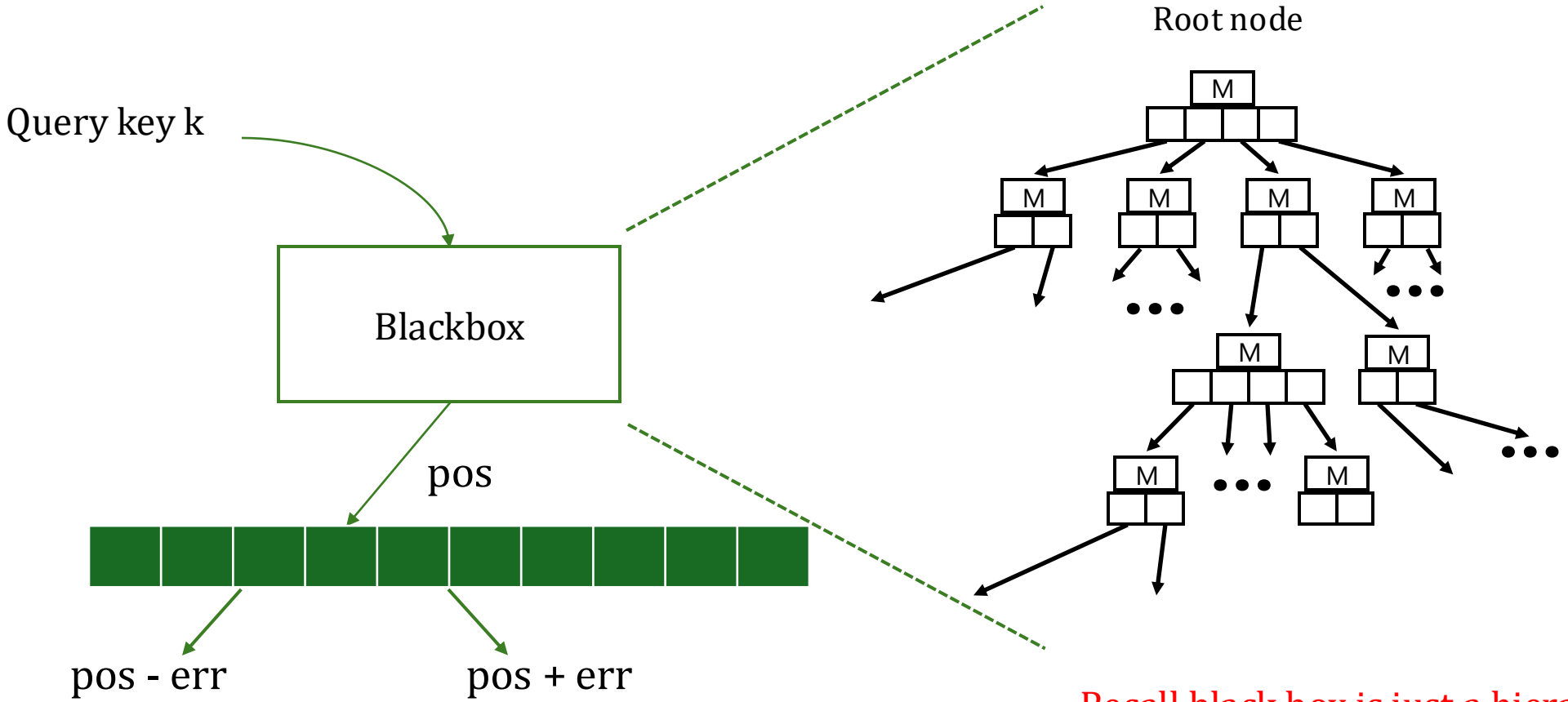
Succinct Position-Value Mapping for Learned Indexes Using Wavelet Trees

Anwesha Saha

BOSTON UNIVERSITY

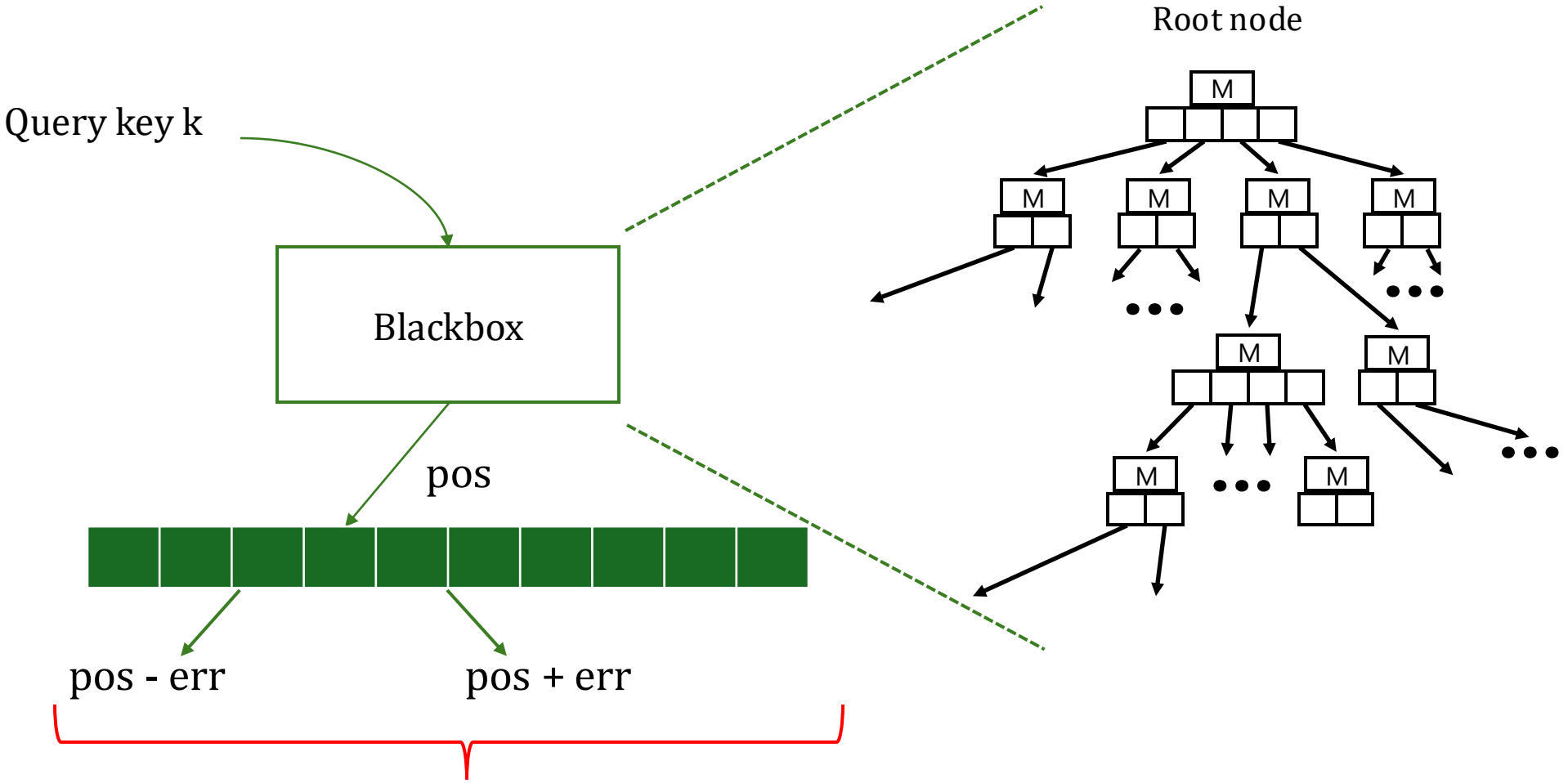
DiSC lab

Learned Indexes

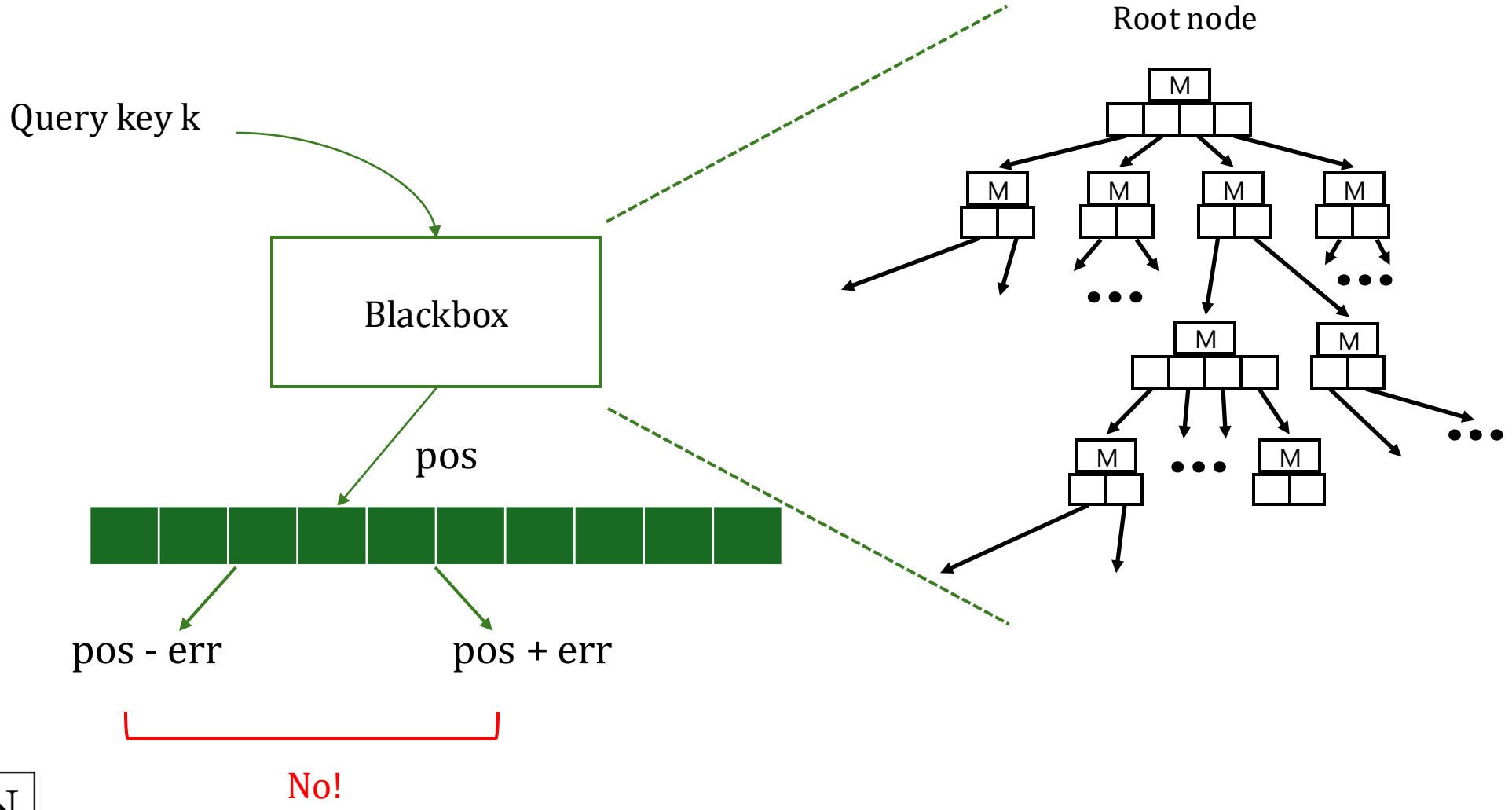


Recall black box is just a hierarchy of simple regression models

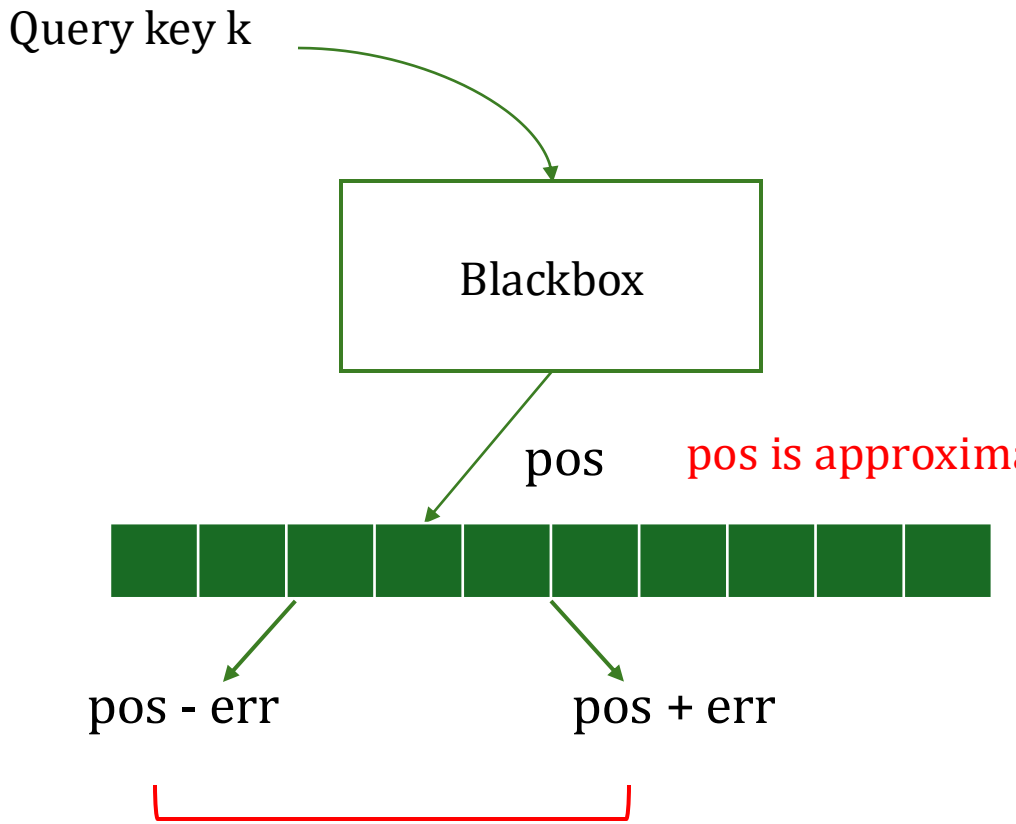
Limitations?



