ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **10**, Number 4, 2007, 323–333

An externalization of the k-d tree

M. ADAM

Department of Mathematics and Computer Science,
University of Bucharest
Str. Academiei 14, Bucharest, Romania
E-mail: madam@math.math.unibuc.ro

Abstract. External searching is a fundamental problem with many applications in relational, object-oriented, spatial and temporal databases. The k-d tree or the multi-dimensional binary search tree used for associative searching, developed by J. L. Bentley in 1975, is an internal memory data structure. In this paper we develop an externalization of the k-d tree which occupies linear space, answers an orthogonal range query in square root time and supports updates in logarithmic time amortized.

Keywords: k-d tree, B-tree, orthogonal range searching, indexing, global rebuilding.

1. Introduction

External two-dimensional searching appears under different forms in practical applications, specially in databases domain. In the last years there has been much effort toward developing worst-case I/O-efficient external memory data structures for range searching in two dimensions by transforming a number of time/space efficient data structures for internal memory into I/O-efficient external ones.

This paper considers the k-d tree in two-dimensional space and develops an externalization of this data structure which preserves its space and time worst-case optimal bounds.

1.1. Memory model and related research

Given a set of N points in the plane we want to be able to find efficiently all points $p(x, y)$ contained into the query rectangle $q(x_1, y_1, x_2, y_2)$ with lower-left corner (x_1, y_1) and upper-right corner (x_2, y_2) i.e. $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. Let A

be their number, M the number of points that can fit into internal memory and B the size of a disk block ($B < M < N$). For notational simplicity, we define $n = N/B$, $m = M/B$ and $a = A/B$ to be the data set size, the internal memory size and, respectively, the query output size in units of blocks rather than points. An I/O operation (or simply I/O) is defined as a transfer of a block of data between internal and external memory. Computation can only be performed on objects in internal memory. The measures of performance in this model are the number of I/Os used to solve a problem and the amount of space used on disk, internal computation time being ignored. As practical values for the parameters defined above we have: $B = 10^3,\ M = 10^6,\ N = 10^{10}$ or 10^{12}.

B-tree is the most fundamental one-dimensional external memory data structure, developed by Bayer and McCreight [4], which corresponds to the internal memory balanced search tree. It uses linear space ($O(n)$ disk blocks) and supports updates in $O(\log_B N)$ I/Os. One-dimensional range queries, asking for all elements in the tree contained in the query interval can be answered in $O(\log_B N + a)$ I/Os. The space, update and query bounds obtained by the B-tree are the bounds we would like to obtain (optimal bounds) in general for more complicated problems. The above bounds have already been obtained for the two special cases of two-dimensional range searching: diagonal corner query and 3-sided planar query.

For two-dimensional range search, Subramanian and Ramaswamy [12] proved that $O(\log_B^c N)$ query require $\Omega\big((N/B)(\log_B N/\log_B \log_B N)\big)$ space, for any constant $c \geq 1$, in a natural external memory version of the pointer machine model of Chazelle [6]. A similar bound in a slightly different model using their optimal structure for 3-sided queries, was proved by Arge and al. [2] (*logarithmic query structure*). Kanth and Singh proved that $O(n)$ space requires $\Omega(\sqrt{n})$ query time and they obtained for their *O-tree* [9] the same bounds using ideas similar to the ones used by van Kreveld and Overmars in *divided k-d trees* [10]. In [3] is described an *O-tree* slightly different than the structure developed by Kanth and Singh, which uses an *external k-d-tree* at the terminal level. Grossi and Italiano [8] developed the elegant linear space *cross-tree*, which answers queries in $O(\sqrt{n} + a)$ I/Os. The *external k-d-tree*, the *O-tree* and the *cross-tree*, named *linear space structures*, can all be extended to d-dimensions in a straightforward way obtaining a $O(n^{1-1/d} + a)$ I/Os query bound.

