PRACTICAL IMPLEMENTATION OF RANK AND SELECT QUERIES *

Rodrigo González
Dept. of Computer Science, University of Chile, Chile
rgonzale@dcc.uchile.cl

Szymon Grabowski
Computer Engineering Department, Technical University of Łódź, Poland
sgrabow@zly.kis.p.lodz.pl

Veli Mäkinen
Technische Fakultät, Universität Bielefeld, Germany
vmakinen@cs.helsinki.fi

Gonzalo Navarro
Dept. of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract Research on succinct data structures has made significant progress in recent years. An essential building block of many of those techniques is a data structure to perform *rank* and *select* operations over a bit array. The first operation tells how many bits are set up to some position, and the second the position of the i-th bit set. Albeit there exist constant-time solutions that require sublinear extra space, the practicality of those solutions against more naive ones has not been carefully studied. In this paper we show some results in this respect, which suggest that in many practical cases the simpler solutions are better in terms of time and extra space.

Keywords: Succinct data structures, bit arrays.

*First author supported by Mecesup Grant UCH 0109 (Chile). Fourth author supported by Fondecyt Grant 1-020831 (Chile).

Introduction

Recent years have witnessed an increasing interest on *succinct* data structures, motivated mainly by the growth over time on the size of textual information. Among the most important data structures of this kind are the succinct data structures for pattern matching, commonly known as *succinct full-text indexes*. There is a wide variety of these indexes, each one with a different trade-off between search time and space occupancy [2, 14, 5, 11, 3]. Interestingly, some of them, so-called *self-indexes*, do not need the original text to operate, as they contain enough information to recreate it.

An essential building block of many of those proposals is a data structure to perform *rank* and *select* operations over a bit array $B[1 \dots n]$ with n bits. The first operation, $rank(B, i)$, is the number of 1's in $B[1 \dots i]$, $rank(B, 0) = 0$. The second, $select(B, j)$, gives the position of the j-th bit set in B.

The first results achieving constant time on $rank()$ and $select()$ [10, 1]used $n + o(n)$ bits: n bits for B itself and $o(n)$ additional bits for the data structures used to answer $rank()$ and $select()$ queries. Further refinements [12–13]achieved constant time on the same queries by using $nH_0(B) + o(n)$ bits overall, where $H_0(B)$ is the zero-order entropy of B. Recent lower-bounds [9]reveal that those results are almost optimal also on the $o(n)$ terms.

It is unclear how efficient are in practice those solutions, nor how irrelevant is the $o(n)$ extra space in practice. In this paper we focus on the most promising solutions among those that use $n + o(n)$ bits.

All our experiments ran on an AMD Athlon of 1.6 GHz, 1 GB of RAM, 256 KB cache, running Linux. We use the GNU gcc compiler for C++ with full optimizations. We measure user times.

1. Rank Queries

1.1 The Constant-Time Classical Solution

The constant-time solution for *rank* is relatively simple [6, 10–1]. We divide the bit array into blocks of length $b = \lfloor \log(n)/2 \rfloor$ (all our logarithms are in base 2). Consecutive blocks are grouped into superblocks of length $s = b\lfloor \log n \rfloor$.

For each superblock j, $0 \leq j \leq \lfloor n/s \rfloor$ we store a number $R_s[j] = rank(B, j \cdot s)$. Array R_s needs overall $n/b = O(n/\log n)$ bits.

For each block k of superblock $j = k$ div $\lfloor \log n \rfloor$, $0 \leq k \leq \lfloor n/b \rfloor$, we store a number $R_b[k] = rank(B, k \cdot b) - rank(B, j \cdot s)$. Array R_b needs $(n/b) \log s = O(n \log \log n / \log n)$ bits.

Finally, for every bit stream S of length b and for every position i inside S, we precompute $R_p[S,i] = rank(S,i)$. This requires $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log\log n)$ bits.

The above structures need $O(\frac{n}{\log n} + \frac{n \log\log n}{\log n} + \sqrt{n} \log n \log\log n) = o(n)$ bits. They compute *rank* in constant time as $rank(B,i) = R_s[i \text{ div } s]$ $+R_b[i \text{ div } b] + R_p[B[(i \text{ div } b) \cdot b + 1 \ldots (i \text{ div } b) \cdot b + b], i \bmod b]$.

