$S(1)$-trees: Space Saving Generalization of B-Trees with $1/2 - \varepsilon$ Utilization

Konstantin V. Shvachko
Research Center for Information Systems
Program Systems Institute, Pereslavl-Zalessky, Russia
E-mail: shv@namesys.botik.ru
http://namesys.botik.ru/~shv

January 29, 1998

Abstract

B-trees have currently become standard data structure for representing dynamic dictionaries in external memory. The main disadvantage of *B*-trees is that they lead to an exhaustive waste of memory, when lengths of records stored in a *B*-tree differ greatly from each other. The paper suggests a new type of balanced trees which generalize traditional *B*-trees, and avoid their drawback mentioned above. The new data structure, called $S(1)$-trees, restricts not the number of records per node, but rather their total length. We define utilization of $S(1)$-trees which is different from that of *B*-trees, and prove the $1/2 - \varepsilon$ lower bound for it, where ε is inversely proportional to the tree branching. Logarithmic running time algorithms for search, insertion and deletion are presented. $S(1)$-trees can be considered as an alternative to the classical *B*-trees for the majority of dynamic dictionary applications.

1 Introduction

The problem of maintaining dynamic dictionaries is a classical problem of the theory of data structures. As usual, "dynamic dictionary" means a data structure for storing the dictionary elements, called keys (records), together with algorithms of access to, insertion, and deletion of a key.

Several tree data structures have been developed that provide for efficient solutions to the problem. Since access time to a key in a tree-like structure depends on height of the tree, the balanced trees are preferable. However, fully balanced trees occur rarely. Therefore, so called weak balance conditions were considered that lead to approximately equal access time for different tree keys rather than exactly equal, as in case of the fully balanced trees.

Probably, the first data structure for the problem was studied by G.M.Adel'son-Vel'skii and E.M.Landis [AVL62], [F73]. The approach is based on binary trees that satisfy the following balance condition. For any tree node the difference between the heights of its two child subtrees is at most 1. These trees, called AVL-trees, allow of logarithmic time algorithms of search, insertion and deletion, and are appropriate for maintaining dictionaries in the internal memory.

Another variant of week balance condition is used in 2-3 trees, which for the first time were examined by J.Hopcroft in 1970 (unpublished, see [LD91]). In contrast to AVL-trees all leaf nodes of a 2-3 tree are located at the same level of the tree. Balancing here is achieved by varying the out-degrees of the internal nodes, which can be either 2 or 3 [GS78], [Y78]. It is interesting to note that any 2-3 tree can be simply transformed into an AVL-tree.

B-trees, proposed by R.Bayer [B72], [BM72], generalize the notion of 2-3 trees. They have the same regular structure, but the out-degree of the tree internal nodes may range between $q+1$ and $2q+1$, for a fixed parameter q, called the tree order. From a practical viewpoint B-trees are suitable for maintaining dictionaries in external memory. While a number of variants of B-trees have been examined ([K71], [W86], [TF82], [W93]) all of them are based on the same idea of counting the number of keys in (equivalently, the number of children of) a node.

What is important for us is that with at least q and at most $2q$ keys in each node, any n-vertex B-tree is going to hold no less than qn, and no more than $2qn$ keys. In other words we say that the tree utilization is lower–bounded by $1/2$, where utilization is the ratio of the total number of keys, contained in the tree, to the maximal number $2qn$ of keys that can contain a B-tree with n nodes.

The disadvantages of B-trees have been widely discussed (see e.g. [K71], [PS92], and [S93]) in the context of this space lower bound. The bottom line is that B-trees utilize memory well only when the keys are of (almost) identical length, while otherwise they lead to an exhaustive waste of memory.

Probably for the first time, the problem that using B-trees for storing variable length keys is wasteful was mentioned in [K71] (with a reference to an unpublished result of Thomas H. Martin). The idea leads to a modification of B-trees that allows however to keep the algorithms almost unchanged. This idea was precisely formulated, developed, and analyzed in [PS92].

In [PS92] it is also suggested a new tree data structure, called B-trees with unfixed key length, which is based on that it restricts the total sum of lengths of keys in a node rather than the number of keys per node. These trees allow of logarithmic time algorithms of search, insertion and deletion.

It was proven that any such tree with n nodes must contain a key set of total length greater than $pn/4$, where p is the constant, which sets the maximum for total length of keys in one node. In other words, the tree utilization exceeds $1/4$.

Utilization for the new trees is different from the utilization of B-trees. In our case utilization is the ratio of the total length of keys, contained in the tree, to the maximal length np of keys that can contain such a tree with n nodes.

Utilization computed for B-trees in that way, can not be bounded by any constant greater than 0. It is easy to show that if the lengths of keys differ from each other drastically, the utilization of length for B-trees can drop to any desirable (or rather

undesirable) number, say, to 1/10.

This paper generalizes the notion of B-tree with unfixed key length by introducing $S(1)$-trees (read as sweep-one-trees). $S(1)$-trees combine the structure of the unfixed key length trees with high branching of usual B-trees. It is shown that B-trees and 2-3 trees are the special cases of $S(1)$-trees.

The $1/2 - \varepsilon$ utilization lower bound is proven for $S(1)$-trees, where ε is inversely proportional to the tree branching. So the same utilization value as for B-trees is achieved, but in case of $S(1)$-trees it means the 'pure' utilization, that is, no restrictions on the key length is assumed. The lower bound is shown to be unimprovable.

The running time bounds for $S(1)$-trees remain unchanged compared to the case of B-trees with unfixed key length.

[M77] is another work that deals with variable length keys. In [M77] a strategy for allocating keys, having variable lengths, across nodes of a B^*-tree is proposed. "This strategy results in shallow trees with fast access time". The idea is to put short keys nearer to the root, and the longest keys nearer to the leaves, in order to obtain higher branching of the tree internal nodes. McCreight's ideas can be used in implementations for making the basic algorithms for $S(1)$-trees more efficient on average.

D.Wood in [W93] emphasizing the importance of B^+-trees – a modification of B-trees whose internal part forms a B-tree, – says that "today, they are considered to be the only data structure to use for large external files, when the total ordering is important" (p.393), and also that "for external dictionaries, the B^+-tree is the only choice" (p.392). However $S(1)$-trees appear to be more efficient than traditional B-trees for many applications. Historically for the first time $S(1)$-trees were implemented in the Starset programming language for representing set data aggregates [G94].

The paper is organized as follows. In Section 2 we remind the reader the definition of B-trees, and show their inefficiency in case of variable key length. Then we introduce our data model, and compare the definitions of the two tree data structures. Starting from simple examples, and considering height 1 trees, we prove in Section 3 our main result: the $1/2-\varepsilon$ lower bound of utilization of $S(1)$-trees. Algorithms for the three basic operations over $S(1)$-trees, described in Section 4, are based on a rather simple idea that in order to restore broken balance of a node it is necessary to consider its neighbors, and to relocate keys within the nodes if possible.

2 Basic definitions

2.1 Trees

Definition 2.1 *Let K be a finite set of elements called keys. Define* tree *inductively:*

1. *Object Λ, called empty tree, is a tree,*
2. *If $k_1, \ldots, k_m \in K$ and $T_0, T_1, \ldots, T_m$ are trees, then tuple $\langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle$ is also a tree.*

By *subtree* of non-empty tree $T = \langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle$ we mean T itself and any subtree of any tree T_i $(0 \leq i \leq m)$.

If S and R are subtrees of T, and R is a subtree of S, then R is called a *descendant* of S in T, and S is called an *ancestor* of R in T. If S and R are subtrees of T, such that $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ and $S_i = R$ for some i, then R is called the *i-th direct descendant of* S, and S is called the *direct ancestor of* R in T.