Many other external data structures based on the partitioning of the data points or of the embedded space, such as *grid files*, various *quad-trees*, *space-filling curves*, *k-d-B trees*, *hB-trees* and various *R-trees* have been proposed. Often, these structures are preferred in applications, because they are relatively simple, require linear space and in practice perform well most of the time. However, they all have highly suboptimal worst-case performance, which deteriorates after repeated updates. The relevant literature is vast and was surveyed in [7], [13].

1.2. Overview of our results

Our main result in this paper is an optimal space, update and query time external memory data structure for two-dimensional orthogonal range searching, obtained by externalizing the well-known k-d tree.

In Section 2 we remember, in a few words, the standard k-d tree and review its main performances, emphasizing the fact that it is a static search structure.

In Section 3 we describe our idea to externalize the k-d tree using a B-tree variant. Here we give an algorithm for the construction of a dynamic index by splitting an overflowed internal node and local rebuilding the subtree rooted in it.

In Section 4 we present and analyze the basic operations on our external k-d tree. The search operations are represented by point query and orthogonal range query. Next we describe the strategy to rebalance the tree as a consequence of repeated updates and we calculate the amortized cost of this operation. Finally, we present in detail the insertion and deletion of a data point together with an amortized cost per update operation.

2. The structure in internal memory

Bentley's *k-d tree* [5] is a binary search tree that generalizes the 1-d tree or the ordinary binary search tree to $\mathbb{R}^k$. The most common variant of the point k-d tree in $\mathbb{R}^2$ (and the one we focus in this section) partitions the underlying space with the N data points by cycling through the two axis in a predefined and constant order. We obtain a binary tree of height $O(\log_2 N)$ with the N points stored in its leaves. The internal nodes represent a recursive decomposition of the plane by means of axis-orthogonal lines that partition the set of points into two subsets of equal size. In this way, with each node v of the tree is naturally associated a rectangular region R_v, and the nodes on any given level of the tree partition the plane into disjoint regions. In particular, the regions associated with the leaves represent a partition of the plane into rectangular regions containing exactly one point each.

The following theorem synthetizes the performances of the k-d tree [5].

Theorem 1. *Given N data points in the plane, a k-d tree can be built by inserting points in random order into an initially empty tree in time $O(N \log_2 N)$, the average cost of insertion, as well as of searching for a point is $O(\log_2 N)$. Deletion of nodes from k-d trees is considerably more complex than for binary search trees (unlike the binary search tree, not every subtree of a k-d tree is itself a k-d tree), its cost being dominated by $O(\sqrt{N})$, but in the average is still $O(\log_2 N)$. Finally, the cost of a range search is $O(2\sqrt{N})$.*

The expected performance of a k-d tree holds under the assumption that it is random. However, in practical applications, this assumption does not always hold.

Unfortunately, it is difficult to maintain a balanced tree structure over insertions/deletions of data points, so the k-d tree remains an efficient static search structure.

3. An external dynamic k-d tree

From the previous section, the k-d tree is a binary tree having in its leaves the data points. We know that B-tree is a fundamental one-dimensional external memory

data structure which uses linear space and supports queries and updates in logarithmic time. In order to obtain an externalization of the k-d tree similar to a B-tree, we will build a tree structure of order B with $O(n)$ leaves containing between $B/2$ and B data points each.

Lemma 1. *A static external k-d tree for storing a set S of N data points in the plane, given a priori, uses linear space and can be constructed in $O(n \log_B n)$ I/Os.*

Proof. We first create, with $O(n \log_m n)$ I/Os, two lists with the points in S, one sorted by x-coordinates and the other sorted by y-coordinates. Then, we create the index tree structure T top-down with the following recursive algorithm:

(1) If we consider to begin partitioning of S with the x-coordinate, we scan the list sorted by x-coordinates and determine the $O(B)$ vertical split values which make the root of T.

(2) Then, we partition the two sorted lists according to these values in $O(B)$ sublists with $O(n)$ I/Os and continue the partitioning by y-coordinates.

(3) In this way, we recursively construct the rest of the tree, level by level, the process being stopped when the sublists resulted by partitioning have between $B/2$ and B data points each.

