Chapter 9

van Emde Boas Trees

9.1 Introduction

The successor operations can be performed by any balanced binary tree in $O(\log n)$ time and linear space. Under the comparison model, on a pointer machine, we cannot do better, because we know that $\Omega(n \log n)$ is required for sorting, and solving the successor set problem in less-than-logarithmic time would also allow us to sort faster.

It is fortunate that we are not limited to the comparison model! By taking advantage of simple operations like division and modulus, we can do far better.

Given a universe $U = \{0, 1, 2, ..., u-1\}$, maintain a subset $S \subseteq U (|S| = n, |U| = u)$, subject to the following operations :

- **insert(x)** — Add $x \in U$ to S, i.e., $S \leftarrow S \cup \{x\}$.
- **delete(x)** — Remove x from S, i.e., $S \leftarrow S - \{x\}$.
- **succ(x)** — Return the minimum $y \in S$, such that $y \geq x$. (Note that $x \in U$ but x need not be in S.)
- **pred(x)** — Return the maximum $y \in S$, such that $y \leq x$.
- **find(x)** — Return whether $x \in S$.

Example: Consider a set S = {4,19,30} from the universe U = {0,1,...,100},
Then **pred**(10) = 4, **succ**(19) = 30 and **succ**(30) = ∞.

9.2 Complexity :

All operations require time : $O(\log \log \boldsymbol{u})$.
For now, space = $\theta(u)$. Later use randomization to get linear space($\theta(n)$).
Intuition : $\log \log \boldsymbol{u}$ comes from $\mathrm{T}(u) = \mathrm{T}(\sqrt{u}) + 1$

Example: Suppose that U contains the integers represented by 64-bit words. Then $\log \log 2^{64} = 6$.

9.3 Array-Based Naive Solutions

Consider the following two ways of storing elements in an array.

Choice 1:

$$\chi[i] = \begin{cases} 1 & \text{if } i \in S. \\ 0 & \text{otherwise} \end{cases}$$

Insertion: O(1), Queries: O($\boldsymbol{u}$)

Example:

0	1	2	3	4	5	6	7	8			u-1
1	0	1	0	0	0	1	1	0	......	......	0

succ(1) = 2 .

Choice 2:

$$\Psi[i] = \begin{cases} j & \text{if j=}\mathbf{succ}\text{(i).} \\ \Lambda & \text{if no succ, ie largest element} \end{cases}$$

Insertion: O($\boldsymbol{u}$), Queries: O(1)

Example:

0	1	2	3	4	5	6	7	8			u-1
0	2	2	6	6	6	6	7	Λ	......	......	

succ$(2) = 6$.

Choice 1 wins over choice 2.

9.4 A Recursive Approach

The van Emde Boas tree (named after its inventor) works by repeatedly splitting the set into smaller pieces, each of whose size is the square root of its parent's size. Recall that if $T(u) = T(\sqrt{u}) + O(1)$ then $T(u) = O(\log \log u)$.

We start with an unpleasant assumption - that we have an unlimited amount of space available for our use. The raw deterministic van Emde Boas tree consumes an astounding $O(u)$ space! Later on, we will improve this to $O(n)$, giving us a much more practical algorithm.

Most of the time, odd universe sizes are not really required, and in any case they make little difference to the performance of the algorithm. On the other hand, if we restrict the size of the universe to a convenient subset of the powers of two, we can conveniently replace square root, division, and modulus operations with simple bitshifts and masks.

Assumption 64. *The universe size $u = 2^{2^k}$ for some integer k.*

Before we begin, it will be helpful to state what our final results are.

Claim 65 (Time and space bounds of the randomized van Emde Boas tree)**.** *The randomized version of the van Emde Boas 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 $O(\log \log u)$ time. The structure takes $O(n)$ space.*

9.5 Basic Construction

Let's assume (for the sake of induction) that we have a solution widget $W_{\sqrt{u}}$ for the successor set problem of size $\sqrt{u}$. We will refer to $W_{\sqrt{u}}$ as a subwidget. We use the subwidgets $W_{\sqrt{u}}$ to construct W_u, the solution for the problem of size u. The idea is to break up the universe U into $\sqrt{u}$ contiguous subuniverses, each of which is of size $\sqrt{u}$. Each subuniverse is solved with $W_{\sqrt{u}}$. Then we use these solutions to solve W_u.