A *graph* is a very natural and customary representation for this kind of tree. For each tree T a (labeled) graph $G(T) = \langle V(T), E(T) \rangle$ that represents it, consists of the vertex set $V(T)$, that is, the set of all non-empty subtrees of T, and the set of edges $E(T)$, which is defined in the following way. If S and R are non-empty subtrees of T, such that R is the i-th direct descendant of S, then the pair $\langle S, R \rangle$ is an edge labeled by number i. Each vertex $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ in the tree is labeled by the sequence of keys $\langle l_1, \ldots, l_m \rangle$.

The *out-degree* of a vertex is the number of its **non-empty** direct descendants. Vertex S is called a *leaf* if its out-degree is 0. Vertex $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ is an *internal vertex* if its out-degree equals $m + 1$. Vertex T is called a *root* of tree T.

We say that vertex F is a *common ancestor* of vertices S and R, iff F is an ancestor for both S and R. F is said to be a *nearest common ancestor* of vertices S and R, if F is a common ancestor of S and R, and if none of the common ancestors for the vertices S, R is a descendant of F.

A sequence of natural numbers $(i_1, \ldots, i_n)$ is called a *path of length* n in tree T, iff there exists a sequence of non-empty subtrees $S_0, S_1, \ldots, S_n$ of T, such that $S_0 = T$ and each S_j is the i_j-th direct descendant of $S_{j-1} (j = 1, \ldots, n)$. We say in this case that the path leads to vertex S_n, and length n is said also to be a *level* of S_n in T.

Height of a tree is its maximal path length.

For each tree T, let $K(T)$ denote the set of keys that T contains:

$$K(\Lambda) = \emptyset, \qquad K(\langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle) = \{k_1, \ldots, k_m\} \cup K(T_0) \cup \ldots \cup K(T_m)$$

Let $k(T)$ denote the set of keys that the root vertex of tree T contains:

$$k(\Lambda) = \emptyset, \qquad k(\langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle) = \{k_1, \ldots, k_m\}$$

Thus for each vertex S of a tree, $K(S)$ denotes the set of keys that are contained in subtree S, and $k(S)$ is the set of keys from $K(S)$ that does not belong to any proper subtree of S.

2.2 B-trees

Definition 2.2 *Let $\ll$ be a linear order on a set of keys K. Let $q > 0$ be a natural number. A tree T is called* B-tree of order q *iff either $T = \Lambda$, or $T \neq \Lambda$ and the following conditions hold:*

1. *For each vertex $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ of T*

   $$l_1 \ll \ldots \ll l_m \quad \text{and} \quad \forall i (1 \leq i \leq m \Rightarrow (\forall k \in K(S_{i-1}))(\forall k' \in K(S_i))(k \ll l_i \ll k'))$$

2. *Each vertex of tree T is either an internal vertex or a leaf*

3. *All paths in T from the root to leaves have equal length*

4. *For each vertex* S *of* T $\quad |k(S)| \leq 2q$

5. *For each non-root vertex* S $\quad |k(S)| \geq q$

Proposition 2.1 *If* T *is an* n*-vertex* B*-tree of order* q*, then* $|K(T)| - |k(T)| \geq q(n-1)$.

This important property of B-trees is a consequence of Definition 2.2. Together with Proposition 2.2 it makes B-trees applicable for representing ordered data.

The three basic operations of *search*, *insertion* and *deletion* of a key in a tree can be efficiently performed for B-trees. The classical algorithms for the operations are well known (see [K71], [W86], [TF82], [LD91]), and all of them have logarithmic time complexity.

Proposition 2.2 *If* B*-tree* T *has* n *vertices, then search, insertion and deletion can be performed in time* $O(logn)$ *for it.*

2.3 B-trees with variable length keys

B-trees are used when amount of information to be maintained is large and can not be stored in internal memory. In this case all information is stored on disk. The machine can work with only a part of data at a moment, reading it every time from disk by pieces of fixed size, called blocks.

A typical problem when the data structure of B-trees is used is the problem of maintaining dictionaries. Dictionary is a set of keys. Initially the dictionary is empty. The problem is to support three main operations: search, insertion to, and deletion from the dictionary.

Generally, the dictionary can store records of very arbitrary structure. For simplicity let us suppose that keys are words in this section.

In practice it is usually supposed that word length is bounded by some constant. For B-trees it means that blocks are composed of cells of fixed capacity equal to the maximal word length. Words are stored in cells – one word per cell. As words can have different lengths up to the maximal possible, some cells can be filled not completely. Our aim is to capture this loss of storage resource.

Consider a B-tree where each cell can contain 8 symbols, but the input for the dictionary mostly consists of four-letter words. In this case most cells are half filled, that is, a tree node is at most half filled on average even if it contains the maximal number of words. When the upper bound of word length is small, then the amount of space lost is not critical. But if we allow words of length 100 as well as of length 1, then the loss of efficiency may be too big.

Formally. Let a set of keys K be the set $\Sigma^{(l)}$ of all words of length not exceeding l. Let an n-vertex B-tree T contain $|K(T)|$ words of the total length $g(T)$. A tree T can not contain more than $2qn$ words. So the sum of their lengths is at most $2qnl$. If the average word length is, for example, $l/10$ then the total length of words in T is bounded by the following values

$$\frac{qnl}{10} \leq g(T) \leq \frac{qnl}{5}$$

We see that Proposition 2.1 does not reflect the real situation adequately. And we conclude that it is reasonable to use B-trees when the length of keys (words) is almost constant. For example, if the dictionary consists of words of equal length, i.e., $K = \Sigma^l$.

We define here a tree data structure, which generalizes B-trees, and is intended for efficient representation of arbitrary key sets. Informally, the difference between B-trees and the new trees, called $S(1)$-trees, is that nodes of a $S(1)$-tree are not composed of cells of fixed capacity, but rather store keys one by one without any extra space between them. Thus instead of bounding the number of keys, we bound the total length of the keys in a node.

For this kind of trees we can give a more adequate definition of utilization, which is the total length of keys of a $S(1)$-tree divided by the maximal capacity of a $S(1)$-tree, having the same number of nodes as the given tree.

It is important that utilization defined for B-trees in this way can not be bounded by any constant greater than zero.

2.4 $S(1)$-trees

Let μ be a *weight function* that maps each key $k \in K$ to its weight $\mu(k)$, which is a positive natural number. If $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ is a vertex of tree T then its *weight* is $\mu(S) = \sum_{i=1}^{m} \mu(l_i)$. A *complete weight* of the tree is the sum $M(T) = \sum \mu(S)$ of weights of all vertices of T.

Let T be a tree that satisfies the first three conditions (1, 2, 3), called *structural properties*, of Definition 2.2 of a B-tree. We introduce a *neighboring relation* on the vertex set of T as follows. Let L and R be vertices of the same level of the tree. Consider a set of keys $I = K(L) \cup K(R)$. Then vertex L is said to be a *left neighbor* of vertex R, and R is said to be a *right neighbor* of L, iff there exists a **unique** key k belonging to the key set $K(T) \setminus I$ that satisfies the following property

$$(\forall l \in K(L))(\forall r \in K(R))(l \ll k \ll r)$$

Key k is called a *delimiting key* for neighboring nodes L and R. We say also that k *separates* the neighbors.

Note, that the delimiting key for any pair of neighboring nodes belongs to the nearest common ancestor of the neighbors.

Informally, the nodes L and R are neighbors, iff they are located at one level of the tree, and no other tree node of the same level are located between L and R.

