STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVII**, Number 2, 2002

FRINGED-QUADTREES: A NEW KIND OF DATA STRUCTURE

CLARA IONESCU

Abstract. In our everyday life we have to deal with different problems that, in most cases, need new data structures. At first sight, these structures do not look like any known data structure. This paper presents a data structure wehave called "fringed-quadtree". This structure is a tree with nodes that may be roots or leaves. A root-node may have at most four leaves. These trees may be built considering some rules that are presented in the paper. Due to the specific queries, the pointers will be ascending for the root-nodes and descending for the leaves. The time complexity of the described algorithms is logarithmic or linear and the memory space needed has the order $O(n)$.

1. Introduction

Data from database management systems are processed using special software programs. These have a lot of tools, but, in order to retrieve data, the users often need to create data structures, which will maintain the hierarchy between the elements and will perform searching as quick as possible. For example, the well-known *multilevel marketing systems (MLM)* are working on basis of various rules. These systems need to be able to retrieve records based on some hierarchy, (which are not stored explicitly in the database) in order to calculate the financial rights of persons from the system.

In this paper we present the fringed-quadtree, designed in order to have a suitable data structure for such a database. The quadtree, in a conventional approach, is a tree structure where every node may have at most four descendents. It was introduced for spatial data by Finkel and Bentley [Fink74].

This paper is organized as follows. We first define (section 2) the requirements of the model and the issues that must be considered in choosing a representation. This depends on the nature of the queries involving them, and on the type of operations that must be performed to answer them. Section 3 describes the manner of the building of such quadtrees. We discuss the implementation issues of the building in the subsection 4.0.1. For implementation we propose appropriate

2000 *Mathematics Subject Classification.* 68P05, 68P20.

1998 *CR Categories and Descriptors.* E.1 [**Data**]: Data Structures; H.2.1 [**Information Systems**]: Database Management – *Logical Design.*

search structures, for example balanced binary search trees [Adel62], [Knut73], [Wirt76] or B-trees [Come79]. Section 4 describes possible query types, and subsection 4.0.2 presents their implementations. Section 5 evaluates the performances of the algorithms (the storage and execution time requirements).

2. Problem Description

We will build a data structure having the following properties:

(1) Based on some specific rules, the elements are grouped in *buckets*. A bucket consists of at most five elements.

(2) The data associated to the nodes follows a hierarchy depending on the time-factor (the insertion time) and the parent-node.

(3) The structure contains two types of links (pointers) between its elements. The first link type specifies the *parent* of a given element. Obviously, each node has a single such pointer. The second link type is used for pointing to the descendents of a node; this is a *child*-type link. There will be four such pointers for each node, because they are used for retrieving the elements in the bucket corresponding to a parent-node.

(4) The structure contains two types of nodes: *roots* and *leaves*. *We define a root as a node that is referred as parent by at least one node, and a leaf as a node that is not referred as parent by any node in the structure.* The proper tree-structure consists of root-nodes, where each root is the nucleus of its own bucket. Such abucket may contain at most four other nodes that are leaves. These leaves are “hanged on the root like some fringes”.

(5) The dynamics used for building the structure leads to a quad-tree spanned (using the pointers) from its bottom part to its upper part, because only the *parent* pointers may be linked between the root-type nodes of the tree.

(6) When a new element is inserted, its *parent* must be given. This way, the hierarchical position of the new node is specified. The insertion of a new element may cause some changes in the bucket of its parent, as follows:

   (a) If the *parent* is a leaf, then the new node becomes a leaf and takes the place of *its own parent*. After this “replacement”, the *parent* pointers do not change, but the parent of the inserted node is no longer the child of a node (a leaf), but it becomes a root; hence, it is now a node in the quad-tree.

   (b) If the element is *not* the first leaf of its parent, it becomes one of the children (leaves) of the *parent*.

It follows that we have a quad-tree that is built based on ascending pointers and each node may have at most four descending pointers. We may notice that

a node may be referred as *parent* by five nodes, but the first such node "leaves" the parent's bucket. *The specificity of the dynamic used for building the buckets consists in the fact that the first child Z of a leaf Y becomes a child of the node X that referred Y as its child.* Hence, the parent of a leaf may be the node that contains the leaf in its bucket, or another root on the *parent pointers* path.

3. Building Fringed-Quadtrees

Initially, the structure consists of a single element, denoted by A. By convention, A is the only node predefined having root-type from the beginning. The bucket of the first element is built in a slightly different manner than the other buckets because the first element of the tree does not have a parent. Hence, it will be directly referred as *parent* by at most four nodes.

