Cache Oblivious Search Trees via Binary Trees of Small Height

Gerth Stølting Brodal* Rolf Fagerberg* Riko Jacob*

Abstract

We propose a version of cache oblivious search trees which is simpler than the previous proposal of Bender, Demaine and Farach-Colton and has the same complexity bounds. In particular, our data structure avoids the use of weight balanced B-trees, and can be implemented as just a single array of data elements, without the use of pointers. The structure also improves space utilization.

For storing n elements, our proposal uses $(1+\varepsilon)n$ times the element size of memory, and performs searches in worst case $O(\log_B n)$ memory transfers, updates in amortized $O((\log^2 n)/(\varepsilon B))$ memory transfers, and range queries in worst case $O(\log_B n + k/B)$ memory transfers, where k is the size of the output.

The basic idea of our data structure is to maintain a dynamic binary tree of height $\log n + O(1)$ using existing methods, embed this tree in a static binary tree, which in turn is embedded in an array in a cache oblivious fashion, using the van Emde Boas layout of Prokop.

We also investigate the practicality of cache obliviousness in the area of search trees, by providing an empirical comparison of different methods for laying out a search tree in memory.

1 Introduction

Modern computers contain a hierarchy of memory levels, with each level acting as a cache for the next. Typical components of the memory hierarchy are: registers, level 1 cache, level 2 cache, main memory, and disk. The time for accessing a level in the memory hierarchy increases from one cycle for registers and level 1 cache to figures around 10, 100, and 100,000 cycles for level 2 cache, main memory, and disk, respectively [13, p. 471], making the cost of a memory access depend highly on what is the current lowest memory level containing the element accessed. The evolution in CPU speed and memory access time indicates that these differences are likely to increase in the future [13, pp. 7 and 429].

As a consequence, the memory access pattern of an algorithm has become a key component in determining its running time in practice. Since classic asymptotical analysis of algorithms in the RAM model is unable to capture this, a number of more elaborate models for analysis have been proposed. The most widely used of these is the I/O model of Aggarwal and Vitter [1], which assumes a memory hierarchy containing two levels, the lower level having size M and the transfer between the two levels taking place in blocks of B elements. The cost of the computation in the I/O model is the number of blocks transferred. This model is adequate when the memory transfer between two levels of the memory hierarchy dominates the running time, which is often the case when the size of the data significantly exceeds the size of main memory, as the access time is very large for disks compared to the remaining levels of the memory hierarchy.

Recently, the concept of *cache oblivious* algorithms has been introduced by Frigo et al. [12]. In essence, this designates algorithms optimized in the I/O model, except that one optimizes to a block size B and a memory size M which are *unknown*. This seemingly simple change has significant consequences: since the analysis holds for any block and memory size, it holds for *all* levels of the memory hierarchy. In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to each level automatically. Furthermore, the characteristics of the memory hierarchy do not need to be known, and do not need to be hardwired into the algorithm for the analysis to hold. This increases the portability of implementations of the algorithm, which is important in many situations, including production of software libraries and code delivered over the web. For further details on the concept of cache obliviousness, see [12].

Frigo et al. [12] present optimal cache oblivious algorithms for matrix transposition, FFT, and sorting. Bender et al. [5], give a proposal for cache oblivious search trees with search cost matching that of standard (cache aware) B-trees [4]. While most of the results in [5, 12] are of theoretical nature, [12] contains some preliminary empirical investigations indicating the competitiveness of cache oblivious algorithms. The authors

*BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf,rjacob}@brics.dk. Partially supported by the FET part of the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

declare the determination of the range of practicality of cache oblivious algorithms an important avenue for future research.

In this paper, we study further the subject of cache oblivious search trees. In the first part, we propose a simplified version of the cache oblivious search trees from [5], achieving the same complexity bounds. In particular, our data structure avoids the use of weight balanced B-trees of Arge and Vitter [3], and it can be implemented in a single array of data elements without the use of pointers. Our structure also improves space utilization, implying that for given n, a larger fraction of the structure can reside in lower levels of the memory hierarchy. The lack of pointers also makes more elements fit in a block, thereby increasing the parameter B. These effects tend to decrease running time in practice. For storing n elements, our data structure uses $(1+\varepsilon)n$ times the element size of memory. Searches are performed in worst case $O(\log_B n)$ memory transfers, updates in amortized $O((\log^2 n)/(\varepsilon B))$ memory transfers, and range queries in worst case $O(\log_B n + k/B)$ memory transfers, where k is the size of the output. This matches the asymptotic complexities of [5]. We note that as in [5], the amortized complexity of updates can be lowered by the technique of substituting leaves with pointers to buckets each containing $\Theta(\log n)$ elements and maintaining the size bound of the buckets by splitting (merging) overflowing (underflowing) buckets. The price to pay is that ranges cannot be reported in the optimal number $\Theta(k/B)$ of memory transfers, since the buckets can reside in arbitrary positions in memory.

The basic idea of our data structure is to maintain a dynamic binary tree of height $\log n + O(1)$ using existing methods [2, 14], embed this tree in a static binary tree, which in turn is embedded in an array in a cache oblivious fashion, using the van Emde Boas layout [5, 19, 22]. The static structures are maintained by global rebuilding, i.e. they are rebuilt each time the dynamic tree has doubled or halved in size.

In the last part of this paper, we try to assess more systematically the impact of the memory layout of search trees by comparing experimentally the efficiency of the cache-oblivious van Emde Boas layout with a cache-aware layout based on multiway trees, and with classical layouts such as Breath First Search (BFS), Depth First Search (DFS), and inorder. Our results indicate that the nice theoretical properties of cache oblivious search trees actually do carry over into practice. We also implement our proposal, and confirm its practicality.

1.1 Related work. One technique used by our data structure is a cache oblivious layout of static binary search trees permitting searches in the asymptotically optimal number of memory transfers. This layout, the *van Emde Boas layout*, was proposed by Prokop [19, Section 10.2], and is related to a data structure of van Emde Boas [21, 22].

