Oblivious Computation

Part III - OptORAMa

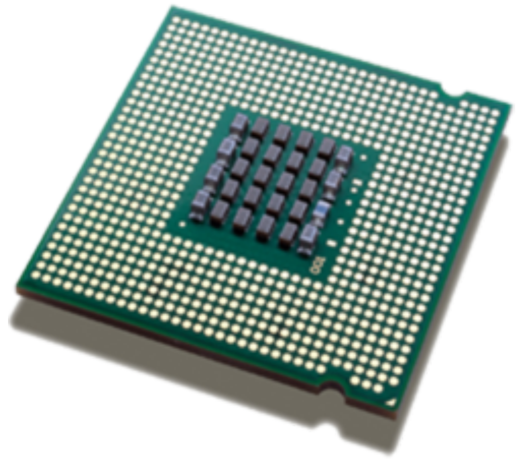
Gilad Asharov

Bar-Ilan University

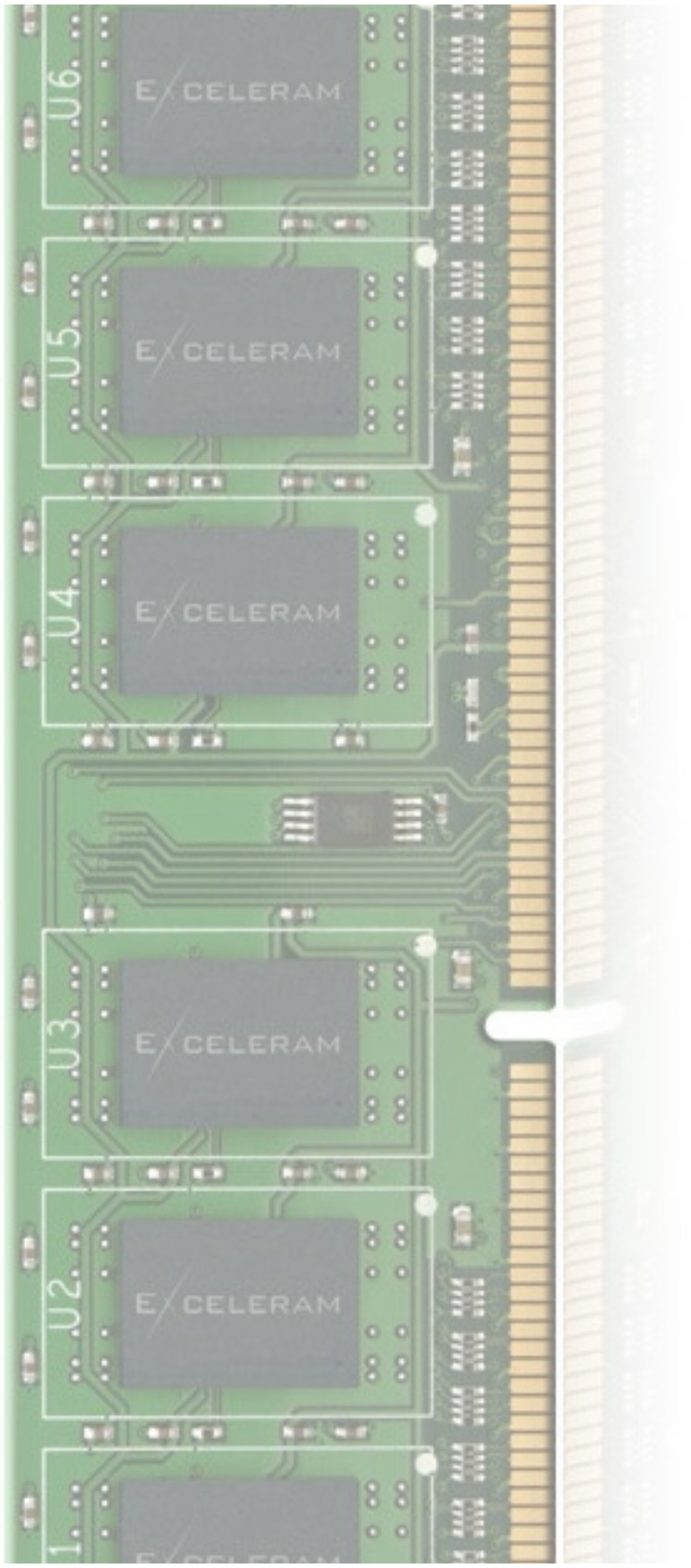
The 12th Bar-Ilan Winter School on Cryptography
Advances in Secure Computation

Center for Research in Applied Cryptography and Cyber Security

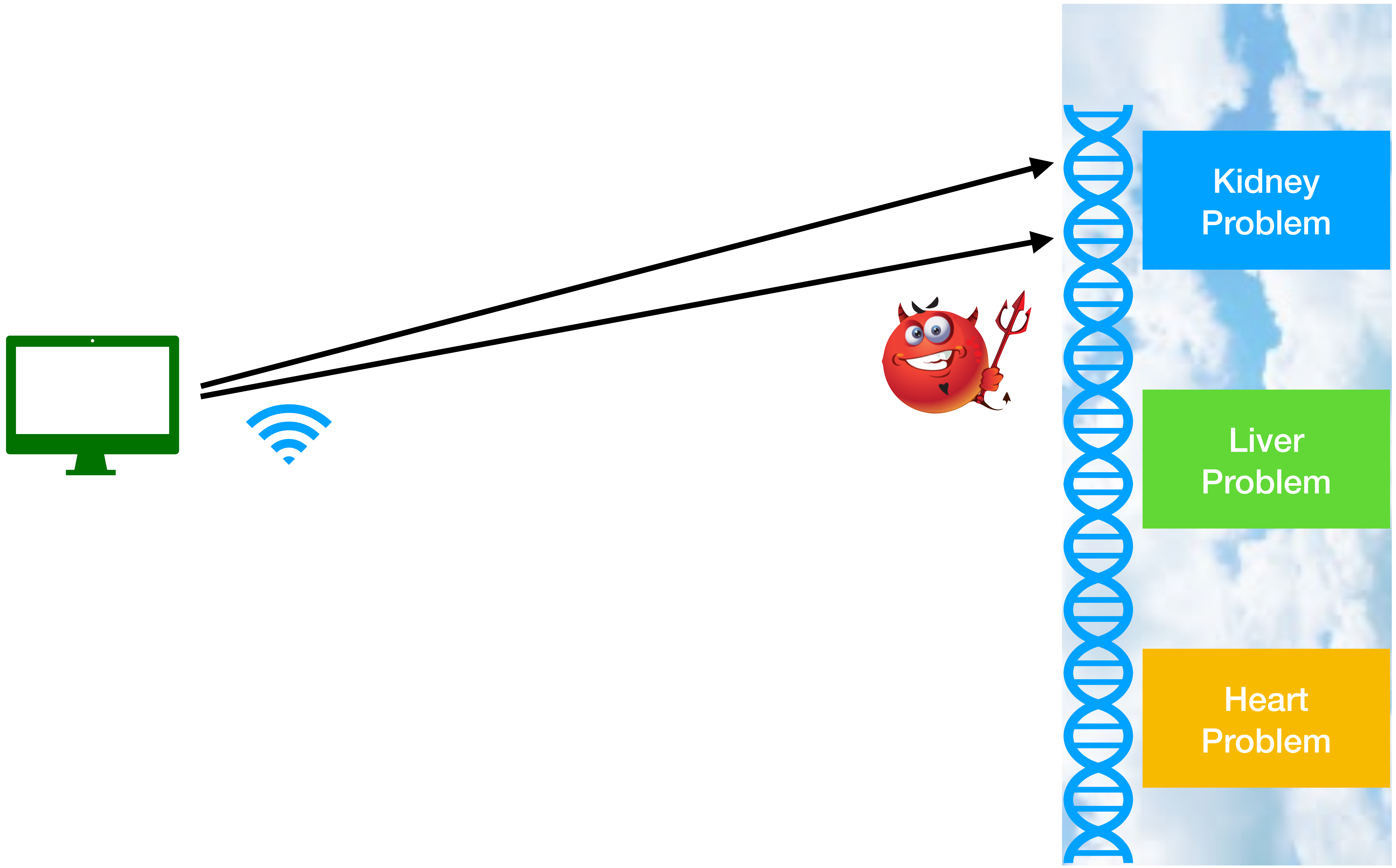
Access Patterns Reveal Information!



secure processor



Access Patterns Reveal Information!



Oblivious RAM Compiler: State of the Art

Lower bound: $\Omega(\log N)$

[GoldreichOstrovsky'96, LarsenNeilsen'18]

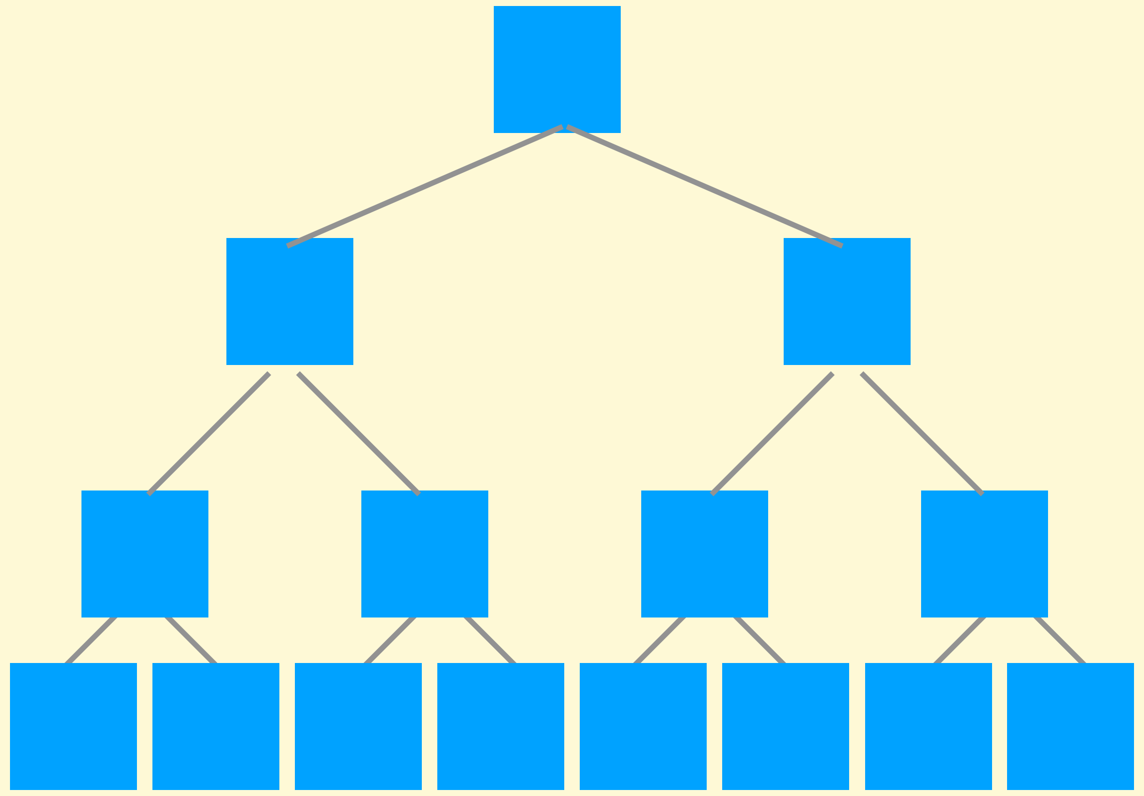
Hierarchical

[O90,GO96]

$$O(\log N)$$

Computational security

[OptORAMa'20]



Tree based ORAM

[Shi,Chan,Stefanov11]

$$O(\log^2 N)$$

Statistical security

[PathORAM,CircuitORAM]

BIU Center for Research in Applied Cryptography and Cyber Security

OptORAMa

[Asharov, Komargodski, Lin, Nayak, Peserico, Shi'20]

There exists an ORAM with O(log N) worst-case overhead

🎉 **Asymptotically Optimal!** 🎊

- Computational Security (OWF)
  - Matches [LN'18]
- PRF -> Random Oracle
  - Statistical security
  - Matches [GO'96]

- **Word size:** log N
- Client's memory size O(1) words
- Passive server
- **Balls and bins** model
- Large hidden constant
- Based on hierarchical ORAM

A Short Tutorial

Hierarchical Solution

$O(\log^3 N), \ldots, O(\frac{\log^2 N}{\log \log N})$

[Ostrovsky'90],…,[KLO12]

PanORAMa

$O(\log N \log \log N)$

Patel,Persiano,Raykova,Yeo'18

OptORAMa

$O(\log N)$

Center for Research in Applied Cryptography and Cyber Security

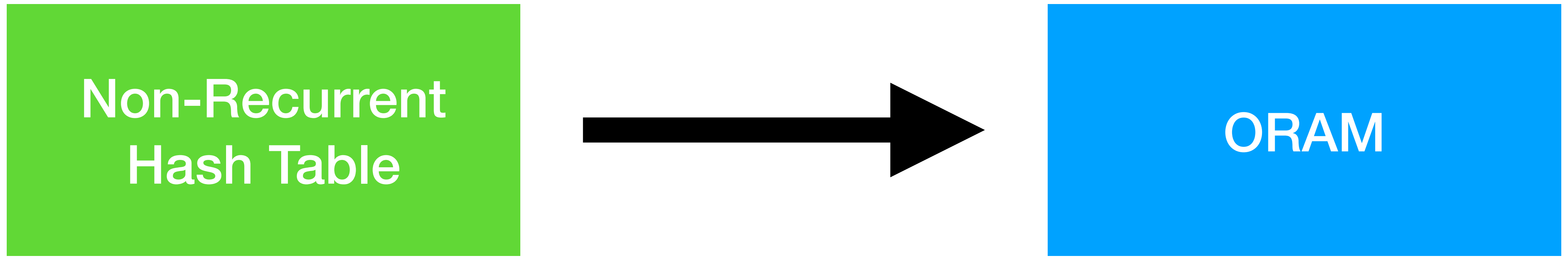
Hierarchical Solution

$O(\log^3 N), \ldots, O(\frac{\log^2 N}{\log \log N})$

[Ostrovsky'90],...,[KLO12]

Hierarchical ORAM

[Goldreich and Ostrovsky 1996]



Hierarchical Solution

$O(\log^3 N), \ldots, O(\frac{\log^2 N}{\log \log N})$

[Ostrovsky'90],...,[KLO12]

Non-Recurrent Hash Table

Build(X):

X is an array of pairs <addr,val>

Lookup(addr):

If addr $\in$ X, return val; otherwise return $\perp$

Also supports "dummy lookups" (addr $= \perp$)

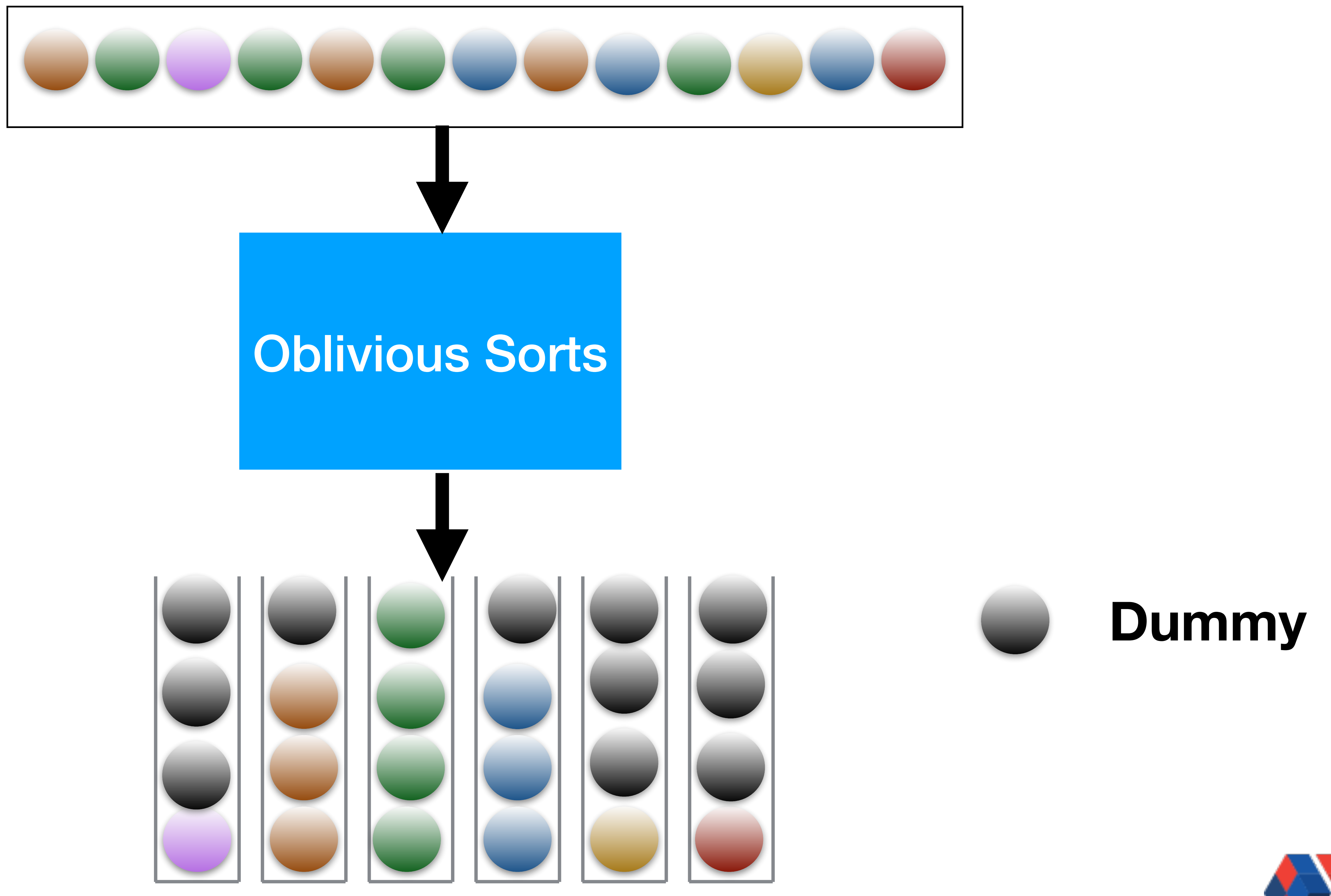
Security holds as long as each addr **is looked up at most once!**

