WHAT ARE THE BASIC PRINCIPLES FOR SORTING AND SEARCHING?

A Andersson

Rapporteur: Dr Robert Stroud

WHAT ARE THE BASIC PRINCIPLES FOR SORTING AND SEARCHING?

Arne Andersson
Department of Computer Science
Lund University
Box 118, S–221 00 Lund, Sweden
arne@dna.lth.se

Abstract

In today's curricula for basic algorithms and data structures, comparison-based algorithms play by far the most central role. Algorithms based on other methods, like radix sorting and trie structures, are often overlooked or marginalised. However, some recent research findings give reason to question this situation. This article contains an overview of these results. Some experimental experiences are also discussed.

1. Introduction

What is a reasonable model for developing and analysing algorithms for sorting and searching? If we simplify the question and consider only sequential computers, there are two major alternatives:

- The keys are regarded as atomic items which can be compared at a constant cost. However, this *comparison-based model of computation* does not always reflect reality since it excludes many useful operations on data, like shifting, indirect addressing etc. Furthermore, long keys, occupying many machine words, can not be compared in constant time.

- The keys are regarded as binary strings, each one contained in one or more words (registers). In this case, the *word length* becomes a natural part of the model. A comparison does not necessarily take constant time, on the other hand we may use a larger variety of operations on data. This model allows for comparison-based algorithms to be used but also for algorithms like tries, bucket sort, radix sort etc, which are based on indirect addressing (i.e. the keys, or parts of the keys, are used as addresses in arrays).

The second model, sometimes referred to as the *RAM model* [1], reflects real computers more accurately, and algorithms like radix sort are known to perform very well in many applications. Nevertheless, it is a common belief that comparison-based algorithms are the best general choice; this belief is reflected in today's textbooks for basic algorithms and data structures. Algorithms based on indirect addressing are considered to be efficient only in the special case when the keys come from a small universe or when the word length of the used computer is small.

This general view may be summarised in the following rule of thumb for chosing algorithms:

> *Use indirect addressing if the word length is small.*
> *Use comparisons if the word length is large.*

However, some recent findings give reason to question this rule. In particular, it has been shown that algorithms based on indirect addressing can be combined with *packed computation* (to be discussed below) to suit large word sizes. Therefore, we would like to suggest the following rule of thumb:

> *Use indirect addressing if the word length is small.*
> *Use indirect addressing and packed computation if the word length is large.*
> *Use comparisons in special cases.*

[1] Note that the term RAM is used for many models. There are also RAM models with infinite word length.

2. Model of computation

We use a unit-cost RAM with word size w and we assume that the n keys to be sorted or searched for are w-bit keys that can be treated as binary strings or integers. Thus, our sorting and searching problems has *two* parameters, the number of keys and the word size. The goal is to find algorithms that are efficient for all possible combinations of n and w. In cases where the complexity is expressed only in terms of n, it is supposed to hold for any possible value of w, and vice versa.

(The fact that we do not consider long strings may seem like a restriction, but long strings give no advantage to comparison-based algorithms. In fact, long strings are better handled by tries and radix sorting than by comparisons, see the example at the end of this section.)

In this model, we can use other operations than comparisons, for instance indirect addressing, shifting and bitwise logical operations. Without loss of generality, we assume that $w = \Omega(\log n)$. This assumption is indeed modest; if it does not hold, the number n (or a pointer) would not fit in a constant number of words.

The assumption that the keys can be treated as binary strings or integers is realistic, also for real numbers (*cf.* IEEE 754 floating-point standard [17, p. 228]). In fact, it is not easy to find a reasonable problem specification where this assumption does not hold while comparisons can be computed in constant time. It seems much easier to come up with a problem where the comparison-based model is questionable. Consider the following example: We wish to sort n keys, where each key occupies L words. In this case, a single comparison may take as much as $\Omega(L)$ time. Hence, the assumption of constant-time comparisons would not hold; sorting these keys with heapsort, quicksort, or mergesort would take $\Theta(Ln \log n)$ time. If, on the other hand, indirect addressing is used, the problem of sorting n strings of L words each can be reduced to the problem of sorting n words in $O(nL)$ time [6]; this reduction time is optimal since we must read the input at least once.

3. Overview

The basic purpose of this article is to briefly discuss some new theoretical results:

- First, we discuss deterministic data structures and algorithms that use superlinear space
  - tries are used to decrease key length;
  - the resulting short keys are handled efficiently by packing them tightly.

  With these methods, the worst-case cost of sorting is $O(n \log\log n)$ and the worst-case cost of searching is $O(\sqrt{\log n})$. (Tries use superlinear space, but with randomisation, i.e. hash coding, the space can be reduced to linear.)
- Next, we discuss deterministic data structures and algorithms that use linear space
  - the fusion tree; with this data structure the worst-case cost of sorting is $O(n \log n / \log\log n)$ and the amortised cost of searching is $O(\log n / \log\log n)$.
  - a new data structure, the *exponential search tree*; the bounds on deterministic sorting and searching in linear space are improved to $O(n\sqrt{\log n})$ and $O(\sqrt{\log n})$, respectively.
- Finally, we mention some practical experiences, as well as some simple, practical algorithms.

4. Simple algorithms which use superlinear space

In this section, we study some rather simple algorithms that allow sorting and searching asymptotically faster than comparison-based algorithms. We use indirect addressing and large arrays. As a consequence, these algorithms will need much space. However, all algorithms presented here can be fit into linear space with randomisation (i.e. with universal hashing [13]).

The word size of our RAM, as well as the length of our keys, is w. In our algorithms, we will also consider shorter keys. In cases when the key length may differ from w we use b or k, to denote the key length.

