Hashed Patricia Trie: Efficient Longest Prefix Matching in Peer-to-Peer Systems

Sebastian Kniesburges and Christian Scheideler

Department of
Computer Science
University of Paderborn
D-33102 Paderborn
Germany
seppel@upb.de,scheideler@upb.de

Abstract. The design of efficient search structures for peer-to-peer systems has attracted a lot of attention in recent years. In this paper we address the problem of longest prefix matching and present an efficient data structure called hashed Patricia trie. Our hashed Patricia trie supports *Prefixsearch(x)* and *Insert(x)* in $\mathcal{O}(\log |x|)$ hash table accesses and *Delete(x)* in $\mathcal{O}(1)$ hash table accesses when $|x|$ is the number of bits used to encode x. That is the costs only depend on $|x|$ and not the size of the data structure. The hash table accesses may be realized by any distributed hash table (DHT).

1 Introduction

In this paper we consider the problem of longest prefix matching in peer-to-peer systems. We will define the problem and present a data structure that efficiently supports longest prefix matching with the help of any common DHT, e.g. Chord [14] or Pastry [13].

Definition 1 (Longest Prefix Matching). *Given a set of binary strings S and a binary string x, find $y \in S$ such that y has the longest common prefix with x of all strings in S. The prefix of a string is a substring beginning with the first bit. If there are several keys with a longest prefix, any one of them is returned as y.*

Finding the longest prefix match from a database of keywords is an old problem with a number of applications, ranging from string matching problems and IP lookup in Internet routers to computational geometry. Another important application are range queries in distributed file-sharing applications that have become very popular with the rise of peer-to-peer systems.

In order to provide a data structure for the problem of longest prefix matching we need the following operations:

Insert(x): this adds the key x to the set S. If x already exists in S, it will not be inserted a second time.

Delete(x): this removes the key from the set S, i.e., $S := S - \{x\}$.

PrefixSearch(x): this returns a key $y \in S$ that has a longest common prefix with x.

We will present a data structure called *hashed Patricia trie* for these operations, which is based on an extension of the Patricia trie [10] over the data set embedded into a hash table. With this data structure, *PrefixSearch(x)* and *Insert(x)* can be executed in $\mathcal{O}(\log(|x|))$ hash table accesses while *Delete(x)* only needs a constant number of hash table accesses. Moreover, the number of changes in the data structure for each *Insert(x)* and *Delete(x)* operation is a constant. That is, the costs of the presented operations are independent of the size of the data structure as they only depend on the length of the input $|x|$. Our bounds on the hash table accesses imply that only a few messages have to be sent when using the well-known distributed hash table (DHT) approach in order to realize our data structure. Most importantly just $\mathcal{O}(1)$ updates in the DHT are needed per operation, so it is easy to keep the DHT synchronized. We further point out that our solution is asymptotically optimal in the use of memory space, requiring only Θ(sum of all key lengths) of memory. This implies that the data structure can easily be implemented on top of a DHT without producing an overhead in terms of memory usage.

2 Related Work

There is a large amount of literature on how to design efficient search structures in computer science (e.g., [4, 6]). A search problem closely related to the longest prefix matching is the predecessor search problem. In a predecessor search we are given a key and the problem is to find $y = \max\{z \in S \mid z \leq x\}$. In our context with strings, a natural interpretation of $z \leq x$ is that z is lexicographically smaller than x (i.e., the first letter different in x and z is lexicographically smaller in z than in x). In this case, either the closest predecessor or the closest successor of x has the longest prefix match with x.

When using standard balanced search trees to solve the longest prefix problem like in [3] or the related problem of predecessor search on $n = |S|$ keys, the worst case execution time is in $\mathcal{O}(\log n)$ due to traversing the edges of the tree. Patrascu et al. [11] gave tight bounds of $\Theta(\log l)$ for the runtime of predecessor search on a data structure in a RAM model with words of length $l = \Theta(\log n)$ (in a polynomial universe) and near linear space. The upper bound results from using the van Emde Boas data structure presented in [5]. The van Emde Boas tree comes with a specific layout limiting its use to a small range of values to be space efficient. When hashing its entries as we do it with our extended Patricia trie, then the van Emde Boas tree can be used to run *PrefixSearch(x)*, *Insert(x)* and *Delete(x)* in time $\Theta(\log |x|)$ in the worst case (w.h.p., where the probability is due to the hashing). However achieving a constant runtime for *Delete(x)* and a constant number of data structure updates for each operation would require major modifications in the van Emde Boas tree which is already a non-trivial, recursively defined data structure. We believe that our approach is much more

intuitive and simple to understand. The van Emde Boas data structure is essentially a cleverly reduced and mapped trie.