Another technique used is the maintenance of binary search trees of height $\log n + O(1)$ using local rebuildings of subtrees. The small height of the tree allows it to be embedded in a perfect binary tree (a tree with $2^k - 1$ internal nodes and optimal height) which has only a constant factor more nodes. Techniques for maintaining small height in binary trees were proposed by Andersson and Lai [2], who gave an algorithm for maintaining height $\lceil \log(n+1) \rceil + 1$ using amortized $O(\log^2 n)$ work per update. By viewing the tree as a linear list, this problem can be seen to be equivalent to the problem of maintaining n elements in sorted order in an array of length $O(n)$, using even redistribution of the elements in a section of the array as the reorganization primitive during insertions and deletions of elements. In this formulation, a similar solution had previously been given by Itai et al. [14], also using amortized $O(\log^2 n)$ work per update. In [9], a matching $\Omega(\log^2 n)$ lower bound for algorithms using this primitive was given.

Both the van Emde Boas layout and the technique of Itai et al. were used in the previous proposal for cache oblivious search trees [5]. The difficulty of this proposal originates mainly from the need to change the van Emde Boas layout during updates, which in turn necessitates the use of the weight balanced B-trees of Arge and Vitter [3]. By managing to use a static van Emde Boas layout (except for occasional global rebuildings of the entire structure), we avoid the use of weight balanced B-trees, and arrive at a significantly simpler structure.

Another improvement in our data structure is to avoid the use of pointers. The term *implicit* is often used for pointer-free implementations of trees and other data structures which are normally pointer based. One early example is the heap of Williams [23]. There is a large body of work dealing with implicit data structures, see e.g. [7, 11, 18] and the references therein. In that work, the term *implicit* is often defined as using only space for the n elements stored, plus $O(1)$ additional space. In the present paper, we will abuse the terminology a little, taking *implicit* to mean a structure stored entirely in an array of elements of length $O(n)$.

We note that independently, a data structure very similar to ours has been proposed by Bender et al. [6]. Essentially, their proposal is leaf-oriented, where ours is node-oriented. The leaf-oriented version allows an easy implementation of optimal scanning from any given location (the node-oriented version needs successor pointers for this), whereas the node-oriented version allows an implicit implementation, with the associated increase in B and decrease in memory usage.

The impact of different memory layouts for data structures has been studied before in different contexts. In connection with matrices, significant speedups can be achieved by using layouts optimized for the memory hierarchy—see e.g. the paper by Chatterjee et al. [8] and the references it contains. LaMarca and Ladner consider the question in connection with heaps [16]. Among other things, they repeat an experiment performed by Jones [15] ten years earlier, and demonstrate that due to the increased gaps in access time between levels in the memory hierarchy, the d-ary heap has increased competitiveness relative to the pointer-based priority queues. For search trees, B-trees are the standard way to implement trees optimized for the memory hierarchy. In the I/O-model, they use the worst case optimal number of memory transfers for searches. For external memory, they are the structure of choice, and are widely used for storing data base indexes. Also at the cache level, their memory optimality makes them very competitive to other search trees [17, p. 127].

Recently, Rahman and Raman [20] made an empirical study of the performance of various search tree implementations, with focus on showing the significance of also minimizing translation look-aside buffer (TLB) misses. Based on exponential search trees, they implemented a dynamization of the van Emde Boas layout supporting searches and updates in $O(\log_B(n) + \log\log n)$ memory transfers. They compared it experimentally to standard B-trees and three-level cache aware trees, and reported that the cache oblivious trees were better than standard B-trees but worse than the cache aware structures.

1.2 Preliminaries. As usual when discussing search trees, a *tree* is rooted and ordered. The *depth* $d(v)$ of a node v in a tree T is the number of nodes on the simple path from the node to the root. The *height* $h(T)$ of T is the maximum depth of a node in T, and the *size* $|T|$ of T is the number of nodes in T. For a node v in a tree, we let T_v denote the *subtree* rooted at v, i.e. the subtree consisting of v and all its descendants, and we let the height $h(v)$ of v be the height of T_v. A *complete tree* T is a tree with $2^{h(T)-1}$ nodes.

A *search tree* will denote a tree where all nodes store an element from some totally ordered universe, and where all elements stored in the left and right subtrees of a node v are respectively smaller than and larger than the element at v. We say that a tree T_1 can be *embedded* in another tree T_2, if T_1 can be obtained from T_2 by pruning subtrees. In Figure 1 is shown the embedding of a search tree of size 10 in a complete tree of height 5.

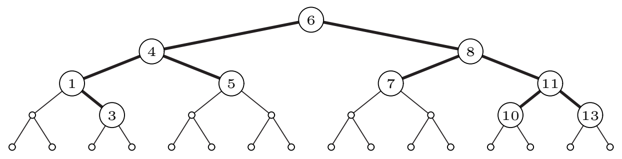

Figure 1: The embedding of a search tree with height 4 and size 10 in a complete tree with height 5

2 Memory Layouts of Static Trees

In this section we discuss four memory layouts for static trees: *DFS*, *inorder*, *BFS*, and *van Emde Boas* layouts. We assume that each node is represented by a node record and that all node records for a tree are stored in one array. We distinguish between *pointer based* and *implicit* layouts. In pointer based layouts the navigation between a node and its children is done via pointers stored in the node records. In implicit layouts no pointers are stored; the navigation is based solely on address arithmetic. Whereas all layouts have pointer based versions, implicit versions are only possible for layouts where the address computation is feasible. In this paper we will only consider implicit layouts of complete trees. A complete tree of size n is stored in an array of n node records.

DFS layout The nodes of T are stored in the order they are visited by a left-to-right depth first traversal of T (i.e. a preorder traversal).

Inorder layout The nodes of T are stored in the order that they are visited by a left-to-right inorder traversal of T.

