I/O Efficient Dynamic Data Structures for Longest Prefix Queries*

Moshe Hershcovitch[1] and Haim Kaplan[2]

[1] Faculty of Electrical Engineering, moshik1@gmail.com
[2] School of Computer Science, haimk@cs.tau.ac.il,
Tel Aviv University, Tel Aviv 69978, Israel

Abstract. We present an efficient data structure for finding the longest prefix of a query string p in a dynamic database of strings. When the strings are IP-addresses then this is the IP-lookup problem. Our data structure is I/O efficient. It supports a query with a string p using $O(\log_B(n) + \frac{|p|}{B})$ I/O operations, where B is the size of a disk block. It also supports an insertion and a deletion of a string p with the same number of I/O's. The size of the data structure is linear in the size of the database and the running time of each operation is $O(\log(n) + |p|)$.

1 Introduction

We consider the *longest prefix problem* which is defined as follows. The input consists of a set of strings $S = \{p_1 \ldots p_n\}$ which we shall refer to as *prefixes*. We want to preprocess S into a data structure such that given a query string q we can efficiently find the longest prefix in S which is a prefix of q or report that no prefix in S is a prefix of q. We focus on the dynamic version of the problem where we want to be able to insert and delete prefixes to and from S, respectively.

The main application of this problem is for packet forwarding in IP networks. In this application a router maintains a set of prefixes of IP addresses according to some routing protocol such as BGP (Border Gateway Protocol). When a packet arrives, the router finds the longest prefix of the destination address of the packet and sends the packet on the outgoing link associated with this longest prefix. A longest prefix match query in this settings is often called *IP-lookup*.

The rapid growth of the Internet has brought the need for routers to maintain large sets of prefixes, and to perform longest prefix match queries at high speeds [7]. A main issue in the design of routers is the size of the expensive high speed memory used by the router for packet forwarding. One can reduce the size of this expensive memory by using external memory components. Therefore the I/O efficiency of the algorithm we use is very important. In software implementations the entire data structure may not fit into the cache and we may flip parts of it back and forth from the main memory or the disk. In hardware implementations it may be too expensive to fit the entire data structure in the memory which is

* This work is partially supported by United states - Israel Binational Science Foundation, project number 2006204.

integrated in the chip that implements the forwarding algorithm itself. So parts of the data structure may reside in external memory devices (such as DRAM). In such designs the communication between the main chip and the external memory becomes the performance bottleneck of the system.

In addition to good I/O performance an efficient data structure for IP-lookup should be able to perform queries at the line rate (which is about 40 Gbps and more today). The data structure has to be scalable since the number of prefixes a router has to maintain is growing rapidly as well as the length of these prefixes. Finally, we need to support fast updates mainly due to instabilities in backbone routing protocols and security issues.

Our computational model. Our algorithm works in the classical *pointer machine model*[19] using only *comparisons* to manipulate prefixes. This is in contrast with many other algorithms for IP-lookup that use bit manipulations on IP addresses. See Section 1. This restriction which we obey does not come at the cost of a complicated data structure. On the contrary, our data structure is much simpler than previously known data structure with the same guarantees.

We develop our data structures in two steps. First we reduce the longest prefix problem to the problem of finding the shortest segment containing a query point. For this data structure we assume that each prefix fits into a constant number of computer words so that we can compare two endpoints of segments in $O(1)$ time. In the second step we relax this assumption and deal with variable length strings with no apriori upper bound on their length. We assume that the strings are over an ordered alphabet Σ and that we can compare two characters of Σ in $O(1)$ time. We do not require direct access to the characters of each prefix.

To analyze I/O performance we use the standard external memory model where memory is partitioned into blocks of size B, and we count the number of blocks that we have to transfer from slow to fast memory in order to perform the operation. This quantity is *the number of I/O operations* performed by the operation.[20].

Overview of our results. We first consider the problem of maintaining a dynamic set of segments for point stabbing queries. Specifically, we consider a dynamic *nested family of segments* where each pair of segments are either disjoint or one contains the other. We develop a data structure that given a point p can efficiently find the shortest segment containing p.

Our data structure for this problem which is based on a segment tree (with large fan-out) is particularly simple. In a segment tree we map a segment s to every node v, such that s contains[3] v, and does not contain the parent of v. Typically one maintains at each node v all the segments that map to v in some secondary structure [18, 13, 12]. We make the crucial observation that for our point stabbing query it is sufficient to maintain for each node v only the shortest segment which maps to v.

