

Data Structures and Algorithms in Python

www.appmillers.com

Elshad Karimov



@karimov_elshad



in/elshad-karimov

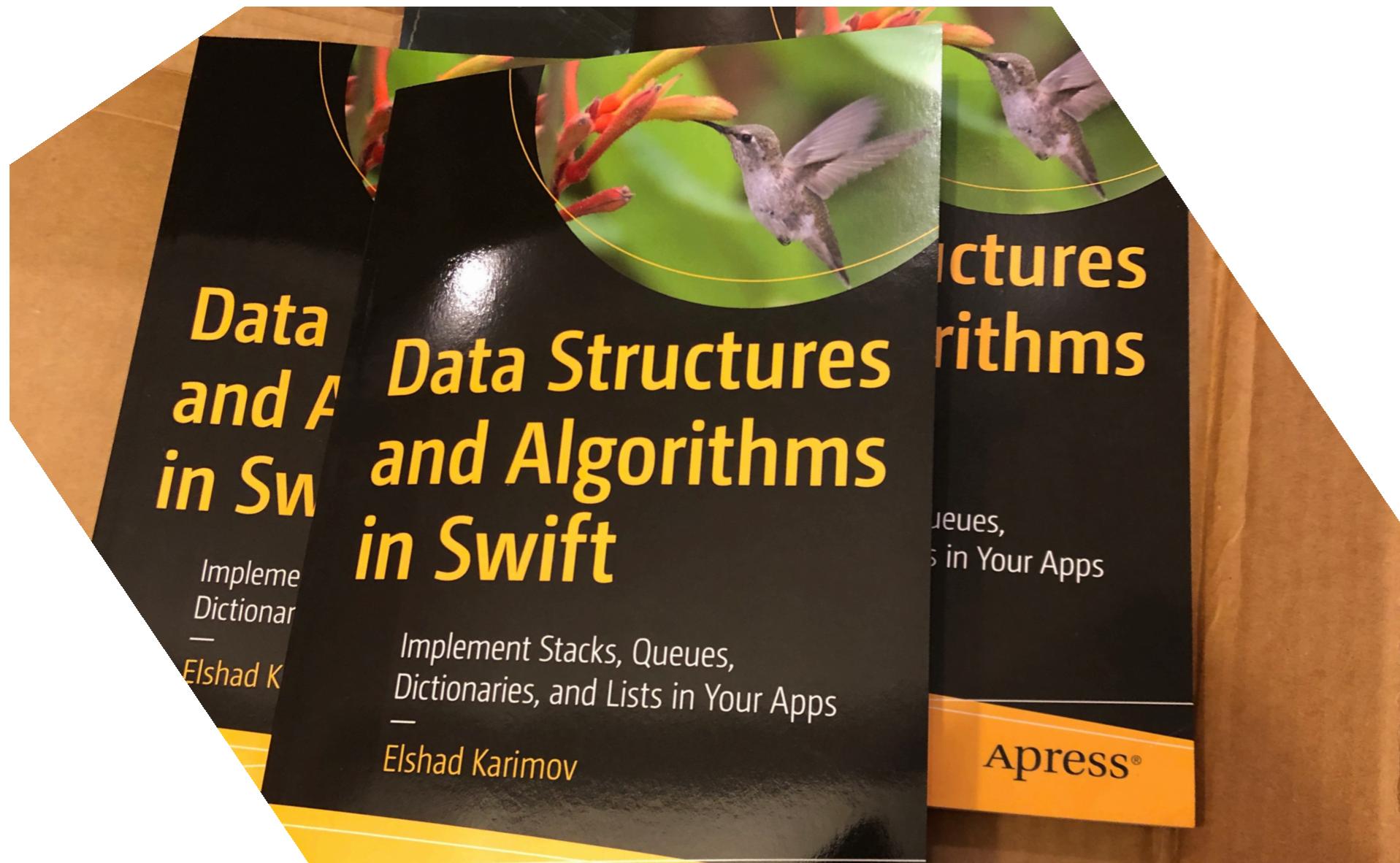


PYTHON For EVERYONE

- PART 1 - Python Basics
- PART 2 - Builtin Data Structures
- PART 3 - Intermediate Python
- PART 4 - Automate Daily Routine Tasks
- PART 5 - Graphical User Interface
- PART 6 - Working with Databases and APIs
- PART 7 - Advanced Python
- PART 8 - Data Analyses and Visualization
- PART 9 - Building Your Portfolio



Complete Python Bootcamp for Everyone From Zero to Hero



Elshad Karimov



@karimov_elshad

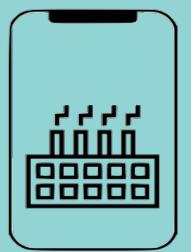
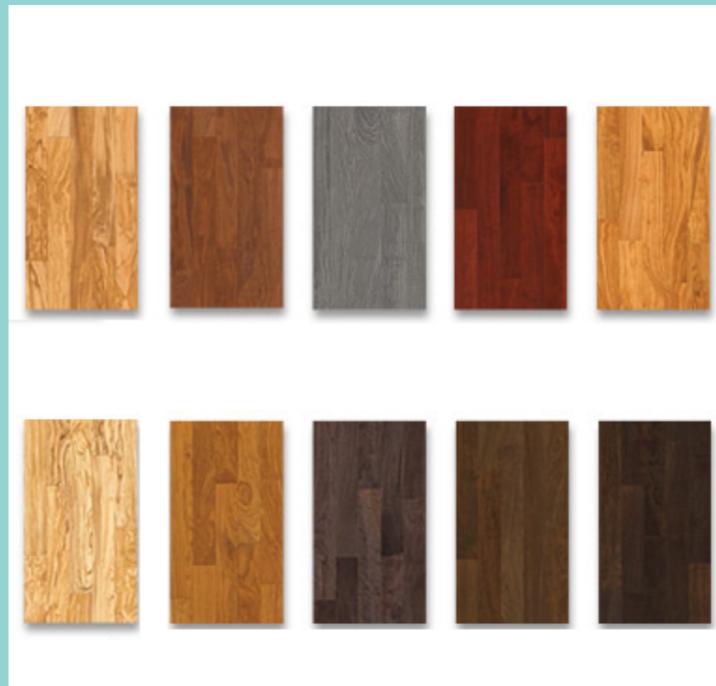


in/elshad-karimov



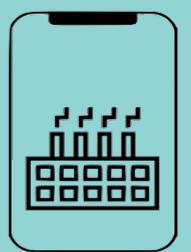
What are Data Structures?

- **Data Structures** are different ways of organizing data on your computer, that can be used effectively.



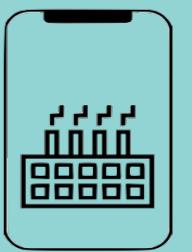
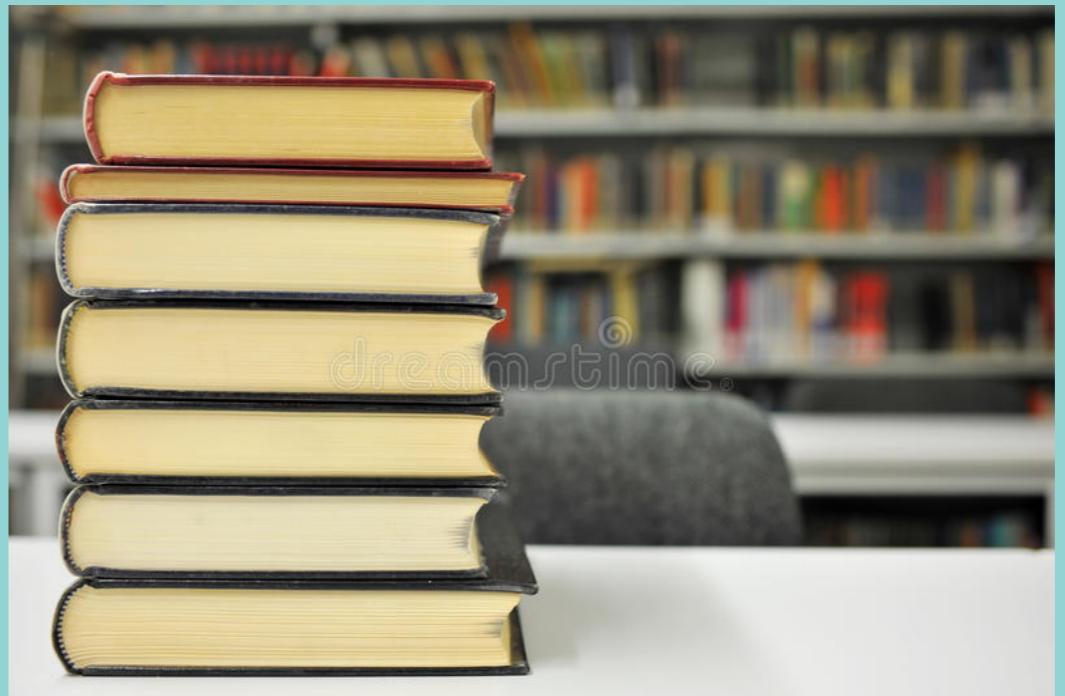
What are Data Structures?

- **Data Structures are different ways of organizing data on your computer, that can be used effectively.**



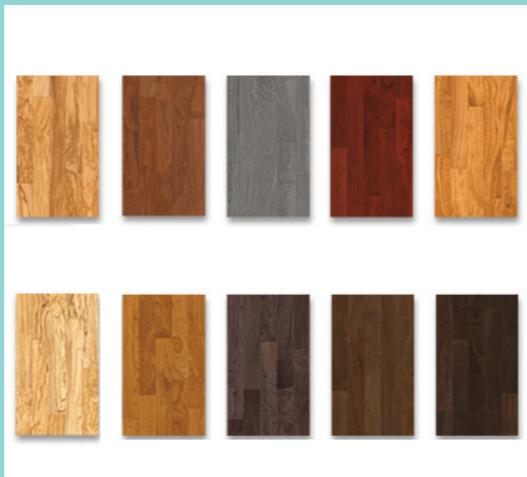
What are Data Structures?

- Data Structures are different ways of organizing data on your computer, that can be used effectively.



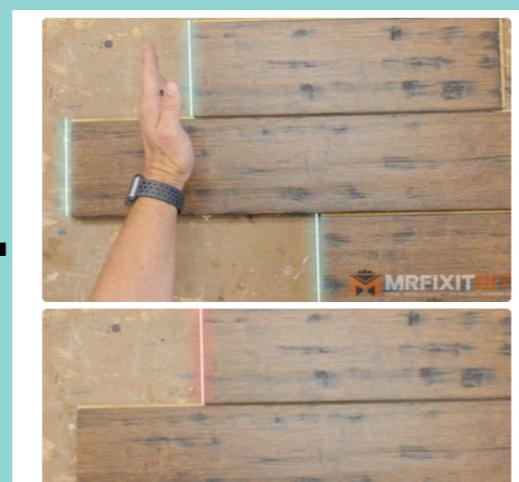
What is an Algorithm?

- Set of steps to accomplish a task



Step 1: Choosing flooring

Step 2: Purchase and bring

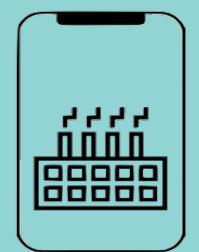
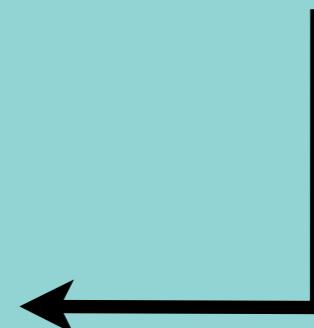


Step 5: Trim door casing

Step 4: Determine the layout



Step 3: Prepare sub flooring



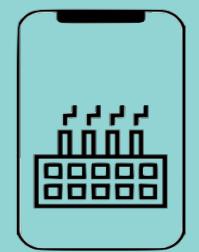
Algorithms in our daily lives



Step 1 : Go to bus stop

Step 2 : Take a bus

Step 3: Go to office



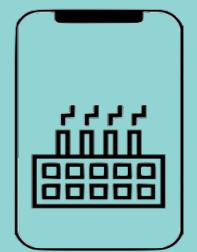
Algorithms in our daily lives



Step 1 : Go to Starbucks

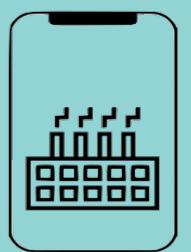
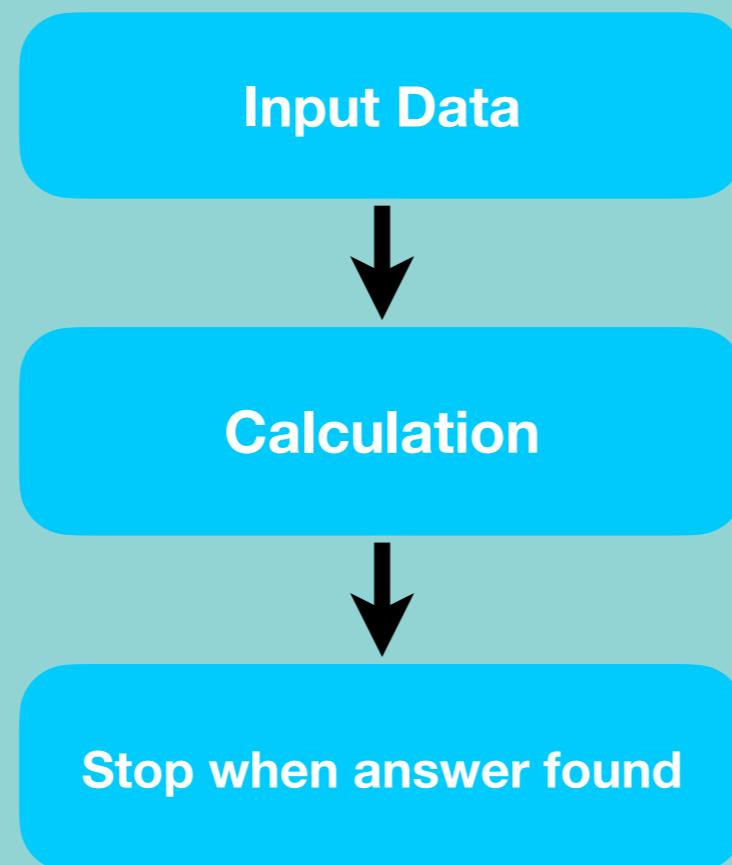
Step 2 : Pay money

Step 3: Take a coffee



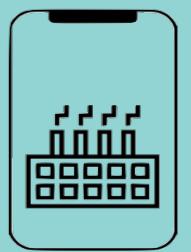
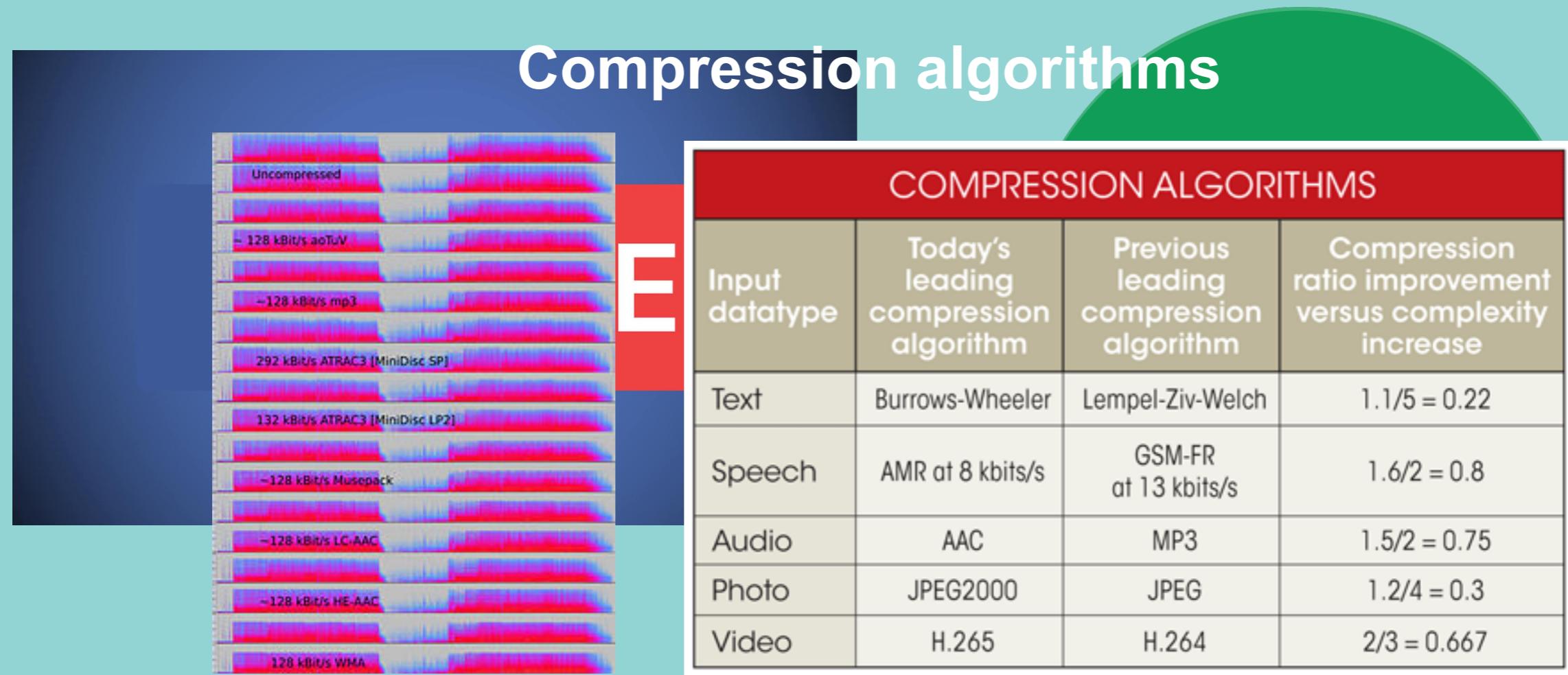
Algorithms in Computer Science

- Set of rules for a computer program to accomplish a task



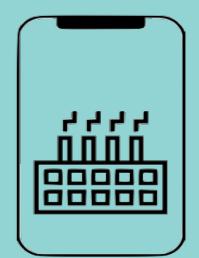
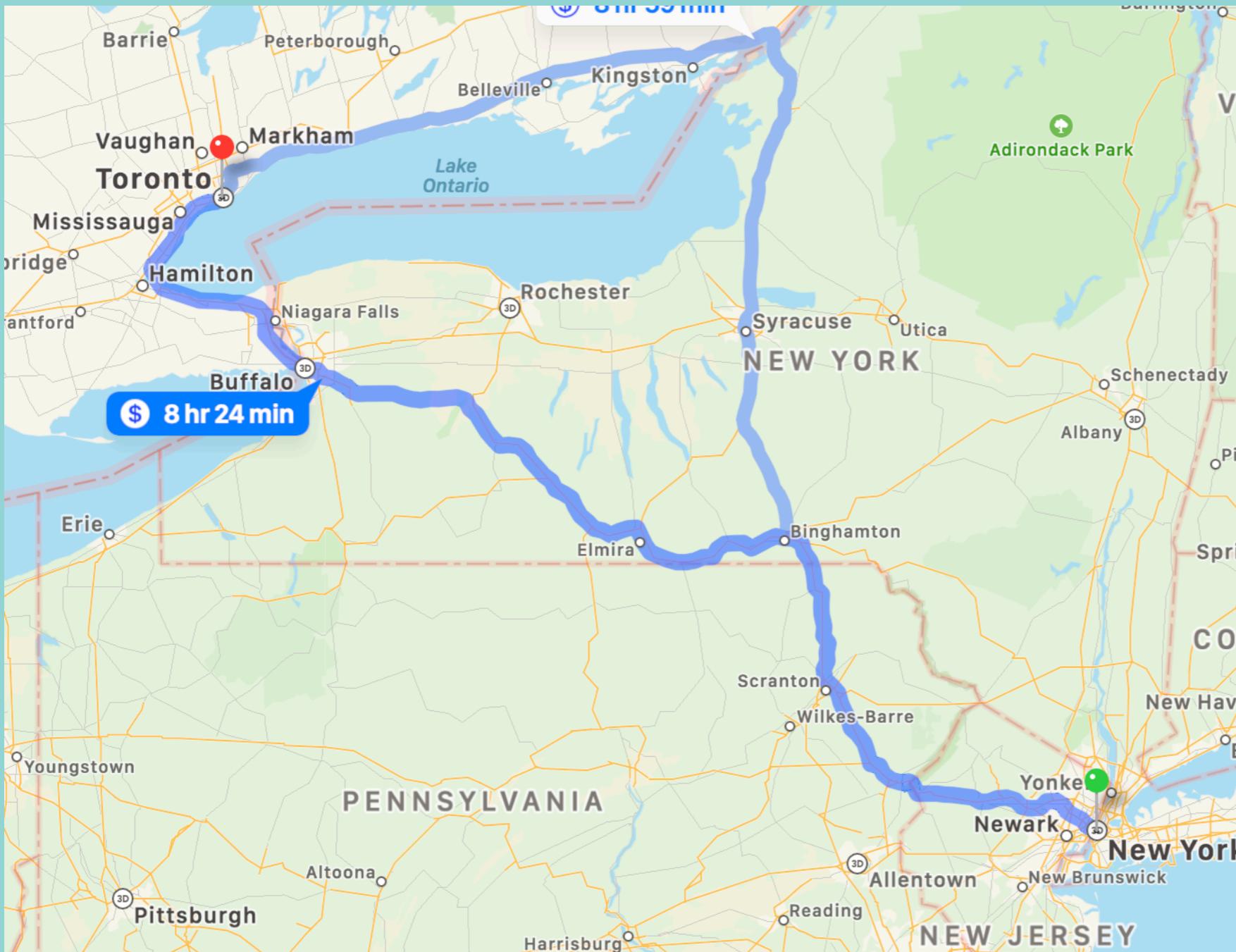
Sample algorithms that are used by big companies

- How do Google and Facebook transmit live video across the internet?



Sample algorithms that are used by big companies

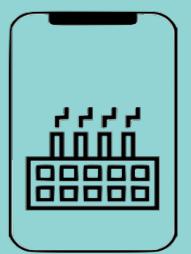
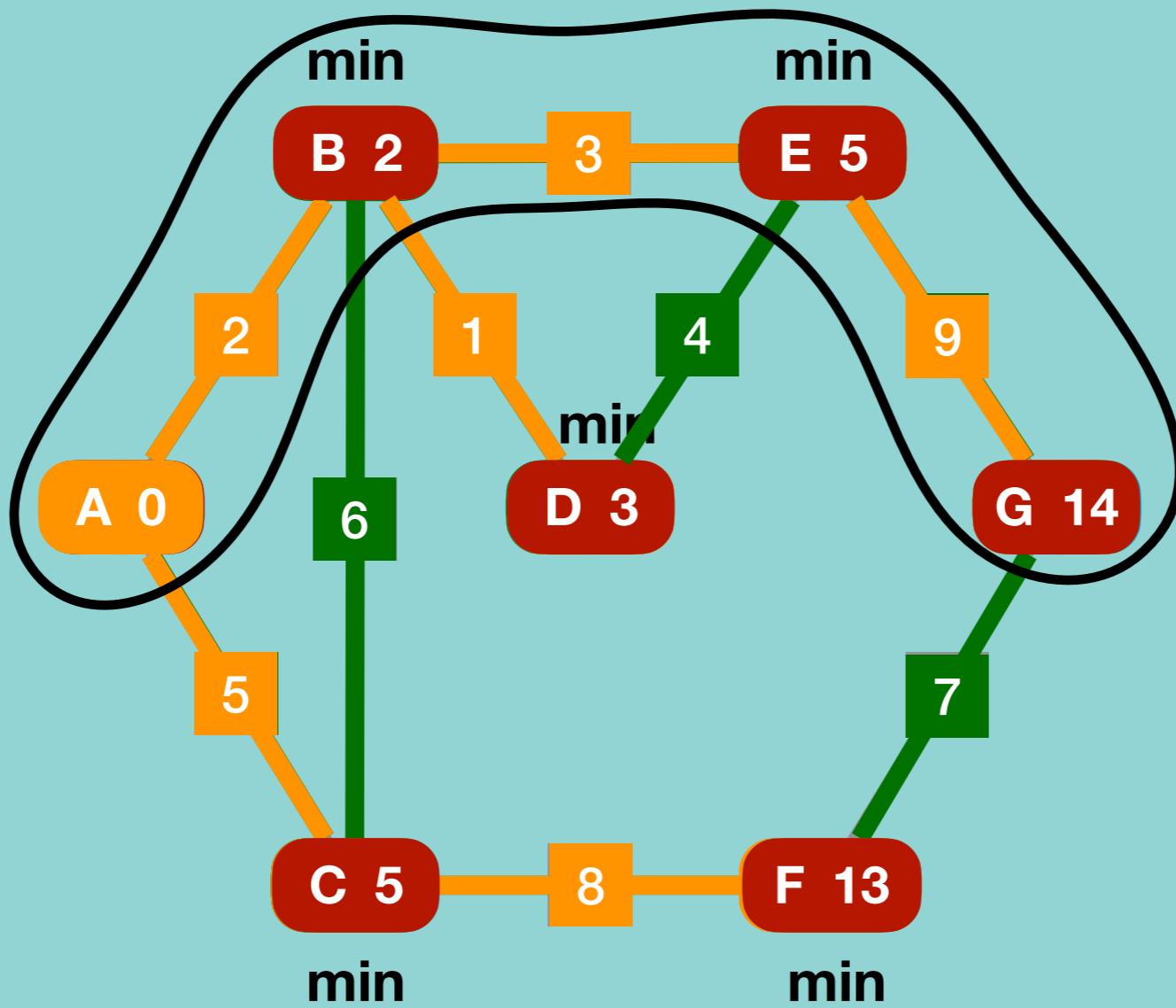
- How to find shortest path on the map?



Sample algorithms that are used by big companies

- How to find shortest path on the map?

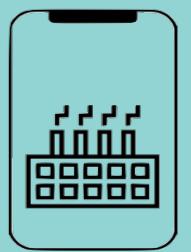
Dijkstra's algorithm



Sample algorithms that are used by big companies

- How to arrange solar panels on the International Space Station?

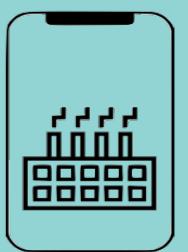
algorithms



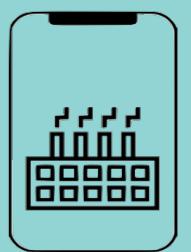
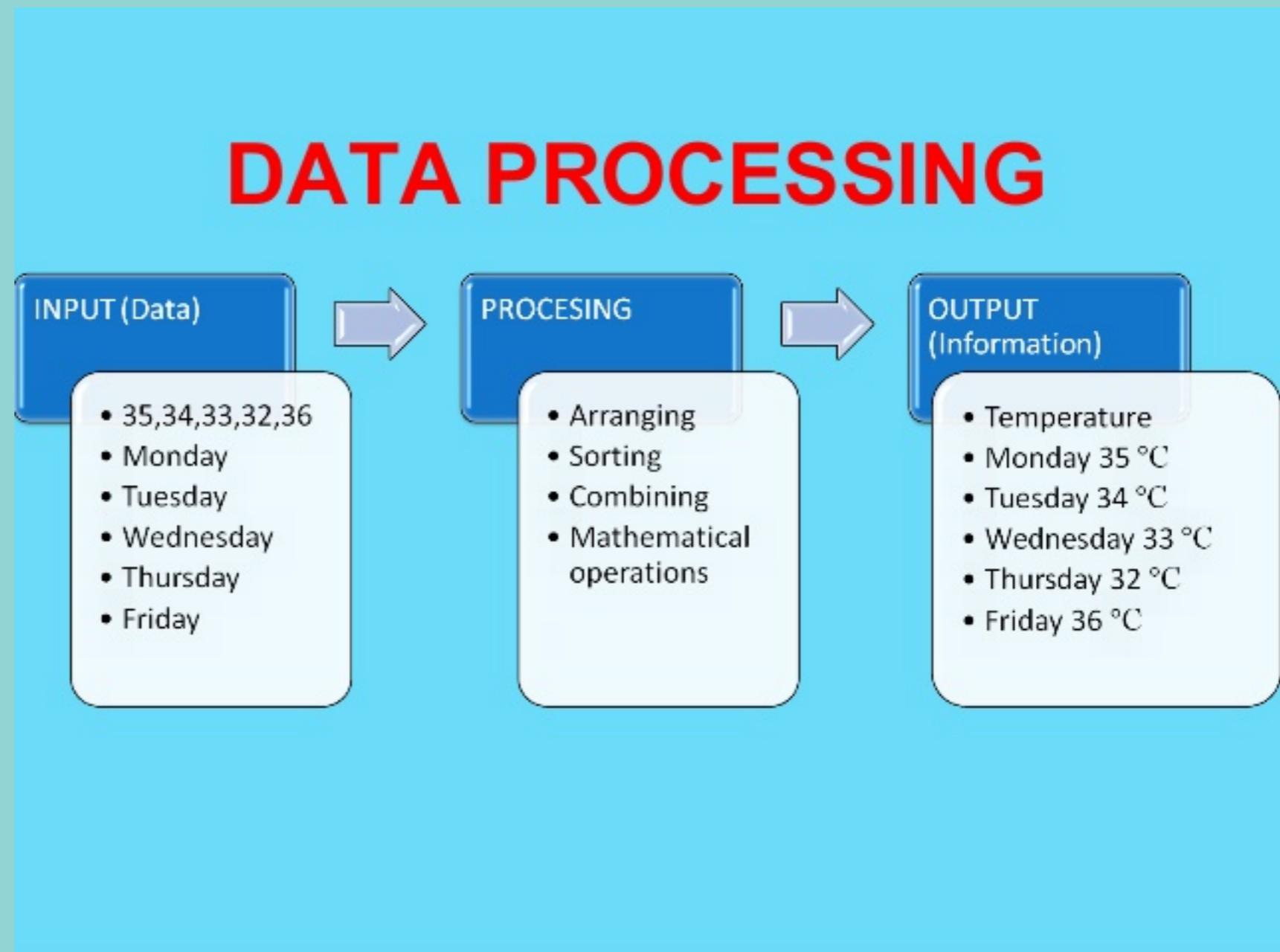
What makes a good algorithm?

1. Correctness

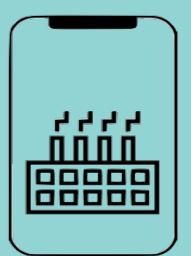
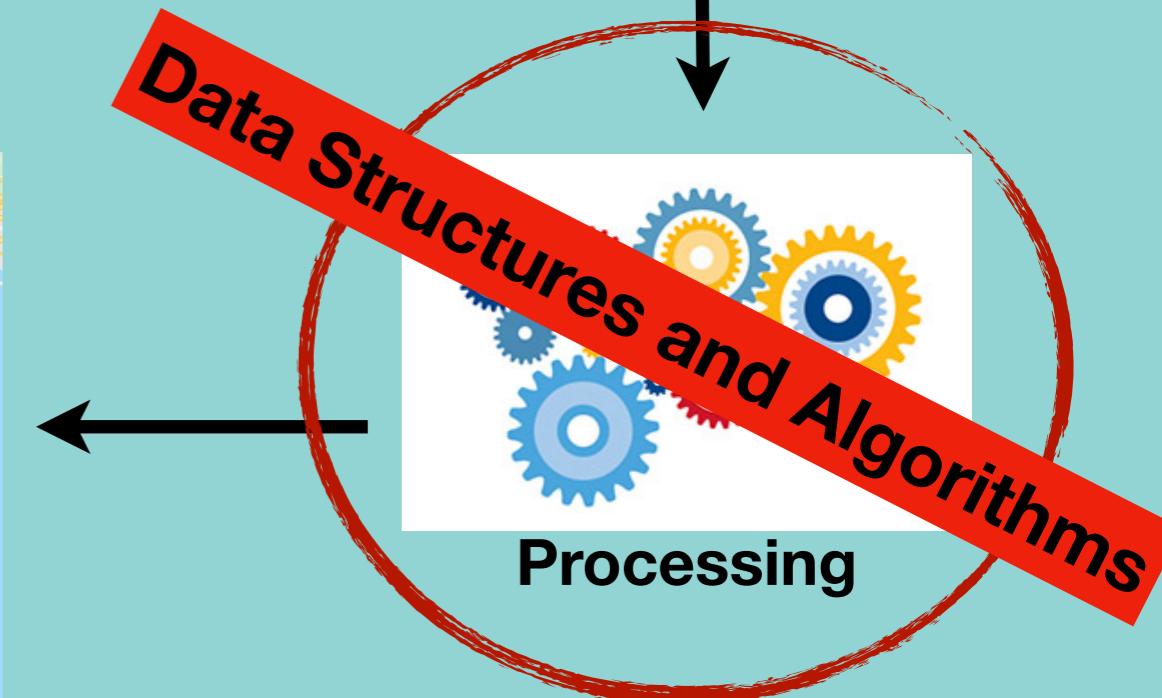
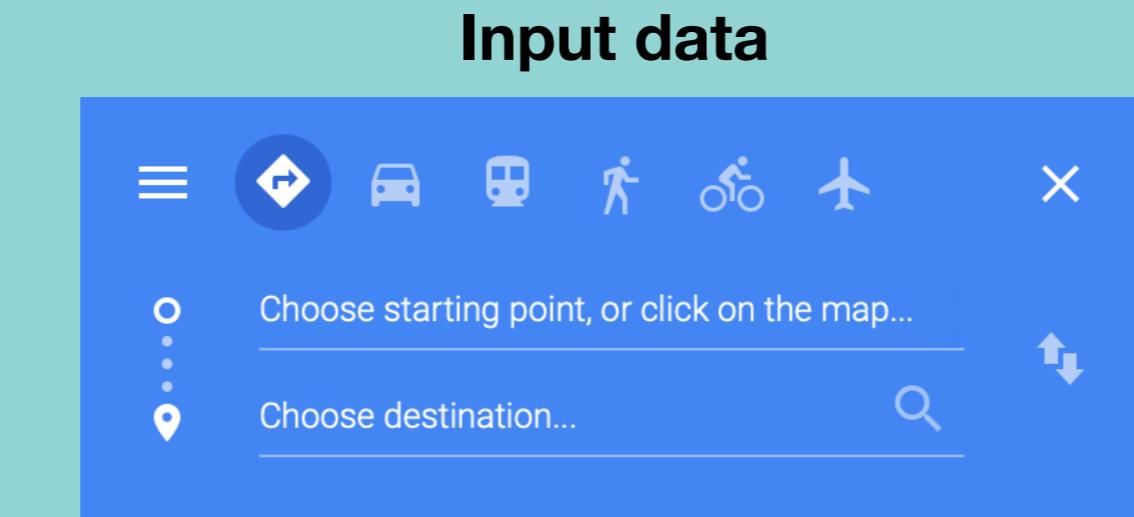
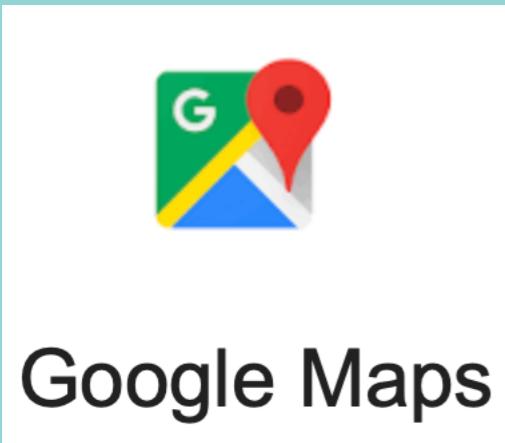
2. Efficiency



Why are Data Structures and Algorithms important?



Why are Data Structures and Algorithms important?

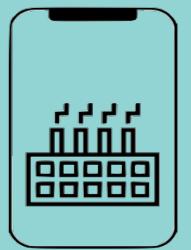
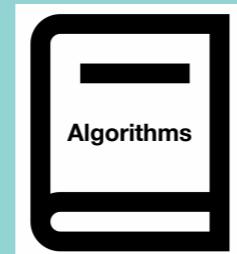


Why are Data Structures and Algorithms important?

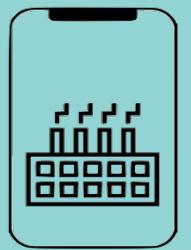


COMPUTER SCIENCE

ALGORITHMS



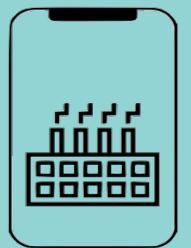
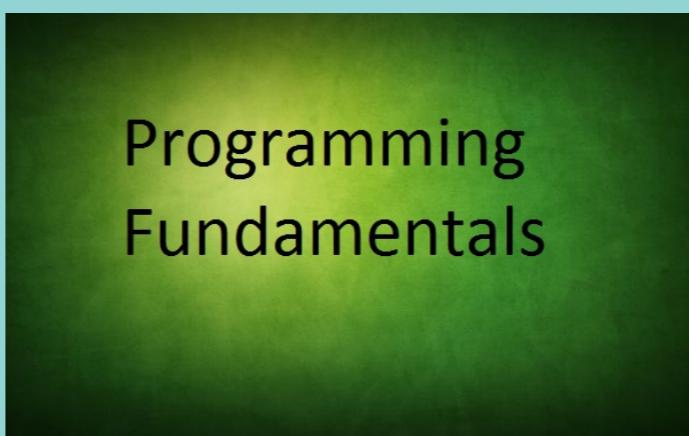
Why are Data Structures and Algorithms important?



Why are Data Structures and Algorithms in INTERVIEWS?

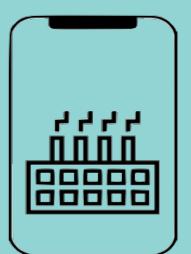
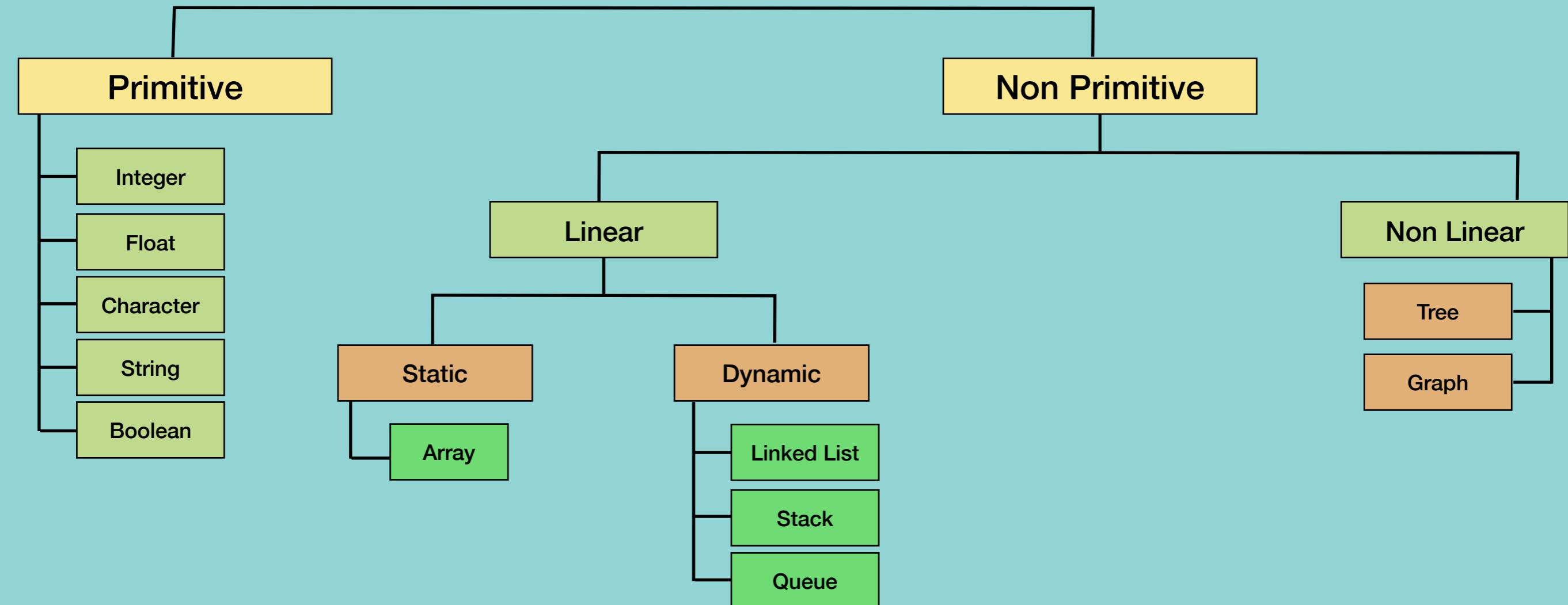


- **Problem solving skills**
- **Fundamental concepts of programming in limited time**



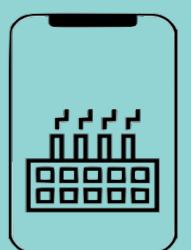
Types of Data Structures

Data Structures



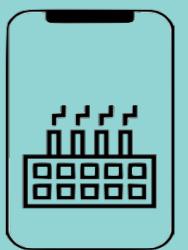
Primitive Data Structures

DATA STRUCTURE	Description	Example
INTEGER	Numbers without decimal point	1, 2, 3, 4, 5, 1000
FLOAT	Numbers with decimal point	3.5, 6.7, 6.987, 20.2
CHARACTER	Single Character	A, B, C, F
STRING	Text	Hello, Data Structure
BOOLEAN	Logical values true or false	TRUE, FALSE



Types of Algorithms

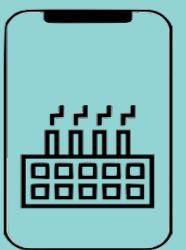
- **Simple recursive algorithms**
- **Divide and conquer algorithms**
- **Dynamic programming algorithms**
- **Greedy algorithms**
- **Brute force algorithms**
- **Randomized algorithms**



Types of Algorithms

Simple recursive algorithms

```
Algorithm Sum(A, n)
    if n=1
        return A[0]
    s = Sum(A, n-1) /* recurse on all but last */
    s = s + A[n-1] /* add last element */
return s
```

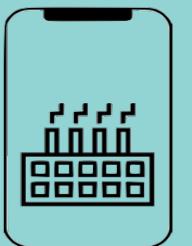


Types of Algorithms

Divide and conquer algorithms

- Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
- Combine the solutions to the subproblems into a solution to the original problem

Examples: Quick sort and merge sort



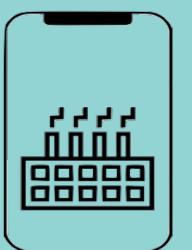
Types of Algorithms

Dynamic programming algorithms

- They work based on memoization
- To find the best solution

Greedy algorithms

- We take the best we can without worrying about future consequences.
- We hope that by choosing a local optimum solution at each step, we will end up at a global optimum solution



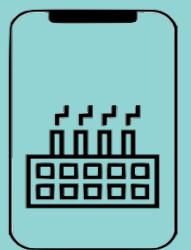
Types of Algorithms

Brute force algorithms

- It simply tries all possibilities until a satisfactory solution is found

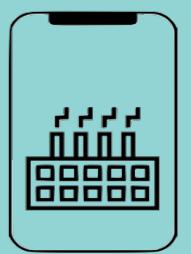
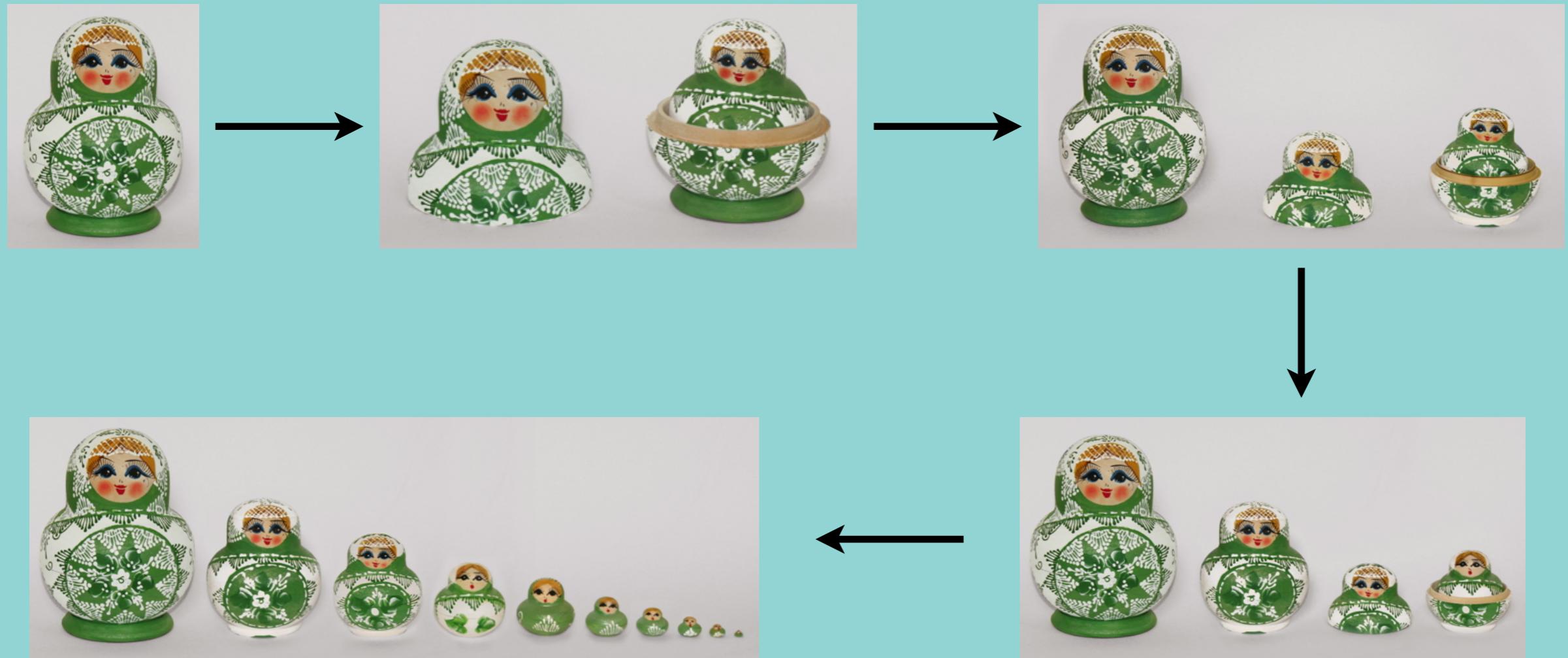
Randomized algorithms

- Use a random number at least once during the computation to make a decision



What is Recursion?

Recursion = a way of solving a problem by having a function calling itself



What is Recursion?

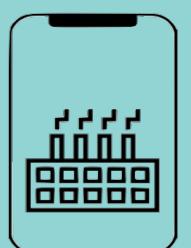
Recursion = a way of solving a problem by having a function calling itself



- Performing the same operation multiple times with different inputs
- In every step we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion, otherwise infinite loop will occur.

A programmer's wife tells him as he leaves the house: "While you're out, buy some milk."

He never returns home and the universe runs out of milk. 🤣🤣🤣

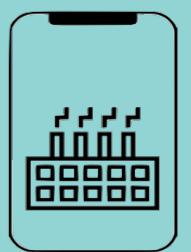


What is Recursion?

Recursion = a way of solving a problem by having a function calling itself

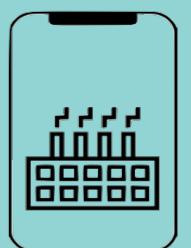


```
def openRussianDoll(doll):
    if doll == 1:
        print("All dolls are opened")
    else:
        openRussianDoll(doll-1)
```



Why Recursion?

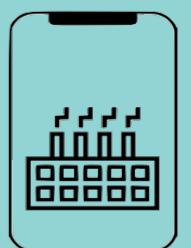
1. Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use
 - when to choose recursion?
 - ▶ If you can divide the problem into similar sub problems
 - ▶ Design an algorithm to compute nth...
 - ▶ Write code to list the n...
 - ▶ Implement a method to compute all.
 - ▶ Practice
2. The prominent usage of recursion in data structures like trees and graphs.
3. Interviews
4. It is used in many algorithms (divide and conquer, greedy and dynamic programming)



How Recursion works?

1. A method calls it self
2. Exit from infinite loop

```
def recursionMethod(parameters):  
    if exit from condition satisfied:  
        return some value  
    else:  
        recursionMethod(modified parameters)
```



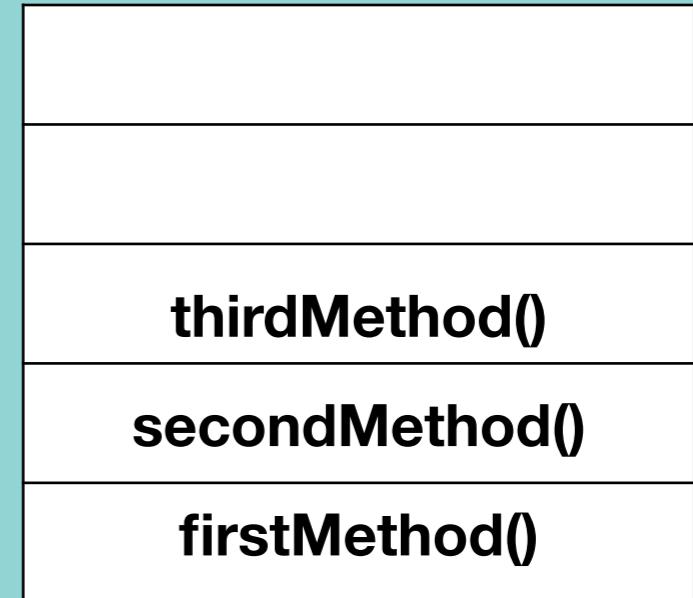
How Recursion works?

```
def firstMethod():
    secondMethod()
    print("I am the first Method")

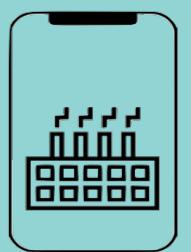
def secondMethod():
    thirdMethod()
    print("I am the second Method")

def thirdMethod():
    fourthMethod()
    print("I am the third Method")

def fourthMethod():
    print("I am the fourth Method")
```



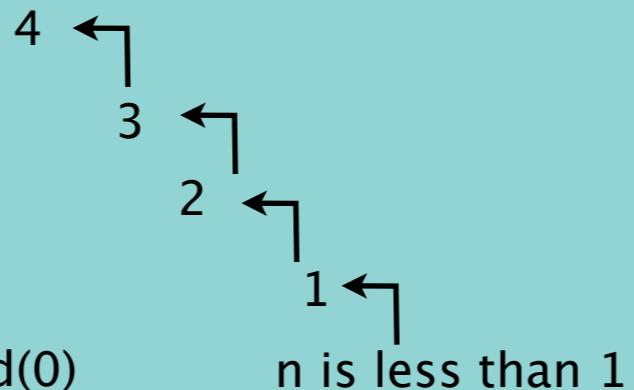
STACK Memory



How Recursion works?

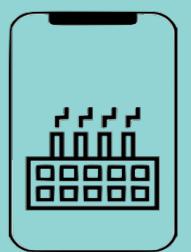
```
def recursiveMethod(n):
    if n<1:
        print("n is less than 1")
    else:
        recursiveMethod(n-1)
        print(n)
```

recursiveMethod(4)
 ↳ recursiveMethod(3)
 ↳ recursiveMethod(2)
 ↳ recursiveMethod(1)
 ↳ recursiveMethod(0)



recursiveMethod(1)
recursiveMethod(2)
recursiveMethod(3)
recursiveMethod(4)

STACK Memory

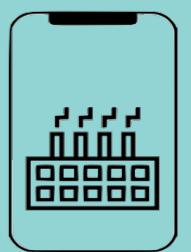


Recursive vs Iterative Solutions

```
def powerOfTwo(n):
    if n == 0:
        return 1
    else:
        power = powerOfTwo(n-1)
        return power * 2
```

```
def powerOfTwoIt(n):
    i = 0
    power = 1
    while i < n:
        power = power * 2
        i = i + 1
    return power
```

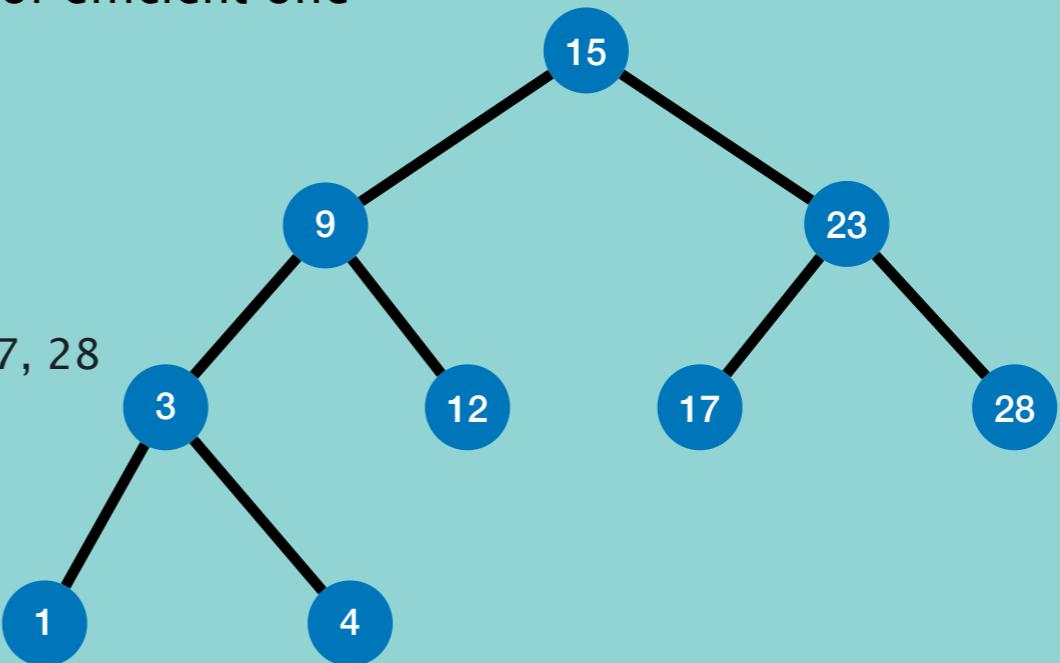
Points	Recursion	Iteration	
Space efficient?	No	Yes	No stack memory require in case of iteration
Time efficient?	No	Yes	In case of recursion system needs more time for pop and push elements to stack memory which makes recursion less time efficient
Easy to code?	Yes	No	We use recursion especially in the cases we know that a problem can be divided into similar sub problems.



When to Use/Avoid Recursion?

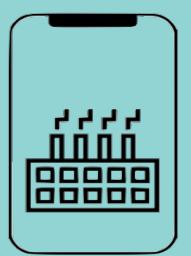
When to use it?

- When we can easily breakdown a problem into similar subproblem
- When we are fine with extra overhead (both time and space) that comes with it
- When we need a quick working solution instead of efficient one
- When traverse a tree
- When we use memoization in recursion
 - preorder tree traversal : 15, 9, 3, 1, 4, 12, 23, 17, 28



When avoid it?

- If time and space complexity matters for us.
- Recursion uses more memory. If we use embedded memory. For example an application that takes more memory in the phone is not efficient
- Recursion can be slow



How to write recursion in 3 steps?

Factorial

- It is the product of all positive integers less than or equal to n.
- Denoted by $n!$ (Christian Kramp in 1808).
- Only positive numbers.
- $0!=1$.

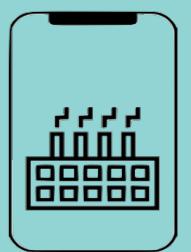
Example 1

$$4! = 4 * 3 * 2 * 1 = 24$$

Example 2

$$10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 36,28,800$$

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$



How to write recursion in 3 steps?

Step 1 : Recursive case – the flow

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1 \longrightarrow n! = n * (n-1)!$$

↓
 $(n-1)!$

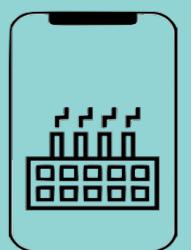
$$(n-1)! = (n-1) * (n-1-1) * (n-1-2) * \dots * 2 * 1 = (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

Step 2 : Base case – the stopping criterion

- $0! = 1$
- $1! = 1$

Step 3 : Unintentional case – the constraint

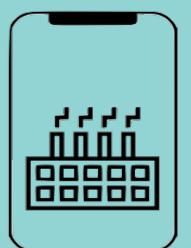
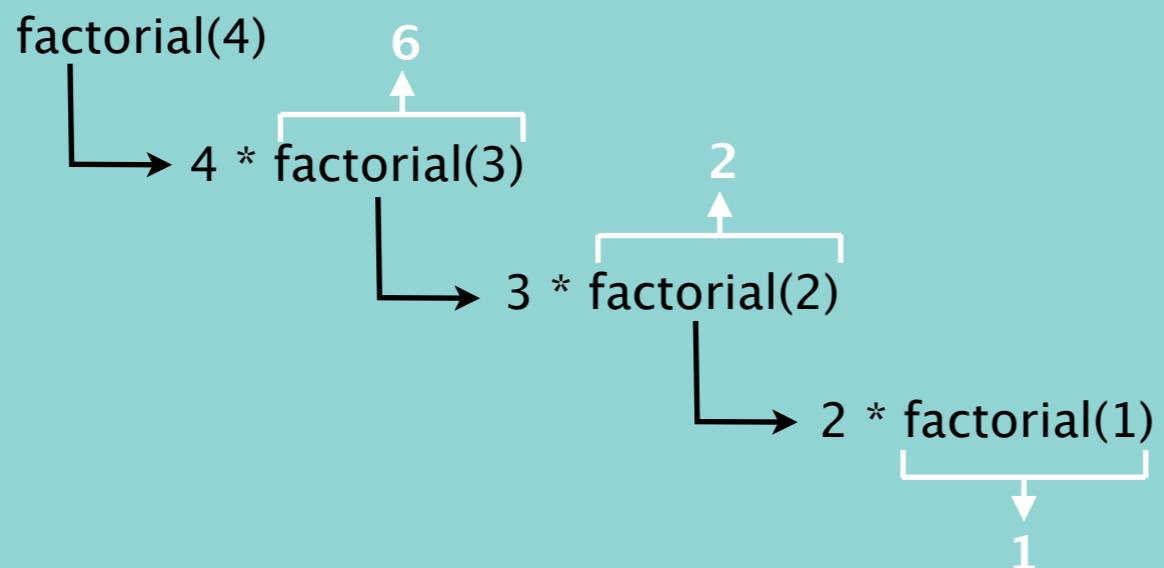
- $\text{factorial}(-1)$??
- $\text{factorial}(1.5)$??



How to write recursion in 3 steps?

```
def factorial(n):
    assert n >= 0 and int(n) == n, 'The number must be positive integer only!'
    if n in [0,1]:
        return 1
    else:
        return n * factorial(n-1)
```

$$\text{factorial}(4) = 24$$



Fibonacci numbers - Recursion

Fibonacci sequence is a sequence of numbers in which each number is the sum of the two preceding ones and the sequence starts from 0 and 1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Step 1 : Recursive case – the flow

$$5 = 3 + 2$$

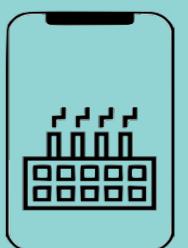
$$f(n) = f(n-1) + f(n-2)$$

Step 2 : Base case – the stopping criterion

- 0 and 1

Step 3 : Unintentional case – the constraint

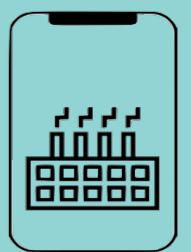
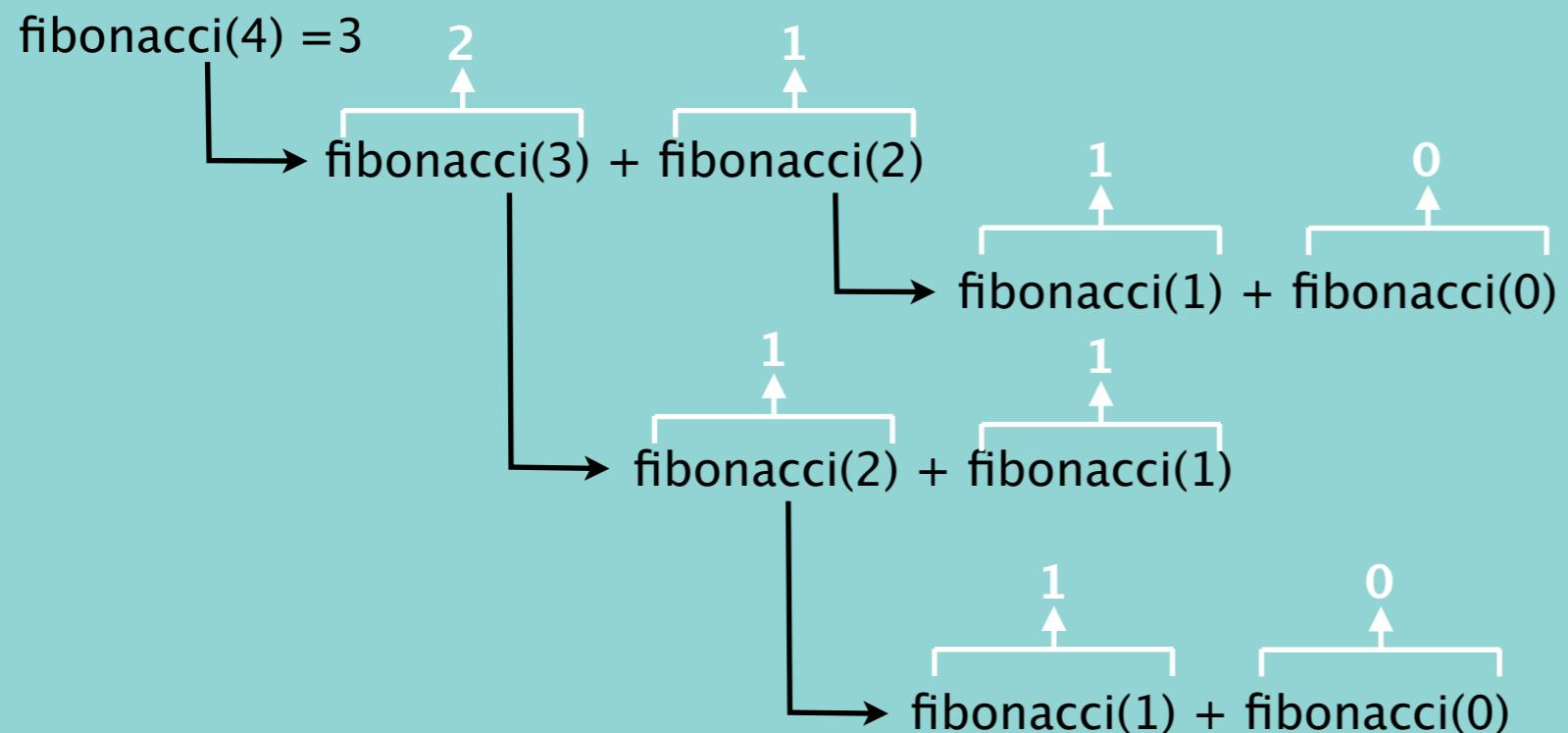
- fibonacci(-1) ??
- fibonacci(1.5) ??



Fibonacci numbers - Recursion

```
def fibonacci(n):
    assert n >=0 and int(n) == n , 'Fibonacci number cannot be negative number or non integer.'
    if n in [0,1]:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...



Interview Questions - 1

How to find the sum of digits of a positive integer number using recursion ?

Step 1 : Recursive case – the flow

10 $10/10 = 1$ and Remainder = 0

$$f(n) = n \% 10 + f(n / 10)$$

54 $54/10 = 5$ and Remainder = 4

112 $112/10 = 11$ and Remainder = 2

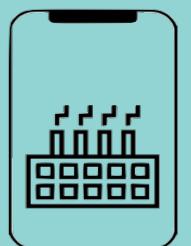
$11/10 = 1$ and Remainder = 1

Step 2 : Base case – the stopping criterion

- $n = 0$

Step 3 : Unintentional case – the constraint

- $\text{sumofDigits}(-11)$??
- $\text{sumofDigits}(1.5)$??



Interview Question 2 - How to calculate power of a number using recursion?

$x^n = x * x * x * \dots (n \text{ times}) \dots * x$

Step 1 : Recursive case - the flow

$$2^4 = 2 * 2 * 2 * 2$$

$$x^a * x^b = x^{a+b}$$

$$x^3 * x^4 = x^{3+4}$$

$$x^n = x * x^{n-1}$$

Step 2 : Base case - the stopping criterion

$$n = 0$$

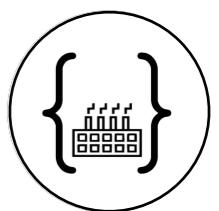
Step 3 : Unintentional case - the constraint

power(-1,2) ??

power(3.2, 2) ??

power(2, 1.2) ??

power(2, -1) ??



Interview Questions - 3

How to find GCD (Greatest Common Divisor) of two numbers using recursion?

Step 1 : Recursive case – the flow

GCD is the largest positive integer that divides the numbers without a remainder

$$\gcd(8,12) = 4$$

$$8 = \cancel{2} * \cancel{2} * 2$$

$$12 = \cancel{2} * \cancel{2} * 3$$

Euclidean algorithm

$$\gcd(48,18)$$

Step 1 : $48/18 = 2$ remainder 12

Step 2 : $18/12 = 1$ remainder 6

Step 3 : $12/6 = 2$ remainder 0

$$\gcd(a, 0)=a$$

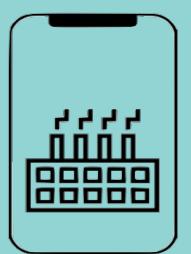
$$\gcd(a, b) = \gcd(b, a \bmod b)$$

Step 2 : Base case – the stopping criterion

- $b = 0$

Step 3 : Unintentional case – the constraint

- Positive integers
- Convert negative numbers to positive



Interview Questions - 4

How to convert a number from Decimal to Binary using recursion

Step 1 : Recursive case – the flow

Step 1 : Divide the number by 2

Step 2 : Get the integer quotient for the next iteration

Step 3 : Get the remainder for the binary digit

Step 3 : Repeat the steps until the quotient is equal to 0

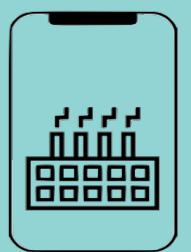
10 to binary

1000

$$f(n) = n \bmod 2 + 10 * f(n/2)$$

Division by	Quotient	Remainder
10/2	5	0
5/2	2	1
2/2	1	0
1/2	0	1

$$\begin{aligned} & 101 * 10 + 0 = 1010 \\ & 10 * 10 + 1 = 101 \\ & 1 * 10 + 0 = 10 \end{aligned}$$



Dynamic programming & Memoization

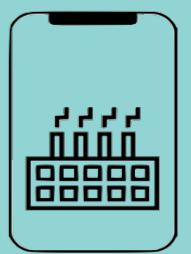


900



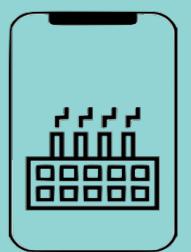
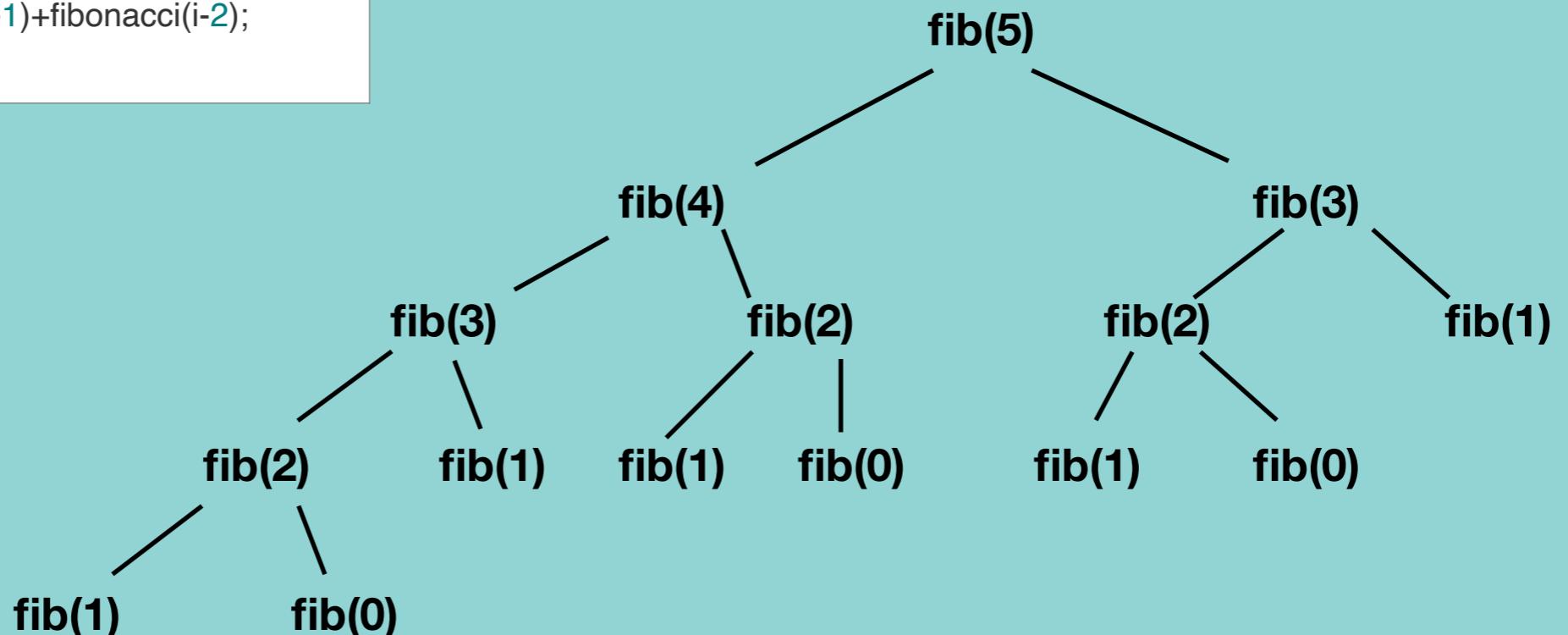
850

Total : $900 + 850 = 1750$



Dynamic programming & Memoization

```
static int fibonacci(int i) {  
    if (i == 0) return 0;  
    if (i == 1) return 0;  
    return fibonacci(i-1)+fibonacci(i-2);  
}
```



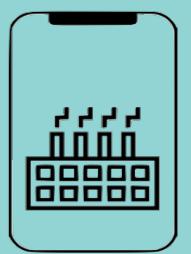
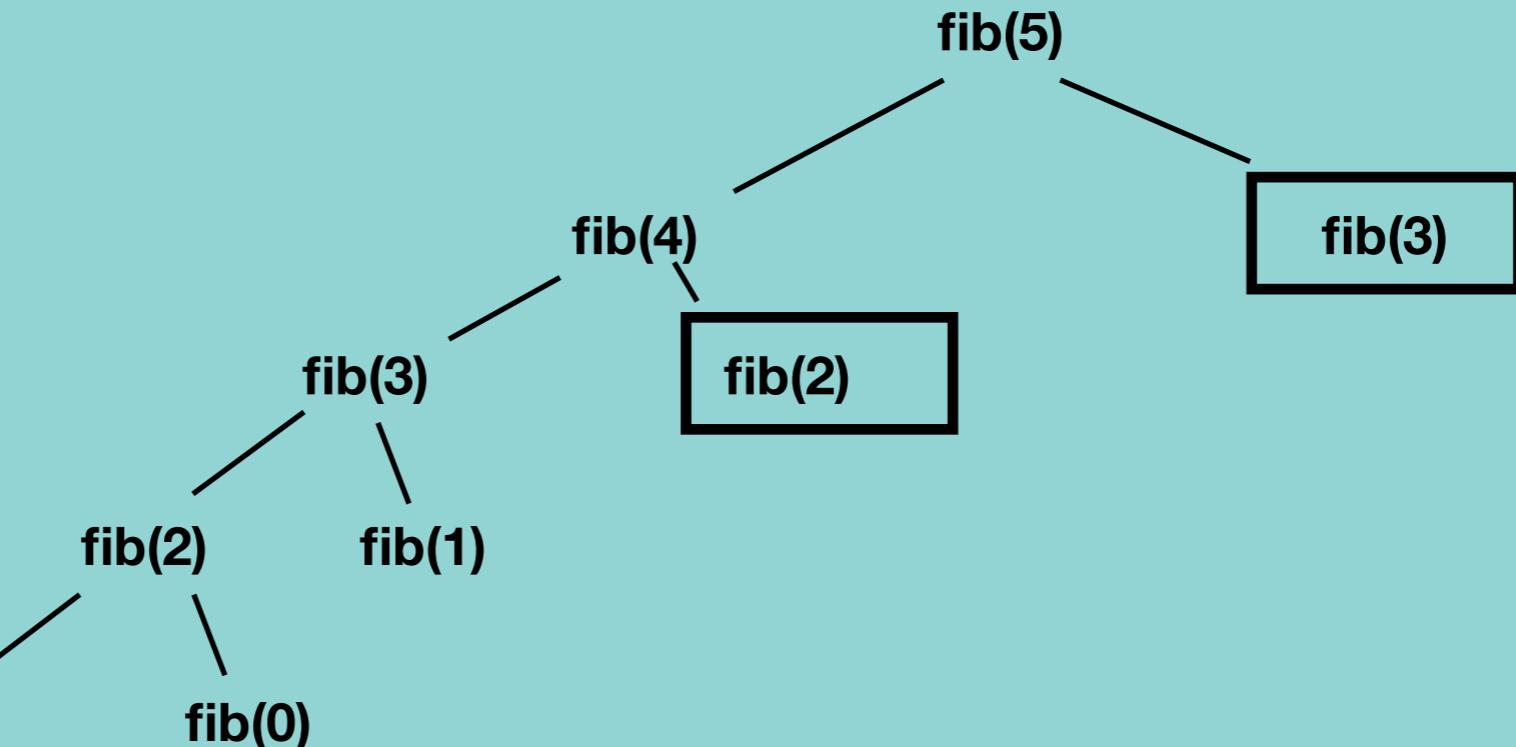
Dynamic programming & Memoization

```
static int fib(int n)
{
    /* Declare an array to store
    Fibonacci numbers. */
    int f[] = new int[n+2]; // 1 extra
    to handle case, n = 0
    int i;

    /* 0th and 1st number of the
    series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

