The String B-Tree: A New Data Structure for String Search in External Memory and its Applications. *

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

Roberto Grossi
Dipartimento di Sistemi e Informatica
Università di Firenze

April 1998

Abstract

We introduce a new text-indexing data structure, the *String B-Tree*, that can be seen as a link between some traditional external-memory and string-matching data structures. In a short phrase, it is a combination of B-trees and Patricia tries for internal-node indices that is made more effective by adding extra pointers to speed up search and update operations. Consequently, the String B-Tree overcomes the theoretical limitations of inverted files, B-trees, prefix B-trees, suffix arrays, compacted tries and suffix trees. String B-trees have the same worst-case performance as B-trees but they manage unbounded-length strings and perform much more powerful search operations such as the ones supported by suffix trees. String B-trees are also effective in main memory (RAM model) because they improve the online suffix tree search on a dynamic set of strings. They also can be successfully applied to database indexing and software duplication.

Keywords: B-tree, Patricia trie, compound attributes and database indexing, external-memory data structures, magnetic and optical disks, prefix and range searching, string searching and sorting, suffix array, suffix tree, text indexing.

AMS(MOS) subject classifications: 68P05, 68P10, 68P20, 68Q20, 68Q25.

---

[*] The results described in this paper were presented at the *ACM Symposium on Theory of Computing* (1995), see [18]. The first author was supported by MURST of Italy and by a Post-Doctoral Fellowship at the Max-Planck-Institut für Informatik, Saarbrücken, Germany (ferragin@di.unipi.it). The second author was supported by MURST of Italy (grossi@dsi.unifi.it).

1 Introduction

Large-scale heterogeneous electronic text collections are more available now than ever before and range from published documents (e.g. electronic dictionaries and encyclopedias, libraries and archives, newspaper files, telephone directories, textbook materials, etc.) to private databases (e.g., marketing information, legal records, medical histories, etc.). A great number of texts are spread over Internet every day in the form of electronic mail, bulletin boards, World Wide Web pages, etc. Online providers of legal and newswire texts already have hundreds of text gigabytes and will soon have terabytes. Many applications treat large text collections that change over time, such as data compression [49, 50, 14, 35], computer virus detection [28], genome data banks [21], telephone directory handling [12] and software maintenance [7]. Last but not least, databases can also be considered dynamic text collections because their records are essentially byte sequences that change over time. In this context, indexing data structures and searching engines are fundamental tools for storing, updating and extracting useful information from data in external storage devices (e.g., disks or CD-ROMs). However, while main memory is a high-speed electronic device, external memory is essentially a low-speed mechanical device. Main-memory access times have decreased from 30 to 80 percent a year, while external-memory access times have not improved much at all over the past twenty years [38]. Nevertheless, we need external storage because we cannot build a main memory having an unbounded capacity and single-cycle access time. Ongoing research is trying to improve the input/output subsystem by introducing some hardware mechanisms such as disk arrays, disk caches, etc. [38], and is investigating how to arrange data on disks by means of some efficient algorithms and data structures that minimize the number of external-memory accesses [45]. We therefore believe that the design and analysis of external-memory text-indexing data structures is very important from both a theoretical and a practical point of view.

Surprisingly enough, in scientific literature, no good worst-case bounds have been obtained for algorithms and data structures manipulating *arbitrarily-long strings* in external memory. As far as traditional external-memory data structures are concerned, inverted files [39], B-trees [9] and their variations, such as Prefix B-trees [10, 15], are well-known and ubiquitous tools for manipulating large data but their worst-case performance is not efficient enough when their keys are arbitrarily long. As far as string-matching data structures are concerned, suffix arrays [22, 33], Patricia tries [22, 36] and suffix trees [34, 48] are particularly effective in handling unbounded-length strings which are small enough to fit into main memory. However, they are no longer efficient when the text collection becomes large, changes over time and makes considerable use of external memory. Their worst-case inefficiency is mainly due to the fact that they have to be packed into the disk pages in order to avoid that too many pages remain almost empty after a few updates. In the worst case, this situation can seriously degenerate in external memory. In Section 5, we discuss in detail the properties and drawbacks of these tools.

As a result, the design of external-memory text-indexing data structures whose performance is provably good in the worst case is important. In this paper, we introduce a new data structure, the *String B-Tree* [1] which achieves this goal. In a short phrase, it is a com-

---

[1] The original name of the data structure was SB-tree [18, 19]. Recently, Don Knuth pointed out the

bination of B-trees and Patricia tries for internal-node indices that is made more effective by adding extra pointers to speed up search and update operations. In a certain sense, String B-trees link external-memory data structures to string-matching data structures by overcoming the theoretical limitations of inverted files (modifiability and atomic keys), suffix arrays (modifiability and contiguous space), suffix trees (unbalanced tree topology) and prefix B-trees (bounded-length keys). The String B-tree is the first external-memory data structure that has the same worst-case performance as regular B-trees but handles unbounded-length strings and performs much more powerful search operations such as the ones supported by suffix trees.

We formalize our operations by means of two basic problems. We use standard terminology for an s-character string $X[1, s]$ by calling $X[1, i]$ a *prefix*, $X[j, s]$ a *suffix* and $X[i, j]$ a *substring* of X, for $1 \leq i \leq j \leq s$. We say that there is an *occurrence* of a *pattern* string P in X if we can find a substring $X[i, i + |P| - 1]$ equal to P.

Problem 1 (Prefix Search and Range Query). Let $\Delta = \{\delta_1, \ldots, \delta_k\}$ be a set of text strings whose total length is N. We store Δ and keep it sorted in external memory under the insertion and deletion of individual text strings. We allow for the following two queries: (1) Prefix Search(P) retrieves all of Δ's strings whose prefix is pattern P; (2) Range Query(K', K'') retrieves all of Δ's strings between K' to K'' in lexicographic order. We let *occ* denote the number of strings retrieved by a query.

Problem 1 represents the typical indexing problem solved by B-trees, here generalized to treat unbounded-length strings. For example, let us examine string set $\Delta = \{$'ace', 'aid', 'atlas', 'atom', 'attenuate', 'by', 'bye', 'car', 'cod', 'dog', 'fit', 'lid', 'patent', 'sun', 'zoo'$\}$. Prefix Search('at') retrieves strings: '**at**las', '**at**om' and '**at**tenuate' (here, $occ = 3$), while Range Query('cap', 'left') retrieves strings: 'car', 'cod', 'dog' and 'fit' (here, $occ = 4$).

Problem 2 (Substring Search). Let $\Delta = \{\delta_1, \ldots, \delta_k\}$ be a set of text strings whose total length is N. We store Δ in external memory and maintain it under the insertion and deletion of individual text strings. We allow for the query: Substring Search(P) finds all of P's occurrences in Δ's strings. We denote the number of such occurrences by *occ*.

Problem 2 extends Problem 1 because it deals with arbitrary *substrings* of Δ's strings. For example, Substring Search('at') retrieves occurrences '**at**las','**at**om', '**at**tenu**at**e' and 'p**at**ent' (here, $occ = 5$). This generalization inevitably complicates the update operations because, while updating Δ in Problem 1 only involves a single text string, in Problem 2 it involves all of its suffixes.

We investigate Problems 1 and 2 in the classical two-level memory model [16]. It assumes that there is a fast and small main memory (i.e., random access memory) and a slow and large external memory (i.e., secondary storage devices such as magnetic disks or CD-ROMs). The external memory is assumed to be partitioned into transfer blocks, called *disk pages*, each of which contains B atomic items, like integers, characters and pointers. We call B *the disk page size* and a disk page reading or writing operation *disk access*. According to [16], we analyze and provide asymptotical bounds for: (a) the total number

---

existence of a different data structure named "SB-tree" [37], where the "S" stands for "sequential".

of disk accesses performed by the various operations; (b) the total number of disk pages occupied by the data structure.

In the scientific literature there are several indexing data structures that can be employed to efficiently solve Problems 1 and 2. We discuss them in detail in Section 5. We wish to point out here that Problem 1 can be solved by a plain combination of B-trees and Patricia tries for internal nodes. This takes $O(\frac{p}{B}\log_B k + \frac{occ}{B})$ disk accesses for Prefix Search(P), and $O(\frac{m}{B}\log_B k)$ disk accesses for inserting or deleting a string of length m in Δ. Although interesting as $p/B < 1$ in practical cases, this combination does not achieve the optimal theoretical bounds as shown below. As far as Problem 2 is concerned, this combination takes $O(\frac{p}{B}\log_B N + \frac{occ}{B})$ disk accesses for Substring Search(P), and $O((\frac{m}{B}+1)m\log_B N)$ disk accesses for inserting or deleting (all the suffixes of) a string of length m in Δ. Notice that the latter bound is quadratic in m because a string insertion/deletion might require to *entirely rescan* all of its suffixes from the beginning, thus examining overall $\Theta(m^2)$ characters. Another interesting solution is given by a single Patricia trie built on the whole set of suffixes of Δ's strings [13]. This achieves $O(\frac{h}{\sqrt{p}} + \log_p N)$ disk accesses for Substring Search(P), where $h \leq N$ is Patricia trie's height. Inserting or deleting a string in Δ costs at least as searching *for all* of its suffixes individually. These two solutions are practically attractive but do not guarantee provably good performance in the worst case.

Our main contribution is to show that the data structure resulting from the plain combination of B-trees and Patricia tries can be further refined and made more effective by adding extra pointers and proving new structural properties that avoid the drawbacks previously mentioned. By means of String B-trees, we achieve the following results:

Problem 1:

- Prefix Search(P) takes $O(\frac{p+occ}{B} + \log_B k)$ worst-case disk accesses, where $p = |P|$.
- Range Query(K', K'') takes $O(\frac{k'+k''+occ}{B} + \log_B k)$ worst-case disk accesses, where $k' = |K'|$ and $k'' = |K''|$.
- Inserting or deleting a string of length m in string set Δ takes $O(\frac{m}{B}+\log_B k)$ worst-case disk accesses.
- The space usage is $\Theta(\frac{k}{B})$ disk pages, while the space occupied by string set Δ is $\Theta(\frac{N}{B})$ disk pages.

Problem 2:

- Substring Search(P) takes $O(\frac{p+occ}{B} + \log_B N)$ worst-case disk accesses, where $p = |P|$.
- Inserting or deleting a string of length m in string set Δ takes $O(m \log_B(N + m))$ worst-case disk accesses.
- The space used by both the String B-tree and string set Δ is $\Theta(\frac{N}{B})$ disk pages.

The space usage of String B-trees in Problem 1 is proportional to the number k of Δ's strings rather than to their total length N, because we represent the strings by their logical pointers. It turns out that the space occupied is asymptotically *optimal* in both Problems 1 and 2. The constants hidden in the big-Oh notation are small. Additionally, the String B-tree operations take asymptotically *optimal* CPU time, i.e., $O(Bd)$ time when our algorithms read or write d disk pages, and they only need to keep a *constant number* of disk pages loaded in main memory at any time.

1.1 Further Results in External Memory

Let us examine the *parameterized pattern matching* problem, introduced by [7] for identifying duplication in a software system. The problem consists of finding the program fragments that are identical except for a systematic renaming of their parameters. In this case, the program fragments are represented by some parameterized strings, called *p-strings*. A suffix tree generalization, called *p-suffix tree* [7], allows us to search for p-strings online and to identify p-string duplications by ignoring parameter renaming. P-suffix trees and the other p-string algorithms [4, 26, 30] are designed to work in main memory and have to deal with the dynamic nature of parameter renaming. We can formulate Problems 1 and 2 for p-strings and then apply String B-trees to them by means of some minor algorithmic modifications. Consequently, the aforementioned theoretical results regarding strings can be extended to p-strings. Our search bound improves the one obtained in [7, 30] for large alphabets, even when the p-string set is static. We refer the interested reader to Section 6.1 for further details.

Let us now examine the databases that treat variable-length records (*not* necessarily *textual* databases), and in particular, their *compound attribute* organization [31] and [29, Sect 6.5], in which the lexicographic order of some records' combinations is properly maintained. An example of this is indexing an employee database according to the string obtained by concatenating employee's name, office and phone number. Prefix B-trees [9] are the most widely-used tool in managing compound attribute organizations. However, since they work by copying some parts of the key strings, they cause data duplication and space overhead. Conversely, String B-trees fully exploit the lexicographic order and take advantage of the prefix shared by any two (consecutive) key strings. As a consequence, we can use String B-trees to support this organization *without having to copy the attributes* in the data structure because we can interpret each variable-length record as a text string of arbitrary length and so use our solution to Problem 1. The space usage of String B-trees is proportional to the number of key strings and not to their total length; thus, String B-trees achieve much better worst-case space saving with respect to prefix B-trees. We refer the interested reader to Section 6.2 for more details.

1.2 Results in Main Memory (RAM model)

Fixing $B = O(1)$, the String B-tree can be seen as an augmented 2–3-tree [2] that allows us to obtain some interesting results in the standard RAM model, due to its balanced tree topology.

- We improve the online search in suffix trees when they store a dynamic set of strings whose characters are taken from a large alphabet [3, 25]. Specifically, we reduce the searching time from $O(p \log N + occ)$ to $O(p + \log N + occ)$ by using our solution to Problem 2. This was previously achieved by [33] only for a *static* string set by means of suffix arrays.
- We implement dynamic suffix arrays [17] in *linear* $O(N)$ space without using the naming technique of [27]. We still obtain an alphabet-independent search and the updates run within the same time bounds as in [17]. We refer the reader to Section 6.3.
- We obtain a tight bound, i.e., $\Theta(N + k \log k)$, in the comparison model for the problem of sorting Δ's strings online. We start out with an empty String B-tree and then insert Δ's strings one at a time by means of the procedure used in Problem 1. This approach requires a total of $O(N + k \log k)$ comparisons. The lower bound $\Omega(N + k \log k)$ holds because we must examine all of the N input characters and output a permutation of k strings. A straightforward use of compacted tries [29] would require $O(N + k^2 \log k)$ comparisons in the worst case. A recent optimal approach based upon *ternary search trees* has been described in [11].

The rest of this paper is organized as follows. In Section 2, we introduce String B-trees and discuss their main properties and operations. We give a formal, detailed description of them in Sections 3 and 4. In Section 5, we review and discuss some previous work on the most important data structures for manipulating external-memory text collections with the aim of clarifying String B-trees' main properties and advantages. In Section 6, we study the applicability of String B-trees. We conclude the paper with some open problems and some suggestions for further research.

2 The String B-Tree Data Structure

We assume that each string in the input set Δ is stored in a contiguous sequence of disk pages and represent the *strings* by their *logical pointers* to the external-memory addresses of their first character, as shown in Figure 1. We can therefore locate the disk page containing the i-th character of a string by performing a constant number of simple arithmetical operations on its logical pointer. When managing keys in the form of logical pointers to arbitrarily-long strings we are faced with two major difficulties that are not usually encountered in other fields, such as computational geometry [23]:

- We can group $\Theta(B)$ logical pointers to strings into a single disk page but, unfortunately, if we only read this page, we are not able to retrieve strings' characters.
- We can compare any two strings character-by-character but this is extremely inefficient if repeated several times because its worst-case cost is proportional to the length of the two strings involved each time. We call this problem *rescanning*, due to the fact that the same input characters are (re)examined several times.

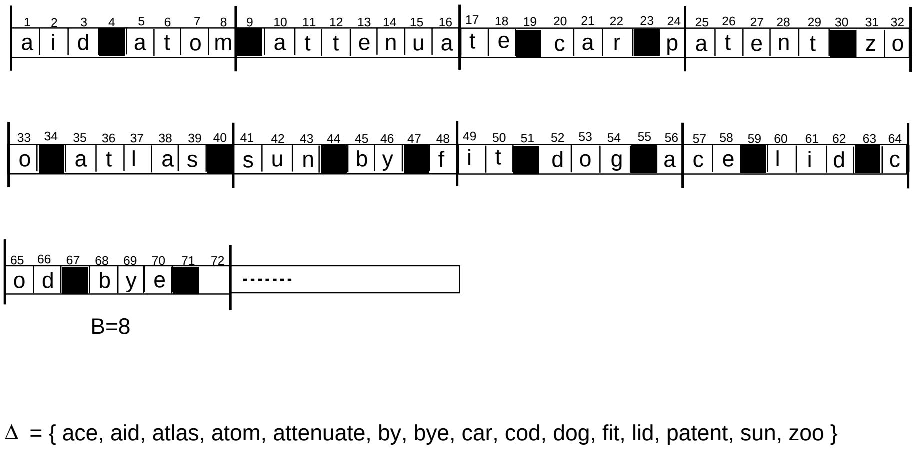
B=8
Δ = { ace, aid, atlas, atom, attenuate, by, bye, car, cod, dog, fit, lid, patent, sun, zoo }

Figure 1. An example of storing string set Δ in external memory. The strings are not put in any particular order. Disk is represented by a linear array with disk page size $B = 8$. The logical pointers to Δ's strings are their starting positions in external memory. For example, 48 is the logical pointer to string 'fit' and 14 is the logical pointer to suffix 'nuate'. The black boxes in the disk pages denote special endmarkers that prevent two suffixes of Δ's strings from being equal.

Consequently, we believe that a proper organization of the strings and a method for avoiding rescanning are crucial to solve Problems 1 and 2 with provably good performance in the worst case, and we show how to do this in the rest of this section.

We begin by describing a B-tree-like data structure that helps us to solve Problem 1 by handling keys which are logical pointers to arbitrarily-long strings. Since the worst-case bounds obtained are not the ones claimed in the introduction, we perform another step and transform the B-tree-like data structure into a *simplified version* of the String B-tree by properly organizing the logical pointers inside its nodes by means of Patricia tries. This combination is described in Section 2.1, where we introduce new structural properties that allow us to design a search procedure which avoids the rescanning problem previously mentioned, thus showing how to solve Problem 1 efficiently. Finally, we show in Section 2.2 how to obtain the *final version* of the String B-tree for solving Problem 2 by adding some extra pointers and proving further properties that are crucial to achieve our bounds.

2.1 Prefix Search and Range Query (Problem 1)

We start out by describing a B-tree-like data structure which gives us an initial, rough solution to Problem 1. As previously stated, we represent strings by their logical pointers. The input is a string set Δ whose total number of characters is N. We denote by $\mathcal{K} = \{K_1, \ldots, K_k\}$ the set of Δ's strings in lexicographic order, denoted by $\leq_L$. We assume that strings $K_1, \ldots, K_k$ reside in the B-tree leaves, which are linked together to form a bidirectional list, and only some strings are copied in the internal nodes—we obtain the so-called B$^+$-tree [15]. We denote the ordered string set associated with a node π by $\mathcal{S}_\pi$, where $\mathcal{S}_\pi \subseteq \mathcal{K}$, and denote $\mathcal{S}_\pi$'s leftmost string by $L(\pi)$ and $\mathcal{S}_\pi$'s rightmost string by $R(\pi)$. We store each node π in a single disk page and put a constraint on the number of its strings:

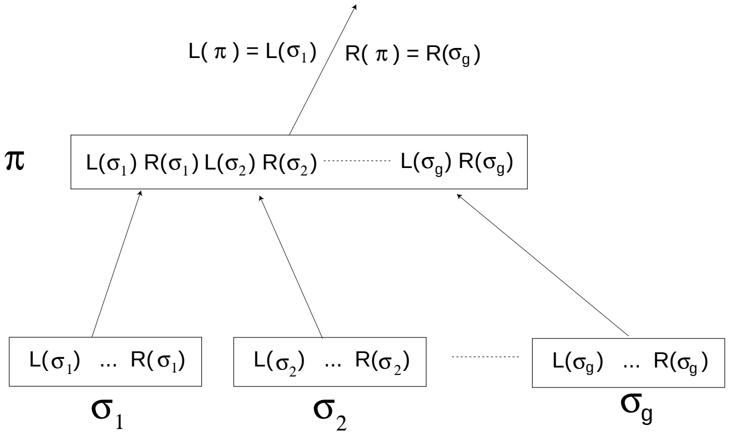

Figure 2. The logical layout of a B-tree internal node π having $g = n(\pi)$ children.

$b \leq |\mathcal{S}_\pi| \leq 2b$, where $b = \Theta(B)$ is an even integer properly chosen to let a single node fit into a disk page. We allow the root to contain less than b strings.