Our data structure performs a query, and an insertion or a deletion of a segment in $O(\log n)$ time and $O(\log_B(n))$ I/O operations. We manipulate the segments only via comparisons of their endpoints.

[3] A segment contains a node if it contains all points in its subtree.

We use this result to solve the longest prefix problem as follows. We associate a segment $[pL, pR]$ with each string p, where L and R are two new characters, smaller and larger than all other characters, respectively. We then apply the previous data structure to this set of segments. This gives the data structure for the case where we assume that two prefixes can be compared in $O(1)$ time.

To handle strings with no fixed bound on their lengths we combine this idea with the powerful string B-tree of Ferragina and Grossi [9]. This data structure is a B-tree carefully designed for storing strings. For efficient searches and updates it uses a *Patricia trie* [14] in each node. We show how to maintain the information which we need for longest prefix queries using the string B-tree.

Since our data structure is based on a B-tree it is also I/O efficient. If we pick the size of a node so that it fits in a disk block of size B, we obtain that a query or update with a string q performs $O(\log_B(n)+|q|/B)$ I/O operations. The data structure requires $O(n/B)$ disk blocks in addition to the blocks required to store the strings themselves. The time for query or update with a string q is $O(\log(n)+|q|)$. (This is as efficient as with tries implemented carefully [16], but tries cannot be implemented I/O efficiently [6]).

Previous related results. There has been a lot of work mainly in the networking community on the IP-lookup problem. The different data structures can be classified into three families: trie based structures (See for example [7] and the references there), hash based structures (See for example [11] and the references there), and tree based structures. In the rest of this section we focus on dynamic tree based solutions with worst case guarantees that are related to our approach.

Sahni and Kim [15] describe a solution based on a collection of red-black trees that requires linear space and logarithmic time per operation. Feldmann and Muthukrishnan [8] proposed Fat Inverted Segment tree (FIS). This data structure supports queries in $O(\log\log n + \ell)$ time, where ℓ is the number of levels in the segment tree. The space requirement is $O(n^{1+1/\ell})$, and insert and delete take $O(n^{1/\ell}\log n)$ time, but there is an upper bound on the total number of insertions and deletions allowed. Suri et al. [18] proposed a data structure which is similar to ours in the sense that it is both a segment tree and a B-tree. But they store in each node all the segments which are mapped to it and therefore achieve logarithmic worst case time bound per operation and linear space only for IP-addresses. Lu and Sahni [13] suggested an improvement of the segment tree of Suri that stores each prefix only in one place. They maintain other bit vectors in internal nodes and their update operations are quite complicated. Our structure, which can be extended to general strings (and general segments) and uses only comparisons, is simpler than all the solutions mentioned above. In particular it is much cleaner than the latter two B-tree based implementations when applied to IP addresses.

Kaplan, Molad, and Tarjan [12] considered the problem of point stabbing a dynamic nested set of segments. In their setting, which is more general than ours, each segment has a priority associated with it and we want to find the segment of minimum priority containing a query point. They present a data structure performing query and update in $O(\log n)$ time that requires linear space. It uses both a balanced search tree and a dynamic tree [17] and thereby more complicated than ours (when applied to the special case where the priority of an interval is its length).

In recent years, external memory data structures have been developed for a wide range of applications [20]. A classical I/O efficient data structure is the B-tree [2]. This is a search tree in which we choose the degree of a node so that it occupies a single block. The *string B-tree* of Ferragina and Grossi [9] is a fundamental extension of the B-tree for storing unbounded strings. The main idea is to use a *Patricia trie* [14] in each node to direct the search. Unfortunately this data structure by itself does not solve the longest prefix problem.

Agarwal, Arge, and Yi [1] improved a more general data structure of Kaplan, Molad, and Tarjan [12] for stabbing-min queries against general segments (not necessarily nested). This data structure is based on a *B-tree* and can be implemented so that it is I/O efficient. Specifically, a data structure for n intervals uses $O(n/B)$ disk blocks and $O(\log_B(n))$ I/O operations for query and update. This data structure is quite complicated and assumes that endpoints of intervals can be compared in $O(1)$ time. Therefore it is not directly applicable for longest prefix queries in a collection of unbounded strings.