Since we have $\sqrt{u}$ subwidgets, we need to be able to determine which of these subwidgets an element is contained in (assuming the element is contained in the set). By convention, X_H is the widget that contains X, while X_L is the element that is stored in widget X_H.

Mathematically Strict	Restricted u	
$X_L = X \mod \sqrt{u}$	$X_L = X - (X_H << 2^{k-1})$	High-order bits of X
$X_H = \frac{X - X_L}{\sqrt{u}}$	$X_H = X >> 2^{k-1}$	Low-order bits of X
	($<<$ and $>>$ are bitshifts)	

As an example, if $u = 2^{32}$:

$$X = 2221820653_{10} = \boxed{1000010001101110}\boxed{0100101011101101}_2$$

$$X_L = 0100101011101101_2 = 19181_{10}$$

$$X_H = 1000010001101110_2 = 33902_{10}$$

We store the element X_L in widget X_H, but how do we store widget X_H? For now, we store it in an array. We use the convention $A[N]$ to refer to the Nth widget, and $A[X_H]$ to refer to widget X_H.

Invariant 9.5.1 (Storage of elements in subwidgets)**.** *Let there be a van Emde Boas Tree T which stores set $S \subset U$, and let X_L and X_H be defined as above with respect to X. The van Emde Boas tree contains an array A of subwidgets. The array is of size $\sqrt{u}$, and each subwidget in A has a universe size of $\sqrt{u}$. For each $X \in U$, if and only if $X \in S$, then X_L is stored in subwidget $A[X_H]$.*

In order to do **search**, we calculate X_H and X_L, find the widget $A[X_H]$, and recursively do the query **search**$(A[X_H], X_L)$.

For successor, we calculate X_H and X_L and find the widget $A[X_H]$. We then query $A[X_H]$ for the successor to X_L. The recursive call may return an indication that there is no successor; if that's the case, then we look for the next widget that contains any element at all, and return the minimum element from that widget. And finding the next nonempty widget is itself a successor search on the set of nonempty widgets!

To accomodate this requirement, we will add one more widget $W_{\sqrt{u}}$. We will call this the **index widget** or W_{index}. For convenience, when required we will refer to S_{index}, the set stored in W_{index}.

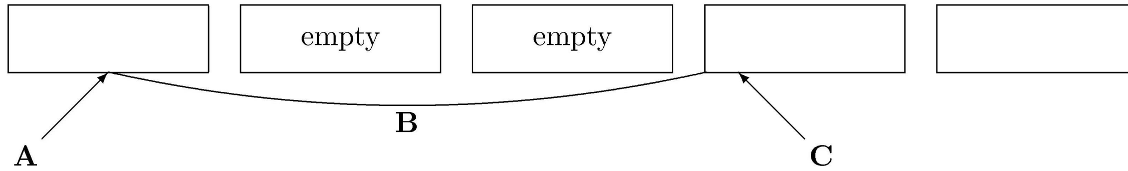

Figure 9.1: The search for the successor among subwidgets. (A) A successor query looks at the original widget that would contain the element. (B) The element is not found, so we search for the next nonempty widget. (C) The successor must be the first element of the widget found in (B).

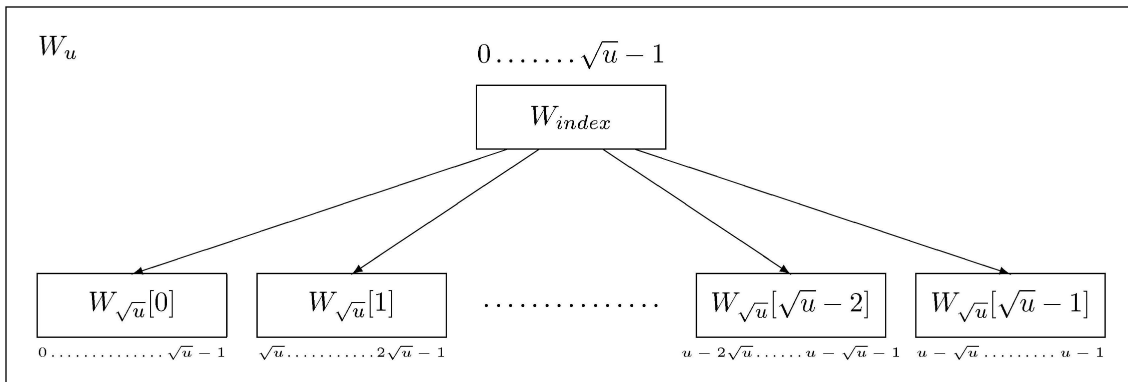

Figure 9.2: The incomplete van Emde Boas tree structure.