Definition 2.3 *Let $\ll$ be a linear order on a key set K, and μ be a weight function on K. Let $p > 0$ and $q > 0$ be natural numbers. A tree T is called* $S(1)$-tree of rank p and order q *iff either $T = \Lambda$ or $T \neq \Lambda$ and the following conditions hold:*

1. *For each vertex $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ of T*

   $$l_1 \ll \ldots \ll l_m \text{ and } \forall i (1 \leq i \leq m \Rightarrow (\forall k \in K(S_{i-1}))(\forall k' \in K(S_i))(k \ll l_i \ll k'))$$

2. *Each vertex of tree T is either an internal vertex or a leaf*

3. *All paths in T from the root to leaves have equal length*

4. For each vertex S of T its weight $\mu(S) \leq p$

5. For each pair of neighboring vertices L and R with delimiting key k

$$\mu(L) + \mu(k) + \mu(R) > p$$

6. For each non-root vertex S $\quad |k(S)| \geq q$

$S(1)$-tree is a generalization of B-tree in the following sense.

Example 2.1 Consider weight function μ_0 that identically equals 1 on K. Then each B-tree of order q will be a $S(1)$-tree of rank $2q$ and order q with the weight function μ_0. Let's verify conditions 4 and 5 of Definition 2.3.

(4) $\mu_0(S) = |k(S)| \leq 2q$.

(5) Consider a pair of neighboring nodes L and R with delimiting key k

$$\mu_0(L) + \mu_0(k) + \mu_0(R) = |k(L)| + 1 + |k(R)| \geq q + 1 + q > 2q$$

2.5 Comparing the data structures

Formally, in B-trees (Definition 2.2) we count the number of keys in each tree node, and demand this number to be at most $2q$. $2q$ is the node capacity here, which means that a node can not hold more than the given number of keys.

Condition 5 of Definition 2.2, called the balance condition, is intended to provide the following two basic properties of B-trees. First, it guarantees that all (non-root) nodes of any B-tree are at least half filled, that is, the number of keys allowed in a node varies between q and $2q$. Second, it provides high branching of the tree internal nodes, which varies between $q + 1$ and $2q + 1$.

In $S(1)$-trees (Definition 2.3) we take into account mostly not the number of keys in a node but rather their total weight. This total weight is upper-bounded (condition 4) with constant p, which is the tree rank, and which determines the weight capacity of the tree nodes.

The balance condition 5 of Definition 2.3 is formulated not locally for each node as in case of B-trees, but in terms of neighboring nodes. It provides high density of a $S(1)$-tree in general even if some of the nodes are almost empty. Such nodes in the tree are "balanced" with their neighbors, which must be almost full in this case.

Condition 5 at the same time can not guarantee high branching of the $S(1)$-tree internal nodes. That is why condition 6 is used, which states a lower bound for the number of keys, stored in one node, as in B-trees. The condition forces the tree to branch intensively, thus making it "shorter", but "wider". We will see later how this affects the space lower bounds and basic algorithms.

It is important to note that for $S(1)$-trees the tree order q doesn't depend on the node capacity so strictly, as in case of B-trees. The order can vary between 1 and $\left\lfloor \frac{p}{2} \right\rfloor$.

Therefore, in $S(1)$-trees the two properties of high density and high branching are not combined in one condition, and this is the main reason why the class of $S(1)$-trees is broader than that of B-trees.

Another reason for it is that keys in a node can be counted by means of an appropriate weight function, like μ_0 from Example 2.4, where B-trees are shown to be the special case of $S(1)$-trees.

As mentioned above the balance condition for $S(1)$-trees is formulated in terms of neighboring nodes, unlike B-trees, where the balance condition can be checked within one node.

Our ongoing research is to analyze $S(b)$-trees (pronounced as sweep-be-trees) for an arbitrary parameter b. Intuitively, parameter b here determines the number of neighbors of a node that should be examined in order to check the balance condition.

For B-trees, as we know, this number equals 0. From this point of view one can consider B-trees of order q as $S(0)$-trees of rank $2q$, order q, and weight function $\mu_0 \equiv 1$ on K.

For $S(1)$-trees the number of neighbors is 1, that is, for any node one should check that the node is balanced with both of its neighbors.

For $b = 2$, the balance condition should be applied to the following three vicinities of any node of an $S(2)$-tree:

1. the vicinity, composed of the current node, its left neighbor, and the left neighbor of the left neighbor;

2. the vicinity, composed of the current node, and its left and right neighbors;

3. the vicinity, composed of the current node, its right neighbor, and the right neighbor of the right neighbor.

For arbitrary b, the number of vicinities that should be checked for each node is at most $b+1$, and the number of neighbors and neighbors of neighbors, and so on, is exactly b. So 'sweep' here means the number of neighboring nodes that must be "seen" from each of the tree nodes.

3 Space lower bounds for $S(1)$-trees

3.1 Simple bounds

In this section we analyze the space efficiency of rank p $S(1)$-trees. Our complexity measure, called *utilization* and denoted by Δ, is a ratio of the total weight $M(T)$ of a given tree T to the maximal possible weight np of any n-vertex $S(1)$-tree of rank p

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

Our aim is to build a lower bound for utilization Δ. The simplest, but the most general, space lower bound for utilization of $S(1)$-trees gives

Proposition 3.1 *Let T be an n-vertex $S(1)$-tree of rank p. Then*

$$\Delta(T) \geq \frac{1}{p}$$

Proof. Each vertex of the tree contains at least one key, whose weight is at least one. Thus for any vertex S its weight $\mu(S) \geq 1$. Then the total weight of T is $M(S) \geq n$, and its utilization is $\Delta(T) = \frac{M(T)}{np} \geq \frac{1}{p}$. □

Proposition 3.2 *Let T be an n-vertex $S(1)$-tree. If $n \geq 3$ then*

$$\Delta(T) > \frac{1}{4}$$

Proof. Consider a partition of the set of leaves of T into disjoint pairs of neighboring nodes $\{L_i, R_i\}$, $i = 1, \ldots, s$. Let k_i be a delimiting key for each pair $\{L_i, R_i\}$. Then

$$M(T) \geq \sum_{i=1}^{s} (\mu(L_i) + \mu(k_i) + \mu(R_i)) > sp$$

An out-degree of any internal node of a $S(1)$-tree is at least 2. Thus the number of leaves x of the tree is greater than the number of its internal nodes, i.e.

$$x > \frac{n}{2}$$

On the other hand, since the pairs of neighboring nodes $\{L_i, R_i\}$ are disjoint, then the number of pairs is

$$s = \left\lfloor \frac{x}{2} \right\rfloor$$

This implies that $s \geq \frac{n}{4}$ and $M(T) > \frac{np}{4}$. That is,

$$\Delta(T) = \frac{M(T)}{np} > \frac{1}{4}$$

□

This is also a simple lower bound. And this is the best we can get in the case, when the tree order q is not restricted. But as we will see below the $\frac{1}{4}$ lower bound is attainable only for the very special case of $S(1)$-trees. In more regular situation we can obtain better bounds.

3.2 The main theorem

Let us consider two important examples.

Example 3.1 Let $T = \langle T_0, k, T_1 \rangle$ be a $S(1)$-tree consisting of $n = 3$ vertices. Then by Definition 2.3, $M(T) = \mu(T_0) + \mu(k) + \mu(T_1) > p$ and hence $\Delta(T) > 1/3$.

Example 3.2 Let a tree $T = \langle T_0, k_1, T_1, k_2, T_2 \rangle$ have $n = 4$ vertices. Let their weights be $\mu(k_1) = \mu(k_2) = 1$, i.e., $\mu(T) = 2$, and let $\mu(T_0) = \mu(T_2) = 1$, and $\mu(T_1) = p$. This tree is a $S(1)$-tree and we may see that

$$\Delta(T) = \frac{\mu(T) + \mu(T_0) + \mu(T_1) + \mu(T_2)}{4p} = \frac{1}{4} + \frac{1}{p}$$

For large enough values of p ($p > 12$) the following inequality holds

$$1/4 < \Delta(T) < 1/3$$

As our next step in deriving a more general theorem let's consider trees of height 1.