We distribute the strings among the B-tree nodes as follows: We partition $\mathcal{K}$ into groups of b consecutive strings each, except for the last group which can contain from b to $2b$ strings. We map each group to a leaf, say π, and form its string set $\mathcal{S}_\pi$ in such a way that we can retrieve $\mathcal{K}$ by scanning the leaves rightwards and by concatenating their string sets. Each internal node π has $n(\pi)$ children $\sigma_1, \ldots, \sigma_{n(\pi)}$ and its ordered string set $\mathcal{S}_\pi = \{L(\sigma_1), R(\sigma_1), \ldots, L(\sigma_{n(\pi)}), R(\sigma_{n(\pi)})\}$ is obtained by copying the leftmost and rightmost strings contained in its children, as shown in Figure 2. (Actually, we could only copy one string from each child but this would make our algorithms more complex.) Since $n(\pi) = \frac{|\mathcal{S}_\pi|}{2}$, each node has from $\frac{b}{2}$ to b children except for the root and the leaves, and the resulting number of B-tree levels is $H = O(\log_{b/2} k) = O(\log_B k)$. We call H its *height*, and number these levels by starting from the root (level 1). See Figure 3 for an example.

Problem 1 can be solved by using the B-tree-like layout described above. We only discuss the Prefix Search(P) operation in detail. It is based on an interesting observation introduced by Manber and Myers [33]: *the strings having prefix P occupy a contiguous part of $\mathcal{K}$*. In the example described in Section 1, the strings having prefix P = 'at' all range from string 'atlas' to string 'attenuate'. Consequently, we only have to retrieve $\mathcal{K}$'s leftmost and rightmost strings whose prefix is P because the rest of the strings to be retrieved lie in $\mathcal{K}$ between these two strings. In our case, these strings occupy a contiguous sequence of B-tree leaves—i.e., the ones storing the logical pointers 35, 5 and 10 in Figure 3. In another observation of theirs, Manber and Myers identify the leftmost string whose prefix is P: *this string is adjacent to P's position in $\mathcal{K}$ according to the lexicographic order $\leq_L$*. In the example given in Section 1, if P = 'at', its position in $\mathcal{K}$ is between strings 'aid' and 'atlas'; in fact, 'atlas' is the leftmost string we are looking for. A symmetrical observation holds for the rightmost string and so we do not discuss it here. Since $\mathcal{K}$ is a dynamic set partitioned among the B-tree leaves, we can use Manber and Myers' observations in our B-tree-like layout. We therefore answer Prefix Search(P) by focusing on the retrieval of P's position in $\mathcal{K}$. We represent P's position in the whole set $\mathcal{K}$ by means of a pair (τ, j),

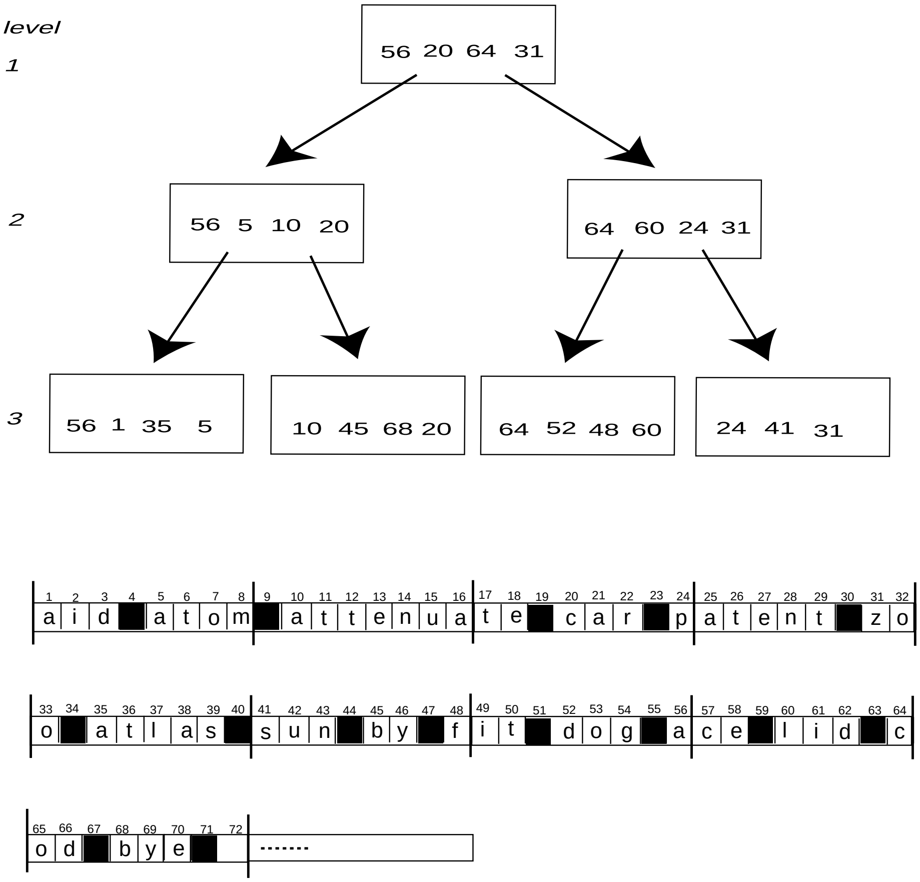

Figure 3. An example of B-tree-like layout (upper part) and its input string set Δ (lower part). Set $\mathcal{K} = \{$'ace', 'aid', 'atlas', 'atom', 'attenuate', 'by', 'bye', 'car', 'cod', 'dog', 'fit', 'lid', 'patent', 'sun', 'zoo'$\}$ is obtained by sorting Δ. The strings in $\mathcal{K}$ are stored in the B-tree leaves by means of their logical pointers 56, 1, 35, 5, 10, …, 31.

such that τ is the leaf containing this position and $j-1$ is the number of $\mathcal{S}_\tau$'s strings lexicographically smaller than P, where $1 \leq j \leq |\mathcal{S}_\tau|+1$. We also say that j is P's position in set $\mathcal{S}_\tau$. In our example for P = 'at', τ is the leftmost leaf in Figure 3 and $j = 3$, where $\mathcal{S}_\tau$ is made up of the strings pointed by 56, 1, 35 and 5.

In Figure 4, we illustrate the algorithmic scheme for identifying pair (τ, j), where we denote the procedure that determines P's position in a set $\mathcal{S}_\pi$ by PT-Search(P, $\mathcal{S}_\pi$). We begin by checking the two trivial cases in which P is either smaller than any other string in $\mathcal{K}$ (Step (1)) or larger than any other string in $\mathcal{K}$ (Step (2)). If both checks turn out to be false, we start out from $\pi = root$ in Step (3) and perform a downward B-tree traversal by maintaining the *invariant*: $L(\pi) <_L P \leq_L R(\pi)$ for each node π visited (Steps (4)–(8)). In visiting π, we load its disk page and apply procedure PT-Search in order to find P's position j in string set $\mathcal{S}_\pi$, namely, we determine its two adjacent strings verifying $\widehat{K}_{j-1} <_L P \leq_L \widehat{K}_j$. If π is a leaf, we stop the traversal. If π is an internal node, we have the following two cases:

(1) If strings $\widehat{K}_{j-1}$ and $\widehat{K}_j$ belong to two distinct children of π, say $\widehat{K}_{j-1} = R(\sigma')$ and

```
if $P \leq_L K_1$ then $\tau :=$ leftmost leaf; $j := 1$; return$(\tau, j)$;
if $P >_L K_k$ then $\tau :=$ rightmost leaf; $j := |\mathcal{S}_\tau| + 1$; return$(\tau, j)$;
$\pi := root$;
while true do /* Invariant: $L(\pi) <_L P \leq_L R(\pi)$ */
    Load $\pi$'s page and let $\mathcal{S}_\pi = \{\widehat{K}_1, \ldots, \widehat{K}_{2n(\pi)}\}$;
    $j :=$ PT-Search($P$, $\mathcal{S}_\pi$);      /* $\widehat{K}_{j-1} <_L P \leq_L \widehat{K}_j$ */
    if $\pi$ is a leaf then $\tau := \pi$; return$(\tau, j)$;
    if $\widehat{K}_j = L(\sigma)$, for a child $\sigma$ of $\pi$ then
                  $\tau := \sigma$'s leftmost descending leaf; $j := 1$; return$(\tau, j)$;
    if $\widehat{K}_j = R(\sigma)$, for a child $\sigma$ of $\pi$ then $\pi := \sigma$;
endwhile
```

Figure 4. The pseudocode for identifying pair (τ, j) that represents P's position in $\mathcal{K}$.

$\widehat{K}_j = L(\sigma)$ for two children σ' and σ, then the two strings are *adjacent* in the whole set $\mathcal{K}$ due to B-tree's layout. This determines P's position in $\mathcal{K}$. We therefore choose τ as the leftmost B-tree leaf that descends from σ and conclude that P is in the first position in $\mathcal{S}_\tau$ because $L(\tau) = L(\sigma) = \widehat{K}_j$.

(2) If both $\widehat{K}_{j-1}$ and $\widehat{K}_j$ belong to the same child, say $\widehat{K}_{j-1} = L(\sigma)$ and $\widehat{K}_j = R(\sigma)$ for a child σ, then we set $\pi := \sigma$ in order to maintain the invariant and continue the B-tree traversal on the next level recursively.

At the end of this traversal, we find the pair (τ_L, j_L) that represents the position of $\mathcal{K}$'s leftmost string having prefix P. In the same way, we can determine the pair (τ_R, j_R) that represents the position of $\mathcal{K}$'s rightmost string having prefix P. We go on to answer Prefix Search(P) by scanning the linked sequence of B-tree leaves delimited by τ_L and τ_R (inclusive) and by listing all the strings from the (j_L)-th string in $\mathcal{S}_{\tau_L}$ up to the $(j_R - 1)$-th string in $\mathcal{S}_{\tau_R}$.

The search described so far is similar to the one used for regular B-trees, especially if we implement procedure PT-Search by performing a binary search of P in set $\mathcal{S}_\pi$ and examining $O(\log_2 |\mathcal{S}_\pi|) = O(\log_2 B)$ strings. While this binary search does not cost anything more in regular B-trees, in this case, once we load π's disk page, we have to pay $O(\frac{p}{B} + 1)$ disk accesses to *load each string examined and compare it* to P because we represent the strings by their logical pointers. Consequently, a call to PT-Search takes $O((\frac{p}{B} + 1) \log_2 B)$ disk accesses in the worst case. It follows that this simple approach for Prefix Search calls PT-Search H times and thus takes a total of $O(H\,(\frac{p}{B} + 1) \log_2 B) = O((\frac{p}{B} + 1) \log_2 k)$ disk accesses plus $O(\frac{occ}{B})$ disk accesses for retrieving the strings delimited by leaves τ_L and τ_R. This bound is the same as for the suffix array search without any auxiliary data structures [33], and worse than the one we claimed in the introduction. Nevertheless, the B-tree-like layout gives us a good starting point for finding an efficient implementation of Prefix Search.

We now carry out another step in the B-tree-like layout by plugging a *Patricia trie* [36] into each B-tree node in order to organize its strings properly and support searches that

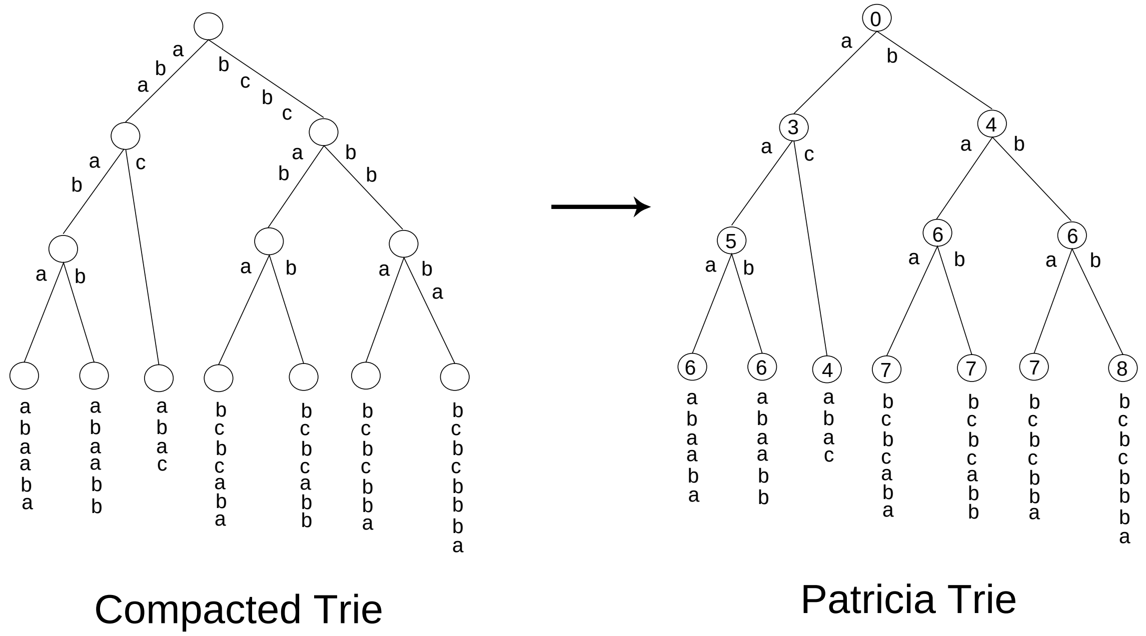

Figure 5. The number labeling an internal node u denotes the length of the string spelled out by the downward path from the root to u.

compare *only one string of set* $\mathcal{S}_\pi$ *in the worst-case rather than the* $\log_2 |\mathcal{S}_\pi|$ *ones* required for a binary search. We call the resulting data structure the *simplified String B-tree.* [2]

Let us examine a node π in the String B-tree and the Patricia trie PT_π plugged into it. We can define PT_π in two steps: (1) We build a compacted trie [29] on $\mathcal{S}_\pi$'s strings (see Figure 5, left). (2) We label each compacted trie node by the length of the substring stored into it and we replace each substring labeling an arc by its first character only, called *branching character* (see Figure 5, right). On one hand, the Patricia trie loses some information with respect to the compacted trie because we delete all the characters in each arc label except the branching character. On the other hand, the Patricia trie has two important features that we discuss below: (i) it fits $\Theta(B)$ strings into one B-tree node independently of their length; (ii) it allows to perform lexicographic searches by branching out from a node without further disk accesses. It is worth noting that a compacted trie might satisfy feature (i) by representing the substrings labeling its arcs via pairs of pointers to their external-memory positions; however, feature (ii) would be no longer satisfied because of the pairs of pointers and this would increase the number of disk accesses taken by the search operation.

We now show how to exploit some new properties of Patricia tries for implementing the PT-Search procedure in two phases. Due to its features, hereafter we will call this search procedure *blind search*:

---

[2] We were not able to find any source in the research literature referring to a data structure based on B-trees and Patricia tries for internal nodes, and resembling the simplified String B-tree. Probably some programmers know such a data structure. Nonetheless, we highlight new structural properties that are crucial to achieve optimal worst-case bounds for Problem 1.

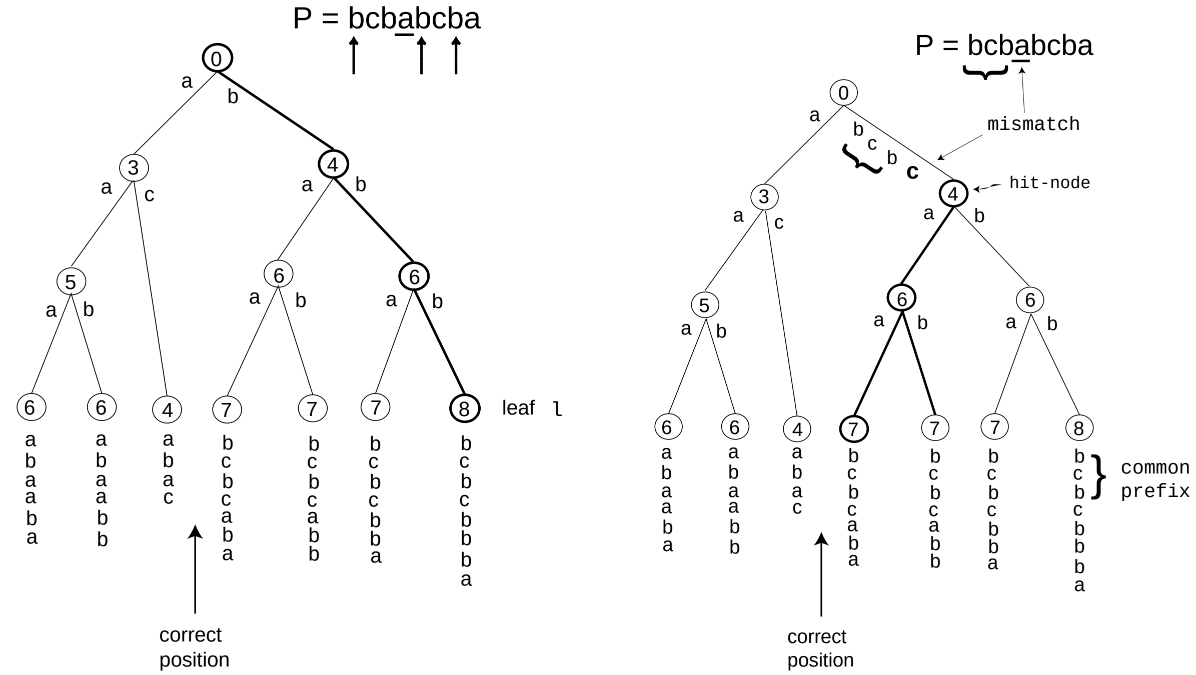

Figure 6. (Left) An example of the first phase in blind search. The marked arcs are the traversed ones. (Right) An example of the second phase in blind search. The hit node is circled, and the marked arcs are the ones traversed to find P's position in $\mathcal{S}_\pi$.

- In the first phase, we trace a downward path in PT_π to locate a leaf l, which does not necessarily identify P's position in $\mathcal{S}_\pi$. We start out from the root and only compare some of P's characters with the branching characters found in the arcs traversed until we either reach a leaf, say l, or no further branching is possible. In the latter case, we choose l to be a descending leaf from the last node traversed.
- In the second phase, we load l's string and compare it to P in order to determine their common prefix. We prove a useful property (Lemma 3.5): *Leaf l stores one of $\mathcal{S}_\pi$'s strings that share the* **longest** *common prefix with P.* We use this common prefix in two ways: we first determine l's shallowest ancestor (the *hit node*) whose label is an integer equal to, or greater than, the common prefix length of l's string and P. We then find P's position by using P's mismatching character to choose a proper Patricia trie leaf descending from the hit node.

We give an example of PT-Search(P, $\mathcal{S}_\pi$) in Figure 6, where P = 'bcbabcba'. In particular, Figure 6(left) depicts the first phase in which l represents the rightmost leaf. It is worth noting that l does not identify P's position in $\mathcal{S}_\pi$ because we do not compare P's mismatching character (i.e., $P[4]$ = 'a') and thus we induce a "mistake." We determine P's correct position in the second phase, illustrated in Figure 6(right). We start out by determining the common prefix of l's string and P (i.e., 'bcb') and then we find l's shallowest ancestor (the hit node) whose label is greater than $|\text{'bcb'}| = 3$. After that, we use mismatching character $P[4]$ = 'a' to identify P's correct position $j = 4$ by traversing the marked arcs in Figure 6(right). It is worth noting that we only load the disk pages that store prefix 'bcbc' in l's string because the Patricia trie is stored in π's disk page, thus making the branching characters available. In this way, we do not take more than $O(\frac{p}{B}+1)$ disk accesses to execute PT-Search.

