Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Benjamin Schlegel, Rainer Gemulla, Wilfgang Lehner

k-ary search on modern processors

Erstveröffentlichung in / First published in:

DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 28.07.2009. ACM. S. 62-60. ISBN 978-1-60558-701-1

DOI: https://doi.org/10.1145/1565694.1565705

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-792241

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

k-Ary Search on Modern Processors

Benjamin Schlegel
Technische Universität Dresden
benjamin.schlegel@tu-dresden.de

Rainer Gemulla
IBM Almaden Research Center
rgemull@us.ibm.com

Wolfgang Lehner
Technische Universität Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

This paper presents novel tree-based search algorithms that exploit the SIMD instructions found in virtually all modern processors. The algorithms are a natural extension of binary search: While binary search performs one comparison at each iteration, thereby cutting the search space in two halves, our algorithms perform k comparisons at a time and thus cut the search space into k pieces. On traditional processors, this so-called k-ary search procedure is not beneficial because the cost increase per iteration offsets the cost reduction due to the reduced number of iterations. On modern processors, however, multiple scalar operations can be executed simultaneously, which makes k-ary search attractive. In this paper, we provide two different search algorithms that differ in terms of efficiency and memory access patterns. Both algorithms are first described in a platform independent way and then evaluated on various state-of-the-art processors. Our experiments suggest that k-ary search provides significant performance improvements (factor two and more) on most platforms.

1. INTRODUCTION

Searching is a fundamental operation that is used in almost every domain of computer science. The classical problem is to retrieve from a dataset of disjoint key/value pairs (i) the value for a single given key or (ii) all the pairs within a range of two given keys. For example, suppose that the dataset consists of a set of persons. Then, a single-key query might ask for a person with a given name, while a range query might ask for all persons born between 01/01/1980 and 12/31/1980.

The classical search problem has been studied extensively in the literature. Apart from linear search, one distinguishes sort-based, tree-based, and hash-based search algorithms. Hash-based algorithms are well-suited for single-key lookups, but they require memory over and above that to store the base data[1] and—since the distribution of keys to buckets is randomized—they perform poorly in the presence of range or nearest-key queries. In this paper, we restrict our attention to the former two classes, i.e., sort-based and tree-based search. When the dataset is sorted, binary search constitutes the provably best algorithm of the sort-based class of algorithms in terms of time complexity. Each step of a binary search halves the search space by performing one comparison so that the total number of comparisons is logarithmic in the dataset size. Similar arguments hold for tree-based search based on a binary search tree.

The downside of binary search is that it does not make use of the SIMD capabilities of modern processors. On IBM's Cell processor, for example, the cost of a single 32-bit scalar comparison is identical to the cost of four 32-bit scalar comparisons executed simultaneously. A naive execution of binary search would therefore "waste" three comparisons at each step. A natural idea to boost the speed of searching is, therefore, to not run binary search but k-ary search, $k > 2$, where each step divides the search space into k parts based on the outcome of $k - 1$ comparisons (e.g., $k = 5$ for the Cell with 4-way vector registers). Although this approach does not affect the asymptotic time complexity of search, it might significantly reduce the actual execution cost.

In this paper, we take a closer look at k-ary search on SIMD architectures. Our goal is to determine which SIMD operations are essential for k-ary search to be more efficient than binary search and by what margin the former outperforms the latter on selected processors. Furthermore, we show that an additional reduction in execution time can be achieved by rearranging the dataset in an order more appropriate for k-ary search. Our reordering is based on a linearization of a k-ary search tree.

In what follows, we consider a somewhat idealistic scenario in which the dataset is stored in main memory and is static (or updated infrequently, e.g., on a daily or monthly basis). Although these restrictions do not always hold in practice, they allow us to focus on the advantages of modern architectures over more conventional ones. To see this, suppose that the dataset is not stored in memory but on hard disk. Then, the I/O cost of reading the data would dominate the cost of search and usually outweigh any performance gain due to SIMD instructions. For the former reason, one typically keeps as much of the data in memory as possible, in which case k-ary search is attractive. Similarly, if the dataset is updated frequently with respect to the number of searches, the maintenance cost of both the sorted-array representation and the linearized-tree representation become substantial.

[1] Although some modern hashing techniques [11] can achieve up to 95-99% occupancy.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

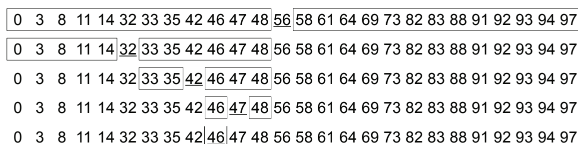


Figure 1: Binary search for key 46 with $n = 26$

In this case, balanced search trees appear to be the method of choice. We conjecture that SIMD instructions will also be valuable to speed up search on those trees, but this is beyond the scope of our current work.

2. PREREQUISITES

In this section, we review various implementations of binary search and discuss their advantages and disadvantages when run on modern processors. We also provide a brief overview of the SIMD instructions found in these processors and summarize previous work to exploit these instructions for search.

In what follows, we assume that the input to the search algorithm is (1) a sorted array of unique keys[2] $A = (a_1, \ldots, a_n)$, where $a_i < a_j$ for $1 \leq i < j \leq n$, and (2) a search key s. When $s \in A$, the algorithm returns index i^* such that $a_{i^*} = s$. Otherwise, the algorithm reports that $s \notin A$. In order to be able to exploit SIMD capabilities, we restrict our attention to the case where the keys are of a type natively supported by the underlying processor architectures, i.e., integer and floating point types.

2.1 Binary Search

Binary search is a dichotomic divide-and-conquer search algorithm. In each iteration, the algorithm starts by picking the median key as the so-called *separator key*. The separator divides the search space into two equally-sized sets of keys, henceforth called *partitions*: the keys strictly smaller than the separator (left partition) and the keys strictly larger than the separator (right partition). Note that, since A is sorted, the median can be found at the center position—i.e., $a_{\lfloor n/2 \rfloor}$ in the first iteration—and that the two sets of keys are contiguous. The next step is to compare the separator and the search key. If both are equal, the desired key has been found and the search terminates. Otherwise, the search process is repeated using either the left or the right partition as its input, depending on whether the search key is smaller or larger than the separator, respectively. When the selected partition is empty, we have $s \notin A$ and the search terminates. It can be shown that the algorithm performs $h = \lceil \log_2(n+1) \rceil$ iterations in the worst case and $h - (2^h - h - 1)/n > h - 2$ iterations on average.

