Journal of Discrete Algorithms 8 (2010) 373–387

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

ISB-tree: A new indexing scheme with efficient expected behaviour ☆

Alexis Kaporis [a], Christos Makris [a,b], George Mavritsakis [a], Spyros Sioutas [c], Athanasios Tsakalidis [a,b], Kostas Tsichlas [b,d], Christos Zaroliagis [a,b,*]

[a] Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece
[b] Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26500 Patras, Greece
[c] Department of Informatics, Ionian University, 49100 Corfu, Greece
[d] Department of Informatics, Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 1 June 2009
Accepted 27 June 2010
Available online 12 August 2010

Keywords:
B-tree
External memory data structure
Interpolation search
Data indexing

ABSTRACT

We present the *interpolation search B-tree* (*ISB-tree*), a new cache-aware indexing scheme that supports update operations (insertions and deletions) in $O(1)$ worst-case block transfers and search operations in $O(\log_B \log n)$ expected block transfers, where B represents the disk block size and n denotes the number of stored elements. The expected search bound holds with high probability for a large class of (unknown) input distributions. The worst-case search bound of our indexing scheme is $O(\log_B n)$ block transfers. Our update and expected search bounds constitute a considerable improvement over the $O(\log_B n)$ worst-case block transfer bounds for search and update operations achieved by the B-tree and its numerous variants. This is also verified by an accompanying experimental study.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

More than three decades after its invention, B-tree [6,8] and its variants remain the ubiquitous external memory data structure for indexing and organizing large data sets with numerous applications, especially in database systems. Its popularity is mainly due to the stable and guaranteed performance for search and update (insertion and deletion) operations, which both cost $O(\log_B n)$ block transfers in the worst-case, with B and n representing the number of elements in a disk block and the number of stored elements, respectively. The most heavily used application is the efficient answering of one-dimensional range search queries using $O(\log_B n + r)$ block transfers, where $R = rB$ is the number of elements reported. In such a query, the elements in a range $[z_1, z_2]$ can be found by first searching the B-tree for z_1 and then performing an in-order traversal in the tree from z_1 to z_2. These bounds hold for any cache-aware disk-access model, that is, a model that accounts memory transfers in disk blocks of known sizes, as these transfers are the dominating operation w.r.t. time. In this paper, we consider one of the most known and widely used such models, namely the two-level memory hierarchy model introduced in [2,36]. In this model, the memory hierarchy consists of an internal (main) memory and an arbitrarily large external memory (disk) partitioned into blocks of size B. The data from the external to the main memory and vice versa are transferred in blocks (one block at a time).

☆ This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contracts No. IST-2002-001907 (integrated project DELIS) and No. FP6-021235-2 (project ARRIVAL), and the Action PYTHAGORAS of the Operational Programme for Educational & Vocational Training II, with matching funds from the European Social Fund and the Greek Ministry of Education.

* Corresponding author at: Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece.
E-mail addresses: kaporis@ceid.upatras.gr (A. Kaporis), makri@ceid.upatras.gr (C. Makris), mayritsa@ceid.upatras.gr (G. Mavritsakis), sioutas@ionio.gr (S. Sioutas), tsak@ceid.upatras.gr (A. Tsakalidis), tsihlas@ceid.upatras.gr (K. Tsichlas), zaro@ceid.upatras.gr (C. Zaroliagis).

1570-8667/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2010.08.001

A vast number of variants of the B-tree have been proposed since its appearance in order to improve its performance in practice for various applications — B^+-trees [8], B^*-trees [8,16], SB-trees [26], weight-balanced B-trees [4], level-balanced B-trees [1], and others [5,22,29–31]; see the excellent survey by Vitter [35] for an extended accounting of these and other variants and their applications — to make it parallel for use in multi-disk environments [32], to tune it for concurrency and recovery purposes [18,33], to extend it to cover other than the original field [11], etc. However, to the best of our knowledge, the aforementioned search and update bounds of B-tree and its variants remained untouched all these years. The same applies to the one-dimensional range search query bound, although some variants (with B^+-tree being the most popular) offer a slightly different search procedure, where the leaves are linked together and hence allow for sequential access. Regarding the update operation, it should be noted that it consists of three consecutive phases: a search phase (to locate the place of the update), an element-updating phase (to insert the new element, or delete the located element), and a rebalancing phase (to restore the B-tree structure). Excluding the first phase (search operation), the dominating phase of an update operation is the rebalancing one, since the element-updating phase takes typically $O(1)$ block transfers (and/or time). In the case of B-tree and its variants, the rebalancing phase requires $\Theta(\log_B n)$ block transfers in the worst-case. This implies that the update operation takes $\Theta(\log_B n)$ block transfers, even in the case where the update position (block within which the update will take place) is given. Note that there are certain applications (see e.g., [9,19]) which justify the exclusion of the search phase in an update operation: once the requested element has been found, then the next element to be searched is located "near by" and hence a new search is redundant.

In this work, we present a new indexing scheme, called *ISB-tree* (Interpolation Search B-tree), that supports search operations in $O(\log_B \log n)$ expected block transfers *with high probability* (w.h.p.) for a large class of input distributions (including both uniform and non-uniform classes) described below, and update operations in $O(1)$ block transfers, provided that the update position is given. The search bound implies that a one-dimensional range search query can be supported in $O(\log_B \log n + r)$ expected block transfers with high probability. The worst-case block transfers for the search operation in our indexing scheme are $O(\log_B n)$.

To achieve our expected search bound we consider a rather general scenario of μ-*random* insertions and *random* deletions, where μ is a so-called *smooth* probability density [3,25]. An insertion is μ-random if the key to be inserted is drawn randomly with density function μ; a deletion is random if every key present in the data structure is equally likely to be deleted [17]. In our update scenario, we slightly favour insertions over deletions to avoid deleting all elements; i.e., the next operation in the update sequence is a μ-random insertion with probability $p > 1/2$ and a random deletion with probability $1 - p$. Informally, a distribution defined over an interval I is *smooth* if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks). Smooth distributions are a superset of uniform, bounded, and several non-uniform distributions (e.g., the class of regular distributions introduced by Willard [37]).

Our indexing scheme is a two-level data structure. The upper level is a non-straightforward externalization of the static interpolation search tree presented in [14]. The lower level is a forest of buckets, each of which is implemented by a new variant of the classical B-tree, the *Lazy B-tree*, which we introduce here. The lazy B-tree, which may be of independent interest, supports the search operation in $O(\log_B n)$ block transfers and an update operation in $O(1)$ block transfers, provided that the update position is given. However, a straightforward combination of the above structures does not necessarily lead to better bounds, since:

(i) the number of elements within a bucket may grow arbitrarily large, as insertions are performed; and
(ii) we strive for creating a *robust indexing scheme*, that is, a data structure that works correctly *without* a priori knowledge of the particular smooth distribution μ.

To overcome these problems, we employ a combinatorial game of bins and balls introduced in [14] that allows to upper bound the number of elements in a bucket, and to approximate an unknown distribution by an almost uniform one.

To the best of our knowledge, this is the first work that uses the dynamic interpolation search paradigm in the framework of indexing data in external memory.

External data structures related to our approach are those based on hashing [16,23,35]. The main representatives of external memory hashing methods include: extendible hashing [10], linear hashing [20], and external perfect hashing [12]. These hashing schemes and their variants need $O(1)$ expected block transfers for answering search queries, but they share various disadvantages when compared to our structure:

(i) they do not support range queries;
(ii) their expected case analysis usually assumes *uniform* input distributions (or input distributions that produce uniform hash key values); and
(iii) they exhibit poor worst case performance.

An effort was made to alleviate these disadvantages by combining B-trees with hashing in a two level data structure scheme called *bounded disorder* [21]. The bounded disorder scheme has time complexity dominated by that of the B-tree, as well as by the size of the data nodes which are structured as hash tables, and thus suffers from the same weaknesses as hash tables and B-trees when compared to our structure.

The remainder of the paper is organized as follows. In Section 2, we discuss preliminary notions and results that are used throughout the paper, and define formally the class of smooth probability distributions. In Section 3, we discuss the *Lazy B-tree*. The main result of this paper, the *ISB-tree*, with the complexity analysis of its operations is discussed in Section 4. Section 5 provides an experimental evaluation with synthetic and real data of our theoretical findings. We conclude in Section 6. A preliminary version of this work appeared in [15].

2. Preliminaries

In this section, we briefly review B-trees and the static interpolation search tree, and formally define the class of non-uniform input distributions that we consider.

2.1. The B-tree

The *B-tree* is a $\Theta(B)$-ary tree (with the root possibly having smaller degree) built on top of $\Theta(n/B)$ leaves. The degree of internal nodes, as well as the number of elements in a leaf, is typically kept in the range $[B/2, B]$ such that a node or leaf can be stored in one disk block. All leaves are on the same level and the tree has height $O(\log_B n)$. This guarantees that a search operation can be accomplished within $O(\log_B n)$ block transfers. An insertion is performed in $O(\log_B n)$ block transfers by first searching down the tree for the relevant leaf l. If there is room for the new element in l, then we simply store it there. Otherwise, we split l into two leaves l' and l'' of approximately the same size and insert the new element in the relevant leaf. The split of l results in the insertion of a new routing element in the parent of l, and thus the split may propagate up the tree. Propagation of splits can often be avoided by sharing some of the (routing) elements of the full node with a non-full sibling. A new (degree 2) root is produced when the root splits and the height of the tree grows by one. Similarly, a deletion can be performed in $O(\log_B n)$ block transfers by first searching down the tree for the relevant leaf l and then removing the deleted element. If this results in l containing too few elements, then we either fuse it with one of its siblings (corresponding to deleting l and inserting its elements in a sibling), or we perform a share operation by moving elements from a sibling to l. Fuse operations may also propagate up the tree and eventually result in the height of the tree decreasing by one.

