# R-TREE DATA STRUCTURE FOR SPATIAL INDEXING

## Seminar (IT290) Report

Submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

by SULAIMAN MUHAMMAD
(17IT143)

DEPARTMENT OF INFORMATION TECHNOLOGY NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL,
MANGALORE -575025

APRIL, 2019

# D E C L A R A T I O N

I hereby *declare* that the *Seminar (IT290) Report* entitled "**R-Tree Data Structure for Spatial Indexing"** which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a ***bonafide report of the work carried out by me.*** The material contained in this project report has not been submitted to any University or Institution for the award of any degree.

Sulaiman Muhammad
17IT143
Signature of the Student:

Department of Information Technology

Place : NITK, SURATHKAL
Date : 11th April 2019

# CERTIFICATE

This is to certify that the Seminar entitled **" R-Tree Data Structure for Spatial Indexing "** has been presented by **Sulaiman Muhammad**, a student of IV semester B.Tech. (I.T), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on 11th April 2019, during the even semester of the academic year 2018 - 2019, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Examiner-1 Name
Signature of the Examiner-1 with Date

Examiner-2 Name
Signature of the Examiner-2 with Date

Guide Name
Signature of the Guide with Date

Place: NITK Suratkal

Date: 11th April 2019

# ABSTRACT

In order to handle spatial data efficiently, as required in computer aided design and geo-data applications, a database system needs an index mechanism that will help it retrieve data items quickly accordmg to their spatial locations. However, traditional indexing methods are not well suited to data oblects of non-zero size located m multi-dimensional spaces. As an effective way for fetching data from database, index of a spatial database can rapidly access the data requested by a specific query without traversing whole database. In particular, a spatial database contains a large volume of data in GIS, so an index for database is necessary. R tree is a popular method for accessing spatial database. It was invented by Guttman in 1984. R trees are direct extension of B-tree in multi–dimensional space and a height-balanced tree as all leaf nodes appear on the same level. In this report, a dynarmc index structure called an R-tree which meets this need, and give algorithms for ADT operations like searching, insertion and deletion is descibed and implemented. Also, k-NN algorithm for query searching is implemented. Finally, few variants of the R-Tree are discussed.

TABLE OF CONTENTS

# CHAPTER 1. INTRODUCTION

**R-Trees** are tree data stuctures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Antonin Guttman in 1984 and has found significant use in both theoretical and applied contexts. A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest gas station" (although not taking roads into account). The R-tree can also accelerate nearest neighbour search for various distance metrics, including great-circle distance.

## 1.1 Spatial Indexing and R-Trees

Spatial data management has been an active area of intensive research for more than two decades.
In order to support spatial objects in a database system several issues should be taken into consideration such as: spatial data models, indexing mechanisms, efficient query processing, cost models. One of the most influential access methods in the area is the R-tree structure proposed by Guttman in 1984 as an effective and efficient solution to index rectangular objects in VLSI design applications. They aimed at handling geometrical data, such as points, line segments, surfaces, volumes, and hypervolumes in high-dimensional spaces. R-trees were treated in the literature in much the same way as B-trees. Since then, several variations of the original structure have been proposed towards providing more efficient access, handling objects in high-dimensional spaces, supporting concurrent accesses, supporting I/O and CPU parallelism, efficient bulk-loading. It seems that due to the modern demanding applications and after the academia has paved the way, recently the industry recognized the use and necessity of R-trees. The simplicity of the structure and its resemblance to the B-tree, allowed developers to easily incorporate the structure into existing database management systems in order to support spatial query processing.

