MACQUARIE University
SYDNEY·AUSTRALIA

COMP6210: BIG DATA

ASSIGNMENT 2: R-TREE IMPLEMENTATION

Name: Mai Vy (Vivian) Nguyen

Student ID: 47554029

Table of Contents

1. Program execution requirements

1.1. Program environment

OS Environment	MacOS
CPU	Apple M2 Pro
RAM	16GB
Language	Python
Version	Python 3.11.4
Software	Visual Studio Code

1.2. Input files and parameters

- Input files:
- “R_tree_Contruction.txt”: dataset (points) file
- “200Range.txt”: range query file
- The setting of the directory: in one folder

2. Program documentation

2.1. Program organisation

File Name	Description
“Squential_Scan.py”	The file script processes spatial range queries on a set of data points. It reads points from "R_tree_construction.txt" and 200 queries from "200Range.txt", performs a sequential scan to count points within each query range, and outputs the results, along with total and average query processing times, to "Sequential_output.txt".
“Rtree.py”	The file reads points from "R_tree_construction.txt" and 200 queries from "200Range.txt". Then the data is split into two sections to construct two separate R-trees using point-based division. For each specified range in

	"200_range.txt", it traverses the respective R-trees to count the points within these ranges, outputting the totals to "Rtree_output.txt".
“Rtree_DC.py”	Same with “Rtree.py”. However, in Divide and Conquer, the point set is divided into two halves, and then two separate R-trees are constructed for these subsets, allowing for more efficient organization and query processing of spatial data.

Class	**Description**
“Node”	Class(Node) in an R-tree serves as a building block for the tree structure, encapsulating attributes such as a unique identifier, a collection of child nodes, and the node's Minimum Bounding Rectangle (MBR). For leaf nodes, it additionally holds a list of data points, while all nodes maintain a reference to their parent node, facilitating efficient tree traversal.
“RTree”	Class(RTree) encompasses functions crucial for constructing and managing the R-tree structure, including methods to insert data points, handle node overflows, and compute Minimum Bounding Rectangles (MBRs) for efficient spatial querying.

2.2. Function description

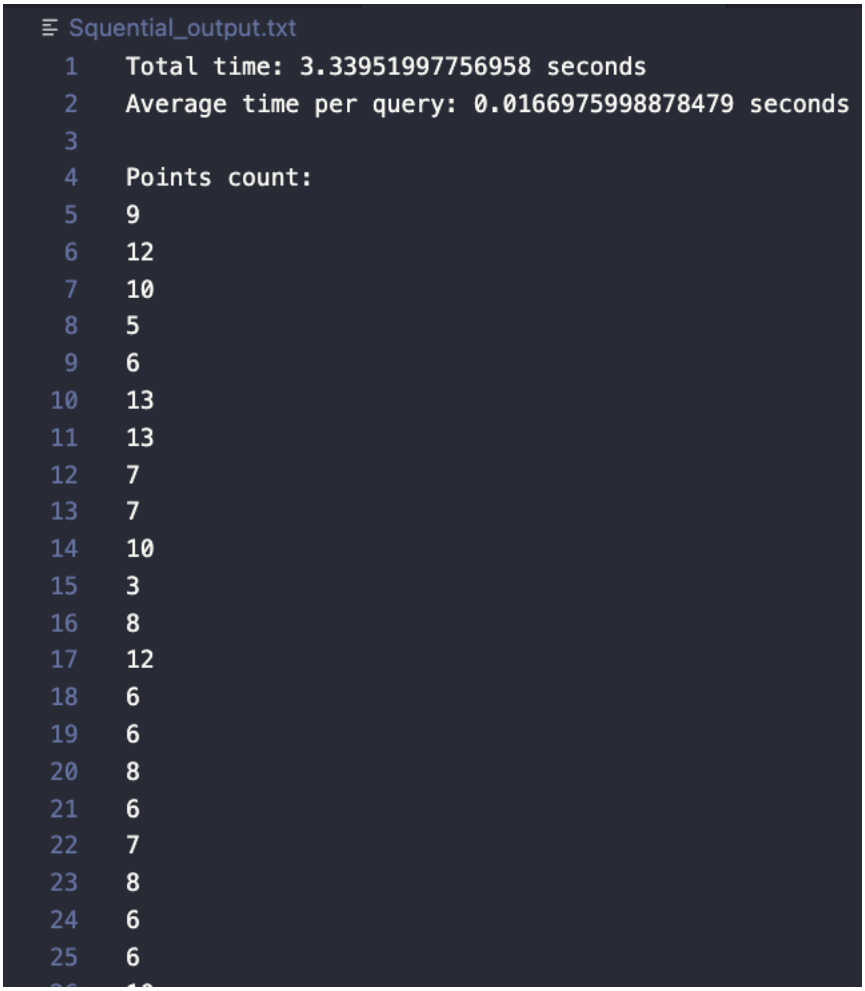
Function Name (parameters)	Description (detailed information)
__init__ (self)	Initialize a new instance of the “Node” or “Rtree”.
perimeter(self)	Calculates and returns the half perimeter of the node’s MBR.
is_overflow(self)	Check if the node has more child nodes or data points than a specified limit “B=4” • For leaf nodes: check if the length of “data_points” exceeds “B=4”. • For internal nodes: check if the length of “child_nodes” exceeds “B=4”.
is_root(self)	Determine if the node is root by checking if its parent is “None”. A node is the root if it has no parent (“self.parent” is “None”).
is_leaf(self)	Identify if the node is a leaf node, indicated by having no child nodes.
query(self, node, query)	Execute a query on the R-Tree starting from the given note. • For leaf nodes: check each point directly. • For internal nodes: check if the child’s MBRs intersect with the query range.
is_covered(self, point, query)	Check if a given “point” lies within bounds of a “query” rectangle.

is_intersect(self, node, query)	Determine if the MBR of a node intersects with a query rectangle, using center points and dimensions of the rectangles.
insert(self, u, p)	Insert a data point “p” into the R-tree, starting at node “u”. If a node overflows because of the insertion, it triggers overflow handling.
choose_subtree(self, u, p)	Select the most appropriate child node of “u” to insert a new point “p” to minimize the increase in perimeter after insertion.
peri_increase(self, node, p)	Calculate the increase in the perimeter of a node’s MBR if a new point “p” is added to it.
handle_overflow(self, u)	Handle the overflow condition in a node “u” by splitting it into two nodes (“u1” and “u2”) If “u” is the root, a new root is created If “u1” and “u2” are added as child to parent
split(self, u)	Split an overflowing node “u” into two separate nodes (“best_s1” and “best_s2) If “u” is a leaf node, its data points are sorted by x and y coordinates. If “u” is an internal node, it sorts “child_nodes” by the corners of MBR
add_child(self, node, child)	Add a child node to a given node and updates the MBR of the parent node accordingly
add_data_point(self, node, data_point)	Add a data point to a node and update the node’s MBR to encompass the new point.

