Lower bounds for heap operations

- Lower bound of $\Omega(n \log n)$ for comparison based sorting.
- As a consequence, at least one among the operations INSERT and EXTRACT-MIN takes $\Omega(\log n)$ time.
- Otherwise HEAPSORT would break down such a lower bound barrier.
- But what about keys in a specific range?

Heaps and sorting in linear time

- There exists faster sorting methods when keys are between **1** and **n** (where **n** is the number of elements).

- In particular, **RADIX SORT** takes O(n + k) time to sort **k** elements from **1** to $\mathbf{n}^{c}$ where **c** is a natural constant.

- van Emde Boas trees support priority queue operations in O(log log u) when all keys belong to the range from 0 to u – 1.

Van Emde Boas Trees

- Data structure that supports each of the following operations in **O(log log u)** time, where all keys belong to the range **{0, 1, ..., u-1} :**
  - INSERT/DELETE
  - MEMBER
  - MINIMUM/MAXIMUM
  - SUCCESSOR/PREDECESSOR
- We assume, by simplicity, that **u** is an exact power of two (**$u = 2^k$** for some natural number **k**).
- We also assume that there are not duplicated elements.

Preliminary approaches

In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- Direct adressing.
- Superimposing a binary tree structure.
- Superimposing a tree of constant height.

Preliminary approaches

In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- **Direct adressing.**
- Superimposing a binary tree structure.
- Superimposing a tree of constant height.

Direct adressing

- The direct-addressing stores the dynamic set as a bit vector.
- To store a dynamic set of values from the universe **{0, 1, ..., u-1}** we maintain an array **A[0 .. u-1]** of **u** bits.
- Entry **A[x]** holds **1** if the value **x** is in the dynamic set and **0** otherwise.
- INSERT, DELETE and MEMBER operations can be performed in O(1) time with this bit vector.
- MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR can take O(u) in the worst case since we might have to scan through O(u) elements.

Direct adressing

- S = {2, 3, 4, 5, 7, 14, 15} and u = 16.

0	0	1	1	1	1	0	1	0	0	0	0	0	0	1	1
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

- MINIMUM (A) = 2
- MAXIMUM (A) = 15
- SUCCESSOR (A, 7) = 14
- PREDECESSOR (A, 7) = 5

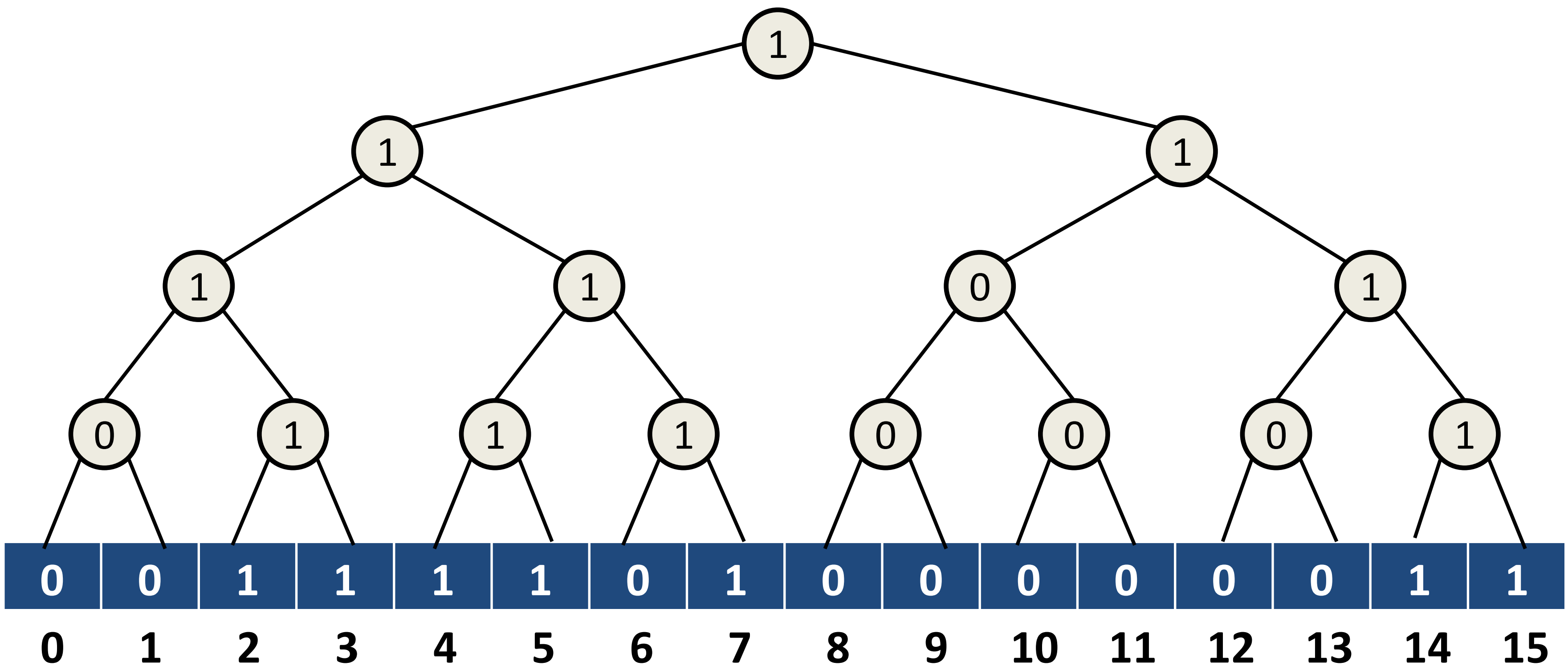
Preliminary approaches

In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- Direct adressing.
- **Superimposing a binary tree structure.**
- Superimposing a tree of constant height.

Superimposing a binary tree structure

- One can short-cut long scans in the bit vector by superimposing a binary tree on the top of it.



Superimposing a binary tree structure

- One can short-cut long scans in the bit vector by superimposing a binary tree on the top of it.
- Entries of the bit vector form the leaves of the binary tree.
- Each internal node contain 1 if and only if in its subtree contains 1.
- Bit stored in an internal node is the logical-or of its children.

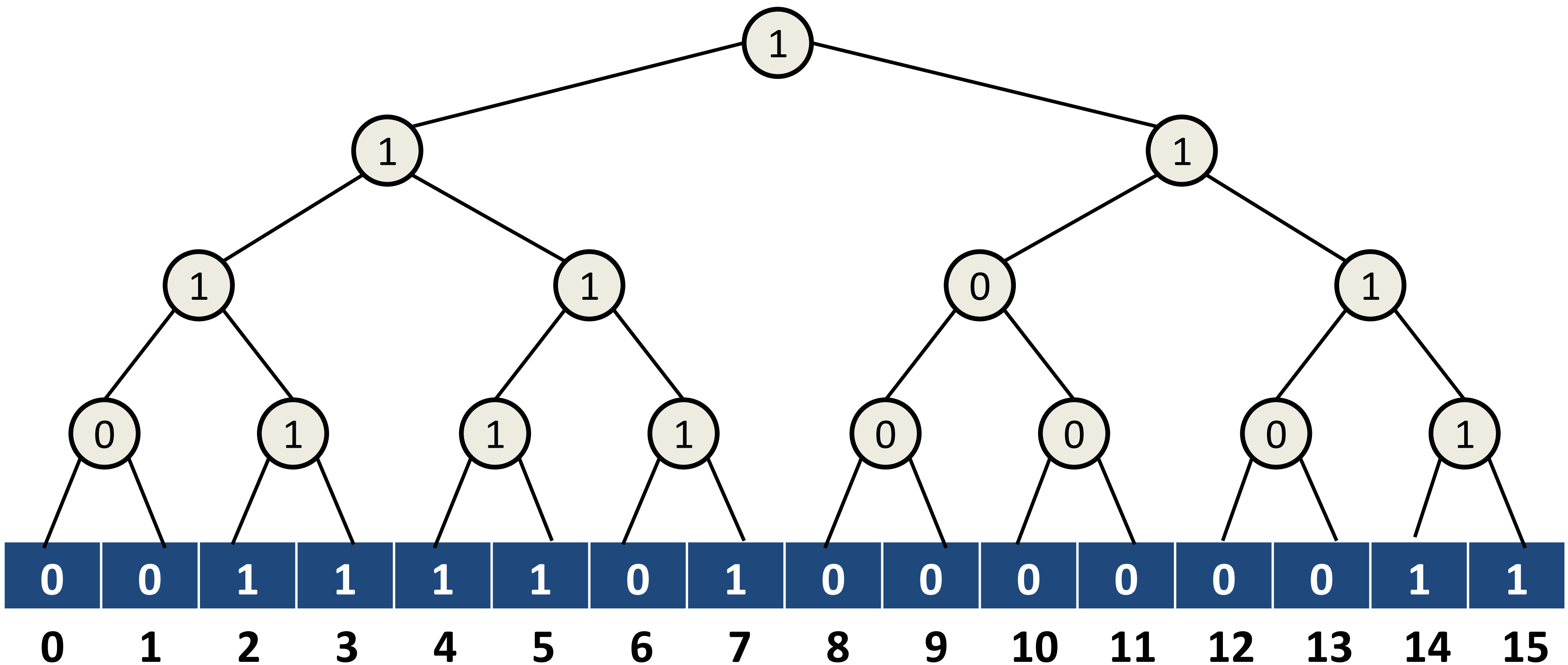
Superimposing a binary tree structure

- MINIMUM:
  - **Algorithm**: start at the root and head down toward the leaves, always taking the **leftmost** node containing 1.
  - **Time complexity**: O(log u)

- MAXIMUM:
  - **Algorithm**: start at the root and head down toward the leaves, always taking the **rightmost** node containing 1.
  - **Time complexity**: O(log u)

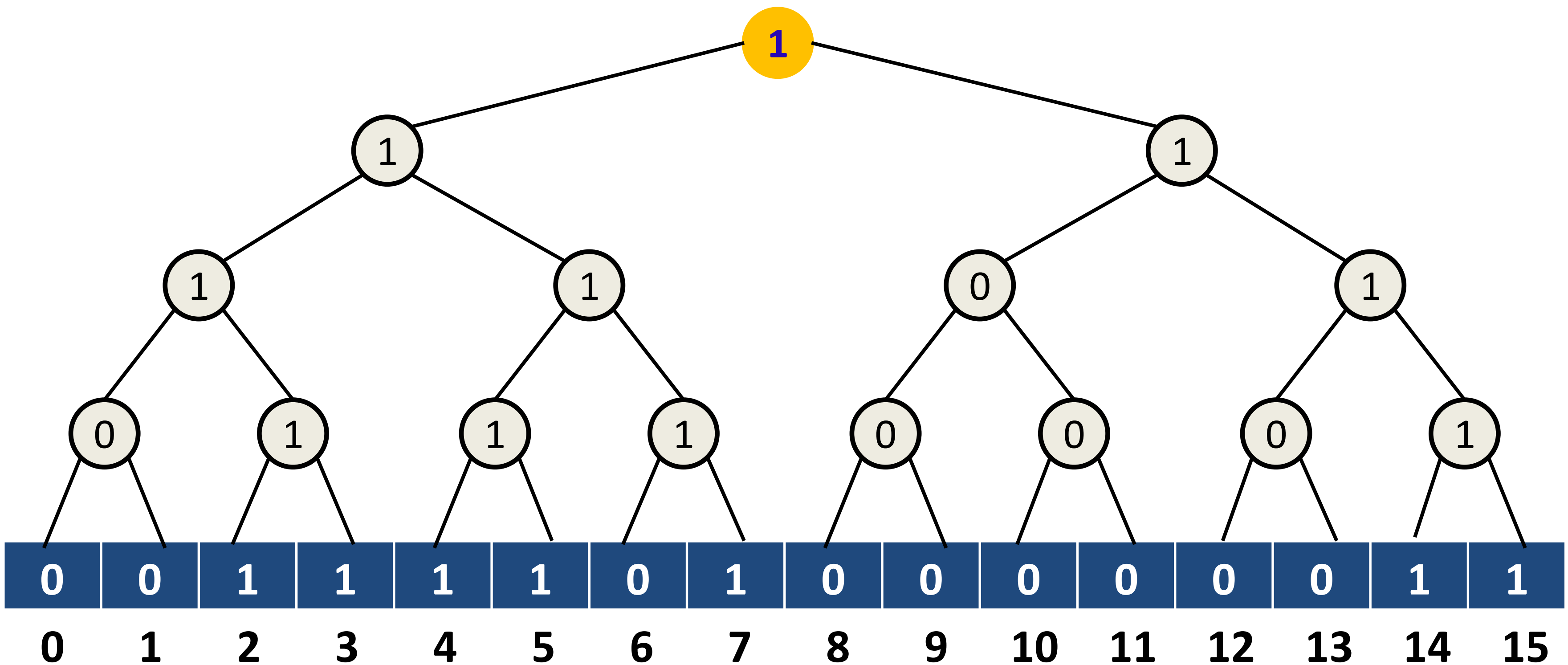
Superimposing a binary tree structure

- MINIMUM operation:



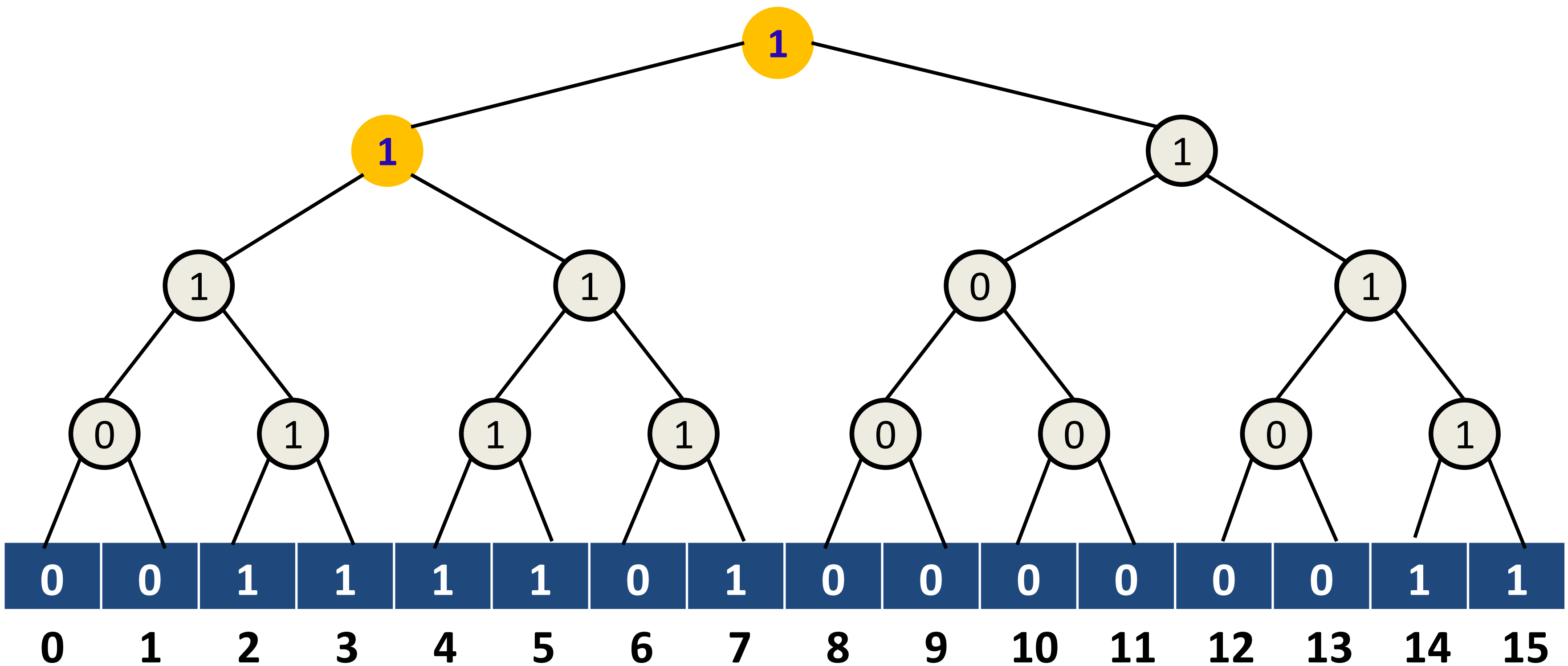
Superimposing a binary tree structure

- MINIMUM operation:



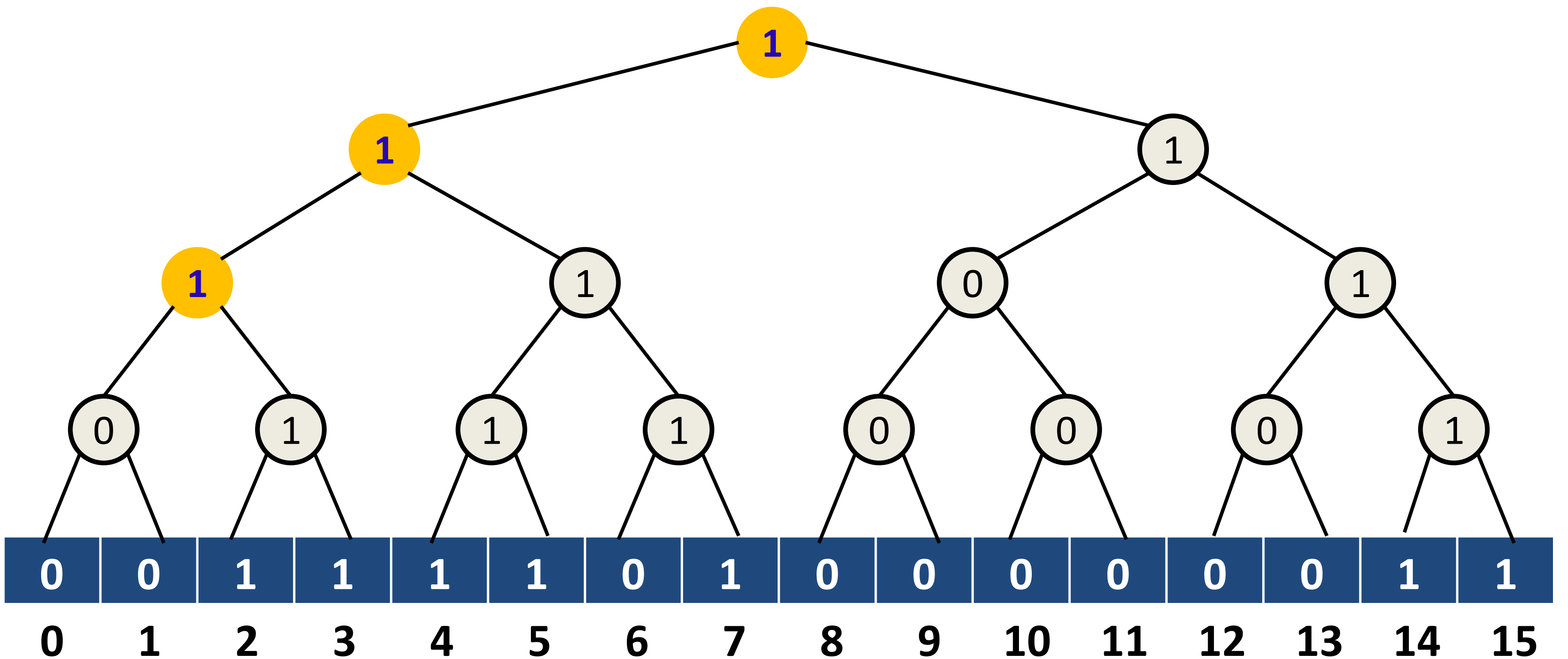
Superimposing a binary tree structure

- MINIMUM operation:



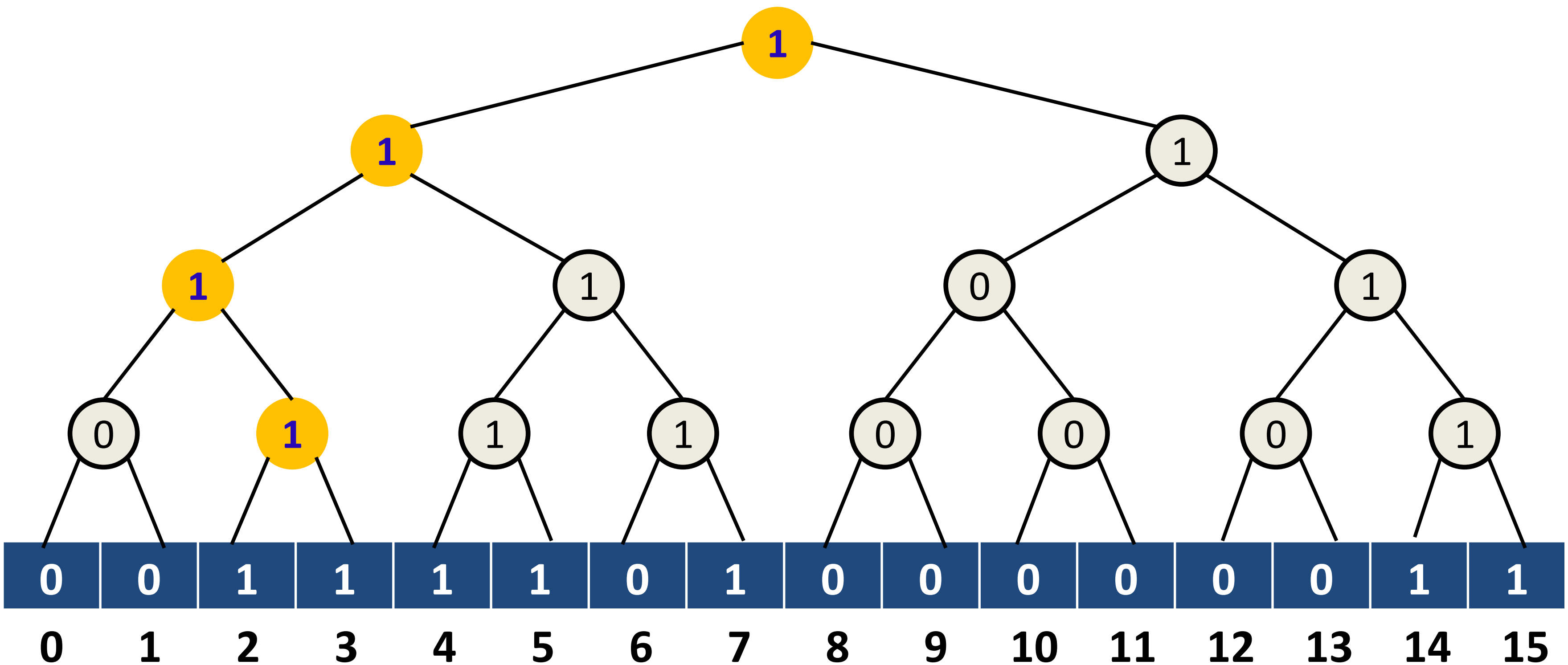
Superimposing a binary tree structure

- MINIMUM operation:



