NA-Trees: A Dynamic Index for Spatial Data*

YE-IN CHANG, CHENG-HUANG LIAO+ AND HUE-LING CHEN
Department of Computer Science and Engineering
+Department of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, 804 Taiwan
E-mail: changyi@cse.nsysu.edu.tw

In non-standard database applications, such as geographic information processing or CAD/CAM, methods of access are required that support efficient manipulation of multidimensional geometric objects on secondary storage. Spatial data consists of spatial objects made up of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension. Being able to respond to spatial queries in a flexible manner places a premium on the appropriate representation of the spatial data. In order to be able to deal with proximity queries, an efficient spatial indexing strategy is needed. In this paper, we consider the problem of retrieving spatial data via *exact match queries* and *range queries* from a *large*, *dynamic* index, where an *exact match query* means finding the specific data object in a spatial database and a *range query* means reporting all data objects which are located in a specific range. By *large*, we mean that most of the index must be stored in secondary storage. By *dynamic*, we mean 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 NA-tree), is shown to be a solution to this problem. An NA-tree is designed for paged secondary storage to minimize the number of disk accesses during a tree search. From our simulation, we show that our NA-tree has a lower search cost (in terms of number of visited nodes) than the R-tree, R^+-tree, or R^*-tree.

Keywords: exact match queries, range queries, R-trees, R^+-trees, R^*-trees, spatial index

1. INTRODUCTION

Modern database systems are no longer limited to business applications. Non-standard applications such as robotics, computer vision, computer-aided design, and geographic data processing are becoming increasingly important, and geometric data plays a crucial role in many of these new applications [14]. In non-standard database applications, such as geographic information processing or CAD/CAM, methods of access are required that support efficient manipulation of multidimensional geometric objects on secondary storage. Spatial data consists of spatial objects made up of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension. Examples of spatial data include cities, rivers, roads, counties, states, crop coverage, mountain

Received September 1, 2000; revised April 18, 2001; accepted January 25, 2002.
Communicated by Ming-Syan Chen.
* This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-88-0408-E-110-004.

ranges, parts in a CAD system, etc. Examples of spatial properties include the extent of a given river, or the boundary of a given county, etc. Often it is also desirable to attach non-spatial attribute information such as elevation heights, city names, etc. to the spatial data. Such databases are used in many applications, including environmental monitoring, space, urban planning, resource management, and geographic information systems (GIS) [11, 31, 34, 35].

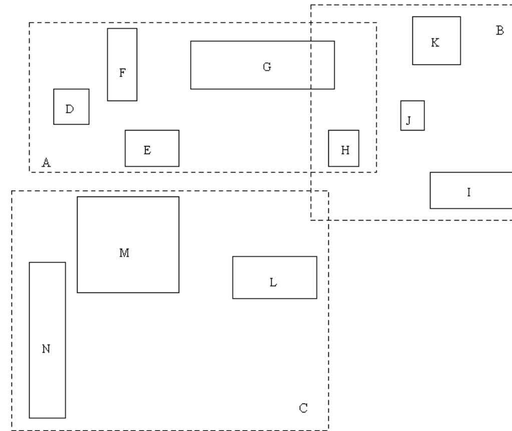
Consider an information retrieval system involving a file whose records are cities on a map, say of the continental United States. The cities could be stored in a certain data structure with latitude and longitude serving as the keys. Queries could take on many forms. An *exact match query* might be, "What is the city at 43°3' N latitude and 88° W longitude?" To find all cities in the Oklahoma Panhandle, one could pose a *range query* defining a rectangle bounded by latitudes 36°30' and 37° and longitudes 100° and 103° [6].

An index based on objects' spatial locations 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 [16]. However, none of these solutions is efficient, and therefore specialized structures are required to handle multi-dimensional queries [3, 5, 15, 23, 39]. Several hierarchical data structures have been proposed for handling multi-dimensional data. The k-d tree [6], grid method [7], K-D-B-tree [33], BD tree [9], grid file [8], hB-tree [26], MD tree [27], and G-tree [23] have been developed for handling point data. For region (non-zero size) data, the R-tree [16], R^+-tree [38], R^*-tree [4], R-file [18], GBD tree [29] and X-tree [1] have been developed. The quadtree [12] has been extended to manage points, lines, regions, and volume data.

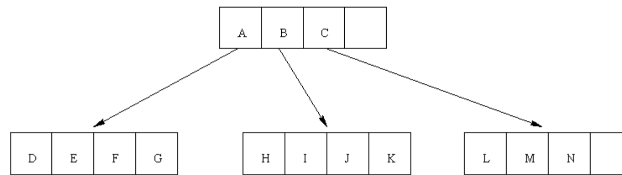
Among the access methods for region data, R-trees [10, 16, 19, 25, 28, 30, 41, 42], R^+-trees [38] and R^*-trees [4] are R-tree-based strategies, which have been widely referenced and compared in many new proposed spatial indexing strategies [2, 17, 21]. Moreover, they have been used in many advanced database applications, including multimedia [44], image databases [32, 40] and OLAP [36] which are based on R-trees, and Video Databases [24] which are based on R^+-trees.

An R-tree is a B^+-tree-like structure which stores multidimensional rectangles as complete objects without clipping them or transforming them to higher dimensional points [16]. R-trees are balanced trees that correspond to a nesting of *d*-dimensional intervals. An R-tree does not support exact match queries very well due to the unlimited growth and overlap of cells. Fig. 1 shows the structure of an R-tree. It is very hard to control the overlap during the dynamic splits of R-trees. Moreover, an R-tree may lead to performance losses during search operations; sometimes, one may have to traverse several search paths. For example, for a search window surrounding rectangle *H* shown in Fig. 1(a), both subtrees rooted at nodes *A* and *B* must be searched, although only the latter will return the qualifying rectangle *H*.

The main advantage of R^+-trees [38] compared to R-trees is the improved search performance, especially in the case of exact match queries. An R^+-tree avoids overlapping rectangles in intermediate nodes of the tree. However, the insertion and deletion of data objects may in turn be much more complicated. Fig. 2(a) indicates a different



(a)



(b)

Fig. 1. An R-tree example: (a) grouping in an R-tree; (b) an R-tree structure.

grouping of the rectangles in Fig. 1(a), and Fig. 2(b) shows the corresponding R^+-tree. (Note that the number of nodes in the R^+-tree is always greater than that of the R-tree because of the duplicate entries (for example, object *G* in Fig. 2).

Based on a careful study of the behavior of R-trees under different data distributions, Beckmann et al. [4] confirmed that the insertion phase is critical for good search performance. The design of the R^*-tree [4] therefore introduces a policy called *forced reinsert*. If a node overflows, it is not split right away. Rather, *p* entries are removed from the node and reinserted into the tree. The parameter *p* may vary. The R^*-tree strategy also takes the following objectives into account [11]: (1) region perimeter should be minimized; (2) overlap between bucket regions at the same tree level should be minimized; (3) storage utilization should be maximized. Fig. 3 shows an example of an R^*-tree, in which the sum of region perimeters is smaller than that of Fig. 1(a).

In this paper, we consider the problem of retrieving multikey records via exact match queries, and range queries from a large, dynamic index. By *large*, we mean that most of the index must be stored in secondary storage. By *dynamic*, we mean that insertions and deletions are intermixed with queries, so that the index cannot be built beforehand [33]. A new data structure, a Nine-Areas tree (denoted NA-tree), is presented as

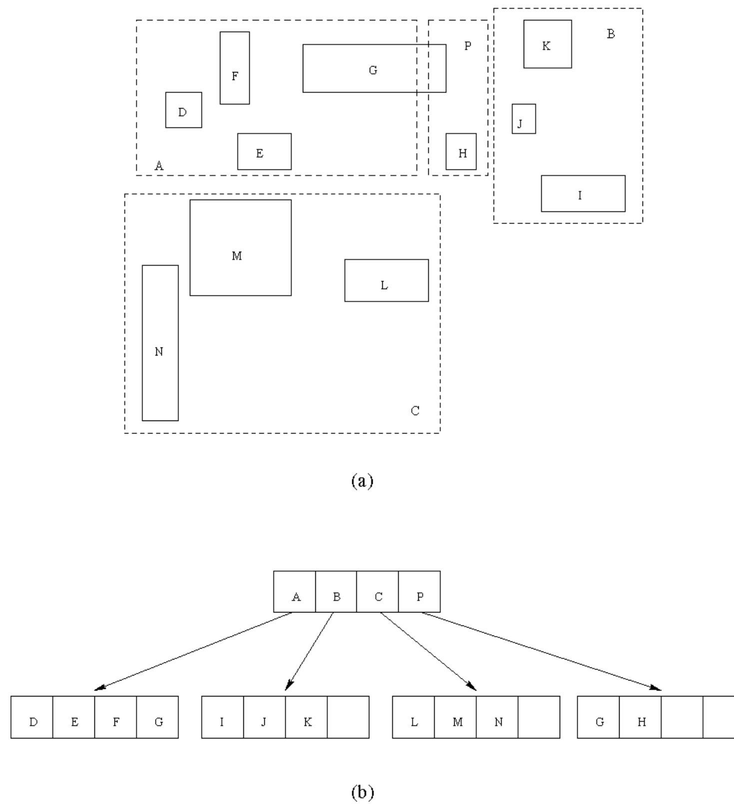


Fig. 2. An R^+-tree example: (a) grouping in an R^+-tree; (b) an R^+-tree structure.

a solution to this problem. Basically, our strategy is motivated by Kumar's ordering property [23]. In Kumar's ordering property, partitions are numbered in a way such that partitions which are spatially close to one another in a multi-dimensional space are also close in terms of their partition numbers. From our simulation, we show that our NA-tree has a lower search cost (in terms of number of visited nodes) than the R-tree [16], R^+-tree [38] or R^*-tree [18].

The rest of the paper is organized as follows. Section 2 presents our proposed NA-tree and describes algorithms for exact match queries and range queries. A comparison of the performance of NA-trees with R-trees, R^+-trees, and R^*-trees is reported in section 3. Finally, section 4 gives conclusions.

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. Finally, we present some difficult cases that some other tree structures are hard to handle, but which the NA-tree can easily solve them.

