University of Padova - DEI
Bioinformatics course, academic year 2001-2002

Term paper
Hierarchy-conscious Data Structures for String Analysis

PhD Student: Carlo Fantozzi, XVI ciclo

1 Introduction

This paper describes some aspects of the analysis of strings defined over an alphabet Σ; the strings may be related to a biological phenomenon (DNA base pairs, ammino acid sequences...), but this is not strictly necessary since the same analysis techniques also apply to different types of information such as natural linguistic texts, dictionaries or transaction records. The term "string analysis" is quite generic and it encompasses a broad family of operations on strings; some of them are:

- gathering of statistics (number of occurrences of each symbol, average word length, etc.) on a string;
- identification of *patterns* (repeated words or symbols, frequent words, etc.) inside a string;
- search for the presence of known words or substrings inside a string;
- evaluation of a suitably defined "distance" between pairs of strings.

Such operations, and even the plain storage of the strings, rely on suitable *data structures*. In this survey paper we will focus on some data structures that have been developed to efficiently deal with *memory hierarchies*. The importance of these structures can be appreciated by considering that all modern computers exhibit a memory hierarchy, i.e. an interconnection of memory modules with different sizes and speeds, and that the explicit management of the hierarchy typically leads to noticeable performance improvements.

The rest of the paper is organized as follows. Section 2 explains what a memory hierarchy is, then it introduces some theoretical models of computation that describe memory hierarchies. Section 3 is a survey of the most prominent data structures that literature offers to cope with memory hierarchies. The theoretical models of Section 2 are used to evaluate which string operations can be efficiently performed with such structures, and also how much time and memory space does it take to build them.

2 Memory Hierarchies and Cost Models

Memory hierarchies arise from a technological limitation. Nowadays, the technology of processors is advancing more quickly than the technology of memories, and this statement has been true since many years from now. As a consequence, there is a widening gap between the speed of processors and the speed of memory: the memory is not able to feed the processor enough data to keep its functional units busy. This is an unacceptable bottleneck, so the memory architecture has been shaped trying to avoid it in as many practical cases as possible. Since it is easier to build a fast memory device if it contains few cells, memory has become a cascade connection of modules with increasing size but decreasing speed: relevant data are kept in the faster modules as long as possible, so as to minimize accesses to the slower cells. This cascade connection is called a *memory hierarchy*. A memory hierarchy works well in practice because memory accesses always exhibit a certain degree of *locality of reference*: informally speaking, during a limited time

frame accesses point to a small pool of consecutive memory cells, which can then be moved to the faster levels of the hierarchy. Up to now such data movements have been performed automatically by the hardware or the software, not by the programmer: anyway, it is increasingly difficult to maintain this automatism without hampering performance. The hierarchy is becoming steeper and more complex: a typical hierarchy now includes CPU registers, L1 cache, L2 cache, main memory, disks, tapes, and even the network can be regarded as a level of the hierarchy; the access times for L1 and disk differ by approximately 6 orders of magnitude. This complexity is hardly managed through automatic mechanisms: exposing even part of it to the programmer usually leads to significant performance improvements. Theoretical models of computation have been designed that take memory hierarchy into consideration; each of them focuses on some different aspects of the hierarchy, and no universally accepted model has emerged yet. In what follows we survey the most popular models by grouping them into two broad families: Section 2.1 is reserved to models for external memory (i.e. disks), and Section 2.2 deals with models for internal memory (i.e. cache RAM and DRAM).

2.1 External memory

The issue of external memory has been widely studied because it arose early in the database community, which has long faced the need to deal with massive amounts of data. In recent years, "external memory" has almost been synonymous with "hard disk", since this is the most effective medium offered by current technology to store data that do not fit in main memory. Accurate models of disks have been proposed in the literature (see for example [27, 28]), but their complexity make them impractical for the theoretical analysis of algorithms. As a consequence, simpler models have been developed.

The most successful of these models is certainly the I/O model [4, 29], which exhibits one RAM-like processing unit connected with a finite-size random access memory. If the processor needs data that are not in main memory, it must load them from one or more disks; data are fetched in *blocks*, which typically span many memory cells. The metric to measure the performance of an algorithm is the number of block transfers it requires; this is the reason that gives the model its name. This cost model makes sense since an access to disk is several orders of magnitude slower than an access to RAM (nanoseconds versus milliseconds), therefore RAM computations can be taken to be "free". If multiple disks are available then they can be used in parallel, i.e. the processor can access a different block from each disk in a single unit of time. As a whole, the I/O model is described by the following parameters:

- the size M of the internal (random access) memory;
- the size B of a block;
- the number D of available disks.

In the rest of the paper we will stick to the above notation unless otherwise noticed. Vitter and shriver also proposed [30] a parallel version of the model (often referred to as "the" parallel disk model), but we will not deal with it in this paper.

2.2 Internal Memory

From the very beginning control over accesses to external memory has been given to the programmer, while things have been quite different in the realm internal memory. Up to recent years the presence of multiple levels of cache was considered something the programmer need not (or should not) be in control of; as a consequence, it was not necessary to deal with cache parameters in theoretical models. This approach has now shown its weaknesses, and a wealth of models for hierarchical memory has appeared. The most popular ones are:

- the *Hierarchical Memory Model* (HMM) of Aggarwal et al. [2]; access to memory cell i takes time $f(i)$, where $f(\cdot)$ is a suitable nondecreasing function. A variant of this model exists which supports block transfer [3];
- the *Parallel Memory Hierarchy* (PMH) model of Alpern et al. [6];
- the *Uniform Memory Hierarchy* (UMH) model. The first author of the paper is, once again, Bowen Alpern [5].

Among these models, the most successful one is probably the HMM, which is very simple and therefore easy to use as a performance analysis tool. Aggarwal et al. [2] also introduce a concept of *uniformly optimal* algorithms, i.e. algorithms which exhibits fair performance across a set of memory hierarchies. Anyway, it seems that this concept has not been as successful as the model.

Each of these models pays attention to some cache design parameters which are considered essential for good performance, such as cache size, cache line size, interconnection bus width, etc. The aspects which are considered crucial to performance vary from model to model, therefore an algorithm that is tuned for speed on one of them is difficult to analyze on a different one. This theoretical problem reflects the fact that different cache designs exist in real machines, and that the quest for maximum performance usually forces a programmer to know the specific parameters of the cache hierarchy he/she is dealing with.

