Routing on Longest-Matching Prefixes

Willibald Doerhinger, Member, IEEE, Günter Karjoth, and Mehdi Nassehi, Member, IEEE

Abstract—This article describes the dynamic prefix tries—a novel data structure with algorithms for insertion, deletion, and retrieval to build and maintain a dynamic database of binary keys of arbitrary length. These tries extend the concepts of compact digital (Patricia) tries to support the storage of prefixes and to guarantee retrieval times at most linear in the length of the input key irrespective of the trie size, even when searching for longest-matching prefixes. The new design permits very efficient, simple and nonrecursive implementations of small code size and minimal storage requirements. Insert and delete operations have strictly local effects, and their particular sequence is irrelevant for the structure of the resulting trie, thus maintaining at all times the desired storage and computational efficiency. The algorithms have been successfully employed in experimental communication systems and products for a variety of networking functions such as address resolution, maintenance and verification of access control lists, and high-performance routing tables in operating system kernels.

I. INTRODUCTION

THE retrieval of context information identified by a sequence of binary digits, or binary keys, is an operation that is literally ubiquitous in computer networks [5], [24]. Applications range from the mapping of connection identifiers to state information for protocol processing of various kinds, to assigning and checking access or security tokens for users of shared network resources, to directory look-ups such as for address resolution purposes [7]. The particular example that initiated the design of our new algorithms stems from modern routing protocols. These protocols view addresses as unstructured binary sequences, and they distribute reachability information in the form of address prefixes, relaying data and, control packets according to the longest matching prefix rule [3], [4], [11], [12], [14], [19], [22]. That is, reachability information is distributed between routers in the form of variable-length binary sequences, each of which describes the potentially huge set of end-system addresses with the given binary sequence as prefix [5], [13], [24]. Given a destination address, a router bases its forwarding decisions on that address prefix among all those known to it, that constitutes the longest prefix of the destination address. Hence, a need arises to organize the address prefixes of which a router becomes aware over time into dynamic routing tables, or forward information bases [11], [26], such that searches for longest matching prefixes may be performed very efficiently.2

In many such applications, the respective databases of binary keys fluctuate in size rather dynamically, and the employed algorithms often execute in environments, such as operating system kernels in which:

- Code and storage efficiency is crucial.
- Strict stack limitations require nonrecursive and robust implementations.
- The processing time is at a premium, such that reliable upper bounds for the algorithmic complexity of all operations are required independent of the size of the database—most prominently so for the retrieval operation.

Our observations led us to derive the following list of requirements for a database of binary keys to be widely applicable in networking environments:

- Keys must be allowed to be of variable length and prefixes of each other.
- Operations must be provided for random insertion and deletion of keys without requiring recursion.
- The worst-case times for retrievals should have formally proven upper bounds independent of the size of the database, even for longest-matching prefix searches.
- Insertion and deletion operations should permit a flexible trade-off between performance and auxiliary information required in the database, and in all cases variants should be possible for which the execution time does not depend on the size of the database.
- The storage complexity should be kept to a minimum.
- Implementations that are simple and easy to understand should be possible.

A survey of applicable data structures and algorithms [10], [18] revealed that none of them meets all of our requirements without major modifications. Be it that no [9], [21] or only inadequate [2] delete operations are defined, support of prefixes is ruled out or inflicts significantly additional overhead [18], [21], nonrecursive implementations become intimidatingly complex [25], or the bounds for the algorithmic complexity depend on the size of the databases [27] or exhibit probabilistic nature [25], [27]. Hence, we decided to extend and improve the most efficient general binary trie structure known to date, the Patricia tries [1], [15], [16], [20], [21], [28], by defining a deletion operation that always restores the respective prior state of the trie and whose execution time is
independent of the size of the database, and by integrating support for storing and retrieving (longest) matching prefixes without compromising the well-proven algorithmic efficiency [23] for the case when no prefixes are present.

The next section defines the data structures for dynamic prefix tries (DP-Tries), the name of which was chosen to reflect the built-in support for random insertion and deletion of keys that may be prefixes of each other. Then we present the algorithms for insertion, deletion, and retrieval and prove their relevant properties. We conclude the article with a section on the performance of DP-Tries.

II. DATA STRUCTURES AND DEFINITIONS

The algorithms presented here operate on (binary) keys, defined as nonempty bit sequences of arbitrary but finite size. Keys are represented as \( k = b_0 \ldots b_{l-1} \), with elements \( k[i] = b_i \in \{0, 1\} \), \( 0 \leq i \leq |k| \), with \( |k| = l - 1 \) representing the width of \( k \) and \( |k| = l \) its length. Following standard terminology, a key \( k' \) is said to be a (strict) prefix of key \( k \), denoted by \( k' \preceq k \) (\( k' < k \)), iff \( |k'| \leq |k| \) \( (k' \preceq k) \) holds, and \( k'[i] = k[i] \), \( 0 \leq i \leq |k'| \). In particular, two keys are identical iff they are prefixes of each other. Keys are stored in nodes, each comprised of an index, at most two keys, and up to three links to other nodes. That is, a given node \( n \) has the following components [cf. Fig. 1(a)]:

**Index(n)**

The relevant bit position for trie construction and retrieval—the prefix to this bit position is common among all the keys in the subtrie that has node \( n \) as its root—the index allows the nonrelevant bits to be skipped, thus obviating one-way branches.

**Leftkey(n)**

A key with \( \text{LeftKey}[	ext{Index}(n)] = 0 \) or \( \text{NIL} \).

**Rightkey(n)**

A key with \( \text{RightKey}[	ext{Index}(n)] = 1 \) or \( \text{NIL} \).

