

Algorithms and Data Structures for Data Science

Linked Lists 2 and Multi-Dimensional Lists

CS 277

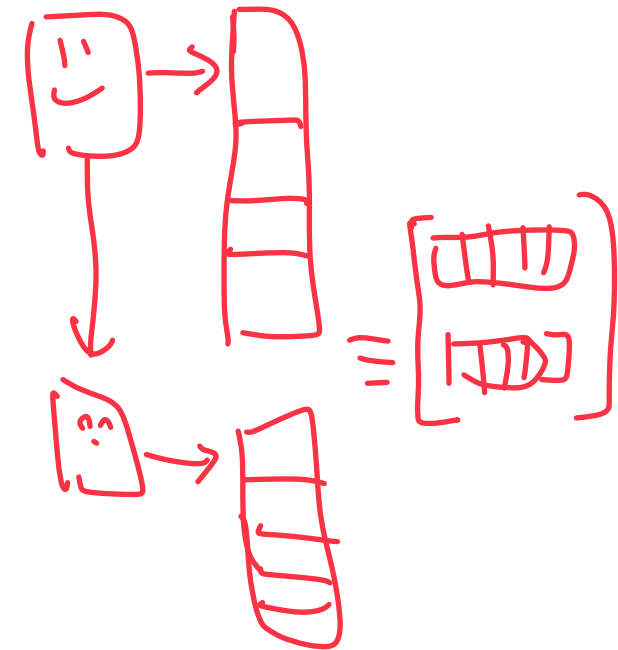
February 12, 2024

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



Announcements

Reminder: Exam 0 this week!

→ 2 question (coding)

Reminder: MP 0 due this Wednesday

↔ M17

Informal Early Feedback form out

Informal Early Feedback

An anonymous survey about the class

If 70% of class completes, everyone gets bonus points

5 or 10
points

Please provide constructive criticism and positive feedback

Learning Objectives

Review Big O in the context of linked lists

→ Implementation

Be able to justify the choice of a linked list vs array list

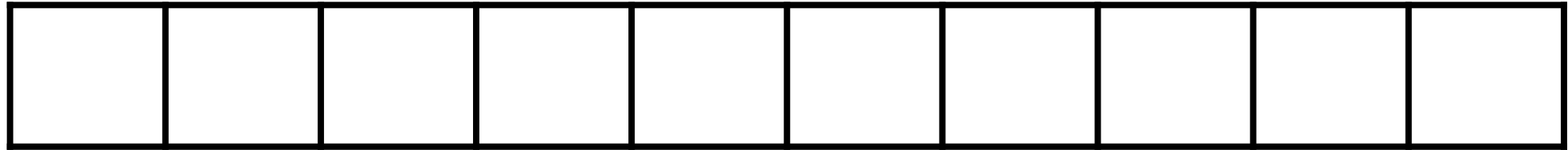
Extend knowledge of lists into two dimensions

Create and modify 2D lists using built-in and NumPy methods

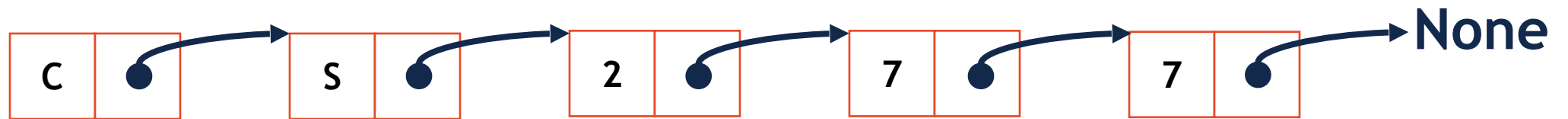
→ .. Stack
Queue
Recursion
Trees

(Theoretical) List Implementations

1. Array List



2. Linked List



Linked List Node

new object

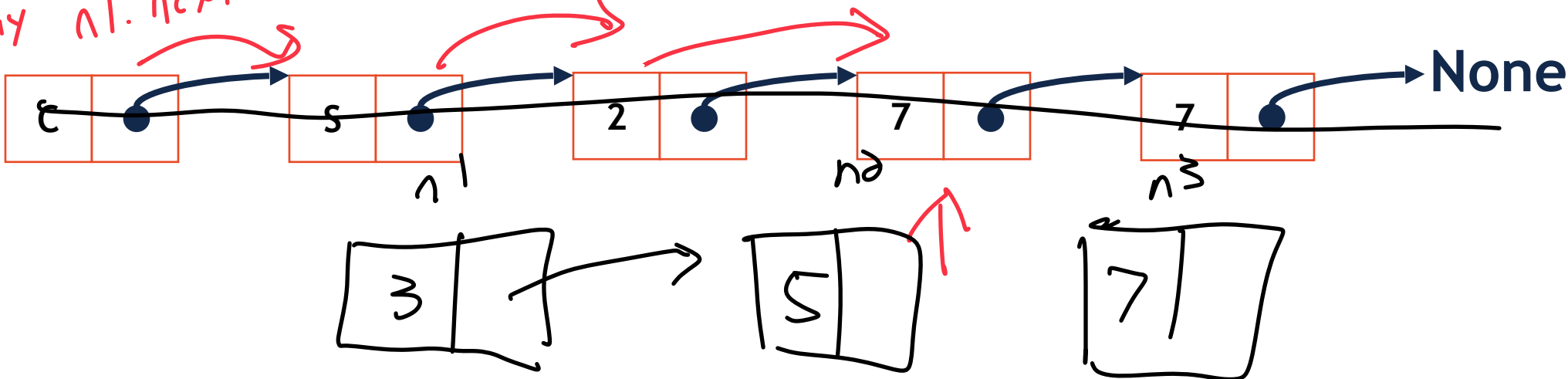
```
1 class Node:  
2     def __init__(self, data, next=None):  
3         self.data = data  
4         self.next = next  
5
```

```
1 n1 = Node(3)  
2 n2 = Node(5)  
3 n3 = Node(7)  
4 n1.next = n2  
5 n2.next = n3  
6  
7 curr = n1  
8 print(curr.next.next.data)
```

instance variable

self = n1

Just for n1
when I say n1.next

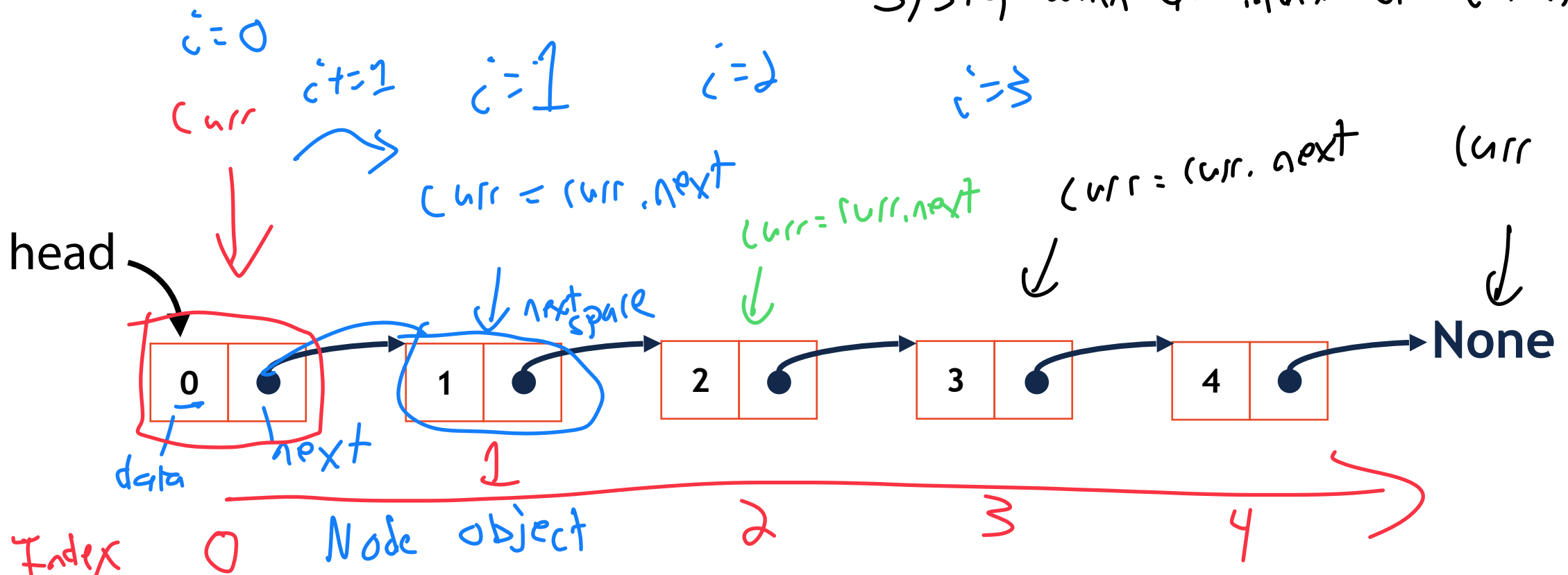


In-Class Exercise: Linked List `__getitem__()`

```
1 ll = linkedList()  
2 for i in range(5):  
3     ll.add(i)  
4  
5 print(ll[3])
```

ll(5)

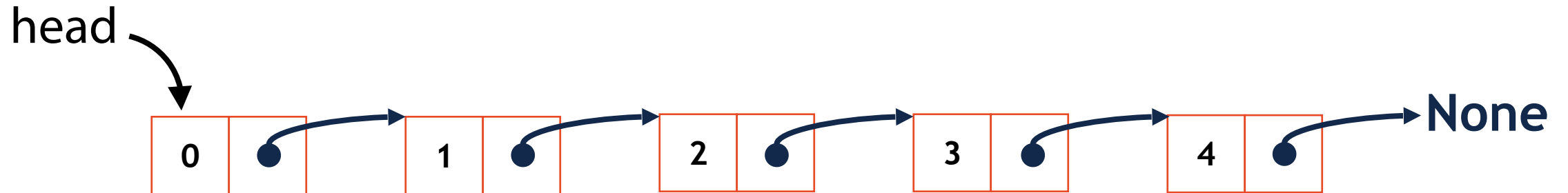
- 1) Create a new node variable
`curr = ll.head`
- 2) LOOP by setting `curr = curr.next`
- 3) Stop when at index or `curr = None`



