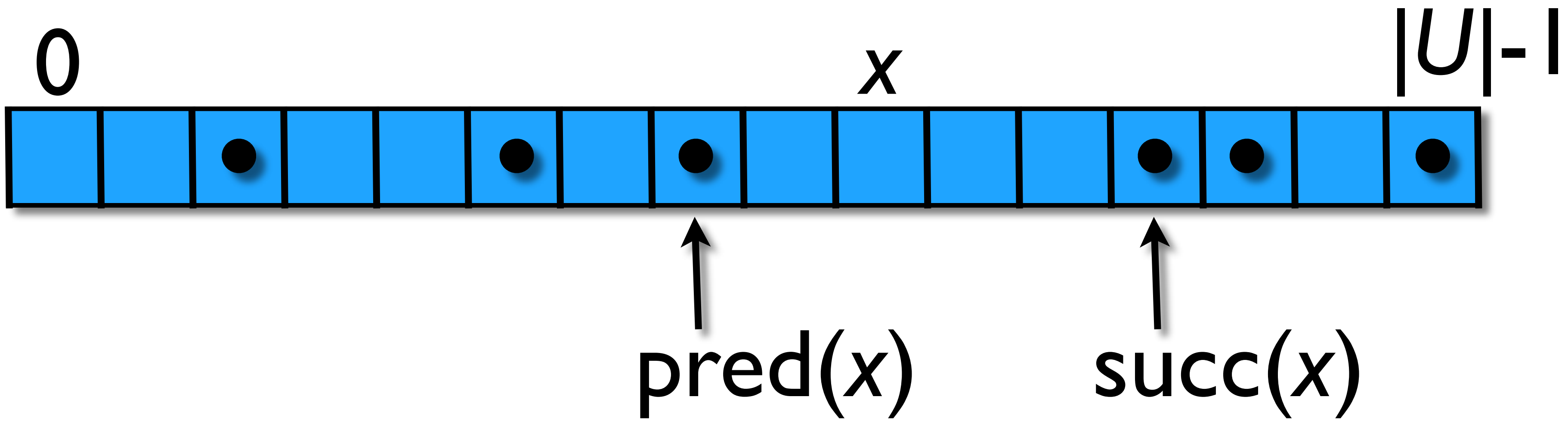
Lecture 3: Predecessor Data Structures

Johannes Fischer

Predecessor Queries

- S: n objects from a SORTED universe U
- given $x \in U$:
  - pred(x)=max$\{y \leq x : y \in S\}$
  - succ(x)=min$\{y \geq x : y \in S\}$

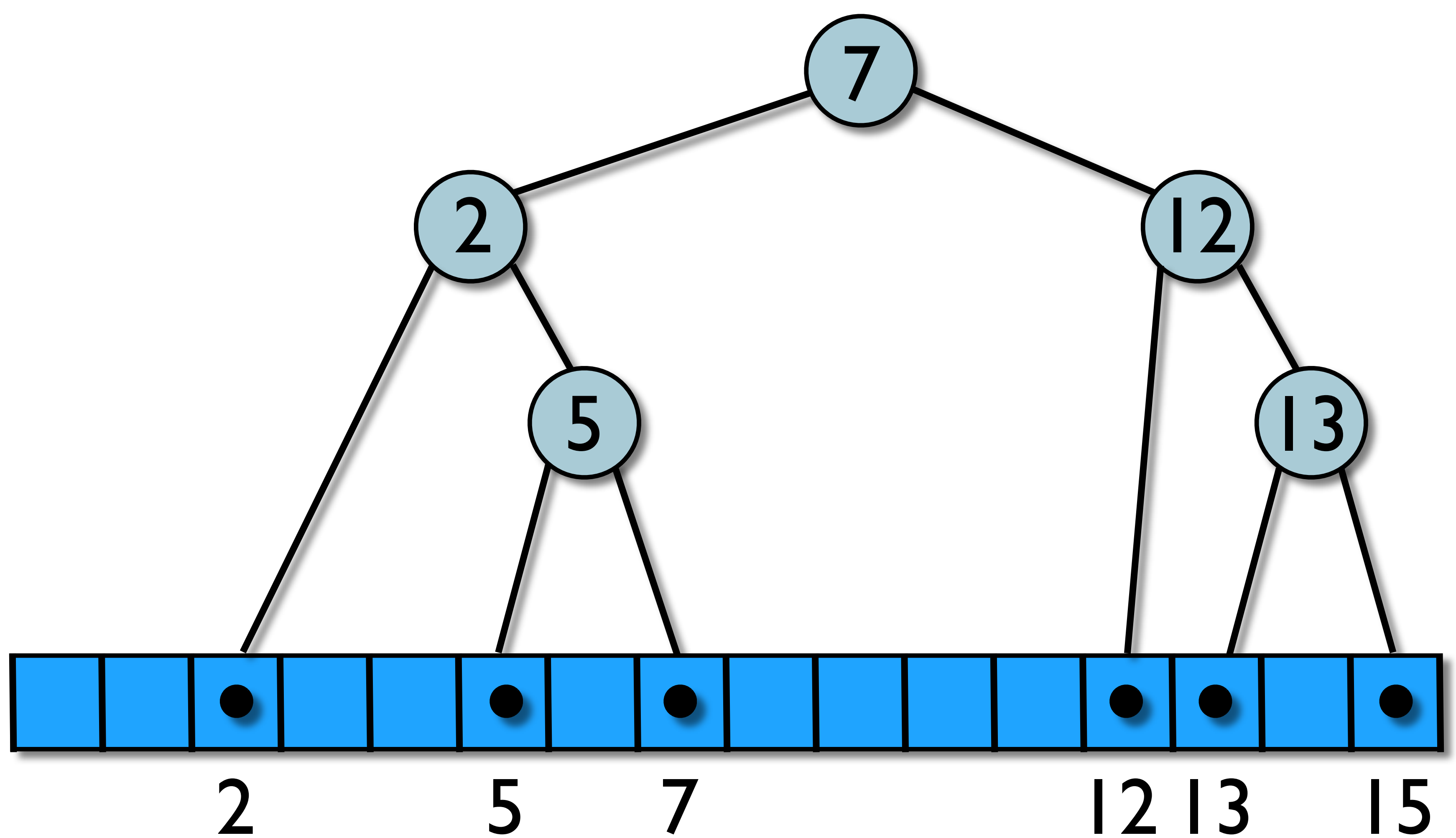


Applications

- very powerful/versatile
  - hash-table functionality
  - min/max ⇝ heaps/priority queues
  - 1D-nearest neighbor
  - 1D-range queries
  - IP-forwarding (prefix matching)

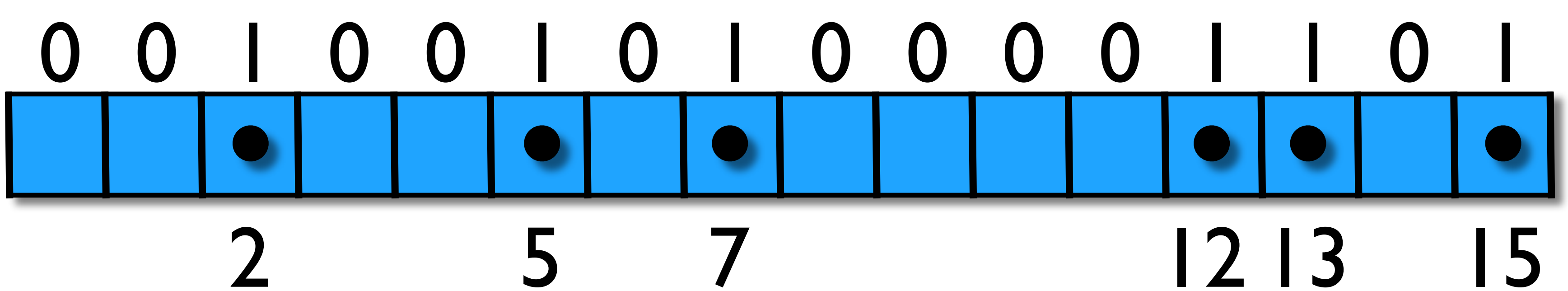
Baseline Algorithms

- balanced binary **search tree** over *S*
  - all ops (pred, succ, insert, ...) $O(\lg n)$ time
  - space $O(n)$