Non-Recurrent Hash Table

- Balls into bins
- Each level has a PRF key K - mark ball `addr` to bin PRF_K(`addr`)

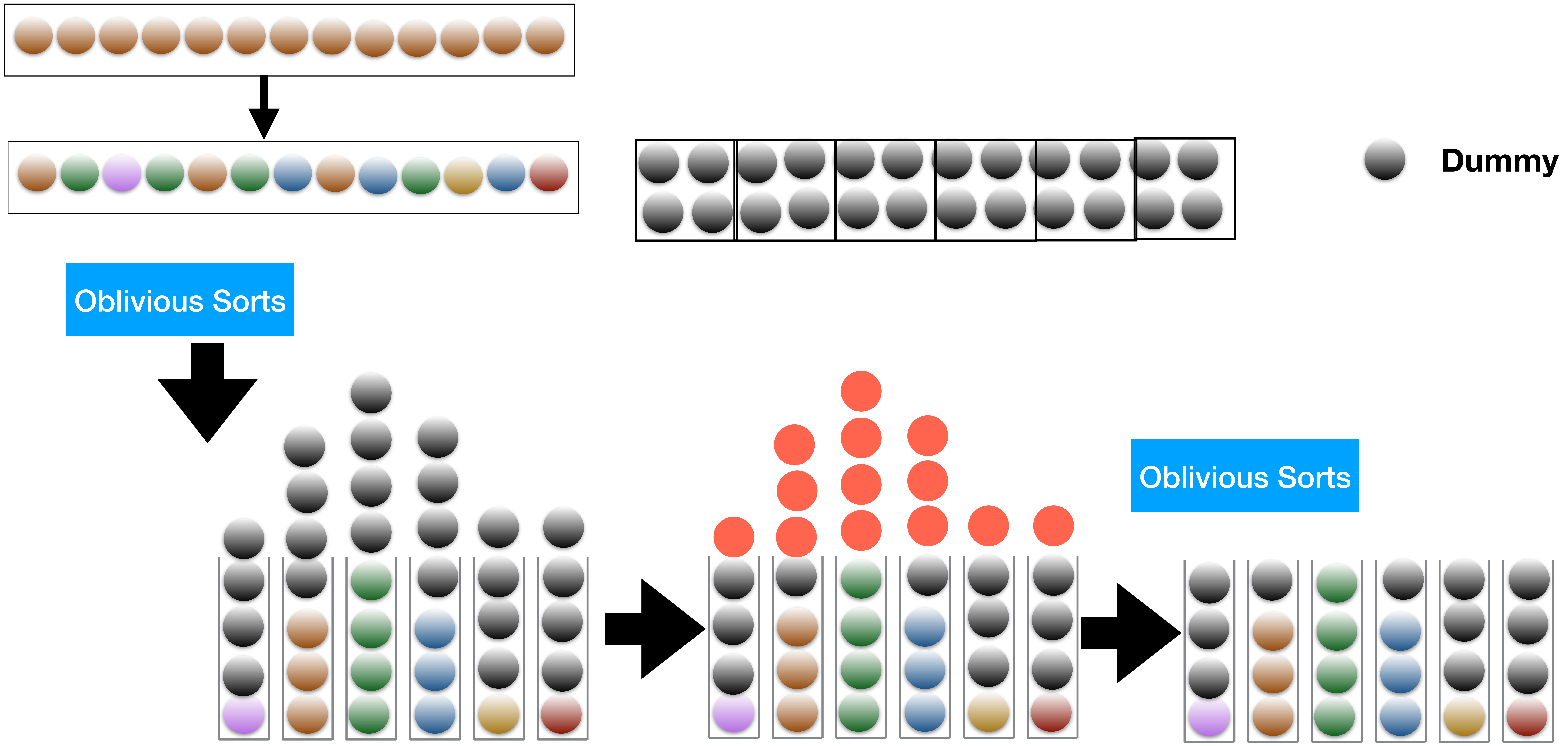
Build O(n log n), Lookup O(log n ω(1))

Implementation:



$2n/\log^2 n$ bins of size : $\log^2 n$

"Bin Packing"



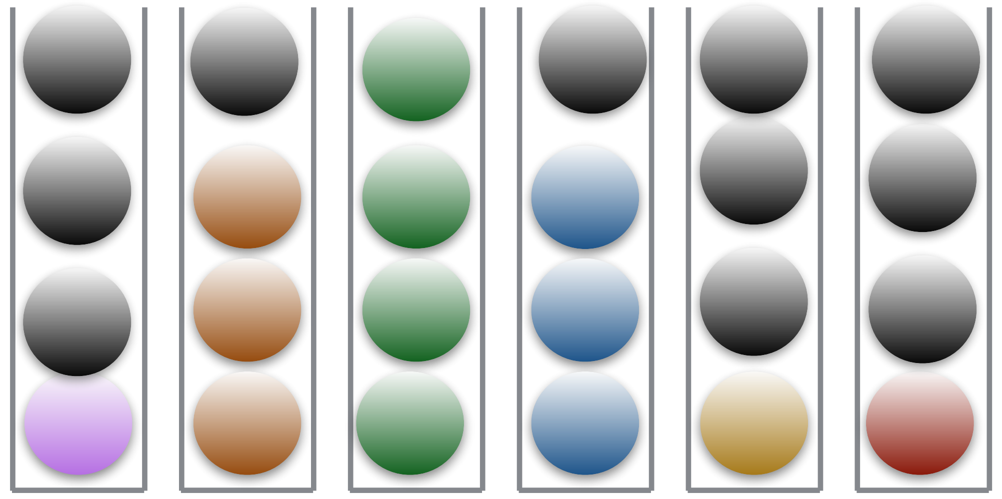
Lookup

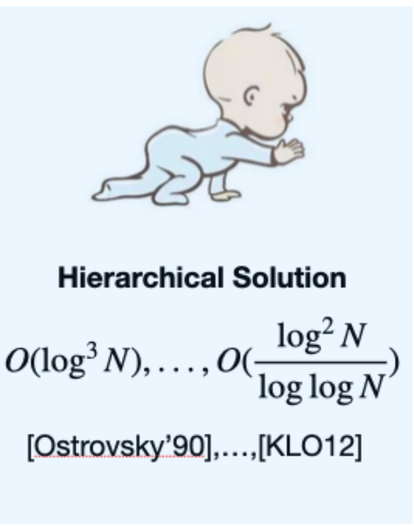
It is guaranteed that we do not look for the same **addr** twice!

- Lookup(`addr`): visit bin PRF_K(`addr`) and scan for `addr`
- Lookup(dummy): visit and scan a random bin

Simulate Build: Oblivious sorts - easy
Simulate Lookup: Each Lookup() -> scan a random bin

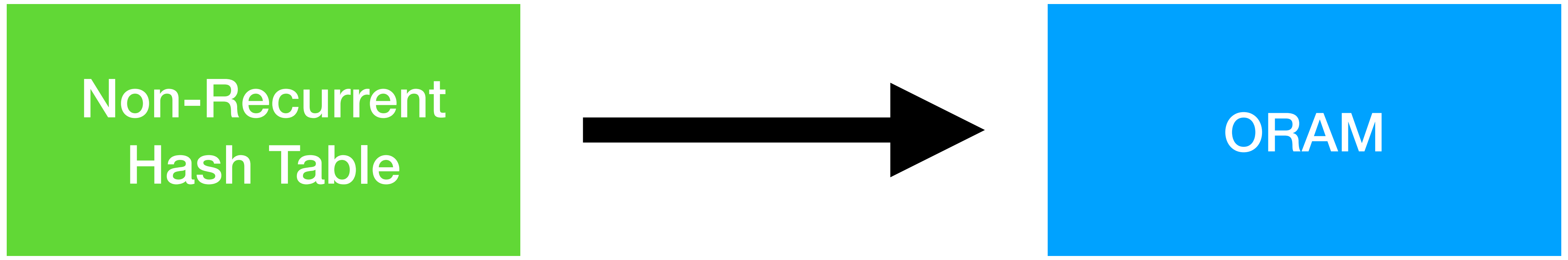
Cost: Build — $O(n \log n)$**, each lookup** $O(\log^2 n)$

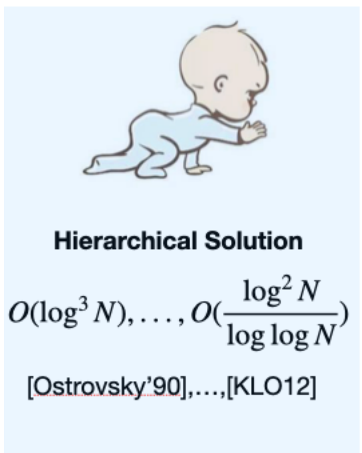




Hierarchical ORAM

[Goldreich and Ostrovsky 1996]

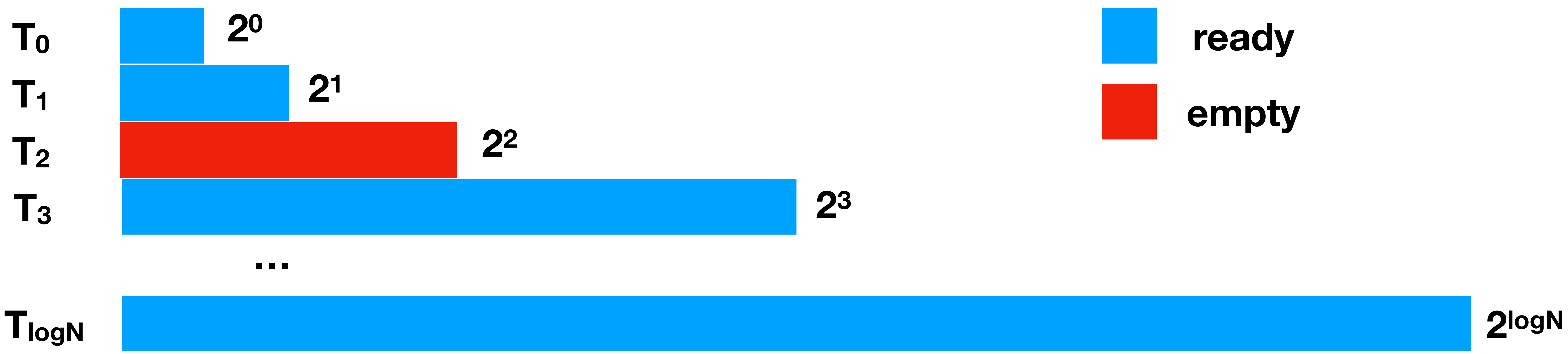


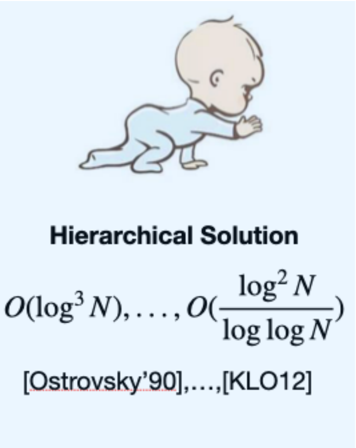


Access (op,addr,data*)

Phase I: Lookup

Phase II: Build



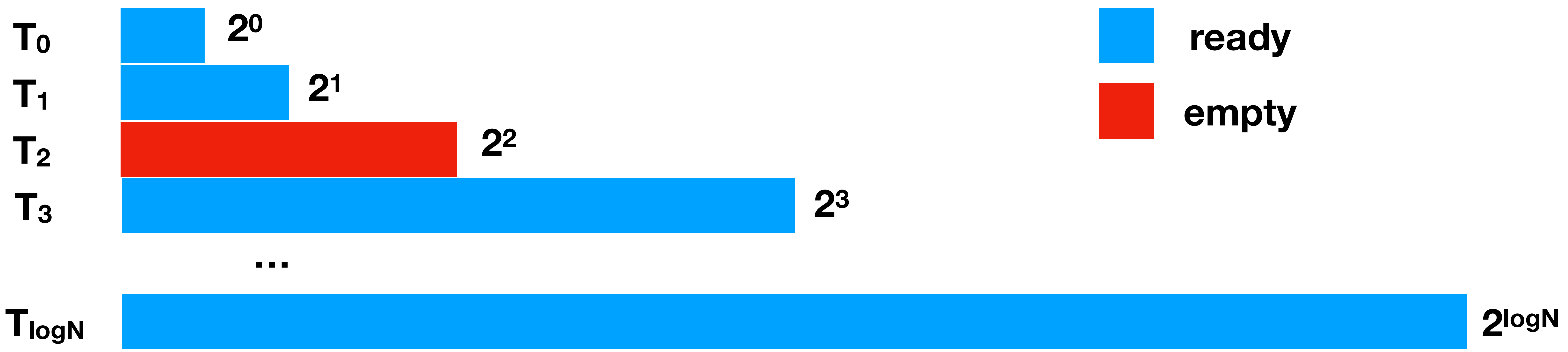


Access (op,addr,data*)

Phase I: Lookup

Perform **Lookup**(`addr`) in $T_1, \ldots, T_{\log N}$
If item found in T_i, then **Lookup**(⊥) in $T_{i+1}, .., T_{\log N}$

Phase II: Build



Hierarchical Solution

$O(\log^3 N), \ldots, O(\frac{\log^2 N}{\log \log N})$

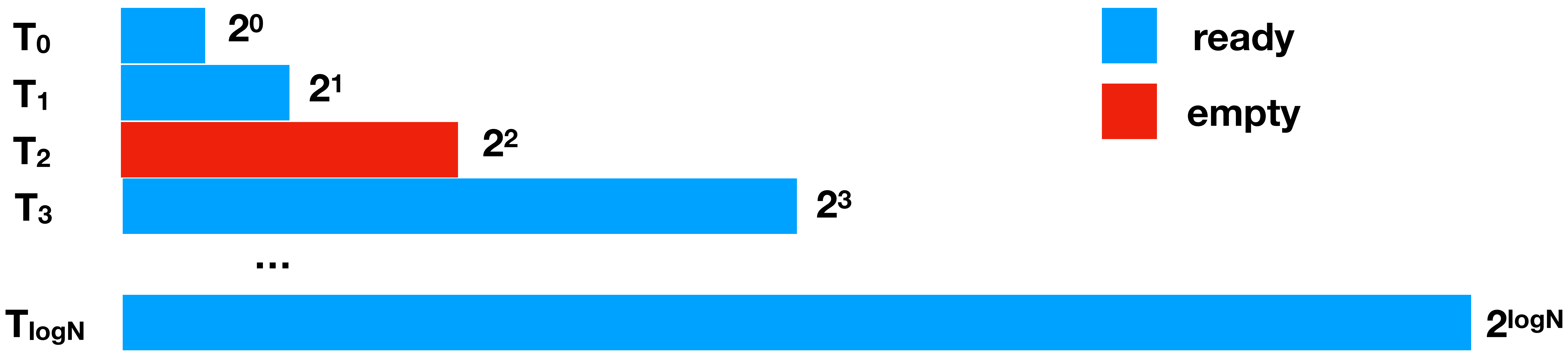
[Ostrovsky'90],…,[KLO12]

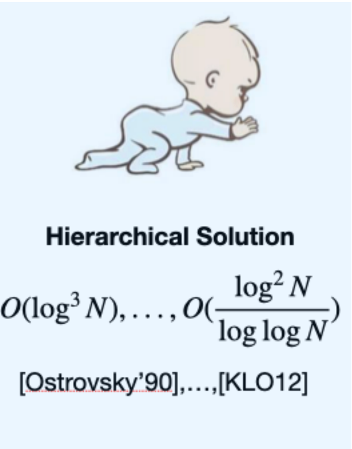
Access (op,addr,data*)

Phase I: Lookup

If op=`read`, then store the found item as **v**
If op=`write`, then ignore the found item and use **v = data***

Phase II: Build



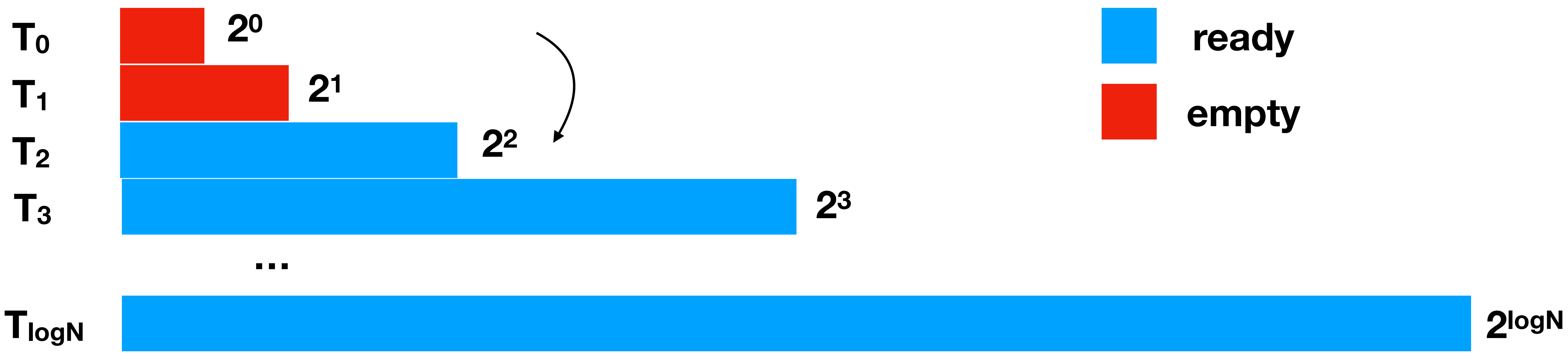


Access (op,addr,data*)

