DSA PROJECT

R-TREE

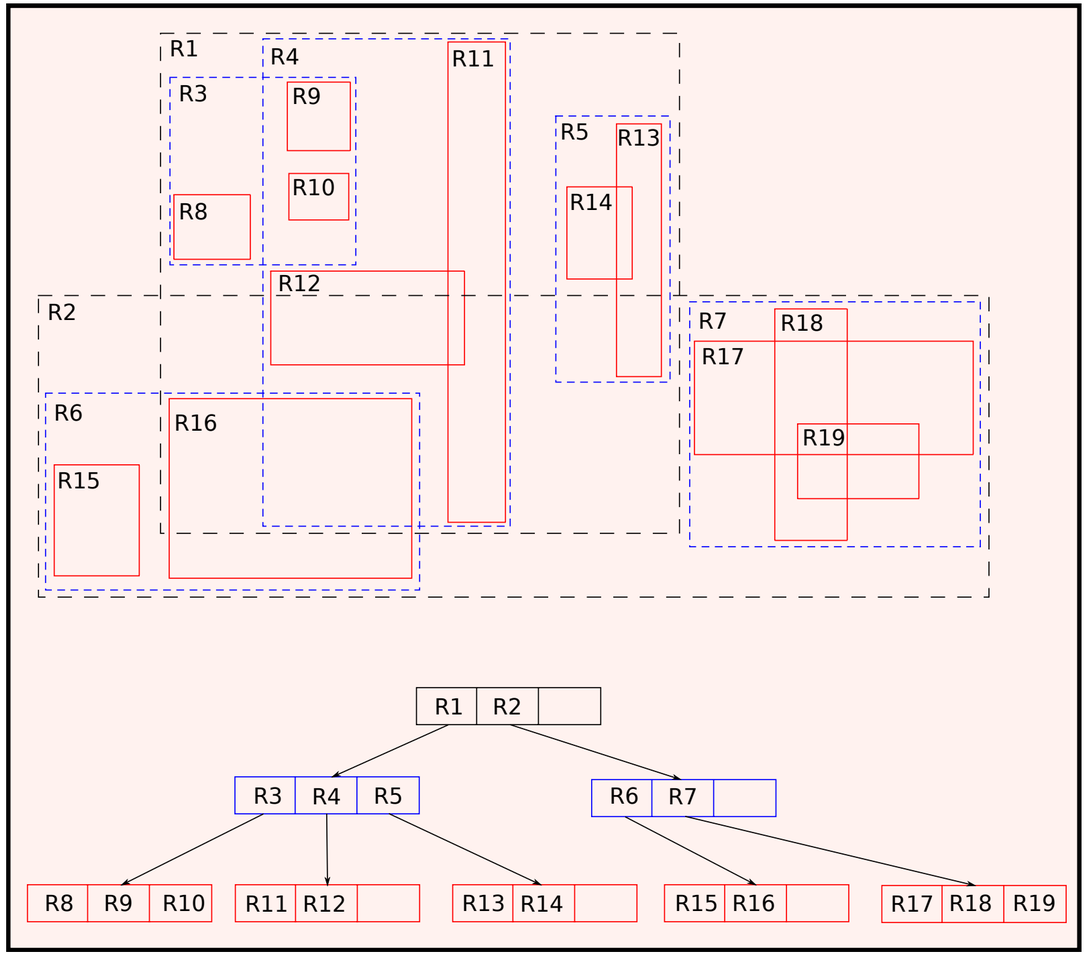
ADVANCED DATA STRUCTURE

- IMPLEMENTATION
- VISUALIZATION
- APPLICATION

What is R-TREE ?

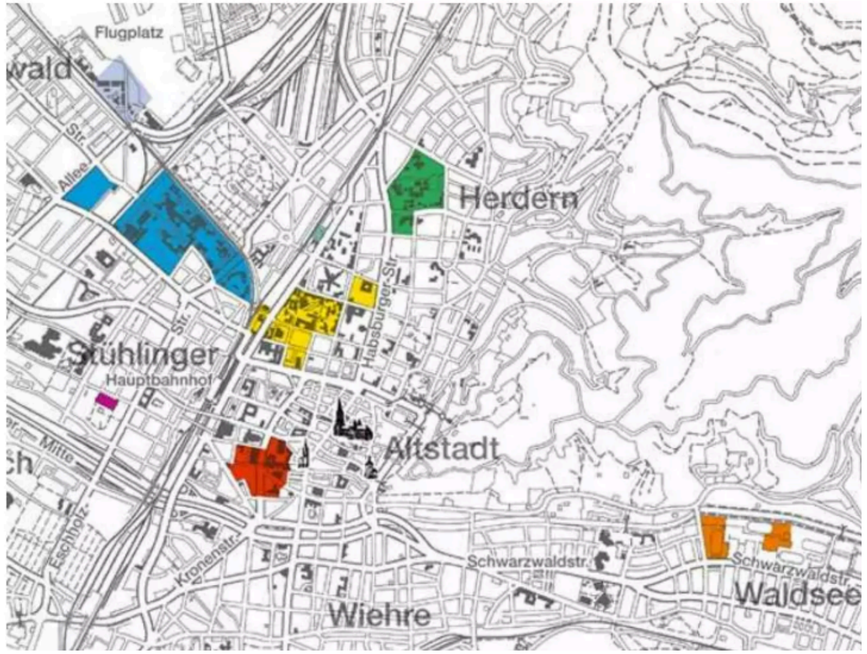
Definition : A spatial index structure used for indexing multi-dimensional data, such as the locations of objects in a 2D space.

- – R-tree organizes objects in groups.
- – Each group is represented by a rectangle.
- – Rectangles cover nearby objects.
- – Queries check intersections with rectangles first.
- – Leaf level has single-object rectangles.
- – Higher levels group more objects in rectangles.