Spatial indexing benefits applications like Geographical Information System, Computer Aided Designing and Multimedia etc. Many spatial index structures like quad tree, k-d tree, R-Tree and grid files etc., have been proposed in the literature. R-Tree is found to be the most widely used spatial index structure for its efficiency in performance and simplicity in implementation. In this index structure, the objects are represented as Minimum Bounding Rectangles (MBRs). R-Tree consists of root, non-leaf and leaf nodes. The nodes in R-Tree have a maximum and minimum capacity. When a node reaches the maximum capacity, the node is split such that the number of objects in each node lies between the minimum and maximum limit. Different variants of R-Tree have been proposed in the literature to give better performance in case of some queries or applications. The benchmark queries in spatial domain include range, nearest neighbor, join, topological and nearest surrounder queries. There are many variants in each of these queries.

## 1.2 R-Tree Idea

The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree; the "R" in R-tree is for rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.

As with most trees, the searching algorithms (e.g., intersection, containment, nearest neighbour search) are rather simple. The key idea is to use the bounding boxes to decide whether or not to search inside a subtree. In this way, most of the nodes in the tree are never read during a search. Like B-trees, this makes R-trees suitable for large data sets and databases, where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory. When data is organized in an R-tree, the neighbors within a given distance r and the k nearest neighbours of all points can efficiently be computed using a spatial join. This is beneficial for many algorithms based on such queries

The key difficulty of R-tree is to build an efficient tree that on one hand is balanced (so the leaf nodes are at the same height) on the other hand the rectangles do not cover too much empty space and do not overlap too much (so that during search, fewer subtrees need to be processed). R-trees do not guarantee good worst-case performance, but generally perform well with real-world data. While more of theoretical interest, the (bulk-loaded) Priority R-Tree variant of the R-tree is worst-case optimal, but due to the increased complexity, has not received much attention in practical applications so far.

# CHAPTER 2. LITERATURE REVIEW

## 2.1 **A State-of-Art in R-Tree Variants for Spatial Indexing**

*Authors : Lakshmi Balasubramanian, M. Sugumaran*

Nowadays, indexing has become essential for fast retrieval of results. Spatial databases are used in many applications which demand faster retrieval of data. These data are multi-dimensional. Designing index structure for spatial databases is current area of research. R-Tree is the most widely used index structure for multi-dimensional data. Many variants of R-Tree has evolved with each performing better in some aspect like query retrieval, insertion cost, application specific and so on. In this work, state-of-art of variants in R-Tree is presented. This paper provides an idea of the present development in spatial indexing and paves way for the researchers to explore and analyze the difficulties and trade-offs in the work.

## 2.2 **The R+-Tree : A Dynamic Multi-Dimensional Index for Objects**

*Authors :* T*imos Sellis, Nick Roussopoulos and Christos Faloutsos*

The problem of indexing multidimensional objectsis considered. First, a classification of existing methods is given along with a discussion of themajor issues involved in multi-dimensional data indexing. Second, a variationto Guttman’s R-trees (R+-trees) that avoids overlapping rectangles in intermediate nodes of the tree is introduced.Algorithms for searching, updating, initial packingand reorganization of the structure are discussed indetail. Finally, we provide analytical results indicating that R+-trees achieve up to 50% savings indisk accesses compared to an R-tree when searching files of thousands of rectangles

## 2.3 **Fast k Nearest Neighbour Search for R-tree Family**

*Authors : Joseph Kuan, Paul Lewis*

A simplifed k nearest neighbour (knn) search for the R-tree family is proposed in this paper. This method is modifed from the technique developed by Roussopoulos. The main approach aims to eliminate redundant searches when the data is highlycorrelated. We also describe how MINMAXDIST calculations can be avoided using MINDIST as the onlydistance metric which gives a significant speed up. Our method is compared with Roussopoulos et al.'s knn search on Hilbert R-trees in different dimensions, and shows that an improvement can be achieved on clustered image databases which have large numbers of data objects very close to each other. However, our method only achieved a marginally better performance of pages accessed on randomly distributed databases and random queries far from clustered objects, but has less computation intensity.