Invariant 9.5.2 (Index widget corresponds to set of non-empty widgets)**.** *Let there be a van Emde Boas tree T which stores set $S \subset U$. For each subwidget $A[i]$ that is part of T, if and only if **empty**($A[i]$) $= false$, then $i \in S_{index}$.*

How do we do insertion in the structure? We insert X_L into widget $A[X_H]$, and also update the index, inserting X_H into W_{index}.

Now we have a problem. What recurrence relations apply to **successor** and **insert** now?

successor	$T(u) = 3T(\sqrt{u}) + O(1)$
insert	$T(u) = 2T(\sqrt{u}) + O(1)$

Without going into details, these both solve to $T(u) = \Omega(\log^p u)$ for some $p \geq 1$. Even if we weren't looking for sub-logarithmic bounds, $O(\log u)$ would be a terrible running time. It is clear that we can only make one full recursive call.

Figure 9.3: Pseudocode for the first version of the van Emde Boas tree. Neither **successor** nor **insert** is $O(\log \log u)$, because both potentially require two recursive calls.

```
successor(X, Tree) {
    X_H = most significant half of X
    X_L = least significant half of X
    TrialAnswer = successor(X_L, Tree.WidgetArray[X_H])
    if (TrialAnswer is valid) return TrialAnswer
    TrialWidget = successor(X_H, Tree.WidgetIndex)
    if (TrialWidget is invalid) return invalid
    TrialAnswer = successor(−∞, Tree.WidgetArray[TrialWidget])
    return TrialAnswer
}

insert(X, Tree) {
    X_H = most significant half of X
    X_L = least significant half of X
    insert(X_L, Tree.WidgetArray[X_H])
    insert(X_H, Tree.WidgetIndex)
}
```

Observation 66. *In order to achieve time bounds of* $O(\log \log u)$*, we can only make one recursive call that does significant work. Everything else must be* $O(1)$*.*

9.6 Insertion and Successor Queries - Second Attempt

Let's start by examining successor, where we have three separate recursive calls to make. We have two cases that successor deals with. Either $A[X_H]$ contains the successor to X_L, or it does not. If $A[X_H]$ contains the desired successor, then we only need to make that one recursive call.

What exactly happens when $A[X_H]$ does not contain X_L's successor? First, this means that the largest element in $A[X_H]$ is smaller than X_L. To verify this point, we don't actually *need* to make an entire call to successor - if we had a quick way of checking the maximum element, we could do it in constant time, and then move on to a call to successor on *either*

$A[X_H]$ *or* W_{index}, and not both.

We should also notice that, whatever widget is found by the call to **successor**(X_H, W_{index}), we don't need to make a full call to successor there either! All we really need is the minimum element from that widget - which is the only element that could be the successor we are looking for.

Observation 67. *If retrieving the minimum and maximum elements in any widget takes constant time, then* ***successor*** *only requires one actual recursive call.*

What about insertion? This, too, is wasteful. The only time we need to make two recursive calls is when the *first* element is inserted into $A[X_H]$. Otherwise, we don't actually need to insert X_H into W_{index}, because it's already there. It's easy enough to maintain a count of the elements in a widget, so that a call to something like empty will take constant time, and we can eliminate the second recursive call in all but one case - when X_H is not yet in W_{index}.

Clearly it needs to be inserted at *some* point. So we need to reduce the cost of the first insertion into $A[X_H]$ to $O(1)$ - and incidentally, we also need to maintain the minimum and maximum elements that we rely on for insertion. The way to do this is to store the minimum and maximum elements without ever inserting them into a subwidget. This way, retrieving the minimum and maximum elements takes constant time, and inserting the first element into a widget also takes constant time. This allows us to remove the extra recursive call.

Observation 68. *If the minimum and maximum values in a van Emde Boas tree are maintained separately from the subwidgets, without being inserted into the subwidgets, then* ***insert*** *only requires one recursive call.*

We have completed the van Emde Boas tree. We have eliminated the extra recursive calls, and therefore these versions of **insert** and **successor** take $O(\log \log u)$ time. At this point, the other required operations can be built along similar lines. Predecessor is an exact mirror of successor. Deletion is somewhat tricky, but still takes $O(\log \log u)$ time.