The tree T resulted has height $\log_B n$, $O(n)$ leaves with points, $O(n)$ internal nodes with the partitioning (routing) values and therefore its construction cost is $O(n \log_B n)$ I/Os. □

In order to maintain a balanced tree structure over updates of data points, we make some remarks:

- The routing values in internal nodes of the tree cannot move from one level to another adjacent level in the tree, due to the fact that there is no total order in the plane.
- The split of an internal node that points to $O(B)$ blocks with $O(B)$ data points each, by a value belonging to the orthogonal axis, will charge $O(B)$ points 1 I/O.
- In a tree where every two adjacent levels have internal nodes with routing values from two different axis, a node split propagation up to the root is a costly operation.

With these remarks in mind, we begin the construction of a dynamic index in the following way:

(1) We start with one block of data points which is divided repeatedly in the same direction, say x-axis, until we obtain the $\Theta(B)$ partitioning points that make the first internal node r of the tree.

(2) Later, when one of its $\Theta(B)$ leaves overflows, r will split in the orthogonal direction into two nodes of equal size, and the partitioning value on the y-axis will make the new root. An example with a 2-3-4 search tree is presented in Fig. 1.

(3) The tree construction continues upwards: if a data block overflows and its parent is full, the split will take place before the first not full internal node on the path up to the root.

Finally, the tree obtained has all leaves with data blocks on the last level, the internal nodes from one given level contain routing values on the same axis and two adjacent levels correspond to orthogonal axis. Here, we must say that the tree grows due to a node split, a costly operation which implies the local rebuilding of the subtree with the root in that node (Lemma 1).

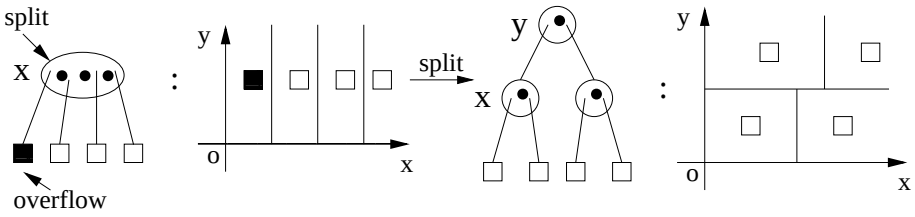

Fig. 1. An overflow at the blackened leaf produces a split of its parent.

There is yet another idea to construct a dynamic index, slightly different from that already presented:

(1) We start with one block of points which is divided repeatedly in the same direction, say x-axis, until we obtain the $\Theta(B)$ routing values of the tree root r.

(2) Then, when one of its $\Theta(B)$ leaves overflows, this block will be divided in the orthogonal direction (y-axis). Therefore, the data blocks may appear on the last two levels of the tree. This situation is presented in Fig. 2.

Remark. Due to this partitioning in another direction that advances downwards the tree, it will appear internal nodes having degree less than $B/2$ that have to be grouped in blocks in order to better use the space.

(3) When during this process a data block overflows and its partitioning determines the appearance of data blocks on a third level in the tree, it requires a split before the first not full internal node on the ascending path to the root (Figure 2).

Remark. In this way, the tree grows upwards too and its balance is improved.

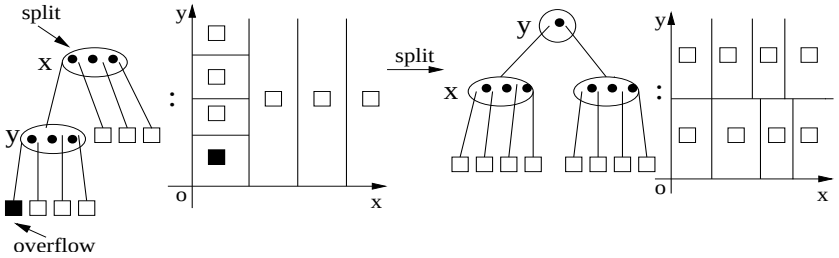

Fig. 2. An overflow at the blackened leaf produces a split of the root, which improves the balance of the tree.

