DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **50**, 1999

S(b)-Tree Library: an Efficient Way of Indexing Data

Konstantin V. Shvachko

Abstract. We present a library for maintaining external dynamic dictionaries with variable length keys. A new type of balanced trees, called $S(b)$-trees, is introduced which contrary to the well-known B-trees provide optimal packing of keys of variable length, while the data access time remains logarithmic, the same as for B-trees. $S(b)$-trees are implemented as a stand-alone library. The library functionality includes means for creating, storing, and performing the basic operations for $S(b)$-trees. The library documentation, source code, and executables are available at `http://namesys.botik.ru/~shv/stree`

Introduction

Indexing is a common mechanizm used in database retrieval. Any database programming system provides means for the creation and maintanence of indexes. Most index implementations are based on balanced trees.

The indexing problem is usually referred to as the problem of *"maintaining dynamic dictionaries"*. A *dynamic dictionary* is a data structure for storing dictionary elements, called keys, together with algorithms for accessing, inserting, and deleting a key.

A number of variants of balanced trees are known. AVL-trees [**AVL62**], 2-3-trees ([**LD91**], [**Y78**]), red-black-trees [**W93**] and some other variants of balanced trees with low branching of internal nodes that are mostly appropriate for representation of *internal dictionaries*.

From the practical viewpoint, B-trees ([**B72**], [**BM72**], [**K73**], [**TF82**], [**W86**]), together with all their modifications, like B^+-trees [**W93**], B^*-trees [**M77**], (a,b)-trees [**LD91**], are considered to be the most common data structure for representing indexes in *external memory*. Unfortunately, B-trees have a serious disadvantage ([**K73**], [**PS92**]) in the case of variable length keys, since in this case they can lead to an exhaustive waste of memory.

First published in *"DIMACS Series in Discrete Mathematics and Computer Science"*, vol. 50, pp.207-222, 1999, published by the American Mathematical Society.

This project was supported in part by the Russian Foundation of Basic Research (grant #96-01-01005). The project was started within the DIMACS Chellenge'96.

We introduce a new type of balanced trees, called $S(b)$-trees, which contrary to B-trees provide optimal packing of keys of variable length, while the data access time remains logarithmic, the same as for B-trees.

$S(b)$-trees are implemented as a stand-alone library. The library functionality includes means for creating, storing, and performing the basic operations for $S(b)$-trees. The implementation is written in C++ and a pure C function interface is provided. The $S(b)$-tree library is designed to be platform-independent, and is supposed to run both on UNIX, and MS Windows platforms.

1. Common properties of balanced trees

Balanced trees are intended to provide an efficient solution to the problem of maintaining dynamic dictionaries.

A balanced tree stores keys chosen from a finite set of keys K. K is linearly ordered, and the keys are placed in the tree according to the ordering. Each node of the tree contains a number of keys. Leaves don't contain anything else, while the internal nodes additionally contain references to other nodes. The number of references in an internal node equals the number of keys in the node plus 1. An internal node S composed of m keys is denoted by

$$S = (S_0, k_0, ..., S_i, k_i, S_{i+1}, ..., k_{m-1}, S_m)$$

Trees satisfying the properties below are called *structured trees*.

- All paths in the tree from the root to a leaf have equal lengths.
- For any node S of the tree, the keys it stores are located in S according to K's linear ordering, that is k_i is less than k_j for $i < j$.
- For any key k_i of an internal tree node $S = (S_0, k_0, ..., S_i, k_i, S_{i+1}, ..., k_{m-1}, S_m)$ all keys located in the sub-tree accessible via the reference S_i to the left of k_i are less than k_i, and all keys located in the sub-tree accessible via the reference S_{i+1}, which is to the right of k_i, are greater than k_i.

The last property is the bases of an efficient tree search. The search algorithm is common to all structured trees. Starting from the root, one should look for the given key k in the current node, and either find the key in the node (the searching is finished) or find a pair of adjacent keys k_{i-1}, and k_i such that $k_{i-1} \le k \le k_i$. In the latter case searching is continued in the sub-tree referenced by S_i.

2. B-Trees

Definition 2.1. *A* B-tree *of order q is a structured tree whose nodes except for the root contain at least q, and at most $2q$ keys. The root must contain at least 1 key, and at most $2q$.*

The restrictions associated with the tree order q provide high branching of the B-tree internal nodes, and guarantee a fast tree searching.

It is well known that searching, insertion, and deletion in a B-tree can be performed in time proportional to the height of the tree, which is at most $\log_2 n$, where n is the number of nodes in the tree.

A *utilization* $\delta(T)$ of a B-tree T with n nodes is defined to be the ratio of the total number $|T|$ of keys in the tree to the maximal possible number $2qn$ of keys in

an n-node B-tree. Since the minimal number of keys is qn, and the maximal is $2qn$ the utilization is bounded by

$$\frac{1}{2} \leq \delta(T) = \frac{|T|}{2qn} \leq 1$$

The 1/2 lower bound appears to be acceptable, but in practice things look different. Indeed, in an implementation of B-trees it is natural to use fixed size blocks to store the tree nodes – one block per node. According to the definition of B-trees each block must fit $2q$ keys. Suppose that the key set K consists of keys of different lengths, and that their maximal length is l. Then the block size should be at least $2ql$. This means that if a node stores keys of size $l/10$, then it is only 1/10 full even if the number of keys is maximal. Therefore, actually we cannot guarantee any lower bound greater than 0 for the utilization of the tree when key lengths are taken into account.

