CS 4973/7280: Data Str & Alg Scalable Comp Spring 2025

Lecture 2 — January 13, 2025

Prof. Prashant Pandey *Scribe: Dhruv Chauhan*

1 Overview

In the last lecture we covered Van Emde Boas Trees. We talked about how operations like **insert, delete, successor, predecessor, and find can be done in worst case $O(lglgU)$ time, but takes $O(U)$ space where U is the universe size.** This is because the last level of the tree will have a total of U-size bit array. In this lecture we will talk about how we can **reduce the space to be $O(n)$** (linear in number of keys) while keeping other **operations amortized $O(lglgU)$ time**.
Note: We can only use integer keys because they can be bounded in terms of their length.
Delete pseudocode is given below.

1.1 Delete in a VEB Tree (Version 5)

Algorithm 1 Delete Operation in Van Emde Boas Tree

```
1:  procedure DELETE(V, x)                                  ▷ Case for deleting v.min
2:      if x = v.min then
3:          i = v.summary.min
4:          if i = φ then
5:              v.min = v.max = φ
6:              return
7:          end if
8:          x = v.min = index(i, v.cluster[i].min)
9:      end if
                                                            ▷ Deleting the element
10:     DELETE(v.cluster[high(x)], low(x))
11:     if v.cluster[high(x)].min = φ then
12:         v.summary.delete(high(x))
13:     end if
                                                            ▷ Case for deleting v.max
14:     if x = v.max then
15:         if v.summary.max = φ then
16:             v.max = v.min
17:         else
18:             i = v.summary.max
19:             v.max = index(i, v.cluster[i].max)
20:         end if
21:     end if
22: end procedure
```

2 X-Fast Trees

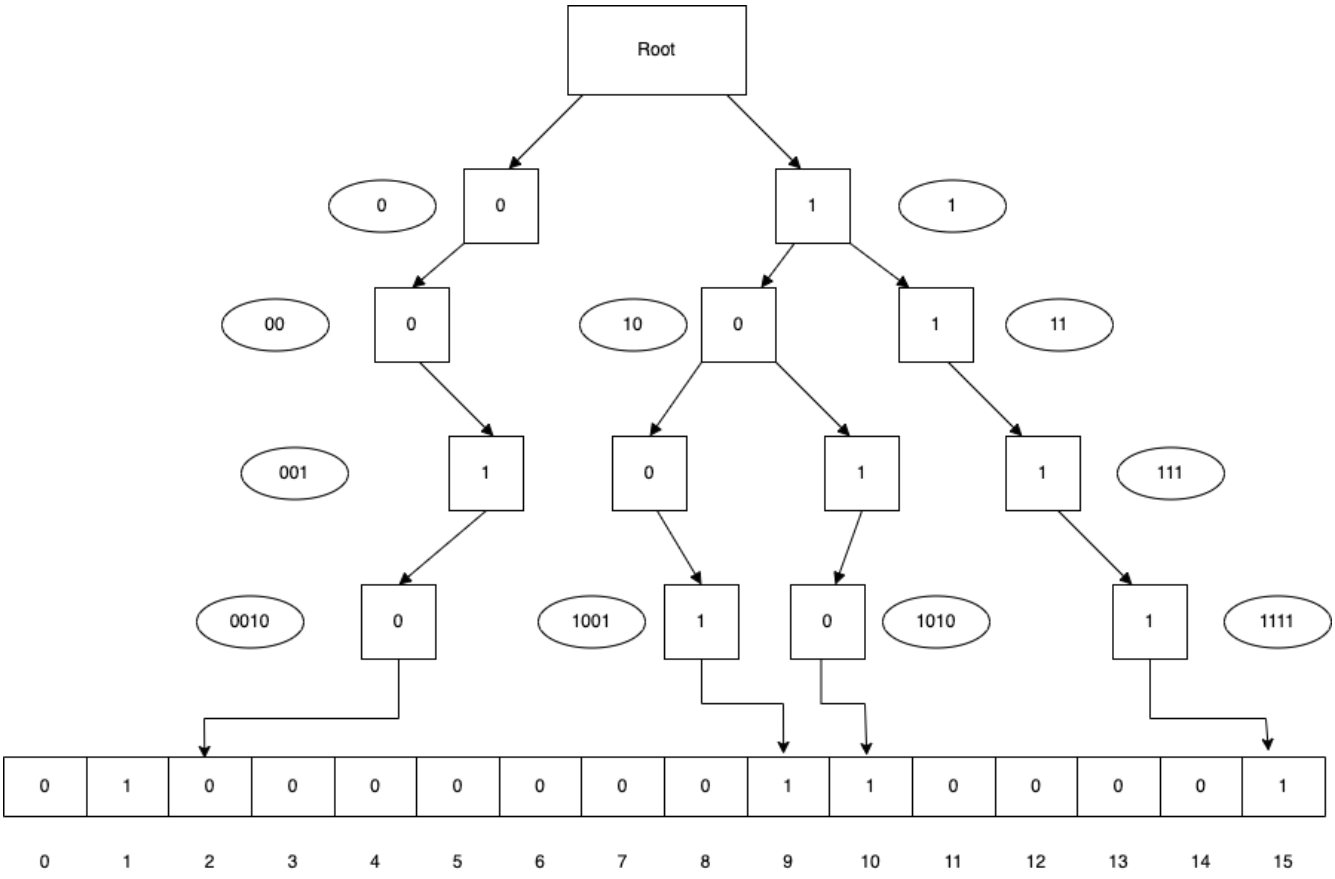
The motivating question for today's lecture is reducing space from $O(U)$ to $O(n)$. If we have a 64-bit universe space, the amount of memory a VEB would use is around 2^{31} GB, which is impractical with the machines we use. So, instead of being linear in the universe space, we will try and be linear in the number of keys.