A *trie* is a search tree in which every edge represents a letter and every node represents a string consisting of the letters of the unique path from the root to that node. Thus the depth of a trie for a set of strings S is equal to the maximum length of a string in S and therefore can be much larger than the depth of a balanced search tree. In the presented solution we will use a Patricia trie, which is a compressed trie, i.e. edges do not represent letters but strings and a path in a trie can be compressed to an edge in the Patricia trie. A formal definition will be given later.

Trie hashing has been used in [7] in an approach close to the hardware level called HEXA to reduce the memory needed to store a node of a trie and its information by avoiding string pointers to the next node. This approach is validated for IP lookup in routers and string matching. Our approach follows the idea of HEXA in some points, as we only store labels of the edges and no string pointers. But since we are concentrating on a higher level by using our data structure in a distributed setting like a DHT, we focus on the number of DHT accesses instead of minimizing the computations and memory usage, although our approach is asymptotical optimal in the memory usage.

Trie hashing has also been used, for example, in file systems [8, 9], IP lookup [15] and distributed hash tables [12] in order to provide efficient longest prefix searching. While the update cost of these data structures is expensive in the worst case (i.e., linear in the size of the key), Waldvogel et al. [15] as well as Ramabhadran et al. [12] use binary search on the prefix length to obtain a runtime of $\mathcal{O}(\log |x|)$ resp. limit the number of messages to $\mathcal{O}(\log |x|)$ for *PrefixSearch(x)*.

The approach of Ramabhadran et al. called Prefix Hash Tree [12] is probably most comparable to ours. Both are based on Trie hashing and both are designed for DHT-based peer-to-peer systems. The main advantages of our hashed Patricia trie compared to the Prefix Hash Tree are the minimized update costs. Using the Prefix Hash Tree, insertions and deletions can lead to worst case costs of $\mathcal{O}(|\text{max. key length}|)$. This is due to the buckets that contain the keys. If the number of keys stored in a bucket rises beyond its maximum size the bucket is split, and it might be split up to $\mathcal{O}(|\text{max. key length}|)$ times (when all keys are distributed to the same child). The same problem occurs in case of the deletion of a key. Ramabhadran et al. present a workaround by allowing the maximum size of the buckets to be exceeded. A Patricia trie is a compressed trie and thus contains less nodes on average, so less hash table entries are needed in our solution. In a hashed Patricia trie keys of arbitrary lengths can be stored instead of a fixed length in the Prefix Hash Tree. However, notice that the Prefix Hash Tree also supports range queries, which are not considered in this paper.

A better solution for prefix search in distributed systems is due to Awerbuch and Scheideler [2]. They embed a skip graph [1] based on the entries in S into a distributed system using consistent hashing and store the addresses of the nodes for every link of the skip graph so that each link traversal in the skip graph just means a single message hop in the distributed system. While this

allows the system to execute *PrefixSearch(x)*, *Insert(x)* and *Delete(x)* in $O(\log n)$ communication rounds, the update cost when nodes join and leave the system can be much larger than in a DHT. With the hashed Patricia trie we minimize the update costs and furthermore present a data structure applicable on any kind of DHT that provides write and read commands.

3 Our Contribution: Hashed Patricia Trie

In this section we describe our hashed Patricia trie. We first review the structure of an ordinary Patricia trie, then describe its modifications, and finally show how to combine it with a hash table.

3.1 Patricia Trie

The concept of a Patricia trie was first proposed by Morrison in [10]. Suppose for the moment that all keys in the given key set K have the same length (i.e., consist of the same number of letters). In this case, the Patricia trie of these keys is a compressed trie in which every edge now represents a string (instead of just a single letter), every inner node (except for the root) has two children, and every node represents a string resulting from the concatenation of the strings along the unique path from the root to that node. See Figure 1 for an example.

In the following, we will restrict ourselves to keys representing binary strings. For every node v in the Patricia tree, let $b(v)$ be the binary string identifying it (i.e., in Figure 1, $b(k5) = 10101$). Let $|b(v)|$ be the length of $b(v)$. To maintain keys with arbitrary lengths, we also allow inner nodes with just one child in the Patricia trie, but only if such a node represents a key in the key set K (that is, node lists in the trie cannot be compressed beyond these nodes).

