Comparing Integer Data Structures for 32 and 64 Bit Keys

Nicholas Nash* and David Gregg

Dept. of Computer Science, Trinity College Dublin, Ireland
{nashn, dgregg}@cs.tcd.ie

Abstract. In this paper we experimentally compare a number of data structures operating over keys that are 32 and 64-bit integers. We examine traditional comparison-based search trees as well as data structures that take advantage of the fact that the keys are integers, such as van Emde Boas trees and various trie-based data structures. We propose a variant of a *burst trie* that performs better in both time and space than all the alternative data structures. Burst tries have previously been shown to provide a very efficient base for implementing cache efficient string sorting algorithms. We find that with suitable engineering they also perform excellently as a dynamic ordered data structure operating over integer keys. We provide experimental results when the data structures operate over uniform random data. We also provide a motivating example for our study in *Valgrind*, a widely used suite of tools for the dynamic binary instrumentation of programs, and present experimental results over data sets derived from Valgrind.

1 Introduction

1.1 Background and Motivation

Maintaining a dynamic ordered data structure over a set of ordered keys is a classic problem, and a variety of data structures can be used to achieve $O(\log n)$ worst-case time for insert, delete, successor, predecessor and search operations, when maintaining a set of n keys. Examples of such data structures include AVL trees [10], B-trees [2,10] and red-black trees [4]. Red-black trees in particular see widespread use via their GNU *C++* STL implementation [19].

Where the keys are known to be integers, better asymptotic results can be obtained by data structures that do not rely solely on pair-wise key comparisons. For example, the stratified trees of van Emde Boas [21] support all operations in $O(\log w)$ worst-case time, when operating on w-bit keys, while Willard's q-fast tries [22] support all operations in $O(\sqrt{w})$ worst-case time.

Such data structures are attractive because of their superior worst-case times compared to comparison-based data structures. However, it is a significant challenge to construct implementations that reveal their better asymptotic

* Work supported by the Irish Research Council for Science, Engineering and Technology (IRCSET).

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 28–42, 2008.

performance, especially without occupying a large amount of extra space compared to comparison-based data structures.

In this paper we experimentally evaluate the performance of a variety of data structures when their keys are either 32 or 64-bit integers. In particular we find that a carefully engineered variant of a *burst trie* [7] provides the best performance in time, and for moderate to large numbers of keys it requires less space than even a well implemented comparison-based search tree.

A noteworthy application of our data structure occurs in the dynamic binary instrumentation tool *Valgrind* [12]. Valgrind comprises a widely used suite of tools for debugging and profiling programs. Internally, Valgrind frequently queries a dynamic ordered data structure that maps machine words to machine words. On current platforms, these machine words are either 32 or 64 bits in length. At present, Valgrind uses an AVL tree to perform these mappings. A significant performance improvement could be obtained by replacing the AVL tree with a more efficient data structure. We present experimental results for a number of data structures when used to track every data memory access done by a program, as for example some Valgrind-based tools do.

1.2 Related Work and Contributions

In this paper we compare the performance of a carefully engineered variant of a burst trie in both time and space to AVL trees, red-black trees and B-trees. Aside from these commonplace general purpose data structures, we also experimentally examine the performance of two slightly more ad-hoc data structures [5,11] which are tailored for the case of integers keys, and have been shown to perform well in practice. We briefly describe these data structures in the remainder of this section.

Dementiev *et al.* [5] describe the engineering of a data structure based on stratified trees [21] and demonstrate experimentally that it achieves superior performance to comparison-based data structures. We refer to their engineered data structure as an S-tree. Although highly efficient in time, the S-tree is tailored around keys of 32-bits in length and generalizing the data structure to 64-bit keys would not be feasible in practice because of the large amount of space required to maintain efficiency. Indeed, even for 32-bit keys the data structure requires more than twice as much space as a typical balanced search tree.

Korda and Raman [11] describe a data structure similar to a q-fast trie [22] and experimentally show that it offers performance superior to comparison-based data structures. Unlike the S-tree data structure engineered by Dementiev *et al.* this data structure is not restricted to 32-bit keys and requires less space in practice. We now briefly describe the features of Korda and Raman's data structure relevant to our discussion. We refer to their data structure as a KR-trie. A KR-trie consists of a path compressed trie containing a set of *representative keys*, $K_1 < K_2 < \cdots < K_m$. Associated with each representative key K_i is a bucket data structure B_i containing the set of keys $\{k \in S : K_i \leq k < K_{i+1}\}$ for $i < m$, and $\{k \in S : k \geq K_m\}$ for $i = m$, where S is the entire set of keys in the data structure.

Each bucket contains between 1 and $b-1$ keys. When a new key is inserted into the data structure the compressed trie is first searched for its predecessor key, giving a representative key K_i. If the associated bucket B_i already contains $b-1$ keys, a new representative key is added to the compressed trie that partitions the bucket into two new buckets containing $b/2$ keys each. Deletions operate in a similar manner to insertions, except that when two adjacent buckets B_i and B_{i+1} contain fewer than $b/2$ keys in total the keys of B_{i+1} are inserted into B_i and K_{i+1} is deleted from the trie. A search in the data structure is accomplished by a predecessor query in the compressed trie, followed by a search in the relevant bucket data structure.

