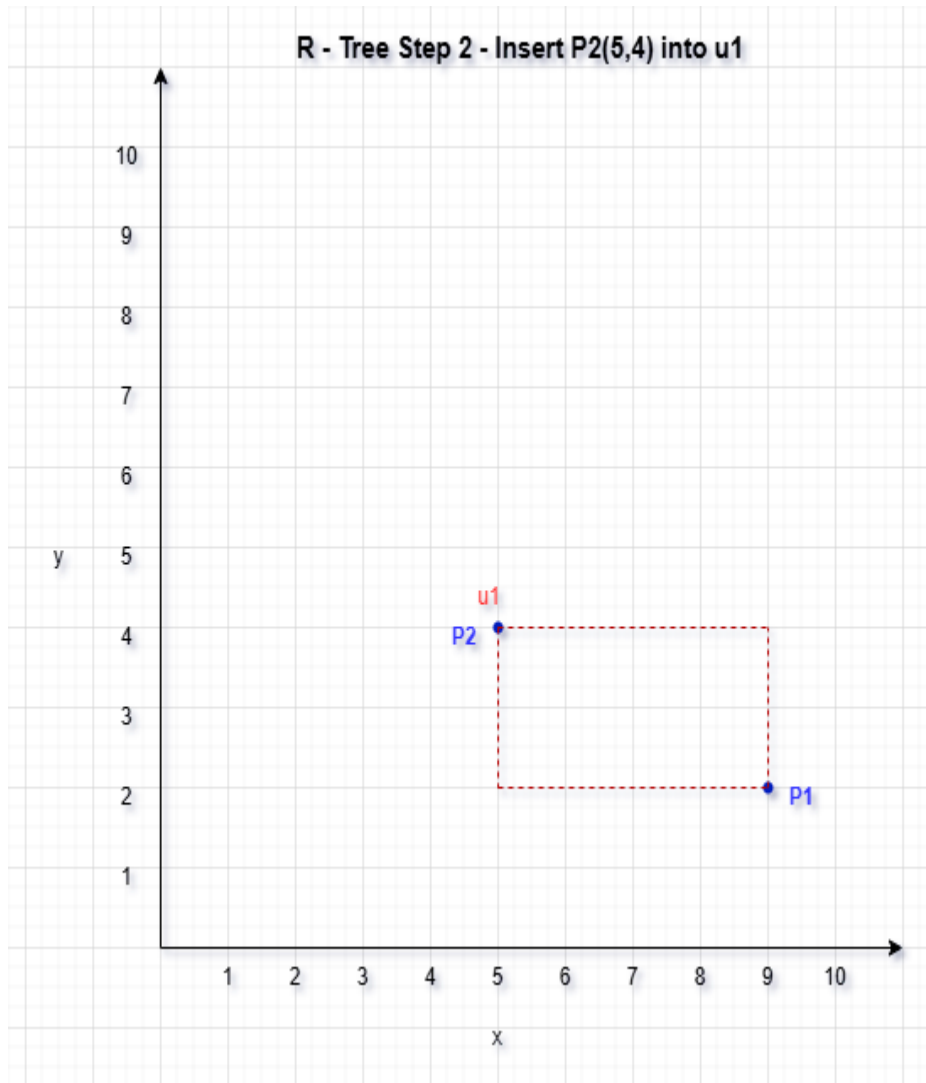
$X_{min} = 5$  (from P2)

$X_{max} = 9$  (from P1)

$Y_{min} = 2$  (from P1)

$Y_{max} = 4$  (from P2)

- New MBR spans from **(5, 2)** to **(9, 4)**



**Figure 6 : Insertion of P2 and MBR**

- u1 remains the root and a leaf node.
- Still under capacity (2 entries).

### **Step 3: Insert Point P3**

- Point P3 (6, 7) is inserted into the existing leaf node u1, which currently holds P1 and P2.
- Since this brings the total entries to 3 (equal to the fan-out limit), insertion proceeds without a split.

### **Updated MBR for Node u1:**

- After inserting P3, the MBR is recalculated to include all three points:

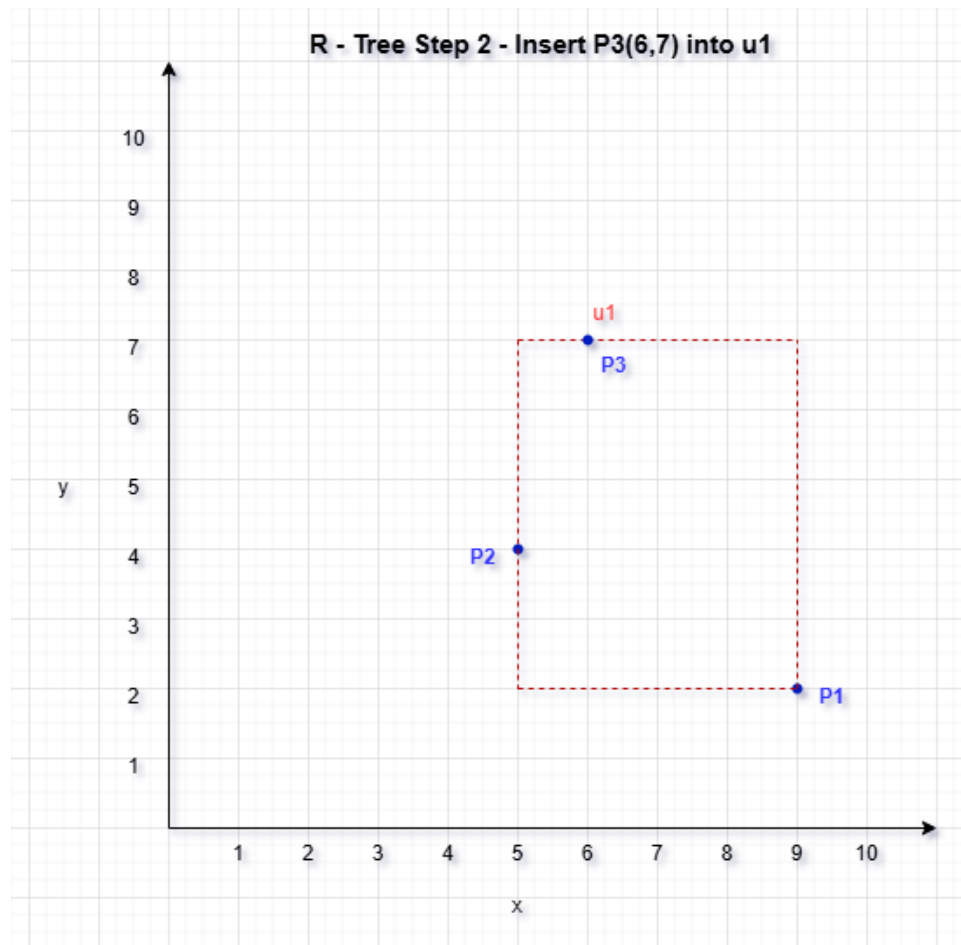
$X_{min} = 5$  (from P2)

$X_{max} = 9$  (from P1)

$Y_{min} = 2$  (from P1)

$Y_{max} = 7$  (from P3)

- New MBR spans from (5, 2) to (9, 7)



**Figure 7 : Insertion of P3 and Updated MBR**

- Node u1 is now at full capacity (3 entries).
- The tree remains balanced and simple at this stage.

#### **Step 4: Insert Point P4 and Split Node u1**

- Point P4 (8, 6) is inserted into node u1, which already holds P1, P2, P3.

- Since  $u_1$  exceeds its fan-out of 3, a split is triggered.

### **Perform Node Split:**

- To split, we create two new child nodes,  $u_2$  and  $u_3$  and distribute the points between them to minimize the total MBR perimeter.

To determine the best way to split, we compare the MBR perimeters by sorting on both the X-axis and Y-axis, and then choose the option that minimizes the total perimeter. This approach helps to provide a balanced distribution and minimizes the overall perimeter.

### **Process for Each Option:**

#### **1. Sorting on the X-axis (P2, P3, P4, P1):**

Split as  $u_2 = (P2, P3)$  and  $u_3 = (P4, P1)$ .

#### **Calculate the MBR perimeters:**

For  $u_2$ :  $X_{min} = 5, X_{max} = 6, Y_{min} = 4, Y_{max} = 7 = \text{Perimeter} = 2 \times ((6 - 5) + (7 - 4)) = 8$

For  $u_3$ :  $X_{min} = 8, X_{max} = 9, Y_{min} = 2, Y_{max} = 6 = \text{Perimeter} = 2 \times ((9 - 8) + (6 - 2)) = 10$

**Total Perimeter =  $8 + 10 = 18$**

#### **2. Sorting on the Y-axis (P1, P2, P4, P3):**

Split as  $u_2 = (P1, P2)$  and  $u_3 = (P4, P3)$ .

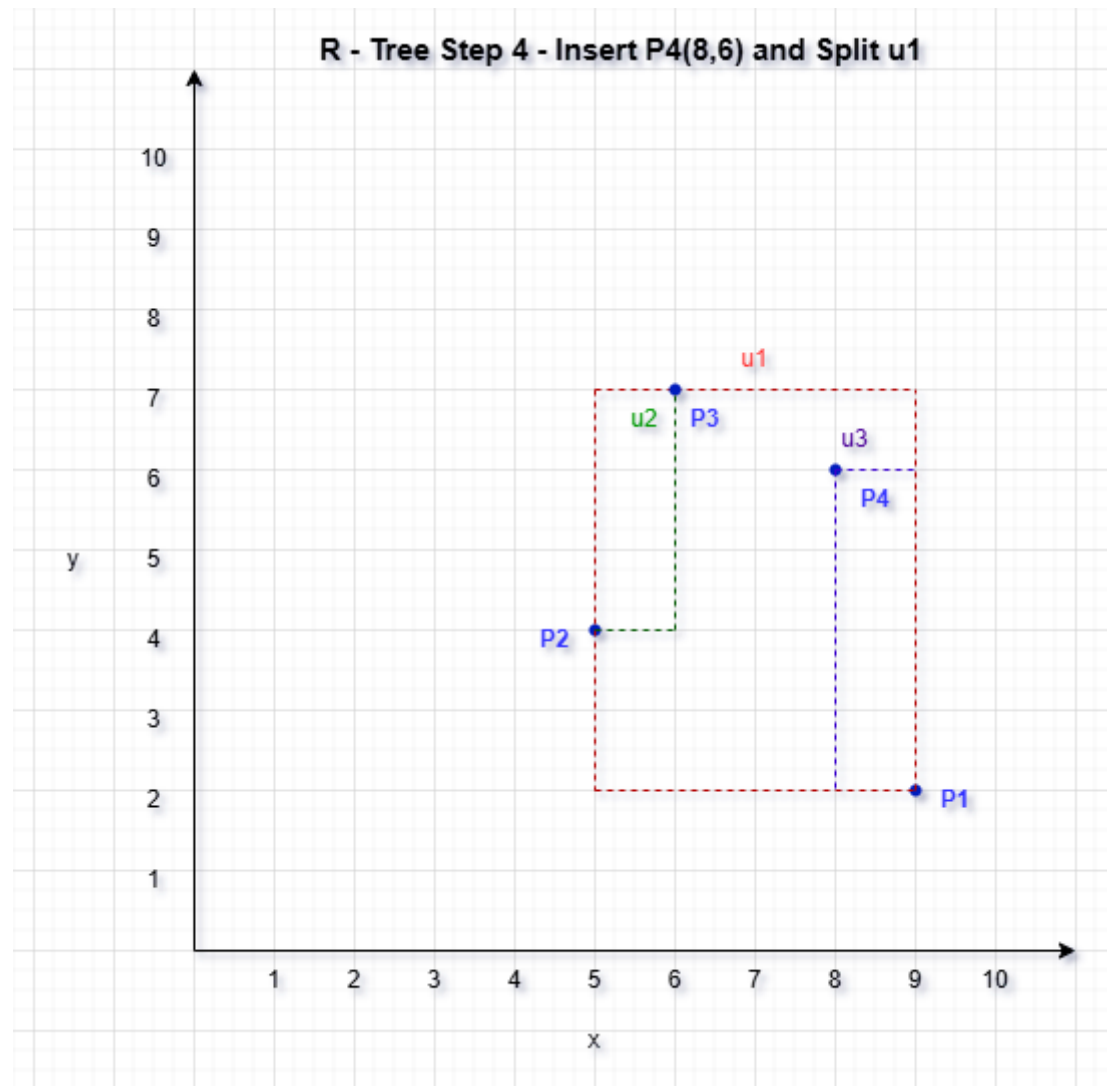
#### **Calculate the MBR perimeters:**

For  $u_2$ :  $X_{min} = 5, X_{max} = 9, Y_{min} = 2, Y_{max} = 4 = \text{Perimeter} = 2 \times ((9 - 5) + (4 - 2)) = 12$

For  $u_3$ :  $X_{min} = 6, X_{max} = 8, Y_{min} = 6, Y_{max} = 7 = \text{Perimeter} = 2 \times ((8 - 6) + (7 - 6)) = 6$

**Total Perimeter =  $12 + 6 = 18$**

Since both sorting approaches result in the same total perimeter (18), either option could be chosen. However, to maintain a more balanced distribution of points between the two child nodes, sorting along the X-axis would be preferable, as it splits the points more evenly. Therefore, X-axis sorting is chosen as the optimal split for creating the two child nodes.



**Figure 8 : Insertion of P4 and Split of Root Node (X-Axis)**

- u1 is now an internal node
- u2 and u3 are new leaf nodes

### **Step 5: Insert Point P5**

- Point P5 (1, 9) is evaluated for insertion.
- Both child nodes (u2 and u3) are not full (each has 2 points).
- We compute the MBR expansion for each to determine optimal placement.

### **MBR Expansion Comparison:**

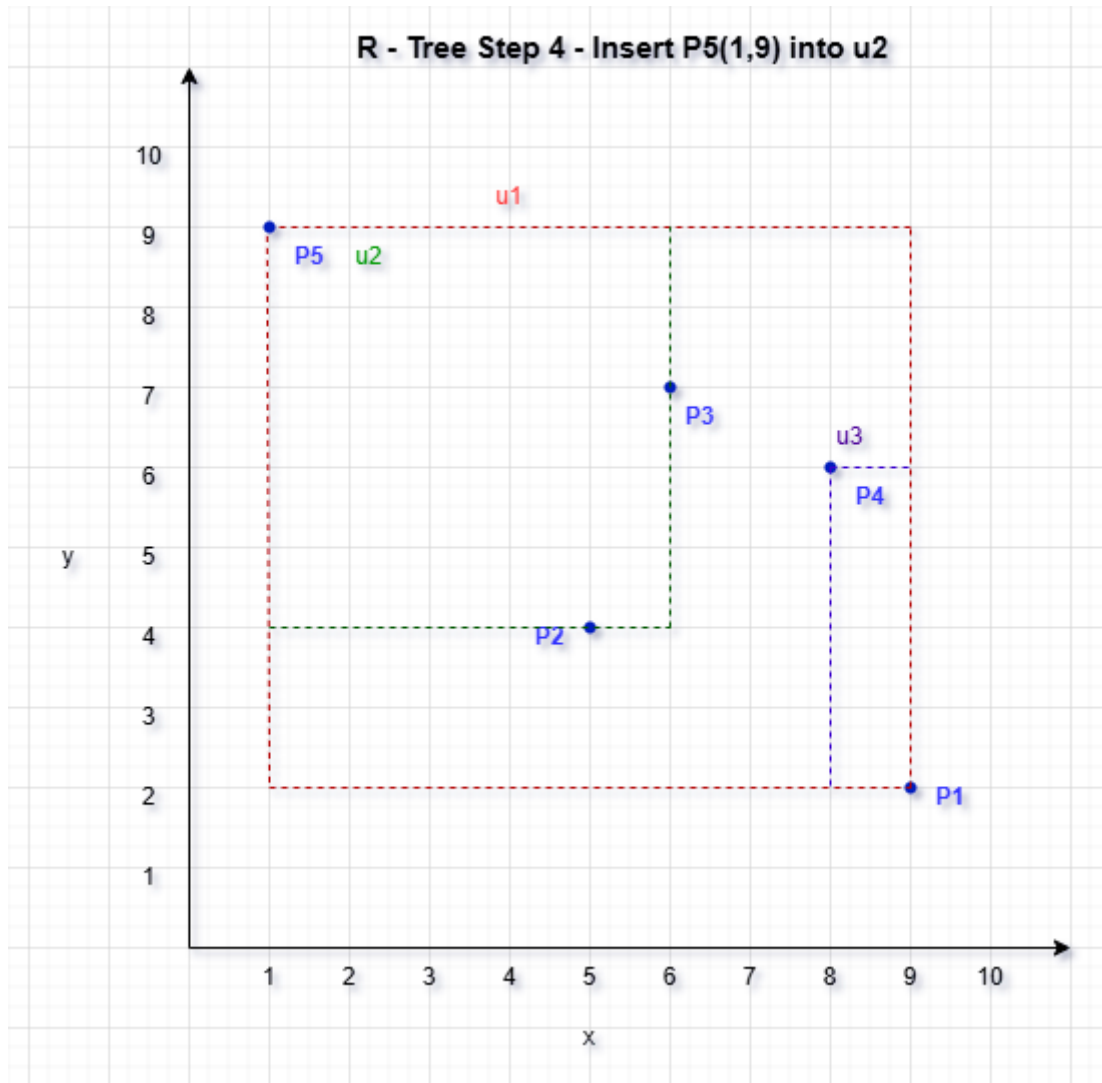
u2 (P2, P3):

- Current MBR: (5, 4) to (6, 7)
- New MBR with P5: (1, 4) to (6, 9)
- Perimeter =  $2 \times ((6-1) + (9-4)) = 20$

u3 (P4, P1):

- Current MBR: (8, 2) to (9, 6)
- New MBR with P5: (1, 2) to (9, 9)
- Perimeter =  $2 \times ((9-1) + (9-2)) = 30$

**u2 chosen (less expansion)**



**Figure 9 : Insertion of P5 and the updated MBR**

- u2 is now full.
- u3 still has 1 slot.