In our Patricia trie data structure, every node v stores $b(v)$ and the edge labels to the father and the two children, denoted by $p_{-}(v)$, $p_0(v)$ (the child whose edge label starts with 0), and $p_1(v)$ (the child whose edge label starts with 1), respectively. Node v also stores up to two keys, $key_1(v)$ and $key_2(v)$. If there is a key $k \in K$ with $k = b(v)$, then $key_1(v)$ is set to k and is otherwise empty. If there is a key $k \in K$ with prefix $b(v)$ that is longer than $b(v)$, then $key_2(v)$ is set to any one of these keys and is otherwise empty. As we will see, it is possible to make sure that every key $k \in K$ is stored at most once in $key_2(w)$ of some Patricia node w, which is important to ensure efficient updates. In order to remember this place, each node v with $b(v) \in K$ stores a string $r(v)$ with the property that for the node w with $b(w) = r(v)$, $key_2(w) = b(v)$.

It has already been shown that binary search is possible on a hashed trie [12], but to enable binary search on a hashed Patricia trie we need further nodes. For this we need some notation. Given two binary strings $a = (a_m, \cdots a_1, a_0)$ and $b = (b_m, \cdots b_1, b_0)$ (possibly filled up with leading 0s so that a and b have the same length), let $msd(a, b)$ be the unique bit position so that $a_j = b_j$ for all $j > msd(a, b)$ and $a_{msd(a,b)} \neq b_{msd(a,b)}$. So $msd(a, b)$ is the most significant bit in which the two strings differ. Now, let us replace every edge $\{v, w\}$ in the

Patricia trie by two edges $\{v, u\}$, $\{u, w\}$ where the label $b(u)$ of u is a prefix of $b(w)$. More precisely, let $\ell_1 = |b(v)|$ and $\ell_2 = |b(w)|$, and let $m = msd(\ell_1, \ell_2)$, then $b(u)$ is the prefix of $b(w)$ of length $\sum_{i=m}^{\lfloor \log \ell_2 \rfloor + 1} (\ell_2)_i \cdot 2^i$. (If u happens to have the same label as v or w, then we just merge it with that Patricia node and declare the result a Patricia node.)

As an example, assume we have two connected Patricia nodes v, w with $b(v) = 01001$ and $b(w) = 010010011$. Then $\ell_1 = |b(v)| = 5 = (101)_2$ and $\ell_2 = |b(w)| = 9 = (1001)_2$ and $m = msd(\ell_1, \ell_2) = 3$. The resulting msd-node u gets the prefix $b(u) = 01001001$ of length $8 = (1000)_2$. E.g. in Fig. 2.

The intuition for the introduction of the msd-nodes is to use the msd-nodes for the binary search. Performing a binary search on a normal Patricia trie can lead to problems. Assume in round i a prefix k with length $|k| = l_i$ is searched, i.e. *read(k)* is performed. If the Patricia trie (and the later described hash table) does not contain such a key, it is not clear whether the prefix length should be increased or decreased, as there could still be a key with a prefix longer than l_i. Therefore we introduce the msd-nodes v with the property that the length of $b(v)$ is calculated such that it will be found by the binary search.

We will call the resulting trie an *msd-Patricia trie* and the added nodes *msd-nodes* (while the original nodes are called *Patricia nodes*). In our data structure, each msd-node v stores $b(v)$ and the edge labels to the father and its child, using the variables $p_{-}(v)$ (for the father) and $p_0(v)$ or $p_1(v)$ (depending on whether the edge label to the child starts with 0 or 1). Note that the number of nodes in the *msd-Patricia Trie* is asymptotical the same as in the Patricia trie, i.e. we overload the Patricia trie only by a constant factor (one msd-node between a pair of connected Patricia nodes).

