# Skip trees, an alternative data structure to Skip lists in a concurrent approach *

Xavier Messeguer †

May 20, 1997

**Abstract**

We present a new type of search trees, called Skip trees, which are a generalization of Skip lists. To be precise, there is a one-to-one mapping between the two data types which commutes with the sequential update algorithms.

A Skip list is a data structure used to manage data bases which stores values in a sorted way and in which it is insured that the form of the Skip list is independent of the order of updates by using randomization techniques. Skip trees inherit all the properties of Skip lists, including the time bounds of sequential algorithms.

The algorithmic improvement of the Skip tree type is that a concurrent algorithm *on the fly* approach can be designed. Among other advantages, this algorithm is more compressive than the one designed by Pugh for Skip lists and accepts a higher degree of concurrence because it is based on a set of local updates.

From a practical point of view, although the Skip list should be in the main memory, Skip trees can be registered into a secondary or external storage. Therefore we analyse the ability of Skip trees to manage data bases in comparison with B-trees.

Nous presentons un nouveau type d'arbres, que nous appellons Skip trees, les quels sont une généralisation des Skip lists. Concrètement, il existe un isomorphisme entre eux qui commute avec les algorithmes séquentiels.

Une Skip list est une structure de données ordonnées qui sert pour manager des base de données. Une propriété intéressante est que la configuration des Skip lists ne résulte pas de l'ordre d'introduction des donnes parce qu'on utilise des techniques aléatoires. Des Skip trees héritent toutes les propriétés des Skip lists, surtout les bornes des algorithmes séquentiels.

Mieux encore, l'utilisation des Skip trees nous permet de construire un algorithme concurrent *on the fly*. Cette solution offre de grands avantages: l'algorithme est plus compréhensible que celui construit par Pugh pour les Skip lists, et il accepte plus de concurrence parce que les transformations sont locales.

D'un point de vue pratique, malgré que les Skip lists doivent être enregistrées en mémoire centrale, les Skip trees peuvent être stockés sur un support secondaire. En

*This research was supported by the ESPRIT BRA Program of the EC under contract no. 7141, project ALCOM II.

†Departament de Llenguatges i Sistemes Informàtics.Universitat Politècnica de Catalunya.Pau Gargallo 5, 08028-Barcelona, Spain.Contact e-mail: `messeguer@lsi.upc.es`

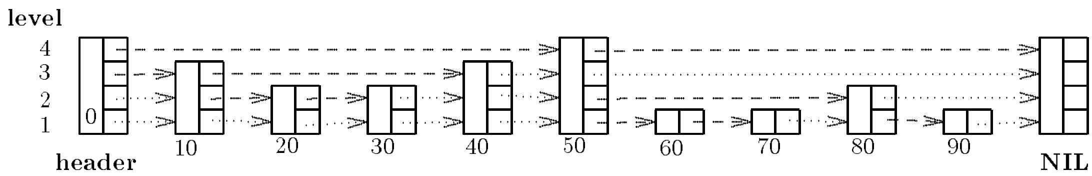


Figure 1: Skip list

conséquence, nous étudions l'habilité des Skip trees pour indexer des bases de données en comparison avec les B-trees.

# 1 Introduction

This paper follows on from the attempt to design a concurrent algorithm *on the fly* approach for Skip lists.

The Skip list [22] is a data structure that stores values in a sorted order (see Figure 1). It plays an important role because, by means of randomization, the form of the Skip list is independent of the order of updates. The expected efficiency of updates is comparable with other equilibrated structures such as AVL trees and B-trees. In the last five years there have been a great number of reports analysing its properties [3, 11, 18, 24].

The concurrent *on the fly* approach, which allows a very high degree of concurrence, is inspired by [5]. This paper proposes a Garbage collection concurrent algorithm consisting of the *main algorithm*, which dynamically puts nodes onto the structure and takes them away, and the *garbage collector*, which marks the removed nodes and gathers them together into a garbage list. The goal was to design the garbage collector with a set of local evolution rules that assume *temporal atomicity* (small number of assignments and tests) and *spatial atomicity* (a fixed small set of neighboring nodes). J.L.W. Kessels was the first to apply this approach to search trees, specifically AVL trees [10]: once new keys have been inserted, the tree is balanced with a set of local evolution rules. This approach was further applied in [1, 13, 14, 15].

The concurrent algorithm on Skip lists by W. Pugh [21] is far from being composed of local rules and needs some kind of circular pointers to maintain information. These two facts cast doubts as to the comprehensiveness and correctness of the algorithm. We think that the main difficulty derives from the sequential update algorithms: recall that the insertion and deletion ones are based on the path followed in the search stage, therefore, global information is needed to modify local data.

As the main obstacle to designing concurrent algorithms takes place in the structure of Skip lists (a set of linked lists), we propose a new data type called Skip trees, composed of trees and inspired by the path followed by the sequential search algorithm in Skip lists[1]. The improvement is that we are not obliged to store the path followed in the search process because this information has been added to nodes of trees, therefore the design of algorithms can be based on sets of local rules. Moreover, as there is a one-to-one mapping between

[1] This idea was applied by T. Papadakis in his PhD thesis [17] to introduce deterministic Skip list.

