Fortgeschrittene Datenstrukturen — Vorlesung 5

Stephan Erb

17.11.2011

1 Fusion Trees

A *fusion tree* [FW93] is an integer data structure for fast predecessor and successor queries. It requires $\mathcal{O}(n)$ space and supports queries within $\mathcal{O}(\log_w n)$, where n is a number of elements from a universe $U = [0, u-1] = [0, 2^w - 1]$.

The following discussion is restricted to static fusion trees which support neither insert nor delete. It shall furthermore be noted that fusion trees have a theoretical interest only, as the involved constant factors preclude practicality [FW93].

1.1 Top-Level Idea

Essentially, a fusion tree is a B-tree with branching factor $b = w^{\frac{1}{5}}$ and height $\Theta(\log_w n)$. It is used to realize predecessor and successor queries via top down traversals, by finding the virtual insertion position for the queried element q. This is analogous to the implementation on top of balanced binary trees.

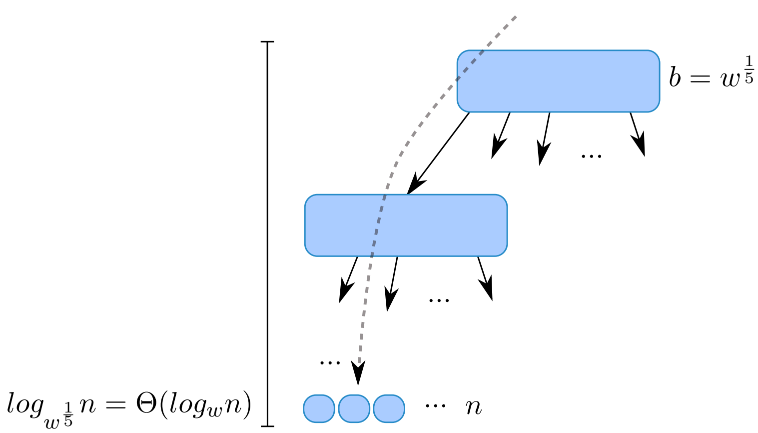


Figure 1: Structure of a fusion tree

Given the height of such a tree and the desired runtime of $\mathcal{O}(\log_w n)$, a traversal may only spend $\mathcal{O}(1)$ time at each node. This poses the following challenges:

- The $b = w^{\frac{1}{5}}$ keys of a node, each of size w, have to be compressed. They have to fit into a machine word of $\Theta(w)$ bits, as otherwise, those cannot even be read in $\mathcal{O}(1)$.

  We will achieve this using so-called *sketches* (see section 1.2).

- The $sketch(q)$ of a queried element q has to be compared to the compressed keys of a node, in order to determine where to proceed the top-down traversal. This comparison has to happen in $\mathcal{O}(1)$ time.

  We will perform this comparison simultaneously for all keys using bit-parallel computations (see section 1.5).

- The $sketch(q)$ has to be computable in $\mathcal{O}(1)$ time.

  We will achieve this using a clever definition of *sketch* and multiplications (see section 1.4).

1.2 Fusion Tree Nodes

The keys of a node $x_0 < x_1 < ... < x_{b-1}$ can be embedded into a tree that is labeled according to their binary representation:

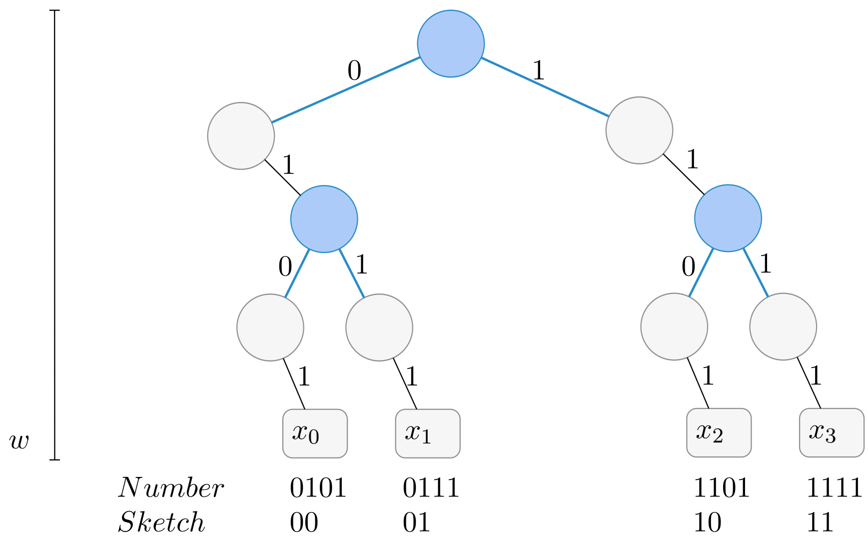


Figure 2: Keys in a trie. Branching nodes are marked in blue.