3. Definition of $S(b)$-trees

To improve tree utilization a *weight function* μ for the key set K have to be defined. From the practical viewpoint the weight is the number of bytes the key is stored in. The weight is different to the length since sometimes additionally to the key itself it is necessary to store some extra information, like the key length or the ending zero byte. For any key k, $\mu(k)$ denotes the *weight* of k. $\mu_{\max}(K)$ denotes the maximum of key weights in K. For any node S, $\mu(S)$ denotes the node weight, that is the total weight of keys contained in S, while $M(S)$ denotes the total weight of keys contained in the tree rooted at S.

A $S(b)$-tree (read as *sweep b tree*) is characterized by the following three parameters:

(1) b – the *locality parameter*,
(2) q – the tree *order*, and
(3) p – the tree *rank*.

The tree order q specifies the minimal number of keys in a non-root node of the tree. The root must contain at least one key. The tree rank p specifies the maximal total weight of keys in a node. We say that a tree node S is *well-formed* if the number of keys $|S| \geq q$ ($|S| \geq 1$ for the root), and if the total weight of the node $\mu(S) \leq p$.

Tight packing of keys in $S(b)$-trees is provided by the *incompressibility* property.

Let us consider $m + 1$ adjacent nodes of the same level of the tree and m *delimiting keys*, which separate the references to the adjacent nodes in the nodes' common parent. Such a collection of nodes, and their delimiting keys is called a *sweep*, and m is defined to be the length of the sweep. The sweep of length m is called *compressible* if one can construct a sweep of smaller length containing all the keys of the initial sweep, and composed of at most m well-formed nodes, and at most $m - 1$ delimiting keys. Otherwise, the sweep is called *incompressible*. Therefore, a *tree is m-incompressible* if any sweep of length m in it is incompressible.

DEFINITION 3.1. *An $S(b)$-tree of order q, and rank p is a structured tree that is incompressible with respect to the locality parameter b, such that*

(1) $b \geq 0$
(2) $q \geq b \;\&\; q \geq 1$
(3) $p \geq 2q\,\mu_{\max}(K)$

The restrictions above are essential for the correctness proofs.

In [**PS92**], [**S95Obn**], [**S95CMA**] we proved that $S(b)$-trees generalize B-trees, meaning that if the weight function μ equals 1 on K then each B-tree of order q is a $S(0)$-tree of order q, and rank $2q$.

4. Utilization of $S(b)$-trees

A *utilization* $\Delta(T)$ of an n-node $S(b)$-tree of rank p is the ratio of the total weight of the tree $M(T)$, that is the sum of weights of keys stored in the tree, devided by the maximum possible weight of an n-node $S(b)$-tree of rank p:

$$\Delta(T) = \frac{M(T)}{np}$$

We sketch a proof below of the following key result

THEOREM 4.1. *For any linearly ordered weighted finite key set K, and for any $\varepsilon > 0$ the three parameters $b > 0$, $q \geq b$ and $p \geq 2q\mu_{\max}(K)$ can be chosen in such a way that for any $S(b)$-tree T of order q, and rank p, composed of keys from K and containing at least $(b+1)^2$ vertices,*

$$\Delta(T) > 1 - \varepsilon$$

where ε is inversely proportional to b.

Under the assumptions of the theorem, for any n–node $S(b)$-tree we prove that

$$\Delta(T) > \frac{b}{b+1}\frac{q}{q+1} - \frac{b+1}{n}$$

The proof of the lower bound is based on the following technical lemma.

LEMMA 4.1. *Consider an n–node structured tree T of order q such that $n > q+1$. Then the number of leaves x of the tree is bounded by*

$$x > \frac{q(n-1)+1}{q+1}$$

Lemma 4.1 can be proven using induction by the tree height.

To prove the theorem let us consider an n-node $S(b)$-tree T and a sweep σ of T, composed of all leaves of the tree together with their delimiting keys. Note that this sweep includes all keys contained in the tree. Let us partition σ into the maximal number s of disjoint sub-sweeps $\sigma_i = S_0^i, k_1^i, S_1^i, \ldots, k_b^i, S_b^i$ $(i = 1, \ldots, s)$ of length b.

By Definition 3.1 all sweeps of length b in T are incompressible, which implies that $\mu(\sigma_i) > bp$. Therefore, since the chosen sweeps are disjoint we have

$$M(T) = \mu(\sigma) \geq \sum_{i=1}^{s} \mu(\sigma_i) > bps$$

Let x be the number of leaves of T. Then the number of chosen sub-sweeps is

$$s = \left\lfloor \frac{x}{b+1} \right\rfloor$$

Applying Lemma 4.1 we obtain

$$s > \frac{x}{b+1} - 1 > \frac{1}{b+1}\frac{q(n-1)+1}{q+1} - 1 > n\frac{1}{b+1}\frac{q}{q+1} - \frac{1}{b+1}\frac{q-1}{q+1} - 1$$

Using the two bounds above we get

$$\Delta(T) = \frac{M(T)}{np} > b\frac{s}{n} > \frac{b}{b+1}\frac{q}{q+1} - \frac{1}{n}(\frac{b}{b+1}\frac{q-1}{q+1} + b)$$

which implies the desired bound.

