# 一個有效空間資料擷取的九方區域樹

# NA-trees: A Nine-Areas Tree for Efficient Data Access in Spatial Database Systems

張玉盈†, 廖正煌‡
Ye-In Chang†, and Cheng-Huang Liao‡
†資訊工程學系 ‡應用數學學系
†Dept. of Computer Science and Engineering ‡Dept. of Applied Mathematics
國立中山大學
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China
{E-mail: changyi@cse.nsysu.edu.tw}

摘要

在本篇論文中，我們考慮在一個大型且動態的資料中做正確符合查詢，所謂正確符合查詢即是在空間資料庫中找出某一個特定的資料物件；而所謂的大型，指的是絕大多數的資料都存放在第二儲存體中；動態意謂能隨時做插入及刪除物件的指令，所以資料結構可以不被事先建立。在這篇論文中，一個新的資料結構；九方區域樹 *(NA-tree)* 即是爲了解決此種問題而產生的。由我們模擬實驗分析中，證明出我們的九方區域樹比起*R* 樹 *(R-trees)*、*(R⁺-trees)* 和*R* 檔案 *(R-files)* 都有較少的搜尋代價。

(關鍵詞: 正確符合查詢, *G* 樹, 範圍查詢, *R* 檔案, *R* 樹, $R^+$樹, 空間索引)

## Abstract

*In this paper, we consider the problem of retrieving spatial data via exact match queries from a large, dynamic index, where an exact match query means to find the specific data object in a spatial database. By large, it means that most of the index must be stored on secondary memory. By dynamic, it means that insertions and deletions are intermixed with queries, so that the index cannot be built beforehand. A new data structure, a Nine-Areas tree (denoted as a NA-tree), is presented as a solution to this problem. From our simulation study, we show that our NA-trees has lower search cost (in terms of number of visited nodes) than R-trees, $R^+$-trees, and R-files.*

(Key Words: *exact match queries, G-tree, range queries, R-files, R-trees, $R^+$-trees, spatial index.*)

[1]This research was supported by National Science Council of the Republic of China, NCS-88-0408-E-110-004

## 1 Introduction

An index based on objects' spatial location is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using one-dimensional ordering of key values, such as B-tree and ISAM indexes, do not work because the search space is multi-dimensional [9]. None of these solutions is, however, efficient, and therefore specialized structures are required to handle multidimensional queries [12]. Several hierarchical data structures have been proposed for handling multidimensional data. The k-d tree [1], grid method [2], K-D-B-tree [16], BD tree [5], grid file [4], hB-tree [13], MD tree [14], and G-tree [12] have been developed for handling point data. For region (non-zero size) data, the R-tree [9], $R^+$-tree [17], R-file [10], and GBD tree [15] have been developed. The quadtree [7] have been extended to manage points, lines, regions, and volume data.

In this paper, we consider the problem of retrieving multikey records via exact match queries. By *large*, it means that most of the index must be stored on secondary memory. By *dynamic*, it means that insertions and deletions are intermixed with queries, so that the index cannot be built beforehand [16]. A new data structure, a Nine-Areas tree (denoted as NA-tree), is presented as a solution to this problem. In this paper, we experience with the implementation of the NA-tree and present the results of an experimental performance comparison with three related structures: R-trees [9], $R^+$-trees [17], and R-files [10]. In particular, we aim at (1) efficient processing of exact match queries in large spatial data distributed uniformly, and (2) maintaining a reasonable lower bound on average disk utilization. From our simulation study, we show that our NA-trees has lower search cost (in terms of number of visited nodes) than R-trees, $R^+$-trees and R-files.

## 2 NA-Trees (Nine-Areas Trees)

In this Section, we first describe the bucket numbering scheme. Next, we describe the details of our structure. Then, we give algorithms for performing insertions and deletions operations, respectively. Finally, we present some difficult cases that some other tree structures are hard to handle, but the NA-tree can solve them easily.

### 2.1 The Bucket Numbering Scheme

A spatial object, e.g., a polygon, can take an arbitrary shape. A common way to characterize an object is by specifying its bounding rectangle, which is oriented parallel to the coordinate axes, say $X$ and $Y$. Thus an object $O$ is hereafter represented by its four bounding coordinates, $X_l$, $X_r$ (i.e. the leftmost and rightmost $X$ coordinates, respectively), $Y_b$, and $Y_t$ (i.e. the bottommost and topmost $Y$ coordinates, respectively). For simplicity, we assume that no two objects have identical $X$ or $Y$ bounding coordinates [19]. In our approach, we use two points, $L(X_l, Y_b)$ and $U(X_r, Y_t)$, to represent a spatial object, where $L$ is the lower left coordinate and $U$ is the upper right coordinate of the object.

