DTU

02/11/23

Real-Time Visual and Machine Learning Systems

Agenda – Module 1 part A

- Memory Hierarchies in Hardware
- Memory Hierarchies in Software
- Memory Allocations and Data Structures
- Smart Pointers
- Graph Structures
- Garbage Collectors
- Computational Graphs
- Exercises

Memory Hierarchies in Hardware

- The speed of RAM and disks vs. the speed of processors
- Not the whole picture – core count, cache sizes, efficiency, sophistication of branch prediction, bus bandwidth

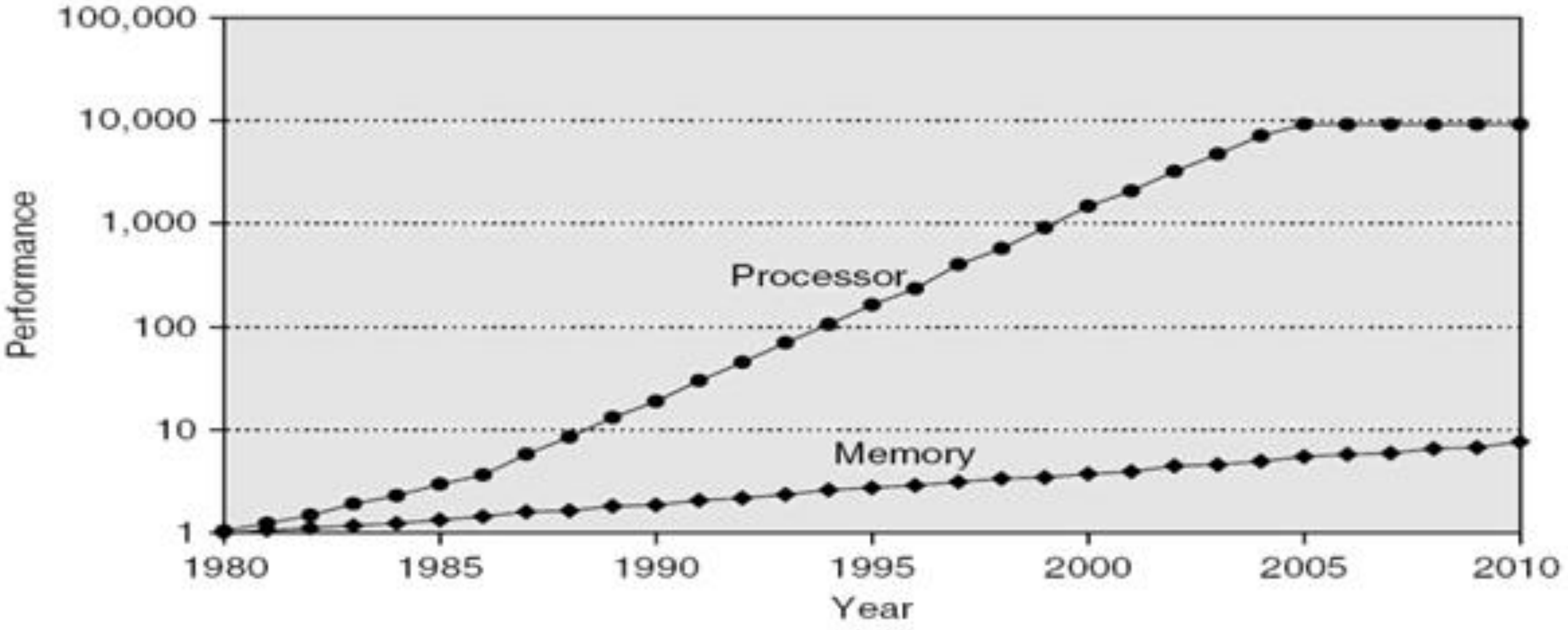
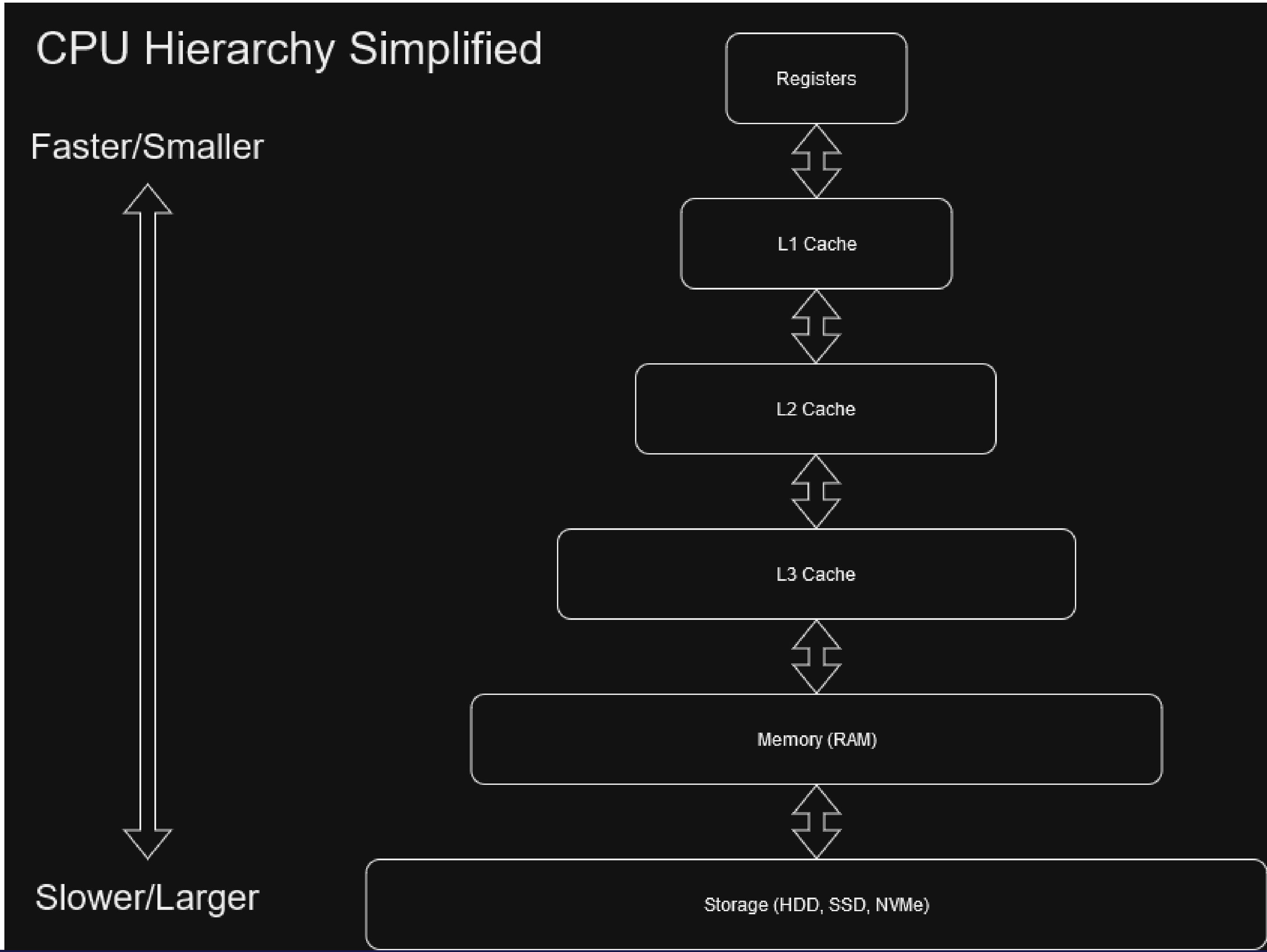