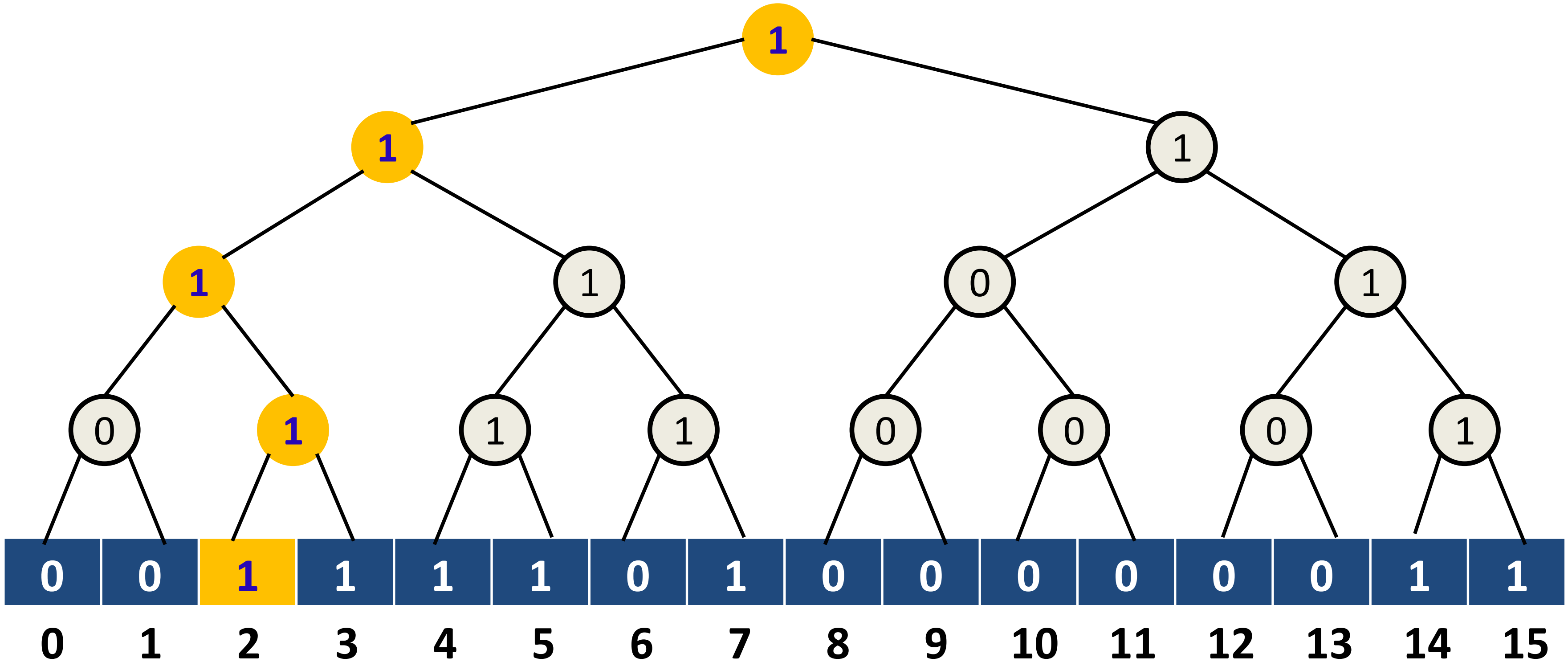
Superimposing a binary tree structure

- MINIMUM operation:



Superimposing a binary tree structure

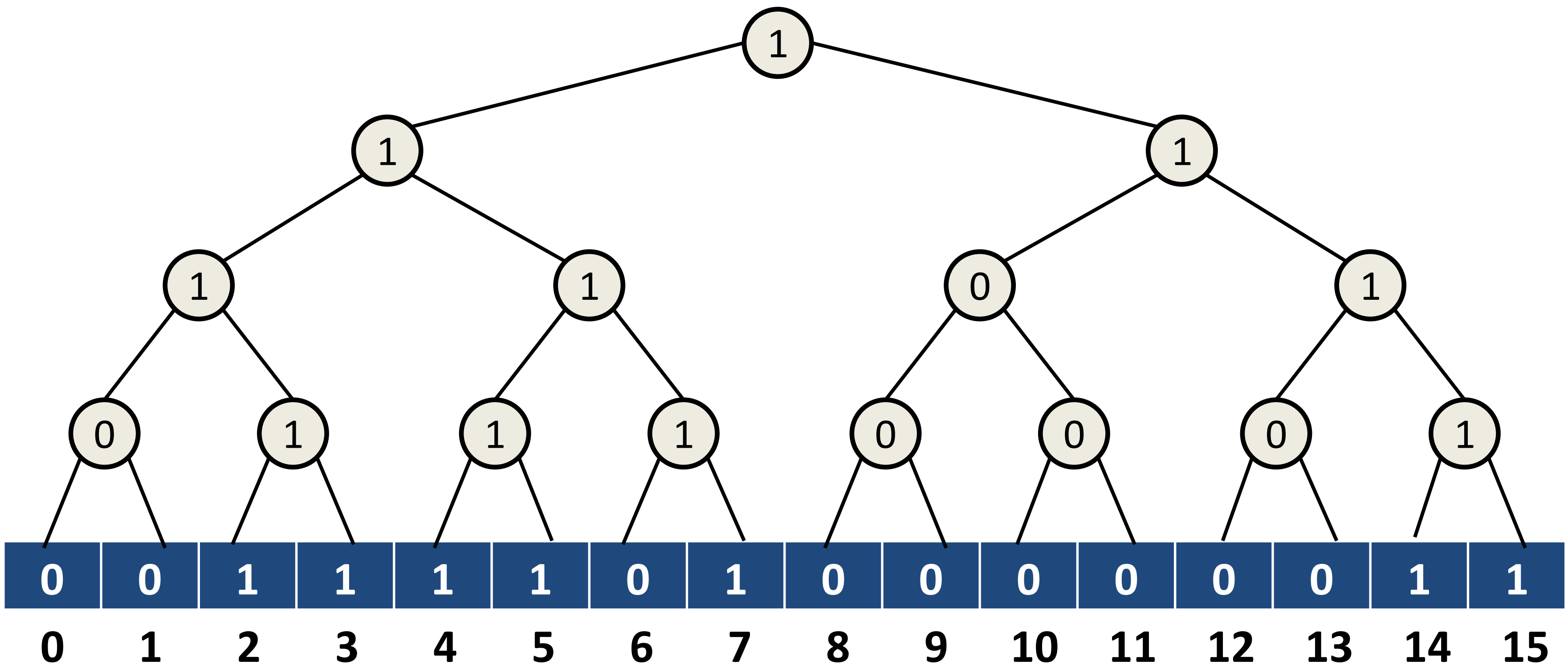
- MINIMUM operation:



Minimum found!

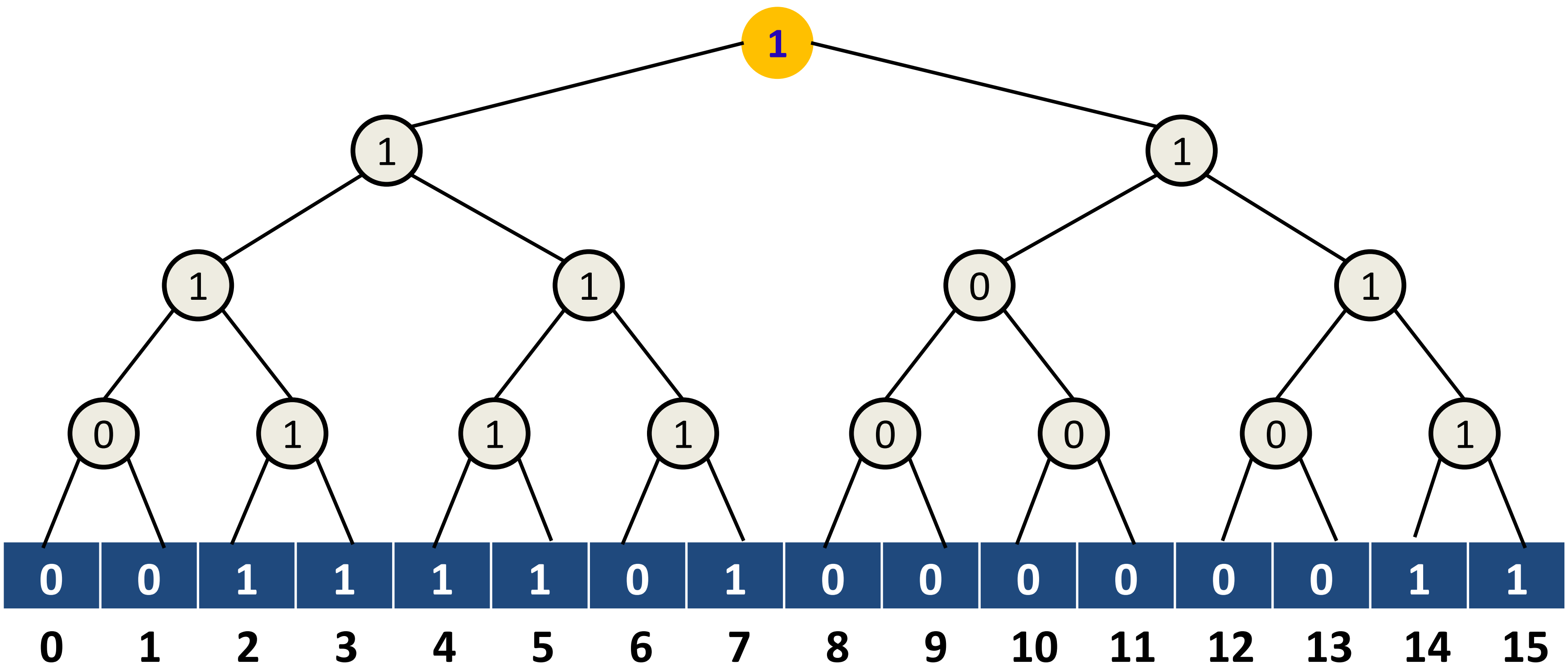
Superimposing a binary tree structure

- MAXIMUM operation:



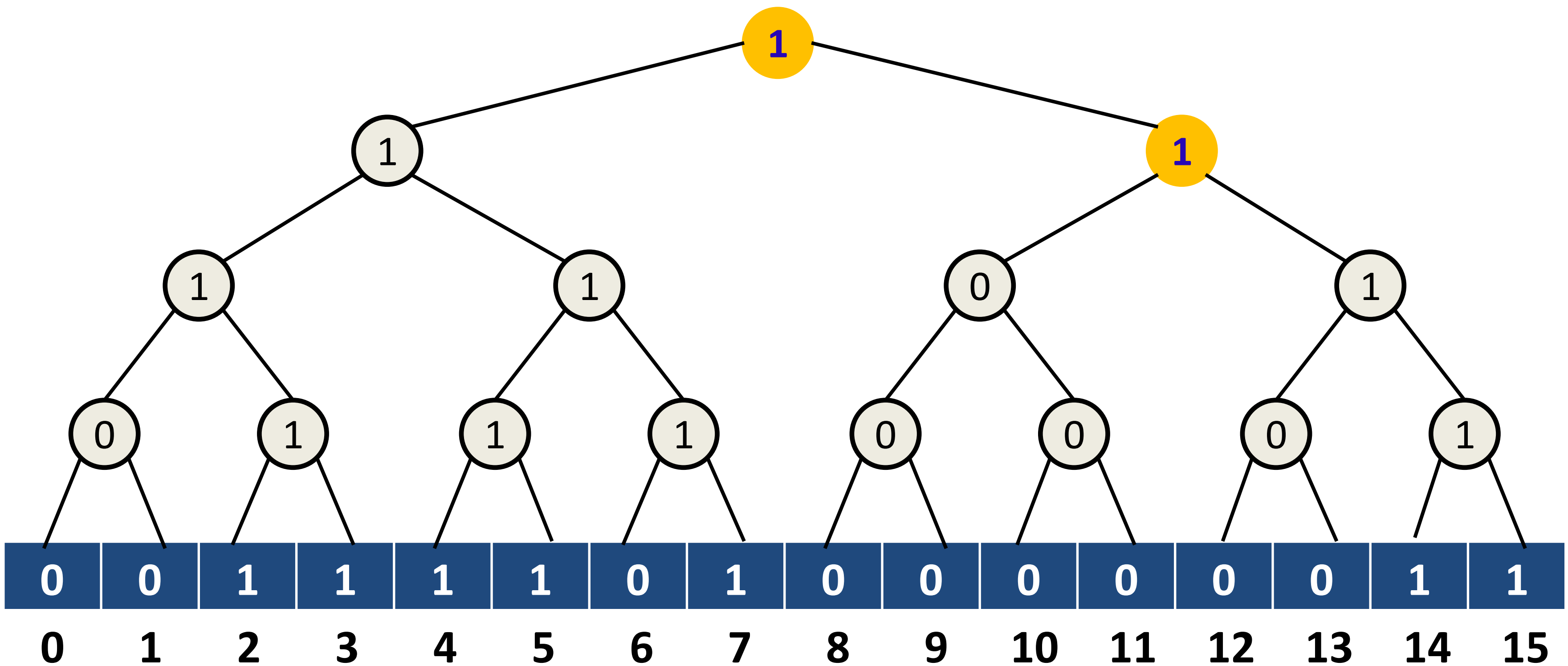
Superimposing a binary tree structure

- MAXIMUM operation:



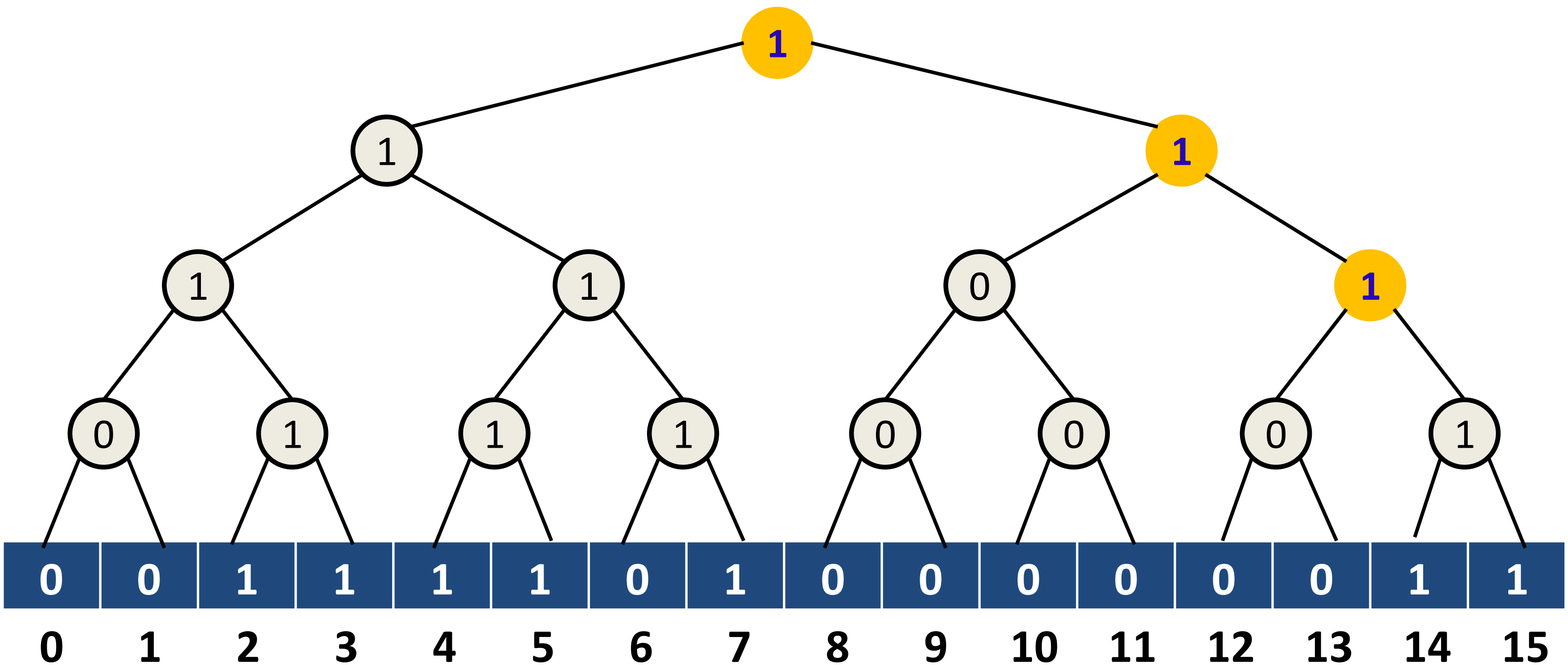
Superimposing a binary tree structure

- MAXIMUM operation:



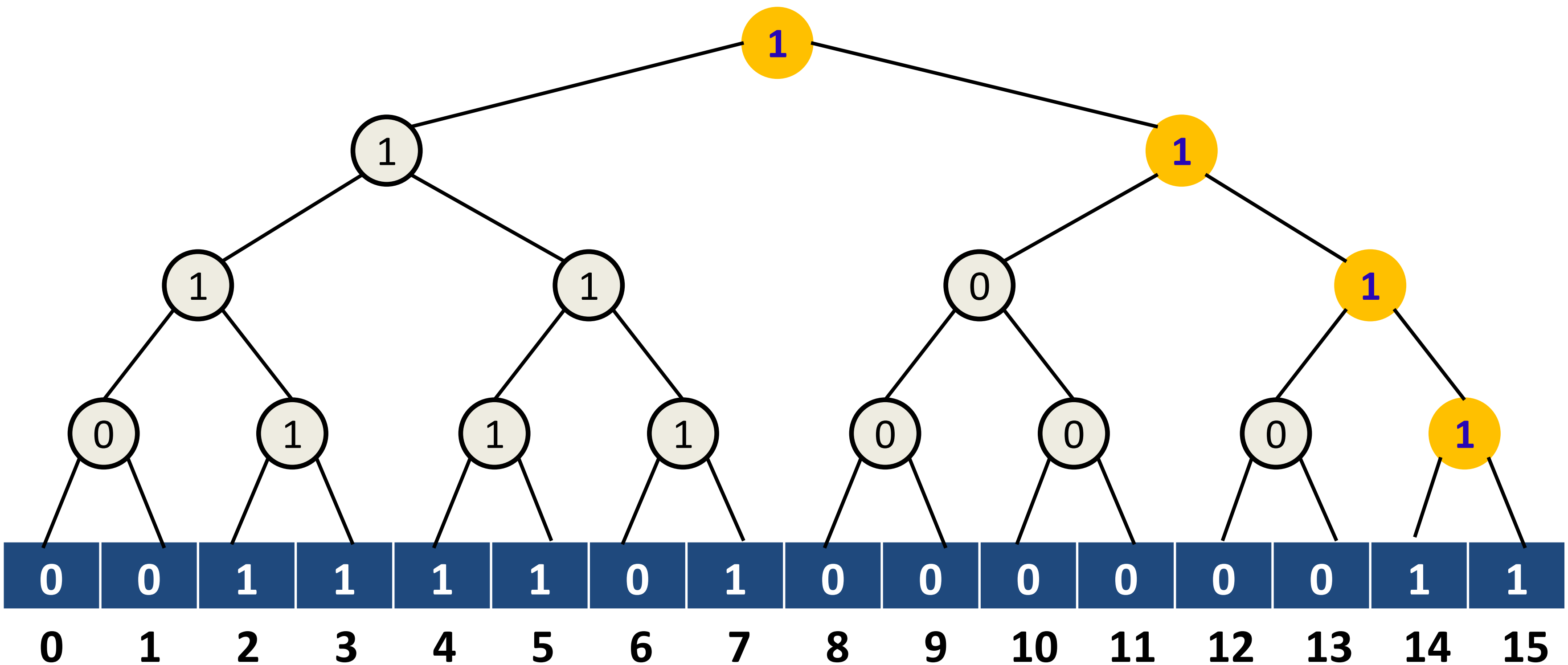
Superimposing a binary tree structure

- MAXIMUM operation:



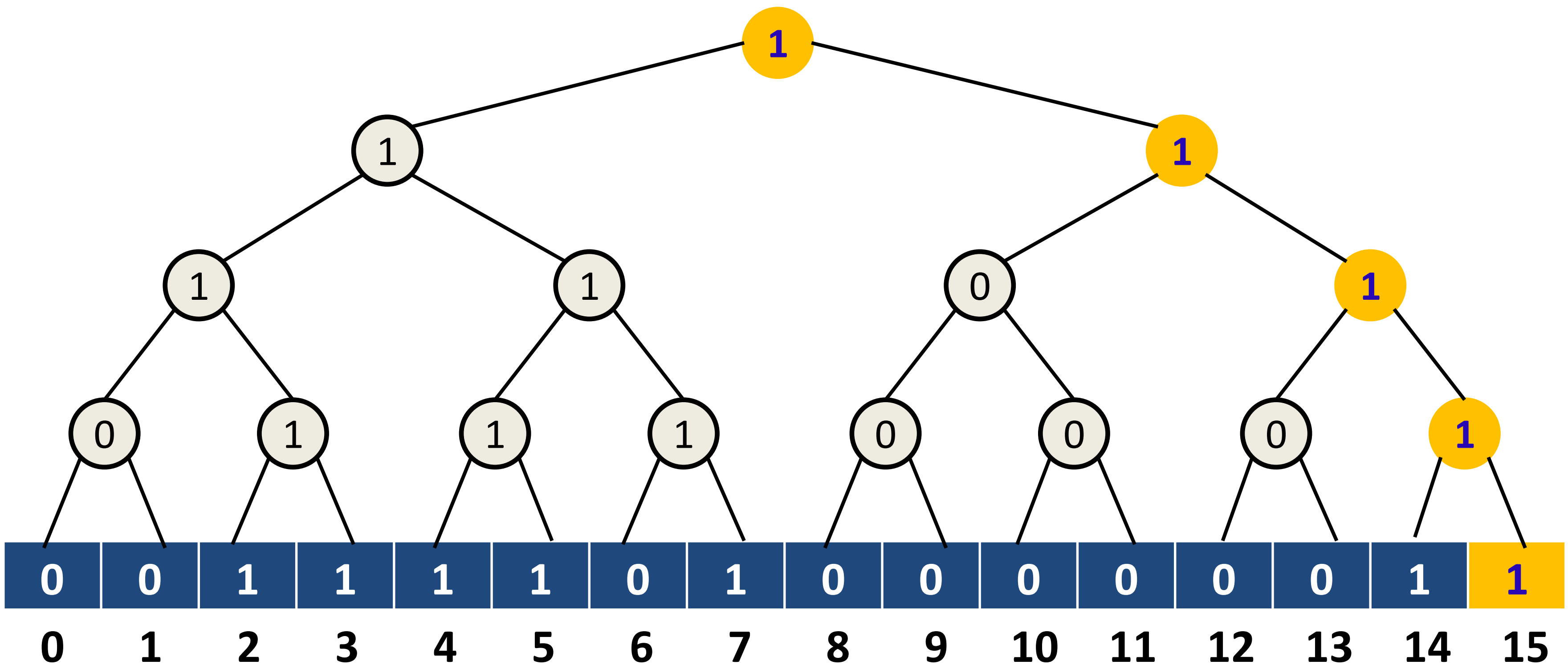
Superimposing a binary tree structure

- MAXIMUM operation:



Superimposing a binary tree structure

- MAXIMUM operation:



Maximum found!

