6.851: Advanced Data Structures Spring 2012

Lecture 12 — April 3, 2012

Prof. Erik Demaine *Scribes: Kent Huynh (2012), Shoshana Klerman (2012), Eric Shyu (2012)*

1 Introduction

We continue our analysis of integer data structures, focusing this lecture on *fusion trees*. This structure employs some neat bit tricks and word-level parallelism. In particular, we discuss the following techniques necessary to understand the workings of a fusion tree: *sketching*, which allows certain w-bit words to be compressed to less than w bits, *parallel comparison*, where multiple words can be compared for the cost of a single word, and finally the computation of the most significant set bit of a word in constant-time.

2 Overview of Fusion Trees

We first describe the key results of fusion trees, as well as the model we will be assuming for the majority of the exposition. As with the van Emde Boas trees described in the previous lecture, we will be working under the word RAM model (transdichotomous RAM with C-style operations) and we are looking to store n w-bit integers statically. Under these assumptions, the fusion tree covered here and detailed by Fredman & Willard in their original papers ([1], [2]) performs predecessor/successor operations in $O(\log_w n)$ time, and require $O(n)$ space, cf. the van Emde Boas trees which run in $O(\log w)$ time and require $O(n)$ space. Other models and variants of interest include:

- AC^0 RAM version [3]: the model is restricted to operations with constant-depth (but unbounded fan-in and fan-out) circuits. In particular, multiplication is not permitted, since it requires a circuit of logarithmic depth in the number of bits (this model was more relevant in the past when multiplication was very costly relative to other operations like additions; this is no longer the case due to optimizations such as pipelining);
- Dynamic version via exponential trees [4]: this version achieves $O(\log_w n + \lg\lg n)$ *deterministic* update time, i.e. a $\lg\lg n$ overhead over the static version;
- Dynamic version via hashing [5]: this version achieves $O(\log_w n)$ *expected* update time. This is based on performing sketching 'more like' hashing. **OPEN:** Can this bound be achieved *with high probability*?

3 The General Idea

The underlying structure of a fusion tree is a B-tree, with branching factor $w^{1/5}$; actually, any small constant power suffices, since the height of the tree will be $\Theta(\log_w n)$. The main issue in obtaining this bound arises when searching a node during a predecessor search: we would like to achieve $O(1)$ time for this operation, which appears impossible since it seems to require at least reading in $O(w^{1/5} \cdot w) = O(w^{6/5})$ bits. However, this (and predecessor/successor) can actually be done in the desired time bound with $k^{O(1)}$ preprocessing. The main idea is to distinguish the set of keys in a node with less than w bits, which is the basis behind the next section. The rest of this lecture is all about how to achieve $O(1)$ time for a precessor/success search in a single fusion tree node.

4 Sketching

Since for each node in a fusion tree there are at most $k = w^{1/5}$ keys, it is feasible to represent these keys by only $w^{1/5}$ bits and still be comparable, given the lower bound of $\lceil \log w^{1/5} \rceil$ bits. Indeed, this can be accomplished as follows. Let the keys be $x_0 \leq x_1 \leq \cdots \leq x_{k-1}$; each key x_i can be represented as a path in a binary tree of depth w, where the left branch at the i-th node from the root is taken if the i-th most significant bit of x_i is 0, and otherwise the right branch is taken. Then if all k keys are overlaid on the same tree, then it is evident that the resulting paths branch out at at most $k - 1$ nodes (this is easily formalized by induction). In essence, at most $k - 1 = w^{1/5} - 1$ bits matter in ordering the x_i. See figure 1 for an example.