BFS layout The nodes of T are stored in the order they are visited by a left-to-right breath first traversal of T.

van Emde Boas layout The layout is defined recursively: A tree with only one node is a single node record. If a tree T has two or more nodes, let $H_0 = \lceil h(T)/2 \rceil$, let T_0 be the tree consisting of all nodes in T with depth at most H_0, and let $T_1, \ldots, T_k$ be the subtrees of T rooted at nodes with depth $H_0 + 1$, numbered from left to right. We will denote T_0 the *top tree* and $T_1, \ldots, T_k$ the *bottom trees* of the recursion. The van Emde Boas layout of T consists of the van Emde Boas layout of T_0 followed by the van Emde Boas layouts of $T_1, \ldots, T_k$.

Figure 2 gives the implicit DFS, inorder, BFS, and van Emde Boas layouts for a complete tree with height four.

We now discuss how to calculate the position of the children of a node v at position i in the implicit layouts.

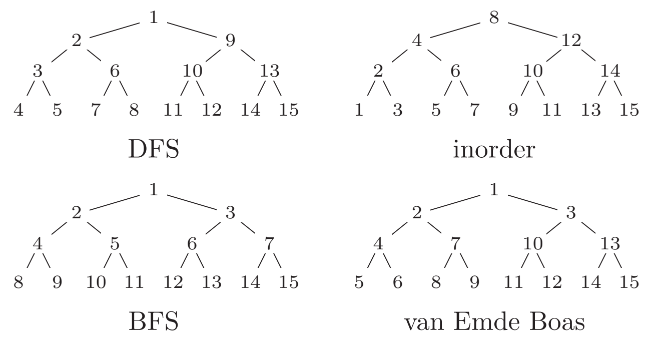

Figure 2: The DFS, inorder, BFS, and van Emde Boas layouts for a complete tree with height 4. Numbers designate positions in the array of node records

For the BFS layout, the children are at position $2i$ and $2i+1$—a fact exploited already in the 1960s in the design of the implicit binary heap [23]. For the DFS layout, the two children are at positions $i+1$ and $i+2^{h(v)-1}$, and in the inorder layout the two children are at positions $i-2^{h(v)-2}$ and $i+2^{h(v)-2}$.

For the implicit van Emde Boas layout the computations are more involved. Our solution is based on the fact that if we for a node in the tree unfold the recursion in the van Emde Boas layout until this node is the root of a bottom tree, then the unfolding will be the same for all nodes of the same depth. In a precomputed table of size $O(\log n)$, we for each depth d store the size $B[d]$ of this bottom tree, the size $T[d]$ of the corresponding top tree, and the depth $D[d]$ of the root of the corresponding top tree. When laying out a static tree, we build this table in $O(\log n)$ time by a straightforward recursive algorithm.

During a search from the root, we keep track of the position i in a BFS layout of the current node v of depth d. We also store the position $Pos[j]$ in the van Emde Boas layout of the node passed at depth j for $j < d$ during the current search. As the bits of the BFS number i represents the left and right turns made during the search, the $\log(T[d]+1)$ least significant bits of i gives the index of the bottom tree with v as root among all the bottom trees of the corresponding top tree. Since $T[d]$ is of the form 2^k-1, these bits can be found as i AND $T[d]$. It follows that for $d > 1$, we can calculate the position $Pos[d]$ of v by the expression

$$Pos[d] = Pos[D[d]] + T[d] + (i \text{ AND } T[d]) \cdot B[d] .$$

At the root, we have $i = 1$, $d = 1$, and $Pos[1] = 1$. Navigating from a node to a child is done by first calculating the new BFS position from the old, and then finding the value of the expression above.

The worst case number of memory transfers during a top down traversal of a path using the above layout schemes is as follows, assuming each block stores B nodes. With the BFS layout, the topmost $\lfloor\log(B+1)\rfloor$ levels of the tree will be contained in at most two blocks, whereas each of the following blocks read only contains one node from the path. The total number of memory transfers is therefore $\Theta(\log(n/B))$. For the DFS and inorder layouts, we get the same worst case bound when following the path to the rightmost leaf, since the first $\lceil\log(n+1)\rceil - \lceil\log B\rceil$ nodes have distance at least B in memory, whereas the last $\lfloor\log(B+1)\rfloor$ nodes are stored in at most two blocks. As Prokop [19, Section 10.2] observed, in the van Emde Boas layout there are at most $O(\log_B n)$ memory transfers. Note that only the van Emde Boas layout has the asymptotically optimal bound achieved by B-trees [4].

We note that DFS, inorder, BFS, and van Emde Boas layouts all support efficient range queries (i.e. the reporting of all elements with keys within a given query interval), by the usual recursive inorder traversal of the relevant part of the tree, starting at the root.

We argue below that the number of memory transfers for a range query in each of the four layouts equals the number of memory transfers for two searches plus $O(k/B)$, where k is the number of elements reported. If a range reporting query visits a node that is not contained in one of the search paths to the endpoints of the query interval, then all elements in the subtree rooted at the node will be reported. As a subtree of height $\lceil\log(B+1)\rceil$ stores between B and $2B-1$ elements, at most k/B nodes with height larger than $\lceil\log(B+1)\rceil$ are visited which are not on the search paths to the two endpoints. Since subtrees are stored contiguously for both the inorder and DFS layouts, a subtree of height $\lceil\log(B+1)\rceil$ is stored in at most three blocks. The claimed bound follows for these layouts. For the van Emde Boas layout, consider a subtree T of height $\lceil\log(B+1)\rceil$. There exists a level in the recursive layout where the topmost levels of T will be stored in a recursive top tree and the remaining levels of T will be stored in a contiguous sequence of bottom trees. Since the top tree and each bottom tree has size less than $2B-1$ and the bottom trees are stored contiguously in memory, the bound for range reportings in the van Emde Boas layout follows.

