7. Full-Text Indexes in External Memory

Juha Kärkkäinen* and S. Srinivasa Rao

7.1 Introduction

A *full-text index* is a data structure storing a text (a string or a set of strings) and supporting *string matching queries*: Given a pattern string P, find all occurrences of P in the text. The best-known full-text index is the suffix tree [761], but numerous others have been developed. Due to their fast construction and the wealth of combinatorial information they reveal, full-text indexes (and suffix trees in particular) also have many uses beyond basic string matching. For example, the number of distinct substrings of a string or the longest common substrings of two strings can be computed in linear time [231]. Gusfield [366] describes several applications in computational biology, and many others are listed in [359].

Most of the work on full-text indexes has been done on the RAM model, i.e., assuming that the text and the index fit into the internal memory. However, the size of digital libraries, biosequence databases and other textual information collections often exceed the size of the main memory on most computers. For example, the GenBank [107] database contains more than 20 GB of DNA sequences in its August 2002 release. Furthermore, the size of a full-text index is usually 4–20 times larger than the size of the text itself [487]. Finally, if an index is needed only occasionally over a long period of time, one has to keep it either in internal memory reducing the memory available to other tasks or on disk requiring a costly loading into memory every time it is needed.

In their standard form, full-text indexes have poor memory locality. This has led to several recent results on adapting full-text indexes to external memory. In this chapter, we review the recent work focusing on two issues, full-text indexes supporting I/O-efficient string matching queries (and updates), and external memory algorithms for constructing full-text indexes (and for sorting strings, a closely related task).

We do not treat other string techniques in detail here. Most string matching algorithms that do not use an index work by scanning the text more or less sequentially (see, e.g., [231, 366]), and are relatively trivial to adapt to an externally stored text. Worth mentioning, however, are algorithms that may generate very large automata in pattern preprocessing, such as [486, 533, 573, 735, 770], but we are not aware of external memory versions of these algorithms.

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

In information retrieval [85, 314], a common alternative to full-text indexes is the *inverted file* [460], which takes advantage of the natural division of linguistic texts into a limited number of distinct words. An inverted file stores each distinct word together with a list of pointers to the occurrences of the word in the text. The main advantage of inverted files is their space requirement (about half of the size of the text [85]), but they cannot be used with unstructured texts such as biosequences. Also, the space requirement of the data structures described here can be significantly reduced when the text is seen as a sequence of atomic words (see Section 7.3.2).

Finally, we mention another related string technique, compression. Two recent developments are compressed indexes [299, 300, 361, 448, 648, 649] and sequential string matching in compressed text without decompression [40, 289, 447, 575, 576]. Besides trying to fit the text or index into main memory, these techniques can be useful for reducing the time for moving data from disk to memory.

7.2 Preliminaries

We begin with a formal description of the problems and the model of computation.

The Problems Let us define some terminology and notation. An *alphabet* Σ is a finite ordered set of *characters*. A *string* S is an array of characters, $S[1,n] = S[1]S[2] \dots S[n]$. For $1 \le i \le j \le n$, $S[i,j] = S[i] \dots S[j]$ is a *substring* of S, $S[1,j]$ is a *prefix* of S, and $S[i,n]$ is a *suffix* of S. The set of all strings over alphabet Σ is denoted by Σ^*.

The main problem considered here is the following.

Problem 7.1 (Indexed String Matching). Let the *text* $\mathcal{T}$ be a set of K strings in Σ^* with a total length N. A *string matching query* on the text is: Given a *pattern* $P \in \Sigma^*$, find all occurrences of P as a substring of the strings in $\mathcal{T}$. The static problem is to store the text in a data structure, called a *full-text index*, that supports string matching queries. The dynamic version of the problem additionally requires support for insertion and deletion of strings into/from $\mathcal{T}$.

All the full-text indexes described here have a linear space complexity. Therefore, the focus will be on the time complexity of queries and updates (Section 7.4), and of construction (Section 7.5).

Additionally, the *string sorting problem* will be considered in Section 7.5.5.

Problem 7.2 (String Sorting). Given a set $\mathcal{S}$ of K strings in Σ^* with a total length N, sort them into the lexicographic order.

The Model Our computational model is the standard external memory model introduced in [17, 755] and described in Chapter 1 of this volume. In particular, we use the following main parameters:

N = number of characters in the text or in the strings to be sorted
M = number of characters that fit into the internal memory
B = number of characters that fit into a disk block

and the following shorthand notations:

$$\begin{aligned} \mathrm{scan}(N) &= \Theta\,(N/B) \\ \mathrm{sort}(N) &= \Theta\left((N/B)\log_{M/B}(N/B)\right) \\ \mathrm{search}(N) &= \Theta\,(\log_B N) \end{aligned}$$

The following parameters are additionally used:

K = number of strings in the text or in the set to be sorted
Z = size of the answer to a query (the number of occurrences)
$|\Sigma|$ = size of the alphabet
$|P|$ = number of characters in a pattern P
$|S|$ = number of characters in an inserted/deleted string S

For simplicity, we mostly ignore the space complexity, the CPU complexity, and the parallel (multiple disks) I/O complexity of the algorithms. However, significant deviations from optimality are noted.

With respect to *string representation*, we mostly assume the *integer alphabet model*, where characters are integers in the range $\{1,\ldots,N\}$. Each character occupies a single machine word, and all usual integer operations on characters can be performed in constant time. For internal memory computation, we sometimes assume the *constant alphabet model*, which differs from the integer alphabet model in that dictionary operations on sets of characters can be performed in constant time and linear space.[1] Additionally, the *packed string model* is discussed in 7.5.6.

7.3 Basic Techniques

In this section, we introduce some basic techniques. We start with the (for our purposes) most important internal memory data structures and algorithms. Then, we describe two external memory techniques that are used more than once later.

[1] With techniques such as hashing, this is *nearly* true even for the integer alphabet model. However, integer dictionaries are a complex issue and outside the scope of this article.

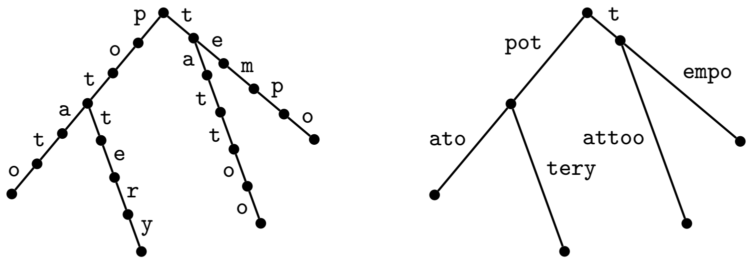

Fig. 7.1. Trie and compact trie for the set {`potato`, `pottery`, `tattoo`, `tempo`}

7.3.1 Internal Memory Techniques

Most full-text indexes are variations of three data structures, suffix arrays [340, 528], suffix trees [761] and DAWGs (Directed Acyclic Word Graphs) [134, 230]. In this section, we describe suffix arrays and suffix trees, which form the basis for the external memory data structures described here. We are not aware of any adaptation of DAWG for external memory.