Phase I: Lookup

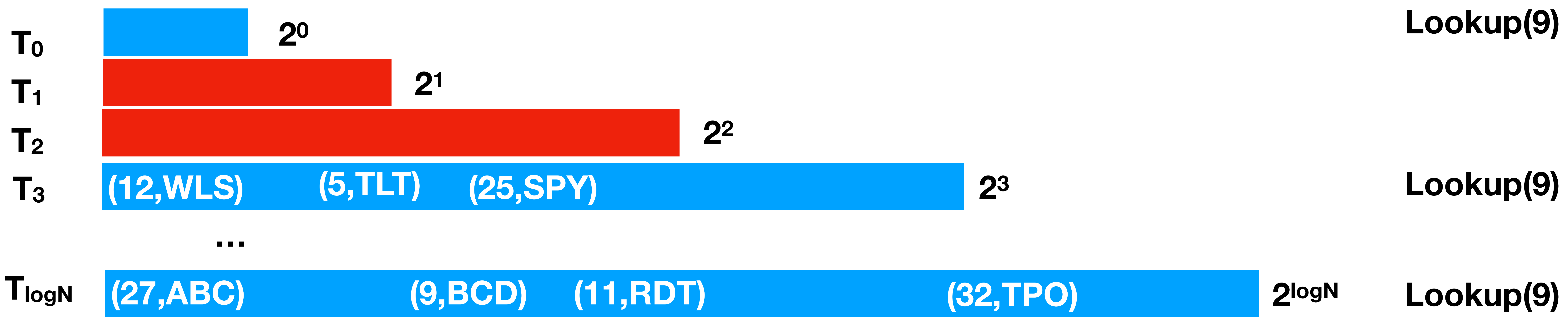
Phase II: Build

Find the first empty level l, and run T_l.**Build**($T_1 \cup \ldots \cup T_{l-1} \cup$ {<`addr`,**v**>})
Mark $T_1,\ldots,T_{l-1}$ as `empty` and T_l as `ready`



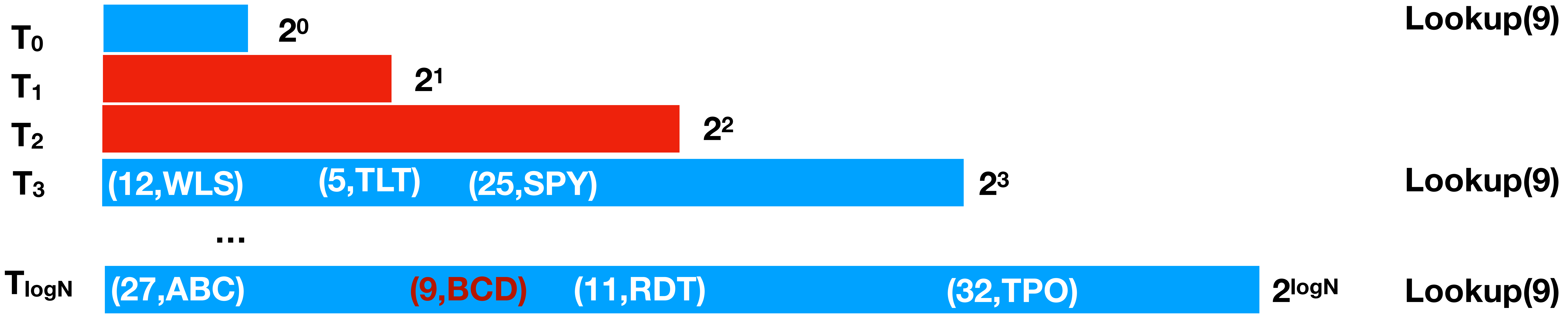
Invariant: never query the same `addr` twice between two Rebuilds

Read(9)



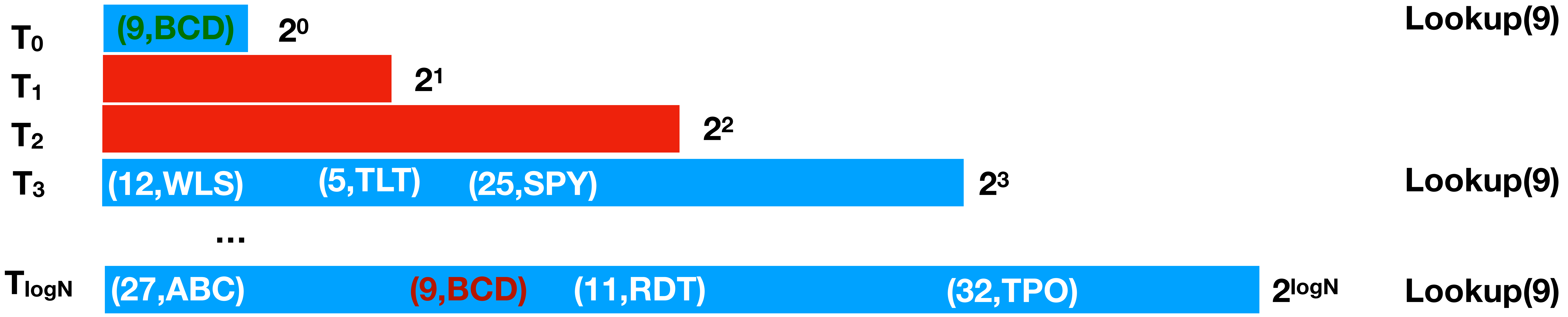
Center for Research in Applied Cryptography and Cyber Security

Read(9)

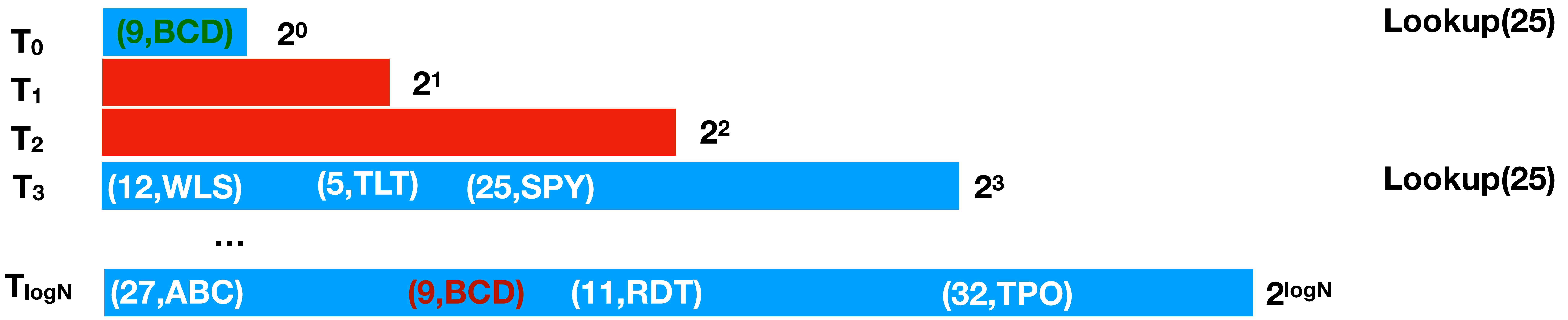


T_0
2^0
Lookup(9)
T_1
2^1
T_2
2^2
T_3
(12,WLS)
(5,TLT)
(25,SPY)
2^3
Lookup(9)
...
T_{logN}
(27,ABC)
(9,BCD)
(11,RDT)
(32,TPO)
2^{logN}
Lookup(9)

Read(9)

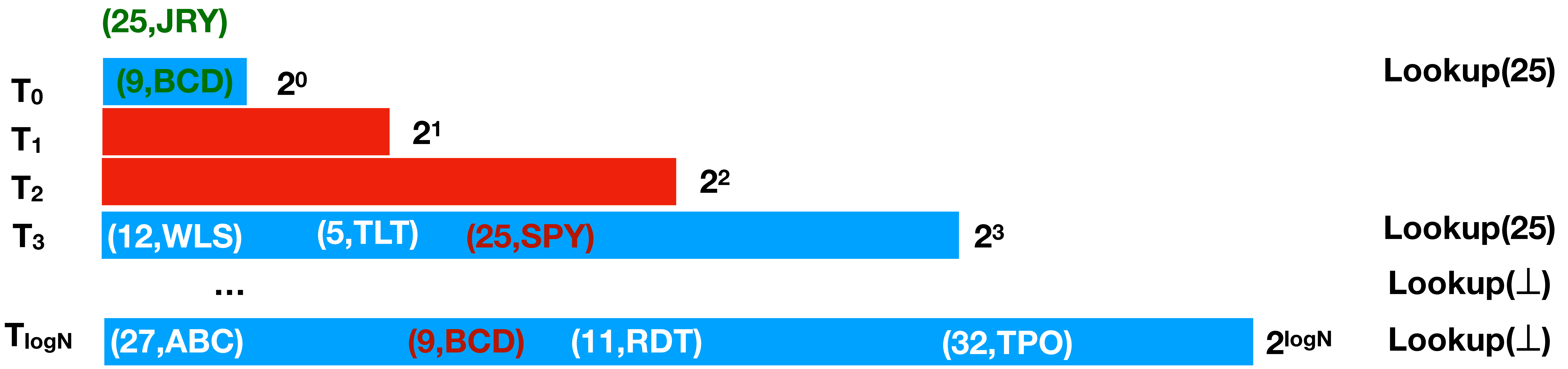

T_0
(9,BCD)
2^0
Lookup(9)
T_1
2^1
T_2
2^2
T_3
(12,WLS)
(5,TLT)
(25,SPY)
2^3
Lookup(9)
...
T_{logN}
(27,ABC)
(9,BCD)
(11,RDT)
(32,TPO)
2^{logN}
Lookup(9)

Write(25,JRY)

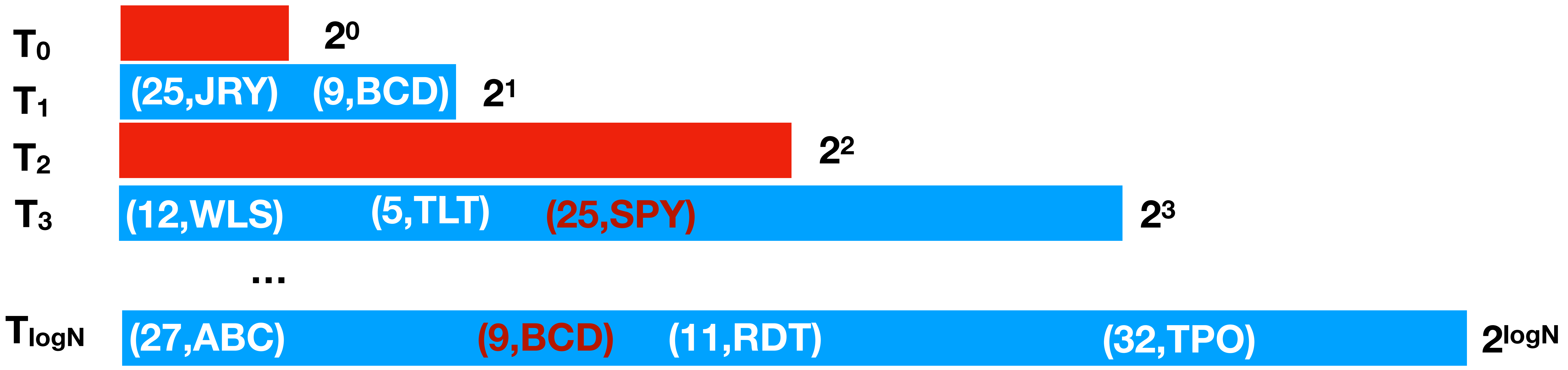


T_0
(9,BCD)
2^0
Lookup(25)
T_1
2^1
T_2
2^2
T_3
(12,WLS)
(5,TLT)
(25,SPY)
2^3
Lookup(25)
...
T_{logN}
(27,ABC)
(9,BCD)
(11,RDT)
(32,TPO)
2^{logN}

Write(25,JRY)

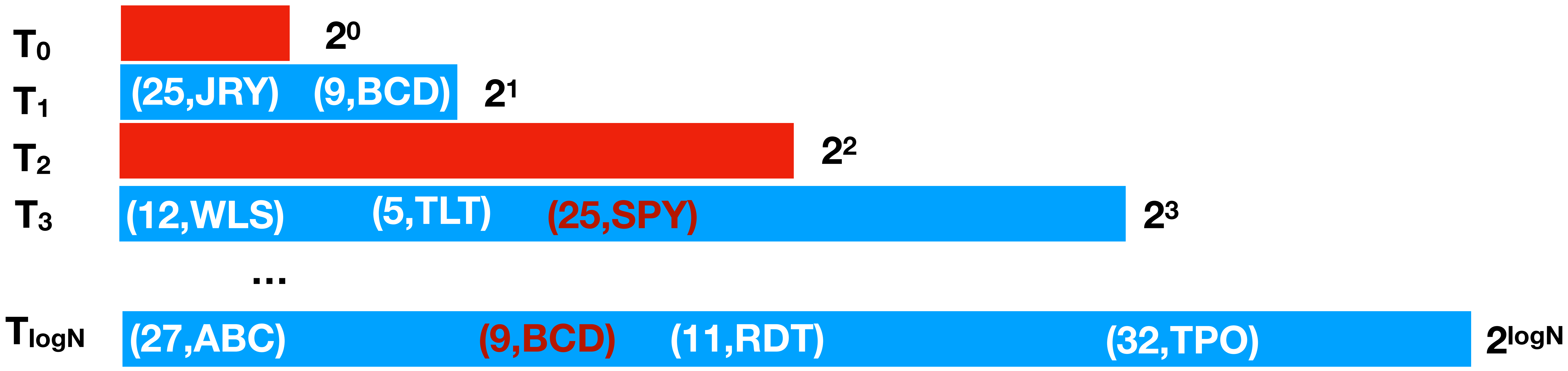

(25,JRY)
T0
(9,BCD)
2^0
Lookup(25)
T1
2^1
T2
2^2
T3
(12,WLS)
(5,TLT)
(25,SPY)
2^3
Lookup(25)
...
Lookup(⊥)
TlogN
(27,ABC)
(9,BCD)
(11,RDT)
(32,TPO)
2^logN
Lookup(⊥)

