Memory Management and Dynamic Data Structures on GPUs

Prashant Pandey, Northeastern University, Boston
https://prashantpandey.github.io/

Managing Parallelism Workshop, Simons Institute, Berkeley
Fall 2025

Goal: high-performance data processing on GPUs

Scalable data structures

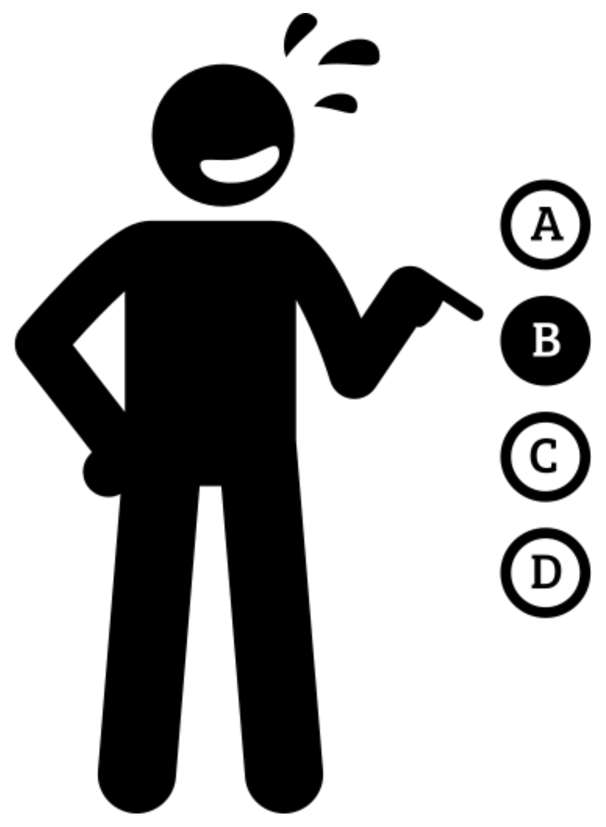
Streaming graphs

Spare tensor contractions

GPU applications: query processing, graph analytics, ML, quantum science.

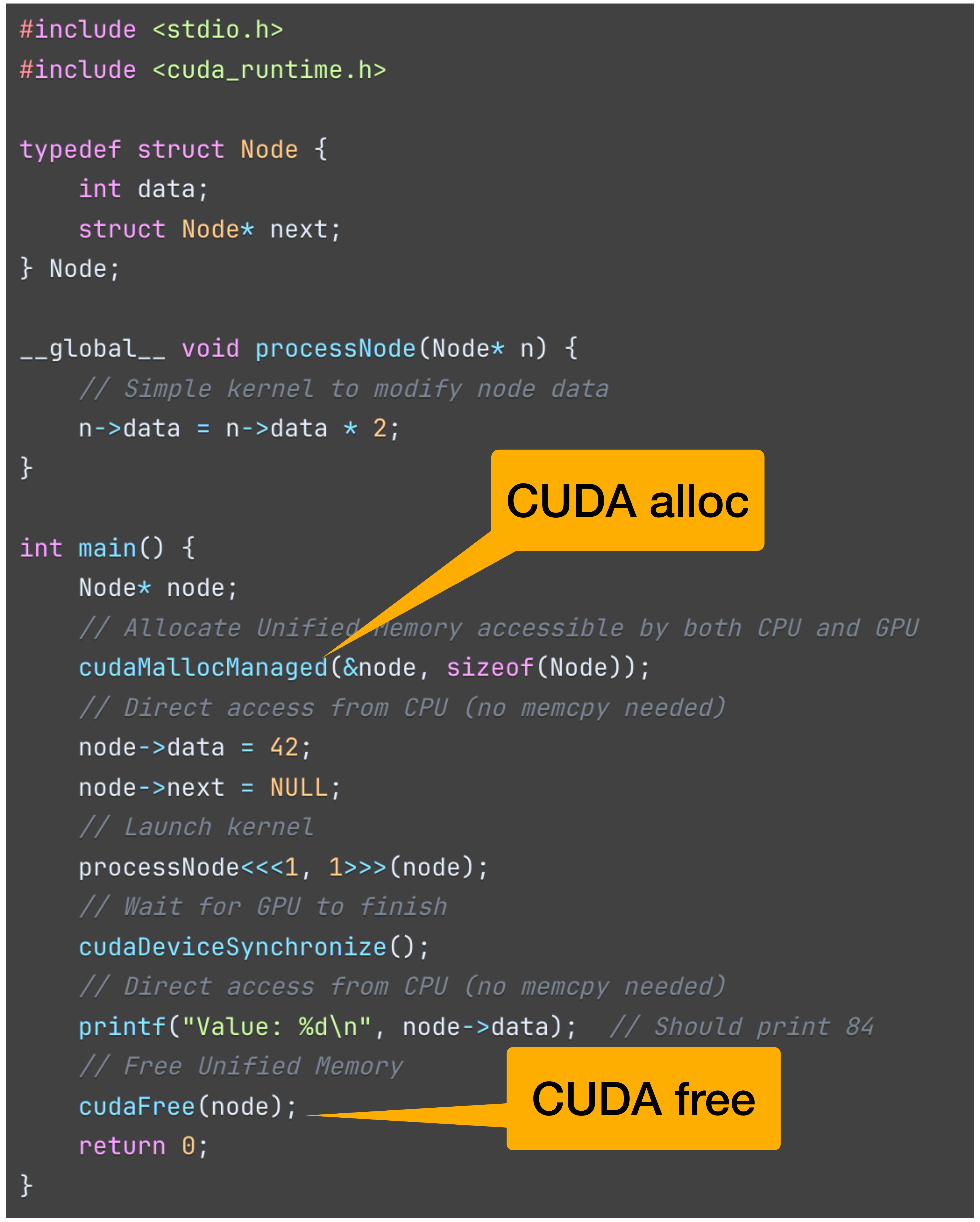
Need a general-purpose GPU memory manager

Fast allocations/frees



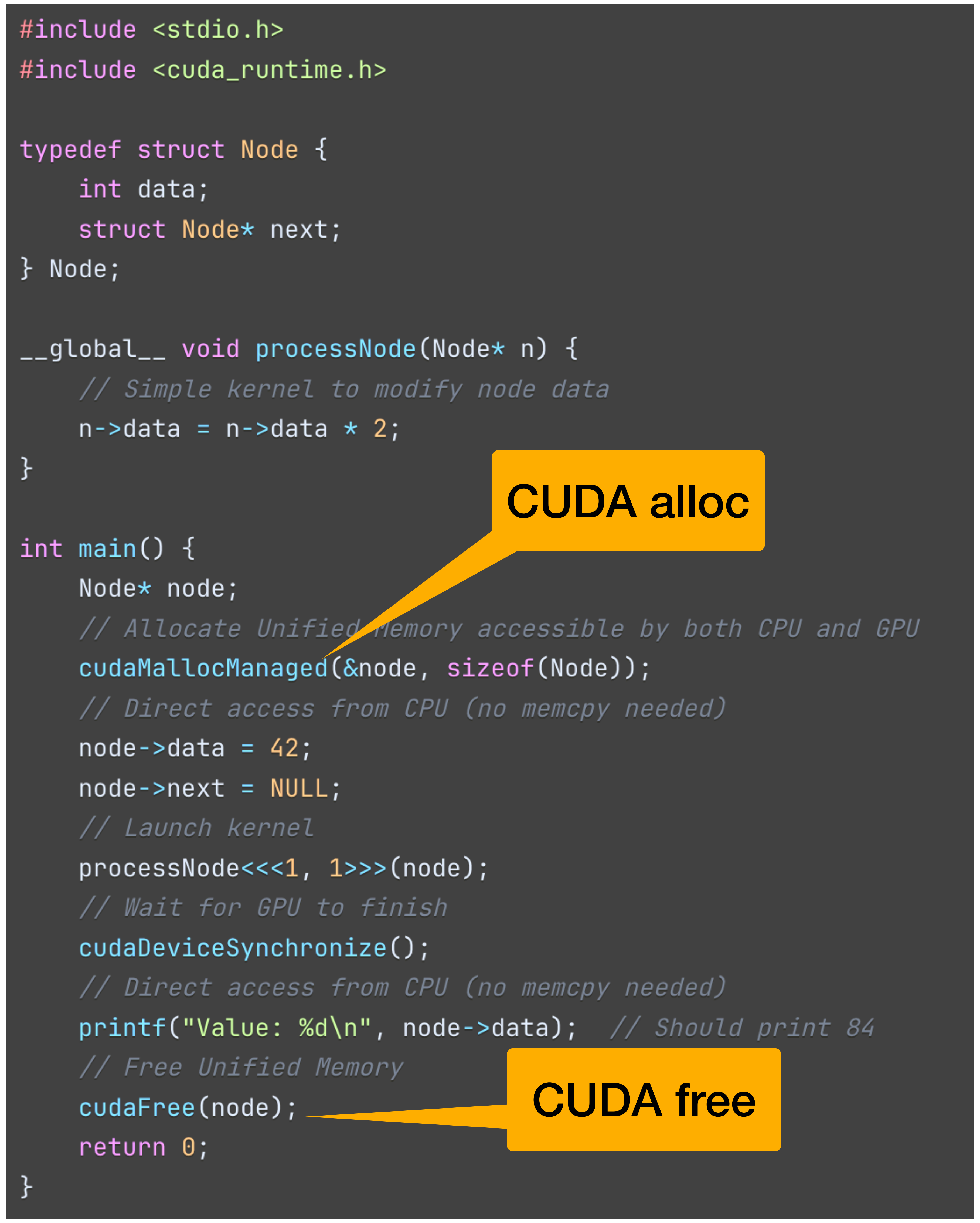
Arbitrary sizes

Problem: allocation/free on GPUs is slow

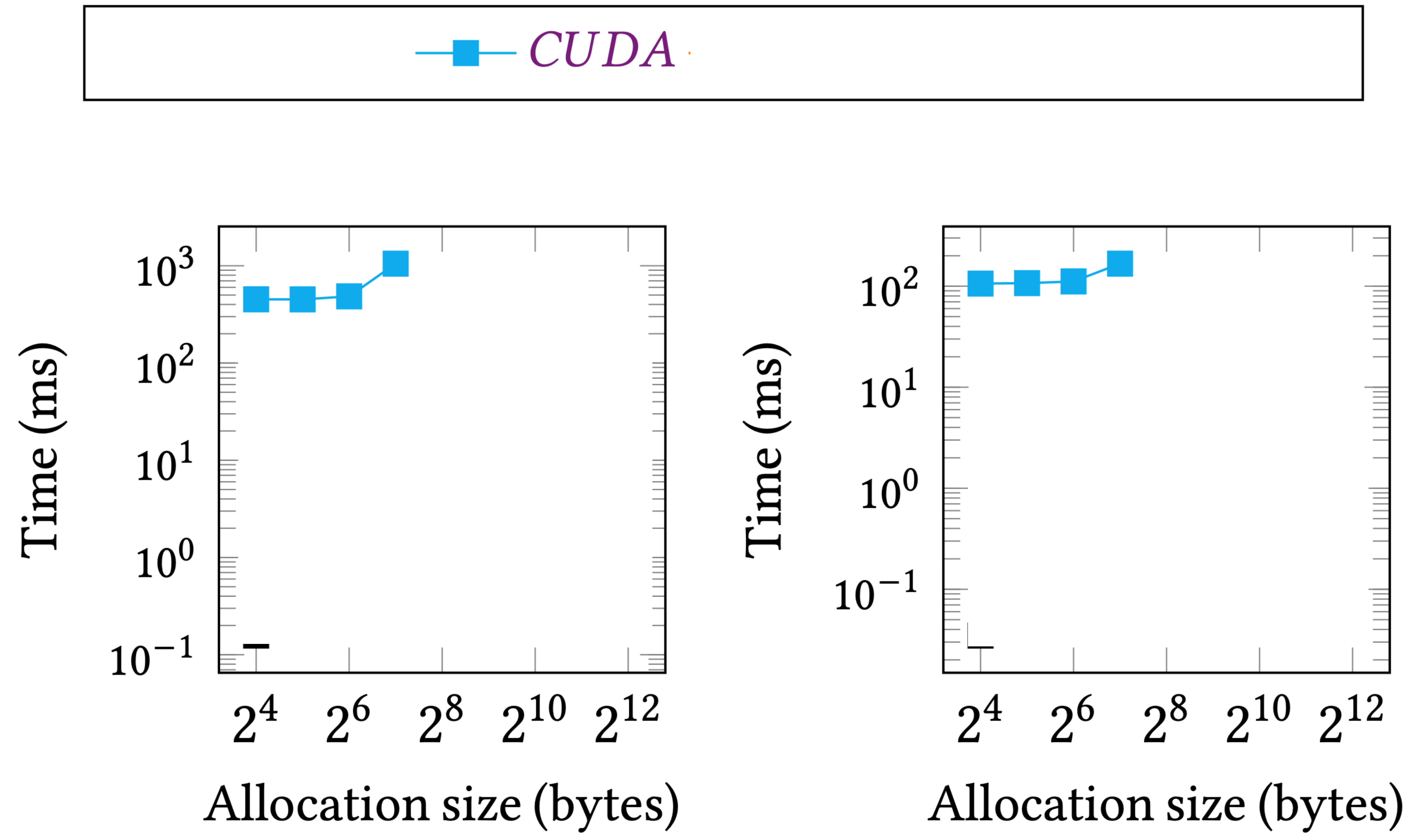


```c
#include <stdio.h>
#include <cuda_runtime.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

__global__ void processNode(Node* n) {
    // Simple kernel to modify node data
    n->data = n->data * 2;
}

int main() {
    Node* node;
    // Allocate Unified Memory accessible by both CPU and GPU
    cudaMallocManaged(&node, sizeof(Node));
    // Direct access from CPU (no memcpy needed)
    node->data = 42;
    node->next = NULL;
    // Launch kernel
    processNode<<<1, 1>>>(node);
    // Wait for GPU to finish
    cudaDeviceSynchronize();
    // Direct access from CPU (no memcpy needed)
    printf("Value: %d\n", node->data);  // Should print 84
    // Free Unified Memory
    cudaFree(node);
    return 0;
}
```

Problem: allocation/free on GPUs is slow

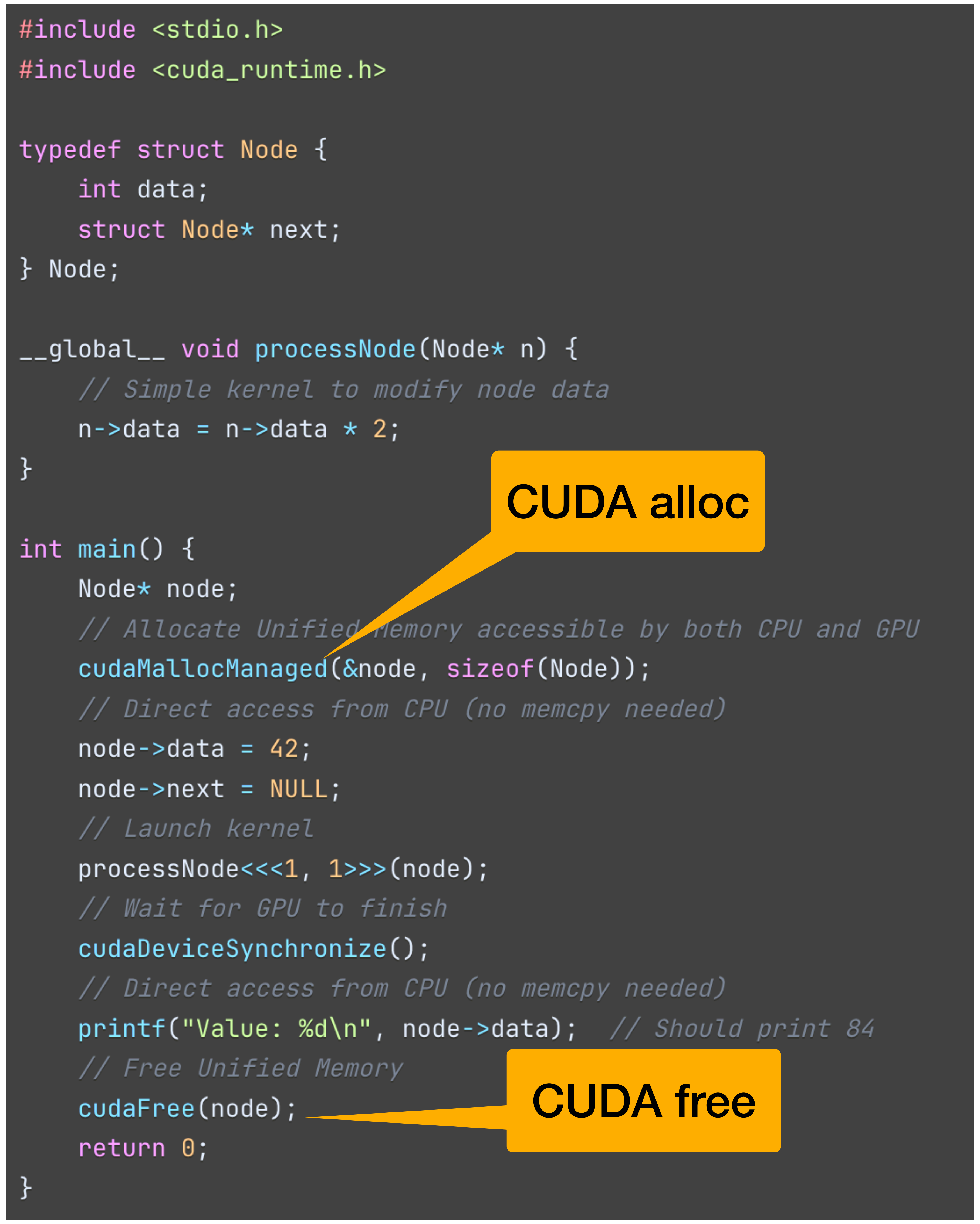


```c
#include <stdio.h>
#include <cuda_runtime.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

__global__ void processNode(Node* n) {
    // Simple kernel to modify node data
    n->data = n->data * 2;
}

int main() {
    Node* node;
    // Allocate Unified Memory accessible by both CPU and GPU
    cudaMallocManaged(&node, sizeof(Node));
    // Direct access from CPU (no memcpy needed)
    node->data = 42;
    node->next = NULL;
    // Launch kernel
    processNode<<<1, 1>>>(node);
    // Wait for GPU to finish
    cudaDeviceSynchronize();
    // Direct access from CPU (no memcpy needed)
    printf("Value: %d\n", node->data);  // Should print 84
    // Free Unified Memory
    cudaFree(node);
    return 0;
}
```

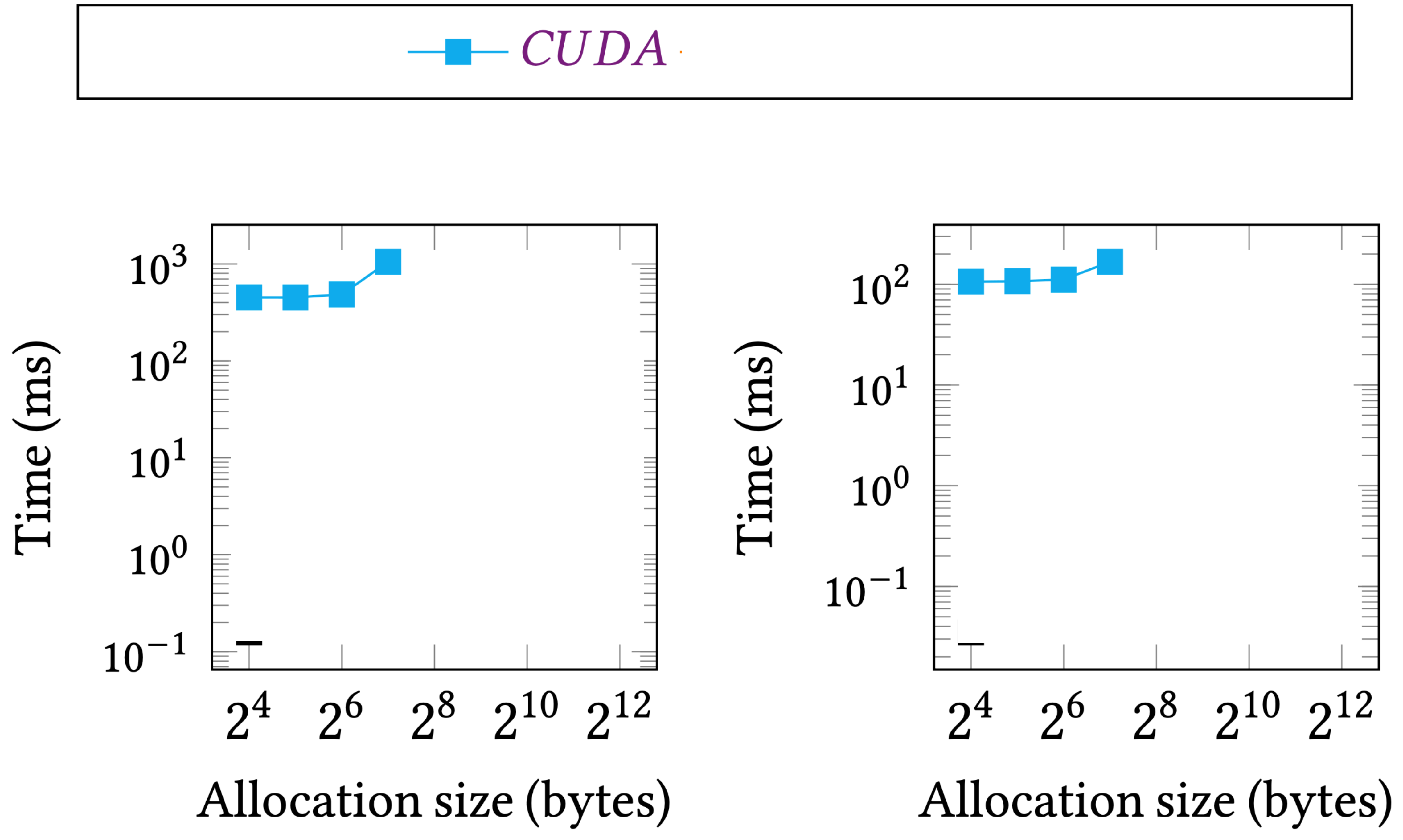


Allocations **Free**

Problem: allocation/free on GPUs is slow



```
#include <stdio.h>
#include <cuda_runtime.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

__global__ void processNode(Node* n) {
    // Simple kernel to modify node data
    n->data = n->data * 2;
}

int main() {
    Node* node;
    // Allocate Unified Memory accessible by both CPU and GPU
    cudaMallocManaged(&node, sizeof(Node));
    // Direct access from CPU (no memcpy needed)
    node->data = 42;
    node->next = NULL;
    // Launch kernel
    processNode<<<1, 1>>>(node);
    // Wait for GPU to finish
    cudaDeviceSynchronize();
    // Direct access from CPU (no memcpy needed)
    printf("Value: %d\n", node->data);  // Should print 84
    // Free Unified Memory
    cudaFree(node);
    return 0;
}
```



Allocations **Free**

CUDA malloc/free can take **~sec!**

Dynamic data structures and applications on GPUs

Metagenomic data processing in MHM
ACDA 2023

Jasper: App. Nearest Neighbor Search
(Ongoing)

GPU Filters
PPoPP 2023

Streaming graphs
PPoPP 2024

GPU Hash tables
ALENEX 2026

Gallatin: GPU Memory Manager
PPoPP 2024

https://github.com/saltsystemslab/

Gallatin: GPU Memory Manager
PPoPP 2024

Hunter McCoy

The speed and generality trade-off

Allocator	Arbitrary allocations	High performance
CUDA Allocator		
Xmalloc [HRG+10]*		
ScatterAlloc [SKK+12]*		
FDGMalloc [WWW+13]*		
RegEff [VH+14]		
Halloc [AP+14]*		
RWMalloc [PLY+22]		
Ouroboros [WMP+20]*		

*: Can fall back on CUDA allocator.

The speed and generality trade-off

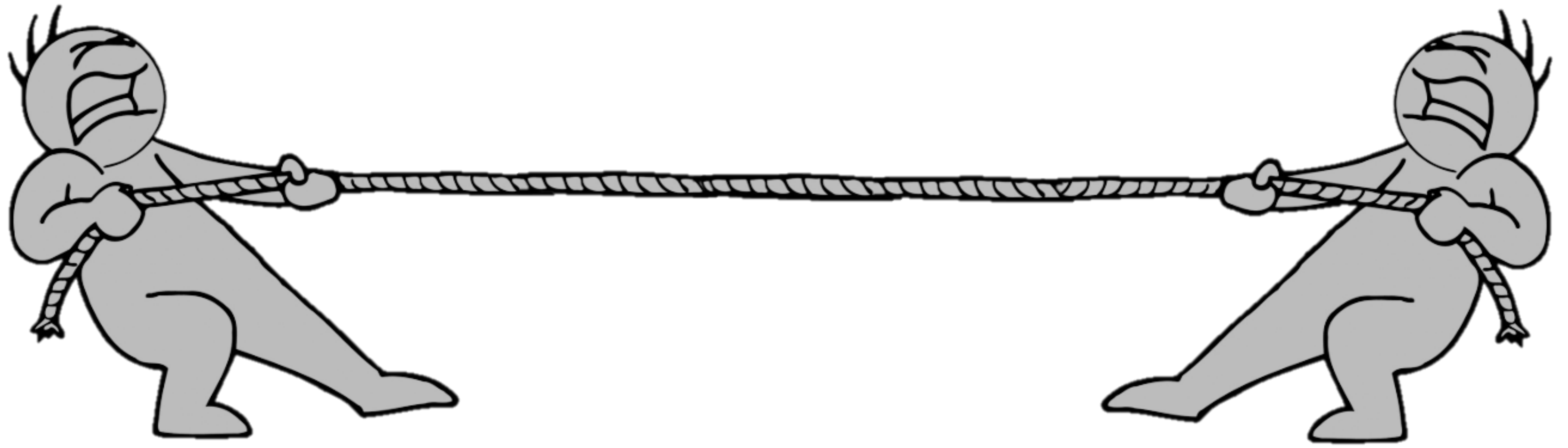
Allocator	Arbitrary allocations	High performance
CUDA Allocator	✓	✗
Xmalloc [HRG+10]*	✓	✗
ScatterAlloc [SKK+12]*	✗	✓
FDGMalloc [WWW+13]*	✗	✗
RegEff [VH+14]	✗	✓
Halloc [AP+14]*	✗	✓
RWMalloc [PLY+22]	✗	✓
Ouroboros [WMP+20]*	✗	✓

*: Can fall back on CUDA allocator.