There are many other non-comparison-based data structures in addition to the two just mentioned, both practical and theoretical. Two practical examples are LPC-tries [13] and the cache-friendly tries of Achyra *et al.* [1], however, we believe these data structures are less efficient or less general than the two data structures described above. For example, LPC-tries offer search operations more efficiently than binary search trees, but insertions are slower. In contrast, the data structures described above perform better than binary search trees for all operations. The tries of Achyra *et al.* appear efficient, but require knowledge of cache parameters and focus only on trie search. It is not clear how operations like predecessor and successor could be efficiently implemented, since hash tables are used inside the trie nodes.

The contribution of our experimental study is to show that a carefully engineered data structure based on the burst trie described by Heinz *et al.* [7] performs better than both the S-tree and KR-trie data structures described above, as well as the traditional comparison-based data structures.

The work of Heinz *et al.* focuses on the problem of *vocabulary accumulation*, where the keys are variable length strings. The only operations performed are insert and search, with a final in-order traversal of the burst trie. In contrast, we consider the case of integer keys with all the operations usually associated with a dynamic ordered data structure.

The contributions of our work are as follows:

- We provide a thorough experimental comparison of dynamic data structures over 32 and 64-bit integer keys. We provide time and space measurements over random data as well over data sets that occur in Valgrind, a notable application of such data structures.
- We show that burst tries extend efficiently to a dynamic ordered data structure, showing how the operations usually associated with such data structures can be implemented efficiently through careful engineering.
- We show that the data structure is more efficient in time than the best previous data structures that have been engineered for the case of integer keys. We also show that for large numbers of keys, the data structure requires less space than even space efficient implementations of comparison-based search trees.

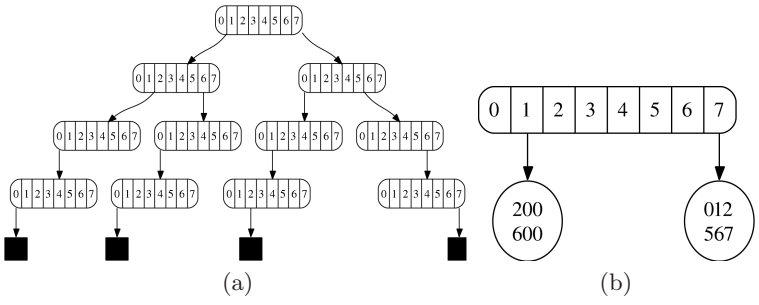


Fig. 1. (a) Shows a trie holding the keys 1200, 1600, 7012 and 7567. The leaves of the trie (black squares) hold the satellite data associated with the keys. A corresponding burst trie is shown in (b).

2 Background

In this section we provide the definition of a burst trie and some basic background information regarding the data structure.

We assume the burst trie contains fixed length keys, each of length l. A key is a sequence $k_1 \cdots k_l$ where each k_i, $1 \leq i \leq l$ is drawn from a set of digits $\{0, \ldots, U-1\}$. In practice, our keys are 32 and 64-bit integers and we often choose $U = 256$, for implementation reasons. Given a trie T over a set of keys, we call a node *small* only if its parent has more than c descendant leaves, but the node itself has at most c descendant leaves. A burst trie with bucket size c is obtained from a trie T by replacing every small node x in T with a bucket data structure containing the keys corresponding to the leaves descendant from x and discarding all descendants of small nodes. It follows that if two keys $k_1 \cdots k_m$ and $k'_1 \cdots k'_m$ reside in the same bucket of a burst trie at depth d, then $k_i = k'_i$ for $1 \leq i < d$, and only their suffixes need be stored in the bucket data structure. Figure 1(a) shows an example of a trie while Figure 1(b) shows a burst trie corresponding to it.

Although we refer to what has just been described as a burst trie, using some kind of bucketing in a trie is an old technique. Sussenguth [20] provides an early suggestion of the technique, while Knuth analyses bucketed tries [10]. In addition, Knessl and Szpankowski [8,9] analyse what they refer to as b-tries — tries in which leaf nodes hold up to b keys.

We use the term burst trie of Heinz *et al.* [7] because their work was the first to provide a large scale investigation of alternative bucket data structures, the time and space trade-offs in practice resulting from bucketing, and the *bursting* of bucket data structures during insertions, which we describe below.

Searching in a burst trie is similar to searching in a conventional trie. The digits of the key are used to determine a path in the trie that either terminates with a NIL pointer, in which case the search terminates unsuccessfully, or a bucket is found. In the latter case, the search finishes by searching the bucket data structure for the key suffix.

