Lecture 5: Fusion Trees (ctd.)

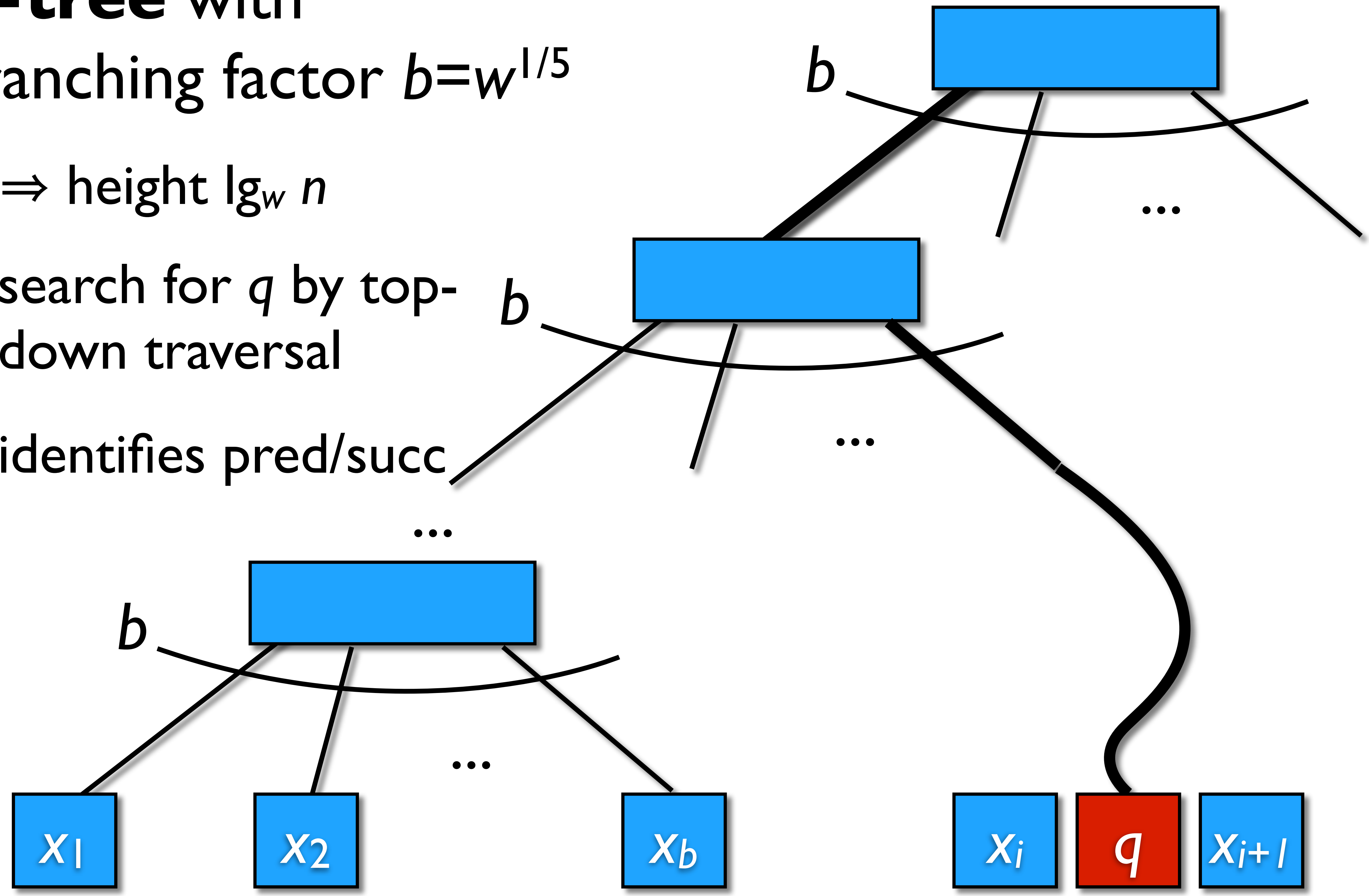
Johannes Fischer

Fusion Trees

fusion tree	
pred/succ	O(lg n / lg w) w.c.
construction	$O(n)$ w.c. + SORT(n,w)
space	$O(n)$ w.c.

Idea

- **B-tree** with branching factor $b = w^{1/5}$
  - ⇒ height $\lg_w n$
  - search for q by top-down traversal
  - identifies pred/succ



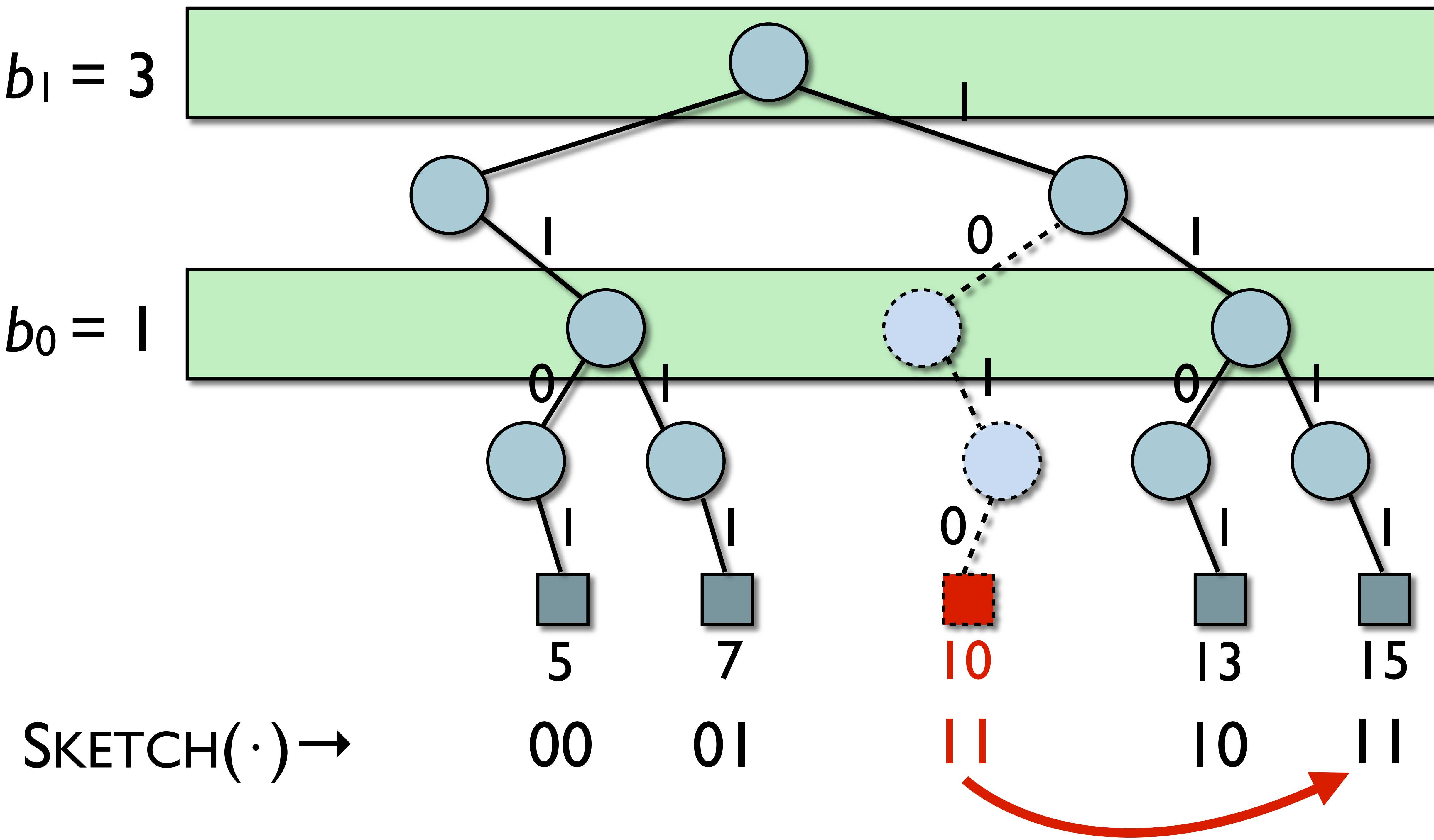
Sketches

- **important bits** $b_0 < \ldots < b_{r-1}$ for $x_0 < \ldots < x_{b-1}$: levels with **branching** nodes in trie
- SKETCH(x_i) = x_i restricted to important bits
- SKETCH **preserves order**:

  $$\text{SKETCH}(x_i) < \text{SKETCH}(x_j) \Leftrightarrow x_i < x_j$$

- problem: does **not** hold for query q: SKETCH(x_i) < SKETCH(q) $\leq$ SKETCH(x_{i+1}) $\not\Rightarrow x_i < q \leq x_{i+1}$

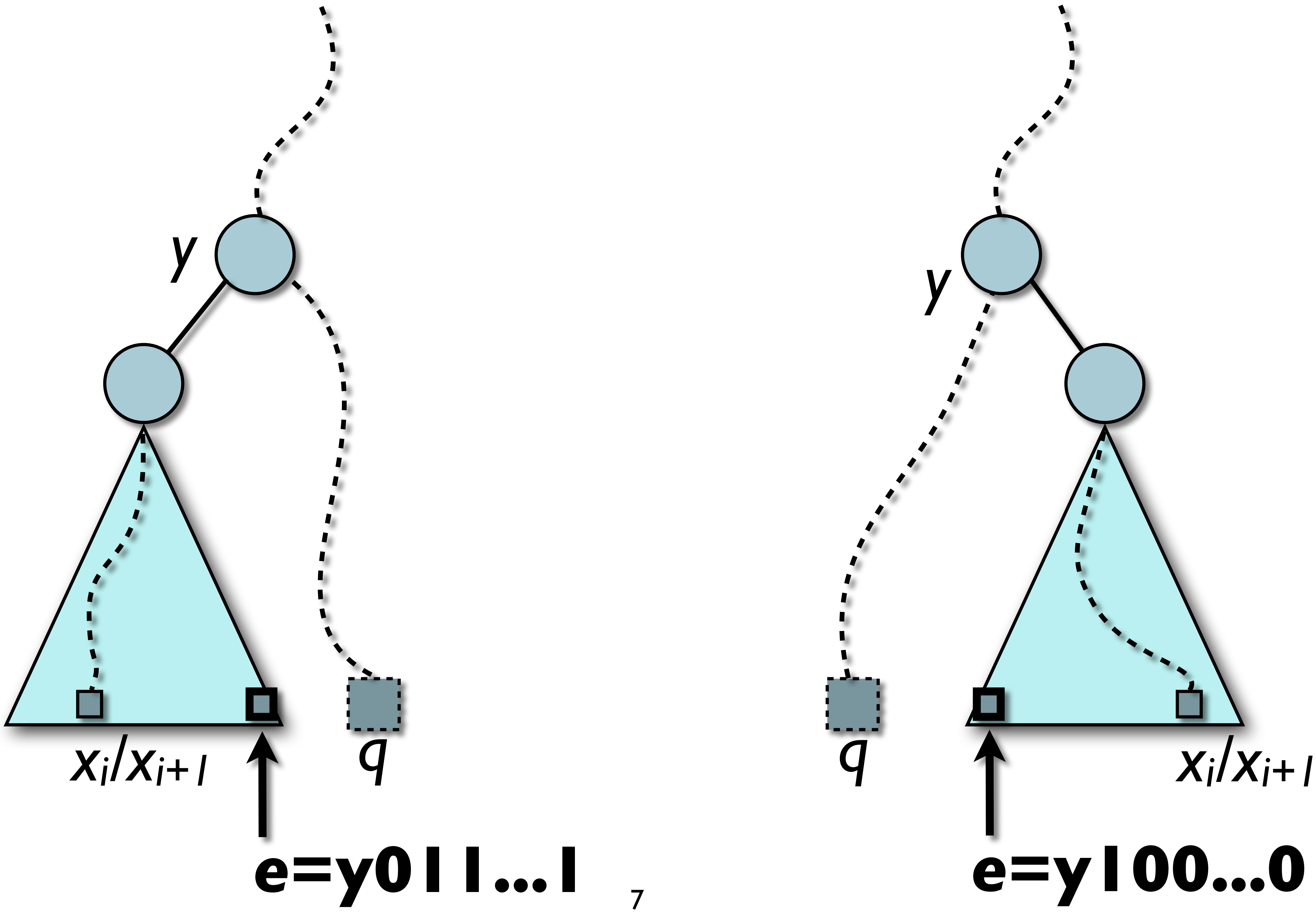
Example



Right Insertion Point

- SKETCH(x_i) ≤ SKETCH(q) ≤ SKETCH(x_{i+1})
- $y \leftarrow \max\{LCP(q,x_i), LCP(q,x_{i+1})\}$
  - point where q deviates from x_i/x_{i+1}
  - **most significant bit** of q XOR x_i/x_{i+1}
- subtree of y must contain pred **or** succ
  - repeat computation in that subtree

Right Insertion Point



Summary

- at every node of the B-tree with keys $x_0 < \ldots < x_{b-1}$ and important bits $b_0 < \ldots < b_{r-1}$
  - compute SKETCH(q) in $O(1)$ time
  - find i such that SKETCH(x_i) $\leq$ SKETCH(q) $\leq$ SKETCH(x_{i+1})
  - $y \leftarrow \max\{\text{MSB}(q \text{ XOR } x_i), \text{MSB}(q \text{ XOR } x_{i+1})\}$
  - $e \leftarrow q_{w-1} \ldots q_{y-1} 011\ldots1$ *or* $q_{w-1} \ldots q_{y-1} 100\ldots0$
  - compute SKETCH(e) in $O(1)$ time
  - find i' such that SKETCH($x_{i'}$) $\leq$ SKETCH(e) $\leq$ SKETCH($x_{i'+1}$)
  - then $x_{i'}$ or $x_{i'+1}$ is predecessor or successor of q

TODO 1

TODO 2

TODO 3

TODO 1: MSB in *O*(1) time

- possible, but **complicated** if not supported by CPU:

$$
\begin{aligned}
&t_1 \leftarrow h \,\&\, (x \mid ((x \mid h) - l)), \quad \text{where } h = 2^{g-1}l \text{ and } l = (2^n - 1)/(2^g - 1);\\
&y \leftarrow (((a \cdot t_1) \bmod 2^n) \gg (n-g)) \cdot l, \quad \text{where } a = (2^{n-g} - 1)/(2^{g-1} - 1);\\
&t_2 \leftarrow h \,\&\, (y \mid ((y \mid h) - b)), \quad \text{where } b = (2^{n+g} - 1)/(2^{g+1} - 1);\\
&m \leftarrow (t_2 \ll 1) - (t_2 \gg (g-1)),\ m \leftarrow m \oplus (m \gg g);\\
&z \leftarrow (((l \cdot (x \,\&\, m)) \bmod 2^n) \gg (n-g)) \cdot l;\\
&t_3 \leftarrow h \,\&\, (z \mid ((z \mid h) - b));\\
&\lambda \leftarrow ((l \cdot ((t_2 \gg (2g - \lg g - 1)) + (t_3 \gg (2g-1)))) \bmod 2^n) \gg (n-g).
\end{aligned}
$$

© D.E. Knuth: *The Art of Computer Programming*, Vol. 4A. Addison-Wesley, 2011.

- note the use of 5 **multiplications**

MSB in $O(\lg w)$ time

- idea:
  - binary search over x
  - with ANDs determine if search interval empty

x= 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 1

$mask$= 1 1 1 1 1 1 1 1

x AND $mask$= 0 0 0 0 0 1 0 1 $\neq 0 \Rightarrow$ continue in left half

MSB in $O(\lg w)$ time

- idea:
  - binary search over x
  - with ANDs determine if search interval empty

x=	0	0	0	0	0	1	0	1	0	0	0	0	1	1	0	1
$mask$=	1	1	1	1												
x *AND* $mask$=	0	0	0	0												

x *AND* $mask$ $=0 \Rightarrow$ continue in right half

MSB in $O(\lg w)$ time

- idea:
  - binary search over x
  - with ANDs determine if search interval empty

x= | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

$mask$= | 1 | 1 |

x AND $mask$= | 0 | 1 | $\neq 0 \Rightarrow$ continue in left half

etc.

MSB in $O(\lg w)$ time

- neat variation of this idea:

```
function MSB(x):
    λ ← 0
    for k=lg(w)-1 downto 0
        z ← x ≫ 2^k
        if (z ≠ 0)
            λ ← λ + 2^k
            x ← z
    return λ
```

- possible: **table lookup** for small blocks

Implementation

```
unsigned int x;  // 32-bit value to find the log2 of
const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00,
                          0xFFFF0000};
const unsigned int S[] = {1, 2, 4, 8, 16};
int i;

register unsigned int l = 0; // result will go here
for (i = 4; i >= 0; i--) // unroll for speed...
{
  if (x & b[i])
  {
    x >>= S[i];
    l |= S[i];
  }
}
```

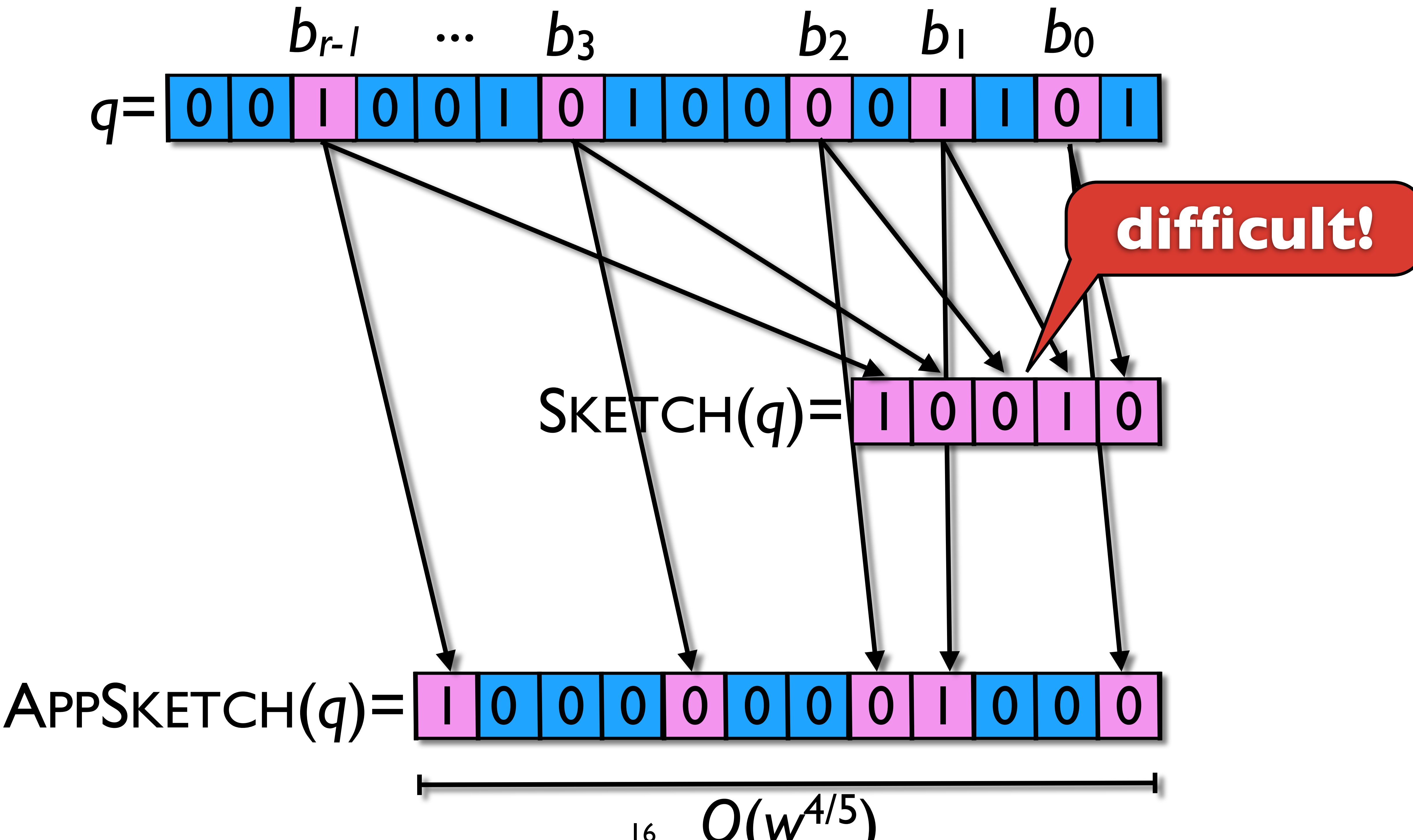
see `http://www-graphics.stanford.edu/~seander/bithacks.html`

Implementation with Table Lookup

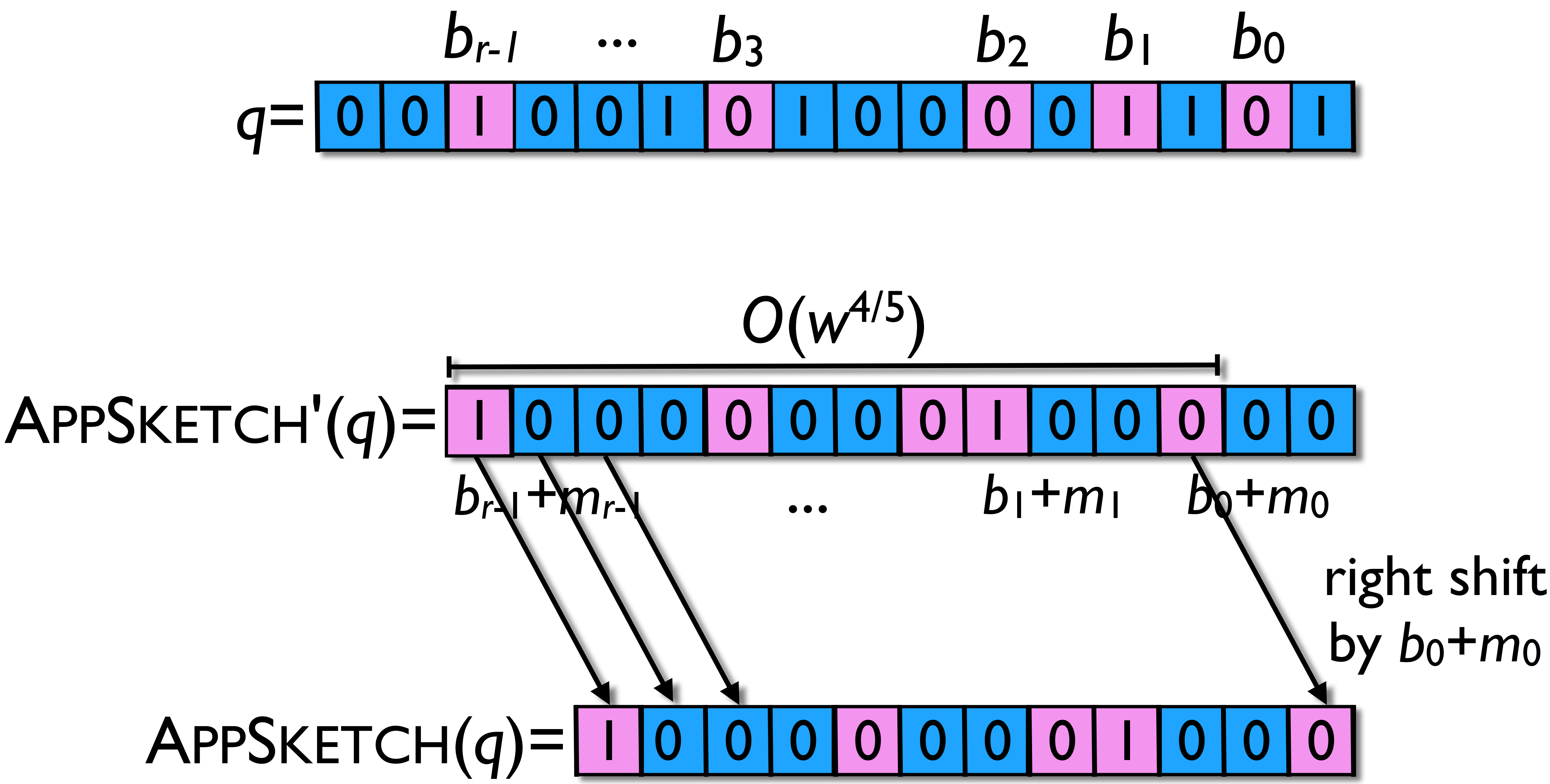
```
unsigned int l = 0;           // will be lg(x)
register unsigned int t, tt; // temporaries
if (tt = x >> 16)
   l = (t = x >> 24) ? 24 + LogTable256[t]
                     : 16 + LogTable256[tt & 0xFF];
else
   l = (t = x >> 8) ? 8 + LogTable256[t] : LogTable256[x];
return l;
```

```
const char LogTable256[256] =
      {
            0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,
            4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
            5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
            5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
            6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
            6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
            6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
            6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
      };
```

TODO 2: Computing SKETCH(q)

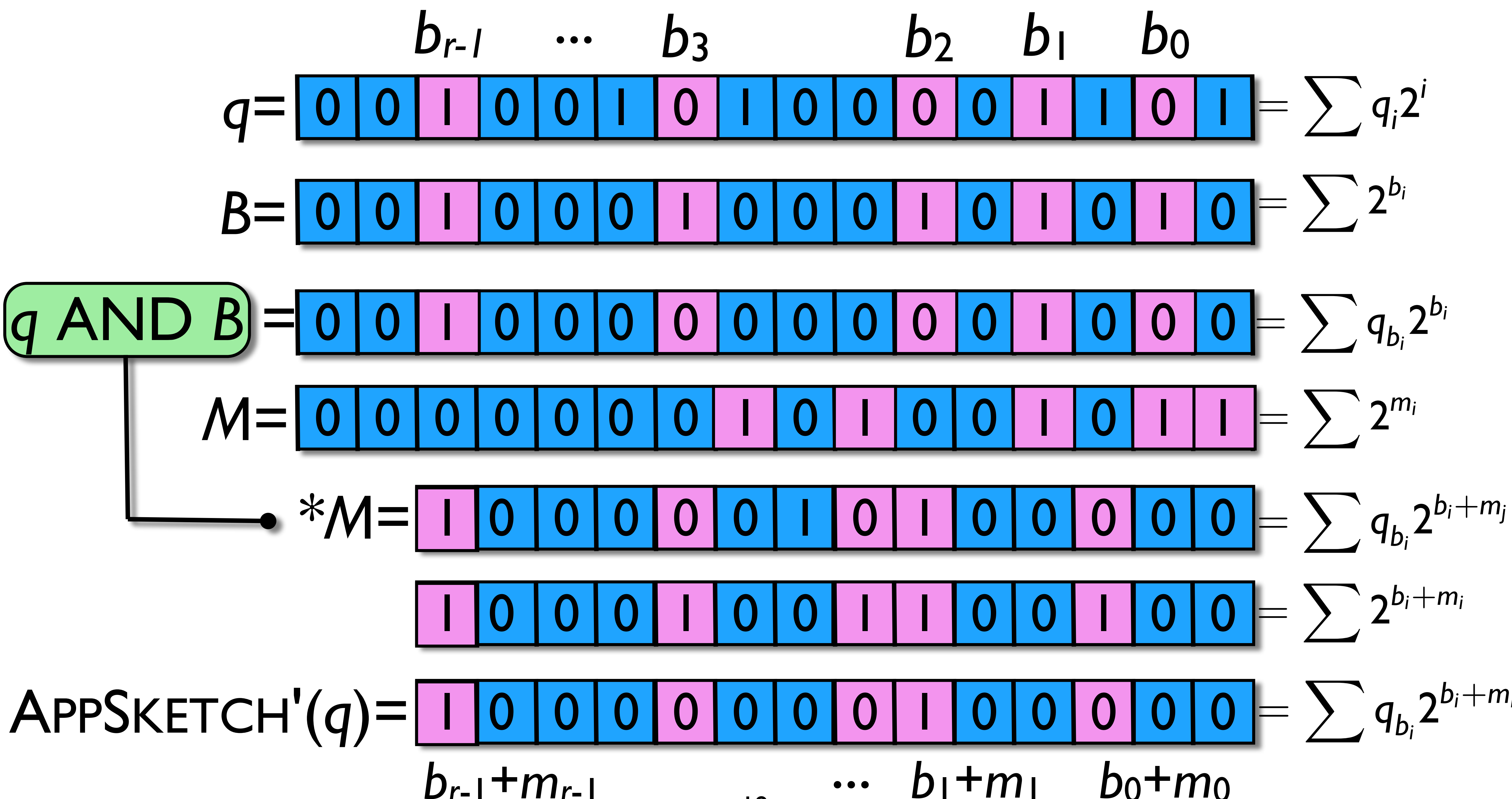


APPSKETCH(q)



b_{r-1} … b_3 b_2 b_1 b_0
q=
0 0 1 0 0 1 0 1 0 0 0 0 1 1 0 1
O(w^{4/5})
APPSKETCH'(q)=
1 0 0 0 0 0 0 0 1 0 0 0 0 0
b_{r-1}+m_{r-1} … b_1+m_1 b_0+m_0
right shift
by b_0+m_0
APPSKETCH(q)=
1 0 0 0 0 0 0 0 1 0 0 0

APPSKETCH'(q)


b_{r-1} … b_3 b_2 b_1 b_0
q= 0 0 1 0 0 1 0 1 0 0 0 0 1 1 0 1 = ∑ q_i 2^i
B= 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 = ∑ 2^{b_i}
q AND B = 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 = ∑ q_{b_i} 2^{b_i}
M= 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 1 = ∑ 2^{m_i}
*M= 1 0 0 0 0 0 1 0 1 0 0 0 0 0 = ∑ q_{b_i} 2^{b_i+m_j}
1 0 0 0 1 0 0 1 1 0 0 1 0 0 = ∑ 2^{b_i+m_i}
APPSKETCH'(q)= 1 0 0 0 0 0 0 0 1 0 0 0 0 0 = ∑ q_{b_i} 2^{b_i+m_i}
b_{r-1}+m_{r-1} … b_1+m_1 b_0+m_0

Requirements on m_i's

- $\sum_{i,j} q_{b_i} 2^{b_i+m_j}$ should contain bits b_i of q at positions $b_i + m_j \Rightarrow$ **no overflow**

  a) $b_i + m_j = b_k + m_l \Leftrightarrow i=k$ and $j=l$

  b) $b_0 + m_0 < b_1 + m_1 < ... < b_{r-1} + m_{r-1}$

  c) $(b_{r-1} + m_{r-1}) - (b_0 + m_0) = O(w^{4/5})$

Existence of m_i's

- strong (a): distinctness **modulo r^3**:
  $b_i + m'_j \equiv b_k + m'_l \pmod{r^3} \Leftrightarrow i=k$ and $j=l$
  - proof by induction:
    - for m'_j **avoid** $m'_l + (b_k - b_i) \pmod{r^3}$
    - for all $0 \leq l < j$, $0 \leq k,i < r \Rightarrow jr^2 < r^3$
    - r^3 possible choices $\Rightarrow m'_j$ exists
- $\Rightarrow$ can make $b_i + m_i$ lie in different blocks of length r^3: $m_j = m'_j + (\lfloor (w-b_j+jr^3)/r^3 \rfloor \cdot r^3)$

20

Existence of m_i's

- Requirements satisfied:

  a) $b_i + m_j$ distinct modulo $r^3 \Rightarrow$ distinct overall

  b) $b_i + m_i$ lie in consecutive r-blocks
  $\Rightarrow b_0 + m_0 < b_1 + m_1 < \ldots < b_{r-1} + m_{r-1}$

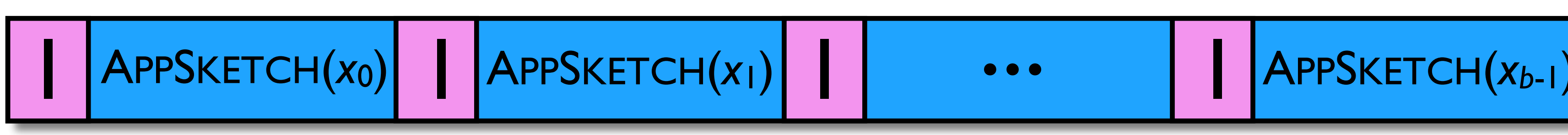
  c) $b_0 + m_0 = w$
  $b_i + m_i \leq b_{i-1} + m_{i-1} + r^3$
  $r<w$
  $\Rightarrow (b_{r-1} + m_{r-1}) - (b_0 + m_0) \leq w+r^4 = O(w^{4/5})$

TODO 3: Finding q's Insertion Point

- B-tree node v with $x_0 < \ldots < x_{b-1}$: find i s.th. APPSKETCH(x_i) $\leq$ APPSKETCH(q) $\leq$ APPSKETCH(x_{i+1})
- $b = O(w^{1/5})$ sketches of size $O(w^{4/5})$ $\Rightarrow$

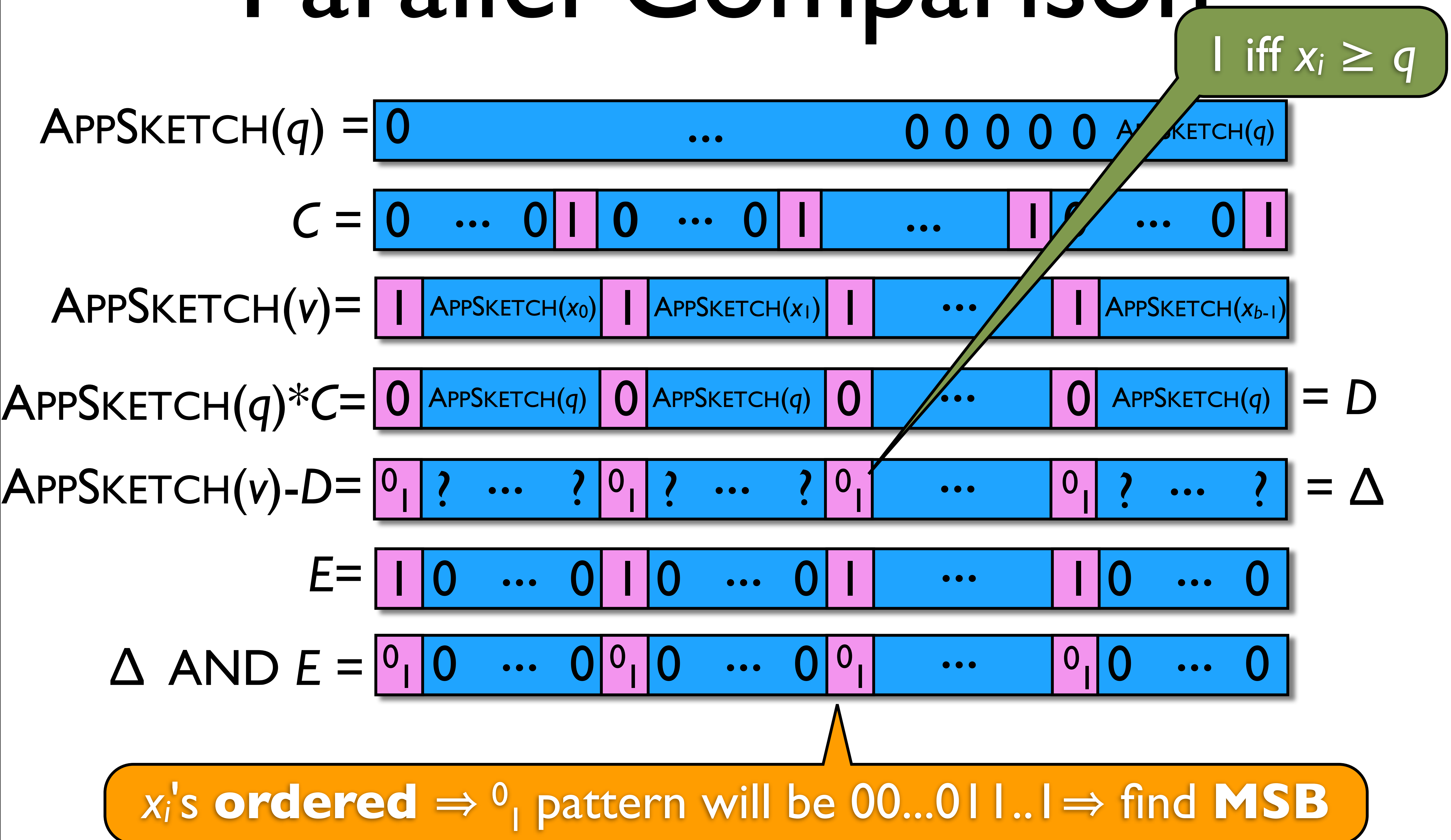
  b sketches **fused**=packed in **O(1) words**
- separate by 1-bits:

APPSKETCH(v)=



22

Parallel Comparison



Summary

- multiple keys per B-tree node in $O(1)$ time
- **broadword computing**:
    - subtraction: parallel comparison
    - AND: masking
    - multiplication:
        - create copies
        - bit permutation
- $O(\lg n / \lg w)$: **beat information theoretic barrier** of $O(n \lg n)$