Figure 25.2

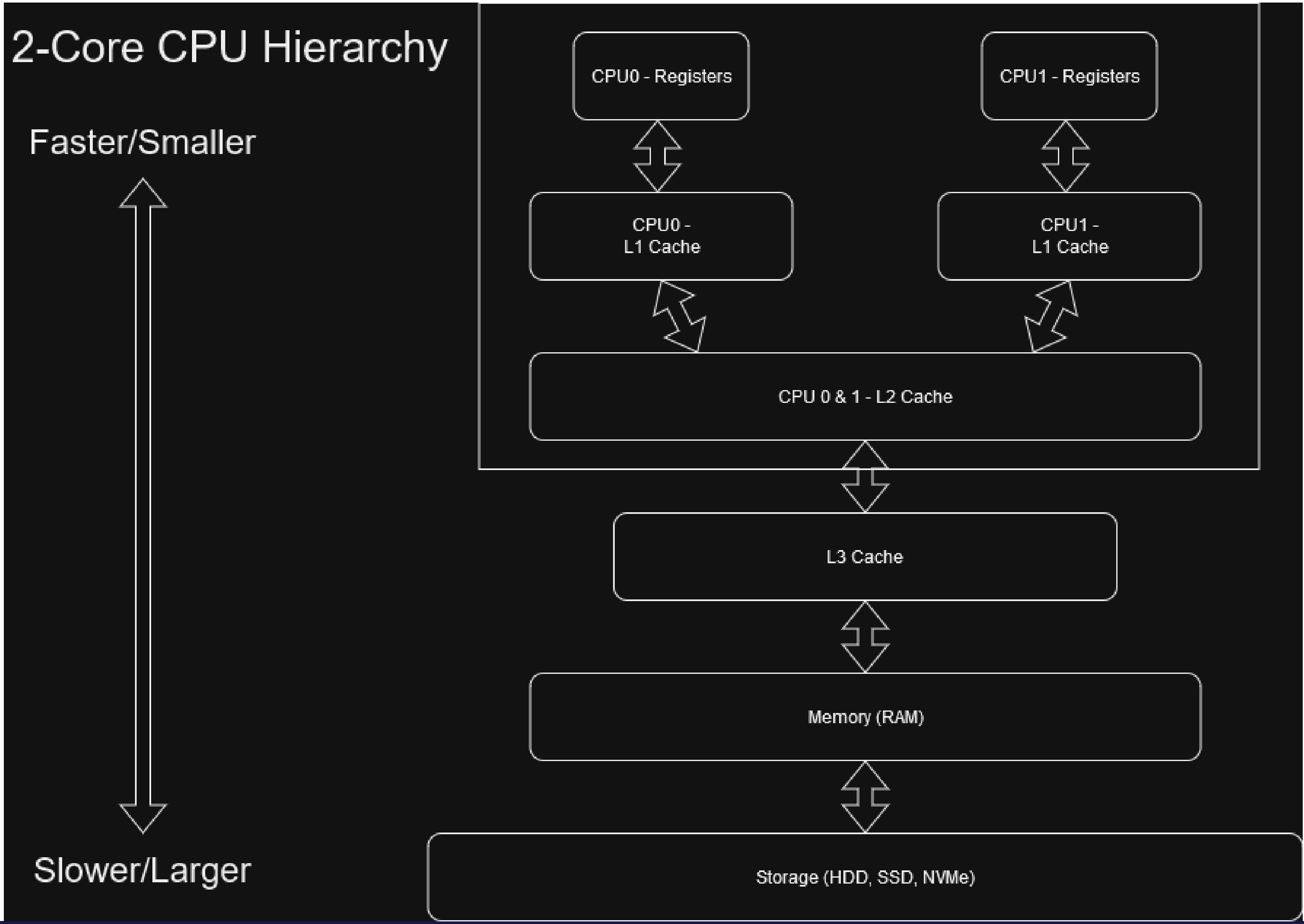
Memory Hierarchies in Hardware

From slowest to fastest



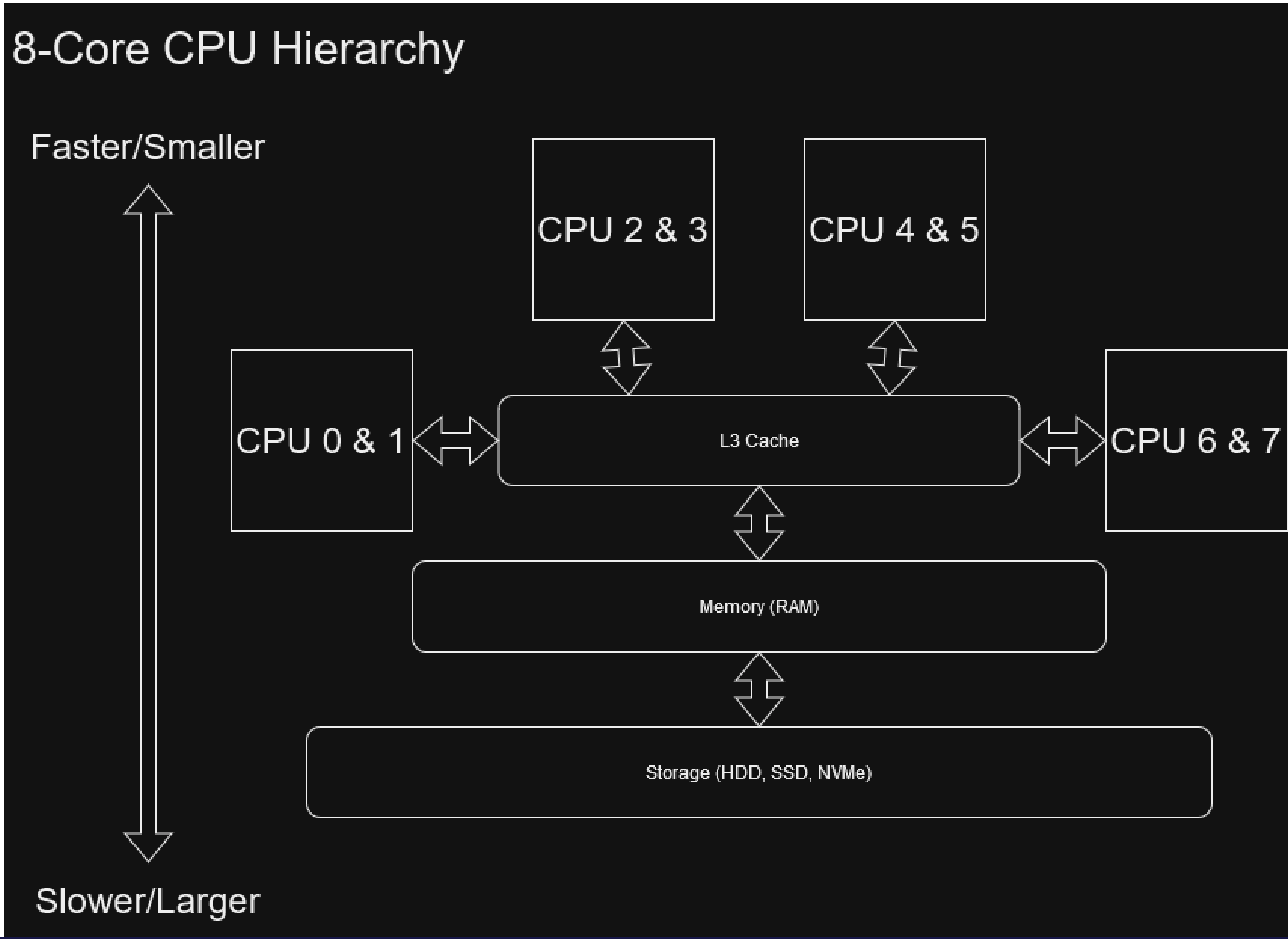
Memory Hierarchies in Hardware

Different levels can be shared differently



Memory Hierarchies in Hardware

Different levels can be shared differently



Memory Hierarchies in Hardware

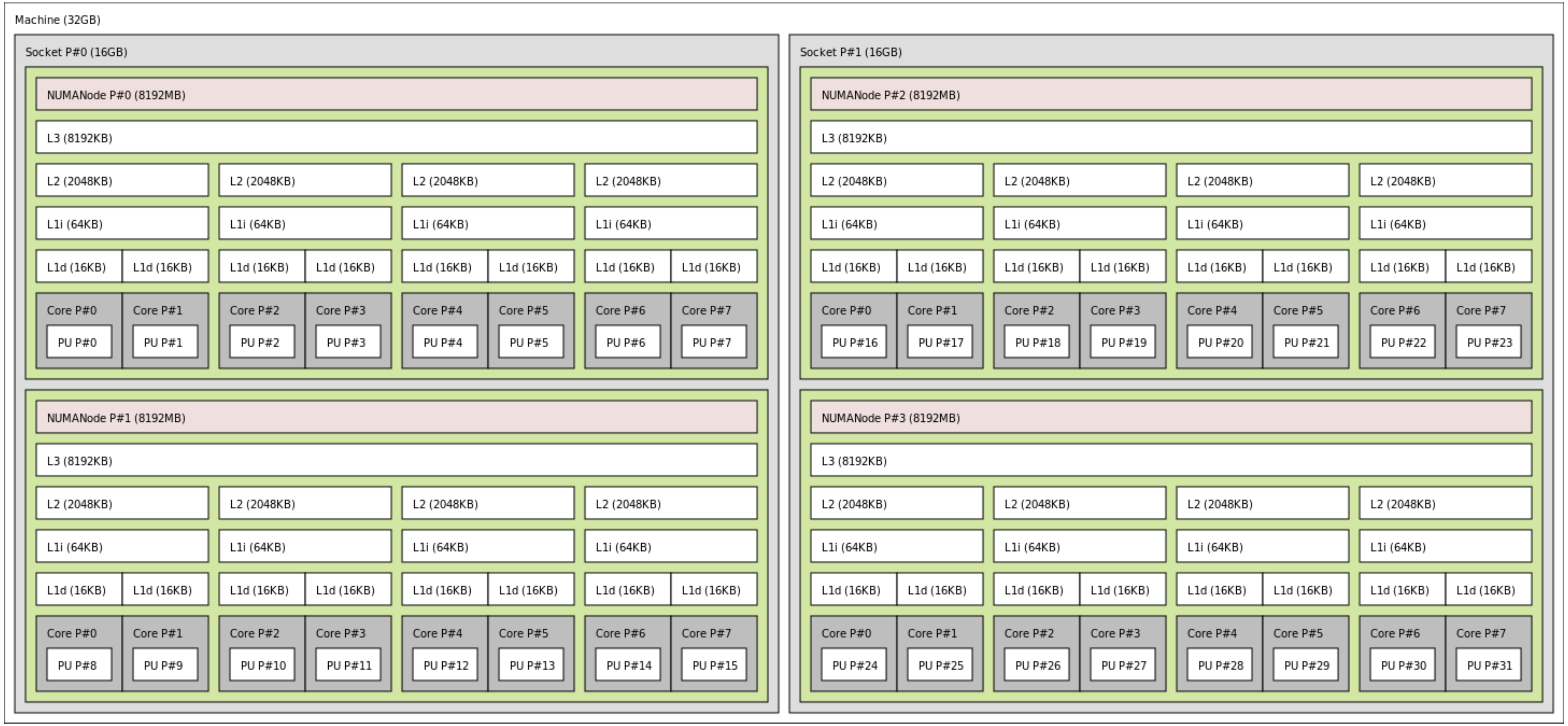


Image Link

Memory Hierarchies in Hardware – GPU (H100)

Image Link

Memory Hierarchies in Hardware
GPU (H100)

Image Link

Memory Hierarchies in Software
Memory Allocations – in C!

```c
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;
```

Memory Hierarchies in Software
Memory Allocations – Spot the Error!

