CS 224: Advanced Algorithms Spring 2017

Lecture 2 — 26 January, 2017

Prof. Jelani Nelson *Scribe: Daniel Alabi Spring'17, David Liu Fall'14*

1 Overview

In the last lecture, we discussed the Word RAM model, Van Emde Boas trees, and X-fast tries which are the basis for Y-fast tries.

In this lecture we complete our discussion of Y-fast tries and discuss Fusion trees. Fusion trees show just how powerful the Word RAM model is. We exploit word-level parallelism to perform operations (like masking, comparisons) in constant time.

2 Y-fast Tries

Y-fast tries make use of both X-fast tries and balanced Binary Search Trees (BSTs) to improve the space usage to $\Theta(n)$ from $\Theta(nw)$ when using X-fast tries.

2.1 Use of X-fast Tries

Recall that w is the number of bits in a machine word and $u = 2^w$ is the universe. In the base level, we have u bits where the 1 bits are connected together in a doubly-linked list, for use in identifying the predecessor of an element. Then we OR the bits recursively, starting from the base level, until we get to the top level. Then to find the predecessor of an element $i \in [u]$, if the i-th bit is a 1, we just use the doubly-linked list to find the predecessor element. Else, we go up the trie until we find the first 1 on our way up, then we go down again by the provenance OR path of this 1 bit. If the path we follow down is to the right of this "first" 1, then when we get to the base level, the predecessor of i will be to the left of leaf node we land on. If we go the left of this "first" 1, then the leaf we end up at will be the predecessor of i.

We end up using $\Theta(nw)$ space because we use a hash table to store the 1-value leaf nodes and to represent the doubly-linked list.

2.2 With Balanced BSTs

Instead of storing all elements in the X-fast trie, we only store representatives from a Balanced BST (BBST). Each BBST group will store $\Theta(w)$ elements (between $w/2$ and $2w$ elements). The representative for each group can be the smallest element in the BBST. So at the base level of the X-fast trie, we will have $\Theta(\frac{n}{w})$ representatives from the BBSTs so that in total we use $\Theta(\frac{n}{w} \cdot w) = \Theta(n)$ space in the X-fast trie. In addition, each BBST stores $\Theta(w)$ elements and so collectively uses $\Theta(n)$ space. Thus, in total, $\Theta(n)$ space is used.

We have to maintain the requirement that each BBST msut have between $w/2$ and $2w$ elements. Insertions that will cause any BBST to have $2w+1$ elements must trigger a split of the BBST into two and the new representative will be added into the X-fast trie. Deletions that will cause any BBST to have less than $w/2$ must trigger a merge of the BBST with a neighboring BBST. Note that the splits and merges will happen after $\Theta(w)$ updates so the amortized cost of splits and merges is constant because of the size restriction of the BBSTs.

The trick of using representative elements from another data structure (in this case, the BBSTs) in a "macro" data structure (in this case, the X-fast trie) is called *indirection.* So we end up reducing the space requirement from $\Theta(nw)$ to $\Theta(n)$. The predecessor operation still takes $O(\log w)$ as before and updates take $O(\log w)$ asymptotically.

3 Fusion Trees

The Fusion tree data structure was introduced by Fredman-Willard [1] in 1993. With this data structure, we could use $\Theta(\log_w n)$ time for the predecessor problem. So combined with Y-fast tries, we could perform the predecessor operation in $O(n\sqrt{\log n})$ (since $O(\log w) = O(\sqrt{logn})$ for runtime $O(\min\{\log w, \log_w n\})$. We show how to use fusion trees to solve the static predecessor problem in $O(\log_w n)$ time using $O(n)$ space. Since we focus on the static version of the problem, we can preprocess the data (can be done in polynomial time – $O(n(w \log n)^c)$ time). We can also solve the dynamic predecessor problem in the same time [2, 3].

Fusion trees are based on (a, b)-trees which stores pointers to a number of children nodes between $a(\geq 2)$ and b. So each node in the tree stores between $a-1$ and $b-1$ keys. Usually, $a \approx \frac{b}{2}$. The height of an (a, b) tree is $\Theta(\log_b n)$ and query time is $\Theta(\log n)$ ($= height \cdot timepernode = \Theta(\frac{\log n}{\log b} \cdot \log b)$). If $b \approx w$, we can take advantage of the Word RAM model to search a node in constant time so that the query time becomes $\Theta(\log_w n)$. The question we aim to answer is: *how can we create a node we can search in constant time?*. The key insight it to store a compressed version of the keys (of which there are $r = w^{1/5}$ of them). The goal is to compress each key to use $O(r^4) = O(w^{4/5})$ space which would make each node use $O(w)$ space and then we would take advantage of word-level parallelism to search a node in constant time. There are roughly three problems we need to tackle:

- *Sketching:* We need to sketch/compress the r keys so that each key uses $O(r^4) = O(w^{4/5})$ space. From henceforth, let us denote `sk(y)` as the sketched bits of an item y.
- *Parallel Comparisons:* We need to compare the sketch of a query (say q) to the sketches of all elements in a node. In other words, if $x_0, \ldots, x_{r-1}$ is stored in a node, we need to be able to compare `sk(q)` to all $\mathtt{sk}(\mathtt{x_i})$ simultaneously.
- *MSB in Constant Time*: We need to find the position of the Most Significant (Set) Bit in constant time. Recall that we are counting from left (highest) to right (lowest). Let `MSB(q)` return the most significant set bit position in q. A naive way to determine the MSB is to do either linear or binary search to determine the first 1 from the left. But in the Word RAM model, we can do this in **constant time**!

3.1 Sketching the Keys

For each node, there are r keys: $x_0 < x_1 < \cdots < x_{r-1}$.

The sketch $\texttt{sk}(\texttt{x}_\texttt{i})$ for each key x_i will correspond to the "branch" bits, the bits at bit positions where at least two keys first diverge in bit value. These "branch" bits can be used to differentiate **all** the x_is.

For example, suppose we have the following four keys: $x_0 = 0000, x_1 = 0010, x_2 = 1100$, and $x_3 = 1111$. Then x_0 and x_1 diverge at the 1st bit position and x_1 and x_2 diverge at the 3rd position. So, the branch bits will be derived from the 1st and 3rd bit positions for each key. Thus, $\texttt{sk}(\texttt{x}_0) = 00$, $\texttt{sk}(\texttt{x}_1) = 01$, $\texttt{sk}(\texttt{x}_2) = 10$, and $\texttt{sk}(\texttt{x}_3) = 11$. It might be instructive for you to draw a binary tree where starting from the root, a left branch corresponds to a 0 and a right branch corresponds to a 1. The height of the tree will be w and the branch bits will be gotten from the levels where at least two keys branch out from (downwards). Note that there are at most $r - 1$ "branch" bits.

Ideally, we want to use our sketching scheme as is. But a new query q could end up in the wrong interval. More concretely, we would like

$$\texttt{sk}(\texttt{x}_\texttt{i}) \leq \texttt{sk}(\texttt{q}) \leq \texttt{sk}(\texttt{x}_{\texttt{i+1}})$$

to imply

$$x_i \leq q \leq x_{i+1}$$

But this is not the case right now. For example, using the four keys above to get the branch bit levels , then $\texttt{sk}(\texttt{0101}) = 00$ but clearly its sketch is in the wrong place – between x_0 and x_1. The query $q = 0101$ is between x_1 and x_2 so we need to "fix" the procedure so that $\texttt{sk}(\texttt{x}_1) \leq \texttt{sk}(\texttt{q}) \leq \texttt{sk}(\texttt{x}_2)$.

3.1.1 The Sketching "Fix"

Notice that the reason $\texttt{sk}(\texttt{q})$ ends up in the wrong interval is that there is a bit level for which no 2 keys from $x_0, \ldots, x_{r-1}$ diverge from until the query q shows up. In our example, q diverges down-right from the path followed by both x_0 and x_1.

The key insight for our "fix" is to look at the first node where q diverges from some x_i. Let y be the shared prefix between q and x_i until they diverge. There are two cases to consider:

1. *q falls to the left*: Set $e = y1000\cdots0$
2. *q falls to the right*: Set $e = y0111\cdots1$

Notice that $\texttt{sk}(\texttt{e})$ fits in the interval where q fits (in the tree determined by our keys). Specifically, $\texttt{sk}(\texttt{x}_1) \leq \texttt{sk}(\texttt{e}) \leq \texttt{sk}(\texttt{x}_2)$ since in our example, e would be 0011 so $\texttt{sk}(\texttt{e}) = 01$ And this is what we want!

Now, *how do we (programatically) determine y?.* We use our MSB routine to obtain

$$P = \max(\texttt{MSB}(\texttt{q} \oplus \texttt{x}_\texttt{i}), \texttt{MSB}(\texttt{q} \oplus \texttt{x}_{\texttt{i+1}}))$$

where $\oplus$ is the XOR operation and (x_i, x_{i+1}) is the interval in the tree sandwiching sk(q). In our example, $x_i = x_0, x_{i+1} = x_1$ since sk(q) $= 00 \in [00, 01]$.

Then we can use P to determine y by copying over the most significant P bits of x_i. Note that we are still yet to discuss how MSB actually works but we know that it is a constant time operation.

3.1.2 Computing sk(x)

The essence of computing the sketches in the first place is so that each node uses $O(w)$ space. To do this, for each key x, sk(x) would have to occupy $O(r^4) = O(w^{4/5})$ bits.