Lemma 3.3 *Let an n-vertex $S(1)$-tree T of rank p be of height 1. Then*

$$M(T) \geq \left\lfloor \frac{n-1}{2} \right\rfloor p + n - 2$$

Proof. Fix $n \geq 3$, and let $m = n - 2$.

Consider an n-vertex tree $S = \langle S_0, l_1, S_1, \ldots, l_m, S_m \rangle$ of height 1 with the following weights of its vertices: $\mu(l_i) = 1$ for all $i = 1, \ldots, m$, i.e., weight of the root is $\mu(S) = m$; even leaves are of weight $\mu(S_{2i}) = 0$, and odd leaves are of weight $\mu(S_{2i+1}) = p$ ($i = 0, \ldots, \lfloor m/2 \rfloor$). Note that S is not a $S(1)$-tree, since even descendants of the root are empty trees, while odd are not. So root S is neither a leaf, nor an internal vertex. This contradicts Definition 2.3 (c. 2). It is easy to see that $M(S) = \left\lfloor \frac{n-1}{2} \right\rfloor p + n - 2$. We are going to prove that for any n-vertex $S(1)$-tree $T = \langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle$ of rank p and of height 1, its weight $M(T)$ is at least $M(S)$.

From the set $V = \{T_0, T_1, \ldots, T_m, T_{m+1}\}$ of all vertices of T, where $T_{m+1} = T$, we choose a subset W consisting of those vertices T_i whose weights $\mu(T_i)$ are less than weights $\mu(S_i)$ of the corresponding vertices of tree S.

$$W = \{T_i \in V \mid \mu(T_i) < \mu(S_i)\}$$

If $W = \emptyset$, then $M(T) \geq M(S)$. Let $W \neq \emptyset$, then the following two properties hold:

1. $T \notin W$,
2. if $T_i \in W$, then i is odd.

So let $W = \{T_{i_1}, \ldots, T_{i_r}\}$, where indices $0 < i_j \leq m$ ($j = 1, \ldots, r$) are odd numbers. Consider the sum $M(T)$ and group addends in the following way

$$M(T) = [\mu(T_{i_1-1}) + \mu(k_{i_1}) + \mu(T_{i_1})] + \cdots + [\mu(T_{i_r-1}) + \mu(k_{i_r}) + \mu(T_{i_r})] + \Sigma_0(T)$$

Combine addends of the sum $M(S)$ similarly

$$M(S) = [\mu(S_{i_1-1}) + \mu(l_{i_1}) + \mu(S_{i_1})] + \cdots + [\mu(S_{i_r-1}) + \mu(l_{i_r}) + \mu(S_{i_r})] + \Sigma_0(S)$$

Let us compare the two sums. First, consider $\Sigma_0(T)$, which is the weight sum of vertices that do not belong to W, plus weights of a number of keys from the root of T. According to the definition of set W, and since the keys from the root of S have minimal possible weight 1, we get

$$\Sigma_0(T) \geq \Sigma_0(S)$$

Second, for all $j = 1, \ldots, r$

$$\mu(T_{i_j-1}) + \mu(k_{i_j}) + \mu(T_{i_j}) \geq p + 1 = 0 + 1 + p = \mu(S_{i_j-1}) + \mu(l_{i_j}) + \mu(S_{i_j})$$

This means that $M(T) \geq M(S)$. □

Corollary 3.4 *Let $d \geq 2$ be the out-degree of the root node of a rank p $S(1)$-tree T having height 1. Then*

1. *For any even d* $\quad \Delta(T) \geq \frac{1}{2}\frac{d}{d+1} + \frac{1}{p}\frac{d-1}{d+1}$
2. *For any odd d* $\quad \Delta(T) \geq \frac{1}{2}\frac{d-1}{d+1} + \frac{1}{p}\frac{d-1}{d+1}$
3. *If $d \neq 3$, then* $\quad \Delta(T) \geq \frac{1}{3} + \frac{1}{3p}$
4. *If $d = 3$, then* $\quad \Delta(T) \geq \frac{1}{4} + \frac{1}{2p}$
5. *For all d* $\quad \Delta(T) \geq \left(\frac{1}{2} + \frac{1}{p}\right)\frac{d-1}{d+1}$

Proof. Estimates 1, 2 follow from the definition of utilization Δ, by substituting $n = d+1$ to the bound (see Lemma 3.3) for $M(T)$ and discovering the meaning of integer division for the cases of, respectively, even and odd d's.

Estimates 3, 4 are the special cases of estimates 1, 2, and estimate 5 is their generalization. $\square$

The following simple combinatorial inequality will be actively used below.

Lemma 3.5 *Let $\varepsilon, a_i, b_i (i = 1, \ldots, m)$ be positive real numbers such that $a_i/b_i \geq \varepsilon$. Then*

$$\sum_{i=1}^{m} a_i \Big/ \sum_{i=1}^{m} b_i \geq \varepsilon$$

Proof. $a_i \geq \varepsilon b_i$. Hence, $\sum_{i=1}^{m} a_i \geq \varepsilon \sum_{i=1}^{m} b_i$. $\square$

Theorem 3.6 *Let T be an n-vertex $S(1)$-tree of rank p and order q.*

1. *In general case, for $n > p/2$* $\quad \Delta(T) > \frac{1}{4}$
2. *If none of the internal vertices of T has out-degree 3, then for $n > p$* $\quad \Delta(T) > \frac{1}{3}$
3. *If $q > 3$, then for $n > (q+1)p/2$* $\quad \Delta(T) > \frac{1}{2}\frac{q}{q+2}$

Proof. Consider the following partition of the vertex set V of T. First choose subsets V_i $(i = 1, \ldots, s_1)$. Each V_i consists of all leaves of T outcoming from a common direct ancestor and of the ancestor itself. Consider now a tree T' obtained from the initial tree T by discarding the vertices that belong to the union of the sets V_i. Let us consider a set of leaves of T' and their predecessors, and construct new subsets V_{s_1+i} $(i = 1, \ldots, s_2)$ from them in the same manner as it was done for tree T. Throw out selected vertices from T' and continue the process until either the remaining tree is empty or it consists of the unique root vertex T. Suppose that the subsets $V_1, \ldots, V_s$ have been built to that moment. Then in the former case (when the empty tree is what remains) let $V_{s+1} = \emptyset$, and in the latter case let $V_{s+1} = \{T\}$. Hence we got a partition $V = V_1 \cup \cdots \cup V_s \cup V_{s+1}$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$.

Each vertex set V_i $(i \leq s)$ together with the edges that connect them in T determines a $S(1)$-tree of height 1. For referring to these trees we will use the same symbols V_i as for their vertex sets. Let $n_i = |V_i|$. Then $\sum_{i=1}^{s+1} n_i = n$ and also $\sum_{i=1}^{s+1} M(V_i) = M(T)$.

