Fusion trees can be implemented with AC^0 instructions only (note)

Arne Andersson
Lund University
arne@dna.lth.se

Peter Bro Miltersen*
BRICS†, University of Aarhus
bromille@daimi.aau.dk

Mikkel Thorup
University of Copenhagen
mthorup@diku.dk

Abstract

Addressing a problem of Fredman and Willard, we implement fusion trees in deterministic linear space using AC^0 instructions only. More precisely, we show that a subset of $\{0, \ldots, 2^w - 1\}$ of size n can be maintained using linear space under insertion, deletion, predecessor, and successor queries, with $O(\log n / \log\log n)$ amortized time per operation on a RAM with word size w, where the only computational instructions allowed on the RAM are functions in AC^0. The AC^0 instructions used are not all available on today's computers.

1 Introduction

Fredman and Willard [FW93], based on earlier ideas of Ajtai, Fredman, and Komlos [AFK84], introduced the *fusion tree*. A fusion tree is a data structure maintaining a subset S of $U = \{0, 1, \ldots, 2^{w-1}\}$ under insertions, deletions, predecessor and successor queries (i.e. for any element x of U, we can find the predecessor and successor of x in S). The model of computation for the fusion tree is a random access machine whose registers contain w-bit words (i.e. members of U), and with an instruction set which includes unit-cost addition, subtraction, multiplication, comparison, and bit-wise Boolean `AND`. The fusion tree maintains a set of size n using $O(n)$ space and amortized time $O(\log n / \log\log n)$ per operation. An immediate corollary to the existence of the fusion tree is that n w-bit keys can be sorted in time $O(n \log n / \log\log n)$ and space $O(n)$ on a RAM with word size w.

*Supported by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

†Basic Research in Computer Science, Centre of the Danish National Research Foundation

Fredman and Willard point out that multiplication is not an AC^0 instruction, that is, there are no circuits for multiplication of constant depth and of size (number of gates) polynomial in the word length. Here, the gates are negations, and $\wedge$- and $\vee$-gates with unbounded fan-in. They pose as an open question if the fusion tree can be implemented using AC^0 instructions only. One motivation for this question is that AC^0 is the class of functions which can be computed in constant time in a certain, reasonable, model of hardware. In this model, it is therefore the class of functions for which we can reasonably assume unit cost evaluation.

In this paper, we solve this problem by showing that, given a small set of non-standard AC^0 instructions in addition to the more standard ones available in programming languages such as C (e.g. addition, comparison, bitwise Boolean operations, and shifts), the fusion tree can be implemented, with the same asymptotic space and time bounds as in [FW93].

Our presentation can also be seen as an alternative explanation of the basic mechanisms in fusion trees. We believe that our use of special-purpose instructions in place of the ingenious use of multiplication in [FW93] may make our presentation easier to understand for the casual reader. Thus, we argue that not only the use of non-AC^0 operations, but also much of the technical complexity of fusion trees, are artifacts of the particular choice of instructions in current hard-ware.

It should be noted that the transformation to AC^0 operations presented in this paper, can be applied similarly to the atomic heaps and q-heaps in Fredman and Willard's later paper [FW94].

2 Model of computation and notation

We use a RAM with word size w and we consider n w-bit keys that can be treated as binary strings or (unsigned) integers. Note that since n is the size of a subset of $\{0, \ldots, 2^w - 1\}$, we have that $w \geq \log n$. We shall also assume that $\sqrt{w}$ is a power of two.

A w-bit word will sometimes be viewed as a concatenation of $\sqrt{w}$ *fields*. Each field is of length $\sqrt{w} - 1$; to the left of each field is the *test bit* of the field. By a *bit pointer* we mean a $(\log w)$-bit key; such a key can be used to specify a bit-position within a word. Without loss of generality, we assume that $\log w < \sqrt{w} - 1$ and hence a bit pointer fits in a field. As an example, if $w = 64$ a word contains 8 fields of length 7, one test bit is stored with each field.