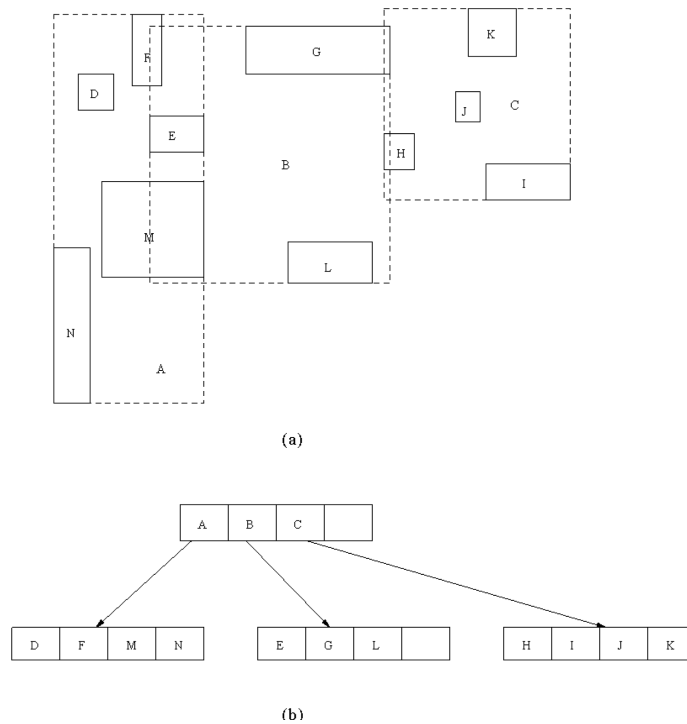


Fig. 3. An R*-tree example: (a) grouping in an R*-tree; (b) an R*-tree structure.

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 [43]. In our approach, we use two points, $L(X_l, Y_b)$ and $U(X_r, Y_t)$, to represent a spatial object, as shown in Fig. 4, where L is the lower left coordinate and U is the upper right coordinate of the bounding rectangle.

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.

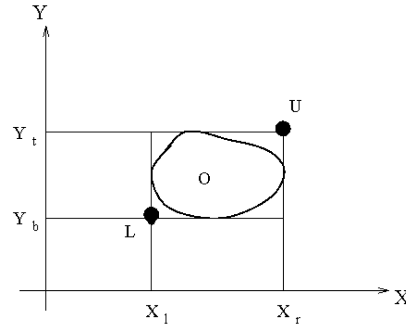


Fig. 4. An object representation: $O(L(X_l, Y_b), U(X_r, Y_t))$.

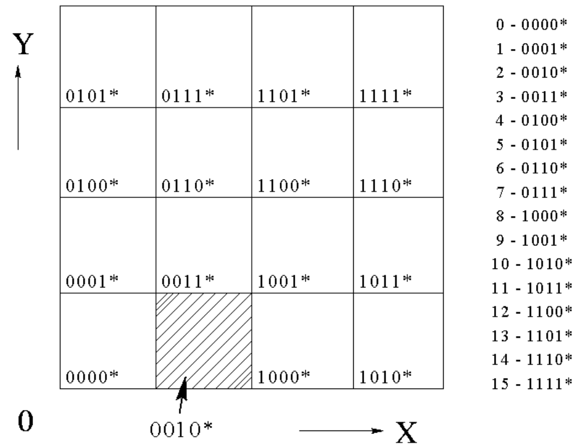


Fig. 5. Space decomposition and DZ expression.

Fig. 5 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 [29]. We convert the bucket numbers from binary to decimal form. The legend alongside Fig. 5 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 increases from southwest to northeast, as shown in Fig. 6. Fig. 6(b) shows the direction of the increasing order of bucket numbers in Fig. 6(a), which is called a N-order Peano curve [22]. This observation has motivated us to design a new data structure for spatial indexing. First, two points, $L(X_l, Y_b)$ and $U(X_r, Y_t)$, are used to record the region of a spatial object. Next, the corresponding bucket numbers of $L(X_l, Y_b)$ and $U(X_r, Y_t)$ are calculated. The resulting pair of bucket numbers is called the *spatial number*. The spatial number can be used 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 Fig. 7, the spatial number of object O is (12, 26). A variable, *Max_bucket*, is used to record the maximum bucket number (in decimal form) of this area. In Fig. 5, the maximum bucket number is 15 (1111), i.e., *Max_bucket* = 15.

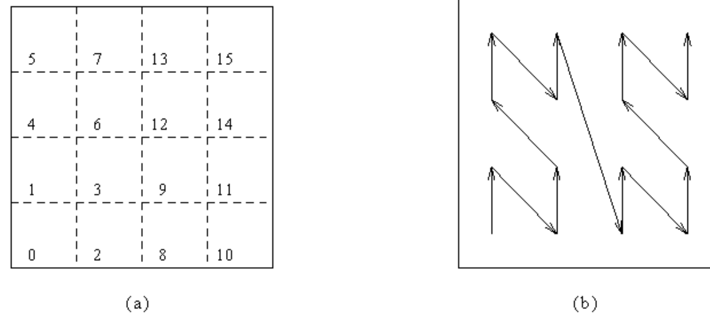


Fig. 6. The bucket-numbering scheme: (a) bucket numbering; (b) N-order peano curve.

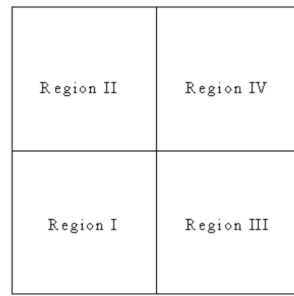
21	23	29	31	53	55	61	63
20	22	28	30	52	54	60	62
17	19	25	27	49	51	57	59
16	18	24	26	48	50	56	58
5	7	13	15	37	39	45	47
4	6	12	14	36	38	44	46
1	3	9	11	33	35	41	43
0	2	8	10	32	34	40	42

Fig. 7. An example of the bucket numbering scheme, $O(l, u) = (12, 26)$.

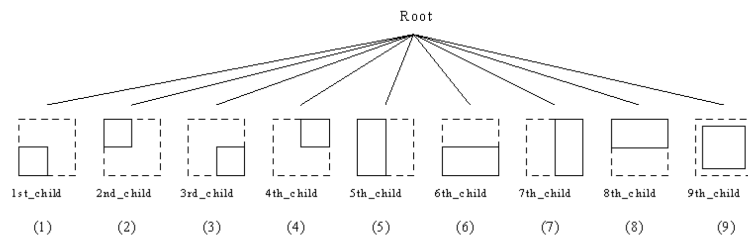
2.2 Data Structure

Generally, tree structures handling multi-dimensional data are constructed with two types of nodes: internal nodes and leaf nodes. In our method, an internal node can have nine, four, three, or two children. Since a leaf has no children, leaves are terminal nodes. Data can only be stored in a leaf, not in an internal node.

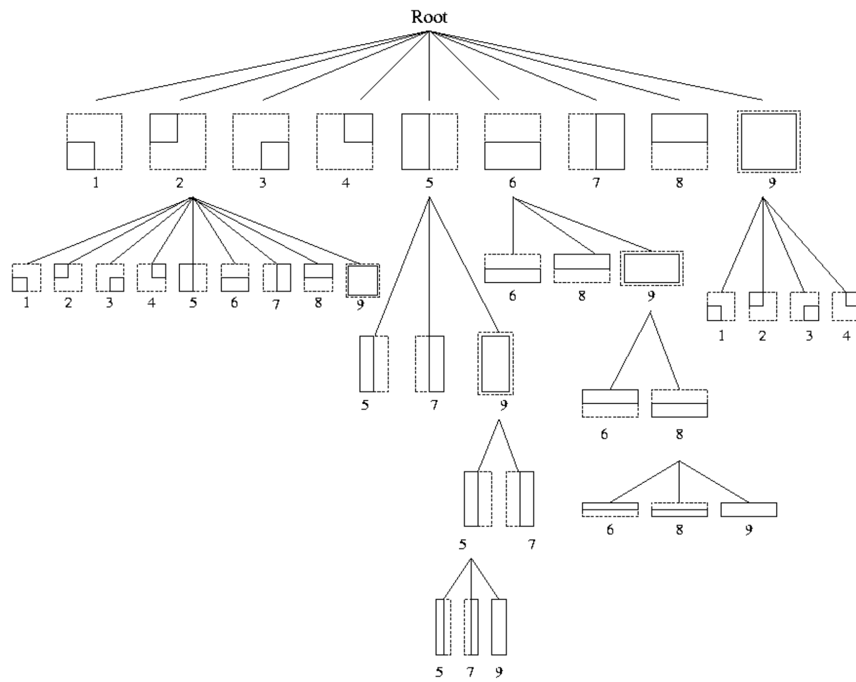
An NA-tree is a structure based on data location and organized by the spatial numbers. First, the whole spatial region is decomposed into four regions. Let *region I* be the bucket numbers between 0 and $\frac{1}{4}(Max_bucket + 1) - 1$, *region II* be the bucket numbers between $\frac{1}{4}(Max_bucket + 1)$ and $\frac{1}{2}(Max_bucket + 1) - 1$, *region III* be the bucket numbers between $\frac{1}{2}(Max_bucket + 1)$ and $\frac{3}{4}(Max_bucket + 1) - 1$, and *region IV* be the bucket numbers between $\frac{3}{4}(Max_bucket + 1)$ and *Max_bucket*, as shown in Fig. 8(a).



(a)



(b)



(c)

Fig. 8. The basic structure of a NA-tree: (a) four regions; (b) nine cases; (c) one of the possible tree structures.

Based on this decomposition, we find that when an object is lying on the space, only nine cases are possible (as shown in Fig. 8(b)). Thus, for an object $O(l, u)$, an index (internal) node p in an NA-tree may have the following nine children:

(1) If both l and $u \in$ region I, O is the first child of node p.
(2) If both l and $u \in$ region II, O is the second child of node p.
(3) If both l and $u \in$ region III, O is the third child of node p.
(4) If both l and $u \in$ region IV, O is the fourth child of node p.
(5) If $l \in$ region I and $u \in$ region II, O is the fifth child of node p.
(6) If $l \in$ region I and $u \in$ region III, O is the sixth child of node p.
(7) If $l \in$ region III and $u \in$ region IV, O is the seventh child of node p.
(8) If $l \in$ region II and $u \in$ region IV, O is the eighth child of node p.
(9) If $l \in$ region I and $u \in$ region IV, O is the ninth child of node p.