A bucket is numbered as a binary string of 0's and 1's, the so-called DZ expression. The relationship between the space decomposition process and the DZ expression is as follows.

1. Symbols '0' and '1' in a DZ expression correspond to lower and upper half regions, respectively, for each binary division along the $y$-axis. When a space is divided on the $x$-axis, '0' indicates the left half, and '1' indicates right half sub-area.

2. The leftmost bit corresponds to the first binary division, and the $n$'th bit corresponds to the $n$'th binary division of the area made by the *(n-1)*'th division.

Figure 1 shows an example of these regions, and the DZ expression of the dark area is '0010*', because the area corresponds to "the lower half of the right half of the lower half of the left half" of the entire space [15]. Here, we convert the bucket numbers from binary to decimal form. The legend alongside in Figure 1 shows the equivalent binary representations of the bucket numbers appearing on the grid itself in a decimal form.

Based on this bucket-numbering scheme, we observe that the uptrend of bucket number is increased from southwest to northeast, as shown in

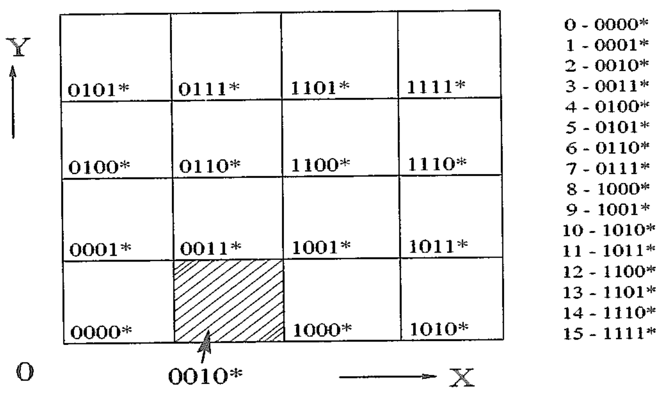



Figure 1: Space decomposition and DZ expression

Figure 2. Here, Figure 2-(b) shows the direction of the increasing order of bucket numbers in Figure 2-(a), which is called a N-order Peano curve [11]. This observation has motivated us to design a new data structure for spatial indexing. First, we use two points, $L(X_l, Y_b)$ and $U(X_r, Y_t)$, to record the region of a spatial object. Next, we calculate the corresponding bucket number of $L(X_l, Y_b)$ and $U(X_r, Y_t)$, respectively. The resulting pair of bucket number is noted as *spatial number*. That is, we can use the spatial number to record an object. For convenience, we use $O(l, u)$ to denote the spatial number, where $l$ is the bucket number of $L(X_l, Y_b)$ and $u$ is the bucket number of $U(X_r, Y_t)$. For example, in Figure 3, the spatial number of object $O$ is (12, 26). Moreover, we use a variable, $Max\_bucket$, to record the maximal bucket number (in a decimal form) of this area. In Figure 1, the maximal bucket number is 15 (1111). That is, $Max\_bucket = 15$.

### 2.2 Data Structure

Generally, tree structures handling multidimensional data are constructed with two types of nodes: internal nodes and leaf nodes. In our method, an internal node can have nine children, three children, or even just one. Since a leaf has no children, leaves are terminal nodes. Data can only be stored in a leaf, not in an internal node (unless it has only one child).

A NA-tree is a structure based on data classification by the bucket numbers. First, we decompose the whole spatial region into four regions. We let *region I* be the bucket numbers between 0 to $\frac{1}{4}(Max\_bucket+1)-1$, *region II* be the bucket numbers between $\frac{1}{4}(Max\_bucket + 1)$ to $\frac{1}{2}(Max\_bucket + 1) - 1$, *region III* be the bucket numbers between $\frac{1}{2}(Max\_bucket + 1)$ to $\frac{3}{4}(Max\_bucket + 1) - 1$, and *region IV* be the bucket numbers between $\frac{3}{4}(Max\_bucket + 1)$ to $Max\_bucket$, as shown in Figure 4-(a).

