**Key words:** compact hashing, hash trees, invertible hash function, trie structures, text compression, adaptive prediction

## Abstract

This paper shows how trees can be stored in a very compact form, called "Bonsai," using hash tables. A method is described that is suitable for large trees that grow monotonically within a predefined maximum size limit. Using it, pointers in any tree can be represented within $6 + \lceil \log_2 n \rceil$ bits per node where $n$ is the maximum number of children a node can have. We first describe a general way of storing trees in hash tables, and then introduce the idea of compact hashing which underlies the Bonsai structure. These two techniques are combined to give a compact representation of trees, and a practical methodology is set out to permit the design of these structures. The new representation is compared with two conventional tree implementations in terms of the storage required per node. Examples of programs that must store large trees within a strict maximum size include those that operate on trie structures derived from natural language text. We describe how the Bonsai technique has been applied to the trees that arise in text compression and adaptive prediction, and include a discussion of the design parameters that work well in practice.

## Introduction

When storing large trees, space is often at a premium. For many applications, pointers consume most of the space. These are most commonly represented in $\lceil \log_2 S \rceil$ bits where $S$ is the size of memory. Of course, array based implementations can store pointers in $\lceil \log_2 M \rceil$ bits where $M$ is the number of the nodes available for tree nodes. However, there is implicit information in the size of nodes and their location in memory that can be exploited to further reduce the size of pointers.

This paper shows how trees can be stored in an extremely compact form, called "Bonsai," using hash tables. The pointers in any tree can be represented using $6 + \lceil \log_2 n \rceil$ bits per node where $n$ is the maximum number of children a node can have. Note that this figure is the total size of all $n$ pointers together, not the size of each individual one. Since the method involves hashing, an overhead must be added to the space allocation to minimize collisions. The amount needed is governed by the desired access time, but 40% is reasonable for most applications.

The Bonsai method is suitable for large trees which grow monotonically within a predefined maximum size limit. Collisions are handled probabilistically and there is a chance that an insertion will fail. This can be made arbitrarily small by increasing the number of bits used to resolve collisions. In this paper we design to an error rate of $1/M$ at 80% occupancy, but allocating one additional bit per node yields a rate of $1/M^2$.

There are many applications that must store large trees within a strict maximum size and can tolerate a low, but non-zero, probability of error. Examples are the PPM text compression method[1] and the REACTIVE KEYBOARD predictive text generation system,[2] which both store trie data structures derived on the fly from the text seen so far. (A trie, from the word re*trie*val, is a type of tree data structure designed to speed node retrieval.) For example, a compression model formed by storing successive 7-tuples of characters from a 770,000-character book (book1 of the Calgary Compression Corpus[1]) produces a trie with 760,000 nodes, each of which includes a character, a count, and pointers. The pointers associated with a node occupy a total of 8 bits using the Bonsai method as against 64 bits with conventional address pointers. Thus, in round terms, a Bonsai node needs only 3 bytes versus the 10 bytes required for a conventional representation. Allowing a 20% overhead for collision control, the whole tree consumes slightly more than a third of the 7.6 Mbytes normally required.

Using a Bonsai, a constant average penalty per access is paid; this is independent of the size of the tree and proportional to memory occupancy. The main restriction of the method is that while leaf nodes can easily be added or deleted, inserting or deleting internal nodes is expensive. This is because a pointer to a node is coded with respect to its parent's physical location to reduce its size. It turns out that when an internal node is deleted, all its children—and all their children, and so on—must be repositioned in memory if the space it uses is to be reclaimed.

The next section describes a general way of storing trees in hash tables. Following that we introduce the idea of compact hashing, which underlies the Bonsai structure. These two techniques are then combined to give a compact representation of trees. The section following that compares the new technique with two conventional tree implementations in terms of the storage required per node, and describes how it has been applied to trees used in PPM and the REACTIVE KEYBOARD.

## Hash trees

We first review how to represent a tree using hash tables. The description is independent of any particular hashing algorithm; an implementation using compact hashing is described later.

Hash tables store values associated with a single *key*. A hash function is computed from the key and used as a first-guess location when seeking it in the table. If the location is occupied by another key, one of many possible *collision algorithms* is used to select further locations for searching. Some information about the key must be stored in each occupied location to permit disambiguation. In most designs the stored key is somewhat larger than an index into the hash table; however, using the Bonsai method outlined later the stored key can be greatly reduced.

## Hash tables

A hash table consists of an array of keys and their associated values. The values stored with a particular key do not materially affect the algorithms below and are therefore omitted. Let **K** be the set of all possible keys, and store the table in an array $T[k], 0 \leq k < M$, where $M$ is the size of the table. Associated with each key $K \in \mathbf{K}$ is an *initial hash address* $i = h(K)$, where $h : \mathbf{K} \rightarrow \{0 \ldots M-1\}$. When a key is being stored in the memory this is the first location tried. If it is empty, the key is placed there. There are many possible choices of $h$; a simple one is described shortly. The general intention is to randomize the key so that the probability of two keys having the same value of $h(K)$ is no more than chance.

If the initial hash location $i$ already has a key stored in it, a collision algorithm is invoked. This either moves the incumbent key or searches elsewhere for an empty location. One result is that a key $K$ may not be stored in $T[h(K))]$ but could end up in some quite different location. Rather than getting involved immediately in the complexities of particular collision algorithms, $T$ will initially be assumed to be a two dimensional array $T[i, j], 0 \leq i < M$, where $j \geq 0$ is called the "collision number." The first key that hashes to location $i$ will be stored in $T[i, 0]$, the second in $T[i, 1]$, and so on.

