View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Munin - Open Research Archive

DeltaTree: A Practical Locality-aware Concurrent Search Tree (*IFI-UIT Technical Report* **2013-74**)

Ibrahim Umar Otto J. Anshus Phuong H. Ha

{ibrahim.umar, otto.anshus, phuong.hoai.ha}@uit.no

High Performance and Distributed Systems Group

Department of Computer Science

UiT The Arctic University of Norway

(formerly University of Tromsø)

October 10, 2013

Abstract

As other fundamental programming abstractions in energy-efficient computing, search trees are expected to support both high parallelism and data locality. However, existing highly-concurrent search trees such as red-black trees and AVL trees do not consider data locality while existing locality-aware search trees such as those based on the van Emde Boas layout (vEB-based trees), poorly support concurrent (update) operations.

This paper presents DeltaTree, a practical locality-aware concurrent search tree that combines both locality-optimisation techniques from vEB-based trees and concurrency-optimisation techniques from non-blocking highly-concurrent search trees. DeltaTree is a k-ary leaf-oriented tree of DeltaNodes in which each DeltaNode is a size-fixed tree-container with the van Emde Boas layout. The expected memory transfer costs of DeltaTree's *Search, Insert* and *Delete* operations are $O(log_B N)$, where N, B are the tree size and the *unknown* memory block size in the ideal cache model, respectively. DeltaTree's *Search* operation is wait-free, providing prioritised lanes for *Search* operations, the dominant operation in search trees. Its *Insert* and *Delete* operations are non-blocking to other *Search, Insert* and *Delete* operations, but they may be occasionally blocked by maintenance operations that are sometimes triggered to keep DeltaTree in good shape. Our experimental evaluation using the latest implementation of AVL, red-black, and speculation friendly trees from the Synchrobench benchmark has shown that DeltaTree is up to 5 times faster than all of the three concurrent search trees for searching operations and up to 1.6 times faster for update operations when the update contention is not too high.

Contents

1 Introduction 5

2 Dynamic Van Emde Boas Layout 7
- 2.1 Notations . . . 7
- 2.2 Static van Emde Boas (vEB) Layout . . . 8
- 2.3 Relaxed Cache-oblivious Model and Dynamic vEB Layout . . . 8

3 ΔTree Overview 11

4 Detailed Implementation 13
- 4.1 Function specifications . . . 13
- 4.2 Synchronisation calls . . . 14
- 4.3 Wait-free and Linearisability of search . . . 14
- 4.4 Non-blocking Update Operations . . . 19
- 4.5 Memory Transfer and Time Complexities . . . 21

5 Experimental Result and Discussion 22

6 Related Work 27

7 Conclusions and Future Work 27

List of Figures

1 An illustration for the van Emde Boas layout 6
2 An illustration for the new dynamic vEB layout 9
3 Search illustration 10
4 Depiction of ΔTree universe U 11
5 (a)*Rebalancing*, (b)*Expanding*, and (c)*Merging* operations on ΔTree 13
6 Wait and check algorithm 14
7 Cache friendly binary search tree structure 15
8 A wait-free searching algorithm of ΔTree 16
9 Update algorithms and their helpers functions 19
10 Merge and Balance algorithm 20
11 Performance rate (operations/second) of a ΔTree with 1,023 initial members. The y-axis indicates the rate of operations/second. 24
12 Performance rate (operations/second) of a ΔTree with 2,500,000 initial members. The y-axis indicates the rate of operations/second. 25

List of Tables

1 Cache profile comparison during 100% searching 26

1 Introduction

Energy efficiency is becoming a major design constraint in current and future computing systems ranging from embedded to high performance computing (HPC) systems. In order to construct energy efficient software systems, data structures and algorithms must support not only high parallelism but also data locality [Dal11]. Unlike conventional locality-aware data structures and algorithms that concern only whether data is on-chip (e.g. data in cache) or not (e.g. data in DRAM), new energy-efficient data structures and algorithms must consider data locality in finer-granularity: where on chip the data is. It is because in modern manycore systems the energy difference between accessing data in nearby memories (2pJ) and accessing data across the chip (150pJ) is almost two orders of magnitude, while the energy difference between accessing on-chip data (150pJ) and accessing off-chip data (300pJ) is only two-fold [Dal11]. Therefore, fundamental data structures and algorithms such as search trees need to support both high parallelism and fine-grained data locality.

However, existing highly-concurrent search trees do not consider fine-grained data locality. The highly concurrent search trees includes non-blocking [EFRvB10, BH11] and Software Transactional Memory (STM) based search trees [AKK+12, BCCO10, CGR12, DSS06]. The prominent highly-concurrent search trees included in several benchmark distributions are the concurrent red-black tree [DSS06] developed by Oracle Labs and the concurrent AVL tree developed by Stanford [BCCO10]. The highly concurrent trees, however, do not consider the tree layout in memory for data locality.

Concurrent B-trees [BP12, Com79, Gra10, Gra11] are optimised for a known memory block size B (e.g. page size) to minimise the number of memory blocks accessed during a search, thereby improving data locality. As there are different block sizes at different levels of the memory hierarchy (e.g. register size, SIMD width, cache line size and page size) that can be utilised to design locality-aware layout for search trees [KCS+10], concurrent B-trees limits its spatial locality optimisation to the memory level with block size B, leaving memory accesses to the other memory levels unoptimised. For example, if the concurrent B-trees are optimised for accessing disks (i.e. B is the page size), the cost of searching a key in a block of size B in memory is $\Theta(log(B/L))$ cache line transfers, where L is the cache line size [BFJ02]. Since each memory read basically contains only one node of size L from a top down traversal of a path in the search tree of B/L nodes, except for the topmost $\lfloor log(L+1) \rfloor$ levels. Note that the optimal cache line transfers in this case is $O(log_L B)$, which is achievable by using the van Emde Boas layout.

A van Emde Boas (vEB) tree is an ordered dictionary data type which implements the idea of recursive structure of priority queues [vEB75]. The efficient structure of the vEB tree, especially how it arranges data in a recursive manner so that related values are placed in contiguous memory locations, has inspired cache oblivious (CO) data structures [Pro99] such as CO B-trees [BDFC05, BFGK05, BFCF+07] and CO binary trees [BFJ02]. These researches have demonstrated that the locality-aware structure of the vEB layout is a perfect fit for cache oblivious algorithms, lowering the upper bound on memory transfer complexity.

Figure 1 illustrates the vEB layout. A tree of height h is conceptually split between nodes of heights $h/2$ and $h/2+1$, resulting in a top subtree T of height $h/2$ and $m = 2^{h/2}$ bottom subtrees $B_1, B_2, \cdots, B_m$ of height $h/2$. The $(m+1)$ subtrees are located in

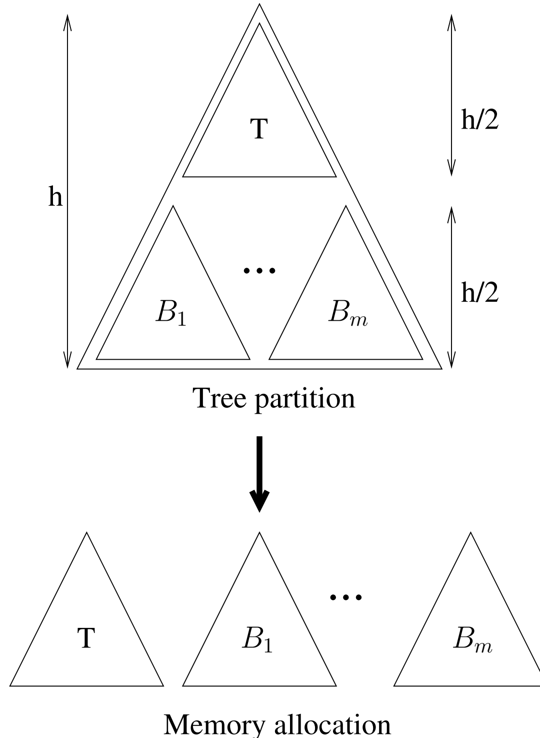

Figure 1: An illustration for the van Emde Boas layout

contiguous memory locations in the order $T, B_1, B_2, \cdots, B_m$. Each of the subtrees of height $h/2^i, i \in \mathbb{N}$, is recursively partitioned into $(m+1)$ subtrees of height $h/2^{i+1}$ in a similar manner, where $m = 2^{h/2^{i+1}}$, until each subtree contains only one node. With the vEB layout, the search cost is $O(log_B N)$ memory transfers, where N is the tree size and B is the *unknown* memory block size in the I/O [AV88] or ideal-cache[FLPR99] model. The search cost is optimal and matches the search bound of B-trees that requires the memory block size B to be known in advance. More details on the vEB layout are presented in Section 2.