We suppose that, at each moment of time, only one node referring a certain parent may be inserted. Due to the fact that each existing node may be specified as *parent*, at a certain moment of time, the number of insertions may be equal to the number of nodes in the tree that do not have complete buckets.

Let we see the way a bucket is built. *After building the first bucket, no more leaves having as direct parent the root of the bucket may be inserted.* The structure of the buckets may change because when new insertions are performed, the *existent leaves* are replaced by their own new leaves.

We denote by X_0 the first node that refers X as its *parent* and by X_i, $i = 1 \dots 4$ the other nodes that refer X as their *parent*. The first child X_0 of X became child of the parent of X, and because A has no parent, A_0 does not exist. We denote the first child of X_i by X_{i0}, the others four children by X_{ij}, $j = 1 \dots 4$. For a better view of the notes we will rename the nodes wich became of root-type.

(1) The root A becomes the parent of the first leaf A_1.

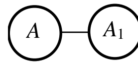

Figure 1: The first two nodes

(2) A is referred as the parent of the new node A_2. A_1 has a leaf, so it is referred as the parent of A_{10}; at this moment A_1 gains its independence and leaves the bucket of A, becoming a root (B). A_{10}, its first child, becomes a leaf of A.
(3) A refers to A_3 as its child; B (formerly A_1) is referred as parent by B_1 and A_2 is referred as parent by A_{20}; A_2 gains its independence and leaves the bucket of A becoming a root (C); A_{20} becomes a leaf of A. A_{10} is

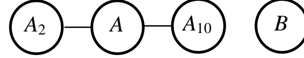

Figure 2: A_1 becomes a root (B)

referred as parent by A_{100}; A_{10} also gains independence and leaves the bucket of A becoming a root (D); its first child, A_{100}, becomes a leaf of A.

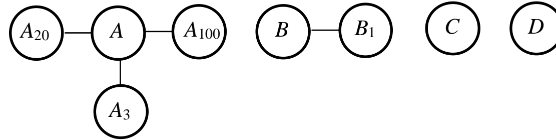

Figure 3: The structure has eight nodes; the bucket of A contains three leaves and the bucket of B contains one leaf; C and D do not yet have any leaf. A, B, C and D are root-nodes.

(4) A_4 is the new child of A, B_2 is the new child of B, C_1 is the new child of C and D_1 is the new child of D; then, A_{100} is referred as parent by A_{1000} and leaves the bucket of A, becoming a root (E), and A_{1000} becomes a leaf of A. A_{20} is referred as the parent by A_{200}; A_{20} becomes the root F and A_{200} becomes a leaf of A. A_3 is referred as parent by A_{30}; it becomes a root (G) and A_{30} becomes a leaf of A; B_1 is referred as parent by B_{10}, so it becomes a root (H) and B_{10} becomes a leaf of B.

The first loop ends here (the bucket of A is complete). One should not substitute "complete" with "finalized", because this term refers only to the number and it does not refer to the content of the bucket. A may not be referred as parent by any new node, but the content of its bucket changes due to the leaves of the leaves of A.

(5) We suppose that, in the next step, all nodes (except A, because its bucket is complete), roots and leaves, are referred as parents by new elements.

Figure 5 shows the bucket of A has the same number of elements, but A_{1000} was replaced by A_{10000}, A_{200} was replaced by A_{2000}, A_{30} was replaced by A_{300} and A_4 was replaced by A_{40}. Simultaneously, the sizes of the buckets of B, C, D, E, F, G, H increase and the leaves that became roots are I, J, K, L, M, N, P, O.

One may notice that at each step the number of nodes is increased by the number of root type nodes not having complete buckets and the leaves.

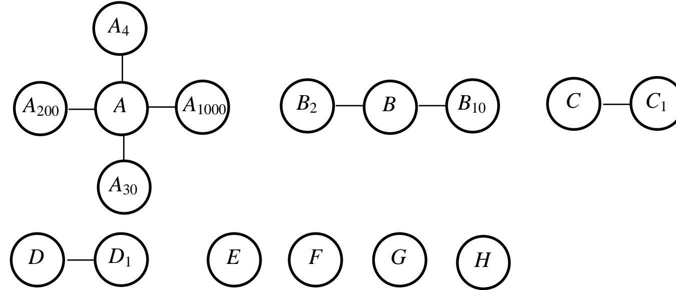

Figure 4: The links between the nodes after the fourth step of the insertions. There are eight roots (A, B, C, D, E, F, G, H) and eight leaves (A_{1000}, A_{200}, A_{30}, A_4, B_{10}, B_2, C_1, D_1).