Baseline Algorithms

- **bit vector** marking members of S
  - insert/delete $O(1)$
  - pred/succ $O(u)$
  - space $O(u)$



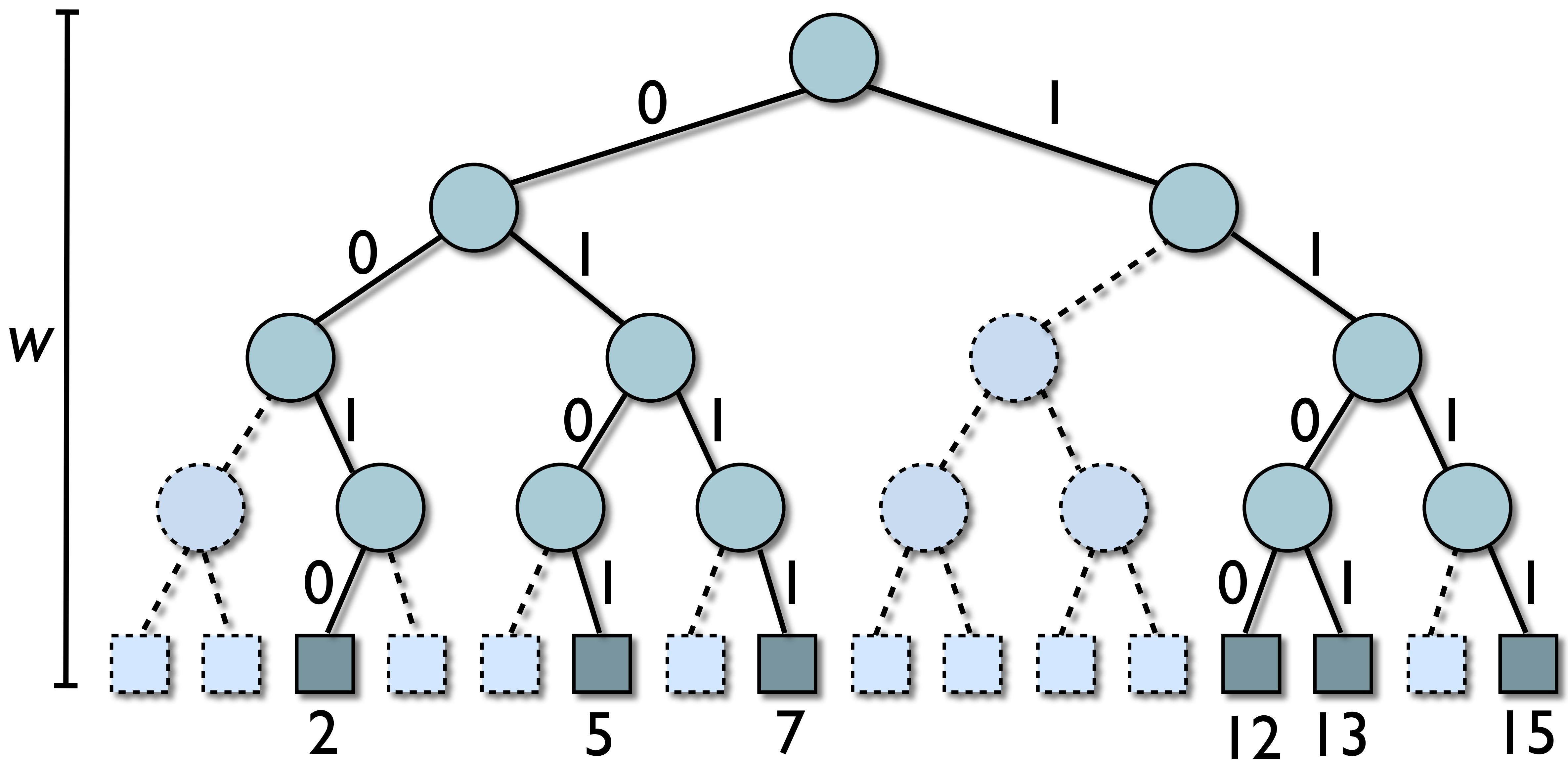
y-Fast Tries

- S static, $U = [0,u] = [0,2^w\text{-}1]$
  - all ops $O(\lg w) = O(\lg\lg u)$ time
- D. E. Willard [Inform. Proc. Lett. 1983]

y-fast tries	static	dynamic
pred/succ	$O(\lg w)$ w.c.	$O(\lg w)$ w.c.
insert/delete	n.a.	$O(\lg w)$ exp. & amort.
construction	$O(n)$ exp.+SORT(n,w)	n.a.
space	$O(n)$ w.c.	$O(n)$ w.c.

