CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 9 — February 14, 2023

Prof. Jelani Nelson *Scribe: Martin Toft*

1 Overview

In the last lecture we started looking the Word RAM model. In this lecture we will continue looking the Word RAM model, more specifically the data-structures y-fast tries and Fusion trees.

2 y-fast tries

We ended last lecture by looking at x-fast tries, which have:

- $O(nw)$ space
- $O(\lg \lg u)$ query
- $O(\lg u)$ update

y-fast tries improves X-fast tries, and have

- $O(n)$ space
- Same query time
- $O(\lg \lg u)$ (amorized) update

2.1 How it works

y-fast tries are made up of two data structures: The top half is x-fast trie, and the lower half consists of balanced binary search trees (BBSTs).

The keys are divided into groups of $O(\lg u)$ consecutive elements and stored in a BBST. To facilitate efficient insertion and deletion, each group contains at least w/4 and at most 2*w elements. For each group, a representative r is chosen. These representatives are stored in the x-fast trie. The groups have pointers to their predecessor and successor.

What this separation of x-fast trie and BBSTs means for our search is that we first search the x-fast trie and find the correct representative, then search the BBST and find the correct element.

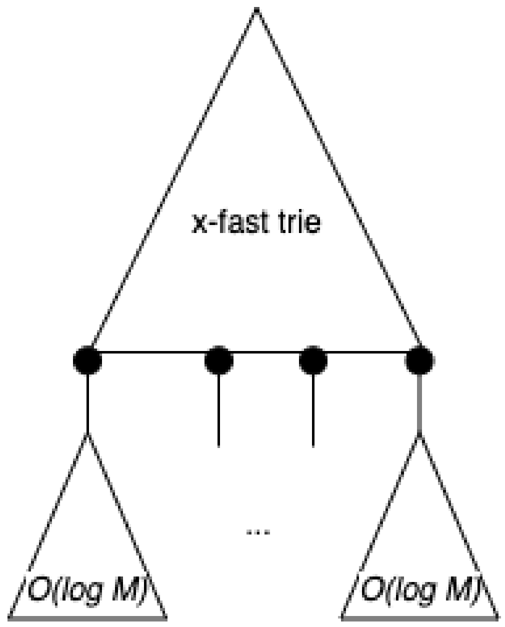

Figure 1: Y-trie

2.2 Runtimes and space

Since the x-fast trie stores $O(n/\log M)$ representatives and each representative occurs in $O(\log M)$ hash tables, it requires $O(n/\log M) * O(\log M) = O(n)$ space. The BBST store n elements in total which uses $O(n)$ space. The total space used by y-fast tries in total is therefore $O(n) + O(n) = O(n)$.

Querying is done by first finding the correct group using x-fast query, then searching through that group's BBST. This is done in $O(\lg\lg u)$ time.

When it comes to insertion, this is done by first finding the correct group, then inserting into group. If the group becomes too big, i.e., larger than 2w, we split the BBST into two, and remove its representative. We then pick a representative for each of the new BBSTs and insert these into the x-fast trie. One of the new representatives simply replace the old one, while the other needs to be inserted into a new slot, i.e., by setting a 0 to a 1. When setting a 0 to a 1, this forces us to move up x-fast trie, updating the value of the internal nodes, which takes $O(w)$ time. However, this is done at most once for every $O(w)$ insertion, hence it takes $O(1)$ time (amortized). In total, it takes $O(\lg\lg u)$ time to find a group + $O(1)$ amortized to handle splitting group and adding new rep to x-fast tree, which in total is equals to $O(\lg\lg u)$.

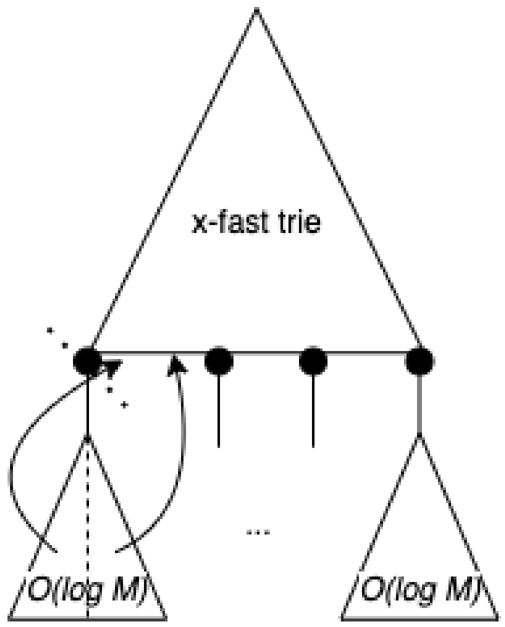

Figure 2: BBST split into two new BBSTs where new representatives are inserted into x-fast trie

3 Fusion trees

A fusion tree is essentially a B-tree with branching factor of $w^{(1/5)}$. There two types of fusion trees: static and dynamic. Dynamic fusion trees will not be discussed in this class, but those who are interested can look it up [2]. Instead, we will focus on static fusion trees [1]. Our goal: Static predecessor in $O(\lg wn)$ time.