Our lemmas will be expressed as problem reductions. For this purpose, we use the notation below. For the searching problem, we assume that an ordered set is maintained and that operations like range queries and neighbour queries are supported.

Definition 1 *A dictionary is ordered if neighbour queries and range queries are supported at the same cost as member queries, and if the keys can be reported in sorted order in linear time.*

Definition 2 *Let $F(n, b)$ be the worst-case cost of performing one search or update in an ordered dictionary storing n keys of length b.*

Definition 3 *Let $T(n, b)$ be the worst-case cost of sorting n keys of length b.*

The methods discussed in this section are fairly simple. Furthermore, they can all be implemented with a simple instruction set. All necessary instructions are standard, they are even in AC^0. (An instruction is in AC^0 if it is implementable by a constant depth, unbounded fan-in (AND,OR,NOT)-circuit of size $w^{O(1)}$. An example of a non-AC^0 instruction is multiplication [9].)

4.1. Tries and range reduction

One way to simplify a computational problem is by *range reduction*. In this case, we replace our b-bit keys with k-bits keys, $k < b$.

Lemma 1 $F(n, b) = O(b/k) + F(n, k)$.

Hint of proof: We store the keys in a trie of height b/k. At each node, k bits are used for branching, and each node contains an array of length 2^k. A trie is illustrated in Figure 1. In this example $b = 12$ and $k = 3$. The array entries that contain pointers to subtries are marked with filled boxes. As an example: the smallest (leftmost) key is $\boxed{000}\boxed{100}\boxed{000}\boxed{011}$.

When searching for a key x or x's nearest neighbour, we traverse a path down the trie. At each node we use k bits in x as an address in the node's array. If x is present, the search ends at a trie node. Otherwise, the trie traversal ends when trying to reach a non-existing subtrie. In this case, we have to locate the nearest existing subtrie. Once this subtrie is found, we can find x's nearest neighbour; we may for example assume that each subtrie contains a reference to its smallest and largest keys. When k is large, the array itself can not be used to locate the nearest existing subtrie, since we then may have to scan too many empty entries. Instead, we note that the array entries (subtries) represent a set of k-bit keys which can be stored in some suitable data structure. In this way, each node contains an array used for trie traversal plus a local data structure.

The cost of traversing the trie is proportional to the height of the trie, that is $O(b/k)$. After the trie traversal, one search is made in a local data structure; the cost of this is $F(n, k)$ (the local data structure can not contain more than n keys).

Updates are rather straightforward. □

For sorting, there is a similar lemma:

Lemma 2 $T(n, b) = O(nb/k) + T(n, k)$.

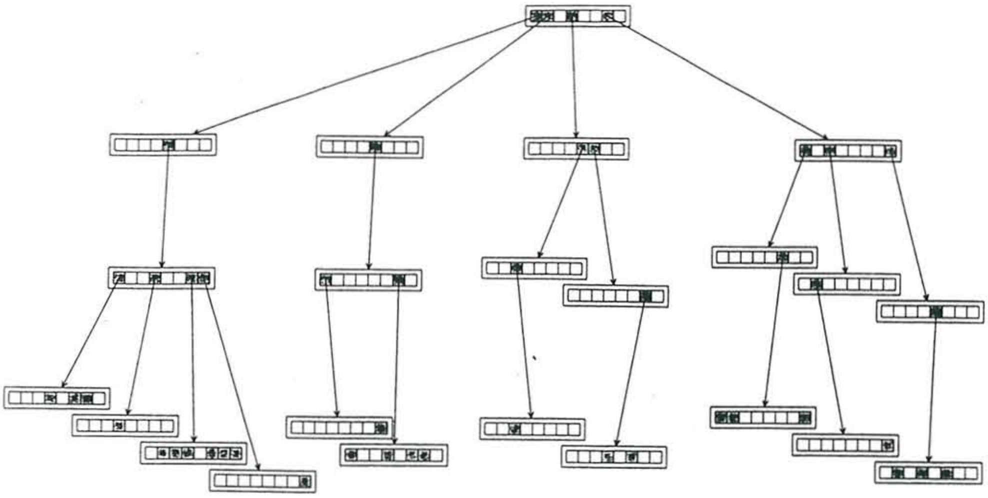


Figure 1: *A trie.*

Hint of proof: We use a trie as above, with some minor differences. In each node, we use an array as before, but instead of an additional local data structure we just keep an unsorted edge-list in each node telling which array entries are being used. Then, after all keys have been inserted into the trie in $O(nb/k)$ time, we tag each list element with the node it belongs to. We collect all lists into one list which is sorted in $T(n, k)$ time. After the sorting, we traverse the list and create a sorted edge-list at each node. Then, an inorder traversal of the trie according to the edge-lists gives the sorted sequence.

There is one problem with this approach: the number of list elements, corresponding to the number of edges in the trie, may be larger than n. To avoid this problem, we have to remove some edges from the lists. If we take away one element from each edge list, the total number of list elements will be less than n. In each edge list the removed edge can be inserted after the sorting, by just scanning the sorted list; the total cost for this is linear. In this way, the recursion equation of the lemma holds. □

Applying the lemmas above recursively, we can improve the complexity significantly. These reductions were first used in van Emde Boas trees [22, 23, 24] and in the sorting algorithm by Kirkpatrick and Reisch [18].

