Implementation of R-Trees using Corner Based Splitting

Project 4 - Group 14

Chaitanya Sethi - 2020B3A71961P
Mayank Verma - 2020B3A70841P
Satvik Jain - 2020B3A70791P
Sidartha Sankara Pati - 2020B2A70687P
Stavya Puri - 2020B5A70912P

Table of Contents

Introduction to R-trees

Definition

Rectangle tree, or R-trees is a spatial data structure for dynamic indexing multi-dimensional data like points, rectangles or polygons.

Key Idea

Recursive partitioning of data space into smaller *bounding rectangles*(represent group of data points) optimized to have related data points stored together.

Need for R-Trees

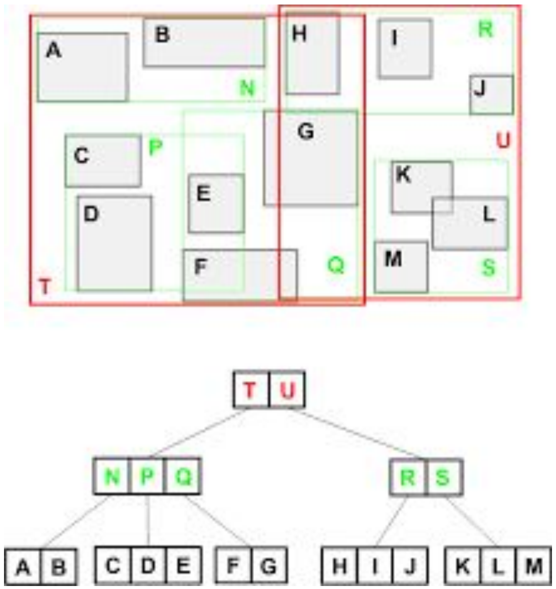
Introduced in 1984 by *Antonin Guttman*, R-trees emerged as the first efficient solution for spatial inquiry, having huge applications in areas like CAD, Multimedia Applications.

Structure of R-Trees

1. Height-balanced tree like B-tree
2. Designed to minimize number of visits to nodes for operations.
3. Spatial Data is comprised by ***Minimum Bounding Rectangle(MBR)*** upto the root.

Properties of R-trees

1. Every **leaf node** contains between **m(min)** and **M(Max)** index records unless it is the root. Thus, the root *can have less entries than m*.
2. For each **index record** in a leaf node, I is the **smallest rectangle** that spatially contains the **n-dimensional data object** represented by the indicated tuple.
3. Every **non-leaf node** has children ***between m and M***, except the root
4. For each entry in a **non-leaf node**, **i is the smallest rectangle** that spatially contains the rectangles in the **child node.**
5. The **root node** has **at least two children** unless it is a leaf
6. All leaves appear on the same level - *Balanced Tree Property*



Structures of Nodes and Parameters

Leaf Node Structure:

Index Record Structure : (I, *tuple - identifier*)

Tuple-identifier: Refers to tuple in database

I: n-dimensional rectangle(MBR) of index spatial object: $I = (I_0, I_1, \ldots, I_{n-1})$

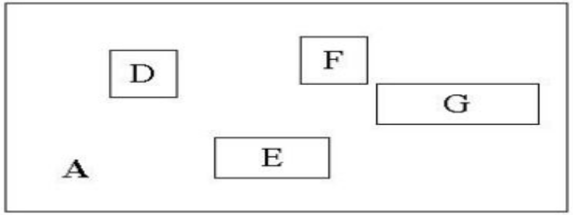
Ii: Closed bounded interval describing extend of object in dimension i.

Non-Leaf Node Structure:

Index Record Structure: (I, *child - pointer*)

child-pointer: Address of a lower node in R-tree

I: Covers all rectangles in child node's entries



Parameters:

Overall number of index records = N; Maximum number of nodes = $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + 1$

In one node: Minimum No. of entries = m; Maximum number of Entries = M;

Then, **($m \leqslant M/2$)**

m - of High importance as determines (I) Speed of Algorithm (II) Height of R-Tree

Overflow - If greater than M index records are inserted in node

Underflow - If less than m index records exist in node

Reorganisation Required!