The speed and generality trade-off

Fast allocations

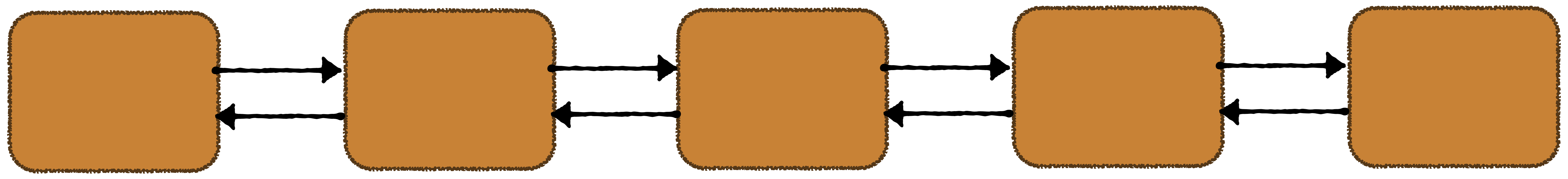
Flexible sizes



Question: can we provide arbitrary allocation sizes and high performance?

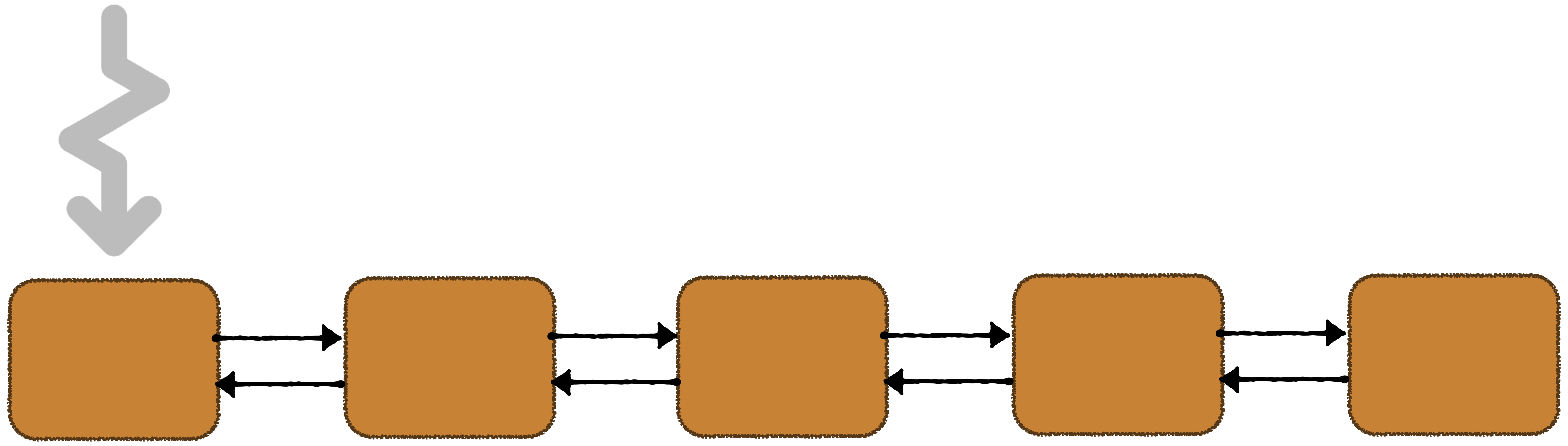
Why do existing GPU memory managers struggle?

1. Linked lists are flexible but slow



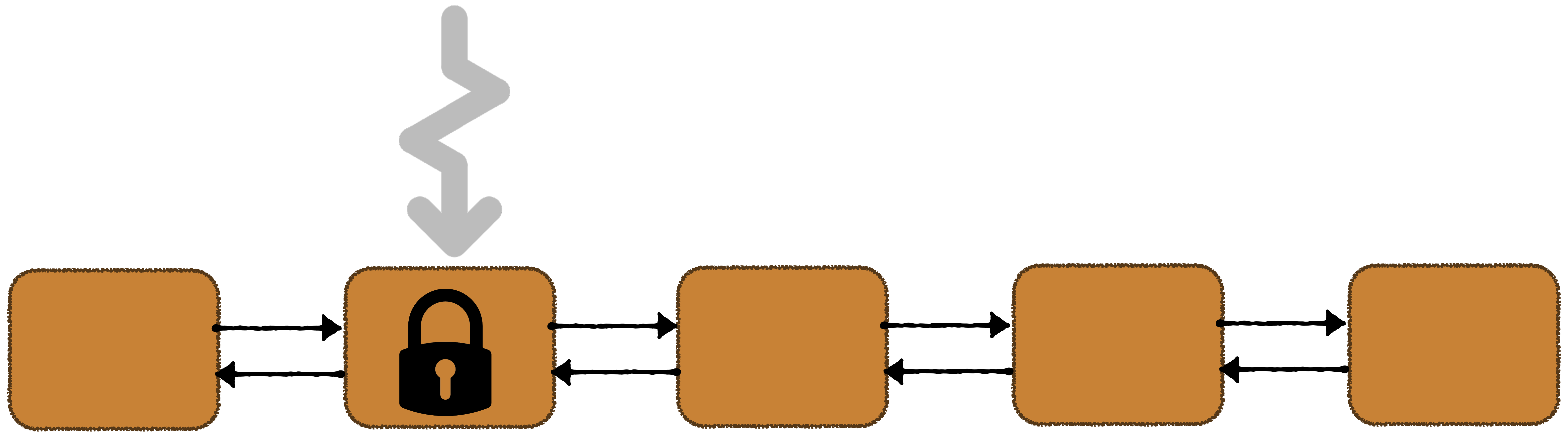
Allocators: CUDA allocator, Xmalloc [HRG+10], RegEff [VH+14]

1. Linked lists are flexible but slow



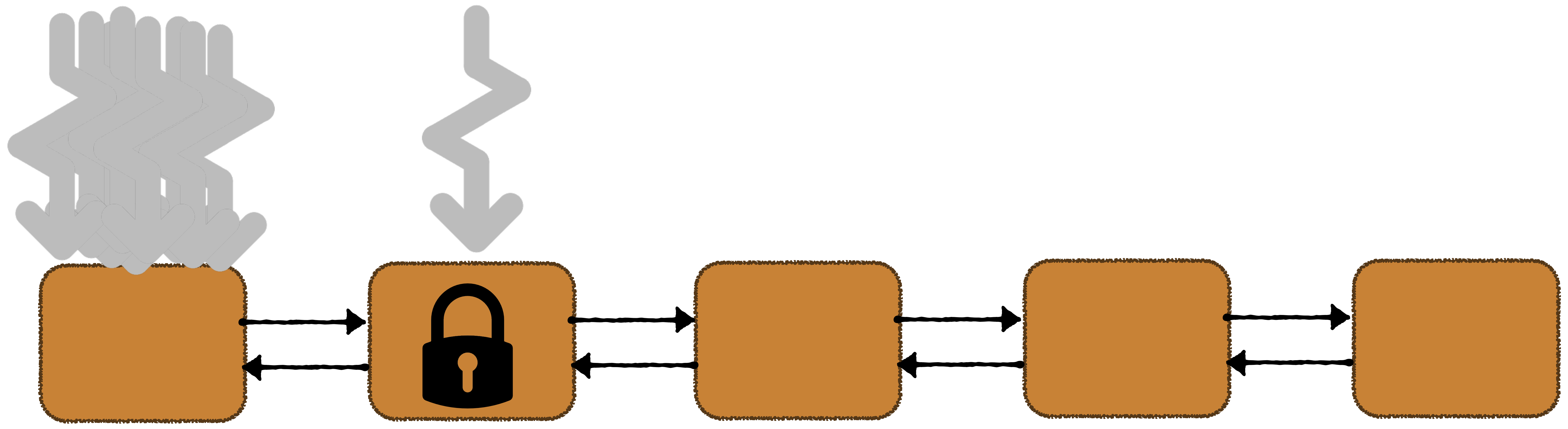
Allocators: CUDA allocator, Xmalloc [HRG+10], RegEff [VH+14]

1. Linked lists are flexible but slow



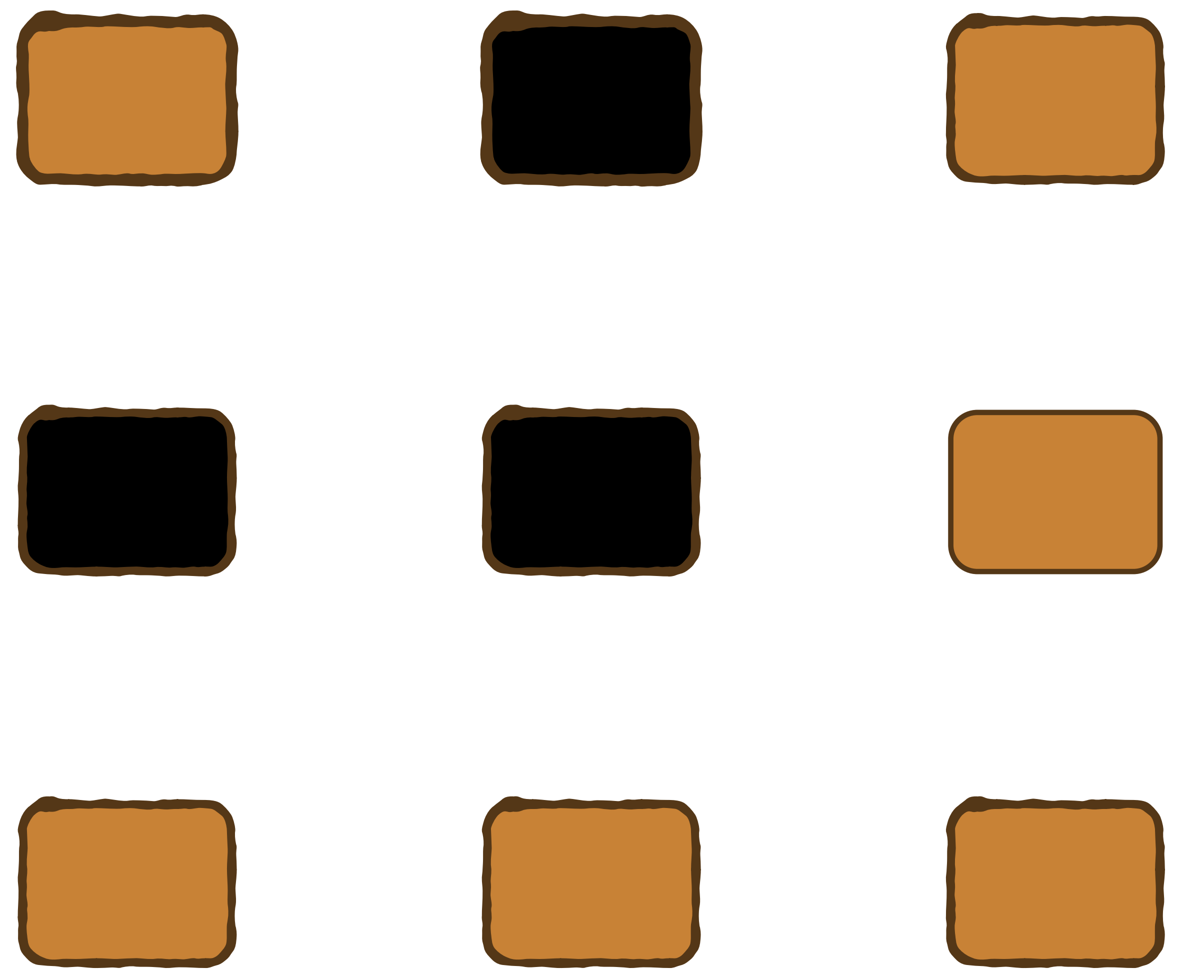
Allocators: CUDA allocator, Xmalloc [HRG+10], RegEff [VH+14]

1. Linked lists are flexible but slow



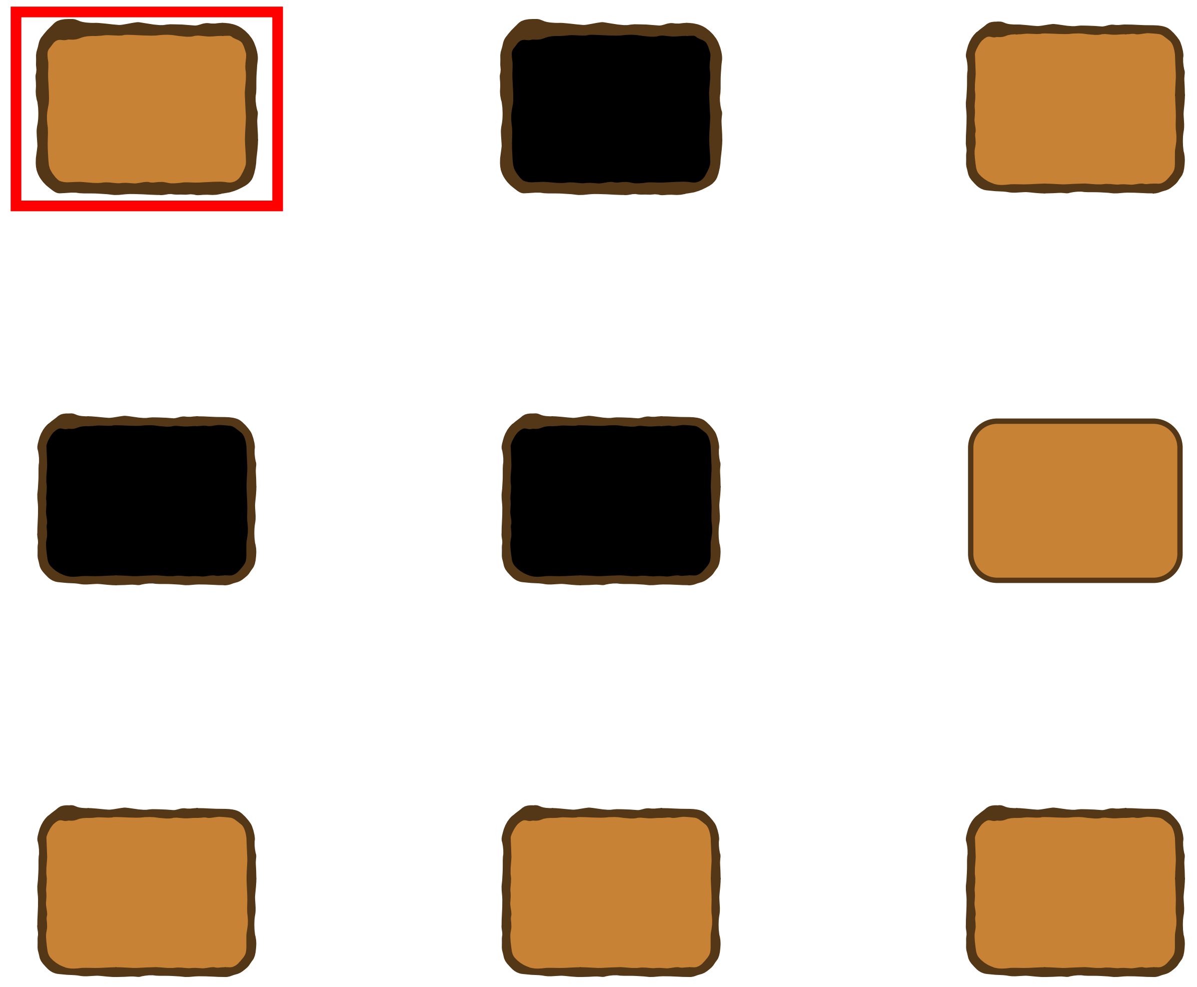
Allocators: CUDA allocator, Xmalloc [HRG+10], RegEff [VH+14]

2. Random walks are fast but don't scale



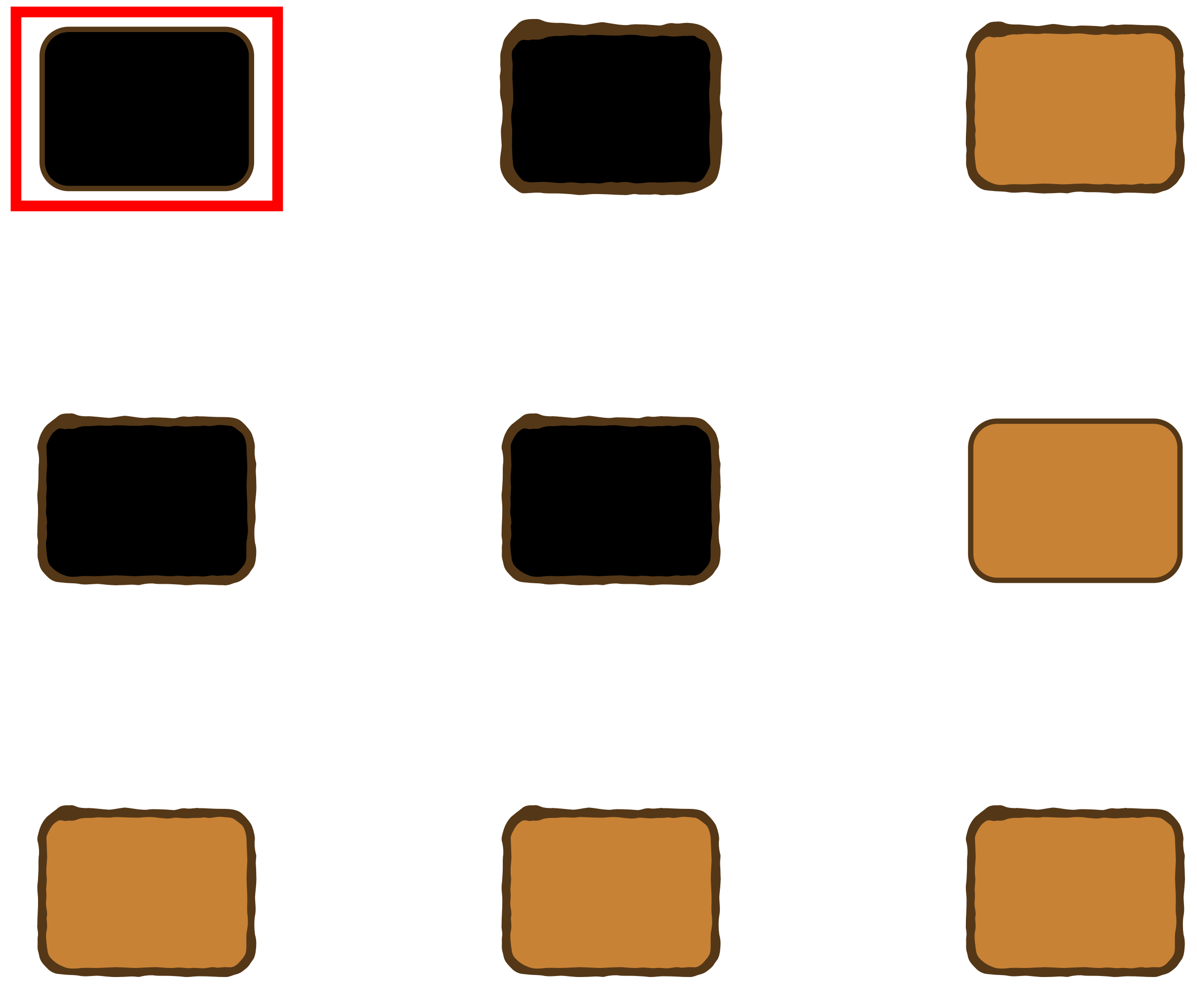
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don't scale



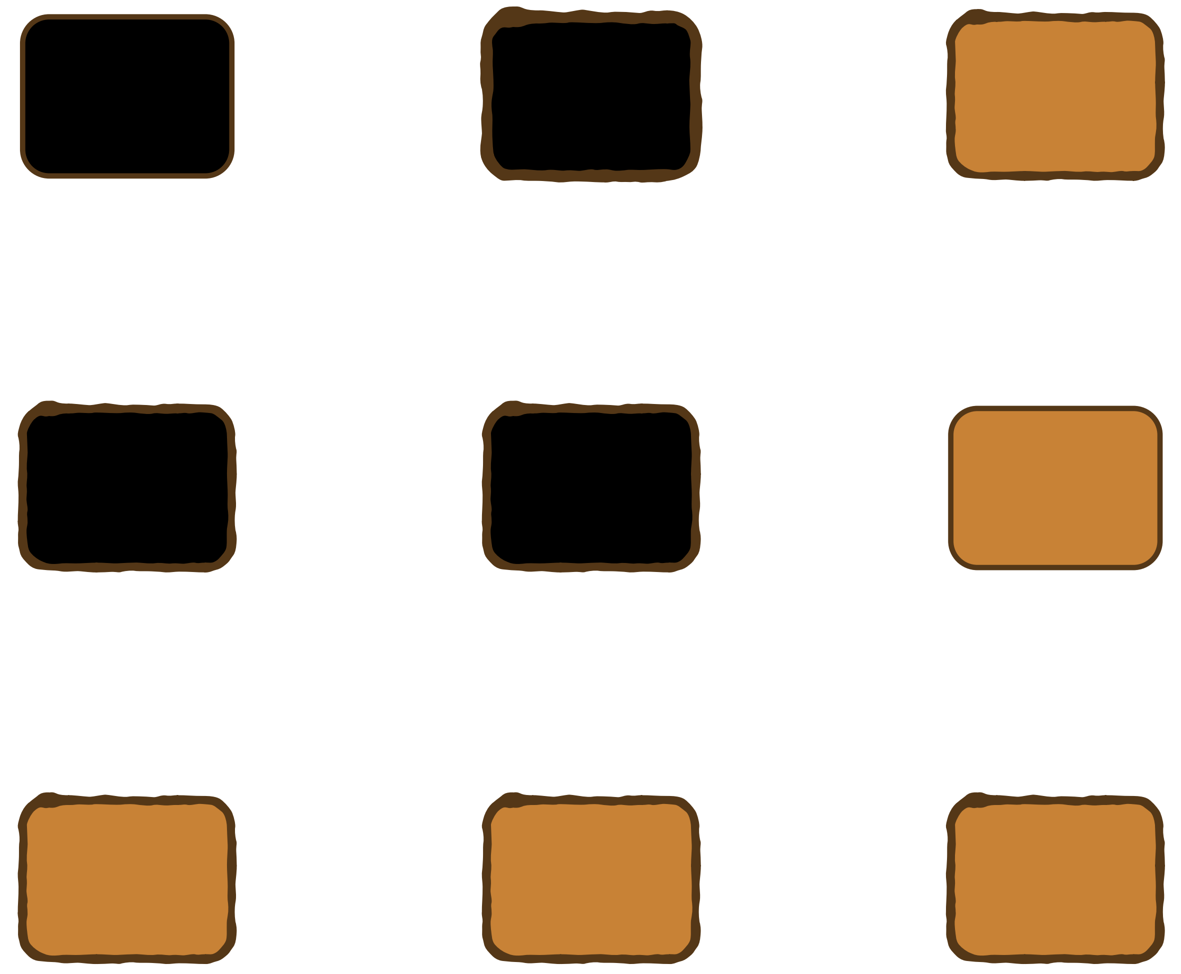
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don't scale



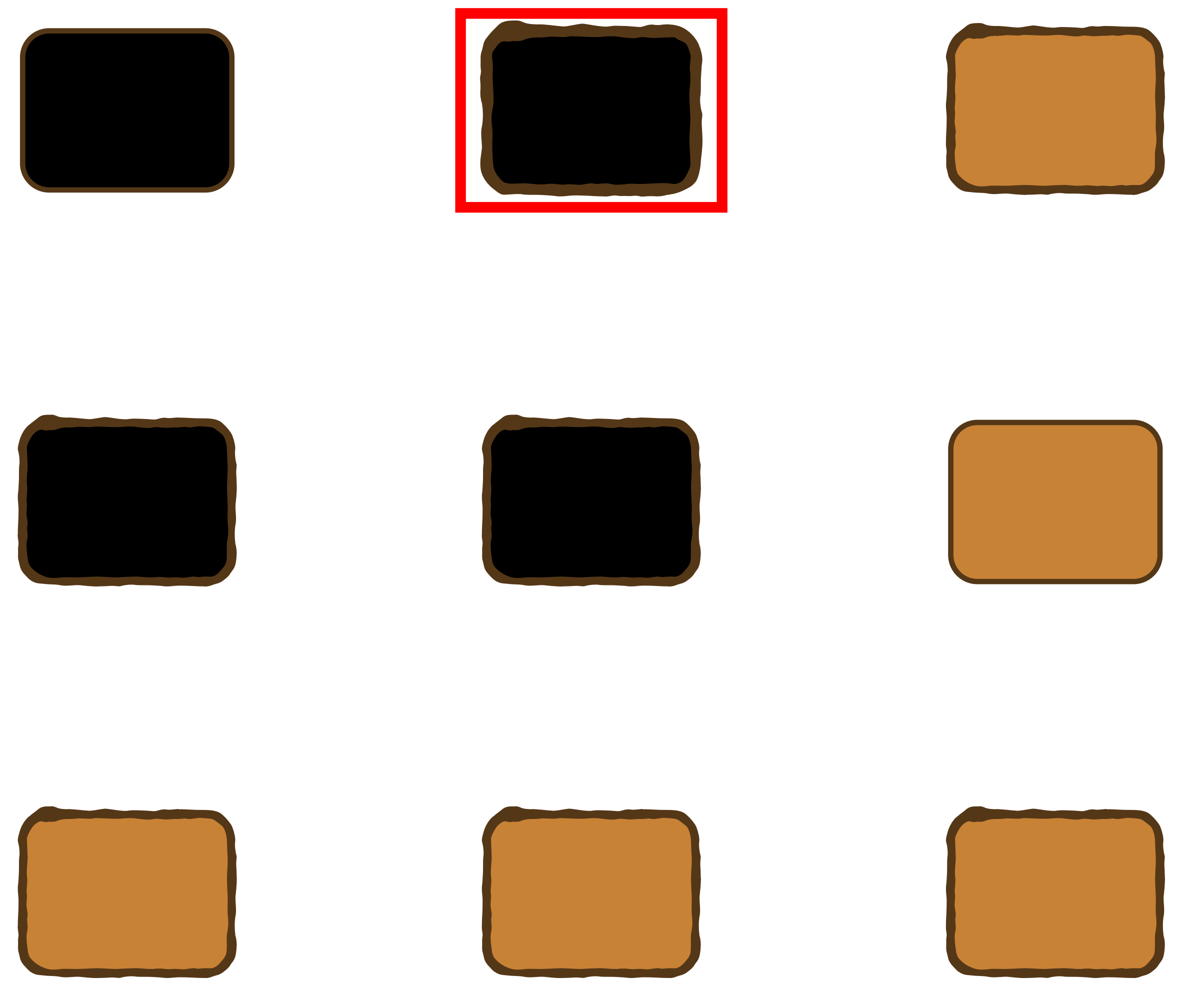
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don’t scale