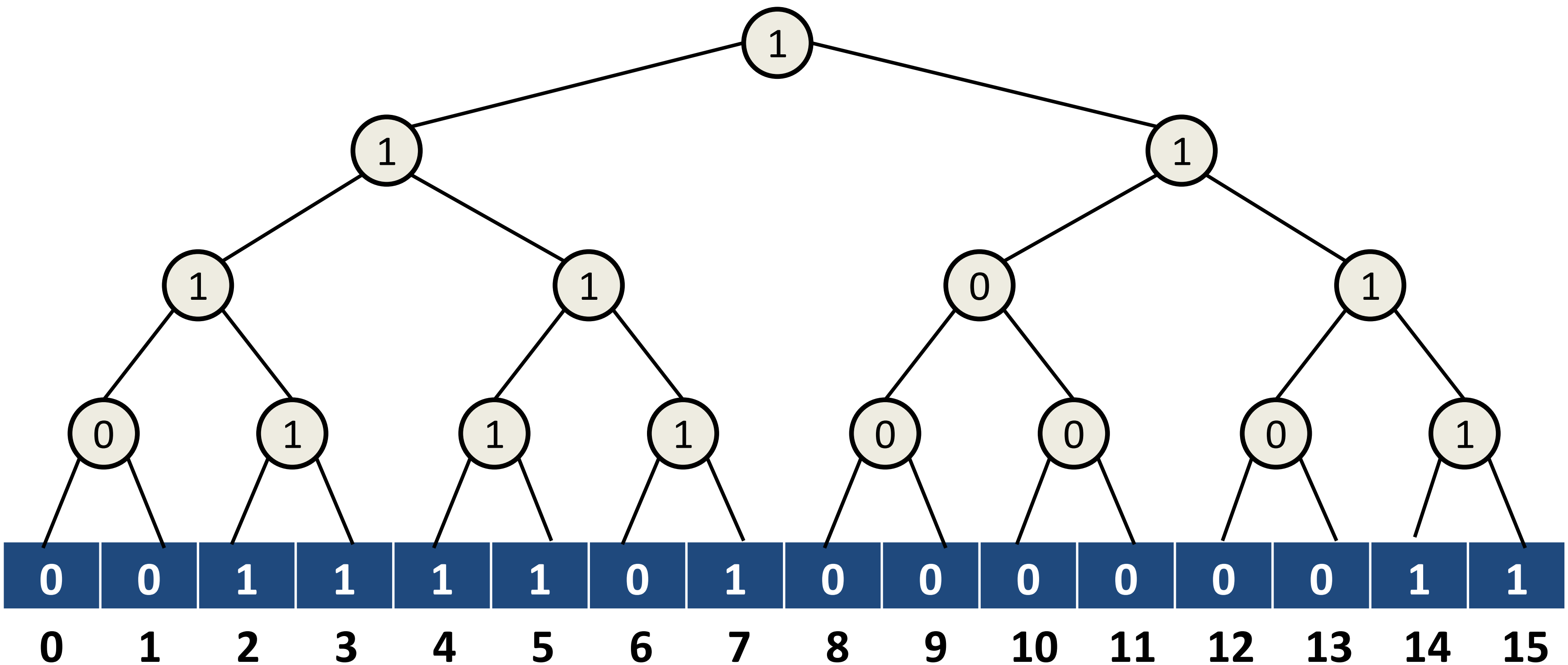
Superimposing a binary tree structure

- PREDECESSOR (A, x) :
  - Start at the leaf indexed by **x** and head up toward the root until we enter a node from the **right** and this node has a 1 in its **left** child **z**.
  - Head down through node **z**, always taking the **rightmost** node containing a 1.
  - **Time complexity**: O(log u).

- SUCCESSOR (A, x) :
  - Start at the leaf indexed by **x** and head up toward the root until we enter a node from the **left** and this node has a 1 in its **right** child **z**.
  - Head down through node **z**, always taking the **leftmost** node containing a 1.
  - **Time complexity**: O(log u).

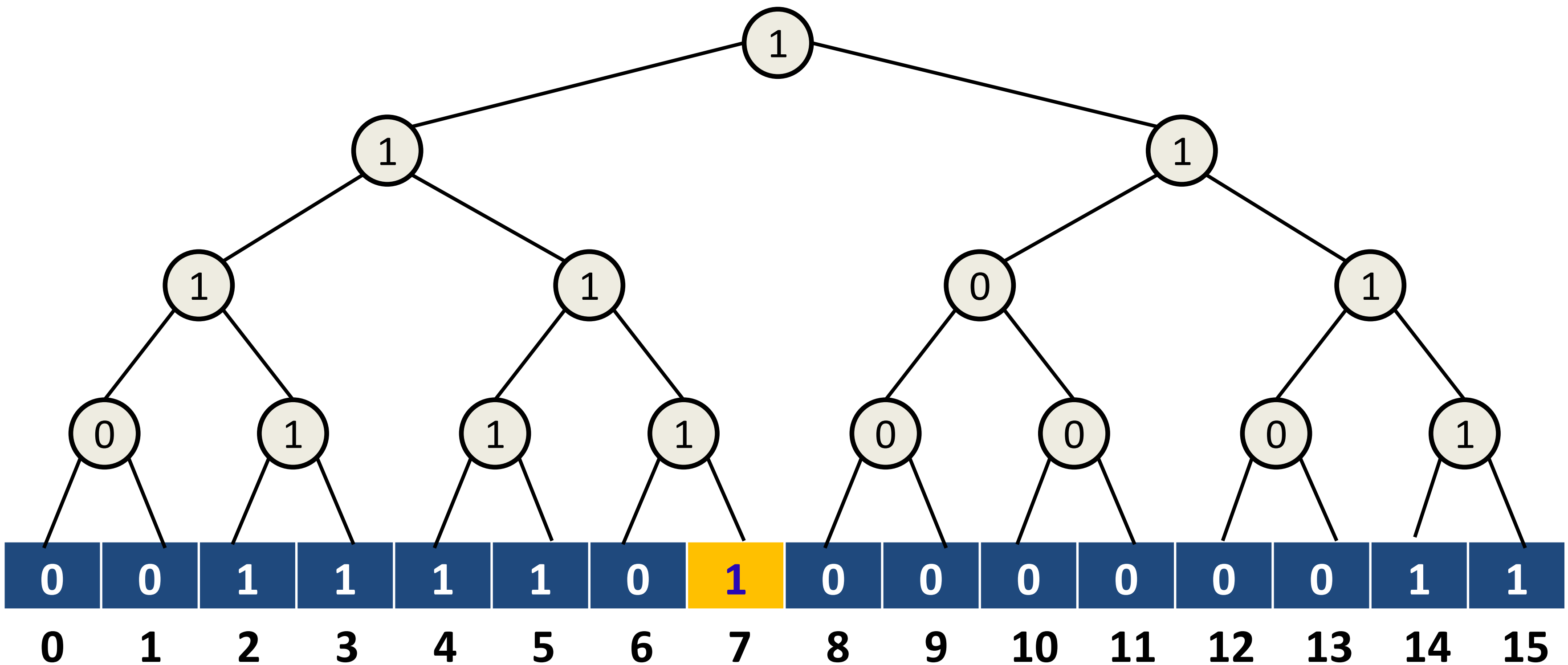
Superimposing a binary tree structure

- PREDECESSOR (A, 7):



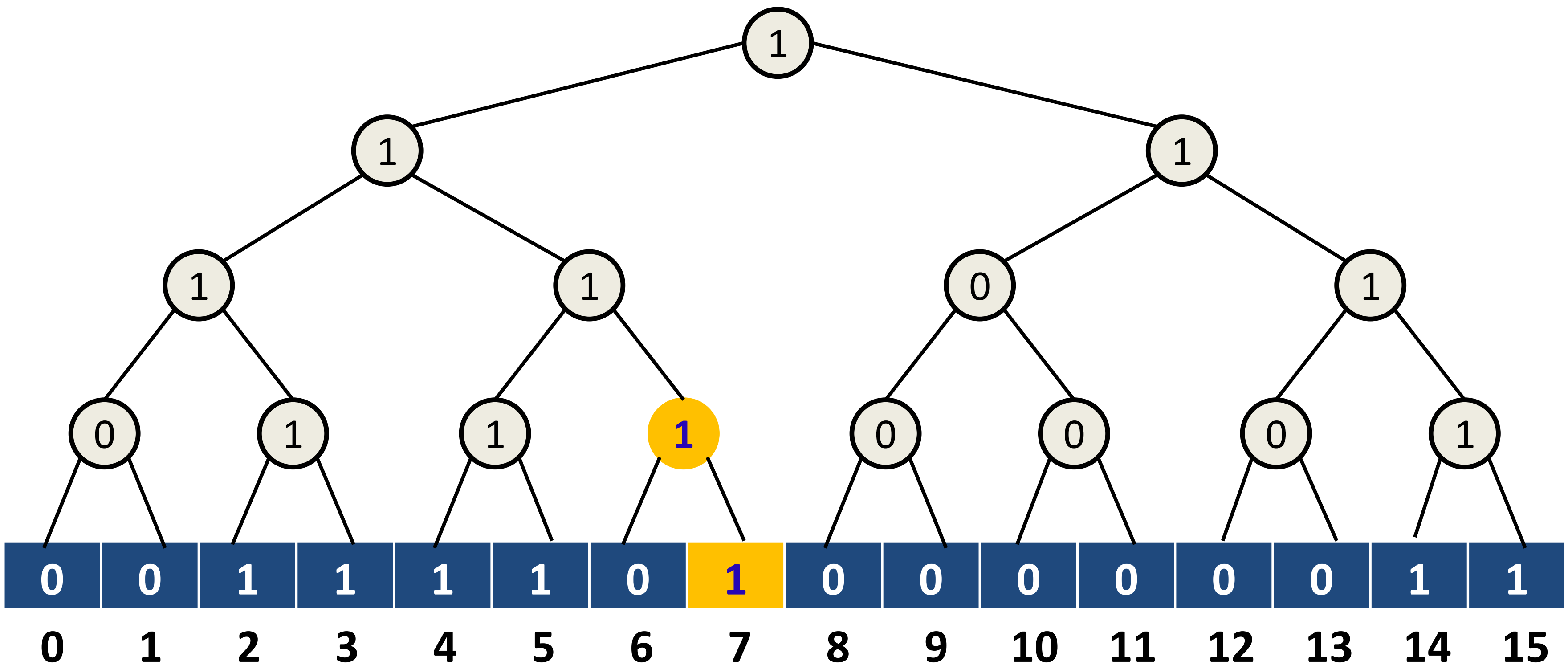
Superimposing a binary tree structure

- PREDECESSOR (A, 7):



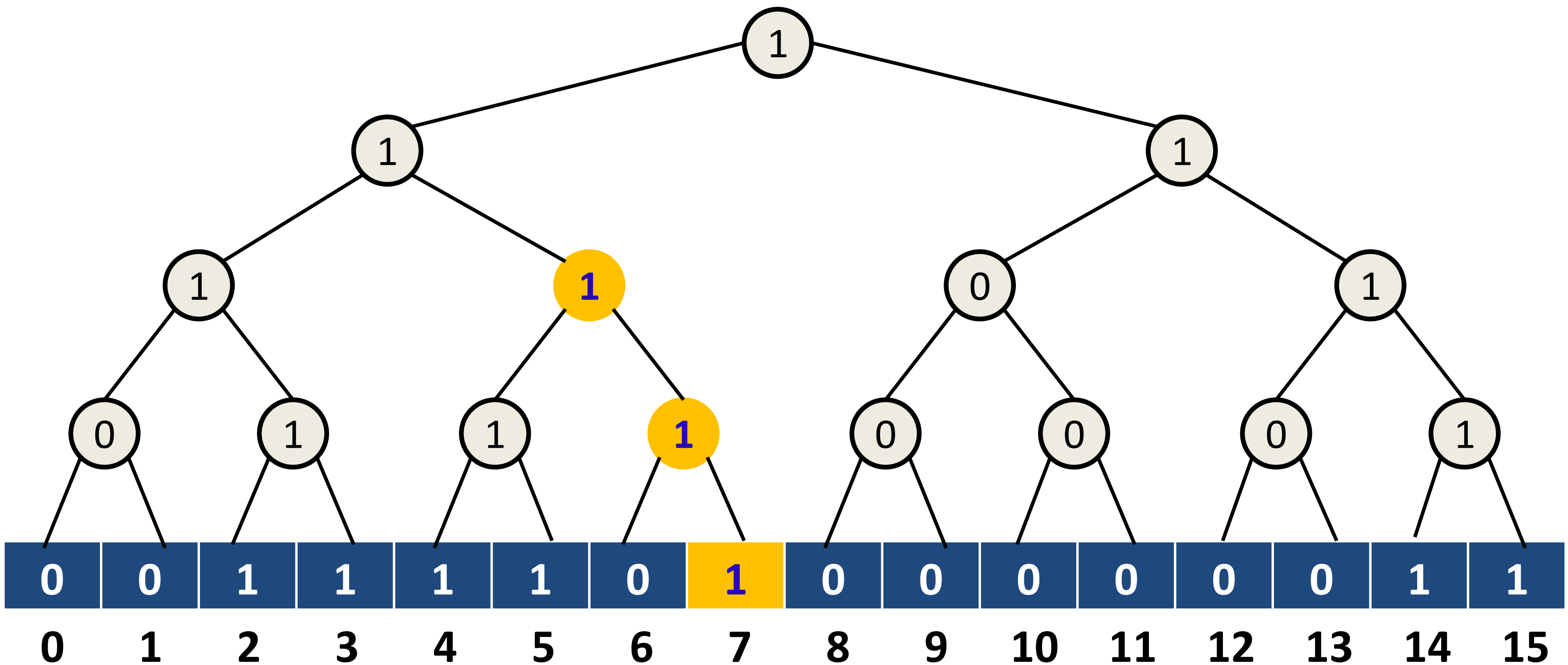
Superimposing a binary tree structure

- PREDECESSOR (A, 7):



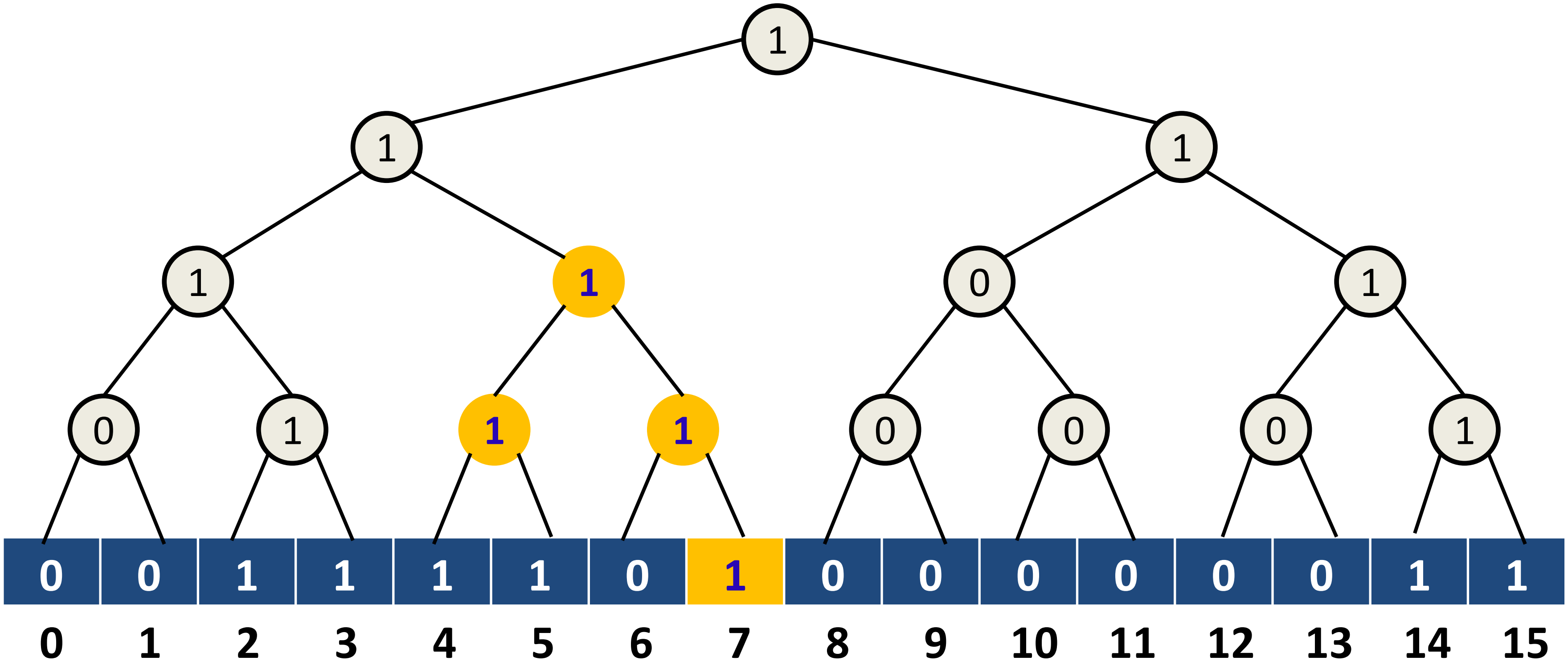
Superimposing a binary tree structure

- PREDECESSOR (A, 7):



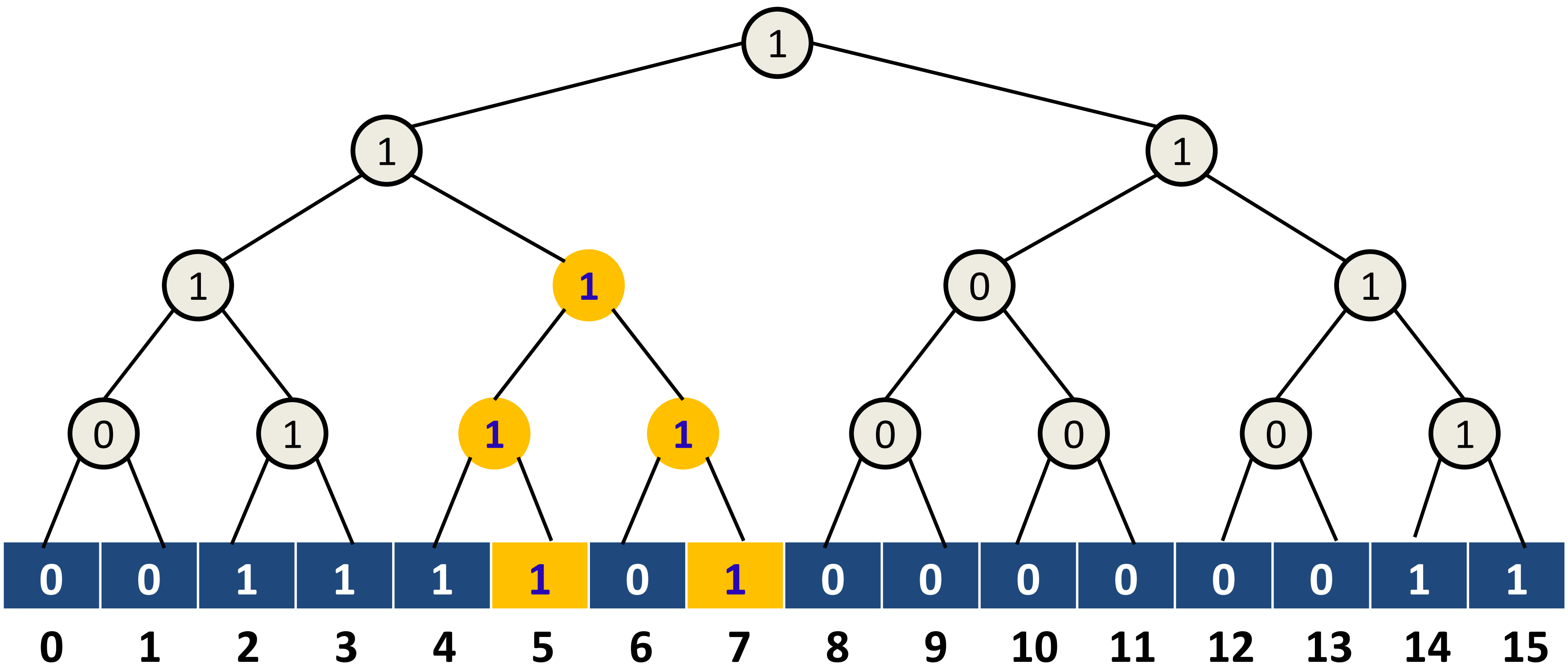
Superimposing a binary tree structure

- PREDECESSOR (A, 7):



