van Emde Boas Trees and a Faster Shortest Path Algorithm

Tomer Azuz

Vrije Universiteit Amsterdam

25/08/2022

Overview

Overview

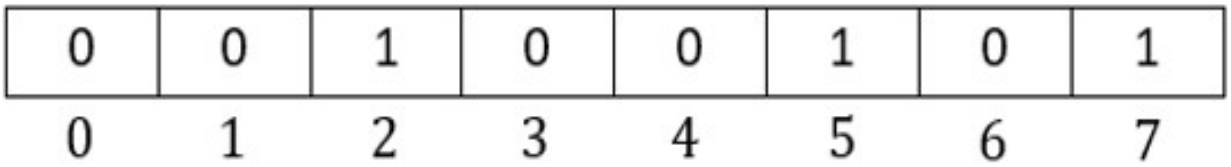
van Emde Boas Trees

- A data structure that supports all dynamic-set operations in $O(\log \log u)$ worst-case time
  - *insert*, *delete*, *lookup*, *successor*, *predecessor*, *min*, *max*
- Matches a lower bound for the predecessor problem

Overview

Bit Vector

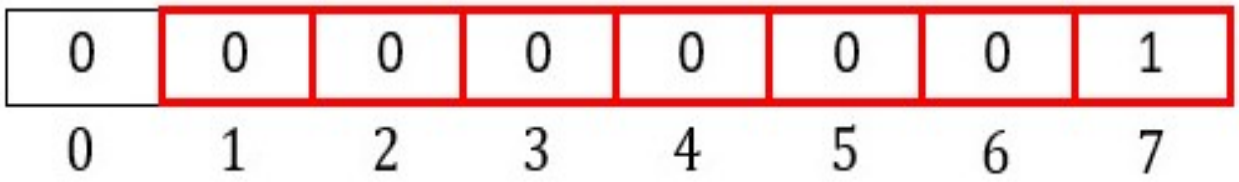
- Maintain a vector of n bits
- Elements in the set correspond to the indices in which the bit is set



A bit vector representing the set $\{2, 5, 7\}$

Bit Vector - *successor*(x)

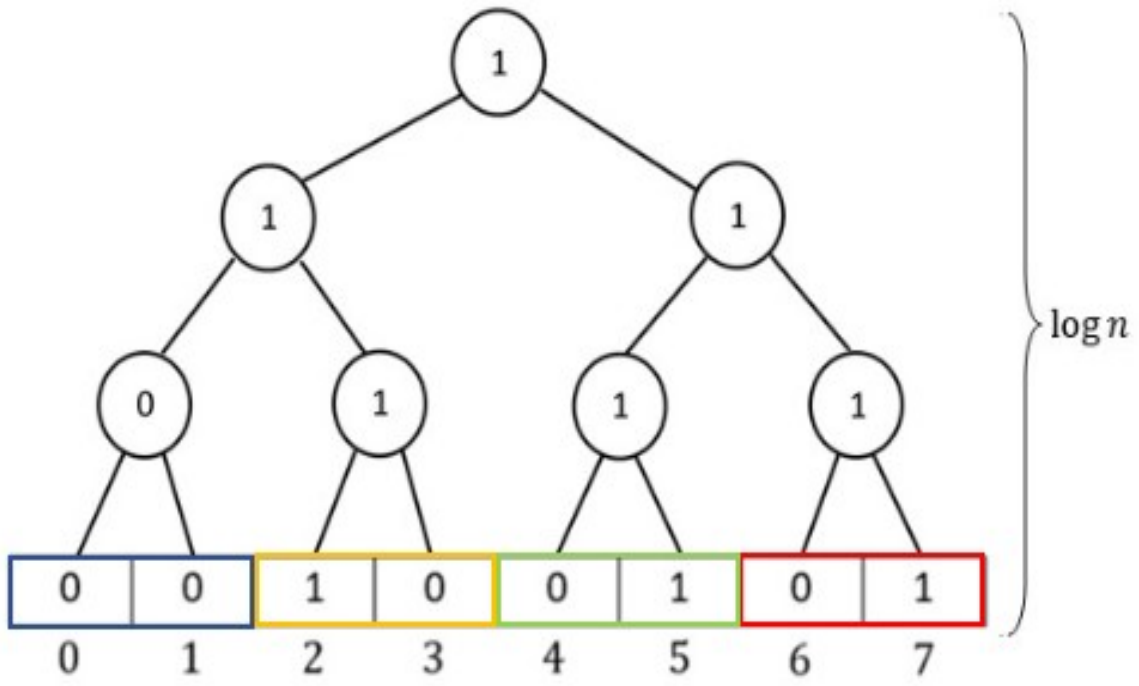
- Starting from index $x + 1$, scan all bits until a 1 is encountered
- $O(n)$ worst-case running time



