Engineering a Sorted List Data Structure for 32 Bit Keys*

Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders
MPI Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany,
[dementiev,kettner,sanders,snej]@mpi-sb.mpg.de.

Abstract

Search tree data structures like van Emde Boas (vEB) trees are a theoretically attractive alternative to comparison based search trees because they have better asymptotic performance for small integer keys and large inputs. This paper studies their practicability using 32 bit keys as an example. While direct implementations of vEB-trees cannot compete with good implementations of comparison based data structures, our tuned data structure significantly outperforms comparison based implementations for searching and shows at least comparable performance for insertion and deletion.

1 Introduction

Sorted lists with an auxiliary data structure that supports fast searching, insertion, and deletion are one of the most versatile data structures. In current algorithm libraries [9, 2], they are implemented using comparison based data structures such as ab-trees, red-black trees, splay trees, or skip lists (e.g. [9]). These implementations support insertion, deletion, and search in time $\mathcal{O}(\log n)$ and range queries in time $\mathcal{O}(k + \log n)$ where n is the number of elements and k is the size of the output. For w bit integer keys, a theoretically attractive alternative are van Emde Boas stratified trees (vEB-trees) that replace the $\log n$ by a $\log w$ [12, 10]: A vEB tree T for storing subsets M of $w = 2^{k+1}$ bit integers stores the set directly if $|M| = 1$. Otherwise it contains a root (hash) table r such that $r[i]$ points to a vEB tree T_i for 2^k bit integers. T_i represents the set $M_i = \{x \bmod 2^{2^k} : x \in M \wedge x \gg 2^k = i\}$.[1] Furthermore, T stores $\min M$, $\max M$, and a *top data structure* t consisting of a 2^k bit vEB tree storing the set $M_t = \left\{x \gg 2^k : x \in M\right\}$. This data structure takes space $\mathcal{O}(|M| \log w)$ and can be modified to consume only linear space. It can also be combined with a doubly linked sorted list to support fast successor and predecessor queries.

However, we are only aware of a single implementation study [13] where the conclusion is that vEB-trees are of mainly theoretical interest. In fact, our experiments show that they are *slower* than comparison based implementations even for 32 bit keys.

*Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[1]We use the C-like shift operator '$\gg$', i.e., $x \gg i = \lfloor x/2^i \rfloor$.

In this paper we address the question whether implementations that exploit integer keys can be a practical alternative to comparison based implementations. In Section 2, we develop a highly tuned data structure for large sorted lists with 32 bit keys. The starting point were vEB search trees as described in [10] but we arrive at a nonrecursive data structure: We get a three level search tree. The root is represented by an *array* of size 2^{16} and the lower levels use hash tables of size up to 256. Locating entries in these tables is achieved using hierarchies of bit patterns similar to the integer priority queue described in [1].

Experiments described in Section 3 indicate that this data structure is significantly faster in searching elements than comparison based implementations. For insertion and deletion the two alternatives have comparable speed. Section 4 discusses additional issues.

More Related Work

There are studies on exploiting integer keys in data structures supporting a more restricted set of operations. In particular, sorting has been studied extensively. Other variants are priority queues (e.g. [1]), or data structures supporting fast search in static data [6]. Furthermore, dictionaries can be implemented very efficiently using hash tables.

However, none of these data structures is applicable if we have to maintain a sorted list dynamically. Simple examples are sweep-line algorithms [3] for orthogonal objects,[2] best first heuristics (e.g., [7]), or finding free slots in a list of occupied intervals (e.g. [4]).

2 The Data Structure

We now describe a data structure `Stree` that stores an ordered set of elements M with 32-bit integer keys supporting the main operations element insertion, element deletion, and `locate`(y). Locate returns $\min(x \in M : y \leq x)$.