Nodes with more than one child are branching nodes. Being part of a trie, these nodes represent the so-called *important bits* of our keys. Only these bits are required to differentiate all keys of a node. Their indices are $b_0 < b_1 < ... < b_{r-1}$. Given that at most $b-1$ branching nodes are required to differentiate b keys within a trie, the number of important bits is bounded by $r < b = w^{\frac{1}{5}}$.

The notion of important bits allows us to define a compression function for keys:

Definition 1. *A sketch restricts an element to the important bits at the positions b_0 to b_{r-1}:*

$$sketch(x_i) = sketch(\sum_{j=0}^{w-1} 2^j x_{ij}) = \sum_{j=0}^{r-1} 2^j x_{ib_j}$$

- *sketch* preserves the order of keys: iff $x_0 < ... < x_{b-1}$ then $sketch(x_0) < ... < sketch(x_{b-1})$.
- b *sketches* can be *fused*[1] into one machine word of size $\Theta(w)$: $b \cdot r \leq b^2 = w^{\frac{2}{5}}$.

Sketches are stored alongside of the original keys. An example can be found in figure 2.

[1] thus the name of this data structure

1.3 Querying Nodes

Given that sketches are solely based on the important bits of the keys, an additional important bit may be required to distinguish an arbitrary element q. For example, given the trie illustrated in figure 2, $sketch(1010) = 11$ would lead to the wrong conclusion that q fits in between x_2 and x_3. Thus:

$$sketch(x_i) < sketch(q) \le sketch(x_{i+1}) \nRightarrow x_i < q \le x_{i+1}$$

Fortunately, $sketch(q)$ is still sufficient to overcome this problem and to compute the correct predecessor and successor keys for q:

Let x_i and x_{i+1} be the neighbors of q according to *sketch*. The path to q within the trie will at least deviate from one of the paths to these elements. Such a point of deviation corresponds to the *longest common prefix (LCP)* of the paths in question. Thus, $y = max\{LCP(q, x_i), LCP(q, x_{i+1})\}$ yields the deepest node[2] where the path to q deviates from the paths to x_i and x_{i+1}.

Importantly, y can *never* be a branching node. Otherwise, the corresponding bit would have been included in the sketch and prevented $sketch(q)$ from falling in between the sketches of the particular x_i and x_{i+1} in the first place.

There are two problematic cases depending on q being in the right or left subtree of y:

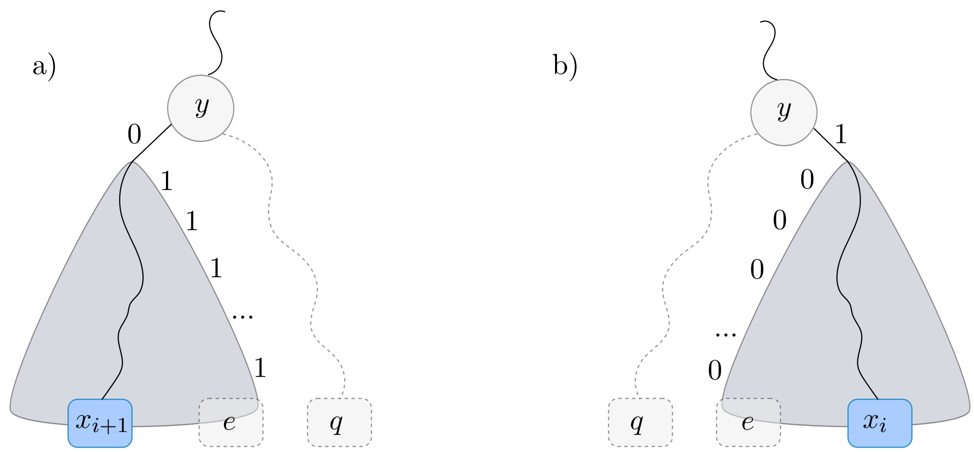


Figure 3: Problematic cases of how q might relate to its neighbors.

We use an element e to resolve those cases. There can be no other key between e and q, as otherwise y would have to be a branching node.