Let us start with an observation that underlies almost all full-text indexes. If an occurrence of a pattern P starts at position i in a string $S \in \mathcal{T}$, then P is a prefix of the suffix $S[i, |S|]$. Therefore, we can find all occurrences of P by performing a prefix search query on the set of all suffixes of the text: A *prefix search query* asks for all the strings in the set that contain the query string P as a prefix. Consequently, a data structure that stores the set of all suffixes of the text and supports prefix searching is a full-text index.

The simplest data structure supporting efficient prefix searching is the lexicographically sorted array, where the strings with a given prefix always form a contiguous interval. The *suffix array* of a text $\mathcal{T}$, denoted by $SA_{\mathcal{T}}$, is the sorted array of pointers to the suffixes of $\mathcal{T}$ (see Fig. 7.2). By a binary search, a string matching (prefix search) query can be answered with $\mathcal{O}(\log_2 N)$ string comparisons, which needs $\mathcal{O}(|P| \log_2 N)$ time in the worst case. Manber and Myers [528] describe how the binary search can be done in $\mathcal{O}(|P| + \log_2 N)$ time if additional (linear amount of) information is stored about longest common prefixes. Manber and Myers also show how the suffix array can be constructed in time $\mathcal{O}(N \log_2 N)$. Suffix arrays do not support efficient updates.

The trie is another simple data structure for storing a set of strings [460]. A *trie* (see Fig. 7.1) is a rooted tree with edges labeled by characters. A node in a trie represents the concatenation of the edge labels on the path from the root to the node. A trie for a set of strings is the minimal trie whose nodes represent all the strings in the set. If the set is *prefix free*, i.e., no string is a proper prefix of another string, all the nodes representing the strings are leaves. A *compact trie* is derived from a trie by replacing each maximal branchless path with a single edge labeled by the concatenation of the replaced edge labels (see Fig. 7.1).

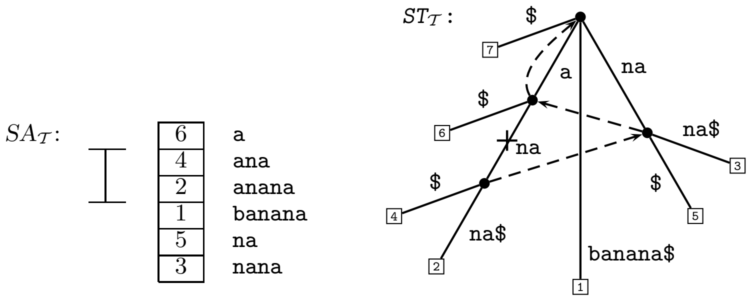

Fig. 7.2. Suffix array $SA_{\mathcal{T}}$ and suffix tree $ST_{\mathcal{T}}$ for the text $\mathcal{T} = \{\texttt{banana}\}$. For the suffix tree, a sentinel character `$` has been added to the end. Suffix links are shown with dashed arrows. Also shown are the answers to a string matching query $P = \texttt{an}$: in $SA_{\mathcal{T}}$ the marked interval, in $ST_{\mathcal{T}}$ the subtree rooted at $+$. Note that the strings shown in the figure are not stored explicitly in the data structures but are represented by pointers to the text.

The *suffix tree* of a text $\mathcal{T}$, denoted by $ST_{\mathcal{T}}$, is the compact trie of the set of suffixes of $\mathcal{T}$ (see Fig. 7.2). With suffix trees, it is customary to add a sentinel character `$` to the end of each string in $\mathcal{T}$ to make the set of suffixes prefix free. String matching (prefix searching) in a suffix tree is done by walking down the tree along the path labeled by the pattern (see Fig. 7.2). The leaves in the subtree rooted at where the walk ends represent the set of suffixes whose prefix is the pattern. The time complexity is $\mathcal{O}(|P|)$ for walking down the path (under the constant alphabet model) and $\mathcal{O}(Z)$ for searching the subtree, where Z is the size of the answer.

The suffix tree has $\mathcal{O}(N)$ nodes, requires $\mathcal{O}(N)$ space, and can be constructed in $\mathcal{O}(N)$ time. Most linear-time construction algorithms, e.g. [540, 736, 761], assume the constant alphabet model, but Farach's algorithm [288] also works in the integer alphabet model. All the fast construction algorithms rely on a feature of suffix trees called suffix links. A *suffix link* is a pointer from a node representing the string $a\alpha$, where a is a single character, to a node representing α (see Fig. 7.2). Suffix links are not used in searching but they are necessary for an insertion or a deletion of a string S in time $\mathcal{O}(|S|)$ [297] (under the constant alphabet model).

7.3.2 External Memory Techniques

In this section, we describe two useful string algorithm techniques, Patricia tries and lexicographic naming. While useful for internal memory algorithms too, they are particularly important for external memory algorithms.

Suffix arrays and trees do not store the actual strings they represent. Instead, they store pointers to the text and access the text whenever necessary. This means that the text is accessed frequently, and often in a nearly random manner. Consequently, the performance of algorithms is poor when the text is stored on disk. Both of the techniques presented here address this issue.

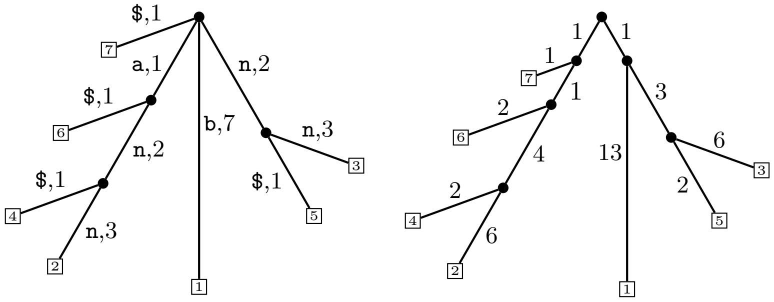

Fig. 7.3. Pat tree $PT_{\mathcal{T}}$ for $\mathcal{T} = \{\texttt{banana\$}\}$ using native encoding and binary encoding. The binary encoding of characters is `$=00`, `a=01`, `b=10`, `n=11`.

The first technique is the *Patricia trie* [557], which is a close relative of the compact trie. The difference is that, in a Patricia trie, the edge labels contain only the first character (branching character) and the length (skip value) of the corresponding compact trie label. The Patricia trie for the set of suffixes of a text $\mathcal{T}$, denoted by $PT_{\mathcal{T}}$, is called the *Pat tree* [340]. An example is given in Fig. 7.3.

The central idea of Patricia tries and Pat trees is to delay access to the text as long as possible. This is illustrated by the string matching procedure. String matching in a Pat tree proceeds as in a suffix tree except only the first character of each edge is compared to the corresponding character in the pattern P. The length/skip value tells how many characters are skipped. If the search succeeds (reaches the end of the pattern), all the strings in the resulting subtree have the same prefix of length $|P|$. Therefore, either all of them or none of them have the prefix P. A single string comparison between the pattern and some string in the subtree is required to find out which is the case. Thus, the string matching time is $\mathcal{O}(|P| + Z)$ as with the suffix tree, but there is now only a single contiguous access to the text.

Any string can be seen as a binary string through a *binary encoding* of the characters. A prefix search on a set of such binary strings is equivalent to a prefix search on the original strings. Patricia tries and Pat trees are commonly defined to use the binary encoding instead of the native encoding, because it simplifies the structure in two ways. First, every internal node has degree two. Second, there is no need to store even the first bit of the edge label because the left/right distinction already encodes for that. An example is shown in Fig. 7.3.