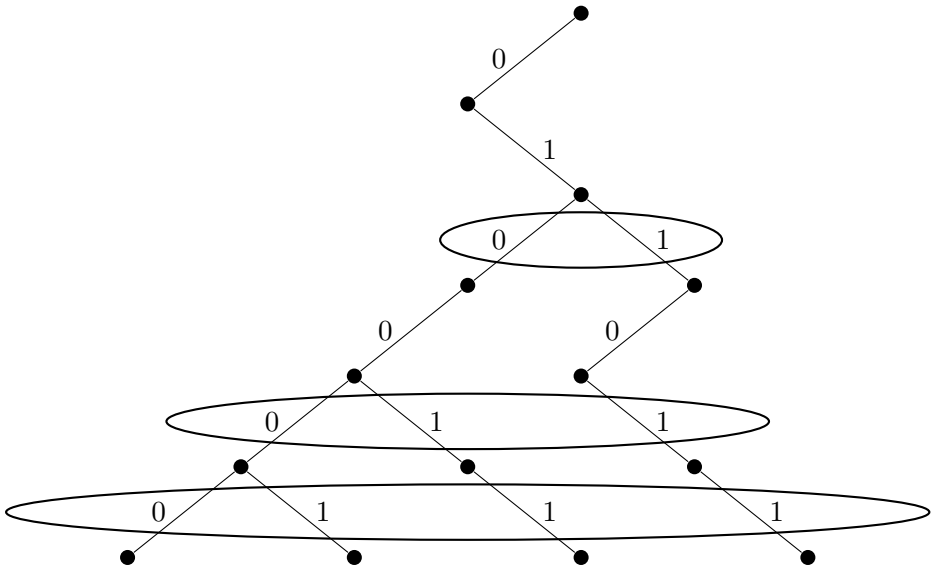

Figure 1: An example of the *sketch* function with 4 keys. The levels corresponding to the 3 bits sketched are circled.

In particular, let the bits of the corresponding nodes be in positions $b_0, b_1, \ldots, b_{r-1}$ (where $r \leq w^{1/5}$).

Then the *perfect sketch* of x (denoted by $sketch(x)$) is the r-bit string where the i-th bit is the b_i-th bit of x. Clearly, the sketch operation preserves order among the x_i, since each sketch keeps the bits that distinguish all the x_i in the right order. Sketching also allows all keys to be read in constant time, since each sketch has $O(w^{1/5})$ bits so the total size of all sketches is $O(kw^{1/5}) = O(w^{2/5}) = o(w)$ bits. Under some models, such as the AC^0 model, the perfect sketch operation is a single operation [3]. Later in this lecture we will see how to perform a sufficient approximation using multiplication and standard C-style operations.

However, this raises another problem. The search for a given query q may be such that q is not equal to any of the x_i (since there are no restrictions on the values of the arguments to predecessor/successor). Hence, the path of q in the binary tree may diverge from the other paths of x_i at a node which does not correspond to one of the bits $b_0, \ldots, b_{r-1}$; in that case, the location of $sketch(q)$ among the $sketch(x_i)$ will not necessarily be equivalent to the location of q among the x_i. In fact, the location of $sketch(q)$ can be completely different from the location of q. This can be resolved by the technique of *desketchifying* as discussed next.

5 Desketchifying

By modifying the search for q, we can still obtain the predecessor or successor of q without any additional (asymptotic) runtime overhead. Let x_i and x_{i+1} be the sketch neighbours of q, i.e. $sketch(x_i) \leq sketch(q) \leq sketch(x_{i+1})$. Then we determine the longest common prefix (equivalently, the lowest common ancestor in the fusion tree) of the actual elements between either q and x_i, or q and x_{i+1}. Suppose this prefix p has length y; then the node n corresponding to this prefix is the highest such that the path for q diverges from the path of every key in the fusion node. In particular, there are no keys in the child subtree of n which contains the path of q. Since the other child subtree of n contains a key of the fusion node (either x_i or x_{i+1}) it must contain the predecessor (or successor) of q. This can be determined as follows:

- If the $(y+1)$-st bit of q is 1, then q's predecessor belongs in the $p0$ subtree, so we search for the predecessor of $e = p011\cdots1$, i.e. the right-most tree node in the $p0$ subtree.
- If the $(y+1)$-st bit of q is 0, then q's successor belongs in the $p1$ subtree, so we search for the successor of $e = p100\cdots0$, i.e. the left-most tree node in the $p1$ substree.

In both cases, the search will successfully find the requisite key because all the sketch bits in the prefix of e and the target will match, and all the sketch bits in the suffix of e (following the first y bits) will be either the highest (when searching for predecessor) or lowest (when searching for successor). Once one of the predecessor/successor is found, the other can be determined simply by checking the appropriate adjacent sketch word in the fusion node. See figure 2 for an example.

