DATABASE MANAGEMENT SYSTEMS LABORATORY

CS39202

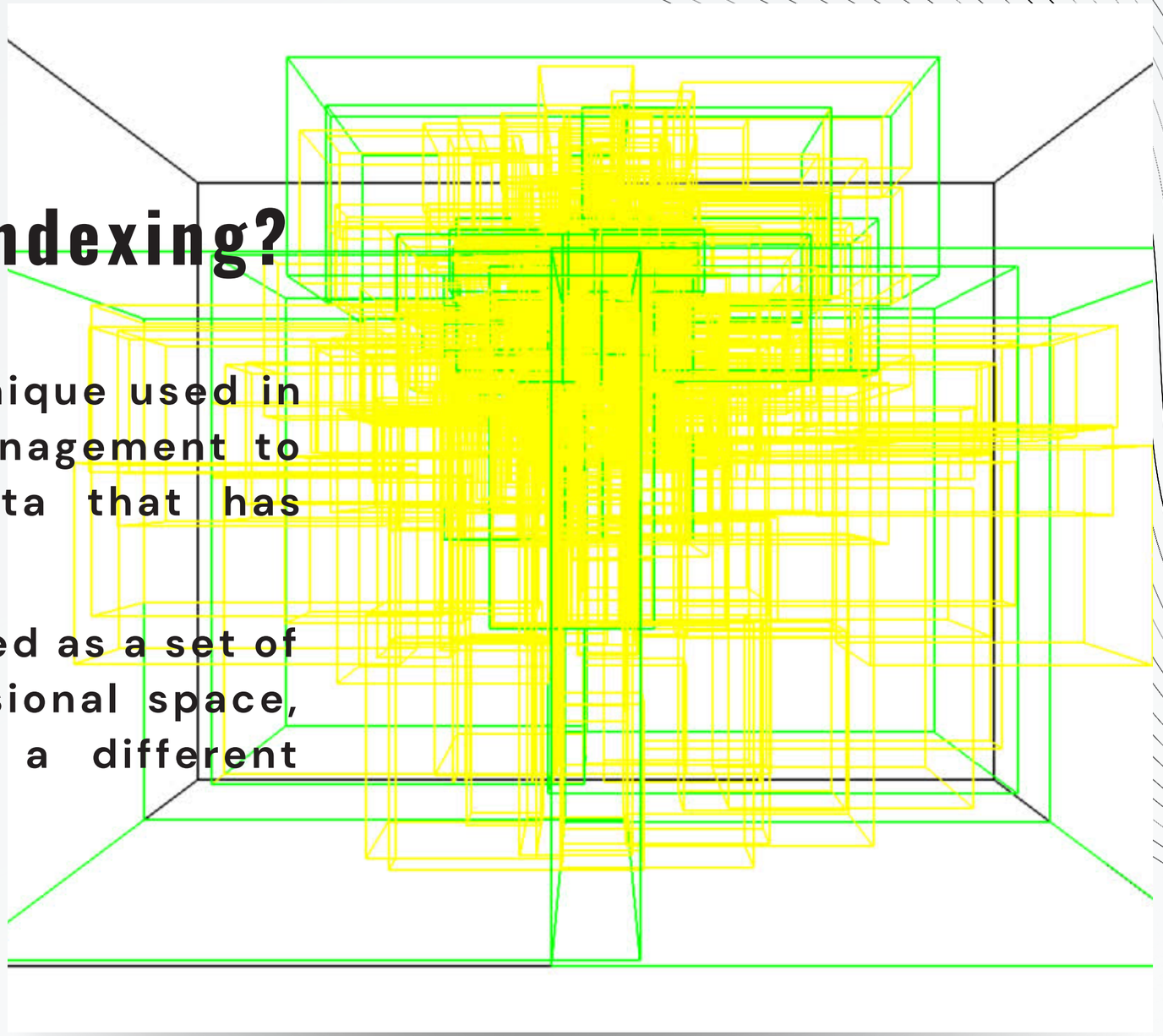
Term Project

HIGH DIMENSIONAL INDEXING

Implementation of R-tree

What is High Dimensional Indexing?

- High-dimensional indexing is a technique used in computer science and database management to efficiently search and retrieve data that has multiple dimensions or attributes.
- In a database, data can be represented as a set of points or objects in a multi-dimensional space, where each dimension represents a different attribute of the data.



What is Its Use?

Multidimensional indexing can be highly useful in a variety of scenarios, such as when searching for a specific location or set of locations within a geographic area, when analyzing large datasets with multiple attributes, or when querying time-series data for specific periods or trends. It can also be used to optimize query performance and reduce the amount of data that needs to be processed, leading to faster and more efficient processing of data.