Brodal and Fagerberg [5] obtained a cache oblivious (see Section 4) data structure for manipulating strings. This data structure, which is essentially a trie can be used to obtain an I/O efficient (though complicated) solution for the static version of the longest prefix problem.

The outline of the rest of the paper is as follows. In section 2 we present our basic ideas using the assumption that strings are of constant size. In section 3 we combine our ideas with the string B-tree to obtain a general I/O efficient solution. In section 4 we suggest a future research.

2 B Tree for Longest Prefix Queries

Our input is a set of prefixes $S = \{p_1 \ldots p_n\}$ which are strings over the alphabet Σ. We think of each prefix p as a segment $I(p) = [pL, pR]$, where L and R are two special characters not in Σ, L is smaller than all characters in Σ and R is larger than all characters in Σ. The longest prefix p of a query q corresponds to the the shortest segment $I(p)$ containing q. We describe a dynamic data structure to maintain a set of nested segments such that we can find the smallest segment containing a query point. Although we can apply our data structure to any set of nested segments we present our result using the string terminology and the set of segments $\{I(p) \mid p \in S\}$.

Let $P = \{p_iL,\ p_iR \mid p_i \in S\}$ be the set of endpoints of the prefixes in S. We store P ordered lexicographically at the leaves of a B^+ *tree* T. Each internal node x of T has $n(x)$ children, where $b \leq n(x) \leq 2b$. If x is a leaf then it stores $n(x)$ endpoints of P, where $b \leq n(x) \leq 2b$. (See Figure 1.)

To simplify the presentation we assume that a leaf x with $n(x)$ endpoints has $n(x) + 1$ “dummy” children. From now on when we say a leaf of T, we refer to one of these dummy nodes, and we refer to x as a *height-1 node*. Each endpoint in a height-1 node plays the role of a key separating two consecutive

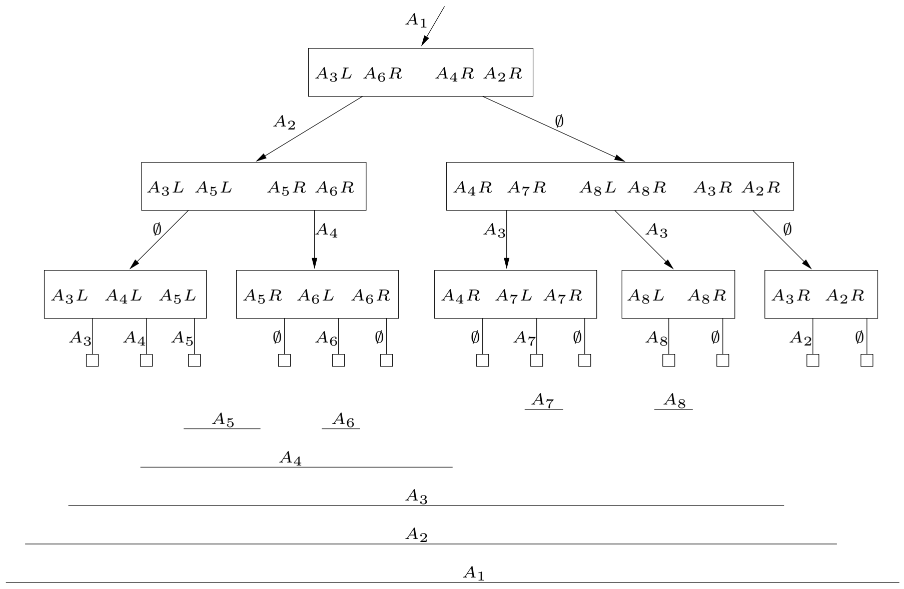

Fig. 1. A B^+ *tree* with $b = 2$ storing the prefixes $A_1, \ldots, A_8$. Rectangles correspond to internal nodes and squares correspond to dummy leaves. In each height-1 node we show the endpoints that it stores. In each internal node v of height > 1 we show the spans of its children which are also used as the keys which direct the search. (Note that when we use the spans as keys, a search can never reach the first dummy leaf in each height-1 node. Therefore we do not need to keep longest prefixes of these nodes and we do not show them in the figure.) The span of a child u of v is the closed interval from the point depicted to the left of the edge from v to u to the point depicted to the right of the edge from v to u. On each edge $(p(v), v)$ we show the longest prefix of v.