Rebuild

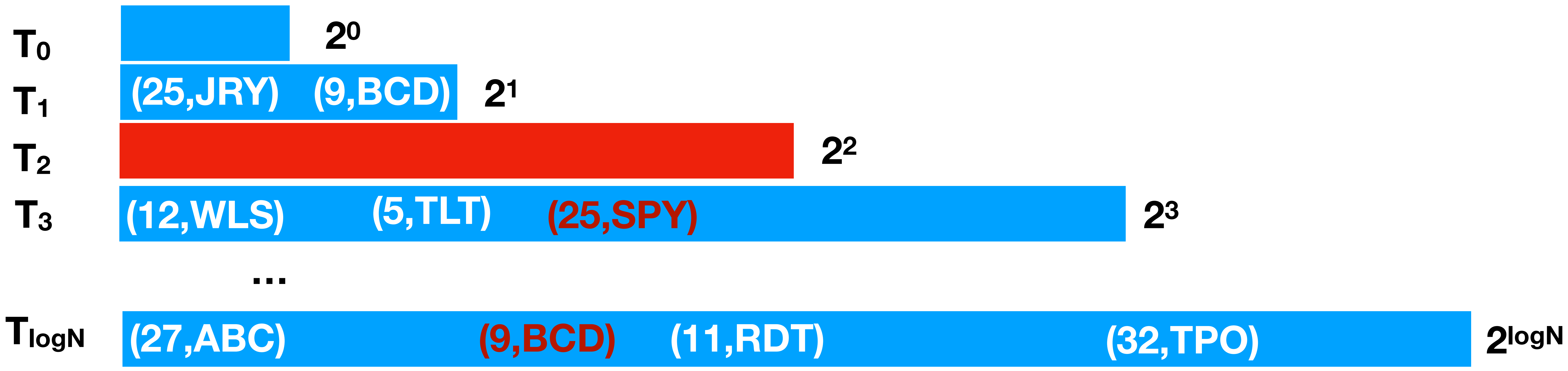


T0
2^0
T1
(25,JRY)
(9,BCD)
2^1
T2
2^2
T3
(12,WLS)
(5,TLT)
(25,SPY)
2^3
…
TlogN
(27,ABC)
(9,BCD)
(11,RDT)
(32,TPO)
2^logN

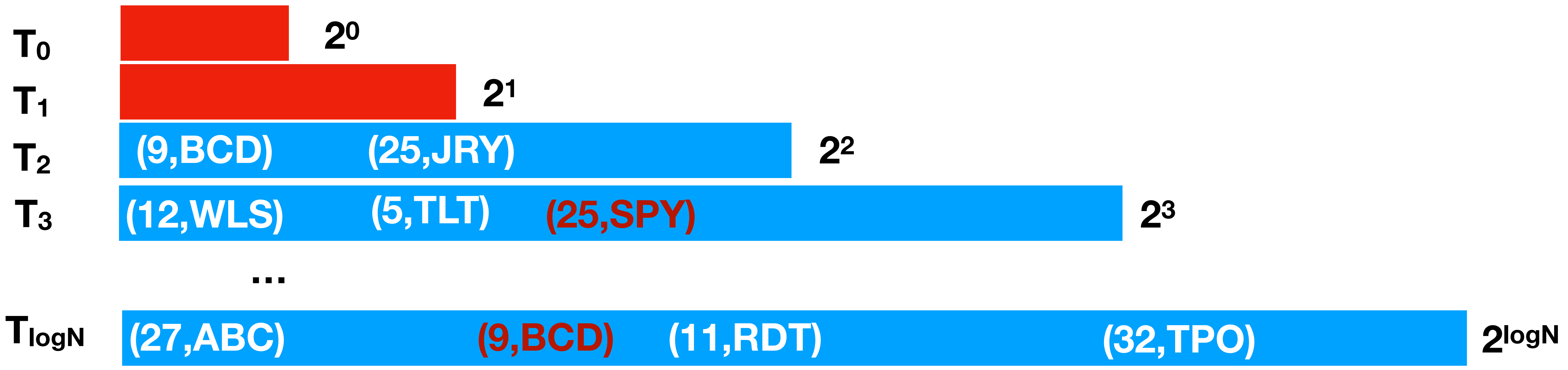
After Some More Accesses…

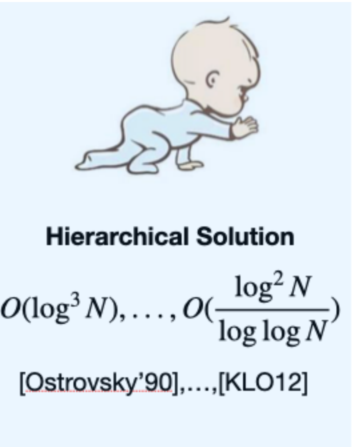


After Some More Accesses…



After Some More Accesses…



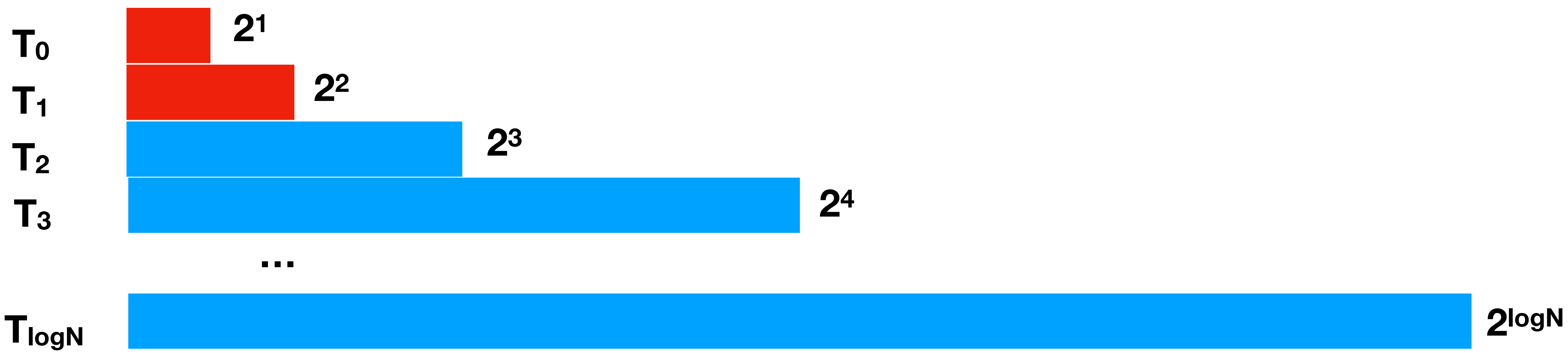


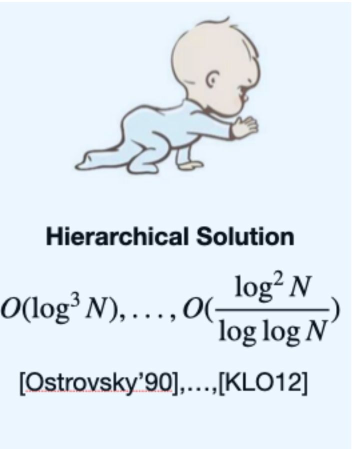
Total Cost - Basic Hierarchical ORAM

Lookup: perform lookup in $\log N$ levels, each requires $\log^2 N$ — $O(\log^3 N)$

Rebuild: Rebuild level i every 2^i accesses, over N accesses: — $O(\log^2 N)$

$$\sum_{i=1}^{\log N} \frac{N}{2^i} \cdot 2^i \cdot \log 2^i = N \cdot \sum_{i=1}^{\log N} i \approx N \log^2 N$$





Improvements [GM'11,KLO'12]

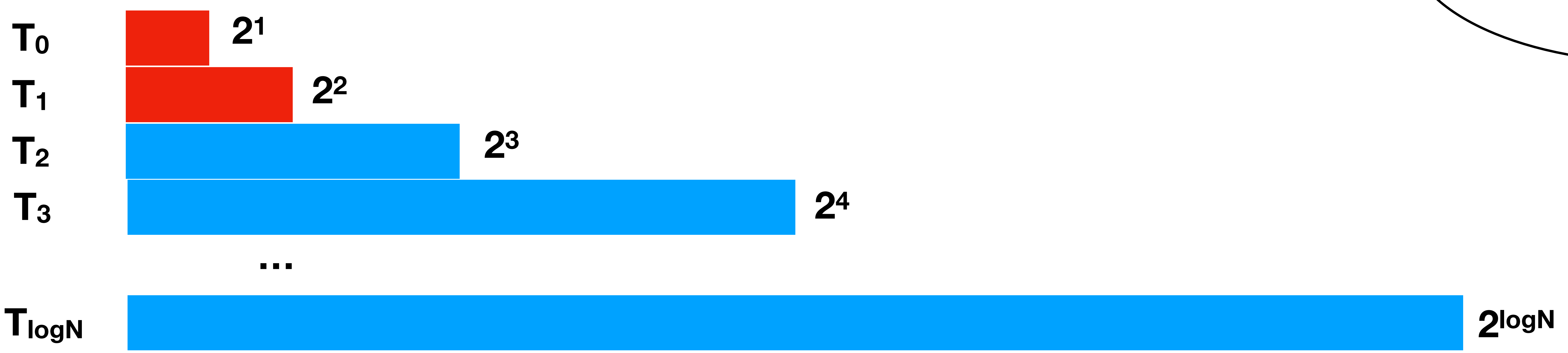
Lookup: perform lookup in $\log N$ levels, each requires ~~$\log^2 N$~~ effectively $O(1)$

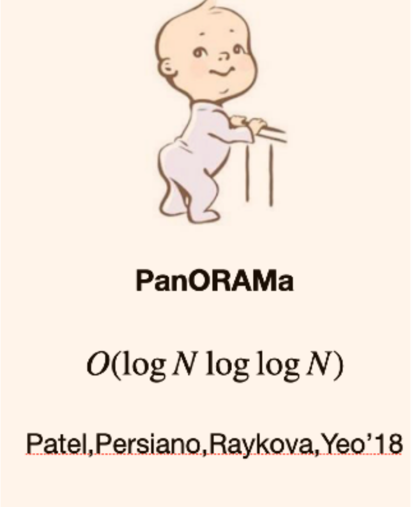
~~$O(\log^3 N)$~~
$O(\log N)$

Rebuild: Rebuild level i every 2^i accesses

$O(\log^2 N)$

Using hash tables on the bins themselves + stashes

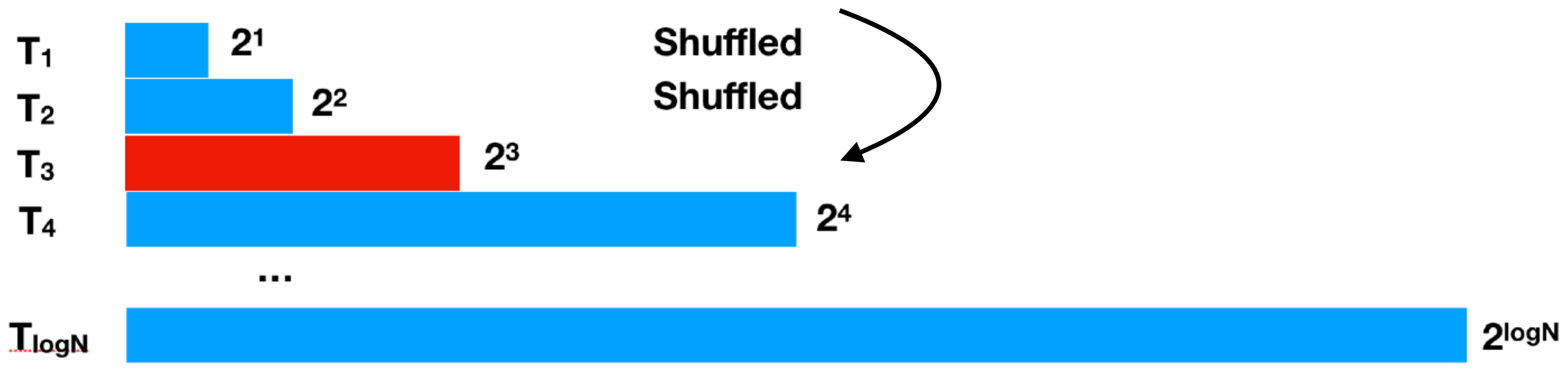




From Hierarchical ORAM to PanORAMa

- **PanORAMa**: **Rebuild** HT for a ***randomly shuffled*** input in $O(N \log \log N)$
  - All elements that were not visited - are still randomly shuffled in the eye of the adversary!

- **But…**
  - Each layer is shuffled, but the concatenation is not shuffled
    - PanORAMa showed how to “intersperse” arrays in $O(N \log \log N)$