Superimposing a binary tree structure

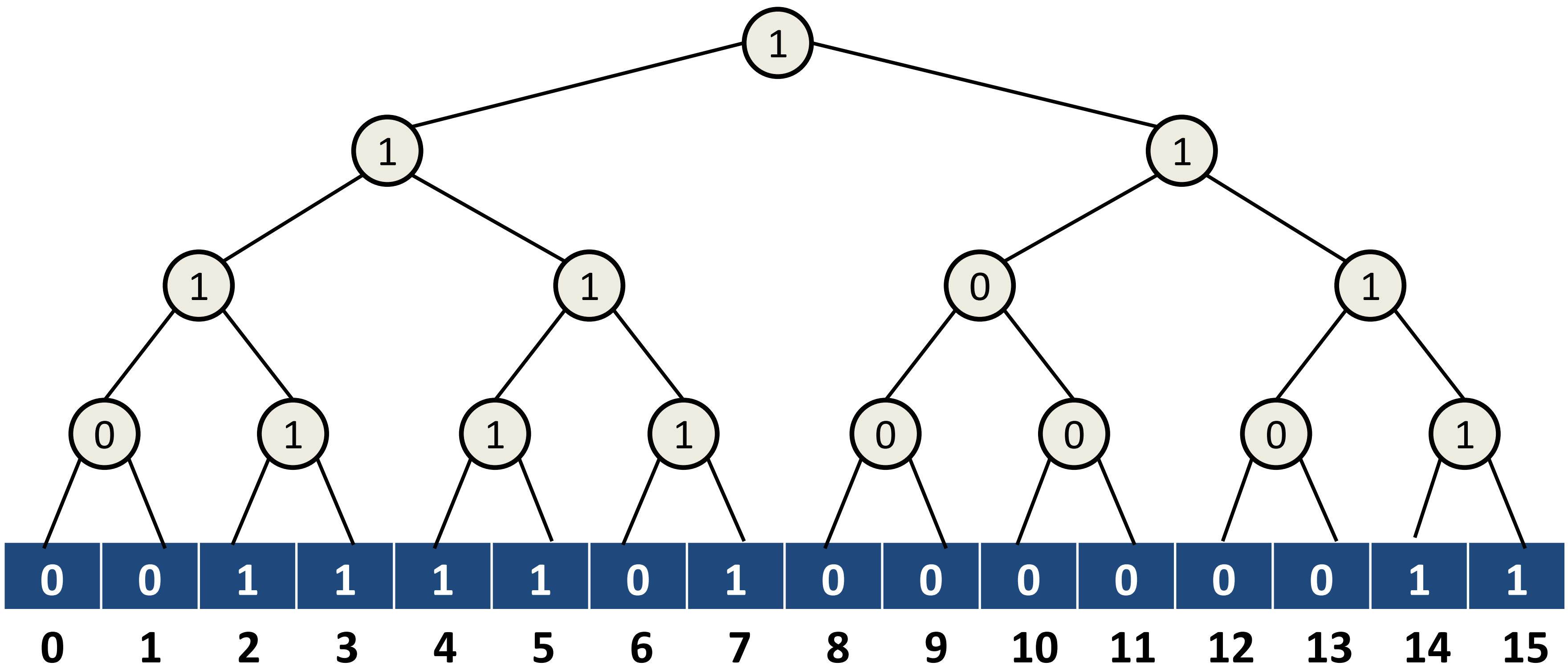
- PREDECESSOR (A, 7):



PREDECESSOR (A, 7) = 5

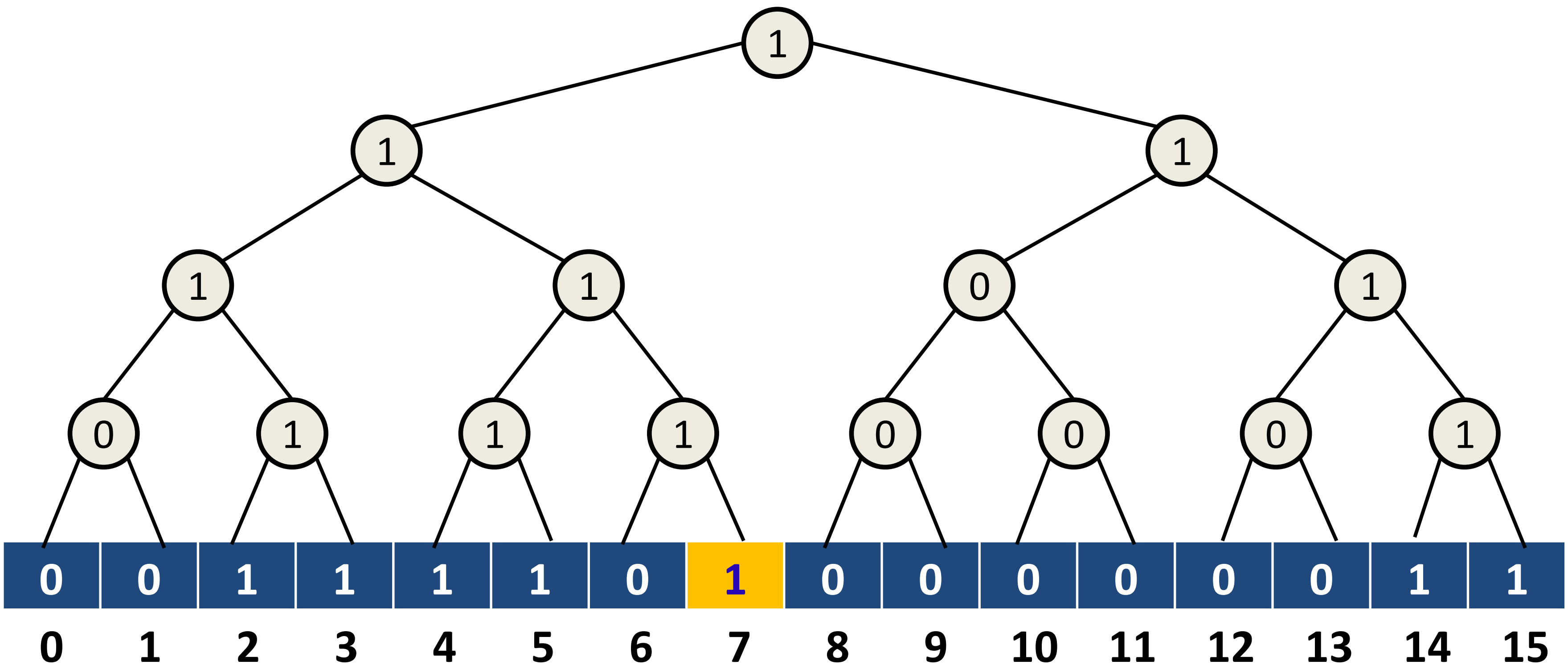
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



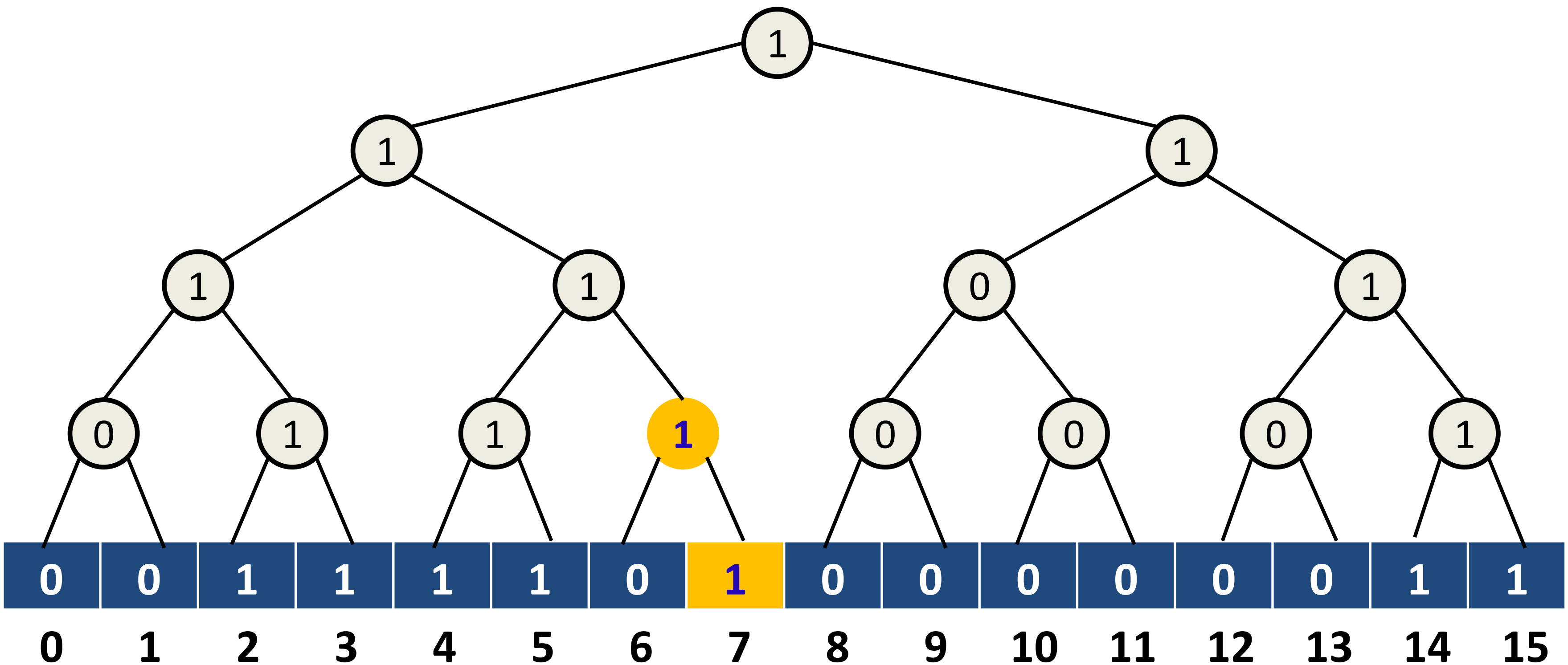
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



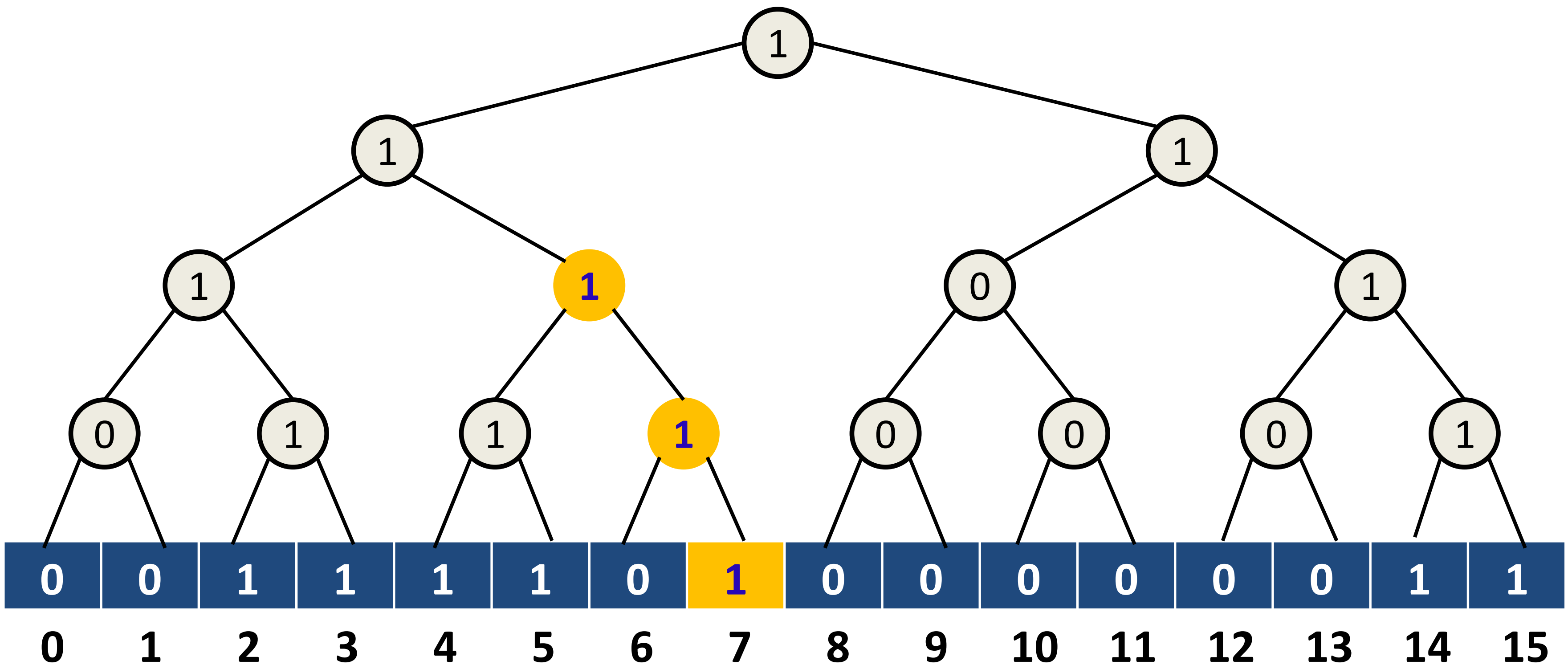
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



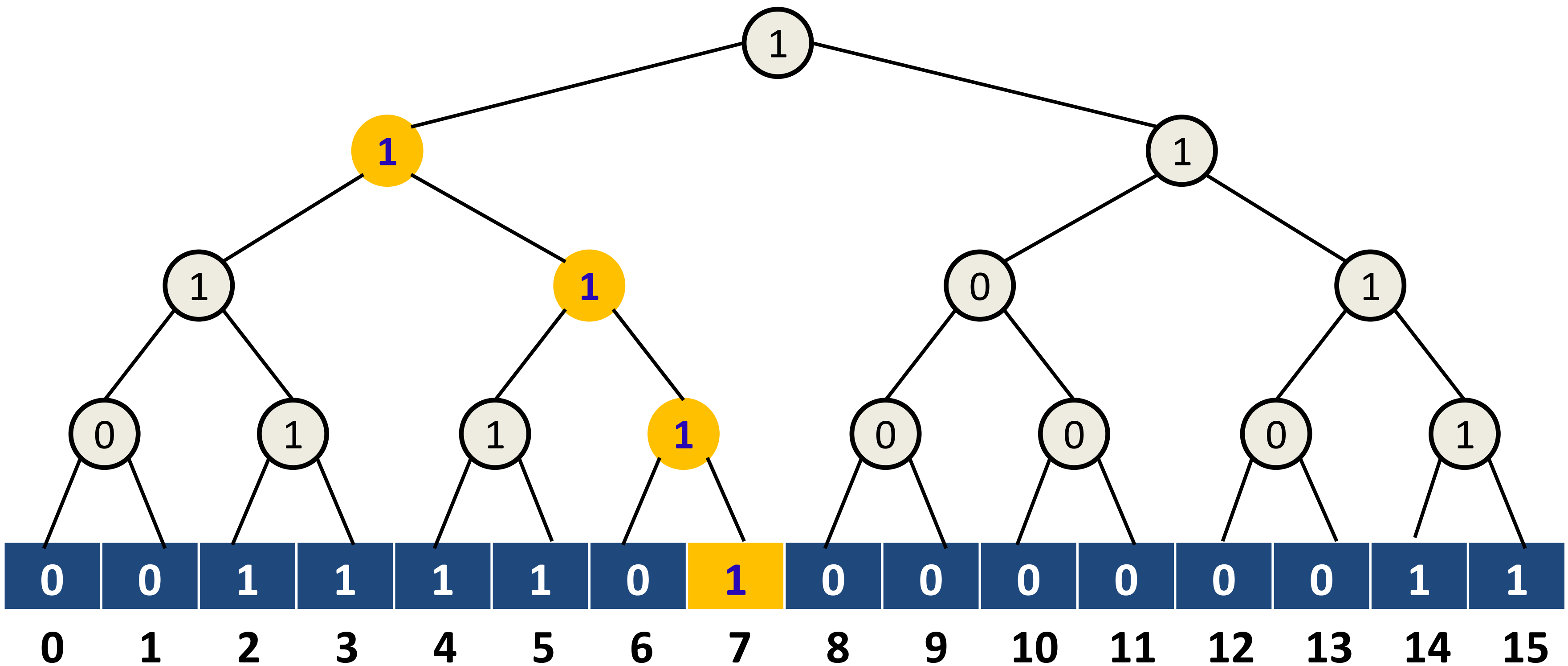
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



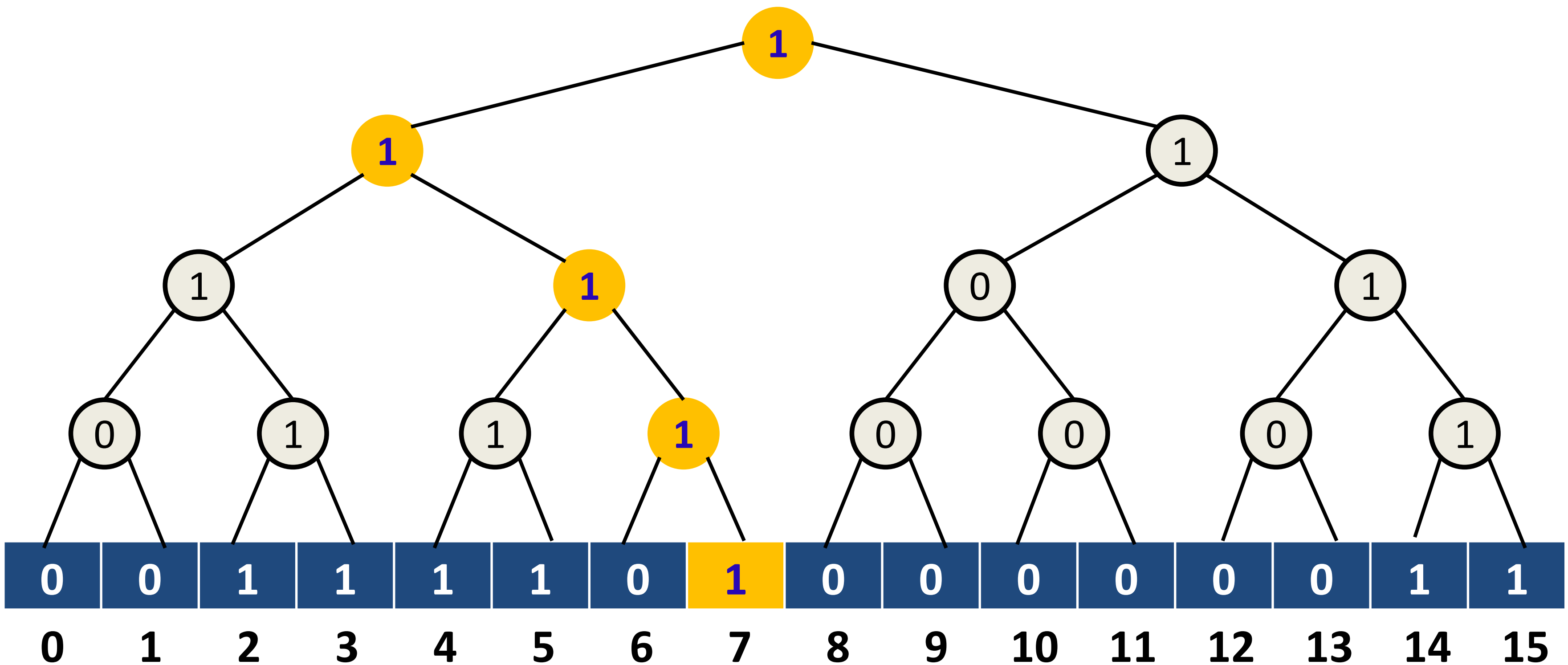
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



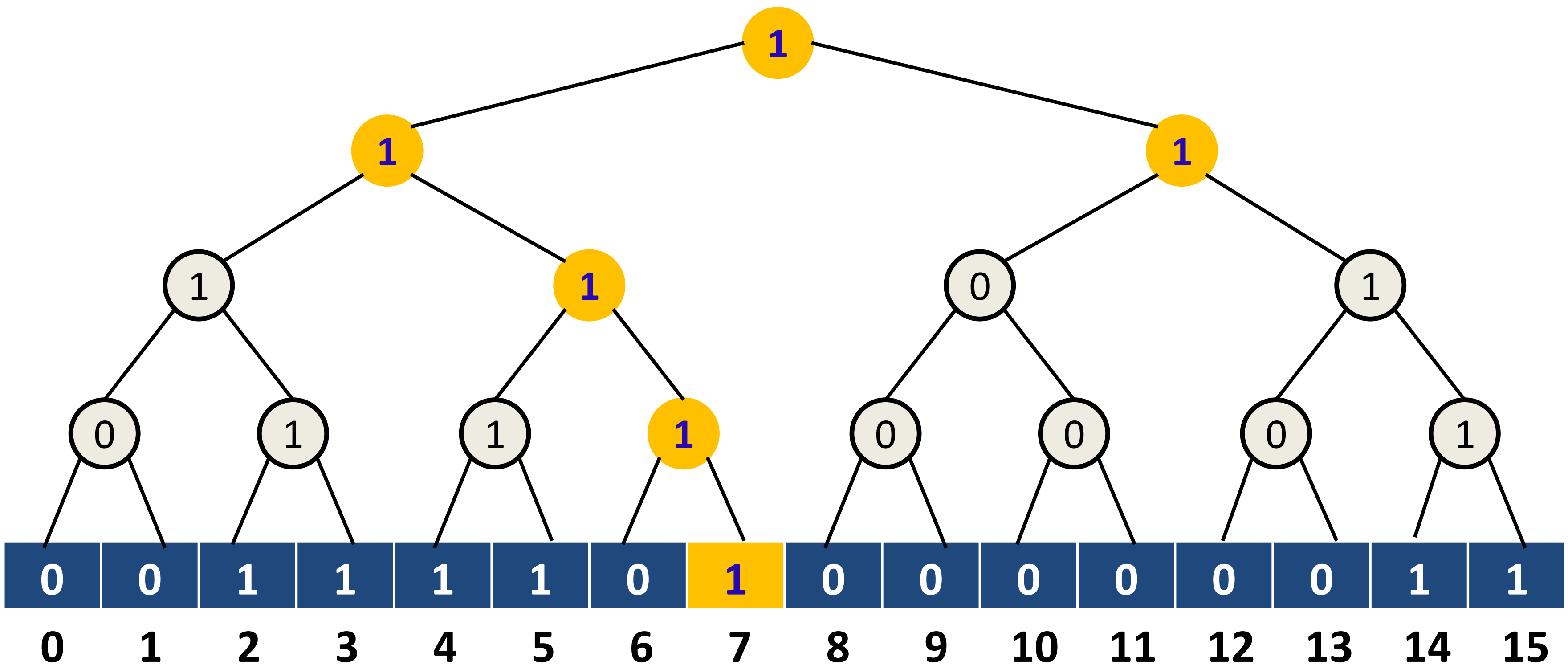
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