We use the following notation: For an integer x, $x[i]$ represents the i-th bit, i.e., $x = \sum_{i=0}^{31} 2^i x[i]$. $x[i..j]$, $i \leq j+1$, denotes bits i through j in a binary representation of $x = x[0..31]$, i.e., $x[i..j] = \sum_{k=i}^{j} 2^{k-i} x[i]$. Note that $x[i..i-1] = 0$ represents the empty bit string. The function `msbPos`(z) returns the position of the most significant nonzero bit in z, i.e., `msbPos`$(z) = \lfloor \log_2 z \rfloor = \max\{i : x[i] \neq 0\}$.[3]

Our `Stree` stores elements in a doubly linked sorted *element list* and additionally builds a *stratified tree* data structure that serves as an index for fast access to the elements of the list. If `locate` actually returns a pointer to the element list, additional operations like successor, predecessor, or range queries can also be efficiently implemented. The index data structure consists of the following ingredients arranged in three levels, root, Level 2 (L2), and Level 3 (L3):

[2] General line segments are a nice example where a comparison based data structure is needed (at least for the Bentley-Ottmann algorithm) — the actual coordinates of the search tree entries change as the sweep line progresses but the relative order changes only slowly.

[3] `msbPos` can be implemented in constant time by converting the number to floating point and then inspecting the exponent. In our implementation, two 16-bit table lookups turn out to be somewhat faster.

The root-table r contains a plain array with one entry for each possible value of the 16 most significant bits of the keys. $r[i] = \texttt{null}$ if there is no $x \in M$ with $x[16..31] = i$. If $|M_i| = 1$, it contains a pointer to the element list item corresponding to the unique element of M_i. Otherwise, $r[i]$ points to an L2-table containing $M_i = \{x \in M : x[16..31] = i\}$. The two latter cases can be distinguished using a flag stored in the least significant bit of the pointer.[4]

An L2-table r_i stores the elements in M_i. If $|M_i| \geq 2$ it uses a hash table storing an entry with key j if $\exists x \in M_i : x[8..15] = j$. Let $M_{ij} = \{x \in M : x[8..15] = j, x[16..31] = i\}$. If $|M_{ij}| = 1$ the hash table entry points to the element list and if $|M_{ij}| \geq 2$ it points to an L3-table representing M_{ij} using a similar trick as in the root-table.

An L3-table r_{ij} stores the elements in M_{ij}. If $|M_{ij}| \geq 2$, it uses a hash table storing an entry with key k if $\exists x \in M_{ij} : x[0..7] = k$. This entry points to an item in the element list storing the element with $x[0..7] = k, x[8..15] = j, x[16..31] = i$.

Minima and Maxima: For the root and each L2-table and L3-table, we store the smallest and largest element of the corresponding subset of M. We store both the key of the element and a pointer to the element list.

The root-top data structure t consists of three bit-arrays $t^1[0..2^{16} - 1]$, $t^2[0..4095]$, and $t^3[0..63]$. We have $t^1[i] = 1$ iff $M_i \neq \emptyset$. $t^2[j]$ is the logical-or of $t^1[32j]..t^1[32j+31]$, i.e., $t^2[j] = 1$ iff $\exists i \in \{32j..32j + 31\} : M_i \neq \emptyset$. Similarly, $t^3[k]$ is the logical-or of $t^2[32k]..t^2[32k + 31]$ so that $t^3[k] = 1$ iff $\exists i \in \{1024k..1024k + 1023\} : M_i \neq \emptyset$.

The L2-top data structures t_i consists of two bit arrays $t_i^1[0..255]$ and $t_i^2[0..7]$ similar to the bit arrays of the root-top data structure. The 256 bit table t_i^1 contains a 1-bit for each nonempty entry of r_i and the eight bits in t_i^2 contain the logical-or of 32 bits in t_i^1. This data structure is only allocated if $|M_i| \geq 2$.

The L3-top data structures t_{ij} with bit arrays $t_{ij}^1[0..255]$ and $t_{ij}^2[0..7]$ reflect the entries of M_{ij} in a fashion analogous to the L2-top data structure.

