History-independence: A fresh look at the case of R-trees

Theodoros Tzouramanis
Department of Information & Communication Systems Engineering,
University of the Aegean,
Karlovassi, Samos, 83200, Greece
ttzouram@aegean.gr

ABSTRACT
The deterministic tree-based database indexing structures unintentionally retain "additional" information about previous operations records of the index itself and about the database users' past activities. Privacy is jeopardized when a user gains illegitimate access to an indexing structure and accesses this "additional" confidential information from the internal representation of the index. This paper is the first to explore the design of history-independent access methods for spatial and multidimensional databases and proposes the History-Independent R-tree (HIR-tree), an R-tree variant which does not reveal information about the sequence of insertions, deletions and updates that have been applied to it and only reveals the outcome of these historical operations. It differs from the traditional spatial access methods in that the HIR-tree has the additional property of hiding its modification history. This property is achieved through the use of randomization by the update algorithms.

Categories and Subject Descriptors
H.2.2 [**Database Management**]: Physical Design – *Access methods*. H.2.8 [**Database Management**]: Database Applications - *Spatial databases and GIS*. K.4.1 [**Computers and Society**]: Public Policy Issues – *Privacy*.

General Terms
Algorithms, Performance, Design, Security.

Keywords
Algorithms, privacy, spatial and multidimensional databases, history-independence, access methods, R-tree.

1. INTRODUCTION

Electronic information, assumed erased, re-surfaces: Where high security and privacy protection are required, information that cannot be retrieved via the legitimate interface of a system (*e.g.*, what is visible on screen), should not be retrievable by any means. A simple illustration is a database indexing structure with lazy deletions that retains information believed to be long deleted: if this information is not part of the interface, the index should hermetically secure it in its internal representation and disclose nothing about the allocation in the data server memory.

This work focuses on indexing methods that are *history-independent*, *i.e.* it is impossible from the internal representation of the index to deduce any information not revealed by its current state. Indexing methods are considered to support insert, delete, update and search operations. Here, the only history not contained in the current state is the order of insertions, deletions and updates that have led to this state.

Providing history-independence to an indexing method is an interesting and intuitive problem from the point of view of research. Examples of application could be :(1) an indexing method used for storing the voting records in a paperless electronic voting system, (2) a spatial indexing method used in a computer-aid design (CAD) application to store the layout data of a newly designed complex system as with the Formula One Scandal in 2007 [15], (3) a spatial indexing method used in a military application to store military installations regional locations: In all three cases, the contents of the storage system should not reveal the order in which the objects of an operation were entered, be it the order in which the ballots were cast, the steps of a design process or the order of the locations of the military installation.

A large body of research aims to make data structures and access methods persistent [4], in order to make it possible to reconstruct previous states of the data structure from the current one [14]. This paper aims at the very opposite, *i.e.* to ensure that strictly no information can be deduced about the past. Hence an alternative name for history-independence is *anti-persistence* [13].

The paper explores the question of history-independence in multidimensional and spatial databases and proposes an anti-persistent indexing method for privacy preserving spatial database applications. Section 2 reviews related work. Section 3, formulates the problem and presents the relevant terminology. The History-Independent R-tree is defined and described in Section 4. Section 5 concludes and looks ahead to possible extensions to this work.

2. RELATED WORK

While this property can be incorporated in other tree-based spatial access methods as well, the present work introduces a method for efficiently incorporating history independence in R-trees [6]. The R-tree was chosen because it is the most popular of access methods for indexing spatial databases. Figure 1 shows an example set of four rectangles and two possible R-tree representations of these rectangles.

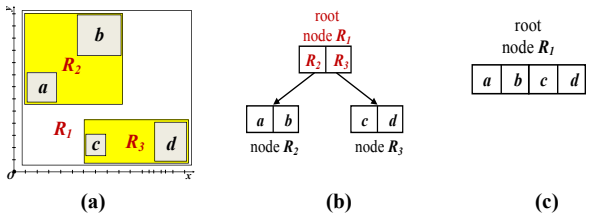

Figure 1. (a) some data objects, and (b) & (c) two possible R-tree representations for these objects.