Figure 1 illustrates the algorithm's search for key 46 in an example dataset with 26 keys. In each iteration, the separator key is underlined and the left and right partitions are framed. The algorithm performs five iterations in total, which is the worst case for this dataset. The average number of iterations is 4.

Knuth [7] presents various implementations of the generic binary search algorithm. The most widely known implementation uses two variables l (left) and r (right) to store the indexes of the boundaries of the search space. The separator can be found at index $\lfloor (l + r)/2 \rfloor$. This implementation, as well as all the other implementations given by Knuth, heavily relies on branches. In fact, there are three branches in each iteration: one branch for negative termination, one for positive termination, and one for choosing the next partition. As shown by Ross [10], branch misses can lead to serious performance penalties.

2.2 SIMD Instructions For Searching

Apart from their regular instruction sets, modern processors provide SIMD instruction sets (for single instruction, multiple data). For example, Intel's Nehalem, Intel's Xeon, and AMD's Phenom processors support the so-called *Streaming SIMD Extensions* (SSE) instruction sets, which provide 128-bit registers that can hold four 32-bit keys. By supporting simultaneous execution of instructions on multiple elements, these instructions are particularly well-suited to accelerate numerical calculations or computation-intensive applications such as multimedia applications, scientific applications, or encryption. Recent research has shown that data-intensive applications like sorting [2, 4], hash-based search [11], and relational query processing [5, 12] can also benefit from SIMD instructions. This paper complements these techniques by providing efficient methods for sort-based search.

In this section, we briefly review the SIMD instructions that we will be exploiting for search. We assume throughout that the registers of the processor of interest are vectors of $k - 1$ scalars. We classify the instructions into instructions for data loading, element-wise instructions, and horizontal instructions.

Loading. The first instruction takes as input a scalar value and produces as output a vector consisting of $k - 1$ copies of that scalar value. This instruction will mainly be used to generate copies of the search key. The next instruction takes as input an address in memory and, starting from the given address, loads $k-1$ consecutive scalars into a register. This function will mainly be used to load the input data.

Element-wise instructions. We make use of element-wise versions of arithmetic integer functions $(+, -, *, /)$ and comparison functions $(<, \leq, =, \geq, >)$. All functions store their results in another register, which can be treated as a vector of integers for both arithmetic functions and comparison functions ($0 =$ false, $-1 =$ true). We assume that the number of processor cycles required for an element-wise SIMD instruction is significantly smaller than the processing time of the corresponding $k-1$ scalar instructions. This assumption is easily satisfied in practice, where the cost of a SIMD instruction is typically the same as the cost of a *single* scalar instruction.

Horizontal instructions. Our proposed search algorithm relies on so-called horizontal SIMD instructions, which work across the scalars in a register. Our algorithm will make use of either one of the following two horizontal functions: (1) computation of the sum of elements in a vector, (2) computation of a bitmap having the ith bit set to 1 if and only if the most significant bit of the ith scalar is set to 1 (i.e., it is negative). Most architectures provide at least one of these two instructions. If not, they can be emulated by other instructions at the cost of some performance.

[2]The restriction to unique keys simplifies exposition but is not crucial. When run with duplicate keys, our algorithms find *any* copy of the search key; other copies can be found to the left and to the right of the result.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

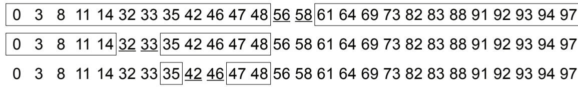


Figure 2: SIMDized binary search for key 46 with $n = 26$, $k = 3$

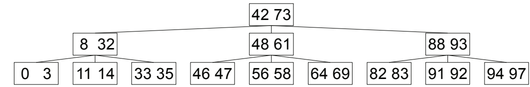


Figure 3: A k-ary search tree for $k = 3$

2.3 Binary Search With SIMD Instructions

Zhou et al. [12] proposed a SIMDized version of binary search that is geared towards small datasets (up to a few hundred keys). The key idea of their approach is to replace the comparison of just the separator with the search key by the comparison of the separator *and its* $k-2$ *subsequent keys* with the search key; see Figure 2 for an example with $k = 3$. An implementation would $<$-compare a register holding $k-1$ copies of the search key with a register that contains the separator and its $k-2$ subsequent keys. If the result of the comparison consists only of -1s (true), all $k-1$ keys are larger than the search key and the search traverses to the left partition. If it consists of only 0s (false), all $k-1$ keys are smaller than the search key and the search traverses to the right partition.[3] Otherwise, the register contains a sequence of 0s followed by a sequence of -1s, in which case the search terminates. The search key, if present in A, then resides at the position in the input register that corresponds to the last 0 in the result of the comparison.

To analyze SIMDized binary search, we conceptually "condense" the dataset to a sequence of $\lceil n/(k-1) \rceil$ blocks of $(k-1)$ elements, retaining their order. Each iteration of SIMDized binary search can be seen to operate on one of the resulting blocks; it either accepts the block or branches to the left or right partition. This mirrors exactly what a binary search algorithm on the condensed dataset would do if it were to find one of the blocks. We can therefore plug in the complexity results for plain binary search, replacing all occurences of n by $\lceil n/(k-1) \rceil$. We find that the worst-case number of iterations of the SIMDized binary search algorithm is given by $\lceil \log_2(n/(k-1)+1) \rceil \geq \lceil \log_2(n+1) - \log_2(k-1) \rceil$. Compared to plain binary search, the worst-case number of iterations is reduced by at most $\log_2(k-1)$ iterations when $(k-1)$ is a power of 2 and by at most $1+\log_2(k-1)$ iterations otherwise. Modern processors typically provide $k = 5$ for 32-bit search keys, which translates to a reduction of the worst-case cost by at most two iterations. Similar arguments hold for the average case.