Let us prove estimate 3. Let $a = \frac{d-1}{d+1}$, where $d = q + 1 > 4$ is a lower bound for the out-degrees of internal vertices of T. Using the above, and estimate 5 from Corollary 3.4 we conclude that utilization of a height 1 $S(1)$-tree is majorized by a monotonically increasing function of the out-degree of the tree root. Therefore if the out-degree of the root of V_i $(i \leq s)$ is not less than d, then for $\varepsilon = a\left(\frac{1}{2} + \frac{1}{p}\right)$ we have

$$\frac{M(V_i)}{n_i p} \geq \varepsilon$$

Consider the case when $V_{s+1} \neq \emptyset$, that is, the height of T is even. Then

$$\begin{aligned} \Delta(T) &= \frac{M(T)}{np} = \frac{\sum_{i=1}^{s} M(V_i) + \mu(T)}{np} \geq \frac{\sum_{i=1}^{s} M(V_i) + 1}{np} = \\ &= \frac{\sum_{i=1}^{s} M(V_i) + (pa/2+1) - pa/2}{np} = \frac{\sum_{i=1}^{s} M(V_i) + (pa/2+1)}{\sum_{i=1}^{s} n_i p + p} - \frac{a}{2n} \end{aligned}$$

Since $\frac{pa/2+1}{p} > \varepsilon$ and $\frac{M(V_i)}{n_i p} \geq \varepsilon$ $(i \leq s)$, then using Lemma 3.5 we have estimate 3 of the theorem. For $n > p/2$

$$\Delta(T) \geq \frac{a}{2} + \frac{a}{p} - \frac{a}{2n} > \frac{1}{2}a = \frac{1}{2}\frac{q}{q+2}$$

Now consider the case when $V_{s+1} = \emptyset$, that is, the height of T is odd.

If the out-degree of the root node of T is not less than d, then by Lemma 3.5 and Corollary 3.4 we easily get

$$\Delta(T) = \frac{M(T)}{np} = \frac{\sum_{i=1}^{s} M(V_i)}{\sum_{i=1}^{s} n_i p} \geq \varepsilon > \frac{1}{2}a$$

If the out-degree of the root node of $T = \langle T_0, k_1, T_1, \ldots, k_m, T_m \rangle$ is less than d, that is, $m < q$, then by estimate 5 of Corollary 3.4

$$\frac{M(V_i)}{n_i p} \geq \varepsilon$$

for all $i < s$, and by estimate 4 of the corollary we have

$$\frac{M(V_s)}{n_s p} > \frac{1}{4}$$

where $n_s = m + 2$.

Utilization in this case is

$$\begin{aligned} \Delta(T) &= \frac{M(T)}{np} = \frac{\sum_{i=1}^{s-1} M(V_i) + M(V_s)}{np} = \\ &= \frac{\sum_{i=1}^{s-1} M(V_i) + (n_s pa/2 + M(V_s)) - n_s pa/2}{np} = \frac{\sum_{i=1}^{s-1} M(V_i) + (n_s pa/2 + M(V_s))}{\sum_{i=1}^{s-1} n_i p + n_s p} - \frac{an_s}{2n} \end{aligned}$$

We see that

$$\frac{n_s pa/2 + M(V_s)}{n_s p} > \frac{a}{2} + \frac{1}{4} > \varepsilon$$

since $p \geq 2q\mu_{max}(K)$ and $q > 3$, that is, $\frac{a}{p} < \frac{1}{p} < \frac{1}{6} < \frac{1}{4}$.

Now using Lemma 3.5, and the fact that $m \leq d - 2$, for all $n > dp/2$ we obtain

$$\Delta(T) \geq \varepsilon - \frac{an_s}{2n} = \frac{a}{2} + \frac{a}{p} - \frac{a(m+2)}{2n} > \frac{1}{2}a$$

This completes the proof of estimate 3.

Analogously using Lemma 3.5 and Corollary 3.4 we can prove estimates 1 and 2 for the values of $\varepsilon = \frac{1}{4} + \frac{1}{2p}$ and $\varepsilon = \frac{1}{3} + \frac{1}{3p}$, respectively. Finally the following lower bounds can be obtained:

1) For all $n > p/2$ $\quad \Delta(T) \geq \frac{1}{4} + \frac{1}{2p} - \frac{1}{4n} > \frac{1}{4}$

2) For all $n > p$ $\quad \Delta(T) \geq \frac{1}{3} + \frac{1}{3p} - \frac{1}{3n} > \frac{1}{3}$ $\quad \square$

Summary:

In the general case (estimate 1), when the out-degree of internal nodes is not restricted, 1/4 is the lower bound of Proposition 3.2. And as it can be seen from Example 3.2 this estimate is almost sharp. But the trees with that low utilization have very specific

structure. Namely, almost all of its internal nodes should have out-degree 3. Otherwise the lower bound rises up to 1/3 (estimate 2). And, finally, estimate 3 guarantees that the more highly branching the tree is the higher its lower bound of utilization. The limit as the out-degree goes to infinity of these lower bounds is 1/2, which is the upper bound of lower bounds for utilization of $S(1)$-trees.

Corollary 3.7 *Let K be a key set. Then for any $\varepsilon > 0$ there exist two parameters p, and q, such that for any $S(1)$-tree of rank p order q having $n > (q+1)p/2$ nodes its utilization is*

$$\Delta(T) > \frac{1}{2} - \varepsilon$$

Proof. Let $q = 1$ if $\varepsilon \geq \frac{2}{3}$, and let $q = \frac{2}{\varepsilon} - 2$ otherwise.
If $q = 1$, then by Theorem 3.6

$$\Delta(T) > \frac{1}{4} > 0 > \frac{1}{2} - \varepsilon$$

If $q > 1$, then using again the Theorem we get

$$\Delta(T) > \frac{1}{2}\frac{q}{q+2} = \frac{1}{2} - \frac{2}{q+2} = \frac{1}{2} - \varepsilon$$

□

4 Algorithms

4.1 Searching

The *search* operation for $S(1)$-trees is performed in the usual way for tree data structures. We will not present a detailed search algorithm here, but will give an informal description of it. The search algorithm for a key k in a $S(1)$-tree T consists of at most logarithmically many steps of local search in a tree node. It begins from the root node. For each node $S = \langle S_0, k_1, S_1, \ldots, k_m, S_m \rangle$ the search algorithm scans the key set of S in order to find (using e.g. binary search algorithm) a key k_i which is greater than or equal (according to the linear order $\ll$ on the set of keys K) to the given key k. If $k_i = k$ then the search is finished, otherwise the computations are continued. If all the keys from $k(S)$ are less then k, then the next vertex to be explored is vertex S_m, in other cases it should be vertex S_{i-1}. We repeat the procedure with each new vertex we descend into until either an equal key is found, or a leaf is reached. If key k is not found in the leaf then the search is finished, and we return as our result that the key is not found.

The running time of the algorithm is at most the height h of the tree. But h does not exceed binary logarithm of the number of the tree vertices, since the out-degree of each vertex is at least 2. This proves

Proposition 4.1 *If $S(1)$-tree T has n vertices, then the search time is at most $\log n$.*

We will refer to this algorithm as procedure SEARCH(T,k). The result of the procedure is a three-tuple ($TrueValue, S, k_i$). $TrueValue$ determines whether key k is found in T. S is the vertex examined last by the procedure, and k_i is a key from S that is greater than or equal to k. If all keys in S are less than k then SEARCH returns a special value which is denoted by k_{m+1}, where $m = |k(S)|$. If the key is not in T then the point of insertion is returned, and value of $TrueValue$ indicates that the search failed. This allows the use of the same procedure for search, insertion, and deletion.

4.2 Insertion

Generally the class of $S(1)$-trees is not closed under *insertions*.

Example 4.1 Let the $S(1)$-tree T of rank p contain a unique key k of weight $\mu(k) = c > p/2$. Try to insert a key $l \neq k$ of the same weight $\mu(l) = c$ to T. It is impossible, since there does not exist a $S(1)$-tree consisting of exactly two keys whose weights exceed half of the tree rank p.

