Fusion Tree Sorting*

Luis A. A. Meira
School of Technology
University of Campinas, Brazil

Rogério H. B. de Lima
Institute of Science and Technology
Federal University of São Paulo, Brazil

Abstract

The sorting problem is one of the most relevant problems in computer science. Within the scope of modern computer science the sorting problem has been studied for more than 70 years. In spite of these facts, new sorting algorithms have been developed in recent years. Among several types of sorting algorithms, some are quicker; others are more economic in relation to space, whereas others insert a few restrictions in relation to data input. This paper is aimed at explaining the fusion tree data structure, which is responsible for the first sorting algorithm with complexity time smaller than $n \lg n$. The $n \lg n$ time complexity has led to some confusion and generated the wrong belief of being the minimum possible for this type of problem.

1 Introduction

The sorting problem is perhaps the most studied problem in Computer Science. Its use is implicit in intermediate stages of almost all existing programs, such as database, spreadsheets, multimedia, etc. In operational research context, it is present in many optimization steps. In addition, sorting has been studied by computer science for over 70 years. The currently and broadly used merge sort algorithm was proposed by Von Neumann in 1945 [3].

The sorting problem consists of receiving a sequence $A = (a_1, \ldots, a_n)$ of n numbers as input. The solution consists of a nondecreasing permutation $A' = (a'_1, \ldots, a'_n)$ of A. Although this work is focused on integers, the extension for rationals, floating point and character strings tend to be straight.

All the sorting algorithms present characteristics that make them somehow more or less competitive in relation to their peers. Some of these characteristics are the sorting type, either stable or non-stable, extra space utilization for algorithm execution, and sorting time. Some algorithms can be quicker than others, depending on the characteristics of input data. For instance, selection sort tends to be advantageous when n is small. Insertion sort tends to be rapid when the vector is partially sorted. Counting sort is advantageous when the difference between the maximum and the minimum element is limited.

The most broadly known sorting algorithms are comparison-based ones, such as merge sort, heap sort, insertion sort and quick sort, in addition to the counting-based ones, such as, for example, counting sort, bucket sort and radix sort. The counting-based algorithms require an

*This research was partially supported by the State of São Paulo Research Foundation (FAPESP grant 2013/00836-1).

input sequence with some restrictions. When such restrictions are satisfied, these algorithms can solve the sorting problem in linear time.

There is a lower bound of $\Omega(n \lg n)$ comparisons for sorting algorithms [7]. Such limit is based on a decision tree with $n!$ leaves, each of them representing an input vector permutation. Each permutation is a candidate to solve the problem. Provided that a comparison can distinguish two branches of a tree, a minimum of $\lg(n!) = \Theta(n \lg n)$ comparisons are required to sort a vector through a comparison-based sorting algorithm in the worst case. This lower bound was misinterpreted, thus generating a false belief in terms that sorting is a $\Omega(n \lg n)$ problem. Such limit does not apply, for example, to algorithms using other operations rather than comparisons during the sorting process. The counting sort is able to sort a vector without performing any kind of comparison between the elements.

The algorithm under analysis in this paper is a comparison-based one and makes $\Theta(n \lg n)$ comparisons. However, $(\lg n)^{1/5}$ numbers are compared in $O(1)$. This means that multiple operations are performed in constant time.

Results: The sorting algorithm $O(n \lg n / \lg \lg n)$ under analysis in this paper is known to the literature. Our contribution consists of detailing the fusion tree data structure and the related sorting algorithm $O(n \lg n / \lg \lg n)$ proposed by [6]. A full version of this paper is available [8, 5].

1.1 Computational Model

Consider a computer working with w-bit words. This computer is able to perform elementary operations such as addition, subtraction, multiplication, division, and remainders with w-bit integers in constant time. For example, a 64-bit computer has the capacity of processing 64 bits in constant time.

The general sorting case deals with integers with an arbitrary precision. For an integer with mw-bits, it is required m accesses to the memory before completing the number reading. This paper works with the restricted sorting case where numbers are integers with w bits. Such numbers are in the range $\{-2^{w-1}, \ldots, 2^{w-1}\}$ stored as binary integers, with 1 bit for the signal. Some special attention is needed to deal with repeated numbers. Thus, no repetition is assumed to simplify the explanation.