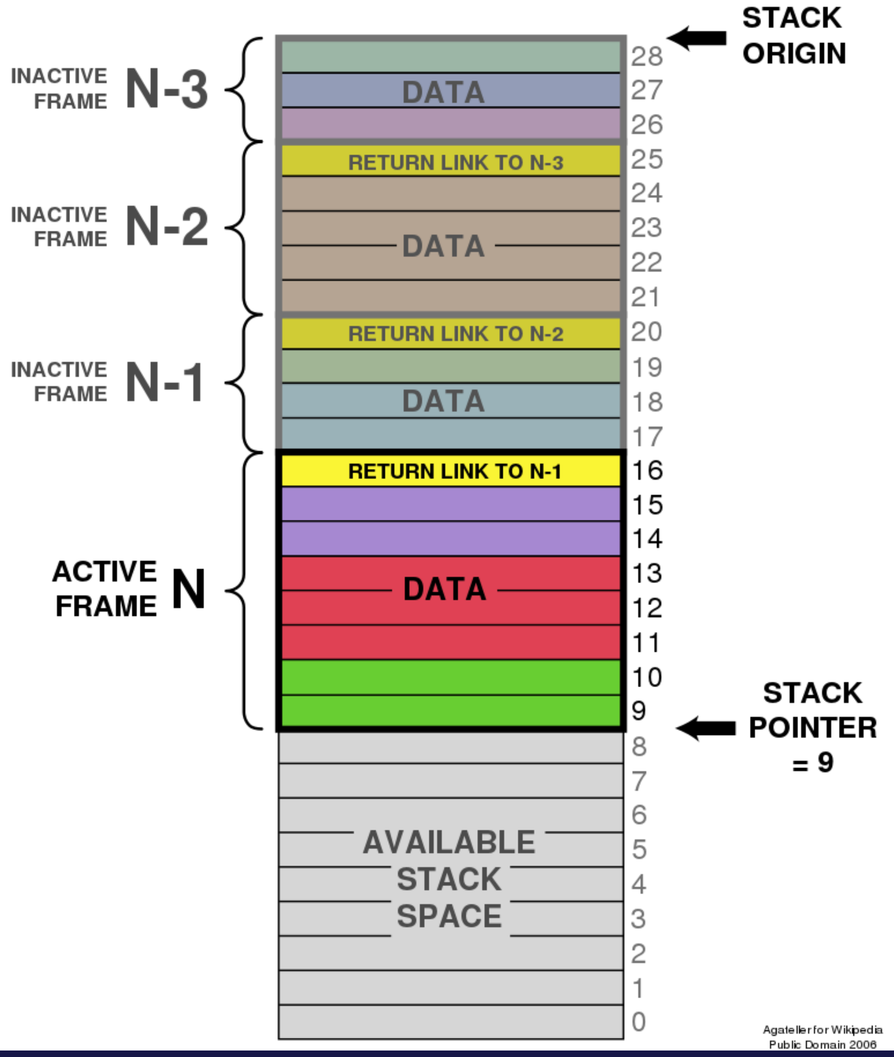
```
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;

free(integer_array);
```

Memory Hierarchies in Software
Memory Allocations – Spot the Error!

```
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;

free(base_integer_array);
```

Memory Hierarchies in Software
Where does the memory come from?

- Function calls
- Variables with a size known at compile time



[Image Link]

Memory Hierarchies in Software
Where does the memory come from?

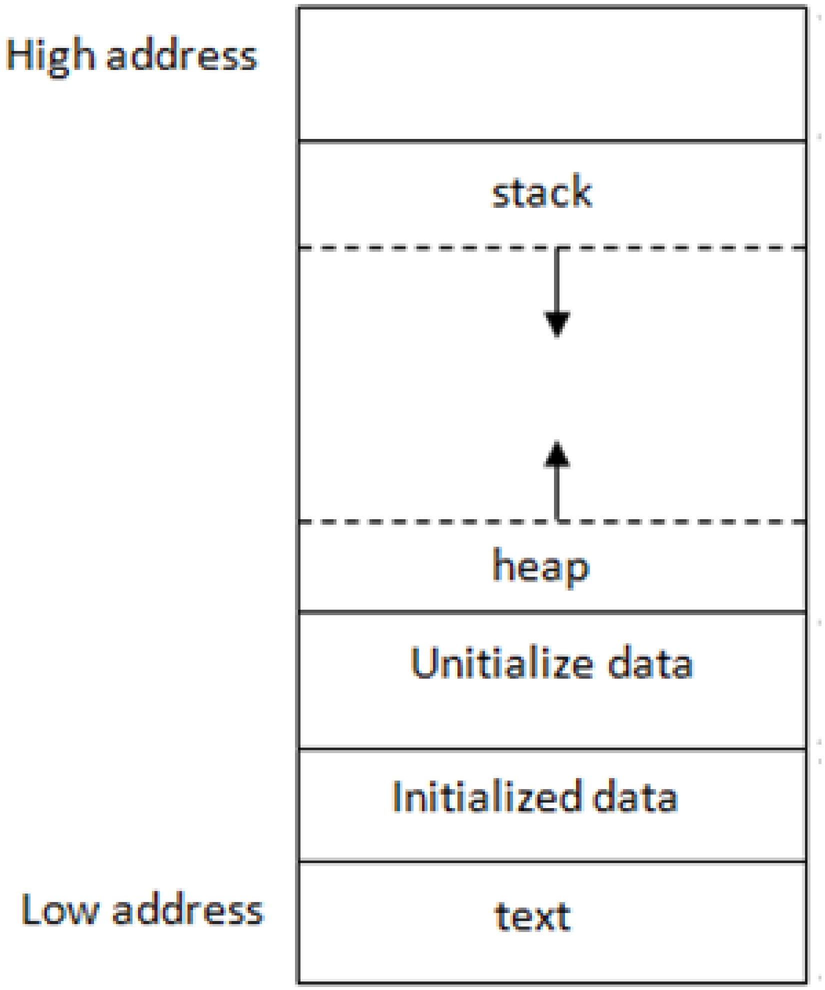
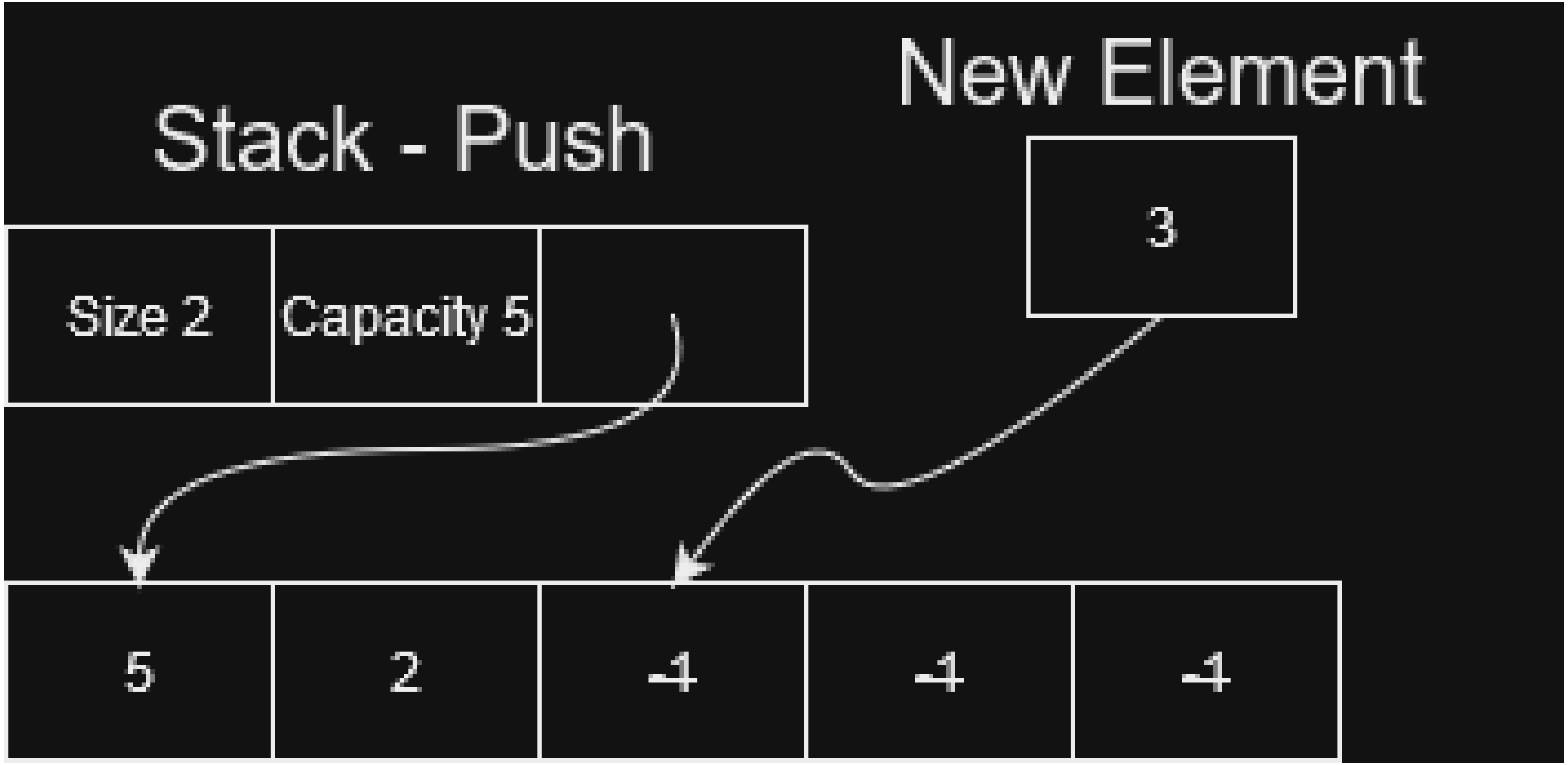