It is now clear that putting Patricia tries and the previously described B-tree layout together, we avoid the binary search in the nodes traversed and thus reduce the overall complexity from $O((\frac{p}{B}+1)\log_2 k)$ to $O((\frac{p}{B}+1)H) = O((\frac{p}{B}+1)\log_B k)$ disk accesses. However, this bound is yet not satisfactory and does not match the one claimed in the introduction. The reason is that at each visited node we are rescanning P from the beginning. We avoid rescanning and obtain the final optimal bound by designing an improved PT-Search procedure that derives directly from the previous one but exploits the String B-tree layout and the Patricia trie properties better. It takes *three* input parameters $(P, \mathcal{S}_\pi, \ell)$, where the additional input parameter ℓ satisfies the property that there is a string in $\mathcal{S}_\pi$ whose first ℓ characters are equal to P's. PT-Search$(P, \mathcal{S}_\pi, \ell)$ returns pair (j, lcp), where j is P's position in $\mathcal{S}_\pi$ (as before) and the additional output parameter lcp is the common prefix length of l's string and P computed in the blind search. A comment is in order at this point. We can show that $lcp \geq \ell$ (see Lemma 3.6) and can therefore design a *fast incremental* PT-Search that compares P to l's string by only loading and examining *the characters in positions* $\ell+1, \ldots, lcp+1$. As a result, PT-Search now only takes $\lceil\frac{lcp-\ell}{B}\rceil + 1$ disk accesses (see Theorem 3.8).

We now go back to the algorithmic scheme for finding P's position in the whole set $\mathcal{K}$. The above considerations allow us to modify the pseudocode in Figure 4 by adding instruction $\ell := 0$ to Step (3) and by replacing Step (5) with:

(5) $(j, \ell) :=$ PT-Search$(P, \mathcal{S}_\pi, \ell)$

We are now ready to analyze Prefix Search's complexity. As previously mentioned, we have to search for $\mathcal{K}$'s leftmost and rightmost strings having prefix P by identifying the pairs (τ_L, j_L) and (τ_R, j_R). We do this by means of our modified pseudocode which traverses a sequence of nodes, say $\pi_1, \pi_2, \ldots, \pi_H$. The cost of examining π_i is dominated by Step (5), which takes $d_i = \lceil\frac{\ell_i - \ell_{i-1}}{B}\rceil + 1 \leq \frac{\ell_i - \ell_{i-1}}{B} + O(1)$ disk accesses because we execute PT-Search with $\ell = \ell_{i-1}$ to compute $lcp = \ell_i$. The total cost of this traversal is $\sum_{i=1}^{H} d_i = \frac{\ell_H - \ell_0}{B} + O(H) = O(\frac{p}{B} + \log_B k)$ disk accesses. We use the fact that it is a telescopic sum, where $\ell_0 = 0$, $\ell_H \leq p$ and $H = O(\log_B k)$. Subsequently, we retrieve $\mathcal{K}$'s strings having prefix P by examining the leaves of the String B-tree delimited by τ_L and τ_R in $O(\frac{occ}{B})$ disk accesses. The total cost of Prefix Search(P) is therefore $O(\frac{p+occ}{B} + \log_B k)$ disk accesses. We refer the reader to Section 4.1 for a detailed, formal discussion of this result.

The simplified String B-tree layout has the considerable advantage of being *dynamic* without requiring any contiguous space. A new string K can be inserted into Δ like regular B-trees, that is, by inserting K into $\mathcal{K}$ in lexicographic order. We identify K's position in $\mathcal{K}$ by computing its pair (τ, j). We then insert K into string set $\mathcal{S}_\tau$ at position j. If $L(\tau)$ or $R(\tau)$ change in τ, then we extend the change to τ's ancestors. After that, if τ gets full (i.e., it contains more than $2b$ strings), we say that a *split* occurs. We create a new leaf σ and install it as an adjacent sibling of τ. We then split string set $\mathcal{S}_\tau$ into two roughly equal parts of at least b strings each, in order to obtain τ's and σ's new string sets. We copy strings $L(\tau), R(\tau), L(\sigma)$ and $R(\sigma)$ in their parent node in order to replace the old strings $L(\tau)$ and $R(\tau)$. If τ's parent also gets full because it has two more strings, we split it. In the worst case, the splitting can extend up to the String B-tree's root and the resulting String B-tree's height can increase by one.

The deletion of a string from Δ is similar to its insertion, except that we are faced with a leaf that gets half-full because it has less than b strings. In this case, we say that a *merge* occurs and we join this leaf and an adjacent sibling leaf together: we merge their string sets and propagate the merging to their ancestors. In the worst case, the merging can extend up to the String B-tree's root and so the height can decrease by one.

The cost for inserting or deleting a string is given by its searching cost plus the $O(\log_B k)$ rebalancing cost. We can prove the following result:

Theorem 2.1 (Problem 1). *Let Δ be a set of k strings whose total length is N.* Prefix Search(P) *takes $O(\frac{p+occ}{B} + \log_B k)$ worst-case disk accesses, where $p = |P|$.* Range Query(K', K'') *takes $O(\frac{k'+k''+occ}{B} + \log_B k)$ worst-case disk accesses, where $k' = |K'|$ and $k'' = |K''|$. Inserting or deleting a string of length m takes $O(\frac{m}{B} + \log_B k)$ worst-case disk accesses. The space occupied by the String B-tree built on Δ is $\Theta(\frac{k}{B})$ disk pages and the space required by string set Δ is $\Theta(\frac{N}{B})$ disk pages.*

2.2 Substring Search (Problem 2)

We now show how solve Problem 2, in which the input is a string set $\Delta = \{\delta_1, \ldots, \delta_k\}$ whose total number of characters is $N = \sum_{h=1}^{k} |\delta_h|$. We denote the suffix set by $SUF(\Delta) = \{\delta[i, |\delta|] : 1 \leq i \leq |\delta| \text{ and } \delta \in \Delta\}$, which therefore contains N lexicographically ordered suffixes. As previously mentioned, Problem 2 concerns with a more powerful Substring Search(P) operation that searches for P's occurrences in Δ's strings, i.e., it finds all the length-p substrings equal to P. Since each of these occurrences corresponds to a suffix whose prefix is P—i.e., $\delta[i, i+p-1] = P$ if and only if P is a prefix of $\delta[i, |\delta|] \in SUF(\Delta)$—our problem is actually to retrieve all of $SUF(\Delta)$'s strings having prefix P. We therefore turn a Substring Search(P) on string set Δ into a Prefix Search(P) on suffix set $SUF(\Delta)$. For example, let us examine the String B-tree shown in Figure 7 and search for P = 'at'. We have to retrieve $occ = 5$ occurrences: '**at**las','**at**om', '**at**tenu**at**e' and 'p**at**ent'. The suffixes having prefix P and corresponding to these occurrences have their logical pointers (i.e., 16, 25, 35, 5 and 10) stored in a contiguous sequence of leaves in Figure 7. As a result, we can set the string set $\mathcal{K} = SUF(\Delta)$ and its size $k = N$ and execute Prefix Search(P). The total cost of answering Substring Search(P) is therefore $O(\frac{p+occ}{B} + \log_B N)$ worst-case disk accesses by Theorem 2.1.

Although this transformation notably simplifies the search operation, it introduces some updating problems that represent the most challenging part of solving Problem 2. We wish to point out that the insertion of an individual string Y into string set Δ, where $m = |Y|$, consists of inserting all of its m suffixes into suffix set $SUF(\Delta)$ in lexicographic order. Consequently, we could consider inserting one suffix at a time, say $Y[i, m]$, with $d_i = O(\frac{|Y[i,m]|}{B} + \log_B N)$ disk accesses by Theorem 2.1 with $\mathcal{K} = SUF(\Delta)$ and $k = N$. The total insertion cost would be $\sum_{i=1}^{m} d_i = O(m\,(\frac{m}{B}+1) + m \log_B(N+m))$ disk accesses and this is worse than the $O(m \log_B(N+m))$ worst-case bound we claimed in the introduction. The

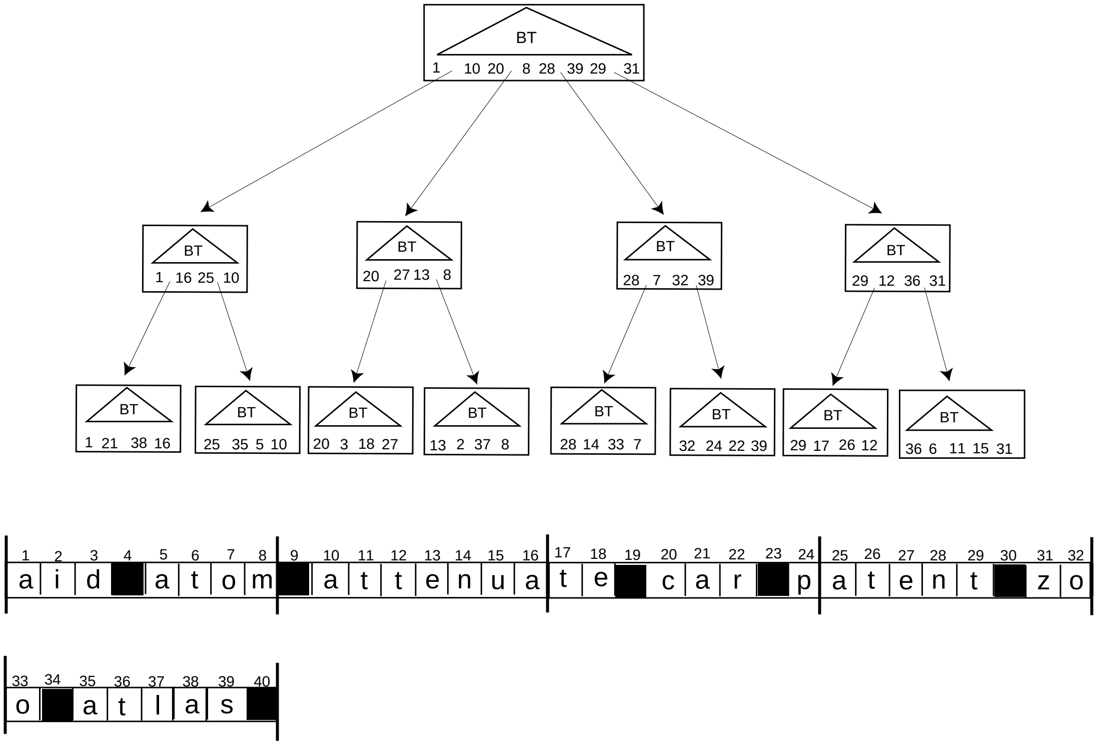

Figure 7. An example of an String B-tree layout for solving Problem 2 on string set $\Delta = \{$‘aid’, ‘atlas’, ‘atom’, ‘attenuate’, ‘car’, ‘patent’, ‘zoo’$\}$. Here, $b = 4$ and $\mathcal{K} = \{$‘aid’,‘ar’,‘as’, …, ‘uate’,‘zoo’ $\}$.

problem here is that we treat the m inserted suffixes like arbitrary strings and this causes the *rescanning* problem. The solution lies in the fact that they are all part of the same string. Consequently, we *augment* the simplified String B-tree by introducing two types of auxiliary pointers which help us to avoid rescanning in the updating process: One type is the standard *parent* pointer defined for each node; the other is the *succ* pointer defined for each string in $SUF(\Delta)$ as follows. *The succ pointer for* $\delta[i, |\delta|] \in SUF(\Delta)$ *leads to String B-tree's leaf containing* $\delta[i+1, |\delta|]$. If $i = |\delta|$, then we let *succ* be a self-loop pointer to its own leaf, i.e., the leaf containing $\delta[i, |\delta|]$. We only describe the logic behind Y's insertion here because its deletion is simpler, and treat the subject formally in Sections 4.2–4.5.

We insert Y's suffixes into the String B-tree storing $SUF(\Delta)$ at the beginning, going from the longest to the shortest one. We proceed by induction on $i = 1, 2, \ldots, m$ and make sure that we satisfy the following two conditions *after* $Y[i, m]$'s insertion:

(a) Suffixes $Y[j, m]$ are stored in the String B-tree, for all $1 \le j \le i$, and $Y[i, m]$ shares its first h_i characters with one of its adjacent strings in the String B-tree.

(b) All the *succ* pointers are correctly set for the strings in the String B-tree except for $Y[i, m]$. This means that $succ(Y[i, m])$ is the only dangling pointer, unless $i = m$, in which case it is a self-loop pointer to its own leaf.

We refer the reader to the self-explanatory pseudocode illustrated in Figure 8 for further details. We assume that Conditions (a) and (b) are satisfied for $i - 1$. By executing

```
procedure SB-Insert($Y$);

        $m := |Y|$;
        for $i = 1, 2, \ldots, m$ do
(1)         find the leaf $\pi_i$ that contains $Y[i,m]$'s position;
(2)         insert $Y[i,m]$ into $\pi_i$;
(3)         if a split occurs then rebalance the String B-tree;
                                   redirect some $succ$ and $parent$ pointers;
(4)         $succ(Y[i-1,m])$ := leaf containing $Y[i,m]$;
(5)         if $i = m$ then $succ(Y[i,m]) := succ(Y[i-1,m])$;     /* self-loop pointer */
        endfor
```

Figure 8. The insertion algorithm.

Steps (1)–(5), we make $succ(Y[i,m])$ be the new dangling pointer and satisfy Conditions (a) and (b) for i. We therefore go on by setting $i := i + 1$ and repeat the insertion for the next suffix of Y. The two main problems arising in the implementation of the insertion procedure are:

- Step (1): We have to find $Y[i,m]$'s position without any rescanning.
- Step (3): We have to rebalance the updated String B-tree by redirecting some *succ* and *parent* pointers efficiently.

We now examine the problem of finding $Y[i,m]$'s position (Step (1)). For $i = 1$, we find $Y[1,m]$'s position by traversing the String B-tree analogously to Prefix Search($Y[1,m]$). We take a different approach for the rest of Y's suffixes ($i > 1$) to avoid rescanning and inductively exploit Conditions (a) and (b) for $i - 1$. When finding $Y[i,m]$'s position, instead of starting out from the root, we traverse the String B-tree from the last leaf visited in the String B-tree (i.e., the one containing $Y[i-1,m]$). Since $Y[i-1,m] = Y[i-1]\,Y[i,m]$, we would be tempted to use the $succ(Y[i-1,m])$ pointer to identify $Y[i,m]$'s position directly but cannot because the pointer is dangling by Condition (b). However, we know that $Y[i-1,m]$ shares its first h_{i-1} characters with one of its adjacent strings by Condition (a). *We therefore take the succ-pointer of this adjacent string*, which is correctly set by Condition (b), and reach a leaf which verifies the following property: *it contains a string that shares the first* $\max\{0, h_{i-1} - 1\}$ *characters with* $Y[i,m]$ (Lemma 4.8). We continue the insertion by performing an upward and downward String B-tree traversal leading to leaf π_i, which contains $Y[i,m]$'s position. Since we can prove that $h_i \geq \max\{0, h_{i-1} - 1\}$ (Corollary 4.9), our algorithm avoids rescanning by only examining Y's characters in positions $i + \max\{0, h_{i-1} - 1\}, \ldots, i + h_i$. We show that this "double" String B-tree traversal correctly identifies π_i with $\frac{h_i - \max\{0, h_{i-1}-1\}}{B} + O(\log_B(N + m))$ disk accesses (Lemma 4.10).

After $Y[i,m]$'s insertion in its leaf, we have to rebalance the String B-tree if a split occurs (Step (3)). A straightforward handling of *parent* and *succ* pointers would take $O(B \log_B(N + m))$ worst-case disk accesses per inserted suffix because: (i) each node split operation can redirect $\Theta(B)$ of these pointers from possibly distinct nodes; (ii) there can be $H = O(\log_B(N+m))$ split operations per inserted suffix. In Section 4.5, we show how to obtain an $O(\log_B(N+m))$ amortized cost per suffix and then devise a general strategy based on node *clusters* to achieve $O(\log_B(N+m))$ in the worst case.

As far as the worst-case complexity of $Y[i,m]$'s insertion is concerned, we take $d_i = \frac{h_i - \max\{0, h_{i-1}-1\}}{B} + O(\log_B(N+m))$ disk accesses, where $h_0 = 0$ and $h_i \leq m$. As a result, a total of $\sum_{i=1}^{m} d_i = O(\frac{m}{B} + m \log_B(N+m)) = O(m \log_B(N+m))$ disk accesses are required for inserting Y into Δ. It is worth noting that we achieve the same worst-case performance as for the insertion of m integer keys into a regular B-tree; but additionally, our bound is proportional to the number of inserted suffixes rather than their total length, which is bounded by $\Theta(m^2)$. We give a formal, detailed discussion of the update operations in Sections 4.2–4.5 and prove the following result:

Theorem 2.2 (Problem 2). *Let Δ be a set of strings whose total length is N.* Substring Search(P) *takes $O(\frac{p+occ}{B} + \log_B N)$ worst-case disk accesses, where $p = |P|$. Inserting a string of length m in Δ or deleting it takes $O(m \log_B(N+m))$ worst-case disk accesses. The space occupied by both the String B-tree and the string set Δ amounts to $\Theta(\frac{N}{B})$ disk pages.*

We begin our formal discussion with a technical description of the Patricia trie data structure and its operations (Section 3). We then give a technical description of the String B-tree data structure and discuss its operations in detail (Section 4).

3 A Technical Description of Patricia Tries

We let Σ denote an ordered alphabet and $\leq_L$ denote the lexicographic order among the strings whose characters are taken from Σ. Given two strings X and Y that are not each other's prefix, we define $lcp(X,Y)$ to be their longest common prefix length, i.e., $lcp(X,Y) = k$ iff $X[1,k] = Y[1,k]$ and $X[k+1] \neq Y[k+1]$. This definition can be extended to the case in which X is Y's prefix (or vice versa) by appending a special endmarker to both strings. The following fact illustrates the relationship between the lexicographic order $\leq_L$ and the *lcp* value:

Fact 3.1. *For any strings X_1, X_2, Y such that either $X_1 \leq_L X_2 \leq_L Y$ or $Y \leq_L X_2 \leq_L X_1$: $lcp(X_1,Y) \leq lcp(X_2,Y)$.*

Let us now consider an ordered string set $\mathcal{S} = \{X_1, \ldots, X_d\}$ and assume that any two strings in $\mathcal{S}$ are not each other's prefix. We use the shorthand $max\_lcp(Y,\mathcal{S})$ to indicate the maximum among the *lcp*-values of Y and $\mathcal{S}$'s strings, i.e., $max\_lcp(Y,\mathcal{S}) = \max_{X \in \mathcal{S}} lcp(Y,X)$. We say that an integer j is Y's *position in set* $\mathcal{S}$ if exactly $(j-1)$ strings in $\mathcal{S}$ are lexicographically smaller than Y, where $1 \leq j \leq d+1$. The following fact illustrates the relationship between the *max_lcp* value and $\mathcal{S}$'s strings near Y's position:

Fact 3.2. *If j is Y's position in $\mathcal{S}$, then*

$$max\_lcp(Y,\mathcal{S}) = \begin{cases} lcp(Y, X_1) & \text{if } j = 1 \\ \max\{lcp(X_{j-1}, Y), lcp(Y, X_j)\} & \text{if } 2 \leq j \leq d \\ lcp(X_d, Y) & \text{if } j = d+1. \end{cases}$$

We introduce a definition of Patricia tries that is slightly different from the one in [36], but it is suitable for our purposes. A *Patricia trie* $PT_{\mathcal{S}}$ built on $\mathcal{S}$ satisfies the following conditions (see Figure 5):

(1) Each arc is labeled by a branching character taken from Σ and each internal node has at least two outgoing arcs labeled with different characters. The arcs are ordered according to their branching characters and only the root can have one child.

(2) There is a distinct leaf v associated with each string in $\mathcal{S}$. We denote this string by $W(v)$. Leaf v also stores its string length $len(v) = |W(v)|$.

(3) If node u is the *lowest* common ancestor of two leaves l and f, then it is labeled by integer $len(u) = lcp(W(l), W(f))$ (and we let $len(root) = 0$). Specifically, leaf l (resp., f) descends from u's outgoing arc whose branching character is the $(len(u) + 1)$-st character in string $W(l)$ (resp., $W(f)$).