Update_mbr(self, node)	Update the MBR of a node, recalculating it based on the current child nodes on data points.

2.3. Program results

Sequential-Based Method:



R-Tree Based Method:

Rtree_output.txt

```
The time-cost of building up the R-Tree is: 18.826796770095825 seconds.
The query processing time is: 0.02881312370300293 seconds.
The average query processing time per point is: 0.0001440656185150146 5 seconds.
Points count:
9
12
10
5
6
13
13
7
7
10
3
8
12
6
6
8
6
7
8
6
6
```

Divide-and-Conquer:

Rtree_DC_output.txt

```
The time-cost of building up the R-Tree is: 19.630218982696533 seconds.
The query processing time is: 0.018118858337402344 seconds.
The average query processing time per point is: 9.059429168701172e-05 seconds.
Points count:
9
12
10
5
6
13
13
7
7
10
3
8
12
6
6
8
6
7
8
6
6
```

Utilizing an R-tree-based approach for search operations results in an average query processing speed that is 116 times quicker than the Sequential-Based Method. Moreover, incorporating Divide and Conquer into the R-tree-based search further reduces the average query time by 37% compared to using the R-tree method alone.

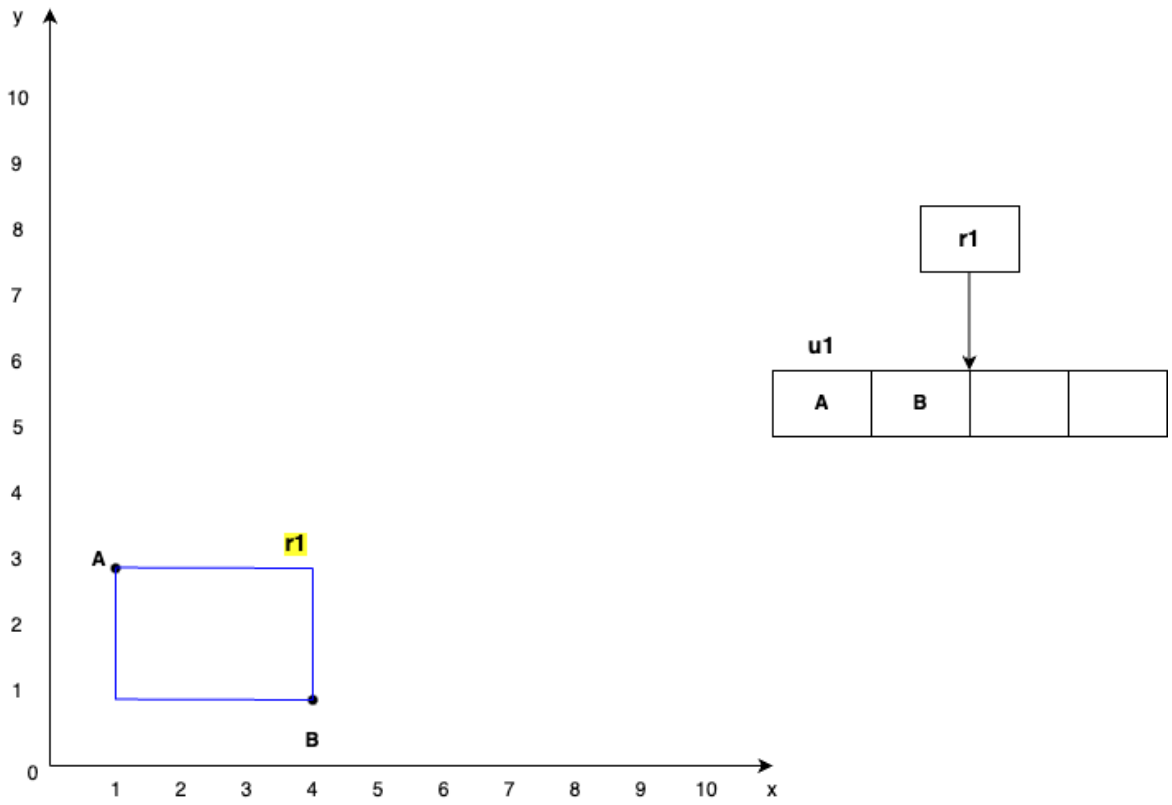
3. Analysing the Working of R-Tree

3.1. R-Tree establishment

3.1.1 R-Tree Insertion

Take B = 4 so we have 4 points in each node

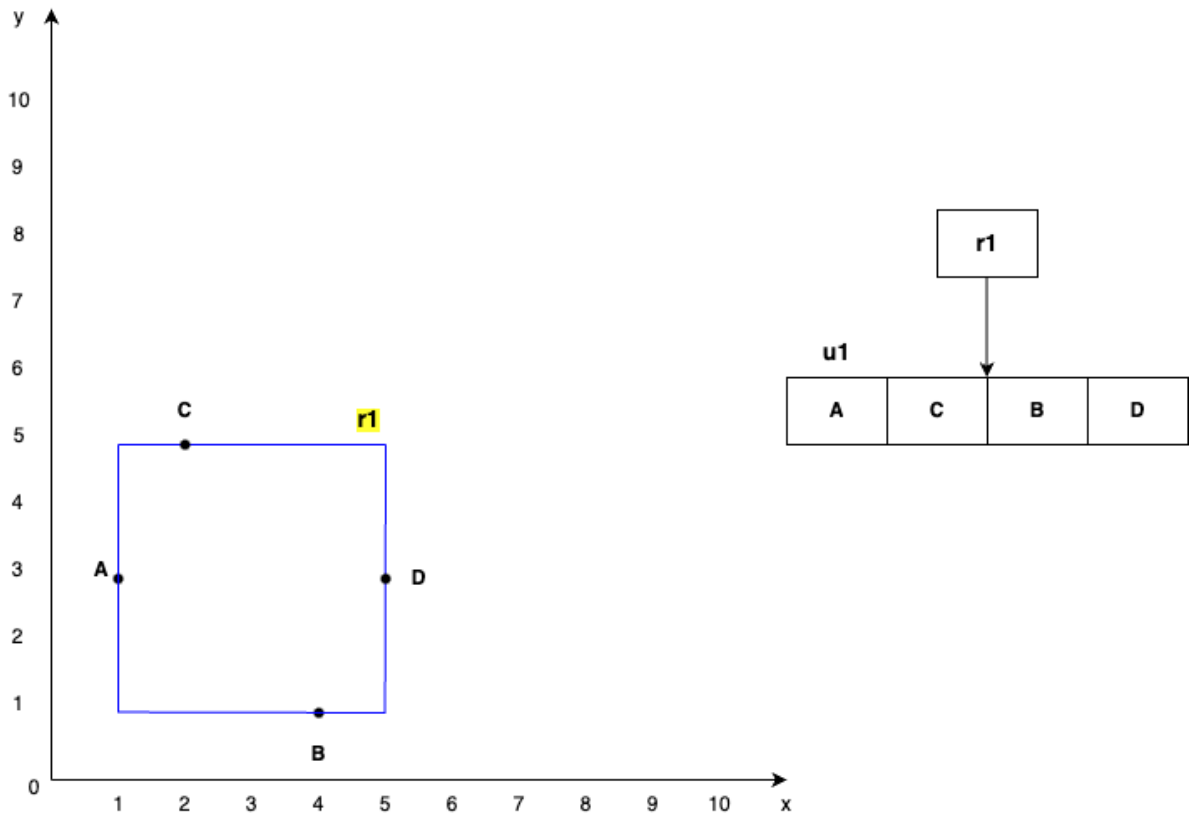
- **Insert A (1,3) and B (4,1)**



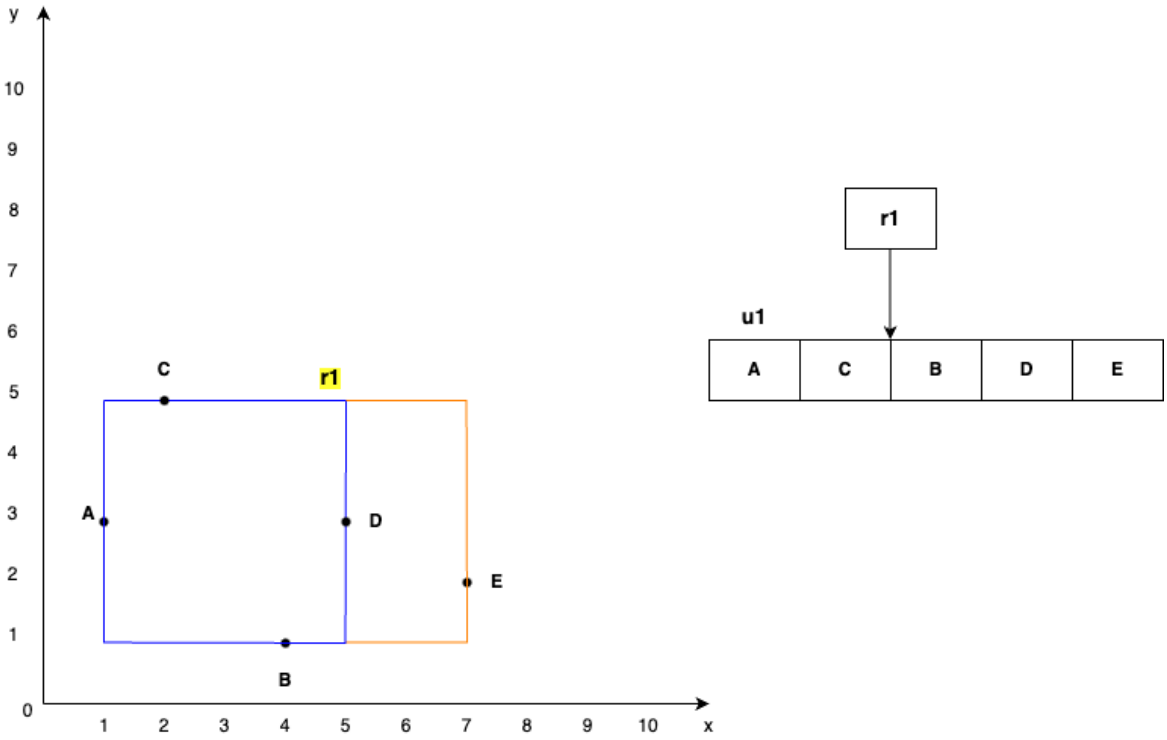
- **Insert C (2,5) and D (5,3)**

Before the R-tree establishment, the points are sorted based on their X and Y dimensions:

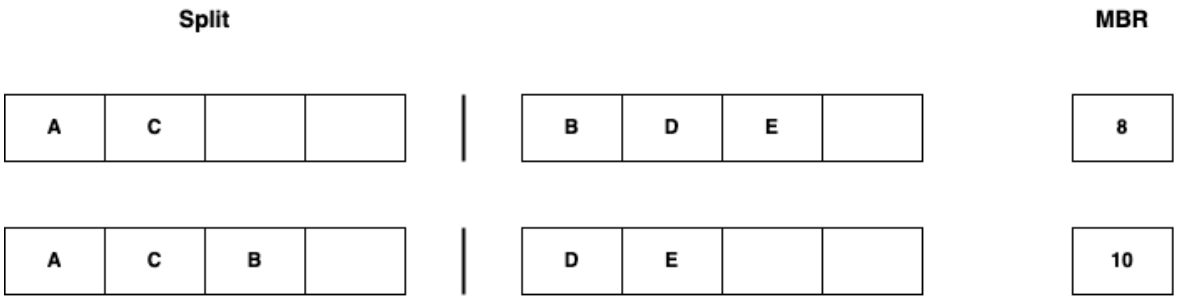
- Based on X-dimension: A(1,3) -> C(2,5) -> B(4,1) -> D(5,3)
- Based on Y-dimension: since yA = yD = 3, we do not consider to Y-dimension.



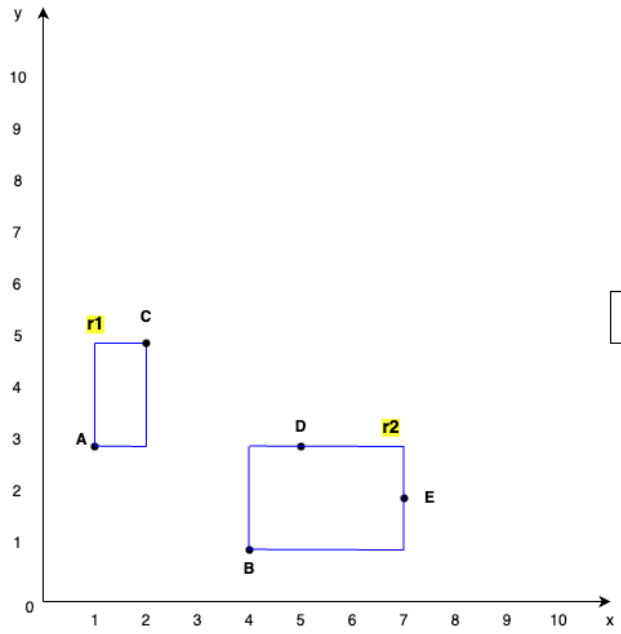
- **Insert E (7,2)**

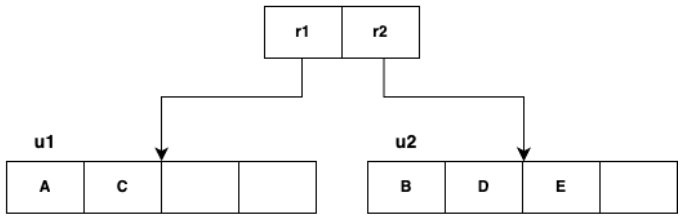


Due to the parameter B=4, the node experiences an overflow. By applying the rule 0.4B, which equates to 2, it's determined that each node must contain a minimum of 2 points.