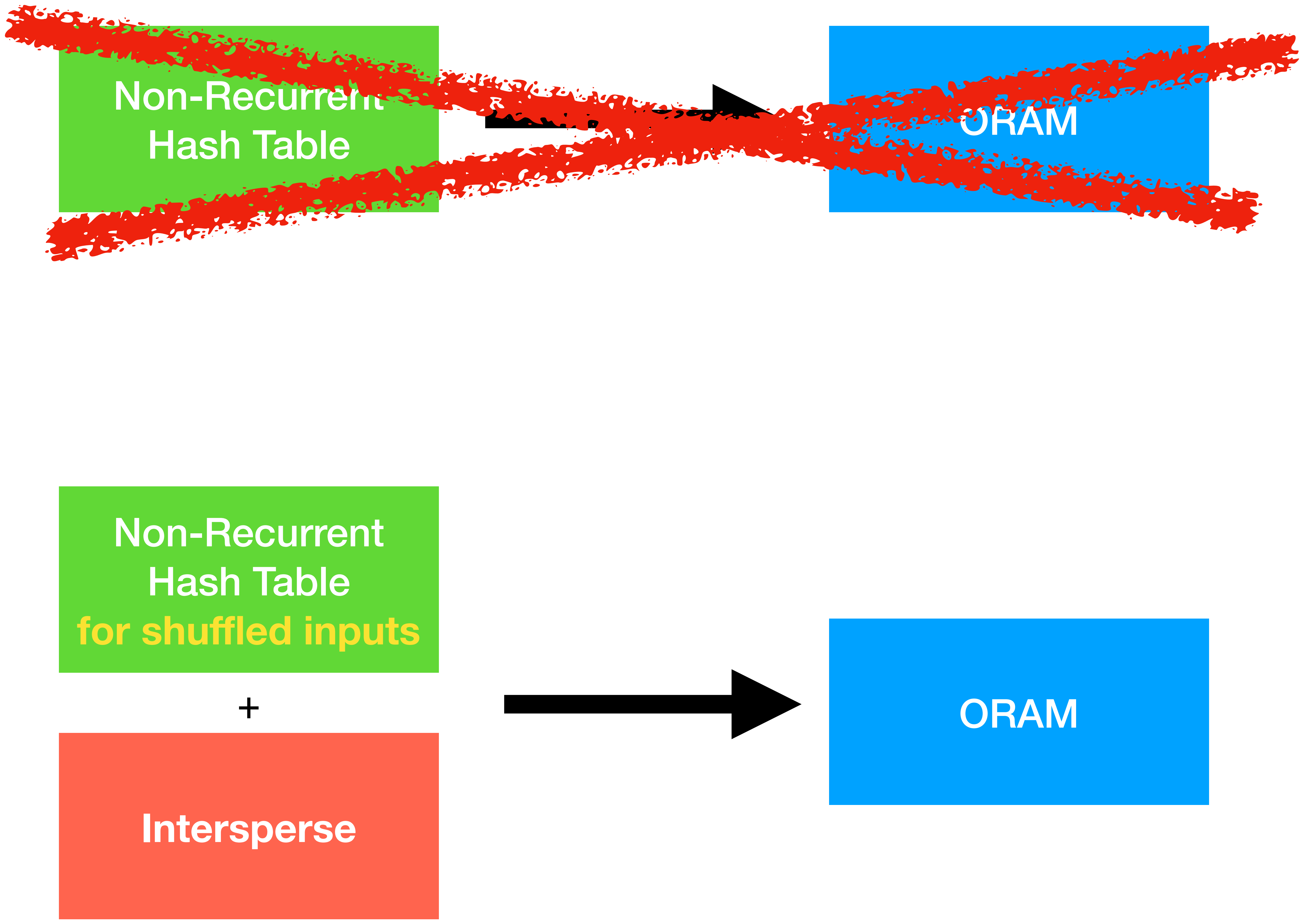


PanORAMa

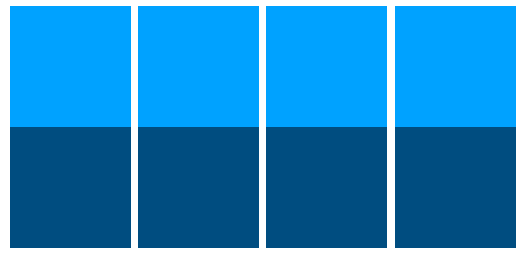
$O(\log N \log \log N)$

Patel,Persiano,Raykova,Yeo'18

PanORAMa



Intersperse

I_0  Shuffled $|I_0| = n_0$

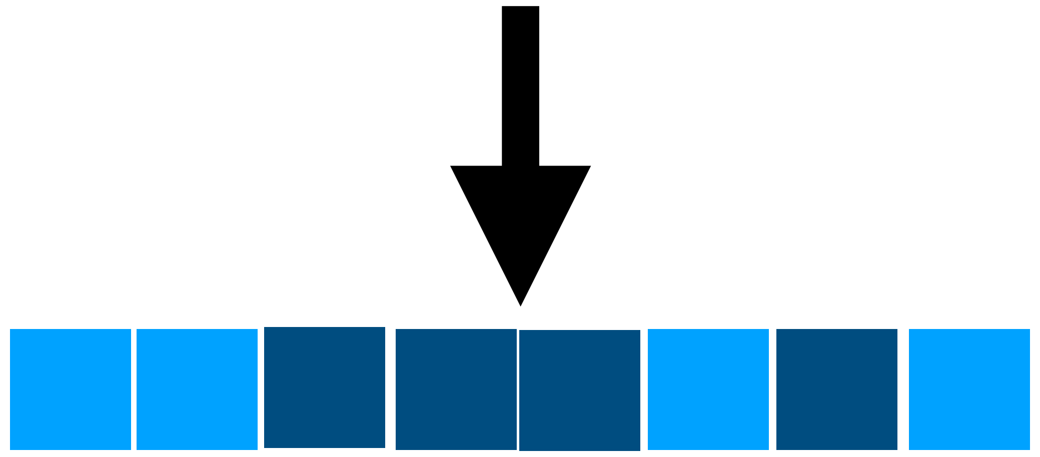
I_1 Shuffled $|I_1| = n_1$

Generate random Aux with n_0 zeros, n_1 ones $(n_0 + n_1 = n)$

0	0	1	1	1	0	1	0

,

Oblivious route



$$\binom{n}{n_0} \cdot n_0! \cdot n_1! = n!$$

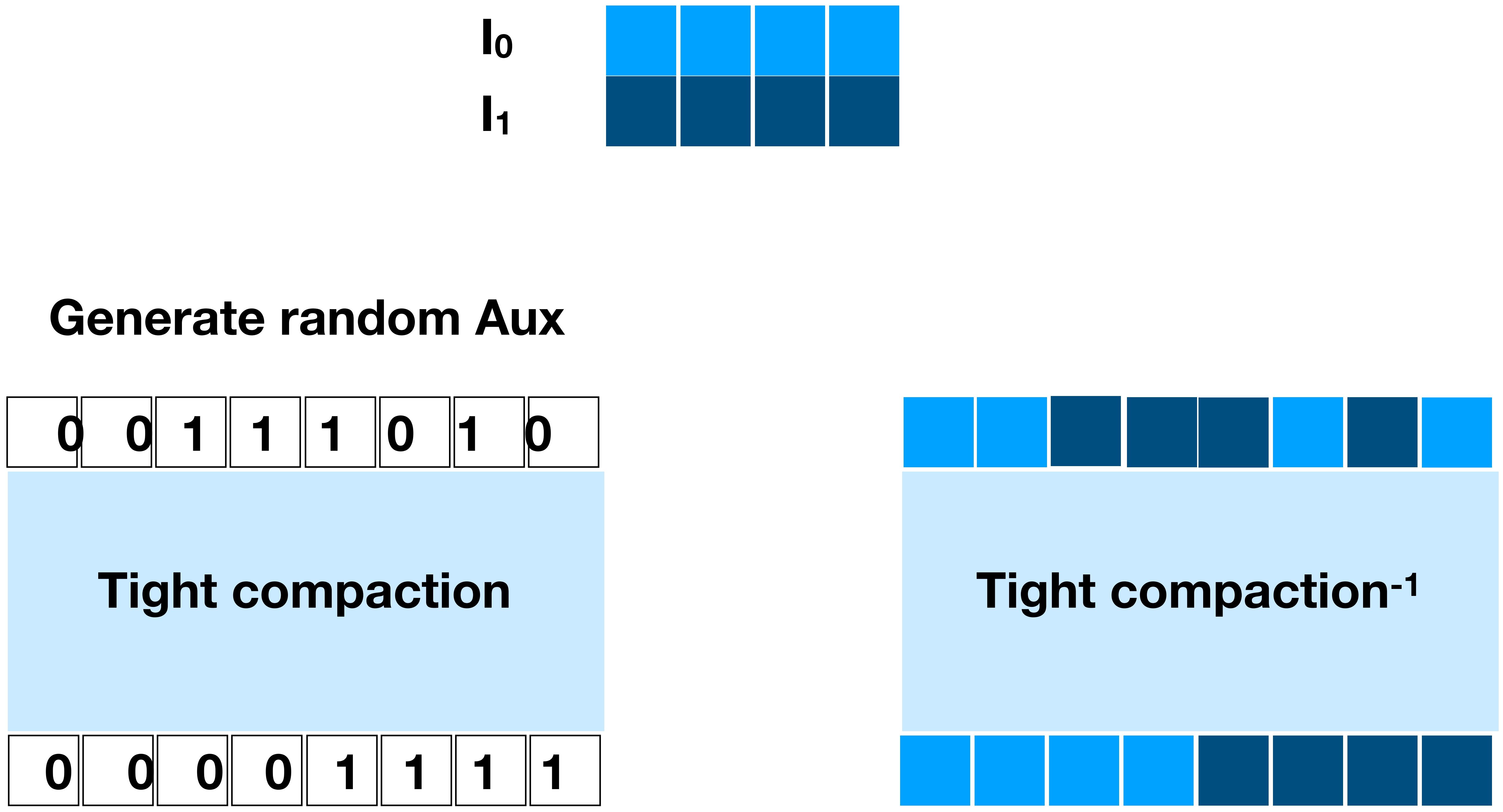
$$n = n_0 + n_1$$

Challenge: Move the elements **Obliviously**

PanORAMa: Implemented in O(n log log n)

Intersperse From Oblivious Tight Compaction

Generate random Aux



Remember all “move balls”

Perform same “swaps”

Intersperse in $O(n)$!

Rebuilding Hash Tables in Linear Time

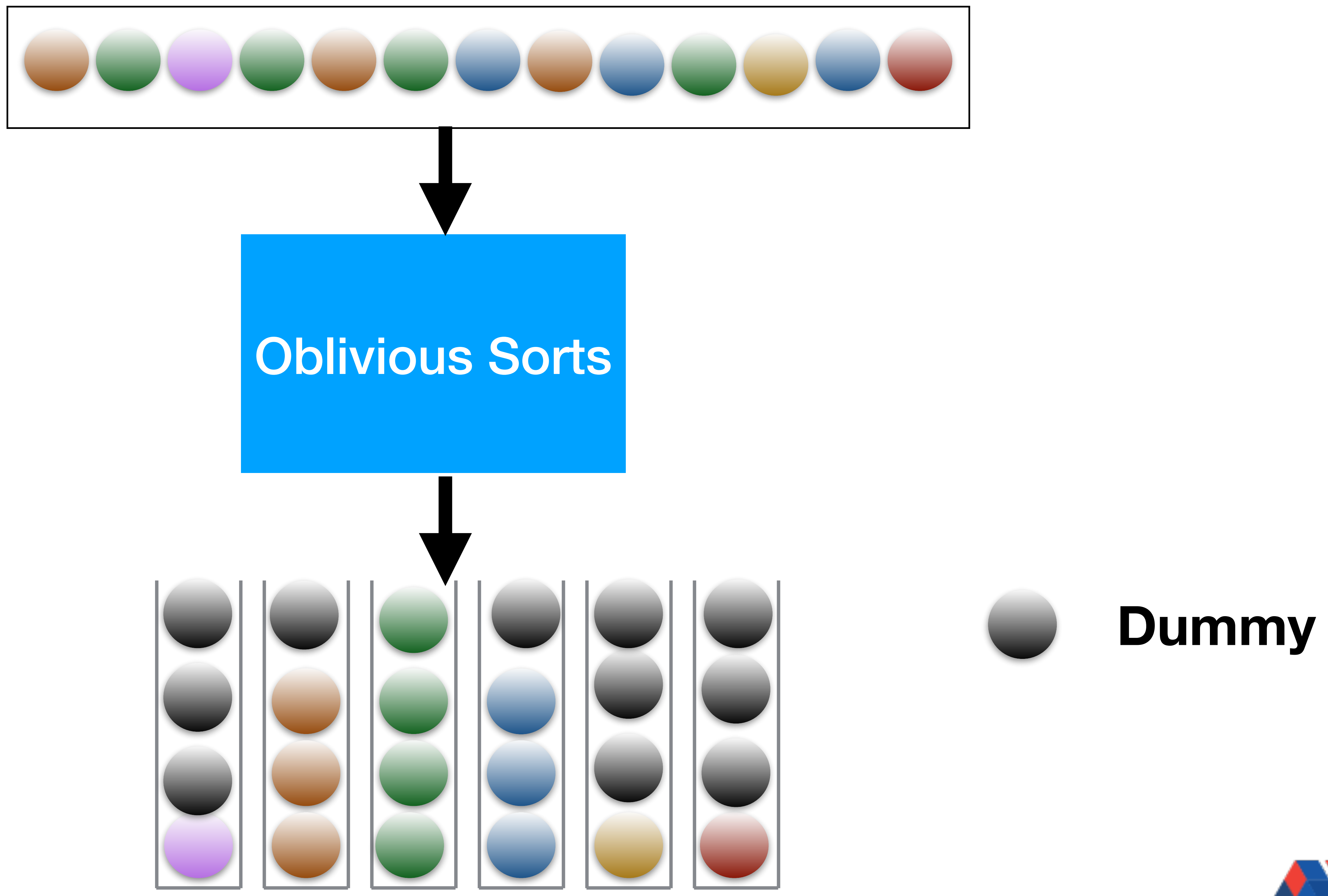
Weaker Primitive (But Suffices!) — Assumes Permuted Inputs

Warmup:
Goldreich and Ostrovsky

- Balls into bins
- Each level has a PRF key K - mark ball `addr` to bin $PRF_K(\texttt{addr})$

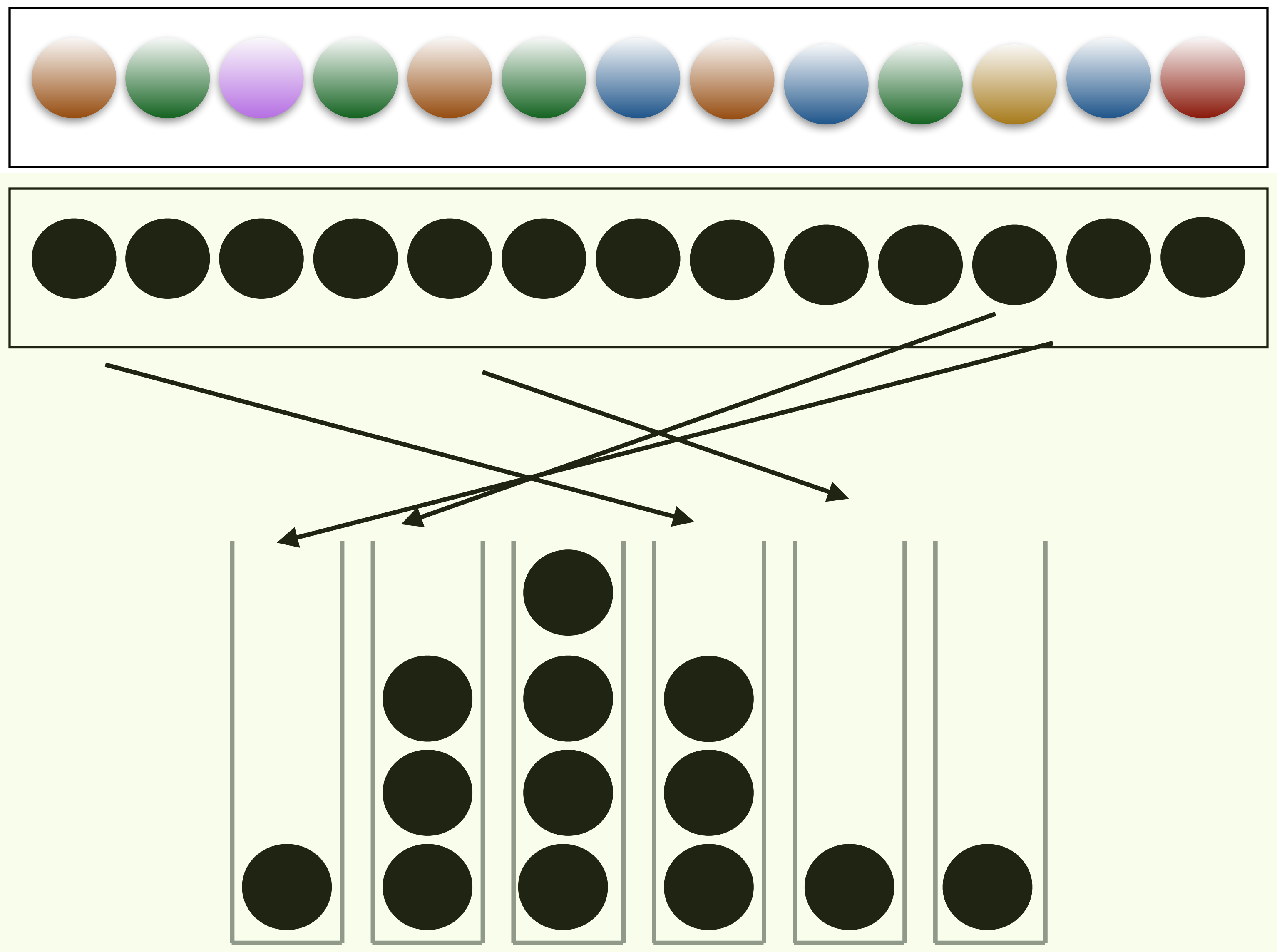
Build O(n log n), Lookup O(log n ω(1))

Implementation:

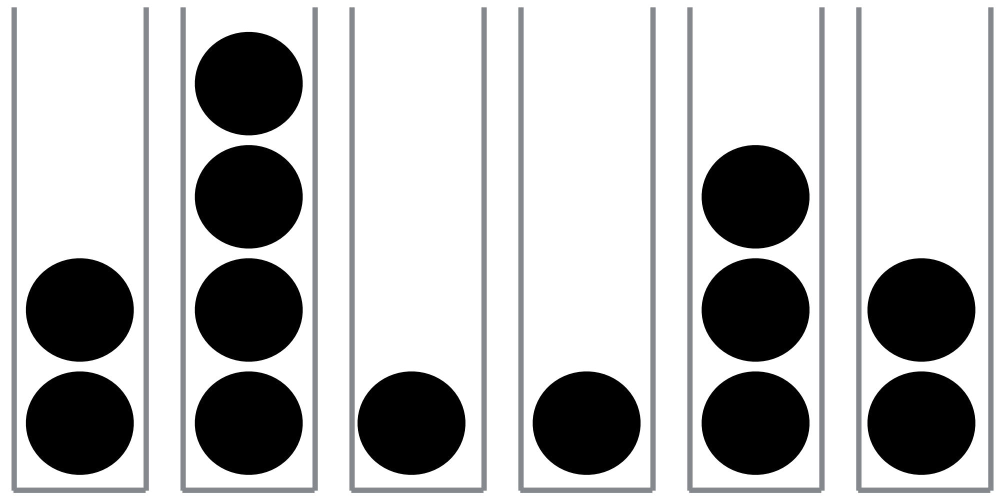


Build(X) where X is Randomly Permuted?

Build



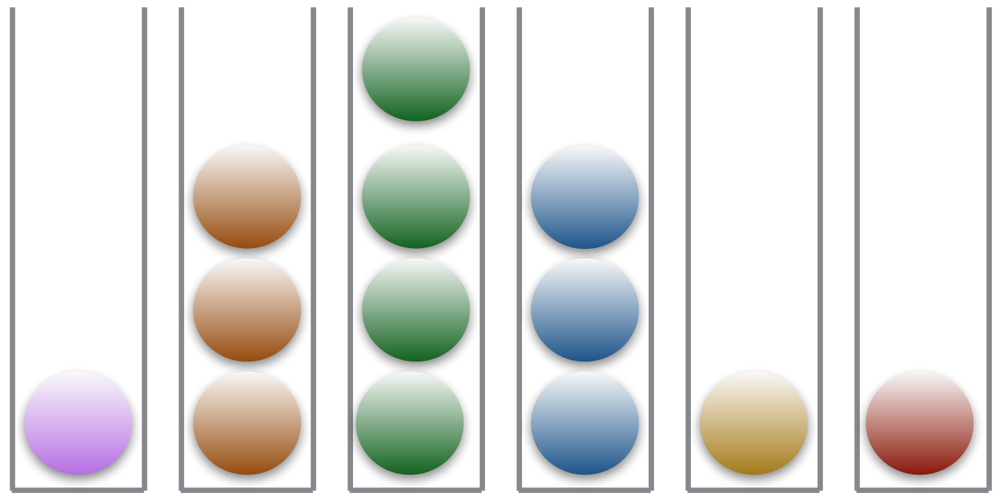
Is it secure?



n “dummy” lookups

No!

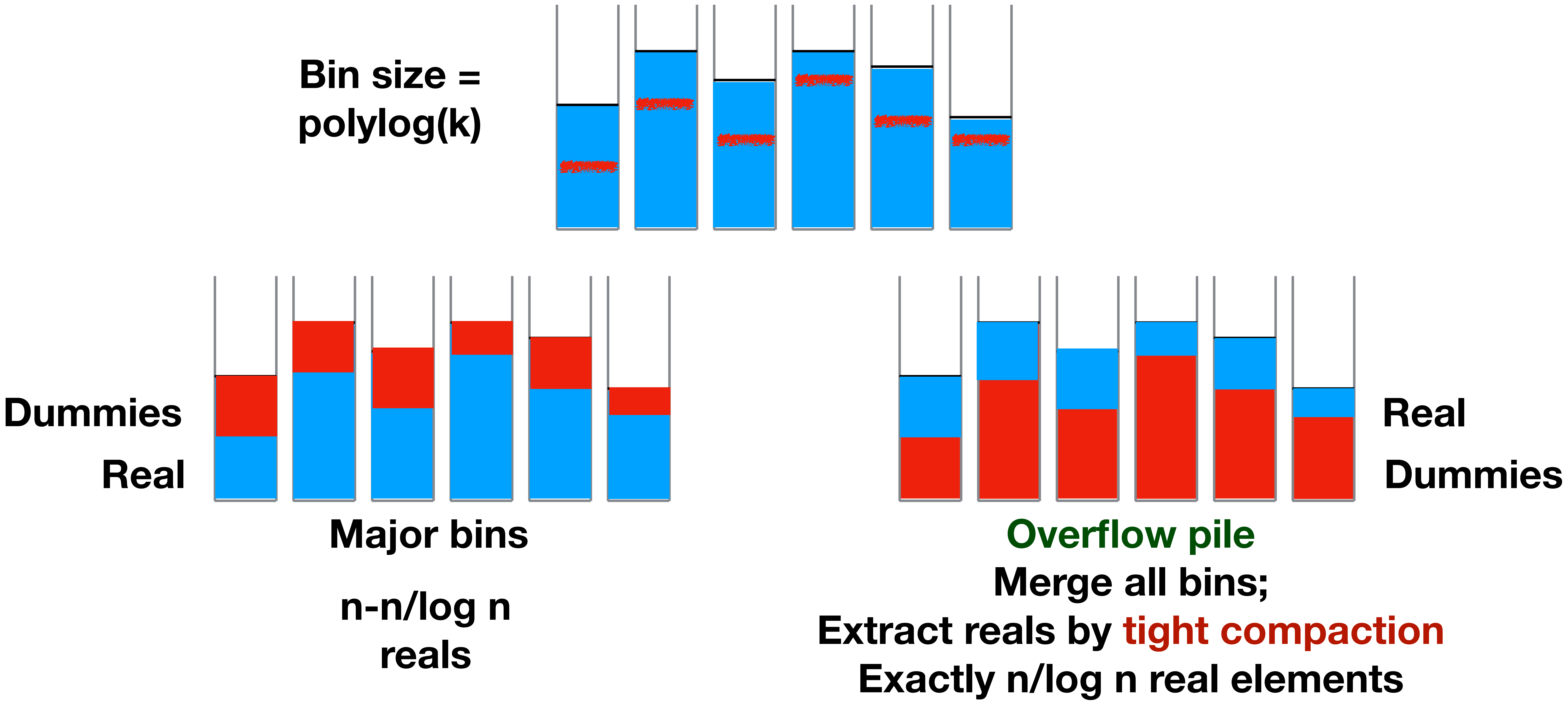
An adversary can distinguish between



n “real” lookups

OptORAMa: Build

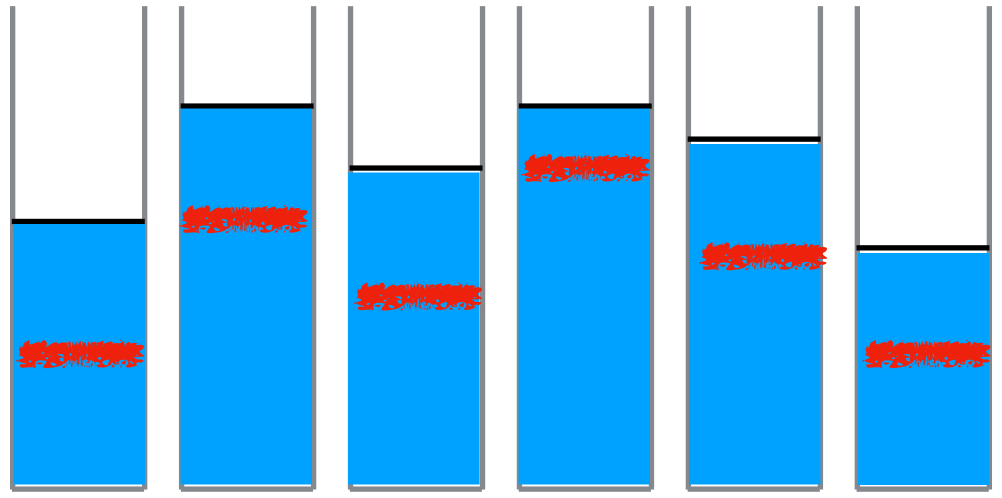
1) Throw the n elements into n/polylogk bins according to a PRF key K - **reveal access pattern**
2) Sample an independent (secret) loads of throwing n’ = n-n/log n **balls into the bins**
3) Truncate to the secret loads and pad with dummies; move truncated elements to **overflow pile**
4) **Build** each major bin using **smallHT**; build **overflow pile** using **cuckoo hash**