SAC'12, March 25-29, 2012, Riva del Garda, Italy.

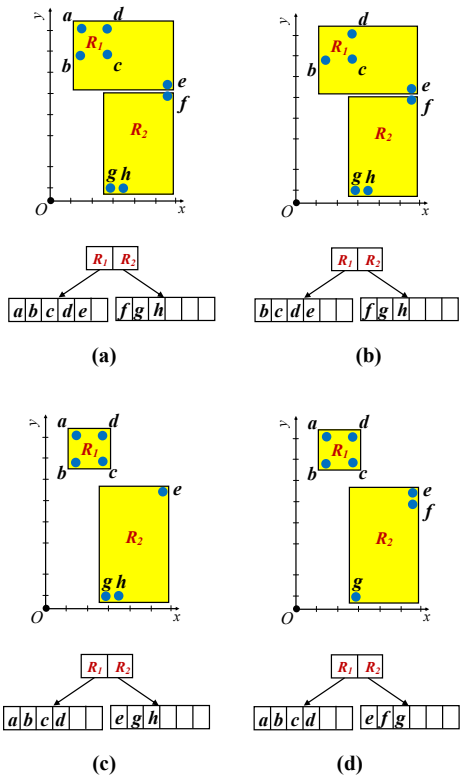

Figure 2. (a) an R-tree storing some points, (b) the R-tree prior to the last insertion, assuming that the last point to be inserted is any of those in the group $\{a, b, c, d\}$ (the figure assumes that the last point to be inserted is point a however the case of points b, c and d does not differ), (c) the R-tree before the last insertion assuming that the last point to be inserted is any of those in the group $\{e, f\}$, (d) the R-tree before the last insertion assuming that the last point to be inserted is any of those in $\{g, h\}$.

The simplicity and excellent performance of the R-tree have led to many variations being developed. They differ mainly in their splitting strategies. We focus on one of these variations, the Hilbert-R-tree [9] which uses space-filling curves, and specifically the Hilbert curve, to impose a linear ordering on the spatial data objects. Given the ordering, every node has a well-defined set of sibling nodes; thus, deferred splitting can be used, unlike most of the other R-tree designs (including [6]).

The structure of the R-tree (and, to the best of the author's knowledge, the structure of all R-tree variants, including the Hilbert R-tree) does not have a unique memory representation for the same dataset. Moreover, this representation depends on the order in which the data were inserted, deleted and updated into the tree. Therefore, the R-tree is a history-dependent indexing method and a malicious user (henceforth called "the observer"), by observing its internal representation, might be able to infer restricted-access information regarding the sequence of insertions, deletions or updates that have been applied to it. For instance, let us assume that the R-tree of Figure 1b has a minimum//maximum node capacity of 2//4 rectangles and that an observer gains access to the memory representation of the tree. The observer may then conclude with absolute certainty that at least one more item had been inserted into the tree at an earlier time that was deleted subsequently, before the observer gained access to the index, because if only insertions had been performed until now then the R-tree should have one leaf only: the R-tree in the state illustrated in Figure 1b should consequently not be accessible information.

It is not the case that the information disclosure regarding the history of insertions and deletions into the R-tree decreases as the maximum node capacity increases because the larger the tree becomes, the more opportunities the observer may have for investigation, by examining groups of nodes of the tree that have spatial proximity. See Figure 2a: assuming that deletions are not permitted and that the minimum//maximum R-tree node capacity is 3//6 points, the observer might infer that one of the points in the group $\{a, b, c, d\}$ was the last point that was inserted into the R-tree, having noted that if the object inserted last into the R-tree was any point from the group $\{e, f\}$ (or from the group $\{g, h\}$), then the R-tree memory representation before that last object insertion would have been the one in Figure 2c (or, respectively, in Figure 2d), therefore the R-tree in its final representation would store four objects in each leaf.

Micciancio first dealt with history-independence explicitly, and [11] studied the efficient incorporation of history-independence in the context of main-memory 2-3 trees. Micciancio realized that the structure of the tree gives away information about the order in which nodes have been inserted. To secure the privacy of the sequence of the modification operations, he proposed the Oblivious Tree, very similar to the main-memory 2-3 Tree, but history-independent. Insert and delete are defined as randomized algorithms; when a leaf is inserted or deleted, the Oblivious Tree makes local changes to the topology of the tree, based on the outcomes of a sequence of coin tosses.