This work considered a computational model that is able to read and write any memory position in constant time, which is known as RAM memory. The RAM memory model is acceptable, though coexisting with the sequential access memory model. In the sequential access memory model, the tape needs to be moved up to the desired position before reading, thus spending linear time to read an integer. The merge sort algorithm is famous for keeping the complexity $O(n \lg n)$ even in a sequential memory model.

Several $o(n \lg n)$ have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm [4]. All the results assume that the computer memory is divided into addressable w-bit words. It is assumed that the computer is capable of processing $w = \log n$ bits in constant time. In the case of n integers of w bits in the memory, the maximum memory address will have at least $\lg n$ bits. Notice that the number of bits in the problem is $nw \geq n \lg n$. This means that one operation for each bit is $\Omega(n \lg n)$.

For a better understanding of the sorting process, we shall first show how to sort n numbers using the B-tree data structure. The fusion tree data structure was proposed by [6] and it is a

modified B-tree.

2 B-Trees

B-trees are balanced search trees with a degree t, where $\frac{B}{2} \leq t \leq B$, for constant B. Each node contains a minimum of $\frac{B}{2}$ children and a maximum of B children, except for the leaves, which contain no child, and the root-node, which does not present restriction in the minimum number of children. Each node has $t-1$ keys, and all the leaves are found in the same level. Notice that a degree-4 node has three keys.

In addition, the B-tree respects the following property: Each non-root node has $t-1$ sorted elements $S = (s_1, \ldots, s_{t-1})$. Each non-leaf and non-root node has t children $(f_0, \ldots, f_{t-1})$ where each child is a B-tree. The elements in the f_0 tree are smaller than s_1. The elements in f_i are greater than s_i and smaller than s_{i+1}. The elements in f_{t-1} are all greater than s_{t-1}.

Searching a key $k \notin S$ in a B-tree node requires finding the correct child X to continue the search. If $k < s_1$, the search continues int f_0 child. If $k > s_{t-1}$, the search continues in f_{t-1} child. If $s_i < k < s_{i+1}$, the search continues in f_i child, between s_i and s_{i+1}. The B-tree operations complexity time are related to its height. The following lemma is based in [3].

Lemma 1 *A B-tree with degree $B \geq 4$ and height h respect: $h = O(\log_B n)$*

A sequential search is made to search a key k in a B-tree node. Such search takes $O(B)$ and it is repeated in each B-tree level in the worst case. The result is an $O(B \log_B n)$ overall time to search the key. As B is constant, the complexity is equivalent to $O(\lg n)$.

The key insertion needs an initial search to find the recipient node. If such node is incomplete, the key can be accommodated into the node in $O(B)$. It is the cost to insert an element in a central position of a vector with B elements. If the recipient node is full, it must be split. Let s_m be the vector median. Such element is inserted in the parent node. One node is created with the elements smaller than s_m and other with the elements greater than s_m. Such nodes become the left and the right child of s_m respectively. Both nodes have exactly $\frac{B}{2} - 1$ keys.

A vector can be split in half in $O(B)$ through elementary operations. If the parent node is complete, it must be also split. Such process can propagate up to the root.

To sort a sequence with n elements using a B-tree, all elements must be inserted in an initially empty tree. An in-order traversal result in a sorted sequence. The complexity time to sort n integers is the sum of the time to insert n keys in the tree that is $O(nB \log_B n)$. If B is a constant, such complexity will be $O(n \lg n)$.

3 Fusion Tree

This section describes the fusion tree data structure proposed by [6]. A fusion tree is similar to a B-tree in many aspects. One difference between B-tree and the fusion tree is the B value. In a B-tree the B value is a constant while in a fusion tree the B is a function of n. More precisely, $B = (\lg n)^{\frac{1}{5}}$. Another difference is the time to search a key in a node. The B-tree uses O(B) operations while the fusion tree uses O(1) operations to search a key k in a node.