Lemma 3 $\left\{ \begin{array}{l} F(n,b) = O(1) + F(n, b/2). \\ T(n,b) = O(n) + T(n, b/2). \end{array} \right.$

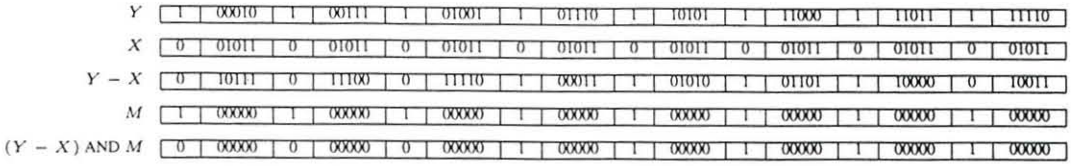
Hint of proof: Apply Lemmas 1 and 2, respectively, with $k = b/2$. □

Lemma 4 $\left\{ \begin{array}{l} F(n,b) = O(\log(b/k)) + F(n,k). \\ T(n,b) = O(n\log(b/k)) + T(n,k). \end{array} \right.$

Hint of proof: Apply Lemma 3 recursively $\log(b/k)$ times. □

4.2. Packing keys

If the word length is small enough—like in today's computers—the range reduction technique discussed above will decrease the key length to a constant at a low cost. However, in order to make a really convincing comparison between comparison-based algorithms and algorithms based on indirect addressing, we must make the complexity independent of the word size. This can be done by combining range reduction with *packed computation.* The basic idea behind packed computation is to exploit the parallelism in a computer; many short keys can be packed in a word and treated simultaneously.



Y	1	00010	1	00111	1	01001	1	01110	1	10101	1	11000	1	11011	1	11110
X	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011
$Y - X$	0	10111	0	11100	0	11110	1	00011	1	01010	1	01101	1	10000	0	10011
M	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000
$(Y - X)$ AND M	0	00000	0	00000	0	00000	1	00000	1	00000	1	00000	1	00000	1	00000

Figure 2: A multiple comparison in a packed B-tree

The central observation is due to Paul and Simon [21]; they observed that one subtraction can be used to perform comparisons in parallel. Assume that the keys are of length k. We may then pack $\Theta(w/k)$ keys in a word in the following way: Each key is represented by a $(k+1)$-bit field. The first (leftmost) bit is a *test bit* and the following bits contain the key. Let X and Y be two words containing the same number of packed keys, all test bits in X are 0 and all test bits in Y are 1. Let M be a fixed mask in which all test bits are 1 and all other bits are 0. Let

$$R \leftarrow (Y - X) \text{ AND } M. \tag{1}$$

Then, the ith test bit in R will be 1 if and only if $y_i > x_i$. All other test bits, as well as all other bits, in R will be 0.

4.3. Packed searching

Lemma 5 $F(n,k) = O\left(\log(w/k) + \frac{\log n}{\log(w/k)}\right)$.

Hint of proof: We use a packed B-tree [2]. The essential operation in this data structure is a multiple comparison as in Expression 1 where the keys in Y are sorted left-to-right and X contains multiple copies of one key x. Then, the rightmost p test bits in R will be 1 if and only if there are p keys in Y which are greater than x. This is illustrated in Figure 2. Hence, by finding the position of the leftmost 1-bit in R we can compute the rank of x among the keys in Y. (Finding the leftmost bit can either be done in constant time with multiplication [15] or via a lookup table [2].)

We use this technique to implement a B-tree with nodes of degree $\Theta(w/k)$. When searching for a k-bit key x in a packed B-tree, we first construct a word X containing multiple copies of x. X is created by a simple doubling technique: Starting with a word containing x in the rightmost part, we copy the word, shift the copy $k+1$ steps and unite the words with a bitwise OR. The resulting word is copied, shifted $2k+2$ steps and united, etc. Altogether X is generated in $O(\log(w/k))$ time.

After the word X has been constructed, we traverse the tree. At each node, we compute the rank of x in constant time with a multiple comparison. The cost of the traversal is proportional to the height of the tree, which is $O(\log n / \log(w/k))$ □

Above, we omitted a lot of details, such as how to perform updates and how pointers within the tree are represented.

4.4. Packed sorting

Above, we compared a query key with all keys in a B-tree node in order to determine the rank of the key. This seems to be a waste of comparisons; a binary search would use less comparisons. Hence, there seems to be room for improvement. For searching, no such improvement is known, but the multiple comparisons can be utilised more efficiently when sorting.

We need a lemma by Batcher [8]. A sequence is *bitonic* if it is the concatenation of a nondecreasing and a nonincreasing sequence, or if it can be obtained as the cyclic shift of such a concatenation.

Lemma 6 *Consider a bitonic sequence $x_1, x_2, \ldots, x_{2k}$ and the two subsequences*

$$L = \min(x_1, x_{k+1}), \min(x_2, x_{k+2}), \ldots, \min(x_k, x_{2k})$$

and

$$R = \max(x_1, x_{k+1}), \max(x_2, x_{k+2}), \ldots, \max(x_k, x_{2k}).$$

The sequences L and R are bitonic, and each element of L is smaller than or equal to each element of R.

The lemma suggests a parallel algorithm for sorting a bitonic sequence of length k, where k is a power of 2, and, in particular, an algorithm for merging two sorted sequences.

- If $k = 1$ halt.
- If $k > 1$ compare the corresponding elements of the left and right half in parallel and generate the sequences L and R, then sort these bitonic sequences in parallel.

Batcher's lemma is well suited for combination with the Paul-Simon technique, as shown by Albers and Hagerup [1].

Lemma 7 $T(n, w/\log n) \leq O(n \log\log n)$.

