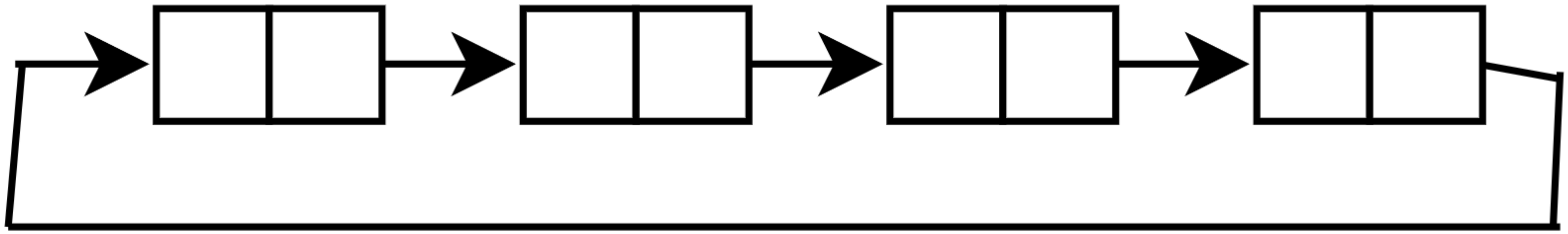
Announcements

- Homework is due next Tuesday! Moyeen will be hosting office hours from 2:20pm-4:20pm in GS 934 tomorrow (Friday)!
- There was a typo with the homework. Question 8 has been corrected on the website now.

Topic 1: Review

What is this?



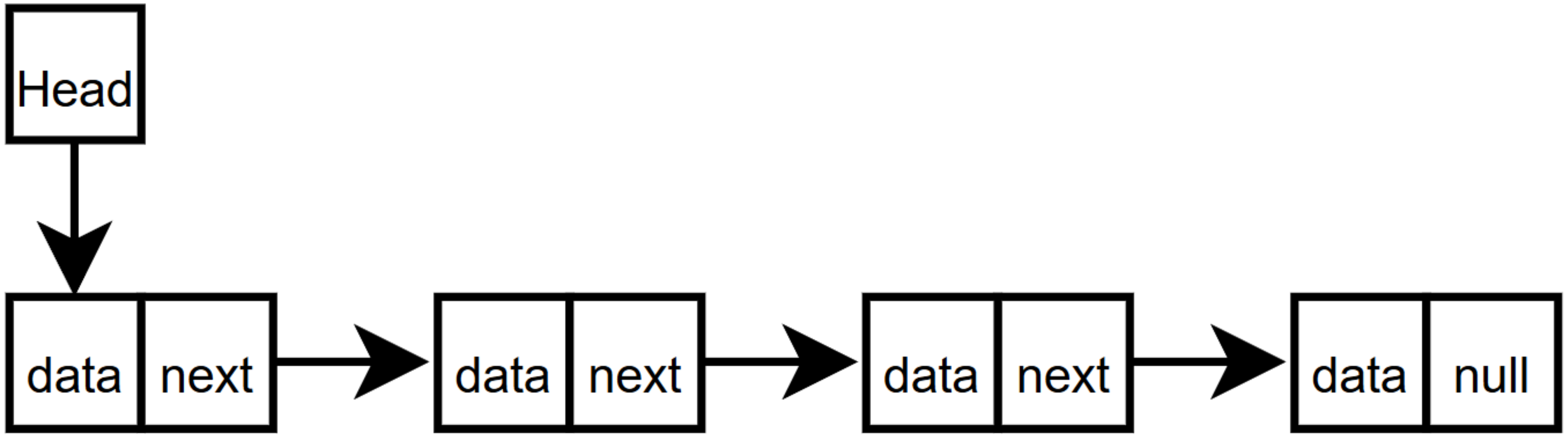
- A circular linked list! …But something’s missing…
- We need to be able to access the list. Hence, we need a “head” pointer

Data Structures

- We can use linked lists to make basic data structures, like a queue or a stack
- A **Queue** takes data in on one end, and outputs the oldest data on the other end (like waiting in a line)
- A **Stack** take in data on one end, and output the most recent data on the same end (like a pile of books or plates)
- An acronym that describes these behaviors is FIFO (first-in, first out) for queues, and LIFO (last-in, first out) for stacks

Back to Linked Lists

- There are lots of different types of linked lists. First, there's the **singly-linked list**



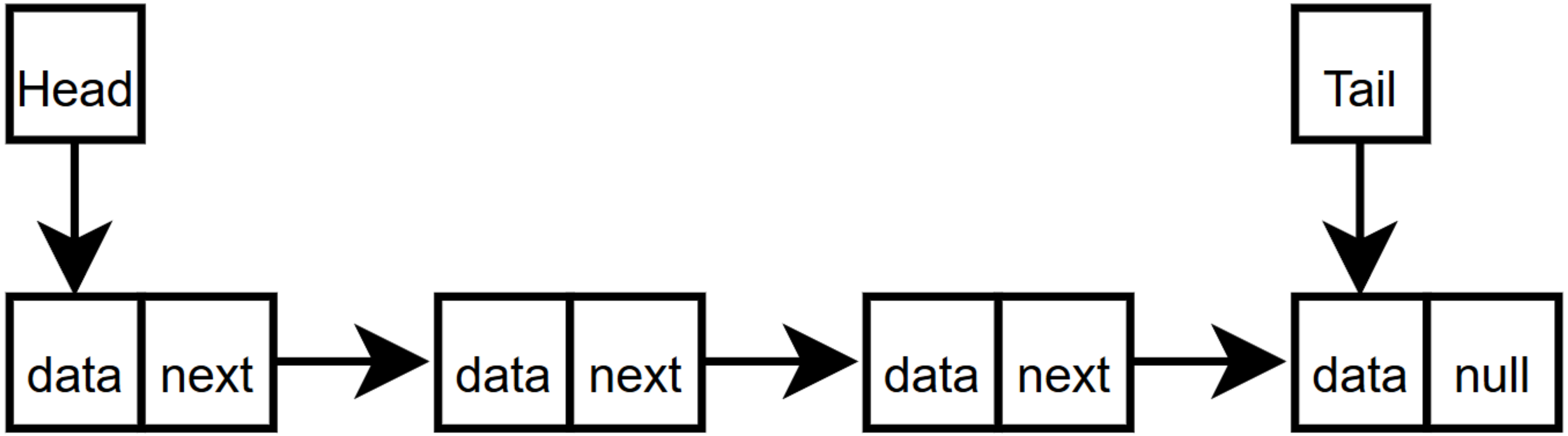
- Remember python's list data structure? The underlying data structure is not actually a linked list at all, it's an array!