## Tree representation

Now consider some tree in which each node has at most $n$ children, numbered 0 to $n - 1$. For the purposes of hash table storage, nodes are uniquely identified by keys referred to as *node keys*. These are constructed as follows:

- if a node is stored at $T[i, j]$, the node key for its $m$'th child is the triple $\langle m, i, j \rangle$;
- a special node key $\langle \text{root} \rangle$ is reserved for the root node.

Figure 1 shows a binary tree encoded in this way. For the sake of a concrete example the nodes are given one-character labels. All nodes whose label alphabetically precedes their parent's are located in its left subtree, while all whose label is greater lie to the right. The Figure shows the nodes with their labels and the position in $T[i, j]$ where they will be stored. Arrows between nodes are labeled with the node key of the child.

The tree is always accessed through the node key $\langle \text{root} \rangle$. In this case, $h(\langle \text{root} \rangle) = 3$, so the root node, labeled "d", is stored at $T[3, 0]$. It has two immediate children labeled "b" and "e" with node keys $\langle 0, 3, 0 \rangle$ and $\langle 1, 3, 0 \rangle$ respectively. The bottom half of the Figure shows the node keys stored in $T[i, j]$, together with the associated labels and the hash function $h$ whose values are randomly chosen for this example. Because $h(\langle 1, 3, 0 \rangle) = 5$, the node labeled "e" is stored at $T[5, 0]$. This in turn has two potential children with keys $\langle 0, 5, 0 \rangle$ and $\langle 1, 5, 0 \rangle$. Only the latter is actually part of the tree, and the former is not stored in $T$. The labels

do not participate in the storage scheme being described and are included solely to aid in understanding the diagram.

Because a child's node key cannot be constructed until a value has been determined for $j$ by placing its parent in the table, the structure must be built from the root up. For the same reason a subtree, once stored, cannot be moved or attached to another node without being rebuilt.

Given the position of a node in the memory—that is, the indices $i, j$—any of its children can be retrieved by constructing the appropriate node key $\langle m, i, j \rangle$, calculating its hash value, and consulting that location in the table. To find a node's parent is even easier, for if the key is $\langle m, i, j \rangle$ its parent is stored in $T[i, j]$. Consequently the representation implicitly makes parent nodes immediately accessible from their children. This supports an algorithm for tree traversal[3,4] which is iterative rather than the usual recursive method; it uses a fixed amount of storage and does not modify the tree.

## Compact hashing

Compact hashing, developed by Cleary,[5] stores keys in a one-dimensional array $T[k]$ rather than the notional two-dimensional array $T[i, j]$ used above. To resolve any conflict between keys that generate the same initial hash address, a collision algorithm based on bidirectional linear probing is employed. This is a particularly efficient version of the general open addressing method[6] which resolves collisions by calculating a sequence of new locations from the original one. Bidirectional linear probing[7] simply searches the sequence of locations immediately above and below the original. By storing nodes that hash to the same location in order of insertion, the collision number $j$ for each one can be determined by counting from the beginning of its collision group and need not be stored explicitly.

The compact hash modifies bidirectional linear probing to reduce its storage requirements significantly. Only part of each key needs to be stored in $T$—if the table is $M$ locations long, the initial hash address $i$, which occupies $\lfloor \log_2 M \rfloor$ bits, can safely be omitted.

An example will help to make this clear. Consider the case when the number of keys in $\mathbf{K}$ is less than $M$. Then every key can be assigned its own location in $T$ without possibility of collision. $T$ degenerates to an ordinary indexed array and the keys need never be stored—though a single bit may be needed to indicate whether or not a particular location is occupied. The same reasoning can be used to show that it is not necessary to hold the entire key in memory even if the key space is larger than $M$.

In general, an average of $\lceil |\mathbf{K}|/M \rceil$ keys are liable to hash to any one location in the table. Bidirectional linear probing ensures that these will be stored as a group of consecutive locations. With each location, enough bits must be stored to distinguish any given key from the other members of its collision group; we call this the "disambiguation" number. A probabilistic argument can be used to determine the maximum number

of keys liable to hash to the same location, and this dictates the number of bits allocated for disambiguation.

Special action must be taken to keep collision groups distinct. In the event that an insertion threatens to overlap another group, a space must be opened for the new node. This is done by moving the threatened group by one location and recording enough information to enable it to be found from its original position. It turns out that a minimum of two extra bits are necessary for each hash table entry. One, the "virgin" bit, is used to indicate whether or not a particular location has been hashed to. The other, the "change" bit, marks the boundaries of collision groups. Together they are used to locate groups and distinguish one group from another. Extra bits can serve to speed up the process of locating the target group, but experiments[5] indicate that no sensible further improvement is obtained when more than 5 additional bits are used at hash table densities of up to 95%.

Multiple collisions can potentially push nodes above or below the physical limits of the table. This can easily be accommodated by making the table circular. Alternatively, additional "breathing room" can be left at the top and bottom of the table—the number of locations required can be determined by probabilistic arguments, and is very small in practice (on the order of 20 locations).[5]

## Bonsai trees

We now apply compact hashing to the tree storage method presented earlier. All the components are now in place to build the tree. The structure of a node is described in the next subsection, and following that we give a suitable hash function for a Bonsai tree.