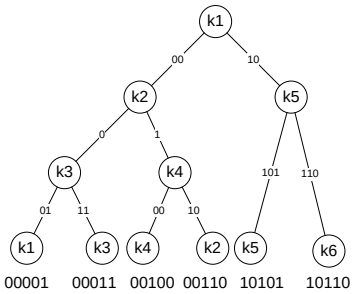

Fig. 1: A Patricia trie

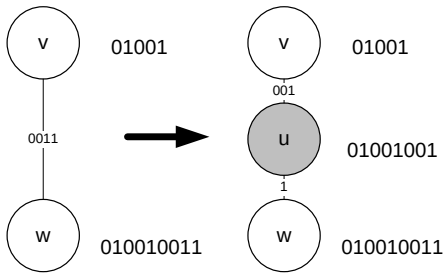

Fig. 2: A msd-node

3.2 Hashed Patricia Trie

We will now show how to combine the msd-Patricia trie with a hash table. For every node (either a Patricia node or an msd-node) its label $b(v)$ will be hashed with an appropriate hash function (see Figure 3 for an example). At this moment

we do not specify the hash function but just assume the existence of two shared memory operations: *write(key, data)* and *read(key)*, where *data* represents the data of a Patricia node or msd-node with label *key*. Using a shared memory approach allows us to directly access nodes whose labels are known instead of traversing the Patricia trie. In order to keep our approach as general as possible, we will bound the cost of our operations in terms of shared memory accesses and discuss later how to best realize the shared memory through a hash table in a distributed setting. We only focus on the communication costs since internal computations are considered to be rather cheap. More specifically, we present operations with a performance as shown in Table 4. For the memory usage the following theorem holds.

Theorem 1. *The hashed Patricia trie needs* $\Theta(\sum_{k\in K}|k|)$ *memory space, where* $\sum_{k\in K}|k|$ *is the sum of the bit lengths of the stored keys.*

Proof. Obviously it takes $\Omega(\sum_{k\in K}|k|)$ space to store all keys. But in a hashed Patricia trie not only the keys are stored, but also the inner nodes and the msd-nodes store their identifier $b(v)$ and the edge labels. We will show, that following our construction scheme the needed memory space is $\mathcal{O}(|K|)$. First, note that the Patricia trie is a binary tree with four kinds of nodes, the root r, which is an inner node with one or two children, the inner nodes V_2 with two children, the inner nodes V_1 with one child that store a $key_1(v) \in K$ and the leaves V_0 also storing keys in $key_1(v)$. Each node of V_2 has two descendants and each node of V_1 and the root one and every node except for the root has a parent, thus the total number of nodes is $2 * |V_2| + |V_1| + 2$, and the total number of nodes is also $1 + |V_2| + |V_1| + |V_0|$. Thus it holds that $|V_2| = |V_0| + 1$. Each $v \in V_2 \cup \{r\}$ stores a key in $key_2(v)$, each key is stored at most once in $key_2(v)$, and its identifier $b(v)$ satisfies $|b(v)| < |key_2(v)|$. Thus for all nodes of $V_2 \cup \{r\}$, $\mathcal{O}(\sum_{k\in K}|k|)$ space is required. For the msd-nodes we know that each msd-node is placed between every pair of connected Patricia nodes and each msd-node v has a child w with $|b(v)| < |b(w)|$. Thus the sum of all lengths of the $b(v)$s of the msd-nodes is less than the sum of all key lengths. At last note that the concatenation of the edge labels on a path from the root to a leaf corresponds to the key of the leaf, and therefore the sum of all edge labels must be less than the sum of the key lengths. Summing up $\Theta(\sum_{k\in K}|k|)$ memory space is needed.

□

3.3 Applications

We can easily implement the hashed Patricia trie on top of a DHT. When using, e.g., Chord [14] or Pastry [13] in order to realize a DHT, then the *write(key,data)* and *read(key)* operations can be realized by sending a single message along at most $\mathcal{O}(\log N)$ sites w.h.p., where N is the number of sites in the system. Hence, we obtain the following result, which improves the best previous results (e.g. in [12]). By using a underlying DHT we leave the cases of churn and load balancing up to the DHT, i.e. our hashed Patricia trie does not affect the quality of the DHT.