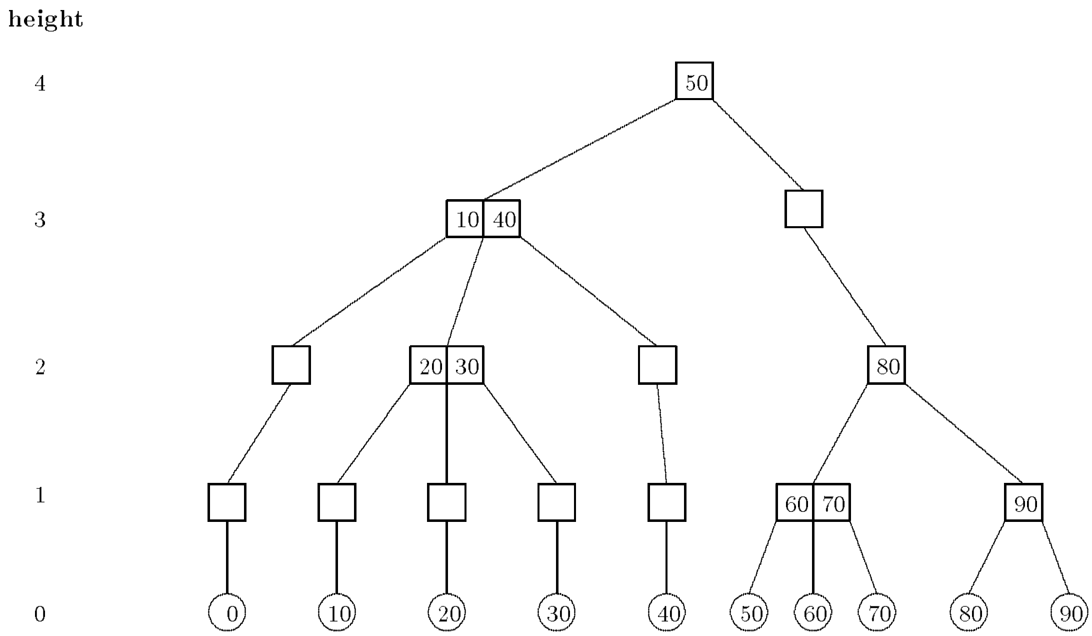


Figure 2: Skip tree

Skip lists and Skip trees that commutes with the update algorithms, the large amount of race properties of Skip lists can be transferred to Skip trees. Let us explain the main ideas of the mapping: considering the Skip list in Figure 1, we group consecutive items with the same level, such as 60 and 70, into the same node, and attach to them the nodes with a lower level (including empty ones) in a sorted order; the Skip tree show in Figure 2 is obtained.

From a dynamic point of view, as the level of each key of Skip lists is given by a random procedure, the height of a node (the height of the subtree rooted at this node) is given by the same random procedure; but it is interesting to note that the length of a node (the number of keys stored in this node) inherits a similar random distribution. Then, the Skip trees can be considered as a class of unbounded random B-trees because all the leaves have the same depth; we say *unbounded random* because the length of the nodes is unbounded and has a random distribution. Moreover, as the expected length of the nodes is proportional to a certain parameter $q : 0 < q < 1$ of the distribution, it holds that Skip trees are close to 2–3 trees for $q \simeq .5$ and close to B-trees for $q \simeq 1$.

From a practical point of view, Skip list should be in the main memory, but if the data is very large the pagination of the structure is necessary. If this pagination is based on sets of consecutive keys, then all update algorithms slow down because they have to change pages many times. Skip trees, thanks to their structure, allow more efficient pagination. Moreover, due to their relationships with B-trees, the improvements proposed by G. Diehr and B. Faaland [4] can be applied.

Recall that mappings between data structures are a well-known topic. L. Guibas and R. Sedgewick [8] embed schemes in a dichromatic frame. T. Papadakis [17] gives us a one-to-one mapping between 2–3 trees and deterministic Skip lists. Later on T. Ottmann, H. Six and D.

Wood [16] prove that there exists a one-to-one mapping that commutes with updates between AVL trees and 1–2 Brother trees.

Finally, Skip trees have the same performance rates as the random search trees by R. Seidel and C. Aragon [23]. However, random search trees should be applied in a different context, because the probability of a key is given by a continuous identically distributed random variable.

This paper has six sections. The second recalls the main properties of Skip lists. The third gives the definition of Skip trees, their local rules and the formal definition of mapping and its proof. The fourth section presents the concurrent and sequential algorithms. The fifth section analyses the ability of Skip trees to manage data bases, and the last section includes the main conclusions and the proposals for further research.

## 2 Skip lists

Skip lists are randomized data structures introduced by W. Pugh in 1990 [22]. Sequential skip list algorithms are very simple to implement, and they provide significant constant factor improvements over balanced and self-adjusting trees. Skip lists are also space efficient, requiring an average of 2 (or fewer) pointers per item and no balance, priority or weight information. Furthermore, the probability of the search time or space complexity exceeding their expected values rapidly approaches 0 as the items number in the skip list increases [24].

A non-empty Skip list (see Figure 1) consists of several non-empty sorted linked lists. All the items are stored in the list of *level* 1. Some of them also belong to the list of level 2, and so forth. Each item $x$ in $S$ has a key denoted as $\mathbf{key}(x)$ and a positive integer $\mathbf{level}(x)$. If $\mathbf{level}(x) = \ell$, it means that $x$ belongs to the linked lists of level $1, 2, \ldots, \ell$. We write $\mathbf{level}(S)$ to denote the maximum level among the levels of its items. The level of $S$ is also called as its *height*.

To implement a skip list, we need to allocate a node for each item. Each node $x$ contains the item and $\mathbf{level}(x)$ pointers. The successor of $x$ at level $l$, denoted $\mathbf{forward}(x, \ell)$, is given by the $\ell$-th *forward* pointer of $x$. A header node, $\mathbf{header}(S)$, which stores a dummy key smaller than any legal key, points to the first node of each linked list. A node called **NIL**, which stores a key greater than any legal key, is pointed by the last node of each of the linked lists.

Given a Skip list $S$ and a node $x \neq \mathbf{NIL}$ and some integer $0 \leq l \leq \mathbf{level}(x)$, we write

$$\begin{aligned}\mathbf{wall}(x, l) \quad = \quad & \text{“the first node } y \text{ to the right of } x\text{, i.e.} \\ & \mathbf{key}(x) < \mathbf{key}(y)\text{, such that } \mathbf{level}(y) > l\text{”.}\end{aligned}$$

For instance, in Figure 1, $\mathbf{wall}(\mathbf{header}(S), 3)$ is the node having key 50.