Calling *successor*(0) on the singleton set $\{7\}$

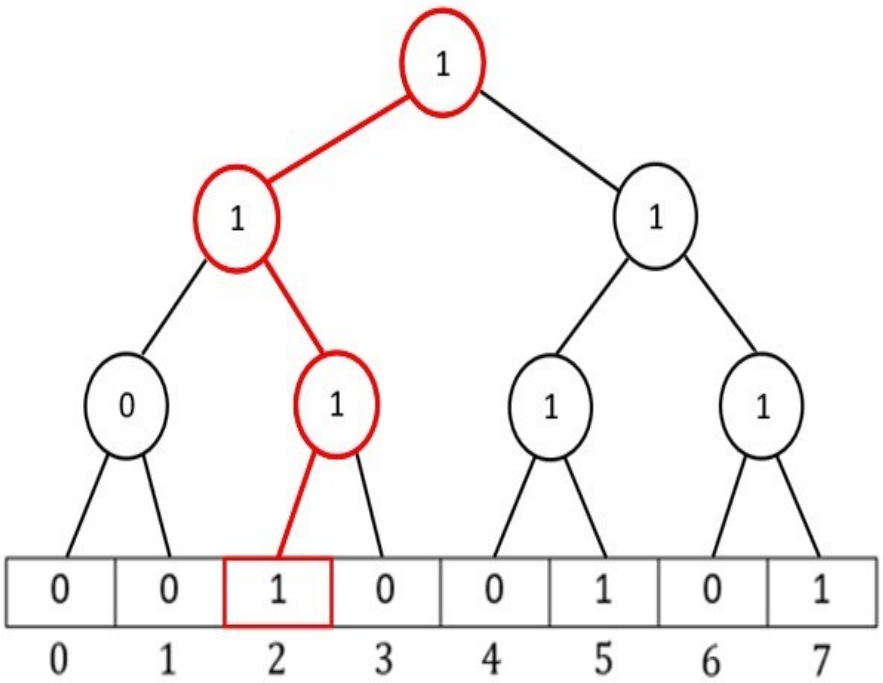
Speeding-up *successor*

- Augment a binary tree on top of the bit vector
- Bits are stored in the leaves
- Each non-leaf node stores the logical-or of its two children

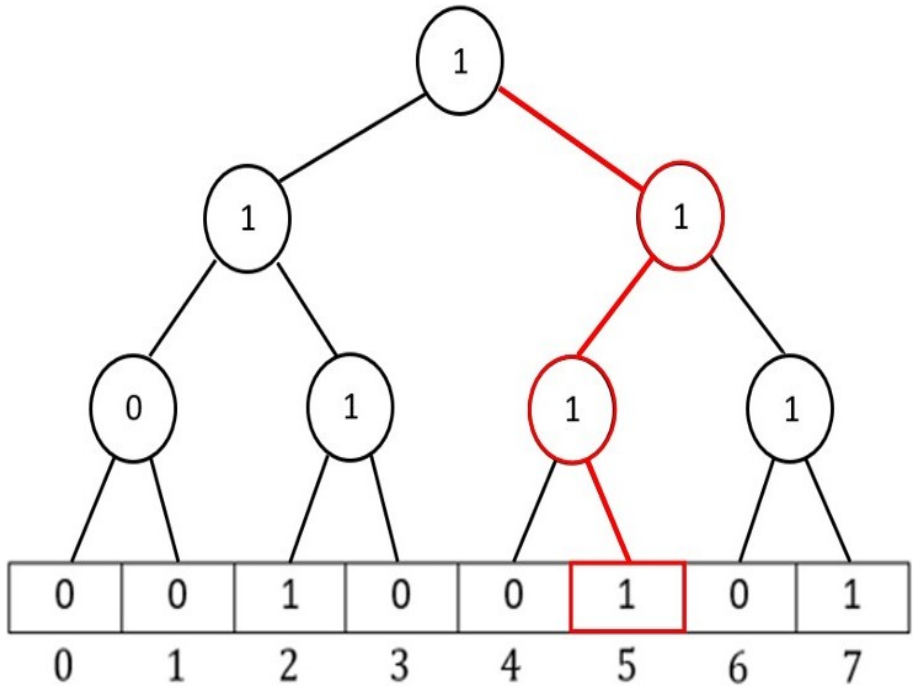


A binary tree augmented on top of a bit vector representing the set $\{2, 5, 7\}$

Successor of 2



Start from index 2, scan the tree bottom-up until a node is accessed from the left and its right child contains a 1