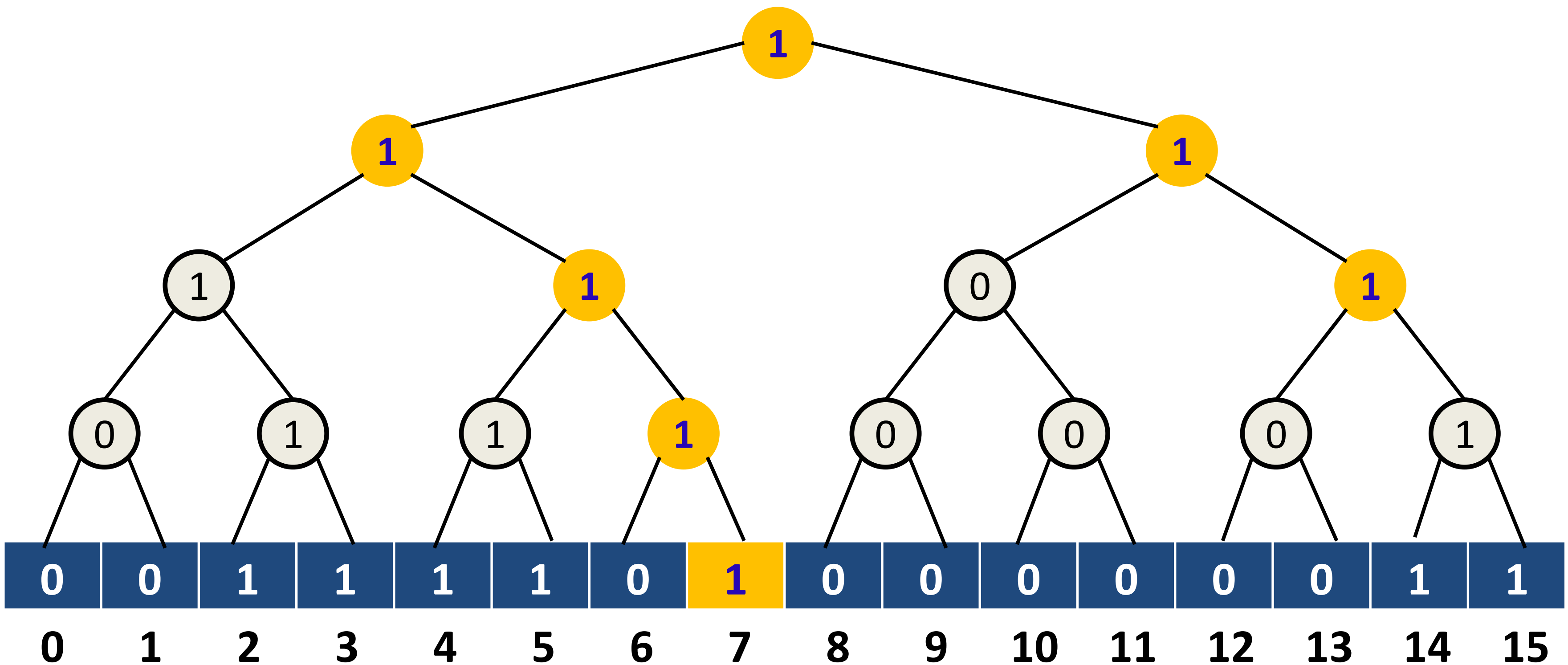
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



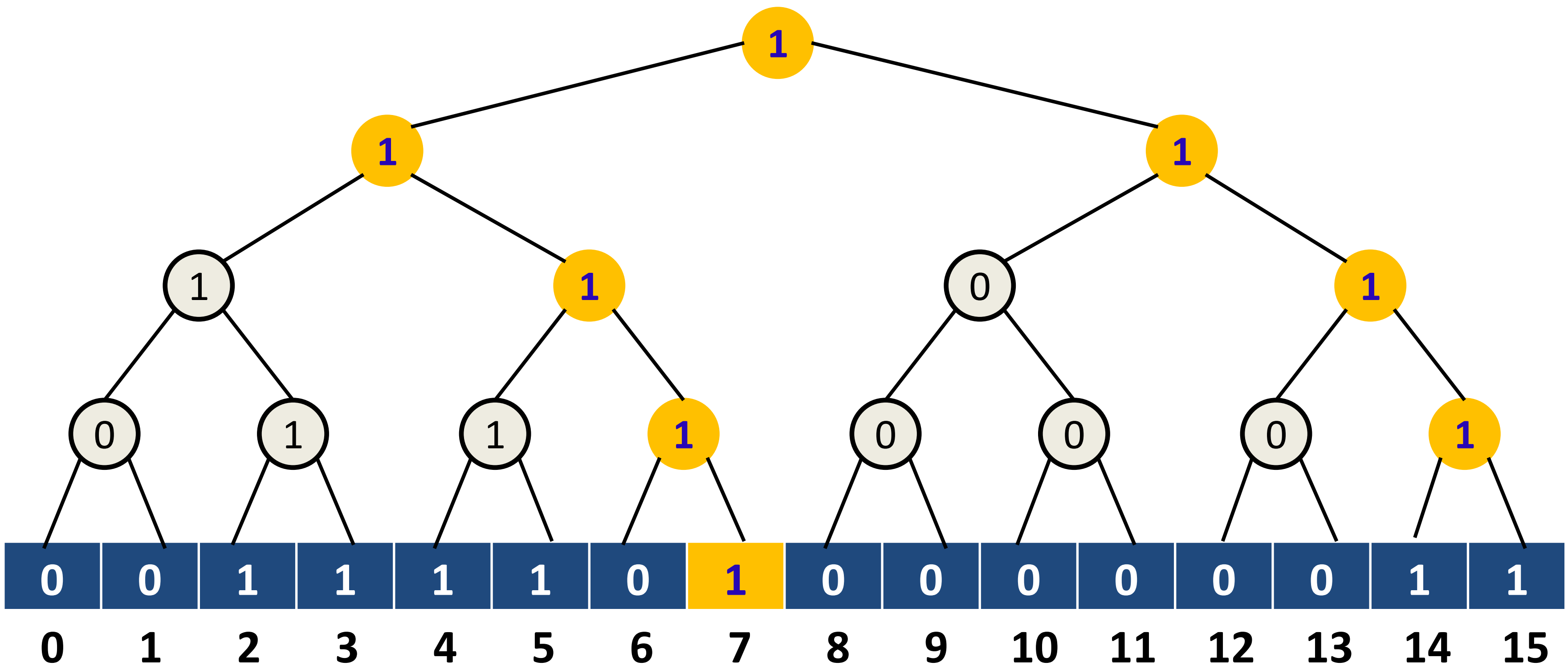
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



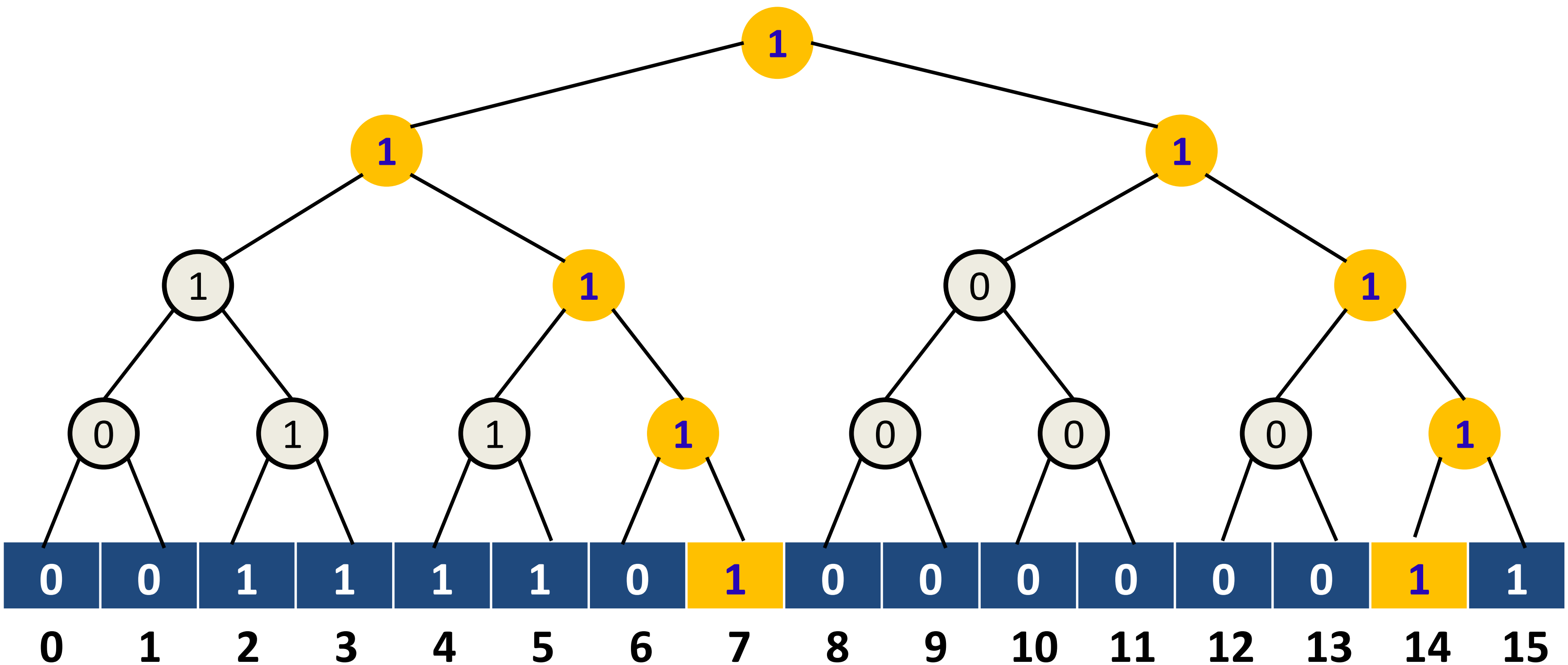
Superimposing a binary tree structure

- SUCCESSOR (A, 7):



Superimposing a binary tree structure

- SUCCESSOR (A, 7):



SUCCESSOR (A, 7) = 14

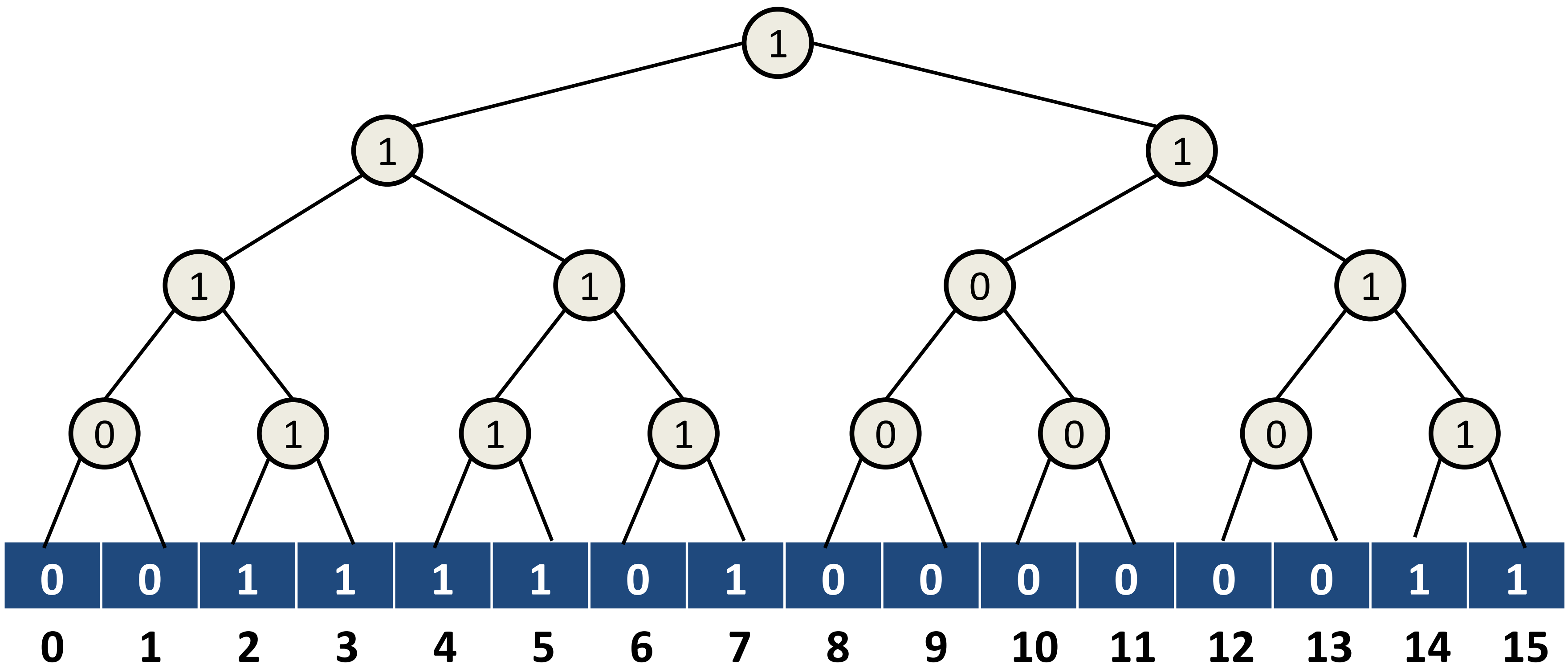
Superimposing a binary tree structure

- INSERTION (A, x) :
  - Start at the leaf indexed by **x** and head up toward the root storing 1 in all nodes belonging to the simple path from **x** to the root.
  - **Time complexity**: O(log u).

- DELETION (A, x) :
  - Start at the leaf indexed by **x** and head up toward the root, recomputing the bit in each internal node on the simple path from **x** to the root as the logical-or of its two children.
  - **Time complexity**: O(log u).

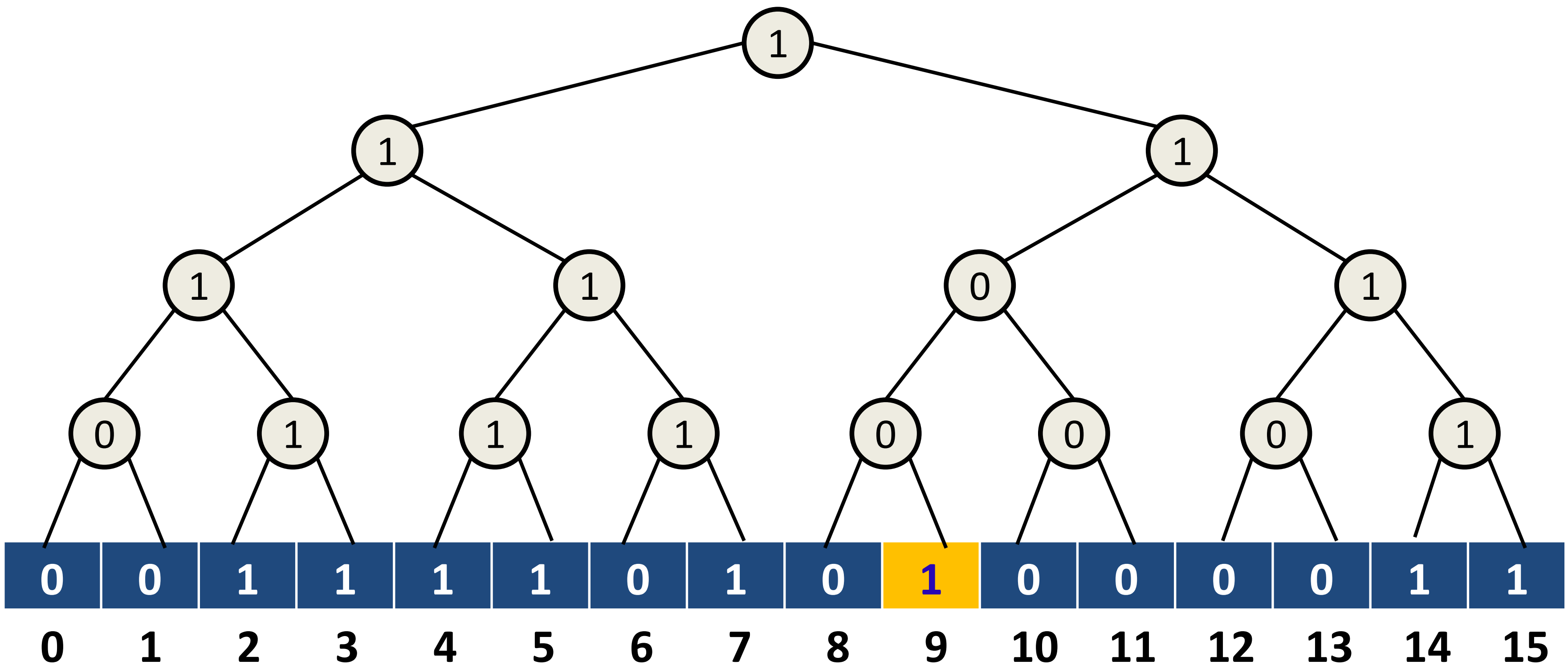
Superimposing a binary tree structure

- INSERTION (A, 9):



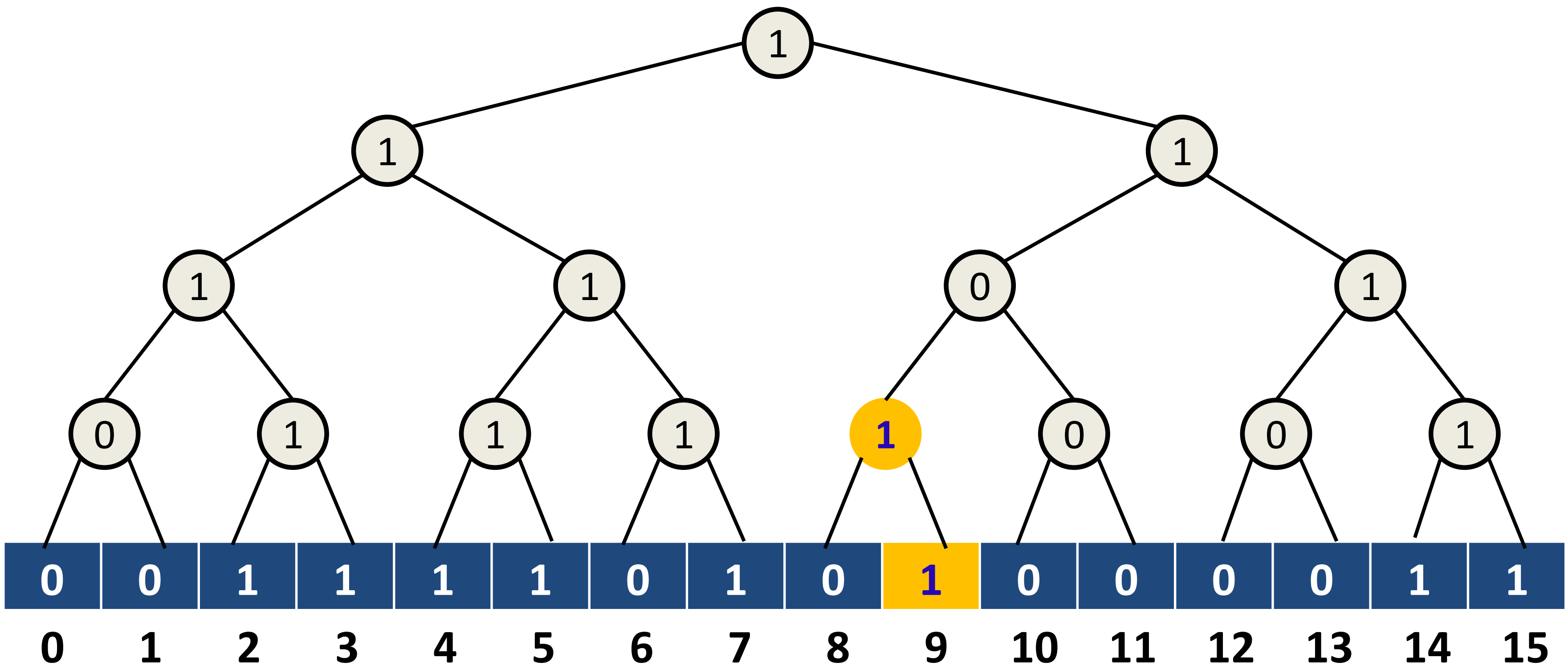
Superimposing a binary tree structure

- INSERTION (A, 9):



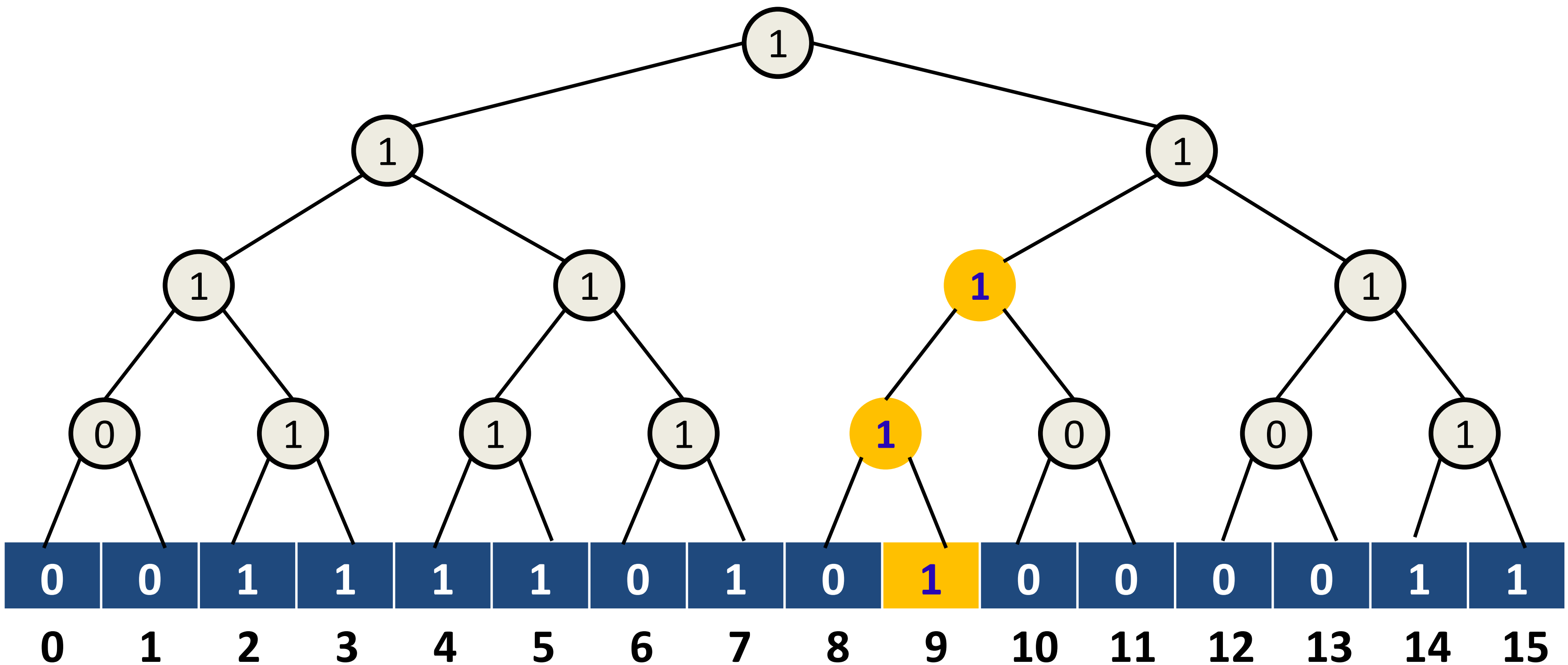
Superimposing a binary tree structure

- INSERTION (A, 9):