# CHAPTER 3. TECHNICAL DISCUSSION

## 3.1 R-Tree Index Structure

An R-tree is an index structure for n-dimensional spatial objects analogous to a B-tree. It is a height balanced tree with records in the leaf nodes each containing an n-dimensional rectangle and a pointer to a data object having the rectangle as a bounding box. Higher level nodes contain similar entries with links to lower nodes. Nodes correspond to disk pages if the structure is disk-resident, and the tree is designed so that a small number of nodes will be visited during a spatial search. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of records representing spatial objects, and each record has a unique identifier which can be used to retrieve it. We approximate each spatial object by a bounding rectangle, i.e. a collection of intervals, one along each dimension:

$$I = ( I_0 , I_1 ,....., I_{n-1} )$$

where n is the number of dimensions and $I_i$ is a closed bounded interval [a,b] describing the extent of the object along dimension i. Alternatively $I_i$ may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely.

Leaf nodes in the tree contain index record entries of the form,

$$( I , \textit{tuple-identifier} )$$

where tuple-identifier refers to a tuple in the database and I is an n-dimensional rectangle containing the spatial object it represents.

Non-leaf nodes contain entries of the form

$$( I , \textit{child-pointer} )$$

where child-pointer is the address of another node in the tree and I covers all rectangles in the lower node's entries. In other words, I spatially contains all data objects indexed in the subtree rooted at I's entry.

Let M be the maximum number of entries that will fit in one node and let m < M be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

(1) Every leaf node contains between m and M index records unless it is the root.

(2) For each index record *( I, tuple-identifier )* 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 between m and M children unless it is the root.

(4) For each entry *( I , child-pointer )* 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 leas. two children unless it is a leaf.

(6) All leaves appear on the same level.

The height of an R-tree containing N index records is at most $\log_m(N)$, because the branching factor of each node is at least m. The maximum number of nodes is [ $N/m + N/m^2 + \ldots + 1$ ]. Worst-case space utilization for all nodes except the root is m/M. Nodes will tend to have more than m entries, and this will decrease tree height and improve space utilization. If nodes have more than 3 or 4 entries the tree will be very wide, and almost all the space will be used for leaf nodes containing index records. The parameter m can be varied as part of performance tuning.