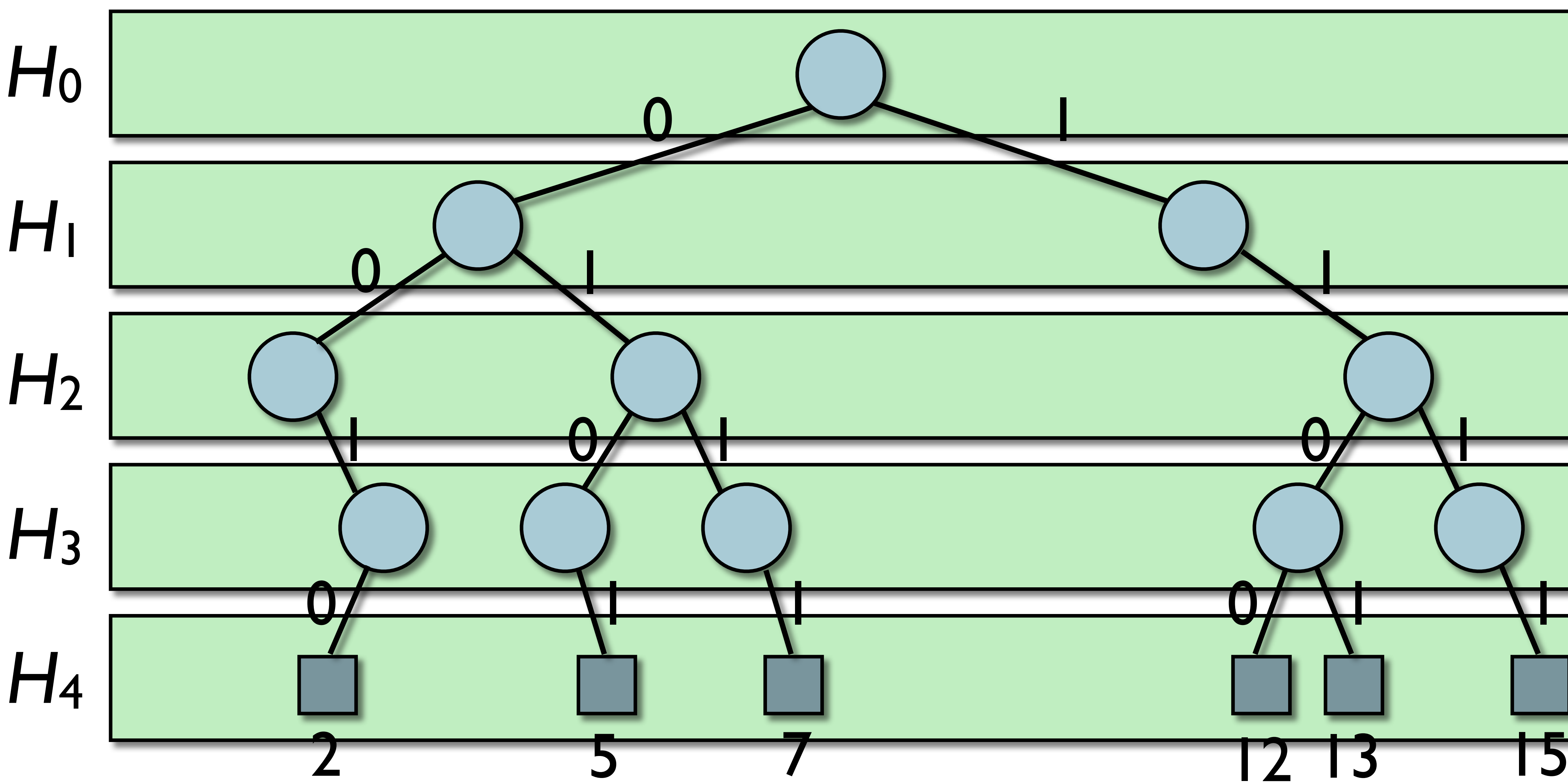
Idea

- bin(x): **binary representation** of x
  - store bin(x) for all $x \in S$ in a **trie**



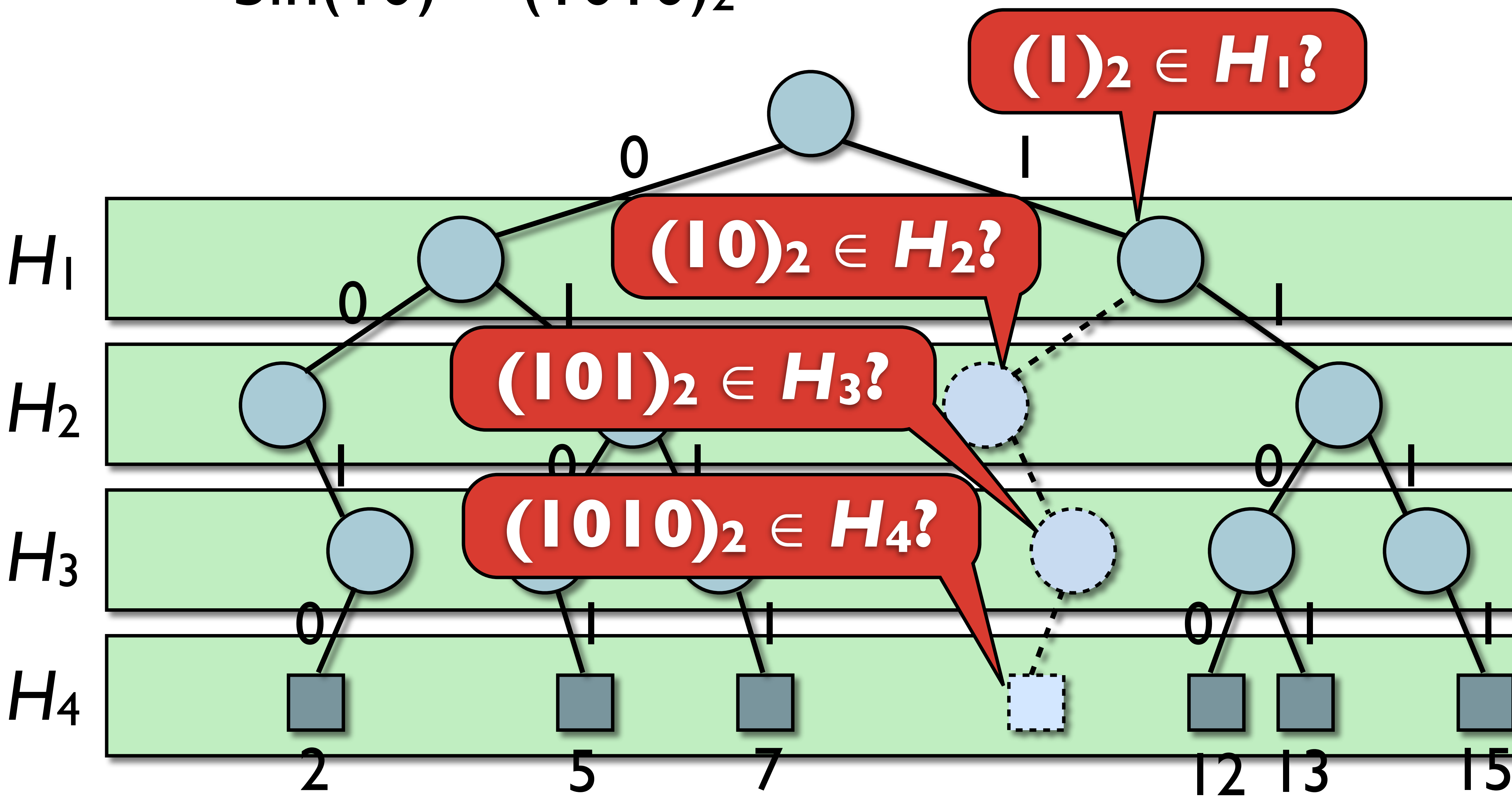
Idea

- need to know if node is there or not
  ⇒ store prefixes in w hash tables (perfect hashing)