Predecessor case: $e = y011...1$ is the right-most element in the left subtree of y. It is the predecessor of q if it exists; otherwise, it has the same predecessor.

Successor case: $e = y100...0$ is the left-most element in the right subtree of y. It is the successor of q if it exists; otherwise, it has the same successor.

A query for $sketch(e)$ yields the true neighbor keys of q. No recursion or further special case handling is required, as e is defined relative to y and therefore does not suffer from the same problem

[2] it can also be computed via the most significant bit: $max\{msb(q \text{ XOR } x_i), msb(q \text{ XOR } x_{i+1})\}$

as q. For example in the successor case, $sketch(e) = sketch(y10...0)$ equals the *smallest* sketch in the right subtree of y. Due to the construction of y, this can only be the sketch of the successor key we were looking for. The case is illustrated in figure 4.

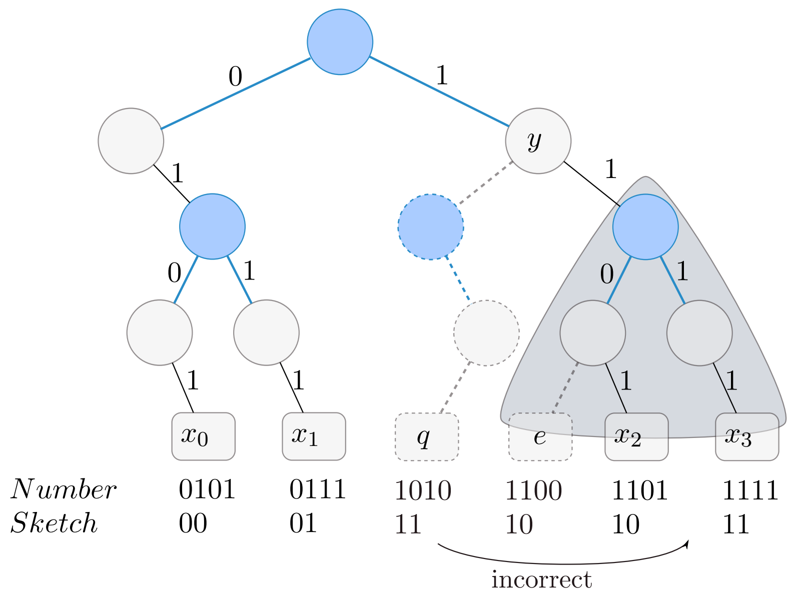


Figure 4: An example of the query problem.

Re-using the idea from *y-fast trees*, we can use a double linked list to navigate between predecessor and successor keys. Knowing both these keys, we can proceed with the top-down traversal.

1.4 Computation of Sketches

We know the important bit positions and want to calculate the *sketch* of a given q. We have to perform this in $\mathcal{O}(1)$ time. It is therefore not possible to just iterate over these positions to extract the corresponding bits.

Actually, computing $sketch(q)$ in O(1) time is difficult. We will therefore resort to a relaxed definition of *sketch* and a clever use of multiplication with a precomputed mask m:

Definition 2. *An appSketch restricts an element to the important bits at the positions b_0 to b_{r-1}. Those bits may be separated by 0's. The number of 0's is given by m.*

$$appSketch(q) = appSketch(\sum_{i=0}^{w-1} 2^i q_i) = \sum_{i=0}^{r-1} 2^{b_i+m_i} q_{b_i} \gg m_0 + b_0$$

Computation of appSketch(q):

1. Mask out all bits at non-important bit positions:
   $q' = q \text{ AND } \sum_{i=0}^{r-1} 2^{b_i} = \sum_{i=0}^{r-1} 2^{b_i} q_{b_i}$
2. Redistribute the important bits using a *multiplication*:
   $appSketch(q)' = q' \cdot m = q' \cdot \sum_{j=0}^{r-1} 2^{m_j} = \sum_{i=0}^{r-1} \sum_{j=0}^{r-1} 2^{b_i+m_j} q_{b_i}$

3. Drop multiplication results we are not interested in:
   $appSketch(q)'' = appSketch(q)'$ AND $\sum_{i=0}^{r-1} 2^{b_i+m_i} = \sum_{i=0}^{r-1} 2^{b_i+m_i} q_i$
4. Right shift by $m_0 + b_0$ to remove unnecessary trailing zeros.