Consider the problem of finding the predecessor or the successor of a key x in a set S. Such problem consists in finding the number immediately above or below x in S. Fusion tree is a data structure similar to B-tree but it solves the predecessor and successor problem in O(1)

inside a node. Given a search key x, the fusion tree is able to find the child branch relative to x in constant time despite the fact that the size of S increases with n.

The following notation present in [6] is needed:

Definition 1 rank(x)*: Given a set of integer numbers S and an integer x, let* rank(x) *be the value* $|\{t \mid t \in S, t \leq x\}|$*. In other words,* rank(x) *represents the number of elements smaller than or equal x.*

The problem of sort n number is equivalent to finding rank(x) for all x. Such function provides the exact x position in the sorted vector. Moreover, rank(x) provides the correct child to continue the search of an element x in a B-tree node.

Fusion tree is based in a trie data structure described in [2]. Next subsection is devoted to the trie data structure.

3.1 Trie data structure [2]

Let a trie be a binary tree with the following construction rule. Given a w-bit integer x, each bit of x is a node in the trie. If the most significant bit of x is 0, x is a left root child. If it is 1, x is a right root child. Such property is recursively applied to each bit of x.

Given an arbitrary integer i, let b_i be the i-th least significant bit. Thus, b_0 is the least significant bit, b_1 is the second least significant bit and so on. Consider two binary integers $s_1 = 11101001$ and $s_2 = 11111001$. Figure 1 shows a trie with s_1 and s_2. The trie leaves are always sorted. We consider rank(x) calculated for all element in the trie.

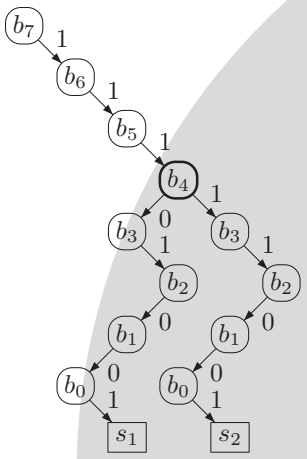

Figure 1: Trie data structure for s_1 and s_2. It is enough to compare b_4 to sort s_1 and s_2.

Consider the following definition:

Definition 2 $\Delta(s_1, s_2)$*: Given two integers s_1 and s_2, let $\Delta(s_1, s_2)$ be the relevant bit between s_1 and s_2, meaning the most significant bit that diverges between s_1 and s_2.*

When a set of integers are compared, some bits are irrelevant and can be discarded. Only a few-bit, named relevant bits, are sufficient to sort a set of integers.

Definition 3 ***relevant bits****: Consider a trie with a set of elements S. The relevant bits of S are the bits for which there is a branch in the trie.*

In the previous example, to compare and sort the binary number $s_1 = 11101001$ e $s_2 = 11111001$, it is sufficient to compare the most significant bit that diverges between s_1 and s_2. Considering the previous numbers, such bit is b_4, with values 0 in s_1 and 1 in s_2. Thus, $\Delta(s_1, s_2) = b_4$. Such bit is used to conclude that s_1 is greater than s_2. All other bits are irrelevant. See Figure 2.

Let $\oplus$ be a bitwise XOR between two words. Given two integers s_1 and s_1, $\Delta(s_1, s_2)$ can be obtained as: $\Delta(s_1, s_2) = \lfloor \lg(s_1 \oplus s_2) \rfloor$.

Consider an integer sequence $S = (s_1, \ldots, s_t)$ and a trie data structure. After inserting all S elements in the trie, a compression will be performed where all irrelevant bits will be discarded. Such new tree will be named patricia trie.

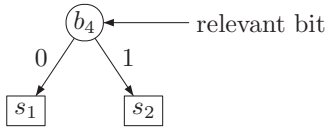

Figure 2: Patricia trie used to compare s_1 and s_2 with the relevant bit.

The patricia trie has each element of S as a leaf. The internal nodes store the respective relevant bit.

Lemma 2 *Given a patricia trie with $S = (s_1, \ldots, s_t)$, the numbers of relevant bit will be at most $t - 1$.*

Lemma 2 is correct because each relevant bit is related to a branch in the patricia trie. The number of branches will be exactly $t - 1$. Eventually, two distinct branches can occur at the same level.

