Thirty nine years of stratified trees
(Invited Paper)

Peter van Emde Boas
ILLC, FNWI, University of Amsterdam,
and Bronstee.com Software & Services B.V., Heemstede
the Netherlands
Email: peter@bronstee.com

Abstract—The stratified tree, also called van Emde Boas tree, is a data structure implementing the full repertoire of instructions manipulating a single subset $A$ of a finite ordered Universe $U = [0\ldots u-1]$. Instructions include member, insert, delete, min, max, predecessor and successor, as well as composite ones like extract – min. The processing time per instruction is $O(\log\log(u))$. Hence it improves upon the traditional comparison based tree structures for dense subsets $A$; if $A$ is sparse, meaning that the size $n = \#A = O(\log(u))$ the improvement vanishes.

Examples exist where this improvement helps to speed-up algorithmic solutions of real problems; such applications can be found for example in graph algorithms, computational geometry and forwarding of packets on the internet.

The structure was invented during a short postdoc residence at Cornell University in the fall of 1974. In the sequel of this paper I will use the original name Stratified Trees which was used in my own papers on this data structure.

There are two strategies for understanding how this $O(\log\log(u))$ improvement can be obtained. Today a direct recursive approach is used where the universe is divided into a cluster of $\sqrt{u}$ galaxies each of size $\sqrt{u}$; the set manipulation instructions decompose accordingly in a instruction at the cluster and galaxy level, but one of these two instructions is always of a special trivial type. The processing complexity thus satisfies a recurrence $T(u) = T(\sqrt{u}) + O(1)$. Consequently $T(u) = O(\log\log(u))$.

However, this recursive approach requires address calculations on the arguments which use multiplicative arithmetical instructions. These instructions are not allowed in the Random Access Machine model (RAM) which was the standard model in the developing research area of design and analysis of algorithms in 1974. Therefore the early implementations of the stratified trees are based on a different approach which best is described as a binary-search-on-levels strategy. In this approach the address calculations are not required, and the structure can be implemented using pointers. The downside of this approach is that it leads to rather complex algorithms, which are still hard to present correctly even today. Another bad consequence was the super linear space consumption of the data structure, which was only eliminated three years later.

In this paper I want to describe the historical backgrounds against which the stratified trees were discovered and implemented. I do not give complete code fragments implementing the data structure and the operations; they can be found in the various textbooks and papers mentioned, including a Wikipedia page. Code fragments appearing in this paper are copied verbatim from the original sources; the same holds for the figures\(^1\).

I. INTRODUCTION

The Stratified Trees which in the literature frequently are named van Emde Boas Trees were discovered in the fall of 1974 during a short postdoc residence at Cornell University. They were designed in order to improve the time complexity of manipulating a subset $A$ of some finite ordered set $U$, called the Universe. The size of the Universe $U$ is denoted $u$, and without loss of generality we may assume that $U = [0\ldots u-1]$. The size of the set $A$ is denoted $n$. The case where $U$ should be considered an infinite set is realised in practice by letting $u$ be much larger than $n$.

Manipulation of such a dynamic set $A$ is a core algorithmic task in areas like graph algorithms, computational geometry and databases, but also in applications like message passing on the Internet; applications which still had to be invented in 1974.

The instructions which we consider are:

1) $\text{insert}(x)$: adds element $x$ to $A$
2) $\text{delete}(x)$: removes $x$ from $A$
3) $\text{member}(x)$: tests whether $x \in A$
4) $\text{card}$: yields the size $n$ of $A$
5) $\text{min}$: yields the smallest element in $A$
6) $\text{max}$: yields the largest element in $A$
7) $\text{successor}(x)$: yields the smallest element in $A$ strictly larger than $x$
8) $\text{predecessor}(x)$: yields the largest element in $A$ strictly smaller than $x$

There are also composite instructions like:

1) $\text{extract} – \text{min}$ : yields the smallest element in $A$ and removes it from $A$
2) $\text{allmin}(x)$ : removes all elements $\leq x$ from $A$

Finally there is an instruction which only can be understood in terms of the binary representation of the numbers in $U$:

---

\(^1\) Due to the IEEE two column format used by ISCIM 2013 the reader will have to use a magnifying glass when reading a printed version; however the proceedings are digital anyhow.
neighbour(x) which yields either the predecessor or the successor of x in A, selecting out of these two possible answers the result which has the most significant bits in common with x. The purpose of this instruction will become clear when we consider the elements in U to be the leaves of a binary tree. Then the neighbour of x is the element in A which is closest in distance when traversing the edges of this tree.

These instructions have preconditions in order to be properly defined: inserting an element already present in A shouldn’t be allowed. Similarly removing an element which is not in A is illegal as well. If A is empty, min and max are not defined, and neither is the successor of an element x ≥ max. It is left to the programmer to ensure that these errors either don’t occur or are resolved by an appropriate action.

When implementing the above instructions there is a conflict between the efficiency requirement of the instructions best supported by direct access like member, insert and delete, and the instructions involving the ordering which are best supported by a list structure like min, max, and predecessor or successor for elements in the set. When implemented as an array or a bit string, member, insert and delete take constant time, while the order based instructions require linear time in the worst case; on linked lists the situation is reversed.

In order to reduce the complexity of the complete set of instructions several ingenious data structures where invented. Examples existing in 1974 are AVL-trees, 2-3 trees, heaps, etc. [2] yielding time complexity $O(\log(n))$ for these instructions, or subsets thereof. A well known example of such a subset is a priority queue, a data structure supporting insert and extractmin with processing time $O(\log(n))$ when implemented on a binary heap.

The data structures mentioned above manipulate elements based on binary comparisons; their behaviour is independent of the size of the Universe $u$ and therefore they work also for infinite universes, like the set of real numbers. As a consequence the time complexity $O(\log(n))$ is perfectly reasonable, given the fact that with the above instructions (in fact just the two priority queue instructions are sufficient) we can sort a set of real numbers, and it is well known that for sorting numbers an $\Omega(n\log(n))$-lower bound holds for comparison based algorithms.

However, in 1974 is was known that this lower bound on sorting numbers didn’t apply to non-comparison based algorithms, as illustrated by the linear time complexity of sorting methods like radix sort and bucket sort [22]. Therefore the question whether the processing time $O(\log(n))$ for our instruction repertoire could be improved by going beyond the comparison based model, was still open. The stratified trees provided a positive answer to this question, substituting the processing complexity $O(\log(n))$ by $O(\loglog(u))$ where $u$ is the size of the Universe rather than the size of the set being manipulated. Consequently, the improvement vanishes when the set A is sparse in the sense that $n = O(\log(u))$. But in areas like graph algorithms where the set A consists of the nodes or edges in the graph, the set is non-sparse so an improvement indeed is achieved.