A recent work [19] has suggested to deal with portability problems through the development of *cache oblivious* algorithms, i.e. algorithms that perform well on *any* cache architecture. Cache oblivious algorithms are designed on an idealized 2-level hierarchy that closely resembles the I/O model: the fastest level is now called the *cache*, while the slowest one is called the *main memory*. The processor can only work on data that reside in the fastest level, which has finite size M. The main difference with the I/O model is that data movements are automatically performed by a hardware circuit which is "magically" supposed to always make the best possible choice: when a cache miss occurs, in fact, the block of B cache cells that is overwritten is the one whose next access is furthest in the future*. The performance measure of an algorithm is the *cache complexity* $Q(n; M; B)$, i.e. the worst-case number of block transfers the algorithm needs when the input size is n. Note that the cache line size B is *not* known to the algorithm, therefore the algorithm designer must prove a bound that is valid for any value of B. Once achieved, this daunting task goes not without reward: [19], in fact, demonstrates that an algorithm with optimal cache complexity on the ideal model is also optimal on a more feasible model with multiple levels of cache – regardless of the cache line size at each level – and a realistic LRU (Least Recently Used) cache line replacement policy.

3 Data Structures

The core of this survey are B-trees, string B-trees, suffix trees and suffix arrays. Before describing them, anyway, we need to introduce some basic data structures: tries, PATRICIA tries and inverted files.

3.1 Tries

A *trie* [14, 18] T_S is a tree that is used to store a set S of strings over an alphabet Σ; "trie" is a substring of the word "re*trie*val". Each edge of the tree is marked with a single symbol of Σ, and there is a one-to-one correspondence between the strings in S and the paths from the root of T_S to a leaf. If two strings $x, y \in S$ are such that $x = zv$ and $y = zw$ (i.e. x and y have a common prefix), then on the tree they share the part of the path corresponding to substring z. A string can be extracted from the tree by following the corresponding path from the root to a leaf; all strings in S can be recovered through a depth-first visit of

*This behavior plainly implies a fully associative cache.

the tree. If T_S resides in a flat memory, the question of whether x belongs to S can be answered in $\Theta(|x|)$ time. Note that the maximum outdegree of a node in T_S is $|\Sigma|$, which can be very high (or even unbounded if $|\Sigma|$ is infinite); to make things worse, if S is small then the outdegree of many nodes will be much lower than the maximum, or even one. These properties make it difficult to efficiently pack the tree into memory. If $|\Sigma|$ is small then linked lists are probably the best choice: if the memory is flat, `insert` and `delete` operations of a string x both take time $O(|\Sigma| \cdot |x|)$. Anyway, space efficiency has an impact on performance if the memory is hierarchical, since elements which are neighbors in a list may be far spaced in memory. In the worst case each pointer jumping towards the successor of a node may incur a cache miss/page fault, thus giving an I/O complexity of $O(|\Sigma| \cdot |x|)$ for `insert`,`delete` and `search` operations. Note that the potential impact on performance is higher if the trie is updated dynamically, since nodes are typically more scattered in memory.

A *compacted trie* is a compressed representation of a trie. A compacted trie can be obtained from the corresponding regular trie T_S by collapsing each sequence of nodes with less than two successors into a single node; the edge entering the new node is labeled with the substring w associated with the path in T_S that has just been collapsed. In a compacted trie every node is branching, therefore the trie can be packed in memory more easily; moreover, there is less potential node scatter since at least substrings like w are now stored in consecutive memory locations. However, update operations now take more time since some nodes may need to be split (because of an `insert` operation) or collapsed (`delete`) to enforce the "compactness" property; what's more, different nodes now have different sizes.

A *PATRICIA trie* [25] is an even more compact representation of a trie, although this time compression involves a loss of information; PATRICIA stands for "Practical Algorithm To Retrieve Information Coded In Alphanumeric". A PATRICIA trie P_S can be obtained by taking the corresponding compacted trie and substituting each substring with a pair (i, a): i is the length of the substring, and a is its first character (also referred to as *branching character*). If Σ is finite-size (as it is in practical applications) then each edge of P_S requires constant space; since each string in S can contribute at most one edge to the trie, this implies that S can be indexed in $O(|S|)$ space by means of a PATRICIA trie. Note, anyway, that many substrings may match the same pair (i, a), therefore the trie only offer partial information about the substrings.

3.2 Inverted files

Another old and simple data structure is the *inverted file* [21]. An inverted file is created by processing a set S of strings to produce an *index file* and a *postings file*. The index file is a lexicographically sorted list of all the substrings of the elements in S that are considered "relevant", which are also named *keywords*. Following any keyword x in the index there is a pointer to a suitable location in the postings file that contains a list of all the occurrences of x in S. The set of keywords is typically a small subset of all the possible substrings, therefore the size of the index file is usually limited and it can fit into main memory. On the contrary, the postings file may be very large and it often resides in secondary storage. If the above hypotheses (index file in main memory, postings file in secondary memory) is true and x is a keyword, then the list of all the k occurrences of x can be retrieved in optimal time $O(\log x + k/B)$. An inverted file is thus more powerful than a trie for keywords, but it does not contain any information for all the substrings that are not keywords. Inverted files are mainly used in natural language processing applications, and more generally in all those situations in which the the set of used words is a small fraction of the set of possible words.

3.3 B-trees

B-trees [8] are one of the canonical data structures for string analysis that were designed with external memory in mind. A B-tree T is a search tree built on a set S of *keys*; the keys are fixed-size, and they may

be tokens for bigger data structures (such as strings over an alphabet Σ) that reside elsewhere. T satisfies the following properties.

1. Every node v contains a certain number $N(v)$ of keys, named $k_1(v), k_2(v), \ldots, k_{N(v)}(v)$, that are stored in lexicographic order.
2. Considered as a whole, the leaves store all the keys in S.
3. Every nonleaf node u also contains $N(u) + 1$ pointers $c_1(u), c_2(u), \ldots, c_{N(u)+1}$ to its children.
4. All the leaves have the same depth h, which is also the height of the tree.
5. The keys of node u partition the keys of the subtrees rooted at u into non-overlapping ranges. In other words, if k is a key that is stored into the subtree associated with $c_j(u)$ then $k_{j-1}(u) \leq k \leq k_j(u)$.

A B-tree is *balanced* in the sense that no internal node can have "too many" or "too few" leaves: to be precise, $b - 1 \leq N(u) \leq 2b - 1$ for every nonleaf node $u \in T$. The parameter b can be tuned for performance; a straightforward optimization rule is to put $b = B/2$, where B is the disk block size of the I/O model. In this way, a node of T can fit into a disk block, thus being accessed efficiently. Several other balancing rules have been defined, giving rise to as many B-tree variants. Here we name two of them.