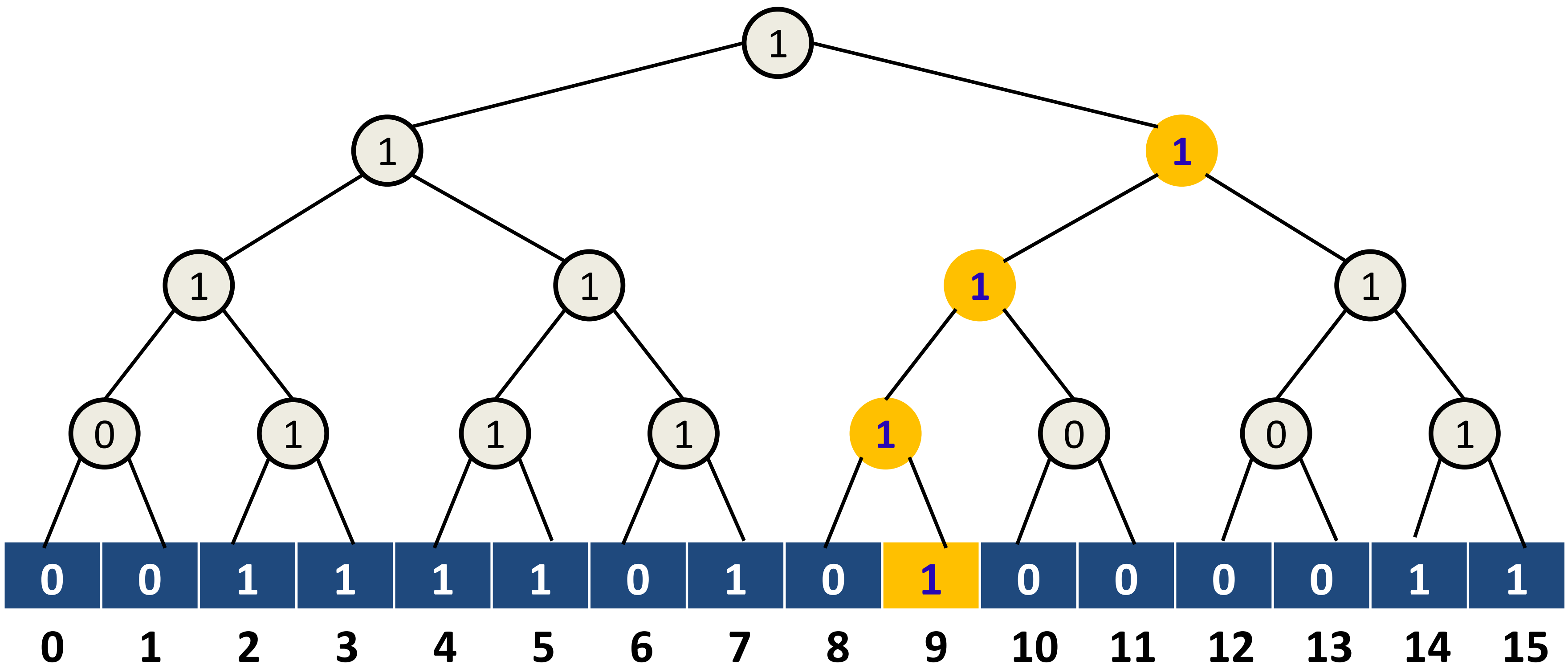
Superimposing a binary tree structure

- INSERTION (A, 9):



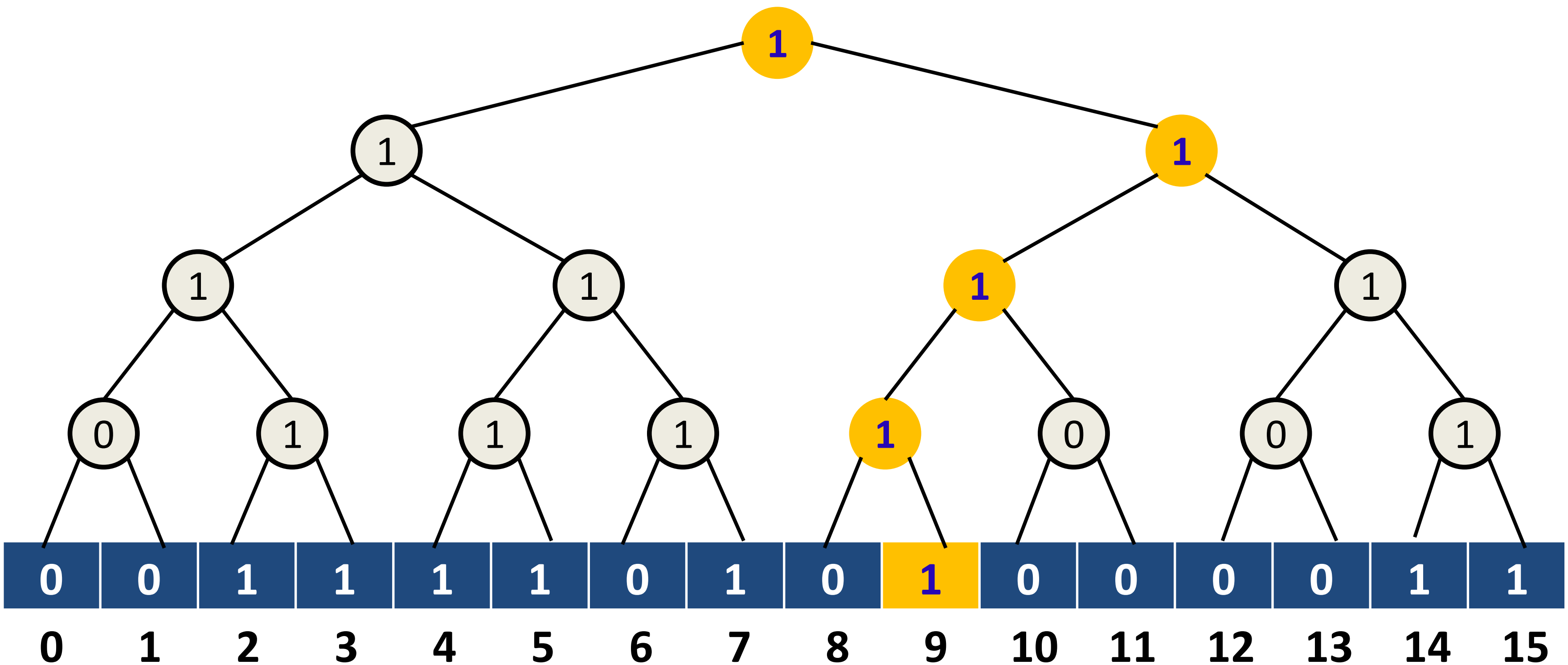
Superimposing a binary tree structure

- INSERTION (A, 9):



Superimposing a binary tree structure

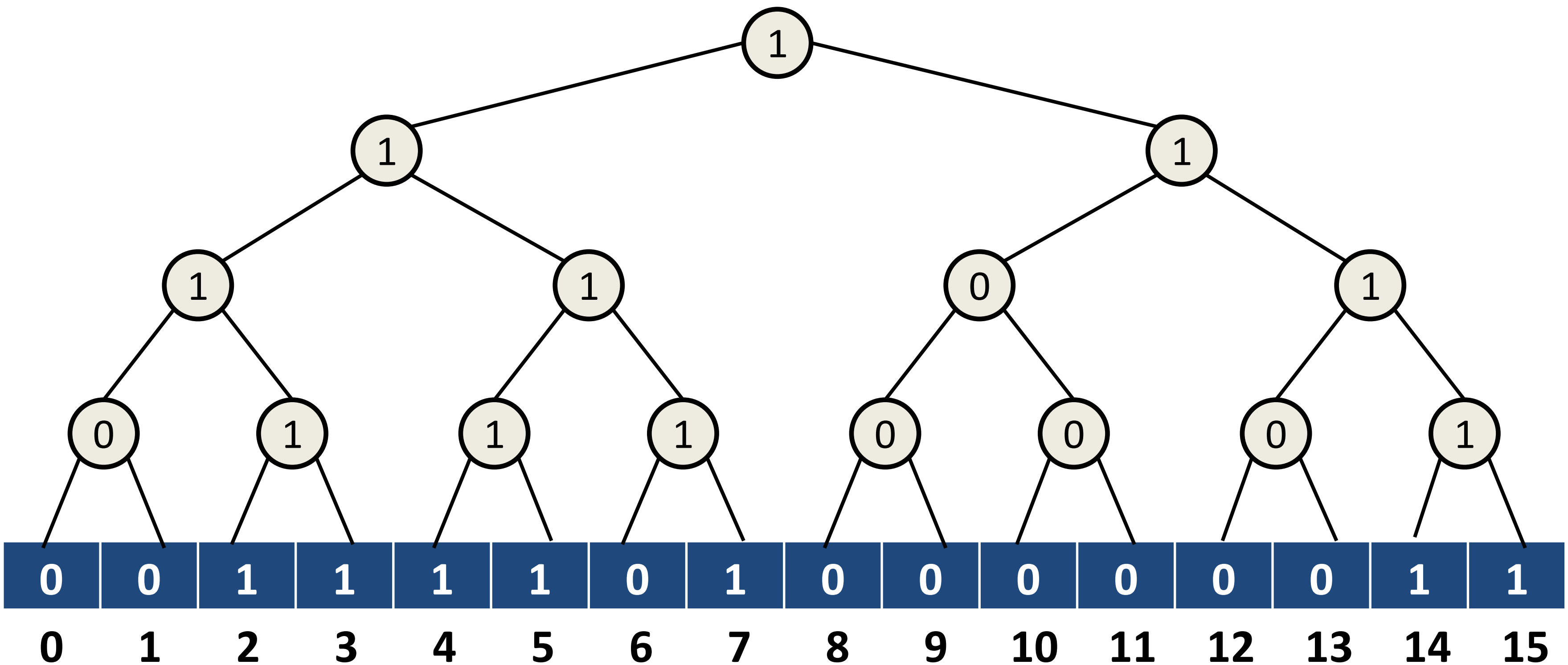
- INSERTION (A, 9):



INSERTION COMPLETED!

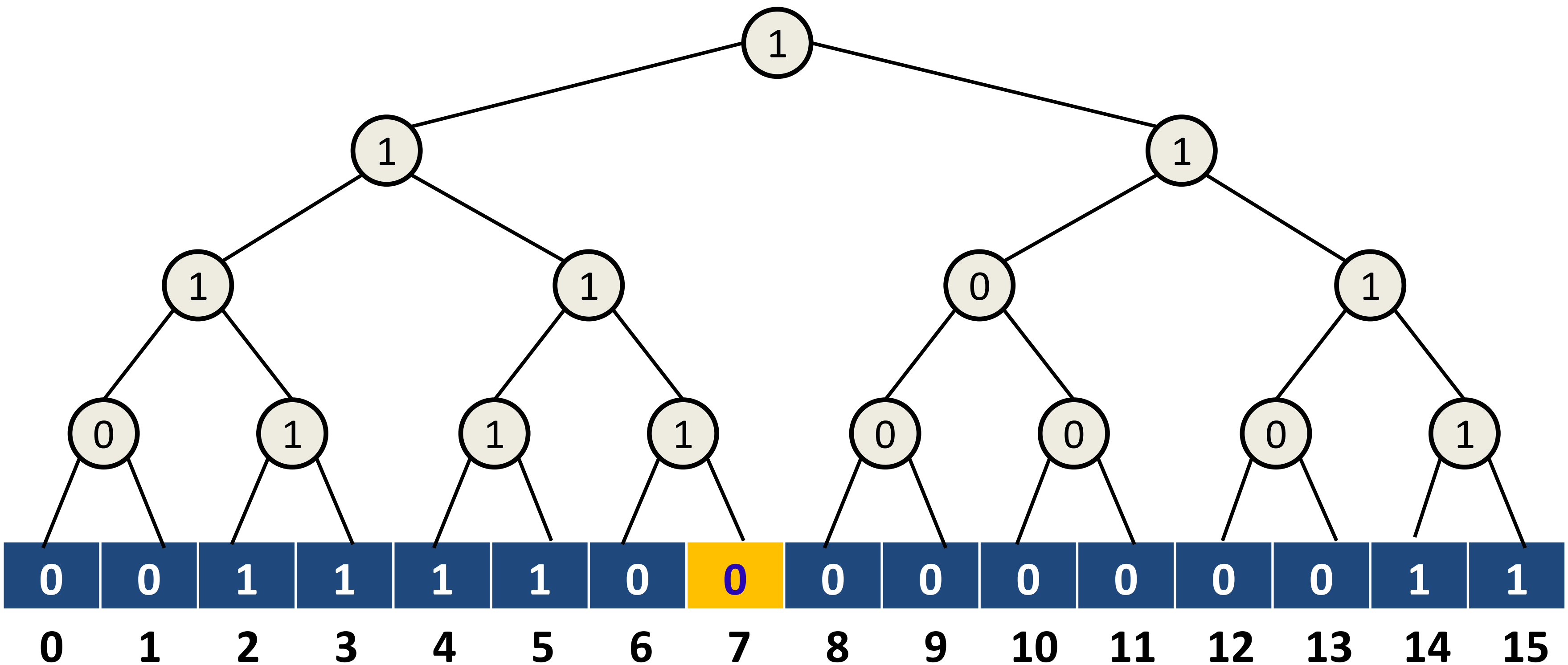
Superimposing a binary tree structure

- DELETION (A, 7):



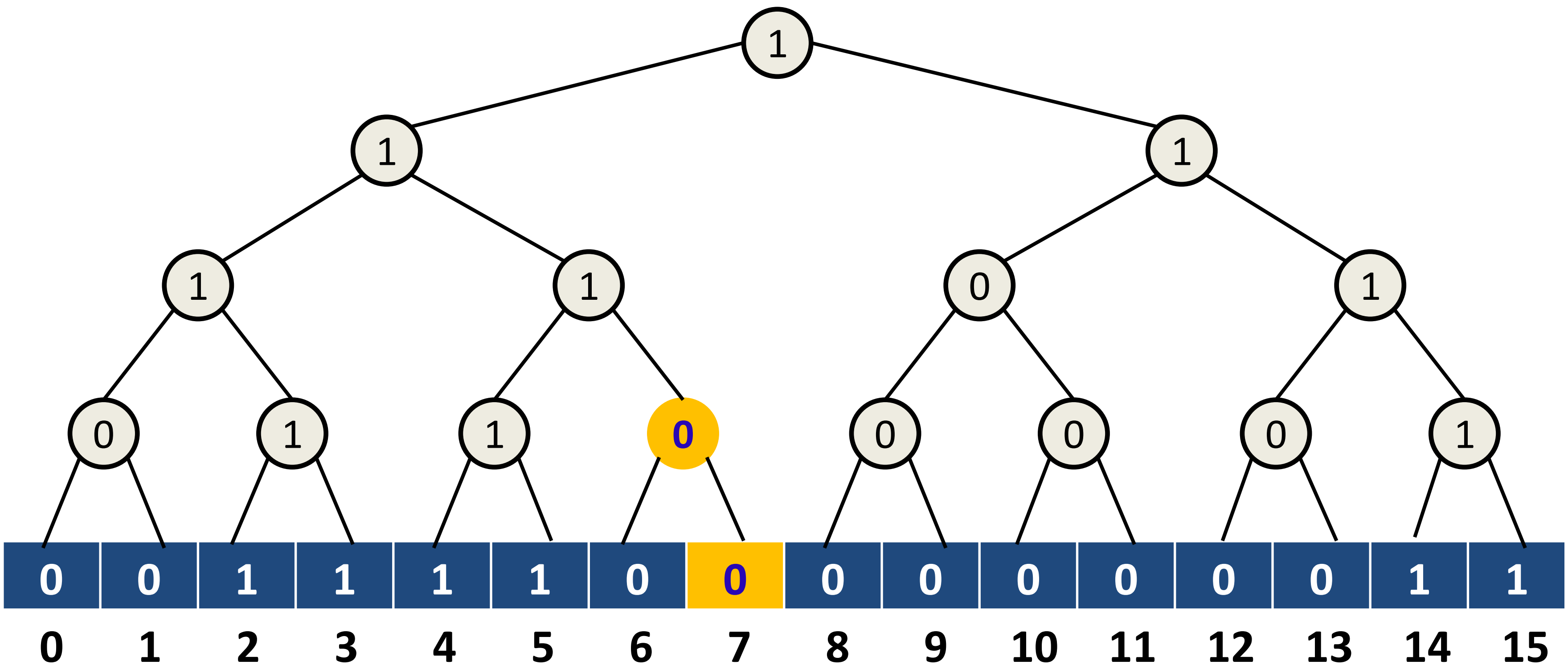
Superimposing a binary tree structure

- DELETION (A, 7):



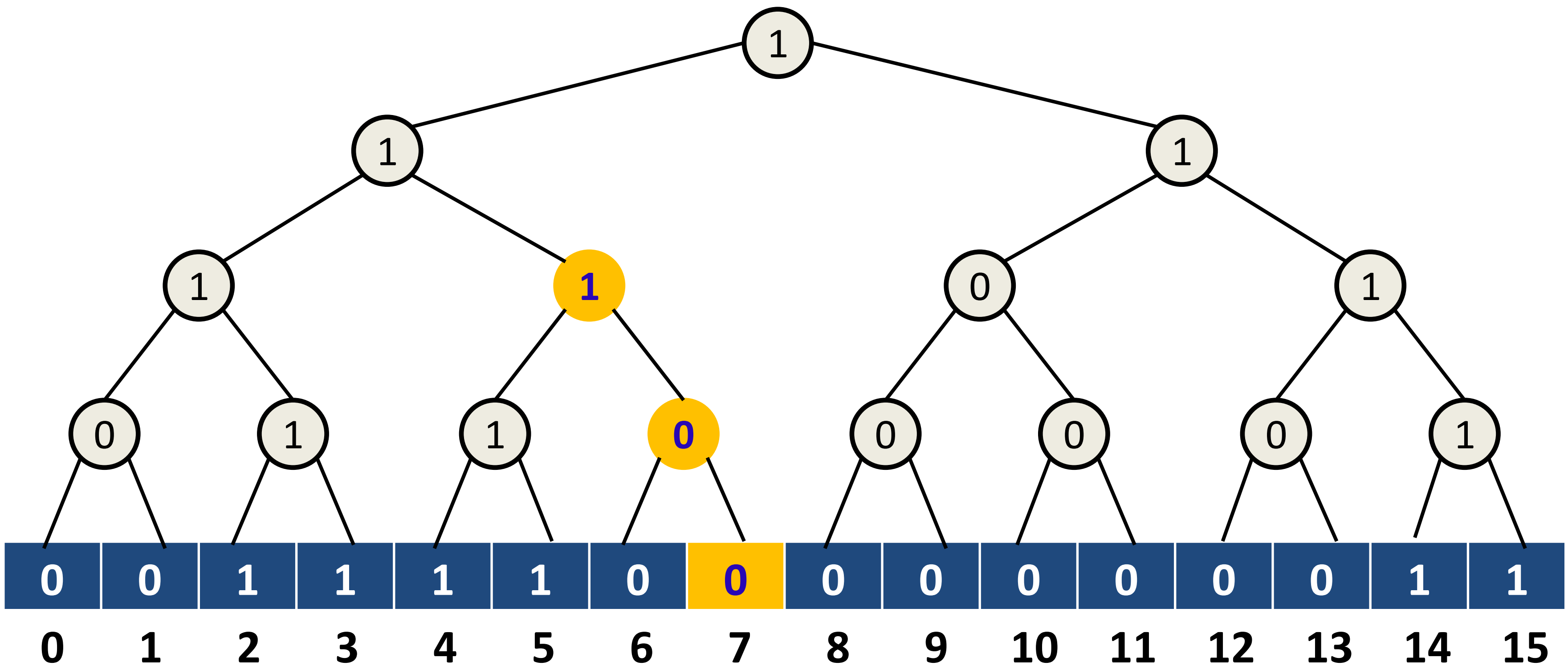
Superimposing a binary tree structure

- DELETION (A, 7):



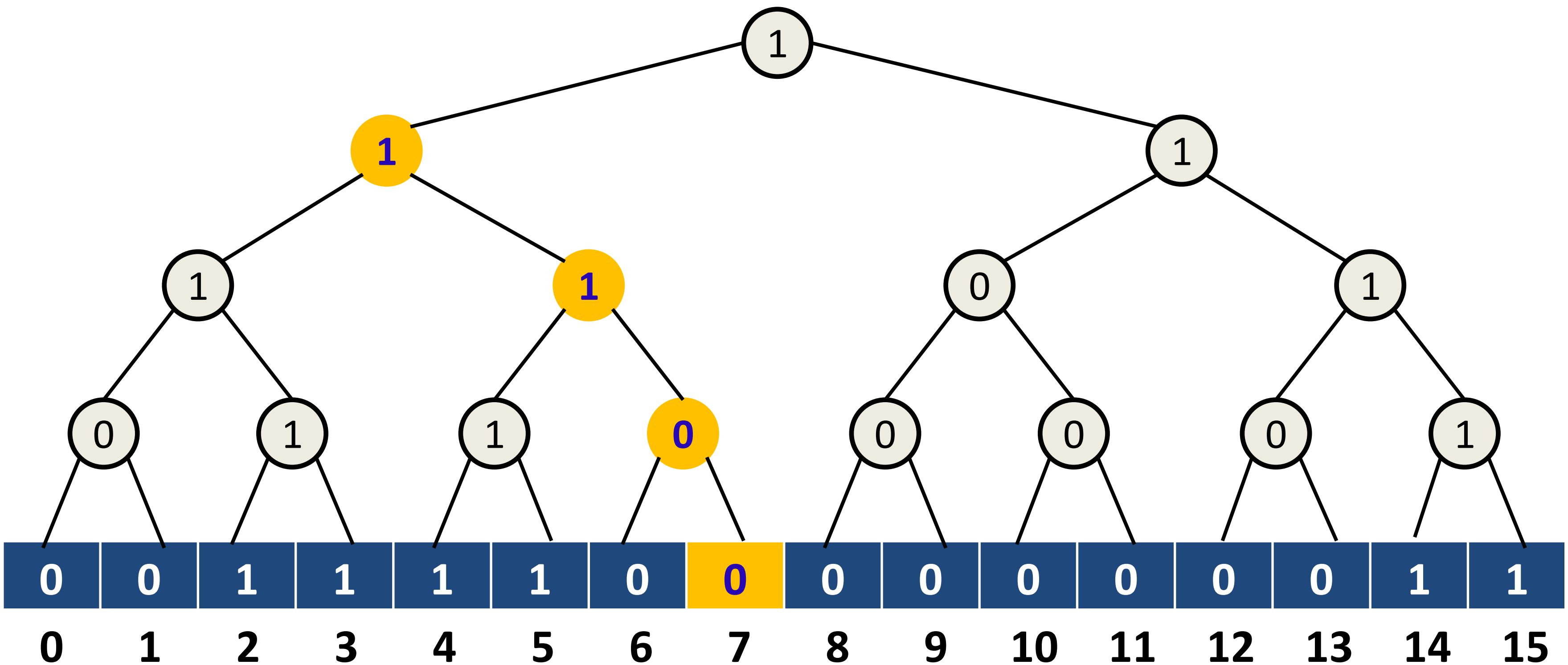
Superimposing a binary tree structure

- DELETION (A, 7):