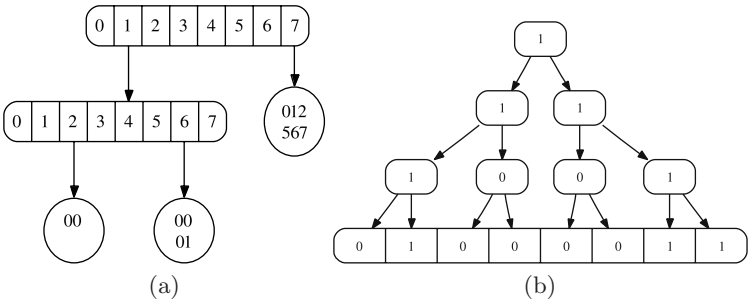


Fig. 2. (a) Shows the burst trie of Figure 1(b) after inserting the key 1601. Assuming the buckets can hold at most two key suffixes, inserting the key 1601 causes the left bucket shown in Figure 1(b) to burst. In (b) an OR-tree is shown, a possible in-node data structure for implementing a burst trie.

Insertion of a key into a burst trie is also straightforward. The digits of the key are used to locate a bucket where the key suffix should be stored. If no such bucket exists, one is created. On the other hand, if a bucket is found and it contains fewer than c keys it need not be burst and the key suffix is simply added to that bucket. Otherwise, if the bucket already contains c keys, it is burst. This involves replacing the bucket with a trie node and distributing the keys suffixes of this bucket into new buckets descending from this new trie node. Figure 2(a) shows an example of a burst operation occuring on the burst trie of Figure 1(b). It is possible that all keys from the burst bucket belong in the same bucket in the newly created node. In this case, the bursting process is repeated.

Deleting a key k from a burst trie is performed by first searching for the bucket where k is stored, as described above. If there is no such bucket, no deletion need occur. Otherwise, k is deleted from some bucket b at a node x. If b is then empty, it is deleted from x. If x then has only NIL child and bucket pointers x is deleted from the trie. This step is repeated, traversing the path from x to the root of the trie deleting ancestors encountered with only NIL child or bucket pointers. The traversal terminates when either a node with a non-nil pointer is encountered, or the root of the trie is reached.

3 Engineering Burst Tries

Although the burst trie data structure described in the preceding section leads to a highly efficient data structure, especially for strings, as shown by Heinz *et al.* [7], a little care must be taken when engineering it for the case of an ordered data structure for integer keys. Our variant of a burst trie makes use of two data structures that have a significant influence on its performance: (1) The bucket data structures at the leaves of the trie, and (2) the data structures inside the nodes of the burst trie. We describe the alternatives for this latter data structure in the next section.

3.1 In-Node Data Structures

Given a node x in a trie-based data structure with branching factor b, and an index i, $0 \leq i < b$, it is often necessary to find $\textsc{Succ}(i)$, that is, the smallest $j > i$ such that $x[j] \neq \textsc{nil}$. This is the bucket or child node pointer directly following $x[i]$. It is also often required to find $\textsc{Pred}(i)$, the largest $j < i$ such that $x[j] \neq \textsc{nil}$. These operations upon nodes are required, for example to support queries on the trie for the smallest key greater than or equal to some given key. We elaborate on the precise use of these operations in Section 3.3.

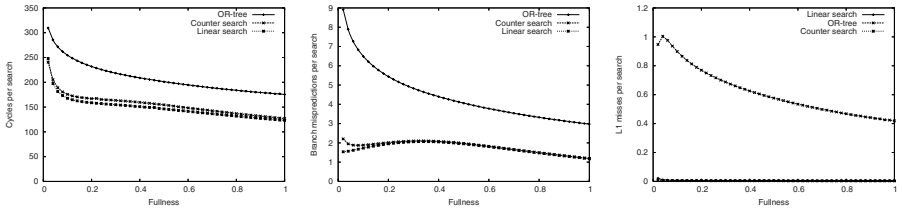
The simplest data structure supporting these predecessor and successor operations is just a linear search over a bit-vector. This data structure requires only $O(1)$ time when a new bucket or child is added or removed from the node, however, Pred and Succ are inefficient, requiring $O(b)$ time.

An alternative in-node data structure is an OR-tree. Figure 2(b) shows an example of this data structure. A breadth-first traversal of an OR-tree can be laid out in an array inside each node, requiring an additional $O(b)$ space compared to a simple bit-vector approach. However, an OR-tree offers all operations in $O(\lg b)$ time.

As a compromise between these two data structures, Pred and Succ can be implemented using $\lceil\sqrt{b}\rceil$ counters. Where the i^{th} counter, $0 \leq i < \lceil\sqrt{b}\rceil$ holds a count of the non-zero bits in the range $[i\lceil\sqrt{b}\rceil, i\lceil\sqrt{b}\rceil + \lceil\sqrt{b}\rceil - 1]$ (except perhaps for the last counter, which covers the range $[b - \lceil\sqrt{b}\rceil, b-1]$). This data structure allows insertions and deletions in $O(1)$ time and supports Pred and Succ in $O(\sqrt{b})$ time, requiring at most $\lceil\sqrt{b}\rceil$ counters to be examined followed by at most $\lceil\sqrt{b}\rceil$ bits.