History independence can also be derived from data structures that have a *canonical* or *unique representation* [13]. To this end, history independence means that if multiple updates occur between two adjacent snapshots, an observer learns nothing as to the order in which the updates occurred and the data server learns nothing if it receives the updates as a batch. In addition, it must not be possible for an observer to learn anything about the keys in one snapshot, given query responses from any other snapshots. In this context, the ordered hashing algorithm of [1] has the unique representation property. Also, Blelloch and Golovin [2] described a history-independent hash table for main memory, supporting insertions, deletions and queries in expected constant time and linear space, while, Naor et al. [12] developed a history-independent dynamic perfect hashing table supporting deletions, based on cuckoo hashing. Finally, Blelloch et al. [3] developed efficient history-independent data structures for problems in computational geometry and Golovin [5] proposed B-treap, a demonstrably uniquely-represented B-tree with guaranteed strong performance in its operation, but significantly complicated and difficult to implement.

This present work introduces a spatial access method the current memory representation of which does not reveal its history, and therefore does not reveal information about the sequence of insertions, deletions and updates that have been applied to it, and only reveals the final result of this sequence of historical operations. The proposed method is called *History-Independent R-tree* (HIR-tree) and is similar to the traditional R-tree and especially to the Hilbert R-tree, with the additional property that the only information conveyed by its structure is the set of rectangles stored in its leaves. This property is achieved through the use of randomization by the update algorithms.

3. PROBLEM FORMULATION AND DEFINITIONS

An indexing structure's *state* comprises of the contents of the indexing structure at a specific time point. A possible *memory representation* of an indexing structure for any given state is the physical contents of memory that represent that state. For the sake of convenience, we shall view a state as the set of memory representations that may represent that state. For example, considering the set of the four spatial objects of Figure 1a, we may, in Figures 1b and 1c, see two possible memory representations of the current state of an R-tree storing them: the first representation of the R-tree consists of two leaves and a root and the second one consists of a single leaf which is also the root of the tree.

An indexing structure is built from a list of *operations*, i.e., a list of insertion, deletions and updates. We assume that each operation takes one state deterministically to another state. Next we define the *state transition graph* based on a similar work on main-memory data structures [7]:

Definition 1 (State Transition Graph): *The state transition graph of an indexing structure is the directed graph induced on states (as vertices) of the indexing structure by the operations (directed labelled edges).*

The path on the graph that leads from a state $S1$ to another state $S2$ represents the sequence of operations that were performed on the structure from the state $S1$ to the state $S2$. It is obvious that if the indexing structure does not support deletions and updates then the graph will be acyclic (a DAG), *i.e.*, it will not be possible to return to a state once visited. On the other hand, if the indexing structure supports deletions and updates then the graph will be strongly connected, *i.e.*, all states will be reachable equally by one another.

On the basis of the above, starting from initialization (*i.e.*, from an empty index), each state on the graph is reachable through at least one path that corresponds to a unique possible sequence of operations. Also, each of these states may have more than one possible memory representation. The goal of history-independence is to make the indexing structure's evolution in time to depend only on the state (*i.e.* on the contents) of the indexing structure and not on the memory representation (*i.e.* not on the path on the transition graph that led to this content). For example, let us consider that there is a period of activity in an indexing structure (*e.g.* insertions, deletions and updates). At some point the observer gains control of the structure, *i.e.* sees exactly what is in the memory representing it. The observer should not be able to deduce any more about the sequence of operations that led to the content than is yielded by the content itself.

Therefore, a practical way of making an indexing structure history-independent is to introduce secret randomness into an indexing structure in such a manner that if an observer is not aware of the random choices then she cannot infer anything about the modification history. Take, for example, a geographic map storage mechanism for the storage of the current positions of military forces: history-independence through the use of randomization prevents the contents of the system from revealing information about the order in which the forces were moved across and only reveals the final position of the forces on the map (analogously to a paper-printed result).