In-Class Exercise: Linked List `__getitem__()`

```
1 ll = linkedList()  
2 for i in range(5):  
3     ll.add(i)  
4  
5 print(ll[3])
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13
```

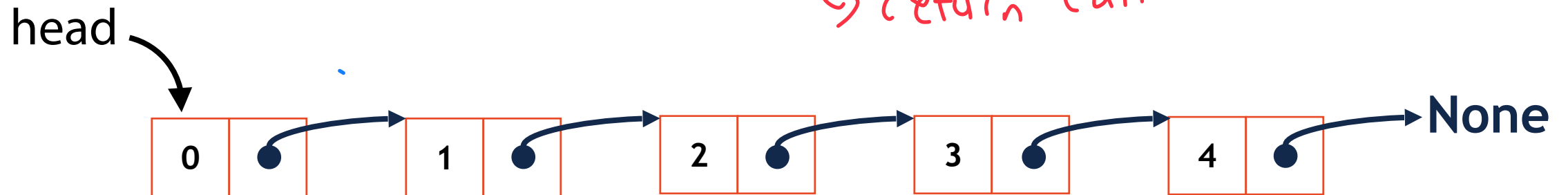


In-Class Exercise: Linked List `__getitem__()`



```
1 ll = linkedList()
2 for i in range(5):
3     ll.add(i)
4
5 print(ll[3])
```

```
1 def __getitem__(self, pos):
2     curr = self.head
3
4     i = 0
5     while (curr and i < pos):
6         curr = curr.next
7         i += 1
8
9     if i == pos:
10        return curr
11    else:
12        raise ValueError("Out of bounds")
13    return None
```



Handwritten annotations:

- `ll.head` (line 1): `ll.head`
- `self.head` (line 2): `self.head`
- `curr = self.head` (line 2): `curr = self.head`
- `curr` (line 5): `curr`
- `curr.next` (line 6): `curr.next`
- `curr` (line 9): `curr`
- `return curr` (line 10): `I found index`
- `raise ValueError("Out of bounds")` (line 12): `raise ValueError("Out of bounds")`
- `return None` (line 13): `return None`
- Line 13: `return curr`
- Line 5: `curr and i < pos`: `'s not None`
- Line 6: `curr = curr.next`: `increment to next Node`

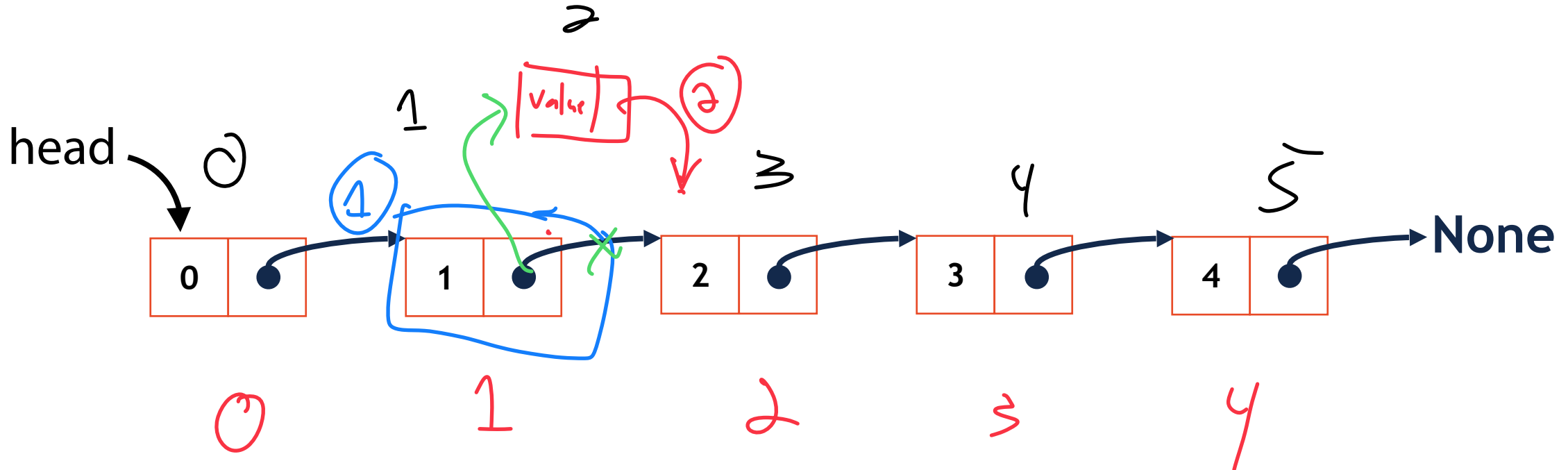
Linked List: insert()

```
1 for i in range(5):  
2     ll.add(i)  
3  
4 ll.insert("Value", 2)  
5 print(ll)
```

1. Find previous node → `--getindex--` (2)
↳ To change Node @ 2, need node @ 1

2. Create a new Node("value"
data, ^{current Node @ 2} next = None)
prev.next

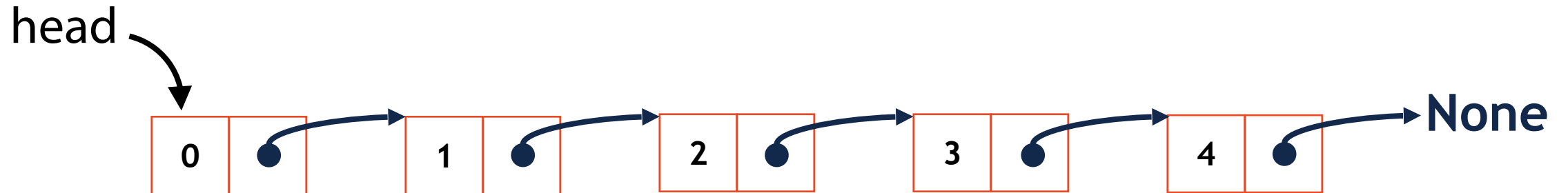
3. Set previous node's next to be new Node



Linked List: insert()

```
1 for i in range(5):  
2     ll.add(i)  
3  
4 ll.insert("Value", 2)  
5 print(ll)
```

```
1 def insert(self, data, pos=0):  
2  
3  
4  
5  
6  
7
```



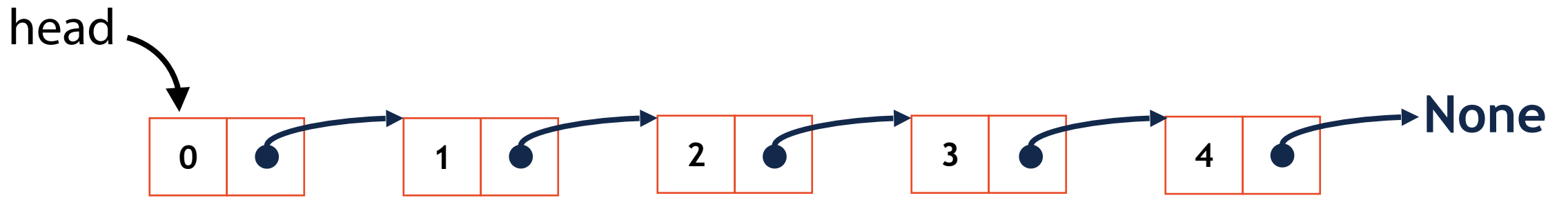
Linked List: insert()

```
1 for i in range(5):  
2     ll.add(i)  
3  
4 ll.insert("Value", 2)  
5 print(ll)
```

```
1 def insert(self, data, pos=0):  
2     if (pos == 0):  
3         self.add(data)  
4     else:  
5         prev = self.__getitem__(pos-1)  
6         temp = prev.next  
7         prev.next = Node(data, temp)
```

If prev = None (pos = -1)
None, next ?

??
pos=0
__getitem__(-1)

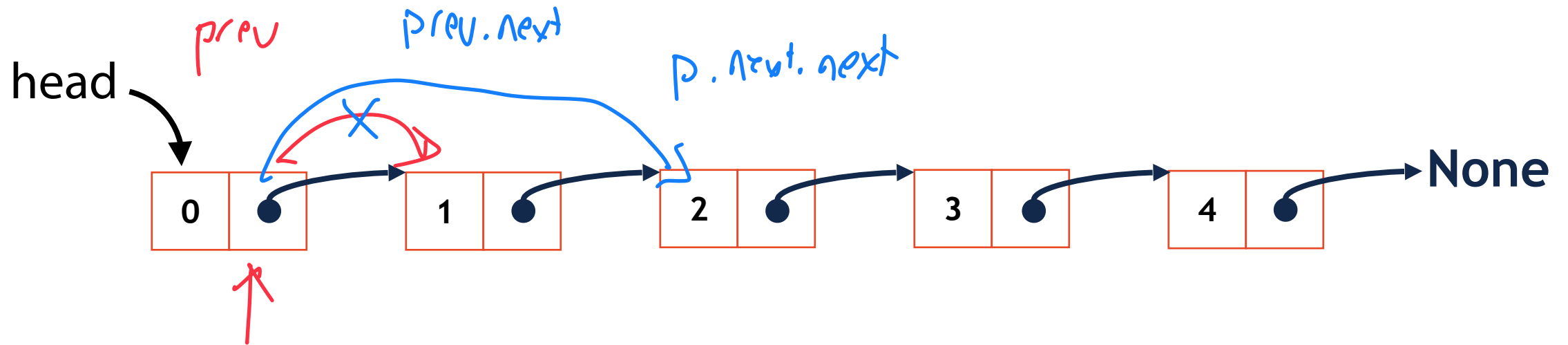


Linked List: delete()