2.2. Input distributions

One of the first works, in the context of internal memory data structures, that investigated non-uniform distributions regarding insertions in an update sequence was that of Willard [37], who introduced the so-called *regular* distributions. A probability density μ is regular if there are constants b_1, b_2, b_3, b_4 such that $\mu(x) = 0$ for $x < b_1$ or $x > b_2$, and $\mu(x) \geqslant b_3 > 0$ and $|\mu'(x)| \leqslant b_4$ for $b_1 \leqslant x \leqslant b_2$. This has been further pursued by Mehlhorn and Tsakalidis [25], who introduced the *smooth* input distributions, a notion that was further generalized and refined in [3]. Given two functions f_1 and f_2, a density function $\mu = \mu[a, b](x)$ is (f_1, f_2)-*smooth* [3] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leqslant c_1 < c_2 < c_3 \leqslant b$, and all integers n, it holds that

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)\, dx \leqslant \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leqslant x \leqslant c_3$ where $p = \int_{c_1}^{c_3} \mu(x)\, dx$.

Intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta \cdot f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2)-smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [3,14]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

2.3. The static interpolation search tree

The *static interpolation search tree* is a searching data structure presented in [14]. It is a static and explicit version of the search trees used in [3,25] that address the classical searching problem, which for our purposes is defined as follows. Consider a random file $S = \{X_1, \ldots, X_n\}$, where each key $X_i \in [a, b] \subset \mathbb{R}$, $1 \leqslant i \leqslant n$, obeys an unknown distribution μ. Let $P = \{X_{(1)}, \ldots, X_{(n)}\}$ be an increasing ordering of S. The goal is to find the largest key $X_{(j)} \in P$ that precedes a *target* element x.

A static interpolation search tree (SIST) corresponding to P stores the elements of P in its leaves and can be fully characterized by three functions $H(n)$, $R(n)$ and $I(n)$, which are non-decreasing and invertible with a second derivative

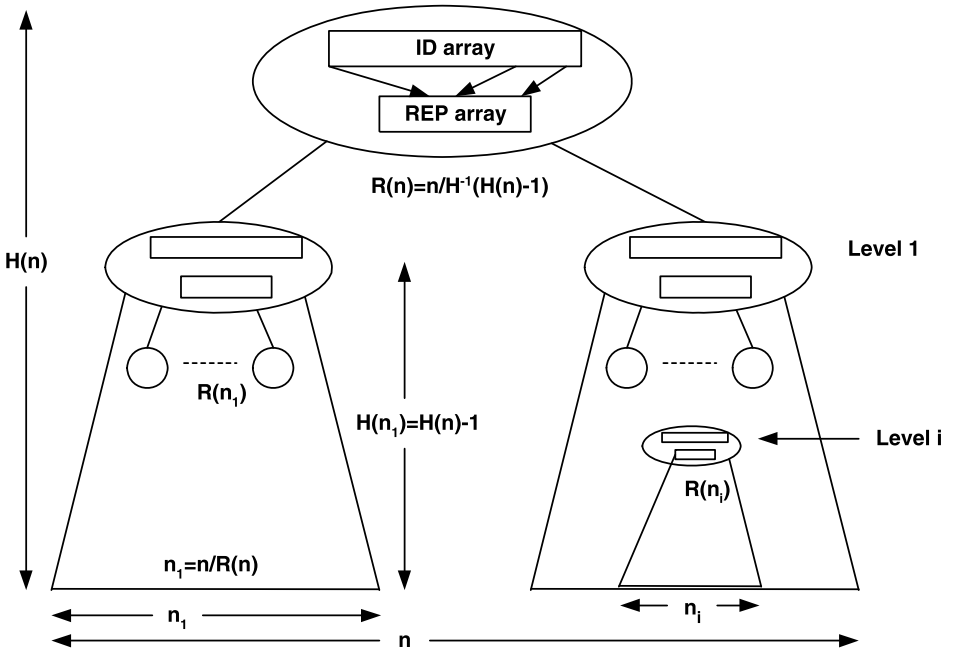

Fig. 1. Static Interpolation Search Tree (SIST).

less than or equal to zero. $H(n)$ denotes the height of the tree, $R(n)$ denotes the out-degree of the root, and $I(n)$ denotes how fine is the partition of the set of elements. The root node of SIST corresponds to the ordered file P of size n (see Fig. 1). Each child (node at depth 1) corresponds to a part of P of size $n_1 = \frac{n}{R(n)}$, i.e., the subtree rooted at each child of the root has $n_1 = n/R(n)$ elements of P (stored at its leaves) and height $H(n_1) = H(n/R(n)) = H(n) - 1$. In other words, achieving a height of $H(n)$ dictates that $R(n) = n/H^{-1}(H(n) - 1)$. Moreover, $H(n)$ should be $o(\log n)$ and not $O(1)$, and $H^{-1}(i) \neq 0$, for $1 \leqslant i \leqslant H(n) - 1$. In order to handle an as large as possible class of distributions μ, the approximation of the sample density should be as fine as possible, implying that $I(n)$ should be as large as possible. Since $I(n)$ affects space, it is chosen as $I(n) = n \cdot g(H(n))$, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, so that the space of SIST remains linear. In general, consider an internal node v at depth $i \geqslant 0$ ($n_0 = n$) and assume that n_i elements of P are stored in the subtree rooted at v, whose keys take values in $[\ell, u]$. Then, v has $R(n_i)$ children, each one corresponding to a subfile of P of size $n_{i+1} = n_i/R(n_i)$. It can be easily verified that $H(n_i) = H(n) - i$, which implies that $n_i = H^{-1}(H(n) - i)$. Since $n_{i+1} = n_i/R(n_i)$, we have that $H(n_i/R(n_i)) = H(n_{i+1}) = H(n_i) - 1 = H(n) - i - 1$ implying that $n_i/R(n_i) = H^{-1}(H(n) - i - 1)$ or that v has degree $R(n_i) = H^{-1}(H(n) - i)/H^{-1}(H(n) - i - 1)$. Moreover, $I(n_i) = n_i \cdot g(n_i) = n_i \cdot g(H(n) - i)$.

Each internal node v of the tree at depth i is associated with an array REP$[1..R(n_i)]$ of sample elements, containing one sample element for each of its subtrees, and an array ID$[1..I(n_i)]$ that stores a set of sample elements approximating the inverse distribution function. The role of the ID array is to partition the interval $[\ell, u]$ into $I(n_i)$ equal parts, each of length $\frac{u-\ell}{I(n_i)}$. The role of the REP array is to partition its associated ordered subfile of P into $R(n_i)$ equal subfiles, each of size $\frac{n_i}{R(n_i)}$. For each node, parent and child pointers are explicitly maintained. The required pointer information can be easily incorporated in the construction of the static interpolation search tree.

Using the ID array, we can interpolate the REP array to determine the subtree in which the search procedure will continue. In particular, for the ID$[1..I(n_i)]$ array associated with node v, it can be easily verified [3,25] that ID$[i] = j$ iff REP$[j] < \ell + i(u - \ell)/I(n_i) \leqslant$ REP$[j + 1]$. In other words, the ID array helps us to interpolate REP by establishing the appropriate offset $j + 1$ in the REP array from which we can start our search to locate faster the particular subtree of v that we will continue our searching procedure.

The above suggests the following searching procedure. Let x be the element we seek. Starting from the root, in each node v of SIST at depth i, compute the index $j = \text{ID}[\lfloor (I(n_i)(x - \ell)/(u - \ell)) \rfloor]$ to interpolate the associated with v REP array, and then search (sequentially) the REP array from REP$[j + 1]$ until you find the appropriate child to continue the search, that is, until you find an index t such that REP$[j + 1 + t] < x \leqslant$ REP$[j + 2 + t]$. If the elements are drawn from a smooth distribution, then $t = O(1)$ (with high probability; see [14,25]).

The aforementioned choice of functions $H(n)$, $R(n)$ and $I(n)$ ensures that an SIST on n elements, drawn from an $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth distribution μ, can be built in $O(n)$ time and space [14]. In the case where $R(n) = n^{\delta}$, $0 < \delta < 1$, and $g(x) = x^{-(1+\varepsilon)}$, $\varepsilon > 0$ – which implies that $H(n) = \Theta(\log\log n)$ and $I(n) = n/(\log\log n)^{(1+\varepsilon)}$ – we get the $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth distribution, which is the largest known such class in the hierarchy defined by the smooth distributions [3].

Throughout the paper, we say that an event E occurs *with high probability* (w.h.p.) if $\Pr[E] = 1 - o(1)$.