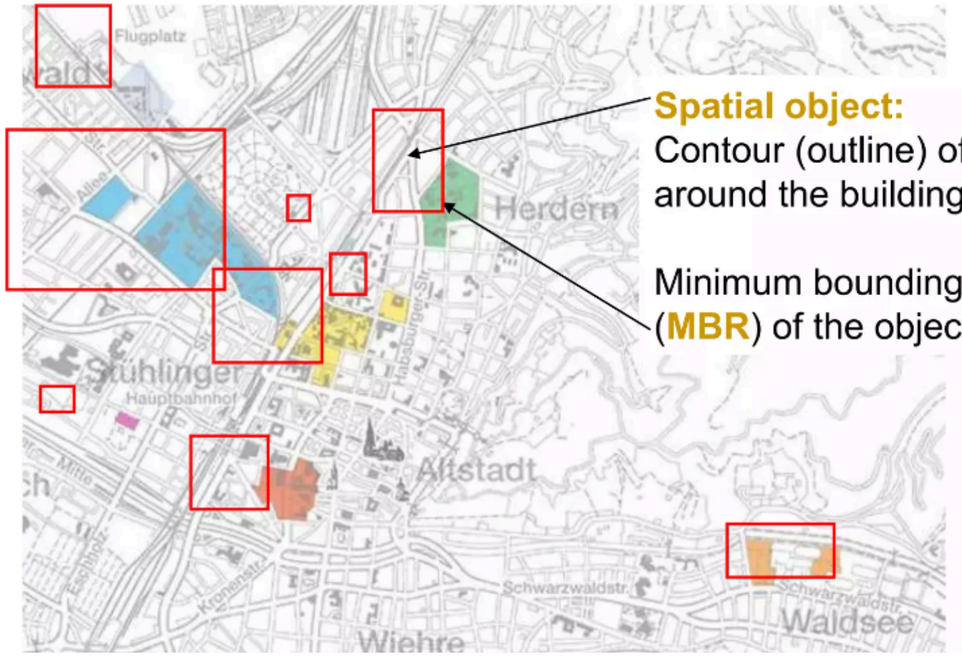


SPATIAL DATA & INDEX

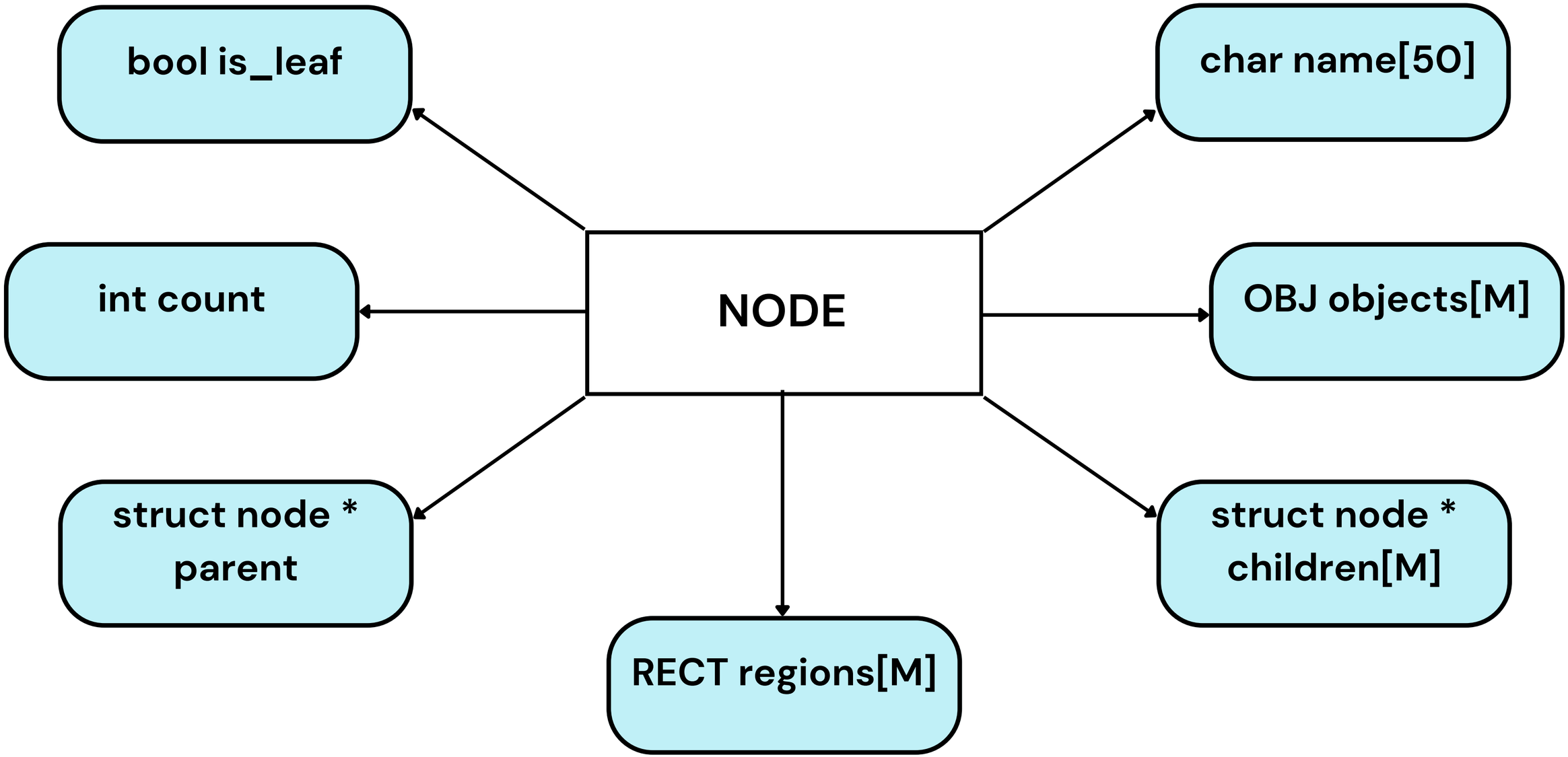
Given a city map, ‘index’ all university buildings in an efficient structure for quick topological search.