The second technique is lexicographic naming introduced by Karp, Miller and Rosenberg [450]. A *lexicographic naming* of a (multi)set $\mathcal{S}$ of strings is an assignment of an integer (the name) to each string such that any order comparison of two names gives the same result as the lexicographic order comparison of the corresponding strings. Using lexicographic names, arbitrarily long strings can be compared in constant time without a reference to

4	ban
2	ana
6	nan
2	ana
5	na$
1	a$$

4	banana
3	anana
6	nana
2	ana
5	na
1	a

Fig. 7.4. Lexicographic naming of the substrings of length three in `banana$$`, and of the suffixes of `banana`

the actual strings. The latter property makes lexicographic naming a suitable technique for external memory algorithms.

A simple way to construct a lexicographic naming for a set $\mathcal{S}$ is to sort $\mathcal{S}$ and use the rank of a string as its name, where the rank is the number of lexicographically smaller strings in the set (plus one). Fig. 7.4 displays two examples that are related to the use of lexicographic naming in Section 7.5.

Lexicographic naming has an application with linguistic texts, where words can be considered as 'atomic' elements. As mentioned in the introduction, inverted files are often preferred to full-text indexes in this case because of their smaller space requirement. However, the space requirement of full-text indexes (at least suffix arrays) can be reduced to the same level by storing only suffixes starting at the beginning of a word [340] (making them no more full-text indexes). A problem with this approach is that most fast construction algorithms rely on the inclusion of all suffixes. A solution is to apply lexicographic naming to the set of distinct words and transform the text into strings of names. Full-text indexes on such transformed texts are called *word-based indexes* [46, 227].

7.4 I/O-Efficient Queries

In this section, we look at some I/O-efficient index structures. In particular, we look at the structures described by Baeza-Yates et al. [83], Clark and Munro [206] and Ferragina and Grossi [296]. We then briefly sketch some recent results.

7.4.1 Hierarchies of Indexes

Baeza-Yates et al. [83] present an efficient implementation of an index for text databases when the database is stored in external memory. The implementation is built on top of a suffix array. The best known internal memory algorithm, of Manber and Myers [528], for string matching using a suffix array is not I/O-efficient when the text and the index reside in external memory. Baeza-Yates et al. propose additional index structures and searching algorithms for suffix arrays that reduce the number of disk accesses. In

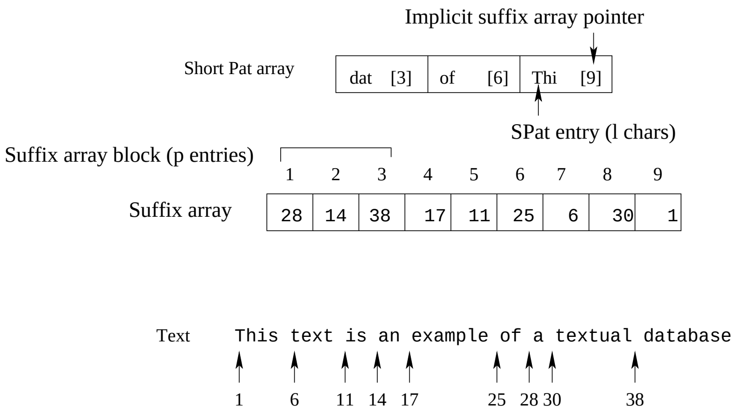

Fig. 7.5. Short Pat array

particular, they introduce two index structures for two and three level memory hierarchy (that use main memory and one/two levels of external storage), and present experimental and analytical results for these. These additional index structures are much smaller in terms of space compared to the text and the suffix array. Though, theoretically these structures only improve the performance by a constant factor, one can adjust the parameters of the structure to get good practical performance. Here, we briefly describe the structure for a two-level hierarchy. One can use a similar approach for building efficient indexes for a steeper hierarchy.

Two-Level Hierarchy. The main idea is to divide the suffix array into blocks of size p, where p is a parameter, and move one element of each block into main memory, together with the first few characters of the corresponding suffix. This structure can be considered a reduced representation of the suffix array and the text file, and is called Short Pat array or SPat array[2]. The SPat array is a set of suffix array entries where each entry also carries a fixed number, say ℓ, of characters from the text, where ℓ is a parameter (see Fig. 7.5). Due to the additional information about the text in the SPat array, a binary search can be performed directly without accessing the disk. As a result, most of the searching work is done in main memory, thus reducing the number of disk accesses. Searching for a pattern using this structure is done in two phases:

First, a binary search is performed on the SPat array, with no disk accesses, to find the suffix array block containing the pattern occurrence. Additional disk accesses are necessary only if the pattern is longer than ℓ and there are multiple entries in the SPat array that match the prefix of length ℓ of the pattern. Then, $\mathcal{O}(\log_2 r)$ disk accesses are needed, where r is the number matching entries.

[2] A suffix array is sometimes also referred to as a Pat array.

Second, the suffix array block encountered in the first phase is moved from disk to main memory. A binary search is performed between main memory (suffix array block containing the answer) and disk (text file) to find the first and last entries that match the pattern. If the pattern occurs more than p times in the text, these occurrences may be bounded by at most two SPat array entries. In this case the left and right blocks are used in the last phase of the binary search following the same procedure.

The main advantages of this structure are its space efficiency (little more than a suffix array) and ease of implementation. See [83] for the analytical and experimental results. This structure does not support updates efficiently.

7.4.2 Compact Pat Trees

Clark and Munro [206] have presented a Pat tree data structure for full-text indexing that can be adapted to external memory to reduce the number of disk accesses while searching, and also handles updates efficiently. It requires little more storage than $\log_2 N$ bits per suffix, required to store the suffix array. It uses a compact tree encoding to represent the tree portion of the Pat tree and to obtain an efficient data structure for searching static text in primary storage. This structure, called a *Compact Pat Tree* (CPT), is then used to obtain a data structure for searching on external memory. The main idea here is to partition the Pat tree into pieces that fit into a disk block, so that no disk accesses are required while searching within a partition.

Compact Representation. To represent the Pat tree in a compact form, first the strings are converted to binary using a binary encoding of the characters, as explained in Section 7.3.2, which gets rid of the space needed to store the edge labels. The underlying binary tree (without the skip values and the pointers to the suffixes at the leaves) is then stored using an encoding similar to the well known compact tree encoding of Jacobson [425]. The compact tree representation of Clark and Munro takes less than three bits per node to represent a given binary tree and supports the required tree navigational operations (parent, left child, right child and subtree size) in constant time.

For storing the skip values at the internal nodes in a Pat tree, they use the observation that large skip values are unlikely (occur very rarely). This low likelihood of large skip values leads to a simple method of compactly encoding the skip values. A small fixed number of bits are reserved to hold the skip value for each internal node. Problems caused by overflows are handled by inserting a new node and a leaf into the tree and distributing the skip bits from the original node across the skip fields of the new node and the original node. A special key value that can be easily recognized (say all 0s) is stored with these dummy leaf nodes. Multiple overflow nodes and leaves can be inserted for extremely large skip values. Note that skip values are not needed at the leaves, as we store the pointers to the suffixes at the leaves.