There are still several issues remaining before our fusion tree will run with the desired time bounds under the word RAM model. First, we demonstrate how to perform an approximation of the perfect sketch in reasonable time. Then we show how to achieve constant runtime of two particular subroutines: finding the location of a $w^{1/5}$-bit integer among $w^{1/5}$ such integers, encoded as a $w^{2/5}$-bit word; and determining the most significant set bit of a w-bit word (this can be used to determine the length of the longest common prefix of two strings by XORing them together). This will conclude our description of the fusion tree's operation.

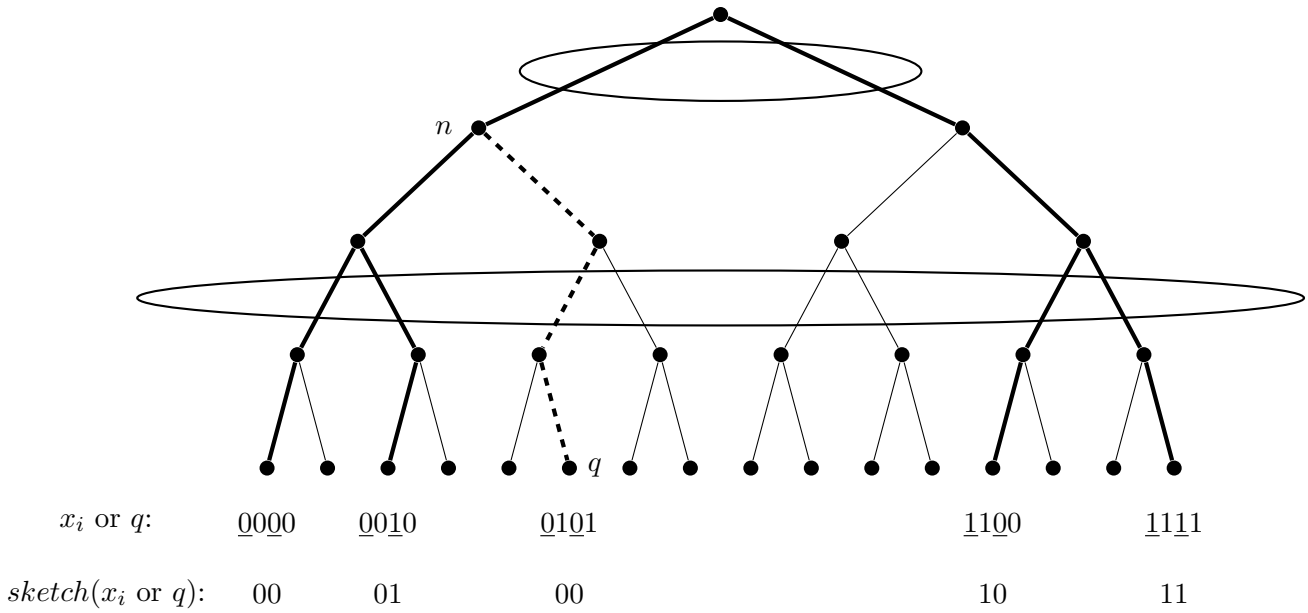

Figure 2: An example when the search query is not among the keys of the fusion node. The paths to the keys are bolded, whereas the path to the query q is dashed; the levels corresponding to the bits sketched are circled as before. Here, the sketch neighbours of q are x_0 and x_1, but x_0 is neither a predecessor nor successor of q.

6 Approximating Sketch

Although a perfect sketch is computable in $O(1)$ time as an AC^0 operation, we want a way to compute an *approximate* sketch on a Word RAM using just multiplication and other standard operations. The hard part about computing a sketch is getting all the bits we care about consecutive and succinct. So this approximate sketch will have all the important bits, spread out in some predictable pattern (independent of the word x we are sketching) of length $O(w^{4/5})$, with some additional garbage between them. But we will be able to apply masks to get just the bits we care about.