3. The Lazy B-tree

In this section we present a new variant of the B-tree, which we call the *Lazy B-tree*. The *Lazy B-tree* is based on a simple but non-trivial externalization of the techniques introduced in [28]. It is another typical case of a two-level data structure. The first level consists of an ordinary B-tree, while the second level consists of buckets of size $O(\log^2 n)$, where n is approximately equal to the number of elements stored in the data structure. The rebalancing operations are guided by the *global rebalancing lemma* given in [28] (see also [9,19]). In this scheme, each bucket is assigned a *criticality* which is a variable number indicating how close this bucket is to be fused or split. Every time we update a bucket we increase its criticality by one. Every $O(\log_B n)$ updates we choose the bucket with the largest criticality and make a rebalancing operation (fusion or split). The update of the lazy B-tree is performed incrementally (i.e., in a step-by-step manner) during the next $O(\log_B n)$ update operations and until the next rebalancing operation. The *global rebalancing lemma* ensures that the size of the buckets will never be larger than $O(\log^2 n)$.

To realize the lazy B-tree, the following problems must be tackled:

(1) the representation of the criticalities of the buckets; and
(2) the representation of each bucket.

Our solution is based on a simple externalization of dynamic sorted linked lists. Such a list E is a sorted linked list storing a number, say k, of elements and supporting the following operations:

- *Search*(q, E): find and return the element q in E if it exists; otherwise its predecessor.
- *Insert*(q, E): insert in E the new item q maintaining the order of elements.
- *Delete*(q, E): delete the existing element q from E.

List E consists of blocks of size B. The search operation is implemented by scanning sequentially all blocks of E until the element is found, in $O(\frac{k}{B})$ block transfers. Operations *Insert* and *Delete* are implemented by first locating the corresponding position and then by performing the update operation.

Assume that the update operation is performed on block b. Consider first the case of an insertion. If b is not full, then the new element is inserted in b. If b is full, then a new block b' is constructed and half of the elements of b are moved to b'. Clearly, the number of block transfers for the insert operation, not taking into account the complexity of the search operation, is $O(1)$ in the worst-case. Now, consider the case of a deletion of an element q. First, q is removed from block b. If the size of b is less than $\frac{B}{2}$, then it is fused with an adjacent block b' given that their combined size is no more than B, otherwise some elements are transferred from b' to b.

In the following, we describe how we can construct a secondary data structure that allows to find the most critical bucket in $O(1)$ worst-case number of block transfers as well as how to represent the buckets, using dynamic external sorted linked lists.

3.1. Maintaining the criticalities of buckets

Consider a set S of k objects and assume that each object is assigned a value between 0 and $\log^2 k$ (throughout the paper with $\log k$ we will denote $\lceil \log k \rceil$). In this section, we show how this set can be maintained such that insertions or deletions of objects are accomplished within $O(1)$ block transfers. In addition, the value of an object can be increased or decreased by 1 in $O(1)$ block transfers, and the object with the largest element is located and removed in $O(1)$ block transfers. All bounds are worst-case.

The set S is partitioned into $\log k$ subsets represented by a list L. The i-th element of L, $1 \leqslant i \leqslant \log k$, stores all objects with value in the range $[(i-1)\cdot \log k, i \cdot \log k)$. The i-th element of L is represented as a sublist L_i whose j-th position, denoted as $L_{i,j}$, corresponds to all objects with value equal to $(i-1)\cdot \log k + j$. Finally, each $L_{i,j}$ is represented again as a list since there may be many objects with exactly the same value. The m-th element stored in list $L_{i,j}$ is denoted as $L_{i,j,m}$, $1 \leqslant m \leqslant k$. Fig. 2 illustrates this 3-level structure.

Lists L, L_i, and $L_{i,j}$, for $1 \leqslant i, j \leqslant \log k$, are implemented as dynamic external sorted linked lists. Actually, we do not store the objects themselves in these lists, but just their values; there is a pointer from every object to its corresponding element in a list whose content is the object's value.

The 3-level structure supports the following operations:

1. *Add*(o, L): inserts the value of an object o in structure L. The addition is always made in $L_{1,1}$.
2. *Remove*(o, L): removes the value of an existing object o from structure L. The position of object o is given.
3. *Inc*(o, L): increases the value of object o by 1 in structure L. The position of object o is given.
4. *Dec*(o, L): decreases the value of object o by 1 in structure L. The position of object o is given.
5. *RemoveMax*(L): returns and deletes the object with the largest value in structure L.

We now turn to the implementation of these operations.

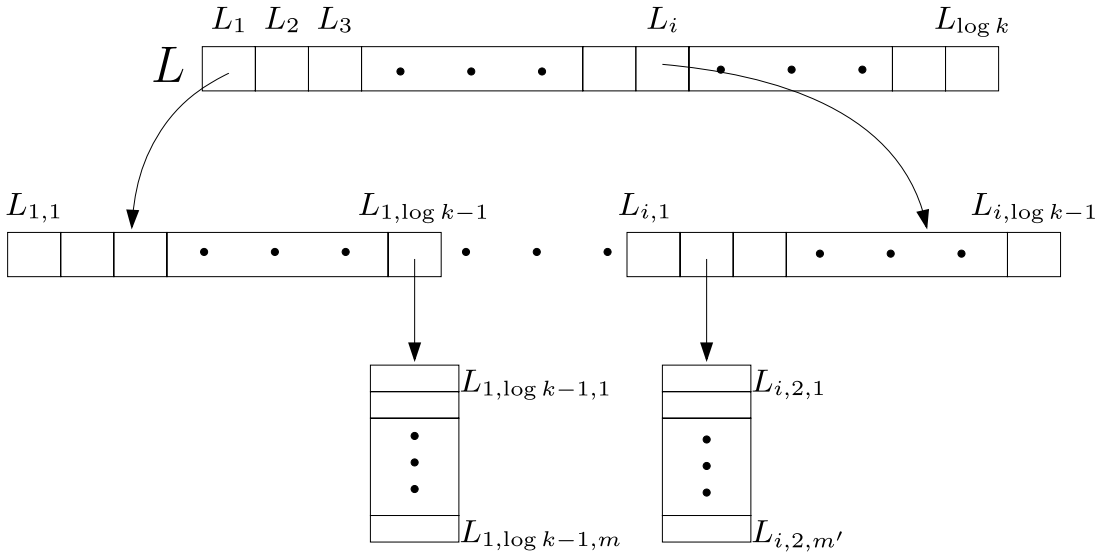

Fig. 2. The 3-level structure of lists for maintaining criticalities; m and m' may be as large as k.

Since the position of an object and (as a consequence) the corresponding list position, say $L_{i,j}$, is given, the *Remove*(o, L) operation can be easily performed in $O(1)$ block transfers. In a similar way, the *Add*(o, L) operation requires $O(1)$ block transfers, since we can initially allocate a pointer from the object o to the list position $L_{1,1}$ and vice versa. Assuming that (the value of) object o lies in list $L_{i,j}$, operation *Inc*(o, L) (operation *Dec*(o, L) has a similar function) is easily performed by moving (the value of) o from $L_{i,j}$ to $L_{i,j+1}$, if $j < \log k$. If $j = \log k$, then (the value of) o is moved to $L_{i+1,1}$. Since the insertion to a list $L_{i,j}$ is performed using $O(1)$ block transfers by inserting it into its head block, the operation of increment by one is also performed within the same bound. Finally, operation *RemoveMax*(L) is also performed using $O(1)$ block transfers by maintaining a pointer to the list $L_{\log k,j}$ for the maximum existing j and taking the first element of this list. Since the lists are always non-empty, it is trivial to update this pointer. The actual deletion of the (value of the) object is also easy to implement in $O(1)$ block transfers.

The above discussion establishes the following lemma.

Lemma 1. *Operations* Add(o, L), Remove(o, L), Inc(o, L), *and* RemoveMax(L) *of the* 3*-level structure can be performed in* $O(1)$ *block transfers.*

3.2. Maintaining buckets

The *Lazy B-tree* is a two-level data structure. The upper level of the structure is an ordinary leaf-oriented B-tree, while the lower level is a set of buckets associated with the leaves of the tree that store all elements. As a result, the part of the data structure that must be analyzed is the lower level. Since buckets may have up to $O(\log^2 n)$ elements, we implement them as a two-layered structure consisting of lists of size $\log n$. The top layer inside a bucket is a list with size at most $\log n$ that guides the search to the lists of the bottom layer. The lists of the bottom layer inside a bucket store at most $\log n$ elements. These lists are implemented as dynamic external sorted linked lists.

To facilitate update operations we define the *fullness* $\Phi(B_i)$ of a bucket B_i as $\Phi(B_i) = \frac{|B_i|}{\log^2 n}$. The update algorithm will ensure that $0.5 \leqslant \Phi(B_i) \leqslant 2$.[1] We also define the *criticality* of a bucket B_i as $\gamma(B_i, n) = \frac{1}{\alpha \log n} \max\{0.7 \log^2 n - |B_i|, |B_i| - 1.8 \log^2 n\}$, for an appropriately chosen constant α. (The values 0.7 and 1.8 are chosen so as to determine how close $|B_i|$ is to its two extreme values of $0.5 \log^2 n$ and $2 \log^2 n$.) A bucket B_i is called *critical* if $\gamma(B_i, n) > 0$.

To maintain the criticalities, the above structure is associated with the 3-level structure of Section 3.1. The elements of S are the buckets $B_i, 1 \leqslant i \leqslant m$, of the ordinary B-tree. Hence, the number of elements in S will be $m = O(\frac{n}{\log^2 n})$. In addition, each bucket B_i maintains a pointer $BTL(B_i)$ to its corresponding element x_i in list $L_{i,j}$, provided that the criticality of B_i is $(i-1)\log m + j$. Accordingly, element x_i maintains a pointer $LTB(x_i)$ to bucket B_i. Fig. 3 illustrates the structure of the lazy B-tree.