To determine the most efficient in-node data structure we conducted a number of experiments, randomly populating a bit-vector and then performing a large number of successor queries. Figure 3(a) shows that the OR-tree is much less efficient than either performing a simple linear scan, or using counters to guide the search. Figures 3(b) and (c) reveal why this is so. Firstly, as Figure 3(b) shows the OR-tree causes by the far most branch mispredictions. Intuitively, one expects that an algorithm with a better time complexity increases the information it extracts from each branch instruction — thus making each branch instruction less predictable. Secondly, as Figure 3(c) shows, the OR-tree has very bad cache performance compared to the linear scan or counter based search. This is to be expected since the the OR-tree's breadth-first layout is cache unfriendly, while the other two algorithms perform linear searches, which make full use of every cache line. Note that the bad performance of the OR-tree is despite the fact that it executes the fewest instructions of any of the algorithms.

We selected the counter based search for our burst trie implementation because it has very similar performance to the linear scan in practice and better performance in the worst-case (i.e. when a trie node is very sparse). Other in-node data structures could be used. For example, recursively applying the $O(\sqrt{b})$ approach essentially leads to a stratified tree which would provide all operations in $O(\lg \lg b)$ time. However, since the branching factors of our trie nodes are never greater than 2^{16} this approach is unlikely to yield a performance benefit.



(a) (b) (c)

Fig. 3. (a) Shows the cycles per successor operation (i.e. time) on randomly populated 2^{14} entry bit vector. "Fullness" denotes the number of randomly inserted bits as a fraction of the total size of the bit vector. The OR-tree performs much worse than either a simple linear scan or counter based search. This is due to the large number of branch mispredictions, shown in (b) and cache misses, shown in (c), incurred by the OR-tree compared to the other data structures. The cache misses shown are level 1 data cache misses, since the entire bit-vector fits in the level 2 cache. These results are averaged over several thousand repetitions, and were gathered using PAPI [6].

In the next section we describe the second important data structure used by burst tries — the bucket data structure.

3.2 Bucket Data Structures

The choice of data structure used for the buckets of a burst trie is critical in achieving good performance. Heinz *et al.* [7] concluded that unbalanced binary search trees holding at most 35 strings offered the best performance as a bucket data structure. They also experimented with linked lists and splay trees [18]. Since the maximum number of keys stored in each bucket is modest (at most 35), a simple bucket data structure, even with bad asymptotic behaviour, may perform well. We experimented with balanced binary trees as well as with sorted arrays as bucket data structures, and found that sorted arrays are far more efficient in practice than the search trees. We also found that a bucket size of about 256 keys gave best performance. It is likely that the sorted arrays incur far fewer cache misses than the search trees. In fact, unsorted arrays of strings have been used as bucket data structures for burst tries as a basis for the *burstsort* algorithm [14,15,16,17], a cache-efficient radix sorting algorithm.

In contrast to the array buckets of the burstsort algorithm, our buckets are sorted and much smaller (in burstsort the buckets are allowed to grow until they reach the size of the processor's 2nd level cache, which can be several megabytes in size) holding at most 256 keys. The buckets are implemented as growable sorted arrays, and an insertion involves possibly doubling the size of the bucket followed by a linear scan to find the correct position for the key to be inserted.

Often the most frequent operation executed on a data structure is a search, and so searching buckets in particular should be efficient. We use a binary search that switches to a linear search when the number of keys which remain to be searched falls below a certain threshold. We found a threshold of between 16 and

32 keys gave a performance improvement over a simple binary search. Our burst trie implementation is designed to provide a mapping from a key to the satellite data associated with that key, which we refer to as the *value* for the key. To improve the spatial locality of searches the keys and values of a bucket should not be interleaved. Rather, all the keys should be stored sequentially, followed by all the values of that bucket. This ensures searching for a key makes better utilization of the processor's cache lines.

If the maximum bucket capacity is c, it takes $O(c)$ time to insert into a bucket and $O(\lg c)$ time to search in a bucket. Finally, since bursting a bucket just involves splitting the sorted sequence of keys it contains into a number of other sorted sequences, bursting a bucket also takes $O(c)$ time.

3.3 Operations

The preceding two sections have described the two main data structures required for extending burst tries to an ordered data structure. We now show how these data structures can be used efficiently to provide a burst trie with all the usual operations associated with a dynamic ordered data structure. Note that in order that predecessor and successor operations are supported efficiently, it is wise to maintain the leaves of the burst trie (i.e. the buckets) in order in a doubly linked list.

Locate. We first describe the *locate* operation, which finds the value associated with the smallest key greater than or equal to a supplied key k (or NIL if there is no such key). Assuming the path in the burst trie determined by k leads to a bucket, then that bucket is searched for the smallest key suffix greater than or equal to k's suffix, and its corresponding value is returned. In this case, the locate operation takes $O(h + \lg c)$ time, where h is the maximum height of the trie, and each bucket holds at most c keys. In the case where k does not lead to a bucket, the in-node data structure is queried to find a bucket requiring $O(\sqrt{b})$ time, thus *locate* requires $O(h + \max(\lg c, \sqrt{b}))$ time.