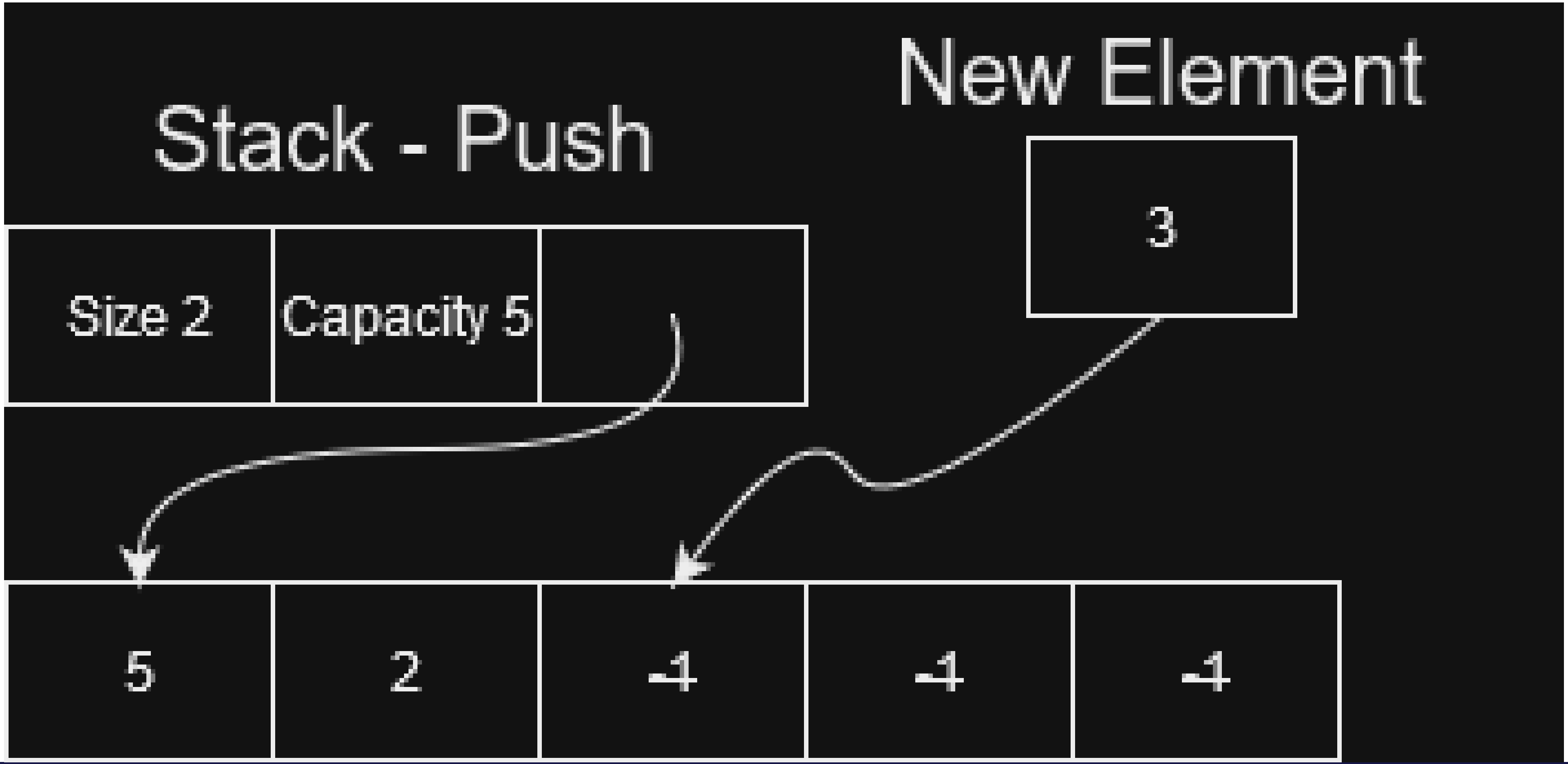
Image Link

Memory Allocations and Data Structures
Dynamic Arrays



Memory Allocations and Data Structures
Dynamic Arrays

Why resizing can be problematic for memory safety (pointers and references).



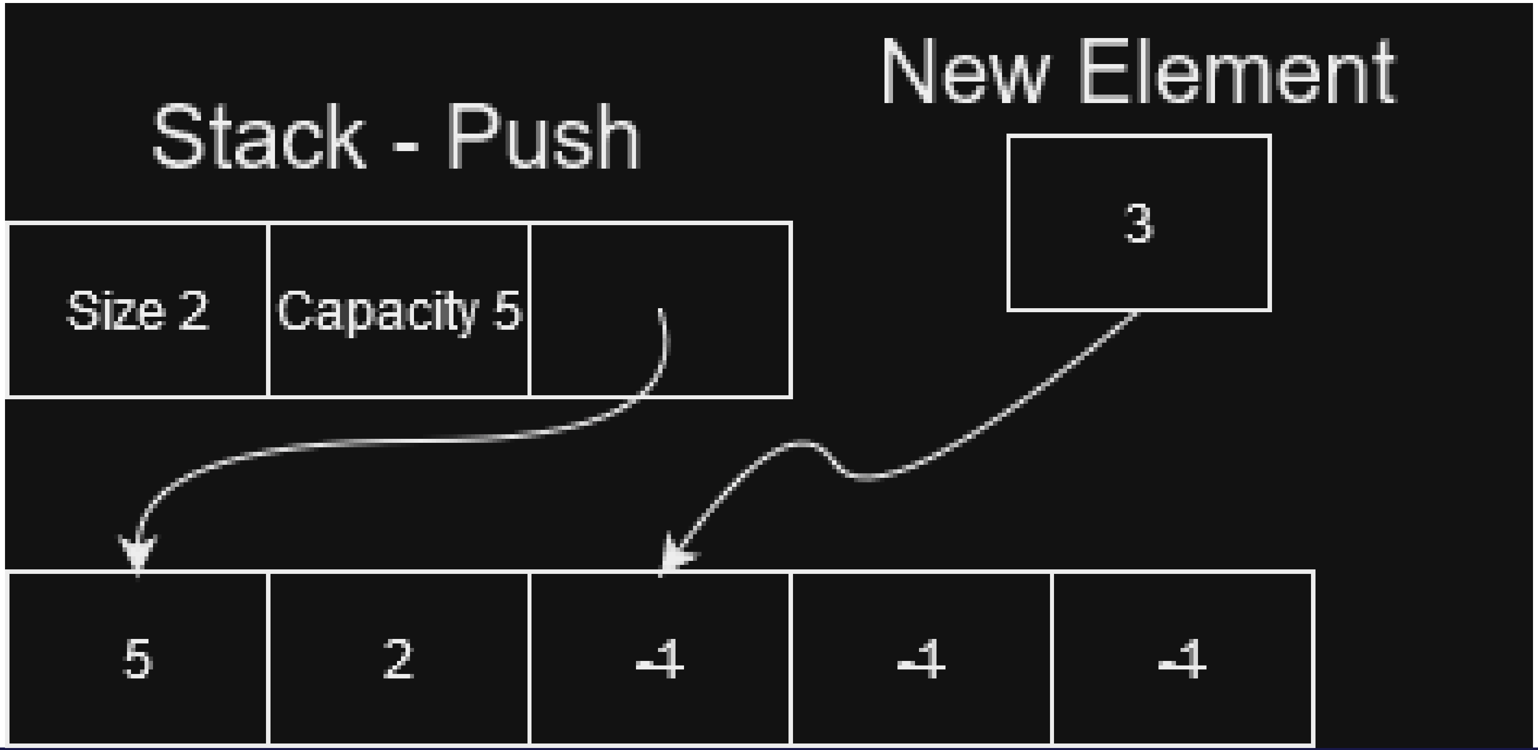
Memory Hierarchies in Software
Reference, Copy, Clone, Move