For the BFS layout, we prove the bound under the assumption that the memory size is $\Omega(B \log B)$. Observe that the inorder traversal of the relevant nodes consists of a left-to-right scan of each level of the tree. Since each level is stored contiguously in memory, the bound follows under the assumption above, as the memory can hold one block for each of the lowest $\lceil\log(B+1)\rceil$ levels simultaneously.

3 Search Trees of Small Height

In the previous section, we considered how to lay out a static complete tree in memory. In this section, we describe how the static layouts can be used to store dynamic balanced trees. We first describe an insertions only scheme and later show how this scheme can be extended to handle deletions and to achieve space usage arbitrary close to optimal.

Our approach is to embed a dynamic tree in a static complete tree by maintaining a height bound of $\log n + O(1)$ for the dynamic tree, where n is its current size. It follows that the dynamic tree can be embedded in a complete tree of height $\log n + O(1)$ and size $O(n)$. Whenever n has doubled, we create a new static tree. The following subsections are devoted to tree rebalancing schemes achieving height $\log n + O(1)$.

Our scheme is very similar to the tree balancing scheme of Andersson [2] and to the scheme of Itai et al. [14] for supporting insertions into the middle of a file. Bender et al. [5] used a similar scheme in their cache oblivious search trees, but used it to solve the "packed-memory problem", rather than directly to maintain balance in a tree. Note that the embedding of a dynamic tree in a complete tree implies that we cannot use rebalancing schemes which are based on rotations, or, more generally, schemes allowing subtrees to be moved by just changing the pointer to the root of the subtree, as e.g. is the case in the rebalancing scheme of Fagerberg [10] achieving height $\lceil \log n + o(1) \rceil$.

3.1 Insertions. Let T denote the dynamic binary search tree, and let H be the upper bound on $h(T)$ we want to guarantee, i.e. the height we will use for the complete tree in which T is embedded. For a node v in T, we let $s(v) = 2^{H-d(v)+1} - 1$ denote the size of the subtree rooted at v in the complete tree. We define the *density* of v to be the ratio $\rho(v) = |T_v|/s(v)$, and define a sequence of evenly distributed *density thresholds* $0 < \tau_1 < \tau_2 < \cdots < \tau_H = 1$ by $\tau_i = \tau_1 + (i-1)\Delta$ for $1 \le i \le H$ and $\Delta = (1-\tau_1)/(H-1)$. We maintain the invariant at the root r of T that $\rho(r) \le \tau_1$. This implies the constraint $n/(2^H - 1) \le \tau_1$, i.e. $H \ge \log(n/\tau_1 + 1)$. If for some N the current complete tree should be valid for all $n \le N$, we let $H = \lceil \log(N/\tau_1 + 1) \rceil$. In the following we assume $\tau_1 \ge 1/2$ and $N = O(n)$, such that $H = \log n + O(1)$.

The insertion of a new element into a tree T of $n \le N - 1$ elements proceeds as follows:

1. We locate the position in T of the new node v via a top down search, and create v.
2. If $d(v) = H+1$, we rebalance T as follows. First, we in a bottom-up fashion find the nearest ancestor w of v with $\rho(w) \le \tau_{d(w)}$. This happens at the root at the latest. We need not store the sizes of nodes explicitly, as we can compute $|T_w|$ by a traversal of T_w. Since the ancestors of v are examined bottom-up one by one, we have already computed the size of one child when examining a node, and it suffices to traverse the subtree rooted at the other child in order to compute the total size. After having located w, we rebalance T_w by evenly distributing the elements in T_w as follows. We first create a sorted array of all elements in T_w by an inorder traversal of T_w. The $\lceil |T_w|/2 \rceil$th element becomes the element stored at w, the smallest $\lfloor (|T_w| - 1)/2 \rfloor$ elements are recursively distributed in the left subtree of w and the largest $\lceil (|T_w| - 1)/2 \rceil$ elements are recursively distributed in the right subtree of w.

In the redistribution step, the use of an additional array can be avoided by compacting the elements into the rightmost end of the complete subtree rooted at v by a right-to-left inorder traversal, and then inserting the elements at the positions described above in a left-to-right inorder traversal.

Lemma 3.1. *A redistribution at v implies $\lfloor \rho(v) \cdot s(w) \rfloor - 1 \le |T_w| \le \lceil \rho(v) \cdot s(w) \rceil$ for all descendants w of v.*

Proof. We prove the bounds by induction on the depth of w. The bounds hold for $w = v$, since by definition $|T_v| = \rho(v) \cdot s(v)$. Let u be a descendant of v, let w and w' be the children of u, and assume the bounds hold for u. Since $\rho(v) \le 1$, we have $|T_u| \le \lceil \rho(v) \cdot s(u) \rceil = \lceil \rho(v) \cdot (1 + s(w) + s(w')) \rceil \le 1 + \lceil \rho(v) \cdot s(w) \rceil + \lceil \rho(v) \cdot s(w') \rceil$. From $s(w) = s(w')$ we get $\lceil (|T_u| - 1)/2 \rceil \le \lceil \rho(v) \cdot s(w) \rceil$. The distribution algorithm guarantees that $|T_w| \le \lceil (|T_u| - 1)/2 \rceil$, implying $|T_w| \le \lceil \rho(v) \cdot s(w) \rceil$.