3.1 How it works

To achieve the desired runtimes for updates and queries, the fusion tree must be able to search a node containing up to $w^{(1/5)}$ keys in constant time. This is done by "sketching", which essentially compress the keys so that they all fit into one machine word, which allows comparisons to be done in parallel.

Reaching the required solution for predecessor in $O(\lg wn)$ time requires 4 main "ingredients"/computations:

1. Sketching (has nothing to do with the research field of the prof.)
2. Word level parallelism, i.e., parallel comparison achieved through compression
3. Power of multiplication
4. Finding the Most Significant Set Bit (MSSB) in $O(1)$ time (connected to point (3))
    - Note: Some machines today have MSSB built into their CPU, so that we don't have to think about finding it. We will however assume that our machine don't have MSSB built into its CPU, and define how to go forward with finding it.

3.2 Sketching

As we noted earlier, sketching is a kind of compression where we want each fusion node to fit in a single machine word. Sketching is done by storing what's called *sketches* of our keys. A sketch is made by grouping all the positions where we have branching (i.e., where both branches of an internal node reach out to nodes). To distinguish two paths, it is sufficient to look at their branching point, i.e., the first bit where any two keys differ. Since there is a maximum of k keys, there will be a maximum of k-1 branching points, which means that we can identify a key using a maximum of k-1 bits. And so, the number of branch bits is < k = w$^{(1/5)}$. An important property of the sketch function is that it preserves the order of the keys, as x_0 < ... < x_{k-1} => sk(x_0) < ... < sk(x_{k-1}) where sk(x_i) is the sketch of the key x_i.

Take Figure 4 below as an example of sketching, where u = 16 and w = 4, we have the keys 0000, 0010, 1100, 1111, and the dashed lines represent branching in that node. This branching gives us the sketches 00, 01, 10 and 11 for the keys, respectively.