We define a *subskiplist* at node/level $(x, l)$ of $S$, denoted $S_{x,l}$ for short, as the Skip list of height $l$, where $x$ acts as a header and $\mathbf{wall}(x, l)$ acts as **NIL**. $S_{x,l}$ contains all node/levels of $S$ reachable from $(x, l)$.

We recall the sequential algorithms:

**Search:** Given a Skip list $S$ and a key $a$, the search procedure returns the unique *node* in $S$ such that $\mathbf{key}(node) < a \leq \mathbf{key}(\mathbf{forward}(node, 1))$. It works moving the key $a$ forward or down through $S$ until it reaches *node*. In any given stage the key is said to be at a

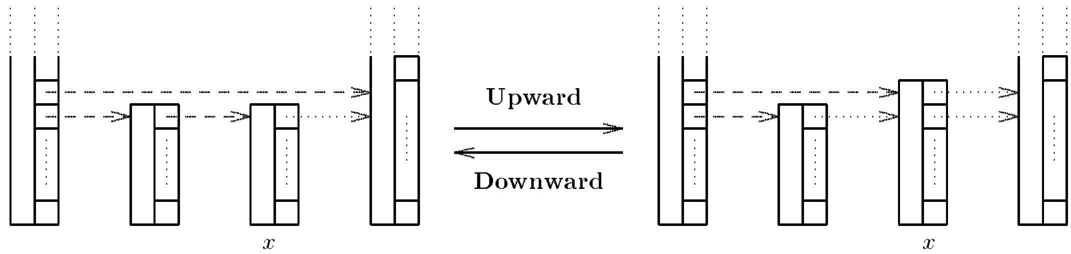


Figure 3: Rules of Skip lists

node/level $(x, \ell)$, designated the *current node/level*. Initially the current node/level is set to $(\mathbf{header}(S), \mathbf{level}(S))$. The search procedure iterates until the current level $\ell$ is 0.

**Insertion:** Assume, w.l.o.g., that the key $a$ to be inserted does not belong to $S$. The insertion has three main phases. First, we search for $a$ to locate the insertion point for the new item, but it is also necessary to collect information about the search path, namely the would-be predecessors of the new item in each list. Second, a random level is chosen and a new node is allocated for the new item. Finally, the third phase modifies the necessary links to add the new node.

The random distribution considered was the negative binomial distribution with random parameter $q$, also designated Pascal or geometric distribution, and denoted $NB(1, q)$. Recall that a negative random variable is the number of failures observed before a success in a series of independents trials, where the probability of success in a trial is $q$. Let $X$ be a random variable with this distribution $NB(1, q)$ , then $Prob\{X = k\} = p^k q$ and the expected value is $p/q$ being $p + q = 1$. [2]

The levels of nodes of a Skip lists with parameter $q$ is given by the independent distributed random variables $NB(1, q) + 1$ with expected mean $1/q$. Consequently the expected level of a Skip list of size $n$ is $O(\log_{1/p} n)$, and the expected time to search, insert or delete a key is $O(\log_{1/p} n)$ [22].

We recall the local rules of Skip lists (see Figure 3) that appear implicitly in the work of Pugh [21] about concurrent Skip lists. Assume that $a = \mathbf{key}(x)$. Firstly we recall the rule $\mathbf{upward}(S, a)$, that corresponds to the increase of the level of node $x$ by one.

$$\begin{aligned}\mathbf{upward}(S, a) \quad &= \quad \text{“increase the level of } x \text{ by one and reconstruct } S, \\ &\quad\quad \text{when } \mathbf{level}(x) = \mathbf{level}(S) \text{, increase } \mathbf{level}(S) \text{ by one”.}\end{aligned}$$

When $\mathbf{level}(x) > 0$ we can define appropriately the inverse rule, namely $\mathbf{downward}(S, a)$, that decreases the level of $x$ by one, in such a way that:

$$\mathbf{downward}(\mathbf{upward}(S, a), a) = \mathbf{upward}(\mathbf{downward}(S, a), a) = S.$$

We finally introduce the rule $\mathbf{attach}(S, a)$ and $\mathbf{unattach}(S, a)$ that attaches and unattaches key $a$ to nodes with level 1.

[2] Some authors count the number of trials instead of the number of failures (see [20]).

# 3 Skip trees

A Skip tree is a search tree. Each internal node $n$ has three registers: **height**$(n)$ gives us the height of the node, **key**$(n)$ stores a possibly empty ordered list of keys and **child**$(n)$ stores an ordered list of pointers. If **key**$(n)$ has $r$ keys, **child**$(n)$ contains $r+1$ pointers to its children. Each leaf $l$ contains only one key. The leaves do not have any children and all of them have the same depth.

Observe that the number of keys of nodes is not bounded and that we allow internal nodes without keys (empty nodes in Figure 2) having only one child.

We define four local rules that will serve us to design the algorithms. Firstly we define **split** and **join** rules starting from the split and join ones defined for 2–3 trees or B-trees but giving due attention to white nodes. Then, we define **attach** and **unattach** rules that attachs or unattach nodes to trees.

Given a Skip tree $T$ and a key $a$ belonging to node $x$ of $T$, we define:

$$\mathbf{split}(T,a) = \begin{cases} x \neq \mathbf{root}(T) & \text{the node } x \text{ is split into two more nodes and} \\ & \text{the key } a \text{ is located in the father of } x. \\ x = \mathbf{root}(T) & \text{the node } x \text{ is split into two more nodes and} \\ & \text{a new node will be created to store } a. \end{cases}$$

In both cases, white nodes will be created or propagated when necessary. The different context of $a$ in $T$ give us different types of split which are straightforward to deduce (see Figure 4).