The vEB-based trees, however, poorly support concurrent update operations. Inserting or deleting a node in a tree may result in relocating a large part of the tree in order to maintain the vEB layout. For example, inserting a node in full subtree T in Figure 1 will affect the other subtrees $B_1, B_2, \cdots, B_m$ due to shifting them to the right in the memory, or even allocating a new contiguous block of memory for the whole tree, in order to have space for the new node [BFJ02]. Note that the subtrees $T, B_1, B_2, \cdots, B_m$ must be located in *contiguous* memory locations according to the vEB layout. The work in [BFGK05] has discussed the problem but not yet come out with a feasible implementation [BP12].

We introduce ΔTree, a novel locality-aware concurrent search tree that combines both locality-optimisation techniques from vEB-based trees and concurrency-optimisation techniques from non-blocking highly-concurrent search trees. Our contributions are threefold:

- We introduce a new *relaxed* cache oblivious model and a novel *dynamic* vEB layout that makes the vEB layout suitable for highly-concurrent data structures with update operations. The dynamic vEB layout supports dynamic node allocation via pointers while maintaining the optimal search cost of $O(log_B N)$ memory transfers

without knowing the exact value of B (cf. Lemma 2.1). The new relaxed cache-oblivious model and dynamic vEB layout are presented in Section 2.

- Based on the new dynamic vEB layout, we develop ΔTree, a novel locality-aware concurrent search tree. ΔTree is a k-ary leaf-oriented tree of ΔNodes in which each ΔNode is a size-fixed tree-container with the van Emde Boas layout. The expected memory transfer costs of ΔTree's *Search, Insert* and *Delete* operations are $O(log_B N)$, where N is the tree size and B is the *unknown* memory block size in the ideal cache model [FLPR99]. ΔTree's *Search* operation is wait-free while its *Insert* and *Delete* operations are non-blocking to other *Search, Insert* and *Delete* operations, but they may be occasionally blocked by maintenance operations. ΔTree overview is presented in Section 3 and its detailed implementation and analysis are presented in Section 4.

- We experimentally evaluate ΔTree on commodity machines, comparing it with the prominent concurrent search trees such as AVL trees [BCCO10], red-black trees [DSS06] and speculation friendly trees [CGR12] from the Synchrobench benchmark [Gra]. The experimental results show that ΔTree is up to 5 times faster than all of the three concurrent search trees for searching operations and up to 1.6 times faster for update operations when the update contention is not too high. We have also developed a concurrent version of the sequential vEB-based tree in [BFJ02] using GCC's STM in order to gain insights into the performance characteristics of concurrent vEB-based trees. The detailed experimental evaluation is presented in Section 5. The code of the ΔTree and its experimental evaluation are available upon request.

2 Dynamic Van Emde Boas Layout

2.1 Notations

We first define these notations that will be used hereafter in this paper:

- b_i (unknown): block size in term of nodes at level i of memory hierarchy (like B in the I/O model [AV88]), which is unknown as in the cache-oblivious model [FLPR99]. When the specific level i of memory hierarchy is irrelevant, we use notation B instead of b_i in order to be consistent with the I/O model.

- UB (known): the upper bound (in terms of the number of nodes) on the block size b_i of all levels i of the memory hierarchy.

- $\Delta Node$: the coarsest recursive subtree of a vEB-based search tree that contains at most UB nodes (cf. dash triangles of height 2^L in Figure 3). ΔNode is a size-fixed tree-container with the vEB layout.

- Let L be the level of detail of ΔNodes. Let H be the height of a ΔNode, we have $H = 2^L$. For simplicity, we assume $H = log_2(UB + 1)$.

- N, T: size and height of the whole tree in terms of basic nodes (not in terms of ΔNodes).

- *density*$(r) = n_r/UB$ is the density of ΔNode rooted at r, where n_r the number of nodes currently stored in the ΔNode.

2.2 Static van Emde Boas (vEB) Layout

The conventional *static* van Emde Boas (vEB) layout has been introduced in cache-oblivious data structures [BDFC05, BFJ02, FLPR99]. Figure 1 illustrates the vEB layout. Suppose we have a complete binary tree with height h. For simplicity, we assume h is a power of 2, i.e. $h = 2^k$. The tree is recursively laid out in the memory as follows. The tree is conceptually split between nodes of height $h/2$ and $h/2 + 1$, resulting in a top subtree T and $m_1 = 2^{h/2}$ bottom subtrees $B_1, B_2, \cdots, B_m$ of height $h/2$. The $(m_1 + 1)$ top and bottom subtrees are then located in consecutive memory locations in the order of subtrees $T, B_1, B_2, \cdots, B_m$. Each of the subtrees of height $h/2$ is then laid out similarly to $(m_2 + 1)$ subtrees of height $h/4$, where $m_2 = 2^{h/4}$. The process continues until each subtree contains only one node, i.e. the finest *level of detail*, 0. Level of detail d is a partition of the tree into recursive subtrees of height at most 2^d.

The main feature of the vEB layout is that the cost of any search in this layout is $O(log_B N)$ memory transfers, where N is the tree size and B is the *unknown* memory block size in the I/O [AV88] or ideal-cache [FLPR99] model. The search cost is the optimal and matches the search bound of B-trees that requires the memory block size B to be known in advance. Moreover, at any level of detail, each subtree in the vEB layout is stored in a contiguous block of memory.

Although the vEB layout is helpful for utilising data locality, it poorly supports concurrent update operations. Inserting (or deleting) a node at position i in the contiguous block storing the tree may restructure a large part of the tree stored after node i in the memory block. For example, inserting new nodes in the full subtree A in Figure 1 will affect the other subtrees $B_1, B_2, \cdots, B_m$ by shifting them to the right in order to have space for new nodes. Even worse, we will need to allocate a new contiguous block of memory for the whole tree if the previously allocated block of memory for the tree runs out of space [BFJ02]. Note that we cannot use dynamic node allocation via pointers since at *any* level of detail, each subtree in the vEB layout must be stored in a *contiguous* block of memory.

2.3 Relaxed Cache-oblivious Model and Dynamic vEB Layout

In order to make the vEB layout suitable for highly concurrent data structures with update operations, we introduce a novel *dynamic* vEB layout. Our key idea is that if we know an upper bound UB on the unknown memory block size B, we can support dynamic node allocation via pointers while maintaining the optimal search cost of $O(log_B N)$ memory transfers without knowing B (cf. Lemma 2.1).

We define *relaxed cache oblivious* algorithms to be cache-oblivious (CO) algorithms with the restriction that an upper bound UB on the unknown memory block size B is known in advance. As long as an upper bound on all the block sizes of multilevel memory is known, the new relaxed CO model maintains the key feature of the original CO model, namely analysis for a simple two-level memory are applicable for an unknown multilevel memory (e.g. registers, L1/L2/L3 caches and memory). This feature enables

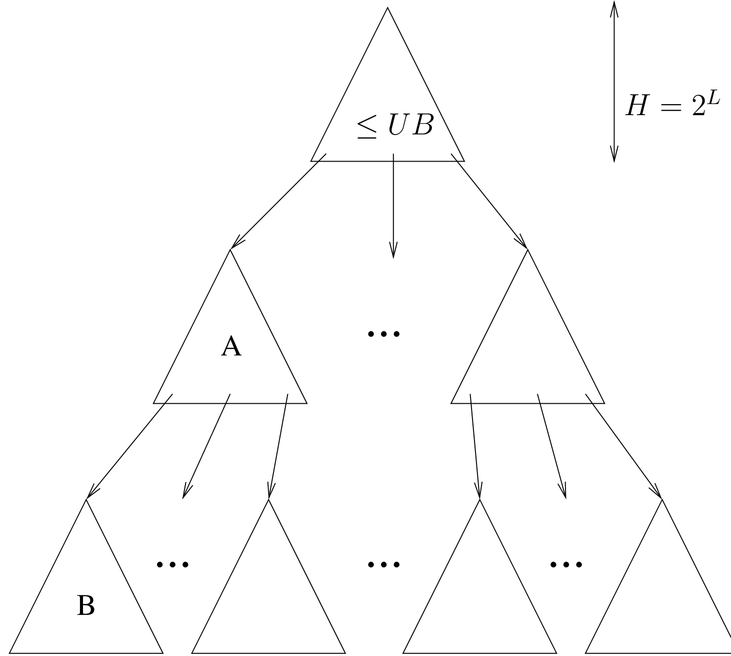

Figure 2: An illustration for the new dynamic vEB layout

designing algorithms that can utilise fine-grained data locality in energy-efficient chips [Dal11]. In practice, although the exact block size at each level of the memory hierarchy is architecture-dependent (e.g. register size, cache line size), obtaining a single upper bound for all the block sizes (e.g. register size, cache line size and page size) is easy. For example, the page size obtained from the operating system is such an upper bound.