This structure is can be implemented with little effort and promises to work fast. Yet, for e.g., $n = 2^{30}$ bits, the $o(n)$ extra space is $6.67\% + 60\% + 0.18\% = 66.85\%$ of n, which is not so negligible.

1.2 Resorting to Popcounting

The term *popcount* (population count) refers to counting how many bits are set in a bit array. We note that table R_p can be replaced by popcounting, as $R_p[S,i] = popcount(S \ \& \ 1^i)$, where "&" is the bitwise *and* and 1^i is a sequence of i 1's (obtained for example as $2^i - 1$). This permits us removing the second argument of R_p, which makes the table smaller. In terms of time, we perform an extra *and* operation in exchange for either a multiplication or an indirection to handle the second argument. The change is clearly convenient.

Popcounting can be implemented by several means, from bit manipulation in a single computer register to table lookup. We have found that the implementation of *Gnu* g++ is the fastest:

```
popc = { 0, 1, 1, 2, 1, 2, 2, 3, 1, ... }
popcount = popc[x & 0xFF] + popc[(x >> 8) & 0xFF]
         + popc[(x >> 16) & 0xFF] + popc[x >> 24]
```

where *popc* is a precomputed popcount table indexed by bytes.

Yet, this table lookup solution is only one choice among several alternatives. The width of the argument of the precomputed table has been fixed at 8 bits and b has been fixed at 32 bits, hence requiring 4 table accesses. In a more general setup, we can choose $b = \log(n)/k$ and the width of the table argument to be $\log(n)/(rk)$, for integer constants r and k. Thus the number of table accesses to compute *popcount* is r and the space overhead for table R_b is $k \log\log(n)/\log(n)$. What prevents us to choose minimal r and k is the size of table *popc*, which is $n^{\frac{1}{rk}} \log\log(n)$. Condition $rk > 1$ yields a space/time trade-off.

In practice, b should be a multiple of 8 because the solutions to *popcount* work at least by chunks of whole bytes. With the setting $s = b \log n$, and considering the range $2^{16} < n \leq 2^{32}$ to illustrate, the overall extra space (not counting R_p) is 112.5% with $b = 8$, 62.5% with $b = 16$, 45.83% with $b = 24$ and 34.38% with $b = 32$.

We have tried the reasonable (k, r) combinations for $b = 16$ and $b = 32$: (1) $b = 32$ and a 16KB *popc* table needing 2 accesses for *popcount*, (2) $b = 16$ and a 16KB *popc* table needing 1 access for *popcount*, (3) $b = 16$ and a 256-byte *popc* table needing 2 accesses for *popcount*, and (4) $b = 32$ and a 256-byte *popc* table needing 4 accesses for *popcount*. Other choices require too much space or too many table accesses. We have also excluded $b = 8$ because its space overhead is too high and $b = 24$ because it requires non-aligned memory accesses (see later).

Figure 1 (left) shows execution times for $n = 2^{12}$ to $n = 2^{30}$ bits. For each size we randomly generate 200 arrays and average the times of 1,000,000 *rank* queries over each. We compare the four alternatives above as well as the mentioned method that does not use tables. As it can be seen, the combination (4), that is, $b = 32$ making 4 accesses to a table of 256 entries, is the fastest in most cases, and when it is not, the difference is negligible.

On the other hand, it is preferable to read word-aligned numbers than numbers that occupy other number of bits such as $\log n$, which can cross word boundaries and force reading two words from memory. In particular, we have considered the alternative $s = 2^8$, which permits storing R_b elements as bytes. The space overhead of R_b is thus only 25% with $b = 32$ (and 50% for $b = 16$), and accesses to R_b are byte-aligned. The price for such a small s is that R_s gets larger. For example, for $n = 2^{20}$ it is 7.81%, but the sum is still inferior to the 34.38% obtained with the basic scheme $s = b \log n$. Actually, for little more space, we could store R_s values as full 32-bit integers (or 16-bit if $\log n \leq 16$). The overhead factor due to R_s becomes now 32/256 (or 16/256), which is at most 12.5%. Overall, the space overhead is 37.5%, close to the non-aligned version. Figure 1 (left) shows that this alternative is the fastest, and it will be our choice for popcount-based methods.