The definition of the history-independent indexing structure therefore is:

Definition 2 (History-Independent Indexing Structure): *An indexing structure is history-independent if for any two sequences of modification operations X and Y that take the indexing structure from initialization ∅ to a state S, the distribution over memory representation after X is performed is identical to the distribution after Y. That is, if we compare the cases* $\varnothing \xrightarrow{X} S$ *and* $\varnothing \xrightarrow{Y} S$ *we will have:* $\forall\ s \in S$, $P(\varnothing \xrightarrow{X} s) = P(\varnothing \xrightarrow{Y} s)$, *where s denotes a possible memory representation of state S and* $P(\varnothing \xrightarrow{X} s)$ *denotes the probability that, starting from representation ∅, the sequence X of operations on the data structure yields representation s of S.*

We have to note that a memory representation s_1 is reachable from another representation s_2, denoted as $s_1 \longrightarrow s_2$ if there is at least one sequence of operations X such that $P(s_1 \xrightarrow{X} s_2) > 0$. In the following sections we will use randomization to construct and prove the validity of a history-independent R-tree variant.

4. THE HISTORY-INDEPENDENT R-TREE

The HIR-tree is a height-balanced tree that consists of intermediate and leaf nodes the layout of which is similar to that of the Hilbert R-tree. The HIR-tree description is as follows:

- Every node contains between b and B entries unless it is the rightmost node on its level or the root.
- Every rightmost node on a level (the root node, respectively) has at least one child node (two child nodes, respectively), unless it is a leaf.
- For each leaf entry <*o_id*, *R*>, *R* is the minimum bounding rectangle (*MBR*) approximation of the spatial object represented by object identifier *o_id*.
- For each non-leaf entry <*ptr*, *R*, *LHV*>, *ptr* is a pointer to a child node, *R* is the *MBR* that completely encloses the rectangles in that descendant node and *LHV* is the largest Hilbert value among the rectangles in the descendant node.
- The Hilbert value that represents every rectangle into the tree is the Hilbert value of its centre.
- All leaves appear at the same level.

The Hilbert R-tree was chosen because of its well defined set of sibling nodes for every node in the tree (a unique property in relation to most of the other R-tree designs).

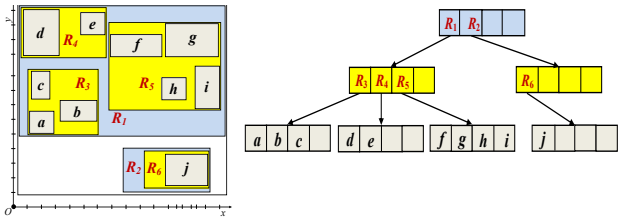

Figure 3. (a) a set of ten multidimensional objects, and, (b) the HIR-tree built on top of these objects.

The description of the HIR-tree differs from the Hilbert R-tree because the minimum node capacity on the Hilbert R-tree is fixed to $\lceil B/2 \rceil$ records and because the nodes along the rightmost path of the HIR-tree are permitted one entry only. This is not essential to the indexing structure but it helps with simplifying the description of the modification operation algorithms. As an example of the tree, Figure 3 illustrates several *MBRs* and the corresponding HIR-tree built on top of these rectangles, assuming a minimum//maximum node capacity of 2//4 records.

In order to insert a new object *e* into the HIR-tree, we start by calculating the Hilbert value of the center of the *MBR* of *e* and we traverse the tree in a B-tree manner, *i.e.*, by starting from the root tree node and by choosing the proper child based on the stored *LHV* values. When a leaf node is reached, *e* is inserted into the leaf and the *MBRs* of all its descendent nodes are enlarged properly in order to completely cover *e.* After the insertion of *e*, the target leaf will store between *b* and *B* entries. However, the final number of entries into the leaf will be chosen by the random outcome of a coin toss, in order to achieve history independence.

For the processing of a point or range query with respect to a query window *q* (which could be either a point or a rectangle), the R-tree traversal policy is followed. More precisely, by starting from the root node, all the tree nodes with *MBR* overlapping the query window *q* are traversed in an umbrella-like top-to-bottom fashion. When the leaf level is reached, all the data rectangles that overlap the query window *q* have to be reported to the user.