**Parent(n)**

A link to the parent node of node \( n \), \( \text{NIL} \) for the root node.

**LeftSubTrie(n)**

A link to the root node of that subtrie of node \( n \) with keys \( k \) such that \( k[\text{Index}(n)] = 0 \), or \( \text{NIL} \).

**RightSubTrie(n)**

A link to the root node of that subtrie of node \( n \) with keys \( k \) such that \( k[\text{Index}(n)] = 1 \), or \( \text{NIL} \).

Finally, a **dynamic prefix trie** is formally defined as any set of nodes and keys that form a binary trie with the properties listed in Lemma 3 of Section III.

To illustrate the data structures and the particular relation between nodes, we will now construct a DP-Trie for a sample set of keys \{1000100, 1001, 10, 11111, 11\} using the graphical representation for nodes depicted in Fig. 1(a). When the first key, 1000100, is inserted into the trie, we obtain Fig. 1(b). The trie consists of only a root node, \( a \). As key 1000100 is the only key of node \( a \), the index takes its maximum value six, i.e., the width of the key. The key has a zero at bit position 6 and hence becomes the left key of the root. Fig. 1(c) depicts the DP-Trie after key 1001 is inserted. The index of node \( a \) becomes three, the first bit position at which the two keys differ, and the common prefix 100 is thus ignored. As the new key has a one in position 3, it becomes the right key of node \( a \). (If the new key had a zero at this bit position and the old key a one, the old key would have been moved to become the right key and the new key would have become the left key.) In this trie, a search is guided to either stored key by the bit value at position 3 [\text{Index}(a)] of an input key; the bit positions 0, 1, and 2 are skipped. For example, a search with input 100100111 returns 1001 as the longest matching prefix.

Adding key 10 results in Fig. 2(a). As key 10 is a prefix to both keys of node \( a \), a new node, \( b \), is created and made the parent of node \( a \). The index of node \( b \) is one, the width of key 10. Key 10 and node \( a \) become, respectively, the left key and the left subtrie of node \( b \), because all the stored keys have zero at position 1. Adding key 11111, we obtain Fig. 2(b). This key differs from key 10 in the bit position 1. So without a change in the index value, node \( b \) can accommodate the new key as its right key. Fig. 2(c) shows the DP-Trie after key 11 is added. Key 11 is a prefix of key 11111. Therefore, key 11111 is replaced by key 11 and pushed lower to the new node, \( c \), that becomes the right subtrie of node \( b \).

Now, let us consider some more search operations. A search for \( k = 10011 \) at node \( b \) is first guided to node \( a \) because \( k[1] = 0 \), and then to 1001; bit positions 0 and 2 are skipped. The latter key is identified as the longest matching prefix after a total of two bit and one key comparison, which are very basic and easily implemented operations in suitable hardware [23]. In contrast, a search for 100011 proceeds to node \( a \) only
to find that 1000100 is not a prefix of the input key. Using the pointer reversal provided by the parent links, the key 10 is identified as the longest matching prefix. Similarly, a search for key 001 branches to node a and terminates unsuccessfully at node b.

Further insertion of the keys depicted in Fig. 3 produces a sample DP-Trie that covers all structural aspects of general DP-Tries and will be used henceforth, as a reference. This figure can also be used to examine search operations involving more than a few nodes. For example, a search for key 1000011 is guided through nodes d, b, a, h, and i, skipping over bit positions 2 and 5. Then through pointer reversal, 1000 is identified as the maximum-length matching prefix.

III. PROPERTIES OF DP-TRIES

In this section, we describe the most relevant properties of DP-Tries that also establish the correctness of the algorithms defined in Section IV. The formal proof of the following results is given in Section VII, including the fact that the algorithms always preserve the stated properties. We begin our discussion with the central estimate for the storage complexity of DP-Tries.

Let us call a Prefix Branch any sequence of nodes $n_i, \ldots, n_k$, such that $n_{i+1}$ is the root of the only subtree of $n_i$ for $i \leq k-1$, and all nodes store exactly one key as strict prefixes of each other. Then we can derive the following result that shows the algorithms presented preserve the storage efficiency of Patricia tries in a naturally generalized way.

Lemma 1—Storage Complexity of DP-Tries: Let $a^+$ denote $\text{Max}(0, a)$ for a number $a$. The algorithms of a DP-Trie guarantee a minimal storage complexity described by the following relation, which is an optimal estimate for general DP-Tries: $\#\text{nodes} \leq \#\text{keys} + (\#\text{PrefixBranches} - 1)^+$.

The example in Fig. 4 shows that the upper bound is optimal for DP-Tries, and the established results for Patricia tries follow from observing that in those tries, no prefix branches may exist.

The next result confirms that the structure of a DP-Trie is determined by the set of inserted keys only.

Lemma 2—Invariance Properties of DP-Tries:

1) For any given set of keys, the structure of the resulting DP-Trie is independent of the sequence of their insertion.
2) The delete operation always restores the respective prior structure of a DP-Trie.
3) The structure of a DP-Trie does not depend upon a particular sequence of insert or delete operations.

The correctness of the algorithms derives from the properties of DP-Tries stated in the next Lemma and the fact that they are invariant under the insert and delete operations (cf. Section VII).

Lemma 3—General Properties of DP-Tries:

N1 For a given node $n$, let Keys$(n)$ denote the set of keys stored in $n$ and its subtries. All such keys have a width of at least Index$(n)$ and hence a length of at least Index$(n) + 1$.

N2 Keys$(n)$ denotes the subset of keys that share the respective common prefix of Index$(n)$ bits in length, denoted by Prefix$(n)$. That is, all keys in Keys$(n)$ match in bit positions up to and including Index$(n) - 1$. If Index$(n)$ is distinguishing, Prefix$(n)$ is of maximum length.

