CS 224: Advanced Algorithms Fall 2014

Lecture 2 — September 4, 2014

Prof. Jelani Nelson *Scribe: David Liu*

1 Overview

In the last lecture we introduced the word RAM model and covered vEB trees to solve the dynamic predecessor problem. If the word size in the word RAM model is w, vEB trees can process update and query in time $\Theta(\log w)$.

In this lecture we will cover fusion trees to solve the static predecessor problem, with query time $O(\log_w n)$[1]. Note that since our pre-processing step may take polynomial time we cannot use the approach outlined in this lecture for sorting. We will not cover using fusion trees for the dynamic predecessor problem, but it has been shown that dynamic fusion trees can be constructed where query is still $O(\log_w n)$ and update is $O(\log_w n + \log\log n)$ [2] or in expectation $O(\log_w n)$ [3].

2 General Data Structure

The fusion tree data structure can basically be thought of a k-ary search tree. Such a tree is similar to a binary search tree, but each internal node, rather than having 1 value, has k values, and $k+1$ children nodes instead of the two child for a binary search tree. As with a binary search tree, all values in the node's leftmost subtree are less than or equal to the smallest value in the node, all values in the node's second leftmost subtree are greater than or equal to the smallest value in the node and less than or equal to the second smallest value in the node, etc.

Let $k = O(w^{1/5})$ (the reason for the 1/5 will become evident later). The height of the tree is then $\Theta(\log_k n) = \Theta(\log_w n)$. Thus, at each level in the tree if we can search a node in constant time, and thus determine which child our query element belongs in constant time, then the algorithm will run in time $\Theta(\log_w n)$ as desired.

3 Algorithm Details

3.1 "Basic Ingredients"

- Multiplication – We will use many times that we can multiply words in a single operation
- Sketch Compression – In order for a node to fit in a constant number of bits (i.e. for us to have any hope of processing a node in constant time) requires sketch compression
- Word-level Parallelism – In order to achieve constant time we will also need to use operations on our sketch compressions that in constant time process information about each item in the node.

- Most Significant Bit in Constant Time – It turns out that we will need to be able to find the most significant bit of a word in constant time.

3.2 Information in Sketch

First we form the tree. We sort the data, then create our k-ary search tree. This takes polynomial time which is ok for pre-processing.

One large difficulty that we will need to deal with is that, naively, storing a single node requires $w \cdot w^{1/5}$ bits, which does not fit in a constant number of words. Thus, we must make a sketch that contains some relevant information about the node that can fit into a constant number of words and is enough for us to be able to determine which subtree a query element falls into.

Consider a single fusion tree node, with elements $x_0 < x_1 < ... < x_{k-1}$. Instead of storing the entirety of each element in our sketch, we will end up storing only a few relevant bits. Consider drawing the binary tree of all possible strings of length w, where going left at the root node corresponds with strings whose largest bit is 0 and going right corresponds with strings whose largest bit is 1. The next level of the binary tree corresponds with the second largest bit, etc. Instead of keeping all bits for our items, we store only bits corresponding to branching levels. A level of the binary tree is called a branching level if there exists two elements x_i and x_j that are identical before the branching level and differ at the branching level. Intuitively, in the picture of the binary tree, this is where the paths corresponding to these two numbers branch.

Consider the example case where $w = 4$, and our elements are $x_0 = 0000$, $x_1 = 0010$, $x_2 = 1100$, and $x_3 = 1111$. The branching levels correspond with the leftmost bit (x_0 and x_1 split from x_2 and x_3) and the third leftmost bit (x_0 splits from x_1 and x_2 splits from x_3).

For any query q, define $\text{sk}(q)$ to be only the bits from branching levels, and let r denote the number of branching levels, noting that $r < k$. For example, $\text{sk}(0010)$ will be 01, as the first and third levels are the branching levels for our example fusion tree node.

Suppose $\text{sk}(q)$ is between $\text{sk}(x_i)$ and $\text{sk}(x_{i+1})$. We would want for q to be between x_i and x_{i+1}. However, this is not necessarily the case. In the above picture, suppose that we queried 0101. The sketch that we would get would be 00, which is not in the correct position.

