6.851: Advanced Data Structures Spring 2010

Lecture 10 — March 8, 2010

Prof. Erik Demaine *Scribe: Nicholas Zehender*

1 Overview

In the last lecture we covered van Emde Boas and y-fast trees, integer data structures which support insert, delete, predecessor, and successor operations in $\log(w)$ time. These data structures are based on the Word RAM model, where we can manipulate a constant number of w-bit words in constant time.

In this lecture we covered fusion trees, as described by Fredman and Willard [1]. A fusion tree is a static data structure storing n w-bit integers which supports predecessor and successor queries in $O(\log_w n)$ time per query and $O(n)$ space. There is also a dynamic version of fusion trees which get $O(log_w(n) + log(log(n)))$ for updates, as described by Andersson and Thorup [2], but we did not cover this. Depending on whether n is small or large compared to w, we can use fusion trees or van Emde Boas trees, and our query time will be $O(\min\left(\frac{\log n}{\log w}, \log w\right))$. This minimum will never be greater than $O(\sqrt{\log n})$, giving us a better than logarithmic bound which only depends on n.

2 Fusion Trees

A fusion tree is a B-tree with a branching factor of $w^{1/5}$. The depth of such a tree is $\log_{w^{1/5}}(n) = \log(n)/\log(w^{1/5}) = 5\log(n)/\log(w)$.

In order to actually achieve $O(\log(n)/\log(w))$ time for queries, we must deal with each node we encounter in constant time. However, there are $w^{1/5}$ keys, each w bits long, in each node: in constant time, we can only do a constant number of operations on a constant number of w-bit words. To solve this problem, we will only compare the "interesting bits" of the keys.

2.1 Interesting bits

Consider what a trie containing the keys would look like. This trie would have height w, the length of the keys, and would have $k = O(w^{1/5})$ leaves (since the keys are distinct).

"Interesting bits" are bits which correspond to levels in this trie with branching nodes—these are the bits we will look at in order to distinguish keys. Since our trie has k leaves, it has $k-1$ branching nodes, and at most $k-1$ interesting bits (since multiple branching nodes might be on the same level). We will call the indices of the interesting bits b_0 through b_{r-1}.

2.2 Sketches

To get the sketch of a word x, we delete all the bits except the interesting bits. Order is preserved: for our keys x_i, we have $x_i < x_j$ if and only if $sketch(x_i) < sketch(x_j)$.

As an example, consider the case where we have four 4-bit words; `0000`, `0010`, `1100`, and `1111`. Only the first and third bits are interesting, so the sketches of these keys are `00`, `01`, `10`, and `11`. When we query the tree, we compare sketches instead of comparing keys, so that we can compare a query's sketch with the sketches of all $w^{1/5}$ keys at a node in constant time.

2.3 Querying the fusion tree

If we query the fusion tree just based on sketches, however, we can encounter a problem. Say we query for `0101`. Its sketch is `00`, but it lies between `0010` (with sketch `01`) and `1100` (with sketch `10`). But knowing the node `0000` is actually useful when querying for `0101`. We can find the longest common prefix of the query and the node found (or, equivalently, the lowest common ancestor in the trie). If the sketch of our query did not exactly match the sketch of one of our keys, then instead we use the key x_i whose sketch is the predecessor or successor of q's sketch, whichever gives the longer common prefix. Call this common ancestor in the trie y.

We can find y by taking the binary XOR of sketch(q) and sketch(x_i), and looking at the most significant bit of the result. This bit cannot be an important bit: if it were, it would have been included in the sketch and the problem would not have occured. Andersson, Miltersen, and Thorup showed that the most significant bit can be found in constant time as an AC^0 operation [3]. It can also be found on most modern CPUs with a single instruction.

There are two symmetric cases, depending on whether our query is in the left or right subtree of y. In our example, the query is in the right subtree. In this case the predecessor of our query is the largest element in the left subtree of y. We know no keys lie in the right subtree of y: since y is not at an interesting level of the trie, no branching could have occured at y. To find the largest element of the left subtree, we query for the sketch of $e = y$`011...11` (or $e = y$`100...00` in the symmetric case). e is easy to compute with binary ANDs and ORs. Since the prefix of e is y`0`, a query for it will fall into the correct subtree. Since all the remaining bits of e are `1`'s, the result of the query will be the key x_i with the greatest sketch in the left subtree—but since the sketches of the x_i's are in the same order as the x_i's themselves, this gives the greatest x_i in the left subtree, which is exactly what we wanted. We can connect the x_i's in a linked list to make it easy to find the successor once we've found the predecessor, or vice versa.

So, the algorithm for querying a fusion tree looks like this:

(a) Compute $sketch(q)$.

(b) Search for $sketch(q)$ among the sketches of the x_i's.

(c) Compute the lowest common ancestor y of q and the key whose sketch is the predecessor or successor of q's sketch.

(d) Compute e, which is either y`01...1` or y`10...0`.

(e) Search for $sketch(e)$ among the sketches of the x_i's. This will give the real predecessor or successor of q: use the linked list to find the other.

(f) Unsketch the results.