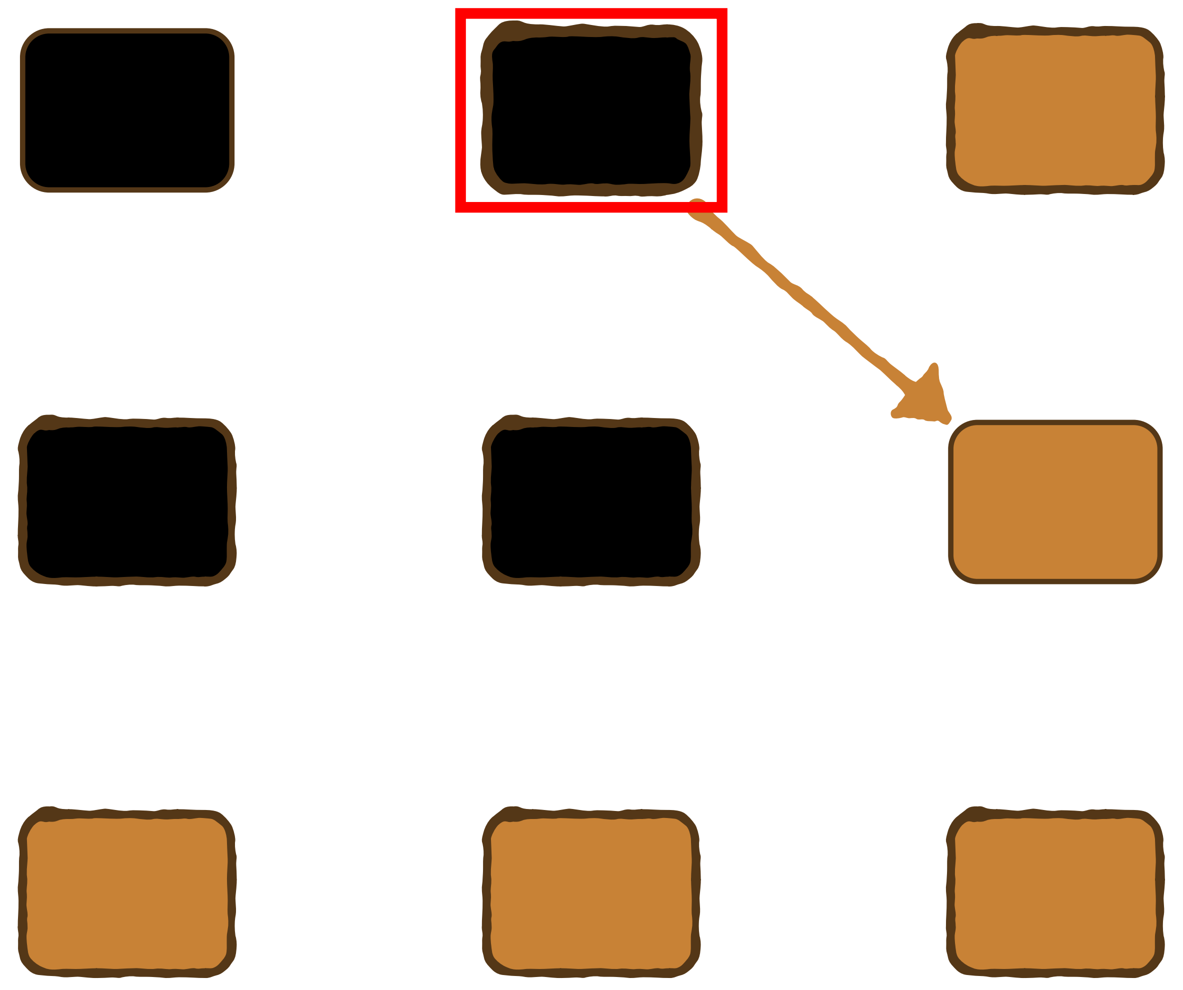
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don’t scale



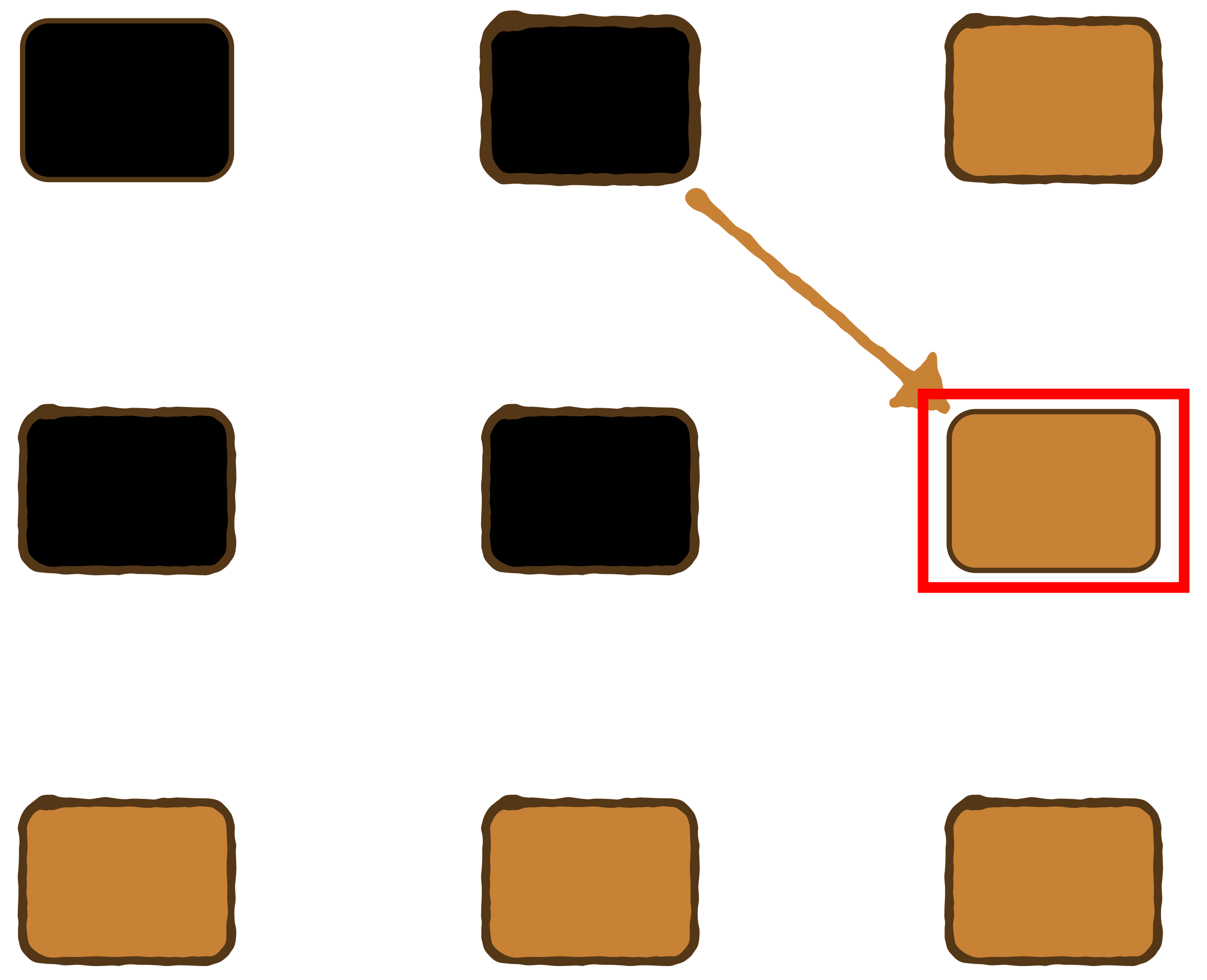
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don’t scale



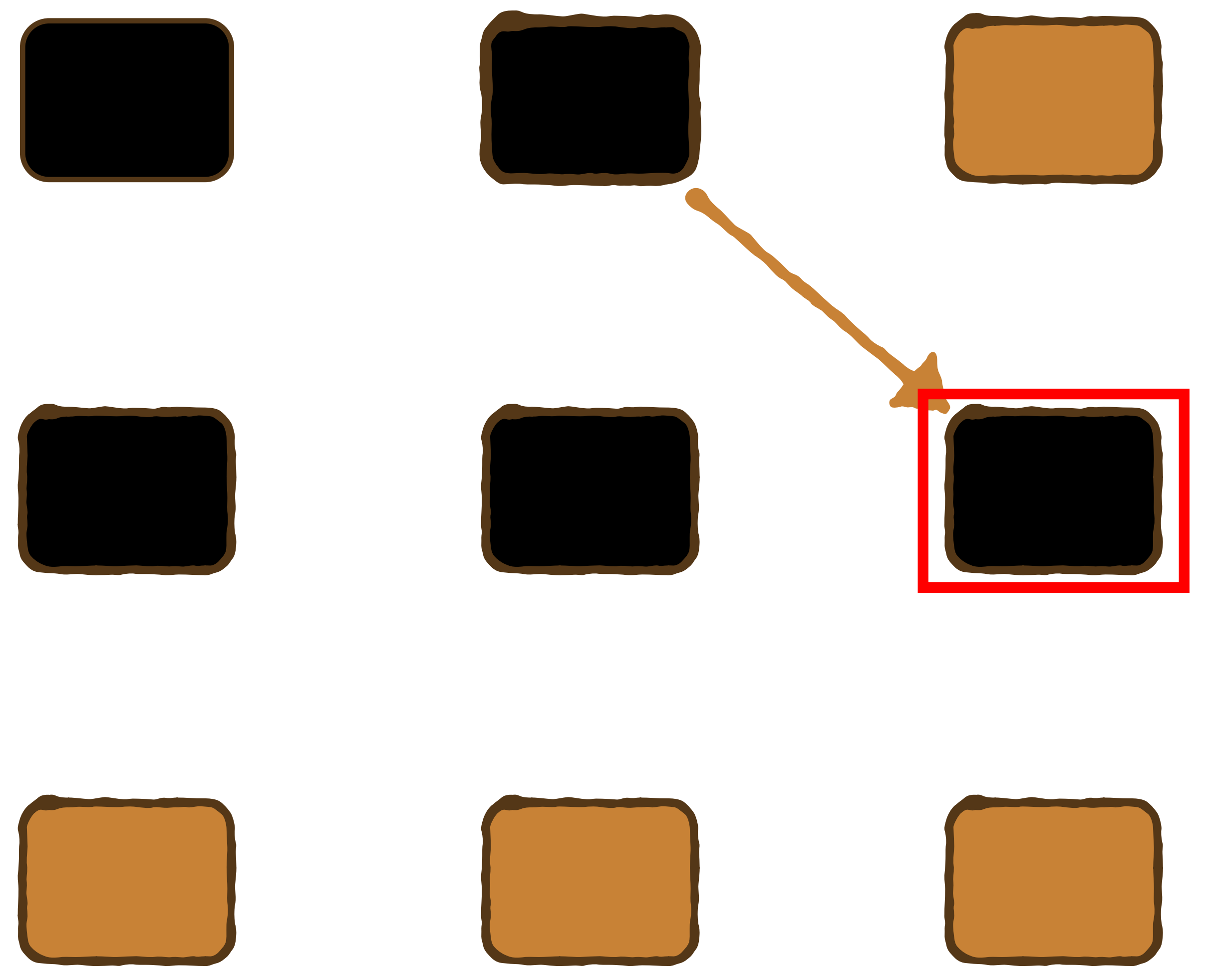
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don't scale



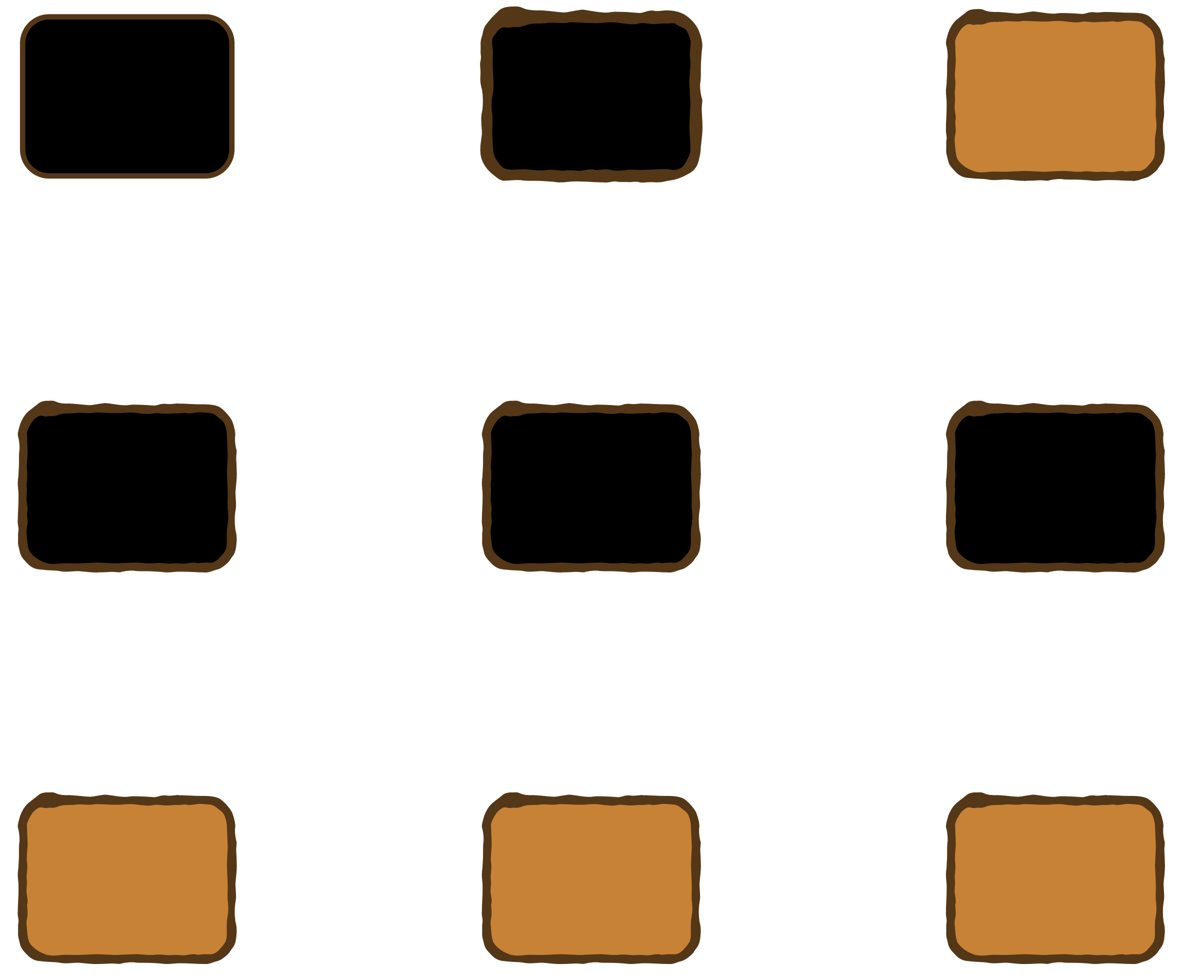
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don’t scale



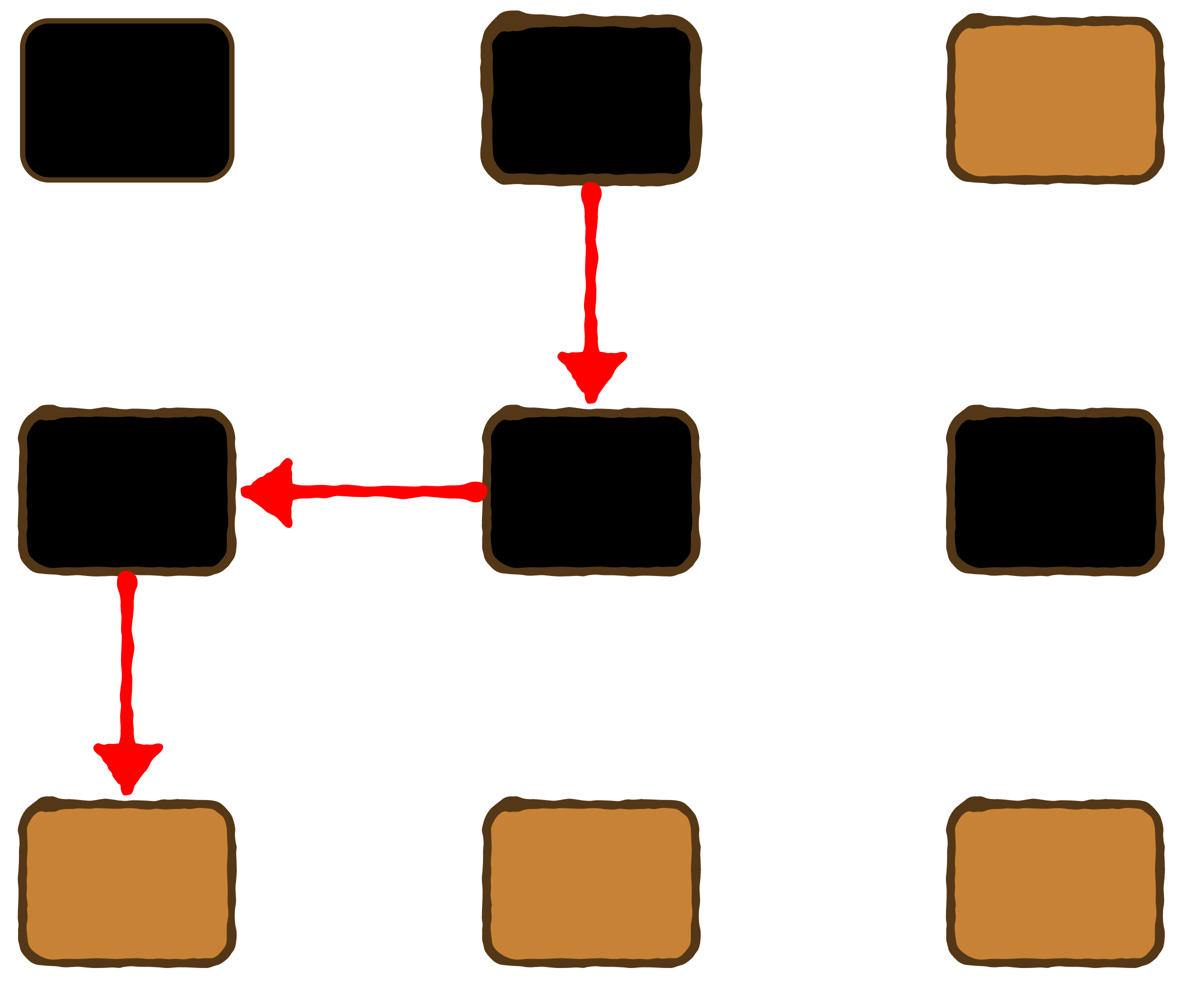
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don’t scale



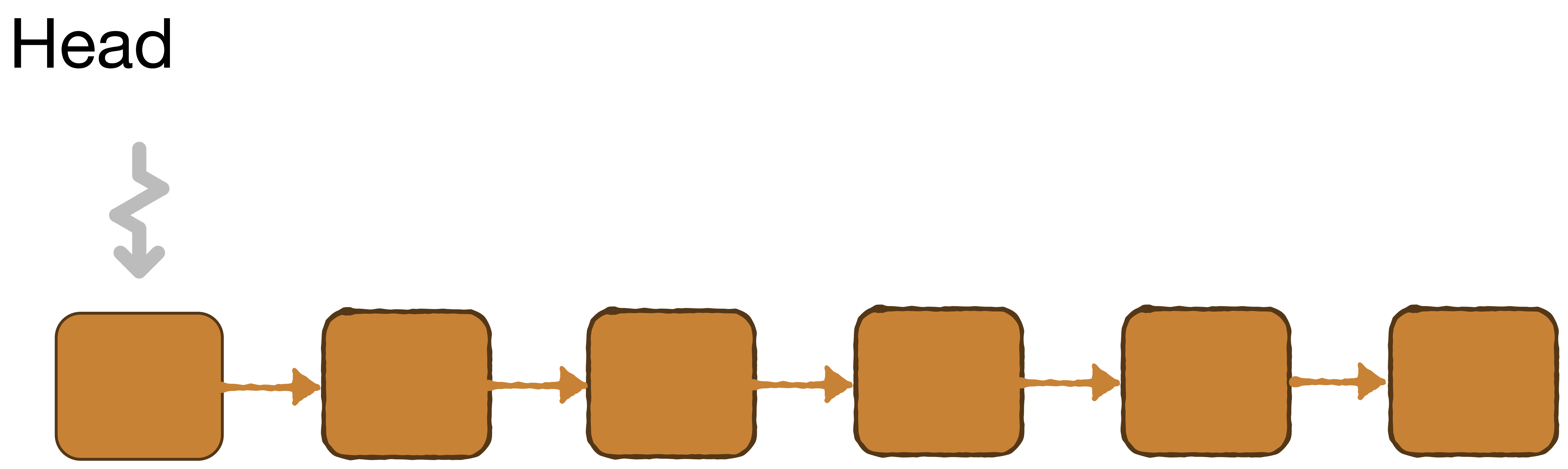
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

2. Random walks are fast but don't scale



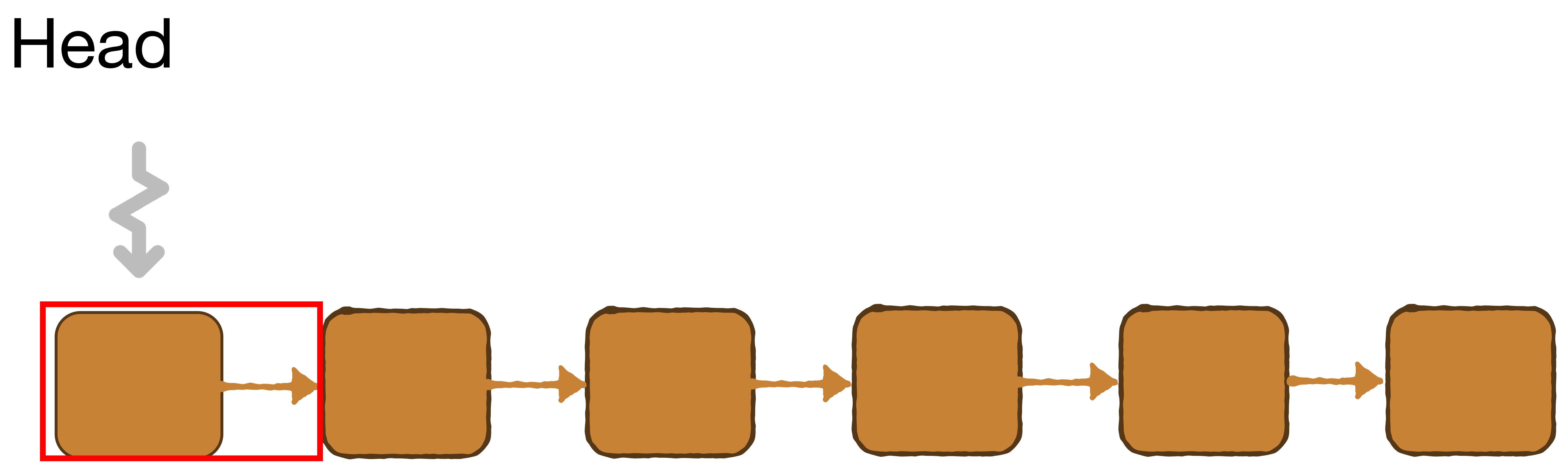
Allocators: Halloc [AP+14], RWMalloc [PLY+22], ScatterAlloc [SKK+12]

3. Queues give fast reuse but limit allocation size



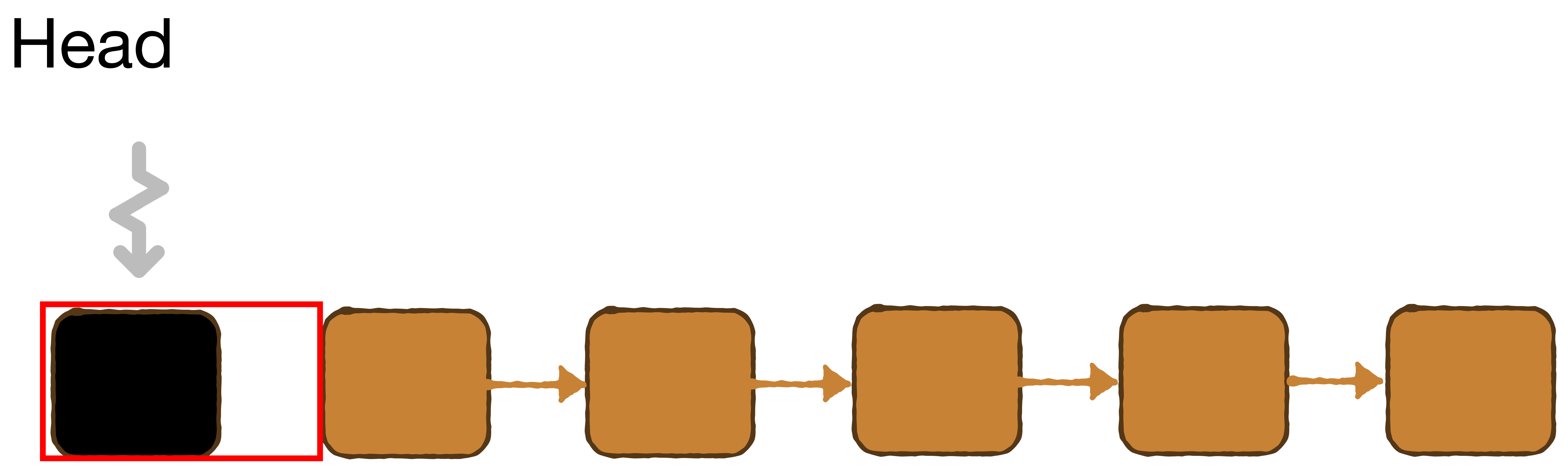
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