Are they Precise?

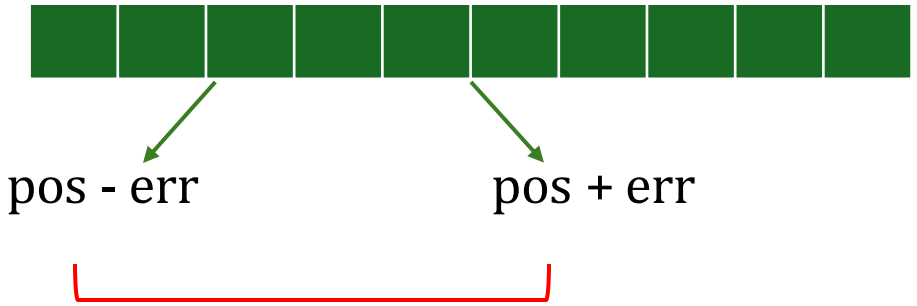


Are Learned Indexes Precise?



pos is approximate → within a range of 32 positions

Are Learned Indexes Precise?

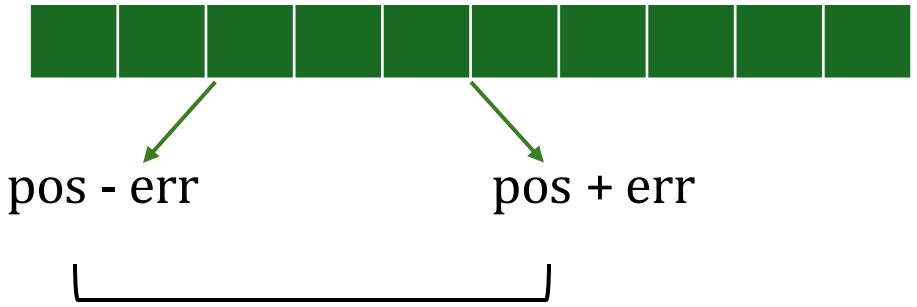


pos is approximate → within a range of 32 positions

↓

Perform binary search within this to find exact value

Are Learned Indexes Precise?

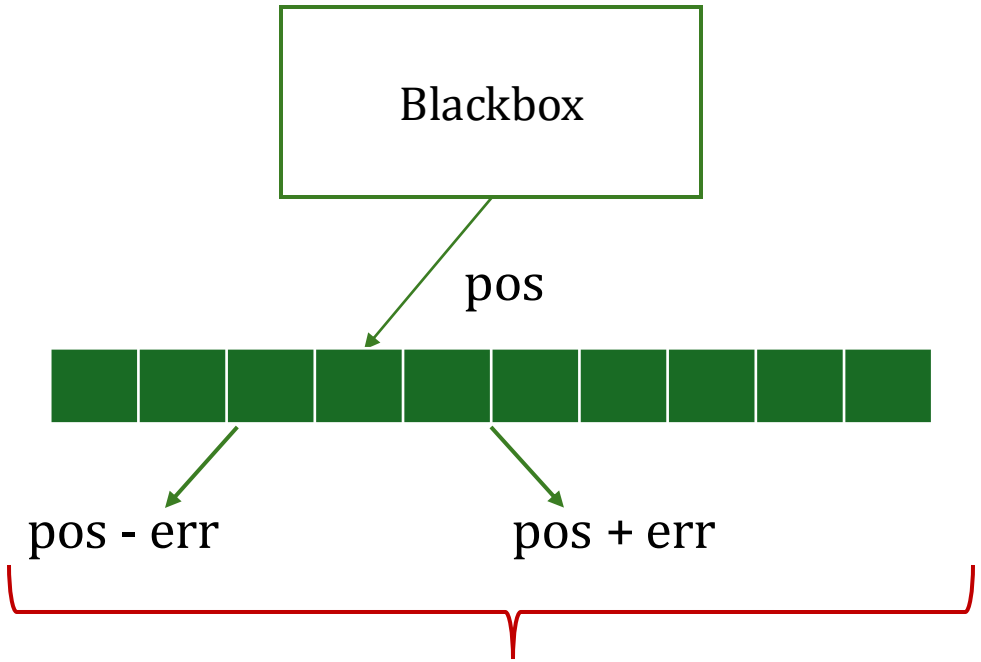


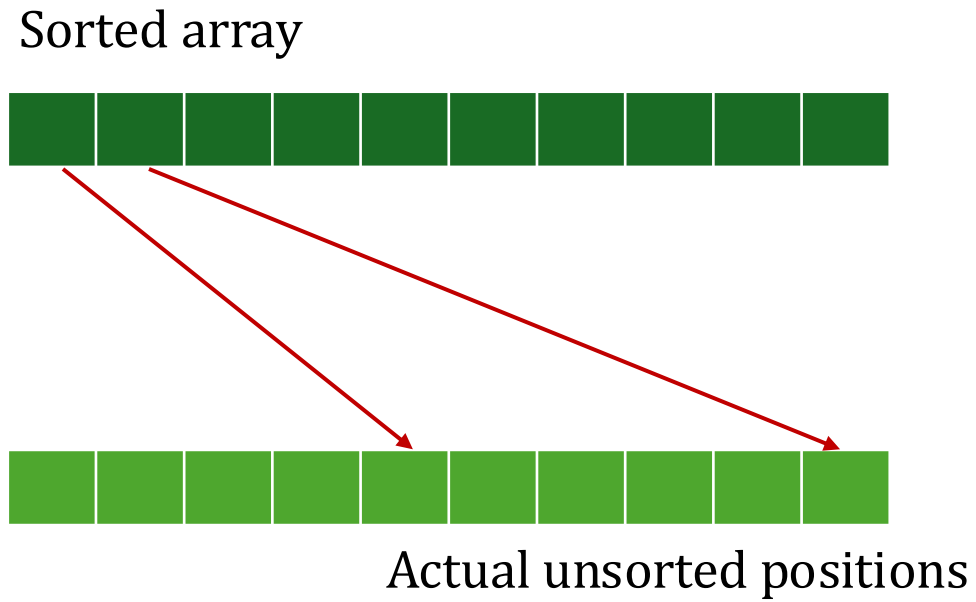
pos is approximate → within a range of 32 positions

Perform binary search within this to find exact value

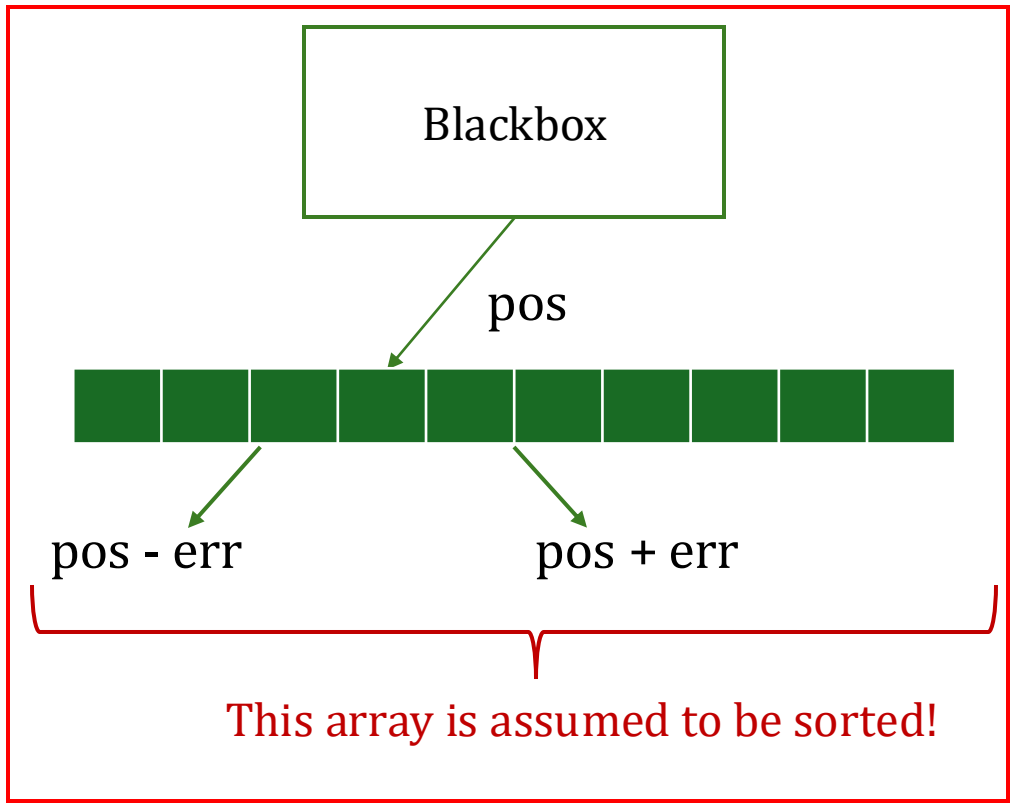
This implies $\log_2(32) = 5$ extra search cost