Note that up to $n = 2^{20}$ bits, the original bit array together with the additional structures need at most 176 KB with $b = 32$, and 208 KB with $b = 16$. Thus our 256 KB cache accommodates the whole structure. However, for $n = 2^{22}$, we need 512 KB just for the bit array. Thus the cache hit ratio decreases as n grows, which explains the increase in query times that should be constant. We make systematic study of the cache effect in Sect. 1.3.

1.3 Using a Single Level Plus Sequential Scan

At this point we still follow the classical scheme in the sense that we have two levels of blocks, R_s and R_b. This forces us to make two memory accesses in addition to accessing the bit array block. We consider now the alternative of using the same space to have a single level of blocks, R_s, with one entry each $s = 32 \cdot k$ bits, and using a single 32-bit integer to store the ranks. To answer a $rank(B,i)$ query, we would first find the latest R_s entry that precedes i, and then sequentially scan the array, popcounting in chunks of $w = 32$ bits, until reaching the desired position, as follows: $rank(B,i) = R_s[i \text{ div } s] + \sum_{j=(((i \text{ div } s)\cdot s) \text{ div } w)+1 \ldots (i \text{ div } w)-1} popcount(B[j \cdot w + 1 \ldots j \cdot w + w]) + popcount(B[(i \text{ div } w) \cdot w + 1 \ldots (i \text{ div } w) \cdot w + w] \ \& \ 1^{i \bmod w})$.

Note that the sequential scan accesses at most k memory words, and the space overhead is $1/k$. Thus we have a space/time trade-off. For example, with $k = 3$ we have approximately the same space overhead as in our preferred two-level version.

Figure 1 (right) compares the execution time of different trade-offs against the best previous alternatives. For the alternative of using only one level of blocks, we have considered extra spaces of 5%, 10%, 25%, 33% (close to the space of our best two-level alternative), and 50%.

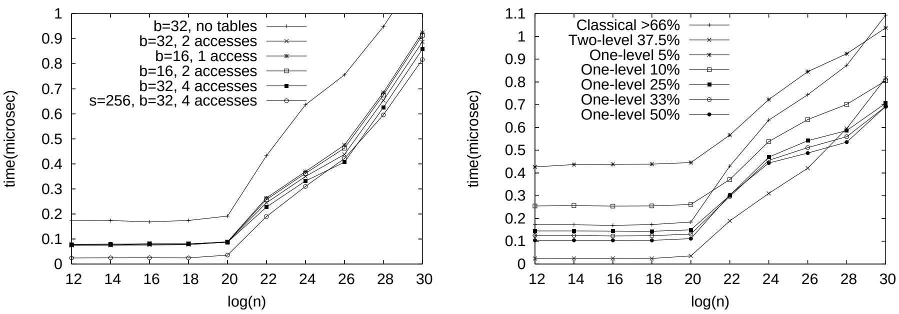

Figure 1. On the left, different popcount methods to solve *rank*. On the right, different mixed approaches to solve *rank*: classical alternative, popcounting with two levels of blocks, and popcounting with one level of blocks.

We can see that the direct implementation of the theoretical solution is far from competitive: It wastes the most space and is among the slowest. Our two-level popcount alternative is usually the fastest by far, showing that the use of two levels of blocks plus an access to the bit array is normally better than using the same space (and even more) for a single level of blocks. Yet, note that the situation is reversed for large n. The reason is the locality of reference of the one-level versions: They perform one access to R_s and then a few accesses to the bit array (on average, 1 access with 50% overhead, 1.5 accesses with 33% overhead and 2 accesses with 25% overhead). Those last accesses are close to each other, thus from the second on they are surely cache hits. On the other hand, the two-level version performs three accesses (R_s, R_b, and the bit array) with no locality among them. When the cache hit ratio decreases significantly, those three nonlocal accesses become worse than the two nonlocal accesses (plus some local ones) of the one-level versions.