**Updated MBRs:**

- u2 (Green): P2, P3, P5 → MBR: (1, 4) to (6, 9)
- u3 (Purple): P4, P1 → remains unchanged
- u1 (Red): Encloses u2 and u3 → updated to (1, 2) to (9, 9)



### **Step:6 Insert P6 (4,7) and Node Split in R-tree :**

R-tree Before Insertion

- Root node: u1

Two child nodes: u2 and u3

u2 contains: P2 (5,4), P3 (6,7), P5 (1,9)

u3 contains: P1 (9,2), P4 (8,6)

### **Insertion Strategy**

Since **u2 is already full** (contains 3 points and has a max fan-out of 3), we must check if **u3 can accommodate P6**.

**u3's MBR Before Insertion:**

$X_{min} = 8, X_{max} = 9$

$Y_{min} = 2, Y_{max} = 6$

Area/perimeter is relatively tight

**If We Insert P6 (4, 7) into u3:**

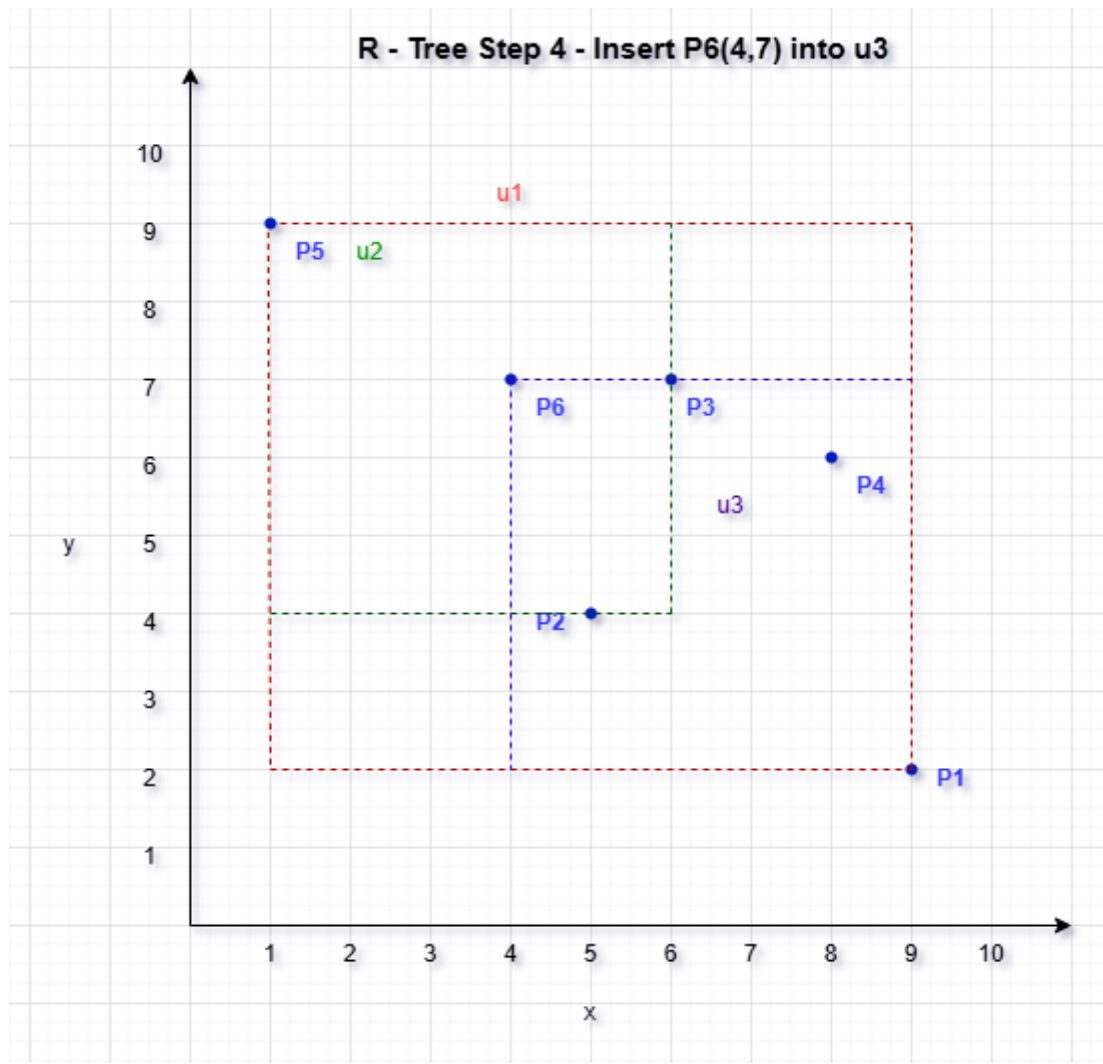
- MBR expands to:

$X_{min} = 4$  (from P6),  $X_{max} = 9$  (P1)

$Y_{min} = 2$  (P1),  $Y_{max} = 7$  (P6)

- **Perimeter** =  $2 \times ((9-4) + (7-2)) = 20$

**u3 has room for 1 more entry → Valid insertion**



**Figure 10 : Insertion of P6 and the updated MBR**

**Diagram Explanation:**

Blue Dots: Represent data points P1 to P6

Red Box (u1): Root MBR that spans all child MBRs

Green Box (u2): MBR enclosing P2, P3, P5

Purple Box (u3): MBR enclosing P4, P1, P6

- P6 was inserted into u3 to avoid splitting full node u2
- This decision minimizes MBR expansion while maintaining balance
- The R-tree now has 6 points organized in a 2-level hierarchy with both child nodes full

### **Step:7 Insert Point 7 (2, 5) into the R-tree :**

Both u2 and u3 are currently at full capacity (3 points each).

We must:

- Compute MBR expansion if P7 were inserted into either node
- But since both nodes are full, any choice will require a split

So we compute which one would result in less MBR expansion, and then split that node.

### **MBR Expansion Comparison**

#### **Inserting into u2:**

- Current MBR (P2, P3, P5): (1, 4) to (6, 9)
- P7 = (2, 5) → already inside this MBR
- **MBR does not need to expand** (excellent spatial fit!)
- Best spatial choice, but **u2 is full** → **must split**

#### **Inserting into u3:**

- MBR: (4, 2) to (9, 7)
- P7 = (2, 5) → Expands Xmin from 4 → 2
- New MBR = (2, 2) to (9, 7)

#### **More expansion**

- Also full → would split

**Conclusion:** Insert P7 into **u2** and **split u2**

### **Split Node u2**

**-u2 contains: P2 (5,4), P3 (6,7), P5 (1,9), and now P7 (2,5)**

We apply **linear split using X-axis** for clarity:

- Sort: P5 (1,9), P7 (2,5), P2 (5,4), P3 (6,7)

- Split:

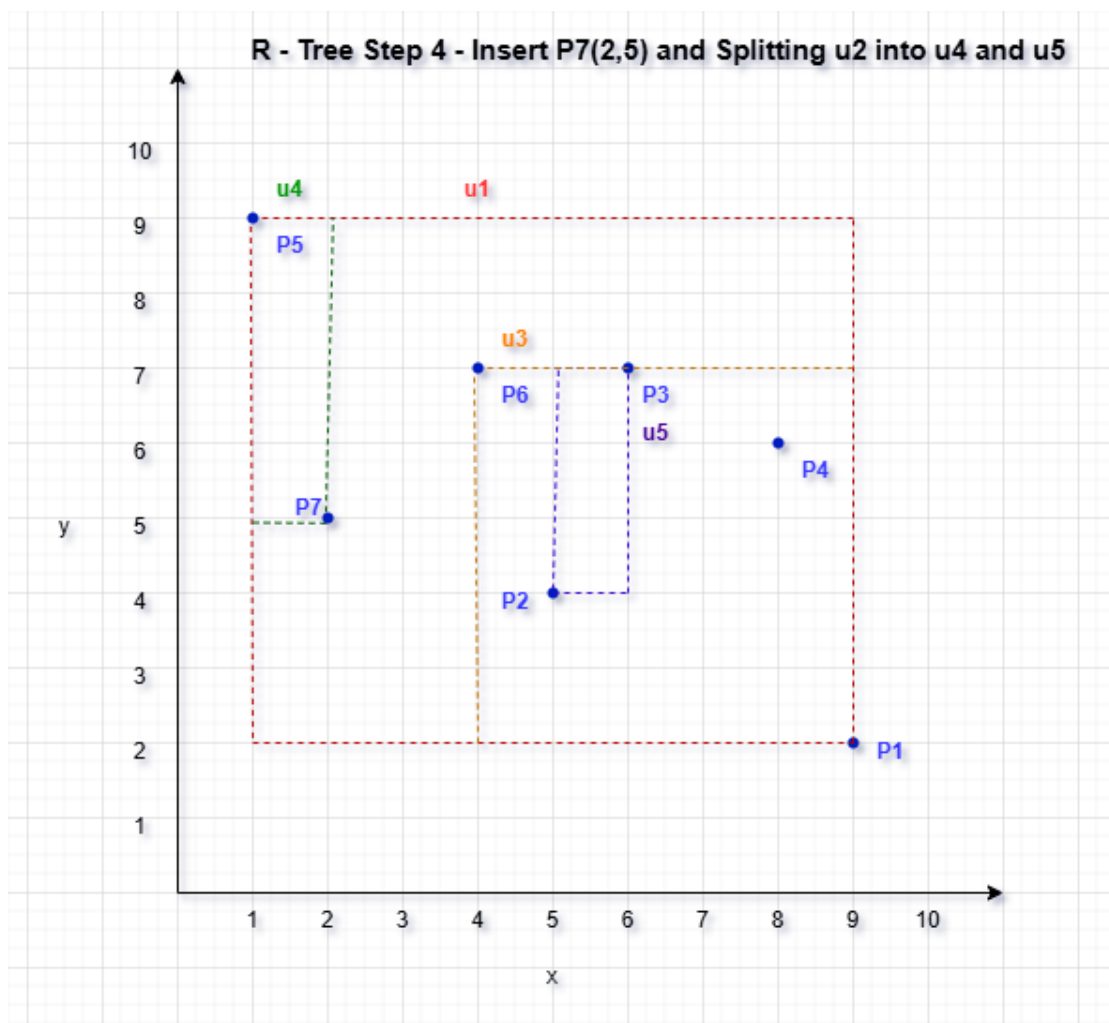
**New u4** = {P5, P7}

**New u5** = {P2, P3}

### Update Tree Structure

We now need to:

- Replace u2 in root u1 with new nodes **u4** and **u5**
- u1 now has 3 children: u4, u5, u3 (no overflow in root yet)



**Figure 11 : Insertion of P7 and the updated MBR**

### Diagram Explanation :

u4 (Green)

Contains: P5 (1, 9), P7 (2, 5)

MBR spans X: 1–2, Y: 5–9

u5 (Purple)

Contains: P2 (5, 4), P3 (6, 7)

MBR spans X: 5–6, Y: 4–7

u3 (Orange)

Contains: P4 (8, 6), P1 (9, 2), P6 (4, 7)

MBR spans X: 4–9, Y: 2–7

u1 (Root)

Encloses all child MBRs

### **Step: 8 Insert Point P8 (1, 4) into the R-tree :**

Insert the data point P8 (1, 4) into the current R-tree structure while minimizing the expansion of Minimum Bounding Rectangles (MBRs) and maintaining tree balance.

### **Insertion Strategy:**

The R-tree at this point has three leaf nodes under the root (u1):

u4: P5, P7

u5: P2, P3

u3: P4, P1, P6

To determine the optimal insertion node for P8, we evaluate the MBR expansion required by each candidate node.

### **MBR Expansion Analysis:**

- **u4** (P5, P7):

Original MBR: (1, 5) to (2, 9)

Inserting P8 (1, 4) extends Ymin  $\rightarrow$  (1, 4) to (2, 9)

New Perimeter =  $2 \times ((2-1) + (9-4)) = \mathbf{12}$

u4 has room (only 2 entries) → Ideal choice

- **u5** (P2, P3):

Would require expanding Xmin to 1

Perimeter increases to **16**

Full → would require split

- **u3** (P4, P1, P6):

Already full

Would expand Xmin to 1

Perimeter increases to **26**

Not suitable

**Decision:**

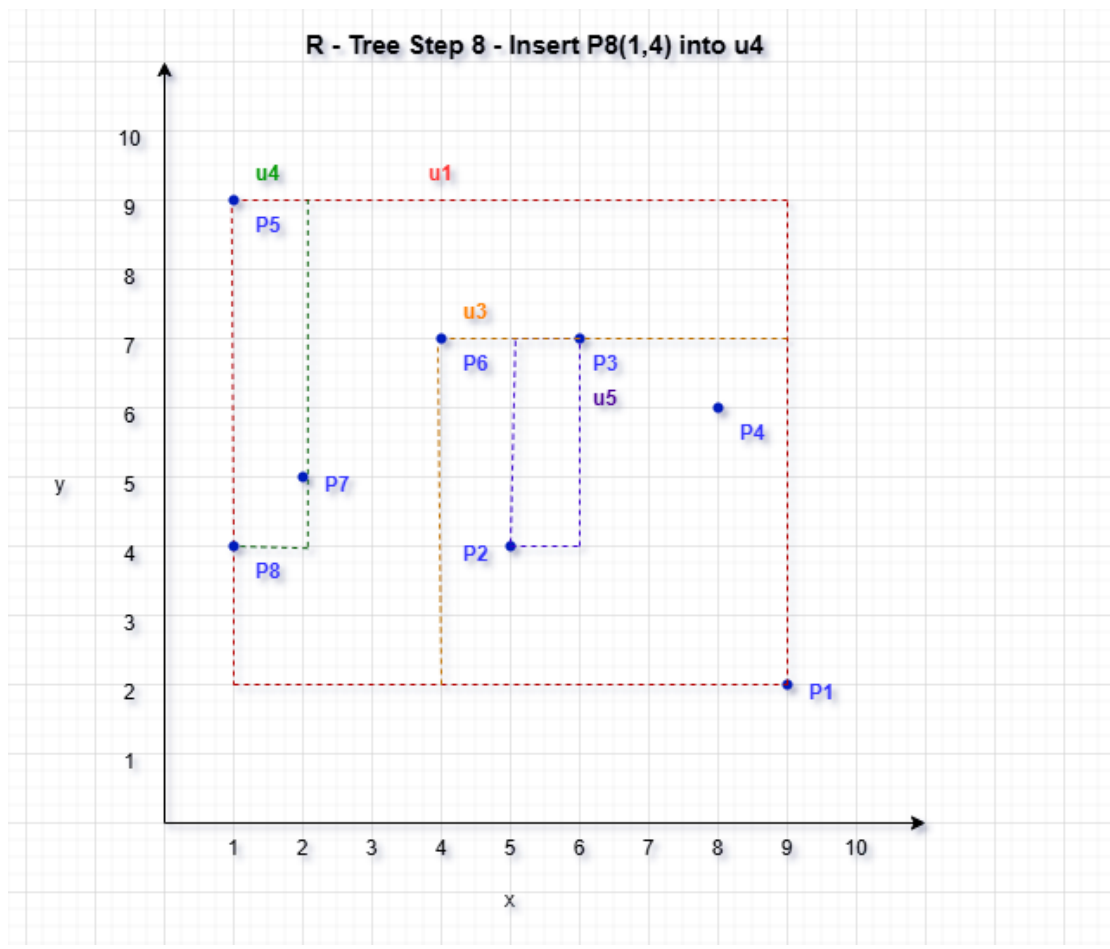
- **P8 is inserted into u4**, which now contains:

**P5 (1, 9)**

**P7 (2, 5)**

**P8 (1, 4)**

- MBR of **u4** updates to: (1, 4) to (2, 9)



**Figure 12 : Insertion of P8 and the updated MBR**

**Diagram Explanation:**

Blue Dots: Represent points P1 through P8

u4 (Green MBR):

Points: P5, P7, P8

MBR spans X: 1–2, Y: 4–9

u5 (Purple MBR):

Points: P2, P3

MBR spans X: 5–6, Y: 4–7

u3 (Orange MBR):

Points: P4, P1, P6

MBR spans X: 4–9, Y: 2–7

u1 (Red MBR):

Root node MBR

Spans all three children u4, u5, and u3

Each MBR is accurately represented by dashed rectangles and labeled.

**Step : 9 Insert Point P9 into the R-tree :**

To insert P9 (4, 7) into the existing R-tree while maintaining minimal MBR expansion and avoiding unnecessary splits.