Each of the above nine children, the following data structure can be used:

```
struct nine_children
    { int parentid;
      int uid;
      struct nine_children *1st_child;
      struct nine_children *2nd_child;
      struct nine_children *3rd_child;
      struct nine_children *4th_child;
      struct nine_children *5th_child;
      struct nine_children *6th_child;
      struct nine_children *7th_child;
      struct nine_children *8th_child;
      struct nine_children *9th_child;
      struct one_list *chain;
      struct nine_children *parent;}
struct one_list
    {data_object [1..bucket_capacity];
     struct one_list *next_ptr;}
```

In this structure, fields *parentid* and *uid* are used to record the type i of its parent and itself, respectively, $1 \leq i \leq 9$, and field *parent* is a pointer pointing back to its parent's node. However, based on different types of children, some fields may not be used. For example, when it is the 5th child or the 7th child, fields 1*st_child*, 2*nd_child*, 3*rd_child*, 4*th_child*, 6*th_child* and 8*th_child* are not used. When it is the 6th child or the 8th child, fields 1*st_child*, 2*nd_child*, 3*rd_child*, 4*th_child*, 5*th_child* and 7*th_child* are not used. When it is the 9th child, as shown in Fig. 8(c), in addition to using fields *parentid, uid*, and *parent*, depending on the type of its parent, it may use one of the following choices: (1) 1*st_child*, 2*nd_child*, 3*rd_child* and 4*th_child* fields; or (2) 5*th_child* and 7*th_child* fields; or (3) 6*th_child* and 8*th_child* fields. Note that field *chain* is used to handle the special case in which many objects cluster together in the same bucket number (which will be discussed later). Leaf nodes in an NA-tree contain index objects entries of the form (*entry_number*, *data*[1..*bucket_capacity*]), where *entry_number* refers to the num-

ber of objects in this leaf node, *data*[1.. *bucket_capacity*] is an array to store object data, and *bucket_capacity* denotes the maximum number of entries which can be stored in the leaf node. Fig. 9 shows an example of an NA-tree structure. Note that we do not split the spatial space; instead we spatially organize the data objects by some rule according to their spatial number (decided by their locations).

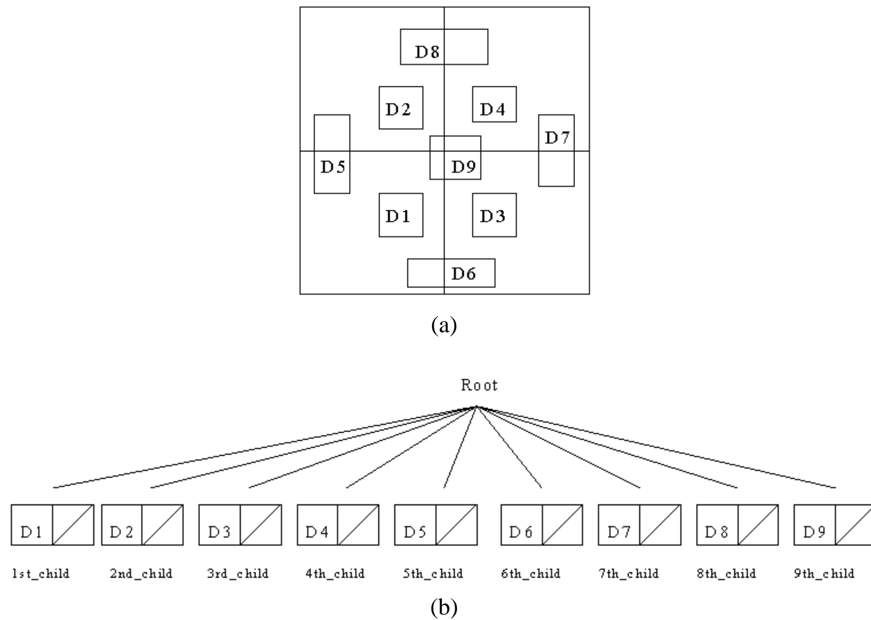


Fig. 9. An example: (a) the data; (b) the corresponding NA-tree structure (bucket_capacity = 2).

2.3 The Insertion Algorithm

This section describes our algorithm for inserting spatial objects into the NA tree. The *Insertion* procedure is shown in Fig. 10. Function *Assign*, procedure *Decision*, and procedure *SplitYN* which are used in the *Insertion* procedure are shown in Figs. 11, 12, and 13, respectively. Basically, inserting a new rectangle in an NA-tree is done by searching the tree according to spatial number and adding the rectangle in the leaf node. Finally, the overflowing node is split and the split may propagate to the child node, if it occurs.

```
procedure Insertion (O(X_l, Y_b, X_r, Y_t));
begin
    l := Assign (X_l, Y_b, b);
    u := Assign (X_r, Y_t, b);
    /* Calculate L(X_l, Y_b)'s and U(X_r, Y_t)'s bucket numbers, (l, u), respectively */
    p := Root;
```

Fig. 10. Procedure *Insertion.*

```
        Decision (l, u, p);
        if p is a leaf node then
        begin
              Add O to node p;
              If node p overflows then SplitYN(O(X_l, Y_b, X_r, Y_t), p);
        end
        else
        begin
              if p^ uid ∈ {1, 2, 3, 4} then Insert1234(O(X_l, Y_b, X_r, Y_t), l, u, p)
              else if p^ uid ∈ {5, 7} then Insert57(O(X_l, Y_b, X_r, Y_t), l, u, p)
              else if p^ uid ∈ {6, 8} then Insert68(O(X_l, Y_b, X_r, Y_t), l, u, p)
              else Insert9(O(X_l, Y_b, X_r, Y_t), l, u, p);
        end;
end;
```

Fig. 10. (Cont'd) Procedure *Insertion.*

```
function Assign (x0, x1, b): integer;
/* compute the bucket number */
/* l0 and h0 are the lower and upper bounds of coordinates along the x-axis. */
/* l1 and h1 are the lower and upper bounds of coordinates along the y-axis. */
/* b is the need number of binary bits to record the bucket number. */
begin
    w = " "; /* null string */
    for k :=1 to b
    begin
        i := k mod 2;
        if (xi < (li + hi)/2) then
        begin
              concatenate "0" to w;
              hi := (li + hi)/2;
        end
        else
        begin
              concatenate "1" to w;
              li := (li + hi)/2;
        end;
              end;
    change binary number (w) to decimal;
    return(w);
end;
```

Fig. 11. Function *Assign.*

```
procedure Decision (l, u, p);
begin
        if (l ∈ I) and (u ∈ I) then p := p^ lst_child
        else if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child
        else if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child
        else if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child
        else if (l ∈ I) and (u ∈ II) then p := p^ 5th_child
        else if (l ∈ I) and (u ∈ III) then p := p^ 6th_child
        else if (l ∈ III) and (u ∈ IV) then p := p^ 7th_child
        else if (l ∈ II) and (u ∈ IV) then p := p^ 8th_child
        else if (l ∈ I) and (u ∈ IV) then p := p^ 9th_child;
end;
```

Fig. 12. Procedure *Decision.*

```
procedure Split YN(O(X_l, Y_b, X_r, Y_t), p );
begin
        if (Split_Region (p) = False) then Add O to p^ chain
            else Split(p);
end;
```

Fig. 13. Procedure *SplitYN.*

In the *Insertion* procedure, the first step in inserting an object, $O(L, U)$, is to compute its spatial number, i.e., the two bucket numbers of L and U. The function *Assign* is called with the coordinates of the point (x_0, x_1) and the number of bits b, where b is the number of bits in the binary form of bucket number. The function *Assign* returns the bucket numbers 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 Fig. 11) is used to compute the DZ expression and return a decimal bucket number. In function *Assign*, l_0 and h_0 denote the lower and upper bounds of coordinates along the x-axis; while l_1 and h_1 denote the lower and upper bounds of coordinates along the y-axis. The constant b (i.e., the required number of binary bits to record the bucket number) in function *Assign* depends on the expected number of data objects. The relationship between the number of objects and the required number of bits to record the bucket numbers is shown in Table 1. Note that given the total number of buckets = 2^b, we have the level of tree from 0 to L, where $L = \lfloor \log_4 2^b \rfloor$. Let NL_i be the maximum number of leaf nodes occurring in level i. We have $NL_0 = 1$, $NL_1 = 9$, $NL_2 = 4 * NL_1 + 4 * 3 + 1 * 4 = 52$, $NL_3 = 4 * NL_2 + 4 * (3 + 3 + 2) + 1 * 4 * NL_1 = 276$, and $NL_4 = 4 * NL_3 + 4 * ((3 + 3 + 2) + (3 + 3 + 2) + (3 + 3)) + 1 * 4 * NL_2 = 1400$.

Next, according to the spatial number, we search the tree and find which leaf node this object belongs to. For the first level, procedure *Decision* is called to decide which of the nine children the spatial number belongs to. In procedure *Decision*, node p will also be updated to go down the tree. After that, if p is a leaf node, this object is inserted into this leaf node. Then, this leaf node is checked to see whether it overflows. If this leaf node overflows, then procedure *SplitYN* is executed.

Table 1. The relationship between the number of objects (*N*) and *b*.

B	*N*	*L*	*b*
$0 \leq B < 256$	$N \leq 1400$	4	8
$0 \leq B < 1024$	$N \leq 6116$	5	10
$0 \leq B < 8192$	$N \leq 26520$	6	13

N: the number of objects
B: the number of buckets
L: the number of levels
b: the number of bits

When an overflow occurs in a leaf node, the *SplitYN* procedure calls function *Split_Region* (shown in Fig. 14) to detect whether this region should be split into more children. In the case that region *p* or some part of region *p* covers only one bucket number, function *Split_Region* returns False which indicates that we should add the object to the linked chain. On the other hand, if function *Split_Region* returns True, procedure *Split* (shown in Fig. 15) will be executed. Procedure *Split* will create a different number of children, depending on different cases of *uid* and *parentid*. In particular, when *uid* = 9, the centroid of each object in *p* is used to decide the new spatial number of the object. Then, each object in *p* is re-inserted according to its new corresponding spatial number. The flowchart of the checking conditions in procedure *Split* is shown in Fig. 16.

```
function Split_Region(p): boolean;
begin
      Split_Region := True;
      if (p^ uid ∈ {1, 2, 3, 4}) and (region p covers only one bucket number) then
            Split_Region := False
      else if (p^ uid ∈ {5, 7}) and (the width of p covers only one bucket number) then
            Split_Region := False
      else if (p^ uid ∈ {6, 8}) and (the height of p covers only one bucket number) then
            Split_Region := False
      else /* uid = 9 */
      begin
            if (p^ parentid ∈ {1, 2, 3, 4}) and (the region of p covers only one bucket number)
                then Split_Region := False
            else if (p^ parentid ∈ {5, 7}) and (the region of p covers only one bucket number)
                then Split_Region := False
            else if (p^ parentid ∈ {6, 8}) and (the region of p covers only one bucket number)
                then Split_Region := False;
      end;
end;
```

Fig. 14. Function *Split_Region*.

```
procedure Split(p);
begin
    if p^ uid ∈ {1, 2, 3, 4} then
        Create all 9 children of p
    else if p^ uid ∈ {5, 7} then
        Create 5th_child, 7th_child, and 9th_child of p
    else if p^ uid ∈ {6, 8} then
        Create 6th_chid, 8th_child, and 9th_child of p
    else /* uid = 9 */
    begin
        if p^ parentid ∈ {1, 2, 3, 4} then
            Create 1st_child, 2nd_child, 3rd_child, and 4th_child of p
        else if p^ parentid ∈ {5, 7} then
            Create 5th_child, 7th_child of p
        else if p^ parentid ∈ {6, 8} then
            Create 6th_child, 8th_child of p;
        Calculate the centroid of each object in p;
        Replace the spatial number of each object in p
            with the spatial number of its centroid;
    end;
    Re-Insert each object in p according to its new corresponding spatial number;
    /* Call Insertion again */
end;
```

Fig. 15. Procedure *Split*.