Based on this decomposition, we find that when an object is lying on the space, only nine

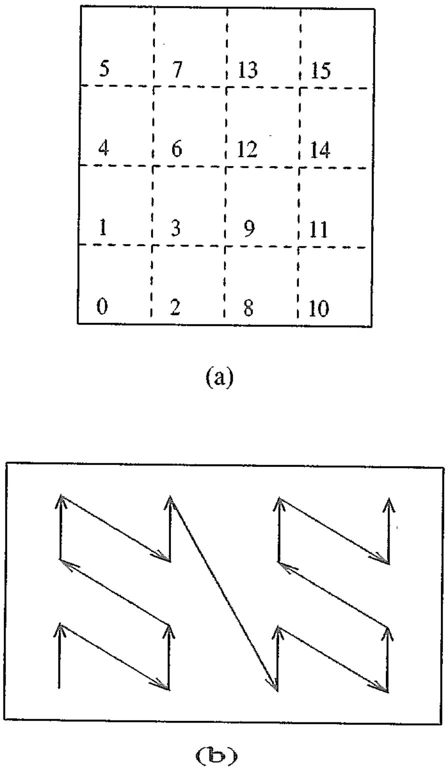



Figure 2: The bucket-numbering scheme: (a) bucket numbering; (b) N-order Peano Curve.

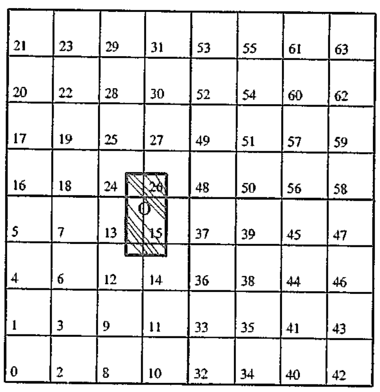



Figure 3: An example of the bucket numbering scheme, O(l, u) = (12, 26)

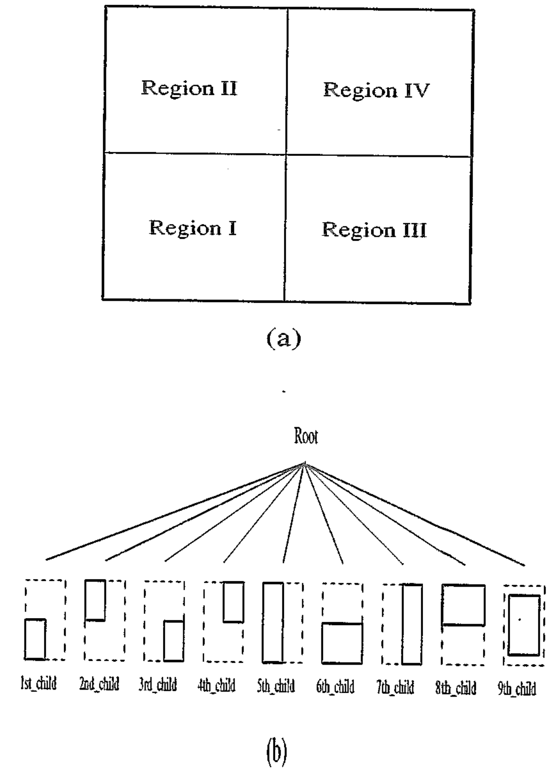



Figure 4: The basic structure of a NA-tree: (a) four regions; (b) nine cases.

cases are possible (as shown in Figure 4-(b)). Thus, an index (internal) node $p$ in a NA-tree may have the following nine children:

(1) for an object $O(l,u)$, $l$ and $u \in$ region I, $O$ is the first child of node $p$.

(2) for an object $O(l,u)$, $l$ and $u \in$ region II, $O$ is the second child of node $p$.

(3) for an object $O(l,u)$, $l$ and $u \in$ region III, $O$ is the third child of node $p$.

(4) for an object $O(l,u)$, $l$ and $u \in$ region IV, $O$ is the fourth child of node $p$.

(5) for an object $O(l,u)$, $l \in$ region I and $u \in$ region II, $O$ is the fifth child of node $p$.

(6) for an object $O(l,u)$, $l \in$ region I and $u \in$ region III, $O$ is the sixth child of node $p$.

(7) for an object $O(l,u)$, $l \in$ region III and $u \in$ region IV, $O$ is the seventh child of node $p$.