**MBR Expansion Evaluation:**

**Option A: Insert into u4**

Full → would require a split

**Option B: Insert into u5**

Current MBR: (5,4) to (6,7)

Adding P9 = (4,7) expands Xmin to 4

New MBR: (4,4) to (6,7)

New Perimeter:  $2 \times ((6-4)+(7-4)) = 10$

u5 has space → optimal

**Option C: Insert into u3**

Full → would require a split

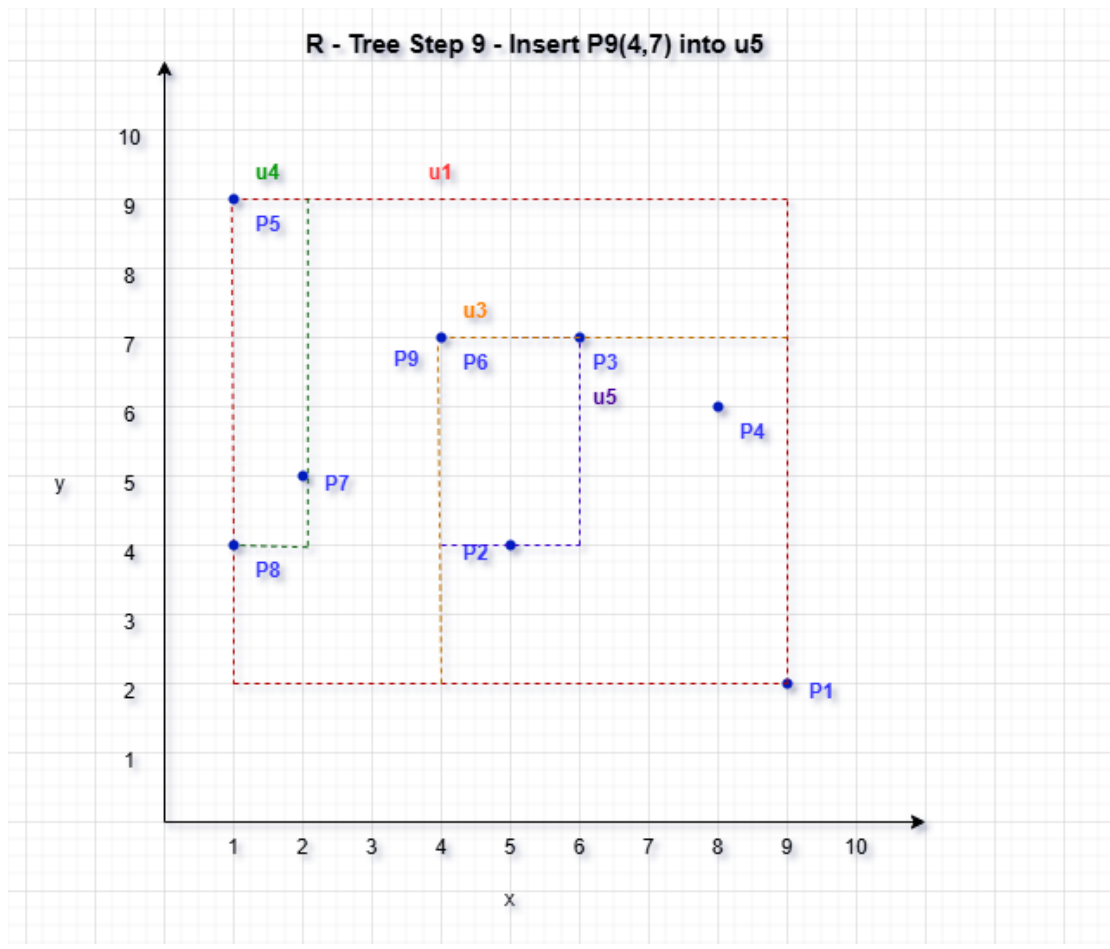
**Decision:**

P9 inserted into u5

u5 now holds: P2, P3, P9

u5 MBR updated to: (4,4) to (6,7)





**Figure 13 : Insertion of P9 and the updated MBR**

**Diagram Explanation:**

- **Blue Dots:** Represent points P1 to P9

- **Green MBR (u4):**

Points: P5, P7, P8

MBR spans X: 1–2, Y: 4–9

- **Purple MBR (u5):**

Points: P2, P3, P9

Updated MBR spans X: 4–6, Y: 4–7

- **Orange MBR (u3):**

Points: P4, P1, P6

MBR spans X: 4–9, Y: 2–7

- **Red MBR (u1):**

Root MBR encapsulating all three child MBRs

**Each rectangle is labeled, and spatial distribution is balanced and clear.**

**Step : 10 Insert Point 10 (ID 10: X=6, Y=2) :**

Insert point P10 (6, 2) into the R-tree while maintaining balance and minimizing the impact on spatial indexing performance.

**Insertion Decision:**

All existing leaf nodes (u4, u5, u3) were already full with 3 entries each. Thus, inserting P10 into any of them would trigger a split.

We need to:

- Calculate which node incurs the **least MBR expansion**
- Insert and **split that node**

**MBR Expansion Cost**

**Option A: u4 (P5, P7, P8)**

MBR: (1, 4) to (2, 9)

P10 = (6, 2) → massive X and Y expansion

New MBR: (1, 2) to (6, 9)

$$\text{Perimeter} = 2 \times ((6-1) + (9-2)) = 28$$

**Option B: u5 (P2, P3, P9)**

MBR: (4, 4) to (6, 7)

P10 = (6, 2) → Ymin expands from 4 → 2

New MBR: (4, 2) to (6, 7)

$$\text{Perimeter} = 2 \times ((6-4) + (7-2)) = 20$$

### Option C: u3 (P4, P1, P6)

MBR: (4, 2) to (9, 7)

P10 = (6, 2) → already inside

- **No MBR change**

- **Perimeter remains the same (22)**

- But u3 is **already full**

### Decision:

- **u5** is the best choice:

Causes less expansion than u4

u3 would be ideal spatially, but it's already full

- u5 is full → we must **split it**

### Split u5:

- u5 currently has: P2 (5,4), P3 (6,7), P9 (4,7), P10 (6,2)

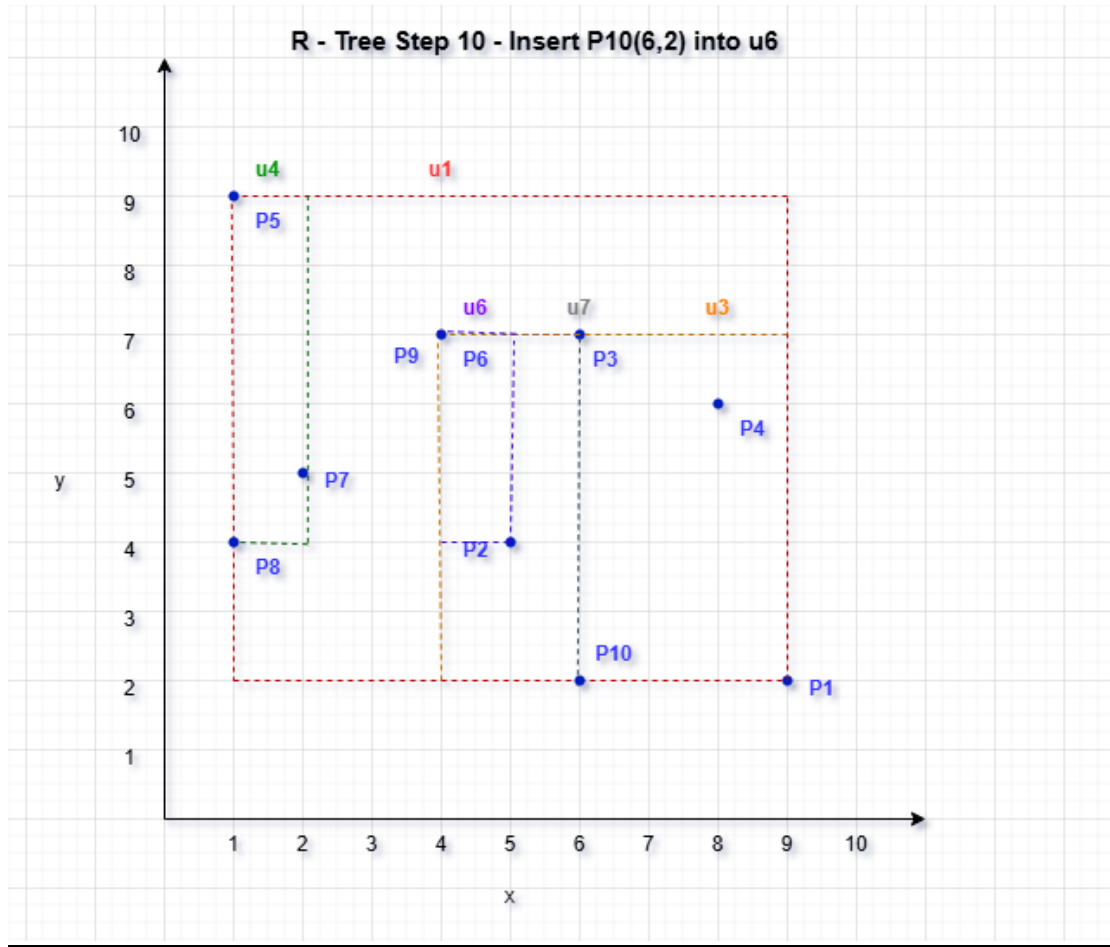
### Sort by X-axis:

→ P9 (4,7), P2 (5,4), P3 (6,7), P10 (6,2)

### Grouping:

**u6** = {P9, P2}

**u7** = {P3, P10}

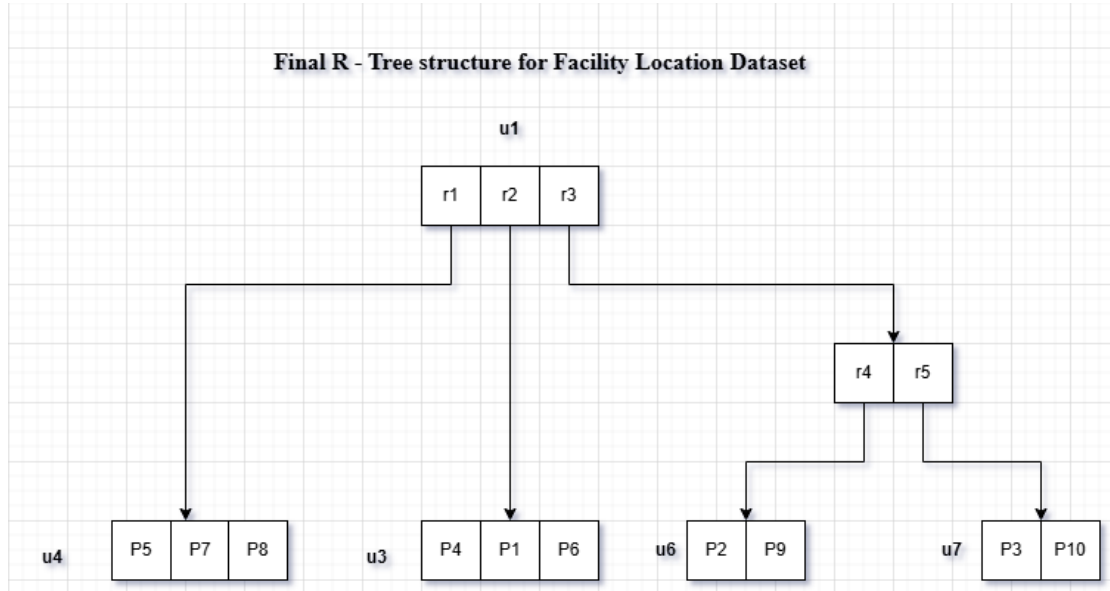


**Figure 14 : Insertion of P10 and the updated MBR**

A spatial indexing structure constructed over ten facility location points (P1 to P10) is depicted in the final R-tree figure. Four leaf nodes, u4, u6, u7, and u3, each of which contains a fraction of the data points, are encapsulated by the root node u1 at the top level. In particular, P5, P7, and P8 are found in u4; P9 and P2 are found in u6; P3 and P10 are found in u7; and P1, P4, and P6 are found in u3. These clusters make sure that no node goes above the three-fan-out maximum.

The Minimum Bounding Rectangle (MBR) of the root node covers the whole spatial extent of all points, while the MBR of each leaf node is contained inside its own MBR. The R-tree supports quick nearest neighbor and range searches and is two layers deep, effectively balancing point distribution. The incremental insertion and split decisions made during construction are reflected in the structure's compactness and spatial access optimization.

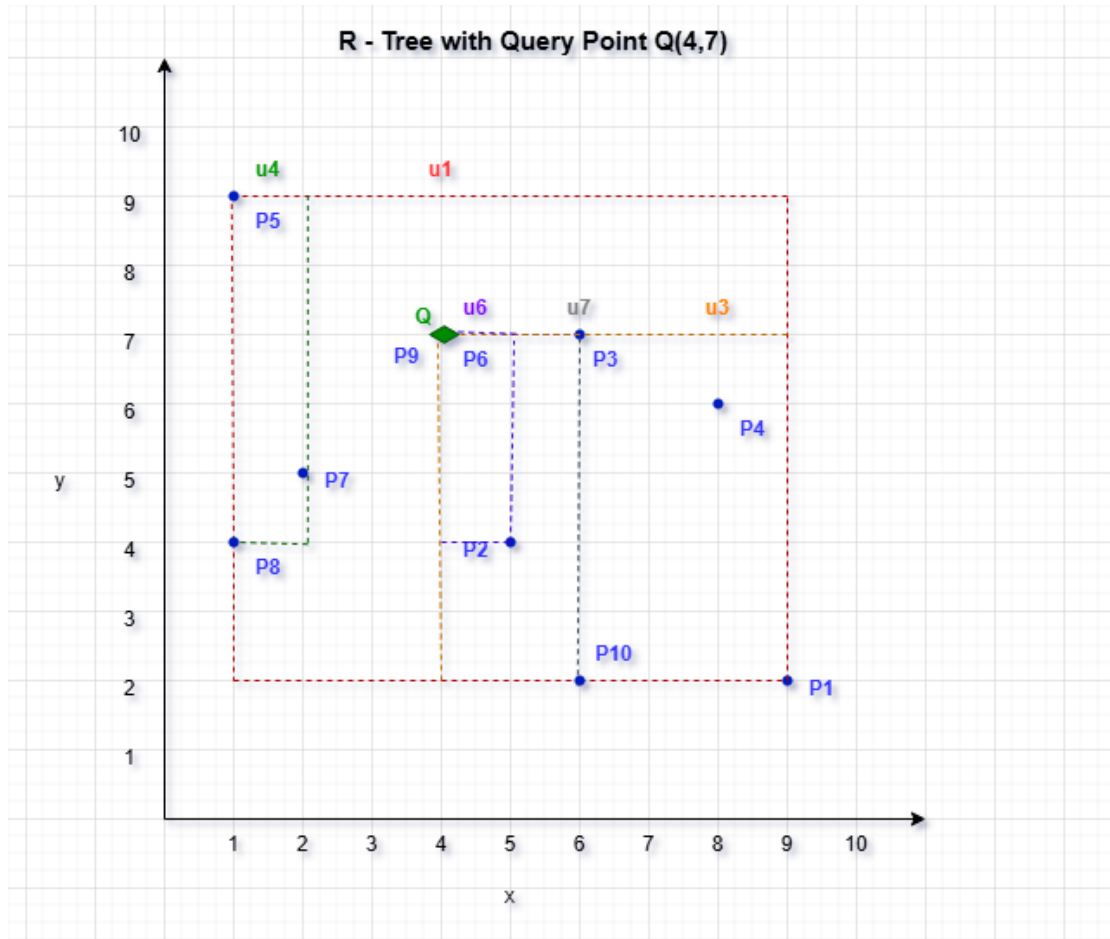
### Final R-Tree Structure for Facility Location Dataset



**Figure 15 : Final R-Tree Structure for Facility Location Dataset**

The dataset is represented in a compact, balanced manner by this R-tree structure. All points are effectively indexed across four leaf nodes beneath a single root node in the final structure, which was constructed sequentially utilizing insertion and node splitting based on spatial criteria (MBR expansion).

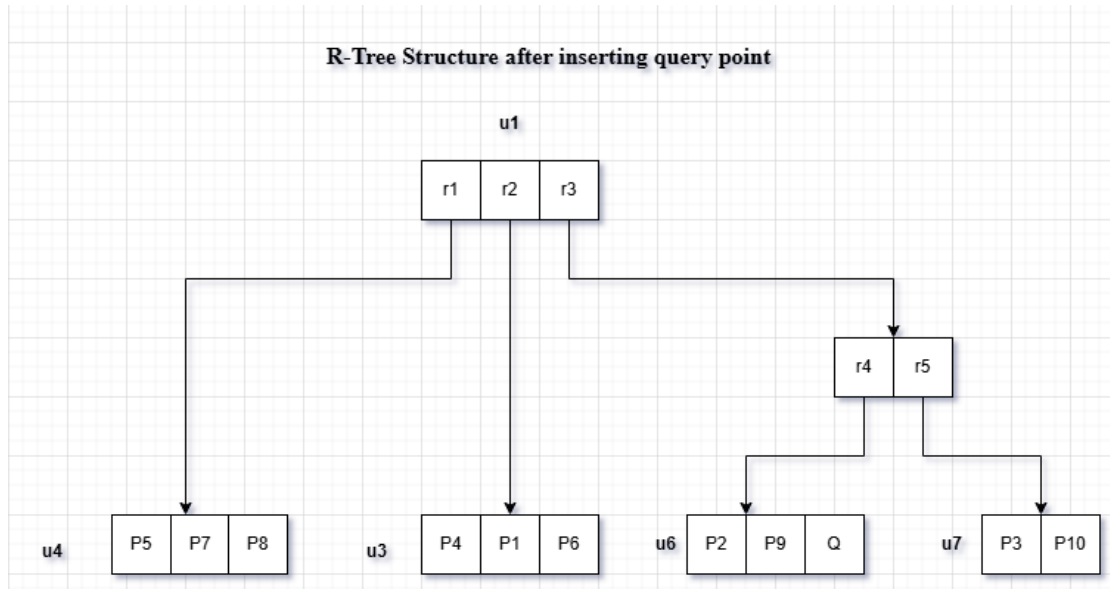
**Figure for R-tree after inserting the query point.**



**Figure 16 : Final R-Tree after adding Query Point (4,7)**

Finally, this is the complete R-tree structure following the addition of the query point Q at coordinates (4,7) and all facility location points. With modifications to the Minimum Bounding Rectangles (MBRs) at different levels as needed, this feature shows how the R-tree adapts to new locations inside the current structure. The hierarchical structure of spatial data is defined by the root node and leaf nodes taken together, which makes access and query processing more effective. The query point Q's insertion highlights the R-tree's dynamic character, which allows it to grow to include more spatial data without sacrificing search effectiveness.