Adding references is the default in Python

Clone is the default in C++

Move is the default in Rust

Copy is mostly for small types



Memory Allocations and Data Structures Accesses and Strides

Cache Lines

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a.first	a.first	a.second	a.second	a.second	a.second	b.first	b.first	b.second	b.second	b.second	b.second	c.first	c.first	c.second	c.second

Memory Allocations and Data Structures
Accesses and Strides

Pad for alignment

Remove pad for storage

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a.first	a.first	a.pad	a.pad	a.second	a.second	a.second	a.second	b.first	b.first	b.pad	b.pad	b.second	b.second	c.second	c.second

Memory Allocations and Data Structures Accesses and Strides

Strides

```
RUNNING ACCESS TESTS WITH 100 data elements for 100000 iterations!
==============================================================
Sequential access: 1 ms
Non-wrapping strided access (2): 1 ms
Non-wrapping strided access (3): 0 ms
Non-wrapping strided access (4): 0 ms
Strided access (1): 1 ms
Strided access (5): 15 ms
Strided access (17): 14 ms
Random access: 77 ms

RUNNING ACCESS TESTS WITH 1000 data elements for 100000 iterations!
==============================================================
Sequential access: 8 ms
Non-wrapping strided access (2): 13 ms
Non-wrapping strided access (3): 11 ms
Non-wrapping strided access (4): 6 ms
Strided access (1): 9 ms
Strided access (5): 134 ms
Strided access (17): 139 ms
Random access: 472 ms

RUNNING ACCESS TESTS WITH 10000 data elements for 100000 iterations!
==============================================================
Sequential access: 89 ms
Non-wrapping strided access (2): 125 ms
Non-wrapping strided access (3): 86 ms
Non-wrapping strided access (4): 63 ms
Strided access (1): 94 ms
Strided access (5): 1311 ms
Strided access (17): 1328 ms
Random access: 19206 ms

RUNNING ACCESS TESTS WITH 100000 data elements for 100000 iterations!
==============================================================
Sequential access: 960 ms
Non-wrapping strided access (2): 1510 ms
Non-wrapping strided access (3): 1081 ms
Non-wrapping strided access (4): 787 ms
Strided access (1): 1003 ms
Strided access (5): 14595 ms
Strided access (17): 15034 ms
Random access: 86041 ms
```

Memory Allocations and Data Structures
ND Arrays

We can do this in 2D with row major ordering

```
let data: [[[i32; 2]; 2]; 2] =
                            [
                                [[1, 2], [3, 4]],
                                [[5, 6], [7, 8]]
                            ];

let x_dimension: usize = 2;
let y_dimension: usize = 2;
let z_dimension: usize = 2;

for x_index in 0..x_dimension {
    for y_index in 0..y_dimension {
        for z_index in 0..z_dimension {
            println("{}", data[x_index][y_index][z_index]);
        }
    }
}
```

Memory Allocations and Data Structures
ND Arrays

We can do this in
2D with column major
ordering

```
let data: [[[i32; 2]; 2]; 2] =
                                 [
                                     [[1, 2], [3, 4]],
                                     [[5, 6], [7, 8]]
                                 ];

let x_dimension: usize = 2;
let y_dimension: usize = 2;
let z_dimension: usize = 2;

for z_index in 0..z_dimension {
    for y_index in 0..y_dimension {
        for x_index in 0..x_dimension {
            println!("{}", data[x_index][y_index][z_index]);
        }
    }
}
```

Memory Allocations and Data Structures ND Arrays – Linearized Indexing

Or in 1D and gain nice properties

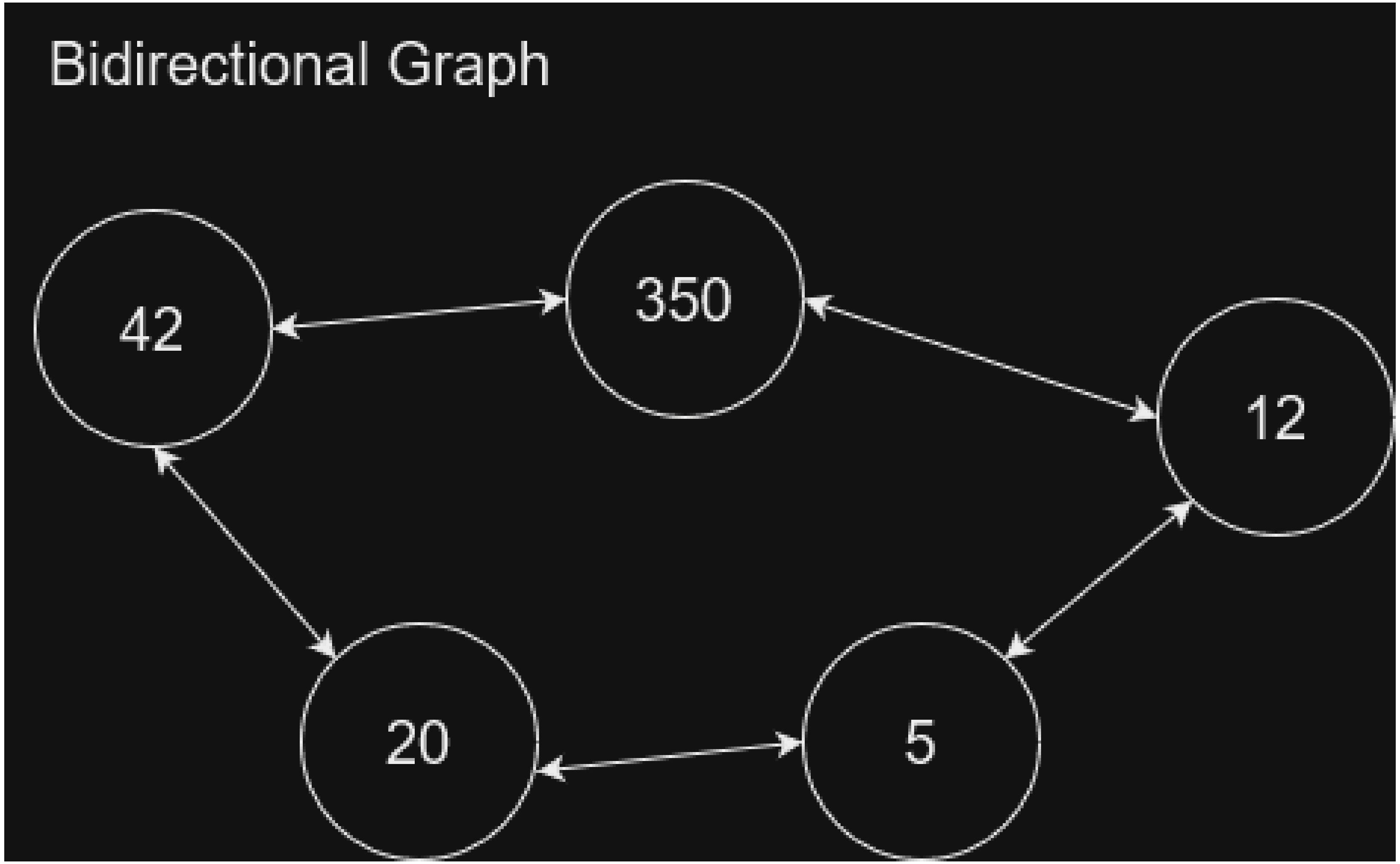
```
let mut data: Vec<i32> = Vec::<i32>::new();
data.push(vec![0, 1, 2, 3]);

let column_count: usize = 2;
let row_count: usize = 2;

for x_index in 0..row_count {
    for y_index in 0..column_count {
        println!("{}", data[x_index * column_count + y_index]);
    }
}
```