Dually, we can define the inverse rule **join**$(T,a)$ that groups together the internal nodes located on both sides of the key $a$ into only one internal node, and puts $a$ between them. In **join**$(T,a)$ the height of $a$ decreases by one. It follows that:

$$\mathbf{join}\,(\mathbf{split}(T,a),a) = \mathbf{split}\,(\mathbf{join}(T,a),a) = T$$

We need a rule that attaches a new key $a$ to $T$.

$$\begin{aligned} \mathbf{attach}(T,a) \quad = \quad & \text{“a new leaf with value } a \text{ is hung in } T \text{ at height } 0 \\ & \text{and the key } a \text{ is located appropriately at level 1. This means,} \\ & \text{if the node is white, fulfill it with } a\text{; otherwise insert} \\ & a \text{ between the keys located at this internal node”.} \end{aligned}$$

Finally, we can easily define the inverse rule **unattach**$(T,a)$ if $a$ has height 1.

## 3.1 Mapping between Skip trees and Skip lists

We define the mapping between Skip trees and Skip lists, and we prove that it is one-to-one function and that local rules of Skip lists commute with local rules of Skip trees. These two facts suggest that the update algorithms of both data types are syntactically identical; they only differ from local calls.

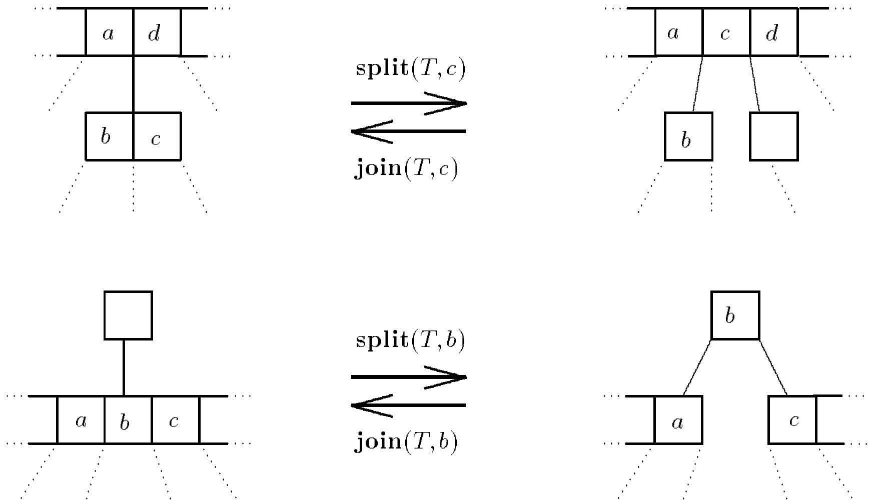


Figure 4: Splits and Joins with white nodes: in the first case when $c$ is split a white node is generated, and in the second case the split of $b$ take off the white node.

#### 3.1.1 One-to-one mapping

Before giving an accurate definition of:

$$\mathcal{T} : \text{Skip lists} \longrightarrow \text{Skip trees}$$

let us explain this top-down transformation informally (see Figure 5). There are three main cases:

- The first case happens when we have a subskiplist $S_{x,l}$ such that $0 < l < \mathbf{level}(x)$ and $\mathbf{forward}(x,l) = \mathbf{wall}(x,l)$. The transformed Skip tree $\mathcal{T}(S_{x,l})$ starts with a white node of height $l$ having as a child $\mathcal{T}(S_{x,l-1})$. Notice that the smallest key of $\mathcal{T}(S_{x,l-1})$ will be $\mathbf{key}(x)$.

- The second case corresponds to $0 < l < \mathbf{level}(x)$ and $\mathbf{forward}(x,l) \neq \mathbf{wall}(x,l)$. Therefore there is at least one node between $\mathbf{forward}(x,l)$ and $\mathbf{wall}(x,l)$. If there is only one node $y_1$ between $\mathbf{forward}(x,l)$ and $\mathbf{wall}(x,l)$, then the Skip tree $\mathcal{T}(S_{x,l})$ has an internal node with key $a_1 = \mathbf{key}(y_1)$ and two children corresponding to the transformations of $S_{x,l-1}$ and $S_{y_1,l-1}$.

- Case 3. The last case happens when we have $l = 0$ or $S_{x,0}$. Therefore we get a leaf having the key $a$ of $x$.

Formally, we define the $\mathcal{T}$ as:

**Definition 1** *Given a* Skip list $S = S(\mathbf{header}, \mathbf{NIL})$ *we define* $\mathcal{T}(S)$ *recursively starting from* $\mathcal{T}(S(\mathbf{header}, \mathbf{NIL}))$*. Given a node/level* $(x,l)$ *we consider the integer* $f \geq 1$ *such that* $\mathbf{forward}^f(x,l) = \mathbf{wall}(x,l)$*, therefore*