In the same manner it is not possible to construct a correct $S(1)$-tree of rank p and order q, containing $2q$ keys of weight $c > p/q$ each.

Let $\mu_{max}(K) = \max\{\mu(k) \mid k \in K\}$. For each key set K we will consider only $S(1)$-trees of rank $p \geq 2q\mu_{max}(K)$, where q is the tree order. The insertion algorithm presented below guaranties that the class of $S(1)$-trees of that rank is closed under insertions.

Informally the insertion algorithm is as follows.

First it verifies by procedure SEARCH(T,k) whether the given key k is already contained in $S(1)$-tree T. If that is the case then insertion is finished. If not, then procedure SEARCH returns a leaf $S = \langle S_0, k_1, S_1, \ldots, k_m, S_m \rangle$ and a key k_i in it, before which key k must be inserted. Next, k is put to the given place of vertex S. If weight of S is small enough to contain additional key k, that is, if $\mu(S) + \mu(k) \leq p$, then insertion is done. Otherwise weight of the extended vertex S should be greater than p. For restoring the structure of the $S(1)$-tree (Definition 2.3 c.4) a *balancing* of S is needed. The balancing leads to the special (local) transformations of the initial tree T. Starting from leaf S transformations are applied only to those vertices of T that are on the path from the root T to S or to their neighbors. Note that generally insertion and balancing of the tree may increase the tree height by 1.

The main part of the insertion algorithm is a procedure-function BALANCE_I. Its input parameter S is the vertex that must be balanced, and its output is the direct ancestor of the balanced node.

Balancing of the input node S is performed by relocating the keys, contained in S and its neighbors. The neighbors and the parents of S are the local variables of procedure BALANCE_I.

Let L be the left neighbor of S and $lkey$ be the key that separates L and S. If the left neighbor does not exist then L is defined to be an empty node (denoted $L = \langle\rangle$, and $lkey$ by definition is a special key, whose weight is $\mu(lkey) = p + 1$.

Let R be the right neighbor of S and $rkey$ denote their delimiting key. If the right neighbor of S does not exist, then $R = \langle\rangle$, and $rkey$ is the special key of weight $\mu(rkey) = p + 1$.

Let F denote the direct ancestor of S, and let FL and FR denote the ancestors of S that contain the delimiting keys $lkey$ and $rkey$, respectively. If the delimiting keys belong

to the same vertex F, then F, FL, and FR denote one and the same vertex. If S is the root, then we suppose $F = FL = FR = \langle\rangle$.

Thus at the beginning of procedure BALANCE_I there are six variables defined for the vertices of the tree.

$$\begin{array}{lcl} F & = & \langle F_0, f_1, F_1, \ldots, S, \ldots, f_x, F_x\rangle \\ FL & = & \langle \ldots, lkey, \ldots\rangle \\ FR & = & \langle \ldots, rkey, \ldots\rangle \\ S & = & \langle S_0, k_1, S_1, \ldots, k_m, S_m\rangle \\ L & = & \langle L_0, l_1, L_1, \ldots, l_y, L_y\rangle \\ R & = & \langle R_0, r_1, R_1, \ldots, r_z, R_z\rangle \end{array}$$

Procedure BALANCE_I begins with forming the four subsets of the key set $k(S)$. These subsets *MLeft*, *MRight*, *MDelimL* and *MDelimR* determine the computational process and are defined by the following properties.

1. A key k from $k(S)$ belongs to *MLeft* iff the sum of the weights of vertex L, delimiting key $lkey$, and the keys from $k(S)$ that are left of k in S, exceeds p. Conceptually, keys in *MLeft* are those that would not fit if we tried to move as many keys as possible into L from S.

2. A key k from $k(S)$ belongs to *MRight* iff the sum of the weights of vertex R, delimiting key $rkey$, and the keys from $k(S)$ that are right of k in S, exceeds p. Conceptually, keys in *MRight* are those that would not fit if we tried to move as many keys as possible into R from S.

3. A key k from $k(S)$ belongs to *MDelimL* iff the total weight of the keys left of k in S is at most p, and the number of these keys is at least q. Conceptually, a key is in *MDelimL* if when resizing S one can select it as the $rkey$ for a validly sized vertex consisting of the members to the left of it.

4. A key k from $k(S)$ belongs to *MDelimR* iff the total weight of the keys right of k in S is at most p, and their number is at least q. Conceptually, a key is in *MDelimR* if when resizing S one can select it as the $lkey$ for a validly sized vertex consisting of the members to the right of it.

Let us define also $MDelim = MDelimL \cap MDelimR$. Sets $\overline{MLeft}$, and $\overline{MRight}$ denote the complements of the respective subsets of $K(S)$.

The following easy statements characterize the intrinsic properties of the sets defined and guarantee the correctness of algorithm BALANCE_I.

5. $(\forall l, k \in k(S))(l \ll k \,\&\, l \in MLeft \Rightarrow k \in MLeft)$

6. $(\forall k, r \in k(S))(k \ll r \,\&\, r \in MRight \Rightarrow k \in MRight)$

7. $(\forall l, k, r \in k(S))(l \ll k \ll r \,\&\, l \in MDelim \,\&\, r \in MDelim \Rightarrow k \in MDelim)$

8. Statements like 7 are also valid for sets *MDelimL* and *MDelimR*.

9. $MDelim \neq \emptyset$

10. $\overline{MLeft} \cap \overline{MRight} = \emptyset \Rightarrow \overline{MLeft} \subseteq MRight \,\&\, \overline{MRight} \subseteq MLeft$

11. $MDelimL \cap \overline{MLeft} = \emptyset \Rightarrow MDelimL \subseteq MLeft$

12. $MDelimR \cap \overline{MRight} = \emptyset \Rightarrow MDelimR \subseteq MRight$

13. $MDelimL \cap \overline{MLeft} = \emptyset \;\&\; MDelimR \cap \overline{MRight} = \emptyset \Rightarrow MDelim \subseteq MLeft \cap MRight$

Procedure BALANCE_I examines consecutively all possible cases of relationship between the sets. It distinguishes four (AI, BI, CI, DI) basic cases of sets relationship.

AI examines the case, when a key d can be chosen from node S, such that

1. left neighbor L joined with delimiting key $lkey$, and the part of S, which is left of d, forms a correct node of a $S(1)$-tree, and

2. right neighbor R joined with delimiting key $rkey$, and the part of S, which is right of d, forms a correct node of a $S(1)$-tree.

In this case the initial nodes L, S and R with delimiting keys $lkey$ and $rkey$ are replaced in the tree by two new nodes with delimiting key d.

BI is the case, when a key satisfying both of the two properties of case AI can not be chosen, but there exists a key d in S that satisfies the first of them. In this case the two neighbors L and S together with delimiting key $lkey$ are replaced by two new vertices composed of keys contained in L and S, and separated by d.

Case CI is symmetrical to BI.

In case DI vertex S doesn't have a key satisfying any of the two properties mentioned above. It means that S can be broken into two well-formed vertices. Here vertex S is replaced by the two new vertices, and their delimiting key is inserted to the direct ancestor F of S.

Properties 1–13 guarantee that the new nodes, formed by the procedure, will be balanced. Replacing the original delimiting keys $lkey$ and $rkey$ as in cases AI, BI, CI, or inserting a new delimiting key as in case DI, the algorithm should change also the ancestors FL, F, FR. The renewed vertex F will be returned as the result of procedure BALANCE_I.

Note that while balancing is not finished, the transformed tree can be not a $S(1)$-tree. For example, one of the tree nodes can have weight that exceeds p, or can contain no keys at all. The latter can happen with node F when case AI of the procedure is implemented. Indeed, suppose that initially F contains a unique key (valid for $q = 1$), say it is the $rkey$. Then after reallocating the keys of vertex S over two neighbors L and R, and removing the (unique) key from F, it becomes a node, whose key set is empty. But this doesn't mean that the renewed node F is empty, as it has a (unique) descendant.