Memory Allocations and Data Structures
ND Arrays – Linearized Indexing

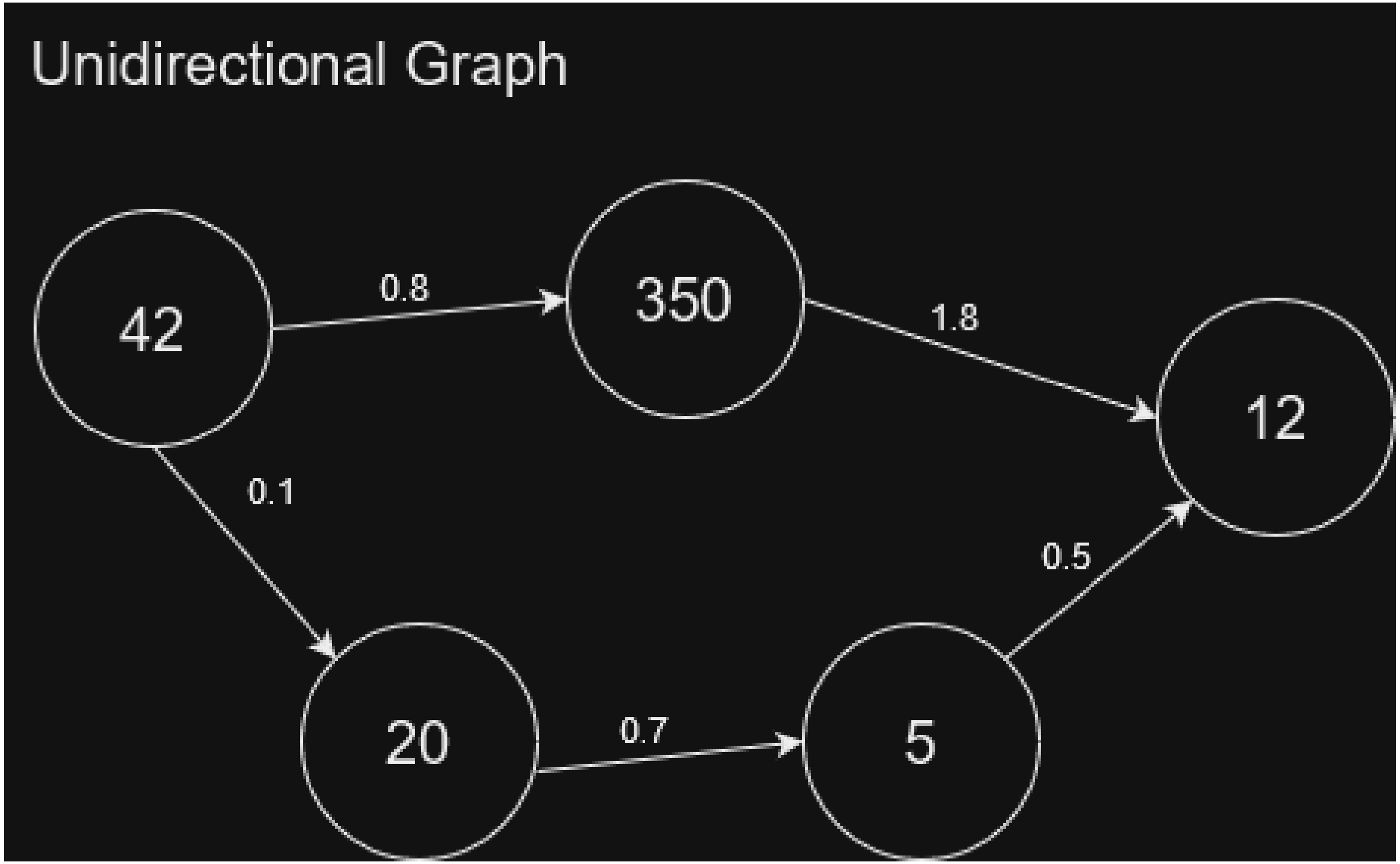
x_index * y_size * z_size + y_index * z_size + z_index

With linearized indexing we gain nice properties like different views on the same data, permutations, easier data transfers and better performance.