When the dataset is small, the savings in terms of the number of iterations are significant. In the example of Figure 2, where $k = 3$, the worst-case number of iterations is reduced from 5 to 4, and the average number of iterations is reduced from 4 to approximately 3.15. Under the (quite realistic) assumption that the cost of an iteration of SIMDized search is roughly the same as the cost of an iteration in plain binary search, this directly results in a performance increase. The speedup is roughly $\log_2(n+1)/[\log_2(n+1)-\log_2(k-1)]$. As mentioned previously, the speedup is significant when the dataset is small but is negligible for large n. In fact, the speedup converges to 1 as $n \to \infty$. The goal of this paper is to provide an algorithm that realizes a substantial speedup when n is large.

[3] If the result is all 0s, then the right-most key might be equal to the search key. This case is handled at the end of the search process: If the comparison of the final iteration yields only -1s, the search key may reside directly to the left of the final separator.

3. K-ARY SEARCH

In k-ary search, we pick $k-1$ separators in each iteration, thereby dividing the search space into k partitions. As in binary search, the separators are chosen so that all partitions have equal size, i.e., they correspond to the i/k-quantiles of the dataset, where $1 \leq i < k$. The entire search process can be visualized using a *k-ary search tree*, in which nodes represent separators and children represent partitions. An example of such a tree is shown in Figure 3.

For expository reasons, we start our discussion of k-ary search using a k-ary search tree as input. Note that the search tree itself is not materialized in practice. We then proceed to algorithms for k-ary search that work directly on the sorted-array representation of the data. Finally, we show that a linearized representation of the k-ary search tree is more amenable for k-ary search than a sorted array. This representation can be leveraged in applications in which reordering of the data is acceptable.

3.1 On a k-ary Search Tree

Suppose for the moment that the search tree is *perfect*, i.e., that every node—including the root node—has precisely $k-1$ entries, every internal node has k successors, and every leaf node has the same depth. For example, the tree shown in Figure 3 is perfect. A perfect search tree can be constructed for any dataset of size $n = k^h - 1$ for some integer $h > 0$. We extend our discussion to non-perfect trees in Section 3.4.

The search consists of the following steps, starting at the root:

- S1: *Load* the $k-1$ separators of the current node into a register R.
- S2: *Terminate* with success if the key is present in the register. Otherwise, terminate with failure when the current node is a leaf.
- S3: *Find* the partition that encloses the search key.
- S4: *Move* to the child node corresponding to that partition. Then, go to S1.

In this section, we focus on steps S2 and S3 only; steps S1 and S4 are discussed in subsequent sections.

The key ingredient to making steps S2 and S3 efficient is to use only a few SIMD instructions and no loops. To achieve this goal, we make use of a register S that contains $k-1$ copies of the search key s. Given such a register, step S2 is straightforward: We run a single element-wise $=$-compare instruction on S and R and terminate with success if the resulting vector is non-zero. We terminate with failure if the depth of the current node (maintained in another register) is equal to the depth of the tree. Each check constitutes a

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

Keys	Bin	Bin2	3-ary	Bin4	5-ary	Bin8	9-ary
$1k$	11	10	7	9	5	8	4
$32k$	16	15	10	14	7	13	5
$1M$	21	20	13	19	9	18	7

Table 1: Comparison of worst-case number of iterations

branch, but both branches are uncritical because their result can be predicted accurately (non-termination is the frequent outcome).

Finding the next partition is more involved. Formally, we want to determine the number m, with $0 \leq m < k$, of the partition that encloses s. To do so, we compare S and R using the element-wise <-comparison instruction. The outcome of this comparison is a vector consisting of m "zeros" (comparison false) followed by $k - m - 1$ "minus ones" (comparison true). Thus, to determine m, we compute the horizontal sum of the result of the comparison and add it to $k-1$. On architectures that do not provide horizontal sums, we can leverage the bitmap instruction that replicates the most significant bit of each element into a single scalar. For example, suppose that $k = 5$ and $m = 1$. The result vector $(0, -1, -1, -1)$ would be translated into the scalar value 7, which has the binary representation 0111. The number of set bits corresponds to $k - m - 1$; it can be determined efficiently using either the "population count" instruction or the "count leading zeros" instruction. Thus, both steps S2 and S3 take only a constant number of instructions that is independent of the value of k.

We now analyze the number of required iterations in dependency of n. Since the tree is perfect, each iteration of steps S1 through S4 reduces the search space by a factor of k (or terminates). For this reason, the algorithm performs $h = \lceil \log_k(n + 1) \rceil$ iterations in the worst case. Compared to binary search, the speedup of the worst case in terms of the number of iterations is $\lceil \log_2(n + 1) \rceil \,/\, \lceil \log_k(n + 1) \rceil \approx \log_2 k$ for large n. For example, when $k = 5$, the speedup is ≈ 2.32, that is, binary search requires more than twice as many iterations as k-ary search. For our running example with $n = 26$, the worst-case number of iterations is 3 for k-ary search, 4 for SIMDized binary search, and 5 for plain binary search.

Table 1 lists the worst-case number of iterations performed by binary search (Bin), SIMDized binary search (Bin[$k-1$]) and k-ary search for various dataset sizes and values of k. Clearly, k-ary search is the more attractive the larger the dataset and the larger the value of k. But even for small datasets (e.g., a page of a B-tree) and $k = 5$, it requires significantly less iterations than its competitors. Future generations of processors will support much larger values of k and thus provide further efficiency gains. For example, Intel recently announced that its upcoming processors will support the AVX instruction set [6] with 256-bit vector registers ($k = 9$ for 32-bit keys) for the 2010 processor generation and up to 512-bit vector registers ($k = 17$) for later generations. Similarly, the upcoming Larrabee GPGPU [9]—a hybrid between a GPU and a multi-core CPU—provides 16-way vector registers ($k = 17$) for integer, single-precision float, and double-precision float instructions.

3.2 On a Sorted Array

We now show how to execute steps S1 and S4 of the above search algorithm on a sorted array A; steps S2 and S3 remain unmodified. As for binary search, we make use of two variables l (initialized to 1) and r (initialized to n) that represent the left and right end of the remaining search space. Then, step S1 takes l and r as input and loads the separators into R. Step S4 takes m as an additional input and updates l and r to point to the next partition (given m).