Of course, the structure still takes $O(u)$ space.

9.7 Space Considerations

The version of the van Emde Boas tree described above is enormously wasteful in space.

Figure 9.4: Pseudocode of second and final version of van Emde Boas tree. Even though **successor** and **insert** both may make multiple recursive calls, only one of the recursive calls does nontrivial work in each, so both are $O(\log \log u)$.

```
successor(X, Tree) {
        if (X > maximum) return invalid
        if (X ≤ minimum) return minimum
        X_H = most significant half of X
        X_L = least significant half of X
        TrialAnswer = successor(X_L, Tree.WidgetArray[X_H])
        if (TrialAnswer is valid) return TrialAnswer
        TrialWidget = successor(X_H, Tree.WidgetIndex)
        if (TrialWidget is invalid) return invalid
        TrialAnswer = successor(−∞, Tree.WidgetArray[TrialWidget])
        return TrialAnswer
}

insert(X, Tree) {
        if (maximum = minimum) {
                if (X > maximum) maximum = X
                else if (X < minimum) minimum = X
                return
        }
        if (X > maximum) swap(X, maximum)
        if (X < minimum) swap(X, minimum)
        X_H = most significant half of X
        X_L = least significant half of X
        if (Tree.WidgetArray[X_H].empty()) {
                insert(X_H, Tree.WidgetIndex)
        }
        insert(X_L, Tree.WidgetArray[X_H])
}
```

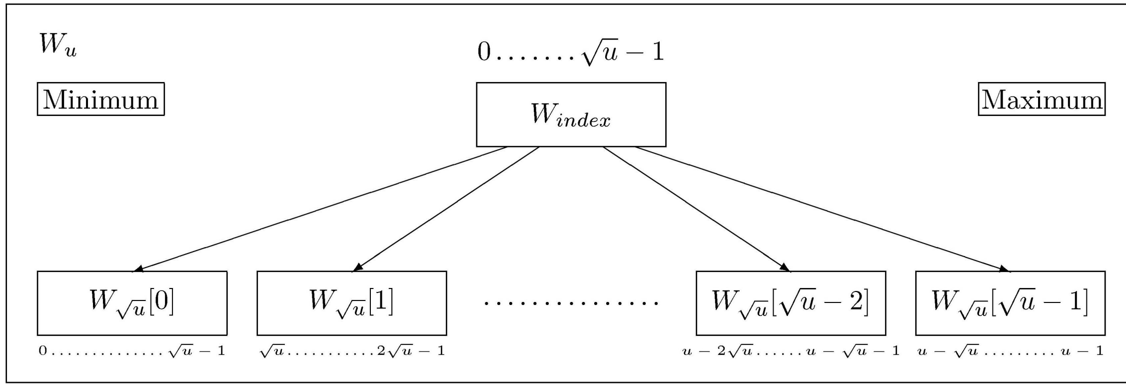

Figure 9.5: The completed van Emde Boas tree.

It is assumed above that every piece of the structure has been pre-instantiated. Thus, the array of widgets that we loosely defined before becomes enormous. In fact, we can immediately prove that the structure takes $\Omega(u)$ space by noting that it has enough widgets to store every possible element from the start. Even if we don't instantiate every possible widget, the array for the topmost widget still takes $\Theta(\sqrt{u})$ space.

Theorem 69 (Space bounds of the deterministic van Emde Boas tree)**.** *The deterministic version of the van Emde Boas tree with universe size u consumes $\Omega(u)$ memory.*

By use of randomization (ie, hashing), and by only instantiating widgets that actually contain an element, we can reduce the space requirements to $O(n)$. What's more, by using the perfect hashing algorithm of Fredman, Komlós, and Szemerédi, as modified by Dietzfelbinger, we can have a hash algorithm with amortized constant-time insertion and deletion with worst-case constant time queries. This allows us to improve the space usage to $O(n)$ without compromising the worst-case query time.

We summarize our complete results in the following theorem.