Hash Tables use open addressing with linear probing [8, Section 6.4]. The table size is always a power of two between 4 and 256. The size is doubled when a table of size k contains more than $3k/4$ entries and $k < 256$. The table shrinks when it contains less than $k/4$ entries. Since all keys are between 0 and 255, we can afford to implement the hash function as a full lookup table h that is shared between all tables. This lookup table is initialized to a random permutation $h : 0..255 \to 0..255$. Hash function values for a table of size $256/2^i$ are obtained by shifting $h[x]$ i bits to the right. Note that for tables of size 256 we obtain a perfect hash function, i.e., there are no collisions between different table entries.

Figure 1 gives an example summarizing the data structure.

[4]This is portable without further measures because all modern systems use addresses that are multiples of four (except for strings).

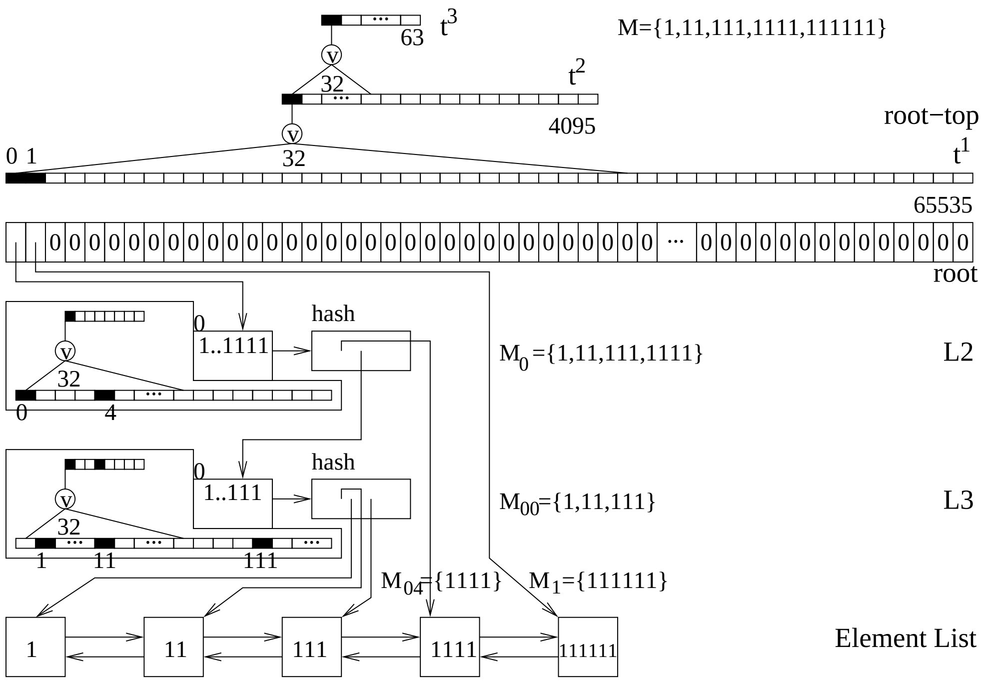

Figure 1: The `Stree`-data structure for $M = \{1, 11, 111, 1111, 111111\}$ (decimal).

2.1 Operations

With the data structure in place, the operations are simple in principle although some case distinctions are needed. To give an example, Figure 2 contains high level pseudo code for locate(y) that finds the smallest $x \in M$ with $y \leq x$. `locate`(y) first uses the 16 most significant bits of y, say $i = y[16..31]$ to find a pointer to M_i in the root table. If M_i is empty ($r[i] =$ `null`), or if the precomputed maximum of M_i is smaller than y, `locate` looks for the next nonzero bit i' in the root-top data structure and returns the smallest element of $M_{i'}$. Otherwise, the next element must be in M_i. Now, $j = y[8..15]$ serves as the key into the hash table r_j stored with M_i and the pattern from level one repeats on level two and possibly on level 3. `locate` in a hierarchy of bit patterns walks up the hierarchy until a "nearby" nonzero bit position is found and then goes down the hierarchy to find the exact position.