### Size of nodes

For each node, fields must be included for the disambiguation number, the virgin bit, the change bit, and the data associated with the node—for example, a character label in the case of a text trie. To determine a suitable size for the disambiguation field, we proceed as follows.

Recall that most of the information in a key is implied by the table index $i$ to which the key was originally hashed. (Note that the node may not actually be stored in this location, because the collision algorithm can move nodes around.) The disambiguation number must encode everything in the key which is in excess of this table index. In our application, the key must identify the node's parent $T[i, j]$, and its own sibling number $m$. This information is contained in the triple $\langle m, i, j \rangle$. This triple is randomized to a number which is then decomposed into two components: the initial hash address, $i$, and the disambiguation number, whose size is equal to that of the $m$ and $j$ fields combined.

The upshot of this is that the disambiguation field is $\lceil \log_2 n \rceil$ bits larger than that introduced above for generic hashing, since the key must now also encode the sibling number $m$. As before, $n$ is the maximum

number of children a node can have. In the case of our binary tree example, $n = 2$ so one extra bit suffices.

The size of the collision number $j$ is potentially unbounded. However, in practice it can be restricted to 4 bits. The argument for this is probabilistic. It has been shown[8] that there exist hash functions which guarantee that almost all sets of keys will be distributed randomly through the hash table. The density of keys stored in the memory is $\rho = N/M$ where $M$ is the number of available locations and $N$ is the number of items stored. The probability that exactly $r$ keys will share some particular initial hash address is

$$e^{-\rho}\frac{\rho^r}{r!}.$$

$\rho$ will always be less than one, and we assume that it is at most 0.8. The probability $\epsilon$ that there will be any location with more than $k$ keys hashing to it is bounded by

$$\epsilon \leq Me^{-0.8}\sum_{r=k+1}^{\infty}\frac{0.8^r}{r!}.$$

Letting $k = 16$ (for a 4 bit $j$ field), $\epsilon \leq 7 \times 10^{-17}M$, and even if $M$ is as large as $10^{10}$ this yields an error probability of $3 \times 10^{-7}$, which is acceptable in many applications. If this is not sufficient let $k = 32$ (5 bit $j$ field) so that $\epsilon \leq 3 \times 10^{-41}M$, giving for practical values of $M$ an error probability that is almost certainly less than the probability of a hardware failure before the tree is constructed. Figure 2 shows the error probability as a fraction of memory size at different hash table densities and for different numbers of $j$ bits. For example, with a tree of $10^9$ nodes at virtually 100% table occupancy—even with just one free location remaining—a 7-bit disambiguation number reduces the chance of a failed insertion to 1 in $10^{208}$!

In summary, we use 2 bits for the virgin and change bits, 4 bits for $j$, and $\lceil \log_2 n \rceil$ bits for $m$. Of these, $j$ and $m$ are stored together as the disambiguation field. Each node key will therefore occupy $6 + \lceil \log_2 n \rceil$ bits in the hash table. For our binary tree example $n = 2$, giving 5 bits for the disambiguation field and a total of 7 bits per node for the compact hash mechanism—for any size of tree.

## Invertible hash function

To store a node key $\langle m, i, j \rangle$ in the hash table it is necessary to construct two numbers: the initial hash address and the disambiguation number. This must be done in a way that allows $\langle m, i, j \rangle$ to be reconstructed from the two numbers together. What follows is a practical guide to how to do this.

The first step is to pack the integers $m$, $i$ and $j$ into a single integer, $c$, using the field sizes:

$$c = (m \times \text{sizeof}(j) + j) \times \text{sizeof}(i) + i = (m \times 2^4 + j) \times M + i.$$

If $M$ is a power of 2, this can be done efficiently by storing $m : j : i$ as bit fields in a machine word. The result $c$ ranges between 0 and $c_{max} - 1$, where $c_{max} = 16nM$. However, this does not allow for the special node

key $\langle \text{root} \rangle$. In practice, it is convenient to allow for more than one tree—and thus more than one root—to be stored in the hash memory. There can be at most $M$ distinct root nodes which can be assigned keys ranging from $16nM$ to $16nM + M - 1$, which increases $c_{max}$ to $(16n + 1)M$.

We use $c$ as the numeric value of the node key $\langle m, i, j \rangle$. The next step is to randomize it and split it into two parts, the index and disambiguation value. Let

$$c' = (c \times a) \bmod p.$$

If $p$ is a prime greater than the maximum possible value of $c$, $a$ an integer between 1 and $p$, and $a^{(p-1)/q} \neq 0 \bmod p$ for all primes $q$ that divide $p - 1$, then $c'$ will be random and the step will be invertible.[9] A suitable value for $a$ can be found by choosing an initial guess, testing that $a^x \bmod p$ is not 1 for all $x$ of the form $(p - 1)/q$ for some prime $q$, and if necessary incrementing the guessed value until this condition is met. In practice, $2p/3$ is a good starting point, and generally few increments are necessary. Table 1 shows suitable randomization values for the case $n = 3$ which will be used below for the Bonsai structure. Here, $p$ is the smallest prime greater than $c_{max} = (16n+1)M = 49M$, and in all cases less than 8 increments were needed to find $a$ from $2p/3$.

