**CS 7280: Data Str Alg Scalable Comp** Spring 2025

# Lecture 1 — January 8, 2025

*Prof. Prashant Pandey* *Scribe: Diandre Sabale*

## 1 Overview

In the last lecture, we discussed the class structure and policies.

In this lecture, we began designing and discussing Van Emde Boas Trees, a way to efficiently store and maintain elements given a limited universe of values. The design and analysis of these data structures discussed in lecture originates from a paper by P.van Emde Boas [1].

## 2 Motivation

Recall that with traditional techniques, performance and size issues may arise when dealing with extremely large amounts of data.

Our goal is to maintain $n$ elements among $\{0, 1, ..., U - 1\}$ subject to the following operations:

- $Insert(x)$: Add a new element $x$ to maintain
- $Delete(x)$: Remove a specific element $x$ from the set
- $Successor(x)$: Find the next largest element in the set compared to $x$ (but note that $x$ does not have to be in the set of maintained elements).

The aim of this implementation is to optimize for the setting where we have a limited universe of values. Our goal is to have $O(\text{lglg}U)$ runtime for all operations while taking $O(n)$ space.

These data structures have applications in settings such as network routers, graphs, and system memory allocation.

## 3 Recurrences

We can consider how to arrive at a runtime of $O(\text{lglg}U)$ by reviewing the runtimes associated with different recurrences.

For example, consider the binary search recurrence: $T(k) = T(\frac{k}{2}) + O(1)$. This corresponds to a runtime of $O(\text{lg}k)$. Note that with an alternative input to the recurrence, we can achieve $T(\text{lg}k) = T(\frac{\text{lg}k}{2}) + O(1)$ and a runtime of $O(\text{lglg}k)$.

However, we choose to consider an input size of $U$. To achieve the desired runtime, the recurrence we must use is instead $T(U) = T(\sqrt{U}) + O(1)$, which corresponds to the desired runtime of $O(\text{lglg}U)$. In a binary tree with height lg$U$ and $U$ leaf nodes, lglg$U$ is achieved by traversing the levels in the tree.

## 4 Version 1: Bit Array of Size $U$

We start with a naive approach. Consider the following array of size $U$:

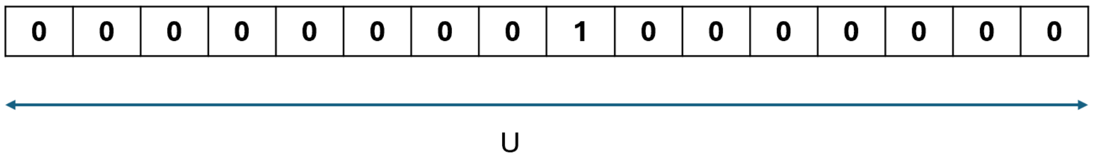


Figure 1: 1 = present element, 0 = absent element

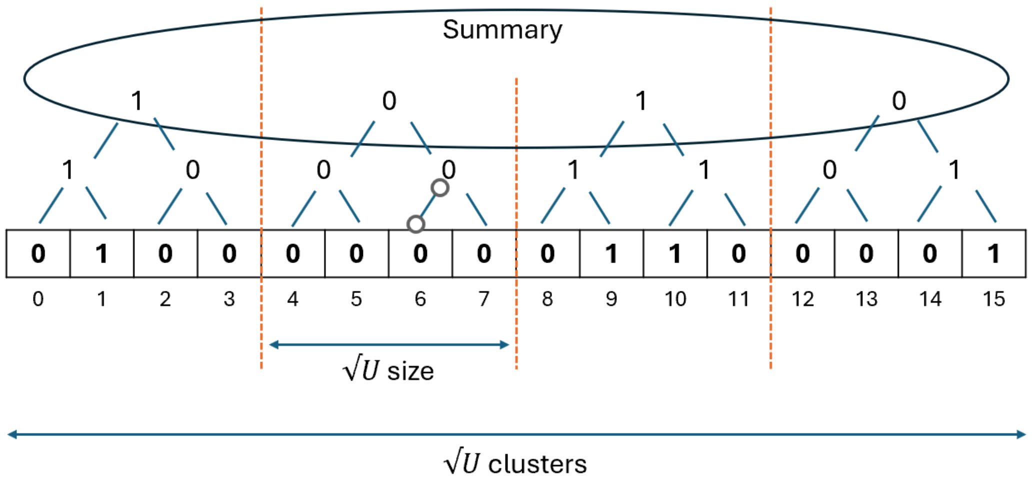
To *Insert* or *Delete*, we simply switch the associated index's value to 1 or 0. So, these operations run in $O(1)$ time.

To find *Successor*, one can perform a simple linear search over the array, corresponding to a run-time of $O(U)$.

This solution also uses $O(U)$ space.

## 5 Version 2: Split Into $\sqrt{U}$ Clusters of Size $\sqrt{U}$

We can split the data structure from version 1 into $\sqrt{U}$ clusters of size $\sqrt{U}$, and additionally store a summary structure describing which clusters have elements:



Note that the intermediate tree structure is not actually stored, it merely demonstrates the comparison process used to generate the final summary values.