- *Weight-balanced B-trees* [7]: the weight of a node v is required to be $w(v) = \Theta\left(b^{ck}\right)$, where b and c are tuning parameters and k is the height of the subtree rooted at v. The weight of v is defined to be the number of elements in the leaves of the subtree rooted at v. For such B-trees, the trivial optimization rule sets $b = \Theta(B)$ and c such that $w(v) \leq B$ for any node v.
- *Level-balanced B-trees* [1]: the balancing constraint is on the number of nodes on each level of the tree; such a number is a decreasing function of the level. (We do not put a formal definition here.)

Regardless of the variant we are considering, an update operation (insertion or deletion of a key) may cause a violation of the balancing rule, thus forcing a partial rebuild of the tree. For example, the operation `insert(`x`)` is performed by traversing the tree until the leaf v is reached which contains the biggest key $y \leq x$; if v has available space (according to the balancing rule) then x is added to it, otherwise a new leaf v' is created and the keys are redistributed between v and v' so that both of them contain the minimum prescribed number of keys. The creation of v' forces to add a new key/pointer pair in the predecessor of v, which in turn may result in another violation of the balancing rule; the restructuring thus propagates towards the root of the tree, and may reach the root in the worst case. Anyway, each level of the tree requires only a constant number of potential I/O operations, therefore `insert(`x`)` takes $O(h)$ I/Os. The analysis for `delete(`x`)` is analogous – although this time merging of nodes is what propagates towards the root – and leads to the same I/O complexity bound. Finally, an update operation like `update(`x`,`y`)` can be regarded as a `delete(`x`)` followed by an `insert(`y`)`.

Consider now the "nice case" in which the strings to be indexed are short and asymptotically have all the same length. In this case the strings can be directly used as keys, and the corresponding B-tree T can be stored using $\Theta(N/B)$ disk blocks, where N is the total length of the strings. Such a bound is clearly optimal. `insert` and `delete` operations both take time $O(\log_B N)$. The tree itself can be built through `insert` operations, but this strategy leads to a suboptimal I/O complexity of $O(N \log_B N)$. The tree construction can be performed more efficiently by first sorting the keys in external memory and then building the tree level after level, from the leaves up; this strategy requires $O\left((N/B)\log_{M/B}(N/B)\right)$ I/O operations, which turns out to be optimal because of the sorting bound of Aggarwal and Vitter [4]. (Remember our observation in Section 2.1, and consider that the keys are stored in sorted order inside the B-tree.) Most of all, T can

now be used to efficiently address a certain number of query types on the strings; three of them are defined below [17].

1. `prefix-search(`K`)`: retrieve, in lexicographic order, all the strings that have K as a prefix.
2. `range-query(`K_1`,`K_2`)`: retrieve, in lexicographic order, all the strings which have a prefix between K_1 and K_2.
3. `substring-search(`K`)`: retrieve, in lexicographic order, all the strings that include K as a substring.

Problems 1 and 2 can be solved through a suitable traversal of the B-tree in I/O time $O\left(\log_B N + k/B\right)$, where k is the number of strings in S matching the query. `range-query(`K_1`,`K_2`)` implies two traversals of the tree (from the root to a leaf) to find the leaves L_1, L_2 associated with K_1 and K_2, then the answer to the query is retrieved by reading the elements in the leaves between L_1 and L_2. The I/O time mentioned above can be proved to be optimal in the comparison model. On the contrary, a plain B-tree is not suited to solve Problem 3. Note that this very problem takes I/O time $O\left(h/\sqrt{|K|} + \log_{|K|} N\right)$ if the index is a PATRICIA trie built on the set of all the suffixes of the strings in S [12]; h is the height of the trie.

Things are quite different if the keys have different lengths, and above all if they are "long", or have unbounded length. In this case, it is extremely inadvisable to use the strings of S as keys to traverse the tree: variations in length lead to a different number of keys inside each disk block, which clashes with the balanced structure of the tree. The problem of building an external memory index structure for a set of arbitrary strings has been addressed by Ferragina and Grossi [17] with the *string B-tree*.

As its name suggests, the string B-tree is a B-tree variant that embodies sophisticated data structures for the management of variable-sized strings. First of all, the key for an arbitrary string in S is now a *pointer* to the disk block containing the first symbol of the string; for all practical purposes, the keys can be therefore considered of equal length. The strings of S are packed one after the other into consecutive disk blocks; different strings are separated by a special character not belonging to Σ, so that the beginning of each string can be correctly identified. The strings are stored in no particular order (i.e. they are not sorted) but each string x occupies consecutive memory blocks, so that the position of th i-th character of x can be determined through a constant number of arithmetic operations. Observe that adding a new string y to S is just a matter of attaching it at the tail of the sequence of previously-stored strings. It is essential to note that the new keys are small, but they do not give any hint about the lexicographic order of the strings: in other words, string x may be lexicographically bigger than string y although the key for x is smaller than the one for y. This observation shows that some information other than the keys must be stored in the nodes of the tree. To begin with, the tree is trivially modified as follows.

1. Each leaf node is augmented with pointers to its predecessor and successor leaves, so that the leaves form a bidirectional list.
2. The keys in each node are sorted according to the lexicographic order of the strings they refer to.
3. An internal node stores both the smallest (leftmost) and the biggest (rightmost) key of all its descendants.

Secondly, the keys in a nonleaf node are not stored in a plain array: instead, they are stored by means of a PATRICIA trie (see Section 3.1) built on the strings associated with the node. In a way, the properties of the trie make it possible to pack $\Theta(B)$ *arbitrary-length* strings into a B-sized disk block, like the plain B-tree did for fixed-length strings. The trie, however, contains only partial information about the strings, namely the information associated with the branching characters. Consider, for example, an instance of

`prefix-search(`K`)` with $K =$ `abracadabra`: among other activities, the string B-tree must be traversed from the root to a leaf to determine the first string in S whose prefix is `abracadabra`. Consider the root node, and suppose an edge departing from the root of the associated trie is labelled with $(\texttt{a}, 4)$: the searching algorithm then follows this edge, reaching an internal node v of the trie. Suppose now that v has only two outgoing edges, marked with $(\texttt{a}, 3)$ and $(\texttt{b}, 4)$: since the fifth character of our prefix is a `c`, the trie does not contain enough information to perform any further branch. Anyway, Ferragina and Grossi demonstrate that "things cannot go really bad", in the sense that

- if the downward branching process leads to a leaf l of the PATRICIA trie, then l contains one of the strings (not necessarily the first!) which share the longest common prefix with K;
- if the downward branching process stops on an internal node v, then the property above is still satisfied by choosing as l any leaf in the subtrie rooted at v.

Exploiting this property, the search for a string/prefix x in the PATRICIA trie can be performed as follows (*blind search*).