Let us now consider an internal node u in $PT_{\mathcal{S}}$ and denote u's parent by $parent(u)$; we let f be one of u's descending leaves. Property (3) suggests that we denote the string implicitly stored in node u by $W(u)$, that is, $W(u)$ is equal to the first $len(u)$ characters of $W(f)$. Arc $(parent(u), u)$ *implicitly* corresponds to a substring of length $(len(u) - len(parent(u)))$ having its first character equal to the branching character $W(f)[len(parent(u)) + 1]$ and the other characters equal to $W(f)$'s characters in positions $len(parent(u)) + 2, \ldots, len(u)$. We can now introduce the definition of *hit node* that is the analog of the *extended locus* notion in compacted tries [34]:

Definition 3.3. *The* hit node *for a pair* (f, ℓ), *such that* f *is a leaf and* $0 < \ell \leq len(f)$, *is* f*'s ancestor* u *satisfying:* $len(u) \geq \ell > len(parent(u))$. *If* $\ell = 0$, *the hit node is the root.*

Patricia tries do not take up very much space: $PT_{\mathcal{S}}$ has d leaves and no more than d internal nodes because only the root can have one child. Therefore, the total space required is $O(d)$ even if the total length of $\mathcal{S}$'s strings can be much more than d.

3.1 Blind Searching in Patricia Tries: PT-Search procedure

We propose a search method that makes use of Patricia trie $PT_{\mathcal{S}}$ to efficiently retrieve the position of an arbitrary string P in an ordered set $\mathcal{S}$. We stated the intuition and logic behind it in Section 2.1 (PT-Search procedure). PT-Search's input is a triplet $(P, \mathcal{S}, \ell)$, where $\ell \leq lcp(P, X)$ for a string $X \in \mathcal{S}$. The output is a pair (j, lcp) in which j is P's position in $\mathcal{S}$ and $lcp = max\_lcp(P, \mathcal{S})$. Let us introduce a special character \$ smaller than any other character in Σ and let us assume without any loss in generality that $P[i] = \$$ when $i > |P|$. We implicitly use the following fact to identify $\mathcal{S}$'s leftmost string whose prefix is P (we can also determine its rightmost one by letting \$ be larger than any other alphabet character).

Fact 3.4. *There is a mismatch between* P *and any other string and, if any of* $\mathcal{S}$*'s strings have prefix* $P[1, |P|]$, *then* P*'s position in* $\mathcal{S}$ *is to their immediate left.*

There are two main phases in our procedure:

First Phase: Downward Traversal. We locate a leaf, say l, by traversing $PT_{\mathcal{S}}$ downwards. We start out from its root and compare P's characters with the branching characters of the arcs traversed. If u is the currently visited node and has an outgoing arc (u, v) whose branching character is equal to $P[len(u) + 1]$, then we move from u to its child v and set $u := v$. We go on like this until we either reach a leaf, which is l, or we cannot branch any further and then choose l as one of u's descending leaves. Leaf l stores one of $\mathcal{S}$'s strings that satisfy the following useful property:

Lemma 3.5. *If we let lcp denote* $lcp(W(l), P)$, *then* $lcp = max\_lcp(P, \mathcal{S})$.

Proof: By way of contradiction, we assume that there is another string X in $\mathcal{S}$, such that $X \neq W(l)$ and $lcp(X, P) > lcp$, and show that we cannot reach leaf l. We have $lcp(W(l), X) = lcp$. Let u denote the lowest common ancestor of l and the leaf storing X. From Property (3) of the Patricia tries, it follows that $len(u) = lcp(W(l), X)$ and $P[len(u)+1] = X[len(u)+1] \neq W(l)[len(u)+1]$ (because $lcp(X, P) > lcp$). Consequently, $W(u)$ is a proper prefix of P and we can branch further out from u to its child v by matching $P[len(u) + 1]$ with branching character $X[len(u) + 1]$. Since the branching character is different from $W(l)[len(u)+1]$, we obtain the contradiction that v is not one of l's ancestors and therefore l cannot be reached at the end of the downward traversal. $\square$

It is worth noting that we retrieve leaf l without performing any disk accesses because we only use the branching characters stored in $PT_{\mathcal{S}}$'s disk page. Furthermore, l's position does not necessarily correspond to P's position in $\mathcal{S}$ (see Figure 6(left)).

Second Phase: Retrieval of P's position in $\mathcal{S}$. We compute $lcp = lcp(W(l), P)$ and the two mismatching characters $c = P[lcp+1]$ and $c' = W(l)[lcp+1]$ (which are well-defined by Fact 3.4) by exploiting the following result:

Lemma 3.6. $lcp \geq \ell$.

Proof: We know that there is a string $X \in \mathcal{S}$, such that $lcp(P, X) \geq \ell$. Moreover, $max\_lcp(P, \mathcal{S}) \geq lcp(P, X)$ by definition. Since $lcp = max\_lcp(P, \mathcal{S})$ by Lemma 3.5, we deduce that $lcp \geq \ell$. $\square$

From Lemma 3.6, we deduce that the first ℓ characters in P and $W(l)$ are definitely equal. We therefore compute lcp, c and c' *by starting out from the* $(\ell+1)$*-st characters in* P *and* $W(l)$ *rather than from their beginning*. Consequently, we only retrieve $\lceil \frac{lcp-\ell}{B} \rceil + 1$ disk pages, namely the ones storing substring $W(l)[\ell + 1, lcp + 1]$. We then detect the hit node, say u, for the pair (l, lcp) by traversing the Patricia trie upwards and find P's position j in $\mathcal{S}$ by using the property that all of $\mathcal{S}$'s strings having prefix $P[1, lcp]$ are stored in u's descending leaves.

Lemma 3.7. *We can compute P's position j without any further disk accesses.*

Proof: We already have lcp, c and c' in main memory. We handle two cases on hit node u and derive their correctness from the Patricia trie properties:

(1) Case $len(u) = lcp$. We let $c_1, \ldots, c_k$ be the branching characters in u's outgoing arcs. None of them match character c. If $c <_L c_1$, then we move to u's leftmost descending leaf z and let $j-1$ be the number of leaves to z's left (z excluded). If $c_k <_L c$, then we move to u's rightmost descending leaf z and let $j-1$ be the number of leaves to z's left (z included). In all other cases, we determine two branching characters, say c_i and c_{i+1}, such that $c_i <_L c <_L c_{i+1}$. We move to the leftmost leaf z that is reachable through the arc labeled c_{i+1} and let $j-1$ be the number of leaves to z's left (z excluded).

(2) Case $len(u) > lcp$. We can infer that all the strings stored in u's descending leaves share the same prefix of length $len(u)$ and we know that $len(u) > lcp > len(parent(u))$. The $(lcp+1)$-st character of them all is equal to c' because l is one of u's descending leaves. If $c <_L c'$, then we move to u's leftmost descending leaf z and let $j-1$ be the number of leaves to z's left (z excluded). If $c' <_L c$, then we move to u's rightmost descending leaf z and let $j-1$ be the number of leaves to z's left (z included). $\square$

It is worth noting that the computation of lcp, c and c' is the only expensive step in the second phase. We can therefore state the following, basic result:

Theorem 3.8. *Let us assume that Patricia trie $PT_{\mathcal{S}}$ is already in main memory and let ℓ be a non-negative integer such that $\ell \leq lcp(X, P)$ for a string $X \in \mathcal{S}$.* PT-Search$(P, \mathcal{S}, \ell)$ *returns the pair (j, lcp) in which j is P's position in $\mathcal{S}$ and $lcp = max\_lcp(P, \mathcal{S})$. It does not cost more than $\lceil \frac{lcp-\ell}{B} \rceil + 1$ disk accesses.*

Proof: The correctness follows from Lemmas 3.5 and 3.7. We now analyze the total number of disk accesses. In the first phase, we do not make any disk accesses and we perform no more than $2d$ character comparisons, as this is the number of branching characters in $PT_{\mathcal{S}}$. In the second phase, we do not require any more than $\lceil \frac{lcp-\ell}{B} \rceil + 1$ disk accesses to compute lcp, c, c' and $O(lcp - \ell + 1)$ character comparisons. Finally, we do not have to make any more disk accesses or more than d character comparisons to determine hit node u and position j. $\square$

3.2 Dynamic operations on Patricia Tries

We now describe how to maintain Patricia tries under concatenate, split, insert and delete operations. These operations will be useful to us further on.

PT-Concatenate$(PT_{\mathcal{S}_1}, PT_{\mathcal{S}_2}, lcp, c, c')$

We let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two ordered string sets, such that $\mathcal{S}_1$'s strings are lexicographically smaller than $\mathcal{S}_2$'s strings. If X is $\mathcal{S}_1$'s rightmost string and Y is $\mathcal{S}_2$'s leftmost string, then the last three input parameters must satisfy $lcp = lcp(X, Y)$, $c = X[lcp+1]$ and $c' = Y[lcp+1]$ (with $c <_L c'$). We use PT-Concatenate to concatenate Patricia tries $PT_{\mathcal{S}_1}$ and $PT_{\mathcal{S}_2}$ in order to create a single Patricia trie $PT_{\mathcal{S}_1 \cup \mathcal{S}_2}$ whose ordered set $\mathcal{S}_1 \cup \mathcal{S}_2$ is obtained by appending $\mathcal{S}_2$'s strings to $\mathcal{S}_1$'s. We build $PT_{\mathcal{S}_1 \cup \mathcal{S}_2}$ by merging $PT_{\mathcal{S}_1}$'s rightmost path with $PT_{\mathcal{S}_2}$'s leftmost path.

Lemma 3.9. PT-Concatenate *makes no disk accesses if its input parameters are in main memory.*

Proof: For technical reasons, we assume that $PT_{\mathcal{S}_1}$ and $PT_{\mathcal{S}_2}$ have a *dummy* root (with $len(dummy) = -1$) connected to their original roots by a *dummy* arc labeled with a *null* character. We examine the nodes on $PT_{\mathcal{S}_1}$'s rightmost path, say $\alpha_1, \ldots, \alpha_j$, in which $W(\alpha_j) = X$, and the nodes on $PT_{\mathcal{S}_2}$'s leftmost path, say $\beta_1, \ldots, \beta_i$, in which $W(\beta_i) = Y$.

We let α_{k+1} be the hit node for (α_j, lcp) and β_{h+1} be the hit node for (β_i, lcp), where $k \in [1, j-1]$ and $h \in [1, i-1]$ (see Definition 3.3). We deduce that the first lcp characters in strings $X, Y, W(\alpha_{k+1})$ and $W(\beta_{h+1})$ are equal. Consequently, strings $W(\alpha_r)$ and $W(\beta_s)$ are one the other's prefix for every $r \in [1, k]$ and $s \in [1, h]$, and thus the (branching) characters in positions $len(\alpha_r) + 1$ and $len(\beta_s) + 1$ are equal in strings $W(\alpha_{k+1})$ and $W(\beta_{h+1})$ (since $len(\alpha_k) < lcp$ and $len(\beta_h) < lcp$ by Definition 3.3). We base the concatenation of $PT_{\mathcal{S}_1}$ to $PT_{\mathcal{S}_2}$ on this and merge paths $\alpha_1, \ldots, \alpha_j$ and $\beta_1, \ldots, \beta_i$ in the following two steps:

In the first step, we concentrate on the sequence $[\alpha_1, c'_1] \ldots [\alpha_k, c'_k]$, in which each pair consists of a node α_r and the branching character c'_r labeling arc (α_r, α_{r+1}), for $r = 1, \ldots, k$. We also consider the similarly-defined sequence $[\beta_1, c''_1] \ldots [\beta_h, c''_h]$. We merge these two sequences into a new one, say , $= [\gamma_1, c_1], \ldots, [\gamma_m, c_m]$ (where $m \leq k + h$), according to the integers $len(\alpha_r)$ and $len(\beta_s)$ ($r = 1, \ldots, k$ and $s = 1, \ldots, h$). If we get a tie while merging (i.e., $len(\alpha_r) = len(\beta_s)$), we turn α_r and β_s into a single node γ_i in , because they must also have the same branching character (see above). We then build a path in final tree $PT_{\mathcal{S}_1 \cup \mathcal{S}_2}$ by scanning sequence , : For each pair $[\gamma_i, c_i]$, we connect γ_i to γ_{i+1} by means of an arc (γ_i, γ_{i+1}), whose label is c_i. We then concatenate all of γ_i's children (except child γ_{i+1}) taken from $PT_{\mathcal{S}_1}$ or $PT_{\mathcal{S}_2}$ (or maybe both). For the last pair $[\gamma_m, c_m]$ in , , we create a dangling arc e whose label is c_m and then go on to the second step.

In the second step, we treat hit nodes α_{k+1} and β_{h+1}. We create a node u that is attached to e and we set $len(u) = lcp$. We make u be the parent of both α_{k+1} and β_{h+1} and label their two incoming arcs with characters c and c', respectively. We make a last check to make sure that there are no one-child nodes. That is, we check to see if $len(u) = len(\alpha_{k+1})$. If this relation holds, then arc (u, α_{k+1}) is contracted by uniting u and α_{k+1} (and their children). We make the same check for β_{h+1}.

The correctness derives from Patricia trie's properties and from the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ are ordered string sets. The whole computation requires $O(d)$ character comparisons (because $|\mathcal{S}_1|$ and $|\mathcal{S}_2|$ are $O(d)$) and no disk accesses because $PT_{\mathcal{S}_1}$ and $PT_{\mathcal{S}_2}$ are assumed to be already in main memory. □

PT-Split$(PT_{\mathcal{S}}, X_j)$

The input is a Patricia trie $PT_{\mathcal{S}}$ and a string $X_j \in \mathcal{S}$, where $\mathcal{S} = \{X_1, \ldots, X_d\}$. The output is given by two Patricia tries $PT^l_{\mathcal{S}}$ and $PT^r_{\mathcal{S}}$ built on string sets $\{X_1, \ldots, X_j\}$ and $\{X_{j+1}, \ldots, X_d\}$, respectively. A split can be considered as the inverse operation of PT-Concatenate$(PT^l_{\mathcal{S}}, PT^r_{\mathcal{S}}, \ldots)$.

Lemma 3.10. PT-Split *makes no disk accesses if its input parameters are in main memory.*

Proof: We identify $PT_{\mathcal{S}}$'s leaf z such that $W(z) = X_j$ and duplicate $PT_{\mathcal{S}}$ by creating two temporary copies $PT^l_{\mathcal{S}}$ and $PT^r_{\mathcal{S}}$. We then start out from z and walk upwards in $PT^l_{\mathcal{S}}$ in such a way that, for each node u visited, we delete all the nodes that descend from u's *right siblings*. If we produce one-child nodes, then we contract their arcs. A similar computation is performed in $PT^r_{\mathcal{S}}$, except that we delete both u and all the nodes descending from u's *left siblings*. The two resulting trees are the Patricia tries built on string sets $\{X_1, \ldots, X_j\}$ and $\{X_{j+1}, \ldots, X_d\}$. The whole computation requires $O(d)$ character comparisons because $|\mathcal{S}| = O(d)$. We make no disk accesses because $PT_{\mathcal{S}}$ is assumed to be already in main memory. □

PT-Insert$(X, PT_{\mathcal{S}}, Z, lcp, c_X, c_Z)$

PT-Insert adds string X to string set $\mathcal{S}$ by maintaining its lexicographic order. We create a leaf f, such that $W(f) = X$, and install f in $PT_{\mathcal{S}}$ to the left of the leaf storing the input string $Z \in \mathcal{S}$. The last three input parameters must satisfy $lcp = lcp(X, Z)$, $c_X = X[lcp+1]$ and $c_Z = Z[lcp + 1]$.

Lemma 3.11. PT-Insert *makes no disk accesses if its input parameters are in main memory.*

Proof: We identify leaf l, such that $W(l) = Z$, and hit node u for (l, lcp). If $len(u) = lcp$, we install leaf f as a child of u and label arc (u, f) with character c_X. If $lcp < len(u)$, then we remove arc $(parent(u), u)$ and its branching character, say $\widehat{c}$. We create another node v and set $len(v) = lcp$. We install v as a child of $parent(u)$ and label $(parent(u), v)$ with $\widehat{c}$. Then we make f and u be v's children and label arcs (v, f) and (v, u) with c_X and c_Z, respectively. Each arc is installed according to its branching character's order. We make no disk accesses because $PT_{\mathcal{S}}$ is assumed to be already in main memory. □

PT-Delete$(X, PT_{\mathcal{S}})$

PT-Delete removes string X from set $\mathcal{S}$ by identifying the leaf $f \in PT_{\mathcal{S}}$ such that $W(f) = X$ and by removing f and maybe its parent, if f becomes its only child. The operation makes no disk accesses if its input parameters are in main memory.

Theorem 3.12. *Let us assume that the Patricia tries and the other input parameters are already in main memory:* PT-Concatenate, PT-Split, PT-Insert *and* PT-Delete *require $O(d)$ character comparisons but no disk accesses.*

4 A Technical Description of the String B-Tree

We now go into the technical details concerning the String B-tree data structure (see Section 2 for our introductory concepts and notation, and Figure 2). Specifically, we let $\mathcal{K} = \{K_1, \ldots, K_k\}$ denote the set of Δ's strings in increasing lexicographic order. As previously stated, we represent $\mathcal{K}$'s strings by their logical pointers and we obtain $\mathcal{K}$ from string set Δ in Problem 1 and from suffix set $SUF(\Delta)$ in Problem 2. We prevent any two strings in $\mathcal{K}$ from being each other's prefix by appending a *distinct endmarker* to each of them. This makes sure that we can correctly build the Patricia trie on any subset $\mathcal{S}_\pi$ of $\mathcal{K}$'s strings. We denote $\mathcal{S}_\pi$'s leftmost and rightmost strings by $L(\pi)$ and $R(\pi)$, respectively.

Each **leaf** π of the String B-tree stores an ordered string set $\mathcal{S}_\pi \subseteq \mathcal{K}$ (where $b \leq |\mathcal{S}_\pi| \leq 2b$) and the Patricia trie PT_π built on $\mathcal{S}_\pi$. We recall here that $\mathcal{S}_\pi$ is obtained by partitioning $\mathcal{K}$ among the leaves in such a way that a left-to-right scanning of them gives the whole set $\mathcal{K}$. Leaf π is augmented with the following information:

(a) Two pointers $next(\pi)$ and $prev(\pi)$ to define the doubly-linked list of leaves in the String B-tree.

(b) The longest common prefix length of $R(prev(\pi))$ and $L(\pi)$ and of $R(\pi)$ and $L(next(\pi))$ together with their mismatching characters.

(c) The *succ* pointers and their inverse $succ^{-1}$ pointers for $\mathcal{S}_\pi$'s strings and the *parent* pointer for π. This information is only introduced for solving Problem 2, when $\mathcal{K}$ is obtained from $SUF(\Delta)$. We remember that $succ(\delta[i, |\delta|])$ points to the leaf containing suffix $\delta[i+1, |\delta|]$. If $i = |\delta|$ then *succ* points to π itself, hence it is a self-loop pointer.

Each **internal node** π of the String B-tree has $n(\pi)$ children $\sigma_1, \ldots, \sigma_{n(\pi)}$ and contains an ordered string set $\mathcal{S}_\pi = \{L(\sigma_1), R(\sigma_1), \ldots, L(\sigma_{n(\pi)}), R(\sigma_{n(\pi)})\}$ obtained by copying the leftmost and the rightmost strings from its children (actually, we could copy only one string from each child but this would make our algorithms more complex). Since $n(\pi) = \frac{|\mathcal{S}_\pi|}{2}$, each node has from $\frac{b}{2}$ to b children (except for the root, in which $2 \leq n(root) \leq b$). Node π also contains the Patricia trie PT_π built on $\mathcal{S}_\pi$ and, when treating Problem 2, it contains:

(c') The *parent* pointer for π.

We have to make sure that a node of the String B-tree (containing Patricia trie, pointers, etc.) can be stuffed into a single disk page so that the occupied space does not exceed disk page size B. We therefore choose a proper value for $b = \Theta(B)$ and the resulting height is $H = O(\log_B k)$. The following simple, useful properties hold:

Property 4.1. Among π's descending leaves, $L(\pi)$ is the lexicographically smallest string and $R(\pi)$ is the lexicographically largest string.

Proof: If π is a leaf, the property clearly follows. Otherwise, we use induction and the fact that π's leftmost (resp., rightmost) child σ satisfies $L(\sigma) = L(\pi)$ (resp., $R(\sigma) = R(\pi)$). See Figure 2. □

Property 4.2. For any two adjacent strings K_i and K_{i+1} in $\mathcal{K}$, we have their longest common prefix length $lcp(K_i, K_{i+1})$ and their two mismatching characters.