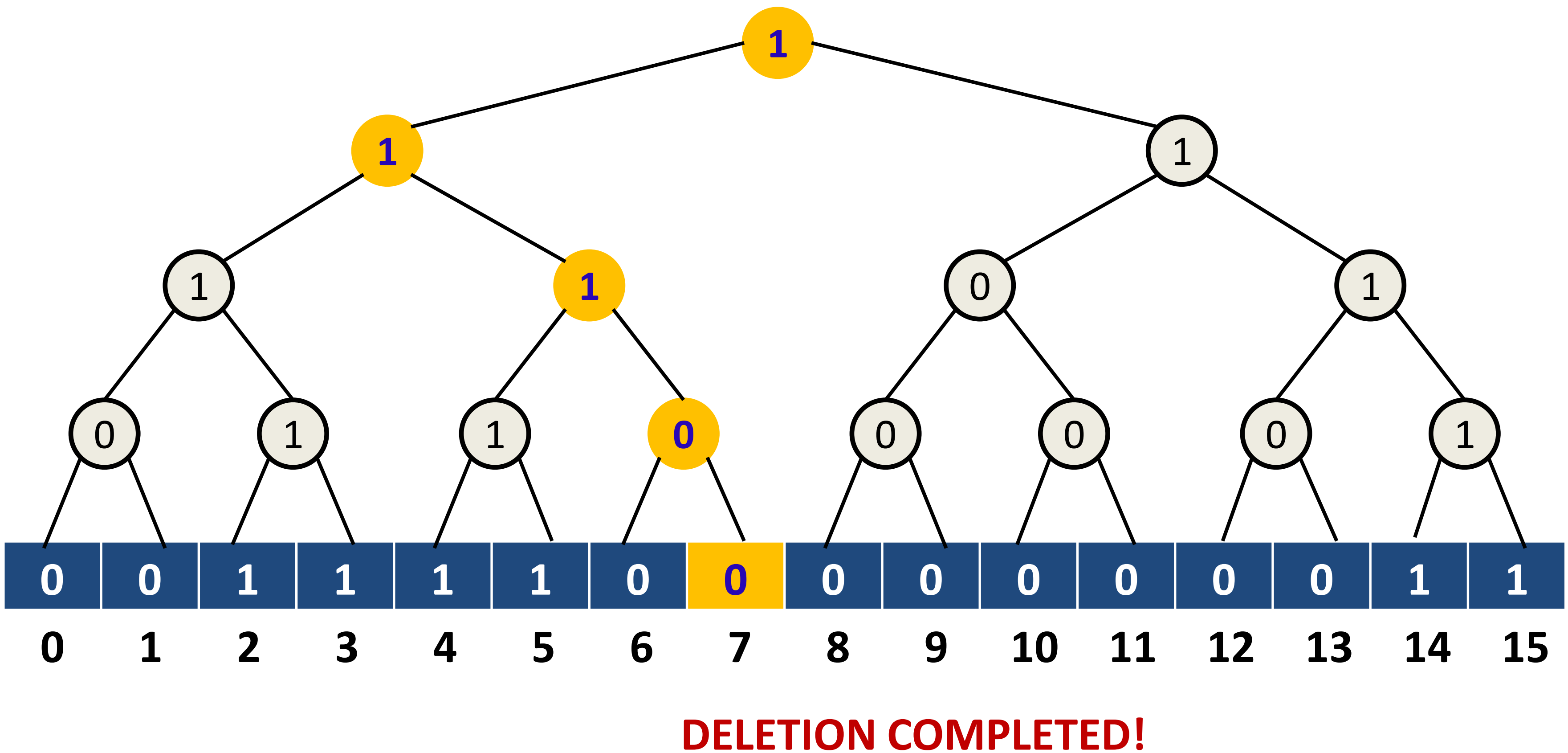
Superimposing a binary tree structure

- DELETION (A, 7):



Superimposing a binary tree structure

- DELETION (A, 7):



Superimposing a binary tree structure

- This approach is only marginally better than just using a balanced search tree.
- MEMBER operation can be performed in O(1) time, what is better than O(log n) for BST.
- If the number of elements **n** is much smaller than size of the universe **u** than BST would be faster for all the other operations.

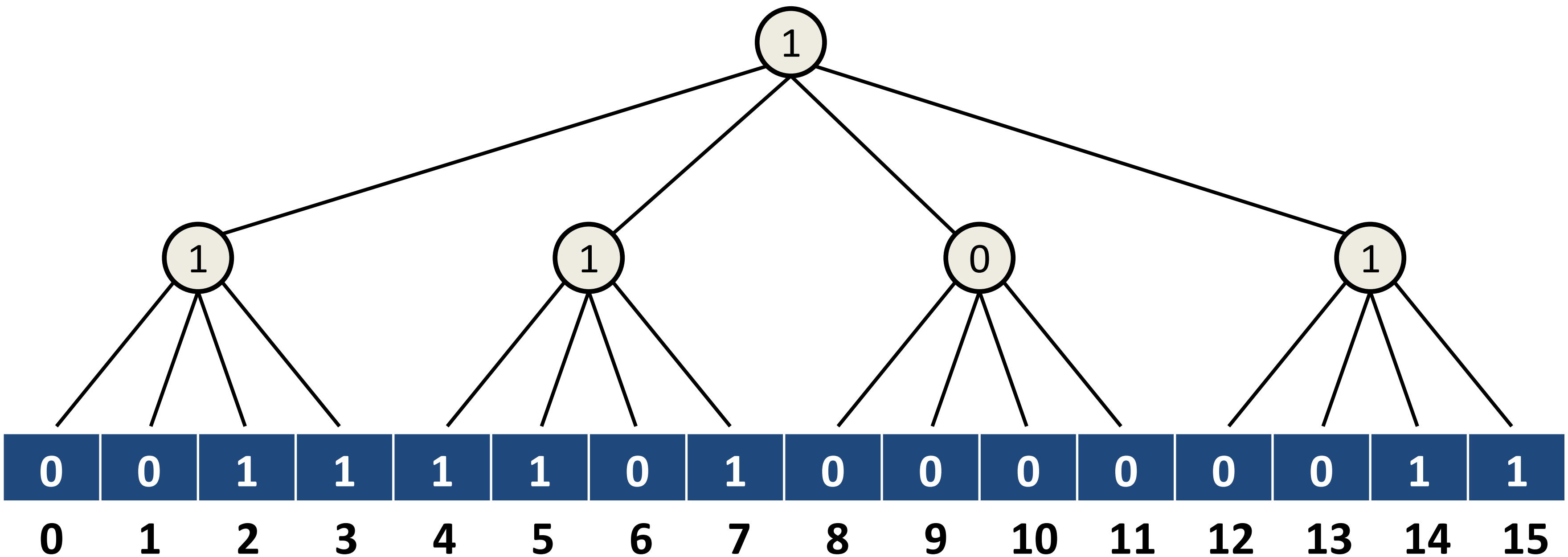
Preliminary approaches

In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- Direct adressing.
- Superimposing a binary tree structure.
- **Superimposing a tree of constant height.**

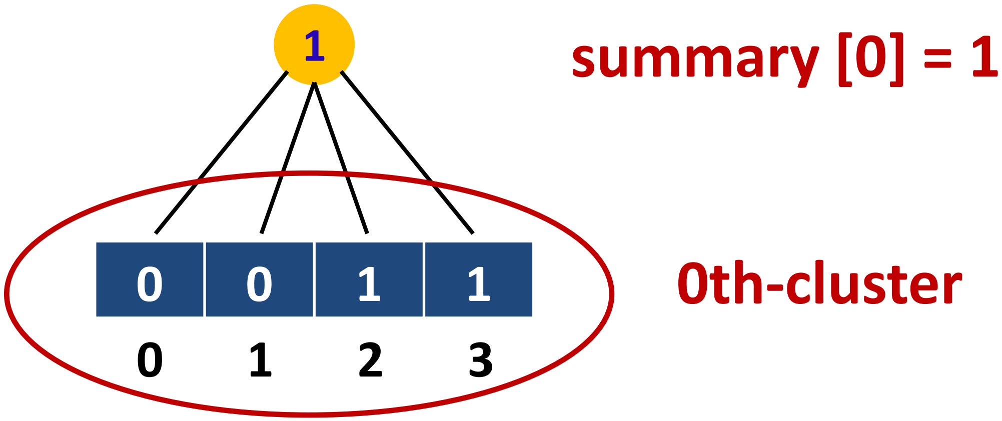
Superimposing a tree of constant height

- Let us assume that $\mathbf{u = 2^{2k}}$ for some natural $\mathbf{k}$.
- Consider that we superimpose a tree of degree $\mathbf{u^{1/2}}$
- The height of the resulting tree would be always 2.



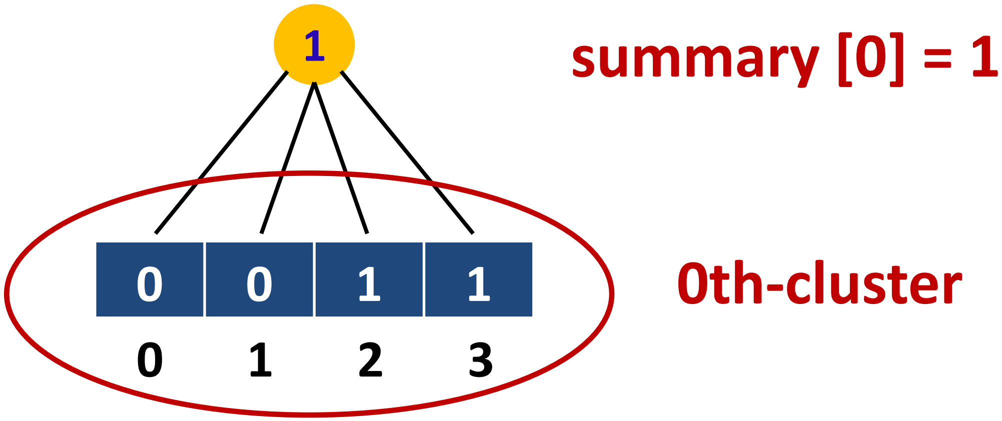
Superimposing a tree of constant height

- Again, each internal node stores the logical-or of the bits within its subtree.
- The $\mathbf{u^{1/2}}$ internal nodes at depth 1 summarize each group (logical-or) of $\mathbf{u^{1/2}}$ values.
- These internal nodes can be seen as an an array **summary[0 .. $\mathbf{u^{1/2}}$ – 1]**



Superimposing a tree of constant height

- Given an index **x**, one can find the cluster that **x** belongs by the expression $\lfloor x/\sqrt{u} \rfloor$
- Operations INSERT and MEMBER can be performed in O(1) time.
- We can use **summary** array to perform all other operations MAX/MIN/PRED/SUC/DEL in $O(u^{1/2})$ time.



Exercise 1

- Modify the data structures introduced so far in order to support duplicate keys.

Exercise 1

- Modify the data structures introduced so far in order to support duplicate keys.

- **Solution**: keep up one integer in the leaves instead of a single bit for counting repetitions. Internal nodes are not affected by this change.

Exercise 2

- Suppose that we superimpose a tree of degree $\mathbf{u^{1/k}}$, instead of $\mathbf{u^{1/k}}$, where $\mathbf{k > 1}$ is an integer constant. What would be the height of such a tree and how long would each of the operations take?

Exercise 2

- Suppose that we superimpose a tree of degree $\mathbf{u^{1/k}}$, instead of $\mathbf{u^{1/k}}$, where $\mathbf{k > 1}$ is an integer constant. What would be the height of such a tree and how long would each of the operations take?

- **Solution**: The height of the tree would be **k** and each operation would take $\mathbf{k \cdot u^{1/k}}$

An interesting recurrence

- Since we are interested on achieving running times of **O(log log u)** let us think about how to obtain these bounds.

- The following recurrence will guide our search for a data structure:

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

An interesting recurrence

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

$$m = \lg u \Rightarrow 2^m = u$$

$$\Rightarrow T(u) = T\left(2^m\right)$$

$$\Rightarrow T\left(2^m\right) = T\left(2^{m/2}\right) + O(1)$$

$$S(m) = T\left(2^m\right)$$

$$\Rightarrow S(m) = S(m/2) + O(1)$$

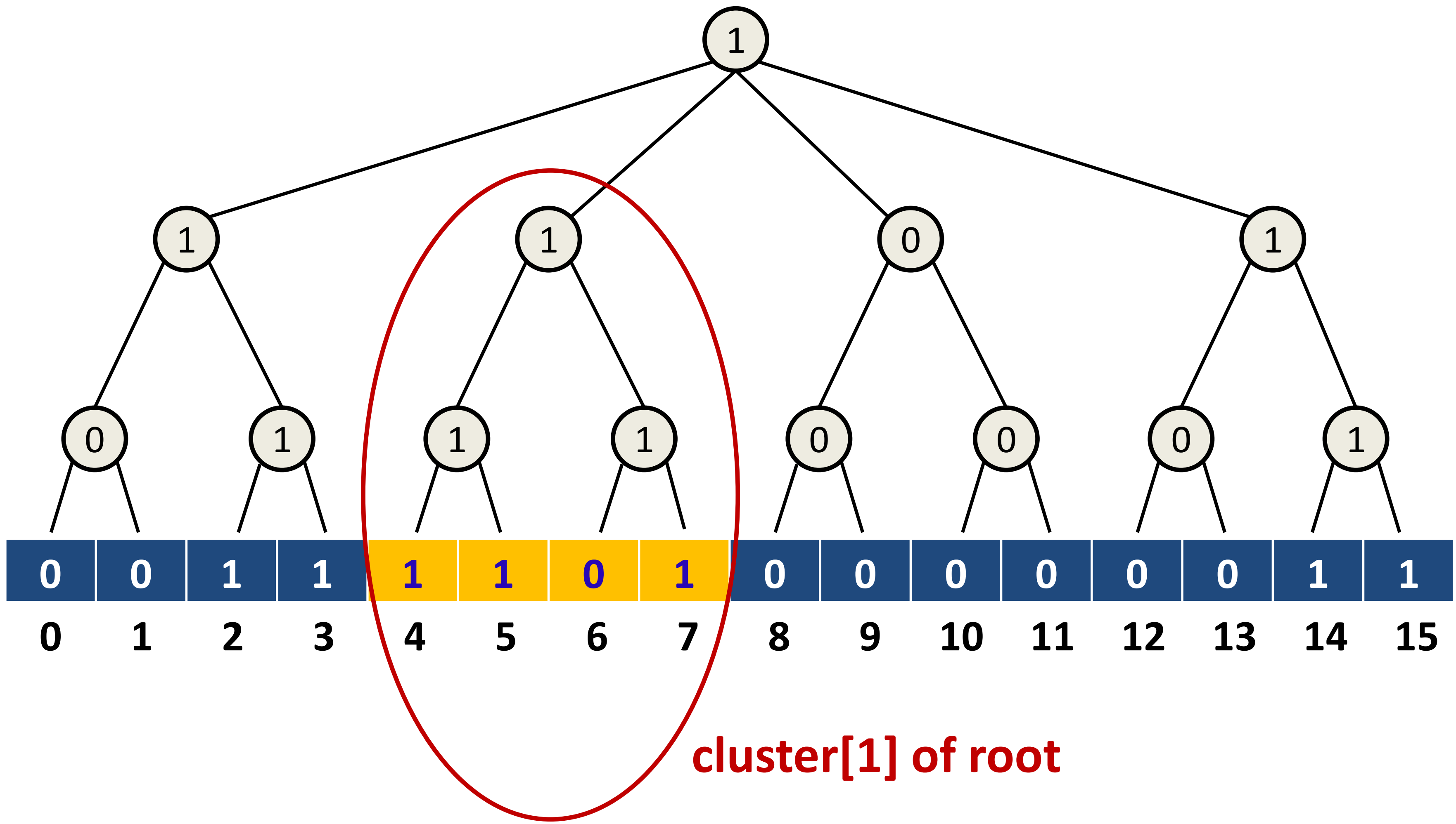
$$\Rightarrow S(m) = \lg m$$

$$\Rightarrow T(u) = T\left(2^m\right) = S(m) = \lg m = \lg \lg u$$

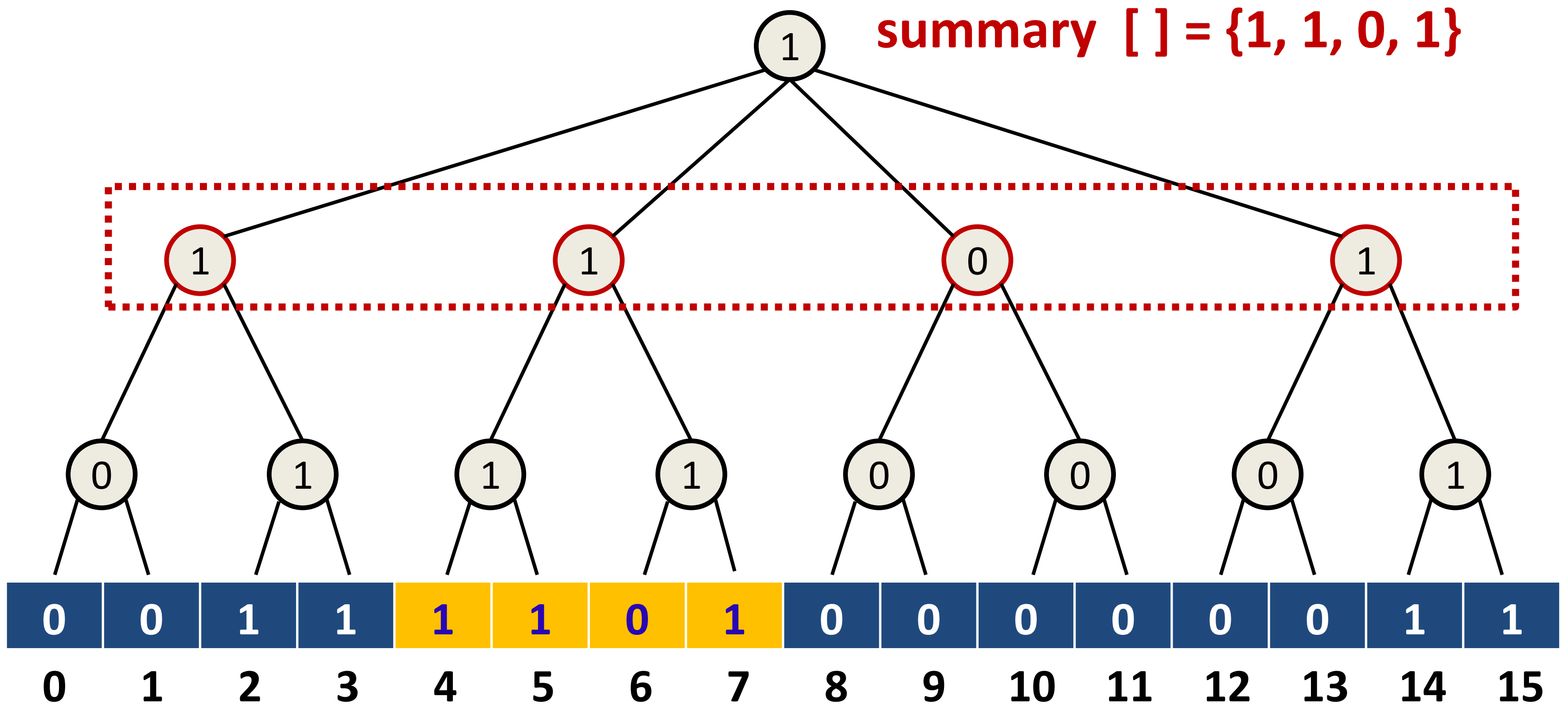
Prototype of van Emde Boas

- The main idea is superimposing a tree of variable degree $\mathbf{u}^{1/k}$.
- Starting with an universe of size **u** we make structures of $\mathbf{u}^{1/2}$ items, which themselves contain structures of $\mathbf{u}^{1/4}$ items, which hold structures of $\mathbf{u}^{1/8}$ items...
- This subdivision process stops when there are only 2 items.
- For simplicity we assume that $u = 2^{2^k}$ for some integer **k**.

Prototype of van Emde Boas



Prototype of van Emde Boas



Prototype of van Emde Boas

Let $u = 2^{2^k}$ be the size of the universe for some integer $k \geq 0$.