1. Trace a downward path using the branching characters as long as possible. If the branching ends on a leaf l then the phase is over, otherwise choose l as an arbitrary leaf in the subtrie of the last node v traversed.
2. Load the string associated with l and compare it with x to determine the longest common prefix x' it actually shares with x.
3. Use $|x'|$ to find the shallowest ancestor u of l whose associated strings have a prefix not lexicographically smaller than x' (it is the shallowest ancestor whose label from the root is more than $|x'|$ characters long).
4. Using the first mismatching character after x' as a key, descend in the subtrie rooted at u until a leaf is reached: the string in such a leaf is the desired answer to the search problem.

The entire PATRICIA trie is stored in a single memory block, therefore it can be taken to main memory in a single I/O operation; the comparison with the string associated with leaf l (Step 2) may be more expensive since it requires $\lceil |x|/B \rceil$ disk I/Os. The main drawback of the above procedure is that a distinct trie scanning and string comparison is required *every time* a B-tree node is traversed, which leads to $O\left((|K|/B + 1)h\right)$ I/O operations to visit the B-tree during `prefix-search(`K`)`. Ferragina and Grossi show that it is possible to do better than this by observing that K is re-examined from the beginning at each level of the B-tree, while this is not necessary since each new level "freezes" an increasingly long prefix of K. In other words, every time we move from node u to node v in the tree a certain number of available strings (i.e. those that are stored in the subtree rooted at u but not in the subtree rooted at v) is discarded, and the remaining strings share a longer prefix with K. Leveraging on this observation, [17] devises an improved trie search procedure which takes the length ℓ of the "frozen" prefix as an input, so that only the characters in positions $\ell + 1, \ell + 2, \ldots$ need to be examined during Step 2 of the procedure. The total I/O cost of a string B-tree traversal is therefore reduced to $O\left(|K|/B + |S|\right)$. Using such an efficient traversal strategy, the following I/O bounds can be proved. We recall that k is the number of strings in S that match a query, and N is the total length of the strings in S.

- A string x can be added or deleted from the string B-tree using $O\left(|x|/B + \log_B |S|\right)$ worst-case disk accesses.
- The construction of the string B-tree takes $O\left(N \log N\right)$ worst-case disk accesses.

- `prefix-search(`K`)` takes $O\left((|K| + k)/B + \log_B |S|\right)$ worst-case disk accesses.
- `range-query(`K_1`,`K_2`)` takes $O\left((|K_1| + |K_2| + k)/B + \log_B |S|\right)$ worst-case disk accesses.
- `substring-search(`K`)` takes $O\left((|K| + k)/B + \log_B N\right)$ worst-case disk accesses.

The last operation is performed through a `prefix-search(`K`)` query using a B-tree built on the set $\mathrm{SUF}(S)$ of the suffixes of the strings in S; the equality $|\mathrm{SUF}(S)| = N$ clearly holds.

The efficiency of B-trees as indexing structures has also been investigated in the context of multi-level memory hierarchies. To be precise, the model that has been adopted in such studies provides $k > 2$ levels of memory; a data movement from level i to level $+1$, $1 \leq i < k$, takes place in blocks of size B_i, therefore each pair (L_i, L_{i+1}) of adjacent levels is similar to a standard, 2-level I/O model. A trivial adaptation to such a memory hierarchy [9] is to build a B-tree in which the nodes stored in level L_1 have size $\Theta(B_1)$, the nodes residing in L_2 have size $\Theta(B_2)$ and so on. Anyway, the above strategy is asymptotically efficient only if $B_1 \geq B_2 \geq \ldots \geq B_{k-1}$, and even in this case the multilevel structure of the tree complicates the handling of `insert` and `delete` operations. What's more, the tree must be rebuilt if any level of the hierarchy changes. Anyway, it has been proved in the literature that a more sophisticated adaptation process leads to *cache-oblivious B-trees*, i.e. trees which guarantee good performances regardless of the parameters of the memory hierarchy. Indeed, on the the ideal cache model of Frigo et al. (see Section 2.2) cache-oblivious B-trees exhibit the same number of memory transfers of a plain B-tree in the I/O model when considering common operations such as `insert(`x`)`, `delete(`x`)` or `prefix-search(`K`)`.

Cache-oblivious B-trees were introduced by Bender et al. [9]. To achieve obliviousness, the authors must store the nodes of the B-tree in a particular order, ensuring that any possible subtree is located in a contiguous block of memory. Moreover, data inside each node must be packed densely enough to guarantee that the cache is exploited efficiently, although not too densely since this makes update operation extremely expensive. As a matter of fact, the cost of update operations appears as the sheer difficulty of the whole procedure, therefore Bender et al. take the necessary precautions to maintain it within reasonable bounds. First of all, the balancing rule of the standard B-tree is too "weak": update operations may violate it too frequently. As a consequence, the authors use a weight-balanced B-tree [7] described by the following rule: the weight $w(v)$ of a nonroot node v at height k satisfy the inequalities

$$\frac{d^{k-1}}{2} \leq w(v) \leq 2d^{k-1},$$

with $w(v) - d^{h-1}/2 = \Theta\left(d^k\right)$ and $2d^{k-1} - w(v) = \Theta\left(d^k\right)$. The parameter d (referred to as the *branching parameter*) must be an integer greater than 4 and can be chosen by the programmer. The above rule guarantees that if a nonroot node v at height k has jus been rebalanced, then the B-tree must incur $\Omega\left(d^k\right)$ update operations before v needs to be rebalanced again. In turn, this ensures a low amortized cost for update operations. The rule also imposes a constraint on the number of descendants each node can have: to be precise, the root has between 2 and $4d$ children and all the other nodes have between $d/4$ and $4d$ children. For efficiency reasons, every leaf of the tree is also augmented with a pointer to its right sibling.

Secondly, the weight-balanced B-tree is placed into memory according to the following rule (*van Emde Boas layout* [26]). Let h be the height of the tree and let $\bar{h}$ be the smallest power of two greater than $h/2$:

1. recursively place into memory the topmost $h - \bar{h}$ levels of the tree (these levels are also given the name of *top subtree*);
2. when the above placement is complete, lay recursively in memory the subtrees corresponding to the remaining $\bar{h}$ levels of the tree (the *bottom subtrees*), in "left to right order" (i.e. the subtrees whose leaves are lexicographically smaller come first).

It can be proved that, with this placement rule, a search in the weight-balanced B-tree[†] requires at most $O(\log_B N)$ memory transfers regardless of the cache block size B.

Finally, we have already observed that the exact placement of the data into memory must reserve some (but not too much) empty space to deal with future `insert` operations. Moving from the ideas of [22], data are stored so as to satisfy the following two guidelines.