dummy leaves. We associate each leaf with the open interval from the endpoint preceding the leaf to the endpoint following the leaf. We call this interval the *span* of the leaf. (Note that the last dummy leaf in a height-1 node and the first dummy leaf in the next height-1 node have the same span.) We define the *span* of an internal node v to be the smallest interval containing the endpoints which are descendents of v.[4] We denote the span of a node v by $span(v)$. We think of T as a segment tree and map each segment $[pL, pR]$ to every node v such that pL is to the left of $span(v)$ and pR is to the right of span(v), and either pL or pR are in $span(p(v))$. We define the longest prefix (or the shorter segment) of v and denote it $LP(v)$ to be the shortest segment mapped to v.[5] If there isn't any segment which is mapped to v we define $LP(v)$ to be empty. ($LP(v)$ is also defined if v is a dummy leaf.)

[4] This is the interval that starts at the leftmost endpoint in the subtree of v, and ends at the rightmost endpoint in the subtree of v.

[5] Note that all segments mapped to v form a nested family of segments, so the shortest among them is unique.

We store *span*(v) and $LP(v)$ with the pointer to node v. Note that when we are at a node v we can use the span values of the children of v as the keys which direct the search. We denote by $B = O(b)$ the maximum size of a node. We pick b so that B is the size of a disk block.

2.1 Finding the longest prefix

Assume we want to find the longest prefix of a query string q. We search the B^+ *tree* with the string q[6] in a standard way and traverse a path A to a leaf of T. We return $LP(w)$, where w is the last node on A, such that $LP(w)$ is not empty.

The correctness of the query follows from the following observations. For each prefix p of q, $I(p)$ is mapped to some node u on A. Therefore p is $LP(u)$ unless some longer prefix is mapped to u. Furthermore, since for every v, $LP(p(v))$ is a prefix of $LP(v)$, it follows that the longest prefix of q must be $LP(w)$, where w is the last node on A for which $LP(v)$ is not empty.

2.2 Inserting a new prefix

To insert a new prefix p we have to insert $I(p)$ into T. We insert pL and pR into the appropriate *height-1* nodes w and w', respectively, according to the lexicographic order of the endpoints. The endpoint pL is inserted into the span of a leaf y and the endpoint pR is inserted into the span of a leaf z. Assume first that $z \neq y$. The span of y is now split between two new leaves: y' that precedes pL and y'' that follows pL. We set the longest prefix of y' to be the longest prefix of y and the longest prefix of y'' to be p. The span of z is now split between two new leaves: z' that precedes pR and z'' that follows pR. We set the longest prefixes of z' to be p and the longest prefix of z'' to be the longest prefix of z. If $y = z$ then the span of y is split between three new leaves y^1, y^2, and y^3. We set the longest prefixes of y^1 and y^3 to be the longest prefix of y, and the longest prefix of y^2 to be p.

There may be nodes v in T, such that after adding p, we have to update $LP(v)$ to be p. Let y be the leaf preceding pL and let z be the leaf following pR. Let u be the lowest common ancestor of y and z. Let u' be the child of u which is an ancestor of y and let u'' be the child of u which is an ancestor of z. We may need to update $LP(v)$ if either:

Case 1: v is a child of a node w on the path from u' to y, which is right sibling of the child w' of w on this path.
Case 2: v is a child of a node w on the path from u'' to z, which is left sibling of the child w' of w on this path.
Case 3: v is a child of u which is a right sibling of u' and left sibling of u''.

[6] In fact we “pretend” to search with a string (not in the data structure) that immediately follows qL in the lexicographic order of the strings.

For each such node v, we know that $span(v) \subset I(p)$ so we change $LP(v)$ to be p, if p is longer than the current $LP(v)$. Since the depth of T is $O(\log_B(n))$, we update $O(B \log_B(n))$ longest prefixes which are stored at $O(\log_B(n))$ nodes.

After inserting pL and pR, if the *height-1* node containing pL and the *height-1* node containing pR have no more than $2b$ children, we finish the insert. Otherwise we have to split at least one of these nodes. We split node v into two nodes v_1 and v_2. Node v_1 is the parent of the first b (or $b+1$) children of v and node v_2 is the parent of the last $b+1$ children of v. Both v_1 and v_2 replace v as consecutive children of $p(v)$. We compute $span(v_1)$ from the span of its first child and the span of its last child, and similarly for $span(v_2)$.

