# Family Trees

## An ordered dictionary with optimal congestion, locality, degree, and search time

Kevin C. Zatloukal * Nicholas J. A. Harvey *

**Abstract**

We consider the problem of storing an ordered dictionary data structure over a distributed set of nodes. In contrast to traditional sequential data structures, distributed data structures should ideally have low congestion. We present a novel randomized data structure, called a *family tree*, to solve this problem. A family tree has optimal expected congestion, uses only a constant amount of state per node, and supports searches and node insertion/deletion in expected $O(\log n)$ time on a system with $n$ nodes. Furthermore, a family tree supports keys from any ordered domain. Because the keys are not hashed, searches have good locality in the sense that intermediate nodes on the search path have keys that are not far outside of the range between the source and destination.

## 1 Introduction

While data structures have long been used to organize data on an individual computer system, the past decade has seen significant work on distributed data structures for organizing nodes and data in a distributed system. This work has proceeded chiefly in three directions: peer-to-peer overlay networks, Scalable Distributed Data Structures, and compact routing structures. This paper describes family trees, which are distributed data structures suitable for use as a peer-to-peer overlay and are an advancement on the existing work in this area.

Peer-to-peer overlay networks are structures for organizing nodes, routing traffic, and searching for data in a distributed system. A primary objective of peer-to-peer overlays is that no node bear a disproportionate amount of work, implying that the overlay's structure should have low congestion. Overlay networks have numerous practical applications, such such as multicast communication schemes [17] and distributed caching mechanisms [8, 19].

Overlays such as CAN [16], Chord [18], Viceroy [15], and Koorde [9] use hashing to distribute nodes and data uniformly in a numeric space. Thus, using the consistent hashing approach [10], they can support a distributed hash table. Each overlay node maintains "pointers" to a set of other nodes, where each pointer is typically just a physical network address. Any node may route a message to any target node by using the appropriate pointers to forward the message along a sequence of intermediate nodes. These overlays differ primarily in their scheme for selecting routing pointers. The schemes used by all of these overlays [16, 18, 15, 9] achieve routing in $O(\log n)$ time, assuming each pointer can be traversed in unit time.[1]

Most existing peer-to-peer overlays require $\Theta(\log n)$ routing pointers per node in order to achieve this routing performance. Viceroy [15] and Koorde [9] are the notable exceptions — they achieve $O(\log n)$ hops with only $O(1)$ pointers per node. Having few routing pointers offers a significant practical benefit since correspondingly less maintenance traffic is required to check integrity of the overlay structure. Alternatively, an overlay with fewer routing pointers may check those pointers more frequently while using the same overall bandwidth.

Skip Graphs [1], SkipNet [6], and their variants [7, 2] are peer-to-peer overlays, based on skip lists, that support *ordered* keys. Ordered overlays have significant practical advantages over distributed hash tables. First, they can take advantage of locality in search requests. For example, a search from `a.mit.edu` for `p.mit.edu` will not require contacting any nodes outside of `mit.edu`. This property does not necessarily hold for distributed hash tables. Second, Skip Graphs [1] and SkipNet [6] support range query operations. For example, they could be used to broadcast a message to all nodes in `mit.edu`. However, these ordered overlays all require $\Theta(\log n)$ pointers per node. Prior to this paper, no existing overlay has used $O(1)$ pointers without requiring that the keys be hashed.

Scalable Distributed Data Structures (SDDSs) were first proposed by Litwin et al. [13] as a means to dynamically distribute buckets of data among the nodes of a

*MIT Computer Science and Artificial Intelligence Laboratory. Email: `{kevinz,nickh}@mit.edu`. This permutation of the authors' names was chosen uniformly at random in order to ensure fair treatment of the alphabetically challenged.

[1] We will use $n$ to denote the number of nodes throughout this paper.

distributed system and to perform efficient search operations across those nodes. Numerous SDDS structures have been proposed including distributed hash tables [13] and ordered, distributed dictionaries [14, 11, 3]. Unlike peer-to-peer overlays, SDDSs are not intended for use as a routing structure; accordingly, SDDSs do not focus on congestion. Family trees could conceivably be used as the basis for an SDDS, but we do not consider such an extension in this paper.

Compact routing structures [4, 5] have significant differences from SDDSs and peer-to-peer overlays. First, the latter two assume connectivity between any pair of nodes (typically via some underlying routing scheme), whereas compact routing structures do not. Second, compact routing structures typically do not allow updates (i.e., node insertions or deletions). Lastly, compact routing structures typically must assign new identifiers to the nodes for routing purposes. Due to these differences, compact routing structures typically can not be used for implementing ordered dictionaries, so we do not discuss them further.

In this paper we address the previously unsolved problem of designing an ordered, distributed dictionary with constant degree, $O(\log n/n)$ congestion, and $O(\log n)$ performance for search and update operations. We present a randomized solution to this problem, family trees, which achieve the update, search, and congestion bounds in expectation and use only nine pointers per node. This work is an improvement over existing distributed data structures which are either not ordered (and hence have poor locality) [15, 9], do not support low-congestion routing [14, 3], or require $\Theta(\log n)$ routing pointers per node [1, 6, 7, 2].

The remainder of this paper is organized as follows. Section 2 describes family trees and proves some of their important properties. Section 3 presents the algorithms for searching in a family tree and proves bounds on performance. Section 4 gives algorithms for inserting and removing nodes from a family tree. Section 5 concludes the paper.