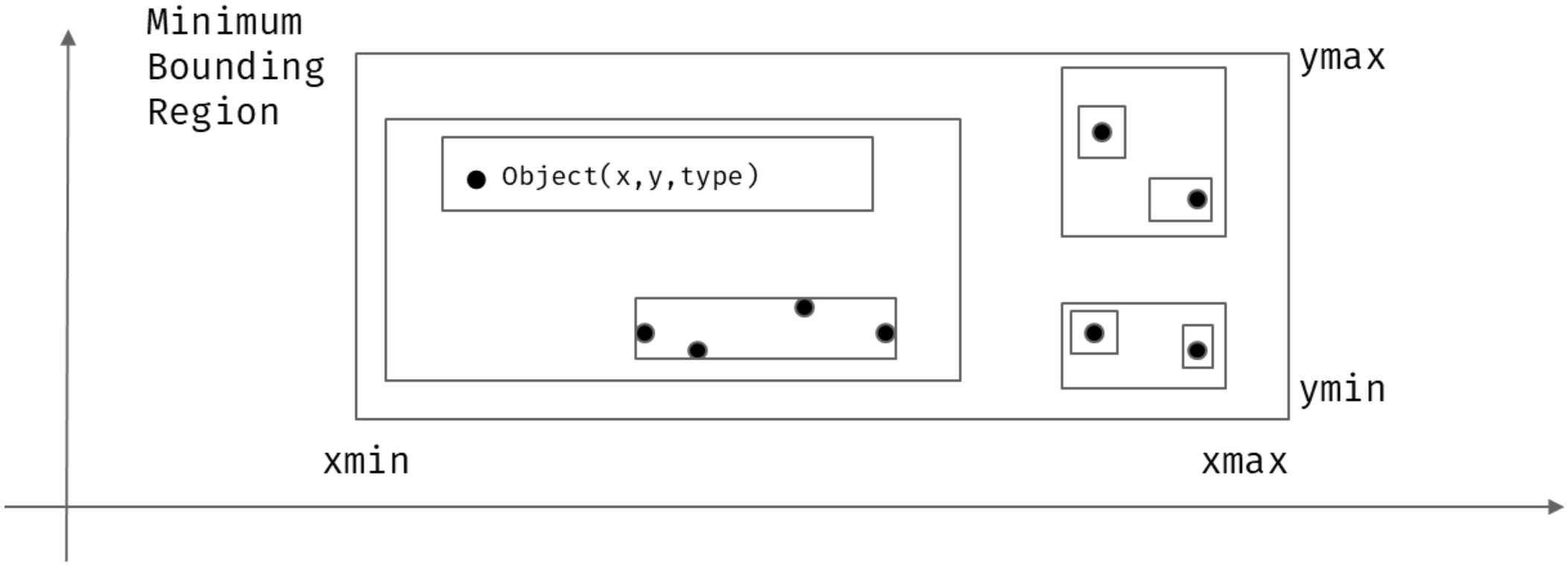
“Index” buildings in an efficient structure for quick search