We will use upper-case characters to denote words and lower-case characters to denote fields. For any bit-string α, we use $|\alpha|$ to denote its length, and for $i = 1, \ldots, |\alpha|$, $\alpha[i]$ is the ith bit in α. In particular, $\alpha[1]$ is the leftmost, and $\alpha[|\alpha|]$ is the rightmost bit of α. Also, for $1 \leq i \leq j \leq |\alpha|$, $\alpha[i..j] = \alpha[i] \cdots \alpha[j]$. Finally, $\texttt{int}(\alpha)$ is number represented by α, i.e. $\texttt{int}(\alpha) = \sum_{i=0}^{|\alpha|-1} 2^i \alpha[|\alpha| - i]$. Note that our indexing of words is slightly non-standard; it is more common to index words from right to left, starting with 0. However, in the main technical part of this paper it is most convenient to think of words as strings, and these are usually indexed from left to right starting with index 1.

Apart from the standard AC^0 instructions (comparison, addition, bitwise Boolean operations and shift), we use the following ones:

`LeftmostOne`(X): returns a bit pointer to the leftmost 1 in X. A simple depth 2 circuit of quadratic size is indicated by:

$$\forall i \leq w : \texttt{int}(\texttt{LeftmostOne}(X)) = i \iff (\bigwedge_{j=1}^{i-1} \neg x[j]) \wedge x[i].$$

In fact, $\texttt{LeftmostOne}(X)$ can be implemented using a constant number of instructions available on present day chips and in programming languages such as C: we just need to convert X to floating point representation and afterwards, return the exponent.

$\texttt{Duplicate}(x, d)$: Returns a word containing copies of the field x in the d rightmost fields.

$\texttt{Select}(X, K)$: The first $\sqrt{w} - 1$ fields in K are viewed as bit pointers; a field is returned, containing the selected bits in X. Not all fields of K need to be used. The test bit of a used field is 1. A depth 3 circuit of size $O(w^{3/2})$ is indicated by:

$$\forall i \leq \texttt{b} : \texttt{Select}(X, K)[i+1] =$$

$$K[(i-1)(\texttt{b}+1)+1] \wedge \bigvee_{j=1}^{w} (j = \texttt{int}(K[(i-1)(\texttt{b}+1)+2..i(\texttt{b}+1)]) \wedge X[j])$$

Furthermore, we assume that the constants $\texttt{b} = \sqrt{w} - 1$ and $\texttt{k} = \log \sqrt{w}$ are known, $\texttt{b}$ is the length of a field.

3 The AC^0 fusion tree

Lemma 1 *Let d be an integer, smaller than, or equal to $\sqrt{w}$. Let Y be a word where the d rightmost fields contain one $\texttt{b}$-bit key each. Furthermore, assume that the d keys are sorted right-to-left, the d rightmost test bits are 0, and that all bits to the left of the d used fields (and their test bits) are 1. Then, given a $\texttt{b}$-bit key x, we can compute the rank of x among the keys in Y in constant time.*

Proof: The crucial observation is due to Paul and Simon [PS80]; they observed that one subtraction can be used to perform comparisons in parallel.

Let M be a word where the d rightmost test bits are 1 and all other bits are 0. In order to compute the rank of x among the keys in Y, we place d copies of x in the d rightmost fields of a word X. We let the test bits of those fields be 1. By the assignment $R \leftarrow (X - Y)$ $\texttt{AND}$ M the ith test bit from the right in R will be 1 if and only if $x \geq y_i$. All other test bits (as well as all other bits) in R will be 0. Hence, from the position of the leftmost 1 in R we can compute the rank of x.

We implement this in the function $\texttt{PackedRank}$ below. First, we compute d, the number of keys contained in Y. This is the same number as the number of set test bits in $\texttt{NOT}$ Y, which can be determined using the function $\texttt{LeftField}$ below. Next, we create the word X and the mask M. Finally, we make the subtraction and extract the rank.

For clarity, we introduce two simple AC^0 functions.

`FillTestBits`(d): returns a word where the d rightmost test bits are set and all other bits are zeroes. Can be implemented with shift, bitwise logical operations, and `Duplicate`.