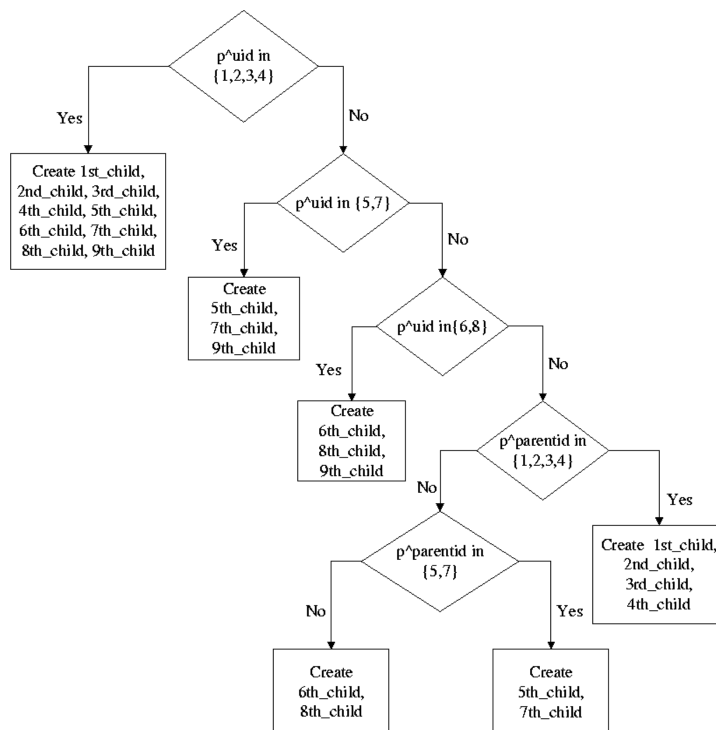


Fig. 16. Flowchart of the checking conditions in procedure *Split*.

Note that in procedure *Split*, when a split occurs in the 9th child, two possible solutions can be applied:

1. Use a chain of buckets for all rectangles overlapping, such a cross point, which is the strategy proposed in R-files [18].
2. Use the location of the centroid of the rectangle to decide which region it should belong to, which is first applied in our strategy. However, for the worst case in which there are many rectangles with the centroids located in the same bucket number, we still have to use strategy 1, a chain of buckets, to handle it.

On the other hand, if *p* is not a leaf node, procedures *Insert*1234 (Fig. 17), *Insert*57 (Fig. 18), *Insert*68 (Fig. 19), or *Insert*9 (Fig. 20) will be called, depending on its *uid* ∈ {1, 2, 3, 4}, {5, 7}, {6, 8} or {9}, respectively. In each of these procedures, the tree is traversed downward through different types of children decided by the spatial number as shown in Fig. 8(c).

```
procedure Insert1234(O(Xl, Yb, Xr, Yt), l, u, p);
begin
   Decision(l, u, p);
   if p is a leaf node then
   begin
        Add O to node p;
        if node p overflows then SplitYN(O(Xl, Yb, Xr, Yt), p);
   end
   else
   begin
        if p^ uid ∈ {1, 2, 3, 4} then Insert1234(O(Xl, Yb, Xr, Yt), l, u, p)
        else if p^ uid ∈ {5, 7} then Insert57(O(Xl, Yb, Xr, Yt), l, u, p)
        else if p^ uid ∈ {6, 8} then Insert68(O(Xl, Yb, Xr, Yt), l, u, p )
        else Insert9(O(Xl, Yb, Xr, Yt), l, u, p);
   end;
end;
```

Fig. 17. Procedure *Insert*1234.

```
procedure Insert57(O(Xl, Yb, Xr, Yt), l, u, p);
begin
      if (l ∈ the left part of p's region) and (u ∈ the left part of p's region) then
            p := p^ 5th_child
      else if (l ∈ the right part of p's region) and (u ∈ the right part of p's region) then
            p := p^ 7th_child
      else p := p^ 9th_child;
      if p is a leaf node then
      begin
```

Fig. 18. Procedure *Insert*57.

```
        Add O to node p;
        if node p overflows then SplitYN(O(X_l, Y_b, X_r, Y_t), p);
    end
    else
    begin
        if p^ uid ∈ {5, 7} then Insert57(O(X_l, Y_b, X_r, Y_t), l, u, p)
        else if p^ uid = 9 then Insert9(O(X_l, Y_b, X_r, Y_t), l, u, p);
    end;
end;
```

Fig. 18. (Cont'd) Procedure *Insert*57.

```
procedure Insert68(O(X_l, Y_b, X_r, Y_t), l, u, p);
begin
    if (l ∈ the bottom part of p's region) and (u ∈ the bottom part of p's region) then
        p := p^ 6th_child
    else if (l ∈ the top part of p's region) and (u ∈ the top part of p's region) then
        p := p^ 8th_child
    else p := p^ 9th_child;
    if p is a leaf node then
    begin
        Add O to node p;
        if node p overflows then SplitYN(O(X_l, Y_b, X_r, Y_t), p);
    end
    else
    begin
        if p^ uid ∈ {6, 8} then Insert68(O(X_l, Y_b, X_r, Y_t), l, u, p)
        else if p^ uid = 9 then Insert9(O(X_l, Y_b, X_r, Y_t), l, u, p);
    end;
end;
```

Fig. 19. Procedure *Insert*68.

```
procedure Insert9(O(X_l, Y_b, X_r, Y_t), l, u, p);
begin
    l := Assign((X_l + X_r)/2, (Y_b + Y_t)/2, b);
    u := l;
    if p^ parentid ∈ {1, 2, 3, 4} then
    begin
        if (l ∈ I) and (u ∈ I) then p := p^ lst_child
        else if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child
        else if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child
        else if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child;
        if p is a leaf node then
        begin
```

Fig. 20. Procedure *Insert*9.

```
            Add O to node p;
            if node p overflows then SplitYN(O(Xl, Yb, Xr, Yt), p);
        end
        else Insert1234(O(Xl, Yb, Xr, Yt), l, u, p);
    end
    else if p^ uid ∈ {5, 7} then
    begin
        if (l ∈ the left part of p's region) and (u ∈ the left part of p's region) then
            p := p^ 5th_child
        else if (l ∈ the right part of p's region) and (u ∈ the right part of p's region) then
            p := p^ 7th_child;
        if p is a leaf node then
        begin
            Add O to node p;
            if node p overflows then SplitYN(O(Xl, Yb, Xr, Yt), p);
        end
        else Insert57(O(Xl, Yb, Xr, Yt), l, u, p);
    end
    else if p^ uid ∈ {6, 8} then
    begin
        if (l ∈ the bottom part of p's region) and (u ∈ the bottom part of p's region) then
            p := p^ 6th_child
        else if (l ∈ the top part of p's region) and (u ∈ the top part of p's region) then
            p := p^ 8th_child;
        if p is a leaf node then
        begin
            Add O to node p;
            if node p overflows then SplitYN(O(Xl, Yb, Xr, Yt), p);
        end
        else Insert68(O(Xl, Yb, Xr, Yt), l, u, p);
    end;
end;
```

Fig. 20. (Cont'd) Procedure *Insert*9.

2.4 The Deletion Algorithm

Fig. 21 shows the *Deletion* procedure. Deletion of a rectangle from an NA-tree is done by first locating the rectangle that must be deleted (by calling function *Exact_match_query* as shown in Fig. 22), and then removing it from the leaf node. Next, this leaf node is checked whether or not it is *empty*, where *empty* means that there are no other objects in this leaf node. When an empty leaf node occurs, it may be merged with other sibling leaves by calling procedure *Merge* (shown in Fig. 23).

```
procedure Deletion (O(X_l, Y_b, X_r, Y_t));
begin
    if Exact_match_query (O(X_l, Y_b, X_r, Y_t)) = True then
        delete O from p
    else show an error message;
    if p is empty then Merge(p);
end;
```

Fig. 21. Procedure *Deletion*.

```
function Exact_match_query (O(X_l, Y_b, X_r, Y_t)): boolean;
begin
    l := Assign(O(X_l, Y_b, b);
    u := Assign(O(X_r, Y_t, b);
    p := Root;
    Decision(l, u, p);
    if p^ uid ∈ {1, 2, 3, 4} then
        Exact_match_query := Search1234(O(X_l, Y_b, X_r, Y_t), l, u, p)
    else if p^ uid ∈ {5, 7} then
        Exact_match_query := Search57(O(X_l, Y_b, X_r, Y_t), l, u, p)
    else if p^ uid ∈ {6, 8} then
        Exact_match_query := Search68(O(X_l, Y_b, X_r, Y_t), l, u, p)
    else Exact_match_query := Search9(O(X_l, Y_b, X_r, Y_t), l, u, p);
end;
```

Fig. 22. Function *Exact_match_query*.

```
procedure Merge(p);
begin
    q := p^ parent;
    release p;
    while q is not a root
    begin
        calculate entries of q;
        /*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;
            release all of q's children;
            n^ parent := q^ parent ;
            n^ uid := q^ uid ;
            n^ parentid := q^ parentid ;
        end;
      q := q^ parent;
    end;
    if (entries ≤ bucket_capacity) then /* q = the root */
```

Fig. 23. Procedure *Merge*.

```
      begin
           create a new leaf node n;
           move objects from q's children to n;
           release all of q's children;
           n^ parent := null;
           n^ uid := 1;
           n^ parentid := 0;
      end;
end;
```

Fig. 23. (Cont'd) Procedure *Merge*.

```
function Search1234(O(X_l, Y_b, X_r, Y_t), l, u, p): boolean;
begin
      if p is a leaf node then
          if O is not in p then
          begin
              show an error message;
              Search1234 := False;
          end
          else
          begin
            output O from p;
            Search1234 := True;
          end
      else
      begin
          Decision(l, u, p);
          if p^ uid ∈ {1, 2, 3, 4} then
              Search 1234 := Search1234(O(X_l, Y_b, X_r, Y_t), l, u, p)
          else if p^ uid ∈ {5, 7} then
              Search 1234 := Search57(O(X_l, Y_b, X_r, Y_t), l, u, p)
          else if p^ uid ∈ {6, 8} then
              Search 1234 := Search68(O(X_l, Y_b, X_r, Y_t), l, u, p)
          else Search 1234 := Search9(O(X_l, Y_b, X_r, Y_t), l, u, p);
      end;
end;
```

Fig. 24. Function *Serach*1234.