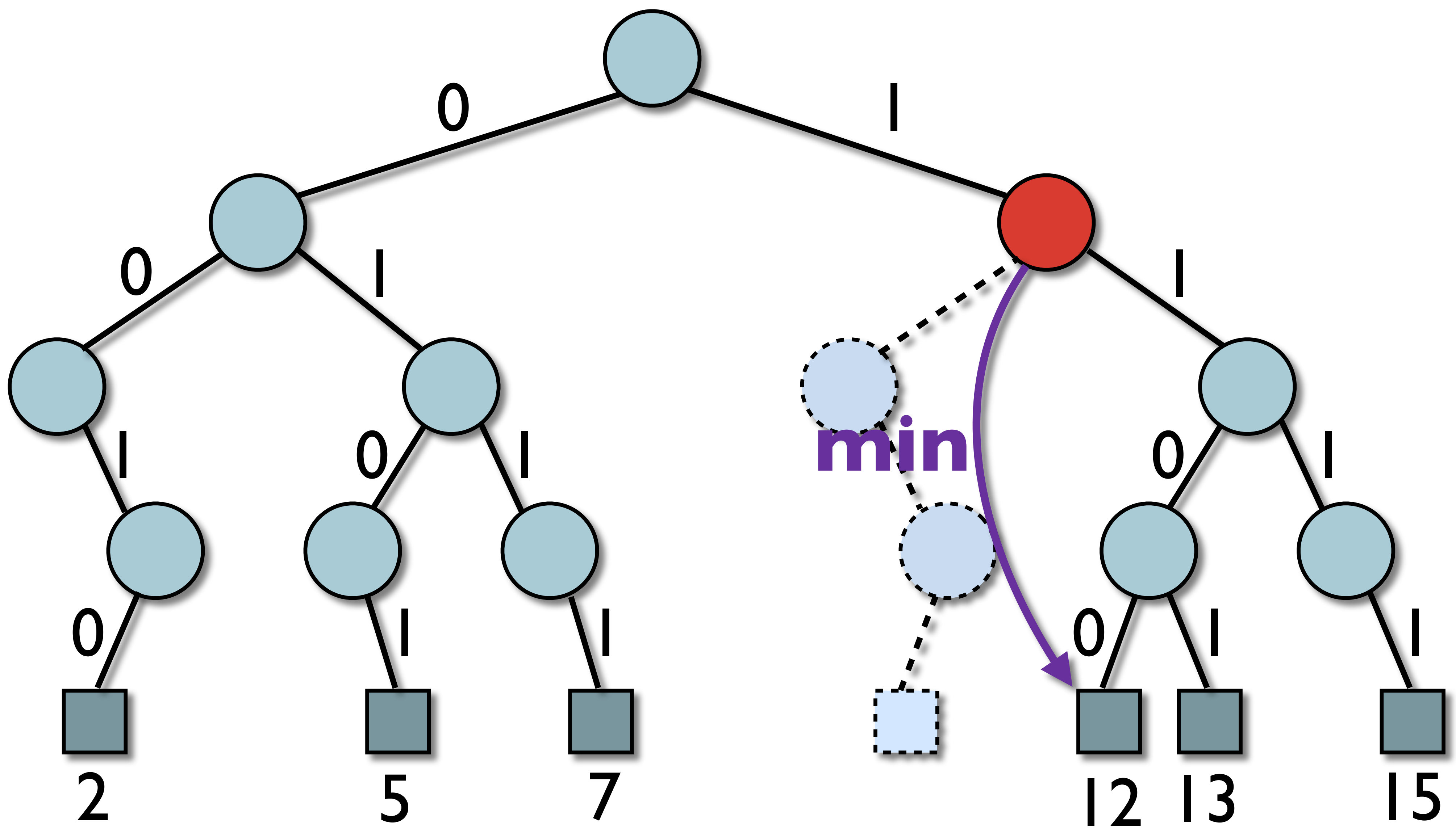
Successor Queries

- example: succ(10)
- bin(10) = $(1010)_2$



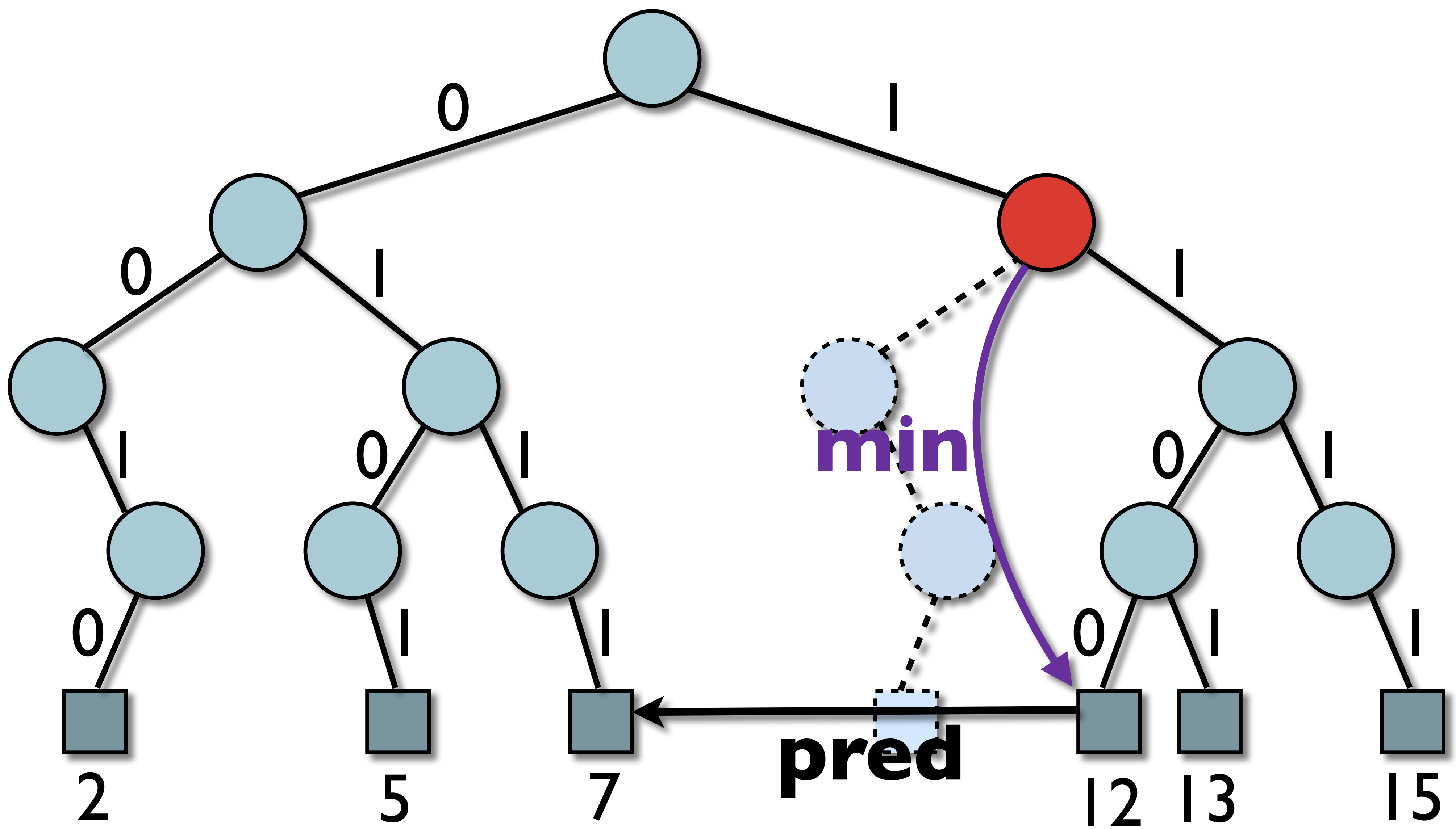
Successor Queries

- example: succ(10)
- bin(10) = $(1010)_2$



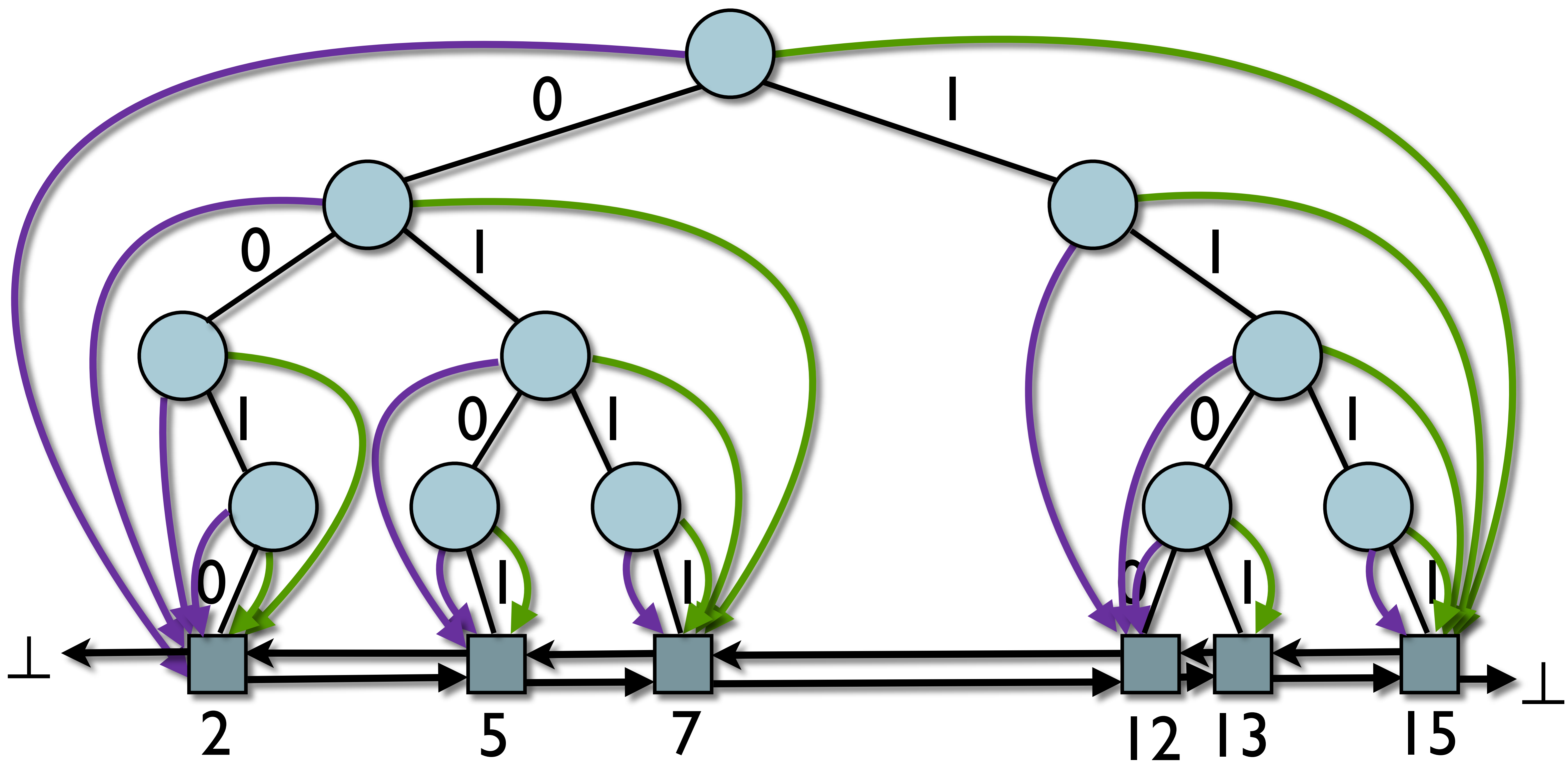
Predecessor Queries

- example: pred(10)
- bin(10) = $(1010)_2$