`LeftField`(Y): If the leftmost 1 in Y is a test bit, and if all test bits to the right of this test bit are set, the number of set test bits is returned. This can be computed as $\mathtt{b}+1-(\mathtt{LeftmostOne}(Y)-1)/(\mathtt{b}+1)$. We don't have a division operation, but since $\mathtt{b}+1=2^{\mathtt{k}}$, we can implement the division by a right shift by k.

Algorithm A: `PackedRank` (Y, x)

A.1. $d \leftarrow \mathtt{LeftField}(\mathtt{NOT}\ Y)$.

A.2. $M \leftarrow \mathtt{FillTestBits}(d)$.

A.3. $X \leftarrow \mathtt{Duplicate}(x, d)\ \mathtt{OR}\ M$.

A.4. $R \leftarrow (X - Y)\ \mathtt{AND}\ M$.

A.5. `return` $\mathtt{LeftField}(R)$.

`PackedRank` clearly runs in constant time. ■

Lemma 2 *Given a word Y and a key x as in Lemma 1, with d strictly less than $\sqrt{w}$ we can generate a new word where x is properly inserted among the keys in Y in constant time.*

Proof: We need a function $\mathtt{InsertField}(Y, x, i)$ which inserts x as the $(i+1)$'st field from the right in Y, pushing fields to the right of the $(i+1)$'st field one field to the right, and sets x's test bit to 0. This can easily be implemented with shift and bitwise Boolean operations.

The function `InsertKey` below implements the lemma.

Algorithm B: `InsertKey` (Y, x)

B.1. `return` $\mathtt{InsertField}(Y, x, \mathtt{PackedRank}(Y, x))$.

■

4 Fusion tree nodes

In this section we give our version of the main building block of fusion trees.

Proposition 3 *Given d sorted w-bit keys, $d \leq \sqrt{w}$, we can construct, in $O(d)$ time, a static data structure using $O(d)$ space, so that predecessor and successor queries can be supported in $O(1)$ worst case time.*

We use such a data structure to represent each node of the fusion tree. Our construction emulates the construction used by Fredman and Willard very closely. However, since we want our presentation to be self-contained, we devote this section to give all the details of the construction.

The main idea is to make use of significant bit positions. View the set of w-bit sorted keys $Y_1, \ldots, Y_d$ as stored in a binary trie. Each key is represented as a path down the trie, a left edge denotes a 0 and a right edge denotes a 1. We get the significant bit positions by selecting the levels in the trie where there is at least one binary (that is, non-unary) node. These bit positions can be computed by taking the position of the first differing bit between all pairs of adjacent keys in $Y_1, \ldots, Y_d$. By extracting the significant bits from each key we create a set of compressed keys $y_1, \ldots, y_d$. Since the trie has exactly d leaves, it contains exactly $d-1$ binary nodes. Therefore, the number of significant bit positions, and the length of a compressed key, is at most $d-1$. Since $d - 1 \leq \sqrt{w} - 1 = \mathtt{b}$, we can pack these compressed keys in linear time by repeated calls to `InsertKey` in Lemma 2.

We need the following AC^0 functions, which can be implemented in a straightforward way:

$\mathtt{DiffPtr}(X, Y)$: returns a bit pointer to the leftmost differing bit between X and Y. (Can be implemented as $\mathtt{LeftmostOne}(X \;\mathtt{XOR}\; Y)$.)

$\mathtt{Fill}(X, p)$: X is a word and p is a bit pointer. Returns a copy of X where all bits to the right of position p have the same value as the bit in position p. If $p = w + 1$, $\mathtt{Fill}(X, p) = X$. (Can be implemented by shifting, addition, and bitwise Boolean operations.)

The procedure `Construct` takes as input a set of sorted keys $Y_1, \ldots, Y_d$ and computes the set of significant bit positions; pointers to these positions are concatenated in sorted order in the word K. Next, a set of compressed keys is created by selecting the bits specified in K from $Y_1, \ldots, Y_d$. These compressed keys are packed in the word Y.