Operation	Description
Create(*N*)	builds a new tree storing a sequence *N* of *n* spatial objects at its leaves
Insert(*ob*, *T*)	inserts a new spatial object *ob* in the tree *T*
Delete(*ob*, *T*)	removes the spatial object *ob* from the tree *T*
Update(*ob*, *T*)	updates the spatial object *ob* from its old value ob_{old} to its new one ob_{new} in the tree *T*
Search(*q*, *T*)	performs a range query with respect to a given query window *q,* in the tree *T*

Table 1. The set of operations that act over HIR-tree.

Before going further into the description of the modification algorithms that operate the HIR-tree, on the basis of Definition 2 we formalize the requirement for this tree-based access method to be history-independent.

Definition 3 (Requirement for achieving History-Independence [11]): *Let M be a set of operations that act over a HIR-tree, and A be a set of algorithms implementing them. The set of algorithms A is history-independent if for any two sequences of operations* $p_1, p_2, ..., p_n$ *and* $q_1, q_2, ..., q_m$ *the following is true. If* $p_1, p_2, ..., p_n$ *and* $q_1, q_2, ..., q_m$ *generate trees storing the same set of rectangles, then the execution of the sequence of algorithms in A implementing* $p_1, p_2, ..., p_n$ *and the execution of those implementing* $q_1, q_2, ..., q_m$ *have identical output probability distributions*.

Table 1 lists the set of operations investigated by the author in a HIR-tree. The algorithms implementing these operations are probabilistic with the exception of the *Search*() operation which acts similarly to the original R-tree and for this reason it will be omitted in the sequel[1]. The tree *T* is generated firstly by running *Create*(*N*) with a sequence *N* of *n* objects (possibly with *n* = 0) and successively by applying a sequence of modification operations *Insert*(*ob*, *T*), *Delete*(*ob*, *T*) and *Update*(*ob*, *T*), where *ob* represents a given spatial object. The *Create*(*N*) and *Insert*(*ob*, *T*) algorithms are defined below, while the *Delete*(*ob*, *T*) and *Update*(*ob*, *T*) algorithms are omitted due to space limitation. However, the *Delete*(*ob*, *T*) operation is analogous to *Insert*(*ob*, *T*) and the *Update*(*ob*, *T*) operation can be implemented by a deletion *Delete*(*ob*, *T*) of a data entry followed by an insertion *Insert*(*ob*, *T*) of a new entry with the same object identifier *o_id* = *ob*.

4.1 The *Create* Operation

Given a sequence *N* of *n* objects, the *Create*(*N*) operation builds the HIR-tree in a bottom-up bulk-loading fashion, level by level, starting from the leaves of the tree. The bulk-loading strategy is based on the packing technique proposed in [8] which supports the pre-ordering of the input data based on the Hilbert value of the centre of the data rectangles. Algorithm 1 illustrates the pseudo code for the randomised *Create*(*N*) operation for the HIR-tree, considering a pre-sorted set *N* of *n* spatial objects.

```
Algorithm 1: the Create() operation
Input:   a presorted list N of n records in the
         form: <MBR of an object, Hilbert value
         of the MBR, Address of the object>.
         The list is sorted in ascending order
         based on the Hilbert value.
Output:  the HIR-tree T
1:  BEGIN
2:  REPEAT {
3:    N_upper = ∅;
          // N_upper is a temporary list with
          // similar description to list N
4:    WHILE N <> ∅ DO {
5:      Choose a random integer d ∈ [b, B];
6:      IF size(N) < d THEN
          //size(N) returns the num of objs in N
7:        Set d := size(N);
8:      Create a new tree node p and insert into
          p the d leftmost records of N; Remove
          these d records from N;
9:      Insert in N_upper the triplet
          <MBR(p), LHV(p), Address(p)>;
        //Address(p) is the address of p on disk
10:     }
11:   N := N_upper;
12: }
13: UNTIL N_upper ≤ 1;
14: RETURN Address(p); // p is now the new root
```

Algorithm 1. Pseudo code for the *Create*(*N*) operation, where *N* represents a list of spatial objects.