In function *Exact_match_query* (shown in Fig. 22), after calculating the patial number, the tree is searched from the root. Depending on the value of *uid* ∈ {1, 2, 3, 4}, {5, 7}, {6, 8} or {9}, function *Search*1234 (shown in Fig. 24), function *Search*57 (shown in Fig. 25), function *Search*68 (shown in Fig. 26), or function *Search*9 (shown in Fig. 27), is called, respectively. In each of those functions, the tree is traversed downward through different types of children decided by the spatial number as shown in Fig. 8(c).

In procedure *Merge* as shown in Fig. 23, node *q* is merged with its children if the number of objects in all children of *q* is less than *bucket_capacity*. This process is repeated upward to the root. If the same case occurs in the root, we have to do the same thing, except that the fields *parent*, *uid* and *parentid* must be reset carefully.

```
function Search57(O(Xl, Yb, Xr, Yt), l, u, p): boolean;
begin
    if p is a leaf node then
      if O is not in p then
      begin
        show an error message;
        Search57 := False;
      end
      else
      begin
        output O from p;
        Search57 := True;
      end
    else
    begin
      if (l ∈ the left part of p's region) and (u ∈ the left part of p's region) then
        p: = p^ 5th_child
      else if (l ∈ the right part of p's region) and (u ∈ the right part of p's region) then
        p: = p^ 7th_child
      else p: = p^ 9th_child;
      if p^ uid ∈ {5, 7} then
        Search57 := Serach57(O(Xl, Yb, Xr, Yt), l, u, p)
      else Serach57 := Search9(O(Xl, Yb, Xr, Yt), l, u, p);
    end;
end;
```

Fig. 25. Function *Search*57.

```
function Search68(O(Xl, Yb, Xr, Yt), l, u, p): boolean;
begin
    if p is a leaf node then
      if O is not in p then
      begin
        show an error message;
        Search68 := False;
      end
      else
```

Fig. 26. Function *Search*68.

```
        begin
            output O from p;
            Search68 := True;
        end
    else
    begin
        if (l ∈ the bottom part of p's region) and (u ∈ the bottom part of p's region) then
            p: = p^ 6th_child
        else if (l ∈ the top part of p's region) and (u ∈ the top part of p's region) then
            p: = p^ 8th_child
        else p: = p^ 9th_child;
        if p^ uid ∈ {6, 8} then
            Search68 := Serach68(O(X_l, Y_b, X_r, Y_t), l, u, p)
        else Serach68 := Search9(O(X_l, Y_b, X_r, Y_t), l, u, p);
    end;
end;
```

Fig. 26. (Cont'd) Function *Search*68.

```
function Search9(O(X_l, Y_b, X_r, Y_t), l, u, p): boolean;
begin
    if p is a leaf node then
        if O is not in p then
        begin
            show an error message;
            Search9 := False;
        end
        else
        begin
            output O from p;
            Search9 := True;
        end
    else
    begin
        l := Assign((X_l + X_r)/2, (Y_b + Y_t)/2, b);
        u := l;
        if p^ parentid ∈ {1, 2, 3, 4} then
        begin
            if (l ∈ I) and (u ∈ I) then p := p^ 1st_child
            else if (l ∈ II) and (u ∈ II) then p := p^ 2nd_child
            else if (l ∈ III) and (u ∈ III) then p := p^ 3rd_child
            else if (l ∈ IV) and (u ∈ IV) then p := p^ 4th_child;
            Search9 := Search1234(O(X_l, Y_b, X_r, Y_t), l, u, p);
        end
```

Fig. 27. Function *Search*9.

```
        else if p^ parentid ∈ {5, 7} then
        begin
              if (l ∈ the left part of p's region) and (u ∈ the left part of p's region) then
                 p: = p^ 5th_child
              else if (l ∈ the right part of p's region) and (u ∈ the right part of p's region) then
                 p: = p^ 7th_child;
              Search9 := Serach57(O(X_l, Y_b, X_r, Y_t), l, u, p);
        end
        else if p^ parentid ∈ {6, 8} then
        begin
              if (l ∈ the bottom part of p's region) and (u ∈ the bottom part of p's region) then
                 p: = p^ 6th_child
              else if (l ∈ the top part of p's region) and (u ∈ the top part of p's region) then
                 p: = p^ 8th_child;
              Search9 := Serach68(O(X_l, Y_b, X_r, Y_t), l, u, p);
        end;
     end;
end;
```

Fig. 27. (Cont'd) Function *Search*9.

2.5 Exact Match and Range Queries

The algorithm to process the exact match query is shown in Fig. 22. For the algorithm to process the range query, as shown in Fig. 28, a recursive approach is applied, since it is possible that more than one internal node will be covered by this search range R(l, u).

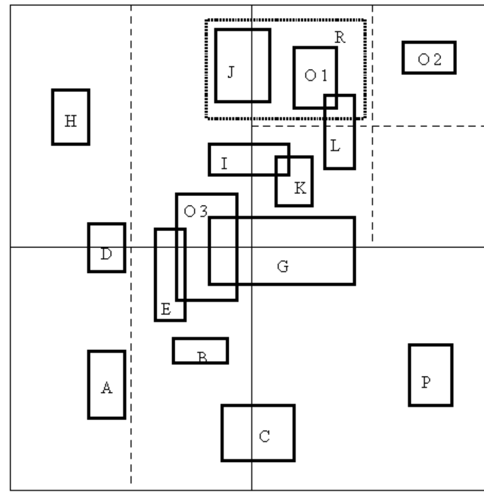
```
procedure Range_query(p, R(l, u));
begin
      if p is a leaf then
         for each object O ∈ p do
         begin
               O_l = Assign(X_l, Y_b, b);
               O_u = Assign(X_r, Y_t, b);
               if (l ≤ O_l) and (O_u ≤ u) then
                  output O;
         end
      else
         begin
            if (l ∈ I) and (u ∈ I) then Range_query(p^ 1st_child, R(l, u));
            if (l ∈ II) and (u ∈ II) then Range_query(p^ 2nd_child, R(l, u));
            if (l ∈ III) and (u ∈ III) then Range_query(p^ 3rd_child, R(l, u));
            if (l ∈ IV) and (u ∈ IV) then Range_query(p^ 4th_child, R(l, u));
            if (l ∈ I) and (u ∈ II) then
```

Fig. 28. Procedure *Range_query*.

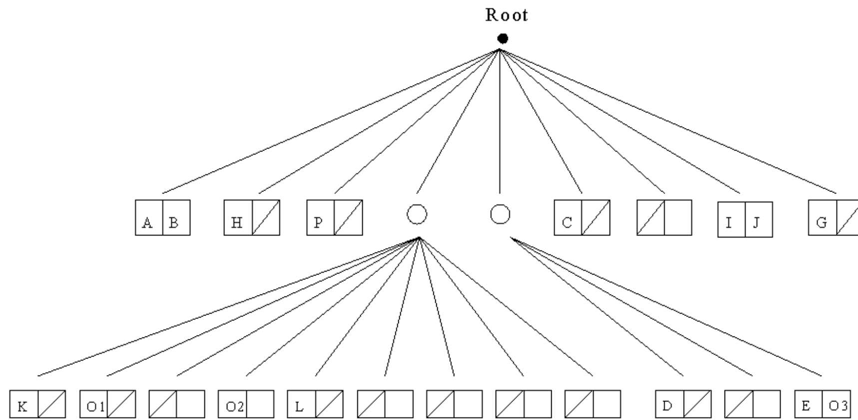
```
            begin
               Range_query(p^ 5th_child, R(l, u));
               Range_query(p^ 1st_child, R(l, u));
               Range_query(p^ 2nd_child, R(l, u));
            end;
         if (l ∈ I) and (u ∈ III) then
            begin
               Range_query(p^ 6th_child, R(l, u));
               Range_query(p^ 1st_child, R(l, u));
               Range_query(p^ 3rd_child, R(l, u));
            end;
         if (l ∈ III) and (u ∈ IV) then
            begin
               Range_query(p^ 7th_child, R(l, u));
               Range_query(p^ 3rd_child, R(l, u));
               Range_query(p^ 4th_child, R(l, u));
            end;
         if (l ∈ II) and (u ∈ IV) then
            begin
               Range_query(p^ 8th_child, R(l, u));
               Range_query(p^ 2nd_child, R(l, u));
               Range_query(p^ 4th_child, R(l, u));
            end;
         if (l ∈ I) and (u ∈ IV) then
            begin
               Range_query(p^ 1st_child, R(l, u));
               Range_query(p^ 2nd_child, R(l, u));
               Range_query(p^ 3rd_child, R(l, u));
               Range_query(p^ 4th_child, R(l, u));
               Range_query(p^ 5th_child, R(l, u));
               Range_query(p^ 6th_child, R(l, u));
               Range_query(p^ 7th_child, R(l, u));
               Range_query(p^ 8th_child, R(l, u));
               Range_query(p^ 9th_child, R(l, u));
            end;
   end;
end;
```

Fig. 28. (Cont'd) Procedure *Range_query*.

Now, we use one example to describe how the range query is processed by using the NA-tree. Consider the state in Fig. 29. We want to find all data objects which are included in the search range *R*. Before procedure *Range_query* is called, we have to calculate the spatial number of the search region by using the *Assign* function, where *Max_bucket* = 63. The spatial number of the search range *R* is (30, 55). Then, the procedure *Range_query* is called. The first recursive call will result in the search of the



(a)



(b)

Fig. 29. An example for the range query.

2*nd_child*, 4*th_child*, and 8*th_child* of the *Root*. Since the 2*nd_child* and 8*th_child* of the *Root* are leaf nodes, rectangle *J* will be found first. When the 4*th_child* of the *Root* is searched, it will call procedure *Range_query* again since it is not a leaf node. Let *p* be the 4*th_child* of the *Root*, the search range in *p* is only covered by 2*nd_child* of *p* at this time. Then, we search down the 2*nd_child* of *p*. Finally, rectangle O_1 is found since the 2*nd_child* of *p* is a leaf node.

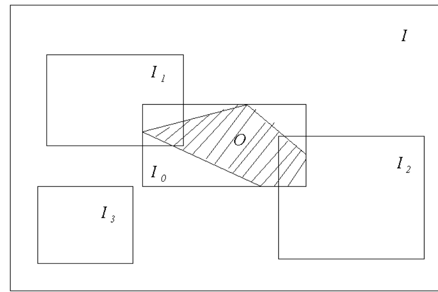


Fig. 30. Case 1 in an R+-tree: the data interval I_0 overlaps the sibling intervals I_1 and I_2.

2.6 Difficult Cases in R+-Trees

The R+-tree allows fast computation of search operators. However, the insertion and deletion of data objects may be much more complicated [14]. First, the insertion of an object O or its data interval I_0 may require the enlargement of *several* sibling intervals (intervals corresponding to sibling nodes). This is especially, but not exclusively, the case if I_0 overlaps several sibling intervals. In Fig. 30, I_0 has to be inserted into both corresponding subtrees. I_1 and I_2 have to be enlarged 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 already exists), and then insert the data interval I_0 into the leaf node (shown in Fig. 31).