NODE STRUCTURE



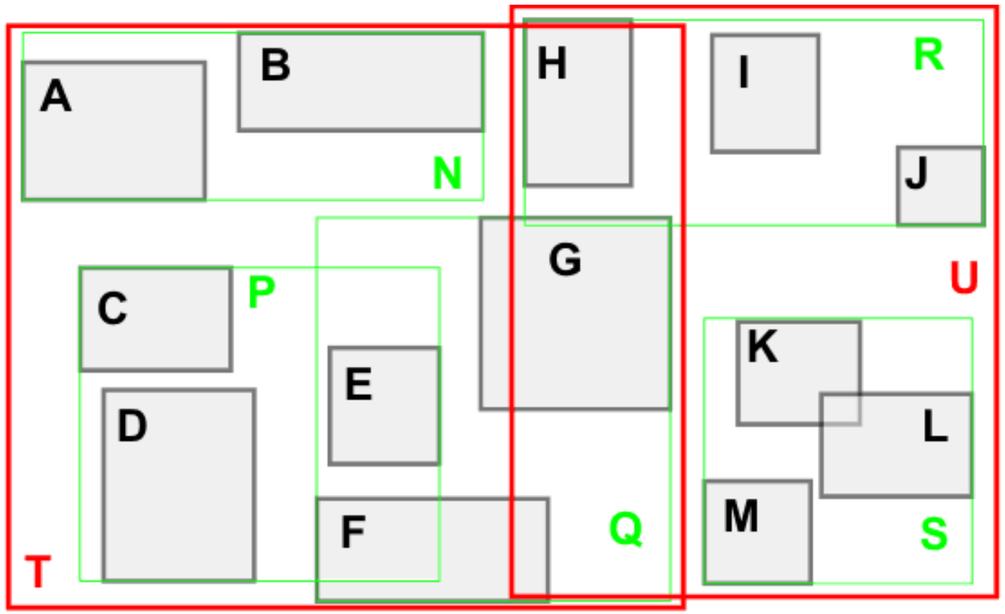
STRUCTURE OF MBR & OBJECT

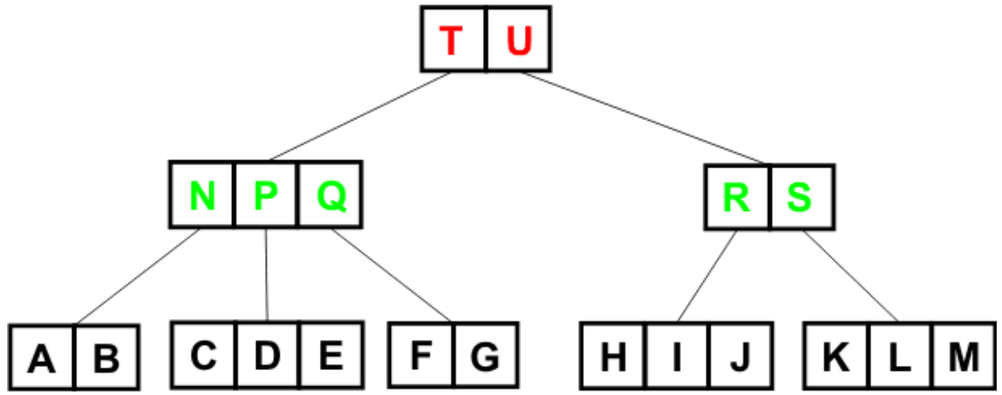


R-TREE : Illustrative Example

m = 2

M = 3





METHODS IMPLEMENTED

1 INSERTION

- Leaf Node
- Internal Node
- Quadratic Splitting

2 SEARCH

- Nearest Neighbor
- K Nearest Neighbor
- In-Radius Objects

3 TRAVERSAL

- Pre-Order Traversal

4 VISUALIZATION & APPLICATION

- SDL2 Library
- Menu Based Live Rendering all Operations

METHODS IMPLEMENTED

```
// Function declaration
NODE create_new_leaf_node();
NODE create_new_internal_node();
R_TREE create_new_r_tree();
RECT create_new_rect(int min_x, int min_y, int max_x, int max_y);
OBJ create_new_object(int x, int y, const char* type_name);
void insert_object_into_node(NODE node, OBJ object, RECT rect);
void insert_region_into_node(NODE parent_node, NODE child_node, RECT region);
long long area_rect(RECT rect);
long long increase_in_area(RECT rect1, RECT rect2);
NODE choose_leaf(NODE node, OBJ object);
RECT bounding_box(NODE node);
int * pick_seeds(NODE node, RECT rect);
bool search_in_node(NODE node, RECT rect);
int pick_next(NODE node1, NODE node2, NODE node, RECT rect);
NODE * quadratic_split_leaf_node(NODE node, OBJ object);
void adjust_tree(R_TREE r_tree, NODE node1, NODE node2, NODE node);
NODE * quadratic_split_internal_node(NODE node, RECT rect, NODE child);
void insert_in_r_tree(R_TREE r_tree, OBJ object);
void pre_order_traversal(NODE node, int depth);
double euclidean_distance(int x1, int y1, int x2, int y2);
bool rect_intersects(RECT rect1, RECT rect2);
void search_in_r_tree(NODE node, RECT rect, int user_x, int user_y, double radius, OBJ found_objects[], int* num_found);
OBJ search_nearest_neighbor(NODE node, RECT rect, int user_x, int user_y, OBJ nearest_neighbor, double* min_distance);
double euclidean_distance(int x1, int y1, int x2, int y2);
int assign_internal_node_names(struct node *node, int region_counter);
void find_k_nearest_neighbors(NODE root, int user_x, int user_y, int K, OBJ* neighbors);
```

Let's construct R-TREE!

With Following Object Points :

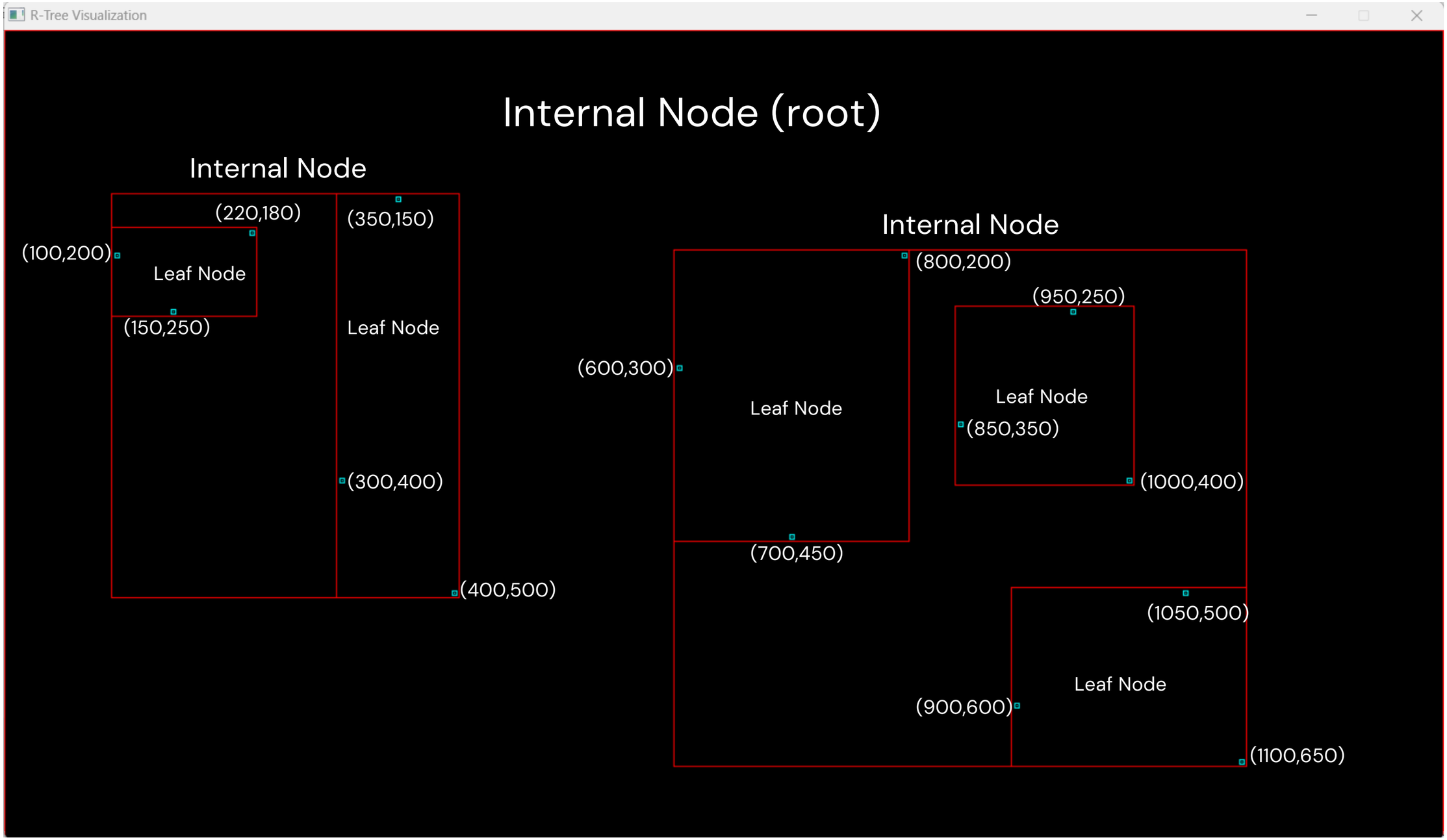
```
100 200 Hotel
150 250 Hospital
300 400 Park
220 180 School
350 150 Library
400 500 Restaurant
600 300 Gym
700 450 Bank
800 200 Mall
850 350 Cafe
900 600 Stadium
950 250 Museum
1000 400 Cinema
1050 500 Airport
1100 650 Beach
```

