CS 6968: Data Str & Alg Scalable Comp Spring 2023

Lecture 2 — Jan. 18, 2023

Prof. Prashant Pandey *Scribe: Alex Crane*

1 Overview

In the last lecture we discussed van Emde Boas Trees, which provide the following operational complexities with a dataset of size n from a universe of size U:

VEB Tree	
Insert/Delete	$O(\log \log U)$
Successor/Predecessor	$O(\log \log U)$
Find	$O(\log \log U)$
Space	$\Theta(U)$

In this lecture our objective is to alter the data structure so that it requires only $\Theta(n)$ space. In section 2 we will complete our discussion of VEB Trees by describing the Insert operation. In section 3 we will review some Prerequisite data structures. In sections 4 and 5 we will discuss X-Fast and Y-Fast Trees.

2 VEB Trees: Insert

In this section we discuss the insertion of elements into a VEB Tree. Consider the following pseudocode:

Algorithm 1: VEB Tree Insert

```
Input: VEB Tree v and item x
if v.min is NONE then
    v.min = x;
    v.max = x;
if x < v.min then
    swap x and v.min;
if x > v.max then
    swap x and v.max;
if v.cluster[high(x)].min is NONE then
    Insert(v.summary, high(x));
Insert(v.cluster[high(x)], low(x));
```

Note that the minimum and maximum values in a cluster should only be stored in the *min* and *max* properties, not in the cluster itself. In this way we guarantee that insertion only requires one non-trivial recursive call. Thus, our recurrence relation is $T(U) = T(\sqrt{U}) + O(1)$, yielding the desired $O(\log \log U)$ runtime.

3 Prerequisite Review

3.1 Linked Lists

Our goal in this lecture is to reduce the space complexity of VEB Trees to $\Theta(n)$ without sacrificing efficiency in the insertion, deletion, successor, and predecessor operations. We therefore must avoid storing empty space, so we will implement our data structure on top of a linked list.

Recall that, given a pointer to the predecessor of x, inserting x into a singly linked list is a constant time operation. Less obviously, deletion of x is also a $O(1)$ operation given a pointer to the successor. This is accomplished by swapping data with the successor node and then deleting it.

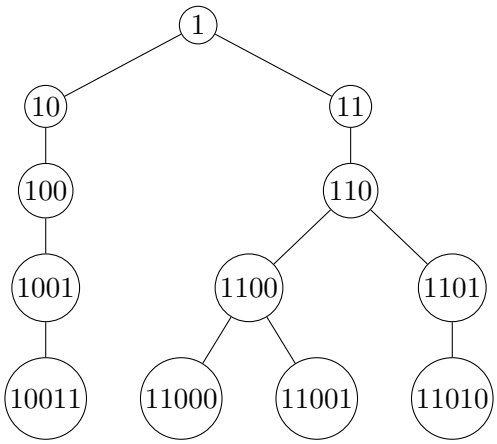
Thus the complexity of insertion and deletion into a singly linked list depends only on the complexity of successor and predecessor operations.

3.2 Tries

A Trie (prefix tree) is a tree which stores words over some alphabet A by letting each node represent a prefix determined by a traversal from the root. Thus, each node has up to $|A|$ children, and during a traversal the choice of child represents a choice of the next character in the prefix.

The space required is $O(|A| \sum_i length(w_i))$. In our context we are considering n bitstrings of equal $(\log U)$ length, so this space complexity becomes $O(n \log U)$. For more detail consider reviewing chapter 8 of the textbook by Peter Brass [1]. Here is an example with $n = 4$, $U = 2^5$. Note that when the words have uniform length, the data are the leaf nodes of the trie.