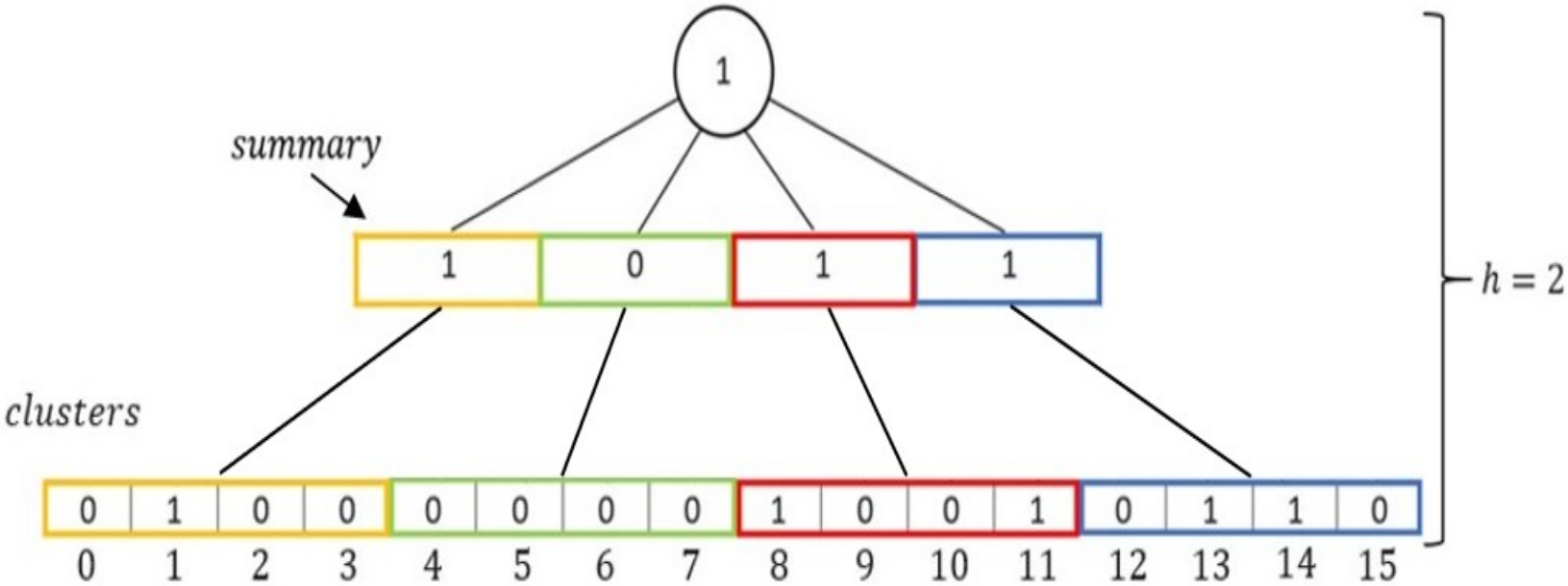
Head down towards the leaves, always taking the leftmost node containing a 1

successor: Time-Complexity

- Height of a binary tree is $\log n$
- Two scans through at most $\log n$ levels are required
- *successor* worst-case running time: $O(\log n)$

Augment a tree of degree $\sqrt{u}$ on top of the bit vector

- u: Range of all possible values stored in the set
- Assume u is an even power of two, such that $\sqrt{u}$ is an integer
- Middle level is an array of $\sqrt{u}$ bits, where each entry i indicates whether cluster i is empty or not
- Bottom level is divided into $\sqrt{u}$ clusters of size $\sqrt{u}$