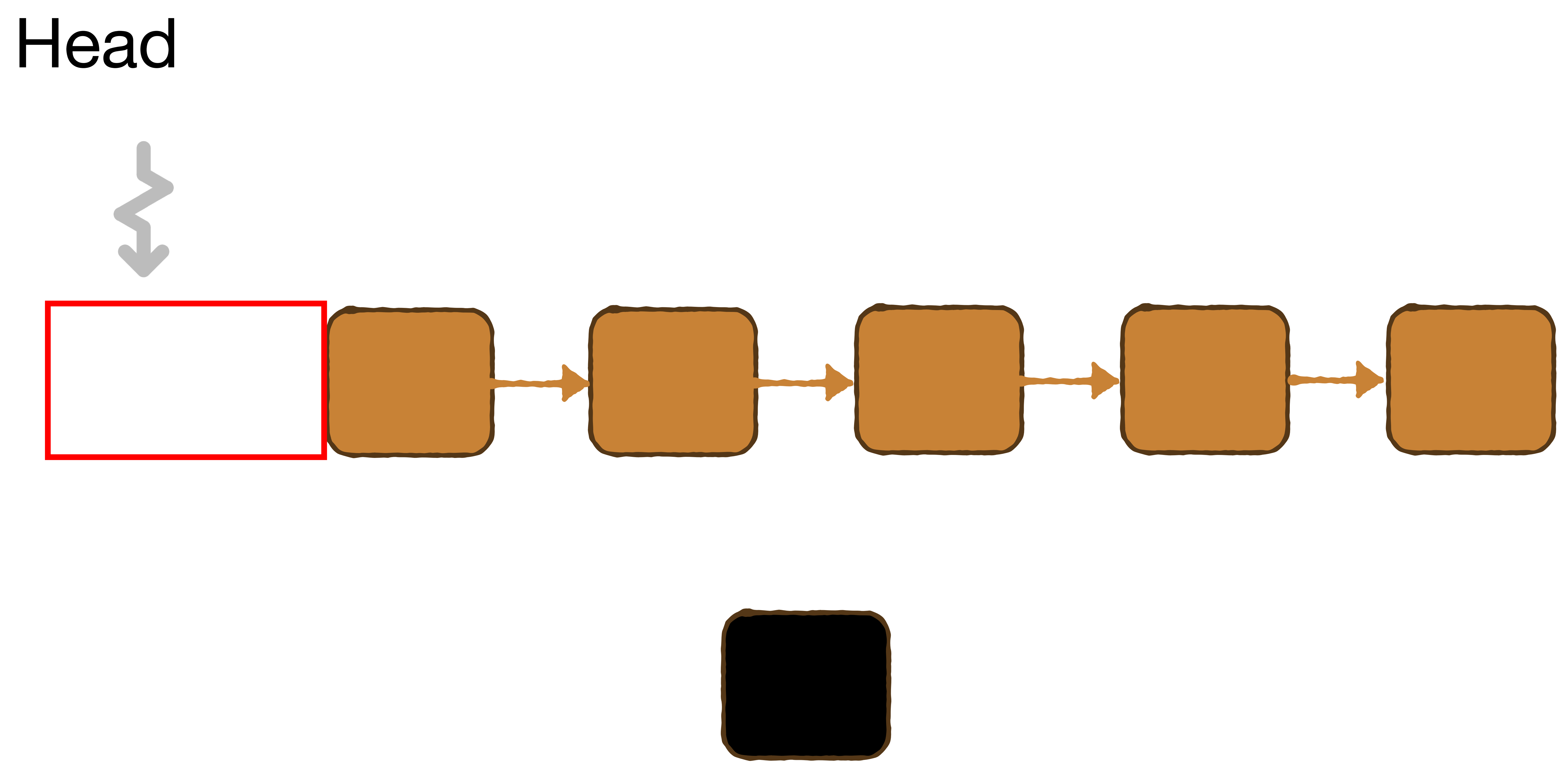
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



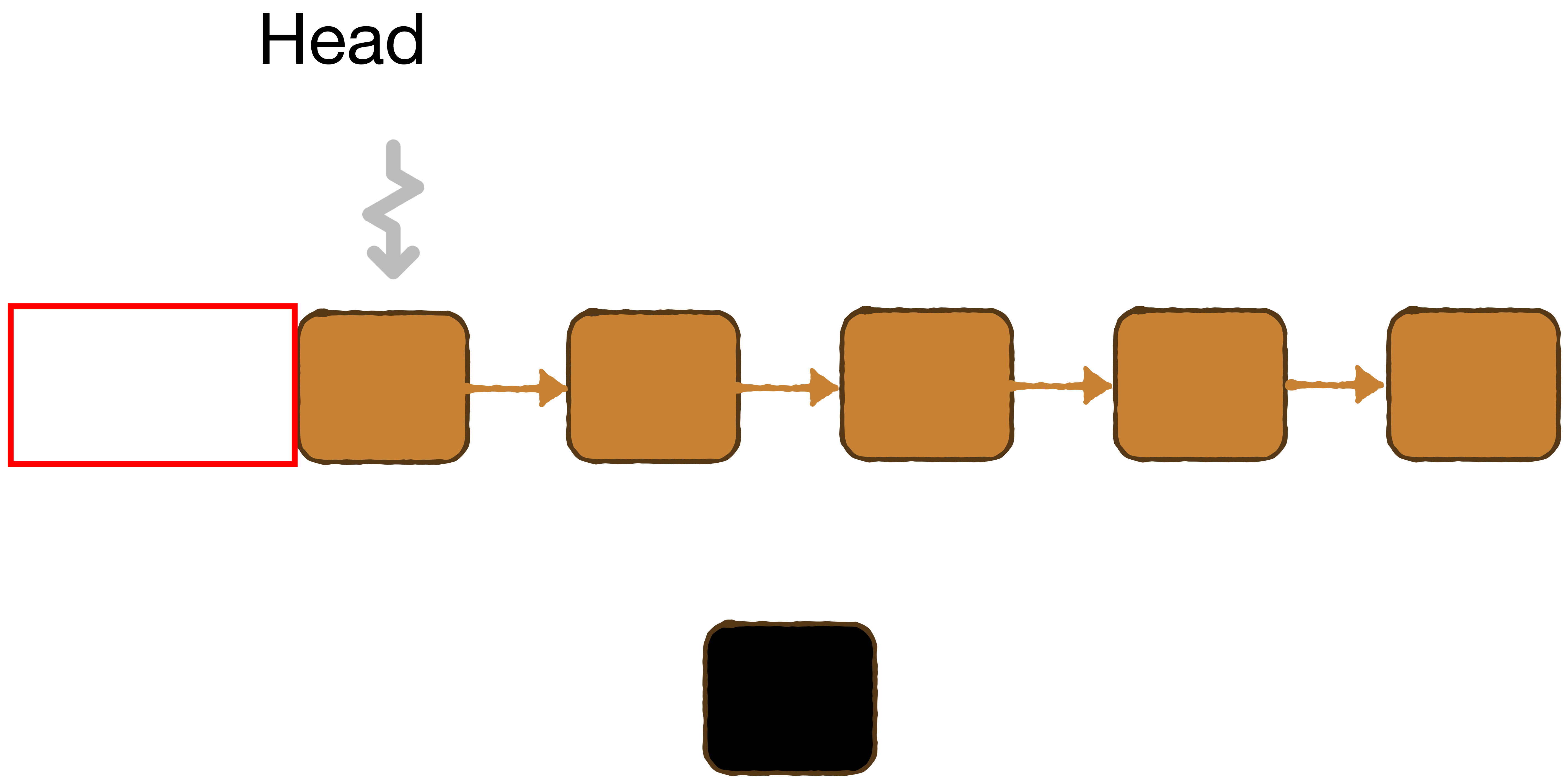
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



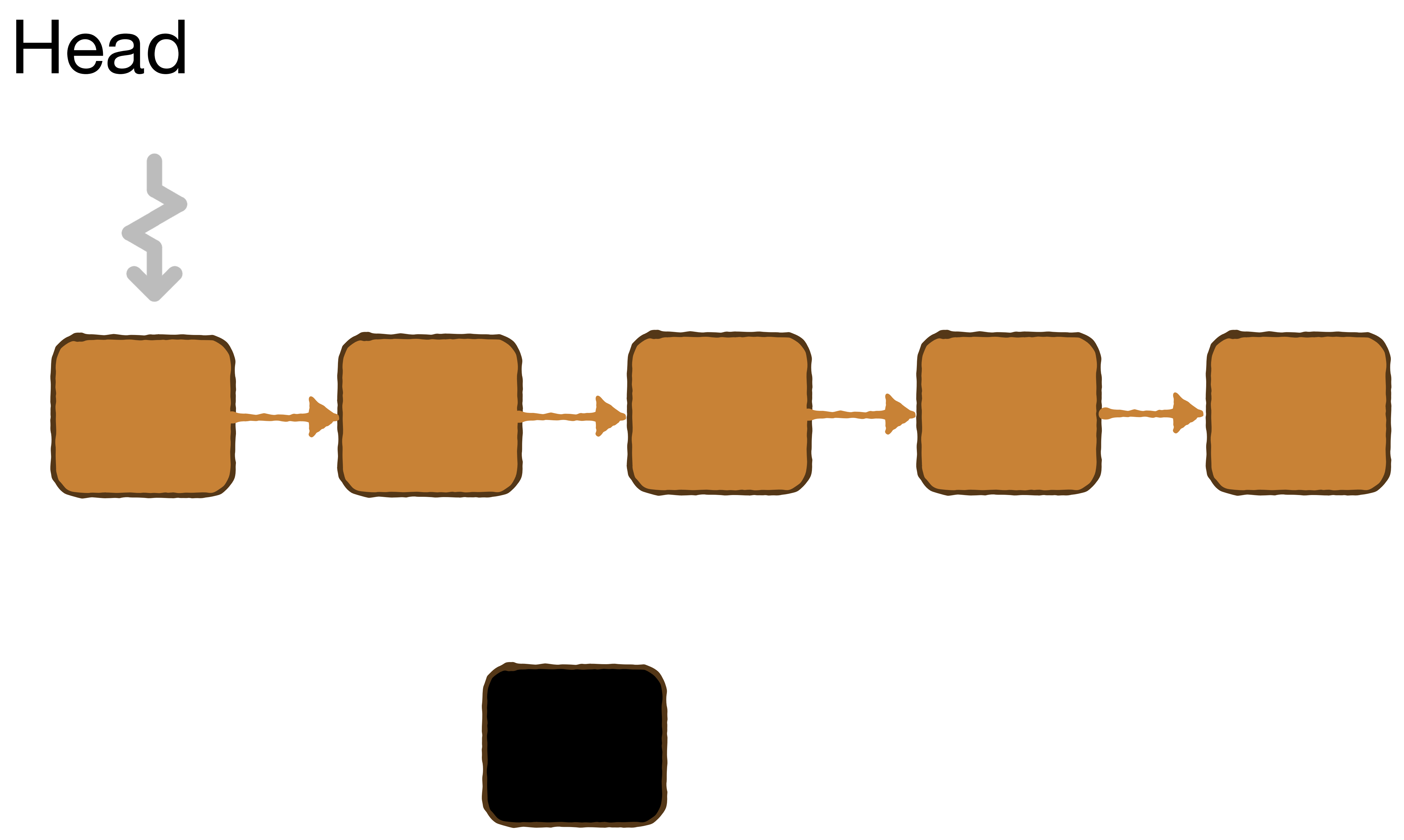
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



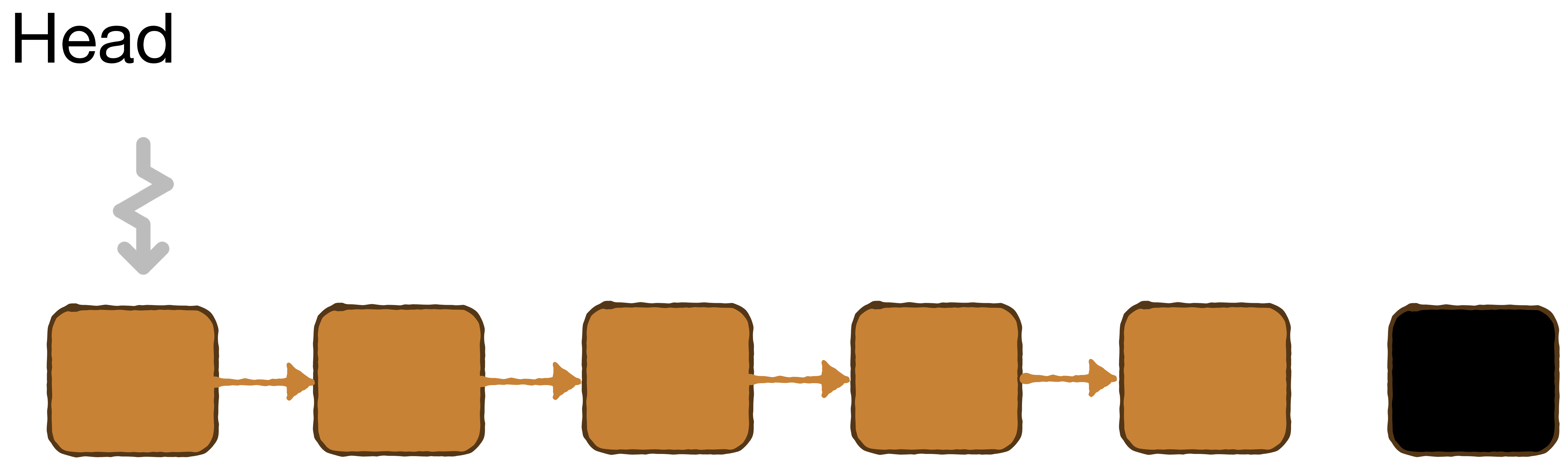
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



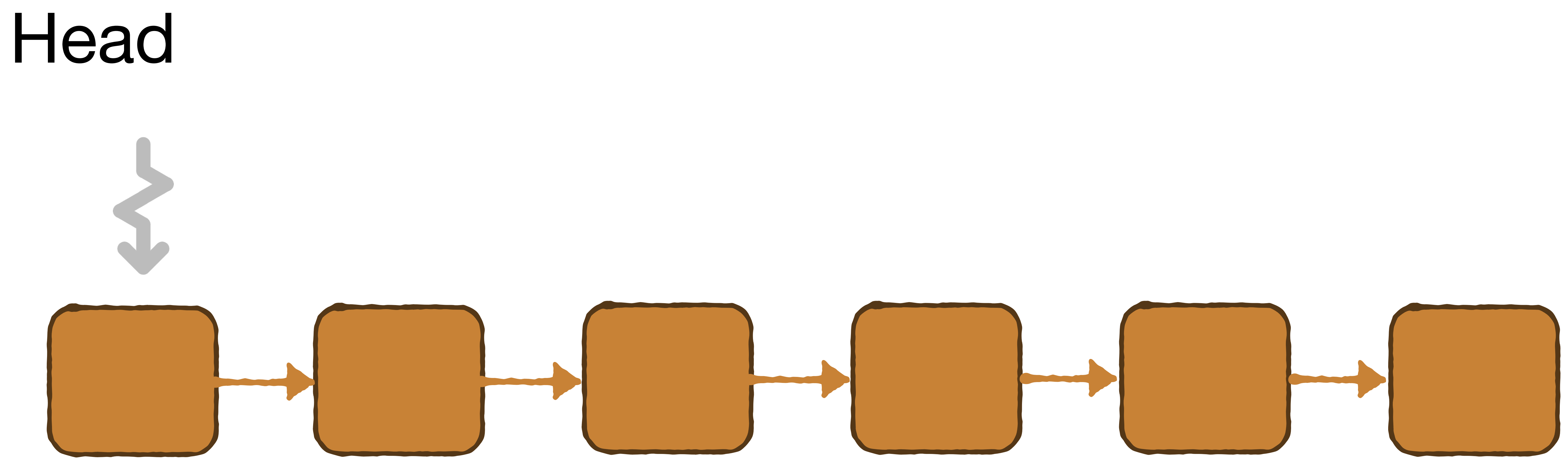
Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



Allocators: Ouroboros [WMP+20]

3. Queues give fast reuse but limit allocation size



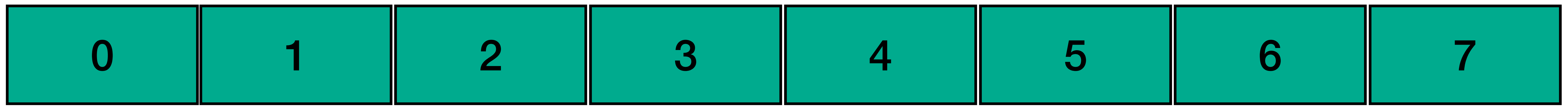
Allocators: Ouroboros [WMP+20]

Memory as an ordered set

Break memory into evenly sized chunks called *segments*

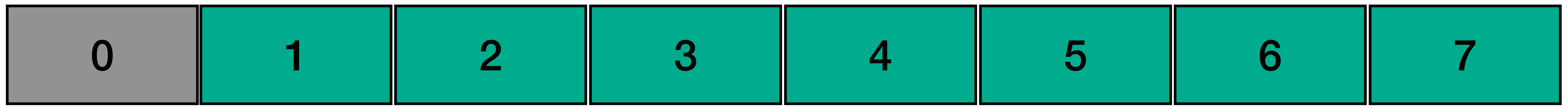
Allocations pull the segment with the smallest ID

- **This minimizes *external fragmentation***



Memory as an ordered set

Break memory into evenly sized chunks called *segments*
Allocations pull the segment with the smallest ID
- **This minimizes *external fragmentation***

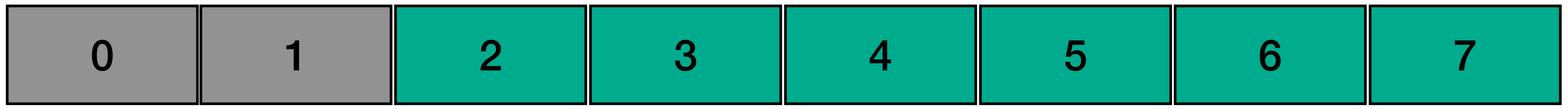


Memory as an ordered set

Break memory into evenly sized chunks called *segments*

Allocations pull the segment with the smallest ID

- **This minimizes *external fragmentation***

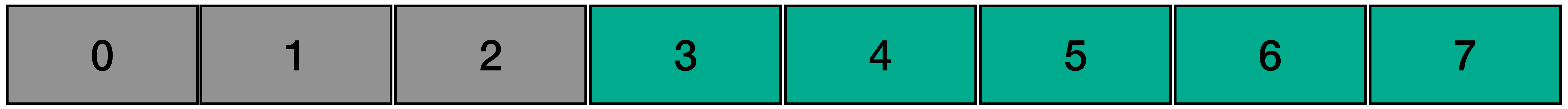


Memory as an ordered set

Break memory into evenly sized chunks called *segments*

Allocations pull the segment with the smallest ID

- **This minimizes *external fragmentation***

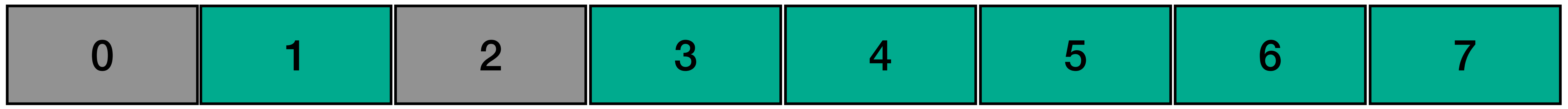


Memory as an ordered set

Break memory into evenly sized chunks called *segments*

Allocations pull the segment with the smallest ID

- **This minimizes *external fragmentation***

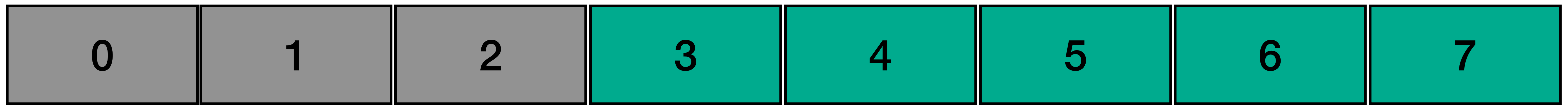


Memory as an ordered set

Break memory into evenly sized chunks called *segments*

Allocations pull the segment with the smallest ID

- **This minimizes *external fragmentation***



What data structure can represent this?

Insertions

Deletions

Ordered search

Highly concurrent!

van Emde Boas tree [van Emde Boas 1977]

Uses O(|*U*|) space for a universe *U*

O(log log |*U*|) insertion, deletion, and successor search

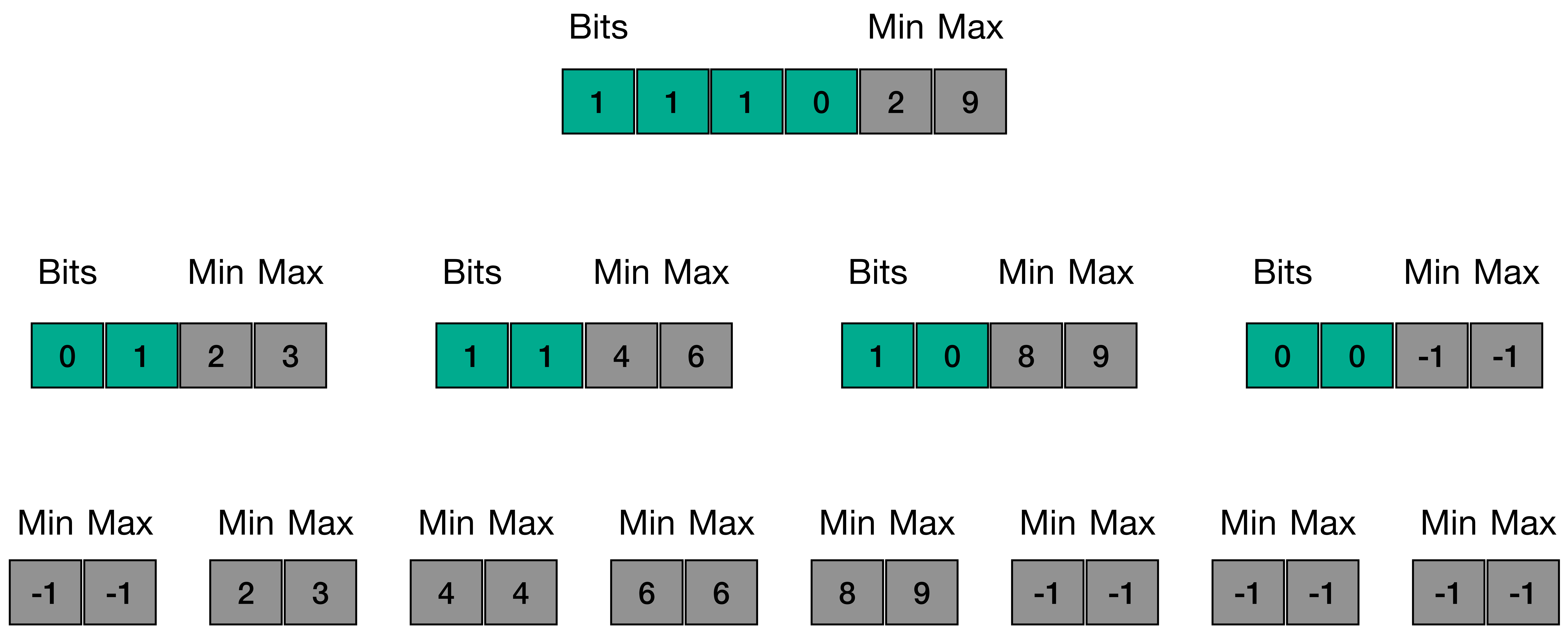
Each node contains:

- **sqrt(|*U*|)-size bitmap representing children**
- **Minimum over the whole subtree**
- **Maximum over the whole subtree**

vEB tree design

Universe: 16 items (0,15)