To search a key x in a patricia trie, each bit of the trie is compared with the correspondent x bit from the root to the leaves. In each node, if the x bit is zero, the search continues in the left branch. If the bit is 1, the search continues in the right branch. Figure 3 illustrates a search of a key x in a patricia trie with elements a, b, c and d. Let TrieSearch(x) be such search result. In Figure 3, TrieSearch(x) = c.

3.1.1 Computing rank(x)

Suppose a set $S = (s_1, \ldots, s_t)$ inserted in a patricia trie. This section will describe how to compute rank(x) for a given key x. An initial search $s' =$ TrieSearch(x) is computed. The s' element has the same values than x in the relevant bits. If s' is equal to x in the remaining bits, rank(x) = rank(s') + 1 and the rank is computed. In the other case, a new search will be performed. First, consider the bit $b' = \Delta(x, s')$.

Lemma 3 *The bit $b' = \Delta(x, s')$ is the new relevant bit in the patricia trie with $S \cup \{x\}$ elements.*

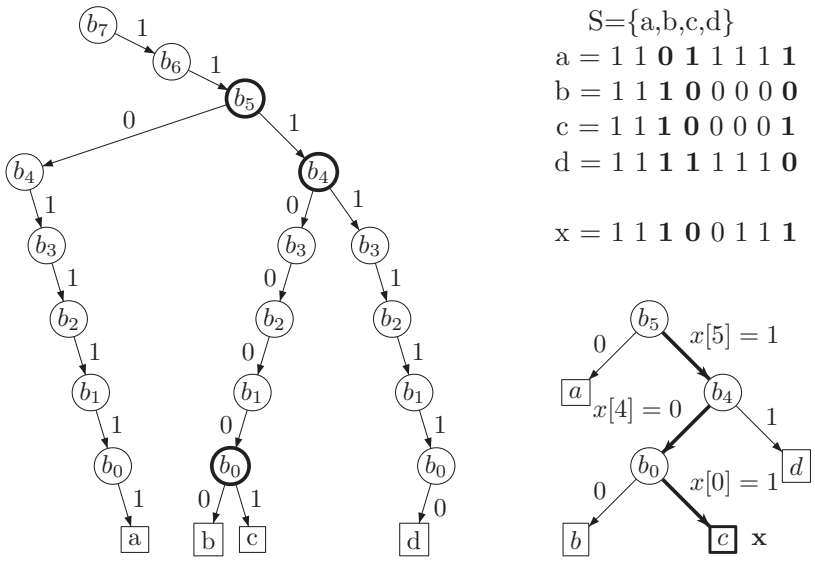

Figure 3: The search of x in the patricia trie data structure.

The rank(x) calculus will be dived in two cases. In the first, the bit b' of x is 1, which means $x[b'] = 1$, while in the second case $x[b'] = 0$.

Lemma 4 *The most significant bits of x and s' are equals. The first bit to diverge is b'. Consider the branch between x and s' in the trie with $S \cup \{x\}$. If $x[b'] = 1$, the x predecessor is the largest element in the $b' = 0$ branch. If $x[b'] = 0$, the x successor is the smallest element in the branch $b' = 1$.*

Case a ($x[b'] = 1$) Figure 5 has an example in which the predecessor of x is the largest element in the subtree highlighted.

A second search is needed to compute rank(x). From the most significant bit to the relevant bit b', the patricia trie search uses the bits of x. Starting from b', the search looks for the largest element in the subtree, i.e., the search will down the tree always to the right branch in direction of the largest element.

A new search key x' will be computed to obtain such behavior in the following way:

$$\begin{array}{rr} & x = x_{w-1}x_{w-2}\ldots x_2x_1x_0 \\ \text{OR} & \qquad\qquad 1\ 1\ 1\ 1 \\ \hline & x' = x_{w-1}x_{w-2}\ldots 1\ 1\ 1\ 1 \end{array}$$

The number of 1's at the end of x' is b'. Such mask can be computed in $O(1)$ as $2^{b'+1} - 1$. When a bit of x is replaced by 1 from b' to b_0, the new search key will find x predecessor. Let $s'' = \text{TrieSearch}(x')$. Then $\text{rank}(x) = \text{rank}(s'') + 1$. Figure 6 has a sample.