This lower bound means that $S(b)$-trees provide almost optimal packing of any given finite set of keys. Special cases of this bound are proven in [**PS92**] and [**S95CMA**].

5. The sketch of the algorithms

The algorithm to *search* in an $S(b)$-tree is the same as for all other balanced trees, (see Section 2).

The algorithms for insertion and deletion in an $S(b)$-tree are more complicated then the corresponding ones for B-trees. The difficulty is that in the $S(b)$-tree case, balancing of a modified node S involves additional b neighboring nodes to the left and to the right. For B-trees, insertions and deletions involve at most one neighbor.

It can be proven that an insertion, and a deletion of a key in an n-node $S(b)$-tree can be performed in at most $C\log(n)$ time where C is a constant that is independant of n and proportional to the locality parameter b.

5.1. Insertion. Using the search algorithm an *insertion* first verifies whether the given key k is contained in the given tree T. If k is in T then the insertion is finished. Otherwise the search procedure returns a leaf S, and a key k_i in it, before which the key k must be inserted. k is inserted in its corresponding place in S, and the insertion starts balancing the tree. Balancing is performed by a special procedure, which is a common part of both the insertion and the deletion algorithms. Balancing starting from the enlarged leaf S balances all the nodes that lay on the path from the root of the tree to S.

5.2. Deletion. Similarly to insertion, a *deletion* begins with a search. If the given key k is not found in the given $S(b)$-tree T, then the deletion is finished. Otherwise the search returns a node S, and a key k_i in it, that must be deleted. Note that S can be either an internal node or a leaf. The case when S is not a leaf can be easily reduced to the case of deletion from a leaf. If S is internal, we consider the sub-tree accessible via the reference S_{i+1}, which is to the right of k_i in S. Take the left most leaf S' in the sub-tree, and the smallest key k' in S', and replace k_i by k' in S. Thus the problem is reduced to the deletion of k' from the leaf node S'. Now let S be a leaf, and k_i be the key that must be deleted from S. The deletion algorithm removes k_i from S and starts balancing the tree with the same balancing algorithm that is used in insertions.

5.3. Balancing. The Balance() procedure is the main common part of the insertion and the deletion algorithms. Balancing starts at the leaf node S given as an input parameter to the procedure. After working on the level of the current node S the procedure takes for balancing the direct parent of S. The process proceeds further up to the tree root. The balanced tree is the result of the procedure Balance().

For any current node S the procedure decides to balance S if one of the following three conditions holds.

(1) One of the $b+1$ sweeps of length b, containing S, is not incompressible.

(2) $|S| < q$.
(3) $\mu(S) > p$.

In the first case the procedure Balance_B() is used for balancing S. If Condition 2 holds for S then Balance_C() is called. And in the case of Condition 3 S is balanced by Balance_W(). When none of the conditions holds, Balance() skips the level.

Each of the three procedures restores the structure of the $S(b)$-tree disturbed locally for one, two or three vertices of the current tree level. While correcting the structure of the tree on the current level, the algorithm should change also the ancestors of S. This can break in turn the balance conditions for the lower level nodes. Such breakdowns are also local, since not more than three lower level nodes can be changed: the direct parent of S, and two its neighbors to the left and to the right of S. Coming to the next level of the tree Balance() merges the modified nodes of the level, and balances them in its entirety.

The computation stops at the tree root. Thus the algorithm examines all the nodes that lay on the path from the root to the modified leaf, and balances them if necessary. Only these nodes and their neighbors (b to the left and b to the right) in the tree can be transformed by the algorithm.

After balancing the structure of the $S(b)$-tree is restored, all tree nodes are well-formed, and all sweeps of the tree are incompressible.

6. The specifications

Let T be a structured tree and S be its node. Below we present a list of variables and instrumental procedures used to describe the algorithms.

6.1. Denotations and definitions. With respect to the current node S a sweep

$$\langle L_{b-1}, l_{b-1}, \ldots, L_0, l_0, S, r_0, R_0, \ldots, r_{b-1}, R_{b-1} \rangle$$

composed of b neighbors and their delimiting keys to the left and to the right of S is called the *vicinity* of S, where

- S is the current node;
- L_i denotes the i-th left neighbor of S;
- R_i denotes the i-th right neighbor of S;
- l_i is the delimiting key of nodes L_i and L_{i-1};
- r_i is the delimiting key of nodes R_i and R_{i-1}.
- F denotes the direct parent of S.
- FL_i (FR_i) denotes the parent of node L_i (R_i).
- Fl_i (Fr_i) denotes the node containing the delimiting key l_i (r_i).
- $L_i, l_i, R_i, r_i, FL_i, FR_i, Fl_i, Fr_i$ are the local variables of the procedures.
- $WW(S)$ means that $\mu(S) \le p$.
- $WC(S)$ means that $|S| \ge q$.
- $WF(S)$ means that $WW(S)$ and $WC(S)$ hold.
- $IC(\sigma)$ means that sweep σ is incompressible.
- $WB(S)$ means that $IC(\sigma)$ holds for any sweep σ of length b containing S.
- $Sweep_b^m(S)$ denotes the m-th ($m = 0, \ldots, b$) sweep of length b containing node S, namely

$$\begin{aligned} Sweep_b^0(S) &= \langle L_{b-1}, l_{b-1}, \ldots, L_0, l_0, S \rangle \\ Sweep_b^b(S) &= \langle S, r_0, R_0, \ldots, r_{b-1}, R_{b-1} \rangle \\ Sweep_b^m(S) &= \langle L_{b-m-1}, l_{b-m-1}, \ldots, L_0, l_0, S, r_0, R_0, \ldots, r_{m-1}, R_{m-1} \rangle \end{aligned}$$