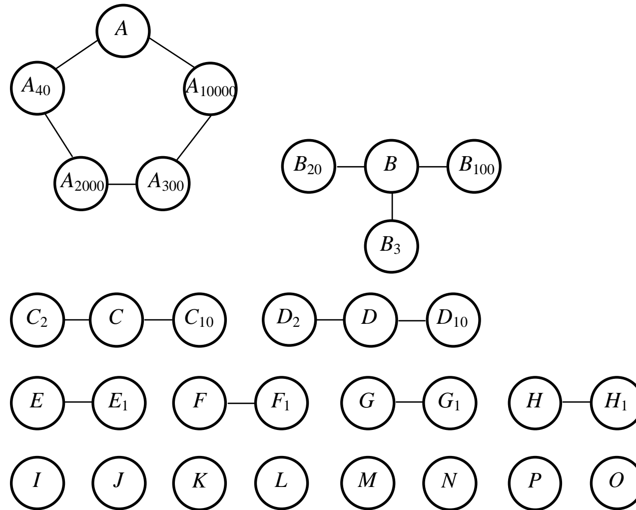

Figure 5: The leaves I, J, K, L, M, N, P, O became roots

The place (position) of an inserted node depends only on the node referred as parent by the node.

Obviously, the order in which the insertions are performed is not necessarily the same as in the example. It is possible to have some nodes referred as parents at most five times in a row and others to stay leaves for a long time.

4. Possible Queries

First of all, the problem of building such a structure must be considered.

Without reducing the generality, we may suppose that the information corresponding to a node is contained in a field called *name*. When designing such a data structure it is obvious that the possible queries that may be performed on the database modelled using this structure have to be considered. It is known that searching queries must be supported by such a database. We suppose that all queries, except for insertions, are performed on a single *name*. The insertion is performed using a search (we look for a node that has as field *name* the value specified as the parent of the new node); hence, two *names* will be given (for the new node and for the parent).

There are many types of queries, such as:

(1) Given a *name*, check whether it exists or not in the database.
(2) Given a *name*, find the *name* of its parent.
(3) Given a *name*, check whether it corresponds to a root or to a leaf; if it corresponds to a leaf, find the content of its bucket (the names of the leaves).
(4) Given a *name*, find all nodes that are ancestors of a node specified by its *name*. The *ancestors* of a node are the parent of the parent (the grandparent), the parent of the grandparent, etc., until the first inserted node is reached.
(5) Given a *name*, find all the names of the nodes for which the node corresponding to the *name* is an ancestor.

At first sight, the structure may be viewed as groups of five elements where we should know the root of each group. We might keep the roots in an alphabetically sorted list and keep trace of the children. In this way, the hierarchy hidden in the model is lost. From a logical point of view, this hierarchy is due to the fact that each node is inserted in the structure as a child of another node. But, here there are no "bosses" and no "subordinates".

In addition, the first child Z of a node Y is "given" to the node X that referred to Y as one of its children. From the logical point of view, X is on a superior level with respect to Y, hence Z and Y become "siblings", even if it looks like Z should be a descendent of Y. Hence, an implementation that uses tree-like data structures in which child-type pointers are used, does not correspond to the real situation from a logical point of view.

We propose an implementation that allows the buckets to be part of the structure (the content of the buckets is changeable) and also allows the hierarchy to be saved.

The root-nodes are maintained in a tree-like structure in which each node represents a bucket consisting in at least one and at most five nodes. For a node, we have the information field (*name*) and six pointers. One of them points to the *parent* specified when the node was inserted. The other five are child-type pointers. The first of them points to the node that was replaced at the insertion time. The other four point to the leaves (the other nodes in the bucket).

The tree-like structure is a quad-tree because each bucket consists in five nodes, one of them being the root of the bucket; hence, each node has at most four "descendents".

4.0.1. *Building Fringed-Quadtrees. Implementation.* We describe the way the quadtree is built based on a sequence of insertions. The following figures show the insertions presented in the example from section 3.

At the first step A_1 is inserted as child of A. For the node A the *parent* pointer is `nil` and *child*[1] will point to A_1. The other child-type pointers are `nil`. For the node A_1, *parent* points to A and the other pointers are `nil`. We use the following convention: the pointer corresponding to the first child of A points to A, in order to have a value different than `nil`.