Theorem 70 (Time and space bounds of the randomized van Emde Boas tree)**.** *The randomized version of the van Emde Boas 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 $O(\log \log u)$ time. The structure takes $O(n)$ space.*

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 $0 \dots 2^k - 1$, 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.*

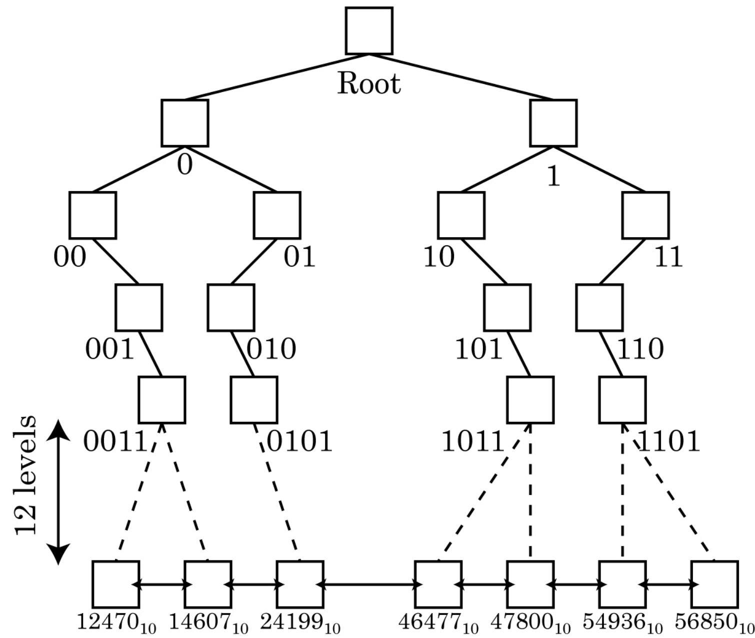

Figure 10.1: A 16-bit X-fast tree containing seven keys.

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.*

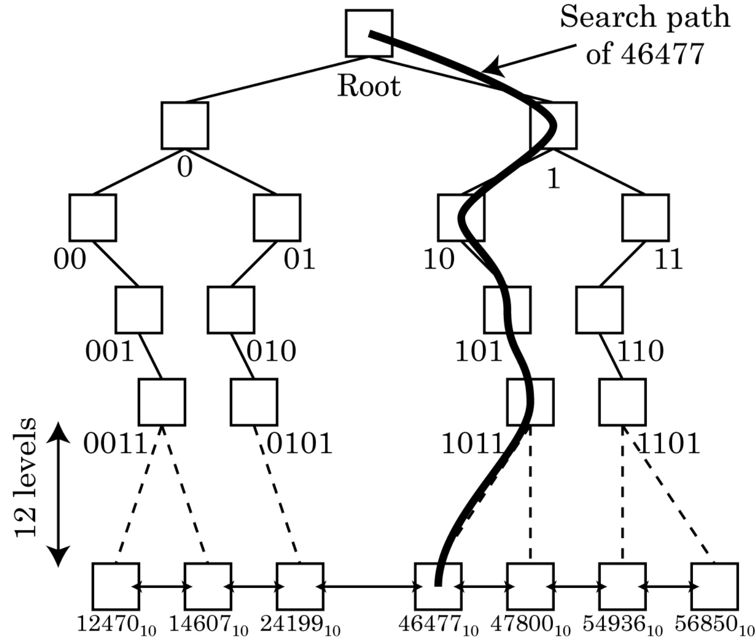

Figure 10.2: The search path of 46477 in the X-fast tree of figure 10.1.

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*

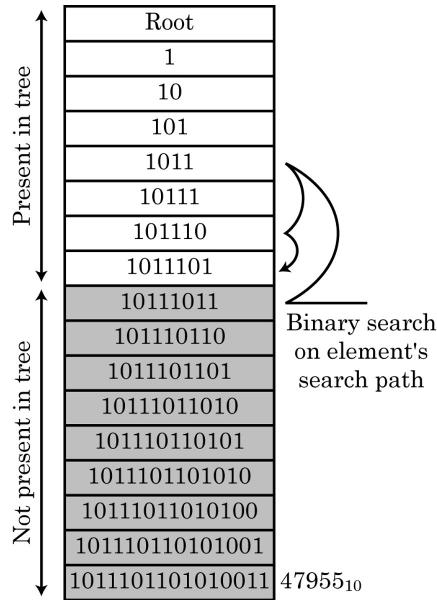

Figure 10.4: A binary search on the search path to 47955 in the X-fast tree of figure 10.1.

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.*