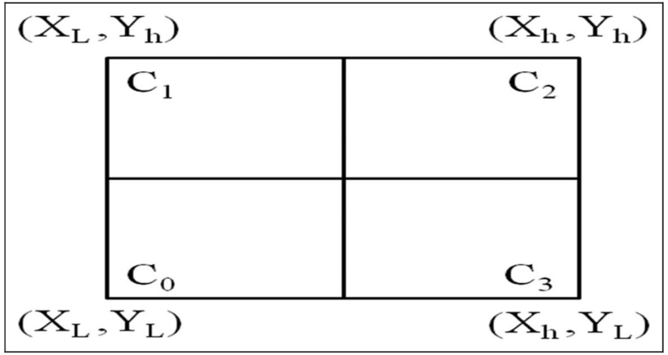
Splitting Algorithms

1. Quad Splitting algorithm : **$O(n^2)$**
2. Al-Badarneh et al. splitting algorithm (NR Algorithm): **O(n logn)**

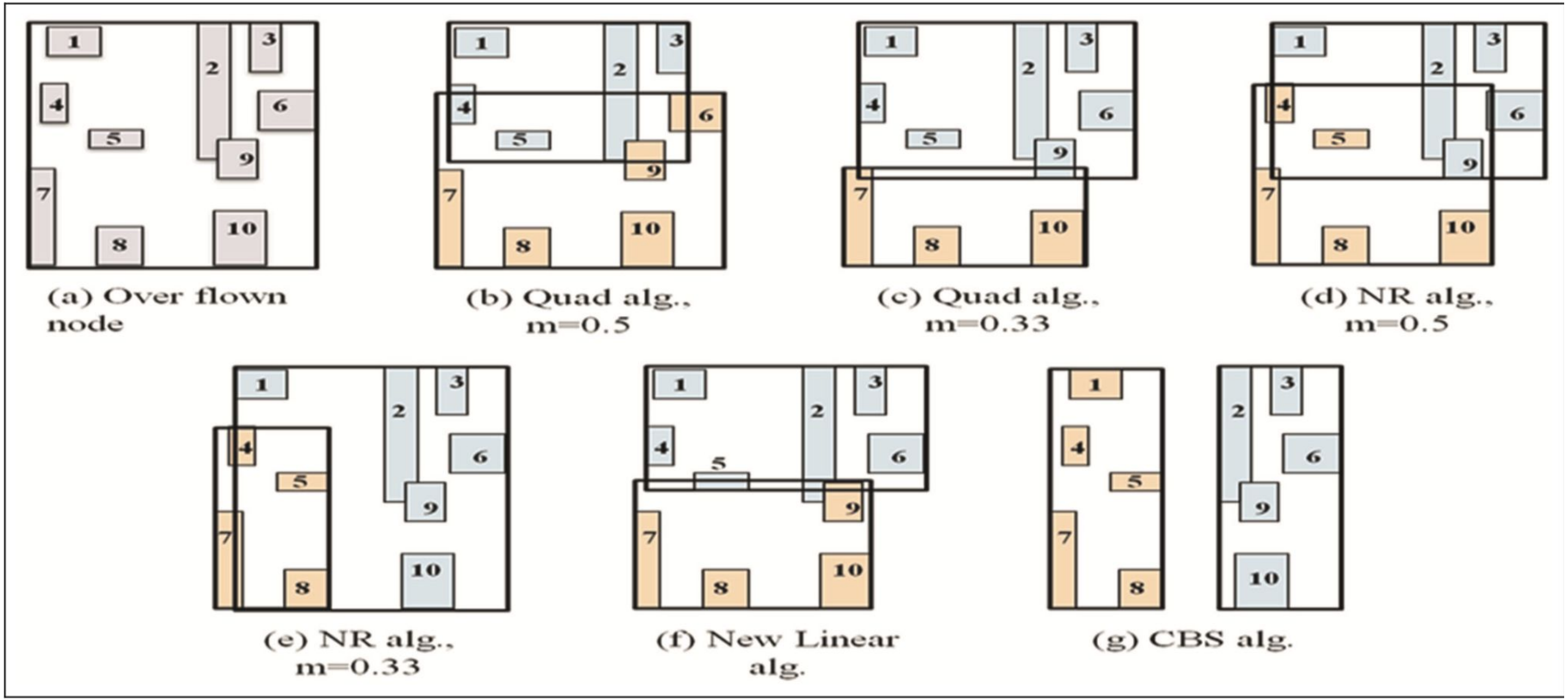
Disadvantages that motivate the study of more efficient algorithms:

- The possible performance deterioration owing to the investigation of several paths from the root to the leaf level while executing a point location query, especially when the overlap of the MBRs is significant.
- A few large rectangles may increase the degree of overlap significantly, leading to performance degradation during range query execution owing to empty space

3. **Corner Based Splitting Algorithm**
   The implementation of CBS which we have used in our project is more efficient as it resolves the two aforementioned problems.



Graphical Comparison of Algorithms



Structures Defined in the Code

Point Structure:

- This Structure is mainly used to define a **point in n-dimensional space** (n=2 in our case).

Rectangle Structure:

- This Structure is mainly used to define a **Rectangle** i.e. a 2-Dimensional geometrical object (mathematically) used for MBRs.

- It is defined using 2 ‘Point’ Structures giving its lowest point i.e. **bottom-most and left-most point** and highest point i.e. **top-most and right-most points**.

RTree Structure:

- This Structure is mainly used to define an **R-Tree** which can be done by using only the **root node** as this root node then points to its children and the process continues later using the functions defined.

```c
typedef struct point {
    int x;//x coordinate
    int y;//y coordinate
} Point;

typedef struct rectangle{
    Point lowest; //Bottom left corner
    Point highest; //Top right corner
} Rectangle;


typedef struct rtree
{
    Node *root;// The root of rtree
} RTree;
```

Structures Defined in the Code

Node Structure:

- This Structure is mainly used to define a **Node** of an R-tree.
- This node can be either a **Leaf node** or a **Non-Leaf Node.**
- **children** node (M- Child Nodes) stores pointers to the **children of the present node** and **points**(M- Points) stores the **Index records i.e our coordinates.** This is used by leaf nodes respectively used later to allocate memory at time of creation of node by the **createNode()** under the if condition based on **isLeaf Flag** mentioned here itself.
- **num_children** gives the **number of children** of the given node that lie between m=2 and M=4 in our case and **Rectangle** named **'mbr'** is also defined for this Node which is the **MBR** of the given node
- The parent pointer points to the parent node of the given node

```
typedef struct Node
{
    int isleaf;
    int num_children;
    struct Node *parent;
    Rectangle mbr;
    struct Node* children[M];
    Point points[M];
} Node;
```

createNode()

Function Declaration: **Node *createNode(Point highest, Point lowest, int isleaf)**

1. This function is to **create a new Node** using the parameters entered by **dynamically allocating memory**.

2. The **Point** parameters are corresponding to the the node’s **highest point** and node’s **lowest point** for defining a MBR(rectangle ‘r’). This ensures an **underflow prevention** at the time of a node creation as here m = 2 i.e. minimum 2 points.

3. The **flag isleaf** corresponding to the nature of the node i.e. if the node is leaf or not.

4. If the new node is a leaf(isleaf = 1) then **we allocate the memory for M=4 points** but if it a non-leaf(isleaf = 0) node **we allocate the memory to its M= 4 children** for adding the data i.e. points or nodes later.