OptORAMa: Build

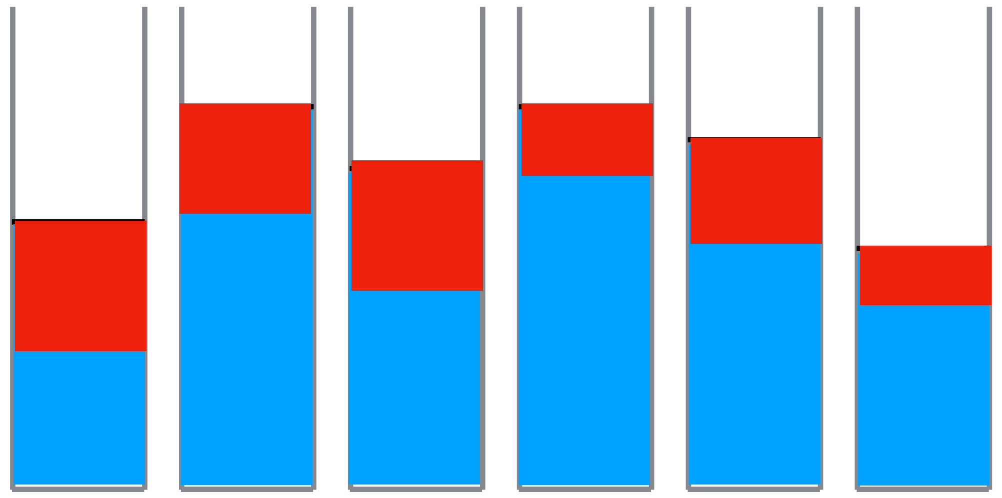
1) Throw the n elements into n/polylogk bins according to a PRF key K - **reveal access pattern**
2) Sample an independent (secret) loads of throwing n' = n-n/log n **balls into the bins**
3) Truncate to the secret loads and pad with dummies; move truncated elements to **overflow pile**
4) **Build** each major bin using **smallHT**; build **overflow pile** using **cuckoo hash**

Bin size = polylog(k)



Dummies

Real



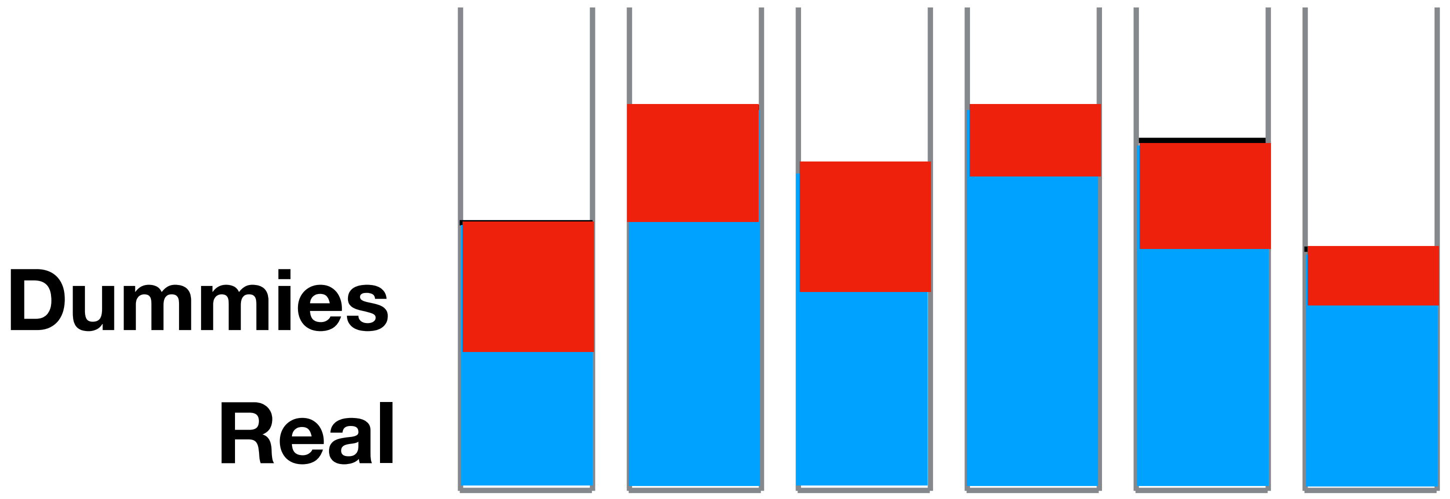
Major bins

n-n/log n reals

Overflow pile (n/logn balls)

Merge all bins;
Extract reals by tight compaction
Exactly n/log n real elements

OptORAMa: Lookup

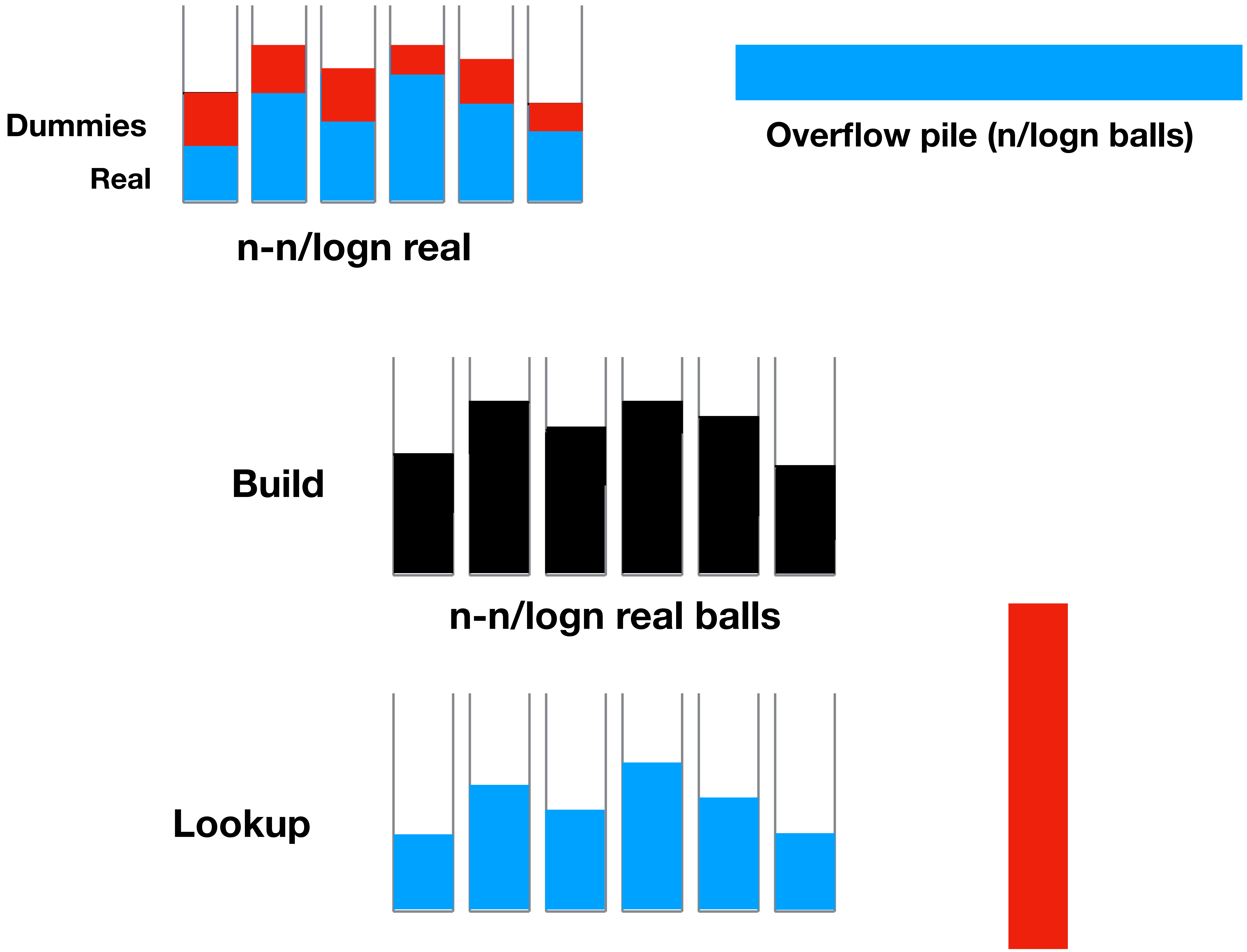


n-n/log n real balls

Overflow pile (n/log n balls)

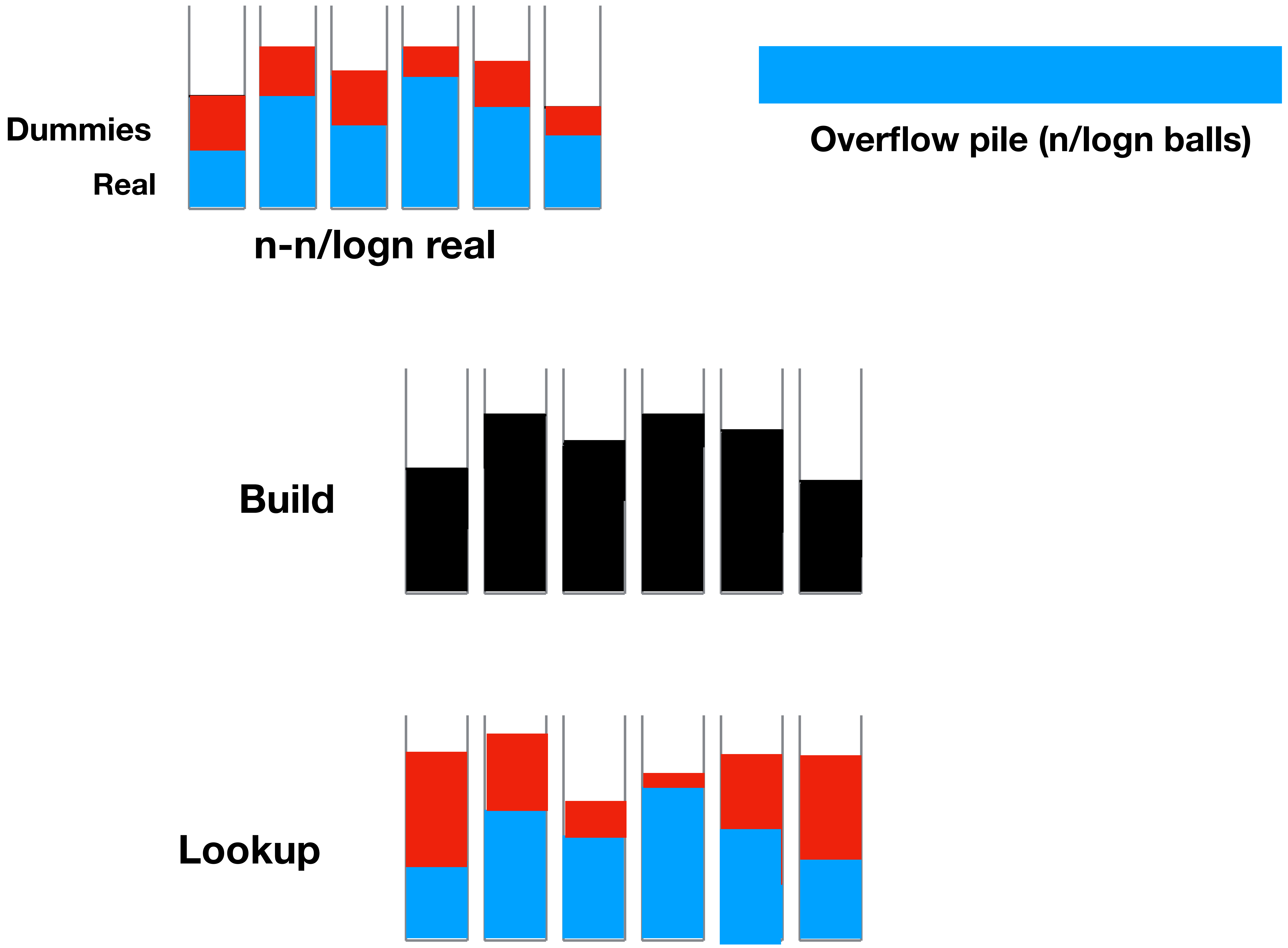
Lookup(addr):
Search in **overflow pile**;
If **found** - visit random bin
Otherwise - visit $PRF_K(addr)$

Security



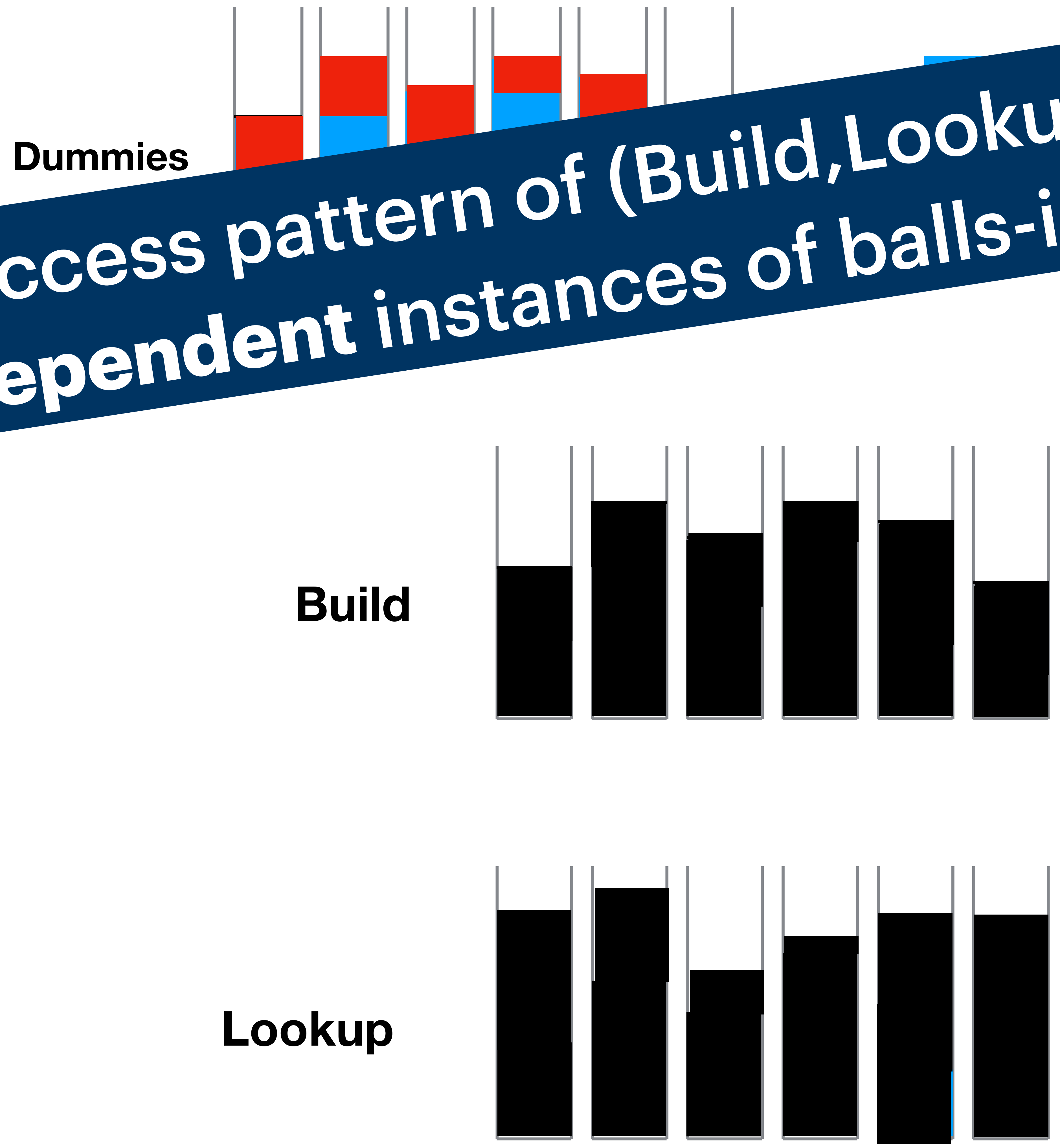
Dummies
Real
n-n/logn real
Overflow pile (n/logn balls)
Build
n-n/logn real balls
Lookup