Insert. If inserting a new key k requires the creation of a new bucket the in-node data structure and doubly linked list of buckets must be updated. This requires finding the two buckets whose keys are the immediate predecessors and successors of k, and can be accomplished in time $O(h + \sqrt{b})$ time. Note that the in-node data structures should be augmented with indices storing the minimum and maximum non-nil pointer at each node, which we refer to as the node's LOW and HIGH fields respectively. The LOW and HIGH fields are used to avoid avoid the process of locating the predecessor and successor buckets requiring $O(h\sqrt{b})$ time. In the case where an existing non-full bucket is found for k, the insertion takes time $O(h+c)$ (recall that the buckets are simply sorted arrays). Finally, in the case where bursting must occur, the insertion can take time $O(hc)$ at worst, since an insertion to a full bucket may repeatedly cause all key suffixes to enter the same new bucket deeper in the trie. A straightforward argument can be used to show insertion requires $O(h + \max(c, \sqrt{b}))$ amortized time.

Other Operations. Deletion is carried out as was described in Section 2, except that when an empty bucket is deleted from a node, the in-node data

structure and linked list of buckets should also be updated. Detecting whether a node should be removed from the trie following a deletion is accomplished by examining its LOW and HIGH fields. Note that when a node x is to be removed from the trie, its parent's in-node data structure should only be updated if the parent itself is not also to be removed as a result of the removal of x. This ensures that $O(\sqrt{b})$ time is spent updating in-node data structures, rather than $O(h\sqrt{b})$ time. Since it takes $O(c)$ time to delete a key from a bucket, deletion takes $O(h + \max(\sqrt{b}, c))$ time in total.

Predecessor and successor operations can be implemented with minor modifications to the locate operation described above. Often, predecessor and successor operations on a data structure are supported via iterators. In this case, by using the linked list of buckets, predecessor and successor both operate in constant time.

4 Results

4.1 Experimental Setup

We now describe the experimental comparison of our burst trie variant with a number of other data structures. For our experiments over 32-bit keys we used an Intel Core 2 processor with a clock-speed of 2.13GHz, a second level cache size of 2MB and 4 GB of main memory. For our experiments over 64-bit keys we used an Intel Core 2 processor with a clock speed of 2.0GHz, a second level cache size of 4MB and 4GB of main memory. Note that our experiments investigate the case where the entire data structure fits in main memory. All results we present below are averaged over several thousand runs. Although not presented below, to verify the robustness of our results we have also conducted experiments on Sun SPARC as well as PowerPC architectures, and observed results very similar to the ones we describe below.

We compare our implementation to the *C++* STL map implementation [19], which uses a red-black tree. We also compare to the AVL tree implementation used internally in Valgrind [12], except that we have implemented a custom memory allocator to reduce memory usage and improve performance. We also include a comparison with an optimized B-tree implementation, as well as with the stratified tree based data structure of Dementiev *et al.* [5], which we refer to below as an S-tree. Finally, we include a comparison with a KR-trie (described in Section 1.2). For the KR-trie we use the same in-node data structures and bucket data structures as we used for the burst trie[1].

We used uniform random data as well as data generated internally by Valgrind to assess the relative performance of the data structures. We used Brent's

[1] This differs slightly from the implementation of the KR-trie described by Korda and Raman [11], since they use fixed size rather than growable arrays as buckets. However, using growable arrays improves performance and reduces memory consumption. Moreover, Korda and Raman do not precisely specify the in-node data structure they use.

[3] pseudorandom number generator implementation for generating both 32 and 64-bit random numbers. The data sets generated using Valgrind consist of the memory addresses of all the data memory accesses performed during the execution of a program. This reflects the use of the data structure to track every memory access performed by a program, as is done by some Valgrind-based tools. We generated data sets for the Linux program Top (a task viewer) as well as three applications from the K Desktop Environment: Amarok (a music player), Konqueror (a web browser and file manager) and KPDF (a PDF viewer). Each of these data sets contain between 10^7 and 10^9 operations in total, and 70-80% of the operations are loads. A load generates a search operation on the appropriate data structure while a store generates an insert operation. Currently, Valgrind uses an AVL tree to perform these operations. Note that iteration over the data structure is also required at certain times, removing the possibility of using a hash table.

4.2 Random Data

Figure 4(a) shows the time per insertion for the data structures for uniform random 32-bit keys. Note that the S-tree and red-black tree use all available memory at 2^{25} and 2^{26} keys respectively. The burst trie performs best of all the data structures, although the S-tree and KR-trie are also competitive. The comparison-based data structures are not as competitive, although the B-tree performs quite well.

Figure 4(b) shows the time per search operation, the vast majority of the searches are for keys that are not in the data structure. Before it runs out of memory, the S-tree data structure performs slightly better than the burst trie. Note that this is the only operation for which the S-tree out-performs the burst trie. The KR-trie performs worse than the burst trie and S-tree, followed by the B-tree. The binary search trees perform quite badly in comparison to these data structures.

Figure 4(c) shows the memory consumption of the data structures. The S-tree is clearly a very memory hungry data structure. The comparison-based data structures are attractive because of their uniform memory overhead. It is noteworthy that the use of a custom memory allocator for the AVL tree allows it to occupy significantly less memory than the red-black tree. The B-tree, which uses nodes consisting of growable arrays of keys of up to 4KB in size, uses the least memory of any of the comparison-based data structures. The KR-trie and burst trie are highly inefficient in their memory usage until the number of keys becomes large. However, when the number of keys becomes large, beyond about 2^{18} (262,144) keys the burst trie in particular has a very modest memory overhead. The rapid increase and then decrease in memory consumption of the KR-trie beginning at 2^{20} keys is a result of the fact that the KR-trie uses a compressed trie to control access to its buckets. Beginning at 2^{20} keys, many compressed branches of the trie are expanded, resulting in poor space utilization of their links and buckets. Subsequently, the buckets and links become filled and used, and space utilization improves again.