OBJECT FORMAT

(x , y , Location Type)

This Points can be imported from .txt file or We can Manually enter each.

OUTPUT (GUI)

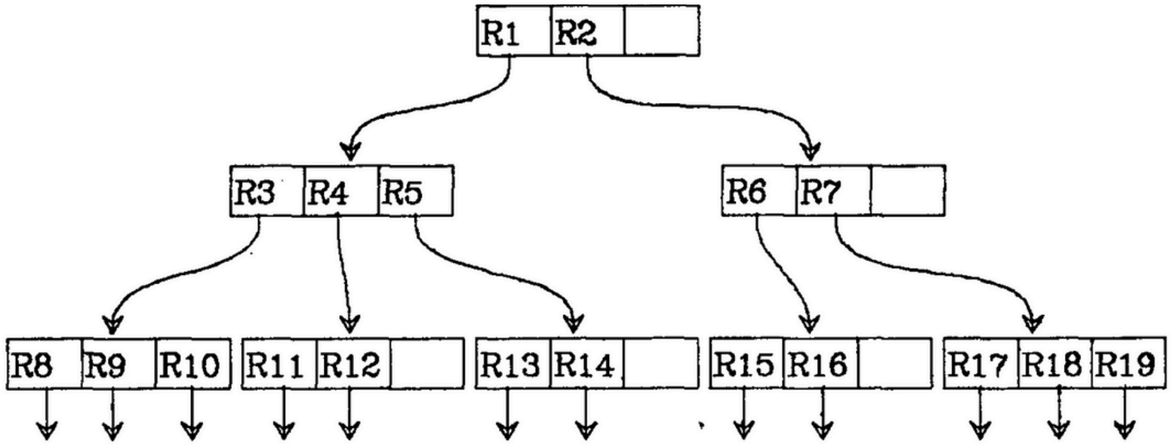

R-Tree Visualization
Internal Node (root)
Internal Node
(220,180)
(350,150)
(100,200)
Leaf Node
(150,250)
Leaf Node
(300,400)
(400,500)
Internal Node
(800,200)
(950,250)
(600,300)
Leaf Node
Leaf Node
(850,350)
(1000,400)
(700,450)
(1050,500)
Leaf Node
(900,600)
(1100,650)

Insertion

1. **Find position for new record:**
   - Use *ChooseLeaf* to select a leaf node L for the new entry E.
2. **Add record to leaf node:**
   - If L has room, insert E.
   - If L is full, invoke *SplitNode* to split L into L and LL, placing E and old entries in appropriate nodes.
3. **Propagate changes upward:**
   - Invoke *AdjustTree* on L, passing LL if split occurred, to adjust tree structure and update covering rectangles.
4. **Grow tree taller:**
   - If root split occurred, create new root with resulting nodes.

Time Compexity

O(m * log(n), where n is the number of nodes in the tree and m is the number of objects within a node.