The rest of this paper is structures as follows: section II presents the two approaches towards obtaining the $O(\loglog(u))$ improvement, the recursive cluster galaxy decomposition and the binary search on levels. Section III describes the state of machine based complexity theory around 1974, and explains the prohibition of multiplicative instructions in the RAM model. In section IV the binary search on levels approach is explained, but for full details the reader is referred to the original source [48]. Section V describes the developments during the first five years when I still was working on stratified trees myself; how the original design was improved and how the problem of super-linear space consumption was solved. It also tells how the recursive decomposition approach returned to the scene. Code fragments describing this approach are presented; this approach has become the dominant one today. Section VI gives a (necessarily incomplete) survey of work on stratified trees by other authors, involving applications, improvements and lower bounds. Section VII gives a conclusion.

II. Two approaches

There are two roads towards obtaining the processing time improvement of $O(\loglog(u))$. It is important to understand these approaches and the difference between them.

A. The recursive cluster galaxy decomposition approach

Today the structure of the stratified trees is best understood in terms of a recursive decomposition of the Universe: the Universe consists of a cluster of galaxies both of size $\sqrt{u}$. Each Galaxy covers the elements of some interval within the universe, and the cluster represents the ordered sequence of these intervals. Both cluster and galaxies can be decomposed recursively until the universe has become sufficiently small. In theoretical descriptions this means that the recursion terminates at universes of size 2; more practice oriented approaches let the recursion terminate at universe size equal the word size of typical computers like 16, 32 or 64. Furthermore for each (sub)structure we maintain the number of inserted elements and the largest and smallest element in a few local variables.

The idea behind this recursive approach is simple. For small universe size (bottom case of the recursion) all instructions take constant time. Similarly one can simply store all the elements in the set $A$ in a few local variables as long as $n$ is small (say $n < 2$). Consider now the case that we want to insert an element $x$ in $A$ at a stage where $A$ contains already some elements. We find the galaxy where $x$ should be inserted, and two possible cases arise: either this galaxy is already populated and it suffices to insert $x$ in this galaxy by a recursive call, or this galaxy is still uninhabited and we must insert this galaxy in the cluster first, but then the action of inserting the element in this newly inserted galaxy will be trivial since it will be its only inhabitant.

A similar analysis holds for the delete instruction; if we remove the last element from some galaxy this removal itself is trivial but then the galaxy itself should be removed from the cluster; otherwise the element only has to be removed from its galaxy. For the successor instruction the decomposition considers two cases as well: either the element is smaller than the maximal element in its galaxy and then the successor is to be found inside this galaxy, or the successor is the smallest element in the successor galaxy in the cluster.
Inspection of these decompositions shows that each instruction is implemented by similar instructions on the cluster and galaxy level (one of each) which would result in an estimate of the processing time \( T(u) \leq T(\sqrt{k}) + T(\sqrt{n}) + O(1) \) and this recurrence solves to \( T(u) = O(\log(u)) \) which means that there is no improvement over the traditional comparison based structures. However our analysis shows that one of the two recursive calls is always of a special type which can be performed in constant time. Hence the time estimate becomes \( T(u) \leq T(\sqrt{n}) + O(1) \) which resolves to \( T(u) = O(\log\log(u)) \).

**B. The binary search on levels approach**

Regardless the simplicity of the strategy as sketched above, history shows that the stratified trees originally were designed using a completely different approach. This second approach is best described as a binary search on levels strategy.

In this approach the elements of the Universe \( U \) are represented by the leaves of a complete binary tree of height \( h = \log(u) \). At each element of the tree (both at the leaves and the internal nodes) we have a mark-bit. A set \( A \) is represented by marking its elements together with all internal nodes on a the path from a marked leaf to the root of the tree as illustrated in figure 1. The marked nodes will be called present in the sequel.

![Fig. 1. Example of a five-element set represented using mark-bits](image)

Additionally, we build a doubly linked list connecting the leaves corresponding to the elements of \( A \). This data structure evidently supports the instructions member, card, min and max in constant time.

\( \text{insert}(x) \) is implemented by traversing the path from the leaf representing \( x \) towards the root until a marked internal node \( v \) is encountered. This node \( v \) will become the lowest branch point on the path of the newly inserted element \( x \). All nodes traversed during this walk are marked. Subsequently starting at this already marked node a downward traversal along marked nodes is performed, always selecting the rightmost (leftmost) marked son, depending on whether leaf \( x \) is a right (left) descendant of node \( v \). The leaf thus found is the neighbour of \( x \) in the set \( A \). It is either the predecessor or the successor of \( x \) depending on whether \( x \) is a right or left descendant of \( v \). Having found this neighbour, the leaf \( x \) now can be inserted in the doubly linked list in constant time.

In order to delete an element \( x \in A \), a similar traversal starting at leaf \( x \) is performed until a branch point (an internal node with two marked sons) is encountered. Nodes encountered during this traversal are unmarked. Next, leaf \( x \) is removed from the doubly linked list in constant time.

In order to compute the predecessor and successor of \( x \), a similar upward traversal is performed, which yields the lowest branch point. A downward traversal subsequently takes us to the neighbour, from which the successor or predecessor can be found in constant time in case we happen to arrive at the wrong side of \( x \) in the sequence of leaves.

The time consumed by these traversals is bounded by the height of the tree \( h \). Therefore the resulting processing time is \( O(h) = O(\log(u)) \). But the actual time used is determined by the distance of this lowest branch point from the leaves.

The simple observation that in the above approach time saved by not traversing all the way up to the root could be used for spending more time at the lowest branch point encountered, allowed me to improve the worst case time bound. If the lowest branch point is encountered at level \( k \) above the leaves (counting from the leaves at level 0) we can spend the remaining \( h - k \) time steps by allowing \( h - k + 1 \) sons for all nodes at level \( k \). The consequence is that in the resulting tree of height \( h \) we are not representing a Universe of size \( 2^h \) but one of size \( 2^k \). When \( u = h \) we have \( h = O(\log(u)/\log\log(u)) \) and this yields an improvement over the \( O(\log(u)) \) bound.

Discussing this improvement to John Hopcroft it immediately became clear that further improvements could be expected. The strategy as explained above uses a trivial linear search strategy at this lowest branch point. If we organise the sons of this branch point in a binary tree, we can allow for branching degrees growing even faster: branching degree \( 2^h-k+1 \) at level \( k \) which would result in a time complexity \( O(\sqrt{\log(u)}) \). A true application of the Divide and Conquer heuristic propagated by the early researchers in the field of algorithm design suggests the use of the very structure I was designing, recursively, at the internal nodes.

There was yet another way of looking at the problem. In this approach the key step is the location of the lowest branch point on the path from a present leaf to the root, or the lowest present ancestor in case the leaf is not present. If these points could be located using binary search on the path connecting the leaf with the root the time complexity could be reduced from \( O(\log(u)) \) to \( O(\log\log(u)) \). Such a method of binary search on levels had been used previously with success by Aho, Hopcroft and Ullman for finding lowest common ancestors in trees [1].

By the end of 1974 a solution following these suggestions had been obtained [45]. Conceptually it is far more complicated than the recursive approach. Moreover I had to struggle with yet another complication, having to do with the required address calculations used by our algorithms. They use multiplicative instructions, and these were not allowed on the machine model I was using. This complication was, at this stage of the history, fatal for the further exploration of the recursive decomposition approach.