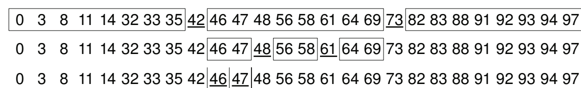


Figure 4: k-ary search for key 46 with $n = 26$, $k = 3$

Step S1 consists of two parts: (a) calculate the indexes of the separators and (b) load them into R. Substep (a) is required because the $k - 1$ separators are stored in non-contiguous memory locations, see Figure 4.[4] With $p = r - l + 1$ denoting the number of elements in the current partition, we find that the separators are stored at positions $l + i \lceil (p + 1)/k \rceil$ for $1 \leq i < k$. We can compute all these positions simultaneously using element-wise SIMD instructions. On some architectures, the computation of the distance $\lceil (p + 1)/k \rceil$ between consecutive separators is expensive because it requires a division. For these architectures, we suggest to precompute separator distances and to store them in a table. The table will be small because there are only $\lceil \log_k(n + 1) \rceil$ possible values for p, one for each iteration. Substep (b) cannot be SIMDized on current processor platforms, see the discussion below. For this reason, we perform a (unrolled) loop with $k - 1$ iterations to load the separators into register R.

Step S4 can be implemented by reusing the separator distances from step S1. We set $l \leftarrow l + 1 + m \lceil (p + 1)/k \rceil$ and $r \leftarrow \min(l - 1 + \lceil (p + 1)/k \rceil, r - 1)$, where as before $0 \leq m < k$ denotes the index of the next partition.

The above algorithm has one major disadvantage when compared to k-ary search on a tree: It requires $k - 1$ non-contiguous memory accesses *executed sequentially* in the load step of each iteration. This totals to $(k - 1) \lceil \log_k(n + 1) \rceil$ such memory accesses in the worst case; this is roughly $(k - 1)/\log_2 k$ more than required by binary search. Processor architectures with gather-and-scatter instruction support alleviate this problem by allowing $k - 1$ *parallel loads* from non-contiguous addresses; the upcoming Larrabee [9] will provide such instructions. For processor architectures without such functionality, the performance of k-ary search is likely to deteriorate as k becomes large. For $k = 5$, however, our experiments suggest that the savings in the number of iterations outweigh the additional cost for non-contiguous memory accesses.

3.3 On a Linearized k-ary Search Tree

In applications where reordering of the data is acceptable, an alternative approach that avoids the aforementioned problems of non-contiguous memory accesses is to use a representation of the data that is more amenable to k-ary search. The key idea is to change the data layout so that the separators in each step are located in contiguous memory locations. Conceptually, this can be achieved by linearizing a

[4]The separators may be stored in contiguous locations in the final iteration.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

42 73 8 32 48 61 88 93 0 3 11 14 33 35 46 47 56 58 64 69 82 83 91 92 94 97
42 73 8 32 48 61 88 93 0 3 11 14 33 35 46 47 56 58 64 69 82 83 91 92 94 97
42 73 8 32 48 61 88 93 0 3 11 14 33 35 46 47 56 58 64 69 82 83 91 92 94 97

Figure 5: k-ary search for key 46 on a linearized tree with $n = 26$, $k = 3$

k-ary search tree in level-order. Applying this idea to the search tree shown in Figure 3, we obtain the array shown in Figure 5. In the appendix, we outline some algorithms for performing this reordering without constructing the search tree and for traversing the linearized representation in sorted order.

To execute k-ary search on the linearized tree, we make use of a single variable l that points to the memory location that contains the separators of the current partition. Initially, l is set to 1. The implementation of steps S1 and S4 is straightforward. Step S1 loads $k-1$ keys starting at index l into register R. In contrast to loading from sorted arrays, this step can be executed in a single instruction. As for step S4, we set $l \leftarrow lk + m(k-1)$.

Both the average and worst-case number of iterations of k-ary search on a sorted array and k-ary search on a linearized tree are identical. Working on a linearized tree has the advantage that steps S1 and S4 become both simpler and more efficient.

3.4 Non-perfect Trees

The discussion thus far has assumed that $n = k^h - 1$ for some integer $h > 0$ so that the (conceptual) search tree is perfect. For other values of n, a perfect k-ary search tree cannot be constructed. Instead, we (again conceptually) construct a *complete tree*. Informally, a tree of height $h = \lceil \log_k(n+1) \rceil$ is complete if (1) removing the leaves at depth $h-1$ yields a perfect tree of height $h-1$ and (2) the leaves at depth $h-1$ grow from left to right. A more formal definition is given in the appendix. Complete trees are well-suited to k-ary search mainly because their level-order linearization does not exhibit any "holes."

The previous algorithms have been presented in such a way that they can be used almost directly with values of n that correspond to complete trees. For search on a linearized tree, the only difference is that the search terminates as soon as $l > n$. For search on a sorted array, we slightly change the way the dataset is partitioned, i.e., we change the table that contains the precomputed distances p between separators. For the first iteration, we store the sizes of all k partitions; the leftmost and rightmost partition each contain $k^{h-1} - 1$ keys (as in the perfect tree) and the remaining keys are equally distributed between the $k-2$ inner partitions. For all subsequent iterations, we reuse the separator distances for perfect trees ($p = k^{h-i} - 1$ for the ith iteration). For this reason, some keys might be checked twice, but the search is still correct and guaranteed to terminate after at most h iterations.[5]

4. EXPERIMENTS

We now present the results of our experimental study on several architectures.

4.1 Experimental Setup

We implemented binary search (Bin), binary search with SIMD instructions (Bin[k-1]), k-ary search on a sorted array (k-ary), and k-ary search on a linearized search tree (k-ary-lt) on different processors including an IBM Power Processor Element (PPE) and a Synergistic Processing Element (SPE) of the Cell Broadband Engine, an Intel Core i7, and an AMD Phenom. All processors provide 128-bit vector registers and support for the required SIMD instructions, e.g., element-wise summation, element-wise multiplication, and replication of a scalar.

All algorithms were implemented in C using C-intrinsics for SIMD instructions. The complete source code of our implementations is available online [3]. Our experiments were run on Linux and, on all platforms, compiled using GCC. Although other compilers such as Intel's ICC or IBM's cell compiler may produce better code on the respective platforms, our goal was to use a single compiler available on all platforms to facilitate comparison across platforms.