N3 The keys in any subtrie of node $n$ share a common prefix of at least Index$(n) + 1$ bits and are at least of such width.

N4 The index of a node $n$ is maximal in the following sense: In a leaf node having only one key $k$, we have Index$(n) = |k|$, and in all other nodes there exist keys in Keys$(n)$ that differ at position Index$(n)$, or one of the keys stored in $n$ has a width of Index$(n)$.

N5 Every node contains at least one key or two nonempty subtries. That is, a DP-Trie contains no chain nodes.
N6 If the left (right) key as well as the left (right) subtrie of node \( n \) are nonempty, then \[ \text{LeftKey}(n) = \text{Index}(n) \]
\( \text{RightKey}(n) = \text{Index}(n) \).

N7 If the left (right) key of a node \( n \) is empty and the left (right) subtrie is not, then the subtrie contains at least two keys.

N8 For a given node \( n \), the left (right) key and all keys in the left (right) subtrie of \( n \) have a 0 (1) bit at position \( \text{Index}(n) \).

P1 \( \text{Prefix}(n) \) increases strictly as \( n \) moves down a DP-Trie from the root to a leaf node, as does \( \text{Index}(n) \). Hence, there are no loops and there always exists a single root node.

P2 Every full path terminates at a leaf node that contains at least one key, and there exists only one path from the root to each leaf node.

IV. ALGORITHMS

This section provides a description of the algorithms for DP-Tries. They are represented in a notation that avoids distracting details, but is kept at a level that allows straightforward translations into real programming languages, such as C, as has been used by the authors for their implementations. The correctness of the algorithms follows directly from the properties of DP-Tries stated in Lemma 3 and the observation that these properties are invariant under the insertion and deletion operations as shown in Section III. We preface the description with the definition of a small set of commonly used operators, predicates, and functions. For convenience of representation, we shall generally denote a trie or subtrie by a pointer to its root node.

The first bit position where two keys \( k \) and \( k' \) differ is called their distinguishing position, and is defined as \( \text{DistPos}(k, k') = \min \{ i | k[i] \neq k'[i] \} \), with the convention that \( \text{DistPos}(k', k) = |k'|+1 \) for \( k' \leq k \). Furthermore, a node is said to be a (single key) leaf node if it has no subtries (and only stores one key). For example, in Fig. 2(c), node \( c \) is a single-key leaf-node, whereas node \( a \) is only a leaf-node, and \( \text{DistPos}(a, b) = 1 \).

The two operations that select the pertinent part of a node relative to an input key are defined below based on the convention that nonexistent subtries or keys are represented by the special value NIL.

\[
\text{Key}(\text{node, key}) \begin{cases} 
\text{return NIL} & \text{if } (\text{key} \geq \text{Index(\text{node})}) \\
\text{Key(\text{node})} & \text{if } (\text{key}[\text{Index(\text{node})}] = 0) \\
\text{Key(\text{node})} & \text{else}
\end{cases}
\]

\[
\text{SubTrie}(\text{node, key}) \begin{cases} 
\text{return NIL} & \text{if } (\text{key} < \text{Index(\text{node})}) \\
\text{SubTrie(\text{node})} & \text{if } (\text{key}[\text{Index(\text{node})}] = 0) \\
\text{SubTrie(\text{node})} & \text{else}
\end{cases}
\]

As an example, consider the nodes in Fig. 2(c). For \( k = 0001 \), we get as SubTrie\((b, k)\) the trie starting at node \( a \), and \( \text{Key}(b, k) = 10 \).

A. Insertion

The insertion of a new key proceeds in three steps. First, the length of the longest common prefix of the new key with any of the keys stored in leaf nodes is identified (STEP1). Based on this information, the node around which the new key is to be inserted is located (STEP2). Finally, the key is added to the trie in the appropriate place (STEP3).

To accomplish the first step, we descend to a leaf node guided by the input key and make default choices where required. That is, as long as the width of the new key is greater than or equal to the respective nodal index, we follow the pertinent subtrie, if it exists. Otherwise we proceed along any subtrie, possibly with some implementation-specific (probabilistic) bias, until we reach a leaf node.\(^3\) The keys in that node may then be used to calculate the longest common prefix (see Fig. 3 and Lemma 4). The node for insertion is identified by backtracking the downward path followed in the first step until a node is reached whose index is less than or equal to the length of the common prefix and the width of the input key, or the root node is reached (cf., Fig. 3). Depending on the prefix relations between the input key and those in the insertion node, the new key may then be inserted above [Fig. 2(a)], in [Fig. 1(b)], or below the selected node [Fig. 2(b)].

We now proceed to the description of the insertion algorithm, making use of the following auxiliary operations, that attempt to select subtries and keys of a node as closely related to an input key as possible. The procedures will be used to formalize the best effort descent of the first step.

\[
\text{ClosestKey(node, key)} \begin{cases} 
\text{return NIL} & \text{if } (\text{key} < \text{Index(\text{node})}) \\
\text{Key(\text{node})} & \text{if } (\text{key}[\text{Index(\text{node})}] = 0) \\
\text{Key(\text{node})} & \text{else}
\end{cases}
\]

\[
\text{ClosestSubTrie(node, key)} \begin{cases} 
\text{return NIL} & \text{if } (\text{key} < \text{Index(\text{node})}) \\
\text{SubTrie(\text{node})} & \text{if } (\text{key}[\text{Index(\text{node})}] = 0) \\
\text{SubTrie(\text{node})} & \text{else}
\end{cases}
\]

\(^3\)This step may be optimized. See Section IV-A4.)
return (any subtrie of node or NIL if none exists)
}

The operation that allocates and initializes new nodes of a
DP-Trie is formalized as

Allocate Node(index, key) {
  local node
  NEWNODE(node) /* Allocate space for a new node */
  LeftKey(node) := RightKey(node) := NIL
  Parent(node) := LeftSubTrie(node) :=
    RightSubTrie(node) := NIL
  Index(node) := index; Key(node, key) := key
  return(node)
}

Given the above definitions, the algorithm for key insertion
is defined in the following procedure. Further detailed
comments on the processing and the storage requirements are
summarized in Table I.