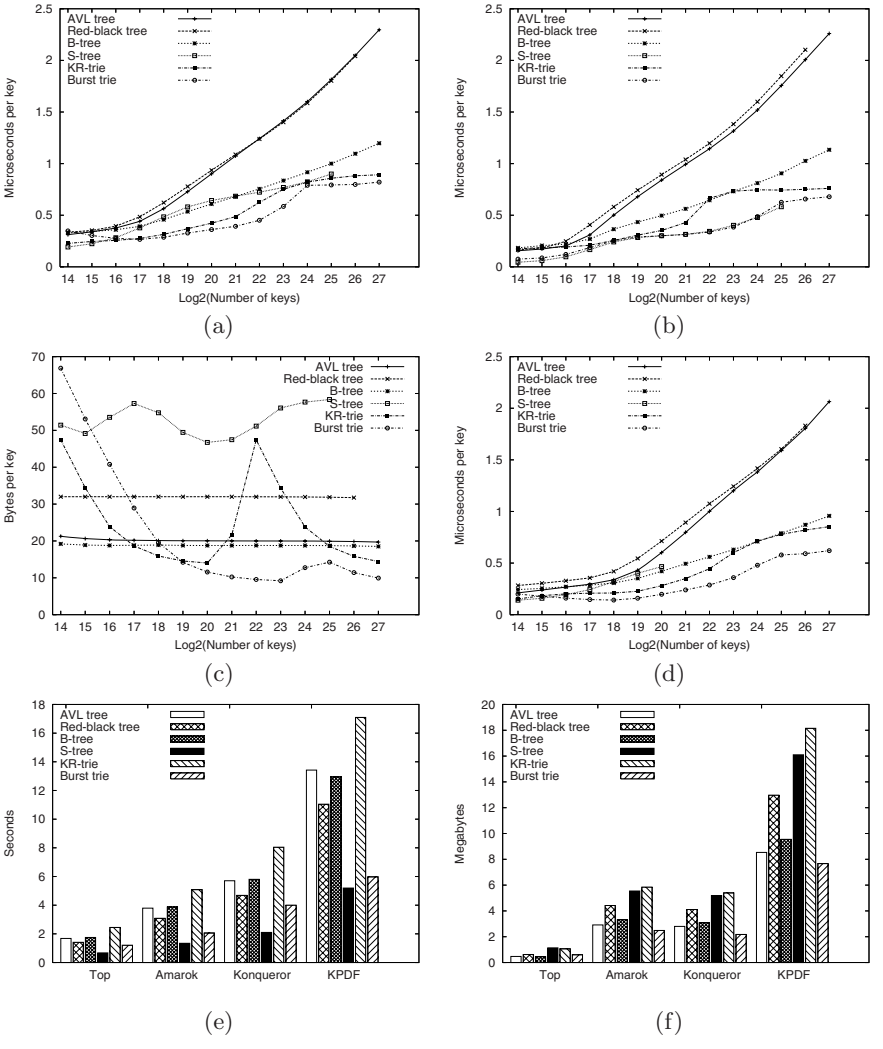


Fig. 4. These figures show a comparison of the data structures when operating on 32-bit keys. (a) Shows the time per insertion operation for the data structures. (b) Shows the time per locate operation for the data structures, a locate operation returns the smallest key greater than or equal to a given key. (c) Shows the number of bytes per key of memory consumed by the data structures following a sequence of insertions. (d) Shows the time for a mixed sequence of equiprobable insertion and deletion operations. The results of (a)—(d) are over uniform random keys. The charts in (e) and (f) show, respectively, the time and space required by the data structures required to process various Valgrind data sets. The results are discussed in Section 4.

Figure 4(d) shows the time per operation required for a mixed sequence of insertions and deletions, which occur randomly and are equiprobable. The burst