We run our experiments with both 32-bit integer keys and 64-bit floating point keys so that we can store 4 keys ($k = 5$) and 2 keys ($k = 3$) in a single register, respectively. We generated a synthetic dataset (the actual values do not affect performance) and measured the number of cycles required to find a given search key. We varied the number of keys from 2^7 keys (512/1024 Byte for 32/64 bit keys) up to 2^{25} keys (128/256 MByte for 32/64 bit keys). Each experiment is repeated 4096 times—using search keys chosen uniformly and at random from the set of all search keys—and results are averaged.

All of our plots have the same layout: The number of keys is shown in $\log_2$ scale on the x-axis, and the y-axis shows the average number of cycles.

4.2 IBM Cell BE: PPE

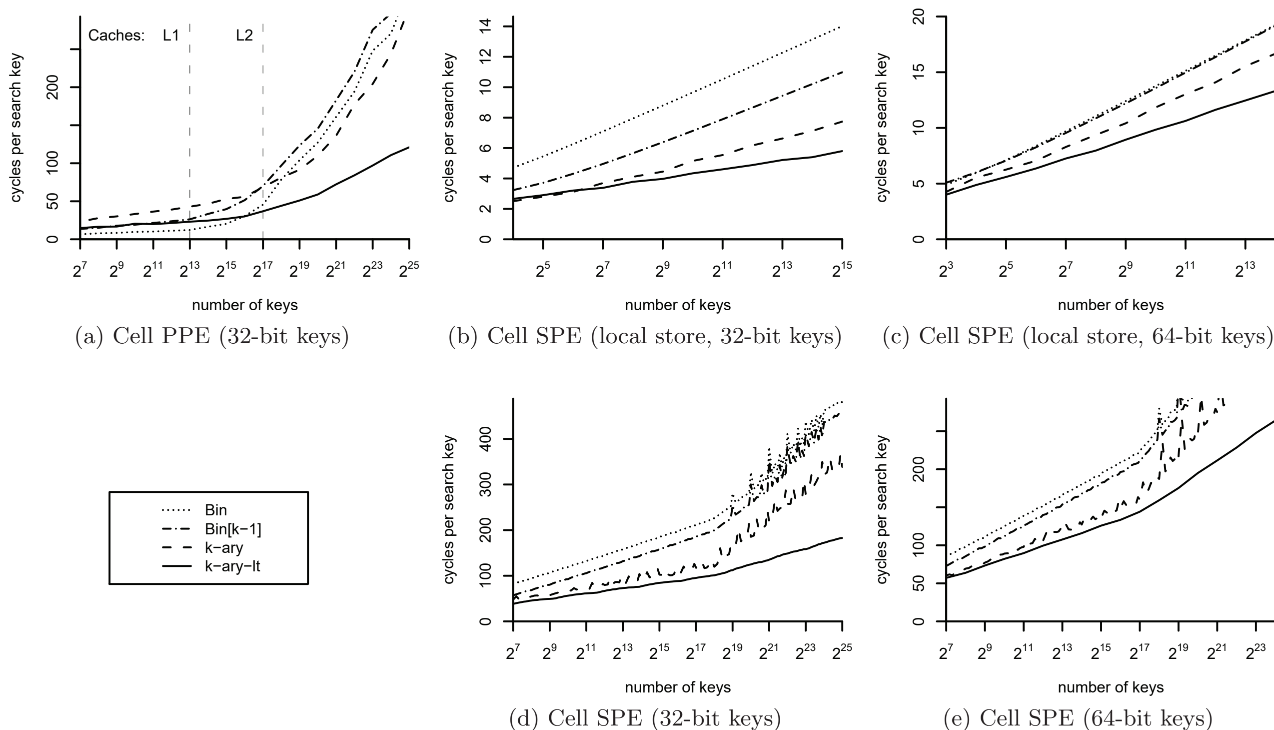
The first experiment was conducted on a PPE core of the Cell Broadband Engine in Sony's Playstation3. The processor is comparable to the PowerPC-970 architecture and includes an "AltiVec unit", which provides all of the SIMD instructions [1, 8] required for k-ary search. It provides 32 KByte instruction cache, 32 KByte L1 cache and 512 KByte L2 cache. Since the PPE has no SIMD instructions for 64-bit scalars, we restricted our experiments to 32-bit integer keys and $k = 5$.

Figure 6a shows the results of our PPE experiments. For small datasets—up to 2^{16} keys—Bin performs best. The reason is that the 128-bit AltiVec unit with its vector registers operates concurrently with the scalar integer/floating-point registers and there is no way to directly move data between both types of registers. Therefore, scalar replication and the horizontal-sum instructions are expensive on the PowerPC platform. As a consequence, the benefit of fewer iterations (and therefore fewer instructions) of Bin[k-1], k-ary, and k-ary-lt is outweighed by the cost of the aforementioned

[5] The additional comparisons could be avoided by storing $3h$ possible values of p for each iteration. To see this, observe that the children of an arbitrary node at depth d are roots of either (i) a perfect tree of height $h-d-1$, (ii) a complete tree of height $h-d-1$, or (iii) a perfect tree of height $h-d-2$. One can show that (ii) applies to at most one node per level. Together with the fact that the number of elements in a perfect tree is completely determined by its height, we conclude that there are at most three different partition sizes at each level. We chose the approach mentioned in the text over this one because the former is much simpler to implement.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705



(a) Cell PPE (32-bit keys) (b) Cell SPE (local store, 32-bit keys) (c) Cell SPE (local store, 64-bit keys)

(d) Cell SPE (32-bit keys) (e) Cell SPE (64-bit keys)

Figure 6: Overview of Cell processor's execution times (lower is better)

instructions. For datasets with more than 2^{16} keys, we find that the situation remains unchanged for Bin[k-1]; this is because it provides a constant reduction in the number of iterations. In contrast, the k-ary search algorithms provide a relative reduction: They first become competitive and then, as the dataset size hits the L2 cache size, significantly outperform Bin. At that point, k-ary-lt provides better scalability than each of the alternative algorithms (2–2.5 times faster than Bin for keys $\geq 2^{20}$). This is because k-ary-lt requires only a single memory access per iteration. To conclude, none of the SIMDized algorithms benefit directly from the SIMD capabilities of the PowerPC-970 processor but may provide performance improvements due to better memory access patterns.