For the lower bound we have $|T_u| \ge \lfloor \rho(v) \cdot s(u) \rfloor - 1 \ge \lfloor \rho(v) \cdot (s(w) + s(w')) \rfloor - 1 \ge (\lfloor \rho(v) \cdot s(w) \rfloor - 1) + (\lfloor \rho(v) \cdot s(w') \rfloor - 1) + 1$. Because $s(w) = s(w')$, we get $\lfloor (|T_u| - 1)/2 \rfloor \ge \lfloor \rho(v) \cdot s(w) \rfloor - 1$. The distribution algorithm guarantees that $|T_w| \ge \lfloor (|T_u| - 1)/2 \rfloor$, implying $|T_w| \ge \lfloor \rho(v) \cdot s(w) \rfloor - 1$. □

Theorem 3.1. *Insertions require amortized $O((\log^2 n)/(1 - \tau_1))$ time and amortized $O(\log_B n + (\log^2 n)/(B(1 - \tau_1)))$ memory transfers.*

Proof. Consider a redistribution at a node v, caused by an insertion below v. By the rebalancing algorithm, we for a child w of v have $|T_w| > \tau_{d(w)} \cdot s(w)$, as the redistribution otherwise would have taken place at w. Immediately after the last time there was a redistribution at v or at an ancestor of v, we by Lemma 3.1 had $|T_w| < \tau_{d(v)} \cdot s(w) + 1$. It follows that the number of insertions below w since the last redistribution at v or an ancestor of v is at least $\tau_{d(w)} \cdot s(w) - (\tau_{d(v)} \cdot s(w) + 1) = \Delta \cdot s(w) - 1$. The redistribution at v takes time $O(s(v))$, which can be covered by charging $O(s(v)/\max\{1, \Delta \cdot s(w) - 1\}) = O(1/\Delta)$ to each of the mentioned insertions below w. Since each created node has at most H ancestors and hence is charged at most H times, the amortized redistribution time for an insertion is $O(H/\Delta) = O(H^2/(1 - \tau_1))$.

Since a top-down search requires $O(\log_B N)$ memory transfers and the redistribution is done solely by in-order traversals requiring $O(\max\{1, s(v)/B\})$ memory transfers, the bound on memory transfers follows. □

Example. Assume that $\tau_1 = 0.9$. This implies that we increase H by one whenever an insertion causes $n > \tau_1(2^H - 1)$. Since increasing H by one doubles the size of the complete tree, this implies that we always have density at least 0.45, i.e. the array used for the layout has size at most $1/0.45n = 2.2n$. Note that the space usage in the worst case is at least $2n$, independently of the choice of τ_1. Since the size of the complete tree doubles each time H is increased, the global rebuilding only increases the amortized update cost by a constant additive term. By Lemma 3.1, all nodes v with depth $H - 2$ in the complete tree, i.e. with $s(v) = 7$, are present in T, since $\lfloor 0.45 \cdot 7 \rfloor - 1 > 0$. The number of memory transfers for range searches is therefore guaranteed to be asymptotically optimal.

3.2 Deletions. One standard approach to add deletions is to simply mark elements as deleted, removing marked nodes by a global rebuilding when, say, half of the elements have been deleted. The disadvantage of this scheme is that locally, elements can end up being sparsely distributed in memory, such that no bound on the number of memory transfers for a range search can be guaranteed.

To support range queries with a worst-case guarantee on the number of memory transfers, the tree T must be rebalanced after deletions. The idea is similar to the scheme used for insertions, except that we now also have lower bound density thresholds $0 \le \gamma_H < \cdots < \gamma_2 < \gamma_1 < \tau_1$, where $\gamma_i = \gamma_1 - (i-1)\Delta'$ for $1 \le i \le H$ and $\Delta' = (\gamma_1 - \gamma_H)/(H - 1)$. For the root r of T we require the invariant $\gamma_1 \le \rho(r) \le \tau_1$.

Deletion is done as described below. Insertions are handled as described in Section 3.1, except that Step 2 is replaced by Step 2 below.

1. First, we locate the node v in T containing the element e to be deleted, via a top down search in T. If v is not a leaf and v has a right subtree, we then locate the node v' containing the immediate successor to e (the node reached by following left children in the right subtree of v), swap the elements at v and v', and let $v = v'$. We repeat this until v is a leaf. If v is not a leaf but v has no right subtree, we symmetrically swap v with the node containing the predecessor of e. Finally, we delete the leaf v from T.
2. We rebalance the tree by rebuilding the subtree rooted at the lowest ancestor w of v satisfying $\gamma_{d(w)} \le \rho(w) \le \tau_{d(w)}$.

THEOREM 3.2. *Insertions and deletions require amortized $O((\log^2 n)/\alpha)$ time and amortized $O(\log_B n + (\log^2 n)/(B\alpha))$ memory transfers, where $\alpha = \min\{\gamma_1 - \gamma_H, 1 - \tau_1\}$.*

Proof. Consider a redistribution at a node v. If the redistribution is caused by an update below a child w of v leading to $|T_w| > \tau_{d(w)} \cdot s(w)$, then the argument is exactly as in Theorem 3.1. Otherwise the redistribution is caused by an update below a child w of v leading to $|T_w| < \gamma_{d(w)} \cdot s(w)$. Immediately after the last time there was a redistribution at v or at an ancestor of v, we by Lemma 3.1 had $|T_w| > \gamma_{d(v)} \cdot s(w) - 2$. It follows that the number of deletions since the last rebuild at v or an ancestor of v is at least $(\gamma_{d(v)} \cdot s(w) - 2) - \gamma_{d(w)} \cdot s(w) = \Delta' \cdot s(w) - 2$. By averaging the redistribution time over the deletions, the amortized redistribution time of a deletion is $O(H/\Delta') = O(H^2/(\gamma_1 - \gamma_H))$. □

Example. Assume $\tau_1 = 0.9$, $\gamma_1 = 0.35$, and $\gamma_H = 0.3$. We increase H by one whenever an insertion causes $n > \tau_1(2^H - 1)$ and decrease H by one whenever a deletion causes $n < \gamma_1(2^H - 1)$. With the parameters above, we have that when H is changed, at least $(\tau_1/2 - \gamma_1)n = 0.1n$ updates must be performed before H is changed again, so the global rebuilding only increases the amortized update cost by a constant additive term. The array used for the layout has size at most $n/\gamma_1 = 2.9n$. By Lemma 3.1, all nodes with depth $H - 2$ (and hence size 7) in the complete tree are present in T, as $\lfloor \gamma_H \cdot 7 \rfloor - 1 > 0$. The number of memory transfers for range searches is therefore asymptotically optimal.