### Final R-Tree Structure after inserting query point



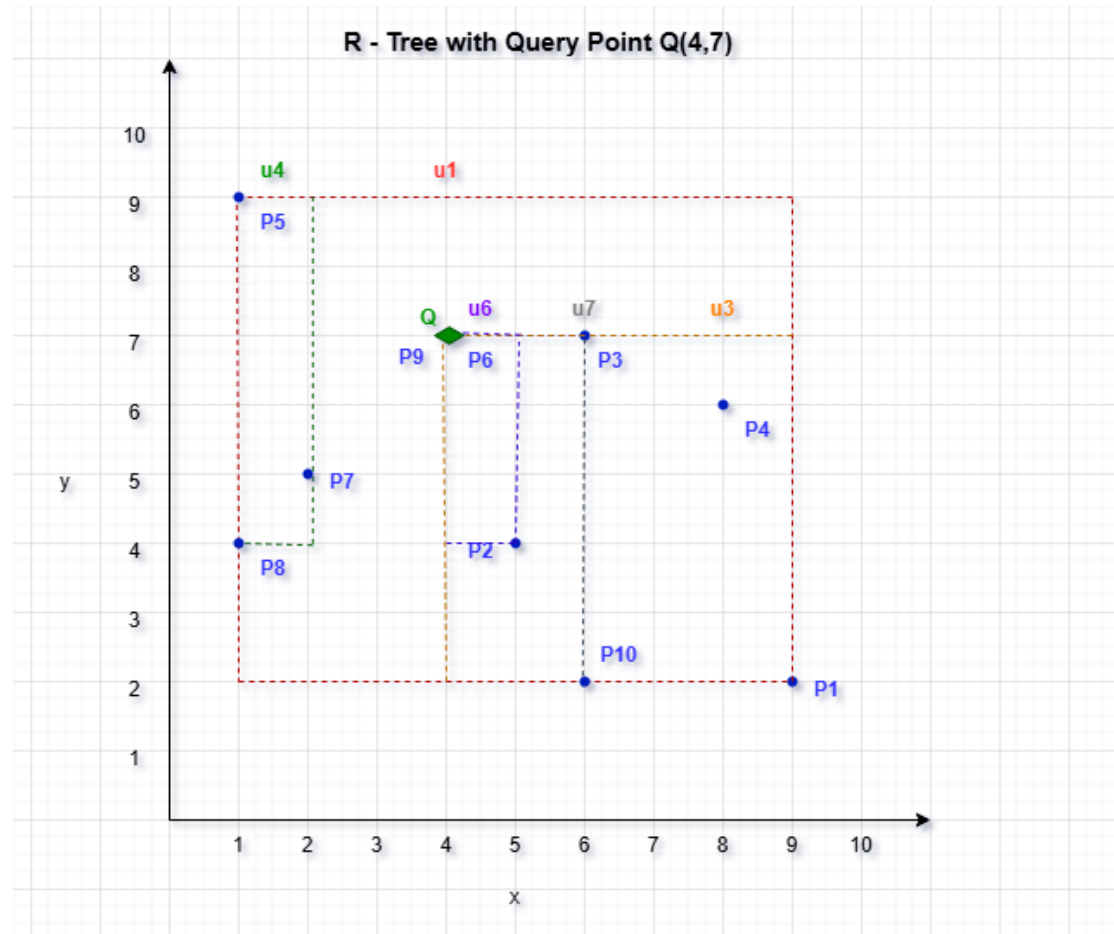
**Figure 17: Final R-Tree Structure after inserting query point**

A root node (u1) and four leaf nodes (u4, u6, u7, and u3), each of which contains a collection of spatially adjacent data points, make form the final R-tree structure for the facility location dataset. In leaf node u6, which already has P9 (4, 7) and P2 (5, 4), the query point Q(4, 7) is added. This choice is based on efficiency and spatial alignment; Q matches P9 exactly, therefore adding it to u6 prevents the existing Minimum Bounding Rectangle (MBR) from growing, but adding it to other nodes, such as u7, would cause needless MBR growth. With no more than three entries per leaf node, the R-tree maintains a compact, balanced structure that facilitates effective spatial indexing and query performance.

## 4.2 The Process of BF Algorithm :

The Best-First (BF) algorithm is a method to find the nearest neighbor to a query point in an R-tree structure.

The goal of the BF algorithm is to quickly locate the closest point by prioritizing nodes based on their distance to the query point. Instead of exploring all nodes, it focuses on those likely to be closest.



**Figure 18 : Initial Steps of the Best-First (BF) Algorithm in an R-tree Structure**

Here, we show how to find the closest neighbor to a query point  $Q = (4, 7)$  using the Best-First (BF) algorithm's first stages in an R-tree structure. In order to prioritize nodes that are likely to contain the nearest points, the BF algorithm first determines the minimum distance (mindist) between each node and the query point.

The facilities' spatial arrangements within their Minimum Bounding Rectangles (MBRs) are depicted in the figure. The BF algorithm refines the search by updating the priority queue and computing the mindist from  $Q$  to each MBR.



## **Step-by-Step Plan for BF Algorithm:**

### **1. Initialize the Priority Queue**

- Insert the root node  $u_1$  into the priority queue.
- Use the minimum distance between the query point  $Q(4, 7)$  and each MBR as the priority.

### **2. Traversal Process**

- Pop the MBR (or point) with the lowest distance from the queue.
- If it's a non-leaf node:

Insert all its children MBRs with their distance to  $Q$ .

- If it's a leaf node:

Compute exact distances from  $Q$  to each data point in that node.

Track the closest point found so far.

### **3. Termination**

- Once the nearest data point is popped from the queue, return it as the result.

## **Initial Setup (Iteration 0)**

Query Point:  $Q(4, 7)$

### **Priority Queue (Min-Heap):**

We start by inserting the root node  $u_1$ :

In a Min-Heap-based priority queue for R-tree search, we begin by inserting the root node  $u_1$  because the query point  $Q$  lies within its Minimum Bounding Rectangle (MBR). Therefore, the distance to  $Q$  (MinDist) is 0, making  $u_1$  the first and highest-priority entry in the queue.

## Iteration 1: Expand u1

We now expand root u1 and enqueue its children:

After inserting the root node u1 into the priority queue, we expand it by enqueueing its child nodes. These are all internal nodes representing groups of data points enclosed within their respective Minimum Bounding Rectangles (MBRs). Specifically:

- Points P5, P7, and P8 are enclosed when u4 is added to the queue. The MBR border is used to compute its distance to the query point Q (MinDist).
- In addition to having points P9 and P2, u6 is enqueued with the MinDist that corresponds to Q.
- After calculating its MBR's distance from Q, u7—which stands for P3 and P10—is added to the queue.
- Finally, after calculating its MinDist, u3, which contains P1, P4, and P6, is added to the queue.

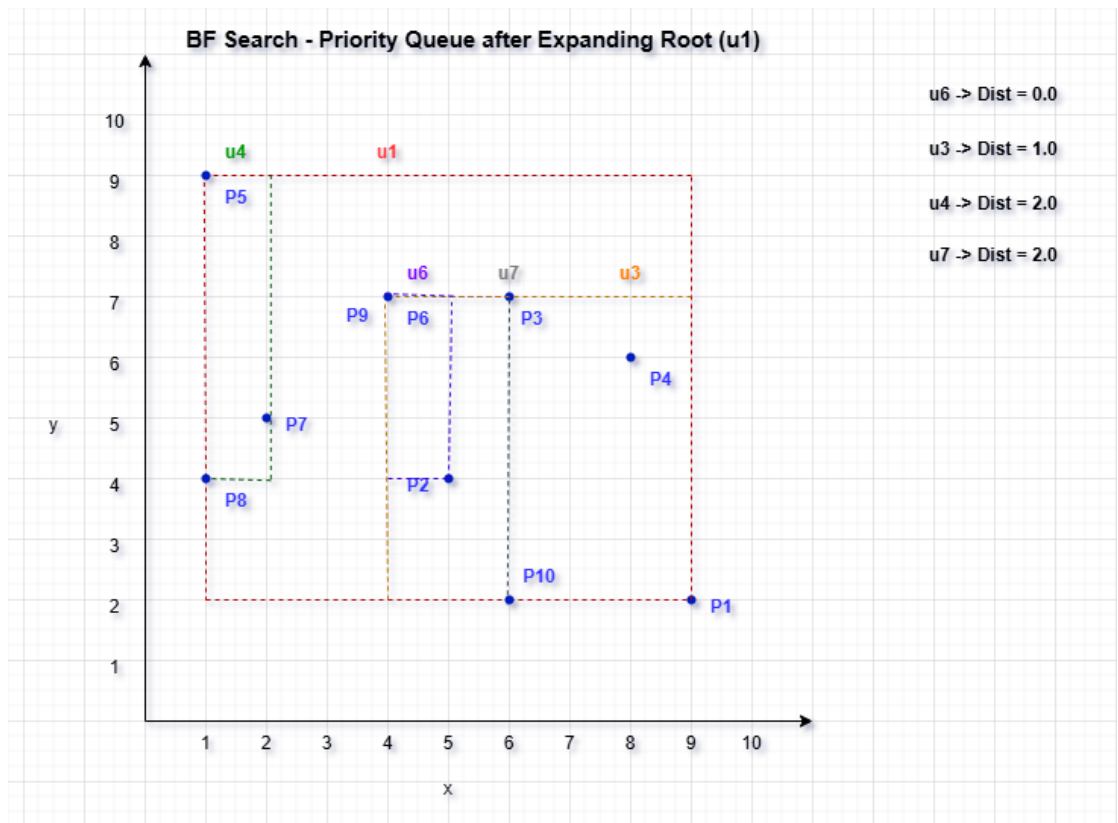
Each of these entries is now prioritized in the queue based on their distance from the query point, ensuring that the closest regions are explored first in the R-tree traversal.

I will now compute the distances from Q(4, 7) to each of these MBRs and update the queue.

## Priority Queue – Iteration 2

Based on the calculated minimum distances (MinDist) from the query point Q(4,7) to the MBRs of the child nodes, we prioritize the nodes in the queue as follows:

- u6 (MinDist = 0.0) is the first priority. The query point is located inside u6's MBR, which also includes points P9 and P2. Since Q and P9 are exactly the same, this node is the most pertinent and closest for investigation.
- Second Priority – u3 (MinDist = 1.0): Q is the next closest region even though it is not inside u3's MBR (which contains P1, P4, and P6) because it is only 1 unit from its nearest edge or corner.
- Third Priority-u4 (MinDist = 2.0): The query point is outside of u4's MBR, which encompasses P5, P7, and P8, and has a 2-unit closest edge distance.
- Fourth Priority: u7 (MinDist = 2.0): Q is likewise two units from u7's MBR, which contains P3 and P10. It has the same distance as u4, therefore it is ranked equally, however because of queue order, it is listed after u4.



**Figure 19 : Visualization of Iteration 2 of the Best-First (BF) Search algorithm**

### Iteration 3 :

Since  $u_6$  has  $\text{MinDist} = 0$ , we expand it next in Iteration 3.

- Expand  $u_6$  and insert its contents (P9, P2, Q) into the priority queue.
- Compute distances from  $Q(4, 7)$  to:

$$P_9(4, 7) \rightarrow 0.0$$

$$P_2(5, 4) \rightarrow \sqrt{[(5-4)^2 + (4-7)^2]} = \sqrt{10} \approx 3.16$$

$$Q(4, 7) \rightarrow 0.0 \text{ (we skip re-checking } Q \text{ itself)}$$

The first point dequeued will be  $P_9$ , the nearest neighbor.

After expanding node  $u_6$ , we examine its data points and insert them into the priority queue:

P9 has a distance of 0.0 from the query point Q(4, 7)—it is an exact match.

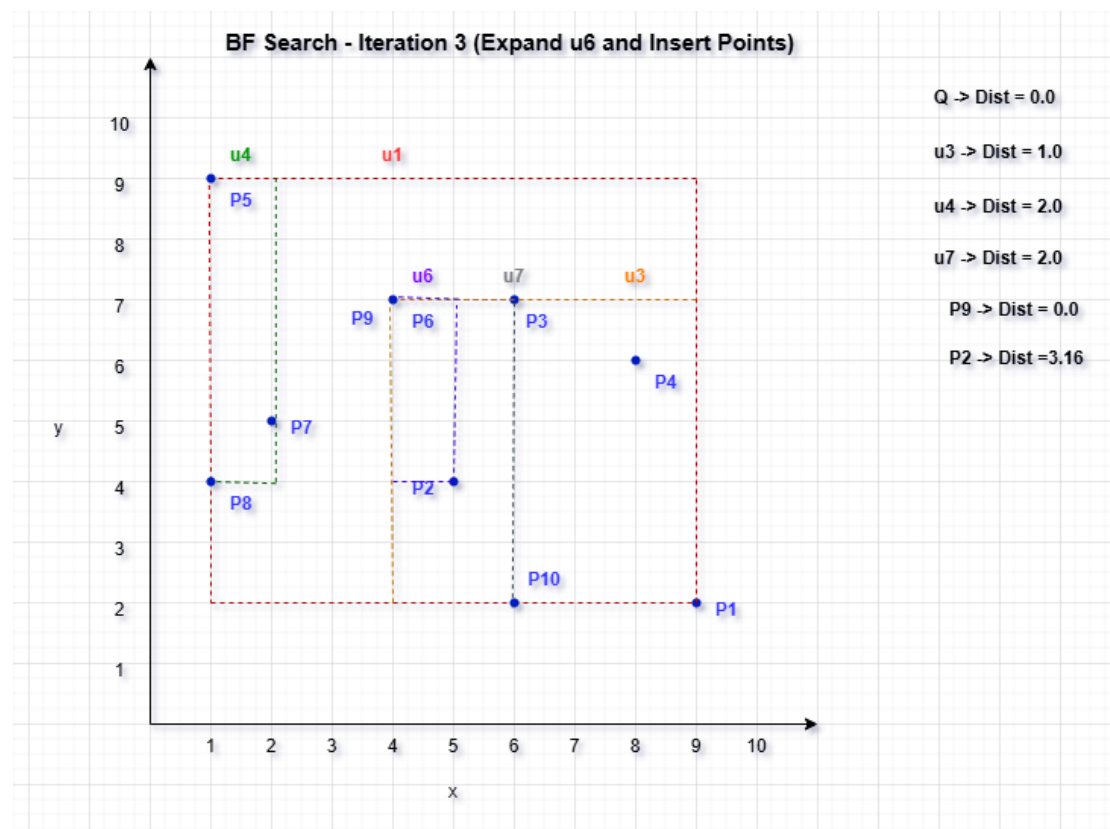
P2 lies farther away, with a calculated distance of approximately 3.16 units from Q.

Since P9 has the smallest distance and is at the top of the priority queue, it is identified as the nearest neighbor. The algorithm terminates at this point because the first data point retrieved from the queue is the closest possible match.