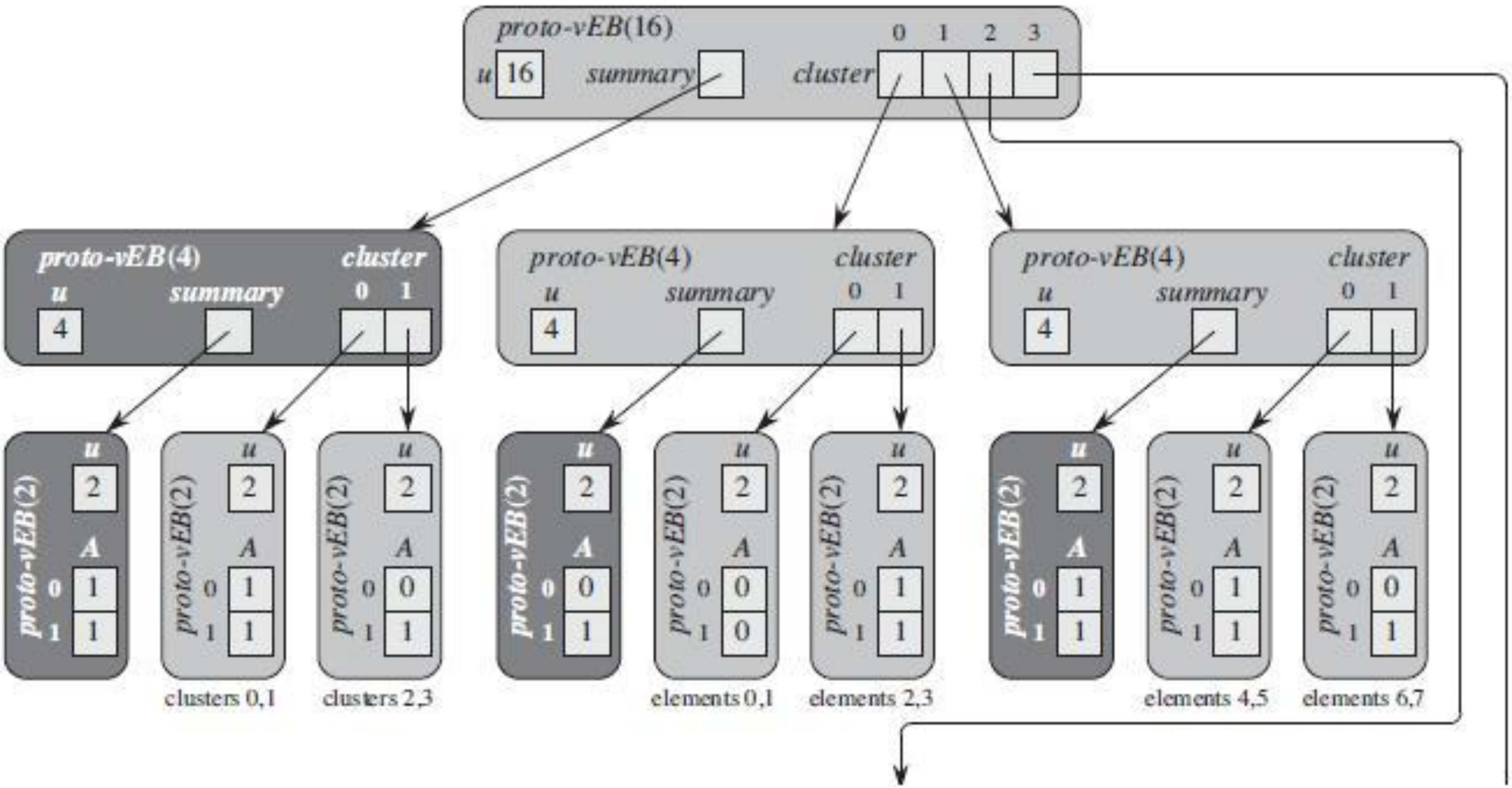
- If **u = 2** then it is the basis size, and the **proto-vEB(u)** contains the array A[0..1] of two bits.

- Otherwise the **proto-vEB(u)** structure is defined recursively as follows:

  - It contains a pointer named **summary** to a **proto-vEB($u^{1/2}$)** structure.

  - Its contains an array **cluster[0 .. $u^{1/2}$ - 1]** of $u^{1/2}$ pointers, each to a **proto-vEB($u^{1/2}$)** structure**.**

Prototype of van Emde Boas



Clusters and positions

- A given value **x** resides in **cluster number** $\lfloor x / \sqrt{u} \rfloor$

- Since that **x** is a binary integer of **lg u** bits, its cluster number is given by the **most significant** (lg u)/2 bits of **x**.

- The **position** in which **x** appears in its cluster is given by the expression $x \bmod \sqrt{u}$

- As a consequence, the position of **x** inside its cluster is given by the **least significant** (lg u)/2 bits of **x**

Clusters and positions

The following functions will be necessary to explore the **proto-vEB** structure:

- **high(x) =** $\lfloor x / \sqrt{u} \rfloor$
- **low(x) =** $x \bmod \sqrt{u}$
- **index(x, y) =** $x\sqrt{u} + y$

Member operation on proto-vEB

```
PROTO-vEB-MEMBER (V, x)
 if V.u == 2
   return V.A[x]
 else
   return PROTO-vEB-MEMBER (V.cluster[high(x)], low(x))
```

Member operation on proto-vEB

- **Running time:**

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$
$$\Rightarrow T(u) = O(\lg\lg u)$$

Finding the minimum of proto-vEB

```
PROTO-vEB-MINIMUM (V)
    if V.u == 2
        if V.A[0] == 1
            return 0
        elseif V.A[1] == 1
            return 1
        else return NIL
    else min-cluster = PROTO-vEB-MINIMUM (V.summary)
        if min-cluster == NIL
            return NIL
        else offset = PROTO-vEB-MINIMUM (V.cluster[min-cluster])
            return index(min-cluster, offset)
```

Finding the minimum of proto-vEB

- **Running time:**

$$T(u) = 2T\left(\sqrt{u}\right) + O(1)$$

$$\Rightarrow T\left(2^{m}\right) = 2T\left(2^{m/2}\right) + O(1)$$

$$\Rightarrow S\left(m\right) = 2S\left(m/2\right) + O(1)$$

$$\Rightarrow S\left(m\right) = \Theta(m)$$

$$\Rightarrow T(u) = T\left(2^{m}\right) = S\left(m\right) = \Theta(m) = \Theta(\lg u)$$

Successor operation on proto-vEB

```
PROTO-vEB-SUCCESSOR (V, x)
    if V.u == 2
        if x == 0  and V.A[1] == 1
            return 1
        else return NIL
    else  offset = PROTO-vEB-SUCCESSOR (V.cluster[high(x)], low(x))
        if offset ≠ NIL
            return index (high(x), offset)
        else
            succ-cluster = PROTO-vEB-SUCCESSOR (V.summary, high(x))
            if succ-cluster == NIL
                return NIL
            else offset = PROTO-vEB-MINIMUM (V.cluster[succ-cluster])
                return index(succ-cluster, offset)
```

Successor operation on proto-vEB

- **Running time:**

$$
\begin{aligned}
&T(u) = 2T\left(\sqrt{u}\right) + O(\lg \sqrt{u}) \\
&\Rightarrow T\left(2^{m}\right) = 2T\left(2^{m/2}\right) + O(m) \\
&\Rightarrow S\left(m\right) = 2S\left(m/2\right) + O(m) \\
&\Rightarrow S\left(m\right) = \Theta(m \lg m) \\
&\Rightarrow T(u) = T\left(2^{m}\right) = S\left(m\right) \\
&\Rightarrow T(u) = \Theta(m \lg m) = \Theta(\lg u \lg \lg u)
\end{aligned}
$$

Insertion on proto-vEB

```
PROTO-vEB-INSERT (V, x)
    if V.u == 2
        V.A[x] = 1
    else
        PROTO-vEB-INSERT (V.cluster[high(x)], low(x))
        PROTO-vEB-INSERT (V.summary, high(x))
```

Insertion on proto-vEB

- **Running time:**

$$T(u) = 2T\left(\sqrt{u}\right) + O(1)$$
$$\Rightarrow T\left(2^m\right) = 2T\left(2^{m/2}\right) + O(1)$$
$$\Rightarrow S\left(m\right) = 2S\left(m/2\right) + O(1)$$
$$\Rightarrow S\left(m\right) = \Theta(m)$$
$$\Rightarrow T(u) = T\left(2^m\right) = S\left(m\right) = \Theta(m) = \Theta(\lg u)$$

The “REAL” van Emde Boas Tree

- Proto-vEB structure was close to what is needed to achieve O(lg lg n) running times.

- It requires too many (2 or more) recursions in most of the operations (time complexity bounds allow only one recursion).

- The “REAL” van Emde Boas tree stores some extra information in order to make at most one recursive call on each of the operations.

The universe size $u = 2^k$

- The universe size $u = 2^{2^k}$ in **proto-vEB** was very restrictive :
  - If **k = 1** then **u = 4**
  - If **k = 2** then **u = 16**
  - If **k = 3** then **u = 256**
  - If **k = 4** then **u = 65536**
  - If **k = 5** then **u = 4294967296**
  - If **k = 6** then **u = 18446744073709551616**

- In the **proto-vEB**, any $u \neq 2^{2^k}$ must be transformed into an **u'** such that **u' = Ω(u²)**

- The **real-vEB** allow the universe size **u** to be **any exact power of two**. In other words, $u = 2^k$ for an integer **k > 0**

The universe size $u = 2^k$

- Since that the universe size is any exact power of two, $\mathbf{u^{1/2}}$ is not necessarily an integer.
- However, this change affects the calculation of functions **high**, **low** and **index** that uses most/least significant bits**.**
  - **high(x) =** $\left\lfloor x / \sqrt{u} \right\rfloor$
  - **low(x) =** $x \bmod \sqrt{u}$
  - **index(x, y) =** $x\sqrt{u} + y$

The universe size u = 2^k

- For convenience, we introduce the concepts of **lower square root** and **upper square root**:
  - **Lower square root**: $\downarrow\sqrt{u} = 2^{\lfloor(\lg u)/2\rfloor}$
  - **Upper square root**: $\uparrow\sqrt{u} = 2^{\lceil(\lg u/2)\rceil}$
- **Examples**:
  - $\downarrow\sqrt{32} = 2^{\lfloor(\lg 32)/2\rfloor} = 2^2 = 4$
  - $\uparrow\sqrt{32} = 2^{\lceil(\lg 32/2)\rceil} = 2^3 = 8$

Redefining high, low and index

- Since that the universe size is any exact power of two, $\mathbf{u^{1/2}}$ is not necessarily an integer.

- However, this change affects the calculation of functions **high**, **low** and **index** that uses most/least significant bits**.**
  - **high(x) =** $\left\lfloor x / \downarrow \sqrt{u} \right\rfloor$
  - **low(x) =** $x \bmod \downarrow \sqrt{u}$
  - **index(x) =** $x \downarrow \sqrt{u} + y$

Redefining high, low and index

- **Example**

  If u = 36 and x = 23, then:

  - **high(23) =** $\left\lfloor 23 / \downarrow\sqrt{32} \right\rfloor = \lfloor 23/4 \rfloor = 5$
  - **low(23) =** $23 \bmod \downarrow\sqrt{32} = 23 \bmod 4 = 3$
  - **index(5, 3) =** $5 \downarrow\sqrt{32} + 3 = 5 \cdot 4 + 3 = 23$

The "REAL" van Emde Boas Tree

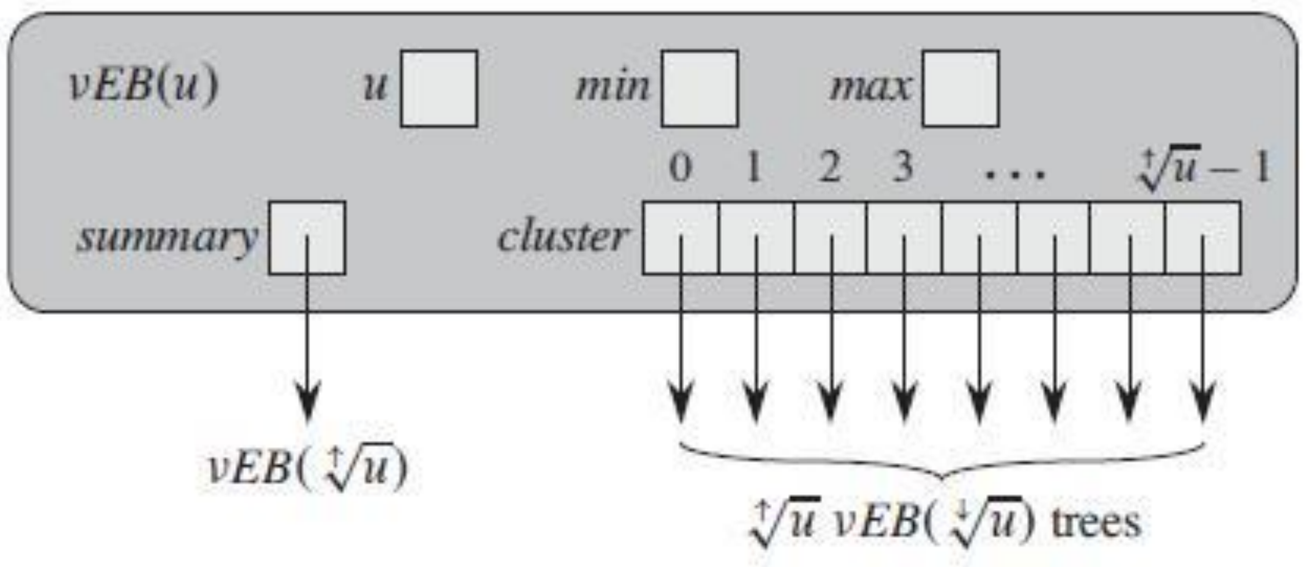
- The **real-vEB** modifies the proto-VEB, containing two new attributes:
  - **min** stores the minimum element in the real-vEB.
  - **max** stores the maximum element in the real-vEB.

- The attribute **summary** points to a **real-vEB($\uparrow u^{1/2}$)** tree.

- The array **cluster** points to **$\uparrow u^{1/2}$** **real-vEB($\downarrow u^{1/2}$)** trees.

- **Important**: The element stored in **min** does no appear in any of the recursive real-vEB trees that the *cluster* array points to.

The "REAL" van Emde Boas Tree



The "REAL" van Emde Boas Tree

- The elements stored in a **real-vEB(u)** tree **V.min** plus all the elements stored in the **real-vEB(↓u$^{1/2}$)** trees pointed to by **V.cluster[0 .. (↑u$^{1/2}$) – 1]**

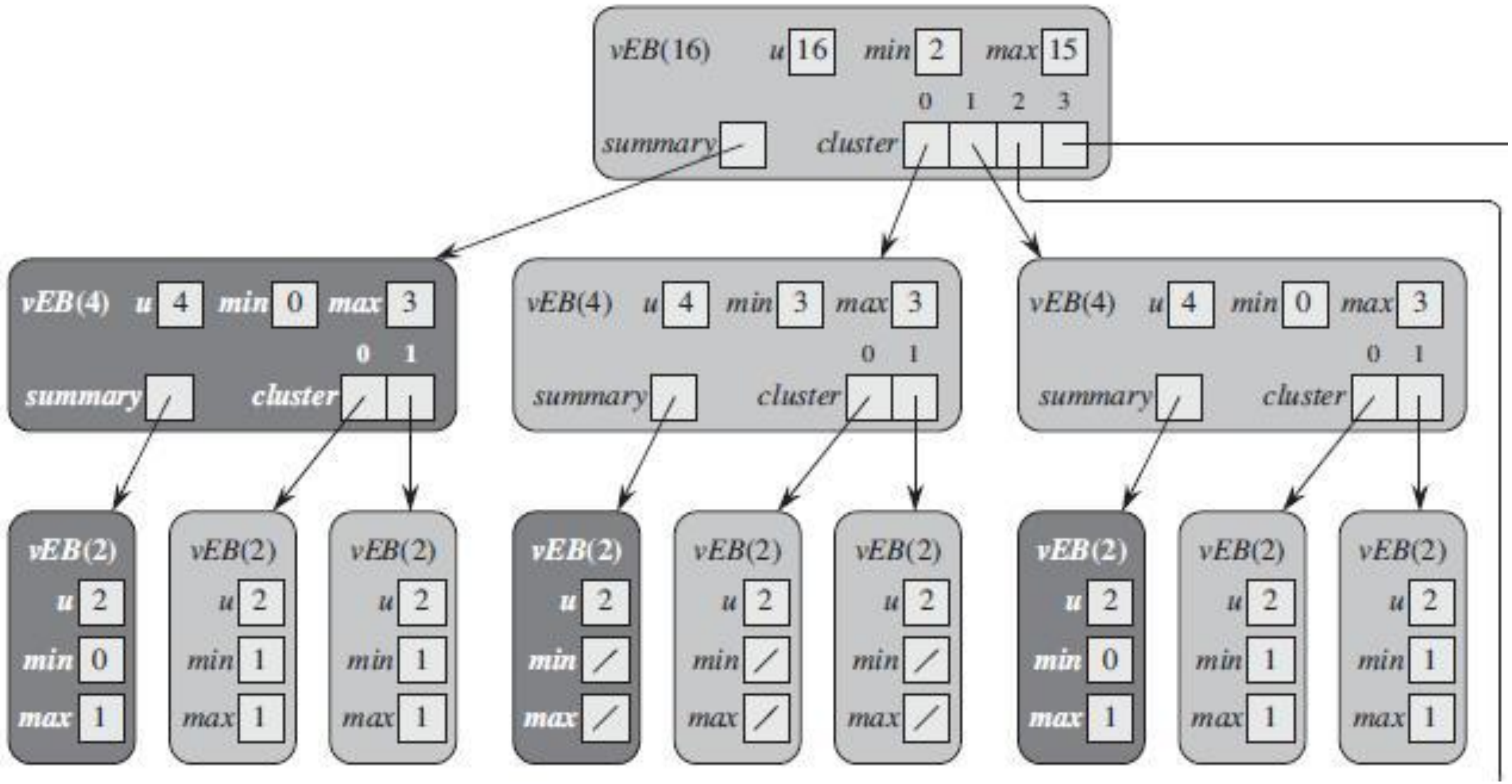
- When a real-vEB contains two or more elements the element stored in **min** does not appear in any of the clusters but the stored in **max** does (unless it is equal to **min**).

- Since the basis size is 2, real-vEB(2) does not need the array **A** of the corresponding proto-vEB(2). The attributes **min** and **max** can be used for it.

- **Important**: In the real-vEB, all the elements can be determined from the **min** and **max** attributes.

The "REAL" van Emde Boas Tree

- **Example**: real-vEB(16) holding the set **{2, 3, 4, 5, 7, 14, 15}**



Minimum and Maximum on vEB tree

vEB-TREE-MINIMUM (V, x)

return V.min

vEB-TREE-MAXIMUM (V, x)

return V.max

- **Running time**: O(1)

Member operation on vEB tree

```
vEB-TREE-MEMBER (V, x)
    if x == V.min or x == V.max
        return TRUE
    elseif  V.u == 2
        return  FALSE
    else  return vEB-TREE-MEMBER (V.cluster[high(x)], low(x))
```

- **Running time:**

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

$$\Rightarrow T(u) = O(\lg \lg u)$$

Successor operation on proto-vEB

```
PROTO-vEB-SUCCESSOR (V, x)
    if V.u == 2
        if x == 0  and V.A[1] == 1
            return 1
        else return NIL
    else  offset = PROTO-vEB-SUCCESSOR (V.cluster[high(x)], low(x))
        if offset ≠ NIL
            return index (high(x), offset)
        else
            succ-cluster = PROTO-vEB-SUCCESSOR (V.summary, high(x))
            if succ-cluster == NIL
                return NIL
            else offset = PROTO-vEB-MINIMUM (V.cluster[succ-cluster])
                return index(succ-cluster, offset)
```

Successor operation on vEB tree

```
vEB-TREE-SUCCESSOR (V, x)
if V.u == 2
    if x == 0  and V.max == 1
        return 1
    else return NIL
elseif V.min ≠ NIL and x < V.min
    return V.min
else  max-low = vEB-TREE-MAXIMUM (V.cluster[high(x)])
    if max-low ≠ NIL  and low(x) < max-low
        offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
        return index (high(x), offset)
    else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
        if succ-cluster == NIL
            return NIL
        else offset = vEB-TREE-MINIMUM (V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

Successor operation on vEB tree

- **Running time:**

$$T(u) \le \max\left\{T\left(\downarrow \sqrt{u}\right), T\left(\uparrow \sqrt{u}\right)\right\} + O(1)$$

$$\Rightarrow T(u) \le T\left(\uparrow \sqrt{u}\right) + O(1)$$

$$m = \lg u$$

$$\Rightarrow T\left(2^{m}\right) \le T\left(2^{\lceil m/2 \rceil}\right) + O(1)$$

$$(\forall m \ge 2)\left\lceil m/2 \right\rceil \le 2m/3$$

$$\Rightarrow T\left(2^{m}\right) \le T\left(2^{2m/3}\right) + O(1)$$

Successor operation on vEB tree

- **Running time:**

$$T\left(2^{m}\right) \leq T\left(2^{2m/3}\right) + O(1)$$

$$S(m) = T\left(2^{m}\right)$$

$$\Rightarrow S(m) \leq S(2m/3) + O(1)$$

$$\Rightarrow S(m) = O(\lg m)$$

$$\Rightarrow T(u) = T\left(2^{m}\right) = S(m) = O(\lg m) = O(\lg \lg u)$$

Predecessor operation on vEB tree

```
vEB-TREE-PREDECESSOR (V, x)
if V.u == 2
    if x == 1  and V.min == 0
        return 0
    else return NIL
elseif V.max ≠ NIL and x > V.max
    return V.max
else  min-low = vEB-TREE-MINIMUM (V.cluster[high(x)])
    if min-low ≠ NIL  and low(x) > min-low
        offset = vEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
        return index (high(x), offset)
    else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
        if pred-cluster == NIL
            if V.min ≠ NIL and x > V.min
                return V.min
            else return NIL
        else offset = vEB-TREE-MAXIMUM (V.cluster[pred-cluster])
            return index(succ-cluster, offset)
```

Predecessor operation on vEB tree

- **Running time:**

$$T(u) \le \max\left\{T\left(\downarrow\sqrt{u}\right), T\left(\uparrow\sqrt{u}\right)\right\} + O(1)$$

$$\Rightarrow T(u) = O(\lg\lg u)$$

Insert operation on proto-vEB

```
PROTO-vEB-INSERT (V, x)
    if V.u == 2
        V.A[x] = 1
    else
        PROTO-vEB-INSERT (V.cluster[high(x)], low(x))
        PROTO-vEB-INSERT (V.summary, high(x))
```

Insert operation on vEB tree

- The insertion operation can be split into two cases.
  The first one occurs when the cluster is empty.

vEB-EMPTY-TREE-INSERT (V, x)

```
1    V.min = x
2    V.max = x
```

- In the second case, one must avoid the two recursive calls, one for the summary and another for a cluster. How can we do it?

- **Answer:** check if the corresponding cluster is empty or not.

Insert operation on vEB tree

```
vEB-TREE-INSERT (V, x)
if V.min == NIL
    vEB-EMPTY-TREE-INSERT (V, x)
else
    if x < V.min
        exchange x with V.min
    if V.u > 2
        if  vEB-TREE-MINIMUM (V.cluster[high(x)]) == NIL
            vEB-TREE-INSERT(V.summary, high(x))
            vEB-EMPTY-TREE-INSERT (V.cluster[high(x)], low(x))
        else vEB-TREE-INSERT (V.cluster[high(x)], low(x))
    if x > V.max
        V.max = x
```

Insert operation on vEB tree

- **Running time:**

$$T(u) \le \max\left\{T\left(\downarrow\sqrt{u}\right), T\left(\uparrow\sqrt{u}\right)\right\} + O(1)$$
$$\Rightarrow T(u) = O(\lg\lg u)$$