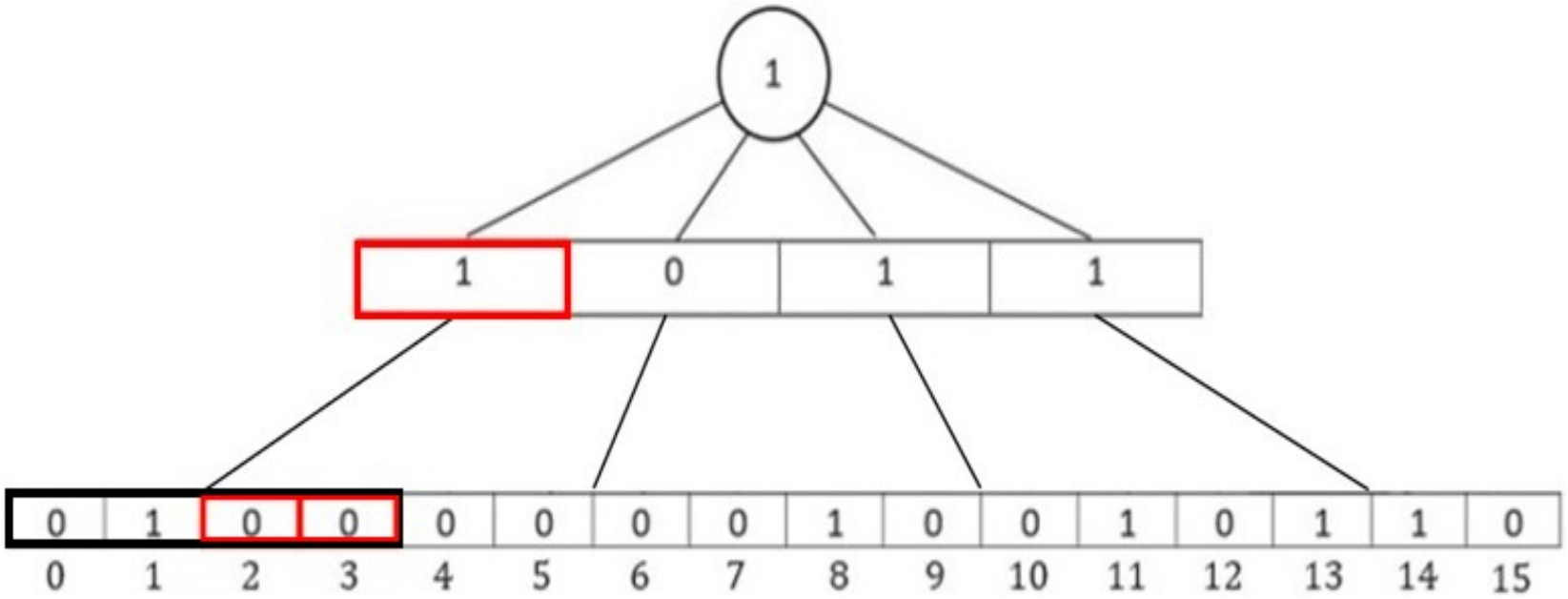


A tree of degree $\sqrt{u}$ augmented on top of a bit vector representing the set $\{1, 8, 11, 13, 14\}$

Definitions

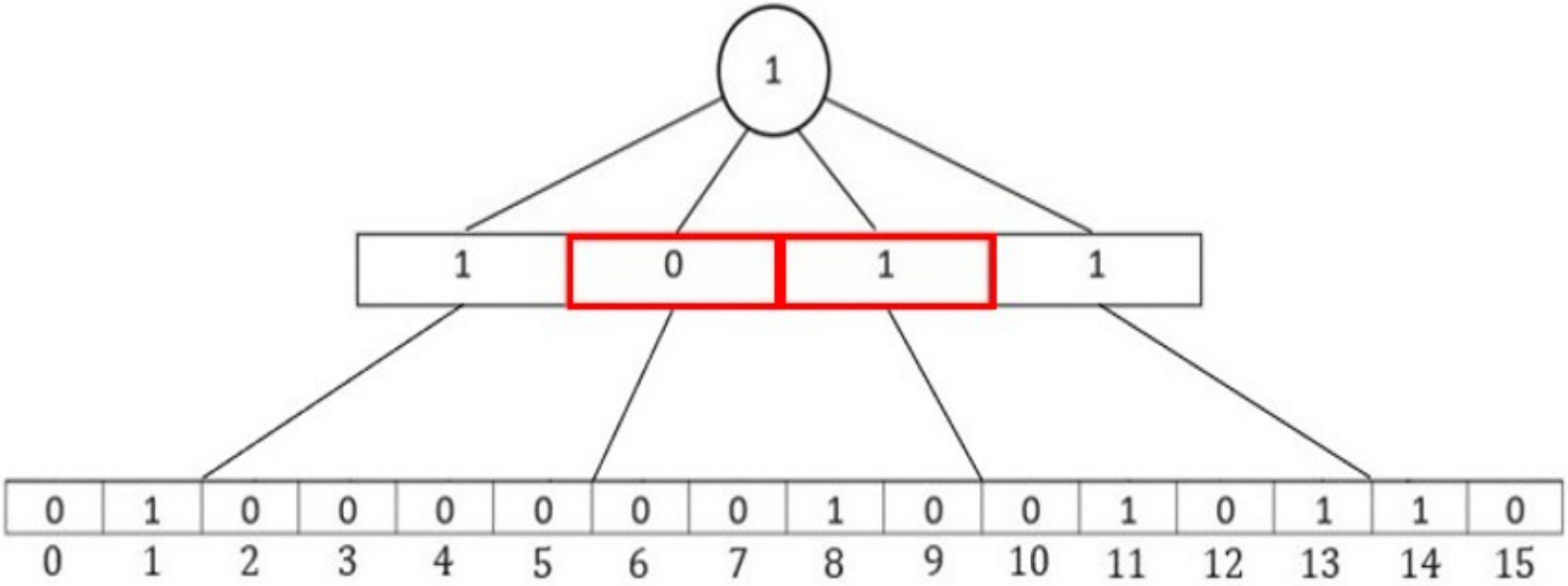
- $high(x)$: Determines the cluster number of x by computing $x/\sqrt{u}$
- $low(x)$: Determines the index of x within its cluster by computing x mod $\sqrt{u}$
- $index(x, y)$: Constructs an element from its cluster number x and its position y within its cluster by computing $x \cdot \sqrt{u} + y$