### Final Result:

Nearest Neighbor to Q(4, 7): P9 (4, 7)

Distance: 0.0 (Exact match)



**Figure 20 : Visualization for Iteration 3 of the Best-First Search algorithm**

### Conclusion :

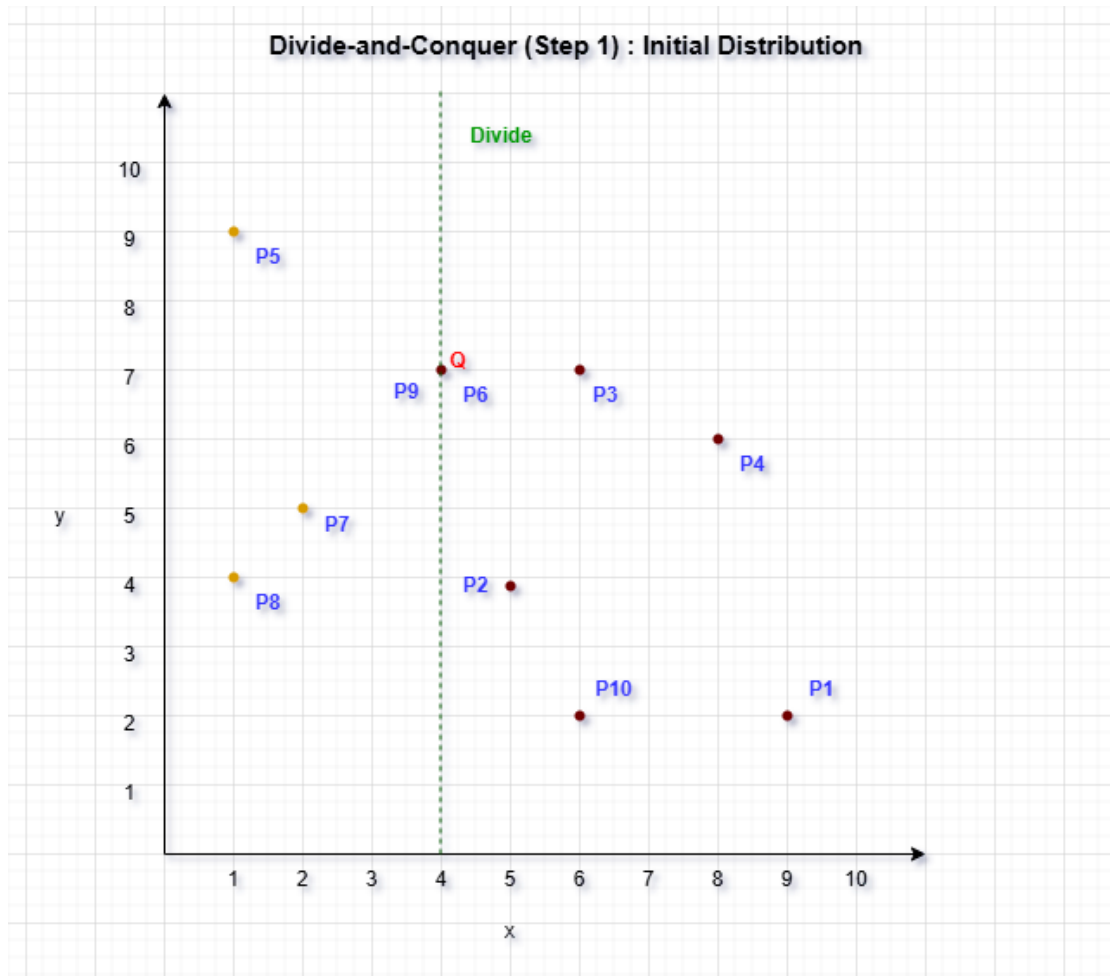
To find the closest facility to the query point Q(4, 7), the BF algorithm was applied to the built R-tree. Starting at node u1, the search expanded nodes according to the smallest distance (MinDist) between the query point and the child nodes' MBRs. The child points of node u6, which had a MinDist of 0.0, were added to the priority queue once it was expanded. P9 (4,7) was the one that most closely matched the query point,

resulting in a distance of 0.0. The search ended instantly since the BF algorithm ensures that the first data point dequeued is the nearest neighbor. As a result, P9 was effectively and precisely determined to be Q's closest neighbor utilizing the R-tree's spatial indexing structure.

### **4.3 Divide-and-Conquer :**

Finding the nearest neighbor in an R-tree structure can be made much more efficient by using the divide-and-conquer technique. The algorithm lowers the quantity of points it must take into account by breaking the search area up into smaller, more manageable portions and examining each one separately. This method reduces pointless evaluations by enabling the BF search algorithm to focus on the regions nearest to the query location.

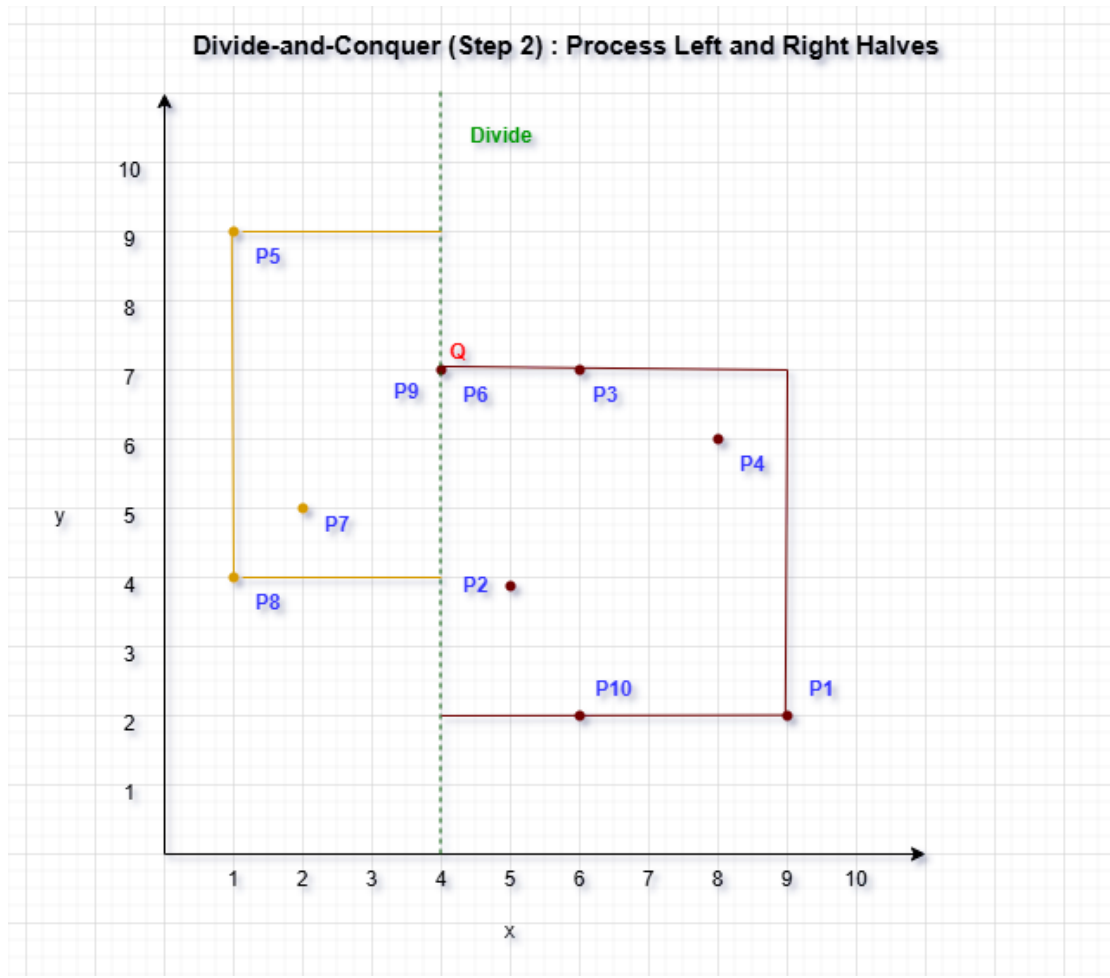
The first stage of the Divide-and-Conquer algorithm, which seeks to identify a query point's closest neighbor, is depicted in the image. Here, the query point  $Q(4, 7)$  and all of the facility points from P1 to P10 are shown on a 2D plane. The space is divided into two subregions, left and right, by a green dashed vertical line at  $x = 4$ . For recursively reducing the search space, this division is essential.



**Figure 21: Step 1 of the Divide-and-Conquer implementation**

The algorithm divides each point into two equal parts after sorting each point according to its x-coordinate. Points P5, P8, and P7, which are to the left of the vertical line, are included in the left subset, which is indicated in orange. The x-values for these sites are fewer than 4. The query point Q, along with P2, P10, P3, P4, and P1, are all included in the purple-colored right subset. These subsets all have x-values greater than or equal to 4.

This arrangement serves as the basis for the Divide-and-Conquer strategy. The method may process each half of the dataset independently by splitting it into two equal portions. The results can then be combined, and it will only cross the dividing line if required. The query point will be taken into account early in the nearest neighbor comparison if it is included in the right half.



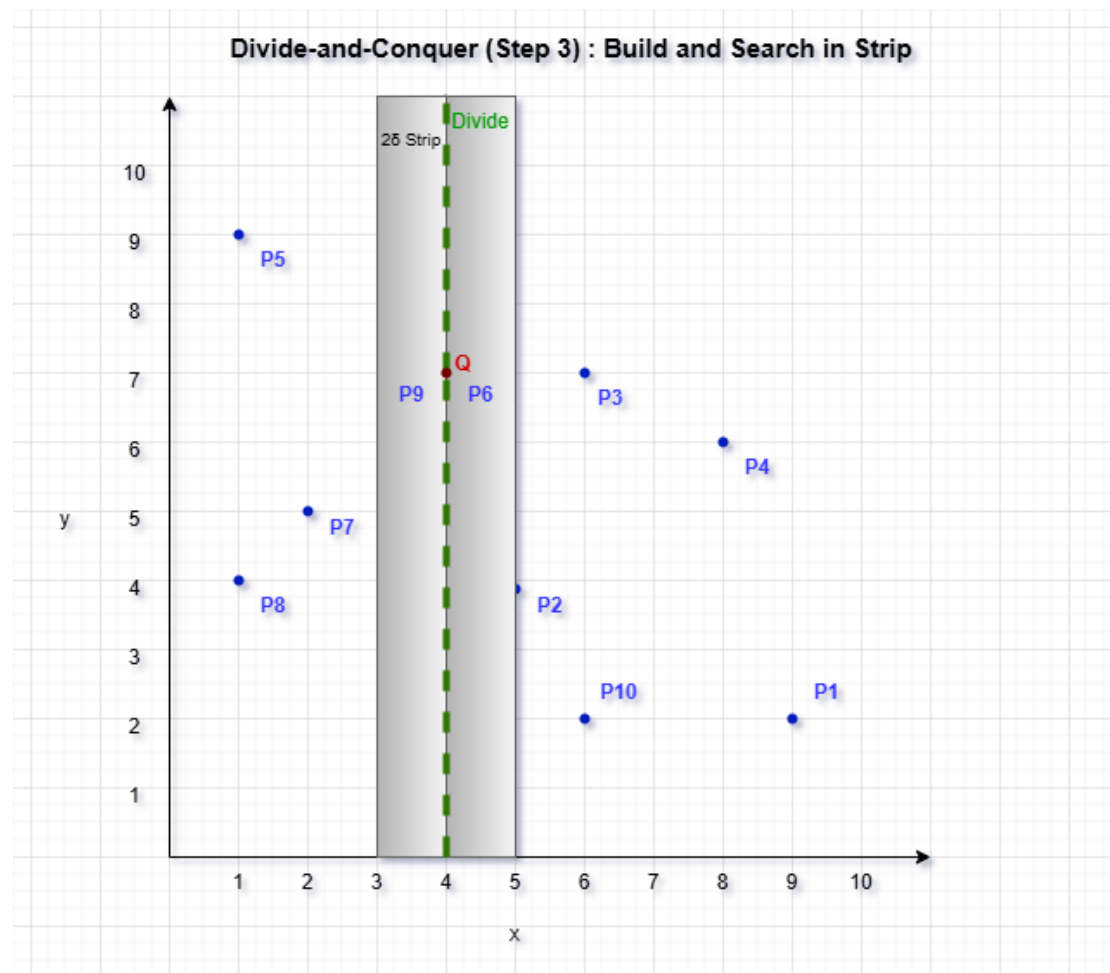
**Figure 22 : Step 2 of the Divide-and-Conquer approach**

Step 2 involves processing each half separately to find the closest neighbor after Step 1 splits the dataset using the median x-coordinate. The recursive processing stage of the Divide-and-Conquer method is represented by the image's two distinct rectangular sections, which are orange for the left half and purple for the right.

The left subset of points (P5, P8, P7, P6, and P9) is enclosed by the orange rectangle. These are the sites where the dividing line ( $x = 4$ ) is less than their x-coordinates. In contrast, the right subset is shown by the purple rectangle and consists of the query point Q, P2, P10, P3, P4, and P1—all of which have x-coordinates greater than or equal to 4. To find the local nearest neighbors, these regions are processed recursively.

The two subgroups are separated by the green dashed line at  $x = 4$ . It is crucial to take into account possible nearest neighbors from both sides of the divide since the query point Q(4, 7) is located precisely on this border. In order to guarantee global optimality, the algorithm might therefore still need to compare across the boundary after

identifying a candidate in one area.



**Figure 23 : Step 3 of the Divide-and-Conquer approach**

After identifying the nearest neighbors within each half, the Divide-and-Conquer algorithm proceeds to Step 3, which involves constructing a vertical strip region centered on the dividing line to check for potential closer pairs across the boundary.

In this case, the minimum distance ( $\delta$ ) found so far—between the query point Q(4, 7) and P9—is 1.0. Using this  $\delta$ , a gray shaded vertical strip of width  $2\delta$  (i.e., 2 units) is drawn around the green dividing line ( $x = 4$ ). This strip spans from  $x = 3$  to  $x = 5$ , as shown by the dashed gray boundaries.

Only the points within this shaded strip are considered for cross-boundary comparisons because any point outside the strip must be at least  $\delta$  units away in the x-direction alone, and thus cannot be closer than the current best. Since the query point Q lies inside this strip, it may be compared with nearby points from both sides—left and right—to ensure the global nearest neighbor is correctly identified.



**Conclusion :**

Sorting all facility location points by their x-coordinates and splitting them in half, left and right, at the median x-value is the first step in the Divide-and-Conquer method for nearest neighbor search. Two balanced spatial zones are created for recursive processing by this first division, as shown in Step 1. To find the point in that half that is closest to the query point  $Q(4, 7)$  in Step 2, each area is analyzed separately.

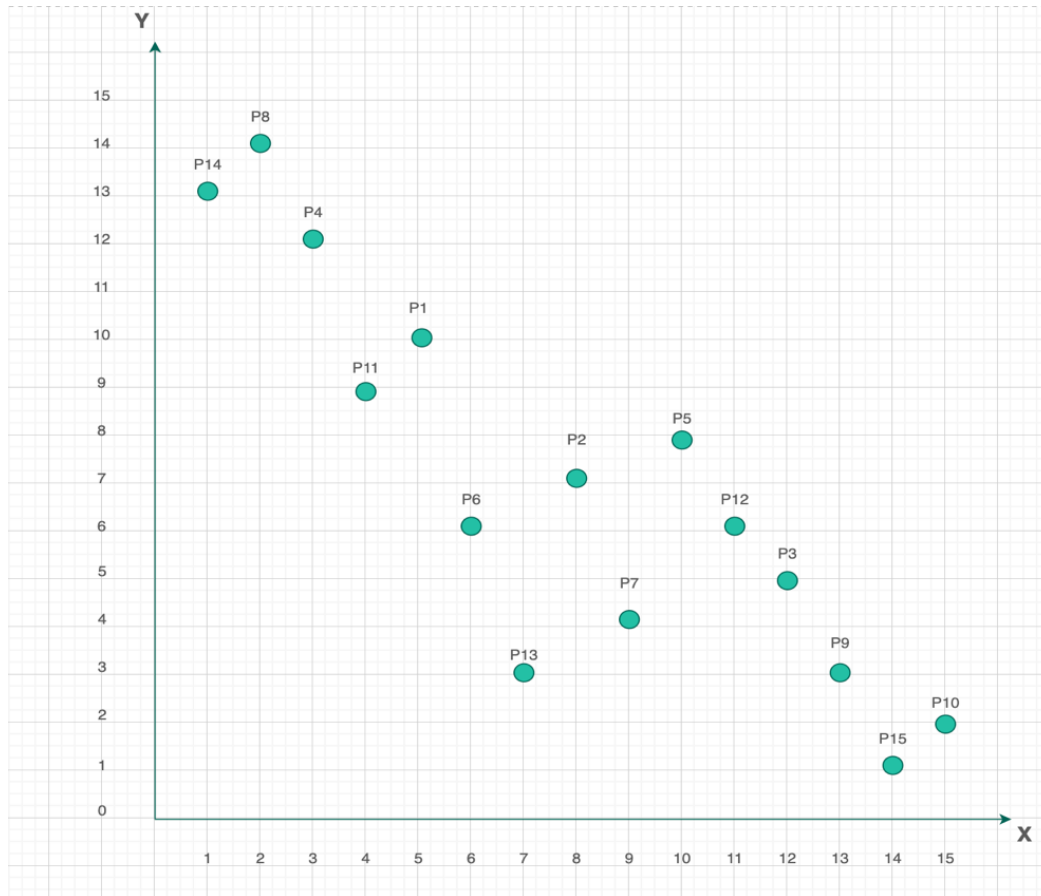
The query point may be relevant to both regions because it is situated exactly on the dividing line. Step 3 adds a vertical strip around the division with a width of  $2\delta$  after calculating the closest distances from both parts. Here,  $\delta$  is the current minimum distance discovered. Only spots near enough to potentially produce a better match across the boundary are included in the search thanks to this strip. The query point  $Q$  in this instance is inside the strip, and when compared to points on both sides, especially  $P_9$ , it can be seen that  $P_9(4, 7)$  is the closest neighbor, with a distance of 0.0. This approach effectively reduces the search space and shows how divide-and-conquer strategies can improve queries that use nearest neighbors, particularly when paired with spatial locality awareness.

## 5. Analyzing the BBS Algorithm based Skyline Search

Using the Branch and Bound Skyline (BBS) Algorithm in conjunction with R-tree, an effective spatial indexing technique, we have extracted the Skyline points from a set of city location data. Finding the finest city places that are undominated by any other point in all dimensions—that is, that offer the best options in terms of spatial attributes—was the main goal. The search for possible skyline candidates was greatly accelerated by the ordered spatial data storage made possible by the R-tree structure.

We have extracted the Skyline points from a set of city location data using the Branch and Bound Skyline (BBS) Algorithm in combination with R-tree, an efficient spatial indexing technique. The major objective was to identify the greatest city locations that offer the best possibilities in terms of spatial features, that is, locations that are not dominated by any other point in all dimensions. The R-tree structure allowed for ordered spatial data storage, which significantly accelerated the search for potential skyline prospects.