The following theorem provides the complexities of the Lazy B-tree.

[1] These values are chosen in order to satisfy the global rebalancing lemma [28].

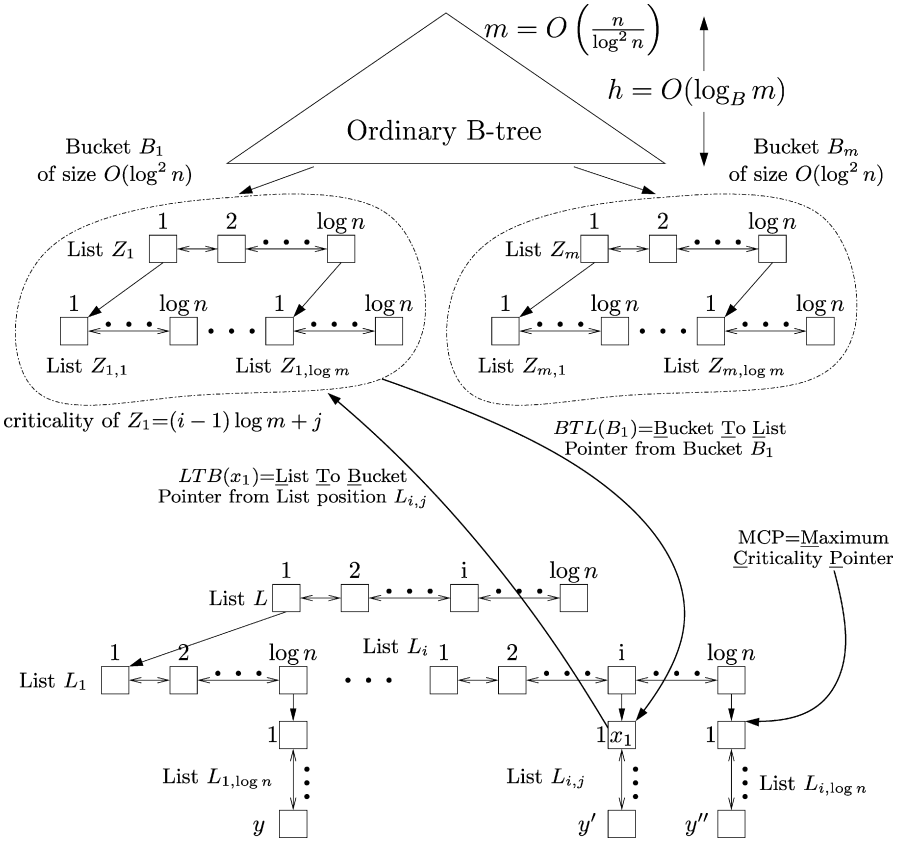

Fig. 3. The Lazy B-tree.

Theorem 1. *The Lazy B-tree supports search operations in $O(\log_B n)$ worst-case block transfers and update operations in $O(1)$ worst-case block transfers, provided that the update position is given.*

Proof. The complexity for a search operation is decomposed into the following steps: (1) the search operation in the B-tree, which is performed in $O(\log_B n)$ block transfers; and (2) the search inside the buckets, which consists of a linear scan of the list of the top layer and of the corresponding list of the bottom layer. Clearly, this procedure can be accomplished in $O(\frac{\log n}{B})$ block transfers. Hence, the search operation is performed in an overall $O(\log_B n + \frac{\log n}{B}) = O(\log_B n)$ block transfers.

We now turn to the update operations. Assume that the update position is in bucket B_i whose list in the top layer is Z_i. Assume also that the update position is in (the dynamically external sorted linked) list $Z_{i,j}$ of the bottom layer of the lists in the bucket. Finally, assume that the update position is in block j and that the respective element in the list L that stores the bucket's criticality is x_i. Blocks and lists must be dynamic in the sense that they must support the fundamental operations of split, fuse and transfer.

If the update operation is the insertion of a new element q, then operation *Insert*$(q, Z_{i,j})$ is invoked. If block j is full, then this block is split and written on two new blocks in the disk. If the update operation is the deletion of element q, then the operation *Delete*$(q, Z_{i,j})$ is invoked. If block j is underfull, then a fuse or a share operation with an adjacent block must be performed. In any case $O(1)$ number of block transfers are necessary. Additionally, every time we update bucket B_i, to which the block j belongs, we must compute its new criticality. If this criticality has been increased by one, then we must additionally perform the *Inc*(x_i, L) operation (similarly, if the criticality has been decreased by one we perform *Dec*(x_i, L)). Note that the position of x_i is denoted by the pointer $BTL(B_i)$ which is maintained by bucket B_i. If B_i had zero criticality and after an insertion its criticality becomes 1, then operation *Add*(B_i, L) is invoked. If the bucket B_i participates in a fuse operation with another bucket B_{i+1} producing the bucket B_{i+1}, then we have to perform operation *Remove*(B_i, L). In all these operations, the corresponding representative x_i in L is provided by the pointer $BTL(B_i)$.

The aforementioned bucket operations of fuse and split can be performed with $O(1)$ block transfers, only if the two-layer lists within a bucket have been properly preprocessed. This implies that incremental steps have to be performed in the lists $Z_{i,j}$ and Z_i during a sequence of update operations. If after an update operation $Z_{i,j}$ has $\frac{\log n}{B} + 1$ blocks, then the last block is moved to a new, under construction, list $Z'_{i,j}$. When both lists have exactly $\log n/B$ blocks, then a new list $Z'_{i,j}$ is constructed. The same procedure applies for the list Z_i. This allows us to make splits of buckets in $O(1)$ block transfers. Fusions are tackled in a similar manner.

Every $\alpha \log_B n$ updates, the most critical bucket is selected (we make use of the $LTB(x_i)$ pointer to B_i). Let this bucket be B_i (provided that L is non-empty), and let its list in the top layer be Z_i. If B_i has non-zero criticality, then the following rebalancing transformations are applied.

1. If $\Phi(B_i) > 1.8$, then *split* the bucket B_i into two parts of approximately equal size, B_i and B_i' each of which has zero criticality.
2. If $\Phi(B_i) < 0.7$ and one of its adjacent buckets (let this be B_{i+1}) has $\Phi(B_{i+1}) \geqslant 1$, then *transfer* elements from B_{i+1} to B_i. Both buckets will have zero criticality after the transfer.
3. If $\Phi(B_i) < 0.7$ and transferring is not possible, then *fuse* with the adjacent bucket B_{i+1}. The bucket after the fuse will have zero criticality.
4. Run the *RemoveMax*(L) operation.

When the *RemoveMax*(L) operation is performed, the corresponding critical bucket becomes non-critical as a result of the application of the rebalancing operations. In addition to the block transfers required to split/fuse buckets, a bucket rebalancing step may require $O(\log_B n)$ block transfers to insert/delete a bucket representative to/from the upper level B-tree. Since the total work to rebalance a bucket is $O(\log_B n)$, we can perform it with $O(1)$ work per update spread over to no more than $\alpha \log_B n$ updates. In [9,19,28] it was proved that the described update procedure guarantees that no bucket will get more than $\Theta(\log^2 \hat{n})$ elements, where $\hat{n}$ is the number of current elements. In other words, if we ensure that each bucket is of that size, then we can guarantee that between two rebalancing operations in the upper level tree no other rebalancing can occur and consequently the incremental spread of work is possible. □

4. The data structure

For ease of exposition we divide this section into two parts. In the first part the main components of the structure are provided, while in the second part the time and space complexity of the various operations are analyzed.

4.1. The ISB-tree

The ISB-tree is a two-level data structure. The lower level is a set of buckets each of which contains a subset of the stored elements. Each bucket is implemented as a Lazy B-tree and is represented by a unique representative. The representatives of the buckets as well as some additional elements are stored in the upper level structure.

The upper level data structure is an external version T of the static interpolation search tree (SIST) [14] (recall its definition from Section 2), with parameters $R(n) = n^{\delta}$, $I(n) = n/(\log\log n)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, and n is the number of stored elements in the tree. That is, the elements are inserted according to an $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth distribution.

The specific choice of δ guarantees the desirable $O(\log_B \log n)$ height of the upper level structure. For each node that stores more than $B^{1+\frac{1}{B-1}}$ elements in its subtree, we represent its REP and ID arrays as static external sorted arrays; otherwise, we store all the elements in a constant number of disk blocks. In particular, let v be a node and n_v be the number of stored elements in its subtree, with $n_v \geqslant B^{1+\frac{1}{B-1}}$. Node v is associated with two external arrays EREP_v and EID_v. The EID_v array uses $O(\frac{I(n_v)}{B})$ contiguous blocks, the EREP_v array uses $O(\frac{R(n_v)}{B})$ contiguous blocks, while an arbitrary element of the arrays can be accessed with $O(1)$ block transfers, given its index. Moreover, the choice of the parameter $B^{1+\frac{1}{B-1}}$ guarantees that each of the EREP_v and EID_v arrays contains at least B elements, and hence we do not waste space (in terms of underfull blocks) in the external memory representation. The upper level, which we shall refer to as *external SIST*, is illustrated in Fig. 4.