Figure 2 illustrates the new dynamic vEB layout based on the relaxed cache oblivious model. Let L be the coarsest level of detail such that every recursive subtree contains at most UB nodes. The tree is recursively partitioned into level of detail L where each subtree represented by a triangle in Figure 2, is stored in a contiguous memory block of size UB. Unlike the conventional vEB, the subtrees at level of detail L are linked to each other using pointer, namely each subtree at level of detail $k > L$ is not stored in a contiguous block of memory. Intuitively, since UB is an upper bound on the unknown memory block size B, storing a subtree at level of detail $k > L$ in a contiguous memory block of size greater than UB, does not reduce the number of memory transfer. For example, in Figure 2, a travel from a subtree A at level of detail L, which is stored in a contiguous memory block of size UB, to its child subtree B at the same level of detail will result in at least two memory transfers: one for A and one for B. Therefore, it is unnecessary to store both A and B in a contiguous memory block of size $2UB$. As a result, the cost of any search in the new dynamic vEB layout is intuitively the same as that of the conventional vEB layout (cf. Lemma 2.1) while the former supports highly concurrent update operations because it utilises pointers.

Let ΔNode be a subtree at level of detail L, which is stored in a contiguous memory block of size UB and is represented by a triangle in Figure 2.

Lemma 2.1 *A search in a complete binary tree with the new dynamic vEB layout needs $O(log_B N)$ memory transfers, where N and B is the tree size and the* unknown *memory block size in the ideal cache model [FLPR99], respectively.*

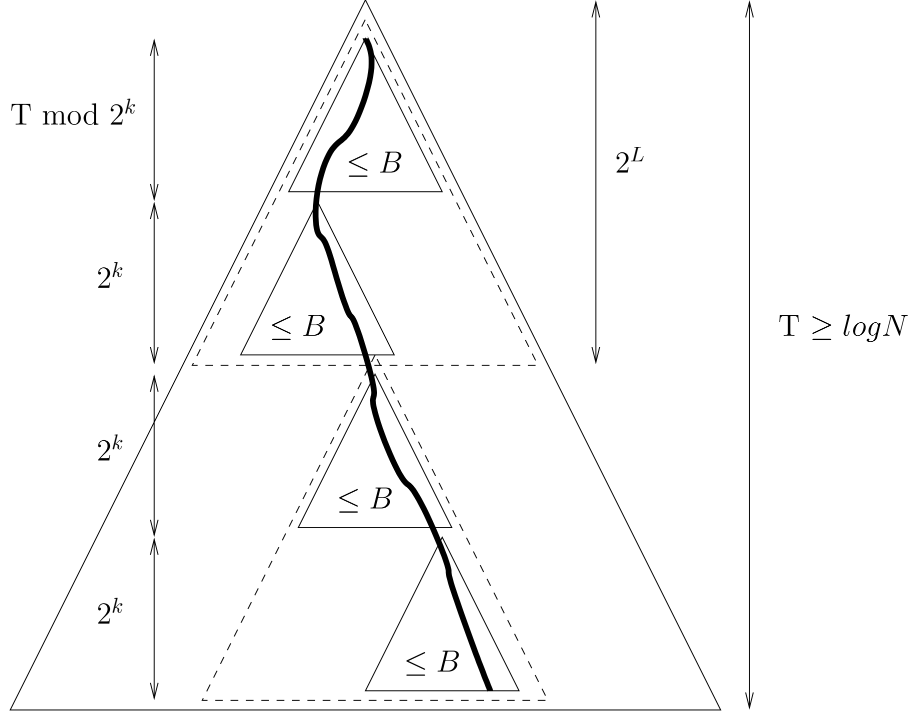

Figure 3: Search illustration

Proof. (Sketch) Figure 3 illustrates the proof. Let k be the coarsest level of detail such that every recursive subtree contains at most B nodes. Since $B \leq UB$, $k \leq L$, where L is the coarsest level of detail at which every recursive subtree contains at most UB nodes. That means there are at most 2^{L-k} subtrees along to the search path in a ΔNode and no subtree of depth 2^k is split due to the boundary of ΔNodes. Namely, triangles of height 2^k fit within a dash triangle of height 2^L in Figure 3.

Due to the property of the new dynamic vEB layout that at any level of detail $i \leq L$, a recursive subtree of depth 2^i is stored in a contiguous block of memory, each subtree of depth 2^k *within* a ΔNode is stored in at most 2 memory blocks of size B (depending on the starting location of the subtree in memory). Since every subtree of depth 2^k fits in a ΔNode (i.e. no subtree is stored across two ΔNodes), every subtree of depth 2^k is stored in at most 2 memory blocks of size B.

Since the tree has height T, $\lceil T/2^k \rceil$ subtrees of depth 2^k are traversed in a search and thereby at most $2\lceil T/2^k \rceil$ memory blocks are transferred.

Since a subtree of height 2^{k+1} contains more than B nodes, $2^{k+1} \geq log_2(B+1)$, or $2^k \geq \frac{1}{2} log_2(B+1)$.

We have $2^{T-1} \leq N \leq 2^T$ since the tree is a *complete* binary tree. This implies $log_2 N \leq T \leq log_2 N + 1$.

Therefore, $2\lceil T/2^k \rceil \leq 4\lceil \frac{log_2 N+1}{log_2(B+1)} \rceil = 4\lceil log_{B+1} N + log_{B+1} 2 \rceil = O(log_B N)$, where $N \geq 2$. $\square$

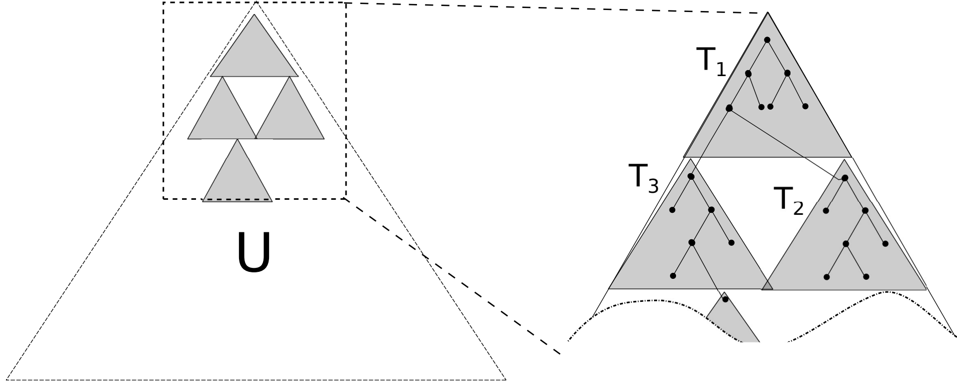

Figure 4: Depiction of ΔTree universe U

3 ΔTree Overview

Figure 4 illustrates a ΔTree named U. ΔTree U uses our new dynamic vEB layout presented in Section 2. The ΔTree consists of $|U|$ ΔNodes of fixed size UB each of which contains a *leaf-oriented* binary search tree (BST) $T_i, i = 1, \ldots, |U|$. ΔNode's internal nodes are put together in cache-oblivious fashion using the vEB layout.

The ΔTree U acts as the dictionary of abstract data types. It maintains a set of values which are members of an ordered universe [EFRvB10]. It offers the following operations: INSERTNODE(v, U), which adds value v to the set U, DELETENODE(v, U) for removing a value v from the set, and SEARCHNODE(v, U), which determines whether value v exists in the set. We may use the term *update* operation for either insert or delete operation. We assume that duplicate values are not allowed inside the set and a special value, say 0, is reserved as an indicator of an EMPTY value.

Operation SEARCHNODE(v, U) is going to walk over the ΔTree to find whether the value v exists in U. This particular operation is guaranteed to be wait-free, and returning **true** whenever v has been found, or **false** otherwise. The INSERTNODE(v, U) inserts a value v at the leaf of ΔTree, provided v does not yet exist in the tree. Following the nature of a leaf-oriented tree, a successful insert operation will replace a leaf with a subtree of three nodes [EFRvB10] (cf. Figure 5a). The DELETENODE(v, U) simply just *marks* the leaf that contains the value v as deleted, instead of physically removing the leaf or changing its parent pointer as proposed in [EFRvB10].

Apart from the basic operations, three maintenance ΔTree operations are invoked in certain cases of inserting and deleting a node from the tree. Operation REBALANCE$(T_v.root)$ is responsible for rebalancing a ΔNode after an insertion. Figure 5a illustrates the rebalance operation. Whenever a new node v is to be inserted at the last level H of ΔNode T_1, the ΔNode is rebalanced to a complete BST by setting a new depth for all leaves (e.g. a, v, x, z in Figure 5a) to $\log N + 1$, where N is the number of leaves. In Figure 5a, we can see that after the rebalance operation, tree T_1 becomes more balanced and its height is reduced from 4 into 3.