(8) for an object $O(l,u)$, $l \in$ region II and $u \in$ region IV, $O$ is the eighth child of node $p$.

(9) for an object $O(l,u)$, $l \in$ region I and $u \in$ region IV, $O$ is the ninth child of node $p$.

For the above nine children, they have three kinds of data structures. The data structures of *1st_child*, *2nd_child*, *3rd_child*, and *4th_child* are as follows:

```
struct nine_children
{ struct nine_children *1st_child;
struct nine_children *2nd_child;
struct nine_children *3rd_child;
```

```
struct nine_children *4th_child;
struct three_children *5th_child;
struct three_children *6th_child;
struct three_children *7th_child;
struct three_children *8th_child;
struct one_list *9th_child;}
```

The data structures of *5th_child* and *7th_child* are as follows:

```
struct three_children
{ struct three_children *5th_child;
struct three_children *7th_child;
struct one_list *9th_child;}.
```

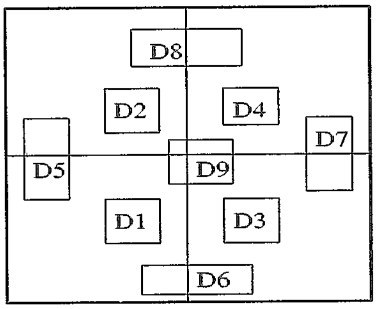
The data structures of *6th_child* and *8th_child* are as follows:

```
struct three_children
{ struct three_children *6th_child;
struct three_children *8th_child;
struct one_list *9th_child; }
```

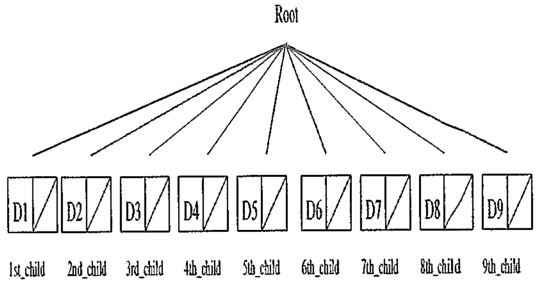
The data structure of *9th_child* is as follows:

```
struct one_list
{ data_object [1..bucket_capacity];
struct one_list *next_ptr;}
```

Leaf nodes in a NA-tree contain index object entries of the form (*entry_number*, *data*[1..*bucket_capacity*]), where *entry_number* refers to the number of objects in this leaf node, *data*[*bucket_capacity*] is an array to store object data, and *bucket_capacity* denotes the maximum number of entries which could be stored in the leaf node. Figure 5 shows an example of a NA-tree structure. Note that we do not split the spatial space; we just classify the spatial data objects by some rules.




(a)




(b)

Figure 5: An example: (a) the data; (b) the corresponding NA-tree structure (bucket_capacity = 2).

## 2.3 Algorithms

This section describes our algorithms for data insertion, deletion and answering exact match queries. The *Insertion* algorithm is shown in Figure 6. Function *Assign* and procedure *Split* which are used in the *Insertion* algorithm are shown in Figures 7 and 8, respectively. Basically, inserting a new rectangle in a NA-tree is done by searching the tree according to data classification and adding the rectangle in the leaf node. Finally, the overflowing node is split and the split may propagated to the children node, if it occurs.

In the *Insertion* algorithm as shown in Figure 6, the first step in inserting an object, $O(L,U)$, is to compute its spatial number. The function *Assign* is called with the coordinates of the point $(x_1, x_2)$ and the number of bits $b$, where we assume that the number of bits in the binary form of bucket number is $b$. The function *Assign* returns the bucket number, $l$ and $u$, of points $L(X_l, Y_b)$ and $U(X_r, Y_t)$, respectively. Therefore, the spatial number of this object is $(l, u)$. The *Assign* function shown in Figure 7 is used to compute the

```
procedure Insertion( O(L,U) );
begin
    l := Assign(X_l, Y_b, b);
    u := Assign(X_r, Y_t, b);
    /* Calculate L's and U's bucket numbers, (l,u), respectively */
    p := Root;
    repeat
    if (l ∈ I) and (u ∈ I) then p := p^ 1st_child;
    if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child;
    if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child;
    if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child;
    if (l ∈ I) and (u ∈ II) then p := p^ 5th_child;
    if (l ∈ I) and (u ∈ III) then p := p^ 6th_child;
    if (l ∈ III) and (u ∈ IV) then p := p^ 7th_child;
    if (l ∈ II) and (u ∈ IV) then p := p^ 8th_child;
    if (l ∈ I) and (u ∈ IV) then p := p^ 9th_child;
    until p is a leaf node;
    Add O into node p;
    if node p is full then Split(p);
end;
```