- Any set of k contiguous data is stored in $O(k)$ contiguous memory cells, therefore the data can be read using $O(k/B)$ memory transfers.
- Inserting or deleting a new datum uses $O\left(\log^2 N/B\right)$ amortized memory transfers.

The cost of insertions (and deletions) is due to the fact that, if the data become too densely (too loosely) packed, then the data are moved and spread out evenly in a bigger (smaller) memory segment.

Let now examine the performance of the cache-oblivious B-tree as an indexing structure. Since k contiguous keys occupy $O(k)$ memory cells and the search for a key in the tree takes $O(\log_B N)$ memory transfers, it is straightforward to conclude that an instance of `prefix-search(`K`)` can be completed with $O(\log_B N + k/B)$ memory transfers in the worst case. The analysis for `insert` and `delete` operations is much more tricky since many different costs must be taken into account. For example, the worst-case cost of executing `insert(`x`)` has four components. (The components for `delete(`x`)` are analogous so we will only focus on insertions in this survey paper.)

1. The cost of searching for x in the cache-oblivious B-tree.
2. The cost of re-packing the data in the leaf node where x is stored.
3. The cost of updating the pointers to any nodes moved as a consequence of the re-packing.
4. The cost of the node-restructuring activities that are needed to enforce the B-tree balancing property.

Costs 1 and 2 can be easily found to be $O(\log_B N)$ and $O\left(\log^2 N/B\right)$, respectively. To determine cost 3, [9] is forced to distinguish between *local* and *long-distance* nodes. Intuitively, a node v is local if its parent and children are within distance B in memory (thus probably residing in the same cache block of v), otherwise it is long-distance. Clearly, the pointer updating process is much more expensive for long distance nodes since many memory blocks must be accessed to take its "family" into cache. The memory layout must then ensure that long-distance nodes are sufficiently far from one another, so that each node movement involves only a small fraction of them: to do so, a certain amount of *buffer* (dummy) *nodes* is stuffed around long-distance nodes. The problem is complicated by the fact that the cache block size B is unknown, so it is impossible to pick out exactly all the long distance nodes: as a consequence, each node that simply *may* be long-distance must be treated as such. Bender et al. demonstrate that the root of each top/bottom subtree and the leaves of any bottom subtree may all be long-distance; as a whole, a fraction $O\left(1/\sqrt{B}\right)$ of the nodes of the B-tree may be long-distance. To separate them, d^{k-1}/k buffer nodes are added before and after each top subtree A, where k is the height of the root of the subtree. It can be proved that, after the insertion of dummy nodes,

- the overall memory space occupied by the B-tree is still $\Theta(N)$;
- any interval I of memory contains $O\left(1 + \frac{\log B}{\sqrt{B}}|I|\right)$ long-distance nodes.

[†] Indeed, the property is valid for all the trees in which the outdegree of the nodes is $O(1)$ and greater than 1.

These properties allow [9] to conclude that the cost of updating the pointers is $O\left(1 + \frac{\log B}{\sqrt{B}} \log^2 N\right)$ memory accesses.

The last cost we must evaluate is Cost 4, due to the restructuring of the B-tree to enforce the balancing property. For an `insert(`x`)` operation, the restructuring is managed as follows.

1. Ascending the tree from the leaf where x has been inserted, find the first ancestor u that violates the balancing property.

2. Let $v_1, \ldots, v_k$ be the children of u. Find the biggest integer k' such that the total weight of $v_1, \ldots, v_{k'}$ is at most $\lceil w(v)/2 \rceil$.

3. Split u into two nodes u_1 and u_2. Update pointers so that $v1, \ldots, v_{k'}$ are children of u_1 and $v_{k'+1}$, …, v_k are children of u_2.

(A delete operation is dealt with in a similar way. The first ancestor u whose weight is too low is found and merged with one of its sibling nodes: the weight of the new node may now be too high, in which case it is split as described above.) After the tree has been restructured, some more memory operations are required to repair the van Emde Boas layout and to put a sufficient number of buffer nodes near every potential long-distance node; we omit the details here. As a whole, both `insert` and `delete` operations require $O(\log N/B)$ amortized memory transfers. By summing costs 1 to 4, we conclude that updating the B-tree takes

$$O\left(1 + \log_B N + \frac{\log B}{\sqrt{B}} \log^2 N\right)$$

amortized memory transfers. This bound simplifies to $O(\log_B N)$ if $B = \Omega\left((\log N)^2 (\log \log N)^4\right)$, which is a mild assumption. Indeed the hypothesis on B can be even milder if the B-tree is stored in two separate arrays: the *leaf array* contains the keys, logically grouped into blocks of size $\Theta(\log N)$, and the *tree array* stores the rest of the tree. The splitting is based on the idea that most memory update operations are concentrated near the leaves of the B-tree because of the balancing property, therefore it is advisable to pack the leaves together in a separate array (note that in the plain van Emde Boas layout the leaves do not occupy consecutive memory positions). After the splitting, only a fraction $O(1/\log N)$ of the update operations touches the tree array, and the amortized cost of an update is reduced to $O\left(1 + \log_B N + \log^2 N/B\right)$.

A recent paper of Brodal et al. [10] obtains the same memory access complexity of [9] through a different data structure, which is plainly called the *cache-oblivious search tree*. This data structure makes no use of B-trees; instead, it is obtained by performing the following operations.

1. Store the keys of S in a dynamic, binary search tree T. The tree is not complete but it is balanced, so that its height is $\log |S| + O(1)$.

2. Embed the dynamic tree into a static, complete binary tree T' having the same height.

3. Store the static tree T' into memory using the van Emde Boas layout.

The embedding of a static tree into the van Emde Boas layout, which is implicit, makes it possible to calculate the position of every node of the tree with simple arithmetics operations[‡]: in other words, no pointers are necessary to link the nodes of the tree, and consequently no pointers need to be updated during `insert` or `delete` operations. Moreover, the fact that T is balanced makes it possible to store T' using $O(N)$ memory cells, which is optimal. Using these properties, it is easy to prove that the cache-oblivious search tree supports `prefix-search(`K`)` operations in $O(\log_B N + k/B)$ worst-case memory transfers.

[‡]To be true, [10] uses a pre-computed vector of size $O(\log |S|)$ to speed up node jumping.