6.2. Instrumental procedures. The following procedures and functions are used for describing the algorithms.

- *Search*(T, k), given an $S(b)$-tree T and key k, verifies whether the key is contained in the tree. The result of the function is $(\mathit{IsFound}, S, k_i)$. *IsFound* is a Boolean variable that specifies whether k is found in the tree or not. S is the node that was visited last while searching in the tree, and k_i is the minimal key of S that is greater than or equal to the given key k.
- *MakeVicinity*(S) initializes the local variables $L_i, l_i, R_i, r_i, FL_i, FR_i, Fl_i, Fr_i$ according to S.
- *Replace*(S, P, Q), substitutes the portion of the node S that coincides with P, with Q.
- Functions *LeftOf*(S, k) and *RightOf*(S, k) specify two portions of S, that are to the left and to the right of key k in S, respectively.
- For each $m = 0, \ldots, b$ we define a function $\mathit{Compress}^m$ that is applied to compressible sweeps of length b. The result is an incompressible sweep of length $b - 1$, composed of b well-formed nodes. Namely, if

$$\mathit{Compress}^m(A_0, a_1, \ldots, a_b, A_b) = \langle C_0, c_1, \ldots, c_{b-1}, C_{b-1} \rangle$$

  then the resulting sweep is obtained by distributing the contents of node A_m between the other nodes of the initial sweep in such a way that $a_i < c_i$ for $i = 1, \ldots, m$, and $c_i < a_{i+1}$ for $i = m, \ldots, b - 1$.
- *ComputeSets*(S) calculates seven subsets of the set of keys of node S: *MLeft*, *MRight*, *MDelimL*, *MDelimR*, *MDelim*, $\overline{\mathit{MLeft}}$, and $\overline{\mathit{MRight}}$, defined below.
- *ChooseAny*(E), *ChooseMax*(E), and *ChooseMin*(E) for the given key set E return, respectively, an arbitrary, the maximum, and the minimum keys of the set.
- *SearchMinimal*(S) looks for the minimal key in the sub-tree rooted at S, returns (S', k') where k' is the minimal key, and S' is the leaf that contains k'.
- *GlueParents*(F) takes the parent F of the current node S, the left, and the right neighbors of F, glues them into one node, and returns the result. A more sophisticated variant of this function is to check before gluing whether the neighbors have been modified, and glue to F only the modified ones.
- *CreateNewRoot*(S) creates the new root of the tree, containing the only reference to the given node S.
- *ReleaseRoot*(). If the root of the tree contains only one reference to a node S, and no keys, then this root is removed, and S is assigned to be the new root of the tree.
- $[P, d, Q]$ creates a new node composed of all keys, and node references (in case of internal nodes), of the given nodes P and Q, with the key d between them. E.g., $[(L_0, l_1, L_1), d, (R_0, r_1, R_1)] = (L_0, l_1, L_1, k, R_0, r_1, R_1)$

If $k(S)$ denotes the set of all keys of node S, then the definitions of the subsets are as follows.

$$
\begin{array}{lcl}
\mathit{MLeft} & = & \{d \in k(S) \mid IC(\langle L_{b-1}, l_{b-1}, \ldots, L_0, l_0, \mathit{LeftOf}(S,d)\rangle)\} \\
\mathit{MRight} & = & \{d \in k(S) \mid IC(\langle \mathit{RightOf}(S,d), r_0, R_0, \ldots, r_{b-1}, R_{b-1}\rangle)\} \\
\mathit{MDelimL} & = & \{d \in k(S) \mid WF(\mathit{RightOf}(S,d))\} \\
\mathit{MDelimR} & = & \{d \in k(S) \mid WF(\mathit{LeftOf}(S,d))\} \\
\mathit{MDelim} & = & \mathit{MDelimL} \cap \mathit{MDelimR} \\
\overline{\mathit{MLeft}} & = & k(S) \setminus \mathit{MLeft} \\
\overline{\mathit{MRight}} & = & k(S) \setminus \mathit{MRight}
\end{array}
$$

The sets, and the *ComputeSets*() procedure are used in Balance_W() to control the process of computation.

6.3. The balance stages. As we mentioned above, to balance the tree on each tree level one of the three balance procedures is called. In each case balancing is performed by redistributing the keys within the b-vicinity of the current node S in such a way that both $WF(Q)$ and $WB(Q)$ hold for any node Q of the vicinity after the balancing is finished. The explicit algorithms are outlined in Appendix.

Procedure BALANCE_W is used to balance the input tree when the current vertex S weight is larger than p ($\neg WW(S)$), while the other two properties $WC(S)$ and $WB(S)$ are satisfied for S. The procedure consists of five stages. The decision on whether a stage should be performed or not is based upon the interrelation of the seven subsets of $k(S)$.