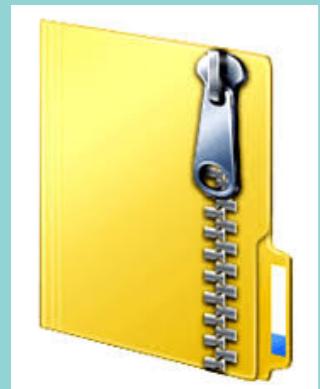
    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers
        in the series
        and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}
```

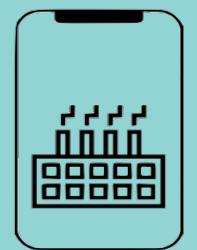


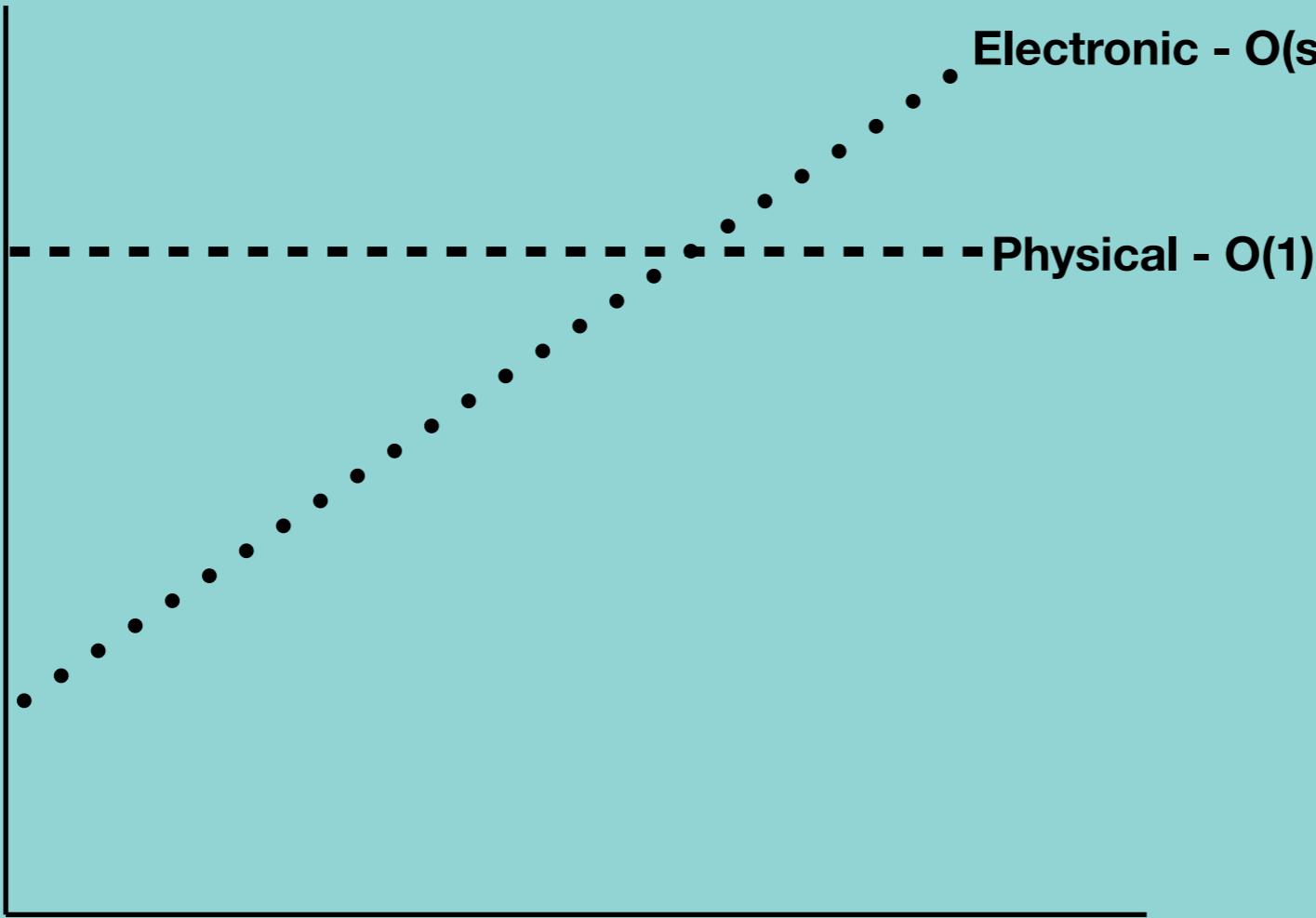
Big O

Big O is the language and metric we use to describe the efficiency of algorithms



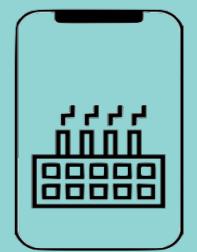
Size : 1TB





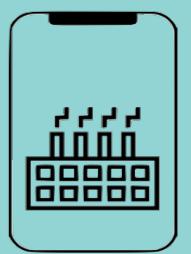
Types

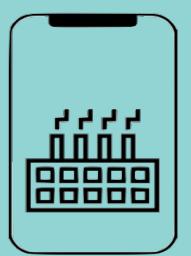
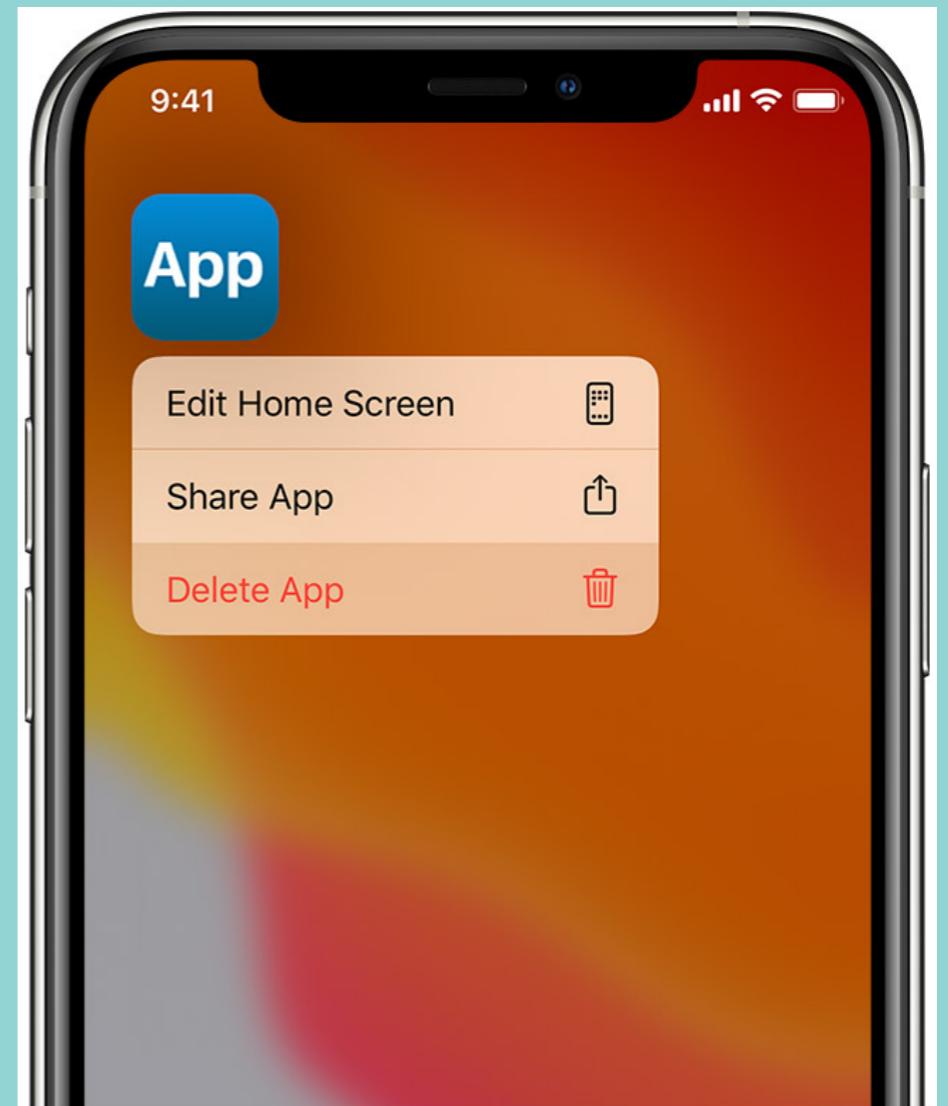
$O(N)$, $O(N^2)$, $O(2^N)$



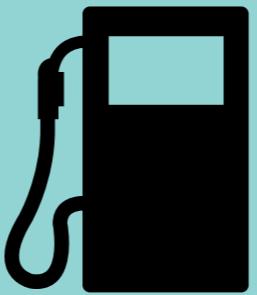
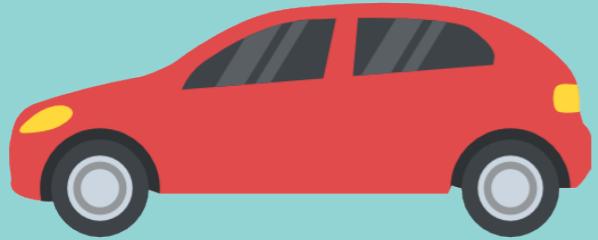


Time complexity : O(wh)

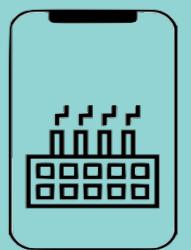




Algorithm run time notations

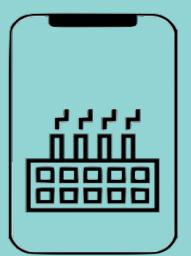
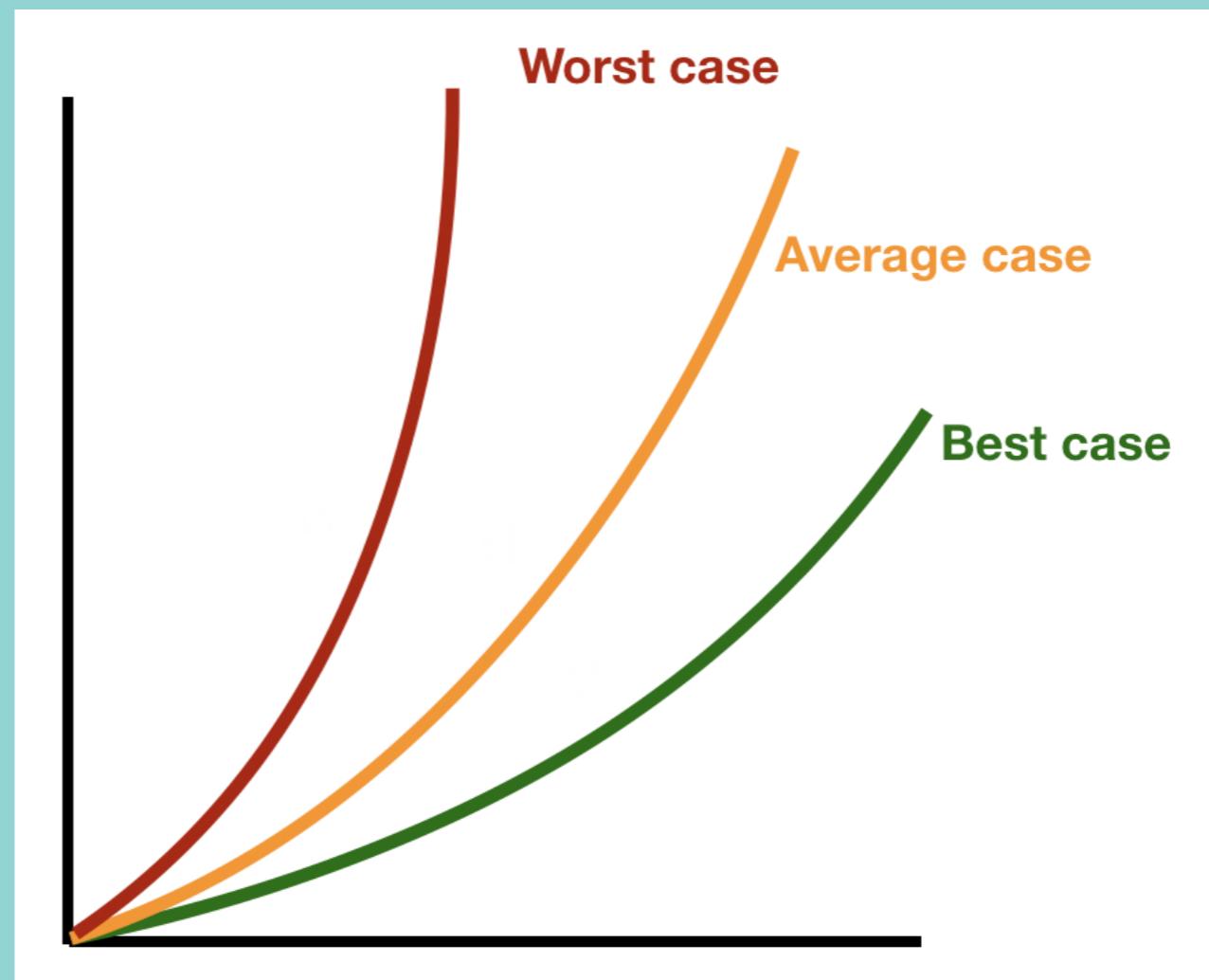


- **City traffic - 20 liters**
- **Highway - 10 liters**
- **Mixed condition - 15 liters**



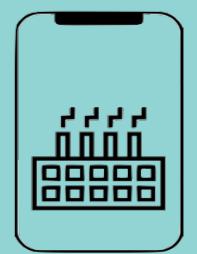
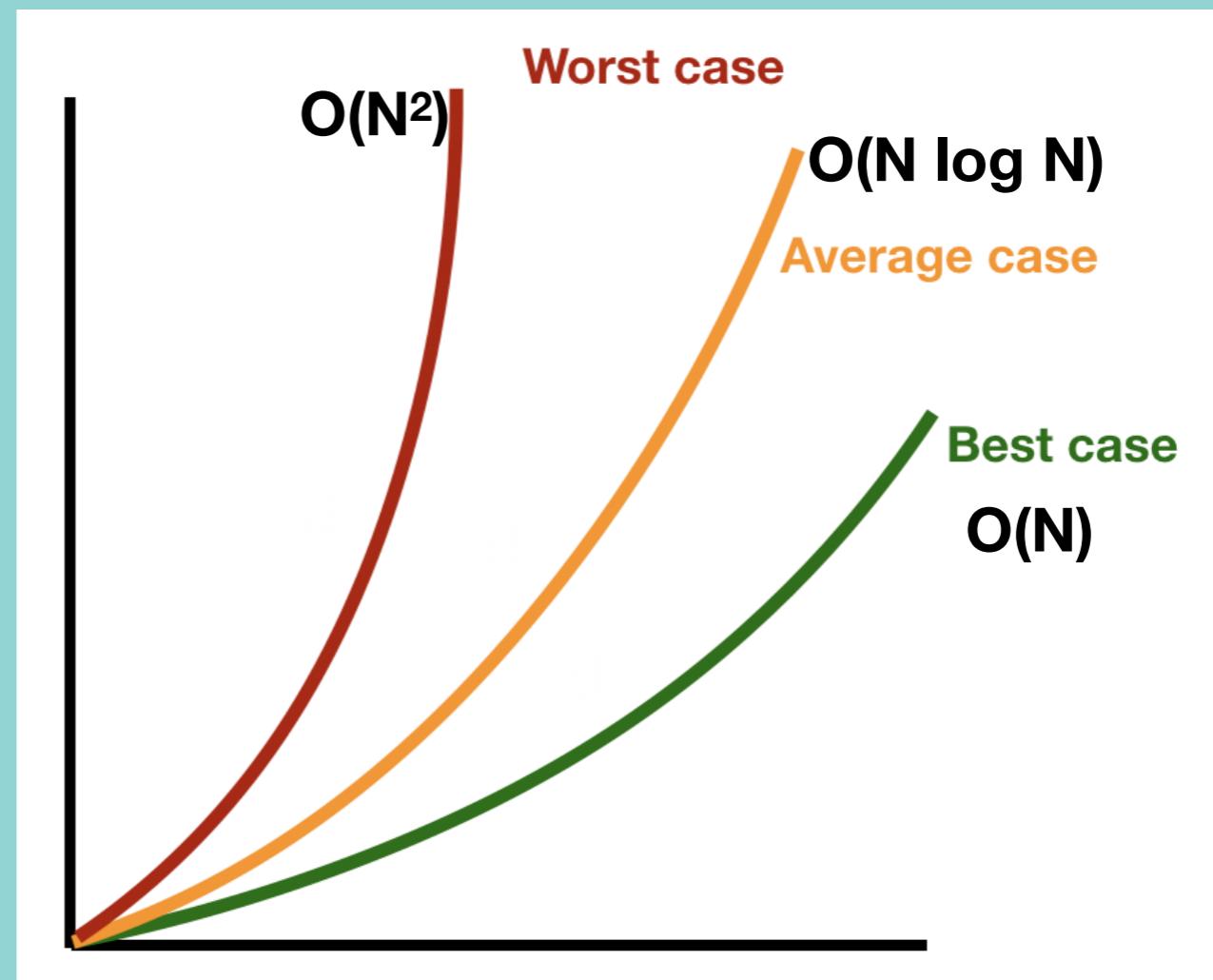
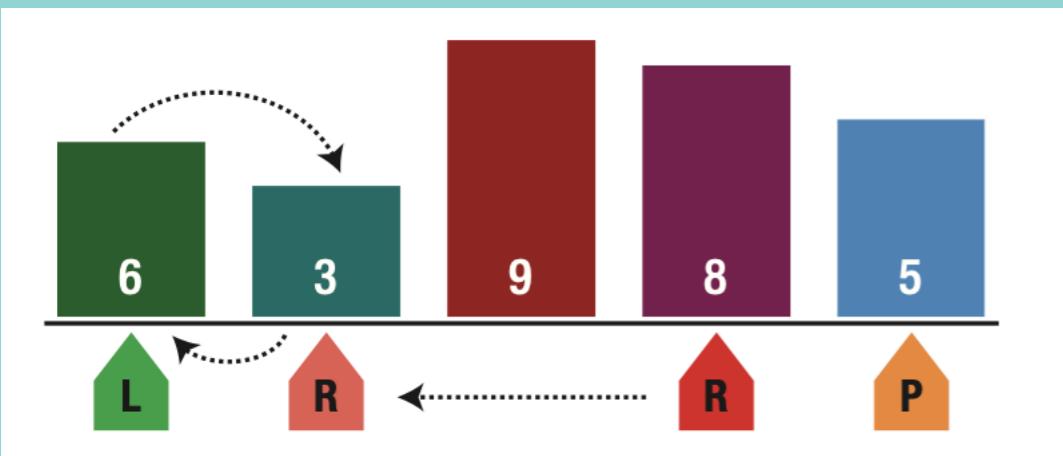
Algorithm run time notations

- Best case
- Average case
- Worst case



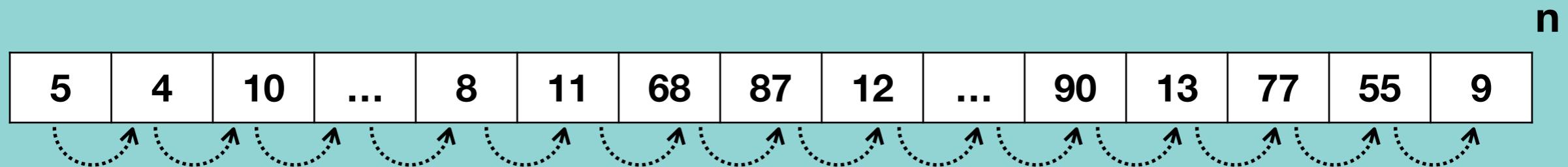
Algorithm run time notations

Quick sort algorithm



Big O, Big Theta and Big Omega

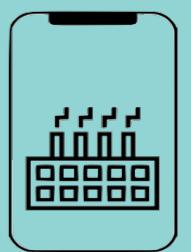
- **Big O** : It is a complexity that is going to be less or equal to the worst case.
- **Big - Ω (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big - Θ)** : It is a complexity that is within bounds of the worst and the best cases.



Big O - $O(N)$

Big Ω - $\Omega(1)$

Big Θ - $\Theta(n/2)$

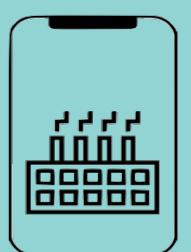


Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(1)$ - Constant time

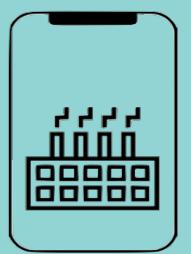
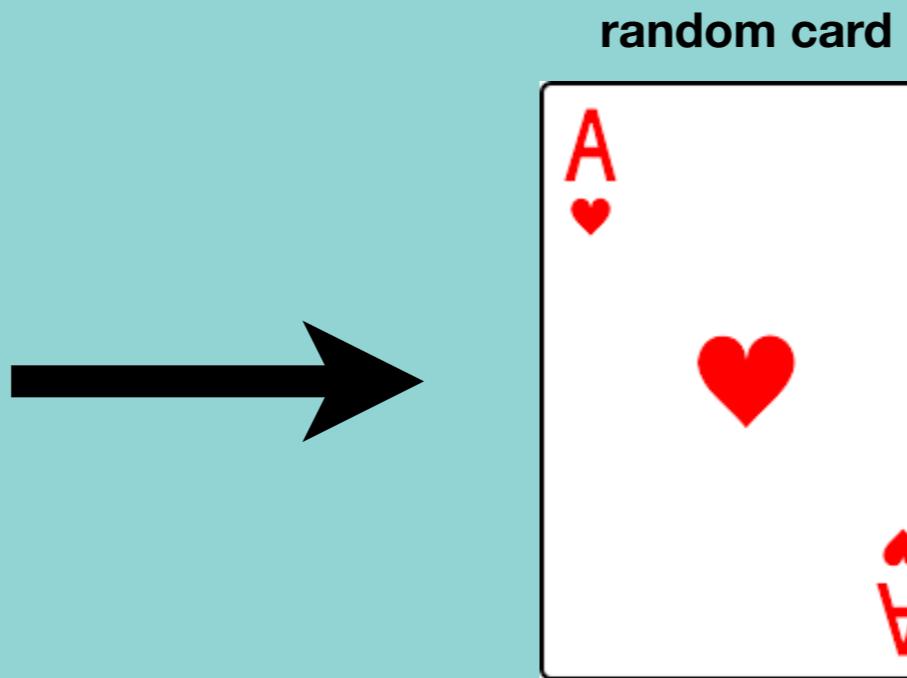
```
array = [1, 2, 3, 4, 5]
array[0] // It takes constant time to access first element
```



Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(1)$ - Constant time

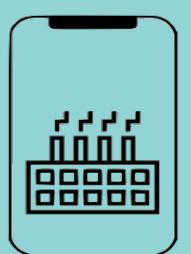


Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N)$ - Linear time

```
array = [1, 2, 3, 4, 5]
for element in array:
    print(element)
//linear time since it is visiting every element of array
```



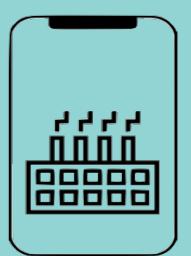
Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N)$ - Linear time



Specific card

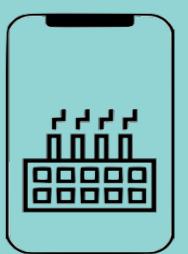


Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(\log N)$ - Logarithmic time

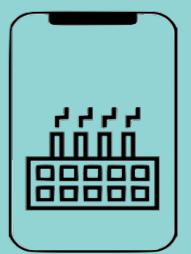
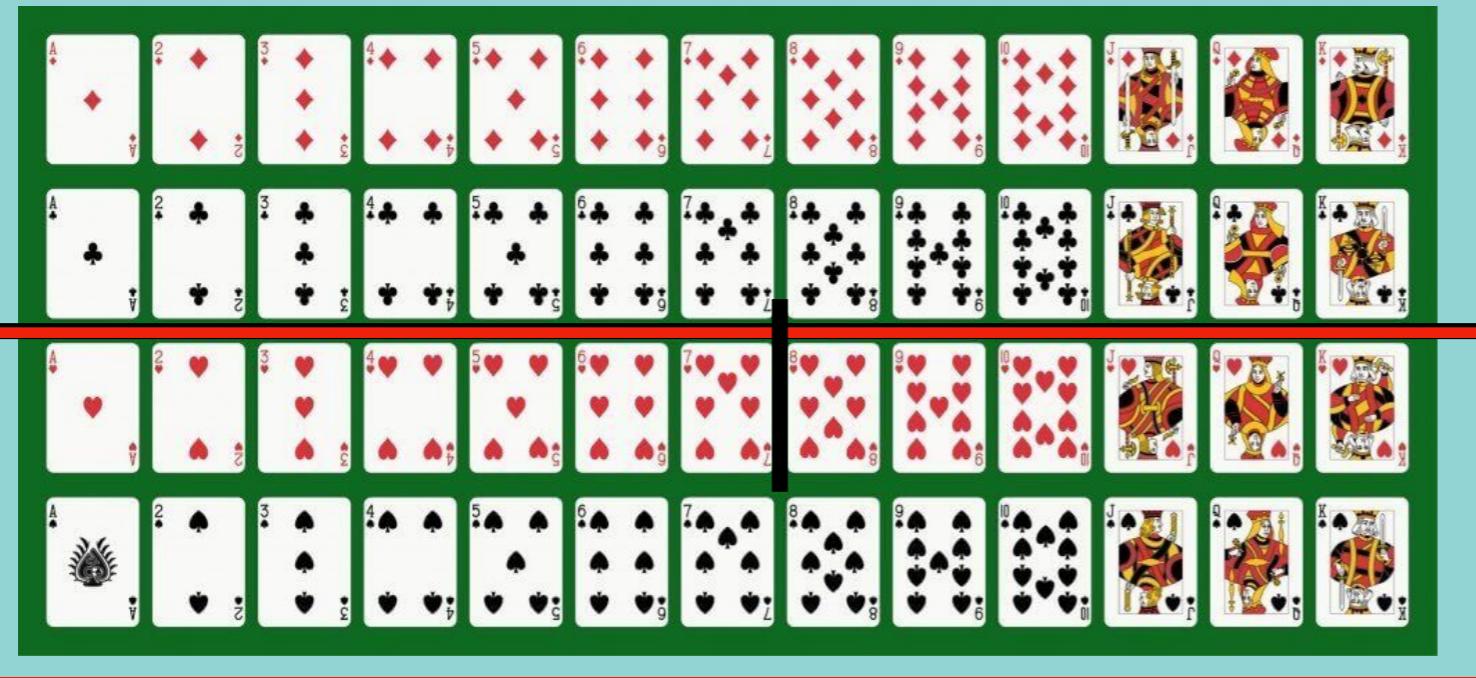
```
array = [1, 2, 3, 4, 5]
for index in range(0,len(array),3):
    print(array[index])
//logarithmic time since it is visiting only some elements
```



Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(\log N)$ - Logarithmic time



Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
O(N ²)	Quadratic	Looking at every index in the array twice
O(2 ^N)	Exponential	Double recursion in Fibonacci

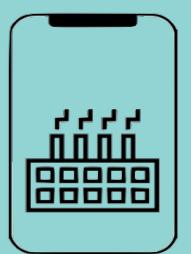
O(LogN) - Logarithmic time

Binary search

```
search 9 within [1,5,8,9,11,13,15,19,21]
compare 9 to 11 → smaller
search 9 within [1,5,8,9]
compare 9 to 8 → bigger
search 9 within [9]
compare 9 to 9
return
```

```
N = 16
N = 8 /* divide by 2 */
N = 4 /* divide by 2 */
N = 2 /* divide by 2 */
N = 1 /* divide by 2 */
```

$$2^k = N \rightarrow \log_2 N = k$$



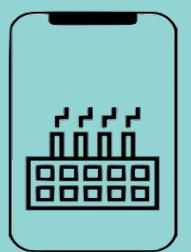
Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N^2)$ - Quadratic time

```
array = [1, 2, 3, 4, 5]

for x in array:
    for y in array:
        print(x,y)
```



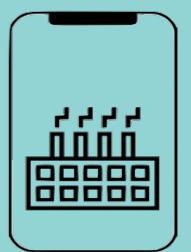
Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N^2)$ - Quadratic time

```
array = [1, 2, 3, 4, 5]

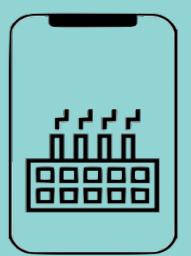
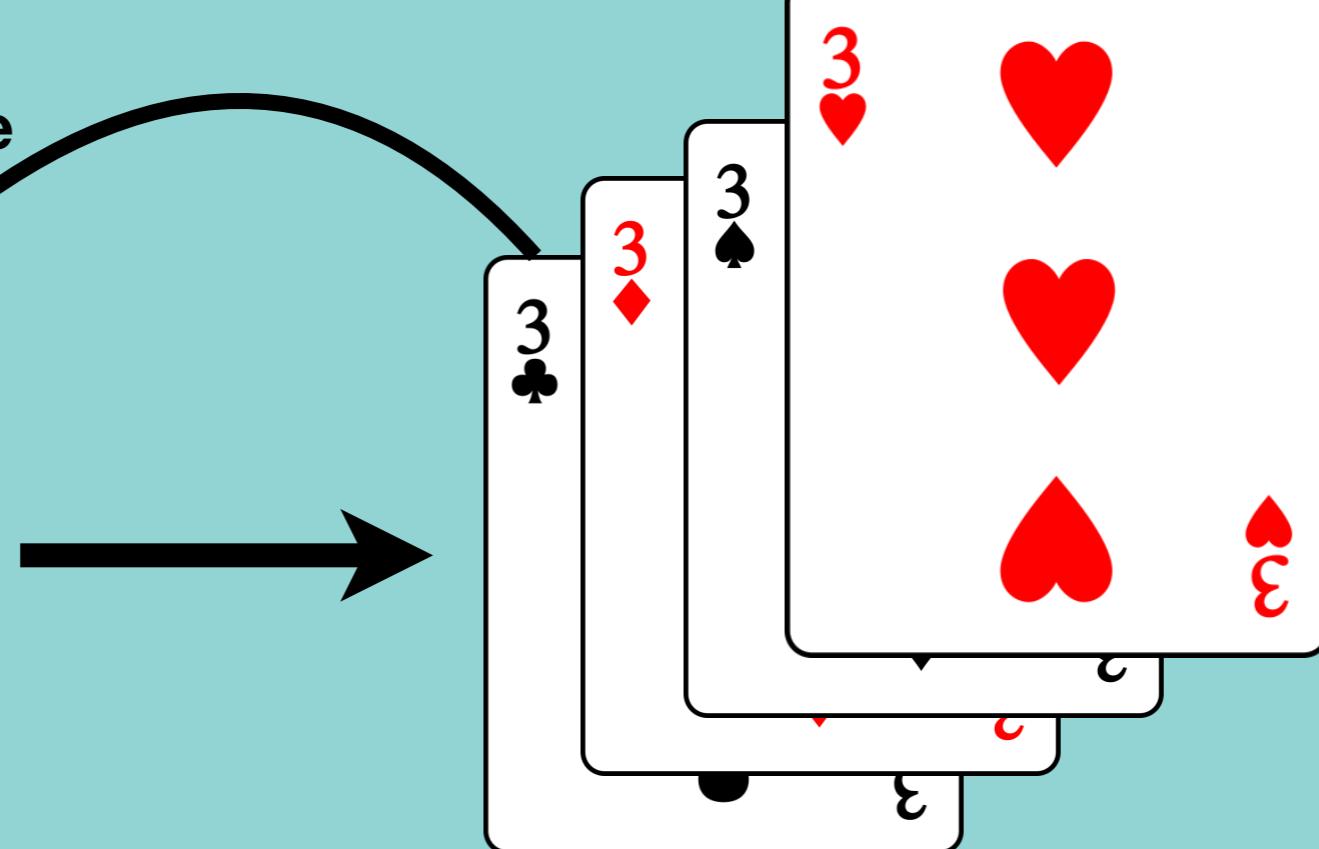
for x in array:
    for y in array:
        print(x,y)
```



Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N^2)$ - Quadratic time

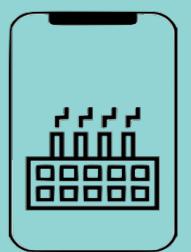


Algorithm run time complexities

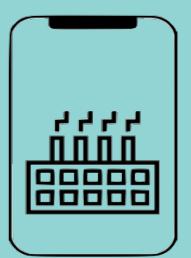
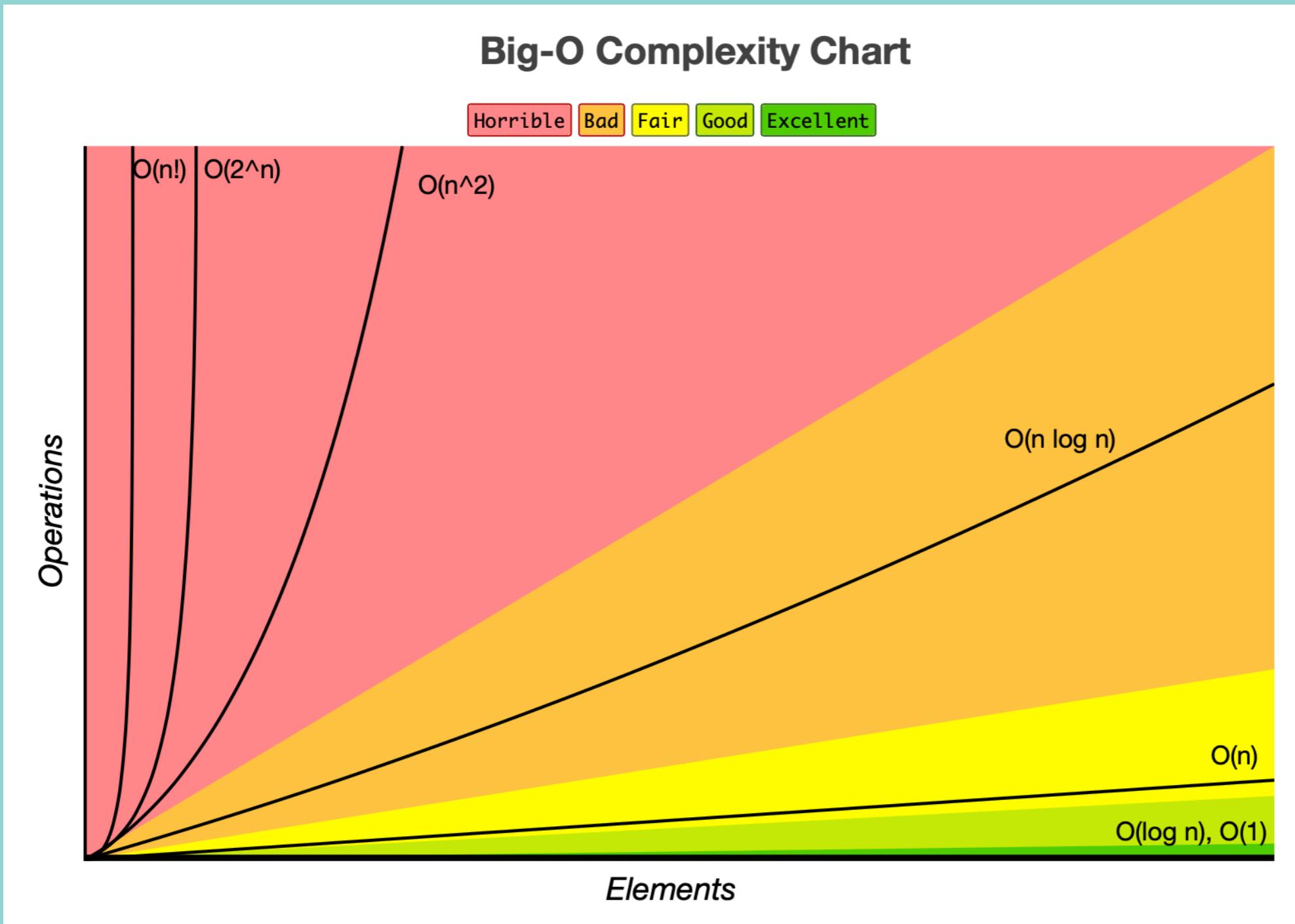
Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(2^N)$ - Exponential time

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```



Algorithm run time complexities



Space complexity

an array of size **n**

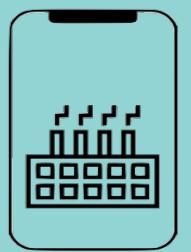
$$a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

O(n)

an array of size **n*n**

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$$

O(n²)

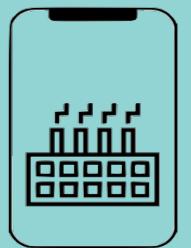


Space complexity - example

```
def sum(n):  
    if n <= 0:  
        return 0  
    else:  
        return n + sum(n-1)
```

1 sum(3)
2 → sum(2)
3 → sum(1)
4 → sum(0)

Space complexity : O(n)

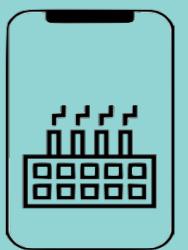


Space complexity - example

```
def pairSumSequence(n):
    sum = 0
    for i in range(0,n+1):
        sum = sum + pairSum(i, i+1)
    return sum

def pairSum(a,b):
    return a + b
```

Space complexity : O(1)



Space complexity - example

Java

```
static int sum(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + sum(n-1);  
}
```

Python

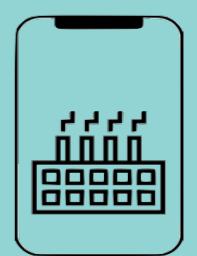
```
def sum(n):  
    if n <= 0:  
        return 0  
    else:  
        return n + sum(n-1)
```

Javascript

```
function sum(n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + sum(n-1);  
}
```

Swift

```
func sum(n: Int) -> Int {  
    if n <= 0 {  
        return 0  
    }  
    return n + sum(n: n-1)  
}
```



Drop Constants and Non Dominant Terms

Drop Constant

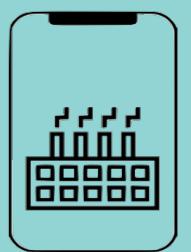
$$O(2N) \longrightarrow O(N)$$

Drop Non Dominant Terms

$$O(N^2+N) \longrightarrow O(N^2)$$

$$O(N+\log N) \longrightarrow O(N)$$

$$O(2^*2^N+1000N^{100}) \longrightarrow O(2^N)$$



Why do we drop constants and non dominant terms?

- It is very possible that O(N) code is faster than O(1) code for specific inputs
- Different computers with different architectures have different constant factors.



**Fast computer
Fast memory access
Lower constant**

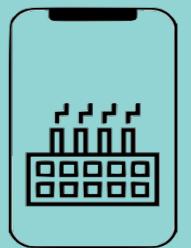


**Slow computer
Slow memory access
Higher constant**

- Different algorithms with the same basic idea and computational complexity might have slightly different constants

Example: $a^*(b-c)$ vs $a^*b - a^*c$

- As $n \rightarrow \infty$, constant factors are not really a big deal



Add vs Multiply

```
for a in arrayA:  
    print(a)
```

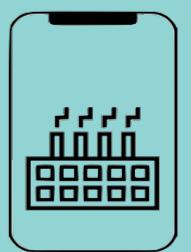
```
for b in arrayB:  
    print(b)
```

```
for a in arrayA:  
    for b in arrayB:  
        print(a, b)
```

Add the Runtimes: $O(A + B)$

Multiply the Runtimes: $O(A * B)$

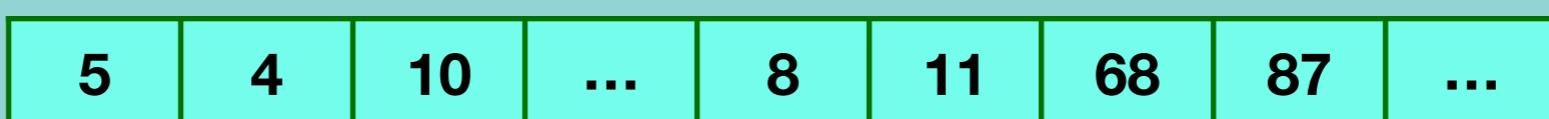
- If your algorithm is in the form “do this, then when you are all done, do that” then you add the runtimes.
- If your algorithm is in the form “do this for each time you do that” then you multiply the runtimes.



How to measure the codes using Big O?

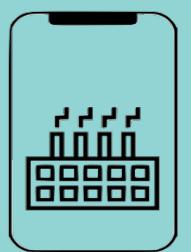
No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple “for” loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	$O(\log n)$
Rule 5	When dealing with multiple statements, just add them up	

sampleArray



```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0] -----> O(1)
    for index in range(1, len(sampleArray)): -----> O(n)
        if sampleArray[index] > biggestNumber: -----> O(1) } -----> O(n)
            biggestNumber = sampleArray[index] -----> O(1) } -----> O(1) }
    print(biggestNumber) -----> O(1)
```

Time complexity : $O(1) + O(n) + O(1) = O(n)$



How to measure Recursive Algorithm?

sampleArray

5	4	10	...	8	11	68	87	10
---	---	----	-----	---	----	----	----	----

```
def findMaxNumRec(sampleArray, n):  
    if n == 1:  
        return sampleArray[0]  
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1))
```

Explanation:

A =

11	4	12	7
----	---	----	---

 n = 4

findMaxNumRec(A, 4) → max(A[4-1], 12) → max(7, 12)=12



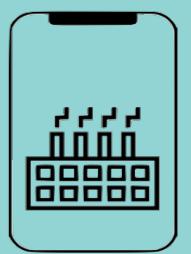
findMaxNumRec(A, 3) → max(A[3-1], 11) → max(12, 11)=12



findMaxNumRec(A, 2) → max(A[2-1], 11) → max(4, 11)=11



findMaxNumRec(A, 1) → A[0]=11



How to measure Recursive Algorithm?

sampleArray

5	4	10	...	8	11	68	87	10
---	---	----	-----	---	----	----	----	----

```
def findMaxNumRec(sampleArray, n):-----> M(n)
    if n == 1: -----> O(1)
        return sampleArray[0] -----> O(1)
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1))-----> M(n-1)
```

$$M(n) = O(1) + M(n-1)$$

$$M(1) = O(1)$$

$$M(n-1) = O(1) + M((n-1)-1)$$

$$M(n-2) = O(1) + M((n-2)-1)$$

{}

$$M(n) = 1 + M(n-1)$$

$$= 1 + (1 + M((n-1)-1))$$

$$= 2 + M(n-2)$$

$$= 2 + 1 + M((n-2)-1)$$

$$= 3 + M(n-3)$$

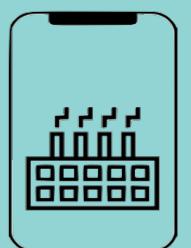
.

$$= a + M(n-a)$$

$$= n-1 + M(n-(n-1))$$

$$= n-1+1$$

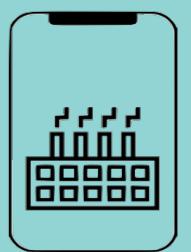
$$= n$$



How to measure Recursive Algorithm that make multiple calls?

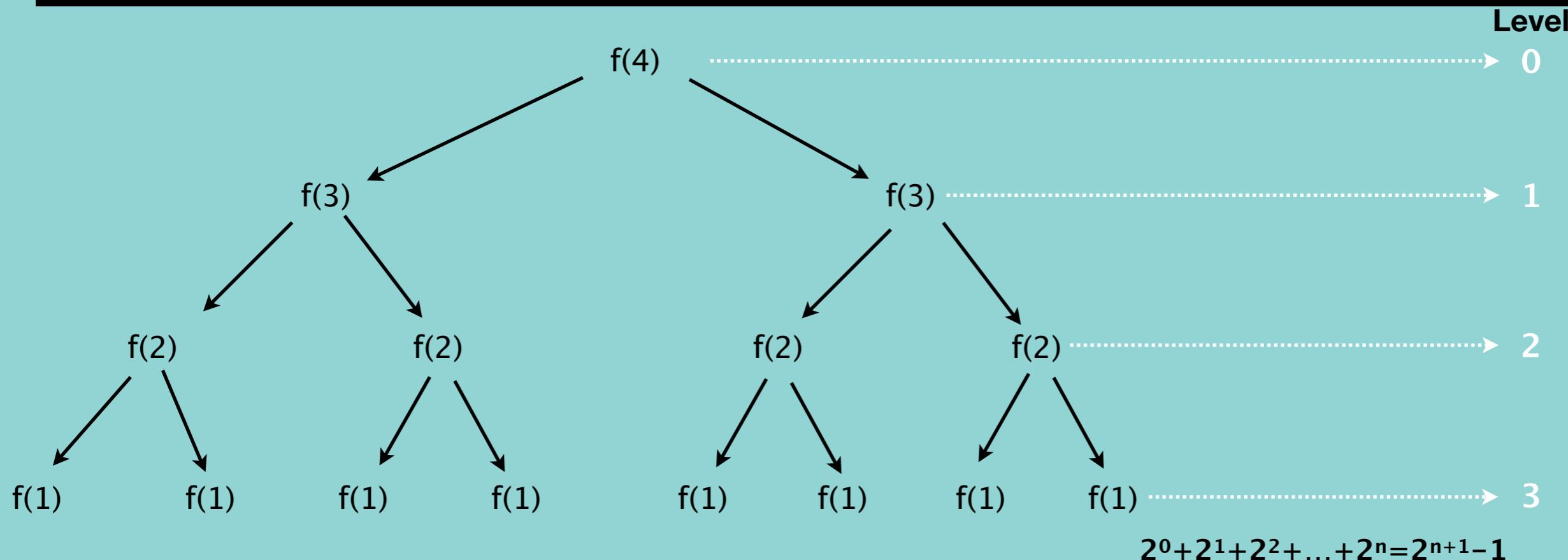
```
def findMaxNumRec(sampleArray, n):
    if n == 1:
        return sampleArray[0]
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1))
```

```
def f(n):
    if n <= 1:
        return 1
    return f(n-1) + f(n-1)
```



How to measure Recursive Algorithm that make multiple calls?

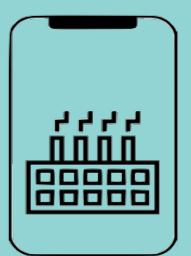
```
def f(n):
    if n <= 1:
        return 1
    return f(n-1) + f(n-1)
```



N	Level	Node#	Also can be expressed..	or..
4	0	1		2^0
3	1	2	$2 * \text{previous level} = 2$	2^1
2	2	4	$2 * \text{previous level} = 2 * 2^1 = 2^2$	2^2
1	3	8	$2 * \text{previous level} = 2 * 2^2 = 2^3$	2^3

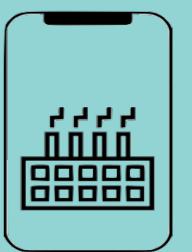
$O(\text{branches}^{\text{depth}})$

AppMillers
www.appmillers.com

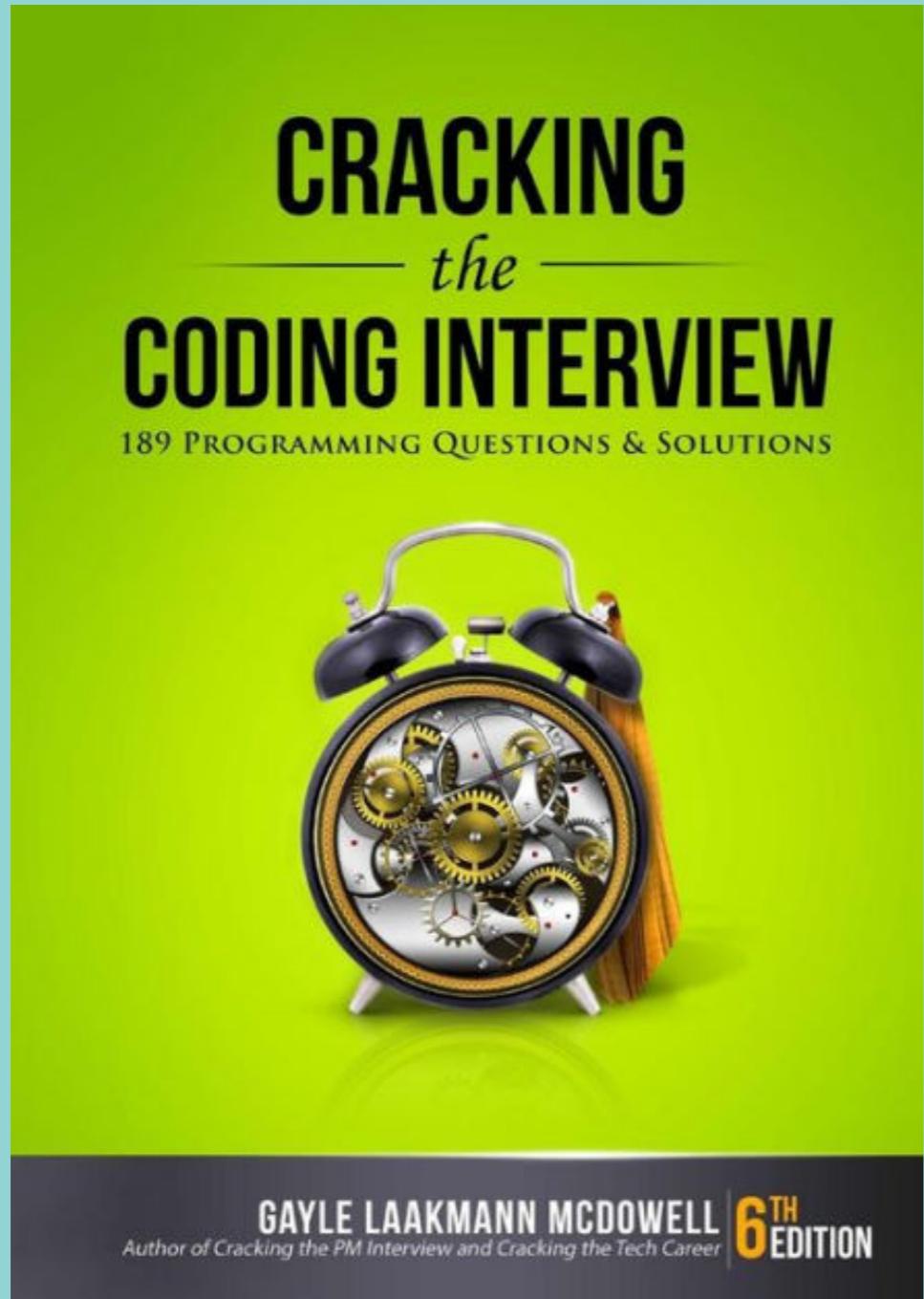


$2^n - 1 \longrightarrow O(2^n)$

Top 10 Big O interview Questions

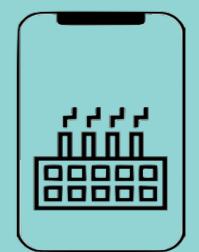


Top 10 Big O interview Questions



in Java 😞

Java to Python 😊



Interview Questions - 1

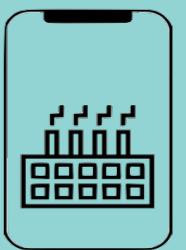
What is the runtime of the below code?

```
def foo(array):
    sum = 0 -----> O(1)
    product = 1 -----> O(1)

    for i in array: -----> O(n)
        sum += i-----> O(1)

    for i in array: -----> O(n)
        product *= i -----> O(1)
    print("Sum = "+str(sum)+", Product = "+str(product)) -----> O(1)
```

Time Complexity : O(N)

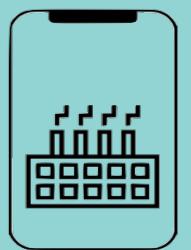


Interview Questions - 2

What is the runtime of the below code?

```
def printPairs(array):
    for i in array:-----> O(n²)}
        for j in array:-----> O(n) } -----> O(n²)
            print(str(i)+", "+str(j)) -----> O(1)
```

Time Complexity : $O(N^2)$



Interview Questions - 3

What is the runtime of the below code?

```
def printUnorderedPairs(array):
    for i in range(0, len(array)):
        for j in range(i+1, len(array)):
            print(array[i] + "," + array[j])
```

1. Counting the iterations

1st $\longrightarrow n-1$

2nd $\longrightarrow n-2$

.

1

$(n-1)+(n-2)+(n-3)+..+2+1$

$=1+2+...+(n-3)+(n-2)+(n-1)$

$=n(n-1)/2$

$=n^2/2 + n$

$=n^2$

Time Complexity : $O(N^2)$

2. Average Work

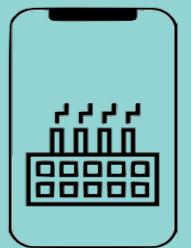
Outer loop – N times

Inner loop?

1st $\longrightarrow 10$
2nd $\longrightarrow 9$
. .
1

$\} = 5 \longrightarrow 10/2$
 $n \longrightarrow n/2$

$n*n/2 = n^2/2 \longrightarrow O(N^2)$



Interview Question - 4

What is the runtime of the below code?

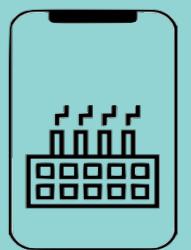
```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            if arrayA[i] < arrayB[j]:
                print(str(arrayA[i]) + "," + str(arrayB[j]))
```

```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            O(1)
```

b = len(arrayB)

a = len(arrayA)

Time Complexity : O(ab)



Interview Question - 5

What is the runtime of the below code?

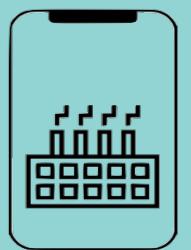
```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            for k in range(0,100000): -----> O(ab)
                print(str(arrayA[i]) + "," + str(arrayB[j])) -----> O(1)
```

a = len(arrayA)

b = len(arrayB)

100,000 units of work is still constant

Time Complexity : O(ab)



Interview Question - 6

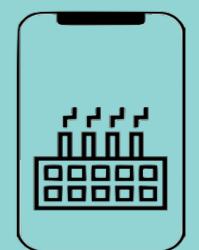
What is the runtime of the below code?

```
def reverse(array):
    for i in range(0, int(len(array)/2)):
        other = len(array)-i-1
        temp = array[i]
        array[i] = array[other]
        array[other] = temp
    print(array)
```

→ O(N/2) → O(N)
→ O(1)
→ O(1)
→ O(1)
→ O(1)
→ O(1)



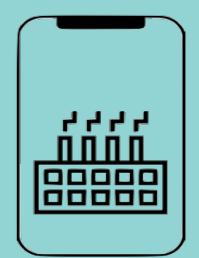
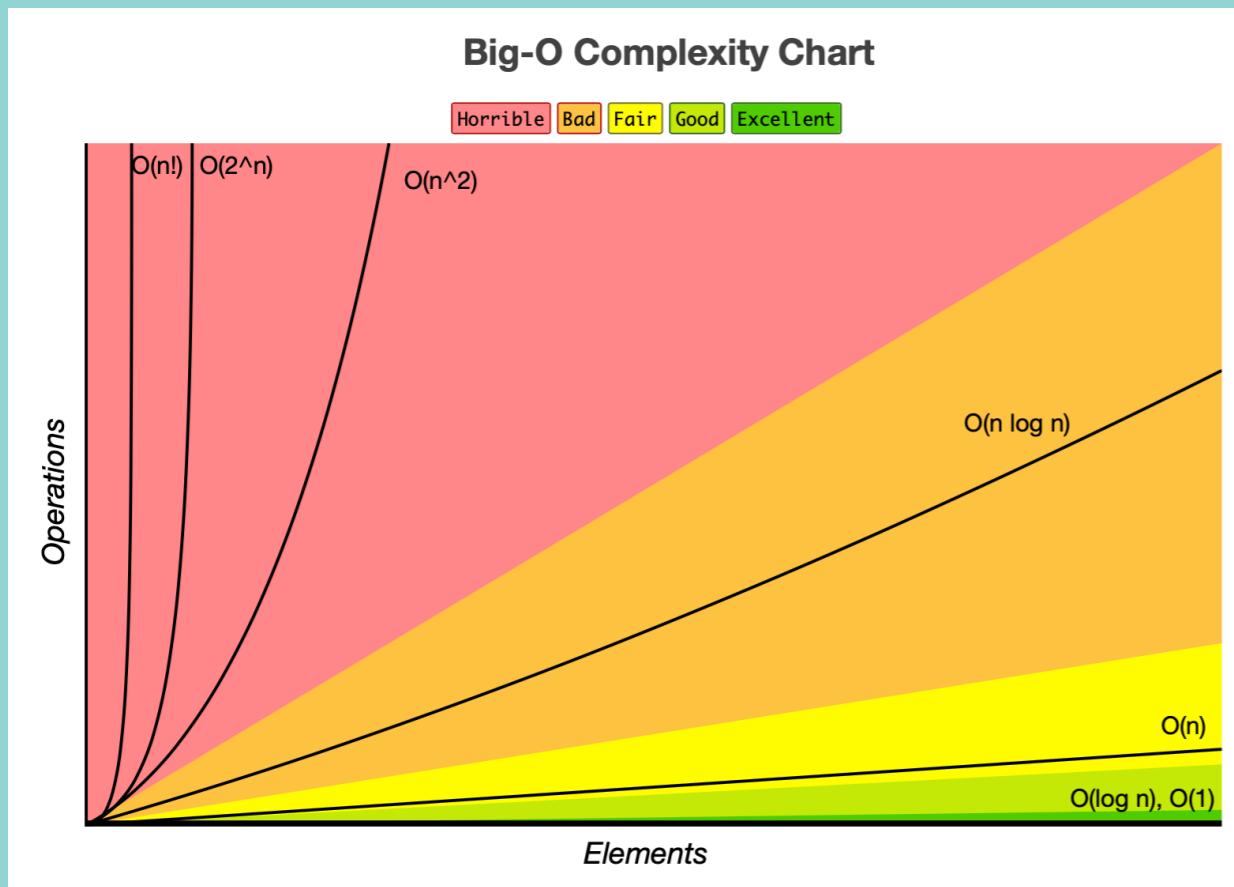
Time Complexity : O(N)



Interview Questions - 7

Which of the following are equivalent to $O(N)$? Why?

1. $O(N + P)$, where $P < N/2 \longrightarrow O(N) \quad \checkmark$
2. $O(2N) \longrightarrow O(N) \quad \checkmark$
3. $O(N+\log N) \longrightarrow O(N) \quad \checkmark$
4. $O(N + N\log N) \longrightarrow O(N\log N) \quad \times$
5. $O(N+M) \quad \times$



Interview Question - 8

What is the runtime of the below code?

```
def factorial(n):-----> M(n)
    if n < 0:
        return -1
    elif n == 0:-----> O(1)
        return 1
    else:
        return n * factorial(n-1)-----> M(n-1)
```

$$n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$$

$$3! = 1 \cdot 2 \cdot 3 = 6$$

$$M(n) = O(1) + M(n-1)$$

$$M(0) = O(1)$$

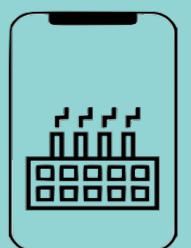
$$M(n-1) = O(1) + M((n-1)-1)$$

$$M(n-2) = O(1) + M((n-2)-1)$$

}

$$\begin{aligned} M(n) &= 1 + M(n-1) \\ &= 1 + (1 + M((n-1)-1)) \\ &= 2 + M(n-2) \\ &= 2 + 1 + M((n-2)-1) \\ &= 3 + M(n-3) \\ &\quad \vdots \\ &= a + M(n-a) \\ &= n + M(n-n) \\ &= n + 1 \\ &= n \end{aligned}$$

Time Complexity : O(N)



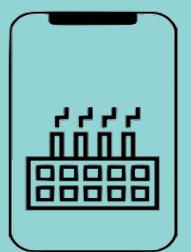
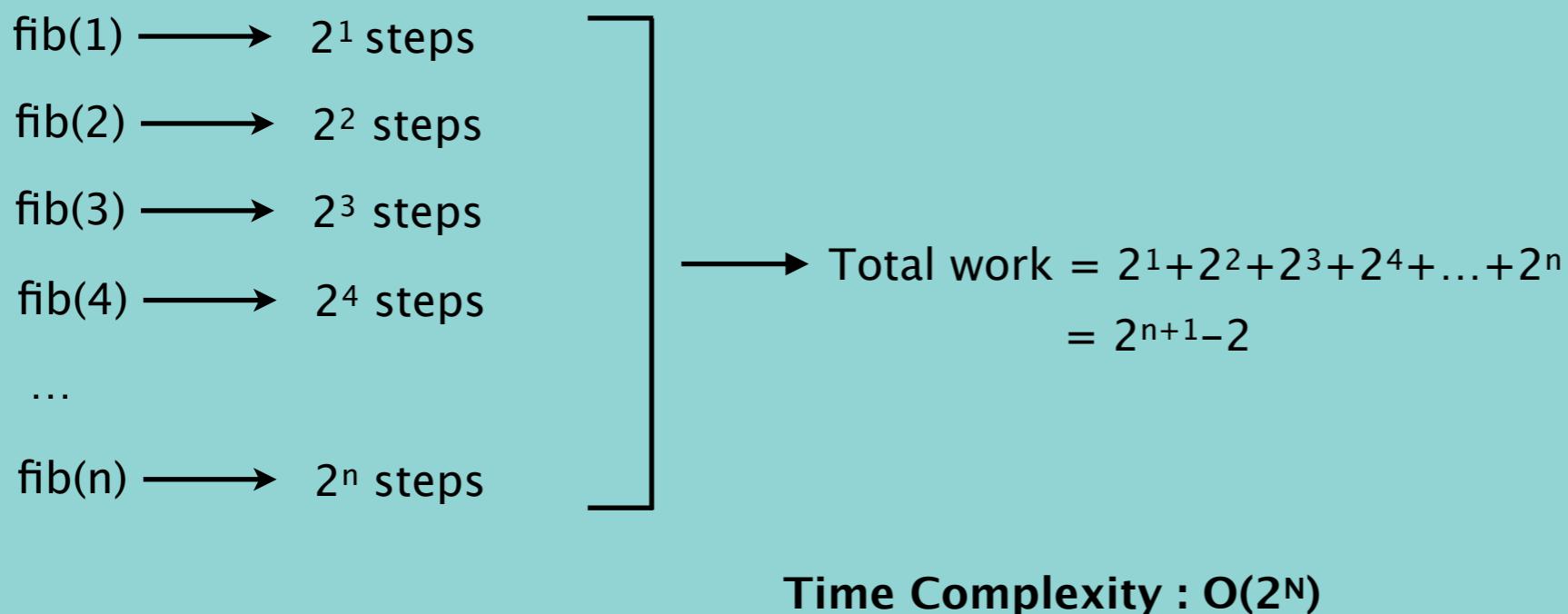
Interview Question - 9

What is the runtime of the below code?

```
def allFib(n):
    for i in range(n):
        print(str(i)+":", " " + str(fib(i)))

def fib(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

branches^{depth} -----> O(2^N)

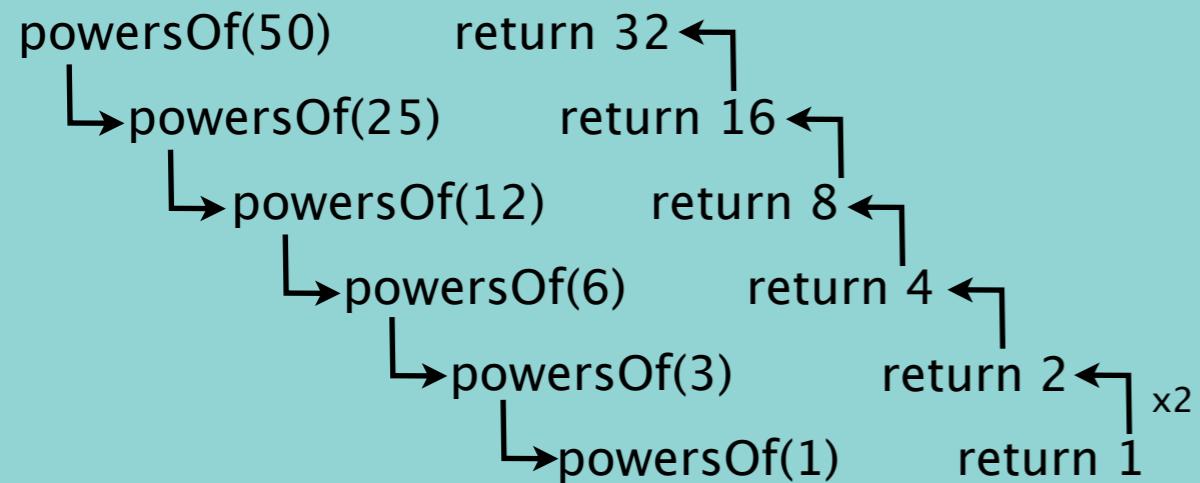


Interview Question - 10

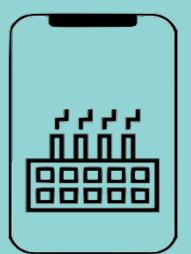
What is the runtime of the below code?

```
def powersOf2(n):
    if n < 1:
        return 0
    elif n == 1:
        print(1)
        return 1
    else:
        prev = powersOf2(int(n/2))
        curr = prev*2
        print(curr)
        return curr
```

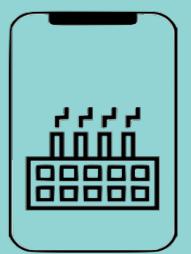
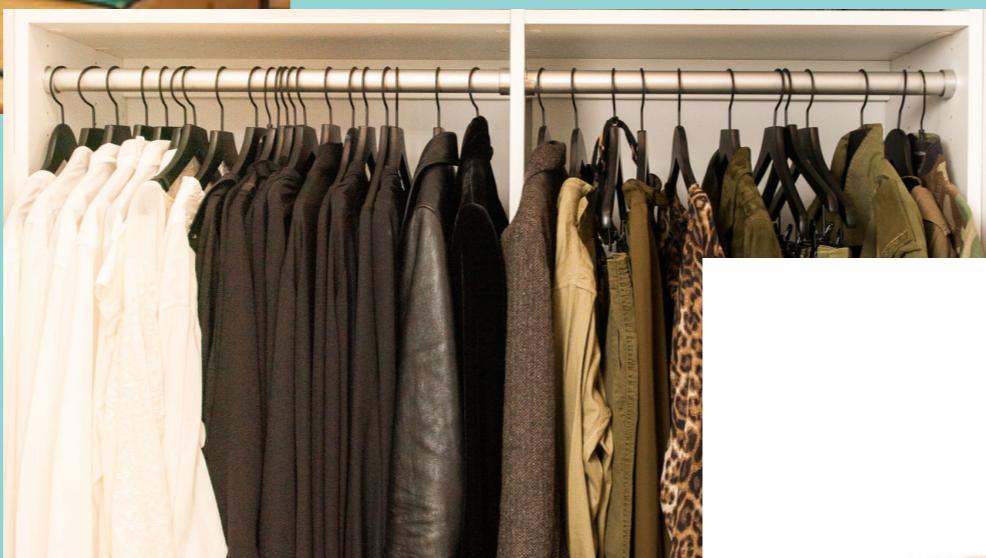
n=50



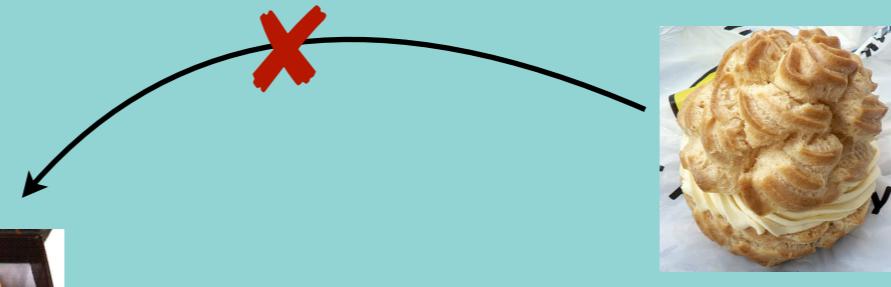
Time Complexity : O(logN)



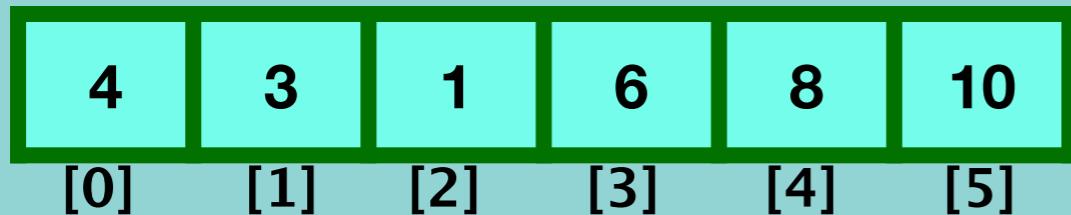
Arrays



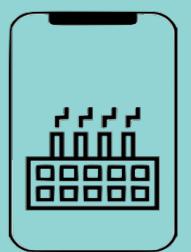
Arrays



- It is a box of macaroons.
- All macaroons in this box are next to each other
- Each macaroon can be identified uniquely based on their location
- The size of box cannot be changed

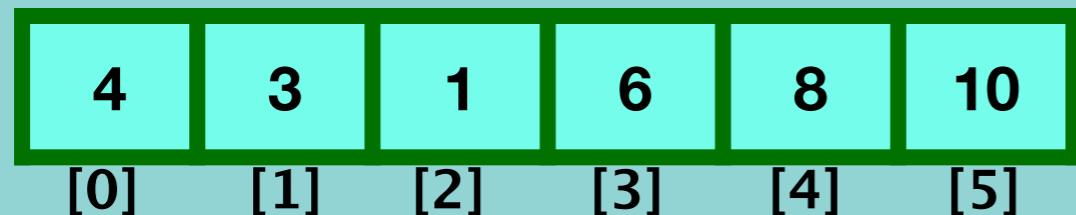


- Array can store data of specified type
- Elements of an array are located in a contiguous
- Each element of an array has a unique index
- The size of an array is predefined and cannot be modified



What is an Array?

In computer science, an array is a data structure consisting of a collection of elements , each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index by a mathematical formula.



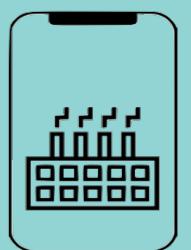
Why do we need an Array?

3 variables

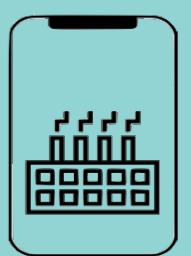
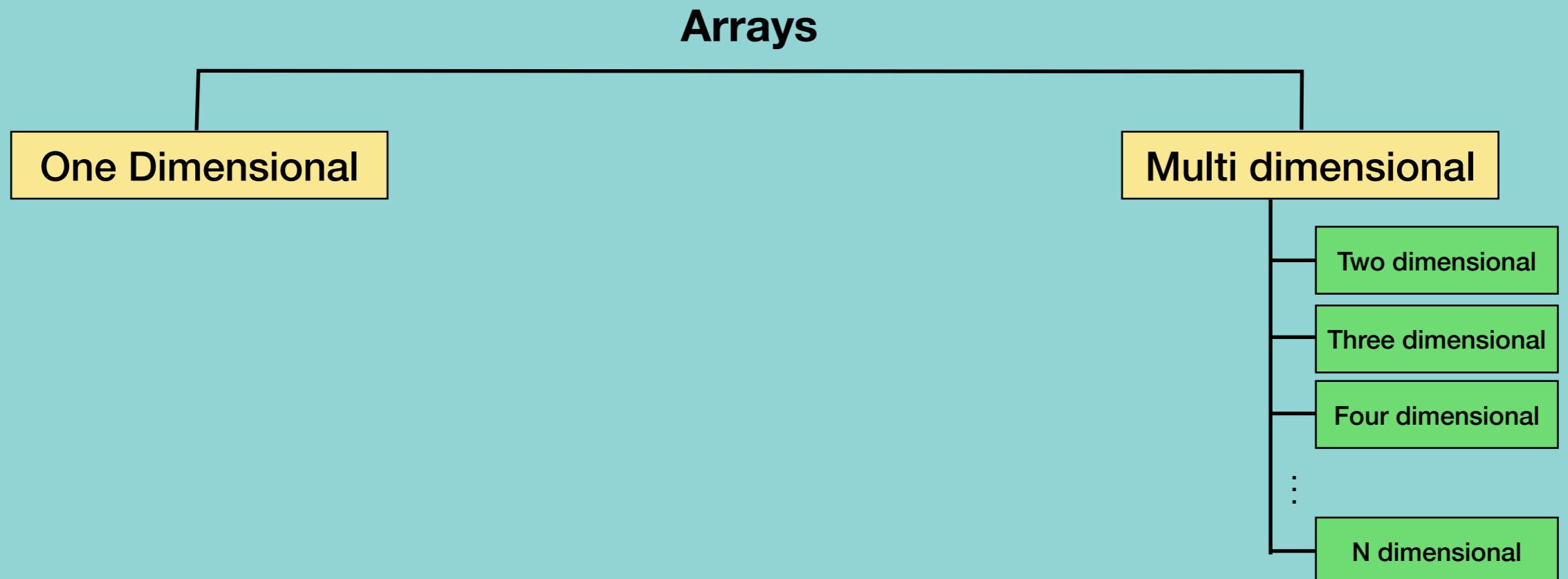
number1
number2
number3

- What if 500 integer?
- Are we going to use 500 variables?

The answer is an **ARRAY**

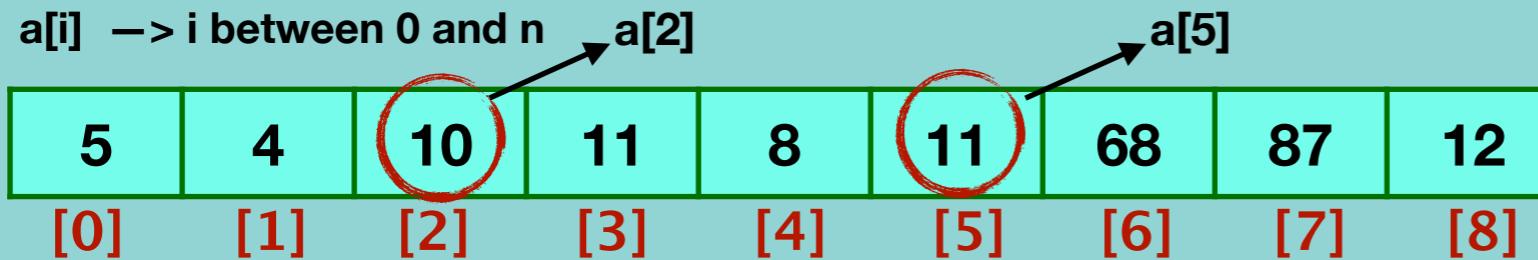


Types of Array

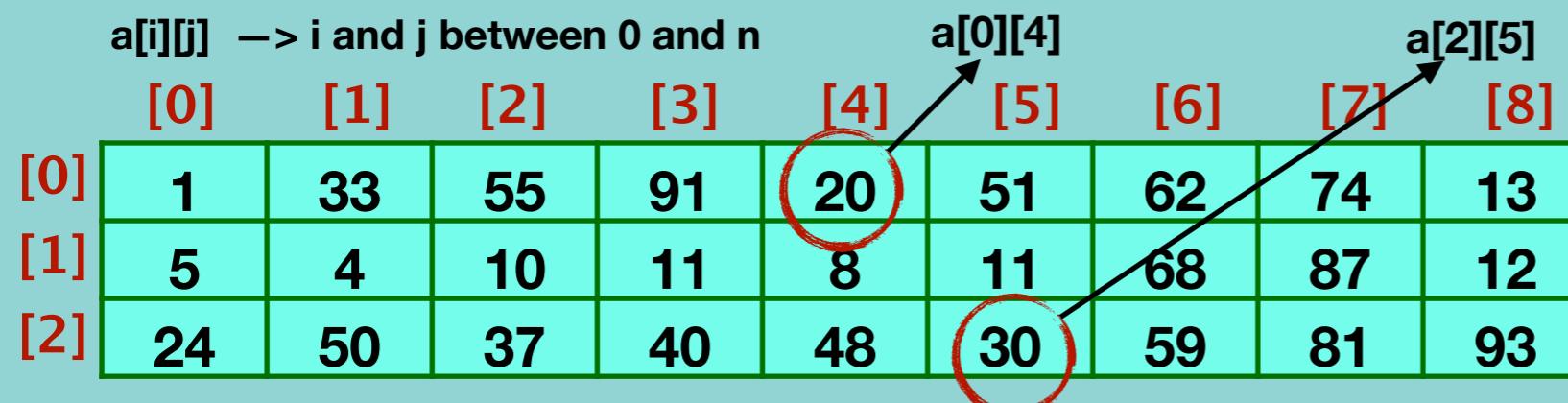


Types of Array

One dimensional array : an array with a bunch of values having been declared with a single index.