Under the assumption that the given text (in binary) is generated by a uniform symmetric random process, and that the bit strings in the suffixes are independent, Clark and Munro show that the expected size of the CPT can be made less than $3.5 + \log_2 N + \log_2 \log_2 N + \mathcal{O}(\log_2 \log_2 \log_2 N / \log_2 N)$ bits per node. This is achieved by setting the skip field size to $\log_2 \log_2 \log_2 N$. They also use some space saving techniques to reduce the storage requirement even further, by compromising on the query performance.

External Memory Representation. To control the accesses to the external memory during searching, Clark and Munro use the method of decomposing the tree into disk block sized pieces, each called a partition. Each partition of the tree is stored using the CPT structure described above. The only change required to the CPT structure for storing the partitions is that the offset pointers in a block may now point to either a suffix in the text or to a subtree (partition). Thus an extra bit is required to distinguish these two cases. They use a greedy bottom-up partitioning algorithm and show that such a partitioning minimizes the maximum number of disk blocks accessed when traversing from the root to any leaf. While the partitioning rules described by Clark and Munro minimize the maximum number of external memory accesses, these rules can produce many small pages and poor fill ratios. They also suggest several methods to overcome this problem. They show that the maximum number of pages traversed on any root to leaf path is at most $1 + \lceil H/\sqrt{B} \rceil + \lceil 2 \log_B N \rceil$, where H is the height of the Pat tree. Thus searching using the CPT structure takes $\mathcal{O}(\text{scan}(|P| + Z) + \text{search}(N))$ I/Os, assuming that the height H is $\mathcal{O}(\sqrt{B} \log_B N)$. Although H could be $\Theta(N)$ in the worst case, it is logarithmic for a random text under some reasonable conditions on the distribution [711].

Updates. The general approach to updating the static CPT representation is to search each suffix of the modified document and then make appropriate changes to the structure based on the path searched. While updating the tree, it may become necessary to re-partition the tree in order to retain the optimality. The solution described by Clark and Munro to insert or delete a suffix requires time proportional to the depth of the tree, and operates on the compact form of the tree. A string is inserted to or deleted from the text by inserting/deleting all its suffixes separately. See [206] for details and some experimental results.

7.4.3 String B-trees

Ferragina and Grossi [296] have introduced the *string B-tree* which is a combination of B-trees (see Chapter 2) and Patricia tries. String B-trees link external memory data structures to string matching data structures, and overcome the theoretical limitations of inverted files (modifiability and atomic keys), suffix arrays (modifiability and contiguous space) and Pat trees (unbalanced tree topology). It has the same worst case performance as B-trees but handles unbounded length strings and performs powerful search operations such as the ones supported by Pat trees. String B-trees have also been applied to

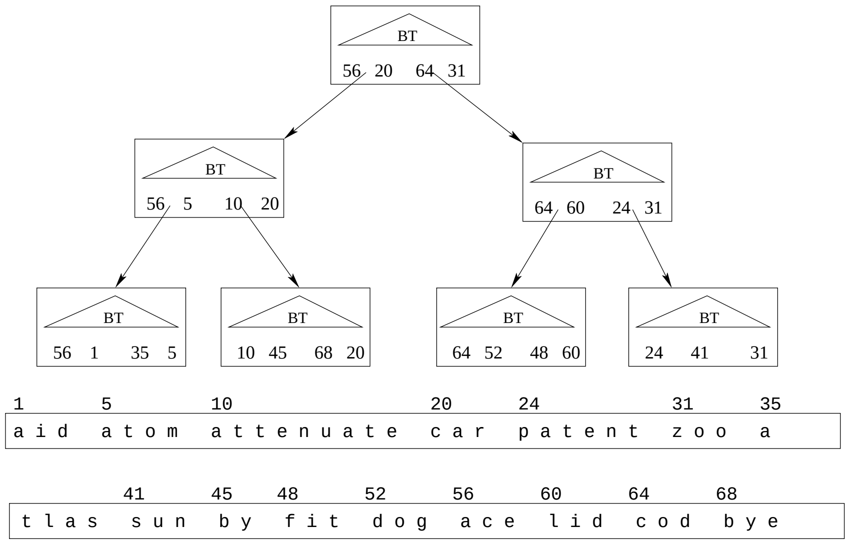

Fig. 7.6. String B-tree

(external and internal) dynamic dictionary matching [298] and some other internal memory problems [296].

String B-trees are designed to solve the dynamic version of the indexed string matching problem (Problem 1). For simplicity, we mainly describe the structure for solving the prefix search problem. As mentioned in Section 7.3.1, a string matching query can be supported by storing the suffixes of all the text strings, and supporting prefix search on the set of all suffixes.

String B-tree Data Structure. Given a set $\mathcal{S} = \{s_1, \ldots, s_N\}$ of N strings (the suffixes), a string B-tree for $\mathcal{S}$ is a B-tree in which all the keys are stored at the leaves and the internal nodes contain copies of some of these keys. The keys are the logical pointers to the strings (stored in external memory) and the order between the keys is the lexicographic order among the strings pointed to by them. Each node v of the string B-tree is stored in a disk block and contains an ordered string set $\mathcal{S}_v \subseteq \mathcal{S}$, such that $b \leq |\mathcal{S}_v| \leq 2b$, where $b = \Theta(B)$ is a parameter which depends on the disk block size B. If we denote the leftmost (rightmost) string in $\mathcal{S}_v$ by $L(v)$ $(R(v))$, then the strings in $\mathcal{S}$ are distributed among the string B-tree nodes as follows (see Fig. 7.6 for an example):

– Partition $\mathcal{S}$ into groups of b strings except for the last group, which may contain from b to $2b$ strings. Each group is mapped into a leaf v (with string set $\mathcal{S}_v$) in such a way that the left-to-right scanning of the string B-tree leaves gives the strings in $\mathcal{S}$ in lexicographic order. The longest common prefix length $lcp(S_j, S_{j+1})$ is associated with each pair (S_j, S_{j+1}) of $\mathcal{S}_v$'s strings.

– Each internal node v of the string B-tree has $d(v)$ children $u_1, \ldots, u_{d(v)}$, with $b/2 \leq d(v) \leq b$ (except for the root, which has from 2 to b children). The set $\mathcal{S}_v$ is formed by copying the leftmost and rightmost strings contained in each of its children, from left to right. More formally, $\mathcal{S}_v$ is the ordered string set $\{L(u_1), R(u_1), L(u_2), R(u_2), \ldots, L(u_{d(v)}), R(u_{d(v)})\}$.

Since the branching factor of the string B-tree is $\Theta(B)$, its height is $\Theta(\log_B N)$.

Each node v of the string B-tree stores the set $\mathcal{S}_v$ (associated with the node v) as a Patricia trie (also called a *blind trie*). To maximize the number b of strings stored in each node for a given value of B, these blind tries are stored in a succinct form (the tree encoding of Clark and Munro, for example). When a node v is transferred to the main memory, the explicit representation of its blind trie is obtained by uncompressing the succinct form, in order to perform computation on it.