Algorithm C: $\mathtt{Construct}\ (Y_1, \ldots, Y_d)$

C.1. $K \leftarrow 0$.

C.2. For $i \leftarrow 1$ to $d-1$, $K \leftarrow \mathtt{InsertKey}(K, \mathtt{DiffPtr}(Y_i, Y_{i+1}))$.

C.3. $Y \leftarrow 0$.

C.4. For $i \leftarrow 1$ to d, $Y \leftarrow \mathtt{InsertKey}(Y, \mathtt{Select}(Y_i, K))$.

When implemented as above, the same bit-pointer may be packed several times in K. This makes no difference.

We can now compute the rank of a query key X in constant time. Let $x = \mathtt{Select}(X, K)$ and let p_X denote the longest common prefix of X and any key in $Y_1, \ldots, Y_d$. Let y_i denote the compressed version of Y_i. Let $q_1 < q_2 < \cdots < q_{d-1}$ be the significant bit positions and let $q_d = w + 1$. Then $x[j] = X[q_j]$ and $y_i[j] = Y_i[q_j]$. Note that if $y_{i_1}[1..j] = y_{i_2}[1..j]$, then $Y_{i_1}[1..q_{j+1} - 1] = Y_{i_2}[1..q_{j+1} - 1]$.

Lemma 4 *If x has rank i in $y_1, \ldots, y_d$, then either Y_i or Y_{i+1} start by p_X.*

Proof: Let i' be such that $Y_{i'}$ has the prefix p_X. Let j be the maximal index such that $q_j \leq |p_X|$. Then $y_{i'}[1..j] = x[1..j]$. However, since i is the rank of x, we either have $x = y_i$ or $y_i < x < y_{i+1}$. This means that for i'' equal to either i or $i+1$, we have $y_{i''}[1..j] = y_{i'}[1..j]$. Hence $Y_{i''}[1..q_{j+1} - 1] = Y_{i'}[1..q_{j+1} - 1]$, but $q_{j+1} - 1 \geq |p_X|$, so $Y_{i''}$ must share the prefix p_X with $Y_{i'}$. ∎

Lemma 5 *Consider a key Z that starts by p_X and where all remaining bits in Z have the same value as X's first distinguishing bit, i.e. the bit of X following p_X. Set $z =$ $\texttt{Select}(Z, K)$. Let z^0 be the result of setting the last bit of z to 0, and let z^1 be the result of setting it to 1. Then $z \in \{z^0, z^1\}$, and then the rank of X among $Y_1, \ldots, Y_d$ is either that of z^0 or that of z^1 among $y_1, \ldots, y_d$.*

Proof: Clearly Z has the same rank as X among $Y_1, \ldots, Y_d$. Let i be the rank of z among $y_1, \ldots, y_d$. If $y_i \neq z$, then i is the correct rank of X. The same holds if $Y_i = Z = X$. However, suppose that $y_i = z$ and $Y_i \neq Z$. Suppose that the first distinguishing bit $X[|p_X| + 1]$ of X is 0. Then there is some $j > |p_X|$, $j \notin \{q_1, \ldots, q_{d-1}\}$, such that $Y_i[j] = 1$. Hence $Y_i > Z$. Since $y_{i'} = y_i$ implies $Y_{i'} = Y_i$, we conclude that the rank of Z among $Y_1, \ldots, Y_d$ is the rank of z^1 among $y_1, \ldots, y_d$.

If $X[|p_X| + 1] = 1$, symmetrically we have that the rank of Z among $Y_1, \ldots, Y_d$ is the rank of z^0 among $y_1, \ldots, y_d$. ∎

We can use Lemma 4 to compute the length of p_X and hence the position of X's first distinguishing bit. Once this position is known, we can apply Lemma 5 to find the proper rank of X.

We encode this in the function `Rank` below. The variable p is used to store the position of X's first distinguishing bit. This is the only place in this section where we, for convenience, deviate slightly from Fredman and Willard's original method: Instead of filling the query keys with 1s (or 0s) and making a second packed searching, they use a lookup table of size $\Theta(d^2)$ in a node of degree d.