1) Find previous

```
1 for i in range(5):  
2     ll.add(i)  
3  
4 ll.delete(1)  
5 print(ll)
```

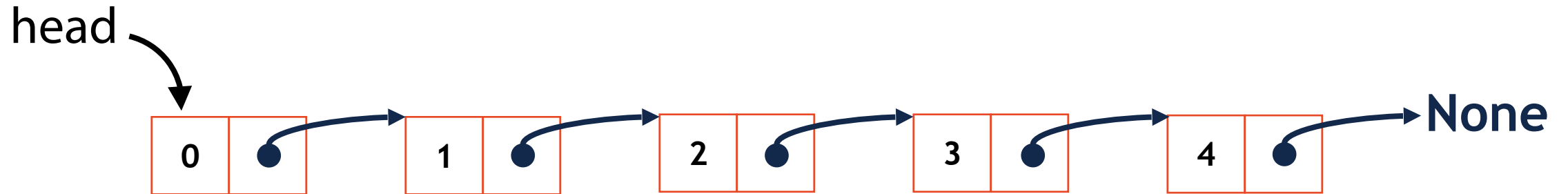
2) $prev.next = prev.next.next$



Linked List: delete()



```
1 for i in range(5):  
2     ll.add(i)  
3  
4 ll.delete(1)  
5 print(ll)
```



Linked List: delete()

removes Node @ index

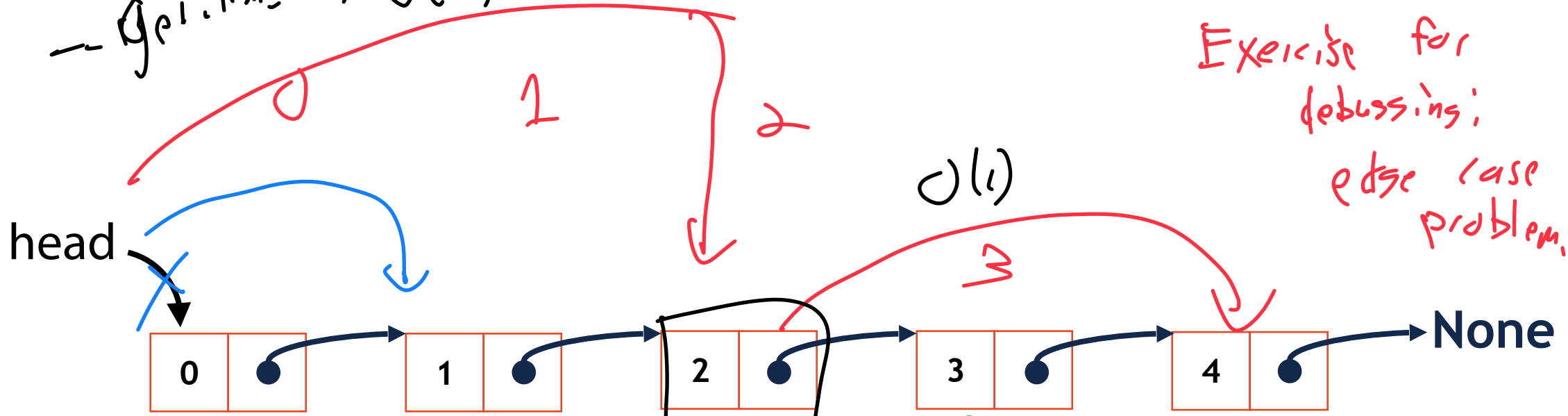
if delete head
↑ head



```
1 for i in range(5):
2     ll.add(i)
3
4 ll.delete(1)
5 print(ll)
```

```
1 def delete(self, i):
2     if i == 0:
3         self.head = self.head.next
4     else:
5         prev = self.__getitem__(i-1)
6         prev.next = prev.next.next
7
```

$-\text{__getitem__}(i) O(n)$



Exercise for debugging:
edge case problem!

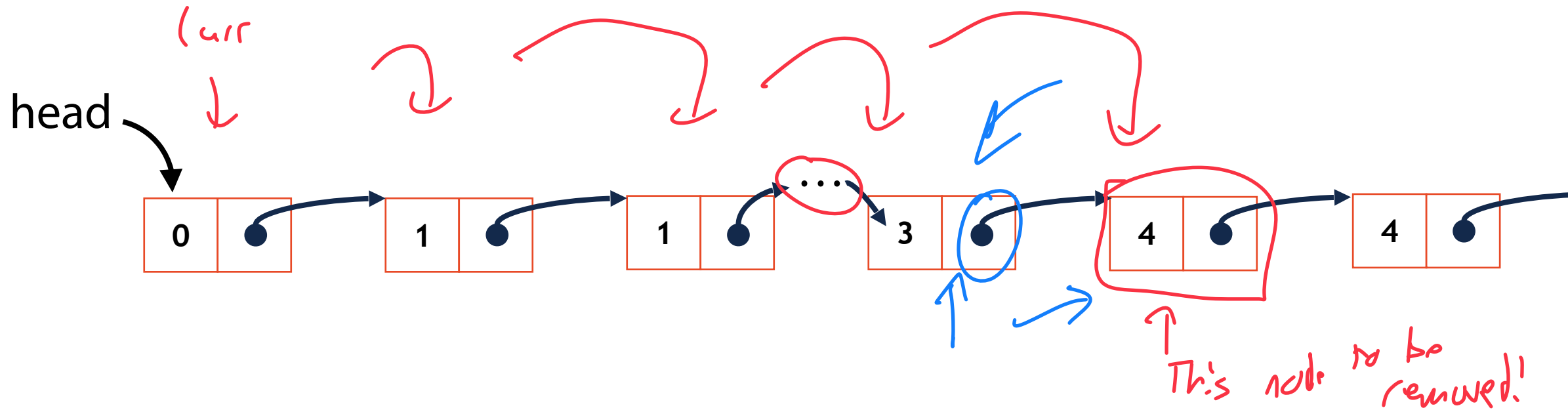
what happens to deleted nodes?

In-Class Exercise: remove()

removes first Node found by value

```
1 ll = linkedList()
2
3 for i in range(5):
4     ll.add(i)
5     ll.add(i)
6
7 ll.remove(4)
```

1) Loop through list using curr.next
↳ if `curr.next.data == val`

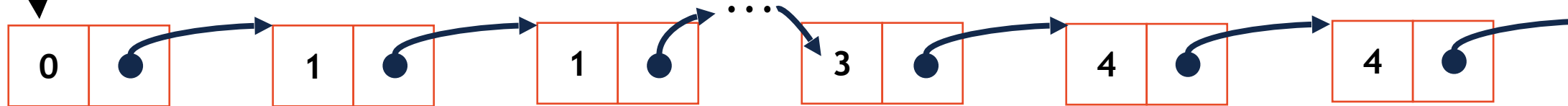


In-Class Exercise: remove()

```
1 ll = linkedList()  
2  
3 for i in range(5):  
4     ll.add(i)  
5     ll.add(i)  
6  
7 ll.remove(4)
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11
```

head

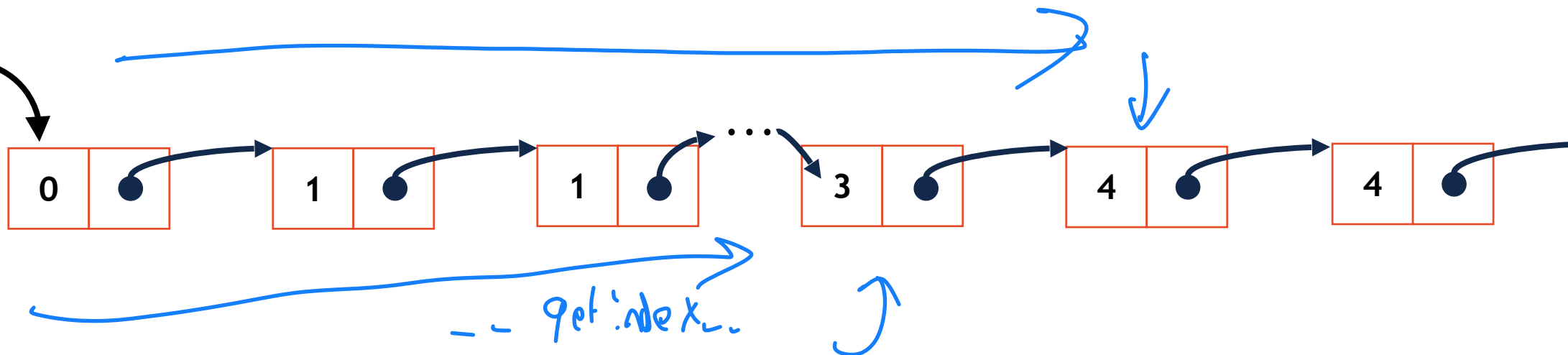


In-Class Exercise: remove()

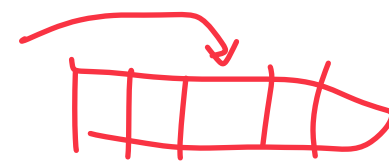
```
1 ll = linkedList()  
2  
3 for i in range(5):  
4     ll.add(i)  
5     ll.add(i)  
6  
7 ll.remove(4)
```


```
1 def remove(self, data):  
2     if self.head == data:  
3         self.head = self.head.next  
4         return  
5  
6     curr = self.head  
7     while(curr.next):  
8         if(curr.next.data == data):  
9             curr.next = curr.next.next  
10            return  
11        else:
```

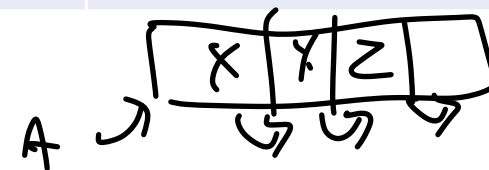
head



Array Implementation



	Singly Linked List	Array
Look up arbitrary location ↳ by <i>index</i>	$O(n)$	$O(1)$ - easily indexed
Insert after given element ↳ given <i>Node / memory loc</i>	$O(1)$ $O(1)$ - easily modified! 	$O(n)$
Remove after given element	$O(1)$	$O(n)$
Insert at arbitrary location ↳ find ↳ insert at <i>given loc</i>	Find is $O(n)$ insert is $O(1)$ $O(n)$	Find $O(1)$ insert $O(n)$ $O(n)$
Remove at arbitrary location	$O(n)$	$O(n)$
Search for an input value	$O(n)$	$O(n)$



Whats next?