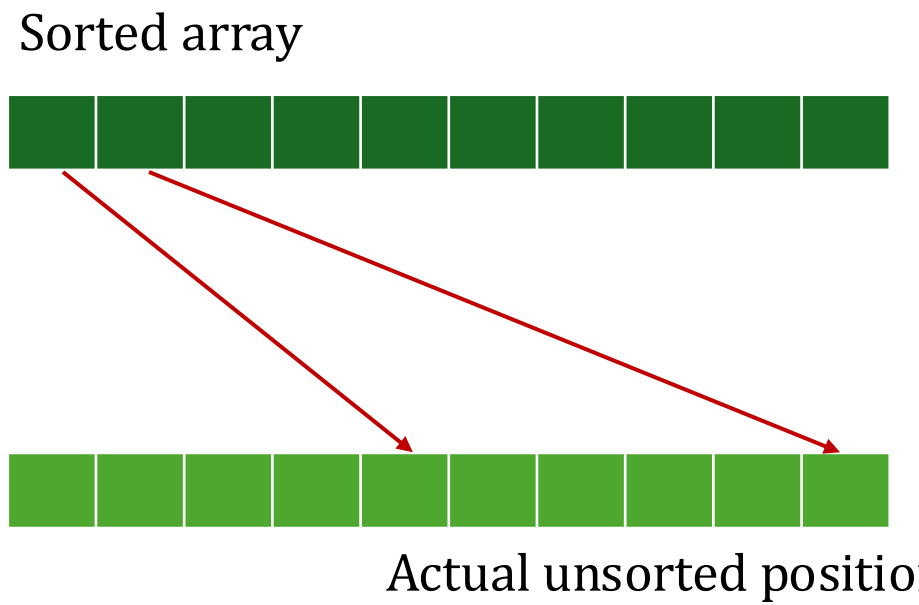
Proposed Solution





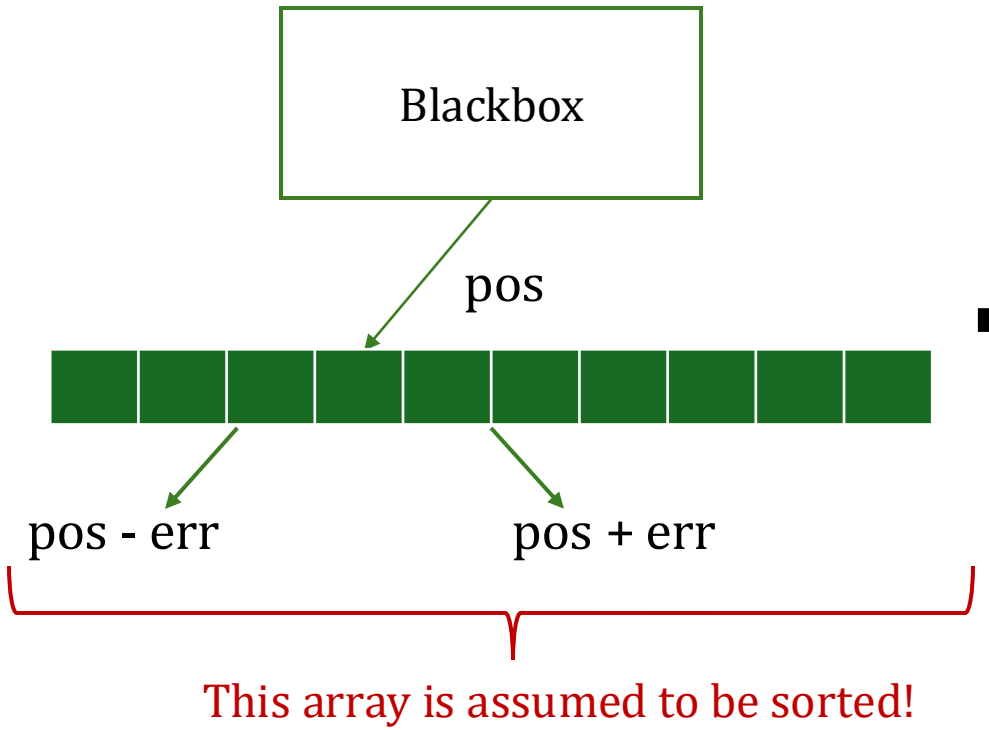
Proposed Solution

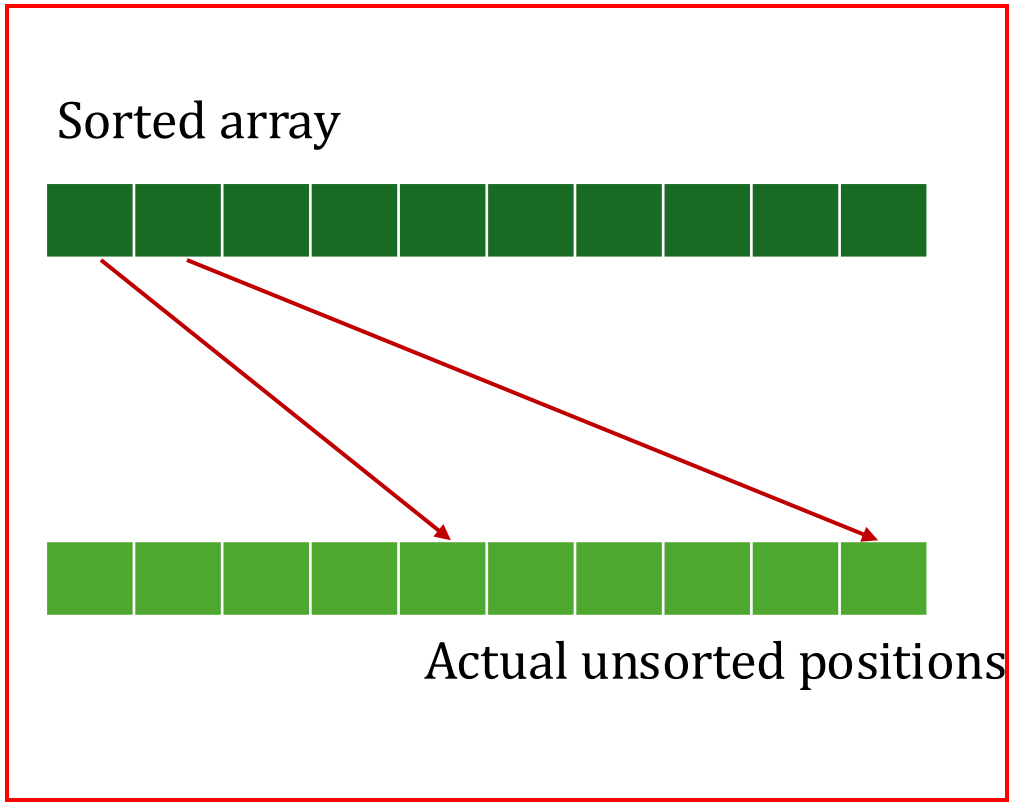




Learned Index

What are we trying to find?



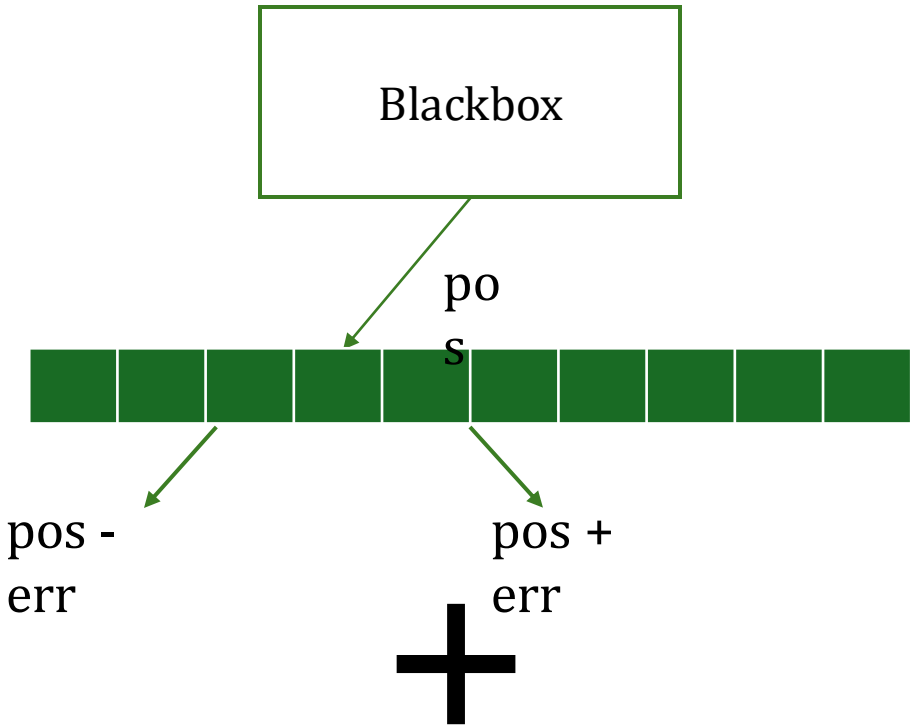


Mapping Scheme

BOSTON UNIVERSITY

DiSC lab

Constraints

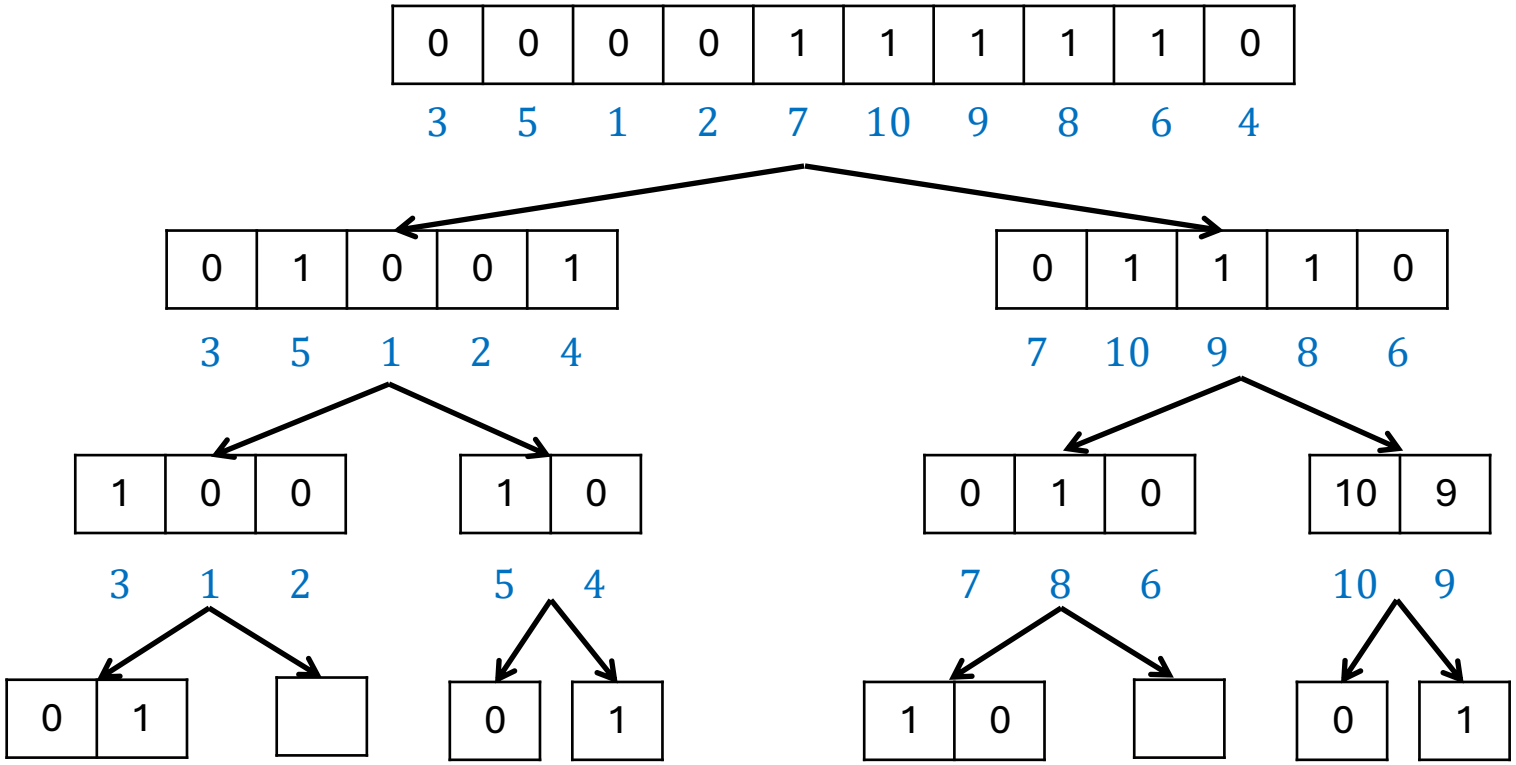


$\leq$

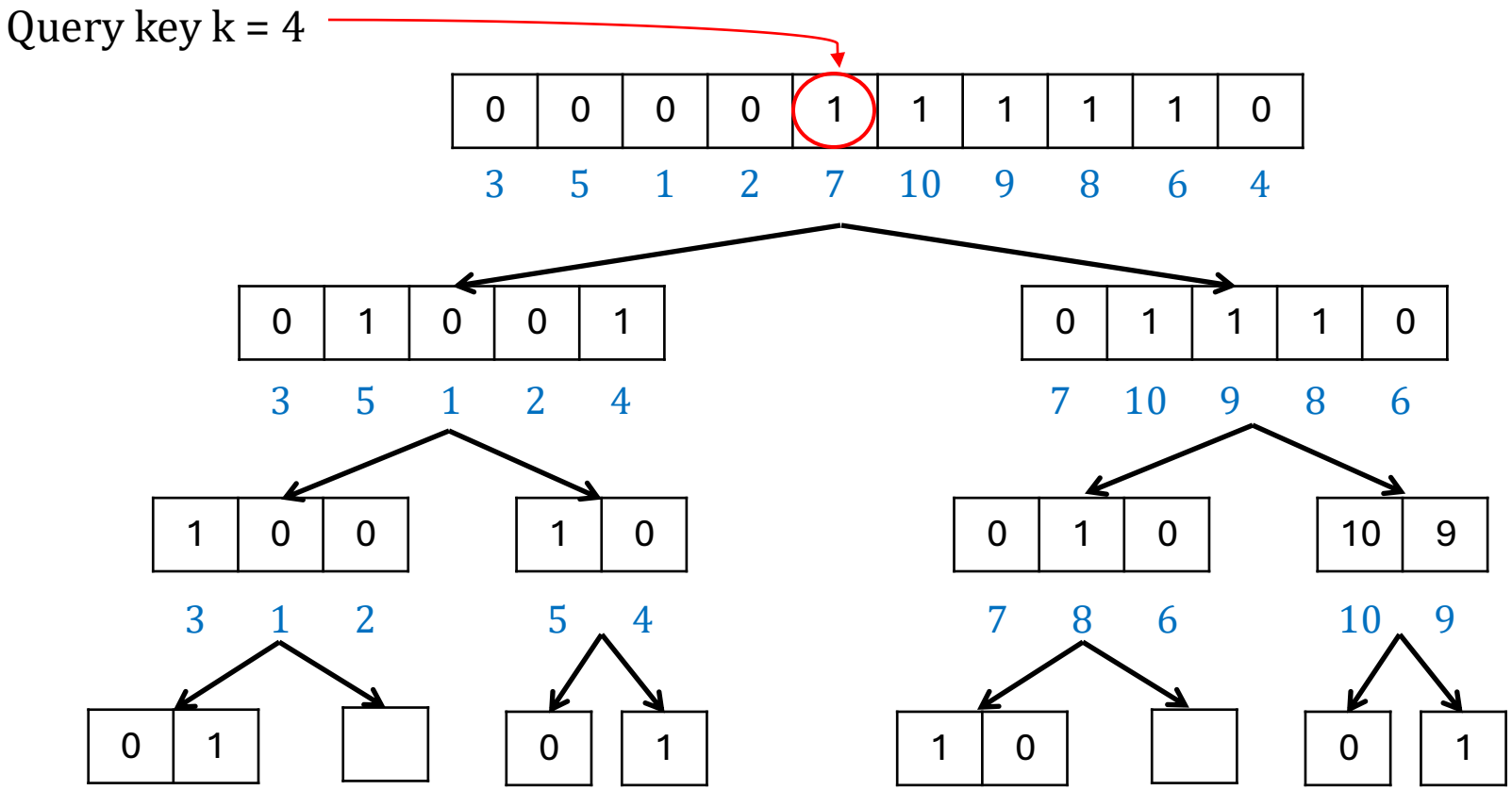
B^+ Tree Cost

5 times
(Mapping Scheme)