Since $c'$ is now random, a suitable hash function is simply $c' \bmod M$, and the corresponding disambiguation value is $\lfloor c'/M \rfloor$. If $M$ is a power of 2, these are available as bit fields.

To reconstruct $\langle m, i, j \rangle$, each of these steps can be inverted. Given a hash table index $i$ and a disambiguation value $b$, first compute

$$c' = b \times M + i;$$

then

$$c = (a^{-1} \times c') \bmod p, \quad \text{where } a^{-1} = a^{p-2} \bmod p.$$

If $c$ is greater than or equal to $16nM$, then the original key must have been the root numbered $c - 16nM$. Otherwise, the values of $m$, $i$ and $j$ are retrieved by

$$\begin{aligned} m &= c \operatorname{div} 16M, \\ i &= c \bmod M, \\ j &= (c \bmod 16M) \operatorname{div} M. \end{aligned}$$

In practice, the physical table is somewhat larger than $T$ to leave breathing room at the top and bottom of the hash table as mentioned earlier. In programming languages such as C this is conveniently done by explicitly adding to $i$ (or subtracting from it) a small constant, say 20, when determining $c$ (or $c'$).

Another practical consideration is the size of the integers involved in the multiplications and divisions that compute $c'$ from $c$ and vice versa. For typical values (e.g. $M > 1024$ in Table 1), a long integer multiplication

of 32 by 32 bits is needed, and a 64-bit product is not directly supported by many contemporary machine architectures. Software emulation of these extended arithmetic operations can substantially increase the time taken to store and retrieve nodes in a Bonsai.

## Comparison with conventional tree representations

The Bonsai method is particularly well suited to the storage of large trees and tries. Such structures occur in a number of applications, including those that involve the compression[1] and prediction[2] of text. This section compares the storage requirements of three different ways of representing tries, with and without the use of compact hashing. The results are summarized in Table 2.

The usual method of representing trees is to include with each node two or more pointers to its children. Though other linear representations are possible,[10] they do not permit the leaves of the tree to be both retrieved and inserted in a time proportional to its depth. There is an important distinction between the way the tree is arranged and the internal structure of the nodes themselves. The best arrangement to use depends on the structure chosen for the nodes. The next subsection introduces three possible arrangements and evaluates their space requirements using conventional pointers. The following one discusses the application of compact hashing to each of the three methods. Finally, we compare the space requirements for pointers and Bonsai trees in an actual application.

### Conventional tree representations

Figure 3 shows three ways of arranging a trie: a multi-way tree, a structure we call a "binary trie," and a linked list representation. The numbers on the left show the corresponding levels in the trie. The first representation has a "root" level because node labels are not stored since they are implicit in the pointers; in the other representations the labels are explicitly stored at the nodes.

In the multi-way tree of Figure 3a, each node contains $n$ pointers together with the data associated with the node. This gives a total of $np + d$ bits per node where $p$ is the number of bits per pointer and $d$ is the number of bits for the data (Table 2). To retrieve on a single key takes one pointer dereference per level in the tree. A tree traversal requires $n$ pointer dereferences per node in the tree (including interior nodes). If $n$ is large this representation is very wasteful of space—entirely in the form of null pointers.

The second representation, the binary trie, uses three pointers per node. Two are used to construct a binary tree of the children of a given tree node. The third pointer indicates the next level of the tree. In addition each node needs a label of $\lceil \log_2 n \rceil$ bits (the character in Figure 3b) to indicate which child within a particular level it is. This gives a total of $3p + d + \lceil \log_2 n \rceil$ bits per node. If there are $c$ children for a particular node, on the order of $\log_2 c$ memory accesses are needed per level to retrieve a key. Three memory accesses are needed

per node when traversing the tree.

The third representation uses a linked list at each level instead of a binary tree, giving $2p + d + \lceil \log_2 n \rceil$ bits per node. The search time is increased to $\frac{1}{2}c$ accesses per level. For a traversal, two accesses are needed per node.

Consider the storage of natural-language text in a trie structure using ASCII characters. The first method (multi-way tree) is extremely space intensive and wasteful as most of the 128 pointers per node are null. The second (binary trie) with three pointers per node is much more space-efficient, but the third (linked-list) with just two pointers generally improves upon it since there are usually few descendents per node on average. Frequency ordering can be used to improve search efficiency whenever the last method is used.

If memory space is at a premium, the size of pointers in a conventional representation can be minimized, at the cost of a little address arithmetic, by storing nodes in a "dense array." Instead of storing full pointers, dense arrays store a $\lceil \log_2 M \rceil$-bit node index, where $M$, as before, is the number of locations available for tree nodes. For example, if $2^{16}$ nodes are allocated for the tree, only 2 bytes are needed per pointer versus perhaps 4 for a standard machine pointer. Pointers are easily recreated by multiplying the stored index by the node size and adding this value to the address of the start of the array.

## Using the Bonsai structure

The Bonsai version of the multi-way tree in Figure 3a requires $d$ bits for data and $6 + \lceil \log_2 n \rceil$ for key information. Allowing for a maximum 80% hash table occupancy, this gives a total of $\frac{5}{4}(6 + d + \lceil \log_2 n \rceil)$ bits per node. This is much smaller than the pointer form (except when $n$ is small and $d$ very large). Each pointer dereference translates to one retrieval of a node key from the hash memory, which in practice is about an order of magnitude slower than a pointer dereference. This difference in speed tends to be masked by the other overheads of the search algorithm. However, the difference shows more markedly during a full traversal of the tree, when much of the time is spent checking null entries at the leaves.