Back to Linked Lists

- So we have Singly-Linked Lists… What can we do with them? What are some common operations?
  - Insert
  - Delete
  - Traverse, maybe searching for a value, or find the length/size of the list
  - Sort
  - isEmpty
  - toString
  - Create
  - Destroy

Linked Lists: Tail References

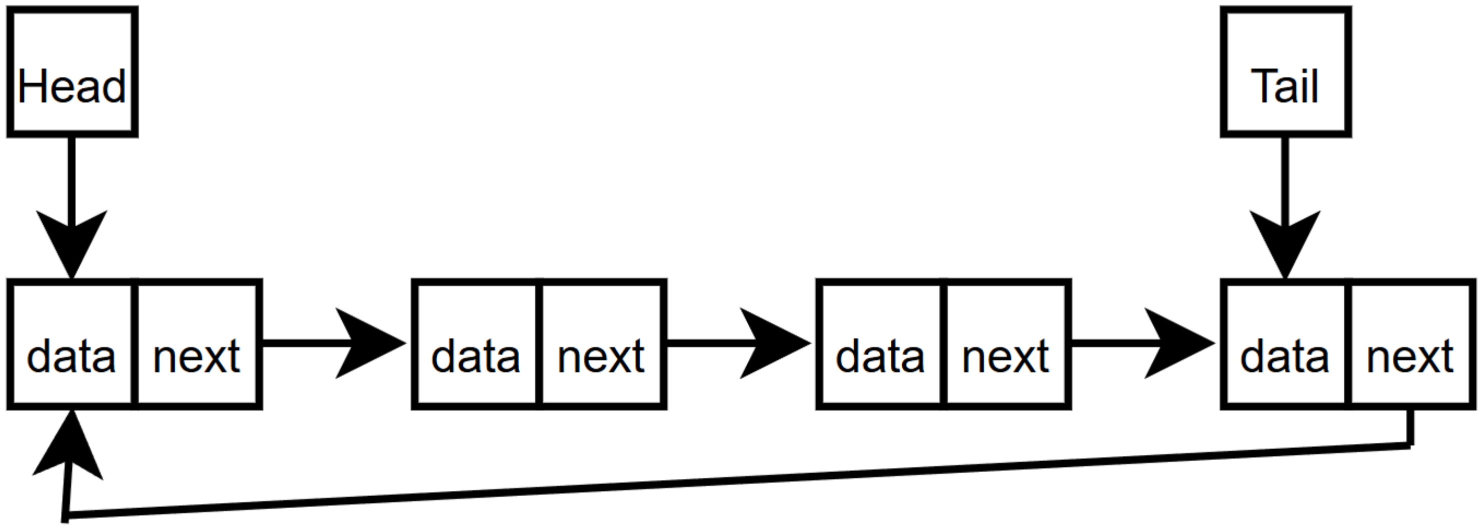
- We have the head pointer. Do we need (or want) a tail pointer? Is it worth the trouble?



- Pros: Makes appending to the end of the list easy! Great for queues!
- Cons: Anything that requires modifying of deleting the end of the list becomes a hassle. Also there’s the memory cost for an extra pointer

Circular Linked Lists

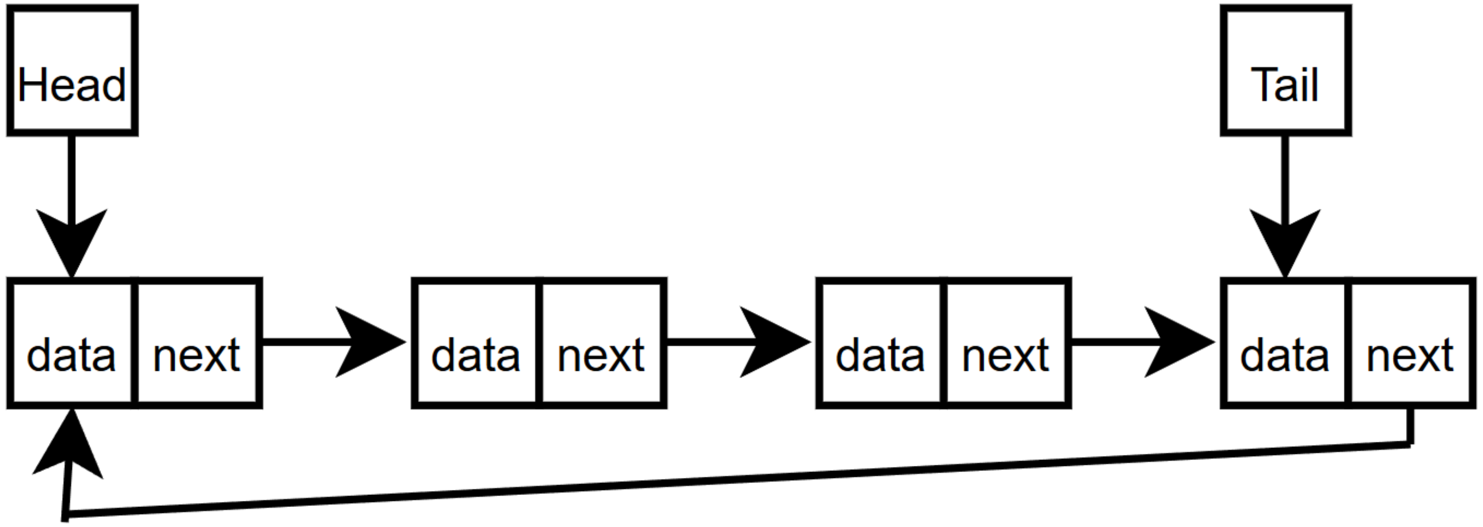
- The idea is to allow easy travel from the last node to the first



- Question: Do we even need the head pointer anymore?
  - Answer: Probably not! It only costs one extra dereference operation to get to the head from the tail pointer!

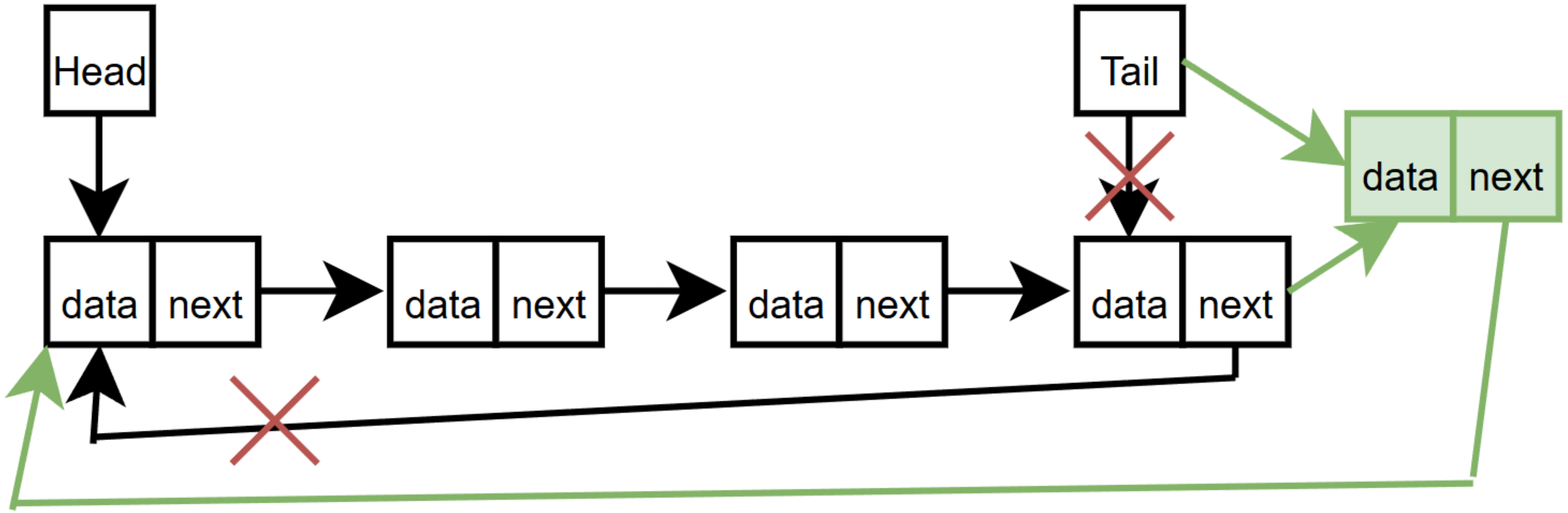
Circular Linked Lists: Operations

- Take a look at this Circular Linked-List. How would we append to it? How many operations would it take?