## 2 Family Trees

A family tree is a data structure for organizing machines or resources in a distributed system. Family trees combine the techniques of Viceroy [15] and SkipNet [6] in a novel manner. Like Viceroy, and the butterfly network [12] on which it is based, nodes are separated into approximately $\lg n$ levels; nodes at level $i$ will have pointers to nodes approximately $2^i \lg n$ positions away in the ordered list of keys. We generate these pointers by separating the nodes at level $i$ randomly into $2^i$ separate ordered lists, in a manner similar to SkipNet. The resulting data structure has a natural analogy to a genealogical family tree, as will be made clear shortly. Figure 1 shows an example instance of a family tree.

**2.1 Definitions.** We will now formally define the properties of each node in the data structure. Each node has a *name ID*, which is an arbitrary element from some ordered domain. This is the normal key for looking up a node in the dictionary. If X is a node, we will denote its name ID by X.NameID. Each node also has a *numeric ID*, denoted X.NumID, which is an infinite sequence of random bits, each bit chosen uniformly at random. Equivalently, the numeric ID is a random real number in $[0, 1)$. As explained in Section 3.2, nodes can also be looked up by their numeric IDs. Thus, by using consistent hashing [10], family trees support a distributed hash table.

Obviously, each node does not generate an infinite sequence of random bits. Each node only needs to generate as many bits as necessary to distinguish its numeric ID from the others. The following proposition bounds the number of numeric ID bits that will need to be generated.

Proposition 2.1. *Every node only generates $O(\lg n)$ numeric ID bits with high probability.*[2]

*Proof.* Let X and Y be nodes. The probability that X and Y choose the first $(c + 2)\lg n$ bits the same is $1/2^{(c+2)\lg n} = 1/n^{c+2}$. Thus, the probability that any node chooses the first $(c + 2)\lg n$ bits the same as X is $(n - 1)/n^{c+2} < 1/n^{c+1}$, and the probability that any node needs more than $(c + 2)\lg n$ bits is less than $n/n^{c+1} = 1/n^c$.

Next, each node X has a *level*, denoted X.Level, which determines the approximate distance, within the ordering by name ID, advanced by the node's level list pointers. The level number is chosen at random from $\{0, \ldots, \lfloor \lg n_0 \rfloor - 1\}$, where $n_0$ is an estimate of $n$. We use an estimate for two reasons. First, it is not clear that $n$ can be precisely computed in a manner that is efficient in both time and congestion. Second, it is helpful that different nodes produce different estimates of $n$ so that nodes are not all updated to the presence of a new level simultaneously: if we computed $n$ exactly, then every node would need to update its level when $n$ reached the next power of 2.

We will discuss the method of estimating $n$ in Section 2.2. In the remainder of this subsection, we will define all of the pointers on a node. These pointers are

[2]Throughout this paper, we say that $X$ is "$O(f(n))$ with high probability" to mean that there exists an $\alpha > 0$ such that $\Pr(X > c\alpha f(n)) < 1/n^c$ for any $c \geq 1$ and for sufficiently large $n$.

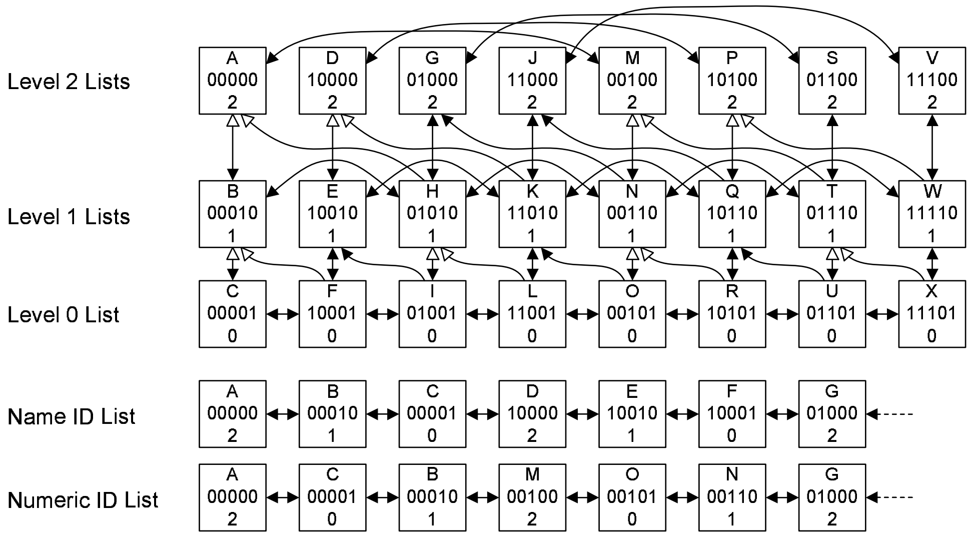


Figure 1: *An example family tree with 24 nodes. Each node, denoted by a square, has a name ID (i.e., key), numeric ID, and level. Level 0 nodes are connected into a single list. The level 1 nodes are divided into two disjoint, interleaved lists: nodes whose numeric IDs start with a 0, and nodes whose numeric IDs start with a 1. At level 2, there are four lists: nodes join the list given by the first two bits of their numeric ID. All of these level lists are sorted by the name IDs of the nodes. Nodes also maintain pointers to their parents, one level higher: white pointers denote a mother, and black pointers denote a father. Nodes also point to their first child, one level lower. Lastly, all nodes belong to a list sorted by name IDs and a list sorted by numeric IDs.*

the only other data associated with each node beyond its name ID, numeric ID, and level.

In a family tree, each node has nine pointers. The first six pointers link each node into three circularly linked lists. The NamePrev and NameNext pointers link all nodes into a list sorted by name ID. The NumPrev and NumNext pointers link all nodes into a list sorted by numeric ID. As mentioned above, the data structure separates the nodes into levels and, within level $i$, into $2^i$ separate lists. Each node chooses its list according to the first $i$ bits of its numeric ID. The LevelPrev and LevelNext pointers link the nodes that have the same level $i$ and have the same first $i$ bits in their numeric IDs into a list sorted by name ID.