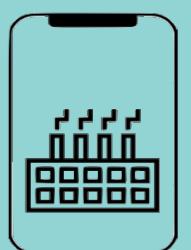
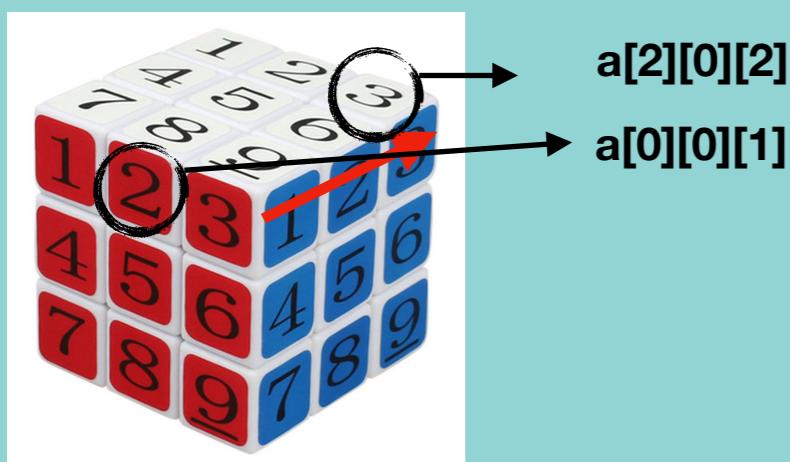


Two dimensional array : an array with a bunch of values having been declared with double index.



Three dimensional array : an array with a bunch of values having been declared with triple index.

$a[i][j][k] \rightarrow i, j$ and k between 0 and n



Types of Array

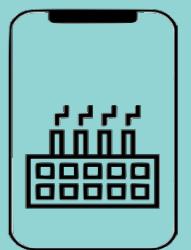
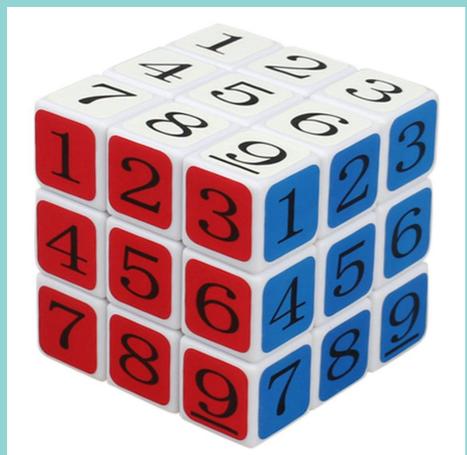
One dimensional array :

5	4	10	11	8	11	68	87	12
---	---	----	----	---	----	----	----	----

Two dimensional array

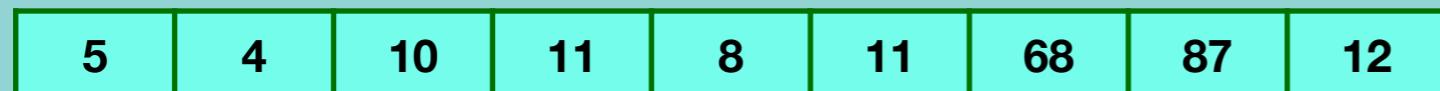
1	33	55	91	20	51	62	74	13
5	4	10	11	8	11	68	87	12
24	50	37	40	48	30	59	81	93

Three dimensional array

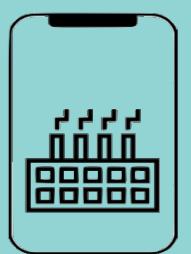
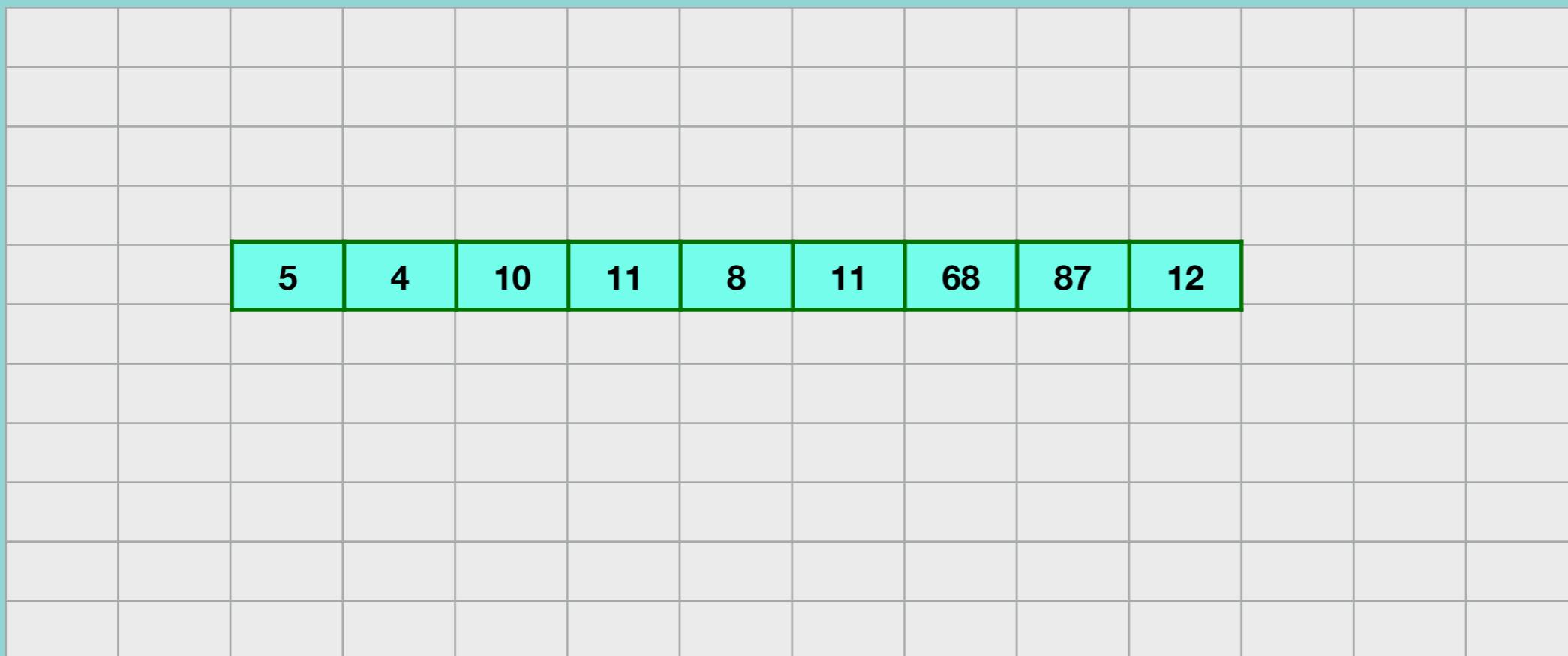


Arrays in Memory

One Dimensional



Memory

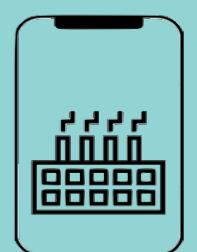
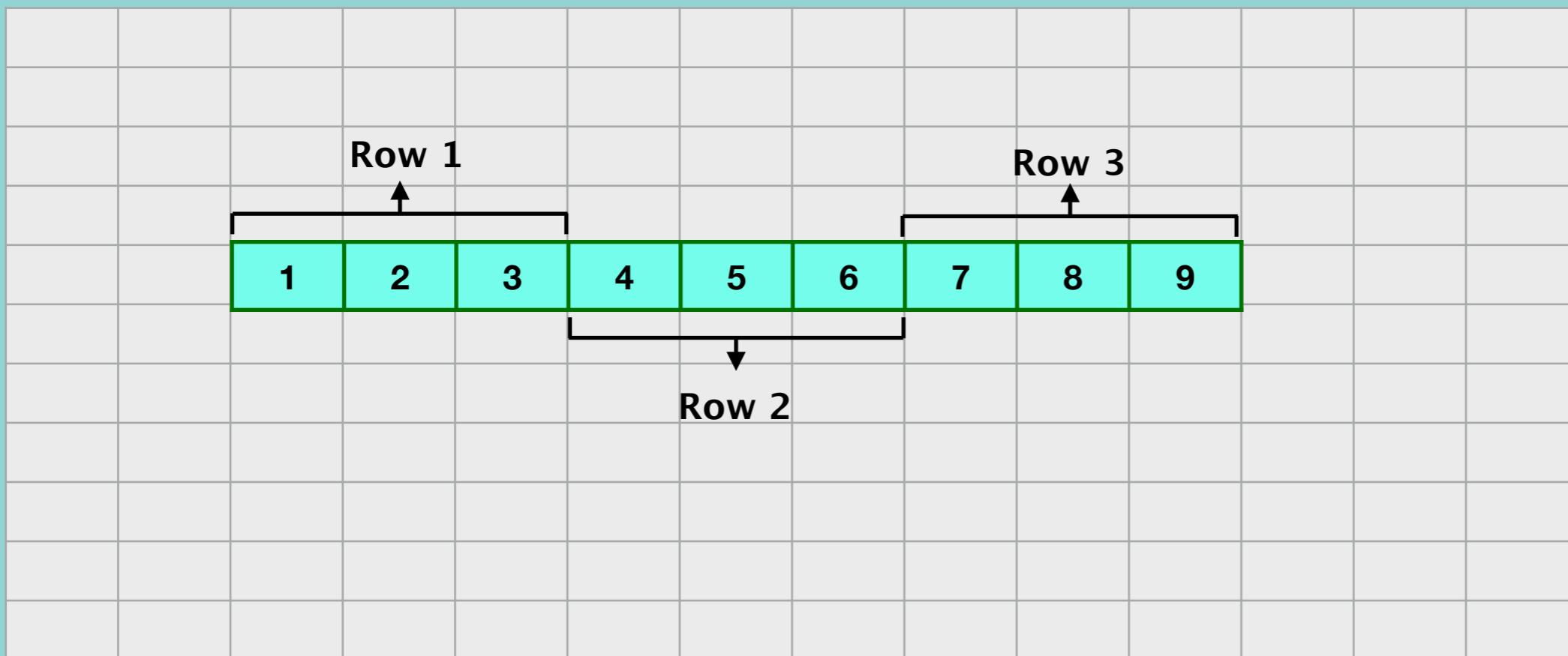


Arrays in Memory

Two Dimensional array

1	2	3
4	5	6
7	8	9

Memory

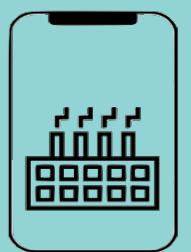
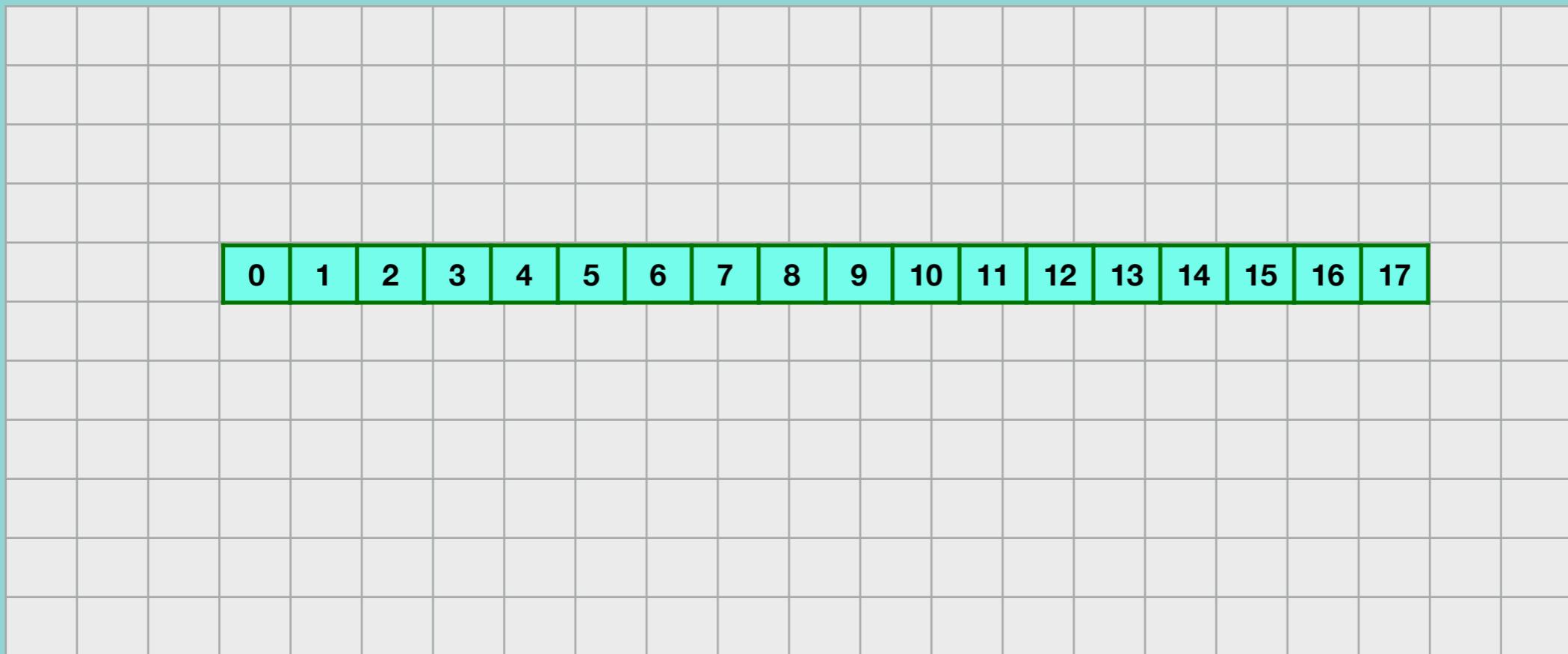


Arrays in Memory

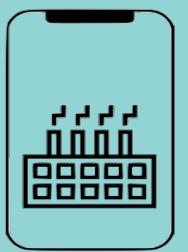
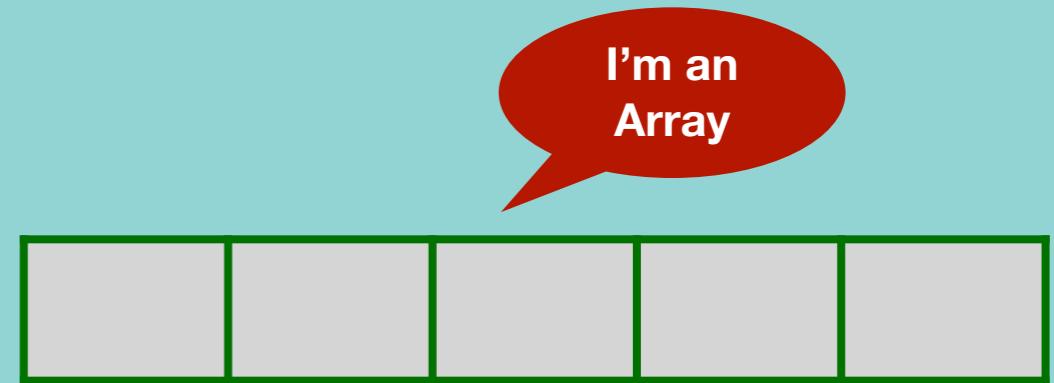
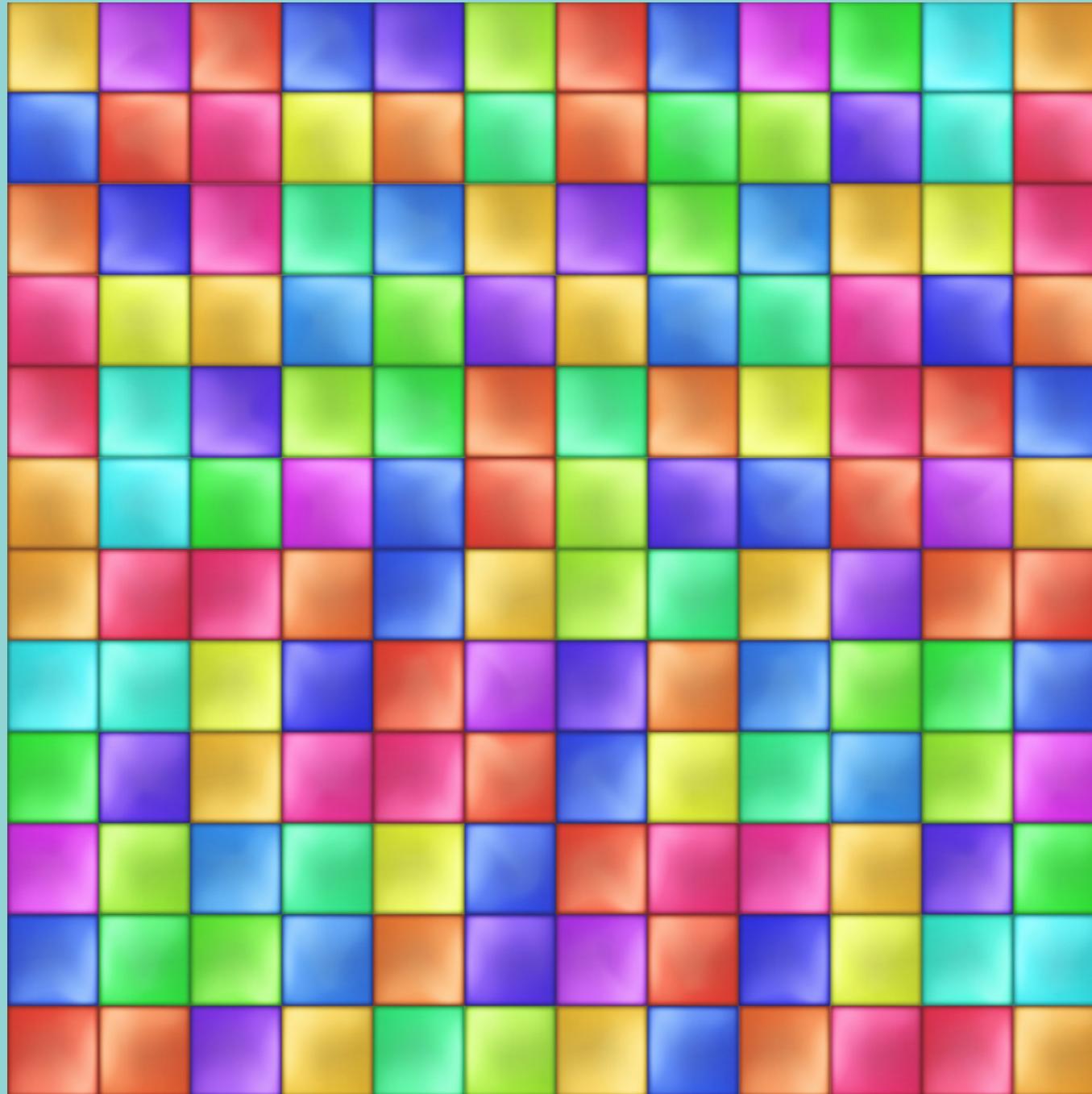
Three Dimensional array

```
[[[ 0,  1,  2],  
   [ 3,  4,  5]],  
  
 [[ 6,  7,  8],  
   [ 9, 10, 11]],  
  
 [[12, 13, 14],  
  [15, 16, 17]]]
```

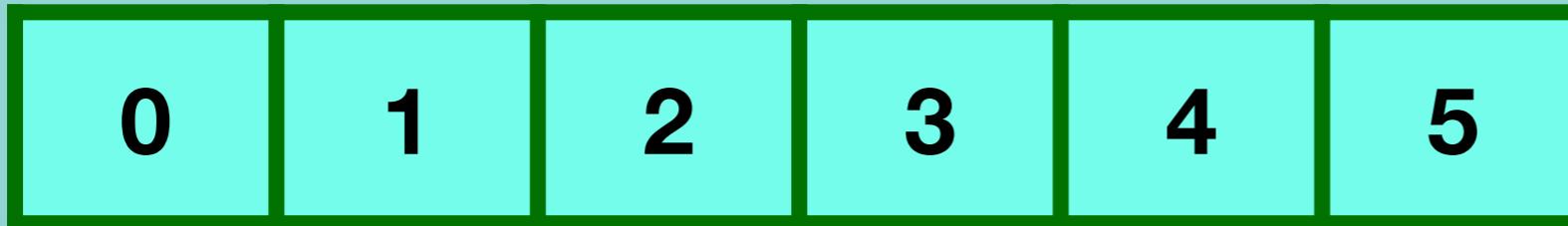
Memory



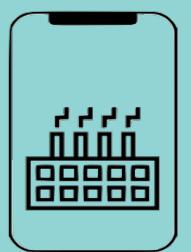
What is an Array?



What is an Array?



Not allowed

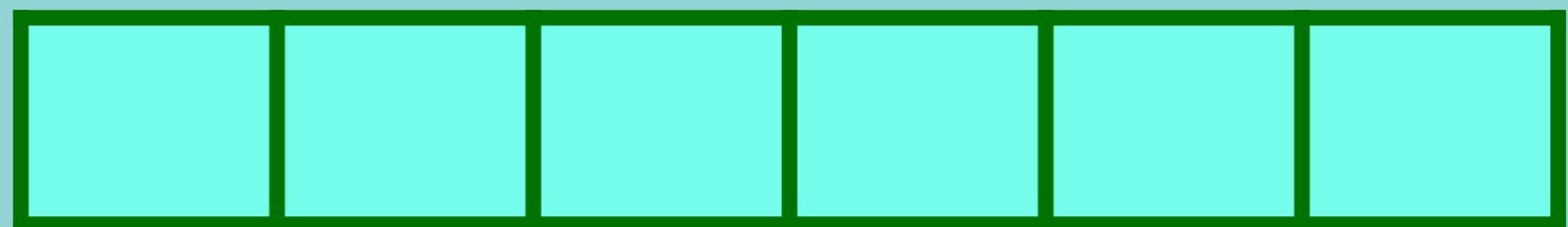


Creating an array

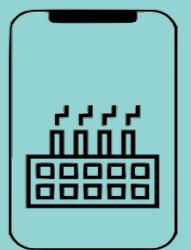
When we create an array , we:

- Assign it to a variable
- Define the type of elements that it will store
- Define its size (the maximum numbers of elements)

myArray =



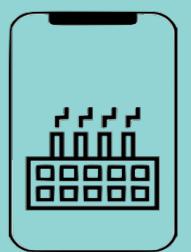
myArray =



Creating an array (Array Module)

```
from array import *
arrayName = array(typecode, [Initializers])
```

Type code	C Type	Python Type	Minimum size in bytes	Notes
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Unicode character	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	



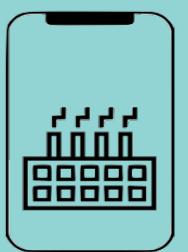
Insertion

myArray =

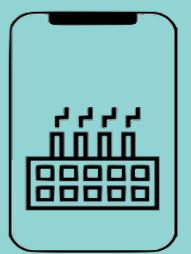
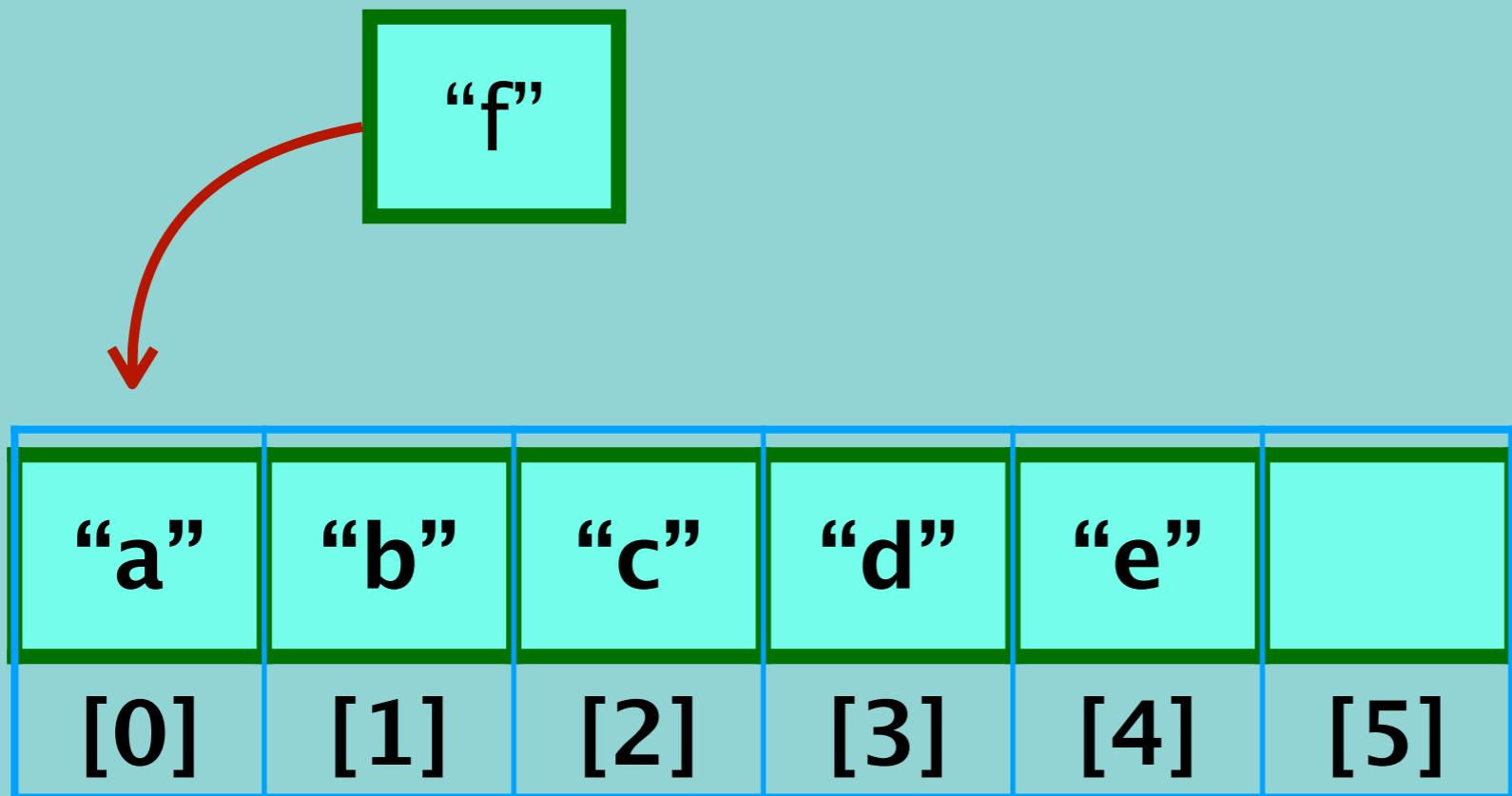
“a”	“b”	“c”			
[0]	[1]	[2]	[3]	[4]	[5]

myArray[3] = “d”

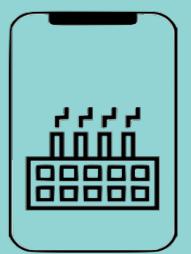
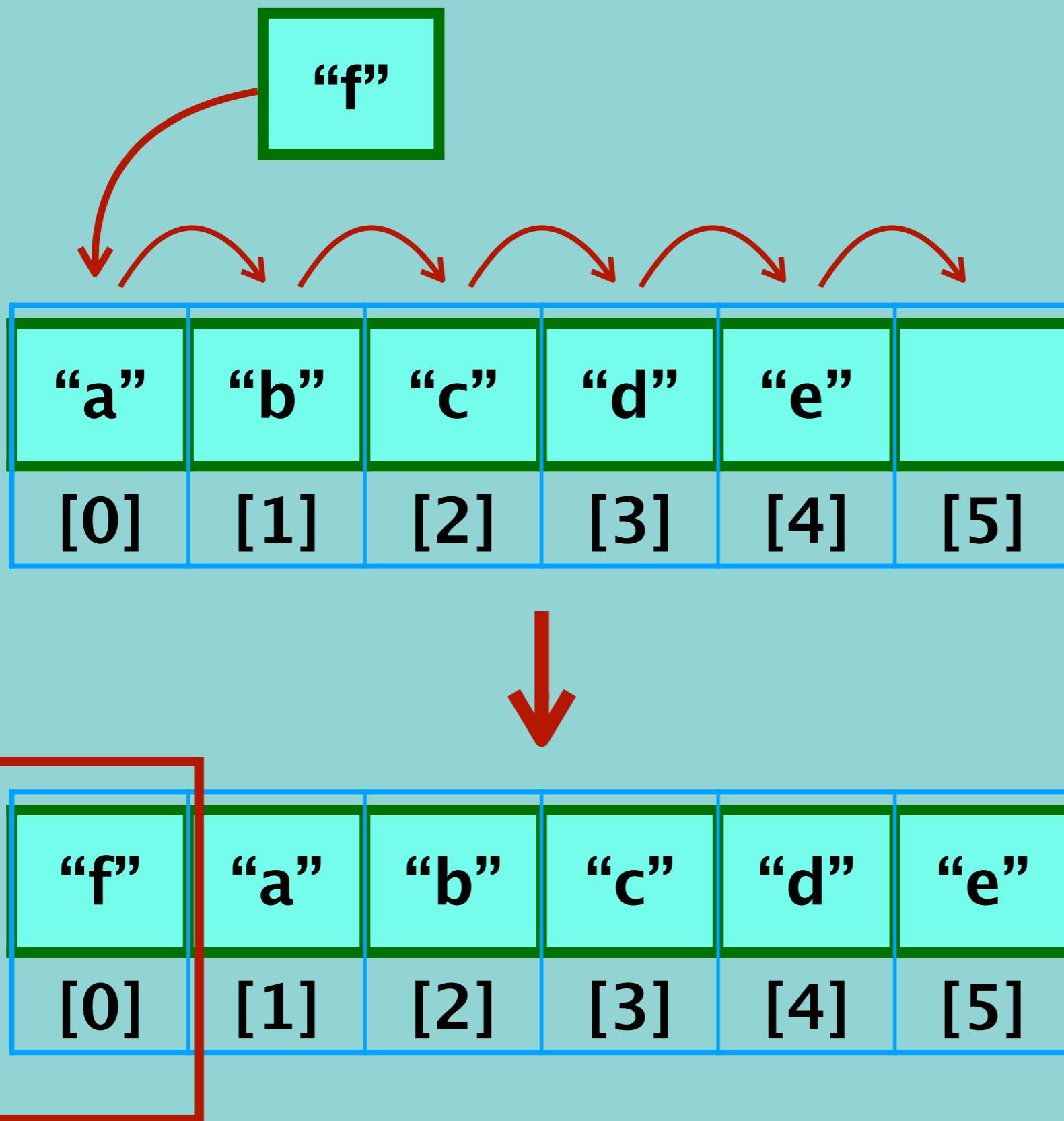
myArray[5] = “f”



Insertion

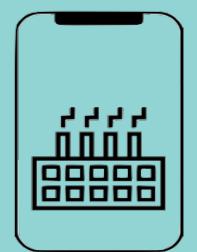
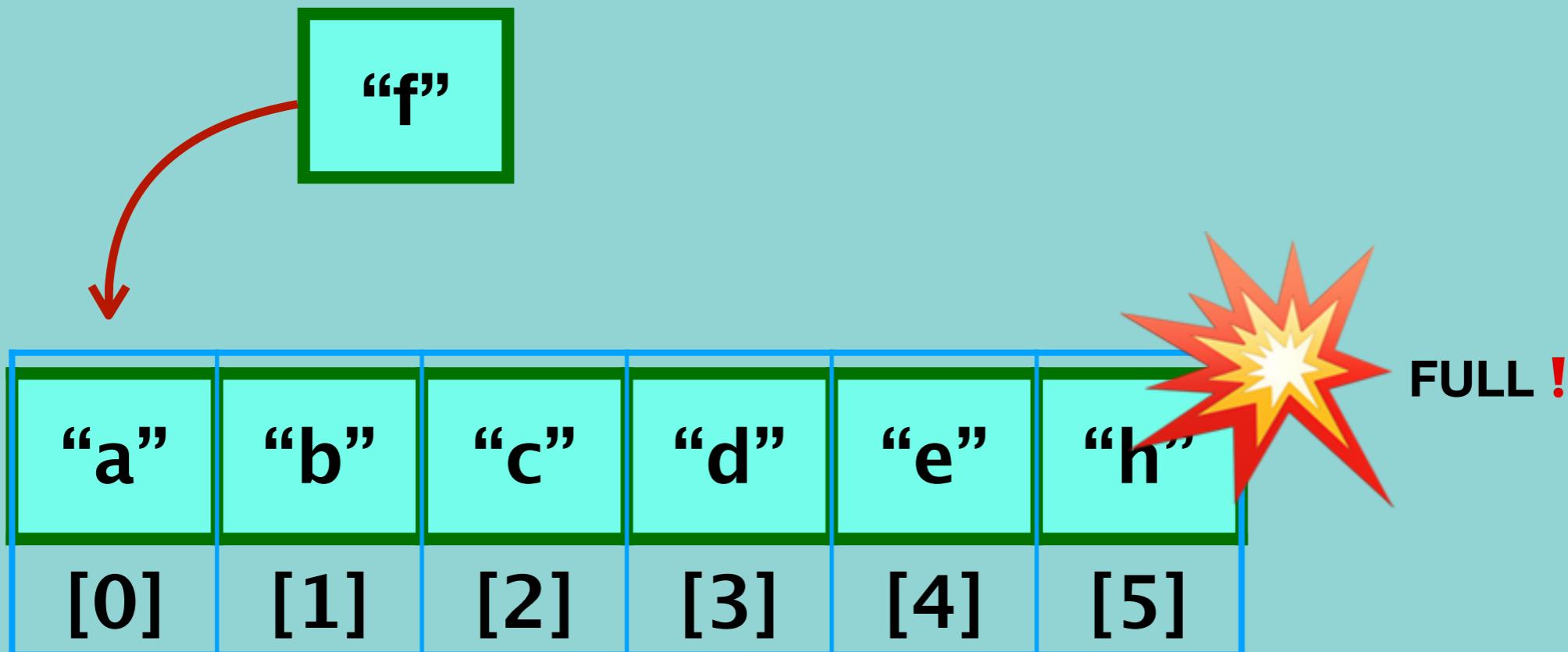


Insertion





Wait A minute! What Happens if the Array is Full?

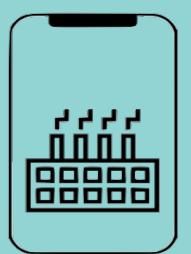


Insertion , when an array is full.

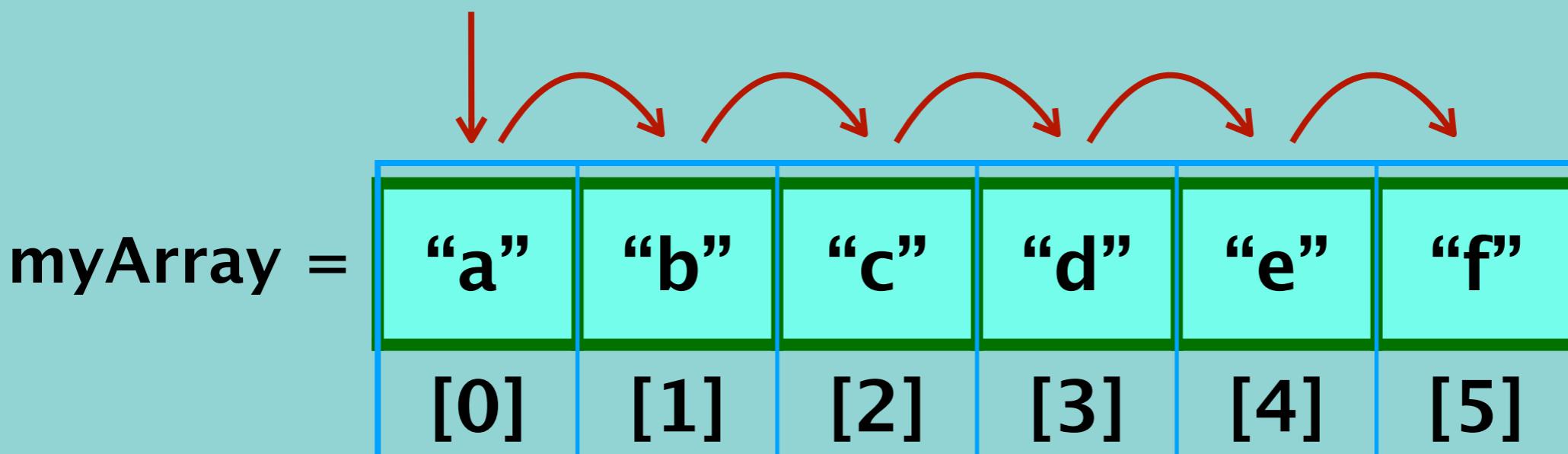
“a”	“b”	“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]



“a”	“b”	“c”	“d”	“e”	“h”						
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



Array traversal



myArray[0] = "a"

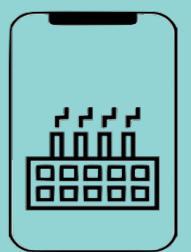
myArray[1] = "b"

myArray[2] = "c"

myArray[3] = "d"

myArray[4] = "e"

myArray[5] = "f"

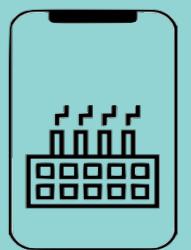


Array traversal

```
def traverseArray(array):
    for i in array: -----> O(n)
        print(i)-----> O(1) }-----> O(n)
```

Time Complexity : $O(n)$

Space Complexity : $O(1)$



Access an element of Two Dimensional Array

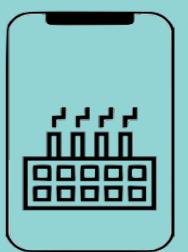
For example:

`myArray =`

“a”	“b”	“c”	“d”	“e”	“f”
[0]	[1]	[2]	[3]	[4]	[5]

`myArray[0] = “a”`

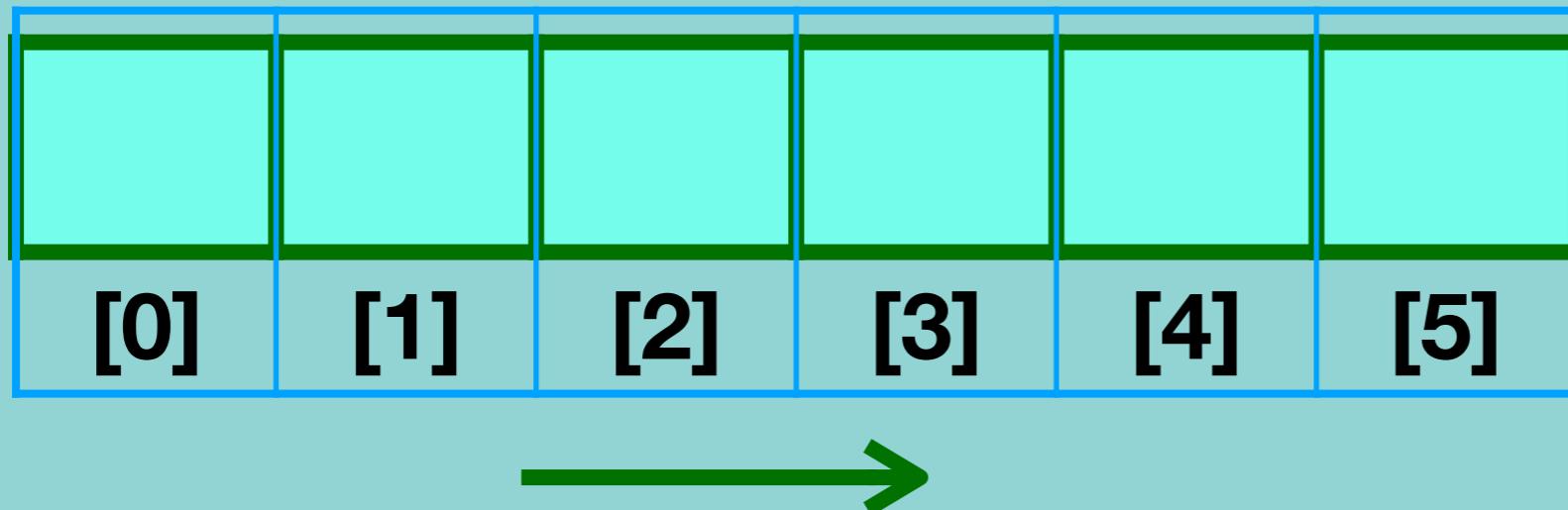
`myArray[3] = “d”`



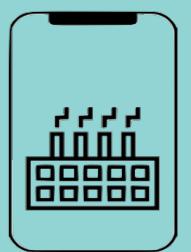
Access array element

How can we tell the computer which particular value we would like to access?

INDEX



<arrayName>[index]

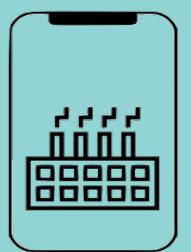


Access an element of array

```
def accessElement(array, index):
    if index >= len(array):-----> O(1)
        print('There is not any element in this index') -----> O(1)
    else:
        print(array[index])-----> O(1)
```

Time Complexity : O(1)

Space Complexity : O(1)

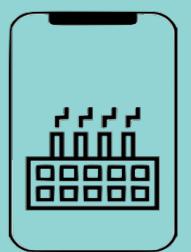
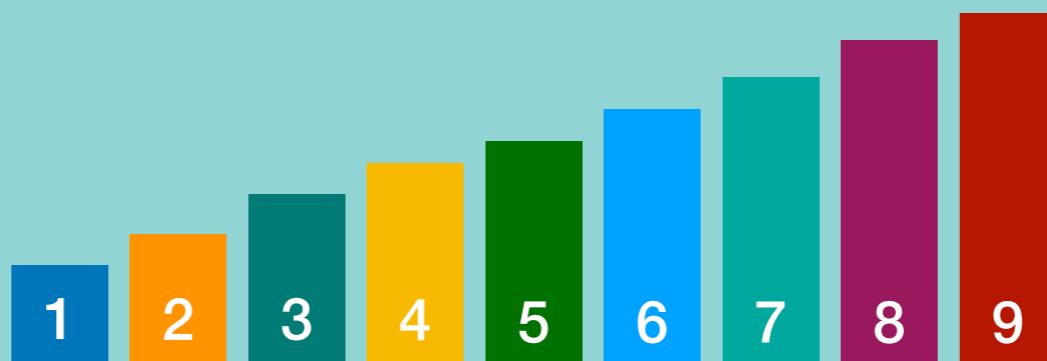


Finding an element

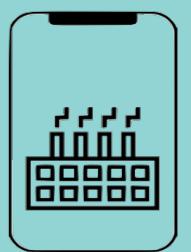
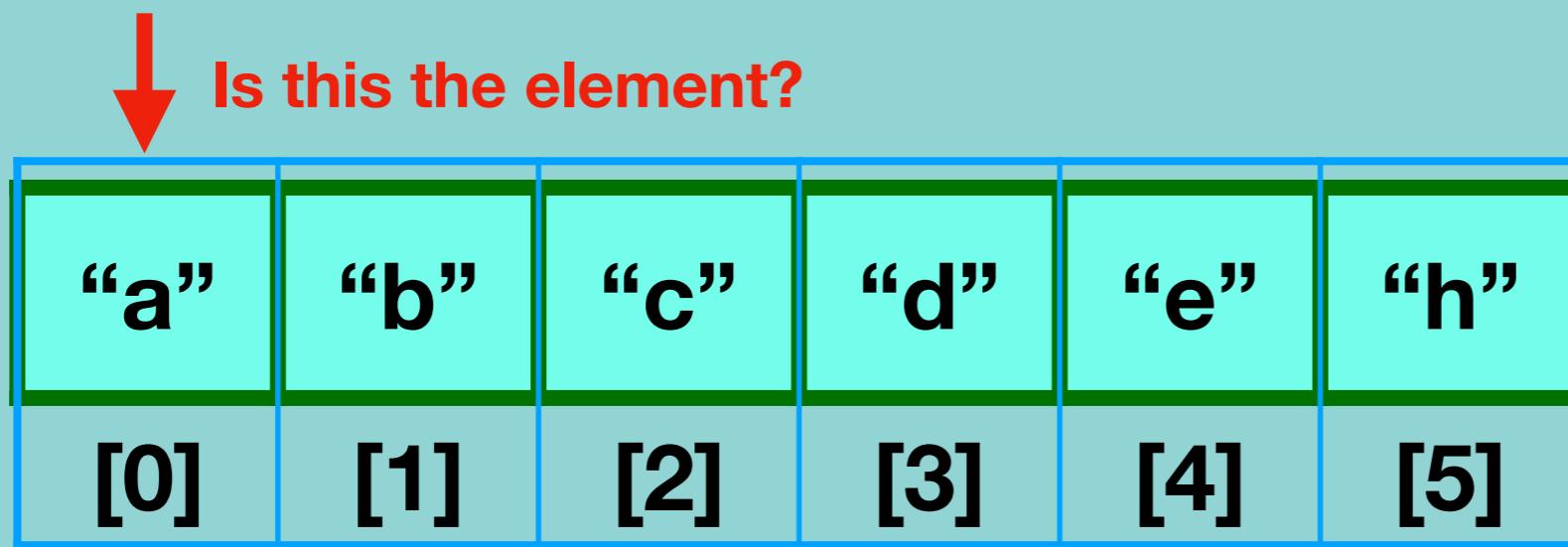
myArray =

“a”	“b”	“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]

myArray[2]



Finding an element



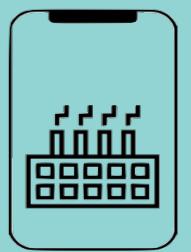
Finding an element

```
def searchInArray(array, value):
    for i in array: -----> O(n)
        if i == value: -----> O(1)
            return arr.index(value)-----> O(n)
    return "The element does not exist in this array" -----> O(1)
```

Time Complexity : $O(n)$

Space Complexity : $O(1)$

```
def searchInArray(array, value):
    for i in array:-----> O(n)
        if i == value: -----> O(1)
            return True -----> O(1)
    return "The element does not exist in this array"-----> O(1)
```

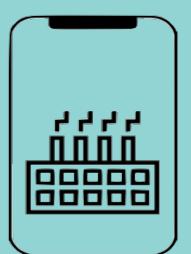


Deletion

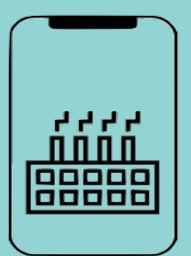
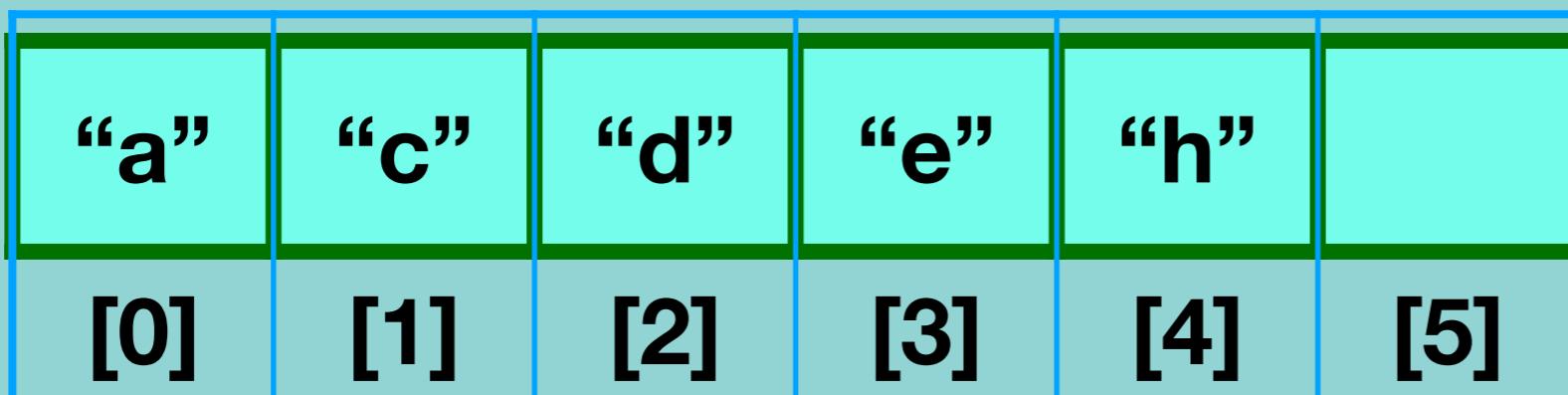
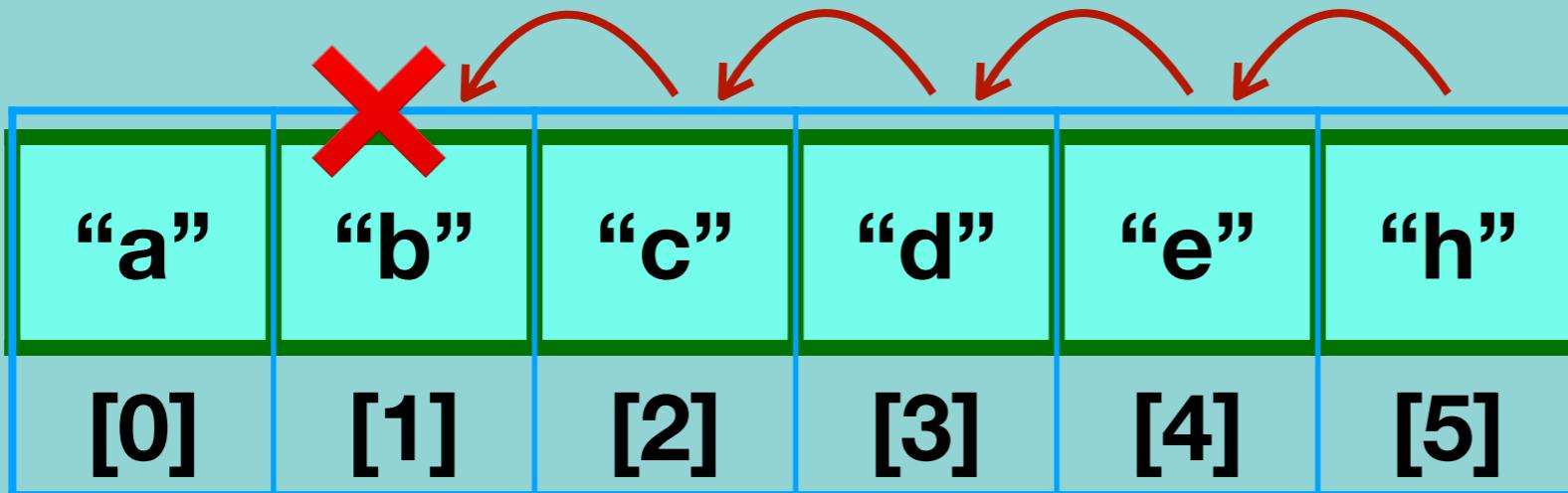
“a”	“b”	“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]

Not allowed!!

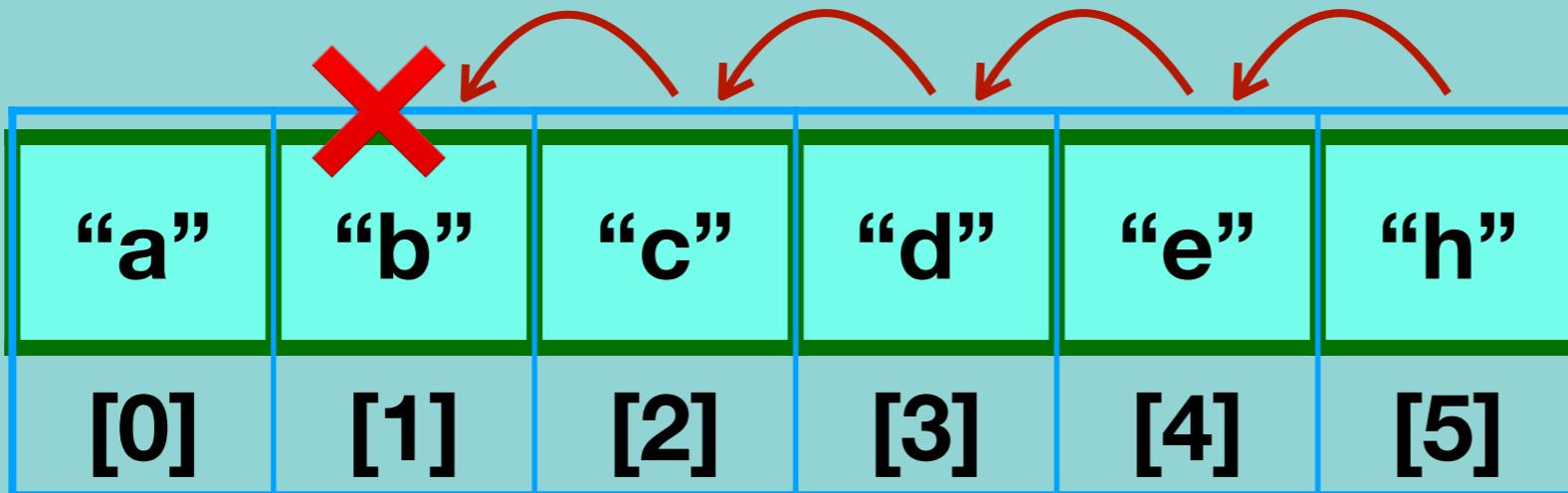
“a”		“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]



Deletion

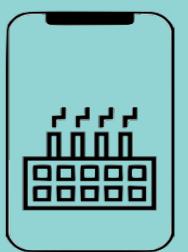


Deletion

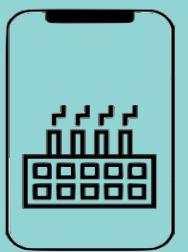


Time Complexity : $O(n)$

Space Complexity : $O(1)$



Inserting a value to two dimensional array



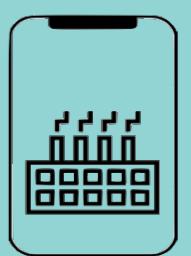
Array type codes in Python

Type code	C Type	Python Type	Minimum size in bytes	Notes
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Unicode character	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

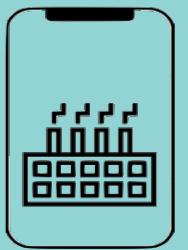
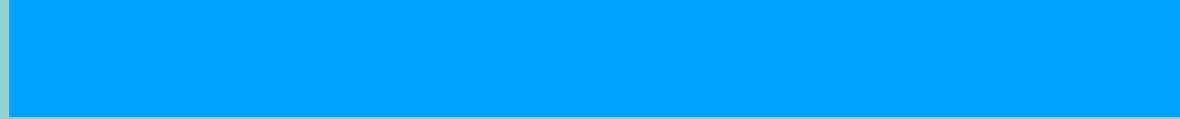
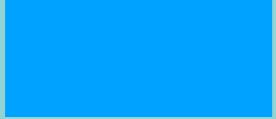
The 'u' type code corresponds to Python's obsolete unicode character ([Py_UNICODE](#) which is `wchar_t`). Depending on the platform, it can be 16 bits or 32 bits.

'u' will be removed together with the rest of the [Py_UNICODE](#) API.

Deprecated since version 3.3, will be removed in version 4.0.

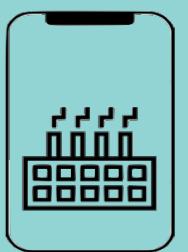


Built in functions of arrays in Python



Time and Space Complexity in One Dimensional Arrays

Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(n)
Inserting a value in an array	O(1)/O(n)	O(1)
Traversing a given array	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)/O(n)	O(1)



Two Dimensional Array

An array with a bunch of values having been declared with double index.

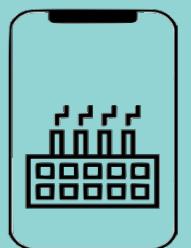
a[i][j] —> i and j between 0 and n

1	33	55	91	20	51	62	74	13
5	4	10	11	8	11	68	87	12
24	50	37	40	48	30	59	81	93

**Day 1 – 11, 15, 10, 6
Day 2 – 10, 14, 11, 5
Day 3 – 12, 17, 12, 8
Day 4 – 15, 18, 14, 9**

When we create an array , we:

- Assign it to a variable
- Define the type of elements that it will store
- Define its size (the maximum numbers of elements)



Insertion - Two Dimensional array

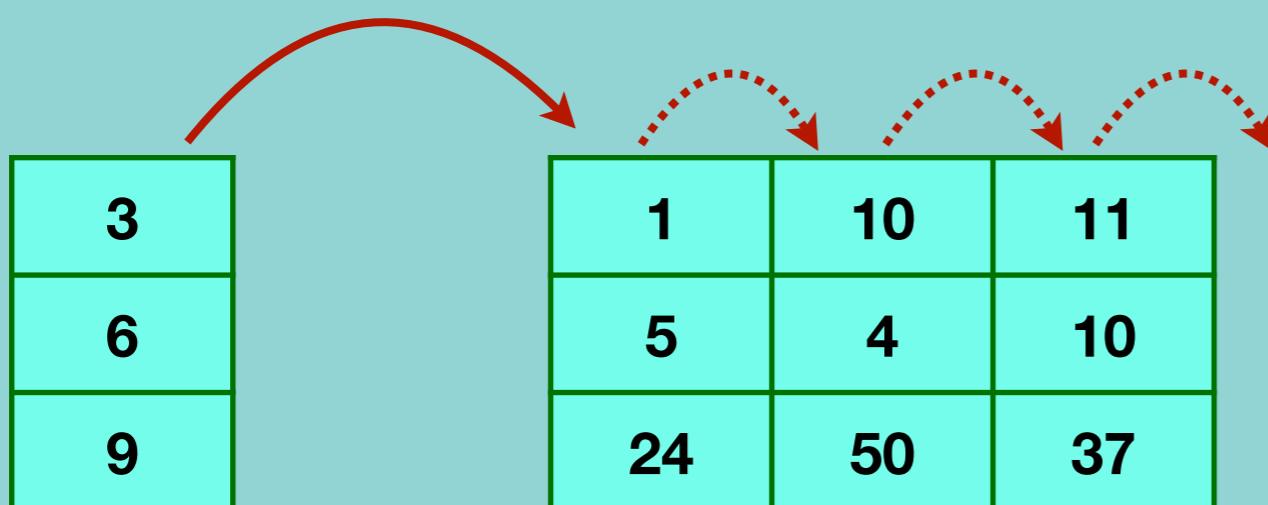
1	10	11
5	4	10
24	50	37

15

?

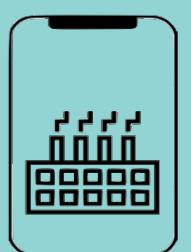
?

Adding Column



3	1	10	11
6	5	4	10
9	24	50	37

Time Complexity = $O(mn)$

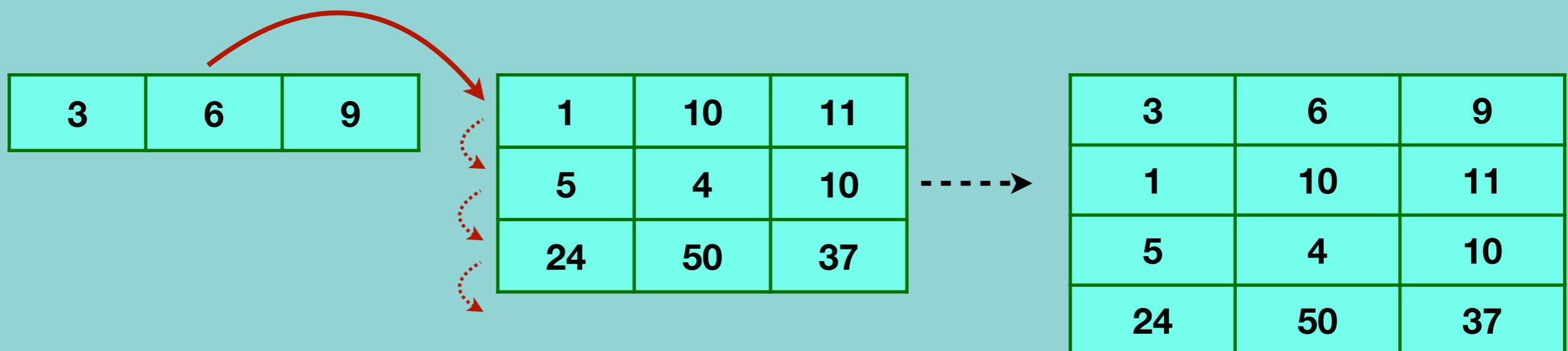


Insertion - Two Dimensional array

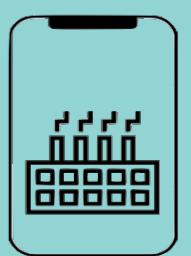
1	10	11
5	4	10
24	50	37

15
?
?

Adding Row



Time Complexity = $O(mn)$



Access an element of Two Dimensional Array

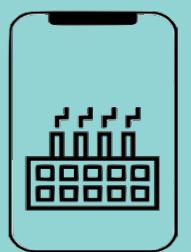
One dimensional array

5	4	10	11	8	11	68	87	12
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Two dimensional array

$a[i][j] \rightarrow i$ is row index and j is column index

[0]	[1]	[2]	[3]	[4]	a[0][4]	[5]	[6]	[7]	a[2][5]
[0]	1	33	55	91	20	51	62	74	13
[1]	5	4	10	11	8	11	68	87	12
[2]	24	50	37	40	48	30	59	81	93



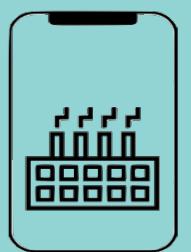
Traversing Two Dimensional Array

1	33	55	91
5	4	10	11
24	50	37	40

1 33 55 91

5 4 10 11

24 50 37 40



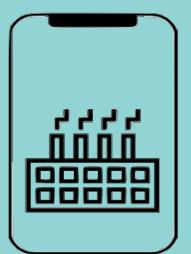
Traversing Two Dimensional Array



Is this the element?

1	33	55	91
5	4	44	11
24	50	37	40

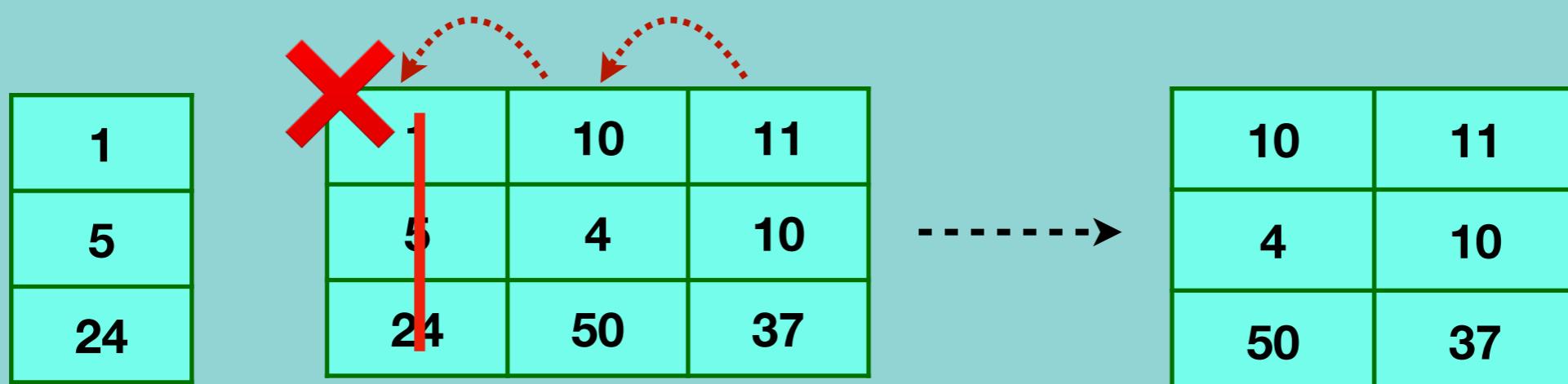
The element is found



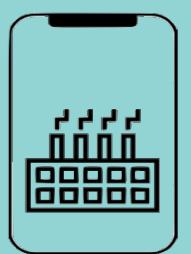
Deletion - Two Dimensional array

1	10	11
5	4	10
24	50	37

Deleting Column



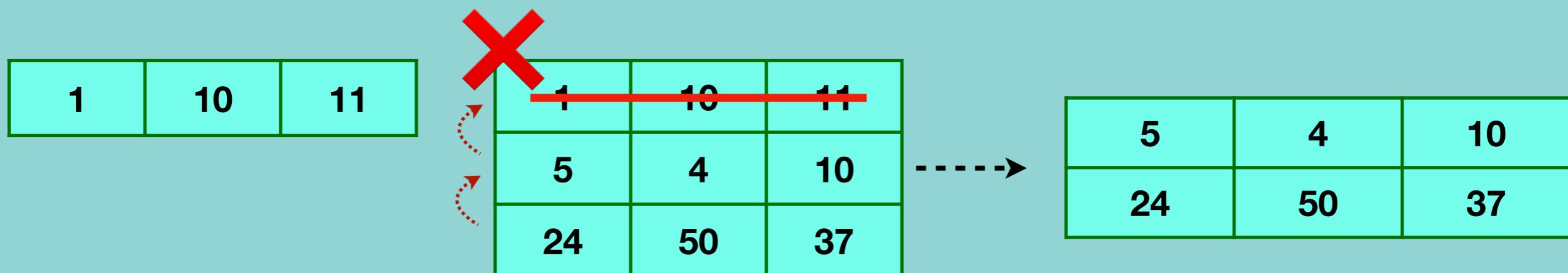
Time Complexity = $O(mn)$



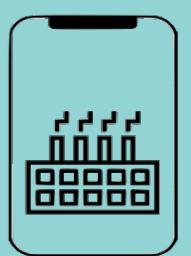
Deletion - Two Dimensional array

1	10	11
5	4	10
24	50	37

Deleting Row

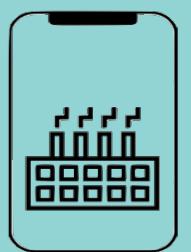


Time Complexity = $O(mn)$



Time and Space Complexity in 2D Arrays

Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(mn)
Inserting a column/row in an array	O(mn)/O(1)	O(1)
Traversing a given array	O(mn)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(mn)	O(1)
Deleting a given value	O(mn)/O(1)	O(1)



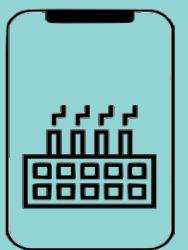
When to use/avoid Arrays

When to use

- To store multiple variables of same data type
- Random access

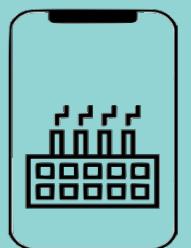
When to avoid

- Same data type elements
- Reserve memory



Summary

- **Arrays are extremely powerful data structures**
- **Memory is allocated immediately**
- **Elements are located in contiguous locations in memory**
- **Indices start at 0**
- **Inserting elements**
- **Removing elements.**
- **Finding elements**



Python Lists

A list is a data structure that holds an ordered collection of items.

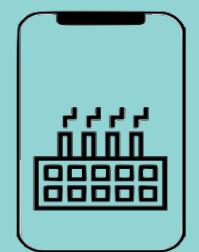


[10, 20, 30, 40] → Integers

['Edy', 'John', 'Jane'] → Strings

['spam', 1, 2.0, 5] → String, integer, float

['spam', 2.0, 5, [10, 20]] → String, integer, float, nested list



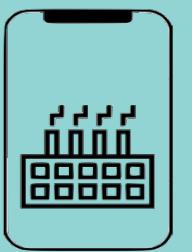
Arrays vs Lists

Similarities

[10, 20, 30, 40]

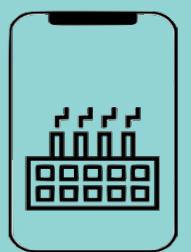
Differences

[10, 20, 30, 40]



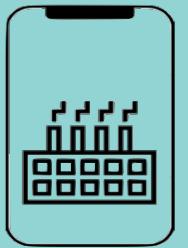
Time and Space Complexity in Python Lists

Operation	Time complexity	Space complexity
Creating a List	O(1)	O(n)
Inserting a value in a List	O(1)/O(n)	O(1)
Traversing a given List	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)/O(n)	O(1)



Array / List Project

AppMillers
www.appmillers.com



Find Number of Days Above Average Temperature

How many day's temperature?

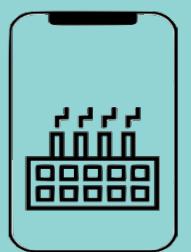
2

Day 1's high temp: 1

Day 2's high temp: 2

Output

Average = 1.5
1 day(s) above average



Array/List Interview Questions - 2

1. Two Sum

Easy 18064 647 Add to List Share

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

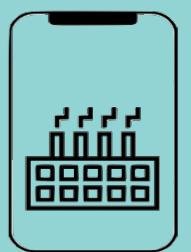
```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

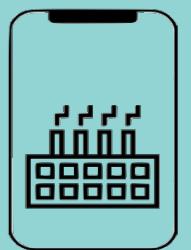


Array/List Interview Questions - 2

Write a program to find all pairs of integers whose sum is equal to a given number.

[2, 6, 3, 9, 11] 9 → [6,3]

- Does array contain only positive or negative numbers?
- What if the same pair repeats twice, should we print it every time?
- If the reverse of the pair is acceptable e.g. can we print both (4,1) and (1,4) if the given sum is 5.
- Do we need to print only distinct pairs? does (3, 3) is a valid pair for given sum of 6?
- How big is the array?

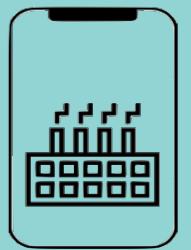


Array/List Interview Questions -7

Rotate Matrix – Given an image represented by an NxN matrix write a method to rotate the image by 90 degrees.

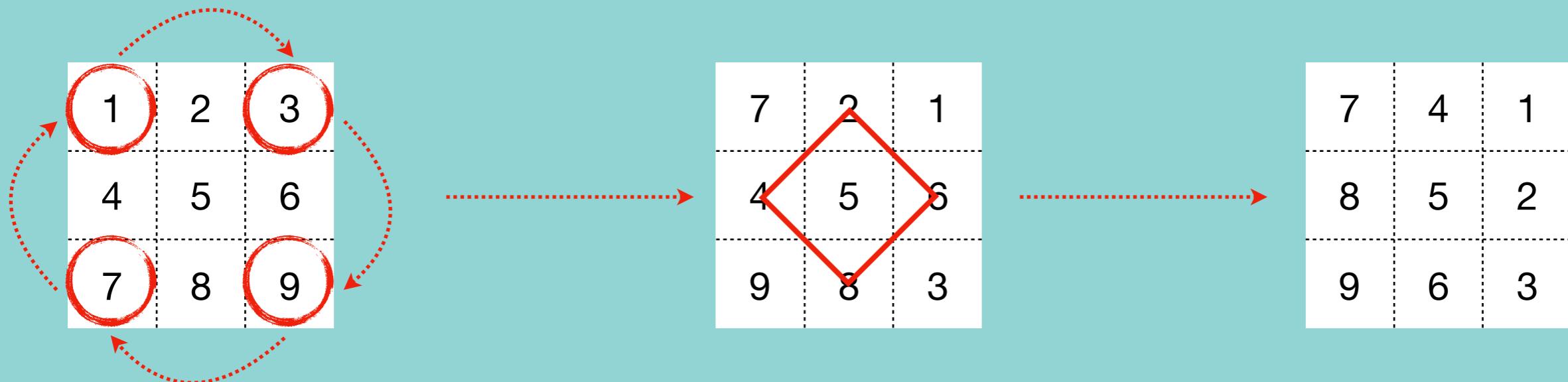
1	2	3
4	5	6
7	8	9

Rotate 90 degrees 

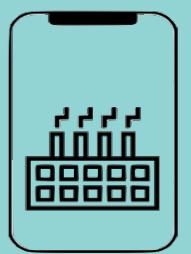


Array/List Interview Questions -7

Rotate Matrix – Given an image represented by an NxN matrix write a method to rotate the image by 90 degrees.