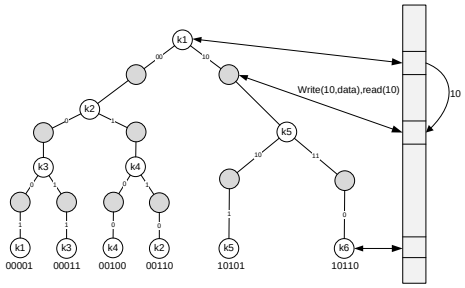

Fig. 3: A Hashed Patricia Trie

	write(key,data)	*read(key)*
PrefixSearch(x)	-	$\mathcal{O}(\log \|x\|)$
Insert(x)	$\mathcal{O}(1)$	$\mathcal{O}(\log \|x\|)$
Delete(x)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Fig. 4: Costs of hashed Patricia trie operations

Theorem 2. *When using Chord or Pastry the PrefixSearch(x) and the Insert(x) operations can be executed in $\mathcal{O}(\log |x| \log N)$ communication rounds w.h.p. The Delete(x) operation requires $\mathcal{O}(\log N)$ communication rounds w.h.p.*

4 Operations

In this section we provide algorithms for the operations *Insert(x)*, *Delete(x)* and *PrefixSearch(x)*. We will also show the correctness and bound their costs.

4.1 PrefixSearch(x)

The algorithm for the *PrefixSearch(x)* operation is based on the idea of binary searching. We assume that $x \in \{0,1\}^\ell$ has the form $(x_1, \ldots, x_\ell)$. The binary search starts with parameters $k = 0$ and $s = \lfloor \log(|x|) \rfloor$. Then in each binary search step the algorithm decides whether there is a key in the Patricia trie with a prefix $(x_1, \cdots x_{k+2^s})$. If so, then k is increased to $k + 2^s$, and otherwise k stays as it is. Also, s is decreased by 1 in any case. At last, the binary search provides some node v in the Patricia trie. From that node on, the Patricia trie is traversed downwards along the route determined by x till the first Patricia node w is found whose label is not a prefix of x, or a leaf is reached. *PrefixSearch(x)* then returns the key stored in w. As we will see, only a constant number of edges have to be traversed from v to w so that our cost bound for *PrefixSearch(x)* holds. The pseudo code for this algorithm is given below.

We will prove the correctness of *PrefixSearch(x)* with the help of two lemmas. First, we will show the correctness of the first while loop (binary search) in *PrefixSearch(x)*. Let y be a key with the longest matching prefix pre_x with x in the data structure. Let v be the last node on the route from the root to pre_x in the msd-Patricia trie. Hence, $|b(v)| \leq |pre_x|$. Let $|b(v)| = \ell$ be represented by $(\ell_m, \cdots, \ell_0)$. Thus $|b(v)|$ can also be described as $\sum_{j=0}^{m} 2^j \cdot \ell_j$. We will now prove the following lemma.

```
if key_1(read(x)) = x or key_2(read(x)) = x then
    return x;
s := ⌊log(|x|)⌋; k := 0; v := read(ε); p := p_{x_1}(v);
while s ≥ 0 do
    v := read(x_1, ··· , x_{k+2^s});
    if v ≠ ∅ then
        k := k + 2^s; p:=(x_1, ··· , x_k) ∘ p_{x_{k+1}}(v);
    else
        if (x_1, ··· x_{k+2^s}) is prefix of p then
            k := k + 2^s;
    s := s − 1;
while b(v) is prefix of x do
    if p_{x_{k+1}}(v) exists then
        k := k + |p_{x_{k+1}}(v)|; v := read(b(v) ∘ p_{x_{k+1}}(v));
    else
        if p_{x̄_{k+1}}(v) exists then
            k := k + |p_{x̄_{k+1}}(v)|; v := read(b(v) ∘ p_{x̄_{k+1}}(v));
        else
            break;
if key_1(v) ≠ nil then
    return key_1(v);
else
    return key_2(v);
```

Algorithm 1: PrefixSearch(x)

Lemma 1. *If v is the last node on the unique path from the root to pre_x with $|b(v)| = \sum_{j=0}^{m} 2^j \cdot \ell_j$ then round k of first the while loop in PrefixSearch(x) will detect the existence of a prefix of x with length $\sum_{j=\lceil \log(|x|)\rceil - k+1}^{m} 2^j \cdot \ell_j$.*