Successor of 1



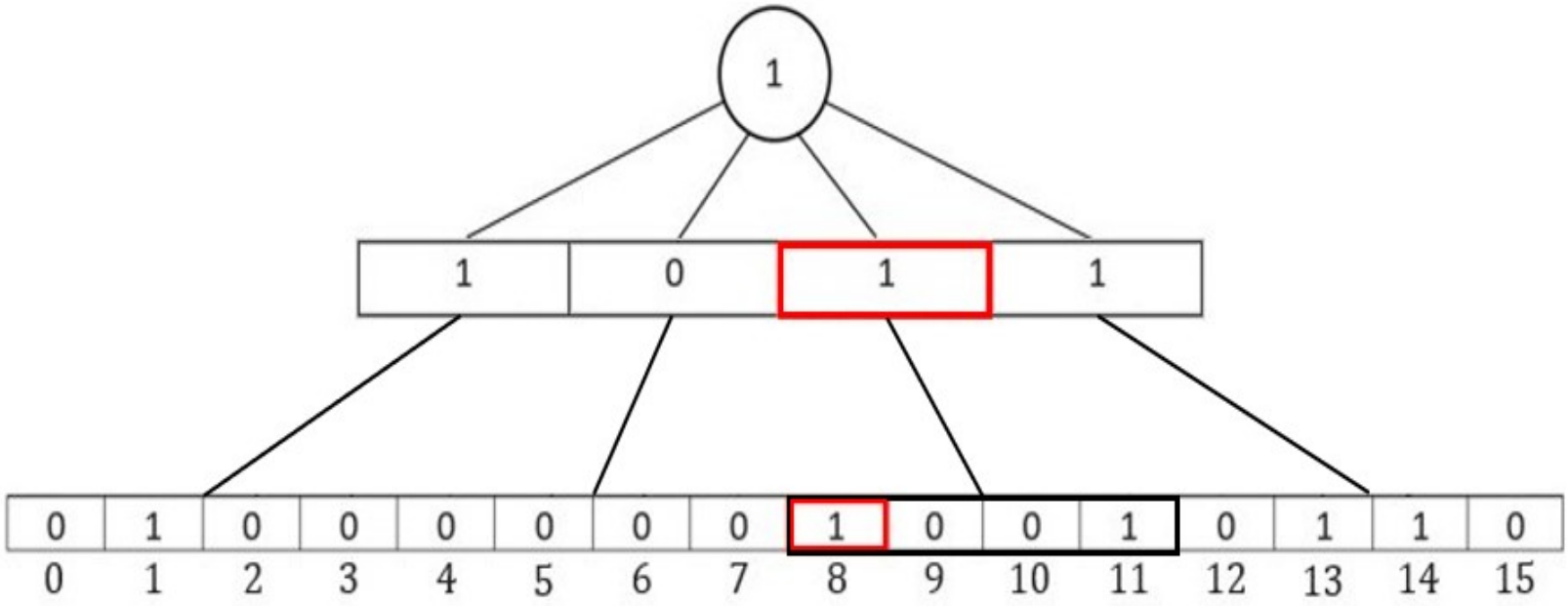
Starting from index 2, linearly scan cluster 0

Successor of 1



Find the next non-empty cluster by scanning the *summary* array

Successor of 1



Scan the next non-empty cluster.
The first position containing a 1 is the successor of 1

successor(x): Time Complexity

In the worst case:

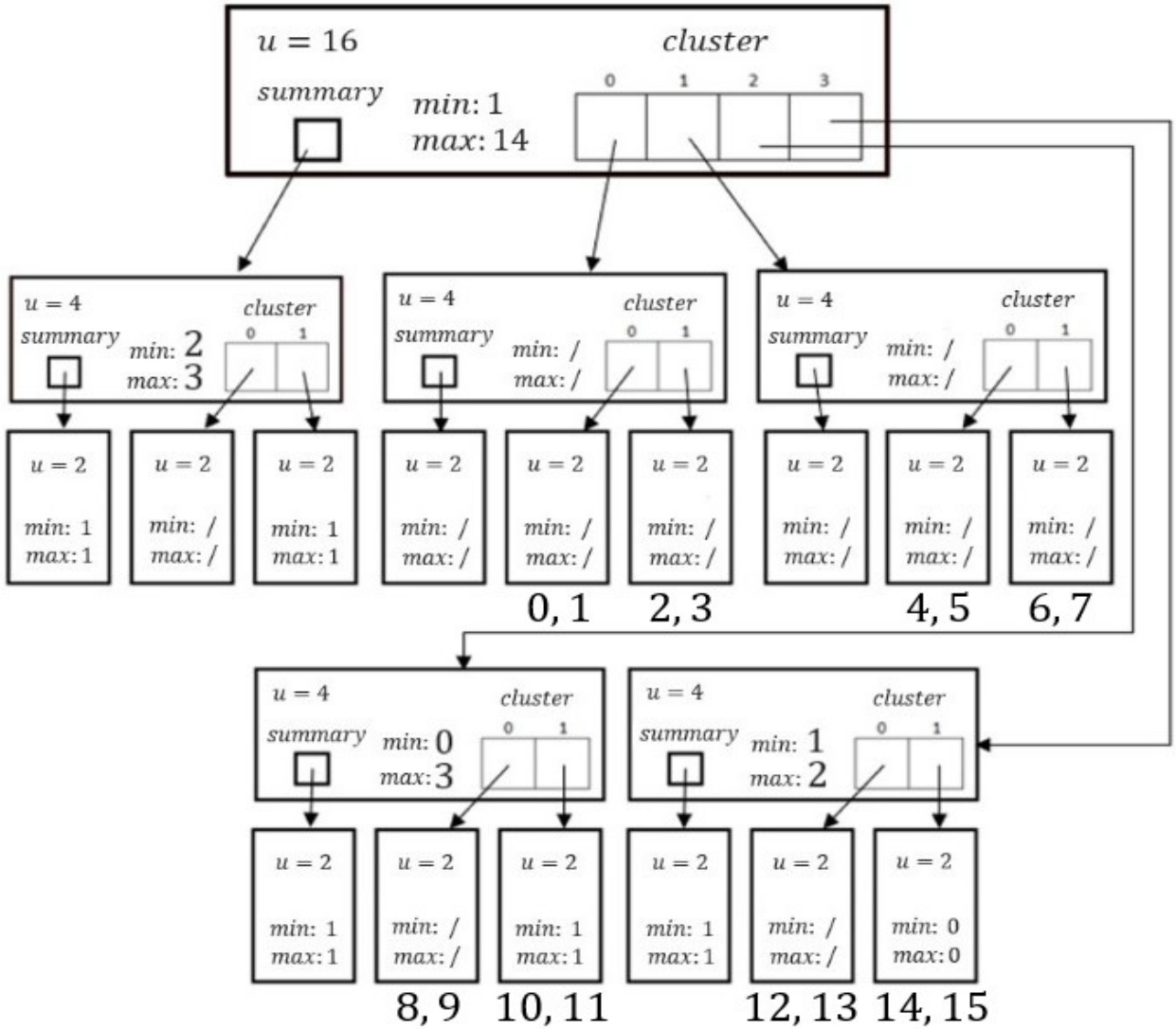
- All $\sqrt{u}$ bits in x's cluster are scanned
- All $\sqrt{u}$ bits in the *summary* array are scanned
- All $\sqrt{u}$ bits in the next non-empty cluster are scanned