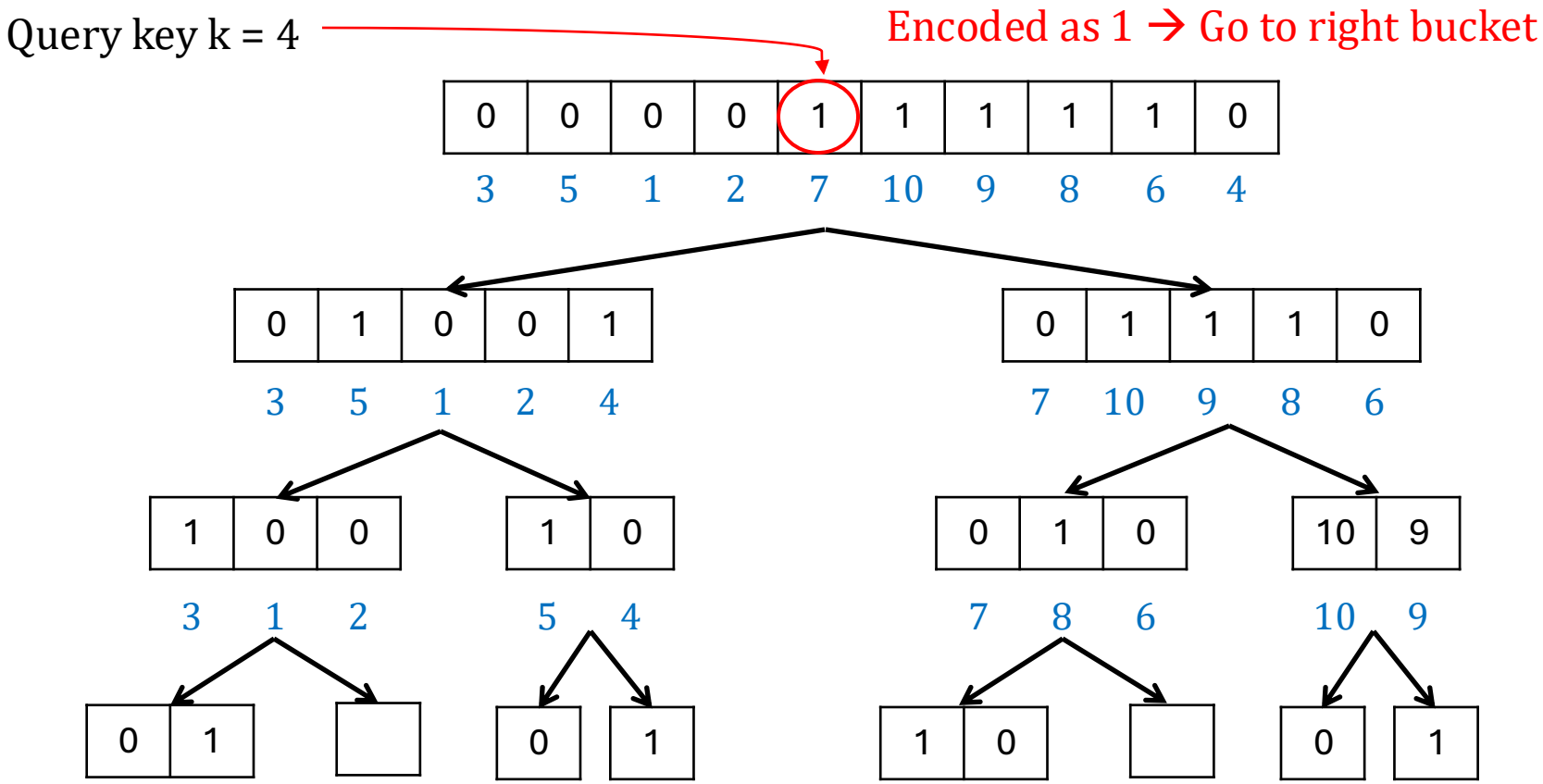
Wavelet Tree



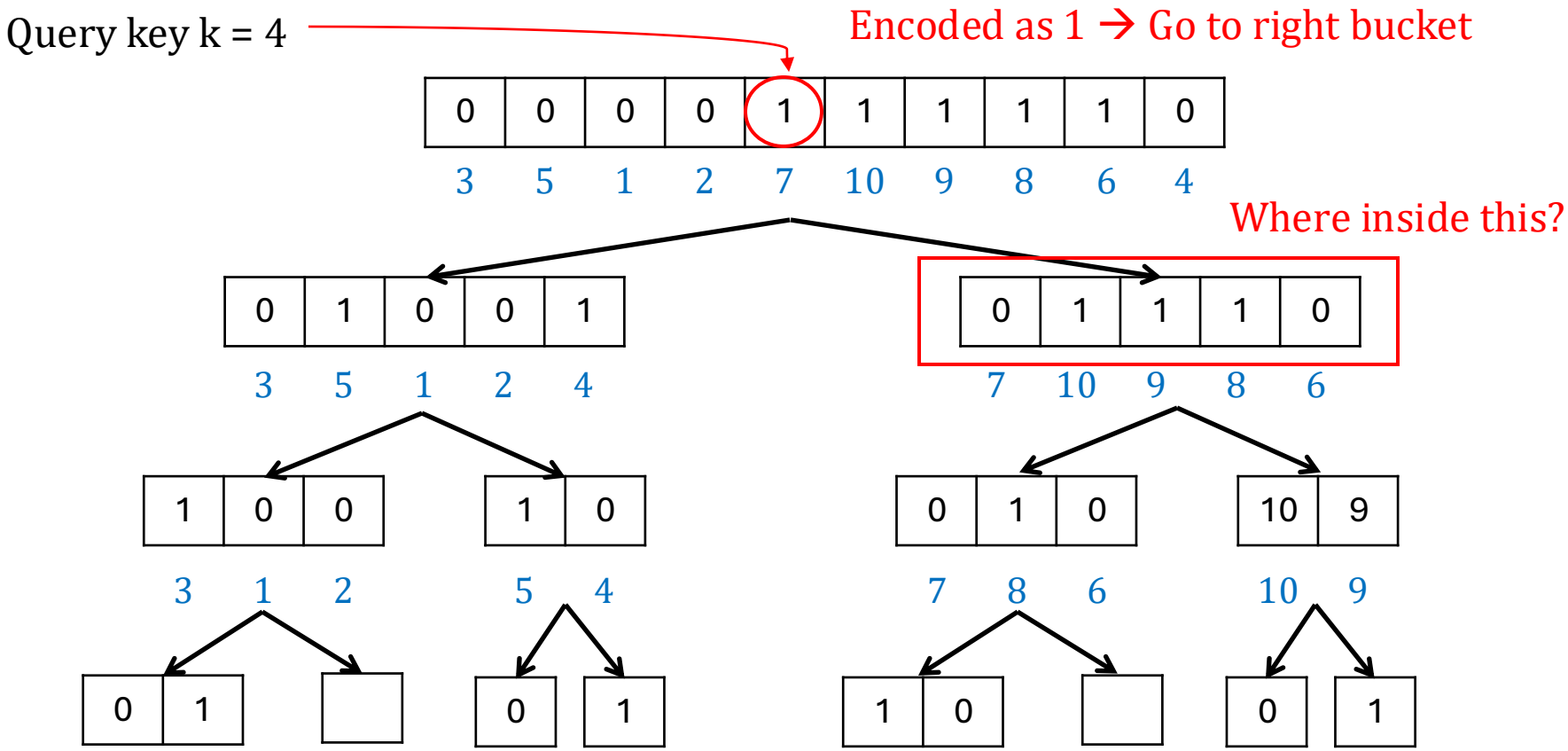
Find Element in a Wavelet Tree



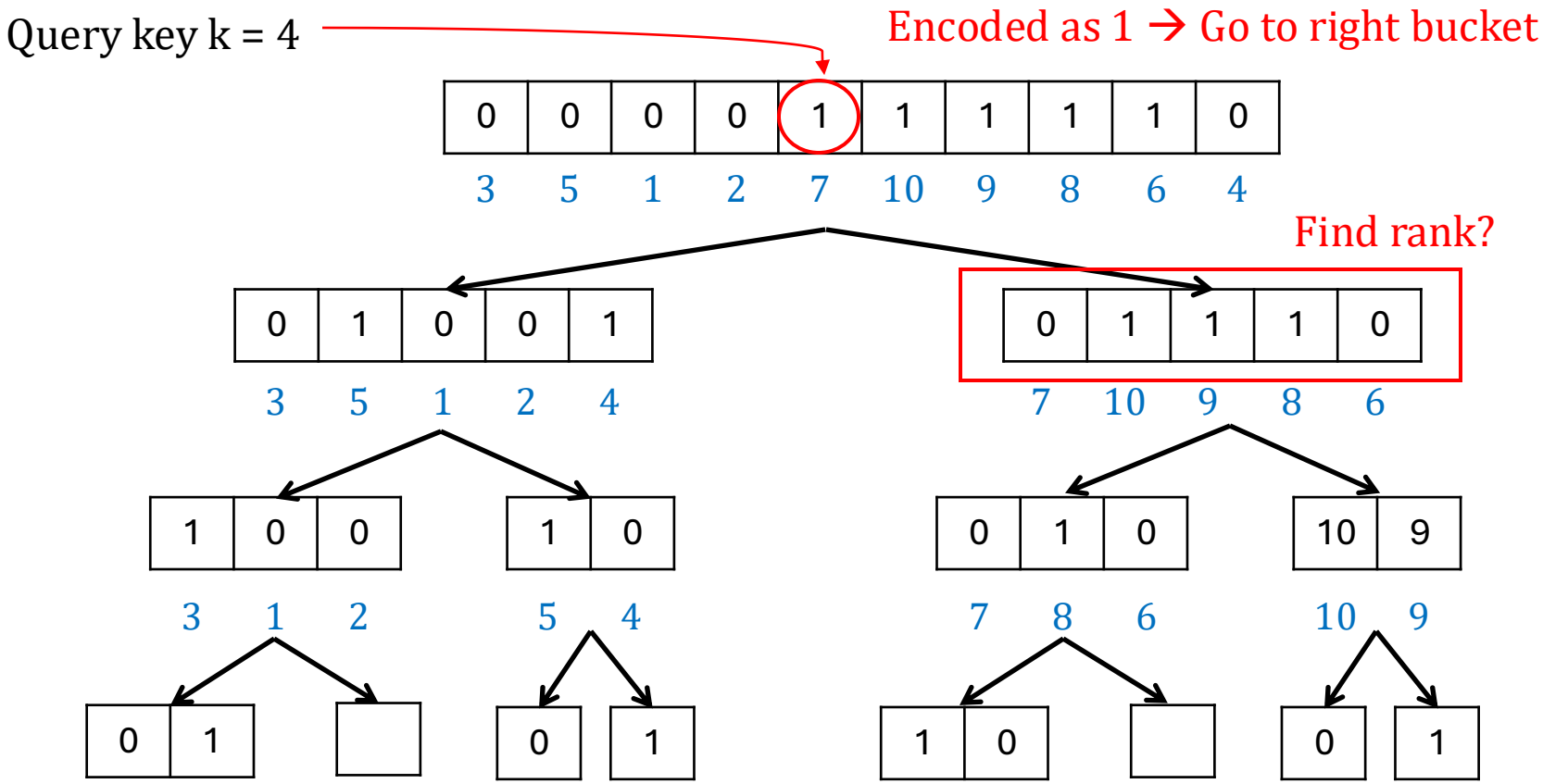
Find Element in a Wavelet Tree



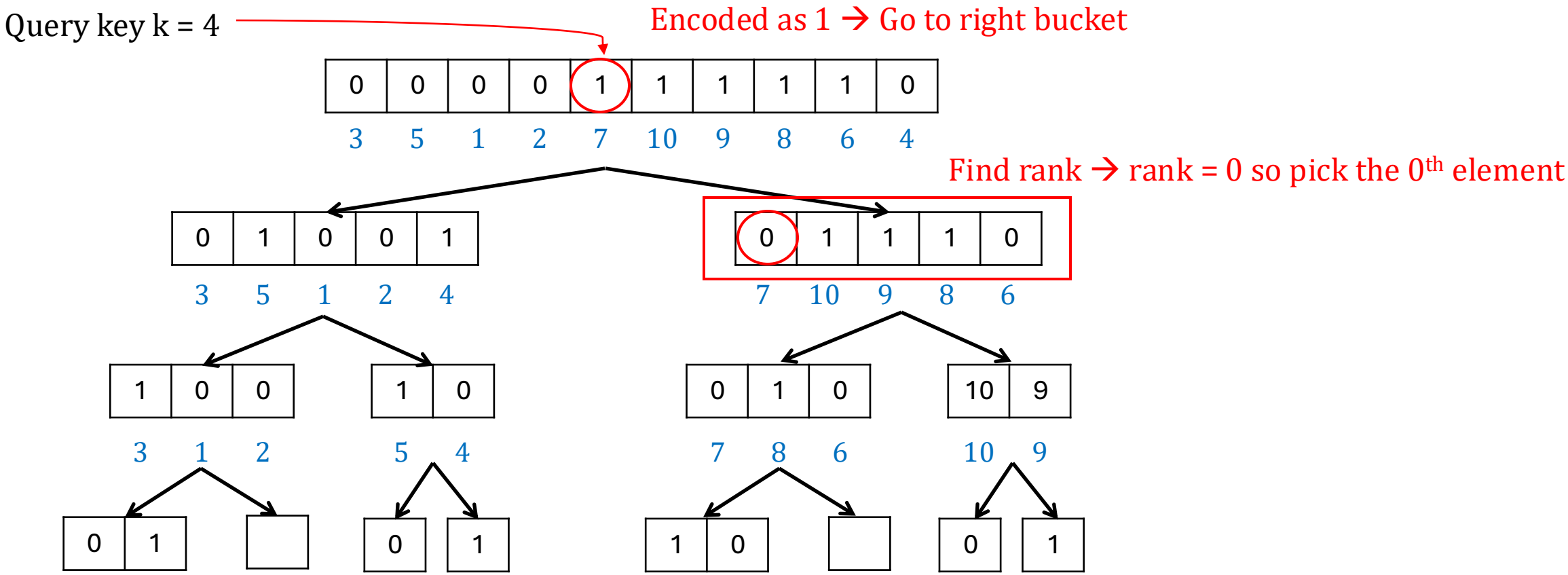
Find Element in a Wavelet Tree



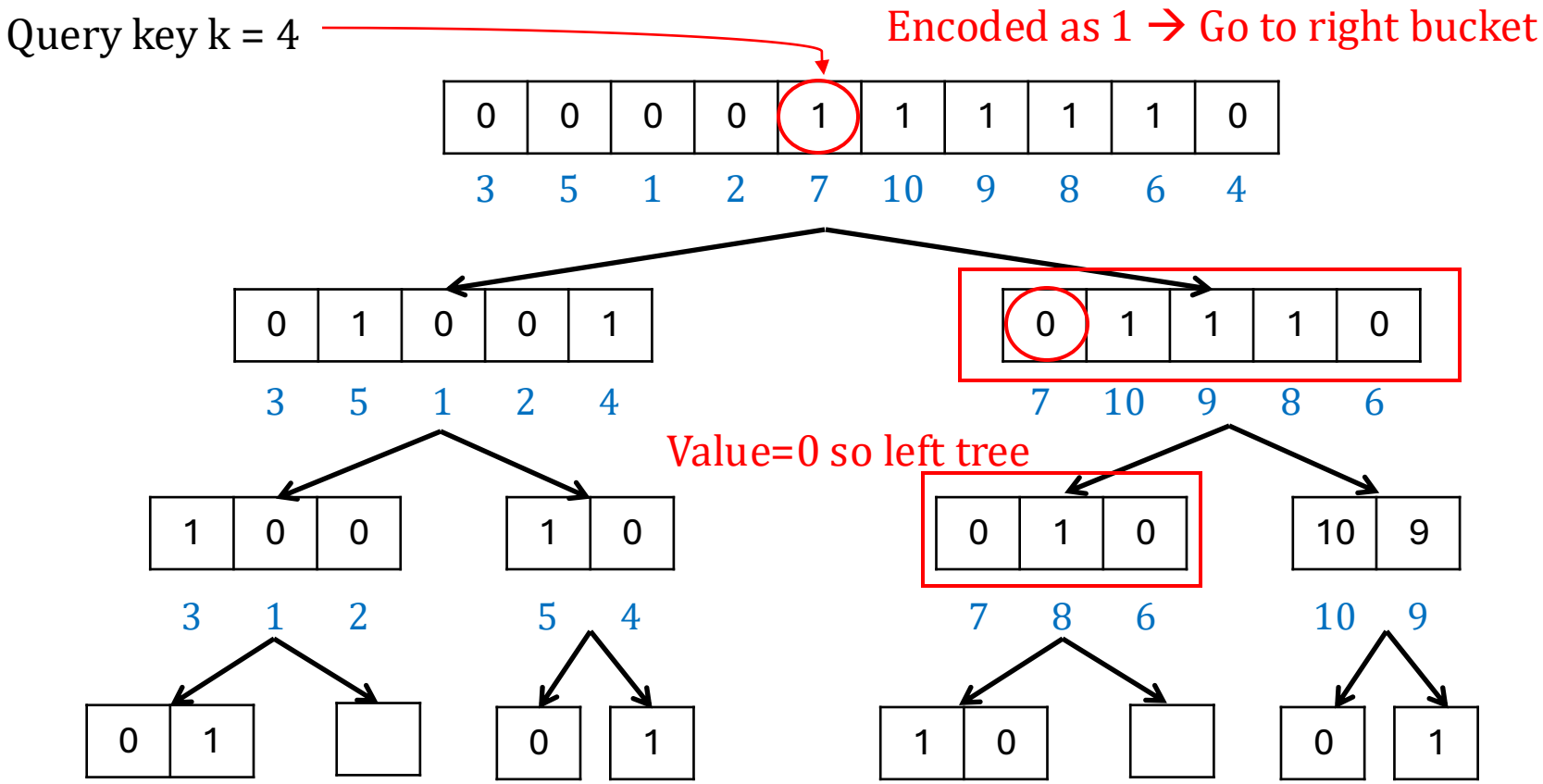
Find Element in a Wavelet Tree