As it is customary, anyway, the analysis of update operations is much more delicate, since it must account for the cost of enforcing the properties of the data structure. The first property to enforce here is the balancing of the dynamic tree T. To understand how this can be done, define the *density* of a node $v \in T$ as $\rho(v) = w(v)/2^{H-d(v)+1}$, where $w(v)$ is the weight of v in T, $d(v)$ is the depth of v (i.e. the number of nodes on the simple path from the root to v) and H is the height of the complete binary tree T' that contains T. In other words, $\rho(v)$ is the ratio between the weight of v in the dynamic tree and the "ideal" weight it should have were the tree complete. Let us now consider an `insert(`x`)` operation: H *density thresholds* $\tau_1, \ldots, \tau_H$ are defined according to the following rules:

1. $0 < \tau_1 < \tau_2 < \ldots < \tau_H = 1$;
2. $\tau_1 \geq |S|/(2^H - 1)$;
3. $\tau_i = \tau_1 + (i-1)(1-\tau_1)(H-1)$ for $1 < i < H$.

If the insertion of x makes the height of T increase to $H+1$ (thus requiring a bigger complete tree for the embedding) then T is rebalanced. Let v be the node associated with the newly-inserted key x: moving up the tree, the nearest ancestor u of v is found satisfying the inequality $\rho(u) \leq \tau_{d(w)}$. The j keys in the subtree rooted at u are then sorted, then the $\lceil j/2 \rceil$-th element in the sorted list is assigned to u, the smallest $\lfloor (j-1)/2 \rfloor$ elements are recursively assigned to the left subtree of u and the remaining elements are recursively stored in the right subtree. `delete` operations are dealt with similarly through a second set $\gamma_1, \ldots, \gamma_H$ of thresholds, with $0 \leq \gamma_H < \gamma_{H-1} < \ldots < \gamma_1 < \tau_1$. It can be proved that insertion and deletions require

$$O\left(\log_B N + \frac{\log^2 N}{\alpha B}\right)$$

amortized memory transfers, with $\alpha = \min\{\gamma_1 - \gamma_H, 1 - \tau_1\}$.

3.4 Suffix trees and suffix arrays

The suffix tree is one of the most common index structures that are used today to deal with *unstructured texts*, i.e. texts which contain no known "words" or keys; genetic data may be considered among these texts. A characteristic of an unstructured text is that *any* possible subsequence of the text is a candidate for a query, so the number of possible queries is quite high (quadratic in the length of the text). A suffix tree efficiently supports such queries, and it can be packed so that it does not take (asymptotically) more space than the text itself.

Let S be the text, i.e. a string[§] built on a specified alphabet Σ that may be finite or infinite, and let N be the length of S. The suffix tree T_S for S is simply a compacted trie built on the set of strings $F = \{x\$ \mid x$ is a suffix of S and $\$ \notin \Sigma\}$; it it clearly $|F| = N$. The special character \$ is added to ensure that each suffix corresponds to a unique position in the text. In what follows, we will denote with $\sigma(v)$ the string associated with a path from v to a leaf in T_S; moreover, we will use the notation $S[i, j]$ to indicate the substring of S composed by characters $S[i]S[i+1]\cdots S[j]$. The suffix tree can be built in less time and stored in less space than an ordinary compacted trie because the elements of F are not arbitrary strings: instead they are all prefixes of a single string. This fact makes it possible to prove some strong properties that can be exploited in the construction of the tree; here we cite only two of them, which are used by the algorithms in the rest of the section.

[§] As usual, multiple strings can be indexed by laying them one after the other, with just one special character in the middle to mark the beginning of each new string.

- *Property 1.* If $\mathrm{LCP}(x, y)$ is the longest common prefix of strings $x, y \in F$ and $\mathrm{LCA}(u, v)$ is the least common ancestor of nodes $u, v \in T_S$, then $\mathrm{LCP}(\sigma(u), \sigma(v)) = |\sigma(\mathrm{LCA}(u, v))|$.
- *Property 2.* Let x be an arbitrary string in F, and let a be an arbitrary character from alphabet Σ. If there is a node u in T_S such that $\sigma(u) = ax$, then there also exist a node $v \in T_S$ such that $\sigma(v) = x$.

The second property suggest to augment the tree with the so-called *suffix links*, i.e. additional pointers from the node associated with ax to the node associated with x. Since each node of T_S has exactly one outgoing suffix link, the introduction of suffix links increases the memory space requirement of T_S only by a constant factor. In a flat-memory model, a suffix tree augmented with suffix links can be constructed in $\Theta(N)$ time if Σ has finite cardinality, or $\Theta(N \log N)$ time if Σ is unbounded (several algorithms exist; the classical reference is [24]). A suffix tree contain a sorted set of strings, therefore the computational complexity of sorting is a lower bound to the complexity of suffix tree construction: as a consequence, any $O(N \log N)$ algorithm is optimal in the comparison model. The same line of reasoning holds if the I/O complexity of sorting is considered. Recently, Farach-Colton et al. have developed an external-memory algorithm to build T_S in sorting I/O complexity [16], which is optimal in the I/O model of computation. The algorithm is based on the following divide-and-conquer strategy.

1. The *odd tree* T_o of T_S is recursively built. The odd tree is the compacted trie built on the suffixes of S that start in odd positions.
2. The *even tree* T_e of T_S is recursively built.
3. T_o and T_e are merged to get T_S.

A deterministic construction of the odd and even trees is detailed in [15]. The idea to build T_o is to sort the $N/2$ pairs of consecutive characters of the form $(S[2i-1], S[2i])$ and eliminate the duplicates in the sorted list. The compacted list is used to build a new string S' over alphabet $\{1, 2, \ldots, N/2\}$: $S'[i]$ is the rank of $(S[2i-1], S[2i])$ in the compacted list. After that, the suffix tree $T_{S'}$ for S' is recursively built. A key property of $T_{S'}$ is that there is an injective map from the internal nodes of $T_{S'}$ at level i and the internal nodes of T_o at level $2i$: exploiting this mapping, T_o can be obtained from $T_{S'}$ in sorting I/O complexity. Once T_o is known, the even tree T_e can be calculated from it: this is possible because each even prefix is just a suitable odd prefix with an extra character attached at its head. The lexicographic order of the leaves of T_e is determined by taking the lexicographically sorted list of the leaves of T_o (which, in turn, is given by an inorder visit of T_o), then sorting it using $S[2i-2]$ as the key for the leaf associated with the odd prefix $S[2i-1, N]$; a stable sorting algorithm must be used. After that, the length of the longest common prefix for each pair of adjacent leaves in T_e is computed by taking advantage of Property 1 of suffix trees (see above). Finally, the even tree T_e is reconstructed from the sorted list of its leaves and the information about longest common prefixes.