Let us consider the R tree shown above.
Let us insert a new entry E in the tree

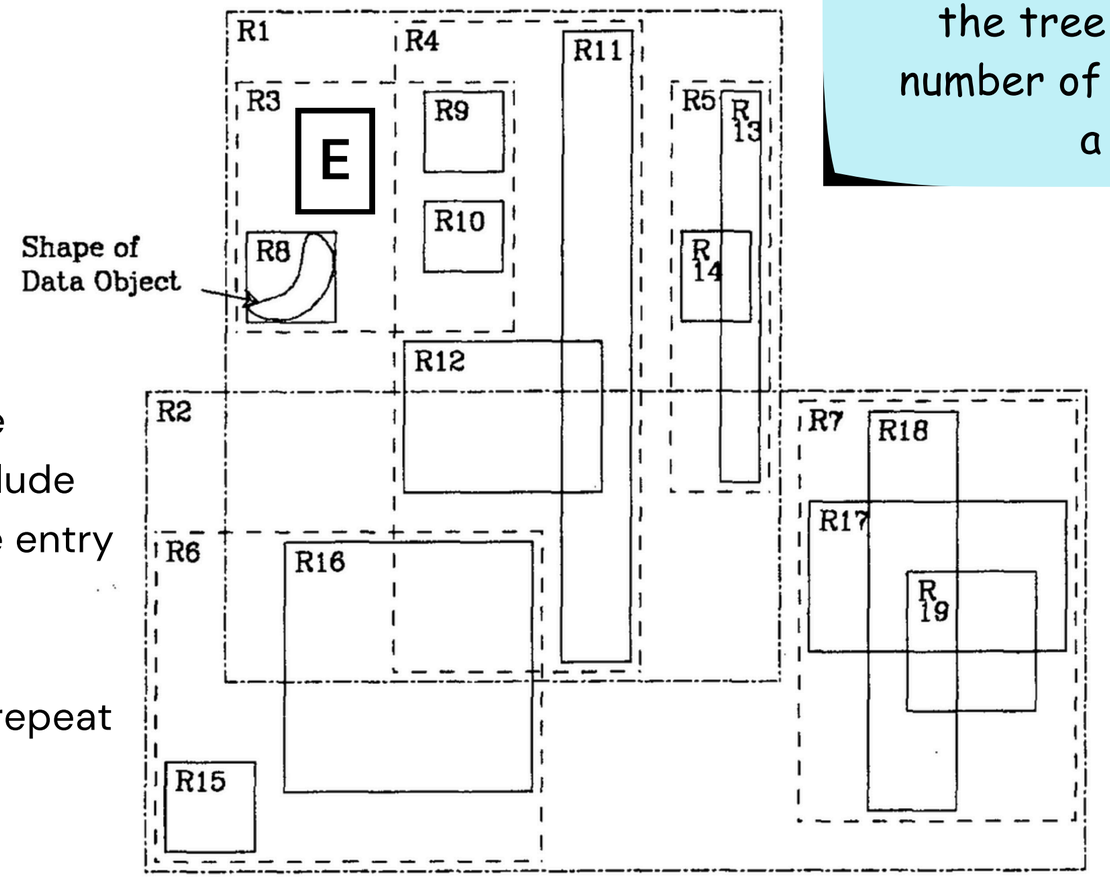
E

ChooseLeaf

1. Start at root
   - Set N as the root node.
2. Check if leaf
   - If N is a leaf, return N
3. Choose subtree
   - If N is not a leaf, select the entry F in N whose rectangle needs the least enlargement to include the new entry E. Resolve ties by choosing the entry with the smallest area
4. Descend until a leaf is reached.
   - Set N as the child node pointed to by F, and repeat the process from step 2

Time Complexity

O(m * log(n), where n is the number of nodes in the tree and m is the number of objects within a node.



This gives us a node overflow condition

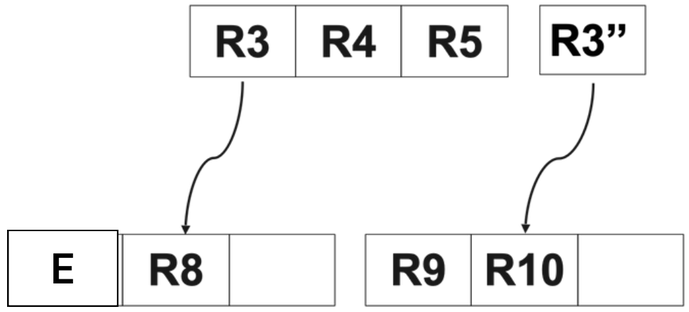
Using the ChooseLeaf function, let us assume the entry E needs to be inserted to the leaf node containing R8, R9 & R10

Quadratic Node Split

Time Complexity

O(m * m), where m is the number of objects within a node.

1. It attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible.
2. The cost is quadratic in M and linear in the number of dimensions.
3. Process
   - Pick first entry for each group
   - Check if done
   - Select entry to assign to each group



PickSeeds

Time Complexity

O(m * log(n), where n is the number of nodes in the tree and m is the number of objects within a node.

1. Select two entries to be the first elements of the groups
2. Process
   - Calculate inefficiency of grouping entries together
   - Choose the most wasteful pai

PickNext

Time Complexity

O(m * log(n), where n is the number of nodes in the tree and m is the number of objects within a node.

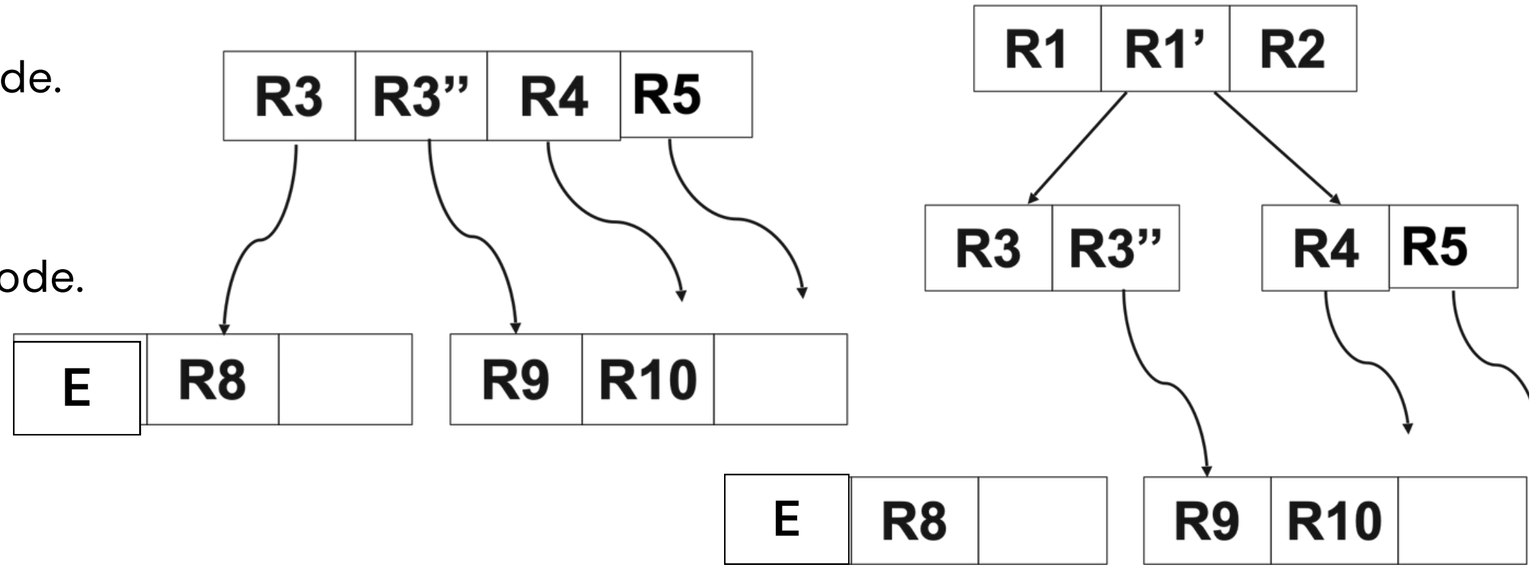
1. Select one remaining entry for classification in a group
2. Process
   - Determine cost of putting each entry in each group
   - Find entry with greatest preference for one group