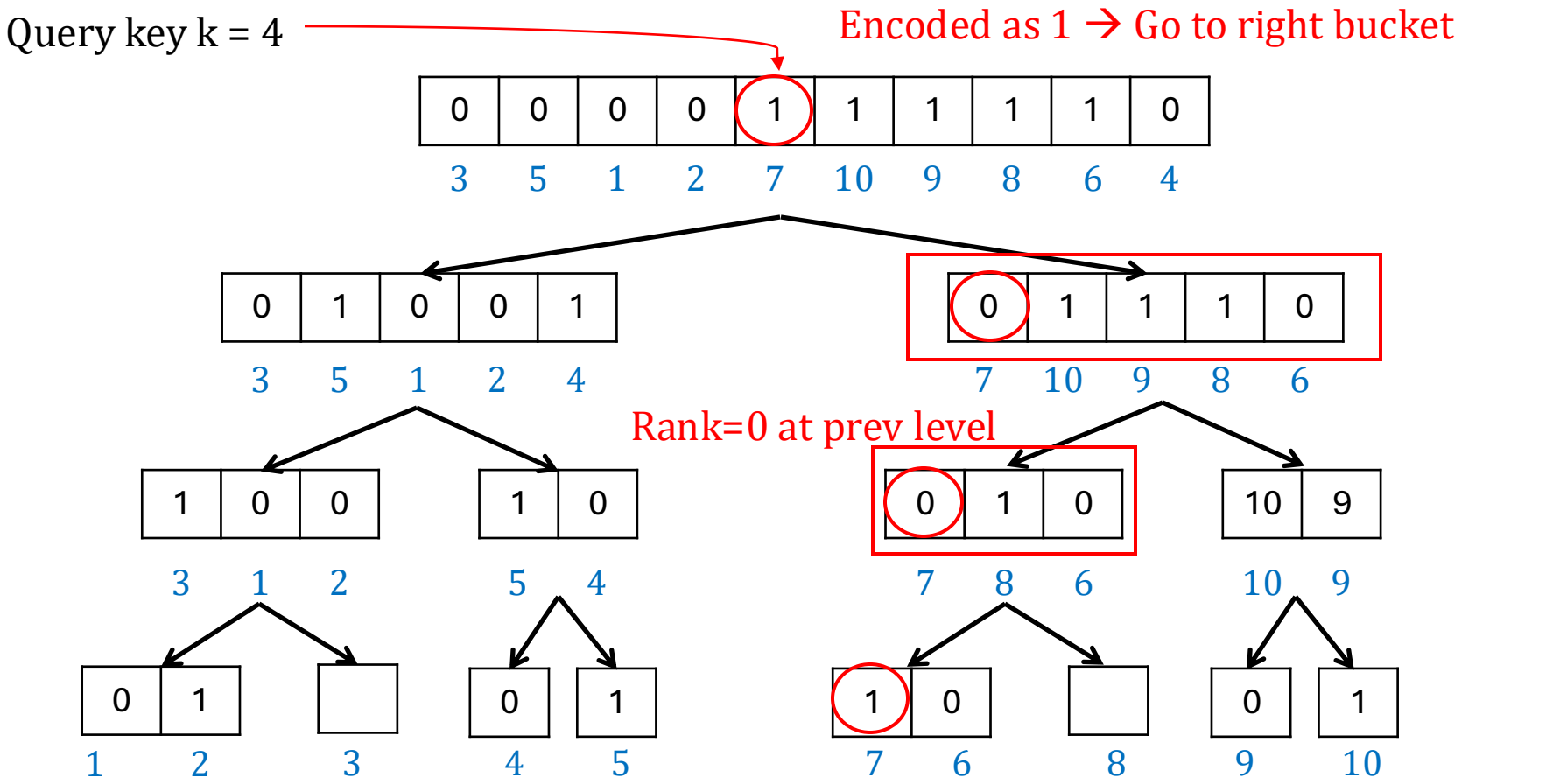
Find Element in a Wavelet Tree



Find Element in a Wavelet Tree

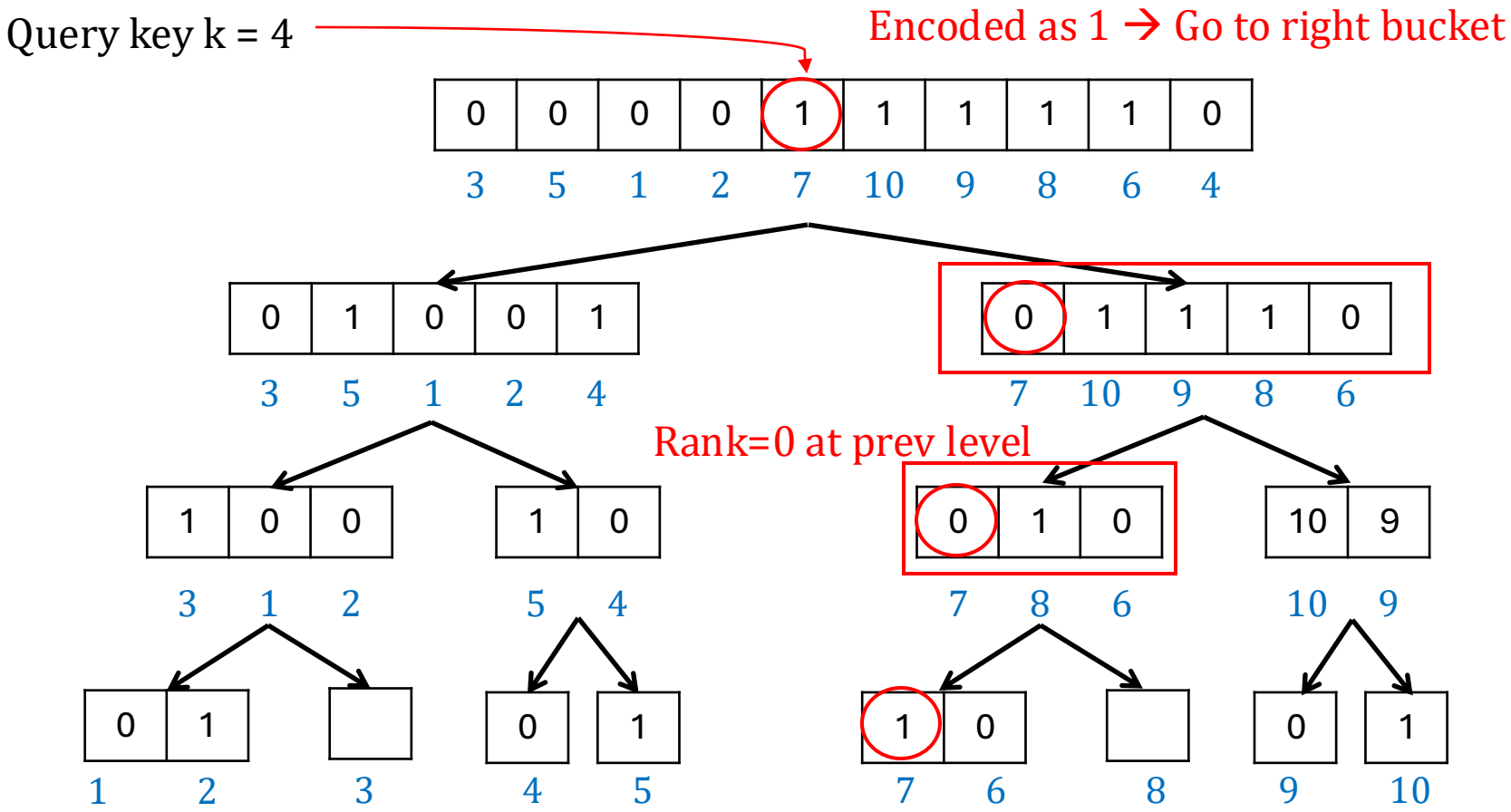


Find Element in a Wavelet Tree



1	2	3	4	5	6	7	8	9	10

Find Element in a Wavelet Tree

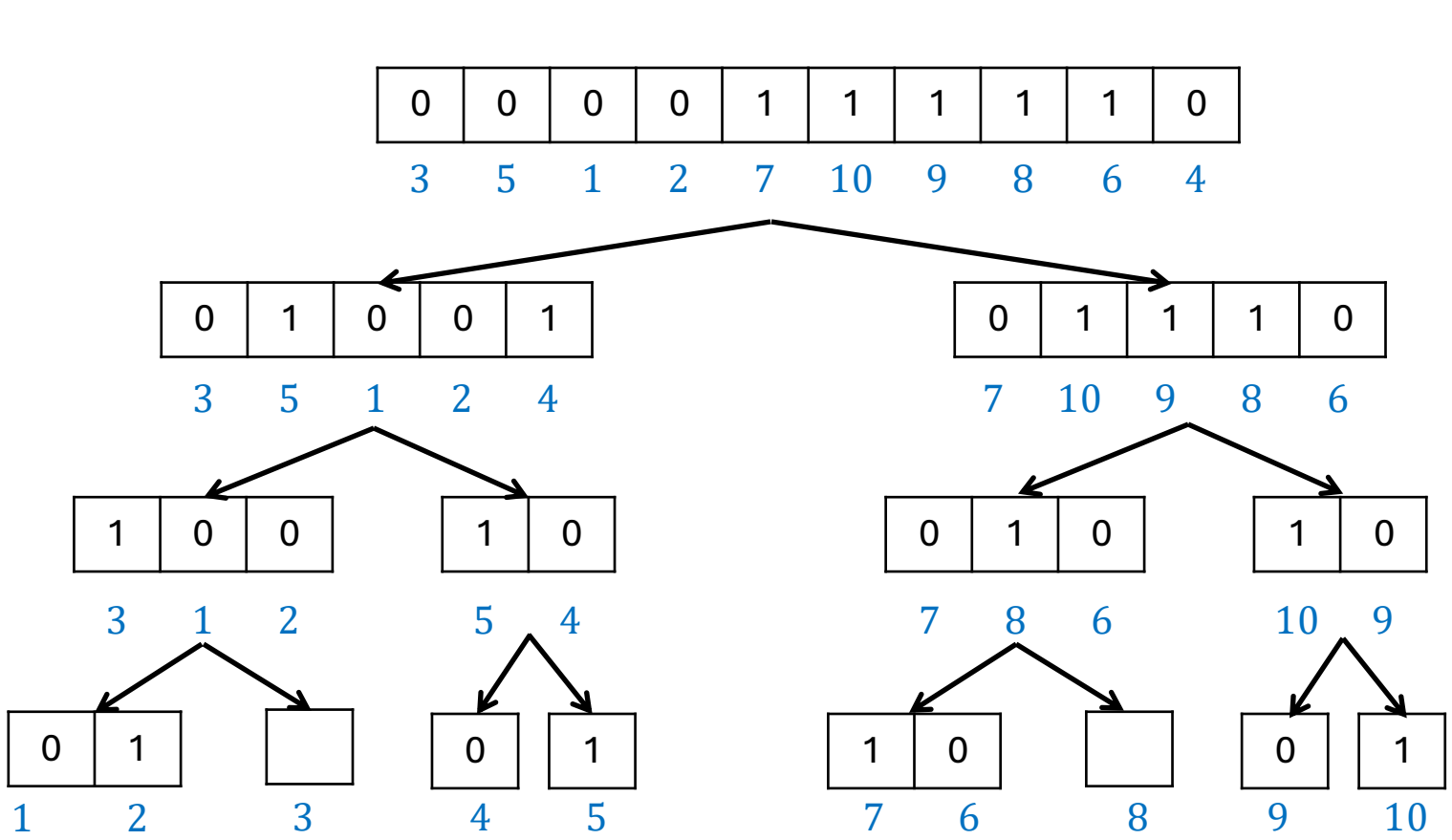


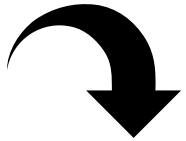
Notice implicit ordering!