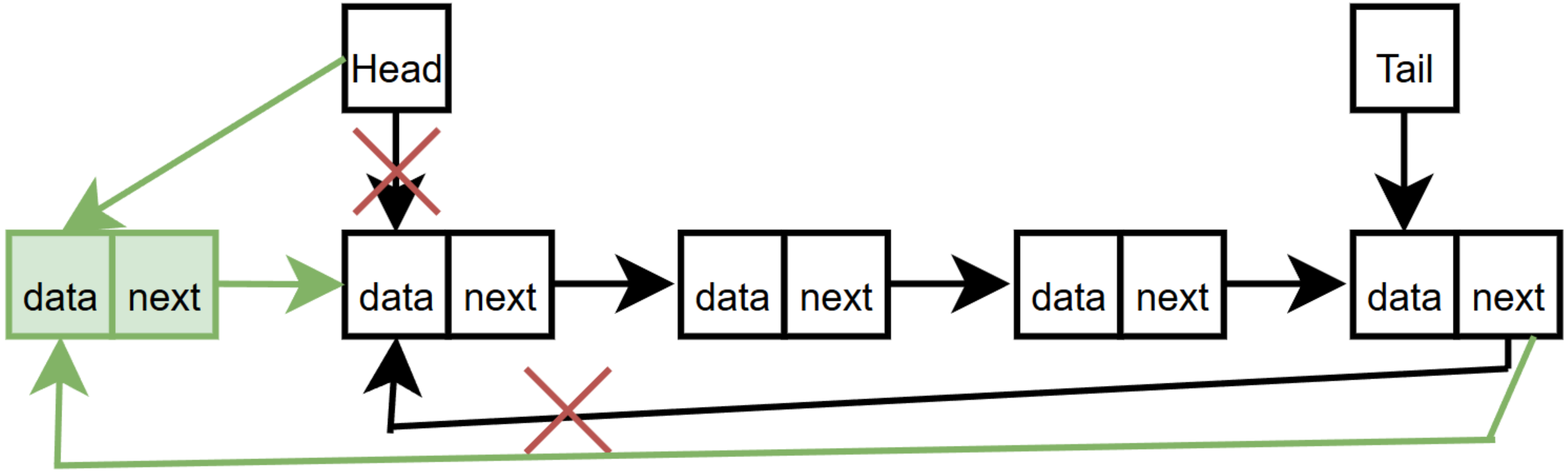
Circular Linked Lists: Operations

- Step 1: set the new node's next pointer to the front node
- Step 2: set the last node's next point to the new node
- Step 3: set the tail pointer to the new node



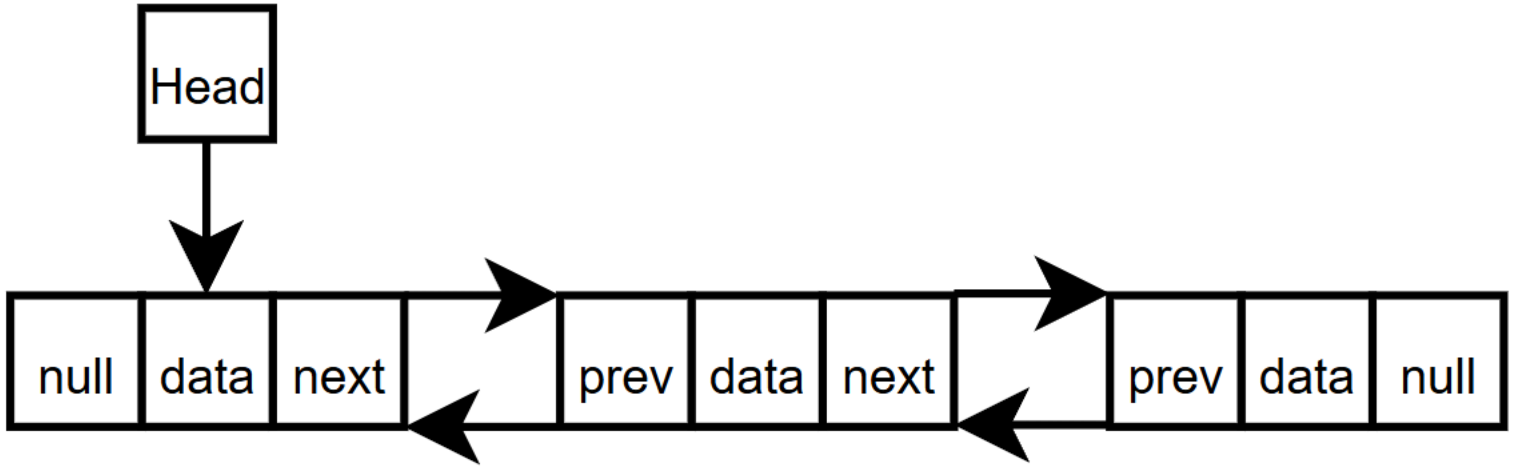
Circular Linked Lists: Operations

- What about prepending to the list? What would the steps be for that?
- Step 1: Set the new node's next pointer to the front node
- Step 2: Set the end node's next pointer to the new node
- Step 3: Set the head pointer to the new node