5. Returns **Node pointer** (From Structure).

```
e* createNode(Point highest,Point lowest,int isleaf){
//This is a function to create a node
Node* newNode = (Node *)malloc(sizeof(Node));
newNode->parent = NULL;
newNode->num_children = 0;
newNode->isleaf = isleaf;
Rectangle r;
r.highest = highest;
r.lowest = lowest;
newNode->mbr = r;
newNode->points[4];
for(int i = 0;i<4;i++){
    newNode->children[i] = (Node*)malloc(sizeof(Node));
}
return newNode;
```

createRtree()

Function Declaration: **RTree *createRtree(Point highest, Point lowest)**

1. This function is to **create a R-Tree** using the parameters entered by **dynamically allocating memory** for the R-Tree.

2. The **Point** parameters are corresponding to the the node’s highest point and node’s lowest point for defining a MBR(rectangle).

3. These are then used to **invoke the createNode()** function for creating a leaf node(isleaf = 1) as a root node is leaf node until any new nodes are added to it.

4. Returns **RTree pointer** (From Structure).

```
}

RTree* createRtree(Point highest, Point lowest){
    //This is a function to create rtree
    RTree* rtree = (RTree *)malloc(sizeof(RTree));
    Node * temp = createNode(highest,lowest,1);
    rtree->root = temp;
    return rtree;
}
```

calculateArea()

Function Declaration: **int calculateArea(Rectangle r)**

1. This function is to **calculate the area of a rectangle** using its implicit parameters of highest and lowest point coordinates.

2. The **Point** parameters within the rectangle are accessed and difference of highest and lowest coord is multiplied to area initialised set to one initially.

3. This process continues upto the number of dimensions DIM, with the highest-lowest coordinate difference being multiplied to area continuously before terminating.

4. Returns integer value of **area.**

contains()

Function Declaration: **bool contains(Rectangle r,Rectangle ele)**

1. This function takes two rectangles(**r and ele**) as the input and checks whether the rectangle ele lies completely inside rectangle r or not.
2. This function is later used in the **searching()** function.

```
bool contains(Rectangle r,Rectangle ele){
    //This functions checks whether ele lies completely inside r or not
    if((ele.lowest.x >= r.lowest.x) && (ele.lowest.y >= r.lowest.y) &&(ele.highest.x<= r.highest.x) &&(ele.highest.y<= r.highest.y))
    {
                return true;
    }
    else
    {
        return false;
    }
}
```

intersects()

Function Declaration: **bool intersects(Node *node, Rectangle *search)**

1. This function is to find if the given node has some overlap with the given rectangle.

2. This function is an intermediate function used primarily in **searching()**

3. The overlap is calculated by comparing the difference between **lowest and highest x and y coordinate pairs** of rectangle points and the node points as greater or less than 0.

4. Returns **Boolean** Value.

```
bool intersects(Rectangle r,Rectangle ele){
    return(ele.highest.x >= r.lowest.x &&
            ele.lowest.x <= r.highest.x&&
            ele.highest.y >= r.lowest.y&&
            ele.lowest.y <= r.highest.y);
}
```

searching()

Function Declaration:

Rectangle* searching(Node* child, Rectangle r)

1. This function takes two parameters, **pointer to node in R-tree(child)** and **Rectangle r** to search for.
2. Using the *helper functions* **intersects** and **contains** invoked on the bounding rectangle of node **child** with same rectangle **r** to search for,
    a. If the **child is a leaf,** printMBR function is invoked to directly print all MBRs of the node **child**
    b. **Else,** for all children of node **child,** the **searching()** function is recursively using the same **rectangle r.**
3. If there is no intersection or containment, the function does not perform any further search.

overlap_area()

Function Declaration: **int overlap_area(Rectangle rect1, Rectangle rect2)**

1. This function takes two rectangles(**r and ele**) as the input and checks whether the rectangle ele lies completely inside rectangle r or not.
2. This function is later used in the **searching()** function.

calculateMbrThroughPoints()

Function Declaration: **Rectangle calculateMbrThroughPoints(Point pt[],int n)**

1. This function is to **calculate the Minimum Bounding Rectangle(MBR)** through a **given Points array**, along with its size. It first creates a new rectangle as an MBR.