Figure 6: The *Insertion* procedure

```
Function Assign(x1, x2, b);
/* compute an initial bucket number */
begin
 P = " ";                /* null string */
 for k:=1 to b
     begin
       i := k mod 2;
       if (x_i < (l_i + h_i)/2) then
         begin
         concatenate "0" to P;
         h_i := (l_i + h_i)/2);
         end
       else
         begin
         concatenate "1" to P;
         l_i := (l_i + h_i)/2);
         end;
     end;
 change binary number (P) to decimal;
 return(P);
end;
```

Figure 7: The *Assign* function

```
procedure Split(p);
begin
   q := p;
   Let p be the index node;
   if p ∈ {1st_child, 2nd_child, 3rd_child,
         and 4th_child of p^ parent} then
      Create all 9 children of p
   else
   begin
      if p ∈ {5th_child and 7th_child of p^ parent}
then
     Create 5th_child, 7th_child, and 9th_child of p;
      if p ∈ {6th_child and 8th_child of p^ parent}
then
     Create 6th_child, 8th_child, and 9th_child of p;
   end;
   Re-Insert all objects in q;
   for each child of p do
      if (child) is full then Split(child);
end;
```

Figure 8: The *Split* procedure

```
procedure Deletion( O(L,U) );
begin
    l := Assign(X_l, Y_b);
    u := Assign(X_r, Y_t);
    /* Calculate L's and U's bucket numbers, (l,u), re-
spectively */
    p := Root;
    repeat
        if (l ∈ I) and (u ∈ I) then p := p^ 1st_child;
        if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child;
        if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child;
        if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child;
        if (l ∈ I) and (u ∈ II) then p := p^ 5th_child;
        if (l ∈ I) and (u ∈ III) then p := p^ 6th_child;
        if (l ∈ III) and (u ∈ IV) then p := p^ 7th_child;
        if (l ∈ II) and (u ∈ IV) then p := p^ 8th_child;
        if (l ∈ I) and (u ∈ IV) then p := p^ 9th_child;
    until p is a leaf;
    if O is not in p then show an error message
    else delete O from p;
    if p is empty then Merge(p);
end;
```

Figure 9: The *Deletion* procedure

DZ expression and return a decimal bucket number.

Next, according to the spatial number, we search the tree and find which leaf node is this object belong to. Finally, we insert this object into this leaf node and checking whether this leaf node is overflow. If this leaf node is overflow, then we execute the procedure *SPLIT*.

Deletion of a rectangle from a NA-tree is done by first locating the rectangle that must be deleted and then removing it from the leaf node. Finally, we will check whether this leaf node is *empty* or not, where *empty* means that there is no other objects in this leaf node. Figure 9 shows the *Deletion* algorithm. When an empty leaf node occurs, this empty node may merge with other sibling leaves. The *Merge* algorithm is shown in Figure 10.

The algorithm to process exact match query, as shown in Figure 11, is similar to the *Deletion* algorithm. To process exact match queries in a NA-tree, we search the tree according to data classification, and then check all data objects in the leaf node.

## 2.4 Difficult Cases in $R^+$-Trees

The $R^+$-tree allows the fast computation of search operators. However, the insertion and deletion of data objects may be much more complicated in turn [8]. First, the insertion of an object $O$ or its data interval $I_O$ may require the enlargement of *several* sibling intervals (i.e. intervals corresponding to sibling nodes). This is especially (but not exclusively) the case if $I_O$ overlaps several sibling intervals. In Figure 12-(a), $I_O$ has

```
procedure Merge(p);
begin
   q := p^ parent;
   release p;
   calculate entries of q;
   /* the entries is the number of objects in all
children of q */
   if (entries ≤ bucket_capacity) then
   begin
      create a new leaf node, n;
      move objects from q's children to n;
      n^ parent := q^ parent ;
      release q;
   end;
end;
```