Informally,

- a key d from $k(S)$ belongs to *MLeft* (*MRight*) iff the current vertex S can be split into two parts in such a way that the fragment of the b-vicinity of S which is to the left (right) of d is incompressible;
- a key d from $k(S)$ belongs to *MDelimL* (*MDelimR*) iff the current vertex S can be split into two parts in such a way that the weight of the part of S that is to the right (left) of d is not greater than p, while the number of keys of the part is at least q.

The "Compression" stage is performed when the intersection $\overline{\mathit{MLeft}} \cap \overline{\mathit{MRight}}$ is not empty, meaning that all keys of S can be distributed between the other nodes of the vicinity and S can be eliminated by that.

The "Move left" stage is performed if $\overline{\mathit{MLeft}}$ is not empty, which means that a number of keys of S can be moved to the left part of the vicinity.

The "Move right" stage is analogous.

If after shifting of as many keys as possible from S into the left and right parts of the vicinity, the weight of the keys remaining in S is still greater than p, then the "Split" stage is performed in order to partition S into two parts at least one of which (actually the left one) is well-formed. If the second node is also well-formed, then balancing is finished.

Otherwise, the "Recursion" stage is invoked. It balances the remaining not well-formed node by recursive call of BALANCE_W. It can be shown that the depth of this recursion is constant.

The procedure BALANCE_B is used to balance the input tree when one of the sweeps $\mathit{Sweep}_b^m(S)$ ($m = 0, \ldots, b$) is compressible ($\neg WB(S)$). Balancing in this case is performed by compressing one or two sweeps of the b-vicinity of S. It can be shown that elimination of more than two vertices of the vicinity is impossible. Starting from the left most sweep $\mathit{Sweep}_b^0(S)$ BALANCE_B scans the vicinity

and compresses each compressible sweep found. After the second compression it terminates.

Procedure BALANCE_C is called when the number of keys in the current node S is less than q ($\neg WC(S)$). In this case we join S with one of it's neighbors and call BALANCE_W for the resulting node if required.

7. $S(b)$-tree library interface

The $S(b)$-tree library implements the algorithms described above, and provides means for creating, storing, and performing the basic operations for $S(b)$-trees. The implementation is written in C++ and a pure C function interface is provided. The $S(b)$-tree library is designed to be platform-independent, and it runs both on UNIX and Windows NT/95 platforms.

7.1. Keys. The *key* type is S_KEY. To create a key a S_CreateKey(Key, String, Length) function is used. Given an array of bytes and its length, it sets the given key to the specified string value.

Note that one can store not only string-valued keys, but keys of an arbitrary structure by providing a conversion of a user defined key to a byte array. E.g., a number can be easily converted to a string using, say, the C run-time library routines itoa(), or ltoa().

A key *weight* equals the length of the byte array plus a constant given by S_EMPTY_KEY_WEIGHT.

7.2. Maintaining the trees. Several functions provide means for creating new $S(b)$-trees, opening existing ones, saving, and closing the tree modification sessions.

In order to create an $S(b)$-tree it is necessary to specify the following parameters.

- FileName is the name of the file where the tree will be stored.
- b is the locality parameter. The better packing you need the greater b should be chosen.
- q is the order of the $S(b)$-tree. It specifies the minimal number of keys in a node, and is intended to provide high branching of the tree internal nodes.
- p is the tree rank. It is the size of the block for storing the tree nodes, meaning that a node weight cannot exceed p.
- MuMax characterizes the key set K, in the way that all keys in K have length not greater than MuMax.

A node weight is the sum of weights of keys contained in the node plus a constant S_EMPTY_NODE_WEIGHT. In the implementation we need to store some header for each tree node that contains particularly the number of keys in the node, which is required for correct reads of tree nodes. Thus, actually the tree rank according to our definitions in Section 4 is $p -$ S_EMPTY_NODE_WEIGHT.

The restrictions for the parameters that provide correctness of the algorithms are as follows:

(1) $b \geq 1$
(2) $q \geq b$ & $q \geq 2$
(3) $p \geq 2q$ MuMax + S_EMPTY_NODE_WEIGHT

S_CreateTree(ResTreeHandle, FileName, b, q, p, MuMax)
creates an empty tree with the specified parameters, and returns the new tree handle ResTreeHandle, which provides access to the tree in other functions.

Another way to get access to an $S(b)$-tree is to load it, which is performed by
S_LoadTree(ResTreeHandle, FileName).

S_CloseTree(TreeHandle)
closes the tree, saves it on disk in the file it was created at or loaded from, and releases the tree handle.

S_SaveTree(TreeHandle)

saves the specified tree on disk in the file it was created at or loaded from, but does not close the tree leaving it accessible via the tree handle.

7.3. The basic tree operations. The main algorithms of search, insertion, and deletion are implemented in the following functions

S_Search(TreeHandle, Key, IsFound)
S_Insert(TreeHandle, Key)
S_Delete(TreeHandle, Key)

7.4. Additional operations. Since the set of keys stored in an $S(b)$-tree is linearly ordered, it is natural to provide access to the first, the last key in the dictionary, the next, and the previous keys with respect to the given one.

S_First(TreeHandle, FirstKey, IsFound)
S_Last(TreeHandle, LastKey, IsFound)
S_Next(TreeHandle, InpKey, NextKey, IsFound)
S_Prev(TreeHandle, InpKey, PrevKey, IsFound)

Note that the given key InpKey should not necessarily be contained in the tree. To find the next (the previous) means to find the minimal (maximal) key in the tree that is greater (less) than the input key.