We also define the EXPAND(v) operation, that will be responsible for creating new ΔNode and connecting it to the parent of the leaf node v. Expand will be triggered only if after INSERTNODE(v, U), the leaf v will be at the last level of a ΔNode and rebalancing will no longer reduce the current height of the subtree T_i stored in the ΔNode. For example

if the expanding is taking place at a node v located at the bottom level of the ΔNode (Figure 5b), or $depth(v) = H$, then a new ΔNode (T_2 for example) will be referred by the parent of node v, immediately after value of node v is copied to $T_2.root$ node. Namely, the parent of v swaps one of its pointer that previously points to v, into the root of the newly created ΔNode, $T_2.root$.

The MERGE($T_v.root$) is for merging T_v with its sibling after a node deletion. For example, in Figure 5c T_2 is merged into T_3. Then the pointer of T_3's grandparent that previously points to the parent of both T_3 and T_2 is replaced to point to T_3. The operations are invoked provided that a particular ΔNode where the deletion takes place, is filled less than half of its capacity and all values of that ΔNode and its siblings can be fitted into a ΔNode.

To minimise block transfers required during tree traversal, the height of the tree should be kept minimal. These auxiliary operations are the unique feature of ΔTree in the effort of maintaining a small height.

These INSERTNODE and DELETENODE operations are non-blocking to other SEARCHNODE, INSERTNODE and DELETENODE operations. Both of the operations are using single word CAS (Compare and Swap) and "leaf-checking" to achieve that. Section 4 will explain more about these update operations.

As a countermeasure against unnecessary waiting for concurrent maintenance operations, a buffer array is provided in each of the ΔNodes. This buffer has a length that is equal to the number of maximum concurrent threads. As an illustration of how it works, consider two concurrent operations INSERTNODE(v, U) are operating inside the same ΔNode. Both are successful and have determined that expanding or rebalancing are necessary. Instead of rebalancing twice, those two threads will then compete to obtain the lock on that ΔNode. The losing thread will just append v into the buffer and then exits. The winning thread, which has successfully acquired the lock, will do rebalancing or expanding using all the leaves and the buffer of that ΔNode. The same process happens for concurrent delete, or the mix of concurrent insert and delete.

Despite INSERTNODE and DELETENODE are non-blocking, they still need to wait at the tip of a ΔNode whenever either of the maintenance operations, REBALANCE and MERGE is currently operating within that ΔNode. We employ TAS (Test and Set) using ΔNode lock to make sure that no basic update operations will interfere with the maintenance operations. Note that the previous description has shown that REBALANCE and MERGE execution are actually sequential within a ΔNode, so reducing the invocations of those operations is crucial to deliver a scalable performance of the update operations. To do this, we have set a density threshold that acts as limiting factor, bringing a good amortised cost of insertion and deletion within a ΔNode, and subsequently for the whole ΔTree. The proof for the amortised cost are given in Section 4 of this paper.

Concerning the EXPAND operation, an amount of memory for a new ΔNode needs to be allocated during runtime. Since we kept the size of a ΔNode equal to the page size, memory allocation routine for new ΔNodes does not require plenty of CPU cycles.

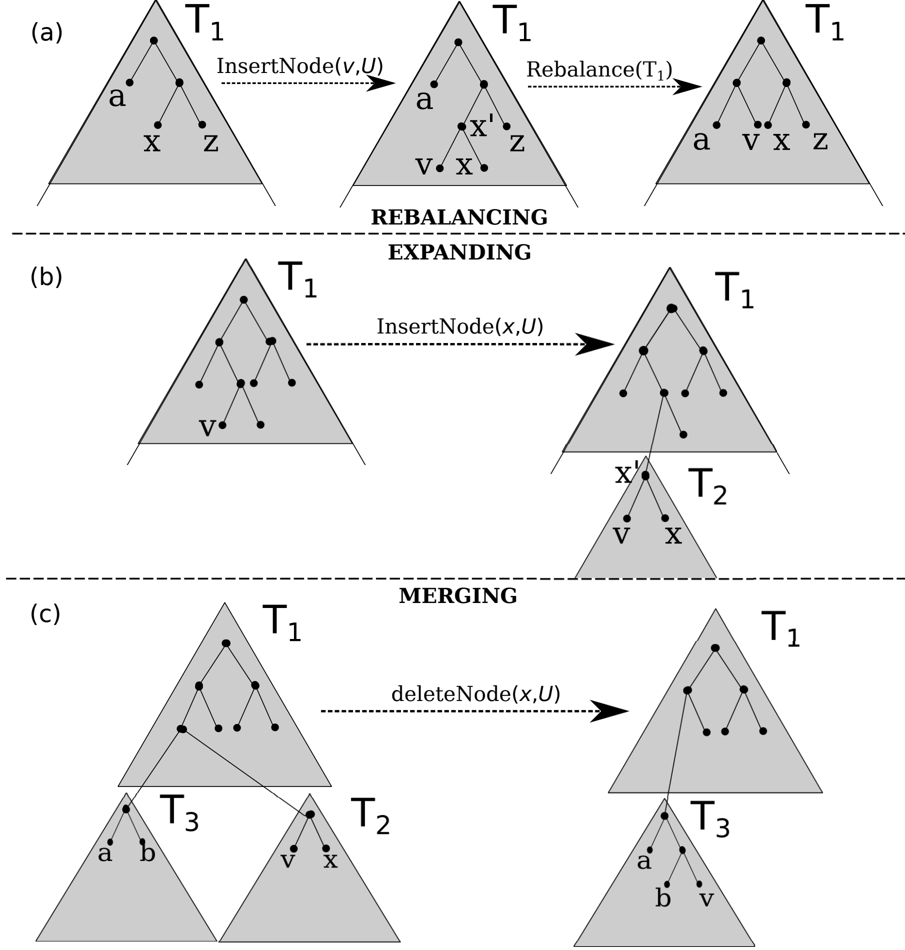

Figure 5: (a)*Rebalancing*, (b)*Expanding*, and (c)*Merging* operations on ΔTree

4 Detailed Implementation

4.1 Function specifications

The searchNode(v, U) function will return **true** whenever value v has been inserted in one of the ΔTree (U) leaf node and that node's *mark* property is currently set to **false**. Or if v is placed on one of the ΔNode's buffer located at the lowest level of U. It returns **false** whenever it couldn't find a leaf node with $value = v$, or v couldn't be found in the last level $T_{tid}.rootbuffer$.

insertNode(v, U) will insert value v and returns **true** if there is no leaf node with $value = v$, or there is a leaf node x which satisfy $x.value = v$ but with $x.mark =$ **true**, or v is not found in the last T_{tid}'s $rootbuffer$. In the other hand, insertNode returns **false** if there is a leaf node with $value = v$ and $mark =$ **false**, or v is found in $T_{tid}.rootbuffer$.

For deleteNode(v, U), a value of **true** is returned if there is a leaf node with $value = v$ and $mark =$ **false**, or v is found in the last T_{tid}'s $rootbuffer$. The value v will be then deleted. In the other hand, deleteNode returns **false** if there is a leaf node with $value = v$ and $mark =$ **true**, or v is not found in $T_{tid}.rootbuffer$.

```
1: function WAITANDCHECK(lock, opcount)
2:     do
3:         SPINWAIT(lock)
4:         FLAGUP(opcount)
5:         repeat ← false
6:         if lock = true then
7:             FLAGDOWN(opcount)
8:             repeat ← true
9:     while repeat = TRUE
```

Figure 6: Wait and check algorithm

4.2 Synchronisation calls

For synchronisation between update and maintenance operations, we define FLAGUP(*opcount*) that is doing atomic increment of *opcount* and also a function that do atomic decrement of *opcount* as FLAGDOWN(*opcount*).

Also there is SPINWAIT(*lock*) that basically instruct a thread to spin waiting while *lock* value is **true**. Only MERGE and REBALANCE that will have to privilege to set $T_x.lock$ as **true**. Lastly there is WAITANDCHECK(*lock*, *opcount*) function (Figure 6) that is preventing updates in getting mixed-up with maintenance operations. The mechanism of WAITANDCHECK(*lock*, *opcount*) will instruct a thread to wait at the tip of a current ΔNode whenever another thread has obtained a lock on that ΔNode for the purpose of doing any maintenance operations.

4.3 Wait-free and Linearisability of search

Lemma 4.1 *ΔTree search operation is wait-free.*

Proof. (Sketch) In the searching algorithm (cf. Figure 8), the ΔTree will be traversed from the root node using iterative steps. When at *root*, the value to search v is compared to *root.value*. If $v < root.value$, the left side of the tree will be traversed by setting $root \leftarrow root.left$ (line 5), in contrary $v \geqslant root.value$ will cause the right side of the tree to be traversed further (line 7). The procedure will repeat until a leaf has been found ($v.isleaf$ = **true**) in line 3.

If the value v couldn't be found and search has reached the end of ΔTree, a buffer search will be conducted (line 15). This search is done by simply searching the buffer array from left-to-right to find v, therefore no waiting will happen in this phase.