Insert(key) {
  local node, distpos, index

  /* Empty trie. Just add a node. Root is the global */
  /* pointer to the root node. */
  if (Root = NIL) then Root := AllocateNode(key, key)

  /* NonEmpty trie: Insertion proceeds in three steps. */
  else
    /* STEP1: Descend to a leaf node, identify the */
    /* largest common prefix, and the index of the */
    /* node to store the new key. Cp. Lemma 4. */
    node := Root /* Start at the Root */
    while (NotLeafNode(node))
      do node := ClosestSubTrie(node, key)
      distpos = DistPos(key, ClosestKey(node, key))
      /* Cp. Lemma 4. */
      index = Min(key, distpos)
      /* of the node to store the new key */

    /* STEP2: Ascend toward the root node, */
    /* identify the insertion node. */
    /* Cp. Lemma 4. */
    while ((index < Index(node)) and (node ≠ Root))
      do node := Parent(node)

    /* STEP3: Branch to the appropriate insert operation. */
    /* See the following subsections */
    if (node = Root)
      InsertInOrAbove(node, key, distpos)
    elseif (SubTrie(node, key) = NIL) then
      InsertWithEmptySubTrie(node, key, distpos)
    else
      InsertWithNonEmptySubTrie(node, key, distpos)
  }

The relevant properties of the local variable distpos and of the
insertion node identified in STEP2 are summarized in the
following Lemma.

Lemma 4—Properties of the Insertion Node: Let n denote
the insertion node of STEP2 of the insertion algorithm, then:
  a) The new key does not belong to any subtrie of n.
  b) If Min(distpos, |key|) ≥ Index(n), then the key be-
     longs to the trie starting at n.
  c) If Min(distpos, |key|) < Index(n), then n is the root
     and the new key needs to be inserted in or above n.
  d) Distpos is the length of the longest common prefix of
     the insert key with any key stored in a leaf node of the trie.

1) InsertInOrAbove: This procedure deals with insertions
above the root node. If the new key is not a prefix of a single
key in the root node, it is simply added and the node index
adjusted accordingly. Otherwise, the key is stored in a new
node that becomes the root, and the current trie is added as
its only subtrie.

InsertInOrAbove(node, key, distpos) {
  local newnode, index := Min(|key|, distpos)

  /* InorAbove-1: Add the new key to the root node. */
  if (|key| ≥ distpos and SingleKeyLeafNode(node)) then
    Index(node) := index
    Key(node, BitComplement(key)) :=
      ClosestKey(node, key)
    Key(node, key) := key

  /* InorAbove-2: Add the new key in a new node above the root and */
  /* the current trie as its subtrie at the appropriate side of the new node */
  else
    newnode := AllocateNode(index, key)
    Parent(newnode) := newnode
    if (|key| ≥ distpos) then /* InorAbove-2:1: No prefix */
      SubTrie(newnode, BitComplement(key)) := node
    else /* InorAbove-2:2: The new key is a prefix of the stored keys! */
      SubTrie(newnode, key) := node
    Root := newnode /* A new Root has been created. */
  }

2) InsertWithNonEmptySubTrie: This procedure covers
the cases of nonempty subtrees at the insertion node. The new key
is added if it fits exactly (see Lemma 3 [N6]), otherwise a
new node with this key is inserted below the located node
and above its subtrie. To improve the readability of the main
algorithm, we introduce an additional subfunction that links
two nodes as required for a given input key.

InsertWithNonEmptySubTrie(node, key, distpos) {
  local newnode, subnode, index := Min(|key|, distpos)
/* NonEmpty-1: The new key fits exactly. */
if (|key| = Index(node)) then Key(node, key) := key
/* NonEmpty-2: A new node is inserted below node */
/* and above the subtree */
else
  subnode := SubTrie(node, key) /* Save pointer. */
  newnode := AllocateNode(index, key)
  ParentSubTrie(node, key) := newnode
  LinkNodes(node, newnode, key)
/* Add the old subtree at the appropriate side */
if (|key| ≥ dispos) then
  /* NonEmpty-2.1: No prefix */
  if (SingleKeyLeafNode(subnode)) then
    /* Garbage collect this node! */
    Key(newnode, BitComplement(Key)) := ClosestKey(subnode, key)
    DeallocateNode(subnode)
  else
    SubTrie(newnode, key) := subnode
  else /* NonEmpty-2.2: New key is prefix */
    SubTrie(newnode, key) := subnode

3) InsertWithEmptySubTrie: This procedure performs the additions of keys in or below nodes whose pertinent subtree is empty. It is the most complex of all insertion functions in that it must distinguish between five cases. The insertion node may not have a respective key stored, the new key and the stored key may be equal, the new or the stored key fits exactly and one of them needs to be stored in a new node below the current one, or, the most involved case, one key is a strict prefix of the other and both keys need to be placed into two separate nodes.