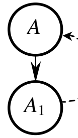

Figure 6: The first step

name	*parent*	*child*[0]	*child*[1]	*child*[2]	*child*[3]	*child*[4]
A	`nil`	A	A_1	`nil`	`nil`	`nil`
A_1	A	`nil`	`nil`	`nil`	`nil`	`nil`

For the node A the second child-type reference appears: *child*[2] points to A_2. The node A_2 is created in the same way as the node A_1 at the previous step. Due to the fact that A_1 refers to A_{10} as its first child, the pointer from the node A to A_1 is replaced by a pointer to A_{10}. Obviously, *parent* of the node A_{10} points to A_1. The node A_1 remains unchanged because its *parent* remains A. Its child-type pointers are `nil` because there are no leaves in its bucket.

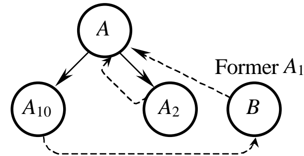

Figure 7: The second step

name	*parent*	*child*[0]	*child*[1]	*child*[2]	*child*[3]	*child*[4]
A	nil	A	A_{10}	A_2	nil	nil
B (formerly A_1)	A	A_{10}	nil	nil	nil	nil
A_2	A	nil	nil	nil	nil	nil
A_{10}	B	nil	nil	nil	nil	nil

A_3 is inserted as a leaf of A, so *child*[3] of the node A points to A_3. B (formerly A_1) receives its first leaf B_1 (the first node that referred it as *parent* became a leaf of A at the previous step), hence *child*[1] of the node B points to B_1. The pointer *parent* of the node B_1 points to B (the former A_1). A_{20}, the first descendent of A_2 is inserted so, *child*[2] of the node A points to A_{20}. The node A_2 (now C) remains unchanged, its *parent* field still points to A. A child of A_{10} (A_{100}) is inserted, hence the field *child*[1] from A is changed; now, it points to A_{100}. The node A_{100} is created in such a way that its *parent* field points to A_{10} (now D).

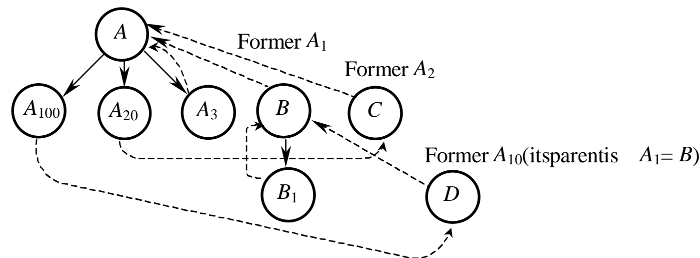

Figure 8: There are four root-type nodes and four leaves

name	*parent*	*child*[0]	*child*[1]	*child*[2]	*child*[3]	*child*[4]
A	nil	A	A_{100}	A_{20}	A_3	nil
B (formerly A_1)	A	D	B_1	nil	nil	nil
C (formerly A_2)	A	A_{20}	nil	nil	nil	nil
D (formerly A_{10})	B	A_{100}	nil	nil	nil	nil
A_3	A	nil	nil	nil	nil	nil
B_1	B	nil	nil	nil	nil	nil
A_{20}	C	nil	nil	nil	nil	nil
A_{100}	D	nil	nil	nil	nil	nil

At the next step, we suppose that each of the eight nodes in the structure is referred as *parent* by a new node that must be inserted. The tree has the pointers presented in the following table:

name	*parent*	*child*[0]	*child*[1]	*child*[2]	*child*[3]	*child*[4]
A	nil	A	A_{1000}	A_{200}	A_{30}	A_4
B	A	D	B_{10}	B_2	nil	nil
C	A	G	C_1	nil	nil	nil
D	B	H	D_1	nil	nil	nil
E (formerly A_3)	A	A_{30}	nil	nil	nil	nil
F (formerly B_1)	B	B_{10}	nil	nil	nil	nil
G (formerly A_{20})	C	A_{200}	nil	nil	nil	nil
H (formerly A_{100})	D	A_{1000}	nil	nil	nil	nil
A_4	A	nil	nil	nil	nil	nil
B_2	B	nil	nil	nil	nil	nil
C_1	C	nil	nil	nil	nil	nil
D_1	D	nil	nil	nil	nil	nil
A_{30}	E	nil	nil	nil	nil	nil
B_{10}	F	nil	nil	nil	nil	nil
A_{200}	G	nil	nil	nil	nil	nil
A_{1000}	H	nil	nil	nil	nil	nil

One may notice that the bucket of A is complete, that is A cannot have any more leaves that refer it as *parent*. But, when elements referring as parents the leaves of A are inserted, the new elements replace the leaves of A, becoming part of the bucket.