In order to understand why these multiplicative instructions were problematic we first must take a look at the machine
model used for Analysis of Algorithms when this research area was being created: the Random Access Machine.

### III. Machine based complexity theory around 1974

Computability theory as we know it today was developed in the years 1920-1940 with computational models described by Post, Church, Gödel, Kleene and above all the simple and widely accepted model invented by Turing. The idea of investigating also the time and space complexity of these models of computation originates from the 1960-ies. The concept of a "feasible computation" was introduced by Edmonds [13]. Hartmanis and Stearns [18] started the research on the complexity of the Turing machine model. Manuel Blum [8] created the extension of Recursion Theory known as Abstract Complexity Theory. By 1972 the fundamental Complexity Classes P, NP, PSPACE, EXPTIME, … had been defined and the P = NP ? problem became a part of our life.

None of these approaches provides definite answers for the real life programmer who wants to solve real problems on a real computer. The structural approach which focuses on the relation between the fundamental complexity classes, compares models based on mutual simulations with a polynomial time bounded overhead and a constant factor overhead in space [52]. However, for practical design and analysis of algorithms a quadratic improvement is highly relevant. In the abstract theory the situation is even worse: all models are basically equivalent, including the pathological ones.

When Knuth started his book series on the art of computer programming [21] he followed a different approach. He introduces in this book a simplified model of a real von Neumann computer called MIX. Algorithms are analysed by a precise estimate of the number of machine instructions on this MIX model. This immediately raises the question to which extent the results obtained by Knuth depend on the details of the MIX model.

When Aho, Hopcroft and Ullman wrote their textbook [2], they based their work on a compromise between these two extremes. They use the Random Access Machine (RAM) as described by Cook and Reckhow [9] as a model. It differs from the MIX by having a reduced set of instructions, but it is also a theoretical model since it has unbounded memory and word size. Furthermore they consider a stored program version of the model called RASP, and prove that the two models can simulate each other in real time.

They present their algorithms in a higher programming language called Pidgin Algol which makes the basic control features of programming like conditionals, loop constructs procedures, functions and recursion directly available. It was safe to do so since by 1974 it was well known how to compile such features in machine instructions. There were no hidden time overheads between the perceived instructions in the Pidgin Algol program and the resulting Machine code program on the RAM. What one looses is information on the precise constant factors involved, but that was felt to be less important; the asymptotic growth of the complexity was the target of the research program, as illustrated by the abundant invocation of the Landau $O(f(n))$ expressions.

The RAM model is basically a single register machine having an infinite memory of infinite length storage cells called words. The memory contents are non-negative integers. The instruction code includes loading constants, and the contents of memory locations using both direct and indirect addressing (where the index of the memory cell being loaded is itself the content of another memory location). For storing data in memory the machine can use both direct and indirect addressing. As flow of control the machine has unconditional and conditional jumps (testing for $= 0$ or positiveness of the accumulator contents). All arithmetic is performed on the single accumulator register. The arithmetic instructions allowed are addition, subtraction, multiplication and (integer) division. The programmer should beware for typical errors like the production of negative results or division by zero.

An earlier model of this type is the register machine introduced already in 1963 by Shepherson and Sturgis [39]. This model in its basic form only has increment and decrement as arithmetic instructions, and it lacks indirect addressing. It is equivalent to the Minsky multi-counter machine [30]. Still this model is Turing Universal, even if only two registers are being used.

For the time complexity Aho and his colleagues use two measures. In the uniform measure one just counts the number of RAM instructions performed. In the logarithmic time measure each instruction is charged for the number of bits in the arguments of the instructions, regardless whether these arguments are addresses or data. Similarly one can for a space measure either count the number of memory locations used, or weigh every location by the size of the values being stored there.

They prove that with respect to the logarithmic time measure the RAM and RASP model are equivalent to the multi tape Turing machine modulo a quadratic time overhead. But if the uniform time measure is used, there is a problem with the multiplication and division instructions, since they can generate exponentially large results in a linear number of steps. Therefore one should exclude multiplicative arithmetic when using the uniform time measure; if only additive arithmetic the uniform time measure and the logarithmic time measure are equivalent up to a quadratic overhead.

That there is a real problem here follows from results obtained by Pratt and Stockmeyer [35], Hartmanis and Simon [17] and Bertoni e.a. [7] or Schönhage [37]: the RAM model with multiplicative arithmetic and uniform time measure is equivalent to models of parallel computation, meaning that it can perform PSPACE complete computations in polynomial time. It is a member of the second machine class [52].

The Turing Universality of the two counter Minsky machine shows that just counting memory cells is not a good idea. Therefore the uniform space measure is never used, unless there is an explicit limit on the size of the values being stored. For the logarithmic space measure there is an issue on how to deal with storage cells which are unused during a computation while cells further down in the program are used. Also the size of the addresses of the memory cells should contribute to the logarithmic cost of such a cell. For details see [40].

The question remains which of these measures is suitable for analysing the complexity of the stratified trees. There
are several reasons for not using the logarithmic time measure. In the first place the main competitive algorithms use comparison based manipulation of lists and trees which are build using pointers. Frequently one just counts the number of comparisons. Address calculations hardly play a role and arithmetic is hardly used at all other than for counting the size of (sub)structures.

Another argument is that use of the logarithmic measure will entail an operation complexity $\Omega(\log(u))$ for just looking at the value being manipulated, so improvement to sub-logarithmic cost is impossible anyhow.

The space measure was a less interesting topic; for the tree based structures space is traditionally measured by the number of nodes and edges in the structure which amounts to the use of the uniform space measure.

However, using the combination of both uniform time and space measures comes with a price: it is no longer appropriate to use multiplicative instructions. These instructions were perceived to be more expensive than the additive ones, both in practice and in theory; the linear time multiplication algorithm of Schönhage [38] still had to be invented.

It is not difficult to see that multiplicative instructions are used in our recursive decomposition for address calculations. Assume that $u = v.w$ and that we divide the universe $[0..u - 1]$ in a cluster of $v$ galaxies each consisting of $w$ elements each. Then element $x$ will be represented by entry $x \mod w$ in galaxy $x \div w$; conversely entry $a$ in galaxy $b$ represents element $a.b + a$ in the universe. This shows that the relation between the arguments of our operations and those involved in the recursive calls are computed using the multiplicative operations.

Another construction we can’t use any more is a two-dimensional array; its indexing formula uses a multiplication.

However, in our algorithms we will in general only multiply and divide by powers of two. This makes the restriction even more unreasonable, since these arithmetical operations are realized by bit-shifts which are available on many real life computers. So we are facing a situation where theory seems to be more restrictive than practice.

Fact is that most authors discussing stratified trees, in particular when discussing the recursive approach, simply are disregarding this prohibition of multiplicative instructions; as long as the arguments don’t increase above the initial values they consider them to be allowed.