The DELETENODE and INSERTNODE algorithms (Figure 9) are non-intrusive to the structure of a tree, thus they won't interfere with an ongoing search. A DELETENODE operation, if succeeded, is only going to mark a node by setting a *v.mark* variable as **true** (line 18 in Figure 9). The *v.value* is retained so that a search will be able to proceed further. For INSERTNODE, it can *"grow"* the current leaf node as it needs to lays down two new leaves (lines 52 and 63 in Figure 9), however the operation never changes the internal pointer structure of a ΔNode, since ΔNode internal tree structure is pre-allocated beforehand, allowing a search to keep moving forward. As depicted in Figure 5(a), after an insertion of v grows the node, the old node (now x') still contains the same value

1: **Struct node** n:
2: member fields:
3: $tid \in \mathbb{N}$, if > 0 indicates the node is root of a ΔNode with an id of tid (T_{tid})
4: $value \in \mathbb{N}$, the node value, default is **empty**
5: $mark \in \{true, false\}$, a value of $true$ indicates a logically deleted node
6: $left, right \in \mathbb{N}$, left / right child pointers
7: $isleaf \in true, false$, indicates whether the node is a leaf of a ΔNode, default is $true$

8: **Struct ΔNode** T:
9: member fields:
10: $nodes$, a group of $(|T| \times 2)$ amount of pre-allocated node n.
11: $rootbuffer$, an array of value with a length of the current number of threads
12: $mirrorbuffer$, an array of value with a length of the current number of threads
13: $lock$, indicates whether a ΔNode is locked
14: $flag$, semaphore for active update operations
15: $root$, pointer the root node of the ΔNode
16: $mirror$, pointer to root node of the ΔNode's mirror

17: **Struct universe** U:
18: member fields:
19: $root$, pointer to the $root$ of the topmost ΔNode ($T_1.root$)

Figure 7: Cache friendly binary search tree structure

as x (assuming $v < x$), thus a search still can be directed to find either v or x. The REBALANCE/MERGE operation is also not an obstacle for searching since its operating on a mirror ΔNode. □

We have designed the searching to be linearisable in various concurrent operation scenarios (Lemma 4.2). This applies as well to the update operations.

Lemma 4.2 *For a value that resides on the leaf node of a ΔNode,* SEARCHNODE *operation (Figure 8) has the linearisation point to* DELETENODE *at line 10 and the linearisation point to* INSERTNODE *at line 9. For a value that stays in the buffer of a ΔNode,* SEARCHNODE *operation has the linearisation point at line 16.*

Proof. (Sketch) It is trivial to demonstrate this in relation to deletion algorithm in Figure 9 since only an atomic operation is responsible for altering the *mark* property of a node (line 18). Therefore DELETENODE has the linearisation point to SEARCHNODE at line 18.

For SEARCHNODE interaction with an insertion that grows new subtree, we rely on the facts that: 1) a snapshot of the current node p is recorded on *lastnode* as a first step

```
1: function searchNode(v, U)
2:     lastnode, p ← U.root
3:     while p ≠ not end of tree & p.isleaf ≠ TRUE do
4:         lastnode ← p
5:         if p.value < v then
6:             p ← p.left
7:         else
8:             p ← p.right
9:     if lastnode.value = v then
10:        if lastnode.mark = FALSE then
11:            return TRUE
12:        else
13:            return FALSE
14:    else
15:        Search (T_tid.rootbuffer) for v
16:        if v is found then
17:            return TRUE
18:        else
19:            return FALSE
```

Figure 8: A wait-free searching algorithm of ΔTree

of searching iteration (Figure 8, line 4); 2) *v.value* change, if needed, is not done until the last step of the insertion routine for insertion of $v > node.value$ and will be done in one atomic step with *node.isleaf* change (Figure 9, line 66); and 3) *isleaf* property of all internal nodes are by default **true** (Figure 7, line 7) to guarantee that values that are inserted are always found, even when the leaf-growing (both left-and-right) are happening concurrently. Therefore INSERTNODE has the linearisation point to SEARCHNODE at line 52 when inserting a value v smaller than the leaf node's value, or at line 63 otherwise.

A search procedure is also able to cope well with a "buffered" insert, that is if an insert thread loses a competition in locking a ΔNode for expanding or rebalancing and had to dump its carried value inside a buffer (Figure 9, line 89). Any value inserted to the buffer is guaranteed to be found. This is because after a leaf *lastnode* has been located, the search is going to evaluate whether the *lastnode.value* is equal to v. Failed comparison will cause the search to look further inside a buffer ($T_x.rootbuffer$) located in a ΔNode where the leaf resides (Figure 8, line 15). By assuring that the switching of a root ΔNode with its mirror includes switching $T_x.rootbuffer$ with $T_x.mirrorbuffer$, we can show that any new values that might be placed inside a buffer are guaranteed to be found immediately after the completion of their respective insert procedures. The "buffered" insert has the linearisation point to SEARCHNODE at line 89.

Similarly, deleting a value from a buffer is as trivial as exchanging that value inside a buffer with an **empty** value. The search operation will failed to find that value when doing searching inside a buffer of ΔNode. This type of delete has the linearisation point to SEARCHNODE at the same line it's emptying a value inside the buffer (line 29). □

```
1:  function insertNode(v, U)                         ▷ Inserting an new item v into ΔTree U
2:      t ← U.root
3:      return insertHelper(v, t)
4:
5:  function deleteNode(v, T)                         ▷ Deleting an item v from ΔTree U
6:      t ← U.root
7:      return deleteHelper(v, t)
8:
9:  function deleteHelper(v, node)
10:     success ← TRUE
11:     if Entering new ΔNode T_x then                ▷ Observed by examining x ← node.tid value
                                                        change
12:         T'_x ← getParentΔNode(T_x)
13:         flagdown(T'_x.opcount)                    ▷ Flagging down operation count on the previ-
                                                        ous/parent triangle
14:         waitandcheck(T_x.lock, T_x.opcount)
15:         flagup(T_x.opcount)
16:     if (node.isleaf = TRUE) then                                      ▷ Are we at leaf?
17:         if node.value = v then
18:             if CAS(node.mark, FALSE, TRUE) != FALSE) then             ▷ Mark it delete
19:                 success ← FALSE                          ▷ Unable to mark, already deleted
20:             else
21:                 if (node.left.value=empty&node.right.value=empty) then
22:                     T_x.bcount ← T_x.bcount − 1
23:                     mergeNode(parentOf(T_x)) ← TRUE ▷ Delete succeed, invoke merging
24:                 else
25:                     deleteHelper(v, node)   ▷ Not leaf, re-try delete from node
26:         else
27:             Search (T_x.rootbuffer) for v
28:             if v is found in T_x.rootbuffer.idx then
29:                 T_x.rootbuffer.idx ← empty
30:                 T_x.bcount ← T_x.bcount − 1
31:                 T_x.countnode ← T_x.countnode − 1
32:             else
33:                 flagdown(T_x.opcount)
34:                 success ← FALSE               ▷ Value not found
35:         flagdown(T_x.opcount)
36:     else
37:         if v < node.value then
38:             deleteHelper(v, node.left)
39:         else
40:             deleteHelper(v, node.right)
41:     return success
```

```
42: function insertHelper(v, node)
43:     success ← TRUE
44:     if Entering new ΔNode T_x then              ▷ Observed by examining x ← node.tid change
45:         T'_x ← getParentΔNode(T_x)
46:         flagdown(T'_x.opcount)                  ▷ Flagging down operation count on the previ-
                                                    ous/parent triangle
47:         waitandcheck(T_x.lock, T_x.opcount)
48:         flagup(T_x.opcount)
49:     if node.left & node.right then              ▷ At the lowest level of a ΔTree?
50:         if v < node.value then
51:             if (node.isleaf = TRUE) then
52:                 if CAS(node.left.value, empty, v) = empty then
53:                     node.right.value ← node.value
54:                     node.right.mark ← node.mark
55:                     node.isleaf ← FALSE
56:                     flagdown(T_x.opcount)
57:                 else
58:                     insertHelper(v, node)       ▷ Else try again to insert starting with the same
                                                    node
59:             else
60:                 insertHelper(v, node.left)      ▷ Not a leaf, proceed further to find the leaf
61:         else if v > node.value then
62:             if (node.isleaf = TRUE) then
63:                 if CAS(node.left.value, empty, v) = empty then
64:                     node.right.value ← v
65:                     node.left.mark ← node.mark
66:                     atomic { node.value ← v
67:                     node.isleaf ← FALSE }
68:                     flagdown(T_x.opcount)
69:                 else
70:                     insertHelper(v, node)       ▷ Else try again to insert starting with the same
                                                    node
71:             else
72:                 insertHelper(v, node.right)     ▷ Not a leaf, proceed further to find the leaf
73:         else if v = node.value then
74:             if (node.isleaf = TRUE) then
75:                 if node.mark = FALSE then
76:                     success ← FALSE             ▷ Duplicate Found
77:                     flagdown(T_x.opcount)
78:                 else
79:                     Goto 63
80:             else
81:                 insertHelper(v, node.right)     ▷ Not a leaf, proceed further to find the leaf
82:     else
83:         if val = node.value then
84:             if node.mark = 1 then
85:                 success ← FALSE
86:         else                                    ▷ All's failed, need to rebalance or expand the
                                                    triangle T_x
```

```
87:         if v already in T_x.rootbuffer then success ← FALSE
88:         else
89:             put v inside T_x.rootbuffer
90:             T_x.bcount ← T_x.bcount + 1
91:             T_x.countnode ← T_x.countnode + 1
92:         if TAS(T_x.lock) then               ▷ All threads try to lock T_x
93:             FLAGDOWN(T_x.opcount)           ▷ Make sure no flag is still raised
94:             SPINWAIT(T_x.opcount)           ▷ Now wait all insert/delete operations to finish
95:             total ← T_x.countnode + T_x.bcount
96:             if total * 4 ⩾ U.maxnode + 1 then          ▷ Expanding needed, density > 0.5
97:                 ... Create(a new triangle) AND attach it on the to the parent of node ...
98:             else
99:                 if T_x don't have triangle child(s) then
100:                    T_x.mirror ← REBALANCE(T_x.root, T_x.rootbuffer)
101:                    SWITCHTREE(T_x.root, T_x.mirror)
102:                    T_x.b_count ← 0
103:                else
104:                    if T_x.bcount > 0 then
105:                        Fill childA with all value in T_x.rootbuffer    ▷ Do non-blocking
                                                                            insert here
106:                        T_x.bcount ← 0
107:            SPINUNLOCK(T_x.lock)
108:        else
109:            FLAGDOWN(T_x.opcount)
110:    return success
```