In order to clarify the way the fringed quad-tree is built, the next table presents an extra *optional* step.

name	*parent*	*child*[0]	*child*[1]	*child*[2]	*child*[3]	*child*[4]
A	`nil`	A	A_{10000}	A_{2000}	A_{300}	A_{40}
B	A	D	B_{100}	B_{20}	B_3	`nil`
C	A	G	C_{10}	C_2	`nil`	`nil`
D	B	H	D_{10}	D_2	`nil`	`nil`
E (formerly A_3)	A	M	E_1	`nil`	`nil`	`nil`
F (formerly B_1)	B	N	F_1	`nil`	`nil`	`nil`
G (formerly A_{20})	C	P	G_1	`nil`	`nil`	`nil`
H (formerly A_{100})	D	Q	H_1	`nil`	`nil`	`nil`
I (formerly A_4)	A	A_{40}	`nil`	`nil`	`nil`	`nil`
J (formerly B_2)	B	B_{20}	`nil`	`nil`	`nil`	`nil`
K (formerly C_1)	C	C_{10}	`nil`	`nil`	`nil`	`nil`
L (formerly D_1)	D	C_{10}	`nil`	`nil`	`nil`	`nil`
M (formerly A_{30})	E	A_{300}	`nil`	`nil`	`nil`	`nil`
N (formerly B_{10})	F	B_{100}	`nil`	`nil`	`nil`	`nil`
P (formerly A_{200})	G	A_{2000}	`nil`	`nil`	`nil`	`nil`
Q (formerly A_{1000})	H	A_{10000}	`nil`	`nil`	`nil`	`nil`
B_3	B	`nil`	`nil`	`nil`	`nil`	`nil`
C_2	C	`nil`	`nil`	`nil`	`nil`	`nil`
D_2	D	`nil`	`nil`	`nil`	`nil`	`nil`
E_1	E	`nil`	`nil`	`nil`	`nil`	`nil`
F_1	F	`nil`	`nil`	`nil`	`nil`	`nil`
G_1	G	`nil`	`nil`	`nil`	`nil`	`nil`
H_1	H	`nil`	`nil`	`nil`	`nil`	`nil`
A_{40}	I	`nil`	`nil`	`nil`	`nil`	`nil`
B_{20}	J	`nil`	`nil`	`nil`	`nil`	`nil`
C_{10}	K	`nil`	`nil`	`nil`	`nil`	`nil`
D_{10}	L	`nil`	`nil`	`nil`	`nil`	`nil`
A_{300}	M	`nil`	`nil`	`nil`	`nil`	`nil`
B_{100}	N	`nil`	`nil`	`nil`	`nil`	`nil`
A_{2000}	P	`nil`	`nil`	`nil`	`nil`	`nil`
A_{10000}	Q	`nil`	`nil`	`nil`	`nil`	`nil`

We recall that in this example we considered all possible insertions that *may be* performed at each moment of time even if this is not compulsory.

We now present an image of the fringed-quadtree described in the previous table. For a better view, the *parent* pointers of the leaves were removed.

Figure 9: The fringed-quadtree

Due to the hierarchy established by the *parent* pointers, the spanning must be performed from the bottom side to the upper-side of the quad-tree. The spanning may be optimised if data is inserted in a search structure such as a balanced binary searching tree or a B-tree [Ione91]. This means that the memory space needed to store a node increases due to the new pointers that refer to the search structure and other needed fields (for example, the factor of balance).

4.0.2. *Queries implementation.* We suppose that we have as searching key the field *name* of the nodes.

(1) For the first type of query we must check whether a given *name* is contained in the database. A search for this *name* will be performed in the search structure.
(2) For the second type of query we must retrieve the *parent* of a given node. If the node is in the search structure, than the *name* field of the node referred by the pointer *parent* is returned.
(3) For the third type of query we must establish whether a node is a root or a leaf. If, for that node, we have *child*[0] = `nil`, it follows that the

node is a leaf; otherwise, the node is a root. This type of query also asks the children of a root (the bucket content) to be retrieved. For a leaf, the algorithm halts here. For a root, the *names* of the nodes referred by the pointers $child[i]$, $i = 1 \ldots 4$ are returned. Obviously, it is not compulsory to have acomplete bucket, so will be returned the nodes pointed by $child[i] \neq$ `nil`.