```
for i = 0 to n  
    temp = top[i]  
    top[i] = left[i]  
    left[i] = bottom[i]  
    bottom[i] = right[i]  
    right[i] = temp
```



Dictionaries

A dictionary is a collection which is unordered, changeable and indexed.

Miller : a person who owns or works in a corn mill

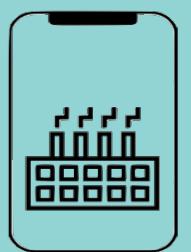


```
myDict = {"Miller" : "a person who owns or works in a corn mill",
          "Programmer" : "a person who writes computer programs"}
```

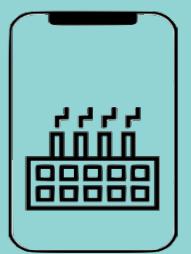
	0	1	2
myArray =	"Miller"	"Programmer"	"App Miller"

myArray[0] → Miller

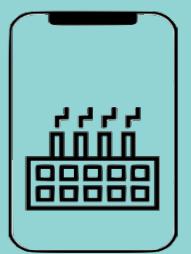
myDict["Miller"] → a person who owns or works in a corn mill



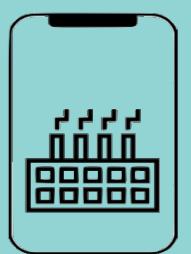
Dictionaries



Dictionaries

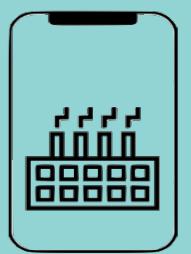
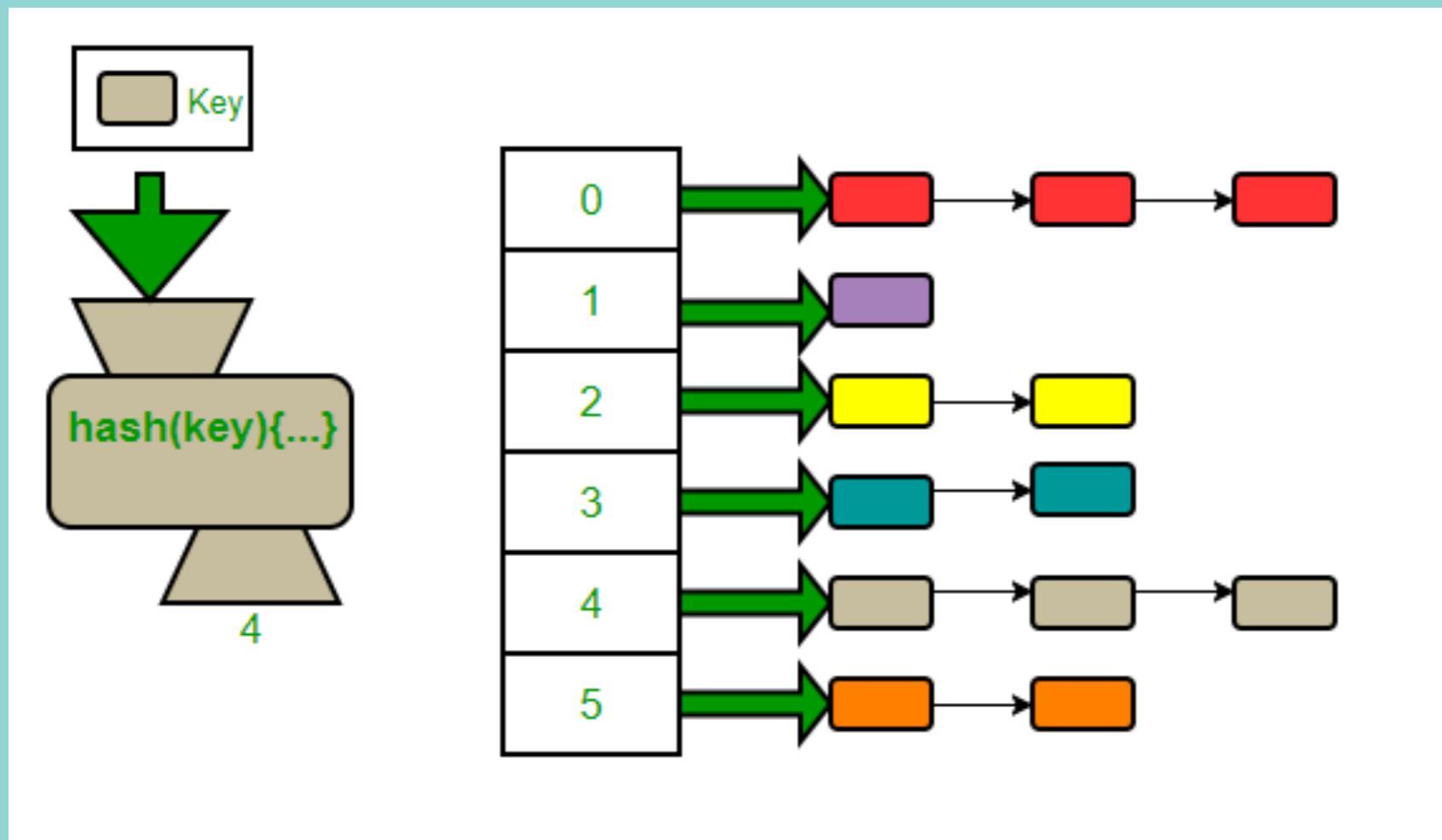


Dictionaries



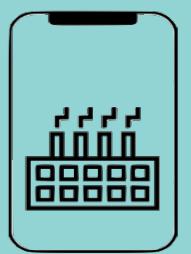
Dictionary in Memory

A **hash table** is a way of doing **key-value lookups**. You store the values in an array, and then use a **hash function** to find the index of the array cell that corresponds to your key-value pair.



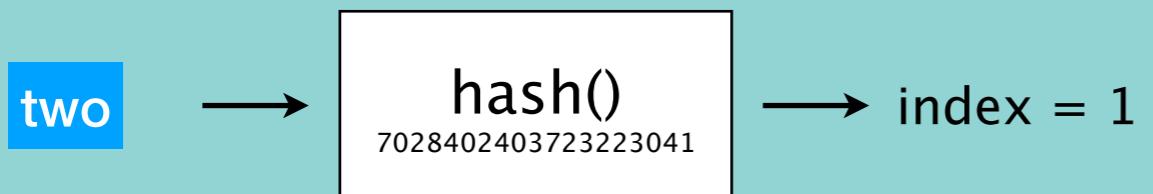
Dictionary in Memory

```
engToSp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

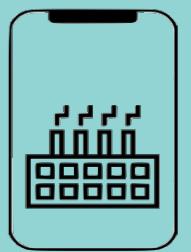


Dictionary in Memory

```
engToSp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

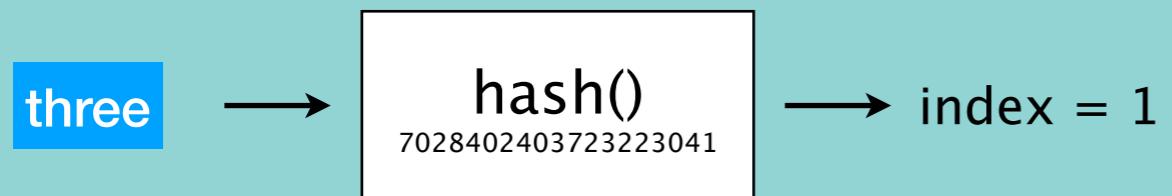


0	
1	two dos
2	
3	one uno
4	

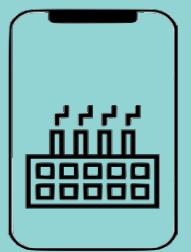


Dictionary in Memory

```
engToSp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

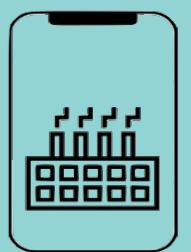
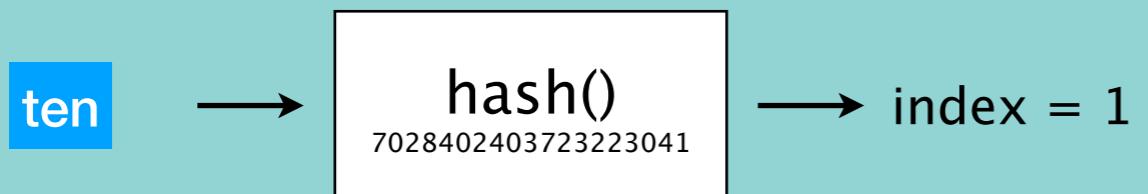


0		
1	two	dos
2		
3	one	uno
4	three	tres



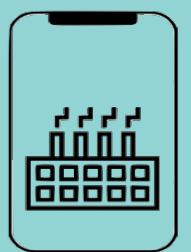
Dictionary in Memory

```
engToSp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```



Dictionary all method

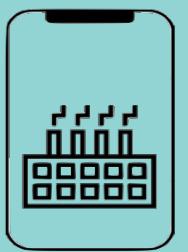
Cases	Return Value
All values are true	TRUE
All values are false	FALSE
One value is true (others are false)	FALSE
One value is false (others are true)	FALSE
Empty Iterable	TRUE



Dictionary any method

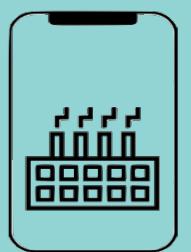
Cases	Return Value
All values are true	TRUE
All values are false	FALSE
One value is true (others are false)	TRUE
One value is false (others are true)	TRUE
Empty Iterable	FALSE

•



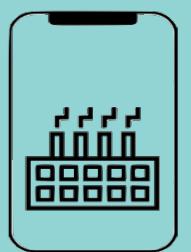
Dictionary vs List

Dictionary	List
Unordered	Ordered
Access via keys	Access via index
Collection of key value pairs	Collection of elements
Preferred when you have unique key values	Preferred when you have ordered data
No duplicate members	Allow duplicate members



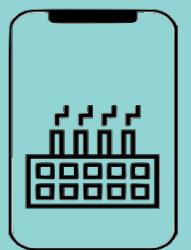
Time and Space Complexity in Python Dictionary

Operation	Time complexity	Space complexity
Creating a Dictionary	$O(\text{len(dict)})$	$O(n)$
Inserting a value in a Dictionary	$O(1)/O(n)$	$O(1)$
Traversing a given Dictionary	$O(n)$	$O(1)$
Accessing a given cell	$O(1)$	$O(1)$
Searching a given value	$O(n)$	$O(1)$
Deleting a given value	$O(1)$	$O(1)$



Tuples

- **What is a tuple? How can we create it ?**
- **Tuples in Memory**
- **Accessing an element of Tuple**
- **Traversing / Slicing a Tuple**
- **Search for an element in Tuple**
- **Tuple Operations/Functions**
- **Tuple vs List**
- **Tuple vs Dictionary**
- **Time and Space complexity of Tuples**



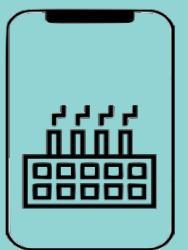
What is a Tuple?

A tuple is an immutable sequence of Python objects

Tuples are also comparable and hashable