Figure 9: Update algorithms and their helpers functions

4.4 Non-blocking Update Operations

Lemma 4.3 *ΔTree Insert and Delete operations are non-blocking to each other in the absence of maintenance operations.*

Proof. (Sketch) Non-blocking update operations supported by ΔTree are possible by assuming that any of the updates are not invoking REBALANCE and MERGE operations. In a case of concurrent insert operations (Figure 9) at the same leaf node x, assuming all insert threads need to "grow" the node (for illustration, cf. Figure 5), they will have to do CAS($x.left$, **empty**, ...) (line 52 and 63) as their first step. This CAS is the only thing needed since the whole ΔNode structure is pre-allocated and the CAS is an atomic operation. Therefore, only one thread will succeed in changing $x.left$ and proceed populating the $x.right$ node. Other threads will fail the CAS operation and they are going to try restart the insert procedure all over again, starting from the node x.

To assure that the marking delete (line 18) behaves nicely with the "grow" insert operations, DELETENODE(v, U) that has found the leaf node x with a value equal to v, will need to check again whether the node is still a leaf (line 21) after completing CAS($x.mark, FALSE, TRUE$). The thread needs to restart the delete process from x if it has found that x is no longer a leaf node.

The absence of maintenance operations means that a ΔNode *lock* is never set to **true**, thus either insert/delete operations are never blocked at the execution of line number 63

```
1:  procedure balanceTree(T)
2:      Array temp[|H|] ← Traverse(T)          ▷ Traverse all the non-empty node into temp array
3:      RePopulate(T, temp)                     ▷ Re-populate the tree T with all the value from
                                                  temp recursively. RePopulate will resulting a
                                                  balanced tree T
4:  procedure mergeTree(root)
5:      parent ← parentOf(root)
6:      if parent.left = root then
7:          sibling ← parent.right
8:      else
9:          sibling ← parent.left
10:     T_r ← triangleOf(root)                  ▷ Get the T of root node
11:     T_p ← triangleOf(parent)                ▷ Get the T of parent
12:     T_s ← triangleOf(sibling)               ▷ Get the T of sibling
13:     if spinTryLock(T_r.lock) then           ▷ Try to lock the current triangle
14:         spinLock(T_s.lock, T_p.lock)        ▷ lock the sibling triangles
15:         flagDown(T_r.opcount)
16:         spinWait(T_r.opcount, T_s.opcount, T_p.opcount)   ▷ Wait for all insert/delete
                                                                operations to finish
17:         total ← T_s.nodecount + T_s.bcount + T_r.nodecount + T_r.bcount
18:         if (T_s & T_r don't have children) & (T_p ⩾ U.maxnode + 1)/2)||T_s ⩾ U.maxnode +
        1)/2)) & total ⩽ (U.maxnode + 1)/2 then
19:             MERGE T_r.root, T_r.rootbuffer, T_s.rootbuffer into T.s
20:             if parent.left = root then      ▷ Now re-do the pointer
21:                 parent.left ← root.left     ▷ Merge Left
22:             else
23:                 parent.right ← root.right   ▷ Merge Right
24:         spinUnlock(T_r.lock, T_s.lock, T_p.lock)
25:     else
26:         flagDown(T_r.opcount)
```

Figure 10: Merge and Balance algorithm

in Figure 6. □

Lemma 4.4 *In Figure 9,* insertNode *operation has the linearisation point against* deleteNode *at line 52 and line 63. Whereas* deleteNode *has a linearisation point at line 21 against an* insertNode *operation. For inserting and deleting into a buffer of a* Δ*Node, an* insertNode *operation has the linearisation point at line 89. While* deleteNode *has its linearisation point at line 29.*

Proof. (Sketch) An insertNode operation will do a CAS on the left node as its first step after finding a suitable *node* for growing a subtree. If value v is lower than *node.value*, the correspondent operation is the line 52. Line 63 is executed in other conditions. A deleteNode will always check a *node* is still a leaf by ensuring $node.left.value$ as **empty** (line 21). This is done after it tries to mark that *node*. If the comparison on line 21 returns **true**, the operation finishes successfully. A **false** value will instruct the insertNode to retry again, starting from the current node.

A buffered insert and delete are operating on the same buffer. When a value v is put inside a buffer it will always available for delete. And that goes the opposite for the deletion case. □

4.5 Memory Transfer and Time Complexities

In this subsection, we will show that ΔTree is relaxed cache oblivious and the overhead of maintenance operations (e.g. rebalancing, expanding and merging) is negligible for big trees. The memory transfer analysis is based on the ideal-cache model [FLPR99]. Namely, re-accessing data in cache due to re-trying in non-blocking approaches incurs no memory transfer.

For the following analysis, we assume that values to be searched, inserted or deleted are randomly chosen. As ΔTree is a binary search tree (BST), which is embedded in the dynamic vEB layout, the expected height of a randomly built ΔTree of size N is $O(\log N)$ [CSRL01].

Lemma 4.5 *A search in a randomly built ΔTree needs $O(\log_B N)$ expected memory transfers, where N and B is the tree size and the* unknown *memory block size in the ideal cache model [FLPR99], respectively.*

Proof. (Sketch) Similar to the proof of Lemma 2.1, let k, L be the coarsest levels of detail such that every recursive subtree contains at most B nodes or UB nodes, respectively. Since $B \leq UB$, $k \leq L$. There are at most 2^{L-k} subtrees along to the search path in a ΔNode and no subtree of depth 2^k is split due to the boundary of ΔNodes (cf. Figure 3). Since every subtree of depth 2^k fits in a ΔNode of size UB, the subtree is stored in at most 2 memory blocks of size B.

Since a subtree of height 2^{k+1} contains more than B nodes, $2^{k+1} \geq \log_2(B+1)$, or $2^k \geq \frac{1}{2}\log_2(B+1)$.

Since a randomly built ΔTree has an expected height of $O(\log N)$, there are $\frac{O(\log N)}{2^k}$ subtrees of depth 2^k are traversed in a search and thereby at most $2\frac{O(\log N)}{2^k} = O(\frac{\log N}{2^k})$ memory blocks are transferred.

As $\frac{\log N}{2^k} \leq 2\frac{\log N}{\log(B+1)} = 2\log_{B+1} N \leq 2log_B N$, expected memory transfers in a search are $O(\log_B N)$. □

Lemma 4.6 *Insert and Delete operations within the ΔTree are having a similar amortised time complexity of $O(\log n + UB)$, where n is the size of ΔTree, and UB is the maximum size of element stored in ΔNode.*

Proof. (Sketch) An insertion operation at ΔTree is tightly coupled with the rebalancing and expanding algorithm.

We assume that ΔTree was built using random values, therefore the expected height is $\mathcal{O}(\log n)$. Thus, an insertion on a ΔTree costs $\mathcal{O}(\log n)$. Rebalancing after insertion only happens at single ΔNode, and it has an upper bound cost of $\mathcal{O}(UB + UB \log UB)$, because it has to read all the stored elements, sort it out and re-insert it in a balanced fashion. In the worst possible case for ΔTree, there will be an n insertion that cost $\log n$ and there is at most n rebalancing operations with a cost of $\mathcal{O}(UB + UB \log UB)$ each.