To achieve this we will use a structure called **X-Fast trees**. This is a data structure that comprises of a **trie, linked-list, and a set of hash tables**, and so it is worthwhile to look at these structures.

2.1 Structure of X-Fast Trees

2.1.1 Trie

A trie, in effect, is a tree in which every node is labelled according to the path which goes from the root to that node. If we have a set of 4 keys $\{2, 9, 10, 15\}$ in our 4-bit universe, then the trie would look something like below



Note: The labels for each node are in the circles, which is just the path taken to that node

2.1.2 Linked list

At the bottom-most layer, it would be useful to keep the keys stored as a doubly linked list in order. This will help in the predecessor/successor queries discussed later.

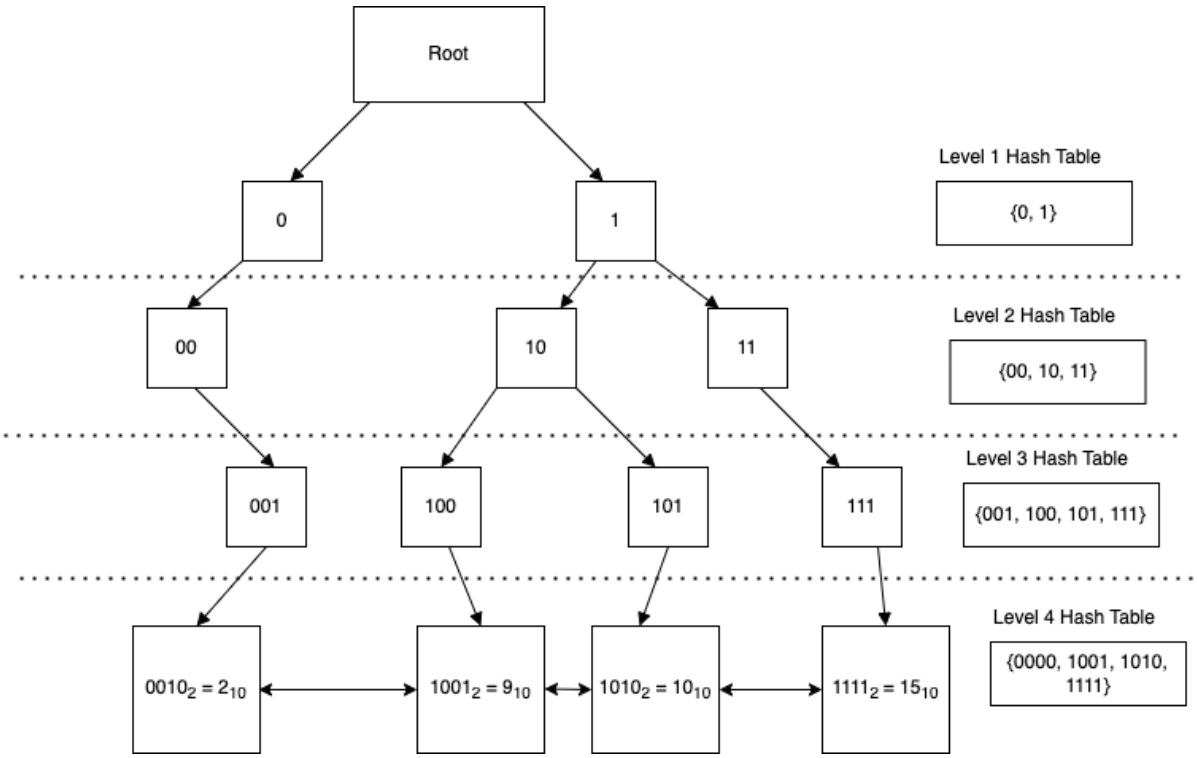
2.1.3 Hash Table

For each layer of the trie, we store a hash table with all the nodes present in that level of the trie. If our universe $U = 2^l$, then there will be $l + 1$ **levels** and a hash table associated with each level. This will help with searching for a particular node being present in the tree in amortized $O(1)$ time.