Worst-case running time of *successor*: $O(\sqrt{u})$

van Emde Boas Tree Structure

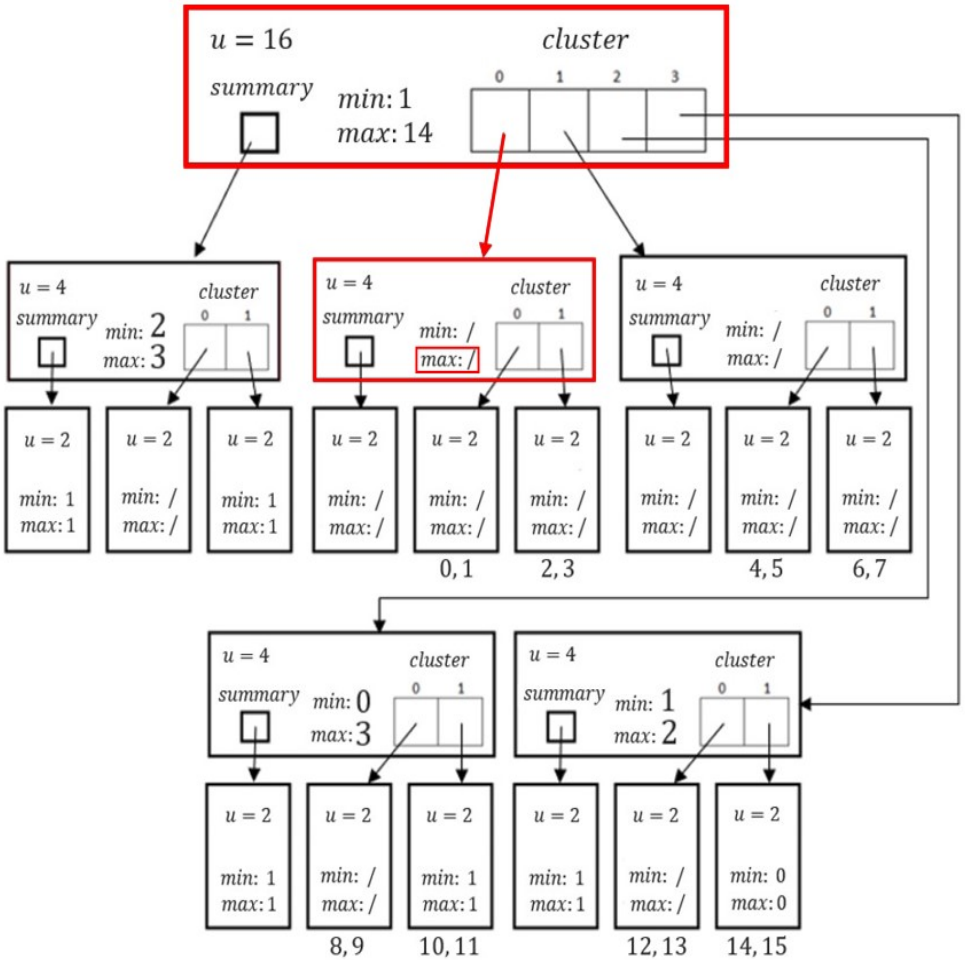
- Each non-leaf vEB node maintains the following attributes:
  - u: Range of all possible values in the set
  - *min* and *max*: Store the minimum and maximum elements
  - *summary*: A pointer to a vEB node of size $\lceil \sqrt{u} \rceil$
  - *cluster*: Array of pointers to $\lceil \sqrt{u} \rceil$ vEB nodes of size $\lfloor \sqrt{u} \rfloor$
  - A leaf vEB node only maintains the attributes u, *min* and *max*
- Recursive structure
  - Starting from a root node of size u, at each level of the tree u decreases by a factor of square root down to a base size of 2
  - The height of the resulting tree is $\log \log u$