IV. IMPLEMENTING BINARY SEARCH ON LEVELS

The binary search on levels approach uses a complicated decomposition structure on the binary tree which was used in the silly method in section II. I will explain how this tree is decomposed in so-called canonical sub-trees, what sort of static information we need in order to navigate this tree, and how the dynamic set $A$ is represented in this structure by assigning appropriate values to fields located at the internal nodes of the tree. By way of example the code fragment for the neighbour instruction is presented and explained. Complete details can be found in the source reference [48].

A. Canonical sub-trees

For reasons of simplicity we assume in this section that $u = 2^d$; the height of the tree structure we will describe equals $h = 2^l$, and the number $l$ will be called the rank of the tree. The cluster of galaxies decomposition of the universe corresponds to dividing the tree in a top-tree of height $h/2 = 2^{l-1}$ and rank $l - 1$ representing the cluster whereas the bottom trees of rank $l - 1$ whose roots are actually the leaves of the top tree represent the galaxies. Each of these trees can be decomposed again in a top and bottom trees of rank $l - 2$. The trees of rank 0 are trivial: a root with two sons. The trees obtained in this recursive decomposition are called the canonical sub-trees in the papers where the stratified trees were first introduced.

The concept of rank is also ascribed to the levels in the tree and therefore also to the nodes in the tree. The leaves of a tree are located at level 0 and the root is at level $h = 2^l$. The rank of level $j > 0$ is the largest number $d$ such that $2^d \leq j$ and $2^{d+1} \not\geq j$. By default rank(0) = $l$ + 1. Note that if rank($j$) = $d$ we have rank($j + 2^d$) > $d$ and rank($j - 2^d$) > $d$; moreover the ranks of these two numbers are different. All numbers $i$ with $j - 2^d < i < j + 2^d$ and $i \not= j$ have rank($i$) < $d$.$^2$ The rank of some node $v$ is the rank of its level in the tree.

A canonical sub-tree of rank $d$ is a sub-tree of height $2^d$ with a root at some level $j$ with rank($j$) $\geq d$ and leaves at the closest level with rank $\geq d$. It follows that either the root or the leaves have rank equal $d$. If the root has rank $d$ the canonical sub-tree is called a bottom tree, and it is called a top tree otherwise.

Canonical sub-trees are furthermore divided in a left part and a right part, the left (right) part consisting of the root and its left (right) sons and their descendants within the sub-tree. This division adds another element of complexity to our structure which has no counterpart in the recursive cluster and galaxies approach; we shall see in the sequel of this section why this further division had to be added.

A final concept is the reach of some node $v$; if rank($v$) = $d$ node $v$ is leaf of an uniquely determined top tree of rank $d$ and the root of a bottom tree of rank $d$. The root of this top tree and the leaves of this bottom tree have rank $\geq d$. Now reach($v$) consists of this bottom tree together with the left or right part of the top tree containing $v$. See figure 2. This figure introduces also names for the relevant sub-trees: $C(v)$ is the rank $d + 1$ canonical sub-tree containing $v$, $UC(v)$ is the top-tree of $C(v)$ and $LC(v)$ is the bottom-tree of $C(v)$ having $v$ as its root. The left and right parts of $LC(v)$ are named $LLC(v)$ and $RLC(v)$.

B. Static information

The next topic to consider is how we can navigate in this tree and its canonical sub-trees. A useful way of indexing nodes in a binary tree is to assign index 1 to the root, and assigning indices $2.m$ and $2.m + 1$ to the left and right son of the node with index $m$. Moving $j$ levels up from the node with index $m$ yields node $m \div 2^j$. The $2^j$ descendants of node $m$ have indices $2^j.m, \ldots, 2^j.m + 2^j - 1$. The leaves of our

$^2$The idea of such a rank function is well illustrated by the lengths of the lines on the British side of a ruler; an abstract definition which will work for general heights $h$ is sketched in [48].
complete tree which will represent the elements of the universe have indices $2^b, \ldots, 2^b + u - 1$.

The problem is that these expressions which allow us to freely navigate in the tree use multiplicative operations; once again - these operations look perfectly innocent since they are easily evaluated by bit-shifts - but still they were considered illegal in the RAM model with the uniform time measure in 1974. Therefore another solution was invented: pre-compute sufficiently many of these navigation steps in advance, so that you can use table look-up when you need to perform this step.

It turns out that the required navigation steps for our algorithms to work are the ancestors at levels 1, 2, \ldots, $2^d$ of some node $v$ of rank $d$; these are precisely the roots of those canonical sub-trees of rank 0, 1, \ldots, $d$ having $v$ as a leaf. In order to have a format which doesn’t depend on the rank of some node it was decided to store at node $v$ pointers toward the roots of all canonical sub-trees containing $v$. There are $\log \log (u)$ pointers of this type at every node yielding a space consumption of $\Omega(u \log \log (u))$ for storing just this static information which will never change during the manipulation of some subset $A$. The fact that you don’t need pointers to ancestors at level $2^l$ for $l > \text{rank}(v)$ doesn’t help us since the leaves at the bottom of the tree all have maximal rank and represent a majority of the nodes in the tree. Using a rank dependent format would save at best a constant factor in space.

This solution to our navigation problems also explains why the original implementation of the stratified tree used pointers rather than addresses; the stratified tree is a solution of our set manipulation problem in the pointer based model of computation, and this is relevant in relation to lower bound results which we will encounter later in this survey.

C. Dynamic information

In our silly solution as illustrated in figure 1 we represent some set $A$ by marking all leaves representing the elements of $A$ and all ancestors of some marked node. Marked nodes are called to be present. The various operations were implemented by locating either the lowest branch-point (a present node with two present sons) above some present node or the lowest marked ancestor above some non-present node (for the Insert and Neighbour instructions); this node will be called a focal-point in our further discussion. The key challenge now is that we don’t want to spend time $\Omega(\log (u))$ by a linear traversal of these ancestor paths connecting leaves with the root; instead we want to use binary search on levels.

Consider some canonical sub-tree $T$ of rank $d$ whose root $v$ is present. There are two cases to consider: either there is just a single present leaf in this canonical sub-tree, and one might as well store at $v$ some information which will allow us to locate this leaf; otherwise there are at least two present leaves in $T$ which means that somewhere in $T$ a branch-point exists (it may be the root $v$ itself). Consider now the leaves of the canonical sub-tree $T'$ of rank $d - 1$ with root $v$ (the top-tree in the recursive decomposition of $T$). If there are at least two present leaves in $T'$ there will be a branch-point inside $T'$; otherwise if $w$ is the unique present leaf in $T'$ there will be a branch-point inside the canonical sub-tree of rank $d - 1$ with root $w$ (one of the bottom-trees in the recursive decomposition of $T$). In order to determine which of the two cases arises we should store some information at the present nodes on this half-way level where $w$ is located.

This leads to the concept of an active node. An internal node is active if information is stored there which is required for locating some focal-point or branch-point during some possible operation. A leaf is active if it is present, and by definition the root is active when $A$ is non-empty (in our original implementation this was achieved by stipulating that $u - 1 \in A$ permanently, but this restriction turned out to be superfluous).