Let x' be x masked so it just has the important bits. So

$$x' = x \ \ AND \ \ \sum_{i=0}^{r-1} 2^{b_i}$$

where b_i represents the i^{th} important bit. Now multiply x' by some mask m (that will have set bits in positions m_j) to get

$$x' \cdot m = (\sum_{i=0}^{r-1} x_{b_i} 2^{b_i})(\sum_{j=0}^{r-1} 2^{m_j}) = \sum_{i=0}^{r-1} \sum_{j=0}^{r-1} x_{b_i} 2^{b_j + m_j}$$

Claim 1. *For any important bits $b_0, b_1, \ldots b_{r-1}$, we can choose $m_0, m_1, \ldots m_{r-1}$ such that*

1. *$b_j + m_j$ are distinct for all j. This means that there are no collisions when we add up all the bits.*

2. $b_0 + m_0 < b_1 + m_1 < \ldots b_{r-1} + m_{r-1}$. *This means that order of our important bits in x is preserved in $x' \cdot m$.*

3. $(b_{r-1} + m_{r-1}) - (b_0 + m_0) = O(w^{4/5})$. *Thus the span of the bits will be small.*

Proof. We'll choose some $m'_0, m'_1, \ldots m'_{r-1} < r_3$ such that $b_j + m'_j$ are all distinct modulo r^3. We'll prove this by induction. Suppose we have picked $m'_0, m'_1, \ldots m'_{t-1}$. Then m'_t must be different than $m'_i + b'_j - b_k \forall i, j, k$. There are t choices for i (since i can be any of the previous choices), and r choices for j, k. Thus, there are a total of $tr^2 < r^3$ things for m'_t to avoid, and we have r^3 choices, so we can always choose m'_t to avoid collisions. So this satisfies property (1).

To satisfy (2) and (3) we intuitively want to spread out $m_i + b_i$ by intervals of r^3. To do this we let

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

We claim without proof that spacing will make

$$m_0 + b_0 < \ldots < m_{r-1} + b_{r-1}$$

and also since $m_0 + b_0 \approx w$ and $m_{r-1} + b_{r-1} \approx w + r^4$ we will have $(b_{r-1} + m_{r-1}) - (b_0 + m_0) \approx r^4 = O(w^{4/5})$. So properties (2) and (3) will be satisfied. □

7 Parallel Comparison

We need to be able to find where $sketch(q)$ lies among the sketches of keys $sketch(x_0) < sketch(x_1) ... < sketch(x_{k-1})$ at a given node in constant time. We can do this parallel comparison with standard operations. We will use 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: $\texttt{1}sketch(x_0) \ldots \texttt{1}sketch(x_{k-1})$.

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 $\texttt{0}sketch(q) \ldots \texttt{0}sketch(q)$. We denote this by $sketch(q)^k$. If the sketches were 5 bits long, we would multiply $sketch(q)$ by `000001...000001` to get $sketch(q)^k$.

Then we subtract this value from the node sketch. This lets us subtract $\texttt{0}sketch(q)$ from each $\texttt{1}sketch(x_i)$ with a single operation: since $\texttt{1}sketch(x_i)$ is always bigger than $\texttt{0}sketch(q)$, 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 `100000...100000` 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. This is a special case of finding the index of the most significant 1 bit. To do this, we can multiply by `000001...000001`: 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. We can AND the result with 1111 and shift right to get the total number of 1s.

In summary, we have:

1. Compute the node sketch.
2. Compute $sketch(q)^k$.
3. Subtract $sketch(q)^k$ from the node sketch.
4. AND the difference with `100000...100000`.
5. Find the most significant bit / number of 1 bits of the result. This is the index of the 0 to 1 transition and the rank of the sketch.

8 Most Significant Set Bit

We conclude with the computation of the index of the most significant set bit of a w-bit word in $O(1)$ time, under the word RAM model. The solution is particularily messy, but it will use all the techniques that we have just seen for fusion trees. The first insight is to split the word x into $\sqrt{w}$ clusters of $\sqrt{w}$ bits. Our strategy is to identify the first non-empty cluster (this is the hardest part), and then the index of the first 1-bit within that cluster.