We now outline the implementation of the remaining operations. A detailed source code is available at `http://www.mpi-sb.mpg.de/~kettner/proj/veb/`.

find(x) descends the tree until the list item corresponding to x is found. If $x \notin M$ a null pointer is returned. No access to the top data structures is needed.

`insert`(x) proceeds similar to `locate`(x) except that it modifies the data structures it traverses: Minima and maxima are updated and the appropriate bits in the top data structure are set. At the end, a pointer to the element list item of x's successor is available so that x can be inserted in front of it. When an M_i or M_{ij} grows to two elements, a new L2/L3-table with two elements is allocated.

```
(* return handle of min x ∈ M : y ≤ x *)
Function locate(y : ℕ) : ElementHandle
    if y > max M then return ∞                                      // no larger element
    i := y[16..31]                                                  // index into root table r
    if r[i] = null  or y > max M_i then return min M_{t^1.locate(i)}
    if M_i = {x} then return x                                      // single element case
    j := y[8..15]                                                   // key for L2 hash table at M_i
    if r_i[j] = null  or y > max M_ij then return min M_{i,t_i^1.locate(j)}
    if M_ij = {x} then return x                                     // single element case
    return r_ij[t_ij^1.locate(y[0..7])]                             // L3 Hash table access

(* find the smallest j ≥ i such that t^k[j] = 1 *)
Method locate(i) for a bit array t^k consisting of n bit words
    (* n = 32 for t^1, t^2, t_i^1, t_ij^1; n = 64 for t^3; n = 8 for t_i^2, t_ij^2 *)
    (* Assertion: some bit in t^k to the right of i is nonzero *)
    j := i div n                                        // which n bit word in b contains bit i?
    a := t^k[nj..nj + n − 1]                            // get this word
    set a[(i mod n) + 1..n − 1] to zero                 // erase the bits to the left of bit i
    if a = 0 then                                       // nothing here → look in higher level bit array
        j := t^{k+1}.locate(j)                          // t^{k+1} stores the or of n-bit groups of t^k
        a := t^k[nj..nj + n − 1]                        // get the corresponding word in t^k
    return nj + msbPos(a)
```

Figure 2: Pseudo code for locating the smallest $x \in M$ with $y \leq x$.

`del`(x) performs a downward pass analogous to find(x) and updates the data structure in an upward pass: Minima and maxima are updated. The list item corresponding to x is removed. When an L2/L3-table shrinks to a single element, the corresponding hash table and top data structure are deallocated. When an element/L3-table/L2-table is deallocated, the top-data structure above it is updated by erasing the bit corresponding to the deallocated entry; when this leaves a zero 32 bit word, a bit in the next higher level of bits is erased etc.

2.2 Variants

The data structure allows several interesting variants:

Saving Space: Our `Stree` data structure can consume considerably more space than comparison based search trees. This is particularly severe if many trees with small average number of elements are needed. For such applications, the 256 KByte for the root array r could be replaced by a hash table with a significant but "nonfatal" impact on speed. The worst case for all input sizes is if there are pairs of elements that only differ in the 8 least significant bits and differ from all other elements in the 16 most significant bits. In this case, hash tables and top data structures at levels two and three are allocated for each such pair of elements. The standard trick to remedy this problem is to store most elements only in the element list. The `locate` operation then first accesses the index data structure and then scans the element list until the right element is found. The drawback of this is that scanning a linked list can cause many cache faults. But perhaps one could develop a data structure where each item of the element list can accommodate several elements. A similar more problem specific approach is to store up to K elements in the L2-tables and L3-tables without allocating hash tables and top data structures. The main drawback of this approach is that it leads to tedious case distinctions in the implementation. An interesting measure is to completely omit the element list and to replace all the L3 hash tables by a single unified hash table. This not only saves space, but also allows a fast direct access to elements whose keys are known. However range queries get slower and we need hash functions for full 32 bit keys.

Multi-sets can be stored by associating a singly linked list of elements with identical key with each item of the element list.