InsertWithEmptySubTrie (node, key, dispos) {
  local storedkey, dpos, newnode, newnewnode, index := Min(|key|, dispos)

  /* Empty-1: The new key may just be added to an existing node */
  if (Key(node, key) = NIL) then Key(node, key) := key

  /* Empty-2: The new key is a duplicate. */
  elseif (Key(node, key) = key) then
    Key(node, key) := key

  /* Empty-3: The stored key fits, and the new one */
  /* needs to be added below node. */
  elseif (|key| = Index(node)) then
    newnode := AllocateNode(key, key)
    LinkNodes(node, newnode, key)

  /* Empty-4: The new key fits, and the stored key */
  /* needs to be pushed down. */
  else
    storedkey := Key(node, key) /* Save the stored key. */
    Key(node, key) := NIL /* Will be moved. */
    dpos := DistPos(key, storedkey) /* Distinguishing position */

    /* Empty-5.1: The keys are not prefixes of each */
    /* other and may hence be stored in one node. */
    if (dpos ≤ Min(|key|, |storedkey|)) then
      newnode := AllocateNode(dpos, key)
      Key(newnode, storedkey) := storedkey
      LinkNodes(node, newnode, storedkey)

    /* Empty-5.2: The stored key is a strict prefix */
    /* of the new key; Each key is stored in a separate new node. */
    elseif (dpos > |storedkey|) then
      newnode := AllocateNode(storedkey, storedkey)
      LinkNodes(node, newnode, storedkey)
      newnewnode := AllocateNode(Other(key), key)
      LinkNodes(newnewnode, storedkey)

    /* Empty-5.3: The new key is a strict prefix of */
    /* the stored key; Each key is stored in a separate new node. */
    /* newnode := AllocateNode(key, key) */
    /* LinkNodes(node, newnode, key) */
    /* newnewnode := AllocateNode(Other(key), key) */
    /* LinkNodes(newnewnode, storedkey) */
  }

4) Algorithmic Complexity: Step3 of the algorithm has a complexity of O(1), and the first two steps are linear in the depth of the trie. In the absence of prefixes, the depth depends logarithmically on the number of keys (cf., Lemma 1 and [17]), whereas the general case may degenerate to a linear list of prefixes. However, by storing at each node the respective common prefix of all keys stored in that node and its subtrues (cf., Lemma 3 [N2] in Section III) the first two steps may be combined and terminated at the latest, when the insert key is exhausted. That is, the insert operation can be performed independent of the size of the database as a trade-off against a small increase in storage complexity. In all cases, Step3 affects, at most, the insertion node, its parent or the root node of one of its subtrues.

B. Deletion

To delete a key, we simply erase it from its node. However, the storage and computational efficiency of DP-Tries hinges
on the fact that the structure of a DP-trie is determined solely by its keys and not by a respective sequence of insert and delete operations. Hence, the deletion algorithm needs to run a garbage-collection function in nodes from which keys have been deleted in order to restore the respective prior trie structure: If a node becomes empty after a key deletion, it is simply removed. Nodes that no longer store keys and have only one subtrie, so-called chain nodes, are also removed by linking their parent node and subtrie directly. When a node becomes a single-key leaf-node, its index is maximized by appropriately swapping its key. As a last step of the garbage-collection function, an attempt is made to move the key from a new single-key leaf node into its parent node. Further comments may be found in Table I.

**DeleteKey(key)**

```plaintext
def DeleteKey(key):
    local node, collnode := NIL, storedkey := NIL

    /* Step1: Search for the key to be deleted */
    node := Root
    if (node ≠ NIL and (key ≥ Index(node))) then
        while ((SubTrie(node, key) ≠ NIL) and
                (key ≥ Index(SubTrie(node, key))))
            do node := SubTrie(node, key)
    /* Step2: Delete the leaf key and garbage collect nodes */
    if (node = NIL) or (Key(node, key) ≠ key)
        return(NotFound)

    /* DeEmpty: Delete an empty node */
    if (Empty(node)) then
        /* DeEmpty-1: The Root is not deleted. */
        if (node ≠ Root) then
            SubTrie(Parent(node), key) := NIL
            collnode := Parent(node)
        /* DeEmpty-2: The Root is deleted. The trie is now empty. */
        else Root := NIL
        DeallocateNode(node)

    /* DeTail: Delete chain nodes */
    elseif (ChainNode(node)) then
        if (node ≠ Root)
            SubTrie(Parent(node), key) := SubTrie(node, key)
            Parent(SubTrie(node, key)) := Parent(node)
            collnode := SubTrie(node, key)
        else
            Root := SubTrie(node, key)
            Parent(Root) := NIL
        DeallocateNode(node)

    /* DeMax: Maximize the index of single-key leaf-nodes */
    elseif (SingleKeyLeafNode(node)) then
        storedkey := Key(node, BitComplement(key))
        LeftKey(node) := RightKey(node) := NIL
        Index(node) := |storedkey|
        Key(node, storedkey) := storedkey
        collnode := node

        /* DeSingle: Handle a single-key subtree */
    elseif ((SubTrie(node, key) ≠ NIL) and
               SingleKeyLeafNode(SubTrie(node, key))) then
        collnode := SubTrie(node, key)

    /* Step3: Last step of garbage collection */
    /* DeGC: Attempt to move keys from single-key */
    /* leaf-nodes to the parent node */
    if (collnode ≠ NIL and
        SingleKeyLeafNode(collnode)) then
        storedkey := ClosestKey(collnode, key)

        if (Parent(collnode) ≠ NIL) and
            (Key(Parent(collnode), storedkey) = NIL) then
            Key(Parent(collnode), storedkey) := storedkey
            SubTrie(Parent(collnode), storedkey) := NIL
            DeallocateNode(collnode)
    } /* End DeleteKey(key) */
```

**Algorithmic Complexity:** The search down the trie is linear in the length of the input key, and the removal of the key and the garbage collection of nodes has a complexity of $O(1)$. Hence, the complexity of the delete operation does not depend on the size of the trie. The deletion has only a local impact on the trie structure in that it affects, at most, the node that stores the pertinent key and the node below and above it.