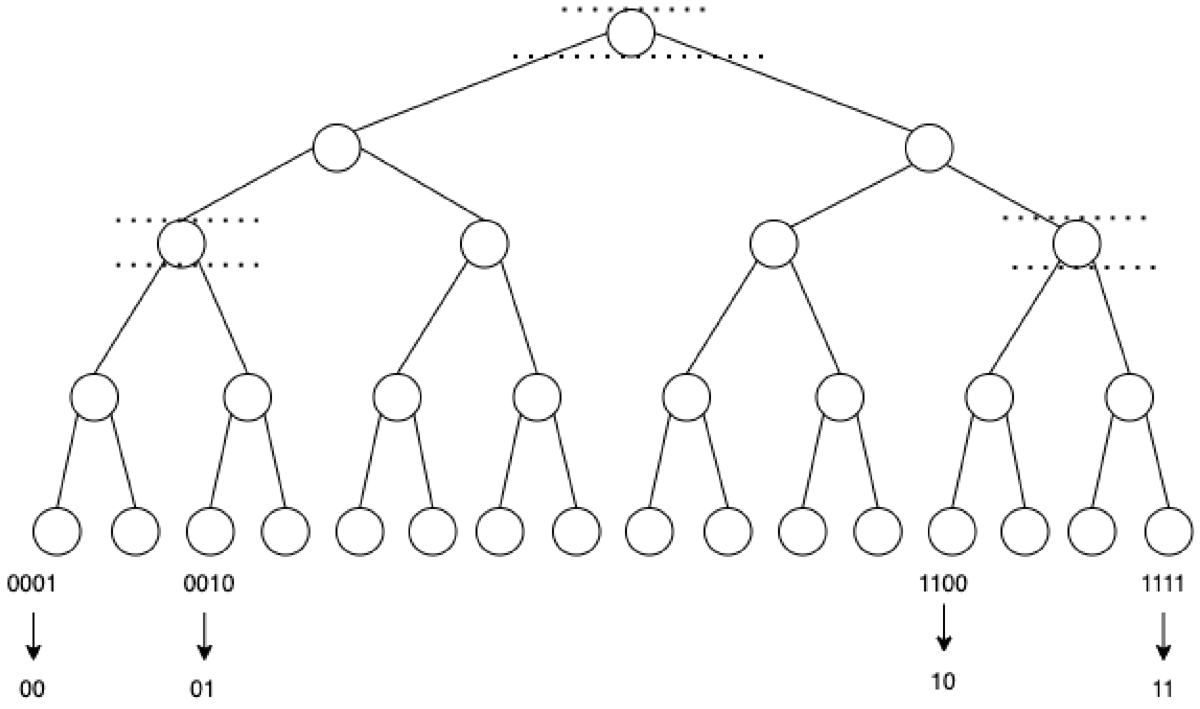

Figure 3: An example of sketching

Problem: What if we add a new key, and this results in two keys having the same sketch, as shown in Figure 4 below?
Fix: Say that sk(x_i) = pred(sk(q)), also learn x_{i+1}. First find highest point where the nodes with the same key branch differently, which we will call y. If the newly added key was added to the right of y (if we fell off y to the right), we set e = y011...1. If we fell off y to the left, we set e = y100...0. If we have multiple leaves with same sketch, we do this successively.
Claim: If we search pred(sk(e)), we will find the correct child branch.

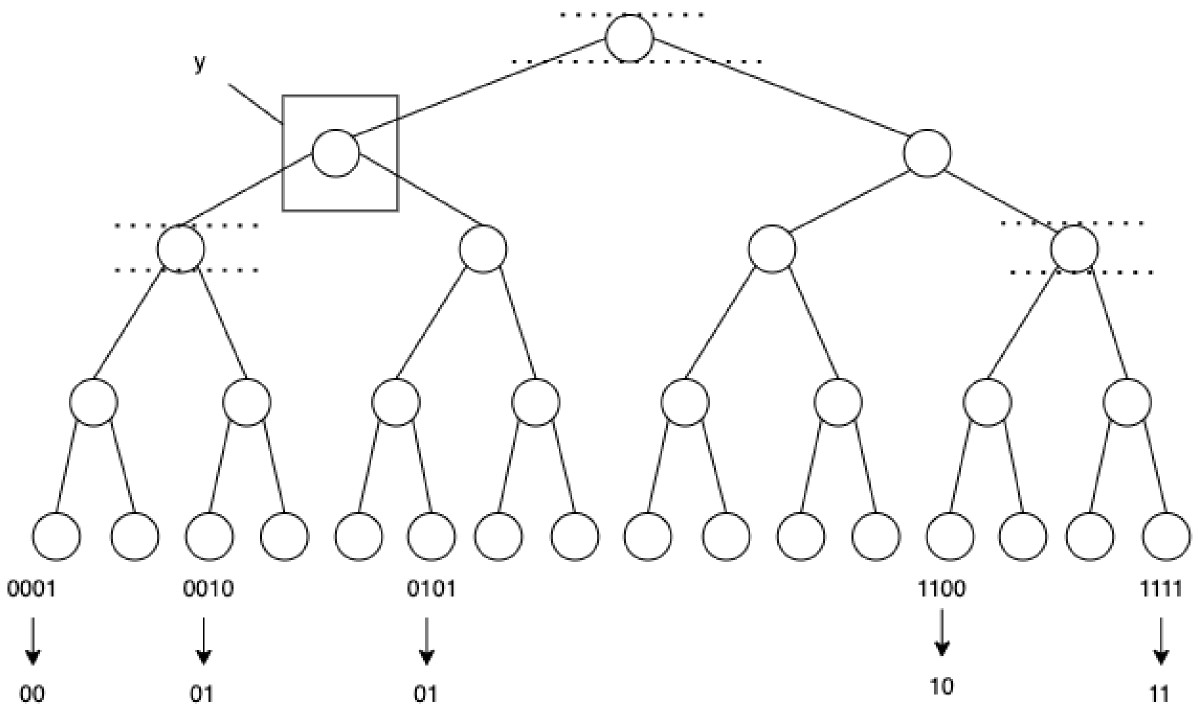

Figure 4: Sketch after adding a new key to tree

3.3 Word level parallelism (parallel comparison)

The purpose of the compression achieved by sketching is to allow all of the keys to be stored in one w-bit word. The sketch of a fusion node is the bit string sk(node) = 1sk(x_0)1sk(x_1)...1sk(x_{k-1}). Essentially, we have packed all sketch words together in one string by prepending a set bit to each of them. The sketch function uses b $\leq r^4$ bits. Each block then uses 1 + b $\leq w^{4/5}$ bits. Because k $\leq w^{1/5}$, the total number of bits in the node sketch of a fusion node is maximum w, which fit into our words.