2.1.4 Auxiliary pointers

For an X-Fast tree, let N be an internal node with **only one child**. Then, this node N will contain an auxiliary pointer to a leaf of the tree. If N only has a right child, then it will store a pointer to that child's left-most descendant. If N only has a left child, then it will store a pointer to that child's right-most descendant.

These pointers will be used in a successor/predecessor search; for now it is enough to keep in mind that they exist. As an example, assume our trie also had 0011_2 in it. This would be a node to the right of 0010_2 as a child of the node 001. In this case, for $N = 00$ (since 00 has only one child), the auxiliary pointer would be to 0010_2 as compared to 0011_2. **Since** 001 **was a right child of** 00**, we looked at** 001**'s leftmost descendant, which is** 0010_2**.**



2.2 Operations using X-Fast Trees

2.2.1 Find

To find if a key X exists in the tree, all we need to do is query for it on the last level's hash table. Since the table contains all keys, we should be able to do the **find operation in $O(1)$ time**.

2.2.2 Insert/Delete

There is nothing special about inserting or deleting items in an X-Fast tree. For either operation, we will have to update the auxiliary pointers and hash tables on every level of the tree, and **thus will spend $O(lgU)$ time**.

2.3 Predecessor/Successor

In X-Fast trees, because of the hash tables stored on each level (compared to pointers to the next level), it is constant time to jump between levels. We are able to use this to binary search different levels of the tree to find predecessor/successor.

The trivial case is when the key already exists in the tree, and so we just query the bottom-most level every time to check if that is the case. If so, we can just return the key to its right using the linked-list in constant time.

In the case the key does not exist, we start binary searching the levels. The idea is to find the top-most level where the prefix of our key matches a node in the tree. Once we find that, we basically use the auxiliary pointer to find the key's successor/predecessor (depending on whether the matched prefix node had a left or right child) in constant time.

Note that once the prefix matches, the node is guaranteed to have only one child, and thus an auxiliary pointer (by construction). This is because if that node is the lowest level it matches a key's prefix with, it cannot have the key's next digit as a child, so it has at most one child. But also since that prefix exists in the tree, it must have at least one child - and thus exactly one child.

Let's take key as $11_{10} = 1101_2$ for which we want to find the successor. Since we binary search, we will take the first 2 symbols, which is 11, and check on level 2's hash table to see if the node exists. Since it does, now we check at level 3's hash table for 110. Since it does not exist, we go back to 11 at level to and look at its auxiliary pointer. Since it had a right child, we know that the auxiliary pointer will point to a number greater than our key, and in fact the successor of our key. So in this case, we can return $1111_2 = 15_{10}$ as our successor. In this case, if we wanted to find the predecessor, we would have just looked at the linked-list and went to the key before the one found using the auxiliary node.

It is worth noting when to use the linked-list to go back and forth depends on whether you want the successor/predecessor and whether the auxiliary pointer was from a right child or left child.

Since we do a binary search on lgU levels and on each level we do constant time work (credit hash tables), our **total time taken is** $O(lglgU)$.

2.4 Space complexity

Since we store hash-tables for the keys on each level, we end up using $O(nlgU)$ **space**.