All that remains is to describe how we compute sketches, and how we compare $sketch(q)$ to all $w^{1/5}$ of a node's key sketches in constant time.

2.4 Computing the sketch in constant time

Calculating the perfect sketch in constant time is not possible on a word RAM, so instead we find an approximate sketch. The approximate sketch has all the interesting bits in the correct order, but can also have extra `0`'s padding it.

We want to be able to store all $w^{1/5}$ key sketches at a node in a constant number of words, so the approximate sketch should take at most $O(w^{4/5})$ bits. We compute the sketch of our query q in constant time in the following way:

(a) Mask out all non-interesting bits: $x' = x$ AND $\sum_{i=0}^{r-1} 2^{b_i}$.

(b) Multiply x' by $m = \sum_{j=0}^{r-1} 2^{m_j}$.

(c) The product is $\sum_{i,j} x_i 2^{b_i+m_j}$; AND it with $\sum_{i=0}^{r-1} 2^{b_i+m_i}$ to get $\sum_{i=0}^{r-1} x_i 2^{b_i+m_i}$.

(d) Right shift by $m_0 + b_0$ to remove unnecessary trailing zeros.

Claim: For any set of b_i's there is a set of m_i's of the same size such that

(a) There are no collisions: $b_i + m_j = b_{i'} + m_{j'}$ if and only if $i = i'$ and $j = j'$.

(b) Order is preserved: $b_0 + m_0 < b_1 + m_1 < \ldots < b_{r-1} + m_{r-1}$

(c) The resulting sketch is short enough: $(b_{r-1} + m_{r-1}) - (b_0 + m_0) = O(r^4) = O(w^{4/5})$

Proof: We will first choose m'_0 through m'_{r-1}, all at most r^3, such that the $b_i + m'_j$ are distinct modulo r^3 for all i, j. This can be done by choosing the values in order. Say that values m'_0 through m'_{t-1} have been chosen already. To choose the next value m'_t, we choose any value not equal to $m'_i + b_j - b_k$ for $0 \le i < t$, $0 \le j < r$, $0 \le k < r$. This will always be possible because we have r^3 total values to choose from, of which at most tr^2 are invalid. Since $t < r$, there must be at least one valid option left.

The values m'_i might not satisfy the order-preserving property. So to get our final values, we let

$$m_i = m'_i + (ir^3) + ((w - b_i) \text{ rounded down to be a multiple of } r^3)$$

The three properties now hold:

(a) $m_i \equiv m'_i \pmod{r^3}$, so all the $b_i + m_j$ are still distinct modulo r^3, implying they are completely distinct.

(b) $m_i + b_i$ is in the r^3-sized interval after $(\lfloor w/r^3 \rfloor + i) \cdot r^3$, so order is preserved.

(c) $m_0 + b_0$ is approximately w, and $m_{r-1} + b_{r-1}$ is approximately $w + r^4$, so their difference is $O(r^4)$.

2.5 Multiple comparisons in constant time

We need to be able to find where *sketch*(q) lies among the sketches of keys at a given node in constant time. We will do so using something called the "node sketch."

Node Sketch: We store all the sketches of the x_i's at a node in a single word by prepending a `1` to each and concatenating them. The result will look like this: `1`*sketch*$(x_0)\ldots$`1`*sketch*(x_{k-1}).

In order to compare *sketch*(q) to all the key sketches with one subtraction, we take *sketch*(q) and make k copies of it in a word `0`*sketch*$(q)\ldots$`0`*sketch*(q). If the sketches were 6 bits long, we would multiply *sketch*(q) by `0000001`...`0000001`. Then we subtract this value from the node sketch. This lets us subtract `0`*sketch*(q) from each `1`*sketch*(x_i) with a single operation: since `1`*sketch*(x_i) is always bigger than `0`*sketch*(q), carrying will not cause the subtractions to interfere with each other. In fact, the first bit of each block will be `1` if and only if *sketch*$(q) \leq$ *sketch*(x_i). After subtracting, we AND the result with `1000000`...`1000000` to mask all but the first bit of each block.

The *sketch*(x_i)'s are sorted in each node, so for some index k we have *sketch*$(q) >$ *sketch*(x_i) when $i < k$ and *sketch*$(q) \leq$ *sketch*(x_i) otherwise. We need to find this index k. Equivalently, we need to find the number of bits which are equal to 1 in the above result. To do this, we can multiply by `0000001`...`0000001`: all the bits which were set to 1 will collide in the first block of the result, so we can find their sum by looking at that block.

References

[1] Michael L. Fredman, Dan E. Willard: *Surpassing the Information Theoretic Bound with Fusion Trees.* J. Comput. Syst. Sci. 47(3): 424-436 (1993)

[2] Arne Andersson, Mikkel Thorup: *Dynamic ordered sets with exponential search trees.* J. ACM (JACM) 54(3):13 (2007)

[3] Arne Andersson, Peter Bro Miltersen, Mikkel Thorup: *Fusion Trees can be Implemented with AC Instructions Only.* Theor. Comput. Sci. (TCS) 215(1-2):337-344 (1999)