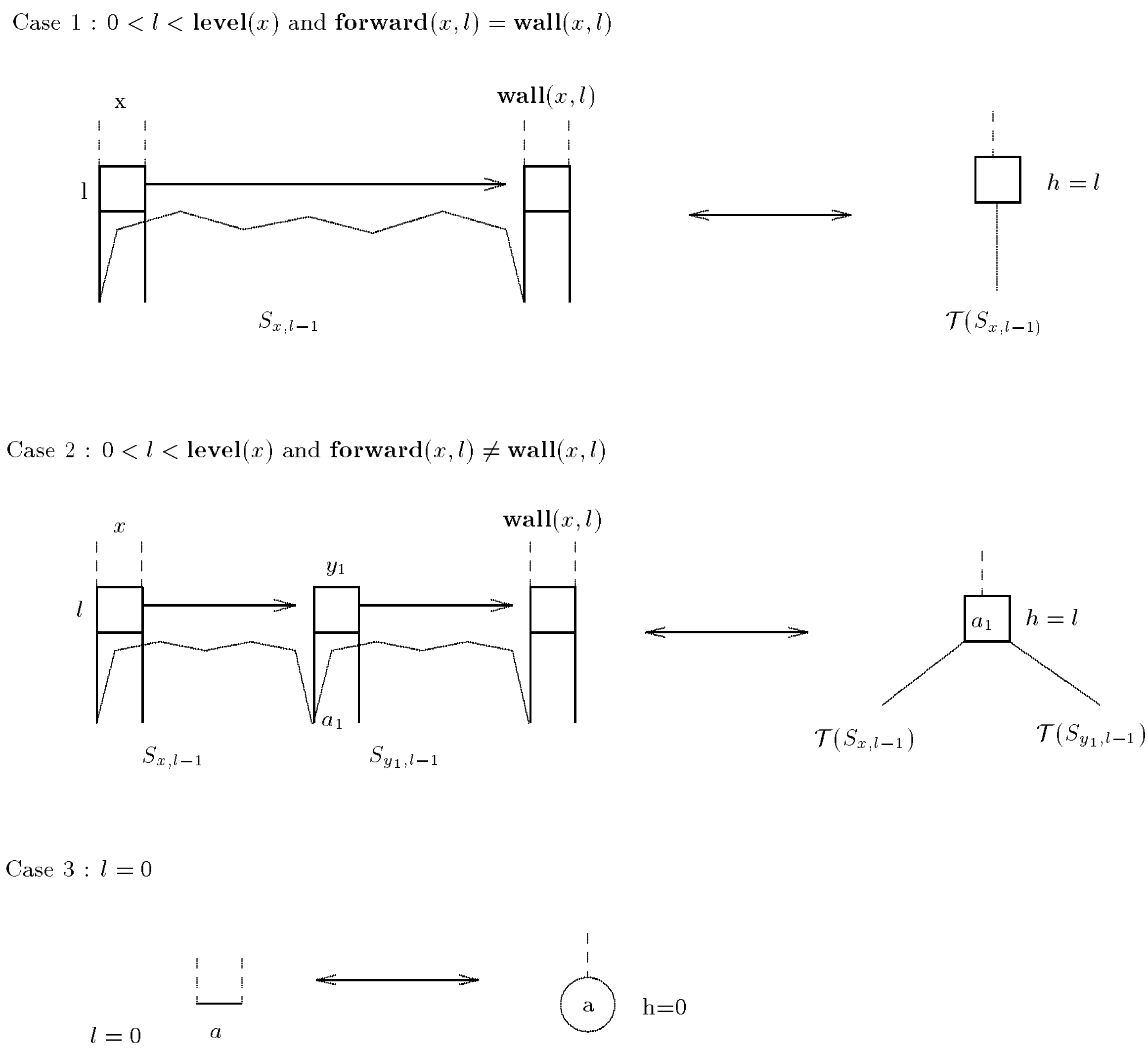


Figure 5: Mapping $\mathcal{T}$ between Skip lists and Skip trees

1. *When $f = 1$ and $l > 0$ we have*

$$\mathcal{T}(S_{x,l}) \quad = \quad \begin{array}{c} \square \\ | \\ \mathcal{T}(S_{x,l-1}) \end{array}$$

2. *When $f > 1$ and $l > 0$ we note $y_i =$*$\mathbf{forward}^i(x,l)$ *(in particular $x = y_0 =$*$\mathbf{forward}^0(x,l)$*), $a_i = \mathbf{key}(y_i)$ and $T_i = \mathcal{T}(S_{y_i,l-1})$ we have:*

$$\mathcal{T}(S_{x,l}) \quad = \quad \begin{array}{ccccccccccc} [ & & a_1 & | & a_2 & | & \ldots & | & a_{f-1} & & ] \\ / & & & | & & | & & | & & & \backslash \\ T_0 & & & T_1 & & T_2 & & T_{f-2} & & & T_{f-1} \end{array}$$

3. *When $l = 0$ we have*

$$\mathcal{T}(S(x,l)) \quad = \quad \begin{array}{c} | \\ \mathbf{key}(x) \end{array}$$

We can easily verify that Skip list given in the Figure 1 transforms to the Skip tree in Figure 2.

**Lemma 1** *The mapping $\mathcal{T}$ :* Skip lists $\longrightarrow$ Skip trees *is a one-to-one function.*

### 3.1.2 Commutation between local rules and mapping

More interestingly, rules on Skip trees match with rules on Skip lists. To be more precisely:

**Lemma 2** *Given a* Skip list $S$ *having all the keys different, the following relationship between rules of* Skip list $S$ *and* Skip tree $\mathcal{T}(S)$ *holds:*

1. *Given a key $a$ belonging to $S$ it holds that $\mathcal{T}(\mathbf{upward}(S,a)) = \mathbf{split}(\mathcal{T}(S),a)$.*

2. *Given a key $a$ which does not appears in $S$, it holds that $\mathcal{T}(\mathbf{attach}(S,a)) = \mathbf{attach}(\mathcal{T}(S),a)$.*

**Proof.** First, give us a proof outline of (1). Assume that $a = \mathbf{key}(x)$ and $l = \mathbf{level}(x)$. As in $\mathbf{upward}(S,a)$ the level of $x$ will be $l+1$, we consider in $S$ the smallest subskip $S'$ containing $x$ with $\mathbf{level}(S') > l$. When $x$ goes up one level it will cut the forward pointer at level $l+1$ of a node $y$ belonging to $S'$. We would need to consider three main cases depending on $y$ and $z =$$\mathbf{forward}(y, l+1)$.