Case b ($x[b'] = 0$) From the most significant bit to the relevant bit b', the patricia trie search uses the bits of x. Starting from b', the search looks for the smallest element in the subtree, i.e., the search will down the tree always to the left branch in direction of the smallest element.

A new search key x' will be computed to obtain such behavior in the following way:

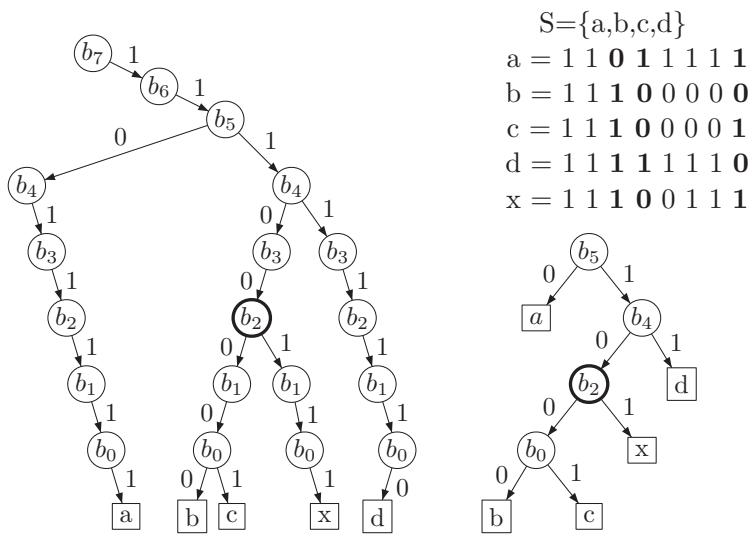

Figure 4: Trie and patricia trie after the x insertion [2].

$$\begin{array}{rr} & x = x_{w-1}x_{w-2}\ldots x_2x_1x_0 \\ \text{AND} & 1\,1\,1\,1\ldots 0\,0\,0\,0 \\ \hline & x' = x_{w-1}x_{w-2}\ldots 0\,0\,0\,0 \end{array}$$

The number of zeros at the end of x' is b'. When a bit of x is replaced by zero from b' to b_0, the new search key will find x successor. Let $s'' = \text{TrieSearch}(x')$.

3.2 Fusion Tree Characteristics

Basically, the fusion tree is a B-tree with degree $B = (\lg n)^{\frac{1}{5}}$, i.e, the degree is an increasing function with respect to the number of elements.

As the height of a B-tree is proportional to $\log_B n$ and the fusion tree has $B = (\lg n)^{\frac{1}{5}}$ so the height h has complexity: $\log_B n = \frac{\lg n}{\lg B} = \frac{\lg n}{\lg(\lg n)^{1/5}} = \frac{\lg n}{\lg \frac{1}{5} \lg n} = O(\frac{\lg n}{\lg \lg n})$.

The time to search the correct child to continue the search in a B-tree node is $O(B)$ using linear search. As the search occurs in each tree level, the overall time is $O(B \log_B n)$ to perform a search. In a fusion tree, the child is found in $O(1)$ in a node and in $O(\log_B n)$ in the tree. As $B = (\lg n)^{\frac{1}{5}}$, the search complexity time is $\log_B n = O(\frac{\lg n}{\lg \lg n})$.

As previously discussed, some irrelevant bits can be discarded in the sort process. A special structure name sketch is created to save only the relevant ones:

Definition 4 sketch*(s): The sketch of a word s consists in discarding all irrelevant bits, keeping the relevant ones. Sketch operations preserve the words order, i.e., $s_i < s_j$ if and only if* $\text{sketch}(s_i) < \text{sketch}(s_j)$.

Figure 3 has the elements a, b, c and d sketches. They are 011, 100, 101 e 110 respectively. The sketches order doesn't change with respect to the original numbers.