Other Key Lengths: We can further simplify and speed up our data structure for smaller key lengths. For 8 and 16 bit keys we would only need the root table and its associated top data structure which would be very fast. For 24 bit keys we could at least save the third level. We could go from 32 bits to 36–38 bits without much higher costs on a 64 bit machine. The root table could distinguish between the 18 most significant bits and the L2 and L3 tables could also be enlarged at some space penalty. However, the step to 64 bit keys could be quite costly. The root-table can no longer be an array; the root top data structure becomes as complex as a 32 bit data structure; hash functions at level two become more expensive.

Floating Point Keys can be implemented very easily by exploiting that IEEE `floats` keep their relative order when interpreted as integers.

3 Experiments

We now compare several implementations of search tree like data structures. As comparison based data structures we use the STL `map` which is based on red-black trees and `ab_tree` from LEDA which is based on (a, b)-trees with $a = 2$, $b = 16$ which fared best in a previous comparison of search tree data structures in LEDA [11].[5] We present three implementations of integer data structures. `orig-Stree` is a direct C++ implementation of the algorithm described in [10], `LEDA-Stree` is an implementation of the same algorithm available in LEDA [13], and `Stree` is our implementation. `orig-Stree` and `LEDA-Stree` store sets of integers rather than sorted lists but this should only make them faster than the other implementations. The implementations run under Linux on a 2GHz Intel Xeon processor with 512 KByte of L2-cache using an Intel E7500 Chip set. The machine has 1GByte of RAM and no swap space to exclude swapping effects. We use the `g++ 2.95.4` compiler with optimization level `-O6`. Elements are random 31 bit integers without associated information. We report the average execution time per operation in nanoseconds averaged over at least 100 000 operations on an otherwise unloaded machine.

[5] To use (2, 16)-trees in LEDA you can declare a `_sortseq` with implementation parameter `ab_tree`. The default implementation for `sortseq` based on skip lists is much slower in our experiments.

Figure 3 shows the time for the `locate` operation. Already comparing the comparison based data structures is interesting. For small n, when the data structures fit in cache, red-black trees outperform $(2, 16)$-trees indicating that red-black trees execute less instructions. For larger n this picture changes dramatically, presumably because $(2, 16)$-trees are more cache efficient.

Our `Stree` is fastest over the entire range of inputs. For small n, it is much faster than comparison based structures up to a factor of 4.1. For random inputs of this size, `locate` mostly accesses the root-top data structure which fits in cache and hence is very fast. It even gets *faster* with increasing n because then `locate` rarely has to go to the second or even third level t^2 and t^3. For medium size inputs there is a range of steep increase of execution time because the L2 and L3 data structures get used more heavily and the memory consumption quickly exceeds the cache size. But the speedup over $(2, 16)$-trees always exceeds 1.5. For large n the advantage over comparison based data structures is growing again reaching a factor of 2.9 for the largest inputs.

The previous implementations of integer data structures *reverse* this picture. They are always slower than $(2, 16)$-trees and very much so for small n.[6]

We tried the codes until we ran out of memory to give some indication of the memory consumption. Previous implementations only reach 2^{18} elements. At least for random inputs, our data structure is not more space consuming than $(2, 16)$-trees.[7]

Figures 4–6 show the running times for insertions and deletions. `Stree` outperforms $(2, 16)$-trees in most cases but the differences are never very big. The previous implementations of integer data structures and, for large n, red-black trees are significantly slower than

[6]For the LEDA implementation one obvious practical improvement is to replace dynamic perfect hashing by a simpler hash table data structure. We tried that using hashing with chaining. This brings some improvement but remains slower than $(2, 16)$-trees.

[7]For worst case inputs, `Stree` and $(2, 16)$-trees are at a significant disadvantage compared to red-black trees.