The lower level structure is a set of ρ buckets, each one implemented as a Lazy B-tree. Let S_0 be the set of elements to be stored, where the elements take values in $[a, b]$. Each bucket $\mathcal{B}_i$, $1 \leqslant i \leqslant \rho$, stores a subset of elements and is represented by the element $rep(i) = \max\{x\colon\ x \in \mathcal{B}_i\}$. The set of elements stored in the buckets constitute an ordered collection $\mathcal{B}_1, \ldots, \mathcal{B}_\rho$ such that $\max\{x\colon\ x \in \mathcal{B}_i\} < \min\{y\colon\ y \in \mathcal{B}_{i+1}\}$ for all $1 \leqslant i < \rho - 1$. In other words, $\mathcal{B}_i = \{x\colon\ x \in (rep(i-1), rep(i)]\}$, for $2 \leqslant i \leqslant \rho$, and $\mathcal{B}_1 = \{x\colon\ x \in [rep(0), rep(1)]\}$, where $rep(0) = a$ and $rep(\rho) = b$.

The ISB-tree is maintained by incrementally performing global reconstructions [27]. Let S_0 be the set of stored elements at the latest reconstruction, and assume that $S_0 = \{x_1, \ldots, x_{n_0}\}$ in sorted order. During the reconstruction the set $S_1 = \{x_{i \cdot \ln n_0}\colon\ i = 1, \ldots, \frac{n_0}{\tau} - 1\} \cup \{b\}$, $\tau = \max\{B, \ln n_0\}$ is defined. The i-th element of S_1 is the representative $rep(i)$ of the i-th bucket $\mathcal{B}_i$, where $1 \leqslant i \leqslant \rho$ and $\rho = |S_1| = \frac{n_0}{\tau}$. An element $x \in S_0$ is stored twice:

1. As a leaf of the external SIST containing x.
2. In $\mathcal{B}_i$, iff $rep(i-1) < x \leqslant rep(i)$, for $i \in \{2, \ldots, \frac{n_0}{\tau}\}$; otherwise ($x \leqslant rep(1)$), x is stored in $\mathcal{B}_1$.

Each leaf in the external SIST maintains a pointer to the bucket in which it belongs. Additionally, each bucket is equipped with a pointer to the leaf containing its representative.

This redundancy which comes from storing elements in buckets as well as in the leaves of the SIST during the reconstruction, may seem curious, but it has a critical role in the analysis of the expected performance of the external SIST. First,

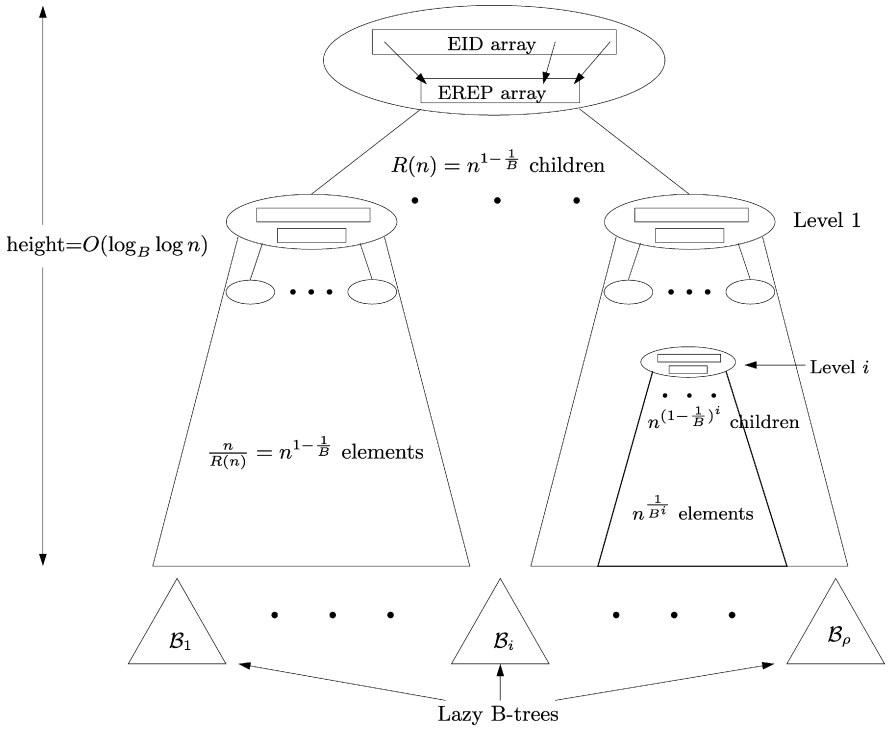

Fig. 4. The ISB-tree on n elements.

the elements of S_0 are stored in the bottom level (buckets) to guarantee that it is highly unlikely that a bucket will become empty due to random deletions (Section 4.2). Second, the elements of S_0 are stored in the top level of the external SIST to guarantee the expected performance of the search procedure in a manner similar to the interpolation search trees presented in [3,25]. This is captured by the following lemma.

Lemma 2. *Let T be an external SIST on a set S of n elements generated by a μ-random distribution and let T' be any subtree of T which spans a subset $S' \subset S$. Then, the elements of S' are also μ-randomly distributed.*

Proof. Similar to the proof of Lemma 4 in [25]. □

We now move to the description of the operations supported by the ISB tree. In order to insert/delete an element, given the position (block) of the update, we simply have to insert/delete the element to/from the *Lazy B-tree* storing the elements of the corresponding bucket. Note that the external SIST is not affected by these updates. This means that there are leaves which do not correspond to elements in buckets while at the same time there are elements which are not stored in the leaves of the external SIST. Each time the number of updates exceeds cn_0, where $0 < c < 1$, the whole data structure is reconstructed. Let n be the number of stored elements at this time. After the reconstruction, the number of buckets is equal to $\lceil \frac{n}{\ln n} \rceil$.

The search procedure for locating a query element x can be decomposed into two phases:

(i) the traversal of internal nodes of the external SIST in order to locate a bucket $\mathcal{B}_i$, and
(ii) the search for x in the *Lazy B-tree* storing the elements of $\mathcal{B}_i$.

Phase (i) starts from the root of the external SIST. It checks the external arrays on the root and by interpolating it decides into which child the search procedure will continue. More specifically, let v be a node in the search path for query element x, n_v be the number of leaves in its subtree, and let l_v and u_v be the minimum and the maximum element, resp., stored in the subtree rooted at v. As we have already mentioned, node v is associated with two external arrays EREP_v and EID_v. To interpolate, we compute the value $i = \lfloor \frac{x-l_v}{u_v-l_v} R(n_v) \rfloor$ and find the index $j = \text{EID}_v[i]$, by retrieving the $\lceil \frac{j}{B} \rceil$-th block of the EID_v array. We then scan the blocks of the EREP_v array, starting from its $\lceil \frac{j}{B} \rceil$-th block, until locating an index l such that $\text{EREP}_v[l] \leqslant x < \text{EREP}_v[l+1]$. Then we continue recursively in the same manner in the l-th child of v, until we reach a leaf. By following the respective pointer of this leaf we locate the corresponding bucket $\mathcal{B}_i$. In this case, the search procedure is concluded by entering phase (ii) and by searching further in the *Lazy B-tree* of the bucket $\mathcal{B}_i$.

4.2. Time and space analysis of the ISB-tree

In this section, we analyze the bounds of the search and update operations. Recall that the elements are inserted according to an unknown $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth distribution μ. Our analysis starts with the following lemma.

Lemma 3. *The traversal of internal nodes of the external SIST requires a number of $O(\log_B \log n)$ expected block transfers with high probability.*

Proof. During the searching phase (i), the algorithm visits a path P of h nodes with the last node being a leaf that points to the root of a Lazy B-tree. Let $u_1, \ldots, u_h$ be the nodes in the path listed in order of visit and consider a node u_i in the path. Let n_i the number of leaves (elements) in the subtree rooted at u_i, which, according to Lemma 2, are μ-randomly distributed. In [25, Lemma 7] it was proven that, for the special case where $\delta = 1/2$ there is a constant c such that the probability that the interpolation procedure takes in u_i more than ℓ steps is bounded from above by $(\frac{c}{\ell})^{\ell\sqrt{n_i}}$. Their analysis can be immediately extended in order to prove that for $\delta = 1 - \frac{1}{B}$ there is a constant c such that the probability that the interpolation procedure takes in u_i more than ℓ steps is bounded from above by $(\frac{c}{\ell})^{\ell n_i^{1-\delta}}$. For $\ell = 2c$ the above bound becomes $(\frac{1}{2})^{2cn_i^{1-\delta}}$. Let q be the probability that there is a node in P for which interpolating takes more than $2c$ steps. Then, it follows that $q \leqslant \sum_{i=1}^{h} (\frac{1}{2})^{2cn_i^{1-\delta}} \leqslant h(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Hence, for the probability p that searching phase (i) takes less than $2ch$ steps, we have that $p \geqslant 1 - h(\frac{1}{2})^{2c(\log n)^{1-\delta}}$. Since $h = O(\log_B \log n_0)$, where $n_0 = O(n)$ is the number of stored elements at the latest reconstruction, we get: $h(\frac{1}{2})^{2c(\log n)^{1-\delta}} \to 0$, as n grows, and thus we conclude that the searching phase (i) takes $2ch = O(\log_B \log n)$ expected block transfers with high probability. □

The insertions and deletions of elements into the ISB-tree can be simulated by a combinatorial game of balls and bins described in [14] for an internal finger-search data structure. In particular, balls correspond to elements and bins to buckets. Insertions of elements into the ISB-tree can be simulated by the insertion of balls into bins according to an unknown smooth probability density μ. Similarly, the deletion of an element from the ISB-tree can be simulated by the deletion of an element from a bin uniformly at random. As in [14], the next operation in the update sequence is a μ-random insertion with probability $p > 1/2$ and a random deletion with probability $1 - p$. This guarantees that with high probability a bucket will never become empty due to random deletions. For this process the following has been proven in [14].