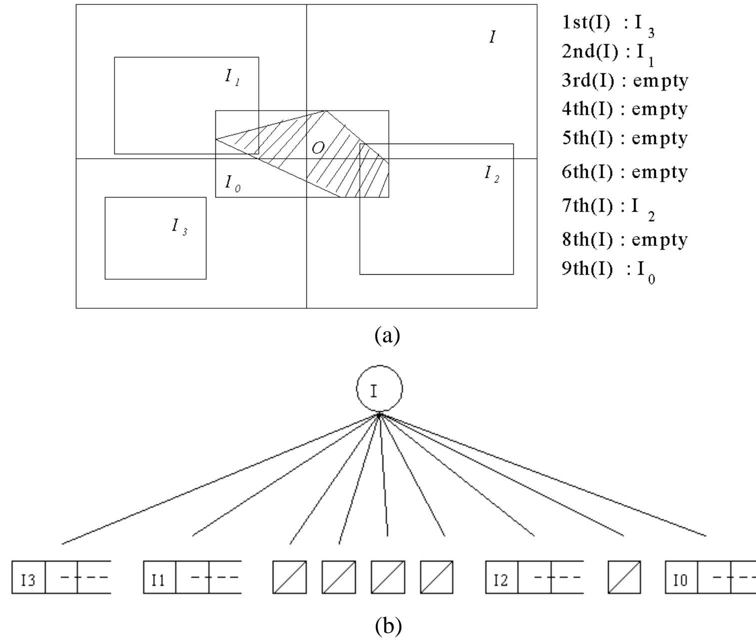


Fig. 31. Case 1 in an NA-tree.

Second, there are situations where the enlargement step inevitably leads to overlaps (shown in Fig. 32). 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 [14]. For this case, the NA-tree approach can create a new leaf node (or perhaps this leaf node already exists), and then insert the data interval I_0 into the leaf node (shown in Fig. 33).

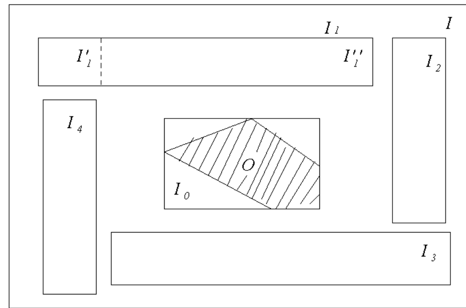
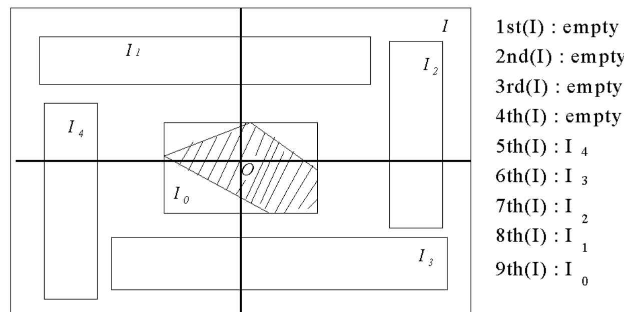
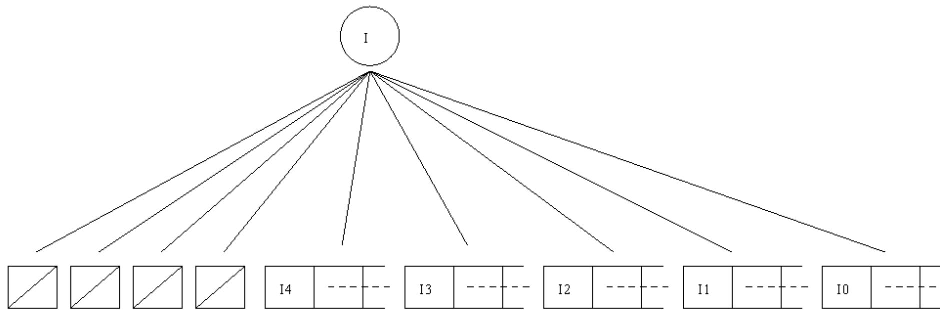


Fig. 32. Case 2 in an R^+-tree: It is not possible to enlarge the sibling intervals $I_1...I_4$ without creating overlaps.



(a)



(b)

Fig. 33. Case 2 in an NA-tree.

3. PERFORMANCE

In this section, we compare the performance of R-trees [16], R^+-trees [38], R^*-trees [4], and NA-trees. Our experiments were performed on a Pentium III 550 MHz, 128 MB RAM, and running Windows 2000.

3.1 Simulation Study

Here we define the search cost, storage utilization, insertion cost, deletion cost, storge cost and height of a tree.

Definition 1. Search cost (C) of a tree T: the number of nodes visited.

Definition 2. Storage utilization (S) of a tree T:

$$S = \frac{\text{The number of data stored in } T}{(\text{The number of leaves in } T) \times P} \times 100\%.$$

Definition 3. Insert cost: the number of internal nodes visited.

Definition 4. Deletion cost: the same as that used for insertion.

Definition 5. Storage cost: the number of internal and external nodes.

Definition 6. Level of a node and height of a tree: Let v be a node and *Root* be the top node of a tree. We represent a child of a node, v, as *v.child*. Then $d(v)$ is the level of node v. The level is defined as

(1) $d(Root) = 0$;
(2) $d(v.child) = d(v) + 1$.

The height of a tree is equal to the maximum number of levels.

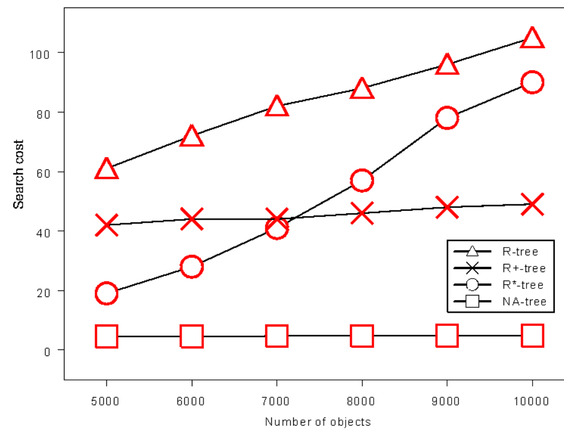
There are two major parameters that characterize such a geometric database; the number N of data objects (the database size) and their average size, *avg_size*, measured in percent of the size of the data space, i.e.,

$$avg_size = \frac{\sum_{i=1}^{N} area_i / N}{\text{the whole data space}} \times 100\%.$$

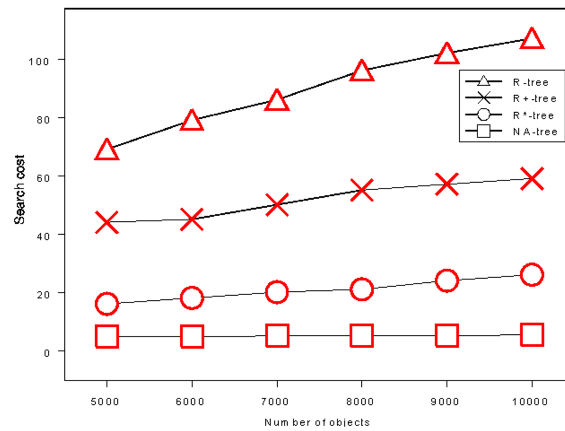
Given that the data space is 1000 * 1000, we took 100 averages of 12 different databases containing 5000, 6000, 7000, 8000, 9000, and 10000 rectangles with average sizes 0.0025%, and 0.0001%. The data objects are uniformly distributed (without overlap) on the whole data space [2, 19, 21, 25, 30, 40-42]. (Note that when overlap occurs between objects, it is possible that R^+-trees will not work [13].) *Bucket_capacity* was assigned to be 10, where the *bucket_capacity* is the maximum number of objects containable in a leaf node. The number of binary bits, b, was assigned to be 13, where b is used to record the

bucket number. (Note that since in this simulation, the maximum number of objects under consideration is 10000, we let b = 13 as explained in Table 1.)

Now, we show some typical results of the search performance of R-trees, R^+-trees, R^*-trees, and NA-trees. Note that to compute the average search cost, for each spatial data file, we create 100 random rectangles to do exact match queries, and then calculate the average search cost. Comparisons in terms of *search cost*, *tree height*, *storage cost*, and *storage utilization* based on the uniform distribution are shown in Figs. 34-37, respectively. In this case, we observe that our NA-tree has the lowest search cost at the expense of a little height of the tree, high storage cost, and low storage utilization. Since based on the current technology, the capacity of secondary storage media (like hard disks or optical disks) is huge and the price is not so expensive, we consider that the performance measure in terms of search cost (which affects retrieval time) may be more important than other performance measures in terms (like storage cost or storage utilization).

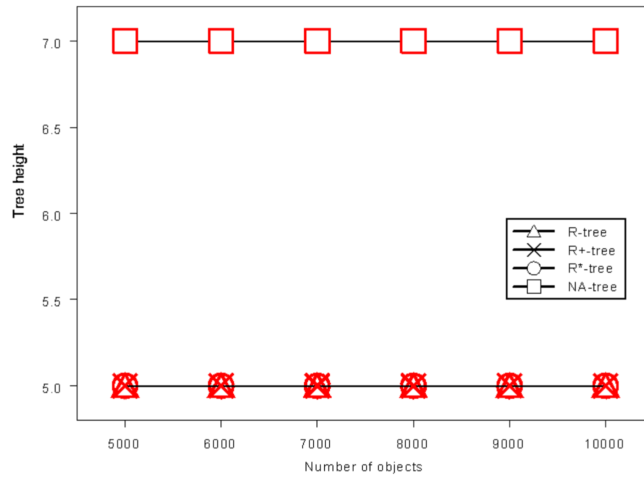


(a)

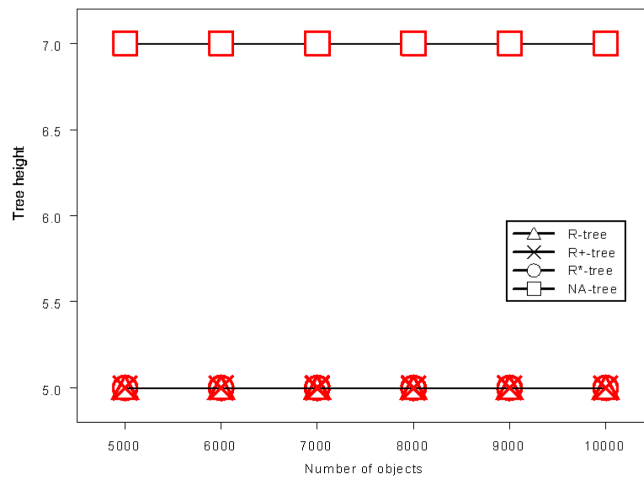


(b)

Fig. 34. A comparison of search cost: (a) *avg_size* = 0.0025%; (b) *avg_size* = 0.0001%.