Whether some internal node should be active or not is determined by an invariant called the properness condition [48]: An internal node $v$ is active iff there exists a branch-point in the interior of the reach of $v$; i.e. there exists a branch-point $u \in R(v)$ which is neither the top nor a leaf of $C(v)$.

At an inactive node no information is stored; its fields will have default values.

The information stored at active leaves consists of two pointers towards their present predecessor and successor in the set $A$ as far as they exist, together with a mark-bit indicating that the leaf is present. For an active internal node $v$ the data consists of four pointers and a mark-bit \footnote{Our implementation used different record formats for leaves and internal nodes in the tree; the case-dependent record format in PASCAL made it easy to implement this difference.}: the pointers $\text{llc}(v)$ and $\text{llmax}$ store the least and largest present leaf in $\text{llc}(v)$ if such leaves exist, and $\text{lrmin}$ and $\text{lrmax}$ do so for $\text{lrmax}(v)$. Note that these are leaves in $\text{llc}(v)$ which means that they are not necessarily leaves of the entire tree. The mark-bit $\text{ub}$ indicates whether there exists a branch-point in between $v$ and the top of $\text{llc}(v)$. Note that an active internal node is present and hence it must have at least one present descendant leaf in $\text{llc}(v)$, so at least two of the four pointers $\text{llcmin}$, $\text{llcmax}$, $\text{lrmin}$ and $\text{lrmax}$ are defined. It is possible to have $\text{llcmin}(v) = \text{llcmax}(v)$ which indicates that there is no branch-point in between $v$ and $\text{llcmax}(v)$; if in this situation also $\text{lrmin}(v)$ is defined the node $v$ itself is a branch-point. Figure 3 illustrates the assignment of values to these pointers at internal nodes for a three-element set $A$.

The representation of a set $A$ now is determined as follows: the leaves representing the elements of $A$ are declared to be
Fig. 3. Example of the internal pointers for a three-element set

present and active and the pointers in the doubly linked list are
given their appropriate values. If \( A \neq \emptyset \) the root is declared to
be present and its four pointers are assigned their proper values.
Next for internal nodes, processing them by rank decreasing
from \( h - 1 \) to 0 it is checked whether they should be made
active according the properness condition, and if so their fields
obtain the required values.

D. Operations

Given the complexity of the representation of the dynamic
set \( A \) presented above it should come as no surprise that
the actual implementations of the operations are complex
as well. Certainly if compared with the code which results
from implementing the recursive cluster-galaxy decomposition
scheme which can be found in the textbook presentations
of my structure [10], [16], [27], [60]. The key operations
which do the work are insert, delete and neighbour. A full
explanation of these operations would require me to copy
the entire section 5 from our original publication [48] which
would fill many pages. Therefore I rather refer the reader to
the original sources. I include and discuss the code for the
neighbour instruction, directly copied from [48].

The programming language used is PASCAL [56]. This
code was produced by my students Kaas and Zijlstra [25]
who on behalf of this contribution earned co-authorship on
the journal publication of my FOCS 1975 paper [47].

The instruction neighbour has only a single argument when
called from the outside; however, in its recursive structure
it requires additional parameters for navigational purposes.
Hence the overall structure of this function consists of the
declaration of an internal function \( \text{neighb} \) which actually
performs the work, which is subsequently invoked on the
appropriate arguments.

\footnote{evidently this is not the algorithm which was used for maintaining the
representation of a dynamic set \( A \) in our implementation.}

The arguments of this internal function are: \( \text{leaf} \) (the
ancestor of the original argument at the bottom level of the
canonical sub-tree under consideration), \( \text{top} \) (the top of this
sub-tree), \( \text{pmin} \) and \( \text{pmax} \) (the leftmost and rightmost present
leaves in the half-tree containing leaf) and \( \text{order} \) (the rank of
the sub-tree). Of the four pointer fields at the node \( \text{top} \) we first
have to inspect those which are located on the same side as
\( \text{leaf} \); in order to simplify the code this choice has been hidden
into the auxiliary functions \( \text{mymin} \), \( \text{mymax} \), \( \text{yourmin} \) and
\( \text{yourmax} \). The auxiliary functions \( \text{minof} \) and \( \text{maxof} \) yield
the least and largest leaves in the entire sub-tree. The left-right
comparison between two nodes is performed by comparing
the position field of the nodes which is part of the static
information and filled with the in-order number assigned to
the node when the tree is initialized in an in-order traversal.

There are several cases where the neighbour can be com-
puted without a recursive call: when the half-tree containing
leaf doesn’t contain a present leaf (\( \text{pmin} = \text{nil} \)), or when leaf
is actually its unique active leaf (\( \text{pmin} = \text{pmax} = \text{leaf} \)); a
third case is when there are several active leaves but leaf is
outside the range between \( \text{pmin} \) and \( \text{pmax} \).

Otherwise we must inspect the ancestor \( \text{hl} \) of \( \text{leaf} \) on
the level half way in between \( \text{leaf} \) and \( \text{top} \). In case \( \text{hl} \)
has no (other) present leaf (indicated by \( \text{hl} \) being inactive or
\( \text{minof}(\text{hl}) = \text{maxof}(\text{hl}) = \text{leaf} \)) a recursive call is
performed on the top-tree with root \( \text{top} \) yielding the neighbour
at that level \( \text{nb} \). The neighbour of \( \text{leaf} \) now is \( \text{minof}(\text{nb}) \) or
\( \text{maxof}(\text{nb}) \) depending on the position of \( \text{hl} \) compared to \( \text{nb} \).
If \( \text{hl} \) has another present leaf a recursive call is performed on
the bottom tree with root \( \text{hl} \) which yields the neighbour at the
right level directly.

As promised: this procedure either terminates in con-
stant time or performs a single recursive call with \( \text{order} \)
decreased by 1, and this suffices for obtaining time complexity
\( O(\log\log(u)) \).

The reader should note however that this procedure in
fact behaves differently from what one would expect if one
follows the recursive cluster-galaxy decomposition approach.
It has to do with the conditions allowing the \textit{neighb} function to terminate without a recursive call. The cases where there is no present leaf or when leaf is the unique present leaf in the recursive approach will look at the entire sub-tree and not just the half-tree to which \textit{leaf} belongs; the very concepts of left- and right sub-trees are simply not defined in the recursive approach.

The operations insert and delete are structured in a similar fashion, but they are even more intricate since they have side effects on the structure.

Finally some remarks on the initialization routine. It performs a mixture of a pre-order and an in-order traversal of the tree being constructed and builds what is to become a proper representation of the empty set \(A\). When creating a node it must assign proper values to the father pointers and to the position fields. What are the proper values depends on the rank and the level of a node. Multiplicative instructions for evaluating the rank for a given level are prevented by pre-computing these ranks and storing the results in a table. Details can be found in [48] but the code is difficult to understand due to various hidden side effects.