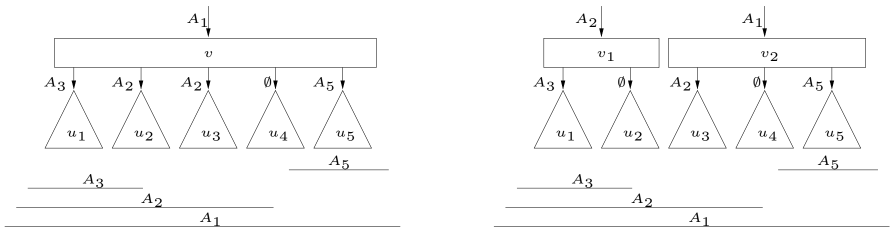

Fig. 2. A node v at the left which is split into nodes v_1 and v_2 to the right. Since $span(v_1) \subseteq I(A_2)$ the prefix A_2, which was the longest prefix of u_2 before the split, is the longest prefix of v_1 after the split. The longest prefix of u_2 after the split is empty.

Clearly we have to update $LP(v_1)$ and $LP(v_2)$. Furthermore, since a segment that was mapped to a child u of v may now be mapped to v_1 or v_2, we may also have to update $LP(u)$ for children u of v_1 and v_2. Other longest prefixes do not change. The following simple observations specify how to update the longest prefixes. In the following if u is a child of v prior the split, then $LP(u)$ refers to the longest prefix of u before the split. Note that since v exists only before the split then $LP(v)$ is the longest prefix of v before the split. Similarly, $LP(v_1)$ and $LP(v_2)$ are the longest prefix of v_1 and v_2, respectively, after the split.

Lemma 1. *Let u be a child of v_1 after the split. If $span(v_1) \subset I(LP(u))$ then after split $LP(u)$ should be empty.*

Proof. Let $p = LP(u)$ since $span(v_1) \subset I(LP(u))$ then the segment $I(p)$ is not mapped to u after the split. Since $I(p)$ was the shortest segment that was mapped to u no other segment is mapped to u after the split. □

Lemma 2. *Let u_1 and u_2 be children of v_1. If $span(v_1) \subset I(LP(u_1))$ and $span(v_1) \subset I(LP(u_2))$ then $LP(u_1) = LP(u_2)$.*

Proof. Since $span(v_1) \subset I(LP(u_1))$ then $span(u_2) \subset I(LP(u_1))$. So $LP(u_1)$ cannot be longer than $LP(u_2)$ since this would contradict the fact that $I(LP(u_2))$ is the shortest segment containing $span(u_2)$. Symmetrically, $LP(u_2)$ cannot be longer than $LP(u_1)$, so they must be equal. □

Lemma 3. *If there exist child u of v_1 such that $span(v_1) \subset I(LP(u))$ then $LP(v_1)$ is $LP(u)$.*

Proof. Obviously $LP(u)$ is mapped to v_1. Furthermore, $LP(u)$ is the longest prefix with this property, since if there is a longer prefix q with this property then q should have been $LP(u)$ before the split. □

Lemma 4. *If there isn't a child u of v_1 such that $span(v_1) \subset I(LP(u))$ then $LP(v_1)$ is equal $LP(v)$.*

Proof. We claim that there exists a child u of v_1 that $LP(u)$ is empty. From this claim the lemma follows since if there is a prefix q longer than $LP(v)$ such that $I(q)$ is mapped to v_1, then q is mapped to u before the split and $LP(u)$ couldn't have been empty.

We prove this claim as follows. Assume to the contrary that $LP(u)$ is not empty for every child u of v_1. Let w be a child of v_1 such that $I(LP(w))$ is not contained in $I(LP(w'))$ for any other child w' of v (w exists since segments do not overlap). From our assumption follows that $I(LP(w)) \subseteq span(v_1)$. Therefore at least one of the endpoints of $I(LP(w))$, say z is in the subtree of v_1. Let w'' be a child of v_1 whose subtree contains z. It is easy to see now that $I(LP(w''))$ and $I(LP(w))$ overlap which is a contradiction. □

A symmetric version of Lemmas 1, 2, 3, and 4 hold for v_2.