- Both $y$ and $z$ verify $\mathbf{level}(y) = \mathbf{level}(z) = l+1$.
- Both $y$ and $z$ verify $\mathbf{level}(y) > l+1$ and $\mathbf{level}(z) > l+1$.
- The node $y$ verifies $\mathbf{level}(y) > l+1$ but node $z$ verifies $\mathbf{level}(z) = l+1$.
- Reciprocally we have, $\mathbf{level}(y) = l+1$ but node $z$ verifies $\mathbf{level}(z) > l+1$.

For each one of these cases there are another four possibilities, depending on the brothers surrounding $x$.

- The node $x$ is surrounded by left and right brothers having exactly level $l$.
- The node $x$ has only a left brother with level exactly $l$.
- The node $x$ has only a right brother having level exactly $l$.
- There are no brothers surrounding $x$ and having level $l$.

The preceding considerations allow us a total of 16 different contexts for $x$ inside $S'$ straightforward to prove ■

We give due attention to inverse rules. Therefore, given a Skip list $S$ with different keys, and a key $a$ with level greater than one, it holds that:

$$\mathcal{T}(\mathbf{downward}(S, a)) = \mathbf{join}(\mathcal{T}(S), a).$$

If the case $a$ has level one, we have $\mathcal{T}(\mathbf{unattach}(S, a)) = \mathbf{unattach}(\mathcal{T}(S), a)$.

# 4 Concurrent and sequential algorithms on Skip trees

The trees are generated by the forthcoming insertion and deletion algorithms, but starting on empty trees. As the insertion is random, these trees are (random) Skip trees, but as people do with Skip lists we only refer to them as Skip trees.

Before designing the algorithms we select the random distribution which determines the height of keys. As the height is equal to the level on Skip list, it is determined by the random variable $1 + NB(1, q)$. Thus by a "Skip tree with parameter $q$" we mean Skip trees whose keys have all been inserted following this random distribution.

We first address the main algorithmic contribution of this paper: the concurrent algorithm *on the fly* approach and its proof of correctness. Later on we derive the sequential algorithms because they can be viewed as concurrent ones acting over one key.

## 4.1 Concurrent algorithm

We design a concurrent algorithm *on the fly* approach. This approach, inspired by [5], suggests defining a set of local rules which modifies the tree with a nondeterministic evolution strategy, until the desired final state is reached. Although locking groups of nodes during critical updates cannot be avoided, *locality* of rules ensures small number of locked keys and for a short time as possible. Observe that the rules explained in section 3 verify these properties.

The correctness of the algorithm derives from the following properties:

**safety** : expresses that if no rule can apply, then the final state has been reached (partial correctness),

**liveness** : expresses that eventually no rules applies (total correctness),

The safety property is easy to test from the guard (precondition) of rules and the proof of liveness needs a variant function that should be positive and strictly decreases at each application of any rule.

### 4.1.1 Search rules

Assume that we search the Skip tree $T$ for the set of items $a_1, a_2, \ldots$. We add to each node a new register, denoted **waiting** bag, that contains all the keys waiting to be percolated. We do not assume any strategy, such as FIFO, LIFO, ..., in the management of the bag. The algorithm introduce all items into the bag of the root, and applies the rule which percolates them down through the tree.

**Rule : Percolation**

> **Guard:** *Node $x$ such that* **waiting**$(x) \neq \emptyset$.
>
> **Behavior:** *An item $a$ is extracted from* **waiting**$(x)$ *bag. If $a \in$* **key**$(x)$ *the key has been found. Otherwise we consider four cases: $i$) if $x$ is a leaf the item does not belongs to the tree and is erased. $ii$) if $x$ is white then $a$ is added to bag* **child**$(x)[0]$. *$iii$) if there is some $i$ such that* **key**$(x)[i] < a <$ **key**$(x)[i+1]$, *then $a$ is added to bag* **waiting**$($**child**$(x)[i])$. *$iv$) if $a$ is smaller than the smaller key (larger than the larger key) then $a$ is added to the leftmost (rightmost) child bag.*
>
> **Spatial scope:** *Node $x$ and one son.*

We prove the correctness:

**Safety:** if no rule applies all the waiting bags are empty, then all keys have been searched.

**Liveness:** let $IN(x)$ the number of nodes contained inside the tree rooted at $x$, and $IN_W = \sum IN(x)|\textbf{waiting}(x)|$ this addition for all nodes. The variant function $IN_W$ strictly decreases.

### 4.1.2 Search and insertion rules

Now we deal with items to be searched only and items to be searched and further inserted. Then we slightly modify the behaviour of **Percolation** in order to take into account the last class of items, and we recall from section 3 the rule **split**.

**Rule : Percolation**

> **Behavior:** *Assume that item $a$, extracted from* **waiting**$(x)$, *should be inserted. If it is found, i.e. is equal to some key of $x$, then remove it from waiting bags. Otherwise when it reaches a leaf, a new key $a$ is added to skiptree (with* **attach** *rule), and its height, namely* rc-height, *is randomly computed and stored with the new key.*

**Rule : Split**

> **Guard:** *Key $a$ of node $x$ has rc-height larger than those of $x$.*
>
> **Behavior:** *Explained in section 3. Observe that the waiting bag is split too.*
>
> **Spatial scope:** *Node $x$ and its parent key.*

We prove the correctness:

**Safety:** if no rules applies all items have been percolated, and the unsuccessful ones have been attached and sent up at their rc-height.

**Liveness:** let $W$ the variant function that stores the number of items contained in waiting bags and $H$ the addition of the differences between the rc-height and the current height of new keys. The variant function $(W, H, IN_W)$ strictly decreases at each step of both rules.

### 4.1.3 Search, insertion and deletion rules

As in the preceding section, we slightly modify the rule **Percolation** in order to take into account the new kind of items to be deleted, and we recall the **split** and **join** rules.