Proof: If both strings belong to the same string set of a leaf in the String B-tree, say π, we take their two corresponding Patricia trie leaves $l_1, l_2 \in PT_\pi$ and find their lowest common ancestor u. By Property (3) of Patricia tries (Section 3), we deduce that $len(u) = lcp(K_i, K_{i+1})$ and the mismatching characters are the two branching characters belonging to u's outgoing arcs that lead to l_1 and l_2, respectively. On the contrary, if both two strings belong to distinct leaves in the String B-tree, say π and σ, we obtain that $K_i = R(\pi)$, $K_{i+1} = L(\sigma)$ and $next(\pi) = \sigma$. We therefore find $lcp(K_i, K_{i+1})$ in π's page (or σ's page) because π stores $lcp(R(\pi), L(next(\pi)))$ together with their mismatching characters by Points (a) and (b) at the beginning of this section. $\square$

4.1 Searching in String B-trees

We use the String B-tree built on string set $\mathcal{K} = \{K_1, \ldots, K_k\}$ for searching an arbitrary pattern string. We stated the intuition behind String B-tree searching in Section 2.1 and described its algorithmic scheme in Figure 4. The input is a pattern P, where $p = |P|$, and the output is the pair (τ, j) identifying P's position in the whole set $\mathcal{K}$, where τ is the leaf of the String B-tree containing this position and j is P's position in string set $\mathcal{S}_\tau$. We now extend this scheme by means of a more powerful searching tool, which will be also used in String B-tree updating. Specifically, we provide a procedure SB-Search-Down(P, π, ℓ) whose input parameters satisfy Condition-A below and whose output is a triplet (τ, j, lcp), such that (τ, j) identifies P's position in $\mathcal{K}$ (as before) and the extra output parameter $lcp = max\_lcp(P, \mathcal{K})$ is the length of P's longest prefix in common with any of $\mathcal{K}$'s strings. We have:

Condition-A(P, π, ℓ)**:** Let π be a node of the String B-tree and ℓ be a non-negative integer:

(A1) There is definitely one of π's strings whose first ℓ characters are equal to P's. That is, $\ell \leq lcp(X, P)$ for a string $X \in \mathcal{S}_\pi$.

(A2) One of π's descending leaves contains P's position in $\mathcal{K}$. That is, $L(\pi) <_L P \leq_L R(\pi)$.

Condition-A1(P, π, ℓ) helps us to avoid rescanning. Condition-A2(P, π, ℓ) states that P's position in $\mathcal{K}$ cannot be to $L(\pi)$'s left or to $R(\pi)$'s right (Fact 3.1) so that SB-Search-Down can find (τ, j) by traversing the String B-tree downward from π. The pseudocode of SB-Search-Down is illustrated in Figure 9 and easily derives from the one shown in Figure 4. We can prove that:

Lemma 4.3. *Let us take a node π of the String B-tree and an integer $\ell \geq 0$ that satisfy Condition-A(P, π, ℓ).* SB-Search-Down(P, π, ℓ) *returns triplet (τ, j, lcp), where τ is the leaf containing P's position in $\mathcal{K}$, j is P's position in string set $\mathcal{S}_\tau$ and $lcp = max\_lcp(P, \mathcal{K})$. It costs $\frac{lcp - \ell}{B} + O(\log_B k)$ disk accesses.*

```
procedure SB-Search-Down($P$, $\pi$, $\ell$);

    while true do
(1)     Load $\pi$'s disk page and let $\mathcal{S}_\pi = \{\widehat{K}_1, \ldots, \widehat{K}_{2n(\pi)}\}$;
(2)     $(j, \ell)$ := PT-Search($P$, $\mathcal{S}_\pi$, $\ell$);      /* $\widehat{K}_{j-1} <_L P \leq_L \widehat{K}_j$ */
(3)     if $\pi$ is a leaf then $\tau := \pi$; $lcp := \ell$; return ($\tau$, $j$, $lcp$);
(4)     if $\widehat{K}_j = L(\sigma)$, for a child $\sigma$ of $\pi$ then
            $\tau := \sigma$'s leftmost descending leaf; $j := 1$; $lcp := \ell$; return ($\tau$, $j$, $lcp$);
(5)     if $\widehat{K}_j = R(\sigma)$, for a child $\sigma$ of $\pi$ then $\pi := \sigma$;
    endwhile
```

Figure 9. The pseudocode for finding triplet (τ, j, ℓ) when Condition-A(P, π, ℓ) holds.

Proof: Without any loss in generality, we assume that $P[i] = \$$ when $i > p$, so that Fact 3.4 holds for $\mathcal{K}$ (i.e., there is definitely a mismatch between P and any other string and if some of $\mathcal{K}$'s strings have a prefix $P[1, p]$, then P's position is to their left.) We refer to the algorithmic scheme given in Figure 9. We first prove that, in Steps (1)–(5), we either identify triplet (τ, j, lcp) or find a child σ of π that maintains Condition-A. In the latter case, we go deeper into the String B-tree by setting $\pi := \sigma$.

Let us examine string set $\mathcal{S}_\pi$ and number its strings in lexicographic order: $\mathcal{S}_\pi = \{\widehat{K}_1, \ldots, \widehat{K}_{2n(\pi)}\}$. We can consider $\mathcal{K}$'s strings to be partitioned into three intervals $(-\infty, \widehat{K}_1)$, $[\widehat{K}_1, \widehat{K}_{2n(\pi)}]$ and $(\widehat{K}_{2n(\pi)}, \infty)$, where intervals $(-\infty, \widehat{K}_1)$ and $(\widehat{K}_{2n(\pi)}, \infty)$ contain all of $\mathcal{K}$'s strings that are strictly smaller than $\widehat{K}_1$ or larger than $\widehat{K}_{2n(\pi)}$, and interval $[\widehat{K}_1, \widehat{K}_{2n(\pi)}]$ contains the strings stored in π's descending leaves. Since we know that $\widehat{K}_1 <_L P \leq_L \widehat{K}_{2n(\pi)}$ by Condition-A2(P, π, ℓ), P's position in $\mathcal{K}$ is inside $[\widehat{K}_1, \widehat{K}_{2n(\pi)}]$ and τ is one of π's descending leaves.

When we load π's page in Step (1), we can refine the partition of $\mathcal{K}$'s strings into intervals

$$(-\infty, \widehat{K}_1), [\widehat{K}_1, \widehat{K}_2], [\widehat{K}_3, \widehat{K}_4], \ldots, [\widehat{K}_{2n(\pi)-1}, \widehat{K}_{2n(\pi)}], (\widehat{K}_{2n(\pi)}, \infty)$$

by Property 4.1. Since Patricia trie PT_π is available in π's disk page, we use Condition-A1(P, π, ℓ) and execute the blind search in $\mathcal{S}_\pi$ by means of PT-Search$(P, \mathcal{S}_\pi, \ell)$ (in Step (2)). This procedure returns P's position j in $\mathcal{S}_\pi$ (i.e., its two adjacent strings verifying $\widehat{K}_{j-1} <_L P \leq_L \widehat{K}_j$) and integer $max\_lcp(P, \mathcal{S}_\pi)$ assigned to ℓ (by Theorem 3.8). At this point, we feel that a comment is in order. Since Fact 3.2 states that equality $max\_lcp(P, \mathcal{S}_\pi) = \max\{lcp(\widehat{K}_{j-1}, P), lcp(\widehat{K}_j, P)\}$ holds, we can deduce that $max\_lcp(P, \mathcal{S}_\pi) = max\_lcp(P, \mathcal{K})$ when strings $\widehat{K}_{j-1}$ and $\widehat{K}_j$ are also adjacent in $\mathcal{K}$. In this situation, we can safely set $lcp := \ell$ because $\ell = max\_lcp(P, \mathcal{S}_\pi)$ and we want parameter lcp to be $max\_lcp(P, \mathcal{K})$ by definition.

In this way, if π is a leaf (Step (3)), we stop searching because $\mathcal{S}_\pi$ is a contiguous part of $\mathcal{K}$ and thus $\tau := \pi$ and $lcp := \ell$.

If π is an internal node, instead, we have to examine two other cases:

(a) $\widehat{K}_j = L(\sigma)$ for a child σ of π (Step (4)). We deduce that P's position is just between intervals $[\widehat{K}_{j-2}, \widehat{K}_{j-1}]$ and $[\widehat{K}_j, \widehat{K}_{j+1}]$ and thus we set τ to be σ's leftmost descending leaf because of Property 4.1. We know that P's position in $\mathcal{S}_\tau$ is the first one because $L(\tau) = L(\sigma) = \widehat{K}_j$. We can set $j := 1$ and $lcp := \ell$.

(b) $\widehat{K}_j = R(\sigma)$ for a child σ of π (Step (5)). This implies that $\widehat{K}_{j-1} = L(\sigma)$ and so P's position in $\mathcal{K}$ is inside interval $[\widehat{K}_{j-1}, \widehat{K}_j]$. We can reduce $\mathcal{K}$'s partition to only three intervals $(-\infty, \widehat{K}_{j-1}), [\widehat{K}_{j-1}, \widehat{K}_j], (\widehat{K}_j, \infty)$ and therefore make Condition-A2(P, σ, ℓ) hold. Condition-A1(P, σ, ℓ) also holds: We know that $\widehat{K}_{j-1}$ and $\widehat{K}_j$ belong to both $\mathcal{S}_\pi$ and $\mathcal{S}_\sigma$ because of the String B-tree layout (see Figure 2) and we must have $\ell = lcp(X, P)$ for an $X \in \{\widehat{K}_{j-1}, \widehat{K}_j\} \subseteq \mathcal{S}_\sigma$. We can therefore set $\pi := \sigma$ and go on in the **while** loop because Condition-A(P, σ, ℓ) holds.

We eventually reach leaf τ by a simple induction on $\pi = \pi_i, \pi_{i+1}, \ldots, \pi_H$, where i is π's level in the String B-tree and H is the height of the String B-tree.

We now analyze the total number of disk pages retrieved. As we go deeper into the String B-tree, we extend P's matched prefix without rescanning its previously examined characters. Consequently, the sequence of values, say $\ell_i \leq \ell_{i+1} \leq \cdots \leq \ell_H$, computed by PT-Search in Step (2) is non-decreasing because of Lemma 3.6. In a generic String B-tree level $s \geq i$, we only need *one disk access* to retrieve node π_s from external memory, and no more than $\lceil \frac{\ell_s - \ell_{s-1}}{B} \rceil + 1$ disk accesses to execute PT-Search$(P, \mathcal{S}_{\pi_s}, \ell_{s-1})$ and compute pair (τ, j) by Theorem 3.8. Since ℓ_{i-1} is input parameter ℓ and ℓ_H is output parameter lcp of SB-Search-Down, its total number of disk accesses does not exceed $\sum_{s=i}^{H} (\lceil \frac{\ell_s - \ell_{s-1}}{B} \rceil + 2) \leq \frac{\ell_H - \ell_{i-1}}{B} + 3H \leq \frac{lcp - \ell}{B} + O(\log_B k)$. □

We are now ready to state our first result:

Theorem 4.4. Prefix Search(P) *can be implemented with* $O(\frac{p+occ}{B} + \log_B k)$ *worst-case disk accesses.*

Proof: We recall that $\mathcal{K}$ is the sorted sequence of Δ's strings. We performed the Prefix Search operation in Section 2.1 by a two-phase procedure in which we first retrieved the leaves π_L and π_R, containing respectively $\mathcal{K}$'s leftmost and rightmost strings whose prefix was P, and then scanned all of the leaves lying between them by using the *next*-pointers. We can now implement Prefix Search(P) by only retrieving π_L and therefore avoid traversing the String B-tree twice.

First of all, we find P's position in $\mathcal{K}$. We check the two trivial cases in which either $P \leq_L K_1$ or $K_k <_L P$ with $O(\frac{p}{B})$ disk accesses by a direct character-by-character comparison. In the former case, we set τ as the leftmost leaf in the String B-tree, $j := 1$ and lcp as the longest prefix matched by direct comparison; in the latter case, we stop searching because there are no occurrences. If both conditions are false (i.e., $K_1 <_L P \leq_L K_k$), we execute SB-Search-Down$(P, root, 0)$. Since Condition-A$(P, root, 0)$ trivially holds, SB-Search-Down correctly returns triplet $(\tau,\, j,\, lcp)$ and takes $O(\frac{p}{B} + \log_B k)$ disk accesses (because $lcp \leq p$; see Lemma 4.3).

We then list all of $\mathcal{K}$'s strings whose prefix is P. Let K_{pos} be the string in $\mathcal{K}$ that occupies position j in $\mathcal{S}_\tau$. We check to see if $lcp \geq p$ (otherwise, there are no occurrences, by Fact 3.1). If it does, then K_{pos} is an occurrence and so we examine $K_{pos}, \ldots, K_{pos+occ-1}$

by verifying that the common prefix length of two adjacent strings K_i and K_{i+1} is at least p. We do not need to make any direct comparisons because $lcp(K_i, K_{i+1})$ is available in the current disk page (Property 4.2). We only access a contiguous part of leaves in the String B-tree and retrieve all the *occ* occurrences with $O(\frac{occ}{B})$ disk accesses because at least $b = \Theta(B)$ suffixes are contained in each accessed leaf (except the first and the last one). □

Corollary 4.5. Range Query(K', K'') *takes* $O(\frac{k'+k''+occ}{B} + \log_B k)$ *worst-case disk accesses, where* $k' = |K'|$ *and* $k'' = |K''|$. Substring Search(P) *takes* $O(\frac{p+occ}{B} + \log_B N)$ *worst-case disk accesses, where* $p = |P|$.

Proof: Range Query(K', K'') can be implemented by searching the positions of K' and K'' in set $\mathcal{K}$ with $O(\frac{k'+k''}{B} + \log_B k)$ disk accesses (by Lemma 4.3) and by listing all of $\mathcal{K}$'s strings lying between K' and K''. It costs $O(\frac{occ}{B})$ disk accesses.

Substring Search(P) can be implemented by letting $\mathcal{K} = SUF(\Delta)$ and $k = N$ and by executing Prefix Search(P) (by Theorem 4.4). □

The search bounds stated in Theorem 4.4 and Corollary 4.5 are asymptotically **optimal** for a large alphabet Σ whose characters can only be accessed by comparisons. We prove the lower bound for the Prefix Search(P) operation by using the *external-memory pointer machine*, introduced in [42] with the aim of generalizing the pointer machine [43] to external memory. This also holds for Range Query and Substring Search operations. In the external-memory pointer machine, a data structure is seen as a graph with a source vertex s. Each vertex is a disk page of size B, which contains no more than B items (i.e., characters, polynomially-bounded integers, or pointers) and can be linked to no more than B vertices. Given a pattern P, a searching algorithm must start from s and traverse the graph according to the following restrictions: (a) Any vertex except s can be accessed only if a vertex leading to it has already been accessed. (b) For each occurrence, at least one vertex has to be accessed. (c) A link among accessed vertices can be changed dynamically, provided that the number of outgoing links in a vertex does not exceed B.

Lemma 4.6. *For a large alphabet* Σ *whose characters can only be accessed by comparisons, searching for a pattern* P *and listing all of its occ occurrences in* $\mathcal{K}$*'s strings requires* $\Omega(\frac{p}{B} + max\{\frac{occ}{B}, \log_B k\})$ *worst-case disk accesses in external memory.*

Proof: We examine the case in which P contains p distinct characters and each of these characters appears at least once in one of $\mathcal{K}$'s strings. We deduce that at least $\lceil \frac{p}{B} \rceil$ pages must be accessed to verify that P is actually an occurrence by a simple adversary argument. As a result, the occurrences must all be stored somewhere explicitly and the total number of accessed pages to list all of them has to be at least $\lceil \frac{occ}{B} \rceil$ because a page can maintain no more than B items. Finally, searching for P is at least as difficult as finding its first character $P[1]$. Since Σ is general, the retrieval of the strings whose first character is $P[1]$ requires $\Omega(\log_B |\Sigma|)$ disk accesses because the graph has maximum vertex degree B. The lower bound follows by letting Σ verify $\log |\Sigma| = \Omega(\log k)$. □

```
procedure SB-Search-Up-Down($P$, $\pi$, $\ell$);
    /* upward traversal */
    $first := 1$;    /* first position */
    while true do
        load $\pi$'s page;
        $last := |\mathcal{S}_\pi| + 1$;    /* last position in $\mathcal{S}_\pi$ */
        $(j, \ell) :=$ PT-Search($P, \mathcal{S}_\pi, \ell$);
        if $j = first$ and $L(\pi) = K_1$ then
            $\tau :=$ the leftmost leaf in the String B-tree; $lcp := \ell$; return ($\tau$, $j$, $lcp$);
        else if $j = last$ and $R(\pi) = K_k$ then
            $\tau :=$ the rightmost leaf in the String B-tree; $lcp := \ell$; return ($\tau$, $j$, $lcp$);
        else if $first < j < last$ then exit-while;
        $\pi := parent(\pi)$;
    endwhile
    /* downward traversal */
    $(\tau, j, lcp) :=$ SB-Search-Down($P, \pi, \ell$);
    return ($\tau$, $j$, $lcp$).
```

Figure 10. The pseudocode for finding triplet (τ, j, lcp) when only Condition-A1(P, π, ℓ) holds.

4.2 More About Searching

SB-Search-Down(P, π, ℓ) is the fundamental procedure for searching in String B-trees and is based on Condition-A(P, π, ℓ). We now discuss what happens if we remove Condition-A2 (i.e., $L(\pi) <_L P \leq_L R(\pi)$) and use only Condition-A1 (i.e., there is an integer ℓ, such that $\ell \leq lcp(X, P)$ for a string $X \in \mathcal{S}_\pi$). In other words, when we load π's page, we assume that we know that the first ℓ characters of P are shared by one of $\mathcal{S}_\pi$'s strings, but *we are no longer sure that P's position is in one of π's descending leaves in the String B-tree*. The investigation of this case will be useful to us when we design the insertion procedure for Problem 2.

We extend procedure SB-Search-Down(P, π, ℓ) to a new procedure which we call SB-Search-Up-Down(P, π, ℓ), whose input parameters *satisfy Condition-A1(P, π, ℓ) only* and whose *output is the same triplet* (τ, j, lcp) as SB-Search-Down's, where pair (τ, j) identifies P's position in $\mathcal{K}$ and $lcp = max\_lcp(P, \mathcal{K})$. The main feature is that we now traverse the String B-tree twice: first we go *upwards* by means of a new search procedure and by only maintaining Condition-A1. We stop this traversal as soon as Condition-A2 is satisfied and then we traverse the String B-tree *downwards* by executing SB-Search-Down because Condition-A holds. The pseudocode is illustrated in Figure 10 where $\mathcal{K} = \{K_1, \ldots, K_k\}$ is the ordered string set.

Lemma 4.7. *Let us take a node π in the String B-tree and an integer $\ell \geq 0$, such that Condition-A1(P, π, ℓ) is satisfied.* SB-Search-Up-Down(P, π, ℓ) *returns the triplet (τ, j, lcp), where τ is the leaf in the String B-tree containing P's position in $\mathcal{K}$, j is P's position in string set $\mathcal{S}_\tau$ and $lcp = max\_lcp(P, \mathcal{K})$. It costs $\frac{lcp-\ell}{B} + O(\log_B k)$ disk accesses.*

Proof: The aim of the two-phase String B-tree traversal (Figure 10) is to increase the number of P's matched characters without having to perform rescanning. Let us assume that $P \leq_L R(\pi)$ without any loss in generality (the case $L(\pi) <_L P$ is identical); thus, leaf τ is to the left of π (included).