We can make each query sketch sk(q) as long as sk(node), so that each word in sk(node) can be compared with sk(q) in one operation, demonstrating word-level parallelism. We compute sk(node) in preprocessing. In runtime, we compute sk(q) = 0sk(q)0sk(q)...0sk(q). We then take the difference of sk(node) and sk(q). The leading bit of each block in the difference will be 1 iff sk(q) $\leq$ sk(x_i), otherwise 0. Since x_i's in the fusion node is in sorted order, the difference between the two sketches will for a while be 0s with some other stuff, then 1s with some other stuff, i.e., sk(node) - sk(q) = 0...0...1...1...1... .

Note: It is important to note that structure is not arbitrary: We know that these blocks of bits are of some width, and that we only care about some parts of them. If we know how many blocks that contain 1 in the leading bit, we could figure out what block is the leftmost block with 1, i.e., the MSSB, which leads us to our next part.

3.4 Power of multiplication

In the last subsection, we ended up with the difference of sk(node) and sk(q) being 0s with some other stuff, followed by 1s with some other stuff. We don't care about the other stuff, so by taking the bitwise AND of the difference and the constant $(10^b)^k$, we clear everything except from the leading bit of each block. After the mulitplication, we can mask by packing together all the ones in the first block and masking away all the other blocks to 0. Finally, we shift right, so that we have a group of 0s followed by a group of 1s. This prepares us for our next and last step, finding the MSSB.

3.5 Approximating the sketch

Before we look at finding MSSB, there is one other thing we need to look at, namely approximating the sketch. With only the standard word operations, such as those of the C programming language, it can be difficult to compute a perfect sketch in constant time. By instead packing the sketch bits into an approximate sketch which has all the important bits but also some irrelevant bits we are able to achieve this. Just like the perfect sketch, the approximate sketch preserves the order of the keys, giving us that sk(x_0) < ... < sk(x_{k-1}).

We compute the approximate sketch by using a bitwise AND between sk(node) and $\sum_{i=0}^{r-1} 2^{b_i}$ which serves as a mask to remove all the non-sketch bits from the key, followed by multiplication with some constant m that shifts the sketch bits into a small range, then masking out all but the shifted sketch bits.

Claim: Given keys $x_0 < ... < x_{k-1}$ and branch bits $b_0 < ... < b_{r-1}$ (r < k), there exists a number m $\in$ 0, ..., u-1, where m = $\sum_{i=1}^{r} 2^{m_i}$ such that

1. For all i,j $\neq$ i',j', m_i + b_j $\neq$ $m_{i'}$ + $b_{j'}$, i.e., m_i + b_j are distinct pairs for all (i, j). This will ensure that the sketch bits don't get altered by the multiplication in such a way that we can't reverse it to find keys later on.

2. m_0 + b_0 < m_1 + b_1 < ..., i.e., m_i + b_i is ordered in strictly increasing order.

3. (m_{r-1} + b_{r-1}) - (m_0 + b_0) $\leq r^4$, i.e., the sketch bits are packaged into a range of maximum size r^4

Proof: Let m_0 = 0 and 1 < t $\leq$ r. Suppose that we already found m_1, ..., m_{t-1}. Pick the smallest key m_t where both point (1) and (2) is satisfied. (1) requires that $m_t \neq b_i$ - b_j + m_s for 1 $\leq$ i, j $\leq$ r and 1 $\leq$ s $\leq$ t-1. This implies that there are less than t$r^2 \leq r^3$ values that m_t must avoid. Since m_t is chosen to be minimal, we further have that (b_t + m_t) $\leq$ (b_{t-1} + m_{t-1}) + r^3, which implies point (3), and thereby proves our claim.

3.6 MSSB

In the next lecture we will look at the last "ingredient", namely MSSB, and how we can find this in constant time.

References

[1] Willard Fredmund. Blasting through the information theoretic barrier with fusion trees. *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, 1990.

[2] Rajeev Raman. Priority queues: small, monotone and trans-dichotomous. *Fourth Annual European Symposium on Algorithms*, page 121–137, 1996.