Lecture 3: Cuckoo Hashing (ctd.) Predecessor Data Structures

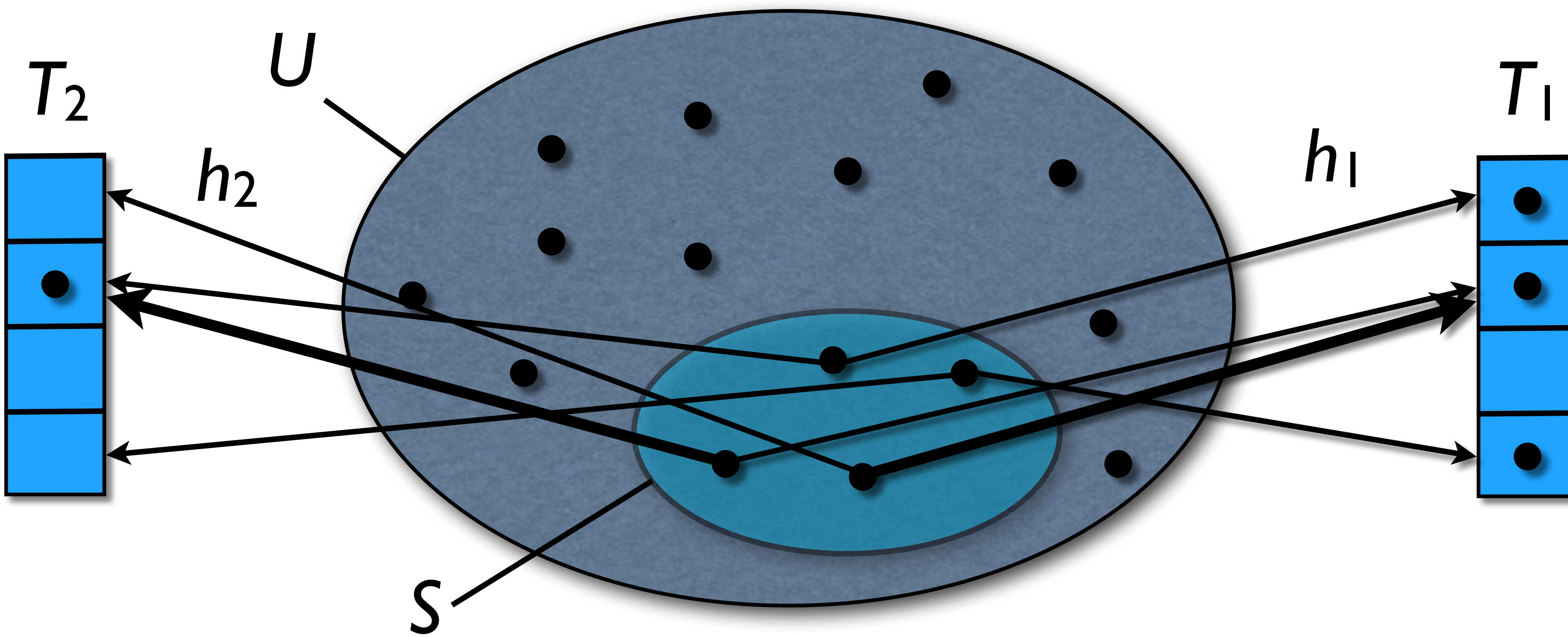
Johannes Fischer

Cuckoo Hashing

cuckoo hashing	
search	$O(1)$ w.c.
insert	$O(1)$ exp., amort.
delete	$O(1)$ exp., amort.
space	$O(n)$ w.c.

Idea

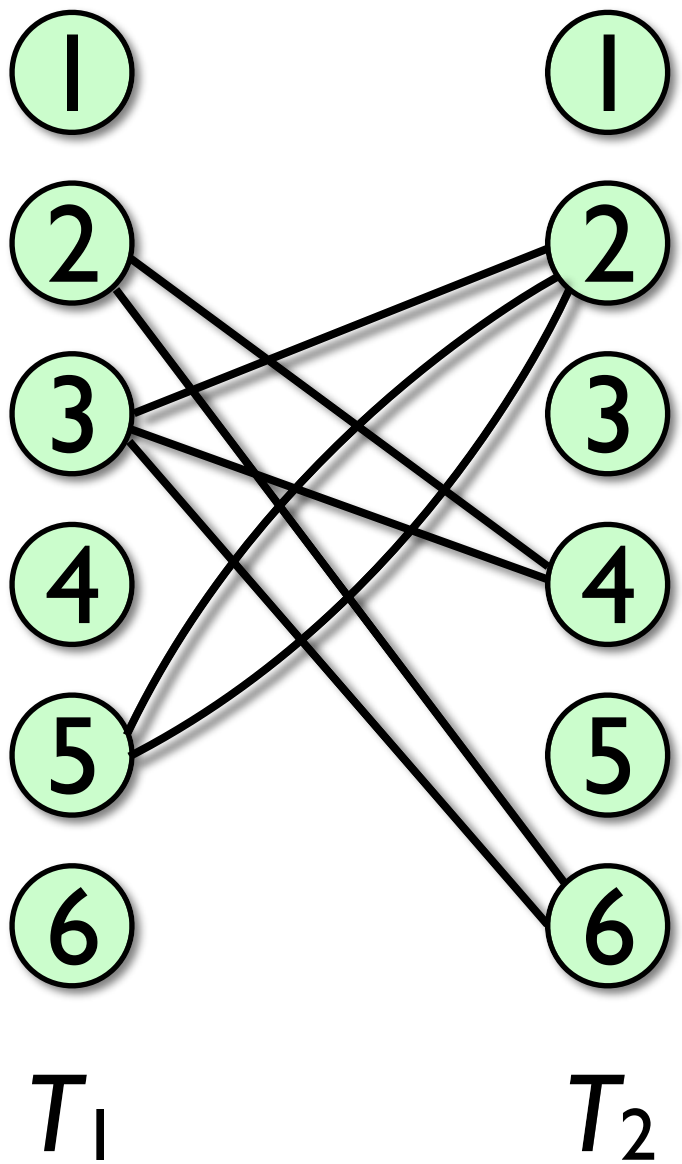
- 2 tables T_1 and T_2 (both of size $\Theta(n)$)
- 2 hash functions h_1 and h_2
  - x either at $T_1[h_1(x)]$ or $T_2[h_2(x)]$



Insertion

```
function insert(x):
    if (search(x)) return
    k ← 1
    repeat maxLoop times:
        swap x with T_k[h_k(x)]
        if (x = ⊥)
            n++; if (n > m/2) rehash(2m)
            return
        k ← 3 - k
    rehash(m); insert(x)
```

Cuckoo Graph



x	$h_1(x)$	$h_2(x)$
A	3	2
B	5	2
C	3	6
D	2	4
E	5	2
F	2	6
G	3	4

- insertions ⇔ walk in cuckoo graph

Analysis

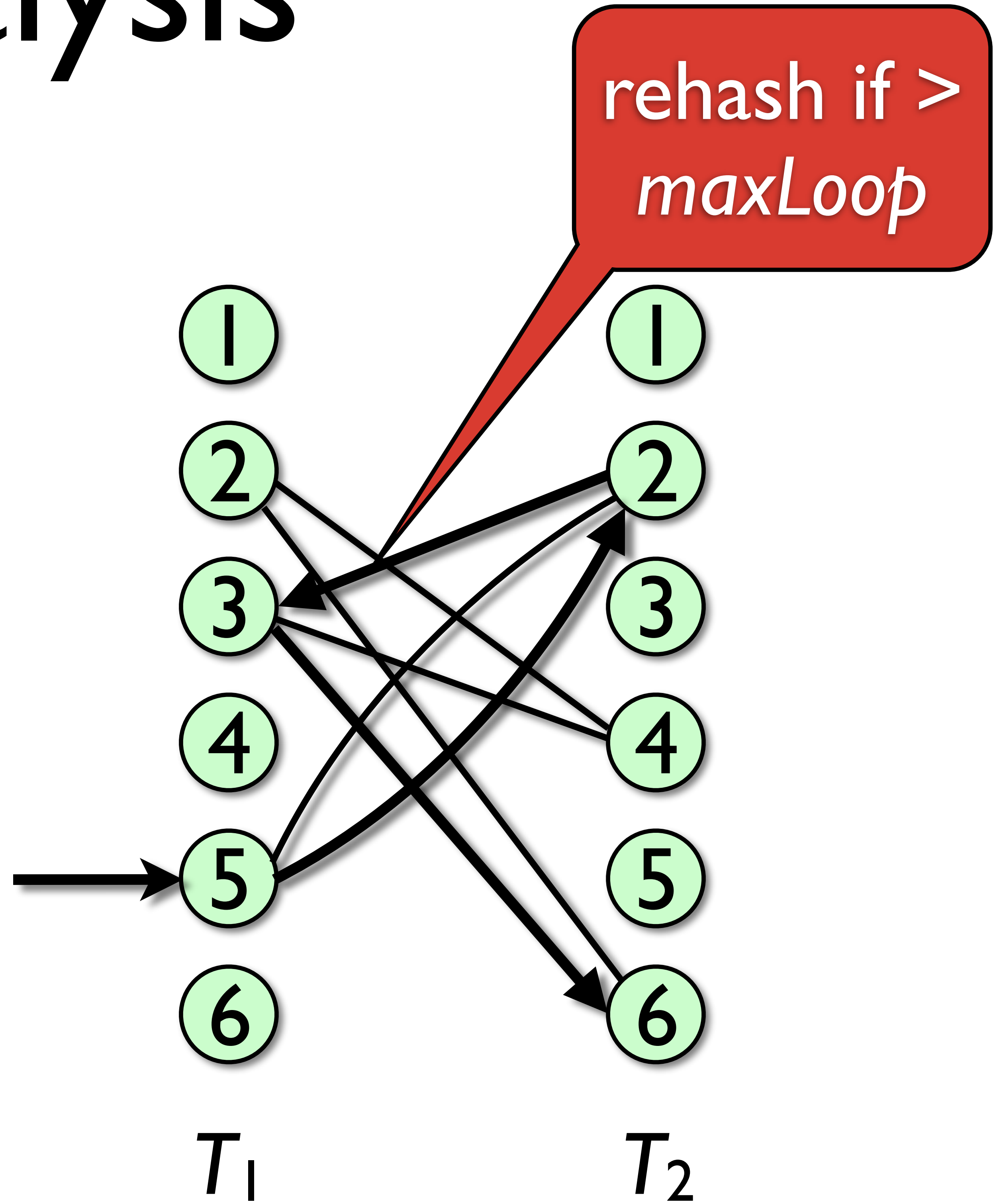
- ⇒ analyze

  probability
  of walks of length
  maxLoop

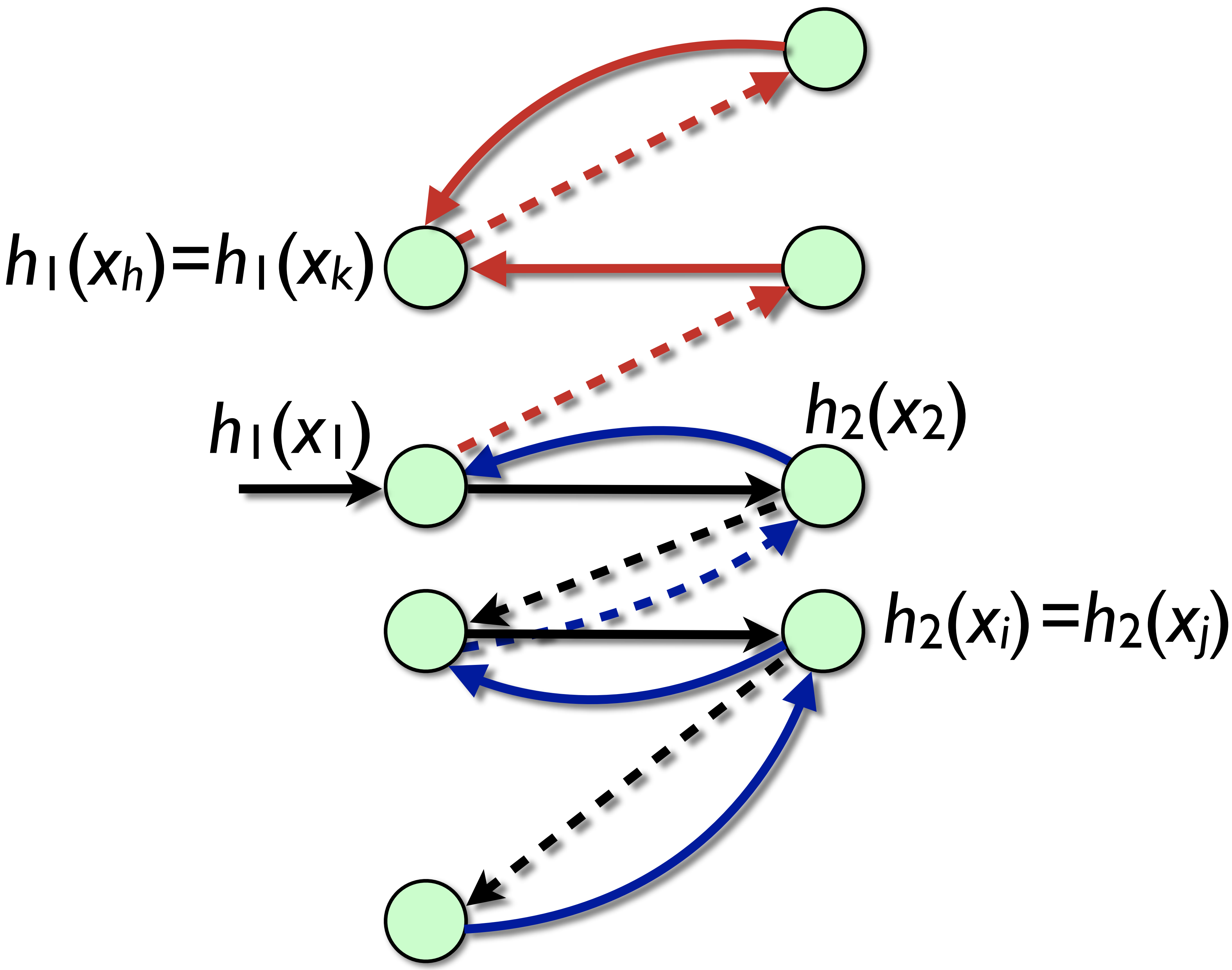
  - cause rehash!

- distinguish 3 cases:
  I. no cycle
  II. 1 cycle
  III. 2 cycles

- fix *maxLoop*=6lg *n*



III: Two Cycles



III: Two Cycles

- Analysis by **counting** # length-t 2-cycle walks in (arbitrary) cuckoo graphs **starting at $\boldsymbol{h_1(x_1)}$** ($N(n,m,t)$)
- walk $h_1(x_1), h_2(x_2), h_1(x_3), \ldots, h_{1/2}(x_t)$ with $h_{1/2}(x_t)$ forming 2nd loop
- $N(n,m,t) \leq t^3 n^{t-1} m^{t-1}$
  - t^3 possible ways of forming 2 loops
  - x_i may be any $s \in S$ ($i \geq 2$) $\Rightarrow n^{t-1}$ choices
  - $h_{1/2}(x_i)$ may be any $h \in [1,m]$ ($i \geq 2$) $\Rightarrow m^{t-1}$ choices

III: Two Cycles

- probability of each possibility is $\leq m^{-2t}$:

$$\mathrm{Prob}[h_1(x_1)=i_1 \wedge h_2(x_1)=j_1 \wedge \ldots \wedge h_1(x_t)=i_t \wedge h_2(x_t)=j_t]$$

$$= \mathrm{Prob}[h_1(x_1)=i_1 \wedge \ldots \wedge h_1(x_t)=i_t] \cdot \mathrm{Prob}[h_2(x_1)=j_t \wedge \ldots \wedge h_2(x_t)=j_t]$$

$$\leq m^{-t} \cdot m^{-t}$$

$$= m^{-2t}$$

III: Two Cycles

- $\Rightarrow$ probability of case (3) (=rehash) at most

$$\sum_{t=3}^{6 \lg n} \frac{t^3 n^{t-1} m^{t-1}}{m^{2t}}$$

$$= \sum_{t=3}^{6 \lg n} \frac{t^3 n^{t-1}}{m^{t+1}}$$

$$= \frac{1}{mn} \sum_{t=3}^{6 \lg n} t^3 \left(\frac{n}{m}\right)^t$$

$$\leq \frac{1}{2n^2} \sum_{t \geq 1} t^3 \left(\frac{1}{2}\right)^t = O\left(\frac{1}{n^2}\right)$$

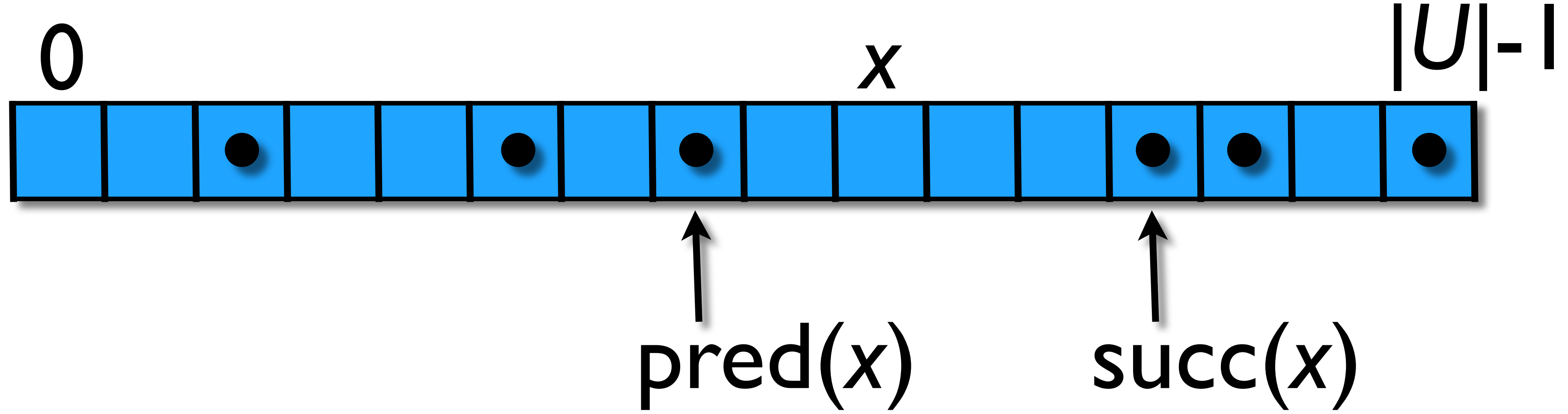
Wrapping Up

- Prob[**1** insert loops *maxLoop* times]=$O(n^{-2})$
- ⇒ Prob[***n*** inserts cause rehash(m)]=$O(n^{-1})$
- ⇒ Prob[rehash **successful**]=1-$O(n^{-1})$
- ⇒ Exp[**# trials** for rehash]=$O(1)$
- ⇒ Exp[**time** for rehash] =$O(n)$
- ⇒ $O(1)$ amortized insert. time (exp.)

```
function insert(x):
    if (search(x)) return
    k ← 1
    repeat maxLoop times:
        swap x with T_k[h_k(x)]
        if (x = ⊥)
            n++; if (n > m/2) rehash(2m)
            return
        k ← 3 - k
    rehash(m); insert(x)
```

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$\{y \geq x : y \in S\}$

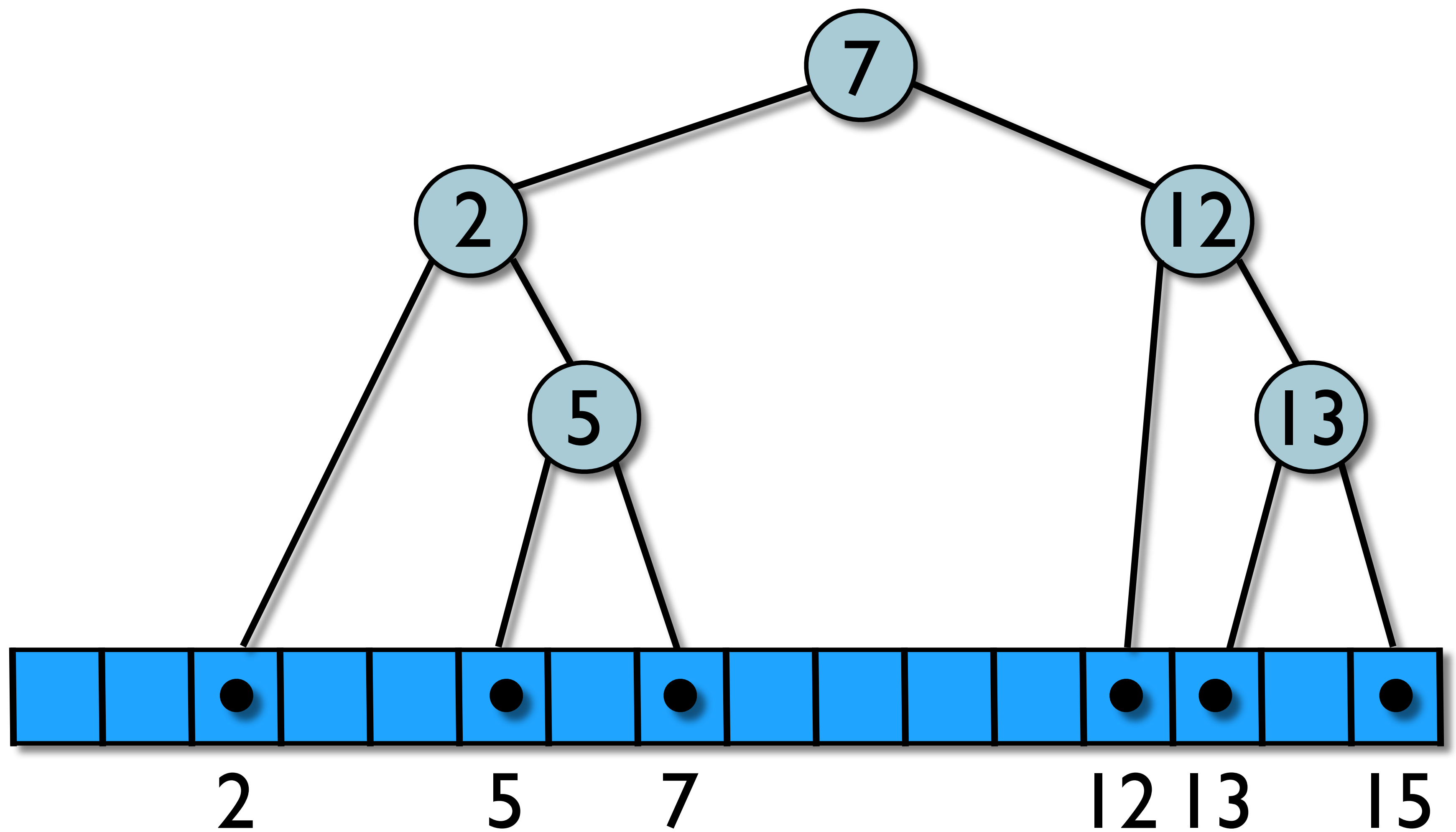


Applications

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

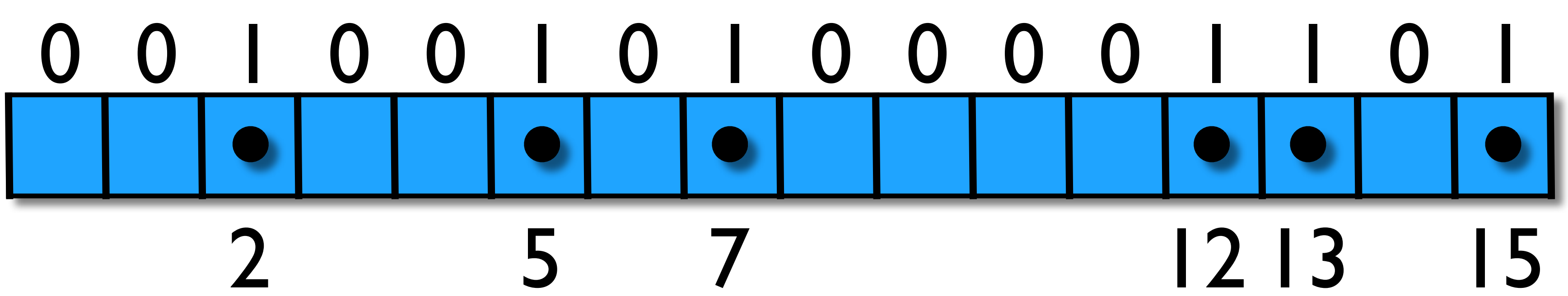
Baseline Algorithms

- 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 $O(u)$



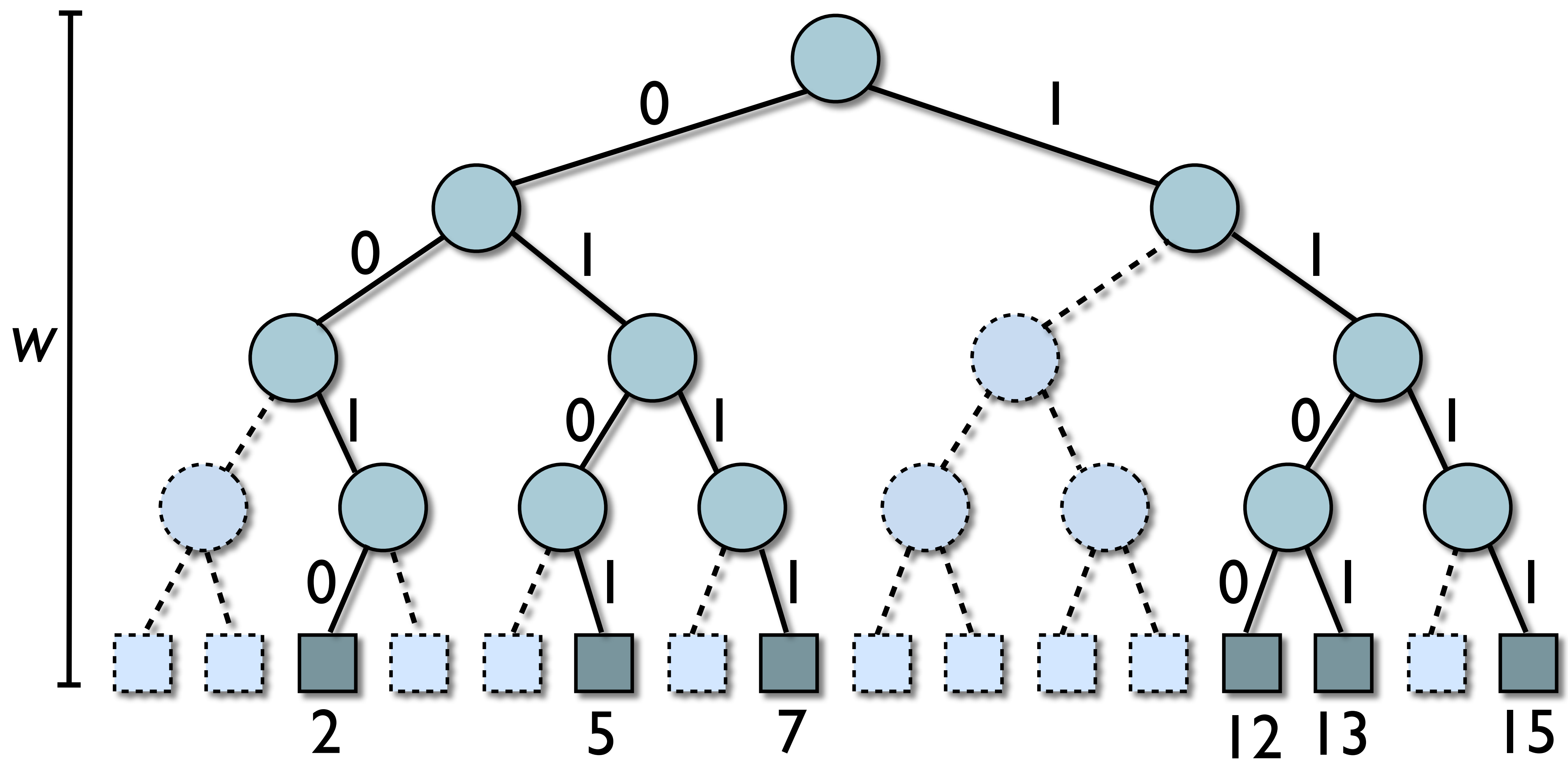
y-Fast Tries

- S static, U = [0,u] = [0,2^w-1]
  - all ops O(lg w) = O(lglg 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.

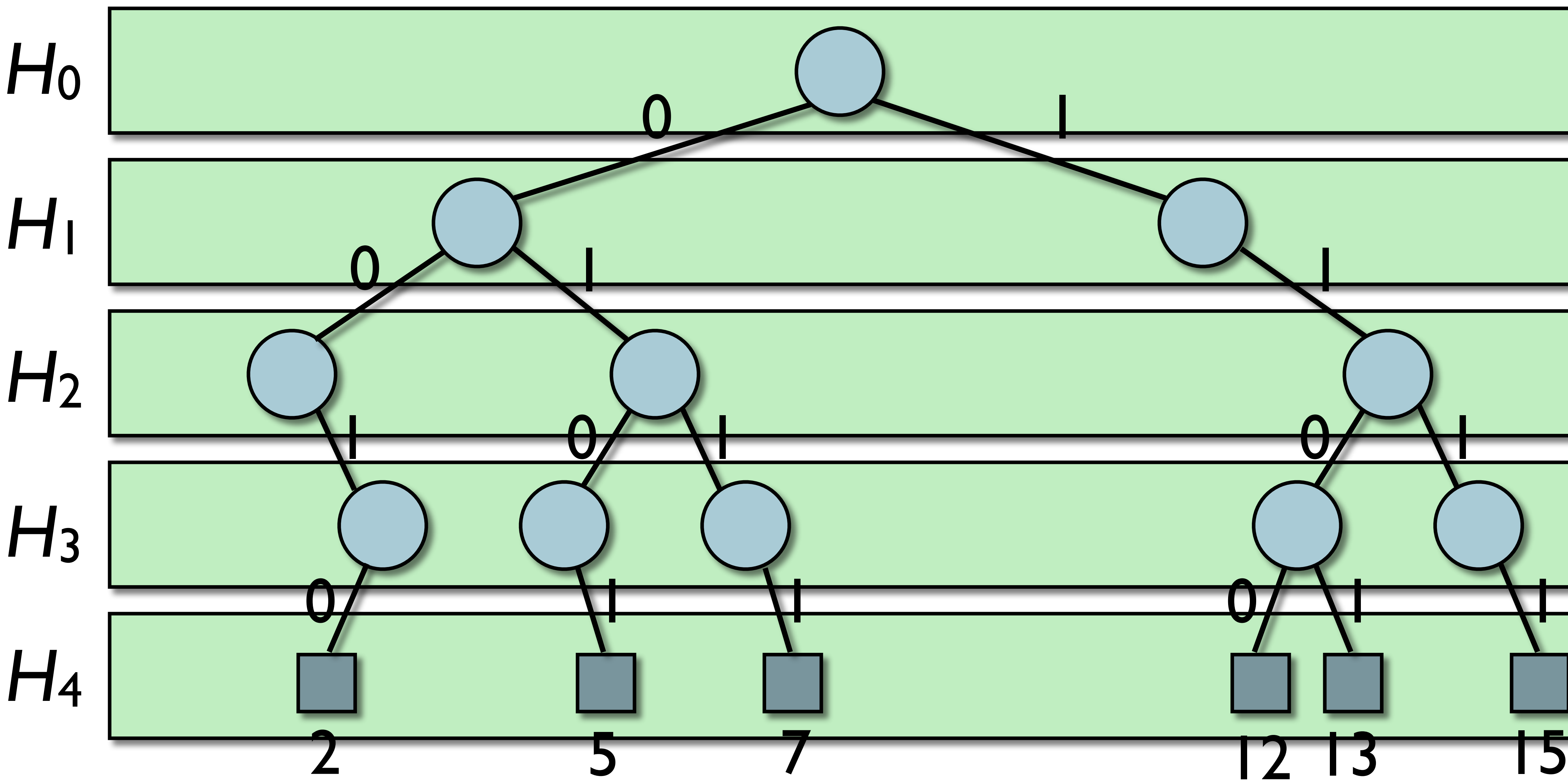
Idea

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



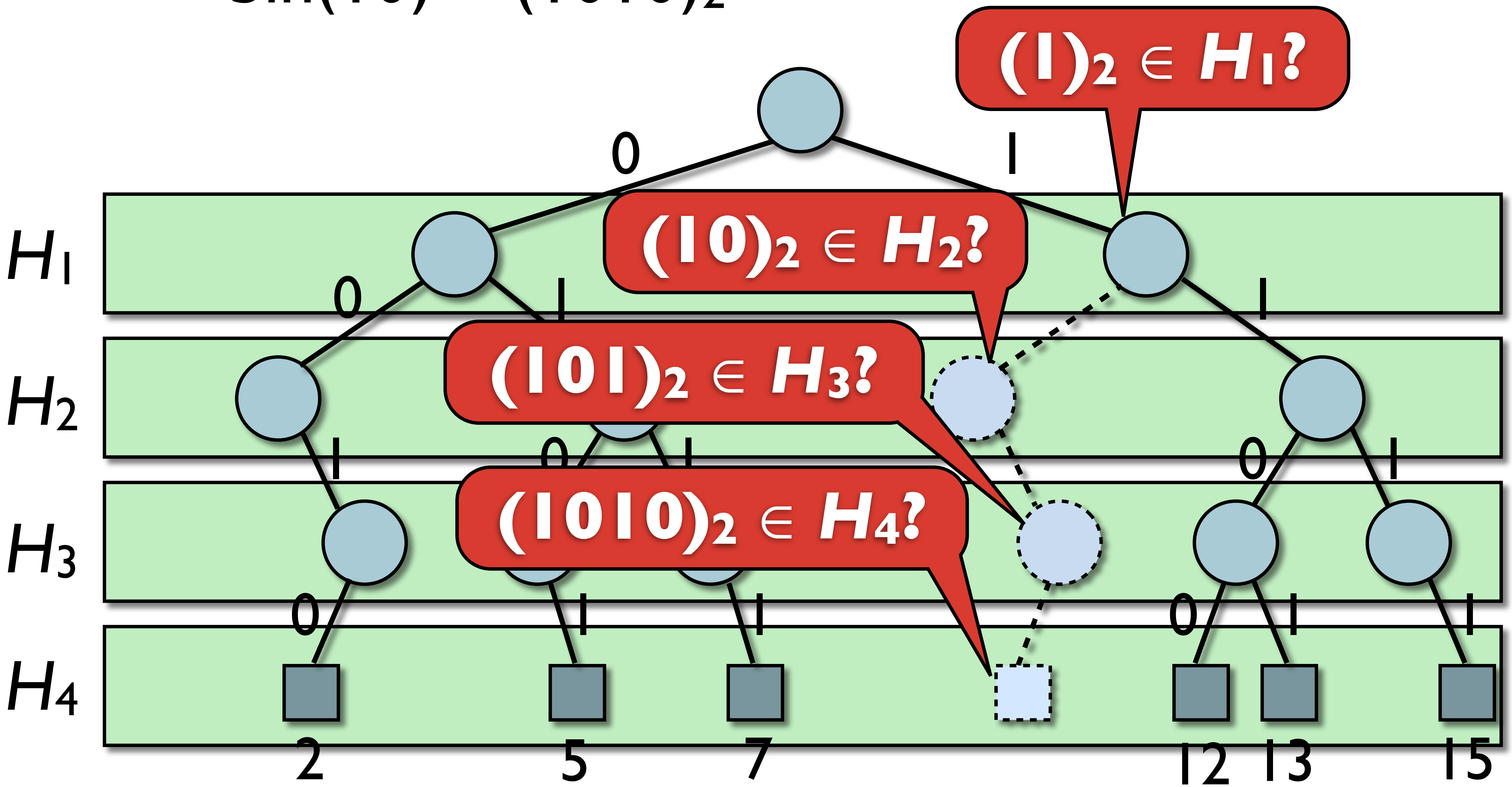
Idea

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



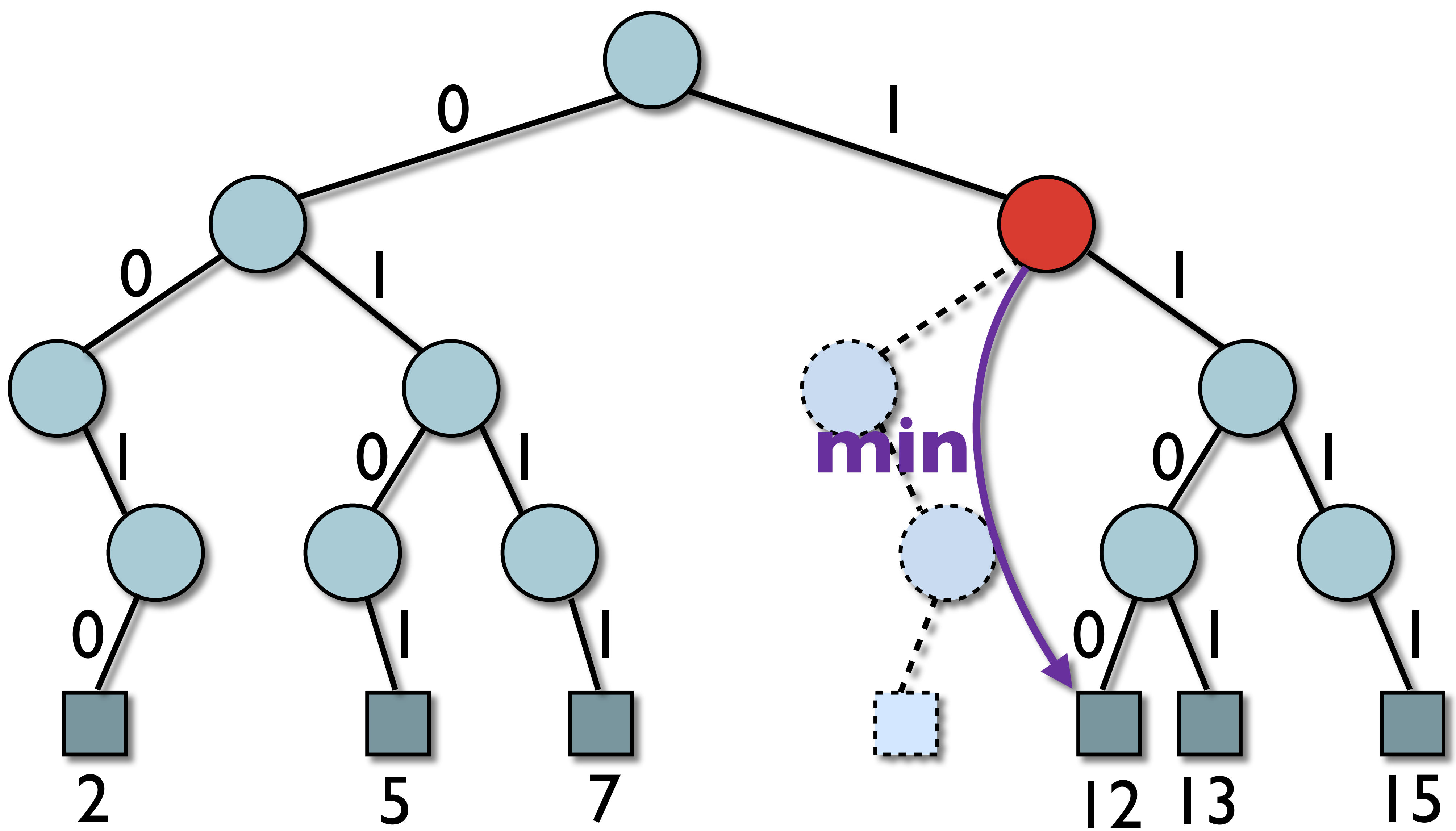
Successor Queries

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



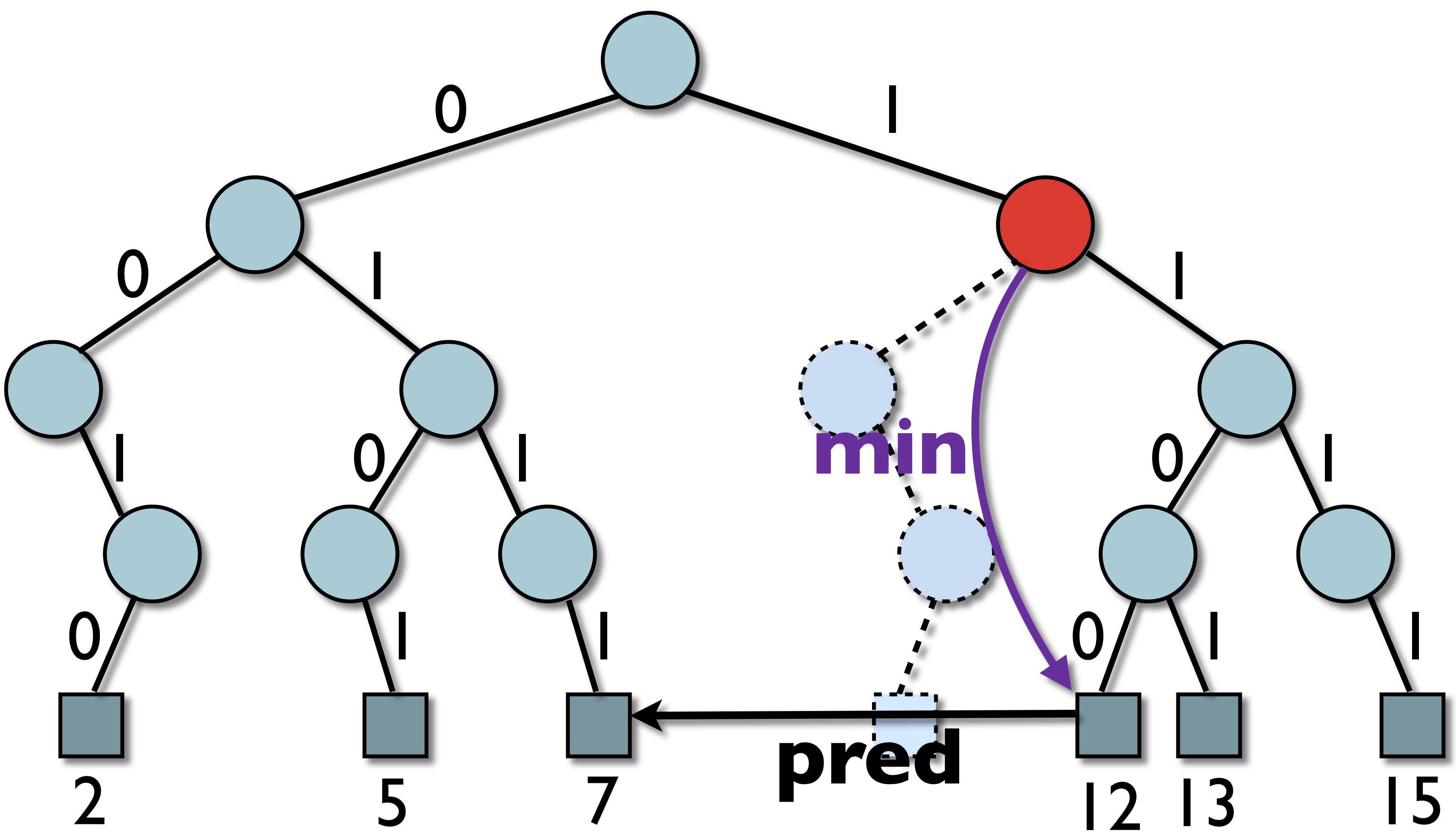
Successor Queries

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



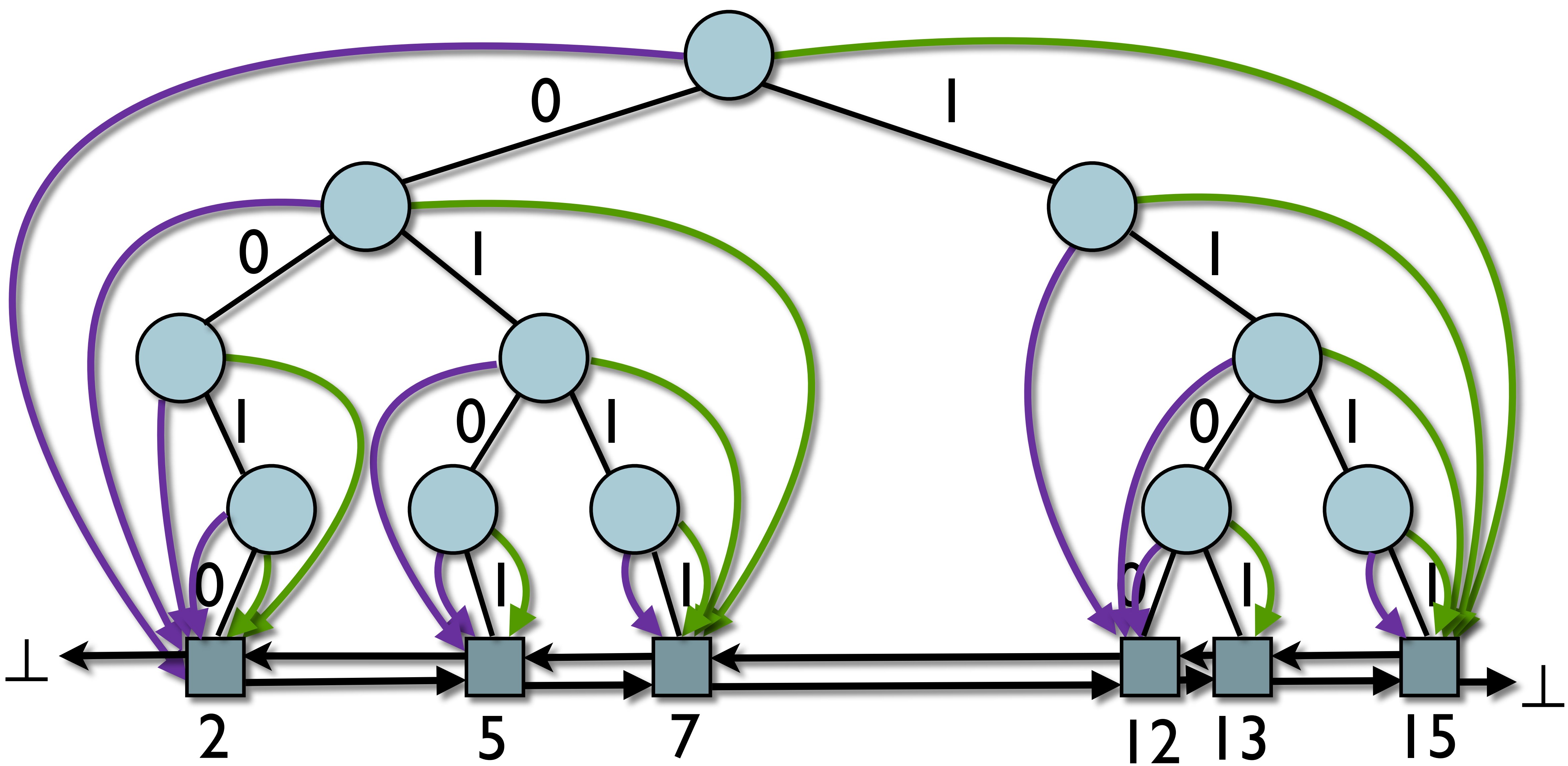
Predecessor Queries

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

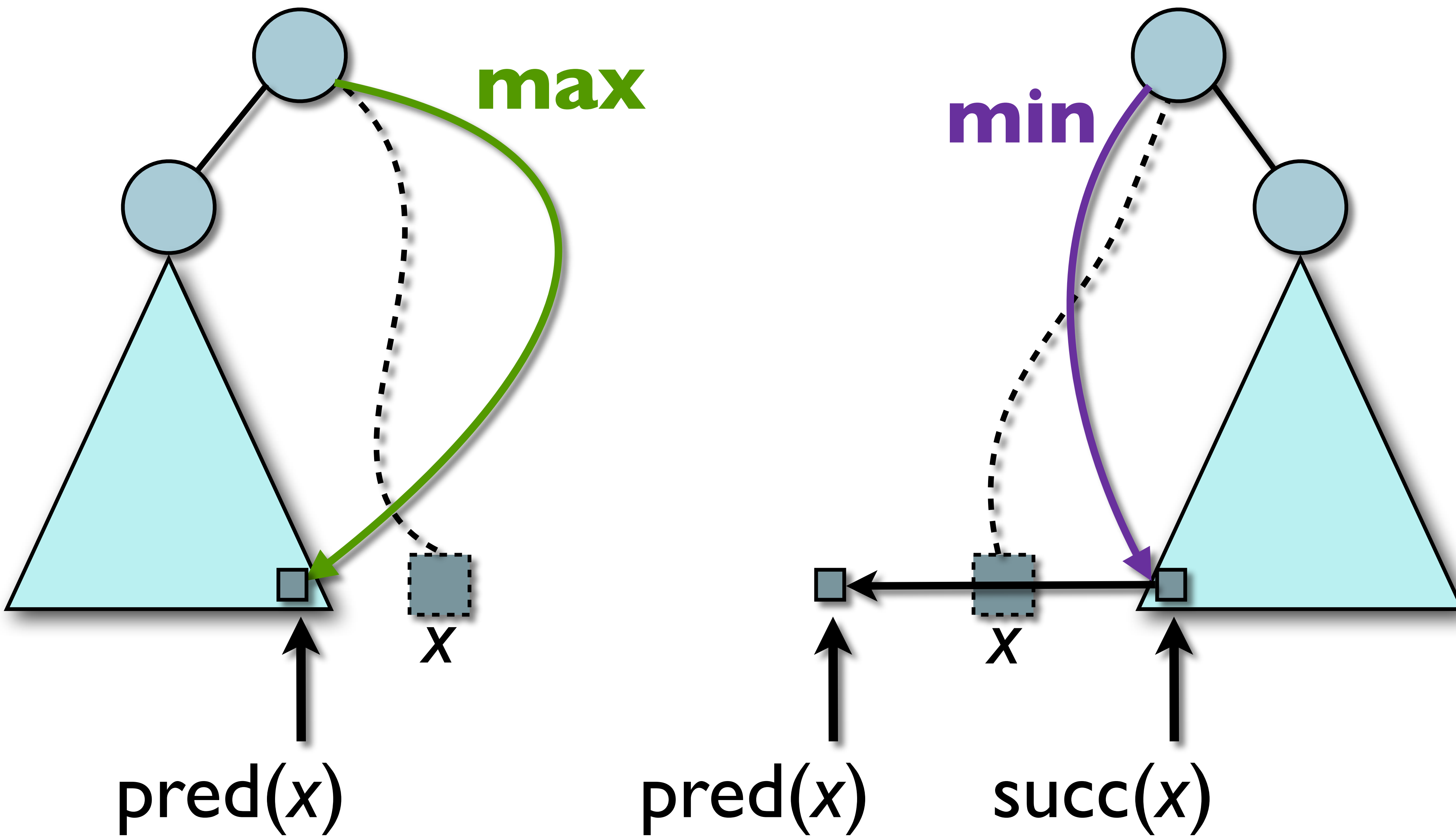


x-Fast Tries

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



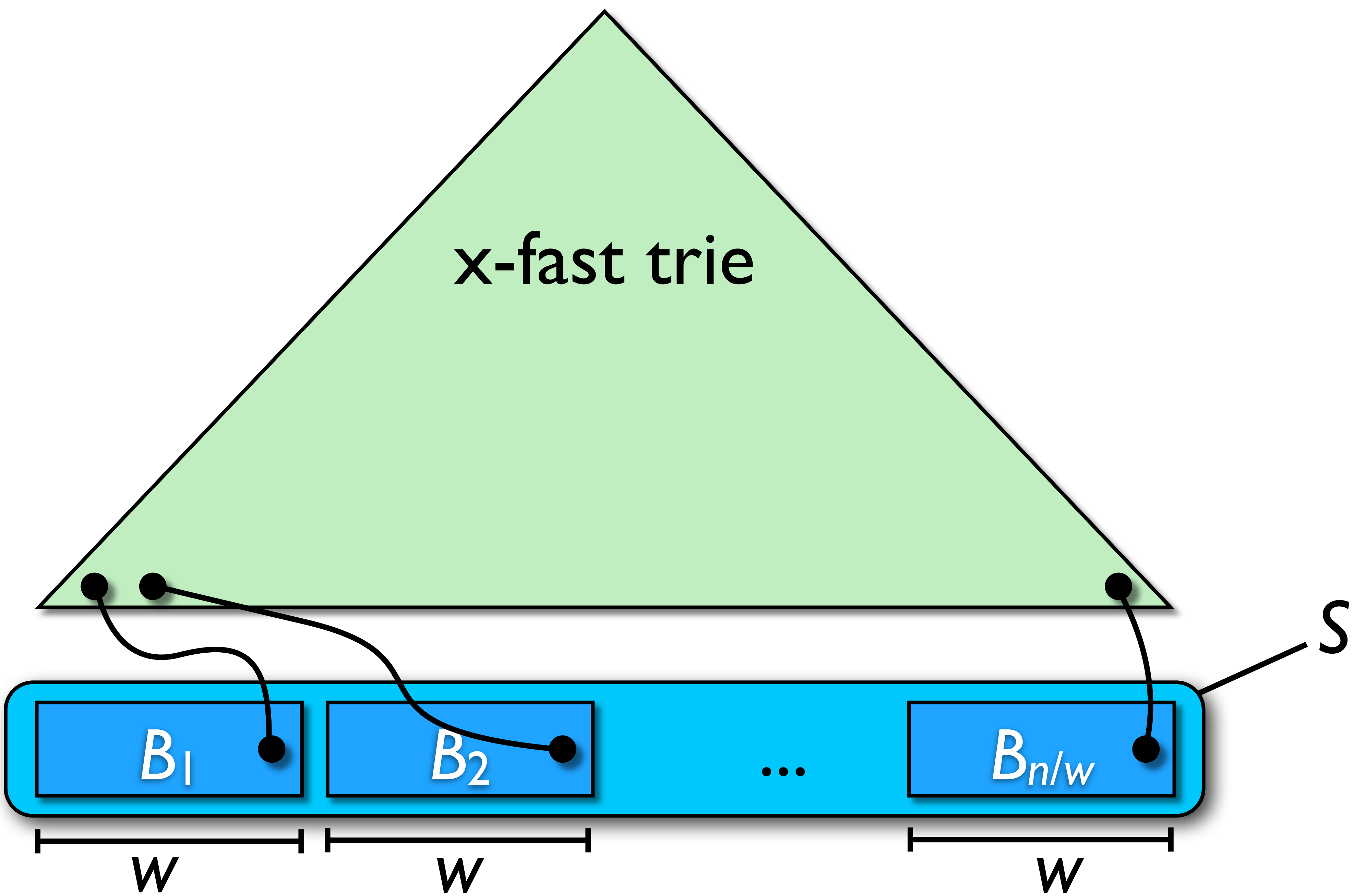
Predecessor Queries



The Final Picture

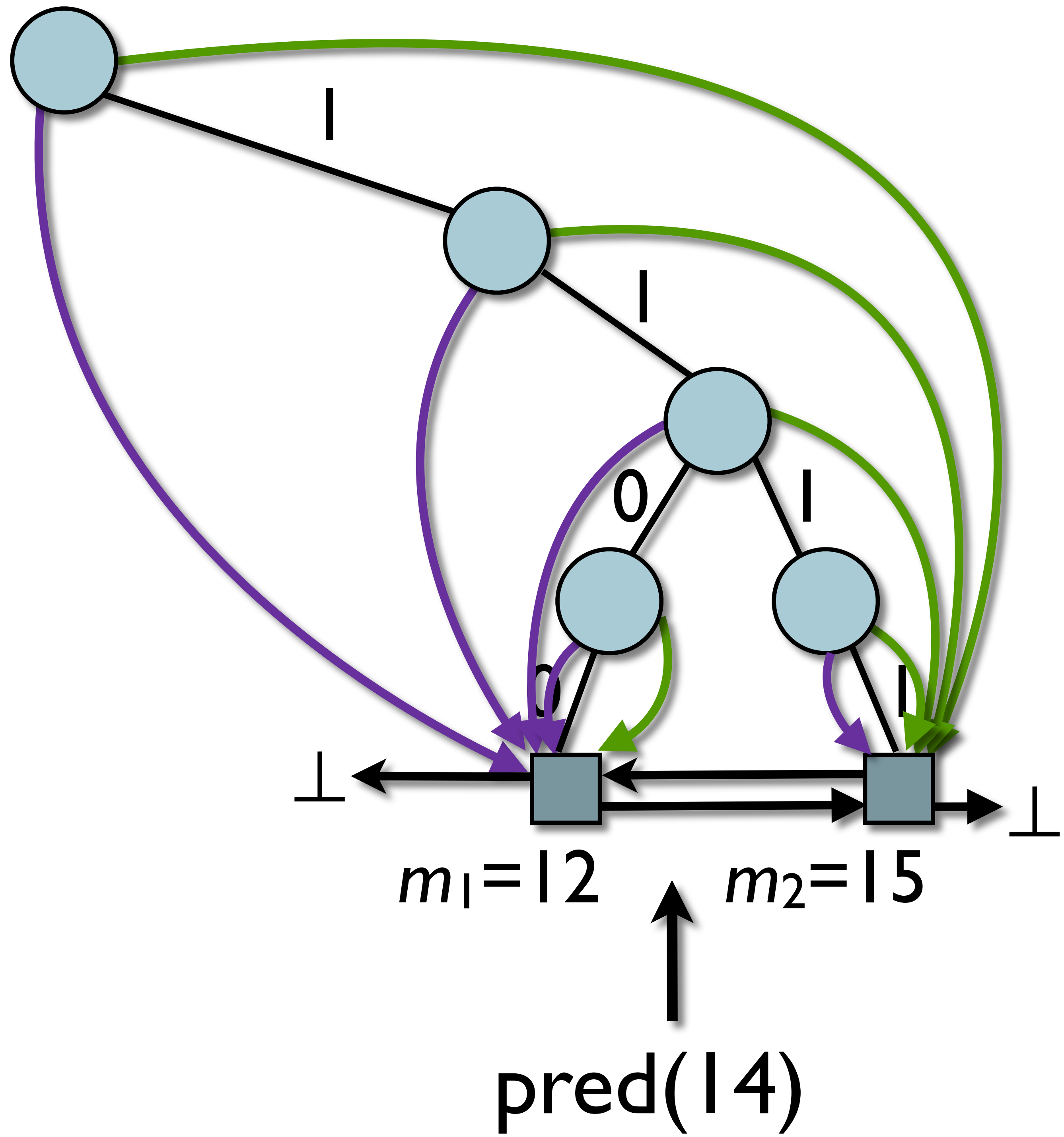
- predecessor $O(w)$, promised $O(\lg w)$
  - use **binary search** over heights → $O(\lg w)$
- space $O(nw)$, promised $O(n)$
  - use **indirection**:
  - blocks of w elements: $B_1,\ldots,B_{n/w}$ (sorted)
  - x-fast trie over $S'=\{m_i : 1 \leq i \leq n/w\}$, $m_i = \max B_i$
  - pred(x): (1) find 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.**

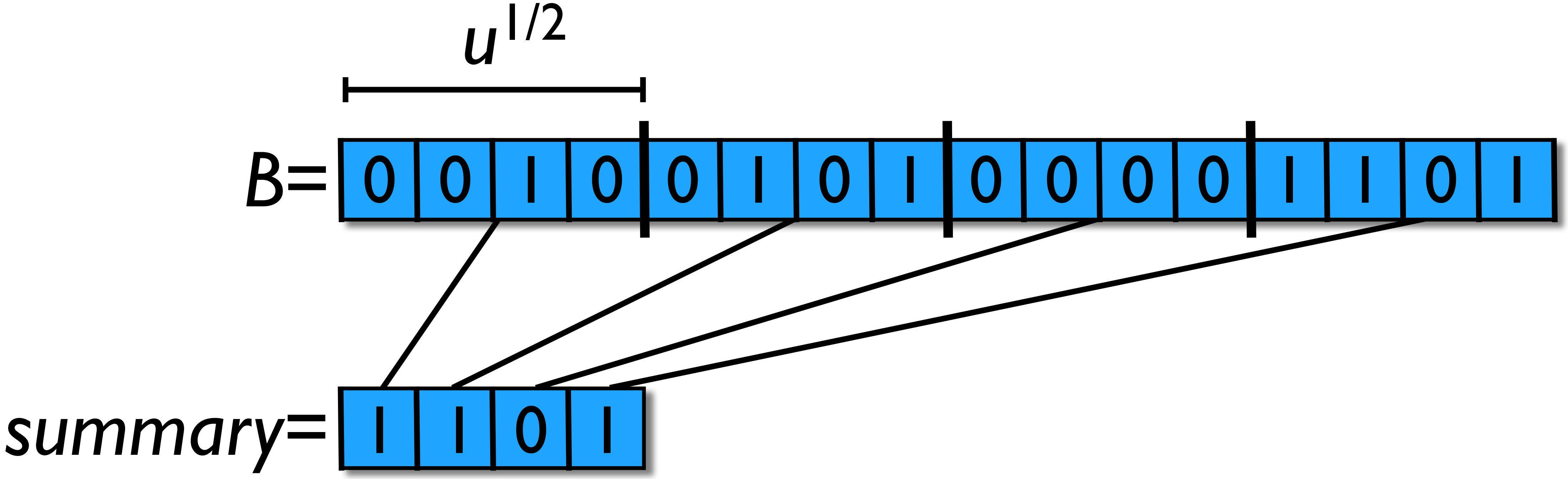
van Emde Boas Trees

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

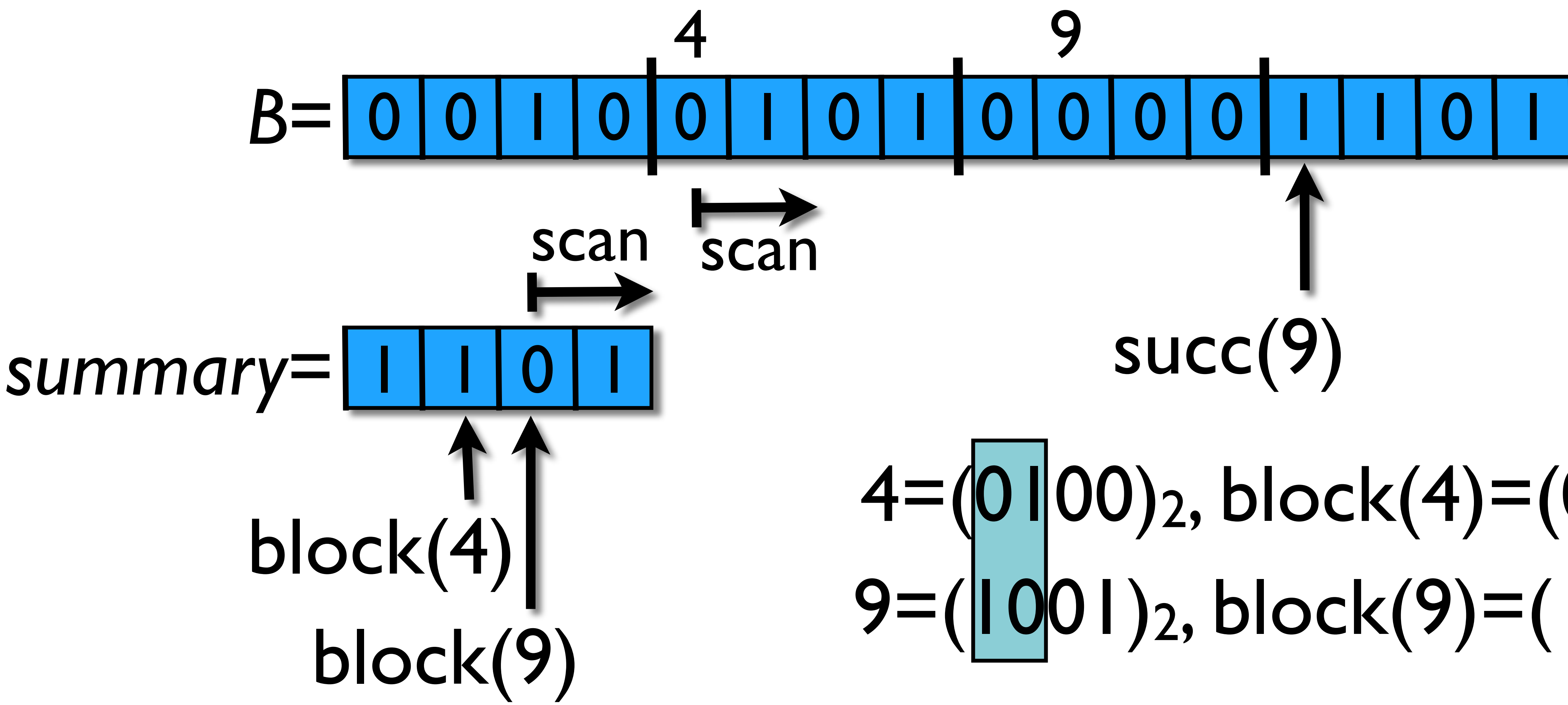
vEB trees	dynamic	
pred/succ	$O(\lg w)$ w.c.	$O(\lg w)$ w.c.
insert/delete	$O(\lg w)$ w.c.	$O(\lg w)$ exp. & amort.
space	$O(u)$ w.c.	$O(n)$ w.c.

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



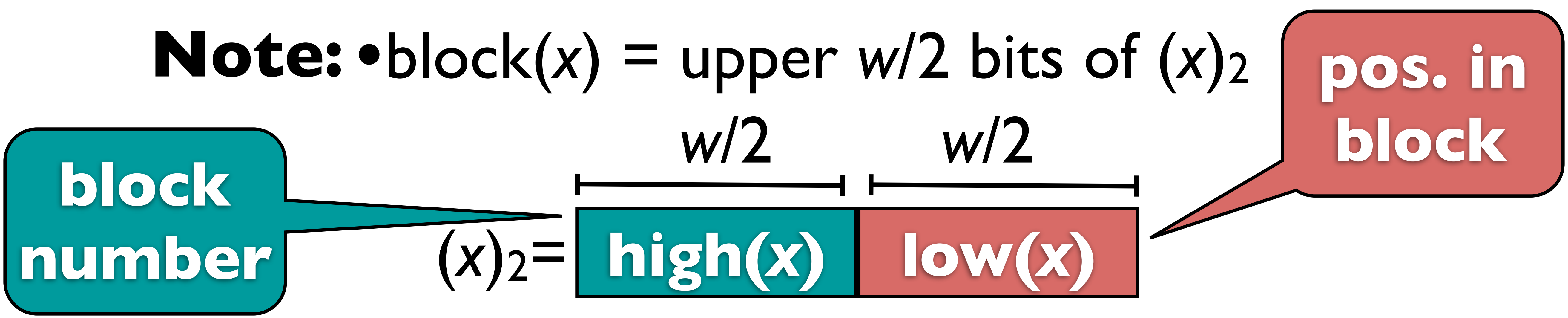
Finding Successors



$4=(0100)_2$, $\mathrm{block}(4)=(01)_2$

$9=(1001)_2$, $\mathrm{block}(9)=(10)_2$

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