The fusion tree central idea is concerned with how it stores the key in each node. Each node contains t keys, for $t < B - 1 = O(w^{1/5})$. As stated in Lemma 2, a trie with $B - 1$ keys

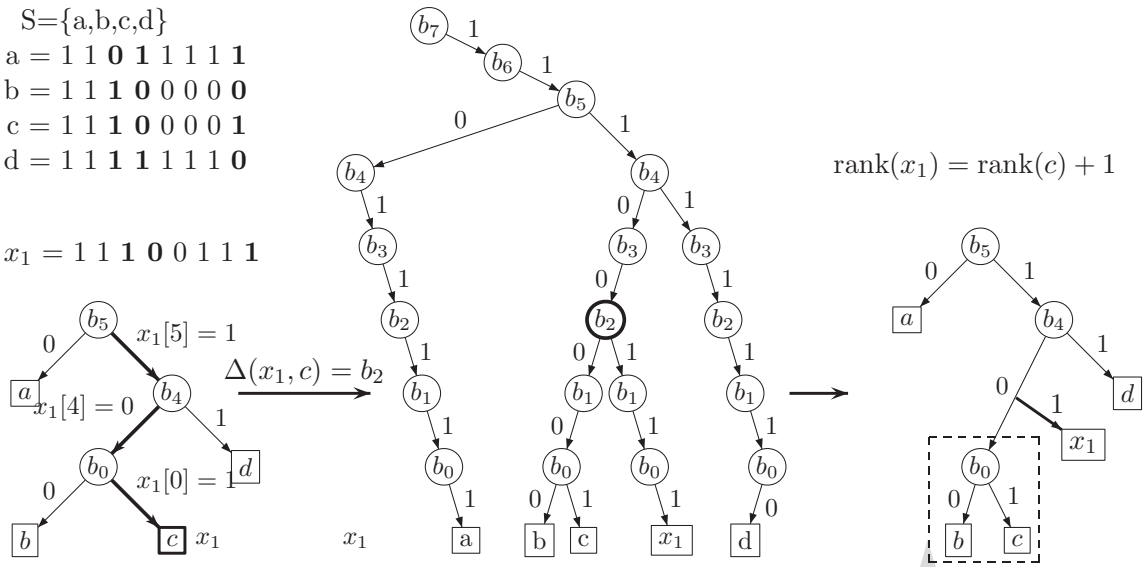

Figure 5: Computing rank(x_1).

has at most $B-2$ relevant bits. A node contains $B-1$ sketches each with $B-2$ relevant bits. Thus, the overall sketch bits in a node are $(B-1)\cdot(B-2) \leq w^{\frac{1}{5}} \cdot w^{\frac{1}{5}} = O(w^{\frac{2}{5}}) = o(w)$.

The sum of sketches bits in a node fits in only one memory word. Thus, each fusion tree node has one word that keeps one sketch for each key plus some bits as defined above:

Definition 5 *(Sketch Node) The sketch node is a node that contains all keys sketches. Such sketches can be stored in only one word. Additionally, there is a separator bit between the sketches whose value is 1. The sketch node will be the concatenation of each key sketch:* $w_{node} =$ 1sketch(s_1)1sketch(s_2)...1sketch(s_t). *Furthermore, sketches are concatenated in nondecreasing order.*

The next subsection will show how to compare a key x with all keys in a node in constant times, based in [9].

3.3 Multiple comparisons in constant time

Consider a fusion tree node with elements $S = (s_1, \ldots, s_t)$. Suppose the relevant bits with respect to S are $(i_1, \ldots, i_{t'})$ with $t' < t$. To compare a search key x with all node keys, first sketch(x) is computed.

To extract the first relevant bit i_1 and store it in the first position of sketch(x), it is computed a bitwise AND between x and a mask with value 1 only in the bit i_1. Once the mask is applied, the bit must be moved to sketch vector position 0. Such movement of delta bits is obtained by a multiplication by 2^{delta}.