The tree construction continues by cycling through the two orthogonal directions, downwards or upwards. In the case of random data points we can reduce the number of node splits and the tree grows especially downwards.

With the first construction algorithm, we can formulate the following definition of this external k-d tree.

Definition. *An external* 2-d *tree* T is a B-tree variant with the following properties:

- All leaves of T (blocks with data points) are on the last level of the tree and contain between $B/2$ and B points each.
- All internal nodes on the same level of T contain branching values from one axis and have degree between $B/2$ and B (the root has degree between 2 and B).
- The two axes come in cyclic order from the last level upwards the tree, until a certain condition is verified, as we will see in the next section, when one axis is maintained constant till the root of T.

4. Basic operations on the structure

Now, we will present and analyze the query and the maintenance of the external k-d tree T constructed on a set S of N data points in the plane which changes dynamically. Although there are several search algorithms for partial match, exact match and variants of nearest neighbor queries in k-d trees, we will consider here only point query and orthogonal range query. As concerns the maintenance of the structure, we will discuss the insertion/deletion of individual points, by developing algorithms efficient in the amortized sense (taking a sequence of update operations).

4.1. Query

A point query on S, that is a search for a given point p_0, can obviously be answered in $O(\log_B n)$ I/Os by following one root-to-leaf path in T corresponding to all nodes v whose region R_v associated contains p_0.

A range query $q\,(x_1, y_1, x_2, y_2)$ on S can be answered with a simple recursive procedure starting at the root r of T and advancing the query to its child v if q intersects the region R_v associated with v. At a leaf u we return the points in u contained in q. If on the first c levels in T it partitions with the same type of lines, say verticals, we descend in T with x_1 and x_2 c levels, in order to find the subtrees with roots on this level, which contain points in q. Their number is a constant for a large B and each of them intersects q with the cost that will be established below. Let T be of this type, with all internal nodes having $\Theta(B)$ children. To bound the number of nodes in T visited when answering q, or equivalently, the number of nodes v where R_v intersects q, we must bound the number of nodes v where R_v intersects a vertical/horizontal line l. We suppose that l is a vertical line and the root r of T corresponds to x-axis. The region R_r associated with r is obviously intersected by l, but as the regions associated with its $\Theta(B)$ children represent a subdivision of R_r with vertical lines, only one region R_v associated with its child v is intersected.

Because the region R_v is next subdivided by horizontal lines, the regions associated with all $\Theta(B)$ children of v are intersected by l. Let $L = O(n)$ be the number of leaves in T. As the children of v are roots of subtrees in T having L/B^2 leaves each, the recurrence for the number of regions intersected by l is $Q(L) = 2 + B \cdot Q(L/B^2)$. This is a general divide-and-conquer recurrence of the form $Q(L) = a \cdot Q(L/b) + c$, with $c = 2$, $a = B$ and $b = B^2$, which has the solution $\Theta\left(L^{\log_{B^2} B}\right) = \Theta\left(L^{1/2}\right)$. Similarly, we can show that the number of regions intersected by a horizontal line is $\Theta\left(L^{1/2}\right)$. Till now, we considered T full, but as a dynamic structure, each internal node of it has between $B/2$ and B children. It exists, obviously, $B/2 \leq B^* \leq B$, so that T can be rebuilt with the same height and having each internal node with exactly B^* children, in order to be verified the previous recurrence. In the worst case, when T is dynamic, we obtain $Q(L) \leq 2 + B \cdot Q\left(L/\left(B^2/2\right)\right)$, with the solution $Q(L) = O\left(L^{1/2+\epsilon}\right)$, where $\epsilon = f(B)$ is a decreasing function with values in $(0, 1/2]$ and close to 0 for a large B. Therefore, the number of nodes v with regions R_v intersected by the boundary of q is $O(\sqrt{n})$.

All the additional nodes visited when answering q correspond to regions completely inside q. Since a region R_v corresponding to an internal node v is only completely contained in q if the regions corresponding to all the leaves in the subtree with the root v are contained in q, the total number of regions completely inside q is $O(A/B)$, where A is the size of the answer to q. Also, $O(A/B)$ is the number of leaves with regions completely inside q, since each leaf contains $\Theta(B)$ points.