Theorem 2. *Consider the aforementioned combinatorial game of n balls and $n/\ln n$ bins, where balls are inserted into bins according to an unknown smooth probability distribution and balls are deleted from a bin uniformly at random. Then, the expected number of balls in a bin is $\Theta(\log n)$ with high probability.*

The following lemma establishes the searching bound within a bucket of the ISB-tree.

Lemma 4. *Searching for an element in a bucket of the ISB-tree takes $O(\log_B \log n)$ expected block transfers with high probability.*

Proof. Immediate from Theorem 1 and the size of each bucket, which is determined by Theorem 2. □

Now, we are ready to prove the main theorem of the paper.

Theorem 3. *Suppose that the upper level of the ISB-tree is an external static interpolation search tree with parameters $R(n_0) = n_0^{\delta}$, $I(n_0) = n_0/(\log\log n_0)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, n_0 is the number of elements in the latest reconstruction, and that the lower level is implemented as a forest of Lazy B-trees. Then, the ISB-tree supports search operations in $O(\log_B \log n)$ expected block transfers with high probability, where $n = \Theta(n_0)$ denotes the current number of elements, and update operations in $O(1)$ worst-case block transfers, if the update position is given. The worst-case update bound is $O(\log_B n)$ block transfers, and the structure occupies $O(n/B)$ blocks.*

Proof. Clearly, from Lemmata 3 and 4, the searching operation takes $O(\log_B \log n)$ expected number of block transfers w.h.p.

Let us now consider the update bound. Between reconstructions the block transfers for an update are clearly $O(1)$, since we only have to update the appropriate Lazy B-tree which can be done in $O(1)$ block transfers (cf. Theorem 1). The reconstructions can be easily handled by using the technique of global rebuilding [19]. With this technique the linear work spent during a global reconstruction of the upper level structure may be spread out on the updates in such a way that a rebuilding cost of $O(1)$ block transfers is spent at each update.

Finally, the worst-case search complexity of $O(\log_B n)$ block transfers can be achieved by using two data structures, an ISB-tree and a Lazy B-tree, and hence storing each element twice. A search for a query element is performed by searching simultaneously both structures and terminating the search when locating for the first time the sought element. The worst-case update and space complexity remain asymptotically unaffected and so the theorem is proven. □

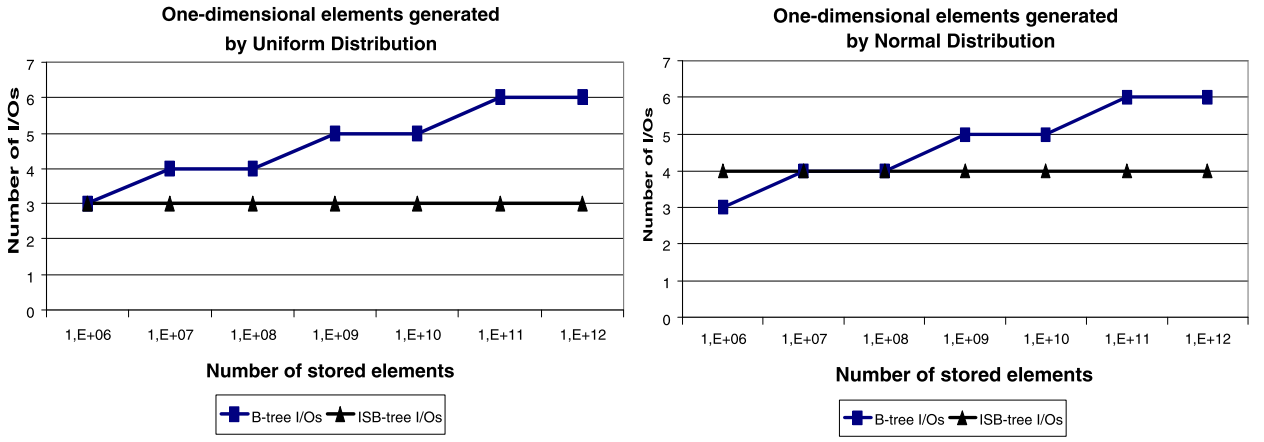

Fig. 5. Search performance for uniform distributions (left) and normal distributions (right).

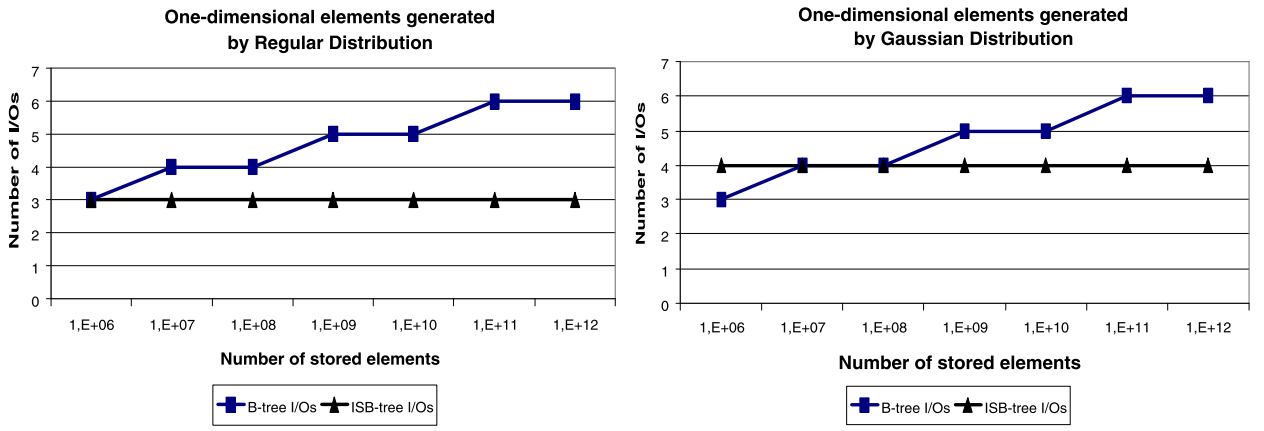

Fig. 6. Search performance for regular distributions (left) and Gaussian distributions (right).

5. Experimental evaluation

In this section, we investigate the practical merits of the ISB-tree. Our prime concern is to (merely) investigate the practical difference of the *asymptotic complexities* (in block transfers) of search and rebalancing operations between the ISB-tree and a cache-aware B-tree. Although there are several cache-aware B-tree variants, all of them exhibit the same asymptotic complexities in block transfers. Consequently, we compare the performance of ISB-tree with a simple variant of the cache-aware B-tree. Moreover, we do not compare the performance of our rebalancing operations (after an update) with hashing schemes and their variants, since the expected-case analysis of such schemes usually assumes *uniform* input distributions (or input distributions that produce uniform hash key values), and hence they exhibit poor worst-case performance for update operations. In our experimental study, we have considered both synthetic and real-world data that are drawn from smooth distributions. Since our prime concern is the evaluation of the performance of the ISB-tree and our data sets concern elements drawn from smooth distributions, there was no reason in implementing the combined structure that keeps in parallel a lazy B-tree to guarantee worst-case performance.

5.1. Synthetic data

We have conducted an experimental study making the customary assumption that the disk page size is 4096 bytes, the length of each key is 8 bytes, and the length of each pointer is 4 bytes. Consequently, each block contains $B = 341$ elements. We considered data sets of size $n_0 \in [10^6, 10^{12}]$ elements generated by a variety of smooth distributions, namely uniform, regular, normal and Gaussian. We compared the implementation of the ISB-tree with that of a B-tree on the same data sets (implementations were carried out in C++ including data types from LEDA [24]). Our main concern was to measure the performance, in simulated block transfers (I/Os), of the search and update operations.

The experimental results regarding the search operations are reported in Fig. 5 and 6. The sequence σ of search operations had length equal to its corresponding data set and the reported values are averages over the whole sequence. Our experiments revealed that the number of block transfers in the ISB-tree remains constant even for gigantic data sets (Ter-

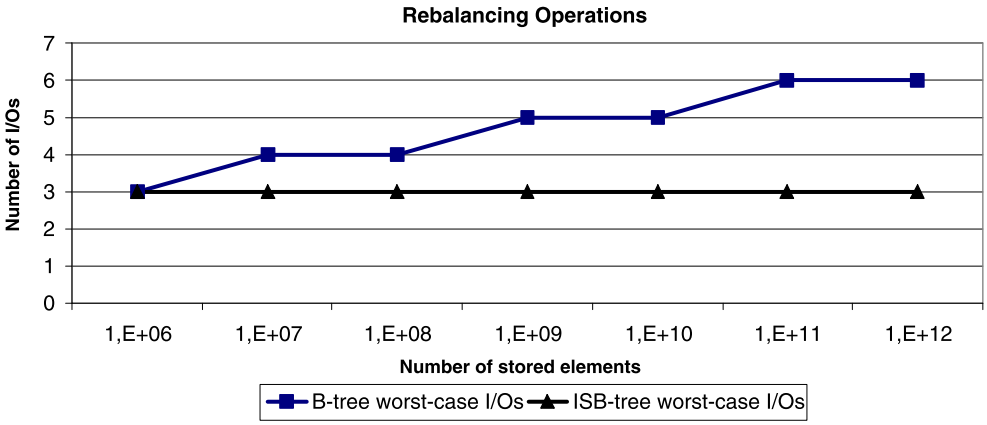

Fig. 7. Block transfers of rebalancing operations after an update.

abytes — TB). Moreover, for data sets larger than 100 GB, the expected number of block transfers is reduced by a factor ranging from 1/3 (for normal and Gaussian distributions) to 1/2 (for uniform and regular distributions) compared with that required by the B-tree. This behaviour can be explained by the theoretical time complexity of the search operation and by the fact that for data sets up to 1 TB and block size of 341 elements, the ISB-tree is a two level structure, where the first level (external SIST structure) consists of only one node equipped with the appropriate EID and EREP arrays, while the second level (lazy B-tree) consists of only one block of elements. Thus, we need 2 block transfers in the first level (one for each array) and 1 block transfer in the second level. Our experiments also show that for uniform and regular distributions, the position of EREP array (which has been located by its corresponding entry in EID) points in almost all cases to the correct subset within which the search has to be continued in the second level. For the case of normal and Gaussian distributions, we often had to move to the immediately next block and this adds one additional block transfer to the search operation. Naturally, for small data sets (smaller than 10 MB), our data structure becomes less efficient than B-trees, due to the overhead of the two-level structure.