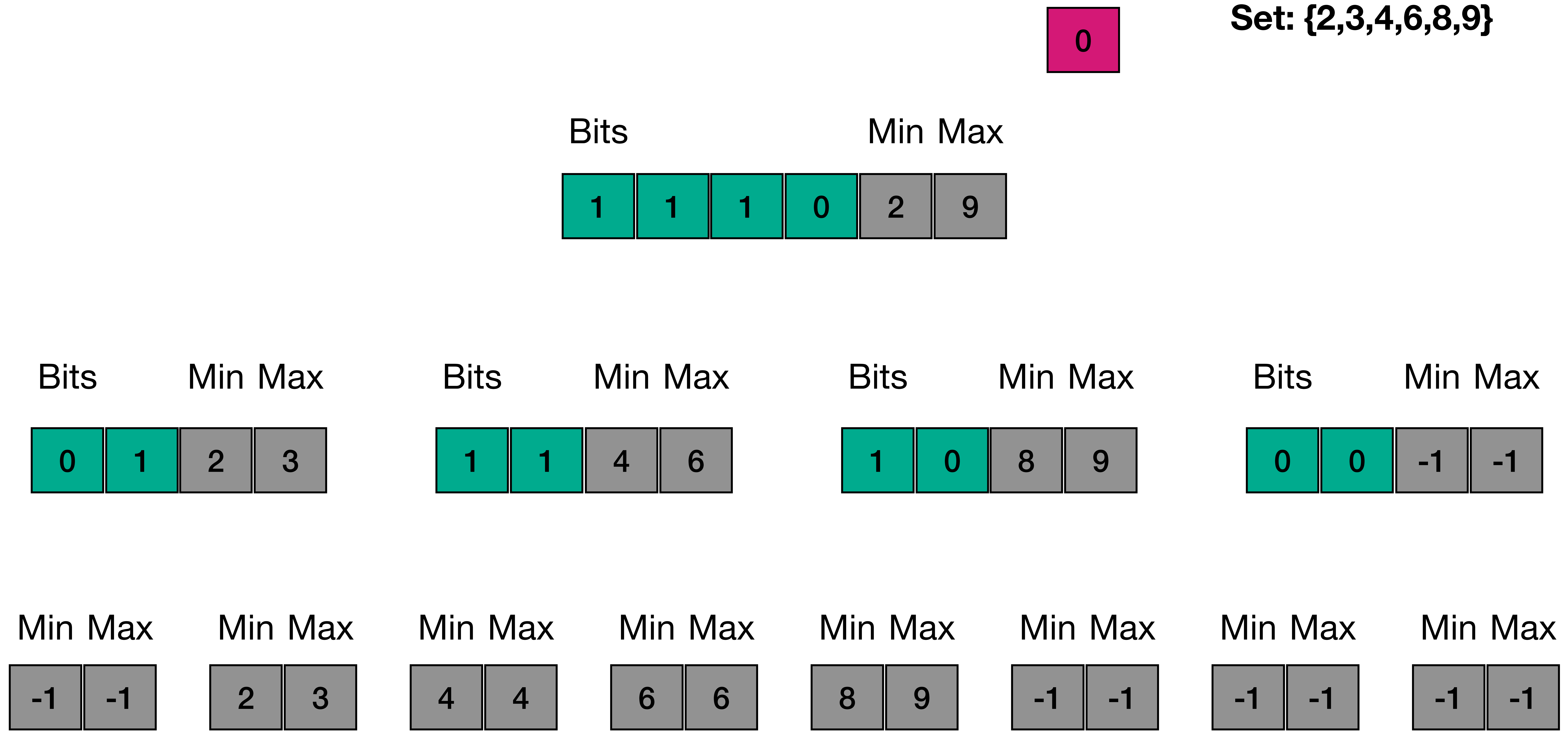
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

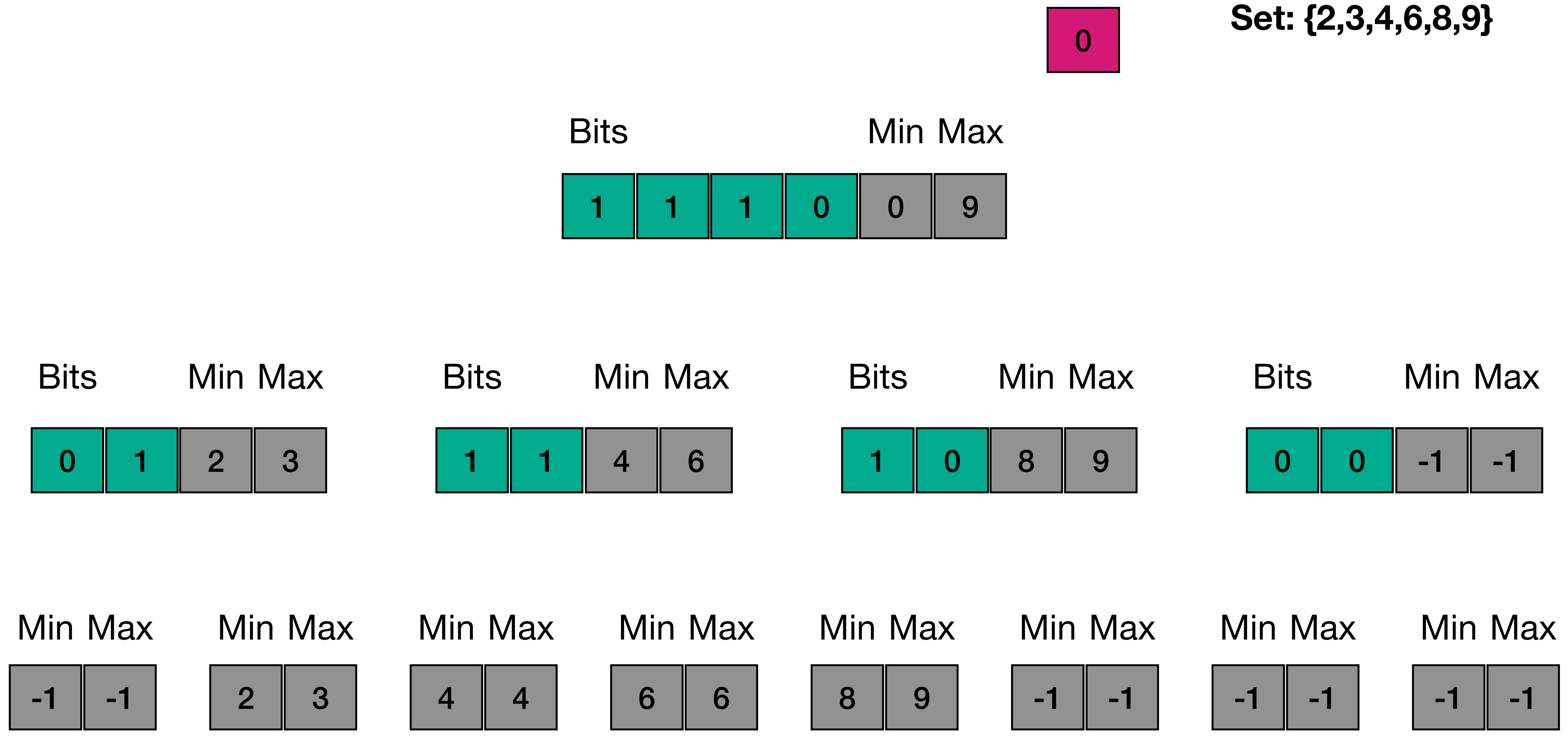
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

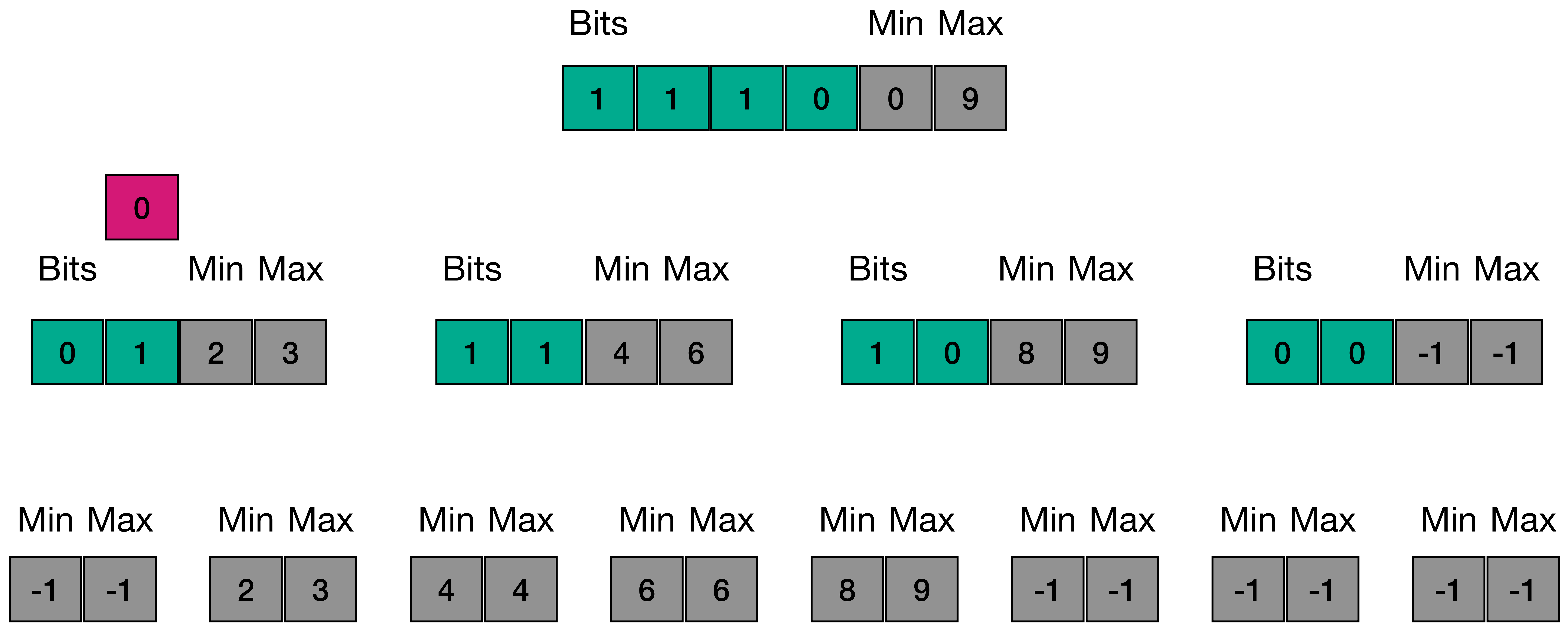
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

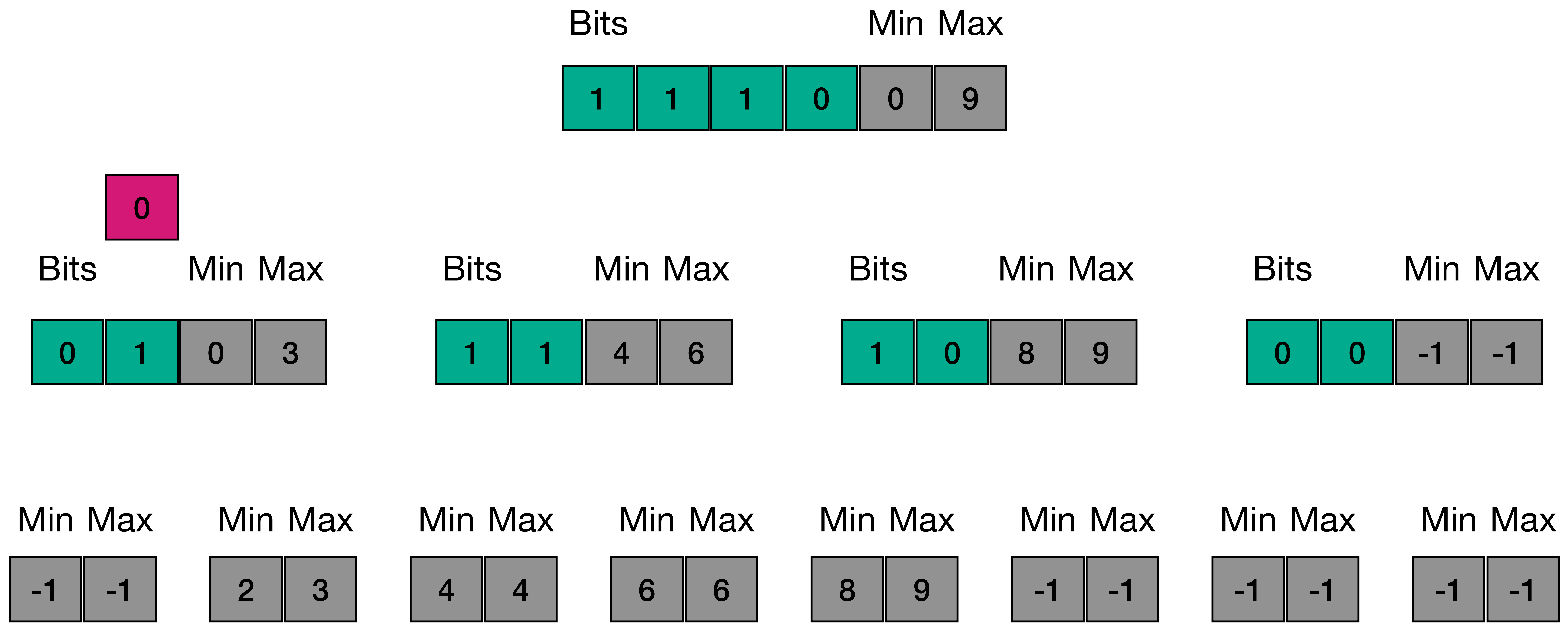
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

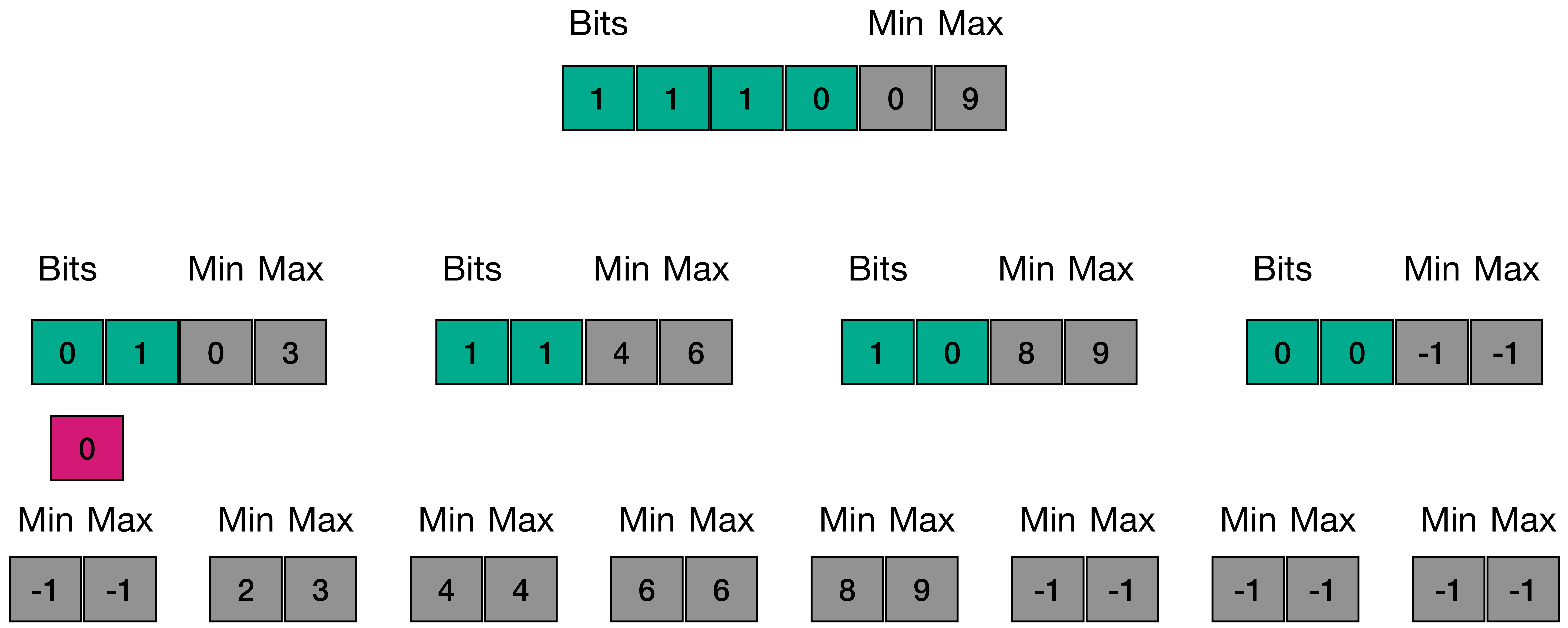
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

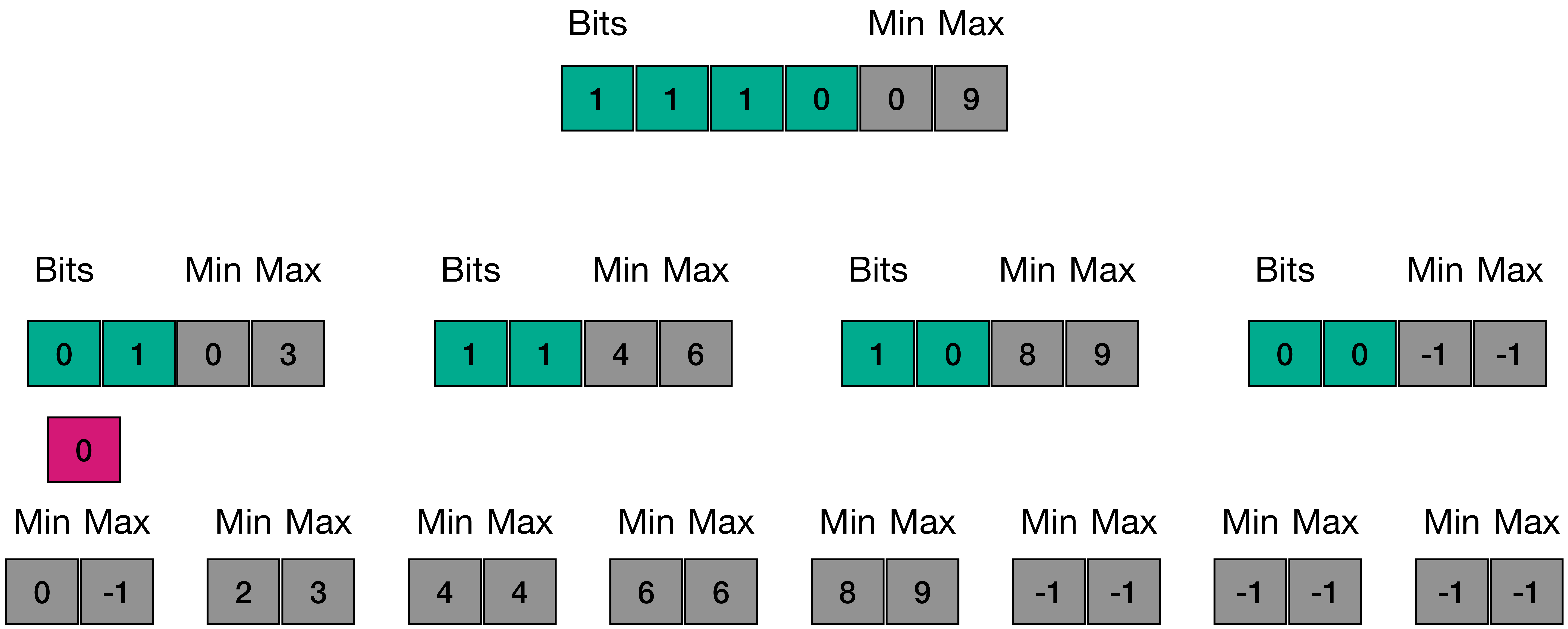
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

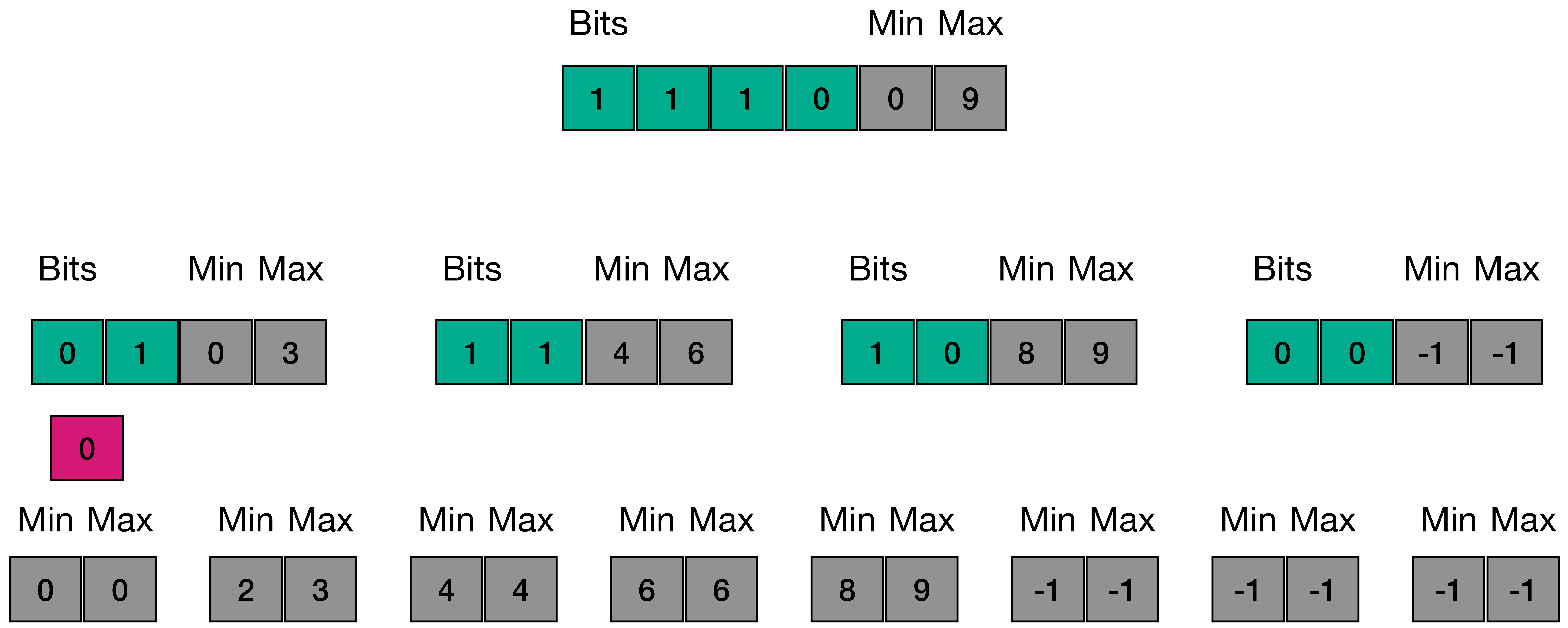
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

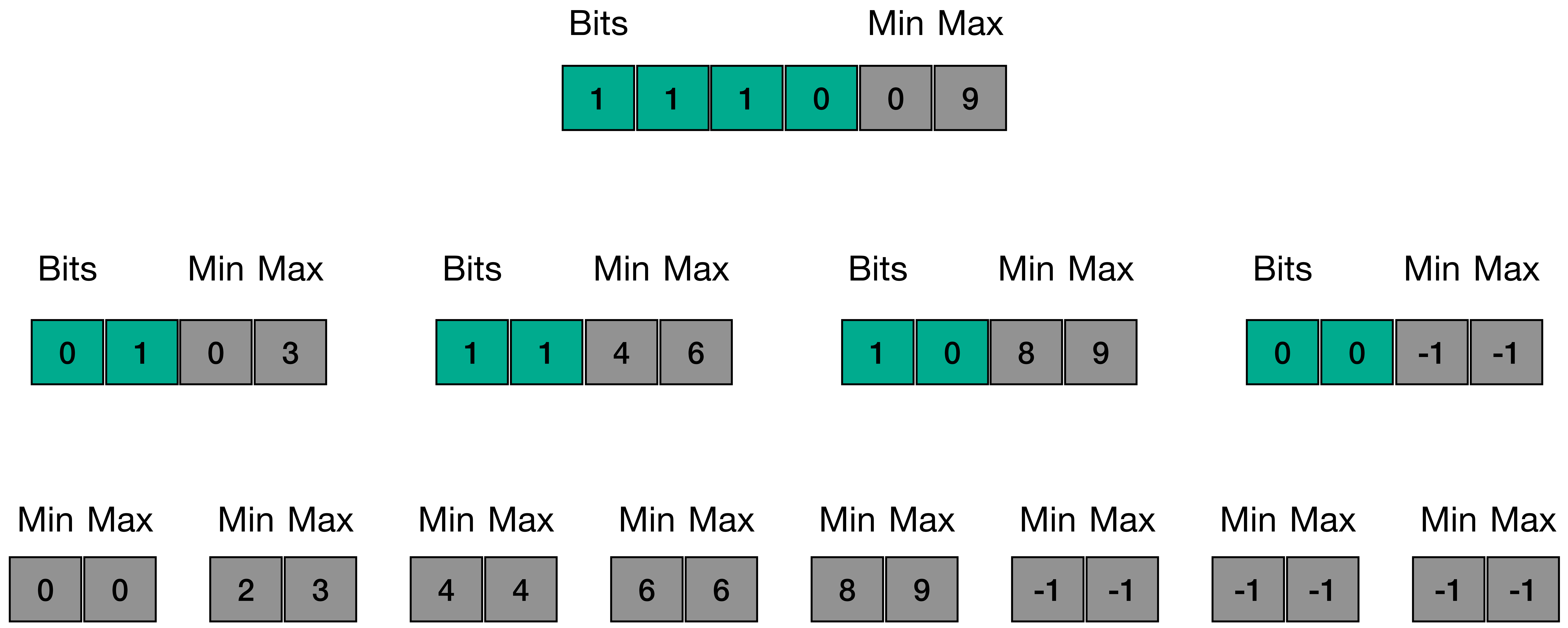
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

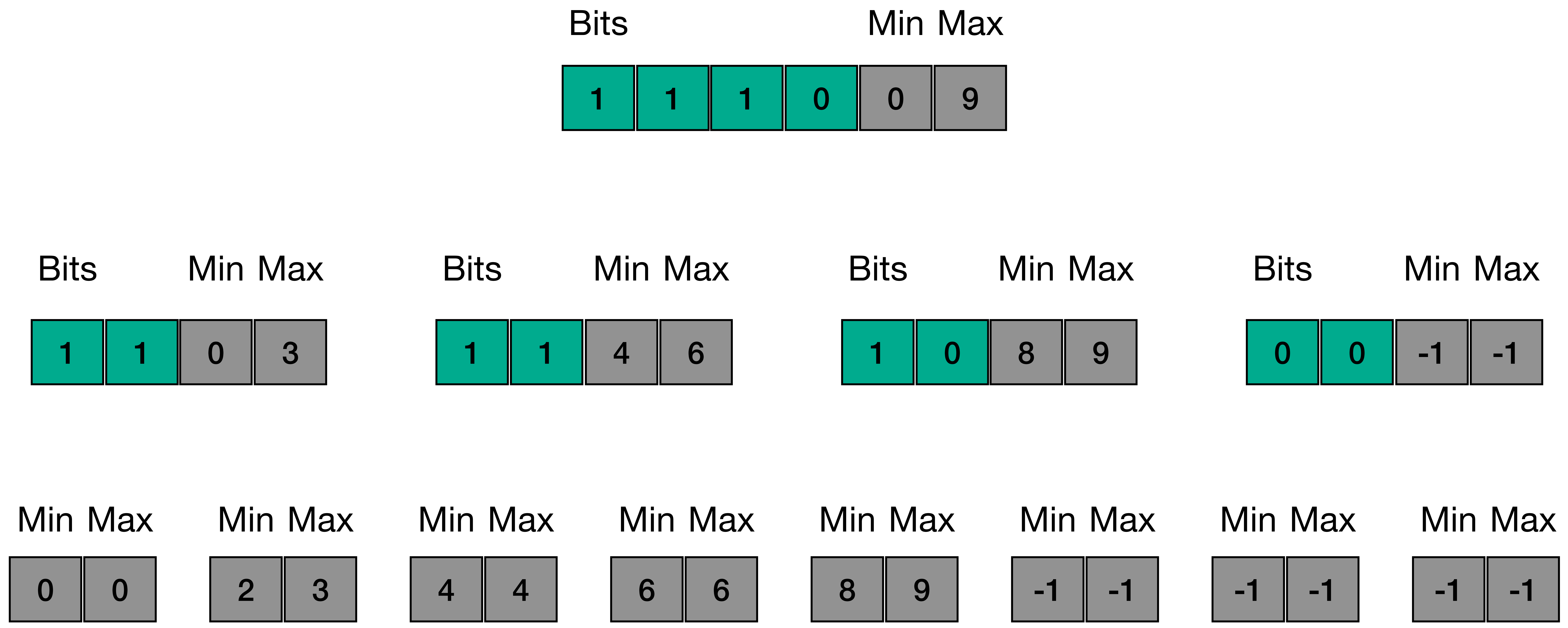
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