With an appropriate m we will still be able to fuse all $w^{\frac{1}{5}}$ keys into a single machine word $\Theta(w)$. We will prove that it is always possible to find such an m. We have to pre-compute it, and it has to satisfy the following criteria:

1. No collisions: $b_i + m_j = b_k + m_l$ iff $i = k$ and $j = l$.
2. Bit order preserved: $b_0 + m_0 < b_1 + m_1 < ... < b_{r-1} + m_{r-1}$.
3. Sketch compact enough: $\underbrace{(b_{r-1} + m_{r-1}) - (b_0 + m_0)}_{\text{distance between first and last bit}} \leq r^4 = w^{\frac{4}{5}}$.

A sketch consists of r terms $b_i + m_i$. To achieve a maximal spread of r^4, each of the terms may deviate at most r^3 from the preceding term $b_{i-1} + m_{i-1}$.

We approach constraints 1 and 3 as follows:

$$b_i + m_j \not\equiv b_k + m_l \mod r^3 \qquad \forall i \neq k \wedge j \neq l$$

When there is only one term (one b_i and one m_j), the condition is trivially satisfied. Now assume by induction that we have already found $m'_0 < m'_1 < ... < m'_{t-1}$ for a $t < r$. To find m'_t, we have to avoid the existing terms realized by the other m_i's:

$$m'_t \neq m'_l + b_k - b_i \qquad \forall_{l,k.i} \text{ with } 0 \leq l < t \text{ and } 0 \leq i, k < r$$

Hence, we must avoid $t \cdot r \cdot r \leq (r-1)r^2$ choices. These are less than our r^3 available choices for the placement of an important bit within a sketch. We can therefore always find a suitable m.

We can calculate a m'_i in order, starting with 0. An example of this approach is illustrated in table 1.

	$b_0 = 0$	$b_1 = 61$	$b_2 = 63$
$m'_0 = 0$	0	7	9
$m'_1 = 1$	1	8	10
$m'_2 = 4$	4	11	13

Table 1: The table contains the values $b_i + m'_j \pmod{27}$ that we have to avoid for $r = 3$ and the given important bits b_i. The table is filled row-wise. Values for a m'_i are tested in consecutive order. In the given example $m'_2 = 2$ and $m'_2 = 3$ had to be skipped because they lead to collisions.

Due to the modulo operation such an m' might not satisfy the order-preserving property of the important bits. This can be corrected by scaling the placement of an important bit with its index:

$$\begin{aligned} m_i &= m'_i + (w - b_i + ir^3 \text{ rounded down to be a multiple of } r^3) \\ &= m'_i + (\lfloor (w - b_i + ir^3)/r^3 \rfloor \cdot r^3) \end{aligned}$$

A sketch then requires $r \cdot r^3$ space and important bits fall into consecutive, non-overlapping blocks:

$$w + r^3(i-1) \leq m_i + b_i < w + r^3 i$$

Let us reconsider the example from table 1 for $w = 64$ and calculate the final m:

$$\begin{aligned}
m_0 &= m'_0 + (\lfloor (w - b_0 + 0r^3)/r^3 \rfloor \cdot r^3) \\
&= 0 + (\lfloor (64 - 0)/r^3 \rfloor \cdot r^3) = 54 \\
m_1 &= m'_1 + (\lfloor (w - b_1 + 1r^3)/r^3 \rfloor \cdot r^3) \\
&= 1 + (\lfloor (64 - 61 + 27)/r^3 \rfloor \cdot r^3) = 28 \\
m_2 &= m'_2 + (\lfloor (w - b_2 + 2r^3)/r^3 \rfloor \cdot r^3) \\
&= 4 + (\lfloor (64 - 63 + 54)/r^3 \rfloor \cdot r^3) = 58
\end{aligned}$$

We observe that we now satisfy all constraints, including the second one:

$$b_0 + m_0 < b_1 + m_1 < b_2 + m_2 \Leftrightarrow 54 + 0 < 28 + 61 < 58 + 63$$

1.5 Comparison of Sketches

The idea outlined in section 1.3 requires us to find x_i and x_{i+1} for a given q so that $appSketch(x_i) < appSketch(q) \leq appSketch(x_{i+1})$. To compare an $appSketch(x_i)$ with $appSketch(q)$, we subtract them and inspect the carry/sign bit. Counting all comparisons won by $appSketch(q)$ will give us its rank within the ordered sketches of a node. To perform all these subtractions in constant time, we fuse all sketches into a single machine word and issue a single, bit-parallel subtraction.