Let $T_{\mathrm{suf}}(N)$ be the I/O complexity of building the suffix tree T_S for a string of length N, and let $T_{\mathrm{merge}}(N)$ be the complexity of merging the odd and even trees of T_S. As a whole, $T_{\mathrm{suf}}(N)$ obeys the following recurrence relation:

$$T_{\mathrm{suf}}(N) = T_{\mathrm{suf}}(N/2) + \alpha T_{\mathrm{sort}}(N) + T_{\mathrm{merge}}(N),$$

where α is a suitable constant. All that remains to do is to determine $T_{\mathrm{merge}}(N)$. The merging phase is the most technically complex part of the algorithm: structural properties of the suffix trees are exploited to efficiently perform this step. The merging algorithm is based on the knowledge of *merge nodes* in both the odd and the even tree; v is said to be a merge node if the node it gives birth to in T_S has descendants coming from both T_o and T_e. Once merge nodes are known, T_S can be built by simultaneously visiting the

Euler tours of T_o and T_e and undertaking appropriate measures every time a merge node is encountered; the calculation of the Euler tours is the most expensive operation of this process, and it can be done in sorting I/O complexity [11]. In turn, merge nodes are located with the aid of *anchor nodes* and *side trees*. A node $v_o \in T_o$ is an anchor node if it has a "twin" node $v_e \in T_e$ such that $\sigma(v_o) = \sigma(v_e)$; anchor nodes clearly exist in pairs. A side tree is a maximal subtree of T_o (or of T_e) that contains no anchor nodes. All anchor nodes and their ancestors are merge nodes, but only some of their descendants have the same property: to be precise, the descendants of anchor nodes which are also merge nodes form paths (called *zippers*) inside the side trees. Farach-Colton et al. demonstrate that both anchor nodes and zippers can be located in sorting I/O complexity: as a consequence, we can conclude that $T_{\text{merge}}(N) = O\left(T_{\text{sort}}(N)\right)$ and the whole suffix tree T_S can be built in sorting I/O complexity.

Although the algorithm of [16] is asymptotically optimal, it is unpractical because of the constant hidden by the big-oh notation; what's more, the building of the tree requires a large amount of working space to store auxiliary data structures. As a matter of fact, all known practical algorithm for suffix tree construction still exhibit a $\Theta\left(N^2\right)$ worst-case I/O complexity, and they typically require between $16N$ and $26N$ memory cells to accomplish their task [12, 13]. Since these space/time complexities are often unacceptable in practice, more compact (albeit less powerful) indexing structures have been developed. Among such structures, the most famous one is undoubtedly the *suffix array*.

The suffix array [23] is usually regarded as a "simplified version" of a suffix tree, and this is the reason why we describe it in this section; anyway, a suffix array can be defined for a generic trie [16]. The suffix array SA_T of a trie T is a pair of arrays: the *sort array* A_T and the *longest common prefix array* LCP_T; $A_T[i]$ points to the i-th leaf of T in the lexicographic order, and LCP_T is the length of the longest common prefix of the strings associated with leaves i and $i+1$, again according to the lexicographic order. If T is a suffix tree, then the corresponding suffix array SA_T is usually stored through the sort array alone. In other words, the suffix array of a suffix tree T_S is a single array A_{T_S} containing the lexicographically sorted sequence of the suffixes of S: $A_{T_S}[i]$ is an integer that indicates the starting position of the i-th suffix inside string S. If $N = |S| \leq 2^{32}$, which is true in most practical cases, then A_{T_S} occupies $4N$ bytes, i.e. four times less than a typical suffix tree. The suffix array can be used to perform `prefix-search(`K`)`. Let $i_{\min} = \min\{i \mid \text{the suffix } S[i] \text{ is not lexicographically smaller than } K\}$, and let $i_{\max} = \max\{i \mid \text{the suffix } S[i] \text{ is not lexicographically bigger than } K\}$; the values of $i_{\min}$ and $i_{\max}$ are determined through a binary search in the array, then the suffixes $S[i_{\min}], S[i_{\min}+1], \ldots, S[i_{\max}]$ are produced in output as the result of the query. The I/O complexity of this algorithm is $O\left((|K|/B)\log N + k/B\right)$, where $k = i_{\max} - i_{\min} + 1$ is the number of results to the query.

It remains to be seen how much time does it take to build a suffix array. A thorough work in this sense has been done by Crauser and Ferragina [13], who analyzed six different algorithms for suffix array construction from both a theoretical and experimental point of view; in this paper, anyway, we will survey only three of them, since the other three offer inferior performances from both points of view. Crauser and Ferragina use the I/O model for their theoretical study, but they choose to perform a separate accounting of bulk I/Os and random I/Os. We have a *bulk I/O* when a sequence of at least two contiguous disk pages is accessed; instead, a *random I/O* is any single-page disk access which is not part of a bulk I/O. This distinction is motivated by current disk technology: in a modern hard disk, in fact, the access time is nearly completely due to the time required to move the disk head and position it accurately over the desired track; once the head is in place, the high rotational speed of the disk and the high areal density of the tracks make it possible to read consecutive blocks very quickly. As a consequence, [13] suggests that bulk I/Os must be considered less expensive than random I/Os: algorithm $\mathcal{A}_1$ may perform a higher number of I/Os than algorithm $\mathcal{A}_2$, but if the fraction of random I/Os is higher for $\mathcal{A}_1$ then it may prove the faster algorithm in practice. The experimental part of [13] proves this observation is well founded.

From a theoretical point of view, the fastest known algorithm for the construction of suffix arrays is

the one proposed by Manber and Myers together with the suffix array itself [23]. The MM algorithm is organized into $\lceil \log_2 N + 1 \rceil$ stages. In the first stage, the N suffixes of S are put into $|\Sigma|$ distinct buckets according to their first symbol; it is assumed that the buckets are arranged following the lexicographic order of the symbols, so that after the first stage the suffixes are lexicographically sorted according to their first symbol. In the subsequent stages, the content of the buckets is further partitioned by sorting (via radix sort) according to an ever increasing number of symbols: to be precise, during stage i the suffixes are lexicographically sorted according to the first 2^i symbols, therefore at the end of the stage the suffixes in the same bucket exhibit a common prefix of length 2^i. After the last stage each bucket contains exactly one suffix, so A_{T_S} is obtained by looking up the buckets in order. While this algorithmic idea is simple, its time/space efficient implementation is not trivial; anyway, it can be proved that each sorting stage has $O(N)$ worst-case I/O complexity and uses $4N$ extra space, hence the algorithm of Manber and Myers requires $O(N \log N)$ I/Os and $8N$ overall space. The I/Os come from repeated sorting operations, and they are all random.