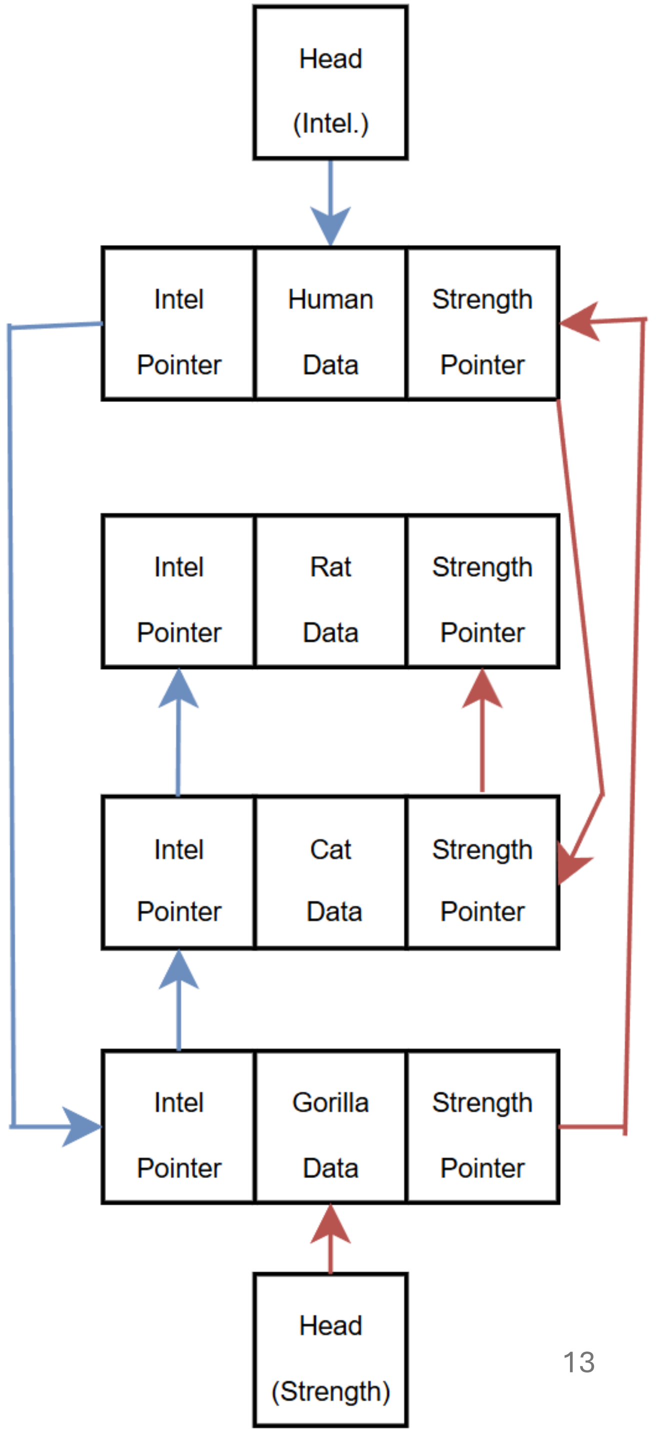
Doubly Linked Lists

- Another type of Linked Lists are **Doubly Linked Lists**. These have an extra pointer pointing to the previous node, allowing for two-way travel along the list
- Cons: memory costs, and operational costs (have to maintain both prev and next pointers for all operations)



Multilists

- A **Multilist** has multiple pointers to other nodes, representing alternate orderings for traversals
- For example, look at this figure, where animals are ranked by strength with one list, and by intelligence with another list
- Multilists can have more than two pointers, depending on how many linked lists you want the node data to be a part of
- Doubly Linked Lists can be considered a special type of multilist!

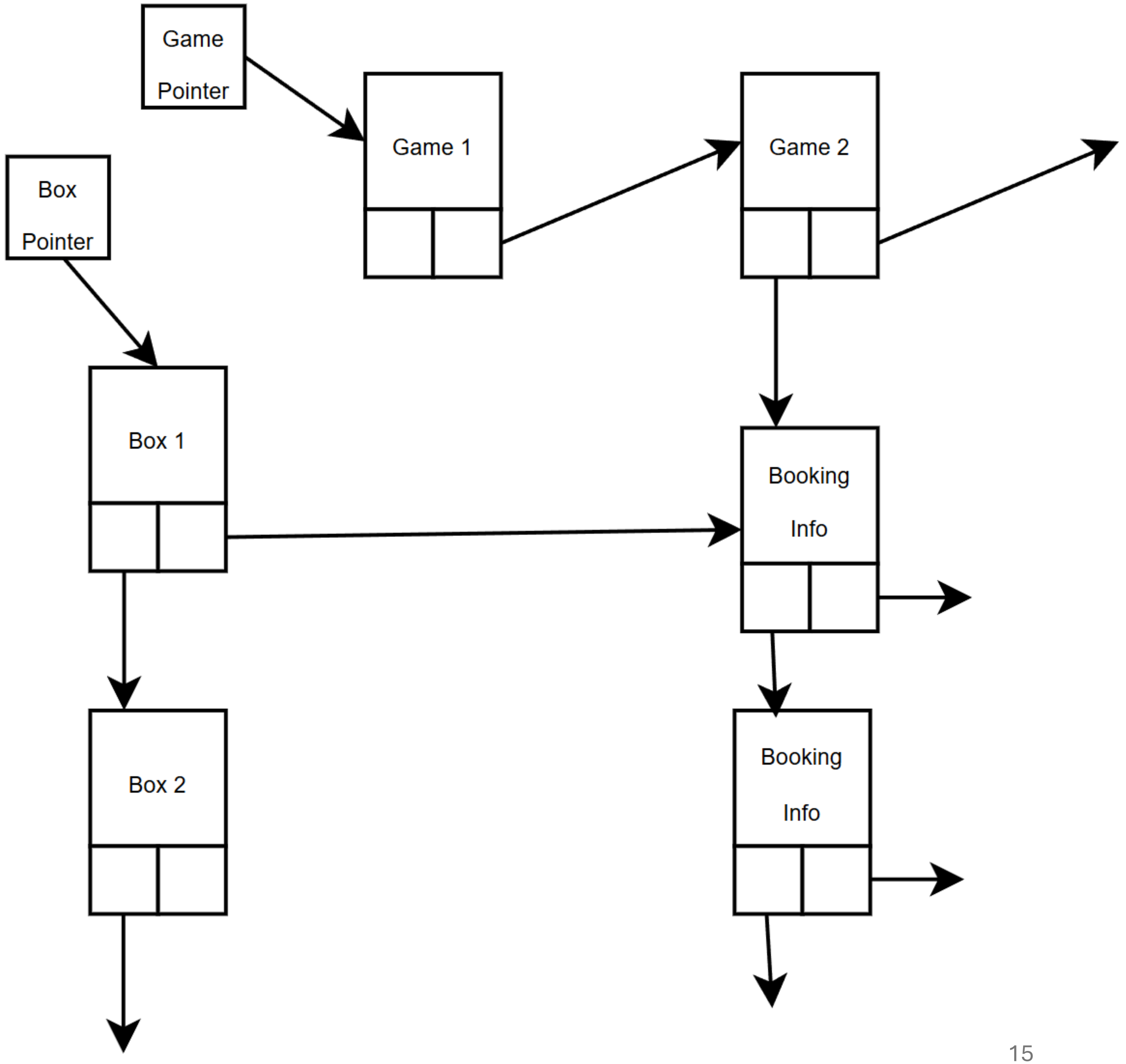


Orthogonal Lists

- An **Orthogonal List** is a group of nodes which have multiple pointers to go to its nearest neighbors node in each dimension (usually up, down, right, and left)
- This allows for efficient representation of sparse matrices (matrices that are mostly empty) or data where traveling by ‘dimension’ is useful.
- For example, consider software that books tickets for a football stadium by Box. People might want to buy tickets for a game, or for a season.

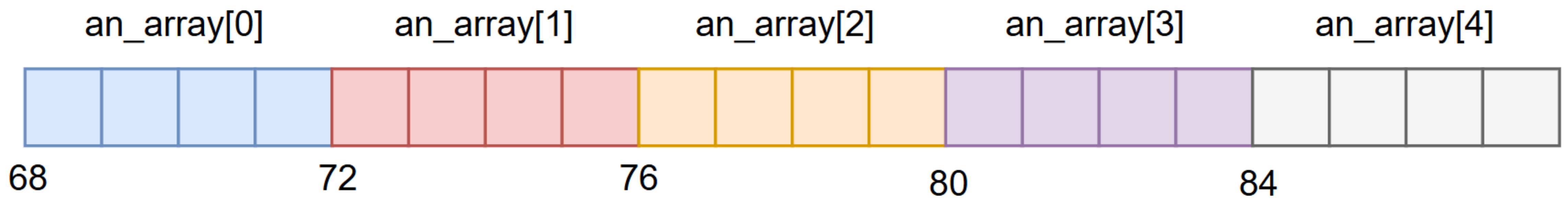
Orthogonal Lists

- How many pointers for each data object would we need if we wanted to be able to travel back/forth and up/down?
- What if we wanted to add a third list, for 3 dimensions? How many pointers would we need to travel in any direction for each data object?