Between the two splits, the first option (A, C) | (B,D, E) has a lower total perimeter of 8 compared to the second option's 10. Therefore, splitting the nodes into (A, C) and (B,D,E) after the insertion of E(7,2) would be the more efficient choice based on the perimeter criteria.



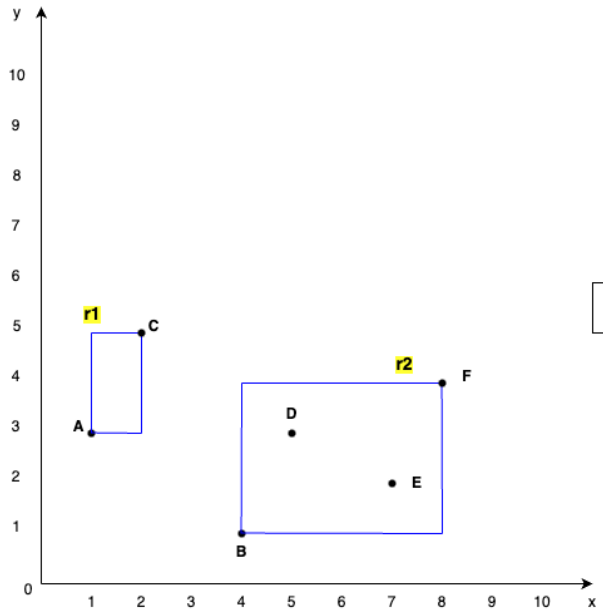


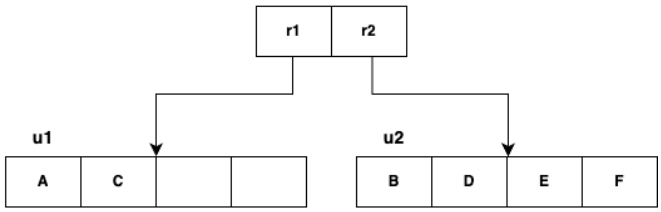
- **Insert F (8,4)**

Split				MBR
A	C			3
A	C	F		9

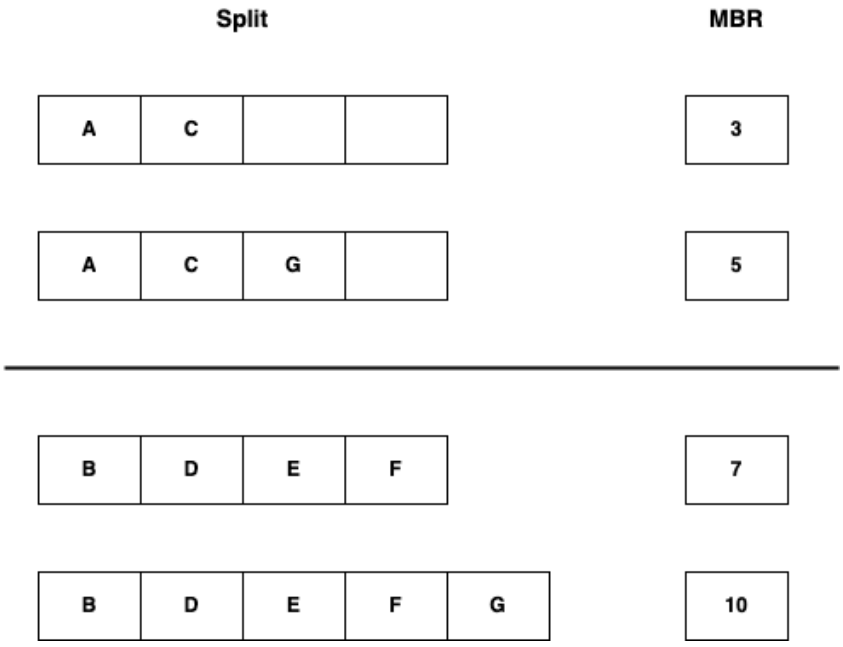
B	D	E		5
B	D	E	F	7

In this case, it's more efficient to add F to the group (B, D, E) since the MBR perimeter for (B, D, E, F) is smaller.

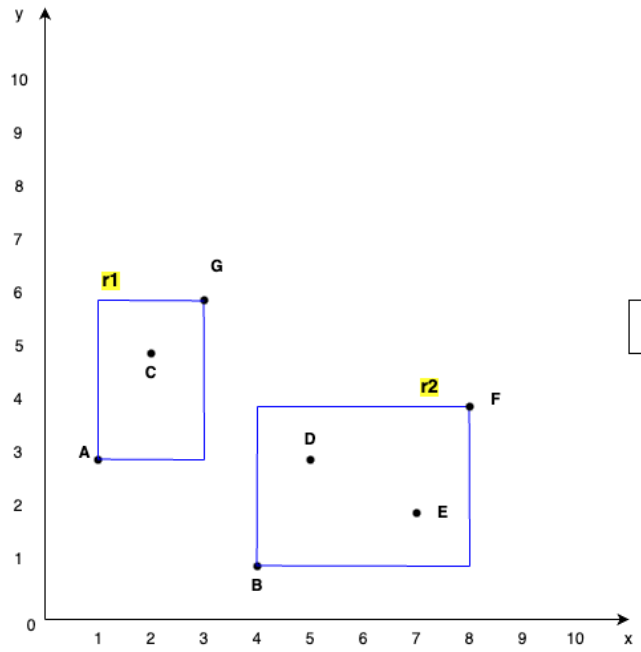


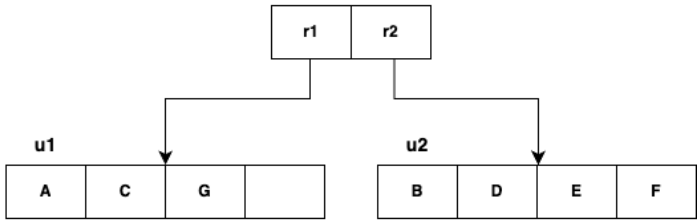


- **Insert G (3,6)**



Using the same method, we can add G into node u1 (A, C, G) since MBR for (A, C, G) is smaller than that of (B, D, E, F, G).