Algorithm D: $\texttt{Rank}(X)$

D.1. $i \leftarrow \texttt{PackedRank}(Y, \texttt{Select}(X, K))$.

D.2. If $i = 0$, $p \leftarrow \texttt{DiffPtr}(X, Y_1)$;

D.3. else $p \leftarrow \max(\texttt{DiffPtr}(X, Y_i), \texttt{DiffPtr}(X, Y_{i+1}))$

D.4. $Z \leftarrow \texttt{Fill}(X, p)$.

D.5. $z \leftarrow \texttt{Select}(Z, K)$.

D.6. $i \leftarrow \texttt{PackedRank}(Y, z)$.

D.7. If $Y_i \leq Z < Y_{i+1}$, return i;

D.8. else $z[\texttt{b}] \leftarrow \neg z[\texttt{b}]$; return $\texttt{PackedRank}(Y, z)$.

Now to find the predecessor of a value in the set, we find the rank of the value, and the predecessor can now be found by a table lookup in the sorted table of the set. This completes the proof of Proposition 3.

5 The fusion tree

Having dealt with fusion nodes, the proof of our main theorem is virtually the same as that of Fredman and Willard. Note that we can allow the B-tree nodes to have higher degree than in the original fusion tree: $\sqrt{w}$ compared to $w^{1/6}$.

Theorem 6 *A set of word sized integers can be maintained using linear space under insertion, deletion, predecessor, and successor queries, with $O(\log n/\log\log n)$ amortized time per operation on an AC^0 RAM.*

Proof: The proof proceeds as in [FW93]:

Proposition 3 allows us to implement a B-tree [BM72] node of degree $d \leq \sqrt{w}$. Searching in such a node takes constant time while splitting, merging, and adding/removing keys take $O(d)$ time. By keeping traditional, comparison-based, weight-balanced trees of size $\Theta(d)$ at the bottom of the B-tree, we can ensure that at most every $\Theta(d)$'th update causes any change in a B-tree node.

The number of B-tree levels is $O(\log n/\log d)$ and the height of a weight-balanced tree is $O(\log d)$. Since $w \geq \log n$, we can choose $d = \Theta(\sqrt{\log n})$ and the theorem follows. ∎

As a final remark, we note that applying a result of Dietz [D89], we can also support rank-operations within the $O(\log n/\log\log n)$ time bound. Dietz showed that a list of atoms can be maintained under the operations *insert*, which inserts a new atom after a specified position in the list, *delete*, which deletes an atom from the list, and *position*, which, given an atom, returns its position in the list, in time $O(\log n/\log\log n)$ per operation. Dietz' list maintenance structure uses only a very basic, AC^0, instruction set. Using the predecessor query of a fusion tree maintaining a set, we can make Dietz' structure maintain a sorted list of the elements of the set. Then, to find the rank in the set of a new value, we just need to find the position in the list of its predecessor in the set. Hence, rank-queries are supported in time $O(\log n/\log\log n)$ in addition to the other operations. This matches a lower bound by Fredman and Saks [FS89] which holds without any assumptions on the instruction set.

References

[AFK84] M. Ajtai, M.L. Fredman, and J. Komlos, Hash functions for priority queues, *Information and Computation* 63:217–225, 1984.

[BM72] R. Bayer and E.M. McCreight, Organization and maintenance of large ordered indexes, *Acta Informatica* 1(3):173–189, 1972.

[D89] P.F. Dietz. Optimal algorithms for list indexing and subset rank, in: *Proceedings of 1st Workshop on Algorithms and Data Structures (WADS)*, pages 39–46, *Lecture Notes in Computer Science*, volume 382, Springer, 1989.

[FS89] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures, in: *Proceedings of 21th ACM Symposium on Theory of Computing (STOC)*, pages 345–254, 1989.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[FW94] M.L. Fredman and D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *Journal of Computer and System Sciences*, 48:533–551, 1994.

[PS80] W. J. Paul and J. Simon. Decision trees and random access machines. In *Proc. International Symp. on Logic and Algorithmic*, Zürich, pages 331–340, 1980.