In the first phase (Steps (2)–(10)), we start from π and traverse the String B-tree upwards until we detect a node that is the *lowest common ancestor* of π (known) and τ (unknown). We prove the correctness of this phase by showing that the **while** loop preserves *both* Condition-A1(P, π, ℓ) and disequality $P \leq_L R(\pi)$ when we assign the new values to π and ℓ. Indeed, Steps (2)–(4) find P's position j in $\mathcal{S}_\pi$ and compute value $\ell = max\_lcp(P, \mathcal{S}_\pi)$ to keep track of the number of P's characters matched by the PT-Search procedure (by Theorem 3.8). At this point, we are faced with the problem of deciding if we have to move upwards in the String B-tree or if the current node π is the lowest common ancestor we are looking for (if so, we begin the second phase). We check a condition in Steps (5) and (6) that represents the "border" case in which we can readily find (τ,j,lcp): If $j = first$ and $L(\pi) = K_1$ (i.e, P is smaller than any of $\mathcal{K}$'s strings), then we return the leftmost leaf in the String B-tree as τ and safely set $lcp = \ell$ (by Theorem 3.8 and Fact 3.1). Otherwise, we decide according to the following possibilities: [3]

- Step (9): $j > first$. We infer that $L(\pi) <_L P \leq_L R(\pi)$ and conclude that π is the first ancestor of τ that we meet in our upward traversal (by Property 4.1). We then exit the **while** loop and begin the second phase (described below). At this point, we are sure that both Condition-A1 and Condition-A2 are satisfied.

- Step (10): $j = first$ and $L(\pi) \neq K_1$. Both Condition-A1$(P, parent(\pi), \ell)$ and $P \leq_L R(parent(\pi))$ are satisfied. Specifically, we verify the former by choosing $X = L(\pi)$ because $lcp(P, L(\pi)) = \ell$ (where $j = first$; see Fact 3.2) and $L(\pi)$ also belongs to $parent(\pi)$'s string set (by the String B-tree layout, see Figure 2). We verify the latter condition because $P \leq_L R(\pi) \leq_L R(parent(\pi))$ holds by the String B-tree layout. We repeat the **while** loop and move to π's parent by setting $\pi := parent(\pi)$, which exists because $L(\pi) \neq K_1$ and so $\pi \neq root$.

In the second phase (Steps (11) and (12)), we trace a downward path starting from node π down to leaf τ. Since Condition-A(P, π, ℓ) holds at the beginning of the second phase (by Step (9)), we can execute SB-Search-Down(P, π, ℓ) to traverse the String B-tree downwards by *starting from* π. We eventually retrieve triplet (τ, j, lcp) (by Lemma 4.3) and return it.

We now analyze the complexity of SB-Search-Up-Down. In the worst case, we execute both the upward and downward String B-tree traversals. During the upward traversal, we take one disk access to load π's page and no more than $\lceil \frac{\ell' - \ell}{B} \rceil + 1$ disk accesses to execute PT-Search(P,S_π,ℓ) by Theorem 3.8, where ℓ' is the new value assigned to ℓ (i.e., $\ell' = max\_lcp(P, \mathcal{S}_\pi) \geq \ell$). If we let ℓ_{up} be the last ℓ value in the upward traversal, the total cost of this traversal is a telescopic sum equal to $\frac{\ell_{up} - \ell}{B} + O(H)$, where $H = O(\log_B k)$ is the String B-tree height. During the downward traversal, we take $\frac{lcp - \ell_{up}}{B} + O(H)$ disk accesses

---

[3] Note that Steps (7) and (8) are not executed because $j < last$, due to our assumption that $P \leq_L R(\pi)$. Similarly, Steps (5) and (6) are not executed when we assume $L(\pi) <_L P$ because $j > first$.

(by Lemma 4.3) because Condition-A(P, π, ℓ_{up}) is satisfied and lcp is the final value of ℓ (i.e., the total number of P's characters examined). If we sum the upward and downward traversal costs, we obtain $\frac{lcp-\ell}{B} + O(\log_B k)$ worst-case disk accesses. $\square$

4.3 String Insertion

We discussed this operation for Problem 1 in Section 2.1 and we saw that its implementation is the most challenging task we have to face to solve Problem 2 (i.e., when $\mathcal{K}$ is obtained from $SUF(\Delta)$). We now treat this problem by describing how to update the String B-tree built on $SUF(\Delta)$ when adding a new string $Y[1,m]$ to Δ. The resulting String B-tree is obtained by inserting all of Y's suffixes into $\mathcal{K} = SUF(\Delta)$ in lexicographic order. In Section 2.2, we discussed the problems that arise when we insert Y's suffixes (i.e., the elimination of rescanning) and explained the algorithmic structure and logic behind our approach. We now go on to formalize these ideas and give a detailed description of the insertion procedure.

Without any loss in generality, we assume that $Y[m]$ is a distinct endmarker; therefore, no two suffixes in $SUF(\Delta \cup \{Y\})$ are each other's prefix. For a fixed i, let us number the strings in $SUF(\Delta) \cup \{Y[1,m], \ldots, Y[i,m]\}$ in lexicographic order and denote them by $SUF_i = \{S_1, S_2, \ldots, S_{N+i}\}$ (we let $SUF_0 = SUF(\Delta)$). We make sure that the String B-tree storing the string set SUF_i satisfies the following condition:

Condition-B(i):

(B1) Suffixes $Y[j,m]$ are stored in the String B-tree, for all $1 \leq j \leq i$, and $Y[i,m]$ shares its first $h_i = max\_lcp(Y[i,m], SUF_{i-1})$ characters with one of its adjacent strings.

(B2) All the *succ* pointers of SUF_i's strings are correctly set except for suffix $Y[i,m]$ whose *succ* pointer is still dangling. If $i = m$, then $succ(Y[i,m])$ is a self-loop pointer to its own leaf.

We let Condition-B(0) hold by convention. We insert Y's suffixes going from its longest to its shortest one by executing the pseudocode illustrated in Figure 8. For $i = 1, 2, \ldots, m$, we insert $Y[i,m]$ into the String B-tree storing SUF_{i-1} and satisfying Condition-B($i-1$) by means of Steps (1)–(5) in Figure 8. We obtain the String B-tree storing SUF_i and satisfying Condition-B(i). When $i = m$, we have the final String B-tree built on $SUF(\Delta \cup \{Y\})$.

In the rest of this section we show how to insert suffix $Y[i,m]$ and only discuss Steps (1)–(3), because Steps (4) and (5) are self-explanatory.

Step (1): We aim at computing triplet (π_i, j_i, h_i), where pair (π_i, j_i) identifies $Y[i,m]$'s position in SUF_{i-1} (i.e., π_i is the leaf in the String B-tree containing this position and j_i is $Y[i,m]$'s position in $\mathcal{S}_{\pi_i}$) and $h_i = max\_lcp(Y[i,m], SUF_{i-1})$ is the length of $Y[i,m]$'s longest prefix in common with any of SUF_{i-1}'s strings. We begin by identifying a crucial node π as follows:

- If $i = 1$, then π is the root of the String B-tree and we safely set $h_0 = 0$.

- If $i > 1$, we know by Condition-B1($i-1$) that a string $S_l \in SUF_{i-1}$ denotes suffix $Y[i-1, m]$ and its first h_{i-1} characters are shared with one of its two adjacent strings. We therefore examine either S_{l-1} (when $lcp(S_{l-1}, S_l) = h_{i-1}$) or S_{l+1} (when $lcp(S_l, S_{l+1}) = h_{i-1}$). Let us assume that we examine S_{l-1} by accessing its leaf in the String B-tree. We let π be the leaf pointed to by $succ(S_{l-1})$, correctly set by Condition-B2($i-1$) because S_{l-1} is not $Y[i-1, m]$.

At this point, we need the following observation about π's string set:

Lemma 4.8. *There is a string X in set $\mathcal{S}_\pi$ that shares at least its first $\max\{0, h_{i-1} - 1\}$ characters with $Y[i, m]$, i.e., $lcp(Y[i, m], X) \geq \max\{0, h_{i-1} - 1\}$.*

Proof: The lemma trivially holds when $0 \leq h_{i-1} \leq 1$ because every string X satisfies $lcp(Y[i, m], X) \geq 0 = \max\{0, h_{i-1} - 1\}$. We therefore assume that $h_{i-1} > 1$, so $\max\{0, h_{i-1} - 1\} = h_{i-1} - 1 \geq 1$. If we let X denote the string obtained by removing S_{l-1}'s first character (i.e., X is S_{l-1}'s second suffix), we can deduce that X must belong to $\mathcal{S}_\pi$ because $succ(S_{l-1}) = \pi$. Moreover, X and $Y[i, m]$ share their first $h_{i-1} - 1$ characters because the first h_{i-1} characters in S_{l-1} and $Y[i-1, m]$ are equal by the above hypothesis. We can therefore conclude that $lcp(Y[i, m], X) \geq \max\{0, h_{i-1} - 1\}$. □

Corollary 4.9. $h_i \geq \max\{0, h_{i-1} - 1\}$.

Proof: Since $h_i = max\_lcp(Y[i, m], SUF_{i-1})$ (by Condition-B1) and since there is a string $X \in \mathcal{S}_\pi \subseteq SUF_{i-1}$ that shares the first $lcp(Y[i, m], X) \geq \max\{0, h_{i-1} - 1\}$ characters with $Y[i, m]$ (by Lemma 4.8), we can conclude that $h_i \geq \max\{0, h_{i-1} - 1\}$. □

Lemma 4.8 and Corollary 4.9 suggest that we can start out from node π and only examine Y's characters in positions $i + \max\{0, h_{i-1} - 1\}, \ldots, i + h_i$ to perform searching. Namely, we execute SB-Search-Up-Down($Y[i, m]$, π, $\max\{0, h_{i-1} - 1\}$). We can prove:

Lemma 4.10. *If Condition-B($i-1$) holds, then we can find triplet (π_i, j_i, h_i) by using $\frac{h_i - \max\{0, h_{i-1} - 1\}}{B} + O(\log_B(N + m))$ worst-case disk accesses, where $1 \leq i \leq m$.*

Proof: For $i = 1$, Condition-A1($Y[1, m], root, 0$) is trivially satisfied for $P = Y[1, m]$, $\pi = root$ and $\max\{0, h_0 - 1\} = 0$. Consequently, we find triplet (π_1, j_1, h_1) by executing SB-Search-Up-Down($Y[1, m]$, $root$, 0) (by Lemma 4.7). It costs $\frac{h_1}{B} + O(\log_B N)$ disk accesses, as $(\pi_1, j_1, h_1) = (\tau, j, lcp)$, $\mathcal{K} = SUF_0$ and $k = N$ in Lemma 4.7.

For $i > 1$, we use Condition-B($i-1$). We know by Condition-B1($i-1$) that a string $S_l \in SUF_{i-1}$ denotes suffix $Y[i-1, m]$ and either $lcp(S_{l-1}, S_l) = h_{i-1}$ or $lcp(S_l, S_{l+1}) = h_{i-1}$. We can check if the former or the latter condition holds because we store $lcp(S_{l-1}, S_l)$ and $lcp(S_l, S_{l+1})$ in the leaves of the String B-tree (by Property 4.2). Condition-B2($i-1$) makes us sure that the $succ$ pointers for S_{l-1} and S_{l+1} are set and we can always reach node π by following one of them. We only need $O(1)$ disk accesses for this computation. After that, since we satisfy Condition-A1($Y[i, m], \pi, \max\{0, h_{i-1} - 1\}$) in node π (by Lemma 4.8), we can execute SB-Search-Up-Down($Y[i, m]$, π, $\max\{0, h_{i-1} - 1\}$) in order to obtain the triplet (π_i, j_i, h_i) (by Lemma 4.7). We know that $h_i \geq \max\{0, h_{i-1} - 1\}$ by Corollary 4.9 and so it costs $\frac{h_i - \max\{0, h_{i-1} - 1\}}{B} + O(\log_B(N + m))$ disk accesses (by letting $(\pi_i, j_i, h_i) = (\tau, j, lcp)$, $\mathcal{K} = SUF_{i-1}$ and $k \leq N + m$ in Lemma 4.7). □

Step (2): We take triplet (π_i, j_i, h_i) and insert suffix $Y[i,m]$ into $\mathcal{S}_{\pi_i}$ before its j_i-th string by means of the PT-Insert procedure. We prove:

Lemma 4.11. *$Y[i,m]$'s insertion in π_i's string set requires one disk access.*

Proof: Let S_r be the string in SUF_{i-1} that corresponds to the (j_i)-th string in $\mathcal{S}_{\pi_i}$. Since $Y[i,m]$ has to be inserted to S_r's left, we execute PT-Insert($Y[i,m], PT_{\pi_i}, S_r, lcp, Y[i+lcp], c$), where the parameters $lcp = lcp(Y[i,m], S_r)$ and $c = S_r[lcp+1]$ are determined as follows: If $r = 1$, then $lcp = h_i$ by Fact 3.2. If $r > 1$, then we know $lcp(S_{r-1}, S_r)$ and their mismatching characters, say c_{r-1} and c_r (by Property 4.2). If $lcp(S_{r-1}, S_r) \geq h_i$, then we are sure that $lcp = h_i$ by Fact 3.2, because r is $Y[i,m]$'s position in SUF_{i-1}. If $lcp(S_{r-1}, S_r) < h_i$, we know that either $lcp = h_i$ (if $c_r = Y[i + lcp(S_{r-1}, S_r)]$) or $lcp = lcp(S_{r-1}, S_r)$ (if $c_r \neq Y[i + lcp(S_{r-1}, S_r)]$). We then access S_r to set $c = S_r[lcp+1]$. $\square$

We can prove another version of Lemma 4.11 in which no disk accesses are needed to insert $Y[i,m]$. However, this would involve a more complex case analysis without improving its overall complexity.

Step (3): If a split occurs after $Y[i,m]$'s insertion (i.e., $|\mathcal{S}_{\pi_i}| > 2b$), we rebalance the String B-tree and possibly redirect some *succ* and *parent* pointers. This rebalancing operation, called SB-Split(π_i), cannot be handled in a straightforward way and so we postpone a detailed discussion of it to Section 4.5. We are now able to state our main result on Y's insertion:

Lemma 4.12. *We can insert a new string $Y[1,m]$ into the String B-tree (i.e., all of Y's suffixes into $SUF(\Delta)$) with $O(m \log_B(N+m))$ disk accesses plus m calls to* SB-Split *in the worst case.*

Proof: We refer to the pseudocode shown in Figure 8. Its correctness follows by induction on Condition-B(i), for $i = 1, 2, \ldots, m$. We assume that Condition-B($i-1$) holds (this is true by convention for $i = 1$). We find triplet (π_i, j_i, h_i) in Step (1) (by Lemma 4.10) and know that $Y[i,m]$ shares its first h_i characters with one of its two adjacent strings (by Fact 3.2). We then insert $Y[i,m]$ into π_i with one disk access in Step (2) (by Lemma 4.11) and maybe handle a split by means of an SB-Split call in Step (3). Consequently, we satisfy Condition-B1(i). In Steps (4) and (5), we set $succ(Y[i-1,m])$ and let $succ(Y[i,m])$ be the only dangling pointer (unless $i = m$) in order to satisfy Condition-B2(i). We conclude that Condition-B(i) holds after Steps (1)–(5). At the end of the insertion process, the validity of Condition-B(m) implies that we have correctly built the String B-tree on $SUF(\Delta \cup \{Y\})$.

As far as its complexity is concerned, Step (1) requires $d_i = \frac{h_i - \max\{0, h_{i-1}-1\}}{B} + O(\log_B(N+m))$ disk accesses by Lemma 4.10. Step (2) takes one disk accesses by Lemma 4.11. Step (3) makes one call to SB-Split. Steps (4) and (5) do not take any disk accesses if we leave $Y[i-1,m]$'s page in main memory for another iteration. The total cost is therefore $O(\sum_{i=1}^{m} d_i) = O(\frac{h_m - h_0}{B} + m \log_B(N+m))$, plus m calls to SB-Split. As previously stated, $h_0 = 0$ and $h_m \leq m$, and so the whole insertion process takes a total of $O(m \log_B(N+m))$ disk accesses, plus m calls to SB-Split in the worst case. $\square$

4.4 String Deletion

We can update the String B-tree by deleting a string $Y[1,m]$ from Δ. We delete all of Y's suffixes from $\mathcal{K} = SUF(\Delta)$ going from the longest to the shortest one. We first locate leaf π_1, which contains suffix $Y[1,m]$, by means of SB-Search-Up-Down($Y[1,m], root, 0$), using $O(\frac{m}{B} + \log_B(N+m))$ disk accesses (by Lemma 4.10). For $i > 1$, we locate leaf π_i, which contains suffix $Y[i,m]$, by simply following the $succ(Y[i-1,m])$ pointer. This takes us no more than $O(1)$ disk accesses.

When we load π_i's page, we delete $Y[i,m]$ from its string set $\mathcal{S}_{\pi_i}$ by executing PT-Delete($Y[i,m], PT_{\pi_i}$). If a merge occurs after $Y[i,m]$'s deletion (i.e., $|\mathcal{S}_{\pi_i}| < b$), then we rebalance the String B-tree by means of SB-Merge(π_i). We postpone our discussion of SB-Merge to Section 4.5. It is worth noting that no *succ* pointers are dangling after a deletion. We obtain:

Lemma 4.13. *We can delete a string $Y[1,m]$ from the String B-tree (i.e., all of Y's suffixes from $SUF(\Delta)$) in $O(m + \log_B(N+m))$ disk accesses plus m calls to* SB-Merge *in the worst case.*

4.5 Handling SB-Split and SB-Merge operations

We first describe a solution for SB-Split and SB-Merge that takes $O(B\log_B(N+m))$ worst-case disk accesses per operation by a straightforward pointer-handling approach. We then improve this solution by means of an accounting method that takes $O(\log_B(N+m))$ *amortized* disk accesses per operation. Finally, we show how to obtain $O(\log_B(N+m))$ *worst-case* disk accesses per operation by means of a clustering technique. We introduce the first two methods to explain the third one (based upon clustering) better. We say that a node π is *full* after an insertion if its string set size $|\mathcal{S}_\pi|$ becomes larger than $2b$; a node π is *half-full* after a deletion if $|\mathcal{S}_\pi|$ becomes smaller than b.

First method: Pointer handling. Let us first examine SB-Split(π), where π is a full leaf. We split set $\mathcal{S}_\pi$ by the PT-Split operation applied to PT_π in order to produce two smaller Patricia tries, say PT_1 and PT_2 (where PT_2 stores b strings). Patricia trie PT_1 takes the place of PT_π inside π's page, while PT_2 is put into a new leaf σ's disk page. We let σ be π's right sibling and update the node information as follows: we properly set π's and σ's pointers (i.e., *prev*, *next* and *parent*) and determine the longest common prefix length of strings $R(\pi)$ and $L(\sigma)$ and their mismatching characters, by Property 4.2.

We update the b pointers *succ* leading to the strings moved from π to σ (they must now point to σ instead of π) and use the inverse $succ^{-1}$ pointers to determine their locations. Moreover, we maintain the String B-tree structure by inserting strings $R(\pi)$ and $L(\sigma)$ into set $\mathcal{S}_{parent(\pi)}$. This insertion may cause $parent(\pi)$ to split; if so, $\frac{b}{2}$ *parent* pointers in its children have to change and point to $parent(\pi)$'s new sibling. When an ancestor becomes full, it makes us insert two strings in its parent and so the splitting process can continue and involve many of π's ancestors until either a non-full ancestor is encountered or a new root is created. In the latter case the height of the String B-tree is increased by one.

The SB-Merge(π) operation (in which π is a half-full leaf) affects one of π's adjacent siblings, say σ. Without any loss in generality, we assume that σ is on π's right. We move σ's two leftmost strings into π, so that π is no longer half-full. However, if σ also becomes half-full, we merge π and σ together by executing a PT-Concatenate operation on their corresponding Patricia tries PT_π and PT_σ (see Section 3.2; the other input parameters lcp, c and c' can be obtained by Property 4.2). Analogously to the SB-Split case, we redirect no more than b pointers *succ* from σ to π and deallocate σ's disk page. PT-Concatenate can also cause both the merging of π's and σ's parents because we delete $R(\pi)$ and $L(\sigma)$ from them, respectively, and the updating of no more than $\frac{b}{2}$ *parent* pointers in $parent(\sigma)$'s children. The merging process can continue and involve many of π's ancestors until either a non-half-full ancestor is encountered or the root is removed. In the latter case, the height of the String B-tree is decreased by one.

It follows that we need $O(1)$ disk accesses to handle each node except for the updating of its incoming *succ* and *parent* pointers, which do not exceed b in number and can be stored in different disk pages. Therefore, their updating takes a total of $O(bH) = O(B \log_B(N + m))$ worst-case disk accesses, while the other operations involved only require $O(H)$ disk accesses. This analysis makes it clear that updating *succ* and *parent* pointers is our major obstacle in achieving an efficient String B-tree update. We focus on this part of the updating process in the rest of this section.