Additionally, there are pointers between levels. If X is a node, then X.Mother points into the list of nodes with level equal to X.Level$+1$, with numeric ID matching the first X.Level bits of X.NumID, and with (X.Level $+ 1$)-th numeric ID bit equal to 0. Of these nodes, X.Mother points to the node whose name ID is closest but *less than* X.NameID, or it points to nil if no such node exists. X.Father is defined identically except its (X.Level $+ 1$)-th bit must be 1.

Lastly, X.FirstChild points into the list of nodes in level X.Level $- 1$ whose numeric IDs match the first X.Level $- 1$ bits of X.NumID. Of these nodes, X.FirstChild points to the node whose name ID is closest but *greater than* X.NameID, or it points to nil if no such node exists.

For example, consider the node with name ID "N" in Figure 1, which we will denote N. Node N is linked into the list sorted by name ID and the list sorted by numeric ID. Since N has chosen level 1, it belongs to one of the two level 1 lists, according to the first bit of its numeric ID. This bit is 0, so N belongs to the same level 1 list as "B", "H", and "T". The first child of N is "O" because this is the next name after "N" in the sole level 0 list. Node "R" is also a child of N. Node "M" is the closest node to "N" in the level 2 list corresponding to numeric ID prefix 00, so node "M" is N.Mother. Similarly, node "G" is N.Father as it is the closest node to "N" in the level 2 list with prefix 01.

**2.2 Estimating $n$.** Each node must estimate $n$ in order to choose an appropriate level. This estimation process is identical to that of Viceroy. Let X be a node and Y be its successor in numeric ID space. (The last node in the list will take the first as its successor.) Then, regarding the numeric IDs as real numbers in $[0, 1)$, we estimate $n$ as $n_0 = 1/d(\text{X.NumID}, \text{Y.NumID})$, where $d(x, y) = y - x \mod 1$. This estimate may be substantially incorrect. However, since we only need the log of the estimate, we can show that every node estimates $\lg n_0 = \Theta(\lg n)$ with high probability.

PROPOSITION 2.2. *Every node estimates $n_0$ such that $\lfloor \lg(n/(c+2)\ln n) \rfloor \le \lfloor \lg n_0 \rfloor \le \lfloor (c+2)\lg n \rfloor$, for any constant c, with high probability.*

*Proof.* First, we will argue the lower bound. Let $d = 1/(n/(c+2)\ln n) = (c+2)\ln n/n$. For a node X to estimate $n$ smaller than $1/d$, every other node must have chosen a numeric ID outside of the range of length $d$ after X.NUMID. (If X.NUMID is near to 1, then this range will be in two pieces: one at the end of $[0, 1)$ and one at the beginning.) The probability that X estimates $n$ this small is $(1-d)^{n-1} = (1-d)^n/(1-d) < n(1-d)^n < n\exp(-dn) = n\exp(-n(c+2)\ln n/n) < n\exp(-(c+2)\ln n) = 1/n^{c+1}$. Thus, the probability that any node estimates $n$ this small is less than $n/n^{c+1} = 1/n^c$.

Now, we argue the upper bound. Let $d = 1/n^{c+2}$. Fix a node X. The probability that some other node Y has a numeric ID within the range of length $d$ after X.NUMID is $d$. Thus, the probability that X estimated $n > 1/d$ is at most $(n-1)d = (n-1)/n^{c+2} < 1/n^{c+1}$. Thus, the probability that any node estimated $n > 1/d$ is less than $n/n^{c+1} = 1/n^c$.

We have shown that the desired bounds hold without the floors. Taking the floor of the estimate and both bounds can only increase the probability that the bounds hold, so the proof is complete.

An advantage of this method for estimating $n$ is that each node's estimate is only dependent on the distance to its successor in the numeric ID list. Thus, each node needs to change its estimate only when its successor changes, which will make insertions and deletions efficient. We will discuss this aspect further in Section 4.

A notable disadvantage of this method is that it makes the levels dependent on the numeric IDs. Furthermore, the levels themselves are not independent. If X has a level of $i$, then we know that there is no other node within a range of $1/2^i$ after it. This means that some other node estimated $n_0 < (n-1)/(1-1/2^i)$ and thus has a level at most $\lfloor \lg((n-1)/(1-1/2^i)) \rfloor - 1$. Clearly, this is a small amount of dependence: we have only shown that some other node has a level slightly less than $E[\lfloor \lg n_0 \rfloor - 1]$. However, even this small amount of dependence complicates analysis. The issue of dependence also exists for Viceroy, although the issue appears to be somewhat more involved for the analysis of family trees.

To handle this difficulty, we use the following approach. Proposition 2.2 shows that all estimates of $n$ are reasonable with high probability. Thus $\Pr(\text{X.LEVEL} = i \mid \text{X estimated } n \text{ reasonably})$ is within a constant factor of $1/\lg n$. This yields a bound on $\Pr(\text{X.LEVEL} = i)$ as follows. In general, if $E$ is some event that holds with high probability and $F$ is any other event, we can bound $\Pr(F)$ using $\Pr(F|E)$ and an additional error term. Specifically, we have $|\Pr(F) - \Pr(F|E)| \le 1/n^c$. As long our probability and expectation bounds introduce no more than $n^\alpha$ such error terms, for some fixed $\alpha$, we can use $\Pr(F)$ and $\Pr(F|E)$ interchangeably while only introducing an error term of $1/n^{c-\alpha-1}$. By choosing suitably large $c$, this error term disappears in the $O$-notation. In the analysis below, we can assume that all nodes estimate $n$ within the bounds of Proposition 2.2 whenever convenient because we will not need to introduce more than $O(n)$ error terms. Thus, we will ignore error terms in the proofs below in order to keep the explanations clear.