Hint of proof: We combine Lemma 6 (with $k = \log n$) with the Paul-Simon technique. We let a word X contain $x_1, x_2, \ldots, x_k$ and a word Y contain $x_{k+1}, x_{k+2}, \ldots, x_{2k}$. After a subtraction as in Expression 1, the test bits will tell which keys should belong to L and R respectively. By copying and shifting test bits, we create bit masks that allow us to extract L and R respectively. □

4.5. Combining

We can now derive our first bounds for sorting and searching. First, we state bounds in terms of w. The following bound holds for searching [22, 23, 24]:

Theorem 1 $F(n, w) = O(\log w)$.

Hint of proof: Apply Lemma 4 with $k = 1$. □

For sorting, there is a similar bound [18]:

Theorem 2 $T(n, w) = O(n \log(w/\log n))$.

Hint of proof: Apply Lemma 4 with $k = \log n$. Keys of length $\log n$ can be sorted in linear time with bucketsort. □

Next, we show how to remove the dependency of word length. For searching, we have [2]:

Theorem 3 $F(n, w) = O(\sqrt{\log n})$.

Hint of proof: If $\log w = O(\sqrt{\log n})$, Theorem 1 is sufficient. Otherwise, Lemma 4 with $k = w/2^{\sqrt{\log n}}$ gives $F(n, w) = O(\sqrt{\log n}) + F(n, w/2^{\sqrt{\log n}})$. Lemma 5 gives that $F(n, w/2^{\sqrt{\log n}}) = O(\sqrt{\log n})$. □

Finally, we get the following bound for sorting [4]:

Theorem 4 $T(n, w) = O(n \log\log n)$.

Hint of proof: If $\log w = O(\log \log n)$, Theorem 2 is sufficient. Otherwise, Lemma 4 with $k = w/\log n$ gives $T(n, w) = O(n \log \log n) + T(n, w/\log n)$. Lemma 7 gives the final bound. □

5. Deterministic algorithms and linear space

The first algorithm that surpassed the comparison-based algorithms independent of word size was the fusion tree [15]. Here, we present a short overview as well as a new, faster, data structure.

The data structures in this section are more complicated than the previous ones. They also need more powerful—but standard—instructions, like multiplication.

Definition 4 *Let $D(n)$ be the worst-case search cost and the amortised update cost in an ordered dictionary storing n keys in $O(n)$ space.*

5.1. Fusion trees

The central part of the fusion tree [15] is a static data structure with the following properties:

Lemma 8 *For any d, $d = O\left(w^{1/6}\right)$, a static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(1)$ worst-case time.*

Hint of proof: The main idea behind the fusion tree is to make use of significant bit positions. We illustrate the basic mechanism with an example, shown in Figure 3. In this small example we have chosen $w = 16$ and $d = 6$. Let Y be the set of keys $y_1, \ldots, y_d$. Assume that Y is stored in a binary trie. Each key is represented as a path down the trie. In the figure, a left edge denotes a 0 and a right edge denotes a 1. For example, y_3 is $\boxed{1010010101011010}$. We get the significant bit positions by selecting the levels in the trie where there is at least one binary (that is, non-unary) node. In this example the levels are 4, 9, 10, and 15, marked by horizontal lines. Create a set Y' of compressed keys $y'_1, \ldots, y'_d$ by extracting these bits from each key. In the example the compressed keys are $\boxed{0000}$, $\boxed{0001}$, $\boxed{0011}$, $\boxed{0110}$, $\boxed{1001}$, and $\boxed{1011}$. These compressed keys are used for packed searching. Since the trie has exactly d leaves, it contains exactly $d - 1$ binary nodes. Therefore, the number of significant bit positions, and the length of a compressed key, is at most $d - 1$. This implies that we can pack the d keys, including test bits, in d^2 bits. Since $d = O\left(w^{1/6}\right)$, the packed keys fit in a constant number of words.

(This extraction of bits is nontrivial; it can be done with multiplication and masking. However, the extraction is not as perfect as described here; in order to avoid problems with carry bits etc, we need to extract some more bits than just the significant ones. Here, we ignore these problems and assume that we can extract the desired bits properly. For details we refer to Fredam and Willard [15])

The d compressed keys may be used to determine the rank of a query key among the original d keys. Assume that we search for $x = \boxed{1010011001110100}$. First, we extract the proper bits to form a compressed key $x' = \boxed{0010}$. Then, we use packed searching to determine the rank of x' among $y'_1, \ldots, y'_d$.. In this case, the packed searching will place x' between y'_2 and y'_3. as indicated by the arrow in Figure 3(b). This is not the proper rank of the original key x, but nevertheless it is useful. The important information is obtained by finding the position of the first differing bit of x and one of the keys y_2 and y_3. In this example, the 7th bit is the first differing bit. and, since x has a 1 at this bit position, we can conclude that it is greater than all keys in Y with the same 6-bit prefix. Furthermore, the remaining bits in x are insignificant. Therefore, we can replace x by the key $\boxed{1010011111111111}$, where all the last bits are 1s. When compressed, this new key becomes $\boxed{0111}$. Making a second packed searching with this key instead, the proper rank will be found.