```
t = 'a', 'b', 'c', 'd', 'e'
```

```
t = ('a', 'b', 'c', 'd', 'e')
```

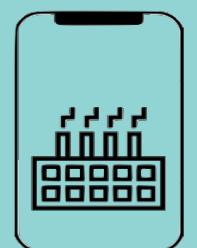
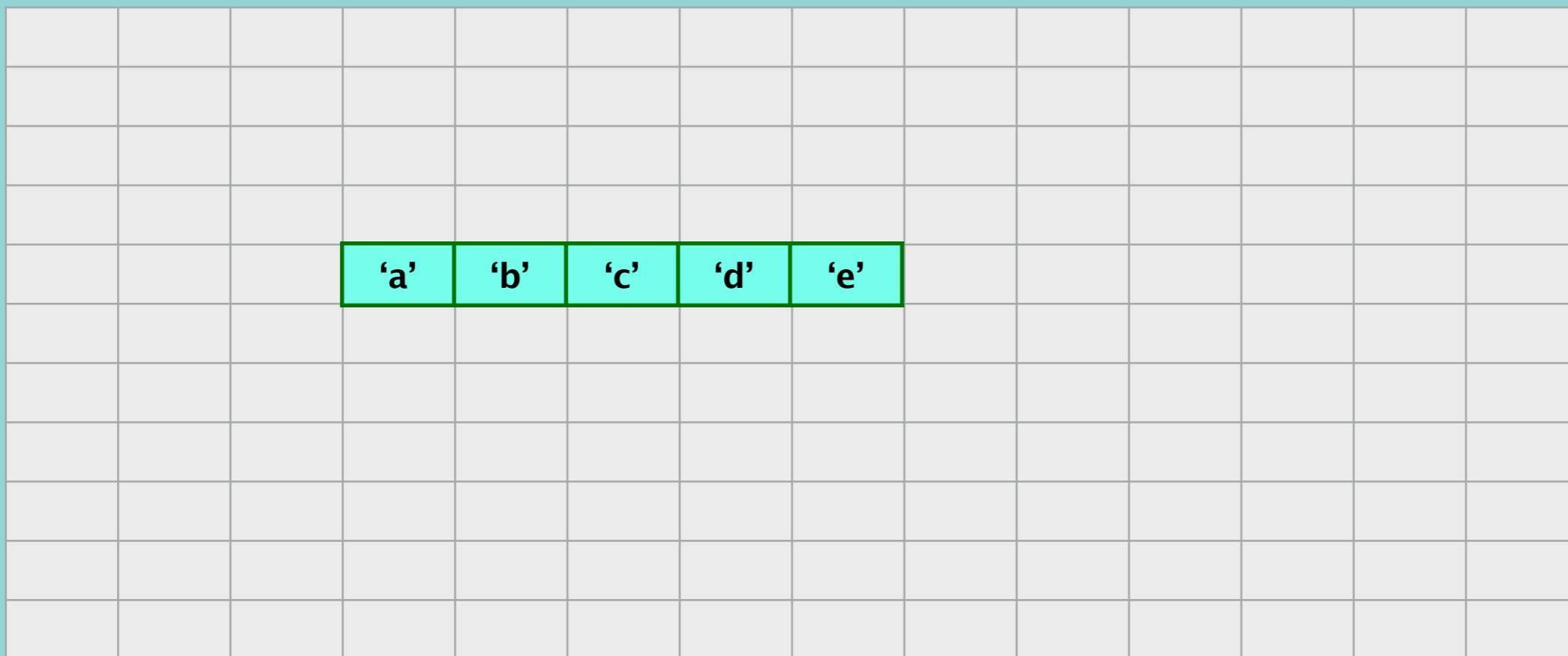


Tuples in Memory

Sample Tuple

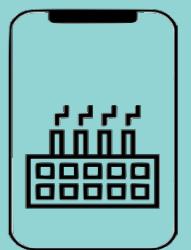
'a'	'b'	'c'	'd'	'e'
-----	-----	-----	-----	-----

Memory



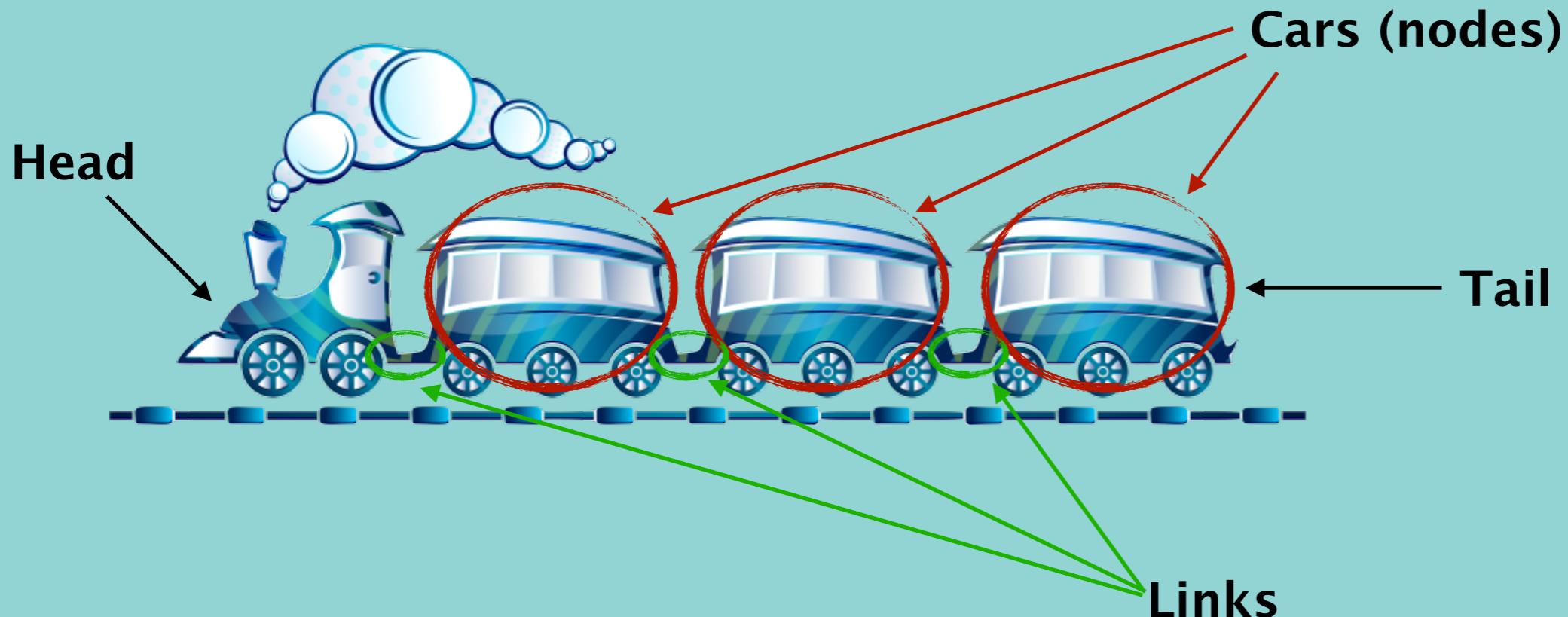
Time and Space Complexity in Python Tuples

Operation	Time complexity	Space complexity
Creating a Tuple	O(1)	O(n)
Traversing a given Tuple	O(n)	O(1)
Accessing a given element	O(1)	O(1)
Searching a given element	O(n)	O(1)

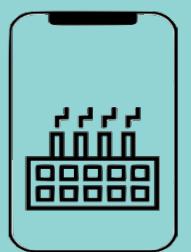


What is a Linked List?

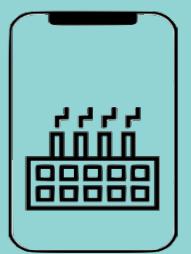
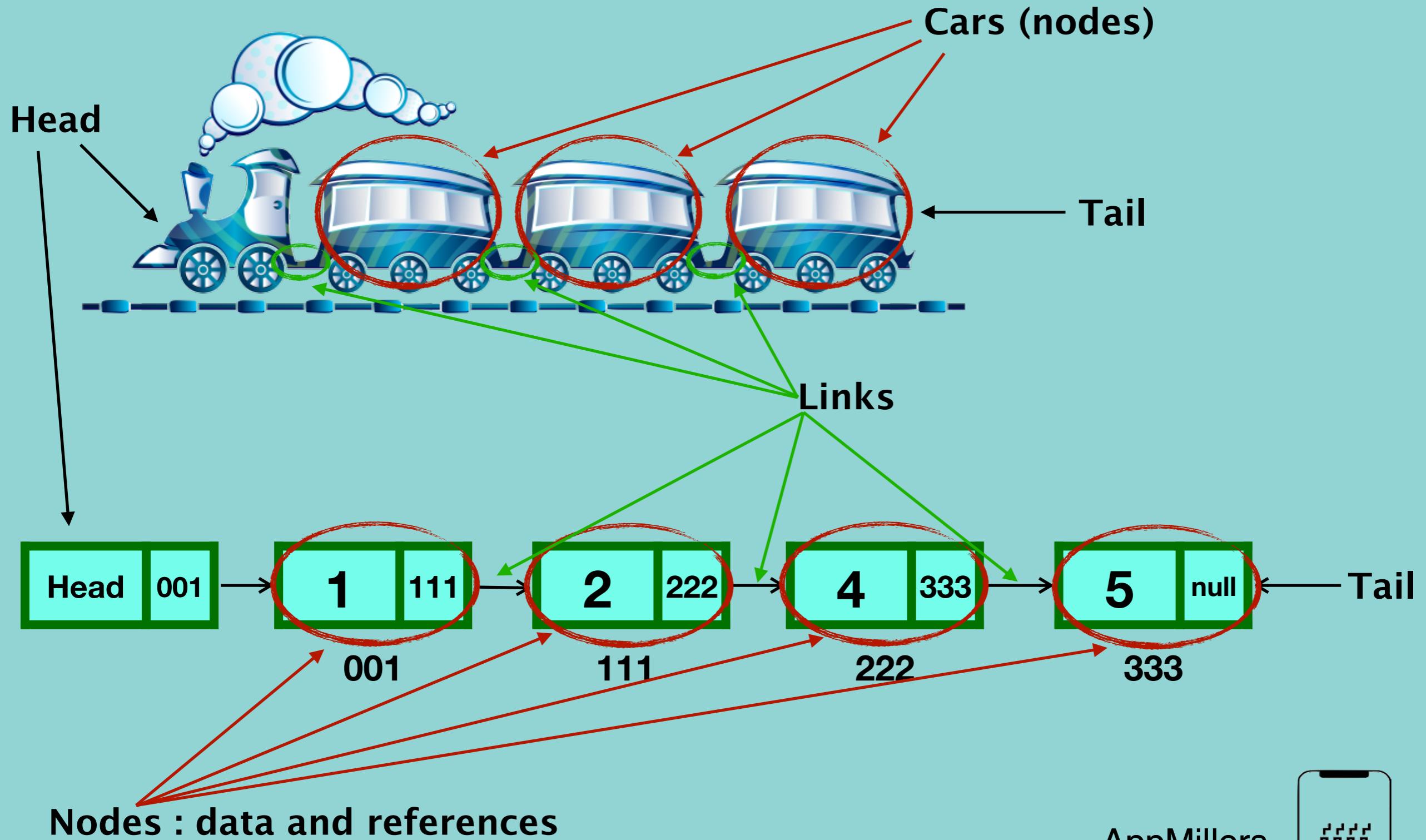
Linked List is a form of a sequential collection and it does not have to be in order. A Linked list is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link.



- Each car is independent
- Cars : passengers and links (nodes: data and links)

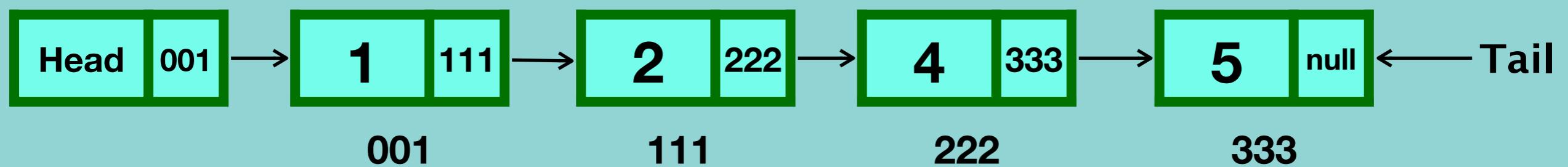


What is a Linked List?



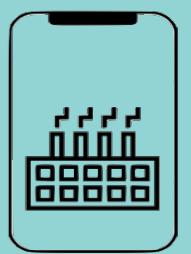
What is a Linked List?

Linked List is a form of a sequential collection and It does not have to be in order. A Linked list is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link.

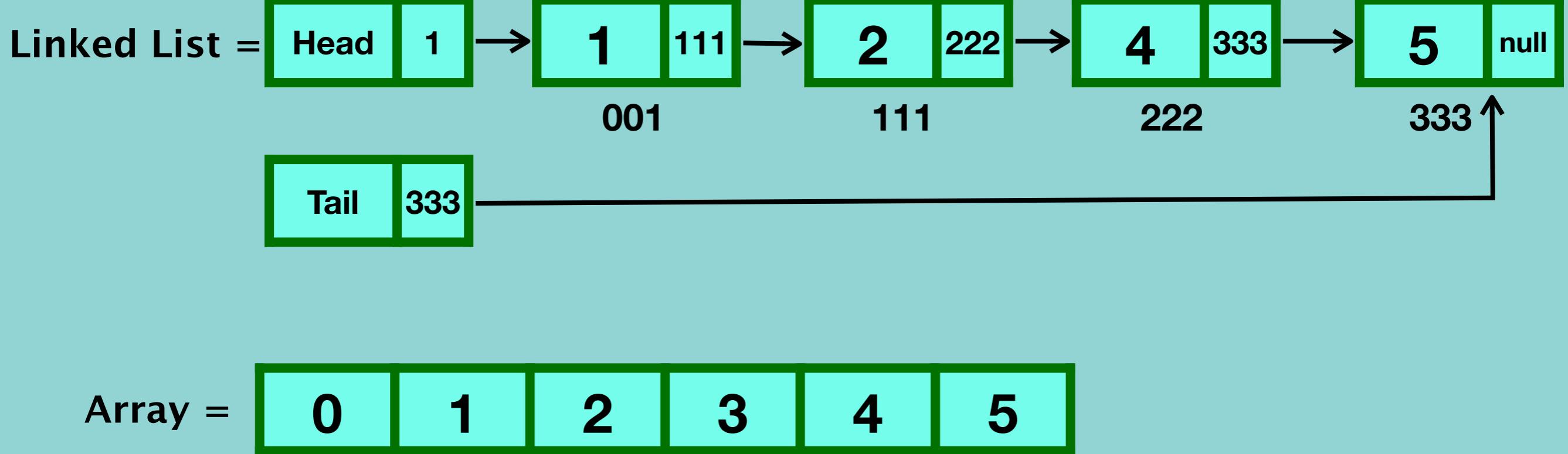


Nodes

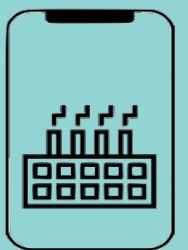
Pointers



Linked Lists vs Arrays



- Elements of Linked list are independent objects
- Variable size – the size of a linked list is not predefined
- Insertion and removals in Linked List are very efficient.
- Random access – accessing an element is very efficient in arrays

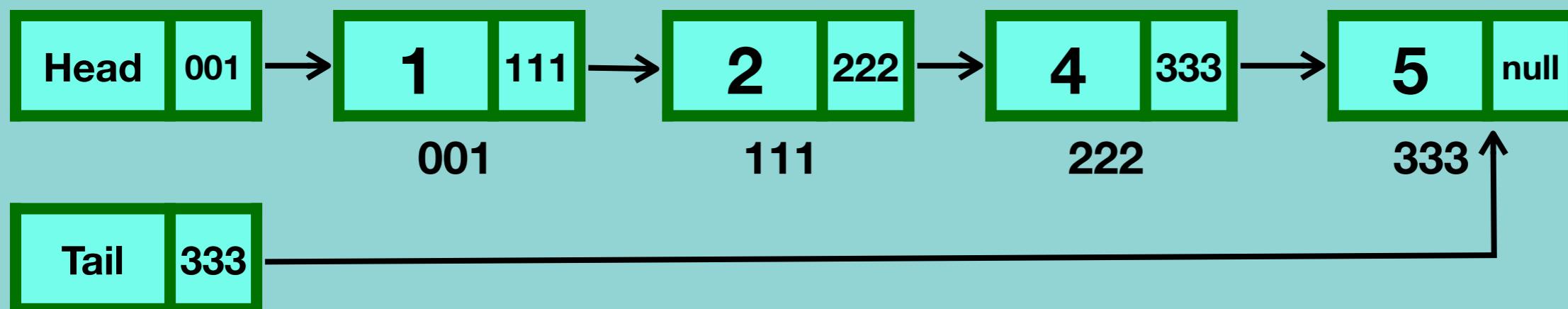


Types of Linked Lists

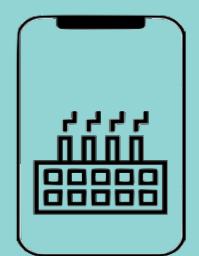
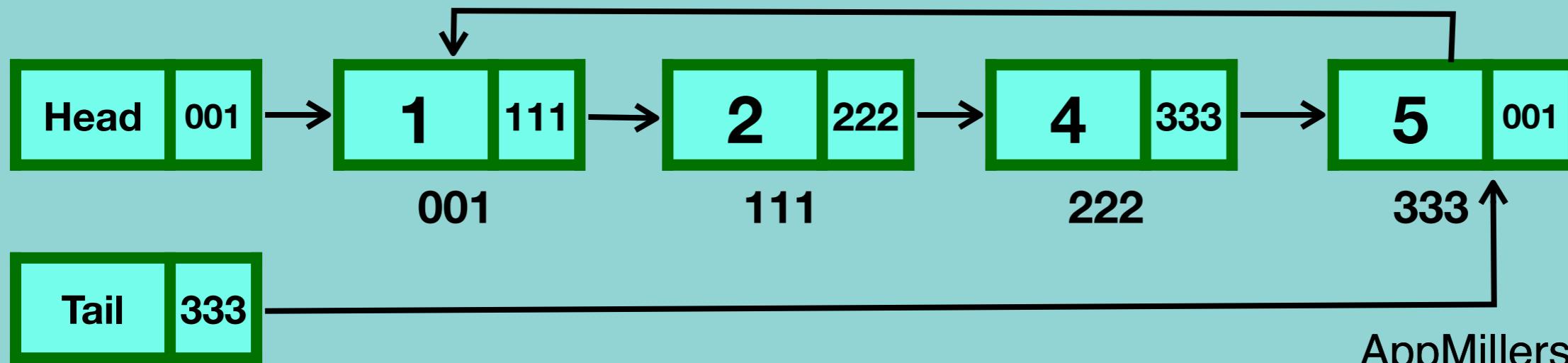
- Singly Linked List
- Circular Singly Linked List

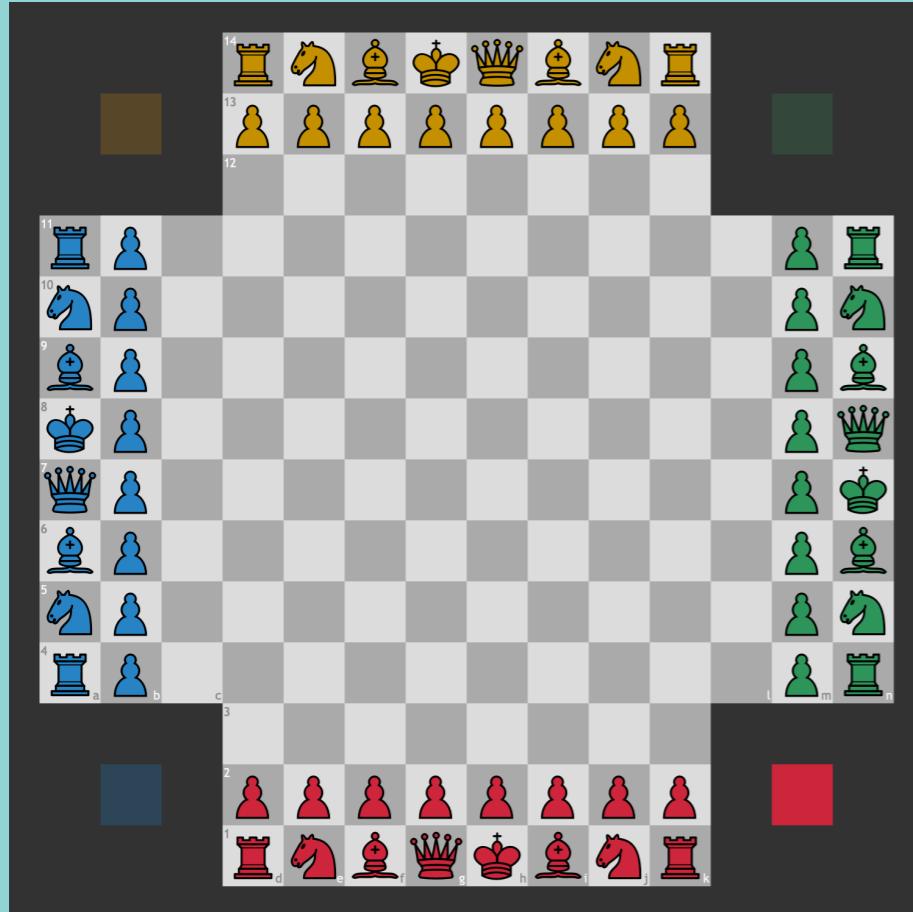
Singly Linked List

- Circular Doubly Linked List

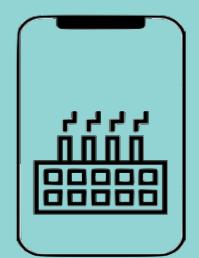
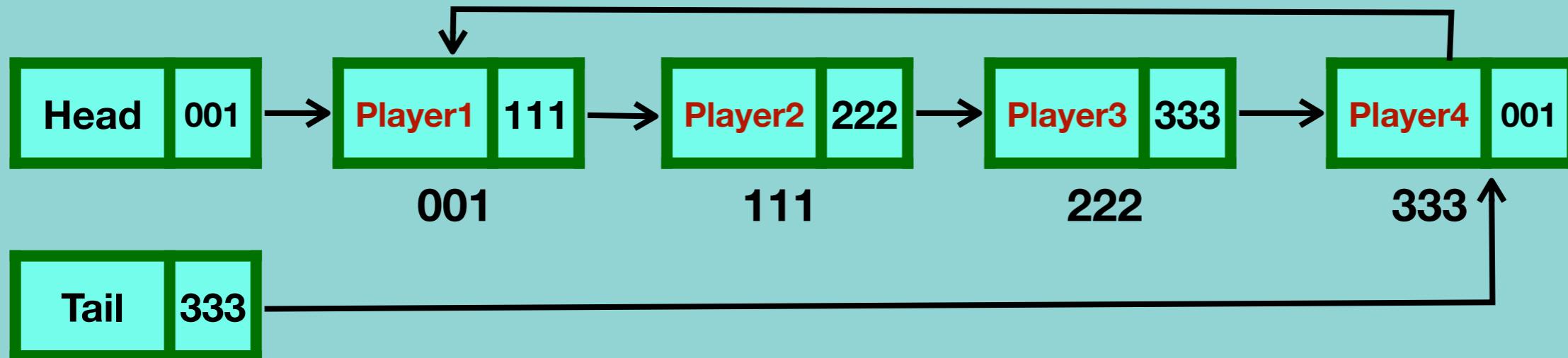


Circular Singly Linked List



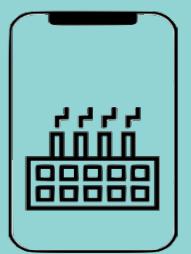
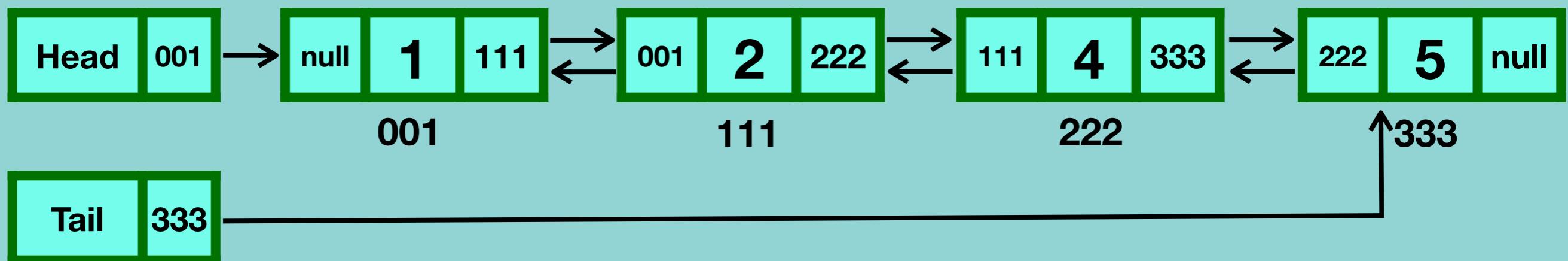


Circular Singly Linked List

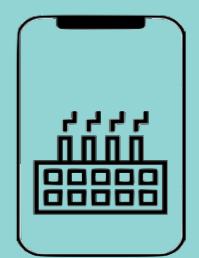
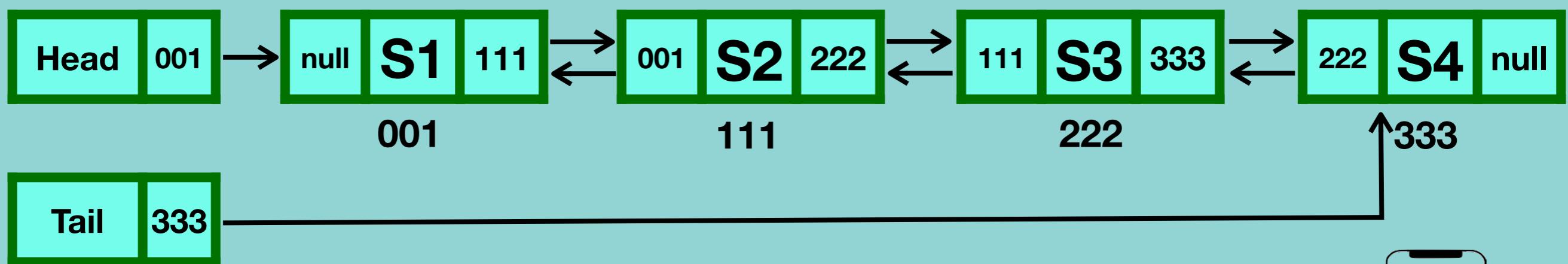
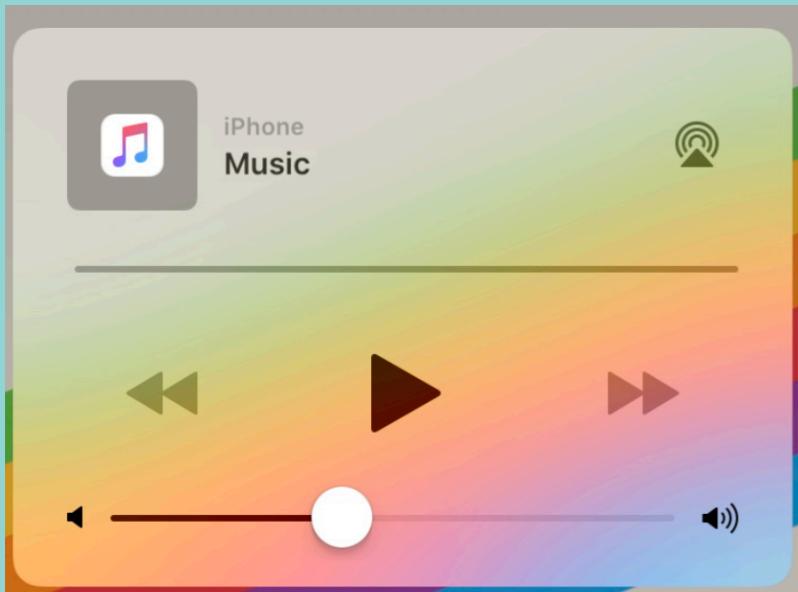


Types of Linked Lists

Doubly Linked List

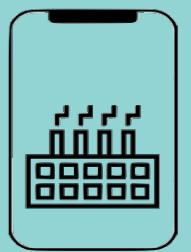
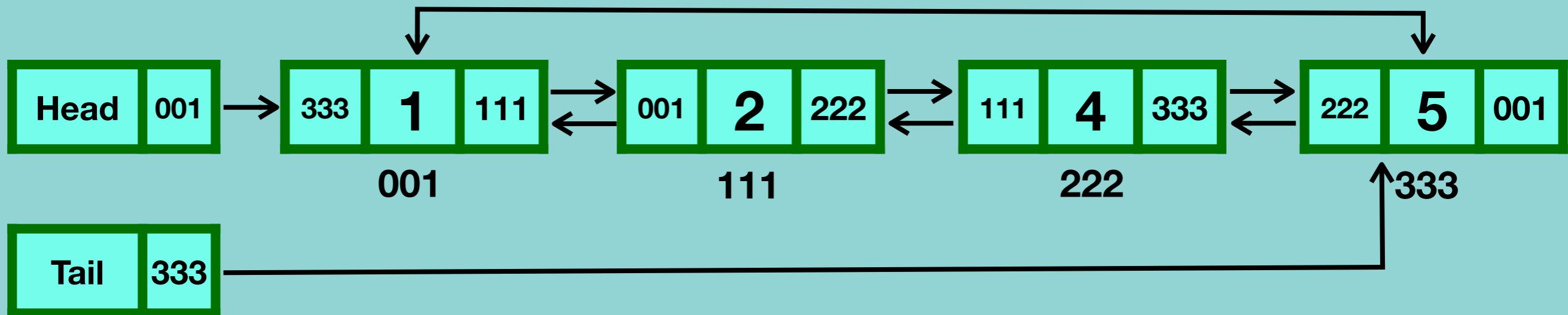


Types of Linked Lists



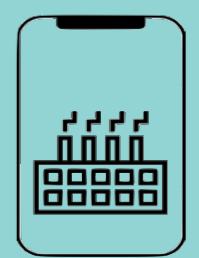
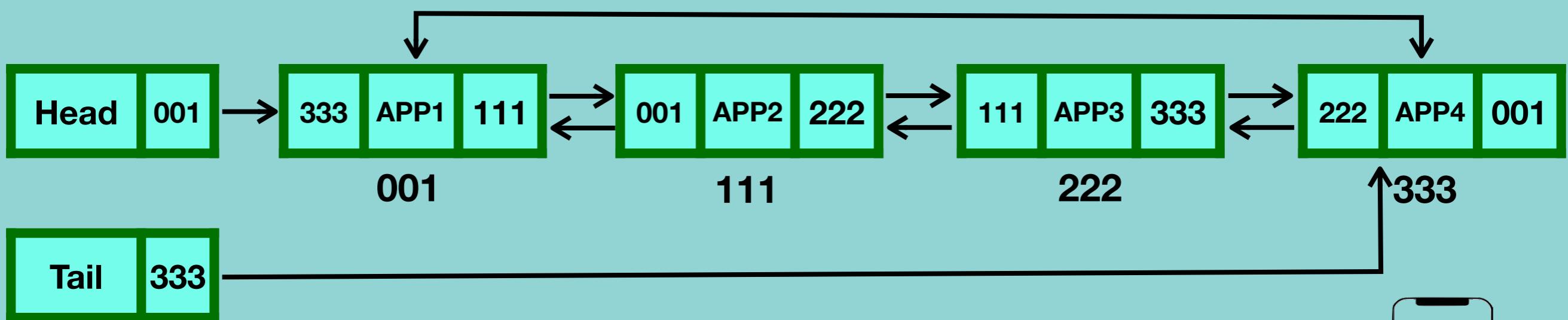
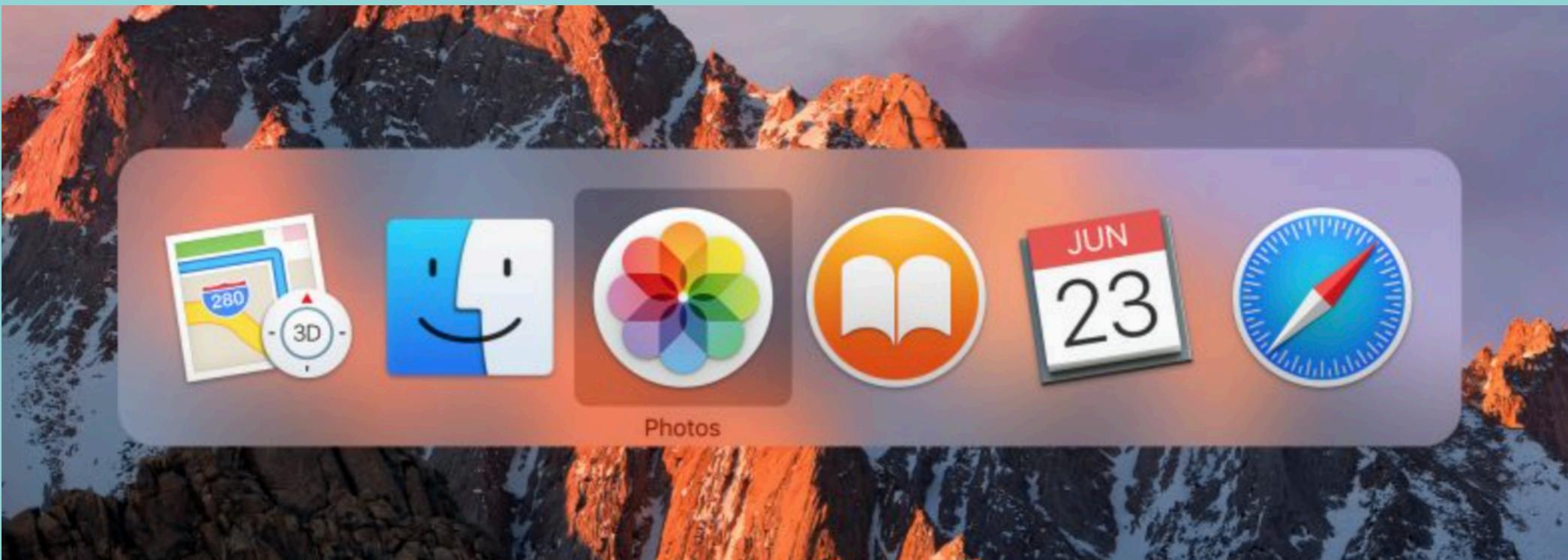
Types of Linked Lists

Circular Doubly Linked List



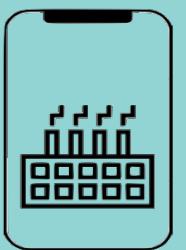
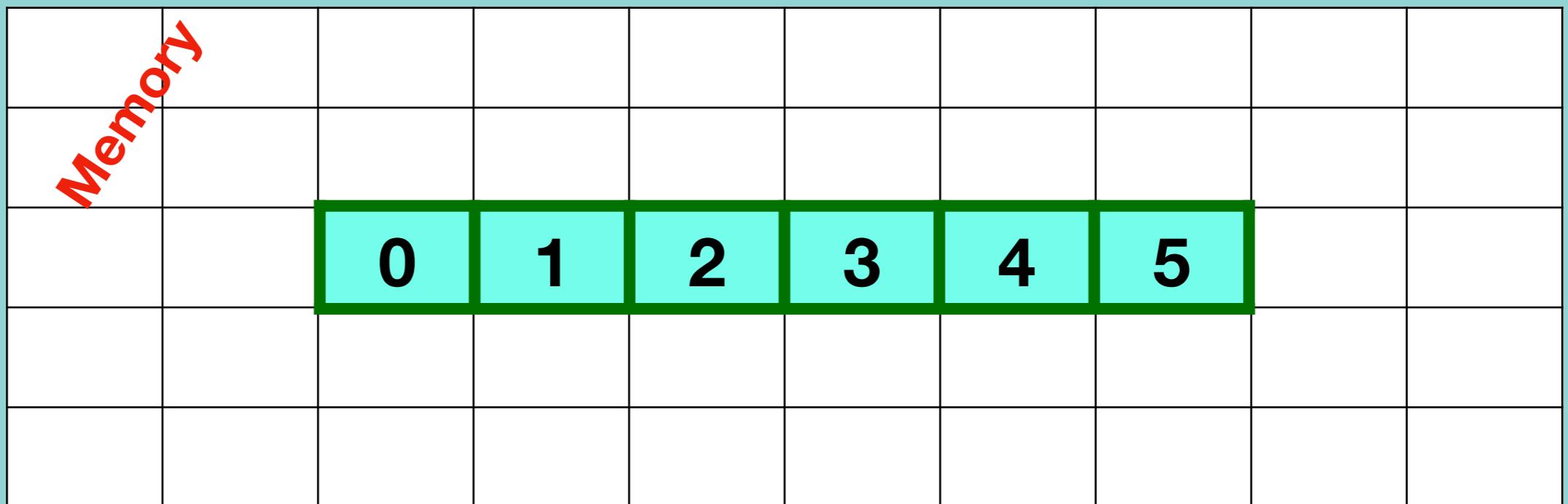
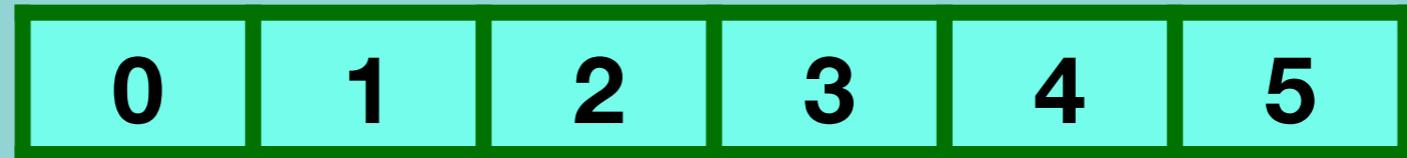
Types of Linked Lists

Cmd+shift+tab



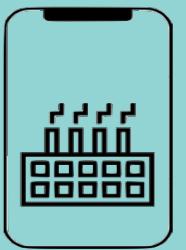
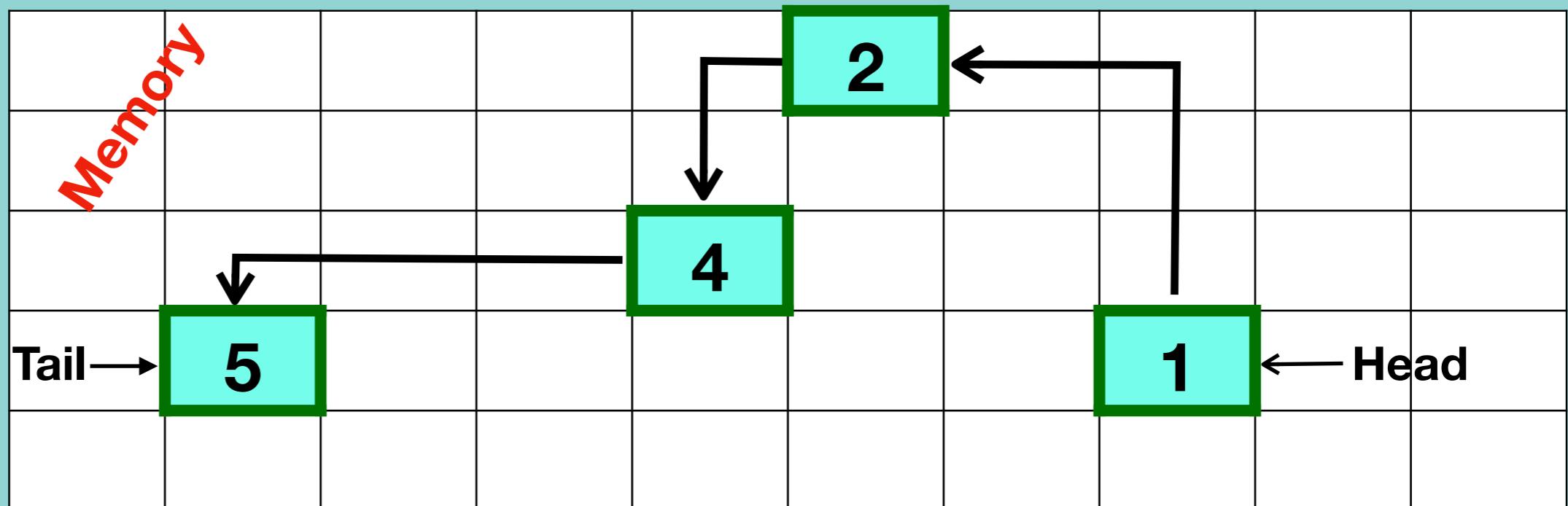
Linked List in Memory

Arrays in memory :

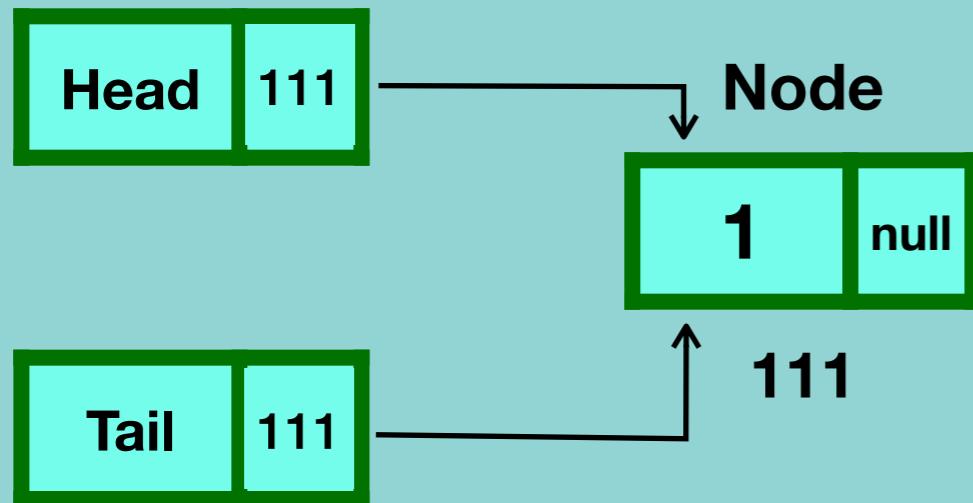


Linked List in Memory

Linked list:



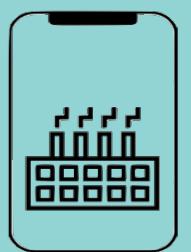
Creation of Singly Linked List



Create Head and Tail, initialize with null

Create a blank Node and assign a value to it and reference to null.

Link Head and Tail with these Node



Creation of Singly Linked List

Create Head and Tail, initialize with null

.....
→ **O(1)**



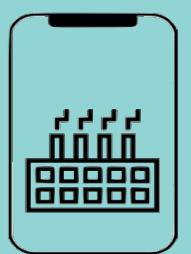
Create a blank Node and assign a value to it and reference to null.

.....
→ **O(1)**



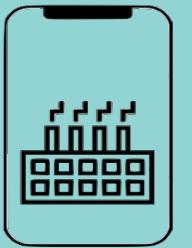
Link Head and Tail with these Node

.....
→ **O(1)**



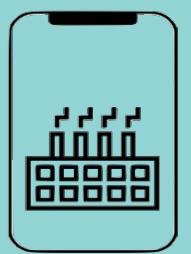
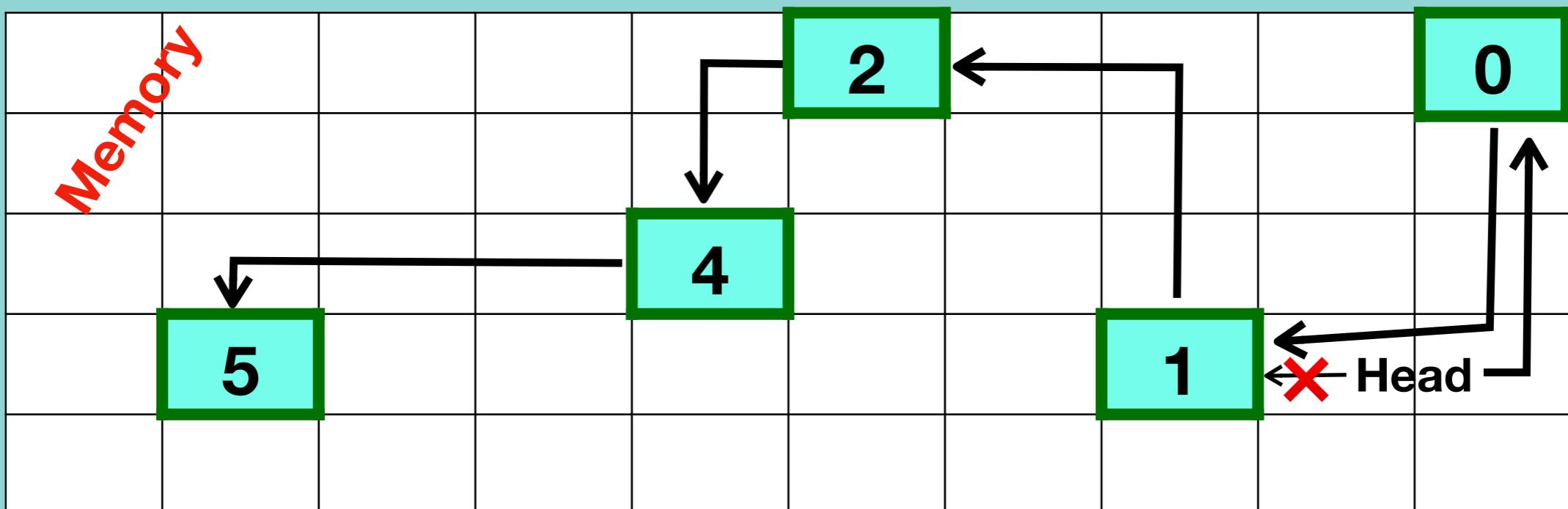
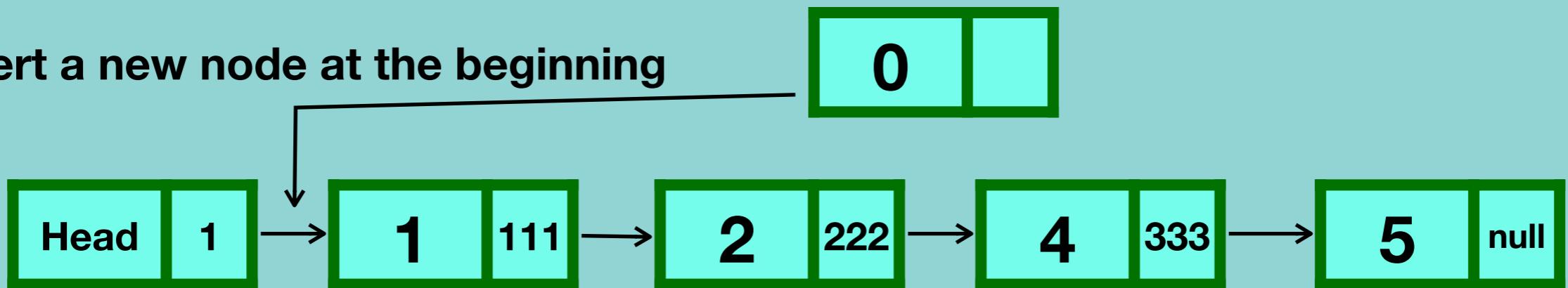
Insertion to Linked List in Memory

1. At the beginning of the linked list.
2. After a node in the middle of linked list
3. At the end of the linked list.



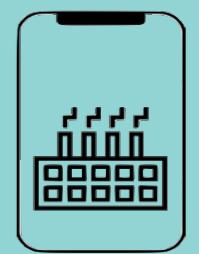
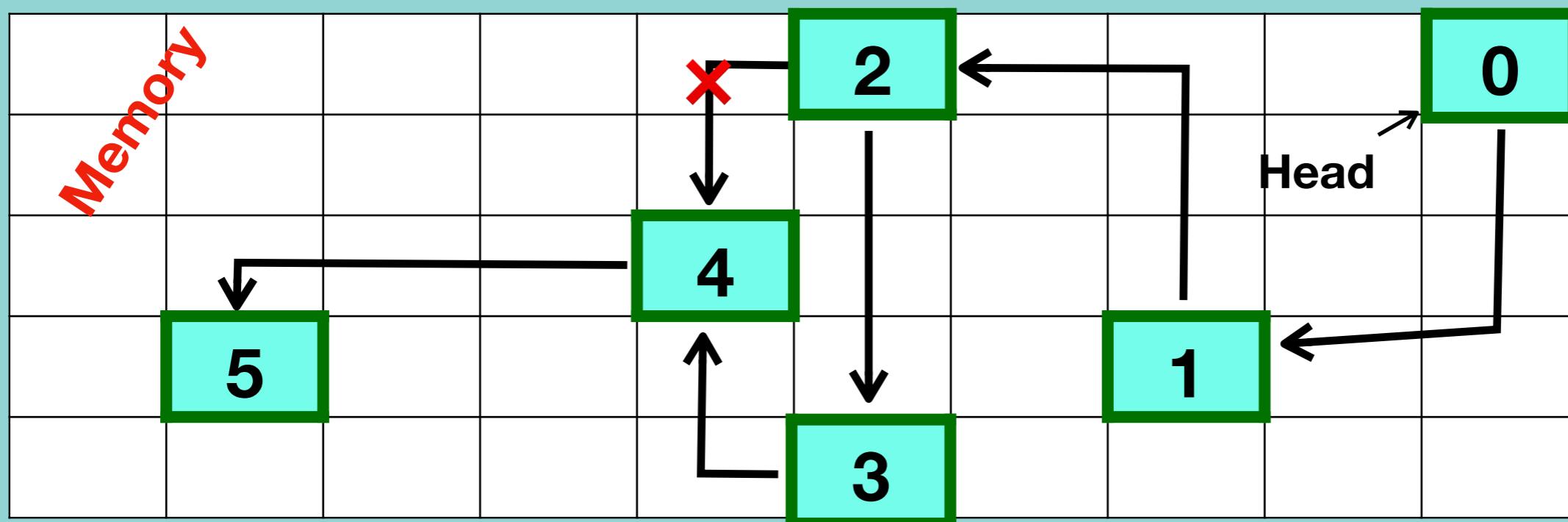
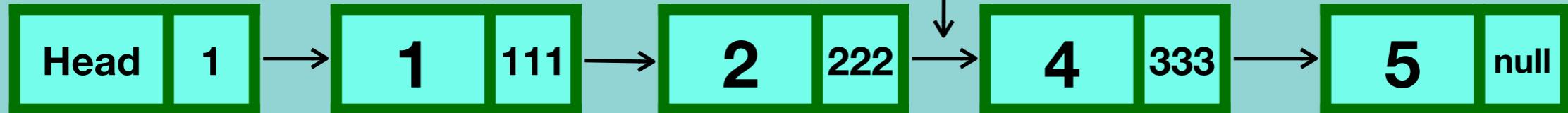
Insertion to Linked List in Memory

Insert a new node at the beginning



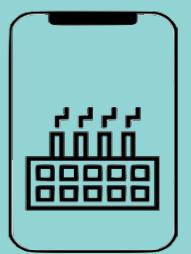
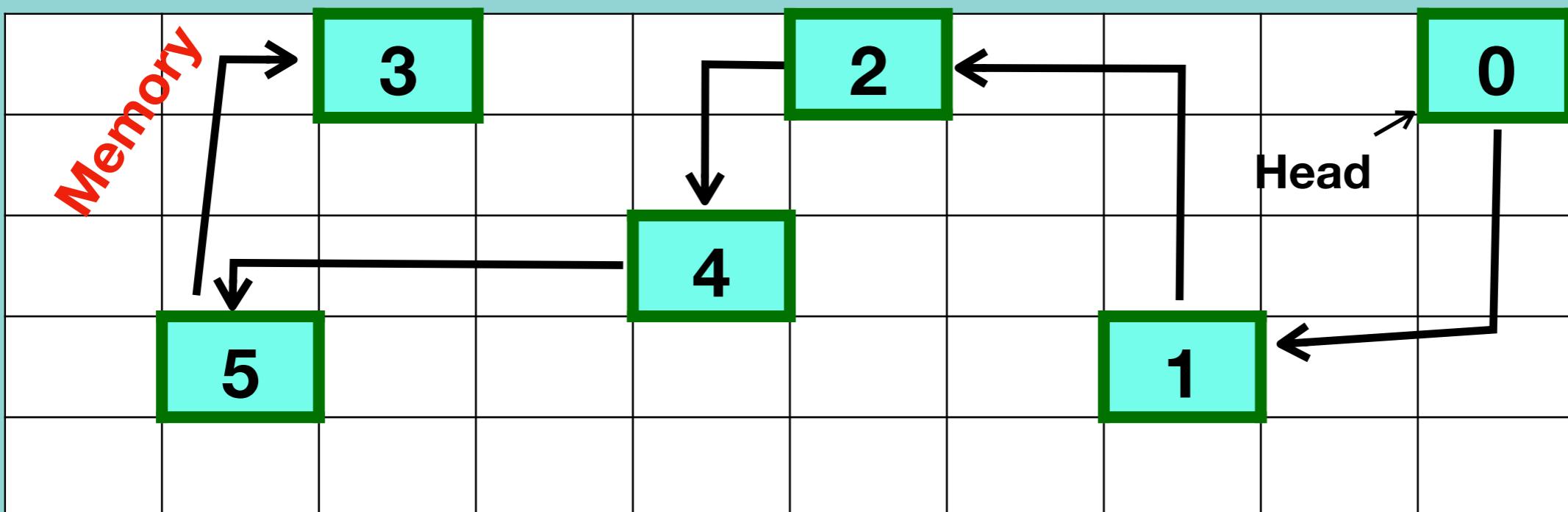
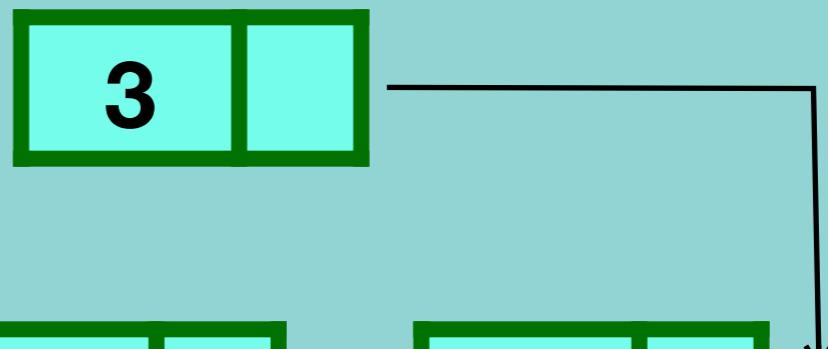
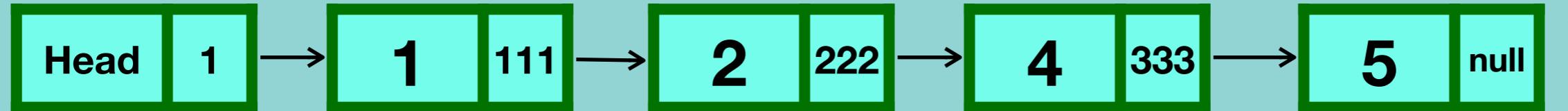
Insertion to Linked List in Memory

Insert a new node after a node

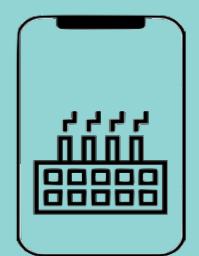
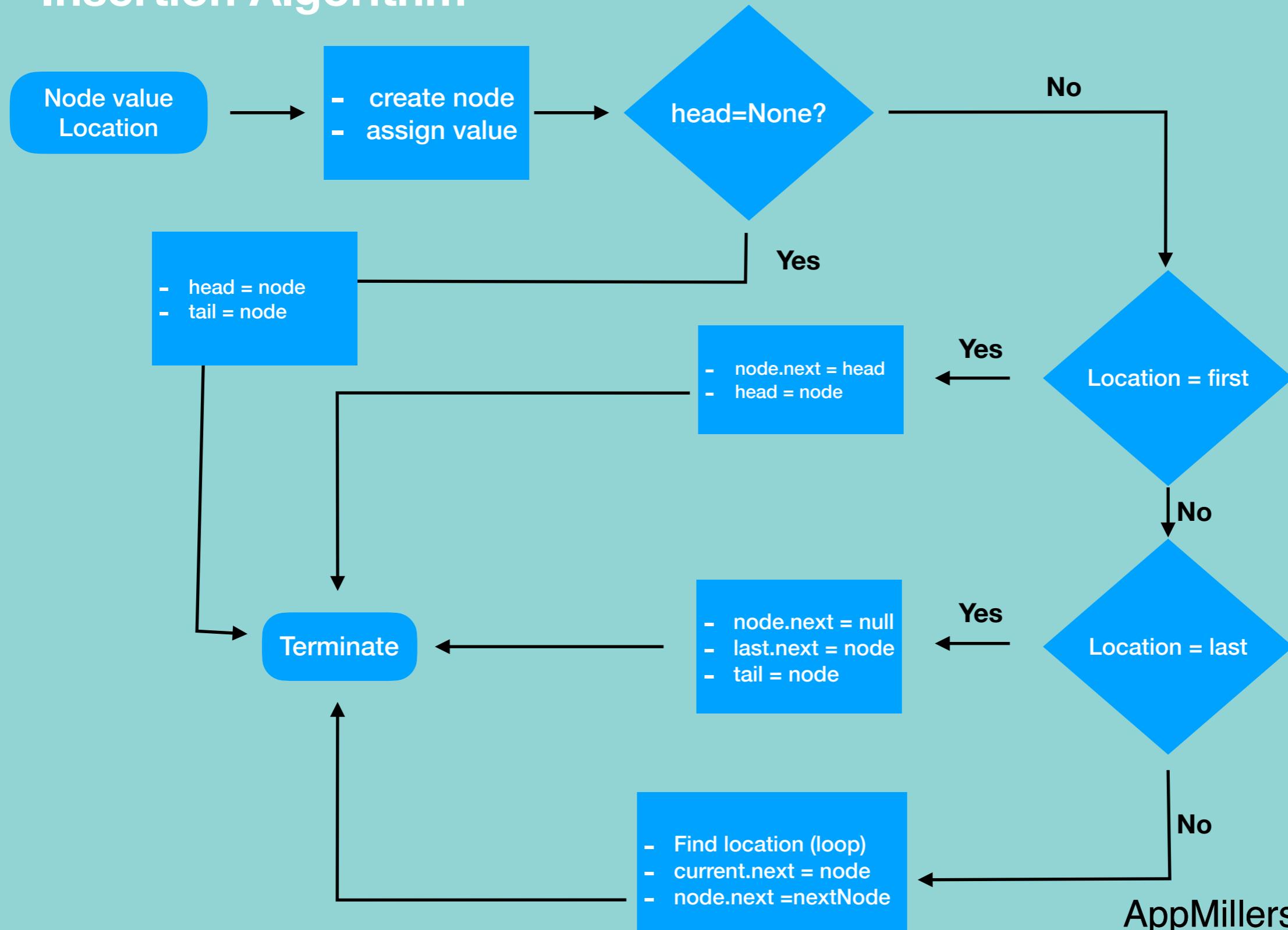


Insertion to Linked List in Memory

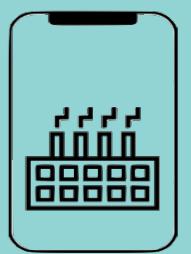
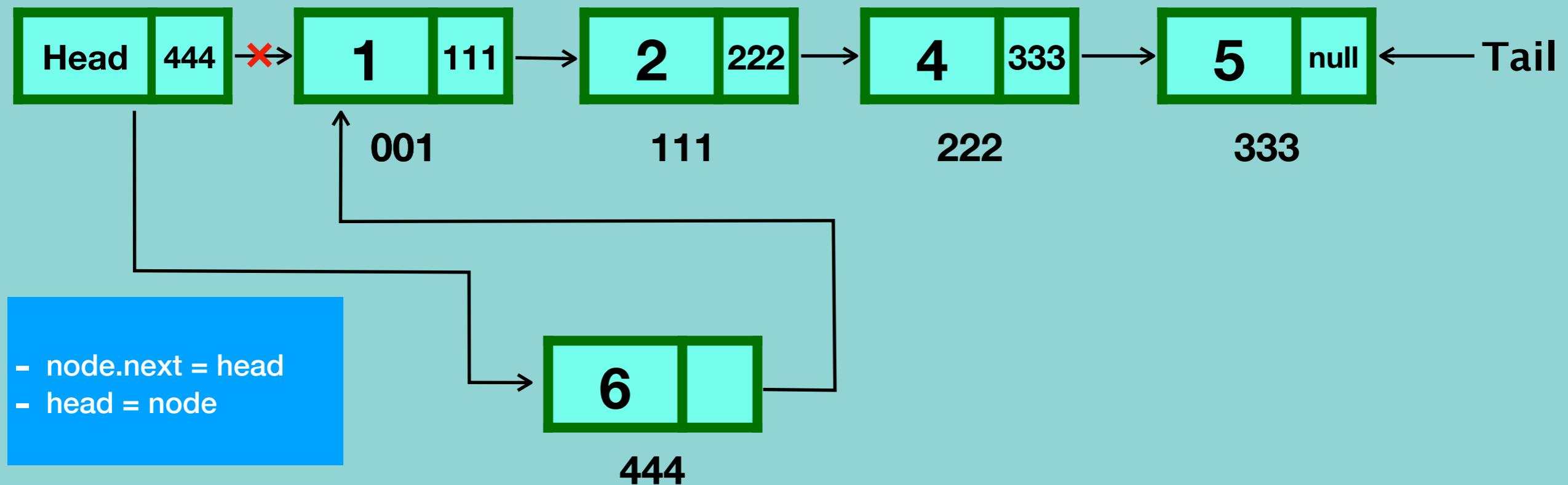
Insert a new node at the end of linked list



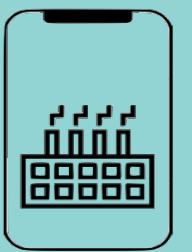
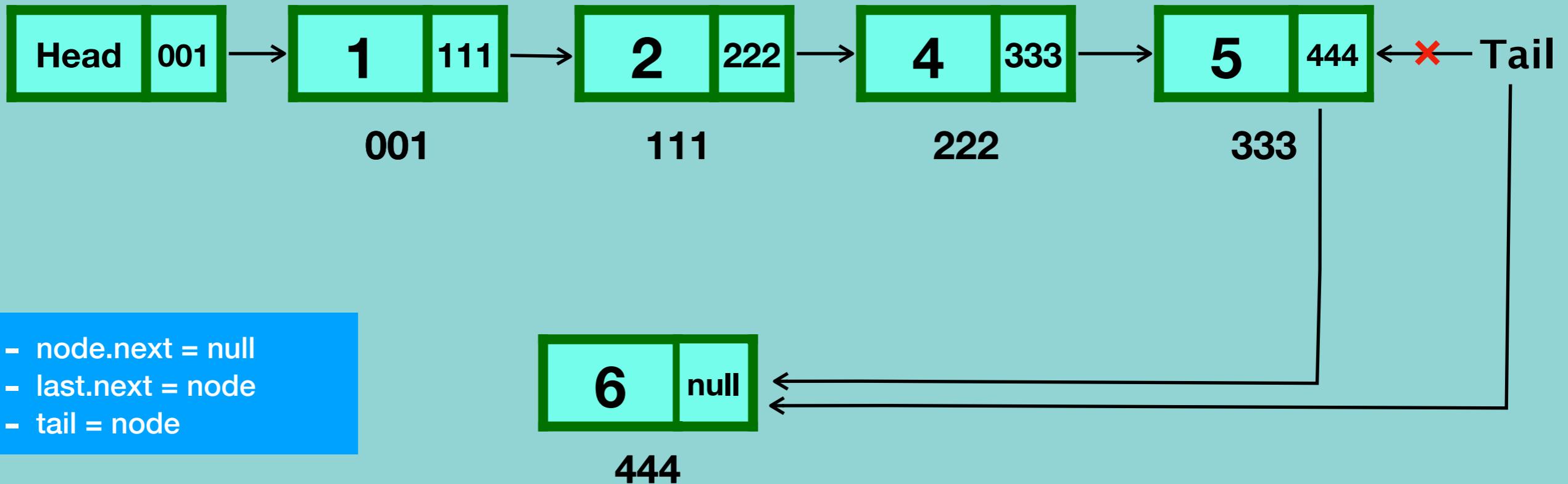
Insertion Algorithm



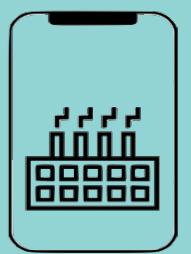
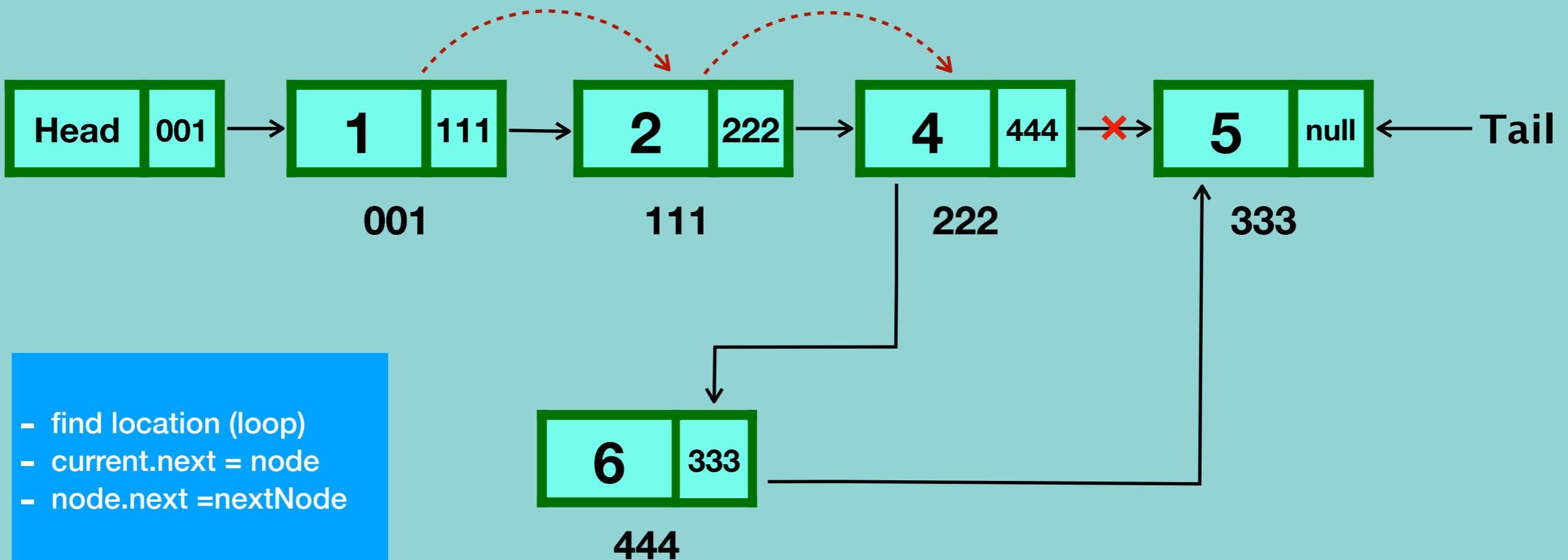
Singly Linked List Insertion at the beginning



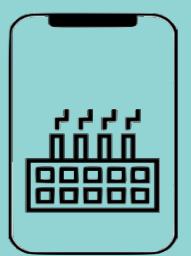
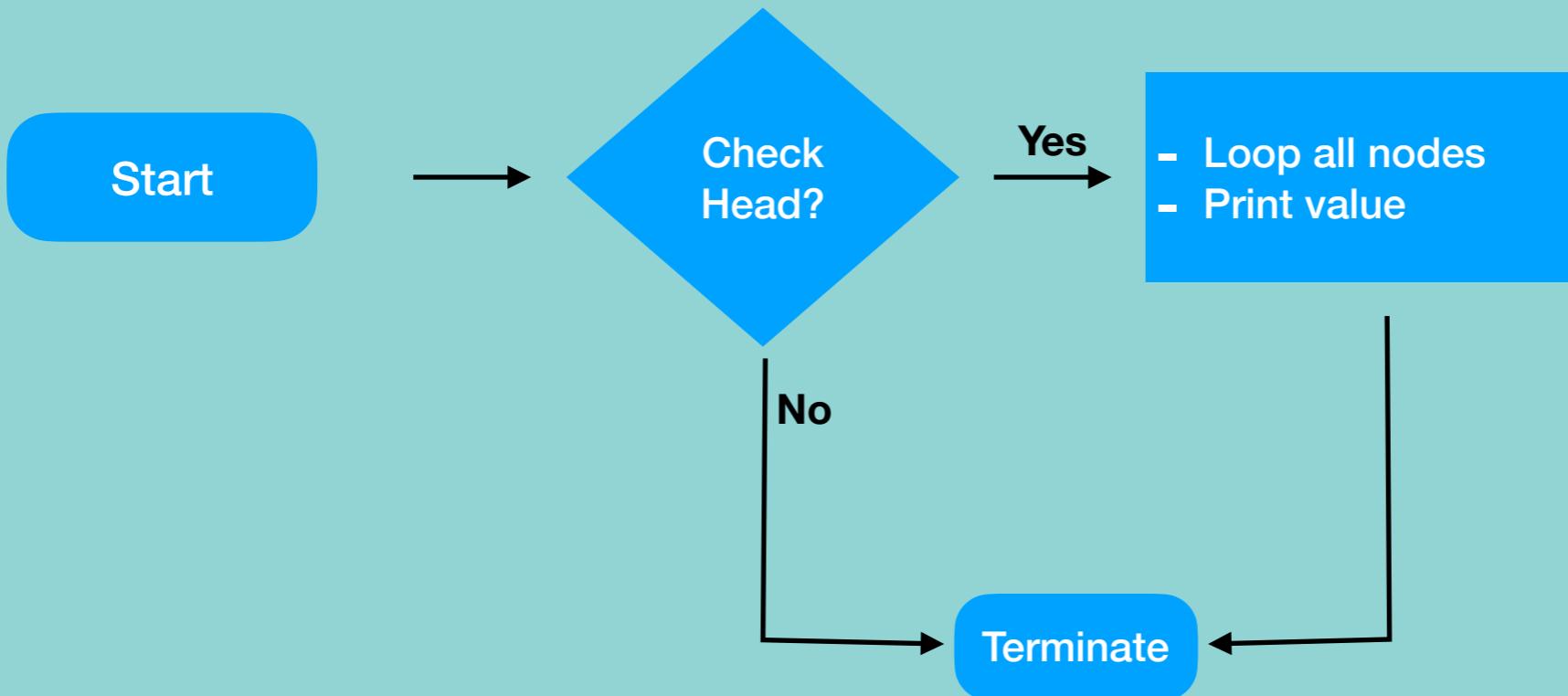
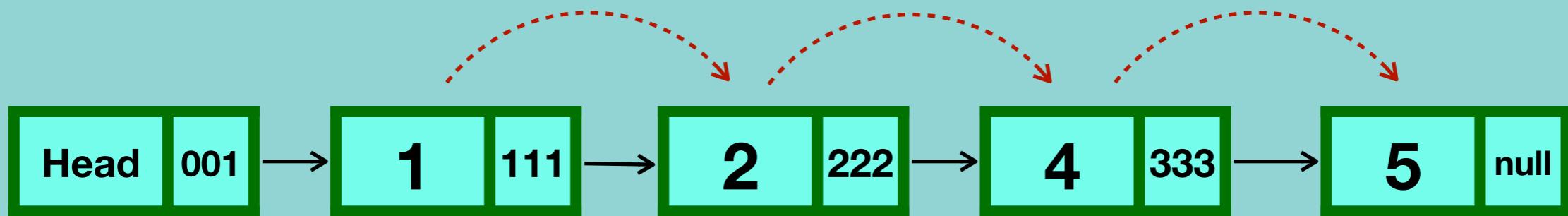
Singly Linked List Insertion at the end



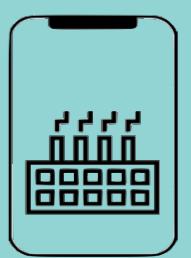
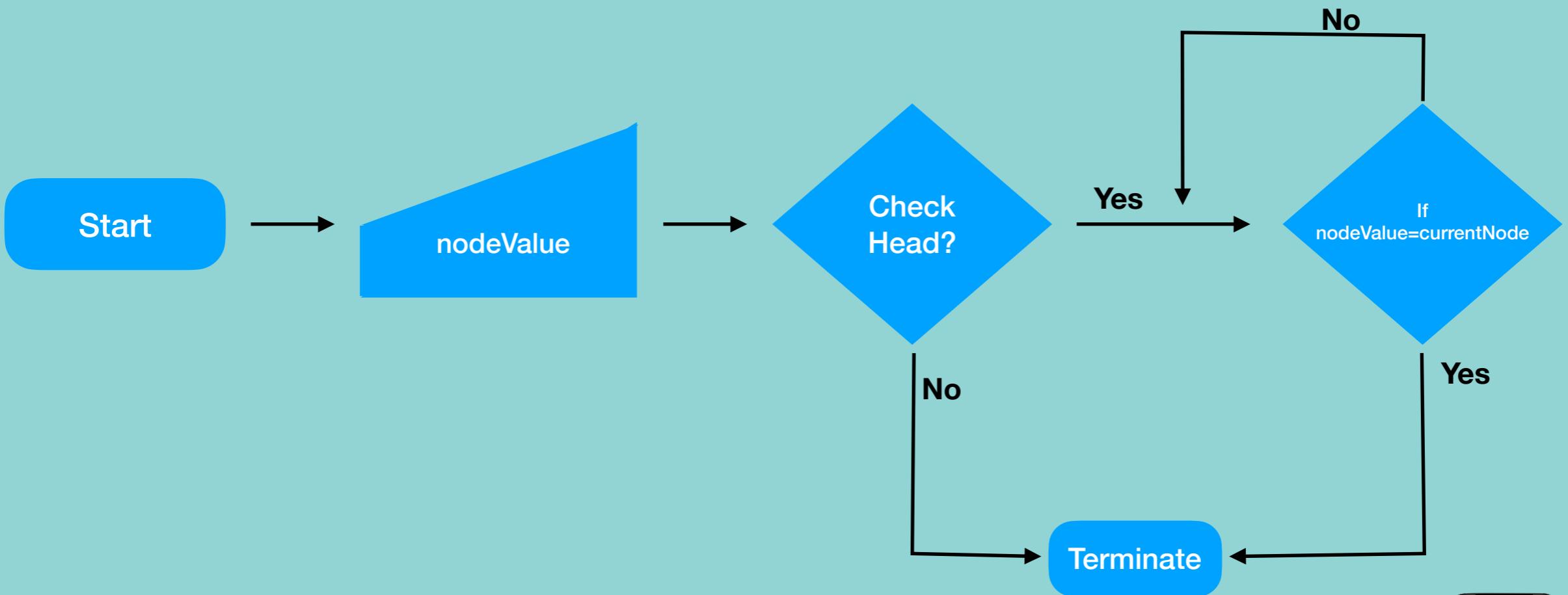
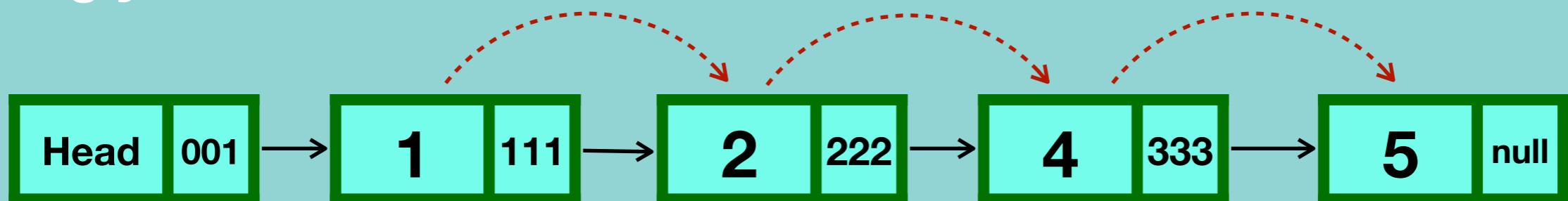
Singly Linked List Insertion in the middle



Singly Linked list Traversal



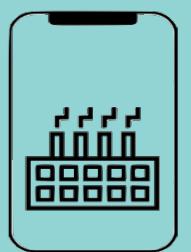
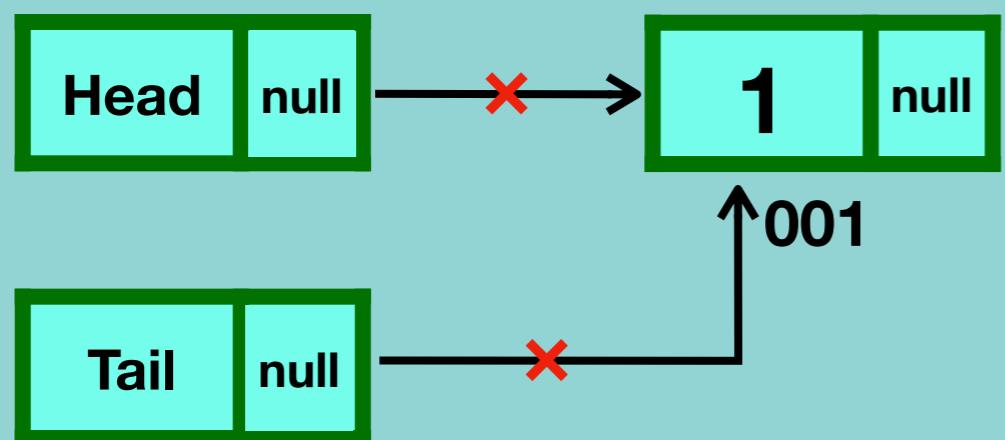
Singly Linked list Search



Singly Linked list Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

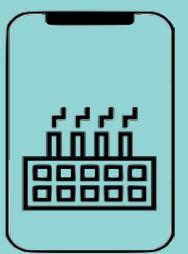
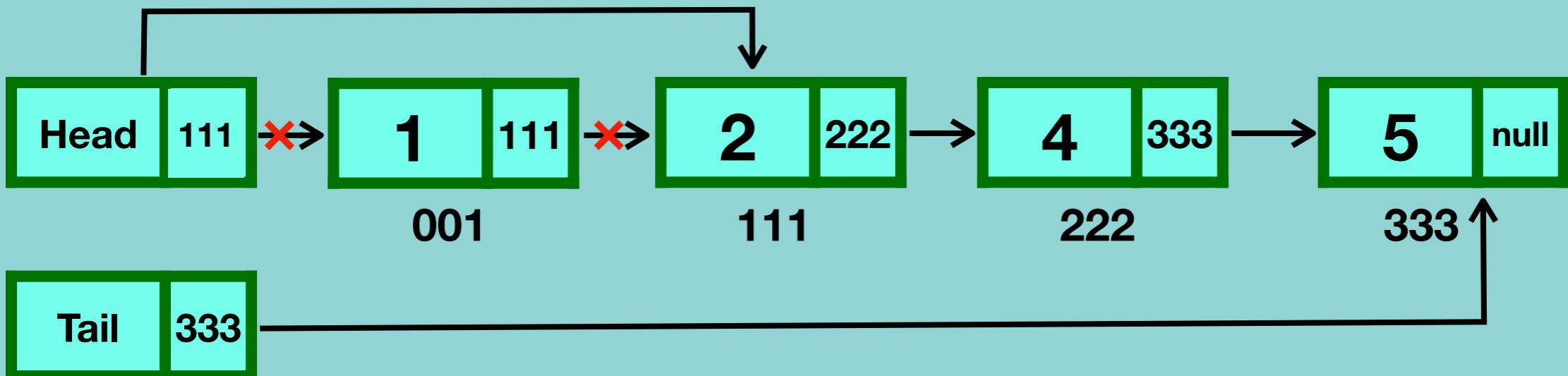
Case 1 - one node



Singly Linked list Deletion

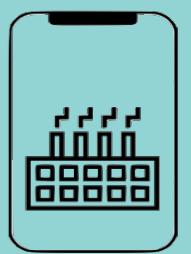
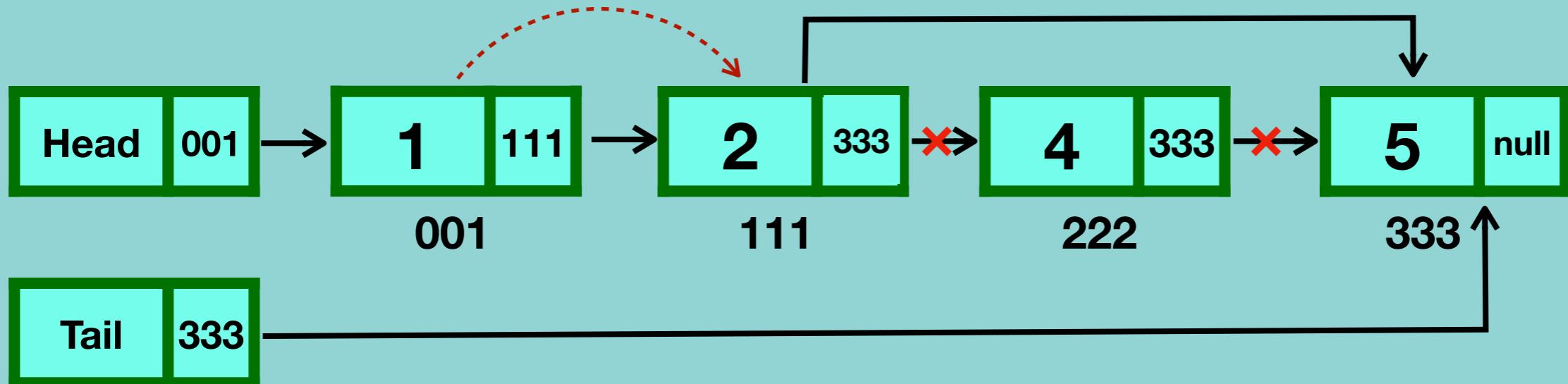
- Deleting the first node
- Deleting any given node
- Deleting the last node

Case 2 - more than one node



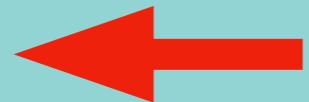
Singly Linked list Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

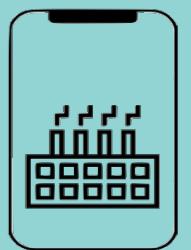
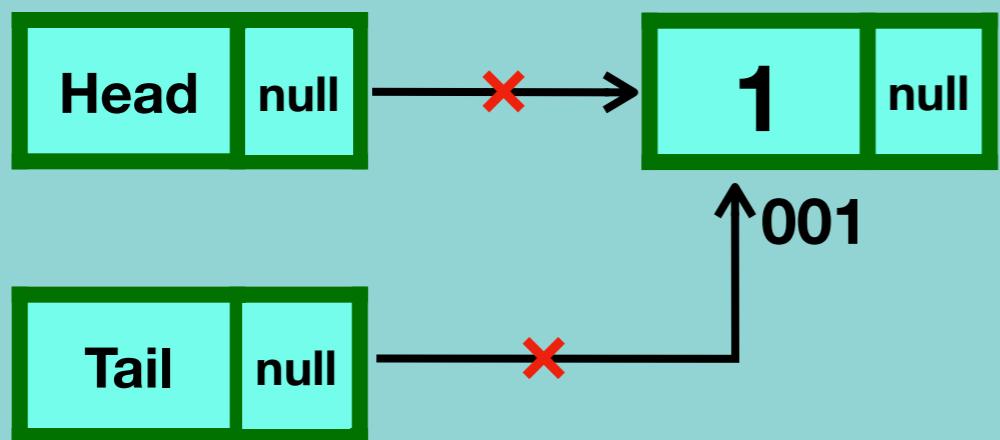


Singly Linked list Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

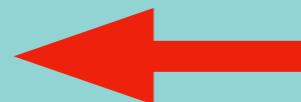


Case 1 - one node

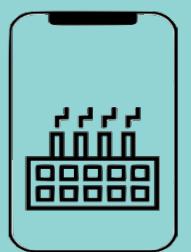
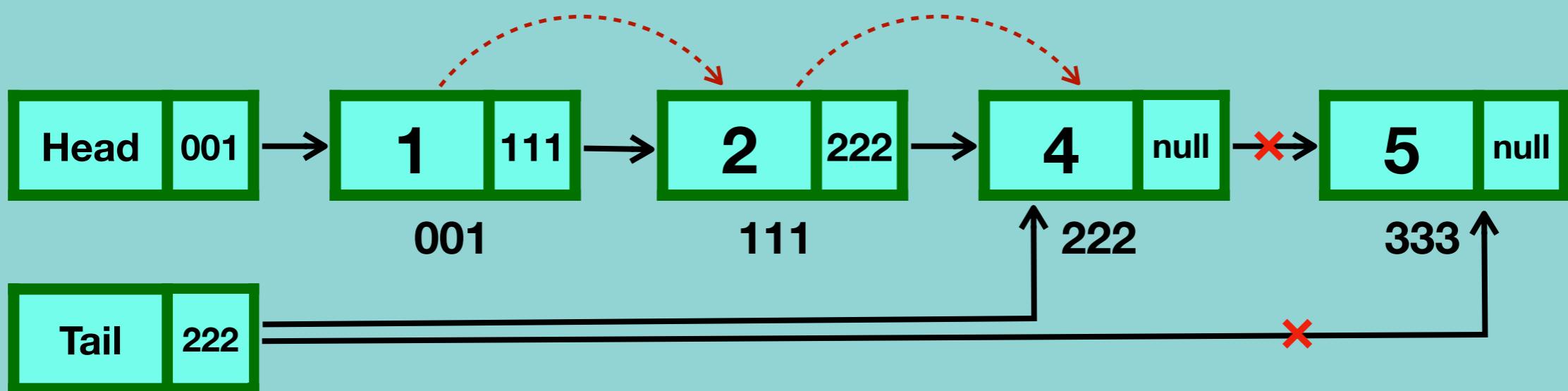


Singly Linked list Deletion

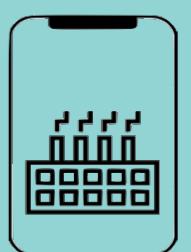
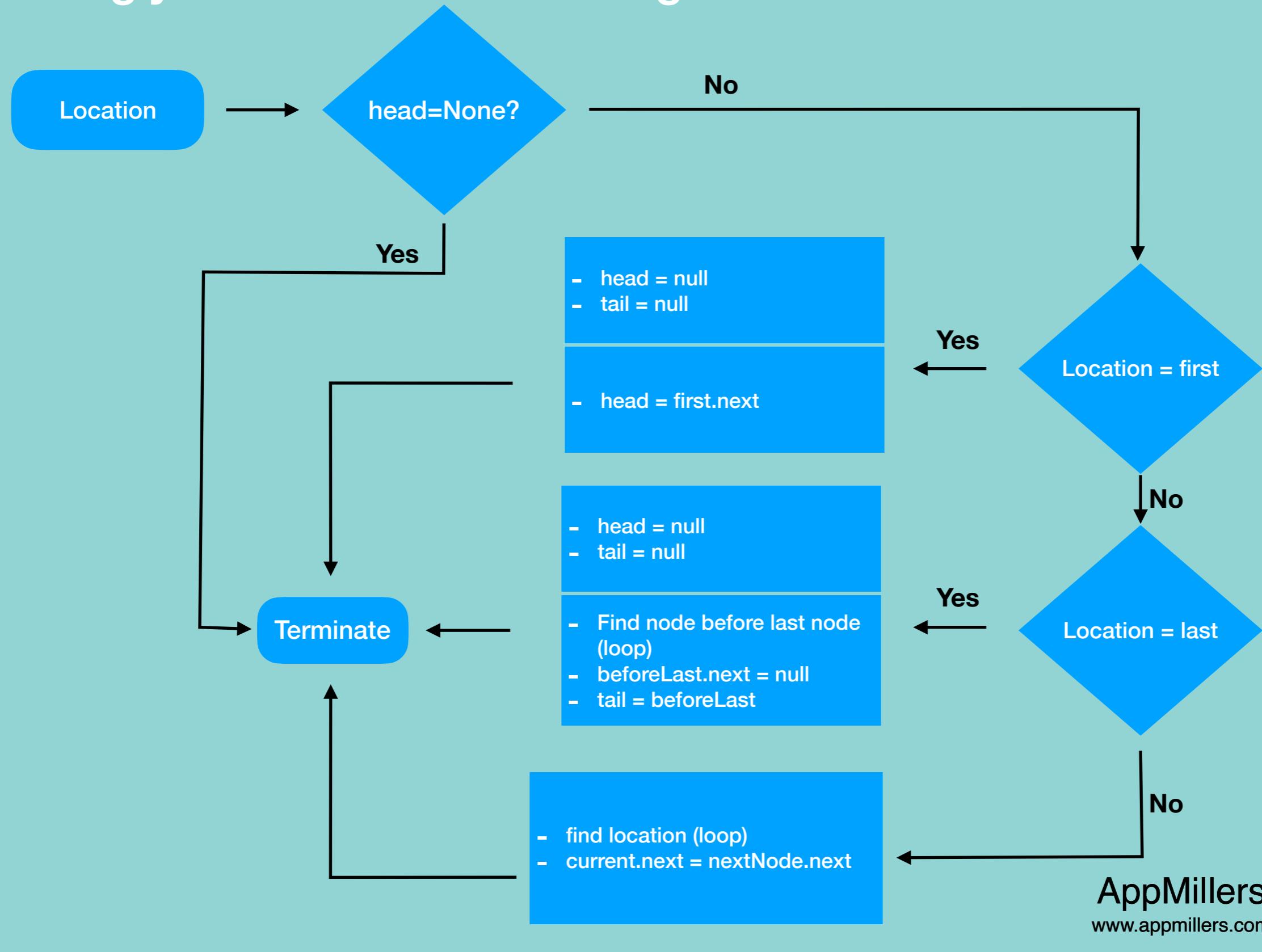
- Deleting the first node
- Deleting any given node
- Deleting the last node



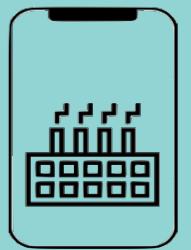
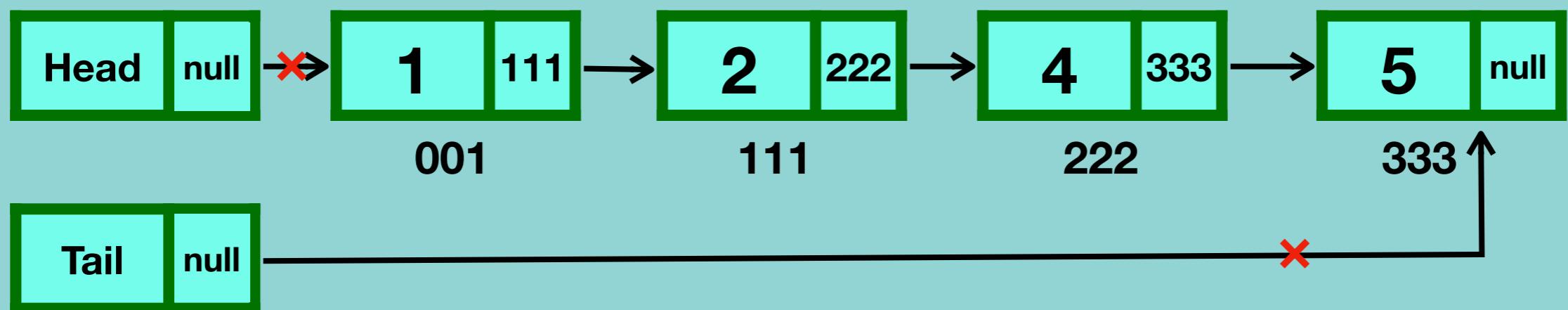
Case 2 - more than one node



Singly Linked list Deletion Algorithm

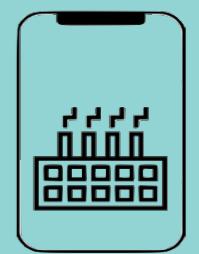
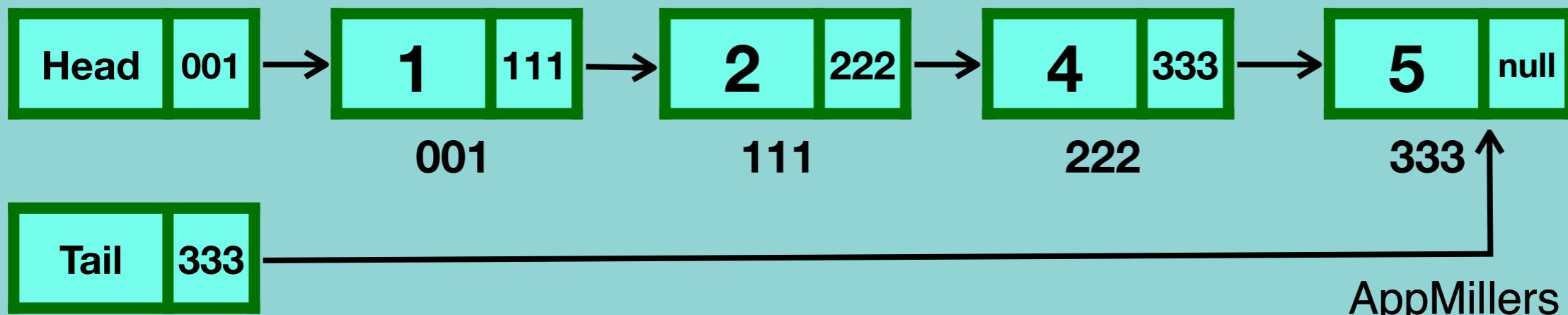


Delete entire Singly Linked list



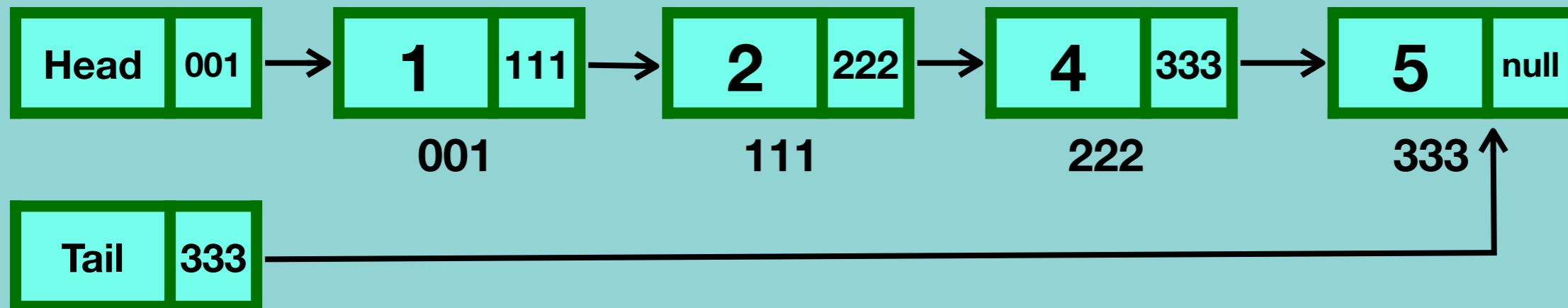
Time and Space complexity of Singly Linked List

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(1)	O(1)

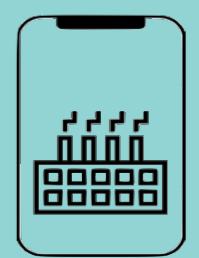
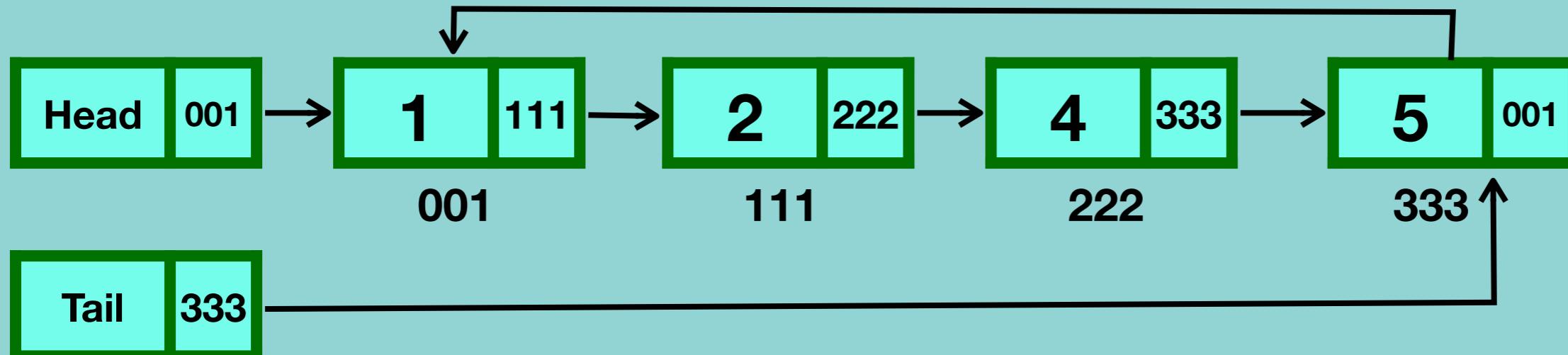


Circular Singly Linked List

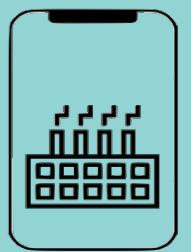
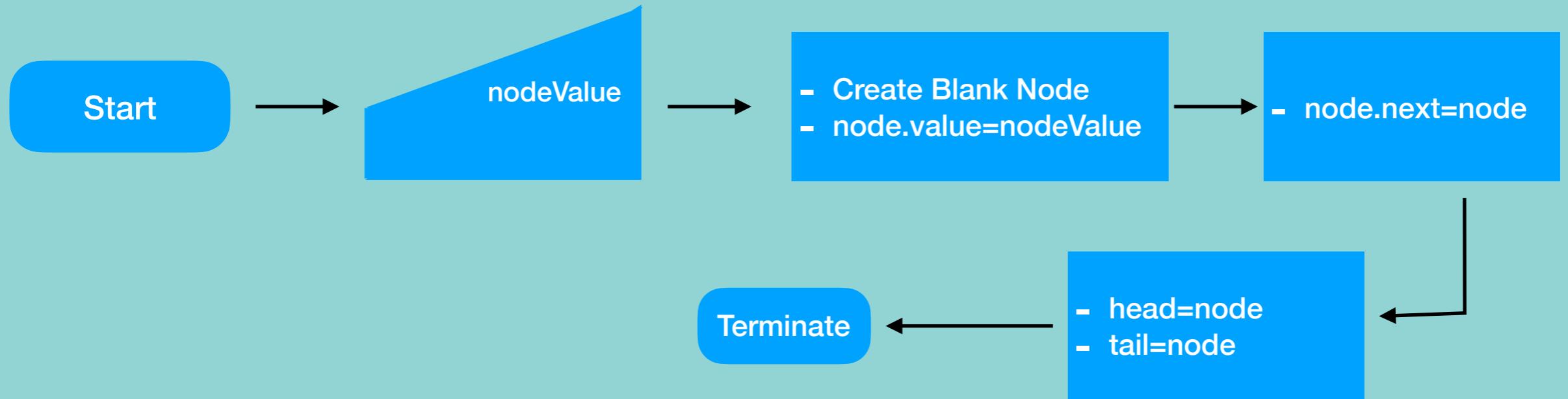
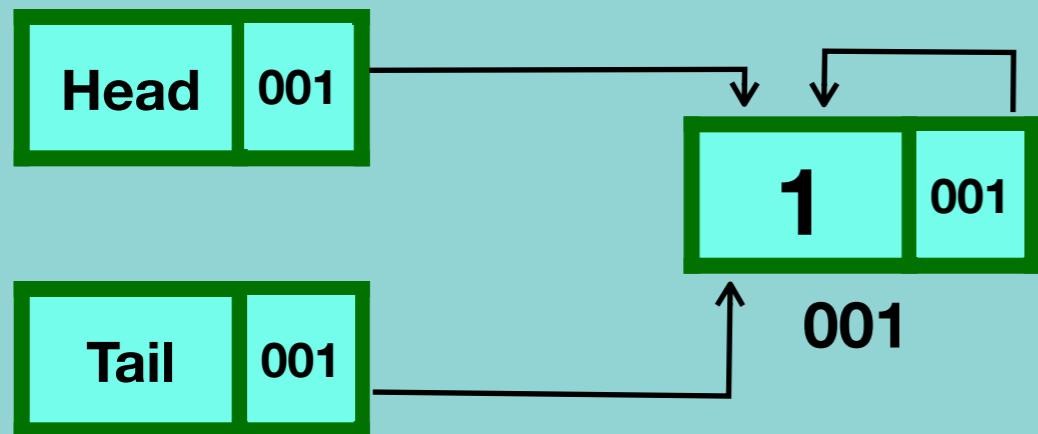
Singly Linked List



Circular Singly Linked List

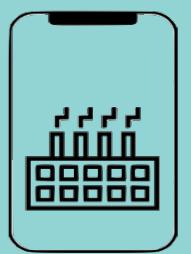
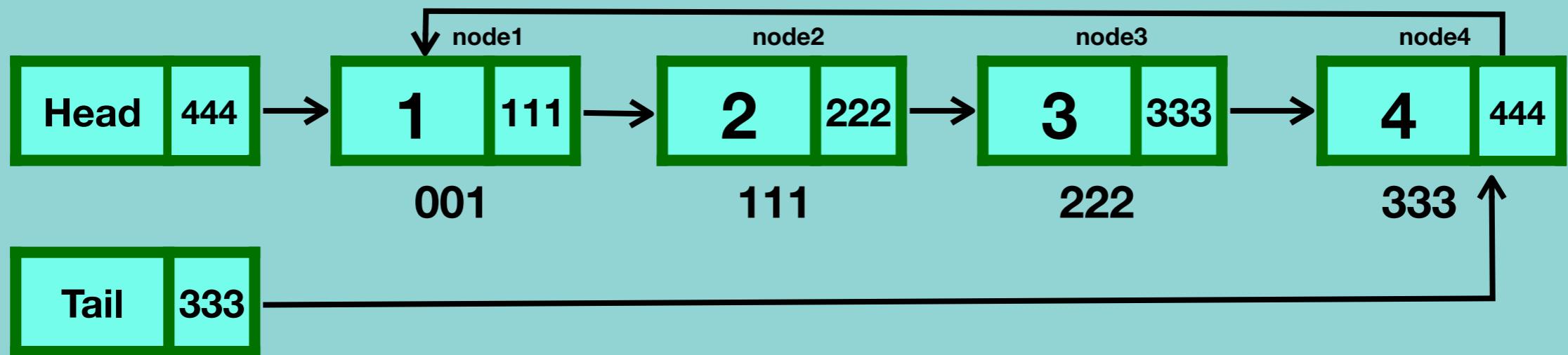


Creation of Circular Singly Linked List



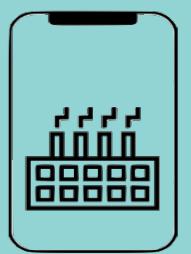
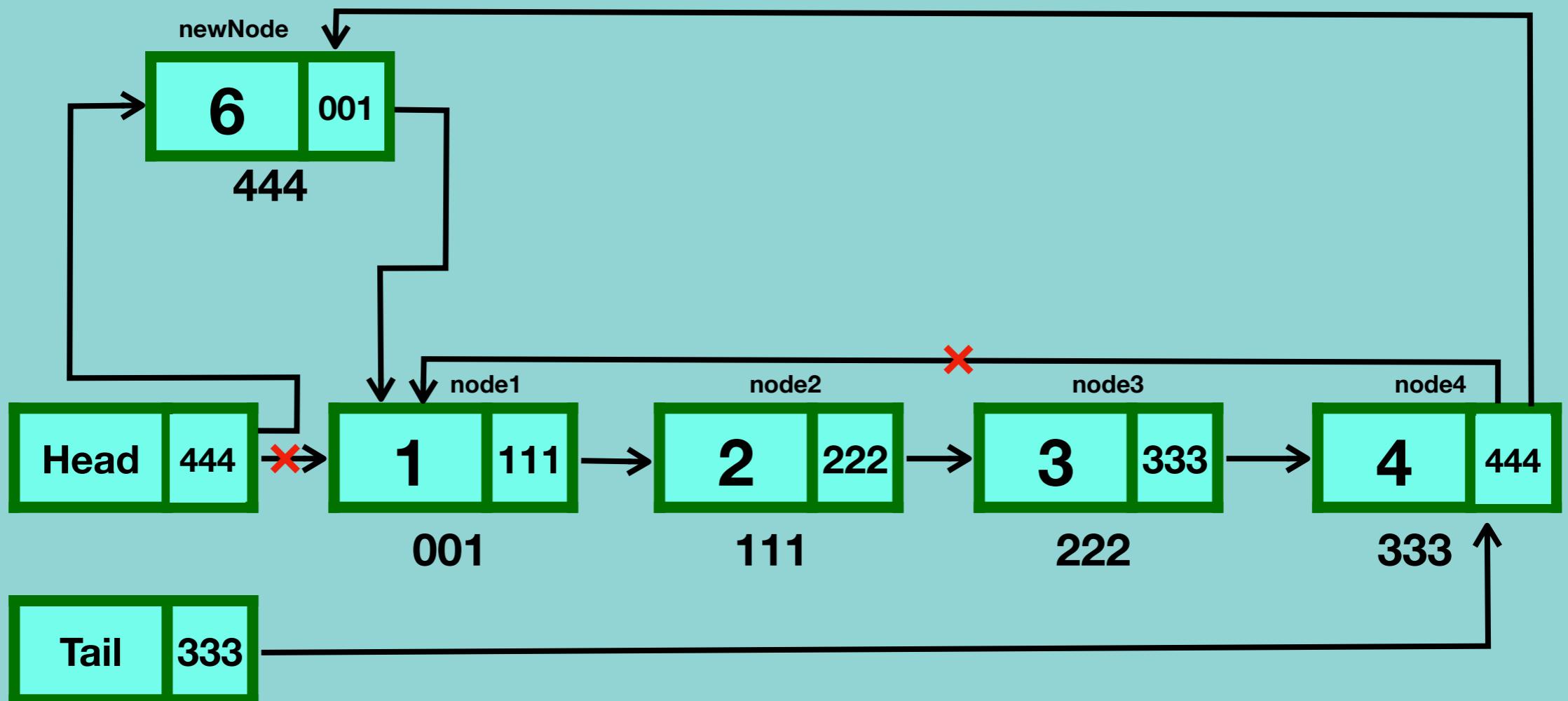
Circular Singly Linked List - Insertion

- Insert at the beginning of linked list
- Insert at the specified location of linked list
- Insert at the end of linked list



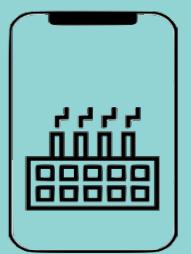
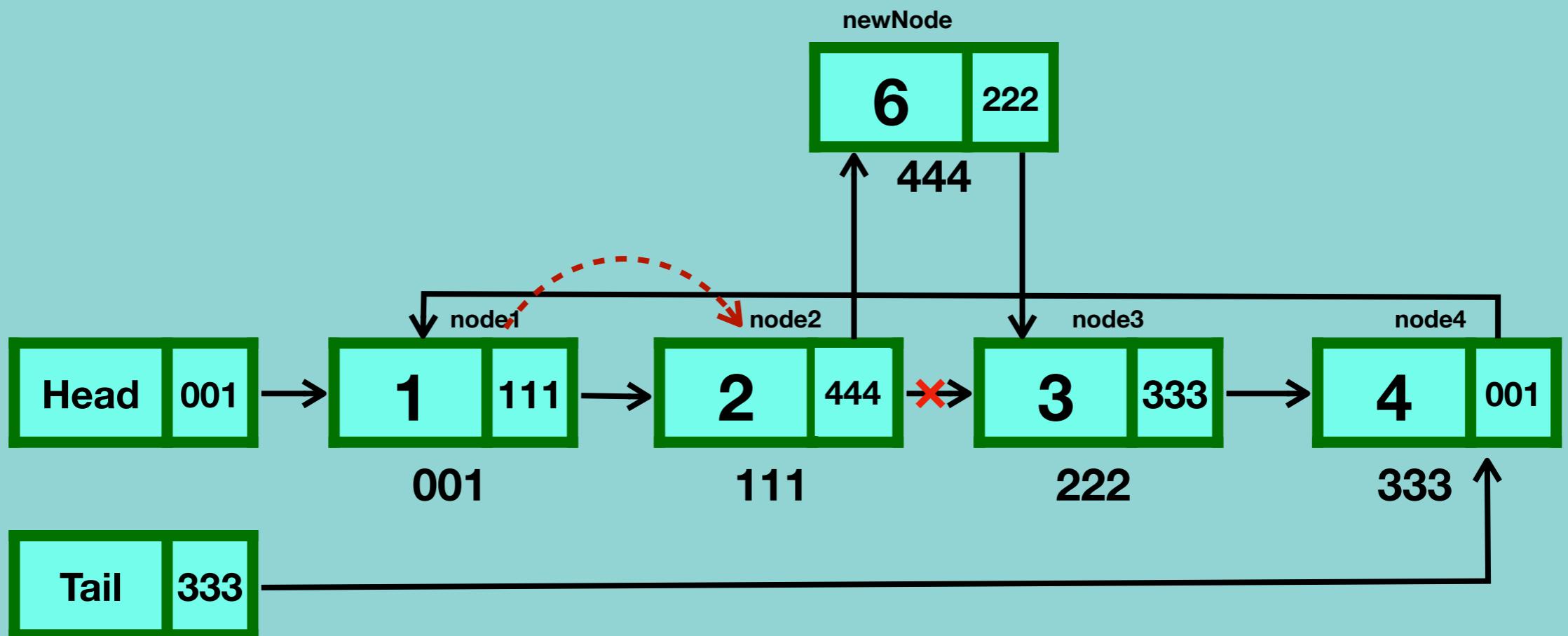
Circular Singly Linked List - Insertion

- Insert at the beginning of linked list



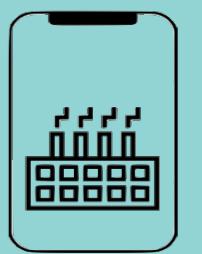
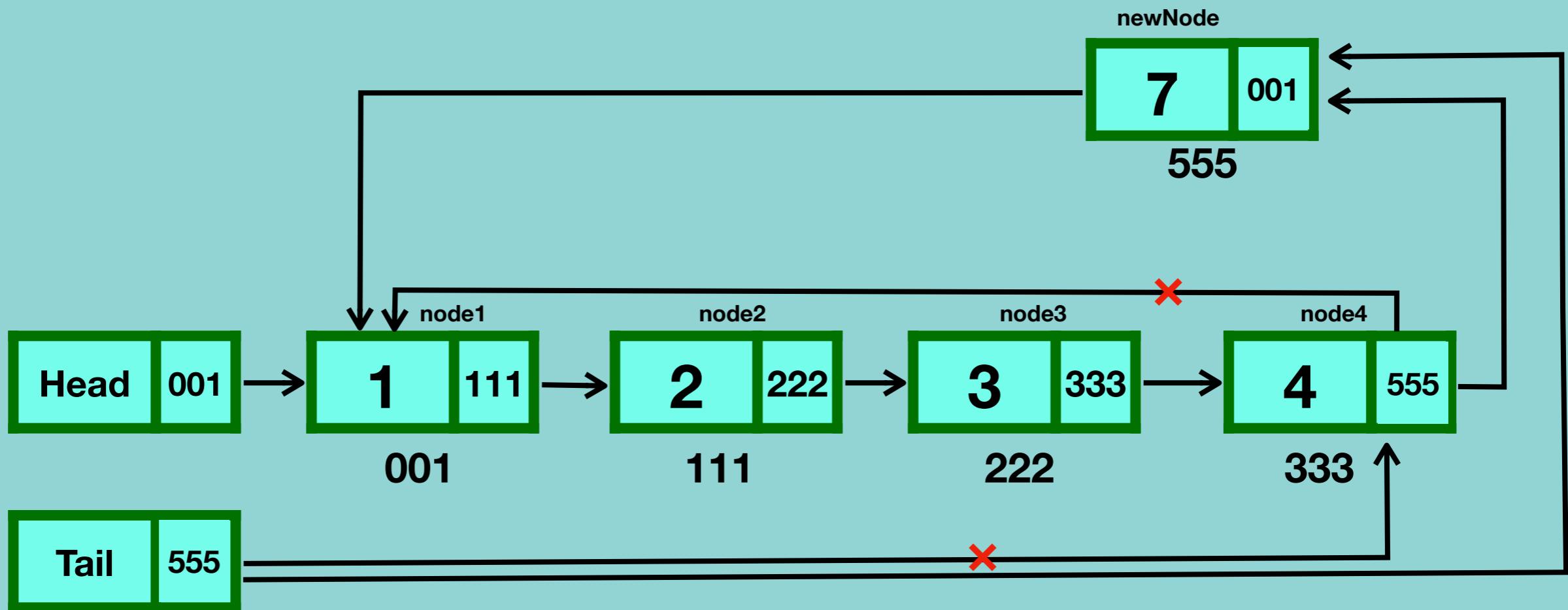
Circular Singly Linked List - Insertion

- Insert at the specified location of linked list

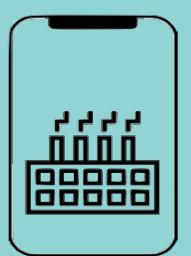
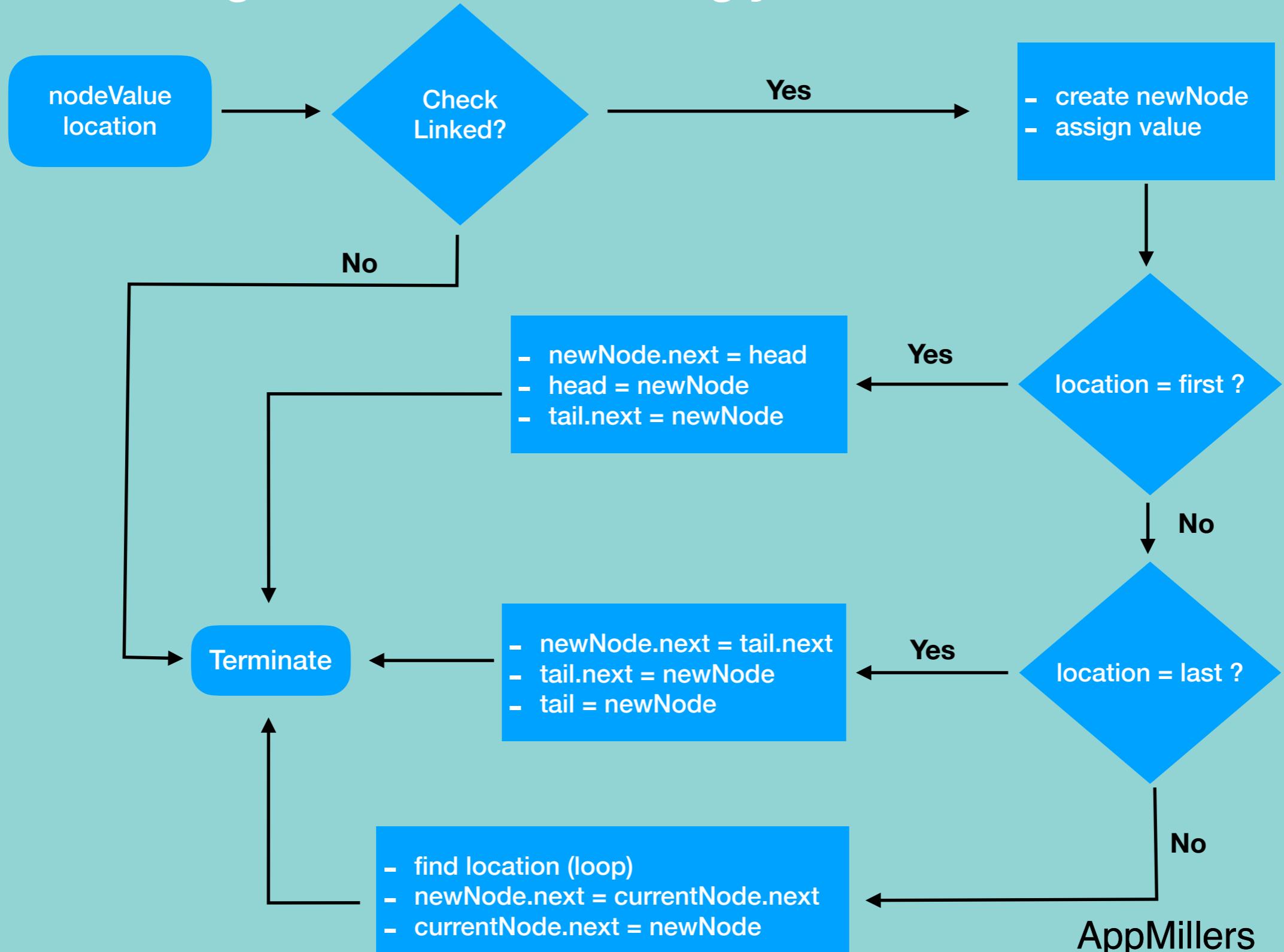


Circular Singly Linked List - Insertion

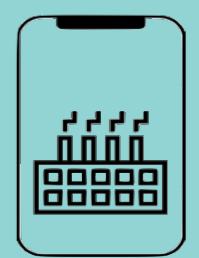
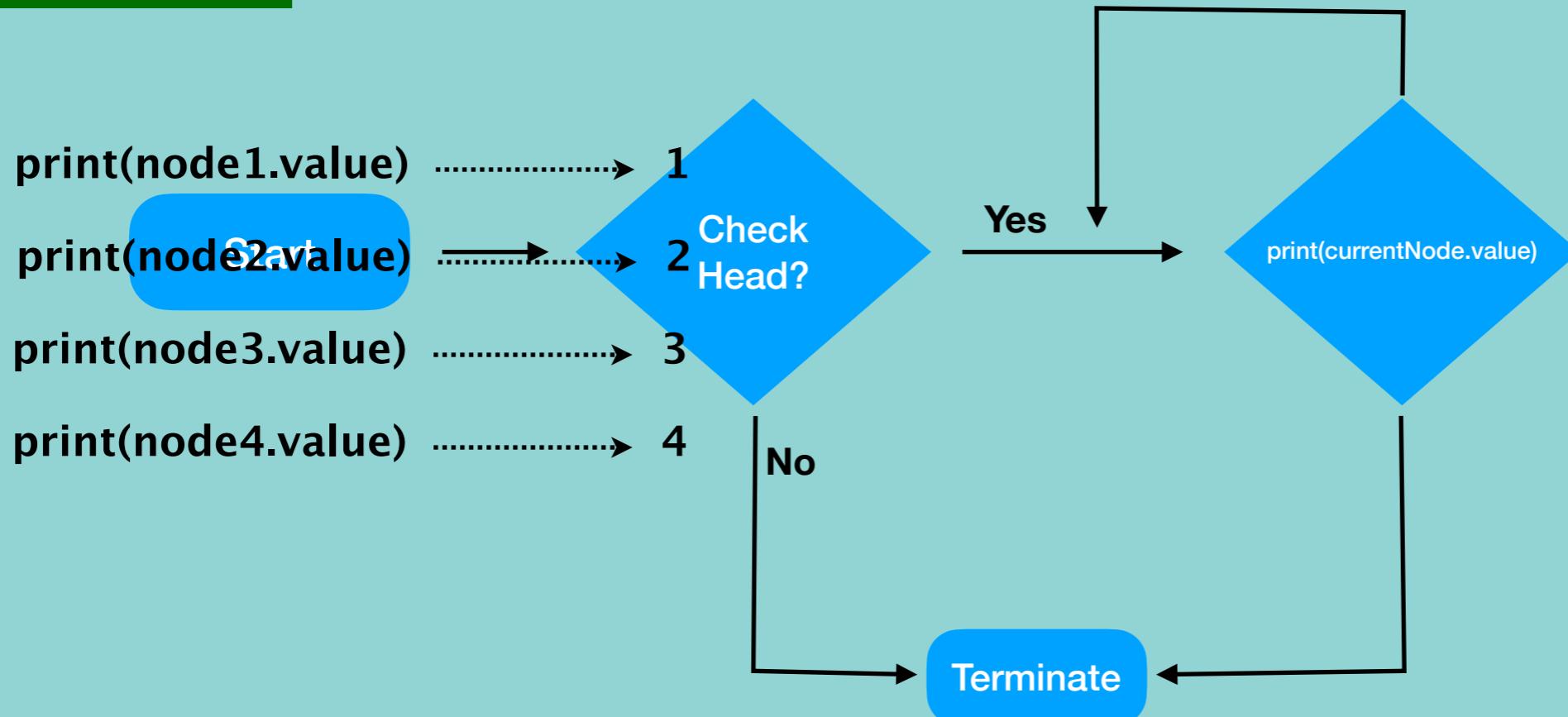
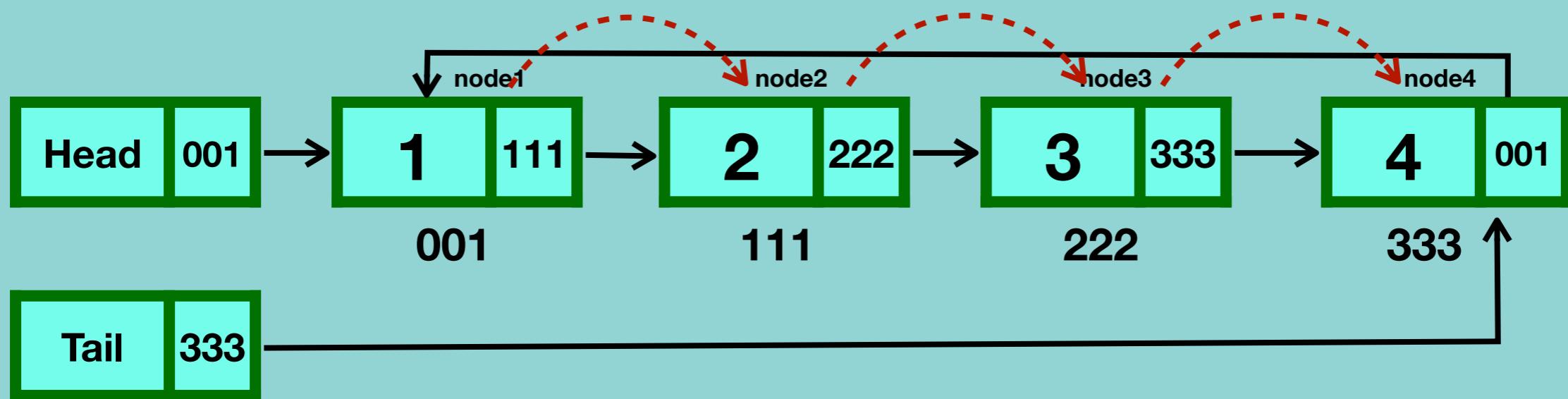
- Insert at the end of linked list



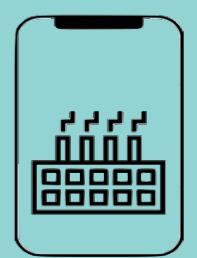
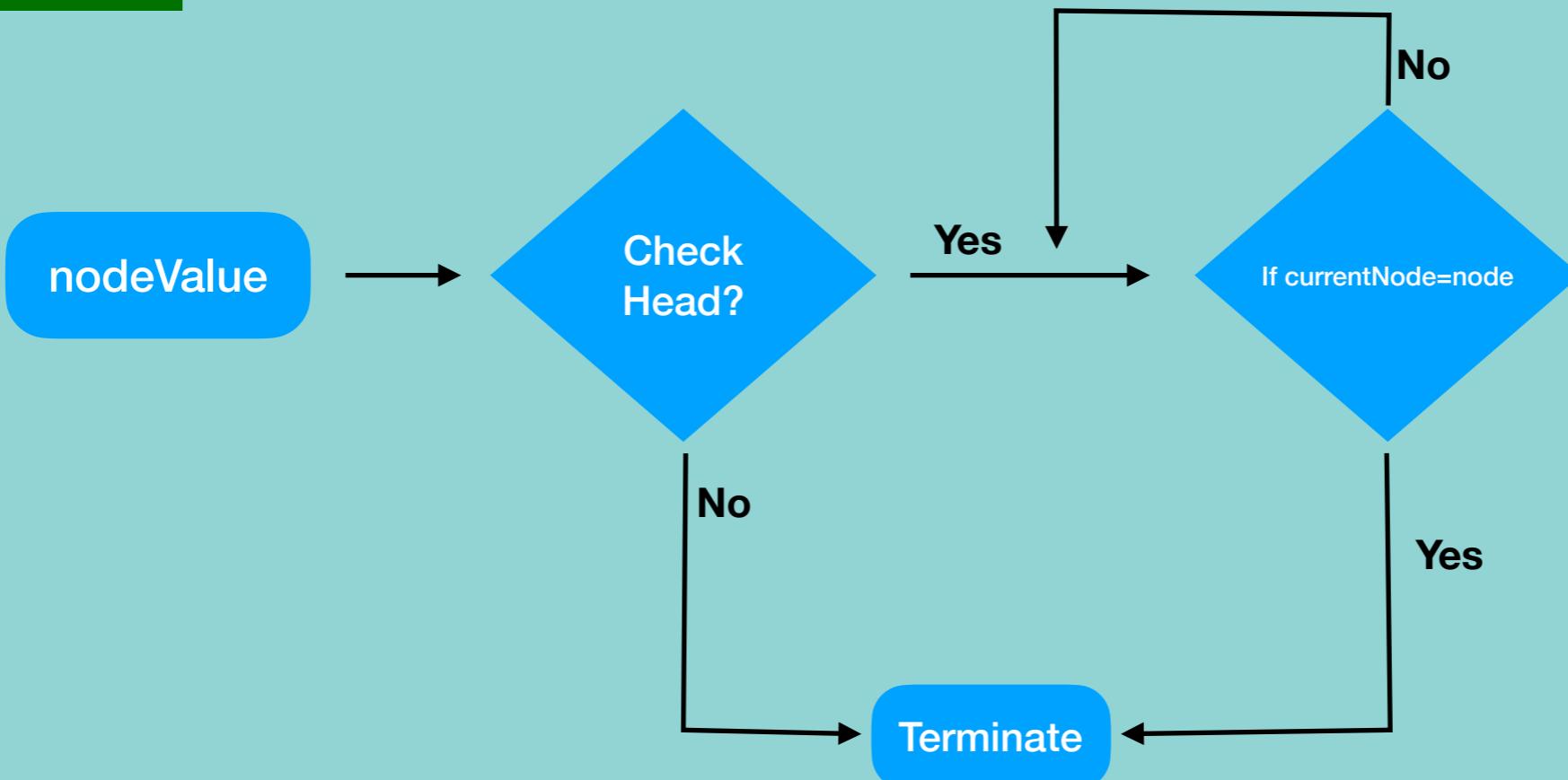
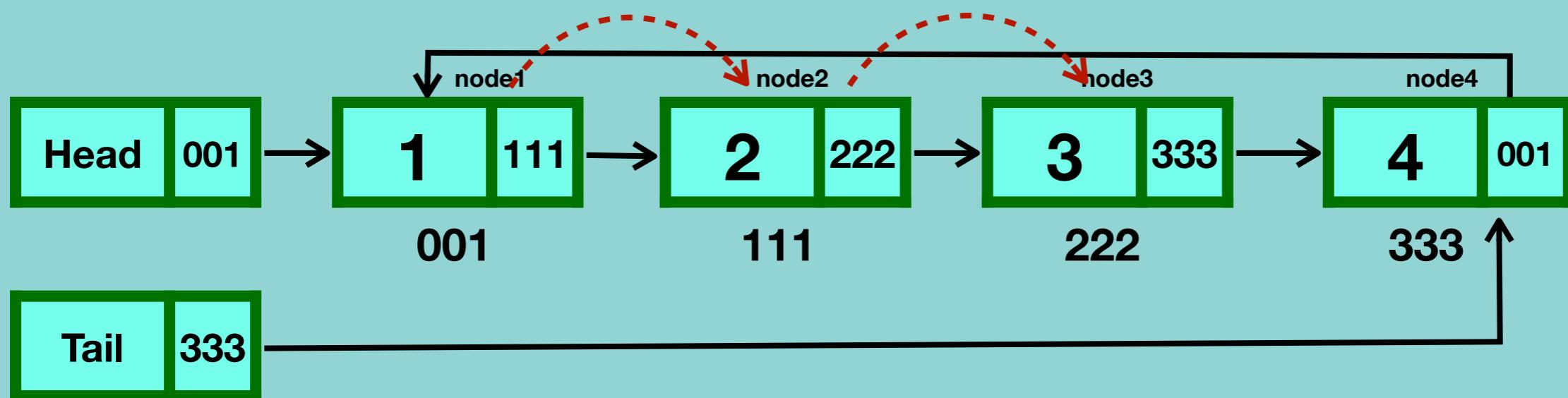
Insertion Algorithm - Circular singly linked list



Circular Singly Linked List - Traversal



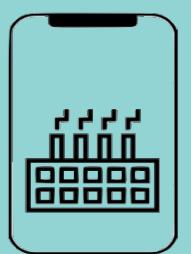
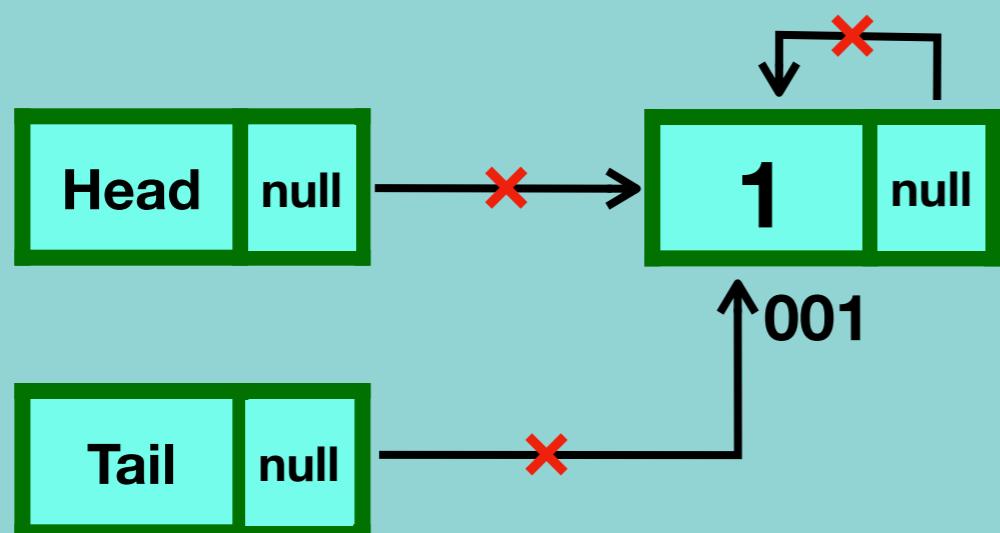
Circular Singly Linked List - Searching



Circular Singly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

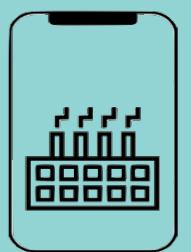
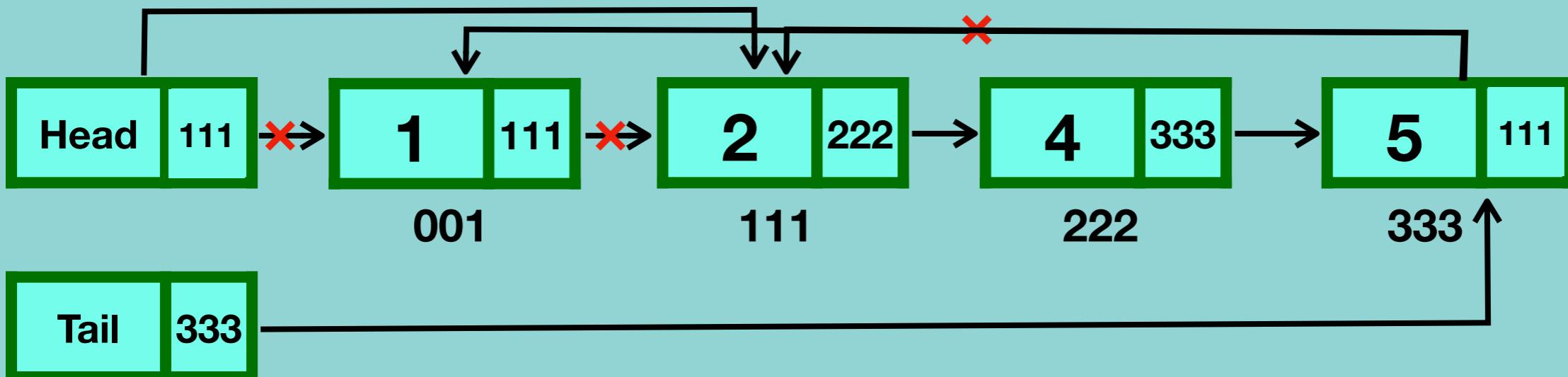
Case 1 - one node



Circular Singly Linked list - Deletion

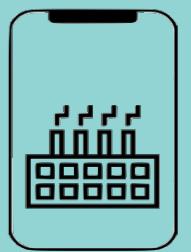
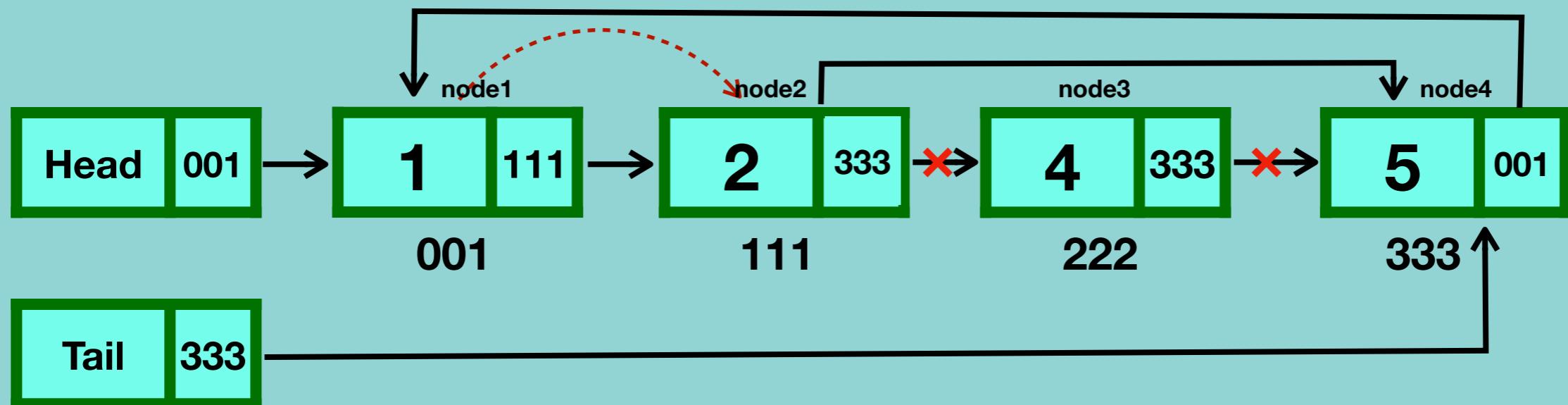
- Deleting the first node
- Deleting any given node
- Deleting the last node

Case 2 - more than one node



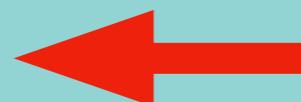
Circular Singly Linked list - Deletion

- Deleting the first node
- Deleting any given node ←
- Deleting the last node

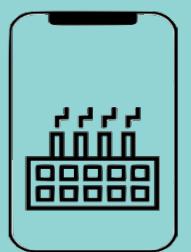
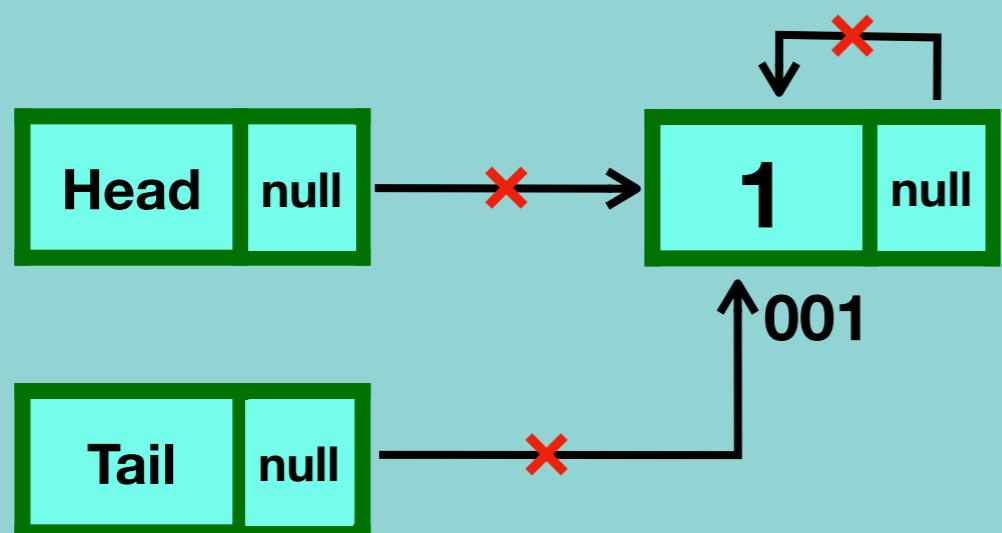


Circular Singly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

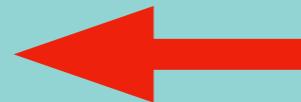


Case 1 - one node

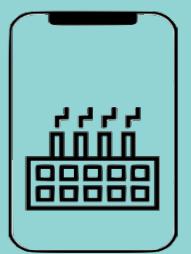
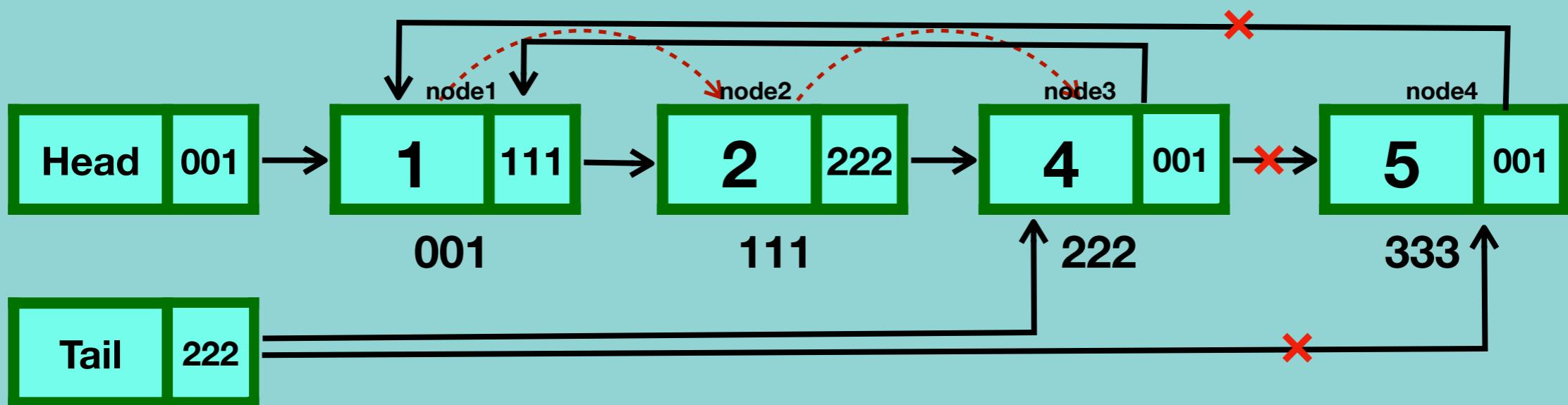


Circular Singly Linked list - Deletion

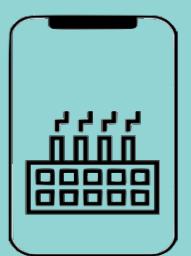
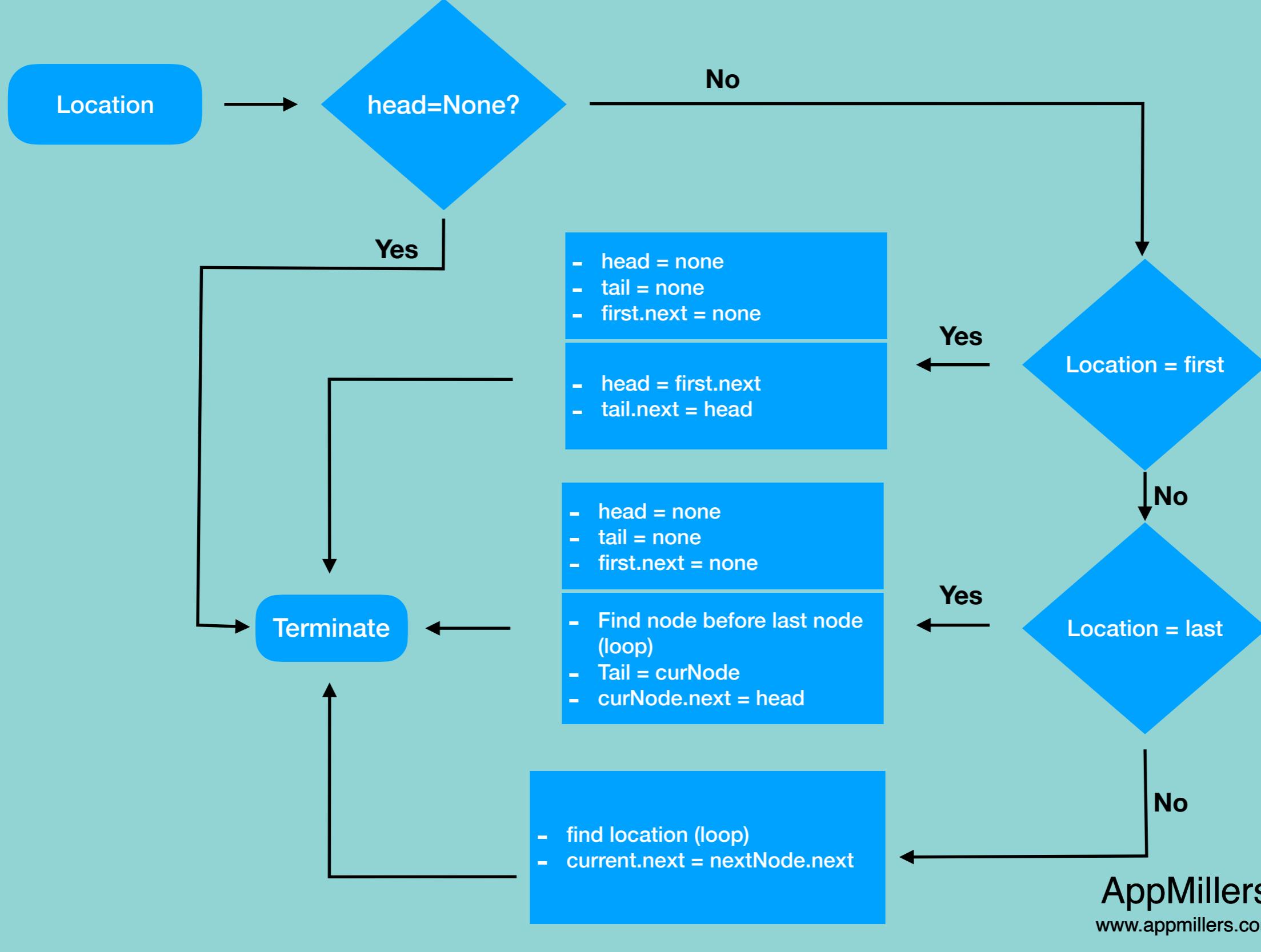
- Deleting the first node
- Deleting any given node
- Deleting the last node



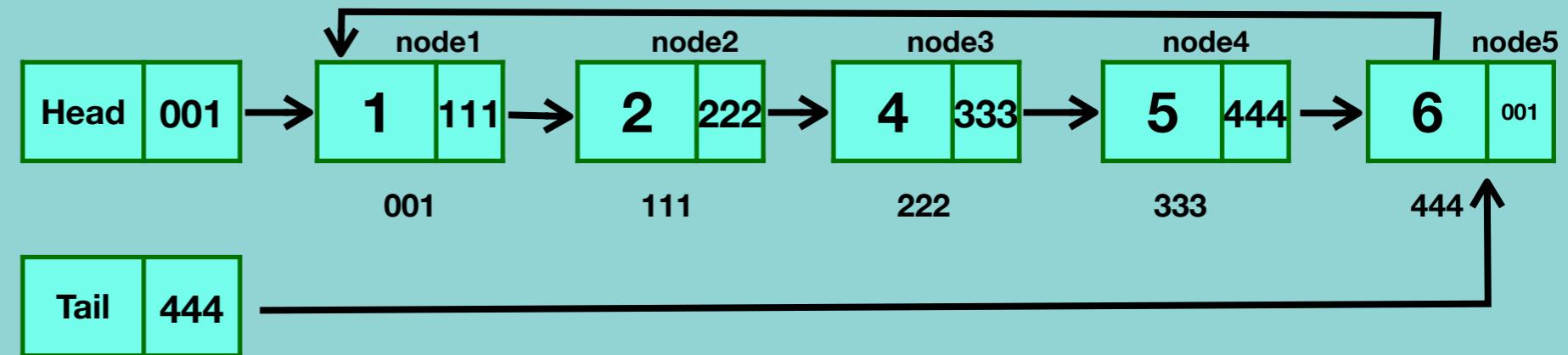
Case 2 - more than one node



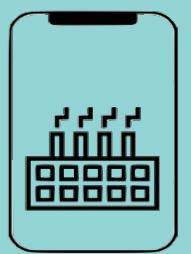
Circular Singly Linked list Deletion Algorithm



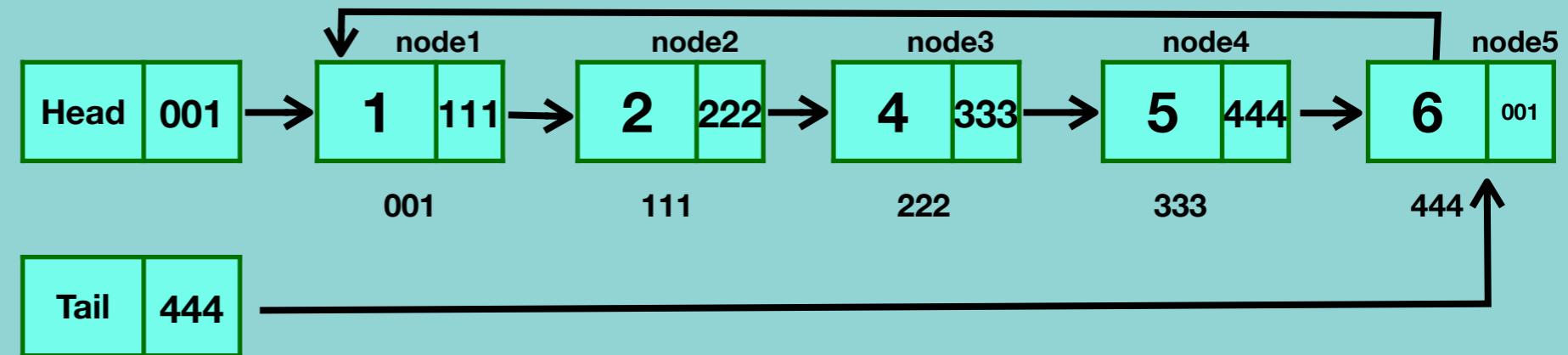
Circular Singly Linked list - Deletion Algorithm



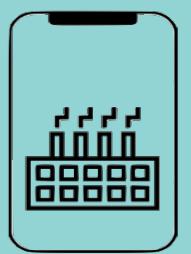
```
deleteNode(head, location):
    if head does not exist:
        return error //Linked list does not exist
    if location = firstNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            head=head.next
            tail.next = head
    else if location = lastNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            loop until lastNode location -1 (curNode)
                tail=curNode
                curNode.next = head
    else // delete middle node
        loop until location-1 (curNode)
            curNode.next = curNode.next.next
```



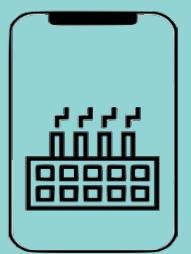
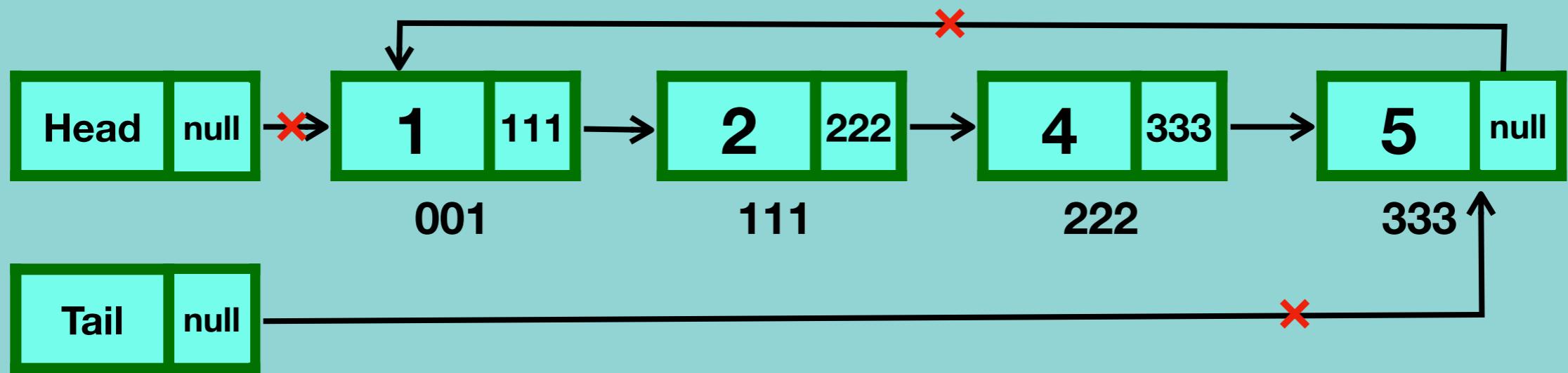
Circular Singly Linked list - Deletion Algorithm



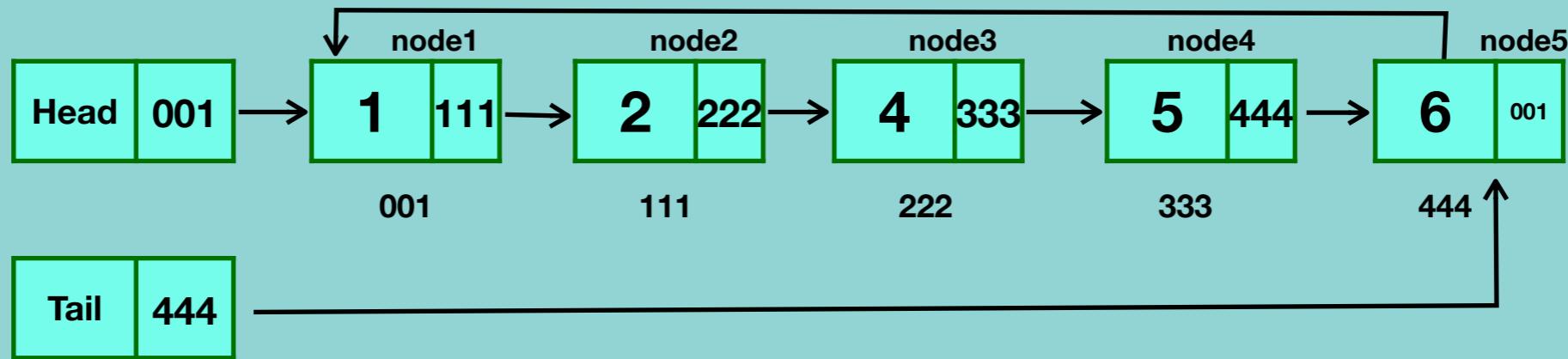
```
deleteNode(head, location):
    if head does not exist:
        return error //Linked list does not exist
    if location = firstNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            head=head.next
            tail.next = head
    else if location = lastNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            loop until lastNode location -1 (curNode)
                tail=curNode
                curNode.next = head
    else // delete middle node
        loop until location-1 (curNode)
            curNode.next = curNode.next.next
```



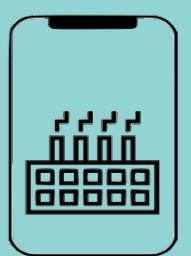
Delete entire Circular Singly Linked list



Delete entire Circular Singly Linked List

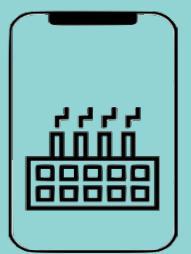
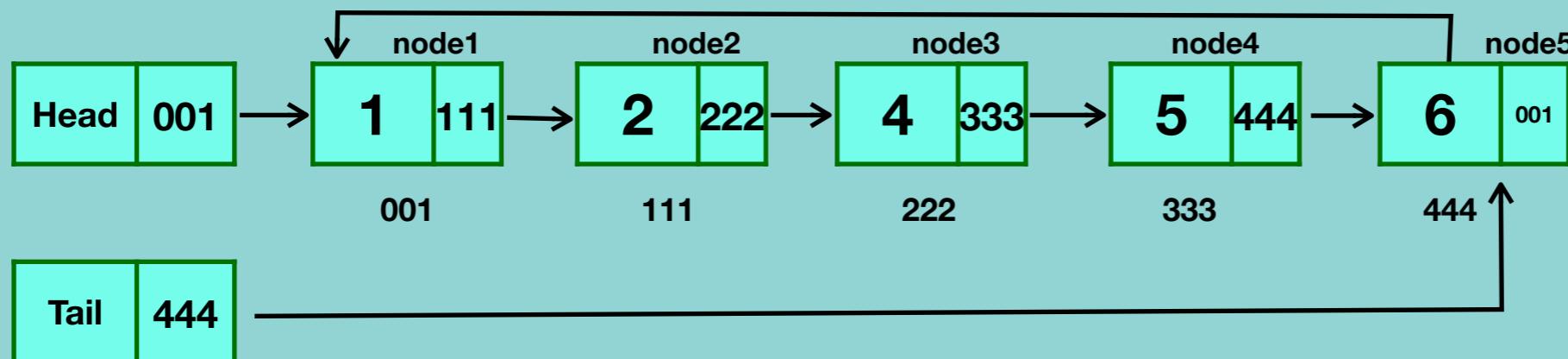