Figure 10: The *Merge* procedure

```
procedure Exact_match_query( O(L,U) );
begin
   l := Assign(X_l, Y_b);
   u := Assign(X_r, Y_t);
   /* Calculate L's and U's bucket numbers, (l,u), re-
spectively */
   p := Root;
   repeat
      if (l ∈ I) and (u ∈ I) then p := p^ 1st_child;
      if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child;
      if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child;
      if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child;
      if (l ∈ I) and (u ∈ II) then p := p^ 5th_child;
      if (l ∈ I) and (u ∈ III) then p := p^ 6th_child;
      if (l ∈ III) and (u ∈ IV) then p := p^ 7th_child;
      if (l ∈ II) and (u ∈ IV) then p := p^ 8th_child;
      if (l ∈ I) and (u ∈ IV) then p := p^ 9th_child;
   until p is a leaf;
   if O is not in p then show an error message
   else output O from p;
end;
```

Figure 11: The *Exact_match_query* procedure

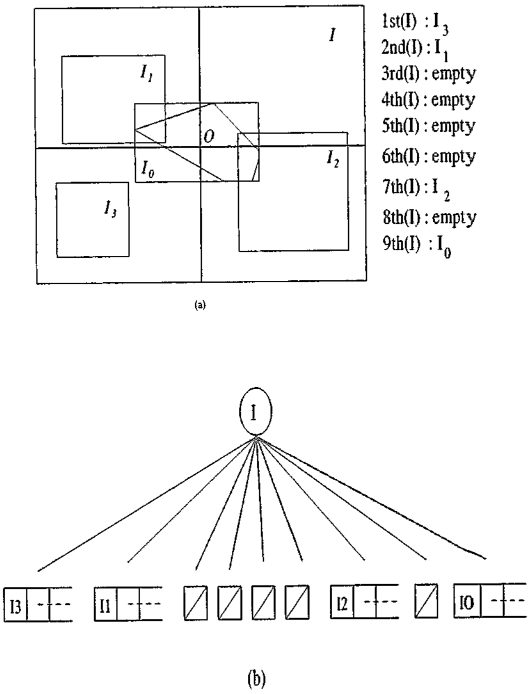



Figure 12: Case 1 in a NA-tree

to be inserted into both corresponding subtrees. $I_1$ and $I_2$ have to be enlarge in such a way that $I_0 \subset I_1 \cup I_2$ (without $I_1$ overlapping $I_2$). Each of these enlargements may require a considerable effort because it is always necessary to test for possible overlaps with sibling intervals. $I_0$ is inserted into all corresponding subtrees; the insertion may therefore cause the creation of several leaf entries. For this case, the NA-tree approach will create a new leaf node (or perhaps this leaf node has already existed), and then insert the data interval $I_0$ into the leaf node (as shown in Figure 12-(b)).

Second, there are situations where the enlargement step *inevitably* leads to overlaps (as shown in Figure 13-(a)). In this case, it is not possible to enlarge the sibling intervals $I_1 ... I_4$ in such a way that $I_0 \subset I_1 \cup ... \cup I_4$ without creating overlaps. It is therefore necessary to split one of the intervals, say $I_1$ into two subintervals $I_1$' and $I_1$" before the enlargement can take place [8]. For this case, the NA-tree approach can create a new leaf node (or perhaps this leaf node has already existed), and then insert the data interval $I_0$ into the leaf node (as shown in Figure 13-(b)).

## 3 Performance

In this Section, we compare the performance of R-trees, $R^+$-trees, R-files, and NA-trees.

All databases used in this performance evaluation are randomly generated sets of rectangles. Each rectangle was displaced at random within

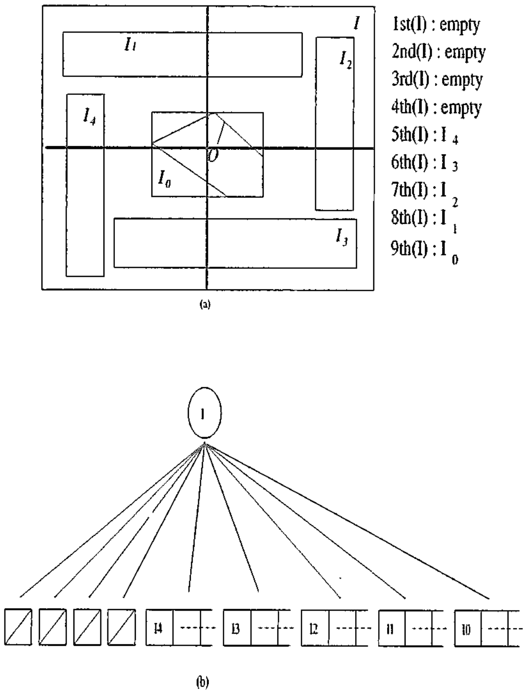



Figure 13: Case 2 in a NA-tree

the given two-dimensional data space; i.e. the data are uniform distribution.