**C. Search**

The search algorithm performs a descent of the trie under strict guidance by the input key, inspecting each of its bit positions, at most, once. Once this first step has terminated, the traversed path is backtracked in search of the longest prefix. The decision not to perform key comparisons on the downward path resulted from a bias toward successful searches. In a networking environment, such as when performing routing decisions, negative results typically resemble error situations that cause data packets to be discarded and error messages to be sent with low priority. Notification or recovery of such errors is not deemed to be overly time critical [5], [24].

**SearchKey(key)**

```plaintext
def SearchKey(key):
    local node := Root

    /* Check for empty tries and short keys */
    if (node = NIL) or (key < Index(node)) then
        return(NIL)

    /* Step1: Downward path */
    while ((SubTrie(node, key) ≠ NIL) and
            (key ≥ Index(SubTrie(node, key))))
```
do node := SubTrie(node, key)

/* Step2: Backtracking to find the longest prefix */
while (node ≠ NIL) and (Key(node, key) ≠ key))
do node := Parent(node)

/* If a node was found, then it stores the longest prefix */
if (node ≠ NIL) then return(Key(node, key))
else return(NIL)
}

Algorithmic Complexity: The complexity of the downward and upward searches are linear in the size of the input key independent of the size of the trie. In most implementations, the operations on the downward path will require only very little processing because only pointers need to be moved and single bits to be compared. On the upward path, the test for prefix relationships may be optimized for a given environment, such as by taking advantage of processor word sizes and instruction sets.

V. PERFORMANCE

The performance of a DP-Trie is evaluated here in terms of the average time it takes to perform insert, delete, and search operations. The measurements are obtained using an implementation of the DP-Trie in ANSI C. In the case of fixed-length keys, a comparison is made with the performance of the well-known AVL tree [18]. The AVL tree algorithms used here were written by T. Bolmarcich of IBM Watson Research Center. Their performance closely matches that of another implementation reported in [25]. The performance results are in terms of the CPU time on an IBM RISC System/6000 model 970, which has a SPEC'92 integer performance of 47.8. The code was compiled using the xlc compiler at the optimization level 3.

Fig. 5 shows the mean insert, delete, and search times, in DP-Trie and AVL tree, as a function of the number of keys, where all keys are four bytes long. Here the keys were derived from uniformly distributed random integers (results were also obtained for keys based on a real set of IP addresses, but no significant difference was observed). Clearly, for all three operations, DP-Trie outperforms AVL tree. This indicates that the unique capability of DP-Trie to perform maximum-length prefix matching is not obtained at the cost of lower performance in the case of fixed-length keys. Also, detailed measurements indicate that the difference between the processing times of the three operations is almost entirely due to the time incurred in the insert and delete algorithms for, respectively, allocating and freeing space in memory. In the environment in which the measurements were performed, it was more costly to free space than to allocate it.

Let us consider the case where keys may be prefixes of each other. Fig. 6 shows the mean search time, in DP-Trie, as a function of the numbers of keys, where three key distributions are considered. Let \( N \) denote the number of keys in the Trie. The key distributions are parameterized by \( p \), the size of prefix, for \( p = 1, 2, 3 \). More specifically, in the \( p \)th distribution, there are \( N^{\frac{1}{2}} \) \( p \)-byte keys and \( N - N^{\frac{1}{2}} \) four-byte keys. The \( p \)-byte keys are randomly generated and each one is used as the prefix of \( N^{1 - \frac{2}{p}} - 1 \) four-byte keys. The remaining \( 4 - p \) bytes of four-byte keys are randomly generated.

Fig. 6 shows that the average search time increases linearly as a function of \( \log(N) \). This indicates that the trees are quite balanced when there are prefixes. It can also be observed in the figure that the shorter the matching prefix, the longer it takes for the search operation to find it. This is a consequence of the fact that the search algorithm starts from the root, moves downward to a leaf and then backtracks to the first node with a matching prefix. Clearly, the shorter the matching prefix, the more nodes are traversed in the backtracking phase. In fact, because the trees are quite balanced, we can easily derive approximations that match the curves in Fig. 6 with high accuracy. Let \( c_0 \) denote the constant part of search time. In a balanced trie, the number of nodes traversed from the root to a leaf is \( \log(N) \). Let \( c_1 \) denote the time spent at each node on the forward path. When there is no complete match, the number of nodes backtracked from the leaf to the node with the matching prefix is \( \frac{4 - p}{4} \log(N) \). Let \( c_2 \) denote the time spent backtracking each node, excluding the time spent for key comparisons. It turns out for the three key distributions considered here, when there is a complete match, exactly one key comparison is made. Otherwise, two key comparisons...
are made, first with a four-byte key and then with a $p$-byte key. Let $c_3$ denote the time it takes for a key comparison. Then the time of a search resulting in a $p$ byte match is $c_0 + c_1 \log(N) + c_2 \left(\frac{2 - p}{3}\right) \log(N) + 2c_3$ and the time of a search resulting in a complete match is $c_0 + c_1 \log(N) + c_3$. These approximations explain three features of Fig 6: a) the search time resulting in a complete match is independent of the length of prefix keys, $p$, so the bottom three curves are almost identical; b) the top three curves, that correspond to partial matches, are equally spaced, with a separation distance of $\frac{c_2}{4} \log(N)$; c) there is a wider gap between these curves and the bottom ones, namely $\frac{c_2}{4} \log(N) + c_3$.