After finishing the insertion procedure the structure of the $S(1)$-tree will be fully recovered.

For describing the algorithms we use the following instrumental procedures and functions.

– Procedure $Replace(S, P, Q)$ replaces a part of vertex S (of the tree), that coincides with P, with Q. E.q., $Replace(S, \langle k \rangle, \langle l \rangle)$ means substitution of key l for key k in vertex S, and $Replace(S, S, Q)$ replaces the whole subtree S of the tree with Q. If Q is an empty object in $Replace(S, P, Q)$ then it means deletion of part P of vertex S.

– Function $Catenate(...)$ with variable number of arguments joins all the arguments in one node. E.g.

$$Catenate(\langle S_0, k_1, S_1\rangle, l, \langle R_0, r_1, R_1\rangle)$$

defines vertex $\langle S_0, k_1, S_1, l, R_0, r_1, R_1\rangle$.

– Functions $LeftOf(S, k)$ and $RightOf(S, k)$ define two parts of vertex S that are on the left and on the right, respectively, of key k. E.q., if $S = \langle S_0, k_1, S_1, k_2, S_2\rangle$, then $LeftOf(S, k_2) = \langle S_0, k_1, S_1\rangle$, and $RightOf(S, k_2) = \langle S_2\rangle$

– S^* denotes a direct ancestor of vertex S. Thus $F = S^*$ in the procedure.

Procedure BALANCE_I(S)
$MLeft := \{k \in k(S) \mid \sum_{l \in k(S), l \ll k} \mu(l) + \mu(lkey) + \mu(L) > p\}$;
$MRight := \{k \in k(S) \mid \sum_{r \in k(S), k \ll r} \mu(r) + \mu(rkey) + \mu(R) > p\}$;
$MDelimL := \{k \in k(S) \mid \sum_{l \in k(S), l \ll k} \mu(l) \leq p \;\&\; |\{r \in k(S) \mid k \ll r\}| \geq q\}$;
$MDelimR := \{k \in k(S) \mid \sum_{r \in k(S), k \ll r} \mu(r) \leq p \;\&\; |\{l \in k(S) \mid l \ll k\}| \geq q\}$;
$MDelim := MDelimL \cap MDelimR$;
$\overline{MLeft} := k(S) \setminus MLeft$;
$\overline{MRight} := k(S) \setminus MRight$;

AI: **if** $\overline{MLeft} \cap \overline{MRight} \neq \emptyset$ **then**
/** choose element from the intersection **/
$d \in \overline{MLeft} \cap \overline{MRight}$;
$Replace(L, L, Catenate(L, lkey, LeftOf(S, d)))$;
$Replace(R, R, Catenate(RightOf(S, d), rkey, R))$;
AI1: **if** $FL = F = FR$ **then**
$Replace(F, \langle lkey, S, rkey\rangle, \langle d\rangle)$;
AI2: **else if** $FL \neq F$ **then**
$Replace(FL, \langle lkey\rangle, \langle d\rangle)$;
$Replace(F, \langle S, rkey, R\rangle, \langle R\rangle)$;
AI3: **else if** $F \neq FR$ **then**
$Replace(FR, \langle rkey\rangle, \langle d\rangle)$;
$Replace(F, \langle L, lkey, S\rangle, \langle L\rangle)$;
fi fi fi
return(F);
fi

BI: **if** $MDelimL \cap \overline{MLeft} \neq \emptyset$ **then**
/*** $MDelimL \cap \overline{MLeft} \subseteq MRight$ ***/
/** choose element from the intersection **/
$d \in MDelimL \cap \overline{MLeft}$;
$Replace(L, L, Catenate(L, lkey, LeftOf(S, d)))$;
$Replace(S, S, RightOf(S, d))$;
$Replace(FL, \langle lkey\rangle, \langle d\rangle)$;
return(F);
fi

CI: **if** $MDelimR \cap \overline{MRight} \neq \emptyset$ **then**
/*** $MDelimR \cap \overline{MRight} \subseteq MLeft$ ***/

```
        /** choose element from the intersection **/
        d ∈ MDelim ∩ ¬MRight;
        Replace(R, R, Catenate(RightOf(S, d), rkey, R));
        Replace(S, S, LeftOf(S, d));
        Replace(FR, ⟨rkey⟩ , ⟨d⟩);
        return(F);
      fi

DI:   if MDelim ∩ MLeft ∩ MRight ≠ ∅ then
        /** choose element from the intersection **/
        d ∈ MDelim ∩ MLeft ∩ MRight;
        Replace(F, ⟨S⟩ , ⟨LeftOf(S, d), d, RightOf(S, d)⟩);
        return(F);
      fi
End_of_Procedure
```

Now we are ready to investigate the insertion procedure, called INCLUSION. After testing (by means of SEARCH) whether the given key should be added to the given tree, and after inserting of the key to the leaf node, specified by SEARCH, procedure INCLUSION starts balancing the tree nodes that lay on the path from the tree root to the enlarged leaf S. The balancing begins from S and proceeds further up to the root node.

In addition to working on the current level of vertex S and replacing S and its neighbors, the algorithm will also modify the ancestors of S (F, FL, FR). This can break to its turn the balance conditions for the ancestors. Namely, the weight of the ancestor, say F, may exceed p, and then we must use procedure BALANCE_I for balancing F. The weight of renewed vertex F may also decrease, compared to its original weight, and become less than is necessary to satisfy together with one of F's neighbors the balance condition 5 of Definition 2.3. In this case we use procedure BALANCE_D.

Precisely, procedure BALANCE_D is used, when either the weight of F plus the weight of one of its neighbors together with weight of the delimiting key does not exceed p, or the number of keys in F less than q. BALANCE_D will be described in the next section.

Computation is stopped after reaching the tree root. Thus the algorithm examines all the nodes that lay on the path from the tree root to the leaf, containing the new key, and balances them if necessary. Only these nodes and their neighbors in the tree can be transformed by the algorithm. The resulting tree would be accessible via variable X.

```
Procedure INCLUSION(T,k)
(TrueValue, S, k_i) := SEARCH(T,k);
/*** S = ⟨Λ, k_1, Λ, ..., k_{i-1}, Λ, k_i, Λ, ..., k_m, Λ⟩ ***/
if TrueValue = TRUE then return(T); fi
Replace(S, ⟨k_i⟩ , ⟨k, Λ, k_i⟩);
/*** S = ⟨Λ, k_1, Λ, ..., k_{i-1}, Λ, k, Λ, k_i, Λ, ..., k_m, Λ⟩ ***/
do
   X := S;
   if μ(S) > p then S := BALANCE_I(S);
   else             S := BALANCE_D(S);
while(S ≠ ⟨⟩);
return(X);
End_of_Procedure
```

The following proposition is a consequence of the insertion algorithm, presented above.

Proposition 4.2 *If $S(1)$-tree T has n vertices, then time of insertion is $O(\log n)$.*

4.3 Deletion

Similarly to the insertion, the *deletion* procedure begins with search. Given a key k and a $S(1)$-tree T, procedure SEARCH looks for k in T. If T does not contain k, then deletion is finished. Otherwise SEARCH returns vertex $S = \langle S_0, k_1, S_1, \ldots, k_m, S_m \rangle$ and a key $k_i = k$ in it that must be deleted.