2. The function allocates 4 integer values **xmin,xmax,ymin,ymax** to the coordinates of the first point. It then traverses all other points in the **Points array**, comparing all these values to corresponding -x and y-coordinates of these points.

3. The xmin,xmax,ymin,ymax are **updated or retained** based on each comparison result, to obtain four distinct coordinates values which are then fed to the rectangle's **highest and lowest coordinates,** making it the MBR.

4. Returns the **rectangle r** having correct coordinates to satisfy MBR property.

newMbrOfParentNode()

Function Declaration:
Rectangle newMbrOfParentNode(Rectangle parent, Rectangle child)

1. This function is to **determine the resulting MBR of parent Rectangle(parent) when a child rectangle(child)** is inserted into **parent**.

2. The function allocates 4 integer values **xmin,xmax,ymin,ymax.** Based on comparisons between **child or parent** rectangles' lowest or highest values, these four values are set to get the maximum and minimum coordinates in x and y dimension as per their naming.

3. A new **Rectangle r** made, whose highest x, lowest x, highest y, lowest y are assigned xmax, xmin, ymax, ymin respectively.

4. Returns the newly made **Rectangle r** (having appropriate coordinates to accommodate the child as well as the parent rectangle)

calculateEnlargement()

Function Declaration: **int calculate_enlargement(Rectangle r, Point p)**

1. This function is to **calculate the incremental area(enlargement)** occupied by the **Rectangle r** into which the new **Point p** is being inserted.

2. The function initialises integer value **old_area** to rectangle’s area, which is done through **invoking calculateArea()** on **r**. The **new_area** is also made.

3. Two **Rectangles r1 and r2** are made, with **r1’s** coordinates lowest and highest being highest to the **Point p value**. The function **newMbrOfParentNode() is invoked** with parameters r(as parent) and r1(as child) and **r2** is assigned this rectangle.

4. The **new_area** is now set to **calculateArea()** on r2, with the function returning the difference **new_area** and **old_area** as the final enlargement.

findMinimumBoundingRectangle()

Function Declaration:
Rectangle findMinimumBoundingRectangle(Node* node[], int numRectangles)

1. This function **finds the MBR** of a **node array node[]**, given **node[]** and **number of rectangles within the node – numRectangles**
2. First, a **Rectangle minBoundingRect** is made, set to the **MBR** of first node in **node array node[]**
3. For **numRectangles** iterations, starting from the second **node[]** entry, the **minBoundingRect rectangle's** lowest and highest coordinates are compared to each node(in **node[]) MBR's lowest and highest coordinates**, with the **minBoundingRect's** coordinates being updated appropriately
4. Returns the **minBoundingRect** as the **MBR for given Node array.**

RectMinimumBoundingRectangle()

Function Declaration:
Rectangle RectMinimumBoundingRectangle(Rectangle node[], int numRectangles)

1. Similar to the last function, this function **finds the MBR for the case where an array of Rectangles is given**, along with **number of rectangles within the node - numRectangles**
2. It has the same working and returns **minBoundingRect** as MBR of given **Rectangle array.**

preOrder()

Function Declaration:
void preOrder(Node *node)

1. This function takes a node as the input and then prints the **MBR** of each node thereafter in the pre-order manner.
2. It makes use of the **printMBR()** function which simply takes a rectangle as the input and then prints the MBR of the rectangle as output.
3. This function is called inside the function **preOrderOfTree(),** which takes an **RTree as the input** and assigns its **Root** as an input parameter to the preOrder function.

insertPointIntoLeaf()