For the binary trie (Figure 3b), the Bonsai representation needs $6 + \lceil \log_2 3 \rceil = 8$ bits for the tree structure and $d + \lceil \log_2 n \rceil$ bits for the data and label, for a total of $\frac{5}{4}(8 + d + \lceil \log_2 n \rceil)$ bits. This is a good choice for a Bonsai tree as it occupies essentially the same amount of memory per node as the previous representation and does not require excessive memory accesses for traversal or retrieval.

For the linked-list representation (Figure 3c), the compact form consumes $6 + \lceil \log_2 2 \rceil = 7$ bits together with $d + \lceil \log_2 n \rceil$ for the data and label for a total of $\frac{5}{4}(7 + d + \lceil \log_2 n \rceil)$ bits, almost the same as above. There is no point in using this because little space is saved for potentially large increases in retrieval time.

A weakness of the Bonsai technique is that null pointers, like all other pointers, are not explicitly stored, making it expensive to check for strings that have not yet occurred in the text. To circumvent this, up to $n$ extra "null-pointer" bits can be included with each node in any of the three representations. This may not

incur a space penalty because byte or word boundary alignments may leave unused bits—particularly in the second and third arrangements when only one or two extra bits are required.

## Storage comparison in a practical application

An example of a system that can benefit from the Bonsai method is the REACTIVE KEYBOARD,[11] a device that accelerates typewritten communication with a computer system by predicting what the user is going to type next, on the basis of already-entered text. It stores this text in a trie which can occupy millions of nodes, and some tree traversal is done to reduce frequency counts of entries in sub-trees of alternative predictions.

Here we analyze the amount of space used by three different storage techniques: the conventional pointer representation, a dense array, and a Bonsai. In the first two cases we choose the most space-efficient storage arrangement; in the third we spend an extra bit per node to speed access to alternative predictions. As discussed above, this is a two-pointer linked list for the first two and a three-pointer binary trie for the third. The results are summarized in Table 3.

For the sake of comparison we first define a Bonsai node and then design a dense array node that occupies an equivalent amount of storage. Three bytes are necessary for the former (see below), and allowing for an 80% hash table occupancy and rounding to a byte boundary gives 4-byte nodes as a target for the dense array. The conventional representation is simply based on a 32-bit address space and needs no further elaboration—nodes occupy 10 bytes each. In all cases the same data is stored at each node: a 7-bit ASCII character and a 7-bit frequency count.

For the Bonsai, the binary trie arrangement was used as a compromise to avoid the high cost of searching associated with a linked list. This requires three pointers per node (see Figure 3b), and consequently $\lceil \log_2 n \rceil = 2$. In order to speed traversal of the subtree of alternative predictions for a given context, which is needed by the application to collect and update frequency statistics, two null-pointer bits were added to each node indicating whether or not the left and right siblings are present. These obviated the need to search for non-existent entries, thereby halving the number of accesses needed for a traversal. The last column of Table 3 shows the number of bits required for a Bonsai node, a total of 24 bits (3 bytes). Note the two extra null-pointer bits come for free because of byte alignment.

The dense array is stored in the linked list arrangement of Figure 3c and each node has two pointers: one to the next alternative at its level and the other to the next higher level's list of continuations. In our case, with nodes containing 7 bit labels and 7 bit counts, two 9 bit dense array indices fit exactly into the target node size of 4 bytes. As the number of nodes is increased, the size of the pointers, and therefore the nodes, must grow accordingly.

Figure 4 plots the number of bytes per node against the maximum number of nodes that can be stored, for all three methods. The number of bytes/node for a Bonsai depends on the maximum allowable hash table

occupancy, and the grey shows the range from 80% to 100% occupancy (top to bottom). The conventional representation is constant at 10 bytes/node but cannot accommodate more than $2^{29}$ nodes. Not shown beyond the left side of the figure is the fact that, for memory sizes of 32 nodes or less(!), the dense array is more space-efficient than the Bonsai (assuming 80% occupancy). For larger memories, Bonsai nodes are significantly smaller than dense array nodes.

## Summary

In terms of space occupied, the Bonsai method improves significantly over conventional representations when storing large trees—it reduces the space occupied by pointers to only $6 + \lceil \log_2 n \rceil$ bits per node where $n$ is the maximum number of children a node can have. However, it does not permit reclaimation of freed nodes (except by rebuilding entire subtrees) and it is somewhat tricky to design and implement. Although its successful operation depends on probabilistic assumptions, it is possible to make the probability of failed insertion so small as to be completely negligible. Insertion and retrieval operations are inevitably slower than in simpler representations, but they can be speeded up considerably by utilizing a small number of additional bits per node.

In typical applications that store large trie structures derived from natural language text, a Bonsai can reduce the total storage requirement to just over a third of what would otherwise have been needed (30 bits vs 80 bits). In these applications an occasional failed insertion can be tolerated. Our experience is that the method performs quickly enough for real-time interactive use on a VAX-11/780.