AdjustTree

1. Initialize:
   - Set current node N as the leaf node.
   - Set NN as the resulting second node if N was split previously.
2. Check if done:
   - Stop if N is the root.
3. Adjust covering rectangle in parent entry:
   - Update covering rectangle of N's entry in parent node.
4. Propagate node split upward:
   - If N has a partner NN:
     - Create new entry for NN in parent node.
     - If no room, split parent node.
5. Move up to next level:
   - Repeat process until reaching the root node.

Time Compexity

O(m * log(n)), where n is the number of nodes in the tree and m is the number of objects within a node.



Time Compexity

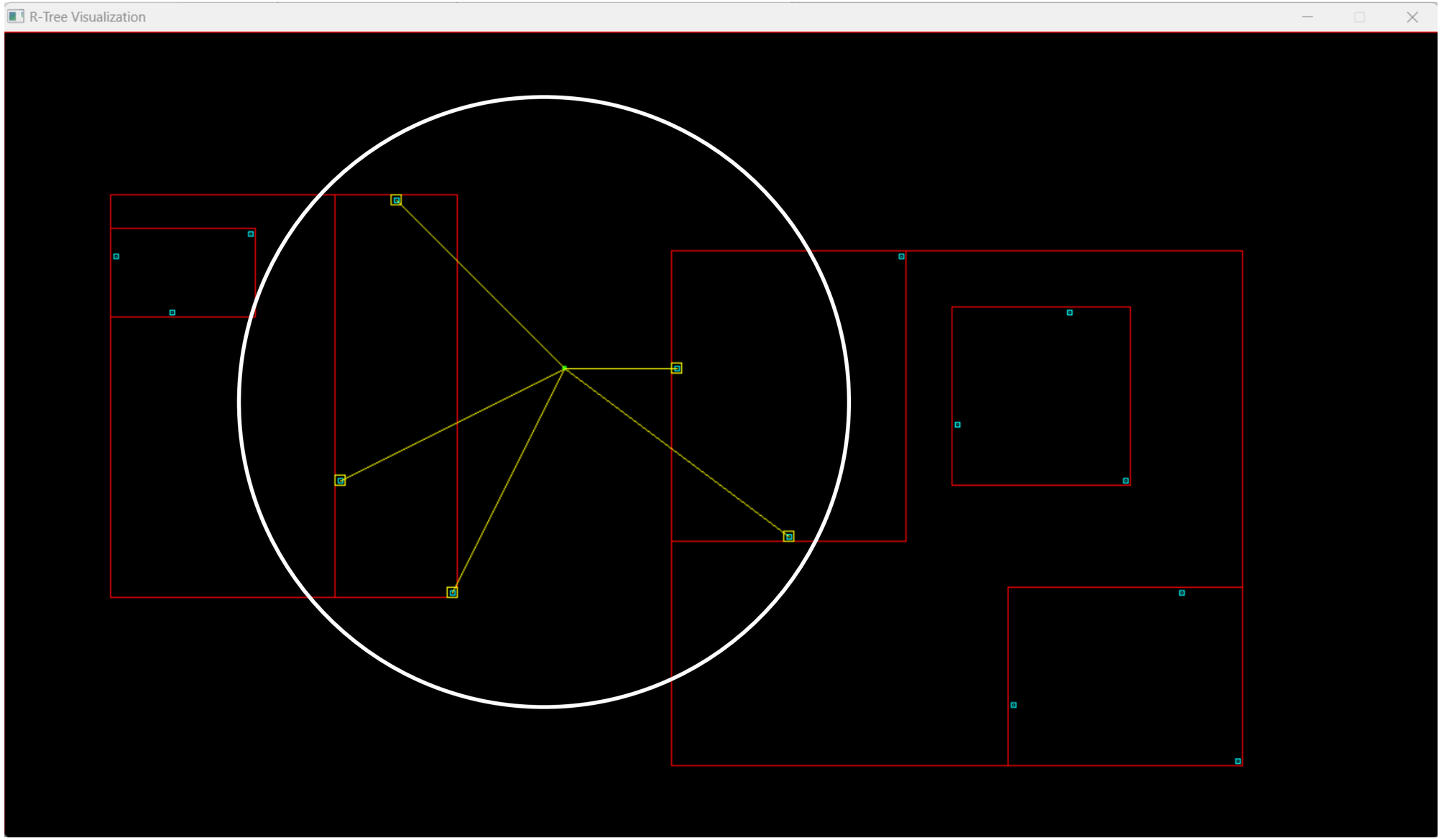
O(log n * m), where n is the number of nodes in the tree and m is the number of objects within a node.

SEARCH OPERATION

In-Radius Objects

- The function recursively traverses the R-tree.
- The search rectangle is created by defining its dimensions based on the user's coordinates (x, y) and the specified radius.
- It checks if the bounding rectangles of internal nodes intersect with the search rectangle.
- For leaf nodes, it checks if object points are within the search rectangle.
- Objects found within the search rectangle are added to the array.
- The count of found objects is updated.
- This objects help the GUI to mark points.