1D Array Storage

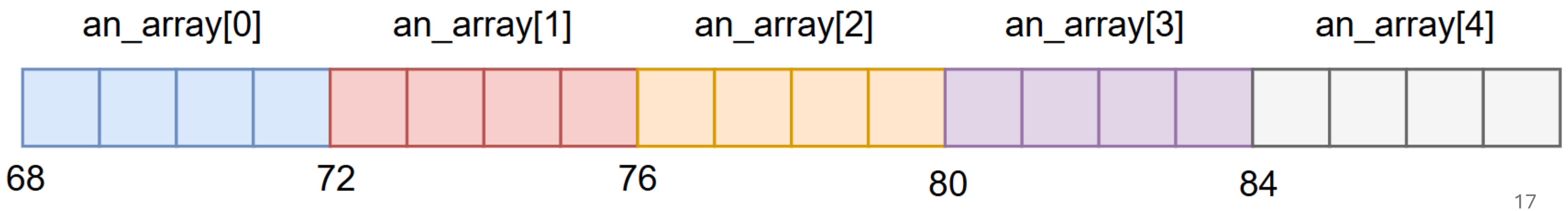
- Unlike linked lists, array elements are stored next to each other (*contiguously*) in storage.
- For example, int[] an_array = new int[5]; might look something like this:



1D Array Storage

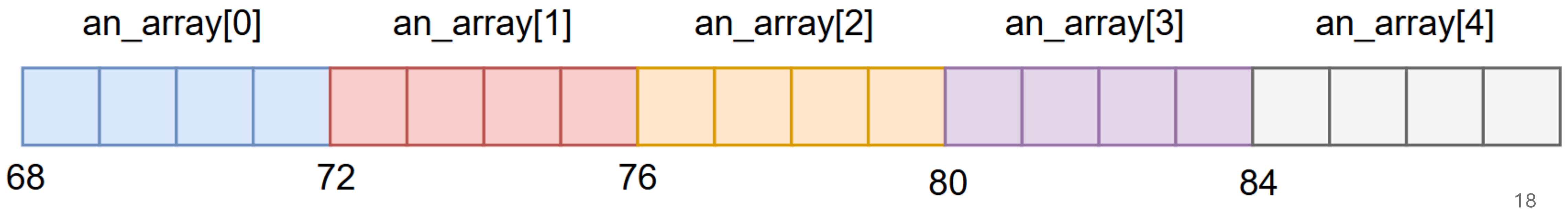
- How can we calculate the address for an arbitrary index in an array? We need to know three things:

1. Base Address of the array (68 for array below)
2. Element Size, esize for short. Since an int is 4 bytes, esize would be 4 for the array below
3. Index of the array that we're interested in. Suppose an_array[3]. What would the calculation for finding it be?



1D Array Storage

- In general, for 1-D arrays, the formula for finding an element is: base_address + esize * index
- Another way of thinking about it is: where do we start, how many jumps to the element we want, and how large does each jump have to be?
- Two final notes:
  - This is a pretty efficient operation!
  - The size of the array doesn't matter for the operation!



2D Array Storage

- First off, what does a 2D array look like in Java?
  - int twod_array[][] = new int [3][4];
  - int twod_array[][] = {{6, 0, 8, 2}, {1, 5, 3, 6}, {9, 4, 7, 1}};
- Which is the row, which is the column for this example?
  - [3] is the rows, [4] is the columns. We’re listing the data row-by-row, **not** column-by-column

2D Array Storage

- So, how do we store a 2D structure in 1D memory? Two options:
- Row-Major Order (RMO): C, C++, Pascal
- Column-Major Order (CMO): Fortran, R, MATLAB
- What about Object-Oriented languages? We'll discuss them in a bit.

	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1

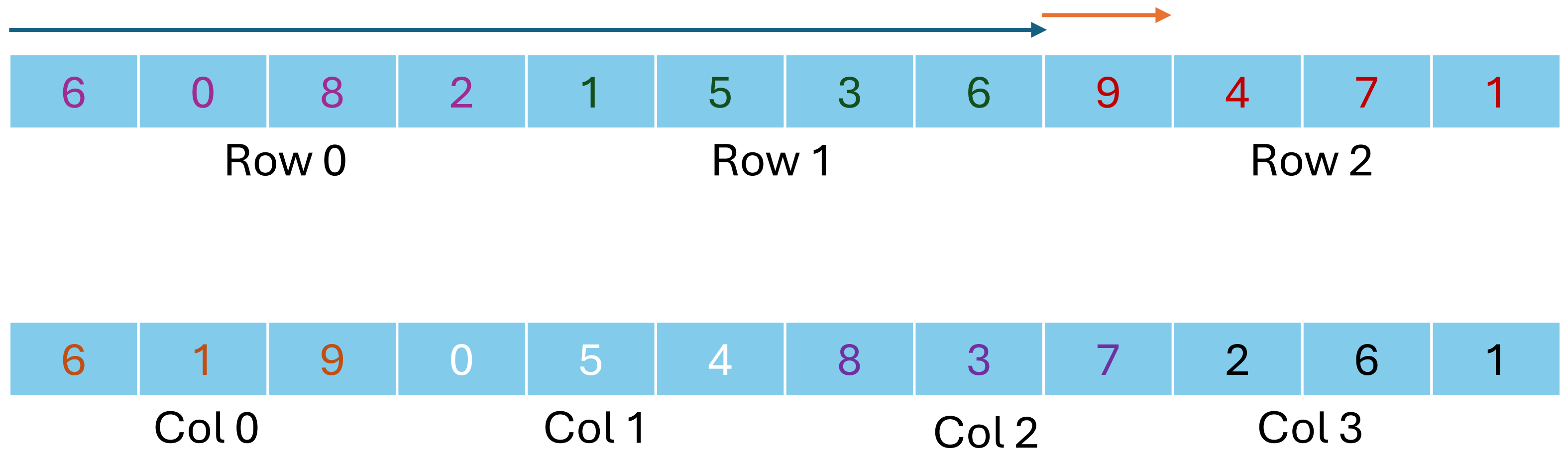
6	0	8	2	1	5	3	6	9	4	7	1
Row 0				Row 1				Row 2			

6	1	9	0	5	4	8	3	7	2	6	1
Col 0			Col 1			Col 2			Col 3		

2D Array Storage

- So we have our ways of storing the data in 1D… But how will this change our address calculation? Suppose we wanted to find twod_array[2][1]. First, what is the number in that spot?
  - 4! Row 2, Column 1. Now, how would we have gotten there, with RMD?
- RMD, we had to jump over the first 2 rows entirely! Then we skipped over 1 element for the columns…

