Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton.
Cache-Oblivious B-Trees.
SIAM J. Computing, volume 35, number 2, 2005.

The B-tree is designed for a two-level hierarchy, and the situation becomes more complex with more than two levels. We need a multilevel structure, with one level per transfer block size. Suppose $B_1 > B_2 > \cdots > B_L$ are the block sizes between the $L+1$ levels of memory. At the top level we have a B_1-tree; each node of this B_1-tree is a B_2-tree; etc. Even when it is possible to determine all these parameters, such a data structure is cumbersome. Also, each level of recursion incurs a constant-factor wastage in storage, in order to amortize dynamic changes, leading to suboptimal memory-transfer performance for $L = \omega(1)$.

1.3. Results. We develop two cache-oblivious search trees. These results are the first demonstration that even irregular and dynamic problems, such as data structures, can be solved efficiently in the cache-oblivious model. Since the conference version [18] of this paper appeared, many other data-structural problems have been addressed in the cache-oblivious model; see Table 1.1.

Our results achieve the memory-transfer bounds listed below. The parameter N denotes the number of elements stored in the tree. *Updates* refer to both key insertions and deletions.

1. The first cache-oblivious search tree attains the following memory-transfer bounds:
   Search: $O(1+\log_{B+1} N)$, which is optimal and matches the search bound of B-trees.
   Update: $O(1+\log_{B+1} N)$ amortized, which matches the update bound of B-trees, though the B-tree bound is worst case.
2. The second cache-oblivious search tree adds the *scan* operation (also called the *range search* operation). Given a key x and a positive integer S, the scan operation accesses S elements in key order, starting after x. The memory-transfer bounds are as follows:
   Search: $O(1+\log_{B+1} N)$.
   Scan: $O(1+S/B)$, which is optimal.
   Update: $O(1+\log_{B+1} N+\frac{\log^2 N}{B})$ amortized, which matches the B-tree update bound of $O(1+\log_{B+1} N)$ when $B=\Omega(\log N \log\log N)$.
   This last relation between B and N usually holds in external memory but often does not hold in internal memory.

In the development of these data structures, we build and identify tools for cache-oblivious manipulation of data. These tools have since been used in many of the cache-oblivious data structures listed in Table 1.1. In Section 2.1, we show how to linearize a tree according to what we call the *van Emde Boas layout*, along the lines of Prokop's static search tree [40]. In Section 2.2, we describe a type of *strongly weight-balanced search tree* [11] useful for maintaining locality of reference. Following the work of Itai, Konheim, and Rodeh [33] and Willard [52, 53, 54], we develop a *packed-memory array* for maintaining an ordered collection of N items in an array of size $O(N)$ subject to insertions and deletions in $O(1+\frac{\log^2 N}{B})$ amortized memory transfers; see Section 2.3. This structure can be thought of as a cache-oblivious *linked list* that supports scanning S consecutive elements in $O(1+S/B)$ memory transfers (instead of the naïve $O(S)$) and updates in $O(1+\frac{\log^2 N}{B})$ amortized memory transfers.

1.4. Notation. We define the *hyperfloor* of x, denoted $\lfloor\!\lfloor x \rfloor\!\rfloor$, to be $2^{\lfloor \log x \rfloor}$, i.e., the largest power of 2 smaller than x.[3] Thus, $x/2 < \lfloor\!\lfloor x \rfloor\!\rfloor \le x$. Similarly, the

[3] All logarithms are base 2 if not otherwise specified.

B-tree	• Simplification via packed-memory structure/low-height trees	[20, 25]
	• Simplification and persistence via exponential structures	[42, 17]
	• Implicit	[29, 30]
Static search trees	• Basic layout	[40]
	• Experiments	[35]
	• Optimal constant factor	[14]
Linked lists supporting scans		[15]
Priority queues		[8, 23, 26]
Trie layout		[6, 19]
Computational geometry	• Distribution sweeping	[22]
	• Voronoi diagrams	[34]
	• Orthogonal range searching	[1, 9]
	• Rectangle stabbing	[10]
Lower bounds		[24]

TABLE 1.1

Related work in cache-oblivious data structures. These results, except the static search tree of [40]*, appeared after the conference version of this paper.*

hyperceiling $\lceil\!\lceil x \rceil\!\rceil$ is defined to be $2^{\lceil \log x \rceil}$. Analogously, we define *hyperhyperfloor* and *hyperhyperceiling* by $\lfloor\!\lfloor\!\lfloor x \rfloor\!\rfloor\!\rfloor = 2^{\lfloor\!\lfloor \log x \rfloor\!\rfloor}$ and $\lceil\!\lceil\!\lceil x \rceil\!\rceil\!\rceil = 2^{\lceil\!\lceil \log x \rceil\!\rceil}$. These operators satisfy $\sqrt{x} < \lfloor\!\lfloor\!\lfloor x \rfloor\!\rfloor\!\rfloor \leq x$ and $x \leq \lceil\!\lceil\!\lceil x \rceil\!\rceil\!\rceil < x^2$.