van Emde Boas Tree



A van Emde Boas tree representing the set $\{1, 8, 11, 13, 14\}$

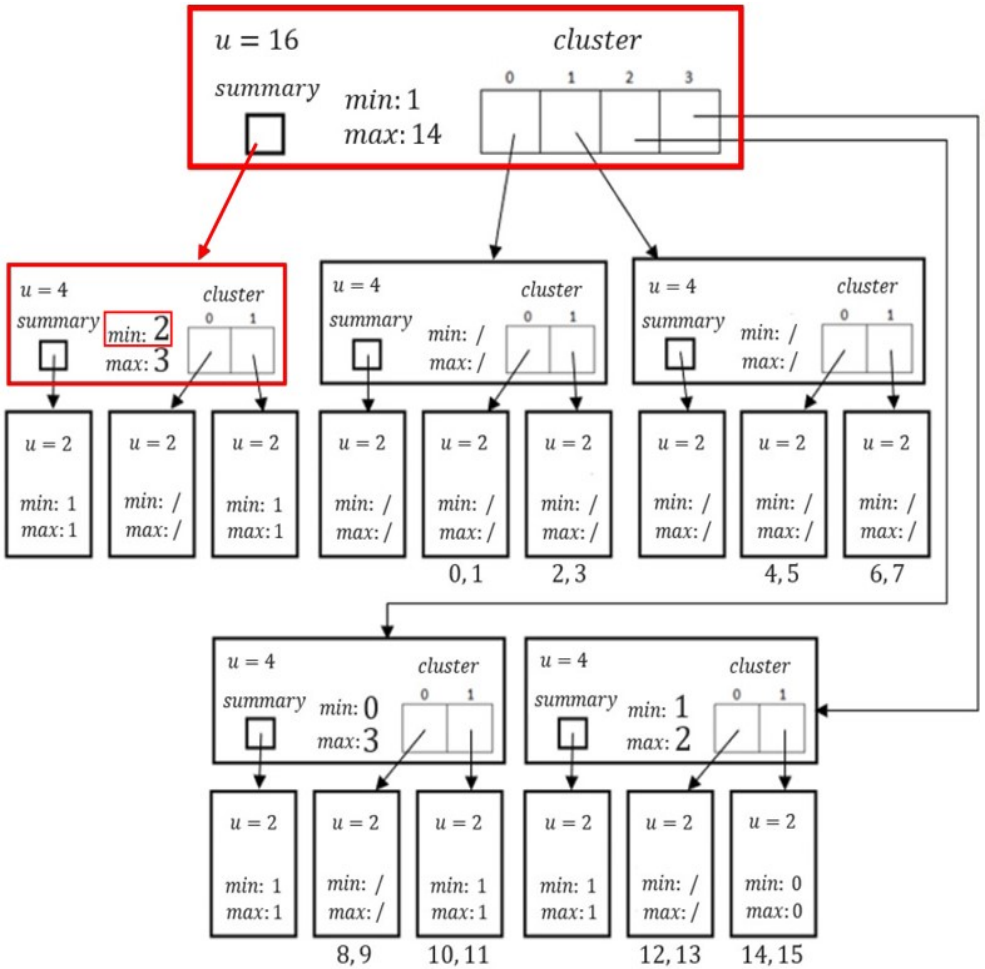
Successor of 1




```
successor(x)
    if V is a leaf:
        if x == 0 and max == 1:
            return 1
        else
            return null
    if V is nonempty and x < min:
        return min
    maxLow = clusters[high(x)].max
    if maxLow != null and low(x) < maxLow:
        offset = cluster[high(x)].successor(low(x))
        return index(high(x), offset)
    else
        succCluster = summary.successor(high(x))
        if succCluster == null:
            return null
        else
            offset = cluster[succCluster].min
            return index(succCluster, offset)
```