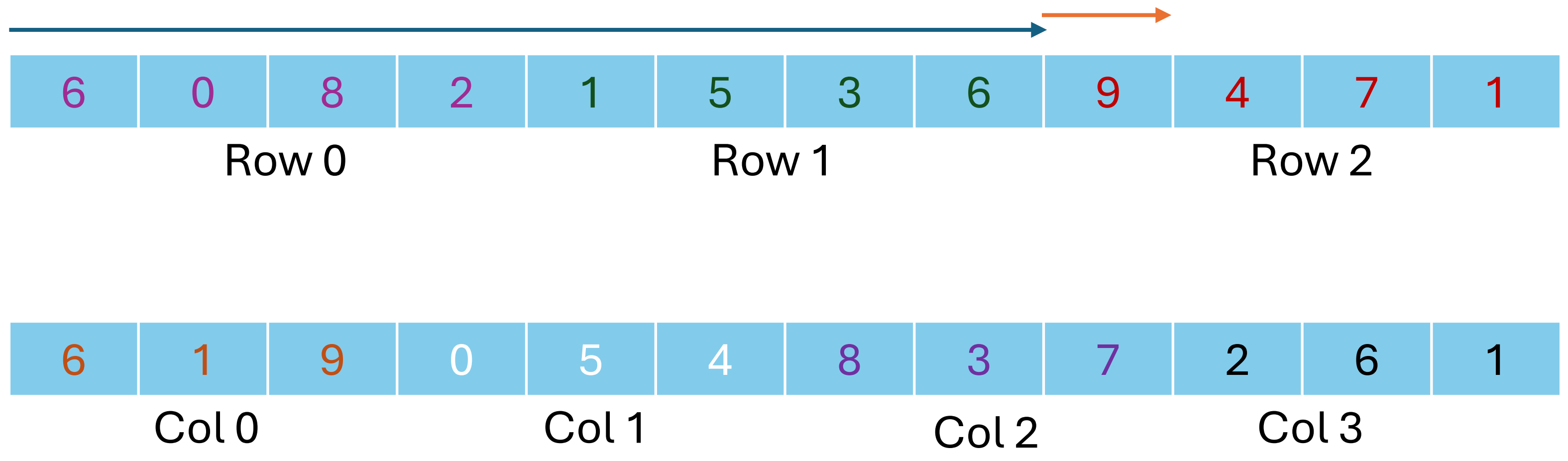
	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1



2D Array Storage

- Our 1D array address calculation required three variables (base_address + esize * index). What additional variables do we need for a 2D calculation?
- We need… row *and* column indices. Plus the number of columns (**NUMCOLS**) per row (how else would we know the size of our jumps to skip over the rows?)

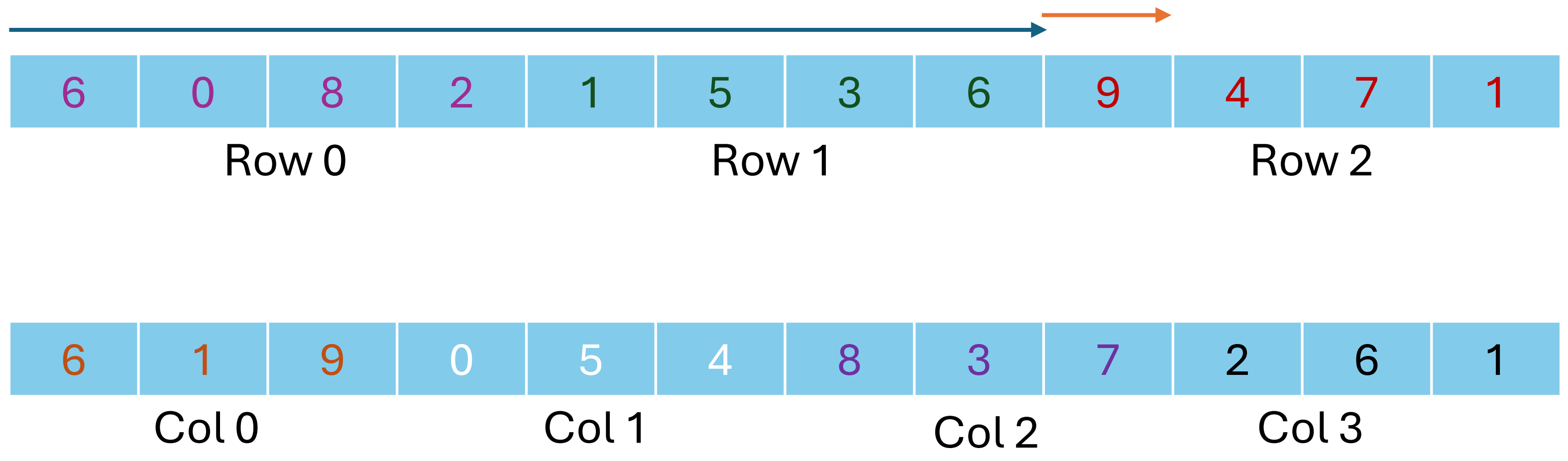
	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1



2D Array Storage

- So, our equation for 2D calculations, for RMO, is:
- address = base + NUMCOLS * row_index * esize + col_index * esize
- Can refactor the esize for: base + esize * (row_index * NUMCOLS + col_index)
- This is 2 additions and 2 multiplications… still pretty efficient!

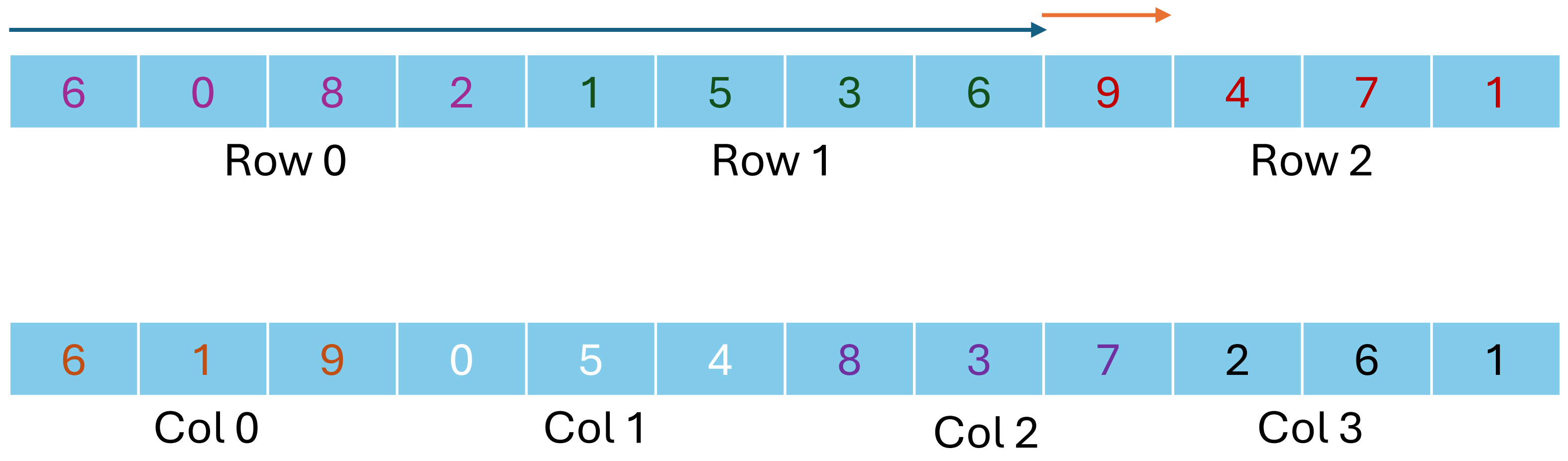
	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1



2D Array Storage

- Time to work a problem! Assume RMO. Given a 2D array (two_d) with 3 rows, 4 columns, a base address of 156, and an esize of 8 bytes, what is the address of two_d[2][1]?
- base + esize * (row_index * NUMCOLS + col_index)
- 156 + 8 * (2 * 4 + 1) = 228!