V. THE DEVELOPMENT OF THE ORIGINAL, AND THE REVIVAL OF THE RECURSIVE DECOMPOSITION APPROACH

The code in the journal paper discussed in the previous section represents the final stage of a sequence of presentations, starting with the 1974 Cornell report. I describe in this section how the perspective on what operations are essential gradually changed during this development. Next we look at Knuth’s classroom note from 1977 and its importance for reviving the recursive decomposition approach.

A. The improvements of the original presentation

The first papers I published on the stratified trees all are based on the binary search on levels approach sketched in the previous section. The oldest paper is the Cornell technical report [45]. The representation used in this original paper is almost identical to the one used in the eventual journal paper [48], but the perspective on the operations is quite different. The primitive operations are the location of the lowest branch (present) point above a given (non) present node (called the focal point previously), and the deletion (insertion) of the path segment between the argument leaf and the focal point. The instruction neighbour is mentioned but its importance remains unrecognised. And to my own dissatisfaction I was unable to design a recursive version of the code for the delete instruction.

The second paper was written for a seminar talk at IRIA in the spring of 1975 [46]. The same representation is used but the operations are different and the neighbour instruction has become more important. Hardly any code fragments are given.

Around this time I had assigned the task of programming a real implementation of my structure as a possible assignment for a term project to a group of students in an advanced programming course which I was teaching with some colleagues during the spring of 1975. Two students, Kaas and Zijlstra, accepted the challenge and eventually produced the PASCAL code contained in a report of the department of Mathematics in my institute [25]. Aside from repairing some hideous errors in my original code and contributing some valuable ideas they found a recursive version of the delete instruction. Having this code available I submitted the result for the FOCS 1975 conference in Berkeley, where it was accepted and published [47]. The binary search on levels approach still is used, but in the expository introduction the idea of the recursive cluster galaxy approach is mentioned. Nothing is done with this idea. The FOCS paper includes code fragments from the Kaas and Zijlstra program.

The journal paper [48] actually merges the two previous reports and Kaas and Zijlstra became co-authors. A preprint is dated December 1975.

In August 1976 a symposium was organised on interfaces between computer science and operations research. For this symposium I gave a presentation on developments in data structures. The text of this presentation was eventually published in 1978 [50], but since this is two years later than the presentation it is uncertain to what extent its contents correspond with the talk given. Fact is that in this paper the cluster galaxy decomposition idea is used as a starting point and the stratified trees are presented as being the result of unwinding the recursion in the recursive data structure implementing this decomposition.

In 1977 I found a solution [49] to another nasty feature of my original structure: its super-linear space consumption. It is based on the cluster galaxy decomposition idea used in a non-recursive way. The structure of the cluster and the galaxies are different: the cluster representing \(m\) galaxies of size \(k\) is implemented by the original structure, but the galaxies are implemented by some structure using linear space with a worse time complexity. A binary tree based structure like my silly structure from figure 1 is OK, but even a linear list will work here. When one realises that every instruction is decomposed in an instruction operating on the cluster level and one on the galaxy level, and uses the fact that the storage being used is the sum of the space for the cluster and that for the galaxies, one obtains the following estimates:

\[
\text{Time: } O(\log\log(m)) + O(k)
\]

\[
\text{Space: } O(m\log(m)) + mO(k)
\]

Taking \(m = u/\log\log(u)\) and \(k = \log\log(u)\) solves the problem.

The code fragments from this paper illustrate the simplicity of this approach. See figure 5. One of my worst mistakes is that I, having arrived at this idea of a hybrid mixture of data structures, didn’t use this for other purposes; it became an important tool for designing dynamic data structures in areas like computational geometry and was used extensively by my colleagues from Utrecht Jan van Leeuwen and Mark Overmars [32].

B. Knuth’s classroom note and its impact on reviving the recursive decomposition approach

I had exchanged papers and reprints with Don Knuth. In reply of the preprint of [49] he send me a copy of his classroom note [23]. To my knowledge this is the first written version of the fully recursive cluster and galaxy decomposition approach.
Multiplicative instructions are used without hesitation. Both the asymptotic complexity and practical implementations for small universes are considered (when $u = 2^{20}$ it consumes 242937 bytes, sufficiently small to fit in core on the IBM 360). For this case a three level decomposition with explicit addresses and bit-shift operations is presented; see figure 6 for the relevant code fragments. The programming language used is inspired by Simula. Note also the final sentence in this page: this is the source of the famous quote from Knuth on program correctness.

The final paper on the stratified trees I wrote myself is another contribution to a colloquium at the Mathematical Centre in Amsterdam [51]. This paper is written in Dutch. Three sections discuss the stratified trees. I first describe the $O(\log(u))$ recursive cluster galaxy decomposition approach following the algorithms as presented in figure 5. The next section sketches the tree model as presented in [48], [49]. In the final section I present my own version of the fully recursive $O(\log\log(u))$ implementation suggested by Knuth with a full reference to his classroom note. I show some code fragments, this time written in some version of Algol 68 [55], in figure 7 and figure 8.

In figure 7 is shown how I dealt with the multiplicative instructions. Thinking in terms of bit-strings they are called head, tail and conc respectively, but since the implied galaxy size $q$ depends on the recursive level there is another hidden argument called order. Therefore head and tail become binary operators, and conc becomes a procedure since Algol 68 doesn’t allow to write ternary operators as operators. Actually these operations were implemented by table look-up. Note that we don’t use a two-dimensional array since its indexing formula would reintroduce the multiplicative instructions we want to eliminate. Instead we use an array of arrays. The final code fragment shows how these tables are precomputed by counting; another code fragment using side effects on $s$.

In figure 8 is shown how I dealt with the multiplicative instructions. Thinking in terms of bit-strings they are called head, tail and conc respectively, but since the implied galaxy size $q$ depends on the recursive level there is another hidden argument called order. Therefore head and tail become binary operators, and conc becomes a procedure since Algol 68 doesn’t allow to write ternary operators as operators. Actually these operations were implemented by table look-up. Note that we don’t use a two-dimensional array since its indexing formula would reintroduce the multiplicative instructions we want to eliminate. Instead we use an array of arrays. The final code fragment shows how these tables are precomputed by counting; another code fragment using side effects on $s$.

The next code fragment is the first insert procedure which is invoked when inserting a first element; it only affects the three integer fields, leaving the pointers untouched. The general insert procedure uses the case construct of Algol 68 depending on the current cardinality of the structure. If that equals 0 first insert is invoked and the procedure terminates; if the cardinality equals 1, the existing element must be inserted in

The next code fragment is the first insert procedure which is invoked when inserting a first element; it only affects the three integer fields, leaving the pointers untouched. The general insert procedure uses the case construct of Algol 68 depending on the current cardinality of the structure. If that equals 0 first insert is invoked and the procedure terminates; if the cardinality equals 1, the existing element must be inserted in
the recursive structure first, but this generates an invocation of first insert both at the top and the bottom level and therefore it takes constant time. If the cardinality equals 2 or more this step is skipped. Now we are ready for inserting the new element. The galaxy to which it belongs is retrieved; if this galaxy is non-empty the element is inserted in this galaxy by a recursive call; otherwise the galaxy is inserted in the cluster and the element is inserted in the galaxy using first insert. Finally the min, max and card fields are adjusted.