**Rule : Percolation**

> **Behavior:** *Assume that item $a$, extracted from* **waiting**$(x)$*, should be deleted. If the key is found then it is colored gray, otherwise the item is sent down (if $x$ is a leave "sent down" means "erased").*

**Rule : Split**

> **Guard:** *Now key $a$ of node $x$ cannot be gray.*
>
> **Behavior:** *The same.*
>
> **Spatial scope:** *Node $x$ and its parent key.*

**Rule : Join**

> **Guard:** *Key $a$ is gray.*
>
> **Behavior:** *Explained in section 3. Observe that if the gray key is located at a leaf we apply* **unattach** *rule to remove it.*
>
> **Spatial scope:** *Key $a$ and both childs.*

We prove the correctness:

**Safety:** if no rules applies all items have been percolated and the grey keys have been removed from the tree.

**Liveness:** let $IN_G = \sum IN(x)$ this addition for all nodes with gray keys. The variant function $(W, IN_G)$ strictly decreases at each application of join rule; therefore, $(W, H, IN_W, IN_G)$ strictly decreases at each step of any rule.

### 4.1.4 Expected magnitudes of Skip trees

The distribution function of the following magnitudes can be deduced:

**Height:** it is determined by the random variable $H = 1 + NB(1, q)$ (like the height of Skip lists). The expected value is $1/q$.

**Length:** Let $L$ be the random variable whose value is the number of keys of nodes. Thnks to the one-to-one mapping, $L$ is equal to the number of consecutive keys with the same level, magnitude which is denoted *gap* by Papadakis [17] and that is determined by the random distribution $L = NB(1, p)$ whose expected value is $q/p$.

The Skip trees inherit from Skip lists the following expected properties:

**Theorem 1** *Let $T$ be a* Skip tree *with parameter $q$ and size $n$ ($q + p = 1$):*

1. *The expected height of $T$ is $O(\log_{1/p} n)$, and the probability that the expected height deviates $k$ times from the expected value decreases as $O(n^{-k})$.*
2. *The expected length of internal nodes is $q/p$, and the probability that it was deviated $c$ times from the expected value is $q^{\lceil cq/p \rceil}$.*
3. *The expected number of splits or joins while updating a key is $1/q$.*

**Proof.** 1. Given a random Skip tree $T$ with parameter $q$ and $n$ keys, the Skip list $S = \mathcal{T}^{-1}(T)$ is an usual Skip list with $n$ keys where the height of nodes is determined by the random variable $1 + NB(1, q)$, then the height of $S$ holds $O(\log_{1/p} n)$. To prove that the expected height decreases exponentially we apply the Chernoff tail bound lemma [2],

$$\text{Prob}\{H \geq a\} \leq E(e^{tH})/e^{ta} \qquad \forall a, t > 0.$$

The expectations for $H$ are $E(e^{tH}) = q/(e^{-t} - p)$ being $q = 1 - p$. Then

$$\text{Prob}\{H \geq c \log_{1/p} n\} \leq \frac{q}{e^{-t} - p} / e^{tc \log_{1/p} n} = \frac{q}{p^{k/c} - p} n^{-k}$$

taking $t = \frac{k}{c} \ln(1/p)$. Observe that this result is well defined for $1 \leq c < k$.

2. Let $L$ be the random variable whose value is the number of keys in a node of a tree $T$, then

$$\text{Prob}\{L > cE(L)\} = \text{Prob}\{L > cq/p\} = \sum_{k \geq cq/p} \text{Prob}\{L = k\} = \sum_{k \geq cq/p} pq^k = q^{\lceil c\frac{q}{p} \rceil}$$

3. The expected number of splits and joins is equal to the expected height of one key, then

$$E(1 + NB(1, q)) = 1 + E(NB(1, q)) = 1 + p/q = 1/q$$

■

## 4.2 Sequential algorithms

These algorithms can be easily derived from the concurrent ones because they can be viewed as concurrent ones acting over one key. Thus they are easily designed, their correctness is ensured by the concurrent one and the time bounds are derived from Skip lists. Clearly we do not need the waiting bags.

- **Search algorithm:** the item is percolated until it is found or it falls from a leaf.
- **Insertion algorithm:** the item is percolated. If the search was unsuccessful a new key is attached and the rc-height is determined. Then it is split until the key reaches its rc-height.

- **Deletion algorithm:** the item is percolated until it is found and coloured or it falls from a leaf. In the first case we join the gray key until it is removed.

The correctness is ensured by the correctness of the concurrent algorithm. Due to the one-to-one mapping, the time of updates has the same performance as the sequential updates time of Skip lists. Given a Skip tree $T$, the inverse $S = \mathcal{T}^{-1}(T)$ is a Skip list. Therefore we transferre the results obtained in Skip lists [22, 7]:

**Theorem 2** *Let $T$ be a* Skip tree *of size $n$ and random parameter $q$. The expected time for searching, inserting or deleting a key is $O(\log_{1/p} n)$ and the probability that the expected height deviates $k$ times from the expected value decreases as $O(n^{-k})$.*

# 5 Close to B-trees

Although a Skip list can not be efficiently broken (or paginated) to store some parts in secondary or external memory, Skip trees have this capability due to their structure. This fact and the closure between B-trees and Skip trees with parameter $q \simeq 1$ suggest the use of Skip trees for handling indices of data bases, such as B-trees do. The following lemma explores this possibility.