Search Algorithm. To search for a given pattern P, we start from the root of the string B-tree and follow a path to a leaf, searching for the position of P at each node. At each internal node, we search for its child node u whose interval $[L(u), R(u)]$ contains P. The search at node v is done by first following the path governed by the pattern to reach a leaf l in the blind trie. If the search stops at an internal node because the pattern has exhausted, choose l to be any descendant leaf of that node. This leaf does not necessarily identify the position of P in $\mathcal{S}_v$, but it provides enough information to find this position, namely, it points to one of the strings in $\mathcal{S}_v$ that shares the longest common prefix with P. Now, we compare the string pointed to by l with P to determine the length p of their longest common prefix. Then we know that P matches the search path leading to l up to depth p, and the mismatch character $P[p+1]$ identifies the branches of the blind trie between which P lies, allowing us to find the position of P in S_v. The search is then continued in the child of v that contains this position.

Updates. To insert a string S into the set $\mathcal{S}$, we first find the leaf v and the position j inside the leaf where S has to be inserted by searching for the string S. We then insert S into the set $\mathcal{S}_v$ at position j. If $L(v)$ or $R(v)$ change in v, then we extend the change to v's ancestor. If v gets full (i.e., contains more than $2b$ strings), we split the node v by creating a new leaf u and making it an adjacent leaf of v. We then split the set S_v into two roughly equal parts of at least b strings each and store them as the new string sets for v and u. We copy the strings $L(v)$, $R(v)$, $L(u)$ and $R(u)$ in their parent node, and delete the old strings $L(v)$ and $R(v)$. If the parent also gets full, then we split it. In the worst case the splitting can extend up to the root and the resulting string B-tree's height can increase by one. Deletions of the strings are handled in a similar way, merging a node with its adjacent node whenever it gets half-full. The I/O complexity of insertion or deletion of a string S is $\mathcal{O}(\mathrm{scan}(|S|) + \mathrm{search}(N))$.

For the dynamic indexed string matching problem, to insert a string S into the text, we have to insert all its suffixes. A straightforward way of doing this requires $\mathcal{O}(\text{scan}(|S|^2) + |S|\text{search}(N + |S|))$ I/Os. By storing additional information with the nodes (similar to the suffix links described in Section 7.3.1), the quadratic dependence on the length of S can be eliminated. The same holds for deletions.

Results. Using string B-tree, one can get the following bounds for the dynamic indexed string matching problem:

Theorem 7.3. *The string B-tree of a text $\mathcal{T}$ of total length N supports*

- *string matching with a pattern P in $\mathcal{O}(\text{scan}(|P| + Z) + \text{search}(N))$ I/Os,*
- *inserting or deleting a string S into/from $\mathcal{T}$ in $\mathcal{O}(|S|\text{search}(N + |S|))$ I/Os,*

and occupies $\Theta(N/B)$ disk blocks.

The space occupied by the string B-tree is asymptotically optimal, as the space required to store the given set of strings is also $\Theta(N/B)$ disk blocks. Also the string B-tree operations take asymptotically optimal CPU time, that is, $\mathcal{O}(Bd)$ time if d disk blocks are read or written, and they only need to keep a constant number of disk blocks in the main memory at any time.

See [295] for some experimental results on string B-trees.

7.4.4 Other Data Structures

Recently, Ciriani et al. [205] have given a randomized data structure that supports lexicographic predecessor queries (which can be used for implementing prefix searching) and achieves optimal time and space bounds in the amortized sense. More specifically, given a set of N strings $S_1, \ldots, S_N$ and a sequence of m patterns $P_1, \ldots, P_m$, their solution takes $\mathcal{O}(\sum_{i=1}^{m} \text{scan}(|P_i|) + \sum_{i=1}^{N}(n_i \log_B(m/n_i))$ expected amortized I/Os, where n_i is the number of times S_i is the answer to a query. Inserting or deleting a string S takes $\mathcal{O}(\text{scan}(|S|) + \text{search}(N))$ expected amortized I/Os. The search time matches the performance of string B-trees for uniform distribution of the answers, but improves on it for biased distributions. This result is the analog of the Static Optimality Theorem of Sleator and Tarjan [699] and is achieved by designing a self-adjusting data structure based on the well-known skip lists [616].

7.5 External Construction

There are several efficient algorithms for constructing full-text indexes in internal memory [288, 528, 540, 736]. However, these algorithms access memory in a nearly random manner and are poorly suited for external construction. String B-trees provide the possibility of construction by insertion, but the construction time of $\mathcal{O}(N\text{search}(N))$ I/Os can be improved with specialized construction algorithms.

In this section, we describe several I/O-efficient algorithms for external memory construction of full-text indexes. We start by showing that the different full-text indexes can be transformed into each other efficiently. Therefore, any construction algorithm for one type of index works for others, too. Then, we describe two practical algorithms for constructing suffix arrays, and a theoretically optimal algorithm for constructing Pat trees. We also a look at the related problem of sorting strings.

For the sake of clarity, we assume that the text consists of a single string of length N, but all the algorithms can be easily modified to construct the full-text index of a set of strings of total length N with the same complexity. Unless otherwise mentioned, disk space requirement, CPU time, and speedup with parallel disks are optimal.

7.5.1 Construction from Another Index

In this section, we show that the different forms of full-text indexes we have seen are equivalent in the sense that any of them can be constructed from another in $\mathcal{O}(\text{sort}(N))$ I/Os. To be precise, this is not true for the plain suffix array, which needs to be augmented with the longest common prefix array: $LCP[i]$ is the *longest common prefix* of the suffixes starting at $SA[i-1]$ and $SA[i]$. The suffix array construction algorithms described below can be modified to construct the LCP array, too, with the same complexity. The transformation algorithms are taken from [290].

We begin with the construction of a suffix array SA (and the LCP array) from a suffix tree ST (or a Pat tree PT which has the same structure differing only in edge labels). We assume that the children of a node are ordered lexicographically. First, construct the Euler tour of the tree in $\mathcal{O}(\text{sort}(N))$ I/Os (see Chapter 3). The order of the leaves of the tree in the Euler tour is the lexicographic order of the suffixes they represent. Thus, the suffix array can be formed by a simple scan of the Euler tour. Furthermore, let w be the highest node that is between two adjacent leaves u and v in the Euler tour. Then, w is the lowest common ancestor of u and v, and the depth of w is the length of the longest common prefix of the suffixes that u and v represent. Thus LCP can also be computed by a scan of the Euler tour.

The opposite transformation, constructing ST (or PT) given SA and LCP, proceeds by inserting the suffixes into the tree in lexicographic order, i.e., inserting the leaves from left to right. Thus, a new leaf u always becomes the rightmost child of a node, say v, on the rightmost path in the tree (see Fig. 7.7). Furthermore, the longest common prefix tells the depth of v (the insertion depth of u). The nodes on the rightmost path are kept in a stack with the leaf on top. For each new leaf u, nodes are popped from the stack until the insertion depth is reached. If there was no node at the insertion depth, a new node v is created there by splitting the edge. After inserting u as the child of v, v and u are pushed on the stack. All the stack operations can be performed with $\mathcal{O}(\text{scan}(N))$ I/Os using an external stack (see Chapter 2). The

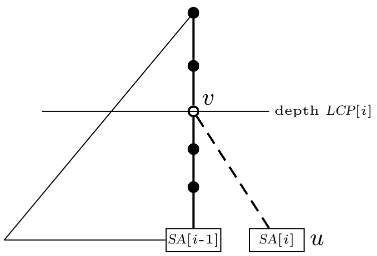

Fig. 7.7. Inserting a new leaf u representing the suffix $SA[i]$ into the suffix tree

construction numbers the nodes in the order they are created and represents the tree structure by storing with each node its parent's number. Other tree representations can then be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.