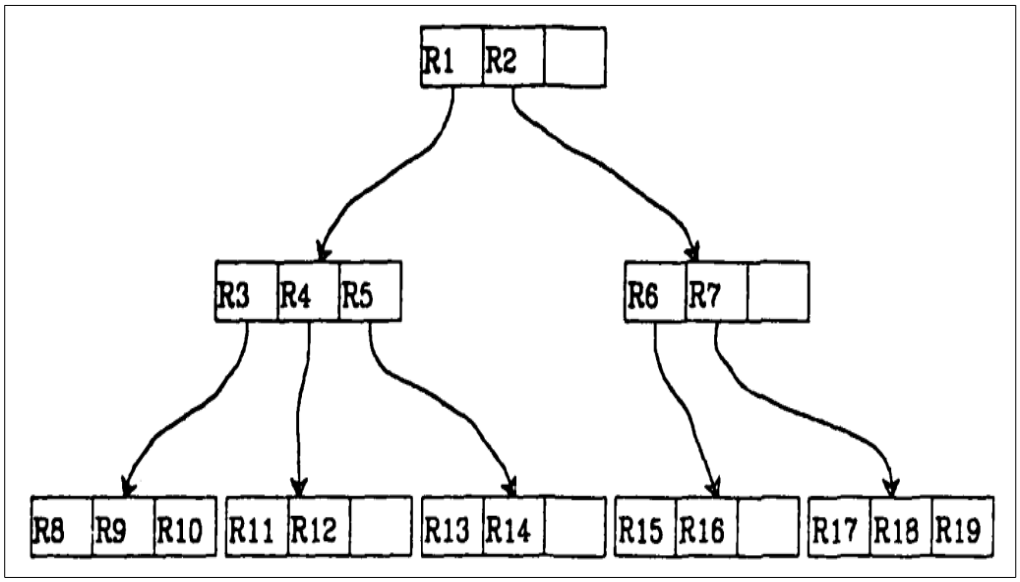


*Figure 3.1*

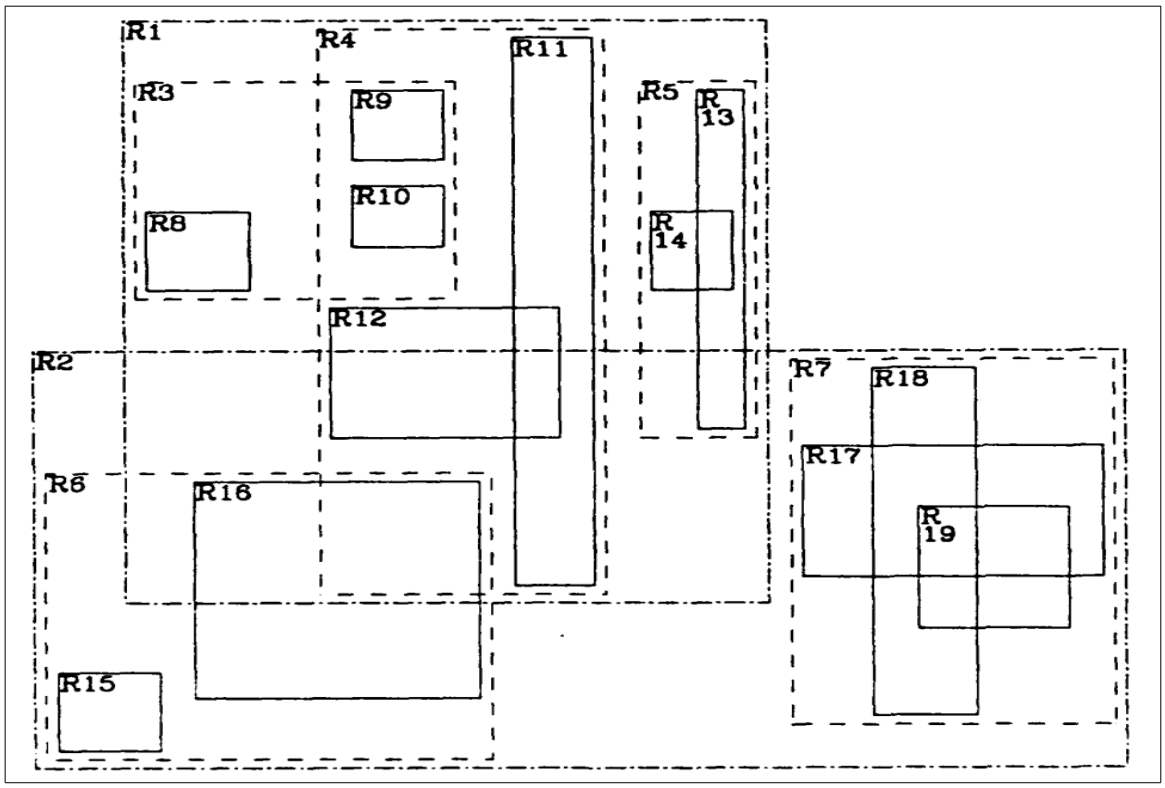


*Figure 3.2*

## 3.2 Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. However when it visits a non-leaf node it may find that any number of sub- trees from 0 to M need to be searched; hence it is not possible to guarantee good worst-case performance. Nevertheless with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area.

In the following we denote the rectangle part of an index entry E by E.1, and the tuple-identifier or child-pointer part by E.p.

**Algorithm Search**

Given an R-tree whose root node is T, find all index records whose rectangles overlap a search rectangle S.

*S1. [Search subtrees.]*

If T is not a leaf, check each entry E to determine whether E.I overlaps S.

For all overlapping entries, invoke Search on the tree whose root node is

pointed to by E.p.

*S2. [Search leaf node.]*

If T is a leaf, check all entries E to determine whether E.I overlaps S.