Function Declaration: **void insertPointIntoLeaf(Node* node, Point[], int n)**

1. For inserting points into a leaf node, this function takes **a Node pointer node, array of Points pt and size of point array n.**
2. For n-iterations, the function sets the **node's** points to the **point array pt's entries**, incrementing number of children within the **node** thereafter.
3. It then creates a **Rectangle r**, invoking **calculateMbrThroughPoints** with with parameters **node→points** as **Points** array and **node→num_children** as size of array.
4. The Rectangle **r** is in fact the **node's MBR,** and is set accordingly.

insertChildIntoNode()

Function Declaration: **void insertChildIntoNode(Node* node, Node* child[], int n)**

1. As evident, this function **inserts** a given **child array(of Node type)** into given **Node node, having Node array child[]** as the **n** number of children to be inserted into **node.**
2. For **n** iterations(considering ith entry), similar to the previous function the **node→children** array's entries are equated to **child[i]**, with **num_children** being incremented and **child[i]→parent** being set to node.
3. A **Rectangle r** is made, following which the **findMinimumBoundingRectangle()** function is invoked on parameters **node→children** as array of nodes and **node→num_children** as number of children to be inserted

AdjustMbr()

Function Declaration: **void AdjustMbr(Node* node)**

1. This function is to **update the MBR of given Node(node)** after any insertion or splitting to maintain correctness of MBRs.
2. It first creates a new Node **new, assigning it to the node**.
3. The function then **invokes newMbrOfParentNode()** with parameters **new→parent→mbr,new→mbr,** assigning **new** to its parent(**new→parent)** until new→parent points to **NULL**

```
void AdjustMbr(Node* node){
    Node* new = (Node*)malloc(sizeof(Node)); new = node;
    while(new->parent!=NULL){
        new->parent->mbr = newMbrOfParentNode(new->parent->mbr,new->mbr);
        new = new->parent;
    }
}
```

ChooseLeaf()

Function Declaration: **Node* ChooseLeaf(Node* node, Point p)**

As declared in the 1984 paper, this **recursive** function acts as one of the *helper functions* for **insertion()**, **to select the optimal leaf where insertion of the** incoming point **p is most suitable** within Node **node.**

a. *Base Case:* Node is leaf: The node is returned as point can be directly inserted.

Else the function relies on enlargement and area factors to determine the chosen child in the following manner:

1. Node **chosen_child** and integer value **min_enlargement set to resultant of invoking calculateEnlargement()** on the **node's** first child's **MBR** alongside point **p.**

2. For the remainder of node's children, considering the ith entry(Iteratively upto **num_children**):
   a. **curr_enlargement** equated to **calculateEnlargement()** on ith node's **MBR**
   b. If **curr_enlargement lesser than min_enlargement**
      i. Previous pointer n is set to i to retain it
      ii. min_enlargement set to curr_enlargement
      iii. Chosen_child set to the ith child of node.

ChooseLeaf()

c. Else if **min_enlargement cur_enlargement are equal:** *Tie-breaking based on area*
   i. **area1 and area2** - two parameters set to **calculateArea()** for **MBR** of ith child’s MBR and nth child’s(the one retained earlier as index of chosen child;set to zero initially) MBR.
   ii. If **ith child’s MBR area(area1)** is smaller than **area2**, the min_enlargement is to be reset to the curr_enlargement, and **chosen_child** is updated like before.

2. **The code continues in other else case, iterating to find the chosen_child** and assigning it to the appropriate child.
3. **Returns the ChooseLeaf() function** on **chosen_child** with the same **point p**.

CBSPoint()

Function Declaration: **Node* CBSPoint(Node* node,Point p)**

1. This is used to implement Corner Based Splitting Algorithm at the leaf node where if a node is already housing 4 points in it and we want to add one more point in the leaf node then CBSPoint() is called.

2. We check if the given node is a root node or not as in case of a root node we will add an additional layer below and both the splitted node will point towards the root as parent.

3. We define counters for four corners and each counter is increased on the basis of the algorithm mentioned in the research paper.