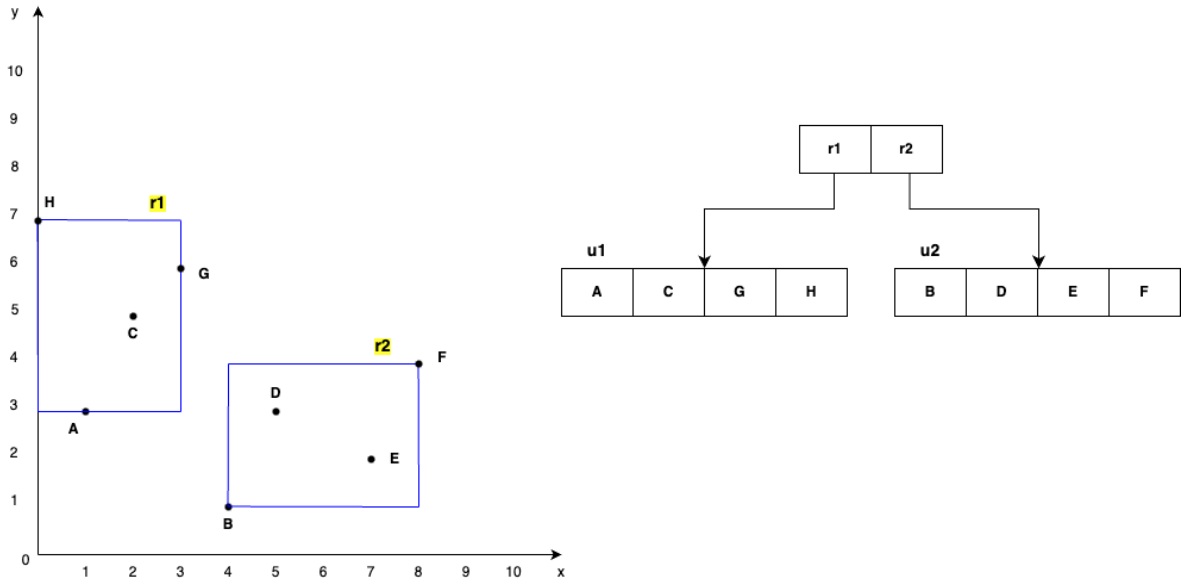




- **Insert H(0,7)**

Split	MBR
A C G	5
A C G H	7
B D E F	7
B D E F H	14

Using the same method, we choose to add H into (A,C,G,H). Because the MBR of (A, C, G, H) is 7 while MBR of (B, D, E, F, H) is 13.