[1] A pseudo code for processing a range query with respect to a window *q* can be found in the original paper of the R-tree [6] as no modification is required. Regarding other popular spatial queries like the *k*-nearest-neighbor, similarity, skyline queries, spatial joins of various kinds, *etc*., the proposed method supports all the original algorithms that act on the R-tree [10].

Assuming a minimum//maximum R-tree node capacity of 2//4 entries and assuming that the outcomes of the coin tossing $d \in \{2, 4\}$ are 3, 2, 4, 2, 3, 2 and 4, the execution of *Create*() algorithm on the set of objects {*a*, *b*, ..., *i*, *j*} of Figure 3a generates the tree shown in Figure 3b (the first four coin tossing outcomes are spent for the grouping of the entries at the leaf level, the next two outcomes for the next higher level, *etc*.).

4.2 The *Insert* Operation

The execution of the insertion algorithm *Insert*(*ob*, *Create*(*N*)) for any sequence *N* of *n* objects and a new object *ob* must obtain the same output distribution as of *Create*(*N'*), where *N'* is the sequence obtained from *N* by adding into it the object *ob*. *Insert*(*ob*, *T*) is illustrated in Algorithm blocks 2-4.

```
Algorithm 2:  the Insert() operation, Part I
Input:        a new spatial object ob to be inserted into the HIR-tree with root p.
Output:       the HIR-tree T
1:  L = L_upper = ∅;
    // The lists L and L_upper contain "signed"
    // entries of the form:
    //<+/-,MBR(p),Hilbert(p),Address(p)>
2:  Insert into L the triplet:
        <+, MBR(ob), Hilbert(ob), Address(ob)>;
3:  Traverse the tree and locate the leaf u_h
        in level h in which the new object
        ob will be inserted;
4:  Keep in main memory all the nodes
        u(0), u(1), ..., u(h-1), u(h) in the
        path from the root u(0) to the leaf
        u(h);
5:  REPEAT {
6:    L_upper = ∅;
7:    p := u(h);
8:    TransferDataFromNode2List(L, L_upper, p);
9:    WHILE L <> ∅ DO {
10:      Choose a random integer d ∈ [b, B];
    //d is selected to be the new degree of u(h);
11:      Algorithm 3;
18:      Algorithm 4;
26:      }
27:   L := L_upper;
28:   h--;
29: }
30: UNTIL L_upper ≤ 1 or h < 0;
31: IF L_upper > 1 THEN
      //the tree's height needs to be increased
32:   p := Create(L);
33: RETURN Address(p);  // p is now the new root
```

Algorithm 2. Pseudo code for the *Insert*(*ob*, *T*) operation, where *ob* is a new spatial object and *T* is a HIR-tree.

The algorithm makes use of two lists *L* and L_{upper} which are both sorted in ascending order based on the Hilbert value. The lists contain "signed" entries of the form <+, *MBR*(*p*), *Hilbert*(*p*), *Address*(*p*)> and <–, (*MBR*(*p*), *Hilbert*(*p*), *Address*(*p*)>, where *p* is an object or a tree node. A list entry marked by the sign '+' ('–') is to be inserted into (deleted from) the tree in the corresponding level. The list *L* (L_{upper}) is used to store tree entries of the level currently under process (of the next upper tree level).

The algorithm also makes use of two auxiliary procedures, the *TransferDataFromNode2List* and *TransferDataFromList2ANode*. The first one transfers all the elements of a node *p* into list *L* all marked with the sign '+'. At the same time the triplet <–, *MBR*(*p*), *LHV*(*p*), *Address*(*p*)> is inserted into list L_{upper} and node *p* is erased by overwriting it with random bits 0 and 1. This may be significant since the disk might leak undesirable information. The second procedure, named *TransferDataFromList2ANode*, transfers entries from list *L* into a new node *p* while at the same time the triplet <+, *MBR*(*p*), *LHV*(*p*), *Address*(*p*)> is inserted in list L_{upper}. It has to be noted that if two identical triplets with different signs +/– co-appear in a list, then they are both removed from the list.