Why not use Traditional Indexing Techniques?

When dimensions increase the efficiency of traditional indexing techniques decreases due to the phenomenon known as "**curse of dimensionality**".
Also these techniques may require a large number of lookups to find the desired data, leading to slower query performance. Thus they can’t be used.

Hegemony of R-Tree

B+ TREES

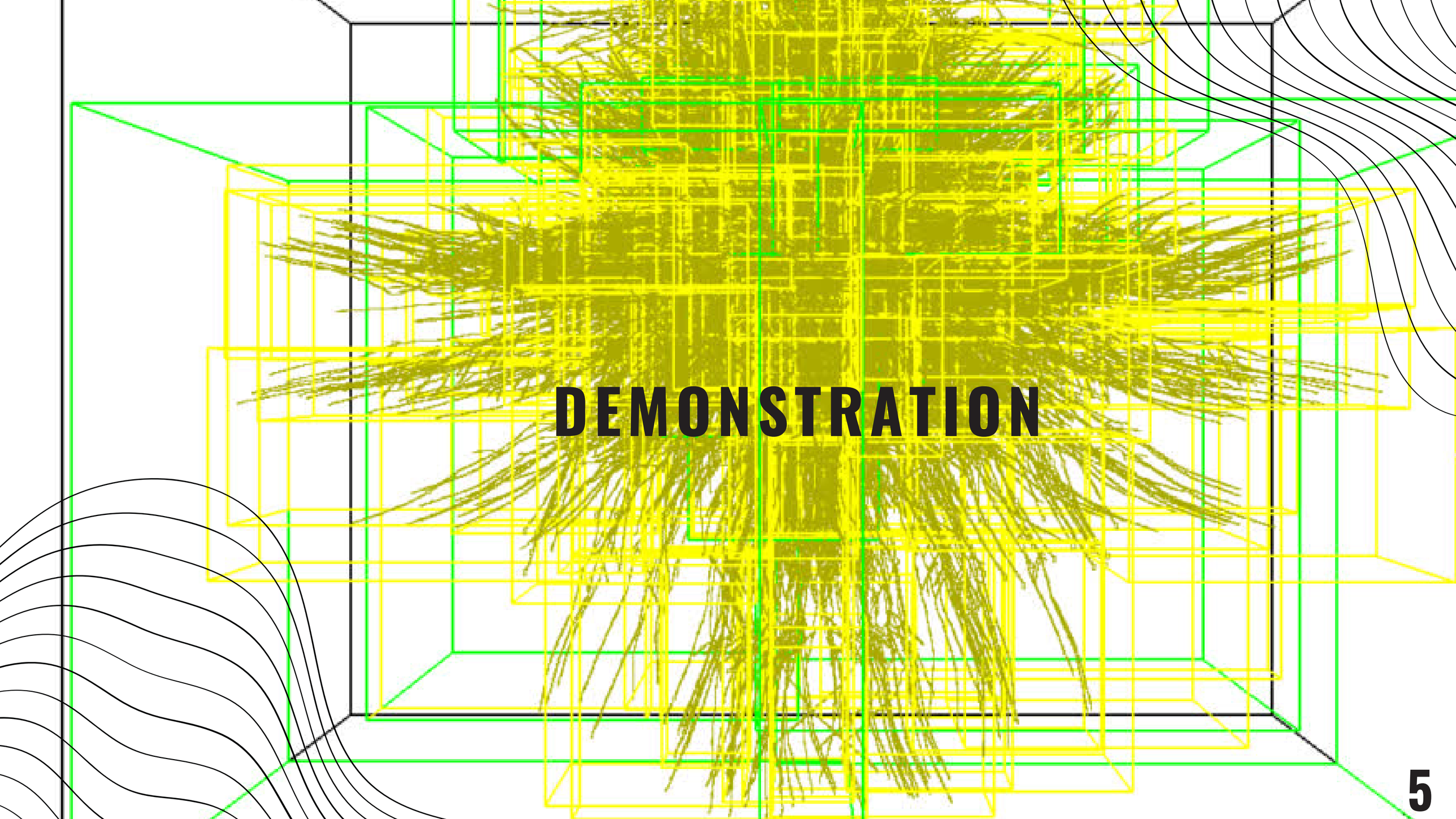
R-Tree is more suited for handling multi-dimensional indexes in comparison to B+ Trees.

KD TREES

R-Trees are more well designed to handle overlapping regions in comparison to KD-Trees.