One practical point is that when it comes to the actual implementation of the hash function, a long integer multiplication is needed (typically 32 by 32 bits for present-day memory sizes), and a 64-bit product is not supported by some contemporary machine architectures. Although early versions of the Reactive Keyboard system used compact hashing to increase their effective storage capability, the versions currently being distributed do not because of migration from the Vax architecture to Suns and personal computers.

## Acknowledgements

This research has been supported by the Natural Sciences and Engineering Research Council of Canada and by the Alberta Heritage Foundation for Medical Research.

## References

1. T.C. Bell, J.G. Cleary and I.H. Witten. *Text compression.* Prentice Hall. Englewood Cliffs, NJ 1990.

2. J.J. Darragh, I.H. Witten and M.L. James. "The Reactive Keyboard: a predictive typing aid" *IEEE Computer 23*(11): 41–49 (November 1990).

3. J.G. Cleary and J.J. Darragh. "A fast compact representation of trees using hash tables" Research Report 83/162/20. Department of Computer Science, University of Calgary (1984).

4. J.J. Darragh. *Adaptive predictive text generation and the Reactive Keyboard.* MSc Thesis. Department of Computer Science, University of Calgary, Calgary, Alberta (1988).

5. J.G. Cleary. "Compact hash tables using bidirectional linear probing" *IEEE Trans Computers C-33*(9): 828–834 (September 1984).

6. T.A. Standish. *Data structure techniques.* Addison Wesley. Reading, Massachusetts (1980).

7. O. Amble and D.E. Knuth. "Ordered hash tables" *Computer J 17*(2): 135–142 (1974).

8. J.L. Carter and M.N. Wegman. "Universal classes of hash functions" *J Computer Systems Science 18*: 143–154 (1979).

9. D.E. Knuth. *The art of computer programming Vol 3: sorting and searching.* Addison Wesley. Reading, MA (1973).

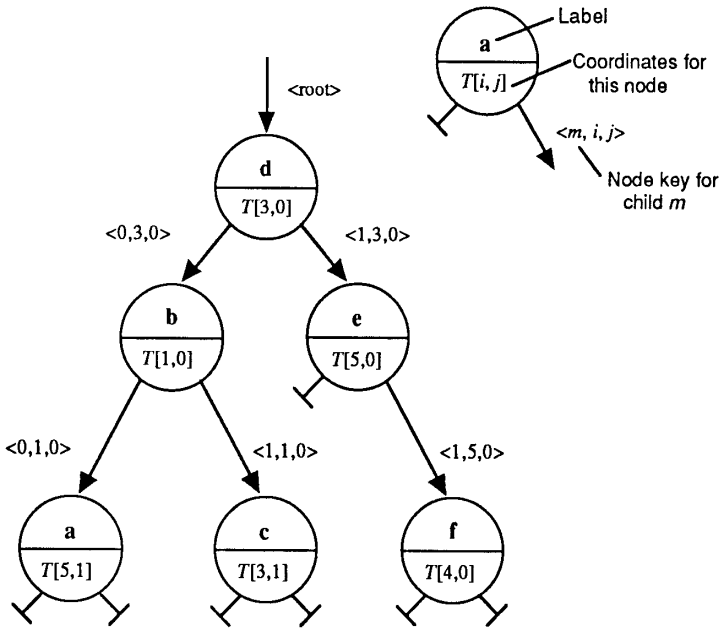
10. F.M. Liang. *Word hy-phen-a-tion by com-put-er.* PhD Thesis. Computer Science Department, Stanford University, CA (1983).

11. J.J. Darragh and I.H. Witten. "Adaptive predictive text generation and the Reactive Keyboard" *Interacting with Computers 3*(1): 27–50 (April 1991).

## List of Figures



## List of Tables




**Hash table**

| $T[i, j]$ | | Hash table index $i$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| $j$ | 0 | — | b<0,3,0> | — | d<root> | f<1,5,0> | e<1,3,0> |
| | 1 | — | — | — | c<1,1,0> | — | a<0,1,0> |

**Hash function**

| $K$ | d<root> | b<0,3,0> | e<1,3,0> | a<0,1,0> | c<1,1,0> | f<1,5,0> |
|---|---|---|---|---|---|---|
| $h(K)$ | 3 | 1 | 5 | 5 | 3 | 4 |