1. Improve coding capabilities on multi-dimensional lists



2. Apply lists towards computational modeling problems in 2D

MPO: Python fundamentals + Big O

MP2: Lists! 😊

Programming Toolbox: 2D Arrays

Lists in Python store objects. Lists in Python are objects.

7	8	9
4	5	6
1	2	3

```
def makeMatrix():
```

make this matrix →

row 1 = [7, 8, 9]

row 2 = [4, 5, 6]

row 3 = [1, 2, 3]

return [row 1, row 2, row 3]

M =

[7, 8, 9]
[4, 5, 6]
[1, 2, 3]

|||

$M[2] = [1, 2, 3]$

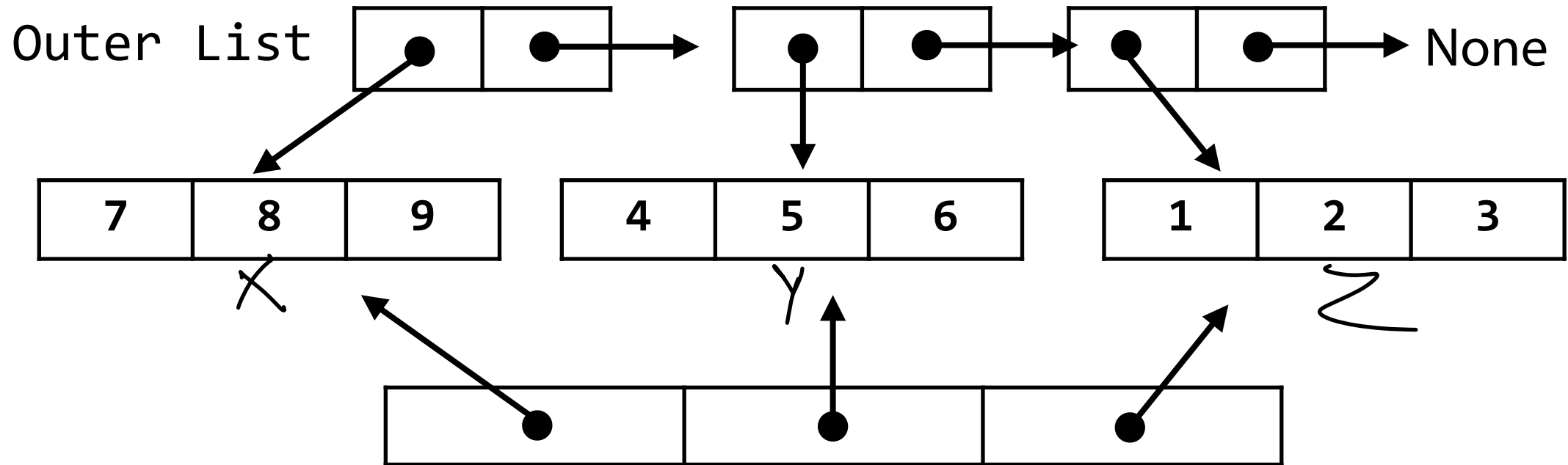
$M[1][0] = 4$

$M[0][2] = 9$

Programming Toolbox: 2D Arrays

If it helps, visualize as a Linked List storing arrays!

7	8	9
4	5	6
1	2	3



Or as an array that is *pointing* to its sub-arrays

Programming Toolbox: 2D Arrays

What shape will this code produce?

```
1  outerList = []
2
3  for i in range(5):
4      innerList = []
5
6      for j in range(5):
7          innerList.append(i+j)
8
9      outerList.append(innerList)
10
11
12
13
14
15
16
17
18
```

Programming Toolbox: 2D Arrays

What values will this list produce?


```
1 outerList = []
2
3 for i in range(5):
4     innerList = []
5
6     for j in range(5):
7         innerList.append(i+j)
8
9     outerList.append(innerList)
10
11
12
13
14
15
16
17
18
```


Programming Toolbox: 2D Arrays



What are the indices of every value of 4 in this list?

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

```
1 outerList = []
2
3 for i in range(5):
4     innerList = []
5
6     for j in range(5):
7         innerList.append(i+j)
8
9     outerList.append(innerList)
10
11
12
13
14
15
16
17
18
```

Programming Toolbox: NumPy

NumPy is optimized for multidimensional arrays of numbers

```
1 import numpy as np
2
3 # Convert list to np list
4 n1 = np.array([1, 2, 3, 4, 5, 6])
5 print(n1)
6
7 # See list shape
8 print(n1.shape)
9
10 # Modify list shape
11 n12 = n1.reshape(3, 2)
12
13 print(n1)
14 print(n12)
15
16 # Create a new list
17 n13 = np.arange(15).reshape(5, 3)
18 n14 = np.zeros((2, 5))
19
20 print(n13)
21 print(n14)
22
23
```

Programming Toolbox: NumPy

Basic operations are applied **elementwise** (to each item of a list)

```
1 n1 = np.arange(4).reshape(2, 2)
2
3 print(n1)
4
5 n12 = n1 * 2
6
7 print(n12)
8
9 # Matrix multiplication
10 # 0*0+1*4      0*0+1*6
11 # 2*0+3*4      2*2+3*6
12 print(n1.dot(n12))
```

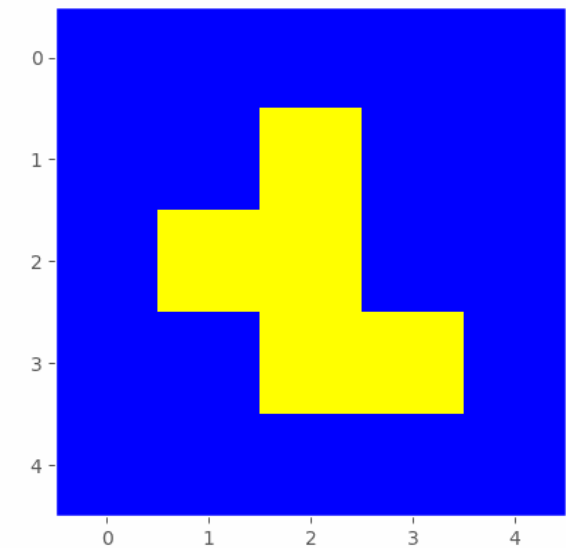
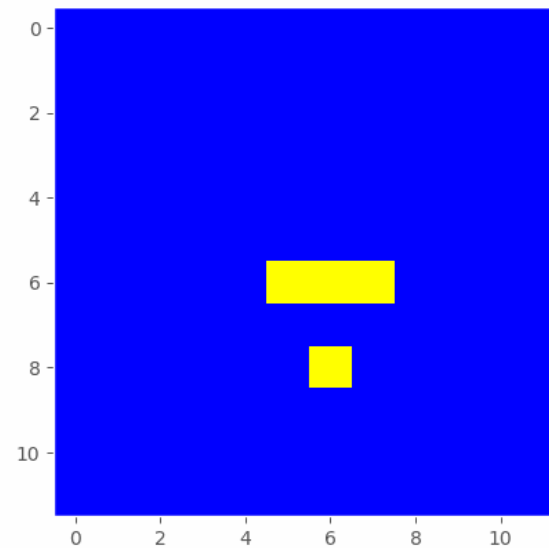
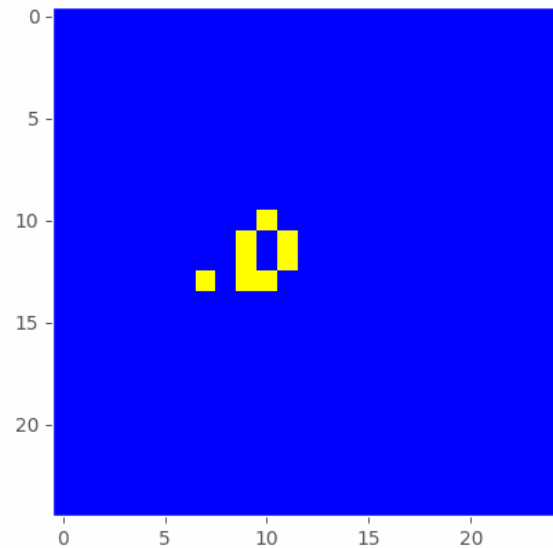
Explore on your own: <https://numpy.org/devdocs/>