Second method: Amortized accounting. We use the accounting method [44] for our amortized analysis and show how to achieve the $O(\log_B(N + m))$ amortized bound per operation. [4] Without any loss in generality, we assume that we have to redirect exactly b pointers (*succ* or *parent*) for every node splitting or merging and we deal with pairs of strings in every insertion or deletion operation (the latter assumption is motivated by the String B-tree layout, see Figure 2). At the beginning, we let each newly-created node contain $\frac{3b}{2}$ strings and we assign an account of zero credits to it. We choose a partitioning of $SUF(\Delta)$'s strings among the nodes of the String B-tree such that they contain $\frac{3b}{2}$ strings each at first. If the root or the rightmost node in a level contains fewer strings, we add some dummy strings. When a node becomes either half-full or full, we show that it has accumulated a sufficiently large number of credits to pay for the $\Theta(b)$ disk accesses needed for the updating of its *succ* or *parent* pointers. We say that a node is *affected* by a split (resp., merge) operation, if it is the updated leaf or one of its children is split (resp., merged).

Let us examine SB-Split and a node π affected by the corresponding splitting process. Its string set size $|\mathcal{S}_\pi|$ is increased by two because we treat pairs of strings. If π is full (i.e., $|\mathcal{S}_\pi| = 2b+2$), then we take one of π's adjacent siblings, say σ, and we assume without any loss in generality that σ is on π's right. We move π's two rightmost strings to σ. If σ also becomes full, we create a new node τ and distribute the $4b+2$ strings in $\mathcal{S}_\pi \cup \mathcal{S}_\sigma$ as follows: π contains the first $\frac{3b}{2}$ strings, τ gets the next $b+2$ strings and σ contains the remaining $\frac{3b}{2}$ strings. Finally, we redirect the $b+2$ *succ* pointers which previously led to nodes π and σ, in order to point to new node τ.

Let us now examine SB-Merge and a node π affected by the corresponding merging process. Its string set size $|\mathcal{S}_\pi|$ is decreased by two because we treat pairs of strings. If π is

---

[4] A similar approach was presented in [32]. We treat the problem in detail here in order to make the subsequent discussion of the worst-case solution clearer.

half-full (i.e., $|\mathcal{S}_\pi| = b-2$), we take one of π's adjacent siblings, say σ, and assume without any loss in generality that σ is on π's right. We move σ's two leftmost strings to π. If σ also becomes half-full, we put all of σ's strings into π and thus form a set of $2b-2$ strings and deallocate σ's disk page. Finally, we redirect b pointers from σ to π.

We can charge the cost for the redirection of the *succ* and *parent* pointers to the previous update operations to achieve the following amortized bound:

Lemma 4.14. *A given string $Y[1,m]$ can be inserted into Δ or deleted from it with $O(m \log_B(N+m))$ amortized disk accesses.*

Proof: Since we use Lemmas 4.12 and 4.13, we do not need any more than $O(m \log_B(N+m))$ disk accesses, plus m calls to either SB-Split or SB-Merge. We show that we have enough credits to pay for redirecting the *succ* and *parent* pointers in each affected node. We maintain the invariant that, if s is a node's string set size, *its account balance must have at least $BL(s) = 3\left|s - \frac{3b}{2}\right|$ credits.* Consequently, we can use $BL(b) = BL(2b) = \frac{3b}{2}$ credits for updating the pointers. We now show how to manage these accounts and maintain the invariant on BL.

We assign $6H$ credits to each SB-Split or SB-Merge call, where $H = O(\log_B(N+m))$ is the current height of the String B-tree. Each affected node (there are no more than H of them) increases or decreases its string set size by two and so we always assign 6 credits to its balance in order to maintain the invariant on BL. We employ the accumulated credits as follows (we assume that b and $\frac{3b}{2}$ are even integers and $b \geq 4$, because each node has at least two children):

Let us consider the splitting of a node π. If π is full, we move two strings and 6 credits to σ's account after deleting them from π's account, which now has at least $BL(2b+2) - 6 = BL(2b)$ credits. If σ also becomes full, then it has at least $BL(2b) + 6 = BL(2b+2)$ credits. We handle π's and σ's splitting by using the credits in their accounts and these, in turn, satisfy the invariant on BL. We have at least $BL(2b) + BL(2b+2) \geq 3b$ credits for distributing strings in π, τ and σ: we give $\frac{3b}{2} - 6$ credits to τ's balance (because $|\mathcal{S}_\tau| = b+2$ and $BL(b+2) = \frac{3b}{2} - 6$) and zero credits to π's and σ's balance (because $|\mathcal{S}_\pi| = |\mathcal{S}_\sigma| = \frac{3b}{2}$ and $BL(\frac{3b}{2}) = 0$). We spend the remaining $\frac{3b}{2} + 6$ credits for updating the $b+2$ pointers redirected to τ (because $\frac{3b}{2} + 6 > b+2$ for $b \geq 4$).

Let us consider the merging of a node π. If π is half-full, we move two strings to π and safely add 6 credits to σ's account after deleting them from π's account. This is sure to maintain the invariant on BL for σ's account. In this way, π's account has at least $BL(b-2) - 6 = BL(b)$ credits on it. If σ also becomes half-full, then it has at least $BL(b-2)$ credits because of the invariant on σ's account. We therefore have at least $BL(b) + BL(b-2) \geq 3b$ credits for concatenating strings in π and σ: we leave $\frac{3b}{2} - 6$ credits in π's new balance (so that $BL(2b-2) = \frac{3b}{2} - 6$) and we spend the remaining credits (at least $\frac{3b}{2}$) to update the b pointers redirected to π.

In conclusion, we always have enough credits for updating the *succ* and *parent* pointers and each operation takes $O(H)$ amortized disk accesses. $\square$

Third method: Worst-case clustering. We move some strings in pairs between two adjacent sibling nodes in a *lazy fashion* to obtain our worst-case bounds. The main idea underlying this approach is that SB-Split and SB-Merge cannot be executed on the same node too frequently. We can distribute their cost incrementally over the other operations that involve the node between any two consecutive SB-Split or SB-Merge operations.

In the previously-described amortized method, when we split a node π, we take one of its adjacent siblings, say σ, and create a node τ between π and σ. We then distribute $\mathcal{S}_\pi \cup \mathcal{S}_\sigma$'s strings among nodes π, τ and σ by using the credits accumulated (see Lemma 4.14's proof). At this point, instead, we delay the distribution process on the subsequent update operations by keeping a pointer from nodes π and σ to node τ and by marking the $b+2$ strings to be moved to τ. The three nodes form a cluster, called *split-cluster*. We move *four* marked strings and their corresponding *succ* or *parent* pointers from π and σ to τ every time we access the split-cluster (i.e., one of its three nodes) to perform some subsequent update operations (insertions or deletions).

When merging a node π with one of its adjacent siblings, say σ, we set a pointer to link them and thus form a *merge-cluster*. We mark the b strings to be moved from σ to π. We move *four* marked strings and their corresponding *succ* or *parent* pointers every time we access the merge-cluster (i.e., one of its two nodes) to perform some subsequent update operations (insertions or deletions).

We also introduce the notion of *singleton clusters*, which are the nodes not involved in split or merge operations. We follow the rule that after moving the last marked string of a split- or merge-cluster, we transform it into three or less singleton clusters with $O(1)$ disk accesses. It is worth noting that our strategy does not affect our search, insert and delete algorithms because we can ignore the underlying clustering in the whole String B-tree structure (except when handling SB-Split and SB-Merge as discussed below).

We now deal with the problem of managing half-full and full nodes in terms of half-full and full clusters. We have to fix the total number of strings (both marked and unmarked) that can be stored in a cluster. A singleton cluster can contain from b to $2b$ strings ($\frac{3b}{2}$ at the beginning). A split-cluster can store from $3b$ to $6b$ strings ($4b+2$ at the beginning). A merge-cluster can have from b to $4b$ strings ($2b-2$ at the beginning). We say that a cluster is *inconsistent* if its number of strings is either below the minimum or above the maximum allowed (depending on the type of cluster). We prove the fundamental property that we *only use* singleton clusters when forming non-singleton clusters. In other words, all the marked strings (and their *succ* and *parent* pointers) in a non-singleton cluster have been moved to form three or less singleton clusters before any other clustering involving them can occur. We can now prove the following result:

Lemma 4.15. *If a cluster is inconsistent, it is a singleton cluster.*

Proof: As stated above, after moving the last marked string in a cluster, we transform the cluster into three or less singleton clusters. By contradiction, let us now assume that a split-cluster is inconsistent. At the beginning, it contains $4b+2$ strings ($b+2$ of them are marked) so the cluster is accessed at least $\frac{b}{2}$ times before becoming inconsistent because we only treat strings in pairs. Since at least $4\frac{b}{2} \geq b+2$ marked strings and pointers are moved, we can conclude that all marked strings are moved and the cluster is decomposed into three or less singleton clusters before becoming inconsistent again. An analogous argument holds when we assume that a merge-cluster is inconsistent. At the beginning, it contains $2b-2$ strings (b of them are marked) and so it is accessed at least $\frac{b-2}{2}$ times before becoming inconsistent. Since at least $4\frac{b-2}{2} \geq b$ marked strings ($b \geq 4$) are moved, we can conclude that all the marked strings are moved and the cluster is decomposed into three or less singleton clusters before becoming inconsistent again. Consequently, only singleton clusters can become inconsistent. $\square$

Theorem 4.16. *A given string $Y[1,m]$ can be inserted into Δ or deleted from it with $O(m \log_B(N+m))$ worst-case disk accesses.*

Proof: An SB-Split affects a leaf-to-root path Π of H nodes and only allows the insertion of two or less strings into each node in Π in the worst case, where H is the current height of the String B-tree. We therefore access H clusters in the worst case and maybe move four marked strings and pointers in some of them. Let us now examine a cluster containing a node π in path Π and let $\mathcal{C}_\pi$ be its cluster. Two cases occur:

(a) $\mathcal{C}_\pi$ is a singleton cluster. If $\mathcal{C}_\pi$ is not inconsistent, we have enough room for the new strings. Otherwise (i.e., $\mathcal{C}_\pi$ is inconsistent), we have to move two strings from π to one of its two adjacent siblings, say σ (let $\mathcal{C}_\sigma$ be its cluster). If $\mathcal{C}_\sigma$ also becomes inconsistent, it is a singleton cluster (Lemma 4.15) and so we create a new split-cluster made up of π and σ. If $\mathcal{C}_\sigma$ does not become inconsistent, it has enough room for π's moved strings. If $\mathcal{C}_\sigma$ is also a non-singleton cluster, we move four marked strings internally in it, and create three or less singleton clusters from it, if it does not contain any marked strings. This computation takes $O(1)$ disk accesses.

(b) $\mathcal{C}_\pi$ is a non-singleton cluster. We insert the new strings in $\mathcal{C}_\pi$ because it cannot be inconsistent (by Lemma 4.15) and we move four marked strings internally in $\mathcal{C}_\pi$. After that, if $\mathcal{C}_\pi$ does not contain any marked strings, we create three or less singleton clusters. This computation takes $O(1)$ disk accesses, too.

We conclude that we need a total of $O(H)$ worst-case disk accesses to handle an SB-Split. As far as SB-Merge is concerned, we can perform an analogous analysis to show that we spend $O(H)$ worst-case disk accesses in this case, too. In brief, updating the String B-tree under the insertion or deletion of a string $Y[1,m]$ requires $O(m)$ SB-Split and SB-Merge operations, and each operation makes $O(H) = O(\log_B(N+m))$ disk accesses in the worst case. Consequently, the bound we claim follows from Lemma 4.12 and 4.13. $\square$

Remark 4.17. The substring searching and updating described in Problem 2 can be solved within the bounds claimed in Theorem 2.2. The relative search bounds are proved in Theorem 4.4, while the update bounds are proved in Theorem 4.16.

5 Previous Work

Several elegant and well-known data structures can be used for solving Problems 1 and 2 mentioned in the introduction. Some of them have good average-case behavior and are good tools in some practical cases. However, they do not support good *worst-case* searching and updating operations. Their inefficiency is mainly due to the methods they use for packing a lot of data into the disk pages in order to avoid that many pages are almost empty after a few updates. This can make their worst-case performance seriously degenerate in external memory.

We go on to survey the most popular tools used for manipulating external-memory text strings and we point out their theoretical limitations when applied to our two problems. These tools can be grouped into two main classes: One kind is explicitly designed to work in external memory and contains inverted files [39], B-trees [9] and their variants, known as prefix B-trees [10, 15]. The other kind is useful for indexing a text in main memory with the aim of performing string matching. It contains compacted tries [22, 36], suffix trees [3, 25, 34, 48] and suffix arrays [22, 33]. They can easily be adapted for use in external memory but at the price of worsening their good performance in main memory.

- **Inverted files** are an important indexing technique for secondary key retrieval [24, 29, 39], in which the roles of records and attributes are reversed. This means that we list the records having a given attribute instead of listing the attributes of a given record. Inverted file's components are called *inverted lists* and occupy very little space (sublinear in many practical cases). We can use inverted files for solving Problems 1 and 2 by interpreting the records as arbitrarily-long texts and the attributes as text substrings (e.g., words, q-grams, etc.). Unfortunately, it is rather difficult to obtain the attributes when treating unformatted texts (e.g., DNA sequences) and to allow for arbitrary substring searches without introducing a lot of duplicate information and significant space overhead. With regard to Problem 2, it turns out that inverted files support very poor queries and updates because they take unnecessary disk accesses in the worst case.

- **Prefix B-trees** are B-tree variations whose leaves contain all the keys and whose internal nodes contain copies of some keys for routing the B-tree traversal. Since the keys are arbitrarily-long strings, we cannot always stuff a group of them into a single prefix B-tree node, which is stored in one disk page of bounded capacity B, because a string can be possibly longer than B. This problem can be overcome by representing the key strings by their logical pointers and employing the so-called *separators* to implement the routing keys in the internal B-tree nodes [10]. Specifically, the separator of keys 'computer' and 'machine' can either be one of them or any short string between them in lexicographic order (such as 'f' or 'do'). It goes without saying that the shortest separators [15] are chosen to save as much space as possible. Two popular, empirical strategies have been devised to keep separators short after a few updates. The first one [10] uses the shortest unique prefix of a key as its separator but it can fail because separator's length can be proportional to key's length and therefore it introduces a lot of duplicate information. This often happens in practice because the keys with a common prefix are adjacent to each other in lexicographic order. The second strategy uses a compression scheme to store the keys in the internal nodes, as in the Unix prefix B-trees [47]. That is, if a key begins with the same n characters as its immediate predecessor, the key is stored with its first n characters replaced by integer n. This approach saves space but it does not prevent a key from having a lot of characters in the rest of its positions $n+1, n+2, \ldots$. In brief, the worst-case performance of prefix B-trees is very good only for *bounded-length keys* (i.e., no more than 255 characters long [47]) because they can exploit B-tree power to solve Problem 1: searching takes $O(\frac{occ}{B} + \log_B k)$ disk accesses and updating takes $O(\log_B k)$ disk accesses. However, this performance becomes poor in the worst case when treating unbounded-length keys, as it occurs in Problem 2.

- **Suffix arrays** and PAT-arrays [22] allow for fast searches whose cost does not depend on the alphabet's size [33]. A suffix array essentially stores all the text suffixes in lexicographic order by means of their logical pointers. Thanks to their simplicity, these data structures can be adapted for use in external memory by partitioning them into contiguous disk pages. There is no limit on key length and searching takes $O(\frac{p}{B}\log_2 N + \frac{occ}{B})$ disk accesses [22]. From a practical point of view, suffix arrays are the most space-efficient indexing data structures available because only a pointer per suffix is stored. Nonetheless, suffix arrays cannot be modified any more than inverted files can be and so the contiguous space needed for storing them can become too constraining when the text strings get longer. A dynamic version for main memory of suffix arrays has been recently proposed in [17]. It can be extended to work in external memory at the price of losing its space optimality (i.e., occupying $O(\frac{N\log_2 N}{B})$ disk pages) and achieving a worse searching bound.

- **Suffix trees** and compacted tries in general are elegant, powerful data structures widely employed in string matching problems [6]. The suffix tree is a compacted trie built on all of the text suffixes: Each arc is labeled by a text substring, where triple (X, i, j) is used to denote a substring $X[i, j]$, and the sibling arcs are ordered according to their first characters, which are distinct. There are no nodes having only one child except the root and each node has associated the string obtained by concatenating the labels found along the downward path from the root to the node. By appending an end-marker to the text, the leaves have a one-to-one correspondence to the text suffixes so each leaf stores a distinct suffix. Suffix trees are also augmented by means of some special node-to-node pointers, called *suffix links* [34], which turn out to be crucial for the efficiency of the searching and updating operations. The suffix link from a node storing a nonempty string, say aY for a character a, leads to the node storing Y and this node always exists. There can be $\Theta(|\Sigma|)$ suffix links leading to a node, where Σ denotes the alphabet, because we can have one suffix link for each possible character $a \in \Sigma$. Suffix trees require linear space and are sometimes called *generalized* suffix trees when treating a string set Δ [3, 25]. Searching for a pattern $P[1, p]$ in Δ's strings requires $O(p\log|\Sigma| + occ)$ time, where occ is the number of pattern occurrences. Inserting a string $X[1, m]$ into Δ or deleting from it takes $O(m\log|\Sigma|)$ time.

Since suffix trees are powerful data structures, it would seem appropriate to use them in external memory. To our surprise, however, they lose their good searching and updating *worst-case* performance when used for indexing large text collections that do not fit into main memory. This is due to the following reasons:

a. Suffix trees have an *unbalanced topology* that is text-dependent because their internal nodes are in correspondence to some repeated substrings. Consequently, these trees inevitably inherit the drawbacks pointed out in scientific literature with regard to paging unbalanced trees in external memory. There are some good average-case solutions to this problem that group $\Theta(B)$ nodes per page under node insertions only [29, Sect.6.2.4] (deletions make the analysis extremely difficult [41]), but they cannot avoid storing a downward path of k nodes in $\Omega(k)$ *distinct* pages in the worst case.

b. Since the outdegree of a node can be $\Theta(|\Sigma|)$, its pointers to children might not fit into

$O(1)$ disk pages so they would have to be stored in a separate B-tree. This causes an $O(\log_B |\Sigma|)$ disk access overhead for each branch out of a node.

c. Branching from a node to one of its children requires further disk accesses in order to retrieve the disk pages containing the substring that labels the traversed arc because labels are pointers in order to occupy constant space.

d. Updating suffix trees under string insertions or deletions [3, 25] requires the insertion or deletion of some nodes in their unbalanced structure. This operation inevitably relies on merging and splitting disk pages in order to occupy $\Theta(\frac{N}{B})$ of them. This approach is very expensive: splitting or merging a disk page can take $O(B|\Sigma|)$ disk accesses because $\Theta(B)$ nodes can move from one page to another. The $\Theta(|\Sigma|)$ suffix links leading to each node moved must be redirected and they can be contained in different pages.

We can conclude that adapting suffix trees to solve Problems 1 and 2 is not efficient in the worst case. Searching for a pattern of length p takes $O(p \log_B |\Sigma| + occ)$ worst-case disk accesses in both problems according to Points a–c. Inserting or deleting an m-length string takes $O(m \log_B |\Sigma| + B|\Sigma|)$ disk accesses in Problem 1 because there can be $O(1)$ page splits or merges as described in Point d; and $O(mB|\Sigma|)$ disk accesses in Problem 2 because there can be $\Theta(m)$ page splits or merges.

From an average-case-analysis point of view, compact trie's performance in external memory is heuristic and usually confirmed by experimentation [5, 22, 40]. Recently, Clark and Munro [13] have obtained an efficient implementation of suffix trees in external memory by compactly representing them via Patricia tries. This data structure allows to solve Problem 2 with $O(\frac{h}{\sqrt{p}} + \log_p N)$ disk accesses for Substring Search(P), where $h \leq N$ is Patricia trie's height. Inserting or deleting a string in Δ costs at least as searching *for all* of its suffixes individually. The solution is practically attractive but does not guarantee provably good performance in the worst case.