Thus, which is the best choice among one and two levels depends on the application. Two levels is usually better, but for large n one can use even less space and be faster. Yet, there is no fixed concept of what is "large", as other data structures may compete for the cache and thus the real limit can be lower than in our experiments, where only the *rank* structures are present.

2. Select Queries

2.1 Binary Searching with Rank

A simple, yet $O(\log n)$ time, solution to $select(B, j)$, is to binary search in B the position i such that $rank(B, i) = j$ and $rank(B, i-1) = j-1$. Hence, the same structures used to compute $rank(B, i)$ in constant time can be used to compute $select(B, j)$ in $O(\log n)$ time.

More efficient than using $rank(B, i)$ as a black box is to take advantage of its layered structure: first binary search for the proper superblock using R_s, then binary search that superblock for the proper block using R_b, and finally binary search for the position inside the block.

For the search in the superblock of s bits, there are three alternatives: ($2a$) binary search using R_b, ($2b$) sequential search using R_b (since there are only a few blocks inside a superblock), and ($2c$) sequential search using popcount. The last alternative consists of simply counting the number of bits set inside the superblock, and has the advantage of not needing array R_b at all. For the search in the last block of b bits, binary search makes little sense because *popcount* proceeds anyway bytewise, so we have considered two alternatives: ($3a$) bytewise search using popcount plus bitwise search in the final byte, and ($3b$) sequential bitwise search in the b bits.

In the case of *select*, the density of the bit array may be significant. We have generated bit arrays of densities (fraction of bits set) from 0.001 to 1. For each density we randomly generated 50 different arrays of each size. For each array, we average the times of 400,000 *select* queries.

The results for the binary search version are almost independent of the density of bits set in B, hence we show in Figure 2 (left) only the case of density 0.4. We first compare alternatives $(2a, 3b)$, $(2b, 3b)$ and $(2c, 3b)$. Then, as $(2c, 3b)$ turns out to be the fastest, we consider also $(2c, 3a)$, which is consistently the best. We have also plotted the basic binary search (not level-wise) to show that it is much slower than any other. In this experiment we have used $b = 32$ and $s = b \log n$.

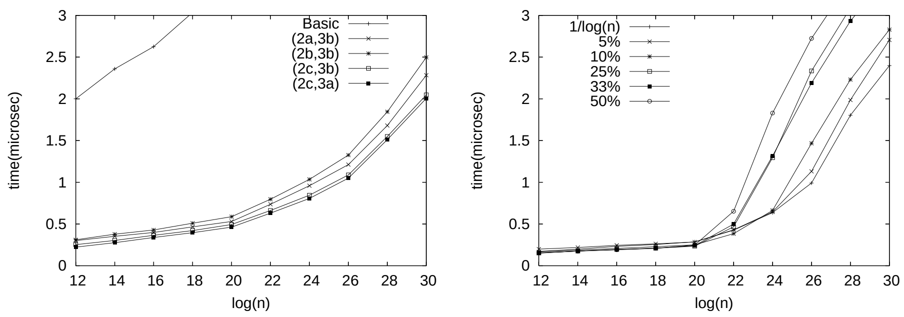

Figure 2. Comparison of alternatives to solve *select* by binary search (left), and comparison of different space overheads for *select* based on binary search (right).

Note that the best alternative only requires space for R_s (that is, $1/32$), as all the rest is solved with sequential scanning. Once it is clear that using a single level is preferable for *select*, we consider speeding up the accesses to R_s by using 32-bit integers (or 16-bits when $\log n \leq 16$). Moreover, we can choose any sampling step of the form $s = k \cdot b$ so that the sequential scan accesses at most k blocks and we pay $1/k$ overhead.

Figure 2 (right) compares different space overheads, from 5% to 50%. We also include the case of $1/\log n$ overhead, which is the space needed by the version where R_s stores $\log n$ bit integers instead of 32 bits. It can be seen that these word-aligned alternatives are faster than those using exactly $\log n$ bits for R_s. Moreover, there is a clear cache effect as n grows. For small n, higher space overheads yield better times as expected, albeit the difference is not large because the binary search on R_s is a significant factor that smoothes the differences in the sequential search. For larger n, the price of the cache misses during the binary search in R_s is the dominant factor, thus lower overheads take much less time because their R_s arrays are smaller and their cache hit ratios are higher. The sequential search, on the other hand, is not so important because only the first access may be non-local, all the following ones are surely cache hits. Actually, the variant of $1/\log n$ overhead is finally the fastest because for $n = 30$ it is equivalent to 3.33% overhead.