<b>ID</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>4</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>	<b>15</b>
<b>X</b>	<b>5</b>	<b>8</b>	<b>12</b>	<b>3</b>	<b>10</b>	<b>6</b>	<b>9</b>	<b>2</b>	<b>13</b>	<b>15</b>	<b>4</b>	<b>11</b>	<b>7</b>	<b>1</b>	<b>14</b>
<b>Y</b>	<b>10</b>	<b>7</b>	<b>5</b>	<b>12</b>	<b>8</b>	<b>6</b>	<b>4</b>	<b>14</b>	<b>3</b>	<b>2</b>	<b>9</b>	<b>6</b>	<b>3</b>	<b>13</b>	<b>1</b>

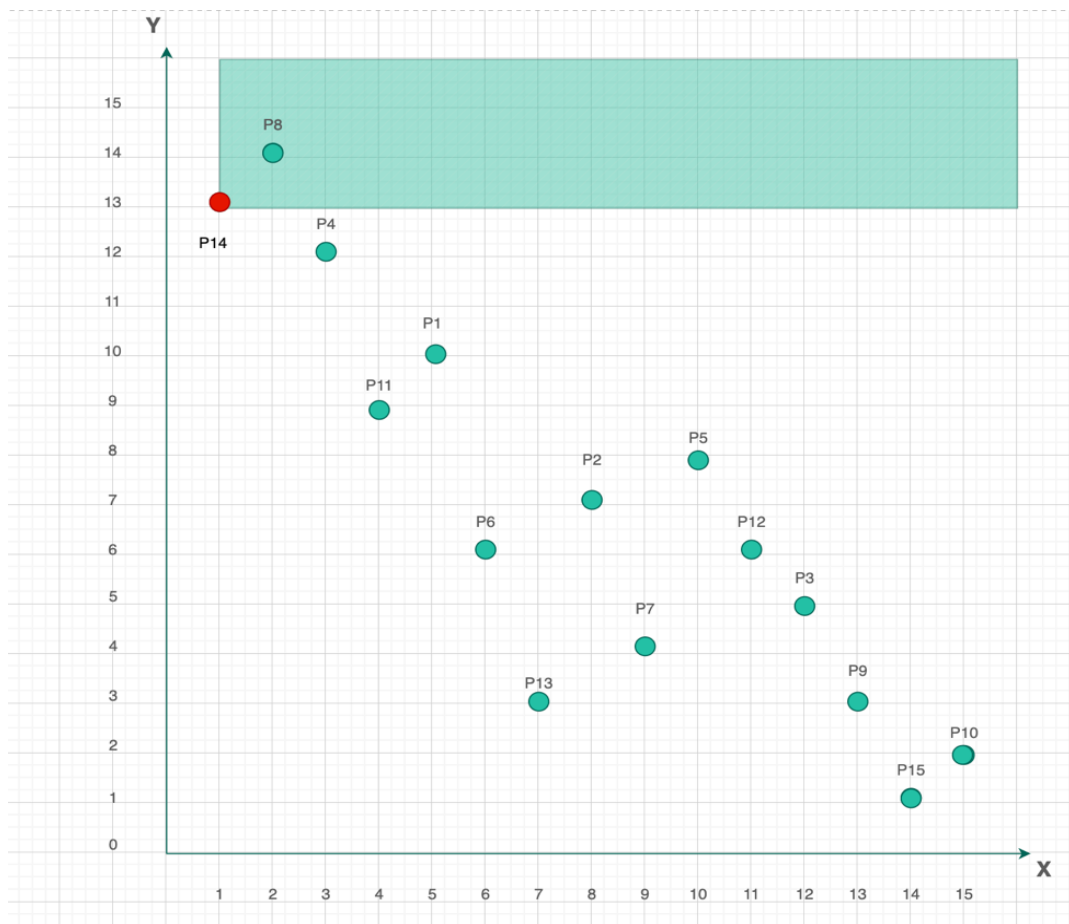


**Figure 24 : Skyline Search Dataset Visualization**

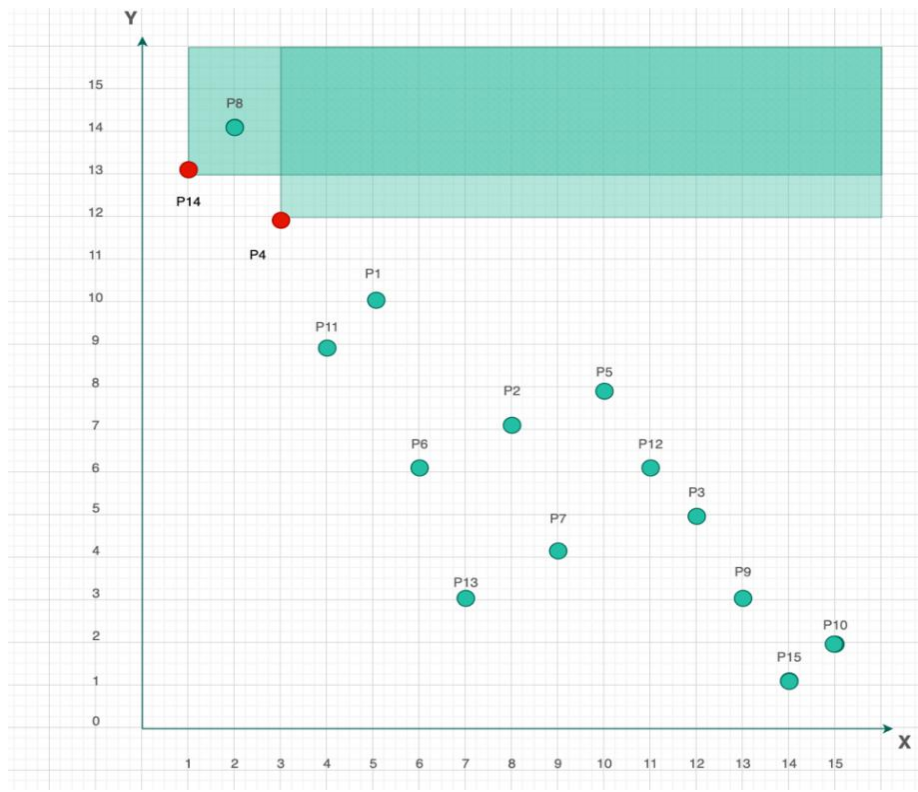
## 5.1 R-Tree Construction Process

A number of illustrations showing how to construct an R-tree demonstrates a methodical way to arrange spatial data for effective query performance, particularly in skyline queries. Potential skyline sites in the early stages, including P1, P2, P5, P12, and P15, are indicated in green, meaning that important characteristics like cost and distance do not dominate them. These green points continue to be surrounded by green-shaded bounding rectangles as the tree grows, indicating their continued significance to the skyline. During searches, these dynamically adjusting rectangles help to focus the search on significant areas by firmly enclosing just the pertinent non-dominated spots.

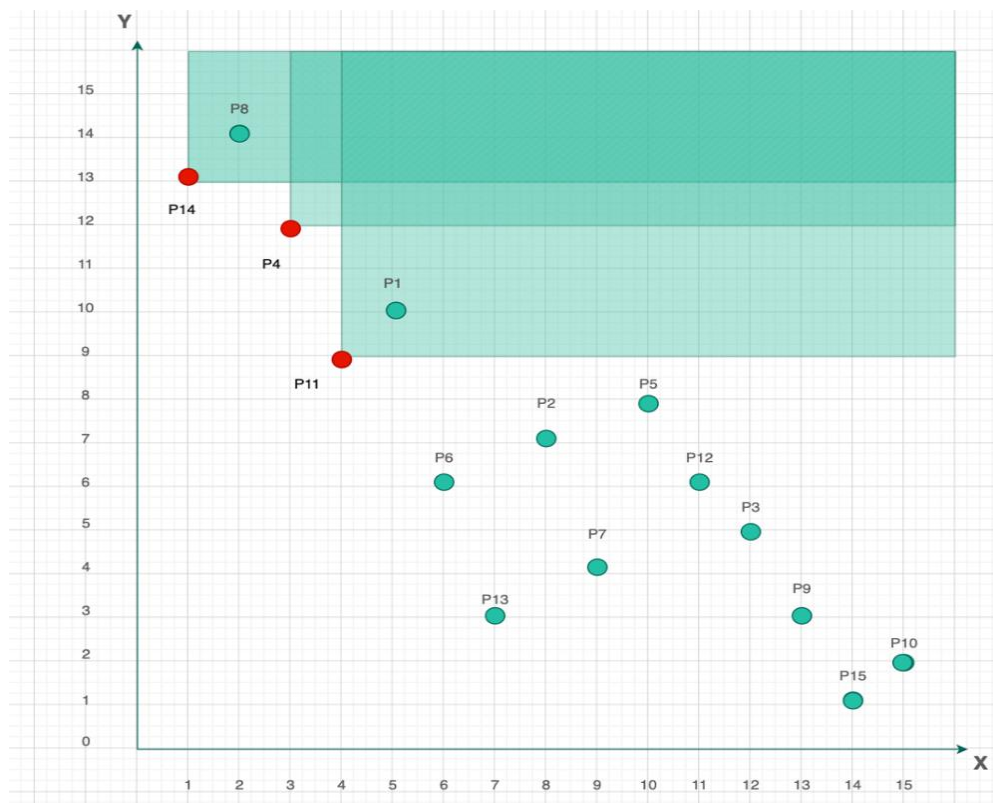
On the other hand, dominated entries are represented by red points that show up later in the diagrams, such as P6, P11, P13, and P14. The algorithm's capacity to efficiently prune unqualified data is demonstrated by the exclusion of these points from additional skyline evaluation when the R-tree structure changes. Because it restricts processing to only those points that could contribute to the final skyline set, this pruning feature is essential to the R-tree's effectiveness.



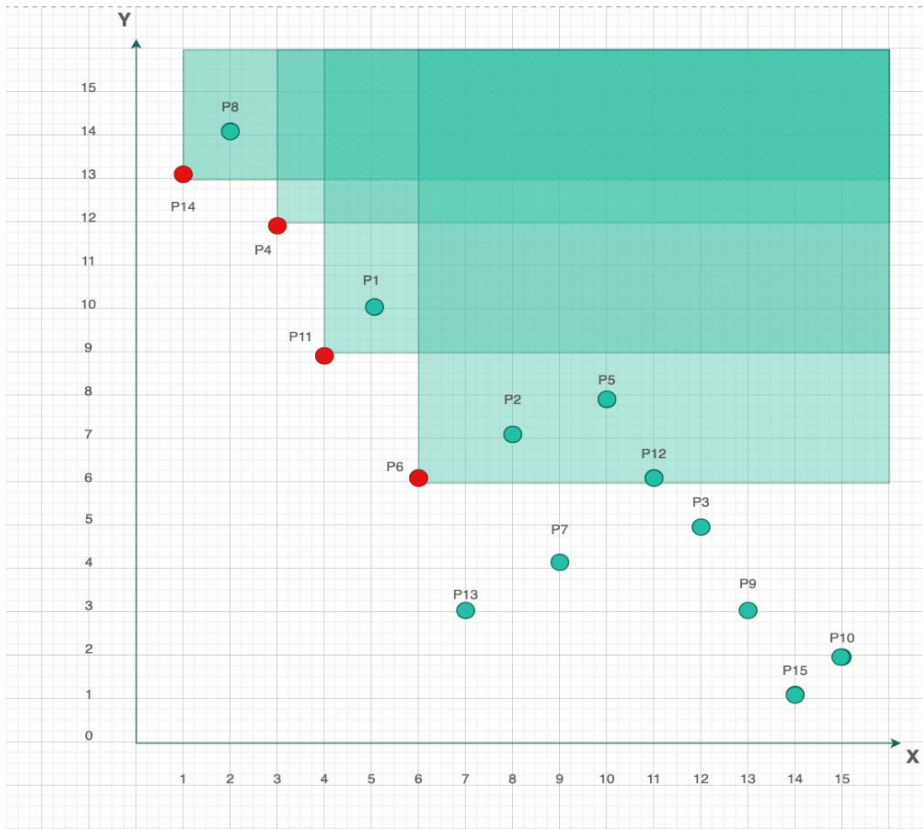
**Figure 25 : Step 1 - Skyline Coordinates from Left to Right**



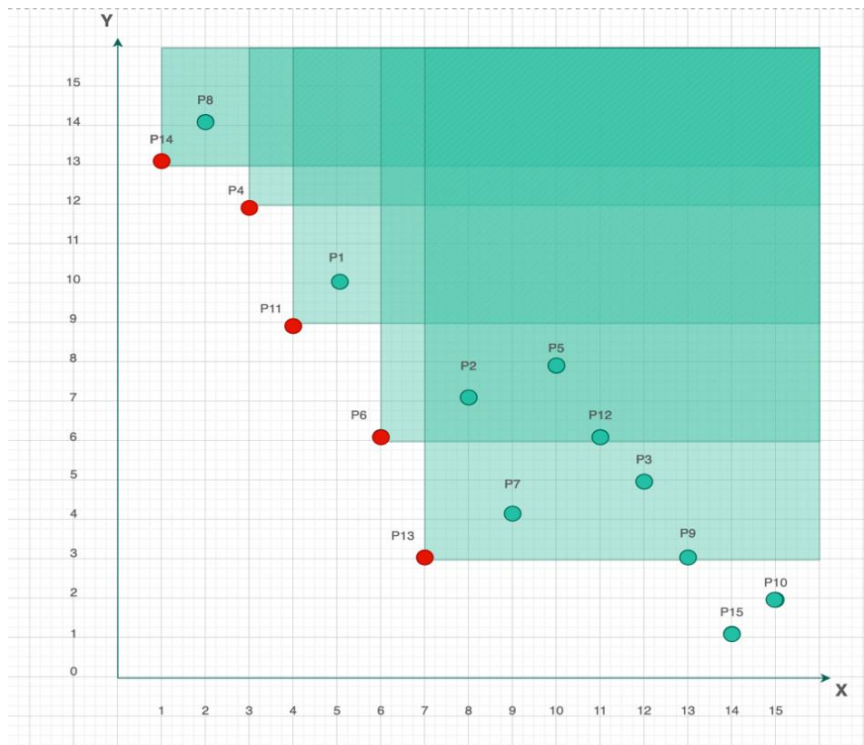
**Figure 26 : Step 2 - Skyline Coordinate points from left to right**



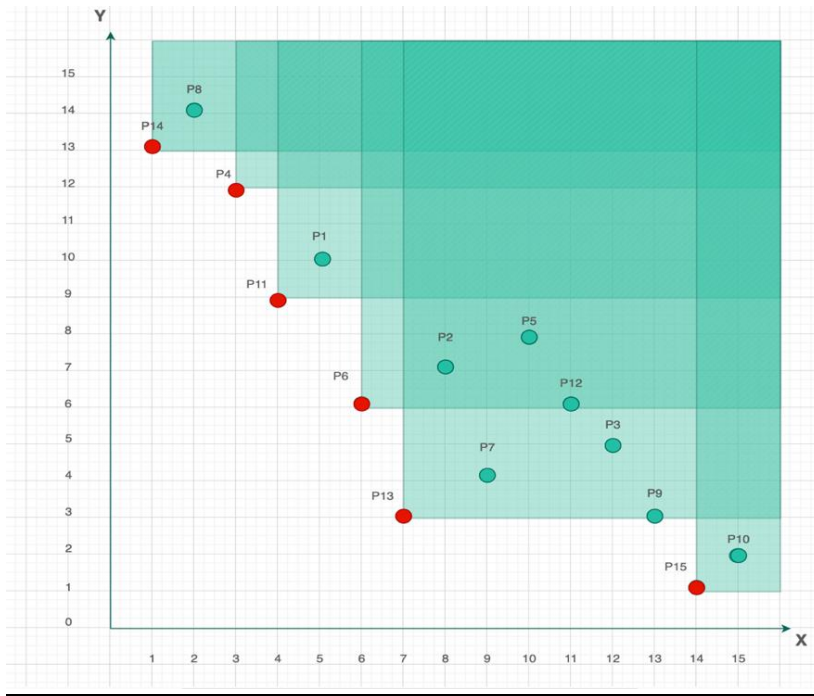
**Figure 27 : Step 3 - Skyline Coordinates from left to right**



**Figure 28 : Step 4 - Skyline Coordinates from left to right**



**Figure 29 : Step 5 - Skyline Coordinates from left to right**



**Figure 30 : Step 6 - Skyline Coordinates from left to right**



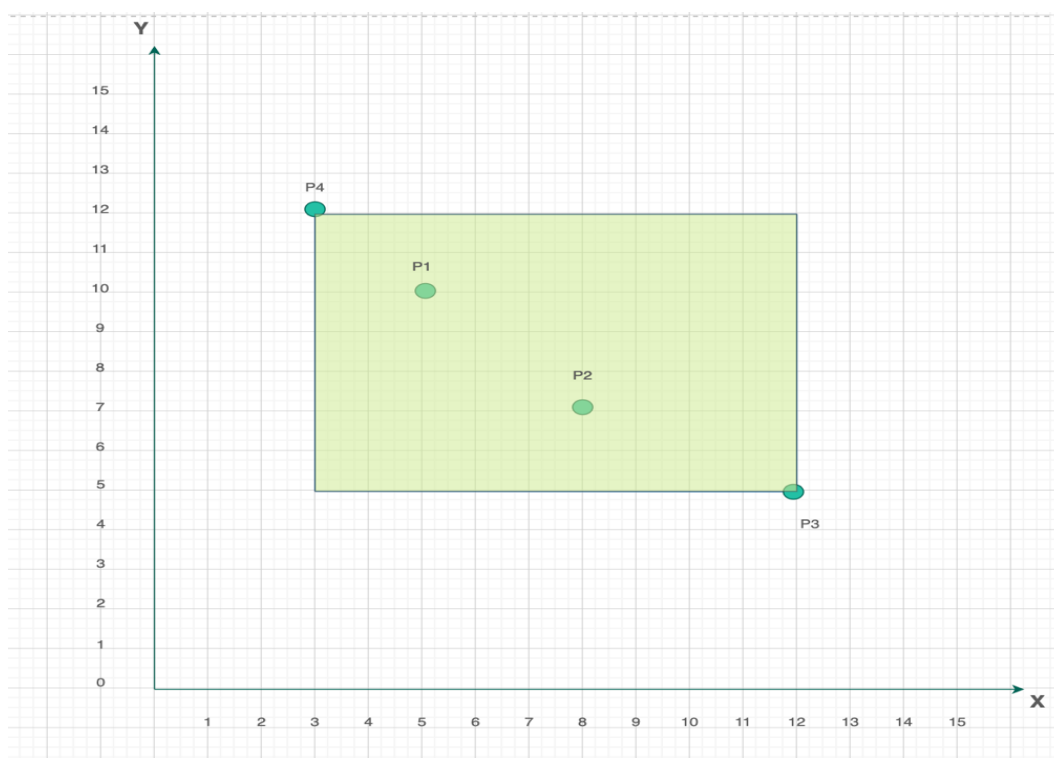
**Figure 31 : Step 7 - Skyline Coordinates from left to right**

Red dotted lines in the last stage show changes made to fine-tune node boundaries as each diagram gradually updates the R-tree structure. In order to maintain the efficiency of the tree structure, these improvements seek to maximize space partitioning and coverage. The R-tree preserves a balanced hierarchy with little node overlap by precisely defining boundaries and only including the most pertinent points. By focusing on the most promising, non-dominated locations, this targeted organization improves the performance of spatial queries and eliminates needless calculations.