BOSTON UNIVERSITY

DiSC lab

Representing the Tree as Flat Arrays





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

BOSTON UNIVERSITY

DiSC lab

Wavelet Tree Optimizations

we use a **32-bit** integer

0101010101010101
0101010101010101

Wavelet Tree Optimizations

we use a **32-bit** integer

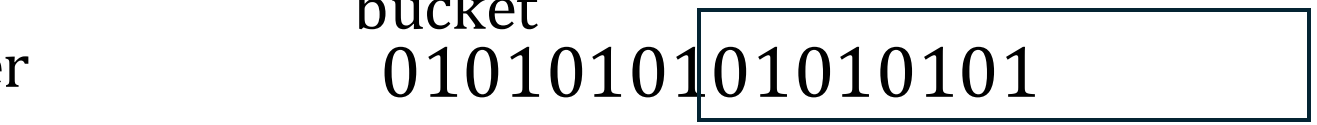
first **16-bits** determine the bucket

0101010101010101
0101010101010101

Wavelet Tree Optimizations

we use a **32-bit** integer

first **16-bits** determine the bucket



next **16-bits** determine value within the bucket

Wavelet Tree Optimizations

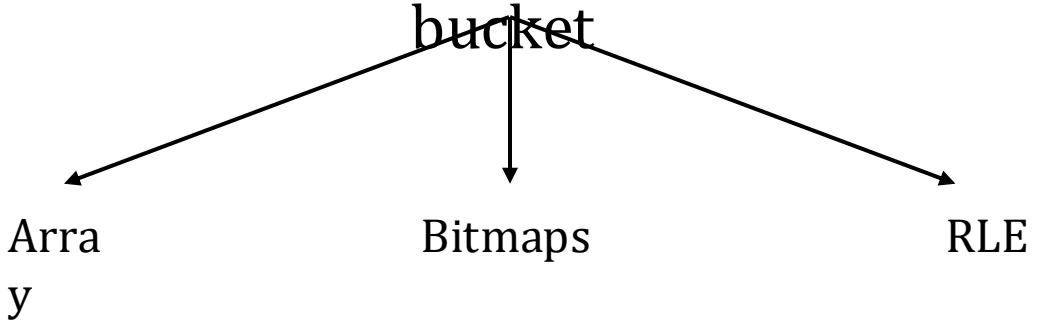
we use a **32-bit** integer

first **16-bits** determine the bucket

0101010101010101

0101010101010101

next **16-bits** determine value within the bucket



Wavelet Tree Shortcomings

Cost of access $\propto$ Height of tree

↓

2-way tree is extremely deep

↓

B^+ Tree usually has 256 branches

Wavelet Tree Shortcomings

Cost of access $\propto$ Height of tree

↓

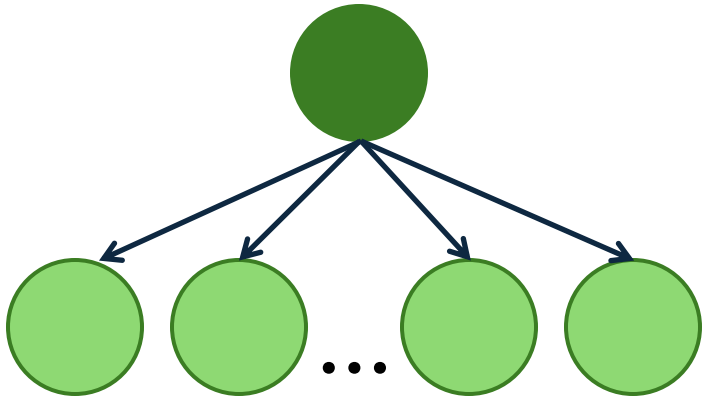
2-way tree is extremely deep

↓

B^+ Tree usually has 256 branches

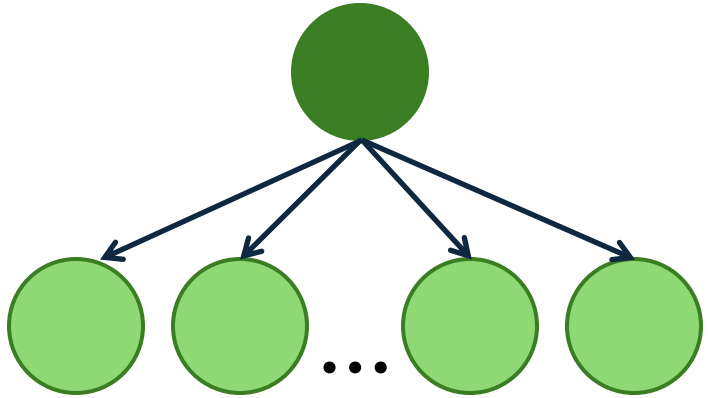
What about a m-way Tree?

Increasing Fanout?



m-ary
tree

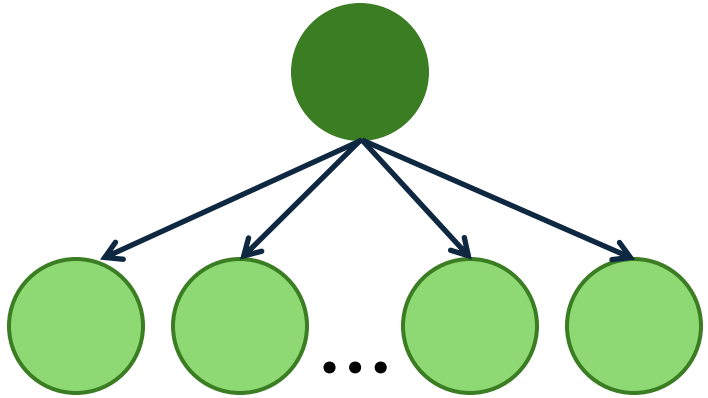
Increasing Fanout?



m-ary
tree

1. Can we continue to use **bits**?

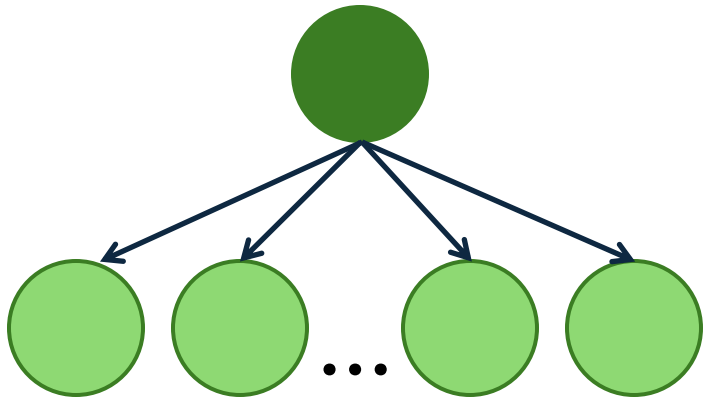
Increasing Fanout?



m-ary
tree

1. Can we continue to use **bits**?
2. How do we maintain **rank**?

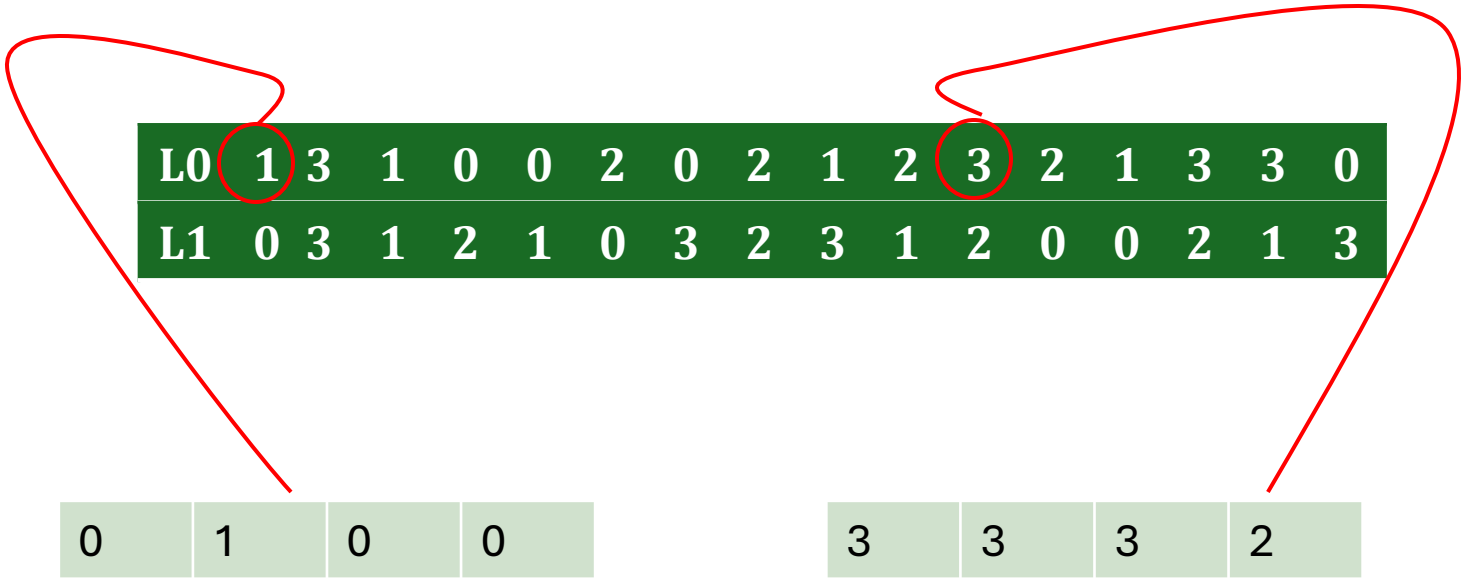
Increasing Fanout?



m-ary
tree

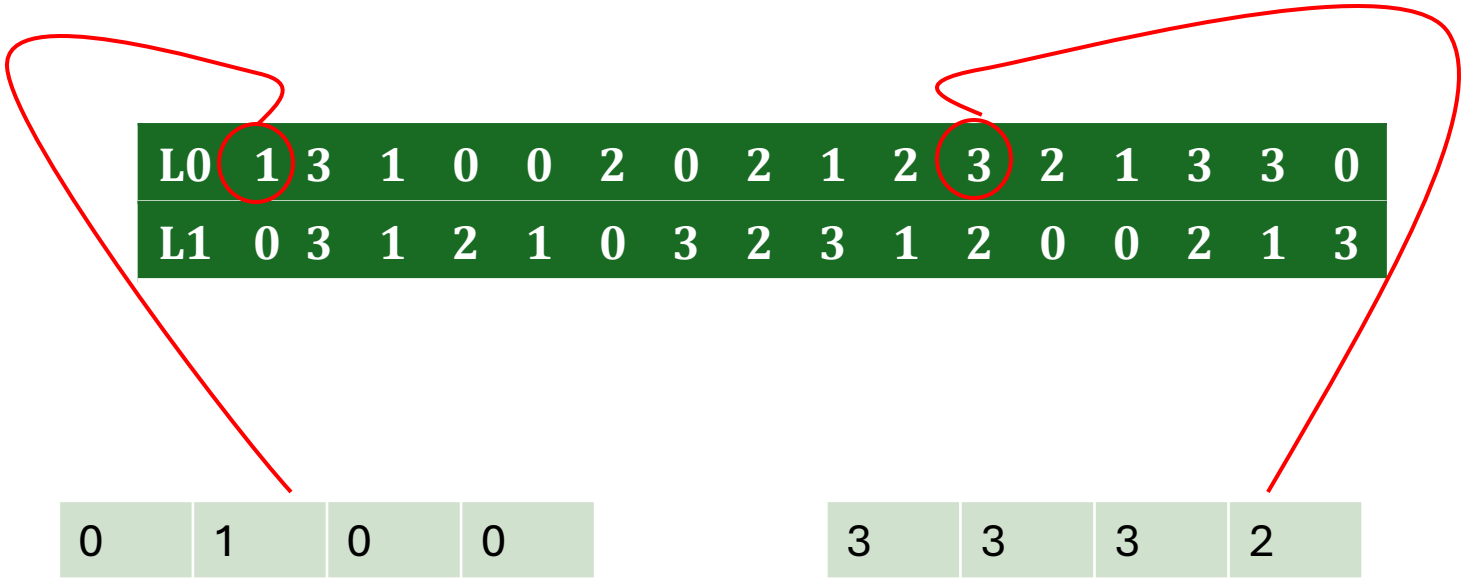
1. Can we continue to use **bits**?
2. How do we maintain **rank**?
3. Can we have fast **access**?