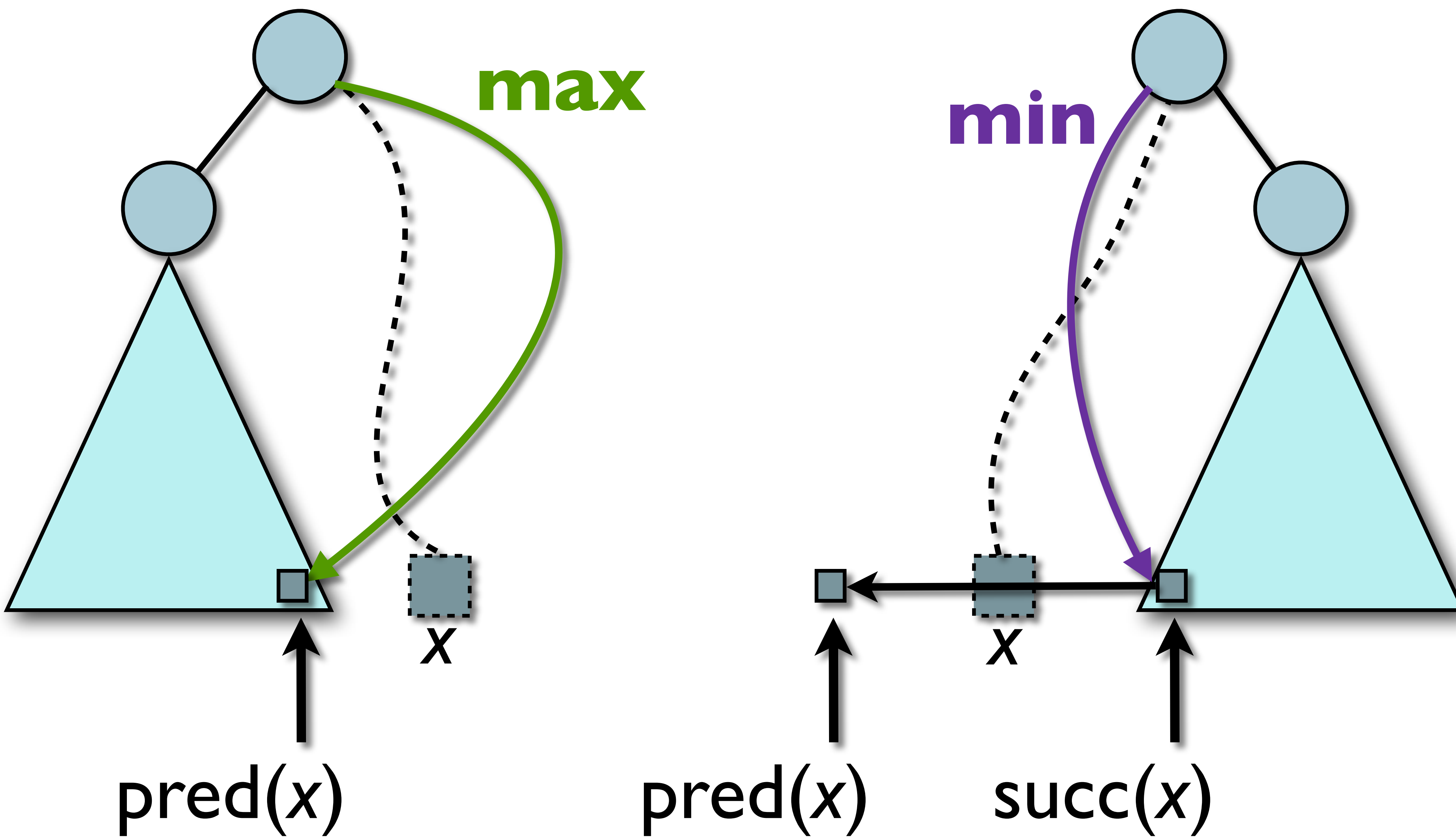


x-Fast Tries

- store min/max for every node
- leaves in a **doubly linked list**



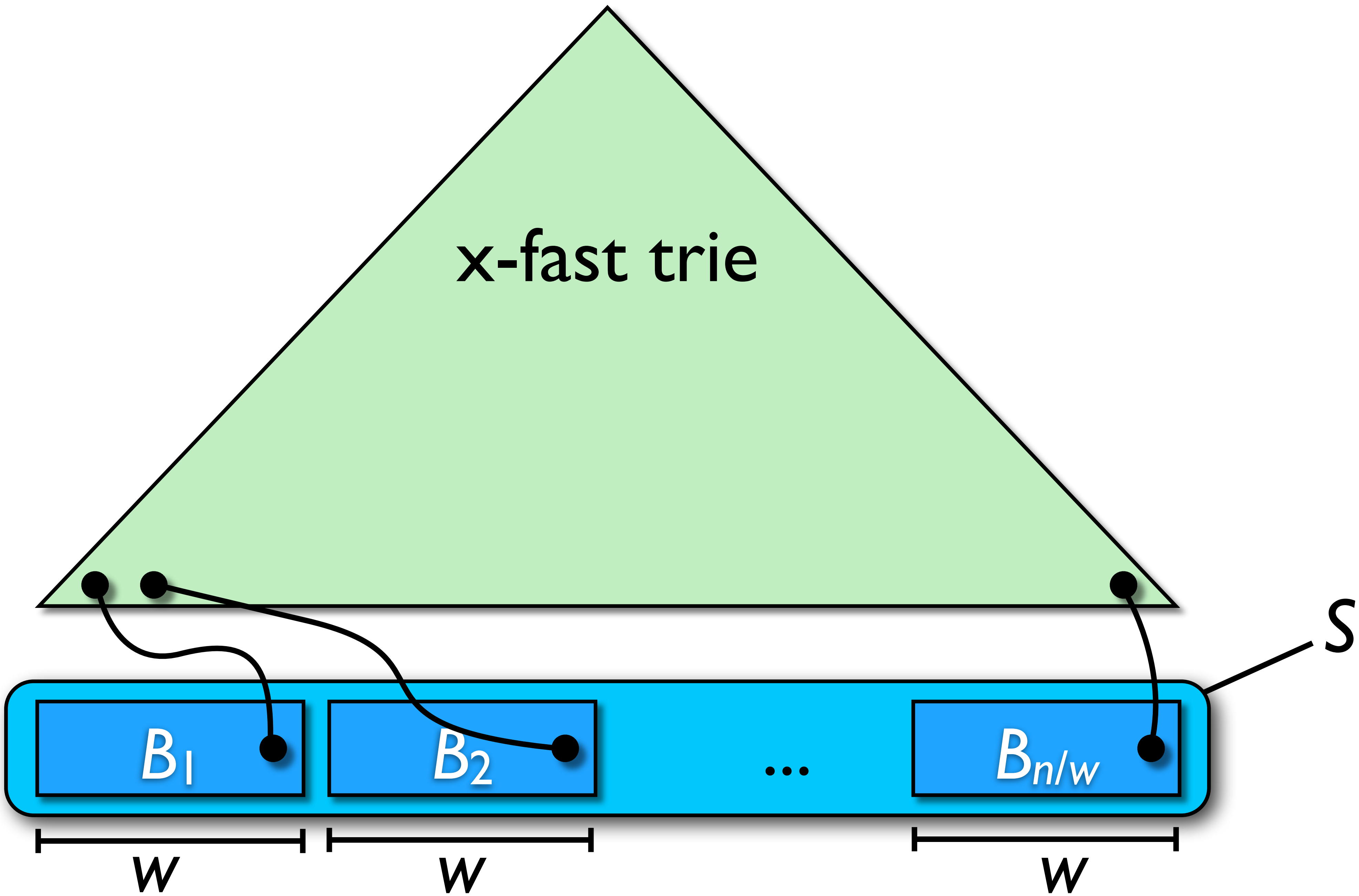
Predecessor Queries



The Final Picture

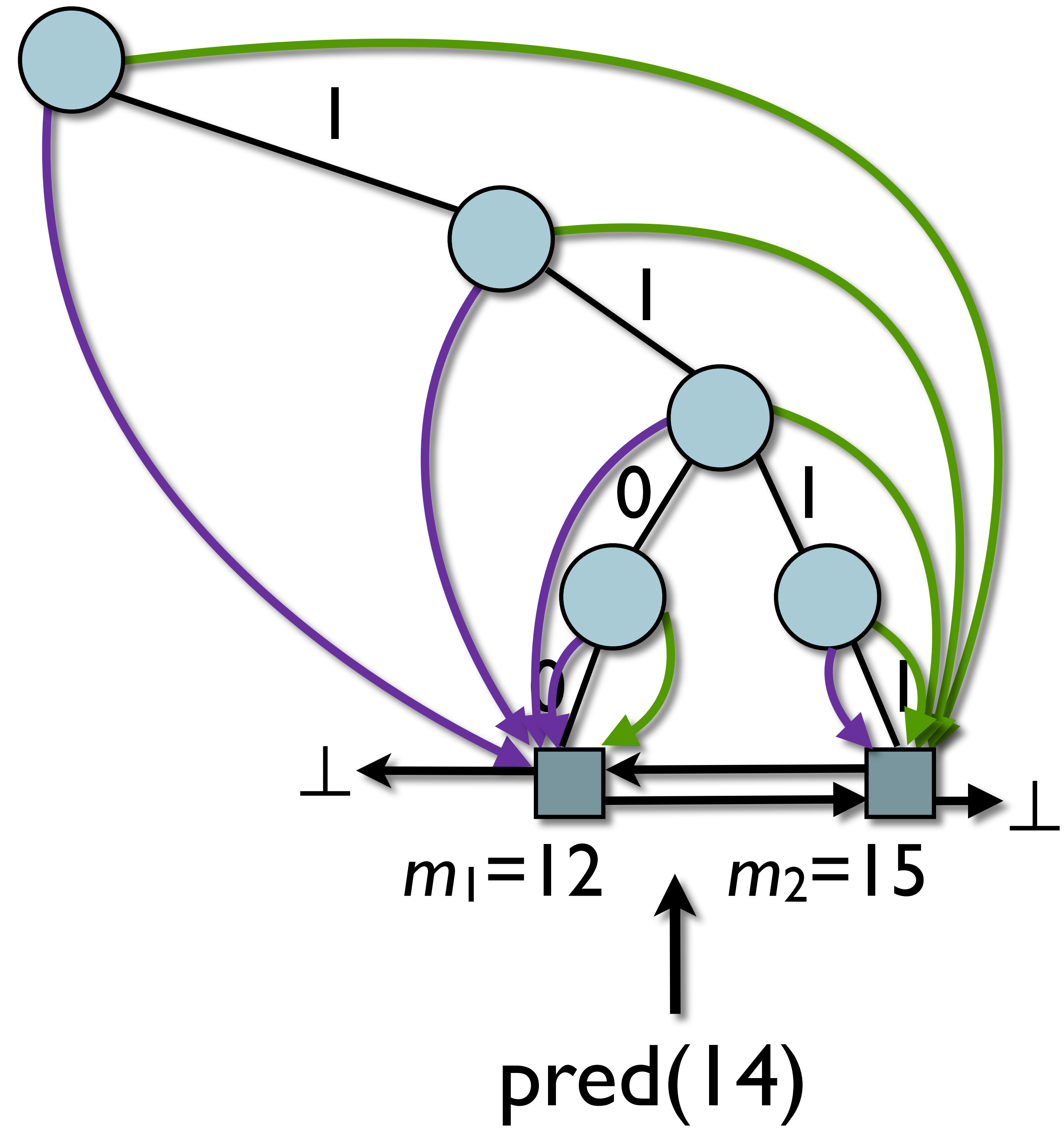
- predecessor $O(w)$, promised $O(\lg w)$
  - use **binary search** over heights → $O(\lg w)$
- space $O(nw)$, promised $O(n)$
  - use **indirection**:
  - blocks of w elements: $B_1, \ldots, B_{n/w}$ (sorted)
  - x-fast trie over $S' = \{m_i : 1 \leq i \leq n/w\}$, $m_i = \max B_i$
  - pred(x): (1) find pred among block maxima (m_p)
    (2) **binary search** block B_{p+1} ($O(\lg w)$)
    (3) result is either (1) or (2)