In this code fragment it is assumed that the recursive data structure has been built previously; no initialisation procedure is described in this paper.

```
node done = instantiate node, min, max;
ref done top;
ref 1 done bottom);

Recursive data structure definition

proc first insert = (ref done queue, list any) void;
  (ord of queue = 1 if min of queue = any;)

proc insert = (ref done queue, list any, order) void;
  if order = 0 then bottom case of nonempty;
  final call for small universe &
  
  else min ord of queue = 1 in
  (first insert (queue, any);)
  next done (old = new node for existing old #
  if ord old = min of queue;
  old old tail = order old tail, old = old = order head old;
  ref done all = top of queue, box = bottom of queue; find all gall;
  first insert (all, old gall, first insert (best, old tail))
  next done # now we are ready for new nodes #
  ref done all = top of queue, box = bottom of queue; find all gall;
  if ord all = 0 then
  next insert (best, meet, order) / else first insert (best, meet, order) /
  next done (new = new node, order = 0) /
  next done (new = new node, order = 0) /
  next done (new = new node, order = 0) /
  next done (new = new node, order = 0) /
  next done (new = new node, order = 0) /

  Code of the insert instruction
```

VI. LITERATURE ON STRATIFIED TREES: IMPROVEMENTS AND APPLICATIONS

The day I am writing this section (August 26 2013) Google scholar informs me that there are 550 papers referencing [48] and 408 referencing [49]. In fact [48] used for several years to be number 1 on the result list if one queries for the name van Emde Boas in Google scholar.

Some of these involve applications leading to a more efficient algorithm for some problem; others involve the complexity of the set manipulation problems themselves. The variety of machine models under consideration has grown. Finally there are lower bound proofs for the operations holding for various restricted machine models, some of which indicating that under specific conditions the \( \Omega(\log\log(u)) \) processing time for the instructions is optimal. There are other scenarios where this lower bound is invalid and where the time for the operations has been improved.

---

5. in a footnote in [20] D.B. Johnson refers to [50] as a less inaccessible simplified exposition by the original author.
6. Only this year I lost this position to another van Emde Boas who happens to be my brother and who now occupies the first two places with papers on the taxonomy of epileptic seizures. No bad feelings; the 7340 results produced by Google scholar seem all to originate from seven authors, all close family members.
The mere fact that some paper includes a reference to our papers doesn’t entail that this paper also discusses it; it may include this reference since it was referenced in some other paper which is discussed. Moreover the context could be negative (look guys, we improved this result). This literature is far to extensive to be covered in this section. Therefore I have selected just some noticeable examples which I could access from home. And I won’t attempt to give much of an explanation since I hardly understand the proofs myself. As told before, I have not looked seriously at stratified trees for over three decades.

A. Applications

In the four original papers there is a section dedicated to reductions between set manipulation algorithms, both in the on-line case where the instructions are to be performed upon arrival, and in the off-line case where the entire instruction sequence is given in advance and may be preprocessed. Reductions should yield at worst a constant factor time overhead and program expansion (in the off-line case). The problems considered are insert and extract min, insert and allmin, and union find. The results consist of establishing the impact of the newly invented stratified trees on the earlier results bij Aho, Hopcroft and Ulman [1],[2].

The final journal paper [48] moreover considers the problem of building a mergeable heap based on our structure. This means that many sets are being manipulated and that the instructions union and find are added. An approach based on Tritter trees with stratified trees at the nodes is proposed. Tritter trees are the trees used by the union-find algorithm by Tarjan [41]. This extends a section in the FOCS paper where the problem of manipulating multiple sets is considered.

There is an evident problem with manipulating multiple sets: the space requirement is multiplied by the number of sets being used leading to a worst case space complexity $O(u^2 \log \log (u))$, which is outrageous. I suggest a solution to this problem by a proposal amounting to dynamic allocation of the structure: allocate only those parts of the stratified trees which you actually need. Analysis shows that you always will need the top tree, but you can allocate the bottom trees at the time their root becomes active. Iterating this idea the space requirement can be reduced to $O(u^{1+\epsilon})$ for every positive $\epsilon$.

With hindsight you can observe that this idea also works for a single stratified tree, so why didn’t I use this approach for solving my problem of the super linear space? Once again it has to do with the prohibition on multiplicative instructions on the RAM and the way we had coped with this problem. The space for the static structure (which in principle can be shared between multiple copies of the stratified tree) already uses space $(u \log \log (u))$ and so do the precomputed tables used for the address calculations in the case of the recursive decomposition approach as presented in [51].

One of the first applications of my structure by others is the result on the longest common subsequence problem from Hunt and Szymanski [19]. It is a dynamic programming method in string manipulation algorithms, which uses priority queue operations on positions in the strings. Note that in the paper as published, the resulting $\log \log (n)$ factor is not claimed in the main theorem, but is added in a second theorem called a theoretical result. At the time the paper was submitted (May 1975) the authors only had access to the Cornell report.

In order to have meaningful applications one needs to consider problems where the universe is bounded and the dynamic set being manipulated is not sparse. Such situations arise for example in Graph theory, Dijkstra’s shortest path algorithm being a representative example of an algorithm using a priority queue. The size of the universe evidently depends on the nature of the edge weights. Such an approach is considered in the paper by Ahuja, Mehlhorn, Orlin and Tarjan [3], but the result in this paper shows that further improvements are possible. This result was further improved by Thorup [43].

An application in computational geometry was given by de Berg, van Kreveld and Snoeyink [12]. This result was used subsequently for packet switching on the internet by Lakshman and Stiliadis [26].

B. Improvements

The first improvement known to me was proposed by Donald B. Johnson [20]. He improves the operation time from $O(\log \log (u))$ to $O(\log \log (d))$ where $d$ denotes the distance in the universe between the argument and its neighbour in the set $A$. The idea of the improvement uses the binary search on levels approach. Rather than finding the focal point of an operation by top-down binary search one should first inspect the ancestors of the argument at levels $1,2,4,16,\ldots$ (these are the roots of the canonical sub-trees having the argument as leaf) until one locates an ancestor above the focal point (something which can be recognized from the values being stored there).

As observed by the author: there seems to be no way to obtain this improvement when the recursive cluster galaxy decomposition approach is used.

The key reason why space $O(u)$ is required even in situations where $n << u$ is that the direct access of an array element, given its index, requires that the storage space for the array is allocated; as is well known from a famous exercise in [2] it is not needed to also initialize this space. The solution of this problem with direct access is to use dynamic perfect hashing. Fredman, Komlós and Szemerédi [14] have shown that you can do so without loss of the $O(1)$ access time.\footnote{Frequently the search leads to the discovery that the source is hidden behind a pay-wall, and therefore only very incomplete information on the contents becomes available.}

Dan Willard proposed this approach [57] in 1983; together with Fredman this evolved in the design of the Fusion trees [15]. See also the survey paper [58].

Mehlhorn and Näher [29] obtained the optimal result combining time $O(\log \log (u))$ and space $O(n)$, improving on Willard who had obtained these bounds only for a static set $A$.


