Chapter 10

y-fast Trees: a randomized alternative to van Emde Boas trees

10.1 Introduction

We have a a universe U of size $u = |U|$ and a set S of size $|S| = n$, $S \subset U$, and we want to implement the following operations:
• insert(x): $S \leftarrow S \ \cup \ \{x\}$
• delete(x): $S \leftarrow S - \{x\}$
• predecessor: pred(x), $x \in U$: return $\max_{k \in S}$ s.t. $k \leq x$
• successor: succ(x), $x \in U$: return $\min_{k \in S}$ s.t. $k \geq x$.
All these operations are to take *expected* and *amortized* time $O(\log \log u)$ (and probably *w.h.p.* as well). The space cost is $O(n) = O(|S|)$.

First, we do the X-fast tree. Then the Y-fast tree.

10.2 The X-Fast Tree

The X-fast tree is another implementation of the successor-predecessor operations. Similarly to the van Emde Boas tree, the X-fast tree recursively splits the search word into two pieces, at each stage picking the half that is required to continue the search. The X-fast tree, however, uses this process to optimize the traversal of a trie.

The X-fast tree is not as fast as the van Emde Boas tree, because it takes $O(\log u)$ time

to do maintenance operations. However, the X-fast tree is a significant part of another structure, the Y-fast tree, and so it is simpler to cover it first.

Claim 71 (Time and space bounds of the X-fast tree)**.** *The X-fast tree answers* ***search****,* ***predecessor*** *and* ***successor*** *queries in* $O(\log \log u)$ *worst-case time. It performs the maintenance operations* ***insert*** *and* ***delete*** *in amortized* $O(\log n)$ *time. The structure takes* $O(n \log u)$ *space.*

10.3 The Structure of an X-Fast Tree

There are three parts to an X-fast tree - the trie, the linked list of stored elements, and a set of hash tables, one per level of the trie.

10.3.1 The Trie

A trie is, in effect, a tree in which every node is labeled according to the path which goes from the root to that node. In the case of the X-fast tree, our trie will be a binary tree in which moving to the left child of a node is represented by a zero, while moving to the right child of a node is represented by a one, so that, in a complete binary tree, each of the nodes at level k would be labeled with one of the values $0 \ldots 2^k - 1$, in order, from left to right. The trie is also augmented with extra pointers which point to the leftmost or rightmost child of that node, as described below.

Definition 72 (Trie Structure in an X-Fast Tree)**.** *Let* S *be the set of elements stored in X-fast tree* T*, and let* $l = \log u$ *be the number of bits of the elements which can be stored in* T*. For all* $Y \in S$*, the trie of* T *contains a unique leaf node at level* l *that corresponds to* Y*. The trie contains no other leaves.*

Definition 73 (Auxilliary Pointers in the Trie)**.** *Let* N *be some internal node of the trie in an X-fast tree, such that* N *has exactly one child. Then,* N *contains an auxilliary pointer to a leaf of the trie. If* N*'s child is a left child, then* N*'s auxilliary pointer points to* N*'s rightmost descendant. If* N*'s child is a right child,* N*'s auxilliary pointer points to* N*'s leftmost descendant.*

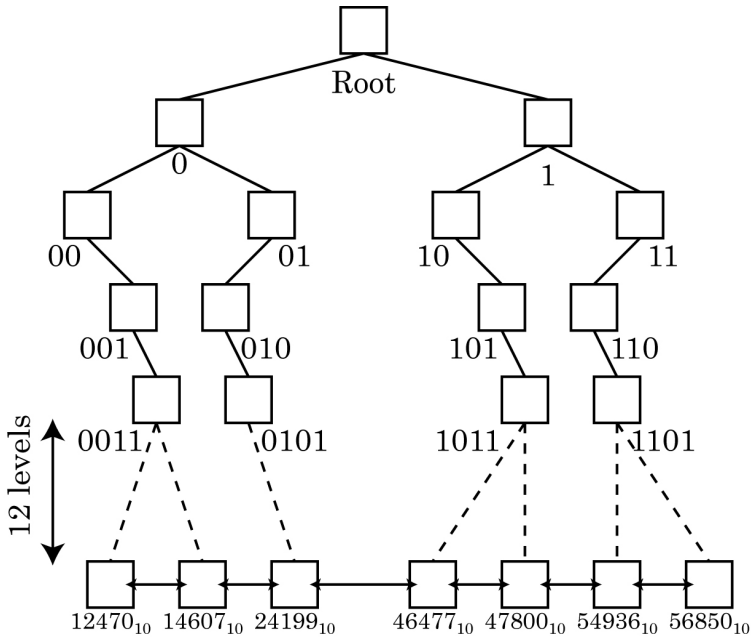

Figure 10.1: A 16-bit X-fast tree containing seven keys.