(4) For the fourth type of query we must retrieve all the ancestors of a given node identified by its *name*.
   (a) We retrieve the given node (query of first type).
   (b) After finding the node we "climb" the tree until we reach the node having the value `nil` for the *parent* pointer. All the *names* of the nodes on the "way-up" are returned.

(5) For the fifth type of query we must retrieve all the nodes having as one of the ancestors a node identified by its *name*. Apparently, this is the inverse of the previous query where all the ancestors of a node had to be found. A closer look leads to the conclusion that, in fact, this is not an inverse, but a generalization. We must find paths that link a node (not the "oldest" ancestor) to certain nodes for which we *do not know* the *names*.
   (a) At the first step we retrieve the node corresponding to the *name* and return it.
   (b) If the pointer $child[0]$ has the value `nil`, the node is a leaf and the algorithm halts.
   (c) Otherwise, we return all the nodes that refer (directly or not) the node $child[0]$ as *parent*, which means we recursively call the algorithm for the node referred by $child[0]$. For the pointers $child[i]$, we "climb", using the *parent* pointers, until we reach a pointer to a child of the current node. At the next step we apply the spanning algorithm for this node.

```
1: procedure SPANNING(r)
2:     write r^.name
3:     if r^.child[0] ≠ nil then
4:         SPANNING(r^.child[0])
5:         for i = 1, 4, 1 do
6:             if r^.child[i] ≠ nil then
7:                 p ← r^.child[i]
8:                 while p^.parent ≠ r do
9:                     p ← p^.parent
10:                end while
```

```
11:             SPANNING(p);
12:         end if
13:     end for
14: end if
15: end procedure
```

5. Complexity Analysis

The memory space needed has the order of magnitude $O(n)$ because for each element we need a node. For the implementation we do not necessarily have to use dynamic memory allocation. We might build a database in which the records contain (apart from the corresponding information) six pointers. For the actual implementation we should find an efficient way to store the leaves (the nodes having all child-type pointers set to `nil`). This is not a waste of time because in a fringed quad-tree there may be $4n/5$ leaves.

Analysing the algorithms for the first three types of queries, it follows that the first step has a time complexity of $O(\log n)$, the time needed for a search in a structure similar to a balanced binary search tree [Ione91].

For the fourth query we have an algorithm running in $O(\log n + h)$ time, where h is the number of nodes returned. For the last query the algorithm runs in $O(\log n + m)$ time, where m is the number of nodes having the given node as ancestor. It follows that the algorithms for the fourth and fifths type of query have the order of magnitude $O(n)$ for the worst case.

For the fourth type of query the worst case is finding the ancestors of the only leaf in a fringed-quadtree in which all nodes (except for the leaf) have exactly one child (all the nodes of the fringed-quadtree must be returned).

For the fifth type of query the worst case is finding all the nodes having the fringed-quadtree root as ancestor (all the other nodes in the fringed-quadtree must be returned).

6. Further work

In this paper we described operations such insertion, search and several types of queries. Since deletion of a leaf-type node is trivial and the deletion of a root-type node is not allowed in the real model, we did not consider for the moment this operation. Obviously, we must be able to delete records from any kind of database, so the next issue will be to find a way of records deletion from a database implemented with fringed-quadtrees, without loosing the hierarchy. While developing the application, we will try to optimize as much as possible all the details regarding the implementation of the fringed-quadtrees.

References

[Adel62] **Adelson-Velskii, G.M., Landis, E.M.:** *An algorithm for the organisation of information*, Soviet Mathematics Doklady, 3:1259-1263, 1962.

[Come79] **Comer, D.:** *The ubiquitous B-tree*, ACM Computing Surveys 11, 2 (June 1979), 121–137.

[Fink74] **Finkel, R.A., i Bentley, J.L.:** *Quad trees: A data structure for retrieval on composite keys*, Acta Informatica, 4:1-9, 1974.

[Ione91] **Ionescu, C., Zsakó, I.** *Structuri arborescente cu aplicaţiile lor*, Editura Tehnică, Bucureşti 1991.

[Knut73] **Knuth, D.E.:** *The Art of Computer Programming, vol.I, Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

[Wirt76] **Wirth, N.:** *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

Babeş-Bolyai University, Cluj-Napoca, Romania
E-mail address: clara@cs.ubbcluj.ro