Proof. The proof is given by induction on k. Assume that $b(v) \neq \epsilon$, i.e. v is not the root of the trie. Firstly we show that the theorem holds for the basis $k = 1$. Hence the binary search needs to detect a prefix of length $2^{\lceil \log(|x|)\rceil}$. Note that $\lceil \log(|x|)\rceil \geq m \geq 0$ as $|b(v)|$ has at most $\lceil \log(|x|)\rceil$ digits. Also note here is no prefix with length $2^{m'}$ and $m' > m$ as v is the last node on the route to pre_x. If $m < \lceil \log(|x|)\rceil$ there is no prefix of length $2^{\lceil \log(|x|)\rceil}$, which corresponds to $\sum_{j=\lceil \log(|x|)\rceil - 1+1}^{m} 2^j \cdot \ell_j = 0$. If $m = \lceil \log(|x|)\rceil$, then we claim that there exists a msd-node u with $b(u)$ being a prefix of $b(v)$ and $|b(u)| = 2^m \leq |b(v)|$ on the path to v. If v is a Patricia node, this msd-node exists because there must be a pair of original nodes u', v' on the route to v with $|b(u')| < 2^m$ and $|b(v')| \geq 2^m$ as $|b(w)|$ is increasing on the route to v beginning with 0 for the root. Then $msd(u', v') = m$. The same arguments hold if v is a msd-node and $|b(v)| > 2^m$. If v is a msd-node and $|b(v)| = 2^m$, $u = v$ and u' is the parent of v and v' its child.

In the induction step we show that, if the theorem holds for any $k < \lceil \log(|x|)\rceil$, it will also hold for $k + 1$. By assuming that the theorem holds for k we know

that a prefix of length $\sum_{j=\lceil \log(|x|)\rceil -k+1}^{m} 2^j \cdot \ell_j$ was found by the binary search. If $\lceil \log(|x|)\rceil -k+1 > m$ there is no prefix of this length and $\sum_{j=\lceil \log(|x|)\rceil -k+1}^{m} 2^j \cdot \ell_j$ is 0, and so we assume from now on $\lceil \log(|x|)\rceil - k + 1 \leq m$. If $l_{k+1} = 0$ the already found prefix is also a prefix of length $\sum_{j=\lceil \log(|x|)\rceil -k+1}^{m} 2^j \cdot \ell_j$. If $l_j = 1$ and there is a node u with $b(u)$ being a prefix of $b(v)$ and thus because of pre_x and $|b(u)| = \sum_{j=\lceil \log(|x|)\rceil -k-1+1}^{m} 2^j \cdot \ell_j$ it will be found by the binary search in round $k+1$. If $\ell_j = 1$ but there is no such node u with $|b(u)| = \sum_{j=\lceil \log(|x|)\rceil -k-1+1}^{m} 2^j \cdot \ell_j$, then there cannot be a Patricia-node u' with $\sum_{j=\lceil \log(|x|)\rceil -k+1}^{m} 2^j \cdot \ell_j \leq |b(u')| < \sum_{j=\lceil \log(|x|)\rceil -k-1+1}^{m} 2^j \cdot \ell_j$, otherwise $l_{\lceil \log(|x|)\rceil -k-1}$ has to be a most significant digit for a pair of Patricia-nodes u', v', because a Patricia-node v' with $|b(v')| \geq \sum_{j=\lceil \log(|x|)\rceil -k-1+1}^{m} 2^j \cdot \ell_j$ exists. The Patricia-node v' is either on the route to v or a child of v or v itself. We know by assumption that the binary search has found the last node w with $|b(w)| = \sum_{j=\lceil \log(|x|)\rceil -k'+1}^{m} 2^j \cdot \ell_j$ $k' \leq k$, that exists in the msd-Patricia trie and set the variable p to $(x_1, \cdots x_{k'}) \circ p_{x_{k'+1}}$. Therefore $(x_1, \cdots x_{k+1})$ has to be a prefix of p and the prefix of length $\sum_{j=\lceil \log(|x|)\rceil -k-1+1}^{m} 2^j \cdot \ell_j$ is detected by the binary search. This proves the theorem. If $b(v) = \epsilon$, i.e. $|b(v)| = 0$, then $m = 0$ and v will be detected as a prefix of x in the end of the binary search. □

Lemma 2. *If v is the last node on the route to pre_x, the next Patricia node v^* storing a key y that has the longest prefix pre_x with x will be found in a constant number of edge traverses.*

Proof. If v is a msd-node, one edge will be traversed in the second while loop to reach v^*. If v is a Patricia node there will be at most two edges traversed in the second loop. So after at most two steps v^* is reached following the edges to the descendants of v that maximize the longest matching prefix. If v has no descendants, $v^* = v$ and the loop breaks. □