```
deleteLinkedList(head, tail):  
    head = null  
    tail.next = null  
    Tail = null
```



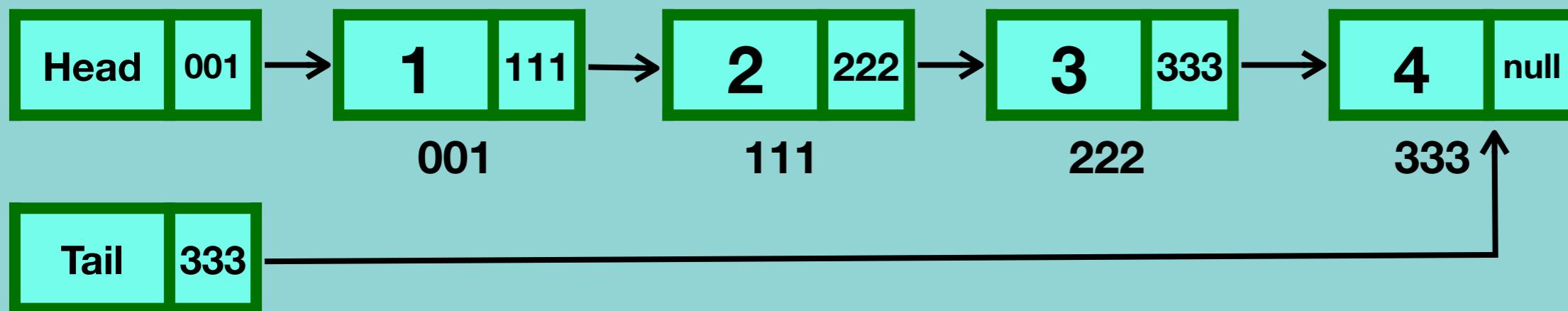
Time and Space complexity of Circular Singly Linked List

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(1)	O(1)

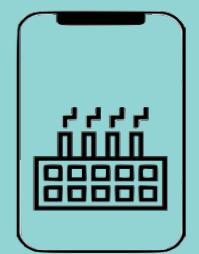
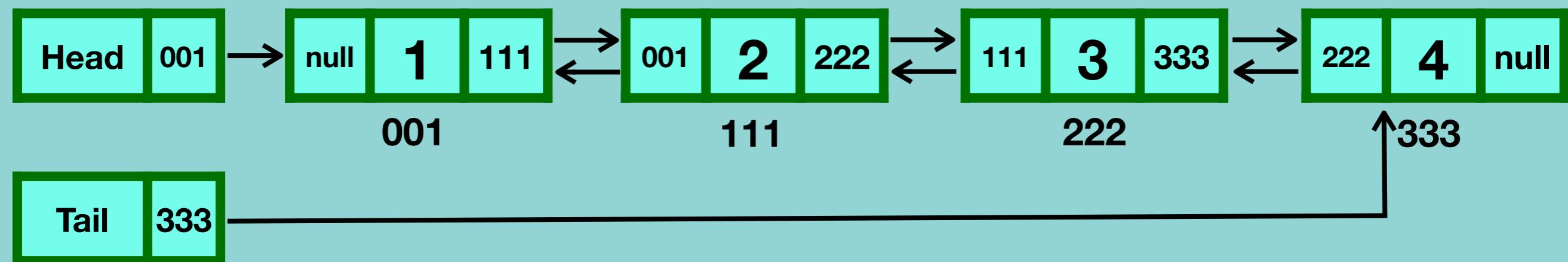


Doubly Linked List

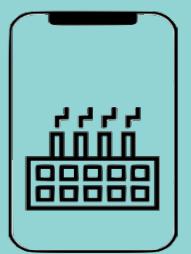
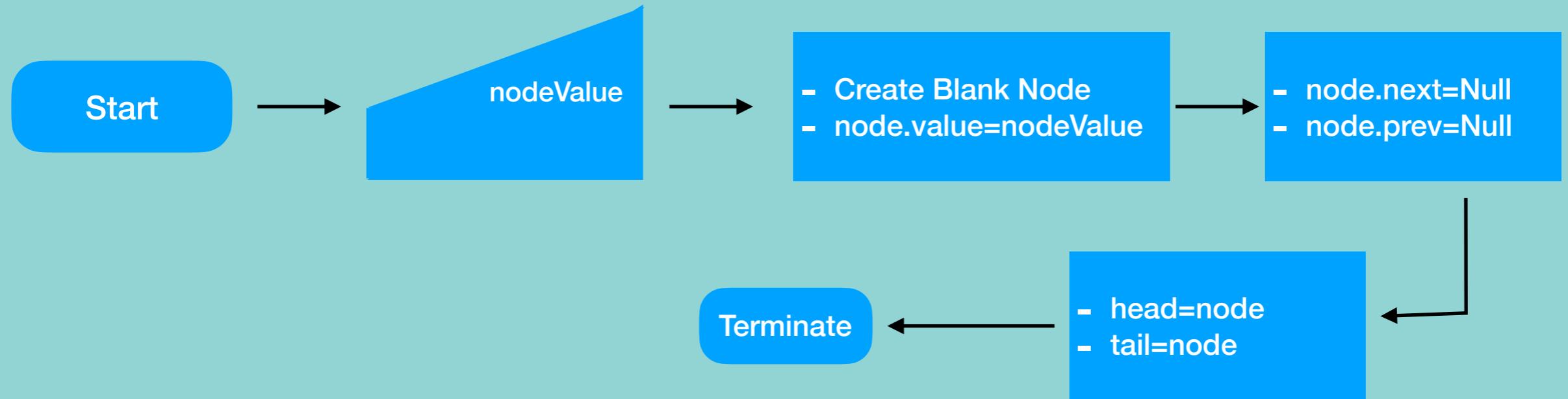
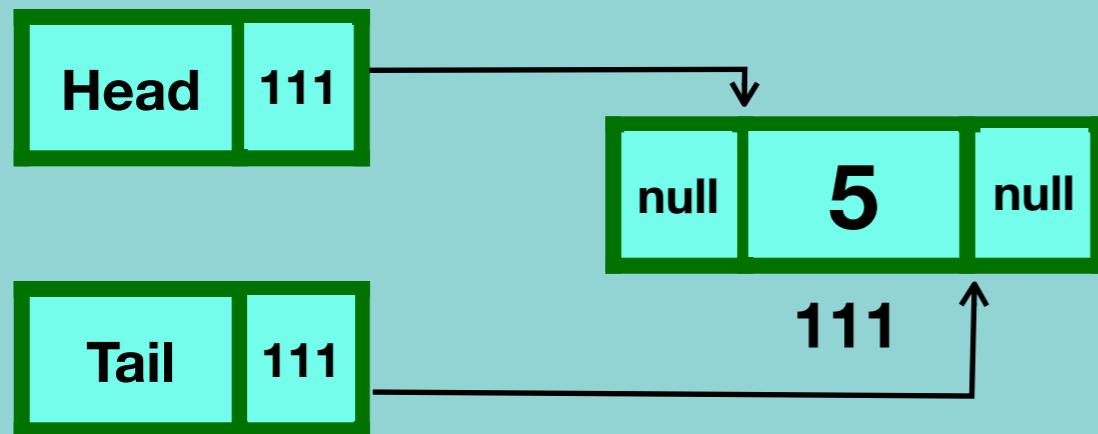
Singly Linked List



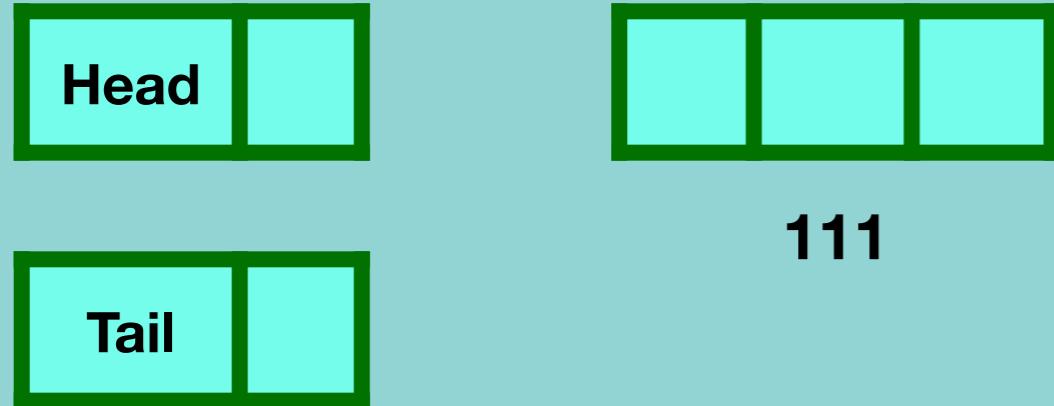
Doubly Linked List



Creation of Doubly Linked List



Doubly Linked List

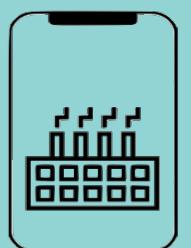


```
createDoublyLinkedList(nodeValue):
```

```
    create a blank node  
    node.value =nodeValue  
    head = node  
    tail = node  
    node.next = node.prev = null
```

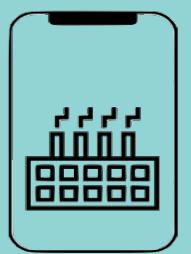
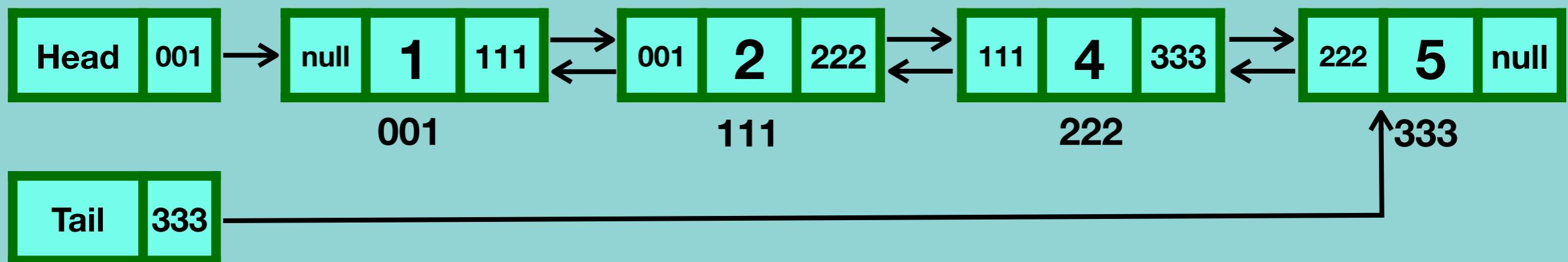
Time complexity : O(1)

Space complexity : O(1)



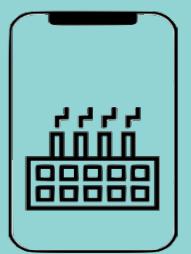
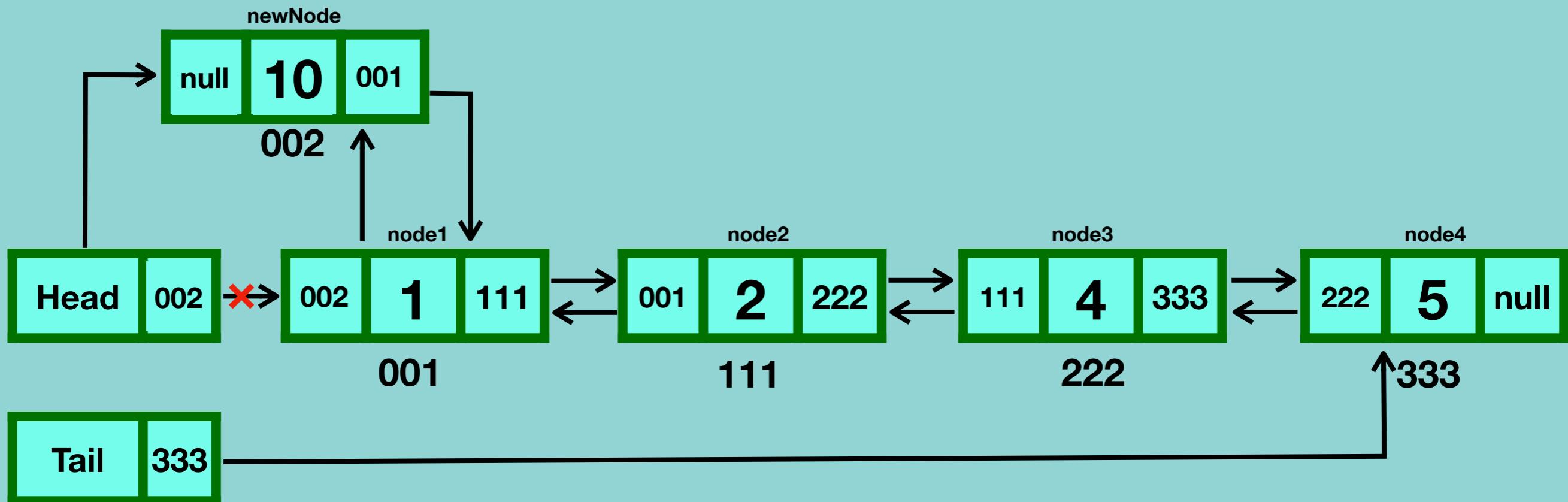
Doubly Linked List - Insertion

- Insert at the beginning of linked list
- Insert at the specified location of linked list
- Insert at the end of linked list



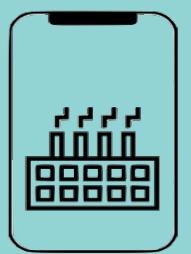
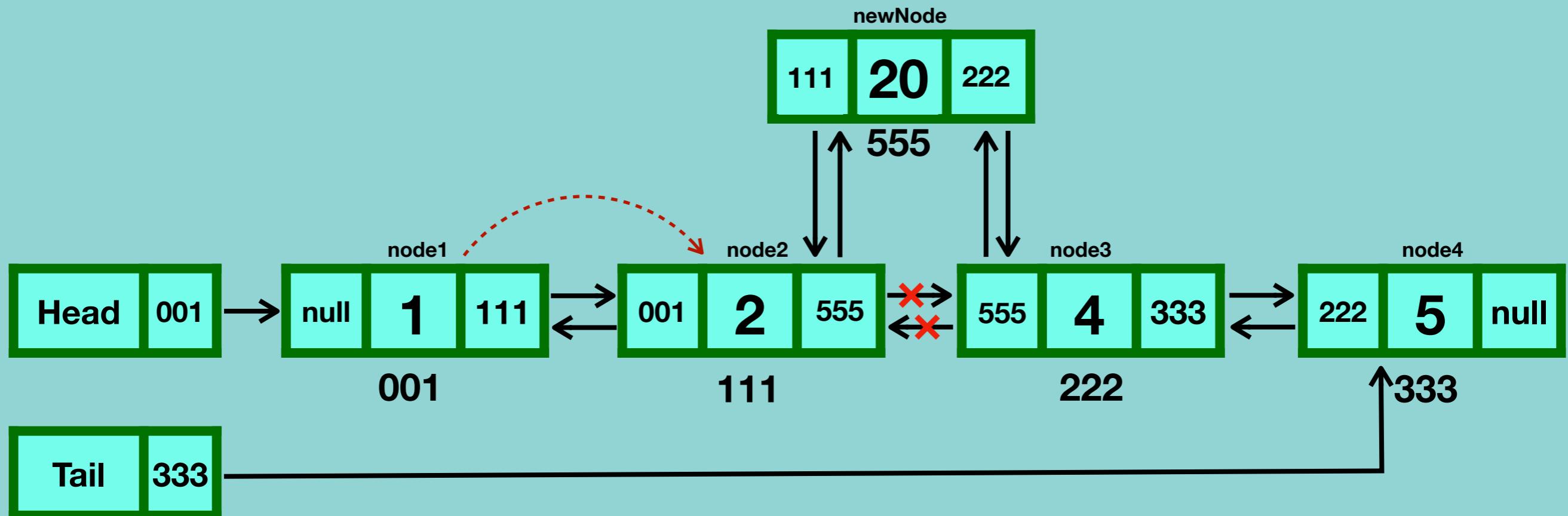
Doubly Linked List - Insertion

- Insert at the beginning of linked list



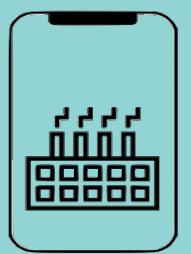
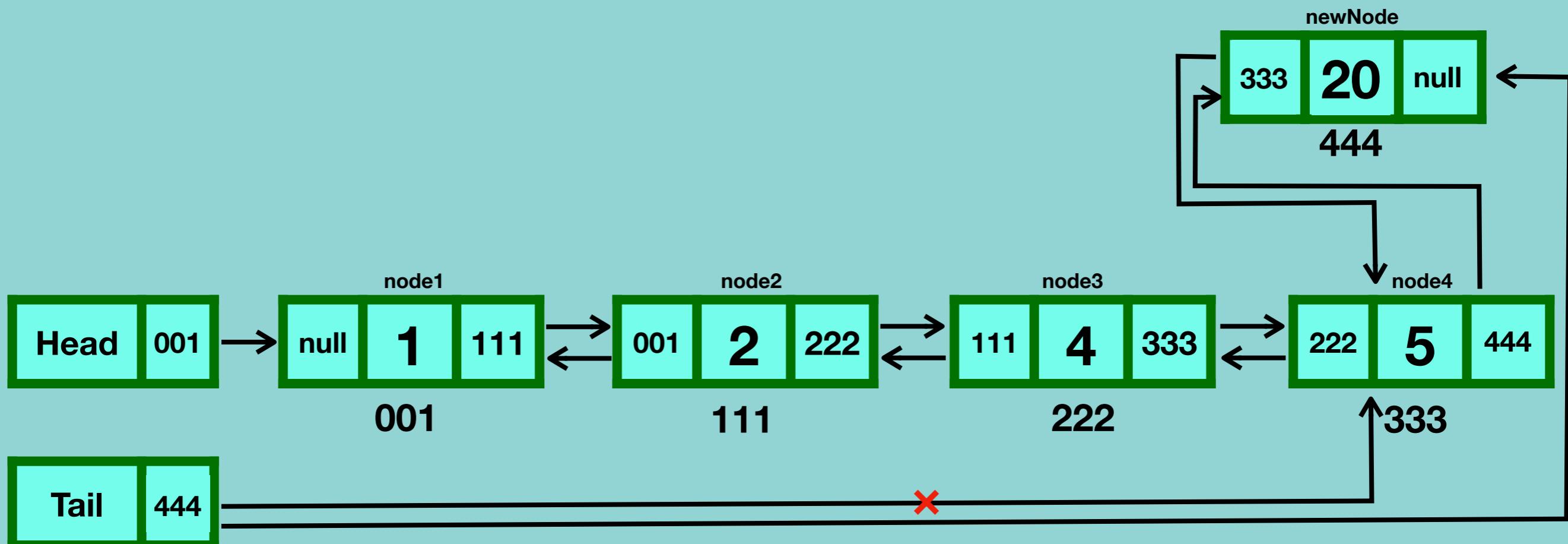
Doubly Linked List - Insertion

- Insert at the specified location of linked list

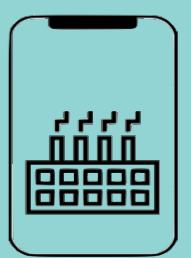
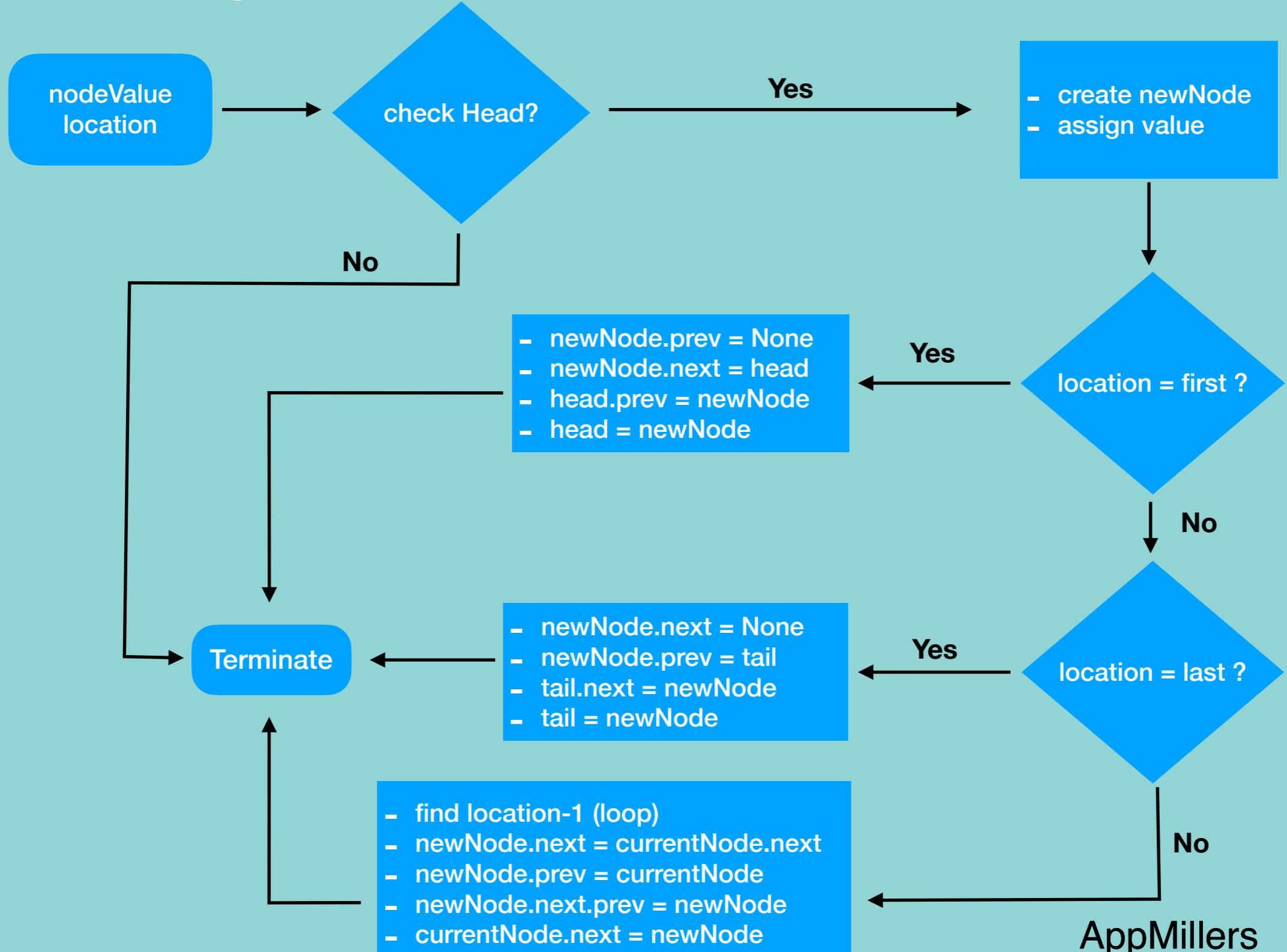


Doubly Linked List - Insertion

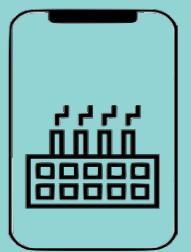
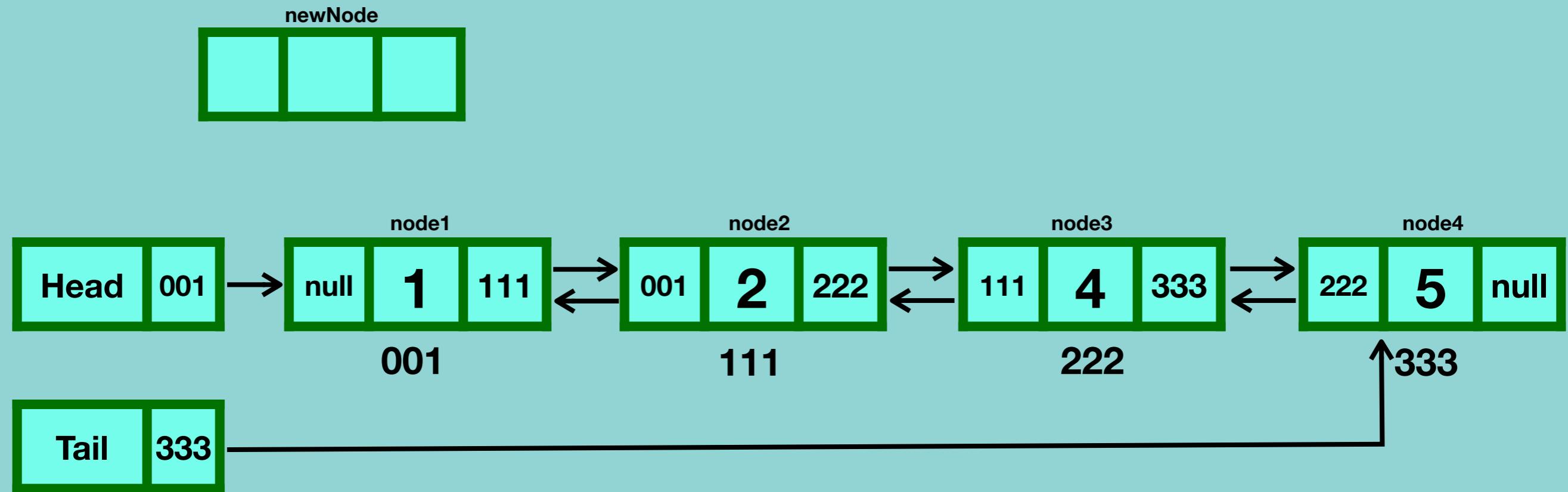
- Insert at the end of linked list



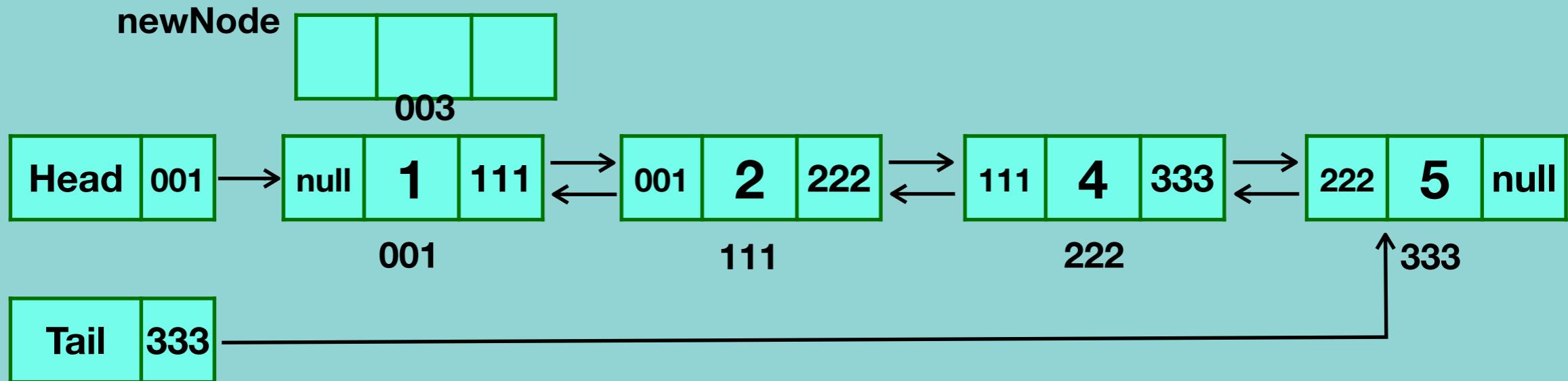
Insertion Algorithm - Doubly linked list



Insertion in Doubly Linked List



Insertion Algorithm - Doubly Linked List

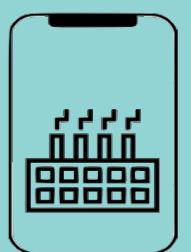


```
insertInDoublyLinkedList(head, nodeValue, location):
```

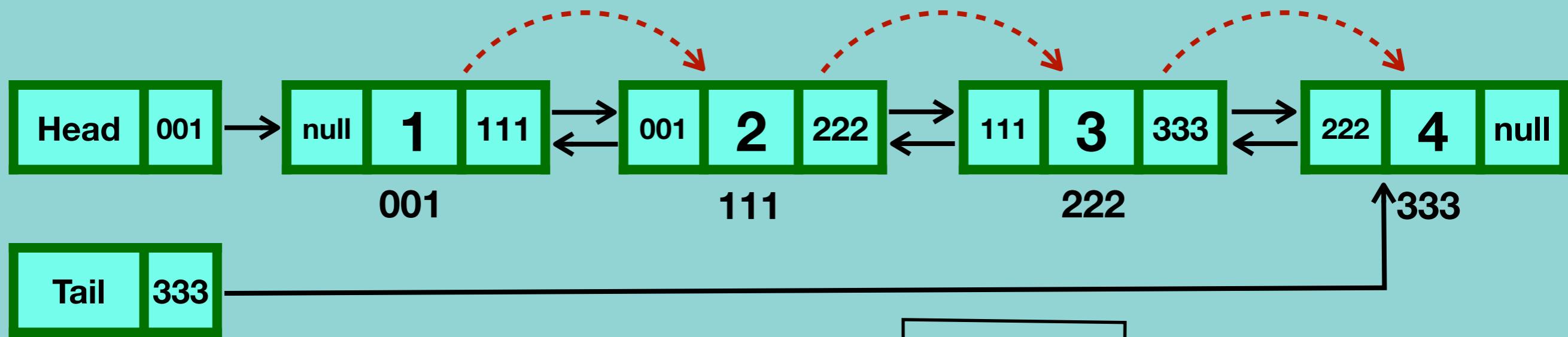
```
if head does not exist:  
    return error //Linked list does not exist  
else  
    create blank node (newNode)  
    newNode.value = nodeValue  
if location = firstNode's location  
    newNode.prev = null  
    newNode.next = head  
    head.prev = newNode  
    head = newNode  
else if location = lastNode's location  
    newNode.next = null  
    newNode.prev = tail  
    tail.next = node  
    tail = node  
else // delete middle node  
    loop until location-1 (curNode)  
        newNode.next = curNode.next  
        newNode.prev = curNode  
        curNode.next = newNode
```

Time complexity : O(1)

Space complexity : O(1)



Doubly Linked List - Traversal

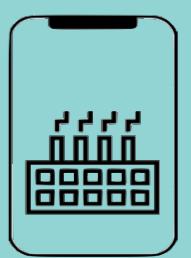
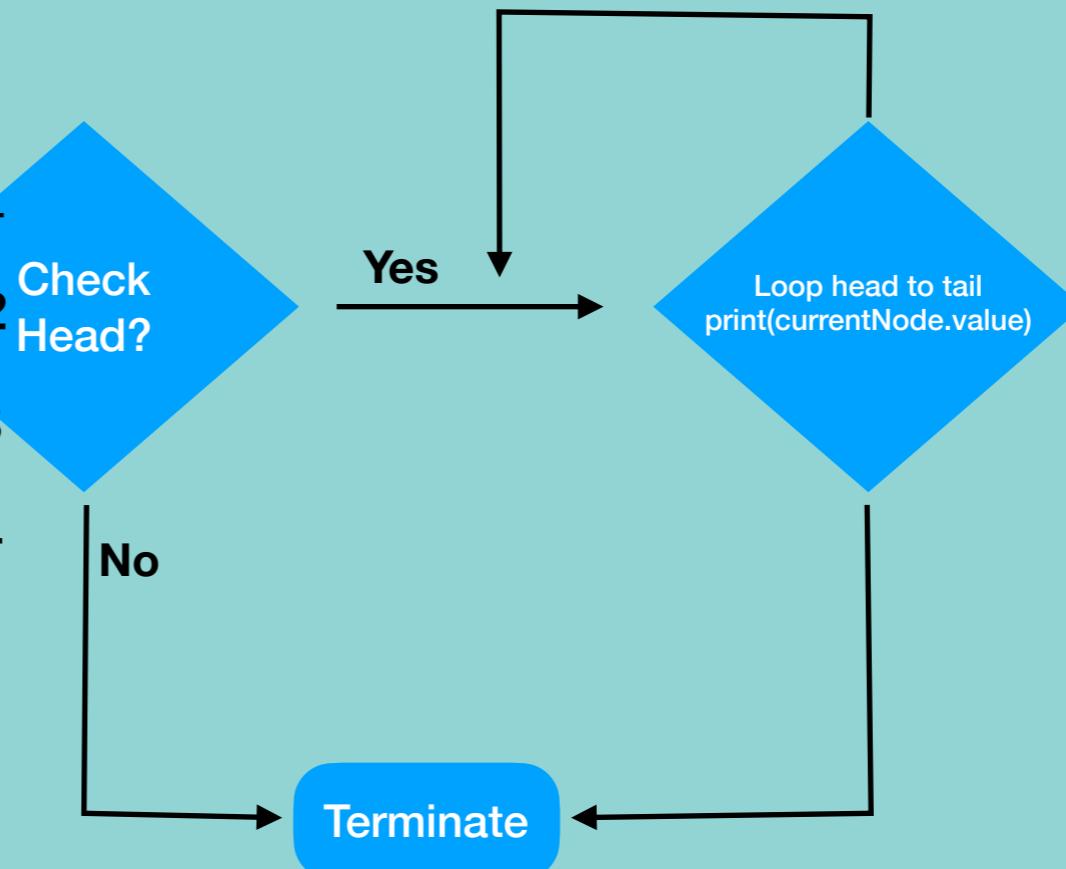


print(node1.value) → 1

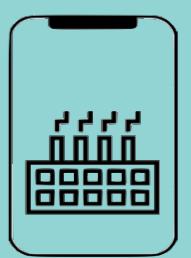
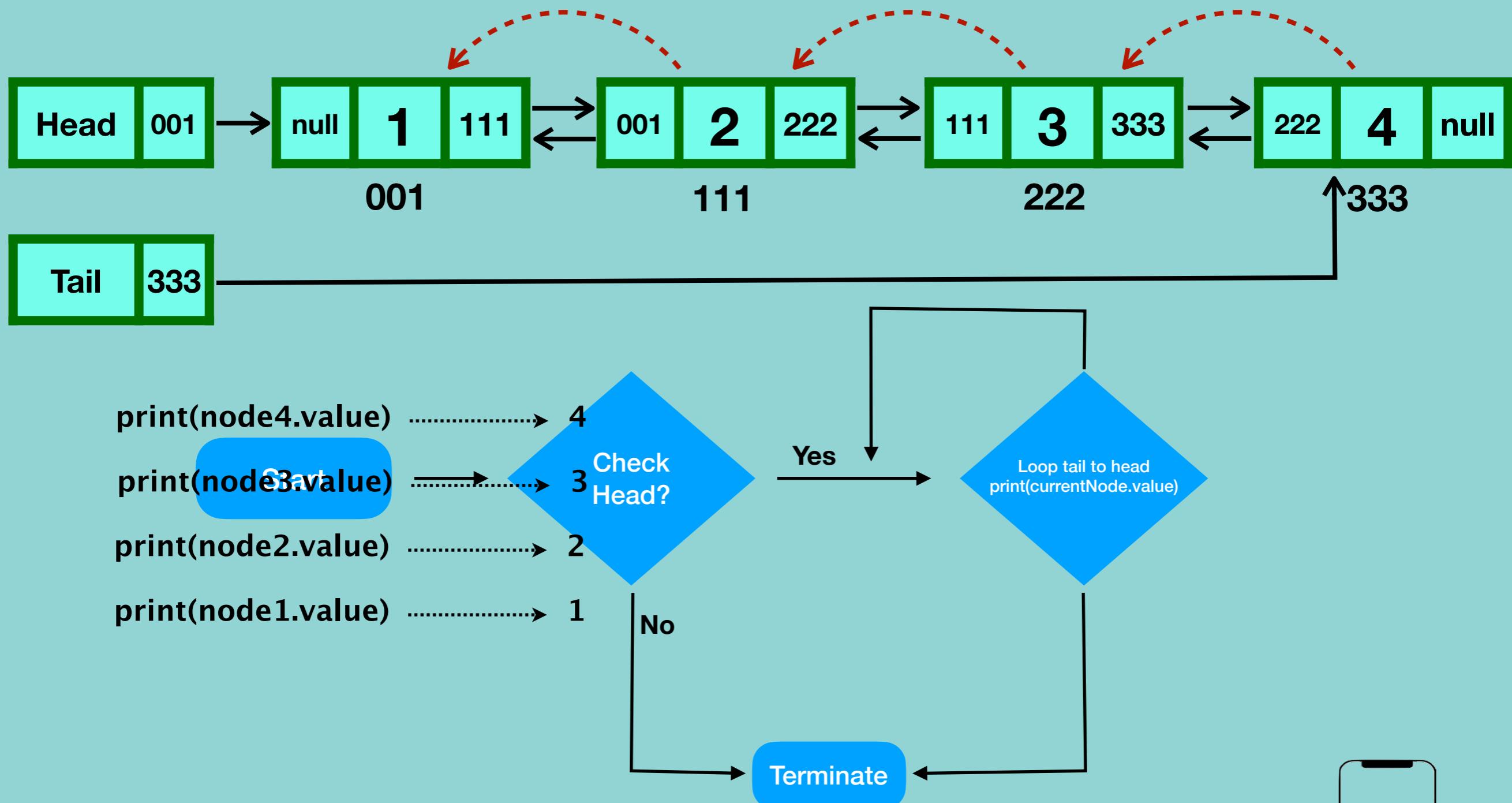
print(node2.value) → 2 Check Head?

print(node3.value) → 3

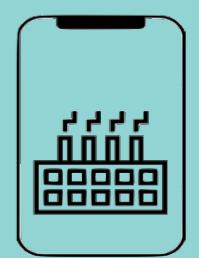
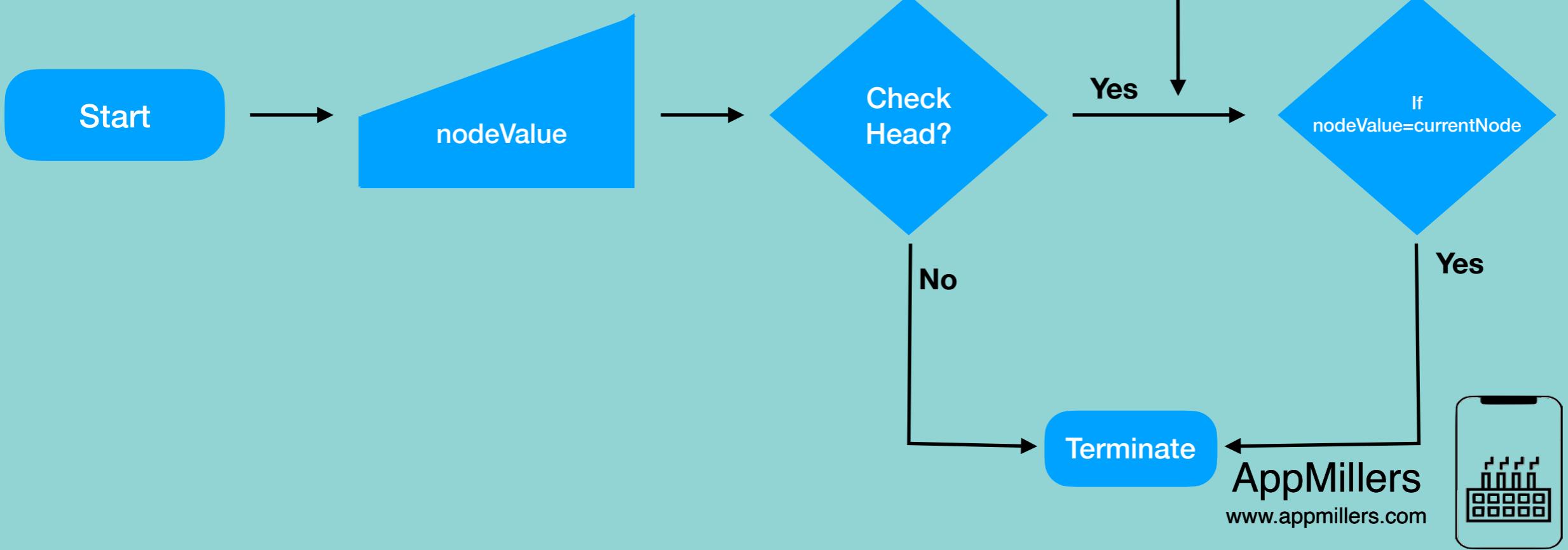
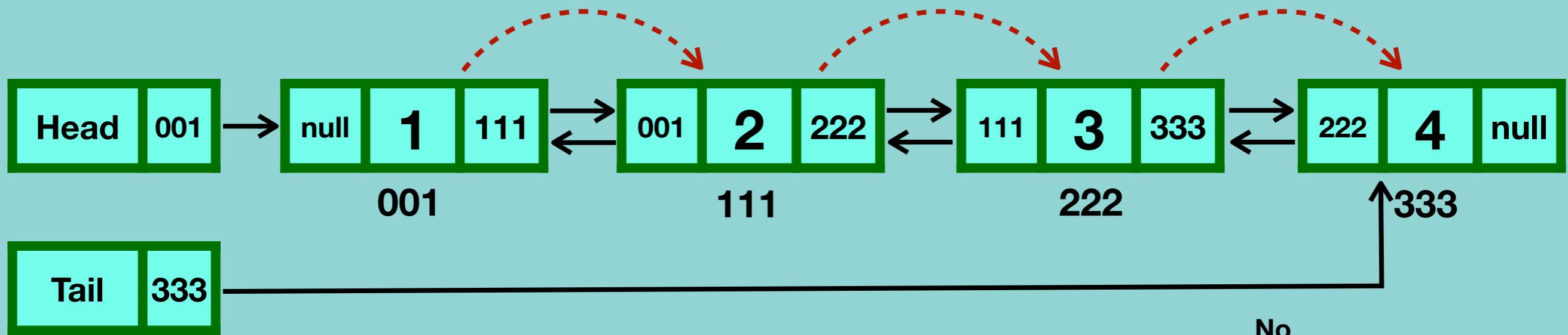
print(node4.value) → 4



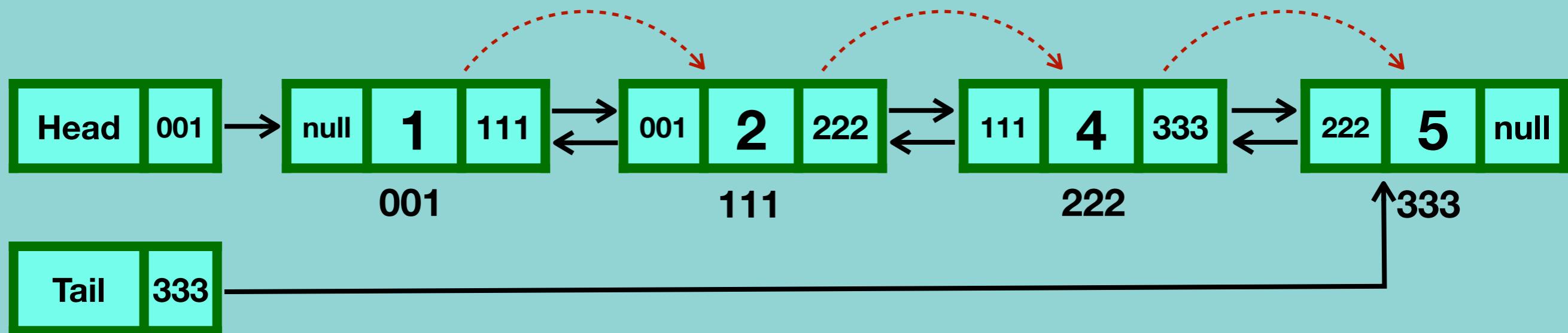
Doubly Linked List - Reverse Traversal



Doubly Linked List - Search



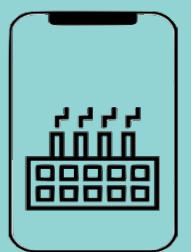
Traversal in doubly linked list



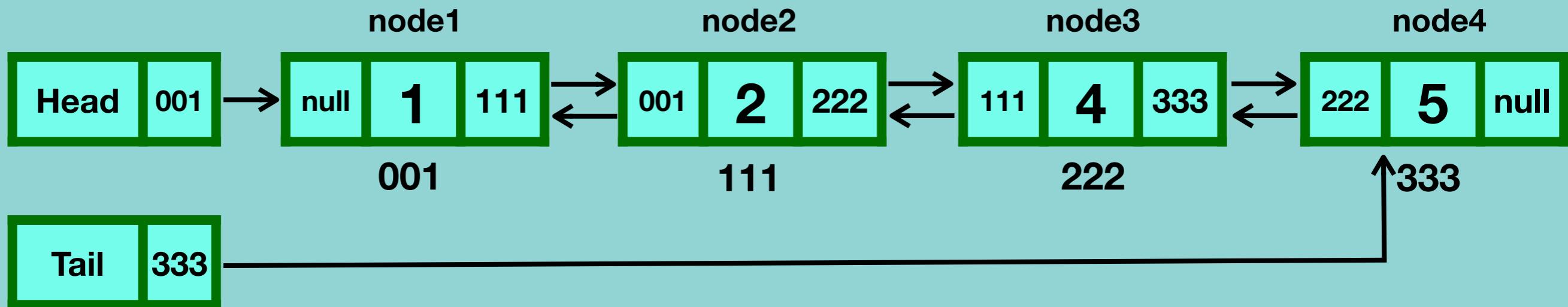
```
traversalDoublyLinkedList():
    if head == null:
        return //There is not any node in this list
    loop head to tail:
        print(currentNode.value)
```

Time complexity : O(n)

Space complexity : O(1)



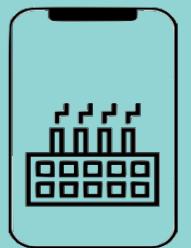
Reverse Traversal in doubly linked list



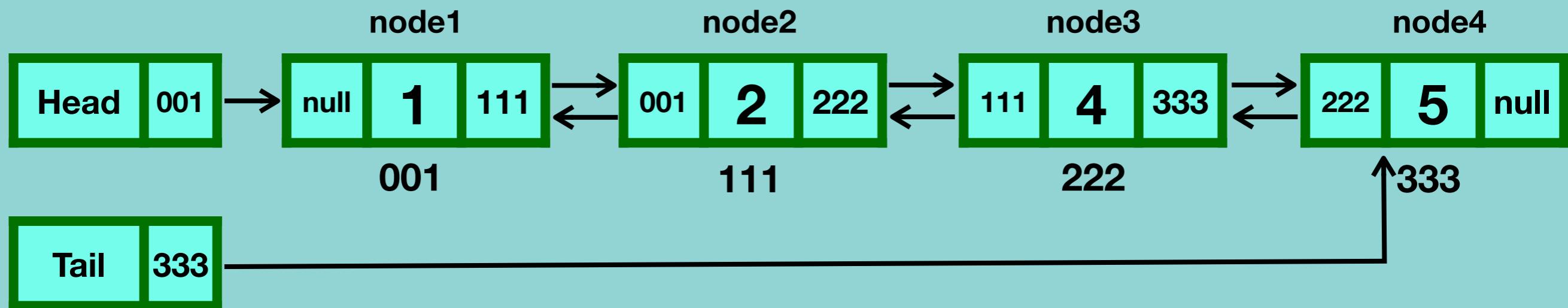
```
reverseTraversalDoublyLinkedList(head):
    if head == null:
        return //There is not any node in this list
    loop tail to head:
        print(currentNode.value)
```

Time complexity : O(1)

Space complexity : O(1)



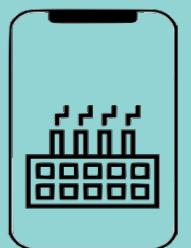
Searching a node in doubly linked list



```
searchForNode(head, nodeValue):
    loop head to tail:
        if currentNode.value = nodeValue
            print(currentNode)
            return
    return // nodeValue not found
```

Time complexity : $O(n)$

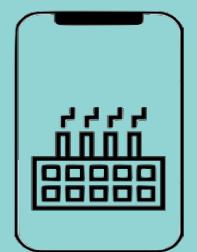
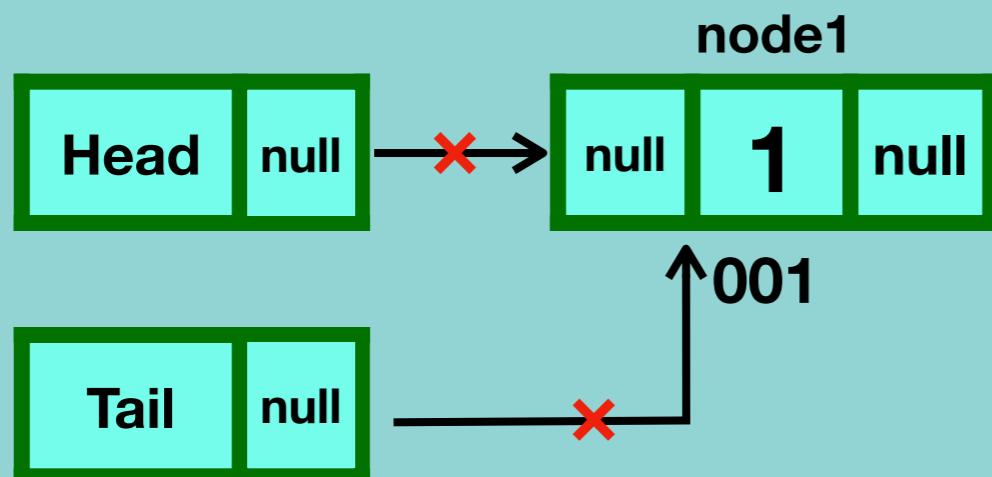
Space complexity : $O(1)$



Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

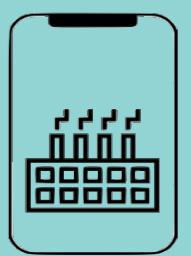
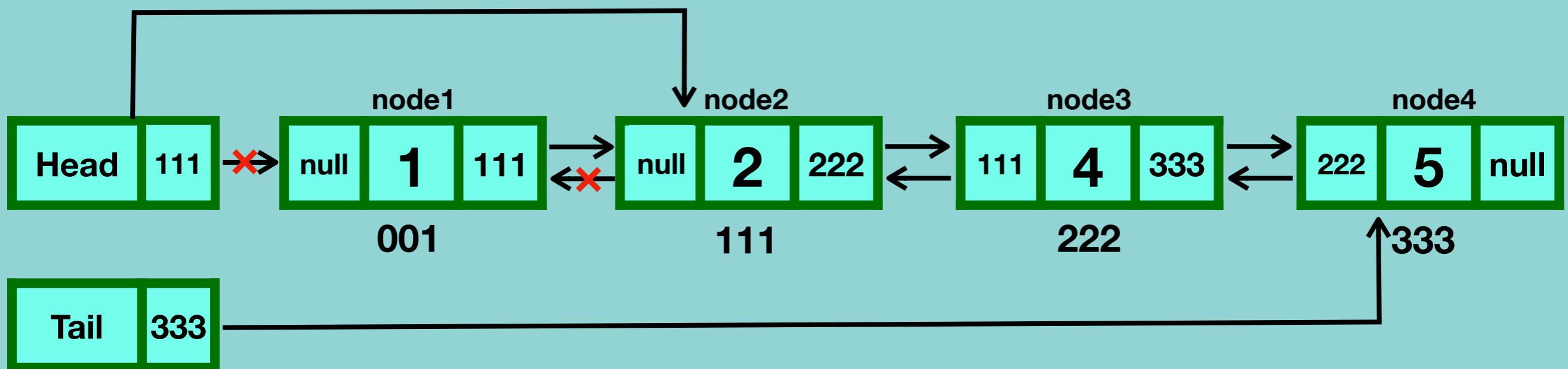
Case 1 - one node



Doubly Linked list - Deletion

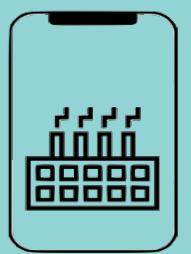
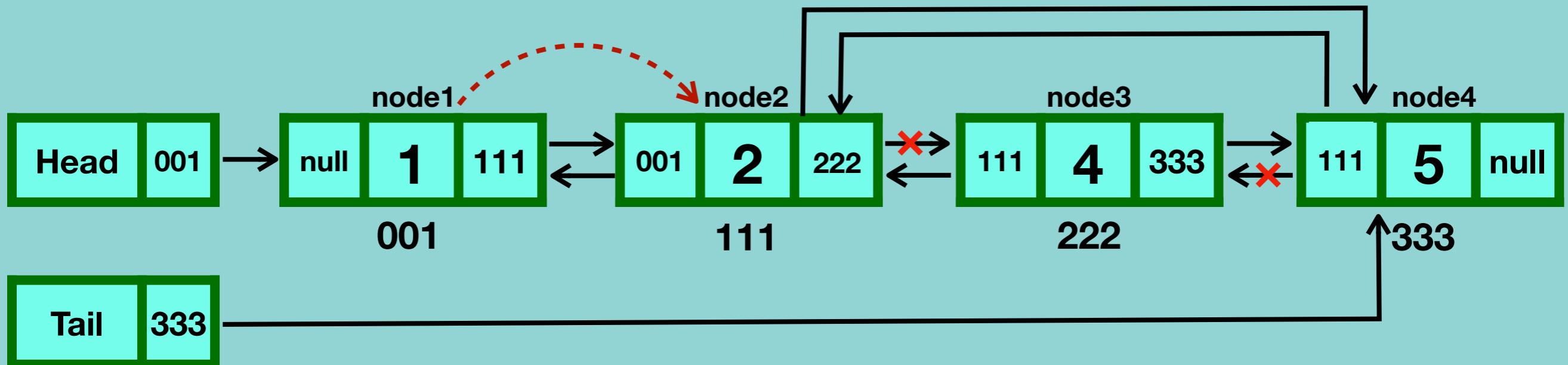
- Deleting the first node
- Deleting any given node
- Deleting the last node

Case 2 - more than one node



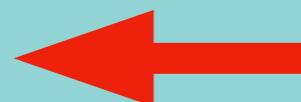
Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

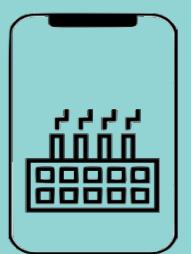
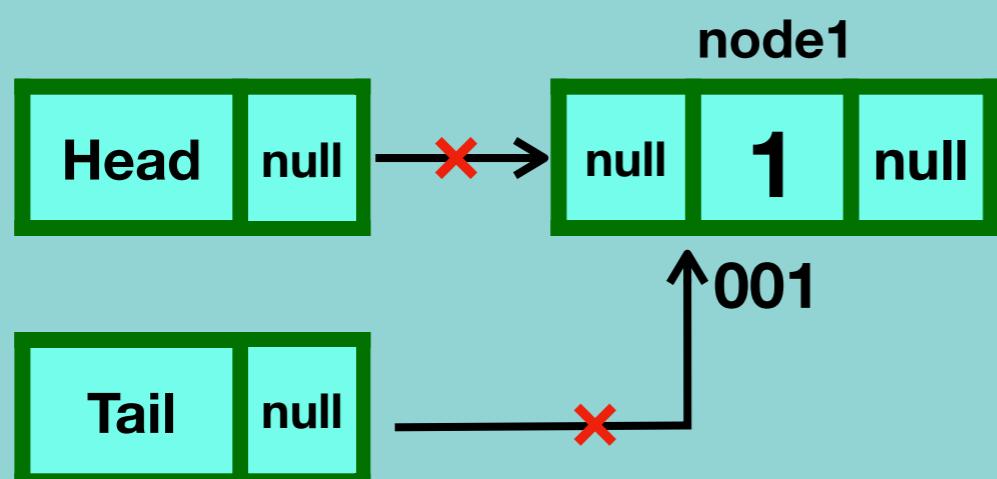


Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

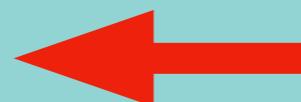


Case 1 - one node

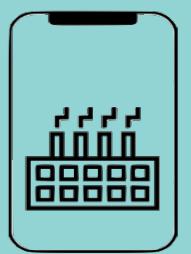
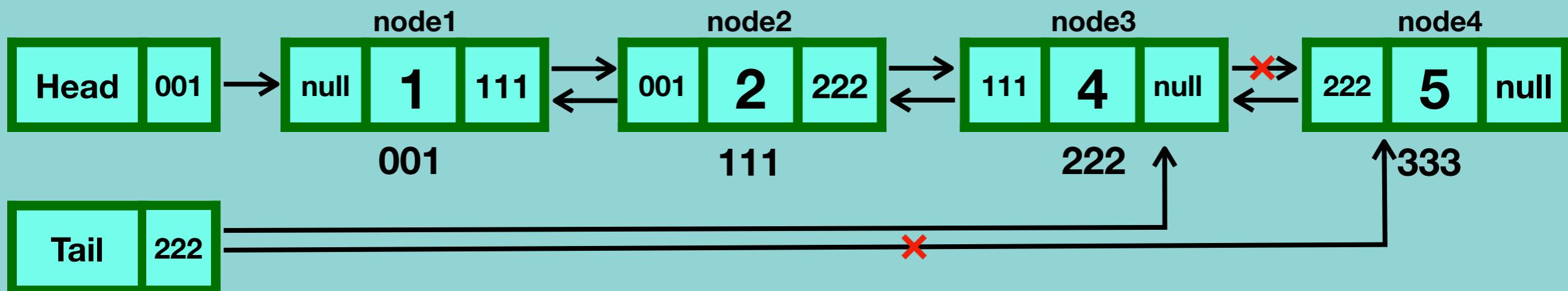


Doubly Linked list - Deletion

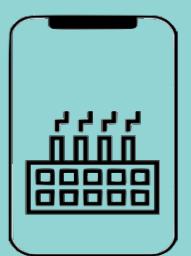
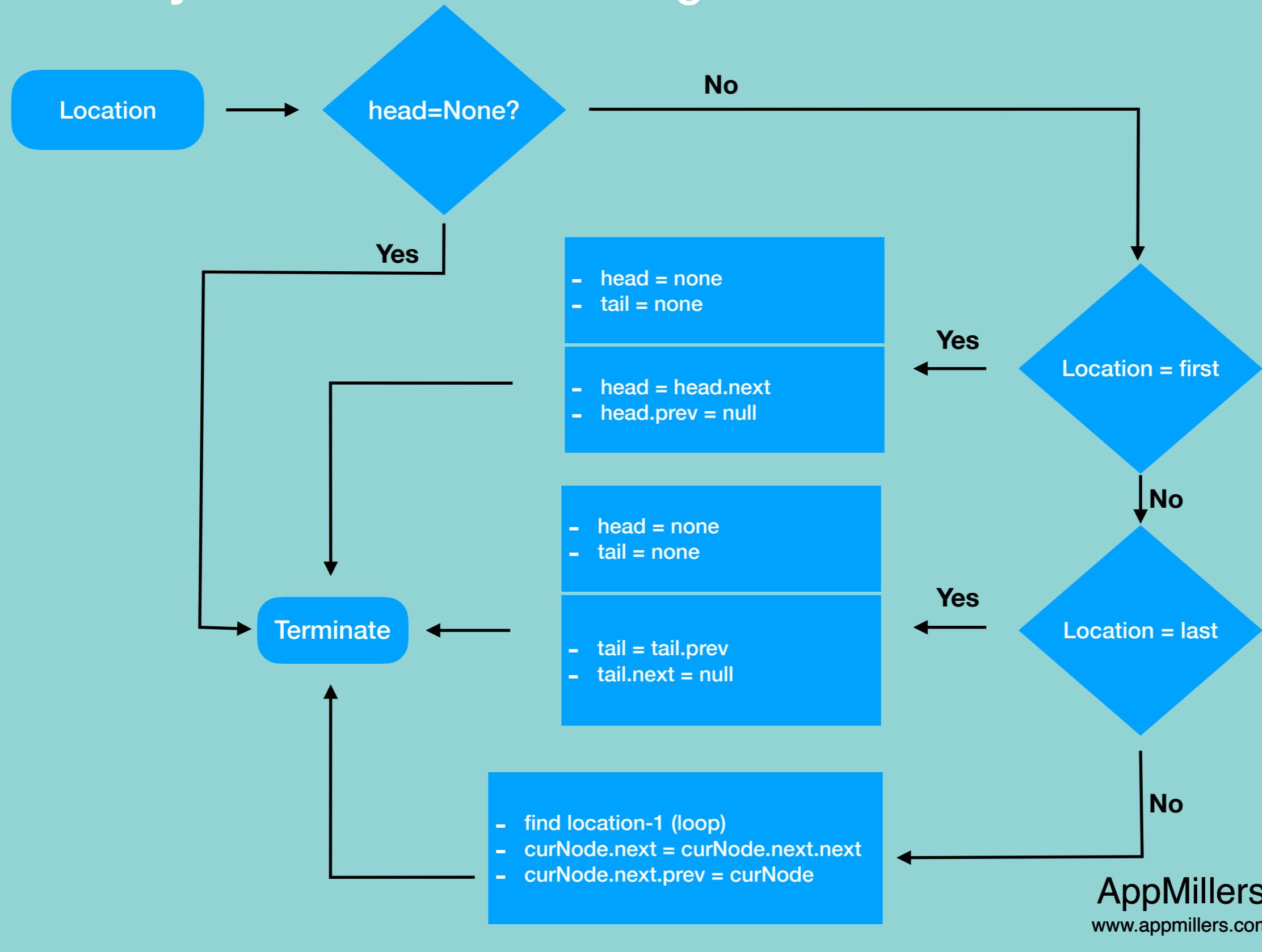
- Deleting the first node
- Deleting any given node
- Deleting the last node



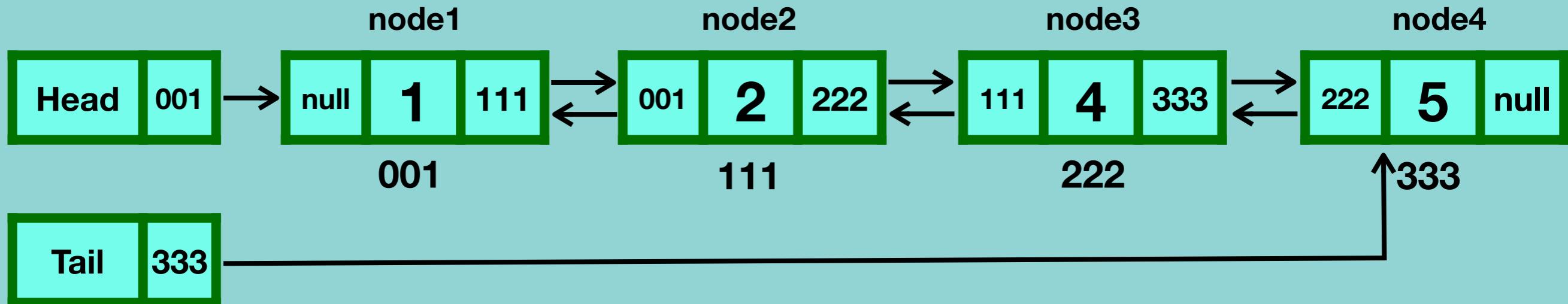
Case 2 - more than one node



Doubly Linked list Deletion Algorithm



Deleting a node in doubly linked list- Algorithm

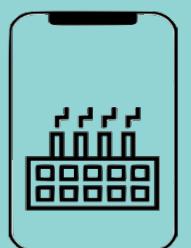