7.5. $S(b)$-tree properties. The rest of the functions return the specified tree parameters: b, q, p, and MuMax, and the tree intrinsic properties:

- Number of nodes in the tree
- Number of keys in the tree
- The tree total weight
- The tree height
- The tree utilization

The prototypes of the functions are:
int S_TreeLocality(TreeHandle)
int S_TreeOrder(TreeHandle)
int S_TreeRank(TreeHandle)
int S_TreeMaxKeyWeight(TreeHandle)
long S_TreeNrNodes(TreeHandle)
long S_TreeNrKeys(TreeHandle)
long S_TreeWeight(TreeHandle)
int S_TreeHeight(TreeHandle)
double S_TreeUtilization(TreeHandle)

8. Conclusion

Balanced trees are the standard data structures for indexing information. The paper introduces a new type of balanced trees, called $S(b)$-trees. $S(b)$-trees generalize well known B-trees for the case of variable length keys. In Theorem 4.1 we present a lower bound of utilization of an $S(b)$-tree, which shows that $S(b)$-trees provide almost optimal packing of any given finite set of variable length keys. We describe logarithmic running time algorithms for the $S(b)$-tree-based dictionary search and update operations. $S(b)$-trees are implemented as a stand-alone library. The library functionality includes means for creating, storing, and performing an extended set of operations for $S(b)$-trees.

Besides the library, $S(b)$-trees were implemented in two different software systems with the common feature that both of them are designed to store completely unstructured data collections.

The high-level universal programming language *Starset* [**G94**] was developed to generalizes the traditional relational database approach. Starset is based on the set data model, which eliminates restrictions of the relational approach, such as multivalued attributes, varying arity, null values, etc. The Starset programming language uses $S(1)$-trees described in [**PS92**] for representing its set data aggregates.

The high-performance Tree File System (TreeFS) was designed to break down a tradeoff common for block oriented file systems where desire to increase the system block size in order to accelerate disk access contradicts the necessity of keeping the size small enough to avoid waste of disk space. A substantially modified variant of $S(1)$-trees is implemented in the *Tree File System*. This is probably the first attempt to represent a whole file system by a balanced tree in a Unix–like operating system. We have conducted experiments that show that most file operations are faster in TreeFS, especially for small files, and that the disk space utilization is higher compared to traditional file systems. TreeFS is implemented as a virtual FS under the Linux operating system.

9. Acknowledgments

I am thankful to Elena A. Zinovieva for helping with the implementation.

References

[AVL62] G.M. Adel'son-Vel'skii, E.M. Landis, *An Algorithm for the Organization of Information*, Soviet Math. Doklady vol. 3, 1972, pp. 1259–1262.

[B72] R. Bayer, *Symmetric binary B-tree: Data Structure and Maintenance Algorithms*, Acta Inf., vol. 1, 4, 1972, pp. 290–306.

[BM72] R. Bayer, E. McCreight, *Organization and Maintenance of Large Ordered Indexes*, Acta Inf., vol. 1, 3, 1972, pp. 173–189.

[G94] M.M. Gilula, *The Set Model for Database and Information Systems*, Addison-Wesley (In Association with ACM Press): Wokingham, 1994.

[K73] D.E. Knuth, *The Art of Computer Programming*, vol. 3 (Sorting and Searching), Addison–Wesley, Reading, MA, 1973.

[LD91] H.R. Lewis, L. Denenberg, *Data Structures and Their Algorithms*, HarperCollins, NY, 1991.

[M77] E.M. McCreight, *Pagination of B*-Trees with Variable-Length Records*, Commun. ACM, vol. 20, 9, 1977, pp. 670–674.

[PS92] A.P. Pinchuk, K.V. Shvachko, *Maintaining Dictionaries: Space-Saving Modifications of B-Trees*, Lecture Notes in Computer Science, vol. 646, 1992, pp. 421–435.

[S93] K.V. Shvachko, *Space-Saving Modifications of B-Trees*, In Proceedings of Symposium on Computer Systems and Applied Mathematics, St.Petersburg, 1993, p. 214.
[S95Obn] K.V.Shvachko, *Optimal Representation of Dynamic Dictionaries by Balanced Trees*, In Proceedings of XI International Conference on Logic, Methodology, and Philosophy of Science, Obninsk, Russia, vol. 2, 1995, pp. 181-186 (in Russian).
[S95CMA] K.V.Shvachko. *Space Saving Generalization of B-Trees with 2/3 Utilization*, Computers and Mathematics with Applications, vol. 30, No.7, 1995, pp. 47-66.
[S95NN] K.V.Shvachko, *A Hierarchy of Weight Balanced Trees*, In Proceedings of II International Conference on Mathematical Algorithms, N.Novgorod, Russia, 1995.
[TF82] T.J. Teorey, D.P. Fry, *Design of Database Structures*, vol. 2, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[W86] N. Wirth, *Algorithms and Data Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
[W93] D. Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley Publishing Company, 1993.
[Y78] A.C.-C. Yao, *On Random 2-3 Trees*, Acta Inf., vol. 9, 1978, pp. 159–170.