The string B-tree described in Section 7.4.3 can also be constructed from the suffix array and the *LCP* array in $\mathcal{O}(\text{sort}(N))$ I/Os with a procedure similar to the suffix tree construction. The opposite transformation is also similar.

7.5.2 Merging Algorithm

The merging algorithm was introduced by Gonnet, Baeza-Yates and Snider [340] and improved by Crauser and Ferragina [227]. The basic idea is to build the suffix array in memory-sized pieces and merge them incrementally. More precisely, the algorithm divides the text $\mathcal{T}$ into $h = \Theta(N/M)$ pieces of size $\ell = \Theta(M)$, i.e., $\mathcal{T} = \mathcal{T}_h\mathcal{T}_{h-1}\ldots\mathcal{T}_2\mathcal{T}_1$ (note the order). The suffix array is built incrementally in h stages. In stage k, the algorithm constructs $SA_{\mathcal{T}_k}$ internally, and merges it with $SA_{\mathcal{T}_{k-1}\ldots\mathcal{T}_1}$ externally.

The algorithm is best described as constructing the *inverse* SA^{-1} of the suffix array SA (see Figs. 7.2 and 7.8). While $SA[i]$ is the starting position of the ith suffix in the lexicographic order, $SA^{-1}[j]$ is the lexicographic rank of the suffix starting at j, i.e., $SA^{-1}[SA[i]] = i$. Obviously, SA can be computed from SA^{-1} by permutation in $\mathcal{O}(\text{sort}(N))$ I/Os. Note that $SA^{-1}[j]$ is a lexicographic name of the suffix j in the set of suffixes.

A stage k of the algorithm consists of three steps:

1. build $SA_{\mathcal{T}_k}$
2. update $SA^{-1}_{\mathcal{T}_{k-1}\ldots\mathcal{T}_1}$ into the $k-1$ last pieces of $SA^{-1}_{\mathcal{T}_k\ldots\mathcal{T}_1}$
3. transform $SA_{\mathcal{T}_k}$ into the first piece of $SA^{-1}_{\mathcal{T}_k\ldots\mathcal{T}_1}$

Let us call the suffixes starting in $\mathcal{T}_k$ the new suffixes and the suffixes starting in $\mathcal{T}_{k-1}\ldots\mathcal{T}_1$ the old suffixes. During the stage, a suffix starting at i is represented by the pair $\langle\mathcal{T}[i\ldots i+\ell-1], SA^{-1}[i+\ell]\rangle$.[3] Since $SA^{-1}[i+\ell]$ is a lexicographic name, this information is enough to determine the order of

[3] The text is logically appended with ℓ copies of the character \$ to make the pair well-defined for all suffixes.

suffixes. The first step loads into internal memory $\mathcal{T}_k$, $\mathcal{T}_{k-1}$, and the first ℓ entries of $SA^{-1}_{\mathcal{T}_{k-1}\ldots\mathcal{T}_1}$, i.e., the part corresponding to $\mathcal{T}_{k-1}$. Using this information, the representative pairs are formed for all new suffixes and the suffix array $SA_{\mathcal{T}_k}$ of the new suffixes is built, all in internal memory.

The second step is performed by scanning $\mathcal{T}_{k-1}\ldots\mathcal{T}_1$ and $SA^{-1}_{\mathcal{T}_{k-1}\ldots\mathcal{T}_1}$ simultaneously. When processing a suffix starting at i, $SA^{-1}[i]$, $SA^{-1}[i+\ell]$, and $\mathcal{T}[i, i+\ell-1]$ are in internal memory. The latter two are needed for the representative pair of the suffix and the first is modified. For each i, the algorithm determines using $SA_{\mathcal{T}_k}$ how many of the new suffixes are lexicographically smaller than the suffix starting at i, and $SA^{-1}[i]$ is increased by that amount. During the scan, the algorithm also keeps an array C of counters in memory. The value $C[j]$ is incremented during the scan when an old suffix is found to be between the new suffixes starting at $SA_{\mathcal{T}_k}[j-1]$ and $SA_{\mathcal{T}_k}[j]$. After the scan, the ranks of the new suffixes are easy to compute from the counter array C and $SA_{\mathcal{T}_k}$ allowing the execution of the third step.

The algorithm performs $\mathcal{O}(N/M)$ stages, each requiring a scan through an array of size $\mathcal{O}(N)$. Thus, the I/O complexity is $\mathcal{O}((N/M)\,\mathrm{scan}(N))$. The CPU complexity deserves a closer analysis, since, according to the experiments in [227], it can be the performance bottleneck. In each stage, the algorithm needs to construct the suffix array of the new suffixes and perform $\mathcal{O}(N)$ queries. Using the techniques by Manber and Myers [528], the construction requires $\mathcal{O}(M\log_2 M)$ time and the queries $\mathcal{O}(NM)$ time. In practice, the query time is $\mathcal{O}(N\log_2 M)$ with a constant depending on the type of the text. Thus, the total CPU time is $\mathcal{O}(N^2)$ in the worst case and $\mathcal{O}((N^2/M)\log_2 M)$ in practice.

Despite the quadratic dependence on the length of the text, the algorithm is fast in practice up to moderate sized texts, i.e., for texts with small ratio N/M [227]. For larger texts, the doubling algorithm described next is preferable.

7.5.3 Doubling Algorithm

The *doubling algorithm* is based on the lexicographic naming and doubling technique of Karp, Miller and Rosenberg [450]. It was introduced for external sorting of strings by Arge et al. [61] (see Section 7.5.5) and modified for suffix array construction by Crauser and Ferragina [227]. We present an improved version of the algorithm.

Let r_k be the lexicographic naming of the set of substrings of length 2^k in the text appended with 2^k-1 copies of the character \$ (see Fig. 7.8). In other words, $r_k(i)$ is one plus the number of substrings of length 2^k that are strictly smaller than the substring starting at i. The doubling algorithm constructs r_k for $k=1,2,\ldots,\lceil\log_2 N\rceil$. As Fig. 7.8 illustrates, $r_{\lceil\log_2 N\rceil}$ is the same as the inverse suffix array SA^{-1}.

Let us see what happens in a stage k that constructs r_k. In the beginning, each position i is represented by the triple $\langle r_{k-1}(i), r_{k-1}(i+2^{k-1}), i\rangle$, and

r_0:		r_1:		r_2:		r_3:		SA^{-1}:	
4	b	4	ba	4	bana	4	banana$$	4	banana
1	a	2	an	3	anan	3	anana$$$	3	anana
5	n	5	na	6	nana	6	nana$$$$	6	nana
1	a	2	an	2	ana$	2	ana$$$$$	2	ana
5	n	5	na	5	na$$	5	na$$$$$$	5	na
1	a	1	a$	1	a$$$	1	a$$$$$$$	1	a

Fig. 7.8. The doubling algorithm for the text `banana`

the triples are stored in the order of the last component. The following steps are then performed:

1. sort the triples by the first two components (which is equivalent to sorting substrings of length 2^k)
2. scan to compute r_k and update the triples to $\langle r_k(i), r_{k-1}(i + 2^{k-1}), i\rangle$
3. sort the triples by the last component
4. scan to update the triples to $\langle r_k(i), r_k(i + 2^k), i\rangle$

The algorithm does $\mathcal{O}(\text{sort}(N))$ I/Os in each stage, and thus requires a total of $\mathcal{O}(\text{sort}(N) \log_2 N)$ I/Os for constructing the suffix array.