Using aggregate analysis, we let total cost for insertion as $\sum_{k=1}^{n} c_i \leqslant n \log n + \sum_{k=1}^{n} UB + UB \log UB \approx n \log n + n \cdot (UB + UB \log UB)$. Therefore the amortised time complexity for insert is $\mathcal{O}(\log n + UB + UB \log UB)$. If we have defined UB such as $UB << n$, the amortised time complexity for inserting a value into ΔTree is now becoming $\mathcal{O}(\log n)$.

For the expanding scenarios, an insertion will trigger $\text{EXPAND}(v)$ whenever an insertion of v in a ΔNode T_j is resulting on $depth(v) = H(T_j)$ and $|T_j| \geqslant (2^{H(T_j)-1}) - 1$. An expanding will require a memory allocation of a UB-sized ΔNode, cost merely $\mathcal{O}(1)$, together with two pointer alterations that cost $\mathcal{O}(1)$ each. In conclusion, we have shown that the total amortised cost for insertion, that is incorporating both rebalancing and expanding procedures as $\mathcal{O}(\log n)$.

In the deletion case, right after a deletion on a particular ΔNode will trigger a merging of that ΔNode with its sibling in a condition of at least one of the ΔNodes is filled less than half of its maximum capacity ($density(v) < 0.5$) and all values from both ΔNodes can fit into a single ΔNode.

Similar to insertion, a deletion in ΔTree costs $\log n$. However merging that combines 2 ΔNodes costs $2UB$ at maximum. Using aggregate analysis, the total cost of deletion could be formulated as $\sum_{k=1}^{n} c_i \leqslant n \log n + \sum_{k=1}^{n} 2 \cdot UB \approx n \log n + 2n \cdot UB$. The amortised time complexity is therefore $\mathcal{O}(\log n + UB)$ or $\mathcal{O}(\log n)$, if $UB << n$.

□

5 Experimental Result and Discussion

To evaluate our conceptual idea of ΔTree, we compare its implementation performance with those of STM-based AVL tree (AVLtree), red-black tree (RBtree), and Speculation Friendly tree (SFtree) in the Synchrobench benchmark [Gra]. We also have developed an STM-based binary search tree which is based on the work of [BFJ02] utilising GNU C Compiler's STM implementation from the version 4.7.2 . This particular tree will be referred as VTMtree, and it has all the traits of vEB tree layout, although it only has a fixed size, which is pre-defined before the runtime. Pthreads were used for concurrent threads and the GCC were invoked with -O2 optimisation to compile all of the programs.

The base of the conducted experiment consists of running a series of ($rep = 100,000,000$) operations. Assuming we have nr as the number of threads, the time for a thread to finish a sequence of rep/nr operations will be recorded and summed with the similar measurement from the other threads. We also define an update rate u that translates to $upd = u\% \times rep$ number of insert and delete operations and $src = rep - upd$ number of search operations out of rep. We set a consecutive run for the experiment to use a combination of update rate $u = \{0, 1, 3, 5, 10, 20, 100\}$ and number of thread $nr = \{1, 2, \ldots, 16\}$ for each runs. Update rate of 0 means that only searching operations were conducted, while 100 update rate indicates that no searching were carried out, only insert and delete operations. For each of the combination above, we pre-filled the initial tree using 1,023 and 2,500,000 values. A ΔTree with initial members of 1,023 increases the chances that a thread will compete for a same resources and also simulates a condition where the whole tree fits into the cache. The initial size of 2,500,000 lowers the chance of thread contentions and simu-

lates a big tree that not all of it will fits into the last level of cache. The operations involved (e.g. searching, inserting or deleting) used random values $v \in (0, 5,000,000], v \in \mathbb{N}$, as their parameter for searching, inserting or deleting. Note that VTMtree is fixed in size, therefore we set its size to 5,000,000 to accommodate this experiment.

The conducted experiment was run on a dual Intel Xeon CPU E5-2670, for a total of 16 available cores. The node had 32GB of memory, with a 2MB L2 cache and a 20MB L3 cache. The Hyperthread feature of the processor was turned off to make sure that the experiments only ran on physical cores. The performance (in operations/second) result for update operations were calculated by adding the number successful insert and delete. While searching performance were using the number of attempted searches. Both were divided by the total time required to finish *rep* operations.

In order to satisfy the locality-aware properties of the ΔTree, we need to make sure that the size of ΔNodes, or UB, not only for Lemma 2.1 to hold true, but also to make sure that all level of the memory hierarchy (L1, L2, ... caches) are efficiently utilised, while also minimising the frequency of false sharing in a highly contended concurrent operation. For this we have tested various value for UB, using 127, $1K - 1$, $4K - 1$, and $512K - 1$ sized elements, and by assuming a node size in the ΔNode is 32 bytes. These values will correspond to 4 Kbytes (page size for most systems), L1 size, L2 size, and L3 size respectively. Please note that L1, L2, and L3 sizes here are measured in our test system. Based on the result of this test, we found out that $UB = 127$ delivers the best performance results, in both searching and updating. This is in-line with the facts that the page size is the block size used during memory allocation [Smi82, Dre07]. This improves the transfer rate from main memory to the CPU cache. Having a ΔNode that fits in a page will help the ΔTree in exploiting the data locality property.

As shown in Figure 11, under a small tree setup, the ΔTree has a better update performance (i.e. insertion and deletion) compared to the other trees, whenever the update ratio is less than 10%. From the said figure, 10% update ratio seems to be the cut-off point for ΔTree before SFtree, AVLtree, and RBtree gradually took over the performance lead. Even though the update rate of the ΔTree were severely hampered after going on higher than 10% update ratio, it does manage to keep a comparable performance for a small number of threads.

For the search performance evaluation using the same setup, ΔTree is superior compared to other types of tree when the search ratio higher than 90% (cf. Figure 11). In the extreme case of 100% search ratio (i.e. no update operation), ΔTree does however get beaten by the VTMtree.

On the other setup, the big tree setup with an initial member of 2,500,000 nodes (cf. Figure 12), a slightly different result on update performance can be observed. Here the ΔTree maintains a lead in the concurrent update performance up to 20% update ratio. Higher ratio than this diminishes the ΔTree concurrent update performance superiority. Similar to what can bee seen at the small tree setup, during the extreme case of 100% update ratio (i.e. no search operation), the ΔTree seems to be able to kept its pace for 6 threads, before flattening-out in the long run, losing out to the SFtree, AVLtree, and RBtree. VTMtree update performance is the worst.

As for the concurrent searching performance in the same setup, the ΔTree outperforms the other trees when the search ratio is less than 100%. At the 80 % search ratio, the VTMtree search performance is the worst and the search performance of the other four