If so, E is a qualifying record.

## 3.3 Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

**Algorithm Insert**

Insert a new index entry E into an R-tree.

*I1. [Find position for new record.]*

Invoke ChooseLeaf to select a leaf node L in which to place E.

*I2. [Add record to leaf node.]*

If L has room for another entry, install E. Otherwise invoke SplitNode to obtain L and LL containing E and all the old entries of L.

*I3. [Propagate changes upward.]*

Invoke ExpandTree on L, also passing LL if a split was performed.

*I4. [Grow tree taller.]*

If node split propagation resulted in the root being split, create a new root whose children are the two nodes resulting from the split.

**Algorithm ChooseLeaf**

Select a leaf node of an R-tree in which to place a new index entry E.

*CLI. [Initialize.]*

Set N to be the root node.

*CL2. [Leaf check.]*

If N is a leaf, return N.

*CL3. [Choose subtree.]*

If N is not a leaf, let F be the entry in N whose rectangle F.I needs least enlargement to include E.I. Resolve ties by choosing the entry with the rectangle of smallest area.

*CL4. [Descend until a leaf is reached.]*

Set N to be the child node pointed to by F.p and repeat from CL2.

**Algorithm ExpandTree**

Ascend from a leaf node L in an R-tree to the root, adjusting covering rectangle! and propagating node splits as necessary.

*ET1. [Initialize.]*

Set N=L. If L was split previously, set NN to be the resulting second node.

*ET2. [Check if done.]*

If N is the root, stop.

*ET3. [Adjust covering rectangle in parent entry.l*

Let P be the parent node of N, and let $E_N$ be N's entry in P.

Adjust $E_N$I so that it tightly encloses all entry rectangles in N.

*ET4. [Propagate node split upward.]*

If N has a partner NN resulting from an earlier split, create a new entry $E_{NN}$ with $E_{NN}$.p pointing to NN and $E_{NN}$.I enclosing all rectangles in AN. Add $E_{NN}$ to P if there is room. Otherwise, invoke SplitNodle to produce P and PP containing $E_{NN}$ and all P's old entries.

*ET5. [Move up to next level.]*

Set N=P and set NN=PP if a split occurred. Repeat from ET2.

Algorithm SplitNode is described in Section 3.5.

## 3.4 Deletion

**Algorithm Delete**

Remove indexrecord E from an R-tree.

*D1. [Find node containing record.]*

Invoke FindLeaf to locate the leaf node L containing E. Stop If the record was not found.

*D2. [Delete record.]*

Remove E from L.

*D3. [Adjust tree.]*
Invoke CondenseTree to adjust the covering rectangles on the path from L to the root, to eliminate under-full nodes, and to propagate node eliminations up the tree.
*D4. [Shorten tree.]*
If the root node has only one child after the tree has been adjusted, make the child the new root.

**Algorithm FindLeaf**
Given an R-tree whose root node is T, find the leaf node con-
taining the index entry E.
*FLI. [Search subtrees.]*
If T is not a leaf, check each entry F in T to determine if F.! overlaps E.I. For each such entry invoke FindLeaf on the tree whose root is pointed to by F.p until E is found or all entries have been checked.
*FL2. [Search leaf node for record.]*
If T is a leaf, check each entry to see if it matches E. If E is found return T.

**Algorithm CondenseTree**
Given an R-tree leaf node L from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them smaller if possible.
*CT1. [Initialize.]*
Set N=L.
Set Q, the set of eliminated nodes, to be empty.
*CT2. [Find parent entry unless root has been reached.]*
If N is the root, go to CT6. Otherwise let P be the parent of N, and let EN be N's entry in P.
*CT3. [Eliminate under-full node.]*
If N has fewer than m entries, delete EN from P and add N to set Q.
*CT4. [Adjust covering rectangle.]*
If N has not been eliminated, adjust EN. to tightly contain all entries in N.
*CT5. [Move up one level in tree.]*
Set N=P and repeat from CT2.
*CT6. [Re-insert orphaned entries.]*
Re-insert all entries of nodes in set Q. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Algorithm Insert, but entries from higher-level nodes must be placed higher in the tree. This is done so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