To simplify notation, we will let the $c$ in Proposition 2.2 be fixed and define the following notation for the upper and lower bounds.

DEFINITION 2.1. *The minimum level estimate is $L_{min} = \lfloor \lg(n/(c+2)\ln n) \rfloor$ and the maximum level estimate is $L_{max} = \lfloor (c+2)\lg n \rfloor$.*

**2.3 Global Properties.** Now that all data belonging to a node has been defined, we examine the global properties of family trees.

PROPOSITION 2.3. *If all name IDs, numeric IDs, and levels are given, then the family tree has only one possible shape.*

*Proof.* Since we do not allow duplicates, the name ID list has only one possible shape. Similarly, the numeric ID list has only one possible shape. Each node belongs to exactly one level list according to its level and the relevant bits of its numeric ID. Each such list is sorted, so they can have only one shape. Lastly, the MOTHER, FATHER, and FIRSTCHILD pointers are completely determined by the shape of the level lists.

COROLLARY 2.1. *The probability that a family tree has a given shape is independent of history.*

*Proof.* By Proposition 2.3, the current shape of the data structure depends only on the name IDs, numeric IDs, and levels of the items currently in the dictionary. The name IDs and numeric IDs themselves are clearly independent of history. As long as each node chose its level by estimating $n$ using its current successor (not a past successor), then the probability distribution of the levels is independent of history. We will ensure that this is true by having a node choose a new level whenever its successor changes.

Next, we turn to properties of family trees that will be useful in analyzing the performance of search and update operations. For all operations, it is necessary to bound the number of levels in the data structure since it may be necessary to visit all of them. We can bound this very tightly.

THEOREM 2.1. *The data structure has no more than* $(c+2)\lg n$ *levels with high probability.*

*Proof.* By Proposition 2.2, no node estimated $\lg n_0$ larger than this. Hence, no node could have chosen a level larger than this.

Since the nodes at level $L$ are separated randomly into $2^L$ lists, the expected length of these lists is less than $n/2^L L_{\min}$. (As mentioned above, we are ignoring the error term of size $1/n^{c-1}$ caused by dependence of levels and numeric IDs.) Straightforward Chernoff bounds show that the length of these lists does not deviate much from its expectation, except at very high levels. We now consider the probability of encountering empty lists at all levels slightly less than $L_{\min}$.

THEOREM 2.2. *Let* $L = \lg n - 2\lg\lg n - 2\lg(c+2)$. *Every list in the family tree with level at most* $L$ *is non-empty with high probability.*

*Proof.* The probability that a list at level $k$ is empty is at most $(1-1/(2^k(c+2)\lg n))^n$. For lists at level at most $L$, this bound is at most $(1-(c+2)\lg n/n)^n < e^{-(c+2)\lg n} = 1/n^{c+2}$. Next, observe that the total number of lists in levels 0 to $L$ is at most $2^{L+1} < 2n$. Thus, applying a union bound over all lists yields the desired result.

The preceding discussions focused on properties of the levels and their lists. Next, we will look at properties that hold for arbitrary individual nodes in a family tree. Let X be a node in a family tree. Both X.FIRSTCHILD and X.LEVELNEXT.FIRSTCHILD point to nodes in the same list (at level X.LEVEL $-$ 1). Consider the number of nodes in this list whose name IDs fall between X.NAMEID and X.LEVELNEXT.NAMEID. This is an important concept, so we give it a name.

DEFINITION 2.2. *Let* X *be a node. Let* $\mathcal{L}$ *be the level list containing* X.FIRSTCHILD. *We denote by* CHILDREN(X) *the sublist of* $\mathcal{L}$ *containing all nodes* Y *such that* X.NAMEID < Y.NAMEID *and* Y.NAMEID < X.LEVELNEXT.NAMEID.

Intuitively, we would expect every node to have about two children since the lower level list is about twice as long. The following theorem shows that the expected length is indeed a constant.

THEOREM 2.3. *For any node* X, |CHILDREN(X)| *is* $O(1)$ *in expectation and* $O(\lg n)$ *with high probability.*

*Proof.* Consider the process of traversing the name ID list, starting at X, and choosing a numeric ID and level for each of the nodes. Let $\mathcal{C}$ denote the list at level X.LEVEL $-$ 1 containing X's children. Our goal is to count the number of nodes that are chosen to belong to $\mathcal{C}$ before choosing X's successor in its level list. This process is a sequence of trials with three outcomes: on a "success", we have chosen X's successor; on a "failure", we have added a new node to $\mathcal{C}$; otherwise, we have a "retry", indicating an unrelated node. Equivalently, we can eliminate the "retry" outcome by thinking of each trial as continuing until the first success or failure. Since there are now only two possible outcomes, it is a Bernoulli trial. Call a success in the Bernoulli trial a "heads" and a failure in the Bernoulli trial a "tails". The remainder of the proof bounds the number of tails before the first heads.