Cellular Automata

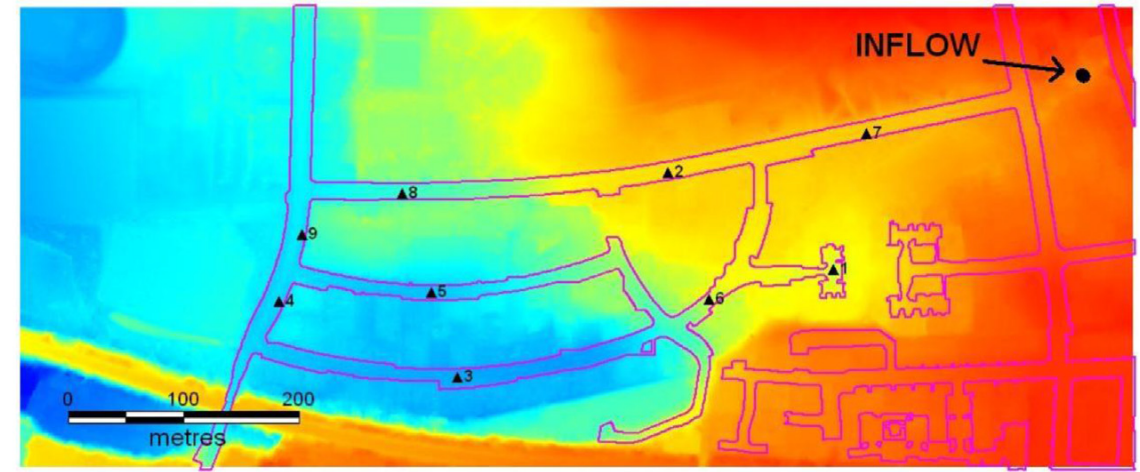
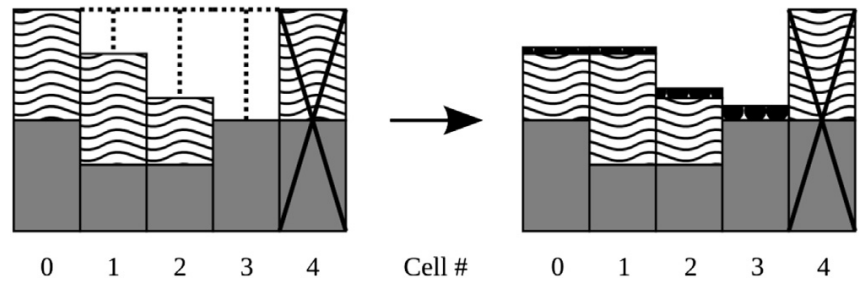
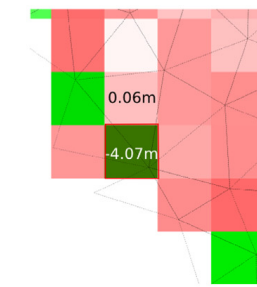
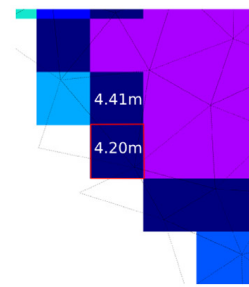
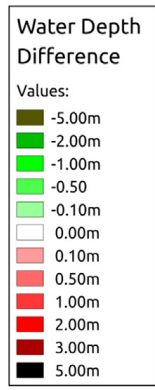
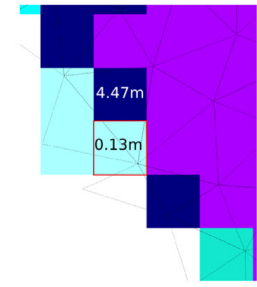
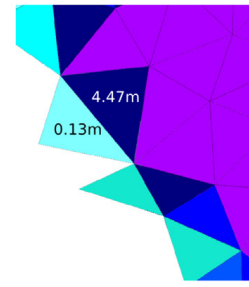
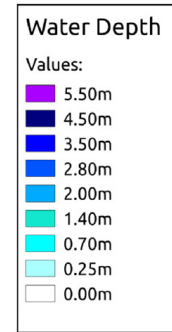
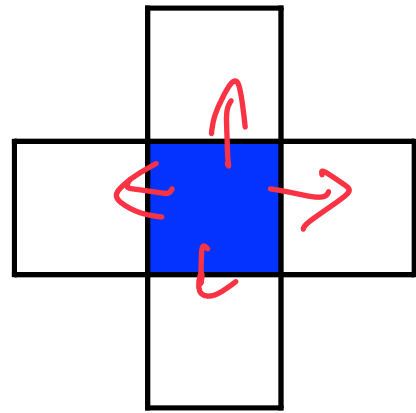
A computational model consisting of a **matrix** and a **set of rules**

Each iteration, the matrix changes based on its current state

There are a number of emergent behaviors that can be discovered!



Flood Analysis Cellular Automata



Formulation of a fast 2D urban pluvial flood model using a cellular automata approach. Ghimire et al 2013
 A weighted cellular automata 2D inundation model for rapid flood analysis. Guidolin et al 2016

The Flood Fill Cellular Automata

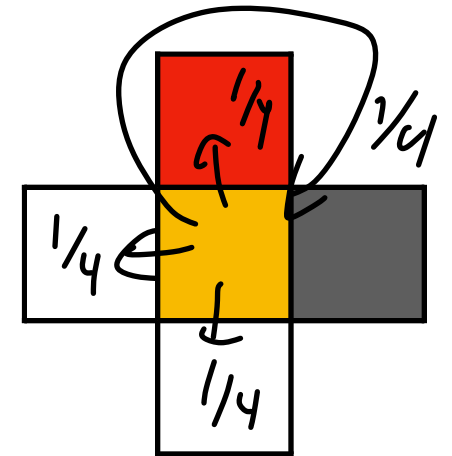
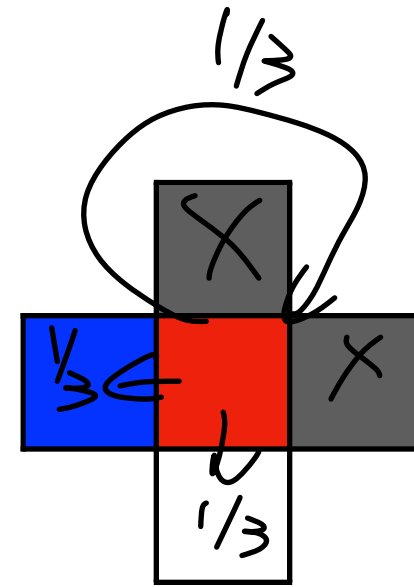
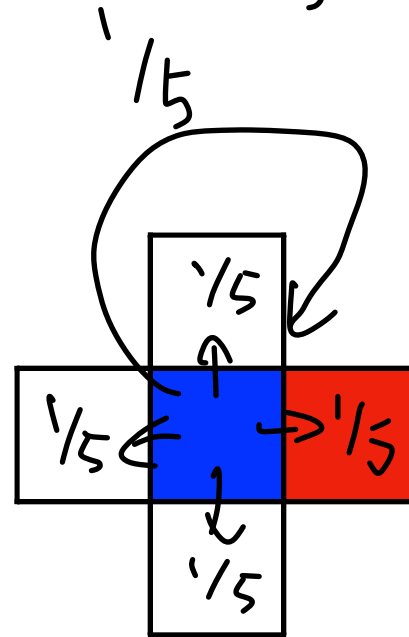
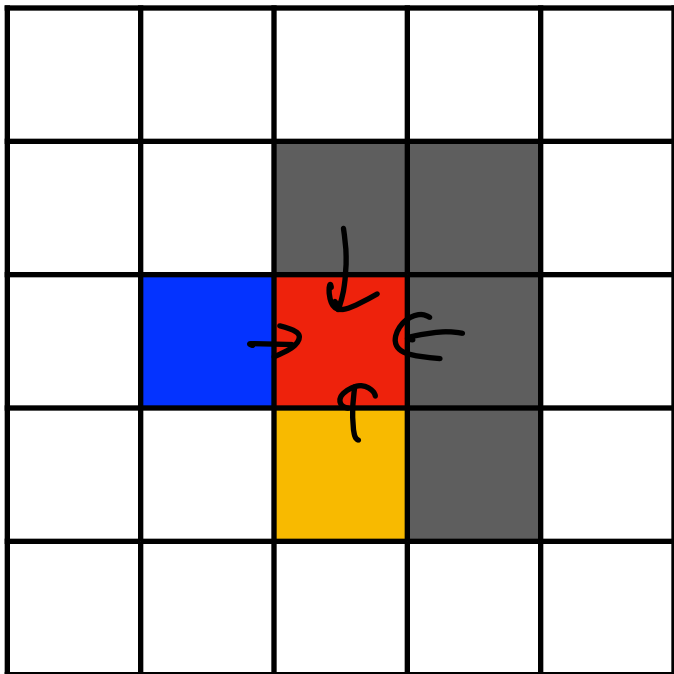
(+) #
-1

We will use a ***much*** simpler model! We have water and impassible barriers.

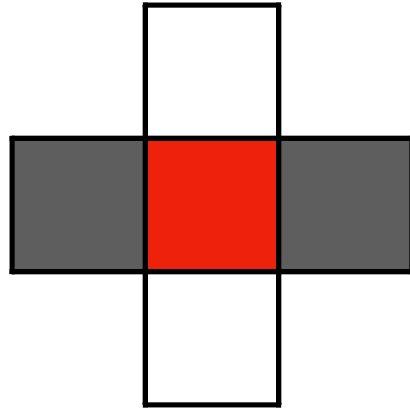
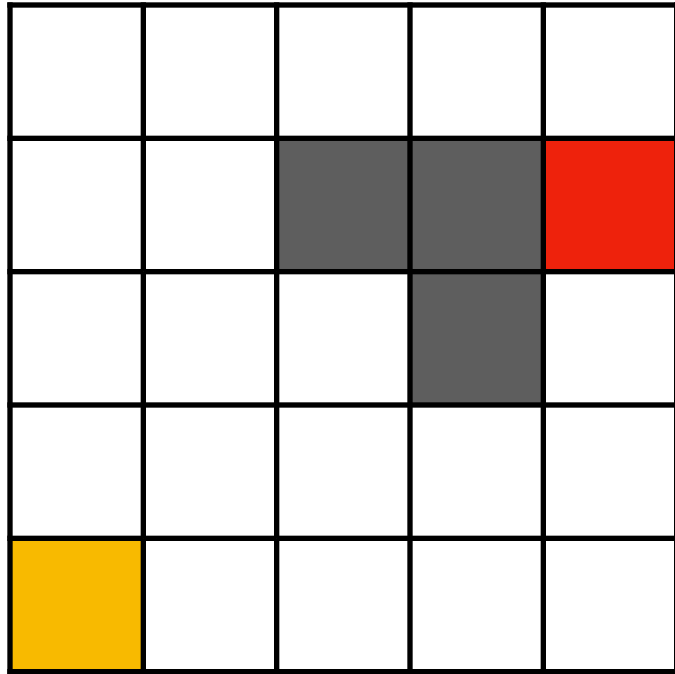
Each step each square splits its water evenly between all nearby cells