4.3 IBM Cell BE: SPE

Our next set of experiments was performed on an SPE of a Cell BE processor of Sony's Playstation3. The architecture of an SPE is different to current x86 architectures. The SPE is a pure SIMD processor, i.e., it supports SIMD instructions only. Each instruction requires 128 bits regardless of whether it applies to two vectors or two scalars. Interestingly, scalar instructions require additional shuffle and mask operations and are thus less efficient than vector instructions. The memory model of an SPE is also different: Instead of caches, each SPE provides a local store of 256 KBytes of memory, which is populated by the application developer using asynchronous DMA requests. Finally, the SPE does support branch hints but does not provide branch prediction. For this reason, branch-heavy code with a large number of branch misses suffers from a significant loss in performance.

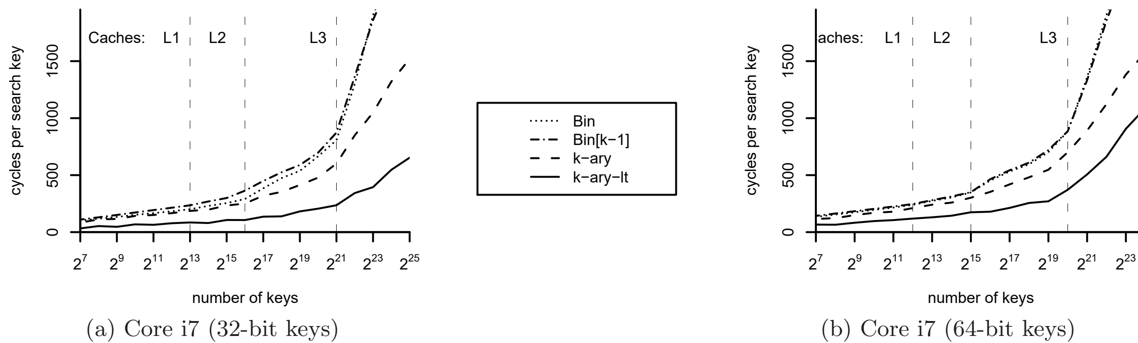
We run two different sets of experiments on the SPE. The first set of experiments make use of only the local store, whereas the second set of experiments run in main memory.

For our local-store experiments, we used up to 192 KByte of memory for the keys (up to 49,152 keys); the rest of the memory is used for instructions and stack variables. The results for 32-bit integer and 64-bit floating point keys are shown in Figures 6b and 6c, respectively. This time, Bin performs worst due to its high number of iterations and branches. The Bin[k-1] algorithm performs better for 32-bit keys but performs similar to Bin for 64-bit keys. This is because, for $k = 3$, the reduction in number of iterations is insignificant. Compared to k-ary and k-ary-lt, the Bin[k-1] algorithm requires more iterations and therefore cycles. One reason for this behavior is that Bin[k-1] relies on an (unpredictable) branch to determine the next partition. In contrast to this, k-ary and k-ary-lt convert this control dependency into a data dependency and thus completely avoid it. Furthermore, and more importantly for large datasets, both k-ary and k-ary-lt provide better scalability. Between these two algorithms, k-ary-lt is superior because it requires four times fewer memory accesses than k-ary .

Figures 6d and 6e show the execution times of the main-memory experiments. The results are very similar to the results for the local store, the main difference being the occurence of some spikes. These spikes result from non-aligned DMA accesses to memory; this can be potentially improved if the implementation makes sure that the source and destination addresses of each memory transfer have the same quadword offset within a 128-byte cache line.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

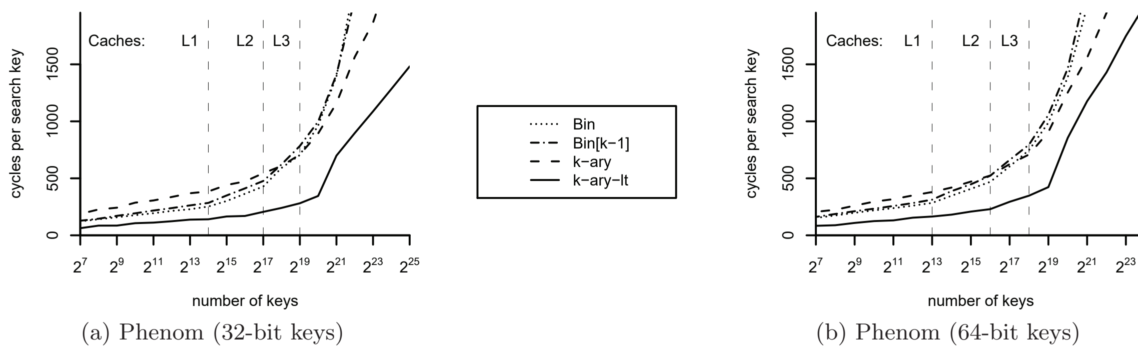
Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705



(a) Core i7 (32-bit keys)

(b) Core i7 (64-bit keys)

Figure 7: Overview of Core i7 processor's execution times (lower is better)



(a) Phenom (32-bit keys)

(b) Phenom (64-bit keys)

Figure 8: Overview of Phenom processor's execution times (lower is better)

4.4 Intel Core i7

The next set of experiments was performed on a 2.66 GHz Intel Core i7 920 processor. This processor supports the SSE, SSE2, SSE3 and SSE4.2 instruction sets, which can be efficiently exploited for k-ary search.

The results of the Core i7 experiments are shown in Figures 7a and 7b. Similar to the PowerPC results, the execution times grow almost linearly (log axis) until the memory required for storing the keys exceeds the L3 cache size (8192KB). The L1 cache (32KB) and L2 cache (256KB) have only a small effect on the execution times. For all key sizes, k-ary and k-ary-lt perform better than Bin and Bin[k-1]. The k-ary algorithm realizes a speedup of 1.2 (for 2^7 keys) to 1.8 (2^{25} keys) for both key sizes while k-ary-lt realizes a speedup of 3 up to 4.5 for 32-bit keys and 2 to 2.5 for 64-bit keys.