These observations imply the following straightforward algorithm to update longest prefixes when we perform a split. If there is a child u of v_1 such that $span(v_1) \subset I(LP(u))$ we set $LP(v_1)$ to be $LP(u)$, otherwise we set $LP(v_1)$ to be $LP(v)$. In addition we set $LP(u)$ to be empty for every child u of v_1 such that $span(v_1) \subset I(LP(u))$. We update the span of v_2 and its children analogously. See Figure 2.

After splitting v we recursively check if $p(v_1)$ or $p(v_2)$ has more than $2b$ children and if so we continue to split them until we reach a node that has no more than $2b$ children.

2.3 Deleting a prefix

To delete a prefix p we need to delete $I(p)$ from T. We first find the longest prefix of p in S denoted by w.[7] Then we delete pL and pR from the *height-1* nodes containing them.

We have to change the longest prefix of every node v for which $LP(v) = p$ to w. Nodes v for which $LP(v)$ may be equal to p are of three kinds as specified in Cases (1), (2) and (3) of Section 2.2

As a result of deleting pL and pR from the *height-1* nodes containing them we may create nodes with less than b children. To fix such node v we either *borrow* a child from a sibling of v or *merge* v with one of its siblings. We omit the details of these rebalancing operations and their affect on longest prefixes from this abstract. The following theorem summarizes the results of this section.

[7] We do that by a query with a string (not in the data structure) that immediately follows pR in the lexicographically order of strings.

Theorem 1. *Assuming each string occupies $O(1)$ words, the B-tree data structure which we described supports longest prefix queries, insertions, and deletions in $O(\log(n))$ time. Furthermore, it performs $O(\log_B(n))$ I/Os per operation, and requires linear space.*

3 String B-tree For Longest Prefix Queries

In a *B-tree*, we assume that $\Theta(b)$ keys that reside at a single node fit into one disk block of size B. However if the keys are strings of variable sizes, which can be arbitrarily long, there may not be enough space to store $\Theta(b)$ strings in a single block. Instead, we can store $\Theta(b)$ pointers to strings in each node, but accessing these strings during the search requires more than a constant number of I/O operations per node. To reduce the number of I/Os, Ferragina and Grossi [9] developed an elegant generalization of a *B-tree* called the *string B-tree* or *SB-tree* for short.

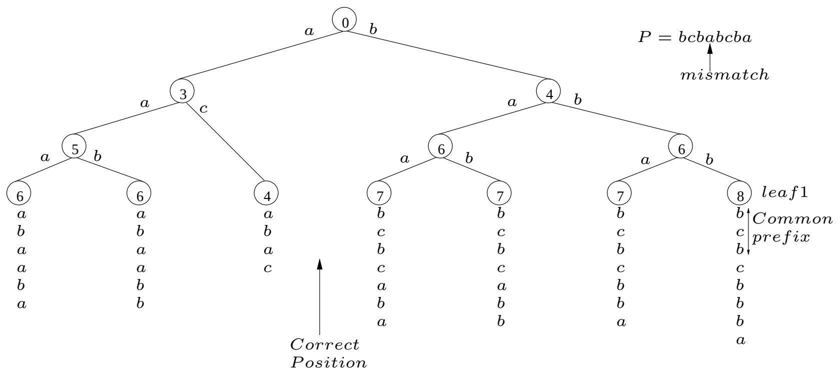

Fig. 3. A Patricia trie of a node in a string B-tree. The number in a node is its string depth. The character on an edge is the branching character of the edge.

An individual node v of an *SB-tree* is shown in Figure 3. Instead of storing the keys at a node v we store a *Patricia trie* [14] of the keys, denoted by $PT(v)$. Using this representation we can perform b-way branching using only $\Theta(b)$ characters that are stored in a constant number of disk blocks of size B. Each internal node ξ of the Patricia trie stores the length of the string corresponding the path from the root to ξ. We call this the *string depth* of ξ. We store with each edge e the first character of the string that corresponds to e. This character is called the *branching character* of e.