10.3.2 The Linked List

Each of the leaves of the trie is part of a linked list connecting every single leaf, in order. As we maintain the X-fast tree, we also maintain this linked list.

10.3.3 The Hash Tables

Let's assume our universe is of size $u = 2^l$. There are then $l + 1$ levels to the trie, and there are also $l + 1$ associated hash tables. Each level of nodes in the trie is also entered into the hash table of the same level. Therefore, if we want to check whether a particular node exists in the tree, we don't need to search the tree but instead can check the hash table.

10.4 Definitions

Definition 74 (The Search Path of an Element of S)**.** *Let T be the trie of an X-fast tree, such that it has $l+1$ levels. For any $Y \in U$, there is, in T, some ordered list of nodes which would be followed if we were doing a tree search for element Y in the trie. This ordered list of nodes is the* ***search path*** *of Y.*

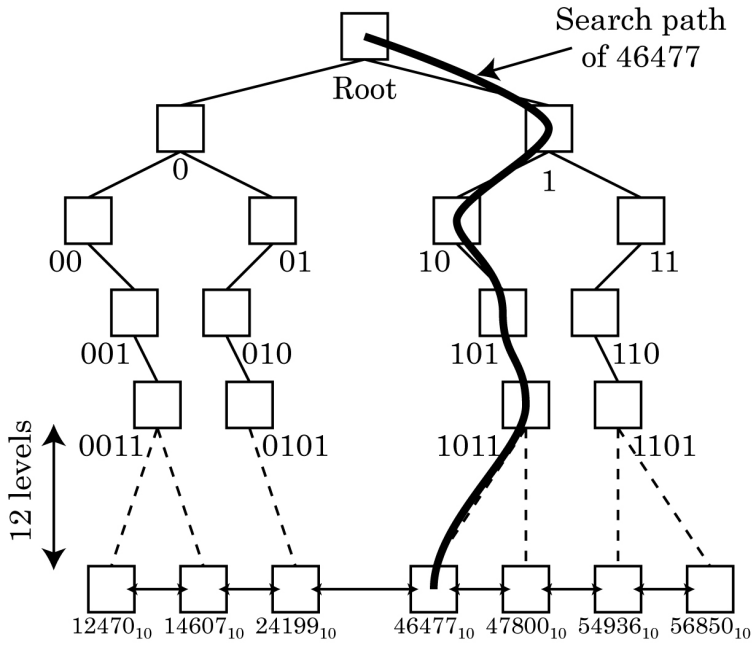

Figure 10.2: The search path of 46477 in the X-fast tree of figure 10.1.

Note that each node in the search path of Y is uniquely identified by a prefix of the binary string of Y; each node is identified by a prefix of a different length.

Definition 75 (The Full Search Path of an Element of U)**.** *Let T be the trie that is part of an X-fast tree which contains every element in U (therefore, T is full). The search path of some element Y in this trie is the* ***full search path*** *of that element.*

10.5 Searching an X-Fast Tree

The structures in an X-fast tree give us a powerful way to search the tree for the information we need. Obviously, if we want to check for the presence of an element, we can just check the lowest (and largest) hash table, but successor and predecessor can be done very quickly as well.

Let's refer to query element Y, and further assume that Y does not exist in the tree - if it does, we can directly check for it and take appropriate action. Then the search path of Y has some lowest node, which we will call N. N must have exactly one child, since it's on the end of the search path. By the properties of tries, if that child is a left child, then N's rightmost descendant must be the predecessor of Y. If the child is a right child, then N's

Figure 10.3: Pseudocode of **successor** for the X-fast tree

```
successor(X, Tree) {
        N = lowest node on search path to X in Tree
        if (N.LeftChild exists) {
                SuccessorNode = N.AuxilliaryPointer
        } else { /* The right child exists and the left child does not */
                PredecessorNode = N.AuxilliaryPointer
                SuccessorNode = PredecessorNode.next
        }
        return SuccessorNode.value
}
```

leftmost descendant must be the successor of Y. Since the leaves form a linked list, we can immediately find the successor from the predecessor, or vice versa.