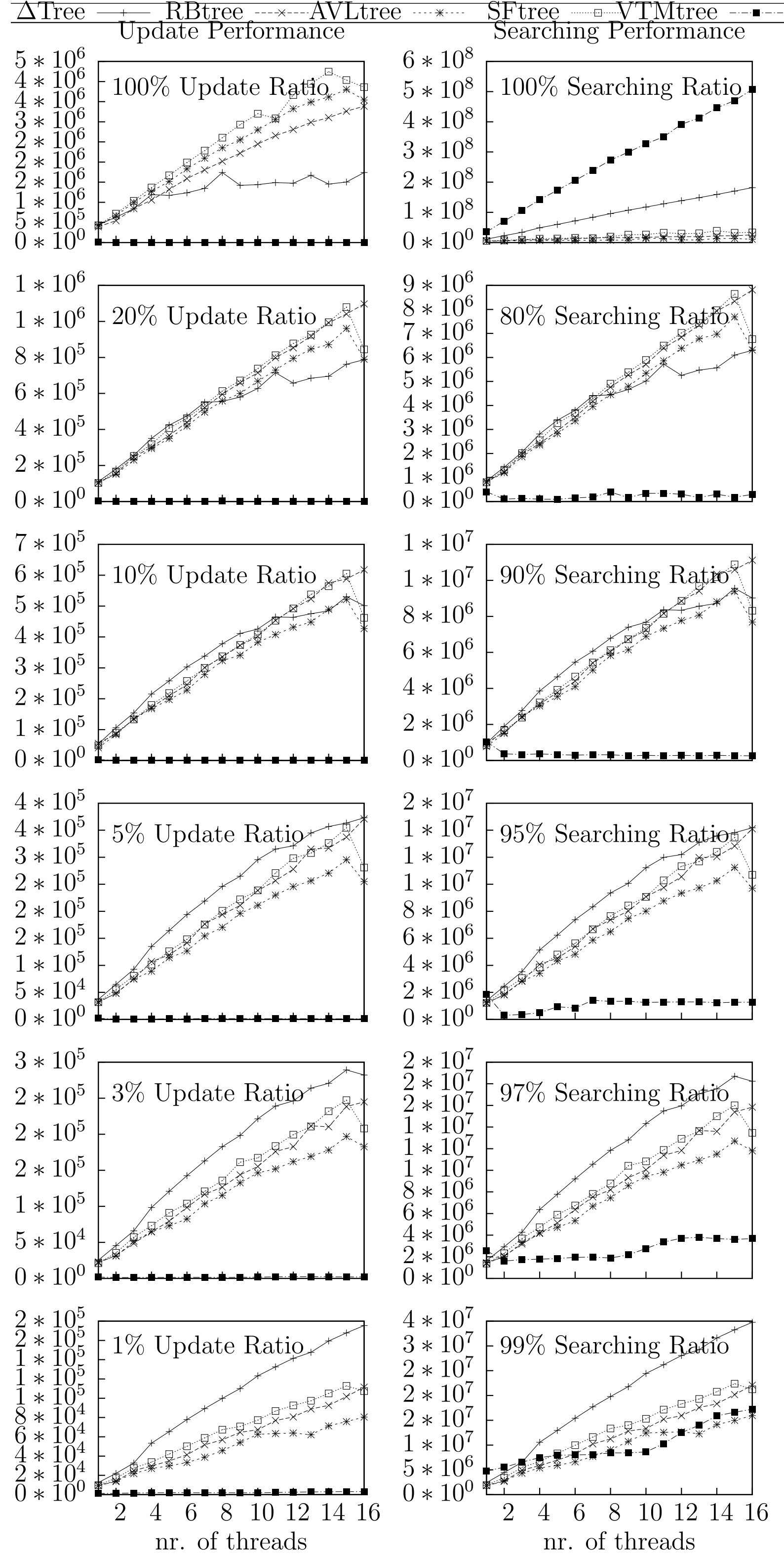

Figure 11: Performance rate (operations/second) of a ΔTree with 1,023 initial members. The y-axis indicates the rate of operations/second.

Figure 12: Performance rate (operations/second) of a ΔTree with 2,500,000 initial members. The y-axis indicates the rate of operations/second.

Table 1: Cache profile comparison during 100% searching

Tree	**Load count**	**Last level cache miss**	**%miss**	**operations /second**
ΔTree ($UB = 5M$)	4301253301	454239096	10.56	469942
ΔTree ($UB = 127$)	4895074596	435682140	8.90	429945
SFtree	3675847131	406925489	11.07	85473
VTMtree	1140447360	62794247	5.51	2261378

trees is comparable. At the extreme case of 100% search ratio, the VTMtree performance is the best.

The ΔTree performs well in the low-contention situations. Whenever the a big tree setup is used, the ΔTree delivers scalable updating and searching performance up to 20% update ratio, compared to only 10% update ratio in the small tree setup. The good update performance of ΔTree can be attributed to the dynamic vEB layout that permits that multiple different ΔNodes can be concurrently updated and restructured. Keeping the frequency of restructuring done by MERGE and REBALANCE at low also contribute to this good performance. In terms of searching, the ΔTree have been showing an overall good performance, which only gets beaten by the static vEB layout-based VTMtree at the extreme case of 100% searching ratio.

In order to get better insight into the performance ΔTree, we conducted additional experiment targeting the cache behaviour of the different trees. In this experiment, two flavours of ΔTree, one using ΔNode size of 127 and another using a size of 5,000,000, together with both VTMtree and SFtree were put to do 100M searching operations. Big ΔNode size in this experiment simulates a *leaf-oriented* static vEB, with only 1 ΔNode involved, whereas the VTMtree simulates a *original* static vEB where values can be stored at internal nodes. Those trees are pre-filled with 1,048,576 random non-recurring numbers within $(0, 5, 000, 000]$ range. The values searched for were randomly picked as well within the same range. Cache profiles were then collected using Valgrind [NWF06] Our test system has 20MB of CPU's L3 cache, therefore the pre-initialised nodes were not entirely contained within the cache ($1048576 \times 32\text{B} > 20\text{MB}$). This experiment result in Table 1 proved that using the dynamic vEB layout were indeed able to reduce the number of cache misses by almost 2%. This is observed by comparing the percentage of cache misses between leaf-oriented static vEB ΔTree ($UB = 5M$) and leaf-oriented dynamic vEB ΔTree ($UB = 127$). However it doesn't translate to a higher update rate due to increasing load count.

It is interesting to see that VTMtree is able to deliver the lowest load count as well as the lowest number of cache misses. This result leads us to conclude that using leaf-oriented tree for the sake of supporting scalable concurrent updates, has a downside of introducing more cache misses. This can be related to the fact that a search in leaf-oriented tree has to always traverse all the way down to the leaves. Although using dynamic vEB really improves locality property, traversing down further to leaf will cause data inside the cache to be replaced more often.

The bad performance of VTMtree's concurrent update on both of the tree setups are inevitable, because of the nature of static tree layout. The VTMtree needs to always maintain a small height, which is done by *incrementally* rebalancing different portions

of its structure[BFJ02]. In case of VTMtree, the whole tree must be locked whenever rebalance is executed, blocking other operations. While [BFJ02] explained that amortised cost for this is small, it will hold true only in when implemented in the sequential fashion.

6 Related Work

The trees involved in the benchmark section are not all the available implementation of the concurrent binary search tree. A novel non-blocking BST was coined in [EFRvB10], which subsequently followed by its k-ary implementation [BH11]. These researches are using leaf-oriented tree, the same principle used by ΔTree and it has a good concurrent operation performance. However the tree doesn't focus on high-performance searches, as the structure used is a normal BST. CBTree [AKK+12] tried to tackle good concurrent tree with its counting-based self-adjusting feature. But this too, didn't look at how an efficient layout can provide better search and update performance.

Also we have seen the work on concurrent cache-oblivious B-tree[BFGK05], which provides a good overview on how to combine efficient layout with concurrency. However its implementation was far from practical. The recent works in both [CGR12, CGR13] provides the current state-of-the art for the subject. However none of them targeted a cache-friendly structure which would ultimately lead to a more energy efficient data structure.

7 Conclusions and Future Work

We have introduced a new *relaxed cache oblivious* model that enables high parallelism while maintaining the key feature of the original cache oblivious (CO) model [Pro99] that analyses for a simple two-level memory are applicable for an unknown multilevel memory. Unlike the original CO model, the relaxed CO model assumes a known upper bound on unknown memory block sizes B of a multilevel memory. The relaxed CO model enables developing highly concurrent algorithms that can utilize fine-grained data locality as desired by energy efficient computing [Dal11].

Based on the relaxed CO model, we have developed a novel *dynamic* van Emde Boas (dynamic vEB) layout that makes the vEB layout suitable for highly-concurrent data structures with update operations. The dynamic vEB supports dynamic node allocation via pointers while maintaining the optimal search cost of $O(log_B N)$ memory transfers for vEB-based trees of size N without knowing memory block size B.

Using the dynamic van Emde Boas layout, we have developed ΔTree that supports both high concurrency and fine-grained data locality. ΔTree's *Search* operation is wait-free and its *Insert* and *Delete* operations are non-blocking to other *Insert, Delete* and *Search* operations. ΔTree is relaxed cache oblivious: the expected memory transfer costs of its *Search, Delete* and *Insert* operations are $O(log_B N)$, where N is the tree size and B is unknown memory block size in the ideal cache model [FLPR99]. Our experimental evaluation comparing ΔTree with AVL, red-black and speculation-friendly trees from the the Synchrobench benchmark [Gra] has shown that ΔTree achieves the best performance when the update contention is not too high.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n°611183 (EXCESS Project, www.excess-project.eu).

The authors would like to thank Gerth Stølting Brodal for the source code used in [BFJ02]. Vincent Gramoli who provides the source code for Synchrobench from [CGR12]. The Department of Information Technology, University of Tromsø for giving us access to the Stallo HPC cluster.

References

[AKK+12] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: a practical concurrent self-adjusting search tree. In *Proceedings of the 26th international conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[BCCO10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, 2010.

[BDFC05] Michael Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35:341, 2005.

[BFCF+07] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 81–92, 2007.

[BFGK05] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 228–237, 2005.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 39–48, 2002.

[BH11] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings of the 15th international conference on Principles of Distributed Systems*, OPODIS'11, pages 207–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[BP12] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 58–67, 2012.

[CGR12] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.

[CGR13] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proceedings of the 19th international conference on Parallel Processing*, Euro-Par'13, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[Dal11] Bill Dally. Power and programmability: The challenges of exascale computing. In *DoE Arch-I presentation*, 2011.

[Dre07] Ulrich Drepper. What every programmer should know about memory, 2007.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, 2006.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, 2010.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[Gra] Vincent Gramoli. Synchrobench: A benchmark to compare synchronization techniques for multicore programming. `https://github.com/gramoli/synchrobench`.

[Gra10] Goetz Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, July 2010.

[Gra11] Goetz Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, April 2011.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 339–350, 2010.

[NWF06] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 2–2, 2006.

[Pro99] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[vEB75] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.