The *Insert* and *Delete* operations will remain the same. However, *Successor* changes - we now have to look in $x$'s cluster, look for the next 1 bit in the summary structure if there are no other elements in $X$'s cluster, then look for the first 1 bit in the newly found cluster. This will take $O(\sqrt{U})$ time, since each cluster and the summary structure contains $\sqrt{U}$ elements.

## 6 Version 3: Index

We can slightly improve performance by considering the index of values within the structure if we choose to continuously split the clusters.

We can define $Index(i, j) = i\sqrt{U} + j$. In other words, for any $x$, we can find $x = i\sqrt{U} + j$, where $0 \leq j \leq \sqrt{U}$ and $0 \leq i \leq \sqrt{U}$. For example, if $x = 9$ and $\sqrt{U} = 4$ like in the example from version 2, then $i = 2$ and $j = 1$.

Alternatively, we can define the following functions as well:

- $High(x) = \lfloor x/\sqrt{U} \rfloor$
- $Low(x) = x \bmod \sqrt{U}$

Now we can redefine the *Insert* (and similarly *Delete*) and the *Successor* functions:

We know that $V = U$. We also know that $V.clusters[i]$ will have a size of $\sqrt{U}$. Plus, $V.summary$ has a size of $\sqrt{U}$ as well.

For $Insert(V, x)$, we simply first insert $Low(x)$ into $V.cluster[High(x)]$, then we insert $High(x)$ into $V.summary$. Since this takes two recursive *Insert* calls, the *Insert* operation has a recurrence of $T(U) = 2T(\sqrt{U}) + O(1)$, corresponding to a runtime of $O(\lg U)$.

Meanwhile, we now define *Successor* as follows:

**Algorithm 1** Version 3 Successor

```
Procedure SUCCESSOR($V$, $x$)
$i = High(x)$
$j = Successor(V.cluster[i], Low(x))$
if ($j = \infty$) then
  $i = Successor(V.summary, i)$
  $j = Successor(V.cluster[i], -\infty)$
end if
Return $Index(i, j)$
End Procedure
```

Since there are three recursive calls to *Successor*, the updated *Successor* function has a recurrence of $T(U) = 3T(\sqrt{U}) + O(1)$, which still does not accomplish the desired $O(\lg\lg U)$ time.

## 7 Version 4: Storing Min and Max

We can adapt version 3 by additionally storing the minimum and maximum values for every $V$. So, the *Successor* operation now becomes:

**Algorithm 2** Version 4 Successor

```
**Procedure** SUCCESSOR($V$, $x$)
**if** $x < V.min$ **then**
    return $V.min$
**end if**
$i = High(x)$
**if** $Low(x) < v.cluster[i].max$ **then**
    $j = Successor(V.cluster[i], Low(x)$
**else**
    $i = Successor(V.summary, i)$
    $j = V.cluster[i].min$
**end if**
return $Index(i, j)$
**End Procedure**
```

Here, since there is only one recursive call, the recurrence becomes $T(U) = T(\sqrt{U}) + O(1)$, meaning that the *Successor* function finally achieves the desired runtime of $O(\text{lglg}U)$.

However, the *Insert* (and *Delete*) function must be adjusted to include updates to the stored minimum and maximum values. For example, the *Insert* function now becomes:

**Algorithm 3** Version 4 Insert

```
**Procedure** INSERT($V$, $x$)
**if** $x < V.min$ **then**
    $v.min = x$
**end if**
**if** $x > v.max$ **then**
    $v.max = x$
**end if**
$Insert(V.cluster[High(x)], low(x))$
$Insert(v.summary, high(x))$
```

Since there are two recursive calls to *Insert* within itself, the runtime of *Insert* now becomes $O(\text{lg}U)$

## 8 Version 5: Lazy Propagation

The limiting factor from version 4 was the runtime of the *Insert* operation. To fix this, we can take advantage of the fact that the minimum value does not always need to be immediately updated within the cluster. The new *Insert* algorithm becomes:

**Algorithm 4** Version 5 Insert

```
**Procedure** INSERT($V$, $x$)
**if** $v.min = \emptyset$ **then**
  $v.min = v.max = x$
  return
**end if**
**if** $x < v.min$ **then**
  swap($x$, $v.min$)
**end if**
**if** $X > v.max$ **then**
  $v.max = x$
**end if**
**if** $v.cluster[High(x)].min = \emptyset$ **then**
  Insert($v.summary$, $High$(x))
**end if**
Insert($v.cluster[High(x)]$, $Low(x)$)
**End Procedure**
```

Note that in the last if-statement, we only insert into the summary when it is absolutely necessary. This allows us to achieve an amortized runtime of $O(\text{lglg}U)$, since approximately once in every $\sqrt{U}$ insertions, it takes a worst-case $O(\text{lg}U)$ time to additionally update the summary, but otherwise we only need to perform a single insertion.

## References

[1] Peter van Emde Boas Preserving Order in a Forest in less than Logarithmic Time. *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, 75–84, IEEE, 1975.