y-Fast Tries



Example

- B_1={2,5,7,**12**}
- B_2 = {13,**15**}



Dynamization

- ~~perfect hashing~~ ⇝ cuckoo hashing
- ~~arrays~~ ⇝ balanced search trees (e.g. AVL)
  - size between $w/2$ and $2w$
  - otherwise split/merge trees
- m_p = ~~maximum~~ ⇝ any separating element
- ⇒ pred/succ in O(lg w) w.c. time

  insert/delete O(lg w) **amort.&exp.**

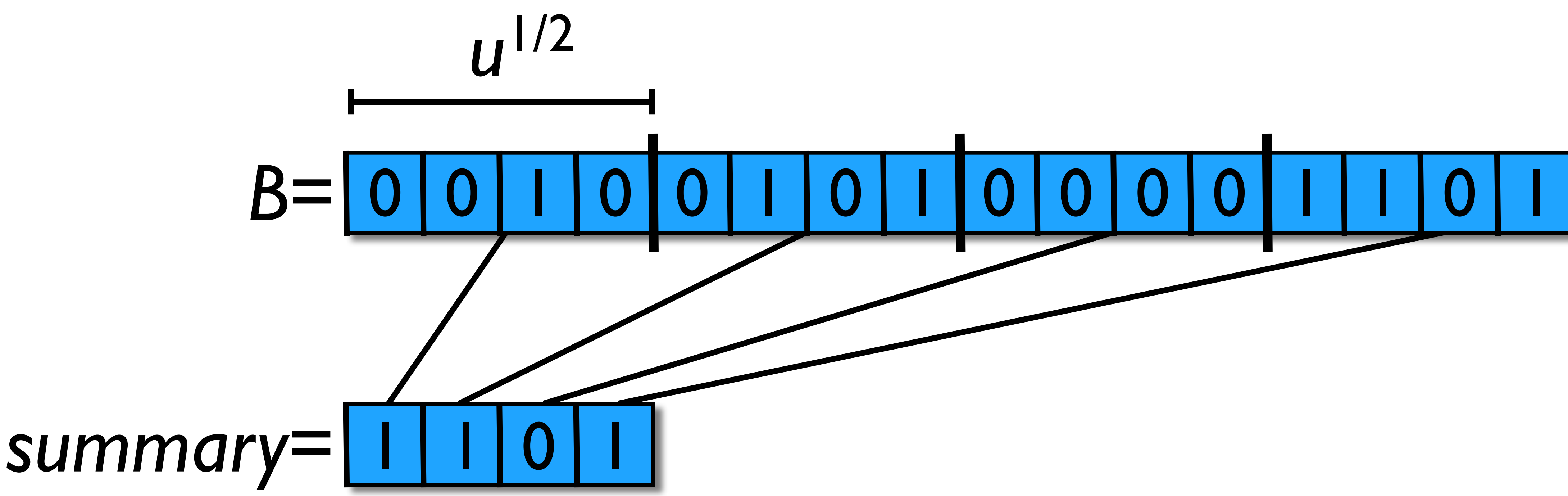
van Emde Boas Trees

- like dynamic y-fast tries
- van Emde Boas [FOCS'75]
- good in practice (→ VL Alg. Engineering)

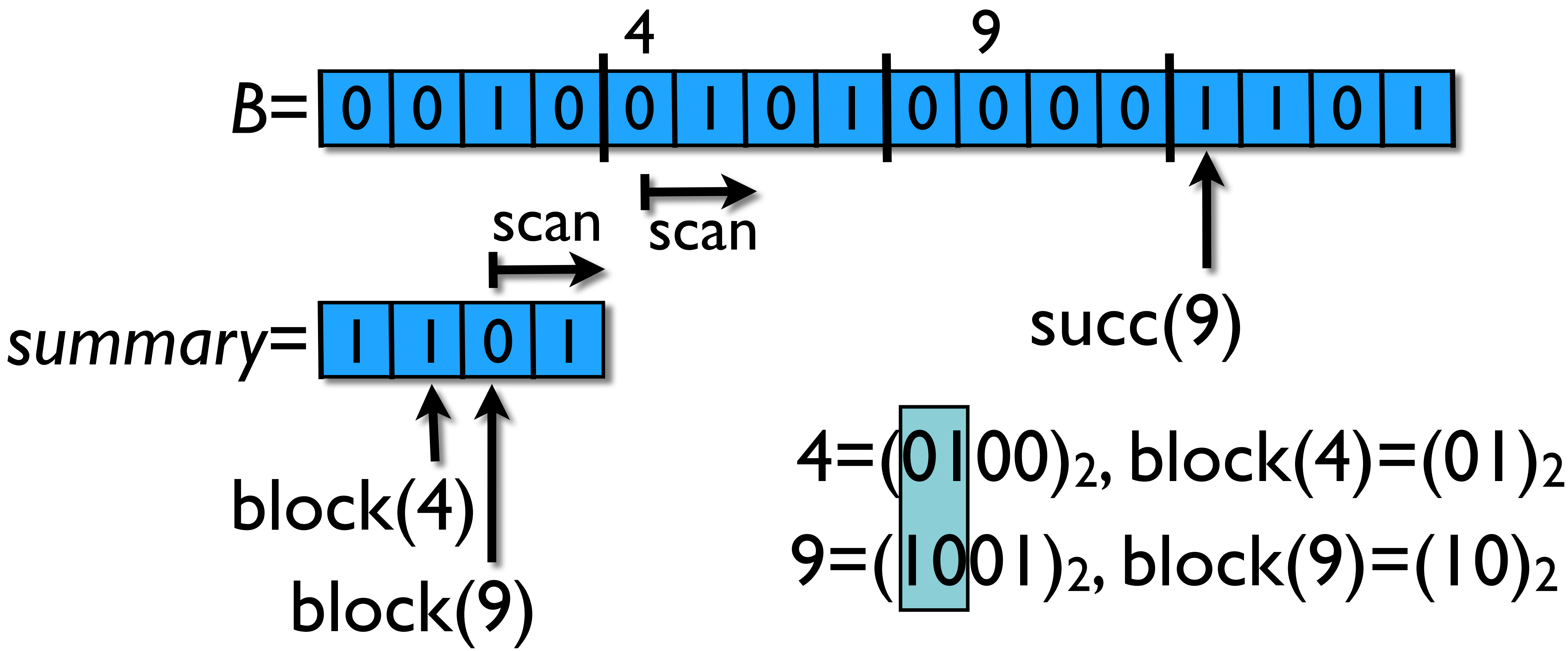
vEB trees	**dynamic**	
pred/succ	$O(\lg w)$ w.c.	$O(\lg w)$ w.c.
insert/delete	$O(\lg w)$ w.c.	$O(\lg w)$ exp. & amort.
space	$O(u)$ w.c.	$O(n)$ w.c.