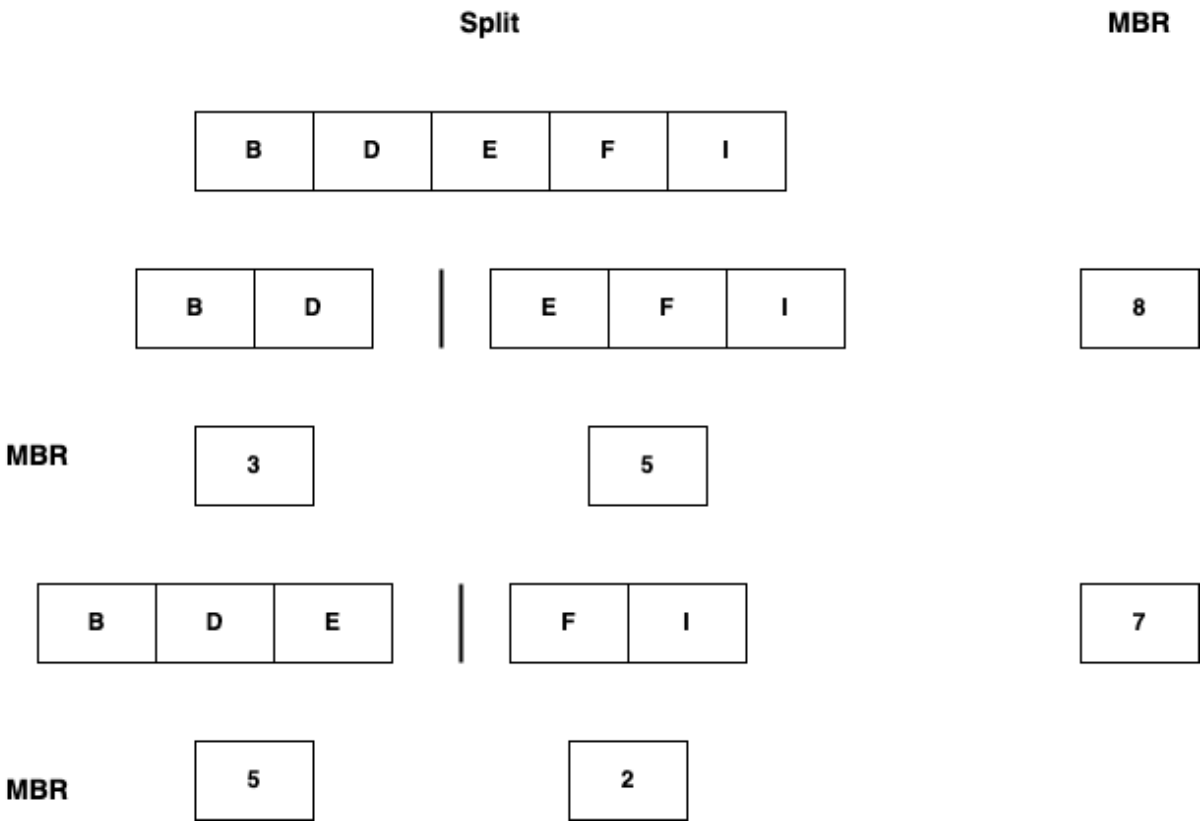


- **Insert I (10,4)**

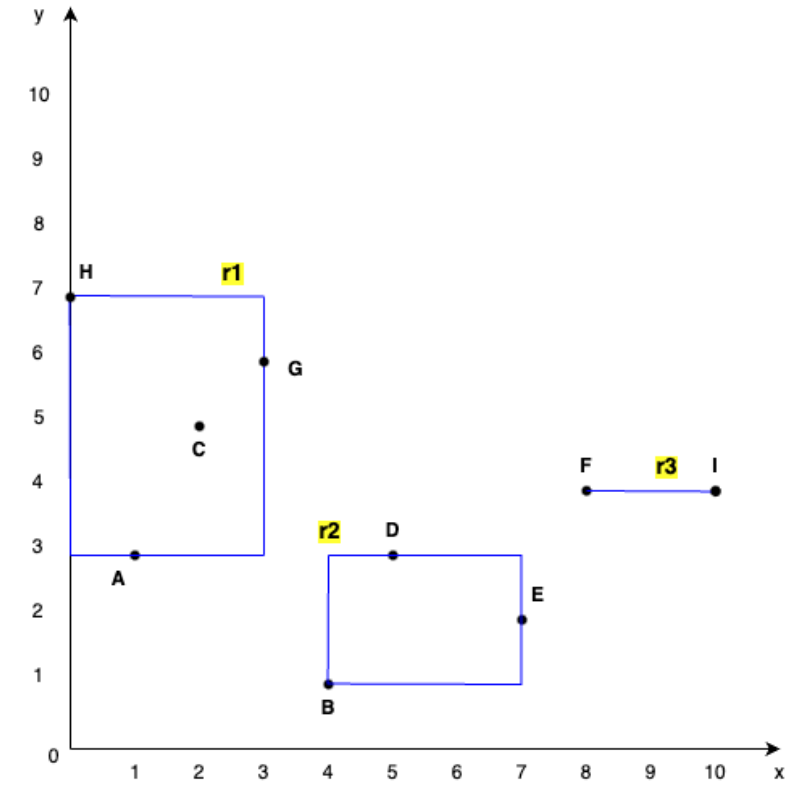
Split					MBR
A	C	G	H		7
A	C	G	H	I	14

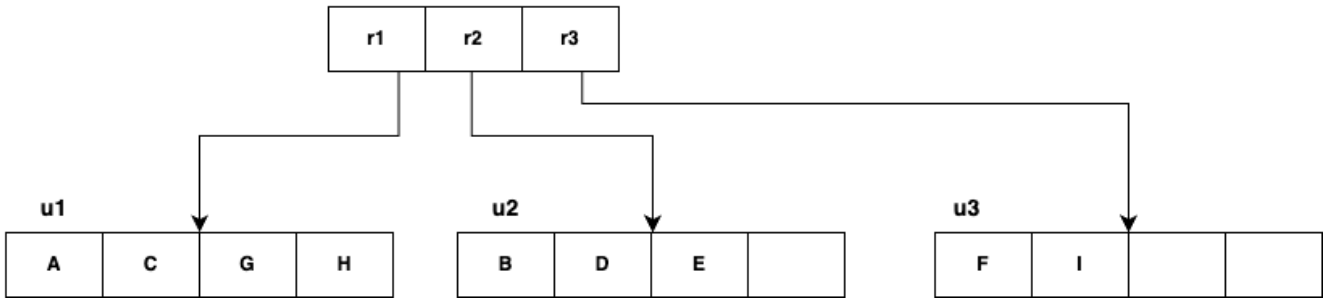
B	D	E	F		7
B	D	E	F	I	9

Using the same method, we choose the third split (B, D, E, F, I) which has a smaller MBR. However, (B, D, E, F, I) has 5 points while B =4, leading to an overflow. Therefore, we need to split the node (B, D, E, F, I) into 2 nodes which have at least 2 points in each.

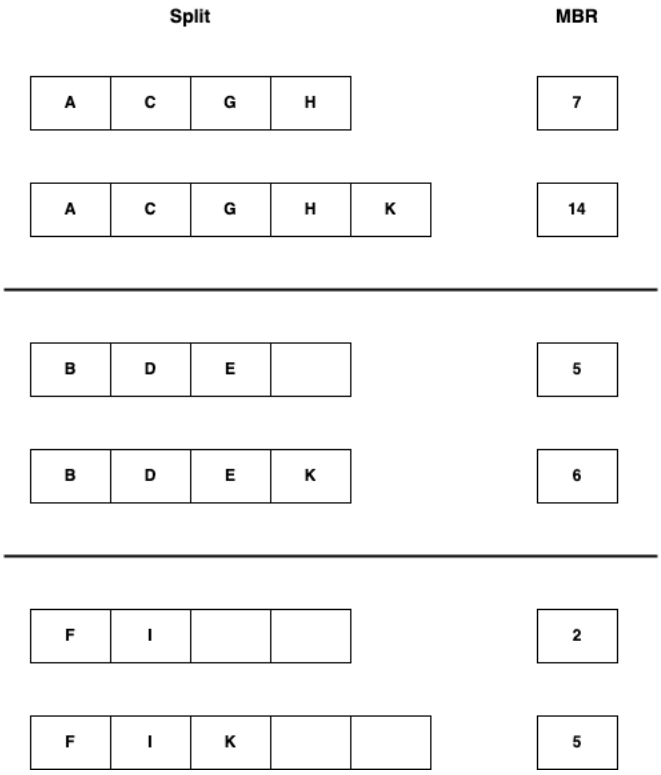


Using the same method, we choose the second split (B, D,E) and (F,I) with the smaller MBR.


y
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
x
H
r1
G
C
F
r3
I
r2
D
A
E
B