First, we must bound the probabilities of success and failure in the three-outcome trial. Let $k =$ X.LEVEL $-$ 1. For the probability of success (finding X's successor in its level list), we have

$$\begin{aligned}\Pr(\text{success}) &\geq 1/2^{k+1}L_{\max} = 1/2^{k+1}(c+2)\lg n\\ \Pr(\text{success}) &\leq 1/2^{k+1}L_{\min} = 1/2^{k+1}\lg(n/(c+2)\ln n)\end{aligned}$$

For the probability of failure (finding another node in $\mathcal{C}$), we have

$$\begin{aligned}\Pr(\text{failure}) &\geq 1/2^{k}L_{\max} = 1/2^{k}(c+2)\lg n\\ \Pr(\text{failure}) &\leq 1/2^{k}L_{\min} = 1/2^{k}\lg(n/(c+2)\ln n)\end{aligned}$$

Using these, we can bound the probability of a tails. Let $\Pr(\text{failure}) = \alpha/2^k\lg n$. Then we have

$$\begin{aligned}\Pr(\text{tails}) &= \Pr(\text{failure})/(\Pr(\text{failure})+\Pr(\text{success}))\\ &< (\alpha/2^k\lg n)/(\alpha/2^k\lg n + 1/2^{k+1}(c+2)\lg n)\\ &= \alpha/(\alpha+1/2(c+2))\\ &= 1/(1+1/2(c+2)\alpha)\end{aligned}$$

To get an upper bound for Pr(tails), we apply an upper bound for $\alpha$. By definition of $\alpha$, we have $\alpha < \lg n/\lg(n/(c+2)\ln n)$. For sufficiently large $n$, we can bound $\alpha < 2$. Thus, for sufficiently large $n$, we can bound Pr(tails) by some constant $p < 1$. Then, the expected number of tails before the first heads is less than $p/(1-p) = O(1)$, one less than the expected value of a geometric distribution. This shows that the expected number of children is $O(1)$.

To complete the proof, we compute a high probability bound on the number of children. The probability that we see at least $k$ tails before a heads is $p^k$. If we let $\beta = 1/p$, then picking $k = c\log_\beta 2\lg n$ gives a probability of $1/\beta^{c\log_\beta 2\lg n} = 1/n^c$.

Definition 2.3. *For any node* X*, let* Parents(X) $=$ {Y | X $\in$ Children(Y)}.

A slight variation on the proof of Theorem 2.3 yields the following result.

Theorem 2.4. *For any node* X*,* |Parents(X)| *is* $O(1)$ *in expectation and* $O(\lg n)$ *with high probability.*

The previous theorems examined the level lists. We now focus on the name ID list and the numeric ID list, both of which contain every node. The next theorem shows that no matter where we are in the name ID or numeric ID list, there is always a nearby node that is at a given level (provided that level is not too large).

Theorem 2.5. *The distance in the name ID list (or numeric ID list) from a node* X *to the nearest node at level* $L \in \{0, \ldots, L_{min}-1\}$ *is* $O(\lg n)$ *in expectation and is* $O(\lg^2 n)$ *with high probability.*

*Proof.* Imagine traversing through the name ID list (or numeric ID list) and choosing the level of each node as we encounter it. The expected distance to a node at level $L$ is

$$
\begin{aligned}
\mathrm{E}[\text{distance}] &< \textstyle\sum_{i=1}^{\infty} i(1-1/L_{\max})^{i-1}(1/L_{\min}) \\
&= (1/L_{\min})\textstyle\sum_{i=1}^{\infty} i(1-1/L_{\max})^{i-1} \\
&= (1/L_{\min})/(1-(1-1/L_{\max}))^2 \\
&= L_{\max}^2/L_{\min} \\
&= O(\lg n)
\end{aligned}
$$

The probability that there are no level $i$ nodes within a distance of $kL_{\max}$ is less than $(1 - 1/L_{\max})^{kL_{\max}} < e^{-k}$. If we choose $k = c \ln n$, then there are no level $i$ nodes within a distance of $c(c+2)\lg n \ln n = O(\lg^2 n)$ with probability less than $1/n^c$.

Another variation on the proofs of Theorem 2.3 yields the following result.

Theorem 2.6. *Let* X *be a node at level* $L$*. The number of nodes whose name ID is between* X.NameID *and* X.LevelNext.NameID *is* $O(2^L \lg n)$ *in expectation and* $O(2^L \lg^2 n)$ *with high probability.*

## 3 Search Operations

In this section, we will describe the search operations and analyze their performance. In the next section, we will consider update operations.

**3.1 Search by Name ID.** The Search-By-Name-ID operation searches from a start node to find the node whose NameID is closest but less than or equal to a given destination name. Figure 2 contains the pseudocode for this operation. For the sake of simplicity, we have assumed that the destination name is greater than the start node's name. The other case is similar.

```
Search-By-Name-ID( start , dest )
  X ← Linear-Search-For-Level-0( start , dest )
  X ← Find-Closest-At-Level-0( X , dest )
  X ← Linear-Search-For-Destination( X , dest )
  return X
Linear-Search-For-Level-0( X , dest )
  while X.NameNext ≠ nil and
        X.NameNext.NameID < dest and
        X.Level > 0
  do X ← X.NameNext
  return X
Find-Closest-At-Level-0( X , dest )
  left ← X.NameID
  while true
  do if X.LevelNext = nil or
        dest < X.LevelNext.NameID
       then break
     P ← X.Mother or X.Father at random
     if P ≠ nil
       then X ← P
       else break
     X ← Level-Search-By-Name-ID ( X , left )
  while true
  do if X.Level > 0
       then X ← X.FirstChild
       else break
     X ← Level-Search-By-Name-ID ( X , dest )
  return X
Level-Search-By-Name-ID( X , name )
  while X.LevelNext ≠ nil and
        X.LevelNext.NameID < name
  do X ← X.LevelNext
  while X.LevelPrev ≠ nil and
        name < X.NameID
  do X ← X.LevelPrev
  return X
Linear-Search-For-Destination( X , dest )
  while X.NameNext ≠ nil and
        X.NameNext.NameID < dest
  do X ← X.NameNext
  return X
```

Figure 2: Search-By-Name-ID *finds the node whose name ID is closest to the given destination.*