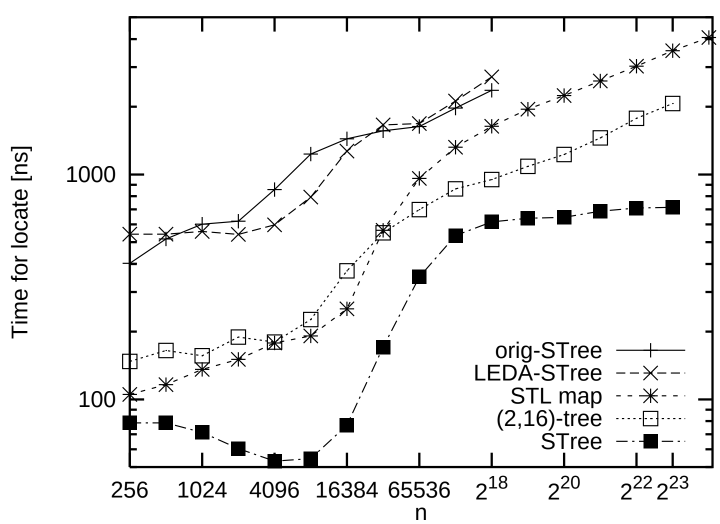

Figure 3: Locate operations for random keys that are drawn independently from the n random elements in the tree.

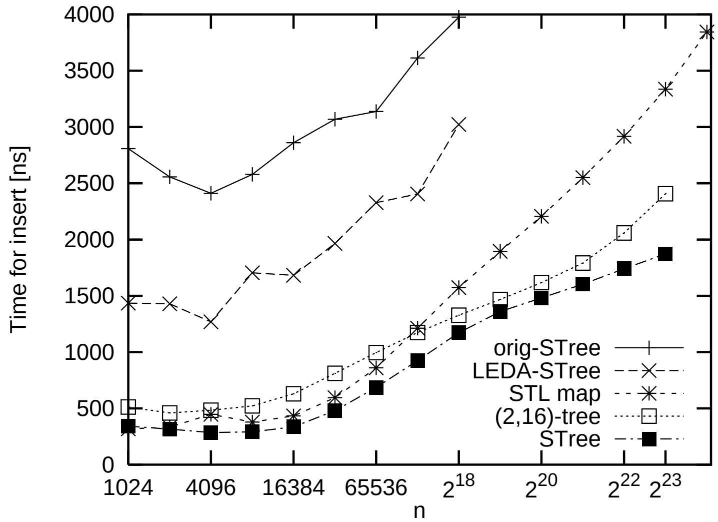

Figure 4: Constructing a tree using n insertions of random elements.

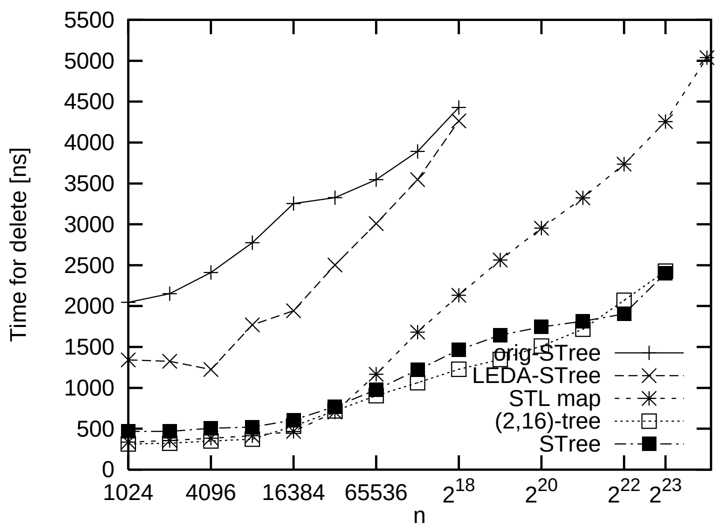

Figure 5: Deleting n random elements in the order in which they were inserted.

`Stree` and $(2, 16)$-trees.

The dominating factor here is memory management overhead. In fact, our first versions of `Stree` had big problems with memory management for large n. We tried the default `new` and `delete`, the g++ STL allocator, and the LEDA memory manager. We got the best performance with with a reconfigured LEDA memory manager that only calls `malloc` for chunks of size above 1024 byte and that is also used for allocating the hash table arrays[8]. The g++ STL allocator also performed quite well.