To obtain all relevant bits of x in the initial position of sketch(x), first a bitwise AND is performed between x and a mask with 1 only in the relevant bits $(i_1, \ldots, i_{t'})$. Such mask will

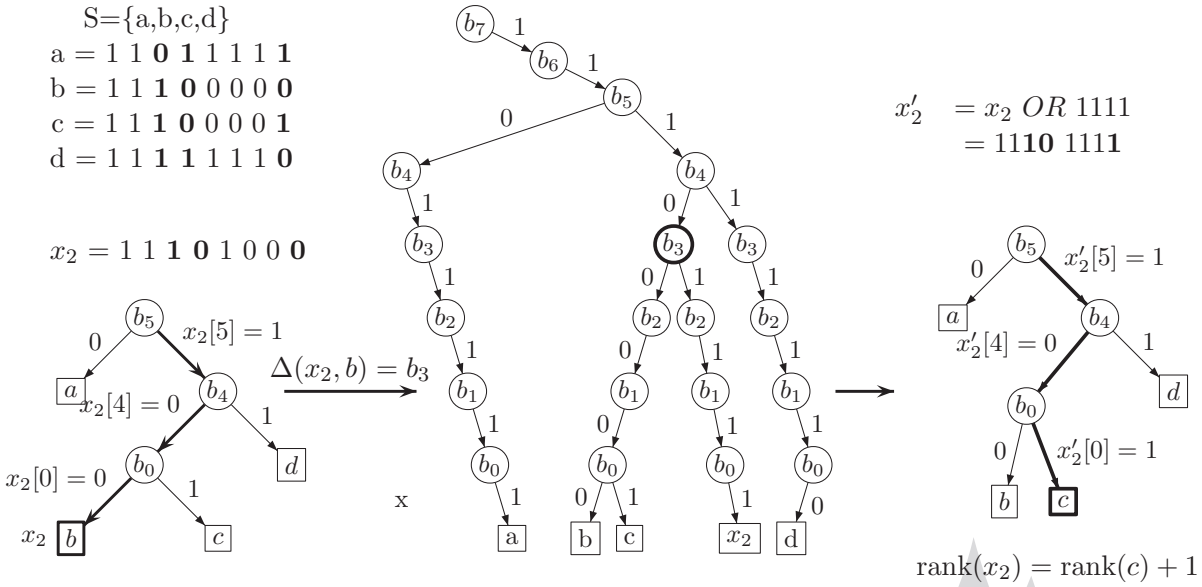

Figure 6: Computing rank(x_2) after the second search in the patricia trie.

be constructed with the patricia trie and is available when the x search is computed. After this, all relevant bits must be moved to the initial position of sketch(x) in $O(1)$ as shown in the figure above.

When an arbitrary number x is multiplied by a predefined constant, it is possible to reposition the bits of x. The problem of repositioning the relevant bits of x to the initial position of sketch(x) in $O(1)$ is a nontrivial task. The work [1] discuss the existence of predefined constants to reposition the relevant bits of x. The result is imperfect because some additional zeros bits are added to the sketch(x). Such additional zero bits do not change the algorithm behavior. The sketch(x) computation is not covered by this work.

Once sketch(x) is computed, its value is concatenated t times in the following way: $w_x =$ 0 sketch(x) 0 sketch(x) ... 0 sketch(x).

Suppose that sketch(x) has 6 bits, so: $w_x = \text{sketch}(x) + \text{sketch}(x) \cdot 2^7 + \text{sketch}(x) \cdot 2^{14} + \ ... = \text{sketch}(x) \cdot (...10000010000001)$.

Thus, w_x is computed from sketch(x) with only one multiplication.

Fact 1 *When subtracting* 1sketch(s_i) − 0sketch(x)*, the result starts with 1 if and only if* sketch(x) ≤ sketch(s_i).

Let sketch(x) = 1111 and sketch(s_i) = 0000, thus 1sketch(s_i) − 0sketch(x) = 10000 − 01111 = **0**0001. As the subtraction result starts with zero, then sketch(x) > sketch(s_i).

Suppose sketch$(x) = 0000$ and sketch$(s_i) = 00001$. Thus 1sketch$(s_i)-$0sketch$(x) = 10001 - 00000 = \mathbf{1}0001$. As the subtraction result starts with 1, sketch$(x) \leq$ sketch(s_i).

To compare x with all words in S in $O(1)$, a subtraction between 1sketch(s) − 0sketch(x) for all $s \in S$ is performed in one operation. That means many comparisons with only one operation. The calculus is $w_{res} = w_{node} - w_x$.