As an example Figure 3 shows a Patricia trie of a node in a string B-tree. The right child of the root has string depth 4 and it's outgoing edges have the branching characters "a" and "b", respectively. This means that the node's left subtrie consists of strings whose fifth character is "a" , and its right subtrie consists of strings whose fifth character is "b". The first four characters in all the strings in the right subtrie of the root are "*bcbc*". Let ξ be a node of the trie whose string depth is $d(\xi)$. To make a branching decision at ξ, we compare the $d(\xi)+1$ character of the string that we search, to the characters on the edges outgoing from ξ. For example, for the string "*bcbabcba*", the search in the trie in Figure 3 traverses the rightmost path of the Patricia trie, examining the characters 1, 5, and 7 of the string which we search.

Unfortunately, the leaf of the Patricia trie that we reach (in our example, the leaf at the far right, corresponding to "*bcbcbbba*") is not in general the correct branching point, from the node of the *SB-tree* represented by this trie, since we did not compare all characters of the string which we search. We fix this by sequentially comparing the string which we search with the key associated with the leaf of the trie which we reached. If they differ, we find the position in which they first differ. In the example the first character of the string "*bcbabcba*" that is not equal to the corresponding character of the key "*bcbcbbba*", is the fourth character. Since the fourth character of "*bcbabcba*" is smaller we know that the string which we search is lexicographically smaller than all keys in the right subtree of the root. It thus fits in between the leaves "*abac*" and "*bcbcaba*". For more details see [9].

Searching each Patricia trie requires constant number of I/O to load it into memory, plus additional I/Os to do the sequential scan of the key associated with the leaf we reached. Therefore our structure as defined so far does not guarantee that the total number of I/Os is $O(\log_B n + \ell/B)$, where ℓ is the length of the string that we search.

To further reduce the number of I/Os Ferragina and Grossi [9] used the leftmost and the rightmost strings in the subtree of a node v as keys at $p(v)$. Recall that we in fact did the same in our B-tree when we use the spans of the children of v as the keys at v. Having the keys defined this way, we can use information from the search in the trie $PT(v)$ of a node v to reduce the number of I/Os in the followings search of the trie $PT(u)$ of a child u of v. Specifically, let s be the string which we search, and let ℓ be the length of the longest common prefix of s and the key at the leaf of $PT(v)$, where the search ended. Then it is guaranteed that the length of the longest common prefix of s and the key at the leaf of $PT(u)$, where the search of s ends is at least ℓ. Thus, we can avoid the first ℓ comparisons and the I/Os associated with them. Ferragina and Grossi [9] also showed how to insert and delete a string in $O(\log_B n + \ell/B)$ time in the worst case.

We now describe how to combine the SB-tree with our algorithm for longest prefix queries so that our input prefixes $S = \{p_1 \dots p_n\}$ can be arbitrarily long. As Ferragina and Grossi [9], we use the endpoints of $span(v)$ as keys at $p(v)$, and represent the keys of each node v in a Patricia trie $PT(v)$. Each leaf of the Patricia trie stores a pointer to the first block containing the key that it corresponds to. We use the same definition of the longest prefix of a node v, denoted by $LP(v)$, as in Section 2. Recall that from these definitions follow that if $LP(v)$ is not empty then $span(v) \subset I(LP(v))$ and therefore $LP(v)$ is a prefix of every key in the subtree of v. Let $span(v) = [KL(v), KR(v)]$. That is $KL(v)$

be the leftmost string in the subtree of v and $KR(v)$ is the rightmost string in the subtree of v. Clearly $LP(v)$ is a prefix of $KL(v)$ and $KR(v)$. The string $KL(v)$ is a key separating v from its sibling in $p(v)$ and therefore corresponds to a leaf in $PT(v)$. So we represent $LP(v)$ by storing its length, and pointer to it, in the leaf of $PT(v)$, that corresponds to $KL(v)$. If $LP(v)$ is empty we encode this by storing zero at the associated leaf.

Finding the longest prefix. We search the *SB-tree* and traverse a path A to a leaf of T. Let w be the last node on A for which $LP(w)$ is not empty. Together with the pointer to w in $p(w)$, we find $|LP(w)|$ and a pointer to $LP(w)$.