4. We create four points array as well for the specific four corners and allocate the points to the corresponding array on the basis of the algorithm. Which states if c0>c2 then C0 goes to N1 otherwise N2 the and for c1 and c3 whichever is more goes to the node which has less entries

5. The Problem arises when both c1 and c3 are equal in this case we go for the least coverage area i.e in whatever case the sum or area of both the new resulting nodes is less is selected.

CBSNodes()

Function Declaration: **Node* CBSNodes(Node* node1,Node* node2)**

1. This function is used to implement **Corner Based Splitting** where node 1 is the parent node and node 2 is the one which needs to be added to the parent node after being inserted as a child and to decide on which axis the overflown node should be split into two new nodes

2. We check if the node is a root node or a non-root node as the splitted nodes later need to be assigned parents which won't take place in root node.

3. We define a counter for each corner, calculate the centre of each entry (object) in node N, record the object with the nearest corner by incrementing that corner's counter.

4. We Create dynamic arrays to store the exact indices for the specific "childs" which are used later to access those specific child nodes via the Point arrays mentioned below.

5. We create Point arrays for storing the array of child nodes split based on their distances to the corners under the condition of preventing Diagonal Split.

CBSNodes()

6. We have also considered the tie-breaker case of c1 = c3 where we try both the combinations of c0 with c3; c2 with c1 and c0 with c1; c2 with c3 and choose the correct based on **minimum area enlargement**

7. This is ensured by using the Rectangle structures for storing temporarily the MBRs using **RectMinimumBoundingRectangle()** and then calculating the area using **calculateArea()** for R1, R2, R3 and R4 whose respective sum is **minimized** as considered from the two cases mentioned above.

AdjustTree()

Function Declaration: **void AdjustTree(Node* node, Node* child)**

This **recursive functions** provides for any **adjustment required within the node provided whenever a given child enters the tree.** It acts as a *helper function for* **insertion().**

a. Base Case: If node inputted is Null, then the function returns.
b. Case I: **Number of children of node < M** - *CHILD IS SIMPLY INSERTED*
    i. **insertChildIntoNode()** is invoked with parameters **node and child**(address), setting it to **leaf** (through isleaf = 1 as parameter)
c. Case II: **Number of children of node ⩾ M** - *CBS SPLITTING NEEDED*
    i. Node **newNode** is assigned memory
    ii. If node is root (node's parent is NULL), then **newNode** is assigned to result of invoking **CBSNodes()** through parameters as **node and child.**
    iii. Else the same process as c.(ii) happens, but additionally
        1. newNode's parent is set to the node's parent
        2. AdjustTree is called recursively using parameters node→parent,newNode, for the remaining tree to be checked for adjustment

The creation of **newNode allows us to recursively call AdjustTree() in the final sub-case.**

Insertion()

Function Declaration: **RTree* insertion(RTree* temp, Point p)**

1. Through a thorough implementation of multiple functions, **the insertion()** function inserts a given **point p into Rtree temp**, while implementing the desired **Corner Based Splitting as needed to develop the R-tree** in the manner as desired.
2. First, a **Node node** is made, and **ChooseLeaf()** is invoked on root of R-tree **temp** with point **p** to be inserted into R-tree. Node is assigned to the result of this.
3. For the case where **node** has lesser children than **M,**
   a. **insertPointIntoLeaf()** is invoked through **node** and point **p(address)**.
   b. The **node is** put through **AdjustMBR(),** to correct MBR after insertion.
4. Else, if **node** has children $\geqslant$ **M,**
   a. Another node **split_node** is created, and *Corner Based Splitting* **is invoked through CBSPoint** on **node**(given as result of **ChooseLeaf()**) and **p**.
   b. Adjustments occur through
      i. **AdjustTree()** - using node→parent and split_node
      ii. **AdjustMbr() -** on **node** itself
5. Returns the **Rtree temp** with the point p **inserted adequately.**

Thank You!