To illustrate the the following procedures, let $\sqrt{w} = 4$ and

$$x = \underbrace{0101}_{\sqrt{w}} \quad \underbrace{0000}_{\sqrt{w}} \quad \underbrace{1000}_{\sqrt{w}} \quad \underbrace{1101}_{\sqrt{w}}$$

1. Identifying non-empty clusters. This is done in $O(1)$ time with a series of bit tricks.

   (a) Identify which clusters have the first bit set. Compute bitwise AND between x and a constant F to get t_1

   $$\begin{array}{rcccc} x = & 0101 & 0000 & 1000 & 1101 \\ F = & 1000 & 1000 & 1000 & 1000 \\ \hline t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \end{array}$$

   (b) Identify if the remaining bits (not first bit of a cluster) are set. Compute bitwise XOR between the previous result and x to get t_2.

   $$\begin{array}{rcccc} x = & 0101 & 0000 & 1000 & 1101 \\ t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \\ \hline t_2 = & 0\underline{000} & 0\underline{000} & 0\underline{000} & 0\underline{000} \end{array}$$

Now we subtract t_2 from F, and if the 1-bit in a cluster of F end up getting borrowed (so that it becomes a 0), then we know that there was something in the corresponding cluster

$$\begin{array}{rcccc} F = & 1000 & 1000 & 1000 & 1000 \\ t_2 = & 0\underline{000} & 0\underline{000} & 0\underline{000} & 0\underline{000} \\ \hline t_3 = & \underline{0}xxx & \underline{1}000 & \underline{1}000 & \underline{0}xxx \end{array}$$

Finally XOR this result with F, to indicate that the remaining bits for a particular cluster are set

$$\begin{array}{rcccc} F = & 1000 & 1000 & 1000 & 1000 \\ t_3 = & \underline{0}xxx & \underline{1}000 & \underline{1}000 & \underline{0}xxx \\ \hline t_4 = & \underline{1}000 & \underline{0}000 & \underline{0}000 & \underline{1}000 \end{array}$$

(c) Now just OR the results from the previous steps, and this will tell us which clusters have set bits in them.

$$\begin{array}{rcccc} t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \\ t_4 = & \underline{1}000 & \underline{0}000 & \underline{0}000 & \underline{1}000 \\ \hline y = & \underline{1}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \end{array}$$

We can view y as the summary vector of all the $\sqrt{w}$ clusters.

2. Compute perfect sketch of y. We will need to do this for the next step, where we perform a parallel comparison and need multiple copies of $sketch(y)$ in a single word. Above we computed y which tells us which clusters have bits in them. Unfortunately these bits are spread, but we can compress them into a $\sqrt{w}$ word by using a perfect sketch. Fortunately, we know exactly how the b_is (the bits that we care about for the sketch) are spaced in this case. We care about the first bit of each $\sqrt{w}$ cluster, which is every other $\sqrt{w}$ bit. So

$$b_i = \sqrt{w} - 1 + i\sqrt{w}$$

To compute the sketch, we claim (without exact proof) that we can use

$$m_j = w - (\sqrt{w} - 1) - j\sqrt{w} + j$$

If we do this, then

$$b_j + m_j = w + (i - j)\sqrt{w} + j$$

will be distinct (no collisions) for $0 \leq i, j < \sqrt{w}$ and also conveniently

$$b_i + m_i = w + i$$

So to get our perfect sketch of $sketch(y)$, we just need to multiply $y \cdot m$ and shift it right by w.

3. Find first 1-bit in $sketch(y)$. This will tell us the first non-empty cluster of x. We use perform a parallel comparison of $sketch(y)$ to all of the $\sqrt{w}$ powers of 2. In our example these are

   0001
   0010
   0100
   1000

   This will tell us the first power of 2 that is greater than $sketch(y)$, which tells us the first set bit in $sketch(y)$. Because we reduced y to $sketch(y)$ which is $\sqrt{w}$ bits, the words generated for parallel comparison take up $\sqrt{w}(\sqrt{w}+1) < 2w$ bits, less than two words, so we can do this parallel comparison in $O(1)$ time.

4. Now that we know the first cluster c of x that has a set bit, we will find the first set bit d of c. To do this, first shift x right by $c \cdot \sqrt{w}$, bitwise AND the result with $\underbrace{11\ldots11}_{\sqrt{w}\text{ bits}}$ to get just the bits in that cluster. Now we perform the exact same type of parallel comparison as in the previous step, to find the first set bit d.

5. Finally, we compute the index of the most signficant set bit to be $c\sqrt{w}+d$.

Each step along the way takes $O(1)$ time, which makes this take $O(1)$ time overall.

References

[1] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, 1-7, 1990.

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

[3] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215:337-344, 1999.

[4] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54:3:13, 2007.

[5] R. Raman. Priority queues: Small, monotone, and trans-dichotomous. *Algorithms - ESA '96*, 121-137, 1996.