Therefore, the previous analysis leads to the following result.

Lemma 2. *An external k-d tree for storing N points in the plane supports point queries in $O(\log_B n)$ I/Os and orthogonal range queries in $O(\sqrt{n} + A/B)$ I/Os, where A is the size of the answer.*

4.2. Updates

During the insertion/deletion of points in/from an external k-d tree T built with N data points, its structure may become out of balance and the searching time will grow due to this fact. In order to rebalance the tree we utilize a local rebuilding strategy, that will be described and analyzed next.

Let $p(x, y)$ be the data point whose insertion produces an overflow at the leaf l and if l splits, $parent(l)$ must split too. Let w be the child of the first not full internal node on the ascending path from l to the root of T. In Fig. 3 is shown an example with the same 2-3-4 search tree and the situation described above. If w splits, a new routing value and a reference will be inserted in $parent(w)$, but this value must be a y-coordinate. Therefore, the entire subtree rooted in w will be rebuilt with $O\left(\frac{N_0}{B} \log_B \frac{N_0}{B}\right)$ I/Os, where N_0 is the number of data points in its leaves (Lemma 1). In other words, with $O\left(\frac{N_0}{B} h_0\right)$ I/Os we can rebalance a subtree with height h_0 having N_0 data points, and insert a new routing value in the parent node of its root on the level $h_0 + 1$. So, by repeating the operation, this node will become full and will split.

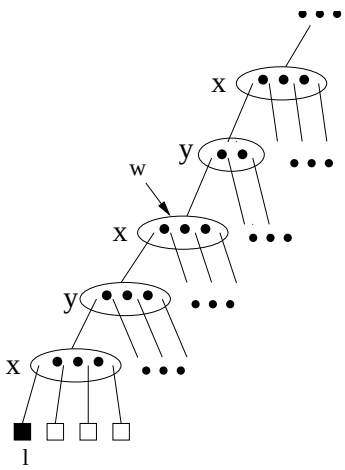

Fig. 3. An overflow at the leaf l produces directly the split of w.

To draw a conclusion, supposing level k in T, the number of I/Os necessary to fill up an internal node on level $k+1$ is given by the following expression:

$$\left(\cdots\left(\left((1\cdot O(B))\,\frac{B}{2}+2\cdot O(B^2)\right)\frac{B}{2}+3\cdot O(B^3)\right)\frac{B}{2}+\cdots+k\cdot O(B^k)\right)\frac{B}{2}=$$

$$\left(\cdots\left(\left(1\frac{1}{2}+2\right)\frac{1}{2}+3\right)\frac{1}{2}+\cdots+k\right)\frac{1}{2}B^k\cdot O(B)=a_k\frac{1}{2^k}B^k\cdot O(B), \qquad (1)$$

where $(1\cdot O(B))\,\frac{B}{2}$ I/Os produce an overflow at level 1, $2\cdot O\left(B^2\right)$ I/Os is the cost of rebuilding a subtree with 2 levels, ... and a_k is given by the recurrence:

$$a_i=i\cdot 2^{i-1}+a_{i-1},\quad \textit{with}\ \ i>1\ \ \textit{and}\ \ a_1=1. \qquad (2)$$

On the other hand, the number of data points that need to be inserted in T in order to have an overflow at an internal node on level $k+1$ results from the expression:

$$\left(\cdots\left(\left(\left(\frac{B}{2}O(B)\right)\frac{B}{2}+\frac{B}{2}O(B)\right)\frac{B}{2}+\frac{B}{2}O(B)\right)\frac{B}{2}+\cdots+\frac{B}{2}O(B)\right)\frac{B}{2}=$$

$$\left(\frac{B}{2}\right)^{k+1}\cdot O(B)+\left(\frac{B}{2}\right)^{k}\cdot O(B)+\cdots+\left(\frac{B}{2}\right)^{2}\cdot O(B)=$$

$$\frac{(B/2)^k-1}{B/2-1}\left(\frac{B}{2}\right)^{2}\cdot O(B)\approx\left(\frac{B}{2}\right)^{k+1}\cdot O(B), \qquad (3)$$

where $\frac{B}{2}O(B)$ inserted points produce an overflow at level 1, $\left(\frac{B}{2}O(B)\right)\frac{B}{2}$ points fill up an internal node at level 2 which is overflowed by another $\frac{B}{2}O(B)$ inserted points, etc.