There are two major parameters that characterize such a geometric database; the number $N$ of data objects (the *database size*) and their *average size, avg_size*, measured in percent of the size of the data space, i.e.,

$$avg\_size = \frac{\sum_{i=1}^{N} area_i / N}{The\ whole\ data\ space} \times 100\%.$$

We took measurements for six different databases containing 500, 1000, 2000, 3000, 4000, and 5000 rectangles of average size 0.0625%. *Bucket_capacity* have been tested for 10, where the *Bucket_capacity* is the maximum number of data containable in a leaf node. Hereafter, we represent the bucket capacity as $P$. Here, the search cost means the number of nodes visited and the insertion (deletion) cost means the number of internal nodes visited.

Now, we show some indicative results of the search performance of R-trees, R$^+$-trees, R-files, and NA-trees. For R-trees, we have implemented the originally published split algorithm (as described in [9]), the linear algorithm. The name, linear, for the split algorithm indicates their time complexity is in relation to the number of entries stored in the R-tree node which is to be split.

First, we make a comparison of the average search cost. For each spatial data file, we create 50 rectangles randomly to do exact match queries, and then calculate the average search cost of them. Figure 14 shows the results for the average size

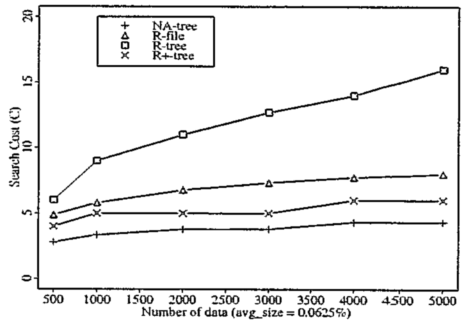



Figure 14: A comparison of search cost for processing an exact match query

| Tree structure | R-file | R-tree | R$^+$-tree | NA-tree |
|---|---|---|---|---|
| Storage utilization (%) | 70 ± 5 | 60 ± 5 | 60 ± 5 | 55 ± 5 |

Table 1: A comparison of storage utilization

0.0625%. From this figure, we observe that our NA-tree has the lowest search cost among these four strategies.

Next, we make a comparison of storage utilization, as shown in Table 1. From these results, we observe that the storage utilization in R-files is about $(70 \pm 5)\%$, R-trees is about $(60 \pm 5)\%$, R$^+$-trees is about $(60 \pm 5)\%$, and NA-trees is about $(55 \pm 10)\%$. Obviously, NA trees decrease the search cost at the cost of decreasing storage utilization.

For the insertion cost, let's concern the cases of the average cost of inserting 500, 1000, 2000, 3000, 4000, and 5000 rectangles based on a uniform distribution. From the result as shown in Figure 15, we observe that NA-trees have lower insertion cost than others.

For the deletion cost, let's concern the cases of average cost of deleting 50 rectangles in 500, 1000, 2000, 3000, 4000, and 5000 rectangles based on a uniform distribution. From the result as shown in

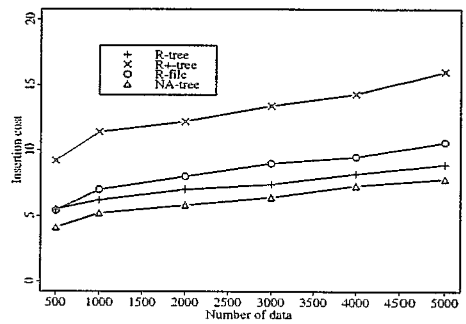



Figure 15: A comparison of insertion cost

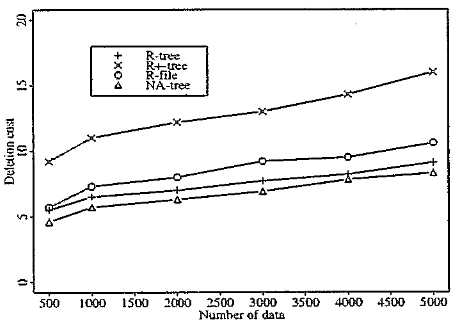



Figure 16: A comparison of deletion cost

Figure 16, we observe that NA-trees have lower deletion cost than others.