**Lemma 3** *Let $T$ be a* Skip tree *of size $n$ and parameter $q$, recall $L$ and $H$ as random variables giving the length of nodes and the height of tree:*

1. $\text{Prob}\{L > cE(L)\} \simeq (eq)^{-c}$ *for larger values of $q$.*
2. $\text{Prob}\{H > 1 + \log_{1/p}(n)\} = 1 - (1 - \frac{p}{n})^n \simeq p$ *for $n >> 1$ and $p \simeq 0$.*

**Proof.** 1. Recall that $\text{Prob}\{L > cE(G)\} \leq \left(q^{q/p}\right)^c$. This expression can be approximated by

$$q^{\frac{q}{p}} = (1-p)^{\frac{1-p}{p}} = \frac{(1-p)^{1/p}}{(1-p)} \simeq (qe)^{-1}$$

2. Recall that the height of each node is also a negative binomial but with parameter $q$: $H = NB(1, q) + 1$. Say $L(n) = \log_{1/p} n$, then

$$\text{Prob}\{H_n > L(n) + 1\} = 1 - \text{Prob}\{H_n \leq 1 + L(n)\} = 1 - \prod_{k=1}^{n} \text{Prob}\{H \leq 1 + L(n)\}$$

because all the keys should have smaller height,

$$= 1 - \prod_{k=1}^{n} \left(1 - p^{L(n)+1}\right) = 1 - \left(1 - p^{L(n)+1}\right)^n = 1 - \left(1 - \frac{p}{n}\right)^n \overset{n>>1}{\simeq} 1 - e^{-p} \overset{p\simeq 0}{\simeq} p$$

because they are independent random variables.

■

The above lemma suggests that Skip trees manage data base as well as large were $q$ and $n$, but taking care about the length of nodes. The probability that the height was greater than the expected one is very small, but this is not true for the length of nodes: for instance, $c = 1$ determines that about $n/e$ nodes grows more than the expected value. This fact suggests implementing Skip trees with nodes of variable length as E. McCreight [12] or G. Diehr and B. Faaland [4] propose for B-trees.

## 6 Conclusions and further research

From a theoretical point of view, the balanced search trees can be separated into two groups depending on the set of local rules: a first group, which uses splits and joins, consisting of B-trees and derivatives, and a second group, which uses rotations, composed of AVL trees and Red-black trees. Therefore, the main theoretical conclusion of this paper is that Skip lists, by means of Skip trees, belong to the first group. Hence, in the trees of this group holds that if the length of the nodes is bounded we are dealing with 2–3 trees, 2–3–4 trees, ..., but if the length is randomly determined by a negative binomial distribution we are dealing with Skip trees (or Skip lists).

A second conclusion is that Skip trees lie between Skip lists and the family of B-trees. Skip trees inherit random characteristics and race properties from Skip lists, and structural and algorithmic improvements from the family of B-trees. For instance, an example of a structural improvement is the defintion, by following C. Douglas [6], of Skip$^*$ trees or Skip$^+$ trees. And an example of algorithmic improvements is that we can translate algorithms from B-trees to Skip trees by taking parameter $q \simeq 1$, or from 2–3 trees to Skip trees by taking parameter $q \simeq 0.5$.

From a practical point of view, Skip trees can be partitioned to paginate them as if they were B-trees and can be applied to manage data bases, taking care, however,with the length of nodes.

Finally, we have designed a concurrent algorithm *on the fly* approach.

We leave for further research the design of massively parallel algorithms for Skip trees. The state of the art is that W. Paul, U. Vishkin and H. Wagener [19] designed a parallel algorithm for 2–3 trees, L. Higham and E. Schenks [9] designed one for B-trees, and J. Gabarró, C. Martínez and X. Messeguer [7] designed a parallel algorithm for Skip lists. Thus, the parallel algorithm of Skip trees will arise from the right mixture of them.

## Acknowledgments

We thank Joaquim Gabarró and Conrado Martínez for helping us with this research.

## References

[1] L. Bougé, J. Gabarró, and X. Messeguer. Concurrent AVL revisited: self-balancing distributed search trees. Technical Report LSI-95-54, LSI-UPC, 1995. Also appeared as Tech.Rep. RR95-45,LIP,ENS Lyon.

[2] H. Chernoff. A measure of asymptotic efficiency for tests of hypothesis based on the sum of observations. *Ann. Math. Statist.*, 23:493–507, 1952.

[3] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.

[4] G. Diehr and B. Faaland. Optimal pagination of B trees with variable-length items. *Communications of the ACM*, 27(3):241–247, 1984.

[5] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the fly garbage collection: an exercise in cooperation. *CACM*, 21:966–975, 1978.

[6] C. Douglas. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[7] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, (158):1–33, 1996.

[8] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In IEEE, editor, *Proc. of 19th Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[9] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *J. of Parallel and Dist. Comp.*, 22:329–335, 1994.

[10] J.L.W. Kessels. On-the-fly optimization of data structures. *CACM*, 26(11):895–901, 1983.

[11] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 1994. To appear.

[12] E. McCreight. Pagination of B* trees with variable length records. *Communications of the ACM*, 20:670–674, 1977.

[13] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996. Also appeared in 10th ACM PODS, 1991.

[14] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *6th ACM PODS*, pages 170–176, 1987.

[15] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrent balancing and updating of avl trees. In *6 ACM PODS*, 1987.

[16] T. Ottmann, H. Six, and D. Wood. On the correspondende between AVL trees and brother trees. *Computing*, 23(1):43–54, 1979.

[17] T. Papadakis. *Skip Lists and Probabilistic Analysis of Algorithms*. PhD thesis, University of Waterloo, 1993. Available as Technical Report CS-93-28.

[18] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[19] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, 1983. Also appeared as "Parallel computation on 2–3 trees" in RAIRO Informatique Théorique, pages 397-404, 1983.

[20] P.E. Pfeiffer. *Probability for applications*. Springer-Verlag, 1992.

[21] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, Apr 1989. Also published as UMIACS-TR-90-80.

[22] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[23] R. Seidel and R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. Appeared in Proc. of 30th Symposium on Foundations of Computer Science, 1989.

[24] S. Sen. Some observations on skip-lists. *Information Processing Letters*, 39(3):173–176, 1991.