2. Tools for Cache-Oblivious Data Structures.

2.1. Static Layout and Searches. We first present a cache-oblivious *static* search-tree structure, which is the starting point for the dynamic structures. Consider a $O(\log N)$-height search tree in which every node has at least two and at most a constant number of children and in which all leaves are on the same level. We describe a mapping from the nodes of the tree to positions in memory. The cost of any search in this layout is $\Theta(1 + \log_{B+1} N)$ memory transfers, which is optimal up to constant factors. Our layout is a modified version of Prokop's layout for a complete binary tree whose height is a power of 2 [40, pp. 61–62]. We call the layout the *van Emde Boas layout* because it resembles the van Emde Boas data structure [47, 48].[4]

The van Emde Boas layout proceeds recursively. Let h be the height of the tree, or more precisely, the number of levels of nodes in the tree. Suppose first that h is a power of 2. Conceptually split the tree at the middle level of edges, between nodes of height $h/2$ and $h/2 + 1$. This breaks the tree into the *top recursive subtree* A of height $h/2$, and several *bottom recursive subtrees* $B_1, B_2, \ldots, B_\ell$, each of height $h/2$. If all nonleaf nodes have the same number of children, then the recursive subtrees all have size roughly $\sqrt{N}$, and ℓ is roughly $\sqrt{N}$. The layout of the tree is obtained by recursively laying out each subtree and combining these layouts in the order A, B_1, $B_2, \ldots, B_\ell$; see Figure 2.1.

If h is not a power of 2, we assign a number of levels that is a power of 2 to the bottom recursive subtrees and assign the remaining levels to the top recursive subtree. More precisely, the bottom subtrees have height $\lceil\!\lceil h/2 \rceil\!\rceil$ $(= \lfloor\!\lfloor h - 1 \rfloor\!\rfloor)$ and

[4]We do not use a van Emde Boas tree—we use a normal tree with pointers from each node to its parent and children—but the order of the nodes in memory is reminiscent of van Emde Boas trees.

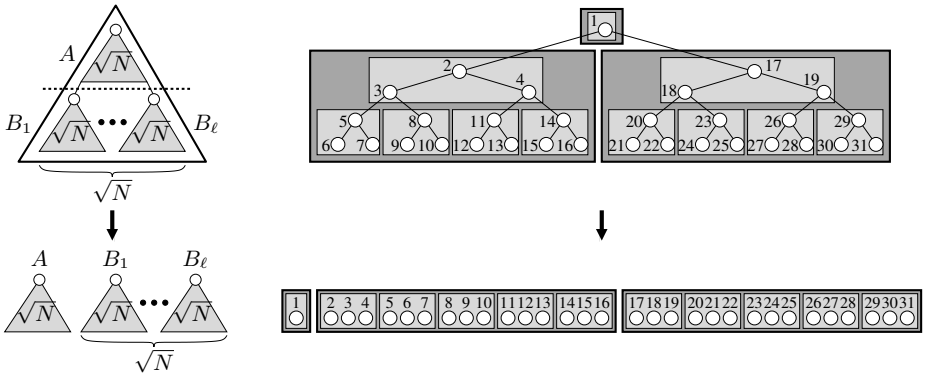

FIG. 2.1. *The van Emde Boas layout. Left: in general; right: of a tree of height* 5.

the top subtree has height $h - \lceil h/2 \rceil$. This rounding scheme is important for later dynamic structures because the heights of the cut lines in the lower trees do not vary with N. In contrast, this property is not shared by the simple rounding scheme of assigning $\lfloor h/2 \rfloor$ levels to the top recursive subtree and $\lceil h/2 \rceil$ levels to the bottom recursive subtrees.

The memory-transfer analysis views the van Emde Boas layout at a particular *level of detail*. Each level of detail is a partition of the tree into disjoint recursive subtrees. In the finest level of detail, 0, each node forms its own recursive subtree. In the coarsest level of detail, $\lceil \log_2 h \rceil$, the entire tree forms the unique recursive subtree. Level of detail k is derived by starting with the entire tree, recursively partitioning it as described above, and exiting a branch of the recursion upon reaching a recursive subtree of height $\leq 2^k$. The key property of the van Emde Boas layout is that, at any level of detail, each recursive subtree is stored in a contiguous block of memory.

One useful consequence of our rounding scheme is the following.