$$appSketch(q) \times \underbrace{\overbrace{00...01}^{r^4+1\ bits}\ \overbrace{00...01}^{r^4+1\ bits}\ ...\ \overbrace{00...01}^{r^4+1\ bits}}_{b\ terms} = \underbrace{0appSketch(q)0appSketch(q)...0appSketch(q)}_{0appSketch(q)\ repeated\ b\ times}$$

$$\Big(1appSketch(x_0)...1appSketch(x_{b-1})\Big) - \Big(0appSketch(q)...0appSketch(q)\Big) = \underbrace{\overbrace{c_0......}^{r^4+1\ bits}\ ...\ \overbrace{c_{b-1}......}^{r^4+1\ bits}}_{b\ terms}$$

We can AND this expression with a mask to zero all non-interesting bits, so that only the first bit of each term remains.

$$\left(\underbrace{\overbrace{c_0......}^{r^4+1\ bits}\ ...\ \overbrace{c_{b-1}......}^{r^4+1\ bits}}_{b\ terms} \right) AND \left(\sum_{i=0}^{b-1} 2^{i(r^4+1)+r^4} \right) = \underbrace{\overbrace{c_0 0...0}^{r^4+1\ bits}\ ...\ \overbrace{c_{b-1}0...0}^{r^4+1\ bits}}_{b\ terms}$$

For these bits we know that

$$c_i = \begin{cases} 0 & \text{if } appSketch(x_i) < appSketch(q) \\ 1 & \text{if } appSketch(x_i) \geq appSketch(q) \end{cases}$$

Sketches preserve order. Instead of counting all $c_i = 0$, we can resort to finding the most significant bit in the result term. It represents the index of the first $sketch(x_i)$ that is larger than $sketch(q)$.

2 Finding the Most Significant Bit

The most significant bit (msb) in a binary number corresponds to the left-most 1. There are different ways to compute it, for example:

1. In $\mathcal{O}(1)$ time with a special operation supported by most modern processors (e.g., BSR (bit scan reverse) on x86).
2. In $\mathcal{O}(\log w)$ time using a binary search over the word. Depending on the existence of 1's in the first half, the search is continued in the first or second half. See listing 1.

Algorithm 1: msb(x)

---

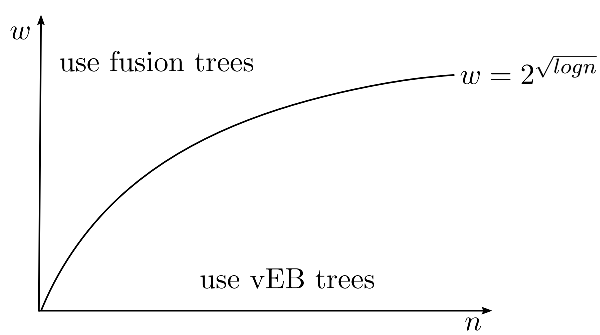
begin
    $\lambda \leftarrow 0$
    for $k = \log(w) - 1$ *downto* 0 **do**
        $z \leftarrow x \gg 2^k$ `// Discard right half`
        if $z \neq 0$ **then**
            $\lambda \leftarrow \lambda + 2^k$
            $x \leftarrow z$ `// Continue search in left half`
        end
    end
    return λ
end

---

3 Predecessor Data Structures Revisited

Van Emde Boas trees and y-fast tries perform well and improve on binary search trees when the number of elements is sufficiently large with regard to the size of the universe (i.e., $\log n \gg \log w$).

Fusion trees are designed for the case when the universe is large with regard to the number of elements. In particular, they can find predecessor / successors in $\mathcal{O}(\log_w n) = \mathcal{O}(\frac{\log n}{\log w})$ time.



Depending on the actual value of n and w, we can choose the right structure (vEB or fusion tree) and achieve $\mathcal{O}(\min\{\log w, \log_w n\})$ time. Because the two terms are equal when $w = 2^{\sqrt{\log n}}$, the minimum will never be greater than $\mathcal{O}(\sqrt{\log n})$.

References

[Dem03] E. Demaine. Fusion trees. *Lecture Notes on Advanced Data Structures, Lecture 4, MIT*, Spring 2003. Available online at http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L4/lecture4.pdf.

[Dem10] E. Demaine. Fusion trees. *Lecture Notes on Advanced Data Structures, Lecture 10, MIT*, Spring 2010. Available online at http://courses.csail.mit.edu/6.851/spring10/scribe/lec10.pdf.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.