As an application example, let us consider the performance of a dynamic address database for a heterogeneous internetworking environment built with DP-Tries as its base [6], [8]. This database maps between addresses of applications and those of the physical nodes they are installed on, all of which employ a common, general address format with three bit-strings of arbitrary and unrestricted sizes, i.e., a triple (AddressFamily, HostAddress, LocalAddress). Thus, each application address is mapped to one or more physical addresses owned by the same node, whereby no restrictions are placed on the mutual compatibility of the respective address components.

Fig. 7 shows the average search time in the address database as a function of the number of nodes in the network. It is assumed here that every node owns $A$ applications and $P$ physical addresses, where each of the node's application addresses are mapped to all of its $P$ physical addresses. Each curve corresponds to a specific value of $(A, P)$. As expected, the average search time increases logarithmically with the number of nodes, because the total number of keys, i.e., application addresses, is equal to $A$ times the number of nodes. Let $d_1$ denote the rate of increase. The spacing between $(1,1)$ and $(10,1)$ curves is $d_1 \log(10)/1$ and that between $(10,3)$ and $(50,3)$ curves is $d_1 \log(50)/10$. The spacing between $(10,1)$ and $(10,3)$ is the constant representing the additional time for retrieving a list of physical addresses that contains three elements instead of one.

Comparing Figs. 5 and 7, we observe the average search time of the address database is about one order of magnitude larger than that of the basic DP-Trie. The reason behind this is the following. The address database evaluated here is part of an actual product. It was made highly reliable by consistency checking at various levels and provides additional functions such as group-address mapping, physical-address-to-application-address mapping and wildcard operations, that result in additional complexity.

The decision to implement the address database using DP-Trie instead of some other search data structures and algorithms was based on the following observations. DP-Trie is the only trie that handles variable-length keys and provides prefix matching. Although linked lists can provide these features, their performance is not acceptable, even for a relatively small number of keys. Furthermore, even for fixed-size keys, DP-Trie performs favorably compared to other tries, such as the AVL Tree.

VI. CONCLUSIONS

This article introduces a novel binary tree, referred to as DP-Trie, and associated algorithms that provide fast searches for longest matching prefixes with efficient storage utilization. No performance penalty is paid for fixed-length keys in that DP-Tries perform better than the state-of-the-art, as represented by AVL Tries.

DP-Tries have been successfully employed in experimental communication systems and products for a variety of networking functions, such as address resolution in a multiprotocol environment with a wide diversity of address formats, for the maintenance and verification of access control lists, and for high-performance routing tables in operating system kernels. With a minor modification to the search algorithm, DP-Tries can allow convenient access to all prefixes matching a given input key, and hence provide a natural implementation mechanism for the many network management applications operating on management information bases. Further applications of DP-Tries are being investigated, such as general alphabets with extensions for multiway branching, for efficient text retrieval, and for bibliographic searches.

VII. PROOFS

This section presents the proofs of the results stated in Section III about DP-Tries. We start with Lemma 3, that lists the properties of DP-Tries establishing the correctness of the algorithms for insertion, deletion, and retrieval.

Proof of Lemma 3: The assertions in Lemma 3 follow from a straightforward induction argument on the number of inserted keys, and by verifying that the cited properties are preserved for each of the insertion cases (cf. Section IV-A) and that the deletion operation (cf. Section IV-B) restores the pertinent prior state of the trie. The full proof has been omitted here, and we refer the reader to Table 1 with comments on the algorithms of Section IV that contain all the arguments required to establish a rigorous proof. The table lists for each insertion case of Section IV-A the respective changes of the number of nodes and keys in a DP-Trie and summarizes the operations performed when a pertinent key is deleted. The
TABLE I

Details

<table>
<thead>
<tr>
<th>Insertion case</th>
<th>Storage</th>
<th>Deletion case</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyTrie: Add the new key node with maximum index.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>InsertAbove1: Add the new key to the old node and adjust the index. The insertion node is the root node, which is a single-key leaf node here.</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>InsertAbove2:1: The new key is not a prefix of the keys in the trie, and index denotes the first position where the keys differ.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>InsertAbove2:2: In the prefix case, the trie is just linked to the new node.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>NonEmpty-1: Add the key.</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>NonEmpty-2:1: As the new key is not a prefix of the subtree keys, the key of an existing single key leaf node may be moved into the new node and the leaf node garbage collected.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>NonEmpty-2:2: Link subtree to new node because keys are prefixes of each other.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-1: Add the new key.</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-2: No change.</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>Empty-3: The new key is added to the new node with maximum index.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-4: Move the old key into a new node below the insertion node.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-5:1: The two keys are stored in a new node below the insertion node because they are not prefixes of each other.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-5:2: The two keys are added in two separate nodes below the insertion node, thus forming a new prefix branch consisting of two nodes.</td>
<td>+2</td>
<td>+1</td>
</tr>
<tr>
<td>Empty-5:3: The two keys are added in two separate nodes below the insertion node, thus forming a new prefix branch consisting of two nodes.</td>
<td>+2</td>
<td>+1</td>
</tr>
</tbody>
</table>

The estimate of the storage complexity follows from Table I.

**Proof of Lemma 1:** The cases Empty-5.2 and Empty-5.3 are the only ones where two nodes have to be added to include one key and where prefix branches are formed. To see that the storage estimate of Lemma 1 also holds for these cases, we conclude first from Table 1 that always \#Nodes ≤ \#Keys + \#PrefixBranches. However, the first time either case occurs, one may assume by Lemma 2 of Section III, and the fact that the width of the stored key is larger than the index of the insertion node, that the stored key was inserted last and into an already existing node. Hence we have prior to the insertion of the new key that \#Nodes < \#Keys, which implies that we need not count the very first prefix branch, and the result follows. The above argument no longer applies for any further occurrence of cases 5.2 and 5.3 because the insertion node may be the same node as in the previous case.