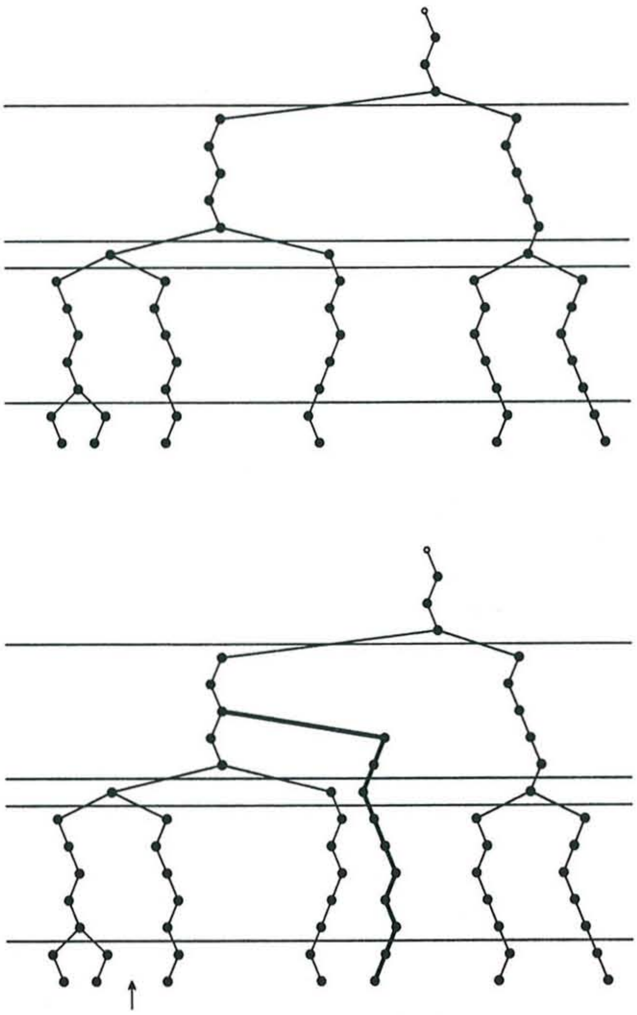


Figure 3: *(a) Selecting bit positions in a fusion tree node.*
(b) Determining the rank of a query key.

Hence, in constant time we can determine the rank of a query key among our d keys. □

The original method by Fredman and Willard is slightly different. Instead of filling the query keys with 1s (or 0s) and making a second packed searching, they use a large lookup table in each node. Fusion trees can be implemented without multiplication, using only AC^0 instructions, provided that some simple non-standard instructions are allowed. A discussion of these possibilities is currently in progress [5].

Theorem 5 $D(n) = O(\log n/ \log\log n)$.

Hint of proof: Based on Lemma 8, we use a B-tree where only the upper levels in the tree contain B-tree nodes, all having the same degree (within a constant factor). At the lower levels, traditional (i.e. comparison-based) weight-balanced trees are used. The reason for using weight-balanced trees is that the B-tree nodes are costly to reconstruct; the trees at the bottom ensure that few updates propagate to the upper levels. In this way, the amortised cost of updating a B-tree node is small.

The amortised cost of searches and updates is $O(\log n/ \log d + \log d)$ for any $d = O\left(w^{1/6}\right)$. The first term corresponds to the number of B-tree levels and the second term corresponds to the height of the weight-balanced trees. Since $w \geq \log n$ (otherwise a pointer would not fit in a word), the cost becomes at most $O(\log n/ \log\log n)$. □

5.2. Exponential search trees

Since their introduction, the fusion trees have been the only available deterministic data structure that uses linear space and supports searching in $o(\log n)$ time and sorting in $o(n \log n)$ time. Recently, an improved data structure has been developed [3].

The basic data structure is a multiway tree where the degrees of the nodes decrease exponentially down the tree.

Lemma 9 *Suppose a static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(S(d))$ worst-case time. Then,*

$$D(n) = O\left(S\left(n^{1/5}\right)\right) + D\left(n^{4/5}\right);$$

Hint of proof: We use an *exponential search tree.* It has the following properties:

- Its root has degree $\Theta(n^{1/5})$.
- The keys of the root are stored in a local (static) data structure, with the properties stated above. During a search, the local data structure is used to determine in which subtree the search is to be continued.
- The subtrees are exponential search trees of size $\Theta(n^{4/5})$.

First, we show that, given n sorted keys, an exponential search tree can be constructed in linear time and space. Since the cost of constructing a node of degree d is $O\left(d^4\right)$, the total construction cost $C(n)$ is given by

$$C(n) = O\left(\left(n^{1/5}\right)^4\right) + n^{1/5} \cdot C\left(n^{4/5}\right) \quad \Rightarrow \quad C(n) = O(n). \tag{2}$$

Furthermore, with a similar equation, the space required by the data structure can be shown to be $O(n)$.

Balance is maintained by joining and splitting subtrees. The basic idea is the following: A join or split occurs when the size of a subtree has changed significantly, i.e. after $\Omega(n^{4/5})$ updates. Then, a constant number of subtrees will be reconstructed; according to

Equation 2, the cost of this is linear in the size of the subtrees $= O(n^{4/5})$. Also, some keys will be inserted or deleted from the root, causing a reconstruction of the root; the cost of this is by definition $O(n^{4/5})$. Amortising these two costs over the $\Omega(n^{4/5})$ updates, we get $O(1)$ amortised cost for reconstructing the root. Hence, the restructuring cost is dominated by the search cost.

Finally, the search cost follow immediately from the description of the exponential search tree. □

Exponential search trees may be combined with various other data structures, as illustrated by the following two lemmas:

Lemma 10 *A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O\left(\frac{\log d}{\log w}+1\right)$ worst-case time.*

Hint of proof: We just construct a static B-tree where each node has the largest possible degree according to Lemma 8. That is, it has a degree of min $\left(d, w^{1/6}\right)$. This tree satisfies the conditions of the lemma. □

Lemma 11 *A static data structure containing d keys and supporting neighbour queries in $O(\log w)$ worst-case time can be constructed in $O\left(d^4\right)$ time and space.*

Hint of proof: We study two cases.