The best alternative is the one that balances the number of cache misses during binary search on R_s with those occurring in the sequential search on the bit array. It is interesting, however, that a good solution for *select* requires little space.

2.2 The Constant-Time Solution

The constant time solution to *select*(B, j) is significantly more complex than for *rank*(B, i). Clark's structure [1]uses a three-level directory tree and requires $\frac{3n}{\lceil \log \log n \rceil} + O(\sqrt{n} \log n \log \log n)$ bits of extra space. For example, for $n = 2^{30}$, the overhead is 60%.

Figure 3 shows the execution times for our implementation of Clark's *select* (we show different lines for different densities of the bit arrays). We note that, although the time is $O(1)$, there are significant differences as we change n or the density of the arrays (albeit of course those differences are bounded by a constant). For lack of space we cannot explain the reason for the curve shapes.

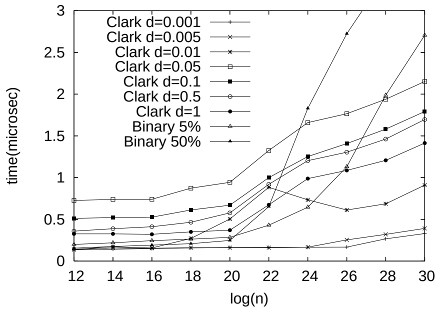

Figure 3. Comparison of Clark's select on different densities and two binary search based implementations using different space overheads.

The plot also shows the time for our binary search versions using 5% and 50% space overhead. For very low densities (up to 0.005 and sometimes 0.01), Clark's implementation is superior. However, we note that for such low densities, the *select* problem is trivially solved by explicitly storing all the positions of all the bits set (that is, precomputing all answers), at a space overhead that is only 32% for density 0.01. Hence this case is not very interesting. For higher densities, our binary search versions are superior up to $n = 2^{22}$ or 2^{26} bits, depending on the space overhead we chose (and hence on how fast we want to be for small n). After some point, however, the $O(\log n)$ nature of the binary search solution shows up, and the constant-time solution of Clark finally takes over. We remark that Clark's space overhead is 60% at least.

3. The Cache Effect

Hardware cache and memory management play a key role in the user time for *rank* and *select* queries. Solutions that are clearly constant time, which is especially obvious for *rank*, turn out to increase almost linearly with $\log n$ because of cache effects (recall Figure 1).

The hardware components involved are the L1 and L2 cache and the TLB (Translation Lookaside Buffer). L1 and L2 are a first- (smaller and faster) and second-level memory caches, while the TLB is a small associative memory that maps virtual to physical page table resolutions. In the Linux version we use, the TLB is sequentially scanned to solve each address, thus the cost of every access to memory depends linearly on the amount of memory allocated [4, Sect. 3.8, pp. 43–44]. This is usually mild because of the locality of reference exhibited by most applications, but this is not the case, for example, with *rank*() queries.

We use Cachegrind (`http://webcvs.kde.org/valgrind/cachegrind`) to measure the effect of L1 and L2 cache. Cachegrind, a part of a larger tool known as Valgrind, is a tool for doing cache simulations and annotating the source line-by-line with the number of cache misses. In particular, it records: L1 instruction cache reads and misses; L1 data cache reads and read misses, writes and write misses; L2 unified cache reads and read misses, writes and writes misses. In our case, we use this tool to count the number of cache hits and misses of *rank* and *select* queries.

From our configuration we know that an L1 miss will typically cost around 10 cycles, and an L2 miss costs around 200 cycles. With the Cachegrind tool, we obtained that there is a very close correlation between the predicted number of cycles (mostly coming from the L1 and L2 misses) and the measured user time for queries. Moreover, optimizing the cost of L1 and L2 misses in terms of number of cycles we get numbers very close to the predicted 10 and 200.