The algorithm of Gonnet et al. [20] similarly takes an incremental approach to the construction of the suffix array: during each step a new portion SA_{int} of the array is calculated in internal memory, then it is merged with the previously-built portion SA_{ext} which is stored in external memory. The string S is loaded into main memory m cells at a time, where $m = \lambda M$ and $\lambda < 1$ is a constant that is chosen to ensure that all necessary data structures fit into main memory; the GBS algorithm then requires $\lceil N/m \rceil = \Theta(N/M)$ steps to complete. For ease of presentation we assume that m divides N evenly. During step i, $1 \leq i \leq N/m$, substring $S[(i-1)m+1, im]$ is taken into main memory, then the suffix array SA_{int} is generated which contains only the suffixes beginning inside $S[(i-1)m+1, im]$; note that this may require further accesses to disk since it may be necessary to know $S[im+1, (i+1)m]$ to perform the comparison of two suffixes that begin in the current block of S. When SA_{int} is ready, it is merged with SA_{ext} with the help of a counter array C that is built as follows: S is scanned from left to right and $C[j]$ is incremented every time a suffix $S[i, N]$ lexicographically lies between $SA_{int}[j-1]$ and $SA_{int}[j]$. With the help of C the position of the new suffixes of SA_{int} inside SA_{ext} can be easily determined, and the merging process then requires just a scan of SA_{ext}. It is not difficult to prove that at the end of step i SA_{ext} is the lexicographically sorted set of suffixes $S[0, N], S[1, N], \ldots, S[im, N]$, therefore $SA_{ext} = A_{T_S}$ when the last step is over. The theoretical analysis of the GBS algorithm demonstrates that it requires $8N$ memory cells and $O\left((N^3 \log M)/(MB)\right)$ I/Os in the worst case, which is completely unappealing if compared to the MM algorithm. Crauser and Ferragina, anyway, make two crucial observations.

- Worst-case analysis is too pessimistic in this case: under very mild assumptions on the input string S (assumptions which are usually verified in practice) the I/O complexity of the algorithm is reduced to $O(N^2/M^2)$.
- Nearly all I/Os are bulk because they are caused by scan operations.

For these reasons the conjecture is made that the GBS algorithm may still be efficient in practice. Indeed, Crauser and Ferragina implemented the algorithms of both [23] and [20], and experimental evidence shows that the GBS algorithm decidedly outperforms the one of Manber and Myers. As a matter of fact, the MM algorithm is faster as long as all its data structures are contained in main memory, but when this is no longer true the performance of the MM algorithm quickly declines because of random I/Os, and it becomes unacceptable even for mildly sized problem instances. This result shows that the I/O model of computation, which has been explicitly designed to model disks, may nonetheless lead to wrong conclusions.

Moving from the above observations, Crauser and Ferragina propose a new algorithm of their own, which we will call the *L-pieces algorithm*. The name is justified by the fact that L sorted arrays A_1, A_2, ..., A_L are built instead of one; array A_i stores the lexicographically ordered set of suffixes $\{S[i, N], S[i+$

$L, N], S[i + 2L, N], \ldots\}$[¶]. Array A_L is built first: the set of strings $\{S[L, 2L - 1], S[2L, 3L - 1], \ldots\}$ is sorted, then each string is replaced by its rank in the sorted list, thus composing an auxiliary text S'. The sort array $A_{T_{S'}}$ for S' is then built in internal memory, and the suffix array is finally obtained by setting $A_L[i] = A_{T_{S'}}[i] \cdot L$, $1 \leq i \leq N/L$. Once A_L is known, the other sorted arrays can be quickly built by observing that a suffix $S[i + kL, N]$ in A_i is the concatenation of character $S[i + kL]$ and the suffix $S[i + 1 + kL, N]$ in A_{i+1}: consequently, A_i can be built from A_{i+1} by sorting N/L pairs of the form $(S[i + kL], \mathrm{pos}_{i+1+kL})$, where pos_j is the rank of suffix $S[j, N]$ in A_{i+1}.

By setting $L = 4$, Crauser and Ferragina obtain a suffix array construction algorithm that needs $6N$ memory cells of working space, $O(\mathrm{sort}(N) \log N)$ random I/Os and $O((N/M) \log(N/M))$ bulk I/Os. In other words, the L-pieces algorithm exhibits a better theoretical complexity than the GBS algorithm, and it also proves to be somewhat faster in practice [13]; anyway, it must be remembered that the suffix array it gives is split into pieces, which may or may not be acceptable depending on the application. In the most favorable case, the query performance with such a structure is slowed down by a constant factor L.

References

[1] Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey Scott Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *In Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA99)*, pages 1116–1127, 1999.

[2] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proc. of the 19th ACM STOC*, pages 305–314, 1987.

[3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS87)*, pages 204–216, Los Angeles, USA, oct 1987.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[6] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In W. K. Gilio, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.

[7] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS96)*, pages 560–569. IEEE Computer Society Press, oct 1996.

[8] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[9] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS2000)*, pages 399–409, Redondo Beach, CA, USA, nov 2000. IEEE Computer Society Press.

[¶]Again, for ease of presentation we suppose that L divides N evenly.

[10] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache-oblivious search trees via trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA02)*, pages 39–48, San Francisco, CA, USA, jan 2002.

[11] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA95)*, pages 139–149, San Francisco, CA, USA, jan 1995.

[12] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In ACM Press, editor, *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithmics (SODA96)*, pages 383–391, Atlanta, USA, jan 1996.

[13] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[14] R. de la Briandais. File searching using variable-length keys. In *Proceedings of the AFIPS Western Joint Computer Conference*, pages 295–298, San Francisco, USA, mar 1959.

[15] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS97)*, pages 137–143, Miami Beach, FL, USA, oct 1997. IEEE Computer Society Press.

[16] Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayut Muthu Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS98)*, pages 174–185, Palo Alto, CA, USA, nov 1998. IEEE Computer Society Press.

[17] Paolo Ferragina and Roberto Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, mar 1999.

[18] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, sep 1960.

[19] Matteo Frigo, Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS99)*, pages 285–297, oct 1999.

[20] Gaston H. Gonnet, Ricardo A. BaezaYates, and Tim Snider. *New indices for text: PAT trees and PAT arrays*, chapter 5, pages 66–82. Information Retrieval: Algorithms and Data Structures. Prentice-Hall, 1992.

[21] H. J. Gray and N. S. Prywes. Outline for a multi-list organized system. In *Proceedings of the 1959 Annual ACM National Conference*, Cambridge, USA, sep 1959.

[22] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP81)*, pages 417–431, jul 1981.

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

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

[25] D.R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, oct 1968.

[26] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT Department of Electrical Engineering and Computer Science, jun 1999.

[27] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[28] Elizabeth A. M. Shriver. *Performance modeling for realistic storage devices*. PhD thesis, Department of Computer Science, New York University, 1997.

[29] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[30] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2-3):148–169, 1994.