4.2 Insert(x)

The *Insert(x)* operation is similar to an insert in a traditional Patricia trie. This means: first, the correct position of x is searched in the trie and then x is inserted by redirecting the pointers of the affected nodes. If the trie is empty we insert a new root w with $b(w) = \epsilon$ and $key_2(w) = x$ and one child v of w with $b(v) = x$ and $key_1(v) = x$ and $r(v) = b(w)$ and add an msd-node between w and v. Otherwise, we use a slightly adapted version of the *PrefixSearch(x)* operation, that returns the Patricia node v in the trie at which the operation stops instead of the key, to search for the right position in the trie. If $key_1(v) = x$ or $key_2(v) = x$, the key is already in the data structure and will not be inserted a second time. If we know v we also know u as the parental Patricia node via the $p_{-}(v)$ edges. If x is a prefix of $b(v)$ there can be three cases. If $x = b(u)$ then $key_1(u) = x$. If $x = b(v)$ then $key_1(v) = x$. And if $|b(u)| < |x| < |b(v)|$ then a node w is inserted between u, v with $b(w) = x$ and $key_1(w) = x$ and the msd-nodes between u, w and w, v are either added or their edges are updated (the old msd-node remains

either between u and w or w and v). If $b(v)$ is a prefix of x, then v has to be a leaf and a node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a child of v. Further $r(w)$ is set to $r(v)$ and $r(v)$ is deleted and $key_2(read(r(v))) = x$ is set. Finally the msd-node between v, w is inserted. Recall that $r(v)$ returns the inner node storing $key_2(read(x)) = x$. If neither x is a prefix of $b(v)$ nor $b(v)$ a prefix of x then a new Patricia node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a sibling of v. If $p_{x_{|b(u)|+1}}$ exists, a Patricia node w' has to be inserted between u, v as the new parent of v and w. Let $j = msd(b(v), x)$, then $key_2(w') = x$ and $b(w') = x_1 \cdots x_{j-1}$. Then $r(w) = b(w')$ and the msd-nodes between u, w' w', v and w', w are added or updated. If $p_{x_{|b(u)|+1}}$ does not exist a node w with $key_1(w) = x$ and $b(w) = x$ is inserted as a child of u. If $key_2(u) = nil$ it is updated to $key_2(u) = x$ and $r(w) = b(u)$. At last the msd-node between u, w is added.

Obviously *Insert(x)* is correct if *PrefixSearch(x)* is correct. E.g. in Fig.5.

4.3 Delete(x)

We assume that the key x exists. Then we hash the key to get the corresponding nodes $v = read(x)$, $w = read(r(v))$ and u as v's parental Patricia node. If $key_2(v) \neq nil$, v is still needed after the deletion of x, so only $key_1(v) = nil$, has to be set, as there is no node w. If $key_2(v) = nil$, v can have one or no child. If there is a child w' it is connected to u, and a new msd-node between u and w' is inserted after the deletion of the msd-nodes caused by v. If v has no child, v is a leaf. In this case if $key_2(u) = nil$, so v was the only child of u, $r(u) = r(v)$ and $key_2(w) = key_1(u)$. If $key_2(u) \neq nil$, the data structure is updated by $key_2(w) = key_2(u)$ and $r(read(key_2(u))) = b(w)$ and $key_2(u) = nil$, as u is no longer necessary for the structure of the Patricia trie. If $key_1(u) \neq nil$, u is still in the data structure due to the key $key_1(u) = b(u)$. Otherwise u can be deleted by connecting the parental Patricia node and the remaining child of u. Then the msd-nodes caused by u are deleted and new msd-nodes between the parental Patricia node and the child are inserted.

Obviously *Delete(x)* only substitutes nodes with one descendant by an edge and therefore maintains the msd-Patricia trie structure correctly. E.g. in Fig.6.

4.4 Complexity Analysis

Next, we analyze the cost of the operations in terms of *write(key, data)* and *read(key)* operations, which are asymptotically upper bounding the cost of all operations if we assume that all the other primitive operations, including computing the length of a string or concatenating two strings, can be done in unit time.