The algorithm can be improved using the observation that, if a name $r_k(i)$ is unique, then $r_h(i) = r_k(i)$ for all $h > k$. We call a triple with a unique first component *finished*. Crauser and Ferragina [227] show how step 2 can be performed without using finished triples allowing the exclusion of finished triples from the sorting steps. This reduces the I/O complexity of stage k to $\mathcal{O}(\text{sort}(N_{k-1}) + \text{scan}(N))$, where N_{k-1} is the number of unfinished triples after stage $k - 1$. We show how step 4 can also be done without finished triples, improving the I/O complexity further to $\mathcal{O}(\text{sort}(N_{k-1}))$.

With only the unfinished triples available in step 2, the new rank of a triple can no more be computed as its rank in the sorted list. Instead, the new rank of a triple $\langle x, y, i\rangle$ is $x+c$, where c is the number of triples in the list with the first component x and the second component smaller than y. This works correctly because $x = r_{k-1}(i)$ already counts the smaller substrings that differ in the first 2^{k-1} characters, and all the triples that have the same first component are unfinished and thus on the list.

The newly finished triples are identified and marked in step 2 but not removed until in step 4, which we describe next. When the scan in step 4 processes $\langle x, y, i\rangle$, the triples $\langle x', y', i'\rangle$, $i' = i + 2^{k-1}$, and $\langle x'', y'', i''\rangle$, $i'' = i + 2^k$ are also brought into memory if they were unfinished after stage $k-1$. The following three cases are possible:

1. If $\langle x'', y'', i''\rangle$ exists (is unfinished), the new triple is $\langle x, x'', i\rangle$.
2. If $\langle x', y', i'\rangle$ exists but $\langle x'', y'', i''\rangle$ does not, $\langle x'', y'', i''\rangle$ was already finished before stage k, and thus y' is its final rank. Then, $\langle x, y', i\rangle$ is the new triple.
3. If $\langle x', y', i'\rangle$ does not exist, it was already finished before stage k. Then, the triple $\langle x, y, i\rangle$ must now be finished and is removed.

The finished triples are collected in a separate file and used for constructing the suffix array in the end.

Let us analyze the algorithm. Let N_k be the number of non-unique text substrings of length 2^k (with each occurrence counted separately), and let s be the largest integer such that $N_s > 0$. The algorithm needs $\mathcal{O}(\mathrm{sort}(N_{k-1}))$ I/Os in stage k for $k = 1, \ldots, s+1$. Including the initial stage, this gives the I/O complexity $\mathcal{O}(\mathrm{sort}(N) + \sum_{k=0}^{s} \mathrm{sort}(N_k))$. In the worst case, such as the text $\mathcal{T} = \mathtt{aaa...aa}$, the I/O complexity is still $\mathcal{O}(\mathrm{sort}(N)\log_2(N))$. In practice, the number of unfinished suffixes starts to decrease significantly much before stage $\log_2(N)$.

7.5.4 I/O-Optimal Construction

An algorithm for constructing the Pat tree using optimal $\mathcal{O}(\mathrm{sort}(N))$ I/Os has been described by Farach-Colton et al. [290]. As explained in Section 7.5.1, the bound extends to the other full-text indexes. The outline of the algorithm is as follows:

1. Given the string $\mathcal{T}$ construct a string $\mathcal{T}'$ of half the length by replacing pairs of characters with lexicographic names.
2. Recursively compute the Pat tree of $\mathcal{T}'$ and derive the arrays $SA_{\mathcal{T}'}$ and $LCP_{\mathcal{T}'}$ from it.
3. Let SA_o and LCP_o be the string and LCP arrays for the suffixes of $\mathcal{T}$ that start at odd positions. Compute SA_o and LCP_o from $SA_{\mathcal{T}'}$ and $LCP_{\mathcal{T}'}$.
4. Let SA_e and LCP_e be the string and LCP arrays for the suffixes of $\mathcal{T}$ that start at even positions. Compute SA_e and LCP_e from SA_o and LCP_o.
5. Construct the Patricia tries PT_o and PT_e of odd and even suffixes from the suffix and LCP arrays.
6. Merge PT_o and PT_e into $PT_{\mathcal{T}}$.

Below, we sketch how all the above steps except the recursive call can be done in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Since the recursive call involves a string of length $N/2$, the total number of I/Os is $\mathcal{O}(\mathrm{sort}(N))$.

The naming in the first step is done by sorting the pairs of characters and using the rank as the name. The transformations in the second and fifth step were described in Section 7.5.1. The odd suffix array SA_o is computed by $SA_o[i] = 2 \cdot SA_{\mathcal{T}'}[i] - 1$. The value $LCP_o[i]$ is first set to $2 \cdot LCP_{\mathcal{T}'}[i]$, and then increased by one if $\mathcal{T}[SA_o[i] + LCP_o[i]] = \mathcal{T}[SA_o[i-1] + LCP_o[i]]$. This last step can be done by batched lookups using $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

Each even suffix is a single character followed by an odd suffix. Let SA_o^{-1} be the inverse of SA_o, i.e., a lexicographical naming of the odd suffixes. Then, SA_e can be constructed by sorting pairs of the form $\langle \mathcal{T}[2i], SA_o^{-1}[2i+1] \rangle$. The LCP of two adjacent even suffixes is zero if the first character does not match, and one plus the LCP of the corresponding odd suffixes otherwise. However, the corresponding odd suffixes may not be adjacent in SA_o. Therefore, to compute LCP_e we need to perform LCP queries between $\mathcal{O}(N)$ arbitrary pairs of odd suffixes. By a well-known property of LCP arrays, the length of the longest common prefix of the suffixes starting at $SA_o[i]$ and $SA_o[j]$ is $\min_{i<k\leq j} LCP_o[k]$. Therefore, we need to answer a batch of $\mathcal{O}(N)$ range minimum queries, which takes $\mathcal{O}(\text{sort}(N))$ I/Os [192].

The remaining and the most complex part of the algorithm is the merging of the trees. We will only outline it here. Our description differs from the one in [290] in some aspects. However, the high level procedure — identify trunk nodes (called odd/even nodes in [290]), overmerge, then unmerge — is the same.

We want to create the Pat tree $PT_\mathcal{T}$ of the whole text $\mathcal{T}$ by merging the Patricia tries PT_o and PT_e. $PT_\mathcal{T}$ inherits a part of the tree from PT_o, a part from PT_e, and a part from both. The part coming from both is a connected component that contains the root. We will call it the *trunk*. Recall that a Patricia trie is a compact representation of a trie. Thus it represents all the nodes of the trie, some of them *explicitly*, most of them *implicitly*. The trunk of $PT_\mathcal{T}$ contains three types of (explicit) nodes:

1. nodes that are explicit in both PT_o and PT_e,
2. nodes that are explicit in one and implicit in the other, and
3. nodes that are implicit in both.

Before starting the merge, the trunk nodes in PT_o and PT_e are marked. This is a complicated procedure and we refer to [290] for details. The procedure may mark some non-trunk nodes, too, but only ones that are descendants of the *leaves* of the trunk. In particular, the nearest explicit descendant of each implicit trunk leaf should be marked.