```
deleteNode(head, location):
```

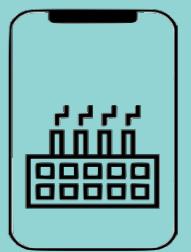
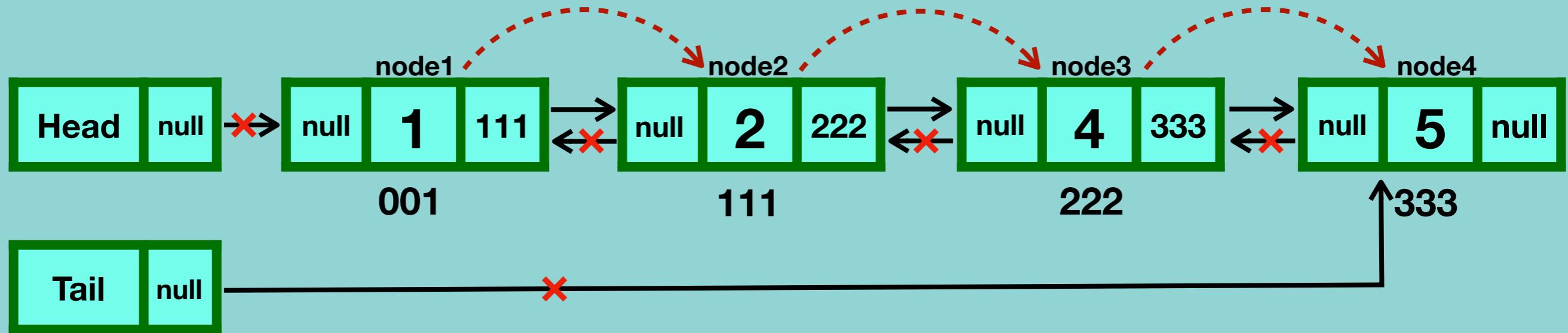
```
    if head does not exist:  
        return error //Linked list does not exist  
    if location = firstNode's location  
        if this is the only node in the list  
            head=tail=null  
        else  
            head=head.next  
            head.prev = null  
    else if location = lastNode's location  
        if this is the only node in the list  
            head=tail=null  
        else  
            loop until lastNode location -1 (curNode)  
                tail=curNode  
                curNode.next = head  
    else // delete middle node  
        loop until location-1 (curNode)  
            curNode.next = curNode.next.next  
            curNode.next.prev = curNode
```

Time complexity : O(n)

Space complexity : O(1)

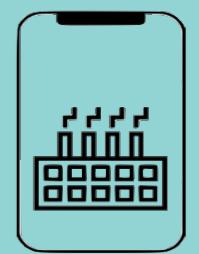
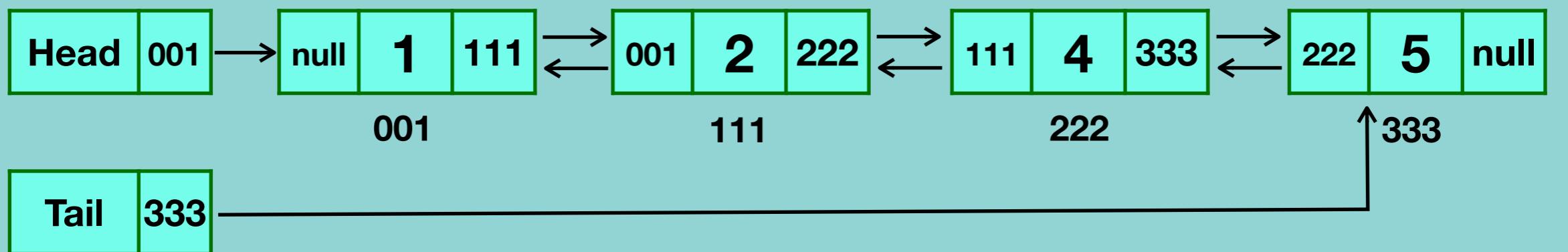


Deleting entire doubly linked list- Algorithm

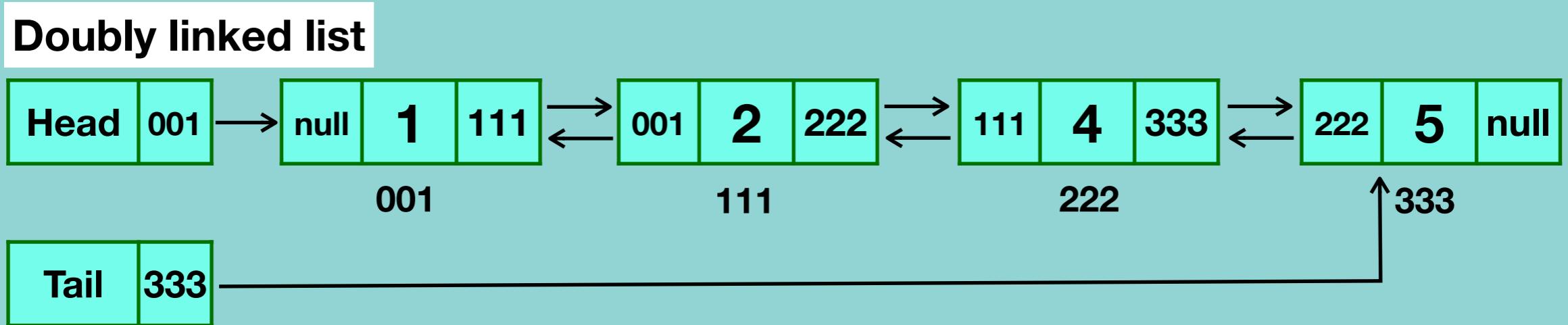


Time and Space complexity of doubly linked list

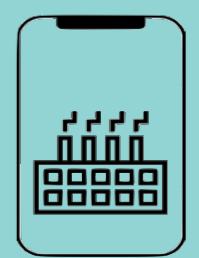
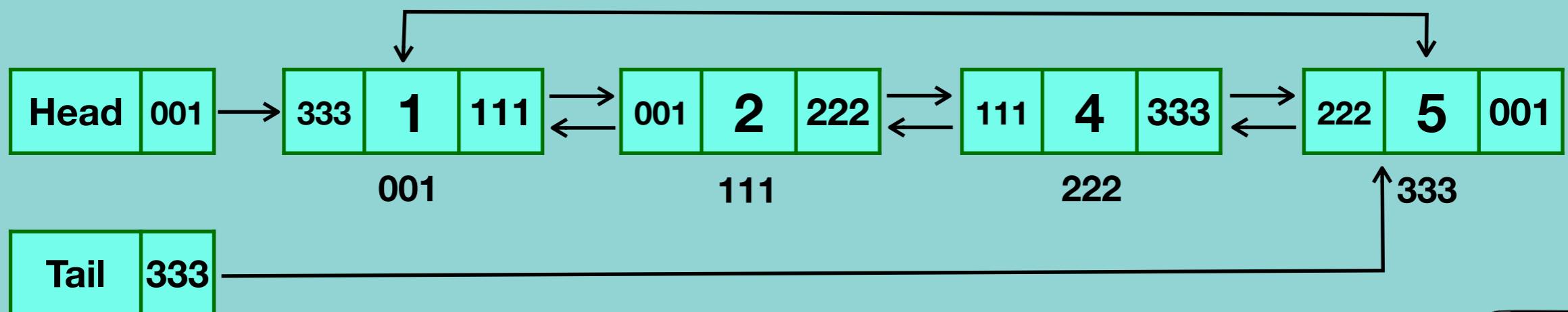
	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing (forward, backward)	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(n)	O(1)



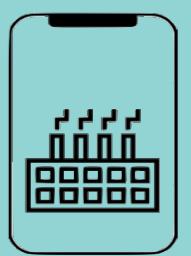
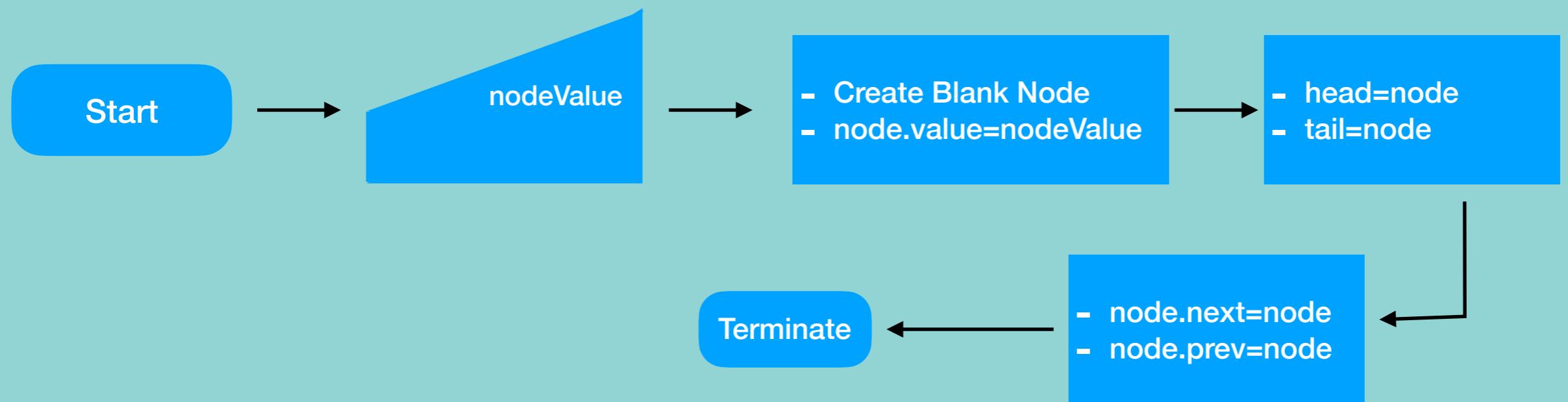
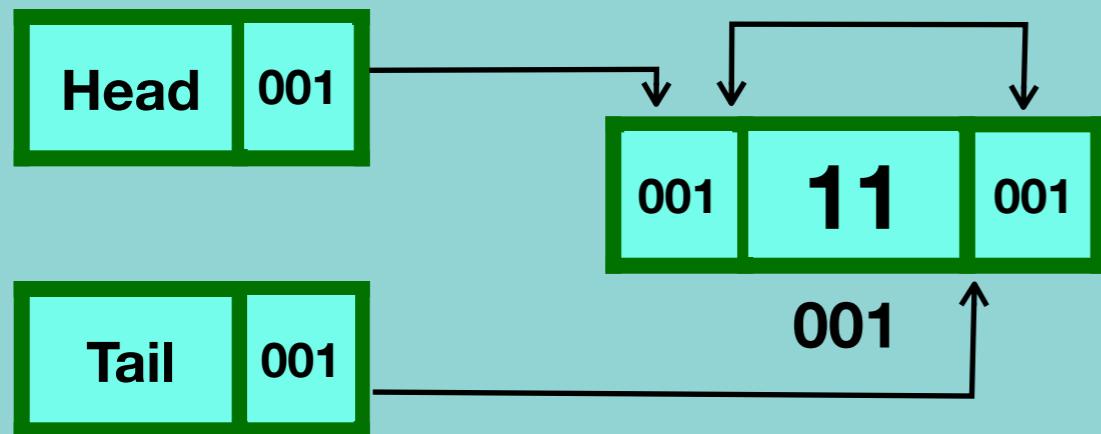
Circular doubly linked list



Circular Doubly linked list

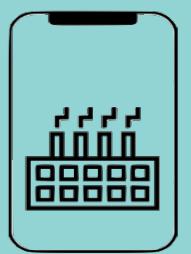
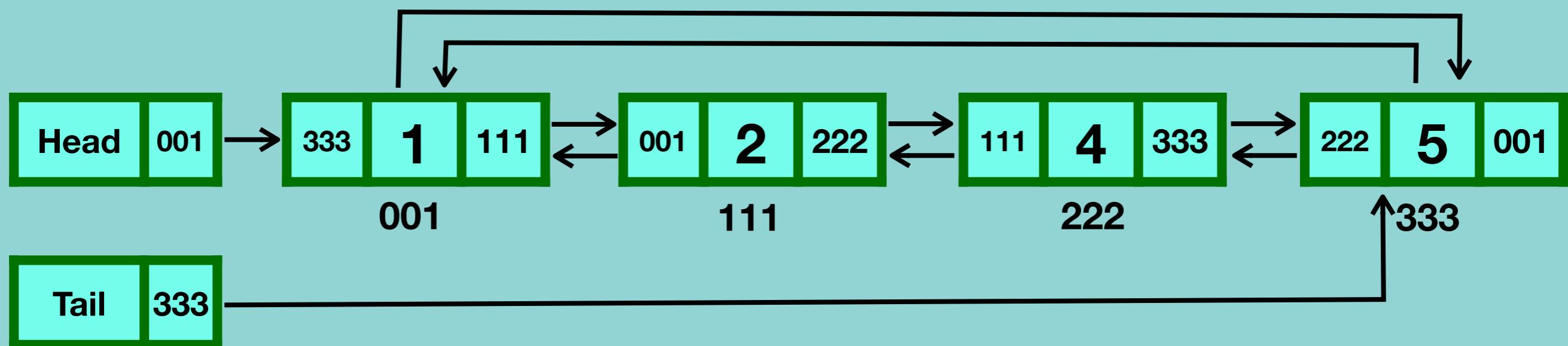


Creation of Circular Doubly Linked List



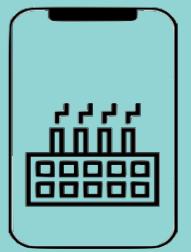
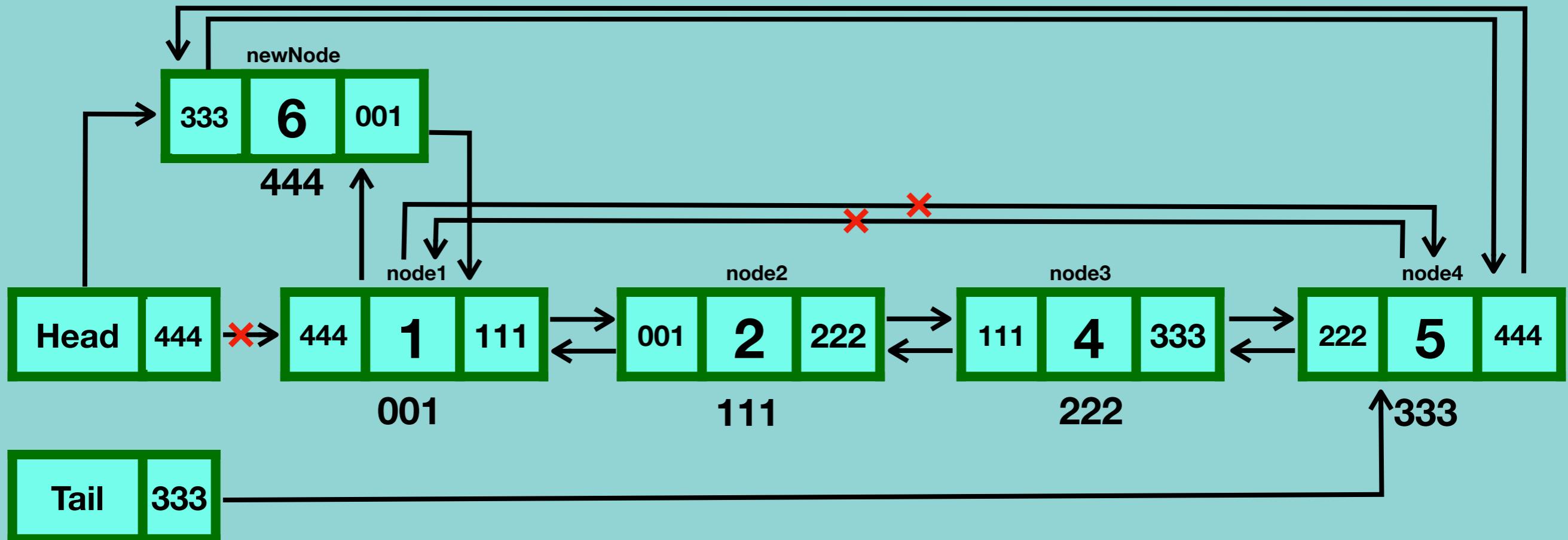
Circular Doubly Linked List - Insertion

- Insert at the beginning of linked list
- Insert at the specified location of linked list
- Insert at the end of linked list



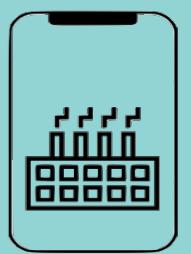
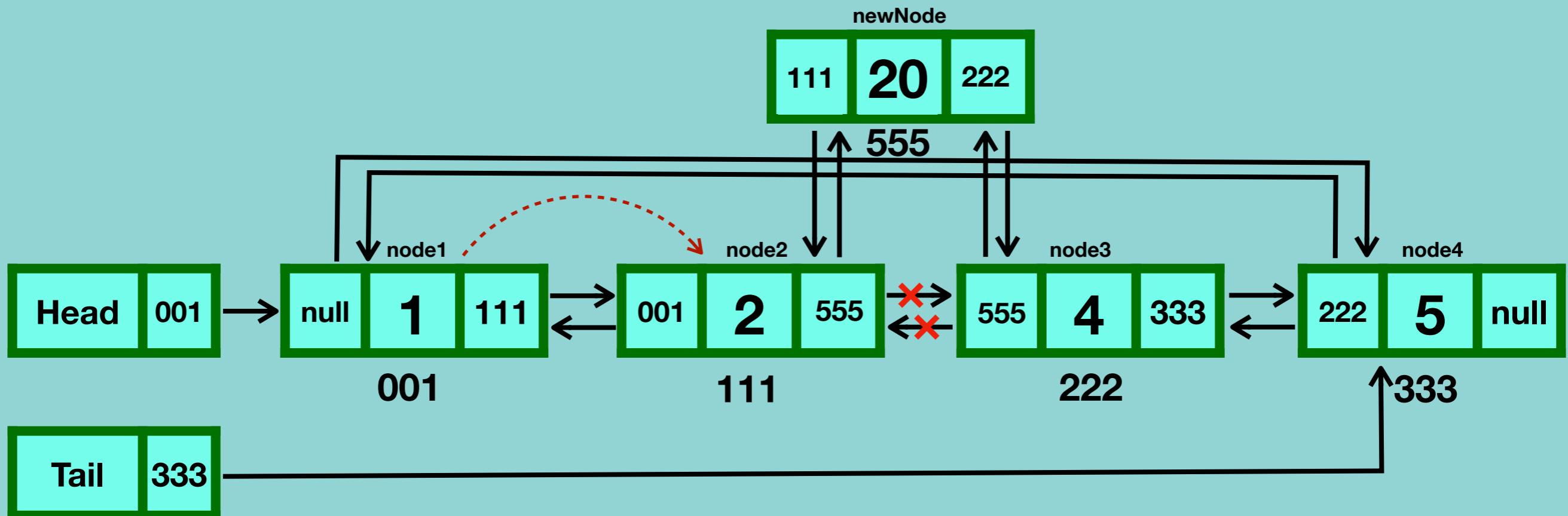
Circular Doubly Linked List - Insertion

- Insert at the beginning of linked list



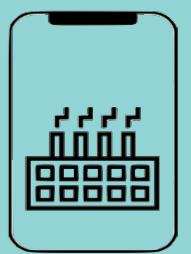
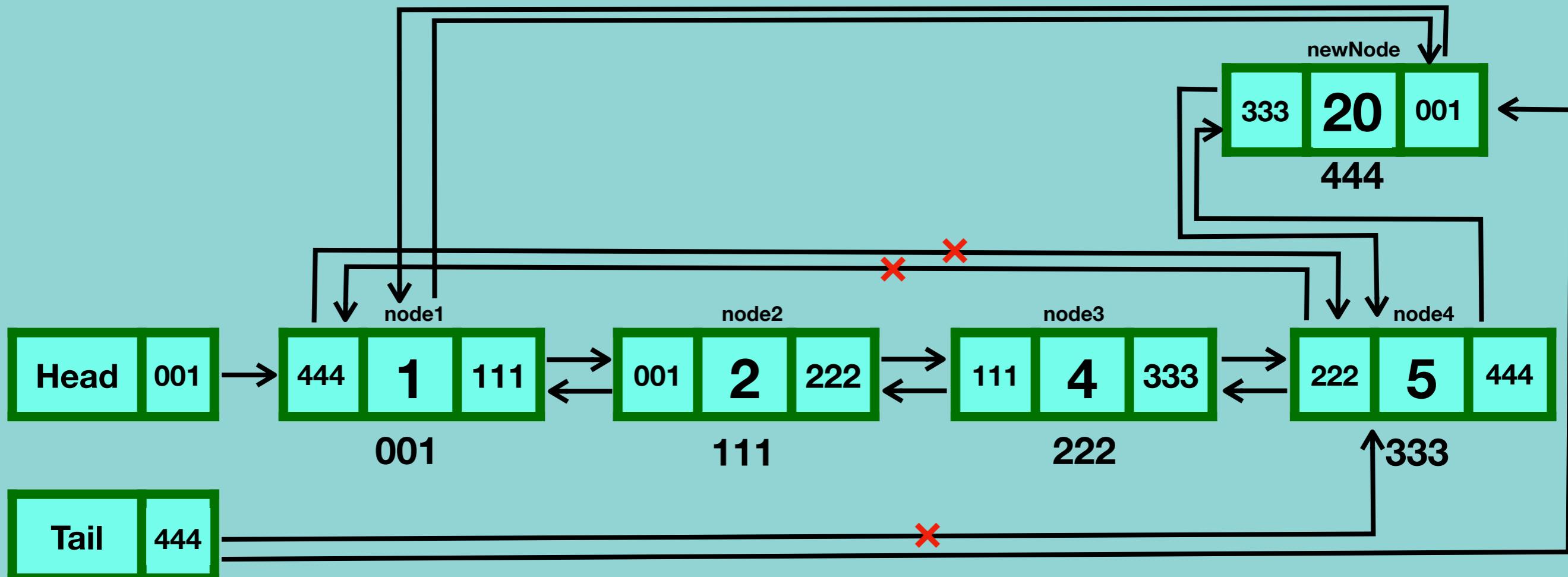
Circular Doubly Linked List - Insertion

- Insert at the specified location of linked list

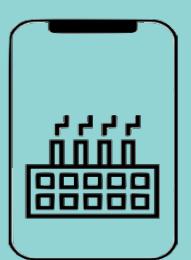
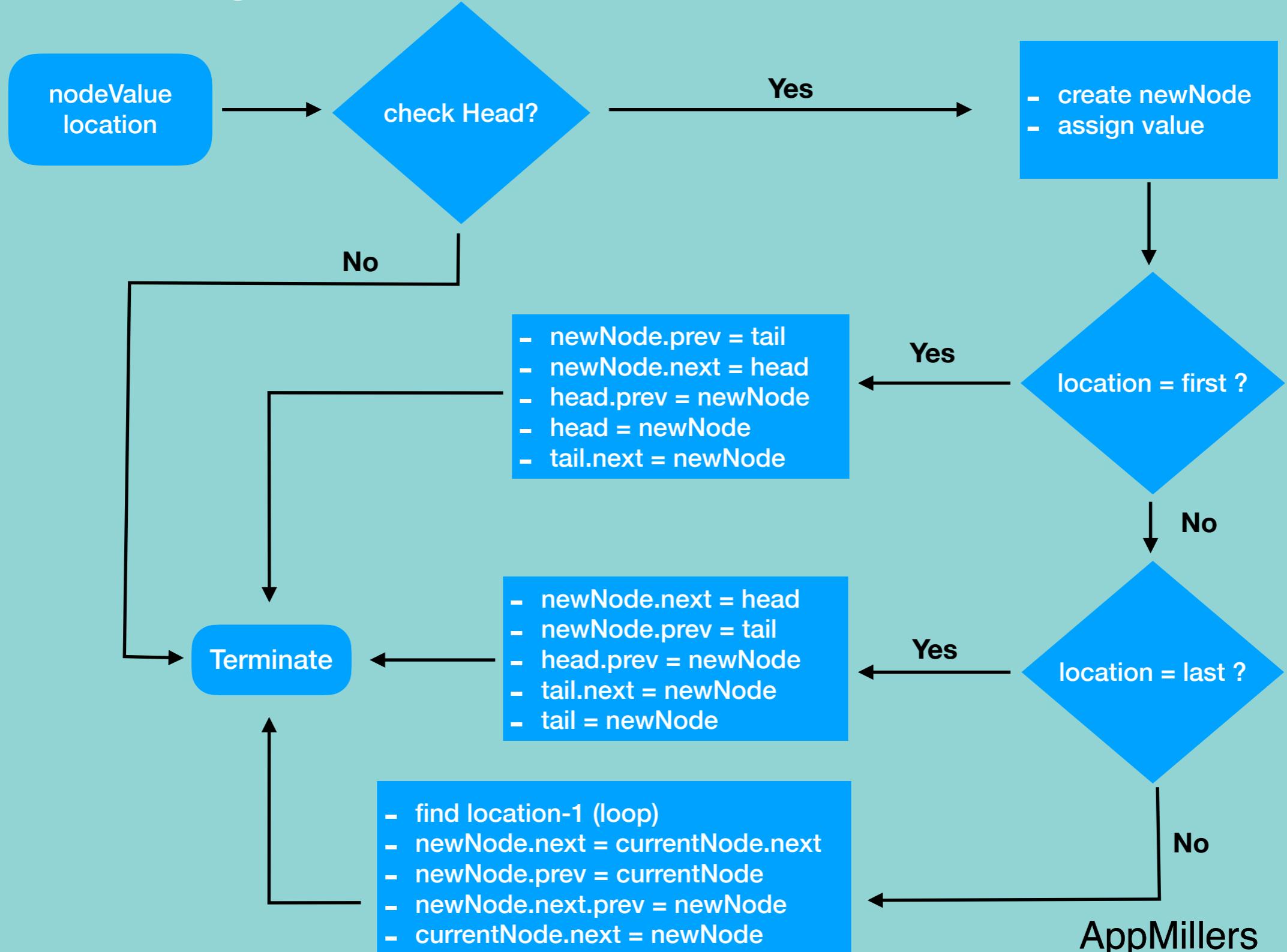


Circular Doubly Linked List - Insertion

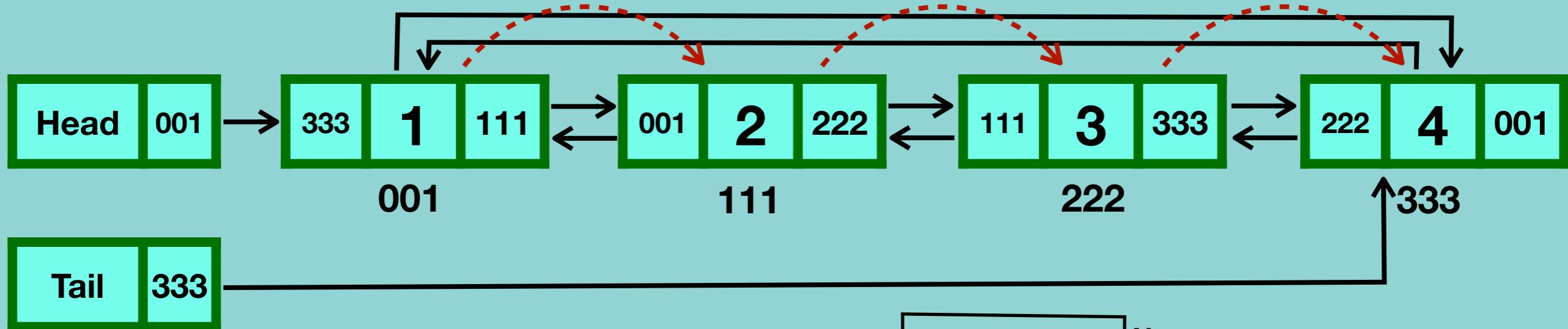
- Insert at the end of linked list



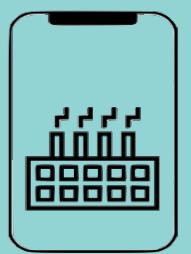
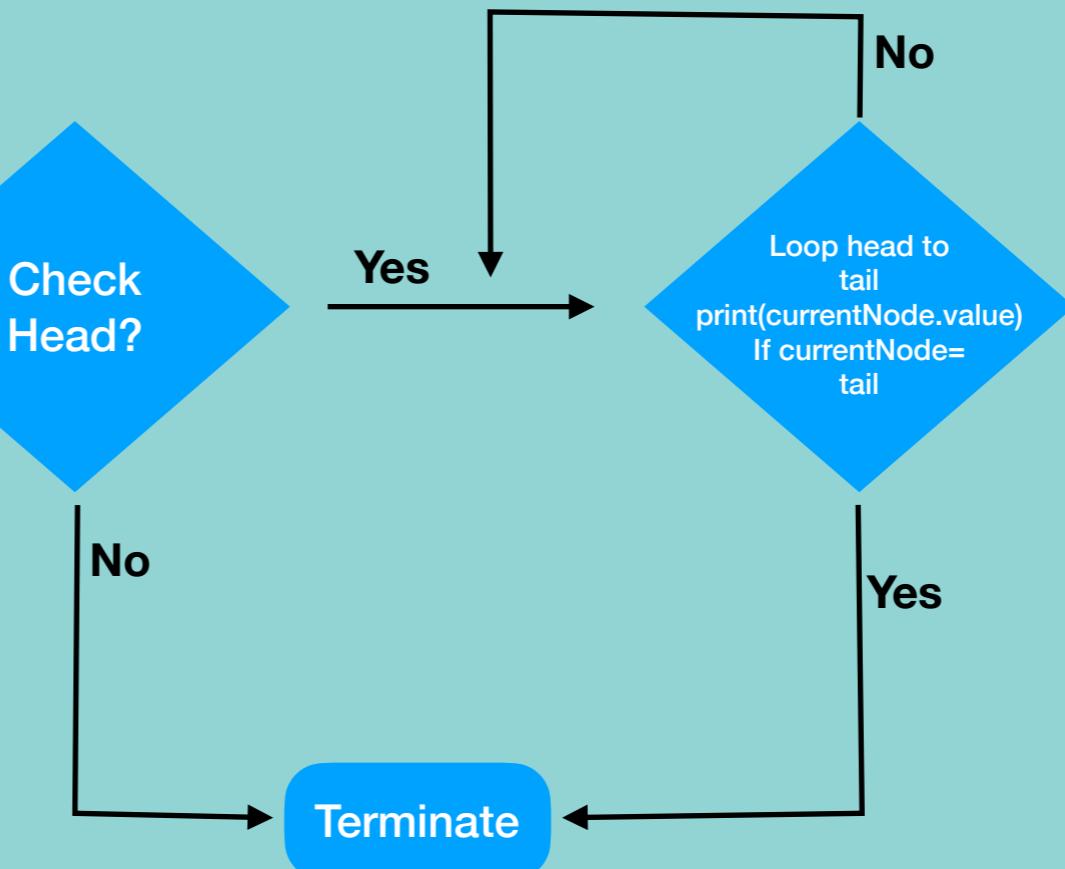
Insertion Algorithm - Circular Doubly linked list



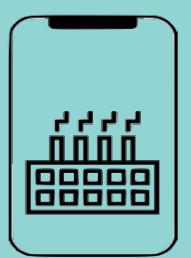
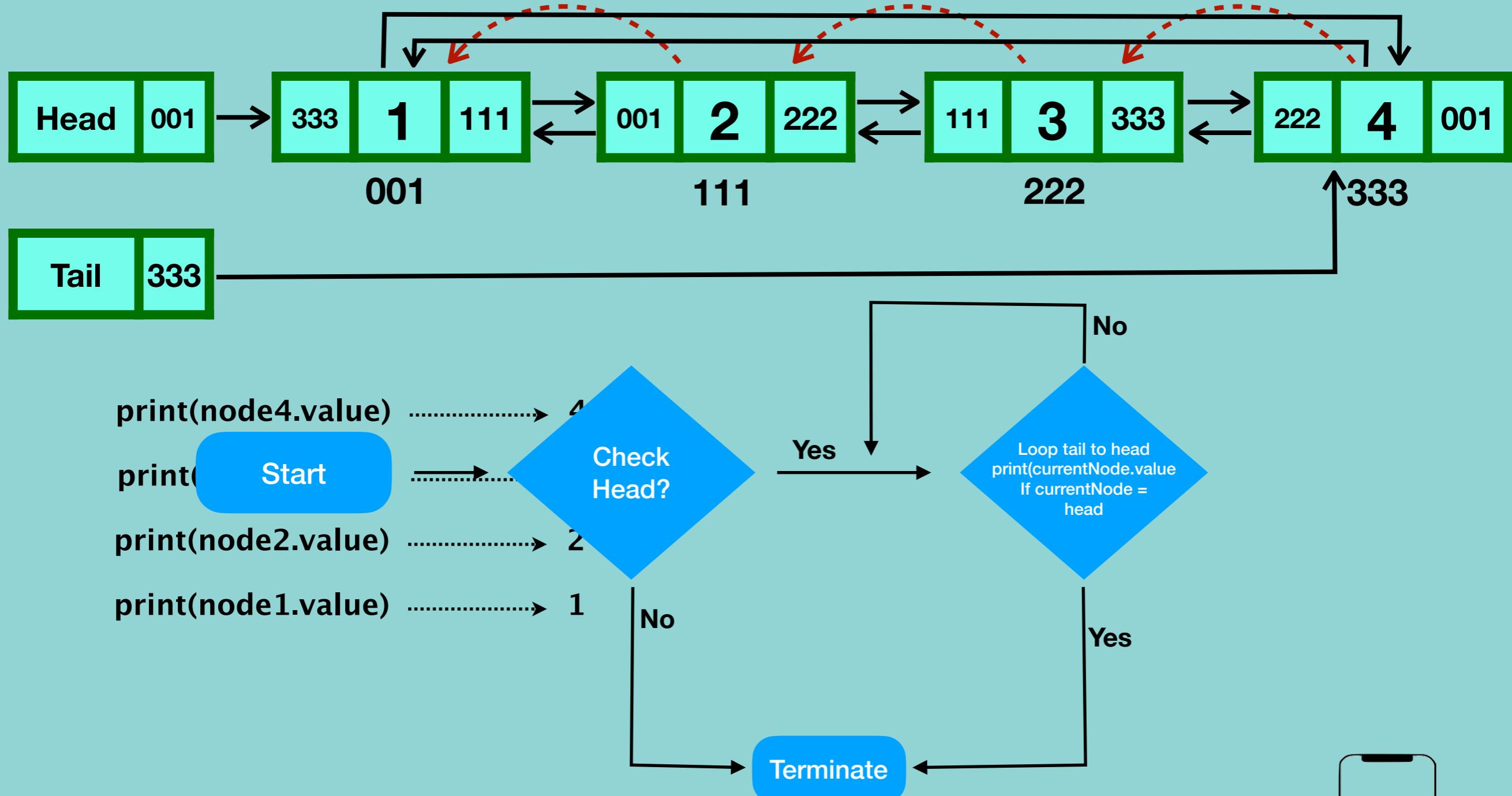
Circular Doubly Linked List - Traversal



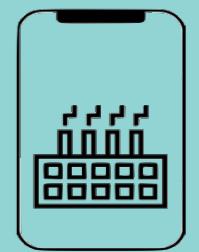
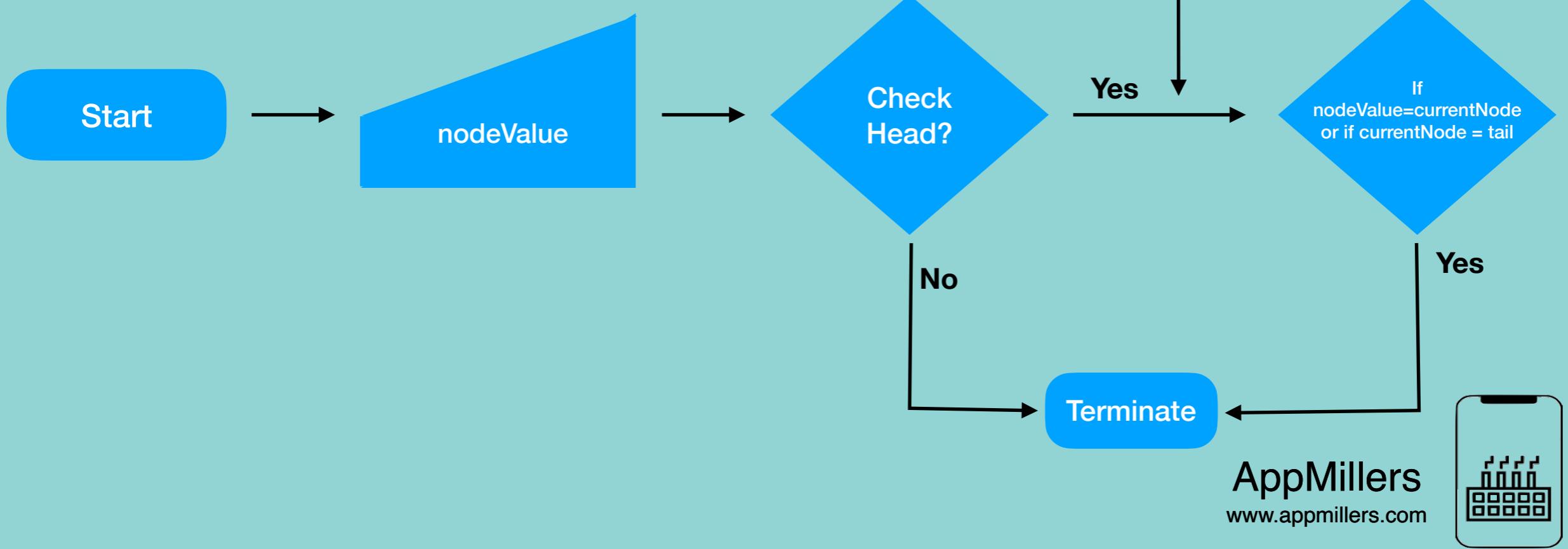
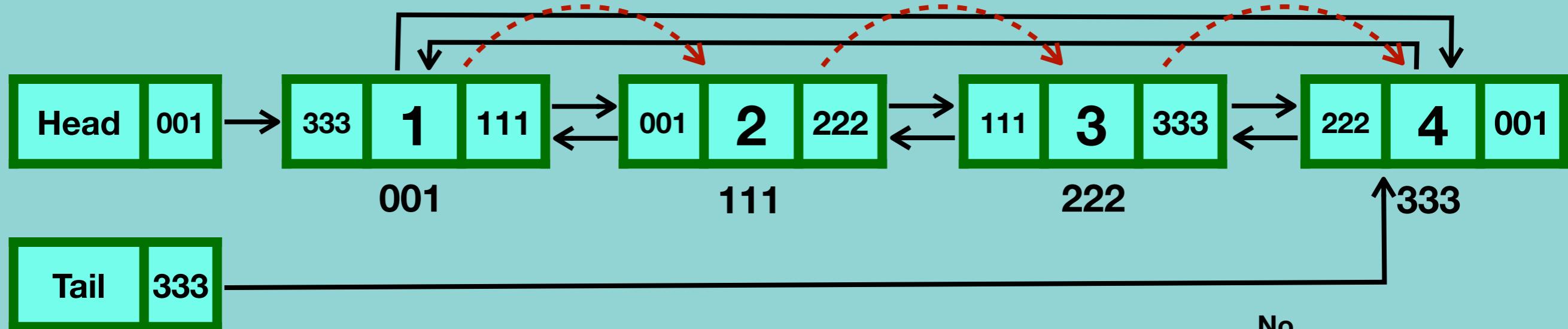
```
print(node1.value) .....> 1  
print() Start .....> 2  
print(node3.value) .....> 3  
print(node4.value) .....> 4
```



Circular Doubly Linked List - Reverse Traversal

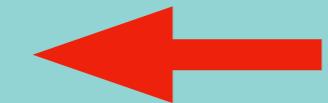


Circular Doubly Linked List - Search

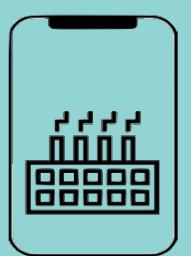
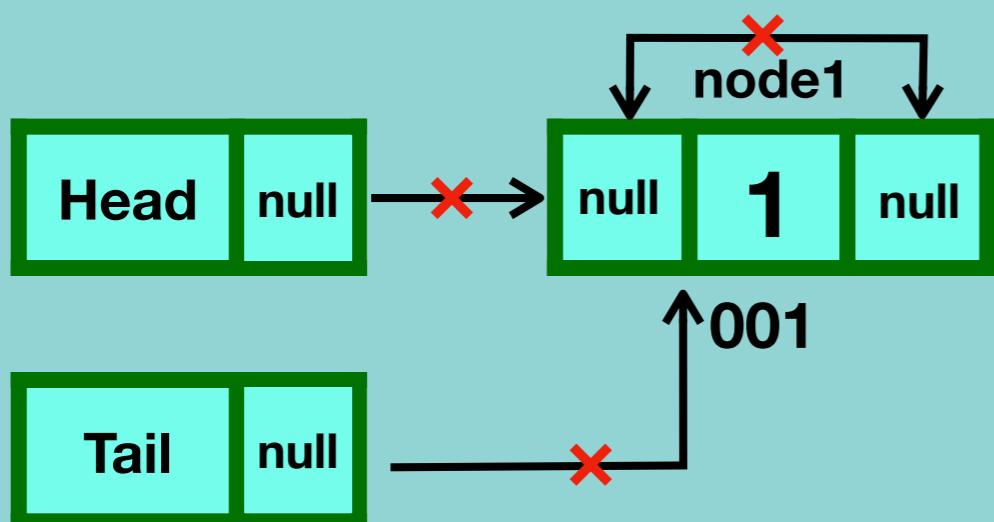


Circular Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node



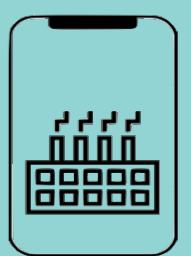
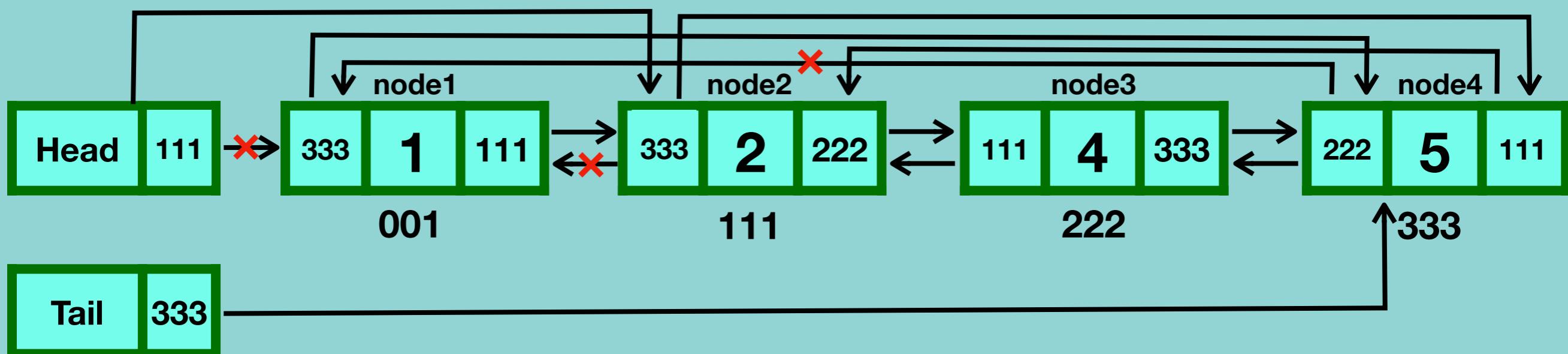
Case 1 - one node



Circular Doubly Linked list - Deletion

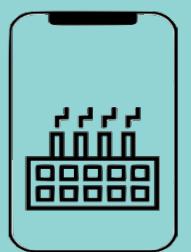
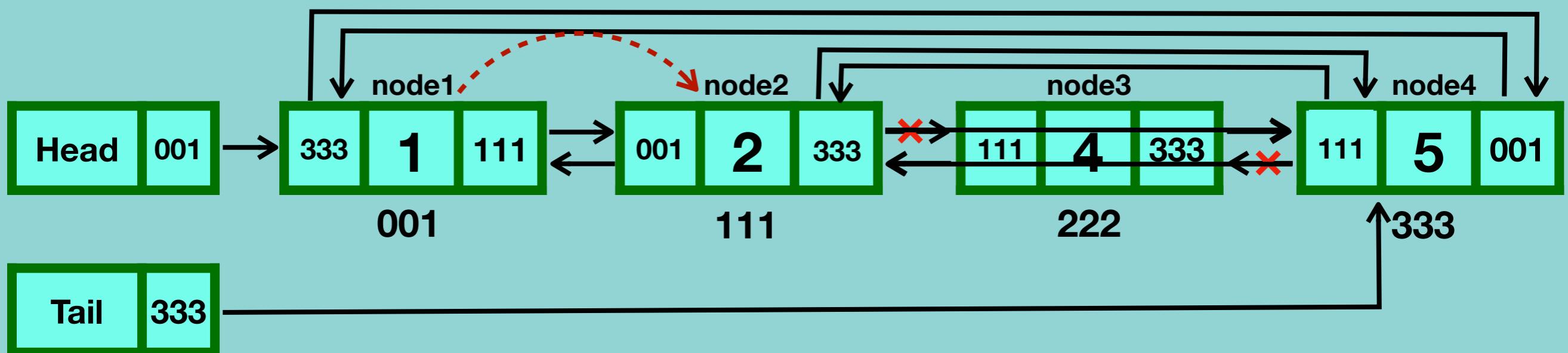
- Deleting the first node
- Deleting any given node
- Deleting the last node

Case 2 - more than one node



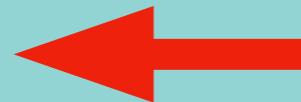
Circular Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

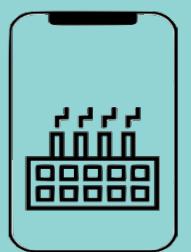
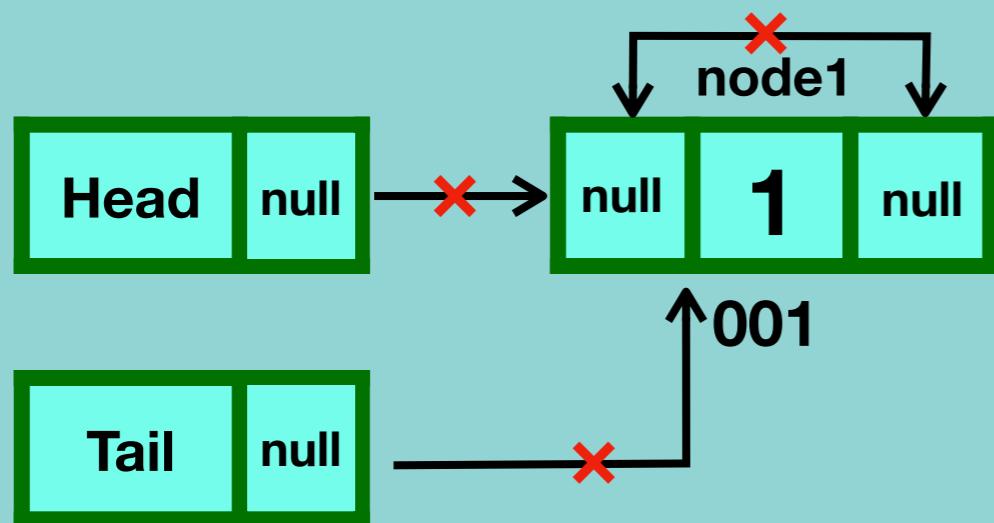


Circular Doubly Linked list - Deletion

- Deleting the first node
- Deleting any given node
- Deleting the last node

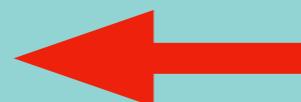


Case 1 - one node

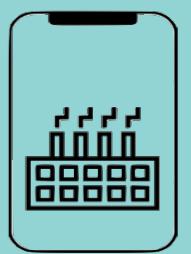
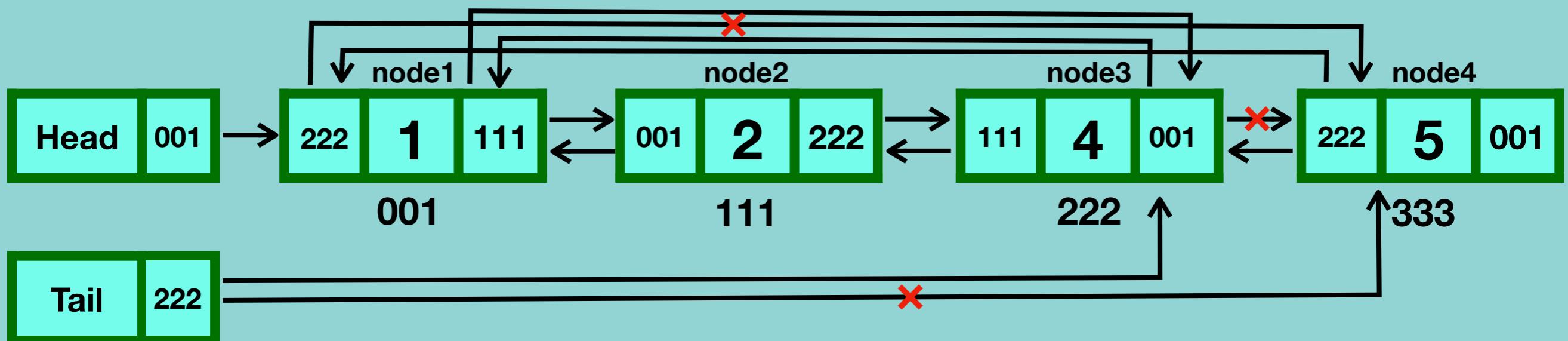


Circular Doubly Linked list - Deletion

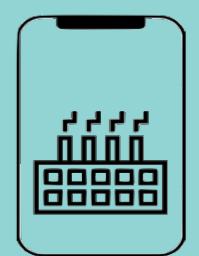
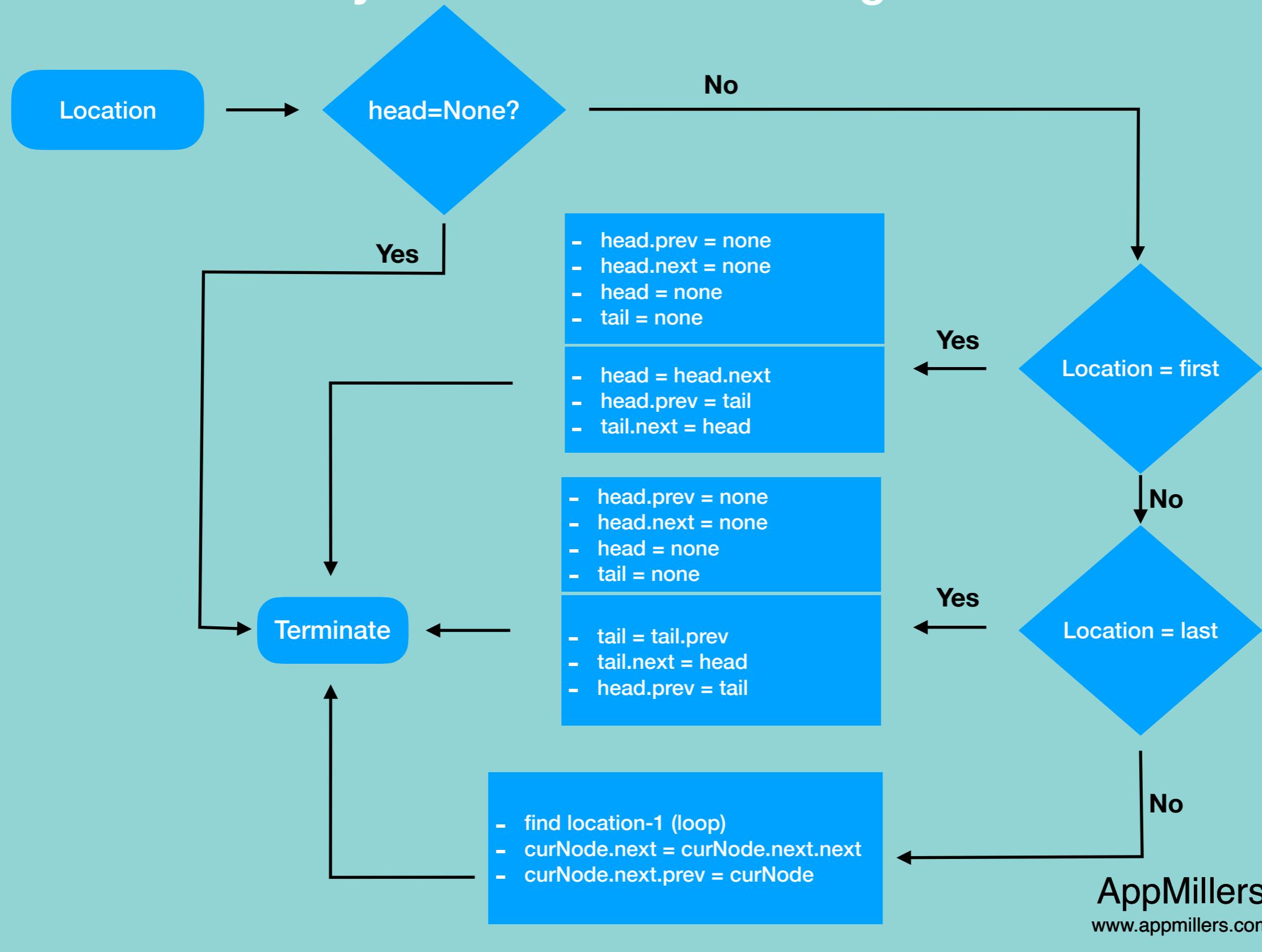
- Deleting the first node
- Deleting any given node
- Deleting the last node



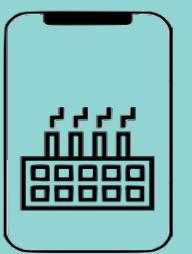
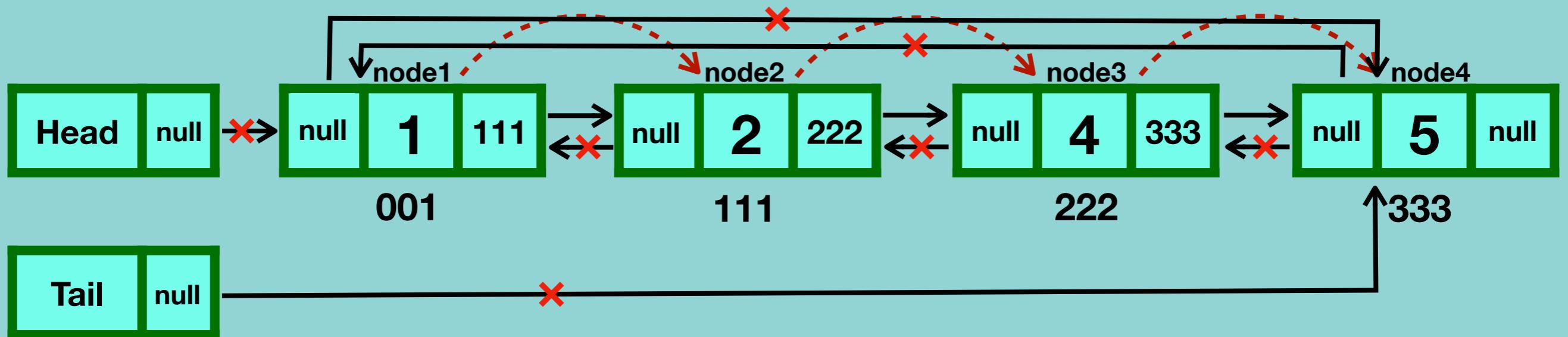
Case 2 - more than one node



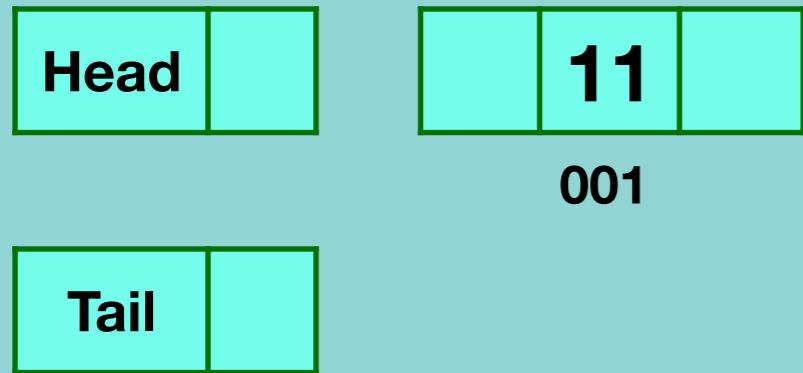
Circular Doubly Linked list Deletion Algorithm



Deleting entire circular doubly linked list

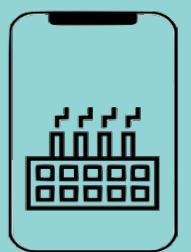


Circular doubly linked list - creation

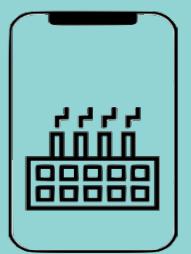
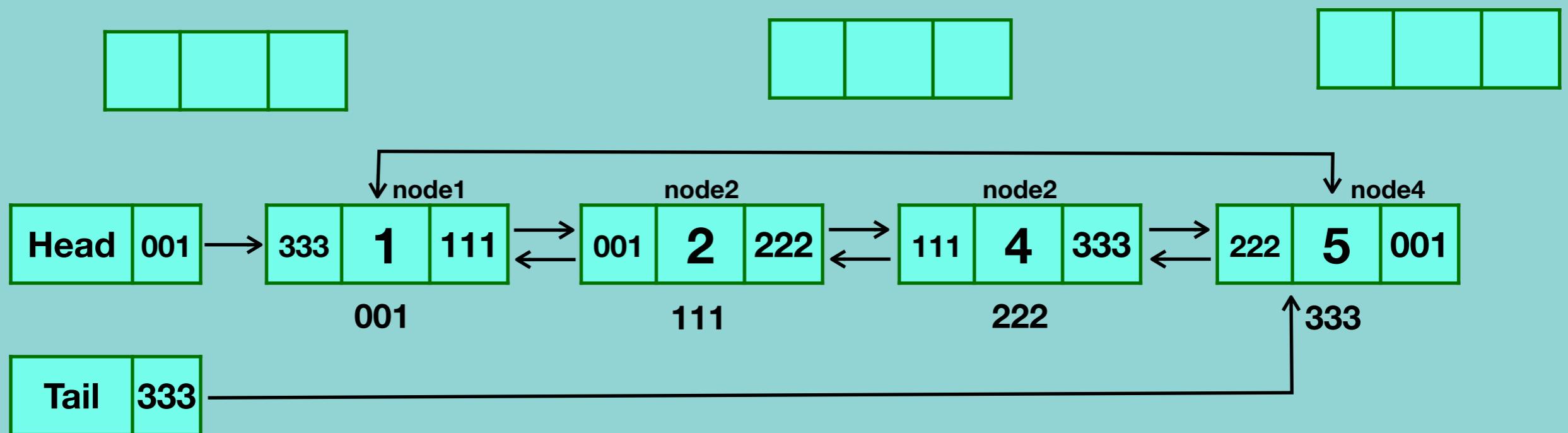


```
createCircularDoublyLinkedList(nodeValue):  
    create a blank newNode  
    newNode.value =nodeValue  
    head = newNode  
    tail = newNode  
    newNode.next = newNode.prev = newNode
```

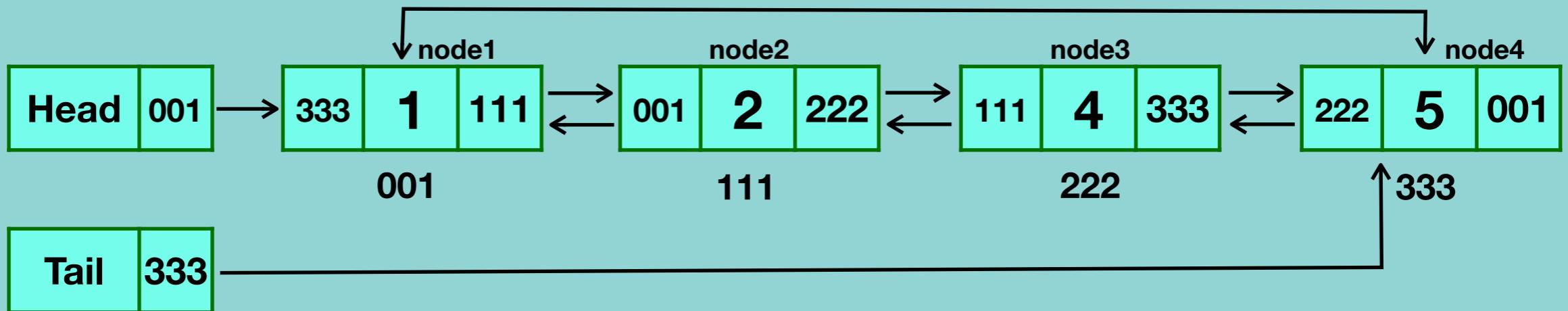
Time complexity : O(1)
Space complexity : O(1)



Circular doubly linked list - insertion



Circular doubly linked list - traversal

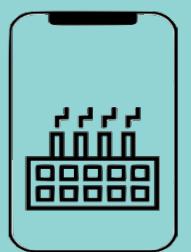


```
traversalCircularDoublyLinkedList():
```

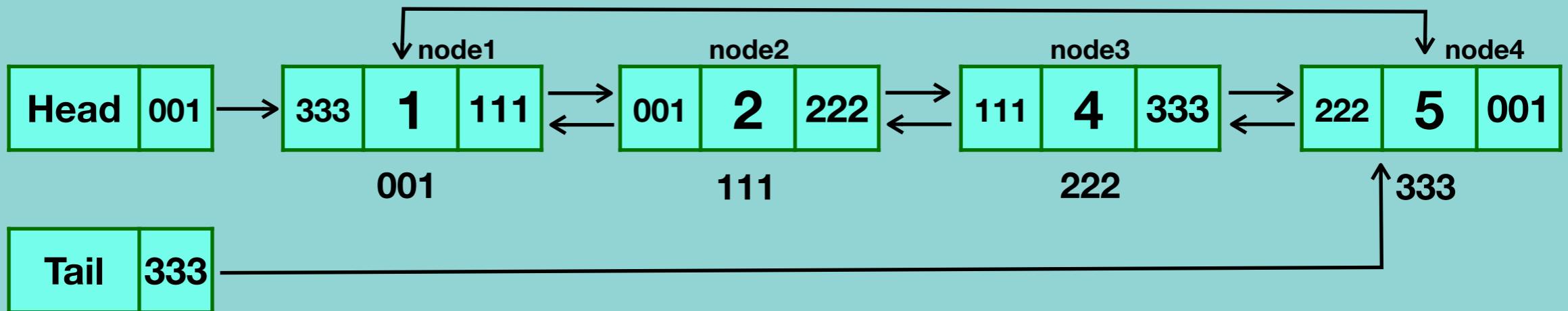
```
if head == null:  
    return //There is not any node in this list  
loop head to tail:  
    print(currentNode.value)  
    if currentNode.value = tail:  
        terminate
```

Time complexity : O(n)

Space complexity : O(1)



Circular doubly linked list - reverse traversal

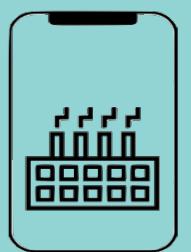


```
reverseTraversalCircularDoublyLinkedList():
```

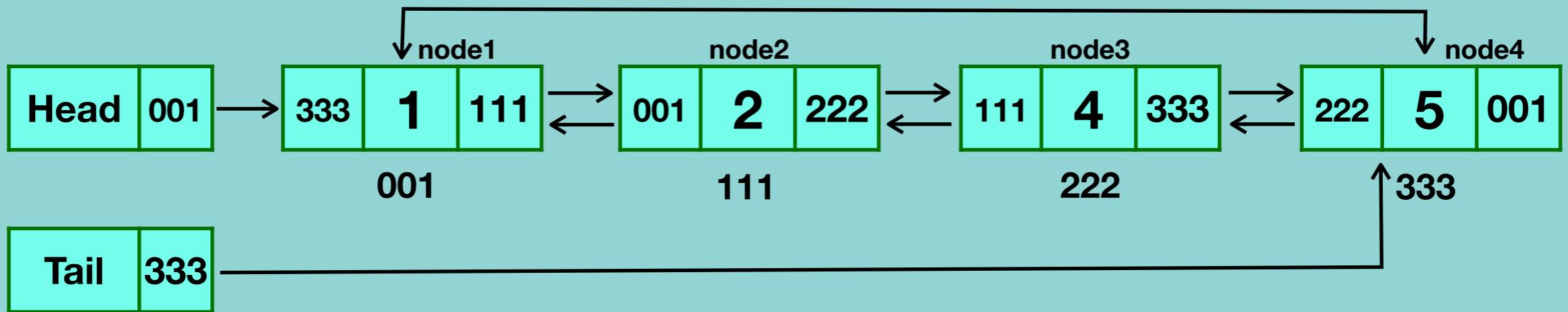
```
if head == null:  
    return //There is not any node in this list  
loop tail to head:  
    print(currentNode.value)  
    if currentNode.value = head:  
        terminate
```

Time complexity : O(n)

Space complexity : O(1)



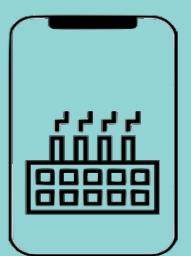
Circular doubly linked list - searching for a node



```
searchForNode(head,nodeValue):
    loop head to tail:
        if currentNode.value =nodeValue
            print(currentNode)
            return
        if currentNode.value =tail:
            terminate
    return //nodeValue not found
```

Time complexity : O(n)

Space complexity : O(1)

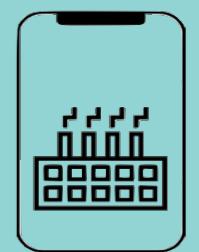
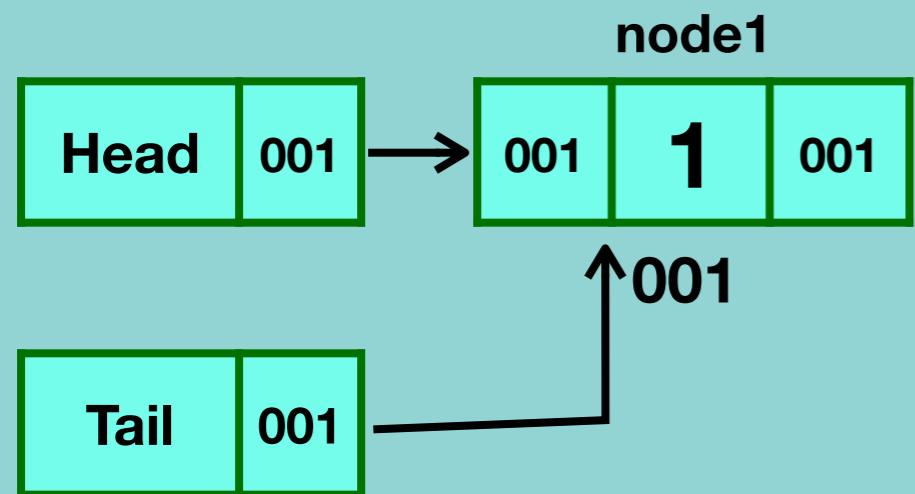


Deleting a node in doubly linked list

- Deleting the first node
- Deleting the last node
- Deleting any node apart from the first and the last

Deleting the first node

Case 1- there is only one node in the list

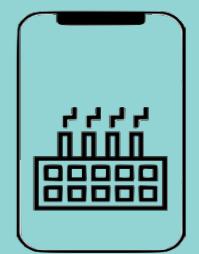
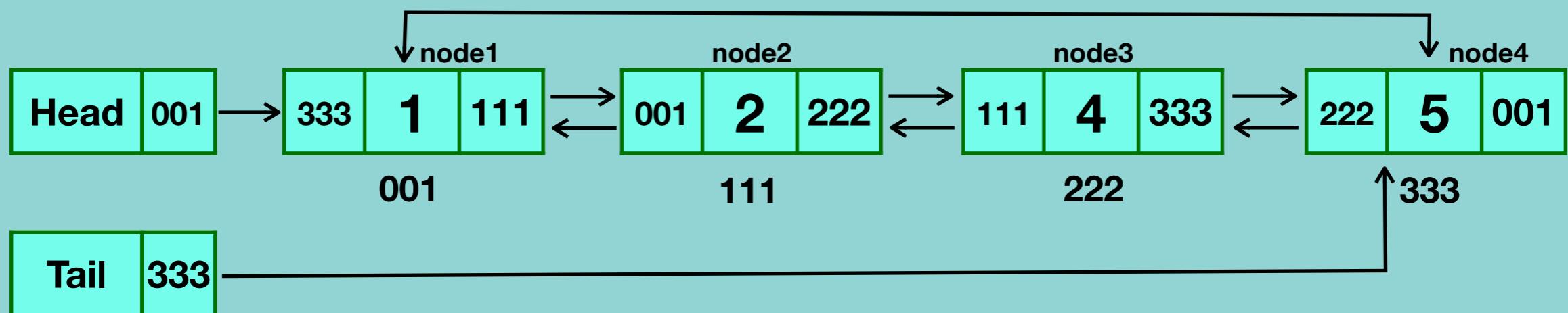


Deleting a node in circular doubly linked list

- Deleting the first node
- Deleting the last node
- Deleting any node apart from the first and the last

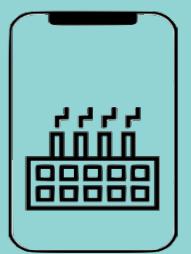
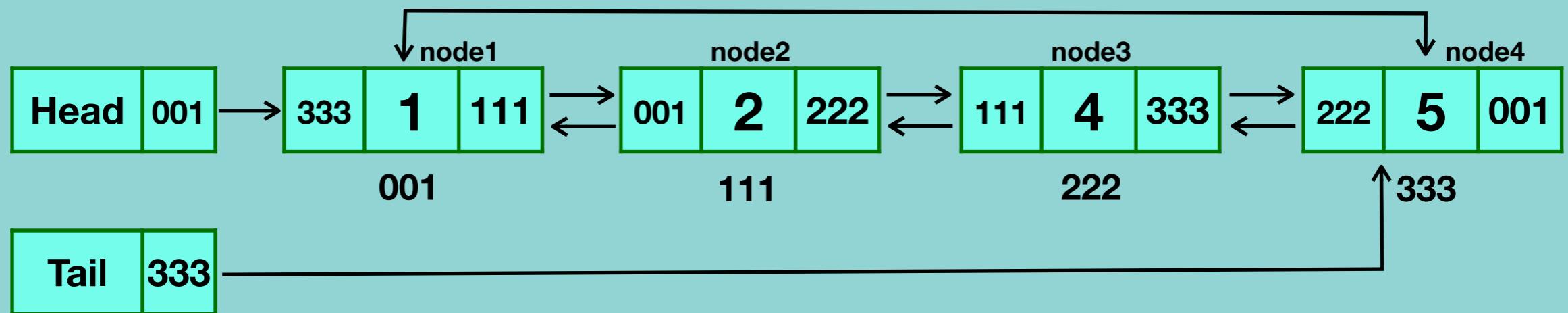
Deleting the first node

Case 1- More than one node in the list



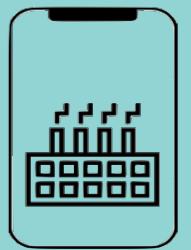
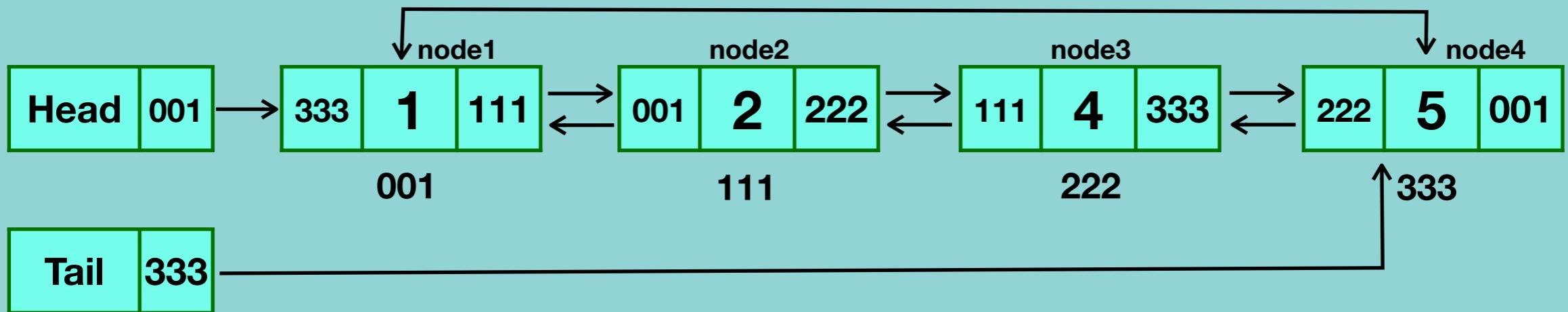
Deleting a node in circular doubly linked list

Deleting the last node node

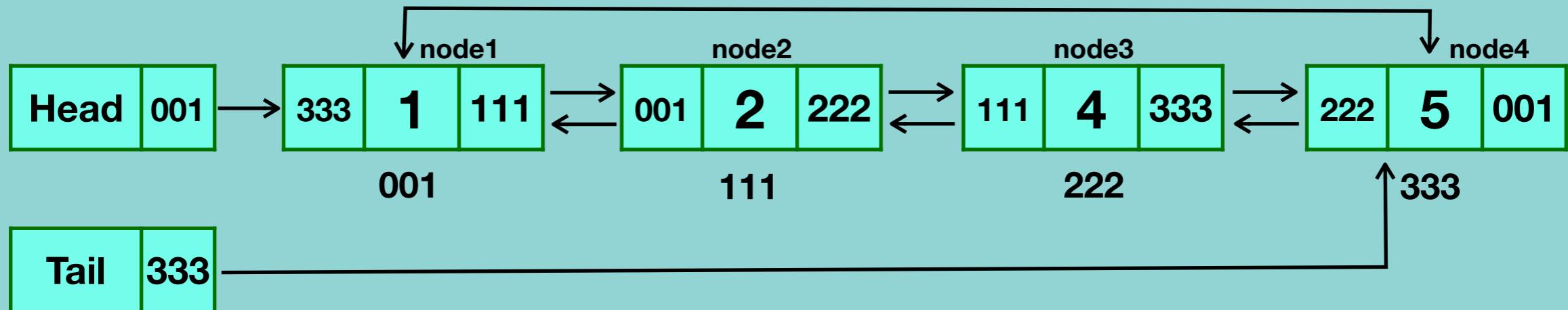


Deleting a node in circular doubly linked list

Deleting the node from the middle



Deleting a node in circular doubly linked list- Algorithm

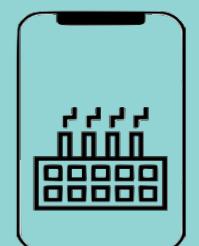