At the beginning of the *Insert*(*ob*, *N*) algorithm the new spatial object *ob* is inserted into list *L*. Then, after locating the appropriate leaf *p* to host the new object *ob*, its entries are transferred into lists *L* and L_{upper} (Line 8). Then by using a new coin toss the algorithm decides which will be the number of records that *p* will host (Line 10). There are two possible cases to distinguish in what follows: if *p* is the rightmost node in its level then Algorithm 3 is executed, otherwise, Algorithm 4 is executed.

```
Algorithm 3: the Insert() operation, Part II
11:     IF p is the rightmost node at
              level h THEN {
12:       IF size(L) < d THEN
13:         Set d := size(L);
14:       TransferDataFromList2ANode(L,
              L_upper, d);
15:       IF size(L) > 0
16:         TransferDataFromList2ANode(L,
                L_upper, size(L));
    // i.e. a new node will be created to be
    // the new rightmost node on this level
17:     }
```

Algorithm 3. Pseudo code for the *Insert*(*ob*, *T*) operation, Part II.

In Algorithm 3, if the new value *d* is larger than the number of records that list *L* hosts, then *d* is set to be equal to the number of entries in *L*. At the end of the operation, list *L* is emptied into at most two sibling tree nodes (Lines 14-16) which will be the new rightmost nodes in the level under process.

```
Algorithm 4: the Insert() operation, Part III
18:   ELSE IF p is not the rightmost
          node at level h {
19:     WHILE size(L) < d THEN {
20:       Locate the right sibling node p'
              of p at level h;
21:       TransferDataFromNode2List(L,
              L_upper, p');
22:       p := p';
23:     }
24:     TransferDataFromList2ANode(L,
            L_upper, d);
25:   }
```

Algorithm 4. Pseudo code for the *Insert*(*ob*, *T*) operation, Part III.

In Algorithm 4, if the new number of records that it was decided would be hosted by p is larger than the number of records that list L hosts, then at most two right sibling nodes p' of p are located and their entries are transferred into list L (Line 21). At the end of the algorithm, a new tree node is created and the d entries of L with the smaller Hilbert value are transferred into the new node.

The *Insert*(ob, T) algorithm continues by locating sibling nodes on the same level to p from left to right (by executing Algorithm 4) until no more records remain in L or until we reach the rightmost node of the tree for the level under process (Algorithm 3 is then executed). At this point the 'while' loop in Lines 9-26 exits and the algorithm starts the same process for the next upper level moving in a bottom-up level-by-level fashion (loop of Lines 5-30). At the end of the process, the height of the tree might need to be increased (or decreased) and this is dealt with in Lines 31-32.

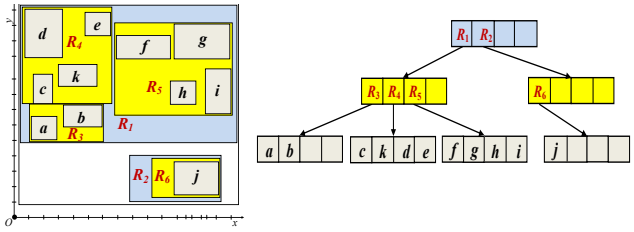

Figure 4. The insertion of object k into the HIR-tree of Figure 3.

Assuming the HIR-tree T on the set of objects $\{a, b, ..., i, j\}$ of Figure 3 and assuming that the outcomes of the coin tossing $d \in \{2, 4\}$ are 2, 4 and 3, the execution of *Insert*(k, T) algorithm for the insertion of the new object k generates the tree shown in Figure 4. The first coin tossing outcome is spent for the grouping of the entries at the leftmost leaf, the next outcome for its right sibling and the last tossing outcome is spent for their parent node.

We will now prove that the *Insert*(ob, T) operation is history-independent, by showing that the probability distribution obtained by running *Create*(N) on a sequence N of objects that produces a HIR-tree T, and then applying an *Insert*(ob, T) operation, where ob represents a given spatial object, is the same as the probability distribution which we would have obtained by running *Create*() directly on the final sequence of the $N' = N \cup \{ob\}$ objects. Therefore, we will prove that *Insert*(ob, *Create*(N)) outputs the same probability distribution as *Create*(N').