Naive Implementation (4-way Tree)



Goal:
Fastest access possible → Rank should be O(1) operation

Naive Implementation (4-way Tree)



Space for storing N elements = levels.N + N.levels.4

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr																	

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr																	

BOSTON UNIVERSITY

DiSC lab

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr																	

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr																	

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr	5																

Proposed Solution

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	40	60	1000	55	32	14	567	98	412	65	234	59	23	876	345	987
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

Mapped Arr	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

BOSTON UNIVERSITY

DiSC lab

The Build Function

Wavelet Arr	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

Divide the array into 4 partitions
B0:[0-3]
B1:[4-7]
B2:[8-11]
B3:[12-15]

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0

BOSTON UNIVERSITY

DiSC lab

The Build Function

Wavelet Arr	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

Populate a flat matrix of integers with encoded values

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L0	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

The Build Function

Wavelet Arr	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

Repartition L0 into 4 children

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L0	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3
L1	0	3	1	2	5	4	7	6	11	9	10	8	12	14	13	15

BOSTON UNIVERSITY

DiSC lab

The Build Function

Wavelet Arr	5	12	4	0	3	11	1	9	7	10	14	8	6	13	15	2

This is the stored array

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0

L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

The Rank Matrix

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

Rank matrix stores cumulative count
for that specific encoding within each bucket

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

BOSTON UNIVERSITY

DiSC lab

The Rank Matrix

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3
R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rank matrix stores cumulative count
for that specific encoding within each bucket

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

BOSTON UNIVERSITY

DiSC lab

The Rank Matrix

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3
~~R1~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~

Rank matrix has 1 row less

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

BOSTON UNIVERSITY

DiSC lab

The Access Function

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

We want to find:
Where does the 2nd sorted element exist in Original Array

2nd sorted element = 23 at index 1

BOSTON UNIVERSITY

DiSC lab

The Access Function

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

We want to find:
Where does the 2nd sorted element exist in Original Array

2nd sorted element = 23 at index 1

The output should be index 12!

BOSTON UNIVERSITY

DiSC lab

The Access Function

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Original	**40**	**60**	**1000**	**55**	**32**	**14**	**567**	**98**	**412**	**65**	**234**	**59**	**23**	**876**	**345**	**987**
Sorted	14	23	32	40	55	59	60	65	98	234	345	412	567	876	987	1000

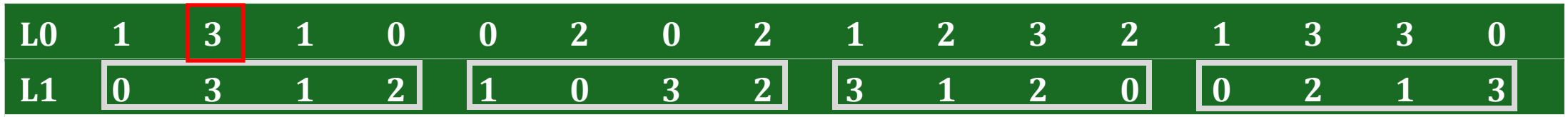
RTP:access(1) = 12

BOSTON UNIVERSITY

DiSC lab

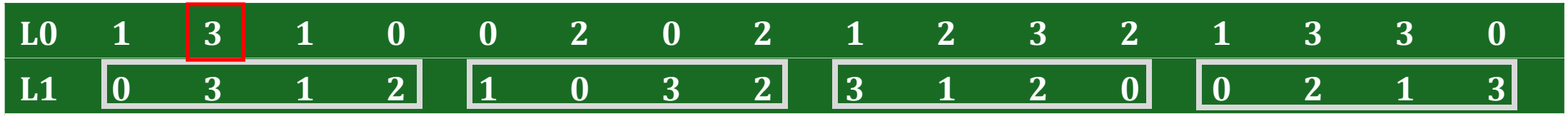
The Access Function

Find element at index 1 at L0



The Access Function

L0[1] = 3 so go to bucket 3 at L1



The Access Function

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

L0[1] = 3 so go to bucket 3 at L1

Every bucket has equal elements

So bucket 3 is at pos = 3.(arity)=12

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

BOSTON UNIVERSITY

DiSC lab

The Access Function

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

L0[1] = 3 so go to bucket 3 at L1

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

Then check rank matrix for R0[1]
Since R[0] = 0 so this is at 12+R[0] = 12

The Access Function

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

L0[1] = 3 so go to bucket 3 at L1

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3

Then check rank matrix for R0[1]
Since R[0] = 0 so this is at 12+R[0] = 12

The Access Function

L0[1] = 3 so go to bucket 3 at L1

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

At L1[12] = 0 so lets go to the 0^{th} bucket

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The Access Function

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

The 0th bucket has only 1 node

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The Access Function

R0	0	0	1	0	1	0	2	1	2	2	1	3	3	2	3	3

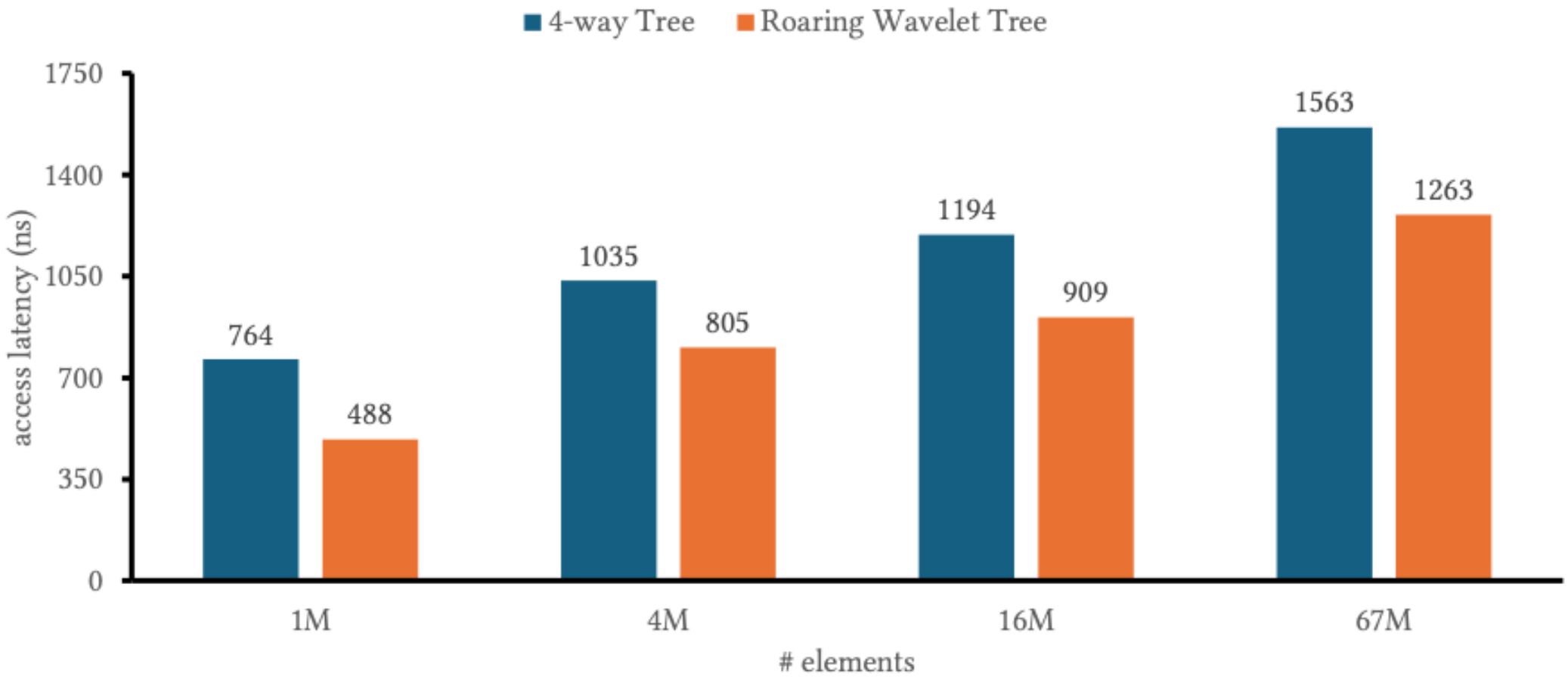
Return the left-to-right ordering

L0	1	3	1	0	0	2	0	2	1	2	3	2	1	3	3	0
L1	0	3	1	2	1	0	3	2	3	1	2	0	0	2	1	3
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

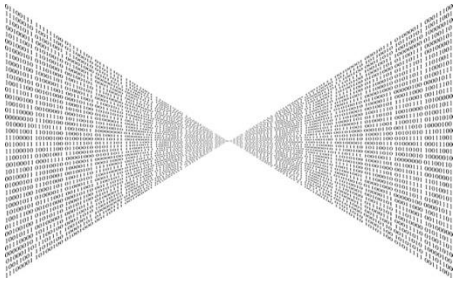
BOSTON UNIVERSITY

DiSC lab

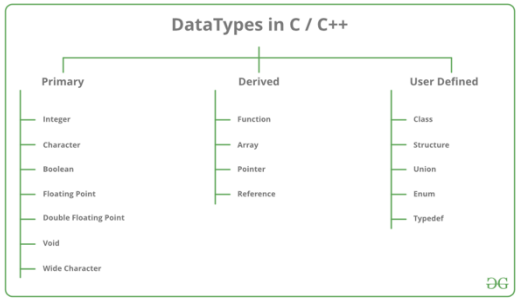
4-way Tree Results



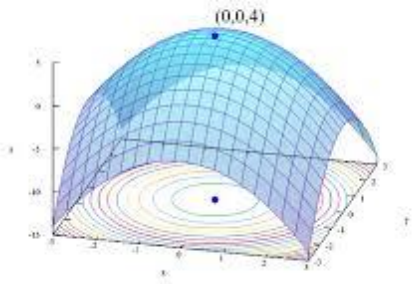
Optimize Further?



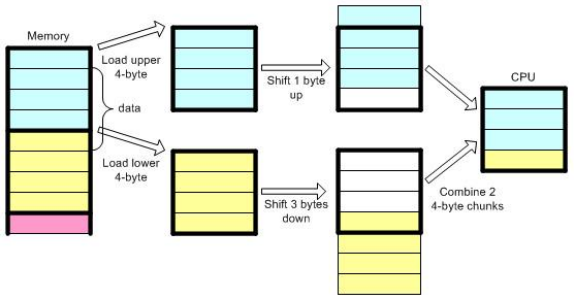
Bitpacking



Data types ⟵ Optimization ⟶ Rank Optimization



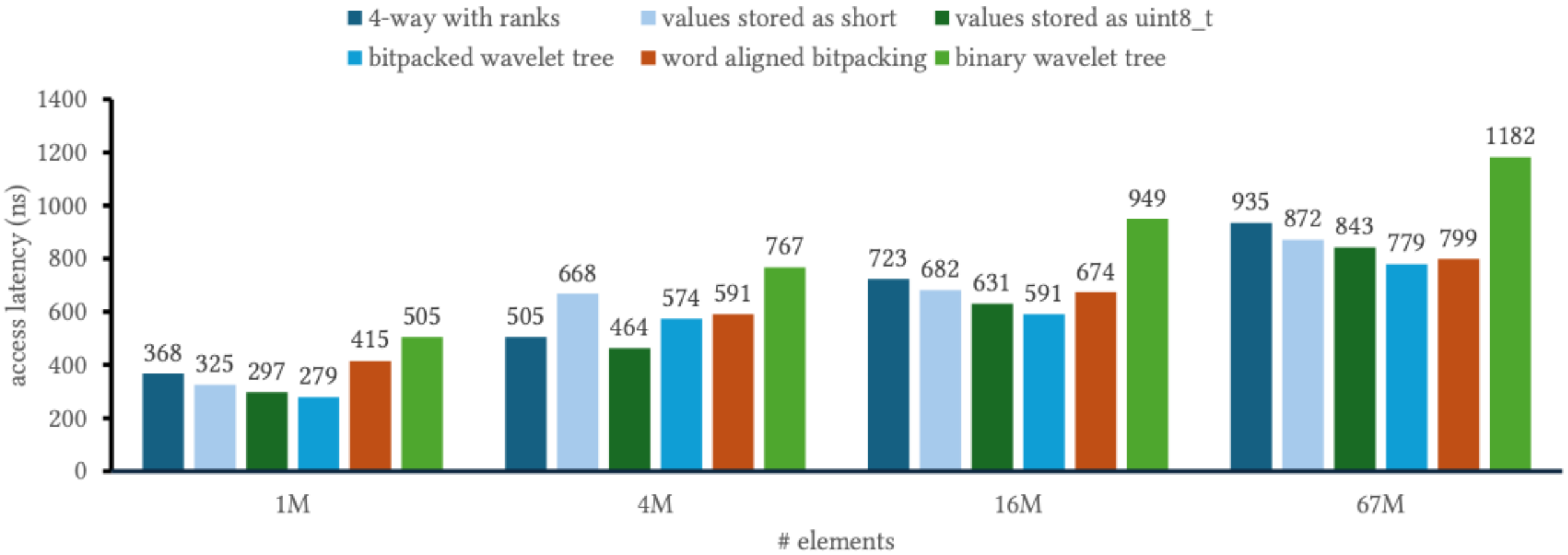
Word Alignment



BOSTON UNIVERSITY

DiSC lab

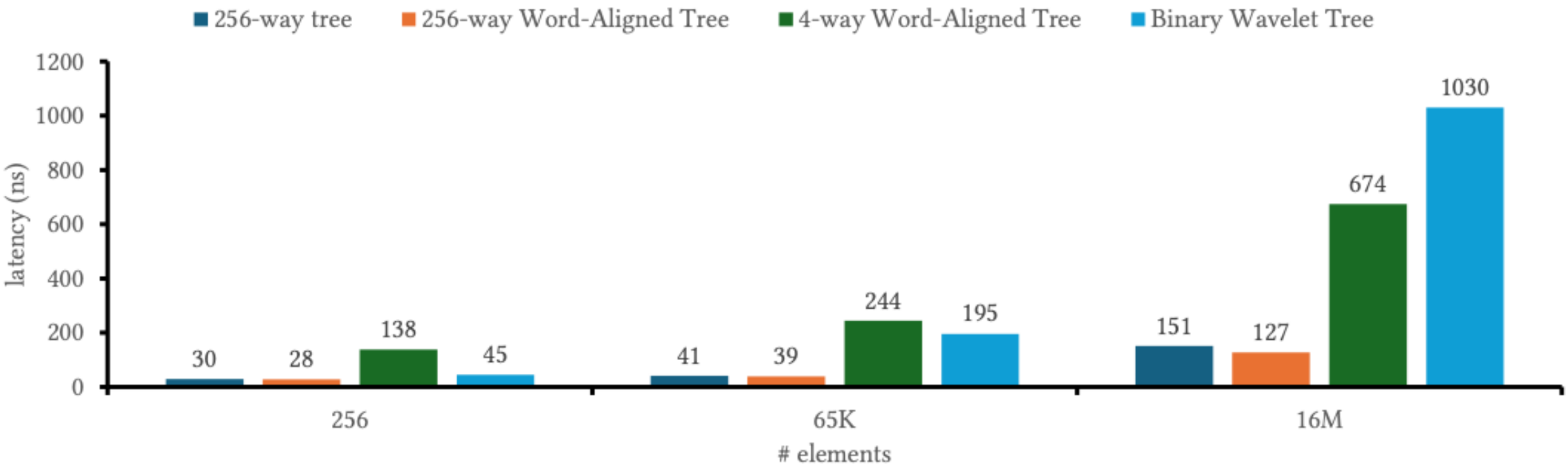
Optimized 4-way Tree



Could We Do Better?