We proceed with the proof of Lemma 4 about the properties of the insertion node of step2 of the insert operation.

**Proof of Lemma 4:** Let nIns denote the insertion node and key the new key to be inserted. If the key belonged to
a subtrie of nIns, we could deduce from N1 and N2 that
\(|key| \geq \text{Index}(\text{subtrie})\), and also \(|distpos| \geq \text{Index}(\text{subtrie})\).
In contradiction to the construction of STEP2 we would thus arrive at \(\text{Min}(|key|, |distpos|) \geq \text{Index}(\text{subtrie})\). This proves a).

b) follows from N2 and the construction of STEP1.

To establish c), we first note that STEP2 implies for \(\text{Min}(|key|, |distpos|) < \text{Index}(\text{nIns})\) that \(\text{nIns}\) is the root node. For \(|key| < \text{Index}(\text{nIns})\) assertion c) follows from N1. If \(|distpos| < \text{Index}(\text{nIns})\) holds, then all keys in the trie match with the new key, at most, up to bit position \(|distpos| - 1\). Hence, the new key cannot be added to \(\text{Keys}(\text{nIns})\) by N1, and c) follows also for this case.

For a proof of d), it suffices to show that keys in nodes that are not traversed in STEP1 cannot have a longer common prefix with \(key\). Assuming the contrary, we may infer from the monotonicity of \(\text{DistPos}(\text{key}, k)\) that there exist two leaf nodes \(n, n'\) with keys \(k\) and \(k'\) such that \(|\text{distpos}| = \text{DistPos}(k, key) < \text{DistPos}(k', key)\), with \(n\) denoting the leaf node from STEP1. Let \(n''\) be the unique node where the paths from the root node to \(n\) and \(n'\) diverge (P2). Then we may conclude that \(|key| \geq \text{Index}(n'')\) and by the construction of STEP1 that \(k[\text{Index}(n'')] = k'[\text{Index}(n'')]\). However, this implies by N2 that \(\text{DistPos}(k, key) = \text{DistPos}(k', key)\leq \text{DistPos}(k, key)\) with \([k'''] = \text{Index}(n'')\) and \([k''] = k[i]\) for \(0 \leq i \leq |k'''|\), contradicting our assumption.

We conclude this section with the proof of Lemma 2 about the invariance properties of DP-Tries.

**Proof of Lemma 2:** Assume that assertion (a) holds for sets of keys with cardinality less than or equal to \(N \geq 1\). Let \(T\) and \(T'\) denote two DP-Tries built by two sequences of insertions of the same set of keys \(K\) of cardinality \(N + 1\). We want to show that \(T = T'\). We will do so by concluding that both tries have identical root nodes and then apply the induction hypothesis on their subtries.

By P1, we may identify root nodes \(R\) and \(R'\), respectively, and by N4 and P1, we see that \(\text{Index}(R) = \text{Index}(R')\). Let \(i\) denote this common index, i.e., \(i := \text{Index}(R)\). If \(R\) contains no keys, N3 implies that all keys in \(K\) are of width \(\geq i + 1\). N5 and N7 imply that both subtries of \(R\) contain at least two keys each, and we may conclude from N6 that \(R'\) cannot contain any keys. However, the induction hypothesis and N8 imply that both subtries of \(R\) and \(R'\) are of the same structure, and hence, \(T = T'\) in this case.

Assume now that \(\text{LeftKey}(R)\) is not empty. If \(\text{LeftSubTrie}(R)\) were empty, by N8 there exists only one key in \(K\) with a zero bit at position \(i\). By N7, \(\text{LeftSubTrie}(R')\) is also empty and hence \(\text{LeftKey}(R') = \text{LeftKey}(R)\). In the case of a nonempty left subtrie, N3, N6, N8, and the induction hypothesis imply again that \(\text{LeftKey}(R') = \text{LeftKey}(R)\) and \(\text{LeftSubTrie}(R') = \text{LeftSubTrie}(R)\). Applying the above reasoning to the right key and subtries completes the proof of (a), (b) follows from the comments listed in Table I for the delete operation, and (c) is a direct consequence of (a) and (b).

**ACKNOWLEDGMENT**

It is a pleasure to acknowledge the valuable comments provided by our colleague L. Svobodova on the initial draft of this paper. Thanks also go to the reviewers for their helpful comments, and to L. M. Pavka for her editorial suggestions.

**REFERENCES**


Willibald Doering (M'95/ACM'95) received the Ph.D. degree in mathematics from the University Karlsruhe, Germany.

He is a faculty member of the computer science department of the Palatinate State College where he lectures in mathematics, computer science and communications. His recent research activities focus on internetworking and routing in large computer networks.

Mehdi Nasseri (M'87) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1987.

He has been with the IBM Zurich Research Laboratory since 1987. He is interested in network-based systems and currently works in the area of electronic commerce.

Dr. Nasseri served as an editor for IEEE TRANSACTIONS ON COMPUTERS from 1990 to 1993.

Günter Karpf received the degree in computer science and the Ph.D. degree from the University of Stuttgart, Germany, in 1980 and 1987, respectively.

He was a Visiting Scientist at the Swedish Institute of Computer Science in Kista, Sweden, in 1986. Since 1986, he has been with the IBM Zurich Research Laboratory as a Research Staff Member. His research interest is in the modeling of distributed systems, their validation and implementation. He is currently working on access control issues in network management and object systems.

Dr. Karpf has been a member of the ACM and of the German Computer Society (GI) since 1981.