Case 1: $w > d^{1/3}$. Lemma 10 gives constant query cost.

Case 2: $w \leq d^{1/3}$. The basic idea is to combine a van emde Boas tree (Theorem 1) with perfect hashing. The data structure of Theorem 1 uses much space, which can be reduced to $O(d)$ by hash coding. Since we can afford a rather slow construction, we can use the deterministic algorithm by Fredman, Komlós, and Szemerédi [14]. With this algorithm, we can construct a perfectly hashed van Emde Boas tree in $O(d^3 w) = o(d^4)$ time. □

Combining these two lemmas, we get a significantly improved upper bound on deterministic sorting and searching in linear space:

Theorem 6 $D(n) = O(\sqrt{\log n})$.

Hint of proof: If we combine Lemmas 9, 10, and 11, we obtain the following equation

$$D(n) = O\left(\min\left(1+\frac{\log n}{\log w}, \quad \log w\right)\right) + D\left(n^{4/5}\right) \tag{3}$$

which, when solved, gives the theorem. □

Theorem 6 implies that the worst-case cost of sorting n keys in linear space is $O(n\sqrt{\log n})$. This is an improvement from $O(n \log n/ \log\log n)$, achieved by the fusion tree. Taking both n and w as parameters, $D(n)$ is $o(\sqrt{\log n})$ in many cases [3]. For example, it can be shown that $D(n) = O(\log w \log\log n)$.

6. Experiments with some practical algorithms

In (most of) today's computers, the word length (16 or 32) is small enough for address-calculation methods to be efficient even without packed computation. We illustrate this by a brief review of some experimental results. Although there are also promising results for searching (among others, the data structure from Theorem 3 seems to be fast in practice [2]), we concentrate on sorting here. For more details on the experiments, we refer to the references [7, 20].

In the first two experiments the following four sorting algorithms were used for string sorting:

Forward radixsort a new, robust, version of left-to-right radix sort [6].

Adaptive radixsort this algorithm is based on recursive bucketing: we split the universe in a linear number of buckets, distribute the keys among the buckets and sort them recursively.

Program C similar to Adaptive radixsort, but the number of buckets is fixed in advance and independent of the size of input. This particular implementation was presented by McIlroy, Bostic and McIlroy [19].

Quicksort the version by Bentley and McIlroy [11].

The two experiments were made on different kinds of input:

Real world data As input data a collection of standard texts from the Calgary/Canterbury Text Compression Corpus [10] was used. In order to get very large input files, all text files were concatenated and the standard Unix "word" file was added.

Badly distributed data Strings containing long common prefixes and only a few different characters were generated and used. The purpose was to get a bad performance for radix sort algorithms.

The results are given in Figures 4 and 5. We conclude that for reasonably well-distributed data, a simple radix sort is the most efficient choice, while Forward radixsort is a more robust alternative. Quicksort is left far behind.

Next, experiments were made on integer sorting. The following algorithms were tested:

Quicksort, Adaptive radixsort the same algorithms as above adjusted for integer sorting.

Heapsort a version by Carlsson [12].

Mergesort a slightly modified version of that by Gonnet and Baeza-Yates [16], somewhat improved by switching to a simple Insertion sort for short sublists.

O(n loglog n) the algorithm of Theorem 4.

It should be noted that the $O(n \log\log n)$ algorithm was implemented in a straightforward way, whereas all other algorithms were carefully optimised.

The results are given in Figure 6. The worst-case efficient algorithms (Heapsort, Mergesort and the $O(n \log\log n)$ algorithm) are indicated by solid lines and the average-efficient algorithms by dashed lines.

We conclude that Adaptive radixsort seems to be the best choice in practice and that the $O(n \log\log n)$ algorithm performs surprisingly well; it is the fastest among the deterministic algorithms. This indicates that the constant factors involved in the algorithm are not as large as one might fear.

7. Conclusion

Methods based on indirect addressing
(tries, bucketing etc)
are fast in theory and practice.

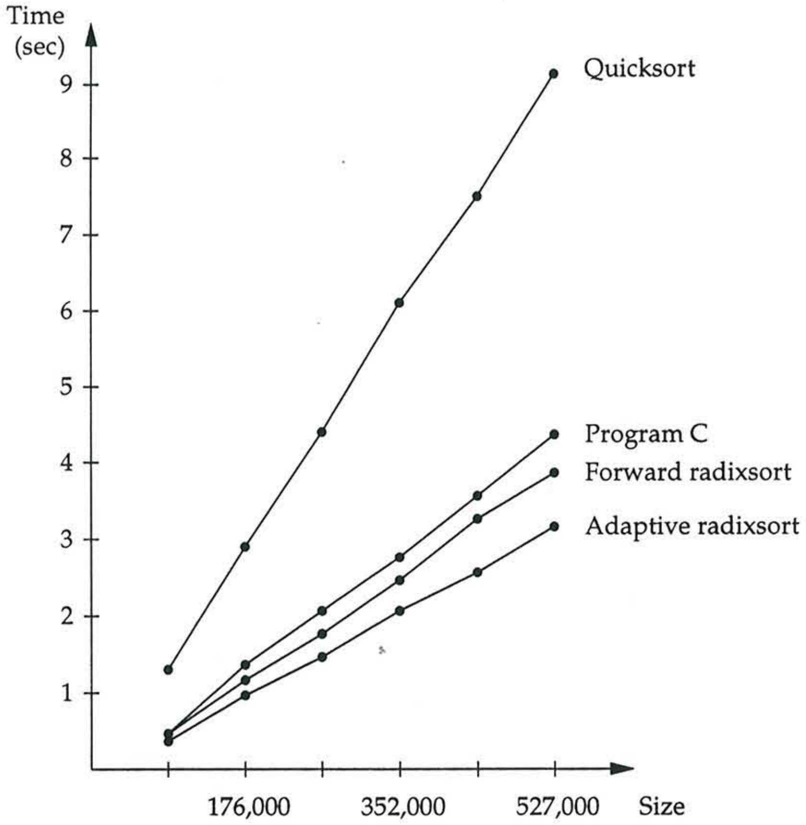


Figure 4: *String sorting, real world data. The number of seconds of CPU time as a function of the number of keys.*