For *rank* queries the miss ratio is stabilized for $n \geq 2^{26}$ (that is, all are L2 misses at that point), but the user time still grows with n. Then the TLB size comes into play, because as the size n of the arrays grows, the search time on the TLB will also grow linearly with n.

For *select* queries the miss ratio never reaches 100%, and the number of cycles due to cache misses (without considering TLB) can perfectly explain the measured user time.

Overall, we obtained the following general model to explain the user time in microseconds: $U'_t = L1_m\, C_{L1_m} + L2_m\, C_{L2_m} + C_{TLB}(n)$. Here, U'_t is the estimated user time, $L1_m$ is the number of L1 misses, $L2_m$ the number of L2 misses, C_{L1_m} is the cost of L1 misses (fixed at $10/(1.6 \times 10^9) \times 10^6$), that is, 10 cycles divided by 1.6 GHz times microseconds), C_{L2_m} the cost of L2 misses (fixed at $200/(1.6\times 10^9)\times 10^6$), and $C_{TLB}(n)$ is the cost to access the TLB.

Given the operation mechanism of the TLB we choose the model $C_{TLB}(n) = a + bn$, where a stands for the time spent on the other CPU operations not related to cache misses nor to the TLB access, and b is related to the TLB traversal time.

Figure 4 (left) compares the actual versus the estimated user time, for *rank* queries over a *rank* structure of two levels where the numbers are word-aligned. This structure has a space overhead of 37.5%. Figure 4 (right) compares the actual versus estimated user time for *select* queries over Clark's *select* structure, with an overhead of 50%. Our least squares estimation gives $C_{TLB}(n) = 0.0233678 + 0.390059 \times 10^{-9}\, n$ for *rank* and $C_{TLB}(n) = 0.330592 + 0.918287 \times 10^{-9}\, n$ for *select*, both with correlation above 0.99. Note that the fitting is better for *rank*, which is expected since *select* executes a varying (albeit constant) number of instructions, whereas we assume a fixed number of instructions.

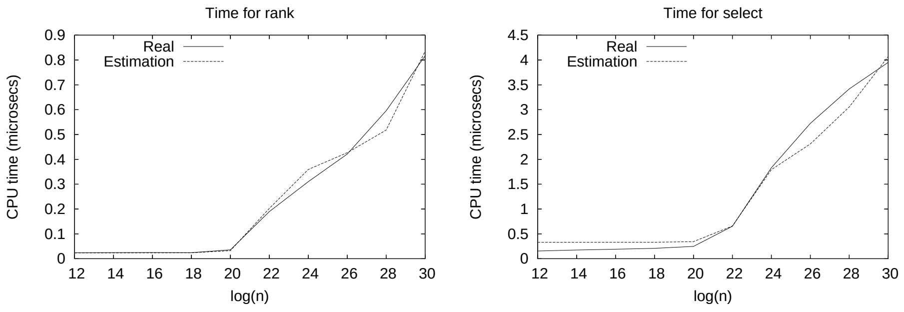

Figure 4. Comparison between estimated user time versus the actual user time.

References

[1] D. Clark. *Compact Pat Trees.* PhD thesis, Univ. Waterloo, 1996.

[2] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *41st IEEE FOCS*, pp. 390–398, 2000.

[3] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th SPIRE*, LNCS 3246, pp. 150–160, 2004.

[4] M. Gorman. *Understanding the Linux Virtual Memory Manager.* Prentice Hall, 2004.

[5] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM SODA*, pp. 841–850, 2003.

[6] G. Jacobson. *Succinct Static Data Structures.* PhD thesis, CMU-CS-89-112, Carn. Mellon Univ., 1989.

[7] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. 15th CPM*, LNCS 3109, pp. 420–433, 2004.

[8] V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Tech. Rep. C-2004-20, Univ. Helsinki, Finland, April 2004.

[9] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. SODA*, 2005.

[10] I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS n. 1180, pp. 37–42, 1996.

[11] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Disc. Alg.*, 2(1):87–114, 2004.

[12] R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th ICALP*, pp. 595–604, 1999.

[13] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM SODA*, pp. 233–242, 2002.

[14] K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proc. 13th ACM-SIAM SODA*, pp. 225–232, 2002.