To prove this, we have to consider how the tree entries are grouped at level h by the *Insert*(ob, T) algorithm. First of all we have to notice that each iteration h of the 'repeat-until' loop of the algorithm modifies the topology of the tree only at level h. Let u_h be the node where a new entry was inserted by *Insert*(ob, T) at level h (note that if u_h is a leaf, the new entry is the object ob). The nodes that lie on the left side of u_h are not affected by *Insert*(ob, T). The node u_h and the nodes that lie on the right side of u_h will obtain exactly the same memory representation as they would have been obtained by the execution of the *Create*(N') operation, however, by using new coin tosses. Since the coin tosses used during the execution of the *Insert*() and *Create*() operations are independent, the probability distribution of the final result of the *Insert*(ob, *Create*(N)) is the same as if we had run the *Create*(N') operation directly on the modified set of the $N' = N \cup \{ob\}$ objects.

5. CONCLUSION

The paper introduced the History-Independent R-Tree (HIR-tree), an R-tree variant with the additional provable property that its evolution in time is independent from the sequence of the historical modification operations performed from initialization to the current state of the structure. The new indexing method supports efficient history-independent randomized algorithms for spatial object insertions, deletions and updates, while at the same time it is space efficient and fast, in an analogous way to the original R-tree. Future plans are to analytically and experimentally verify the space- and time efficiency of the proposed history-independent indexing method. It is expected that this work will open the way to the construction of many other efficient history-independent pointer-based indexing structures for a wide variety of applications.

6. REFERENCES

[1] O. Amble and D. Knuth: Ordered hash tables. *The Computer Journal*, 17(2), pp.135-142 (1974).

[2] G.E. Blelloch and D. Golovin: Strongly history-independent hashing with applications. In Proceedings of the *IEEE Symp. on Foundations of Computer Science*, pp.272–282 (2007).

[3] G.E. Blelloch, D. Golovin, and V. Vassilevska: Uniquely represented data structures for computational geometry. *In Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT)*, LNCS, Vol. 5124, pp.17-28, Springer, Heidelberg (2008).

[4] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan: Making data structures persistent. *J. Comput. Syst. Sci. (JCSS)*, 38(1), pp.86-124 (1989).

[5] D. Golovin: B-treaps: A uniquely represented alternative to B-trees. In Proceedings of the *ICALP Conf.*, pp.487-499 (2009).

[6] A. Guttman: R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD Conference*, pp.47–57 (1984).

[7] J.D. Hartline, E.S. Hong, A.E. Mohr, W.R. Pentney, and E. Rocke: Characterizing history independent data structures. *Algorithmica* 42(1), pp. 57-74 (2005).

[8] I. Kamel and C. Faloutsos: On packing R-trees. In Proceedings of the *Second International ACM Conference on Information and Knowledge Management (CIKM)*, pp. 490-499, Washington D.C., (1993).

[9] I. Kamel and C. Faloutsos: Hilbert R-tree: an improved R-tree using fractals. In Proceedings of the *VLDB Conference*, pp. 500-509, Santiago, Chile (1994).

[10] Y. Manolopoulos, A. Nanopoulos, A. Papadopoulos, and Y. Theodoridis: *R-trees: theory and applications*, Springer-Verlag (2005).

[11] D. Micciancio: Oblivious data structures: applications to cryptography. In Proceedings of the *STOC Conference* (1997).

[12] M. Naor, G. Segev, and U. Wieder: History-independent cuckoo hashing. In Proceedings of the *ICALP Conference Part II*, LNCS Vol. 5126, pp. 631-642. Heidelberg (2008).

[13] M. Naor and V. Teague: Anti-persistence: history independent data structures. In Proceedings of the *Thirty-Third Annual ACM STOC Symposium*, pp. 492–501 (2001).

[14] Y. Tao and D. Papadias: Efficient historical R-trees. In Proceedings of the *SSDBM Conference*, pp.223-232 (2001).

[15] Wikipedia: 2007 Formula One espionage controversy, Available at http://goo.gl/nj4N5, valid as of November 2011.