4.5 AMD Phenom

Our final experiments were performed on a 2.8 GHz AMD Phenom 920. The AMD Phenom supports similar SIMD instructions sets as the Intel Core i7 processor but implements the SSE4A instead of the SSE4.2 instruction set.

Figures 8a and 8b show the results for the Phenom processor. Again, passing the L1/L2 cache size border has only minor effects, but exceeding the L3 cache size (6144KB) has a strong impact on the algorithms for both key sizes. In fact, k-ary is advantageous only for large dataset (1.5 times faster than Bin for 2^{25} keys for both key sizes). The k-ary-lt algorithm is the overall method of choice (2–3 times faster than Bin for 32-bit keys and 1.5–2 times faster for 64-bit keys).

5. CONCLUSIONS

Current sort-based search algorithms do not utilize efficiently the SIMD capabilities of modern processors. In this paper, we presented k-ary search algorithms that improve upon binary search by reducing the search space in each iteration by a factor of $k > 2$, thereby exploiting the data parallelism provided by SIMD architectures. We provided two different k-ary search algorithms, one working on sorted arrays and one using a linearized k-ary search tree representation. Our experiments have shown that k-ary search can provide significant performance benefits over binary search algorithms on most architectures. To leverage these benefits, k-ary search should be implemented in the standard libraries provided for these architectures so that it can be exploited by a large set of applications.

6. REFERENCES

[1] *Cell Broadband Engine Programming Handbook*.

[2] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, 2008.

[3] Code available at: wwwdb.inf.tu-dresden.de/schlegel.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

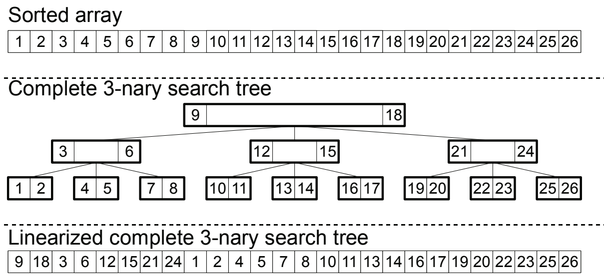


Figure 9: The three representations for $n = 26$, $k = 3$

[4] B. Gedik, R. R. Bordawekar, and P. S. Yu. Cellsort: high performance sorting on the cell processor. In *VLDB*, pages 1286–1297. VLDB Endowment, 2007.

[5] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *DAMON*, pages 1–6, New York, NY, USA, 2007. ACM.

[6] Intel Corporation. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, March 2008.

[7] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[8] *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*.

[9] R. Ronny Ronen. Larrabee: a many-core intel®architecture for visual computing. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 225–225, New York, NY, USA, 2009. ACM.

[10] K. A. Ross. Conjunctive selection conditions in main memory. In *Symposium on Principles of Database Systems*, pages 109–120, 2002.

[11] K. A. Ross. Efficient hash probes on modern processors. *ICDE*, 0:1297–1301, 2007.

[12] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, pages 145–156, New York, NY, USA, 2002. ACM.

APPENDIX

In the paper, we deal with three different representations of the dataset: a sorted-array representation, a search-tree representation, and a linearized-tree representation. In this section, we describe how to convert sorted-array indexes to linearized-tree indexes and vice versa. This conversion is useful to both derive the linearized-tree representation of a dataset without actually building the tree and to perform an ordered scan in the linearized-tree representation.

A. CONVERSIONS FOR PERFECT TREES

We start with a discussion of perfect trees such as the one shown in Figure 9. We then discuss extensions to the class of "complete" trees. We assume throughout that indexes are 1-based. For example, the sorted-array entry 9 in Figure 9 has index 1 in the linearized tree, and the linearized-tree entry at index 4 has index 6 in the array representation.

In what follows, we consistently denote positions in the sorted array by index i and positions in the linearized tree by index j. Thus, our goal is to determine i given j and vice versa. We mark functions that operate on linearized-tree indexes with a star ($*$); functions without such a star operate on sorted-array indexes. Set $H = \lceil \log_k(n+1) \rceil$ to the height of the tree. The depth (counting from 0) of the node containing the sorted-array entry at index i in the search tree is given by

$$d_h(i) = \sum_{x=1}^{h-1} \operatorname{sgn}(i \bmod k^{h-x}) \tag{1}$$

where sgn denotes the signum function and h is set to H. To derive (1), we made use of the fact that $\operatorname{sgn}(i \bmod k^{h-x})$ equals 1 whenever index i resides at depth x or below, and 0 otherwise. For example, in Figure 10, we have $d_3(9) = 0 + 0 = 0$ and $d_3(10) = 1 + 1 = 2$. Computation of d_h takes $O(\log n)$ time, but $d_h(i)$ can be tracked while traversing the tree so that usage of (1) is rarely required (see Appendix B). The offset (0-based) of the entry at index i in its level is given by

$$o_h(i) = \left\lfloor \frac{k-1}{k} \cdot \frac{i}{k^{h-d_h(i)-1}} \right\rfloor, \tag{2}$$

which follows from the observation that the sorted-array indexes at depth $d_h(i)$ are given by $ak^{h-d_h(i)-1}$ for $a = 1, 2, \ldots$ and $a \bmod k \neq 0$—indexes with $a \bmod k = 0$ belong to one of the higher levels. E.g., $o_3(9) = \lfloor 2/3 \cdot 9/3^2 \rfloor = 0$ and $o_3(10) = \lfloor 2/3 \cdot 10/3^0 \rfloor = 6$. To determine the value of j that corresponds to index i, set j to

$$f_h(i) = k^{d_h(i)} + o_h(i),$$

where $k^{d_h(i)}$ denotes the position of the first entry at depth $d_h(i)$ in the linearized search tree. Again, the computation of $o_h(i)$ can be done more efficiently during a tree traversal (Appendix B).

Using similar arguments as above, we find that the tree depth, offset and sorted-array index of the linearized-tree entry at index j are given by

$$d_h^*(j) = \lfloor \log_k j \rfloor,$$
$$o_h^*(j) = j - k^{d_h^*(j)},$$

and

$$f_h^*(j) = k^{h-d_h^*(j)-1} \left\lfloor \frac{k}{k-1} o_h^*(j) + 1 \right\rfloor,$$

respectively.