3.3 Improved densities. The rebalancing schemes considered in the previous section require in the worst case space at least $2n$, due to the occasional doubling of the array. In this section, we describe how to achieve space $(1 + \varepsilon)n$, for any $\varepsilon > 0$. As a consequence, we achieve space usage close to optimal and reduce the number of memory transfers for range searches.

Our solution is the following. Let N be the space we are willing to use (not necessarily a power of two), and τ_1 and γ_1 density thresholds such that $\gamma_1 \le n/N \le \tau_1$.

Whenever the density threshold becomes violated, a new N must be chosen. If $N = 2^k - 1$ for some k, then we can apply the previous schemes directly. Otherwise, assume $N = 2^{b_1} + 2^{b_2} + \cdots 2^{b_k}$, where $b_1, \ldots, b_k$ are non-negative integers satisfying $b_i > b_{i+1}$, i.e. the b_i values are the positions of 1s in the binary representation of N. For each b_i, we will have a tree F_i consisting of a root r_i with no left child and a right subtree C_i which is a complete tree of size $2^{b_i} - 1$. The elements will be distributed among $F_1, \ldots, F_k$ such that all elements stored in F_i are smaller than the elements in F_{i+1}. If F_i stores at least one element, the minimum element in F_i is stored at r_i and the remaining elements are stored in a tree T_i which is embedded in C_i. The trees are laid out in memory in the order $r_1, r_2, \ldots, r_k, C_1, C_2, \ldots, C_k$, where each C_i is laid out using the van Emde Boas layout.

A search for an element e proceeds by examining the elements at $r_1, \ldots, r_k$ in increasing order until e is found or the subtree T_i is located that must contain e, i.e. e is larger than the element at r_i and smaller than the element at r_{i+1}. In the latter case, we perform a top-down search on T_i. The total time for a search is $O(i + b_i) = O(\log N)$ using $O(i/B + \log_B(2^{b_i} - 1)) = O(\log_B N)$ I/Os.

For the rebalancing, we view $F_1, \ldots, F_k$ as being merged into one big tree F, where all leafs have the same depth and all internal nodes are binary, except for the nodes on the rightmost path which may have degree three. The tree C_{i+1} is considered a child of the rightmost node u_i in C_i with $h(u_i) = b_{i+1} + 1$, and with the element of r_{i+1} being a second element of u_i. Note that the elements of F satisfy inorder. For a node v in F, we define $s(v)$ to be the subtree T_v of F plus the number of nodes of degree three, i.e. the number of slots to store elements in T_v, and $|T_v|$ the number of elements stored in T_v. As in Section 3.1 and 3.2, we define $\rho(v) = |T_v|/s(v)$. The rebalancing is done as in Sections 3.1 and 3.2, except that if we have to redistribute the content of v, we will explicitly ensure that the inequality $\lfloor \rho(v) \cdot s(w) \rfloor - 1 \le |T_w| \le \lceil \rho(v) \cdot s(w) \rceil$ from Lemma 3.1 is satisfied for all descendants w of v. That this is possible follows from the inequalities below, where u is a descendant of v and $w_1, \ldots, w_k$ are the children of u for $k = 2$ or 3:

$$\begin{aligned}\lceil \rho(v) \cdot s(u) \rceil &= \left\lceil \rho(v) \cdot \left(k - 1 + \textstyle\sum_{i=1}^{k} s(w_i)\right) \right\rceil \\ &\le k - 1 + \textstyle\sum_{i=1}^{k} \lceil \rho(v) \cdot s(w_i) \rceil \ , \\ \lfloor \rho(v) \cdot s(u) \rfloor - 1 &\ge \left\lfloor \rho(v) \cdot \textstyle\sum_{i=1}^{k} s(w_i) \right\rfloor - 1 \\ &\ge k - 1 + \textstyle\sum_{i=1}^{k} (\lfloor \rho(v) \cdot s(w_i) \rfloor - 1) \ .\end{aligned}$$

Because Lemma 3.1 still holds, Theorem 3.2 also holds. The only change in the analysis of Theorem 3.2 is that for a node v on the rightmost path with a child w, we now have $s(v) \le 4s(w)$, i.e. the bound on the amortized time and number of memory transfers increases by a factor two.

Example. Let $\varepsilon > 0$ be an arbitrary small constant such that when N is chosen, $N = (1 + \varepsilon)n$. Valid density thresholds can then be $\tau_1 = (\delta + 1)/2$, $\gamma_1 = (3\delta - 1)/2$, and $\gamma_H = 2\delta - 1$, where $\delta = 1/(1 + \varepsilon)$ is the density immediately after having chosen N. After choosing an N, at least $N(1 - \delta)/2 = O(N/\varepsilon)$ updates must be performed before a new N is chosen. Hence, the amortized cost of the global rebuildings is $O(1/\varepsilon)$ time and $O(1/(\varepsilon B))$ memory transfers per update. The worst case space usage is $n/\gamma_1 = n(1 + \varepsilon)/(1 - \varepsilon/2) = n(1 + O(\varepsilon))$.

4 Experiments

In this section, we describe our empirical investigations of methods for laying out a search tree in memory.

We implemented the four implicit memory layouts discussed in Section 2: DFS, inorder, BFS, and van Emde Boas. We also implemented a cache aware implicit layout based on a d-ary version of the BFS, where d is chosen such that the size of a node equals a cache line. Our experiments thus compare layouts which in term of optimization for the memory hierarchy cover three categories: not optimized, cache oblivious, and cache aware.

We also implemented pointer based versions of the layouts, where each node stored in the array contains the indices of its children. Compared to implicit layouts, pointer based layouts have lower instruction count for navigation, higher total memory usage, and lower number of nodes per memory block. We implemented one further pointer based layout, namely the layout which arises when building a binary tree by random insertions, placing nodes in the array in order of allocation. We call this the *random insertion* layout.