What about adding more branches?

Extending to the 256-way Tree



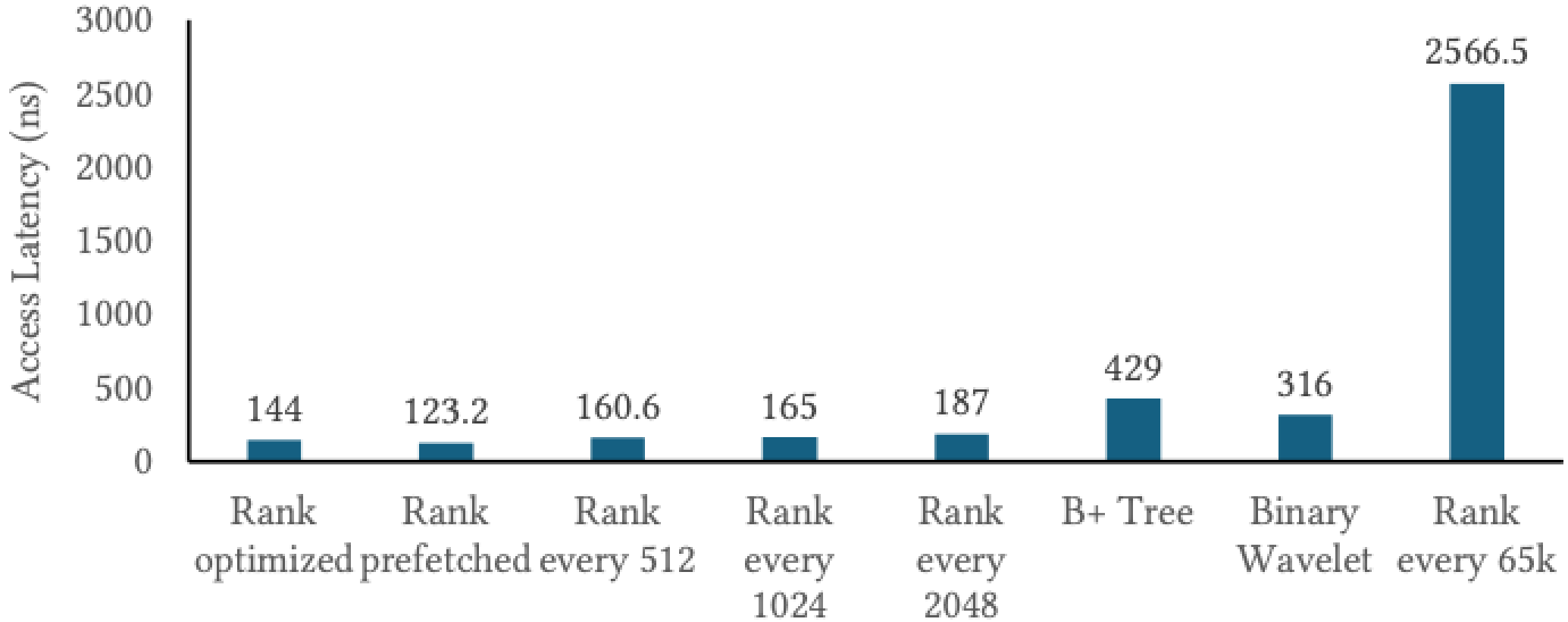
Can We Optimize Even Further?

Key observation:

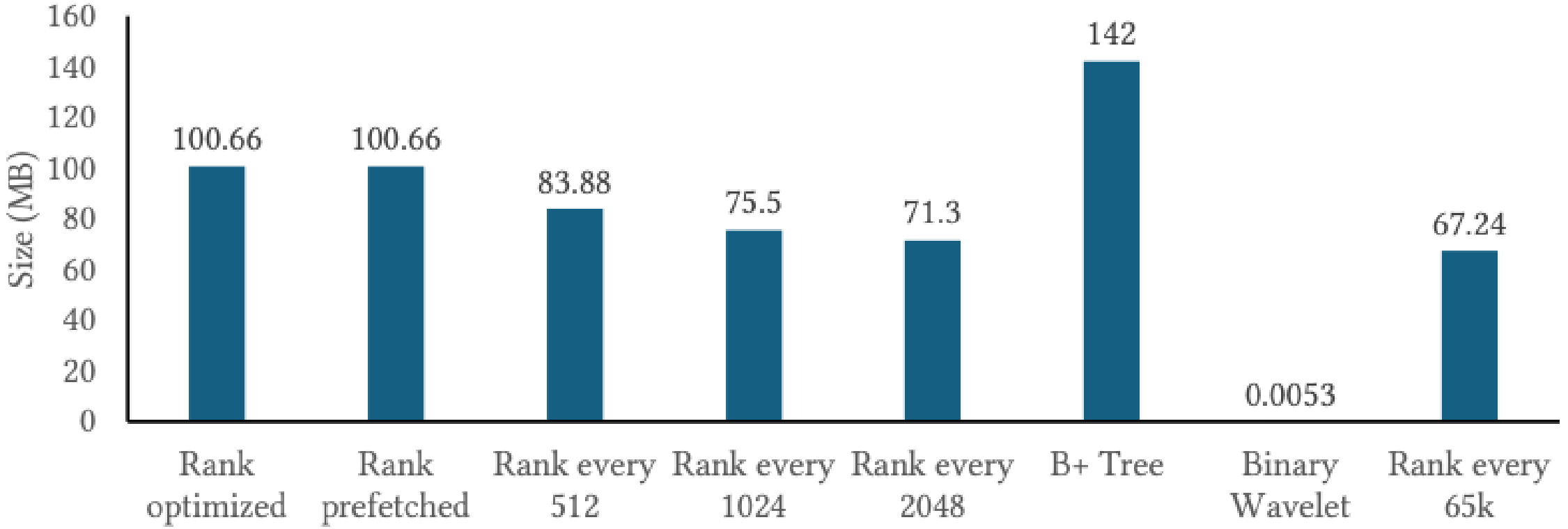
Rank matrix occupies space!

Can we store less for it?

Optimized 256-Way Tree (Latency)



Optimized 256-Way Tree (Space)

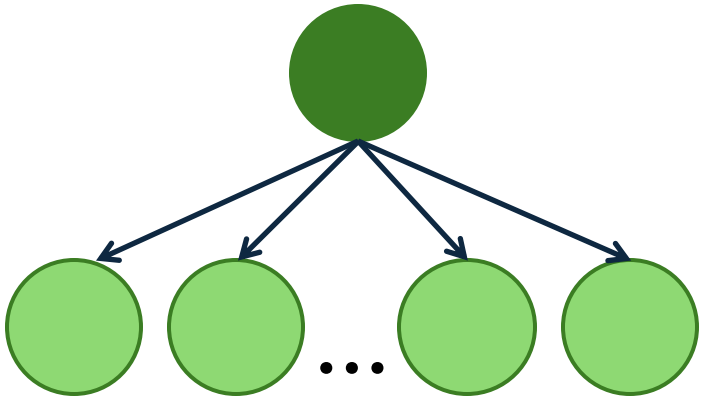


Optimized 256-Way Tree

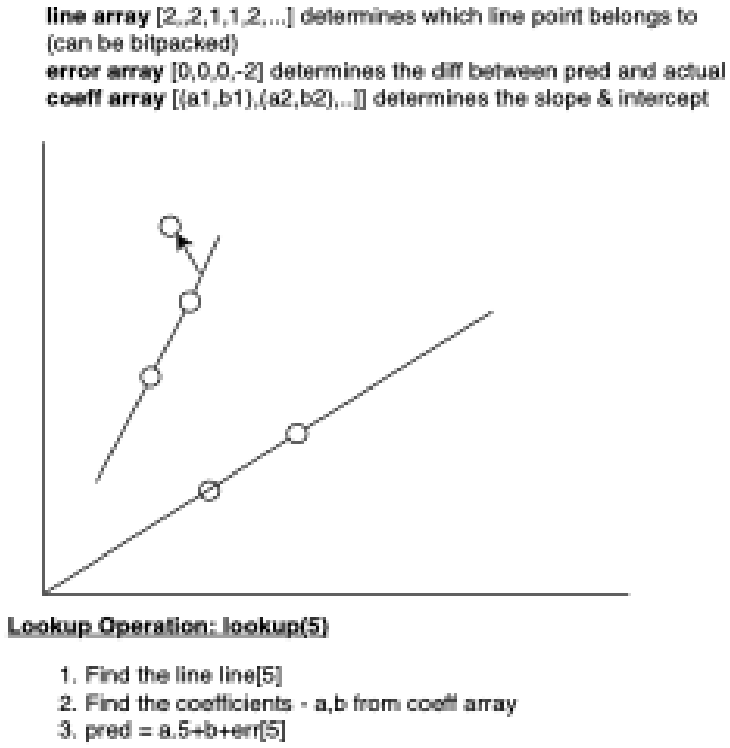
Could we do even better?

Maybe?
Try to make it as close to the roaring binary wavelet tree

Next Steps

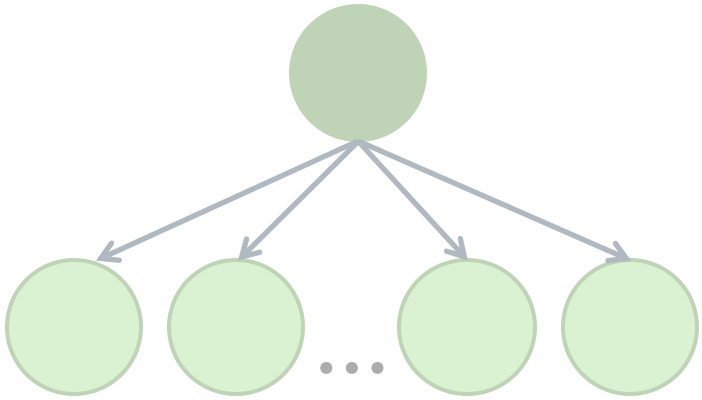


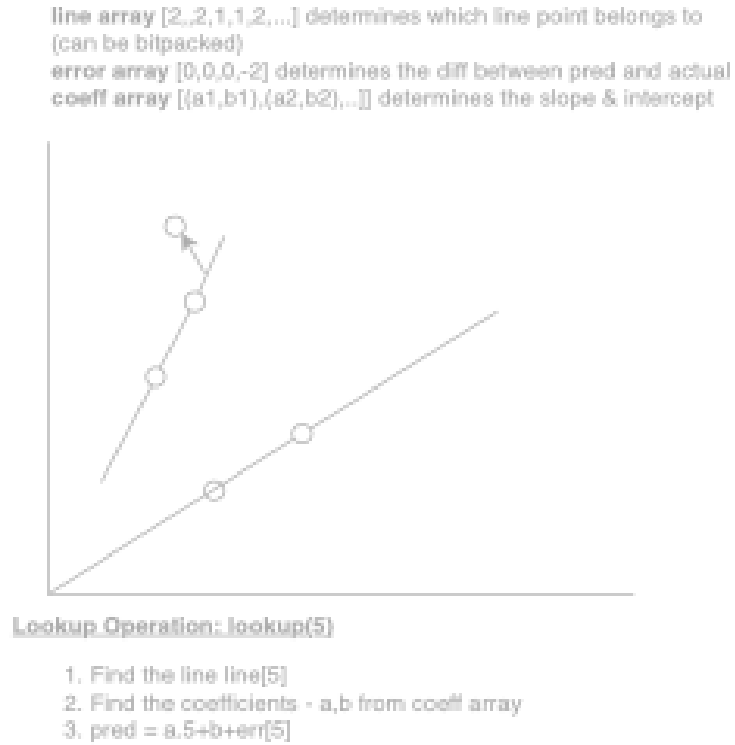
Optimize Further



Constellation Maps

Next Steps





Questions