The algorithm works in four phases. It is important for search locality that the second phase begin at a level 0 node. The purpose of the first phase is to find a level 0 node near the start node. This is accomplished in Linear-Search-For-Level-0 by a linear search in the name ID list. The second and third phases are implemented in the helper function Find-Closest-At-

LEVEL-0. In the second phase, lines 10–19, we move upward until we reach a node X, to the left of where we began, such that *dest* is between X and its successor in that level, or until we reach the top. Line 19 maintains the invariant that X is the closest node to the left of where the second phase began (*left*). The third phase, lines 20–24, is symmetric to the second phase except that we are centered around *dest* and moving downward. This third phase is analogous to a binary search or a skip list search. The node returned from FIND-CLOSEST-AT-LEVEL-0 will be the level 0 node closest to the destination, but it may not be the closest node overall since the destination node may not be at level 0. In the fourth phase, implemented in LINEAR-SEARCH-FOR-DESTINATION, we search through the name ID list to find the node closest to the destination. The correctness of the algorithm is assured by this last phase.

Next, we will analyze the running time. Theorem 2.5 shows that the first phase requires $O(\lg n)$ time in expectation and $O(\lg^2 n)$ time with high probability. Since the number of levels is $O(\lg n)$ with high probability, the outer loops of the second and third phases will execute no more than $O(\lg n)$ times. Each iteration in the second phase does $O(1)$ work except for the call to LEVEL-SEARCH-BY-NAME-ID. Note that each node traversed in this call is a parent of the successor of the node reached by the previous iteration. Theorem 2.4 shows that the number of such parents is $O(1)$ in expectation and $O(\lg n)$ with high probability. Each iteration of the third phase is analogous except that we traverse children instead of parents. Thus the total running time of the second and third phases is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability.[3] The analysis of the fourth phase is identical to that of the first. Thus, the running time of SEARCH-BY-NAME-ID is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability.

**3.1.1 Locality.** In this section, we show that SEARCH-BY-NAME-ID has good locality, in the sense that it does not traverse nodes that are far outside the range between the start and destination.

Before doing so, we note that the pseudocode that we presented in Figure 2 is sequential. A distributed implementation would execute various portions of that code at different nodes. As a practical matter, distributed nodes would cache the name IDs of their neighboring nodes. Using this cache, the distributed search algorithm can determine that a node is beyond the destination without accessing it. We will assume that such caching occurs for our discussion of locality and, later, congestion.

Next, recall that the first phase searches from the start node for the closest level 0 node. To see why this first phase is important for locality, suppose that the start node is at a high level and that the destination name is very close to the start node's name. In this case, the start node's LEVELNEXT and FIRSTCHILD pointers are both likely to point well beyond the destination node, so traversing them would result in poor search locality. Instead, the first phase finds a node at level 0 (so that the expected distance to its successor in the level list is as small as possible).

We will now analyze the expected locality of SEARCH-BY-NAME-ID. To begin, note that the first and fourth phases have *strict* locality, in the sense that they never traverse a node whose name ID is smaller than *start* or greater than *dest*. In the second phase, we may follow a parent pointer to a node whose name ID is smaller than *start*. We want to show that the expected maximum distance from any such node to *start* is $O(D)$, where $D$ is the distance from *start* to *dest*. Similarly, in the third phase, we may follow a FIRSTCHILD pointer to a node whose name ID is greater than *dest*. The analysis of the third phase is symmetric to that of the second.

First, by Theorem 2.6, the expected distance from the leftmost node traversed at level $j$ to *start* is $O(2^j \lg n)$. We can bound the expected maximum distance of all nodes traversed up through level $j$ by the expected sum of these distances. Since these distances increase geometrically, this sum is also $O(2^j \lg n)$. Next, we can condition the expected maximum distance on the highest level reached, which will be $j$ for the previous calculation. Define $h$ to be the smallest value such that $D \leq 2^h \lg n$. Intuitively, we would expect that the highest level reached is $h$ or higher. A short calculation that shows that the probability that the highest level reached is $h+i$ is at most $1/2^{i(i-1)/2}$. Since these probabilities decrease exponentially faster than the conditioned maximum distances increase, the expected maximum distance is $O(D)$.

In comparison, searches in a Skip Graph / SkipNet [1, 6] have strict locality. It is possible to adapt our SEARCH-BY-NAME-ID algorithm to achieve strict locality. To do this, we would add parent pointers that point right and child pointers that point left. These pointers would allow the search to remain between the start and destination nodes during the second and third phases. (The first and fourth phases already have strict locality.) Due to space constraints, we do not consider this variant any further.

[3] There is considerable slack in this argument. It can be shown that these phases run in $O(\lg n)$ time with high probability. Unfortunately, the tighter bound for these phases does not improve the bound of the algorithm as a whole.

```
Search-By-Numeric-ID( start , bits )
37  name ← start . NameID
38  X ← Find-Closest-At-Level-0( start , name )
39  while true
40  do if X.Level = | bits |
41        then return X
42     if bits [ X.Level ] = 0
43        then P ← X.Mother
44        else  P ← X.Father
45     if P = nil
46        then break
47      X ← Level-Search-By-Name-ID( P , name )
48  X ← Linear-Search-By-Num-ID( X , bits , | bits |)
49  if | bits | < ∞
50     then X ←Level-Search-By-Name-ID( X , name )
51  return X
```

Figure 3: *The* SEARCH-BY-NUMERIC-ID *function finds the node whose numeric ID is closest to the given value.*

**3.2 Search by Numeric ID.** The previous subsection considered how to find a node with a given name ID. In this subsection, we show that it is also possible to find the node whose numeric ID is closest to a given number. This can be used to implement consistent hashing and, consequently, a distributed hash table.