Regarding the number of block transfers required for rebalancing after an update operation to the data structure, we again considered the above values of $n_0 \in [10^6, 10^{12}]$ for our initial data sets upon which we performed update sequences of length $n_0/2$ and $2n_0$. The data structure is reconstructed every n_0 operations (i.e., we choose $c = 1$). Our experimental results are reported in Fig. 7. The values represent worst-case block transfers over the update sequence. We observe that the number of rebalancing operations in an ISB-tree is independent of the distribution.

5.2. Real-world spatial data

In this section, we deploy one-dimensional data taken from a real-world spatial dataset "LA rivers and railways" [Tiger1] and "LA streets" [Tiger2], containing 128 971 and 131 461 *M*inimum *B*ounded *R*ectangles (MBRs), respectively; see [34]. The one-dimensional data are taken by the x- and y-projections of MRBs and the values in each axis are normalized in $[0, 10\,000]$. For all experiments, the disk page size is set to 512 bytes, the length of each key to 8 bytes, and the length of each pointer to 4 bytes. Consequently, each block contains $B = 42$ elements. We use a relatively small page size so that the number of nodes in an index simulates realistic situations, where the data set cardinality is higher. A similar methodology was also used in [7].

Fig. 8 and Fig. 9 depict the efficiency of ISB-tree on searching for real spatial one-dimensional data. In particular, in Fig. 8 we measured the number of I/Os required for search operations during the insertion of a total of $2 \times 128\,971 = 257\,942$ and of $2 \times 131\,461 = 262\,922$ x-projections from [Tiger1] and [Tiger2], respectively. Similarly, in Fig. 9 we measured the number of I/Os required for search operations during the insertion of a total of $2 \times 128\,971 = 257\,942$ and of $2 \times 131\,461 = 262\,922$ y-projections from [Tiger1] and [Tiger2], respectively.

Fig. 10 and Fig. 11 depict the efficiency of ISB-tree on updating real spatial one-dimensional data. In particular in Fig. 10 we measured the number of I/Os required for the rebalancing operations during the insertion of a total of $2 \times 128\,971 = 257\,942$ x-projections and of $2 \times 131\,461 = 262\,922$ x-projections from [Tiger1] and [Tiger2], respectively. In the same way, in Fig. 11 we measured the number of I/Os required for the rebalancing operations during the insertion of a total of $2 \times 128\,971 = 257\,942$ y-projections and of $2 \times 131\,461 = 262\,922$ y-projections from [Tiger1] and [Tiger2], respectively.

The experiments above show that our structure requires from 2 to 3 I/Os for both searching and rebalancing operations. This stems from the fact that the MBRs' projections from the data sets [Tiger1] and [Tiger2] follow an almost uniform distribution, due to the almost uniform decomposition of spatial maps. The better performance in [Tiger2] is due to the fact that this is a dense spatial map and hence the derived one-dimensional data produce densely populated elements.

As a final remark, we note that there are applications with uniform key sizes larger than 8 bytes, resulting in a smaller value of B. The main example of such applications involve manipulation of strings (stemming from efficient manipulation

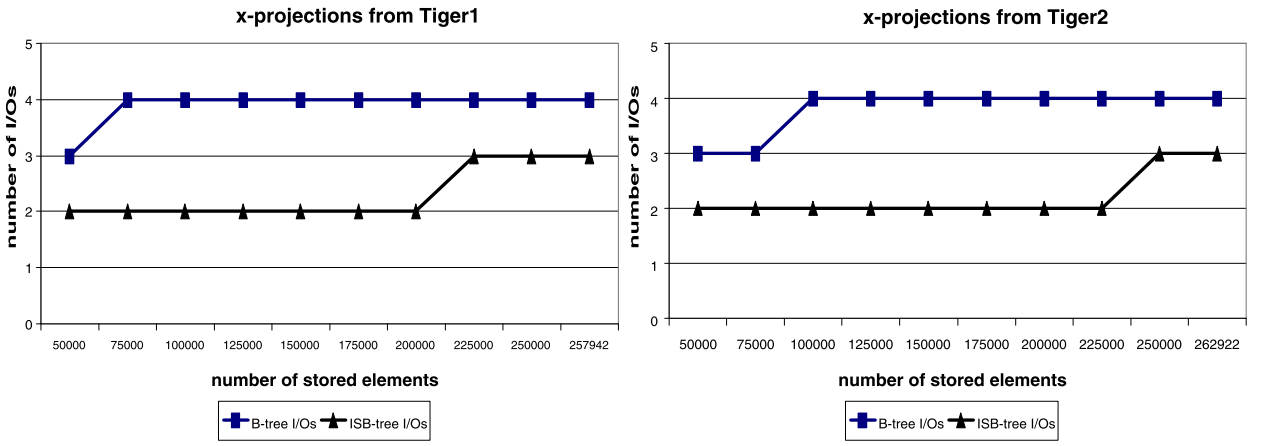

Fig. 8. Search performance for MBR's x-projections of [Tiger1] (left) and [Tiger2] (right).

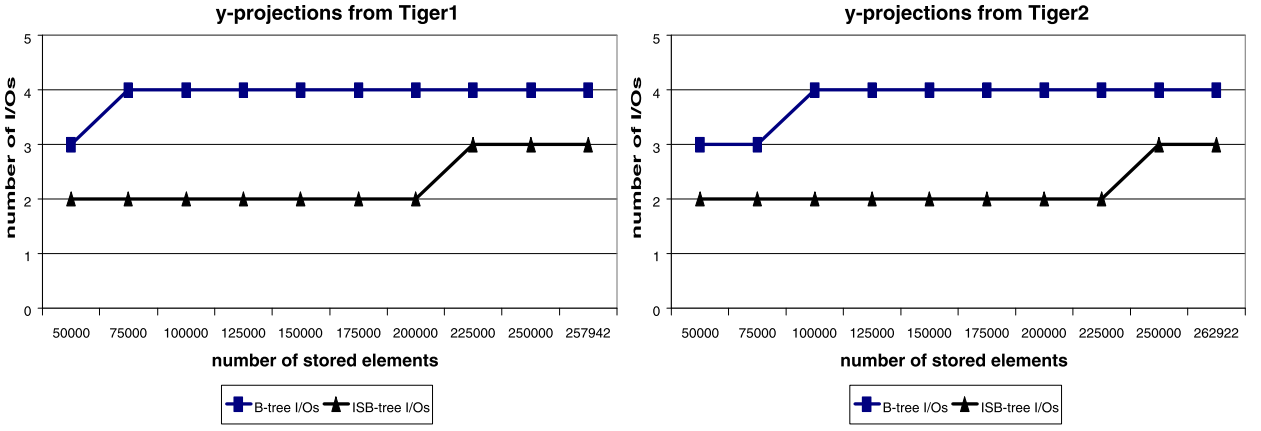

Fig. 9. Search performance for MBRs' y-projections of [Tiger1] (left) and [Tiger2] (right).

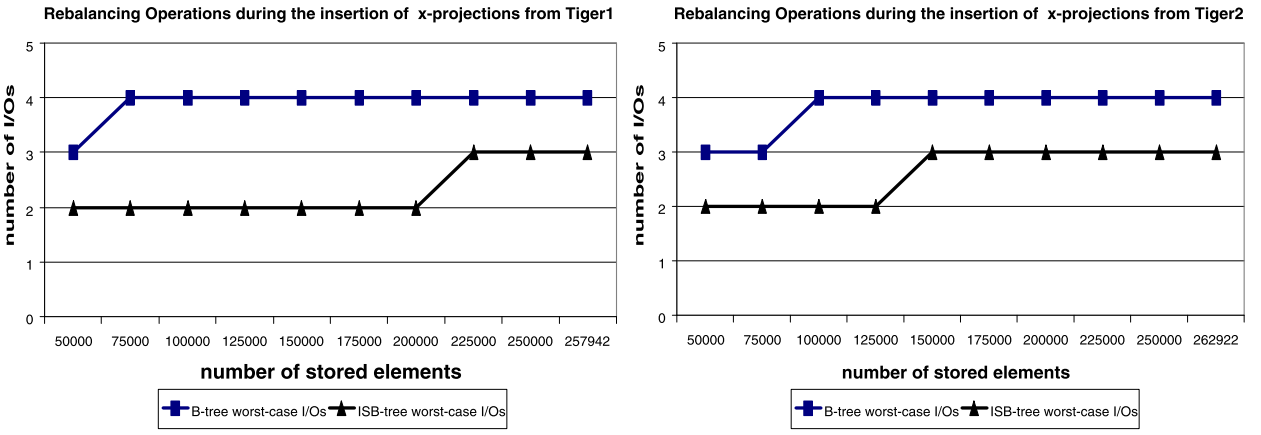

Fig. 10. Performance of rebalancing operations after an update for MBRs' x-projections of [Tiger1] (left) and [Tiger2] (right).

of DNA sequences). In this case, the size of the B may be as small as 2. Consequently, in such cases the ISB-tree will have a much better performance. This, however, is the subject of future investigation.