Security


Dummies
Real
n-n/logn real
Overflow pile (n/logn balls)
Build
Lookup

Security

Access pattern of (Build,Lookup) looks like two independent instances of balls-into-bins processes

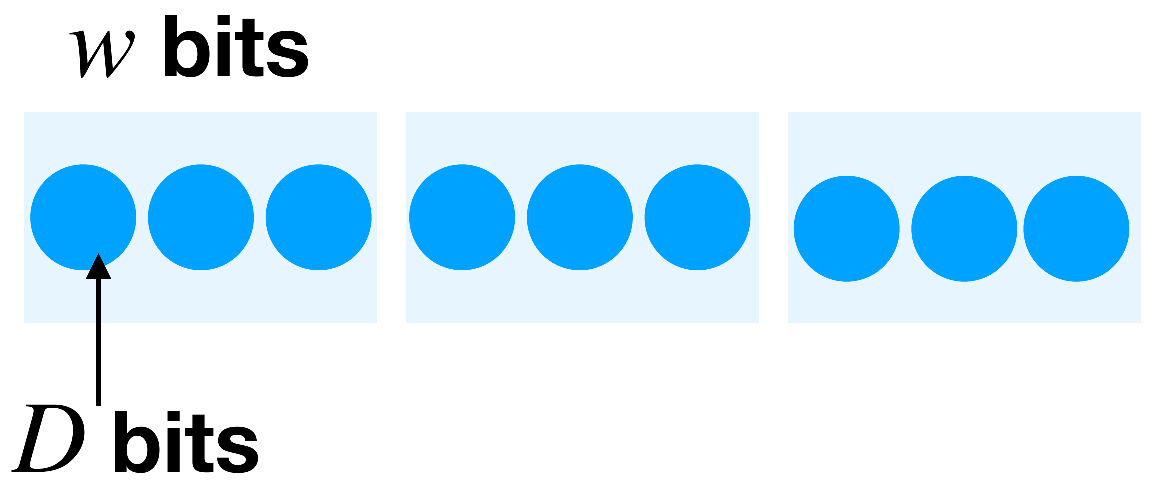


ShortHT

Looking inside the bins

Packing - The Idea

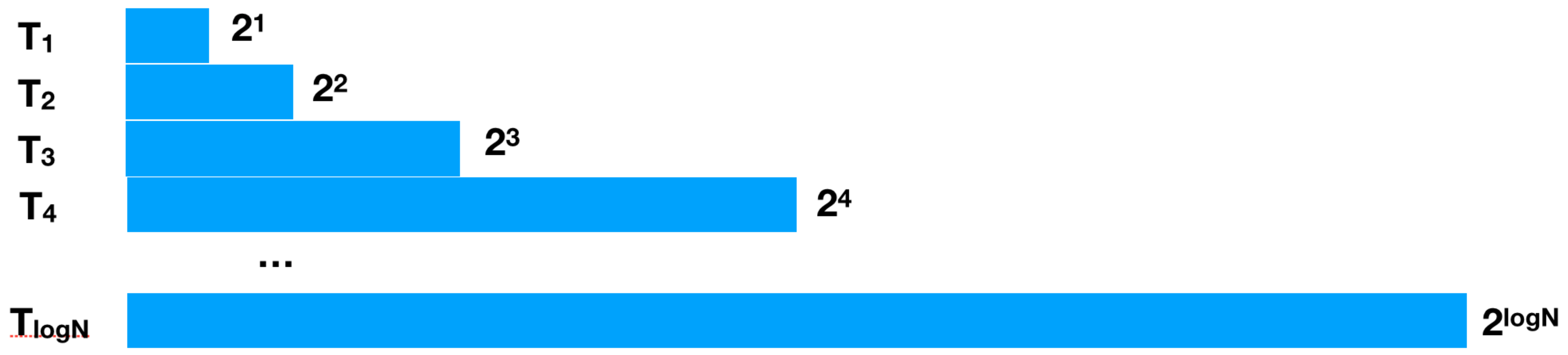
- Given n balls each of size D bits, word size w
- Classical oblivious sort costs $O(\lceil D/w \rceil \cdot n \cdot \log n)$
- What if $D \ll w$?
- **Packing**: put w/D balls in one memory word!



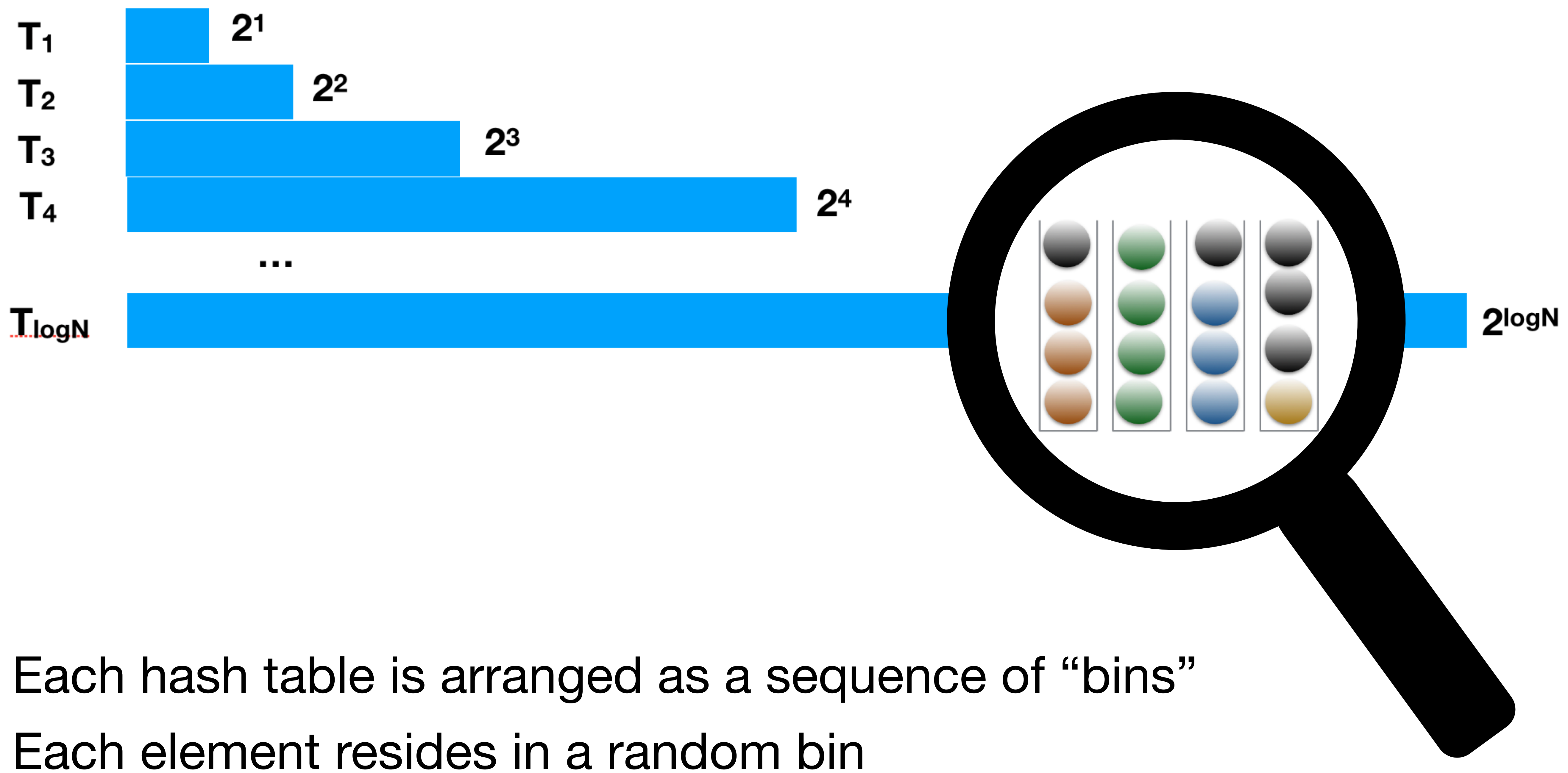
- Can sort in time $O(D/w \cdot n \cdot \log^2 n)$

- When n and D are small (say $n = w^4$ and $D = \log w$), we **can sort in linear time!** ($\frac{n \log^2 n}{w} \leq n$ vs. $n \cdot \log n$)

Where is it Being Used?



Where is it Being Used?



Each hash table is arranged as a sequence of "bins"

Each element resides in a random bin

The size of each bin is $n = \log^4 N$

Previously: build a structure on a bin using oblivious sort $n \log n$ -> $\log \log N$ overhead

We can remove it using the packing trick

From Amortized Complexity to Worst-Case Complexity

De-amortization of Ostrovsky and Shoup '97

We got a taste of $O(\log N)$ overhead — in amortized

Some operations require much longer - $O(N)$

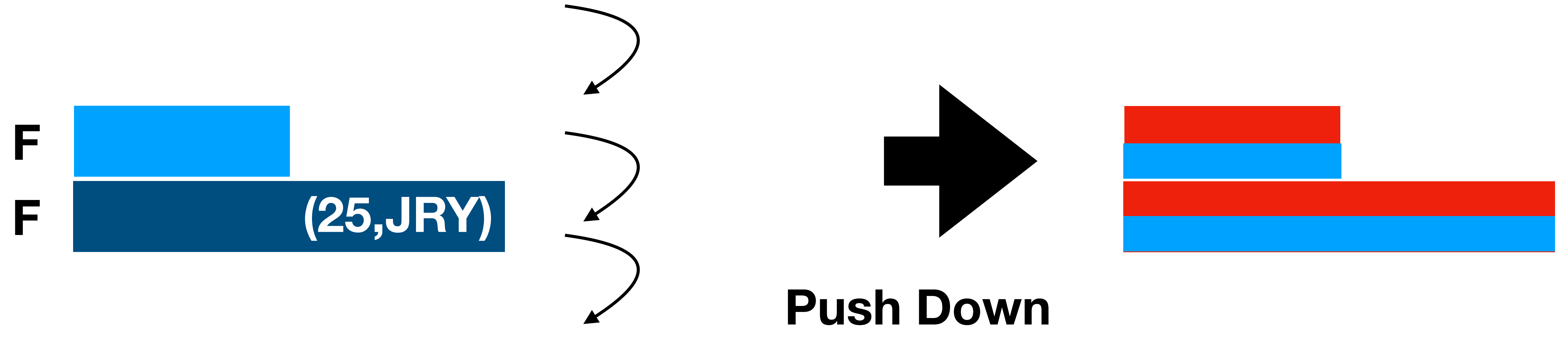
Can we get $O(\log N)$ in worse-case?

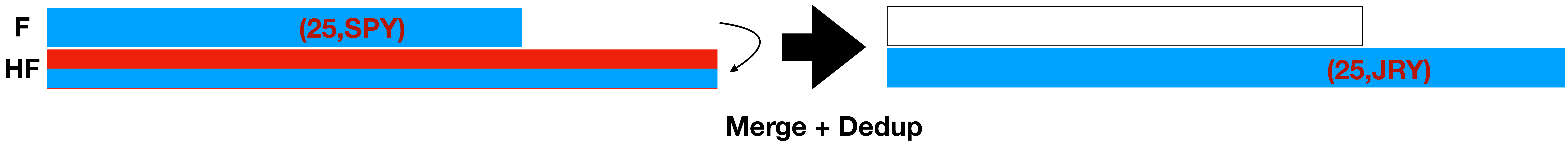
Classic de-amortization technique of hierarchical ORAM **is not compatible with OptORAMa and PanORAMa!**

De-amortization Friendly Rebuild

Instead of “full / empty” -> “full / half full”

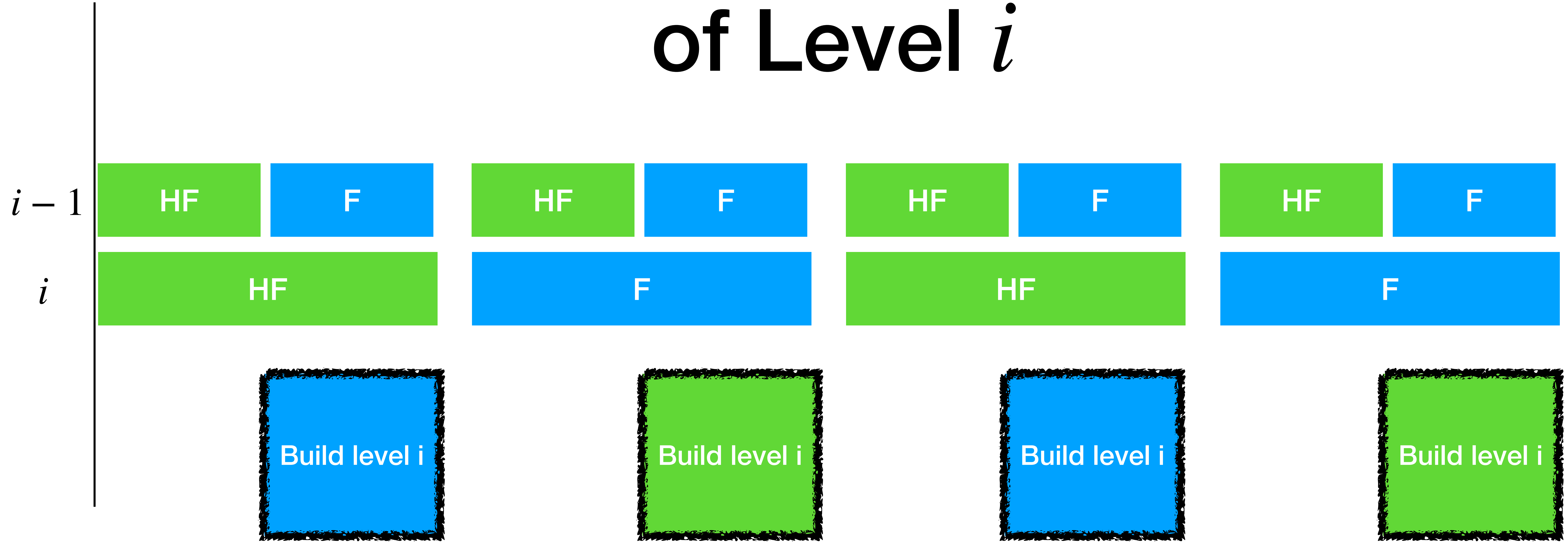
How Does it Help Us?





Easier to de-amortize: Looking at only two consecutive levels

De-amortizing Rebuild of Level i



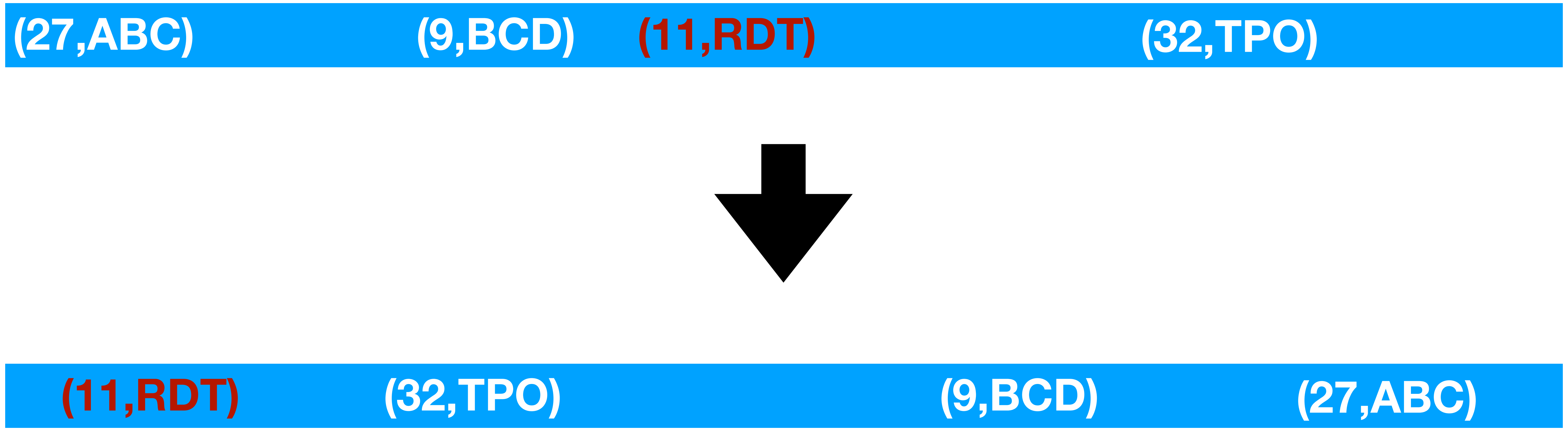
Randomness Reuse

(PanORAMa / OptORAMa)

(27,ABC) (9,BCD) (11,RDT) (32,TPO)

Randomness Reuse

(PanORAMa / OptORAMa)