Idea

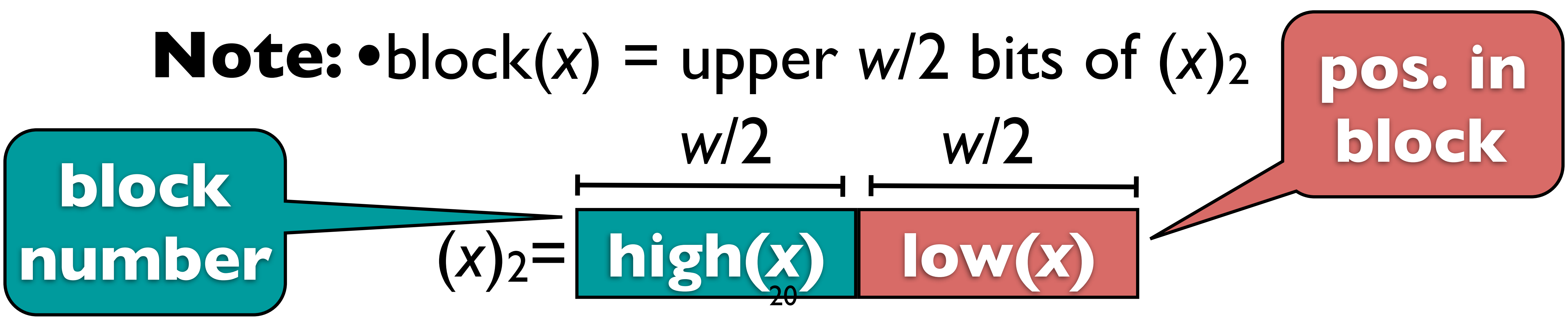
- **bit vector** B marking members of S
- $u^{1/2}$ blocks $B_0, B_1, \ldots$ of length $u^{1/2}$ (= $2^{w/2}$)
  - block(x)= $\lfloor x/u^{1/2} \rfloor$
- **summary** marking non-empty blocks



Finding Successors



Note: • block(x) = upper $w/2$ bits of $(x)_2$



Finding Successors

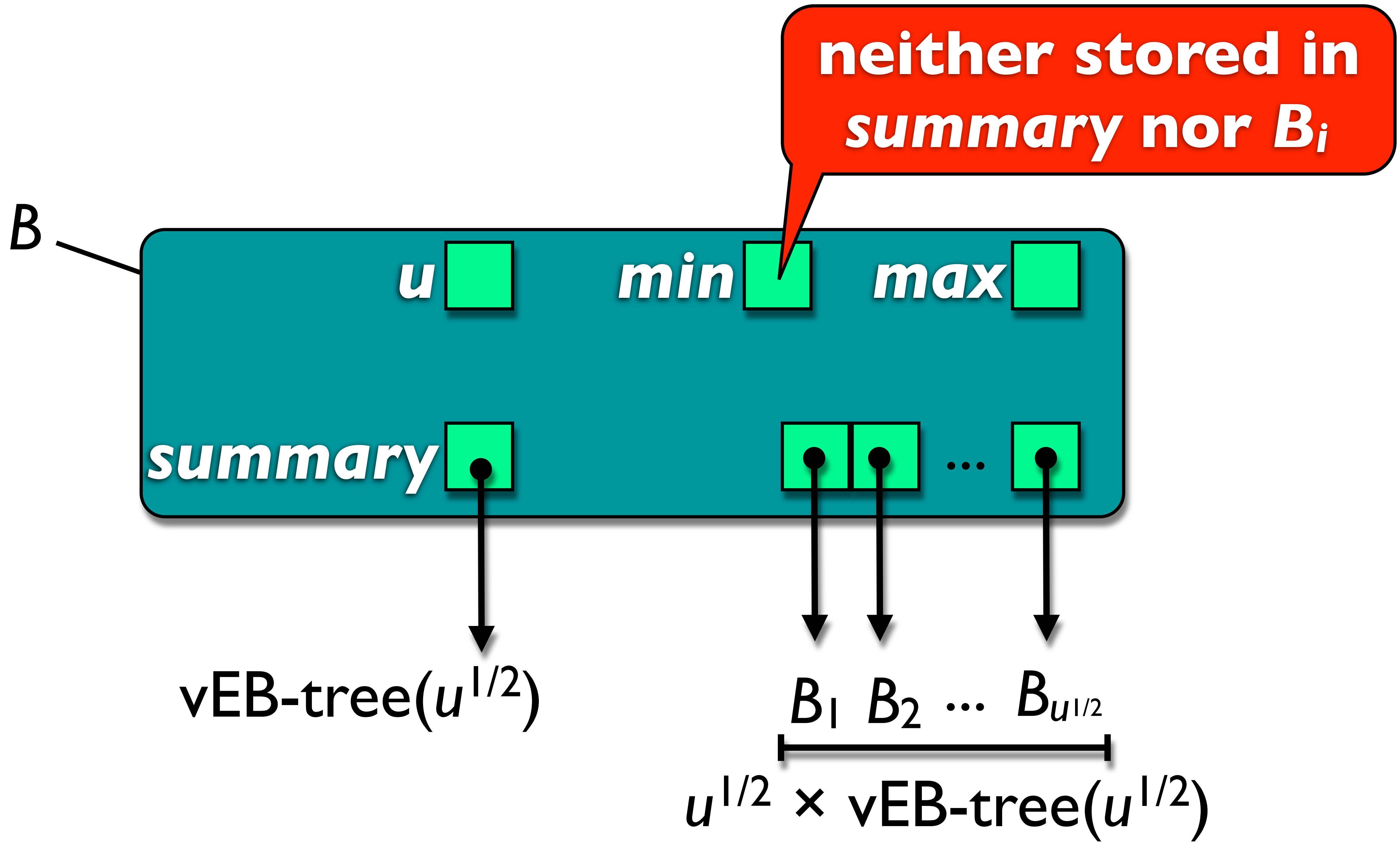
- scanning ≜ successor with **reduced size**
  - use **recursion**

function succ(B, x):
  inblock-succ ← succ($B_{\text{high}(x)}$, low(x))
  if (*inblock-succ* ≠ ⊥)
    return *inblock-succ* + (high(x) × $B.u^{1/2}$)
  else
    succ-block ← succ(B.summary, high(x))
    if (*succ-block* = ⊥) **return** ⊥
    return min($B_{\textit{succ-block}}$) + (*succ-block* × $B.u^{1/2}$)

Running Time

- base case if $B.u$=2
- $T(u) = 2T(u^{1/2}) + O(1)$
  $= \Theta(\lg u)$
- Too **slow**!
- Modify for only **one** recursive call
  - $T'(u) = T'(u^{1/2}) + O(1)$
    $= \Theta(\lg\lg u)$
- **Idea**: storing also **max** saves 1 recursion

vEB Tree Node



Successor Revisited

function succ(B, x):
    if (min(B) $\neq \perp$ **and** $x <$ min(B)) **return** min(B)
    else
        block-max $\leftarrow$ max($B_{\text{high}(x)}$)
        if (*block-max* $\neq \perp$ **and** low(x) $<$ *block-max*)
            inblock-succ $\leftarrow$ succ($B_{\text{high}(x)}$, low(x))
            return *inblock-succ* + (high(x) $\times$ $B.u^{1/2}$)
        else
            succ-block $\leftarrow$ succ(B.summary, low(x))
            if (*succ-block* = $\perp$) **return** $\perp$
            return min($B_{\textit{succ-block}}$) + (*succ-block*$\times B.u^{1/2}$)

Insertions

function insert(B, x):
    if (min(B) = ⊥) min(B) ← max(B) ← x
    else
        if (x < min(B)) swap x with min(B)
        if (min($B_{high(x)}$) = ⊥)
            insert(B.summary, high(x))
        insert($B_{high(x)}$, low(x))
        if (max(B) < x) max(B) ← x

Space

- $S(u) = \overbrace{(1+u^{1/2}) \times S(u^{1/2})}^{\text{recursive structures}} + \overbrace{\Theta(u^{1/2})}^{\text{pointers}}$
  $= \Theta(u)$
- space $\Theta(n)$:
  - store only non-empty blocks recursively
  - use hash tables!
  - summary only if ≥ 1 non-empty block