## 3.5 Node Splitting
When an attempt is made to add an entry to a full node containing M entries, the collection of M+ 1 entries must be divided between two nodes. The division should be done in a way that makes it as unlikely as

possible that both new nodes will need to be examined on subsequent searches. Since the decision to visit a node is based on whether the search area overlaps the covering rectangle for the node's entries, the total area of the two covering rectangles should be minimized. Figure 3.2 illustrates this point. The area of the covering rectangles in the "bad split" case is much larger than in the "good split" case. Note that the same criterion was used in procedure ChooseLeaf to decide where to insert a new index entry: at each level in the tree, the subtree was chosen whose covering rectangle would have to be enlarged least. We now turn to algorithms for partitioning the set of M+ 1 entries into two groups, one for each new page.

### 3.5.1 A Quadratic-Cost Algorithm

This algorithm attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible. The cost is quadratic in M and linear in the number of dimensions. The algorithm picks two of the M+I entries to be the first elements of the groups which will make up the new nodes. The pair chosen is the one that would waste the most area if both entries were put in the same group, i.e. the area of the rectangle covering both entries, minus the areas of the rectangles in the entries, would be greatest. Then the remaining entries are selected one at a time and assigned to a group. At each step the area expansion required to add each entry to each group is calculated, and the entry chosen is the one showing the greatest difference between the two groups in the expansion required to include it.

**Algorithm Quadratic Split**

Divide a set of M+ 1 index entries into two groups.

*QSI. [Pick first entry for each group.]*

Apply Algorithm PickSeeds to choose two entries to be the first elements of the groups.

Assign each to a group.

*QS2. [Check if done.]*

If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.

*QS3. [Select entry to assign.]*

Invoke Algorithm PickNext to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accommo date it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

**Algorithm PickSeeds**

Select two entries to be the first elements of the groups.

*PS1. [Calculate inefficiency of grouping entries together.]*

For each pair of entries El and E2, compose a rectangle J including E1.I and E2.I

Calculate d=area(J) - area(El.I) – area(E2.I).

*PS2. [Choose the most wasteful pair.]*

Choose the pair with the largest d to be put in different groups.

**Algorithm PickNext**

Select one remaining entry for classification in a group.

*PN1. [Determine cost of putting each entry in each group.]*

For each entry E not yet in a group, calculate d1=the area increase required in the covering rectangle of Group 1 to include E.I. Calculate d2 similarly forGroup 2.

*PN2. [Find entry with greatest preference for one group.]*

Choose the entry with the greatest difference between d1 and d2. If more than one has the same lowest difference pick any of them.

### 3.5.2 A Linear-Cost Algorithm

This algorithm is linear in M and in the number of dimensions. First it selects seed entries for the two groups, choosing the two whose rectangles are the most widely separated along any dimension.

Then it processes the remaining entries without ordering them in any special way, placing each in one of the groups. Algorithm Linear Split is identical to Quadratic Split but uses a different version of PickSeeds. PickNext simply chooses any of the remaining entries.

**Algorithm LinearPickSeeds**

Select two entries to be the first elements of the groups.

*LPS1. [Find extreme rectangles along all dimensions.]*

Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.

*LPS2. [Adjust for shape of the rectangle cluster.]*

Normalize the separations by dividing by the width or the entire set along the corresponding dimension.

*LPS3. [Select the most extreme pair.]*

Choose the pair with the greatest normalized separation along any dimension.