Our experiments fall in two parts: one dealing with searches in static layouts, and one dealing with the dynamization method from Section 3.1. In Section 4.2, we report on the results. We tested several combinations and variations of the memory layouts and algorithms, but for lack of space, we only describe a subset representative of our general observations.

4.1 Methodology. The computer used to perform the experiments had two 1 GHz Pentium III (Coppermine) processors, 256 KB of cache, and 1 GB of RAM. The programs were written in C, compiled by the GNU `gcc` compiler version 2.95.2.1 with full optimization (option -O3). The operating system was Linux with kernel version 2.4.3-12smp.

The timing was based on wall clock time. For the search based experiments, we used the `getitimer` and `setitimer` system calls to interrupt the program every 10 seconds, giving us a relative timing precision of roughly 0.001 for most experiments.

The elements were 32 bit in size, as was each of the two pointers per node used in the pointer based layouts. We only report on integer keys—our results with floating point keys did differ (probably due in parts to the different costs of comparisons), but not significantly. We generated uniformly random integers by casting double precision floats returned by `drand48()`. We only searched for present keys.

Where possible, the programs shared source code, in order to minimize coding inconsistencies. We also tried to avoid artifacts from the compilation process by e.g. inlining function calls ourselves.

We performed experiments for $n = 2^k, 2^k - 1, 2^k + 1$, and $0.7 * 2^k$ for a range of k. For n not a power of two, the assumption from Section 2 of dealing with complete trees is not fulfilled. We adapted to this situation by cutting the tree at the boundary of the array: If the address of both children of node v is outside the array, i.e. larger than n, then v is a leaf, if only the right child is outside, it is a degree one node. This works because the addresses of children are higher than that of their parent (which does not hold for the inorder layout, but there, we simply used binary search).

Due to the small difference between the 1 GB RAM size and 2 GB address space, experiments beyond main memory required a different setup. This we achieved by booting the machine such that only 32 MB of RAM was available. However, the bulk of our experiments covered trees contained in cache and RAM.

The source code of the programs, our scripts and tools, and the data we present here are available online under `ftp://ftp.brics.dk/RS/01/36/Experiments/`.

4.2 Results. For all graphs, the y-axis is logarithmic, and depicts the average time for one search for (or insertion of) a randomly chosen key, measured in seconds. All the x-axes depicts $\log_2 n$, where n is the number of keys stored in the search tree. Note that this translates to different memory usage for implicit and pointer based layouts.

Figure 3 compares the time for random searches in pointer based layouts. Pointer based layouts all have the same instruction count per level during a search. This is reflected in the range $n = 2^{10}, \ldots, 2^{14}$ (for which the tree fits entirely in cache), where the three layouts of optimal height behave identically, while the random insertion layout (which has larger average height) is worse. As n gets bigger, the differences in memory access pattern starts showing. For random searches, we can expect the top levels of the trees to reside in cache. For the remaining levels, a cache fault should happen at every level for the BFS layout, approximately at every second level for the DFS layout (most nodes reside in the same cache line as their left child), and every $\Theta(\log_B n)$ levels for the van Emde Boas layout. This analysis is consistent with the graphs.

Figure 4 compares the time for random searches in implicit layouts. For sizes up to cache size ($n = 2^{16}$), it appears that the higher instruction count for navigating in an implicit layout dominates the running times: most graphs are slightly higher than corresponding graphs in Figure 3, and the van Emde Boas layout (most complicated address arithmetic) is the slowest while the BFS layout (simplest address arithmetic) is fastest. For larger n, the memory access pattern shows its effect. The high arity layouts ($d = 8$ and 16) are the fastest, as expected—they are cache-optimized and have simple address arithmetic. The van Emde Boas layout is quite competitive, eventually beating BFS and only being 50% slower than the cache aware layouts.

The inorder layout has bad performance, probably because no nodes in the top part of the tree share cache lines. It is worst when n is a power of two. We believe this as an effect of the limited associativity of the cache: For these n, the nodes of the top of the tree are large powers of two apart in memory, and are mapped to the same few lines in cache.

In Figure 5, we compare the search times for the pointer based and the implicit versions of the BFS and the van Emde Boas layout. The aim is to see how the effect of a smaller size and a more expensive navigation compete against each other. For the BFS, the implicit version wins for all sizes, indicating that its address arithmetic is not slower than following pointers. This is not the case for the van Emde Boas layout—however, outside of cache, the implicit version wins, most likely due to the higher value of B resulting from the absence of pointers.

In Figure 6, we compare the performance of the dynamic versions of some of the data structures. The inorder and the van Emde Boas layout is made semi-dynamic by the method from Section 3.1. For the inorder layout, the redistribution during rebalancing can be implemented particularly simple, just by scans of contiguous segments of the array. We use this implementation here. The random insertion layout is semi-dynamic by definition.

Starting with a bulk of 10,000 randomly chosen elements, we insert bulks of sizes increasing by a factor

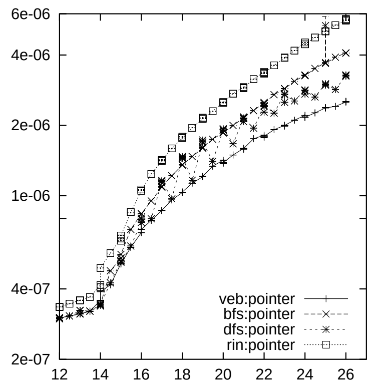

Figure 3: Searches for pointer based layouts

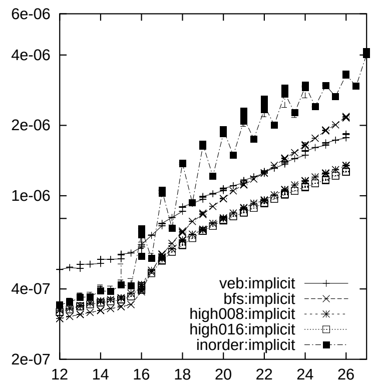

Figure 4: Searches for implicit layouts

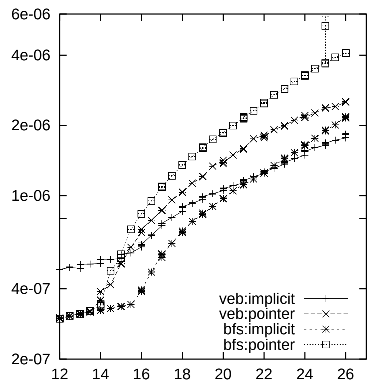

Figure 5: Search time for pointer based and implicit BFS and van Emde Boas layouts