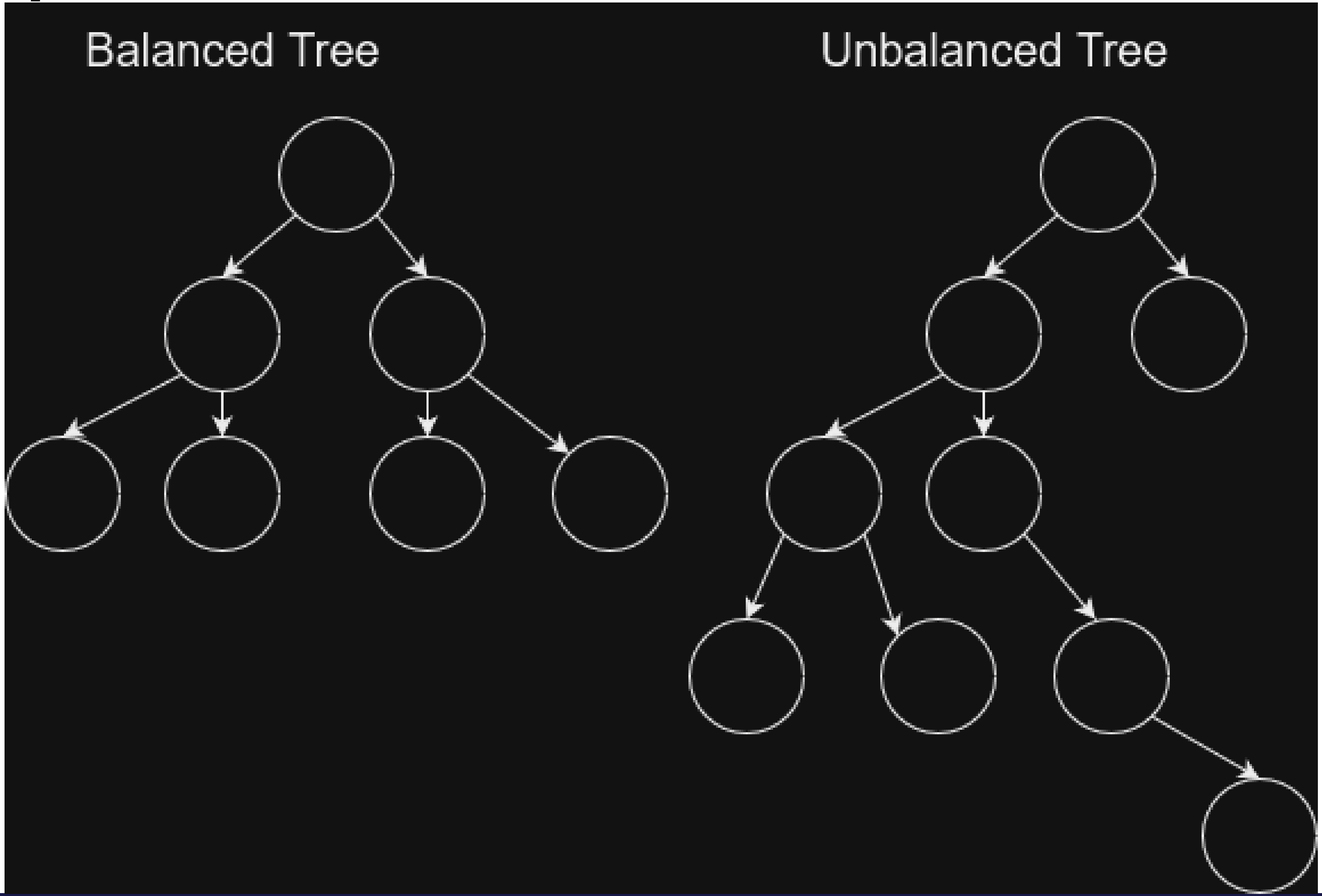
Graph Structures - Bidirectional



Graph Structures – Unidirectional with weighted edges



Graph Structures - Trees

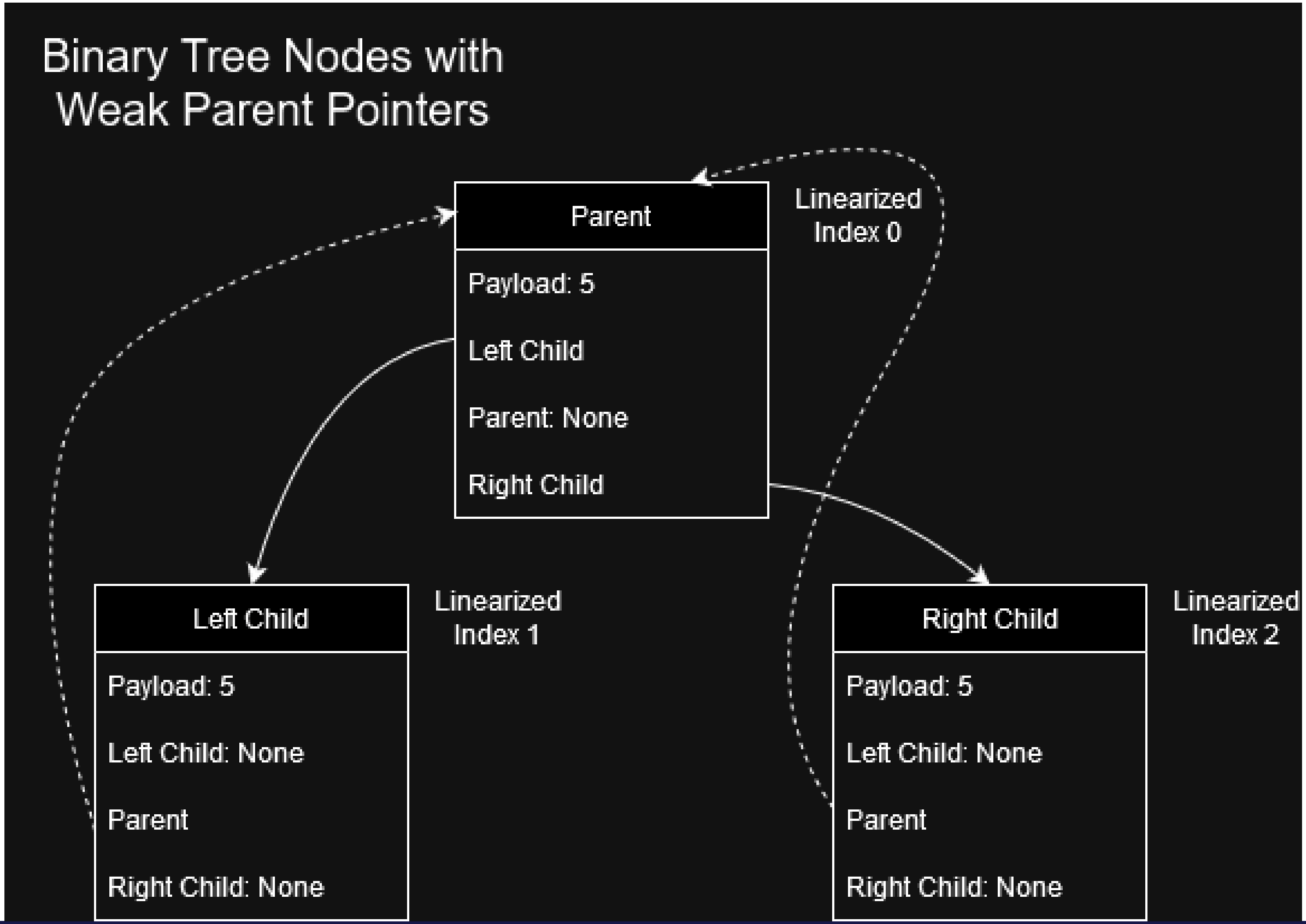


Smart Pointers

Rust and C++11+

- Box<T> or unique_ptr<T>
- Rc<T>, Arc<T> and shared_ptr<T>
- Weak<T> and weak_ptr<T>

Graph Structures



Graph Structures with Indices

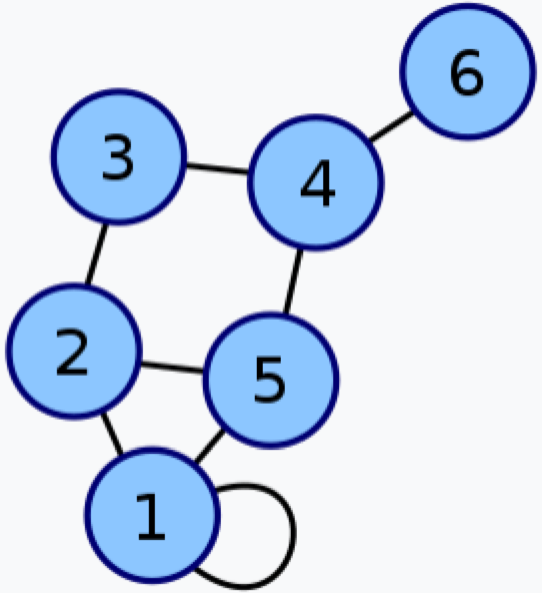
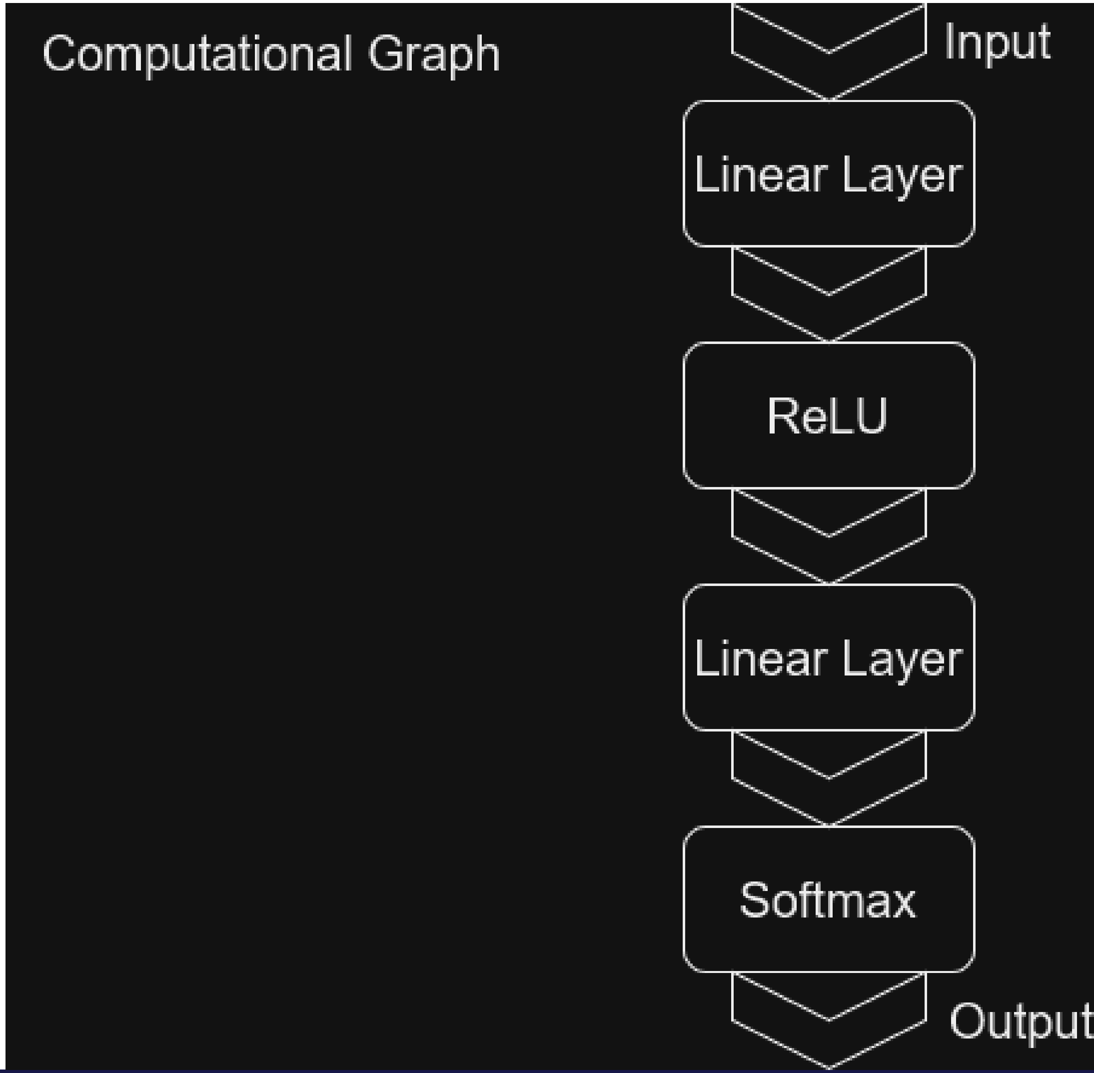
Labeled graph	Adjacency matrix
	$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$ Coordinates are 1–6.

Image Link

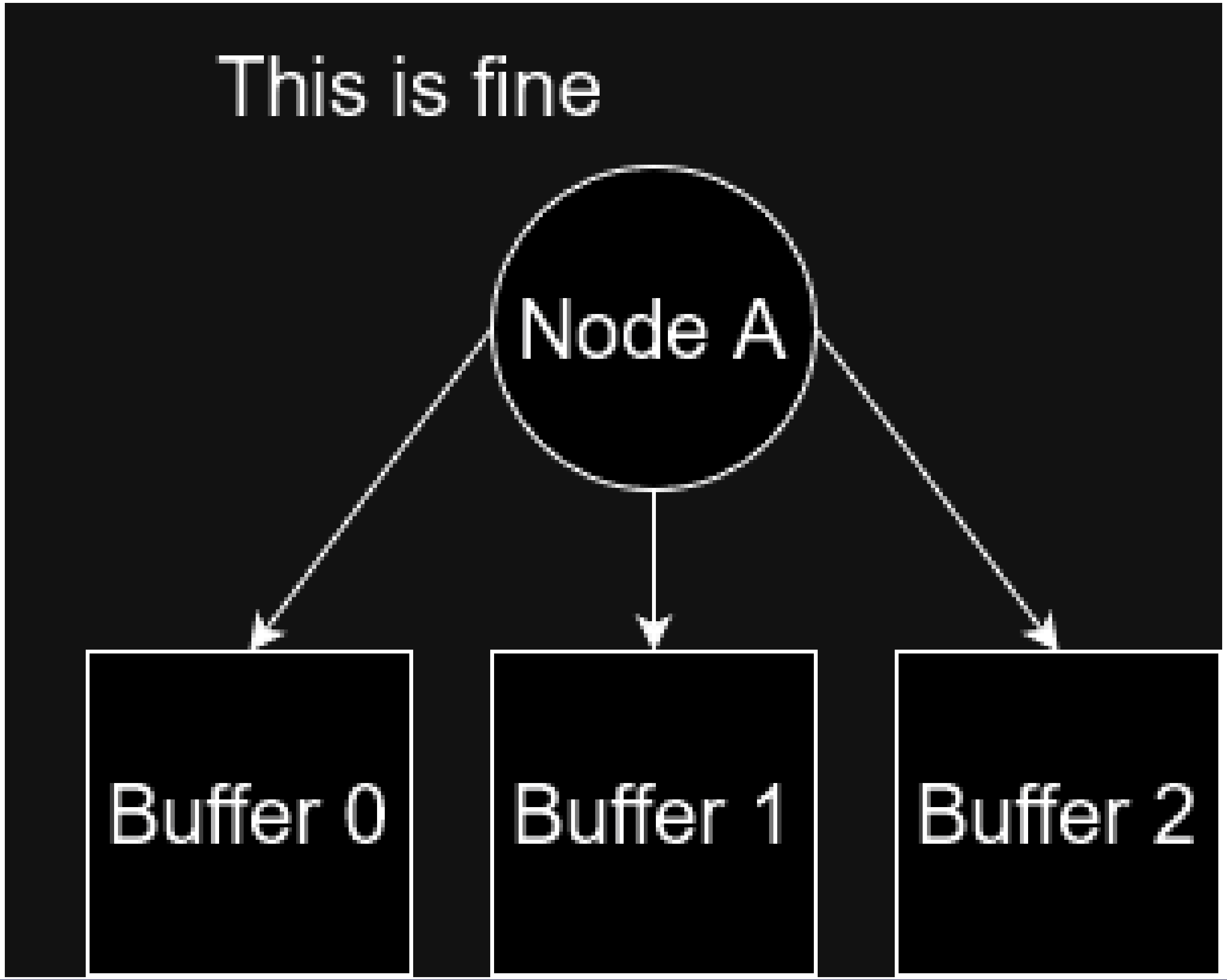
Computational Graphs

```python
# x represents our data
def forward(self, x):
  # Pass data through conv1
  x = self.conv1(x)
  # Use the rectified-linear activation function over x
  x = F.relu(x)

  x = self.conv2(x)
  x = F.relu(x)

  # Run max pooling over x
  x = F.max_pool2d(x, 2)
  # Pass data through dropout1
  x = self.dropout1(x)
  # Flatten x with start_dim=1
  x = torch.flatten(x, 1)
  # Pass data through ``fc1``
  x = self.fc1(x)
  x = F.relu(x)
  x = self.dropout2(x)
  x = self.fc2(x)

  # Apply softmax to x
  output = F.log_softmax(x, dim=1)
  return output
```