Inserting a new prefix. Assume we want to insert a new prefix $p \in S$ to the data structure. We insert pL and pR into the *SB-tree* using the insertion algorithm of the *SB-tree*. As in Section 2.2 there may be nodes v in T, such that after adding p, we need to update $LP(v)$ to be p. Nodes v for which $LP(v)$ may be equal to p are of three kinds as specified in Cases (1), (2) and (3) of Section 2.2. For each such node v we change $|LP(v)|$ to be $|p|$, if $|p|$ is longer than the current value $|LP(v)|$. This is correct since for each of these nodes v, we know that $span(v) \subset I(p)$. Note that all these changes are located at $O(\log_B(n))$ nodes of the *SB-tree*, and therefore we can perform them while doing $O(\log_B(n))$ I/O operations.

After inserting a prefix p we may split node v into two nodes v_1 and v_2. We split a node in the *SB-tree* using the algorithm of Ferragina and Grossi [9]. Splitting may change the longest prefixes. To perform these changes we use the same algorithm as in Section 2.2. To implement this algorithm we need to determine if there is a child u of v_1 such that $span(v_1) \subset I(LP(u))$.

Let u be a child of v_1. We decide if $span(v_1) \subset I(LP(u))$ as follows. Since $LP(u)$ is a prefix of $KL(u)$ and $KR(u)$, and $KL(u)$ and $KR(u)$ are keys in $PT(v_1)$ then there is a path in $PT(v_1)$ that corresponds to the string $LP(u)$. It follows that $LP(u)$ is a prefix of all the keys in $PT(v_1)$, and in particular of $KL(v_1)$ and $KR(v_2)$, if $|LP(u)|$ is not larger than the string depth of the root of $PT(v_1)$. We check if there is a child u of v_2 that $span(v_2) \subset I(LP(u))$ analogously.

Deletion of a prefix is similar, we omit the details from this abstract. The following theorem summarizes the results of this section.

Theorem 2. *The data structure which we described in this section supports longest prefix queries, insertions, and deletions in* $O(\log(n) + |q|)$ *time where* q *is the string which we perform the operation with. Furthermore, it performs* $O(\log_B(n) + |q|/B)$ *I/Os per operation, and requires linear space.*

4 Future Research

The cache oblivious model [10] is a generalization of the I/O model. In this model we seek I/O efficient algorithms which do not depend on the block size. Among the state of the art in this model is a cache-oblivious B-tree [3], and an almost efficient cache-oblivious string B-tree [4] whose query time is optimal but updates are not. An obvious open question is to find a cache oblivious data structure for longest prefix queries.

References

1. P. K. Agarwal, L. Arge, and K. Yi. An Optimal Dynamic Interval Stabbing-Max Data Structure? In *Proceedings of SODA*, pages 803–812, 2005.
2. R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informtica*, 1(3):173–189, 1972.
3. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.
4. M. A. Bender, M. Farach-Colton, and B. C. Kusznaul. Cache-Oblivious String B-Trees. In *Proceedings of PODS*, pages 233–242, 2006.
5. G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, 2006.
6. Erik D. Demaine, John Iacono, and Stefan Langerman. Worst-case optimal tree layout in a memory hierarchy, 2004.
7. W. Eatherton, Z. Dittia, and G. Varghese. Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communications Review*, 34(2):97–122, 2004.
8. A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. In *Proceedings of INFOCOM*, pages 1193–1202, 2000.
9. P. Ferragina and R. Grossi. The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM*, 46(2):236–280, 1999.
10. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of FOCS*, pages 285–297, 1999.
11. J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar. Chisel: A Storage-Efficient, Collision-Free Hash- Based Network Processing Architecture. In *Proceedings of ISCA*, pages 203–215, May 2006.
12. H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic Rectangular Intersection with Priorities. In *Proceedings of STOC*, pages 639–648, 2003.
13. H. Lu and S. Sahni. A B-Tree Dynamic Router-Table Design. *IEEE Transactions on Computers*, 54(7):813–824, 2005.
14. D. R. Morrison. Patricia: Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
15. S. Sahni and K. Kim. O(log n) Dynamic Packet Routing. In *Proceedings of ISCC*, pages 443–448, 2002.
16. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.
17. D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *JCSS*, 26(3):362–391, 1983.
18. S. Suri, G. Varghese, and P. Warkhede. Multiway Range Trees: Scalable IP Lookup with Fast Updates. In *Proceedings of GLOBECOM*, pages 1610–1614, 2001.
19. R. E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computing System Science*, 18:110–127, 1979.
20. J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, 2001.