# 4 Conclusion

In this paper, we have proposed an efficient spatial index strategy, called a NA-tree, which is designed for paged secondary memory and it is dynamic; i.e., it can support arbitrary insertions and deletions of objects without any global reorganizations and without any loss of performance. Moreover, it is efficient to support exact match queries. How to process the *partial match queries* and *best match queries* is the future research work, where a *partial match query* means to report all data objects which are located in a specific line and a *best match query* means to find the nearest neighbor of the specific data object.

This research was supported by National Science Council of the Republic of China, NCS-88-0408-E-110-004.

# References

[1] Jon L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, Vol. 18, No. 9, pp. 509-517, Sept. 1975.

[2] J.L. Bentley and J.H. Friedman, "Data Structure for Range Search," *ACM Computing Surveys*, Vol. 11, No. 4, pp. 397-409, Dec. 1979.

[3] Elisa Bertino and Beng Chin Ooi, "The Indispensability of Dispensable Indexes," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 11, No 1, pp. 17-27, Jan./Feb. 1999.

[4] Henk Blanken, Alle Ubema, Paul Meek, and Bert van den Akker, "The Generalized Grid File: Description and Performance Aspects," *Proc. of IEEE Int. Conf. on Data Eng.*, pp. 380-388, 1990.

[5] Sivarama P. Dandamudi and Paul G. Sorenson, "Algorithms for BD Trees," *Software-Practice and Experience*, Vol. 16, No. 12, pp. 1077-1096, Dec. 1986.

[6] Volker Gaede and Oliver Gunther, "Multidimensional Access Methods," *ACM Computing Surveys.*, Vol. 30, No 2, pp. 170-231, June 1998.

[7] Irene Gargntini, "An Effective Way to Represent Quadtrees," *Communications of the ACM*, Vol. 25, No. 12, pp. 905-910, Dec. 1982.

[8] Oliver Gunther and Jeff Bilmes, "Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 3, No 3, pp. 342-356, Sept. 1991.

[9] Antonin Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. of ACM SIGMOD*, pp. 47-57, 1984.

[10] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer, "The R-File: An Efficient Access Structure for Proximity Queries," *Proc. of IEEE Int. Conf. on Data Eng.*, pp. 372-379, 1990.

[11] K.J. Li and Laurini Robert, "The Spatial Locality and a Spatial Indexing Method by Dynamic Clustering in Hypermap Systems," *Proc. of IEEE Int. Conf. on Data Eng.*, pp. 207-223, 1992.

[12] Akhil Kumar, "G-Tree: A New Data Structure for Organizing Multidimensional Data," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 6, No. 2, pp. 341-347, April 1994.

[13] David B. Lomet and Betty Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM Trans. on Database Systems*, Vol. 15, No. 4, pp. 625-658, Dec. 1990.

[14] Yasuaki Nakamura, Shigeru Abe, Yutaka Ohsawa, and Masao Sakauchi, "A Balanced Hierarchical Data Structure for Multidimensional Data with Highly Efficient Dynamic Characteristics," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 5, No. 4, pp. 682-694, Aug. 1993.

[15] Yutaka Ohsawa and Masao Sakauchi, "A New Tree Type Data Structure with Homogeneous Nodes Suitable for a Very Large Spatial Database," *Proc. of IEEE Int. Conf. on Data Eng.*, pp. 296-303, 1990.

[16] John T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. of ACM SIGMOD*, pp. 10-18, 1981.

[17] Timos Sellis, Nick Roussopoulos and Christos Faloutsos, "The $R^+$-tree: A Dynamic Index for Multi-dimensional Objects," *Proceedings of the 13th VLDB Conf.*, pp. 507-518, Brighton 1987.

[18] Shashi Shekhar, Sanjay Chawla, Siva Ravada, Andrew Fetterer, Xuan Liu, and Chang-Tien Lu, "Spatial Databases–Accomplishments and Research Needs," *IEEE Trans. on Knowledge and Data Eng.*, Vol. 11, No. 1, pp. 45-55, Jan./Feb. 1999

[19] Ching-Der Tung, Wen-Chi Hou, and Jiang-Hsing Chu, "Multi-Priority Tree: An Index Structure for Spatial Data," *Proceedings of International Computer Symposium*, pp. 1285-1290, 1994.