11001
10011
11000
11010



4 X-Fast Trees

X-Fast trees are an intermediate data structure offering the following performances:

X-Fast Tree	
Insert/Delete	$O(\log n)$ (amortized)
Successor/Predecessor	$O(\log \log U)$
Find	$O(\log \log U)$
Space	$O(n \log U)$

4.1 X-Fast Trees: Structure

In an X-Fast tree, our n elements are stored in a sorted doubly-linked list. Additionally, the nodes of this linked list are exactly the leaf nodes of a trie. Each node in the trie gets pointers to its rightmost and leftmost descendants (which are both nodes in our linked list).

Finally, each of the $\log U + 1$ levels of the trie gets an associated hash table, which stores the answers to existence queries for each possible node at that level.

4.2 X-Fast Trees: Find

The find operation is simple: query the hash table associated with the bottom level of the trie.

4.3 X-Fast Trees: Successor/Predecessor

Recall that we represent each $x \in U$ as a bitstring. Then it is natural to consider the sequence of $\log U$ prefixes of an element x, of lengths $1, 2, ... \log U$. If x is in our structure, then each prefix is associated with a node in the trie, and each of those nodes is at a different level.

Thus, for $x \in U$, we can find the longest existing prefix of x in $O(\log \log U)$ time by performing a binary search on the prefix list, using the set of hash tables. Once we have found this longest existing prefix, we can find the successor or predecessor in constant additional time with minimal logic using the pointers to the leftmost and rightmost descendants.

Assume for the moment that x is not already in our data structure (if it is, just use the linked list). Once we have found the longest existing prefix of x, we know that the associated node in the trie has exacrtly one child. If this is a right child, then the successor is the longest existing prefixs's leftmost descendant, which we access in constant time using the aformentioned pointer. If the child is a left child, then the predecessor is the longest existing prefix's rightmost descendant. Once we have found one, we can find the other by using the pointers in the linked list.

X-Fast Trees: Space

Each existing element requires $\log U$ nodes in the trie, one for each prefix. Then the worst case is where our n elements have few overlapping prefixes. Therefore we need $O(n \log U)$ space (note that the hash tables and pointers are linear in the size of the trie).

5 Y-Fast Trees

X-Fast trees improved our space consumption to $O(n \log U)$, but there is still work to be done to reach the desired $O(\log \log U)$. Additionally, X-Fast Trees have *worse* Insert/Delete performance than VEB Trees. The solution to both of these problems is the Y-Fast Tree:

Y-Fast Tree	
Insert/Delete	$O(\log \log U)$
Successor/Predecessor	$O(\log \log U)$
Find	$O(\log \log U)$
Space	$\Theta(n)$

The key idea of the Y-Fast tree is to divide our large universe U into a number of smaller universes, such that for each relevant operation we can (a) efficiently choose the relevant small universe and (b) efficiently complete the operation in the chosen small universe.

Y-Fast Trees: Structure

The data is divided into $O(\log U)$ balanced BSTs constructed from contiguous items. Each BST submits a representative item to be stored in an X-Fast Tree. Thus the X-Fast Tree contains $O(\frac{n}{\log U})$ elements.

Y-Fast Trees: Operations

The selection of the representatives is important, but was beyond the scope of material covered in our lecture. For more detail, consider reading the original paper [2]. In general, the idea is to use successor and predecessor queries on our X-Fast Tree of data representatives to identify which BST we should complete our operation in.

Since each leaf node of the trie points to a BST of size $O(\log U)$, each of our operations of interest require $O(\log \log U)$ time once we have found the appropriate leaf node. Thus, for any of our operations we need only $O(\log \log U)$ time.

Y-Fast Trees: Space

The associated X-Fast Tree has size $O(\frac{n}{\log U})$, and therefore requires

$$O(\frac{n}{\log U} \log U) = O(n)$$

space.

Additionally, the collection of BSTs only stores a total of n elements, so the entire Y-Fast tree requries $O(n)$ space.

References

[1] Brass, Peter. *Advanced Data Structures*, volume 193. Cambridge University Press, 2008.

[2] Willard, Dan E. Log-logarithmic worst-case range queries are possible in space Θ (N). *Information Processing Letters*, Elsevier. 17 (2): 81-84, 1983.