We rather conveniently stored the exact pointer that we must need at an internal node in the auxilliary pointer described above. Therefore, the above operations take constant time, and we only need to find the lowest node on the search path of Y.

Observation 76. *Finding the lowest node on the search path of Y in X-fast tree T allows us to find both **predecessor**(Y) and **successor**(Y) in constant time.*

To find that lowest node, we will do a binary search on the full search path of Y. We can do this, because we know how to find every one of those nodes, since they are all derivable from Y. What's more, we can determine if they exist in constant time using the hash tables. There are $O(\log u)$ nodes in the full search path, so it takes $O(\log \log u)$ time to find the lowest one that is present in the trie.

See figure 10.4 for an example. We are searching the X-fast tree of figure 10.1 for the key $47955_{10} = 1011101101010011_2$. We first check to see if there is a node at level 8 for 10111011. There is none, so we can eliminate the lower levels of the tree from consideration. We then look at level 4 for 1011 (exists), level 6 for 101110 (exists), and level 7 for 1011101 (exists). We can conclude that the lowest node on 47955's search path in the tree is at level 7.

Theorem 77 (Time to Search an X-Fast Tree)**.** *It takes $O(\log \log u)$ time to search for the predecessor or successor of a query element in an X-fast tree. It takes $O(1)$ time to check*

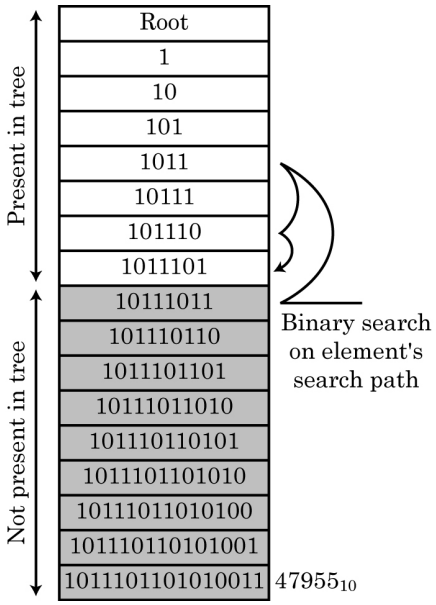

Figure 10.4: A binary search on the search path to 47955 in the X-fast tree of figure 10.1.

if a query element is in an X-fast tree.

10.6 Maintaining an X-Fast Tree

Maintaining an X-fast tree does not involve any special insights. It is necessary, on every update, to walk up the tree for $O(\log u)$ levels, updating the trie, the auxilliary pointers, and the hash tables, in order to insert or delete a element.

10.7 Space Consumption of the X-fast Tree

The X-fast tree can be quite wasteful in space, because it requires $\log u$ nodes for every element in the tree, and most of these nodes are not shared between multiple elements unless most of the elements are in a small cluster of the universe. Since the sizes of the hash tables and the other components of the X-fast tree are linear in the size of the trie, we can conclude that the X-fast tree takes $O(n \log u)$ space.

We can conclude the following about the X-fast tree:

Theorem 78 (Time and space bounds of the X-fast tree)**.** *The X-fast tree answers* ***search****,*

predecessor *and* ***successor*** *queries in* $O(\log \log u)$ *worst-case time. It performs the maintenance operations* ***insert*** *and* ***delete*** *in amortized* $O(\log n)$ *time. The structure takes* $O(n \log u)$ *space.*

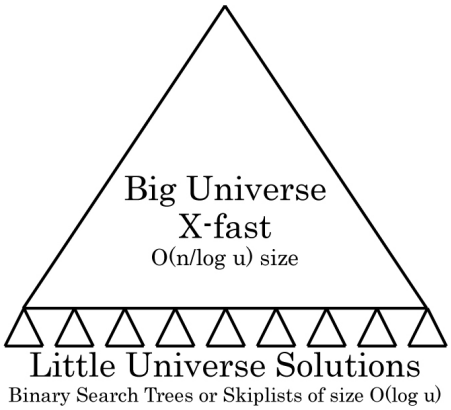

Figure 10.5: The structure of the Y-fast tree.