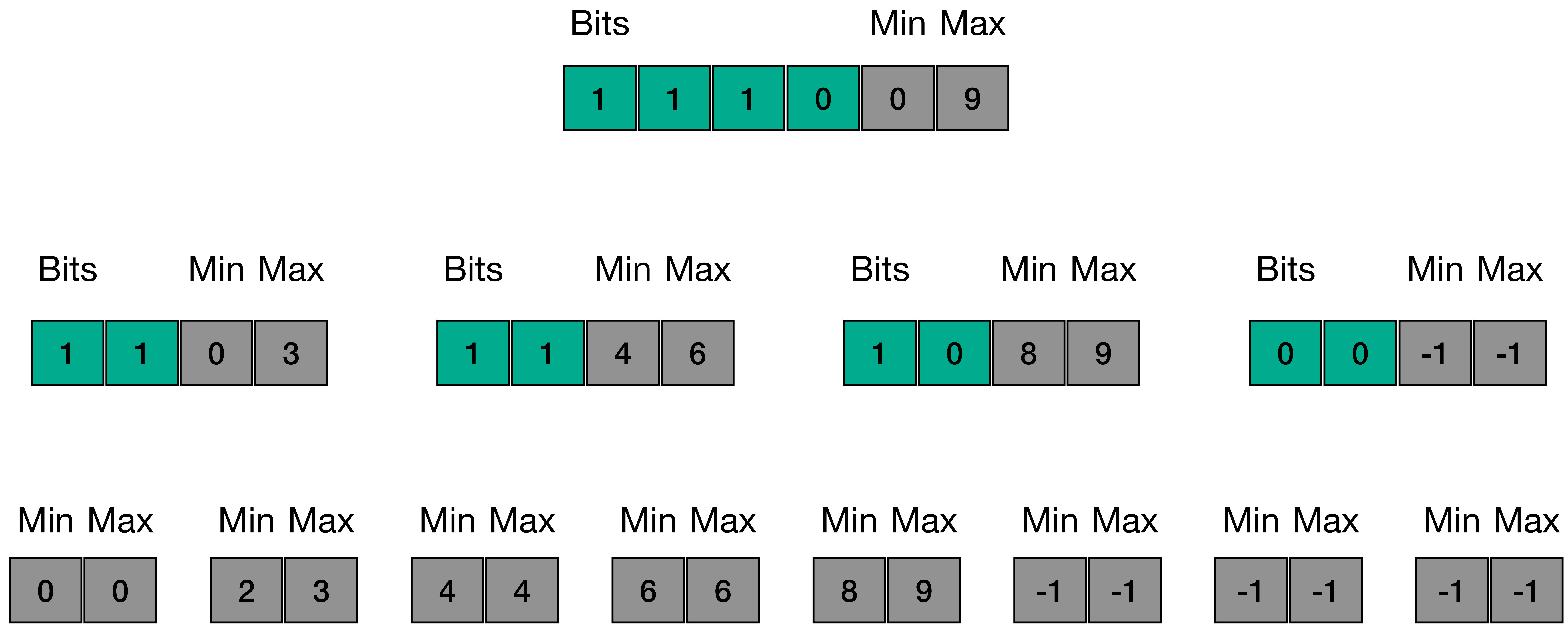
Set: {2,3,4,6,8,9}



vEB tree design

Universe: 16 items (0,15)

Set: {0,2,3,4,6,8,9}



GPU specific modifications

Problem: Nodes can be too large to use atomics

GPU specific modifications

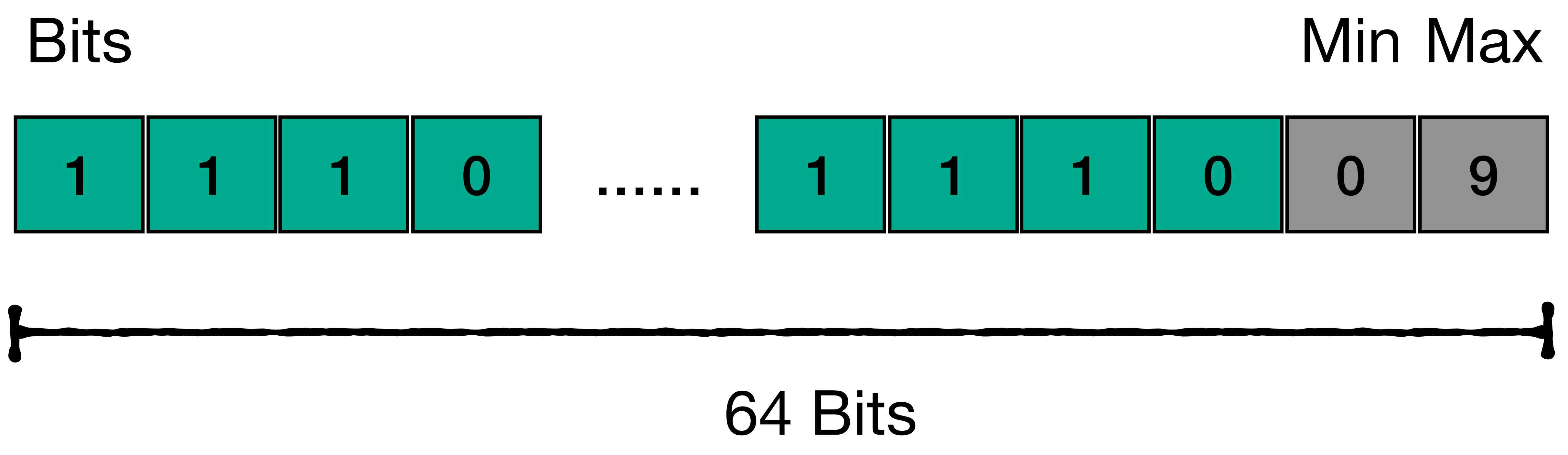
Problem: Nodes can be too large to use atomics

Solution: Cap node size to 64 bits

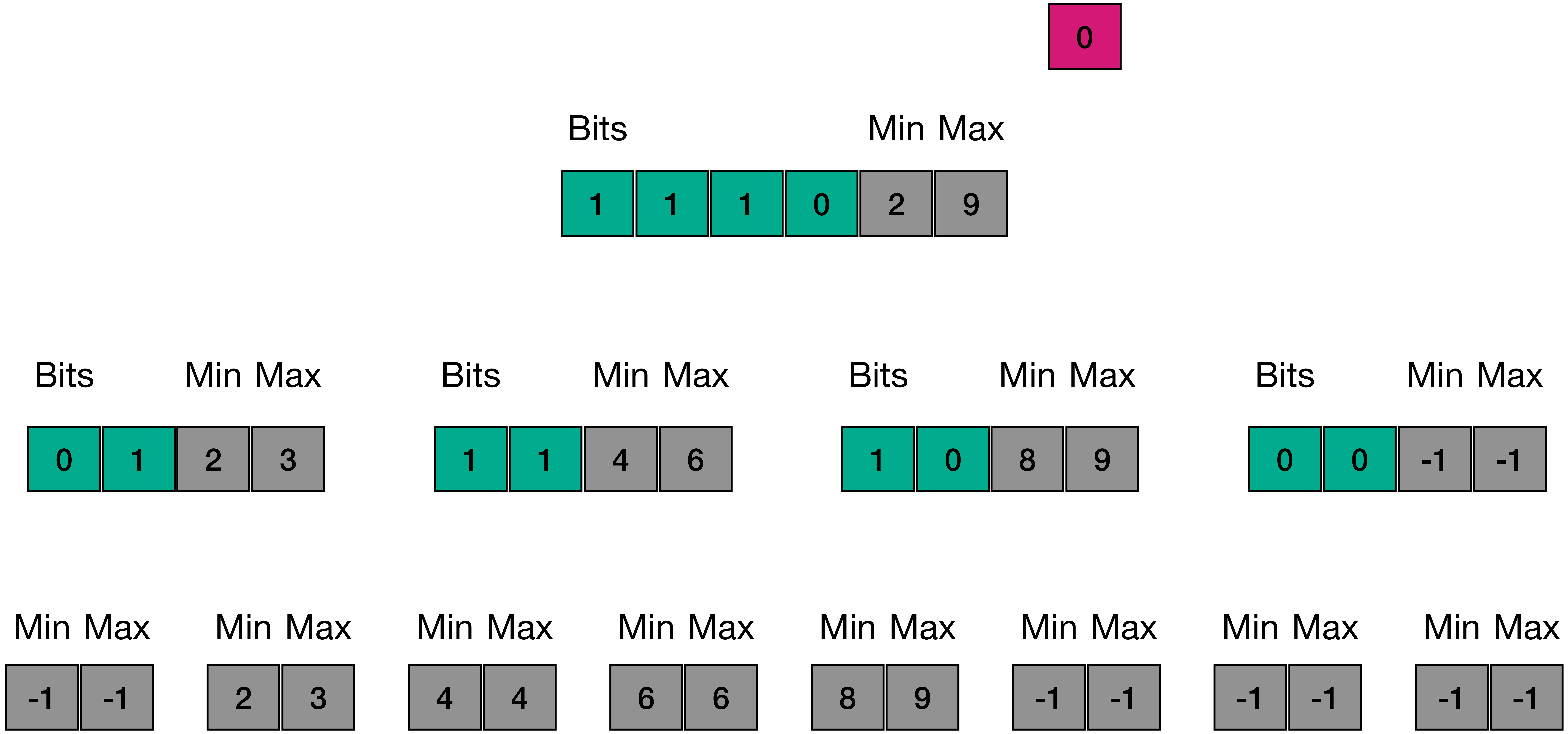
GPU specific modifications

Problem: Nodes can be too large to use atomics

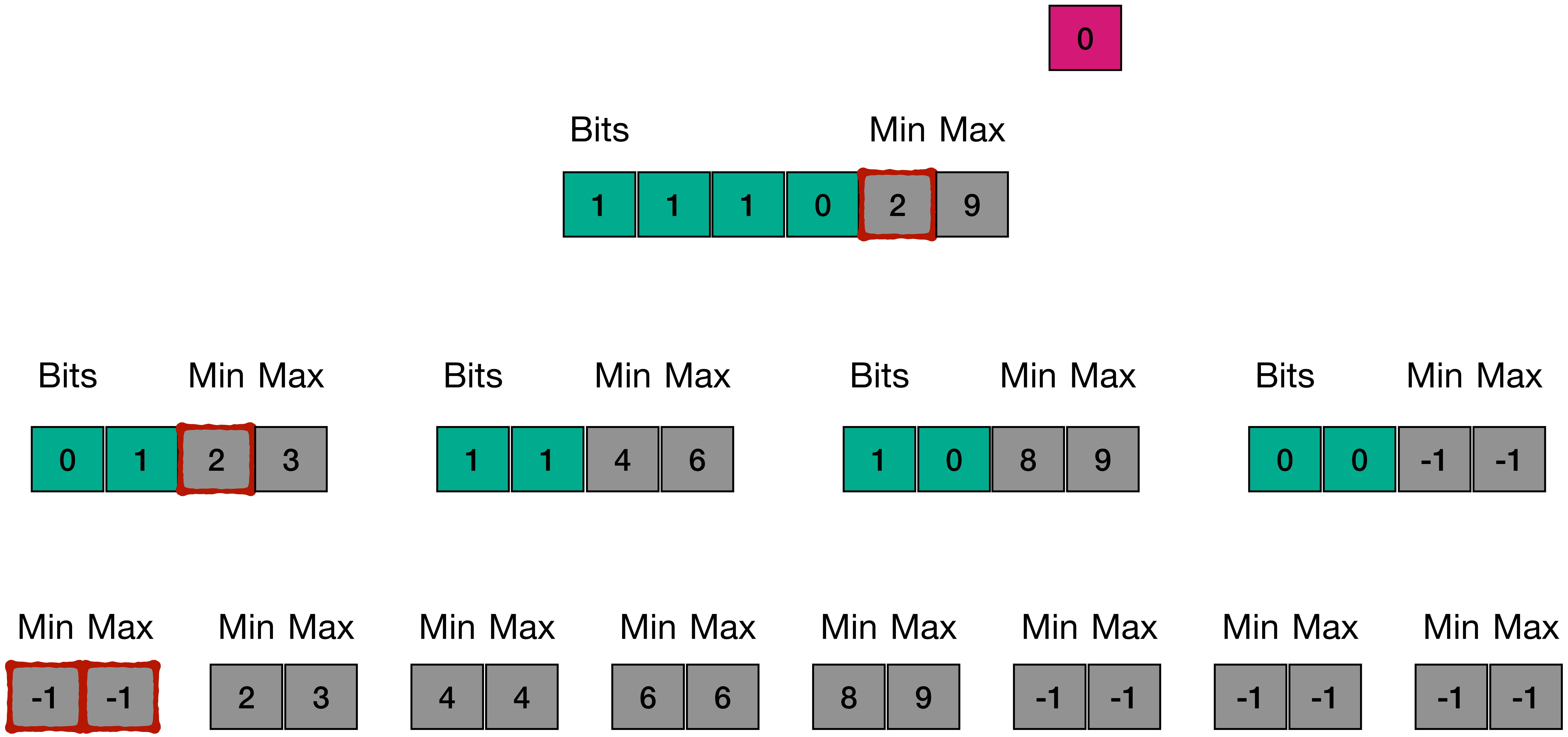
Solution: Cap node size to 64 bits



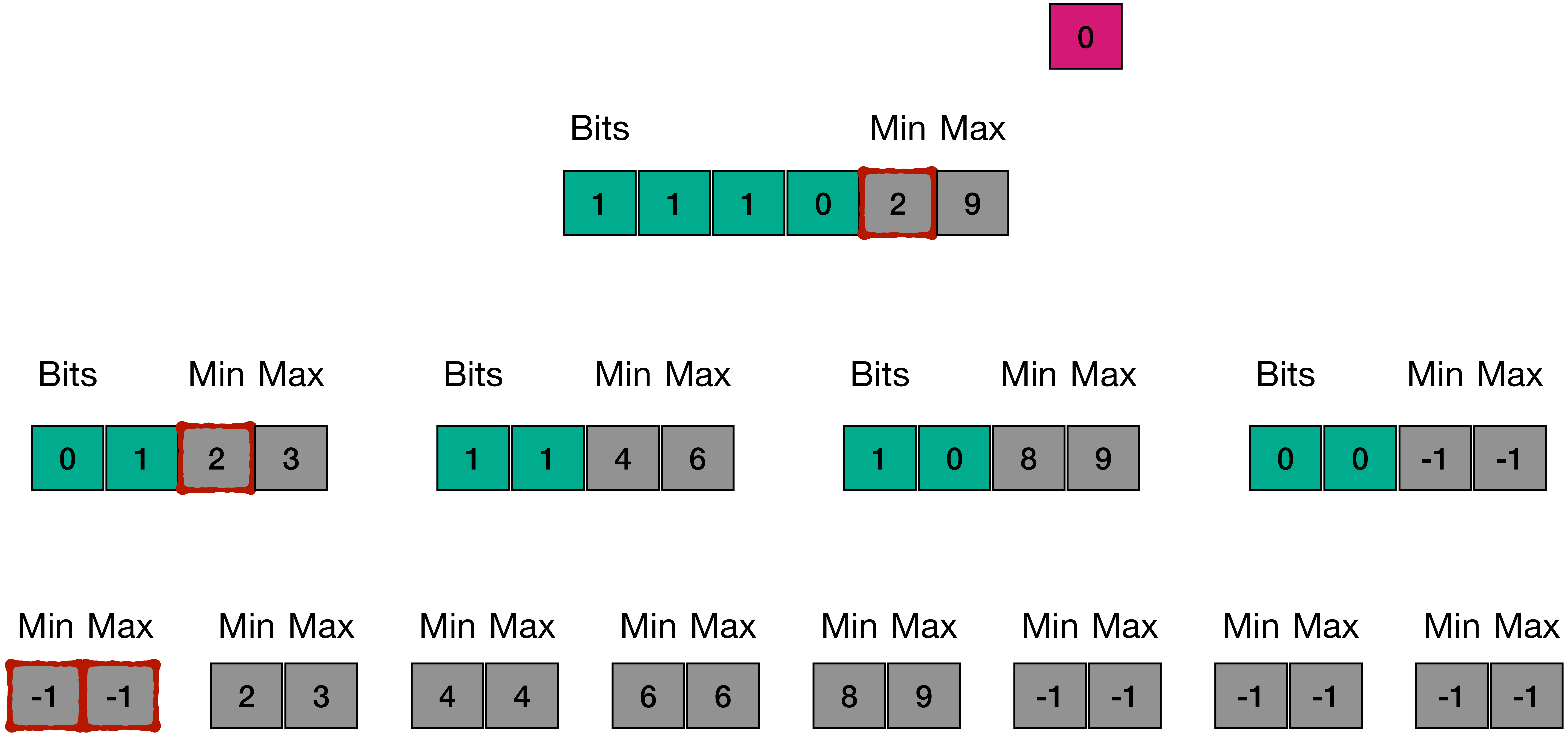
Updates to min/max are expensive



Updates to min/max are expensive



Updates to min/max are expensive



Updating global minimum updates at **every** level!

GPU specific modifications

Problem: Multiple node updates during insertions/deletions

GPU specific modifications

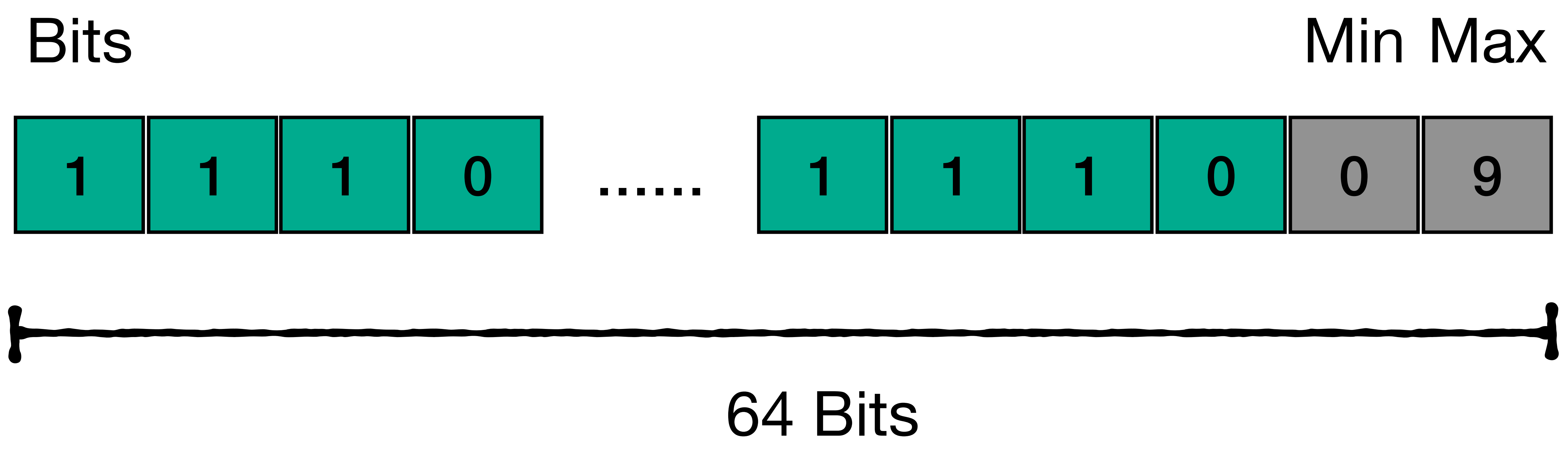
Problem: Multiple node updates during insertions/deletions