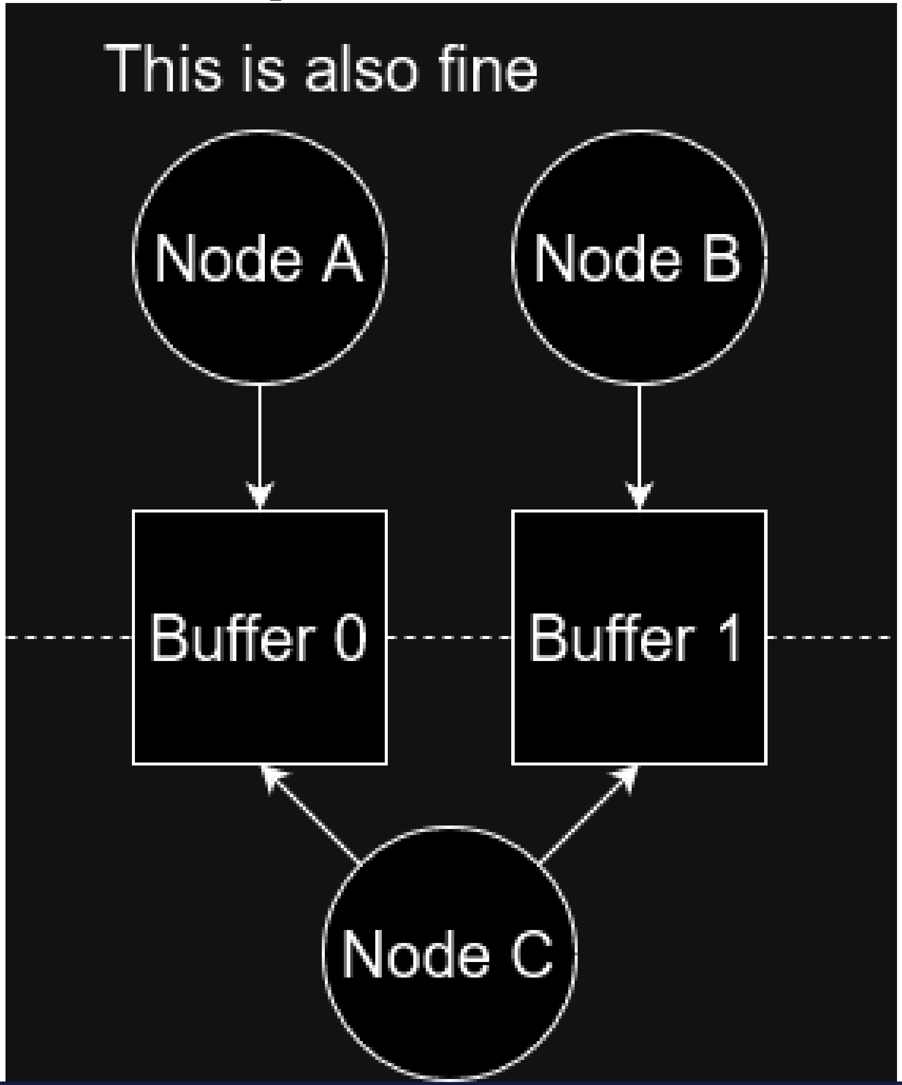
Image Link



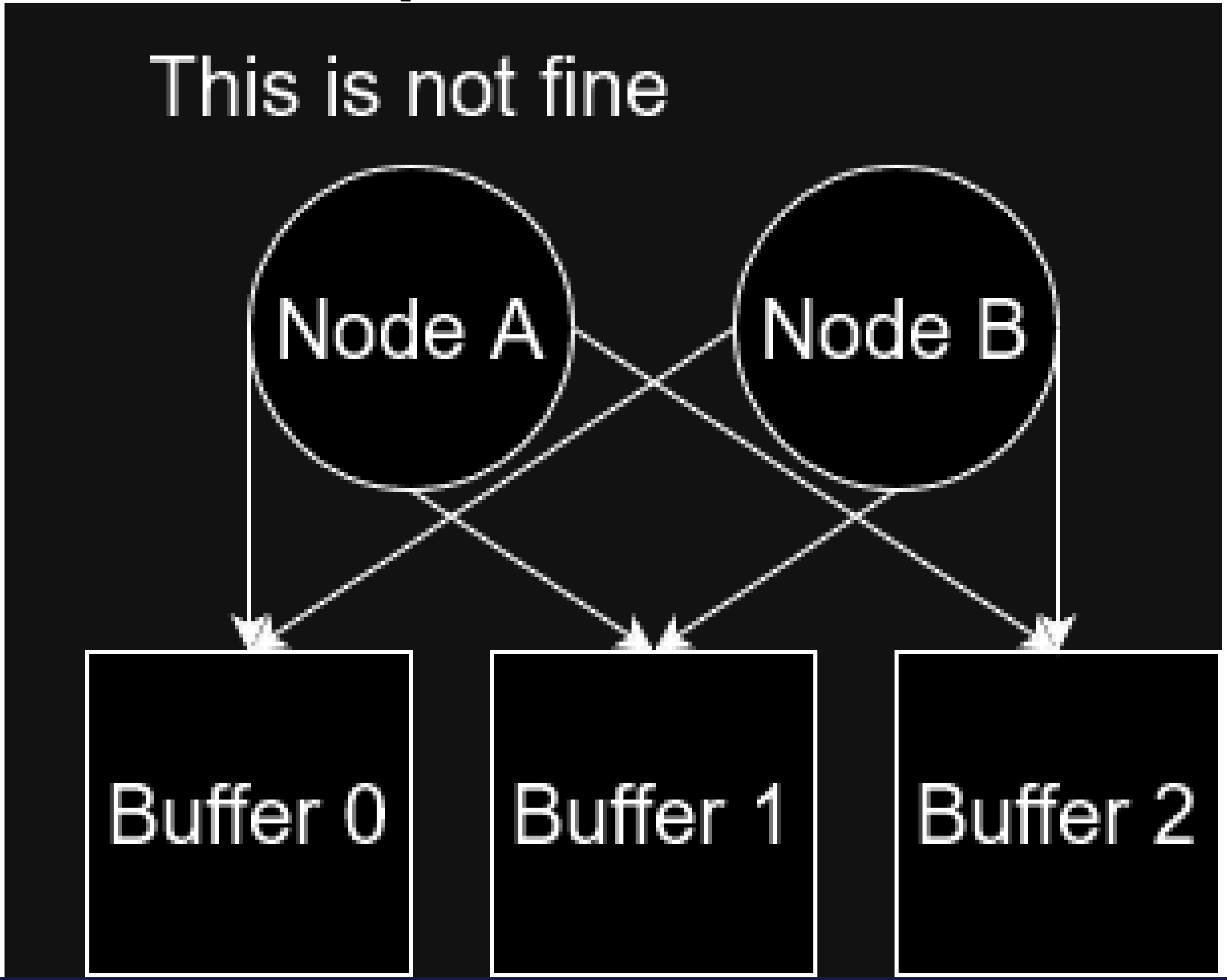
Computational Graphs



Computational Graphs



Computational Graphs & The Borrow Checker



Computational Graphs

These are the same restrictions enforced by Rust's borrow checker

If we build our own computational graph system, we can enforce our own restrictions and checks

This can happen either at compile time, just-in-time or at runtime

Garbage Collectors

Reference counting in languages like Python, C# and Java

Everything is hidden behind a shared pointer!

Add a garbage collector!

Exercises

Draw, write and discuss (the top exercises)