Search within 1's cluster

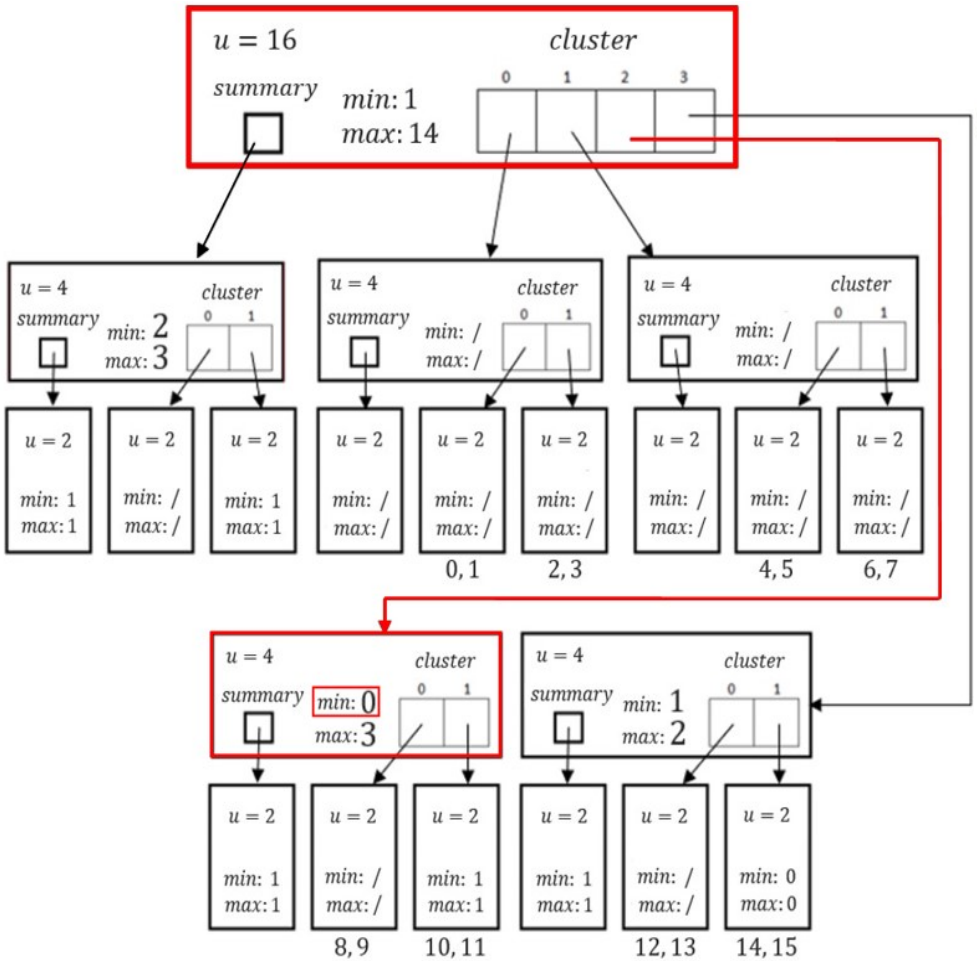
Successor of 1




```
successor(x)
    if V is a leaf:
        if x == 0 and max == 1:
            return 1
        else
            return null
    if V is nonempty and x < min:
        return min
    maxLow = clusters[high(x)].max
    if maxLow != null and low(x) < maxLow:
        offset = cluster[high(x)].successor(low(x))
        return index(high(x), offset)
    else
        succCluster = summary.successor(high(x))
        if succCluster == null:
            return null
        else
            offset = cluster[succCluster].min
            return index(succCluster, offset)
```

Query the *summary* for the next non-empty cluster

Successor of 1




```
successor(x)
    if V is a leaf:
        if x == 0 and max == 1:
            return 1
        else
            return null
    if V is nonempty and x < min:
        return min
    maxLow = clusters[high(x)].max
    if maxLow != null and low(x) < maxLow:
        offset = cluster[high(x)].successor(low(x))
        return index(high(x), offset)
    else
        succCluster = summary.successor(high(x))
        if succCluster == null:
            return null
        else
            offset = cluster[succCluster].min
            return index(succCluster, offset)
```

The minimum element in the next non-empty cluster is the successor of 1

successor: Time Complexity

- Two recursive calls
  - Only one is performed - $O(\log \log u)$
  - Remaining operations have $O(1)$ cost

```
successor(x)
    if V is a leaf:
        if x == 0 and max == 1:
            return 1
        else
            return null
    if V is nonempty and x < min:
        return min
    maxLow = clusters[high(x)].max
    if maxLow != null and low(x) < maxLow:
        offset = cluster[high(x)].successor(low(x))
        return index(high(x), offset)
    else
        succCluster = summary.successor(high(x))
        if succCluster == null:
            return null
        else
            offset = cluster[succCluster].min
            return index(succCluster, offset)
```

Overview

Dijkstra's Algorithm Optimization

- Shortest Path Problem: Find a path with the lowest weight between two vertices in a graph
- Dijkstra's Algorithm Optimization: Use a van Emde Boas tree as a priority queue

	Time Complexity	**Space Complexity**
Array	$O(\|V\|^2)$	$O(\|V\|)$
Min-Heap	$O(\|V\| + \|E\| \cdot \log \|V\|)$	$O(\|V\|)$
Fibonacci Heap	$O(\|V\| \cdot \log \|V\| + \|E\|)$	$O(\|V\|)$
vEB	$O((\|V\| + \|E\|) \cdot \log \log c)$	$O(\|V\| + c)$

Dijkstra's Algorithm Complexity Analysis

Thank you!