Figure 1 Binary tree using hashed representation

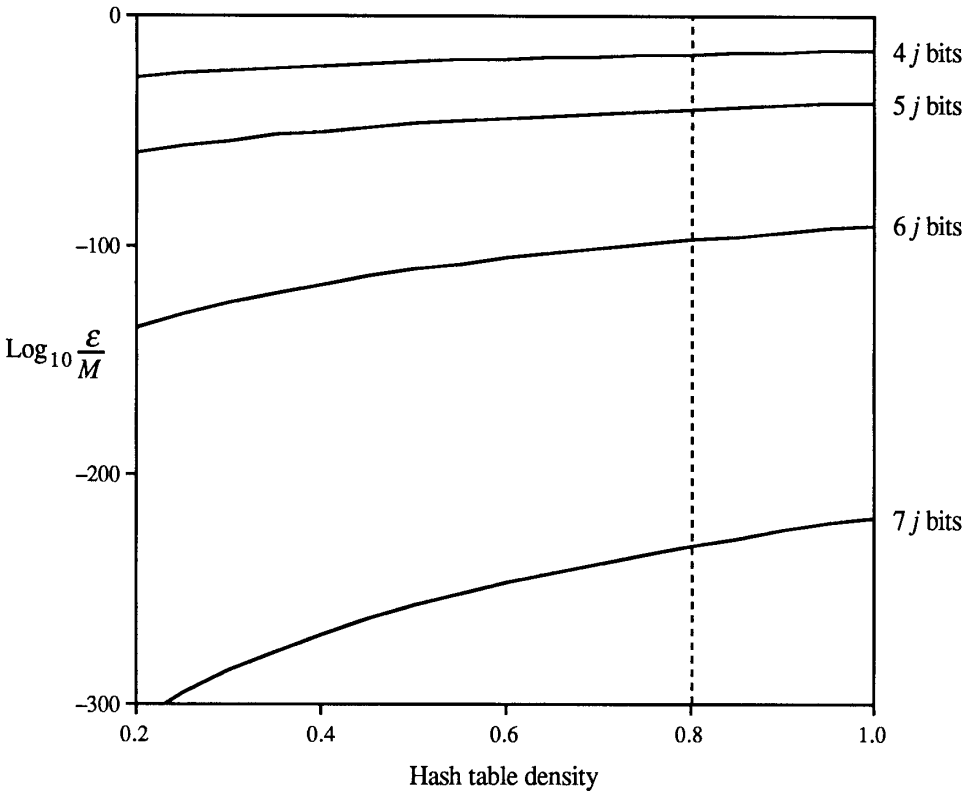



Figure 2 Error probability at different hash table densities

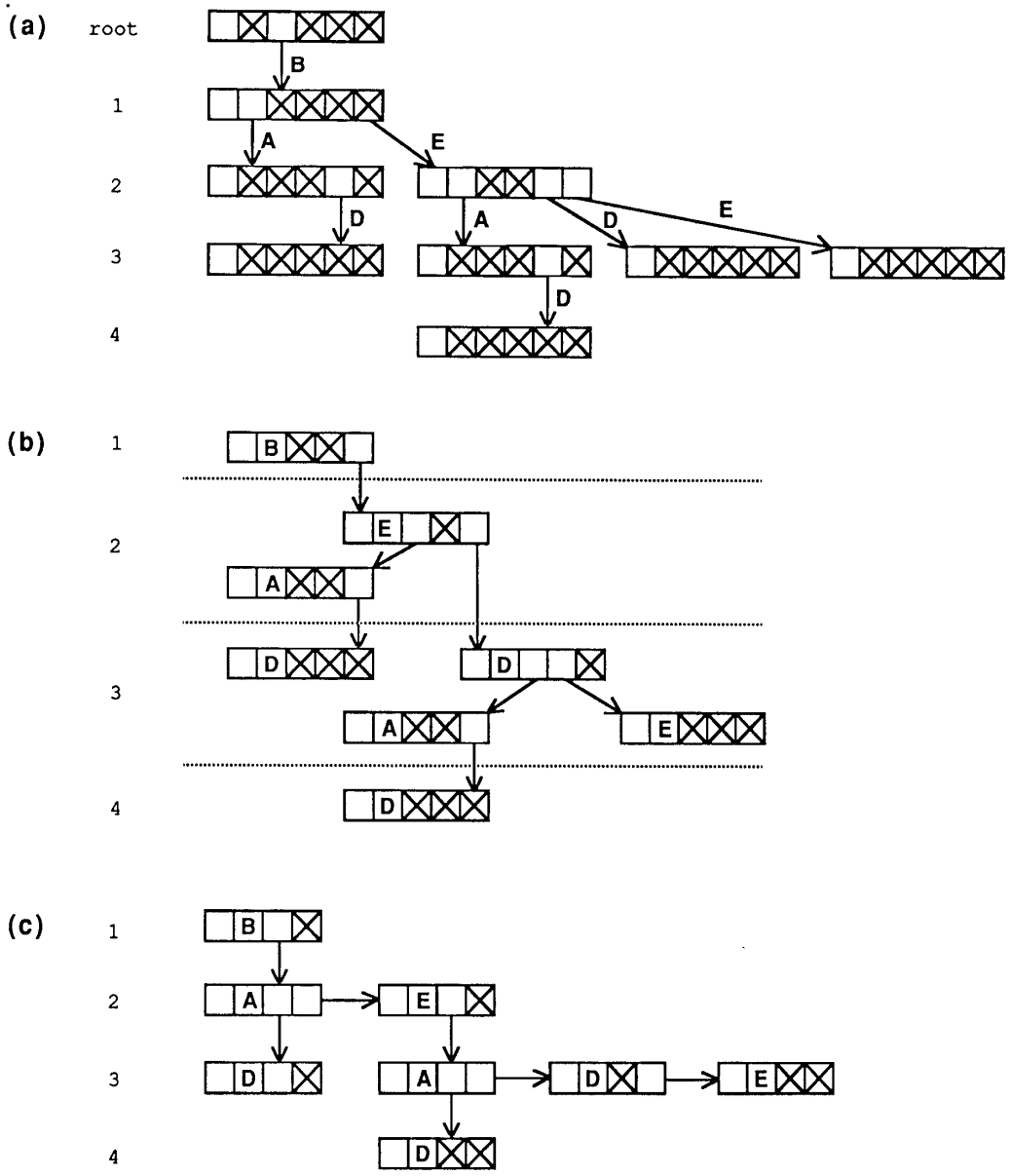



Figure 3 Three tree data structures
(a) Multi-way tree
(b) Binary trie
(c) Linked list