OUTPUT (GUI)



Time Compexity

O(M * log(n)), where M is the maximum number of children per node, and n is the total number of objects in the R-Tree

SEARCH OPERATION

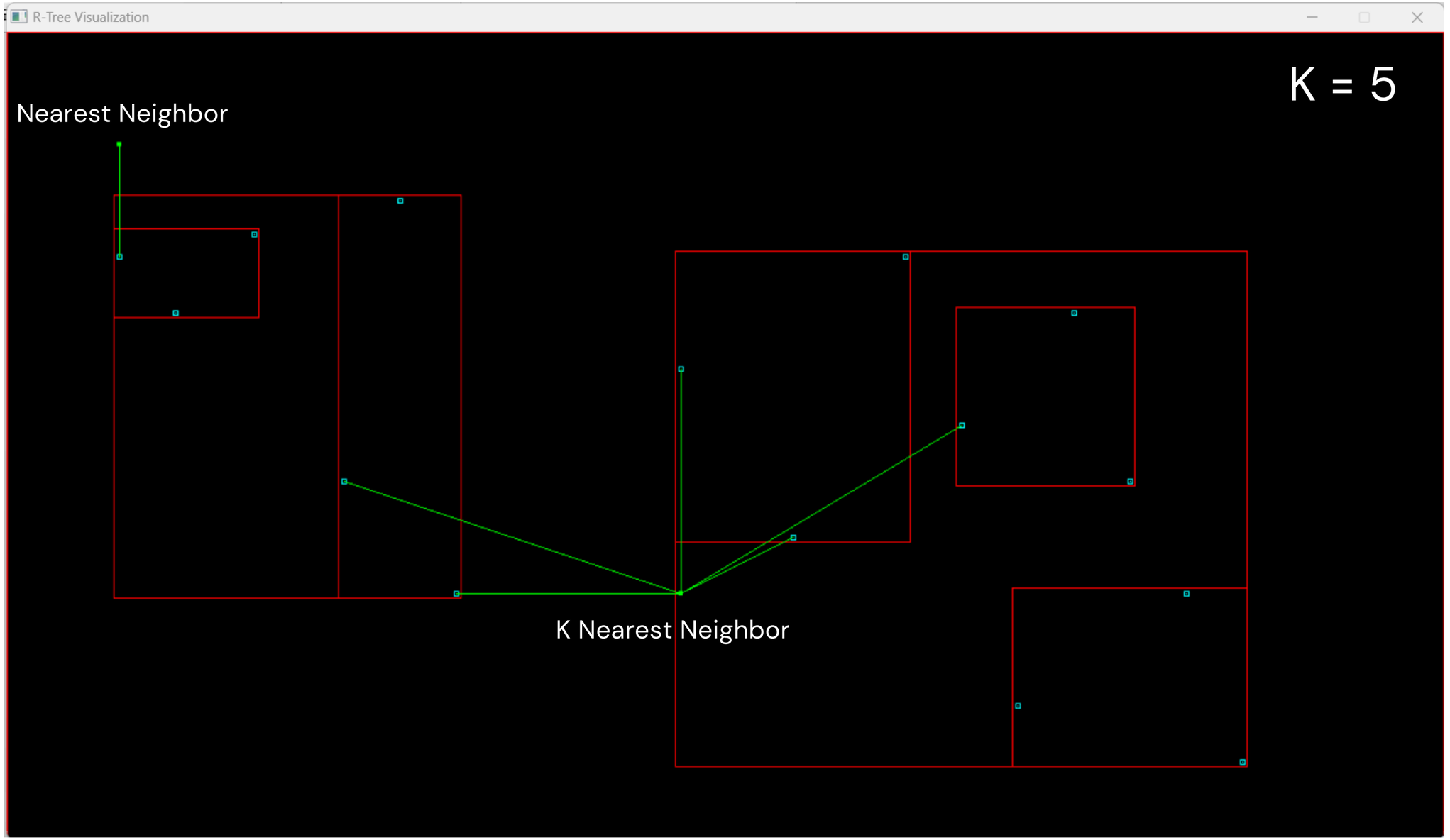
Nearest Neighbor & KNN

Time Compexity

O(K * M * log(n)), where K is the number of nearest neighbors to find, and n is the total number of objects in the R-Tree.

- The function recursively traverses the R-tree.
- User's coordinates (x, y) are specified.
- It checks the bounding rectangles for internal node(s) and get the closest internal nodes using euclidean distance.
- For leaf nodes, it checks if object points euclidean distance is closest.
- Objects found are added to the array for K Nearest Neighbor.
- The count of found objects is updated.
- This objects help the GUI to mark points.

OUTPUT (GUI)


R-Tree Visualization
K = 5
Nearest Neighbor
K Nearest Neighbor

BENEFITS

- Efficient Queries: One of the main benefits of using R-Trees is the efficiency of spatial queries. The R-Tree is designed to maximize data locality, meaning it minimizes the amount of disk I/O operations and makes queries faster.
- Dynamic: R-Trees are dynamic, allowing insertions and deletions as data changes. This allows the R-Tree to adjust to the evolving needs of geospatial data handling.
- Balanced: Like other tree data structures, R-Trees are balanced, meaning the path from the root to any leaf is the same length. This balanced nature guarantees logarithmic time search performance.

REAL-TIME APPLICATIONS

- Google Maps: Utilizes R-trees for indexing and searching spatial data such as maps, locations, and points of interest, providing efficient navigation and location-based services.
- Oracle Spatial: Incorporates R-trees for spatial indexing in Oracle databases, enabling efficient spatial queries and analysis for GIS applications and location-based services.
- Airbnb: Utilizes R-trees for indexing and querying spatial data such as property locations and user searches, facilitating efficient matching of accommodations with user preferences.
- Uber: Utilizes R-trees for spatial indexing in its routing algorithms, optimizing the matching of drivers with passengers and efficient navigation in urban environments.

CONTRIBUTIONS

1. INSERTION - YASH PARDESHI
2. SEARCH - YASH PAWAR
3. TRAVERSAL - YASH PARDESHI
4. VISUALIZATION & APPLICATION - YASH PAWAR

CONTRIBUTORS

1. YASH PARDESHI • 612203130
2. YASH R. PAWAR • 612203143

Thank You!