Appendix A. The main procedures

Procedure Insert(T, k)
($IsFound, S, k_i$) = *Search*(T, k);
if *IsFound* = *True* **then return fi**;
/* k haven't been found means that S is a leaf */
Replace($S, \langle k_i \rangle, \langle k, k_i \rangle$) /* insert k before k_i */
Balance(S); /* balance T starting from leaf S */
EndProcedure

Procedure Delete(T, k)
($IsFound, S, k_i$) = *Search*(T, k);
if *IsFound* = *False* **then return fi**;
if S is not a leaf **then**
    /* S is internal, and contains reference */
    /* S_{i+1}, which is to the right of $k_i$in S */
    (S', k') = *SearchMinimal*(S_{i+1});
    Replace($S, \langle k_i \rangle, \langle k' \rangle$);
    $S := S'$;
    $k := k'$;
fi
/* S is a leaf */
Replace($S, \langle k \rangle, \langle \rangle$) /* delete k from leaf S */
Balance(S); /* balance T starting from leaf S */
EndProcedure

Procedure Balance(S)
while S is not the root **do**
    if $\neg WB(S)$ **then** $F :=$ Balance_B(S);
    else if $\neg WC(S)$ **then** $F :=$ Balance_C(S);
    else if $\neg WW(S)$ **then** $F :=$ Balance_W(S);
    else $S := F$; **continue**;
    fi fi fi
    $S :=$ *GlueParents*(F);
od
/* S is the root now */
if $|S| = 0$ **then**
    ReleaseRoot();
fi
if $\neg WW(S)$ **then**
    CreateNewRoot(S);
    Balance_W(S);

```
fi
EndProcedure

Procedure Balance_W(S)
MakeVicinity(S);
ComputeSets(S);

/* Compression: */
/* Distribute all keys of S between */
/* the other nodes of the vicinity */
if  MLeft ∩ MRight ≠ ∅ then
    d := ChooseAny(MLeft ∩ MRight);
    ⟨X_{b-1}, x_{b-1}, ..., x_1, X_0⟩ := Compress^b(L_{b-1}, l_{b-1}, ..., L_0, l_0, LeftOf(S, d));
    ⟨Y_0, y_1, ..., y_{b-1}, Y_{b-1}⟩ := Compress^0(RightOf(S, d), r_0, R_0, ..., r_{b-1}, R_{b-1});
    for i := 1 to b − 1 do
        Replace(FL_i, ⟨L_i⟩, ⟨X_i⟩);
        Replace(Fl_i, ⟨l_i⟩, ⟨x_i⟩);
        Replace(FR_i, ⟨R_i⟩, ⟨Y_i⟩);
        Replace(Fr_i, ⟨r_i⟩, ⟨y_i⟩);
    od
    Replace(FR_0, ⟨R_0⟩, ⟨Y_0⟩);
    Replace(Fr_0, ⟨r_0⟩, ⟨d⟩);
    if FL_0 = F then
        Replace(F, ⟨L_0, l_0, S⟩, ⟨X_0⟩);
    else
        Replace(F, ⟨S⟩, ⟨X_0⟩);
        Replace(Fl_0, ⟨l_0⟩, ⟨x_1⟩);
        Replace(FL_0, ⟨X_1, x_1, L_0⟩, ⟨X_1⟩);
    fi
    return F;
fi

/* Move left: */
/* Move to the left part of the vicinity */
/* as much keys from S as possible */
if MLeft ≠ ∅ then
    if MDelimL ∩ MLeft ≠ ∅ then d := ChooseAny(MDelimL ∩ MLeft);
    else                         d := ChooseMax(MLeft); fi
    ⟨X_{b-1}, x_{b-1}, ..., x_1, X_0⟩ := Compress^b(L_{b-1}, l_{b-1}, ..., L_0, l_0, LeftOf(S, d));
    Y := RightOf(S, d);
    for i := 1 to b − 1 do
        Replace(FL_i, ⟨L_i⟩, ⟨X_i⟩);
        Replace(Fl_i, ⟨l_i⟩, ⟨x_i⟩);
    od
    Replace(FL_0, ⟨L_0⟩, ⟨X_0⟩);
    Replace(Fl_0, ⟨l_0⟩, ⟨d⟩);
    Replace(F, ⟨S⟩, ⟨Y⟩);
    if WW(Y) then return F; fi
    S := Y;
    MakeVicinity(S);
    ComputeSets(S);
fi

/* Move right: */
/* Move to the right part of the vicinity */
/* as much keys from S as possible */
if MRight ≠ ∅ then
    if MDelimR ∩ MRight ≠ ∅ then d := ChooseAny(MDelimR ∩ MRight);
```

```
    else                                   d := ChooseMin(MRight); fi
    X := LeftOf(S, d);
    ⟨Y_0, y_1, ..., y_{b-1}, Y_{b-1}⟩ := Compress^0(RightOf(S, d), r_0, R_0, ..., r_{b-1}, R_{b-1});
    for i := 1 to b − 1 do
        Replace(FR_i, ⟨R_i⟩, ⟨Y_i⟩);
        Replace(Fr_i, ⟨r_i⟩, ⟨y_i⟩);
    od
    Replace(FR_0, ⟨R_0⟩, ⟨Y_0⟩);
    Replace(Fr_0, ⟨r_0⟩, ⟨d⟩);
    Replace(F, ⟨S⟩, ⟨X⟩);