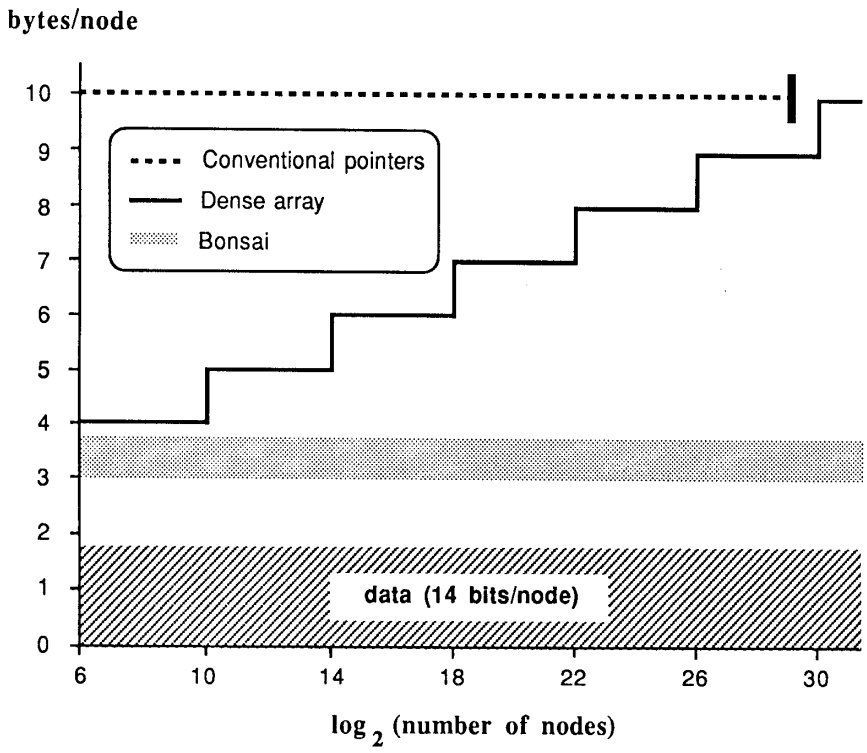



Figure 4 Comparison of node sizes for the three pointer storage methods

| Maximum number of nodes $M$ | Prime $p$ | Muliplier $a$ | Multiplier inverse $a^{-1}$ |
|---|---|---|---|
| 512 | 25097 | 16730 | 18822 |
| 1024 | 50177 | 33450 | 37632 |
| 2048 | 100357 | 66911 | 15846 |
| 4096 | 200713 | 133812 | 180642 |
| 8192 | 401411 | 267606 | 100352 |
| 16384 | 802829 | 535218 | 602121 |
| 32768 | 1605677 | 1070450 | 1204257 |
| 65536 | 3211279 | 2140856 | 963384 |
| 131072 | 6422531 | 4281687 | 6422528 |
| 262144 | 12845069 | 8563378 | 9633801 |
| 524288 | 25690121 | 17126746 | 19267590 |
| 1048576 | 51380233 | 34253493 | 43475582 |
| 2097152 | 102760453 | 68506968 | 51380225 |
| 4194304 | 205520911 | 137013941 | 3 |
| 8388608 | 411041831 | 274027886 | 102760457 |
| 16777216 | 822083597 | 548055730 | 616562697 |

Table 1 Suitable randomization values for a binary trie

| Tree | Bits per node (pointer) | Bits per node (Bonsai) | Memory accesses per level for key retrieval | Memory accesses per node for a traversal |
|---|---|---|---|---|
| Multi-way | $np + d$ | $\frac{5}{4}(6 + d + \lceil \log_2 n \rceil)$ | 1 | $n$ |
| Binary trie | $3p + d + \lceil \log_2 n \rceil$ | $\frac{5}{4}(8 + d + \lceil \log_2 n \rceil)$ | $\log_2 c$ | 3 |
| Linked list | $2p + d + \lceil \log_2 n \rceil$ | $\frac{5}{4}(7 + d + \lceil \log_2 n \rceil)$ | $\frac{1}{2}c$ | 2 |

$p$ — bits in a pointer
$d$ — bits in a data field
$n$ — maximum children per tree node
$c$ — actual number of children for a tree node

Table 2 Space requirements for three tree data structures

| Field | Conventional pointers | Dense array | Bonsai |
|---|---|---|---|
| Data bits | | | |
| ASCII character | 7 | 7 | 7 |
| frequency count | 7 | 7 | 7 |
| Overhead bits | | | |
| virgin bit | 0 | 0 | 1 |
| change bit | 0 | 0 | 1 |
| Disambiguation bits | | | |
| $j$ | 0 | 0 | 4 |
| $\lceil \log_2 n \rceil$ | 0 | 0 | 2 |
| Pointer bits | | | |
| null-pointer bits | 0 | 0 | 2 |
| index bits | 0 | $2 \times 9$ | 0 |
| pointer bits | $2 \times 32$ | 0 | 0 |
| Total bits | 78 | 32 | 24 |
| Node size (rounded) | 10 bytes | 4 bytes | 3 bytes |
| Maximum nodes | $2^{29}$ | $2^9$ | $\infty$ |

Table 3 Calculation of node sizes for the three pointer storage methods