## 5.2 BBS (Branch and Bound Search) Algorithm Process

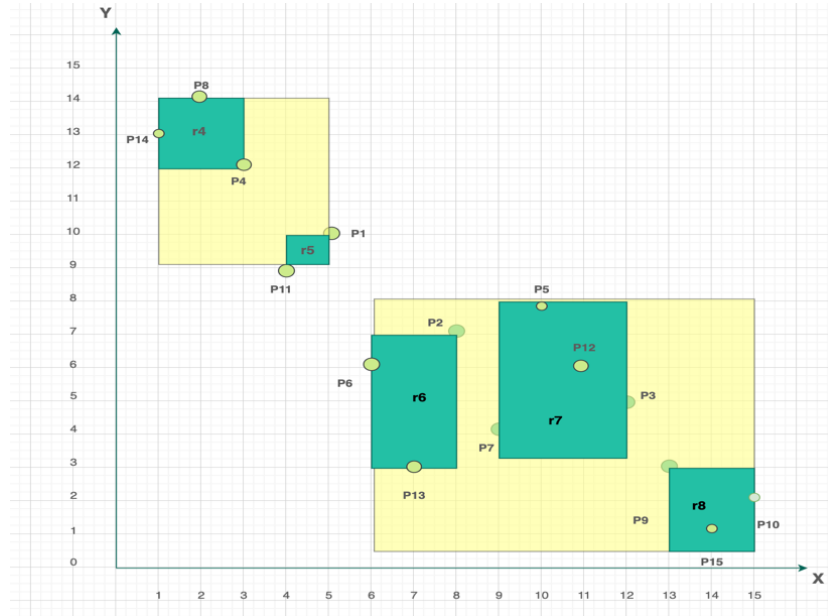
The early phase of R-tree creation utilizing the Minimum Bounding Rectangle (MBR) technique is depicted in the picture below. Points P1, P2, P3, and P4 are now all included in a single, sizable MBR. This first grouping is crucial because it establishes the framework for subsequent spatial subdivision.

The big green rectangle shows how the R-tree starts by combining all of the data points into a single, wide area, giving it a cohesive structure that it may then further enhance. Since it establishes the initial spatial bounds within which the skyline (non-dominated) locations will be searched, this stage is particularly crucial for the Branch and Bound Search (BBS) algorithm. The R-tree exhibits its efficiency by treating all enclosed points as initial skyline candidates inside a single bounding region, hence eliminating the need for excessive spatial checks. This is accomplished by beginning with all points enclosed in a single MBR.





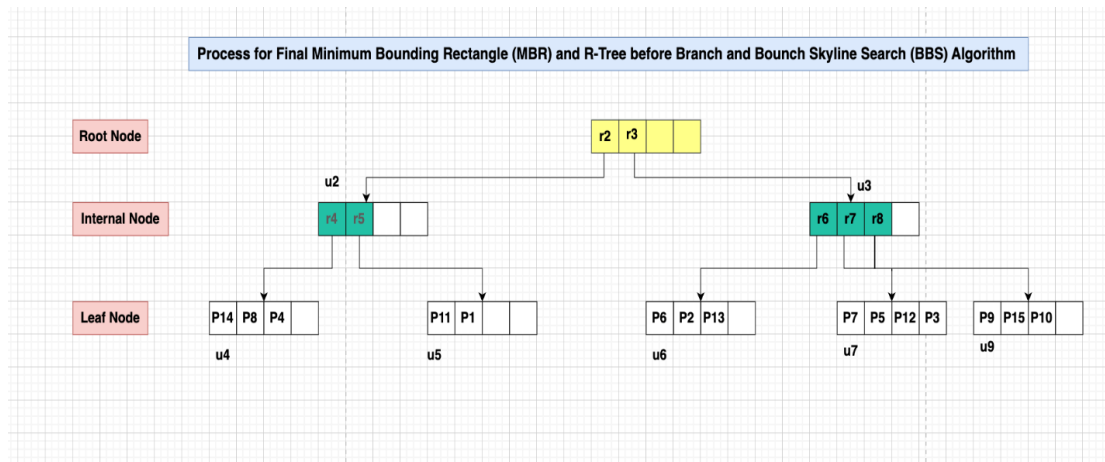
**Figure 32 : Minimum Bounding Rectangles and R-Tree Process**



**Figure 33 : MBR and R-Tree Process**

By dividing into smaller MBRs, each of which contains subsets of spatially closer points, the R-tree structure in the above image starts to refine itself. The hierarchical structure of the R-tree, in which larger regions are methodically divided to more precisely represent spatial closeness, is reflected in these new rectangles, designated r4, r5, r6, and r7. Rectangle r4 now encompasses, for instance, the nearby points P4, P8, and P14; this grouping efficiently reduces the search area when doing spatial operations such as closest neighbor or skyline inquiries.

Each smaller rectangle is the result of a purposeful approach to reduce pointless comparisons and increase query efficiency. The structure streamlines the process by allowing queries to concentrate on focused clusters rather than analyzing every point worldwide. By prioritizing pertinent regions and avoiding repeated evaluations throughout the whole dataset, this segmentation is essential for the Branch and Bound Skyline (BBS) algorithm, which allows for faster identification of skyline points.

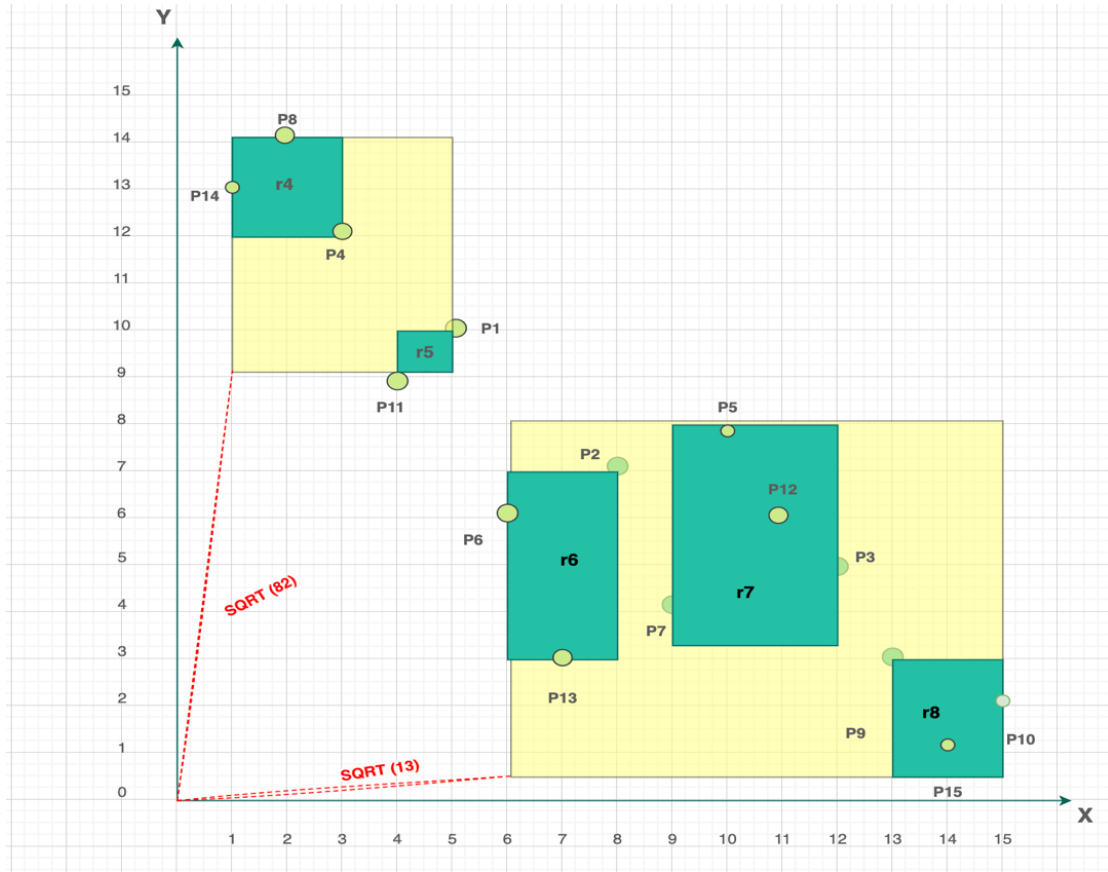


**Figure 34 : Root Node, Internal Node and Leaf Node**

The finalized R-tree structure is shown in the above figure, which effectively arranges geographic data for optimal querying by displaying a distinct hierarchy from the root node down to the leaf nodes. Internal nodes like u2 and u3 branch off from the root node to oversee larger MBRs like r4, r5, r6, r7, and r8, each of which covers distinct point groups. Actual data points like P14, P8, and P4 are found in leaf nodes like u4 at the lowest level, whereas P11 and P1 are found in u5. P6, P2, and P13 are among the points found in r6, whereas other rectangles organize their own subsets.

In order for the Branch and Bound Skyline (BBS) algorithm to effectively exclude areas that do not contribute to possible skyline points and swiftly focus on pertinent regions, this hierarchical structure is essential. In the last phase of query execution, leaf nodes—the most granular level in the tree—are taken into account, enabling exact comparisons only when required.

This concept enhances the speed and accuracy of skyline and spatial inquiries by drastically shrinking the search space. Fast and scalable retrieval is ensured by the BBS algorithm's integration with the R-tree structure, which makes it especially helpful in spatial databases and geographic information systems (GIS). By focusing on the most promising data locations, it expedites the process of picking skyline points and improves query performance overall.



**Figure 35 : Distance from Origin to Root Node**

In order to create a sorted priority list of its child Minimum Bounding Rectangles (MBRs) according to their Euclidean distances from the origin (or the reference point used in the skyline computation), the Branch and Bound Skyline (BBS) algorithm first accesses the root node of the R-tree.

The distances are computed using the standard Euclidean distance formula:

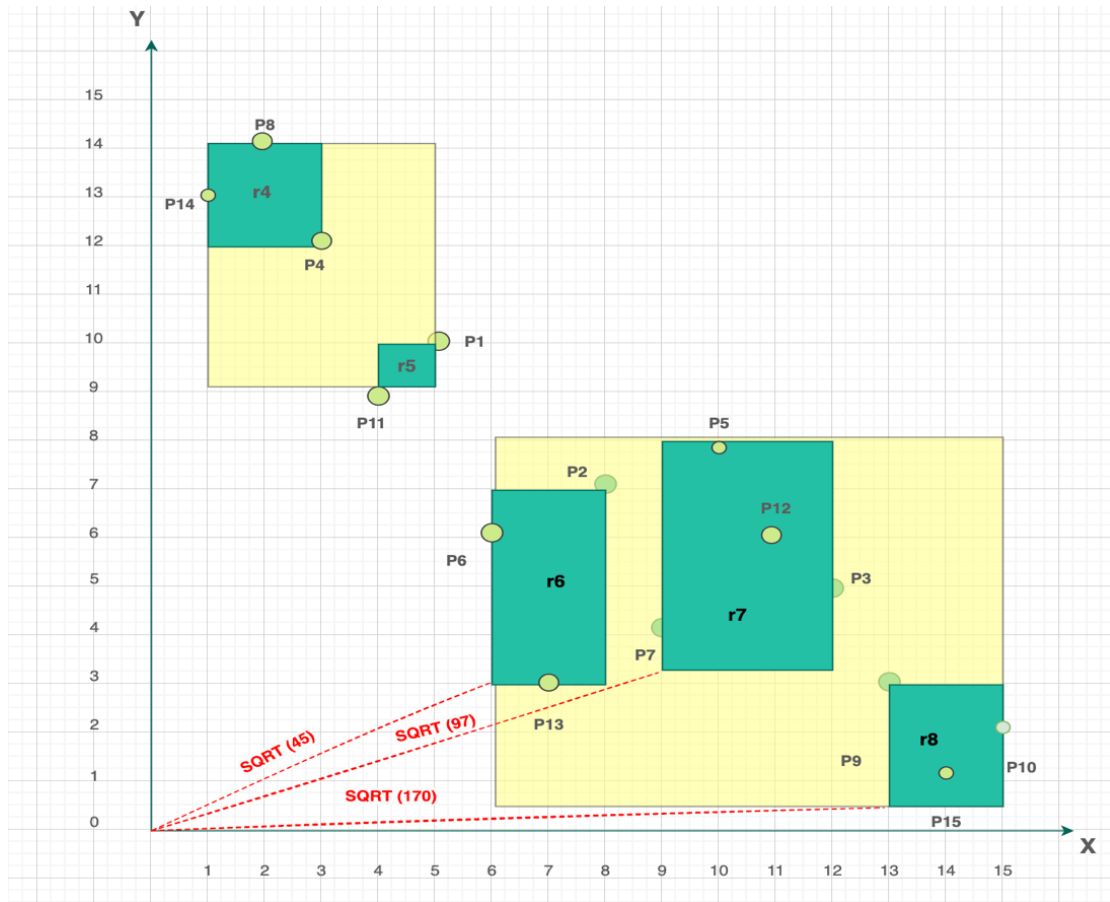
$$\text{Distance to } r2 = \sqrt{1^2 + 9^2} = \sqrt{82}$$

$$\text{Distance to } r3 = \sqrt{6^2 + 1^2} = \sqrt{13}$$

Based on these calculated values, the initial sorted list of MBRs becomes:

$$\{ (r3, \sqrt{13}), (r2, \sqrt{82}) \}$$

This ordering ensures that the BBS algorithm explores the most promising (i.e., closest) regions first, improving the efficiency of skyline point detection.