We have not measured the time for a plain lookup because all the data structures could implement this more efficiently by storing an additional hash table.

[8]By default chunks of size bigger than 256 bytes and all arrays are allocated with `malloc`.

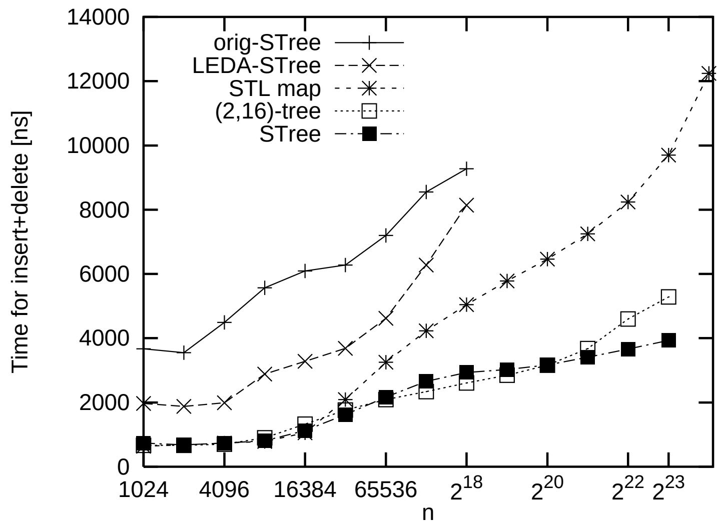

Figure 6: Start with n random elements that are additionally stored in an array. Traverse the array cyclically, delete the current element, replace it by a new random element in the array, and insert this new element into the data structure.

4 Discussion

We have demonstrated that search tree data structures exploiting numeric keys can outperform comparison based data structures. A number of possible questions remain. For example, we have not put particular emphasis on space efficient implementation. Some optimizations should be possible at the cost of code complexity but with no negative influence on speed.

For the final paper we intend to add measurements with operation sequences that try to make vEB trees look as bad as possible (elements $M = \{2^8 i\Delta, 2^8 i\Delta + 255 : i = 0..|M|/2\}$ where $\Delta = \lfloor 2^{25}/|M| \rfloor$, queries for $256j + 128$ for random $j \in 0..2^2 4$). We expect that comparison based data structures are then advantageous for small inputs but that vEB trees still win for large inputs.

An even more interesting test would be to embed the data structure into other algorithms and explore how much speedup can be obtained. However, although search trees are a performance bottleneck in several important applications that have also been intensively studied experimentally (e.g. the best first heuristics for bin packing [5]), we are not aware of real inputs used in any of these studies.[9]

Acknowledgments

We would like to thank Kurt Mehlhorn and Stefan Näher for valuable suggestions.

[9]There are extensive inputs available for dictionary data structure from the 1996 DIMACS implementation challenge. However, they all affect only find operations rather than `locate` operations. Without the need to locate, a hash table would always be fastest.

References

[1] A. Andersson and M. Thorup. A pragmatic implementation of monotone priority queues. In *DIMACS'96 implementation challenge*, 1996.

[2] M. H. Austern. *Generic programming and the STL : using and extending the C++ standard template library.* Addison-Wesley, 7 edition, 2001.

[3] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, pages 643–647, 1979.

[4] P. Berman and B. DasGupta. Multi-phase algorithms for throughput maximization for real-time scheduling. *Journal of Combinatorial Optimization*, 4(3):307–323, 2000.

[5] E. G. Coffman, M. R. Garey Jr., , and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS, 1997.

[6] P. Crescenzi, L. Dardini, and R. Grossi. IP address lookup made fast and simple. In *Euopean Symposium on Algorithms*, pages 65–76, 1999.

[7] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[8] D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

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

[10] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[11] Stefan Näher. Comparison of search-tree data structures in LEDA. personal communication.

[12] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3):80–82, 1977.

[13] Michael Wenzel. Wörterbücher für ein beschränktes Universum (dictionaries for a bounded universe). Master's thesis, Saarland University, Germany, 1992.