6. Conclusions

We presented a new indexing scheme, the ISB-tree, that supports update operations in $O(1)$ worst-case block transfers and search operations in $O(\log_B \log n)$ expected block transfers with high probability for a large class of input distributions.

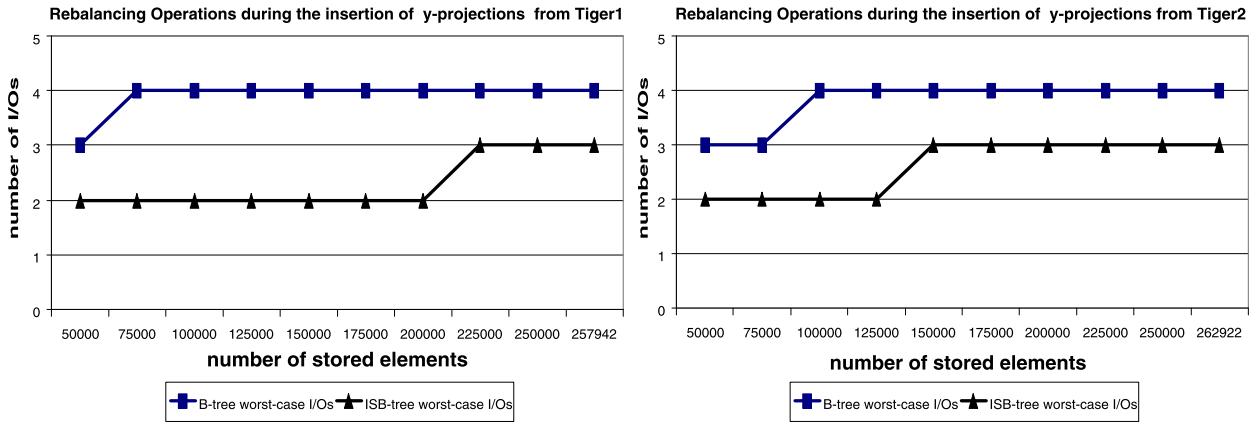

Fig. 11. Performance of rebalancing operations after an update for MBRs' y-projections of [Tiger1] (left) and [Tiger2] (right).

The ISB-tree shoots down for the first time the optimal $O(\log_B n)$ block transfer bound of B-tree and its variants when the updates are drawn from a large class of input distributions. Its analysis is based on the traditional I/O model of [2,36], and we conjecture that it can also be implemented in the cache-oblivious model [13] with the same complexities.

Acknowledgements

We would like to thank Yannis Theodoridis and Apostolos Papadopoulos for several interesting discussions. We are also indebted to the anonymous referees for their comments that improved the presentation of the paper.

References

[1] P.K. Agarwal, L. Arge, G. Brodal, J.S. Vitter, I/O-efficient dynamic point location in monotone planar subdivisions, in: Proc. 10th ACM–SIAM Symp. on Discrete Algorithms – SODA'99, 1999, pp. 11–20.
[2] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.
[3] A. Andersson, C. Mattson, Dynamic interpolation search in $o(\log\log n)$ time, in: Proc. ICALP'93, in: Lecture Notes in Comput. Sci., vol. 700, Springer-Verlag, 1993, pp. 15–27.
[4] L. Arge, J.S. Vitter, Optimal dynamic interval management in external memory, in: Proc. 37th IEEE Symp. on Foundations of Computer Science – FOCS'96, 1996, pp. 560–569.
[5] R.A. Baeza-Yates, Bounded disorder: The effects of the index, Theoret. Comput. Sci. 168 (1996) 21–38.
[6] R. Bayer, E. McCreight, Organization of large ordered indexes, Acta Inform. 1 (1972) 173–189.
[7] N. Beckmann, H. Krigel, R. Schneider, B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, SIGMOD, 1990.
[8] D. Comer, The ubiquitous B-tree, ACM Comput. Surveys 11 (2) (1979) 121–137.
[9] P. Dietz, R. Raman, A constant update time finger search tree, Inform. Process. Lett. 52 (1994) 147–154.
[10] R. Fagin, J. Nievergelt, N. Pippinger, H.R. Strong, Extendible hashing-A fast access method for dynamic files, ACM Trans. Database Syst. 4 (3) (1979) 315–344.
[11] P. Ferragina, R. Grossi, The string B-tree: A new data structure for string search in external memory and its applications, J. ACM 46 (2) (1999) 236–280.
[12] E. Fox, Q. Chen, A. Daoud, Practical minimal perfect hash functions for large databases, Commun. ACM 35 (5) (1992) 105–121.
[13] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proc. 40th IEEE Symp. on Foundations of Computer Science – FOCS'99, 1999, pp. 285–297.
[14] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Improved bounds for finger search on a RAM, in: Algorithms – ESA 2003, in: Lecture Notes in Comput. Sci., vol. 2832, Springer-Verlag, 2003, pp. 325–336, Full version in: Algorithmica, forthcoming.
[15] A. Kaporis, C. Makris, G. Mavritsakis, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, ISB-tree: A new indexing scheme with efficient expected behaviour, in: Algorithms and Computation – ISAAC 2005, in: Lecture Notes in Comput. Sci., vol. 3827, Springer-Verlag, 2005, pp. 318–327.
[16] D.E. Knuth, Sorting and searching, in: The Art of Computer Programming, vol. 3, Addison–Wesley, 1973.
[17] D.E. Knuth, Deletions that preserve randomness, IEEE Trans. Softw. Eng. 3 (1977) 351–359.
[18] P. Lehman, S. Bing Yao, Efficient locking for concurrent operations on B-trees, ACM Trans. Database Syst. 6 (4) (1981) 650–670.
[19] C. Levcopoulos, M.H. Overmars, Balanced search tree with $O(1)$ worst-case update time, Acta Inform. 26 (1988) 269–277.
[20] W. Litwin, Linear hashing: A new tool for files and tables addressing, Proc. of the 6th Internat. Conf. on Very Large Databases, 1980, pp. 212–223.
[21] W. Litwin, D. Lomet, A new method for fast data searches with keys, IEEE Software 4 (2) (1987) 16–24.
[22] D. Lomet, A simple bounded disorder file organization with good performance, ACM Trans. Database Syst. 13 (4) (1988) 525–551.
[23] Y. Manolopoulos, Y. Theodoridis, V. Tsotras, Advanced Database Indexing, Kluwer Academic Publishers, 2000.
[24] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.
[25] K. Mehlhorn, A. Tsakalidis, Dynamic interpolation search, J. ACM 40 (3) (1993) 621–634.
[26] P.E. O'Neil, The SB-tree. An index-sequential structure for high-performance sequential access, Acta Inform. 29 (3) (1992) 241–265.
[27] M. Overmars, J. van Leeuwen, Worst case optimal insertion and deletion methods for decomposable searching problems, Inform. Process. Lett. 12 (4) (1981) 168–173.
[28] R. Raman, Eliminating amortization, on data structures with guaranteed response time, PhD Thesis, Department of Computer Science, University of Rochester, New York, 1992; Technical Report TR-439, 1992.
[29] J. Rao, K. Ross, Cache conscious indexing for decision-support in main memory, in: M. Atkinson, et al. (Eds.), Proc. International Conference on Very Large Databases, vol. 25, Morgan Kaufmann, San Mateo, CA, 1999, pp. 78–89.

[30] J. Rao, K. Ross, Making B+-trees cache conscious in main memory, in: Proc. ACM SIGMOD International Conference on Management of Data, 2000, pp. 475–486.
[31] B. Saltzberg, V. Tsotras, Comparison of access methods for time-evolving data, ACM Comput. Surveys 31 (1999) 158–221.
[32] B. Seeger, P.A. Larson, Multi-disk B-trees, in: Proc. SIGMOD Conference 1991, pp. 436–445.
[33] V. Srinivasan, M.J. Carey, Performance of B+ tree concurrency algorithms, VLDB J. 2 (4) (1993) 361–406.
[34] Y. Theodoridis, The R-tree Portal, http://www.rtreeportal.org, 2003, [Tiger1] and [Tiger2] data sets in: http://www.rtreeportal.org/spatial.html.
[35] J.S. Vitter, External memory algorithms and data structures: Dealing with massive data, ACM Comput. Surveys 33 (2) (2001) 209–271.
[36] J.S. Vitter, E.A.M. Shriver, Optimal algorithms for parallel memory I: Two-level memories, Algorithmica 12 (2-3) (1994) 110–147.
[37] D.E. Willard, Searching unindexed and nonuniformly generated files in $\log\log N$ time, SIAM J. Comput. 14 (4) (1985) 1013–1029.