Suppose we have masked out the important bits for each key, then we can write as

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

Consider "magic number" m such that

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

Lemma 1. *We can choose the m_i's (still w-bit numbers) such that:*

1. *All the $b_i + m_j$ are distinct (r^2 of these sums)*
2. *$b_o + m_0 < b_1 + m_1 < \cdots < b_{r-1} + m_{r-1}$*
3. *$(b_{r-1} - m_{r-1}) - (b_0 + m_0) = O(r^4)$*

The point of lemma 1 is the sum

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

For this sum, since the $b_i + m_j$ are distinct, we do not have to worry about carries and thus we could mask out r positions of $x \cdot m$, from positions $b_0 + m_0$ to $b_{r-1} + m_{r-1}$. But the guarantee we have is that these r positions fit in a contiguous block of $O(r^4)$ bits. We can mask out the this $O(r^4)$ block of bits. In essence, each x occupied $O(w) = O(r^5)$ space but now occupies $O(w^{4/5}) = O(r^4)$ space.

Note that the m_i's are chosen during the preprocessing step. The steps to compute sk(x) (for key x) are:

1. Mask x to keep only branch bits.
2. Compute $x \cdot m$ ($=z$).
3. Mask z to only keep the $b_i + m_i$ bit positions.
4. Shift down so that $b_0 + m_0$ becomes the Least Significant Bit. Then the 0th to $O(r^4)$-th consecutive bits starting from the LSB (moving from right to left) will be the sketch of x.

Proof of Lemma 1 We prove the lemma by induction. Suppose we have selected $m'_1, m'_2, \dots, m'_{t-1}$ such that all sums $b_i + m'_j$ are distinct $(\mod r^3)$ where $t < r$. Then we want to choose the m'_t such that

$$m_t + b_i \neq b_j + m'_k \forall i, j, k$$

(or $m_t = b_j + m'_k - b_j$ of which there are $< r^3$ choices). Now set $m_i = m'_i + i \cdot 2r^3 + (w - b_i)$ where the $w - b_i$ are rounded down the nearest multiple of r^3. Note that m'_i are distinct and since the other summands are a multiple of r^3, then $\mod r^3$, the m_i's must be distinct. And note that we add $i \cdot 2r^3$ so that the m_i's can be increasing and distinct numbers (for part 2 of the lemma).

3.2 Parallel Comparisons

Now we know how to compute the sketches for the keys and we know we can use space $O(r^4) = O(w^{4/5})$. But the goal is to compare the sketch of the query $\mathtt{sk(q)}$ [1] to $\mathtt{sk(x_i)}$ simultaneously.

Suppose a node has values $x_0 < x_1 < \cdots < x_{r-1}$ then we define

- $\mathtt{sk(node)} = [1\mathtt{sk(x_0)}|1\mathtt{sk(x_1)}|\cdots|1\mathtt{sk(x_{r-1})}]$
- $\mathtt{sk^r(q)} = [0\mathtt{sk(q)}|0\mathtt{sk(q)}|\cdots|0\mathtt{sk(q)}]$ which can be computed as $\mathtt{sk(q)} \cdot [0\cdots01|0\cdots01|\cdots|0\cdots01]$ again exploting word-level parallelism for multiplication.

Then consider $\mathtt{sk(node)} - \mathtt{sk^r(q)}$ which would look like
$[0/1\ldots|0/1\ldots|\cdots|0/1\ldots]$. In the $O(r^4)$ blocks, we only care about the left most bit (per block) so we can mask out the rest so we get $[0/1\cdots00|0/1\cdots00|\cdots|0/1\cdots00]$. Note that the left most bit per block containing $\mathtt{sk(x_i)}$ will be 1 iff $\mathtt{sk(x_i)} \leq \mathtt{sk(q)}$. So we can determine the interval (amongst the $\mathtt{sk(x_0)}, \dots, \mathtt{sk(x_{r-1})}$) where $\mathtt{sk(q)}$ should be by finding the most significant (set) bit again using our constant time MSB routine.

3.3 Most Significant Set Bit

Throughout, we assumed that we had a routine `MSB(x)` that can determine the most significant set bit in constant time for any integer N.

We now describe how to implement `MSB(x)`. 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 $F - (x$ XOR (x 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.

[1] we actually use `sk(e)` the "fixed" sketch of q from the previous sections

Now, we'll do more sketch manipulation. A lemma to be shown in problem set 2 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 $0\ldots00100\ldots0$. 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 $\texttt{sk}(\texttt{x}_\texttt{i})$ (this works since as with $\texttt{sk}(\texttt{x}_\texttt{i})$ 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 $\texttt{sk}(\texttt{q})$. 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 $0\ldots00100\ldots0$. 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 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*, 54(3): 13, 2007.

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