Let y be the node in our binary tree where the paths for the query q and the path for x_i or x_{i+1} first diverges. If the path for q falls off the right, let $e = y0111\ldots1$, where here we denote y as the bits corresponding to the path from the root to the node y. If q falls off from the other paths to the left, then let $e = y1000\ldots0$. The claim is that $\text{sk}(e)$ fits among the $\text{sk}(x_i)$ precisely where q fits among the x_i (exercise for you).

Thus, after finding that $\text{sk}(q)$ is between $\text{sk}(x_i)$ and $\text{sk}(x_{i+1})$ (which we will show how to do later), we find y by taking $\max(\text{MSB}(x_i \text{ XOR } q), \text{MSB}(x_{i+1} \text{ XOR } q))$ (where MSB is most significant bit and XOR is bitwise exclusive or). We then form e, compute $\text{sk}(e)$, and find which sketches it is between, telling us the predecessor/sucessor of q.

3.3 Node Sketch Compression

We now need to compress the above so that a node can fit in a single word. Each item in a node will have a sketch that takes space $O(r^4) = O(k^4) = O(w^{4/5})$.

For element x_i, we do the following. First, we mask x_i by using bitwise and, so that for bits corresponding with branching levels, the original bit value in x_i is preserved, and all other bits are replaced with 0s (bitiwse and with a number that is 1 in branching levels, 0 elsewhere). As x has r non-masked bits we can write

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

We now wish to show that we can choose

$$m = \sum_{i=0}^{r-1} 2^{m_i}$$

such that

- all $b_i + m_j$ are distinct
- $b_0 + m_0 \leq b_1 + m_1 \leq \ldots \leq b_{r-1} + m_{r-1}$
- $(b_{r-1} + m_{r-1}) - (b_0 + m_0) = O(r^4)$

To show the first of these points, we will use induction. Supposing we've picked adequate $m'_1, \ldots, m'_{t-1}$, where $t < r$ and the $b_i + m'_j$ are distinct mod r^3 (an additional constraint that will prove useful), we note that we need only to choose m'_t such that it avoids all values $m'_i + b_j - b_z$. As there are at most $tr^2 < r^3$ of these values, there must exist at least one value modulo r^3 that we have not yet hit that we choose for m'_t.

Now, let $m_i = m'_i + 2ir^3 + (w - b_i)/r^3 * r^3$ (the last term being $w - b_i$ rounded down to the nearest multiple of r^3. The lecture had ir^3, but we actually need spacing of $2r^3$ because rounding down might involve subtracting up to another r^3). The first constraint is still satisfied. The second constraint is also satisfied as

$$(2i-1)r^3 + w < m_i + b_i < (2i+1)r^3 + w$$

Lastly, note that the $m_i + b_i$ are located in adjacent blocks where each block is size $O(r^3)$, so thus the difference of $b_0 + m_0$ and $b_{r-1} + m_{r-1}$ is $O(r^4)$.

Now, we note that

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

and thus the bits at location $b_i + m_i$ will give us the values of x_{b_i}. We can mask $x \cdot m$ so that only the $b_i + m_i$ bits are possibly non-zero, and then bit shift down so that $b_0 + m_0$ now corresponds with the smallest bit. Thus we have compute our sketch and store it in a contiguous block of $O(r^4)$ bits in constant time .

3.4 Sketch Manipulation

Let sketch(node), i.e. what we actually store at a node, consist of k consecutive blocks in memory. The first bit of the i'th block is a 1, and the rest of the block is $\text{sk}(x_i)$ (the compressed version). As this is size $O(r^4k) = O(w)$, it can fit in a constant number of words.

Now, for query q, define $\text{sk}^k(q)$ to consist of k consecutive blocks, where the first bit of the i'th block is a 0, and the rest of the block is $\text{sk}(q)$ (the compressed version, which as shown above we can compute in constant time). Note that to form $\text{sk}^k(q)$ in constant time we multiply $\text{sk}(q)$ by a number with k blocks each of form $00\ldots01$.