```
deleteNode(head, location):
```

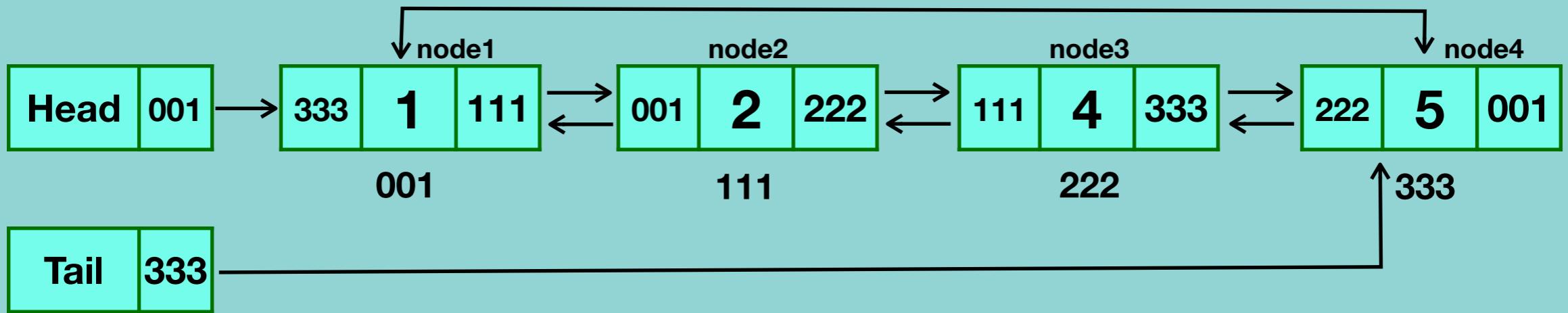
```
if head does not exist:  
    return error //Linked list does not exist  
if location = firstNode's location  
    if this is the only node in the list  
        head.prev=head.next=head=tail=null  
    else  
        head=head.next  
        head.prev = tail  
        tail.next = head  
else if location = lastNode's location  
    if this is the only node in the list  
        head.prev=head.next=head=tail=null  
    else  
        tail = tail.prev  
        tail.next = head  
        head.prev = tail  
else // delete middle node  
    loop until location-1 (curNode)  
    curNode.next = curNode.next.next  
    curNode.next.prev = curNode
```

Time complexity : O(n)

Space complexity : O(1)

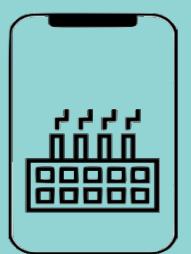


Deleting entire circular doubly linked list- Algorithm



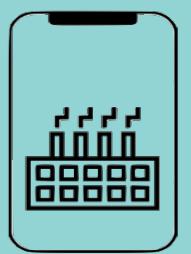
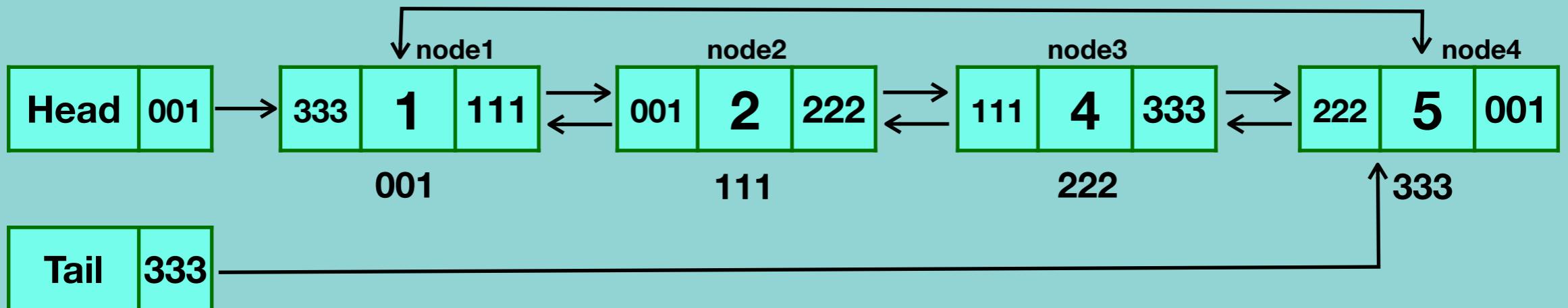
```
deleteLinkedList(head, tail):  
    tail.next = null  
    loop head to tail:  
        curNode.prev=null  
    head=tail=null
```

Time complexity : O(n)
Space complexity : O(1)



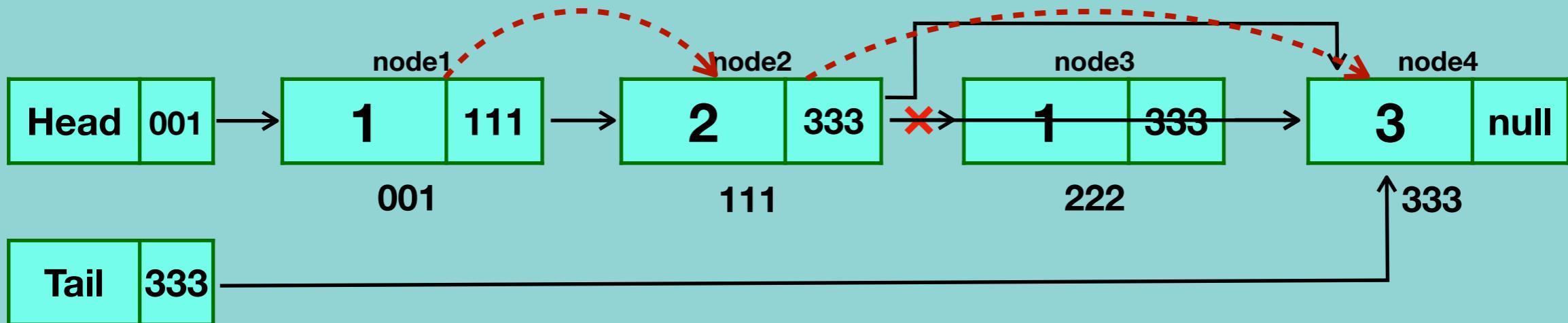
Time and Space complexity of circular doubly linked list

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing (forward, backward)	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(n)	O(1)



Interview Questions - 1 : Remove Duplicates

Write code to remove duplicates from an unsorted linked list



currentNode = node1

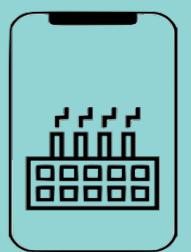
tempSet = {1} → {1,2} → {1,2,3}

while currentNode.next is not None:

If next node's value is in tempSet

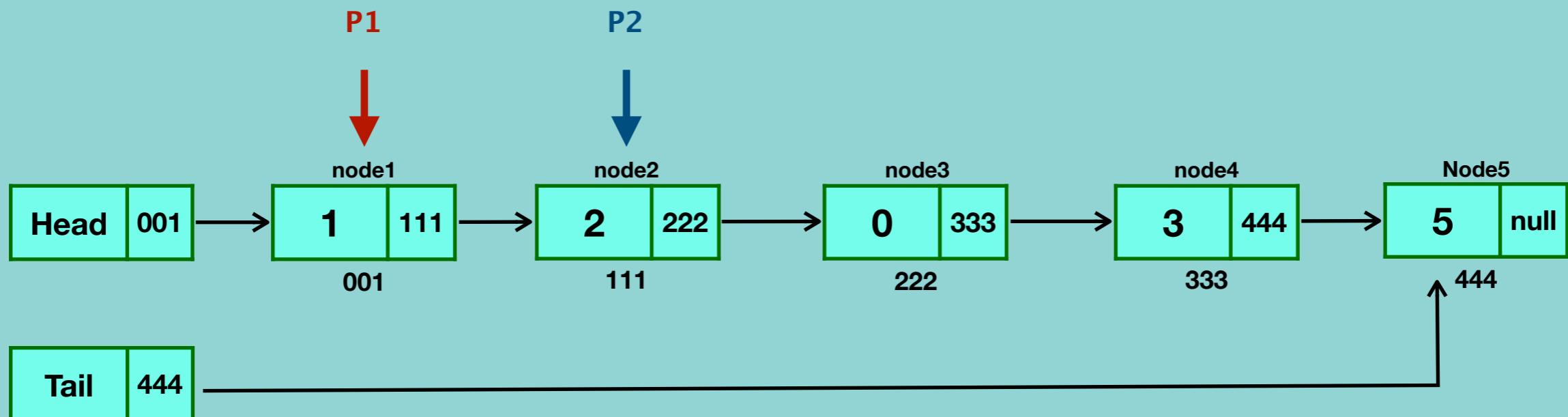
Delete next node

Otherwise add it to tempSet



Interview Questions - 2 : Return Nth to Last

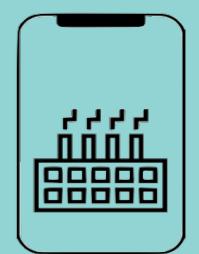
Implement and algorithm to find the nth to last element of a singly linked list.



$N = 2 \longrightarrow \text{Node4}$

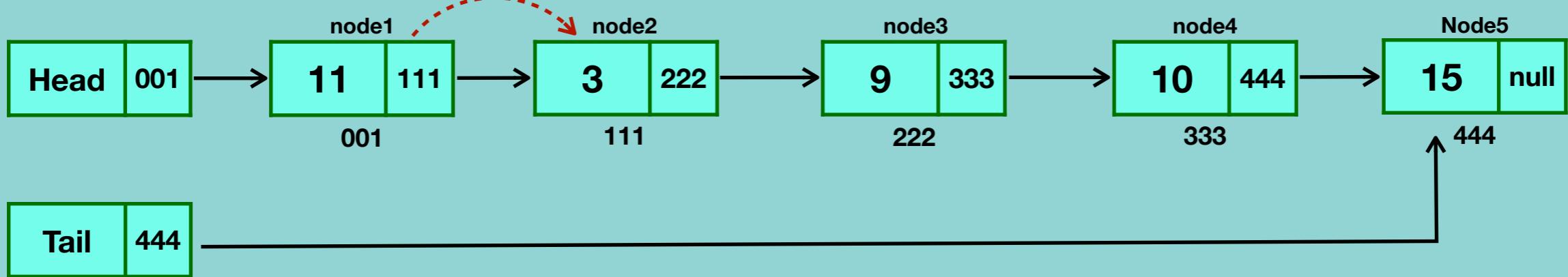
pointer1 = node1
= node2
= node3
= node4

pointer2 = node2
= node3
= node4
= node5



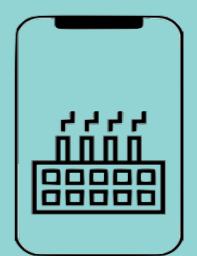
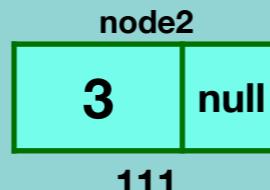
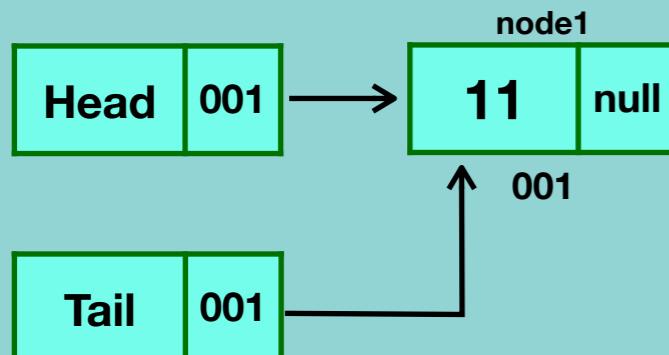
Interview Questions - 3 : Partition

Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.



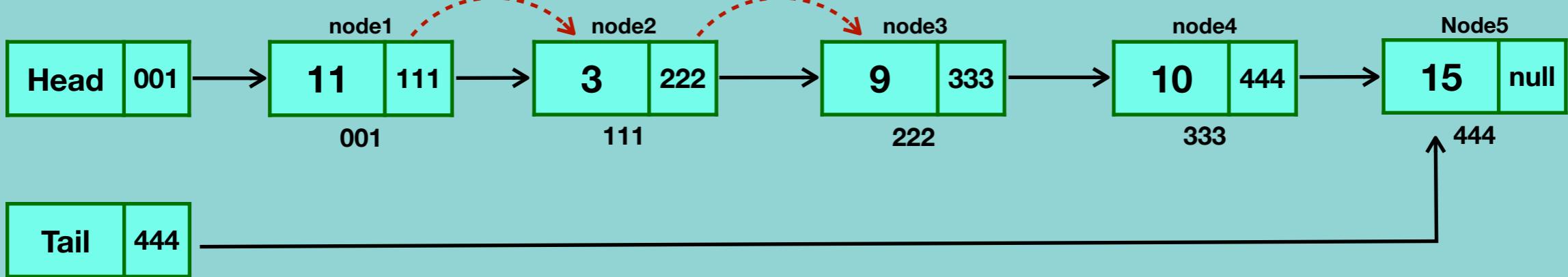
x = 10

```
currentNode = node1  
Tail = node1  
currentNode.next = null
```



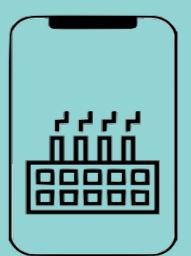
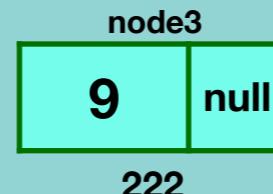
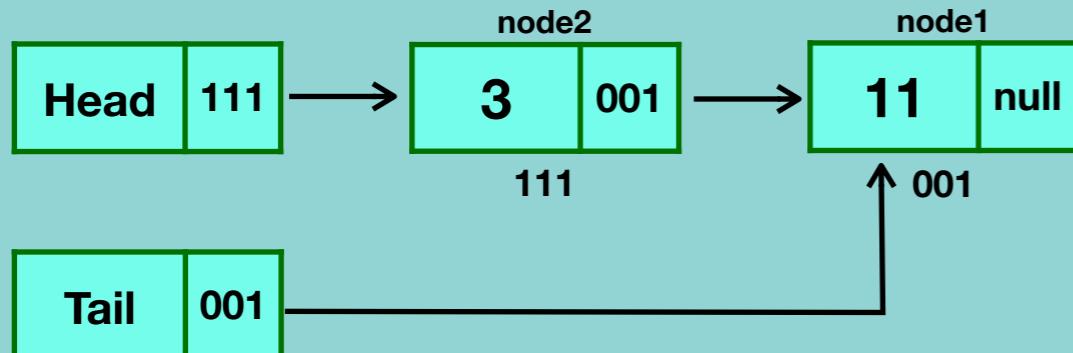
Interview Questions - 3 : Partition

Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.



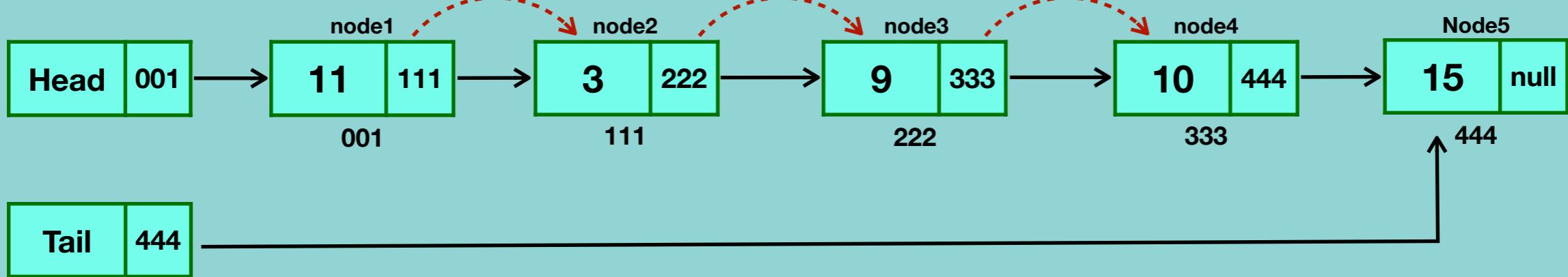
$x = 10$

```
currentNode = node1  
Tail = node1  
currentNode.next = null
```



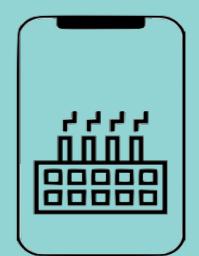
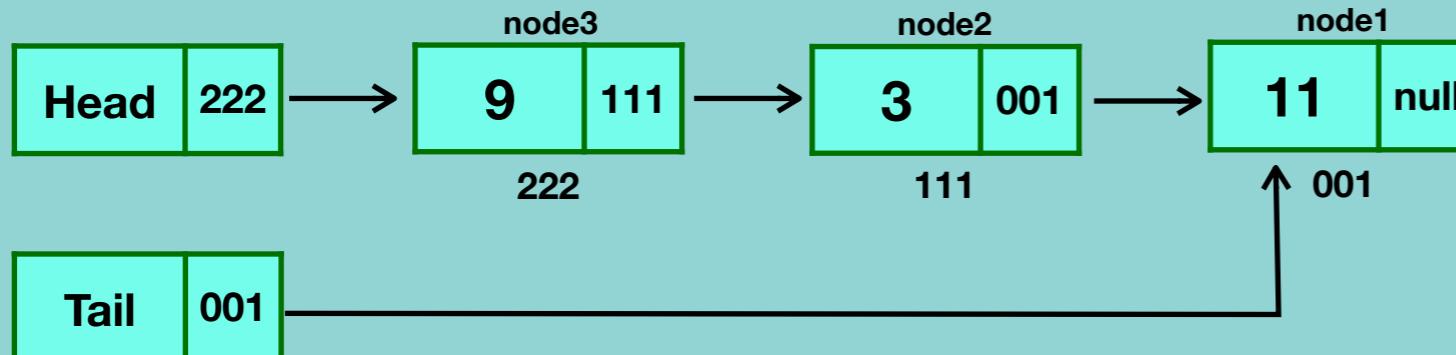
Interview Questions - 3 : Partition

Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.



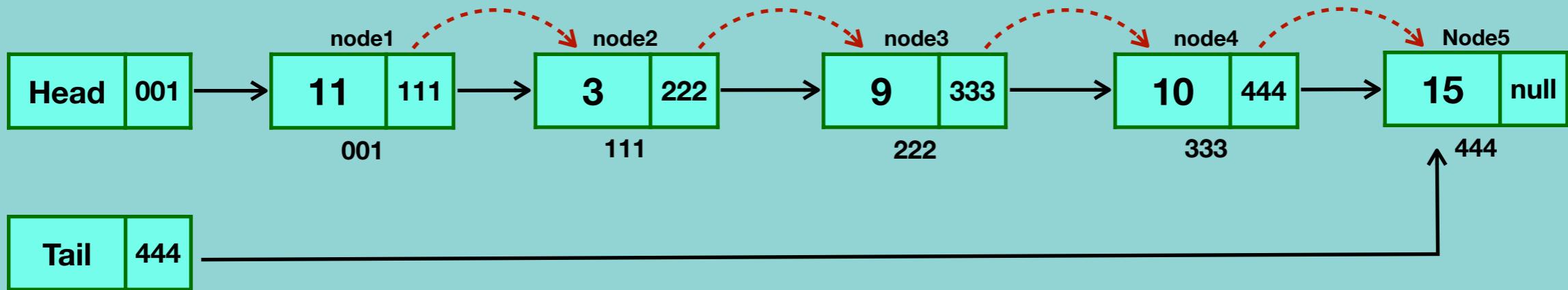
$x = 10$

```
currentNode = node1  
Tail = node1  
currentNode.next = null
```



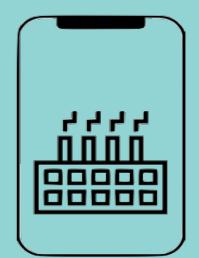
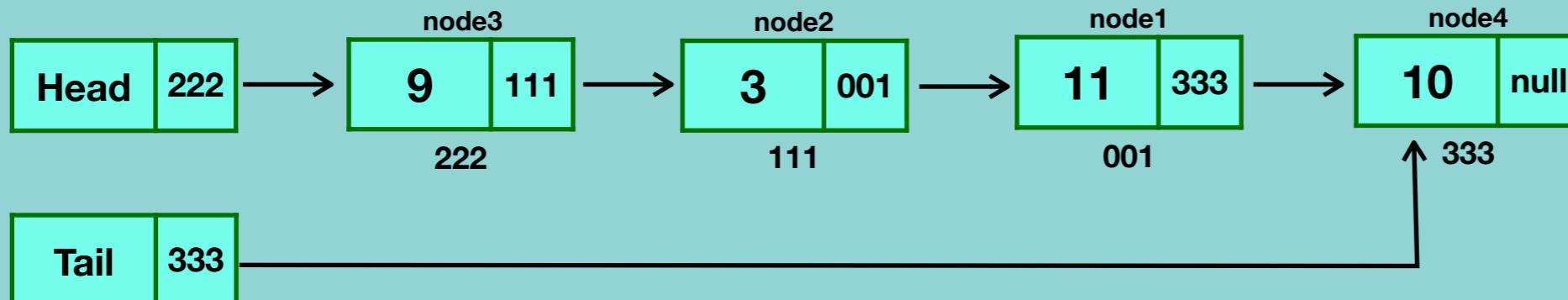
Interview Questions - 3 : Partition

Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.



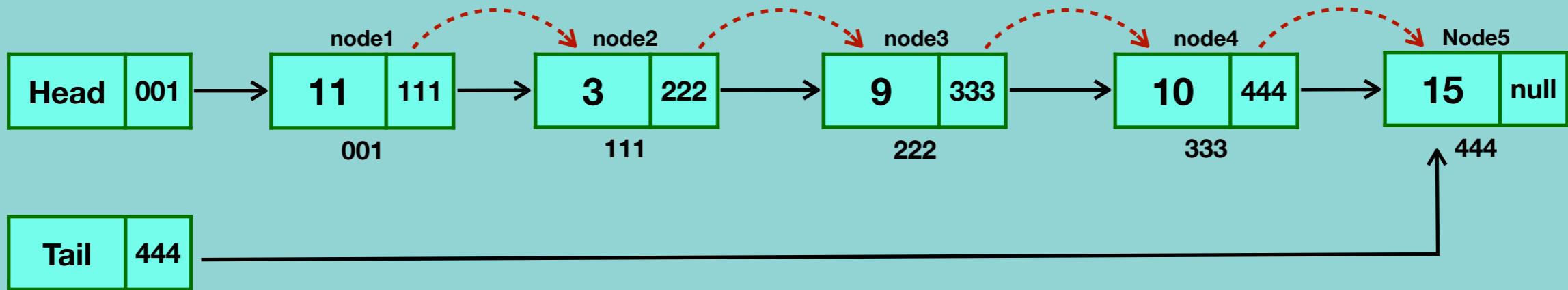
$x = 10$

```
currentNode = node1  
Tail = node1  
currentNode.next = null
```



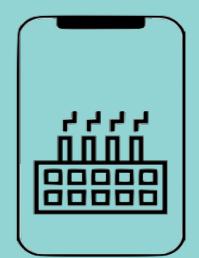
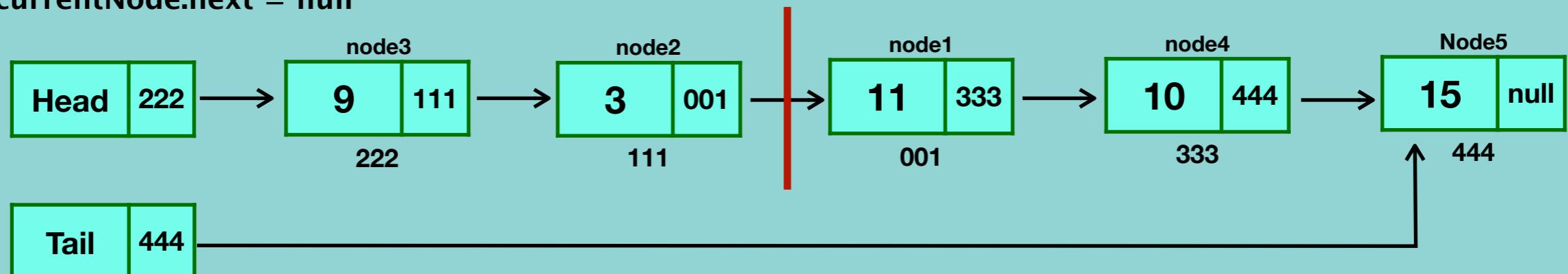
Interview Questions - 3 : Partition

Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.



$x = 10$

```
currentNode = node1  
Tail = node1  
currentNode.next = null
```



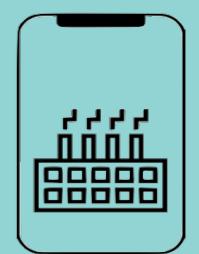
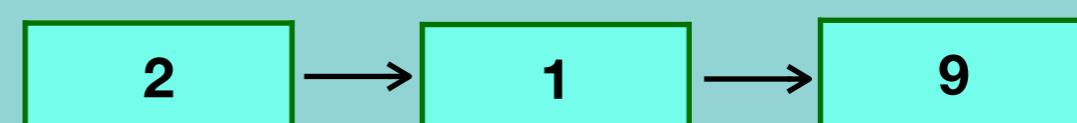
Interview Questions - 4 : Sum Lists

You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

list1 = 7 -> 1 -> 6 → 617
list2 = 5 -> 9 -> 2 → 295 → 617 + 295 = 912 → sumList = 2 -> 1 -> 9

$$\begin{array}{r} 617 \\ + 295 \\ \hline 912 \end{array}$$

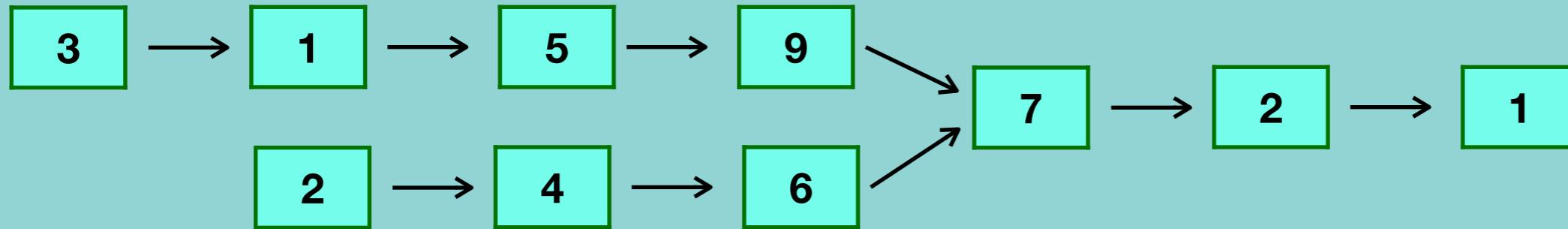
7 + 5 = 12
1+9+1 = 11
6+2+1 = 9



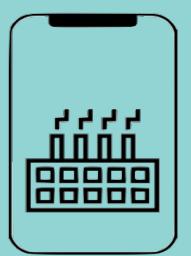
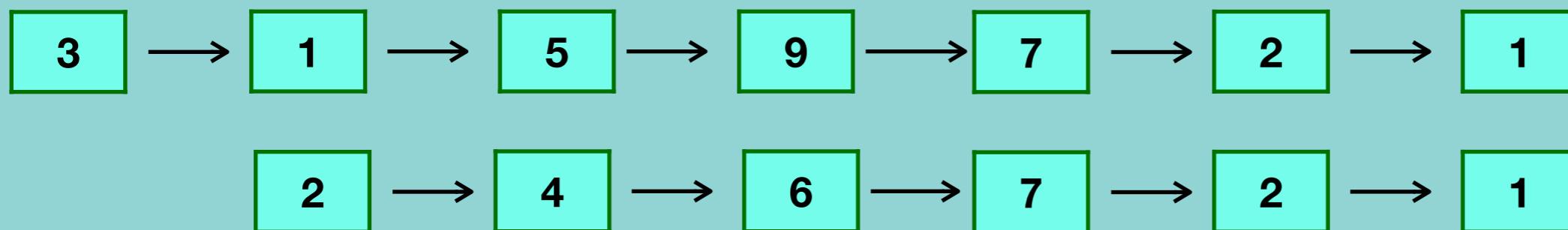
Interview Questions - 5 : Intersection

Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the jth node of the second linked list, then they are intersecting.

Intersecting linked lists



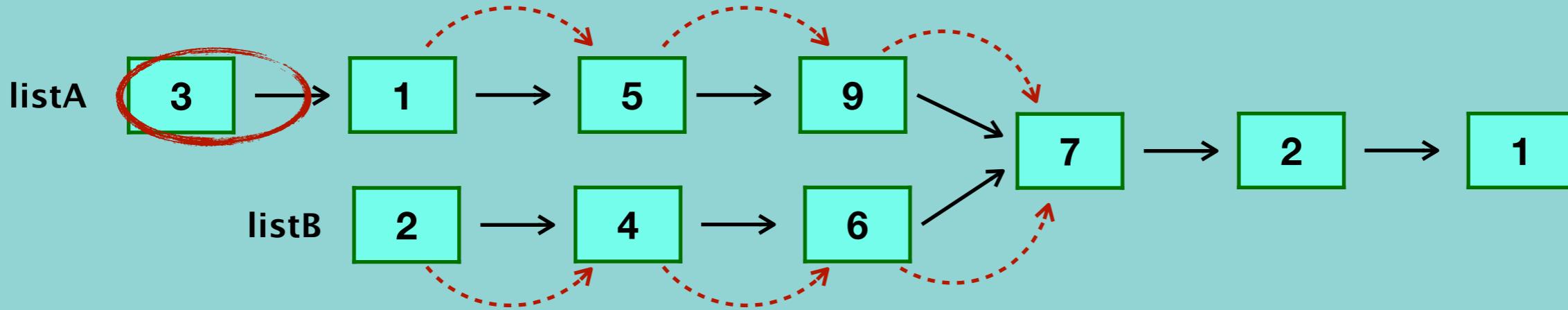
Non – intersecting linked lists



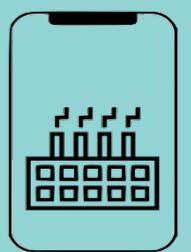
Interview Questions - 5 : Intersection

Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the jth node of the second linked list, then they are intersecting.

Intersecting linked lists

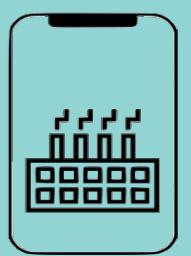
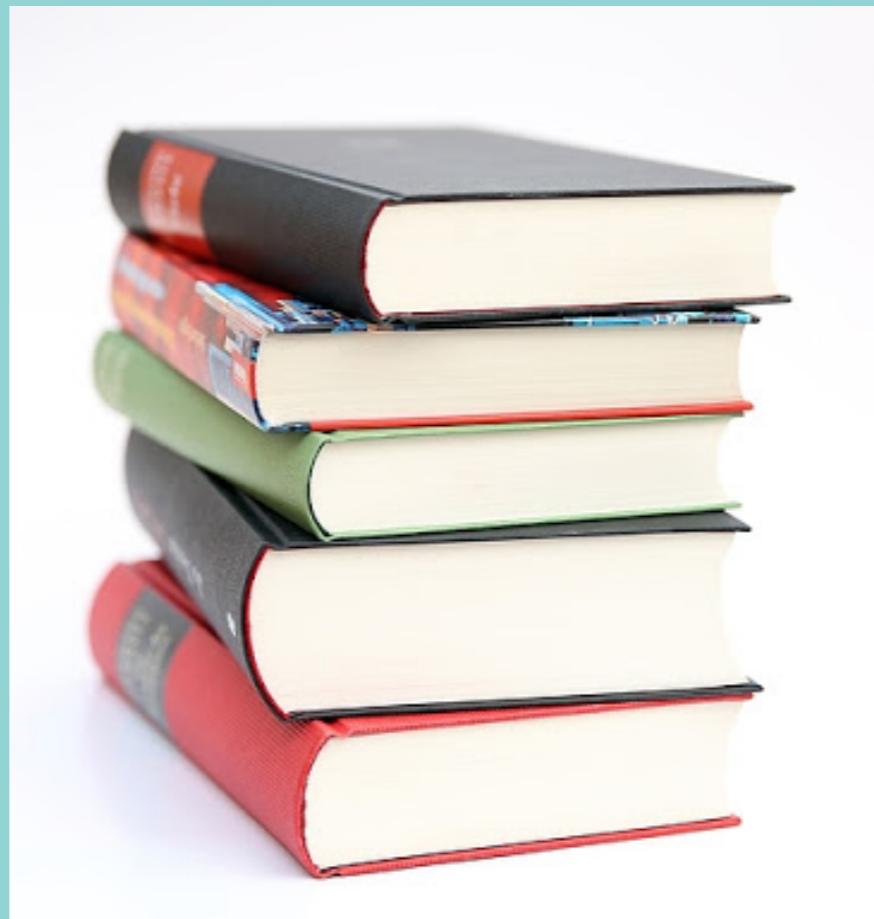


$$\begin{aligned} \text{len(listA)} &= 7 \\ \text{len(listB)} &= 6 \end{aligned} \longrightarrow 7 - 6 = 1$$



What is a Stack?

Stack is a data structure that stores items in a Last-In/First-Out manner.

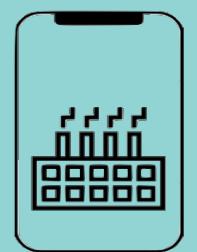


What is a Stack?

Stack is a data structure that stores items in a Last-In/First-Out manner.

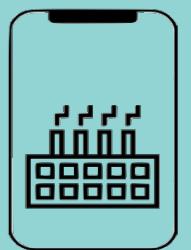
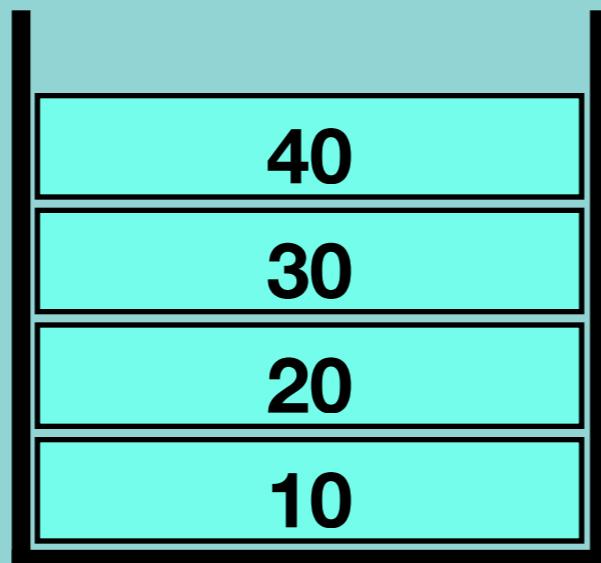


LIFO method

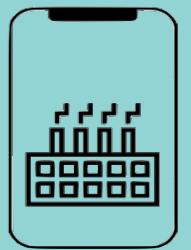
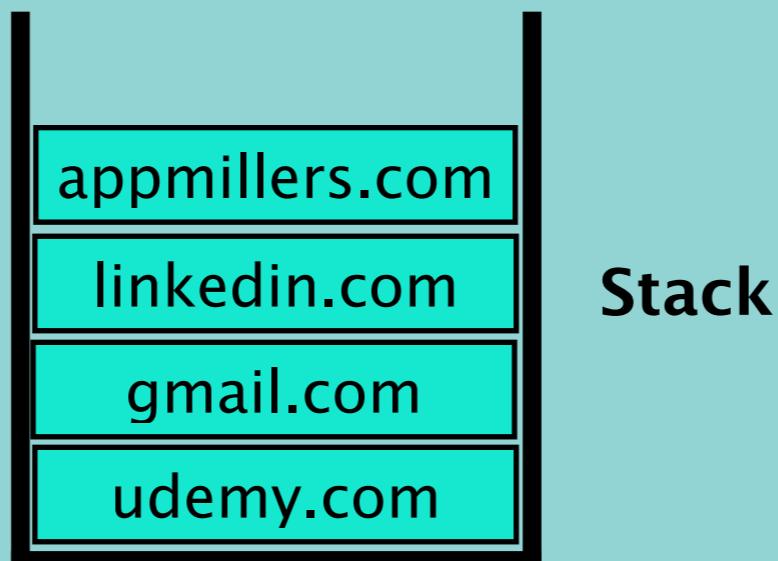
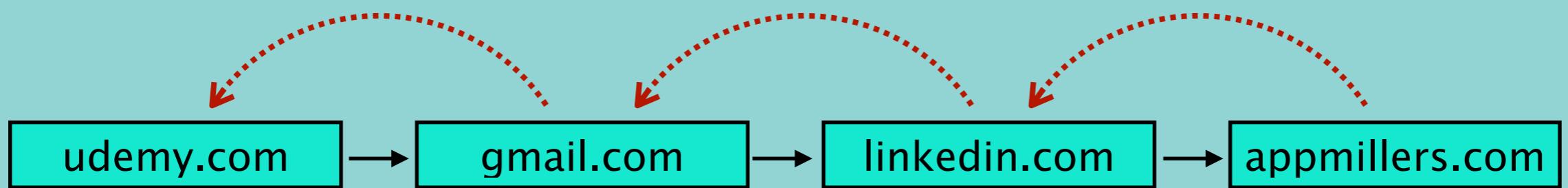
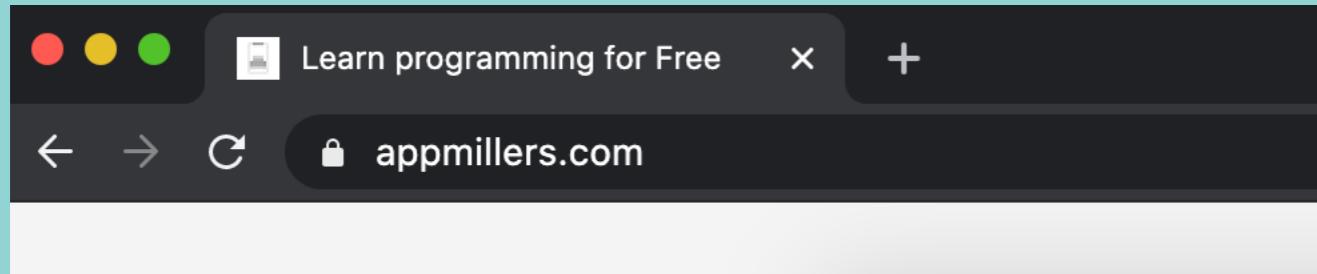


What is a Stack?

Stack is a data structure that stores items in a Last-In/First-Out manner.



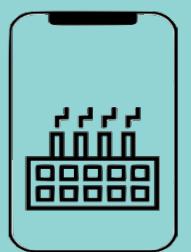
Why do we need a Stack?



Stack operations

- Create Stack
- Push
- Pop
- Peek
- isEmpty
- isFull
- deleteStack

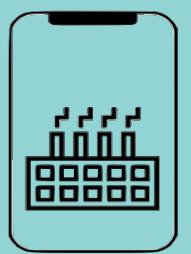
customStack()



Push() method

```
customStack = [ ]
```

```
customStack.push(1)
```

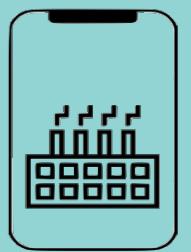



Push() method

customStack = [1]

customStack.push(2)

				1					

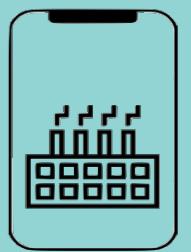


Push() method

customStack = [1,2]

customStack.push(3)

				2						
				1						

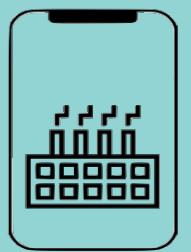


Push() method

customStack = [1,2,3]

customStack.push(3)

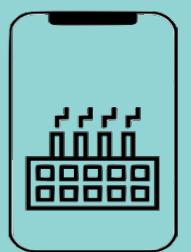
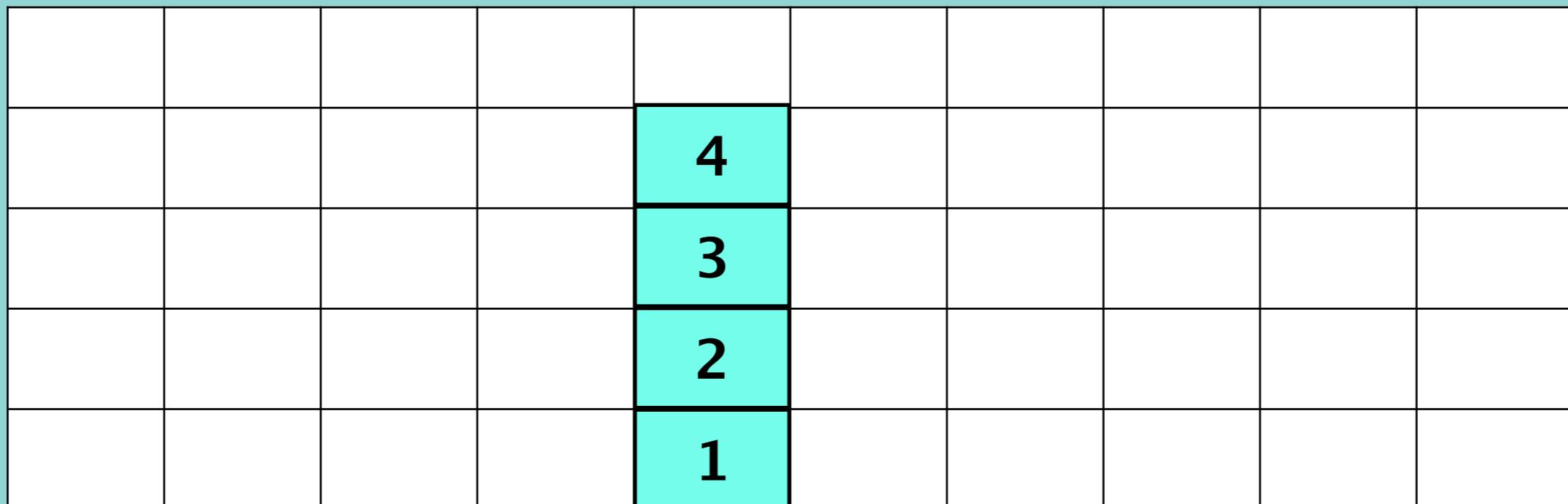
					3					
					2					
					1					



Push() method

customStack = [1,2,3,4]

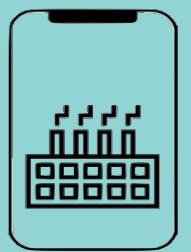
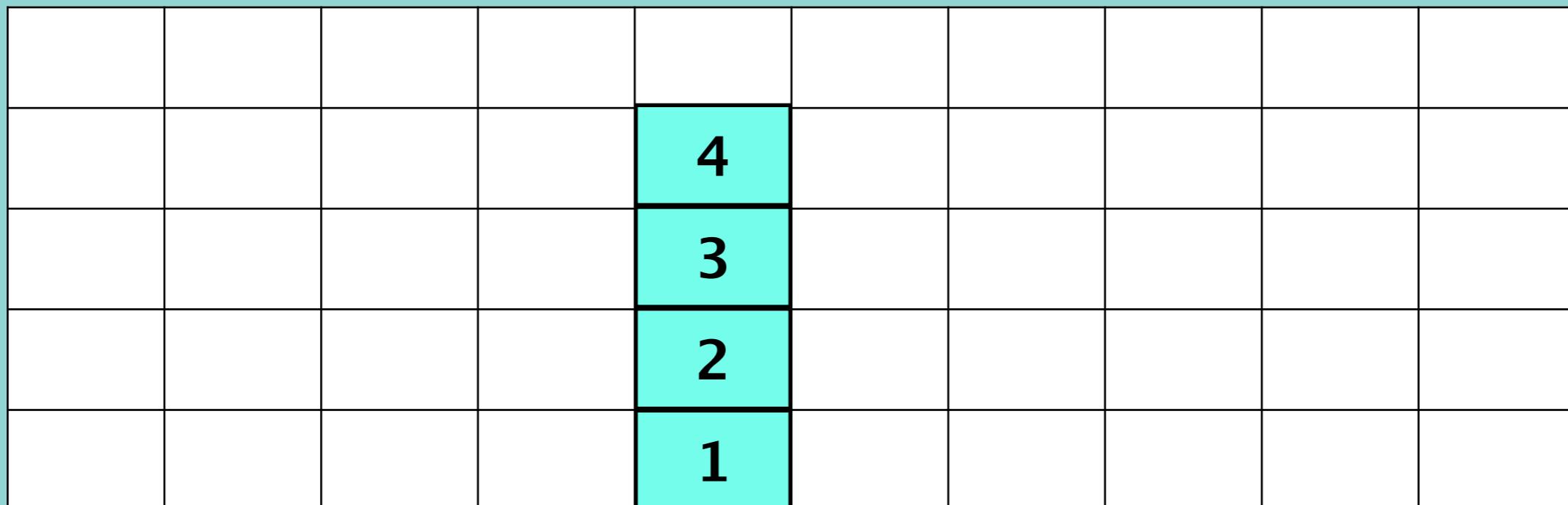
customStack.push(4)



Pop() method

customStack = [1,2,3]4]

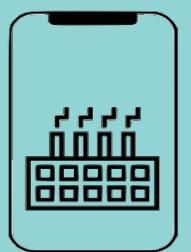
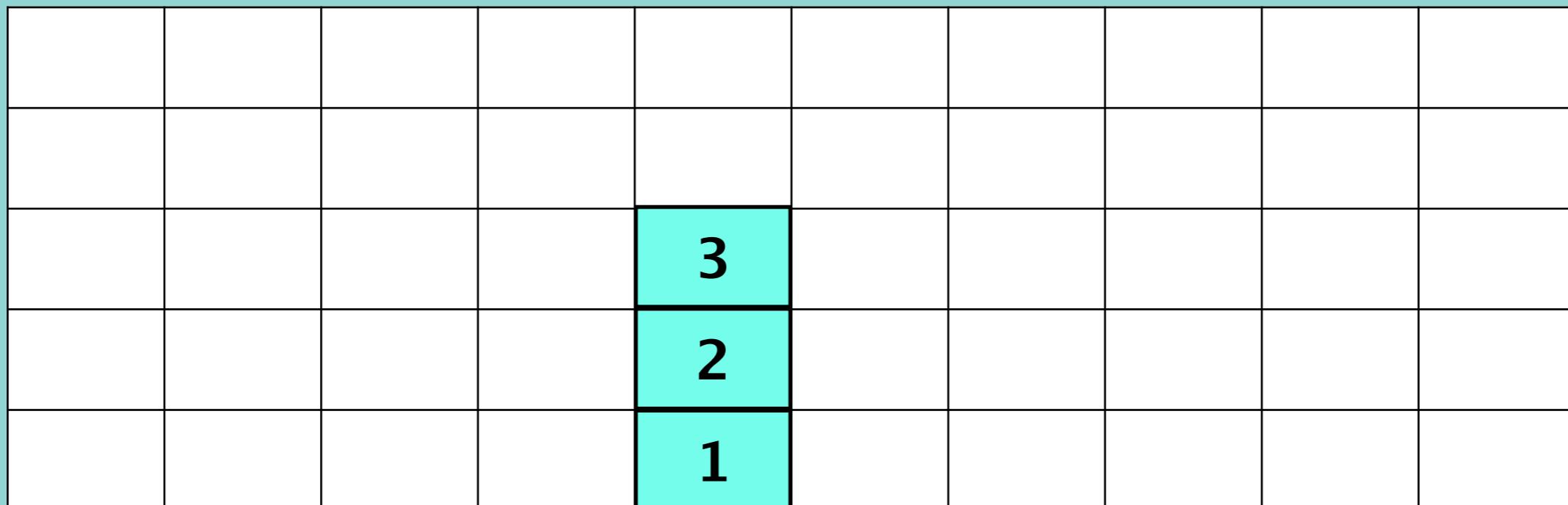
customStack.pop () —→ 4



Pop() method

customStack = [1,2]3]

customStack.pop () —→ 3

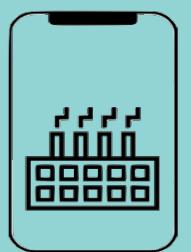


Pop() method

customStack = [1]2

customStack.pop () → 2

				2						
				1						

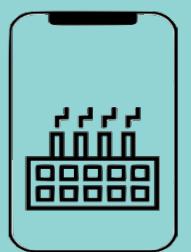


Pop() method

customStack = [1]

customStack.pop() —→ 1

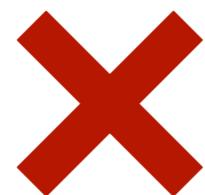
				1					



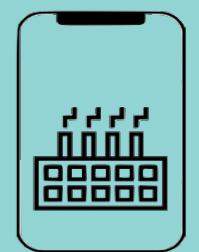
Pop() method

```
customStack = []
```

```
customStack.pop
```



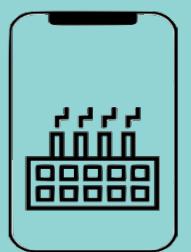
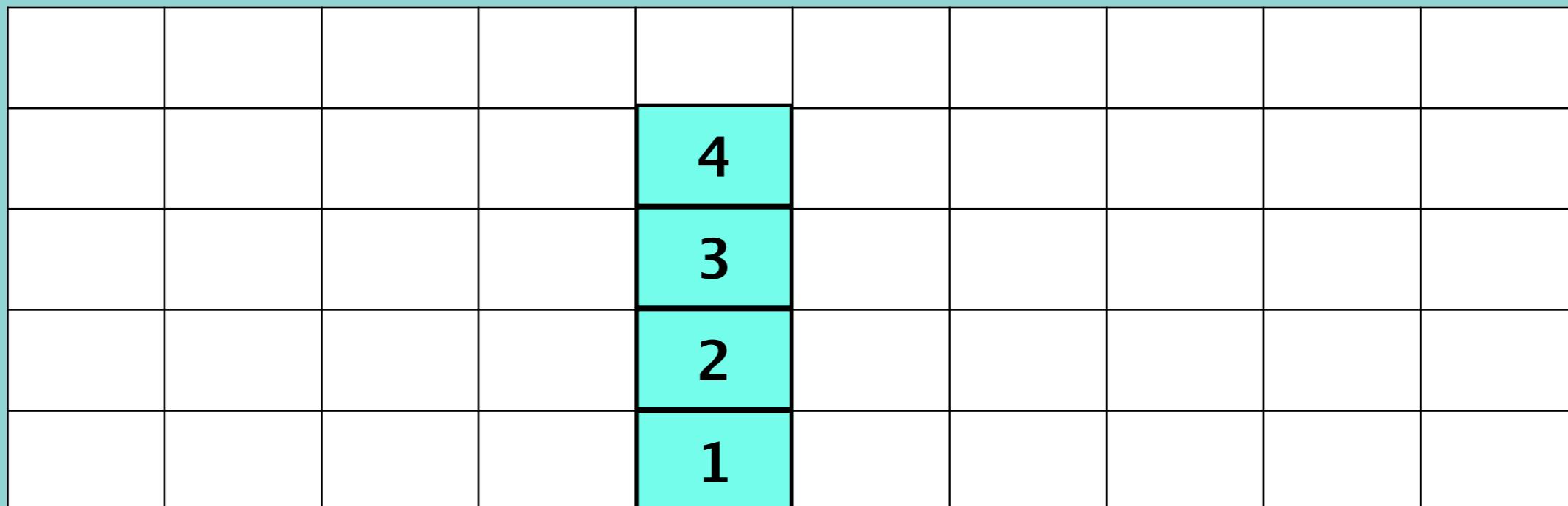
The stack is Empty



Peek() method

customStack = [1,2,3,4]

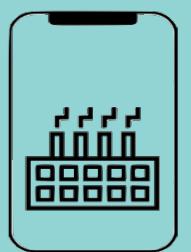
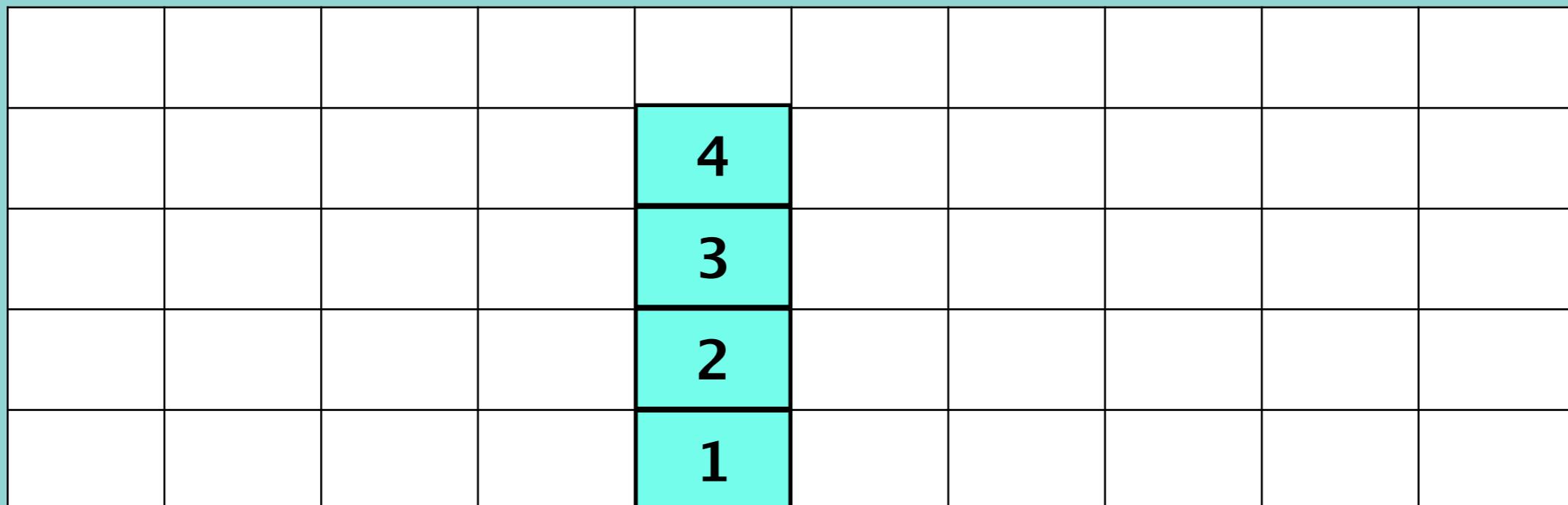
customStack.peek() → 4



isEmpty() method

customStack = [1,2,3,4]

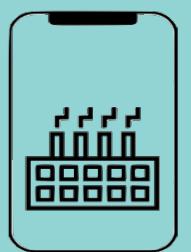
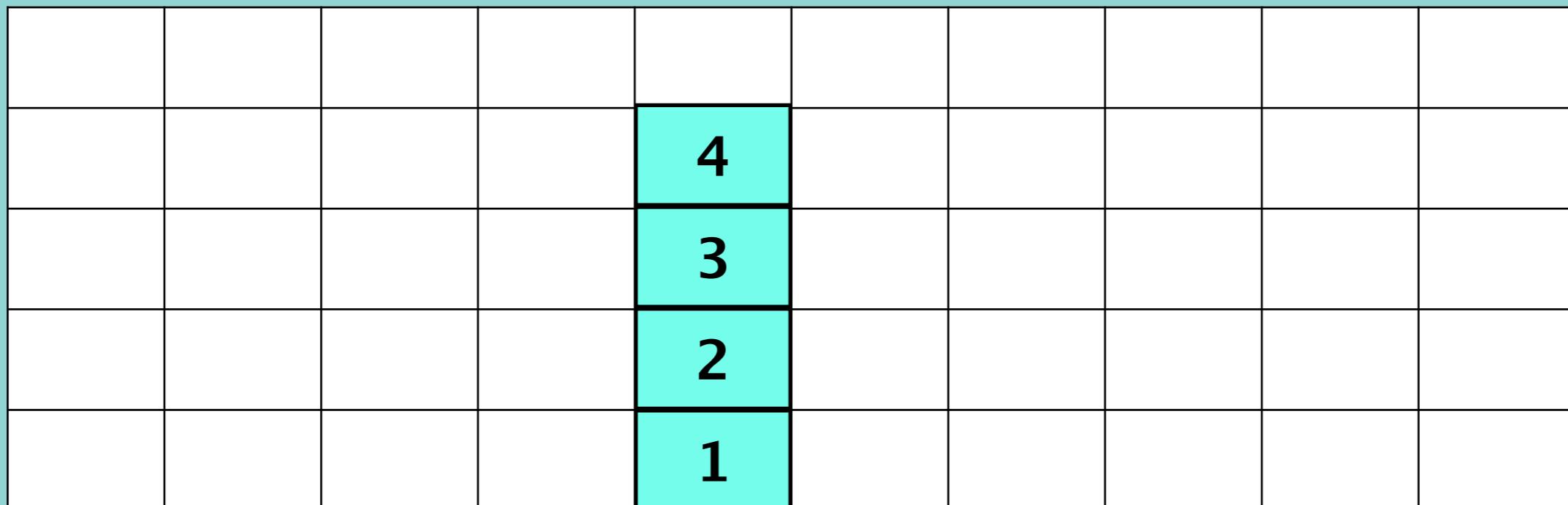
customStack.isEmpty() → **False**



isFull() method

customStack = [1,2,3,4]

customStack.isFull() → **False**

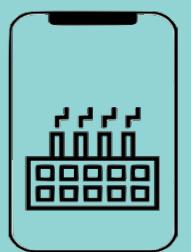


delete() method

customStack = [1,2,3,4]

customStack.delete()

					4						
					3						
					2						
					1						



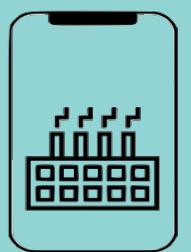
Stack creation

Stack using List

- Easy to implement
- Speed problem when it grows

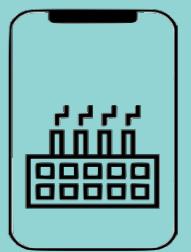
Stack using Linked List

- Fast Performance
- Implementation is not easy



Time and Space complexity of Stack operations with List

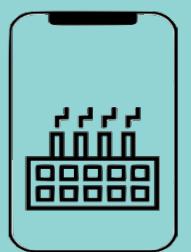
	Time complexity	Space complexity
Create Stack	O(1)	O(1)
Push	O(1) / O(n^2)	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Stack	O(1)	O(1)



Stack using Linked List

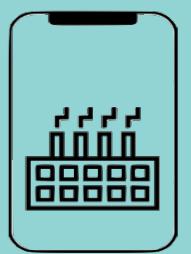
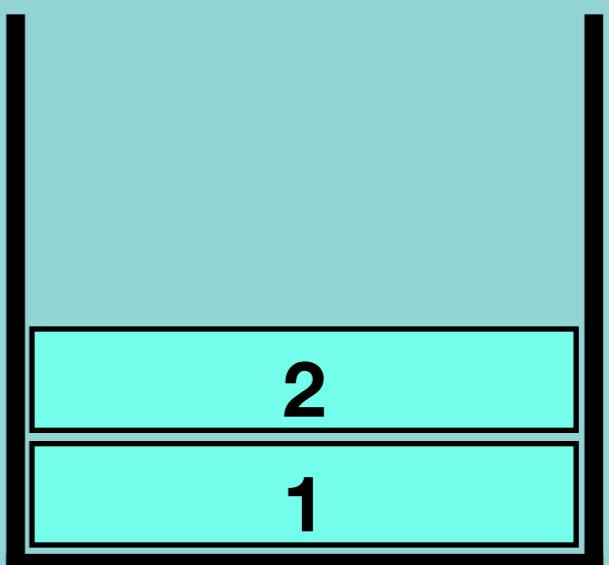
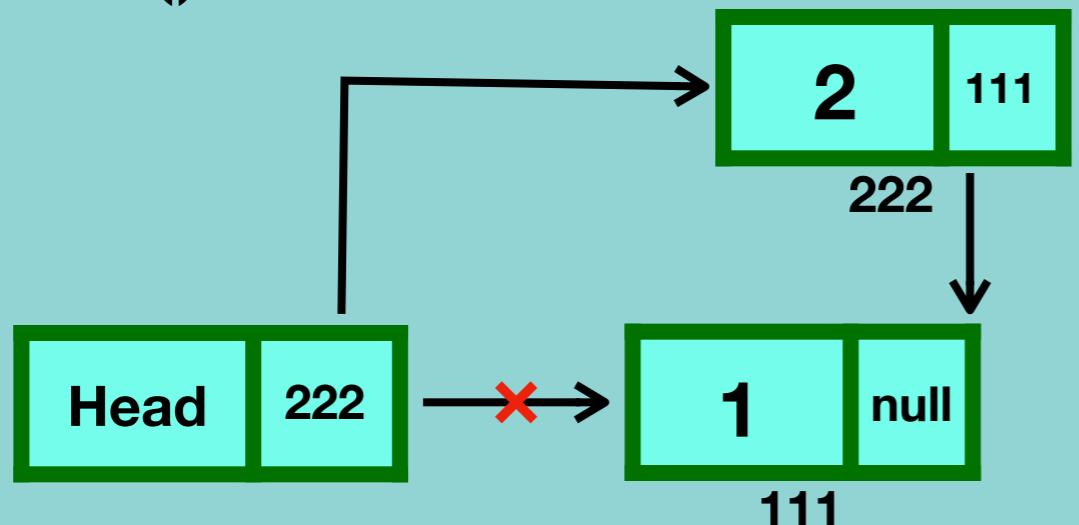
Create a Stack

Create an object of Linked List class



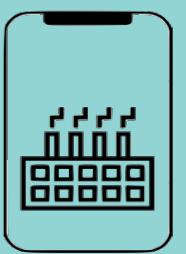
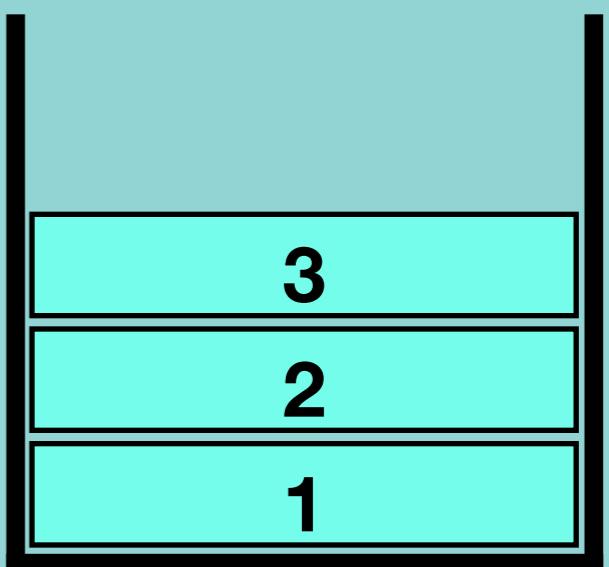
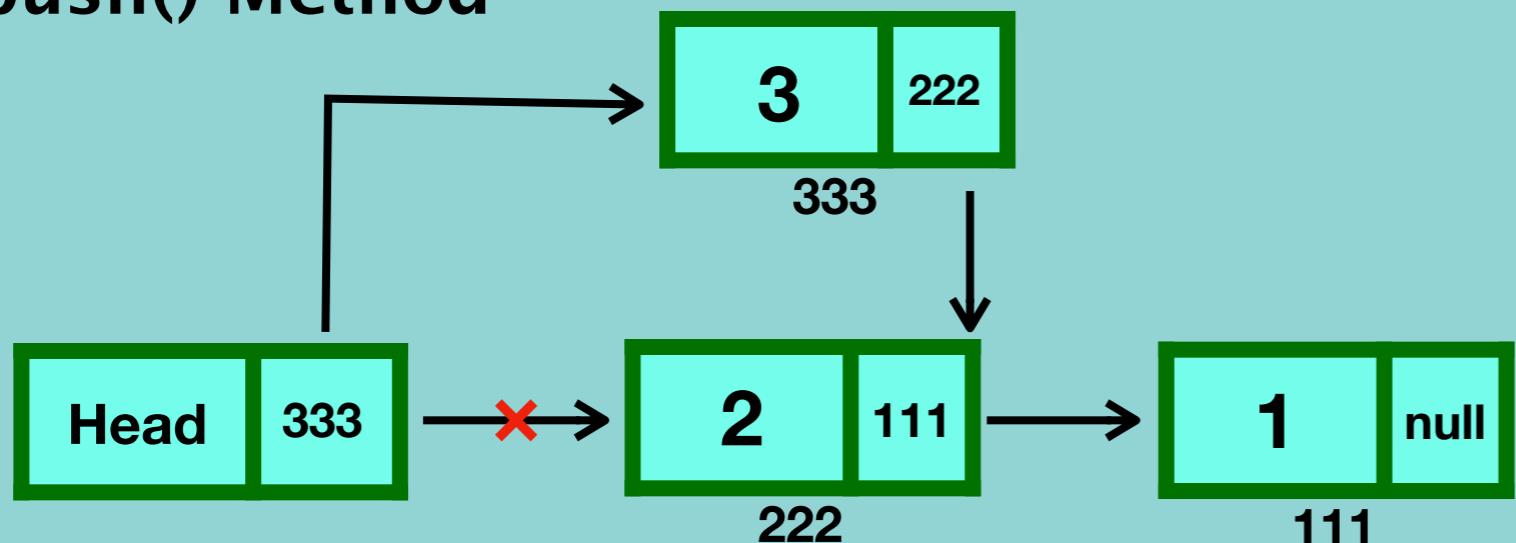
Stack using Linked List

push() Method



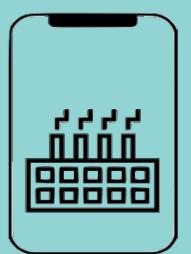
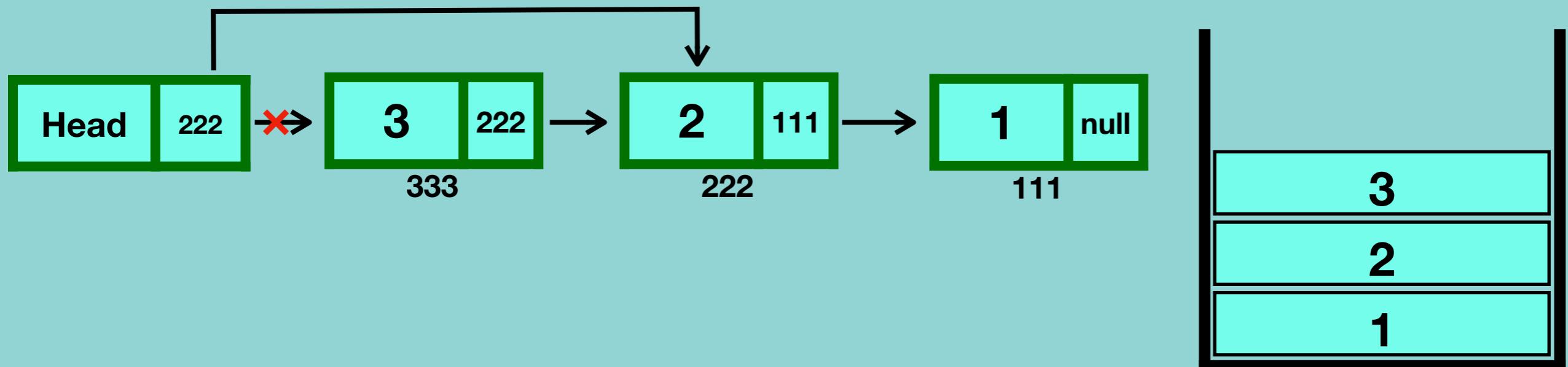
Stack using Linked List

push() Method



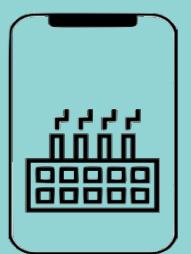
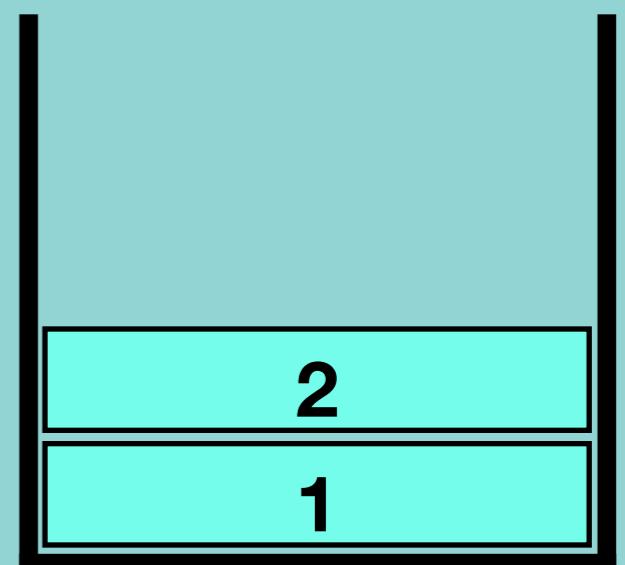
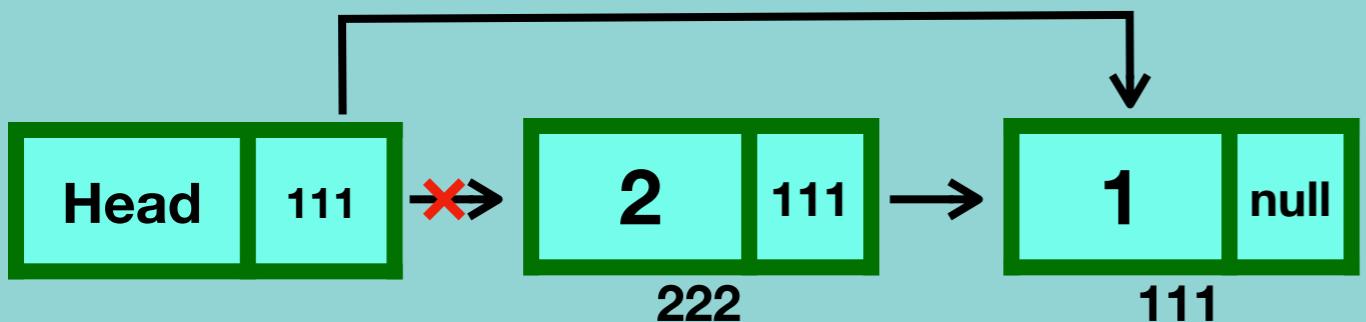
Stack using Linked List

pop() Method



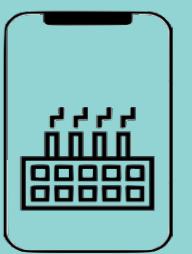
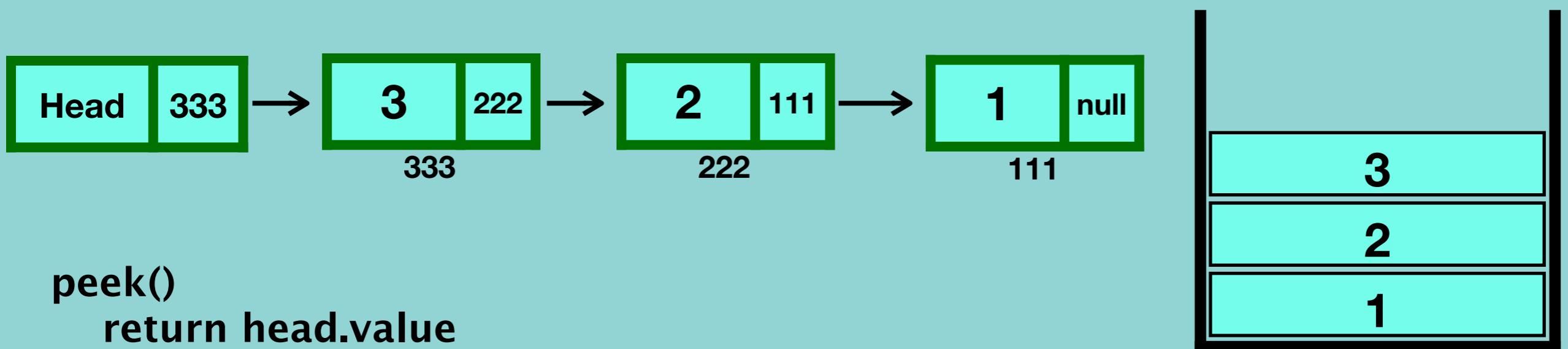
Stack using Linked List

pop() Method



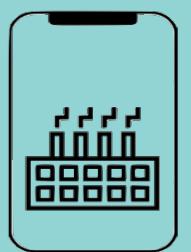
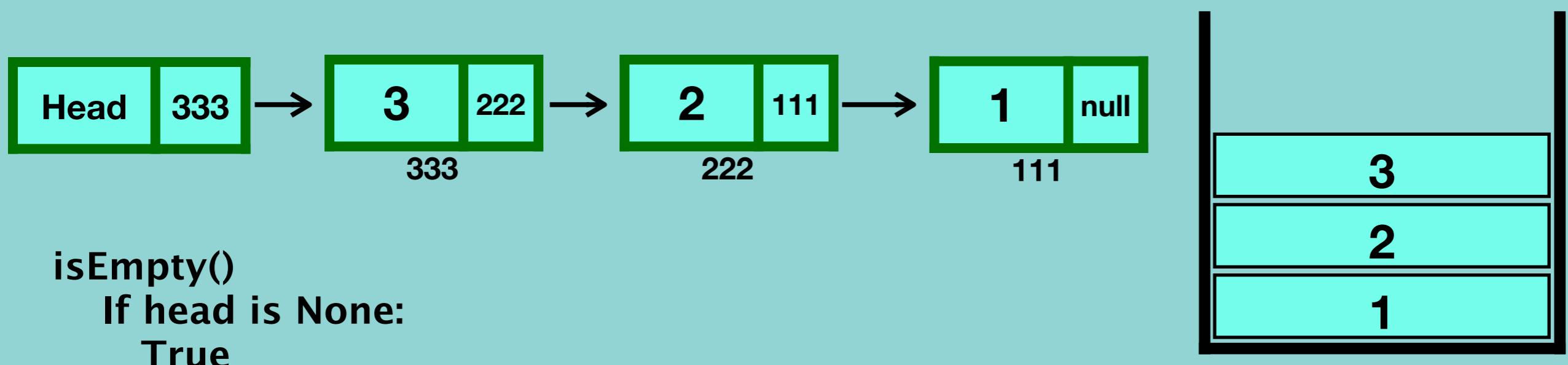
Stack using Linked List

peek() Method



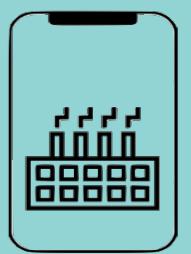
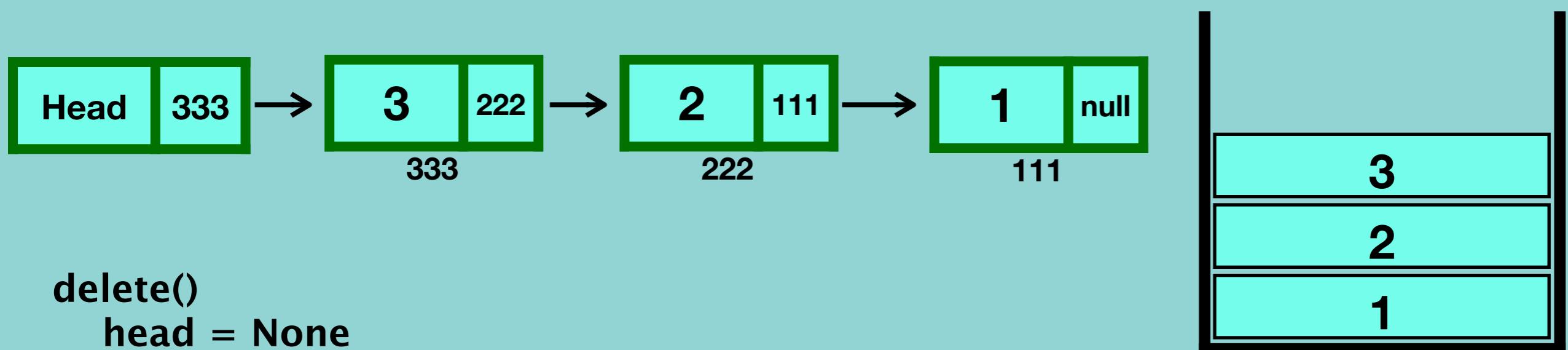
Stack using Linked List

isEmpty() Method



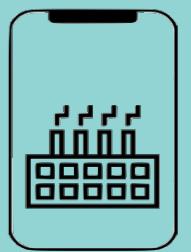
Stack using Linked List

delete() Method



Time and Space complexity of Stack operations with Linked List

	Time complexity	Space complexity
Create Stack	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Stack	O(1)	O(1)



When to use / avoid Stack

Use:

- LIFO functionality
- The chance of data corruption is minimum

Avoid:

- Random access is not possible