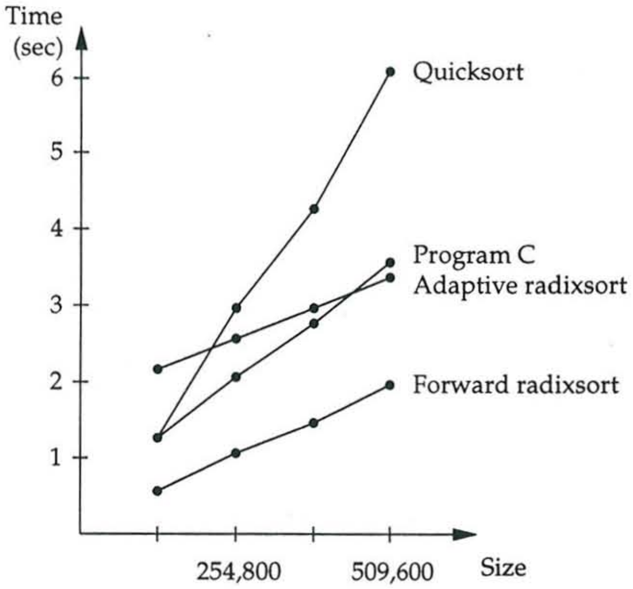


Figure 5: *String sorting, fragmented data.*

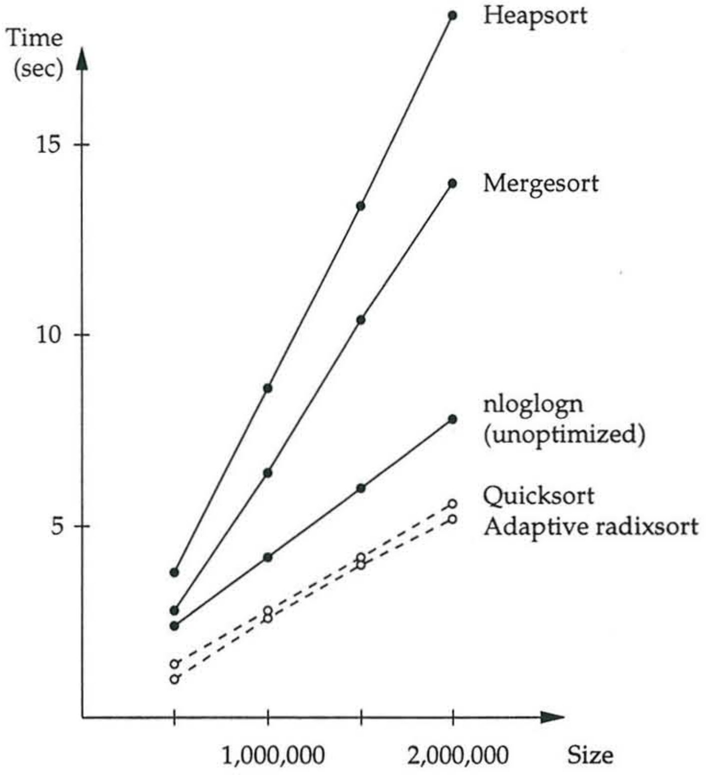


Figure 6: *Integer sorting. Essentially the same results were obtained for random data and ~~fragmented~~ real world data.*

References

[1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd ACM-SIAM SODA*, pages 463–472, 1992.

[2] A. Andersson. Sublogarithmic searching without multiplications. In *Proc.* 36^{th} *IEEE Symposium on Foundations of Computer Science*, pages 655–663. ACM Press, 1995.

[3] A. Andersson. Faster deterministic sorting and searching in linear space. To appear in Proc. IEEE FOCS, 1996.

[4] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings* 27^{th} *ACM Symposium on Theory of Computing*, pages 427–436. ACM Press, 1995.

[5] A. Andersson, P.B. Miltersen, and M. Thorup. Manuscript. In preparation, 1996.

[6] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc.* 35^{th} *Annual IEEE Symposium on Foundations of Computer Science*, pages 714–721. IEEE Computer Society Press, 1994.

[7] A. Andersson and S. Nilsson. manuscript. in preparation, 1996.

[8] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968. Volume 32.

[9] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, 1989.

[10] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[11] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software–Practice and Experience*, 23(11):1249–1265, 1993.

[12] S. Carlsson. *Heaps*. Ph.D. Thesis, Lund University, Sweden, 1986.

[13] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[15] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1994.

[16] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991. ISBN 0-201-41607-7.

[17] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publ., San Mateo, CA, 1994.

[18] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[19] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[20] S. Nilsson. *Radix Sorting & Searching*. Ph.D. Thesis, Lund University, Sweden, 1996.

[21] W. J. Paul and J. Simon. Decision trees and random access machines. In *Logic and Algorithmic: An International Symposium Held in Honour of Ernst Specker*, pages 331–340. L'Enseignement Mathématique, Université de Genevè, 1982.

[22] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 75–84, 1975.