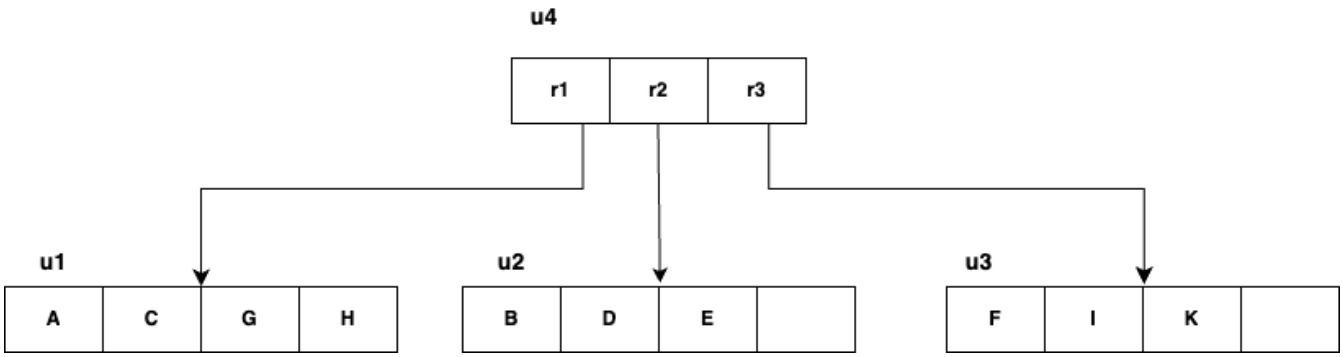

r1
r2
r3
u1
A
C
G
H
u2
B
D
E
u3
F
I

- **Insert K (8,1)**

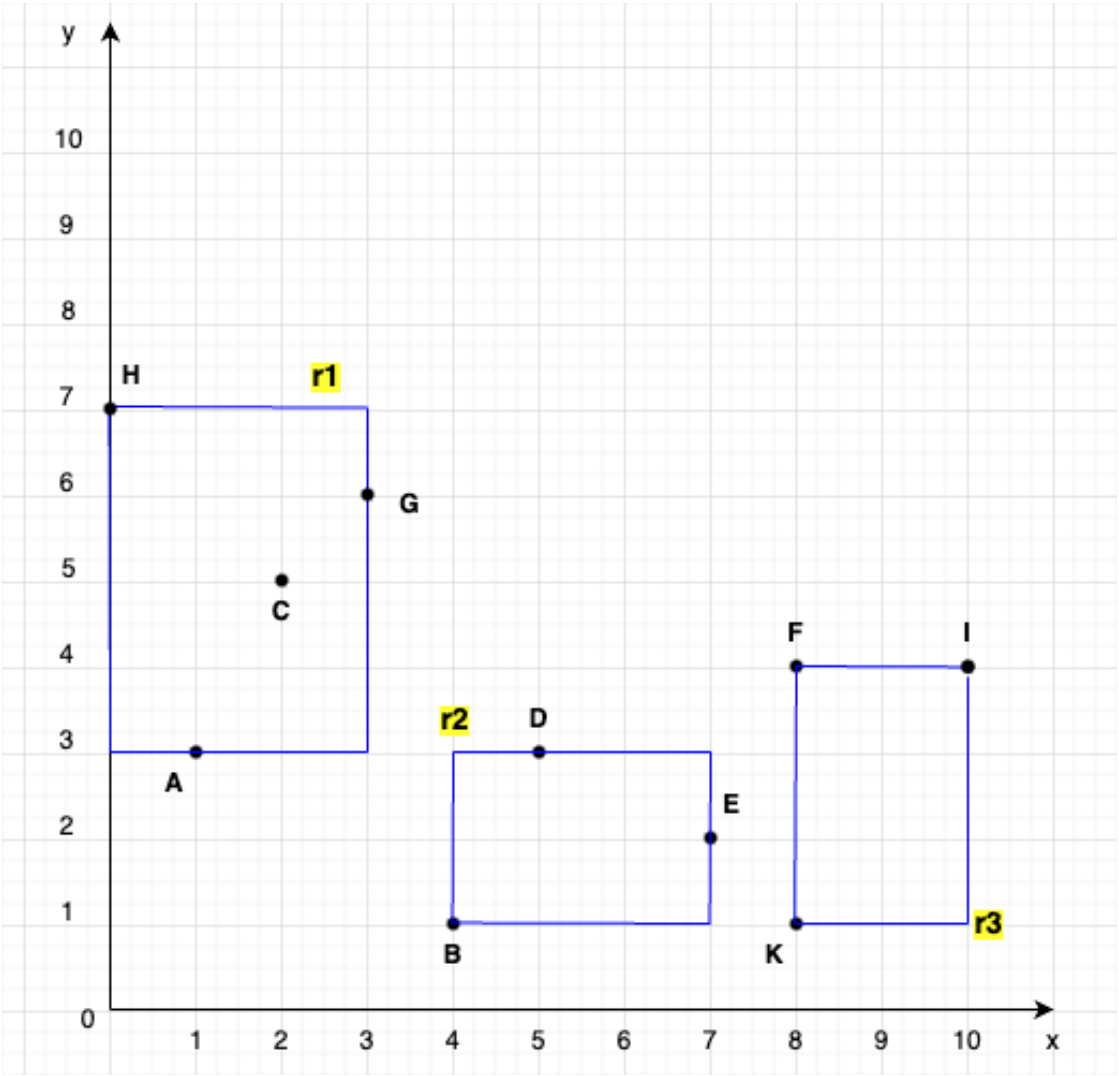


Using the same method, we get (F, I, K) has the smallest MBR.

This is the final R-tree



3.1.2 Bounding Boxes



3.2. Sequential-based search and R-Tree based search

3.2.1. Sequential-based search

For each point, check if its "x" coordinate lies between "x1" and its "y" coordinate lies between "y1" and "y2" of the query rectangle. If the condition is met, increment the count by 1.

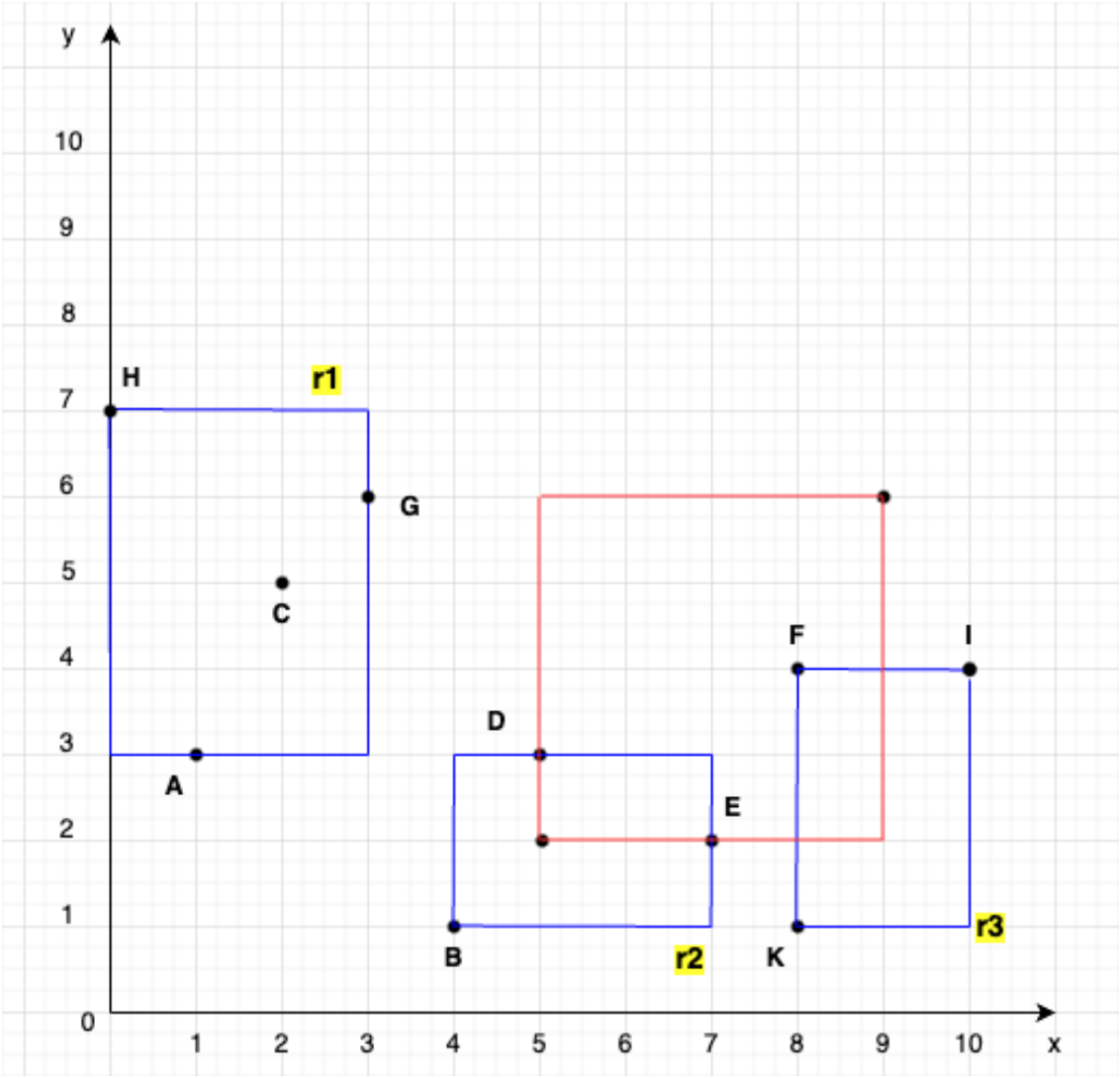
For the query range "x1 = 5", "y1 = 2", "x2 = 9", "y2 = 6", we would check each point:

Points	Check condition	Result
A (1, 3)	$1 < 5$	Not satisfy
B (4,1)	$4 < 5$	Not satisfy
C (2, 5)	$2 < 5$	Not satisfy
D (5, 3)	$5 \leq 5 \leq 9$ $2 \leq 5 \leq 6$	Satisfy
E (7, 2)	$5 \leq 7 \leq 9$ $2 \leq 2 \leq 6$	Satisfy
F (8, 4)	$5 \leq 8 \leq 9$ $2 \leq 4 \leq 6$	Satisfy
G (3, 6)	$3 < 5$	Not satisfy
H (0, 7)	$0 < 5$	Not satisfy
I (10, 4)	$10 > 9$	Not satisfy
K (8, 1)	$5 \leq 8 \leq 9$ $1 < 2$	Not satisfy

From the above checks, only the points D (5, 3), E (7, 2) and F (8, 4) lie within the query box. Therefore, the total count of points within the query box is 3.