(a)



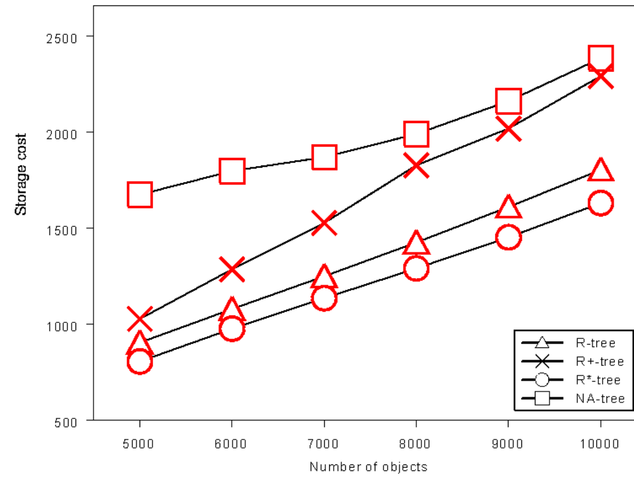
(b)

Fig. 35. A comparison of the height of a tree: (a) *avg_size* = 0.0025%; (b) *avg_size* = 0.0001%.

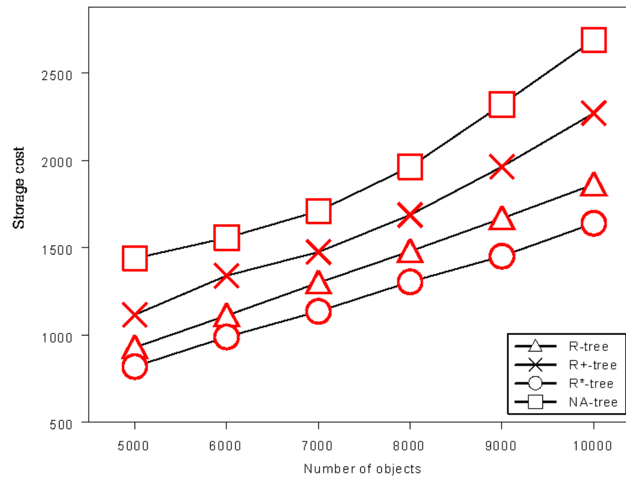
A comparison of storage utilization is summarized in Table 2. Obviously, NA trees decrease the search cost at the expense of decreasing storage utilization.

Table 2. A comparison of storage utilization.

Tree structure	R-tree	R^+-tree	R^*-tree	NA-tree
Storage utilization (%)	60±5	55±5	70±5	45±5



(a)

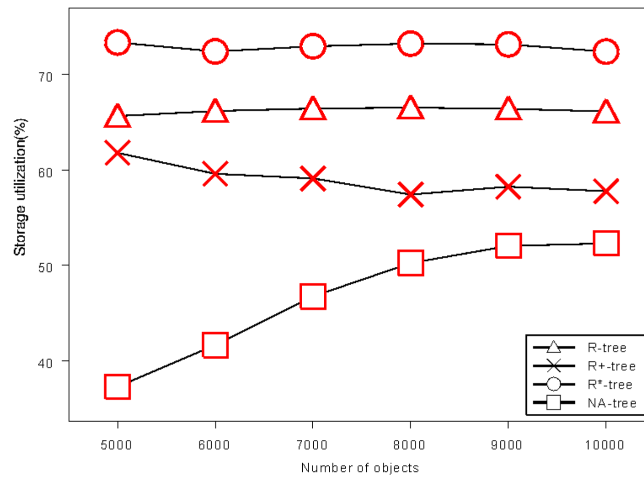


(b)

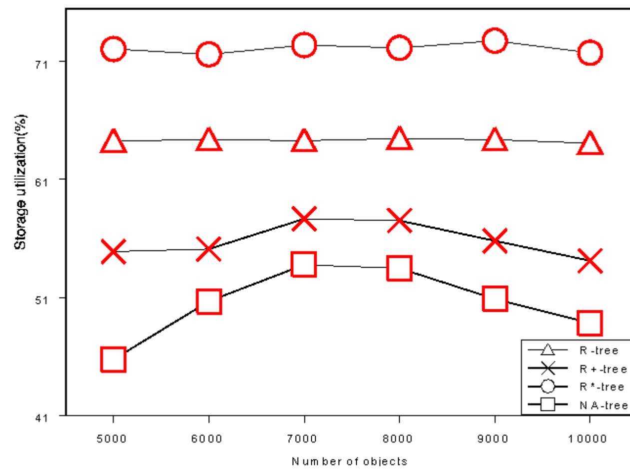
Fig. 36. A comparison of the storage cost: (a) *avg_size* = 0.0025%; (b) *avg_size* = 0.0001%.

For the insertion cost, consider average costs of inserting 5000, 6000, 7000, 8000, 9000, and 10000 rectangles based on a uniform distribution. From the results shown in Fig. 38, we observe that our NA-tree has the lowest insertion cost among these four strategies.

For the deletion cost, again consider the cases of average costs of deleting 100 rectangles in 5000, 6000, 7000, 8000, 9000, and 10000 rectangles based on a uniform distribution. From the results shown in Fig. 39, we observe that our NA-tree has the lowest deletion cost among these four strategies.



(a)

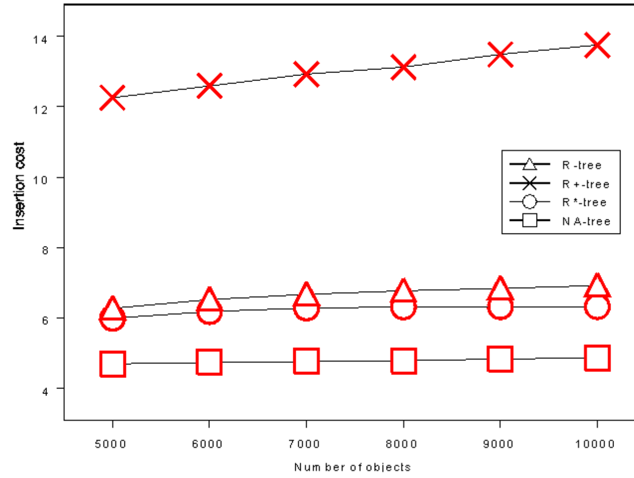


(b)

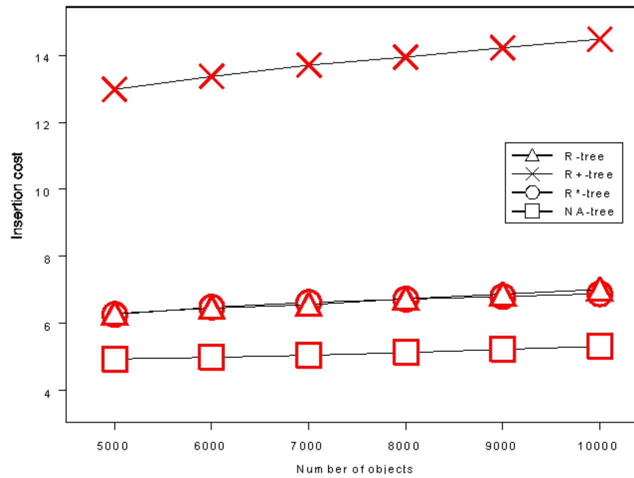
Fig. 37. A comparison of the storage utilization (%): (a) avg_size = 0.0025%; (b) avg_size = 0.0001%.

3.2 Comparisons

The main difference among these data structures lies in the method of spatial decomposition. For example, in R-trees a minimum bounding rectangle is split to minimize the areas. The main difference between R-trees (or R^*-trees) and R^+-trees is that MBRs can overlap each other in R-trees (or R^*-trees), but not in R^+-trees. Although Guttman's R-tree algorithms tried only to minimize the area covered by the bucket regions, the R^*-tree algorithms also take other objectives into account, for example, region perimeters. While our NA-tree does not actually split the spatial space, it just organizes the data objects according to their spatial numbers.



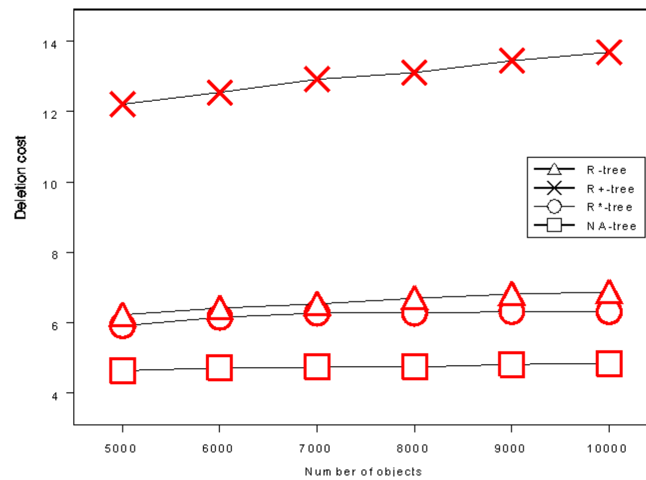
(a)



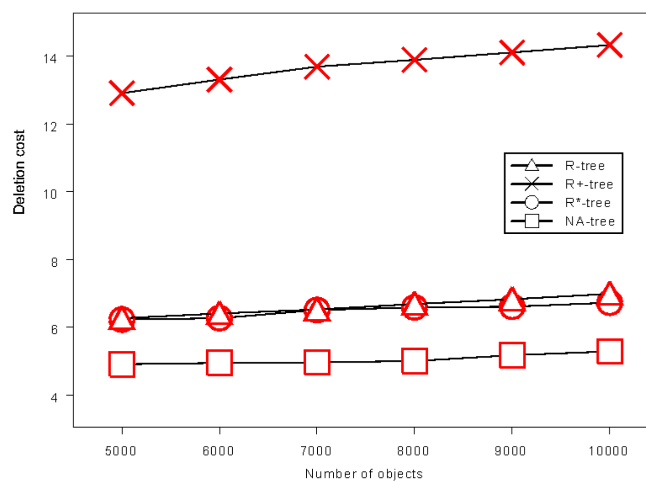
(b)

Fig. 38. A comparison of insertion cost: (a) *avg_size* = 0.0025%; (b) *avg_size* = 0.0001%.

R-trees, R^+-trees and R^*-trees are multi-way balanced trees for very large databases, but heavy-loaded in the data insertion and deletion processes [29]. Although NA-trees are not guaranteed to always be multi-way balanced trees, the insertion and deletion algorithms are easy to implement. Moreover, the search cost for NA-trees is the lowest among these four strategies. The main reason is that a search in an NA-tree is accurately guided by the spatial number (i.e., the pair of bucket numbers), while a search in R-tree-based strategies may try several possible regions covering the object of interest. The above features of R-trees, R^+-trees, R^*-trees and NA-trees are summarized in Table 3.