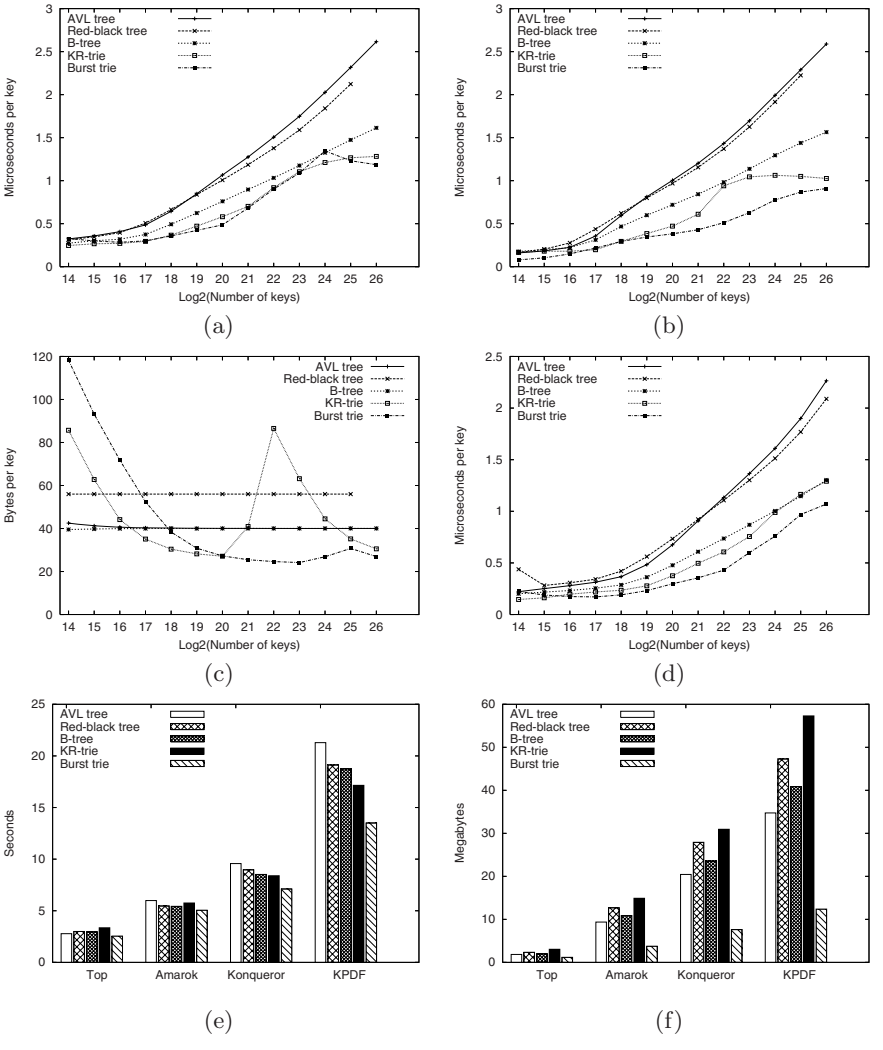


Fig. 5. These figures show a comparison of the data structures when operating on 64-bit keys, and are broadly similar to the results in the 32-bit case, shown in Figure 4. The S-tree is absent because it is restricted to 32-bit keys. (a) Shows the time per insertion operation for the data structures. (b) Shows the time per locate operation for the data structures, a locate operation returns the smallest key greater than or equal to a given key. (c) Shows the number of bytes per key of memory consumed by the data structures following a sequence of insertions. (d) Shows the time for a mixed sequence of equiprobable insertion and deletion operations. The results of (a)—(d) are over uniform random keys. The charts in (e) and (f) show, respectively, the time and space required by the data structures required to process various Valgrind data sets. The results are discussed in Section 4.

trie is also the best performing data structure in this case, although the KR-trie and B-tree also perform quite well. Unfortunately the only S-tree implementation available to us[2] had bugs in its delete operation causing it to fail on inputs larger than 2^{20} keys.

Over the full range of operations, on random 32-bit keys the burst trie's performance is competitive with or superior to that of all the other data structures in time. In addition, when the number of keys is large it also uses the least memory of any of the data structures.

The results for random 64-bit keys are shown in Figure 5(a)—(d). The S-tree is excluded from these results because it is tailored specifically for 32-bit keys, and extending it efficiently to 64-bit keys requires an enormous amount of extra space. For the remaining data structures the results for the 64-bit case are broadly similar to those observed in the 32-bit case. The burst trie and KR-trie perform better than the comparison-based data structures, with the B-tree performing the best of the comparison-based data structures. In addition, it appears the burst trie has the edge over the KR-trie in both time and space.

It is noteworthy that the burst trie achieves its space efficiency in part because it stores a trie over the common prefixes of keys (described fully in Section 2) and as a result only stores key suffixes in buckets. This also improves the cache performance of searches in the buckets. However, the trie of representative keys (described in Section 1.2) stored by the KR-trie does not guarantee that the keys in the same bucket of a KR-trie share a common prefix.

In addition, the space occupied by the burst trie could perhaps be further reduced by ensuring its trie is compressed. However, at least over uniform random data chains of single-children nodes are less probable than in a traditional trie, and so the space saved by compression may be modest. In addition, maintaining a compressed trie can be quite expensive in time.

4.3 Valgrind Workloads

Figure 4(e) shows the time for processing 32-bit Valgrind data sets of various programs (these data sets are described in Section 4.1). The S-tree is the most efficient data structure in time, followed by the burst trie. It is notable that the KR-trie performs the worst on these Valgrind data sets. Figure 4(f) shows the memory consumed by the data structures in processing the data sets. Except for on the smallest trace, the burst trie requires the least memory of any of the data structures. Both the S-tree and KR-trie require much more space than the comparison-based data structures and the burst trie.

Figure 5(e) shows the time for processing the Valgrind data sets in the 64-bit case. The S-tree is excluded because it cannot operate on 64-bit keys. The burst trie is the most efficient data structure, with the KR-trie and B-tree also performing quite well. As Figure 5(f) shows, the burst trie is by far the most space efficient data structure on the data sets.