Solution: Make min/max node local

GPU specific modifications

Problem: Multiple node updates during insertions/deletions

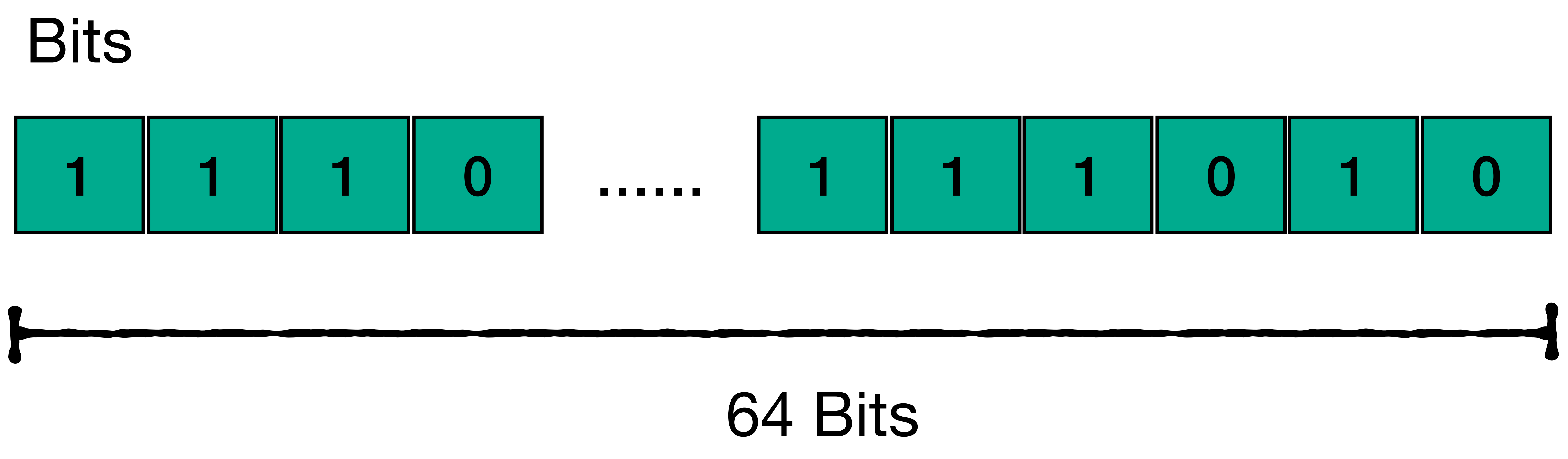
Solution: Make min/max node local



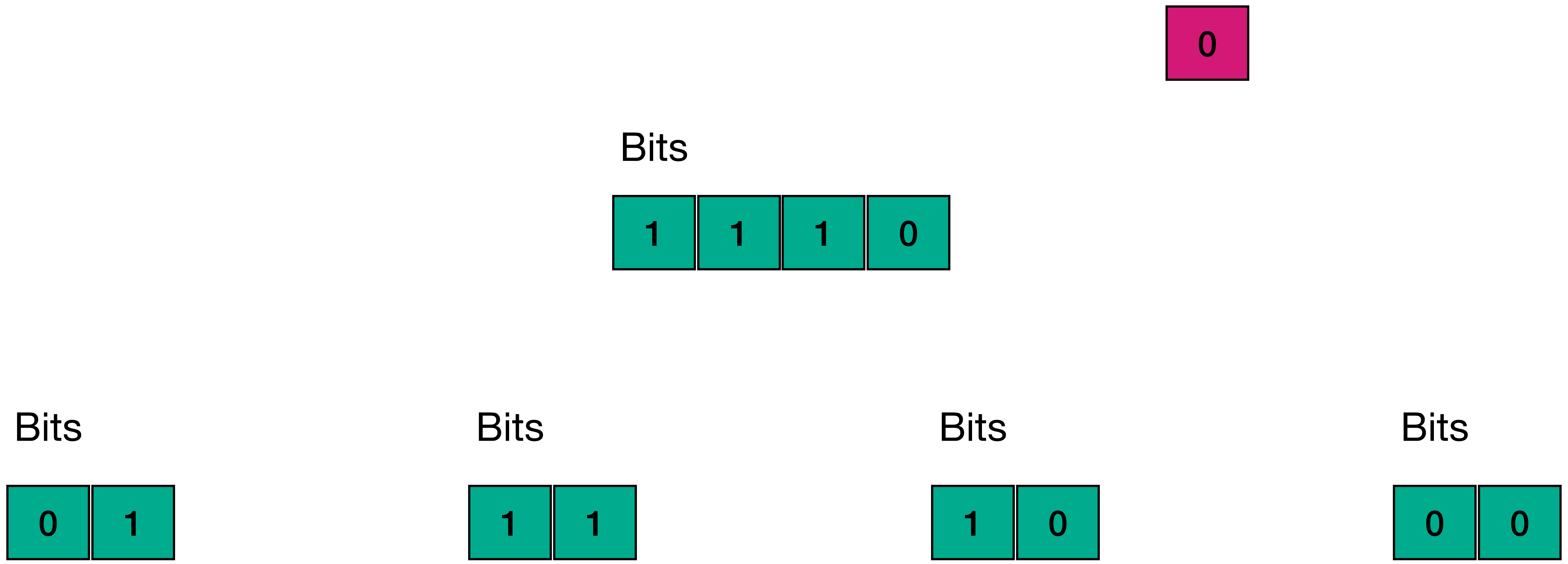
GPU specific modifications

Problem: Multiple node updates during insertions/deletions

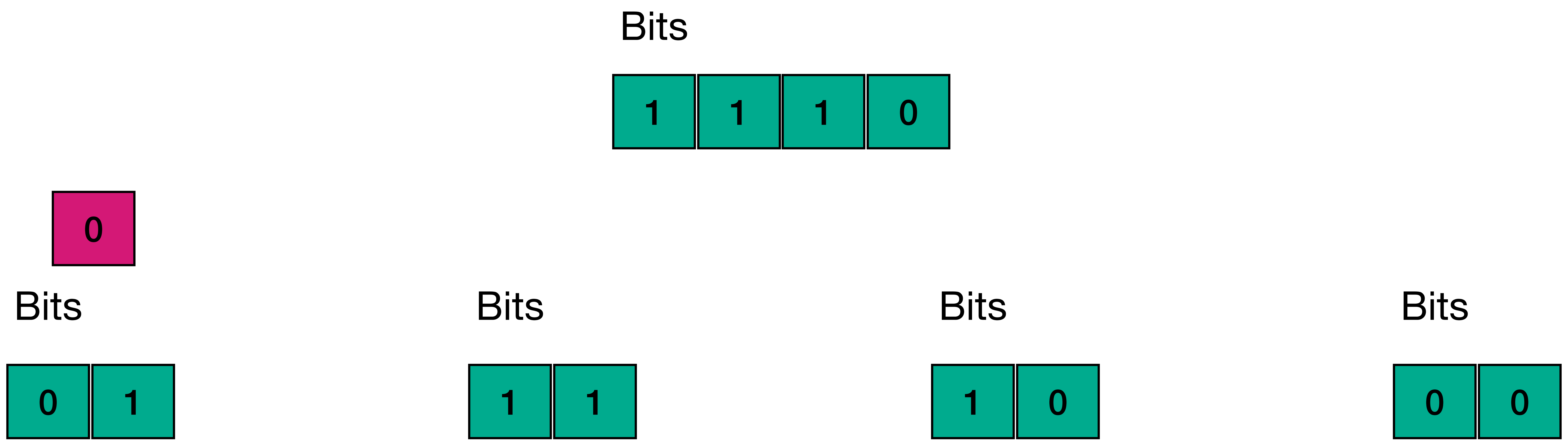
Solution: Make min/max node local



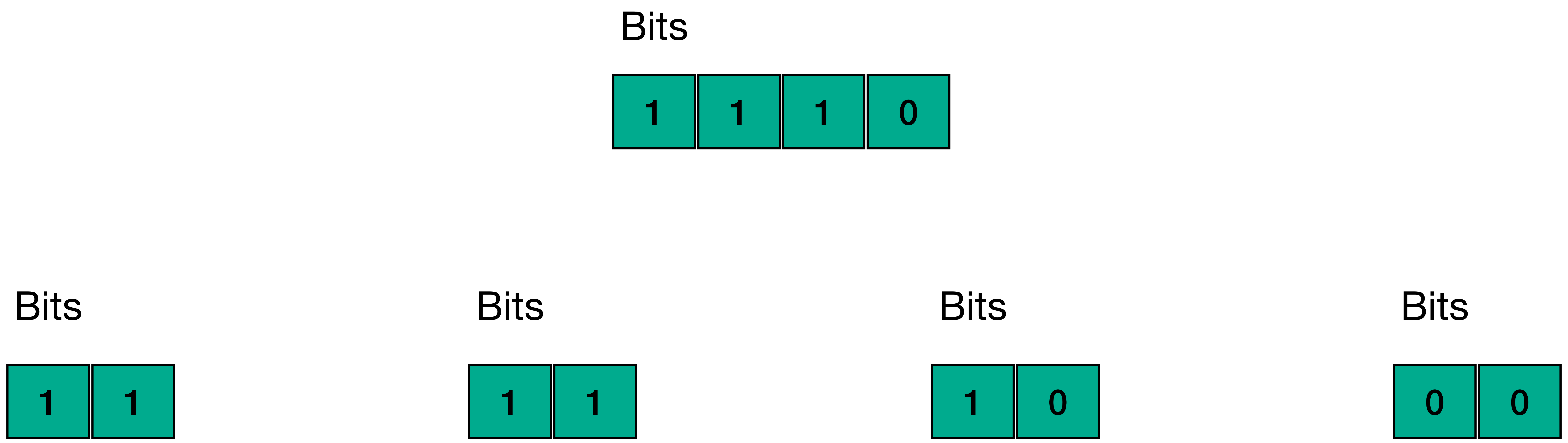
Final vEB design



Final vEB design

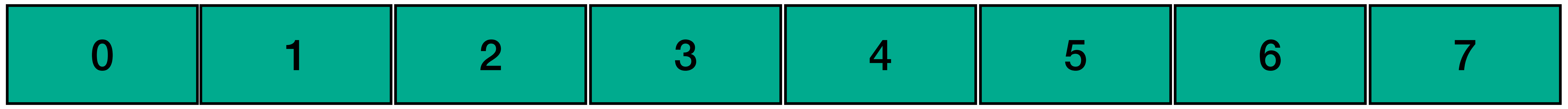


Final vEB design

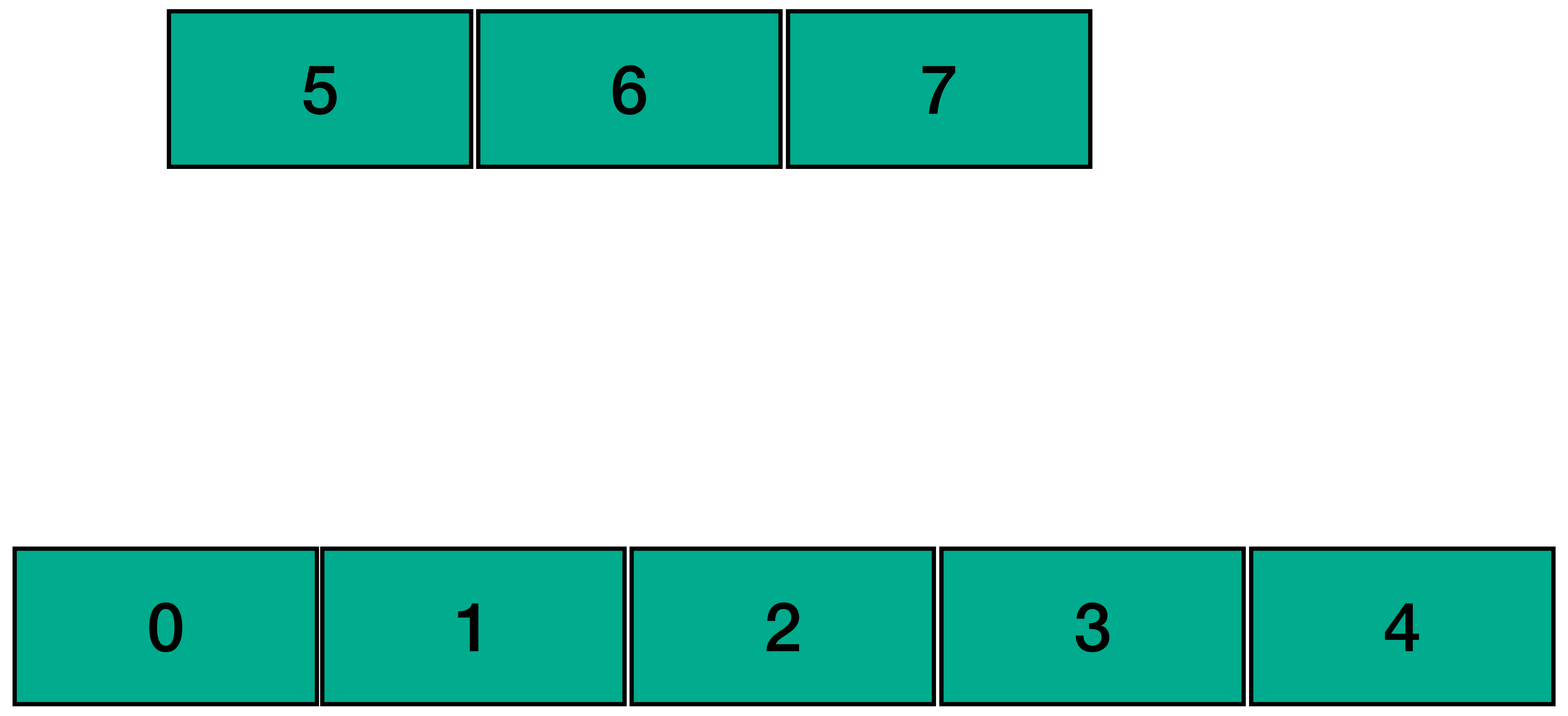


Handling arbitrary allocation sizes

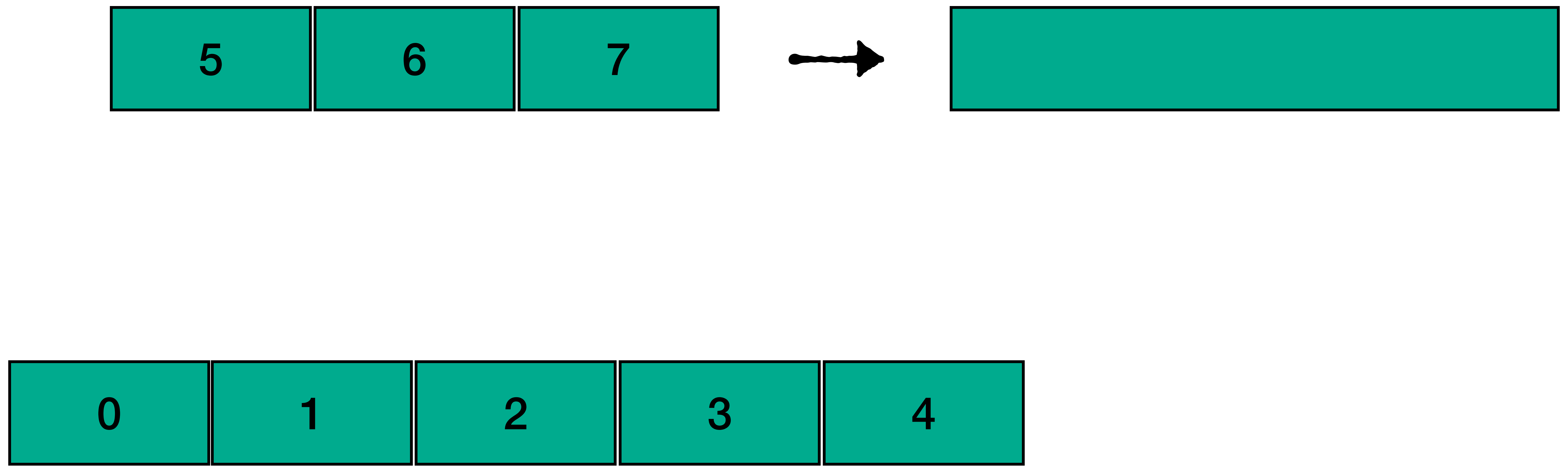
Larger allocation sizes come from the back of the memory



Larger allocation sizes come from the back of the memory



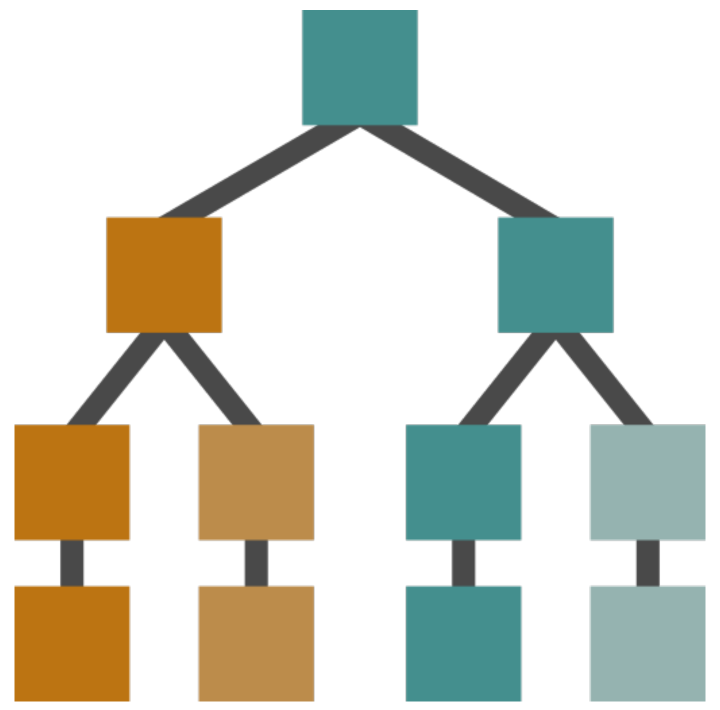
Larger allocation sizes come from the back of the memory

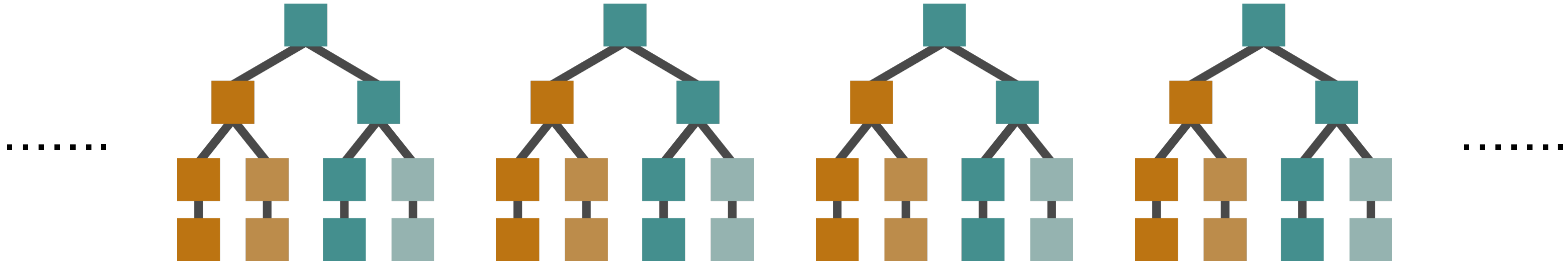


Segments are broken into smaller allocations

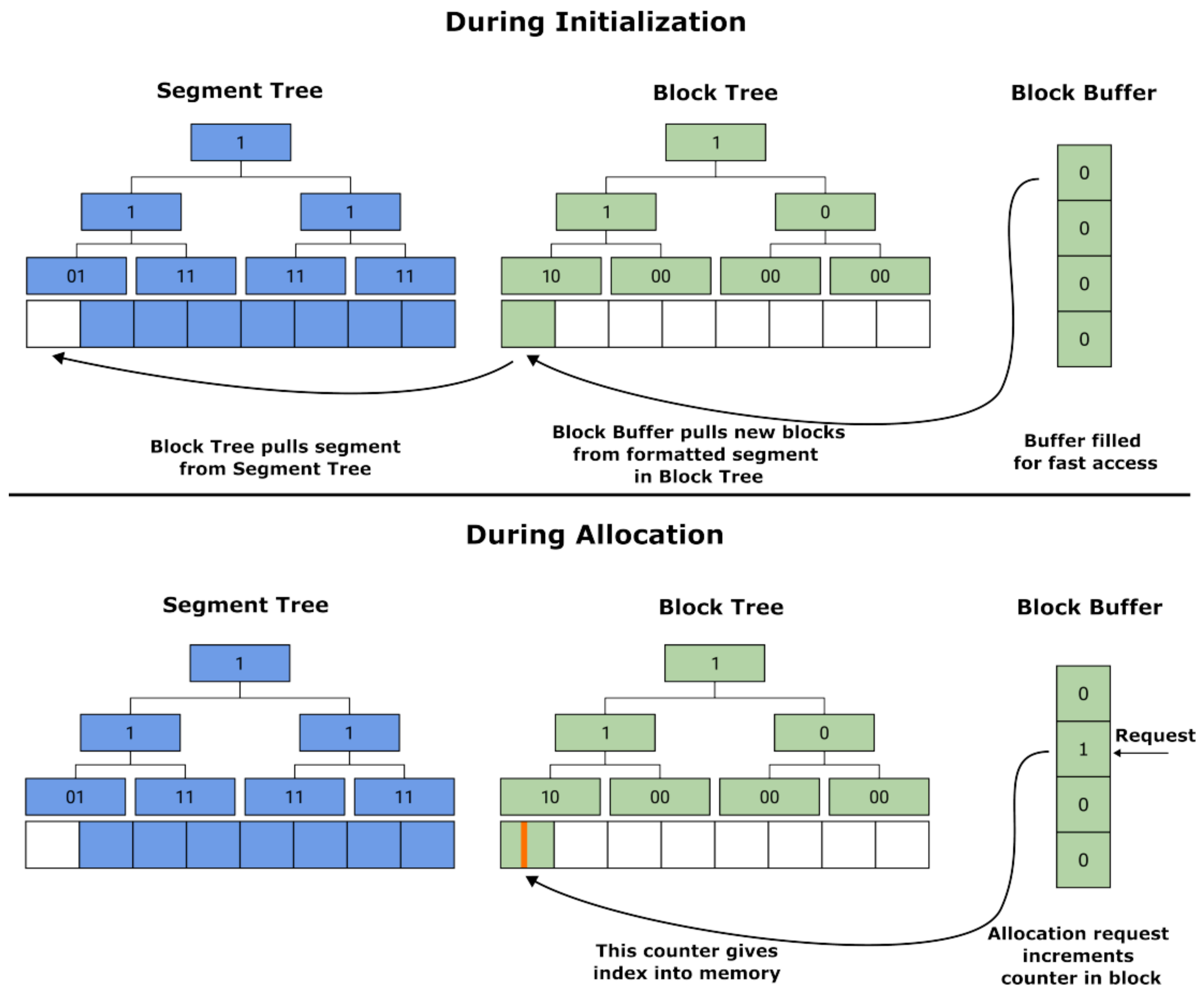
Segment	**Block**	**Slice**
Up to 5 atomics Fully re-usable	Up to 3 atomics Re-usable for a set size	Fastest (32 per atomic) Least re-usable
Largest	Intermediate	Smallest

An ensemble of vEB tree control segments


Segment tree



Gallatin memory manager design


During Initialization
Segment Tree
1
1
1
01
11
11
11
Block Tree
1
1
0
10
00
00
00
Block Buffer
0
0
0
0
Block Tree pulls segment from Segment Tree
Block Buffer pulls new blocks from formatted segment in Block Tree
Buffer filled for fast access
During Allocation
Segment Tree
1
1
1
01
11
11
11
Block Tree
1
1
0
10
00
00
00
Block Buffer
0
1
0
0
Request
This counter gives index into memory
Allocation request increments counter in block

Benckmarks

Expansion of Winter et al. PPOPP ’21 paper

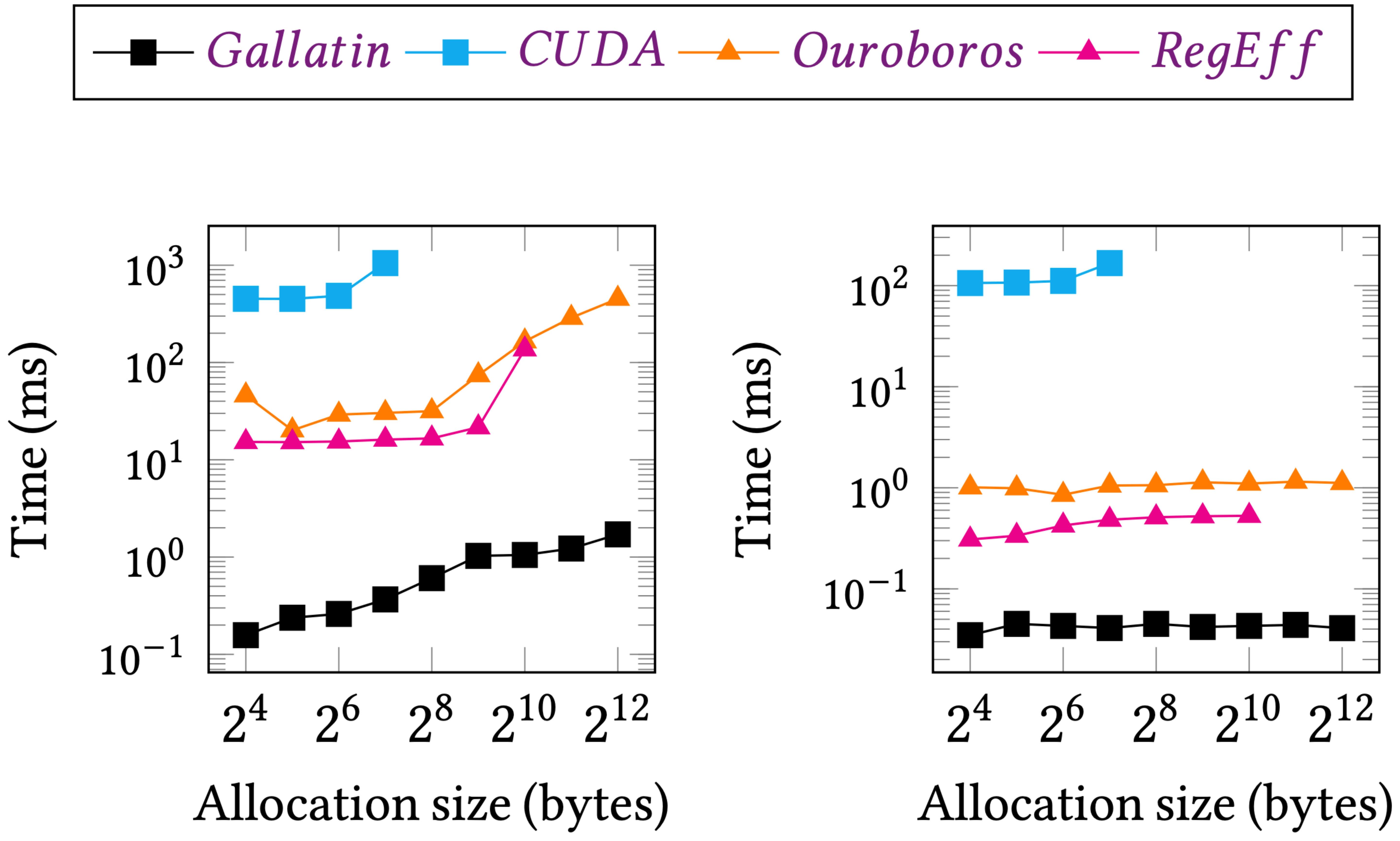
Benchmark tests:

- Single-size allocation/free
- Mixed-size allocation/free
- Weak scaling for single-sized alloc/free
- Memory utilization as a function of allocation size
- Fragmentation as a measure of offsets returned.
- Graph initialization, insertion, and deletion.