(a)



(b)

Fig. 39. A comparison of deletion cost: (a) *avg_size* = 0.0025%; (b) *avg_size* = 0.0001%.

4. CONCLUSIONS

A retrieval query on a geometric database typically requires the fast execution of a geometric search operation such as an exact match or a range search. In order to facilitate such search operations on a large geometric database, the use of suitable index structures is a practical necessity. Indexes should be dynamic with respect to updates of the database, i.e., it should be possible to perform insertions and deletions without having to completely reorganize the index. Furthermore, an index should minimize the number of disk accesses during a search operation. In this paper, we have proposed an efficient spatial index strategy, called an NA-tree, which is designed for paged secondary storage,

Table 3. A comparison.

		R-tree	R^+-tree	R^*-tree	NA-tree
(1)	Region split method	Circumscribed rectangles split to minimize the areas	Circumscribed rectangles split to minimize the areas	Circumscribed rectangles split to minimize the overlap and region perimeters	No split of the spatial space; only organize data based on the spatial number
(2)	Internal nodes overlap (y/n)	Yes	No	Yes	No
(3)	Balance factor	Completely balanced	Completely balanced	Completely balanced	According to the data distribution, may be unbalanced.
(4)	Spatial search performance	Slower than others	Faster than R-tree	Faster than R-tree	Faster than others
(5)	Storage utilization (%)	60 ± 5	55 ± 5	70 ± 5	45 ± 5

and is dynamic, i.e., it can support arbitrary insertions and deletions of objects without any global re-organization. It efficiently supports exact match queries and range queries. From our simulation, we have shown that our NA-tree has a lower search cost than the R-tree, R^+-tree and R^*-tree. Moreover, similar to the R-tree (or R^*-tree) strategy, our NA-tree can handle point data [11, 27]. How to process *partial match queries* and *best match queries* are future research topics, where *partial match query* means to report all data objects which are located in a specific line, and *best match query* means to find the nearest neighbor of a specific data object.

REFERENCES

1. S. Berchtold, D. A. Keim, and H. P. Kriegel, “The X-tree: An index structure for high-dimensional data,” in *Proceedings of the 22nd VLDB Conference*, 1996, pp. 28-39.
2. S. Berchtold, D. A. Keim, H. P. Kriegel, and T. Seidl, “Indexing the solution space: a new technique for nearest neighbor search in high-dimensional space,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, 2000, pp. 45-57.
3. P. Baumann, “Management of multidimensional discrete data,” *VLDB Journal*, Vol. 3, 1994, pp. 401-444.
4. N. Beckmann, H. -P. Kriegel, R. Schneider, and B. Seeger, “The R^*-tree: An efficient and robust access method for points and rectangles,” in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990, pp. 322-331.
5. E. Bertino and B. C. Ooi, “The indispensability of dispensable indexes,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, 1999, pp. 17-27.
6. J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, Vol. 18, 1975, pp. 509-517.
7. J. L. Bentley and J. H. Friedman, “Data structure for range search,” *ACM Computing Surveys*, Vol. 11, 1979, pp. 397-409.

8. H. Blanken, A. Ubema, P. Meek, and B. V. D. Akker, "The generalized grid file: description and performance aspects," in *Proceedings of IEEE International Conference on Data Engineering*, 1990, pp. 380-388.
9. S. P. Dandamudi and P. G. Sorenson, "Algorithms for BD trees," *Software-Practice and Experience*, Vol. 16, 1986, pp. 1077-1096.
10. C. Faloutsos and I. Kamel, "Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension," in *Proceedings of ACM SIGMOD*, 1994, pp. 4-13.
11. V. Gaede and O. Guenther, "Multidimensional access methods," *ACM Computing Surveys*, Vol. 30, 1998, pp. 123-169.
12. I. Gargntini, "An effective way to represent quadtrees," *Communications of the ACM*, Vol. 25, 1982, pp. 905-91.
13. D. Greene, "An implementation and performance analysis of spatial data access," in *Proceedings of IEEE Data Engineering*, 1989, pp. 606-615.
14. O. Gunther and J. Bilmes, "Tree-based access methods for spatial databases: implementation and performance evaluation," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, 1991, pp. 342-356.
15. R. H. Guting, "An introduction to spatial database systems," *VLDB Journal*, Vol. 3, 1994, pp. 357-399.
16. A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD*, 1984, pp. 47-57.
17. P. W. Huang, P. L. Lin, and H. Y. Lin, "Optimizing storage utilization in R-tree dynamic index structure for spatial databases," *The Journal of Systems and Software*, Vol. 55, 2001, pp. 291-299.
18. A. Hutflesz, H. W. Six, and P. Widmayer, "The R-file: An efficient access structure for proximity queries," in *Proceedings of IEEE International Conference on Data Engineering*, 1990, pp. 372-379.
19. I. Kamel and C. Faloutsos, "Parallel R-trees," in *Proceedings of ACM SIGMOD*, 1992, pp. 195-204.
20. A. J. T. Lee, "An efficient access method for spatial databases," *International Computer Symposium*, 1996, pp. 33-40.
21. D. H. Lee and H. J. Kim, "SPY-TEC: An efficient indexing method for similarity search in high dimensional data spaces," *Data and Knowledge Engineering*, Vol. 34, 2000, pp. 77-97.
22. K. J. Li and L. Robert, "The spatial locality and a spatial indexing method by dynamic clustering in hypermap systems," in *Proceedings of IEEE International Conference on Data Engineering*, 1992, pp. 207-223.
23. A. Kumar, "G-tree: a new data structure for organizing multidimensional data," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, 1994, pp. 341-347.
24. T. C. T. Kuo and A. L. P. Chen, "Content-based query processing for video databases," *IEEE Transactions on Multimedia*, Vol. 2, 2000, pp. 1-13.
25. S. T. Leutenegger and M. A. Lopez, "The effect of buffering on the performance of R-trees," in *Proceedings of the 14th International Conference on Data Engineering*, 1998, pp. 164-171.
26. D. B. Lomet and B. Salzberg, "The hB-tree: A multiattribute indexing method with good guaranteed performance," *ACM Transactions on Database Systems*, Vol. 15, 1990, pp. 625-658.

27. Y. Nakamura, S. Abe, Y. Ohsawa, and M. Sakauchi, "A balanced hierarchical data structure for multidimensional data with highly efficient dynamic characteristics," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, 1993, pp. 682-694.
28. M. A. Nascimento and J. R. O. Silva, "Towards historical R-trees," in *Proceedings of ACM Symposium on Applied Computing*, 1998, pp. 235-240.
29. Y. Ohsawa and M. Sakauchi, "A new tree type data structure with homogeneous nodes suitable for a very large spatial database," in *Proceedings of IEEE International Conference on Data Engineering*, 1990, pp. 296-303.
30. D. Papadias, Y. Theodoridis, T. Sellis, and M. Egenhofer, "Topological relations in the world of minimum bounding rectangles: a study with R-trees," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995, pp. 92-103.
31. D. Papadias and Y. Theodoridis, "Spatial relations, minimum bounding rectangles, and spatial data structures," *International Journal on Geographical Information Systems*, Vol. 11, 1997, pp. 111-138.
32. E. G. M. Petrakis and C. Faloutsos, "Similarity searching in medical image databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, 1997, pp. 435-447.
33. J. T. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," in *Proceedings of ACM SIGMOD*, 1981, pp. 10-18.
34. H. Samet, *Design and Analysis of Spatial Data Structures,* Addison-Wesley, Reading Mass., 1990.
35. H. Samet, "Spatial data structure," *Modern Database Systems: The Object Model, Interoperability*, *and Beyond*, W. Kim, ed., Addison Wesley/ACM Press, Reading, MA, 1994, pp. 361-385.
36. S. Sarawagi, "Indexing OLAP data," *Bulletin of the IEEE Computer Society Technique Committee on Data Engineering*, Vol. 20, 1997, pp. 36-43.
37. B. Seeger and H. P. Kriegel, "The buddy-tree: An efficient and robust access method for spatial data base systems," in *Proceedings of the 16th VLDB Conference*, 1990, pp. 590-601.
38. T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th VLDB Conference*, 1987, pp. 507-518.
39. S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C. T. Lu, "Spatial databases − accomplishments and research needs," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, 1999, pp. 45-55.
40. K. L. Tan, B. C. Ooi, and L. F. Thiang, "Indexing shapes in image databases using the Centroid-Radii model," *Data and Knowledge Engineering*, Vol. 32, 2000, pp. 271-289.
41. Y. Theodoridis and T. Sellis, "A model for the prediction of R-tree performance," in *Proceedings of the 15th ACM Symposium on Principles of Database Systems*, 1996, pp. 161-171.
42. Y. Theodoridis, E. Stefanakis, and T. Sellis, "Efficient cost model for spatial queries using R-trees," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, 2000, pp. 19-32.
43. C. D. Tung, W. C. Hou, and J. H. Chu, "Multi-priority tree: An index structure for

spatial Data," in *Proceedings of International Computer Symposium*, 1994, pp. 1285-1290.

44. M. Vazirgiannis, Y. Theodoridis, and T. Sellis, "Spatial-temporal composition and indexing for large multimedia applications," *Multimedia Systems*, Vol. 6, 1998, pp. 284-298.

Ye-In Chang (張玉盈) was born in Taipei, Taiwan, R.O.C., in 1964. She received the B.S. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986, and M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, Ohio, in 1987 and 1991, respectively.

From August 1991 to July 1999, she jointed the faculty of the department of applied mathematics at National Sun Yat-Sen University, Kaohsiung, Taiwan. Since August 1997, she has been a Professor in the department of applied mathematics at National Sun Yat-Sen University, Kaohsiung, Taiwan. Since August 1999, she has been a Professor in the department of computer science and engineering at National Sun Yat-Sen University, Kaohsiung, Taiwan. Her research interests include database systems, distributed systems, multimedia information systems and mobile information systems.

Cheng-Huang Liao (廖正煌) was born in Hsinchu, Taiwan, R.O.C., in 1972. He received the B.S. and M.S. degrees in applied mathematic from National Sun Yat-Sen University in 1995 and 1997, respectively. He is currently a system design engineer of Realtek Semiconductor Corporation in Taiwan. He is doing some developing jobs about the networking device driver.

Hue-Ling Chen (陳慧玲) was born in Tainan, Taiwan, R.O.C., in 1978. She received the B.S. degree in computer and information science from National Chiao Tung University in 2000. She is currently a master student in department of computer science and engineering, National Sun Yat-Sen University. Her research interests include spatial databases and data mining.