**Figure 36 : Distance from Origin to u3 Child Node**

Next, the BBS algorithm visits node u3, which leads to the addition of more MBRs into the sorted list based on their Euclidean distances from the reference point. The distances for the newly encountered MBRs are calculated as follows:

$$\text{Distance to } r6 = \sqrt{(6^2 + 3^2)} = \sqrt{45}$$

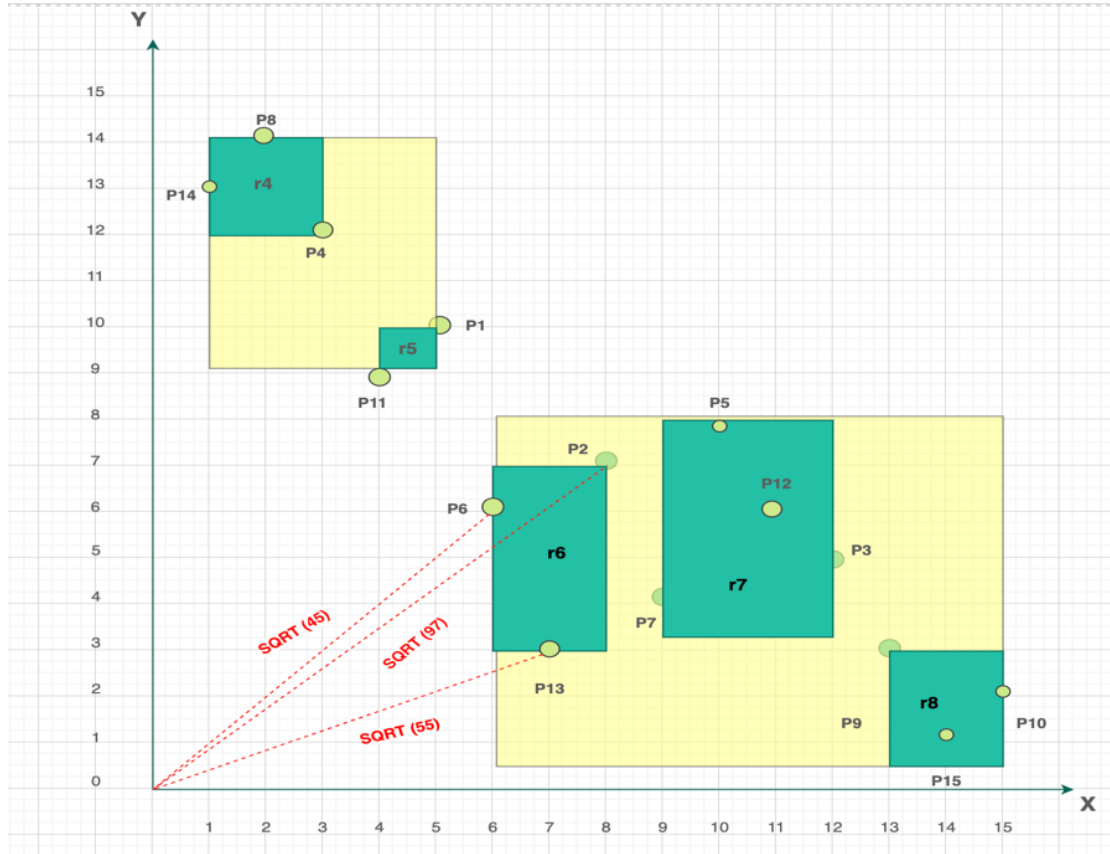
$$\text{Distance to } r7 = \sqrt{(9^2 + 4^2)} = \sqrt{97}$$

$$\text{Distance to } r8 = \sqrt{(13^2 + 1^2)} = \sqrt{170}$$

With these new entries, the updated sorted list becomes:

$$\{ (r6, \sqrt{45}), (r2, \sqrt{82}), (r7, \sqrt{97}), (r8, \sqrt{170}) \}$$

By prioritizing MBRs that are spatially nearby, this method enables the algorithm to efficiently reduce the size of the search area. The BBS algorithm increases the overall performance of the skyline query by identifying skyline points more quickly and avoiding pointless comparisons by first examining the closest bounding rectangles.



**Figure 37 : Distance from Origin to r6 Child Node**

As the BBS algorithm proceeds to the leaf node of r6, it encounters three data points and computes their Euclidean distances from the origin (or reference point). These distances are:

$$P13 = \sqrt{(7^2 + 3^2)} = \sqrt{58}$$

$$P6 = \sqrt{(6^2 + 6^2)} = \sqrt{72}$$

$$P2 = \sqrt{(8^2 + 7^2)} = \sqrt{113}$$

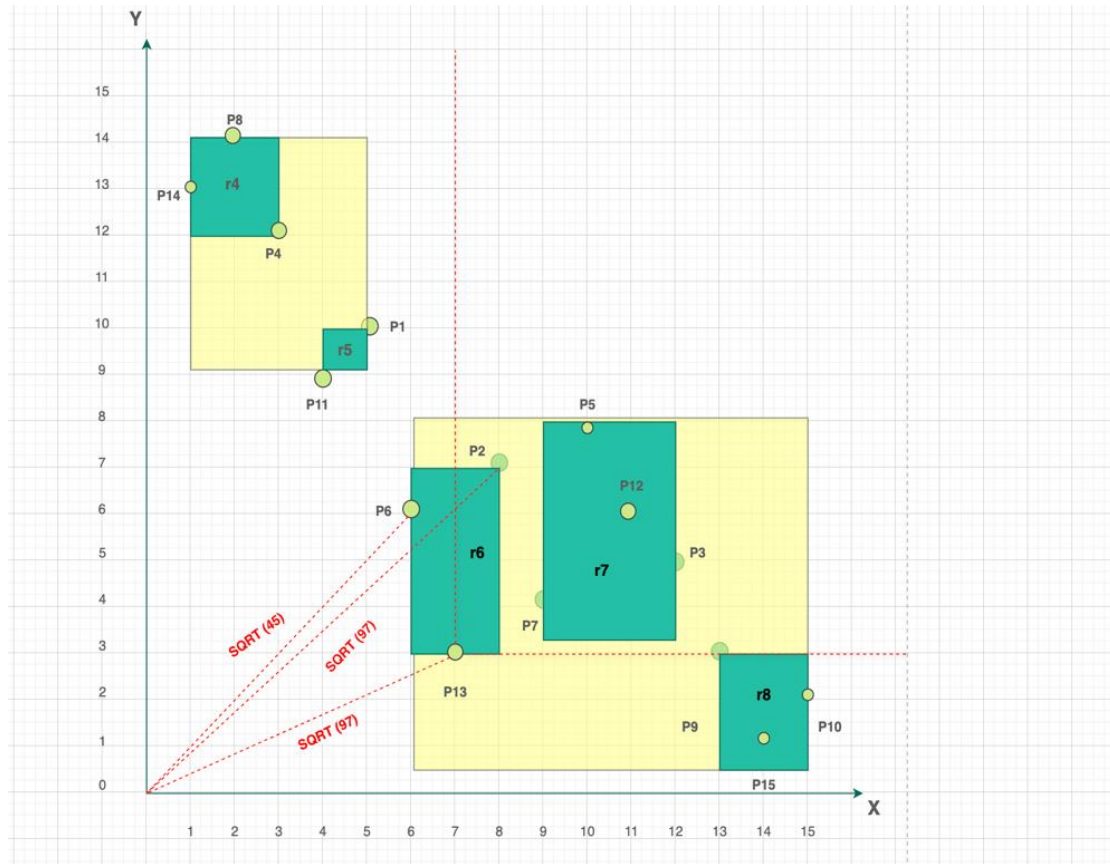
The updated sorted list now becomes:

$$\{ (P13, \sqrt{58}), (P6, \sqrt{72}), (r2, \sqrt{82}), (r7, \sqrt{97}), (P2, \sqrt{113}), (r8, \sqrt{170}) \}$$

P13 and P6 are chosen as the first and second skyline points from this list since, in terms of both dimensions (e.g., lower cost and better proximity), no earlier points surpass them. All of the points in r7 are found to be dominated by either P6 or P13 when compared to MBR r7; hence, r7 is pruned from further evaluation.

Furthermore, because P2 is dominated by both P6 and P13, it is also eliminated from the list of candidates. By removing any unimpressive entries, this phase improves the skyline candidate

set and guarantees that the algorithm only moves on with the most pertinent points. Performance is greatly enhanced by this trimming technique, which increases the efficiency of the skyline evaluation in the next stages.



**Figure 38 : X-coordinates and Y-coordinates (after P13)**

The algorithm then moves on to decompose internal node r2, which is the next closest MBR in the sorted list based on Euclidean distance. Upon expanding r2, its child nodes r5 and r4 are evaluated and inserted into the sorted list. The distances are calculated as:

$$\text{Distance to } r5 = \sqrt{(4^2 + 9^2)} = \sqrt{107}$$

$$\text{Distance to } r4 = \sqrt{(1^2 + 12^2)} = \sqrt{145}$$

This results in an updated sorted list:

$$\{ (r5, \sqrt{107}), (r4, \sqrt{145}), (r8, \sqrt{170}) \}$$

This recursive procedure is carried out by the algorithm, which evaluates and breaks down the remaining internal nodes according to their proximity. Every stage entails determining if a node or point is dominant and can be pruned, or if it contributes to the skyline. The last skyline points—P14, P4, P11, P6, P13, and P15—are finally determined by the algorithm using this effective prioritization and removal technique.

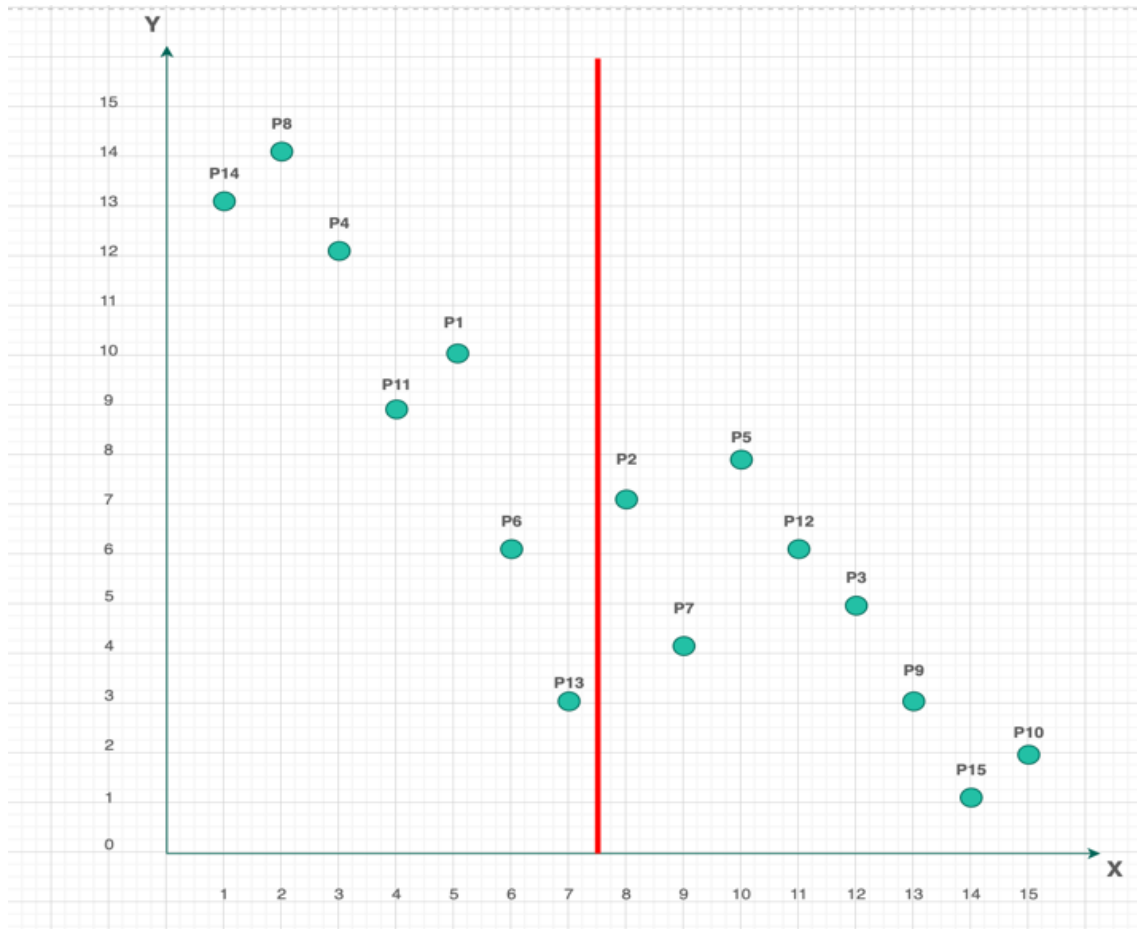
This method guarantees that the Branch and Bound Skyline (BBS) algorithm effectively narrows the search to the most pertinent, non-dominated points—optimizing both accuracy and performance in skyline query processing—by methodically separating only the most promising areas and eliminating dominated entries early.

### **5.3 Divide and Conquer Process :**

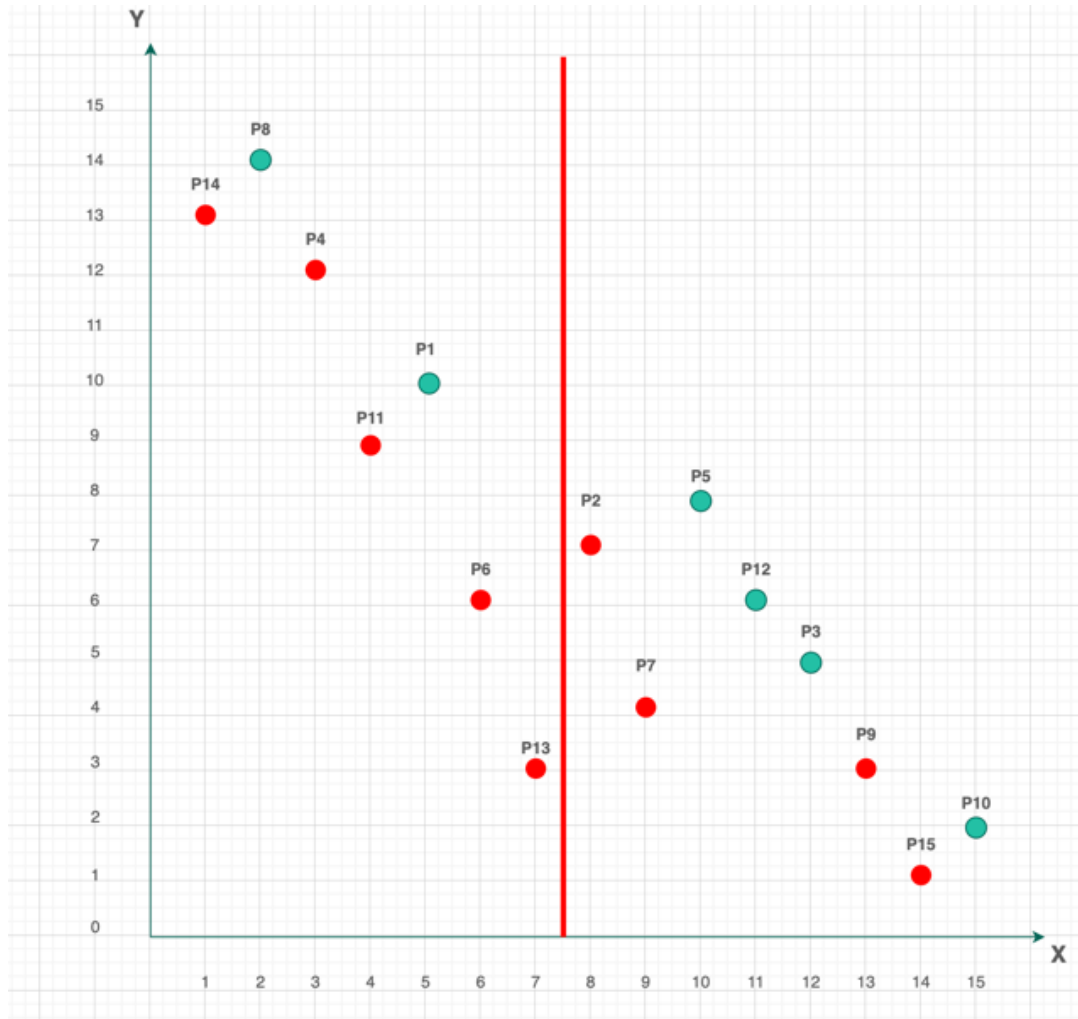
By decreasing the overall processing complexity, the divide-and-conquer strategy is utilized to calculate skyline points effectively. By reducing the greatest x-axis value (15) in half, the dataset is first divided into two sections, yielding a threshold of 7.5. In light of this division:

The following points are part of the left subset (A): {P14, P8, P4, P11, P1, P6, P13}  
The following points are part of the correct subset (B): {P2, P7, P5, P12, P3, P9, P15, P10}

Each subset's skyline points are then determined separately, and the outcomes are then combined to create the final skyline. By restricting comparisons to smaller groups before combining the results—a procedure that is clearly depicted in the accompanying figures—this approach drastically cuts down on processing time and complexity.



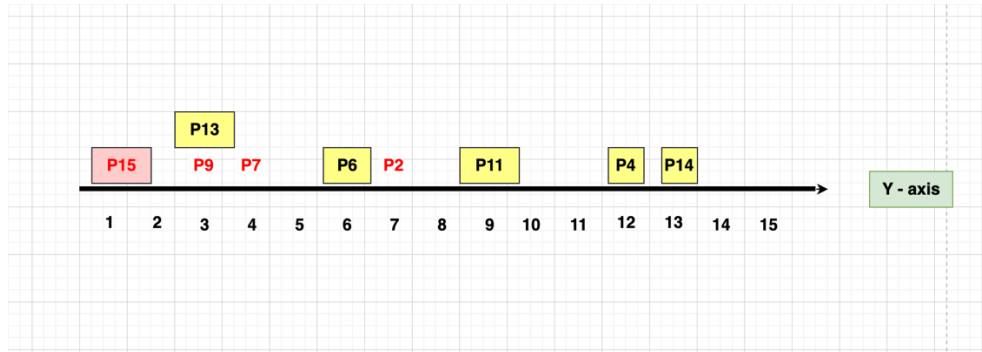




**Figure 39 : Skyline points shown using Divide and Conquer Approach in X-axis**

The significant points from both groups are determined when the dataset is divided and skylines are calculated within each subset. The resulting skyline candidates for the left subset (A) are {P14, P4, P11, P6, P13}. Due of their lack of dominance within their group, these points are kept as possible final skyline members.

The intermediate skyline set for the right subset (B) is {P2, P7, P9, P15}. A 1D dominance check along the y-axis is carried out to finish the skyline by contrasting points from subsets A and B. With the exception of P15, which is left undominated, P13, which has a lower y-coordinate, dominates every point in B throughout this screening. Therefore, in addition to the skyline points from the left subset, P15 is the only point from the right subset that is eligible to be a final skyline point.



**Figure 40 : Skyline Points sorting in Y-axis**

Hence, the final set of skyline points is determined to be (P14, P4, P11, P6, P13, P15).