QUAD TREES

R-trees use a hierarchical structure where each node in the tree is representing a subset of the multi-dimensional space. This structure allows for efficient pruning of the search space resulting in faster processing of queries in comparison to Quad Trees.



DEMONSTRATION

Steps Involved

Representing Images as multi-dimensional vectors using Deep Learning Model

STEP-1

Insert these vectors corresponding to the images in the R-Tree

STEP-2

Enter a query image

STEP-3

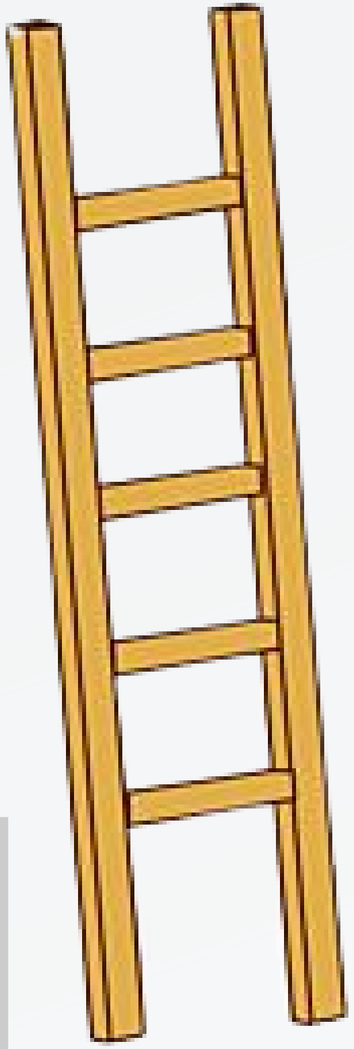
Convert this image into a multi-dimensional vector

SETP-4

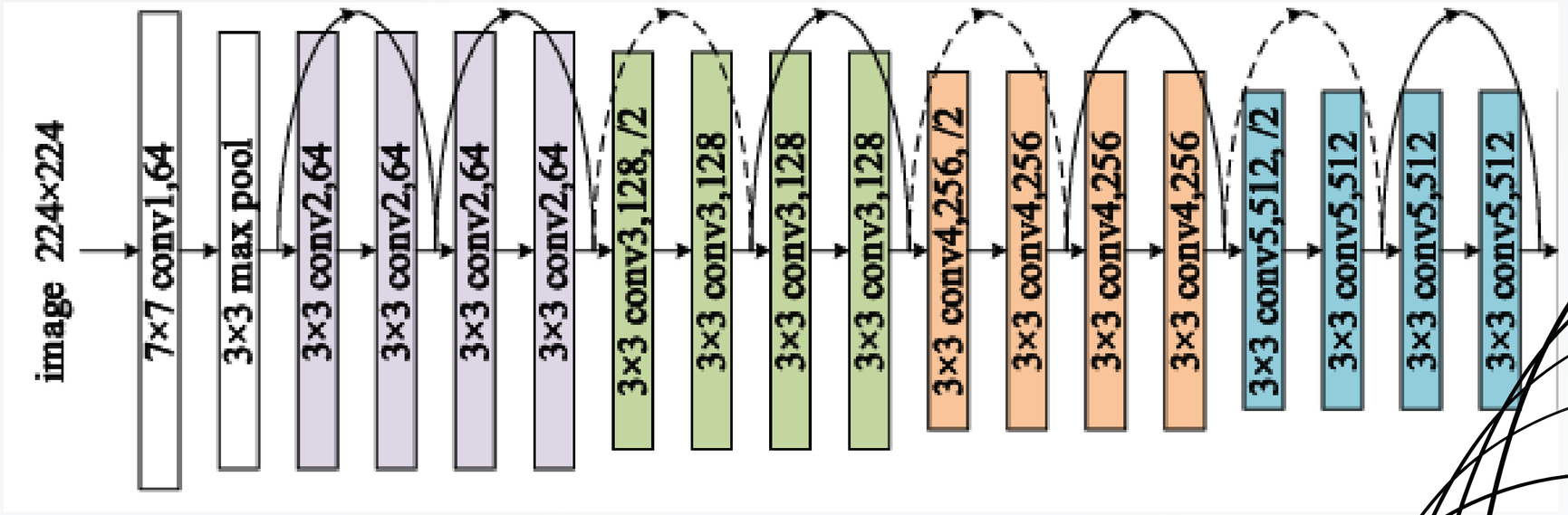
Find nearest neighbors to this multi-dimensional vector

STEP-5

Feature Extraction



We perform feature extraction using a ResNet-18 model pre-trained on ImageNet. Input images are resized to 224x224 and passed through the ResNet backbone to obtain a 1000 dimensional feature vector which is used for R-tree indexing.