Now, we perform the operation $\text{sk(node)} - \text{sk}^k(q)$. Looking at the result, we see that each of our blocks now beings with either a 0 or a 1. The i'th block begins with a 0 if $\text{sk}(x_i) < \text{sk}(q)$, and begins with a 1 if $\text{sk}(q) \geq \text{sk}(x_i)$. Thus, in order to find the desired x_i and x_{i+1} such that $\text{sk}(q)$ falls between $\text{sk}(x_i)$ and $\text{sk}(x_{i+1})$, after masking out $\text{sk(node)} - \text{sk}^k(q)$ to include only the first bit of each block, we need only find the most significant bit, which corresponds to the smallest $\text{sk}(x_{i+1})$ that is greater than $\text{sk}(q)$. Instead of doing MSB, since we'll use what we do here later to show that we can do MSB in constant time, we note that the presence of a 1 in a block is monotone, i.e. $\text{sk}(x_{i+1})$ being larger that $\text{sk}(q)$ tells us that $\text{sk}(x_{i+2})$ is also larger than $\text{sk}(q)$, and thus also has a 1 leading its block. Thus, we can also count the number of 1s to help us find i.

To do this, we multiply our masked result by a word written as r consecutive blocks each of the form $00\ldots01$. In the resulting product, we note that at the most significant bit of the least significant block we can read off whether the box corresponding to $\text{sk}(x_{k-1})$ has a 1, and looking at this position in the k'th least significant block of the product tells us the number of sketches that have a 1. To access this value we then bit shift and mask.

3.5 Most Significant Bit

All that remains now is to be able to compute the most significant bit of a word in constant time. To compute MSB(x), let us conceptually split x into $\sqrt{w}$ blocks, each with $\sqrt{w}$ bits (the first $\sqrt{w}$ bits belong to one block, the next $\sqrt{w}$ bits to the next block, etc). Let F be the word consisting of $\sqrt{w}$ blocks of size $\sqrt{w}$ each, where each block is of form $100\ldots0$.

We first want to find which blocks have non-zero elements. Note that x AND F tells us whether the leading bit of a block is 1. Now, take x XOR (x AND F), which clears the leading bits from each block. We can do a similar trick as in the previous section, where we take $F-(x \text{ XOR } (x \text{ AND } F))$. The first bit of each block tells us whether there exists a non-zero element apart from the leading element (note we need to XOR with F to make it so that a 1 corresponds a 1 existing, rather than no 1s existing). Thus, finally we take (x AND F) OR (F XOR (F AND ($F-(x$ XOR (x AND $F)))$)), and now the first bit of each block tells us whether there exists a 1 in the block.

Now, we'll do more sketch manipulation. A lemma to be shown in homework states that, in our expression for x in the compression section, if all b_i are $i\sqrt{w} + \sqrt{w} - 1$, that there exists m such that multiplying by m puts all of the important bits in consecutive positions. Thus, doing this, then bit-shifting and masking, we are left with a word where the bits represent whether each block has a 1 in it.

Next, we want to find the most significant bit of this word of length $\sqrt{w}$. To do this, we can do

parallel computation on this as before against words of length $\sqrt{w}$ of the form $0 \ldots 00100 \ldots 0$. The i'th one of these words has a 1 in the i'th position from the right, and in our parallel computation algorithm these values correspond with $\mathrm{sk}(x_i)$ (this works since as with $\mathrm{sk}(x_I)$ they are monotone, and together they fit in a constant number of words). Our word telling us which blocks that contain a 1 corresponds to $\mathrm{sk}(q)$. As before, we can then in constant time find the block in which the first 1 occurs, as this is precisely where our word falls among the words of form $0 \ldots 00100 \ldots 0$. Now that we've found the block containing the most significant bit, to find where the first 1 occurs in this block, we then need only to repeat this process as the block is length $\sqrt{w}$ and we are searching for its most significant bit.

References

[1] Michael Fredman, Dan 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*, 54(3):13, 2007.

[3] Rajeev Raman. Priority Queues: Small, Monotone and Trans-dichotomous. *ESA*, 121-137, 1996.