3.2.2. R-Tree based search

Based on the created bounding boxes, we check the intersection of the query range in the bounding boxes.



The R-tree-based search provides an optimized approach to spatial querying, as illustrated by the **Query Range** (red box) delineated by:

x1 = 5, y1= 2, x2 = 9, y2 = 6

- **For r1:**
- Bounded by x1 = 0, y1 = 3, x2 = 3, y2 = 7
- The query does not overlap with r1 since the maximum x-coordinate of r1 is less than the minimum x-coordinate of the query range.

⇨ A, C, G, and H are excluded from the results.A,C,G,H do not lie within the query range

- **For r2:**
- Bounded by x1 = 4, y1 = 1, x2 = 7, y2 = 3

⇨ The query intersects r2 at its upper right boundary.

⇨ D (5,3) is a part of the query result.

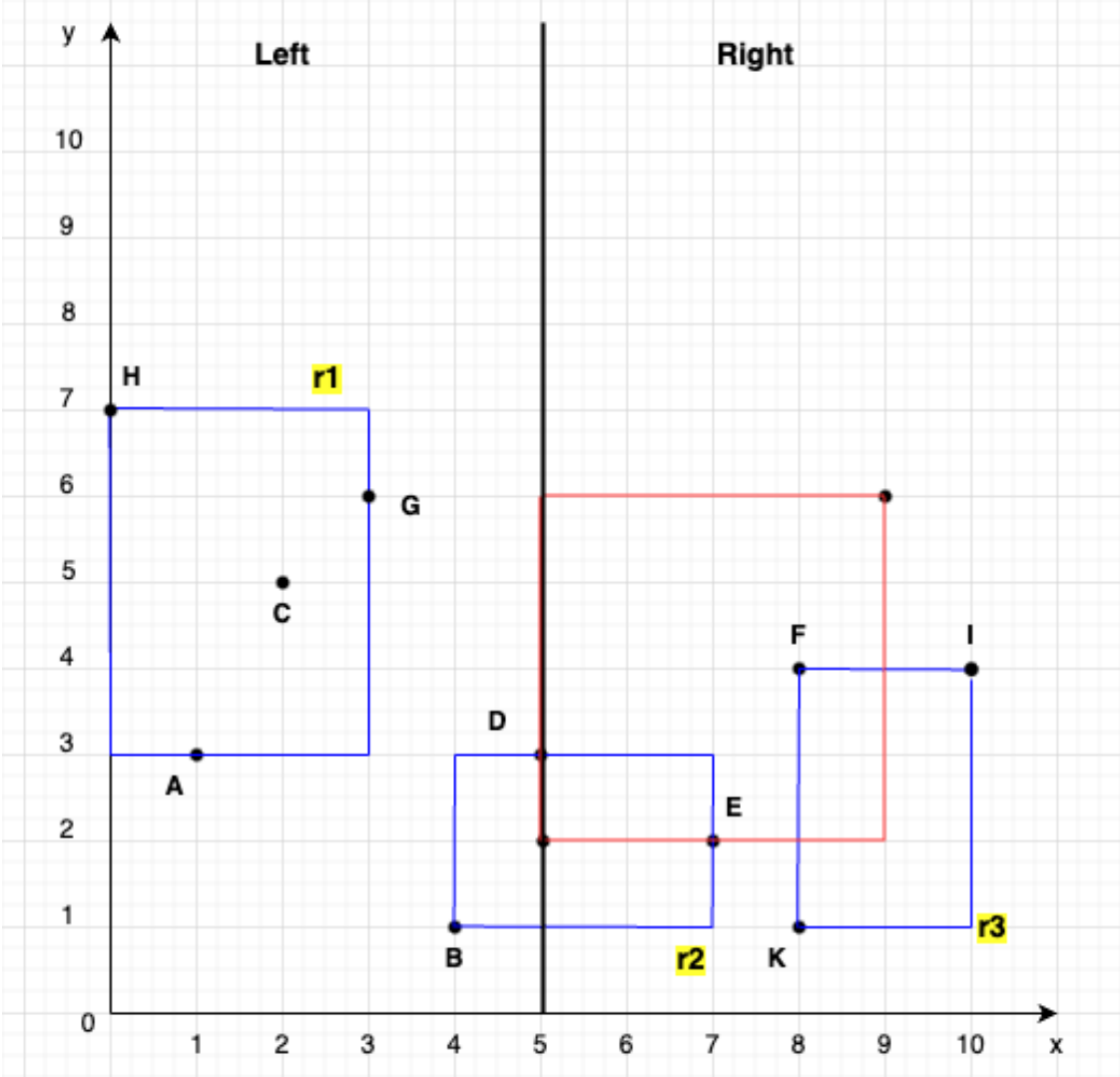
- **For r3:**
- Bounded by x1 = 8, y1 = 1, x2 = 10, y2 = 4
- The query encompasses a significant part of r3

⇨ E (7, 2) and F (8, 4) are encapsulated within the query range.

Therefore, D (5,3), E (7, 2) and F (8, 4) lie within the query range.

Thus, when using the R-tree-based search, the algorithm efficiently identifies the bounding boxes (MBRs) that intersect with the query range. By examining only the points within these intersecting MBRs, we've quickly identified that points D, E, and F are the ones that lie within the given query range. This process showcases the efficiency and speed advantage of using an R-tree based search for spatial queries compared to a full Sequential-based search across all points.

3.3. Divide-and-Conquer in R-Tree-based search



Given a dataset with 10 data points and a maximum x-value of 10, we utilize the Divide and Conquer approach in the R-tree-based search. The dataset is bifurcated at the midpoint x=5.

- **Left Subset:** Points with $x < 5$, namely A, B, C, G, H.
- **Right Subset:** Points with $x \geq 5$, specifically D, E, F, I, K.

Considering the provided **Query Range** (red box) defined by:

- x1 = 5, y1 = 2
- x2 = 9, y2 = 6

- ⇨ The query range exclusively falls within the "Right" subset, given x1 = 5 (inclusive) and x2=9.
- ⇨ Consequently, points in the "Left" subset are not within the query range.
- ⇨ Evaluating the "Right" subset, data points D (5, 3), E (7, 2), and F (8, 4) are encapsulated by the query range.