The SEARCH-BY-NUMERIC-ID function searches from a start node X to find the node whose numeric ID is closest but less than or equal to the given value. This value can be either a finite or an infinite sequence of bits. If the input is a finite sequence of $k$ bits, the function returns a node in the level $k$ list corresponding to the given bits. In general, this level list will contain more than one node, so we will return the node whose name ID is closest to the name of the start node. Figure 3 contains the pseudocode for this operation. Again, for the sake of simplicity, we have assumed that the target node is to the right of the start node.

Conceptually, this algorithm is the complement of SEARCH-BY-NAME-ID. Instead of moving to a high level and then back down, this algorithm first searches for a level 0 node and then moves up. The first phase is implemented in the call to FIND-CLOSEST-AT-LEVEL-0 at line 38. In the second phase, lines 39–47, we start with a level 0 node and in each iteration find a node that matches the given value in one additional bit. Within each level, we find the node whose name is closest to the name of the start node. As with SEARCH-BY-NAME-ID, the first two phases will find a node whose numeric ID is close but is not guaranteed to be the closest to the given value. (In particular, this could happen because an intermediate level list is empty.) In the third phase, lines 48–50, we perform a linear search in the numeric ID list to find the closest node, which ensures correctness.

It remains to analyze the running time of this algorithm. The last subsection showed that the time required by FIND-CLOSEST-AT-LEVEL-0 is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability. The second phase, lines 39–47, is identical to the second phase of SEARCH-BY-NAME-ID except that the parent choices are given. Thus, the same bound on the running time applies: $O(\lg n)$ in expectation, and $O(\lg^2 n)$ with high is probability. The running time of the last phase depends on the level reached in the second phase. By Theorem 2.2, we will reach *at least* level $k = \lg n - 2\lg\lg n - 2\lg(c+2)$ with high probability. This leaves a range of size $1/2^k = \lg^2 n \cdot \lg^2(c+2)/n$ in numeric ID space to be searched. Hence, a loose bound on the running time of the linear search is $O(\lg^2 n)$ in expectation and $O(\lg^3 n)$ with high probability. By conditioning the expectation on the level reached in the second phase, we can tighten this bound to $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability. We omit the details of this analysis due to space constraints. Thus, SEARCH-BY-NUMERIC-ID requires $O(\lg n)$ time in expectation and $O(\lg^2 n)$ time with high probability.

**3.3 Congestion.** We define the *congestion at node* X to be the probability that a search operation with source S and target T, chosen uniformly at random, traverses X. For example, the congestion at the root node of a balanced binary tree is at least $(\frac{n}{2})^2/\binom{n}{2} = \Theta(1)$. Congestion of $\Theta(\lg n/n)$ is optimal when the nodes have constant degree because $\Theta(\lg n)$ nodes must be traversed in most search paths. The theorem below shows that we achieve the optimal bound.

The definition of congestion at a node is a probability where the unknown random variables are the numeric IDs and levels of all nodes in the data structure, the random bits used in the search itself, and the choice of S and T. If we imagine exposing the random bits used by every node in the data structure, then we could look at the *congestion of the family tree*, which is defined to be the maximum congestion at any node. Note that this is a probability on the unexposed random variables. The following theorem shows that the congestion of a family tree does not deviate much from the congestion at an arbitrary node.

THEOREM 3.1. *The congestion at any particular node in a family tree is $O(\lg n/n)$. The congestion of the family tree is $O(\lg^2 n/n)$ with high probability.*

*Proof.* We will analyze SEARCH-BY-NAME-ID. The analysis for SEARCH-BY-NUMERIC-ID is nearly identical. Let X be any node in the family tree. X could be traversed in any of the four phases of the search algorithm. We will consider each in turn and show that, in each, the probability that X is traversed is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability[4].

To be traversed in the first phase, X must lie between S and the closest level 0 node to the right of S. Theorem 2.5 showed that this distance is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability. Thus, the probability that X lies between a randomly chosen S and its closest level 0 node is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability.

For the node X to be traversed in the second phase (upward search), both of the following conditions must hold. First, the X.Level random parent choices must match the corresponding bits of X's numeric ID. This occurs with probability $1/2^{\text{X.Level}}$. Second, S must be between X and X.LevelNext.FirstChild. Since all nodes traversed in the upward search are before S, we will not traverse X if S is before it.[5] If S is after X.LevelNext.FirstChild, then the closest node to S in the previous level is to the right of X.LevelNext, so the parent of that node will be X.LevelNext or further to the right. We can use two applications of Theorem 2.6 to bound the number of nodes between X and X.LevelNext.FirstChild as $O(2^{\text{X.Level}} \lg n)$. Thus, the probability that a randomly chosen S is in this set is $O(2^{\text{X.Level}} \lg n/n)$ in expectation and $O(2^{\text{X.Level}} \lg^2 n/n)$ with high probability. Since these two conditions are independent, we can multiply them to show that the probability that X is traversed in the second phase is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability.

The analysis of the third phase (downward search) is symmetric to that of the second, and the analysis of the fourth phase is identical to that of the first. Adding together the congestion due to each phase, we obtain a bound of $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability.

## 4 Update Operations

In this section, we describe the insert and delete operations and analyze their performance. Analogous congestion bounds follow immediately from these definitions.

**4.1 Insert.** Most of the work required to insert a node is accomplished by calls to the search operations described in Section 3. Finding the predecessor of the new node in the name ID list is a simple matter of calling Search-By-Name-ID. Once this predecessor has been found, linking the node into this doubly-linked list requires updating four pointers. Inserting the new node into the numeric ID list is identical except that the call is instead made to Search-By-Numeric-ID. After this, it remains to set the level list pointers and the inter-level pointers.