Therefore, from the expressions (1) and (3) we obtain the condition to amortize the I/Os at the insertion of data points, i.e. the maximum level number i for which

$$a_i \leq B/2. \tag{4}$$

This condition says that, if during the update of T, its subtree T_0 which has been changed becomes out of balance, the cost of local rebuilding T_0 amortizes only if its height $h_0 \leq i$.

Now we can describe in detail the basic update operations.

Insert. To perform the insertion of a point $p(x, y)$ in T we execute the following steps:

(1) Starting from the root, we perform a point query to find the leaf l with the region R_l associated containing p and insert p in l. On this descending path we keep the address of the last not full internal node u (with $< B$ children).

(2) If l now contains $B + 1$ points and $u = parent(l)$, we simply split l into two leaves containing $B/2$ points each, using a line with the same inclination as the routing values in $parent(l)$, which will be inserted in $parent(l)$.

(3) If l contains $B + 1$ points and $u \neq parent(l)$, we must split the $descendant(u)$ on the path to l and insert a routing value in u thus: if the level corresponding to $descendant(u)$ is at most i, from relation (4), we make a local rebuilding of the subtree rooted in $descendant(u)$, else we rebuild the subtree rooted in the internal node on level i and continue to split all nodes up to $descendant(u)$, all routing values inserted being on the same predefined axis.

So, the tree resulted is shown in Fig. 4, where all leaves are on level 0, between levels 1 and i the axes come in cyclic order and then till the root a predefined axis is maintained constant.

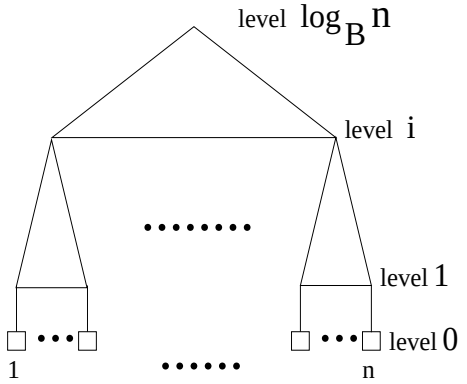

Fig. 4. A schematic representation of the external k-d tree.

The complexity of inserting p in T with N points follows from: $O\left(\log_B \frac{N}{B}\right)$ I/Os to find the leaf l, $O(1)$ I/Os amortized to rebuild a subtree with height $h_0 \leq i$ and $O(1)$ I/Os for every additional split up till the root of T.

Delete. Similarly, to delete a point $p(x, y)$ from T we perform the following steps:

(1) At first, we find and remove p from the relevant leaf l. On the descending path we keep the address of the last not half-full internal node u (with $> B/2$ children).

(2) If l now contains $B/2 - 1$ points, we fuse it with one of its siblings l', i.e., we delete l' and insert its points in l. If this results in l containing more than B points, we split it into two leaves and update the corresponding routing value from $parent(l)$ (share).

(3) If l, after fusing, contains at most B points and $u = parent(l)$, we delete the respective routing value from $parent(l)$, its degree decreasing by one.

(4) If l, after fusing, contains at most B points and $u \neq parent(l)$, fuse operations on internal nodes may propagate up till $descendant(u)$ and we'll act in this way: if on the ascending path to u we find an internal node half-full (with $B/2$ children) v, which has one of its siblings v' not half-full, we make a local rebuilding of the subtree rooted in the partitioning value between v and v' and update this value from $parent(v)$ (share), else let $v = descendant(u)$ and we make a local rebuilding of the subtree rooted in the partitioning value from u between v and one of its siblings, which is half-full too, this triplet from u being replaced by the address of the new subtree's root; as before at insertion, if the level corresponding to the root w of the rebuilding subtree is at most i, from relation (4), we make a local rebuilding of w, else we rebuild the subtree rooted in the internal node on level i and continue fusing up till at most $parent(v)$.