On the 32-bit data sets, the S-tree performs better than the burst trie, however, it requires almost twice as much memory. On the 64-bit data sets, the burst

[2] Obtained from `http://www.mpi-inf.mpg.de/~kettner/proj/veb/index.html`

trie is the best performing data structure, as well as requiring the least space of any data structure.

5 Conclusion

This paper has provided an experimental comparison of efficient data structures operating over 32 and 64-bit integer keys. In particular we have shown that extending burst tries to an ordered data structure for integer keys provides a data structure that is very efficient in both time and space.

In comparisons using uniform random data with AVL trees, red-black trees and B-trees we have shown that for moderate to large sized inputs, burst tries provide all operations more efficiently in both time and space. We have also compared our extended version of burst tries to Dementiev *et al.*'s S-tree data structure based on stratified trees, and found that while Dementiev *et al.*'s data structure is competitive in time, it requires far more memory than a burst trie and is less general, being restricted to 32-bit keys. We have also compared burst tries to KR-tries, a data structure based on Willard's q-fast tries. We carefully engineered an implementation of KR-tries, using the same bucket and node data structures as our burst trie, and found that they are generally slightly less efficient than burst tries. One significant advantage of a burst trie over a KR-trie is that because of a burst trie's organisation, it need only store key suffixes in buckets, improving space usage as well as cache performance.

The data structure presented in this paper has wide applicability, and furthermore our results are robust, having been verified on several different architectures. We have presented results for an application of our data structure in Valgrind where the keys are 32 and 64-bit integers. Our results show that in the 32-bit case only the S-tree data structure operates faster, but the S-tree requires almost twice as much space as the burst trie. In the 64-bit case, the burst trie requires less space and operates more rapidly than any of the alternative data structures. This paper demonstrates, through the application of our data structure in Valgrind together with the results presented over random data, that burst tries should be considered as one of the many alternative data structures for applications requiring a general purpose dynamic ordered data structure over keys such as integers or floating point numbers.

Acknowledgements. The authors are grateful to Julian Seward for all his patient assistance with Valgrind. We also thank the anonymous reviewers for their helpful comments.

References

1. Acharya, A., Zhu, H., Shen, K.: Adaptive algorithms for cache-efficient trie search. In: Goodrich, M.T., McGeoch, C.C. (eds.) ALENEX 1999. LNCS, vol. 1619, pp. 296–311. Springer, Heidelberg (1999)
2. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf. 1, 173–189 (1972)

3. Brent, R.P.: Note on marsaglia's xorshift random number generators. Journal of Statistical Software 11(5), 1–4 (2004)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn., pp. 273–301. MIT Press, Cambridge, MA, USA (2001)
5. Dementiev, R., Kettner, L., Mehnert, J., Sanders, P.: Engineering a sorted list data structure for 32 bit keys. In: Proc. of the Sixth SIAM Workshop on Algorithm Engineering and Experiments, New Orleans, LA, USA, pp. 142–151 (2004)
6. Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., You, H., Zhou, M.: Experiences and lessons learned with a portable interface to hardware performance counters. In: IPDPS 2003: Proc. of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, p. 289.2. IEEE Computer Society, Los Alamitos (2003)
7. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM Trans. Inf. Syst. 20(2), 192–223 (2002)
8. Knessl, C., Szpankowski, W.: Heights in generalized tries and patricia tries. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 298–307. Springer, Heidelberg (2000)
9. Knessl, C., Szpankowski, W.: A note on the asymptotic behavior of the heights in b-tries for b large. Electr. J. Comb. 7 (2000)
10. Knuth, D.E.: The Art Of Computer Programming. Sorting And Searching, 2nd edn., vol. 3, pp. 458–478, 482–491, 506. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
11. Korda, M., Raman, R.: An experimental evaluation of hybrid data structures for searching. In: Proc. of the 3rd International Workshop on Algorithm Engineering (WAE), London, UK, pp. 213–227 (1999)
12. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
13. Nilsson, S., Tikkanen, M.: An experimental study of compression methods for dynamic tries. Algorithmica 33(1), 19–33 (2002)
14. Sinha, R.: Using compact tries for cache-efficient sorting of integers. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 513–528. Springer, Heidelberg (2004)
15. Sinha, R., Ring, D., Zobel, J.: Cache-efficient string sorting using copying. J. Exp. Algorithmics 11, 1.2 (2006)
16. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. J. Exp. Algorithmics 9, 1.5 (2004)
17. Sinha, R., Zobel, J.: Using random sampling to build approximate tries for efficient string sorting. J. Exp. Algorithmics 10, 2.10 (2005)
18. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM 32(3), 652–686 (1985)
19. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
20. Sussenguth, E.H.: Use of tree structures for processing files. Commun. ACM 6(5), 272–279 (1963)
21. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Inf. Process. Lett. 6(3), 80–82 (1977)
22. Willard, D.E.: New trie data structures which support very fast search operations. J. Comput. Syst. Sci. 28(3), 379–394 (1984)