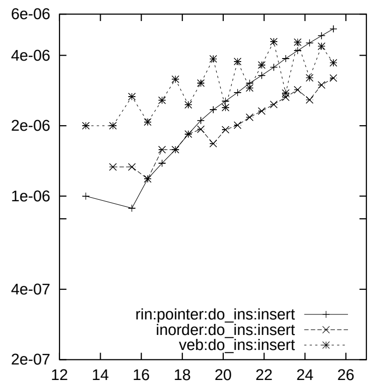

Figure 6: Insert time per element

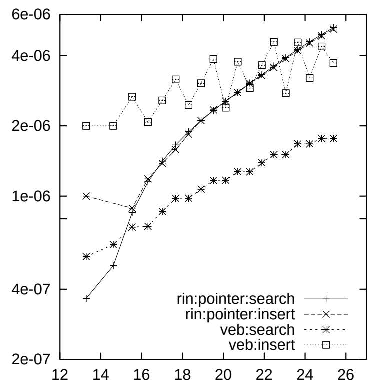

Figure 7: Insert and Search for implicit veb and unbalanced search trees

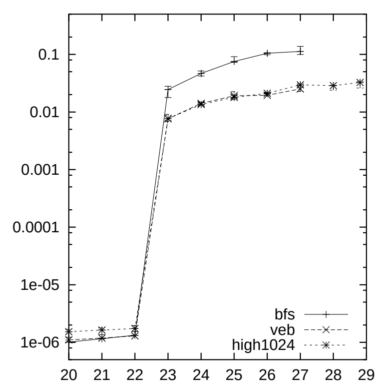

Figure 8: Beyond main memory

of 1.5. We time the insertion of one block and calculate the average time for inserting one element. The amortization in the bounds of the method from Section 3.1 is apparent in the instability of the graphs. In contrast, the unbalanced pointer based search tree has a relatively smooth graph. We remark that the dynamization method of Section 3.1 seems quite competitive, eventually winning over the unbalanced pointer based tree, which for random insertions is known to compete well against standard rebalancing schemes for binary search trees, such as red-black trees (see e.g. [17, p. 127]). The inorder layout is somewhat faster than the van Emde Boas layout, which we think is due to the simpler redistribution algorithm.

In Figure 7, we compare in more detail the performance of the random insertion layout with the implicit, semi-dynamic van Emde Boas layout, showing the time for random insertions as well as for random searches. If the data structure is to be used mainly for searches and only occasionally for updates, the cache oblivious version is preferable already at roughly 2^{16} elements. But even if updates dominate, it becomes advantageous around 2^{23} elements.

In Figure 8, we look at the performance of the layouts as our memory requirement exceeds main memory. As said, for this experiment we booted the machine in such a way that only 32 MB of RAM was available. We compare the van Emde Boas layout, the usual BFS layout, and a 1024-ary version version of it, optimized for the page size of the virtual memory. The keys of a 1024-ary nodes are stored in sorted order, and a node is searched by a fixed, inlined decision tree. We measure the time for random searches on a static tree.

Inside main memory, the BFS is best, but looses by a factor of five outside. The tree optimized for page size is the best outside main memory, but looses by a factor of two inside. Remarkably, the van Emde Boas layout is on par with the best throughout the range.

4.3 Conclusion. From the experiments reported in this paper, it is apparent that the effects of the memory hierarchy in todays computers play a dominant role for the running time of tree search algorithms, already for sizes of trees well within main memory.

It also appears that in the area of search trees, the nice theoretical properties of cache obliviousness seems to carry over into practice: in our experiments, the van Emde Boas layout was competitive with cache aware structures, was better than structures not optimized for memory access for all but the smallest n, and behaved robustly over several levels of the memory hierarchy.

One further observation is that the effects from the space saving and increase in fanout caused by implicit layouts are notable.

Finally, the method for dynamic cache oblivious search tree suggested in this paper seems practical, not only in terms of implementation effort but also in terms of running time.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[2] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 1990.

[3] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. 37th Ann. Symp. on Foundations of Computer Science*, pages 560–569. IEEE Computer Society Press, 1996.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Ann. Symp. on Foundations of Computer Science*, pages 399–409. IEEE Computer Society Press, 2000.

[6] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, 2002.

[7] S. Carlsson, P. V. Poblete, and J. I. Munro. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 1988.

[8] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 444–453. ACM Press, 1999.

[9] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In J. R. Gilbert and R. G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer, 1990.

[10] R. Fagerberg. The complexity of rebalancing a binary search tree. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 19, 1999.

[11] A. Fiat, J. I. Munro, M. Naor, A. A. Schäffer, J. P. Schmidt, and A. Siegel. An implicit data structure for searching a multikey table in logarithmic time. *Journal of Computer and System Sciences*, 43(3):406–424, 1991.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[14] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming, 8th Colloquium*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 1981.

[15] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[16] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithms*, 1:4, 1996.

[17] K. Mehlhorn and S. Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[18] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.

[19] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[20] N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *WAE 2001, 5th Int. Workshop on Algorithm Engineering*, volume 2141 of *LNCS*, pages 67–78. Springer, 2001.

[21] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6:80–82, 1977.

[22] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[23] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.