The case when S is not a leaf can easily be reduced to the case of deletion from leaf. Indeed, if S is not a leaf, then $S_i \neq \Lambda$. Let's replace key k_i in S by the minimal (according to the order $\ll$ on K) key k' from set $K(S_i)$. It is clear that k' belongs to some leaf S', which is the leftmost node of the subtree S. If the replacement of the key k_i in S by k' breaks the balance conditions for S, then they will be restored while balancing the tree. For finding minimal key k' and leaf S', containing it, function $MinKey(S_i)$ is used.

Now let S be a leaf. The deletion procedure removes the given key from S and proceeds to balancing the tree.

The main part of the deletion algorithm is a procedure-function BALANCE_D(S) that balances vertex S and returns the direct ancestor of S. The balancing in this case aims (as opposed to procedure BALANCE_I) at reconstructing the structure of the $S(1)$-tree when condition 5 of Definition 2.3 does not hold, or the number of keys of the input vertex is less than q (condition 6). We use the same notations for the neighbors of the input vertex, their delimiting keys, and the ancestors that were used in procedure BALANCE_I.

For simplicity we present here the variant of procedure BALANCE_D, which is valid for all values of tree order $q \geq 2$, but which is not valid for the case of $q = 1$.

BALANCE_D distinguishes four (AD, BD, CD, DD) basic cases of broken vertex balance.

AD examines the case, when the total weight of nodes L, S and R with two delimiting keys $lkey$ and $rkey$ is not greater than p. Here all of the three nodes should be joined into one new node.

BD is the case, when the total weight of neighboring nodes L and S plus weight of their delimiting key $lkey$ is not greater than p. In this case the two neighboring nodes are replaced in the tree by the new one joining vertices L and S.

Case CD is symmetrical to BD.

In case DD, when vertex S can not be catenated with any of its neighbors, and when the number of its keys is less than q, the algorithm joins vertex S with that of its neighbors, whose direct ancestor is common with S, and for the new vertex procedure BALANCE_I is called.

Procedure BALANCE_D(S)
AD: **if** $\mu(L) + \mu(lkey) + \mu(S) + \mu(rkey) + \mu(R) \leq p$ **then**
  $Q := Catenate(L, lkey, S, rkey, R)$;
  if $FL = F = FR$ **then**
    $Replace(F, \langle L, lkey, S, rkey, R \rangle, \langle Q \rangle)$;
  fi
  /*** $F = \langle F_0, f_1, F_1, \ldots, S, \ldots, f_x, F_x \rangle$ ***/
  if $FL \neq F$ **then** /* $F = FR$ */
    $Replace(L, L, Q)$;
    $Replace(FL, \langle lkey \rangle, \langle f_2 \rangle)$;

```
        Replace(F, F, RightOf(F, f_2));
      fi
      if F ≠ FR then    /* FL = F */
        Replace(R, R, Q);
        Replace(FR, ⟨rkey⟩, ⟨f_{x-1}⟩);
        Replace(F, F, LeftOf(F, f_{x-1}));
      fi
      return(F);
    fi

BD: if μ(L) + μ(lkey) + μ(S) ≤ p then
      Q := Catenate(L, lkey, S);
      if FL = F then
        Replace(F, ⟨L, lkey, S⟩, ⟨Q⟩);
      else    /* F = FR */
        Replace(L, L, Q);
        Replace(FL, ⟨lkey⟩, ⟨rkey⟩);
        Replace(F, F, RightOf(F, rkey));
      fi
      return(F);
    fi

CD: if μ(S) + μ(rkey) + μ(R) ≤ p then
      Q := Catenate(S, rkey, R);
      if F = FR then
        Replace(F, ⟨S, rkey, R⟩, ⟨Q⟩);
      else    /* FL = F */
        Replace(R, R, Q);
        Replace(FR, ⟨rkey⟩, ⟨lkey⟩);
        Replace(F, F, LeftOf(F, lkey));
      fi
      return(F);
    fi

DD: if F = ⟨⟩ then
      if |k(S)| = 0 then    /* S = ⟨S_0⟩ */
        return(S_0);
      else
        return(F);
      fi
    fi
    if |k(S)| < q then
      if FL = F then
        Q := Catenate(L, lkey, S);
        Replace(F, ⟨L, lkey, S⟩, ⟨Q⟩);
      else    /* F = FR */
        Q := Catenate(S, rkey, R);
        Replace(F, ⟨S, rkey, R⟩, ⟨Q⟩);
      fi
      return(F);
    fi
End_of_Procedure
```

Here is the deletion procedure.

Procedure DELETION(T,k)

```
(TrueValue, S, k_i) := SEARCH(T, k);
/*** S = ⟨S_0, k_1, S_1, ..., S_{i-1}, k_i, S_i, ..., k_m, S_m⟩ ***/
if TrueValue = FALSE then return(T); fi
if S_0 ≠ Λ then    /* S is not a leaf */
   (S', k') := MinKey(S_i);
   Replace(S, ⟨k⟩, ⟨k'⟩);
   S = S';
   k = k';
fi
/*** S = ⟨Λ, k_1, Λ, ..., k_{i-1}, Λ, k, Λ, k_i, Λ, ..., k_m, Λ⟩ ***/
Replace(S, ⟨Λ, k, Λ⟩, ⟨Λ⟩);
do
   X := S;
   if μ(S) > p then S := BALANCE_I(S);
   else            S := BALANCE_D(S);    fi
while(S ≠ ⟨⟩);
return(X);
End_of_Procedure
```

The following proposition is a consequence of the deletion algorithm.

Proposition 4.3 *If $S(1)$-tree T has n vertices, then time of deletion is $O(logn)$.*

Acknowledgements

I am grateful to Alexei Semenov for his permanent attention to my scientific work, Mikhail Gilula for supporting this research, Hans Reiser, and Konstantin Gorbunov for fruitful discussions and helpful remarks. I would also like to thank George Gottlob for the so very relevant literature he kindly sent me.

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

[C79] D. Comer, *The Ubiquitous B-tree*, Comp. Surv., vol.11, 2, 1979, pp.121–137

[F73] C.C. Foster, *A Generalization of AVL-Trees*, Communications of the ACM, vol.16, 1973, pp.513–517

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

[GS78] L.J. Guibas, R. Sedgewick, *A Dichromatic Framework for Balanced Trees*, Proceedings, 19-th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp.8–12

[GS86] G.K. Gupta, B. Srinivasan, *Approximate Storage Utilization of B-tree*, Inf. Proc. Lett. vol.22, 1986, pp.243–246

[H85] S.-H.S. Huang, *Height-Balanced Trees of Order(β,γ,α)*, ACM Trans. Database Syst., vol.10, 2, 1985, pp.261–284

[JS89] T. Johnson, D. Shasha, *Utilization of B-tree with Inserts, Deletes, and Modifies*, In Proceedings of the ACM Principles of Database Systems Symposium, 1989, pp.235–244

[K71] D.E. Knuth, *The Art of Computer Programming*, vol.3 (Sorting and Searching), Addison–Wesley, Reading, MA 1973

[L84] C.H.C. Leung, *Approximate Storage Utilization of B-tree: a Simple Derivation and Generalizations*, Inf. Proc. Lett., vol.19, 1984, pp.199–201

[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

[RS81] A.L. Rosenberg, L. Snyder, *Time- and Space-Optimality in B-tree*, ACM Trans. Database Syst., vol.6, 1, 1981, pp.174–193

[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

[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 Structure*, Prentice-Hall, Englewood Cliffs, NJ 1986

[W93] D. Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley Publishing Company, 1993

[W85] W.E. Wright, *Some Average Performance Measure for the B-tree*, Acta Inf., vol.21, 1985, pp.541–557

[Y78] A.C.-C. Yao, *On Random 2-3 Trees*, Acta Inf., vol.9, 1978, pp.159–170