B. RANGE SCANS

In practice, it is often useful to iterate through a range of subsequent keys. To do this, we require a function that derives the index of the next-largest key given the index of the current key. Clearly, for the sorted-array representation, the desired index is given by $n_h(i) = i+1$. For the linearized-tree representation, we may use a double-conversion to obtain

$$n_h^*(j) = f_h\left[f_h^*(j) + 1\right]. \tag{3}$$

When performed directly, this approach is not very efficient. In the rest of this section, we outline a more efficient approach that maintains small data structures in order to compute $n_h^*(j)$ directly.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware. Providence, 2009". S. 62-60. ISBN 978-1-60558-701-1
https://doi.org/10.1145/1565694.1565705

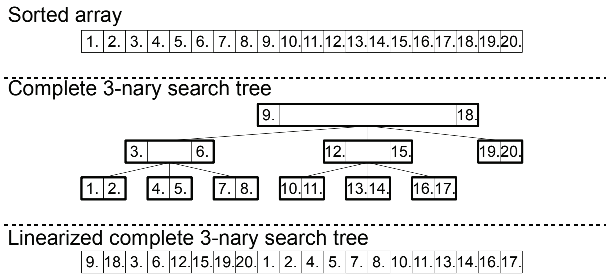


Figure 10: The three representations for $n = 20$, $k = 3$

First, observe that when performing a range scan, the arguments to f_h in (3) are a sequence of increasing sorted-array indexes—i.e., $f_h^*(n^*(j)) = f_h^*(j)+1$—and can thus be easily computed. For the computation of f_h our main goal is to avoid the summation and modulo computations of (1) and the divisions of (2). To do so, we maintain the digits of the base-k representation $B(i)$ of $i-1$, where $i = f_h^*(j)$ is the current position in the sorted array. For example, we have $B(9) = [0,2,2]$ and $B(10) = [1,0,0]$. When read from left to right, the base-k digits can be interpreted as the path taken from the root of the tree to the position of node i, cf. Figure 9. Trailing digits of value $k-1$ denote "unused digits" and the number of those digits is equal to $h-d_h(i)-1$. Since $B(i+1)$ can easily be computed from $B(i)$ (see step 1 below), this gives us an efficient way to compute $d_h(i+1)$. A similar trick can be used to compute $o_h(i+1)$. The idea is to store in an array $O(i)$ the next offset to be used in each level. E.g., $O(9) = [1,2,6]$ and $O(10) = [1,2,7]$. The following simple algorithm updates arrays $B = B(i)$ and $O = O(i)$ to values $B(i+1)$ and $O(i+1)$:

1. Set d to $H-1$, the maximum depth in the tree. Then, as long as $B[d] = k-1$, set $B[d]$ to 0 and decrement d. After this step has been completed, we have $d_h(i+1) = d$ and $o_h(i+1) = O[d]$.
2. Increment both $B[d]$ and $O[d]$.

The algorithm is efficient to compute and allows for quick range scans on the linearized tree representation.

C. CONVERSIONS FOR COMPLETE TREES

We now proceed to complete trees, see Figure 10. Formally, any search tree of height $H = 1$ is complete. For $H > 1$, a search tree is complete if there exist numbers l and r with $0 \le l < k$, $0 \le r < k$, and $l+1+r = k$ such that:

1. The subtrees formed by the first l successors of the root node are perfect and have height $H-1$.
2. The subtree formed by the next successor is complete and has height $H-1$ or $H-2$.
3. The subtrees formed by the remaining r successors are perfect and have height $H-2$.

To simplify exposition, we refer to the largest entry (the rightmost entry) at depth $H-1$ as the *fringe entry*. In Figure 10, the fringe entry is 17.

The fringe entry plays a central role for index conversion. To see this, suppose that we construct a hypothetical perfect tree by filling the lowest level with dummy entries (all larger than the fringe entry). Then, all entries that are smaller than the fringe entry (1–17) have the same position in the linearized representations of both the complete tree and the hypothetical tree. This is because the dummy entries do not "fall in between" these entries. Since all the remaining entries (18–20) are at depths smaller than $H-1$, their position in the linearized complete tree is identical to the position they would have in the linearized perfect tree obtained by omitting all the leaves at depth $H-1$. We can use these observations to construct the conversion functions for complete trees. Making use of the fact that the sorted-array index of the fringe entry is given by $f_H^*(n)$, we find

$$g_n(i) = \begin{cases} f_H(i) & i \le f_H^*(n) \\ f_{H-1}(i - o_H^*(n) - 1) & \text{otherwise} \end{cases}$$

and using similar arguments

$$g_n^*(j) = \begin{cases} f_H^*(j) & f_H^*(j) \le f_H^*(n) \\ f_{H-1}^*(j) + o_H^*(n) + 1 & \text{otherwise.} \end{cases}$$

Our algorithm for efficient range scans (Appendix B) can be applied to perfect trees almost as is. The only change that is required is to remove the entries at position $H-1$ from both B and O as soon as the fringe entry has been processed.

D. TREE CONSTRUCTION

Given the conversion functions, there are two simple methods to directly create the linearized representation of the search tree. The methods differ in the amount of memory and CPU consumption. Both methods take as input a sorted-array representation of the dataset.

The naive method makes use of a temporary array of N elements, which eventually holds the linearized-tree. The algorithm iterates over the dataset and copies each entry to its respective position in the linearized-tree, as determined by the g_n function. (The optimizations of Appendix B apply.)

An alternative method would construct the linearized tree in place. The idea is to treat g_n as a permutation and make use of the cycles in the permutation. We start by moving the item at index 1 to position $g_n(1)$, then the former entry at position $g_n(1)$ is moved to position $g_n(g_n(1))$, and so on. The process stops when index 1 has been overwritten. Next, the algorithm would process index 2, then index 3, and so on. To make sure that every index is moved exactly once, so that correctness is established, we check whether index i has been processed before. This check can be executed by computing $g_n(i), g_n(g_n(i)), \ldots, i$. If the cycle contains an index smaller than i, we know that i has been processed already. Otherwise, i has not been processed. The time complexity of this algorithm is $\mathrm{O}(n^2)$ in the worst case.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden