The Word-RAM and Succinct Data Structures

Efficient Data Structures

Summer 2014

Pat Nicholson

Issues with Implicit Model

- Some drawbacks of the implicit data structure model:
  - The space requirements are overly strict
  - Only comparisons are allowed

- How do *real* computers work?
  - Modern computer architectures deal with *words:*
    - Typically, each word consists of between 32 and 64 *bits*
    - No matter what is being represented it really is just bits
  - Our the model *should* be able to address individual bits

Next Model: The Word-RAM

- Word-RAM memory is of an array of w bit words
  - The *space cost* is the number of words stored
  - The *space cost in bits* is:

    w × number of words stored
  - The *time cost* is the number of *word operations*:

    reads/writes/arithmetic operations*

It is natural to assume that $w = \Omega(\log n)$ since we can’t follow pointers efficiently otherwise.

Drawbacks of the Word-RAM

- Does not consider the memory hierarchy
  - Caching effects are very important in practice
    - Scanning vs. random access
- When combined with big-Oh it can be misleading
  - $\Theta\left(\frac{\log n}{\log \log n}\right)$ is asymptotically smaller than $\Theta(\log n)$…
  - However, $\frac{10 \log n}{\log \log n} > \log n$ for all reasonable values of n

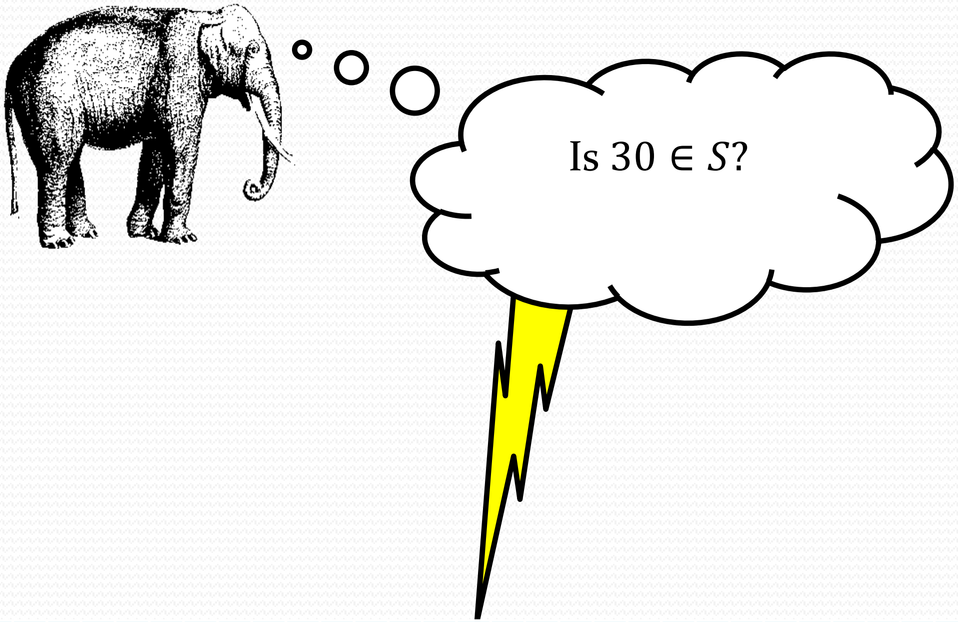
Static Membership

- Recall that in the implicit data structure model described, the static membership problem has a lower bound of $\Theta(\log n)$ time *(due to comparison restriction)*

- Let's look at the problem in the word-RAM:
  - Reasonable assumption: element occupies $\Theta(1)$ words
    - What does this mean in terms of its values?
    - We can assume there is some upper bound u on the max:
      - $\Theta(1)$ words → Elements in range $[0,2^{\Theta(w)}-1]$
      - $u \leq 2^{\Theta(w)}$

Totally Naïve Solution: A Bit Vector

- Given our set S
  - Store a **bit vector** of size u bits:
  - Bit $x \in [0, u-1]$ associated with element x
  - If $x \in S$ set x to 1, otherwise set it to 0

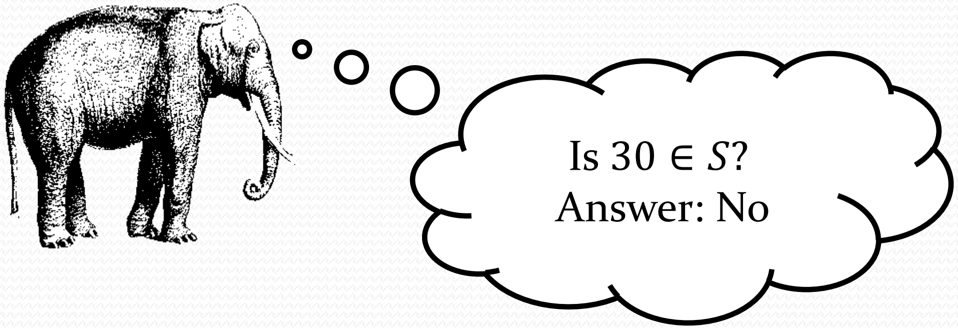


1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

Universe [0,49]

Totally Naïve Solution: A Bit Vector

- Given our set S
  - Store a **bit vector** of size u bits:
  - Bit $x \in [0, u - 1]$ associated with element x
  - If $x \in S$ set x to 1, otherwise set it to 0



Is 30 ∈ S?
Answer: No

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

Universe [0,49]

Bit Vector: Analysis

- Member takes $\Theta(1)$ time
  - Need only look at a single bit (can even do updates)
- Downside: the space usage
  - This occupies $\Theta(u)$ bits…
    - … and u doesn’t necessarily have any relationship with n
    - Usually we want space (in words) to be some function of n
      - Sorted table: n words or, alternatively, $n \lceil \log u \rceil$ bits

- Can we do $\Theta(1)$ time Member queries in $\Theta(n)$ words?

A Useful Hashing Fact

- Hash function $h: U_1 \to U_2$ is *universal* if:

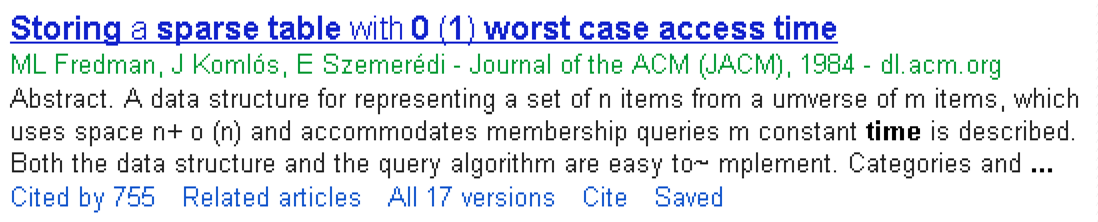
  For any distinct $x, y \in U_1$ we have $\Pr[h(x) = h(y)] \leq \frac{1}{U_2}$

  (Carter and Wegman, 1979)

- Suppose we hash into a quadratic sized table:
  - Let $h\colon U \mapsto n^2$ be a universal hash function
  - What is the probability of having *any* collisions?

    Pr[Some pair of elements collide] < $\#Pairs/U_2 =$
    $n(n-1)/2n^2 < 1/2$
  - Just keep generating such hash functions until it works

- This is (similar to) the *birthday paradox* (23 people in a room)
  - Basis of:



Storing a **sparse table** with **0 (1) worst case access time**

ML Fredman, J Komlós, E Szemerédi - Journal of the ACM (JACM), 1984 - dl.acm.org

Abstract. A data structure for representing a set of n items from a umverse of m items, which uses space n+ o (n) and accommodates membership queries m constant **time** is described. Both the data structure and the query algorithm are easy to~ mplement. Categories and **...**

FKS Hashing: The Big Idea

- Hash all the keys into a table of size n with u.h.f.
  - Let n_i be the number of elements in location i
  - Let $c_{x,y} = 1$ if x collides with y and 0 otherwise
- <u>*Claim*</u>: $\Pr[\sum_i n_i^2 > 4n] < 1/2$

  Proof:

$$E\left[\sum_i n_i^2\right] = E\left[\sum_x \sum_y c_{x,y}\right] =$$

$$n + 2n(n-1)/2n < 2n$$

Markov

Apply my inequality: $\Pr[X \geq a] \leq E[X]/a$

FKS Hashing: Summary

- What does this mean?
  - Hash into a table of size n, then hash each bucket again
  - Easy to build the data structure: expected linear time
  - Shows the power of bitwise operations

- This idea of having multiple *levels* is quite common
  - AKA: Keep doing the trick until it works
  - It is heavily used in *Succinct Data Structures*

How much space do we really need?

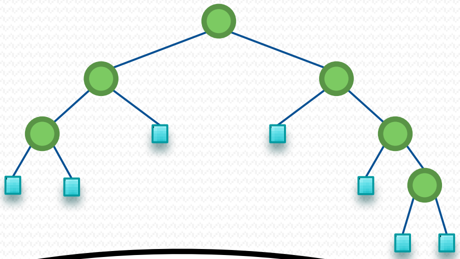
- In the word-RAM model it is not necessarily clear…

- It turns out there is a simple enumerative way:
  1. Figure out what kind of object we want to represent
  2. Figure out how many objects there are of that type
  3. Take the log (base 2) of this number

- This is known as the *information theoretic lower bound*

Information Theory Lower Bound

- Example: Represent full binary trees with $n + 1$ leaves



Catalan

There are about $\dfrac{4^{n}}{n^{\frac{3}{2}}\,\pi^{\frac{1}{2}}}$ of those

- This means we need only $2n - \Theta(\log n)$ bits to represent a tree
  - But if each node has 2 pointers, we are using $2n \log n$ bits…
    - Depending on the type of tree this could be 64 times bigger in practice

Succinct Data Structures

- Main Idea in Combinatorial Enumeration:
  - Count the number of objects of type χ

- Main Idea in Succinct Data Structures
  - Represent object of type χ using $\log|\chi| + o(\log|\chi|)$ bits
  - Support efficient queries on the object

- Our Full Binary Tree Example:
  - How to we represent our tree using $2n + o(n)$ bits…
  - … and support efficient navigation:
    - E.g., move to parent, move to children, return subtree size, etc.

Technical Considerations: Arrays

- We can use shifting to deal with word boundaries
  - Store n numbers, each b-bits, using $\left\lceil \frac{bn}{w} \right\rceil$ bits
    - Thus, we don't waste space
    - $\Theta(1)$ slowdown for accessing the elements

- How big are pointers?
  - We can resize our pointers to use less space
    - General idea: pointers don't need to occupy an entire word
  - Even better: given context, often can use "short" pointers

Fundamental Tool: Rank and Select

- Suppose we are given a bit vector of length u:

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1

- How can we support the following operations:
  - Rank(i): return the number of ones up to position i

Fundamental Tool: Rank and Select

- Suppose we are given a bit vector of length u:

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

- How can we support the following operations:
  - Rank(i): return the number of ones up to position i
    - Example: Rank(20) $= ?$

Fundamental Tool: Rank and Select

- Suppose we are given a bit vector of length u:

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

- How can we support the following operations:
  - Rank(i): return the number of ones up to position i
    - Example: Rank(20) $=$ 5

Fundamental Tool: Rank and Select

- Suppose we are given a bit vector of length u:

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

- How can we support the following operations:
  - Rank(i): return the number of ones up to position i
    - Example: Rank(20) $=$ 5
  - Select(j): return the position of the j-th one
    - Example: Select(7) $=$?

Fundamental Tool: Rank and Select

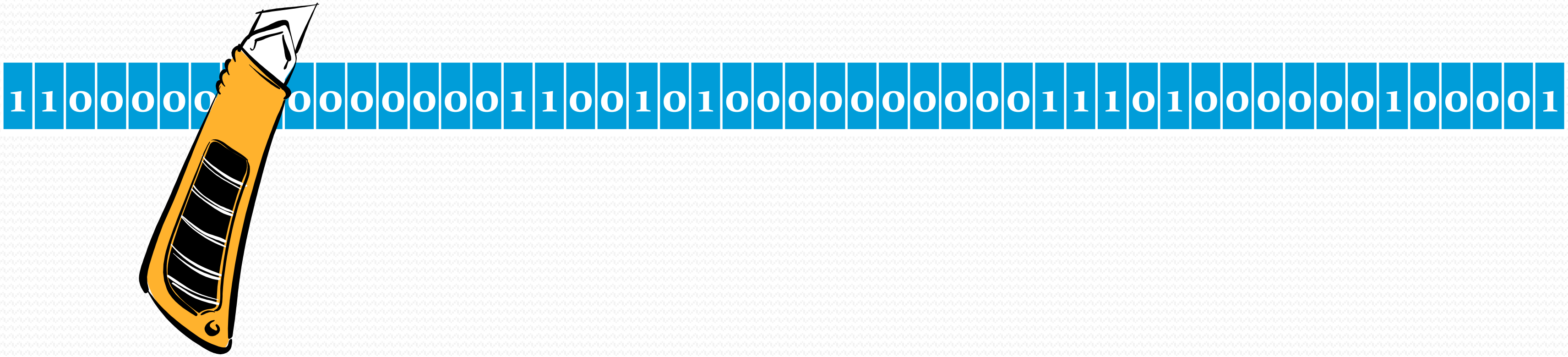
- Suppose we are given a bit vector of length u:

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1

- How can we support the following operations:
  - Rank(i): return the number of ones up to position i
    - Example: Rank(20) $=$ 5
  - Select(j): return the position of the j-th one
    - Example: Select(7) $=$ 23
    - Need some convention: if no j-th one, return -1 or u+1

How do we do it?

- How fast can we answer rank and select queries if we…
  - Don’t care about space?
  - What if we want $\Theta(u)$ bits of space?
  - Can we do better?



One Slide for Rank

- Jacobson (1989) gave an $u + o(u)$ bit solution for rank:
  - Idea: More levels!

1. Break array into *blocks* of size $\log^2 u$ bits
   - Store number of 1s to start of each block
   - Occupies $\Theta\left(\frac{u}{\log u}\right)$ bits
2. Break blocks into *subblocks* of size $\frac{1}{2}\log u$ bits
   - Store number of 1s from start of block to start of each subblock
   - Occupies $\Theta\left(\frac{u \log\log u}{\log u}\right)$ bits!
3. Store a table with all the precomputed answers for each subblock
   - There are $2^{\log u/2}$ such blocks…
   - Occupies $\Theta(\sqrt{u}\log\log u)$ bits (if we use some bit tricks)

One Slide for Rank

- Jacobson (1989) gave an $u + o(u)$ bit solution for rank:
  - Idea: More levels!
    1. Break array into *blocks* of size $\log^2 u$ bi
       - Store number of 1s t art of each
       - Occu $\left(\frac{u}{}\right)$
    2. Brea
       - subblock
       - Occu
    3. Store a table each subblock
       - There are 2 su ks…
       - Occupies $\Theta(\sqrt{u}$ log lo bits (if w e some bit tricks)

$$u + \Theta\left(\frac{u \log \log u}{\log u}\right) \text{ bits,}$$

and *we* decide the constant!

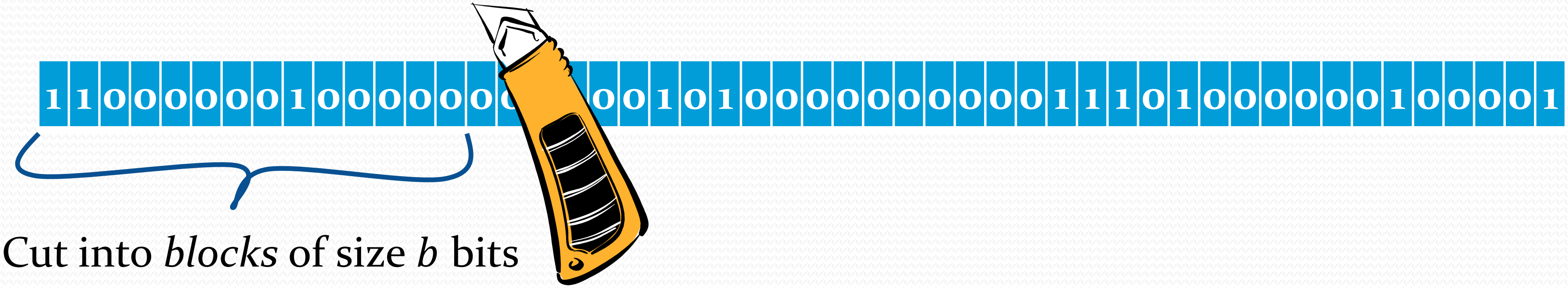
Better Ideas for Rank/Select

- Let's parameterize the problem in terms of the one bits
  - A bit vector of length u containing n *one bits*
  - If $n \ll u$ we should probably be able to do better
    - $\log\binom{u}{n} \leq n \log\left(\frac{eu}{n}\right) + O(1) = \; n \log\left(\frac{u}{n}\right) + \Theta(n)$
  - $\log\binom{u}{n} + \Theta\left(\frac{u}{\text{polylog } u}\right)$ possible for $\Theta(1)$ rank and select…
    - Patrascu (2008); see also related lower bound Patrascu and Viola(2010)
    - Very related to *predecessor search: i.e.,* find the index of the previous one

RaRaRa (Raman, Raman, and Rao 2007)

- A *fully indexable dictionary* (FID) is a data structure for representing a bit vector of length u, that can do:
  - Rank$(i, \{0,1\})$: count the number of zeros *or* ones in the prefix
  - Select$(j, \{0,1\})$: return the index of the j-th zero or one
- RaRaRa's result: a FID occuping $\log\binom{u}{n} + \Theta\left(\frac{u \log\log u}{\log u}\right)$ bits
  - Does all four operations in $\Theta(1)$ time
- *This (or Patrascu's result) is a <u>very</u> useful black box*
  - *We are going to describe it in detail!*

RaRaRa (Raman, Raman, and Rao 2007)



1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0

0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1

Cut into *blocks* of size b bits

- n_i denotes the number of 1s in block i, for $1 \leq i \leq \left\lceil \frac{u}{b} \right\rceil$
- If each block can be stored using $\left\lceil \log \binom{b}{n_i} \right\rceil$ bits:

$$\sum_i \left\lceil \log \binom{b}{n_i} \right\rceil \leq$$

$$\left\lceil \frac{u}{b} \right\rceil + \log \prod_i \binom{b}{n_i} \leq$$

$$\left\lceil \frac{u}{b} \right\rceil + \log \binom{u}{n}$$

Storing & Ranking Blocks

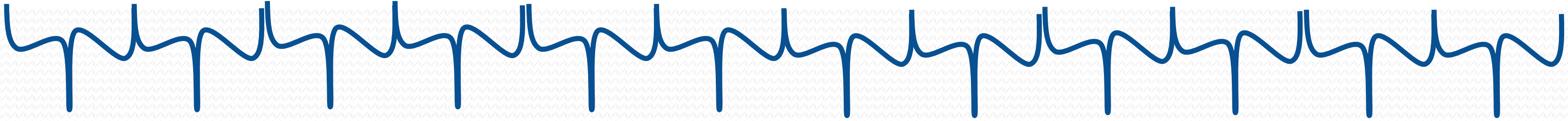
- How many types of blocks are there with n' ones?
- Enumerate them in lexicographic order:
  - Assign each possible one a $\lceil \log\binom{b}{n'} \rceil$-bit number
  - We will call this a lexicographic (lex.) number
- For each $n' \in [1, b]$ build a table that maps each lex. number to its corresponding block of length b:
  - Each table stores injective function: $h_{n_i}\colon 2^{\lceil \log\binom{b}{n_i} \rceil} \to 2^b$
  - Store concatenation of the lex. numbers for each block
    - Now what is the problem?

Storing & Ranking Blocks (2)

- The main issue: lex. numbers are not *one size*
  - We need to know where lex. number i starts
  - We also need to know the value n_i to access h_{n_i}
- How to overcome this?
  - Store two arrays: S and C of length $\left\lceil \frac{u}{b} \right\rceil$
    - $S[i]$ stores the number $\left\lceil \log \binom{b}{n_i} \right\rceil$ using $\Theta(\log b)$ bits
    - $C[i]$ stores the number n_i, also using $\Theta(\log b)$ bits
    - We want to be able to return *partial sums* on these arrays
      - Using S as an example: the sum $\sum_i S[i]$ for *any* $i \in \left[1, \left\lceil \frac{u}{b} \right\rceil\right]$

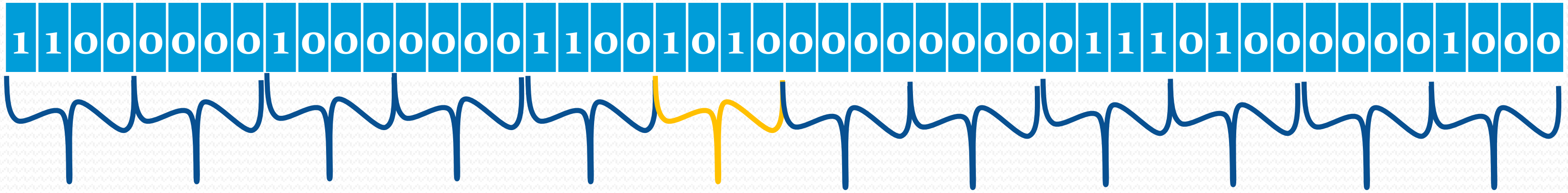
An Illustration with $b = 4$

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0



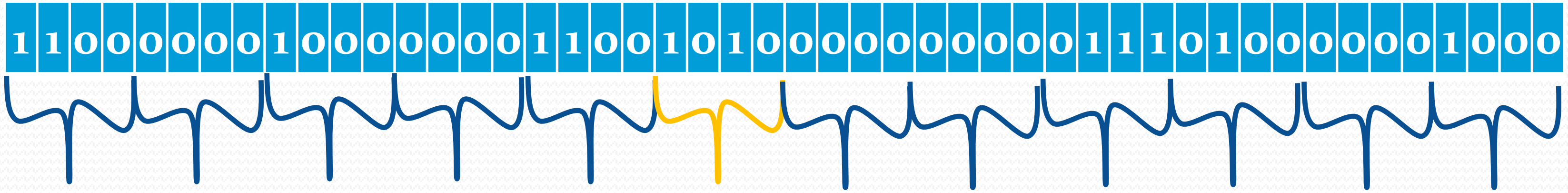
lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

An Illustration with $b = 4$



lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

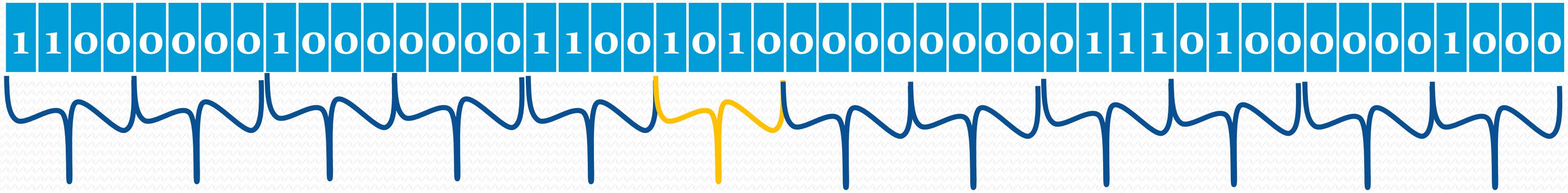
An Illustration with $b = 4$



1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0

lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

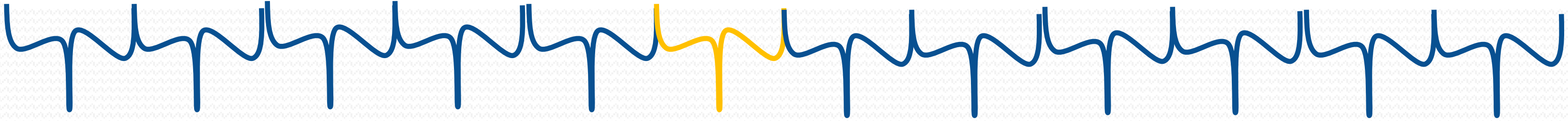
An Illustration with $b = 4$



lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

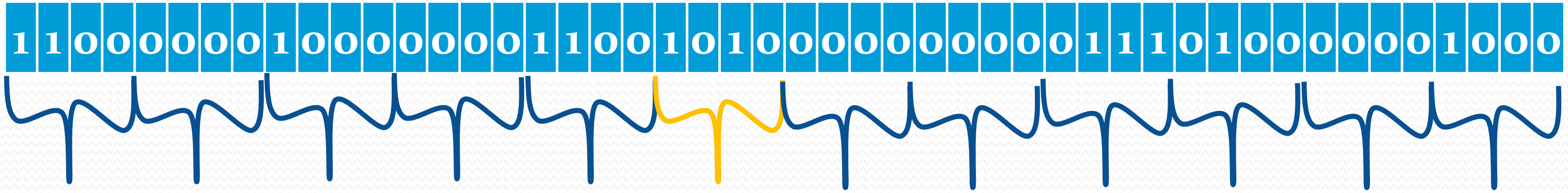
An Illustration with $b = 4$

1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0



lex	000		00		000	<u>001</u>			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

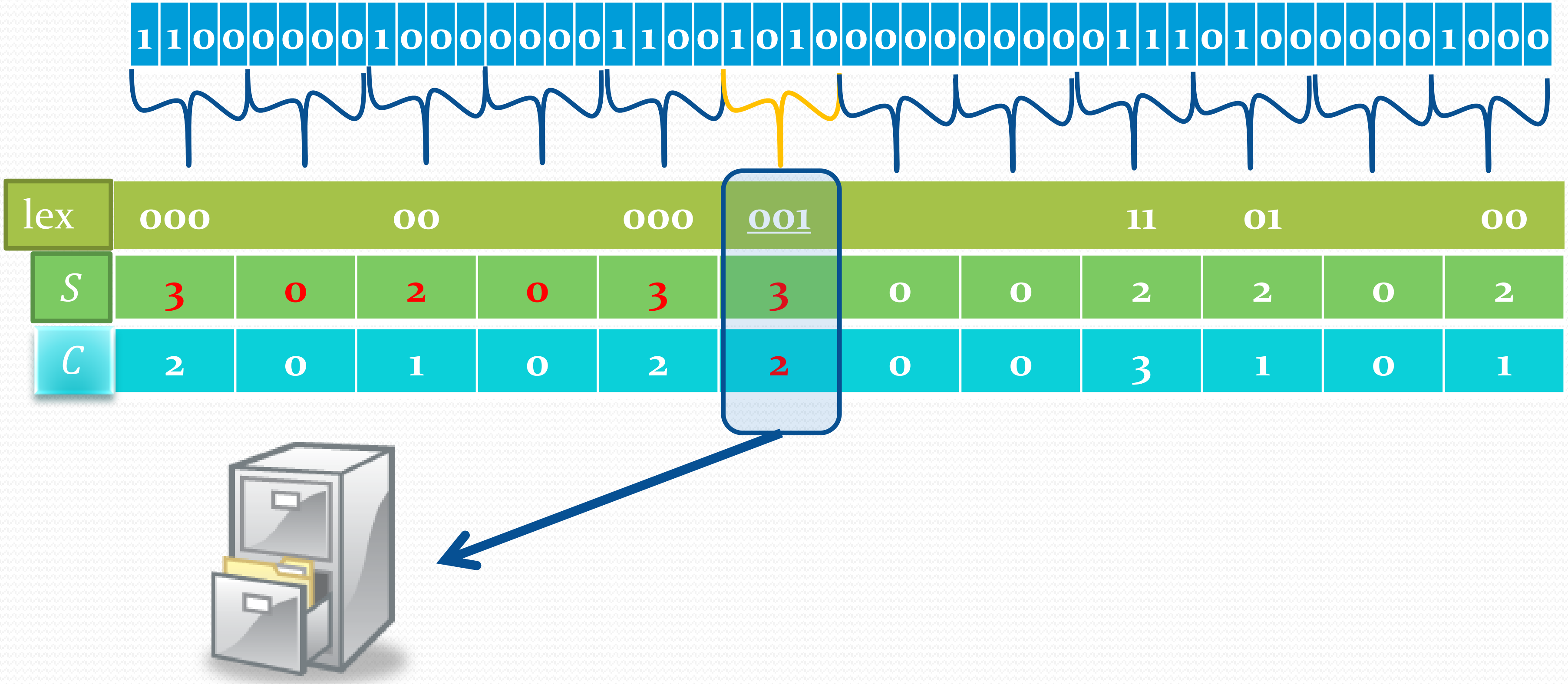
An Illustration with $b = 4$



1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0

lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

An Illustration with $b = 4$



1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0

lex	000		00		000	001			11	01		00
S	3	0	2	0	3	3	0	0	2	2	0	2
C	2	0	1	0	2	2	0	0	3	1	0	1

Digression: Partial Sums

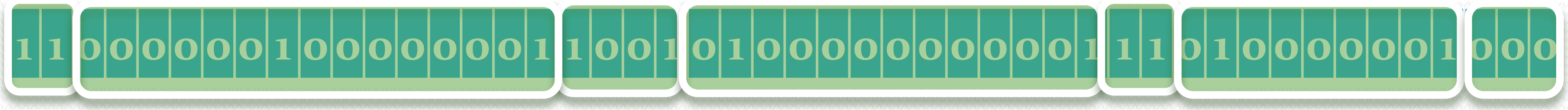
- Supporting partial sums on m elements $S[1..m]$:
  - Suppose each element $< \log^c m$ for some $c > 0$
    - $\sum_i S[i] < i \log^c m$ can be written using $\Theta(\log m)$ bits
  - Write down the sums up to every $\log m$-th element
    - This uses $\Theta\left(\frac{m}{\log m} \times \log m\right) = \Theta(m)$ bits
  - Write down the sums from each offset to each element
    - This uses $\Theta(m \log \log m)$ bits

Storing & Ranking Blocks (3)

- Recap:
  - The concatenated lex. numbers:
    - Occupy $\log\binom{u}{n} + \Theta\left(\frac{u}{\log u}\right)$ bits
  - The arrays S and C enhanced to support partial sums:
    - This occupies $\Theta\left(\frac{u \log\log u}{\log u}\right)$ bits (setting $m = \left\lceil\frac{u}{b}\right\rceil$)
  - All those lookup tables (Also: keep table for counting ones):
    - $\Theta(\sqrt{u}\,\mathrm{polylog}(n))$ can be made u^{ε} for any $\varepsilon > 0$
  - Using these we can easily do *access and rank* (*on* 0 ***and*** 1)
    - “But you said we could do select! What about select?”

Select *is* more complicated

- According to a someone who has implemented this:
    - “In practice you just use binary search.”
- How to do it:
    - Let p be the number of blocks, so around $\frac{2u}{\log u}$
    - Store the answer explicitly for every $\log^2 p$ query:
        - i.e., now we can answer $\text{select}(i \log^2 p)$ for $1 \leq i \leq n/\log^2 p$
    - Unlike rank, the *groups* for select will be non-uniform
        - The elements between each sample are a group

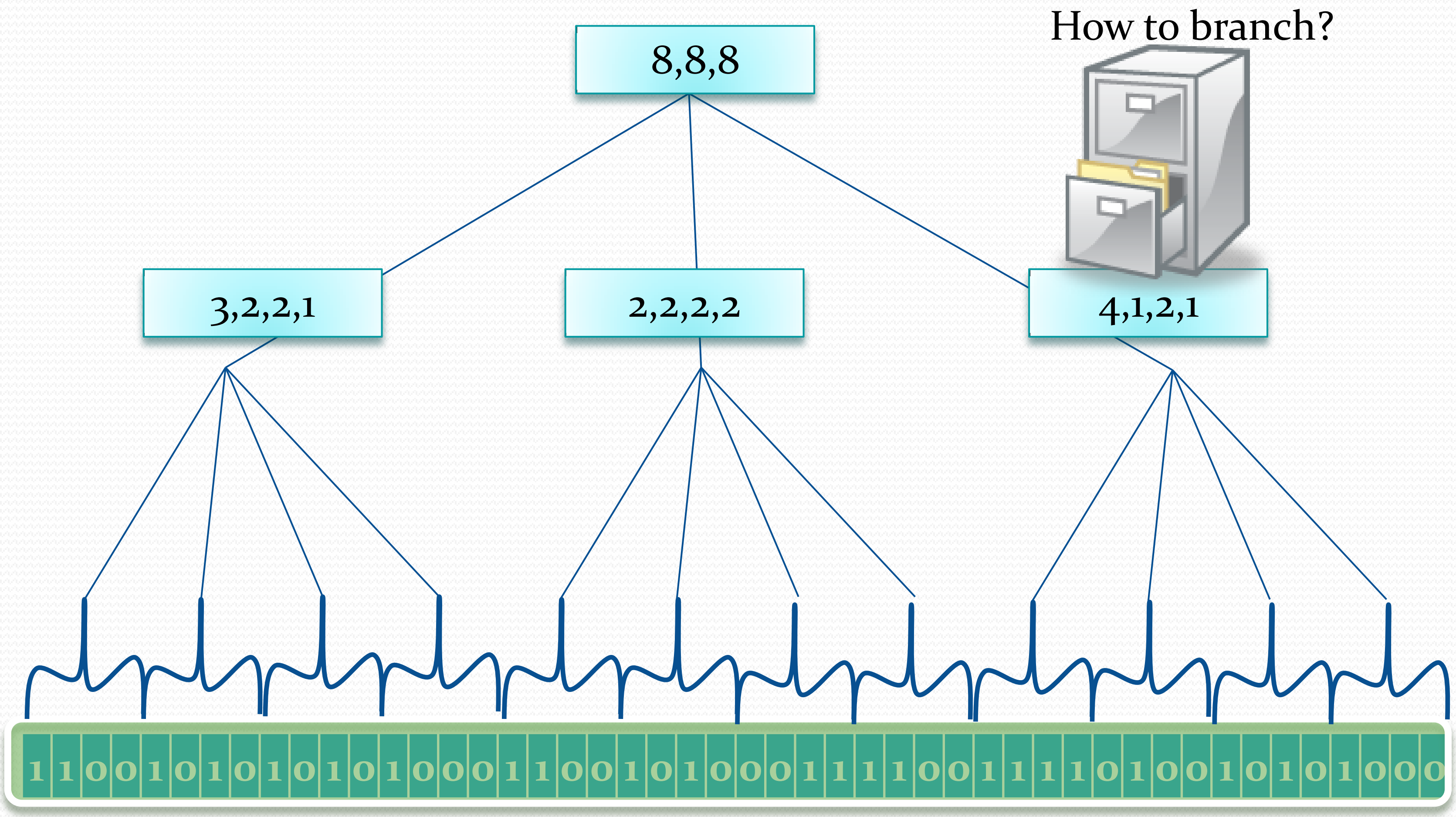


1 1 | 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 | 1 0 0 1 | 0 1 0 0 0 0 0 0 0 0 0 0 1 | 1 1 | 0 1 0 0 0 0 0 0 1 | 0 0 0

Two Kinds of Select Groups

- The “sparse case”
  - The size of the group is $\geq \log^4 p$
    - This is the easy case, as we simply write down the answers
    - There can only be $\left\lceil \frac{u}{\log^4 p} \right\rceil$ such groups: spend $\Theta(\log^3 p)$ bits per
- The “dense case”
  - In this case, we construct a search tree over the group’s blocks
    - Tree will have fan out $\sqrt{\log p}$
      - How tall will the tree be?
    - Each node has array storing # of ones in each child’s subtree
      - Each # is size $\Theta(\log \log p)$ bits (how many ones in whole tree?)…
      - …so an entire array can be packed in a single word!

Don’t try this at home



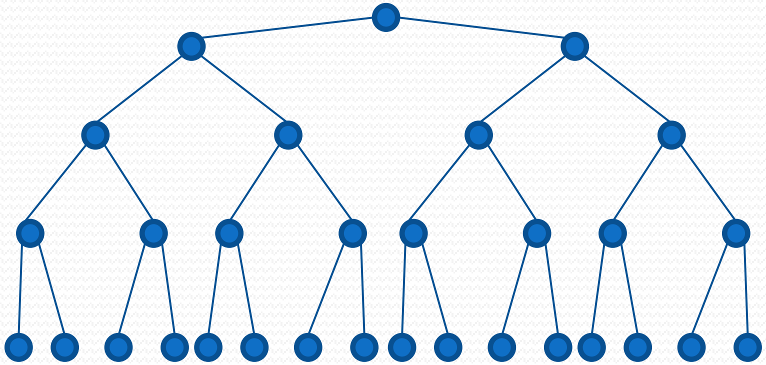
Before this was the entire bit vector, now it’s just *one dense block*

Wrap up

- Total space for the dense groups:
  - Dense group spanning k blocks has $\Theta\left(\frac{k}{\sqrt{p}\log u}\right)$ nodes
  - Each node stores an array of size $\Theta\left(\sqrt{p}\log\log p\right)$ bits
  - Total: $\Theta\left(\frac{u\log\log u}{\log u}\right)$ bits
- We can do the same thing for select on zeros!

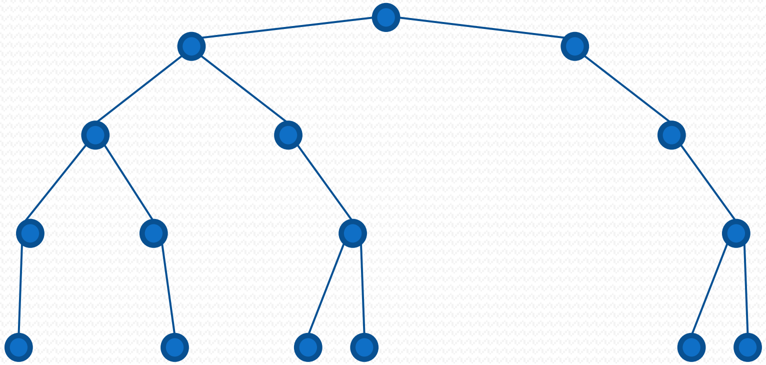
“But what about trees?”

- What does rank and select have to do with trees?
- Remember the heap
  - Left-child of node $i = 2i$
  - Right-child of node $i = 2i + 1$
  - Parent of $i = \lfloor i/2 \rfloor$



“But what about trees?”

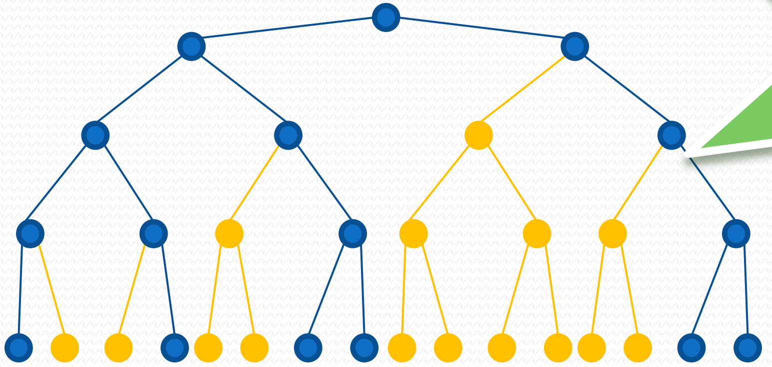
- What does rank and select have to do with trees?
- Remember the heap
    - Left-child of node $i = 2i$
    - Right-child of node $i = 2i + 1$
    - Parent of $i = \lfloor i/2 \rfloor$



“But what about trees?”

- What does rank and select have to do with trees?
- Remember the heap
  - Left-child of node $i = 2i$
  - Right-child of node $i = 2i + 1$
  - Parent of $i = \lfloor i/2 \rfloor$

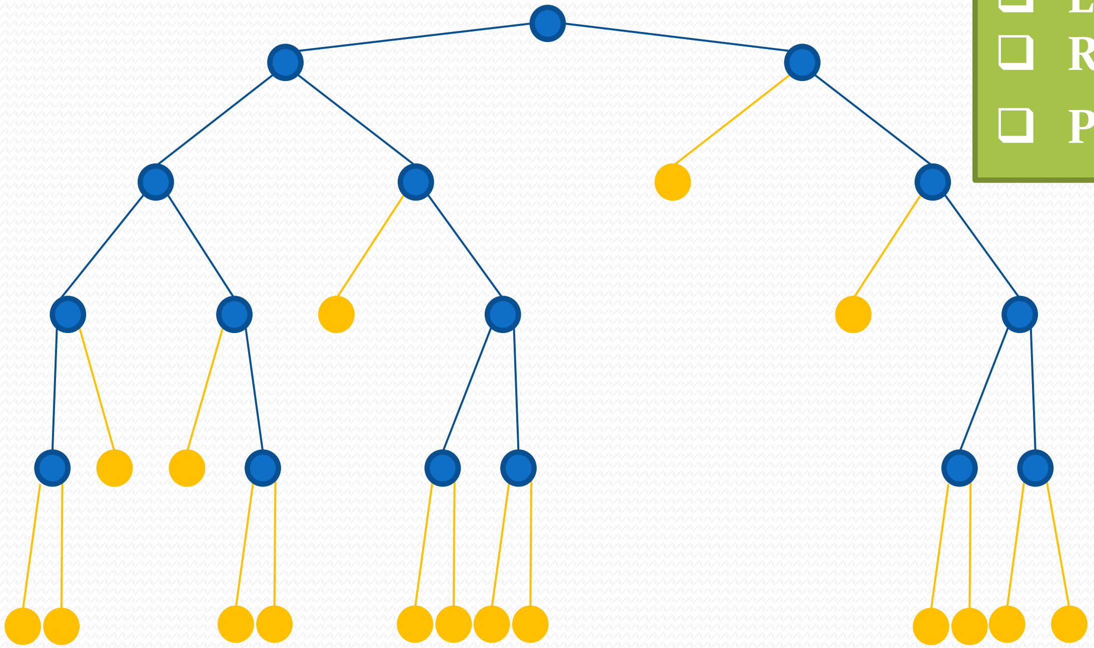
Neat, but it doesn’t use $2n$ bits… or use the *stuff* we just spent <u>a lot of time learning about</u>



Write as a bit vector: 1111101110100011001001100000011

Level Order Binary Marked (Jacobson)

- Make it a complete binary tree (put the leaves in)



- □ Left-child of i $= 2\,\mathrm{rank}(i)$
- □ Right-child of i $= 2\,rank(i) + 1$
- □ Parent of i $= select\left(\left\lfloor \frac{i}{2} \right\rfloor\right)$

Uses
$2n + o(n)$
bits!!!

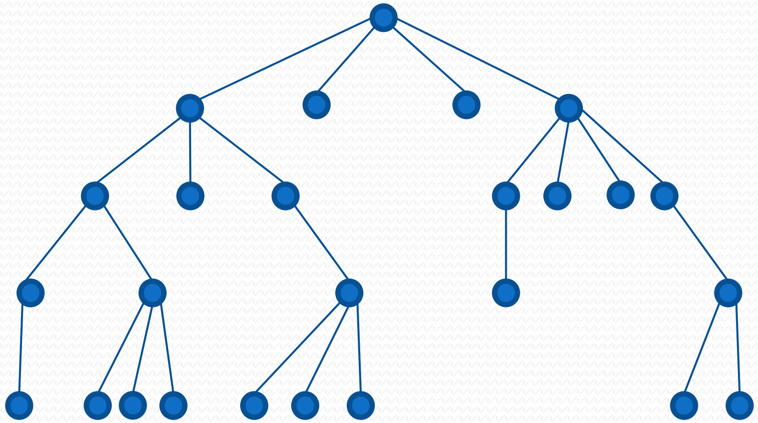
Write as a bit vector:

111110111010110011111000000000000

“What about non-binary trees?”

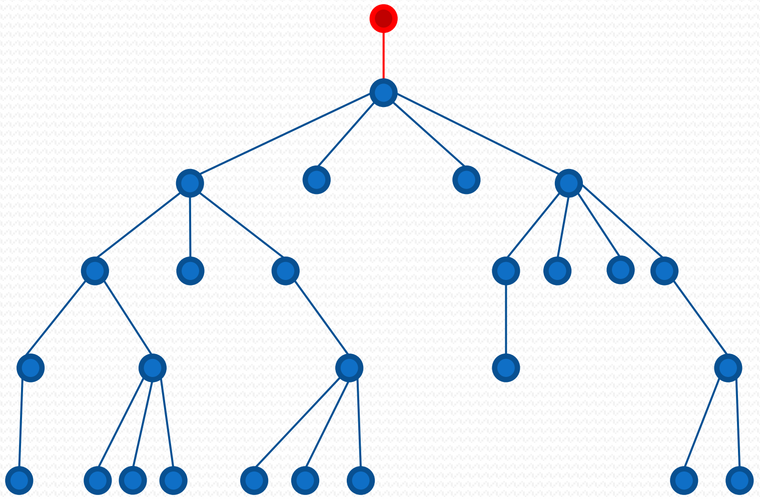
- Ordered trees: uniquely identified by degree sequence
  - Idea: encode these and write them down
    - Several different ways to do this
- Level Ordered Unary Degree Sequence (LOUDS)
  - Also by Jacobson

Numbers in unary:
0 → 0
1 → 10
2 → 110
3 → 1110



“What about non-binary trees?”

- Ordered trees: uniquely identified by degree sequence
  - Idea: encode these and write them down
    - Several different ways to do this
- Level Ordered Unary Degree Sequence (LOUDS)
  - Also by Jacobson



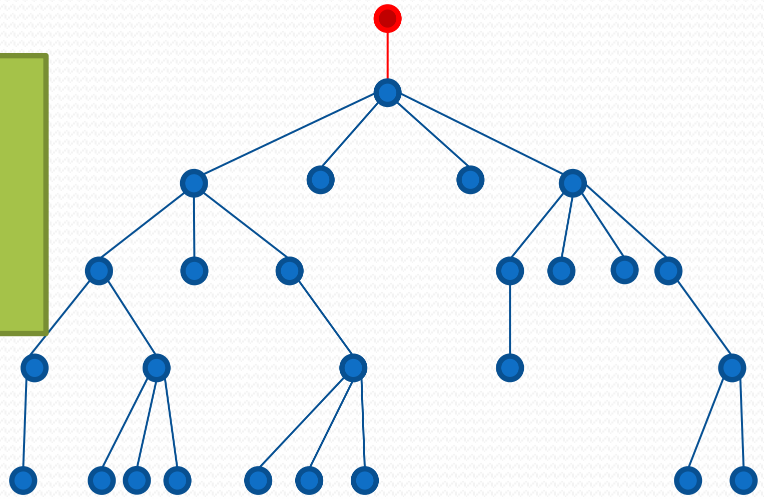
Numbers in unary:
0 → 0
1 → 10
2 → 110
3 → 1110

Add “super root” to make sure each node associated with a zero bit:
10111101110001111011001010001010111011100110000000000

“What about non-binary trees?”

- Ordered trees: uniquely identified by degree sequence
  - Idea: encode these and write them down (Jacobson)
    - Several different ways to do this
- Level Ordered Unary Degree Sequence (LOUDS)

- ❑ **Children of $i \sim select(rank(j,1),0)$ ***
- ❑ **Parent of $i = select(rank(i,0),1)$**
- ❑ **Next Sibling…**
- ❑ **Degree…**



Add “super root” to make sure each node associated with a zero bit:

10111101110001111011001010001010111011100110000000000

*to find child k do this setting j to be the index of the k-th one in unary expansion of i

Other Results

- We talked about two methods: LOBM and LOUDS
- Other methods:
  - Balanced Parentheses (BP)
    - *(Jacobson 1989, Munro & Raman 1997, Munro et al. 2001, Sadakane 2003, Lu & Yeh 2008)*
    - We will talk next time about its use for representing graphs
    - Can also support level ancestor, lowest common ancestor (LCA), and many more operations.
  - Depth-First Unary Degree Sequence (DFUDS)
    - *(Benoit et al. 2005, Jansson et al. 2007)*
    - Can compute subtree size in $O(1)$ time + LOUDS operations
  - Tree Covering (TC)
    - *(Geary et al. 2004, He et al. 2007, Farzan and Munro 2008)*
  - Fully Function (FF)
    - *(Sadakane and Navarro 2010,2012)*

“Universal” Representation

- A result by Farzan, Munro, and Rao (2009):
  - We can represent a tree using $2n + o(n)$ bits such that we can access any block of $\log n$ consecutive bits in the DFUDS, BP, or TC representation, etc., in $\Theta(1)$ time.

- Bottom Line: can do it all in $2n + o(n)$ bits!

- Next Lecture: BP and graphs

Balanced Parentheses

- Last class we looked at rank/select
- Consider the following problem:

((((()())(()()))((()())(()())))(((()())(()()))((()())(()()))))

Balanced Parentheses

- Last class we looked at rank/select
- Consider the following problem:

((((()())(()()))((()())(()())))(((()())(()()))((()())(()()))))

- Find the matching parenthesis

Balanced Parentheses

- Last class we looked at rank/select
- Consider the following problem:

((((()())(()()))((()())(()())))(((()())(()()))((()())(()()))))

- Find the matching parenthesis
  - Looks similar *(kind of… sort of)* to the rank/select problem
  - Supports the following operations (Jacobson 1989, Munro and Raman 1997):
    - Find_Match(i): see picture
    - Excess(i): return difference between # of open/closed at i

Balanced Parentheses

- Last class we looked at rank/select
- Consider the following problem:

((((()())(()()))((()())(()())))(((()())(()()))((()())(()()))))

- Find the matching parenthesis
  - Looks similar *(kind of… sort of)* to the rank/select problem
  - Supports the following operations (Jacobson 1989, Munro and Raman 1997):
    - Find_Match(i): see picture
    - Excess(i): return difference between # of open/closed at i
    - Enclose(i): given pair (opening at i), return smallest “containing” pair

Balanced Parentheses

- Last class we looked at rank/select
- Consider the following problem:

((((()())(()()))((()())(()())))(((()())(()()))((()())(()()))))

- Find the matching parenthesis
  - Looks similar *(kind of… sort of)* to the rank/select problem
  - Supports the following operations (Jacobson 1989, Munro and Raman 1997):
    - Find_Match(i): see picture
    - Excess(i): return difference between # of open/closed at i
    - Enclose(i): given pair (opening at i), return smallest "containing" pair
    - Double_Enclose(i,j): given pairs (opening at i and j), return smallest "containing" pair
    - Many additional operations added later (*Lu & Yeh 2008)*

Jacobson’s Solution for Find_Match

(((()())(()()))(( ))(()())))(((()())(()()))((()())(()()))))

Cut into *blocks* of size *b*

- This won’t really be succinct: $\Theta(n)$ bits
- As you might expect: break it into blocks of size b
- Main Idea:
  - If match is in the same block find it by scanning
  - Alternative case we need some additional observations

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can’t just store the answers for these

((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () (()))

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can't just store the answers for these

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))

- We can, however, store the *excess*

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can’t just store the answers for these

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))

- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can’t just store the answers for these

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	--	--	--	--	--

- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can't just store the answers for these

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	--	--	--	--

- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
    - The number of far parenthesis can be linear
    - Can’t just store the answers for these

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	--	--	--

- We can, however, store the *excess*
- Consider a single block:
    - Mark all of the far parenthesis
    - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can’t just store the answers for these

4	2	4	4	6	0
((()(()()(	))((()()))	(()((())))(	))()()(()(	(((()()))(	))))()()))
1000100000	0000000000	1000000000	0000001000	--	--

- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can't just store the answers for these

4	2	4	4	6	0
((()(()()(	))((()()))	(()((()))(	))()()(()(	((((()()))(	))))()()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

Some Definitions

- A *far parenthesis* has its match in a different block
  - The number of far parenthesis can be linear
  - Can't just store the answers for these

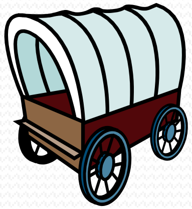
4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

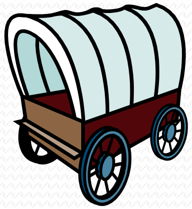
- We can, however, store the *excess*
- Consider a single block:
  - Mark all of the far parenthesis
  - Mark whenever the block of the match changes

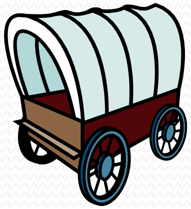
Some Definitions (2)

- We call the parentheses marked by 1 bits *pioneers*

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--





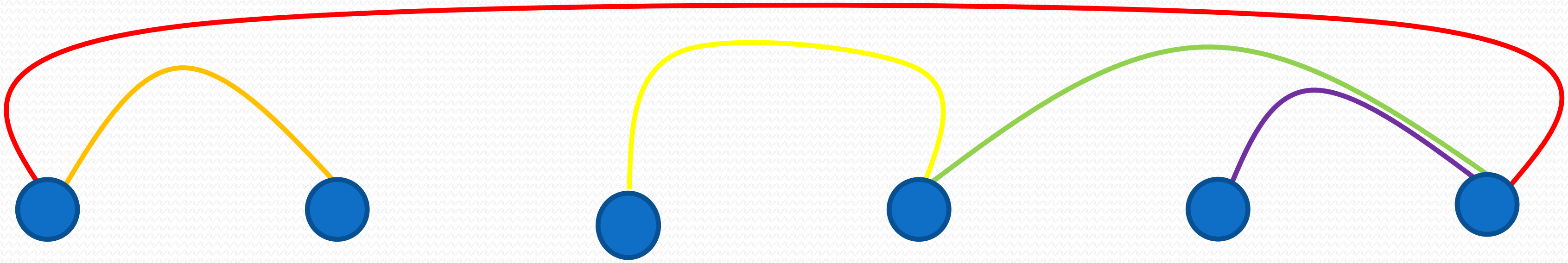


- How many pioneers can there be?

Digression: Pioneers

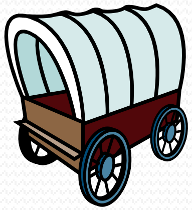
- Let’s think of the blocks as vertices in a graph

((( ) (( ) ( ) (	)) ((( ) ( )))	(( ) ((( ))) (	)) ( ) ( ) (( ) (	(((( ) ( ))) (	)))) ( ) ( )))



- Balanced parentheses → we can draw without crossings
- That means this graph is planar (even better: outerplanar)
  - If we have $m = \left\lceil \frac{n}{b} \right\rceil$ vertices, there can be at most $2m - 3$ edges
  - This means: number of pioneers is sublinear if $b = \omega(1)$ (yay)

Using this Fact



- We can write down the block numbers of the pioneers

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

6 2 4 6 6

6	2	4	6	6

- Store this pioneer information using $\Theta(n \log n / b)$ bits
- Given an arbitrary opening parenthesis:
  - We can find the preceding pioneer using rank/select

Performing Find_Match

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

6	2	4	6	6

- Suppose we want to find the match of the red (
  - Search within block to see if it is matched…
    - in this case “no”
  - Find the preceding pioneer

Performing Find_Match

4	2	4	4	6	0
((()(()()(	))((()()))	(()((()))(	))()()(()(	(((()()))(	))))()()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

6	2	4	6	6

- Suppose we want to find the match of the red (
  - Search within block to see if it is matched…
    - in this case “no”
  - Find the preceding pioneer
    - Determine excess up to *i*
      - In this case: 4
    - Find the *first time* excess reduces to 3 in pioneer block

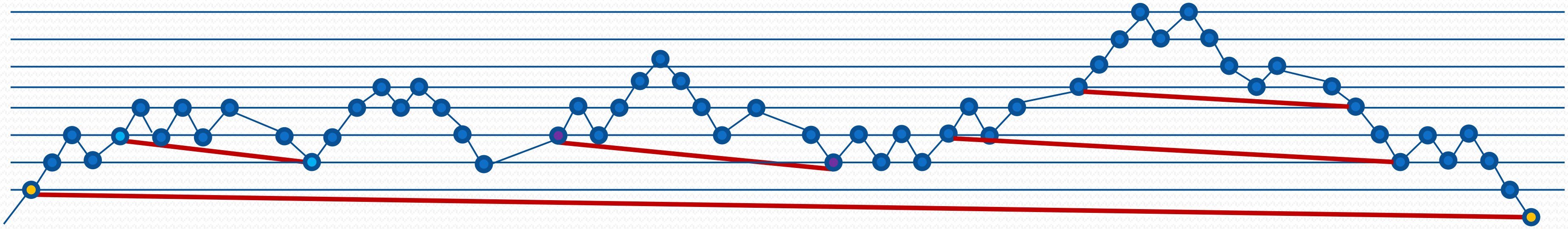
Performing Find_Match

4	2	4	4	6	0
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

6	2	4	6	6

- Suppose we want to find the match of the red (
  - Search within block to see if it is matched…
    - in this case "no"
  - Find the preceding pioneer
    - Determine excess up to *i*
      - In this case: 4
    - Find the *first time* excess reduces to 3 in pioneer block
      - Why??

Stack View



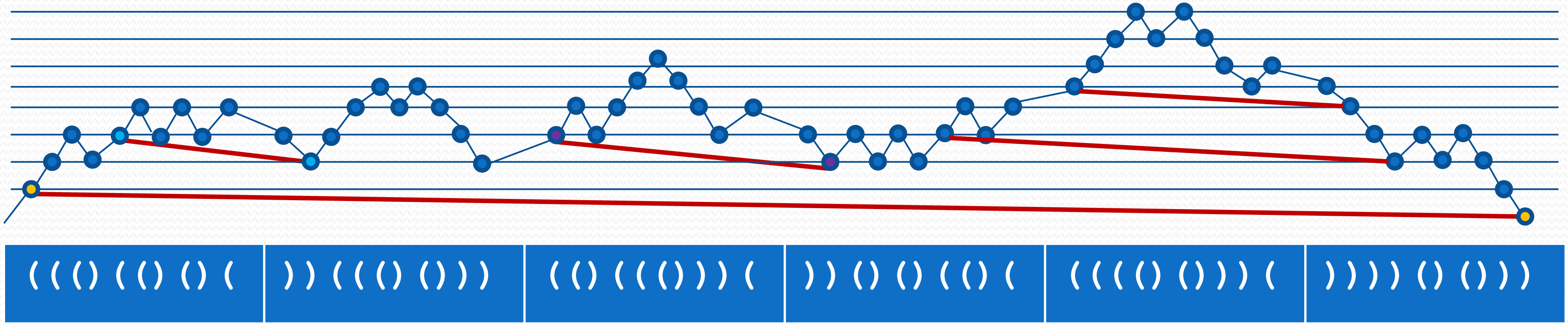
((() (() () (	)) ((() ()))	(() ((())) (	)) () () (() (	(((() ())) (	)))) () ()))
1000100000	0000000000	1000000000	0000001000	1000000000	--

Analysis of Find_Match

- We have described how to find a closing parenthesis
  - The query time was $\Theta(b)$, since we must scan blocks
  - Excess takes $\Theta(b)$ time using scan + block info
- The space is:
  - $2n$ bits for the pioneer bit vector ($+o(n)$ for rank/select)
  - $\Theta\left(\frac{n \log n}{b}\right)$ bits for storing the pioneer blocks
  - $\Theta\left(\frac{n \log n}{b}\right)$ bits for the excess information
  - Set $b = \log n$ and it all works out to be $\Theta(n)$ bits
- Do the same thing for finding an opening parenthesis

Supporting Enclose

- Consider the “stack view” again



((( ) (( ) ( ) (

)) ((( ) ( )))

(( ) ((( ))) (

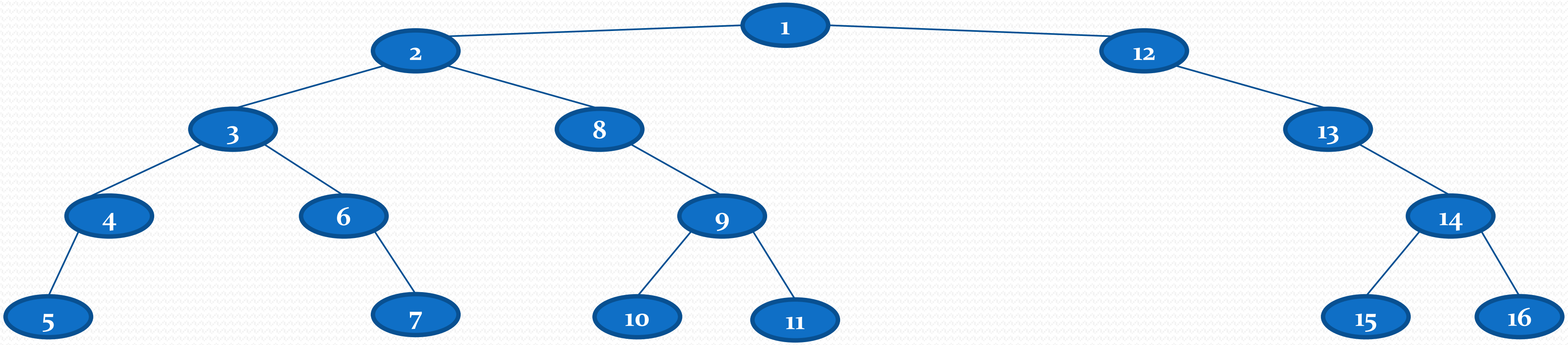
)) ( ) ( ) (( ) (

(((( ) ( ))) (

)))) ( ) ( )))

- Suppose minimum of a block has excess x:
  - Store first block to the right having excess $x - 1$
  - Extra $\Theta\left(\frac{n}{b}\log n\right)$ bits
- Use this + pioneer information to answer queries

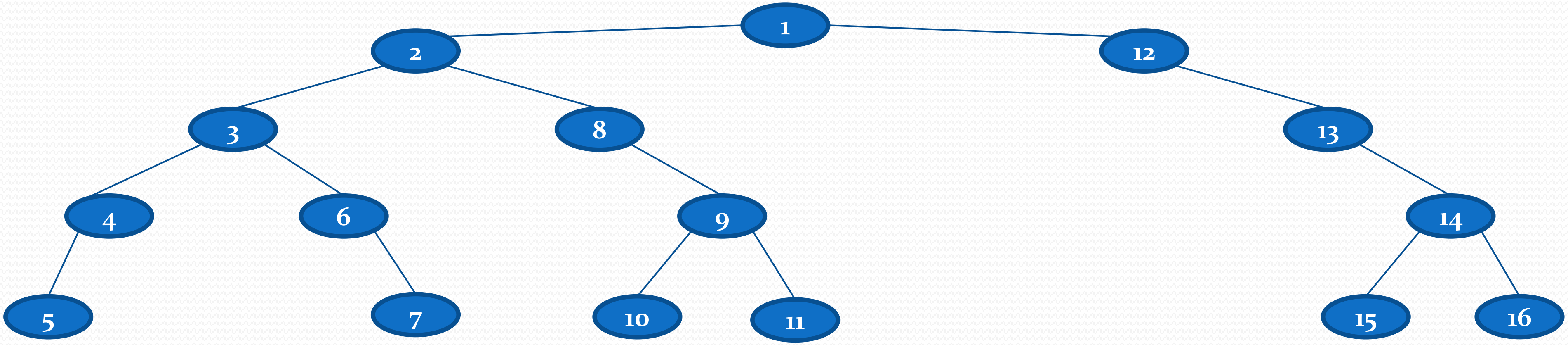
Binary Trees Revisited



Represent a node like so: open-paren left-child right-child close-paren

(1(2(3(4(5))(6(7)))(8(9(10)(11)))(12(13(14(15)(16)))))

Binary Trees Revisited

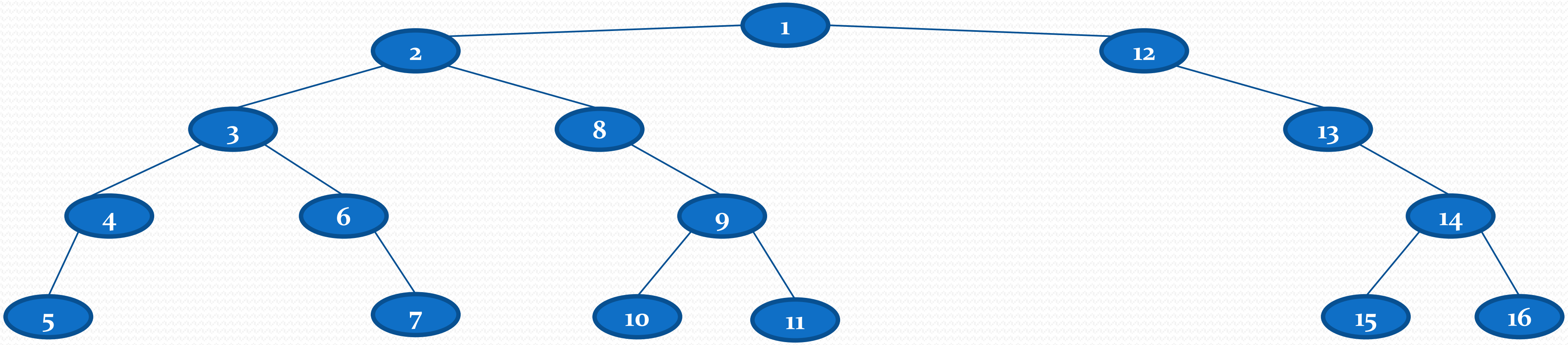


Represent a node like so: open-paren left-child right-child close-paren

((((())(()))((()()))(((()()))))

OK: now look at node 6

Binary Trees Revisited



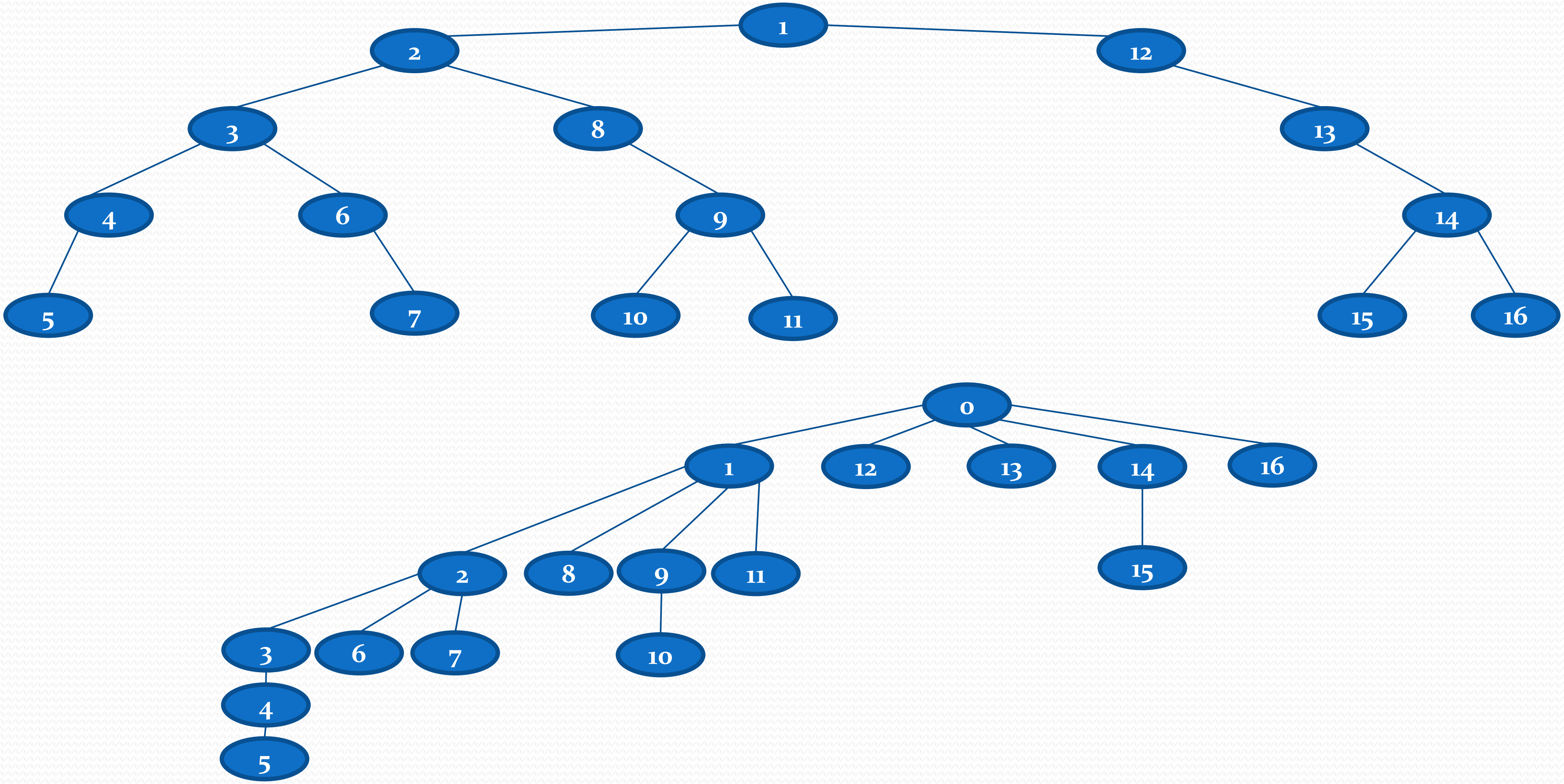
Represent a node like so: open-paren left-child right-child close-paren

((((())(()))((()()))(((()()))))

OK: now look at node 6

Tell me whether 7 is a left or a right child…

Transformation

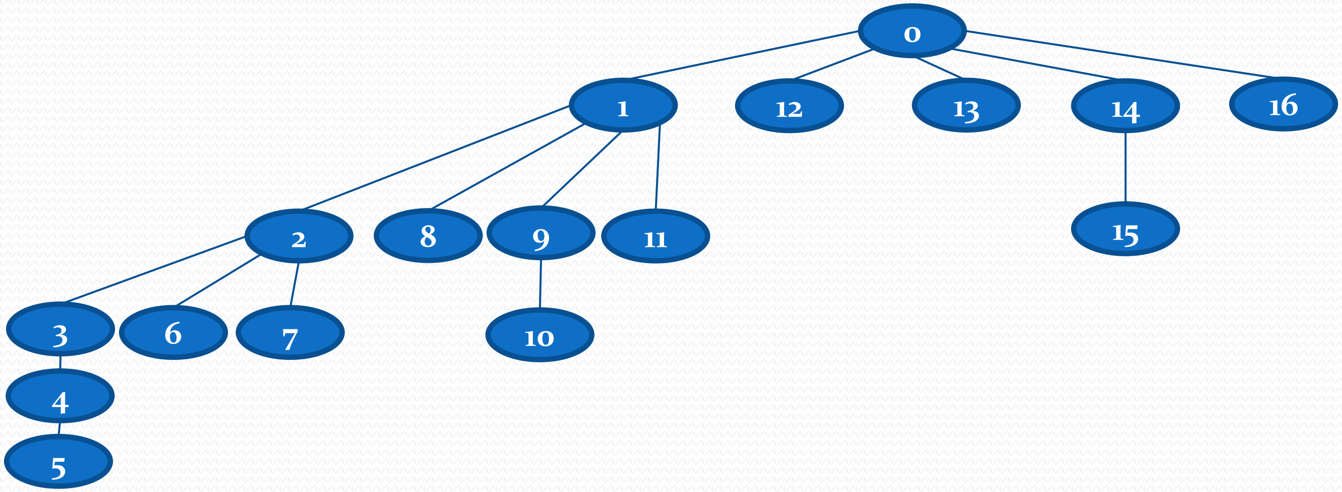


Transformation

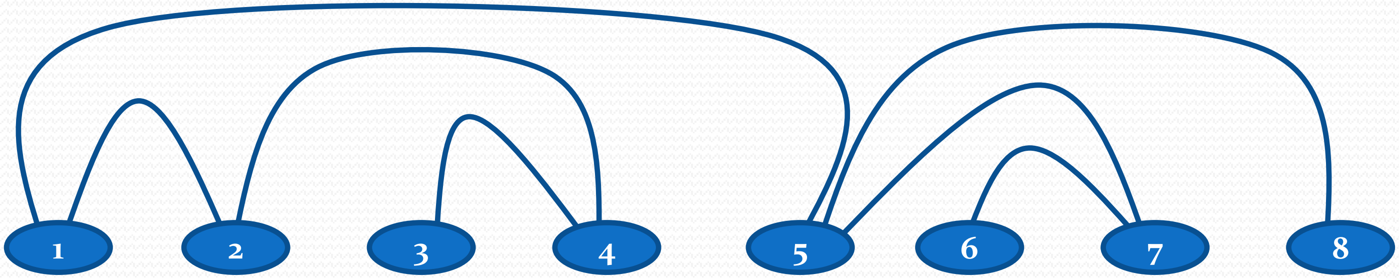
Build BP over the ordered tree instead

Right child?
Left child?
Parent?
Subtree size?

(0(1(2(3(4(5)))(6)(7))(8)(9(10))(11)(12)(13)(14(15))(16))



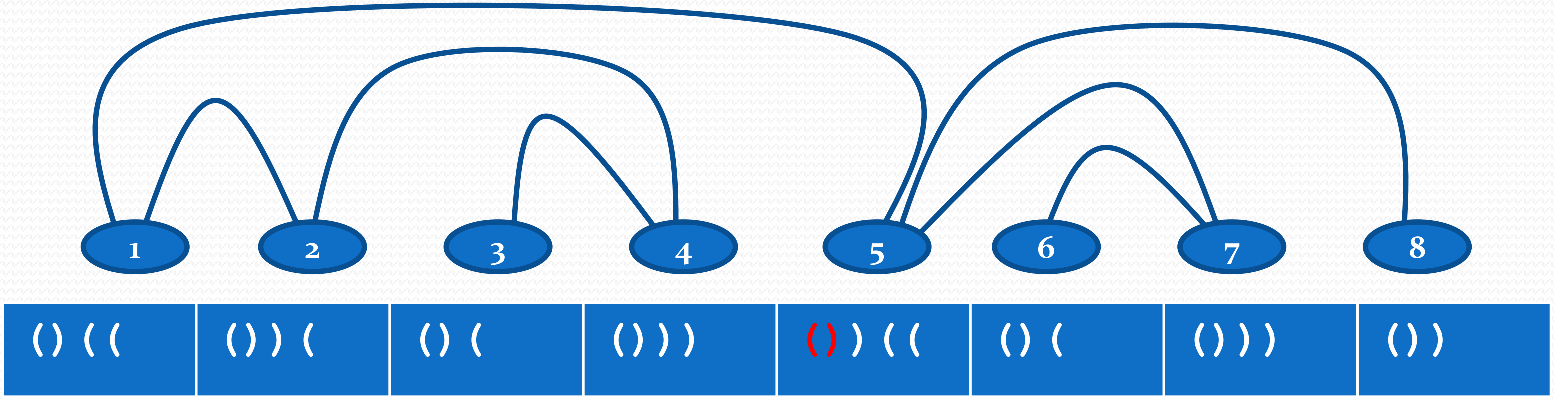
Outerplanar/One-Page Graphs



() ((	()) (	() (	()))	()) ((	() (	()))	())

- Remember bound on number of pioneers…
- We can represent outer planar (i.e., one page graphs)
  - Even works for multi-graphs
- Use rank/select to move from “spine number” to ()
- $\Theta(n)$ bits in total:
  - Can be reduced to $2n + 2m + o(n)$ (Munro and Raman 1997)
    - … using not one… not two… but three levels of blocking!

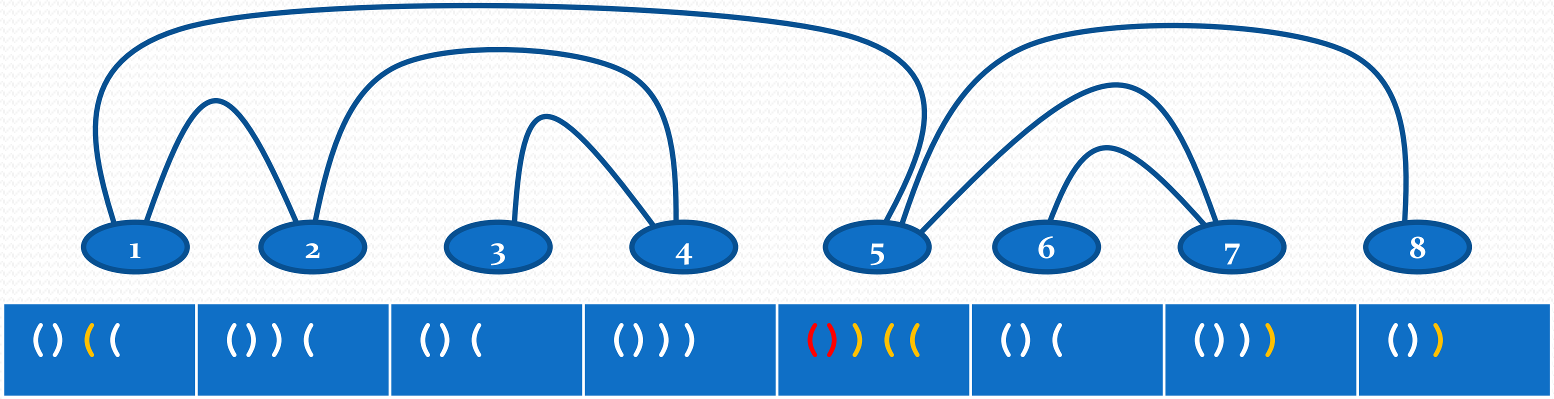
Outerplanar/One-Page Graphs



() ((	()) (	() (	()))	()) ((	() (	()))	())

- Navigation:
  - List neighbours of node i:
    - Find the “adjacent parenthesis” corresponding to i (e.g., $i = 5$)

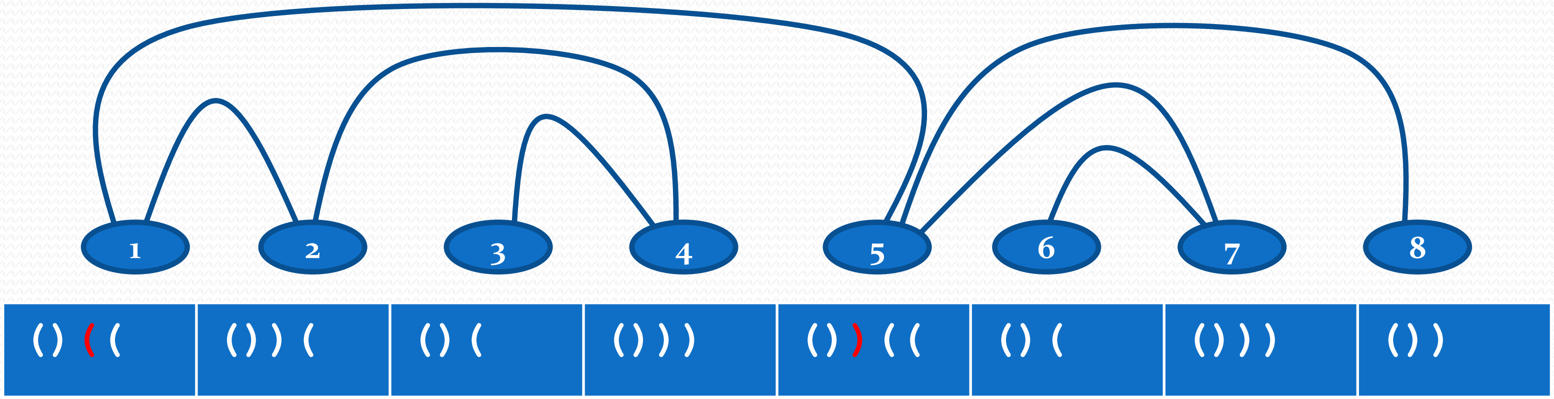
Outerplanar/One-Page Graphs



( ) ((	( )) (	( ) (	( )))	( )) ((	( ) (	( )))	( ))

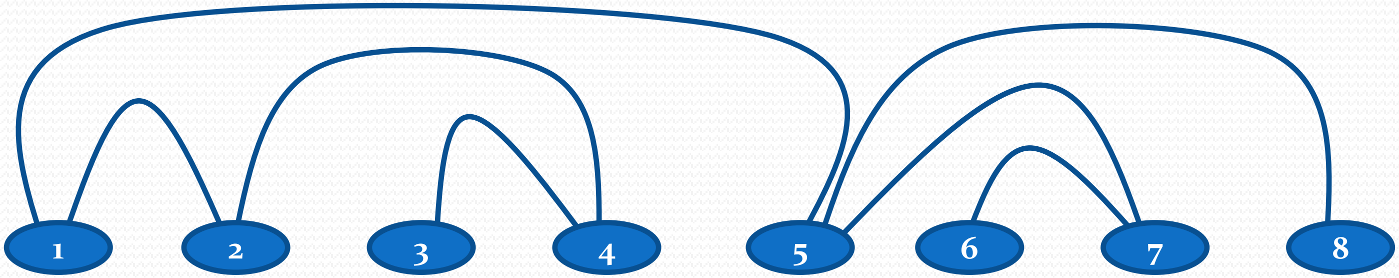
- Navigation:
  - List neighbours of node i:
    - Find the “adjacent parenthesis” corresponding to i (e.g., $i = 5$)
    - For each matching paren. report the label: e.g., 1,7,8

Outerplanar/One-Page Graphs



- Navigation:
  - List neighbours of node i:
    - Find the “adjacent parenthesis” corresponding to i (e.g., $i = 5$)
    - For each matching paren. report the label: e.g., 1,7,8
  - Test Adjacency of (i, j) (e.g., $i = 1, j = 4$):
    - Find first matching pair after i

Outerplanar/One-Page Graphs



() ((	()) (	() (	()))	()) ((	() (	()))	())

- Navigation:
  - List neighbours of node i:
    - Find the “adjacent parenthesis” corresponding to i (e.g., $i = 5$)
    - For each matching paren. report the label: e.g., 1,7,8
  - Test Adjacency of (i, j) (e.g., $i = 1, j = 4$):
    - Find first matching pair after i
    - Find last matching pair after j
    - If neither query yields a “yes” the answer is “no”

Neat!

It works for Planar Graphs too!

- Thanks to a theorem of Yannakakis (1986):

  *There is a linear time algorithm that can embed any planar graph into no more than **<u>four</u>** page graphs.*

  (The "spine numbers" are the same for all pages)

- This means that we can apply the BP representation:
  - We get planar graphs that occupy $8n + 2m + o(n)$ bits
    - Adjacency listing in $O(t + 1)$ time for degree t vertices
    - Adjacency testing in $O(1)$ time
  - Any k-page graph occupies $2kn + 2m + o(nk)$ bits
    - Adjacency listing in $O(k + t)$
    - Adjacency testing in $O(k)$ time

Arbitrary Graphs

- What about non-planar graphs?
- We have been taught:
  - Adjacency list representation:
    - $\Theta((n+m)\log n)$ bits
    - $\Theta(t+1)$ time to report all t neighbours
    - $\Theta(\log n)$ time for adjacency testing (PSSSST: can be improved to $\Theta(\log\log n)$)
  - Adjacency matrix representation:
    - n^2 bits for directed; $\binom{n}{2}$ bits for undirected graph
    - $\Theta(n)$ time for adjacency listing
    - $\Theta(1)$ time for adjacency testing*

Succinct(?) Arbitrary Graphs

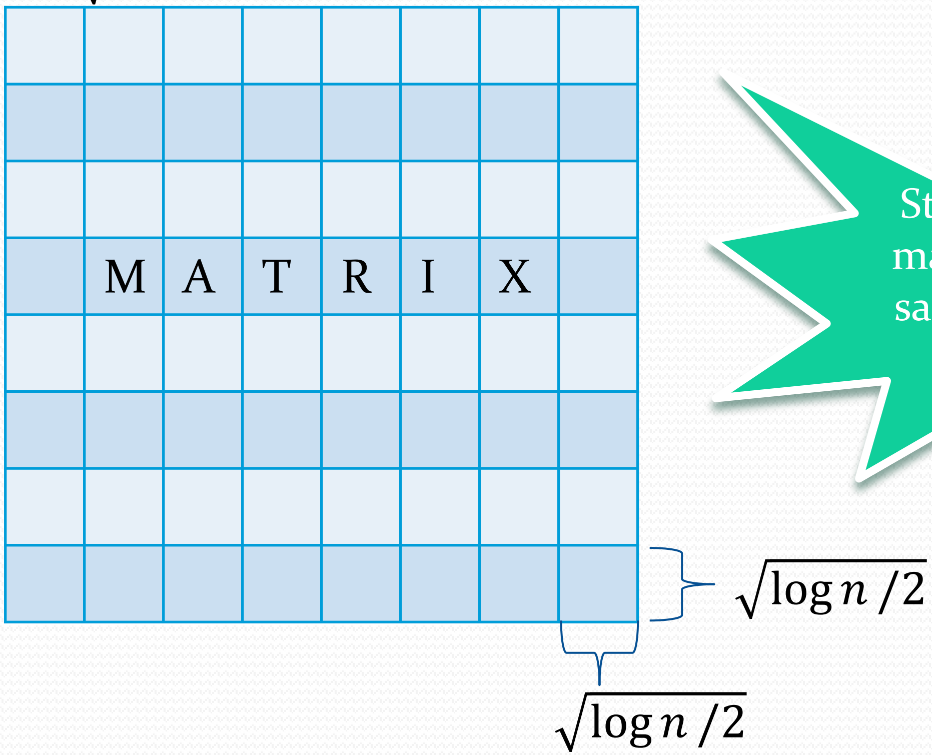
- How many bits to represent a n vertex digraph?
  - $B = \log \binom{n^2}{m}$ if it has m edges
- Idea #1: “Use the FID”
  - Represent each row of the adjacency matrix using a FID
    - Let m_i be the number of 1s in row i
    - This takes $\sum_i \log \binom{n}{m_i} + \Theta(n^2 \log\log n \,/\, \log n)$ bits
      - Or $B + \Theta(n^2 \log\log n \,/\, \log n)$ bits
        - Second term is *little-oh-ish* when

$$m = \omega\left(\frac{n^2}{\log n}\right) \textit{ and } m = o\left(n^2\left(1 - \frac{1}{\log n}\right)\right)$$

        - For now assume the graph is in this range (i.e., dense)
      - Can list “out-neighbours” in $\Theta(1)$ time per element
      - Can test adjacency in $\Theta(1)$ time

What about “in-neighbours”

- How can we report the rows and columns efficiently?
- Idea #2: “$\Theta\left(\frac{n^2 \log\log n}{\sqrt{\log n}}\right)$ is technically $o(n^2)$”



Store each little matrix using the same method as the FID

What about “in-neighbours” (2)

- For each row and each column
  - Construct aux. FID structures with $b = \frac{\sqrt{\log n}}{2}$
    - Access any little row/col. block by fetching the square block

- We have a succinct representation of directed graphs
  - For a particular range of m…
  - *Partial result: not so convincing*

Other Ranges of *m*

- If we want to support *adjacency testing*, *reporting in-neighbours* **and** *out-neighbours* *(Farzan and Munro, 2013):*

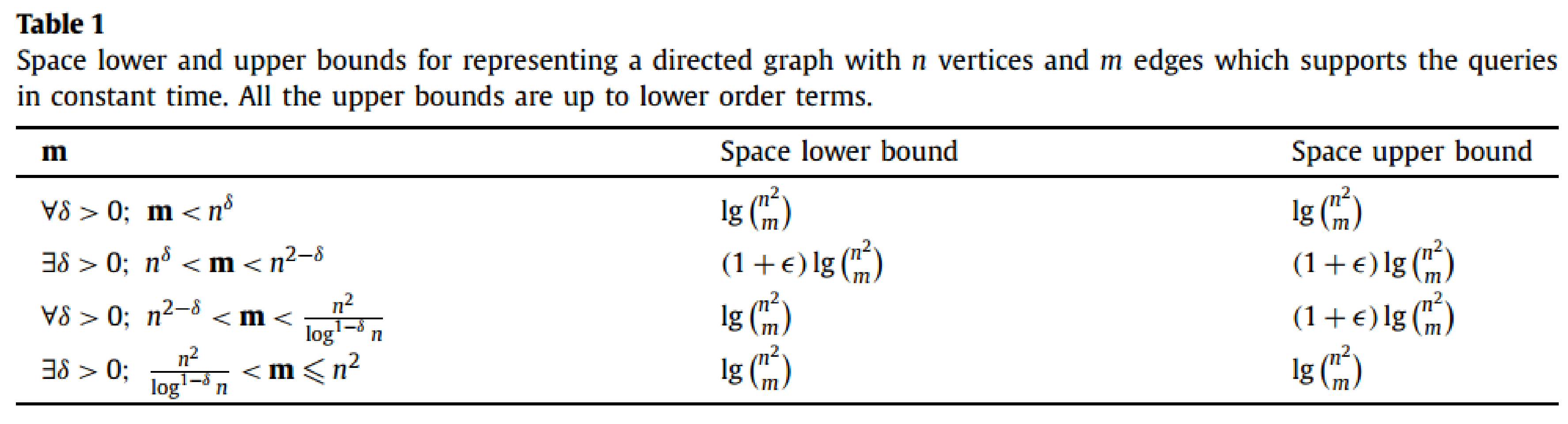


Table 1
Space lower and upper bounds for representing a directed graph with n vertices and m edges which supports the queries in constant time. All the upper bounds are up to lower order terms.

m	Space lower bound	Space upper bound
$\forall\delta > 0;\ \mathbf{m} < n^{\delta}$	$\lg\binom{n^2}{m}$	$\lg\binom{n^2}{m}$
$\exists\delta > 0;\ n^{\delta} < \mathbf{m} < n^{2-\delta}$	$(1+\epsilon)\lg\binom{n^2}{m}$	$(1+\epsilon)\lg\binom{n^2}{m}$
$\forall\delta > 0;\ n^{2-\delta} < \mathbf{m} < \frac{n^2}{\log^{1-\delta} n}$	$\lg\binom{n^2}{m}$	$(1+\epsilon)\lg\binom{n^2}{m}$
$\exists\delta > 0;\ \frac{n^2}{\log^{1-\delta} n} < \mathbf{m} \leqslant n^2$	$\lg\binom{n^2}{m}$	$\lg\binom{n^2}{m}$

- What is going on in the “middle”?
  - Upper bounds: essentially based on a space efficient version of FKS hashing
  - Lower bounds: I will prove this next class

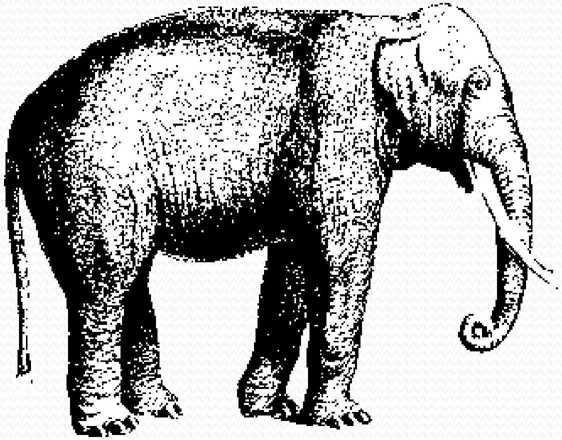
Succinct “FKS-Hashing”

- Another RaRaRa (2007) result that is very useful:
  - ***Theorem:*** *Given a bit vector of u bits, with n one bits, there is a data structure that occupies* $\log\binom{u}{n} + o(n) + O(\log\log u)$ *bits and can support the following:*
    - Rank(i): iff position i is a 1 bit (and therefore also Access(i))
    - Select(i): for all $i \in [1, n]$
- *A nice project: it is essentially FKS hashing + many incremental improvements spread over several papers*
  - *I would like to see a summary of the various techniques*

Lower Bounds (for Data Structures)

- What is the computational model?
  - Cell Probe Model:
    - Data structure D consists of S cells, each containing w bits
    - D supports some set of queries
  - We want to examine trade-offs between
    - The size of a *static* data structure
    - The number of cells, t, that must be probed during a query
      - Intermediate computation is free

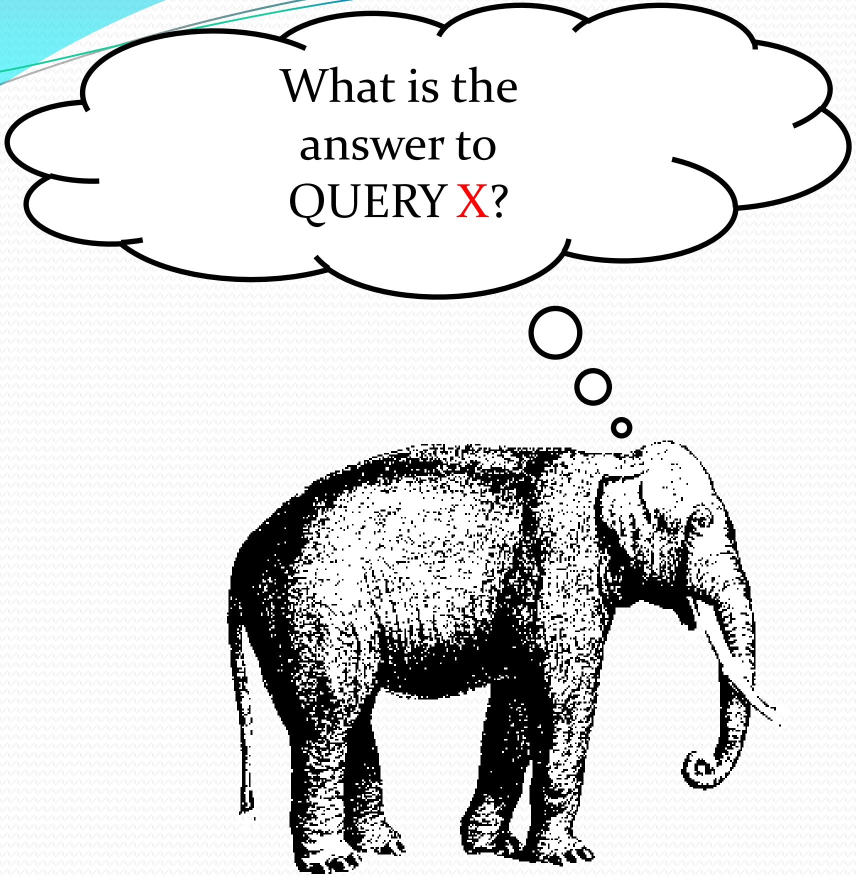
- Why do we care?
  - Cell-Probe Lower Bounds hold in the word-RAM model



USER

???	???	???	???	???
???	???	???	???	???
???	???	???	???	???
???	???	???	???	???
???	???	???	???	???

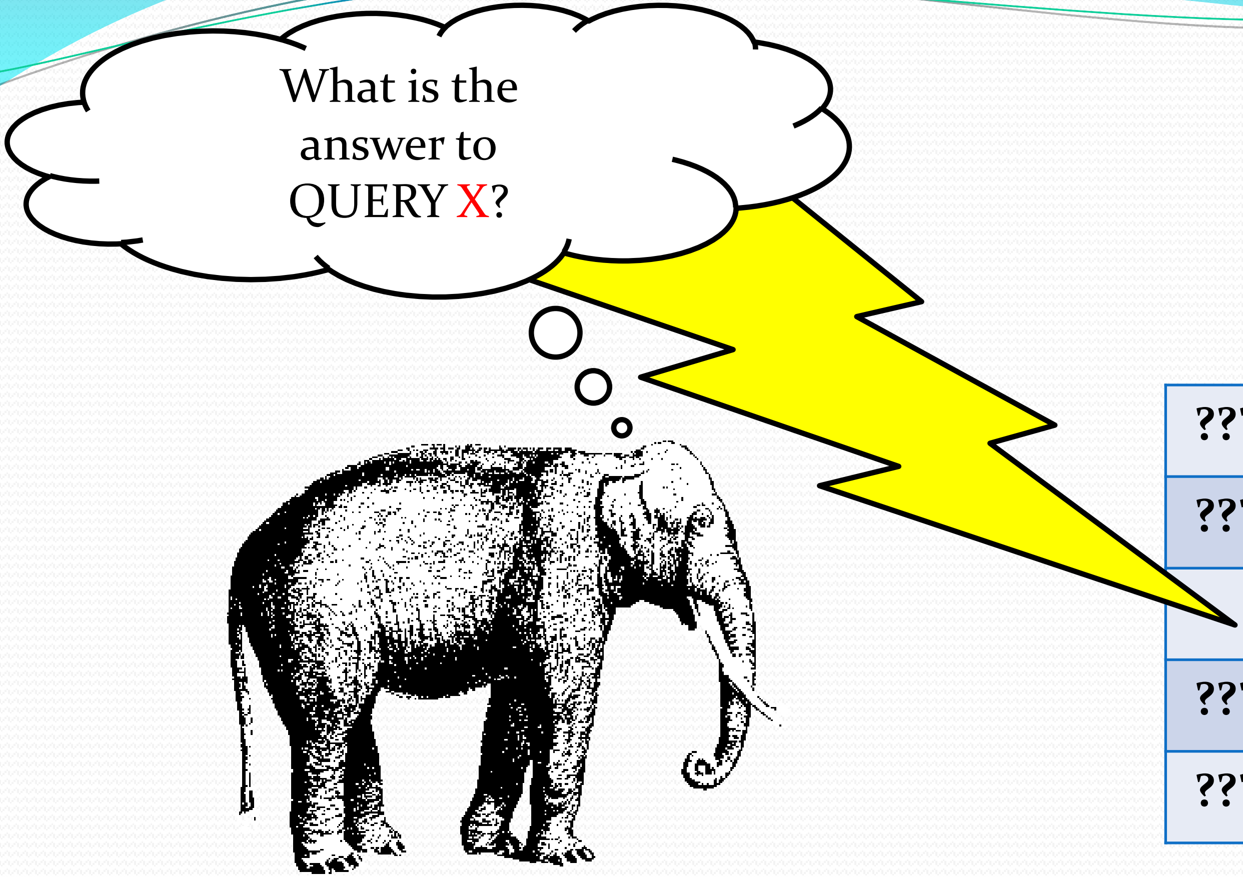
DATA STRUCTURE
s cells



USER

???	???	???	???	???
???	???	???	???	???
???	???	???	???	???
???	???	???	???	???
???	???	???	???	???

DATA STRUCTURE
s cells

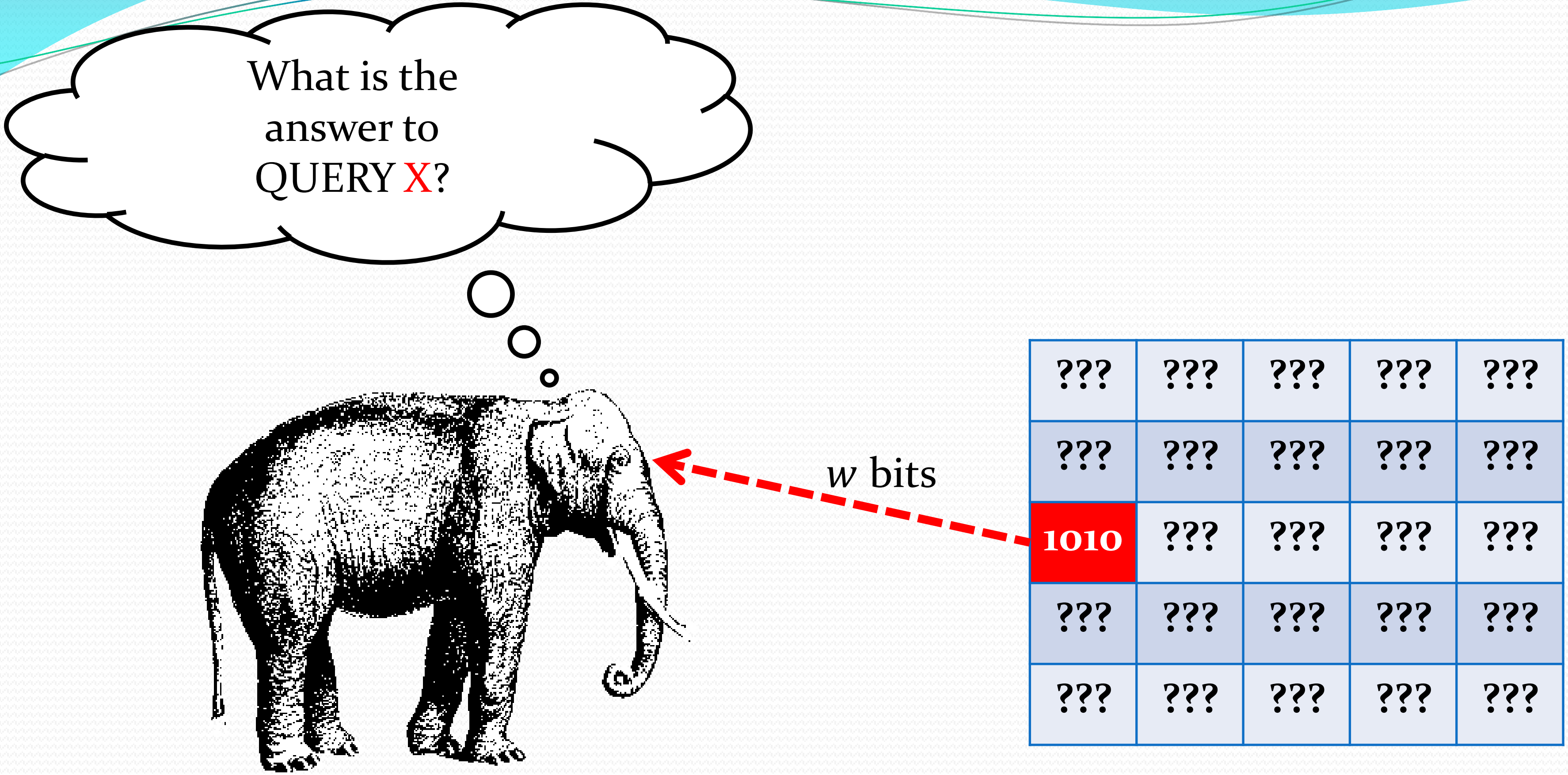


What is the answer to QUERY X?

USER

???	???	???	???	???
???	???	???	???	???
	???	???	???	???
???	???	???	???	???
???	???	???	???	???

DATA STRUCTURE
s cells



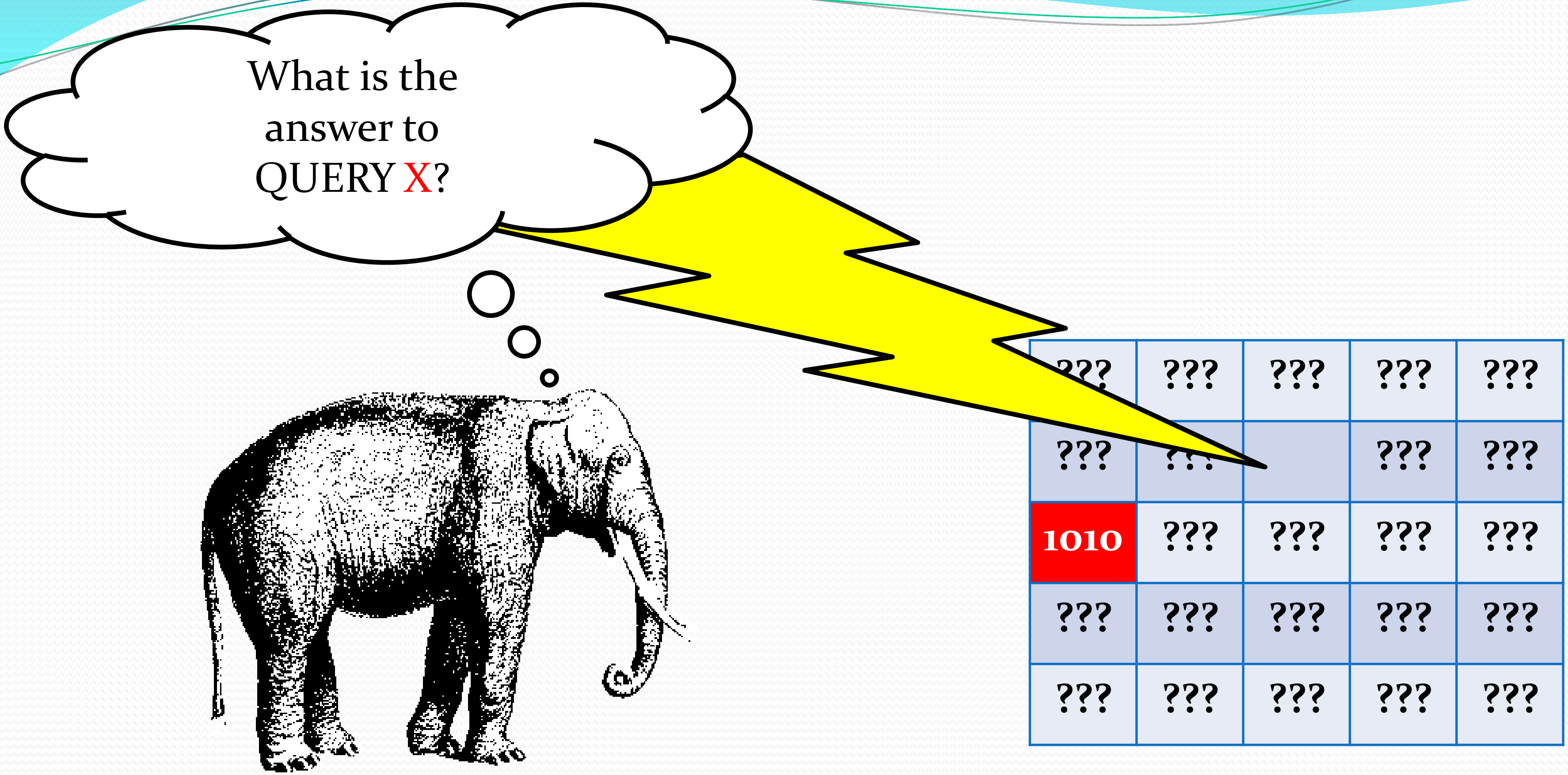
What is the answer to QUERY X?

w bits

???	???	???	???	???
???	???	???	???	???
1010	???	???	???	???
???	???	???	???	???
???	???	???	???	???

USER

DATA STRUCTURE
s cells

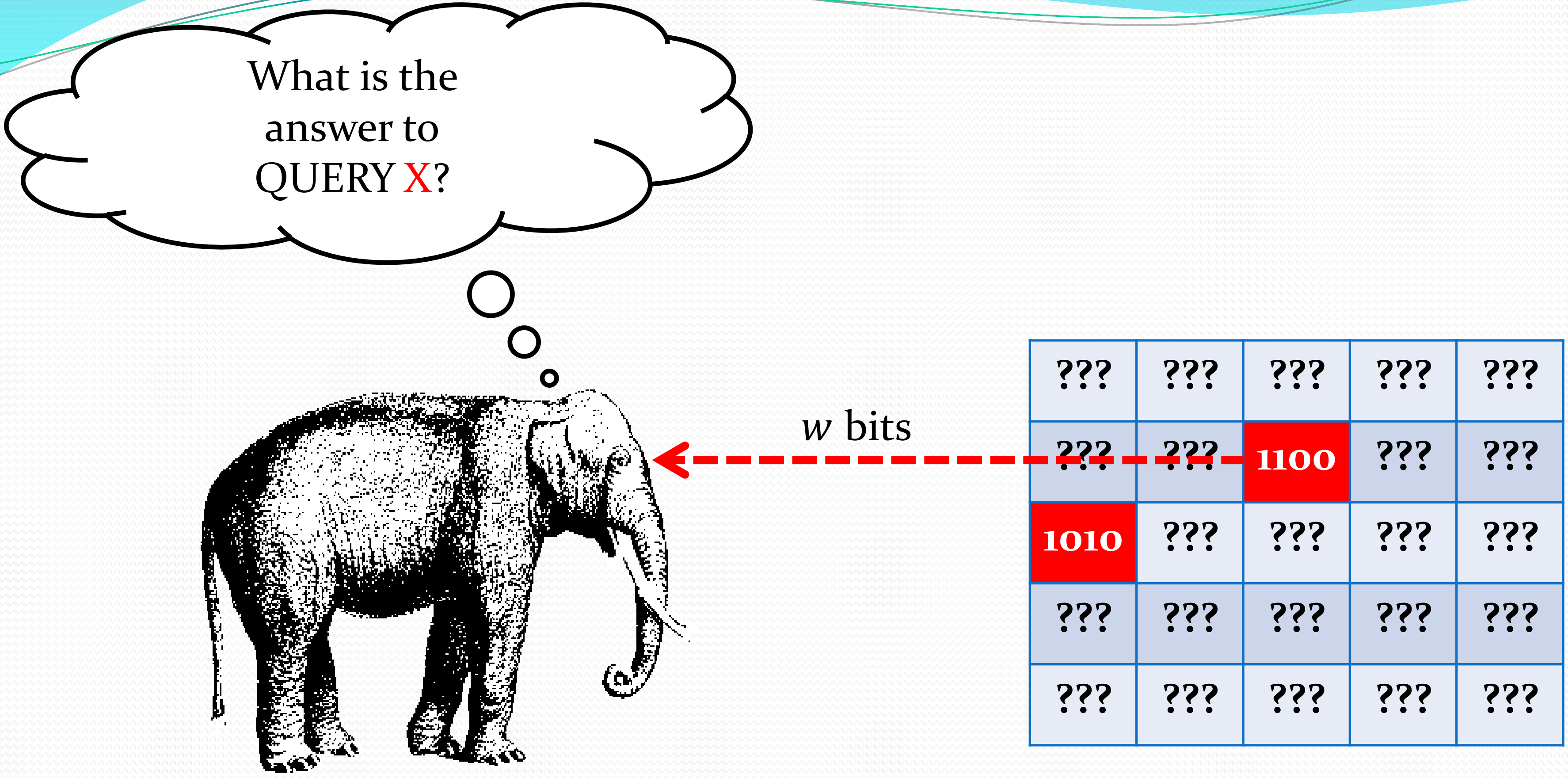


What is the answer to QUERY X?

???	???	???	???	???
???	???		???	???
1010	???	???	???	???
???	???	???	???	???
???	???	???	???	???

USER

DATA STRUCTURE
s cells



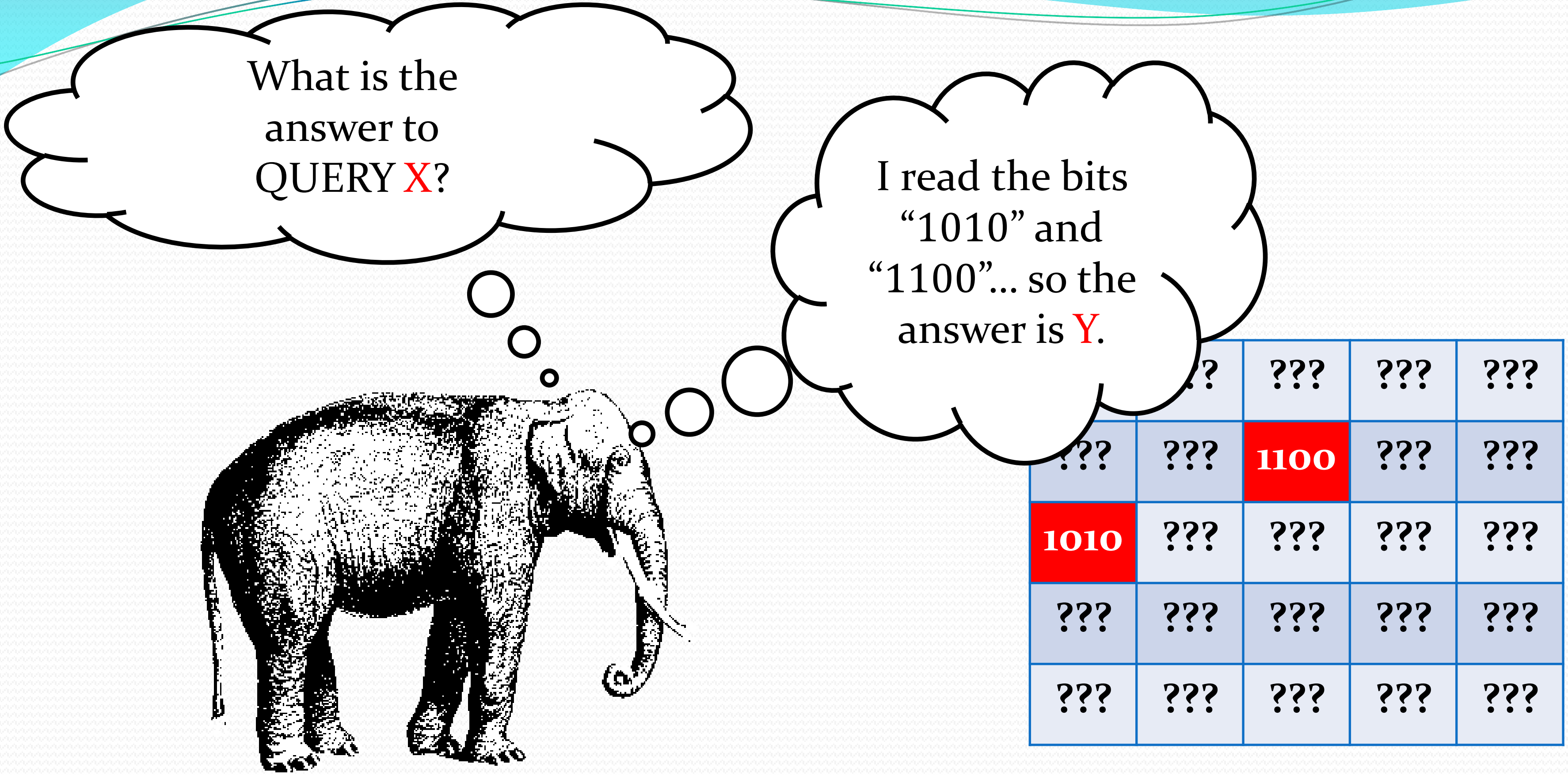
What is the answer to QUERY X?

w bits

???	???	???	???	???
???	???	1100	???	???
1010	???	???	???	???
???	???	???	???	???
???	???	???	???	???

USER

DATA STRUCTURE
s cells



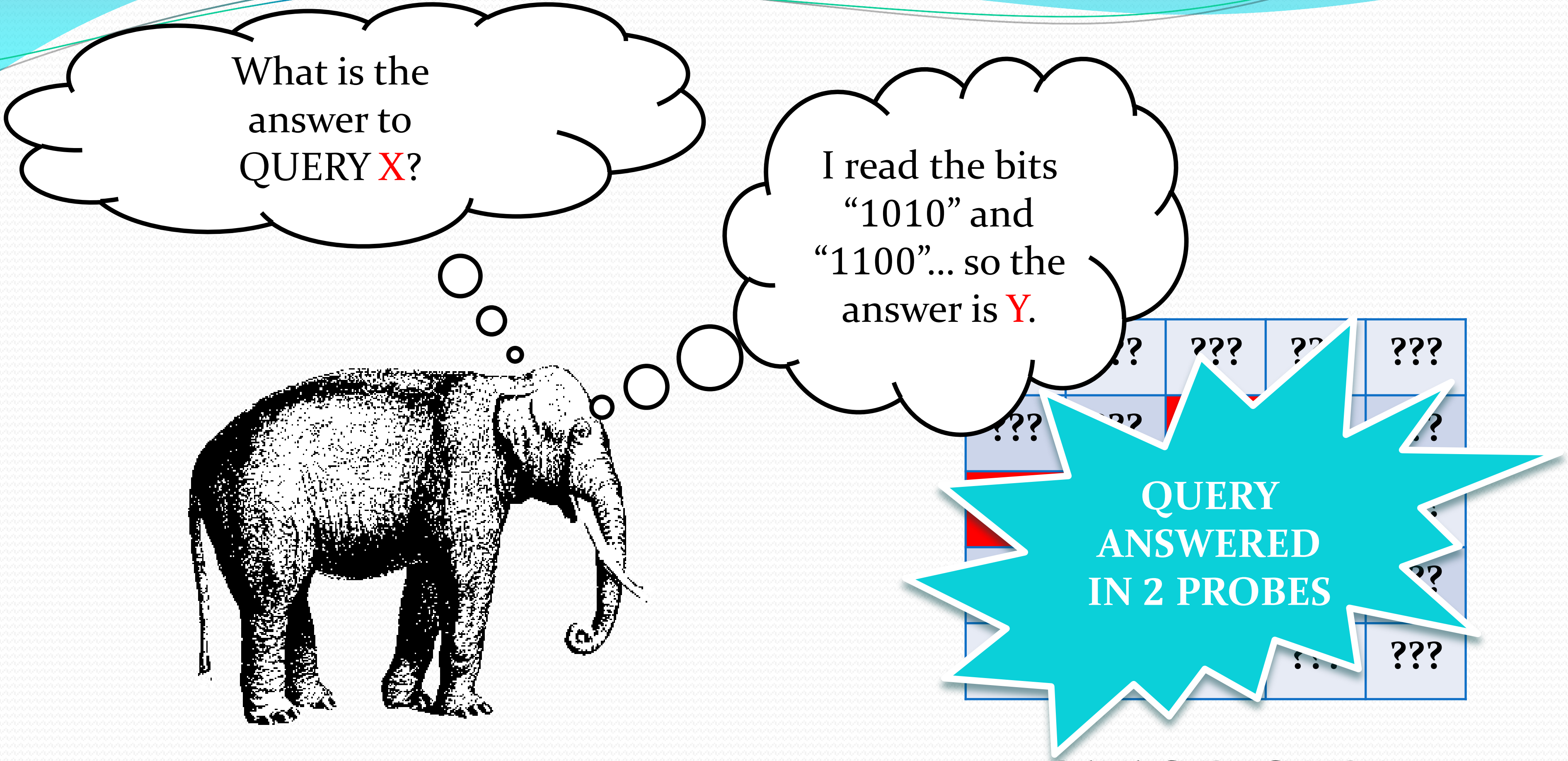
What is the answer to QUERY X?

I read the bits "1010" and "1100"… so the answer is Y.

	??	???	???	???
???	???	1100	???	???
1010	???	???	???	???
???	???	???	???	???
???	???	???	???	???

USER

DATA STRUCTURE
s cells



What is the answer to QUERY X?

I read the bits "1010" and "1100"... so the answer is Y.

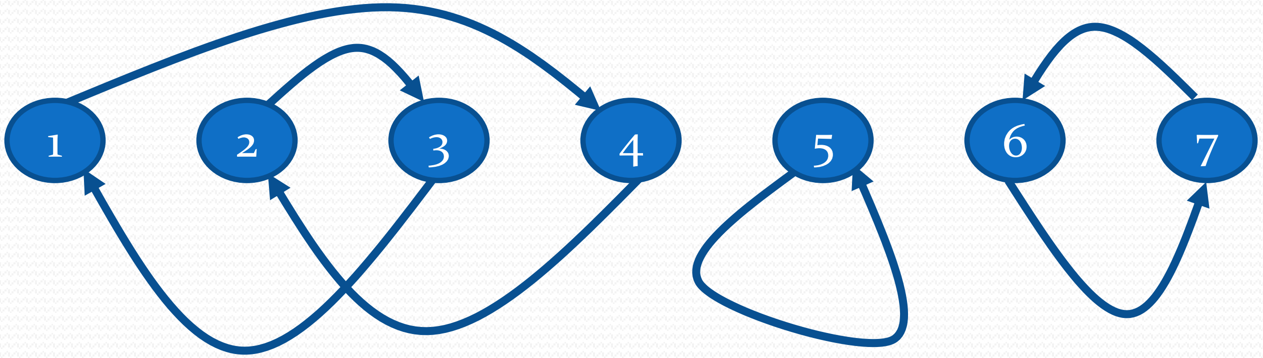
QUERY ANSWERED IN 2 PROBES

USER

DATA STRUCTURE
s cells

Problem #1: Permutations

- Represent a permutation π of size n such that we can compute $\pi(i)$ and $\pi^{-1}(i)$ for any $i \in [1, n]$
- Example: $\pi = (4,3,1,2,5,7,6)$



$\pi(1) = 4$

$\pi^{-1}(4) = 1$

Problem #1: Permutations

- There are $n!$ permutations
  - So, we need about $n \log n$ bits to represent one
- We can just store an array to represent π
  - This takes $n \log n + \Theta(n)$ bits; $\pi(i)$ in $\Theta(1)$ time
- What about computing the inverse? $\pi^{-1}(i)$
  - Simple solution: store ***two*** arrays
    - This takes $2n \log n + \Theta(n)$ bits; $\pi(i)$ and $\pi^{-1}(i)$ in $\Theta(1)$ time
- Can we do better?
  - Yes: using hashing we can, for any <u>constant</u> $\varepsilon > 0$, get $(1 + \varepsilon)n \log n$ bits; $\pi(i)$ and $\pi^{-1}(i)$ in $\Theta(1)$ time

Problem #1: Permutations

- There are $n!$ permutations
  - So, e need about n l g n bits t resent one
- We can ju e an o
  - This takes time
- What a
  - Simple s
    - Th e takes $2n$) in $\Theta(1)$ time
- Can we do bett
  - Yes: using hashing w n, for a <u>constant</u> $\varepsilon > 0$, get $(1+\varepsilon)n \log n$ bits, $\pi(i)$ and $\pi^{-1}(i)$ in $\Theta(1)$ time

NOT
SUCCINCT

Problem #2:Represent Digraphs

- Represent a digraph $G = (V, E)$ such that we can:
  - Report the i-th *in-neighbour* of a node
  - Report the j-th *out-neighbour* of a node

Report in neighbours of vertex: **c-select**(c, j)

Report out-neighbours of vertex: **r-select**(r, i)

0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0
0	0	1	0	0	1	0	0
1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	0
0	0	0	1	0	0	0	0

Problem #2:Represent Digraphs

- We can store an m edge digraph on n vertices using $\log\binom{n^2}{m} + o\left(\log\binom{n^2}{m}\right)$ bits *and* support ***one*** operation in $\Theta(1)$ time via “hashing”

- We can support both operations if the graph is very dense or sparse:
  - $m = o(n^\delta)$ for any constant $\delta > 0$
  - $m = \Omega(n^2/\log^{1-\delta} n)$ for some $\delta > 0$

- For all other ranges the best we can seem to is:

$$(1+\varepsilon)\log\binom{n^2}{m}\ \textit{bits if we want}\ \Theta(1)\ \textit{time for both operations}$$

(again, using “hashing”)

Problem #2:Represent Digraphs

- We can store an m edge digraph on n vertice sing $\log\binom{n^2}{m} +$ $o\left(\log\binom{n^2}{m}\right)$ bits *and* suppo **e** opera $\Theta(1)$ time via "hashing"

- We can su rse:
  - $m = o($
  - $m = \Omega(n^2/$

- For all other ranges es s :

$$(1+\varepsilon)\log\binom{n^2}{m}\ \textit{bits if w}\ \ \textit{nt}\ \Theta(1)\ \ \textit{e for both operations}$$

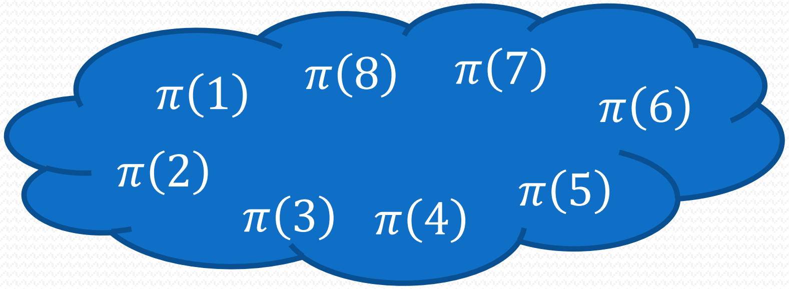
(agai using "hashing")

NOT

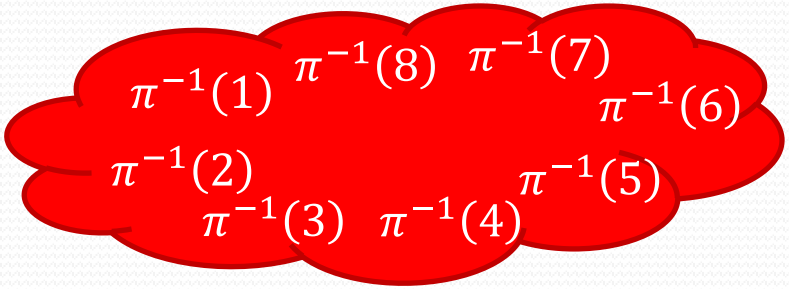
SUCCINCT

Do we need the additive ε?

- Golynski (2009): we **can't** do better for these problems
- Primary reason: the types of queries
    - The *types* of queries have the *reciprocal property*

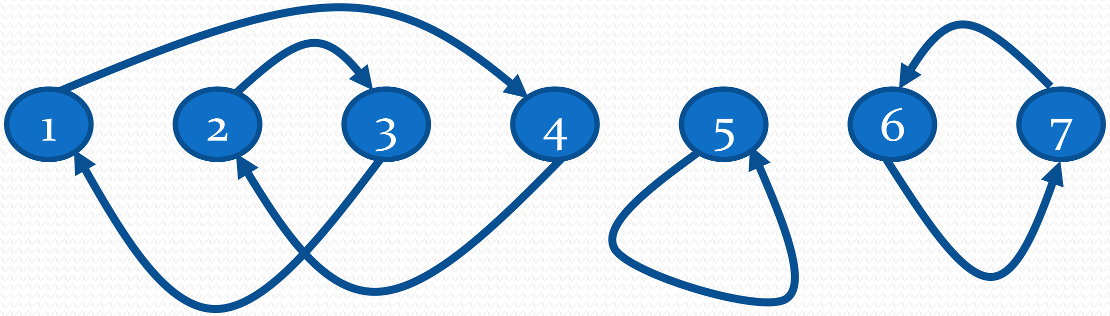

$\pi(1)$ $\pi(8)$ $\pi(7)$ $\pi(6)$ $\pi(2)$ $\pi(3)$ $\pi(4)$ $\pi(5)$

Forward Queries F_B
Example: $\pi(i)$


$\pi^{-1}(1)$ $\pi^{-1}(8)$ $\pi^{-1}(7)$ $\pi^{-1}(6)$ $\pi^{-1}(2)$ $\pi^{-1}(3)$ $\pi^{-1}(4)$ $\pi^{-1}(5)$

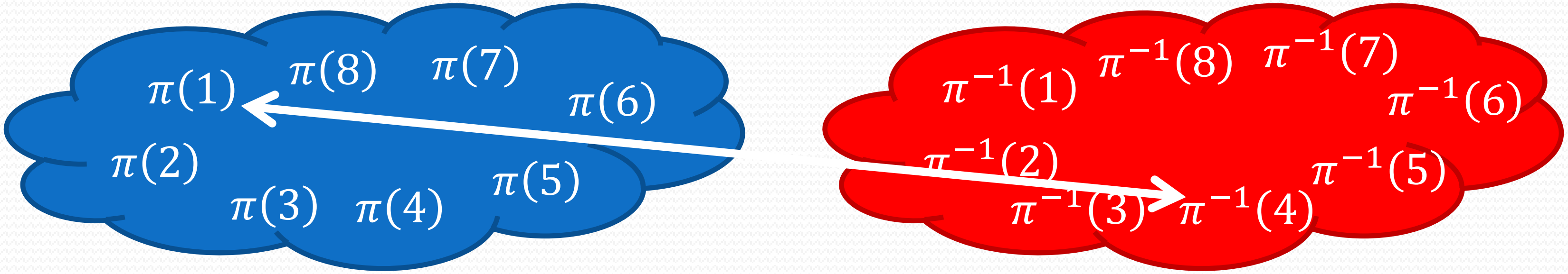
Inverse Queries I_B
Example: $\pi^{-1}(i)$

Object B



Reciprocal Property

- Let F_B be the set of forward queries for object B
- Let I_B be the set of inverse queries for object B
- There is a bijection $\eta\colon F_B \to I_B$ between these sets



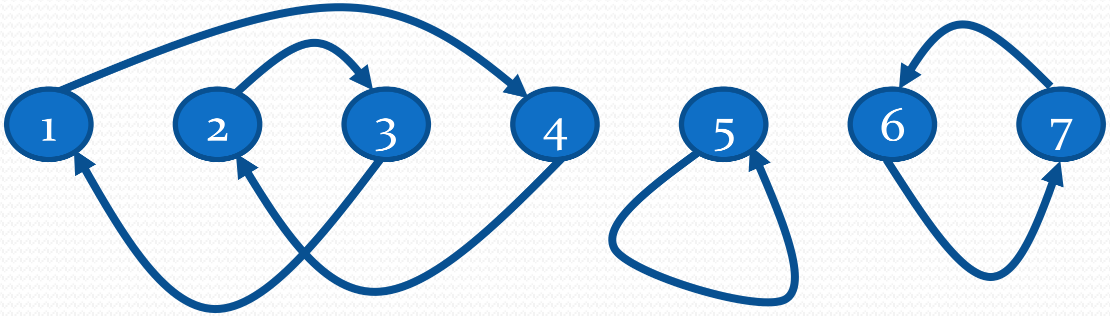
$\pi(1)$ $\pi(8)$ $\pi(7)$ $\pi(6)$ $\pi(2)$ $\pi(3)$ $\pi(4)$ $\pi(5)$

$\pi^{-1}(1)$ $\pi^{-1}(8)$ $\pi^{-1}(7)$ $\pi^{-1}(6)$ $\pi^{-1}(2)$ $\pi^{-1}(5)$ $\pi^{-1}(3)$ $\pi^{-1}(4)$

Forward Queries F_B
Example: $\pi(i)$

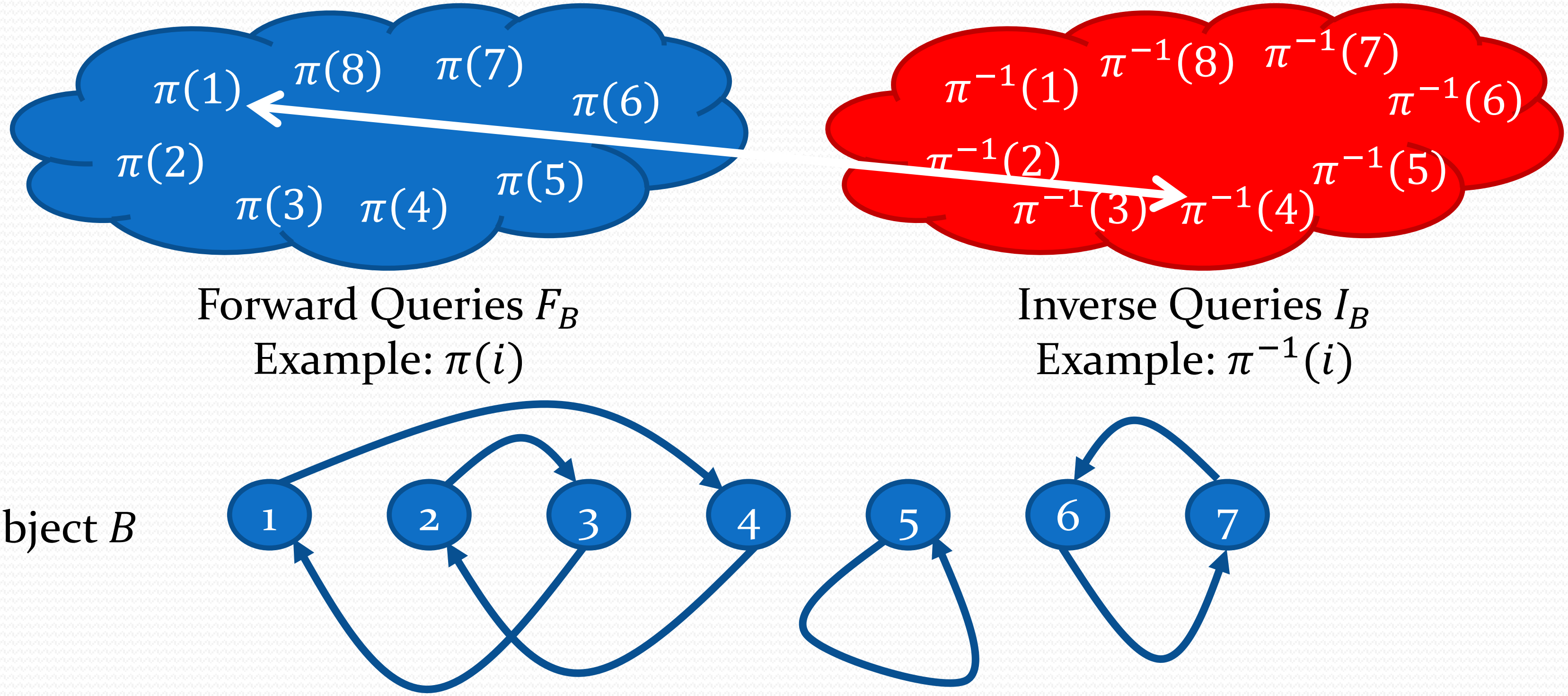
Inverse Queries I_B
Example: $\pi^{-1}(i)$

Object B



Reciprocal Property (2)

- Suppose we have a description of the sets F_B and I_B
- … and we know the answers to $F_B^* \subseteq F_B$ and $I_B^* \subseteq I_B$
- … and for the *remaining queries* we know the bijection



Reciprocal Property (2)

- Suppose we have a description of the sets F_B and I_B
- … and we know the answers to $F_B^* \subseteq F_B$ and $I_B^* \subseteq I_B$
- … and for the *remaining queries* we know the bijection
    - That is: for all queries $F_B' = F_B \setminus F_B^* \setminus \eta^{-1}(I_B^*)$ we know the corresponding inverse query in $I_B' = I_B \setminus I_B^* \setminus \eta(F_B^*)$

If, with the above information we can reconstruct the object B, then B has the *reciprocal property*

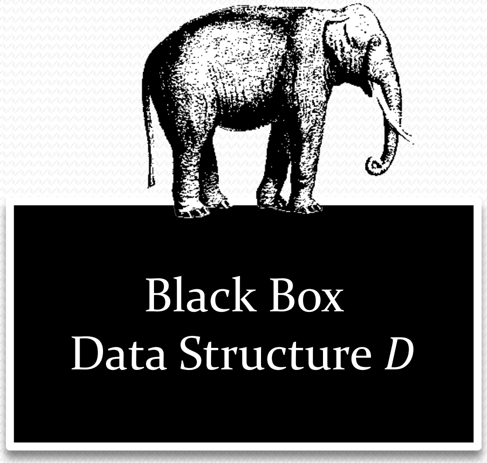
Outline of Lower Bound

- The lower bound is based on *round elimination*
  - Suppose we have a data structure D for representing B
    - B has the reciprocal property
    - Probes t cells in D to answer any forward/inverse query
  - We design a compression algorithm which:
    - In a single round: *deletes* and *protects* some cells in D
    - Writes out some information to *recover* the lost information
    - Does this until a constant fraction of the cells are deleted
  - Under certain conditions:

$$\text{amount written} \ll \text{amount deleted}$$

Outline of Implications

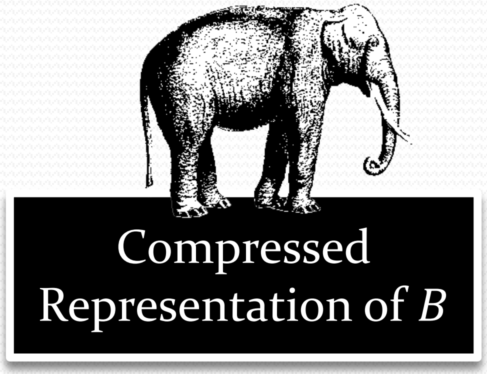
- D can be used to *uniquely identify* B
- Assume object B *requires* Υ *cells* to be represented
- Let R be the # of additional bits for compression
- If D occupies S cells then $(1-\varepsilon)S + \frac{R}{w} + O(1) \geq \Upsilon$
- Therefore, D cannot be succinct if $\frac{R}{w} = o(\Upsilon)$



Black Box
Data Structure D

Outline of Implications

- D can be used to *uniquely identify* B
- Assume object B *requires* Υ *cells* to be represented
- Let R be the # of additional bits for compression
- If D occupies S cells then $(1-\varepsilon)S + \frac{R}{w} + O(1) \ \geq \Upsilon$
- Therefore, D <u>cannot be succinct</u> if $\frac{R}{w} = o(\Upsilon)$



Compressed Representation of B

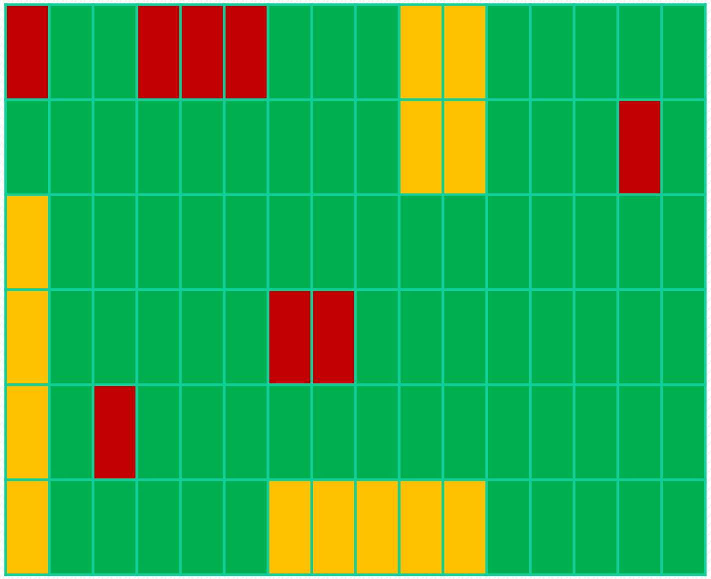
The Lower Bound: Set Up

- Let’s simplify things a bit…
  - Focus on problem #2: representing a digraph

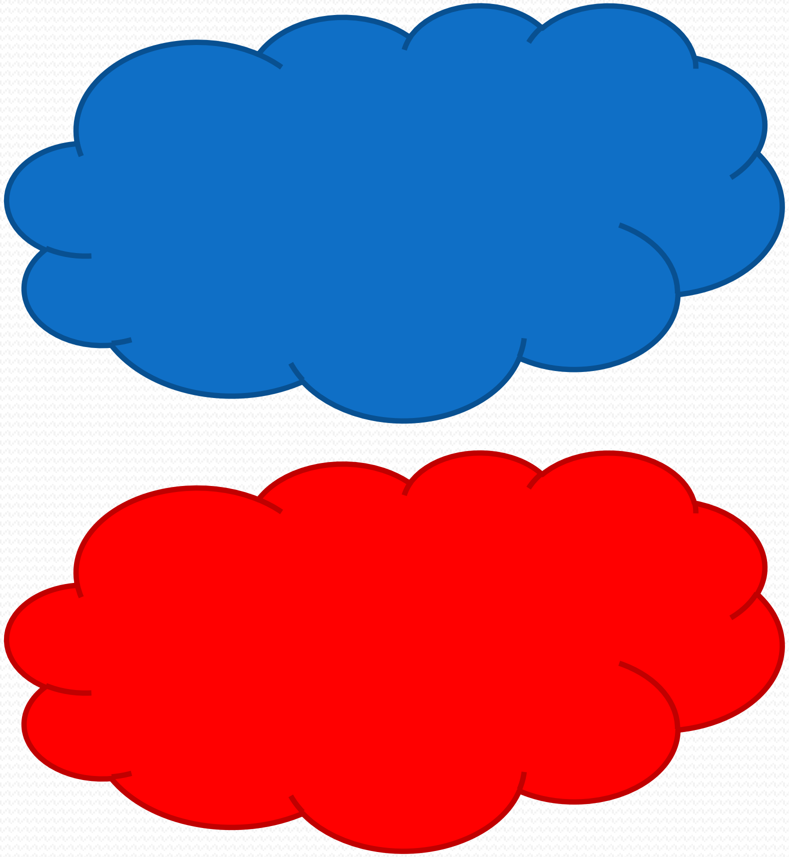
- Store an S cell structure D representing digraph G
  - Assume forward/inverse queries probe $t = \Theta(1)$ cells
  - Let C_k denote number of *remaining cells* before round k:
    - A cell is remaining if *not deleted or protected*
    - **Key Invariant: $\boldsymbol{C_k \geq S/2}$**
  - Let m be the total number edges in G: $m = |F_B| = |I_B|$

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



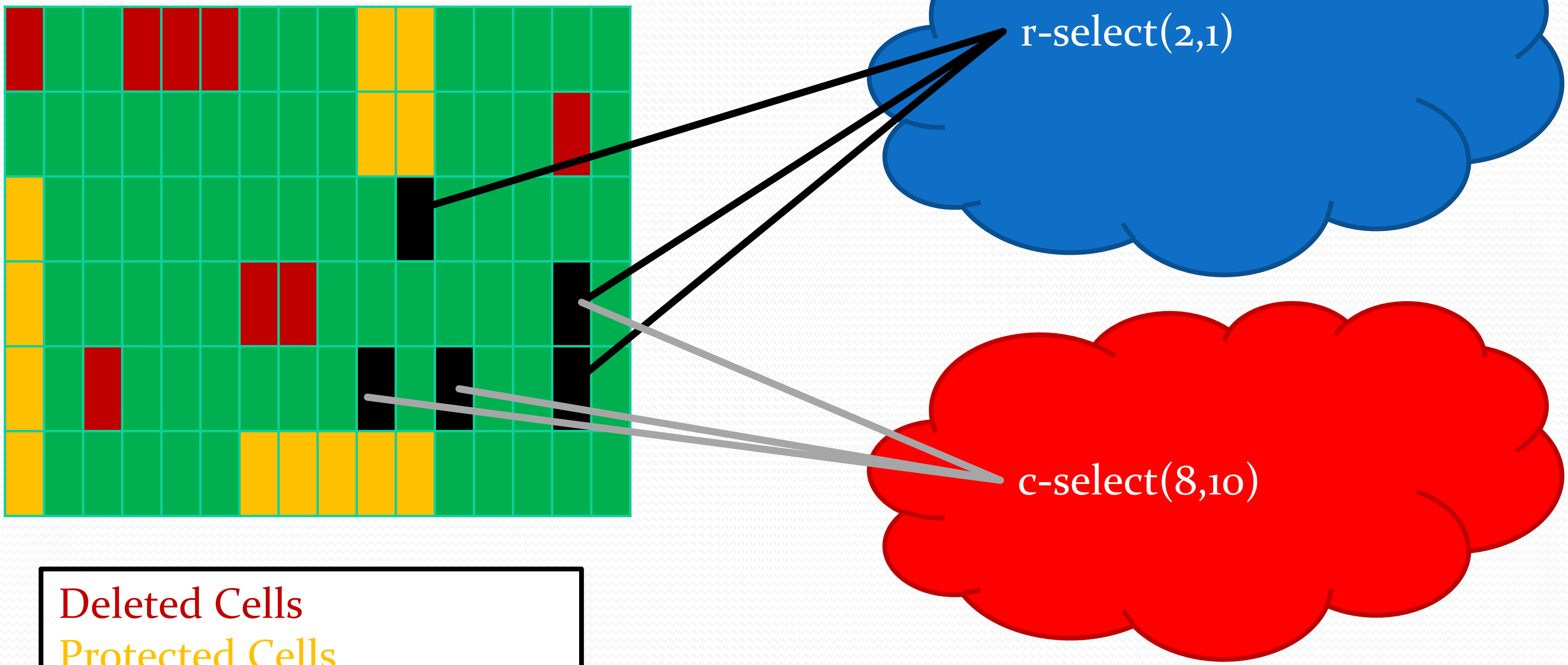
Deleted Cells
Protected Cells
Remaining Cells



Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



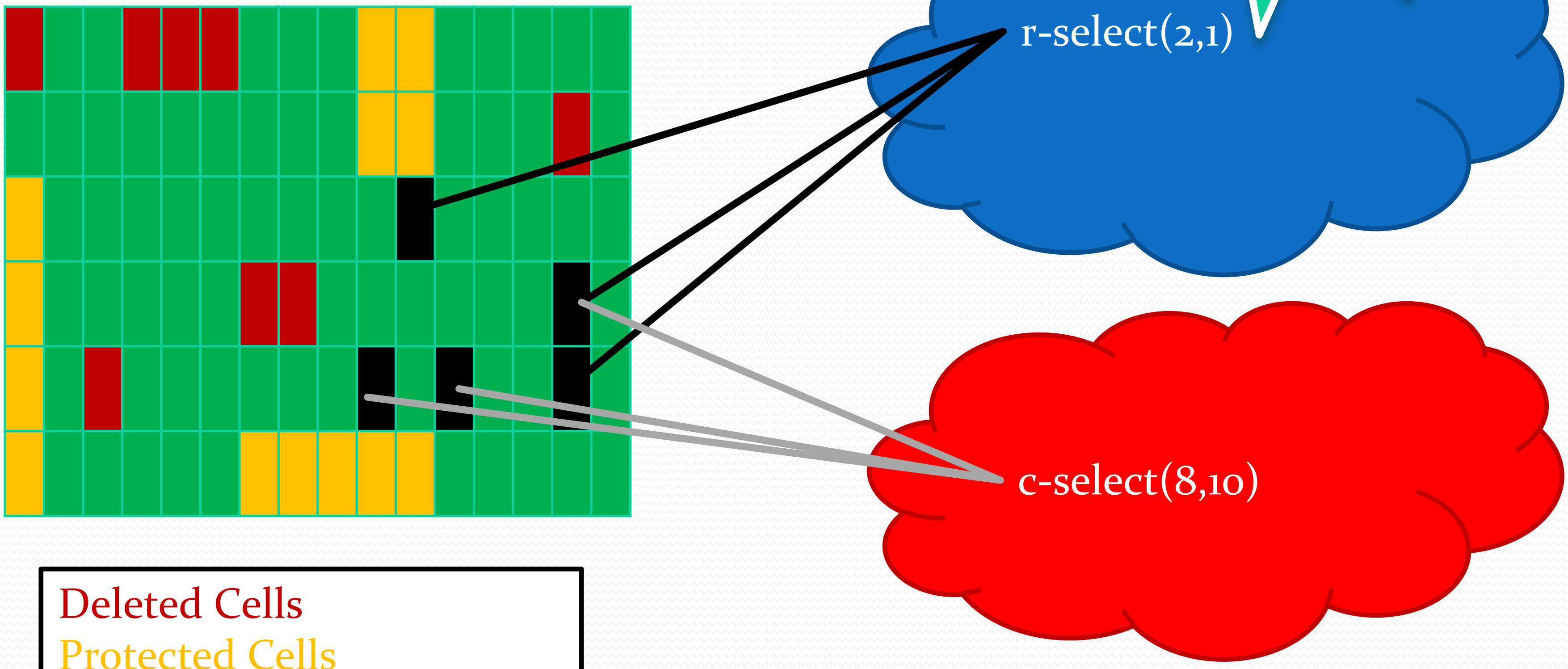
Remaining Queries

Deleted Cells
Protected Cells
Remaining Cells

Proof with Pictures

Each query inspects at most t cells

Data Structure D occupies S cells
C_k cells remaining

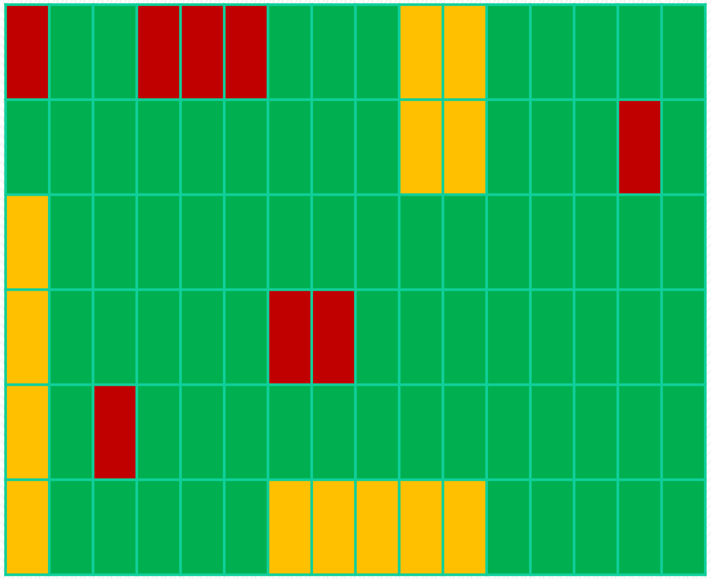


Deleted Cells
Protected Cells
Remaining Cells

Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



Deleted Cells
Protected Cells
Remaining Cells

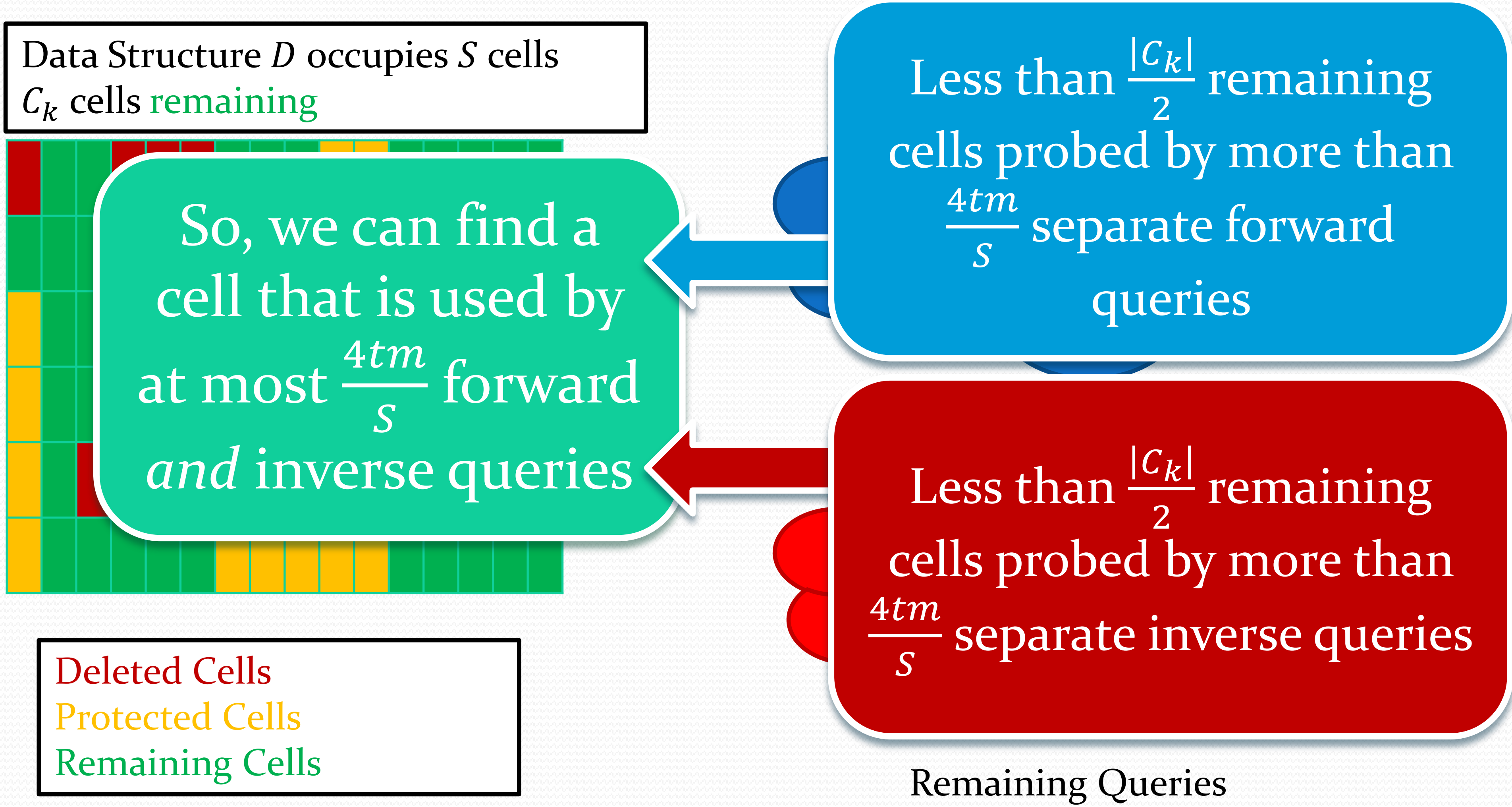
Less than $\frac{|C_k|}{2}$ remaining cells probed by more than $\frac{4tm}{S}$ separate forward queries

Less than $\frac{|C_k|}{2}$ remaining cells probed by more than $\frac{4tm}{S}$ separate inverse queries

Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



So, we can find a cell that is used by at most $\frac{4tm}{S}$ forward *and* inverse queries

Less than $\frac{|C_k|}{2}$ remaining cells probed by more than $\frac{4tm}{S}$ separate forward queries

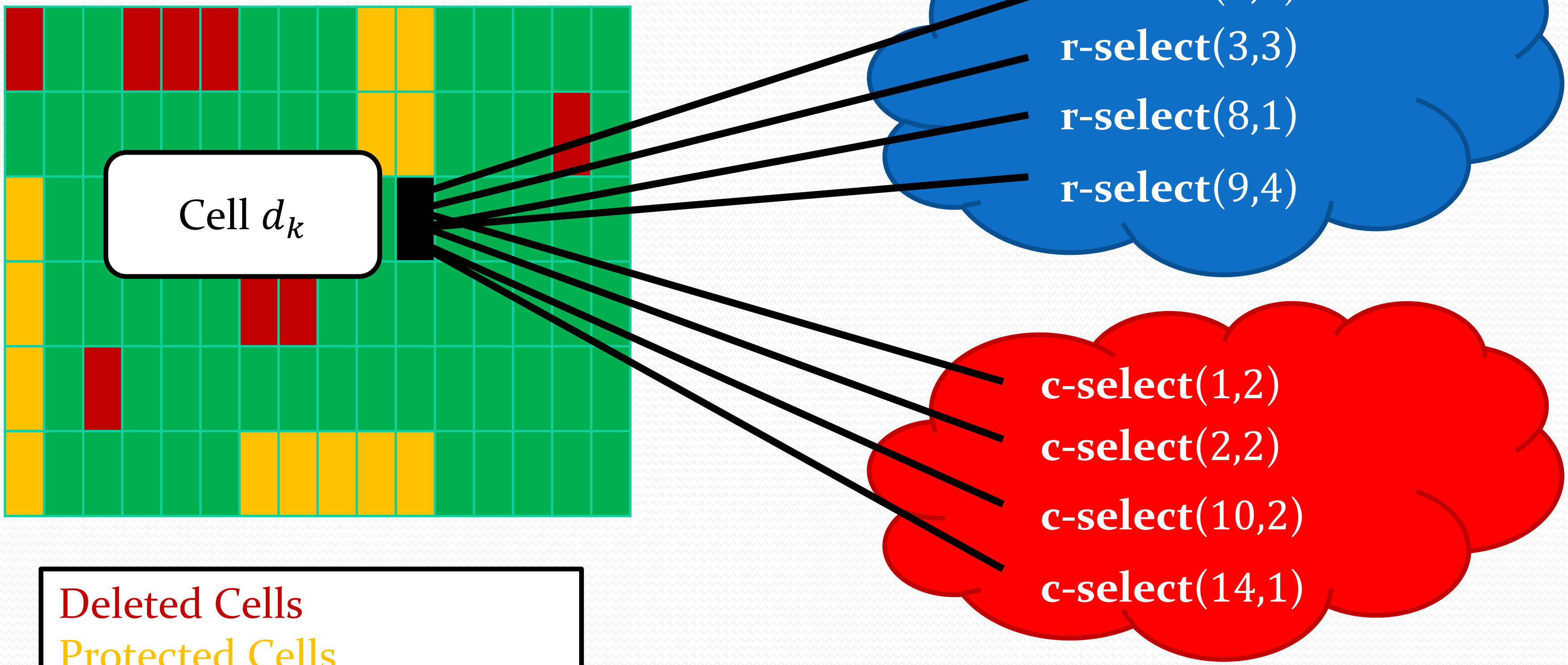
Less than $\frac{|C_k|}{2}$ remaining cells probed by more than $\frac{4tm}{S}$ separate inverse queries

Deleted Cells
Protected Cells
Remaining Cells

Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



Cell d_k

r-select(2,1)
r-select(3,3)
r-select(8,1)
r-select(9,4)

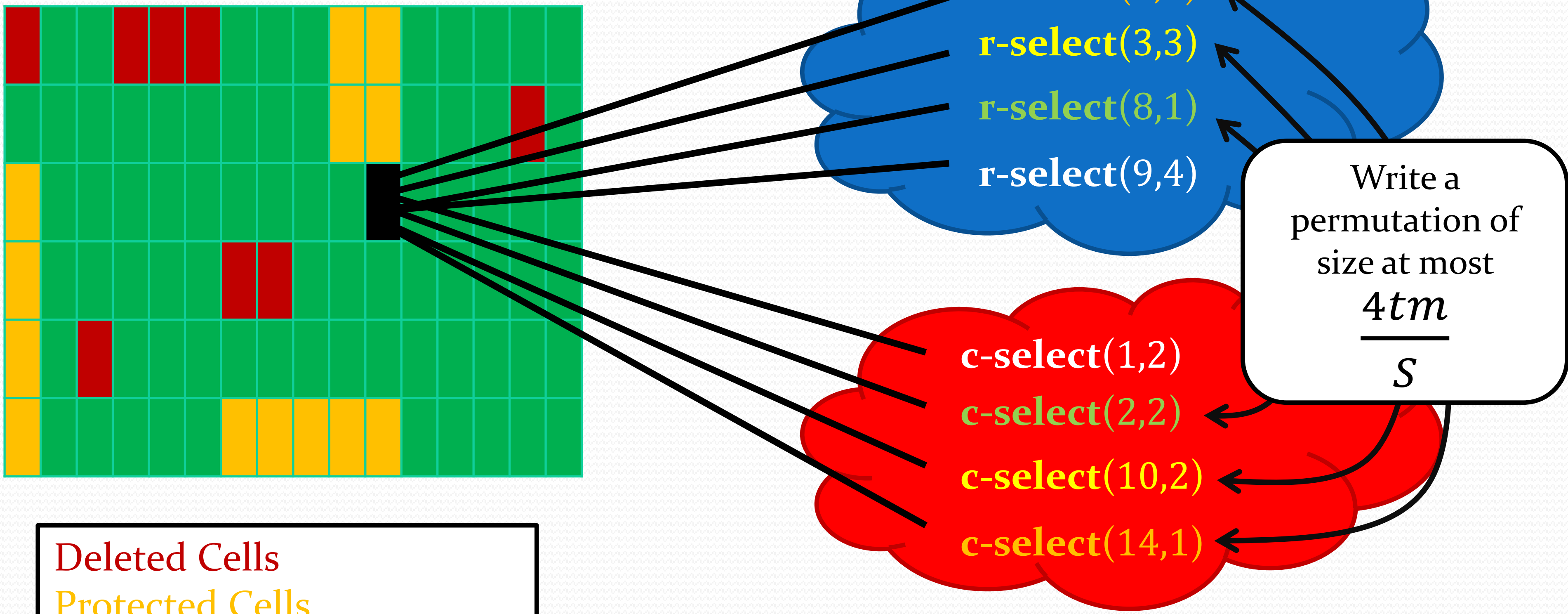
c-select(1,2)
c-select(2,2)
c-select(10,2)
c-select(14,1)

Deleted Cells
Protected Cells
Remaining Cells

Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining

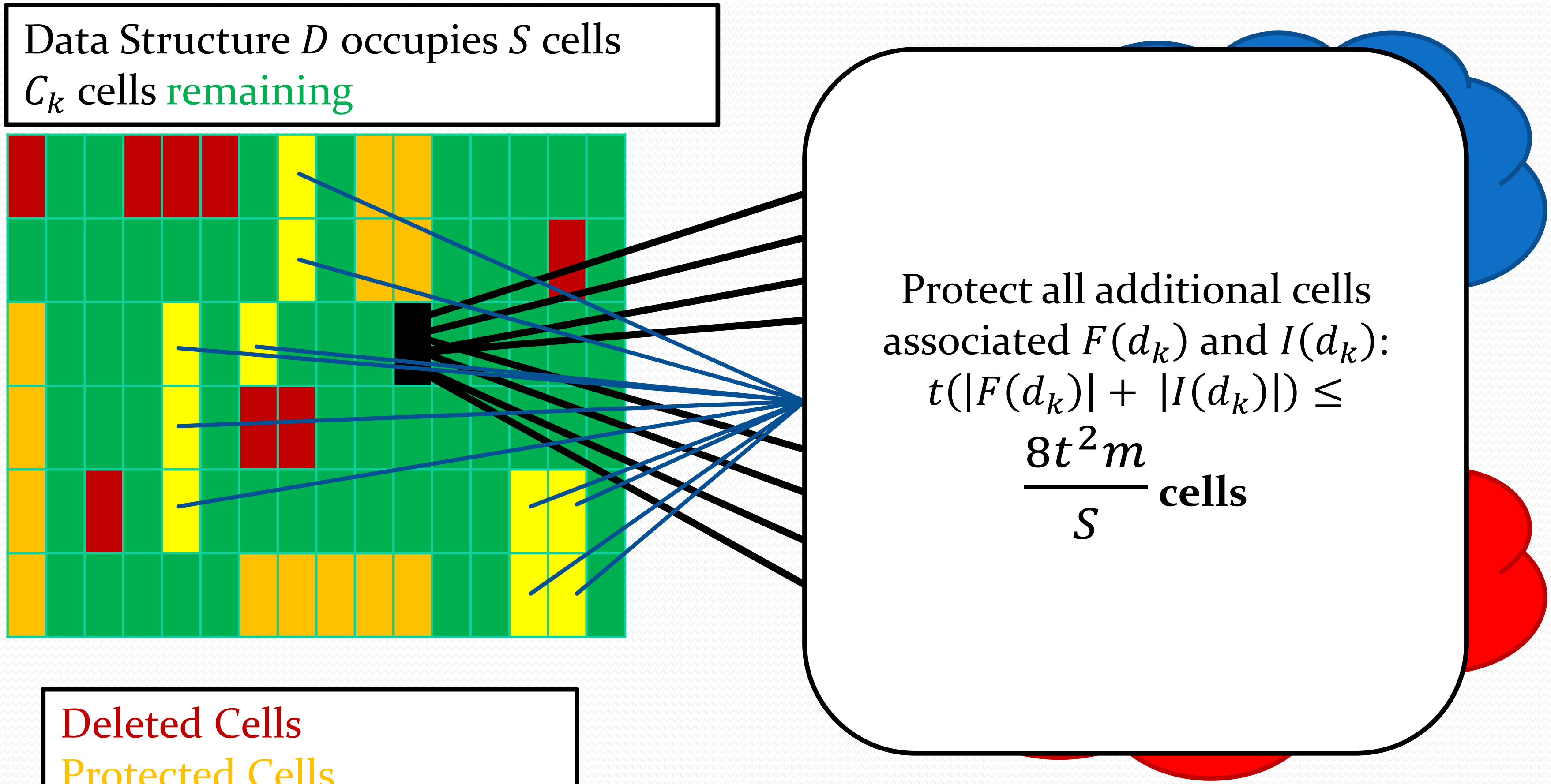


Deleted Cells
Protected Cells
Remaining Cells

Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



Protect all additional cells associated $F(d_k)$ and $I(d_k)$: $t(|F(d_k)| + |I(d_k)|) \leq \frac{8t^2m}{S}$ **cells**

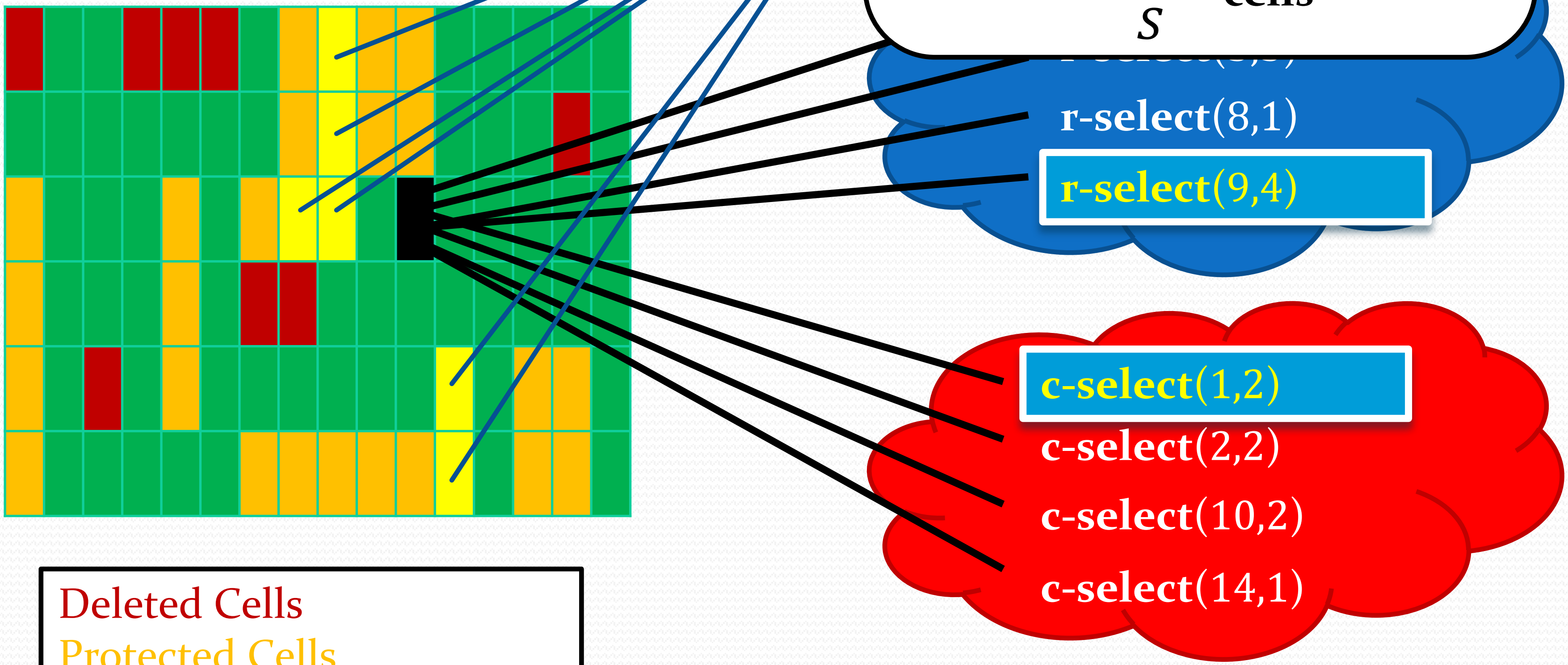
Deleted Cells
Protected Cells
Remaining Cells

Remaining Queries

Proof with Pictur

Data Structure D occupies S cells
C_k cells remaining

Also, for queries whose inverses are not $F(d_k) \cup I(d_k)$: find and protect cells associated with the inverses: at most $t(|F(d_k)| + |I(d_k)|) \le \frac{8t^2m}{S}$ **cells**

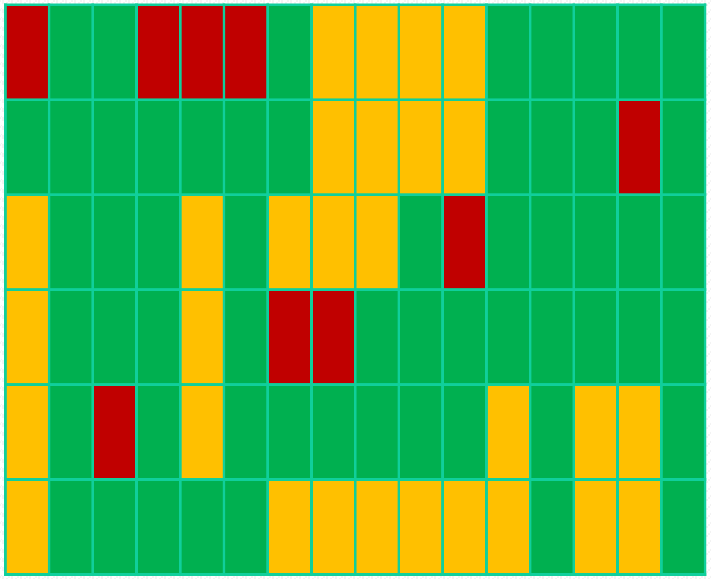


Deleted Cells
Protected Cells
Remaining Cells

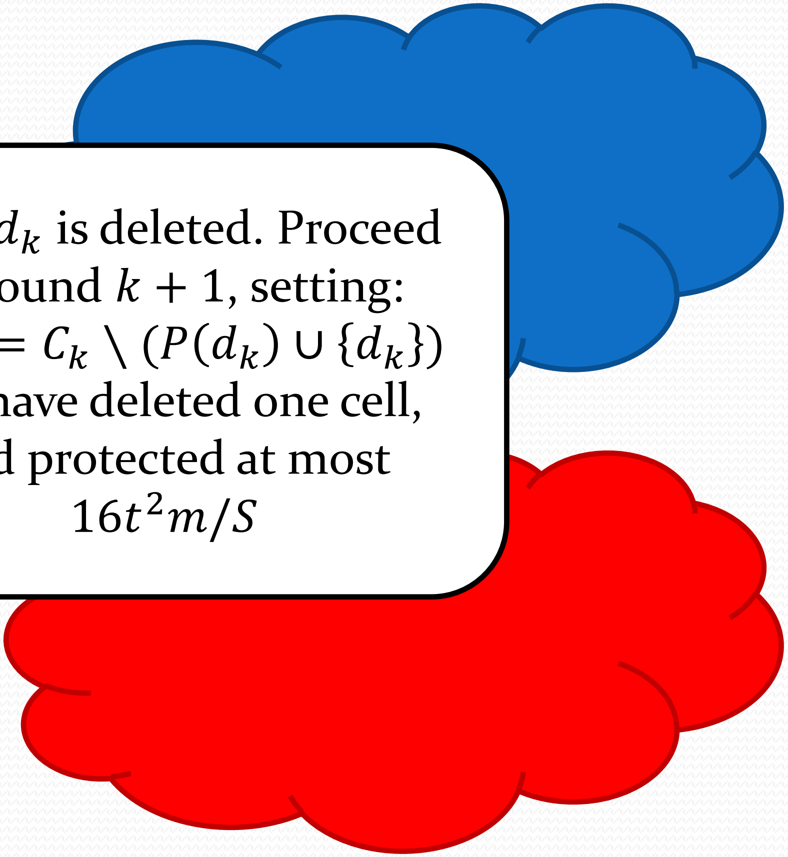
Remaining Queries

Proof with Pictures

Data Structure D occupies S cells
C_k cells remaining



Deleted Cells
Protected Cells
Remaining Cells



Now d_k is deleted. Proceed to round $k+1$, setting:

$$C_{k+1} = C_k \setminus (P(d_k) \cup \{d_k\})$$

We have deleted one cell, and protected at most $16t^2m/S$

Remaining Queries

Remaining Details Without Pictures

- How many cells remain after round k:
- $C_{k+1} = S - \sum_k P(d_i) - k \geq S - \frac{16k(t^2m+S)}{S}$
- So, if the total number of rounds is z then

$$z = S^2/(32(t^2m + S))$$

is sufficient to maintain invariant $C_k \geq S/2$

What to Store?

- Store the locations of the deleted cells
  - This takes $\log\binom{S}{z}$ **bits**
- Store the contents of all non-deleted cells compacted
  - This takes $S\ - z$ **cells** of w bits
- Store all the permutations for deleted cells (lex. order)
  - This takes $z\log\left(\left(\frac{4tm}{S}\right)!\right) \leq z\frac{4tm}{S}\log\frac{4tm}{S}$ **bits**
- Store an encoding of all the queries for rows/columns:
  - This takes $2\log\binom{m+n}{n}$ **bits**

Implications for the Compression

- *If* we can recover G then it must be the case that

$$S - z + \frac{R}{w} \geq \Upsilon - O(1)$$

- Assume $S \geq \Upsilon$ where $\Upsilon = \log\binom{n^2}{m}/w$, and $w = \Theta(\log n)$:

$$R = \log\binom{S}{z} + z\frac{4tm}{S}\log\frac{4tm}{S} + 2\log\binom{m+n}{n}$$

- This simplifies to:

$$R = O\left(\frac{S^2\log\left(\frac{m+S}{S}\right)}{m+S} + \frac{Sm}{m+S}\log\frac{m}{S} + n\log\frac{m+n}{n}\right)$$

So, **if** $m = n^{1+\delta}$ for some constant $\delta \in (0,1)$

then $\frac{R}{w} = o(\Upsilon)$

but $z = \Omega(\Upsilon)$!

How to Recover G (Non-Technical)

- All that remains is to describe how to recover G
- We can simulate queries on the non-deleted part of D
  - There are three types of queries:
    1. Queries that succeed without requesting deleted cells
    2. Queries that fail but their reciprocal succeeds
    3. Queries that fail and their reciprocal fails on same deleted cell
  - We can detect which type of query we are dealing with
    - First one is not a problem
    - For the second and third type:
      - We identify the subset of queries for a deleted cell
      - Enumerate these in the lex. order used to store the permutations
      - Determine whether query *participates* in the permutation or not

Conclusion

- Some operations don't permit a succinct data structure
  - We have seen two:
    - Forward/Inverse in a permutation
    - Listing in and out-neighbours in a digraph
  - Golynski discusses one other:
    - Search and access in a text
- Interesting open problems:
  - For digraphs can bounds be made output sensitive?
  - Bounds only apply when # queries ~ ITLB
    - Can we come up with a more general theorem?

Non-Binary Rank and Select

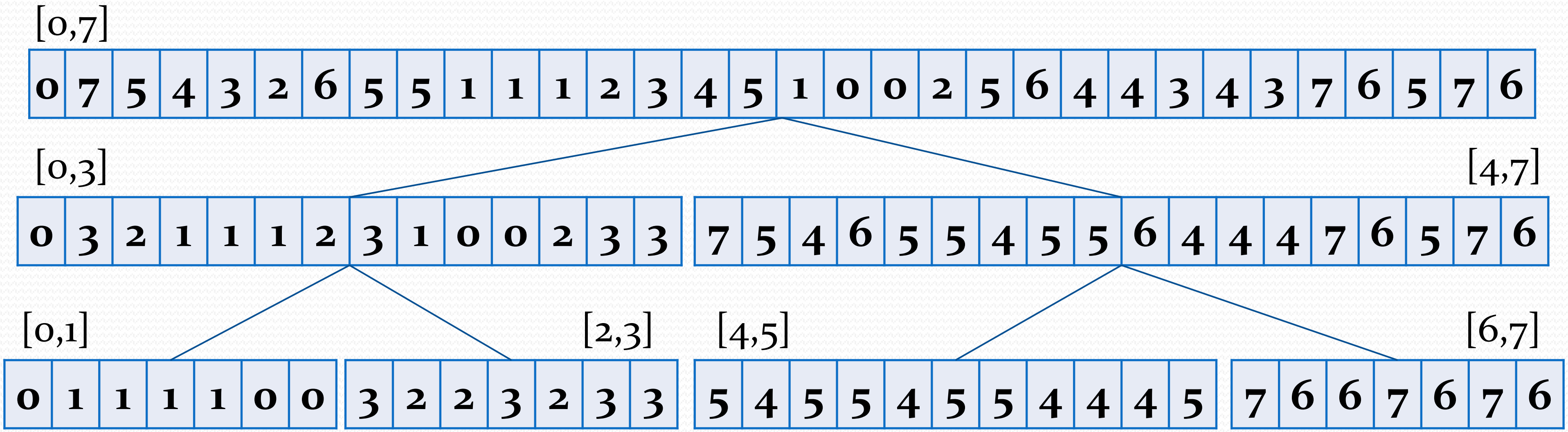
- Consider the following array of n numbers:

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6

- Such an array can represent:
  - Documents: string of n symbols from alphabet $[0, \sigma - 1]$
  - Point Sets: n points on a $n \times \sigma$ grid
- Two "Natural" Operations:
  - Rank(i, α): return the # of occurrences of α up to pos. i
  - Select(i, α): return the index of the i-th α

How To Do It

- Make a tree: Divide alphabet in half at each node



[0,7]

0 7 5 4 3 2 6 5 5 1 1 1 2 3 4 5 1 0 0 2 5 6 4 4 3 4 3 7 6 5 7 6

[0,3]

0 3 2 1 1 1 2 3 1 0 0 2 3 3

[4,7]

7 5 4 6 5 5 4 5 5 6 4 4 4 7 6 5 7 6

[0,1]

0 1 1 1 1 0 0

[2,3]

3 2 2 3 2 3 3

[4,5]

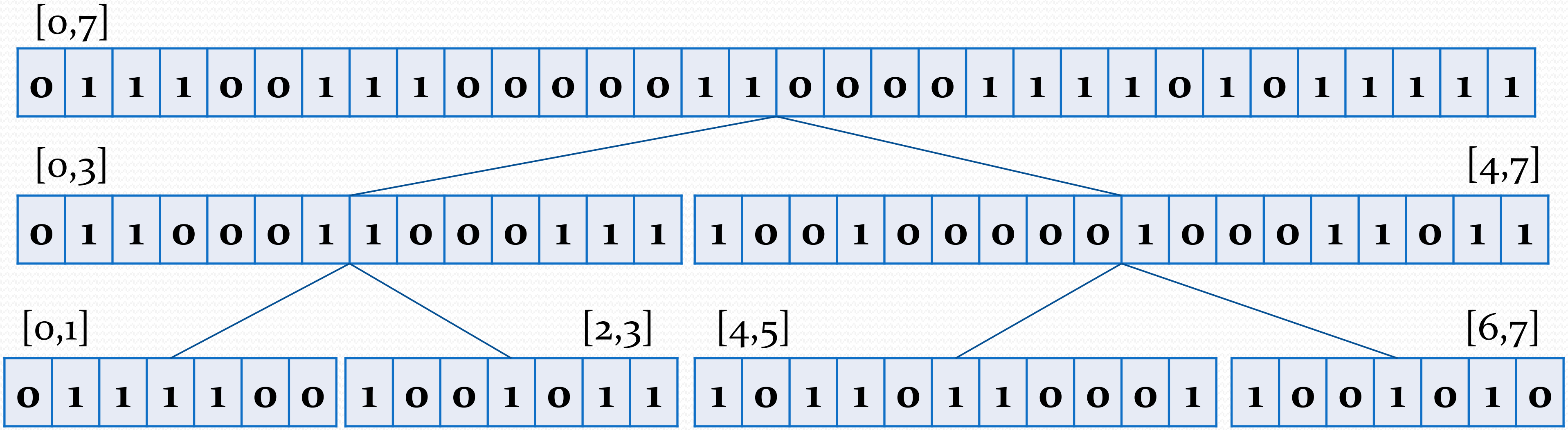
5 4 5 5 4 5 5 4 4 4 5

[6,7]

7 6 6 7 6 7 6

How To Do It

- Make a tree: Just store a bit vector at each node



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

[4,5]

1 0 1 1 0 1 1 0 0 0 1

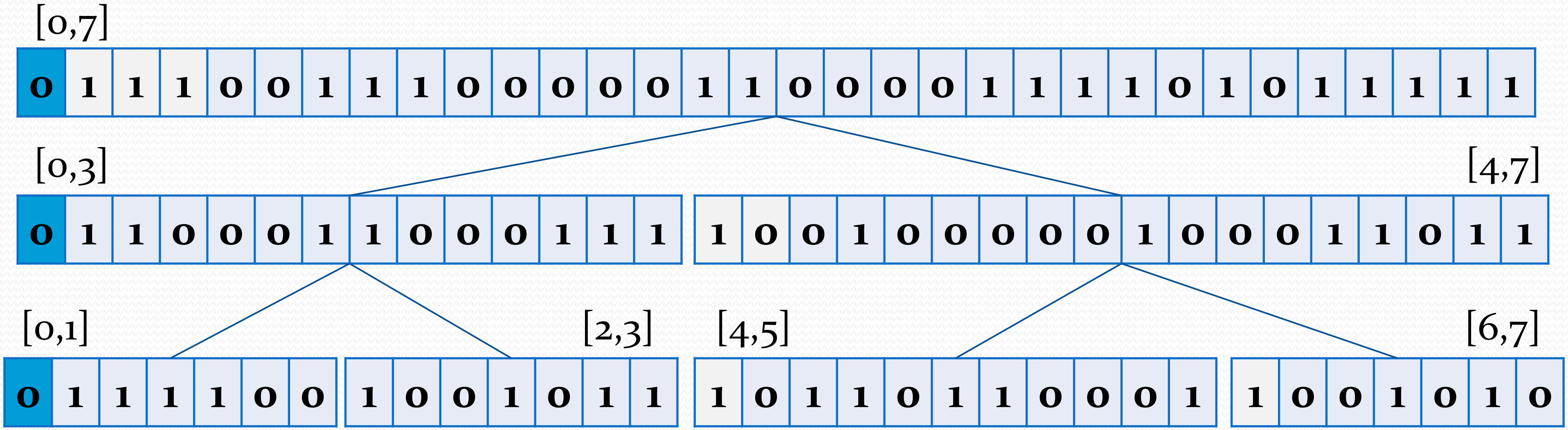
[6,7]

1 0 0 1 0 1 0

This is called a *wavelet tree*

How To Do It

- Make a tree: Just store a bit vector at each node



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

[4,5]

1 0 1 1 0 1 1 0 0 0 1

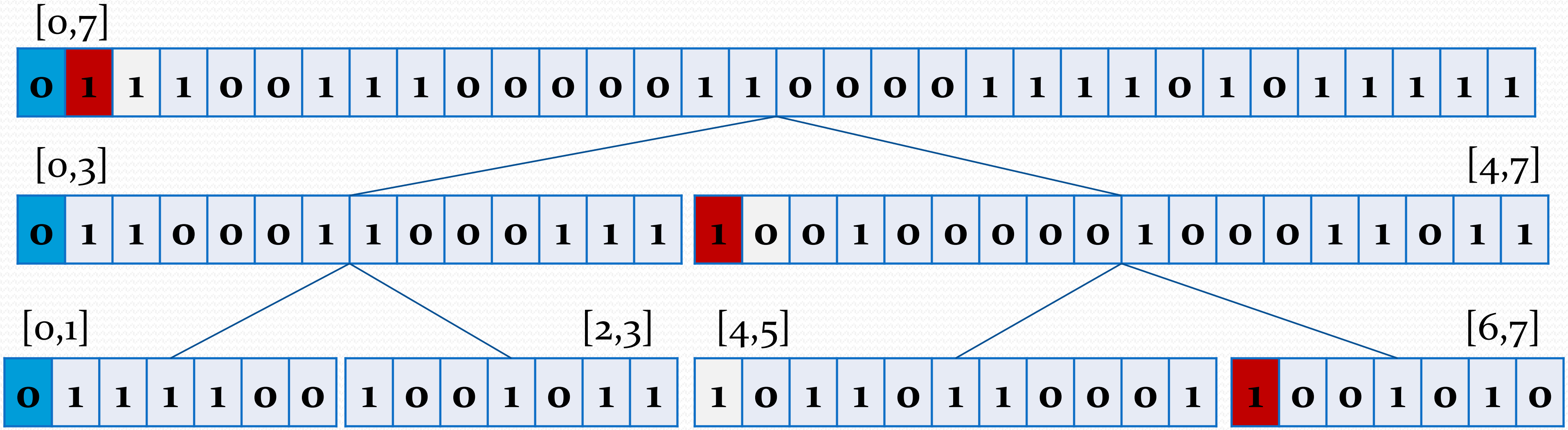
[6,7]

1 0 0 1 0 1 0

000 → 0

How To Do It

- Make a tree: Just store a bit vector at each node



[0,7]
0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]
0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]
1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]
0 1 1 1 1 0 0

[2,3]
1 0 0 1 0 1 1

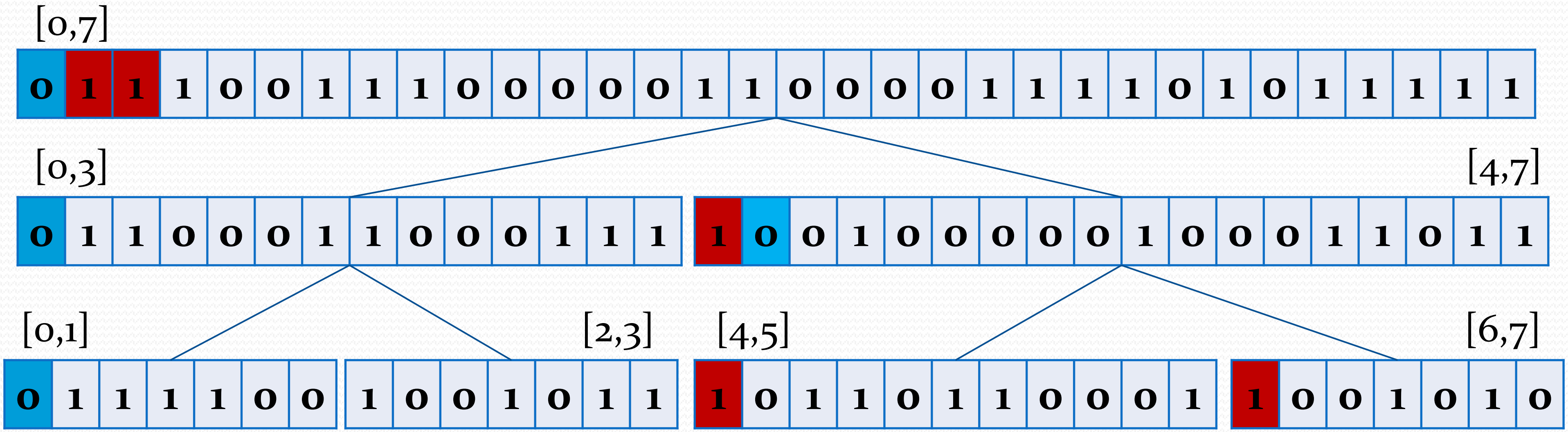
[4,5]
1 0 1 1 0 1 1 0 0 0 1

[6,7]
1 0 0 1 0 1 0

000 → 0
111 → 7

How To Do It

- Make a tree: Just store a bit vector at each node



[0,7]
0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]
0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]
1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]
0 1 1 1 1 0 0

[2,3]
1 0 0 1 0 1 1

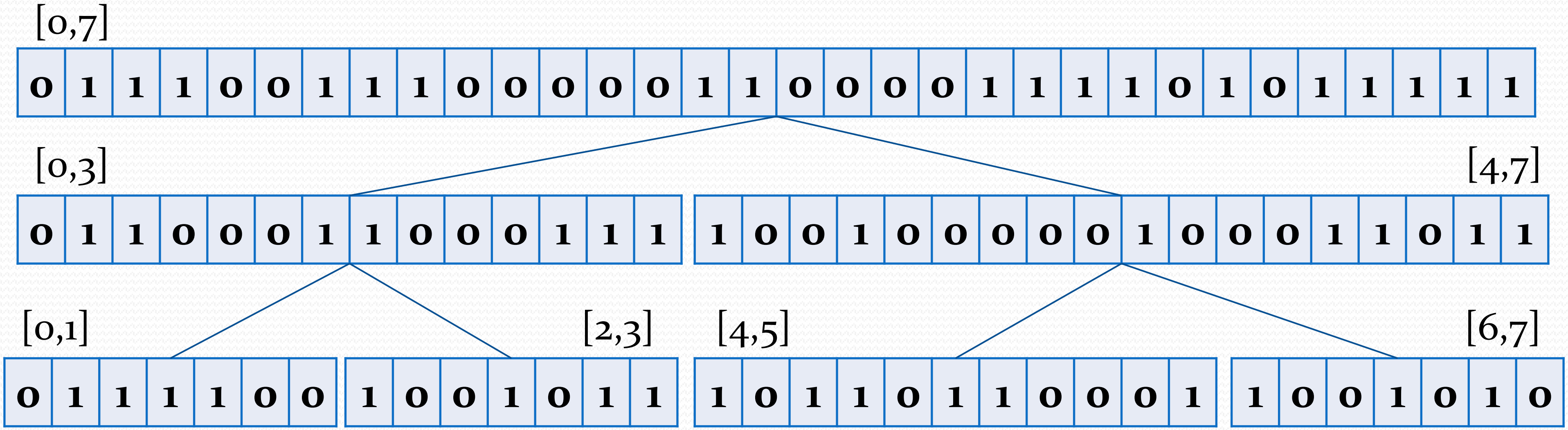
[4,5]
1 0 1 1 0 1 1 0 0 0 1

[6,7]
1 0 0 1 0 1 0

000 → 0
111 → 7
101 → 5
etc.

Space Analysis

- Make a tree: Only need $n \log \sigma + o(n \log \sigma)$ bits



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

[4,5]

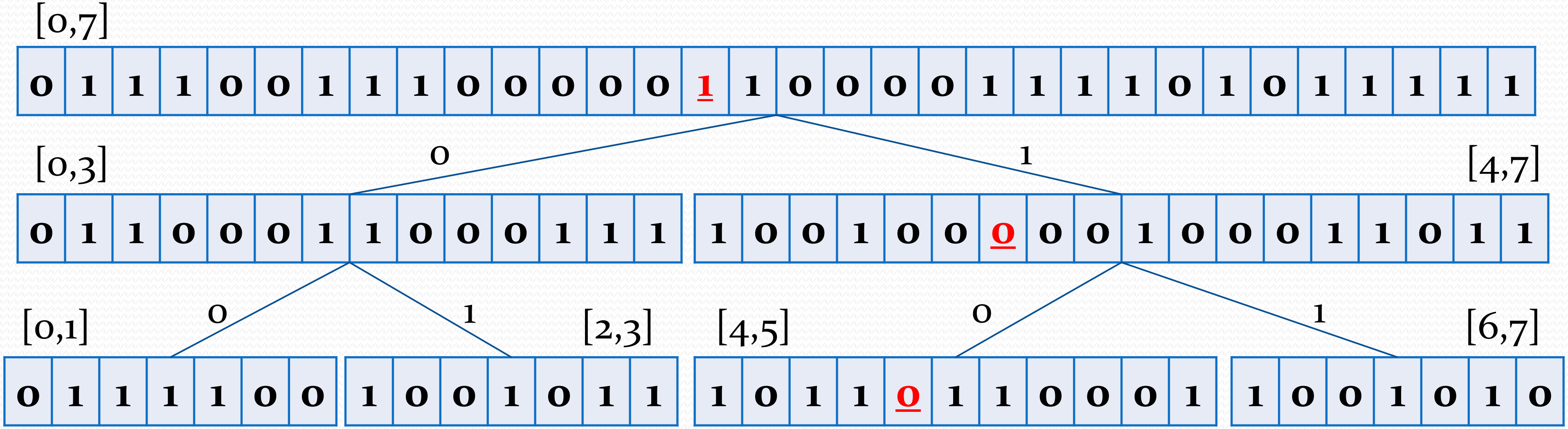
1 0 1 1 0 1 1 0 0 0 1

[6,7]

1 0 0 1 0 1 0

Store each level as a contiguous bit vector in fully indexable dictionary:
$\log \sigma$ levels; each has n bits (plus $o(n)$ redundancy)
Don't need to actually store a "tree"

Basic Operations: Access



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

[4,5]

1 0 1 1 0 1 1 0 0 0 1

[6,7]

1 0 0 1 0 1 0

To perform **Access**(i) in $\Theta(\log \sigma)$ time (in the example **Access**(14)):

1) Access bit i in current node (start at root); call bit value b

2) If not in a leaf:

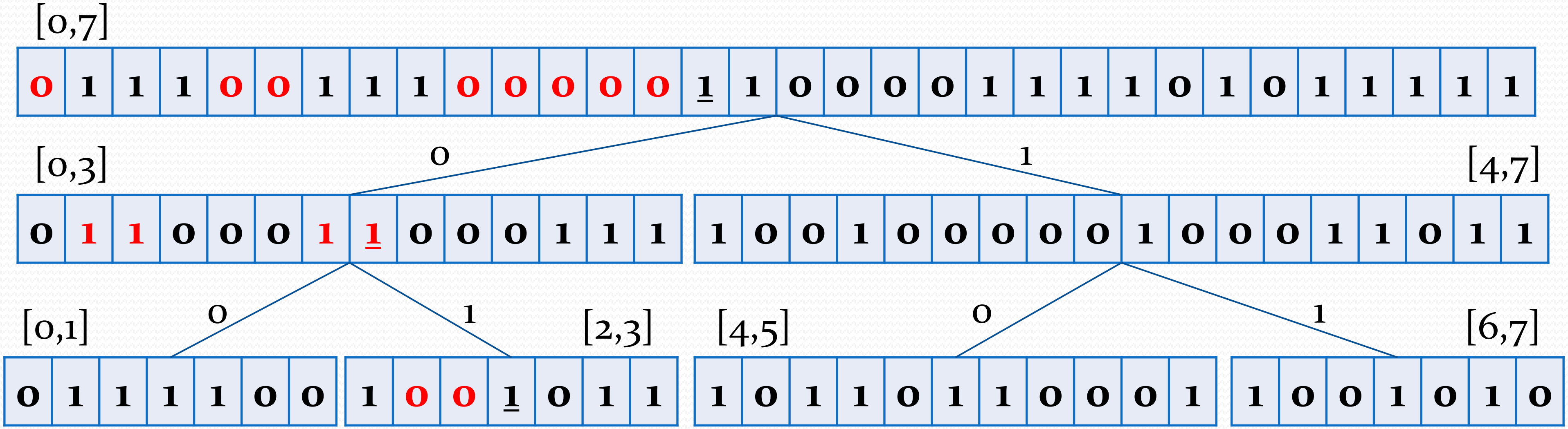
Compute j = **Rank**(i, b) on bit vector in current node

Follow branch b, and recursively **Access**(j) there…

Concatenate all bits b along this path, and return this as the answer: 100 → 4

0 7 5 4 3 2 6 5 5 1 1 1 2 3 4 5 1 0 0 2 5 6 4 4 3 4 3 7 6 5 7 6

Basic Operations: Rank



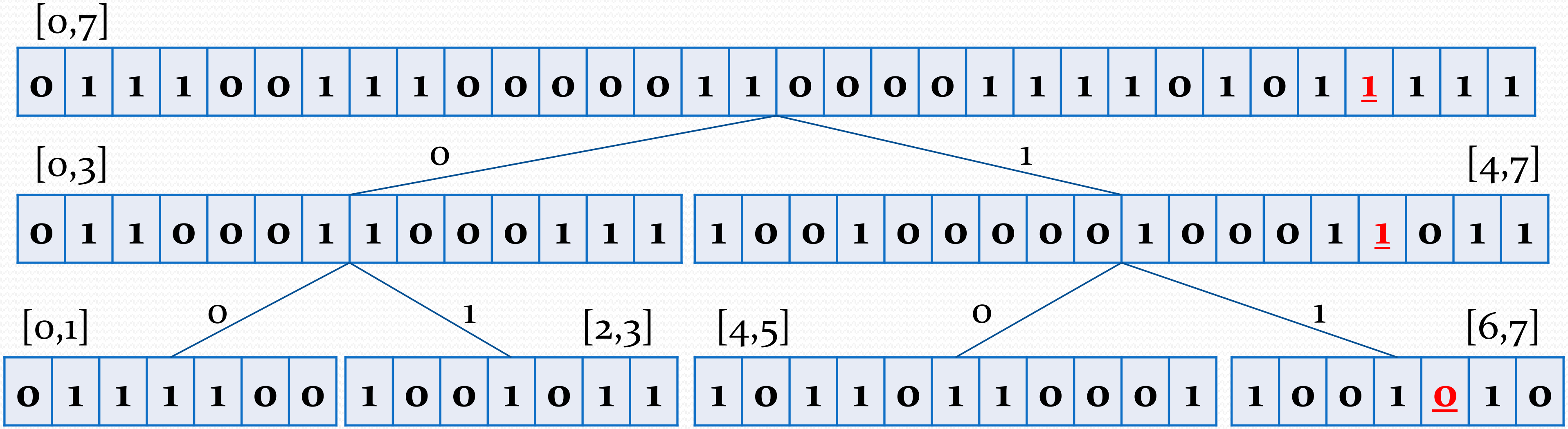
To perform **Rank**(i, α) in $\Theta(\log \sigma)$ time (in the example: **Rank**(14,2)):

1) Compute $j =$ **Rank**(i, b), where b is the *next* most significant bit of α

2) If not in a leaf: branch to node b and recurse setting $i = j$

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6

Basic Operations: Select



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

[4,7]

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

[4,5]

1 0 1 1 0 1 1 0 0 0 1

[6,7]

1 0 0 1 0 1 0

To perform **Select**(i, α) in $\Theta(\log \sigma)$ time *starting at correct leaf* (example: **Select**(3,6)):

1) Compute $j =$ **Select**(i, b), where b is the *next* least significant bit of α
2) If not in the root: move to parent and recurse setting $i = j$

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6

Brief History of the “Wavelet Tree”

- Chazelle (1988): Compact Range Tree
  - His concern was making the space $\Theta(n)$ words
  - Succinct data structures weren’t invented yet…
  - He wanted to solve orthogonal range searching problems
    - We will also focus on these kinds of problems
- Grossi, Gupta and Vitter (2003): Wavelet Tree
  - More or less described the same thing we just covered
  - They were concerned with text indexing problems

These are the same data structure!

(modulo the compressed bit vectors)

Better Space Analysis

- We use fully indexable dictionaries for each level
  - They can use less than n bits… can we do better than $n\log\sigma$?
- *Zeroth Order Empirical Entropy* of an array A:
  - Let n_α be the frequency of symbol $\alpha \in [0, \log\sigma - 1]$
  - Define $H_0(A) = \frac{1}{n}\left(\sum_\alpha n_\alpha \log\frac{n}{n_\alpha}\right)$
    - If all symbols equally likely, then this is just $n\log\sigma$
- Consider a bit vector B, i.e., the case where $\sigma = 2$
  - Using Stirling's Approximation one can prove:

$$\binom{n}{n_1} \leq 2^{nH_0(B)} \text{… so } \log\binom{n}{n_1} \leq nH_0(B)$$

  - What does this mean for the wavelet tree?

Better Space Analysis (2)

- I will write the subscripts in binary here… $n_2 \rightarrow n_{10}$
- Consider array A where $\sigma = [0,3]$ (i.e., wavelet tree with two levels)…
  - How much space to store the root bit vector:
    - Let $m_0 = n_{00} + n_{01}$ and $m_1 = n_{10} + n_{11}$
    - The bit vector is no more than $m_0 \log \frac{n}{m_0} + m_1 \log \frac{n}{m_1}$ bits
  - Children:
    - Bit vectors occupy no more than

$$n_{00} \log \frac{m_0}{n_{00}} + n_{01} \log \frac{m_0}{n_{01}} + n_{10} \log \frac{m_1}{n_{10}} + n_{11} \log \frac{m_1}{n_{11}}$$

- Total Space:
  - $m_0 \log \frac{n}{m_0} + n_{00} \log \frac{m_0}{n_{00}} + n_{01} \log \frac{m_0}{n_{01}} = n_{00} \log \frac{n}{n_{00}} + n_{01} \log \frac{n}{n_{01}}$
  - $m_1 \log \frac{n}{m_1} + n_{10} \log \frac{m_1}{n_{10}} + n_{11} \log \frac{m_1}{n_{11}} = n_{10} \log \frac{n}{n_{10}} + n_{11} \log \frac{n}{n_{11}}$

This is just the entropy of A!

Better Space Analysis (2)

- I will write the subscr ts in binary l .. $n_2 \to n_{10}$
- Co r array A wh = [0,3] let tree with two levels)...
  - How
    - Let m
  - Ch
    - D
    - n_{00} log

The wavelet tree occupies

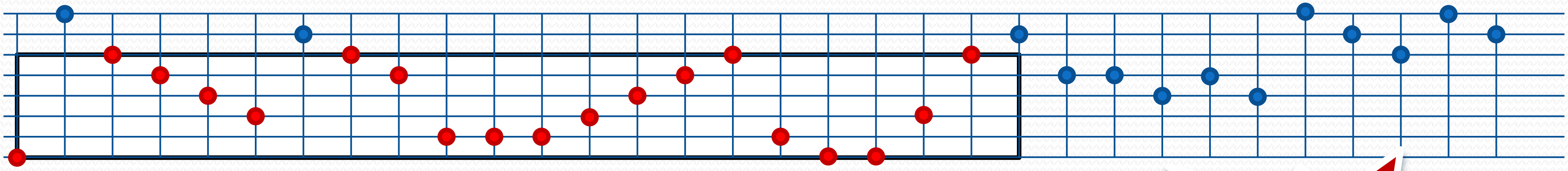
$$nH_0(A) + o\left(\frac{n\log\sigma\log\log n}{\log n}\right) \text{ bits}$$

- Total Space
  - $m_0 \log\frac{n}{m_0} + n_{00}\log\frac{m_0}{n_{00}}$ $\log\frac{m_0}{n_{01}} =$ n_0 $\frac{n}{n_{00}} + n_{01}\log\frac{n}{n_{01}}$
  - $m_1 \log\frac{n}{m_1} + n_{10}\log\frac{m_1}{n_{10}} +$ $_{11}\log\frac{m_1}{n_{11}} = n_{10}\log\frac{n}{n_{10}} + n_{11}\log\frac{n}{n_{11}}$

This is just the entropy of A!

Orthogonal Range Counting

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Two-Sided” Query: [0,20] × [0,5]

19 points

[0,7]

0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

0

1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

[0,1]

0

1

[2,3]

[4,5]

0

1

0	1	1	1	1	0	0

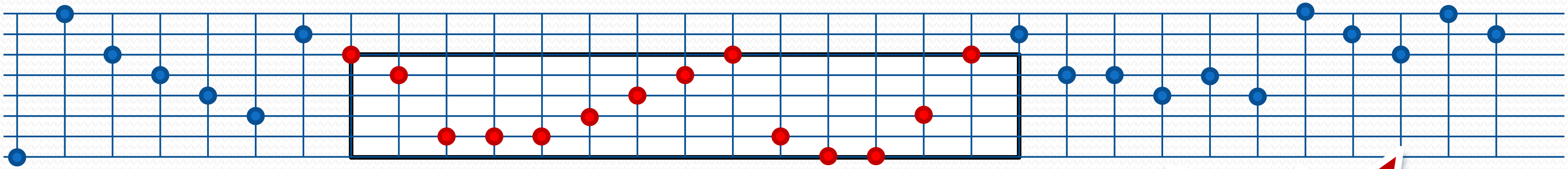
1	0	0	1	0	1	1

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

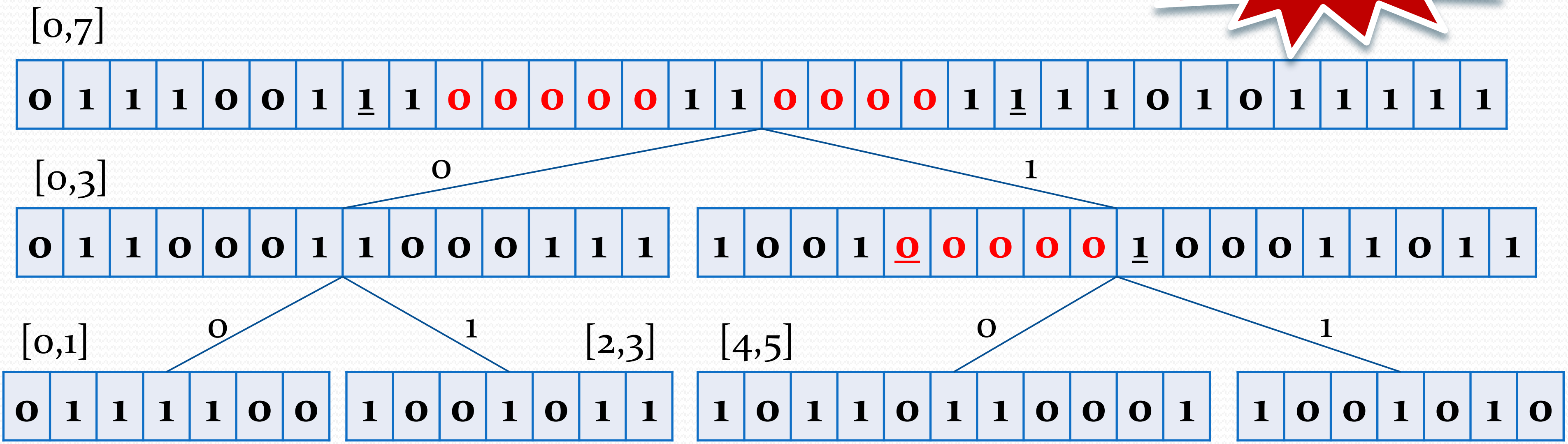
Orthogonal Range Counting

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



"Three-Sided" Query: [7, 20] × [0,5]

14 points



[0,7]

0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

[0,1]

0	1	1	1	1	0	0

[2,3]

1	0	0	1	0	1	1

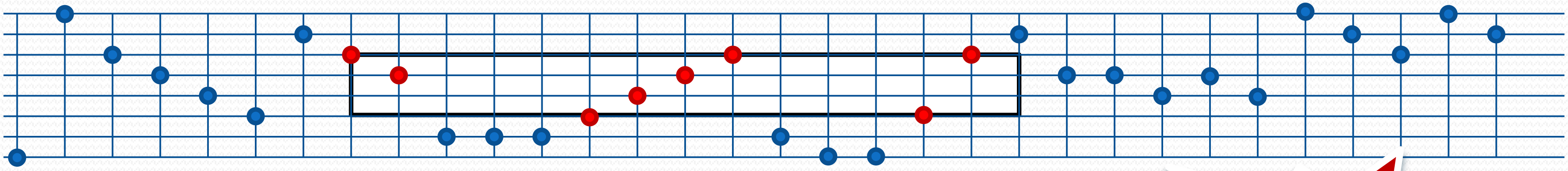
[4,5]

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

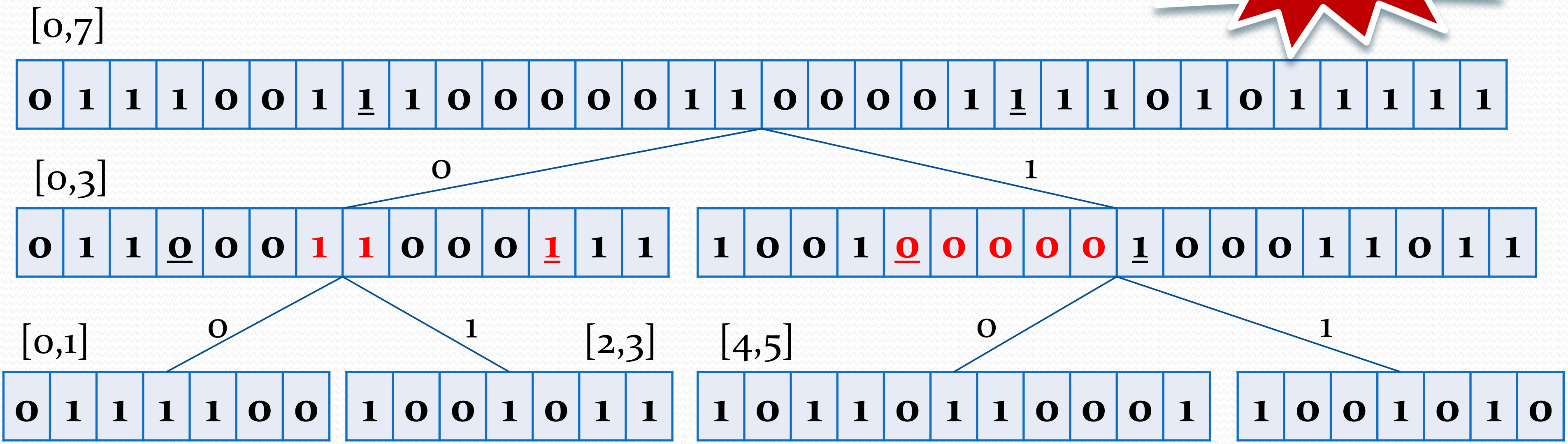
Orthogonal Range Counting

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Four-Sided” Query: [7,20] × [**2**, 5]

8 points



[0,7]

0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

[0,1]

0	1	1	1	1	0	0

[2,3]

1	0	0	1	0	1	1

[4,5]

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

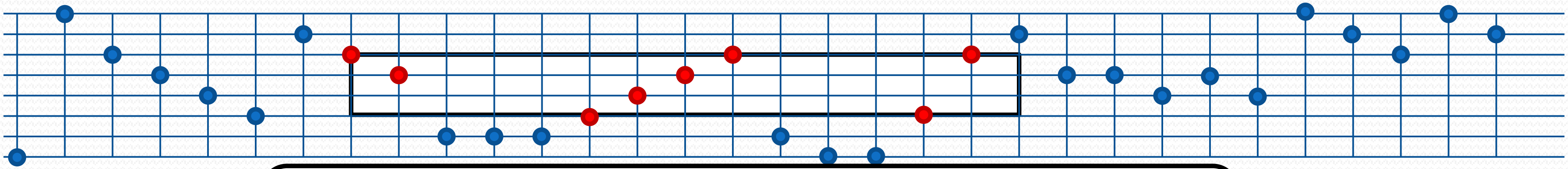
Analysis: Orthogonal Counting

- Two-Sided:
  - Follow one root-to-leaf path
  - Constant time in each node (rank/select)
  - Overall Time: $\Theta(\log \sigma)$
- Three-Sided:
  - Root-to-leaf traversal + cost of two two-sided queries
  - Overall Time $\Theta(\log \sigma)$
- Four-Sided:
  - Root-to-leaf traversal + cost of two three-sided queries
  - Overall Time $\Theta(\log \sigma)$

Orthogonal Range C

Reporting

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



Once we can count the "red guys" we need only select each one and *track* it to its leaf. This yields its *y* value. For the *x* value we can track up to the root.

[0,7]

0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

0

1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

[0,1]

0

1

[2,3]

[4,5]

0

1

0	1	1	1	1	0	0

1	0	0	1	0	1	1

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

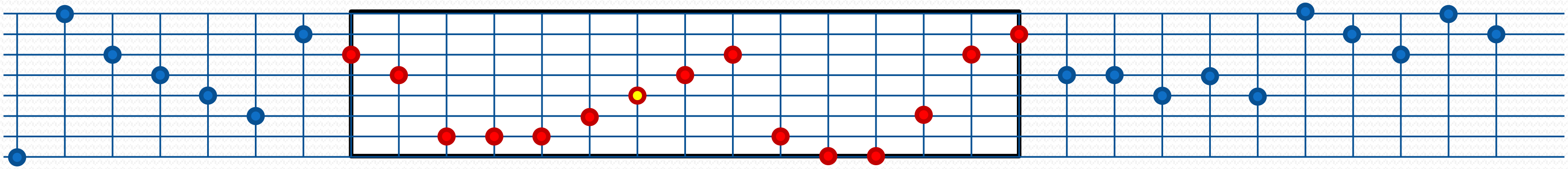
Analysis: Orthogonal Reporting

- Use the counting algorithm to find the "red guys"
  - Find nodes s.t. all subtree elements are in the rectangle
- Track each one to its leaf to determine the *y* value
  - Once we find the *y* value, track to the root for *x* value
- Overall time:
  - If t points are reported, this takes: $\Theta((t+1)\log\sigma)$

Nice additional property: We can report the points sorted by *y* order. Reporting the first point above a "line" is sometimes called a *range successor* or *range next-value* query.

Range Selection (Gagie et al. 2009)

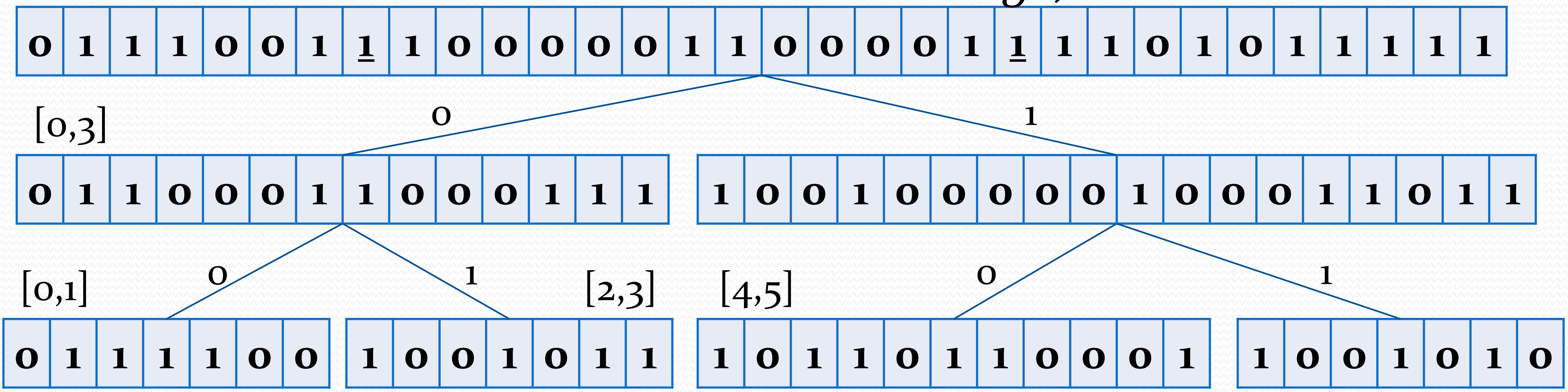
0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



"Two-Sided" Query: [0,20] × [0,5]

Find k-th smallest element: $e.g., k = 9$

[0,7]



0 1 1 1 0 0 1 $\underline{1}$ 1 0 0 0 0 0 1 1 0 0 0 0 1 $\underline{1}$ 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

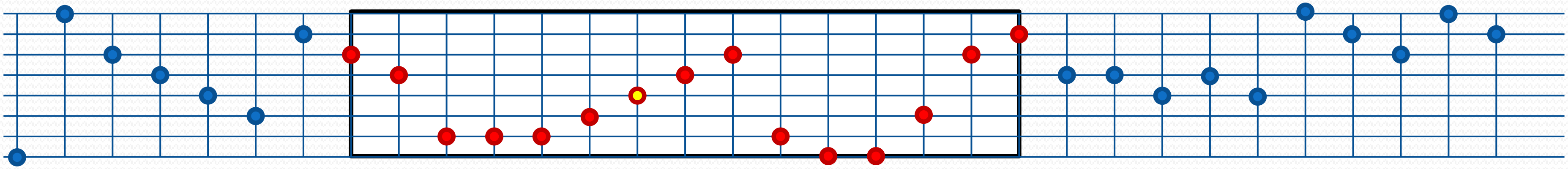
[4,5]

1 0 1 1 0 1 1 0 0 0 1

1 0 0 1 0 1 0

Range Selection (Gagie et al. 2009)

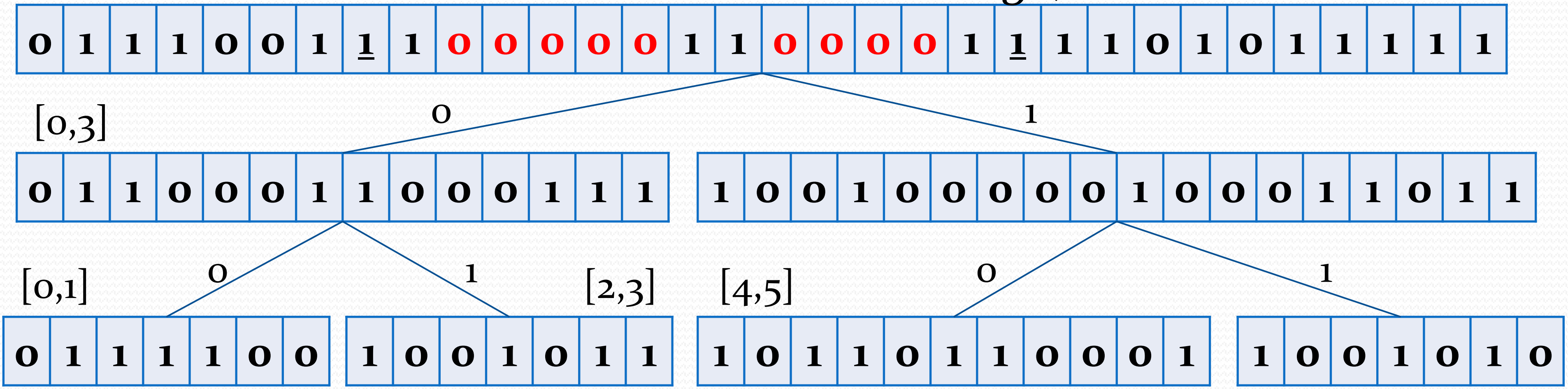
0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Two-Sided” Query: [0,20] × [0,5]

Find k-th smallest element: $e.g., k = 9$

[0,7]



0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

[0,1]

0	1	1	1	1	0	0

[2,3]

1	0	0	1	0	1	1

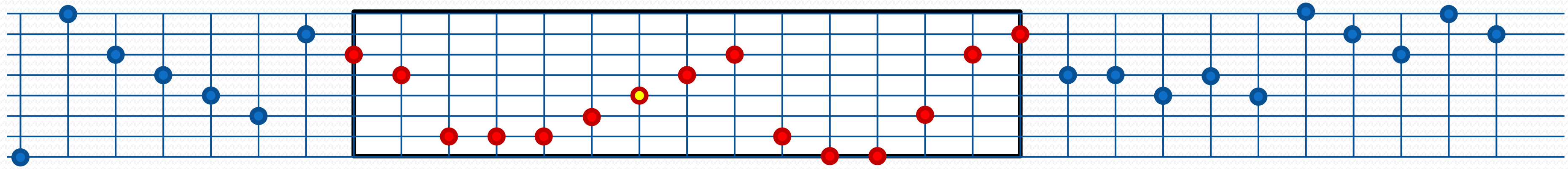
[4,5]

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

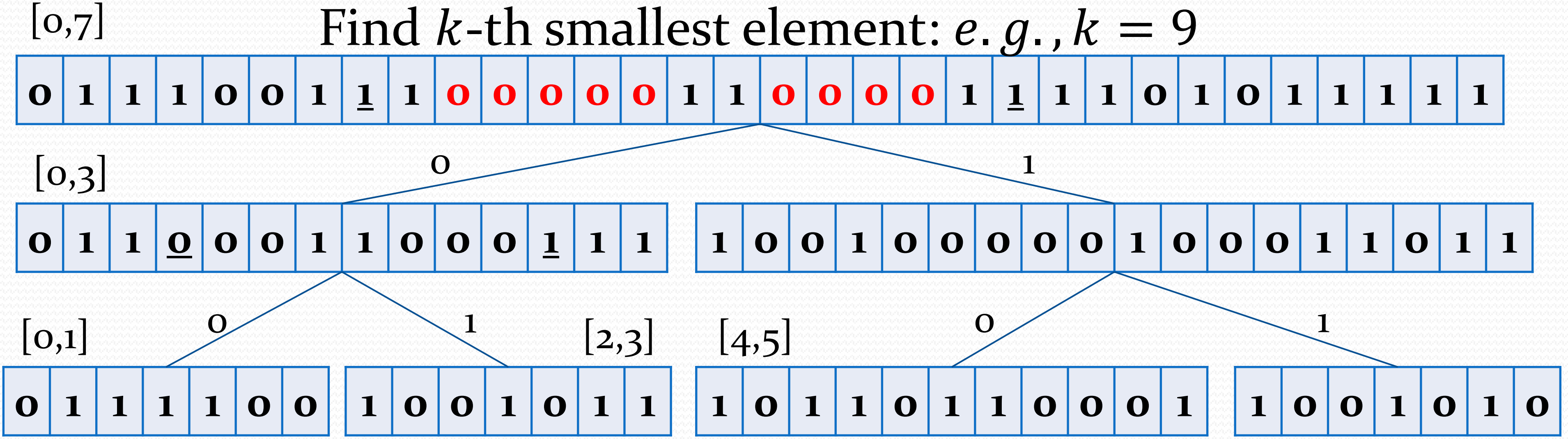
Range Selection (Gagie et al. 2009)

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Two-Sided” Query: [0,20] × [0,5]

Find k-th smallest element: $e.g., k = 9$



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

[0,1]

0 1 1 1 1 0 0

[2,3]

1 0 0 1 0 1 1

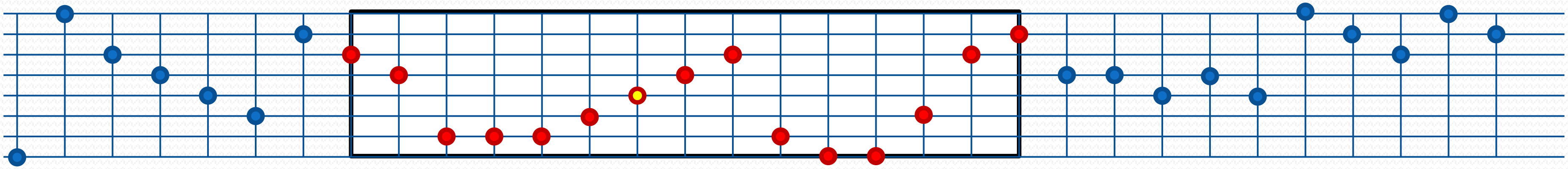
[4,5]

1 0 1 1 0 1 1 0 0 0 1

1 0 0 1 0 1 0

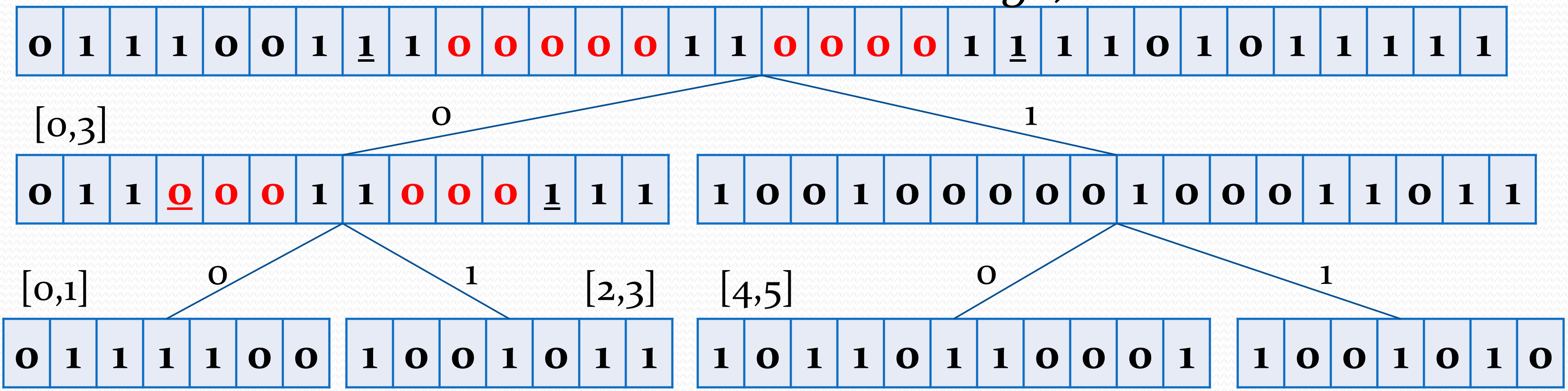
Range Selection (Gagie et al. 2009)

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Two-Sided” Query: [0,20] × [0,5]

Find k-th smallest element: $e.g., k = 9$



[0,7]

0 1 1 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 1 1

0 1

[0,3]

0 1 1 0 0 0 1 1 0 0 0 1 1 1

1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 1

0 1 0 1

[0,1] [2,3] [4,5]

0 1 1 1 1 0 0

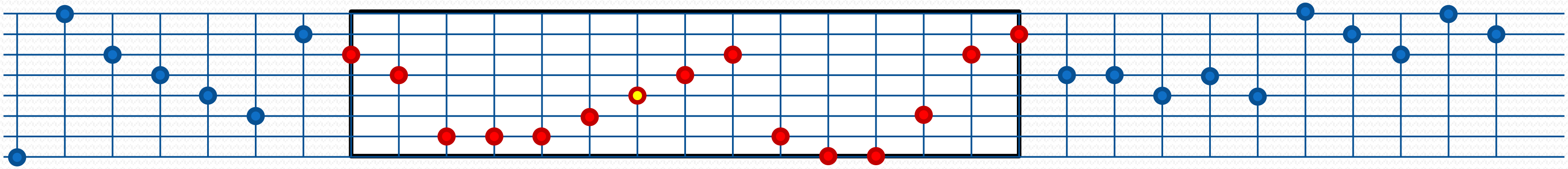
1 0 0 1 0 1 1

1 0 1 1 0 1 1 0 0 0 1

1 0 0 1 0 1 0

Range Selection (Gagie et al. 2009)

0	7	5	4	3	2	6	5	5	1	1	1	2	3	4	5	1	0	0	2	5	6	4	4	3	4	3	7	6	5	7	6



“Two-Sided” Query: [0,20] × [0,5]

Find k-th smallest element: $e.g., k = 9$

[0,7]

0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	1

0   1

[0,3]

0	1	1	0	0	0	1	1	0	0	0	1	1	1

1	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	1

0   1   0   1

[0,1]

0	1	1	1	1	0	0

[2,3]

1	0	0	1	0	1	1

[4,5]

1	0	1	1	0	1	1	0	0	0	1

1	0	0	1	0	1	0

Report Distinct Symbols (Also Gagie et al.)

- Also known as *coloured range reporting*
- Once we can do selection this is a piece of cake:
  - Select for $k = 1$ and report that y value: y_1
  - Count the number n_1 of elements in $[x_1, x_2] \times [0, y_1]$
  - Select $k = n_1 + 1$ and report the y value y_2
  - Count the number n_2 of elements in $[x_1, x_2] \times [0, y_2]$
- Returns all distinct symbols in $\Theta\big((t+1)\log\sigma\big)$ time

Some Improvements

- The wavelet tree is not the end of the story:

Ref.	Access	Rank	Select
(Golynski et al. 2008)	$\Theta\left(\frac{\log\sigma}{\log\log n}\right)$	$\Theta\left(\frac{\log\sigma}{\log\log n}\right)$	$\Theta\left(\frac{\log\sigma}{\log\log n}\right)$
(Golynski et al. 2006), (Barbay et al. 2012)	$\Theta(\log\log\sigma)$	$\Theta(\log\log\sigma)$	$\Theta(1)$
(Golynski et al. 2006), (Barbay et al. 2012)	$\Theta(1)$	$\Theta(\log\log\sigma)$	$\Theta(\log\log\sigma)$
(Belazzougui-Navarro, 2012)	$\Theta(1)$	$\Theta\left(\log\frac{\log\sigma}{\log w}\right)$	$\omega(1)$

Lecture #8: Announcements & Topics

- Exam:
  - July **25**th 10:30-13:30 room 24 (exam)
  - August 25th 12:15-15:00 room 21 (re-exam)
- Assignment #4 Posted
  - Submit Q2 in a text file separate lines
- (Succinct) Dynamic Data Structures

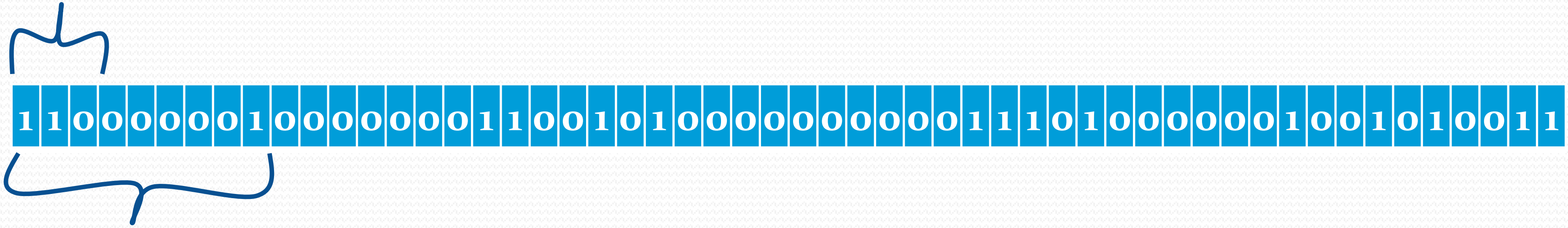
Dynamic Bit Vector

- Let’s consider the following dynamic problem:
  - Support these operations on a bit vector of u bits:
    - Access(i): return the bit at index i
    - Rank(i): return number of 1 bits up to index i
    - Select(i): return the index of the i-th one
    - Flip(i): ***Flip the bit at index i***

  - How can we efficiently support all of these operations?
    - Let’s consider Jacobson’s original solution…

Jacobson’s Solution (Revisited)

Cut *blocks* into *subblocks* of size $\frac{\log u}{2}$ bits

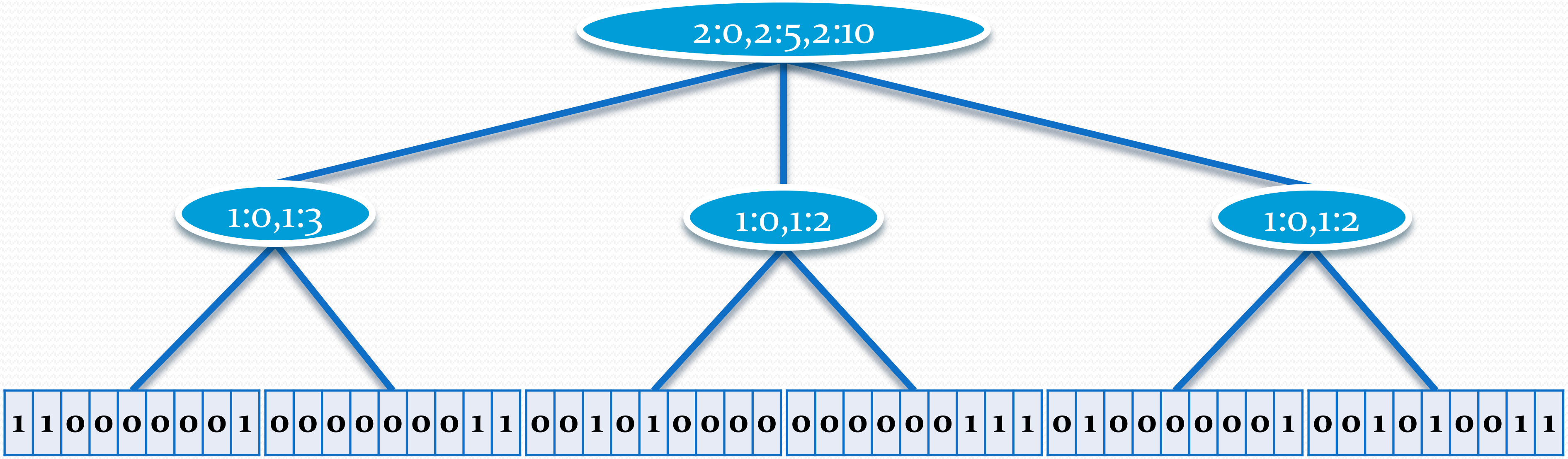


1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0 1 1

Cut into *blocks* of size $\log^2 u$ bits

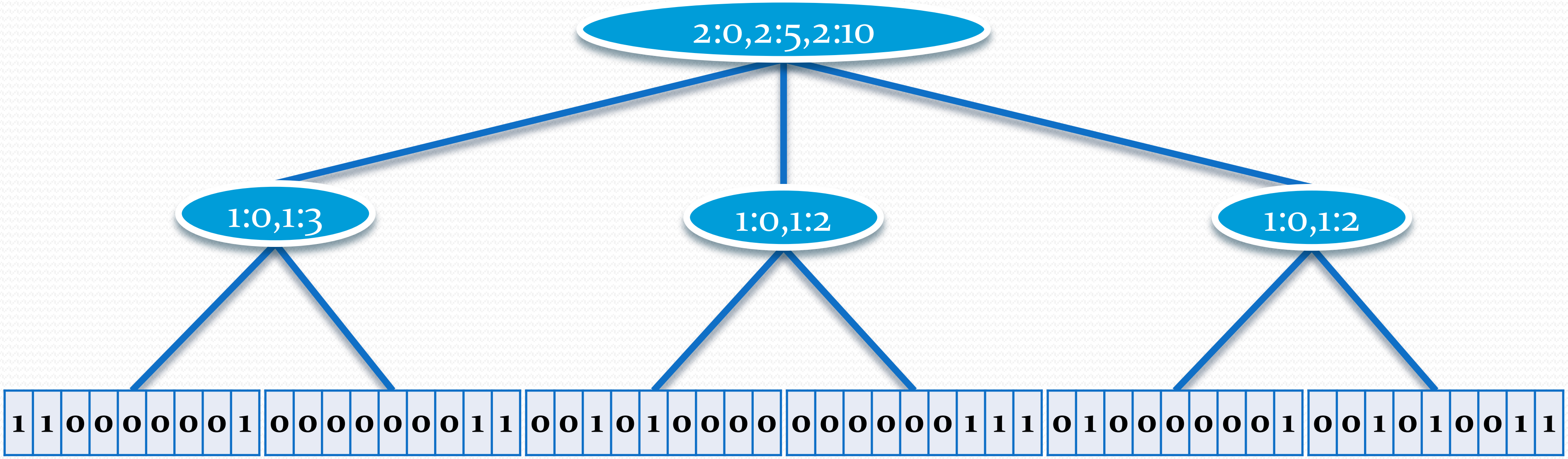
The “Natural” Idea

- Computer Scientists like Trees:
  - *Leaf blocks* of $\log^2 u$ *consecutive* bits
  - Build a tree w/ constant fan out over the leaves
    - *Each node stores, for each child:*
      - *The number of leaves in the subtree*
      - *The number of ones in all subtrees to the left*



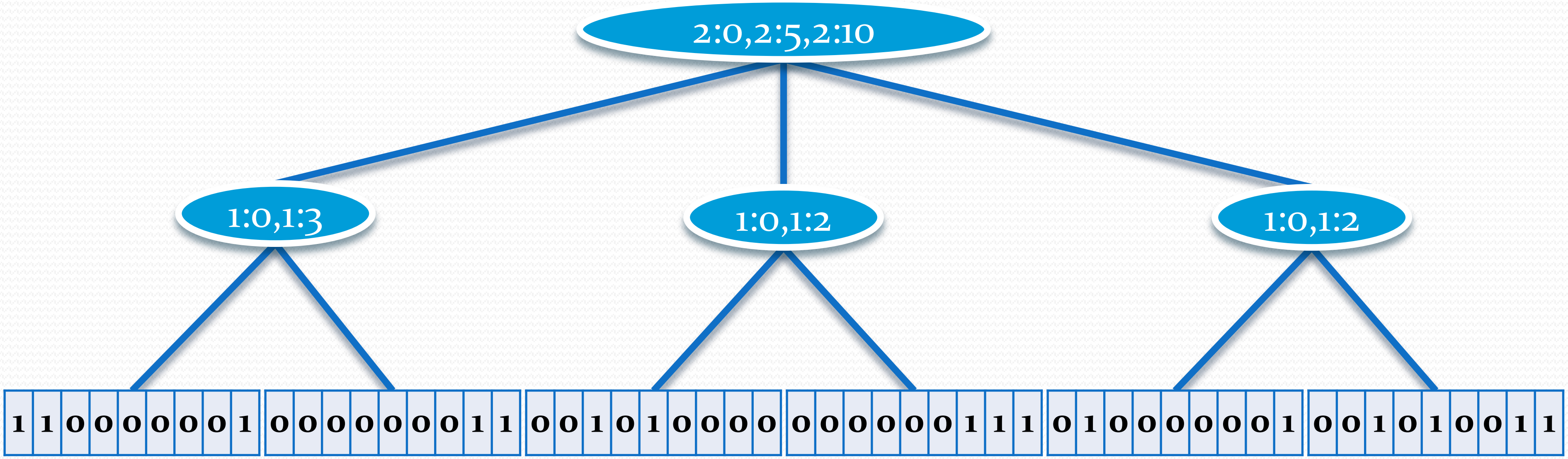
The “Natural” Idea

- Supporting Rank(i) is not so hard:
  - Select branch containing leaf $\left\lfloor \frac{i}{\log^2 u} \right\rfloor$
  - Recurse to child, keeping total of num. of ones to the left
  - At the leaf, use table to compute num. of ones up to pos. i



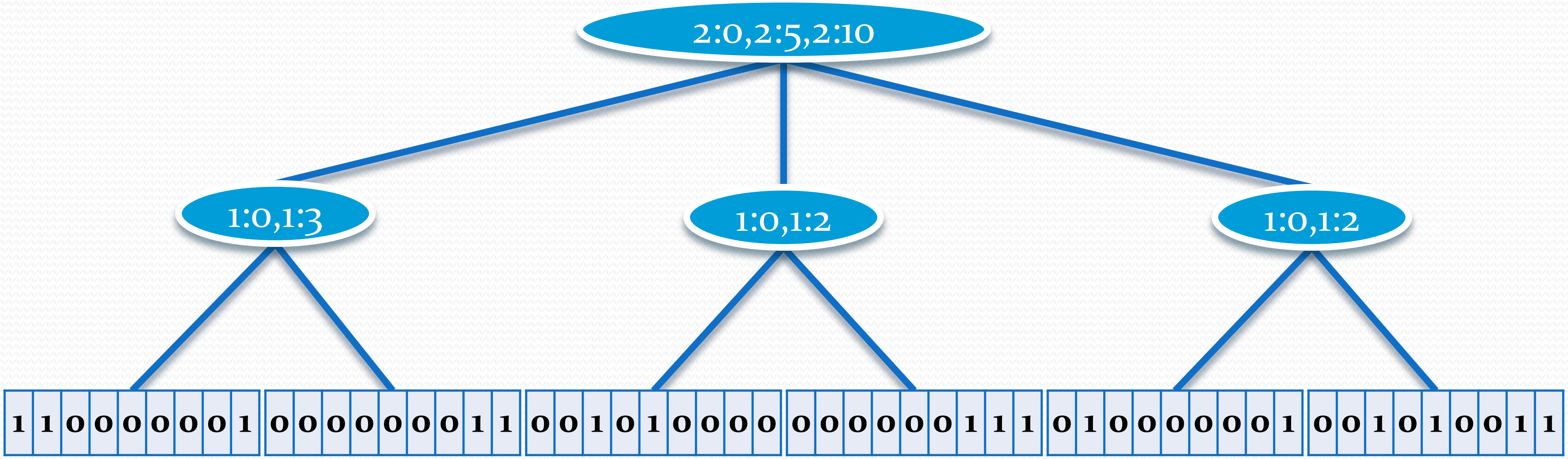
The "Natural" Idea

- Supporting Select(i) is also not so hard:
  - Select branch where i-th one resides and recurse
  - At the leaf, read $\frac{\log u}{2}$ bits at a time to find i-th one



The “Natural” Idea

- What about Flip(i)?
  - Move from root to leaf, adjusting counts in each node
  - Fan out is constant, so we spend $\Theta(1)$ time in each node



2:0,2:5,2:10

1:0,1:3

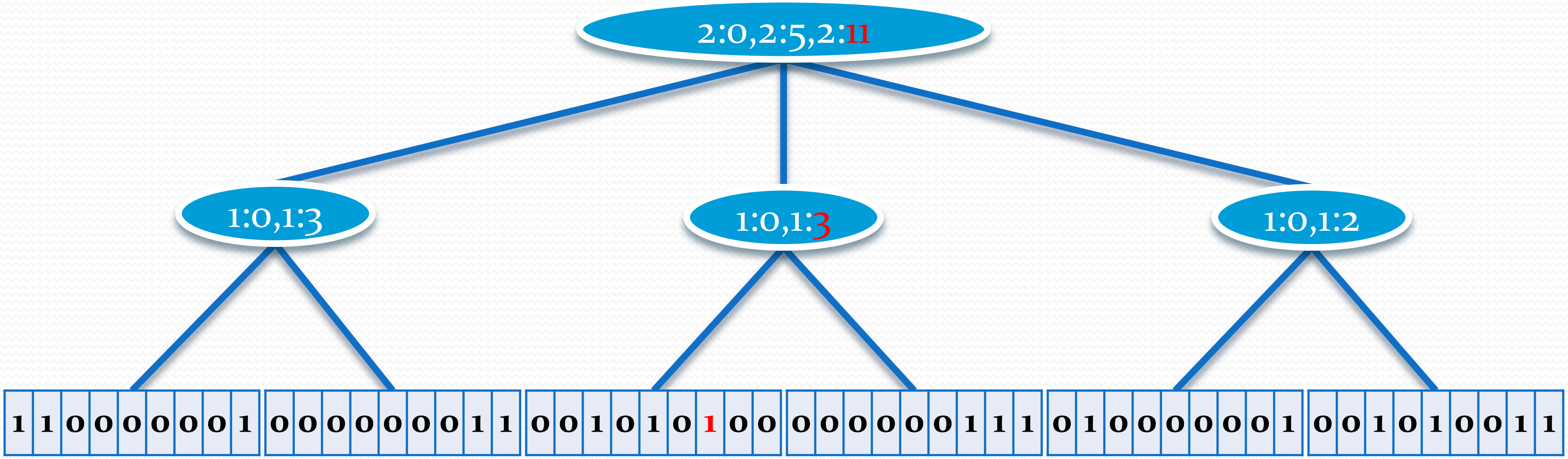
1:0,1:2

1:0,1:2

1 1 0 0 0 0 0 0 1 | 0 0 0 0 0 0 0 1 1 | 0 0 1 0 1 0 0 0 0 | 0 0 0 0 0 0 1 1 1 | 0 1 0 0 0 0 0 0 1 | 0 0 1 0 1 0 0 1 1

The "Natural" Idea

- What about Flip(i)?
  - Move from root to leaf, adjusting counts in each node
  - Fan out is constant, so we spend $\Theta(1)$ time in each node



Good, but not so interesting

- This takes $u + o(u)$ bits… not $H_0(B) + o(u)$
  - We will come back to this issue later…
- We don't *really* want **Flip**(i)
  - We want to be able to **Insert**$(i, \{0,1\})$ or **Delete**(i)
    - "Yeah, yeah, this is not a problem… just resize the leaves!"
      - When a leaf gets too big, split it in two, and rebalance the tree
      - If a leaf gets too small, merge it with some siblings
      - Adds $\Theta(\log u)$ overhead since we copy $\log n$ bits at time

Not so fast! You can't just "*resize the leaves*". We haven't even talked about what the model is for allocating and deallocating memory!

Okay then. What is the Model?

- Standard *Memory Manager* Model (Raman and Rao, 2003):
  - **Allocate**(k):
    - Returns a pointer to a block of 2^k consecutive memory locations ($w2^k$ bits), all initialized to 0, in $\Theta(2^k)$ time. This increases the space usage of the algorithm by $w2^k$ bits.
    - You do not get to have blocks of arbitrary numbers of bits!
  - **Free**(p):
    - Marks the specified block as deleted, and reduces the space usage of the algorithm by the w × "the size of the block"
  - Model <u>**DOES NOT**</u> take fragmentation into account
    - Will come back to this later…

Digression: Dynamic Arrays

- We don't yet know how to do the following *succinctly*:
  - Given an array of maximum length n support:
    - Locate(i): return a pointer to location i
    - Grow(): Increase the size of the array by 1
    - Shrink(): Decrease the size of the array by 1

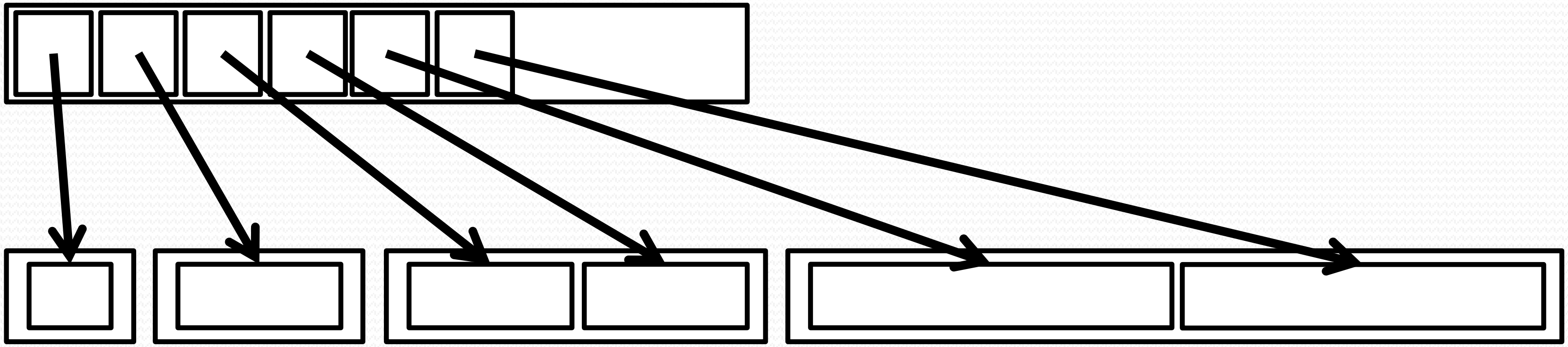
- Idea #1: Standard Doubling Trick: (Double array size when full…)
  - $\Theta(1)$ time for Locate(i)
  - $\Theta(1)$ time for Grow/Shrink (in the *amortized* sense)
  - However, space is quite large:
    - If we halve when reduced to $1/c$ full then space is $\left(c + \frac{c}{2}\right) n$
      - For example: if we halve array when $1/3$ full then space is $4.5n$!

Idea #2: Like-A-Rotated-List

- Recall the rotated list scheme:
  - Keep $\sim\sqrt{2n}$ lists, where list i has length i
- We should try to grow like this instead of doubling…
  - Overall waste would be $\Theta(\sqrt{n})$ which is much better
  - However, we can't allocate non-powers-of-two
  - Furthermore, Locate(i) is a pain:
    - i-th element in list $k = \left\lceil \frac{\sqrt{1+8i}-1}{2} \right\rceil$ in position $i - k(k-1)/2$
      - List number is not constant time to compute due to sqrt!
      - It *is* possible to get around this but we will do something else…

Idea #3: (Brodnik et al. 1999)

- Have conceptual blocks of size 2^i
- Split block i into $2^{\left\lfloor \frac{i}{2} \right\rfloor}$ subblocks of size $2^{\left\lceil \frac{i}{2} \right\rceil}$
  - We need an *index* storing pointers to subblocks



Idea #3: (Brodnik et al. 1999)

- How to grow?

```
If the last subblock $s - 1$ is full:
    If the last block $b - 1$ is full:
        Increment $b$
        If $b$ is odd
            This double the number of subblocks in a block
        Otherwise
            This double the number of elements in a subblock
    If there are no empty subblocks*
        If the index is full, double its size
        Allocate the new subblock
    Increment $s$, $n$, and number of elements in block $s - 1$
```

*When a Shrink() occurs, don’t immediately deallocate…

Idea #3: (Brodnik et al. 1999)

- How much extra space?
    - Number of subblocks is $\Theta(\sqrt{n})$
        - Therefore index has $\Theta(\sqrt{n})$ pointers
    - Last empty subblock has size $\Theta(\sqrt{n})$
    - Therefore overall waste is $\Theta(\sqrt{n})$

Idea #3: (Brodnik et al. 1999)

- How to Locate(i):
  - Let i_2 be the bits of $i + 1$ with leading zeros removed*
  - Let $k = |i_2| - 1$
  - b be the high $\left\lfloor \frac{k}{2} \right\rfloor$ bits of i_2 after the 1
  - c be the low $\left\lceil \frac{k}{2} \right\rceil$ bits
  - Let $p = 2^k - 1$ (num. subblocks in blocks prior to block k)
  - Return element c in subblock $p + b$

*Can find first one in constant time with basic oper.
or, we can just build a lookup table… $\Theta(\sqrt{n})$ space

Lower Bound

- $\Omega(\sqrt{n})$ extra storage is necessary in the worst case for resizable arrays

1. Consider n insertions followed by n deletions
2. Let $f(n)$ be the size of the largest memory block
3. Let $g(n)$ be the number of memory blocks
4. Thus, $f(n)g(n) \geq n$
5. Claim 1: $g(n)$ space for the memory block *headers*
6. Claim 2: $f(n)$ waste after largest mem. block allocated
7. Thus, $\max\{g(n), f(n)\}$ space wasted at some point

Dynamic Bit Vector (Revisited)

- Suppose we wish to support the following operations:
  - Access(i): Return the bit at index i
  - Rank(i): Return number of 1 bits up to index i
  - Select(i): Return the index of the i-th one
  - **Insert**($i, \{0,1\}$): Insert the specified bit at index i
  - **Delete**(i): Delete the bit at index i

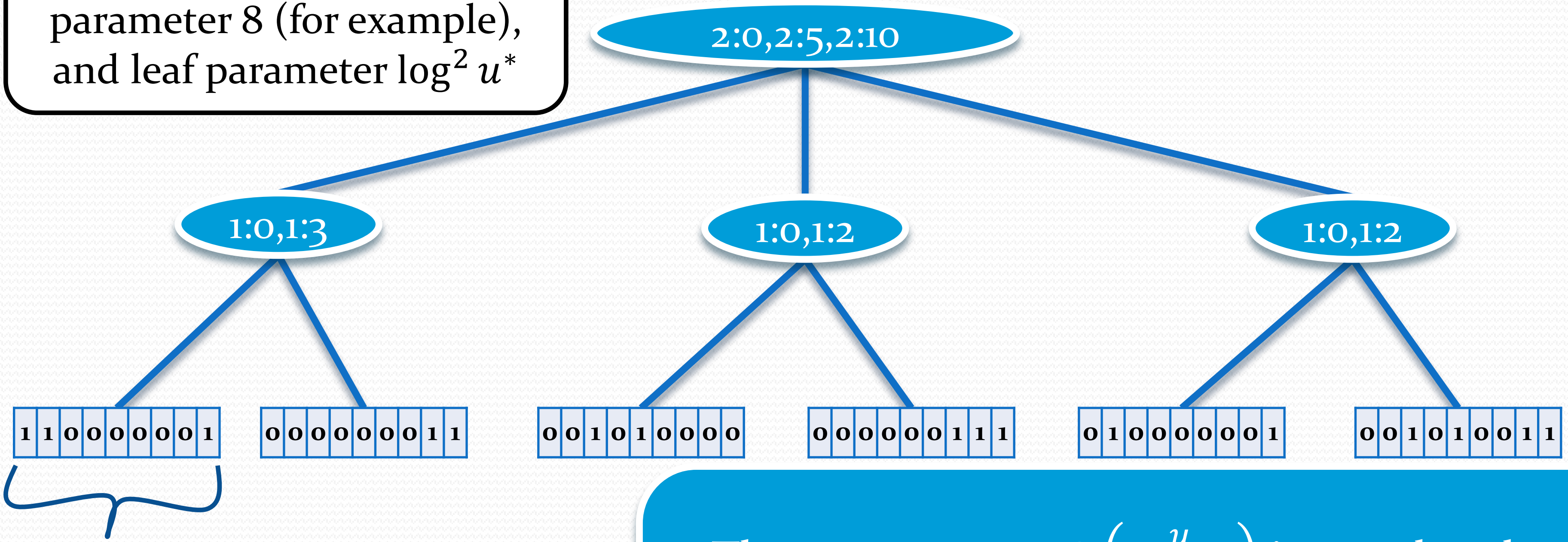
- To simplify, we will assume
  - The bit vector has size $u = \Theta(u^*)$ where u^* is an upper bound
  - The word size $w = \Theta(\log u) = \Theta(\log u^*)$
    - So, we assume u changes, but not by too much…

Black Box: Weight Balanced B-Tree

- *(Arge and Vitter,* 2003*)*: T is a ***weight-balanced B-tree*** with *branching parameter a* and *leaf parameter* k, $a > 4$ and $k > 0$, if the following conditions hold:
  - All leaves of *T* are on the same level and have weight between k and 2k –1.
  - Except for the root, an internal node on level l has weight larger than $a^l k/2$
  - An internal node on level l has weight less than $2a^l k$
  - The root has more than one child.
- Some Useful Properties:
  - Height is $O(\log_a(n/k))$ if tree has weight n
  - Number of splits/fusing operations is $O(\log_a(|T|/k))$
  - All internal nodes have between $a/4$ and $4a$ children
  - Root has between 2 and $4a$ children

Approach #1: Resizable Arrays

WBB-Tree, branching parameter 8 (for example), and leaf parameter $\log^2 u^*$

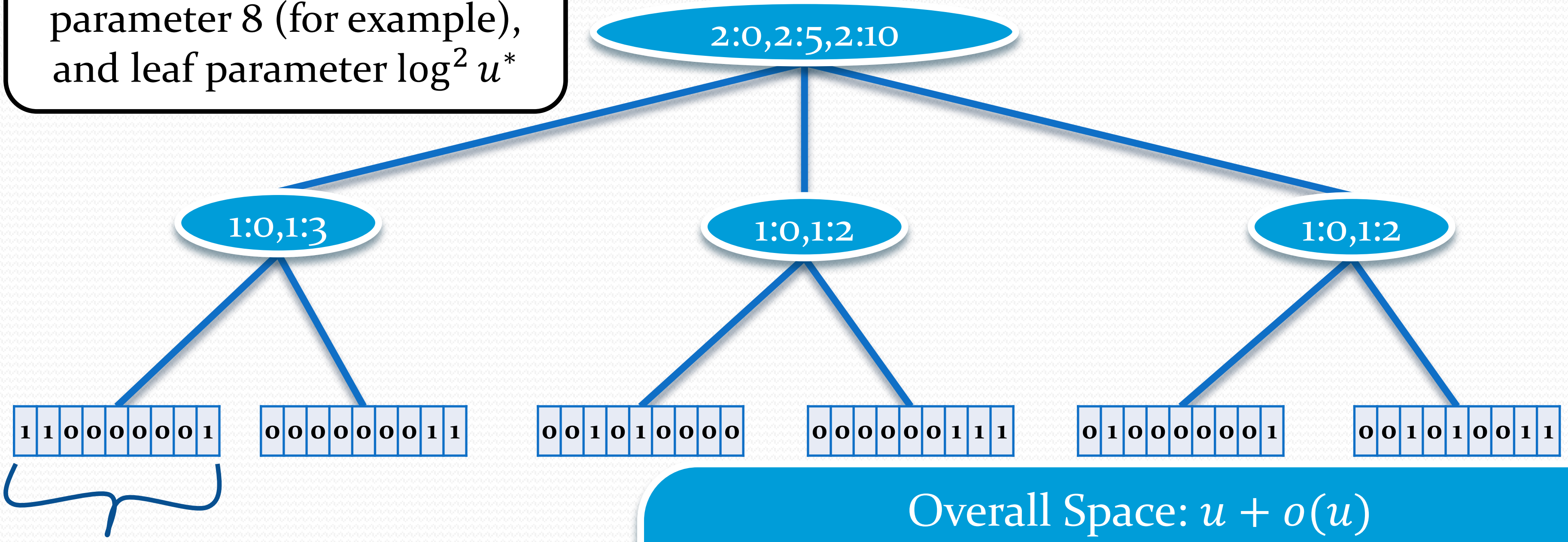


Resizable array of size between $\log^2 u^*$ and $2\log^2 u^* - 1$ BITS
Extra bits per array: $\Theta(\log^{1.5} u^*)$

There are at most $\Theta\left(\frac{u}{\log^2 u^*}\right)$ internal nodes. Also, the extra space used by the resizable arrays is at most $\Theta\left(\frac{u}{\sqrt{\log u^*}}\right)$

Approach #1: Resizable Arrays

WBB-Tree, branching parameter 8 (for example), and leaf parameter $\log^2 u^*$



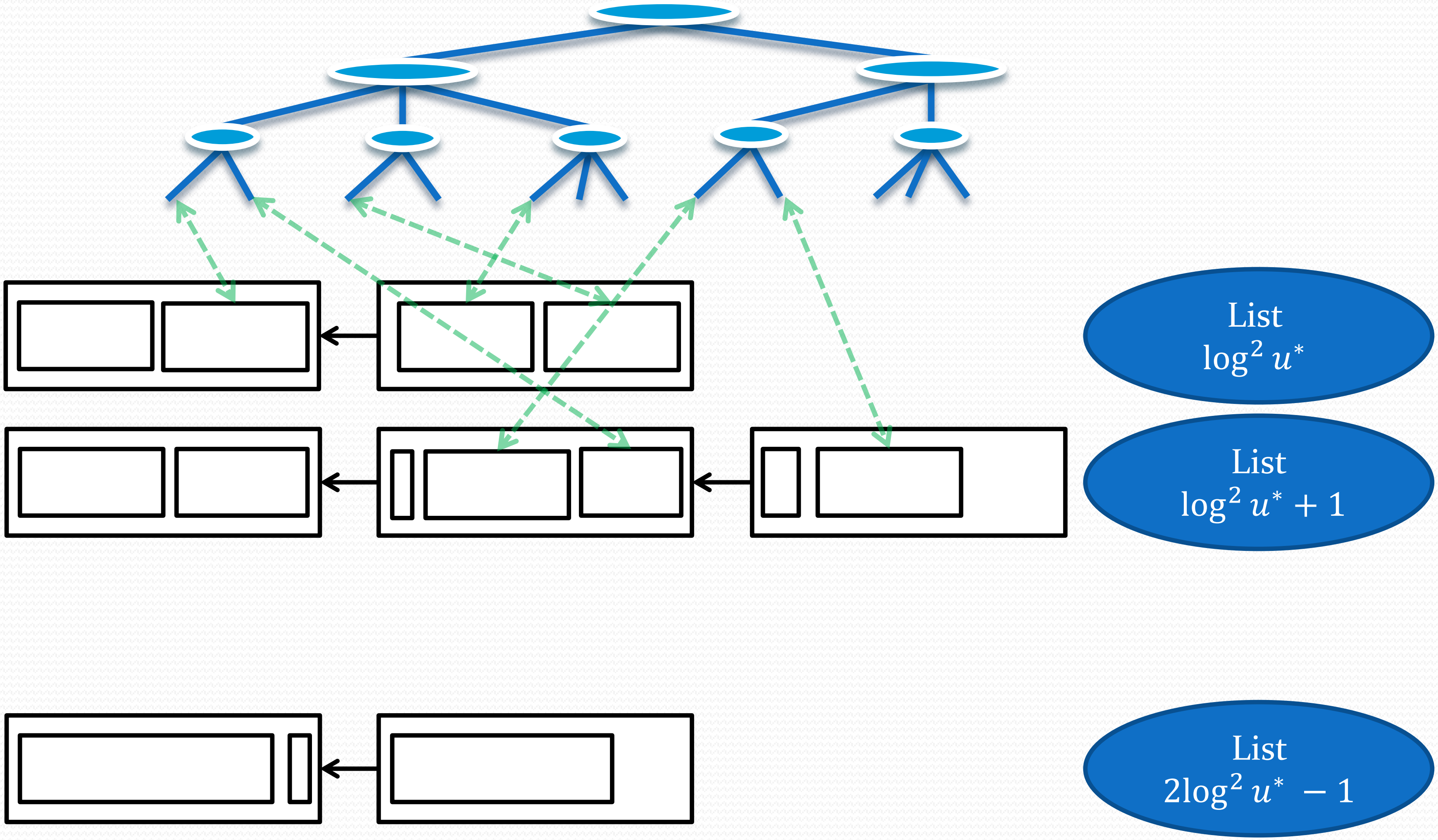
Resizable array of size between $\log^2 u^*$ and $2\log^2 u^* - 1$ BITS
Extra bits per array: $\Theta(\log^{1.5} u^*)$

Overall Space: $u + o(u)$
Updates of leaves: $\Theta(\log u^*)$ *worst case time*
Updating tree: $\Theta(\log u)$ *worst case time*
We can to deal with u changing and not knowing u^*
Using more complicated bit tricks can improve all operations to $\Theta\left(\frac{\log u}{\log\log u}\right)$ in the worst case, which is optimal.

Approach #2: List Based Mem. Manager

- Remember the implicit dictionary (discussed way back when)
  - In the implicit dictionary we kept lists for maniples:
    - List i contained all maniples of i consecutive elements
- Let's apply this approach to the leaves of our WBB-tree
  - As before, each list consists of *nodes*
    - A node stores an array of $2\log^2 u^*$ bits
    - A linked list of pointers back to the tree
  - List i will store all leaves of i bits
    - Always allocate new nodes at the head of a list
    - Fill gaps by swapping with first logical block in head

Approach #2: List Based Mem. Manager



List
$\log^2 u^*$

List
$\log^2 u^* + 1$

List
$2\log^2 u^* - 1$

Approach #2: List Based Mem. Manager

- How much space is wasted?
  - At most one node per list: $\Theta(\log^4 u^*)$ bits…
    - For example: if $u^* = 2^{32}$ bits then waste is 2^{20} bits
  - WBB-tree still takes $\Theta\left(\frac{u}{\log u^*}\right)$ bits
- Better than the other approach for a few reasons:
  - Less space wasted
  - Compression becomes rather trivial
    - Just encode/decode each block on the fly to get $H_0(B) + o(u^*)$ bits
    - Have lists of size $[1, 2\log^2 u^*]$
  - <u>All nodes are the same size</u>
    - We can consider fragmentation in terms of u^*: the max value of u

Approach #2: List Based Mem. Manager

- We allocate:
  - Nodes in our list based memory manager
    - One node per leaf in the WBB-tree
  - Linked list nodes for the back pointers
    - Also one per leaf
  - WBB-tree nodes (we have bounds on how big these are)
    - Again, there are at most Θ(# of leaves) of these
- Suppose we maintain three separate heaps
  - When we allocate one of these types of nodes we use it
  - Instead of freeing it, we put it in the heap of its type
    - These heap will have size $\Theta\left(\frac{u^*}{\log u^*}\right)$… a high watermark bound

Conclusion

- We can apply what we learned about implicit data structures to the word-RAM model… techniques carry over even though the model is very different
- We have sketched how to *dynamize* the succinct data structures presented so far (numerous details omitted)
- Main issues are memory management, dealing with a changing value of u.
- Once we have a dynamic bit vector, we easily get dynamic trees, dynamic wavelet trees, etc.