The key CA trick: Each square calculates simultaneously.

$$Red_{t=1} = \frac{1}{5} \cdot Blue_{t=0} + \frac{1}{3} Red_{t=0} + \frac{1}{4} Orange_{t=0}$$

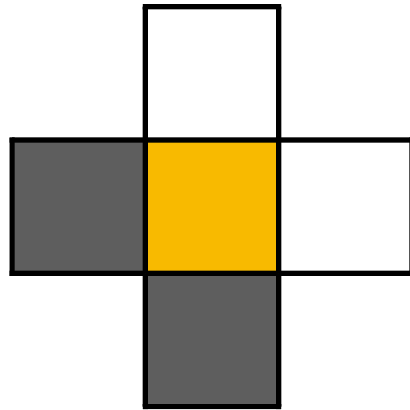


The Flood Fill Cellular Automata



Total cells:

Weight change:



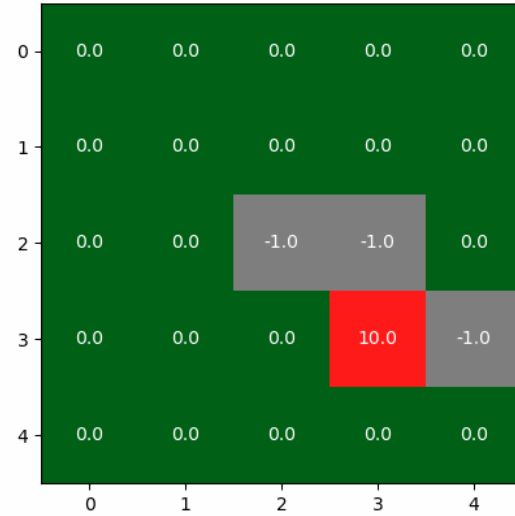
Total cells:

Weight change:

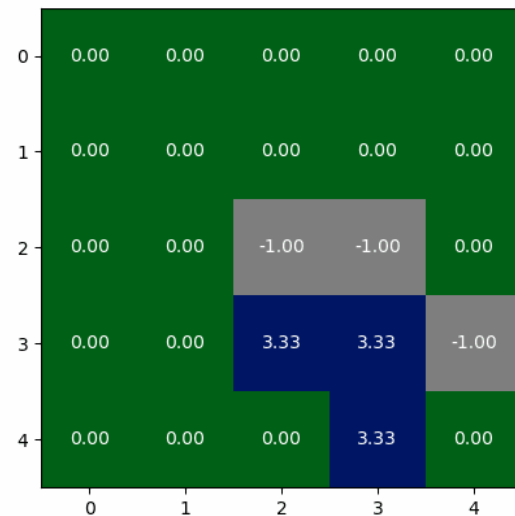
The Flood Fill Cellular Automata



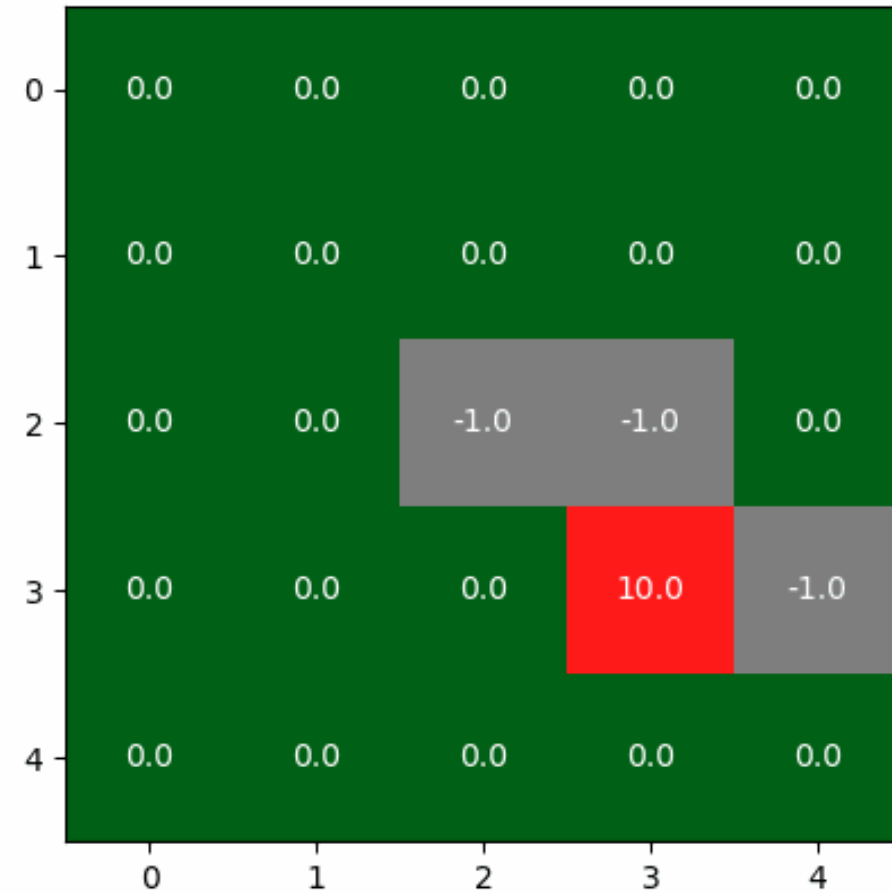
Frame: 0



Frame: 1



Frame: 0



Conway's Game of Life Rules

Developed by John Conway in 1970

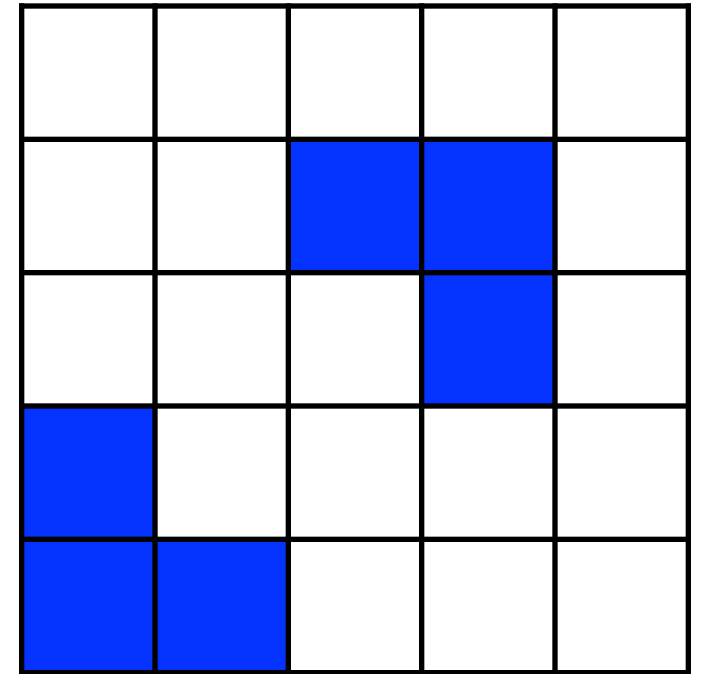
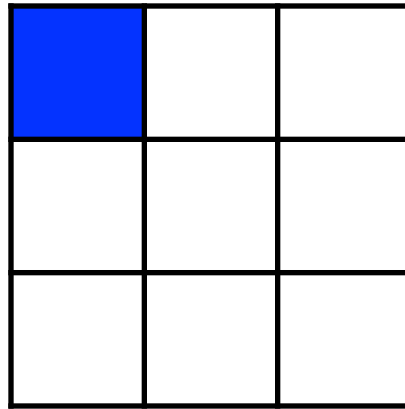
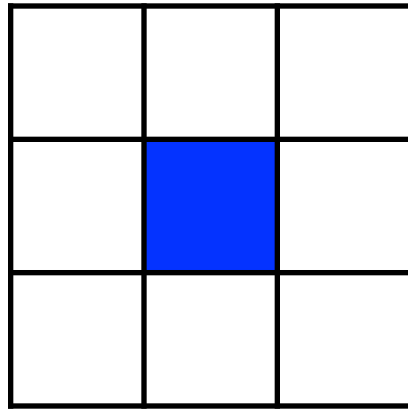
A mostly academic Turing Complete simulation