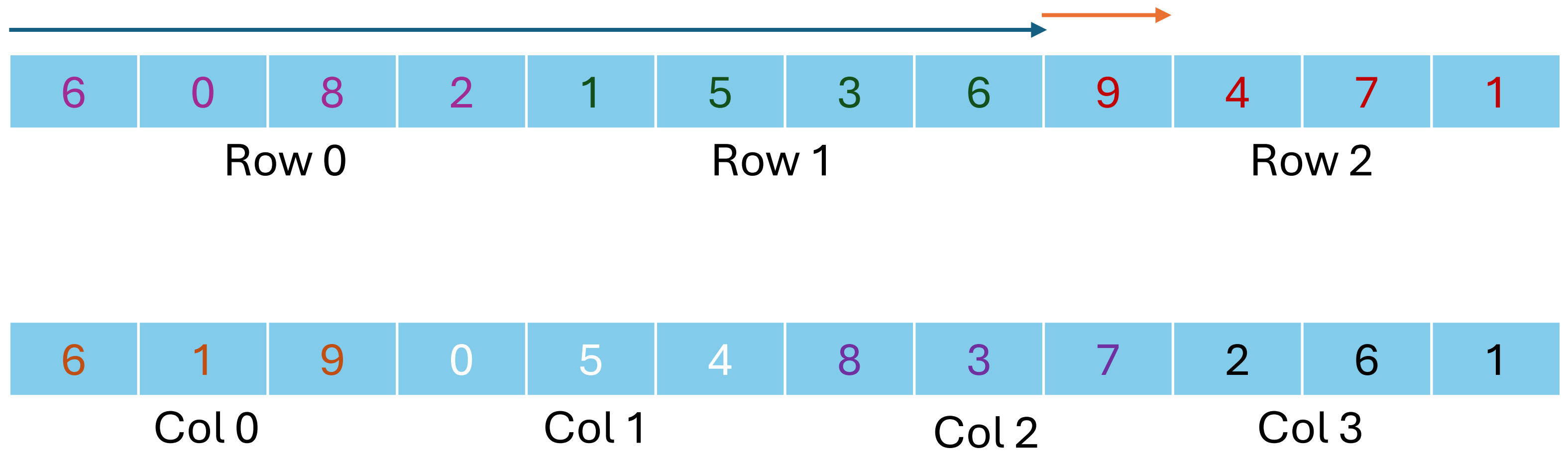
	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1



2D Array Storage

- How would the equation change for CMO? Good studying practice to figure that out! Also good quiz question material…

	0	1	2	3
0	6	0	8	2
1	1	5	3	6
2	9	4	7	1

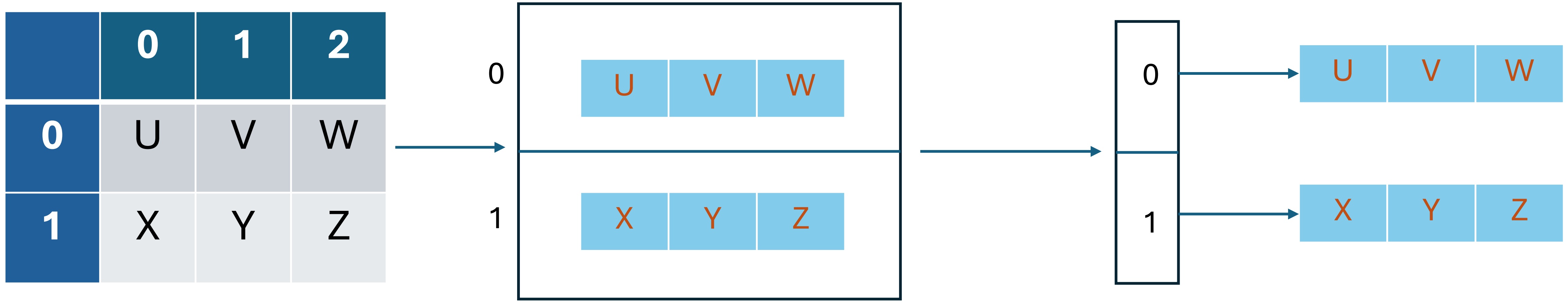


3D Array Storage

- It’s the same idea, but adding one more dimension. Consider a stack of graph paper. Each page is a 2D array, so our 2D equation would work just fine if we can get to the page.
- All we have to do is skip over the number of pages till we find the one we want (our third index, multiplied by the NUMROWS and NUMCOLS)