Implementation of R-Tree

Insertion

- For insertion of a node,we would start from the root node and go to the leaf node where the current entry would be inserted.
- If the leaf node where it would be inserted becomes overfull then it would be split into 2 nodes and change would be propagated to the parent nodes.
- This continues until we find a node which does not become overfull after insertion.

Deletion

- For deletion of a node, first the multi-dimensional vector of the image has to be passed.
- The leaf node is first found where this feature vector is stored.
- After that this feature vector is deleted from that leaf node.
- And if the leaf node now has less than m/2 children then this leaf node is dropped and merged with its siblings. Similarly this process is recursively followed for the parents.

Searching

- Start from the root node and go into k branches which are nearest to the feature vector we are searching
- As we reach the leaf node, check if the current element has lesser distance than any of the current elements present in it
- If this is true, insert this element and remove the element which is farthest from the heap.
- Thus at the end we would get the closest k neighbors

Implementation of R-Tree

INSERTION

```
def insert(self, point):
    if self.root is None:
        self.root = RTreeNode(point, None)
    else:
        node = self._choose_leaf(self.root, point)
        node.children.append(RTreeNode(point, None))
        if len(node.children) > 4:
            self._split_node(node)

def _choose_leaf(self, node, point):
    if node.obj is not None:
        return node
    min_dist = float('inf')
    min_node = None
    for child in node.children:
        dist = self._distance(child.bbox, point)
        if dist < min_dist:
            min_dist = dist
            min_node = child
    return self._choose_leaf(min_node, point)

def _split_node(self, node):
    left, right = self._split_bbox(node.bbox)
    left_node = RTreeNode(left, None)
    right_node = RTreeNode(right, None)
    for child in node.children:
        if self._overlap(left, child.bbox):
            left_node.children.append(child)
        elif self._overlap(right, child.bbox):
            right_node.children.append(child)
```

SEARCHING

```
def nearest(node, point, k, heap):
    if node.is_leaf():
        for obj in node.objects:
            distance = compute_distance(obj, point)
            heap.push(distance, obj)
        while len(heap) > k:
            heap.pop()
    else:
        ct=0
        for child in node.children:
            distance = compute_distance(child.rectangle, point)
            if distance < heap.max_distance() or ct<k:
                ct++
                nearest(child, point, k, heap)
```

DELETION

```
def delete(self, point, node):
    if isinstance(node, LeafNode):
        node.children.remove(point)
        if len(node.children) < self.m // 2:
            if node is self.root:
                if len(node.children) == 0:
                    self.root = None
                return node
            else:
                parent = node.parent
                parent.children.remove(node)
                if len(parent.children) < self.m // 2:
                    parent.merge_or_redistribute()
                return parent
        else:
            return node
    else:  # node is an InternalNode
        for child in node.children:
            if child.bbox.contains(point):
                child = self._delete_point(point, child)
                if len(child.children) < self.m // 2:
                    if child is self.root:
                        if len(child.children) == 0:
                            self.root = None
                        return child
                    else:
                        parent = child.parent
                        parent.children.remove(child)
                        if len(parent.children) < self.m //
                            parent.merge_or_redistribute()
                        return parent
        return node
```

Results

Searching for Cat Images

Query Image

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Query Image

Results of Elephant Images

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Results

Searching for Tiger Images

Query Image

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Query Image

Results of Dog Images

Matching Image 1

Matching Image 2

Matching Image 3

Matching Image 4

Our Project can be Further Extended to:

Audio Signals

Using this we can find similar songs to the song which we enjoy.

Using this we can find research papers similar to the one which we are currently interested in.

Text Data

Medical Data

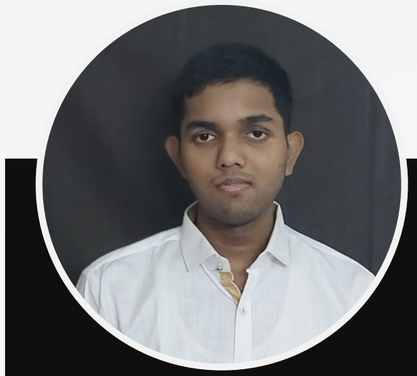
We can input medical data of a patient and find its similarity with data already present to diagnose diseases.

Predict future stock prices by inserting current data and finding its similarity with the previous ones.

Financial Data

THANK YOU

Team: The SQL Squad



Gaurav Malakar

20CS10029

Monish N

20CS30033

Abhijeet Singh

20CS30001

Gopal

20CS30021

Roopak Priydarshi

20CS30042