Theorem 3. *The PrefixSearch(x) operation requires* $\mathcal{O}(\log(|x|))$ *read(key) operations.*

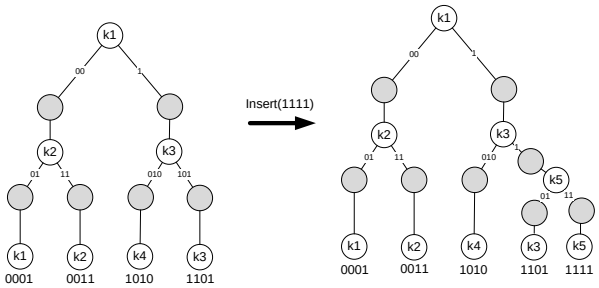

Fig. 5: Insert(x)

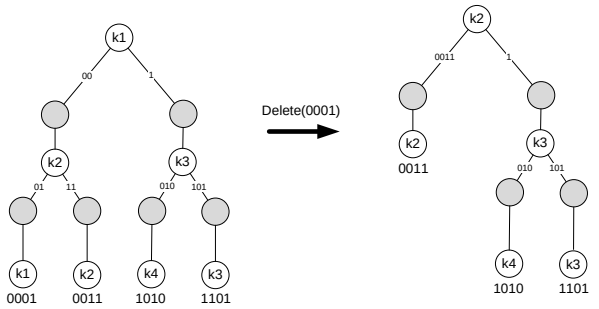

Fig. 6: Delete(x)

Proof. The first part of *PrefixSearch(x)* is the binary search that needs $O(\log(|x|))$ read operations. The second part of the *PrefixSearch(x)* needs at most two further *read(key)* operations. So altogether the theorem follows. □

Theorem 4. *The Insert(x) operation requires* $\mathcal{O}(\log(|x|))$ *read(key) operations and* $\mathcal{O}(1)$ *write(key,data) operations.*

Proof. The *Insert(x)* operation needs one *PrefixSearch(x)* with $\mathcal{O}(\log(|x|))$ *read(key)* operations, and afterwards a constant number of nodes are inserted by *write(key,data)* operations. So *Insert(x)* can be executed with $\mathcal{O}(\log(|x|))$ *read(key)* operations and $\mathcal{O}(1)$ *write(key,data)* operations. □

Theorem 5. *The Delete(x) operation requires* $\mathcal{O}(1)$ *read(key) operations and write(key,data) operations.*

Proof. The *Delete(x)* operation needs a constant number of *read(key)* and *write(key,data)* operations, as only a constant number of Patricia and msd-nodes are modified or deleted. □

5 Conclusion

This paper has investigated the longest prefix matching problem and presented an efficient data structure for this problem. The data structure combines the

advantages of Patricia tries and hash tables in order to support prefix search in worst-case logarithmic hash table accesses w.r.t. the input length, which results in an efficient DHT-based prefix search mechanism.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 384–393. Baltimore, MD, USA (12–14 Jan 2003)
2. Awerbuch, B., Scheideler, C.: Peer-to-peer systems for prefix search. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing. pp. 123–132. PODC '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/872035.872053
3. Bayer, R., Unterauer, K.: Prefix b-trees. ACM Trans. Database Syst. 2(1), 11–26 (1977)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press and McGraw-Hill (2001)
5. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory 10, 99–127 (1977)
6. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison Wesley, Reading, Massachusetts, 2nd edn. (April 1997)
7. Kumar, S., Turner, J.S., Crowley, P., Mitzenmacher, M.: Hexa: Compact data structures for faster packet processing. In: ICNP. pp. 246–255 (2007)
8. Litwin, W.: Trie hashing. In: Lien, Y.E. (ed.) SIGMOD Conference. pp. 19–29. ACM Press (1981)
9. Litwin, W., Roussopoulos, N., Lévy, G., Hong, W.: Trie hashing with controlled load. IEEE Trans. Software Eng. 17(7), 678–691 (1991)
10. Morrison, D.R.: Patricia—practical algorithm to retrieve information coded in alphanumeric. J. ACM 15(4), 514–534 (1968)
11. Patrascu, M., Thorup, M.: Time-space trade-offs for predecessor search. CoRR abs/cs/0603043 (2006)
12. Ramabhadran, S., Hellerstein, J., Ratnasamy, S., Shenker, S.: Prefix hash tree - an indexing data structure over distributed hash tables, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.617
13. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg. pp. 329–350. Middleware '01, Springer-Verlag, London, UK (2001), http://portal.acm.org/citation.cfm?id=646591.697650
14. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 149–160. ACM, New York, NY, USA (2001)
15. Waldvogel, M., Varghese, G., Turner, J., Plattner, B.: Scalable best matching prefix lookups. In: PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing. p. 312. ACM, New York, NY, USA (1998)