In order to choose a level for the new node X, we must first compute the estimate $\ell \approx \lfloor \lg n \rfloor$. To do this, we subtract the numeric IDs of X and its successor (mod 1) and find the first non-zero bit. This will be found before the $(c \lg n)$-th bit with high probability. Next, we choose X.Level uniformly at random from $\{0, \ldots, \ell - 1\}$. Once X has a level, we perform a Search-By-Numeric-ID, using just the first X.Level bits of X.NumID, to find the predecessor of X in its level list. Similarly, we can find X.FirstChild by performing a Search-By-Numeric-ID using just the first X.Level $-$ 1 bits of the numeric ID. We then enumerate all the children of X and update their appropriate parent pointers to point to X. We handle X.Mother and X.Father similarly.

Lastly, we turn to the predecessor of X in the numeric ID list. As mentioned in the proof of Corollary 2.1, we must allow this node to re-estimate $n$ and choose a new level. This ensures that the shape of the family tree is independent of history. Note that the predecessor's numeric ID does not change, only its level does, so the re-estimation process does not cascade to other nodes. The procedure for estimating $n$ and choosing a level was described in the previous paragraph. The procedure for removing this node from its old level is described in the next subsection.

It is easy to see that the running time of Insert is dominated by the six calls to search operations. We argued above that computing the estimate of $\lg n$ takes $O(\lg n)$ time with high probability. The only other work is updating the pointers. There are only nine outbound pointers. The number of inbound pointers is $O(\lg n)$ with high probability by Theorem 2.3 and Theorem 2.4. Thus, the total time required is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability.

**4.2 Delete.** The algorithm for deleting a node X is straightforward. First, we enumerate the nodes with pointers to X and update them to point to the appropriate predecessor or successor. As argued in the previous section, the number of pointers to X is $O(\lg n)$ with high probability. Next, we must allow the predecessor of X in the numeric ID list to re-estimate $n$ and choose a new level. As mentioned earlier, the

[4]By "in expectation" and "with high probability", we are referring to the outcome when the structure of the family tree is revealed. The expected value of the probability is identical to the probability when no information is known, which is our definition of congestion.

[5]We ignore the case where S is before X but the closest level 0 node is after X. This is counterbalanced by the analogous case where X.LevelNext.FirstChild is substituted for X, which we do include in the probability.

predecessor's numeric ID does not change, only its level does, so the re-estimation process does not cascade to other nodes. The re-estimation procedure was described in the previous section. The running time of this part, and of the algorithm as a whole, is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability.

## 5 Conclusion

We have presented the family tree, a new dictionary data structure suited to implementation in a distributed environment. Each node in a family tree has only nine pointers. A family tree can be searched and updated in expected $O(\log n)$ time. Searches and updates both ensure that no node bears more than an $O(\log n/n)$ fraction of the traffic in expectation. These results are optimal for any data structure with $O(1)$ pointers per node. Furthermore, searches in a family tree can take advantage of locality in the input, only searching nodes that have names close to the source and target.

Family trees can be used to implement a peer-to-peer overlay network, as well as a distributed hash table. Like Skip Graphs [1] and SkipNet [6], family trees are ordered by keys from an arbitrary domain. Hence they support efficient range queries and have useful locality properties. Like Viceroy [15] and Koorde [9], a family tree's bounds on degree, congestion, and search and update performance are optimal. Family trees are the first data structure to simultaneously support arbitrary key ordering and achieve these optimal bounds.

A family tree could also be used to implement an in-memory dictionary. This could be advantageous in a situation where the dictionary is being accessed by multiple concurrent readers and writers. Our congestion bounds imply that such an implementation would have low lock contention, assuming uniform access patterns. Furthermore, the family tree could take advantage of locality in the searches made by a given client.

In summary, family trees are a new contribution to the theory of data structures. We have shown that they are optimal in congestion, locality, degree, and search and update performance. Furthermore, family trees have immediate practical applications, including to the implementation of peer-to-peer overlay networks.


## Acknowledgments

We thank John Dunagan for many helpful discussions on this topic and for reviewing drafts of this paper.


## References


[1] J. Aspnes and G. Shah. Skip Graphs. In *14th ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2003.

[2] B. Awerbuch and C. Scheideler. The Hyperring: A Low-Congestion Deterministic Data Structure for Distributed Environments. In *17th International Symposium on Distributed Computing*, Oct. 2003.

[3] P. Bozanis and Y. Manolopolous. DSL: Accomodating Skip Lists in the SDDS Model. In *Workshop on Distributed Data and Structures*, June 2000.

[4] L. Cowen. Compact Routing with Minimum Stretch. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 1999.

[5] C. Gavoille and D. Peleg. Compact and Localized Distributed Data Structures. Technical Report 1261-01, LaBRI, Université Bordeaux I, Aug. 2001.

[6] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[7] N. J. A. Harvey and J. I. Munro. Deterministic SkipNet. In *ACM Symposium on Principles of Distributed Computing*, July 2003.

[8] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*. ACM, July 2002.

[9] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[10] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[11] B. Kroll and P. Widmayer. Balanced distributed search trees do not exist. In *Workshop on Algorithms and Data Structures*, 1995.

[12] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman, 1992.

[13] W. Litwin, M. A. Neimat, and D. A. Schneider. LH* - Linear hashing for distributed files. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[14] W. Litwin, M. A. Neimat, and D. A. Schneider. RP* - A family of order-preserving scalable distributed data structures. In *Conf. on Very Large Data Bases*, 1994.

[15] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*. ACM, July 2002.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[17] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, 2001.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[19] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, July 2002.