The merging process performs a coupled-DFS of PT_o and PT_e using the Euler tours of the trees. During the process the type 1 trunk nodes are easy to merge but the other two types cause problems. Let us look at how the merging proceeds. Suppose we have just merged nodes u_o of PT_o and u_e of PT_e into a type 1 node u. Next in the Euler tours are the subtrees rooted at the children of u_o and u_e. The subtrees are ordered by the first character of the edges leading to them, which makes it easy to find the pairs of subtrees that should be merged. If the edges leading to the subtrees have the same full label, the roots of the subtrees are merged and the merging process continues recursively in the subtree. However, the full labels of the edges are not available in Patricia tries, only the first character and the length. The first character is already known to match. With respect to the edge lengths there are two cases:

1. If the lengths are the same, the subtrees are merged. If the full labels are different, the correct procedure would be to merge the edges only partially, creating a type 3 node where the edges are separated. The algorithm, however, does merge the edges fully; this is called *overmerging*. Only afterwards these incorrectly merged edges are identified and unmerged. What happens in the incorrect recursive merging of the subtrees

does not matter much, because the unmerging will throw away the incorrect subtree and replace it with the original subtrees from PT_o and PT_e. For further details on unmerging, we refer to [290].

2. If the lengths are different, the longer edge is split by inserting a new node v' on it. The new node is then merged with end node v'' of the other edge to form a trunk node v of type 2. This could already be overmerging, which would be corrected later as above. In any case, the recursive merging of the subtrees continues, but there is a problem: the initial character of the edge leading to the only child w' of v' (i.e., the lower part of the split edge) is not known. Retrieving the character from the text could require an I/O, which would be too expensive. Instead, the algorithm uses the trunk markings. Suppose the correct procedure would be to merge the edge (v', w') with an edge (v'', w'') at least partially. Then w'' must be a marked node, and furthermore, w'' must be the only marked child of v'', enabling the correct merge to be performed. If (v', w') should not be merged with any child edge of v'', i.e., if v'' does not have a child edge with the first character matching the unknown first character of (v', w'), then any merge the algorithm does is later identified in the unmerging step and correctly unmerged. Then the algorithm still needs to determine the initial character of the edge, which can be done in one batch for all such edges.

Theorem 7.4 (Farach-Colton et al. [290]). *The suffix array, suffix tree, Pat tree, and string B-tree of a text of total length N can be constructed in optimal $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

7.5.5 Sorting Strings

In this section, we consider the problem of sorting strings, which is closely related to the construction of full-text indexes. The results are all by Arge et al. [61].

We begin with a practical algorithm using the doubling technique.[4] The algorithm starts by storing the strings in a single file, padded to make the string lengths powers of two and each string aligned so that its starting position in the file is a multiple of its (padded) length. The algorithm proceeds in $\mathcal{O}(\log_2 N)$ stages similar to the doubling algorithm of Section 7.5.3. In stage k, it names substrings of size 2^k by sorting pairs of names for substrings of half the length. However, in contrast to the algorithm of Section 7.5.3, the substrings in each stage are *non-overlapping*. As a consequence, the number of substrings is halved in each stage, and the whole algorithm runs in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

The rank of a string of (padded) length 2^k is computed in stage k, where it is one of the named substrings. The number of lexicographically smaller

[4] The algorithm was only mentioned, not described in [61]; thus, the details are ours.

strings of the same or greater length is determined by counting the number of lexicographically smaller substrings that are prefixes of a string. The number of lexicographically smaller, shorter strings is determined by maintaining a count of those strings at each substring. The counts are updated at the end of each stage and passed to the next stage. All the counting needs only $\mathcal{O}(\mathrm{scan}(N))$ I/Os over the whole algorithm.

Arge et al. also show the following theoretical upper bound.

Theorem 7.5 (Arge et al. [61]). *The I/O complexity of sorting K strings of total length N with K_1 strings of length less than B of total length N_1, and K_2 strings of length at least B of total length N_2, is*

$$\mathcal{O}\left(\min\left\{K_1 \log_M K_1, \frac{N_1}{B} \log_{M/B} \frac{N_1}{B}\right\} + K_2 \log_M K_2 + \mathrm{scan}(N)\right).$$

As suggested by the equation, the result is achieved by processing short and long strings separately. The algorithm for long strings works by inserting the strings into a *buffered string B-tree*, which is similar to the string B-tree (Section 7.4.3), but all the insertions are done in one batch using buffering techniques from the buffer tree (see Chapter 2).

Arge et al. also show lower bounds for sorting strings in somewhat artificially weakened models. One of their lower bounds nearly matches the upper bound of the above theorem. (The algorithms behind Theorem 7.5 work in this model.) The full characterization of the I/O complexity of sorting strings, however, remains an open problem.

7.5.6 Packed Strings

So far we have used the integer alphabet model, which assumes that each character occupies a full machine word. In practice, alphabets are often small and multiple characters can be packed into one machine word. For example, DNA sequences could be stored using just two bits per character. Some of the algorithms, in particular the doubling algorithms, can take advantage of this. To analyze the effect, we have to modify the model of computation.

The *packed string model* assumes that characters are integers in the range $\{1, \ldots, |\Sigma|\}$, where $|\Sigma| \leq N$. Strings are stored in packed form with each machine word containing $\Theta(\log_{|\Sigma|} N)$ characters. The main parameters of the model are:

N = number of characters in the input strings
$n = \Theta(N/\log_{|\Sigma|} N)$ = size of input in units of machine words
M = size of internal memory in units of machine words
B = size of disk blocks in units of machine words

Note that, while the size of the text is $\Theta(n)$, the size of a full-text index is $\Theta(N)$ machine words.

Under the packed string model, the results presented in this chapter remain mostly unaffected, since the algorithms still have to deal with $\Theta(N)$ word-sized entities, such as pointers and ranks. There are some changes, though. The worst case CPU complexity of the merging algorithm is reduced by a factor $\Theta(\log_{|\Sigma|} N)$ due to the reduction in the complexity of comparing strings.

The doubling algorithm for both index construction and sorting can be modified to name substrings of length $\Theta(\log_{|\Sigma|} N)$ in the initial stage. The I/O complexity of the index construction algorithm then becomes $\mathcal{O}(\mathrm{sort}(N) + \sum_{k=0}^{s} \mathrm{sort}(n_k))$, where n_k is the number of non-unique text substrings of length $2^k \log_{|\Sigma|} N$. On a random text with independent, uniform distribution of characters, this is $\mathcal{O}(\mathrm{sort}(N))$ with high probability [711]. The I/O complexity of the sorting algorithm becomes $\mathcal{O}(\mathrm{sort}(n + K))$.

7.6 Concluding Remarks

We have considered only the simple string matching queries. Performing more complex forms of queries, in particular *approximate string matching* [574], in external memory is an important open problem. A common approach is to resort to sequential searching either on the whole text (e.g, the most widely used genomic sequence search engine BLAST [37]) or on the word list of an inverted file [51, 84, 529]. Recently, Chávez and Navarro [179] turned approximate string matching into nearest neighbor searching in metric space, and suggested using existing external memory data structures for the latter problem (see Chapter 6).