[23] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[24] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

DISCUSSION

Rapporteur: Dr Robert Stroud

Lecture One

Professor Bird asked how the size of the problems was being measured. Dr Andersson explained that N referred to the number of lines of input. However, it wasn't necessary to scan all the input text, just scan until the strings were different. This optimisation had been applied to both algorithms.

Professor Paterson asked about the lower bounds for the structured model. For sorting, the lower bound was linear. Using a randomised algorithm, the bounds were also linear if the word length was greater than $(log\ N)^{2+\varepsilon}$. For searching, the bounds were almost $O\left(\frac{\sqrt{log\ N}}{log\ log\ N}\right)$. (This was an unpublished result by Bean and Fip.) These results were for a standard instruction set with constant circuit depth - the algorithms used shifting and bitwise logical OR but not multiplication.

Professor Randell asked how sensitive the results were to the initial ordering of the data. Dr Andersson explained that unlike comparison-based algorithms, the key-structured algorithms did not take advantage of pre-sortedness, at least not in a conventional sense. In fact both kinds of algorithms took advantage of different things - the comparison-based algorithms exploited the ordering of the keys whilst the structured algorithms exploited their binary distribution. They worked fast provided that the average number of bits that had to be read in order to distinguish each key was small.

Professor Nievergelt recalled an algorithm proposed 20 years ago that claimed to be able to sort in unit time using a floor function. Was this a kind of radix sort and had the authors forgotten about the *log log N* factor? Dr Andersson said that the flaw in this algorithm was that it assumed that the keys were short and could be packed into large words - this was cheating.

Lecture Two

Professor Paterson asked whether the tree could be changed in the middle of a demonstration. It was possible to perform manual updates via insert/delete and to rebalance the tree, and it was also possible to play the recording backwards and forwards (up to 20 updates in either direction) and to load a tree from a file. However, it wasn't possible to perform arbitrary tree surgery interactively.

Professor Rayward-Smith asked how it was possible to tell if the students understood what you were trying to teach them via the demonstration. Dr Andersson agreed that a demonstration was not enough to make students understand. Perhaps you could step through and ask them to guess what would happen next?

Professor Lee asked whether students could animate their own code. The answer was no - it took about 1-2 days to add a new algorithm to the demonstration. (Incidentally, the code had been written in Simula and the notion of processes had been very helpful.)

Professor Paterson asked whether the demonstration could produce high quality diagrams suitable for printing in papers. Although it was possible to print screen dumps, the program didn't generate Postscript although this feature could easily be added.

Professor Randell remarked that there was a certain school of Computer Scientists who regarded looking at how a program actually worked as reprehensible. Programmers should reason about the behaviour of their programs from pre and post conditions alone and should be concerned with correctness rather than performance. How would Wirth or Dijkstra react to such a demonstration?

Dr Andersson replied that there were a lot of non-trivial issues that needed to be addressed in order to make a nice animation. For example, the ability to stop between steps after making half an update and to interrupt/resume the update process required support from the underlying programming paradigm being used to construct the demonstration. Professor Randell accepted that these were important issues assuming you wanted to show an animation but his point was that some purists would argue that such animations were inappropriate.

Dr Andersson responded by saying that he was an intuitionist rather than a formalist. Although formalism was useful, intuition was extremely important too. People would remember the intuition and use the formalism to convince themselves that the intuition was correct. But you couldn't just teach by demonstration - it needed to be backed up by the theory. Professor Randell agreed, adding that he thought that intuition was also important for inventing new algorithms. Dr Andersson used the example of splay trees to illustrate this point - you could learn a lot by playing around with them for a while and might even gain some new insights that led to a new formalism in the process.

Professor Henderson commented that even formalists will use diagrams sometimes to get your intuition (although perhaps Dijkstra was an exception!). Professor Randell said that he could remember a time when Dijkstra used diagrams - but he now had the zeal of a converted sinner!

Dr Andersson remarked that we all have different intuitions - he needed a blackboard to think but not everyone was like this. Illustrating ideas by example was not trivial - you needed to connect the example to a good explanation of the concept you were trying to illustrate. For this reason, in class he would spend much more time on the actual code for the skew and split operations that he had just demonstrated.

Professor Randell suggested using students as adversaries and testing their understanding by asking them to come up with tricky cases - perhaps this was another avenue to understanding.

Professor Paterson asked what was in the package that Dr Andersson was making available. Was it easy to add new algorithms? Professor Bird asked whether it was possible to display the code used for each algorithm. Dr Andersson replied that the demonstration included Pascal source code for the tree algorithm which was essentially identical to the Simula code that was actually used to implement the algorithm. Although the Simula code was about 5000 lines of code, most of it was concerned with the animation - the actual code for the algorithms being animated should fit on a page.

Professor Randell asked to what extent this kind of animation could be used to explain local search algorithms search as simulated annealing. Dr Andersson replied that if you looked in the kind of "voodoo" books that Professor Lemstra had referred to, you would find lots of pictures to illustrate problems and their solutions but these were illustrations of results rather than algorithms. You could illustrate temporary results during a travelling salesman problem but not how the algorithm worked. And how would you animate a complex job shop? You could use a Gantt chart but it quickly became too complex with more than three processes...

Professor Randell wondered whether it was possible to show the effect of data on local search and perhaps illustrate concepts such as like/unlike. Dr Andersson replied that it was hard to show a dimensionality of 60 in a nice way - although intuition might help, it could also be misleading. He cited the example of the Geometrical algorithms discussed earlier in the seminar and observed that animation sometimes had its drawbacks and that formalism could be used to remove the errors introduced by intuition.