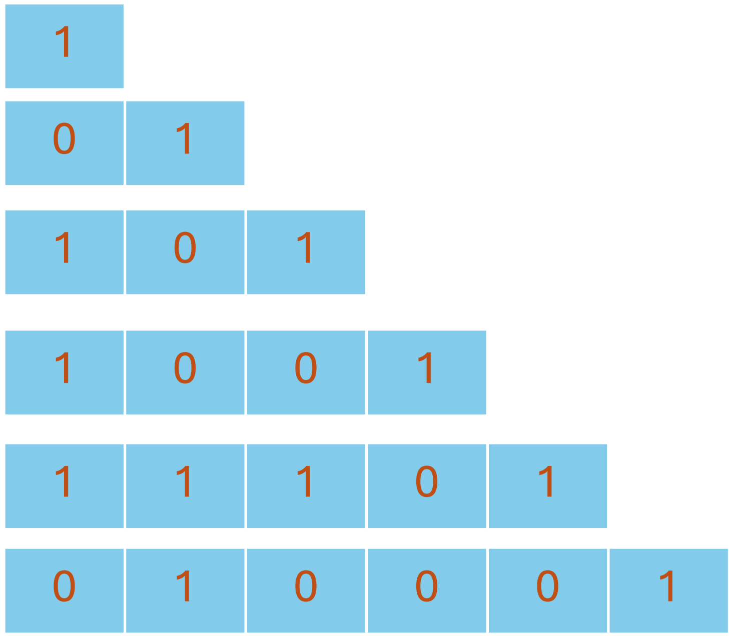
*n*D Arrays in OO Languages

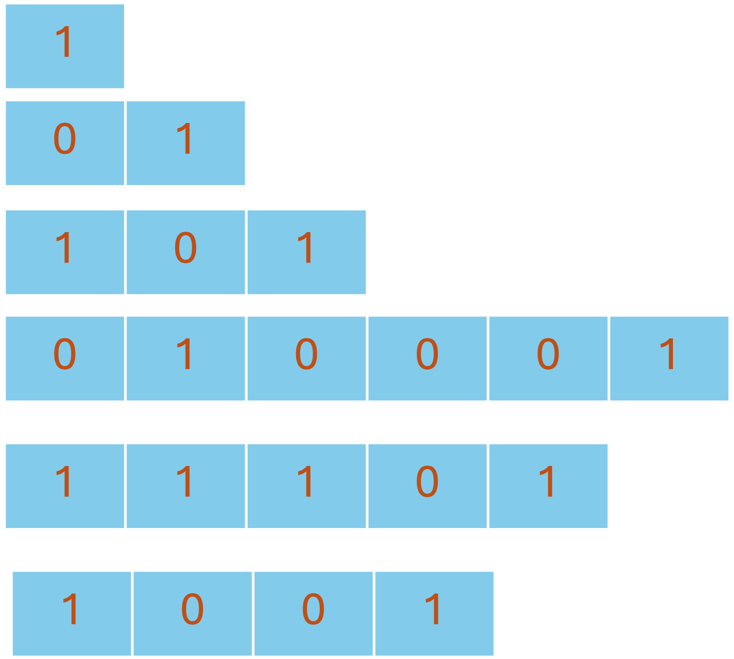
- In Object-Oriented languages, arrays are objects. A 1D array object is a contiguous collection of references to data objects. So what is a 2D array object?
- Start with a 2D array; it’s separated into 1D arrays that are stored in contiguous memory. These arrays hold references to objects.
- Then take another 1D array (called the Iliffe Vector), and have it reference the starts of the 1D arrays where the data references are stored.



*n*D Arrays in OO Languages

- Since each row is a different object, we can have different row sizes. This allows us to store info (potentially) more efficiently!
- For example, we could have a lower left triangular matrix, or a bell-curve-esque matrix

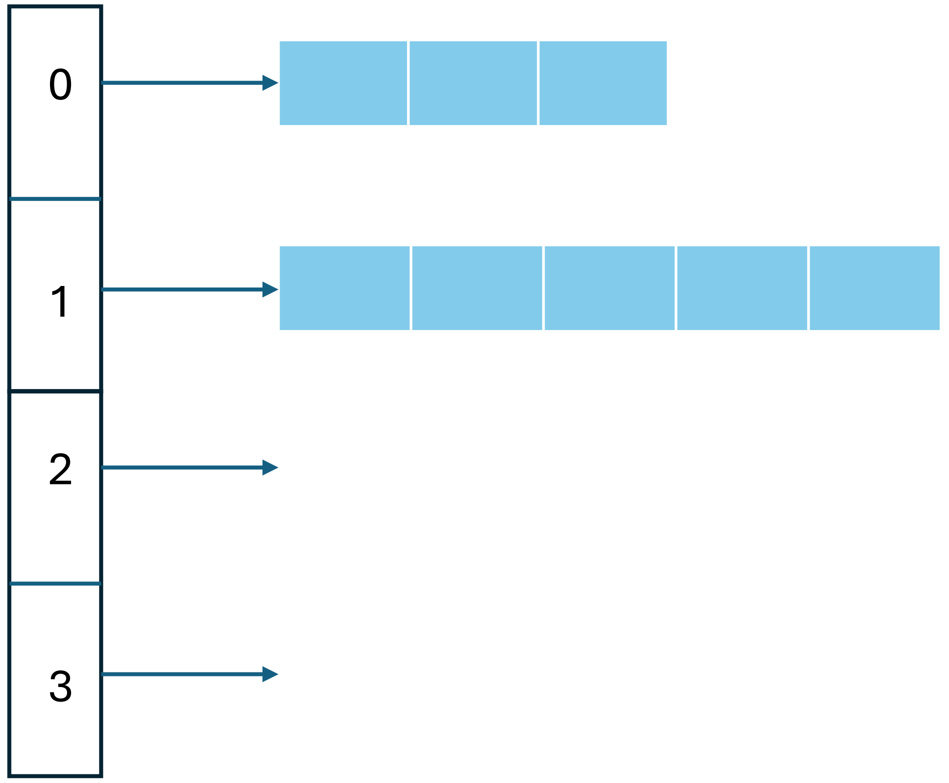




*n*D Arrays in OO Languages

- How to declare such arrays in Java?

```
int [][] nd_array;
nd_array = new int [4][];
nd_array[0] = new int[3];
nd_array[1] = new int[5];
etc…
```



Recursion Review

- To solve a problem recursively, you need answer two questions about the problem:

1. What is somewhat simpler than solving this problem for *n* items? (Could be solving it for n-1, n/2, etc.)
2. How can you use that answer to solve the size *n* problem?

Recursion Example

- Problem: Print the content of a Singly-Linked List from the front to the end.

1. What's somewhat simpler than printing a Singly-Linked List of *n* items from the front to the end?
    - Printing the list of *n-1* items!
2. How does that help print the list on *n* items?
    - We can print the head's data ourselves, and then do the n-1 item list!

Recursion Example

- Problem: Print the content of a Singly-Linked List from the front to the end.
- Let’s use our answers for the two questions to write a pseudocode algorithm!

```
subprogram printLL(given: head) returns nothing
        if head is null:
                return
        else:
                print head’s data
                call printLL with head’s successor
        end if
end subprogram
```

Recursion Example

- This is an example of *tail recursion*, where the last call in the recursive function is the recursive call
- Tail recursion (and many other types of recursion) are often easily replaced with a loop!

```
subprogram printLL(given: head) returns nothing
        if head is null:
                return
        else:
                print head’s data
                call printLL with head’s successor
        end if
end subprogram
```

Recursion Example 2

- Problem: Print the content of a Singly-Linked List from the *end to the front*.

1. What's somewhat simpler than printing a Singly-Linked List of *n* items from the end to the front?
    - Printing the list of *n-1* items!
2. How does that help print the list on *n* items?
    - We'll do the n-1 item list, and *then* print the remaining item ourselves!

- How would we change our pseudocode from earlier to make it work for this new problem?
- Answer: just switch the printing line and the recursive call line!
- Note: this is *head recursion*, where the significant processing happens after the recursive call

Recursion Example 3

- Problem: Totally unlink a Doubly-Linked List

1. What’s somewhat simpler than unlinking a Doubly-Linked List of *n* items?
   - Unlinking a list of *n-1* items!
2. How does that help print the list on *n* items?
   - We can unlink a node, then do the rest of the items… But does order matter?
   - Yes; we have to get to the next node *before* unlinking the current node!

Recursion Example 3

- Problem: Totally unlink a Doubly-Linked List
  - For our pseudocode, would we have to do our recursive call before or after doing the unlinking?
  - Is this head recursion or tail recursion?
  - What is this subprogram missing?

```
subprogram unlink(given: head) returns nothing
    if head is null:
        return
    else:
        call unlink with head’s next
        set head’s prev reference to null
         set head’s next reference to null
    end if
end subprogram
```

Recursion Example 3

- Problem: Totally unlink a Doubly-Linked List
    - We're supposed to *completely* unlink the list! That includes the head and tail pointers.
    - When should we do this in the pseudocode?
    - After we're done unlinking the list… this looks an awful lot like a Destroy function for the list object!

```
subprogram unlinklist(given: nothing) returns nothing
        call unlink(Head)
        set Head and Tail to null
End subprogram
```