2.5 Summary

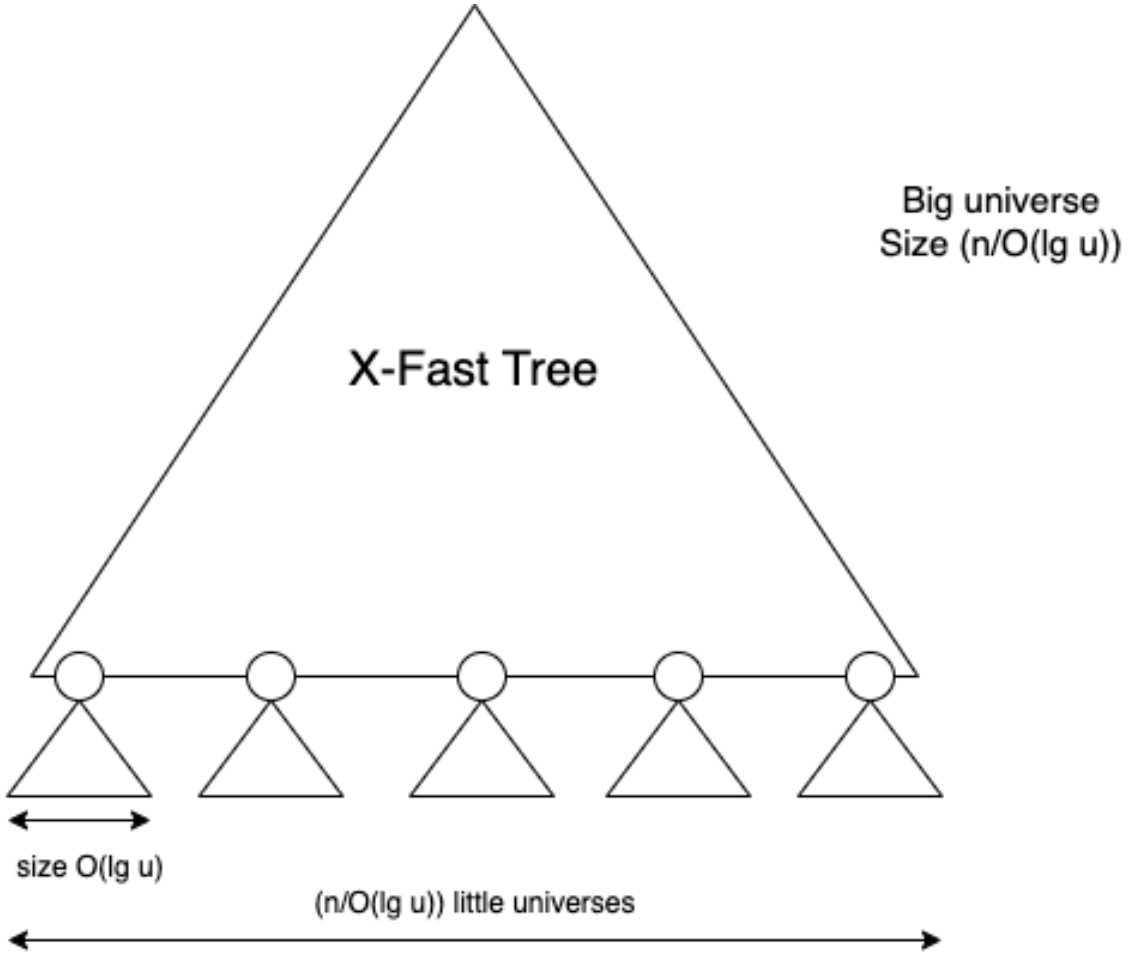
X-Fast trees have helped us reduce space from $O(U)$ to $O(nlgU)$, which is progress towards $O(n)$, but not quite there. We have also lost performance in terms of runtime for insert and delete operations, which have gone from taking $O(lglgU)$ time to $O(lgU)$ time.

We will try and reduce the space and time complexities in the following section.

3 Y-Fast Trees

3.1 Big-universe Little-universe

A Y-Fast tree uses a structure called **big-universe little-universe.** The little-universes are the base cases built on the base of the keys. These could be binary search trees, skip lists, or really any structure that can do operations in $O(lgn)$ time. We will have $O(\frac{n}{lgU})$ of these little universes as the base case, each of size $O(lgU)$ as shown below. Sitting on top of these would be a big-universe, which is an X-Fast tree of size $O(\frac{n}{lgU})$. This will have the bottom-most last as the top-most node of each of the little universe.



3.2 Time Complexity

For operations in the little universes, since each universe is of size $O(lgU)$, any operation will only take time $O(lglgU)$, which is good for us. For the big universe, successor/predecessor already takes $O(lglgU)$ time, and find takes $O(1)$ time. The only problem is when inserting or deleting from the X-Fast tree, in which case the time taken is $O(lgU)$. However, since our little-universes are of

size $O(lgU)$, only one in every $O(lgU)$ operations would "create" (split) a new little universe or "delete" (merge) two little universes. **So, the amortized runtime for these operations also turns out to be $O(lglgU)$.**

3.3 Space Complexity

As talked about above, the space usage for the little universes is $O(lgU)$ space for each of $\frac{n}{O(lgU)}$ universes, which is just $O(n)$. A good way to think about how many little universes we want is to look at how much space an X-Fast tree uses. Since an X-Fast tree would use $O(nlgU)$ space, it would be smart to make the last level $U = \frac{n}{lgU}$. This would mean making $\frac{n}{lgU}$ $O(lgU)$ universes, which is what we do.

3.4 Summary

We were able to use X-Fast trees and other tree structures and combine them in a big-universe little-universe style to make Y-Fast trees. This helped us to reduce the space usage to just $O(n)$, where n is the number of keys, and time complexity to amortized $O(lglgU)$ for the find, insert, delete, successor, and predecessor operations.