\footnote{In fact I have used the results of this paper in some other part of my research [40], but never had the idea to use it for improving my stratified trees.}
better bounds, but it all becomes very complicated. The word-
length of the RAM processor becomes another relevant pa-
parameter, and general $AC_0$ instructions on integers are allowed.
This is far far away from my original intuitions. For example,
lemma 2.1 in [42] cites a result from [5] where two sorted
lists of at most $k$ keys can be merged in time $O(\log(k))$; the
snap being that the two lists are stored in a single RAM word,
and the result should be stored in two RAM words. Evidently
this will require the use of instructions way beyond what was
considered to be allowed in 1974.

For improvements in completely different directions: Wang
and Lin [59] obtain amortized processing time $O(1)$ on a
pipelined system with $O(\log\log(u))$ layers.

Finally the question remains whether any of these ideas
will work if used with real life computers on regular size data;
this has been investigated by Dementiev, Kettner, Mehnert and

C. Lower bounds

During one of our encounters in the 1980-ies Kurt
Mehlhorn told me that his students had proven that my
structure is time-optimal, at least in the pointer based model of
computation used in the original presentation. I presume that
he refers to the result by Mehlhorn, Näher and Alt presented
in [28].

This paper considers the union-split-find problem, but this
problem is equivalent to the insert-delete-predecessor problem.
Just consider the points in $A$ to be the endpoints of intervals
which are split and merged by inserting or removing new
endpoints, and which are to be identified by their right bound-
ary point. The lower bound is proven using graph theoretic
arguments. One investigates the directed graph formed by the
pointers which can be created, destroyed or moved around, but
other operations on pointers have no meaning.

The situation changes when one moves to the cell-probe
model. Here the RAM computation is analysed as a big
decision tree where at any step inspecting some value stored
in a RAM memory location one branches depending on the
possible value stored there. The branching degree at these
decision nodes is exponential in the word-length of the proces-
sor. The arguments become information theoretical and involve
sometimes the use of communication complexity. An example
of this approach can be found in the paper by Peter Bro
Miltersen [31] who improves on earlier results by Ajtai [4].
He obtains a lower bound $\Omega(\sqrt{\log\log(u)})$.

The final papers I want to mention, are papers by M.
Pătraşcu and M. Thorup. In their paper [33], presented at
STOC 38 in 2006, they give a refined lower bound analysis
in the cell probe model. The parameters involved are both the
universe size $u$ and its logarithm $l$, the word length $w$,
the number of elements in the set $n$ and the number of available
memory words $S$. The result shows that at least on part of this
parameter space the stratified trees are optimal. The results are
extended to a randomized model in [34].

VII. CONCLUSION

It is not exceptional that results in computer science or
mathematics do not obtain in their early publications the
transparent treatment which will eventually surface in the
literature. In most cases this is the result of the authors not yet
fully understanding the problems at hand. There are however
also instances where the original author is guided by scientific
social constraints into a direction which makes it hard if not
impossible to express his original intuition. A prime example
of this process is the history of Savitch’ theorem [36] stating
that $PSPACE = NPSPACE$. In his original publication he
was prohibited by the reviewers or the editor to use a recursive
procedure, and the result is that the paper is much harder to
read and to understand the simple idea behind the proof. For
a direct proof following modern intuition which doesn’t use
any machine simulation see [54].

The van Emde Boas tree, when explained today, is always
presented in the recursive cluster galaxy decomposition
approach. This is not the way this structure was published
in the original papers in the period 1974 to 1977, since my
original papers from this period [45], [46], [47], [48] all follow
the binary search on levels approach. The simpler recursive
decomposition approach is worked out in details for the first
time in the unpublished classroom note from Knuth [23]. This
is sufficient reason for arranging to make this note finally
accessible and bibliographically traceable [24].

As I have indicated the two approaches are not equivalent.
The recursive approach is more abstract. There is a corre-
spondence between the cluster galaxy substructures and the
canonical sub-trees, but this correspondence breaks down when
the separation between the left and right half-trees becomes
part of the algorithms. The behaviour of the operations in
the two models are different, as illustrated by the fact that,
when the focal point happens to be located at the root of a
higher ranked canonical sub-tree, in the tree-based approach
recursion can terminate, while in the recursive decomposition
approach the recursion will proceed to the bottom level.
Another indication is the suggested impossibility to obtain
Johnson’s improvement [20] in the recursive decomposition
approach.

My choice was primarily motivated by the prohibition of
using multiplicative instructions in the RAM model; a clear
instance of scientific social control by the environment. On
the other hand it is evident from my writings that I was
fully committed to the tree-based approach, possibly for its
closeness to the silly mark-bit approach which allowed me
to break the $O(\log(n))$ barrier originally. As explained there
is just a hint at the recursive decomposition approach in my
original papers.

In the superficial survey on work during the past 33 years
I have used the themes applications, improvements and lower
bounds. Applications frequently obtain the label of a mere the-
oretical result; by the time the $\log\log(u)/\log(n)$ improvement
is sufficiently large to absorb the constant factor the universe
size $u$ has become large and the space consumption $O(u)$ of
the improved original structure will have become prohibitive;
however the improvements based on perfect hashing give new
hope. I should also stress that these improvements (and also
the lower bounds) consider RAM models which from the
1974 perspective are unrealistic (arbitrary $AC_0$ instructions
and word-size $w >> \log(u)$).

There is a possible version of the structure which I have
not discovered in any of the papers I have looked at. People like Gonnet [16] explicitly give the freedom to use special structures when the universe size has become sufficiently small. But the algorithmic decision to pass from storing the information at the root to starting the allocation and use of substructures always seems to occur when a second element is going to be inserted in the structure under consideration. An dually, the substructures are invalidated when this second element is removed.

As far as the recurrence relation \( T(u) = T(\sqrt{u}) + O(1) \) is concerned this transition might as well occur between 2 and 3, or between \( k \) and \( k+1 \) for any fixed constant \( k \). In this variation of the original idea the constant time used for setting up the substructures when the boundary is passed will increase, but one may hope that there will be far less need for going deeper into the recursion. Presumably this will not work in the worst case since an adversary may submit a chain of instructions crossing this boundary almost always, but it may yield improvements in average or amortized complexity.

ACKNOWLEDGMENT

The topic of this paper was suggested to me by Ilir Capuni suggesting that I should look at the work of the late Mihai Pătraşcu [44]. As indicated this work is way beyond my level of understanding, but it is good to read that my ideas from 39 years ago still are optimal in some parts of the relevant parameter space.

Instead this invitation has triggered me to go back into the past and adjust my own perspective on what really has happened. It has convinced me that the importance of the contribution by Don Knuth [23] is severely underestimated and it has triggered me to honour the promise I made 7 years ago to make this document accessible [24].

REFERENCES


P. van Emde Boas, Egghead betegelt badkamer, valedictory address given march 25 2010, ed. Faculteit der Natuurwetenschappen, Wiskunde en Informatica, University of Amsterdam, in Dutch.