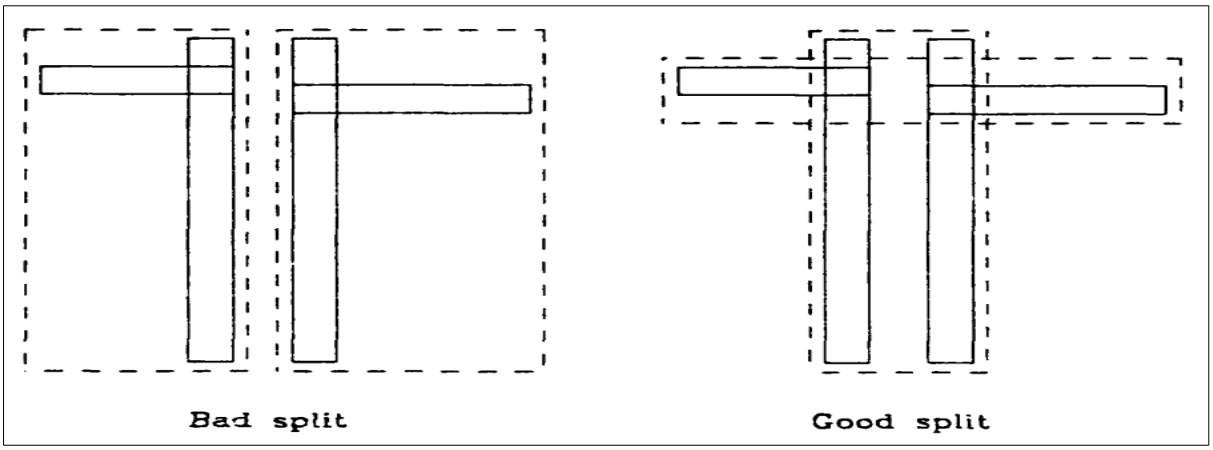


*Figure 3.3*

## 3.6 Variants

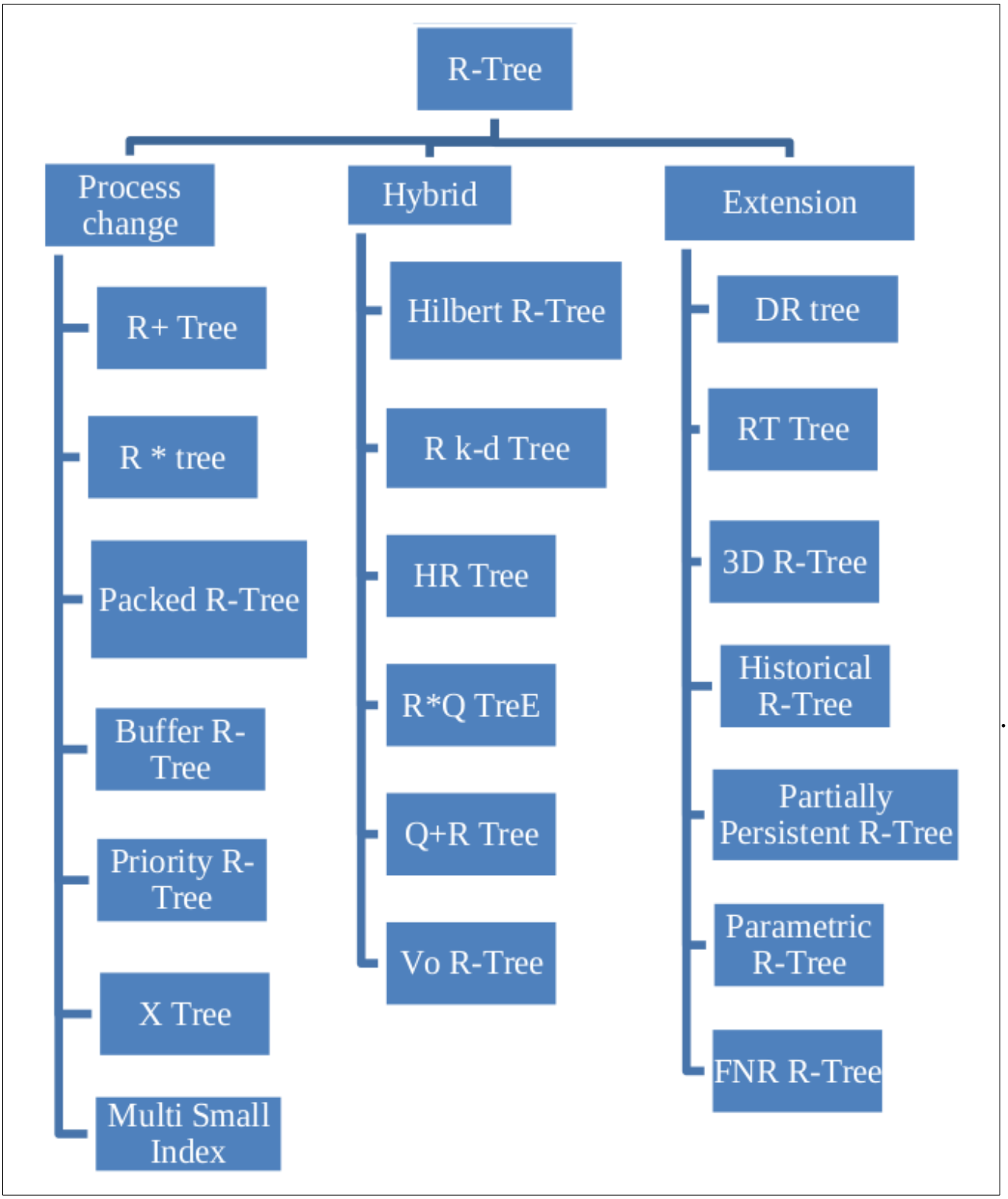


*Figure 3.4*

## 3.7 Nearest Neighbour Search

k-nearest neighbor (k-NN) search is one of the commonly used query in database systems. It has its application in various domains like data mining, decision support systems, information retrieval, multimedia and spatial databases, etc. When k-NN search is performed over large data sets, spatial data indexing structures such as R-trees are commonly used to improve query efficiency.

**Algorithm**

*k-NN1.* Calculate the MIN distance from the point to BB in that node

*k-NN2.* If Min is less than MaxMin ,Enter that node else check for the next BB

*k-NN3.* Calculate the distance of the entries inside that BB and and if found to be less than MaxMin then replace MaxMin'th neighbour with the current one

*k-NN4.* Update MaxMin to be furthest distance of those k neighbours

# 4. CONCLUSION

Although ‘R-trees have grown everywhere’ because of their simplicity and their satisfactory average performance, only a small subset of them have been successfully used by researchers and developers in prototype and commercial database systems. The R-tree is the most influential SAM and has been adopted as the index of choice in many research works regarding spatial and multidimensional query processing. Taking into consideration the work performed so far, we can state that the R-tree is for the spatial databases, what the B-tree is for alphanumeric data types. In fact, a serious reason for its acceptance is exactly the resemblance to the B-tree.

Considering the work performed on R-trees we realize that contains almost all aspects concerning a database system: query processing, query optimization, cost models, parallelism, concurrency control, recovery. This is the main reason why gradually database vendors adopted the R-tree and implemented it in their products for spatial data management purposes.

# 5. REFERENCES

[1] A. Guttman, —R-trees: A Dynamic Index Structure forSpatial Searching‖, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp.47-57, 1984.

[2] Timos K. Sellis, Nick Roussopoulos and ChristosFaloutsos, —The R+-Tree: A Dynamic Index for Multi-Dimensional Objects‖, Proceedings of 13th International Conference on Very Large Data Bases, pp.507-518, 1987.

[3] Antonio Leopoldo Corral Liria (1981). “Algorithms for the Processing of Spatial Queries using R-trees. Pairs Query and its Application on Spatial Databases”

[4] Lakshmi Balasubramanian,M. Sugumaran, —A State-of-Art in R-Tree Variants for Spatial Indexing, International Journal of Computer Applications (0975 – 8887) Volume 42 - No.20, March 2012