10.8 The Y-Fast Tree

From either a theoretical or a practical perspective, the X-fast tree isn't really satisfactory. The maintenance time is fairly high, and since u is usually much greater than n, the $O(\log u)$ time becomes much more of a burden than the $O(\log n)$ time of more mundane structures. The Y-fast tree fixes these problems by adapting the X-fast tree to amortize the maintenance across more operations.

Claim 79 (Time and space bounds of the Y-fast tree)**.** *The Y-fast tree answers* ***search****,* ***predecessor*** *and* ***successor*** *queries in* $O(\log \log u)$ *worst-case time. It performs the maintenance operations* ***insert*** *and* ***delete*** *in amortized* $O(\log \log n)$ *time. The structure takes* $O(n)$ *space.*

10.9 Big Universe versus Little Universe

We can solve the maintenance problem of the X-fast tree by dividing our problem into two unequal pieces. The larger problem will be solved by the X-fast tree, but will be of size $O(\frac{n}{\log u})$. There will be $O(\frac{n}{\log u})$ smaller problems, each of which is of size $O(\log u)$ and which will be solved by a simpler solution. This is known in general as the *Big Universe-Little Universe* approach.

10.9.1 The Little Universe

We get the smaller problems by dividing the set S which is stored in the Y-fast tree into $O(\frac{n}{\log u})$ pieces. Each piece is a contiguous section of the set - in effect, if we took a sorted list of elements of S, each piece would consist of one group of $O(\log u)$ contiguous elements. In an actual implementation, each piece would be maintained between two sizes - say, between $\frac{\log u}{4}$ and $4 \log u$.

The little universe problems are solved with conventional solutions. The requirements are simply that the solution take $O(\log n)$ time for maintenance and queries, and $O(n)$ time to split and merge two of the little-universe problems. Since $n = \log u$, doing work in a little-universe structure takes $O(\log \log u)$ time. As long as we only need to look at a constant number of little-universe structures, we will not exceed the $O(\log \log u)$ barrier.

10.9.2 The Big Universe

The big universe problem is solved with the X-fast tree. Each element stored in the X-fast tree includes a pointer to a corresponding little-universe structure. It's easy to arrange the elements such that querying on the X-fast tree leads to $O(1)$ little-universe structures that may contain the answer (although this condition is trickier to maintain during deletion).

During the course of maintenance, if a little-universe structure gets too large as defined above, it is split into $O(1)$ smaller structures. Each of these then is inserted into the X-fast tree in place of the original. Therefore there are $O(1)$ maintenance operations associated with the split, making each split cost $O(\log u)$. The cost of physically splitting the little-universe structure in half, as noted above, is also $O(\log u)$.

However, since the little-universe structures are of size $O(\log u)$, it means they will only need to be split every $O(\log u)$ steps. Therefore the amortized cost of splits is $O(1)$. A similar argument applies to deletion.

Observation 80 (Maintenance time of little-universe structures)**.** *It takes $O(1)$ amortized time to maintain the little-universe structures at $O(\log u)$ size during Y-fast maintenance.*

As before, the query time is $O(\log \log u)$, which is acceptable.

10.10 Space Usage

The space usage of the little-universe structures, in general, will be $O(n)$ (with respect to the size of their contents). Clearly the total of all of them will be $O(n)$ as well (with respect to the size of the Y-fast tree's contents). As discussed above, the space usage of the X-fast tree is $O(n \log u)$, but in the case of the X-fast tree $n_{Xfast} = O(\frac{n_{Yfast}}{\log u})$, so the actual space usage is $O(\frac{n \log u}{\log u}) = O(n)$.

So we finally have the following result, summarizing the time and space bounds of the Y-fast tree.

Theorem 81 (Time and space bounds of the Y-fast tree)**.** *The Y-fast tree answers* ***search****,* ***predecessor*** *and* ***successor*** *queries in $O(\log \log u)$ worst-case time. It performs the maintenance operations* ***insert*** *and* ***delete*** *in amortized $O(\log \log n)$ time. The structure takes $O(n)$ space.*