LEMMA 2.1. *At level of detail k all recursive subtrees except the one containing the root have the same height of 2^k. The recursive subtree containing the root has height between* 1 *and 2^k inclusive.*

Proof. The proof follows from a simple induction on the level of detail. Consider a tree T of height h. At the coarsest level of detail, $\lceil \log_2 h \rceil$, there is a single recursive subtree, which includes the root. In this case the lemma is trivial. Suppose by induction that the lemma holds for level of detail k. In this level of detail the recursive subtree containing the root of T has height h', where $1 \leq h' \leq 2^k$, and all other recursive subtrees have height 2^k. To progress to the next finer level of detail, $k-1$, all recursive subtrees that do not contain the root are recursively split once more so that they have height 2^{k-1}. If the height h' of the top recursive subtree is at most 2^{k-1}, then it is not split in level of detail $k-1$. Otherwise, the root is split into bottom recursive subtrees of height 2^{k-1} and a top recursive subtree of height $h'' \leq 2^{k-1}$. The inductive step follows. □

LEMMA 2.2. *Consider an N-node search tree T that is stored in a van Emde Boas layout. Suppose that each node in T has between $\delta \geq 2$ and $\Delta = O(1)$ children. Let h be the height of T. Then a search in T uses at most $4\left\lceil \log_\delta \Delta \log_{B+1} N + \log_{B+1} \Delta \right\rceil = O(1 + \log_{B+1} N)$ memory transfers.*

Proof. Let k be the coarsest level of detail such that every recursive subtree

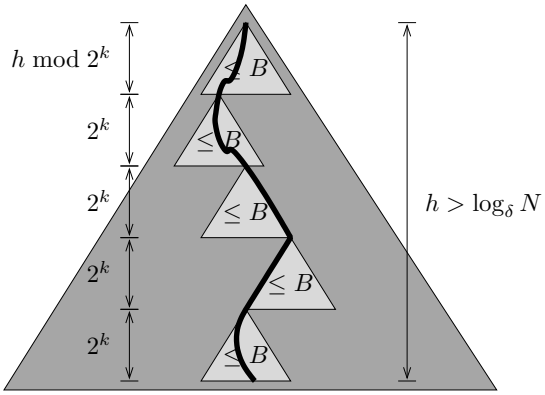

FIG. 2.2. *The recursive subtrees visited by a root-to-leaf search path in level of detail k.*

contains at most B nodes; see Figure 2.2. Thus, every recursive subtree is stored in at most two memory blocks. Because tree T has height h, $\lceil h/2^k \rceil$ recursive subtrees are traversed in each search, and thus at most $2\lceil h/2^k \rceil$ memory blocks are transferred. Because the tree has height h, $\delta^{h-1} < N < \Delta^h$, that is, $\log_\Delta N < h < \log_\delta N + 1$. Because a tree of height 2^{k+1} has more than B nodes, $\Delta^{2^{k+1}} > B$ so $\Delta^{2^{k+1}} \geq B + 1$. Thus, $2^k \geq \frac{1}{2}\log_\Delta(B+1)$. Therefore, the maximum number of memory transfers is

$$2\left\lceil \frac{h}{2^k} \right\rceil \leq 4\left\lceil \frac{1+\log_\delta N}{\log_\Delta(B+1)} \right\rceil = 4\left\lceil \left(1+\frac{\log N}{\log \delta}\right)\left(\frac{\log \Delta}{\log(B+1)}\right) \right\rceil$$
$$= 4\left\lceil \log_\delta \Delta \log_{B+1} N + \log_{B+1} \Delta \right\rceil .$$

Because δ and Δ are constants, this bound is $O(1 + \log_{B+1} N)$. □

2.2. Strongly Weight-Balanced Search Trees. To convert the static layout into a dynamic layout, we use a dynamic balanced search tree. We require the following two properties of the balanced search tree.

PROPERTY 1 (descendant amortization). *Suppose that whenever we* rebalance *a node v (i.e., modify it to keep balance) we also* touch *all of v's descendants. Then the amortized number of elements touched per insertion is $O(\log N)$.*

PROPERTY 2 (strong weight balance). *For some constant d, every node v at height h has $\Theta(d^h)$ descendants.*

Property 1 is normally implied by Property 2 as well as by a weaker property called weight balance. A tree is *weight balanced* if, for every node v, its left subtree (including v) and its right subtree (including v) have sizes that differ by at most a constant factor. Weight balancedness guarantees a relative bound between subtrees with a common root, so the size difference between subtrees of the same height may be large. In contrast, strong weight balance requires an absolute constraint that relates the sizes of all subtrees at the same level. For example, BB[α] trees [38] are weight-balanced binary search trees based on rotations, but they are not strongly weight balanced.

Search trees that satisfy Properties 1 and 2 include weight-balanced B-trees [11], deterministic skip lists [37], and skip lists [41] in the expected sense. We choose to use weight-balanced B-trees defined as follows.

DEFINITION 2.3 (weight-balanced B-tree [11]). *A rooted tree T is a* weight-