6 Some Applications

6.1 P-strings and Software Duplication

The *parameterized pattern matching* problem was introduced in [7] with the aim of finding the program fragments in a software system that are identical except for their systematic change of parameters. The program fragments are in the form of token sequences produced by a lexical analyzer and encoded by some parameterized strings, called *p-strings*. From a formal point of view, p-strings are sequences of characters taken from two disjoint ordered alphabets , and Π, where , contains the fixed symbols (i.e., the fixed tokens) and Π contains the parameter symbols (i.e., identifiers and constants). A *p-match* of two p-strings occurs when one p-string can be transformed into the other by one-to-one parameter renaming. For example, let us take , $= \{a, b\}$ and $\Pi = \{x, y, z\}$. There is a p-match of p-strings $axxbyxa$ and $ayybzya$ by simultaneously replacing x with y and y with z. Given two p-strings X

and Y, there is a *p-occurrence* of X in Y, if there is a p-match of X and $Y[i, i + |X| - 1]$ for an integer i (e.g., there is a p-occurrence of $zbxz$ in $axxbyxa$ for $i = 3$).

A suffix tree generalization to p-strings, called *p-suffix tree*, was introduced in [7] and subsequently improved in [30], to perform online pattern matching on p-strings efficiently. Some other algorithms that were designed for this new paradigm also had to deal properly with the dynamic nature of parameter renaming in p-strings [4, 8, 26]. The main problem in designing efficient p-string algorithms is concerned with Properties (1) and (2), which hold for any ordinary two strings S and T, while Property (2) does not hold for p-strings [7]:

(1) Common Prefix Property: If $aS = bT$, then $S = T$.

(2) Distinct Right Context Property: If $aS = bT$ and $aSc \neq bTd$, then $Sc \neq Td$.

Since Property (2) is used for defining suffix links of suffix tree nodes, it creates some problems for p-suffix tree construction. Since String B-trees only need Property (1) in Lemma 4.8, they work for p-strings after undergoing some slight changes. We let Σ denote alphabet , $\cup\, \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers disjoint from , . We define an operation $\mathsf{prev}(X)$ that transforms a p-string X into a string in Σ^* according to [7]: A constant symbol in , is mapped into itself. A parameter occurrence in Π is mapped into 0 if it is the leftmost one. Otherwise, the parameter occurrence is mapped into the integer that denotes the distance from its previous occurrence's position. For example, $\mathsf{prev}(axxbyxa) = \mathsf{prev}(ayybzya) = a01b03a$. Any two p-strings X, Y have a p-match if $\mathsf{prev}(X) = \mathsf{prev}(Y)$. As stated in [7], given $\mathsf{prev}(Y)$ and one of Y's suffixes, say $Y[j, m]$, an arbitrary character in $\mathsf{prev}(Y[j, m])$ can be computed by a constant number of arithmetic operations.

Let us now examine the natural extension of Problems 1 and 2 to p-strings. We let Problem 2 stand for both and show in Theorem 6.1 below that String B-trees can be extended to solve the problem without any loss in efficiency. Set Δ is made up of some p-strings whose total length is N; set $SUF(\Delta)$ is made up of the *strings* obtained by applying prev to the suffixes of Δ's p-strings, i.e., $SUF(\Delta) = \{\mathsf{prev}(\delta[i, |\delta|]) : 1 \leq i \leq |\delta| \text{ and } \delta \in \Delta\}$. It is worth noting that these suffixes are sorted according to a new order $\leq_L^P$, where $X \leq_L^P Y$ if and only if $\mathsf{prev}(X) \leq_L \mathsf{prev}(Y)$. We now examine the String B-tree built on $SUF(\Delta)$.

In searching for p-strings, we have to transform the pattern p-string into a pattern string in Σ^* by means of prev. We then search for the pattern string in the String B-tree by using the string procedure in Section 4.1. Inserting a p-string $Y[1, m]$ in Δ consists of inserting suffix $\mathsf{prev}(Y[i, m])$ into the current String B-tree, for $i = 1, 2, \ldots, m$. Our considerations in Section 4.3 extend to this case because we only use Property (1). For example, we let $\mathsf{prev}(S_{l-1})$ be a string of $SUF(\Delta)$ that shares its first h_{i-1} characters with $\mathsf{prev}(Y[i-1, m])$, as required by Condition-B$(i - 1)$. We identify node π by means of $succ(S_{l-1})$, which is also well-defined for p-strings due to Lemma 4.8 and Property (1). Finally, we execute $\mathsf{SB\text{-}Search\text{-}Up\text{-}Down}(\mathsf{prev}(Y[i-1, m]),\ \pi,\ \max\{0, h_{i-1} - 1\})$ and continue as in Section 4.3. A p-string deletion can be performed by the procedure described in Section 4.4. We can now state the following result:

Theorem 6.1. *Let Δ be a set of p-strings whose total length is N. The String B-tree built on Δ occupies $\Theta(N/B)$ disk pages. Searching for all the pocc occurrences of a p-string $P[1, p]$ in Δ's p-strings takes $O(\frac{p+pocc}{B} + \log_B N)$ worst-case disk accesses. Inserting a p-string $Y[1, m]$ into set Δ or deleting it takes $O(m \log_B(N + m))$ worst-case disk accesses.*

We can use String B-trees on p-strings in main memory by letting $B = O(1)$. When the alphabet is large (i.e., either $|,| = O(N)$ or $|\Pi| = O(N)$), we achieve an alphabet-independent search that requires $O(p+\log N+pocc)$ time and improves the $O(p \log N+pocc)$ searching bound in [7, 30]. With a large alphabet, String B-tree construction requires the same $O(N \log N)$ time complexity as the p-suffix tree's. For a constant size alphabet, the bounds in [7, 30] are better than ours.

6.2 Database Indexing

We can maintain several indices on the same database *without* copying the (multiple) key strings in the indices but we use our solution to Problem 1. This is important in *compound attribute* organizations [29, Sect 6.5] to maintain the lexicographic order of the records' combined attributes without having to make any copies. As a result, String B-trees turn out to be a powerful tool for indexing databases.

Let us consider a database (*not* necessarily a *text* database) with variable-length records $D = \{R_1, R_2, \ldots, R_k\}$ and an alphabet Σ (e.g., Σ is made up of the ASCII characters). We introduce an *indexing function* $f : D \rightarrow \Sigma^*$ that transforms a record R_i into a string $K_i = f(R_i)$, such that $R_i \leq R_j$ if and only if $K_i \leq_L K_j$, where $1 \leq i, j \leq k$ and $\leq_L$ is the lexicographic order. For example, when R_i is an employee's record, K_i is the birthday in the string format 'YYYYMMDD', where 'YYYY' is the year, 'MM' is the month and 'DD' is the day, or K_i is the string concatenation of some fields in R_i, such as employee's name, office, phone number and so on. Since f maps the records into some strings, we allow f to be powerful enough to handle any kind of *string manipulations* on the original records' fields (e.g., we take some substrings, concatenate them, reverse them, etc.). That is, $K_i = f(R_i)$ is a "virtual string" because it does not necessarily appear in R_i. In this case, the logical pointer for K_i leads to R_i, which we have to apply f to.

We use Problem 1 on string set $\mathcal{K} = \{K_1, \ldots, K_k\}$ and provide an index that *only* requires $O(k)$ space whatever the total string length is (Theorem 2.1). We recompute $f(R_i)$ every time we need to (compare) access a string K_i. Even though $f(R_i)$'s computation might be expensive in some cases, we load and compare *only one string per level of the String B-tree* because of the Patricia trie layout in the nodes of the String B-tree, as shown in Section 3.1. Consequently, our approach actually requires very few string computations and allows us to keep D's records ordered according to a general-purpose indexing function f under the insertion and deletion of individual records (Theorem 2.1).

We can compare our solution to the one obtained by prefix B-trees. They introduce string duplication and require $O(\sum_{i=1}^{k} |K_i|)$ space (usually much larger than k) because they need to store the strings explicitly in the index by means of some heuristics (see Section 5). Conversely, String B-trees exploit lexicographic order better and take advantage of the longest common prefix of any two strings. For example, let α be the longest common prefix of two strings $K_i = \alpha c \beta$ and $K_j = \alpha c' \beta'$, with arbitrarily-long strings α, β, β' and single characters $c \neq c'$. Prefix B-trees [10] store string K_i entirely and string K_j as integer $|\alpha|$ and suffix $c'\beta'$, while String B-trees only use K_i and K_j logical pointers, together with $|\alpha|, c$ and c' in the Patricia trie. Consequently, String B-trees' space usage is proportional to the number of strings involved and not to their total length. We achieve a significant worst-case space saving with respect to prefix B-trees and maintain a very competitive cost.

6.3 Dynamic Suffix Arrays

Dynamic suffix arrays [17] are a dynamic version of suffix arrays [33] and we implement them in *linear* space without using the naming technique of [27]. This allows us to obtain better performance than suffix trees for large alphabets because we can reduce searching time from $O(p \log N + occ)$ to $O(p + \log N + occ)$. This alphabet-independent time bound was previously obtained only for a *static* string set by means of suffix arrays.

The *dynamic suffix array* data structure DSA_Δ combines the flexibility of suffix trees with the lexicographic order of suffix arrays. It is a balanced search tree whose leaves store $SUF(\Delta)$'s strings, in $\leq_L$-order. We let $DSUF(\Delta) \subseteq SUF(\Delta)$ be a set of suffixes *logically deleted* from $SUF(\Delta)$. The dynamic suffix array supports the following operations:

DSA-Search(P): We find the sublist of the suffixes in $SUF(\Delta) - DSUF(\Delta)$ whose prefix is P.

DSA-Insert(Y): We insert all of Y's suffixes into $SUF(\Delta)$.

DSA-Delete(S): We mark a suffix $S \in SUF(\Delta)$ as logically removed and insert it into $DSUF(\Delta)$.

DSA-Undelete(S): We unmark a suffix $S \in DSUF(\Delta)$ and remove it from $DSUF(\Delta)$.

We go on to implement the above operations. We let $B = O(1)$ and use the String B-tree data structure in main memory. It is now a balanced search tree that satisfies the additional constraint that each leaf contains exactly one string. The leaves containing the strings in $SUF(\Delta) - DSUF(\Delta)$ are double-linked in a separate list LS. This list is kept with another list, $lcp(LS)$, that contains the longest common prefix length of any two adjacent strings in LS. The leaves containing $DSUF(\Delta)$'s strings are *marked* as logically deleted and each internal node is also marked if all its descendants are marked recursively. It clearly follows:

Fact 6.2. *Given a leaf s of the String B-tree, the nearest non-marked leaf $s' \in SUF(\Delta) - DSUF(\Delta)$ ON its left (resp., right) can be identified in $O(\log N)$ time.*

The update operations on DSA_Δ can be implemented in a straightforward way by using the corresponding update operations on String B-trees in Sections 4.3 and 4.4. The implementation of DSA-Search(P) is slightly different from the one described in Section 4.1 because it has to take into account the fact that a suffix in $SUF(\Delta)$ having prefix P might be marked as deleted and therefore should not be listed because it belongs to $DSUF(\Delta)$. Our aim now is to find the sublist $\widehat{LS}$ of LS that contains all the suffixes having prefix P within a time complexity that does not depend on $DSUF(\Delta)$'s size, when these suffixes belong to $SUF(\Delta) - DSUF(\Delta)$. We design the search procedure in such a way that if $\widehat{LS}$ is not empty, it returns two leaves v_L and v_R that delimit $\widehat{LS}$. In this way, $\widehat{LS}$'s size does not influence total time complexity.

We find leaves v_L and v_R by searching for the leaf v that stores suffix X_v, such that $X_v = \max\{X \in SUF(\Delta) : X <_L P\}$ according to Theorem 4.4's proof. We then apply Fact 6.2 to leaf v and retrieve v_L by identifying the leftmost unmarked leaf on v's right (inclusive). Leaf v_R can be found symmetrically by means of leaf w, such that $X_w = \min\{X \in SUF(\Delta) : P <_L X\}$. We know that v_L and v_R are between v and w if and only if $\widehat{LS}$ is not empty, and this condition can be checked in $O(\log N)$ time. Moreover, when $v = v_L = v_R$ or $v_L = v_R = w$, we can check to see if P is a prefix of X_v or X_w, respectively. We have:

Theorem 6.3. *The dynamic suffix array DSA_Δ for a string set Δ whose total length is N can be implemented by an augmented String B-tree that occupies optimal $\Theta(N)$ space.* DSA-Search(P) *requires $O(p + \log N)$ time;* DSA-Delete *and* DSA-Undelete *take $O(\log N)$ time;* DSA-Insert *applied on an m-length string takes $O(m \log(N + m))$ time.*

7 Conclusions

In this paper, we have proposed an external-memory data structure, the String B-tree, that efficiently implements operations such as Prefix Search, Range Query, Substring Search, and string insertions and deletions, on a collection of arbitrary-long strings. While its bounds are provably good in the worst case like the ones of regular B-trees, its supported operations are more powerful because it manages strings of arbitrary length. String B-trees can be also successfully applied to several other interesting problems, such as the ones discussed in the introduction and Section 6, and the ones presented in [20]. They also efficiently work in the parallel-disk model [1, 46] by performing disk clustering with the so-called disk-striping technique (see [38] for its description).

Considering their good theoretical bounds, it would be interesting to investigate the practical behavior of String B-trees in order to validate the general approach and single out the theoretical refinements that are also effective in a practical setting. A preliminary set of experiments carried out in [19] have shown that String B-trees are promising: String B-trees lead to fast searches and can be updated in a reasonable amount of time.

Acknowledgments.

We thank Jeffrey Vitter, who showed us a way to adapt the amortized analysis in order to obtain the worst-case bounds for the update operations in Theorem 4.16. We warmly thank Don Knuth who pointed out to us reference [37]. We are grateful to Dany Breslauer for pointing out the relationship between String B-trees and string sorting in the comparison model. We also warmly thank Brenda Baker, Raffaele Giancarlo, Mike Goodrich, Rao Kosaraju, Fabrizio Luccio and Renzo Sprugnoli for their numerous, helpful discussions on the early results of this paper.

References

[1] Aggarwal, A., and Vitter, J. S. The Input/Output complexity of sorting and related problems. *Communications of the ACM* (1988), 1116–1127.

[2] Aho, A. V., Hopcroft, J. E., and Ullman,J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] Amir, A., Farach, M., Galil, Z., Giancarlo, R., and Park, K. Dynamic dictionary matching. *Journal of Computer and System Science 49* (1994), 208–222.

[4] Amir, A., Farach, M., and Muthukrishnan, S. Alphabet dependence in parameterized matching. *Information Processing Letters 49* (1994), 111–115.

[5] Andersson, A., and Nilsson, S. Efficient implementation of suffix trees. *Software–Practice and Experience 25*, 2 (1995), 129–141.

[6] Apostolico, A. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words* (1985), A. Apostolico and Z. Galil, Eds., NATO ASI Series F: Computer and System Sciences, Springer-Verlag, pp. 85–96.

[7] Baker, B. S. A theory of parameterized pattern matching: Algorithms and applications. In *ACM Symposium on Theory of Computing* (1993), pp. 71–80.

[8] Baker, B. S. Parameterized pattern matching by Boyer-Moore-type algorithms. In *ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 541–550.

[9] Bayer, R., and McCreight, C. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (1972), 173–189.

[10] Bayer, R., and Unterauer, K. Prefix B-trees. *ACM Trans. Database Syst. 2*, 1 (1977), 11–26.

[11] Bentley, J. L., and Sedgewick, R. Fast algorithms for sorting and searching strings. In *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1996), pp. 360–369.

[12] Church, K. W., and Rau, L. F. Commercial applications of natural language processing. *Communications of the ACM 38* (1995), 71–79.

[13] Clark, D. R., and Munro, J. I. Efficient suffix trees on secondary storage. In *ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 383–391.

[14] Cleary, J. G., and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications 32* (1984), 396–402.

[15] Comer, D. The ubiquitous B-Tree. *Computing Surveys 11* (1979), 121–137.

[16] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[17] Ferragina, P., and Grossi, R. Fast incremental text editing. In *ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 531–540.

[18] Ferragina, P., and Grossi, R. A fully-dynamic data structure for external substring search. In *ACM Symposium on the Theory of Computing* (1995), pp. 693–702.

[19] Ferragina, P., and Grossi, R. Fast string searching in secondary storage: Theoretical developments and experimental results. In *ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 373–382.

[20] Ferragina, P., and Luccio, L. Dynamic Dictionary Matching in External Memory. *Information and Computation*, 1998 (to appear).

[21] Frenkel, K. A. The human genome project and informatics. *Communications of the ACM 34* (1991), 41–51.

[22] Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992, ch. 5:New indices for text: PAT trees and PAT arrays, pp. 66–82.

[23] Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., and Vitter, J. S. External-memory computational geometry. In *IEEE Foundations of Computer Science* (1993), pp. 714–723.

[24] Gray, H. J., and Prywes, N. S. Outline for a multilist organized system. *Paper 41, Ann. Meeting of the ACM* (1959).

[25] Gusfield, D., Landau, G. M., and Schieber, B. An efficient algorithm for all pairs suffix-prefix problem. *Information Processing Letters 41* (1992), 181–185.

[26] Idury, R. M., and Schäffer, A. A. Multiple matching of parameterized patterns. *Theoretical Computer Science 154* (1996), 203–224.

[27] Karp, R., Miller, R., and Rosenberg, A. Rapid identification of repeated patterns in strings, arrays and trees. In *ACM Symposium on Theory of Computing* (1972), pp. 125–136.

[28] Kephart, J., Sorkin, G., Arnold, W., Chess, D., Tesauro, G., and White, S. Biologically inspired defenses against computer viruses. In *International Joint Conference on Artificial Intelligence* (1995), pp. 1–12.

[29] Knuth, D. E. *The Art of Computer Programming.* Addison-Wesley, 1973, ch. 3: Sorting and Searching.

[30] Kosaraju, R. Faster algorithms for the construction of parameterized suffix trees. In *IEEE Foundations of Computer Science* (1995), pp. 631–639.

[31] Lum, V. Y. Multi-attribute retrieval with combined indexes. *Comm. ACM 13*, 11 (1970), 660–665.

[32] Maier, D., and Salveter, S. C. Hysterical B-trees. *Information Processing Letters 12*, 4 (1981), 199–202.

[33] Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22*, 5 (1993), 935–948.

[34] McCreight, E. M. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2 (1976), 262–272.

[35] Moffat, A. Implementing the PPM data compression scheme. *IEEE Transactions on Communications 38* (1990), 1917–1921.

[36] Morrison, D. R. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15* (1968), 514–534.

[37] O'Neil P. E. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *ACTA Informatica 29* (1992), 241–265.

[38] Patt, N. P. *IEEE Computer: Special Issue "The I/O Subsystem: A candidate for improvement"* (1994).

[39] Prywes, N. S., and Gray, H. J. The organization of a Multilist-type associative memory. *IEEE Trans. on Communication and Electronics 68* (1963), 488–492.

[40] Shang, H. *Trie methods for text and spatial data structures on secondary storage.* PhD thesis, McGill University, 1995.

[41] Sprugnoli, R. On the allocation of binary trees in secondary storage. *BIT 21* (1981), 305–316.

[42] Subramanian, S., and Ramaswamy, S. The P-range tree: A new data structure for range searching in secondary memory. In *ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 378–387.

[43] Tarjan, R. A class of algorithms that require nonlinear time to maintain disjoint sets. *Journal of Computer and System Science 18* (1979), 110–127.

[44] Tarjan, R. E. *Data Structures and Network Algorithms*, vol. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1983.

[45] Vitter, J. S. *Algorithmica: Special Issue on "Large-Scale Memories" 12* (1994).

[46] Vitter, J. S., and Shriver, E. A. Algorithms for parallel memory: Two-level memories. *Algorithmica 12* (1994), 110–147.

[47] Weinberger, P. J. Unix B-trees. Tech. rep., AT&T Bell Laboratories (personal communication).

[48] Weiner, P. Linear pattern matching algorithm. In *IEEE Symp. on Switching and Automata Theory* (1973), pp. 1–11.

[49] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Info. Theory 23*, 3 (1977), 337–343.

[50] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theory 24*, 5 (1978), 530–536.