Benchmark upgrades

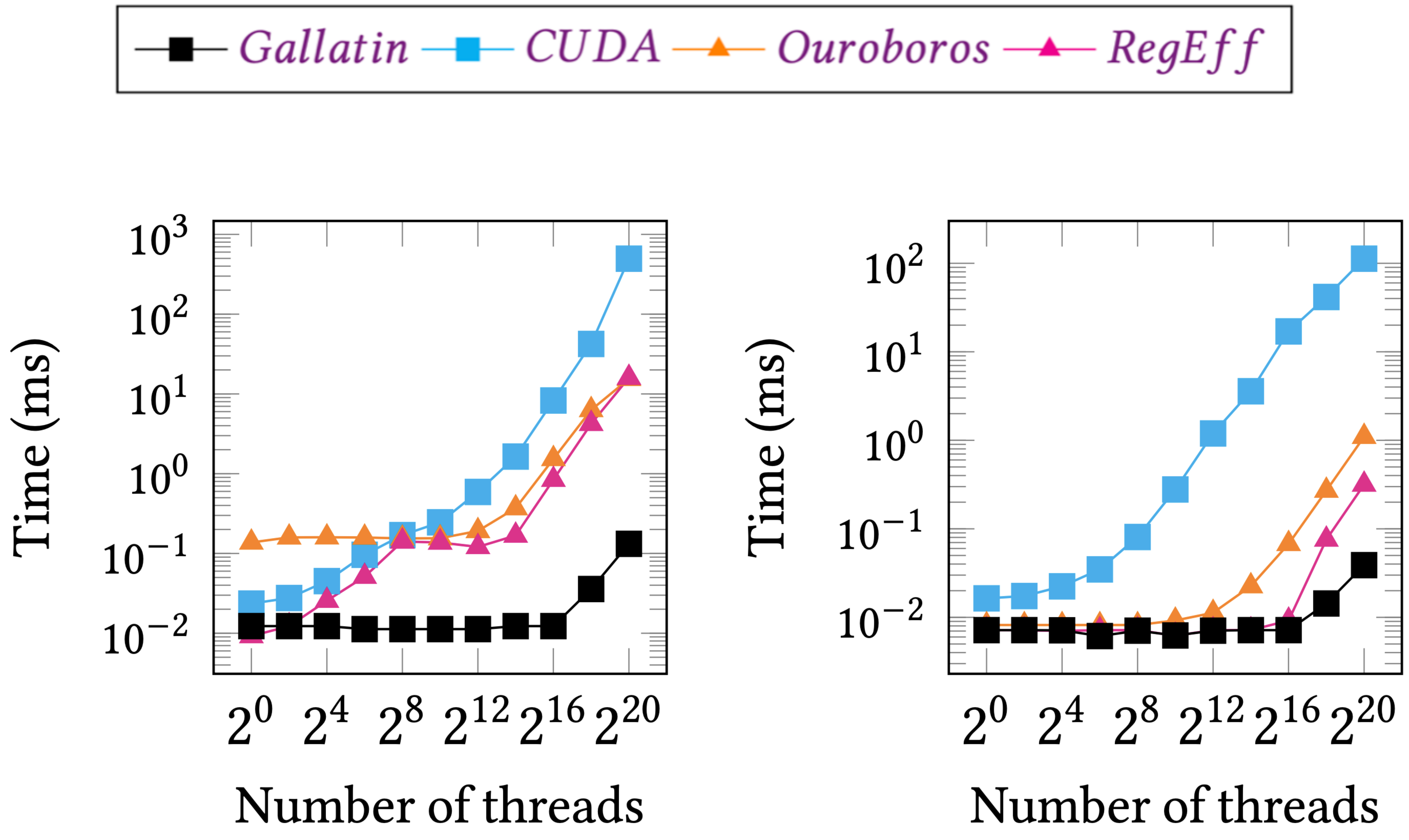
What was updated	Why
#Allocations 100K -> 1 Million	Real use-cases + saturates GPU.
Allocators reset between runs	Performance from a fresh start
Large graph in app. benchmarks	Better represent real graphs

Gallatin is fastest for pure allocation/frees



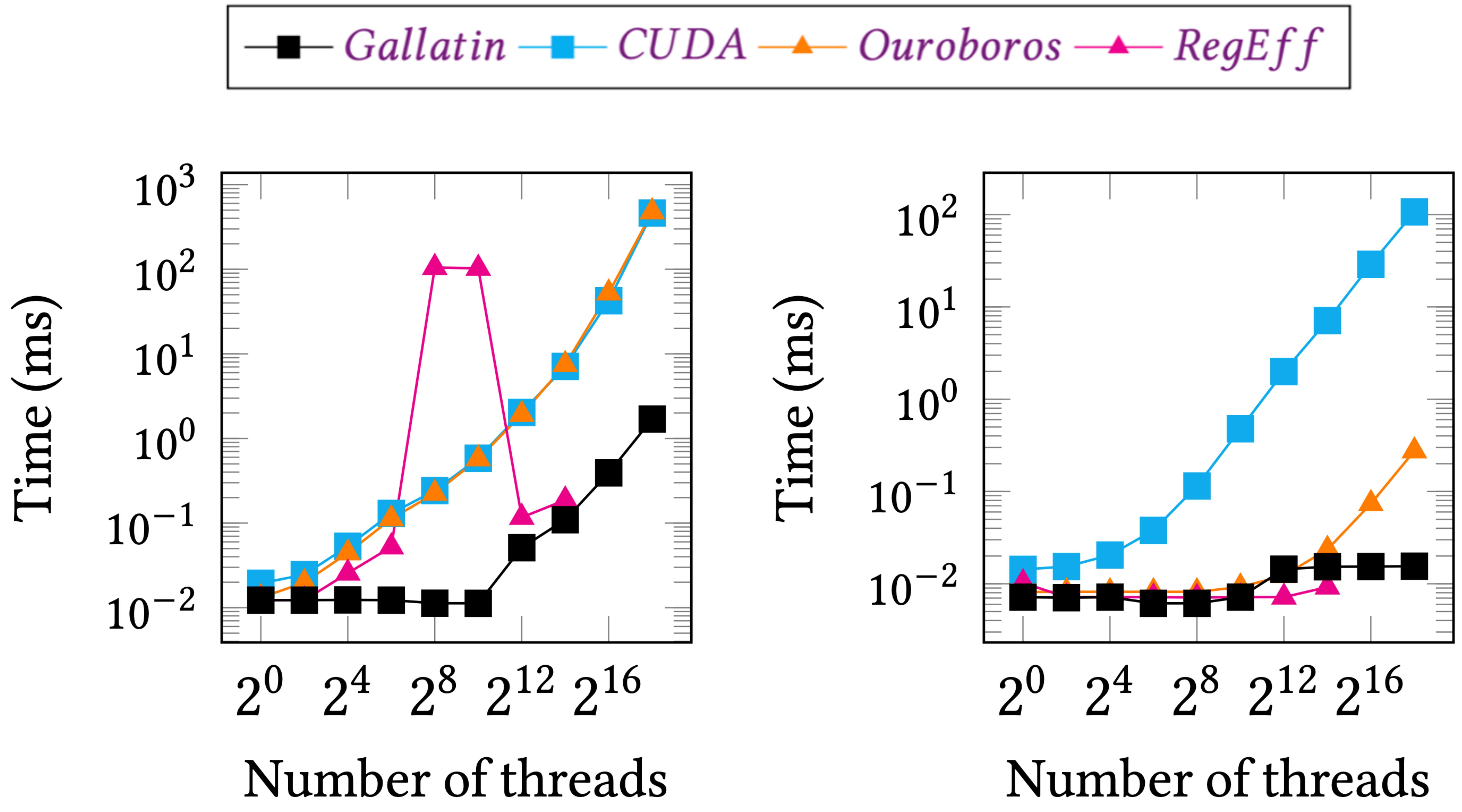
Allocations **Free**

Gallatin is fastest for scaling allocation/frees



Allocations **Free**

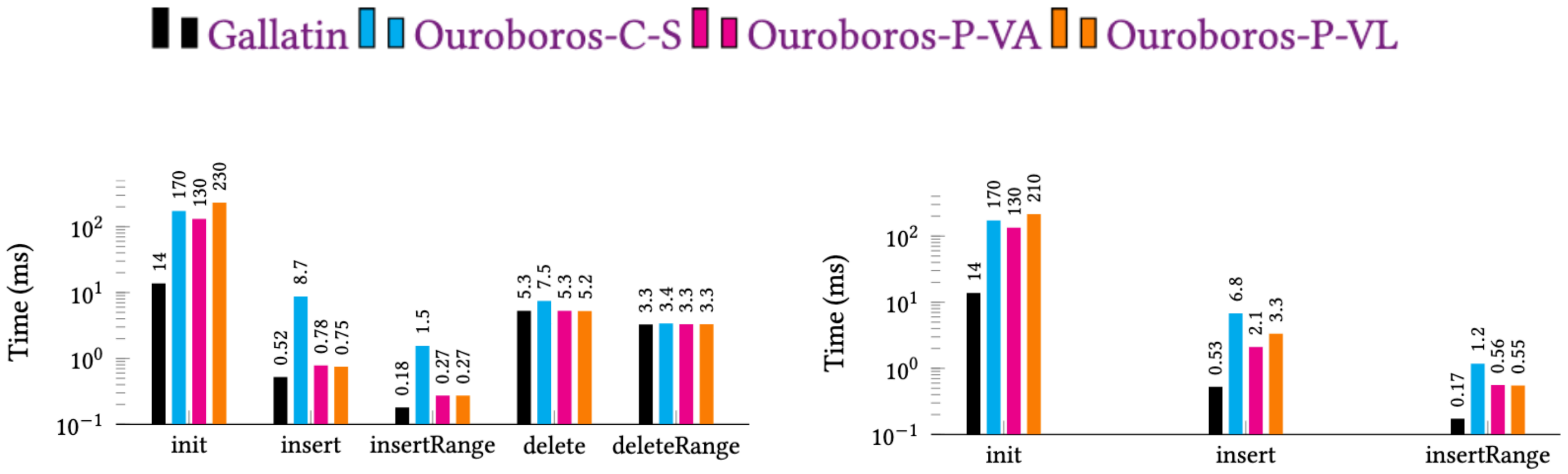
Gallatin is fastest for scaling allocation/frees



Allocations

Free

Gallatin is fastest for graph operations



Gallatin

- Tree-based design supports both fast and arbitrary size allocations
- The van Emde Boas (vEB) tree is amenable to the massive parallelism in GPUs
- A general-purpose allocator can simplify the design of downstream applications and improve performance

https://github.com/saltsystemslab/gallatin/

Dynamic data structures and applications on GPUs

Metagenomic data processing in MHM
ACDA 2023

Jasper: App. Nearest Neighbor Search

GPU Filters
PPoPP 2023

Streaming graphs
PPoPP 2024

GPU Hash tables
ALENEX 2026

Gallatin GPU Memory Manager
PPoPP 2024

https://github.com/saltsystemslab/

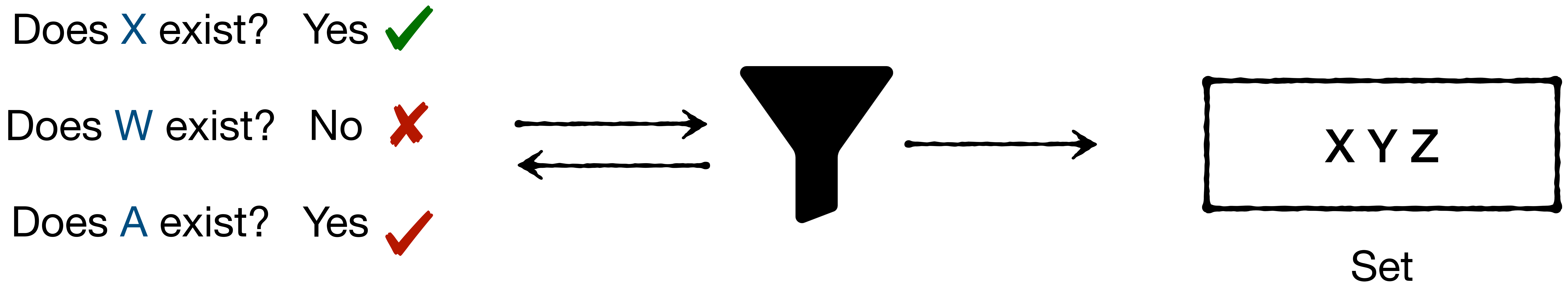
GPU Filters
PPoPP 2023

Hunter McCoy

Steve Hofmeyr

Kathy Yelick

What is a filter data structure?



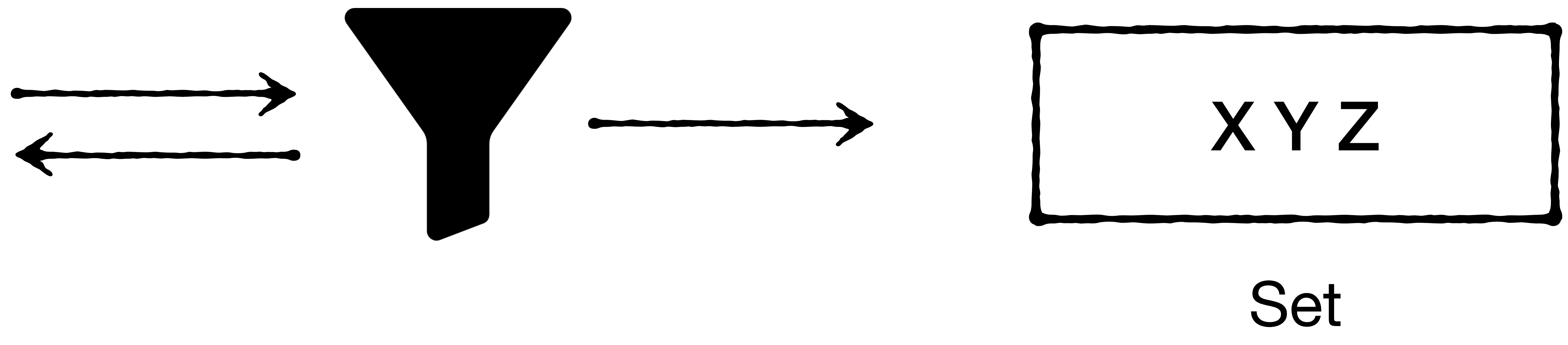
A filter compactly represents a set by trading off accuracy for space efficiency

What is a filter data structure?

Does X exist? Yes ✓

Does W exist? No ✗

Does A exist? Yes ✓



A filter compactly represents a set by trading off accuracy for space efficiency

A filter guarantees a false-positive rate ϵ

$q =$ **query item** $S =$ **set of items**

if q ∈ S, return	**True** with probability 1	true positive
if q ∉ S, return	**False** with probability $> 1 - \epsilon$	true negative
	True with probability $\leq \epsilon$	false positive

False positives with tunable probability

A filter guarantees a false-positive rate ϵ

$q =$ **query item** $S =$ **set of items**

if q ∈ S, return **True** with probability 1 true positive

if q ∉ S, return

$$\begin{cases} \textbf{False} \text{ with probability } > 1-\epsilon & \text{true negative} \\ \textbf{True} \text{ with probability } \leq \epsilon & \text{false positive} \end{cases}$$

One-sided errors

False positives with tunable probability

False-positives enable filters to be compact

$n =$ **number of items** $U =$ **universe of items**

space $\geq n \log(1/\epsilon)$

space $= \Omega(n \log(|U|)$

Filter

Hash table/Tree

False-positives enable filters to be compact

$n =$ **number of items** $U =$ **universe of items**

space $\geq n \log(1/\epsilon)$

Small

space $= \Omega(n \log(|U|))$

Large

Filter

Hash table/Tree

False-positives enable filters to be compact

$n =$ **number of items** $U =$ **universe of items**

space $\geq n \log(1/\epsilon)$

Small

space $= \Omega(n \log(|U|)$

Large

Filter

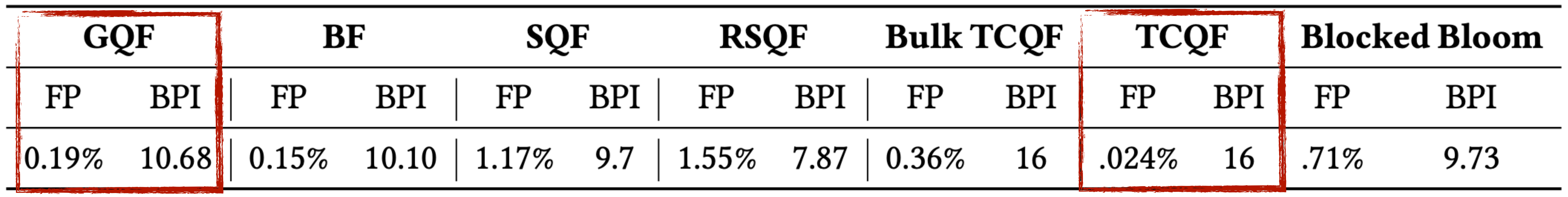
Hash table/Tree

For $\epsilon = 2\,\%$, filters require ~1 Byte/item. Hash table/Tree can take >8-16 Byte/item.

Our results:

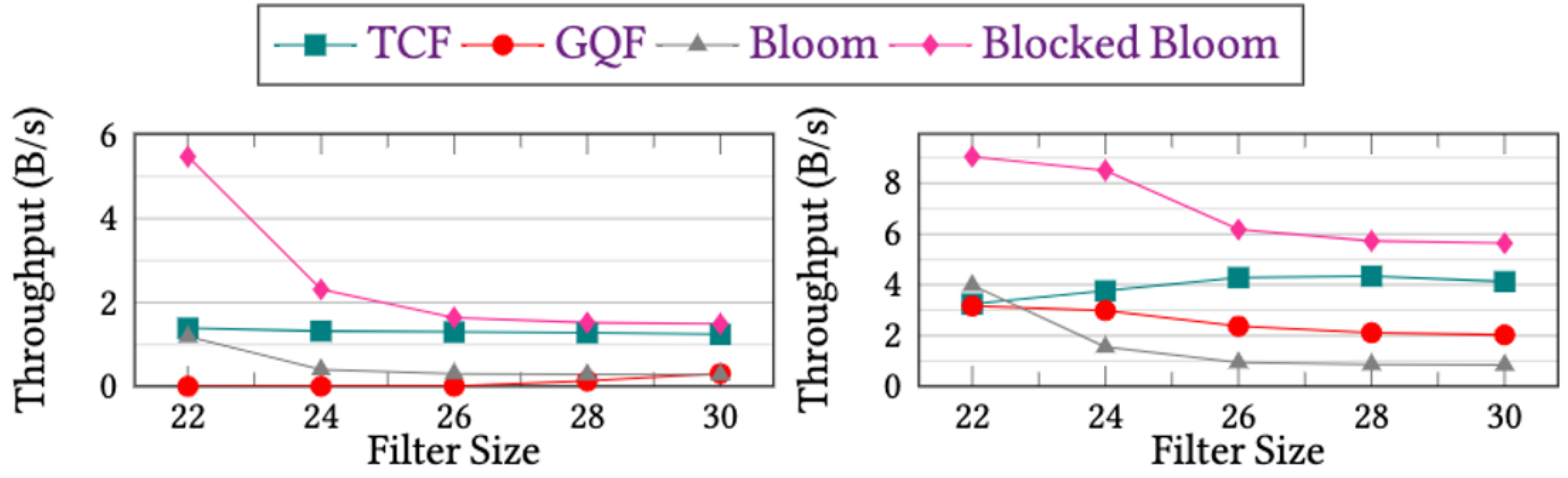
- **Present new GPU filter designs:**
  - Two-Choice Filter (TCF)
    - Stable filter with key-value association/deletion
  - GPU Quotient Filter (GQF)
    - Filter with key-value association/deletion/dynamic counters
- Cooperative groups (CG)-based execution to maximize compute/ memory throughput
- Up to **4.4x faster** than previous GPU filters
- Thread-level point API and host-managed bulk API for easy integration

High Performance Filters for GPUs [PPoPP '23]



GQF		BF		SQF		RSQF		Bulk TCQF		TCQF		Blocked Bloom	
FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI	FP	BPI
0.19%	10.68	0.15%	10.10	1.17%	9.7	1.55%	7.87	0.36%	16	.024%	16	.71%	9.73

Table 2. False-positive rate (FP) and bits per item (BPI) of various filters for experiments in Figure 4 and Figure 3.



(a) Point Inserts. **(b)** Point Queries.

Metagenomic data processing in MHM
ACDA 2023

Hunter McCoy

Steve Hofmeyr

Kathy Yelick

Metagenomic assembly over massive datasets

- **MetaHipMer[1]**
  - A GPU-accelerated metagenomic assembler that runs on exascale systems
- MetaHipMer recently completed the largest co-assembly ever
  - 9,400 nodes on Frontier
  - 37,000 GPUs
  - 71.6 Terabytes assembly over Tara Oceans dataset

1. Hofmeyr, S., Egan, R., Georganas, E. *et al.* Terabase-scale metagenome coassembly with *MetaHipMer*. *Sci Rep* **10**, 10689 (2020). https://doi.org/10.1038/s41598-020-67416-5

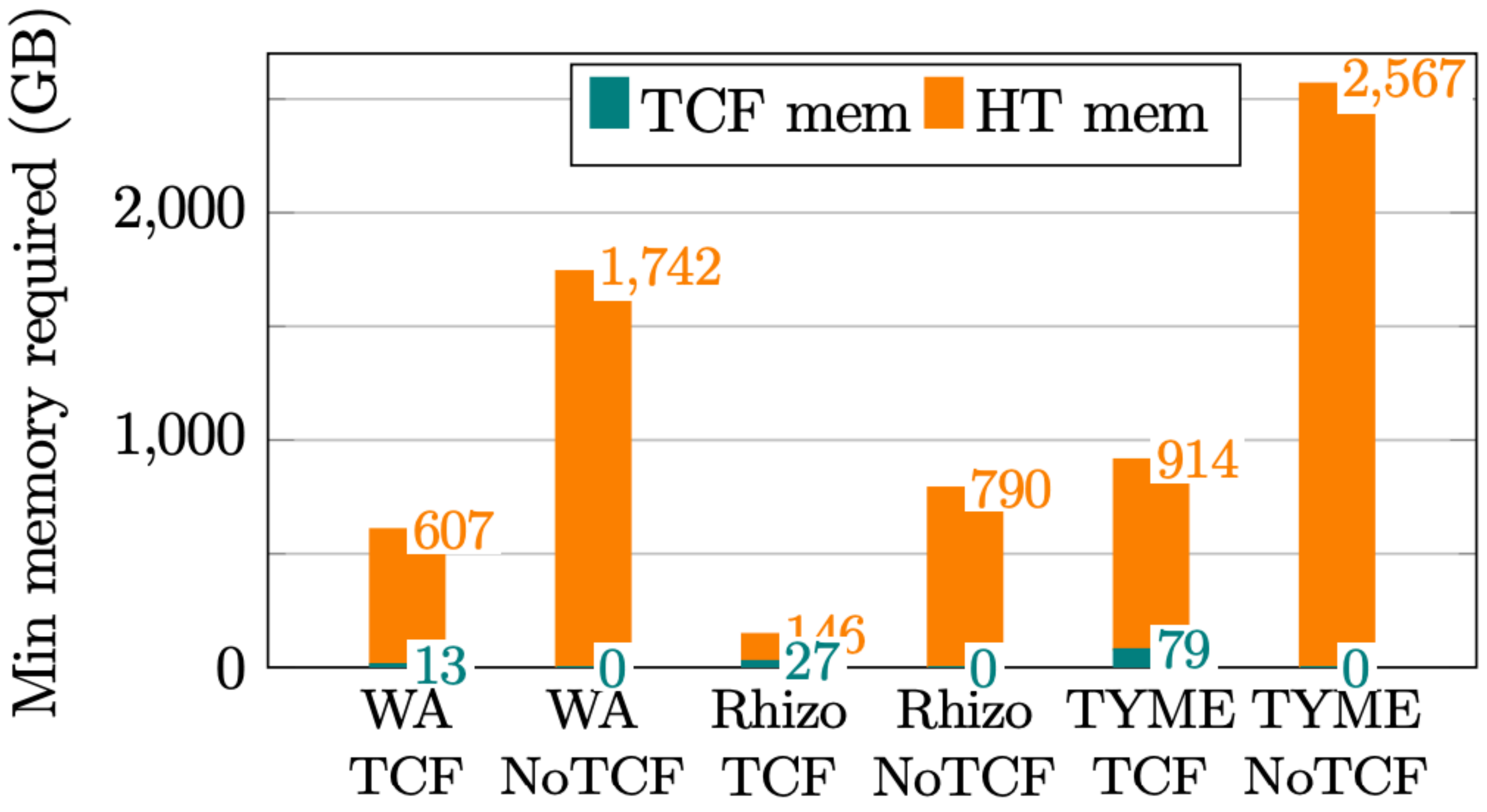
GPUs are fast, but have limited memory

Peak RAM usage is the biggest bottleneck in scaling MHM

Singleton Sieving in MetaHipMer [ACDA ‘23]

- **Using GPU-based filters:**
  - 5.4X reduction in peak RAM usage of most memory constrained phase
  - 43% overall memory reduction
- No loss in assembly quality or runtime
- Reduced load variance across nodes 30% to 13%

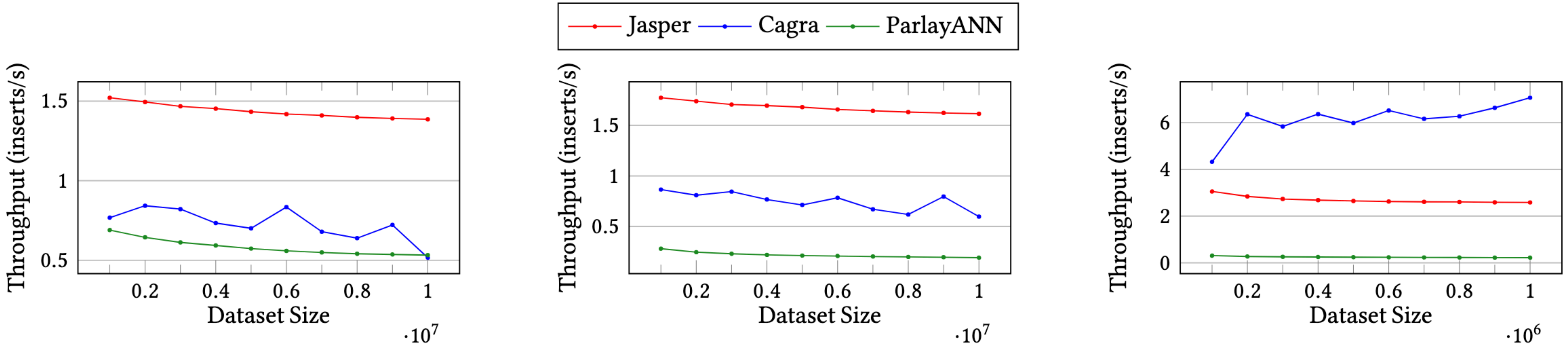
Singleton Sieving in MetaHipMer [ACDA '23]



Jasper: GPU-based Approximate Nearest Neighbor Search

Zikun Wang

Hunter McCoy



(a) BigANN construction 1M-10M **(b) Deep construction 1M-10M** **(c) Gist construction 100k-1M**

Figure 6: Construction performance on A100.

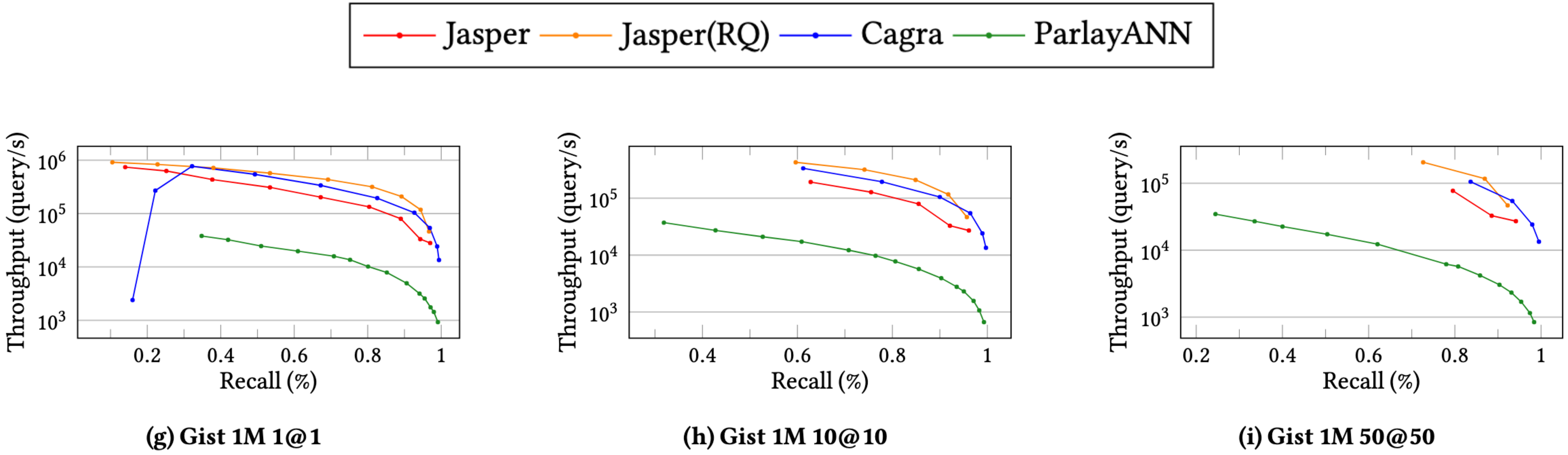


Figure 4: BigANN, Deep, and Gist query recall/throughput curves on A100. X-axis is recall, y-axis is throughput in queries per second. Recall is presented for 1@1, 10@10, and 50@50.

Next steps

- A general-purpose library of high-performance GPU data structures and algorithms
- Providing additional tools and primitives for GPUs
- Scalable (host-device) memory management
- We highly welcome *collaborations* and *contributions*

https://github.com/saltsystemslab/