The ruleset looks at more squares but has easier rules for the final values

A 'simple' but very interesting computational model

Conway's Game of Life Rules *→ testing (conditionally)*

A 'cell' is either alive or dead and has at most 8 neighbors around it



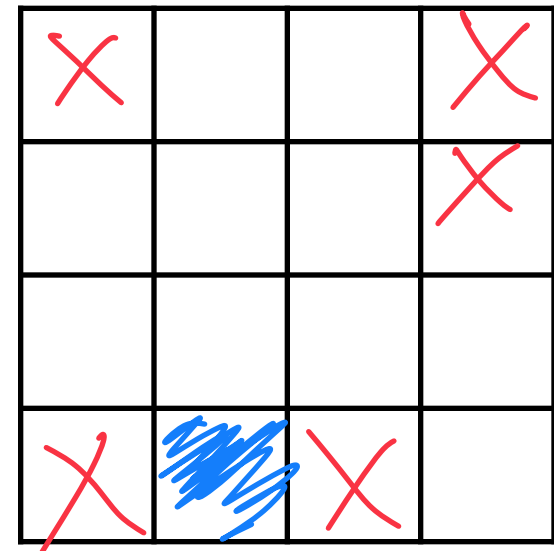
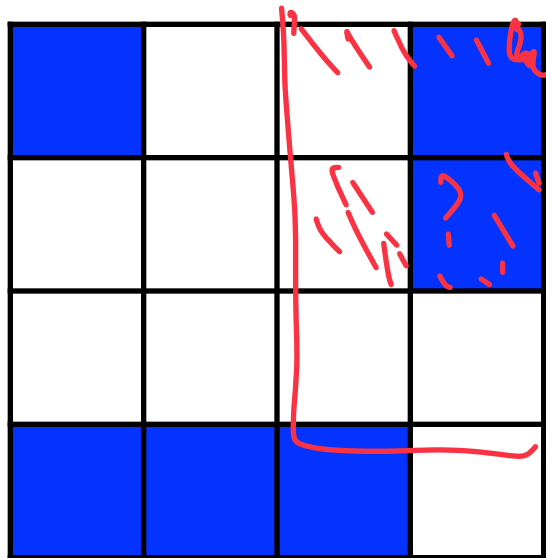
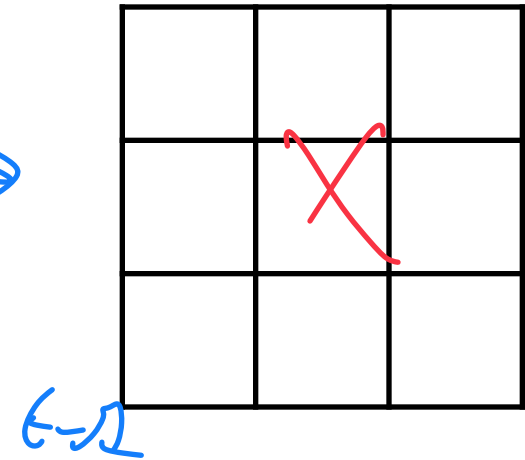
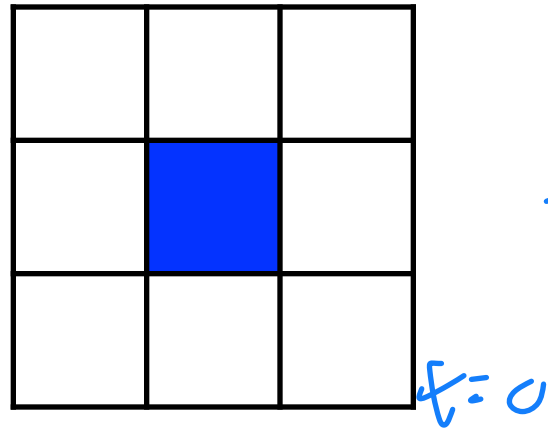
Conway's Game of Life Rules

All cells in a matrix update at the same time according to the following:

1. Any live cell with fewer than two live neighbors dies.
2. Any live cell with two or three live neighbors lives.
3. Any live cell with more than three live neighbors dies.
4. Any dead cells with exactly three live neighbors becomes a live cell.

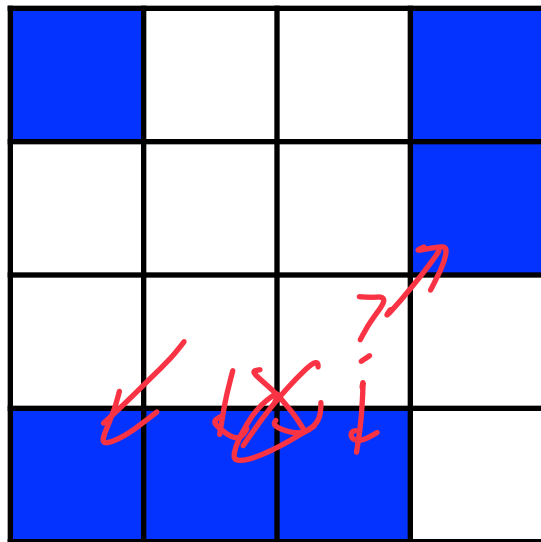
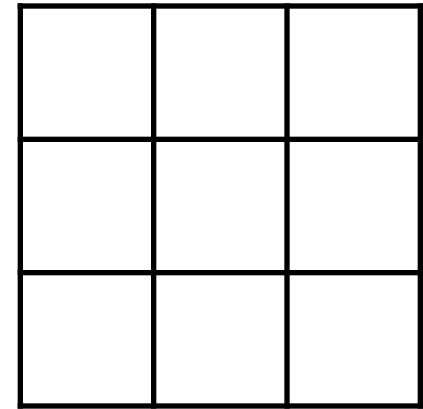
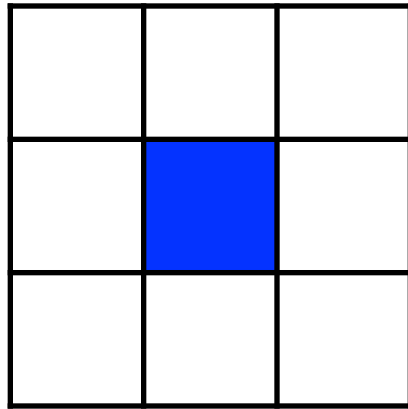
Conway's Game of Life Rules

1. Any live cell with fewer than two live neighbors dies.

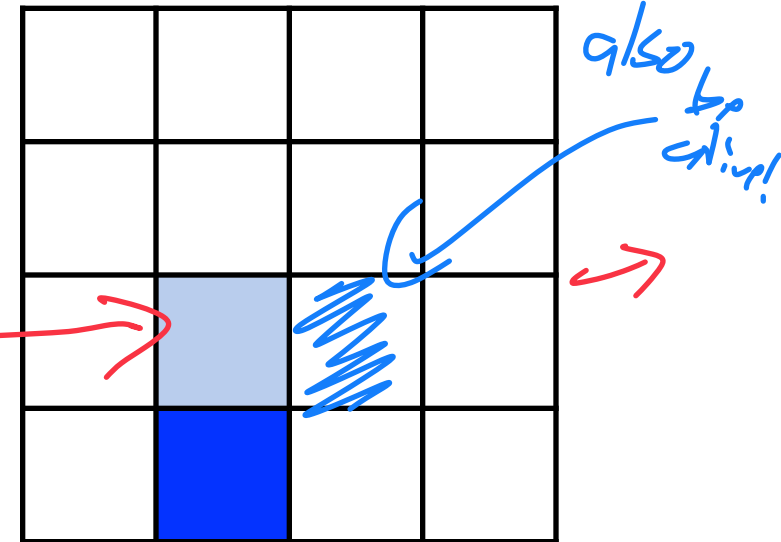


Conway's Game of Life Rules

1. Any live cell with fewer than two live neighbors dies.

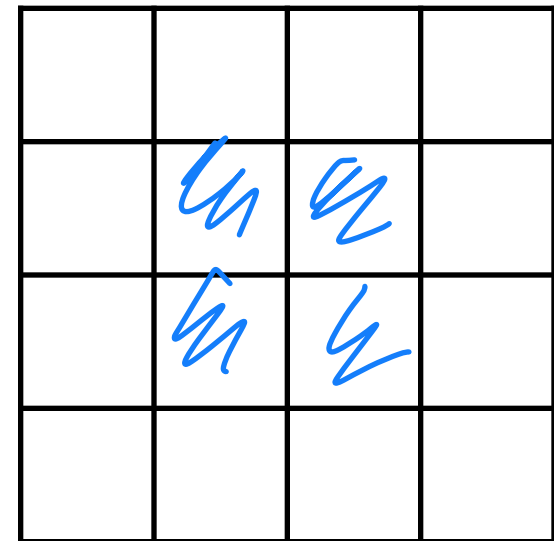
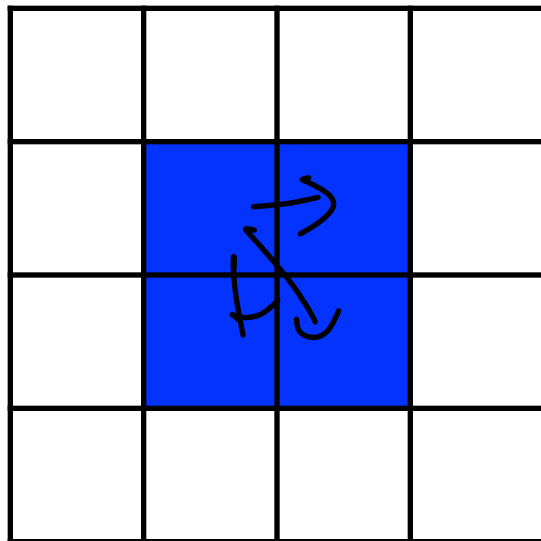
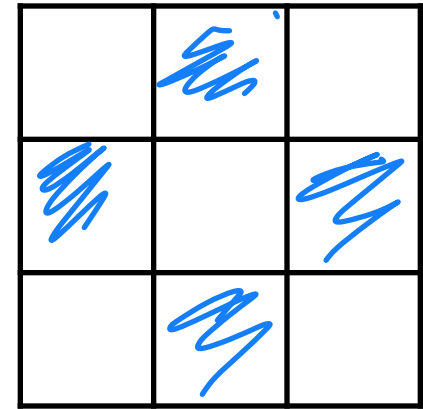
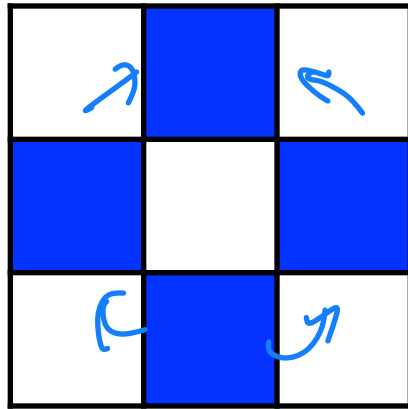