    if WW(X) then return F; fi
    S := X;
    MakeVicinity(S);
    ComputeSets(S);
fi

/* Split: */
/* Even after moving out of S all keys that fit into the */
/* other nodes of the vicinity S is still too large, and */
/* need to be split into two nodes */
if MDelim ≠ ∅ then d := ChooseAny(MDelim);
else                   d := ChooseMax(MDelimR); fi
X := LeftOf(S, d);
Y := RightOf(S, d);
Replace(F, ⟨S⟩, ⟨X, d, Y⟩);
if WW(Y) then return F; fi

/* Recursion: */
/* Even after splitting off a node X the remaining */
/* part Y is still too large, */
/* Balance_W() will be applied to Y again */
F := Balance_W(Y);
return F;
EndProcedure

Procedure Balance_B(S)
MakeVicinity(S);

/* First Compression: */
/* Check which of the first b sweeps of the vicinity is */
/* compressible, and compress the first one found */
for m := 0 to b − 1 do
    if ¬IC(Sweep_b^m(S)) then break; fi
od

if m < b then /* a compressible sweep is found, compress it */
    ⟨X_{b−m−1}, x_{b−m−1}, ..., x_1, X_0, y_0, Y_0, y_1, ..., y_{m−1}, Y_{m−1}⟩ :=
        Compress^{b−m}(Sweep_b^m(S));
    for i := 1 to b − m − 1 do
        Replace(FL_i, ⟨L_i⟩, ⟨X_i⟩);
        Replace(Fl_i, ⟨l_i⟩, ⟨x_i⟩);
    od
    for i := 0 to m − 1 do
        Replace(FR_i, ⟨R_i⟩, ⟨Y_i⟩);
        Replace(Fr_i, ⟨r_i⟩, ⟨y_i⟩);
    od
    if FL_0 = F then
        Replace(F, ⟨L_0, l_0, S⟩, ⟨X_0⟩);
```

```
    else
        Replace(F, ⟨S⟩, ⟨X_0⟩);
        Replace(Fl_0, ⟨l_0⟩, ⟨x_1⟩);
        Replace(FL_0, ⟨X_1, x_1, L_0⟩, ⟨X_1⟩);
    fi
    m := m + 1;
    S := X_0;
    MakeVicinity(S);
fi

/* Second Compression: */
/* Among the remaining sweeps of the vicinity check which */
/* one is compressible, and compress the first one found */
for m := m to b do
    if ¬IC(Sweep_b^m(S)) then break; fi
od

if m > b then return F; fi
/* A compressible sweep is found, compress it */
⟨X_{b-m-1}, x_{b-m-1}, ..., x_1, X_0, y_0, Y_0, y_1, ..., y_{m-1}, Y_{m-1}⟩ := Compress^{b-m}(Sweep_b^m(S));
for i := 0 to b - m - 1 do
    Replace(FL_i, ⟨L_i⟩, ⟨X_i⟩);
    Replace(Fl_i, ⟨l_i⟩, ⟨x_i⟩);
od
for i := 1 to m - 1 do
    Replace(FR_i, ⟨R_i⟩, ⟨Y_i⟩);
    Replace(Fr_i, ⟨r_i⟩, ⟨y_i⟩);
od
if FR_0 = F then
    Replace(F, ⟨S, r_0, R_0⟩, ⟨Y_0⟩);
else
    Replace(F, ⟨S⟩, ⟨Y_0⟩);
    Replace(Fr_0, ⟨r_0⟩, ⟨y_1⟩);
    Replace(FR_0, ⟨R_0, y_1, Y_1⟩, ⟨Y_1⟩);
fi
/* It is proven that not more than 2 nodes of the */
/* vicinity can be shrunken */
return F;
EndProcedure

Procedure Balance_C(S)
MakeVicinity(S);

/* If S has too few keys we glue it with one of */
/* the nearest neighbors, and apply Balance_W */
/* to the result if necessary */
if |F| > 0 then /* at least one key is in F */
  if FL_0 = F then
    X := [L_0, l_0, S];
    Replace(F, ⟨L_0, l_0, S⟩, ⟨X⟩);
  else /* FR_0 = F, since F contains at least one key */
    X := [S, r_0, R_0];
    Replace(F, ⟨S, r_0, R_0⟩, ⟨X⟩);
  fi
else /* there is no keys in F only the reference to S */
  if |FL_0| > 1 then
    X := [L_0, l_0, S];
    Replace(F, ⟨S⟩, ⟨X⟩);
```

```
    Replace(Fl_0, ⟨l_0⟩, ⟨l_1⟩);
    Replace(FL_0, ⟨L_1, l_1, L_0⟩, ⟨L_1⟩);
  else /* |FR_0| > 1 */
    X := [L_0, l_0, S];
    Replace(F, ⟨S⟩, ⟨X⟩);
    Replace(Fr_0, ⟨r_0⟩, ⟨r_1⟩);
    Replace(FR_0, ⟨R_0, r_1, R_1⟩, ⟨R_1⟩);
  fi
fi

if ¬WW(X) then
  F := Balance_W(Y);
fi
return F;
EndProcedure
```

Research Center for Information Systems, Program Systems Institute, Pereslavl–Zalessky, 152140 Russia

E-mail address: shv@namesys.botik.ru