The first bit of each block will indicate if sketch(x) is lesser than or equal or greater than sketch(s_i). As the sketches are sorted in a w_{node}, the first block that starts with 1 must be found. Suppose that the number of bits of a block 0sketch(x) is r. To remove all bits except the first bit of each block, a bitwise AND is performed between w_{res} and a mask with value 1 in the positions r, $2r$, $3r$ and so on. Let w'_{res} the result of such bitwise AND. The next step consists in finding the most significant bit that values 1. Such operation is equivalent to calculate $\lfloor \lg(w'_{res}) \rfloor$ and must be performed in O(1). Such problem is found in the literature [10].

The element $s =$ TrieSearch(x) can be computed from the position of the first 1 in a w'_{res}. Following the steps of previous section, the rank value can be computed in $O(1)$. Thus the correct child to continue the search in a fusion tree is computed in $O(1)$.

3.4 Sorting in $o(n \lg n)$

This work detailed how to search a w-bit word in $O(\frac{\lg n}{\lg \lg n})$ in a fusion tree data structure. It also describes how to sort n elements using B-tree. All elements inserted in a fusion tree result in a sorted set of elements. The paper [11] shows how to transform a static fusion tree in a dynamic one. A dynamic fusion tree is optimized to update keys in $O(\frac{\lg n}{\lg \lg n} + \lg(\lg(n)))$ by update. The resulting sort complexity is $n\left(\log_B n + \frac{\lg n}{\lg \lg n} + \lg \lg n\right) = O\left(n \frac{\lg n}{\lg \lg n}\right)$.

4 Conclusion

This work aimed to describe the fusion tree data structured and the $O(\frac{n \lg n}{\lg \lg n})$ sorting algorithm. Step by step examples is prepared for didactic purposes. Very few materials are available related to this relevant issue. The challenge was to understand many theorems and non trivial concepts and prepare a material to a wide community.

This work let some open questions as *(i)* how to discover the first bit 1 in w-bit word in $O(1)$; *(ii)* how to compute sketch(x) in $O(1)$ and *(iii)* how to create dynamic fusion tree optimized to update keys. Anyway, this work successfully completes the task of detailing the fusion tree data structure, responsible for the first $o(n \lg n)$ sorting algorithm and a basis for many other subsequent algorithms.

Such work also reveals some pitfalls in the use of lower bounds. For instance, if a generic problem needs at least $f(n)$ operations, the real lower bound is $\Omega(f(n)/\lg n)$ because the widely accepted computational models are able to process $\lg n$ bit in $O(1)$.

An opportune future work would be to implement the fusion tree sorting algorithm and compare it with traditional algorithms. Another relevant aspect is the possibility of multiple operations in $O(1)$ and the removal of irrelevant bits. Such possibilities present theoretical and practical consequences. In the theoretical field, the question is which problems could have their complexity decreased with multiple operations in $O(1)$. In applied computing, the use of multiple operations inside a single word and the removal of irrelevant bits can accelerate traditional algorithms.

References

[1] Advanced data structures, mit, lecture 12, prof. erik demaine, 2012.

[2] Miklós Ajtai, Michael L. Fredman, and János Komlós. Hash functions for priority queues. *Information and Control*, 63(3):217–225, December 1984.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, third edition edition, 2001.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2, chapter Chapter 8 - Chapter Notes. MIT press Cambridge, third edition edition, 2001.

[5] Rogério H. B. de Lima and Luis A. A. Meira. Ordenação baseada em árvores de fusão. *ArXiv e-prints http://arxiv.org/abs/1407.6753*, abs/1407.6753, 2014.

[6] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, pages 47:424–436, 1993.

[7] D. E. Knuth. *The Art of Computer Programming*, volume 3. Reading, 2 edition, 1998.

[8] Luis A. A. Meira and Rogério H. B. de Lima. Fusion Tree Sorting. *ArXiv e-prints http://arxiv.org/abs/1411.0048*, October 2014.

[9] scribe: Nicholas Zehender Prof. Erik Demaine. Lecture 10. MIT - Massachusetts Institute of Technology, 3 2010. Advanced Data Structures.

[10] Henry S. Warren. *Hacker's delight*. Pearson Education, 2003.

[11] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29:1030–1049, December 1999.