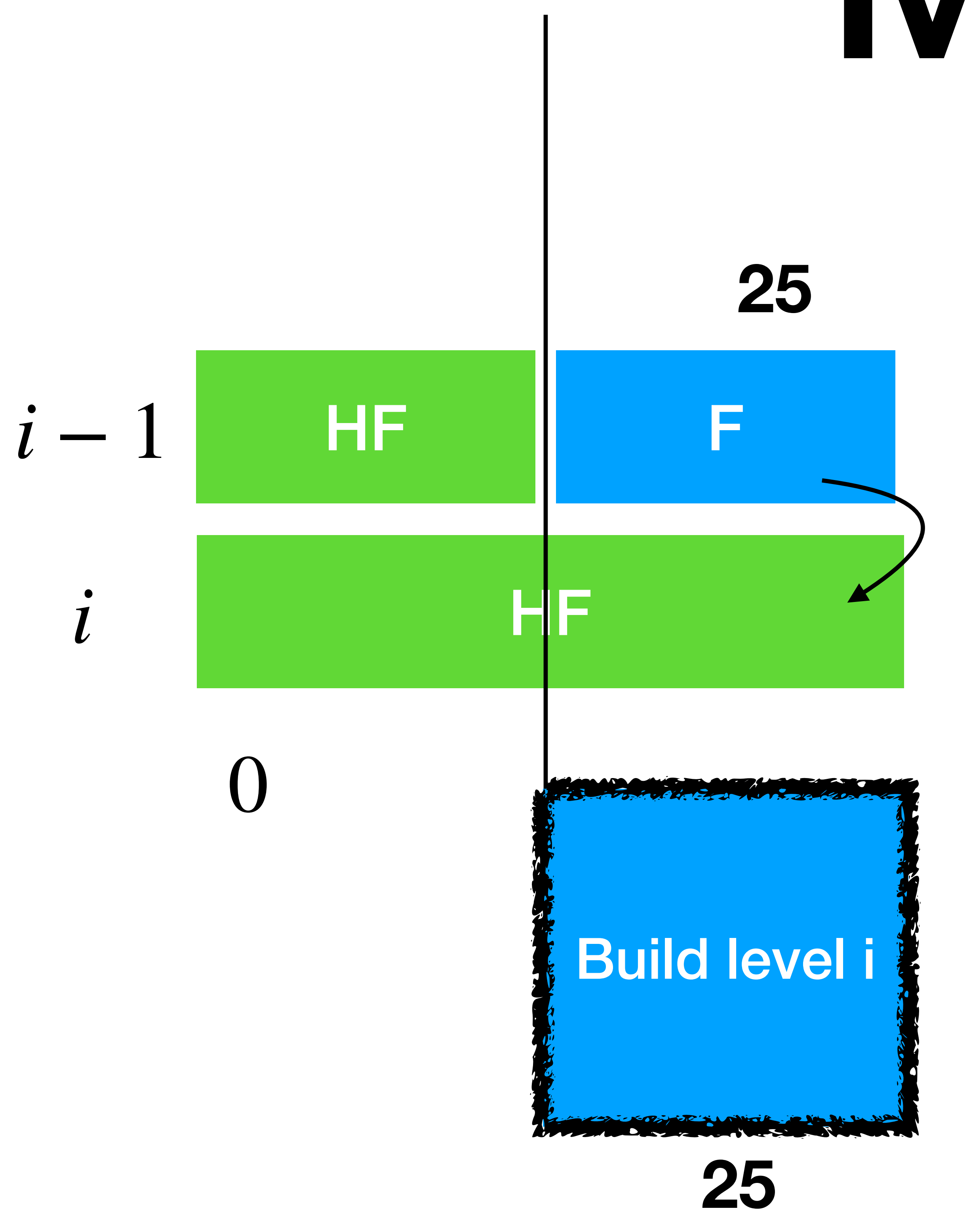


Elements that we did not touch are still randomly shuffled!!

PanORAMa and OptORAMa do not perform full **Rebuild** ->
Use the randomness from previous **Rebuild**

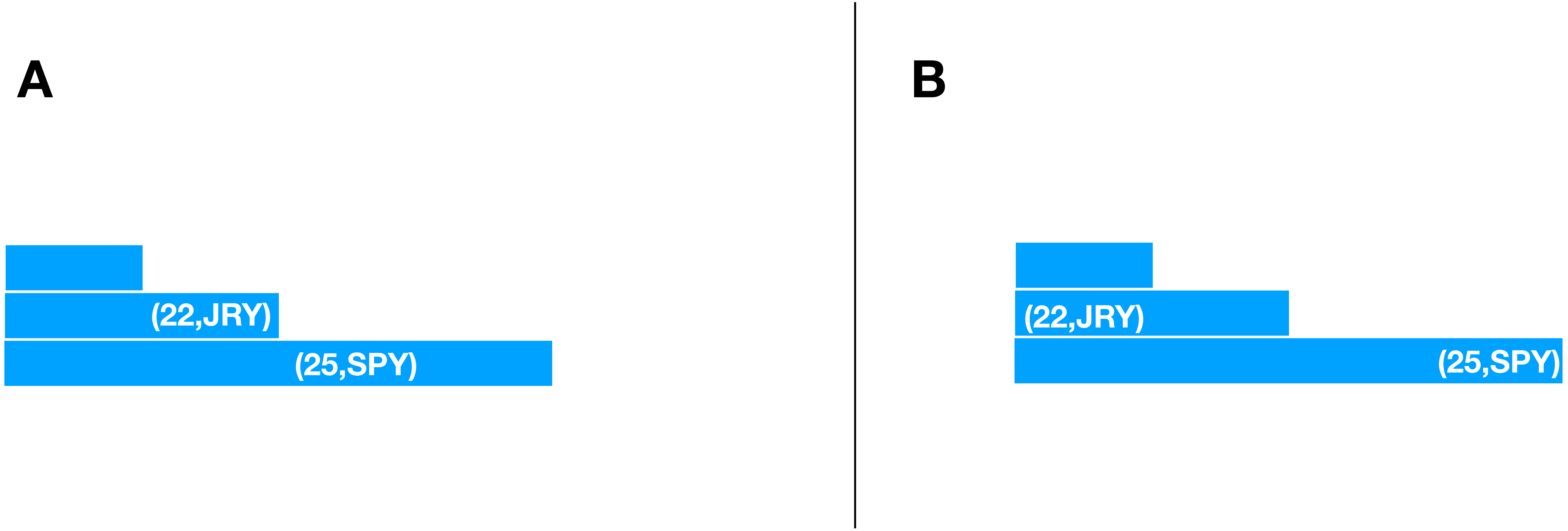
-> Reduced **Rebuild** from $O(n \log n)$ to $O(n)$ work

Main Challenge:



We might re-consume the randomness!

Main Idea

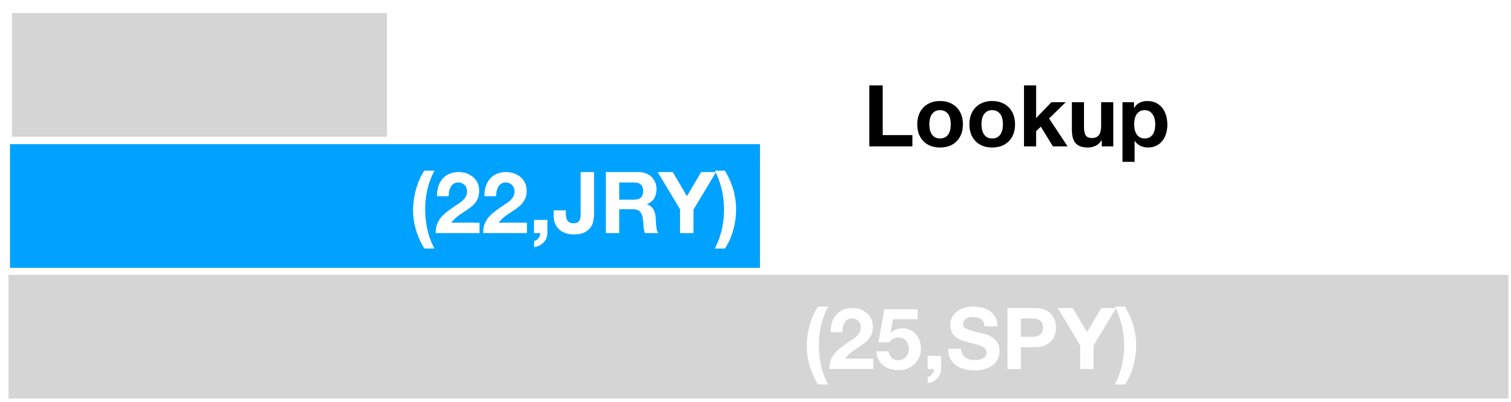


Two copies - same data in each level

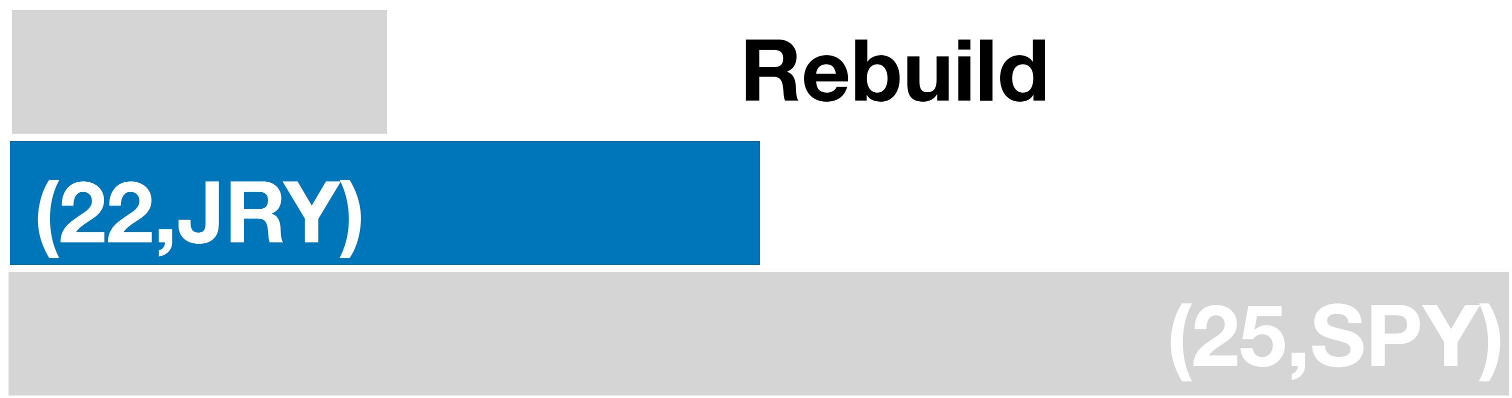
Each level has an active copy, and a copy that is being rebuilt

Main Idea

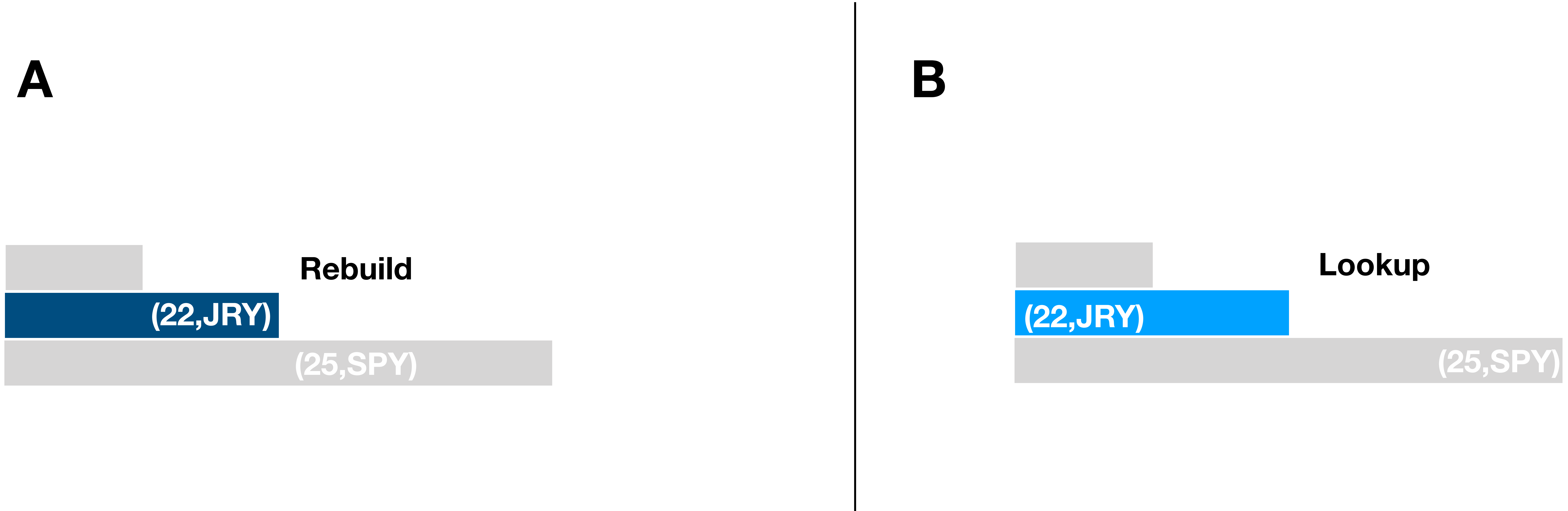
A



B



Main Idea



If the element is found -> put in both copies

Independent randomness!

See:

Asharov, Komargodski, Lin, Shi:

Oblivious RAM with Worst-Case Logarithmic Overhead, CRYPTO 2021

Center for Research in Applied Cryptography and Cyber Security

Conclusions

Lower bound: $\Omega(\log N)$

[GoldreichOstrovsky'96, LarsenNeilsen'18]

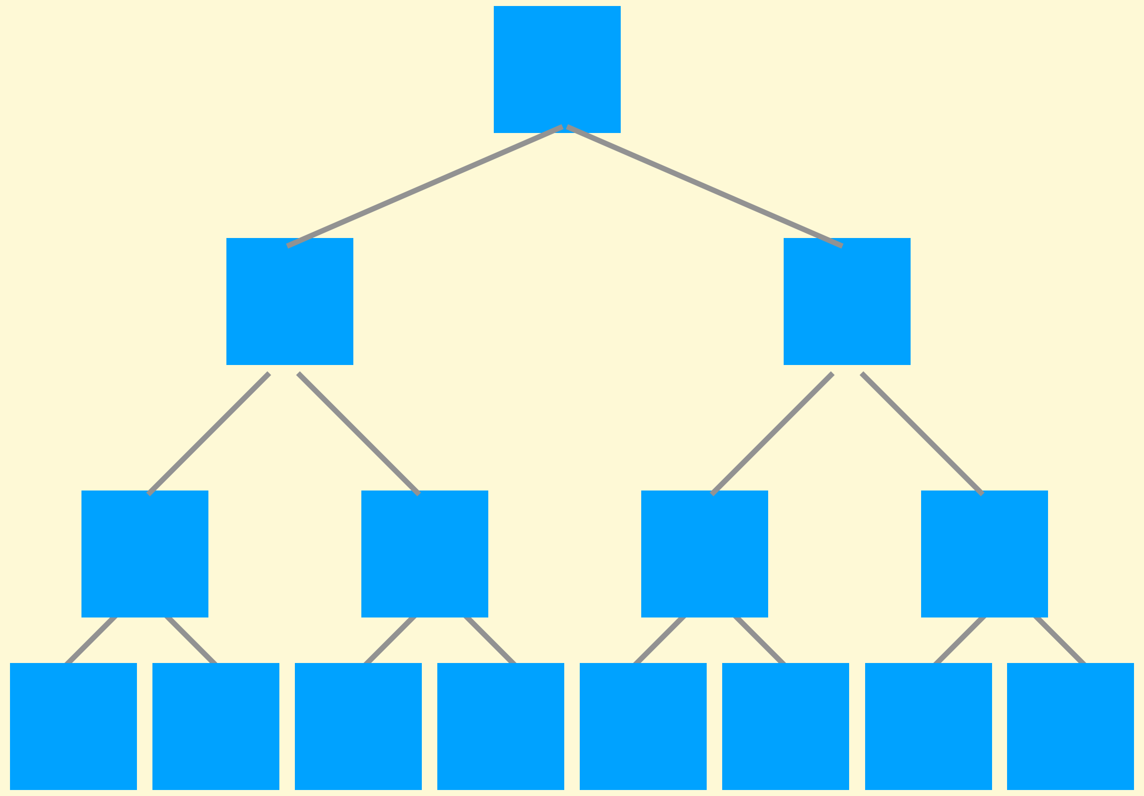
Hierarchical

[O90,GO96]

$$O(\log N)$$

Computational security

[OptORAMa'20]



Tree based ORAM

[Shi,Chan,Stefanov11]

$$O(\log^2 N)$$

Statistical security

[PathORAM,CircuitORAM]

References

Works mentioned in Part III

Goldreich, Ostrovsky:
Software Protection and Simulation on Oblivious RAM, JACM 1996

Ostrovsky, Shoup:
Private Information Storage, STOC 1997

Goodrich and Mitzenmacher:
Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation, ICALP 2011

Kushilevitz, Lu, Ostrovsky:
On the (In)Security of Hash-Based Oblivious RAM and a New Balancing Scheme, SODA 2012

Patel, Persiano, Raykova, Yeo:
PanORAMa: Oblivious RAM with logarithmic Overhead, FOCS 2018

Asharov, Komargodski, Lin, Nayak, Peserico, Shi:
OptORAMa: Optimal Oblivious RAM, EUROCRYPT 2020

Asharov, Komargodski, Lin, Shi:
Oblivious RAM with Worst-Case Logarithmic Overhead, CRYPTO 2021

Thank You!