Any cell w/
3 neighbors alive
is alive



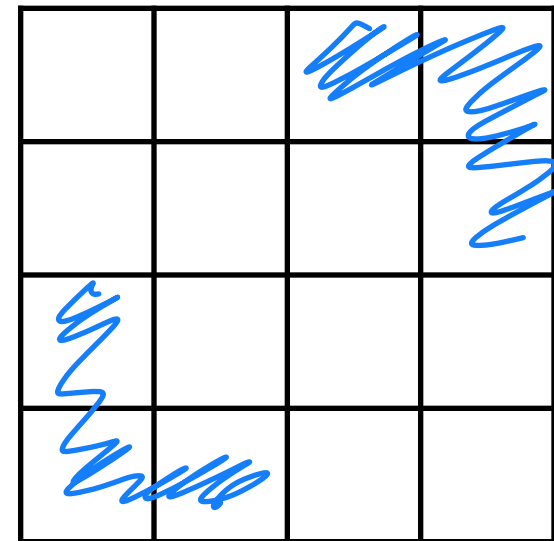
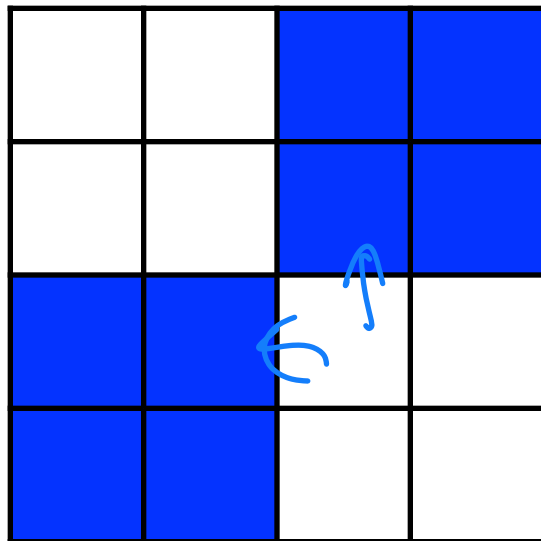
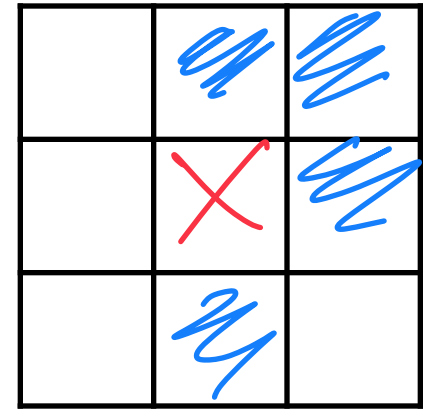
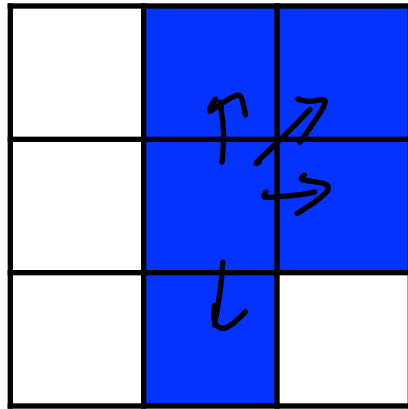
Conway's Game of Life Rules

2. Any live cell with two or three live neighbors lives.



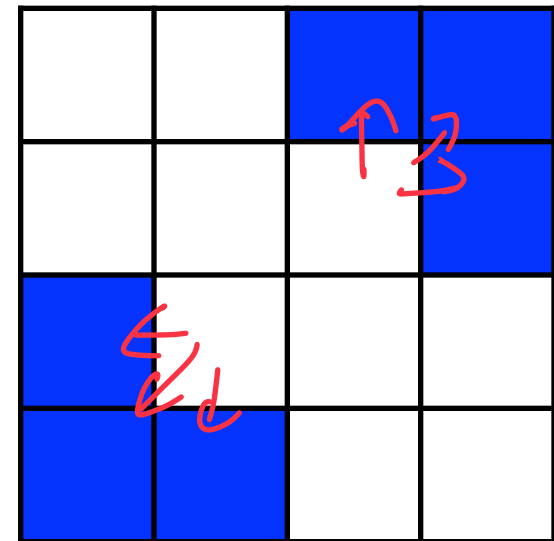
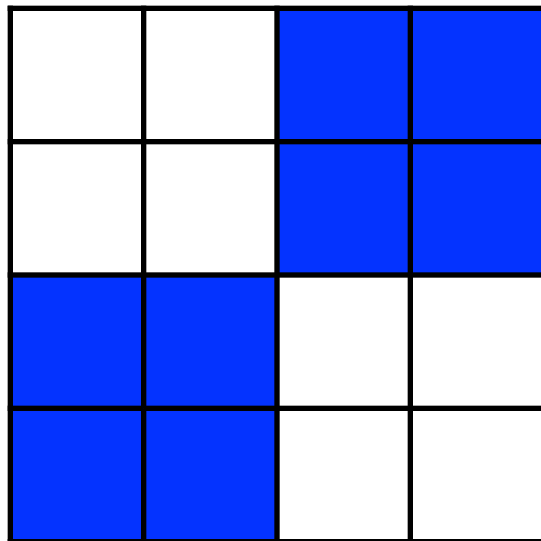
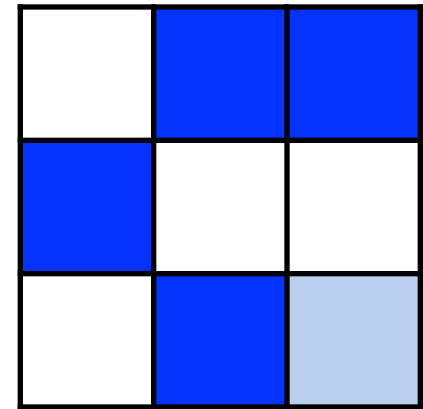
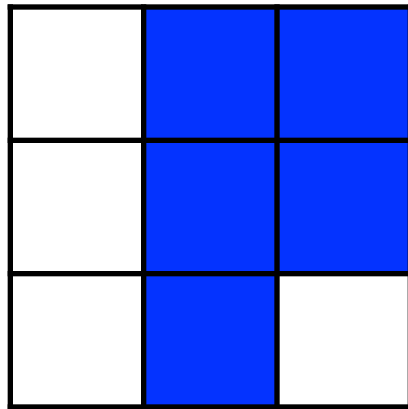
Conway's Game of Life Rules

3. Any live cell with more than three live neighbors dies



Conway's Game of Life Rules

4. Any dead cells with exactly three live neighbors becomes a live cell.

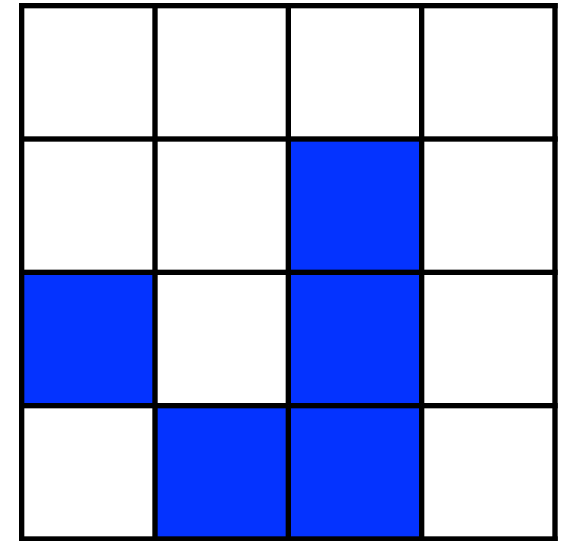
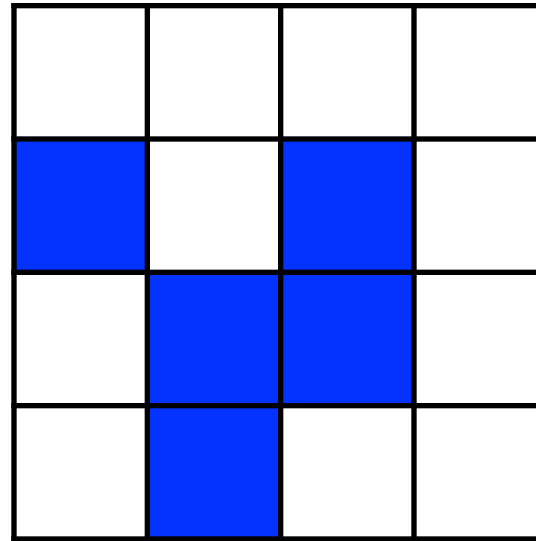
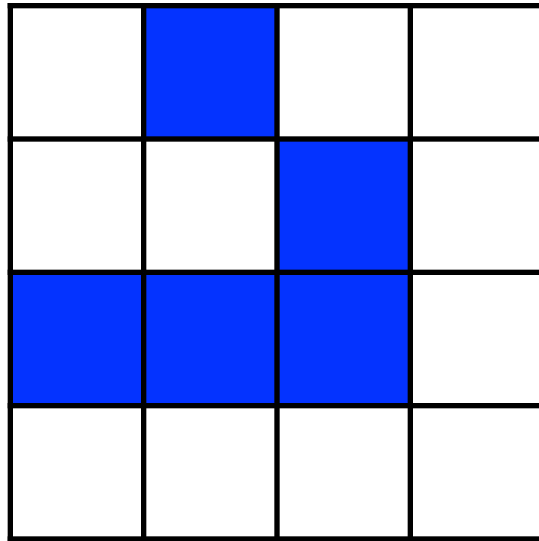


Conway's Game of Life



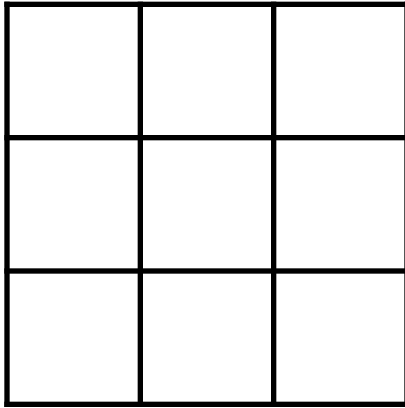
A fun demo version of the game: <https://playgameoflife.com/>

Note: Every cell is updated **at the same time**



For next time: Modifying 2D lists safely

We want to simultaneously update every square in a matrix...



The easiest way to do this is to make a copy of the matrix.