Another strategy to deal with deletions is to mark the deleted point in its leaf and to cover it later with an inserted point, and also ,periodically, rebuild the structure after a specified number of deletions.

The complexity of deleting p from T is of the same type as at insertion and we can formulate a concluding result.

Lemma 3. *An external k-d tree on N points in the plane supports updates in $O(\log_B n)$ I/Os amortized.*

Therefore, based on the three previous lemmas, we can give the final result which is optimal.

Theorem 2. *An external k-d tree for storing a set of N points in the plane uses linear space and supports point queries in $O\left(\log_B \frac{N}{B}\right)$ I/Os, orthogonal range queries in $O(\sqrt{N/B} + A/B)$ I/Os, where A is the size of the answer and updates in $O\left(\log_B \frac{N}{B}\right)$ I/Os amortized.*

5. Conclusions

In this paper we developed an external k-d tree in the plane with optimal worst-case bounds for time and space. Its dynamization uses a local rebuilding strategy

which makes updates optimal in the amortized sense. Although, we do not discuss here about the implementation details of the structure, we appreciate these important in order to reduce the rebuilding time. In connection with this subject, comes the generalization of this structure in higher-dimensional space. The practical performance of this worst-case efficient external memory structure also needs to be investigated.

References

[1] AGARWAL P. K., ARGE L.,PROCOPIUC O.,VITTER J. S., *A framework for index bulk loading and dynamization*, in *Proc. International Colloquium on Automata, Languages and Programming*, LNCS 2076, pp. 115–127, 2001.

[2] ARGE L.,SAMOLADAS V., VITTER J. S., *On two-dimensional indexability and optimal range search indexing*, in *Proc. ACM Symp.Principles of Databases Systems*, pp. 346–357, 1999.

[3] ARGE L., *External Memory Geometric Data Structures*, EEF Summer School on Massive Datasets, Springer Verlag, 2004.

[4] BAYER R.,McCREIGHT E., *Organization and maintenance of large ordered indexes*, Acta Informatica, **1**, pp. 173–189, 1972.

[5] BENTLEY J. L., *Multidimensional binary search trees used for associative searching*, Communications of the ACM, **18**, pp. 509–517, 1975.

[6] CHAZELLE, B., *Lower bounds for orthogonal range searching*, Journal of the ACM, **37**(2), pp. 200–212, 1990.

[7] GAEDE V., GÜNTER O., *Multidimensional access methods*, ACM Computing Surveys, **30**(2), pp. 170–231, 1998.

[8] GROSSI R., ITALIANO G. F., *Efficient cross-tree for external memory*, in Abello J. and Vitter J. S., editors, *External Memory Algorithms and Visualization*, pp. 87–106, American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.

[9] KANTH K. V. R., SINGH A. K., *Optimal dynamic range searching in non-replicating index structures*, in *Proc. International Conference on Database Theory*, LNCS 1540, pp. 257–276, 1999.

[10] VAN KREVELD M. J., OVERMARS M.H., *Divided k-d trees*, Algorithmica, **6**, pp. 840–858, 1991.

[11] PROCOPIUC O., AGARWAL P. K.,ARGE L.,VITTER J. S., *Bkd-tree: A dynamic scalable kd-tree*, in *Proc. International Symposium on Spatial and Temporal Databases*, LNCS 2750, 2003.

[12] SUBRAMANIAN S., RAMASWAMY S., *The P-range tree: A new data structure for range searching in secondary memory*, in *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pp. 378–387, 1995.

[13] VITTER J. S., *External memory algorithms and data structures*, in Abello J. and Vitter J. S., editors, *External Memory Algorithms and Visualization*, pp. 1–38, American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.